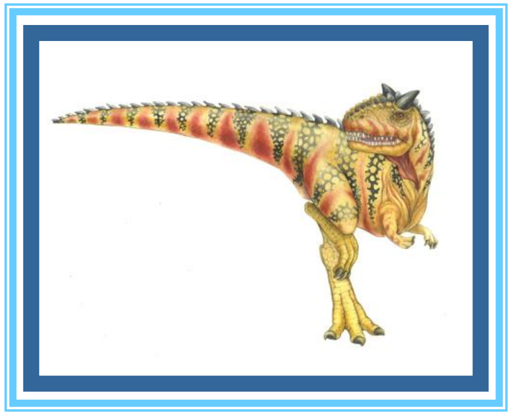
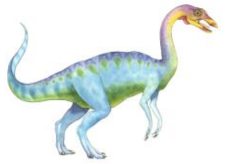
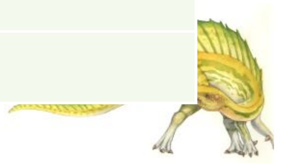
Chapter 2: Operating-System Structures





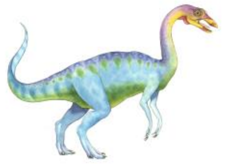
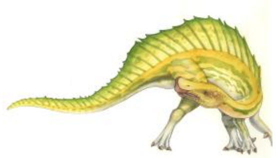
Schedule

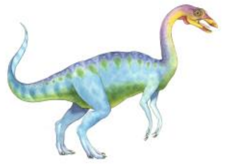
Week 1	Introduction
Week 2	Operating System Structures
Week 3	Processes
Week 4	Threads
Week 5	Threads
Week 6	Midterm 1
Week 7	Synchronization
Week 8	Classical Problems
Week 9	CPU Scheduling
Week 10	CPU Scheduling
Week 11	Midterm 2
Week 12	Deadlocks
Week 13	Memory Management
Week 14	Virtual Memory
Week 15	Virtual Memory



Chapter 2: Operating-System Structures

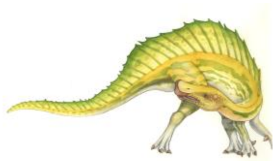
- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Operating System Debugging
- Operating System Generation
- System Boot



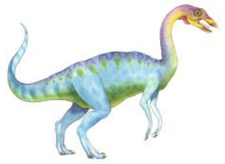
Objectives

- 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



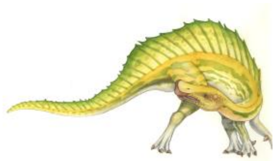
Operating System Services

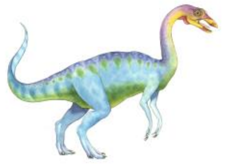
- Operating systems provide:
  - **User interface** - Almost all operating systems have a user interface (**UI**).
    - **Command-Line Interface(CLI)**, text commands
    - **Graphics User Interface (GUI)**,
    - **Batch Interface,** commands and directives are entered intoa file and file is executed
  - **Program execution**
  - **I/O operations**
  - **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
    - Communications may be via shared memory or through message passing (packets moved by the OS)



Operating System Services (Cont.)

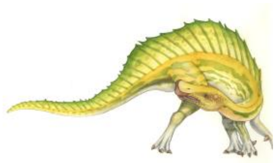
- (Cont.):
  - **Error detection**
    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user’s and programmer’s abilities to efficiently use the system



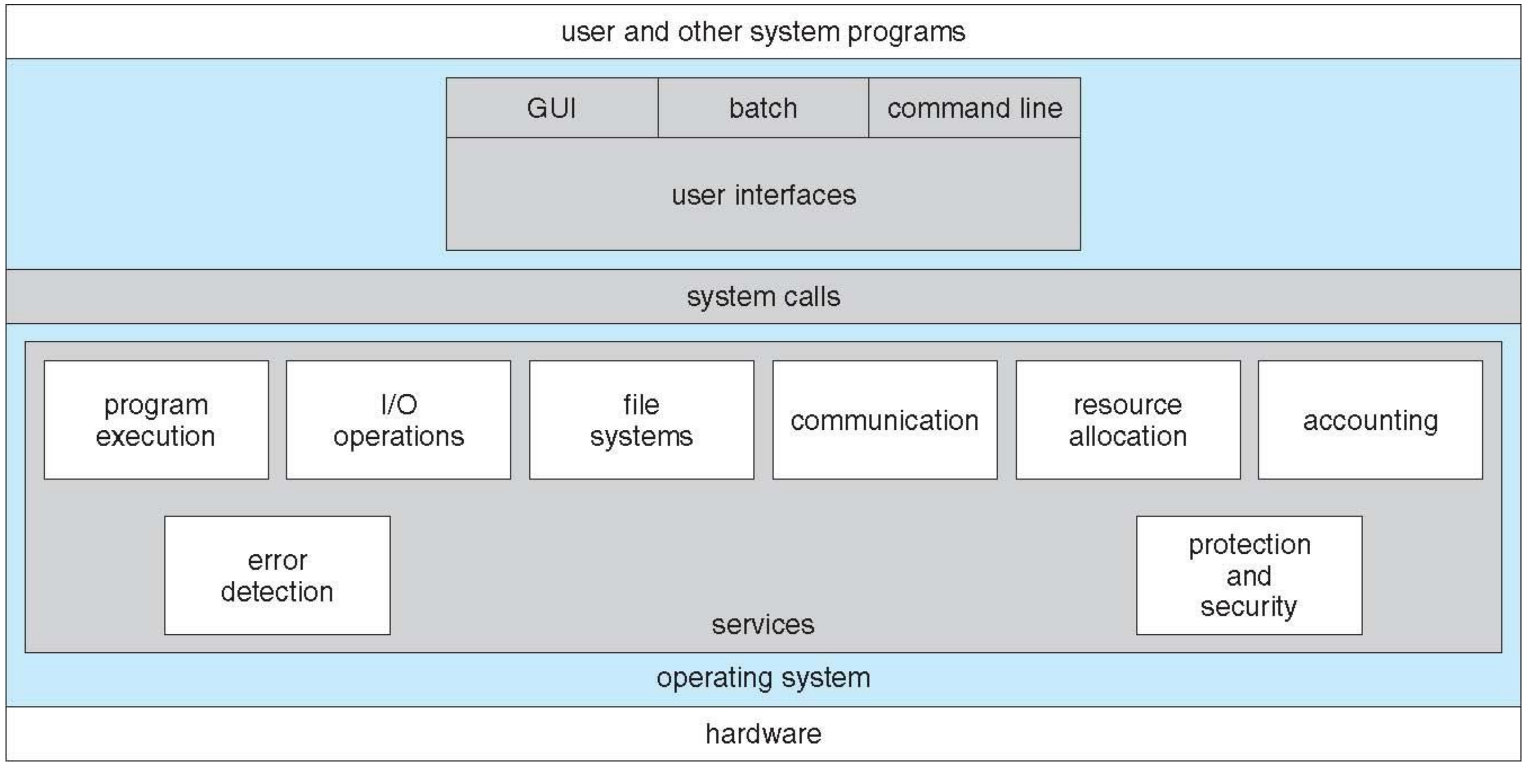


Operating System Services (Cont.)

- **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



A View of Operating System Services

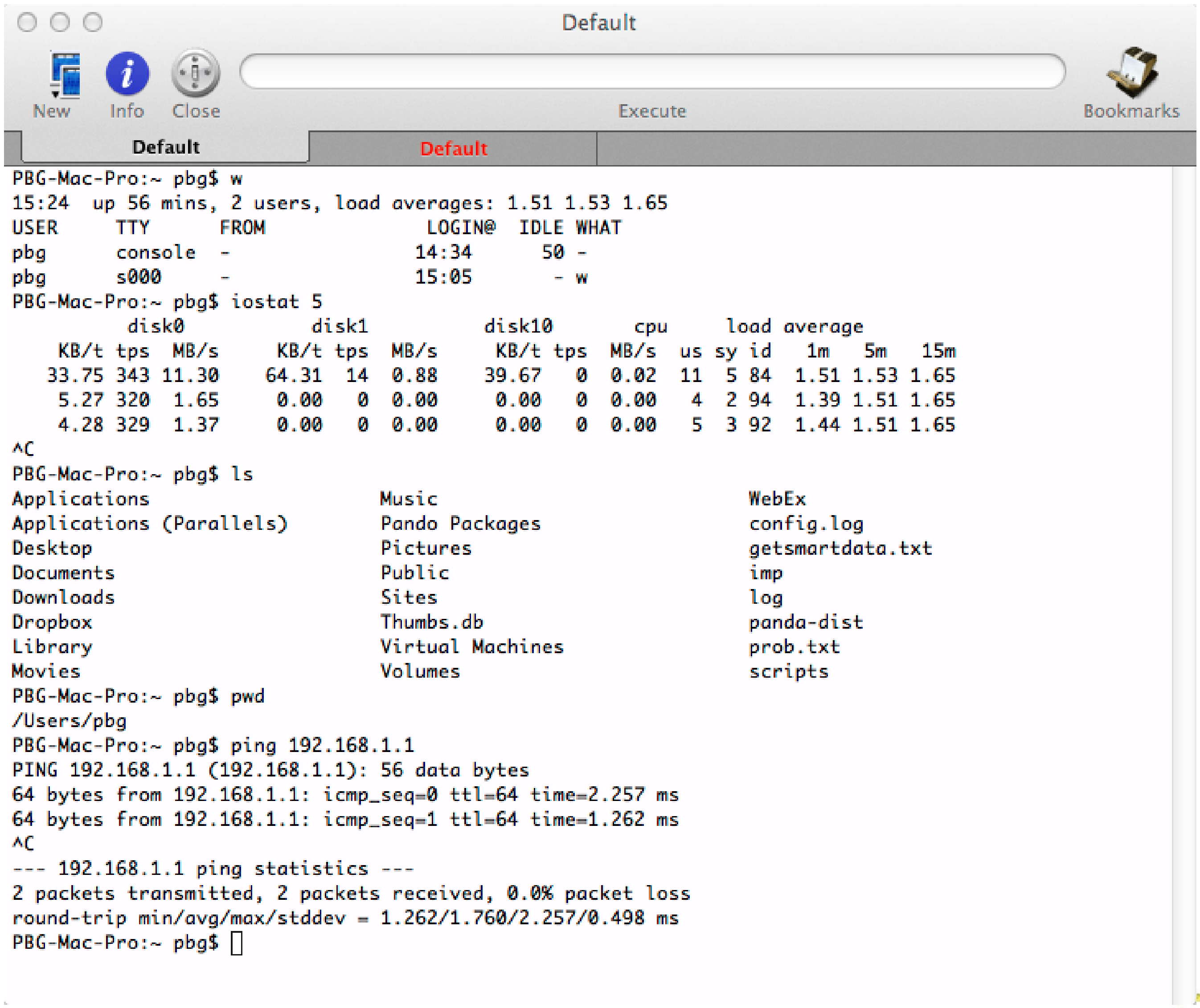


User Operating System Interface - CLI

CLI or **command interpreter** allows direct command entry

- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs
  - If the latter, adding new features doesn't require shell modification
  - Unix example
    - rm file.txt
  - In this way, programmers can add new commands to the system easily by creating new files with the proper names

Bourne Shell Command Interpreter

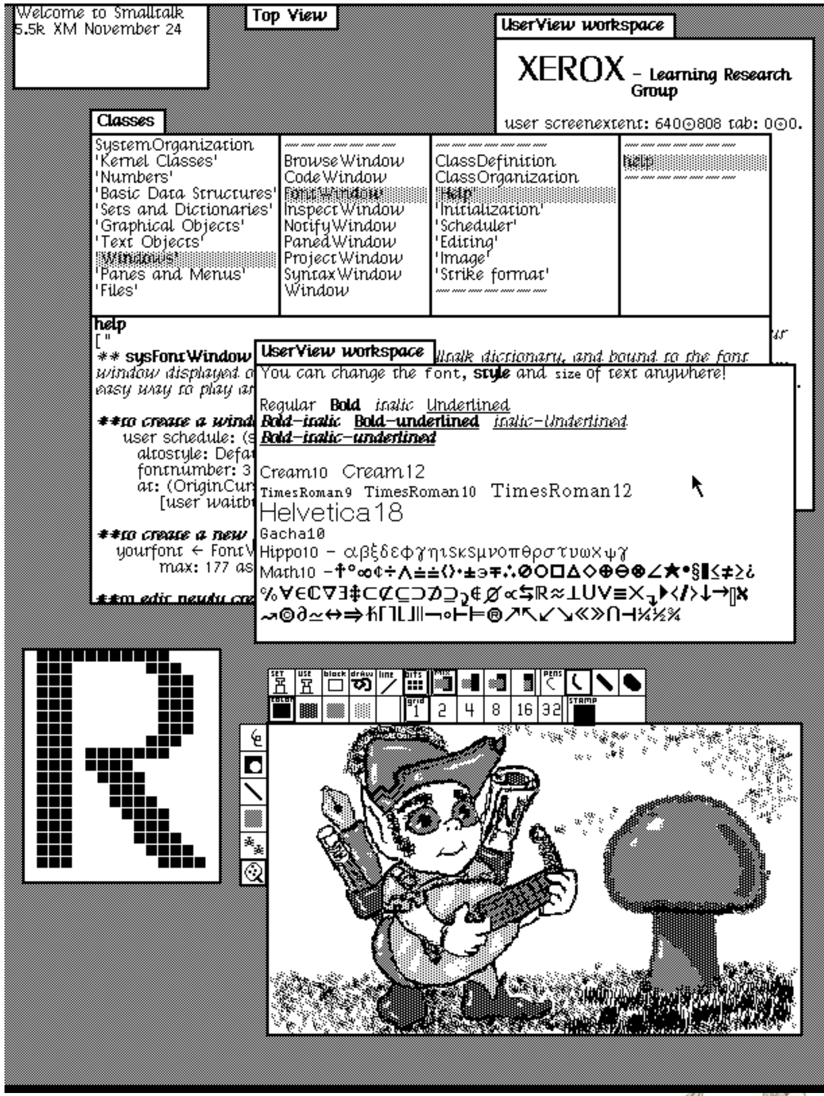


```
PBG-Mac-Pro:~ pbg$ w
15:24  up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER     TTY      FROM              LOGIN@  IDLE WHAT
pbg      console  -                14:34      50 -
pbg      s000     -                15:05       - w
PBG-Mac-Pro:~ pbg$ iostat 5
          disk0           disk1           disk10       cpu     load average
    KB/t tps  MB/s     KB/t tps  MB/s     KB/t tps  MB/s  us sy id   1m   5m   15m
   33.75 343 11.30    64.31  14  0.88    39.67   0  0.02  11  5 84  1.51 1.53 1.65
    5.27 320  1.65     0.00   0  0.00     0.00   0  0.00   4  2 94  1.39 1.51 1.65
    4.28 329  1.37     0.00   0  0.00     0.00   0  0.00   5  3 92  1.44 1.51 1.65
^C
PBG-Mac-Pro:~ pbg$ ls
Applications                    Music                           WebEx
Applications (Parallels)        Pando Packages                  config.log
Desktop                         Pictures                        getsmartdata.txt
Documents                       Public                          imp
Downloads                       Sites                           log
Dropbox                         Thumbs.db                       panda-dist
Library                         Virtual Machines                prob.txt
Movies                          Volumes                         scripts
PBG-Mac-Pro:~ pbg$ pwd
/Users/pbg
PBG-Mac-Pro:~ pbg$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBG-Mac-Pro:~ pbg$ 
```

User Operating System Interface - GUI

- User-friendly **desktop** metaphor interface
  - 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**)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

XEROX PARC



Welcome to Smalltalk
5.5k XM November 24

Top View

UserView workspace

XEROX - Learning Research Group

user screenextent: 640⊙808 tab: 0⊙0.

Classes

SystemOrganization
'Kernel Classes'
'Numbers'
'Basic Data Structures'
'Sets and Dictionaries'
'Graphical Objects'
'Text Objects'
'Windows'
'Panes and Menus'
'Files'

BrowseWindow
CodeWindow
FontWindow
InspectWindow
NotifyWindow
PanedWindow
ProjectWindow
SyntaxWindow
Window

ClassDefinition
ClassOrganization
'Help'
'Initialization'
'Scheduler'
'Editing'
'Image'
'Strike format'

help

help
["
** sysFontWindow
window displayed
easy way to play
**to create a wind
user schedule:
altostyle: Defa
fontnumber: 3
at: (OriginCur
[user waitb
**to create a new
yourfont ← FontW
max: 177 as
**to edit newly cre

UserView workspace

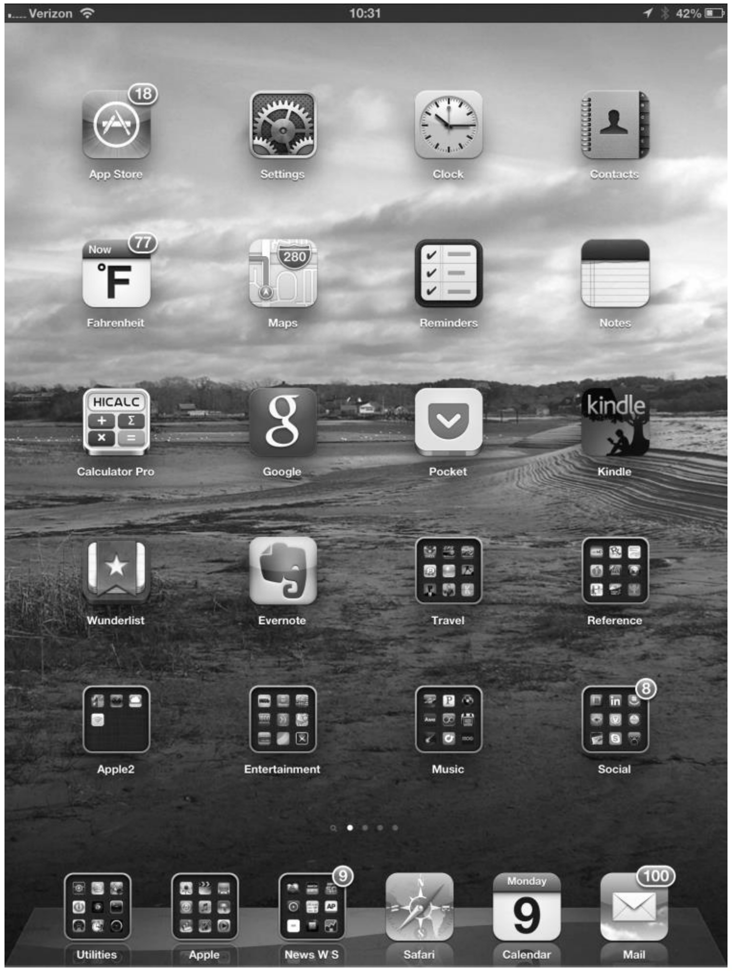
You can change the font, **style** and size of text anywhere!

Regular **Bold** *italic* Underlined
Bold-italic **Bold-underlined** *italic-Underlined*
Bold-italic-underlined

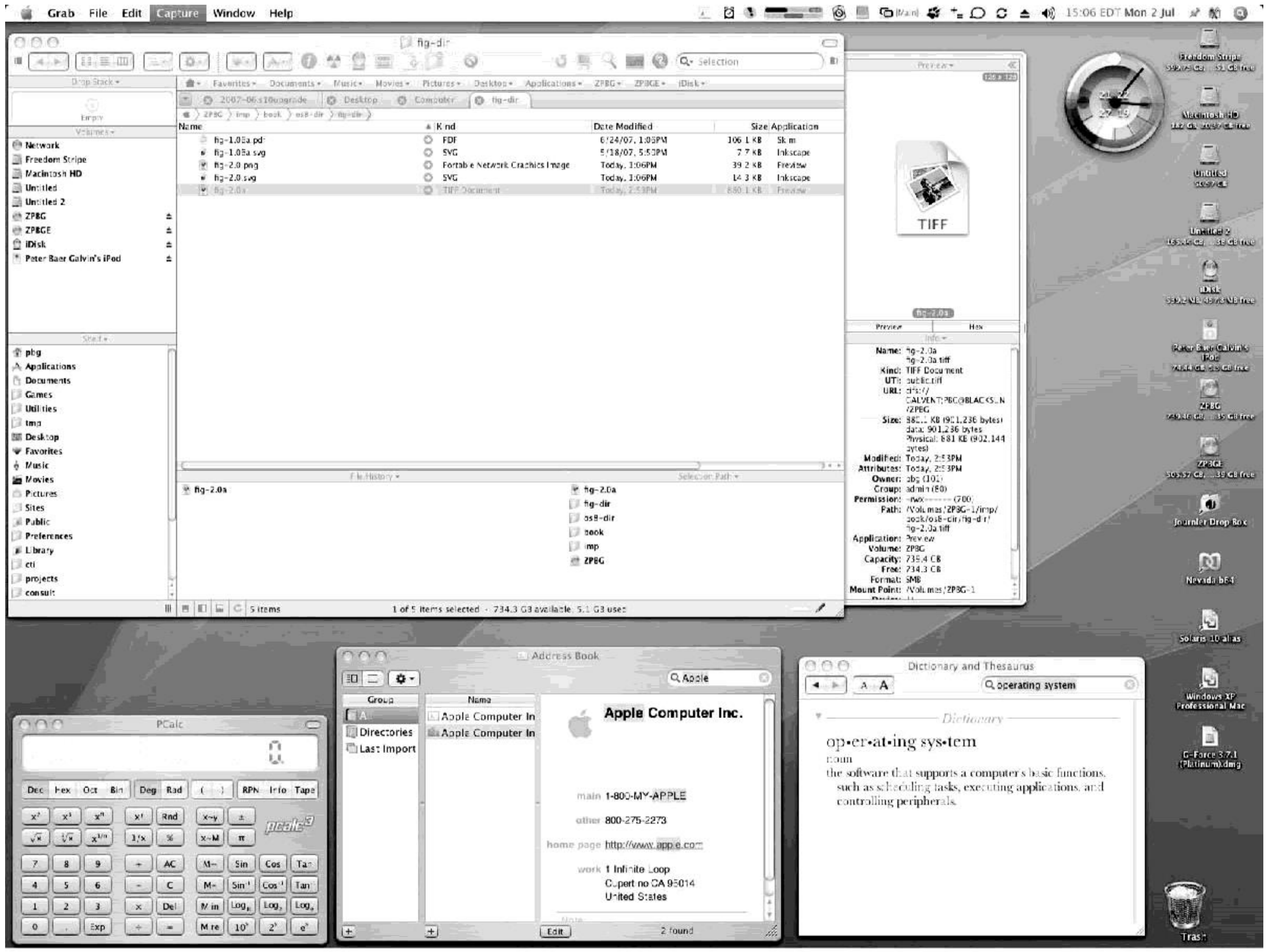
Cream10 Cream12
TimesRoman9 TimesRoman10 TimesRoman12
Helvetica18
Gacha10
Hippo10 – αβξδεφγηιςκςμνοπθρστυωχψγ
Math10 –†°∞¢÷∧≐≐⟨⟩·±∋∓∴∅○□△◇⊕⊖⊗∠★•§▌≤≠≥¿
%∀∈ℂ∇∃⇞⊂⊄⊆⊃⊅⊇↲∉∅∝⇆ℝ≈⊥∪∨≡×↴▸⟨/⟩↓→▯✗
↝©∂≃↔⇒ꝁ⌈⌉⌊⌋‖¬∘⊢⊨®↗↖↙↘«»∩⊣¼½¾

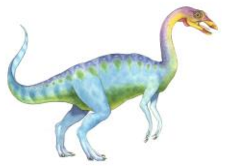
Touchscreen Interfaces

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



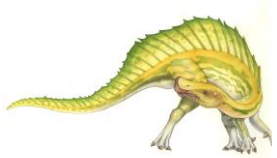
The Mac OS X GUI





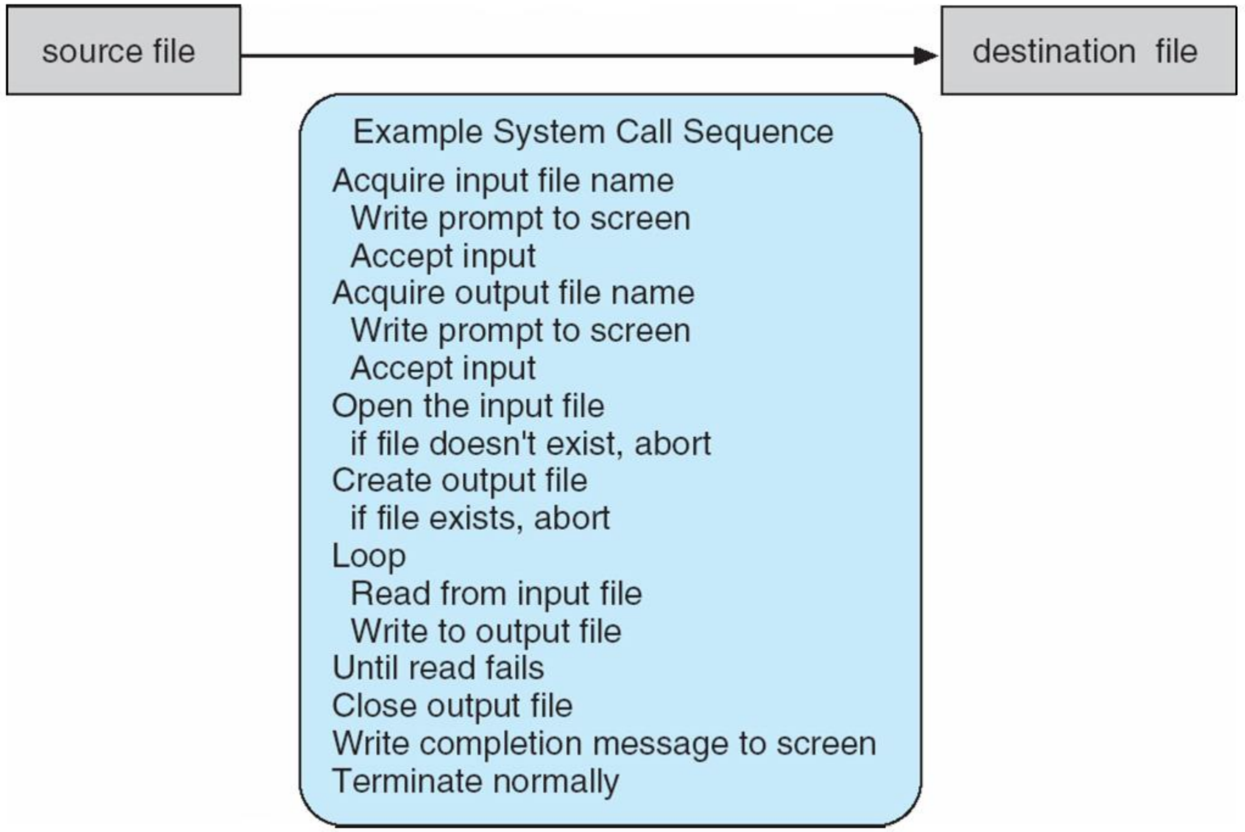
System Calls

- 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)



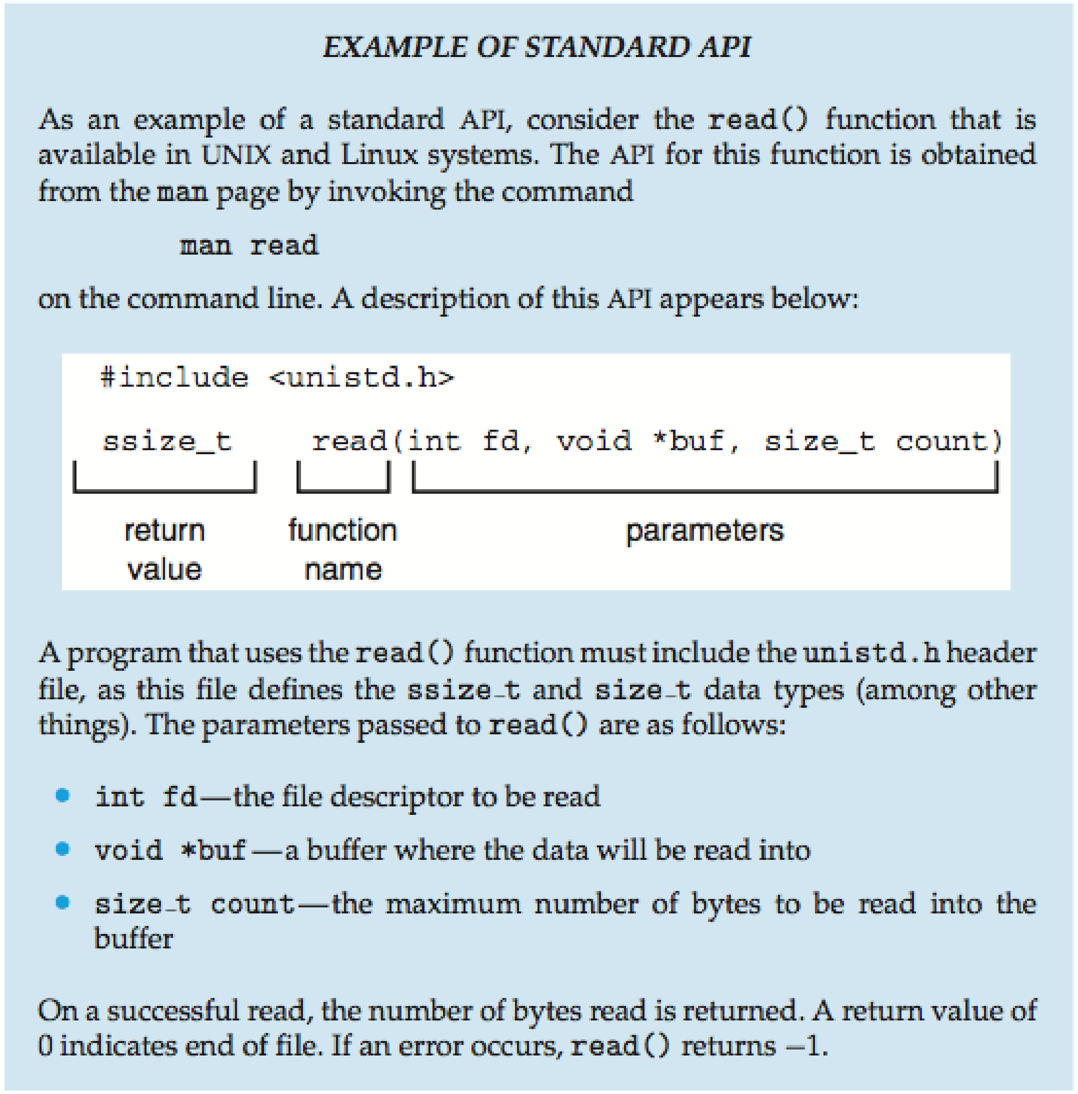
Example of System Calls

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



```
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
```

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:

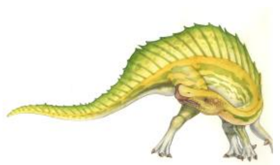
```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

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

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.



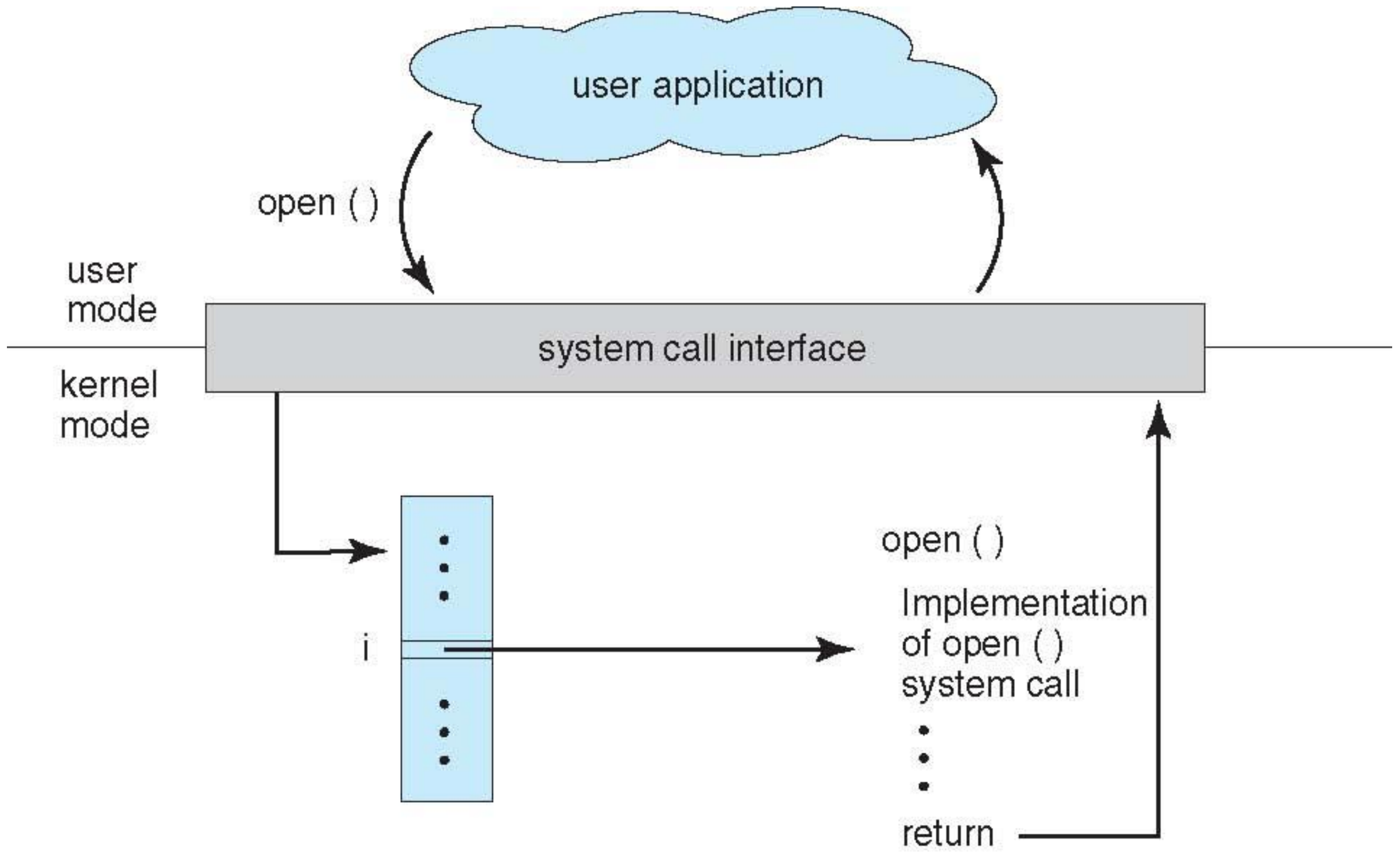
System Call Implementation

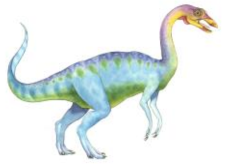
- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing 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)

System Call Portability

- Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer.
- For example, the Windows function CreateProcess() (which unsurprisingly is used to create a new process) actually invokes the NTCreateProcess() system call in the Windows kernel.
- Why would an application programmer prefer programming according to an API rather than invoking actual system calls?
  - Portability

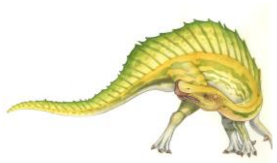
API – System Call – OS Relationship



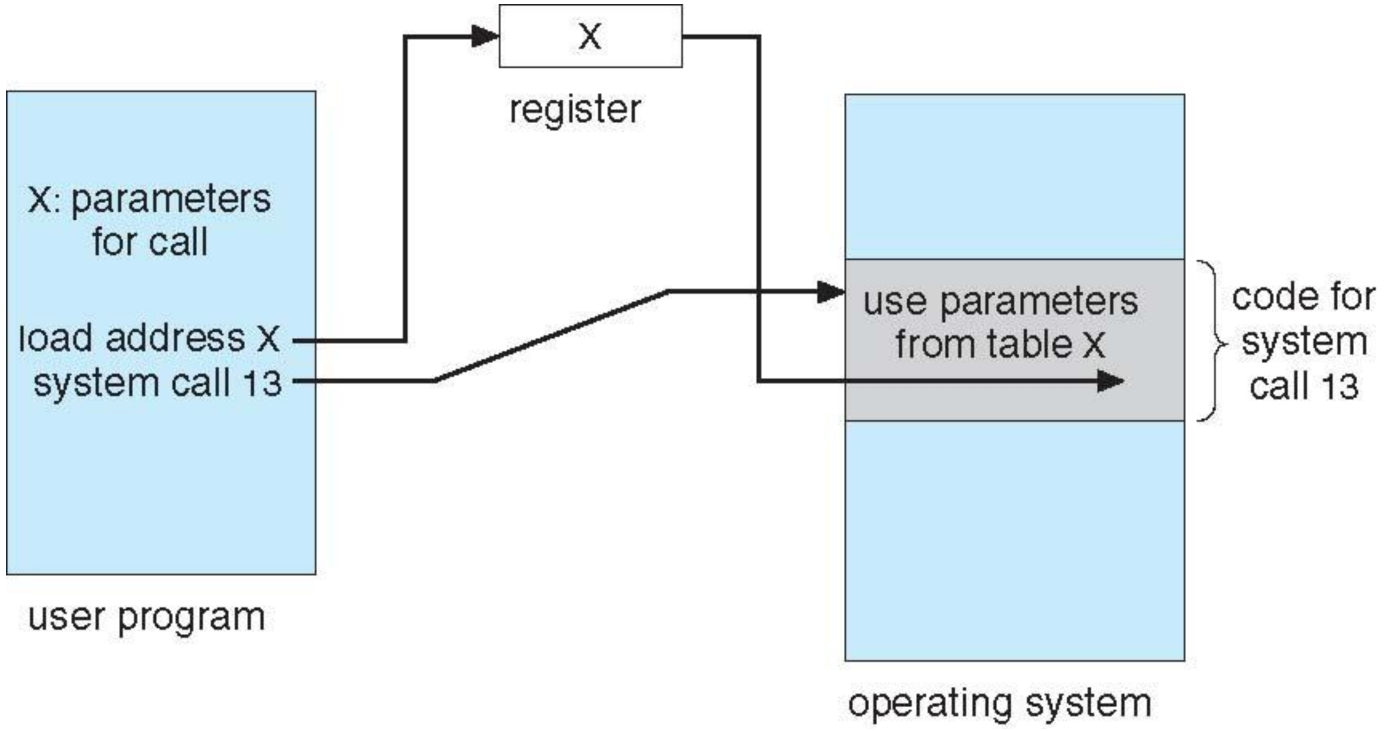


System Call Parameter Passing

- 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

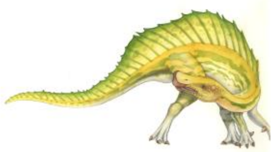


Parameter Passing via Table



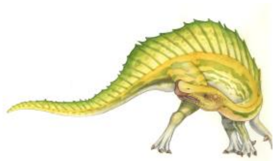
Assembly

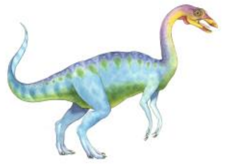
```
msg  db 'Hello again, World!$'   ; $-terminated message
org  0x100          ; .com files always start 256 bytes into the segment
; int 21h is going to want...

mov  dx, msg        ; the address of or message in dx
mov  ah, 9          ; ah=9 - "print string" sub-function
int  0x21           ; call dos services

mov  dl, 0x0d       ; put CR into dl
mov  ah, 2          ; ah=2 - "print character" sub-function
int  0x21           ; call dos services

mov  dl, 0x0a       ; put LF into dl
mov  ah, 2          ; ah=2 - "print character" sub-function
int  0x21           ; call dos services

mov  ah, 0x4c       ; "terminate program" sub-function
int  0x21           ; call dos services
```



Types of System Calls

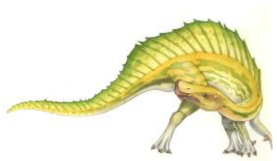
- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
    - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes





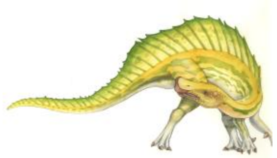
Types of System Calls

- 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



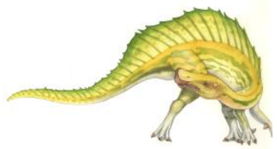
Types of System Calls (Cont.)

- 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

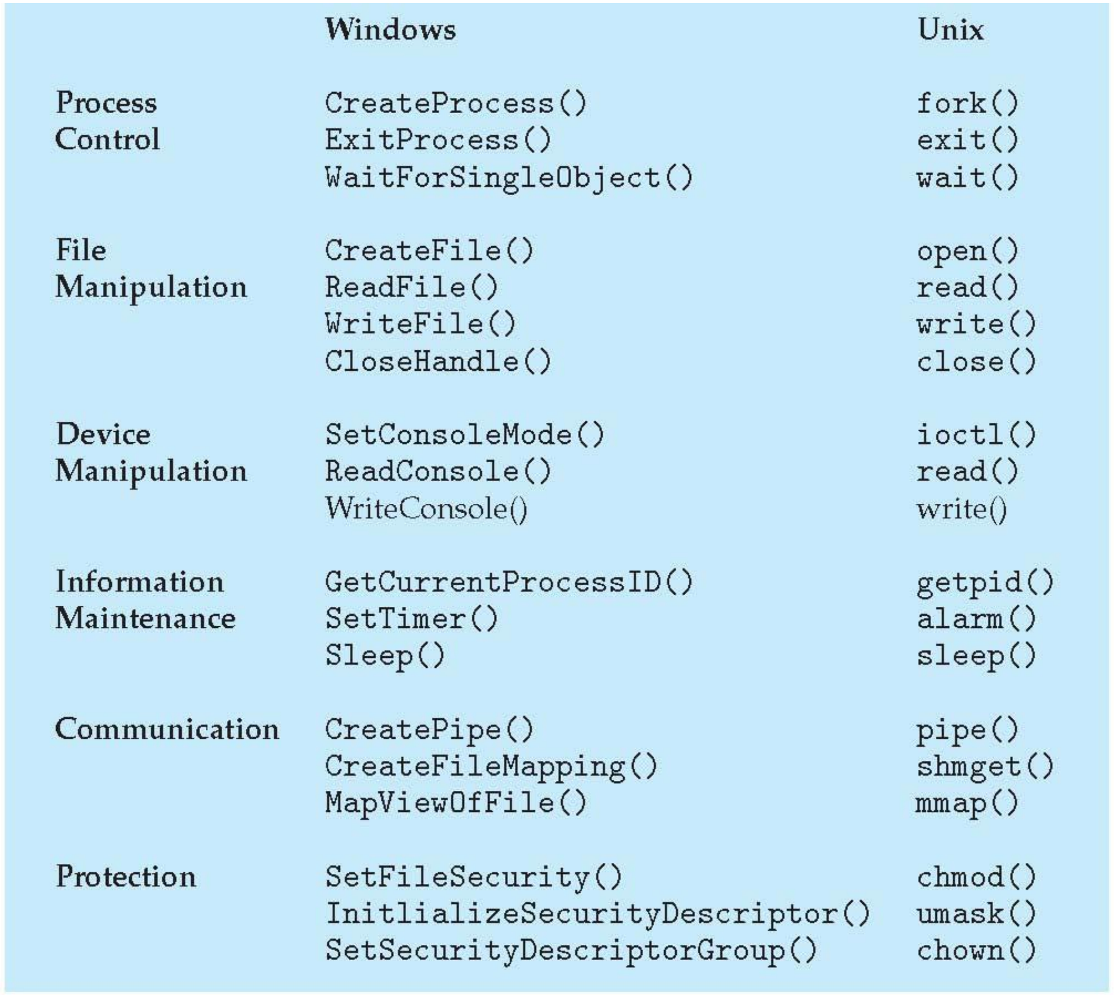


Types of System Calls (Cont.)

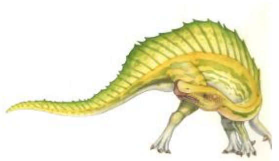
- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access



Examples of Windows and Unix System Calls

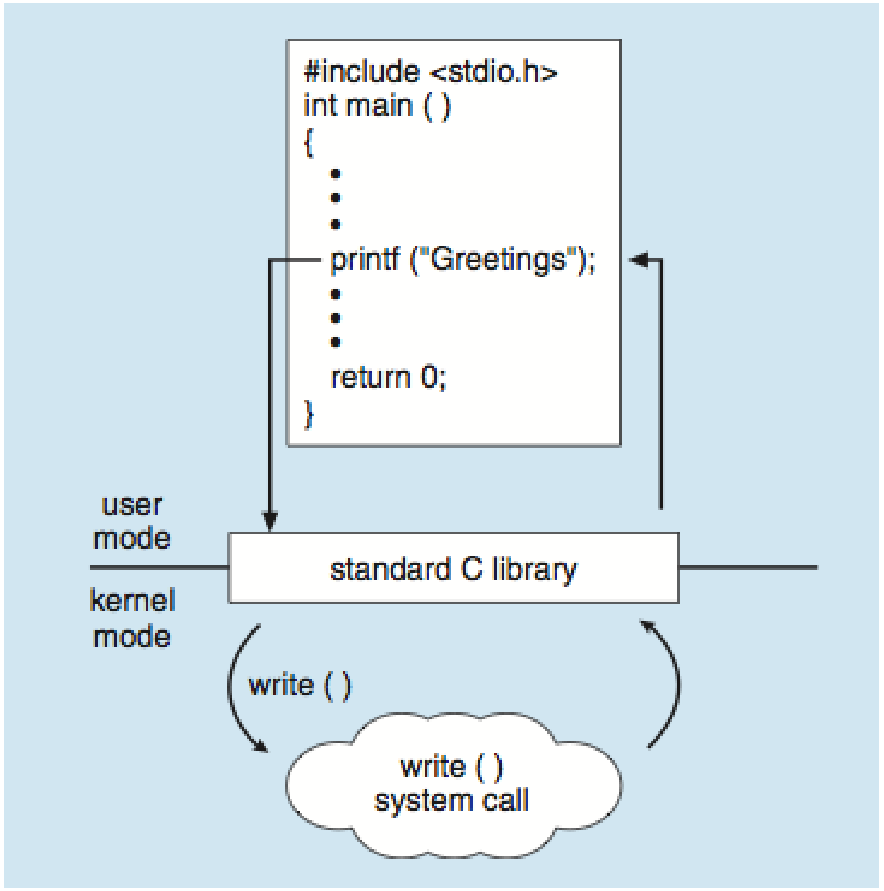


	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information 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



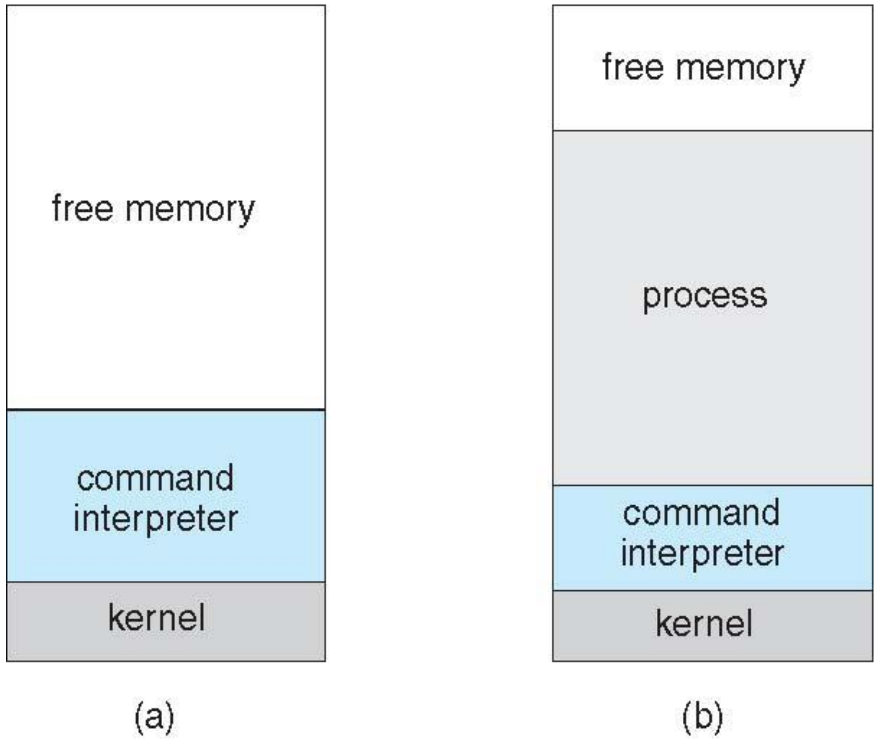
```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

OS 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
  - Modules Approach

Example: MS-DOS

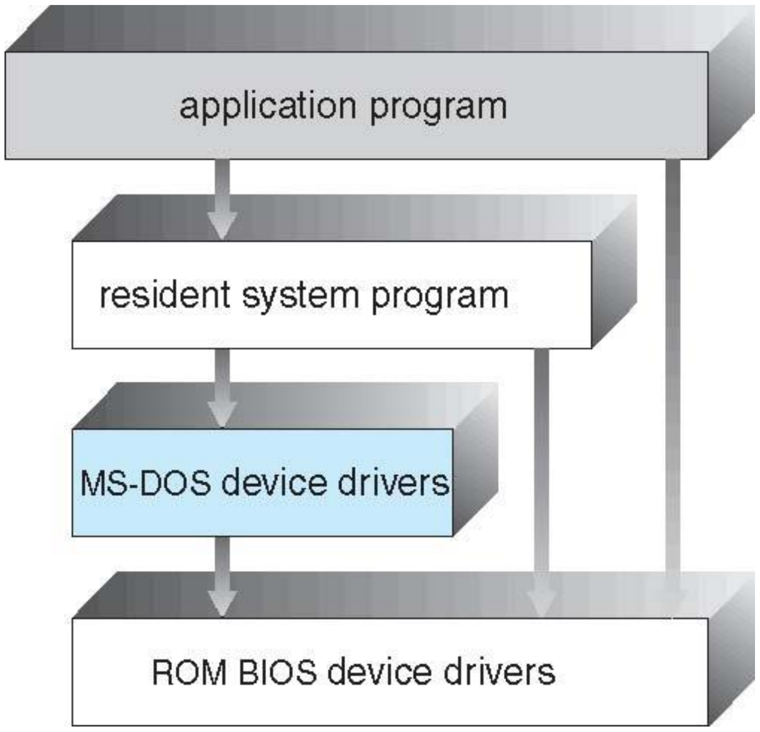
- Single-tasking
- Shell invoked when system booted
- Simple method to run program
  - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded

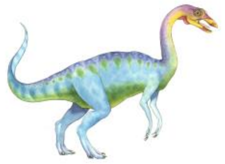


At system startup running a program

Simple Structure -- MS-DOS

- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

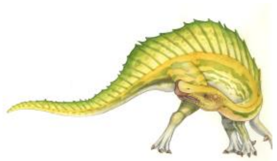




Non Simple Structure -- UNIX

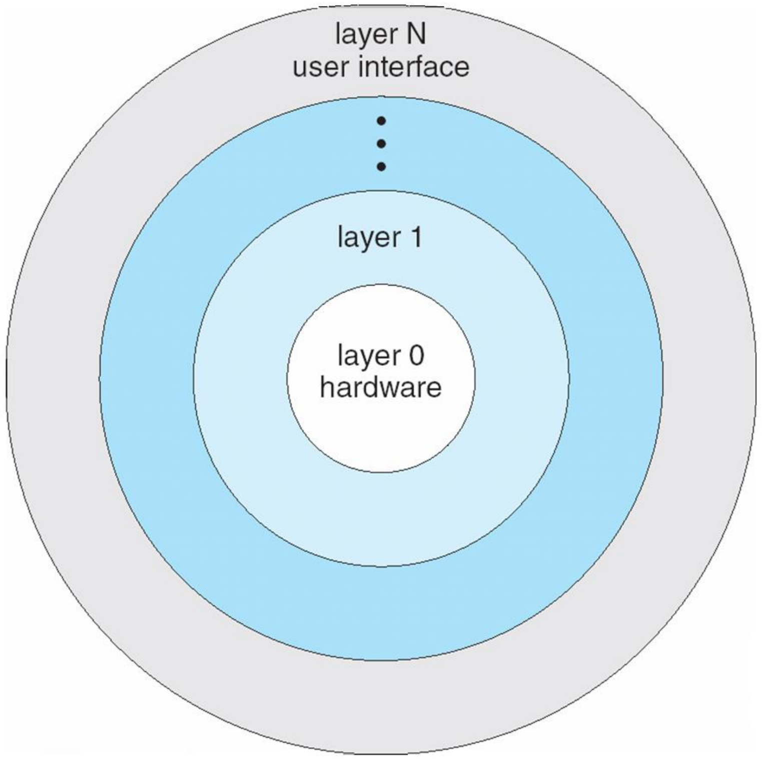
UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts

- 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



Layered Approach

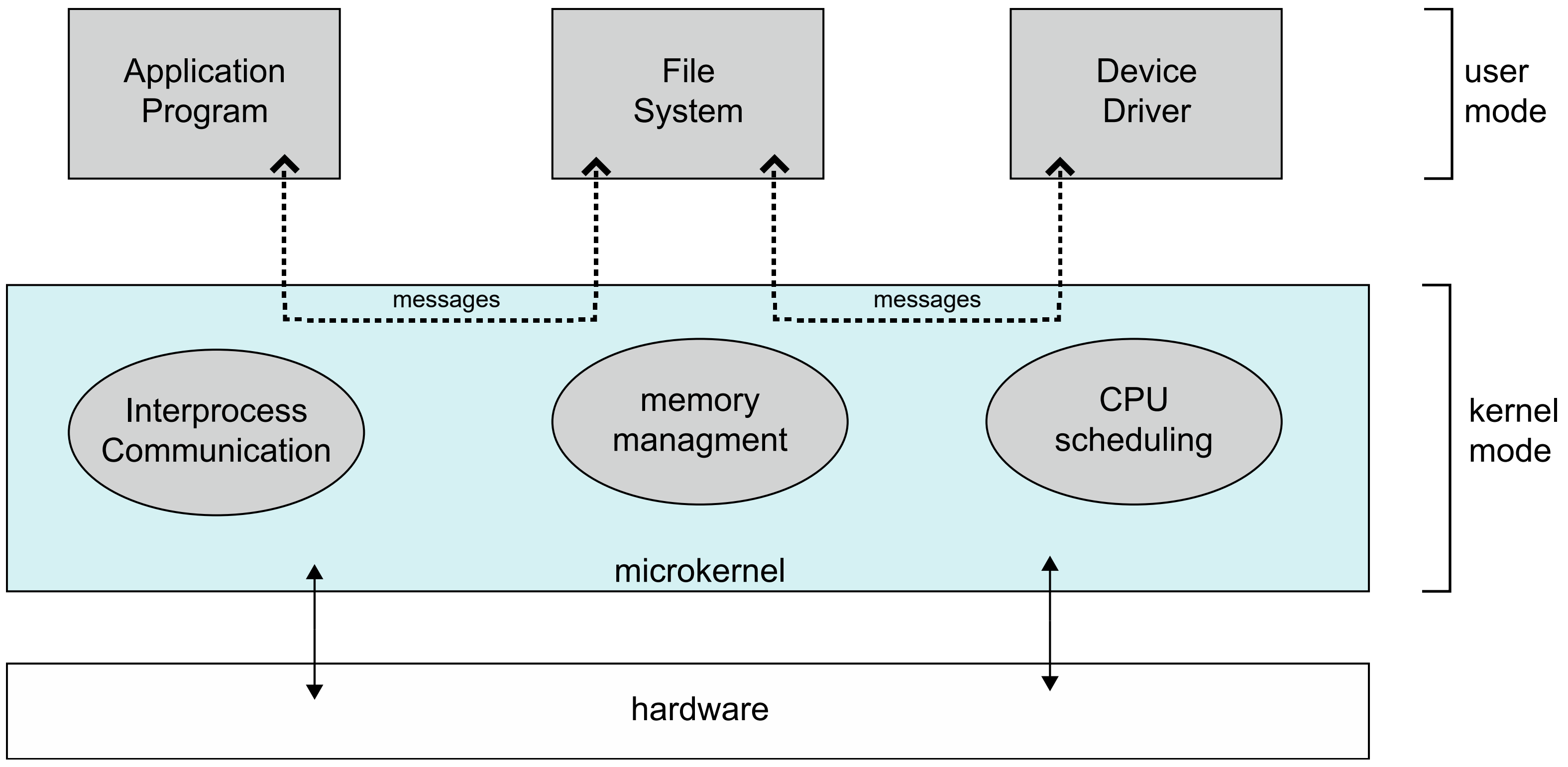
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level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



Microkernel System Structure

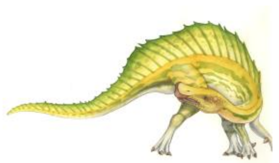
- Moves as much from the kernel into user space
- **Mach** example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system 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



Modules

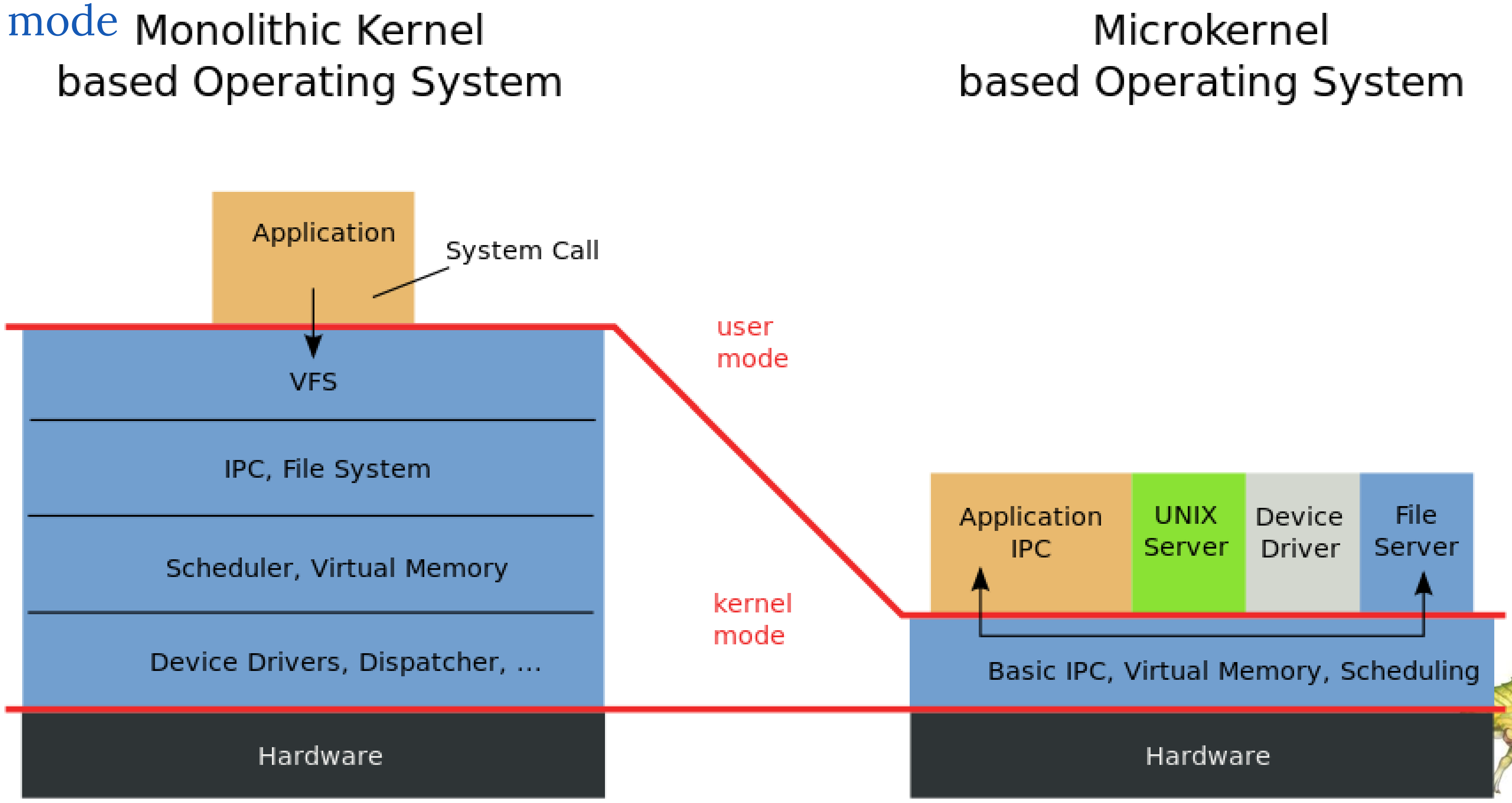
- Many modern operating systems implement **loadable kernel modules. In computing, a loadable kernel module (LKM) is an object file that contains code to extend the running kernel, or so-called base kernel, of an operating system.**
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is **loadable** as needed within the kernel
- Overall, similar to layers but with more flexible
  - Most current Unix-like systems and Microsoft Windows support loadable kernel modules, although they might use a different name for them, such as
    - kernel loadable module (kld) in FreeBSD,
    - kernel extension (kext) in macOS,
    - kernel extension module in AIX,
    - kernel-mode driver in Windows NT and downloadable kernel module (DKM) in VxWorks.
    - They are also known as kernel loadable modules (or KLM), and simply as kernel modules (KMOD).
- Disadvantage : **fragmentation penalty!**



Monolithic vs Microkernel

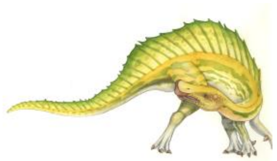
A **monolithic kernel** is an operating system architecture where the entire operating system is working in kernel space and is alone in supervisor mode

A **microkernel** (also known as **μ-kernel**) is the near-minimum amount of software that can provide the mechanisms needed to implement an operating system (OS)


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

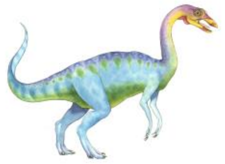
Hybrid Systems

- **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**)



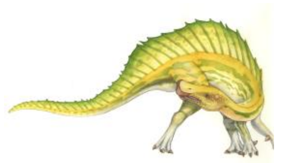
Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- Failure analysis
  - Log files
    - OS generate **log files** containing error information
  - Core dump
    - **Failure of an application** can generate **core dump** file capturing **memory of the process**
  - Crash dump
    - **Operating system failure** can generate **crash dump** file **containing kernel memory**
- ***Kernighan's Law***
  - "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."



Operating-System Debugging

- Performance tuning
  - Monitor system performance
    - Add code to kernel
    - Use system tools like “top”



End of Chapter 2