System Structures

Dr Tarunpreet Bhatia

Assistant Professor

CSED, TIET

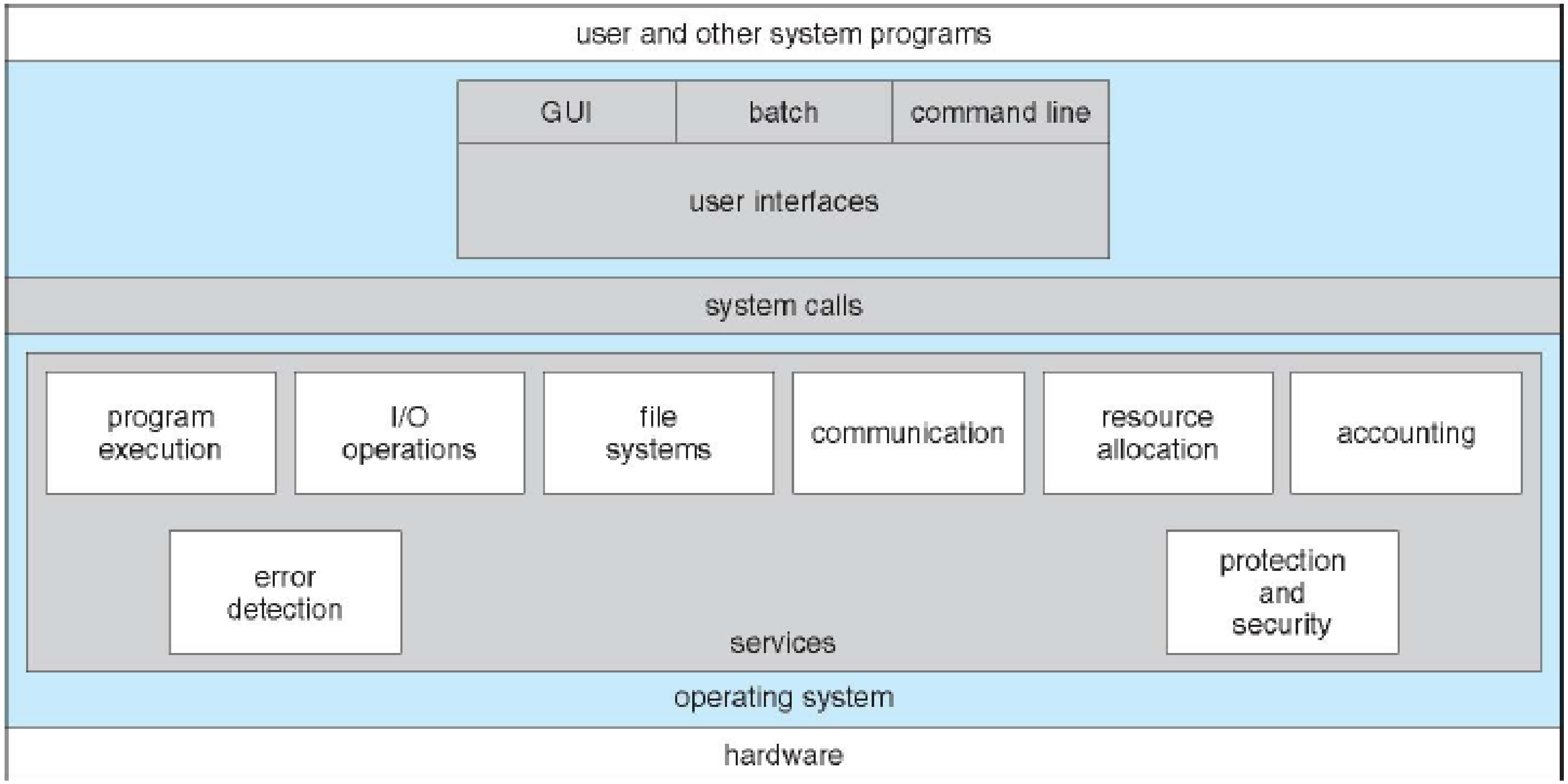
Disclaimer

This is NOT A COPYRIGHT MATERIAL

<u>Content has been taken mainly from the following books</u>:

Operating Systems Concepts By Silberschatz & Galvin,
Operating Systems: Internals and Design Principles By William Stallings

Operating System Services



System Call

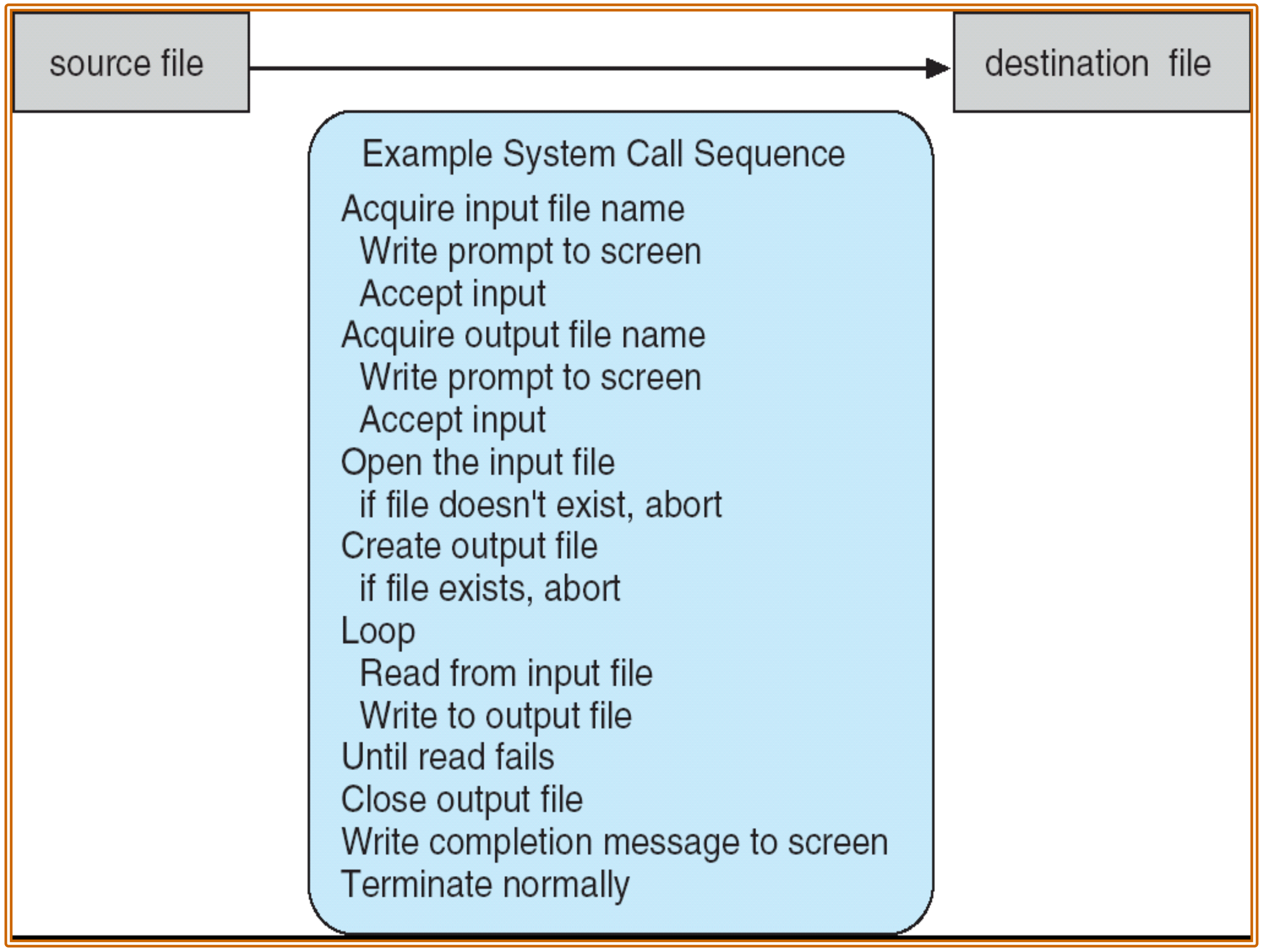
- Programming interface to the services provided by the OS.
- Typically written in a high-level language (C or C++)
- The invocation of an operating system routine.
- Operating systems contain sets of routines for performing various low-level operations.
- For example, all operating systems have a routine for creating a directory. If you want to execute an operating system routine from a program, you must make a system call.

API

- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use.

- Why use APIs rather than system calls? – There are specific reasons for using APIs or Programming with APIs, Instead of direct System Calls.

System Call Sequence when Copying from One File to another



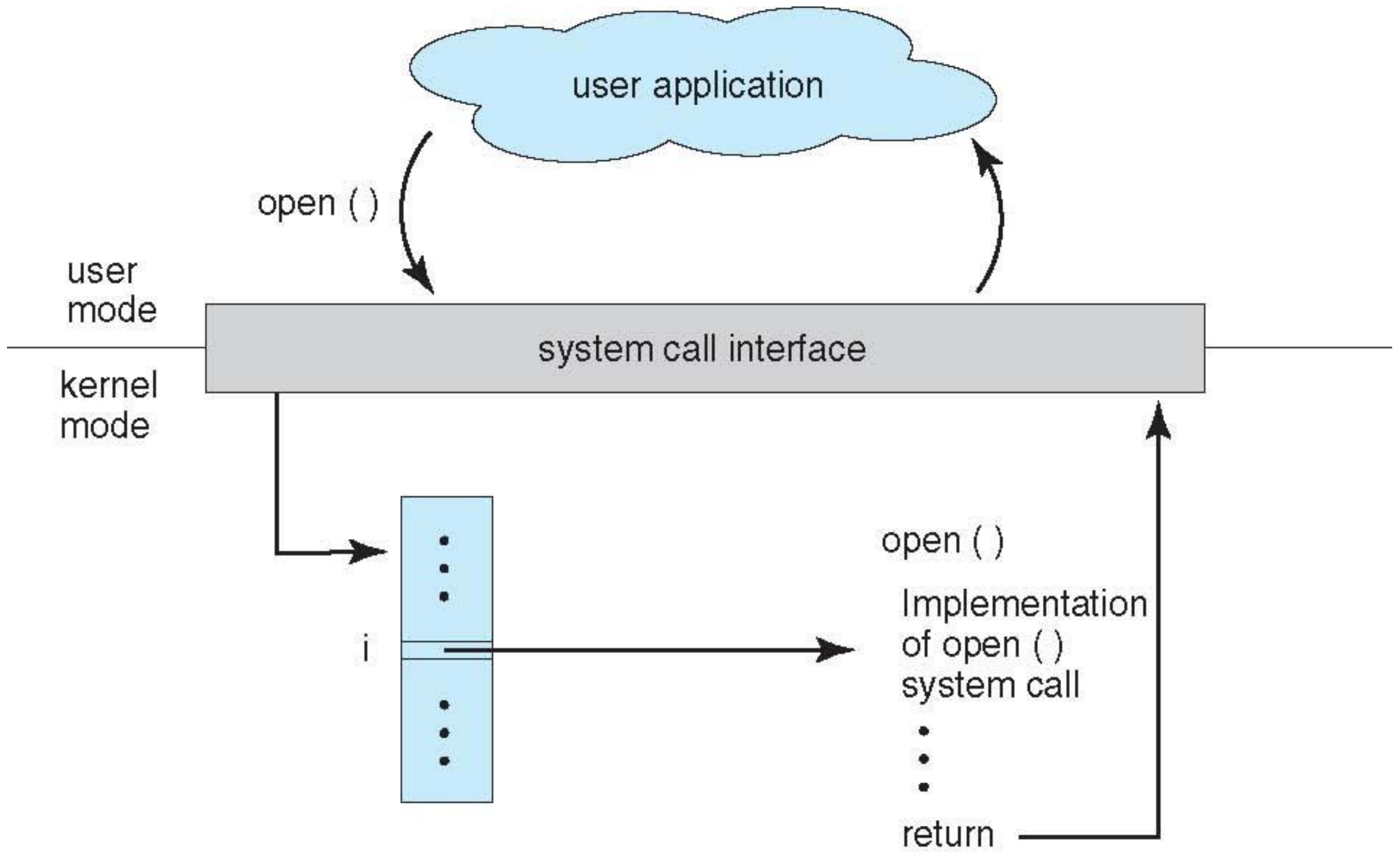
Example System Call Sequence

```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

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

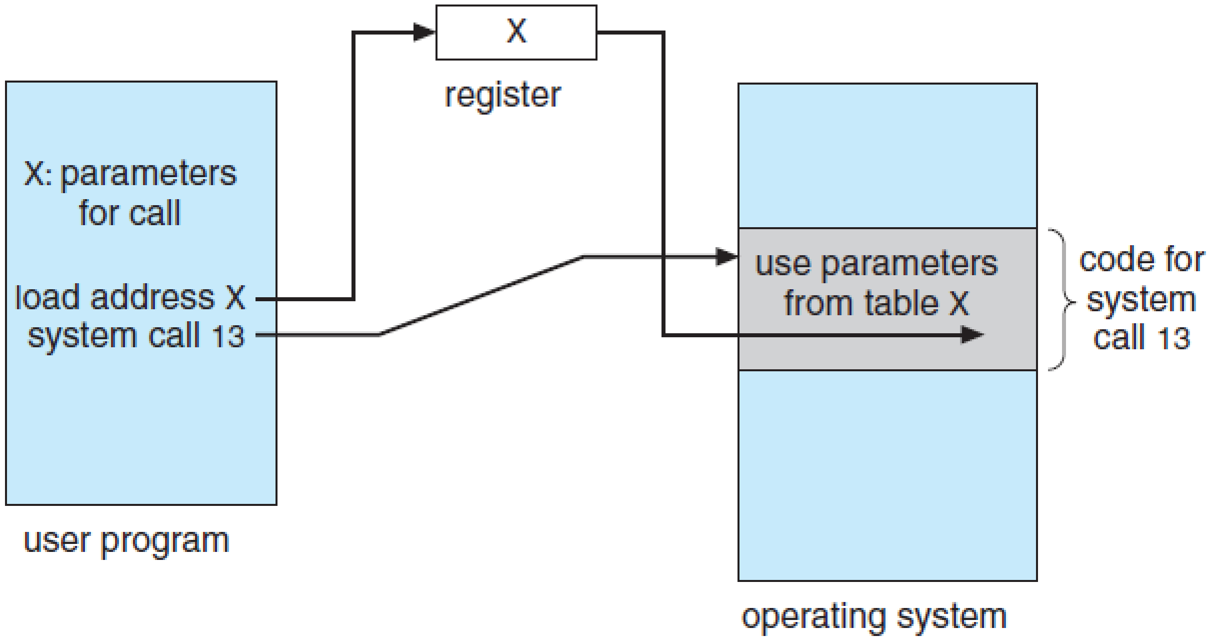
API – System Call – OS Relationship



System Call Parameter Passing

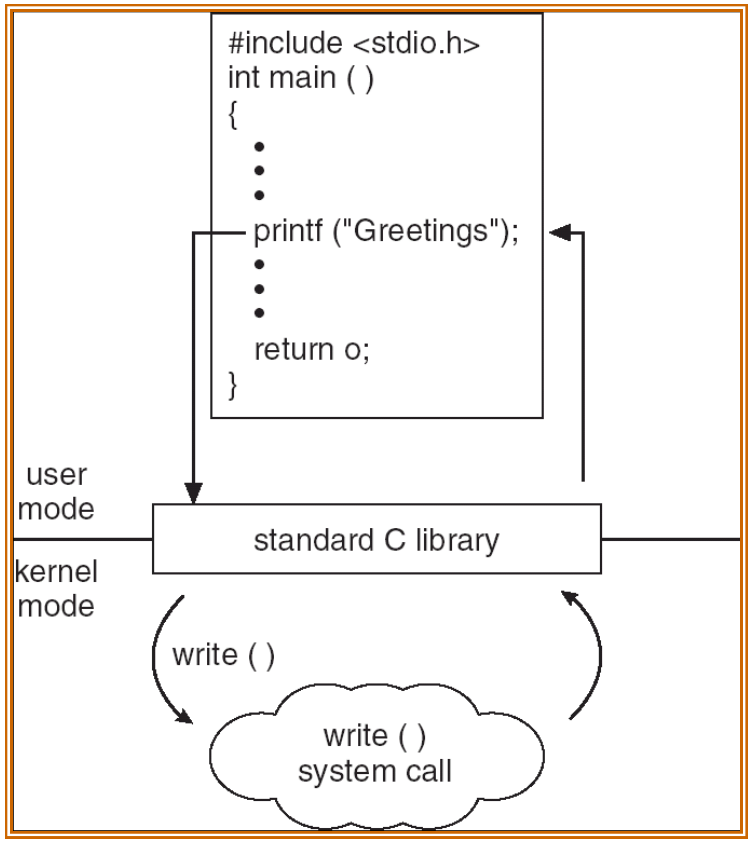
- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters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

Passing of parameters as a table

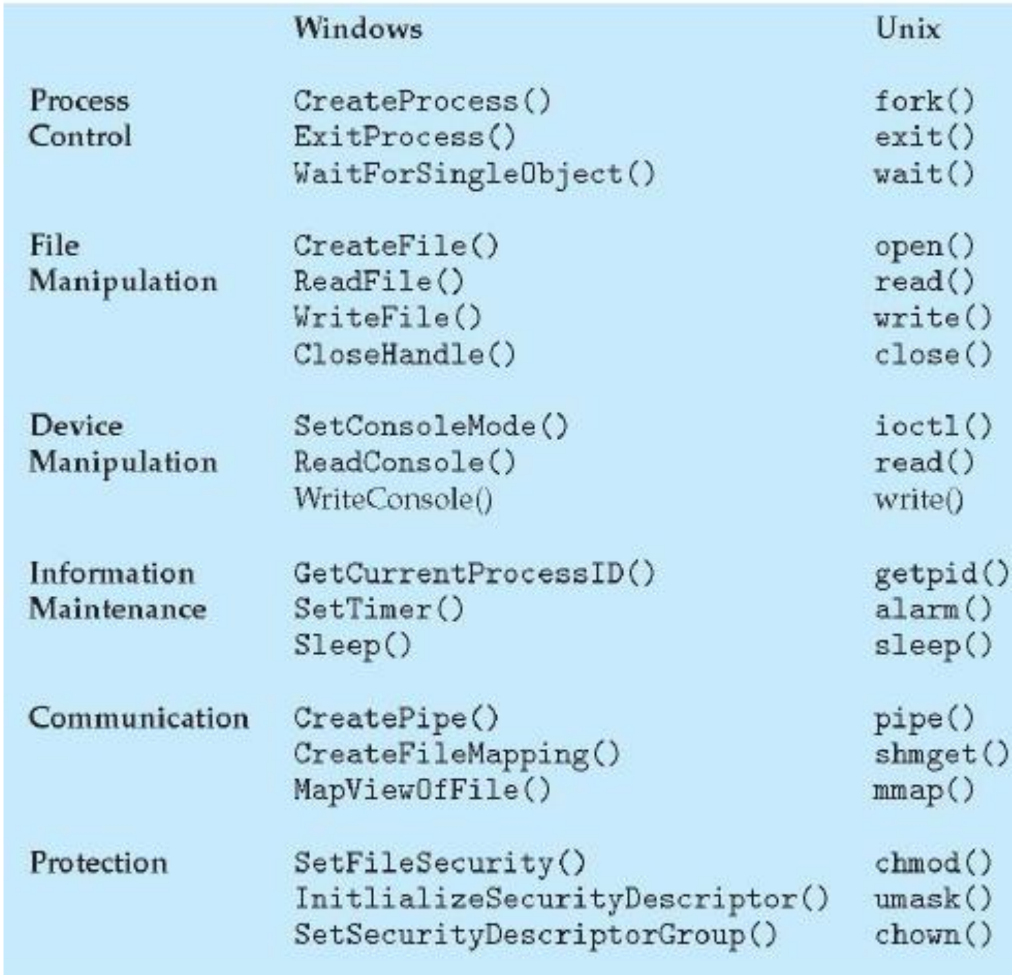


Standard C Library Example

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



Example – Windows/Unix system Call



	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()

System call Vs Library function

```
#include <unistd.h>
#include <sys/syscall.h>
#include <errno.h>

…

int rc;

rc = syscall(SYS_chmod,
"/etc/passwd", 0444);

if (rc == -1)
   fprintf(stderr, "chmod failed, errno
= %d\n", errno);
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>

…

int rc;

rc = chmod("/etc/passwd", 0444);
if (rc == -1)
   fprintf(stderr, "chmod failed, errno
= %d\n", errno);
```

Operating System Design & Implementation

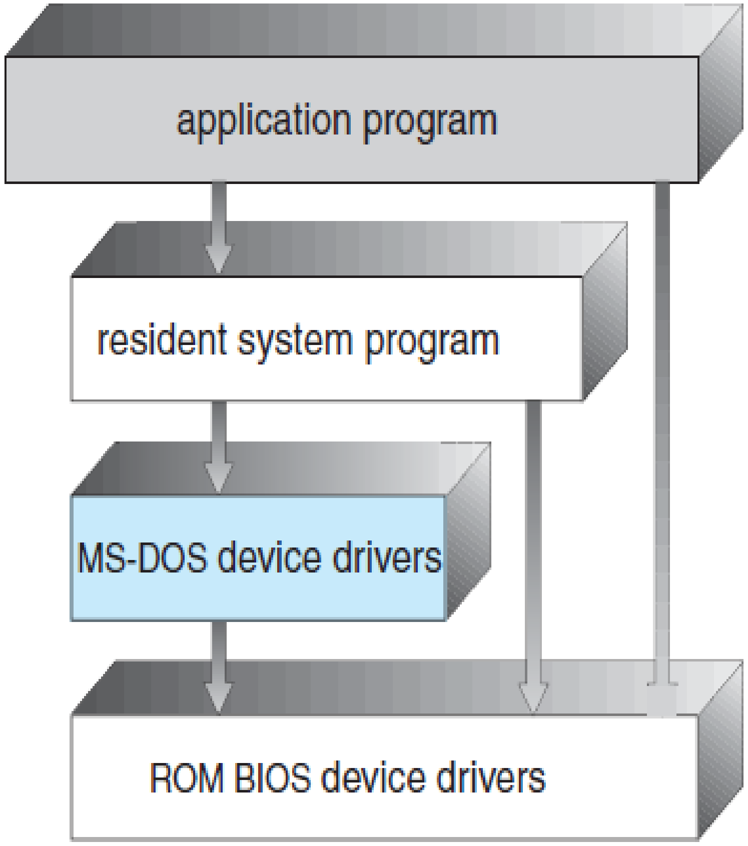
- Important principle to separate

  Policy: What will be done?
  Mechanism: How to do it?

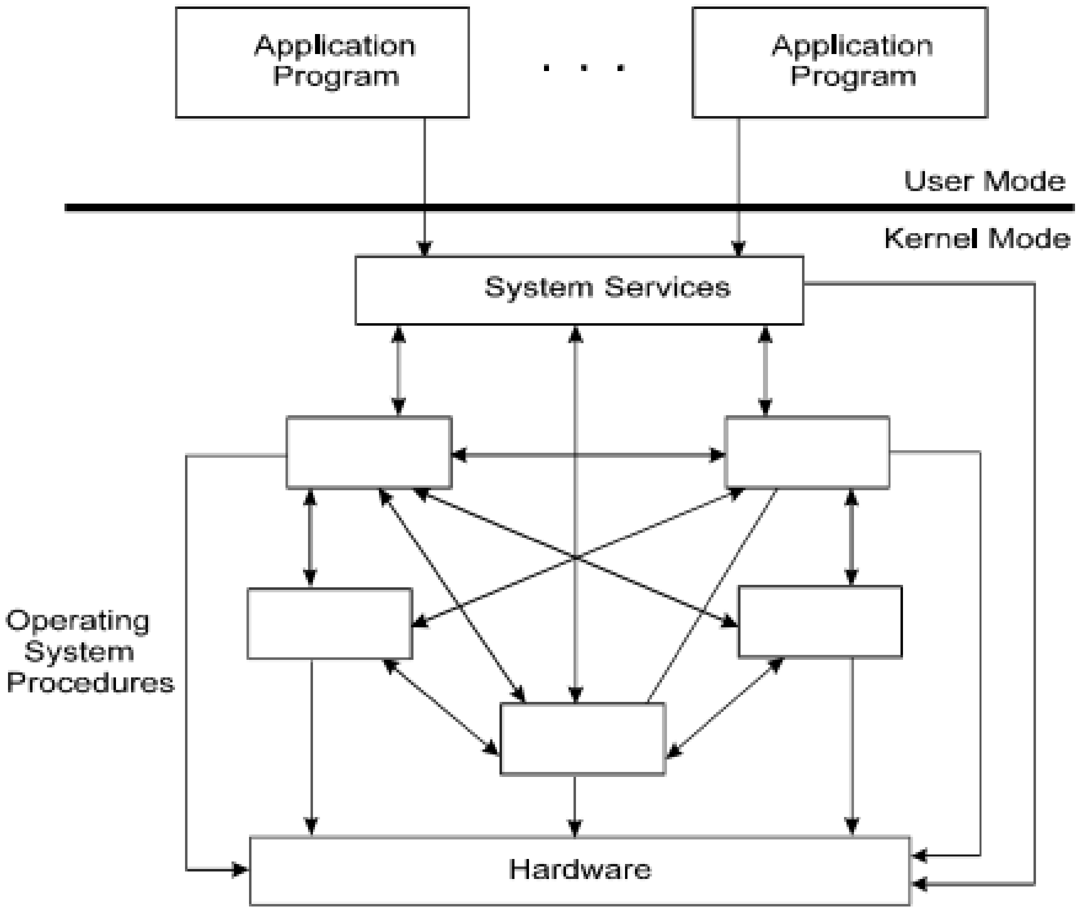
- Mechanisms determine how to do something, policies decide what will be done

  – The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

Simple Structure



Monolithic


Application Program
. . .
Application Program
User Mode
Kernel Mode
System Services
Operating System Procedures
Hardware

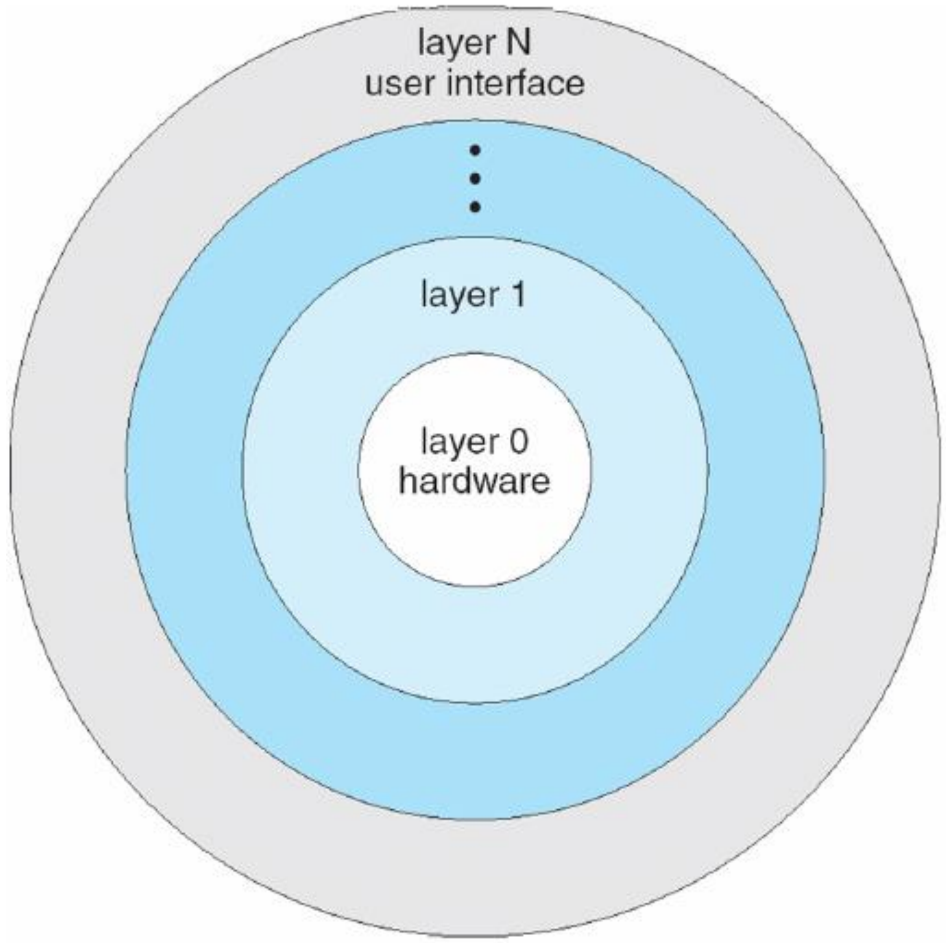
Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers.

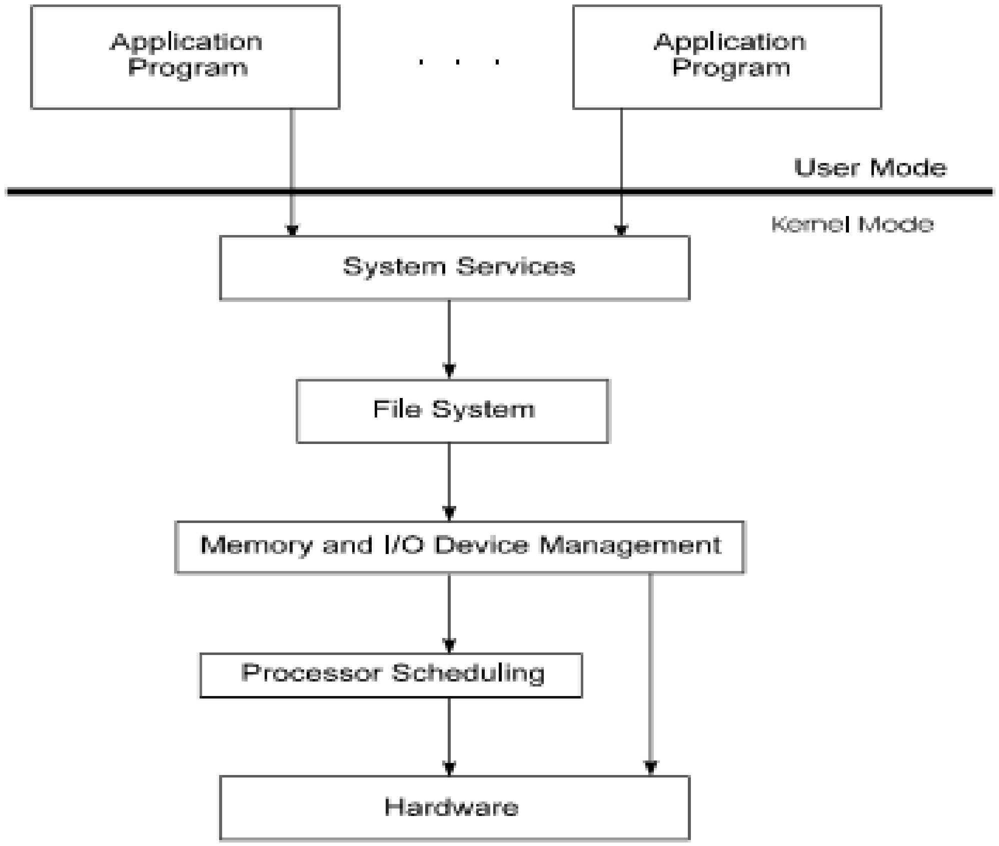
- The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.

- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

Layered View of Operating System



Layered OS


Application Program
. . .
Application Program
User Mode
Kernel Mode
System Services
File System
Memory and I/O Device Management
Processor Scheduling
Hardware

Microkernel System Structure

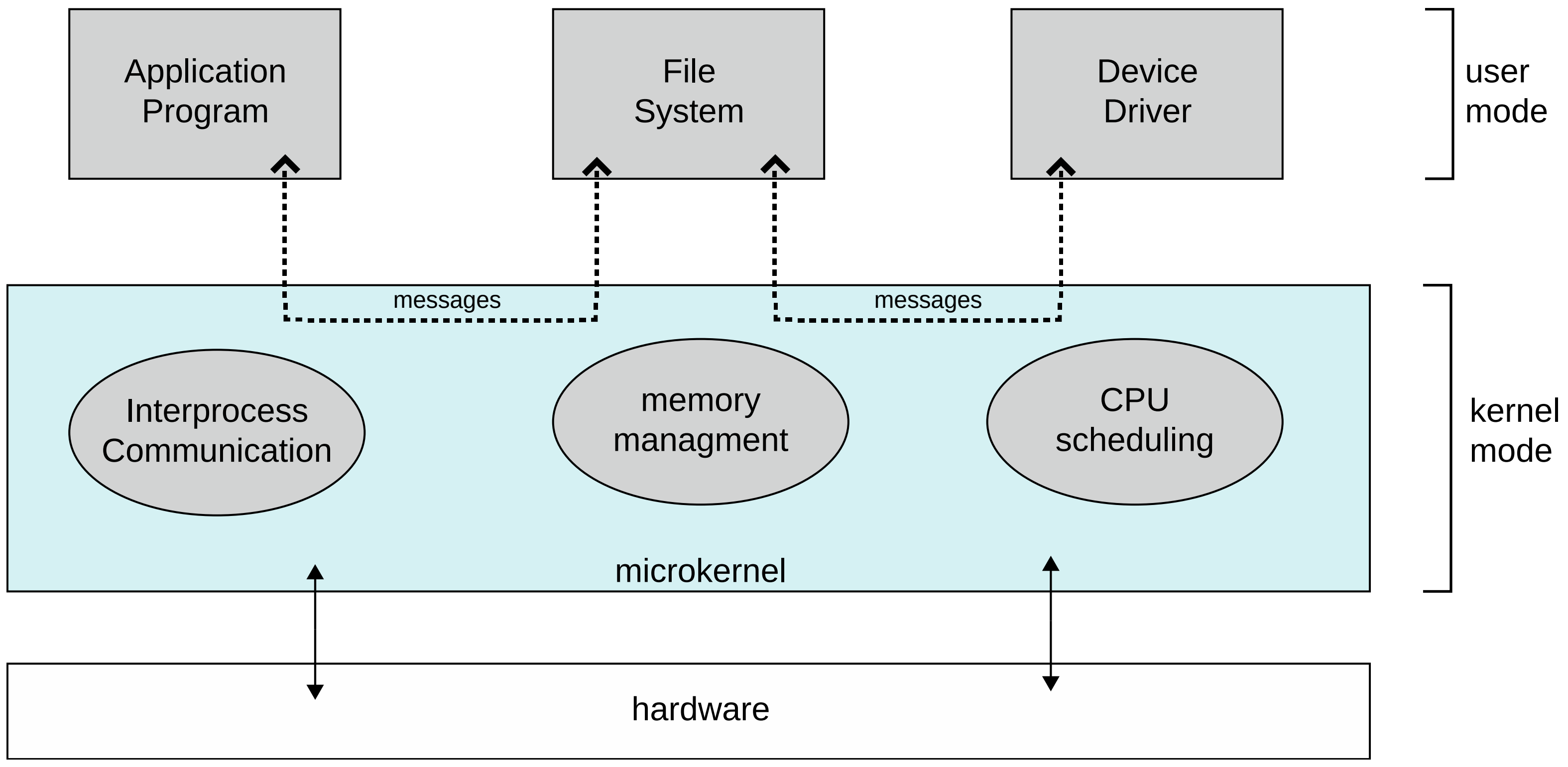
- Moves as much from the kernel into “user” space

- 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 System Structure



Kernel Modules

- Most modern operating systems implement kernel modules
  - ➢ 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