CHITKARA UNIVERSITY

Department of Computer Science & Engineering

Bachelor of Engineering (CSE-AI)

Operating System with Linux

System Calls and Kernel

EXPLORE YOUR POTENTIAL

Contents to Discussed

- System Calls
- Services Provided by System Calls
- Types of System Calls
- Kernal
- Functions of a Kernal
- Types of Kernals

System Calls

- A **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.
- It provides an interface between a process and operating system to allow user-level processes to request services of the operating system.
- All programs needing resources must use system calls.

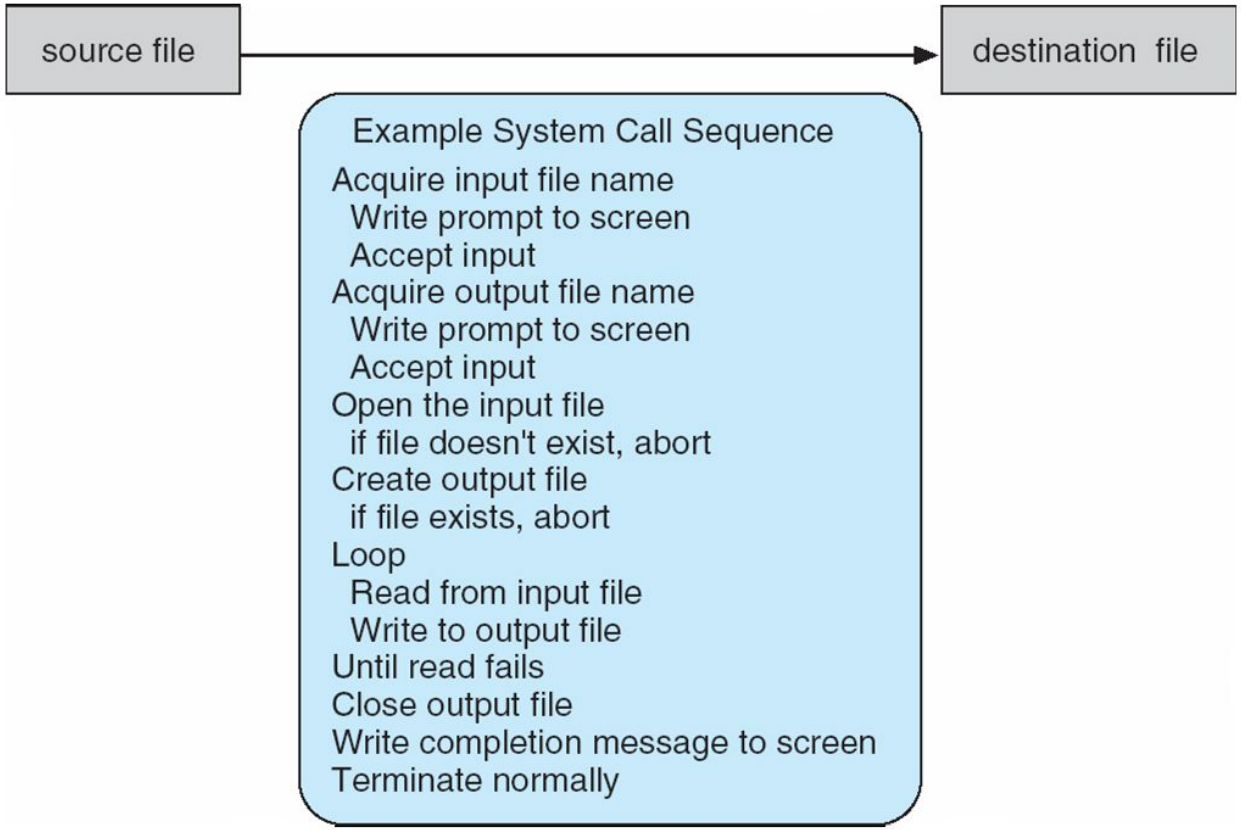
System Calls

- 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), 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



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:

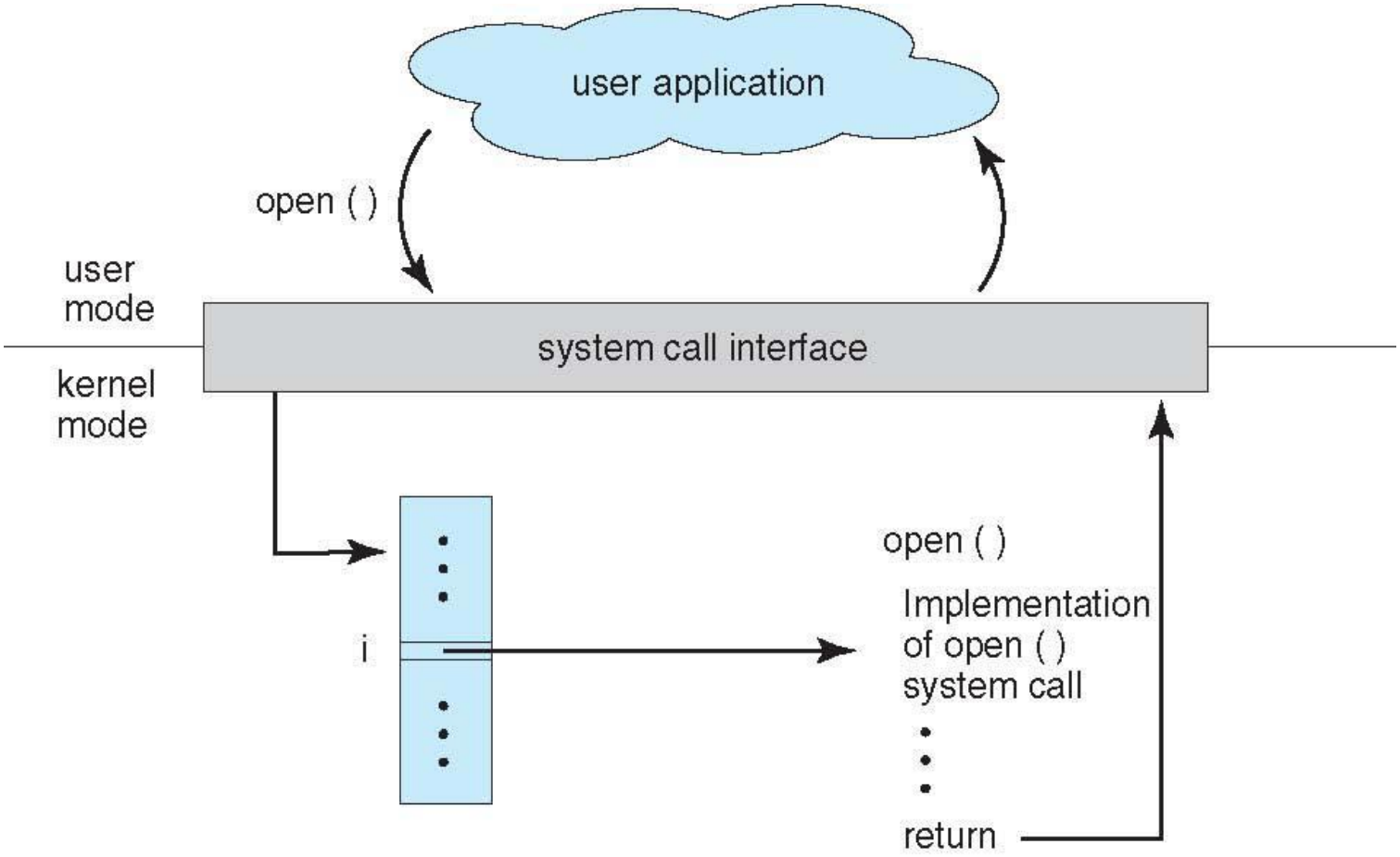
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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
- 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)

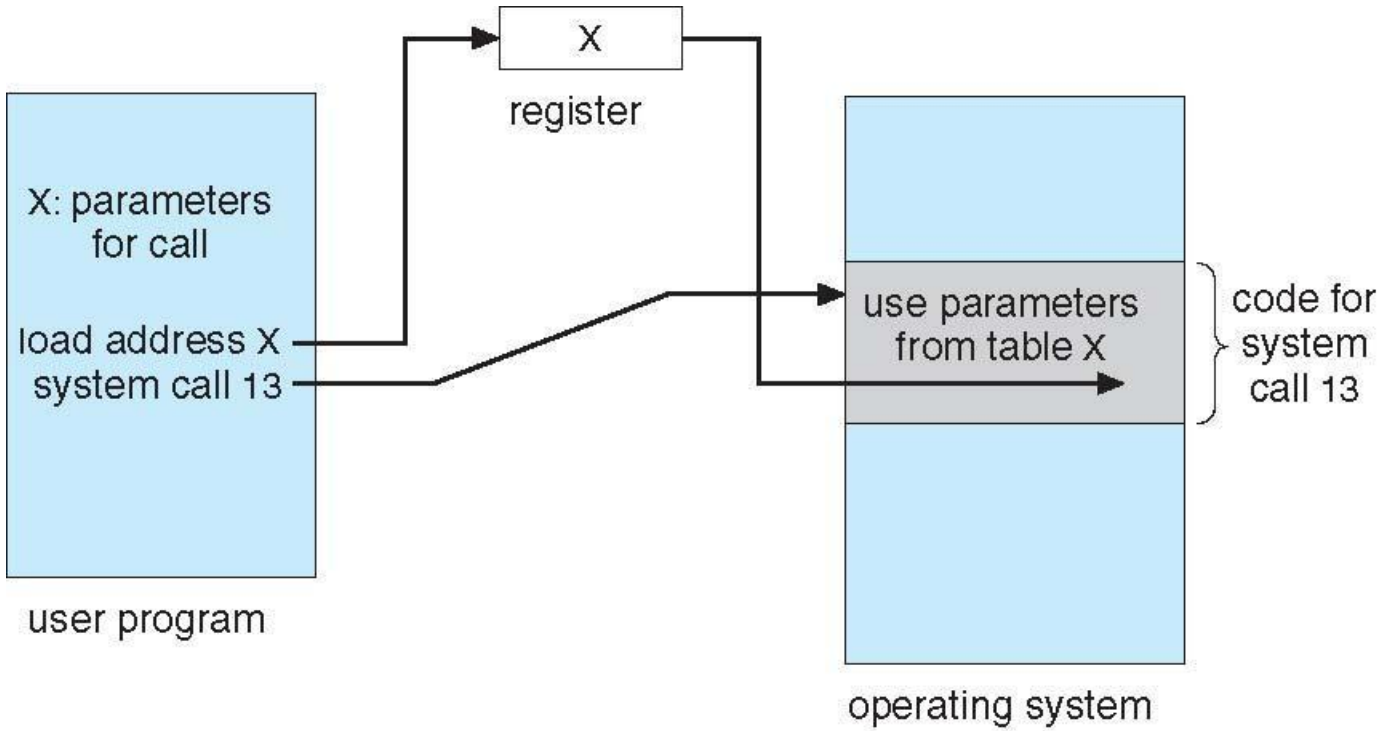
API – System Call – OS Relationship



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

Parameter Passing via Table



Services Provided by System Calls

- Process creation and management
- Main memory management
- File Access, Directory and File system management
- Device handling(I/O)
- Protection
- Networking, etc.

System Calls: Types

- Process control
- File management
- Device management
- Information maintenance
- Communication

Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - 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

- 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

- 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 **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

Types of System Calls

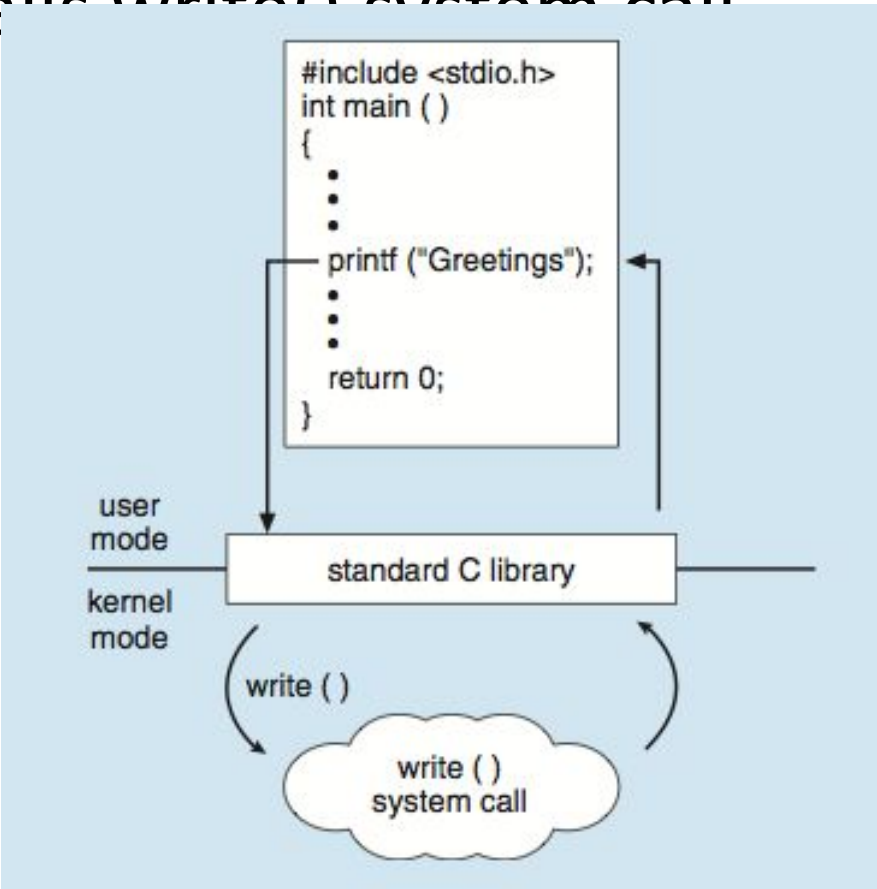
- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Example of System Calls(Windows/ UNIX):

Type of System Calls	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()

Standard C Library Example

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

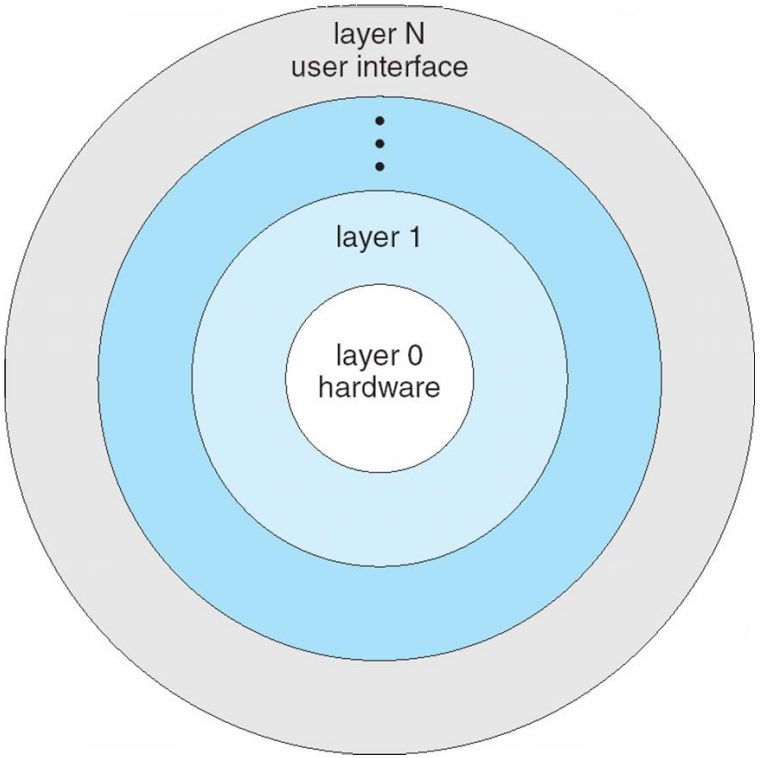


Kernel

- A Kernel is a computer program that has control over everything in the system.
  - Whenever a system starts, the Kernel is the first program that is loaded after the bootloader because the Kernel has to handle the rest of the thing of the system for the Operating System.
  - The Kernel remains in the memory until the Operating System is shut-down.
- The Kernel is responsible for **low-level tasks such as disk management, memory management, task management,** etc.
- It provides an interface between the user and the hardware components of the system.
- Process makes a request to the Kernel through System Call.

Layered Approach of OS

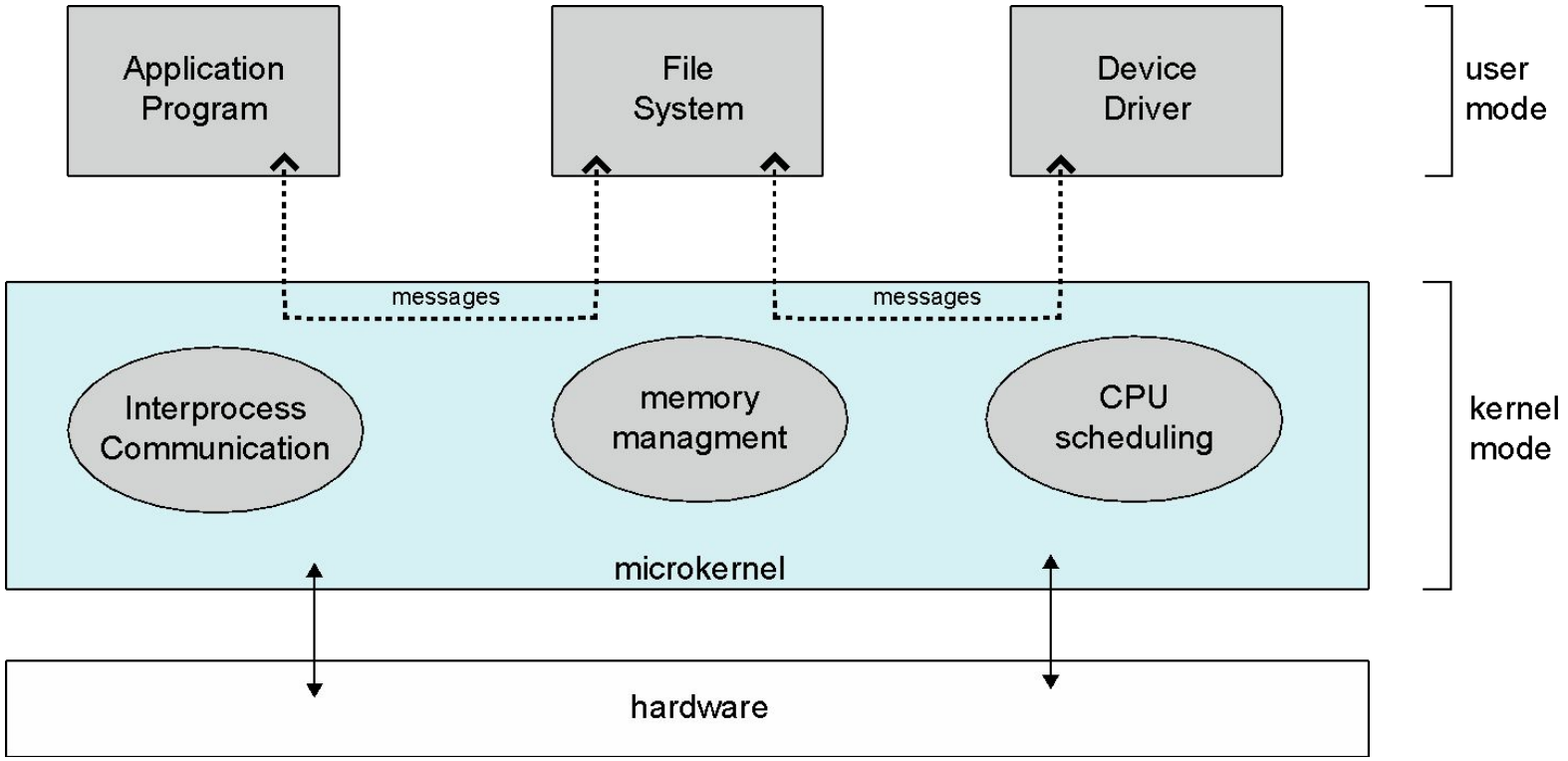
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level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



Microkernel System Structure

- Moves as much from the kernel into user space
- **Mach** 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 System Structure



Kernel

- **Functions of a Kernel:**
  - **Access Computer resource:** A Kernel can access various computer resources like the CPU, I/O devices and other resources. It acts as a bridge between the user and the resources of the system.
  - **Resource Management:** It is the duty of a Kernel to share the resources between various process in such a way that there is uniform access to the resources by every process.
  - **Memory Management:** Every process needs some memory space. So, memory must be allocated and deallocated for its execution. All these memory management is done by a Kernel.
  - **Device Management:** The peripheral devices connected in the system are used by the processes. So, the allocation of these devices is managed by the Kernel.

Kernel

- **Types of Kernels:**
  - **Monolithic Kernels:** User services and the kernel services are implemented in the same memory space
  - **Microkernel:** User services and kernel services are implemented into different spaces
  - **Hybrid Kernel:** A Hybrid Kernel is a combination of both Monolithic Kernel and Microkernel
  - **Nanokernel:** the whole code of the kernel is very small i.e. the code executing in the privileged mode of the hardware is very small
  - **Exokernel:** Resource protection is separated from the management and this, in turn, results in allowing us to perform application-specific customization

References

- Silberschatz, Galvin, Gagne, *Operating System Concepts with Java*, sixth edition.
- Tanenbaum, *Operating Systems: Design and Implementation*
- Peterson and Silberschatz, *Modern Operating Systems*

References

Online Video Link

- https://www.geeksforgeeks.org/operating-systems/
- https://www.geeksforgeeks.org/operating-systems/
- **TEXT BOOKS**

✔ **T1:** Galvin, Peter B.,Silberchatz,A.,"*OperatingSystemConcepts*",AddisonWesley,9th Edition.

✔ **T2:** William Stallings, Operating Systems: Internals and DesignPrinciples,7th edition Pearson Education Limited, 2014ISBN:1292061944, 9781292061948.

REFERENCE BOOKS

✔ **R1:** Andrew Tananbaum, "*Operating System*", PHIL earning.

✔ **R2:** Godbole , Kahate, "*Operating System: A Concept Based Approach*", Tata Mc-Graw-Hill.

THANK YOU