Chapter 2: Operating-System Structures

Chapter 2: Operating-System Structures

- **Operating System Services**
- Dual Mode Operation and System Calls
- Types of System Calls
- System Programs
- Operating System Structure

Objectives

- Three views of an OS… each, respectively, focuses on
  - The services it provides [Ser]
  - The interface it makes available to users and programmers [Int]
  - Its components and interconnections [Com]

- 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 (Cont.) (Helpful to the user)

- 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**)
  - **Program execution** - The system must be able to load a program into memory (**loader**) 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

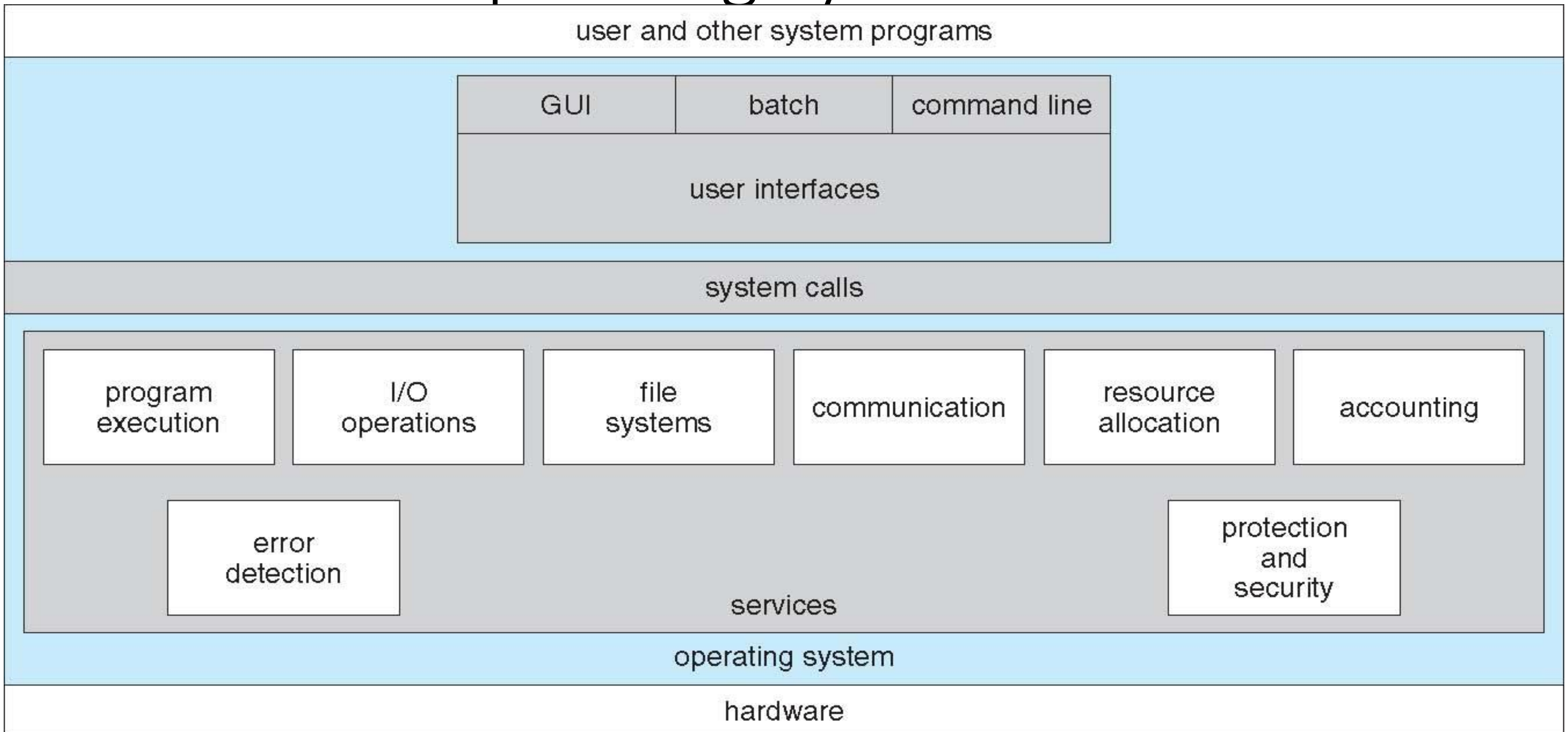
Operating System Services (Cont.) (Helpful to the user)

- 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***
  - **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.) (for efficient operation of the OS)

- 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 (e.g., the number of pages printed, number of hours internet used etc.)
  - **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
    - It 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



	Services
Helpful to User	User Interface (CLI etc.), Program Execution, I/O Operations, File System, Communication, Error Detection
Efficient Operation of OS	Resource Allocation, Accounting, Protection and Security

Chapter 2: Operating-System Structures

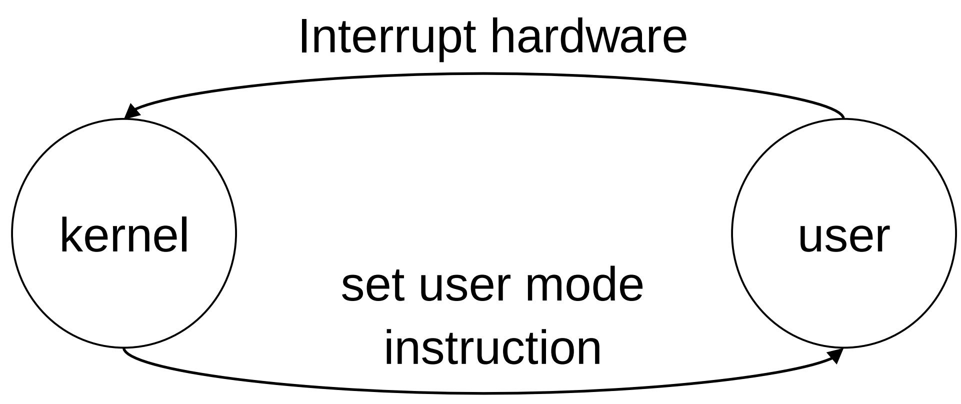
- Operating System Services
- **Dual Mode Operation and System Calls**
- Types of System Calls
- System Programs
- Operating System Structure

Dual-Mode Operation (1)

- Provide hardware support to differentiate between at least two modes of operations:
  - **User mode:** execution done on behalf of a user.
  - **kernel mode:** execution done on behalf of OS.
- Must ensure that a user program could never gain control of the computer in kernel mode.
- Privileged Instructions can be executed only in kernel mode.
- Solution: Mode bit (in Status Register).

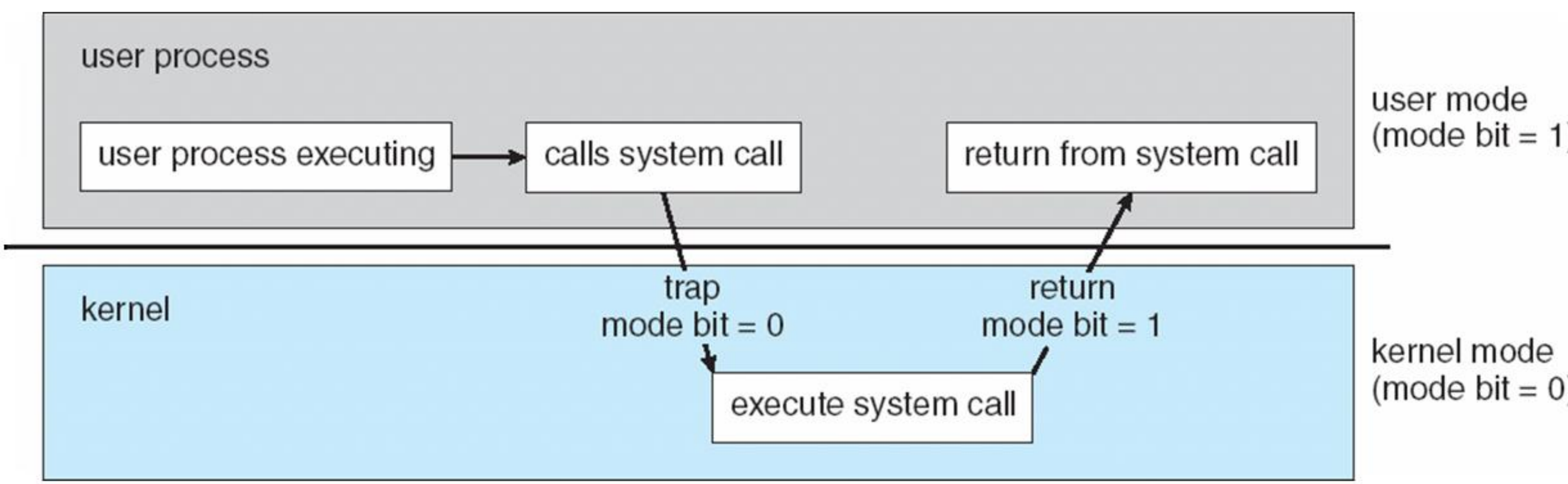
Dual-Mode Operation (2)

- ***Mode bit*** was added to computer hardware (in Status Register) to indicate the current mode: kernel/system (0) or user (1).
- When any type of interrupt occurs, interrupt hardware switches to kernel mode, at the correct service routine in the kernel address space – safe method!

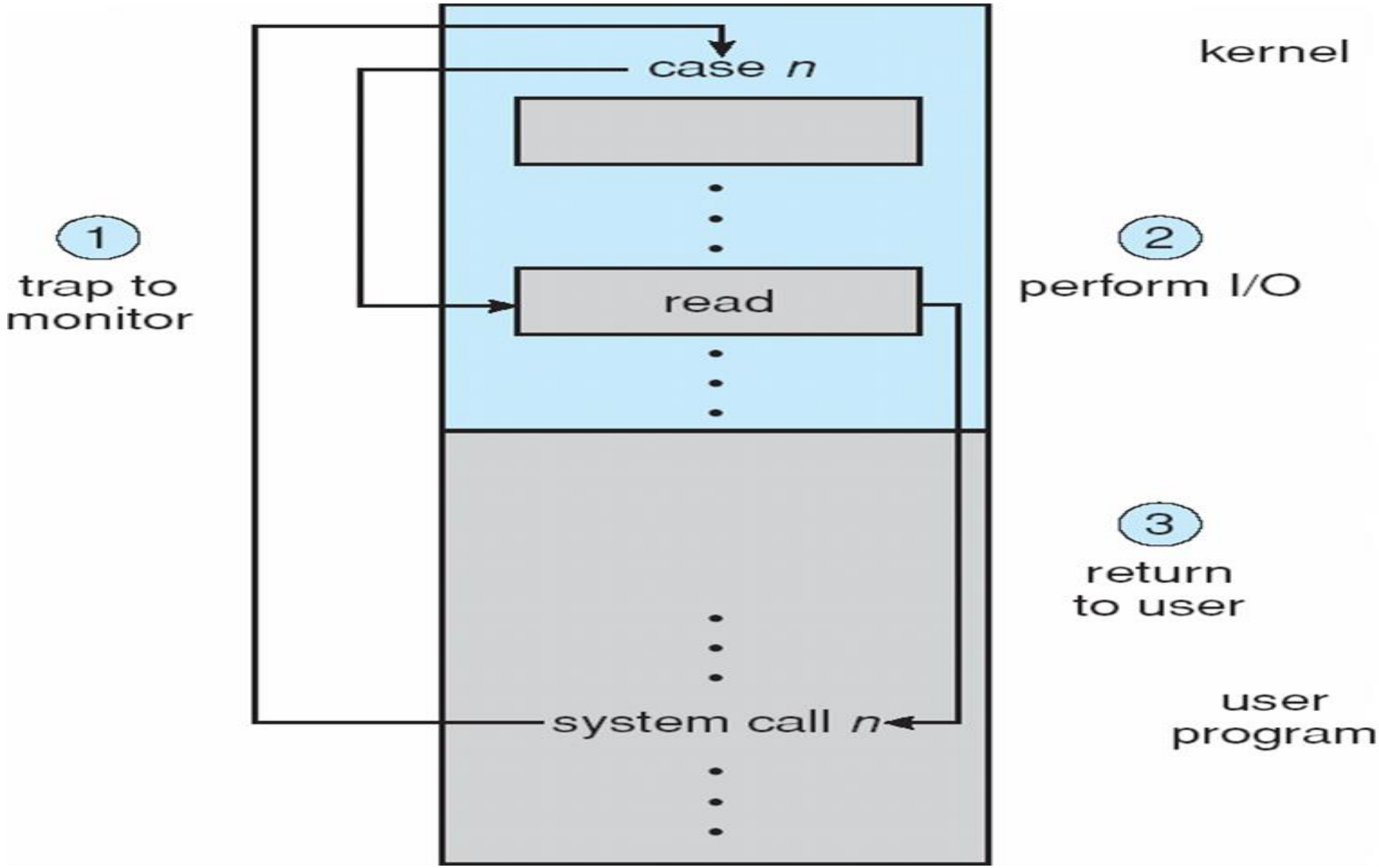


Dual-Mode Operation (cont'd)

- all I/O instructions are privileged instructions
  - instead of performing I/O operation directly, user program must make a **system call**
  - OS, executing in **kernel mode**, checks validity of request and does the I/O
  - input is returned to the program by the OS
  - In most existing systems, switching from user mode to kernel mode has an associated *high cost* in performance.



System Call to Perform I/O



CPU Protection (Timer)

- **Timer** to prevent infinite loop / process hogging resources
  - Timer is set to interrupt the computer after some time period
  - Keep a counter that is decremented by the physical clock.
  - Operating system ***set the counter*** (privileged instruction)
  - When counter becomes zero, it generates an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time
- ***MS-DOS*** originally had no mode bit and no dual mode
  - A user program could wipe out OS
- ***MS Windows 7 and Linux/Unix*** have dual mode feature and provide greater protection.

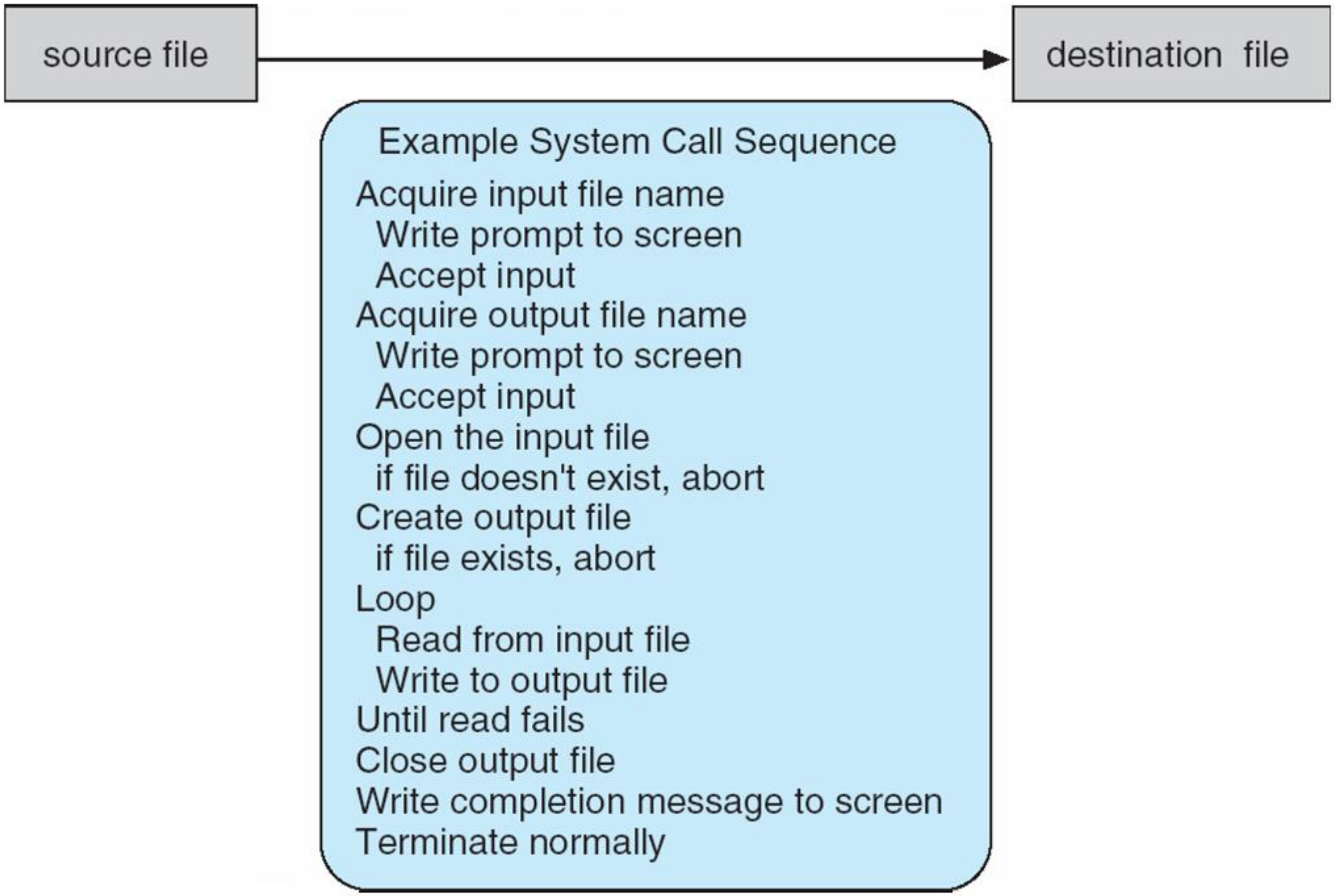
System Calls [Int]

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
  - With hardware-level tasks written in assembly language
- Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct system call use
  - API specifies set of functions available to the programmer
    - Example :functions ***ReadFile()*** or ***CreateProcess()*** in WIN32 API
    - Functions invoke the actual system calls on behalf of the programmer
      - Function ***CreateProcess()*** invokes system call ***NTCreateProcess()***

- 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)

Example of System Calls [Int]

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



Example of Standard API [Int]

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.

Example of System Calls

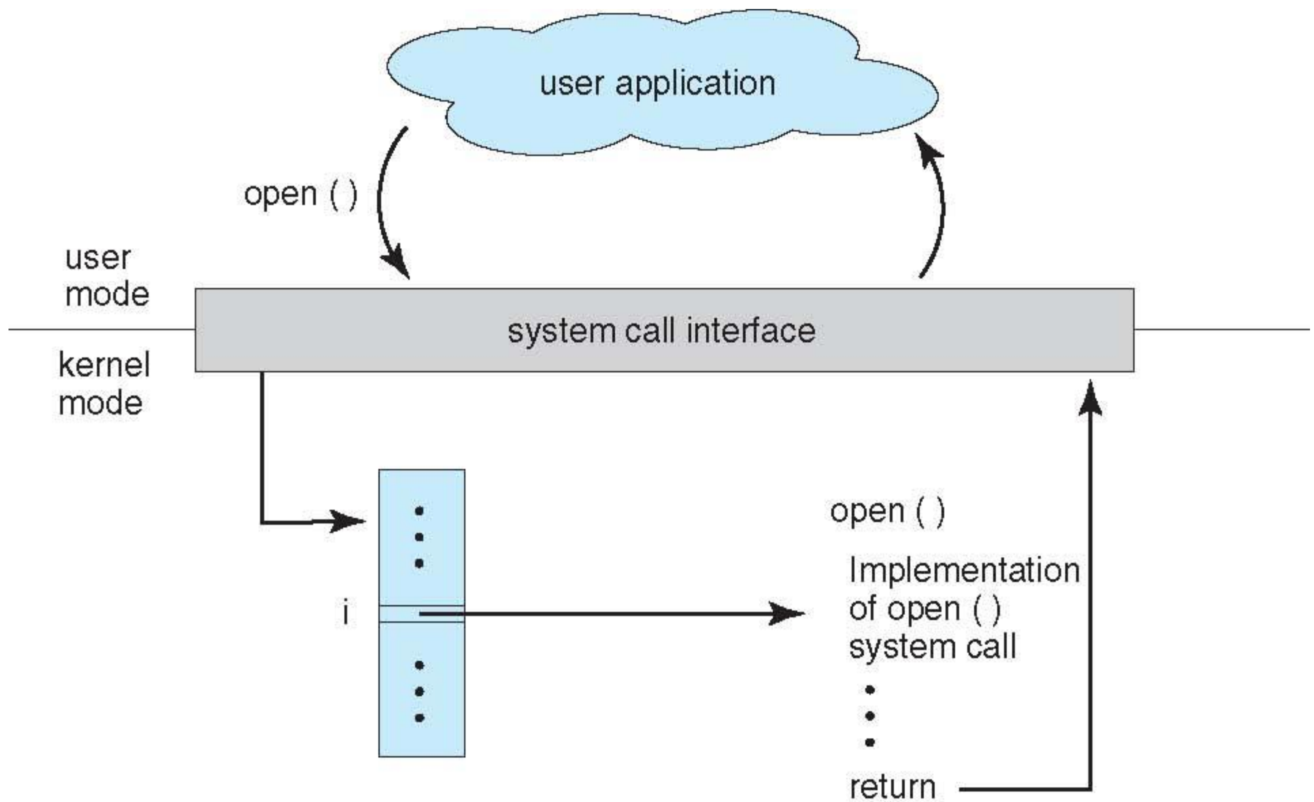
- writing a simple program to read data from one file and copy them to another file.
- The first input that the program will need is the names of the two files:
  - the ***input file*** and the ***output file***. These names can be specified in many ways, depending on the operating-system design.
  - For example, one approach is for the program to ask the user for the names.
- In an *interactive system*, this approach will require a sequence of system calls,
  1. first to ***write*** a prompting message on the screen and then to *read* from the keyboard the characters that define the two files.
  2. Once the two file names have been obtained, the program must ***open*** the input file and the output file. Each of these operations requires another system call.

Example of System Calls

- Possible ***error conditions*** for each operation can require additional system calls.
- **For example:**
  - When the program tries to ***open*** the input file, it may find that there is **no file** of that name or that the file is **protected against access**
  - In these cases, the program should ***print*** a message on the console (another sequence of system calls) and then **terminate abnormally** (another system call)
  - If the input file exists, then we must ***create*** a new output file.
  - We may find that there is **already an output file** with the same name. This situation may cause the program to **abort** (a system call), or we may delete the existing file (another system call) and create a new one (yet another system call).
  - When both files are set up, we enter a loop that *reads* from the input file (a system call) and ***writes*** to the output file (another system call). Each ***read*** and ***write*** must return status information regarding various possible error conditions.

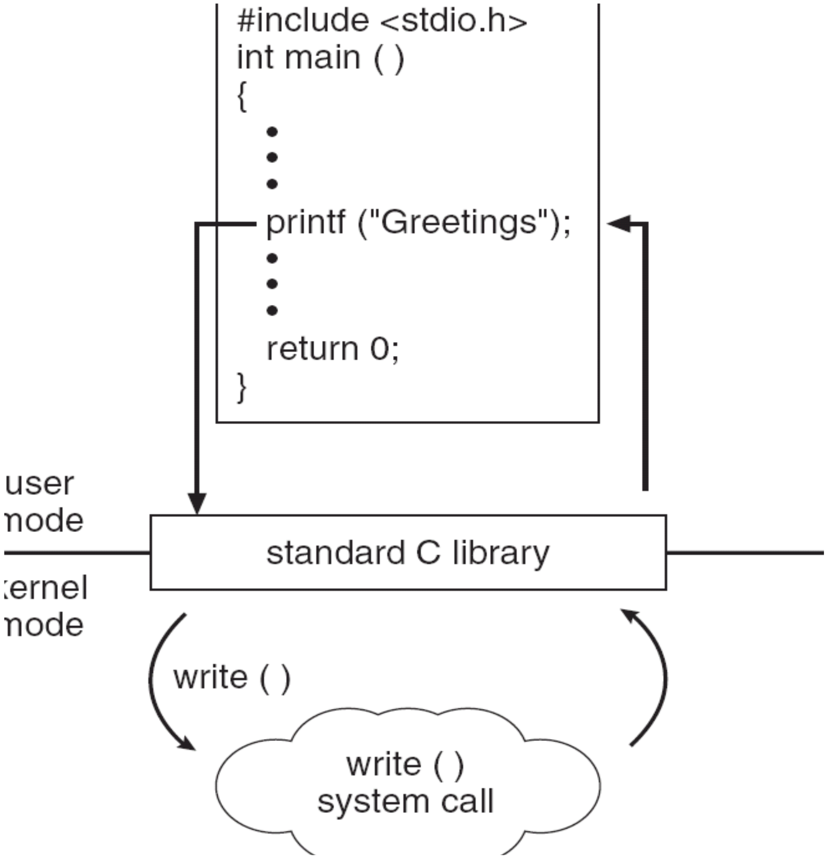
System Call Implementation [API-system call-OS Relationship]

- Typically, a number is 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



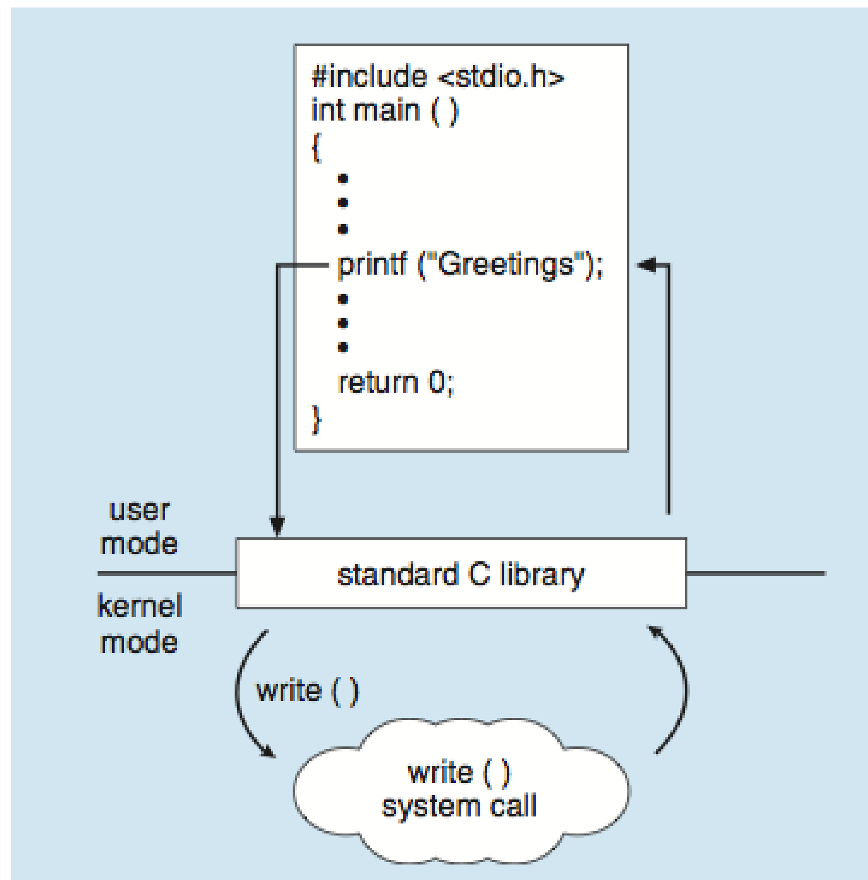
System Call Implementation

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



Standard C Library Example

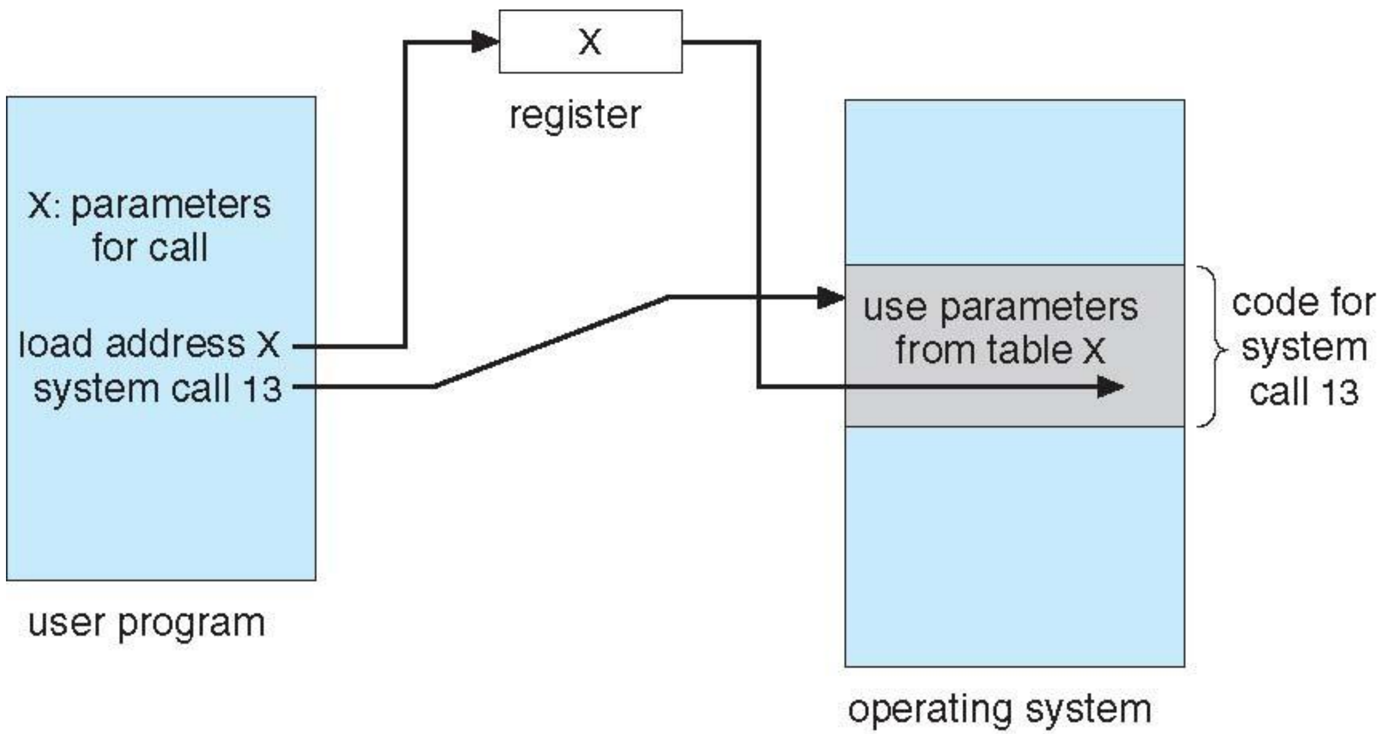
- C program invoking **printf()** library call, which calls **write()** system call
  - The standard C library provides portion of system-call interface for many versions of UNIX and Linux



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 system call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers [upto six parameters]
    - 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 can also be placed, or pushed, onto the ***stack*** by the program and popped off the stack by the operating system.
- Block and stack methods are preferred… do not limit the number or length of parameters being passed

Parameter Passing via Table



Chapter 2: Operating-System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- **Types of System Calls**
- System Programs
- Operating System Design and Implementation
- Operating System Structure

Types of System Calls [Int]

- **Process control**
  - create process (***fork()***), terminate process (***exit***)
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - **Locks** for managing access to shared data between processes

Types of System Calls [Int]

- **File management**
  - create file, delete file
  - open, close file
  - read, write
  - get and set file attributes

- **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

- **Device management**
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

- **Communications**
  - create, delete communication connection
  - send, receive messages of **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

Chapter 2: Operating-System Structures

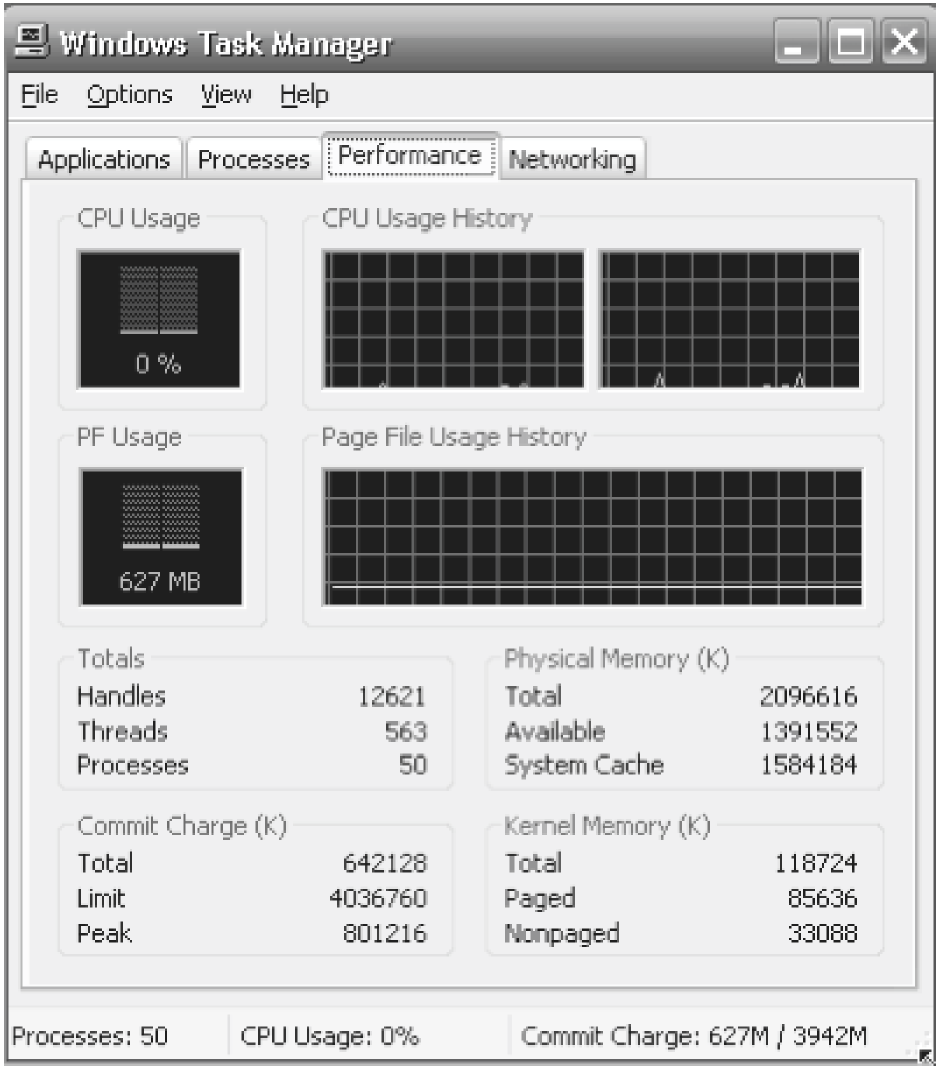
- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- **Operating system debugging and Performance Tuning**
- Operating System Structure

Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- OS generate **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
  - Sometimes using ***trace listings*** of activities, recorded for analysis [details soon]
  - **Profiling** is periodic sampling of instruction pointer to look for statistical trends

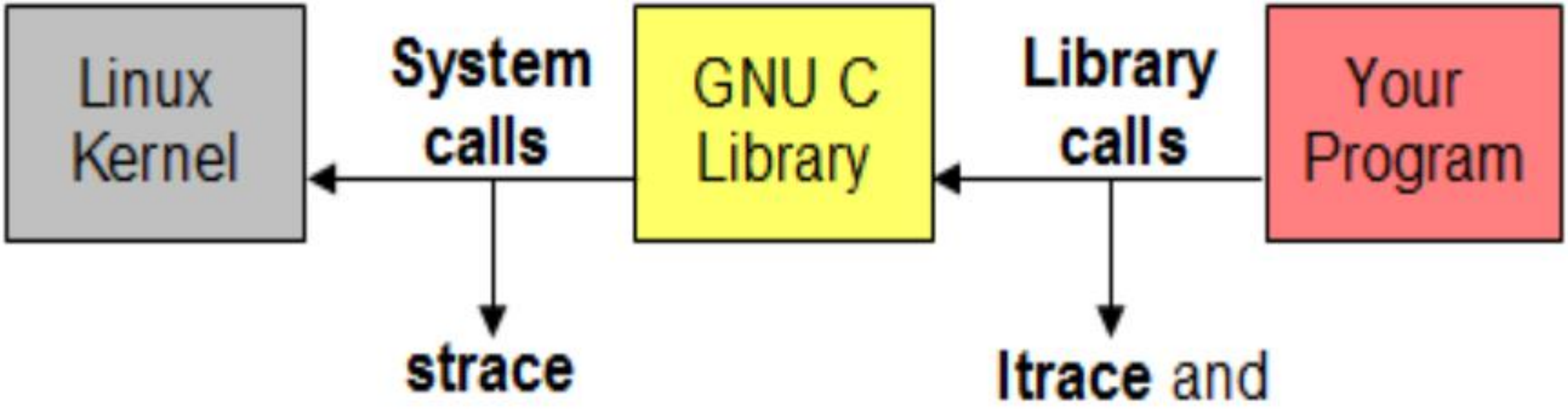
Performance Tuning

- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- For example, Windows Task Manager.



strace and ltrace

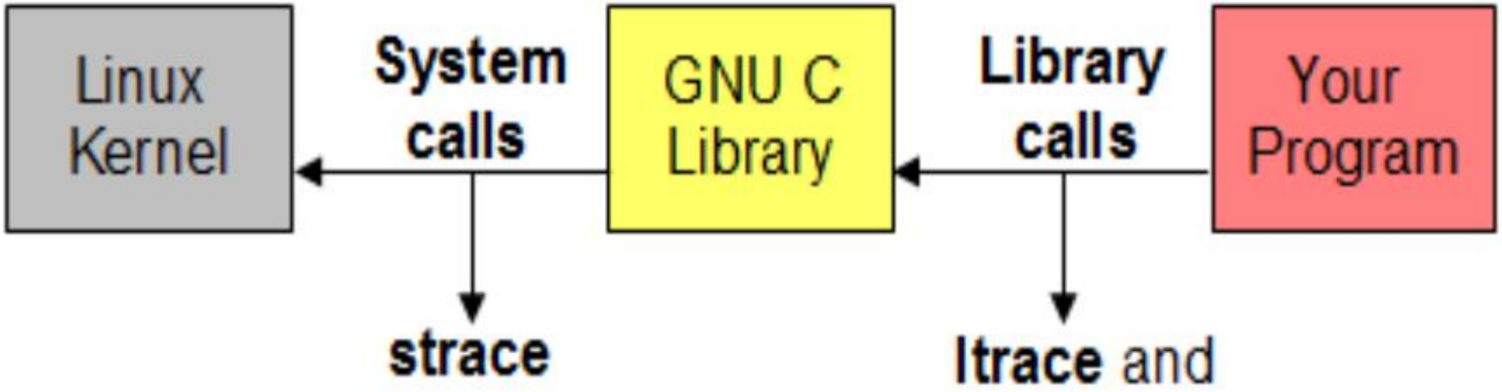
- Both strace and ltrace are powerful command-line tools for debugging and troubleshooting programs on Linux.
- strace and ltrace capture and records all system calls and library calls made by a process as well as the signals received, respectively.



strace

- strace monitors the system calls and signals of a specific program.
- It is helpful when you do not have the source code and would like to debug the execution of a program.
- strace provides you the execution sequence of a binary from start to end.
- For example; (ls command is used to list the items of the current directory)

  strace ls



strace

- Each line in the trace contains the *system call name*, followed by its *arguments in parentheses* and its *return value*.
- Relative timestamp was produced in the beginning of every line

```
root@server1:~# strace -r ls
     0.000000 execve("/usr/bin/ls", ["ls"], 0x7ffd9cc24268 /* 19 vars */) = 0
     0.001027 brk(NULL)                         = 0x56494f38d000
     0.000521 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directo
     0.000550 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
     0.000595 fstat(3, {st_mode=S_IFREG|0644, st_size=33427, ...}) = 0
     0.000482 mmap(NULL, 33427, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f146ba98000
     0.000455 close(3)                          = 0
     0.000446 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CL
     0.000445 read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@k\0\0\0\0\0\0".
     0.000605 fstat(3, {st_mode=S_IFREG|0644, st_size=155296, ...}) = 0
     0.000501 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0
     0.000441 mmap(NULL, 2259632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3,
     0.000425 mprotect(0x7f146b893000, 2093056, PROT_NONE) = 0
     0.000486 mmap(0x7f146ba92000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
     0.000473 mmap(0x7f146ba94000, 6832, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
     0.000545 close(3)                          = 0
     0.000464 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC)
     0.000504 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260A\2\0\0\0\0\
     0.000515 fstat(3, {st_mode=S_IFREG|0755, st_size=1824496, ...}) = 0
     0.000489 mmap(NULL, 1837056, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f1
     0.000453 mprotect(0x7f146b6cf000, 1658880, PROT_NONE) = 0
```

Chapter 2: Operating-System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System debugging and performance tunning
- **Operating System Design and Implementation**
- Operating System Structure

Operating System Design and Implementation

- Design and Implementation of OS not ***"solvable"***, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- **User** goals and **System** goals
  - **User goals** – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - **System goals** – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Operating System Design and Implementation (Cont.)

- Important principle to separate

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

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

- Mechanisms determine ***how*** to do something, policies decide ***what*** will be done
- 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 an OS is highly creative task of **software engineering**

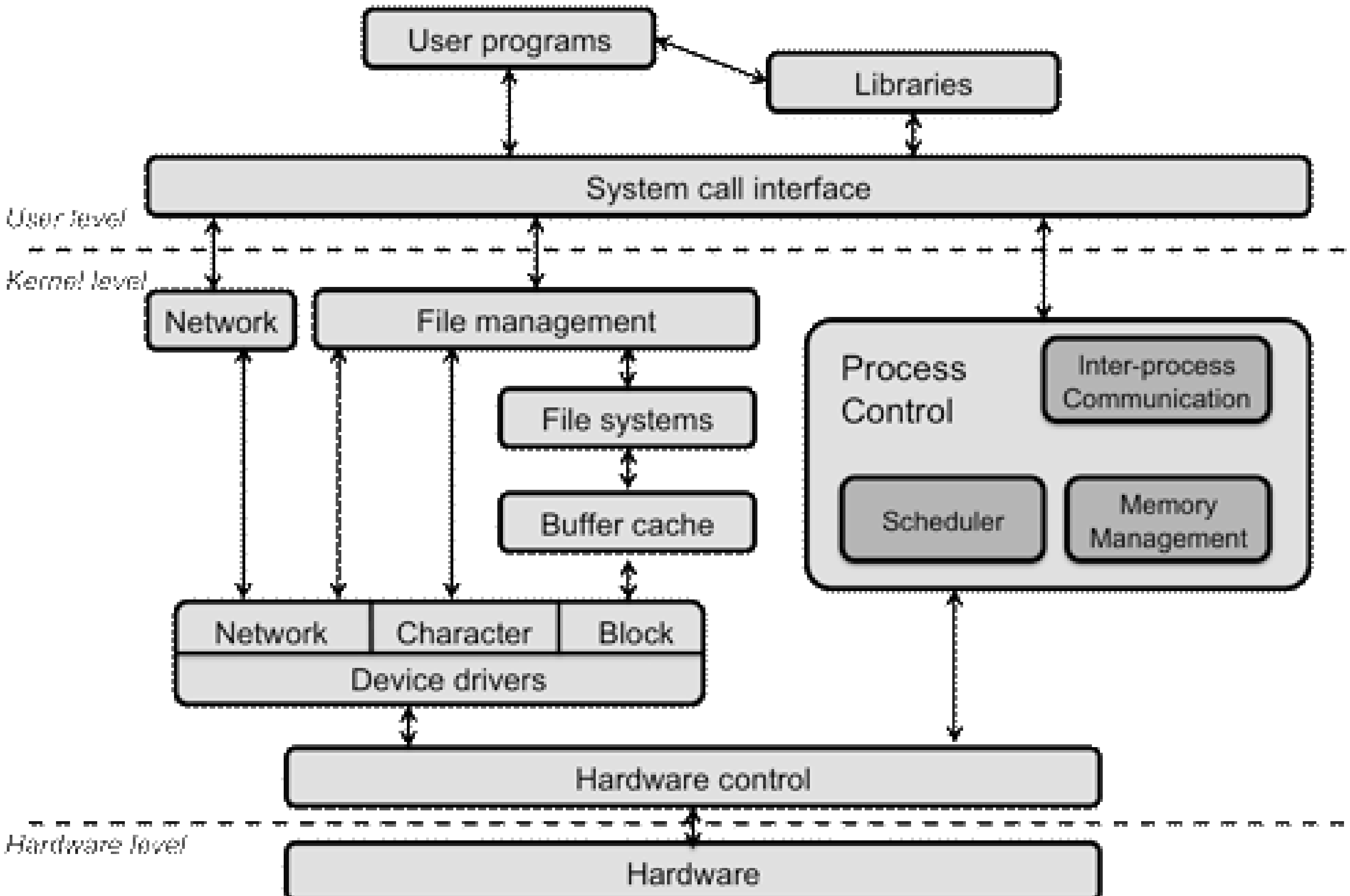
Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - 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 slower

Chapter 2: Operating-System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- **Operating System Structure**

Structure of a Traditional an OS

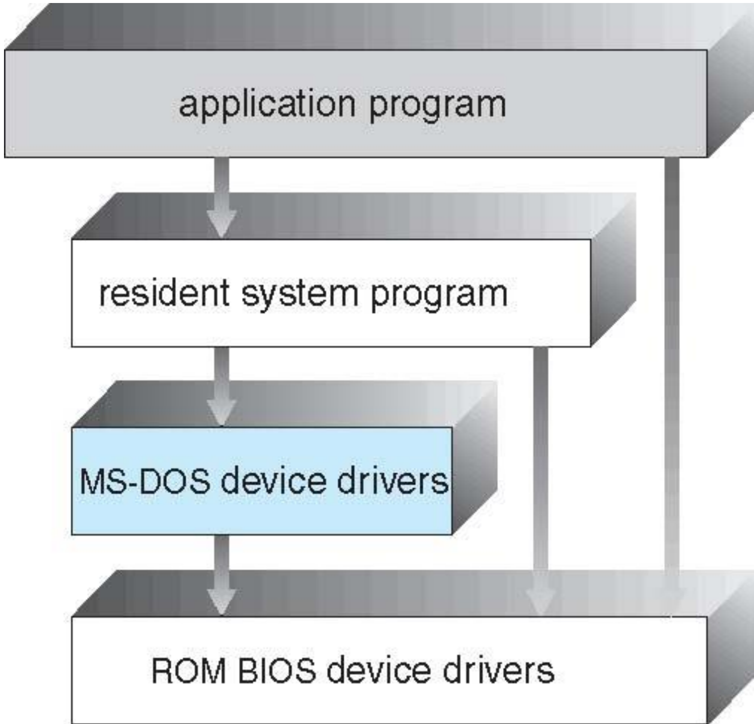


Operating System Structure [com]

- General-purpose OS is very large program
    - Hence, OS must be engineered intelligently for easy use and modification
  - OS design: partition into modules and define interconnections
- Various ways to structure ones [more details soon next]
  - <u>Simple structure – MS-DOS</u>: small kernel, not well separated modules, no protection, limited by Intel 8088 hardware
  - <u>More complex – original UNIX:</u> Monolithic, large kernel, two-layered UNIX (separates kernel and system programs), initially limited by hardware
  - <u>Layered – an abstraction:</u> Modular OS, freedom to change/add modules
  - <u>Microkernel –Mach:</u> Modularized the expanded but large UNIX, keeps only essential component as system-level or user-level programs, smaller kernel, and easy to extend

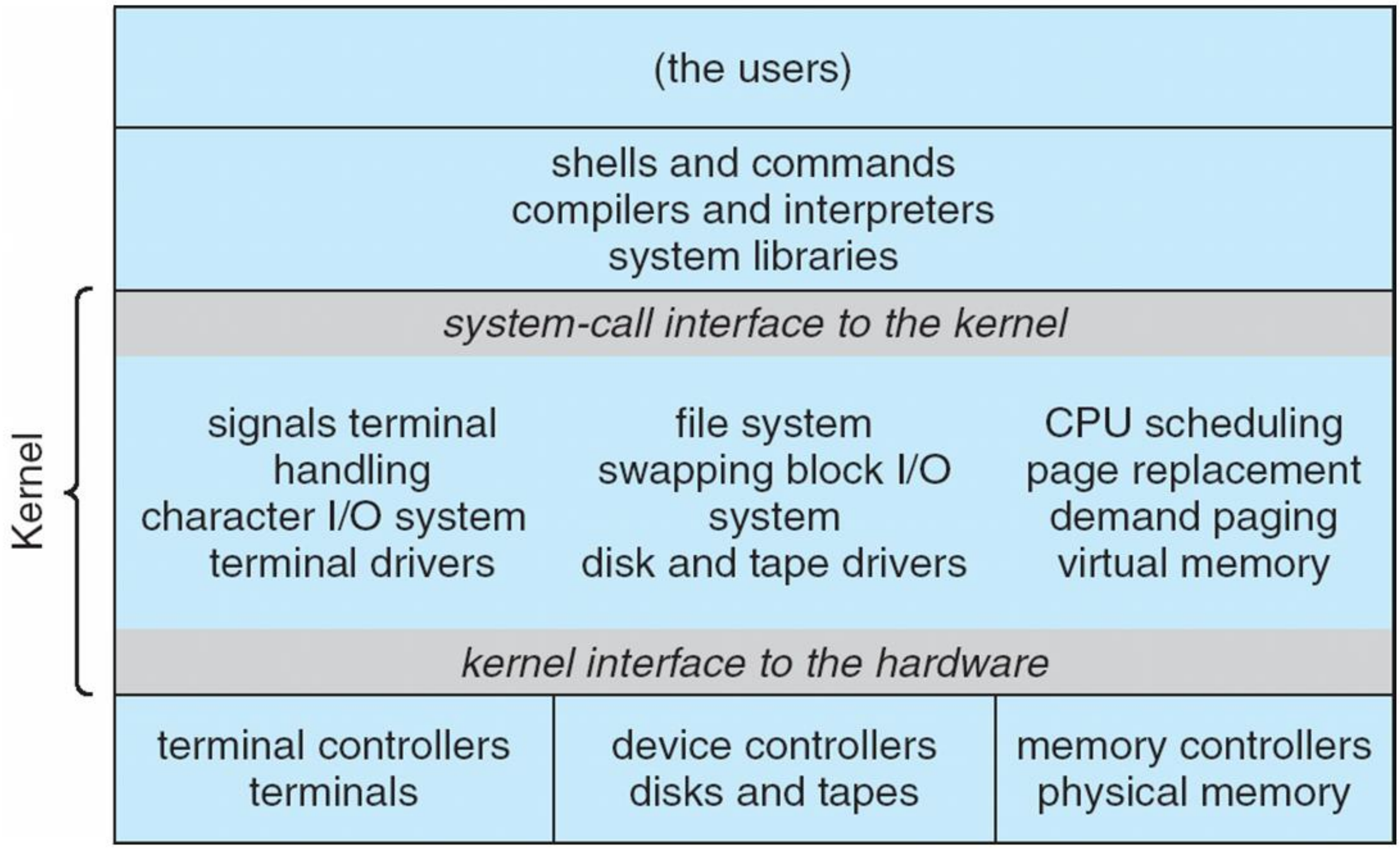
1. Simple Structure -- MS-DOS (1981)

- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - No dual mode, and No hardware protection
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated
  - E.g., Applications can write directly to the display and disk drives
  - Intel 8088 processor had no dual mode
    - Vulnerable
    - No protection



Traditional UNIX System Structure [com]

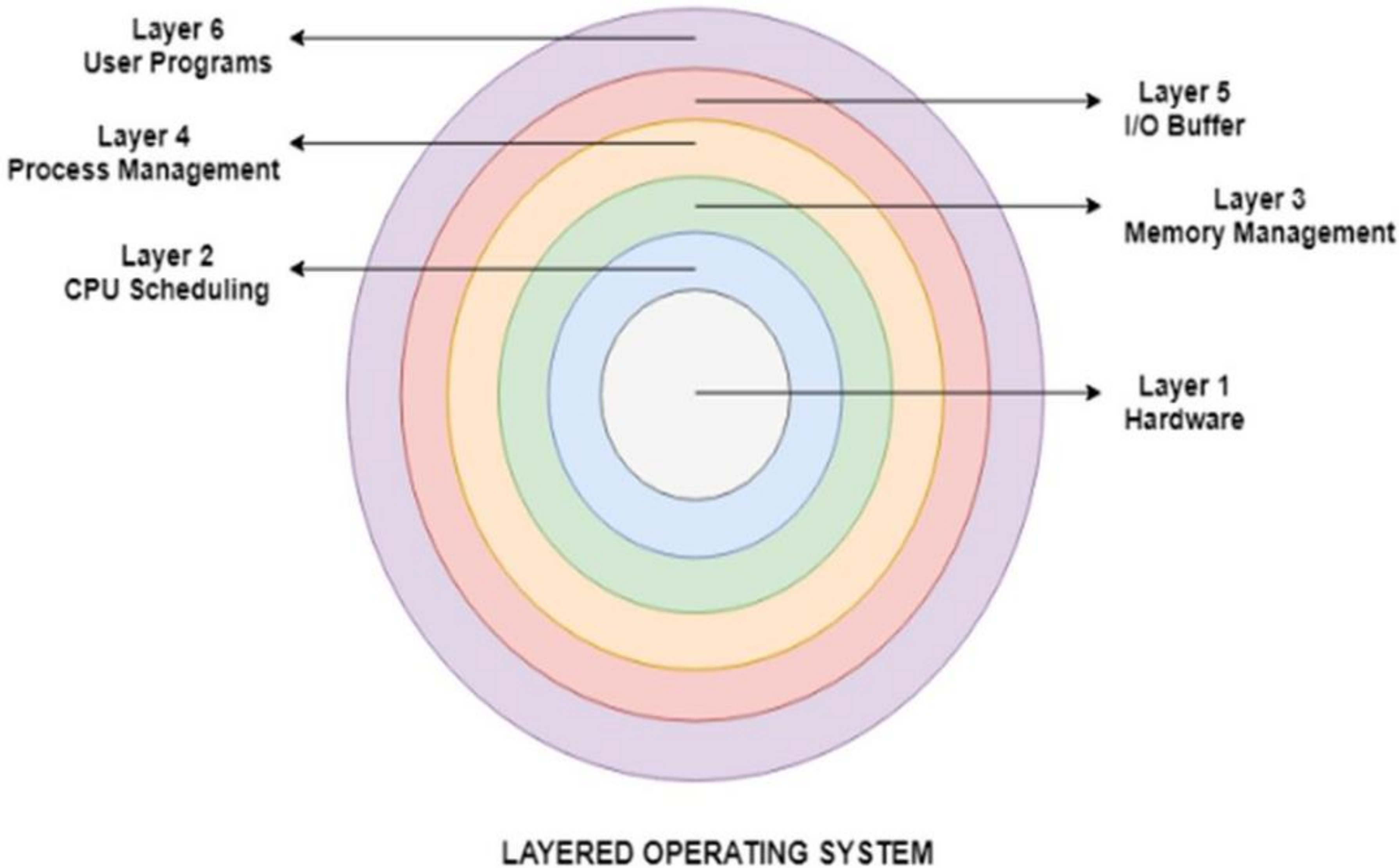
Beyond simple but not fully layered



2. Non Simple Structure -- UNIX [com]

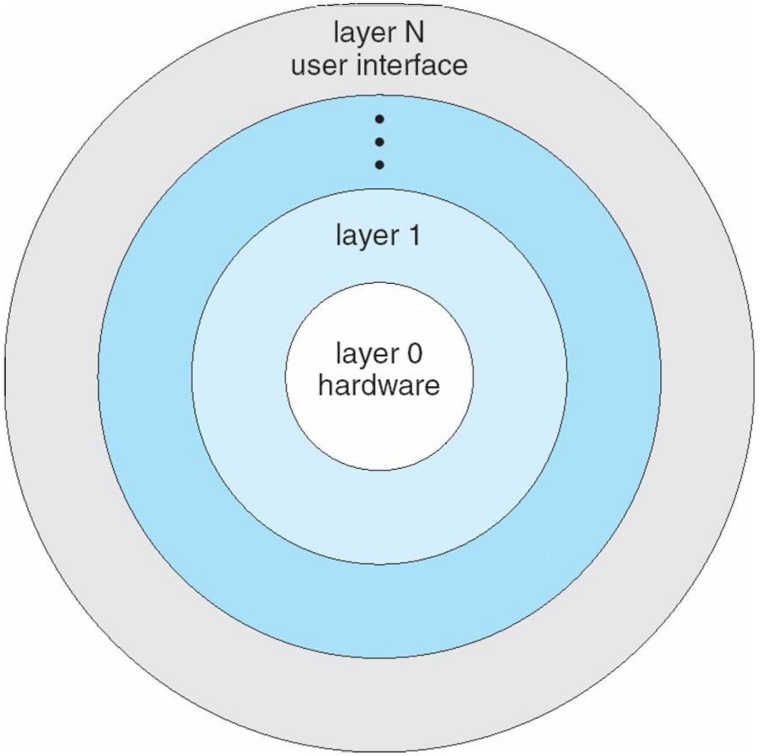
- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Had **monolithic** structure, difficult to implement
    - Consists of everything below the ***system-call interface*** 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
      - Very Large Kernel
- In a monolithic kernel calls between components are simple function calls, as all programmers are familiar with.
      - minimal **overhead** between function calls
      - But difficult to add new functionality

3. Layered Approach [Com]



3. Layered Approach [Com]

- 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
  - Advantage: each layer is
    - Abstraction: data + operations on data
    - Simple to construct
    - Easy to debug and verify
  - Problems:
    - defining the various layers,
    - less efficient than non-layered OS, increased over

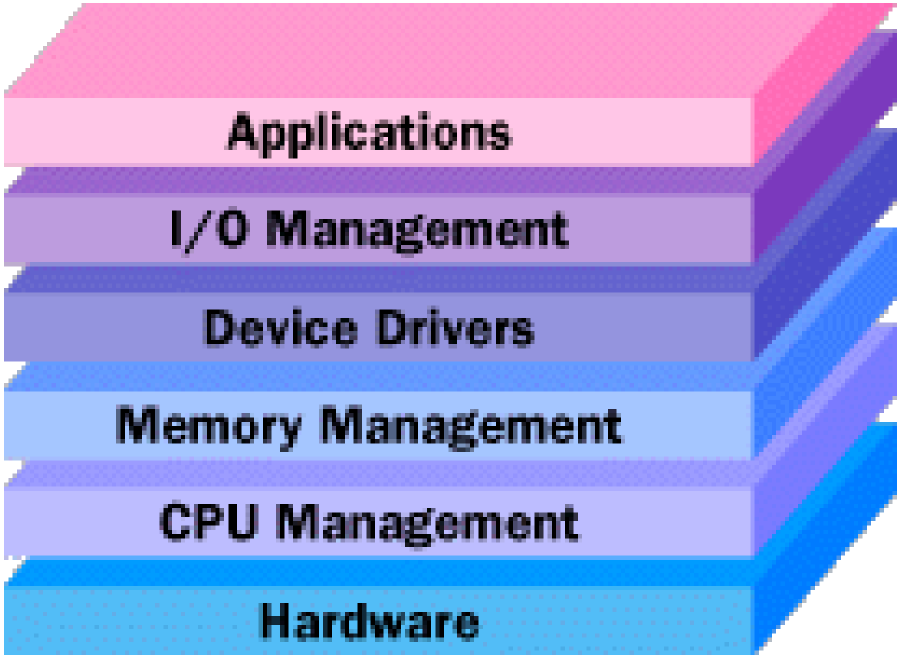


General OS Layers

Complex and careful implementation: As a layer can access the services of the layers below it, so the arrangement of the layers must be done carefully.

Slower in execution :

- If a layer wants to interact with another layer, it sends a request that has to travel through all the layers present in between the two interacting layers.
- Thus it increases response time, unlike the Monolithic system which is faster than this.
- Thus an increase in the number of layers may lead to a very inefficient design.



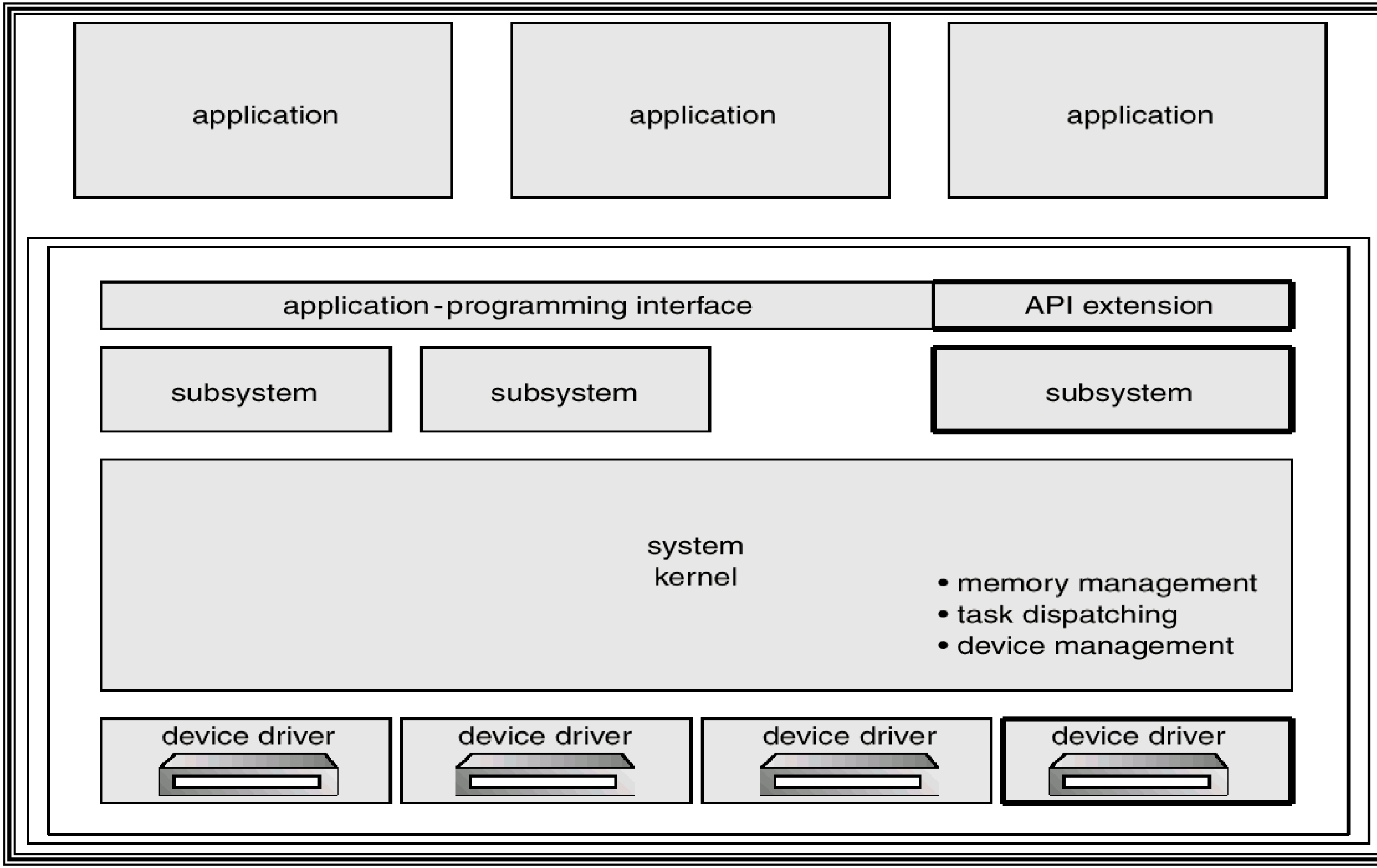
Structure of the THE operating system (1968)

The system built at the Technische Hoge school Eindhoven (THE), Netherland based on the layered approach.

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

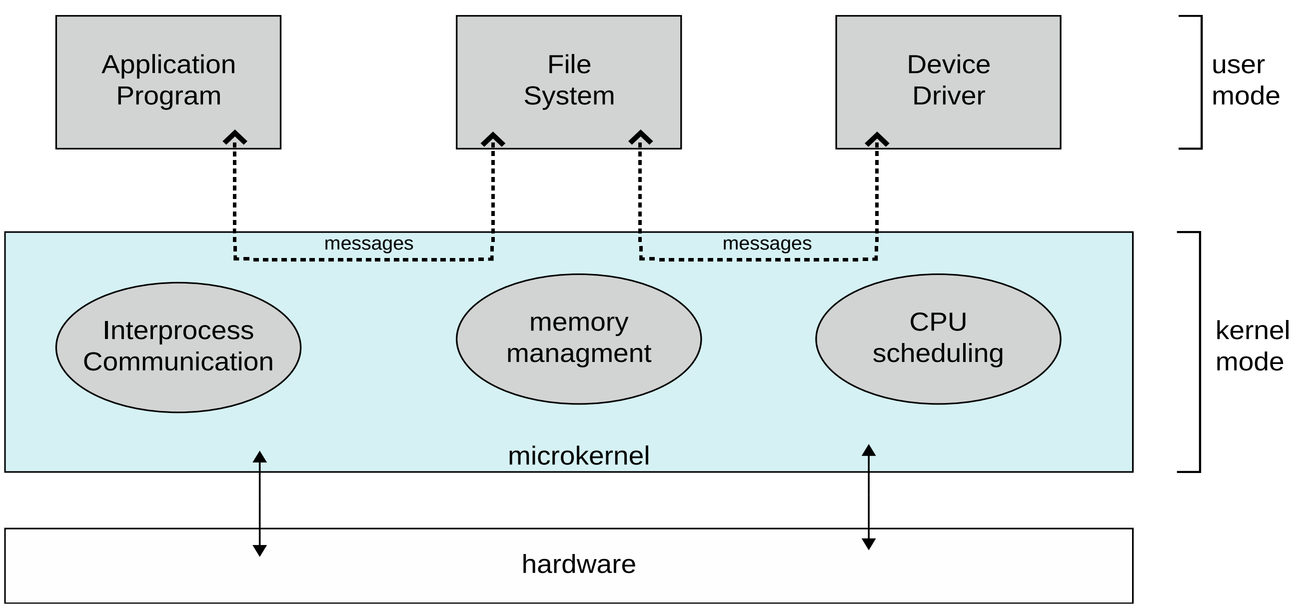
Older Windows System Layers

IBM OS/2 Layer Structure



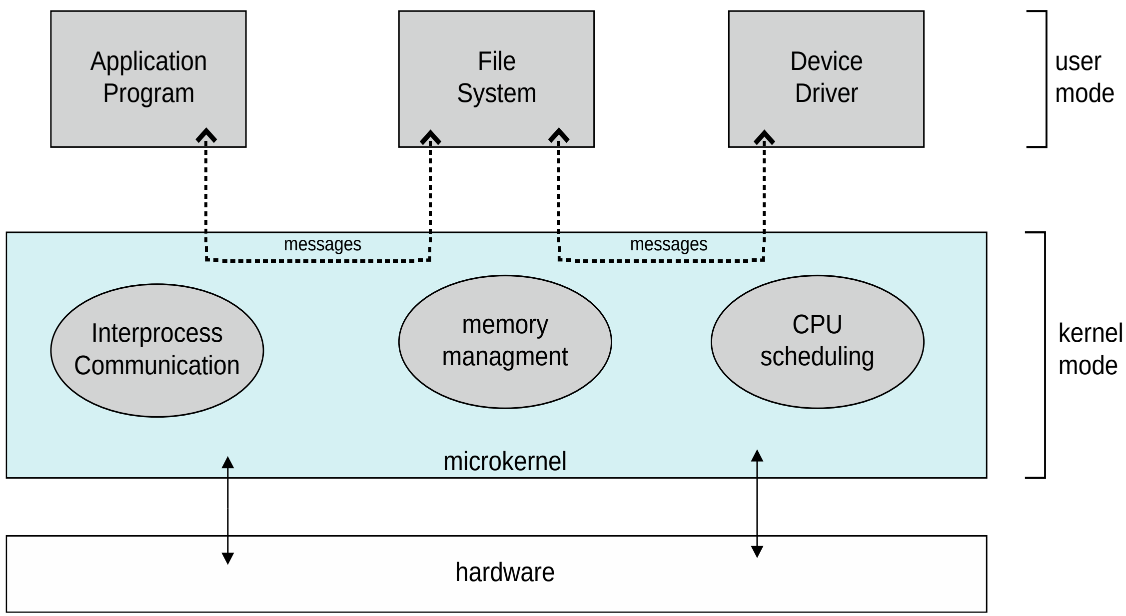
4. Microkernel System Structure [Com]

- Moves as much from the kernel into user space, hence a small kernel
  - Kernel provides : process and memory management, and inter-process comm (IPC)
  - All non-essential components are either user or system programs
- **Mach** is an example of **microkernel**
  - Mac OS X kernel partly based on Mach
- Communication takes place between user modules using **message passing**
  - Function of microkernel: communication between client program and services

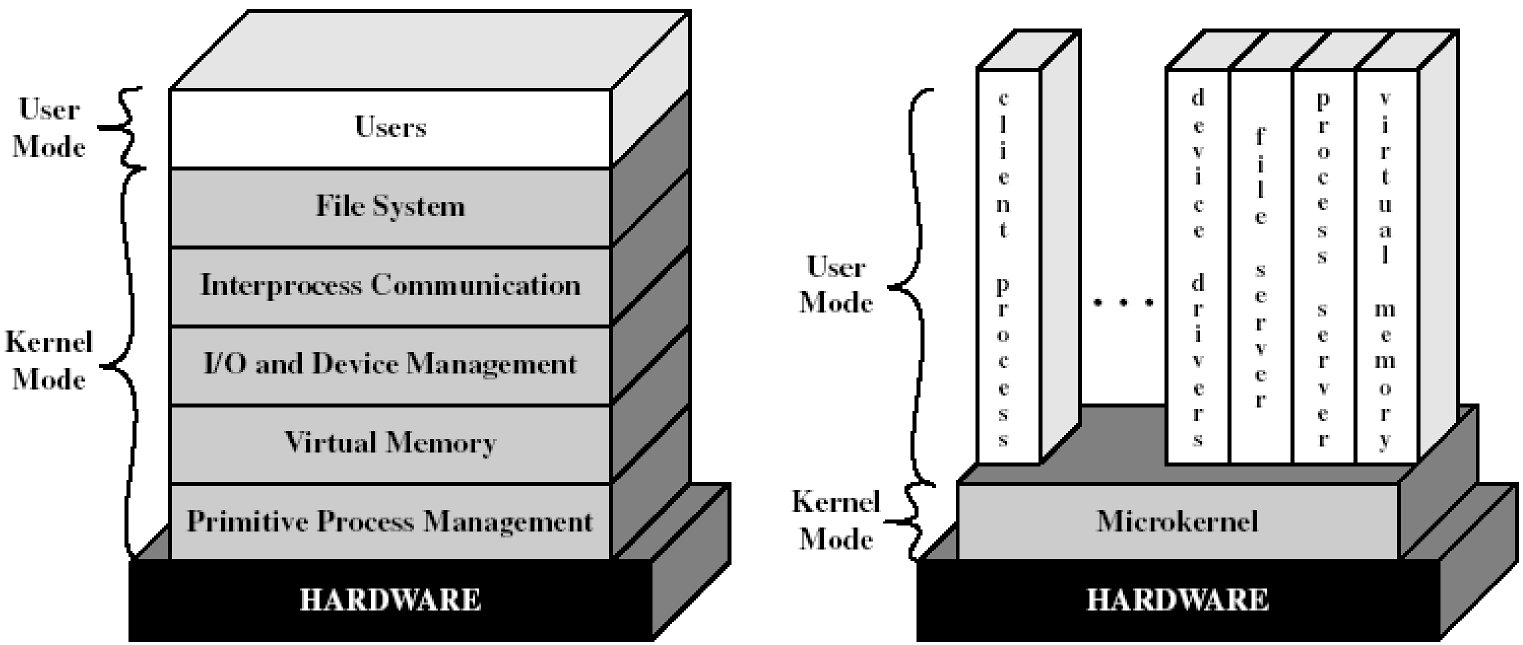


4. Microkernel System Structure

- Benefit:
  - More reliable (less code is running in kernel mode)
  - If a component outside the kernel misbehave, it does not affect the overall system
- Detriments:
  - Performance overhead of user space to kernel space communication
  - Slow message passing implementations are largely responsible for the poor performance



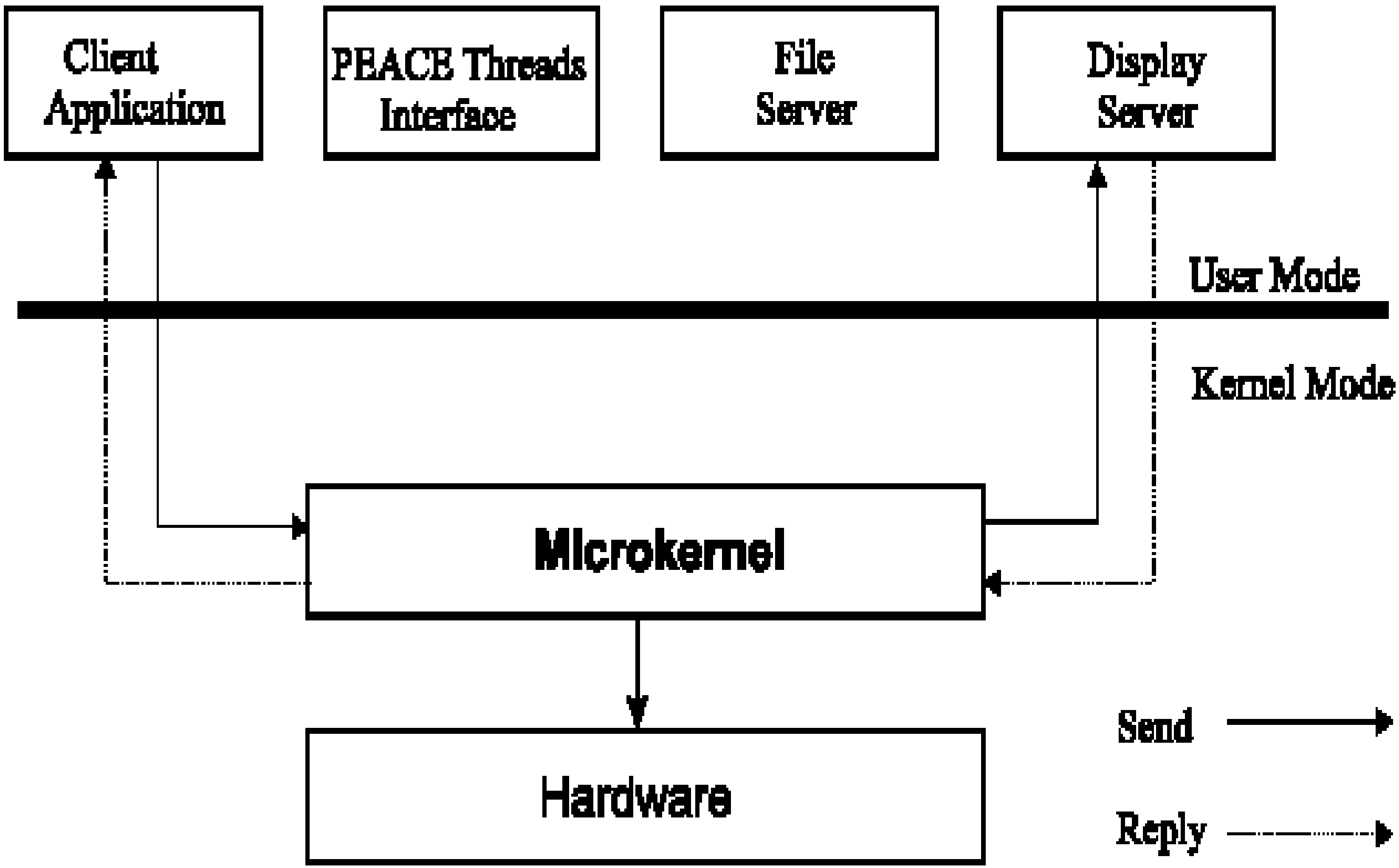
Layered vs. Microkernel Architecture



(a) Layered kernel

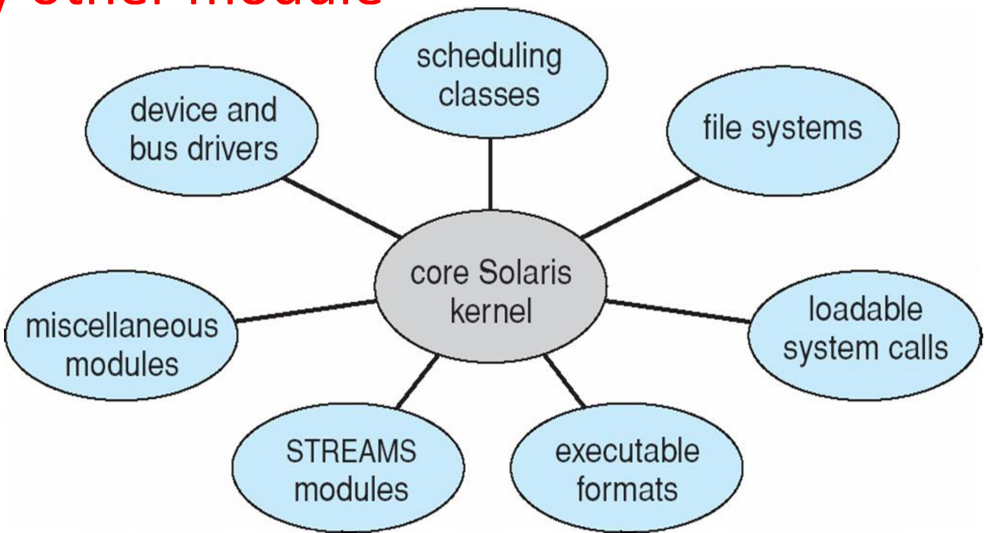
(b) Microkernel

Microkernel Operating System



5. Modules [com]

- Many modern operating systems implement **loadable kernel modules**
    - Kernel provides core services
    - Similar to microkernel
  - 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
  - Each kernel module can call any other module
  - Linux, Solaris, etc



6. Hybrid Systems [Com]

- Most modern operating systems are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels (memory management, IPC, Process scheduling) in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, but retains some behavior of microkernel systems.

End of Chapter 2