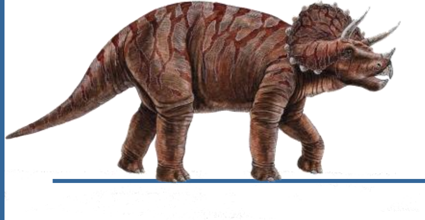
Chapter 2: Operating-System Services

Operating system concepts / Abraham Silberschatz,

Edited by : Dr. Mustafa Shiple

Outline

- System Calls
- System Services
- Linkers and Loaders
- Why Applications are Operating System Specific
- Design and Implementation
- Operating System Structure
- Building and Booting an Operating System
- Operating System Debugging

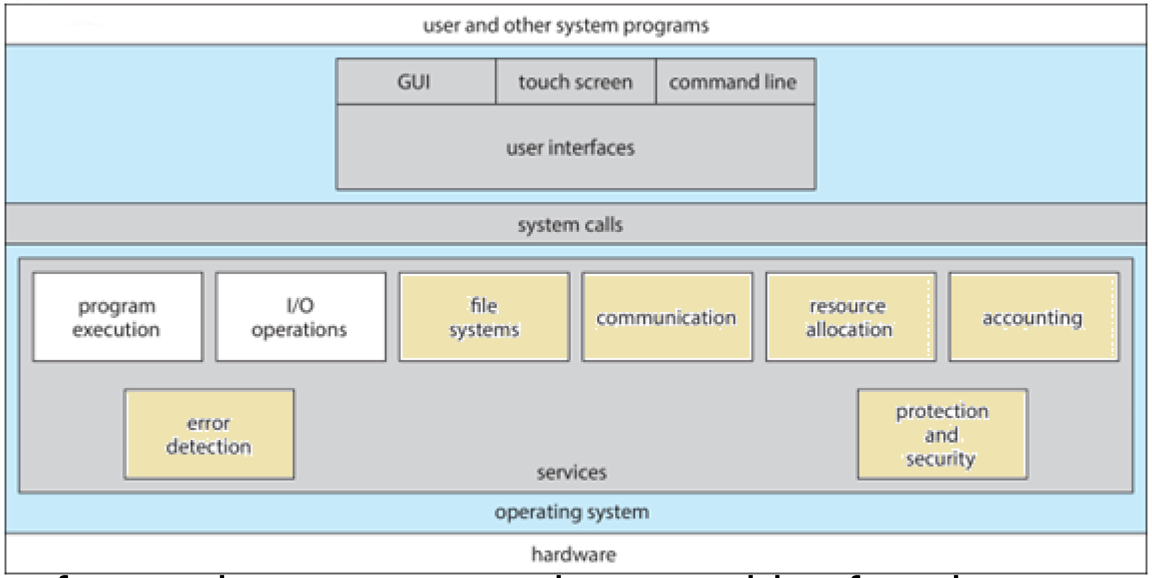


Objectives

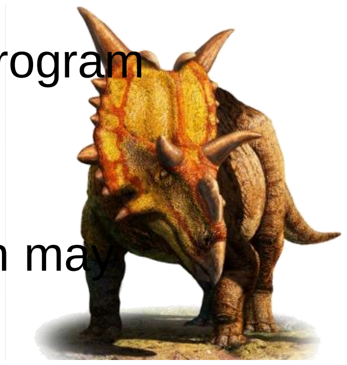
- Identify services provided by an operating system
- Illustrate how system calls are used to provide operating system services
- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems
- Illustrate the process for booting an operating system
- Apply tools for monitoring operating system performance
- Design and implement kernel modules for interacting with a Linux kernel

Operating System Services

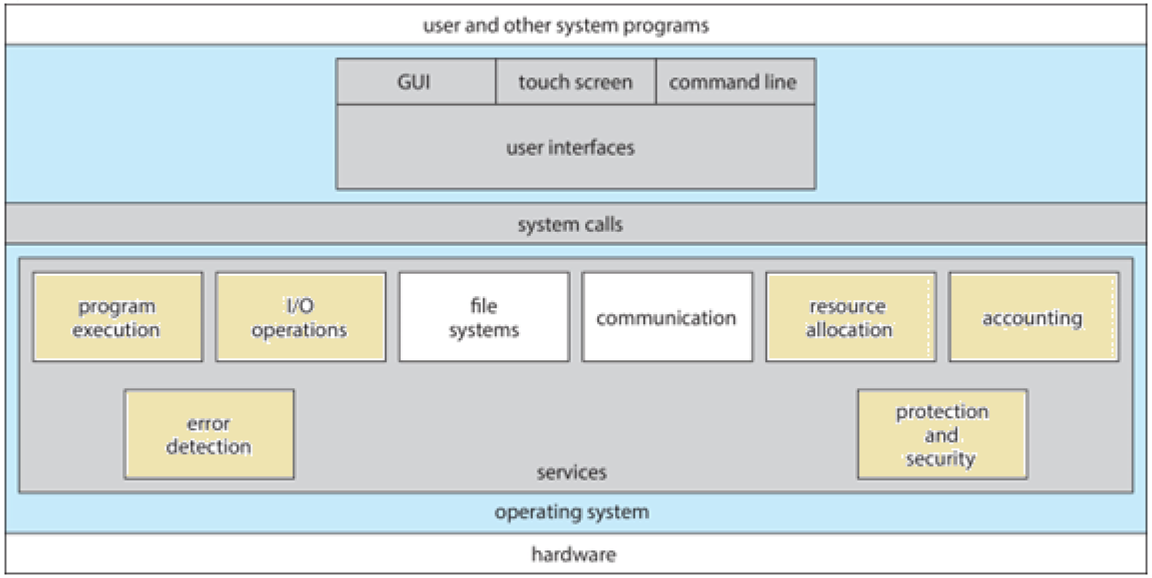
A View of Operating System Services



- One set of operating-system services provides functions:
  - **User interface** - Almost all operating systems have a user interface (**UI**).
    - Varies between **Command-Line** (**CLI**), **Graphics User Interface** (**GUI**), **touch-screen**, **Batch**
  - **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

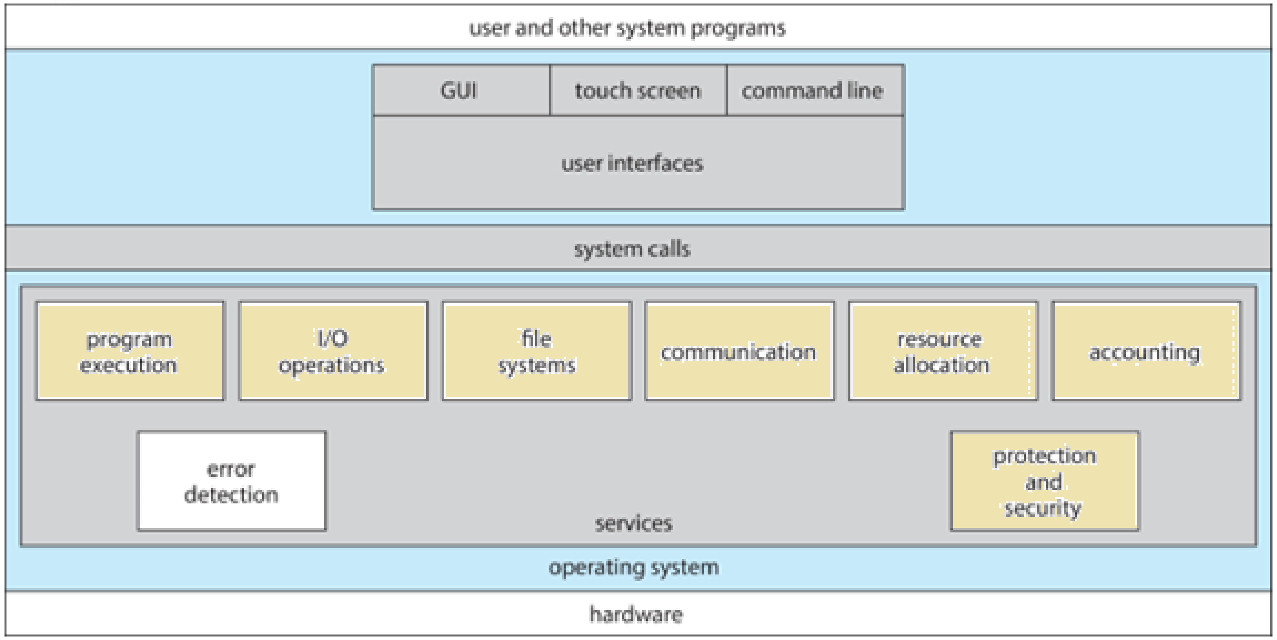


A View of Operating System Services

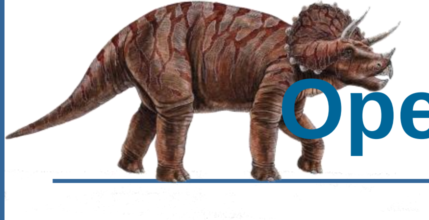


- One set of operating-system services provides functions that are helpful to the user:
  - **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
  - **Communications** – Processes may exchange information, on the same computer or between different computers over a network
    - Communications may be via shared memory or through message passing (packets moved by the OS)

Operating System Services (Cont.)



- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the *CPU*, *memory hardware*, *I/O devices*, and *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



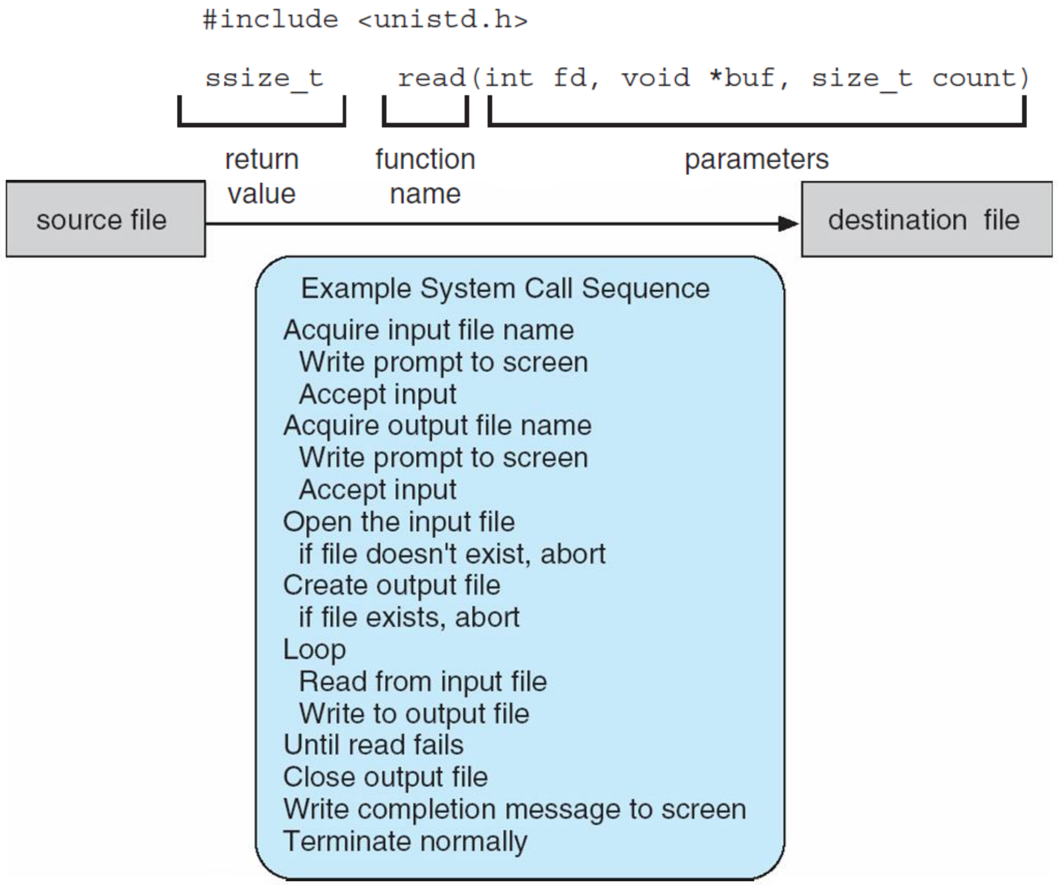
Operating System Services (Cont.)

- 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.
  - **Logging -** 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 (segmentation fault, parity check , stack overflow)
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

Systems calls

System Calls

- System call: Programming interface between user program and 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 (portability)



$$\text{#include <unistd.h>}$$

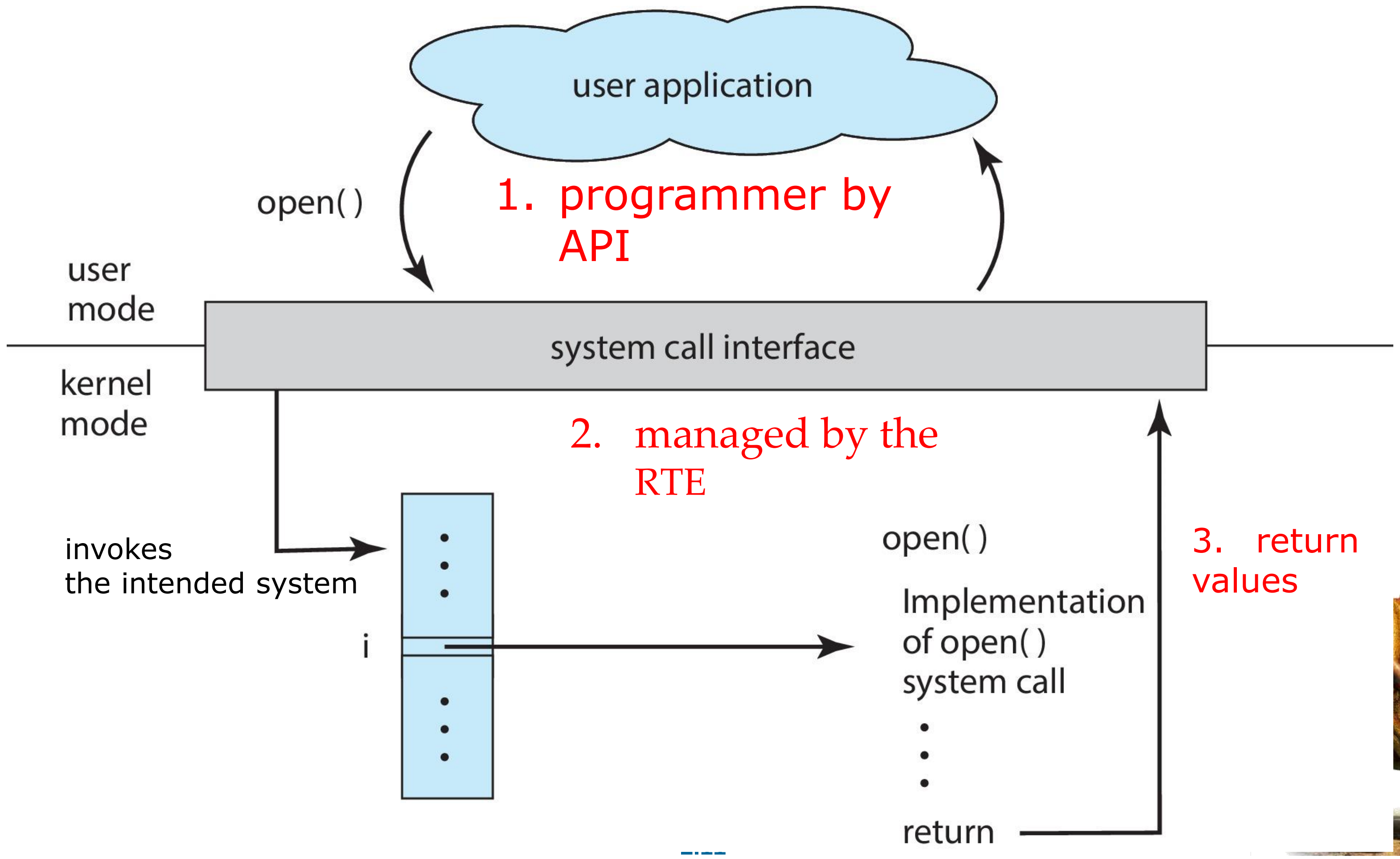
$$\underbrace{\texttt{ssize\_t}}_{\text{return value}}\ \underbrace{\texttt{read}}_{\text{function name}}\underbrace{\texttt{(int fd, void *buf, size\_t count)}}_{\text{parameters}}$$

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
```

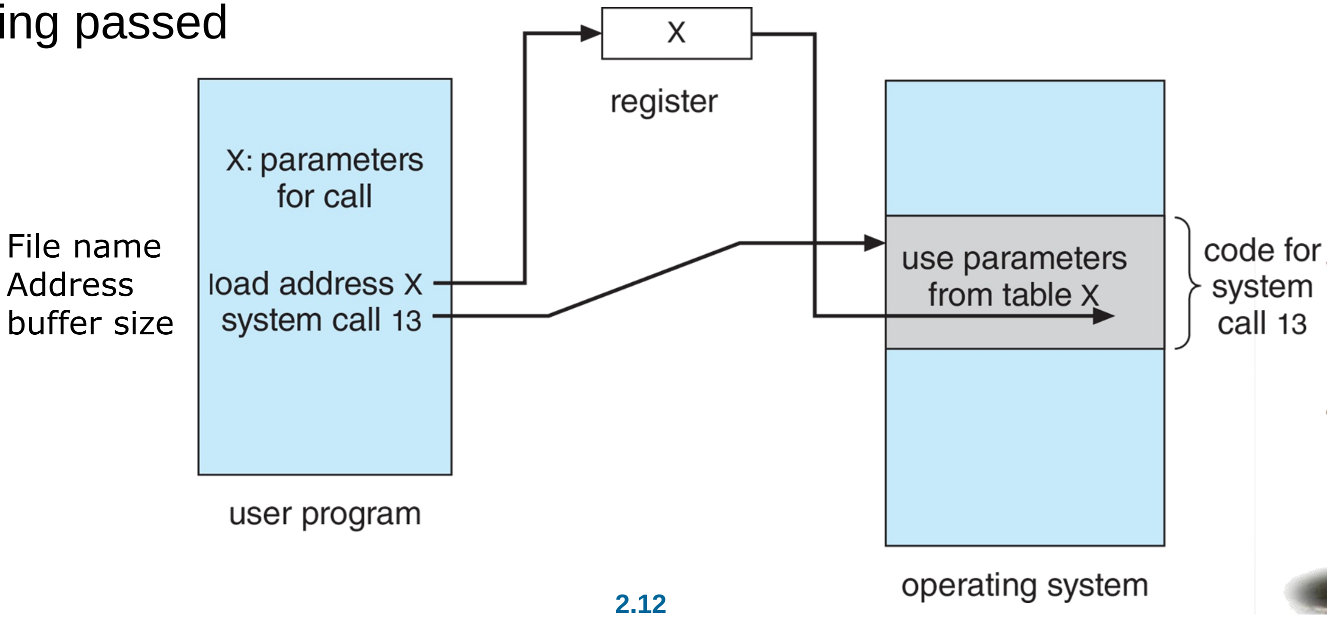
System Call Implementation

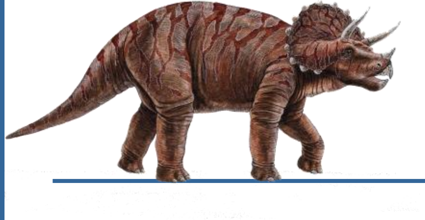
- Typically, a number is associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
  - OS interface hidden from programmer by API (processor switches between two modes and update Mode bit).



System Call Parameter Passing

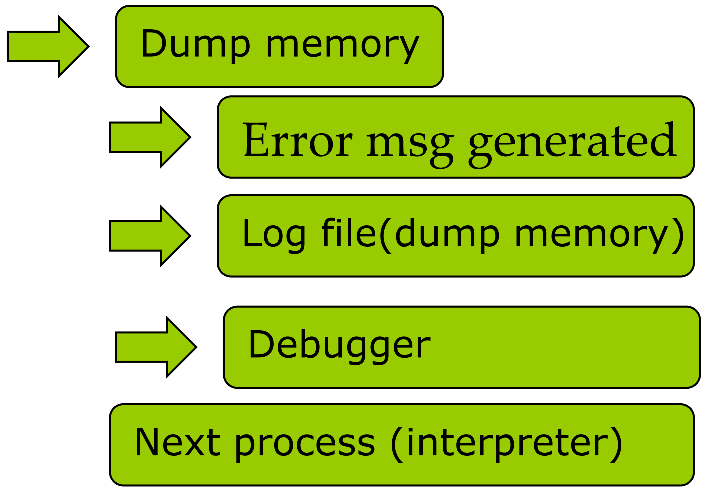
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken 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





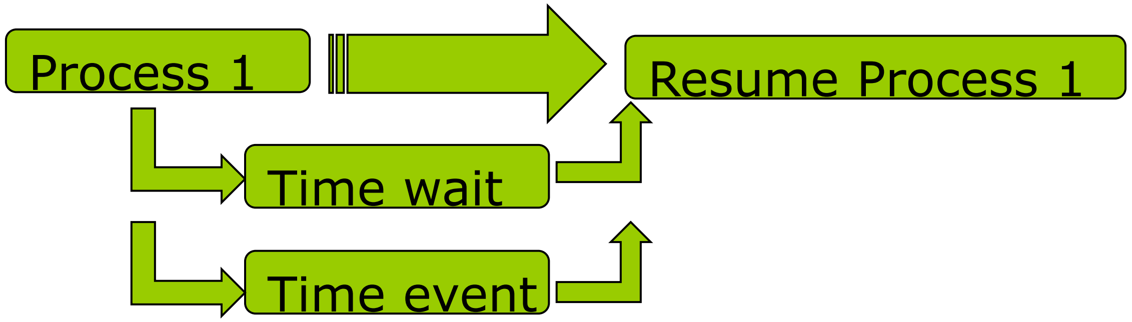
Types of System Calls

- **Process control** (System calls are used for the creation and management of new processes.)
  - Create process, terminate process.
  - End (normal exit), abort (abnormal exit).
  - Load, execute
  - **Debugger** for determining **bugs, single step** execution



Types of System Calls

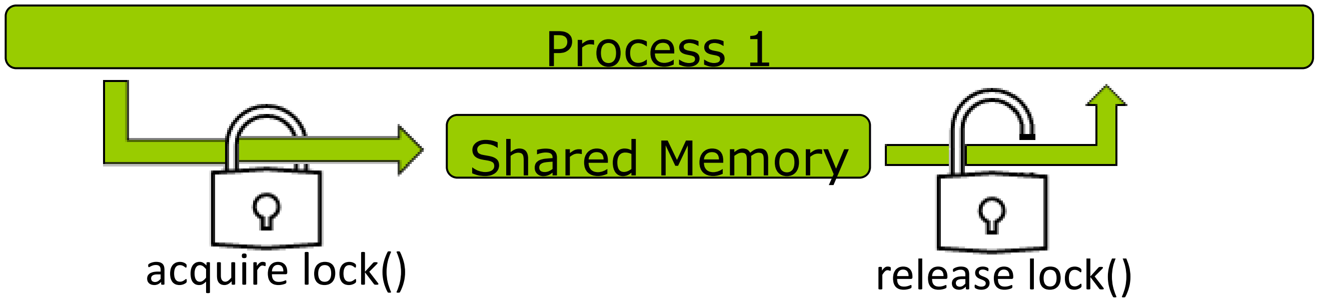
- **Process control** (System calls are used for the creation and management of new processes.)
  - Create process, terminate process.
  - End (normal exit), abort (abnormal exit).
  - Load, execute
  - Get process attributes, set process attributes (Max response time)



- Wait for time Wait event, signal event
- Allocate and free memory
- **Locks** for managing access to shared data between processes

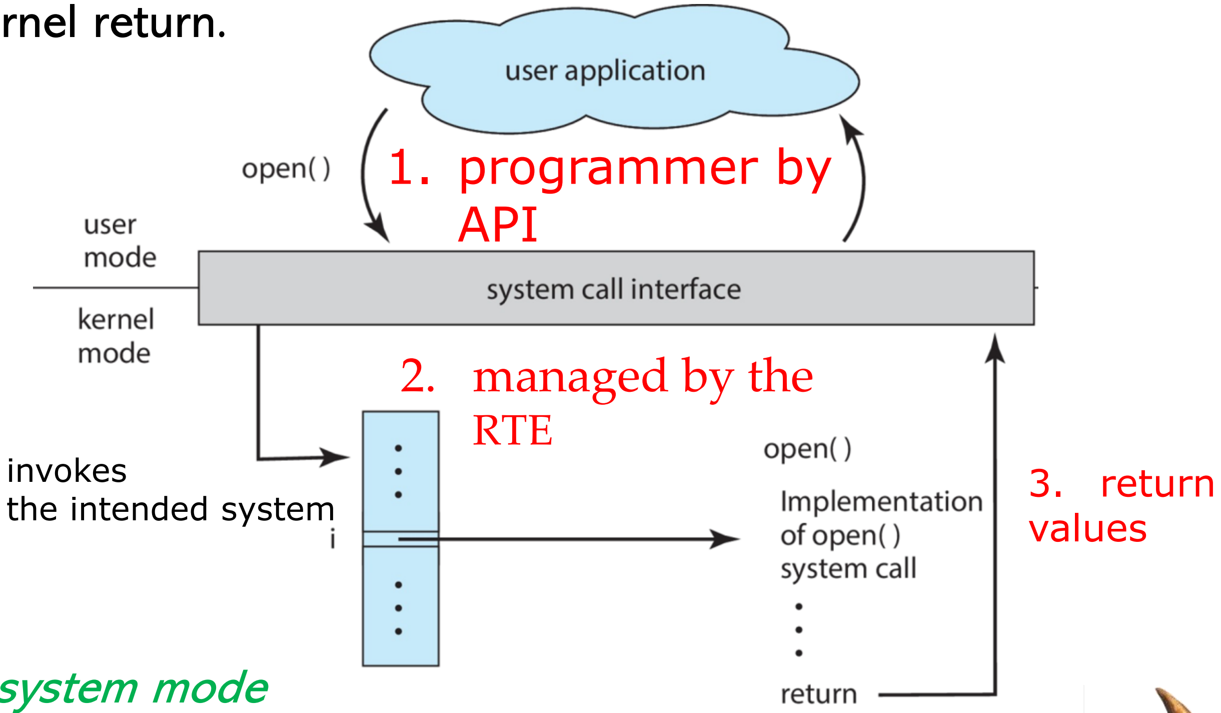
Types of System Calls

- **Process control** (System calls are used for the creation and management of new processes.)
  - **Locks** for managing access to shared data between processes

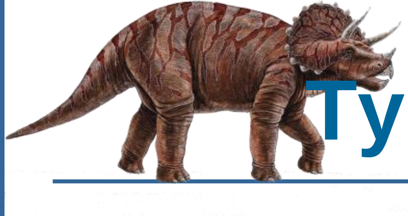


System Call Implementation (summary)

1. Higher language interface (a part of a system library)
   - Executes in *user mode*
   - Implemented to accept a standard procedure call
   - Wait for Kernel return.



2. Kernel part
   - Executes in *system mode*
   - Implements the system service
   - May cause blocking the caller (forcing it to wait)
   - After completion reports either the success or failure of the call then switch to User mode)

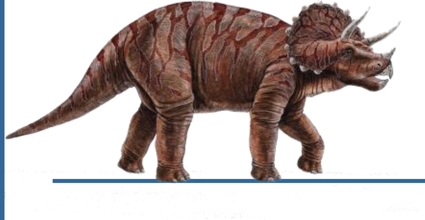


Types of System Calls (Cont.)

provide API to perform those operations using code

- File management
  - create file, delete file
  - open, close file
  - read, write
  - get and set file attributes(file name, file type, protection codes, accounting information, etc)
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

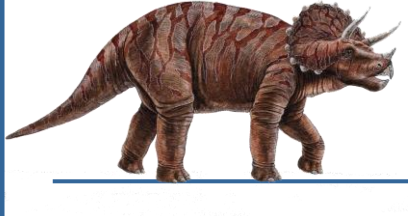
If any device is unavailable, the process will have to wait until sufficient resources are available.



Types of System Calls (Cont.)

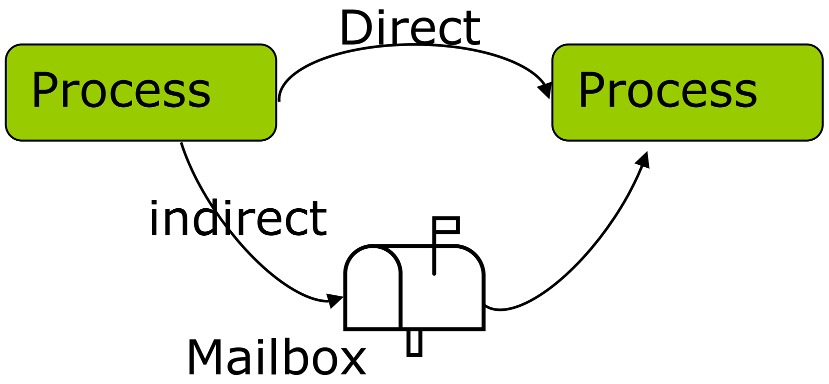
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - Info about memory (occupied, free).
  - get and set process, file, or device attributes

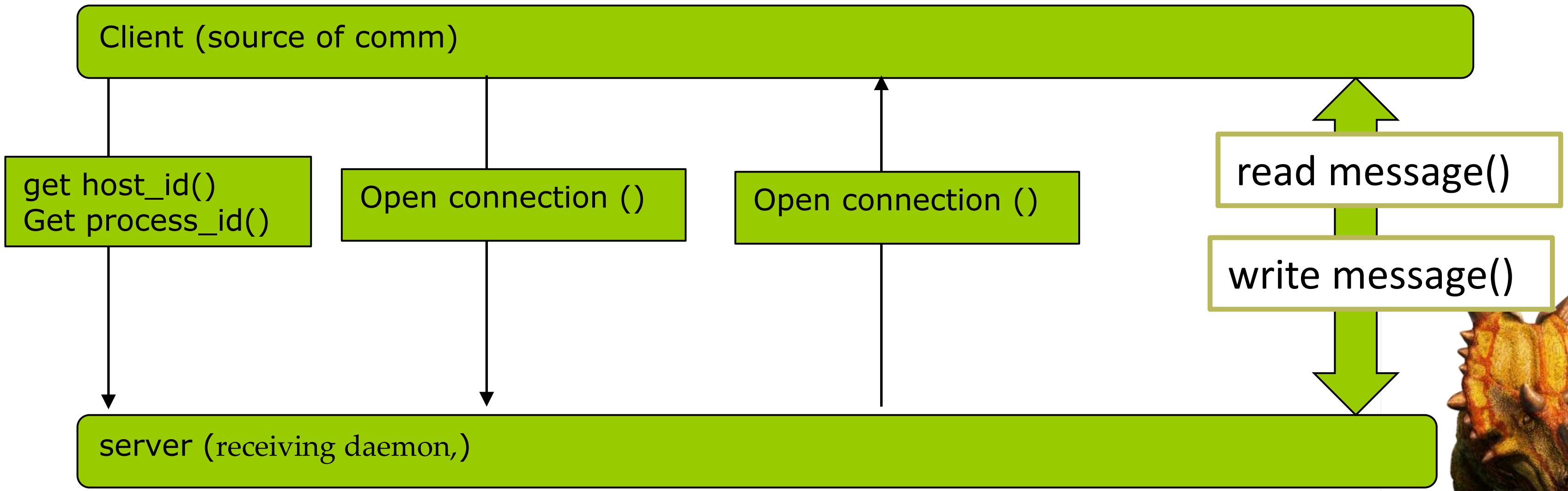
- "Strace", (Linux) systems, lists each system call as it is executed.
- "Single step", (CPU) , for debugging the code.



Types of System Calls (Cont.)

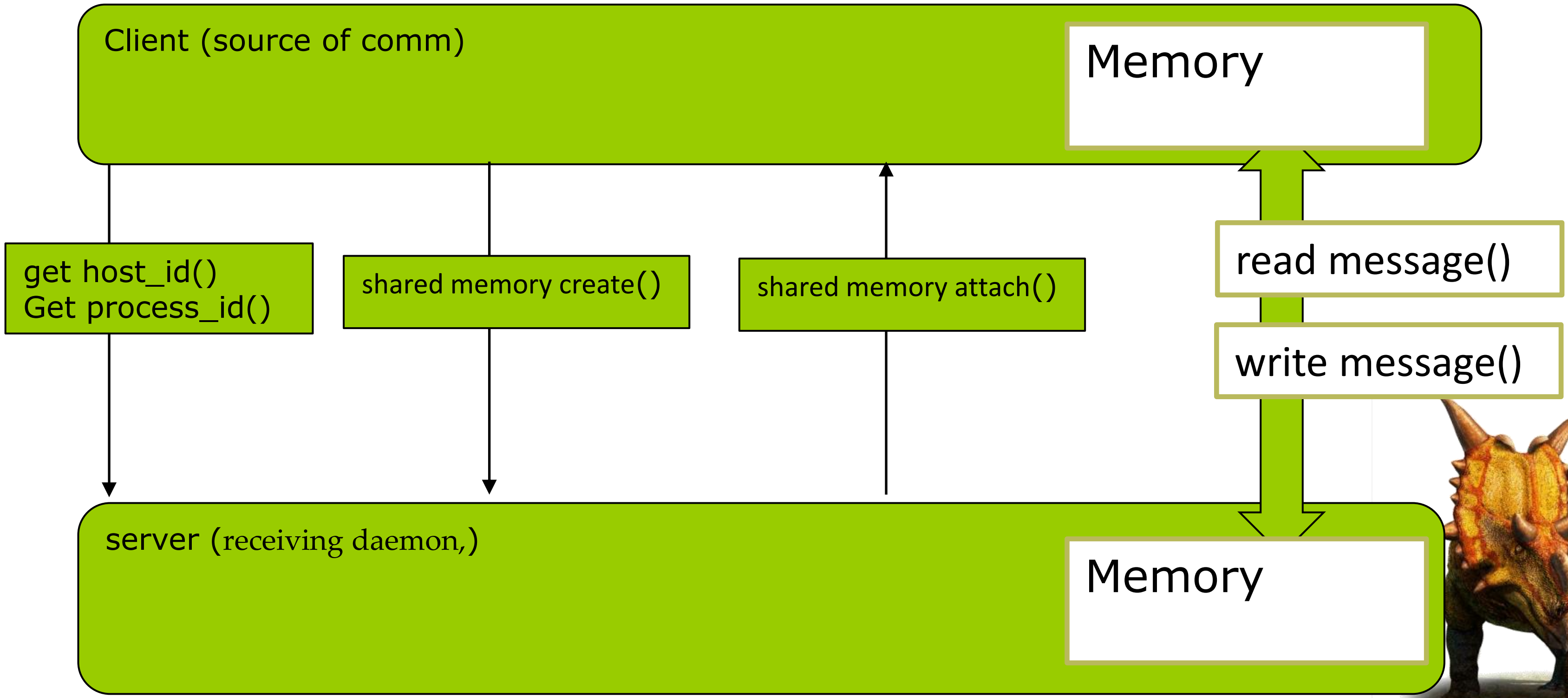
- **Communications**
- Two common models of inter-process communication:
  - **1- Message Passing Model**:
    - Exchanging smaller amounts of data, because no conflicts need to avoid



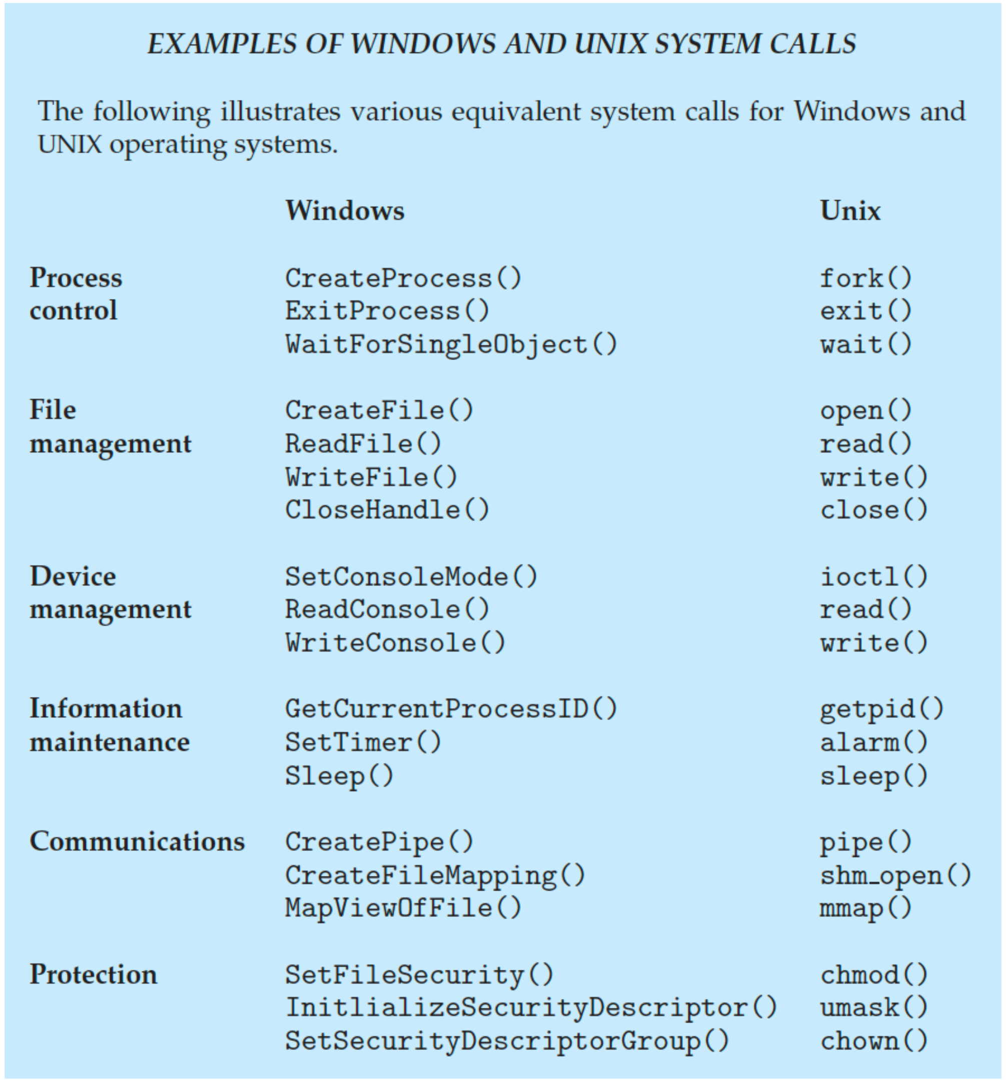


Types of System Calls (Communications)

- Two common models of inter-process communication:
  - **2-Shared-memory model** create and gain access to memory regions
    - Two or more processes agree to remove this
    - Not under the operating system’s control
    - Processes check and examine faults (write simultaneously).
    - Allows maximum speed and convenience of communication



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()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

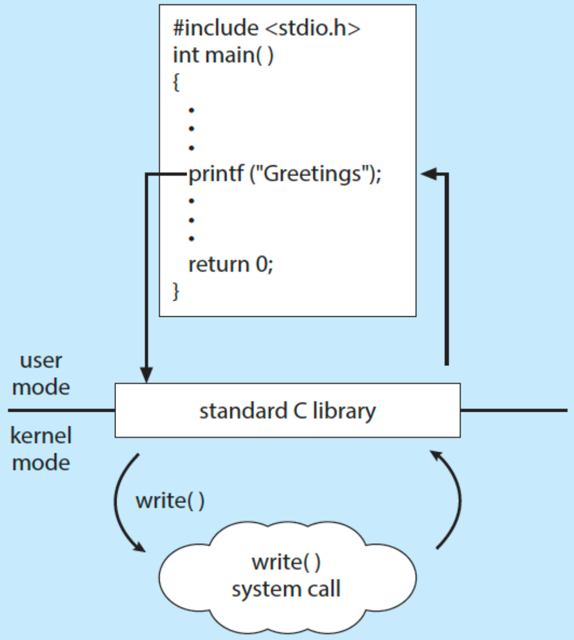
Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

Standard C Library Example

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

THE STANDARD C LIBRARY

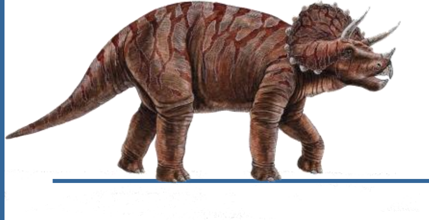
The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



```
#include <stdio.h>
int main( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

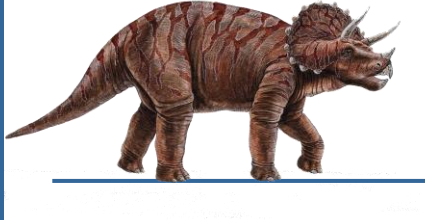
Why Applications are Operating System Specific

- Apps compiled on one system usually not executable on other operating systems
- Each operating system provides its own unique system calls
  - Own file formats, etc.
- Apps can be multi-operating system
  - Written in interpreted language like Python, Ruby, and interpreter available on multiple operating systems
  - App written in language that includes a VM containing the running app (like Java)
  - Use standard language (like C), compile separately on each operating system to run on each
- **Application Binary Interface** (**ABI**) is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc.



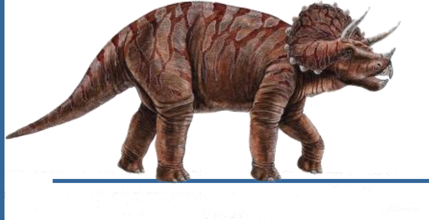
Design and Implementation

- Design and Implementation of OS is 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



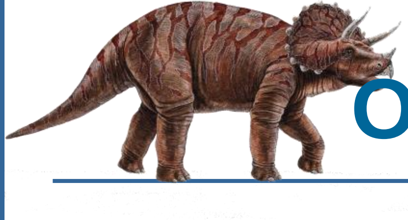
Policy and Mechanism

- **Policy: What** needs to be done?
  - Example: Interrupt after every 100 seconds
- **Mechanism: How** to do something?
  - Example: timer
- Important principle: separate policy from mechanism
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later.



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
- **Emulation** can allow an OS to run on non-native hardware

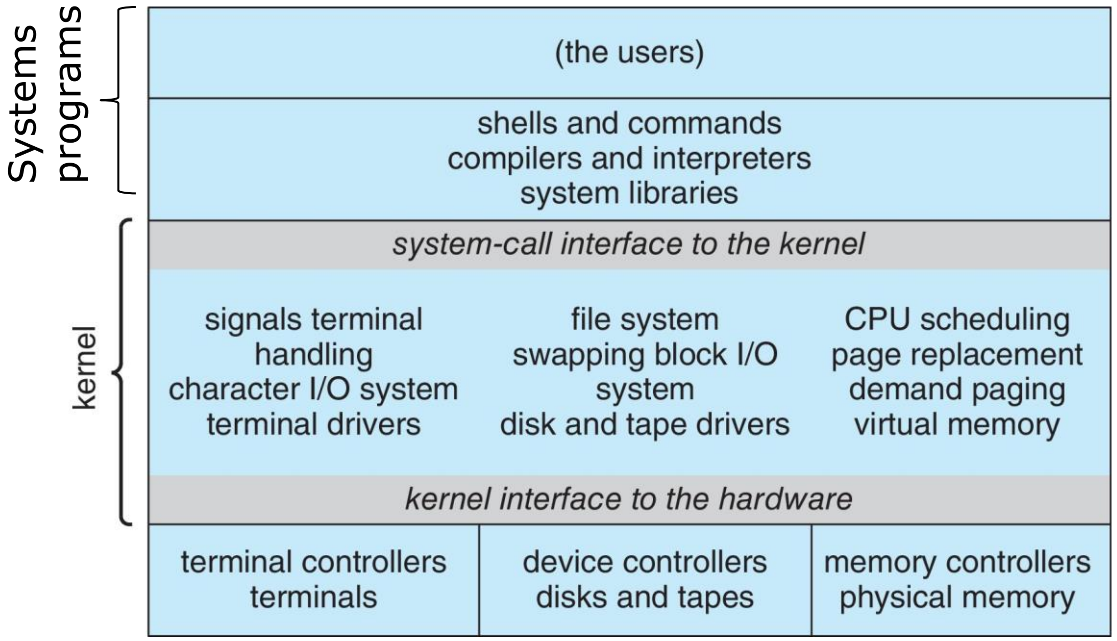


Operating System Structure

- General-purpose OS is very large program
- Various ways to structure ones
    - Simple structure – MS-DOS
    - More complex – UNIX
    - Layered – an abstraction
    - Microkernel – Mach

Monolithic Structure – Original UNIX

- Example: Unix, Linux, Microsoft Windows (95, 98, Me), Solaris, DOS, etc.
- 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, etc



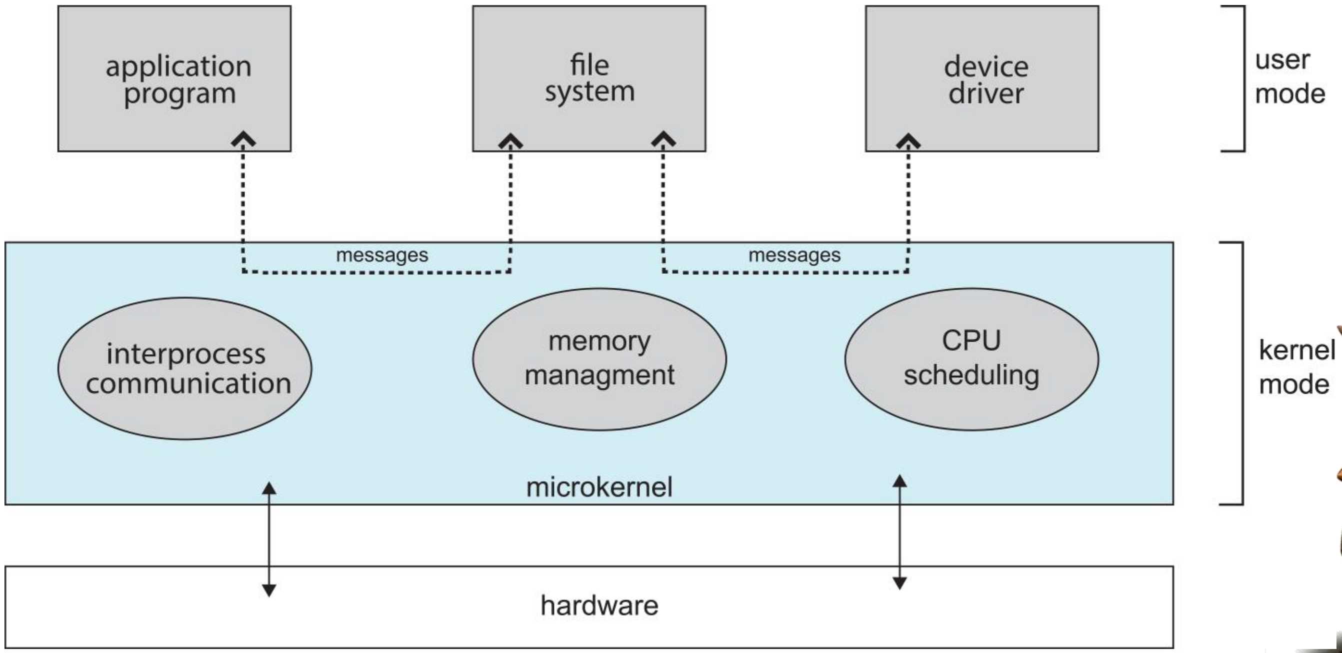
Monolithic Kernel

- Advantages:
  - Simple and easy to implement structure.
  - Faster execution due to direct access to all the services

- Disadvantages:
  - Updating (adding new features or remove obsolete features) is very hard . All the code needs to be rewritten and recompiled to add or remove any feature.
  - If any service fails in the monolithic kernel, it leads to the failure of the entire system.

Microkernels

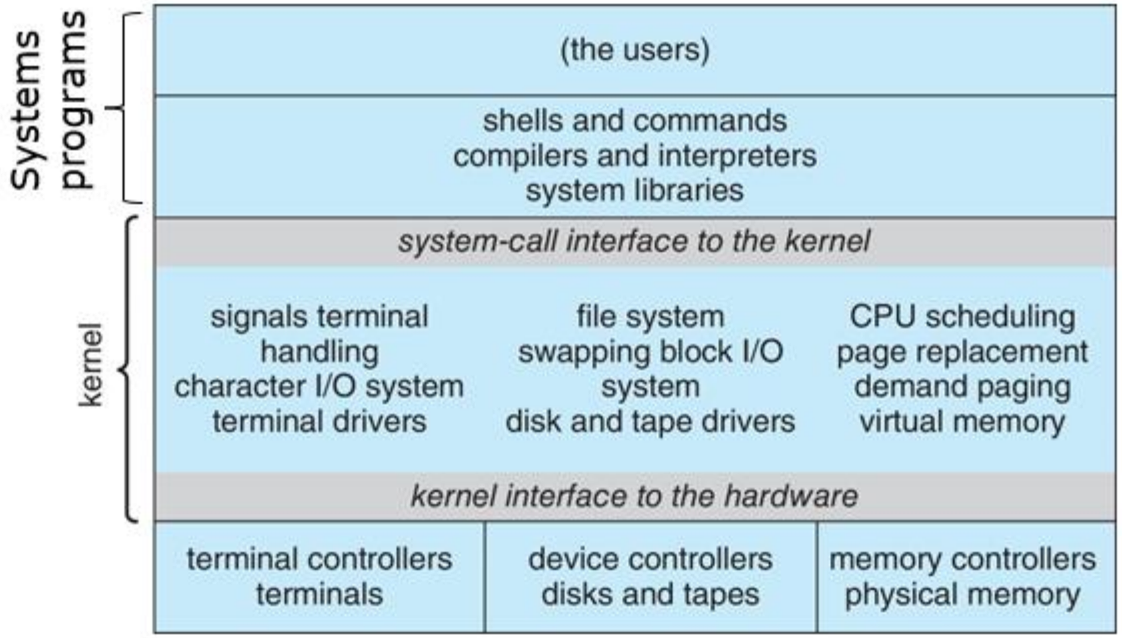
- Moves as much from the kernel into user space
- **Mach** is an example of **microkernel (**Example: Symbian, L4Linux, Mac OS X)
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits: extendable More reliable, More secure
- Detriments: Performance overhead of user space to kernel space communication (speed)

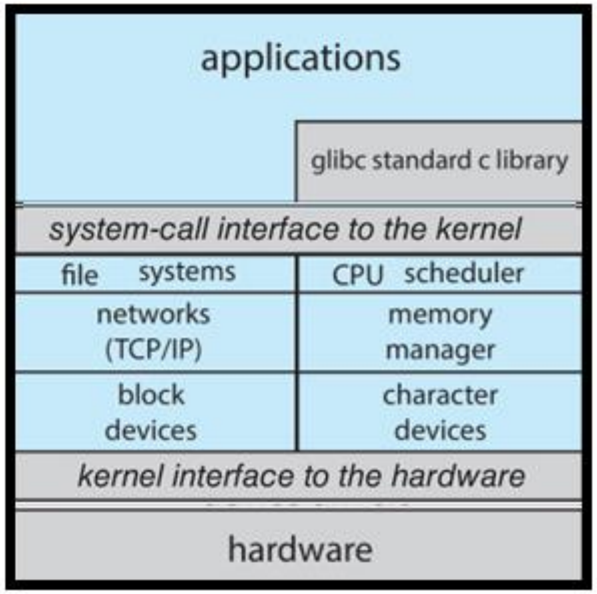


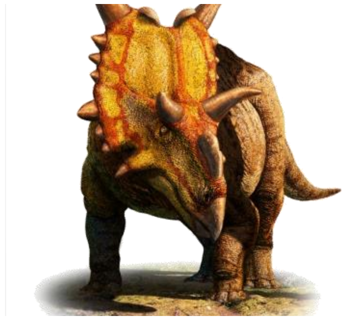
Monolithic kernel	Microkernel.
In a monolithic kernel, user services and kernel services are kept in the same address space.	In a microkernel, user services and kernel services are kept in separate address space.
The monolithic kernel is larger in size.	The microkernel is smaller in size.
Execution speed is faster in the case of a monolithic kernel.	Execution speed is slower in the case of a microkernel.
The monolithic kernel is hard to extend.	The microkernel is easily extendable.
The whole system will crash if one component fails.	If one component fails, it does not affect the working of the microkernel.
Fewer lines of code need to be written for a monolithic kernel.	More lines of code need to be written for a microkernel.
Debugging and management are complex in the case of a monolithic kernel.	Debugging and management are more straightforward as the kernel is smaller in size.
Monolithic OS is easier to design and implement.	Microkernels are complex to design.
Examples of this OS include Unix, Linux, OS/360, OpenVMS, Multics, AIX, BSD etc.	Examples of this OS include QNX, L4Linux, Mac OS X, Symbian, Singularity etc.

Linux System Structure

- Monolithic plus modular design (modular design)
- Their speed and efficiency high

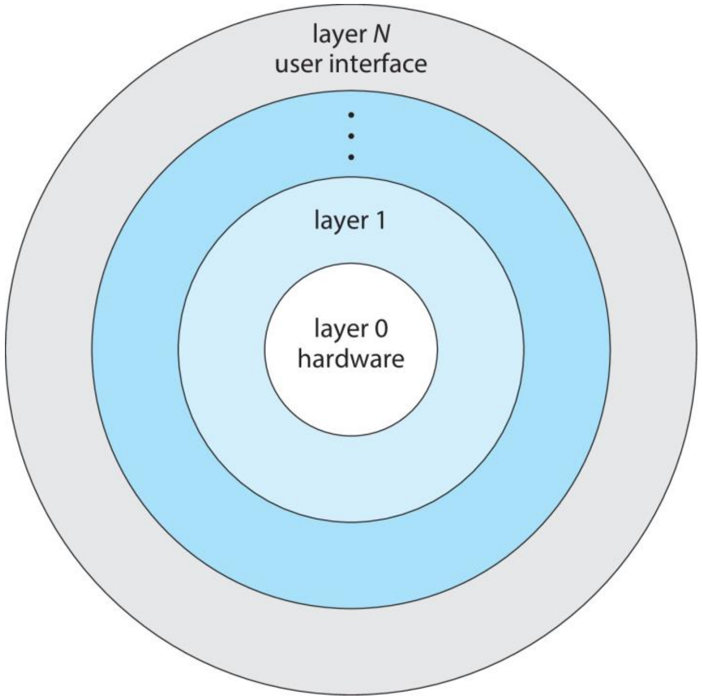


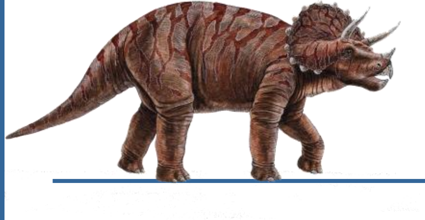




Layered Approach

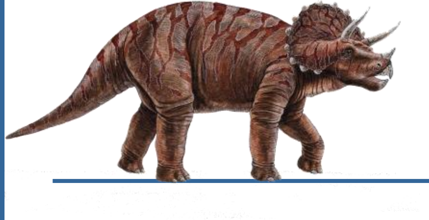
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level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





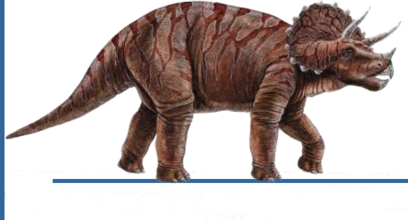
Modules

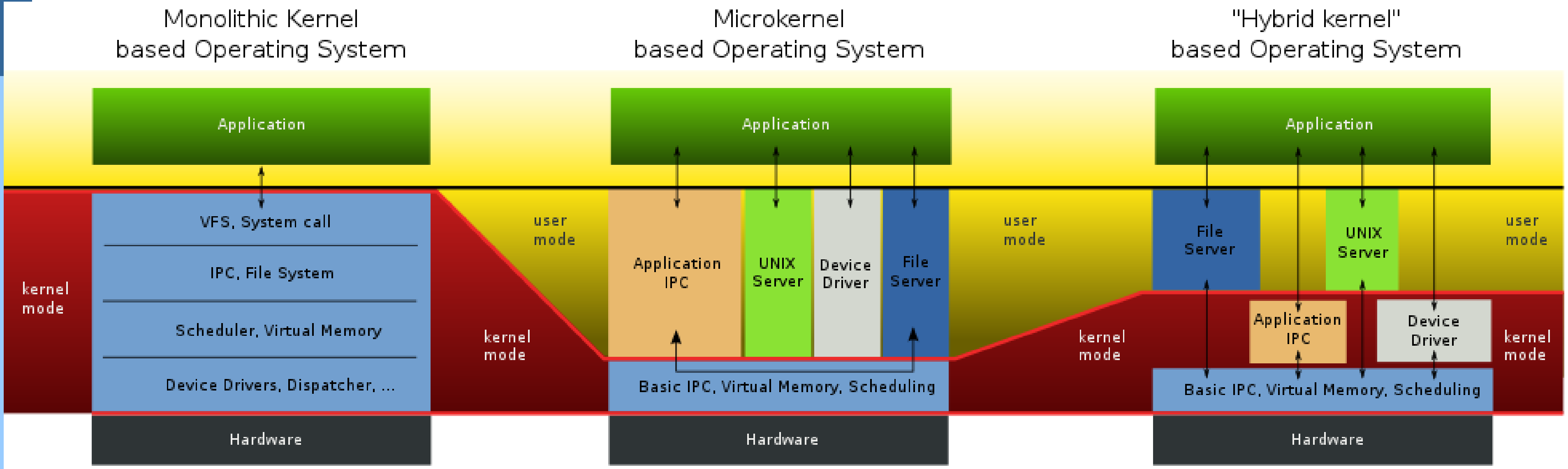
- Many modern operating systems implement **loadable kernel modules** (**LKMs**)
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
  - 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**)




Monolithic Kernel based Operating System
Microkernel based Operating System
"Hybrid kernel" based Operating System
Application
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware
kernel mode
user mode
Application IPC
UNIX Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling
File Server
UNIX Server
Application IPC
Device Driver

End of Chapter 2