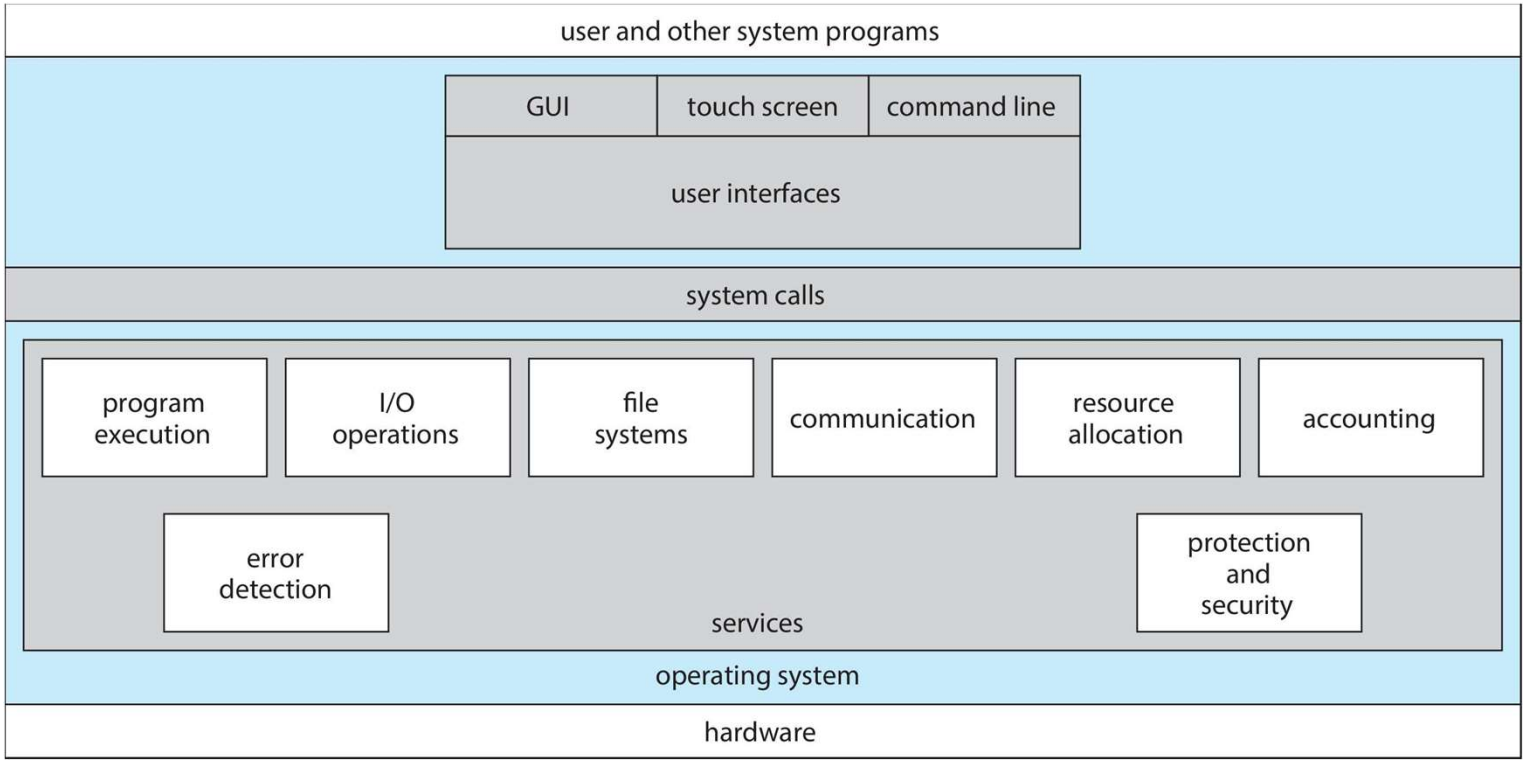
Chapter 2: Operating System Services and Structures

A View of Operating System Services



Operating System Services

- 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).

- Command-Line (CLI), Graphics User Interface (GUI)

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)

Operating System Services

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

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

Operating System Services

- 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

Accounting - To keep track of which users use how much and what kinds of computer resources

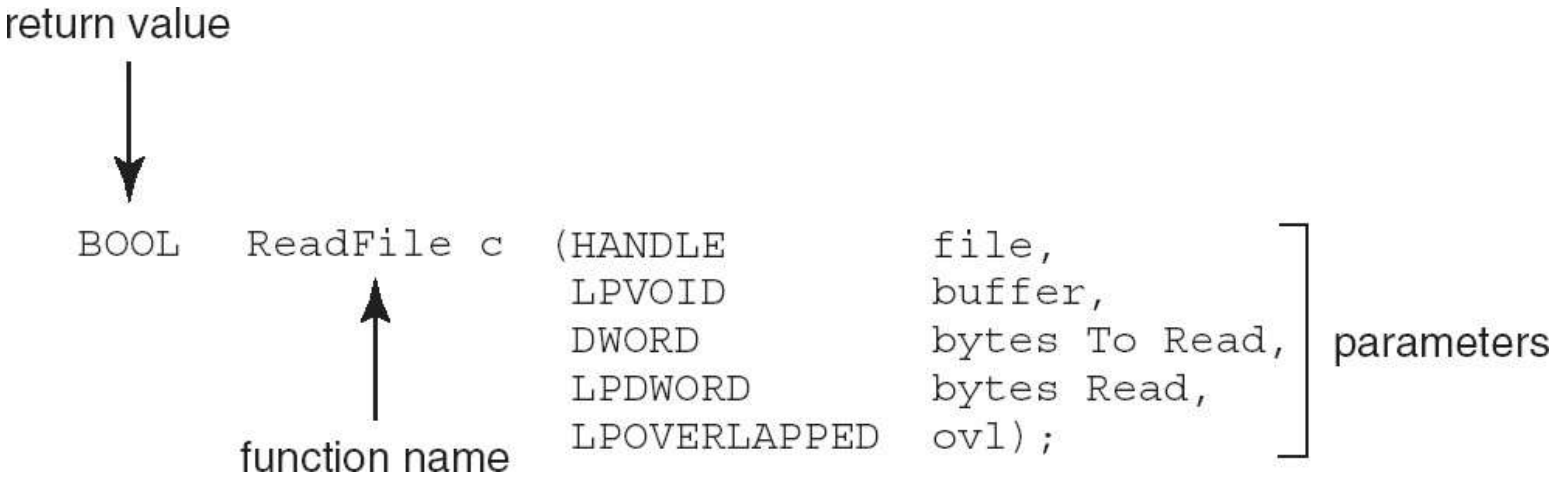
Protection and security - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other

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**) 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)
  - Java API for the Java virtual machine (JVM)

Example of System Calls

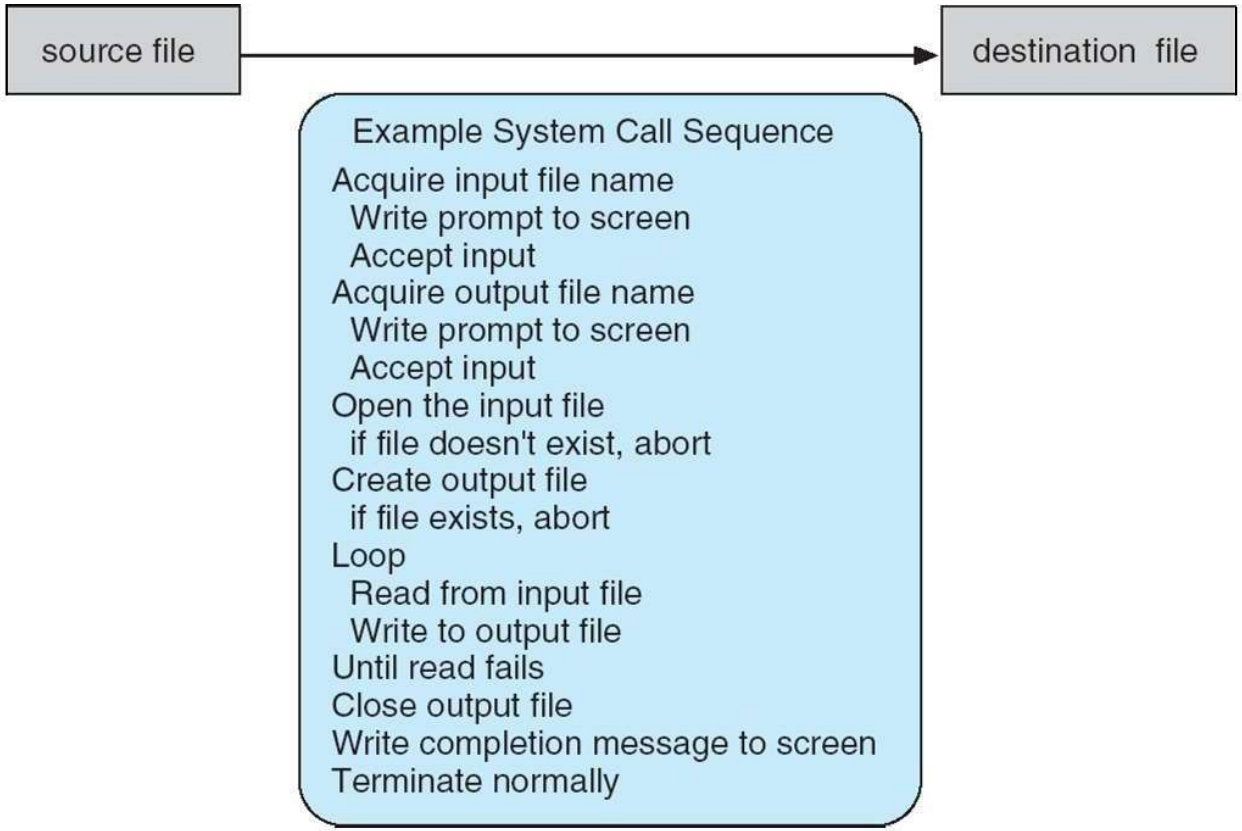
Consider the ReadFile() function in the Win32 API



Behind the scenes, the functions that make up an API typically invoke the actual system calls within the kernel on behalf of the application programmer.

Example of System Calls

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



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)
```

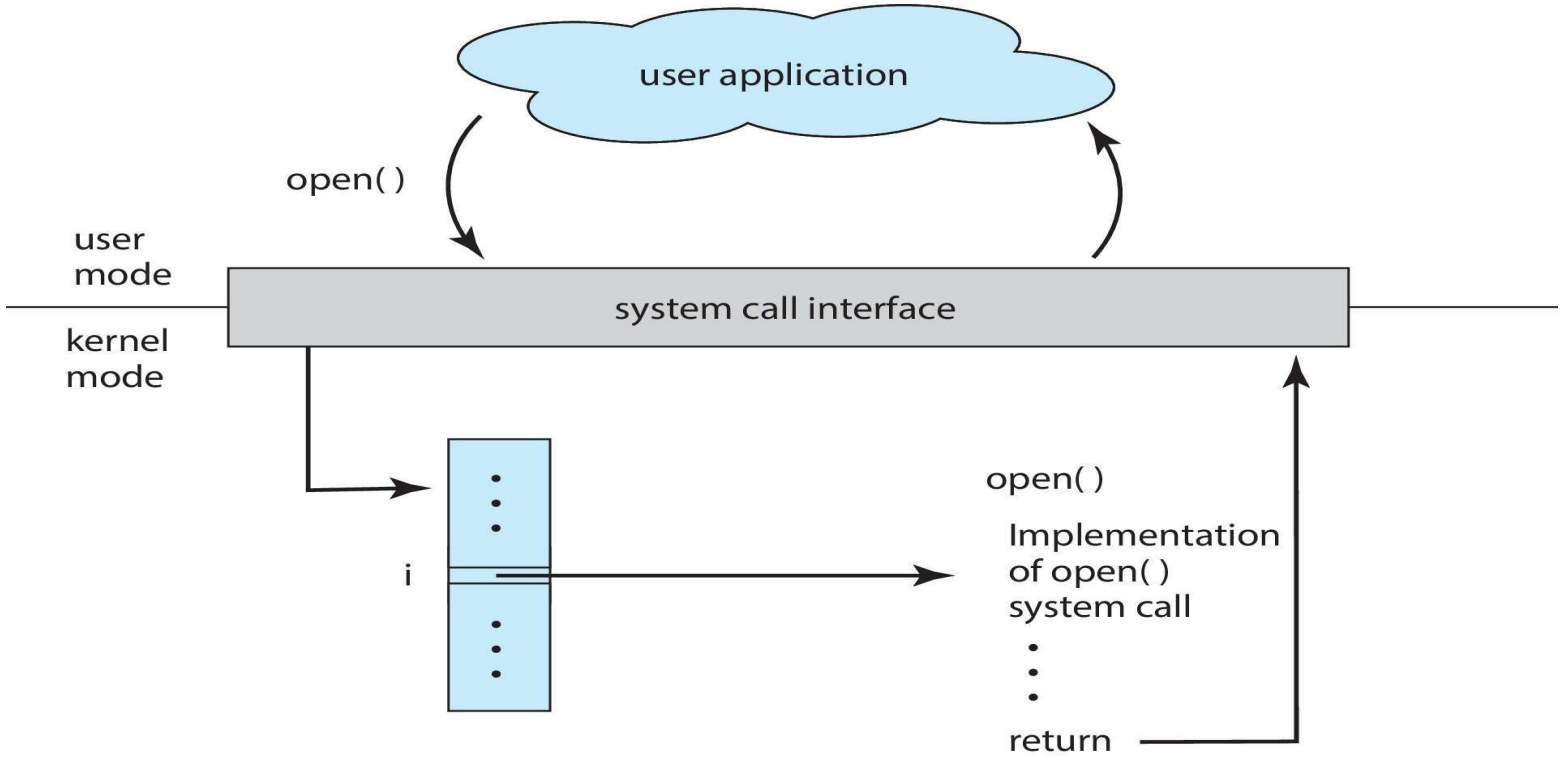
return value — function name — parameters

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 into which the data will be read
- `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.

API – System Call – OS Relationship



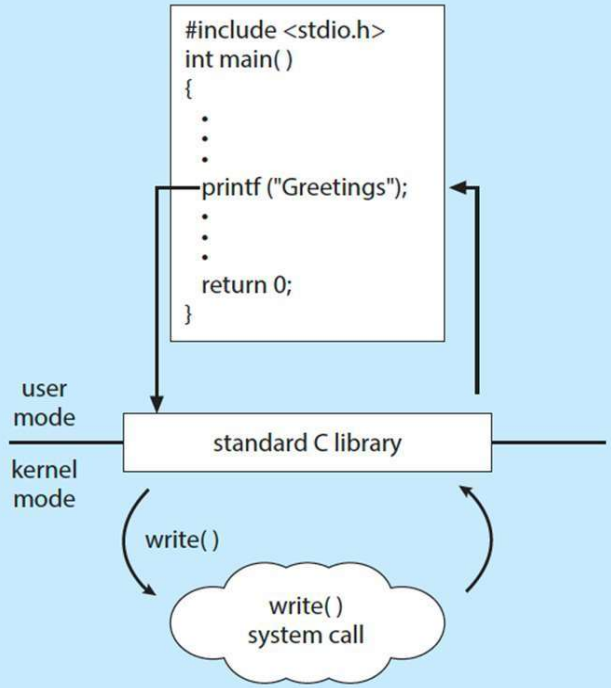
System call interface: run-time support library (set of functions built into libraries included with compiler)

Standard C Library Example

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

THE STANDARD C LIBRARY

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:



Types of System Calls

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - 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 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
- Communications

  create, delete communication connection

  send, receive messages

  transfer status information

  attach and detach remote devices

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()`
	`InitlializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

System Services

- Provide a convenient environment for program development and execution.

  They can be divided into:

  - **File management**
  - **Status information**
  - **File modification**
  - **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

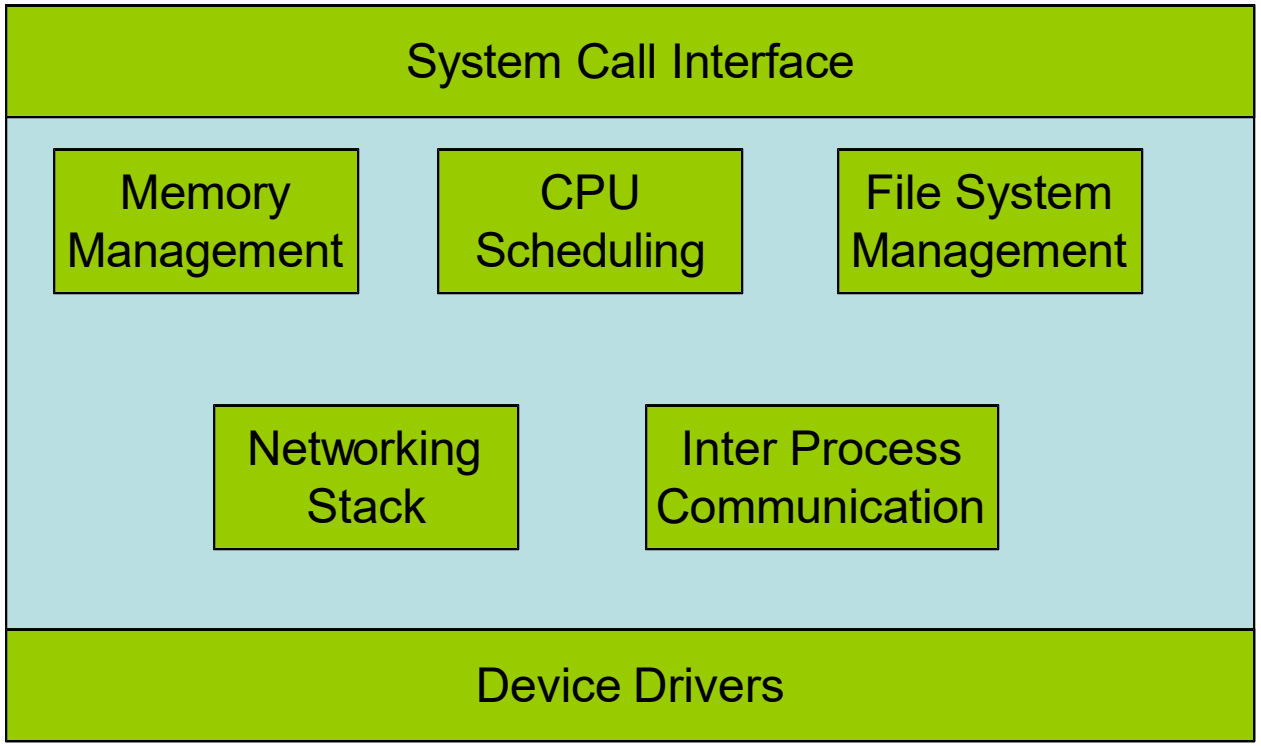
Implementation

- 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

Operating System Structure

- General-purpose OS is very large program
- Various ways to structure ones
  - Simple structure (monolithic) – MS-DOS
  - Layered – an abstraction
  - Microkernel – Mach
  - Modules – Solaris

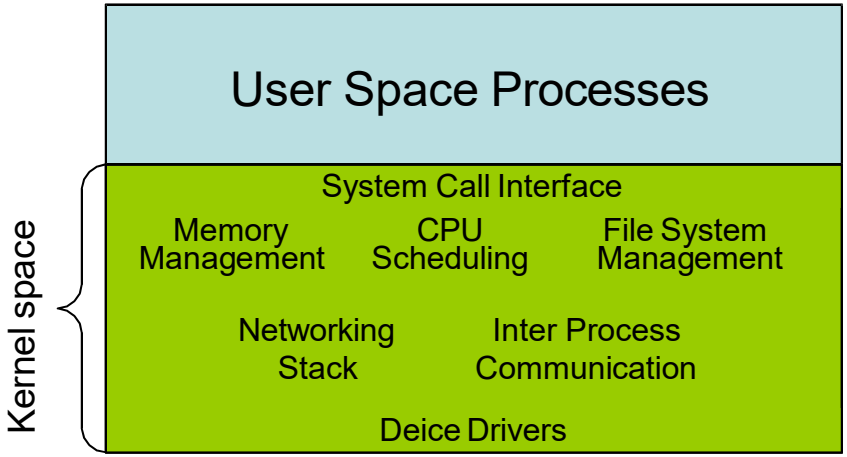
What goesintoan OS?

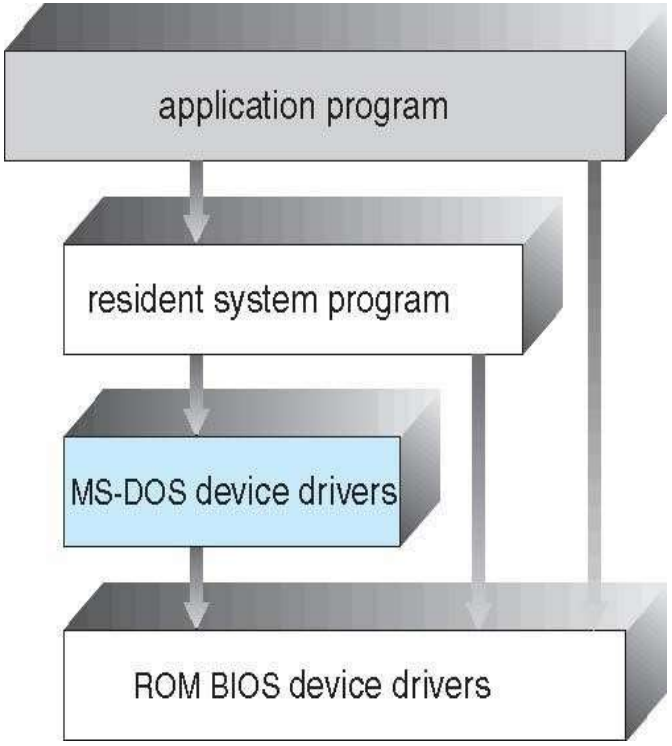


Simple Structure (monolithic)

- Many commercial systems do not have well-defined structures.
- Frequently, such operating systems started as small, simple, and limited systems and then grew beyond their original scope.
- Example: MS-DOS
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

Monolithic Structure





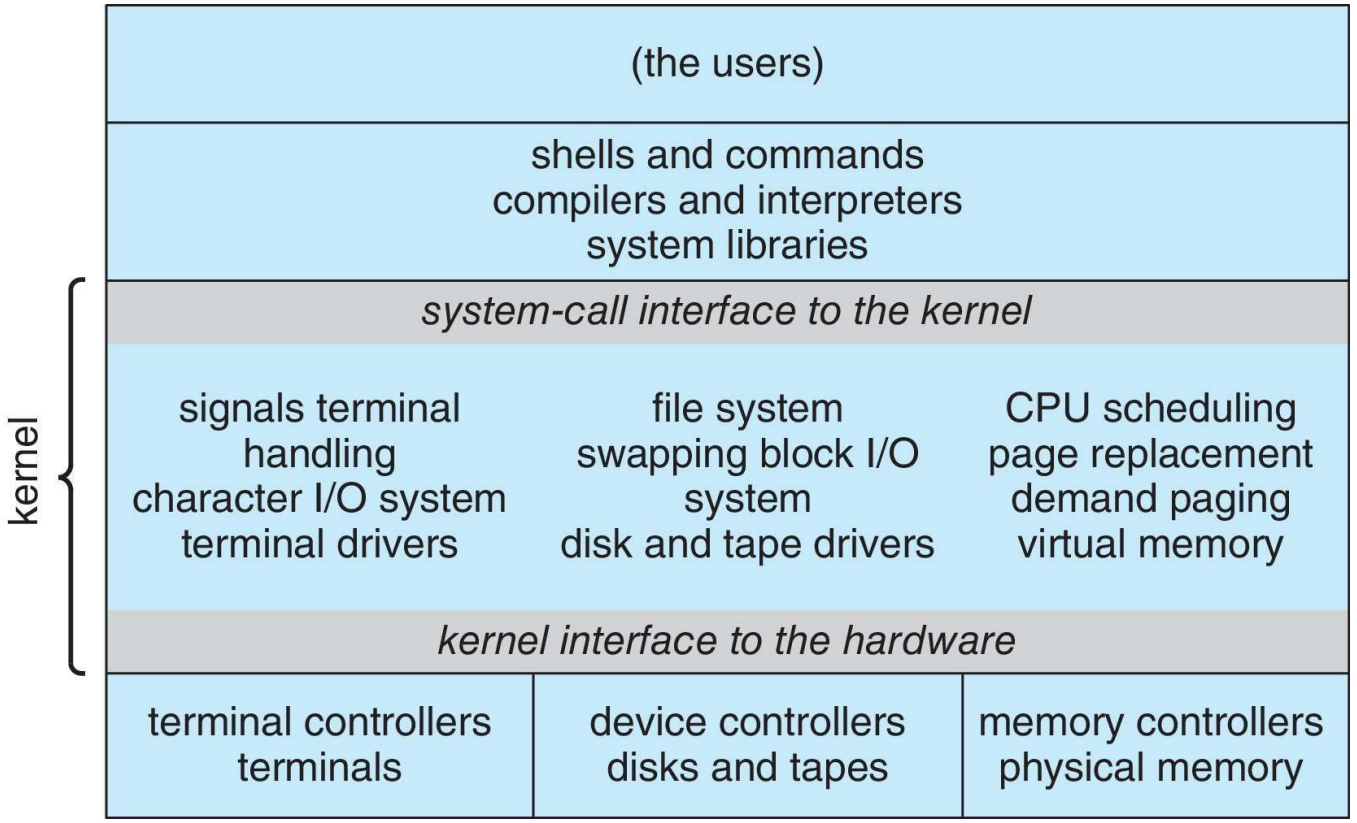
- Linux, MS-DOS, xv6
- All components of OS in kernel space
- Cons: Large size, difficult to maintain, likely to have more bugs, difficult to verify
- Pros: direct communication between modules in the kernel by procedure calls

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
  - 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; a large number of functions for one level

Traditional UNIX System Structure

Beyond simple but not fully layered



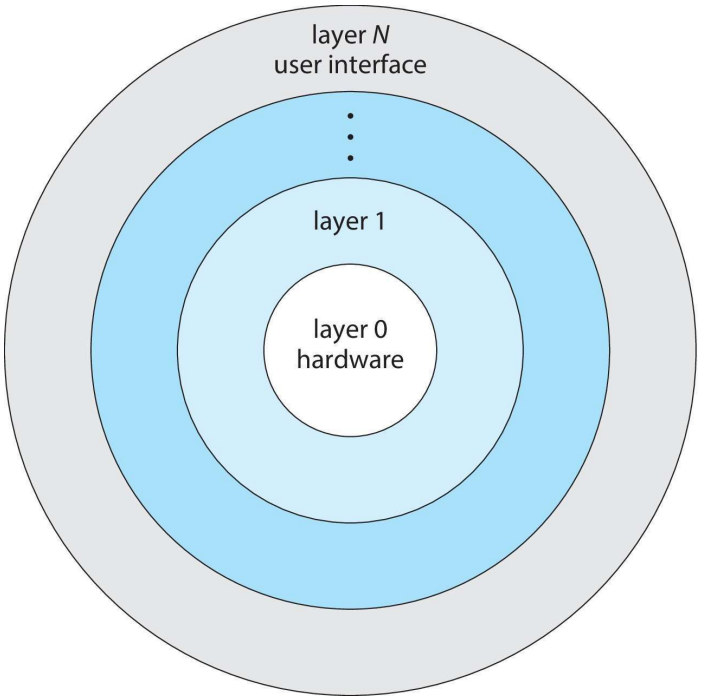
Layered Approach

- 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
- Benefits:

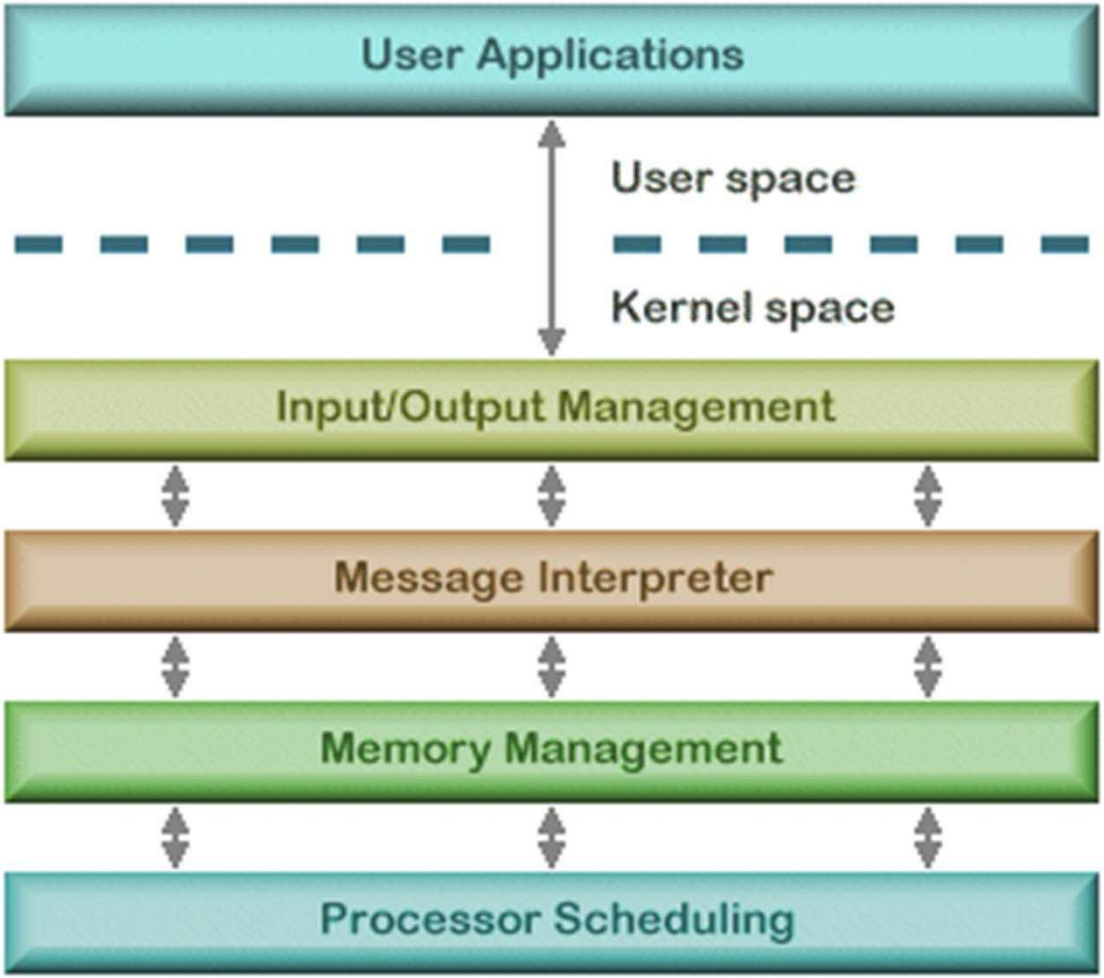
  simplicity of construction and debugging
- Detriments:

  Appropriately defining the various layers

  Less efficient than other types



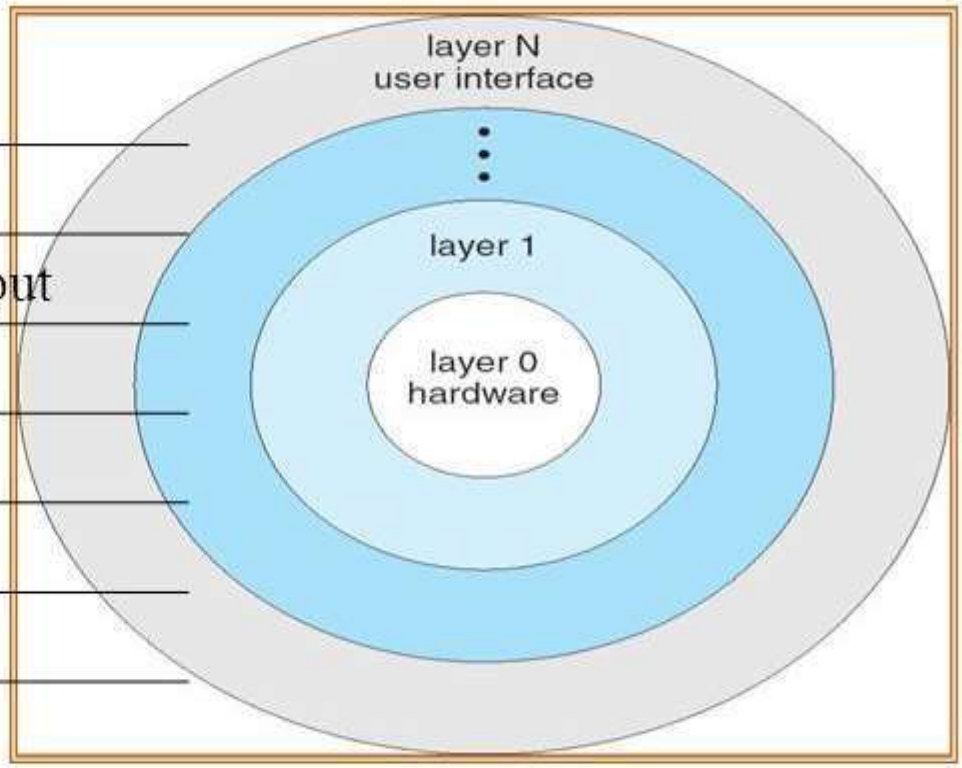
Layered Approach



Layered Approach

Its six layers are as follows:

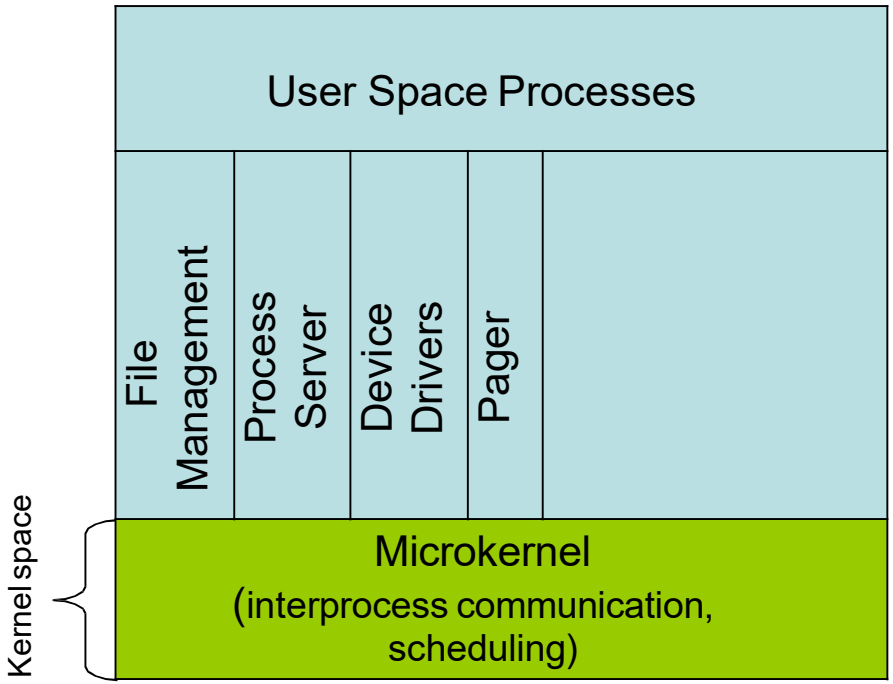
layer 5:	user programs
layer 4:	buffering for input and output
layer 3:	Process management
layer 2:	memory management
layer 1:	CPU scheduling
layer 0:	hardware



Microkernels

- only absolutely essential core OS functions should be in the kernel.
- Less essential services and applications are built on the microkernel and execute in user mode.
- Example: Mach ⟶ Carnegie Mellon University
- Typically, microkernels provide minimal process and memory management, in addition to a communication facility between the client program and the various services that are also running in user space

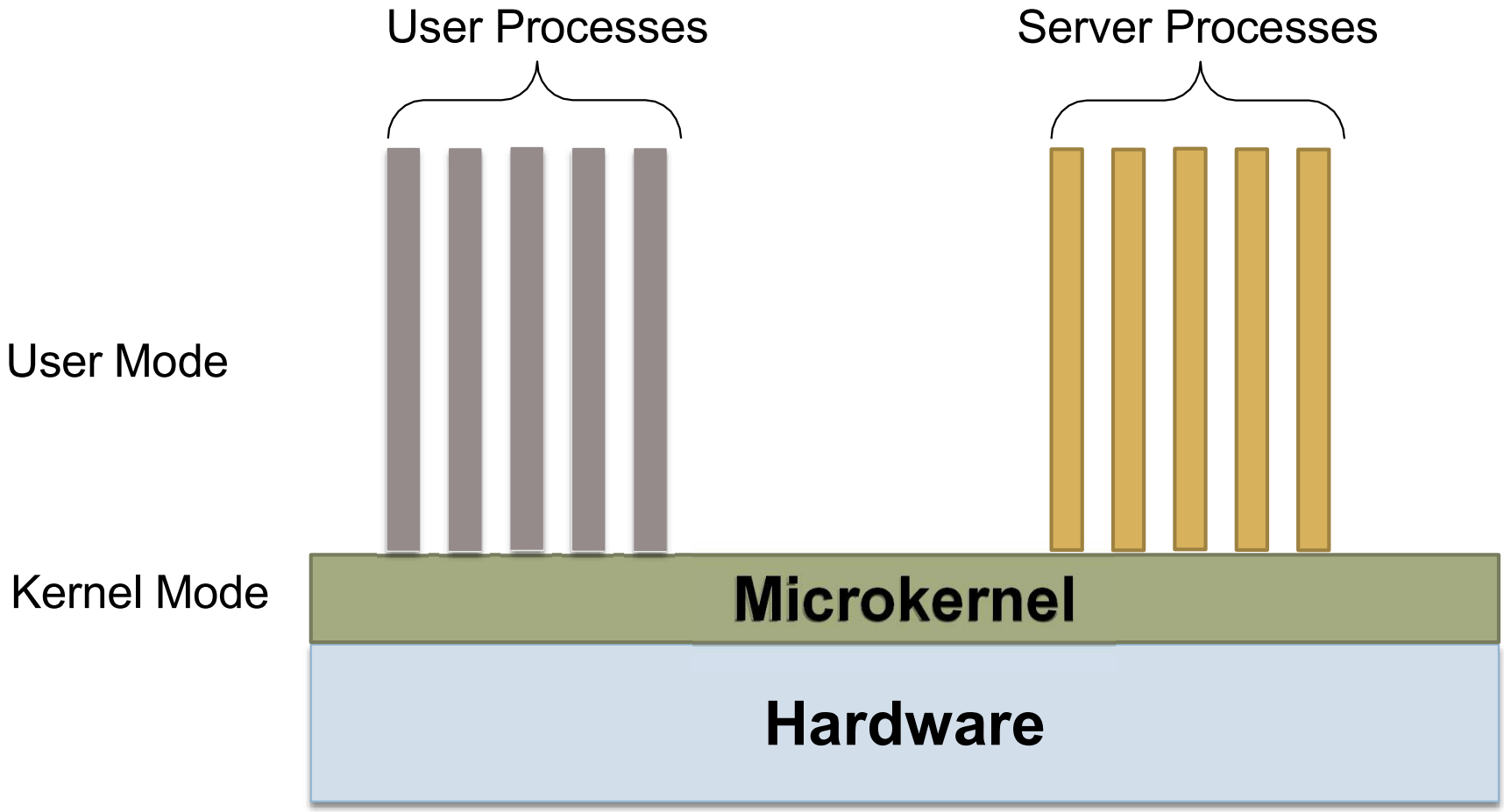
Microkernel



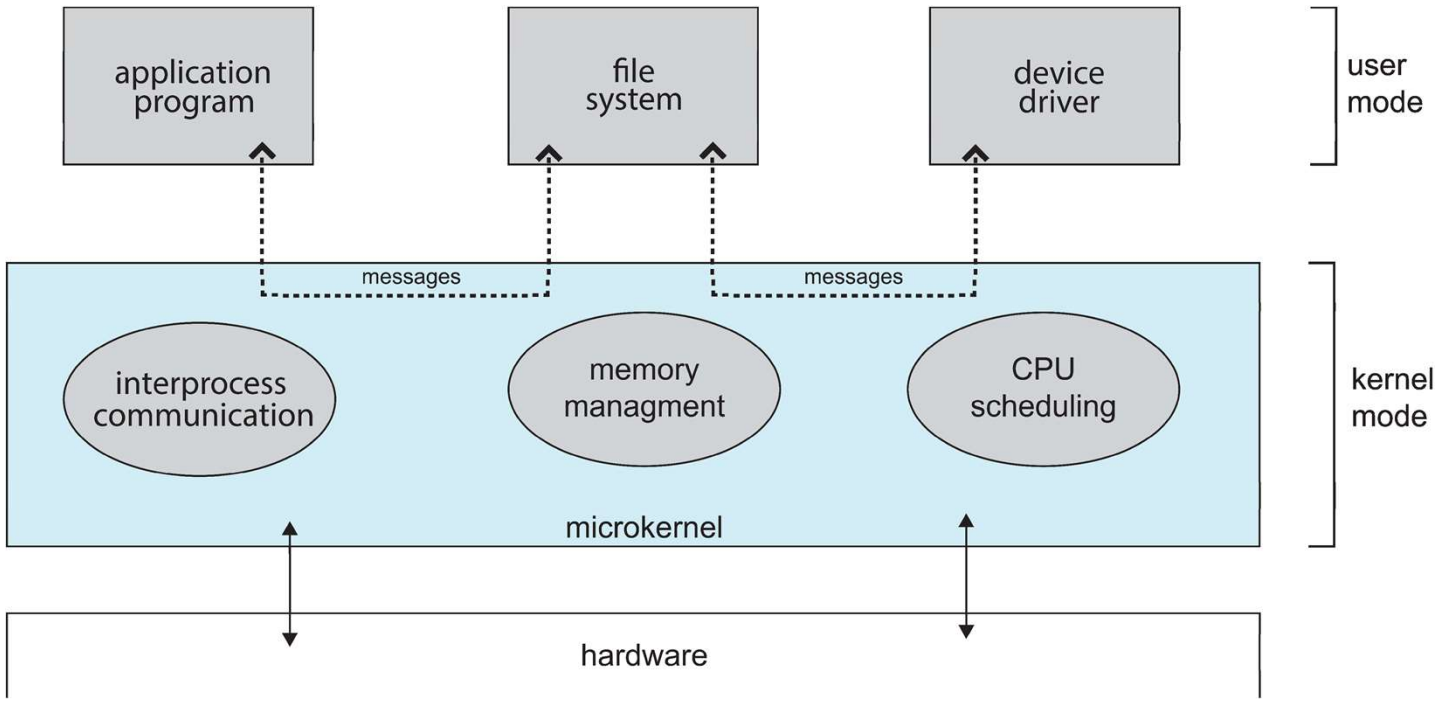
Eg. QNX and L4

- Highly modular.
  - Every component has its own space.
  - Interactions between components strictly through well defined interfaces (no backdoors)
- Kernel has basic inter process communication and scheduling
  - Everything else in user space.
  - Ideally kernel is so small that it fits the first level cache

Microkernels



Microkernel System Structure



Microkernels

- Moves as much from the kernel into user space
- **Mach** is an example of **microkernel**
  - 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 space to kernel space communication

Microkernel vs. Layered Approach

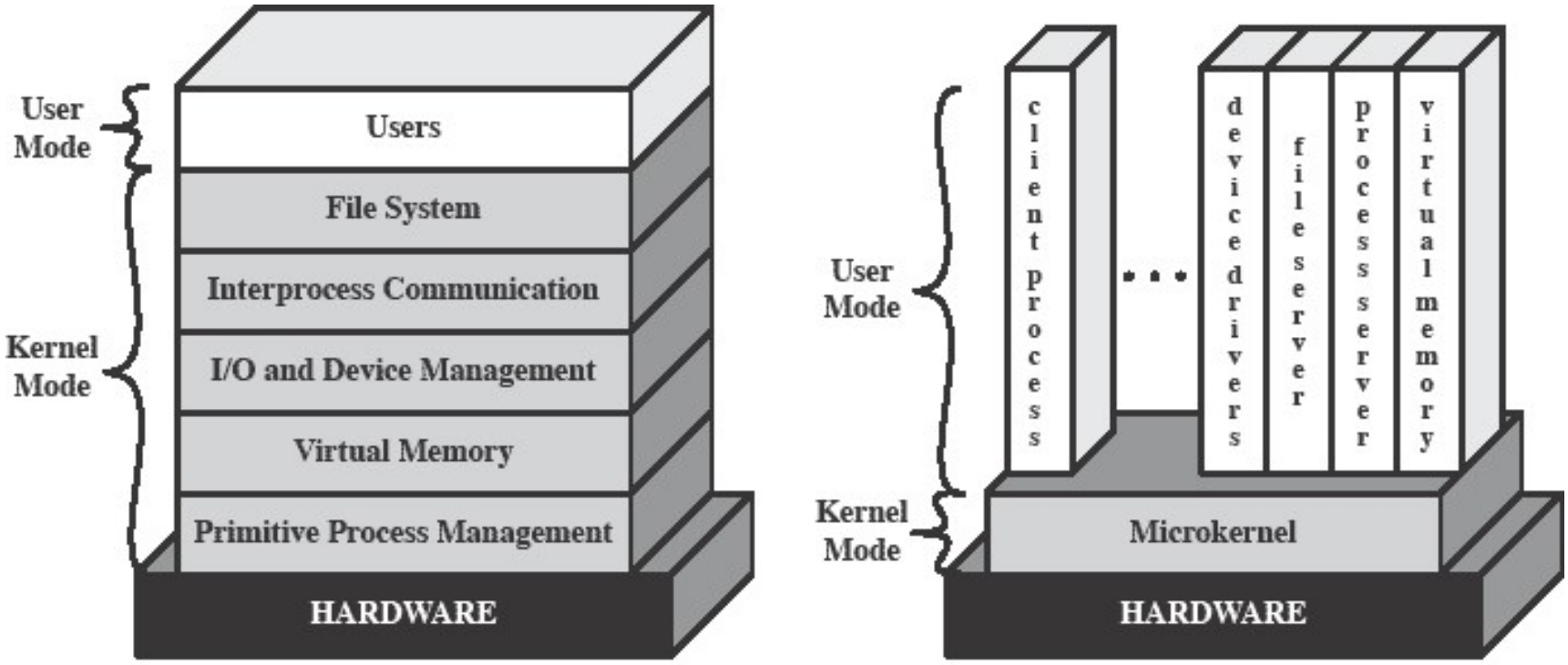
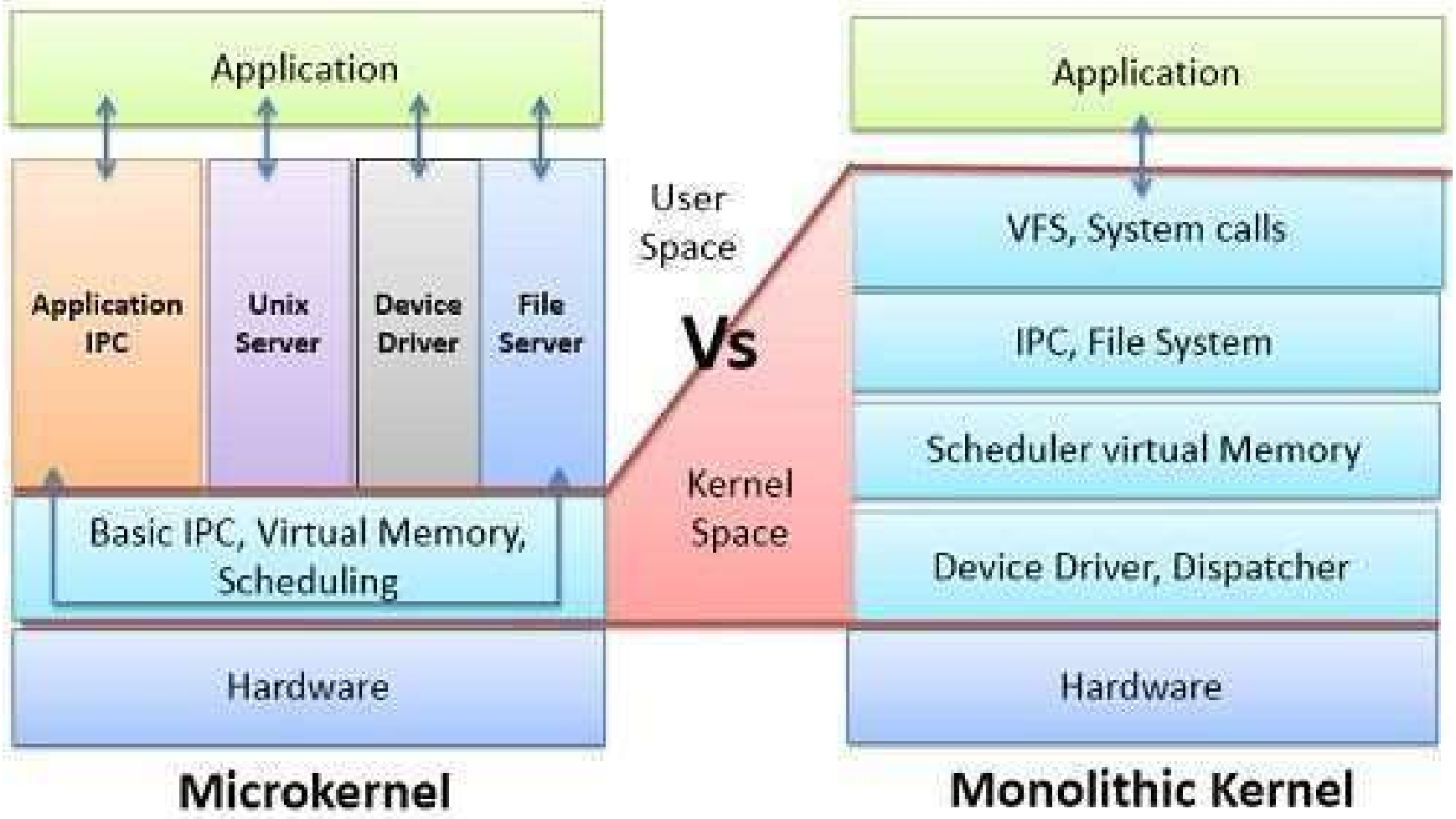


Figure 4.10 Kernel Architecture

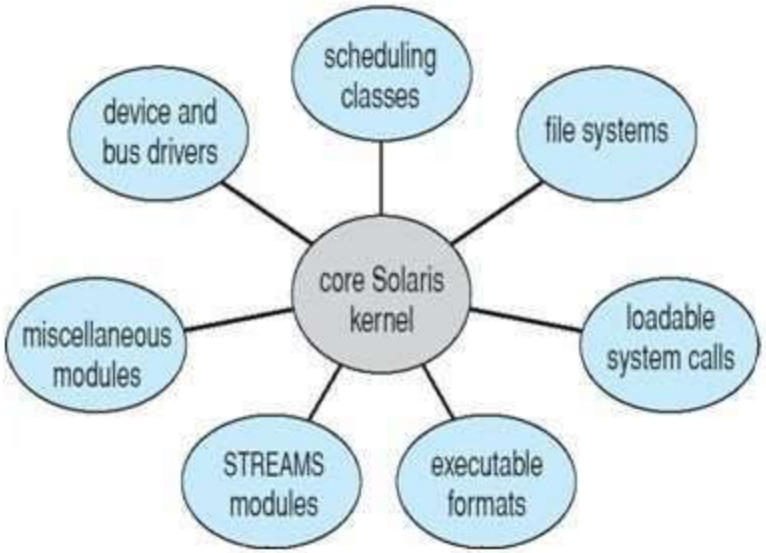

Application
Application IPC
Unix Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling
Hardware
Microkernel
User Space
Vs
Kernel Space
Application
VFS, System calls
IPC, File System
Scheduler virtual Memory
Device Driver, Dispatcher
Hardware
Monolithic Kernel

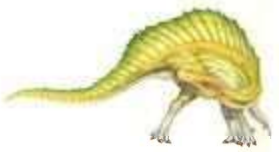
Monolithic vs Microkernels

	Monolithic	**Microkernel**
Inter process communication	Signals, sockets	Message queues
Memory management	Everything in kernel space (allocation strategies, page replacement algorithms,)	Memory management in user space, kernel controls only user rghts
Stability	Kernel more ‘crashable’ because of large code size	Smaller code size ensures kernel crashes are less likely
I/O Communication (Interrupts)	By device drivers in kernel space. Request from hardware handled by interrupts in kernel	Requests from hardware converted to messages directed to user processes
Extendibility	Adding new features requires rebuilding the entire kernel	The micro kernel can be base of an embedded system or of a server
Speed	Fast (Less communication between modules)	Slow (Everything is a message)

Modules

- Many modern operating systems implement **loadable kernel modules**
- Kernel provides only core services
  - The rest is via modules
  - Modules can be loaded as needed
    - Dynamic loading
    - Unloaded when not needed
  - Modules loaded into the kernel space
    - More efficient than microkernel solution
    - Does not use message passing
- More flexible than layered approach
  - Any module can call any other module
  - Calls are over known interfaces





Modules

- Many modern operating systems implement **loadable kernel modules** (**LKMs**)
  - 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 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**)

Linux System Structure

Monolithic plus modular design