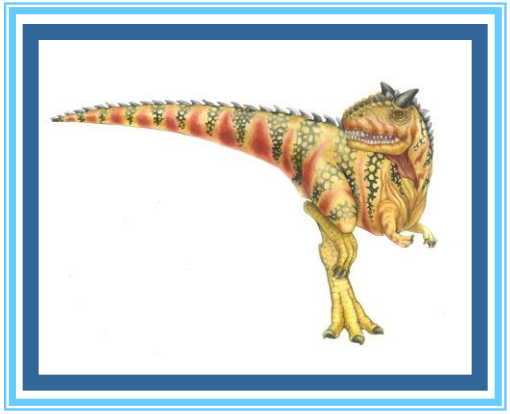
Chapter 2: Operating-System Structures

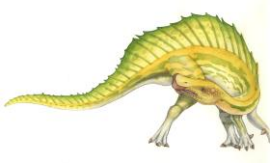


Chapter 2: Operating-System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Operating System Debugging
- Operating System Generation
- System Boot

Objectives

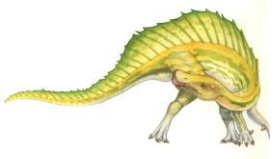
- To describe the services an operating system provides to users, processes, and other systems
- To discuss the various ways of structuring an operating system
- To explain how operating systems are installed and customized and how they boot



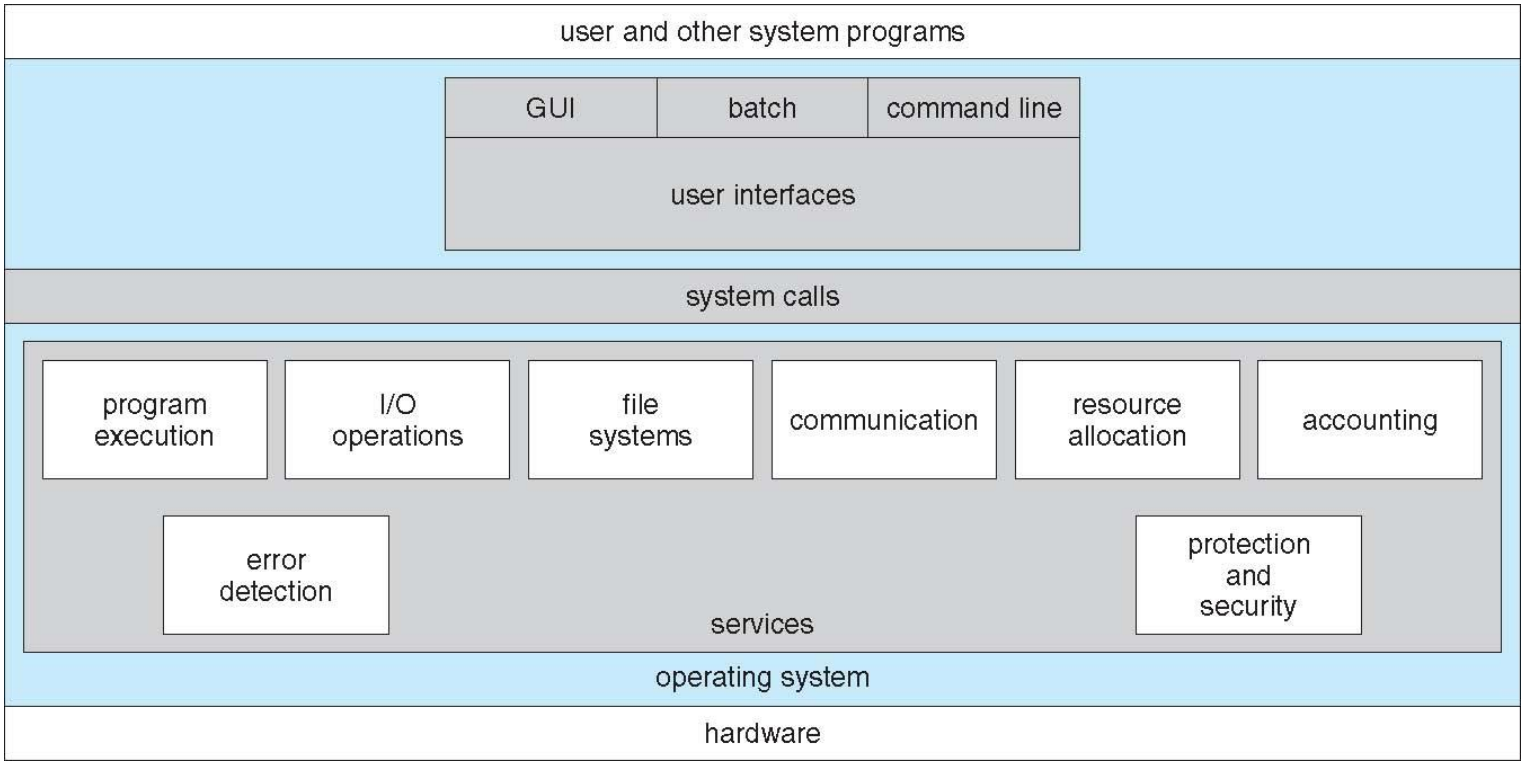
Operating System Services

- 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**).
    - Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **Batch**(commands in files)
  - **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)
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device

    For efficiency and protection, users cannot directly control I/O devices.



A View of Operating System Services

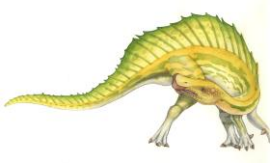


Operating System Services (Cont.)

- **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.
- **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(memory error, power failure), in I/O devices(connection failure in network, paper lack in printer), in user program(arithmetic overflow, access to illegal memory location)
  - 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.)

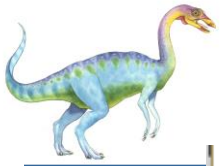
- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources
  - **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



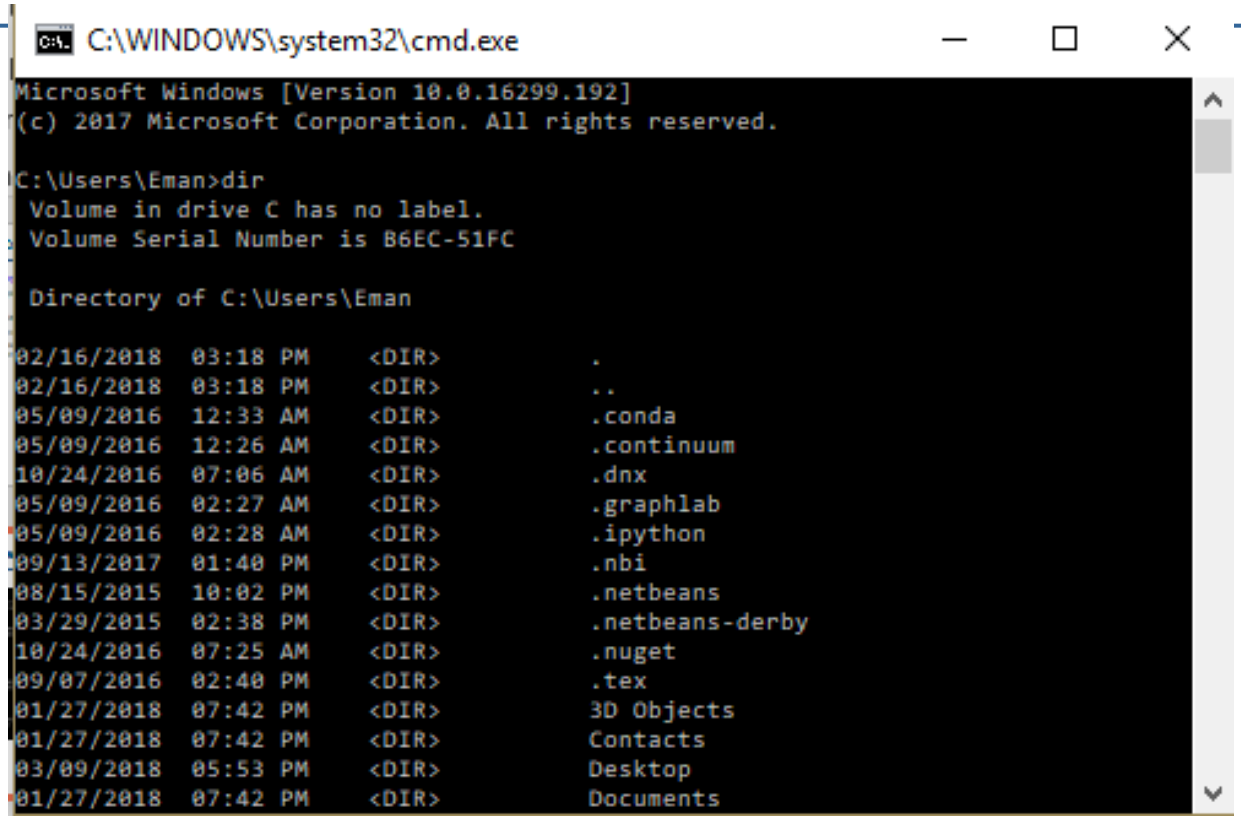
1-User Operating System Interface - CLI

CLI or **command interpreter(shell)** allows direct command entry

- a **shell** is a user interface for access to an operating system's services.
- Sometimes implemented in kernel, sometimes by systems program (Windows and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on (on interactive systems)).
- Sometimes multiple flavors implemented – **shells**
- Most shells provide similar functionality. **https://en.wikipedia.org/wiki/Comparison_of_command_shells**
- Primarily fetches a command from user and executes it
- Many of commands at this level manipulate files: create, delete, list, print, copy, execute etc



windows Shell Command Interpreter

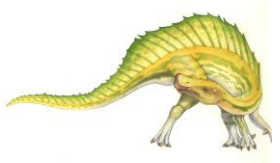


Command Prompt, a CLI shell in Windows

- The command interpreter itself contains the code to execute the command
- Alternative approach: implement most commands through system programs (Used by Unix)
- rm file.txt

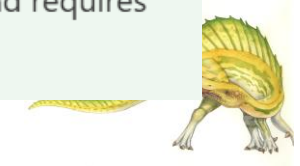
2-User Operating System Interface - GUI

- User-friendly **desktop** metaphor interface
  - Mouse based windows and menu's characterized by a desktop metaphor(Usually mouse, keyboard, and monitor)
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**) or pull down a menu(contains commands)
  - Unix→ Common desktop environment(CDE), X windows System,K desktop Environment(KDE) GNOME are GUI based
  - MAC OS X→ GUI based
  - Microsoft's first version of Windows—Version 1.0—was based on the addition of a GUI interface to the MS-DOS operating system (Later versions→ Cosmetic changes in appearance and functionality).



Operating System Services

Terms	GUI	CUI
Interaction	A user interacts with the computer using Graphics like images, icons.	A user interacts with a computer using commands like text.
Navigation	Navigation is easy.	Navigation is difficult.
Precision	GUI has low precision.	CUI has high precision.
Usage	GUI is easy to use.	CUI is difficult to use and requires expertise.

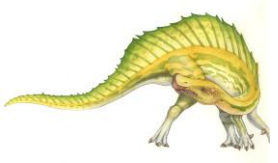


Touchscreen Interfaces

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

Choice of Interface

- The choice of whether to use a command-line or GUI interface is mostly one of personal preference.
- System administrators who manage computers and power users who have deep knowledge of a system frequently use the command-line interface.
  - more efficient, giving them faster access to the activities they need to perform.
  - only a subset of system functions is available in some systems via the GUI, leaving the less common tasks to those who are command-line knowledgeable .
  - Command line interfaces usually make repetitive tasks easier, in part because they have their own programmability.



The Mac OS X GUI

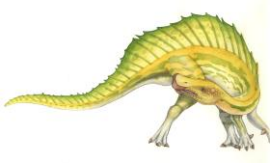
- Mac OS has not provided a command-line interface.
- Mac OS X (which is in part implemented using a UNIX kernel), the operating system now provides both a Aqua interface and a command-line interface

System Calls

- Programming interface to the services provided by the OS
- System calls are the interface through which an application program interacts with the operating system kernel.
- They are functions or routines provided by the operating system that allow user-level processes or applications to request services from the operating system.
- Typically written in a high-level language (C or C++)
- Systems can execute thousands of system calls per second
  - writing a simple program to read data from one file and copy them to another file.
- Programmers never see this level of detail

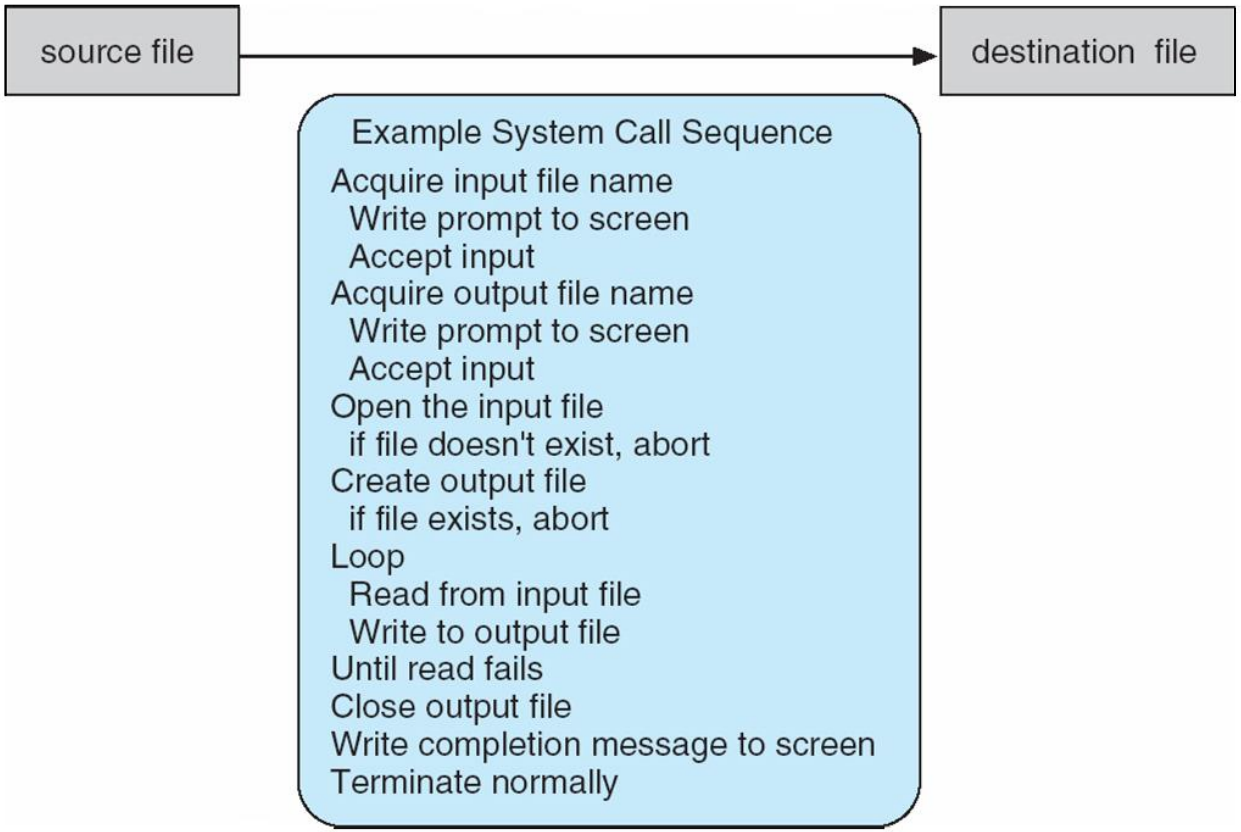
System Calls

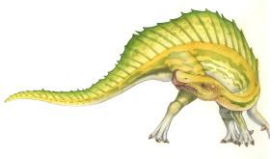
- Designs programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- The API specifies a set of functions available to an application programmer(including the parameters passed to the programs and return values)
- Three most common APIs are Win32 API for 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)
- A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called **libc**.
- Behind the scene→ function typically invoke actual system call on behalf of application programmer.



Example of System Calls

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





System Calls

- Createprocess() function of win32 API actually calls NTCreateProces()
- Why application programmer prefer programming according to API rather than direct system call?
  - Program portability
  - Actual system calls are more detailed and difficult to work with than an API available.

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

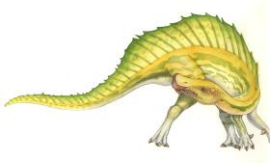
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `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

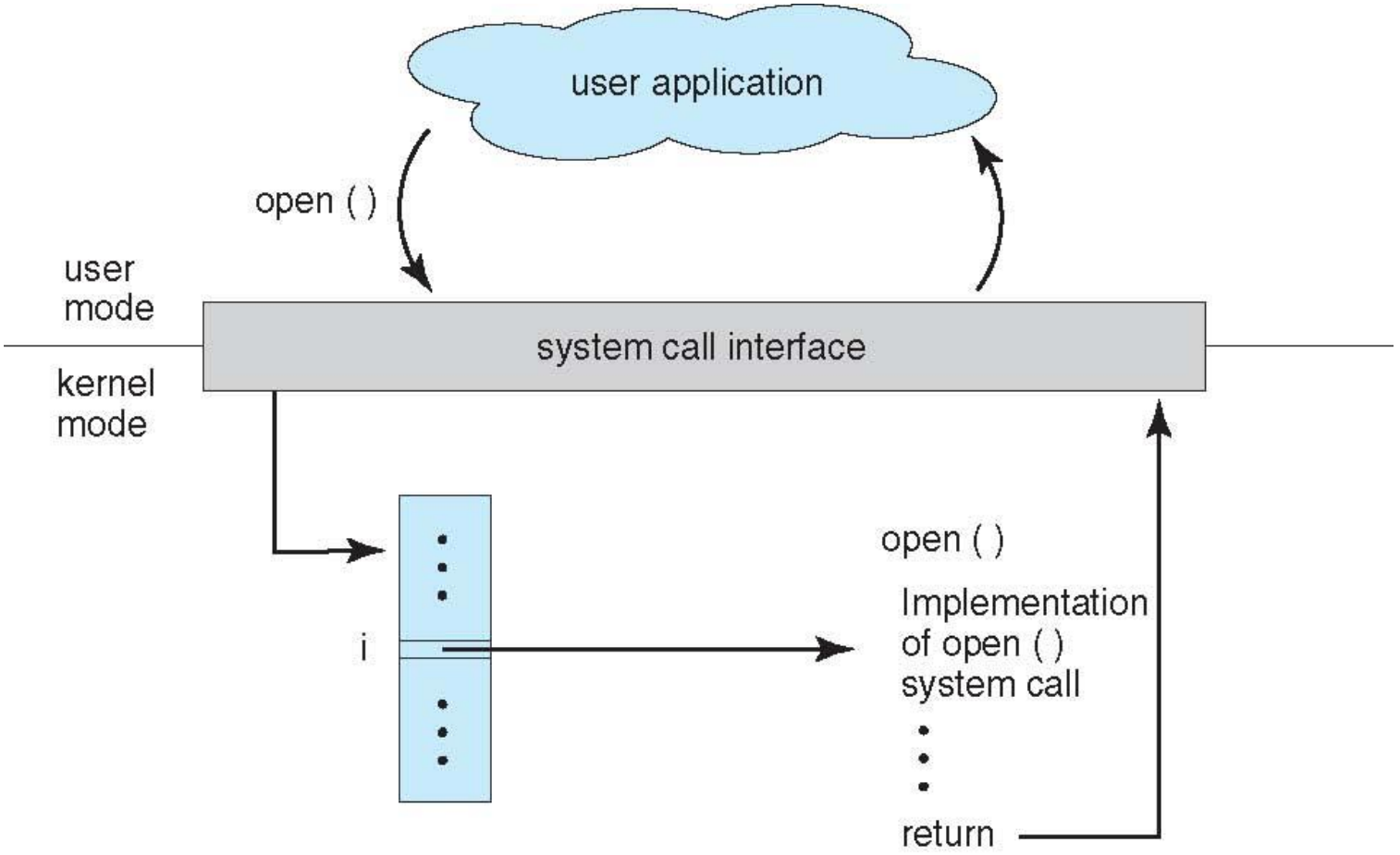
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

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)



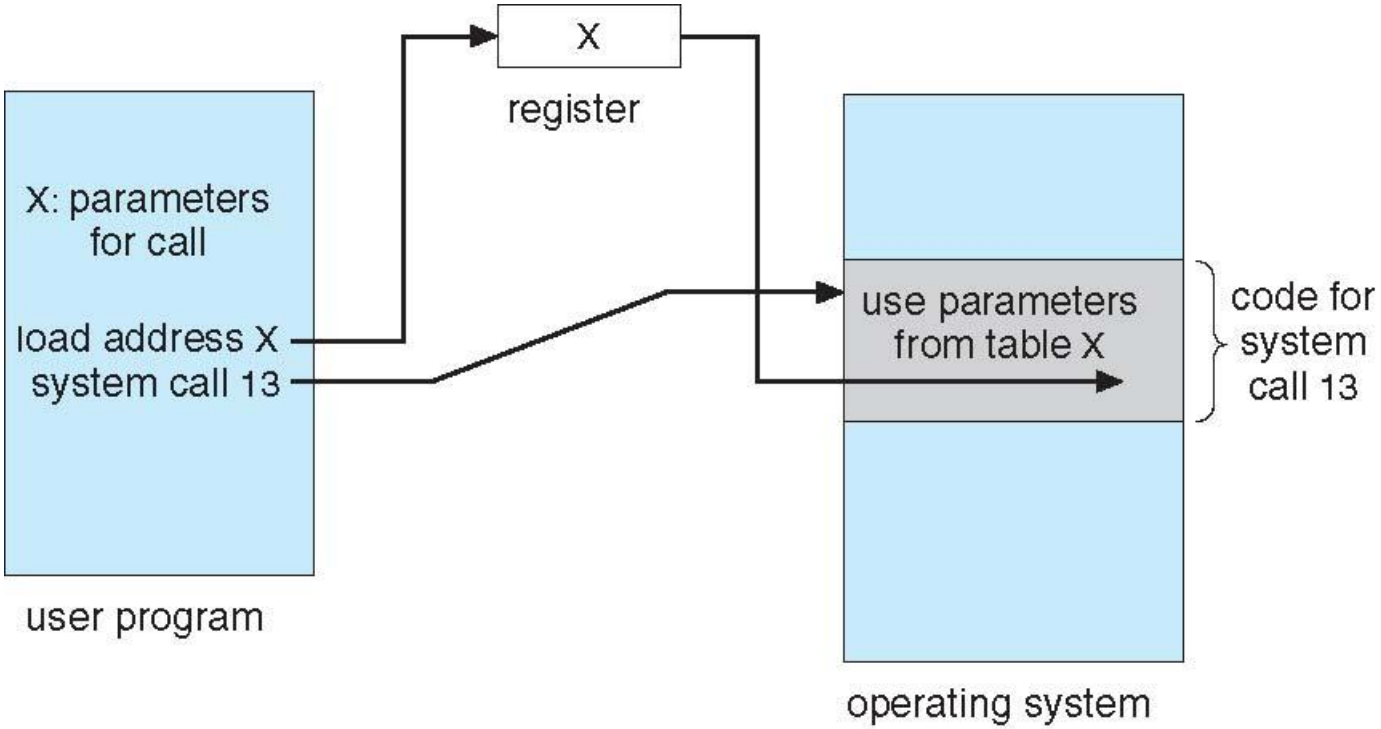
API – System Call – OS Relationship

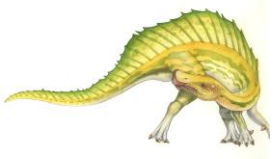


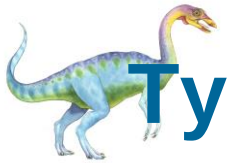
System Call Parameter Passing

- 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



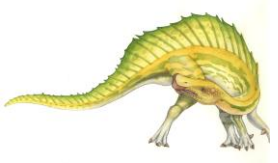


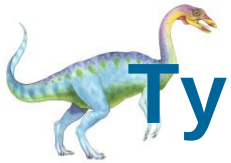


Types of System Calls(six categories)

System calls can be grouped roughly into six major categories:

- **Process control**
- **file manipulation**
- **device manipulation**
- **information maintenance**
  communications
- **protection**





Types of System Calls(six categories)

1-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(shared data.).
  - acquire lock() and release lock().

Types of System Calls(six categories) Process Control example

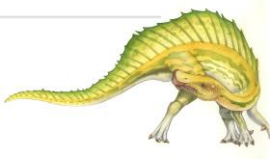
Functionality	Linux System Call / API	Windows System Call / API	Example
Create Process	`fork()`, `exec()`	`CreateProcess()`	Linux: `pid_t pid = fork();` Windows: `CreateProcess(NULL, ...)`
Terminate Process	`exit()`, `kill()`	`ExitProcess()`, `TerminateProcess()`	Linux: `exit(0);` Windows: `ExitProcess(0);`
End / Abort	`abort()`	`TerminateProcess()`	Linux: `abort();` Windows: `TerminateProcess(hProcess, 1);`
Load / Execute	`execve()`, `system()`	`CreateProcess()`	Linux: `execvp("ls", args);` Windows: Part of `CreateProcess()`
Get / Set Process Attributes	`getpid()`, `getppid()`, `setpgid()`	`GetProcessId()`, `SetPriorityClass()`	Linux: `pid_t id = getpid();` Windows: `GetProcessId(hProcess);`
Wait for Time	`sleep()`, `usleep()`, `nanosleep()`	`Sleep()`	Linux: `sleep(1);` Windows: `Sleep(1000);`
Wait / Signal Event	`wait()`, `signal()`, `sem_wait()`	`WaitForSingleObject()`, `SetEvent()`	Linux: `wait(NULL);` Windows: `WaitForSingleObject(hEvent, INFINITE);`
Allocate / Free Memory	`mmap()`, `munmap()`, `malloc()/free()`	`Virtual...loc()`, `VirtualFree()`	Linux: `mmap(NULL, size, PROT_READ

Types of System Calls

2-File management

- create file, delete file
- open, close file
- read, write, reposition
- get and set file attributes

Operation	Windows Command	Linux Command
Create File	`echo "Hello World" > example.txt`	`echo "Hello World" > example.txt`
Delete File	`del example.txt`	`rm example.txt`
Open File	`std::ofstream file("example.txt");` (C++)	`FILE *file = fopen("example.txt", "w");` (C)
Close File	`file.close();` (C++)	`fclose(file);` (C)
Read File	`std::ifstream file("example.txt");` (C++)	`fgets(buffer, 100, file);` (C)

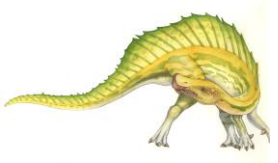


Types of System Calls

3-Device management

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

Operation	Windows System Call	Linux System Call
Request Device	`CreateFile()` (for accessing a device)	`open()` (for opening a device file)
Release Device	`CloseHandle()` (for releasing a device)	`close()` (for releasing a device file)
Read from Device	`ReadFile()`	`read()` (for reading from a device file)



Types of System Calls (Cont.)

4-Information maintenance

- get time or date, set time or date
- get system data, set system data
- get and set process, file, or device attributes

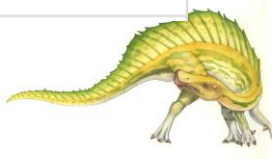
Operation	Windows System Call	Linux System Call
Get Time or Date	`GetSystemTime()` (UTC time) / `GetLocalTime()` (local time)	`time()` (current time) / `localtime()` (local time)
Set Time or Date	`SetSystemTime()` (UTC time) / `SetLocalTime()` (local time)	`settimeofday()` (set system time)
Get System Data	`GetSystemInfo()` (system configuration)	`uname()` (system information)
Set System Data	`SetSystemInfo()` (changes to system config)	No direct equivalent for setting system data
Get Process Attributes	`GetProcessId()` / `OpenProcess()` (to access process info)	`getpid()` (process ID) / `ps` (process attributes)
Set Process Attributes	`SetPriorityClass()` (to set process priority)	`nice()` (to change process priority)
Get File Attributes	`GetFileAttributes()` (to retrieve file attributes)	`stat()` (to retrieve file attributes)

Types of System Calls (Cont.)

5-Communications

- create, delete communication connection
- send, receive messages
- transfer status information
- attach and detach remote devices

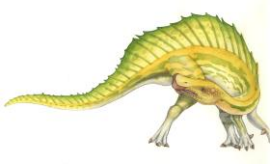
Operation	Windows System Call	Linux System Call
Create Communication Connection	`CreateFile()` (for opening communication port or socket)	`socket()` (for creating a socket)
Delete Communication Connection	`CloseHandle()` (for closing communication handle)	`close()` (for closing socket or communication channel)
Send Messages	`WriteFile()` (for sending data through communication port)	`send()` (for sending data through a socket)
Receive Messages	`ReadFile()` (for receiving data from communication port)	`recv()` (for receiving data through a socket)



Types of System Calls (Cont.)

5-Communications

- Two mechanisms
- **1: Message passing** Model: messages can be exchanged between processes either directly or indirectly through a common mailbox.
  - A connection must be opened.
  - Name of communicators must be known, be it another process on same computer or a process on another computer connected through network.
  - Host name, network identifier(IP)(Hostid and process id by system calls) These identifiers then passed to open and close calls.
    - Get_hostid(), get_processid() do this translation
    - Identifiers are passed to open() and close() calls
    - Or to open_connection and close_connection system call.
  - The recipient process must gives its permission (accept_connection)



Types of System Calls (Cont.)

5-Communications(contd..)

read message() and write message() system calls.

The close connection() call terminates the communication.

2: Shared Memory Model:

- Processes used shared_memory_create() and shared_memory_ attach() system call to create and gain access to regions of memory owned by other processes.
- Shared memory requires that two processes agrees to remove the restriction of not sharing each other memory.
- Read and write data in the shared area.(not under the control of OS… determined by processes)
- These processes are responsible for not overwriting the same location simultaneously.

Types of System Calls (Cont.)

5-Communications(contd..)

- Message passing is useful for exchanging small amount of data.
  - Easier to implement
- Shared memory allows maximum speed and convenience of communication(can be done at memory transfer speed)
- Problems exist, however, in the areas of protection and synchronization between the processes sharing memory

Types of System Calls (Cont.)

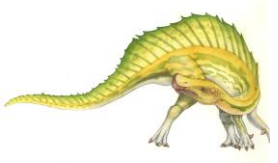
5-Communications(contd..)

Protection Problems

Shared memory allows multiple processes to access the same memory space. Without protection mechanisms:

- **Unauthorized Access**: Any process might read or write to memory it shouldn't, leading to data breaches or corruption.
- **Memory Overwrites**: One process could accidentally or maliciously overwrite data being used by another.
- **Segmentation Faults**: Improper access (like accessing deallocated memory) can cause crashes.

n 👉 **Solution**: Use memory protection features (like page permissions), operating system-level access controls, or memory sandboxing.



Types of System Calls (Cont.)

5-Communications(contd..) Synchronization Problems:

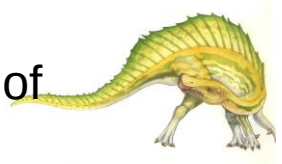
Since multiple processes are accessing the same data, coordination is crucial. Without proper synchronization:

- **Race Conditions**: Two processes may try to read and write at the same time, causing unpredictable behavior.
- **Deadlocks**: Processes waiting on each other's actions might get stuck forever.
- **Inconsistency**: Partial updates by one process could be read by another, resulting in corrupt or incorrect data.

👉 **Solution**: Use synchronization primitives like:

- **Semaphores**
- **Mutexes**
- **Locks**
- **Monitors**

n These ensure that only one process can access critical sections of memory at a time.



Operating System Design and Implementation

- 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
  - **Choice of Hardware:** CPU Architecture (x86, ARM, RISC-V), Number of CPUs/Cores, Memory Capacity and Type, **I/O Devices** (disks, keyboards, GPUs), **Power Constraints** (mobile devices)
  - **Type of Systems:** Batch System, Time-sharing System, Real Time systems, Embedded System, Mobile OS, Distributed OS
- The requirements can, however, be divided into two basic groups: **user goals** and **system goals**.
- **User** goals and **System** goals
  - User goals – operating system should be **convenient to use**, easy to learn, reliable, safe, and **fast** (these specifications are not particularly useful in the system design)
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, **reliable, error-free, and efficient**

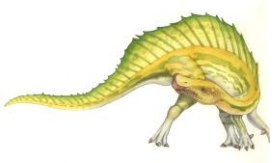
Operating System Design and Implementation (Cont.)

- Important principle to separate

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

  For example, the timer construct is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.

- 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 (example – timer)
  - each change in policy would require a change in the underlying mechanism



Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like 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

Reading Assignment:

Difference between high level and low level programming languages

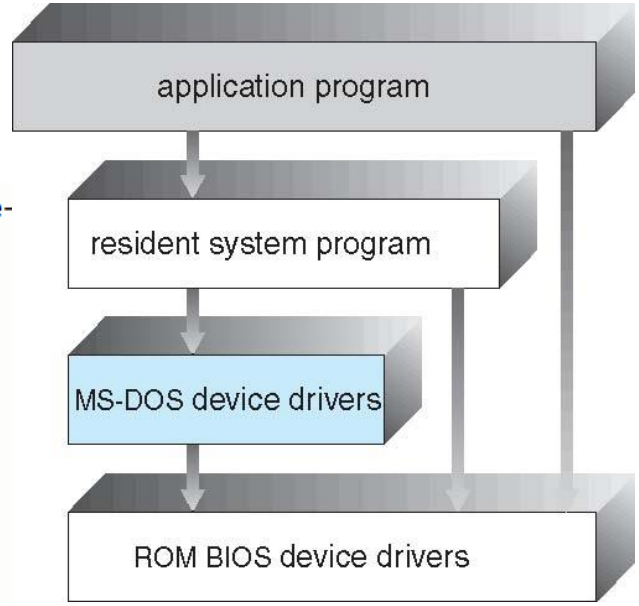
- windows: C++
- linux: C
- mac: Objective C
- android: JAVA, C, C++
- Solaris: C, C++
- iOS 7: Objective-C,Swift,C, C++

Operating System Structure

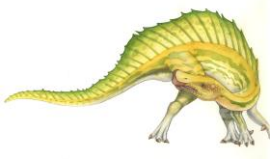
- General-purpose OS is very large program
  - Partition the task into smaller components rather than monolithic system
  - These modules should be well defined(defined inputs, outputs and functions)
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex -- UNIX
  - Layered – an abstrcation
  - Microkernel –Mach
  - Hybrid

Simple Structure -- MS-DOS

- MS-DOS – written to provide the most functionality in the least space



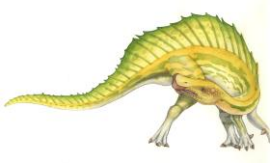
- **Definition**: MS-DOS (Microsoft Disk Operating System) is a single-user, single-tasking OS for x86 PCs, popular in the 1980s.
- **Structure**: Simple, non-layered, monolithic-like.
  - No clear separation between user and kernel modes.
  - Applications directly access hardware (e.g., I/O routines).
- **Characteristics**:
  - **Pros**: Easy to develop, fast due to minimal overhead, suitable for resource-constrained systems.
  - **Cons**: Crash-prone (one program failure crashes the system), low abstraction, security risks (direct hardware access).
- **Example**: Running a game in MS-DOS directly controls graphics hardware.
- **Complexity**: Minimal, no advanced time complexity concerns (operations often (O(1)) or (O(n))).
- **Use Case**: Early IBM PCs, basic file management, and games.



Non Simple Structure -- UNIX

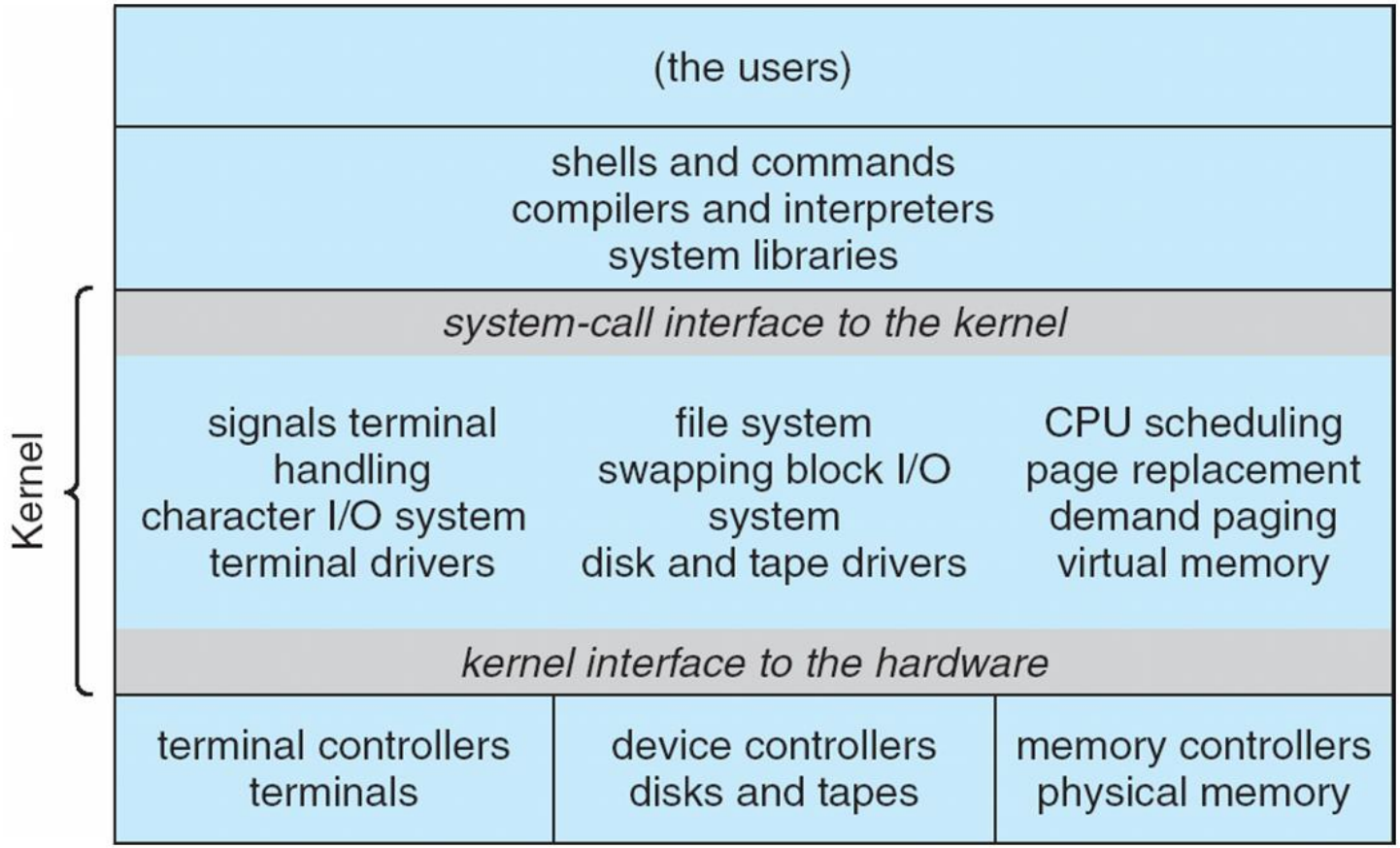
UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts

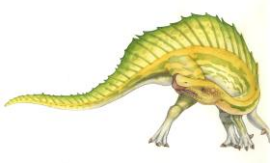
- Systems programs
- The kernel
  - Consists of everything below the system-call interface and above the physical hardware
  - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level
  - almost all the OS was in one big layer, not really breaking the OS down into layered subsystems
  - Monolithic approach



Traditional UNIX System Structure

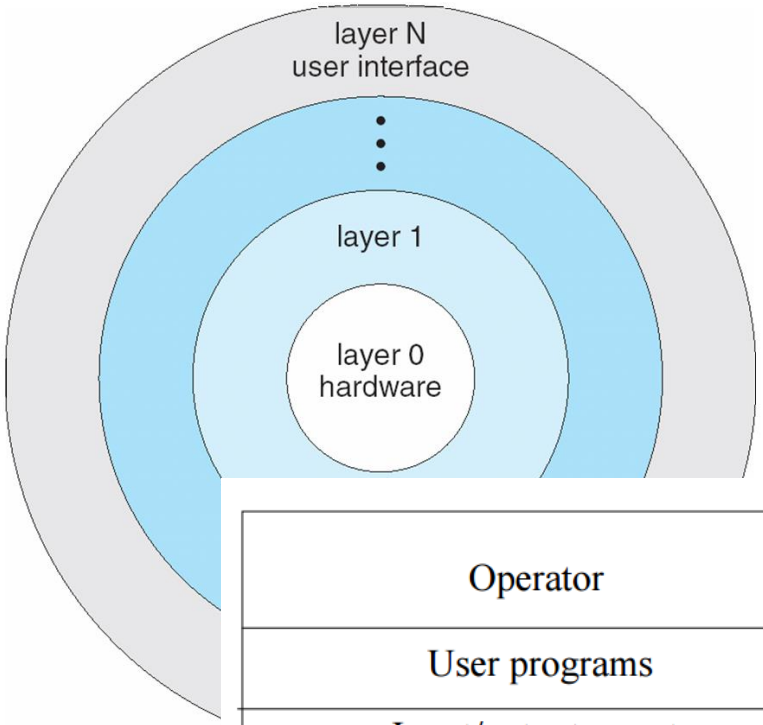
Beyond simple but not fully layered





Layered Approach

- The operating system is divided into a number of layers (levels), each built or 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
- The main *advantage* is **simplicity** of **construction** and **debugging**.
- The main *difficulty* is defining the various layers.
- The main *disadvantage* is that the OS tends to be less efficient than other implementations.
  - Each layer adds overhead to the system call; the net result is a system call that takes longer than does one on a non-layered system

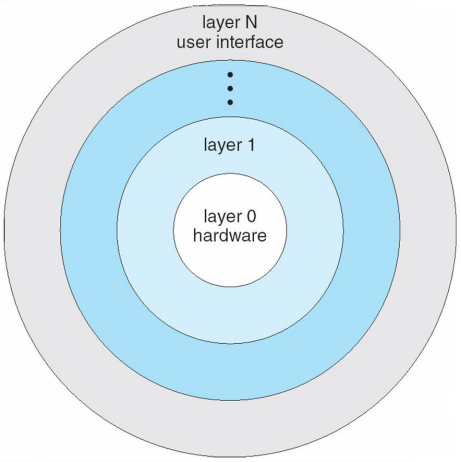


Operator
User programs
Input/output mgmt.
Operator-process com.
Memory mgmt.
Processor allocation and multiprog.

Layered Approach

Structure:

- Bottom layer (Layer 0): Hardware.
- Top layer (Layer N): User interface.
- Each layer uses services of the layer below it.



Characteristics:

- **Pros**: Simplifies design, enhances modularity, easier to maintain and debug.
- **Cons**: Overhead from layer communication, slower system calls, complex layer design.

Example: THE Operating System (1960s) with 6 layers (hardware to user programs).

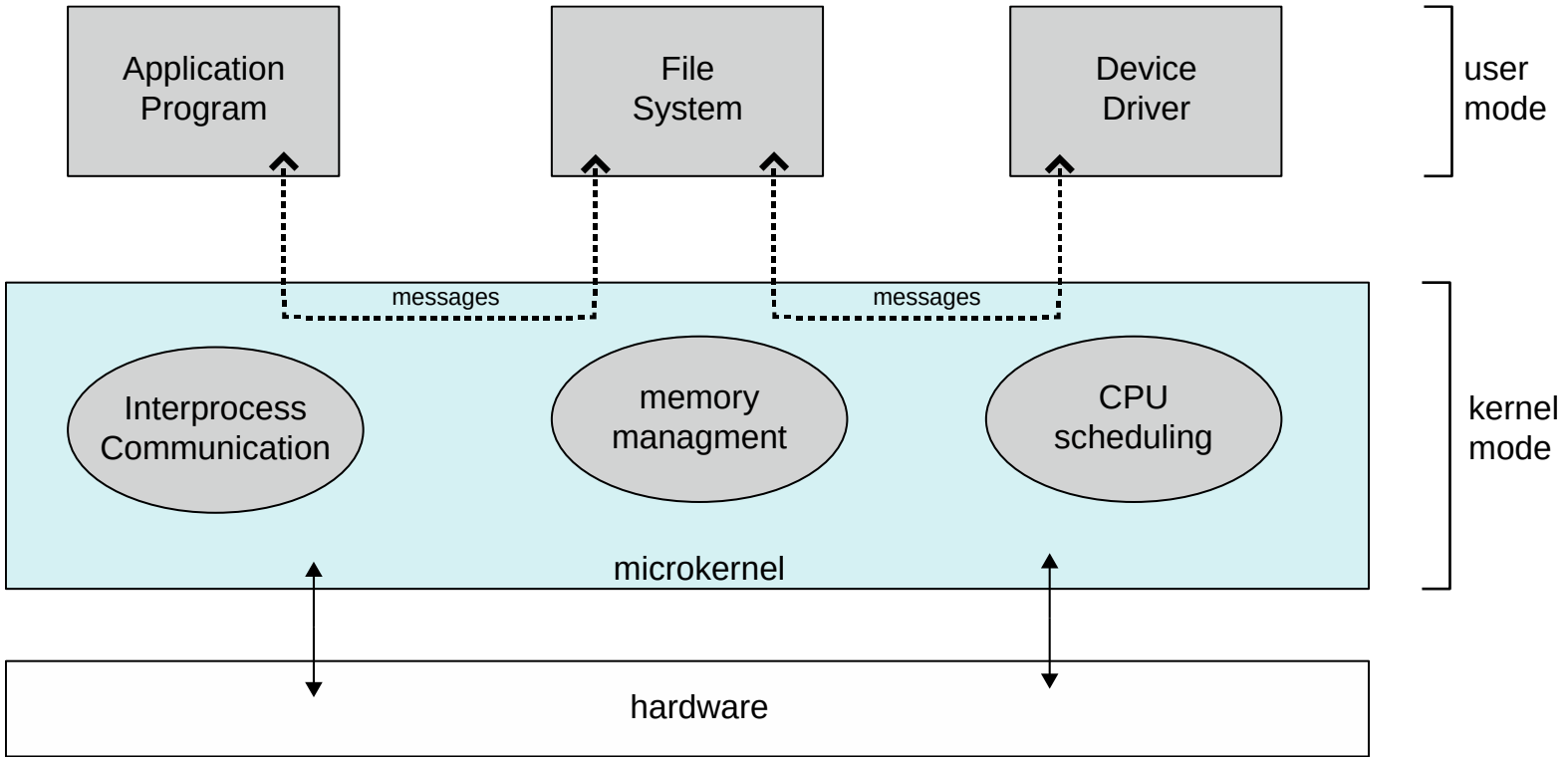
Complexity: System calls may involve (O(n)) or (O(\log n)) operations per layer, accumulating overhead.

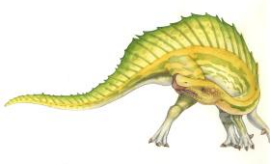
Use Case: Systems requiring clear separation, like early UNIX prototypes.

Microkernel System Structure

- **Definition**: Minimal kernel runs in privileged mode, with most services (e.g., file systems, drivers) in user space.
- **Structure**:
  - Core kernel: Handles basic tasks (e.g., IPC, scheduling, memory management).
  - Services communicate via message passing.
- **Characteristics**:
  - **Pros**: Modular, secure (services isolated), reliable (failure in one service doesn't crash system), portable.
  - **Cons**: Performance overhead from message passing, complex IPC (potentially $O(n)$ or $O(\log n)$).
- **Example**: Mach (used in macOS), Minix (educational OS).
- **Complexity**: Message passing adds latency, often $O(\log n)$ for scheduling or IPC.
- **Use Case**: Embedded systems, high-reliability environments like QNX.

Microkernel System Structure





Modules

Definition: OS with a core kernel that supports dynamically loadable kernel modules (LKMs) for additional functionality.

Structure:

- Core kernel: Basic services (e.g., memory, process management).
- Modules: Loadable at boot or runtime (e.g., device drivers).
- Modules can call each other directly, unlike layered systems.

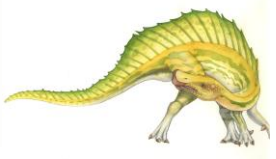
Characteristics:

- **Pros**: Flexible, modular, efficient (no strict layering), easy to extend.
- **Cons**: Less isolation than microkernel, potential stability issues if modules f

Example: Linux (loads drivers like USB as modules), Solaris.

Complexity: Module loading/unloading is typically (O(1)), but module interact may involve (O(n)) or (O(\log n)).

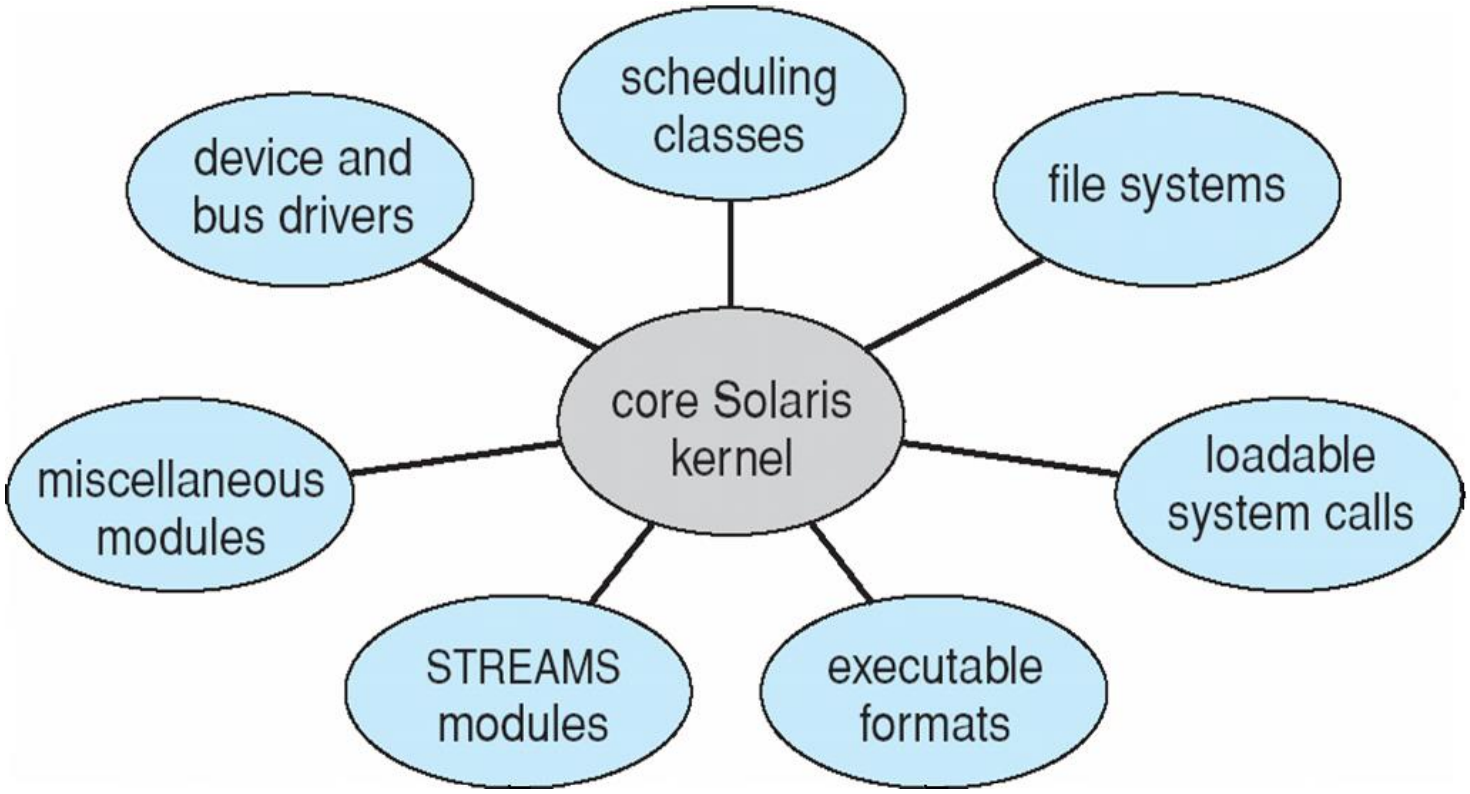
Use Case: Modern OSes needing hardware adaptability (e.g., Linux servers).

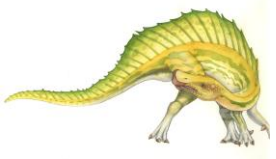


Modules

- How Modular approach is different from Microkernel?
  - **Core Kernel:** Handles essential tasks but is larger than a microkernel, including some built-in services.
  - **Loadable Modules:** Modules (e.g., drivers, network protocols) are loaded into kernel space, sharing the same privilege level as the core kernel.
  - **Direct Interaction:** Modules can call each other or the core kernel directly, avoiding message-passing overhead.

Solaris Modular Approach

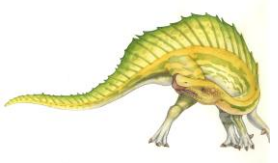




Hybrid structure

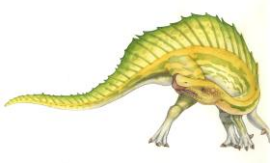
- A **hybrid kernel** is an operating system kernel architecture that attempts to combine aspects and benefits of microkernel and monolithic kernel architectures used in computer OS.
- The idea behind a hybrid kernel is to have a kernel structure similar to that of a microkernel, but to implement that structure in the manner of a monolithic kernel.

- Prominent example of a hybrid kernel is the Microsoft Windows NT kernel.
- All operating systems in the Windows NT family, up to and including Windows 10 and Windows Server 2016, and powers Windows Phone 8, Windows Phone 8.1 use hybrid kernel.

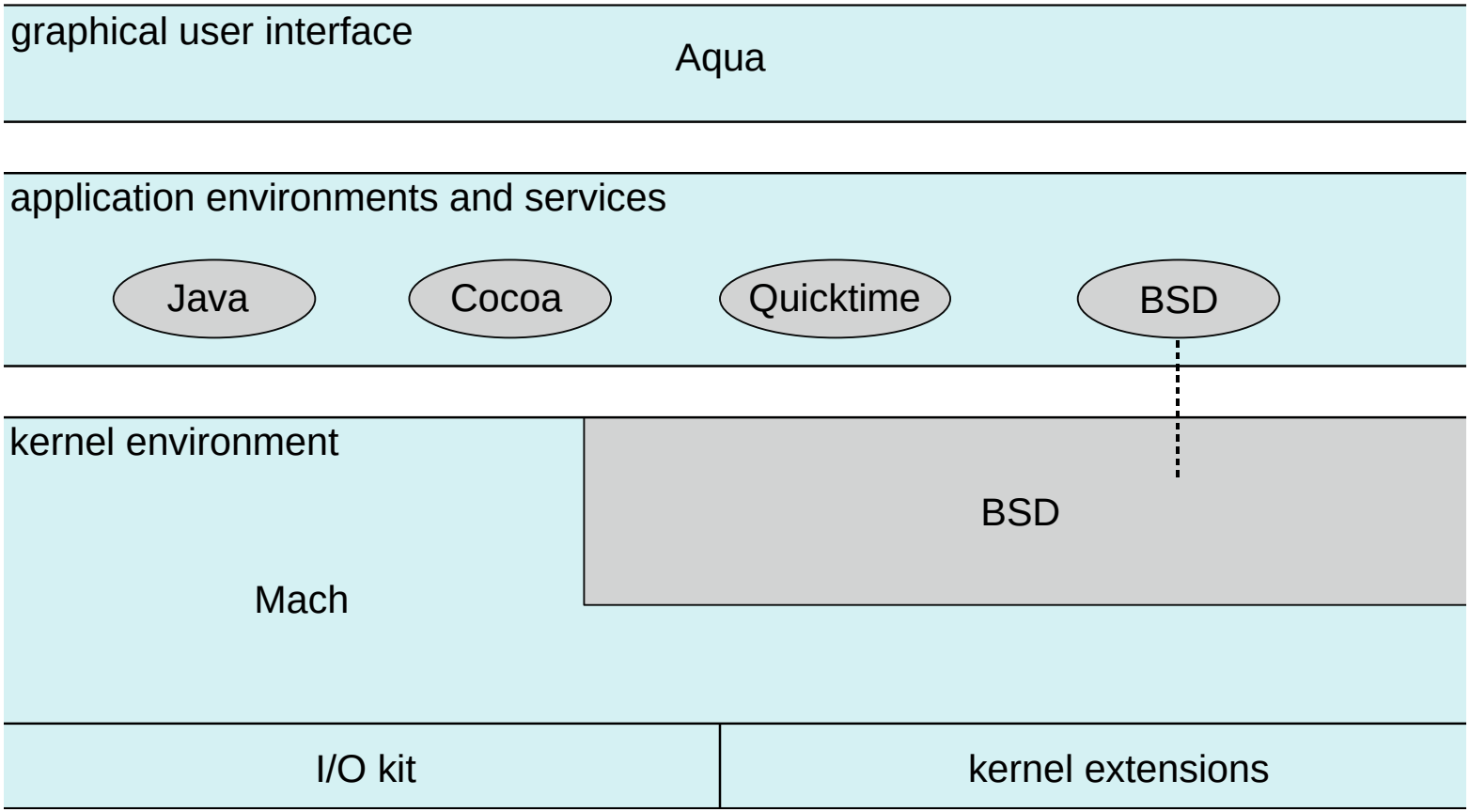


Hybrid Systems

- Most modern operating systems are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels 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**)



Mac OS X Structure



End of Chapter 2