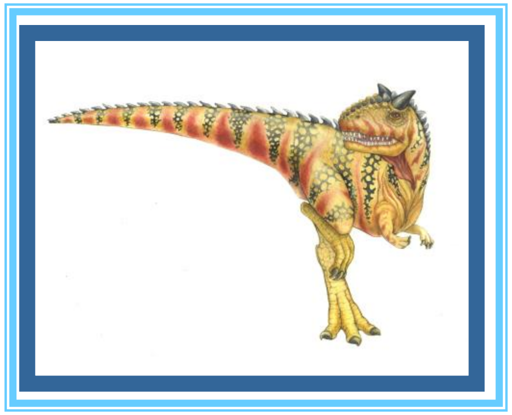
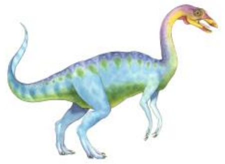
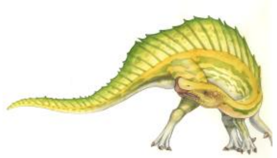
Chapter 2: Operating-System Structures





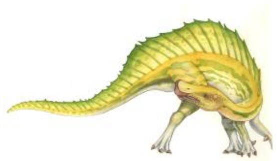
Chapter 2: Operating-System Structures

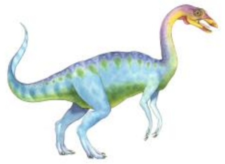
- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Virtual Machines
- Operating System Debugging
- Operating System Generation
- System Boot



Objectives

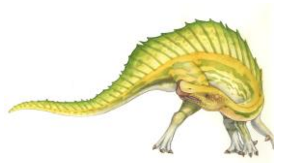
- To describe the services an operating system provides to users, processes, and other systems
- To discuss the various ways of structuring an operating system
- To explain how operating systems are installed and customized and how they boot





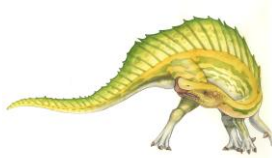
Operating System Services

- An OS provides an environment for the execution of programs.
- OS services helpful to user:
  - User interface - Graphical UI or command line
  - 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
  - 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
  - Error detection – OS needs to be constantly aware of possible errors

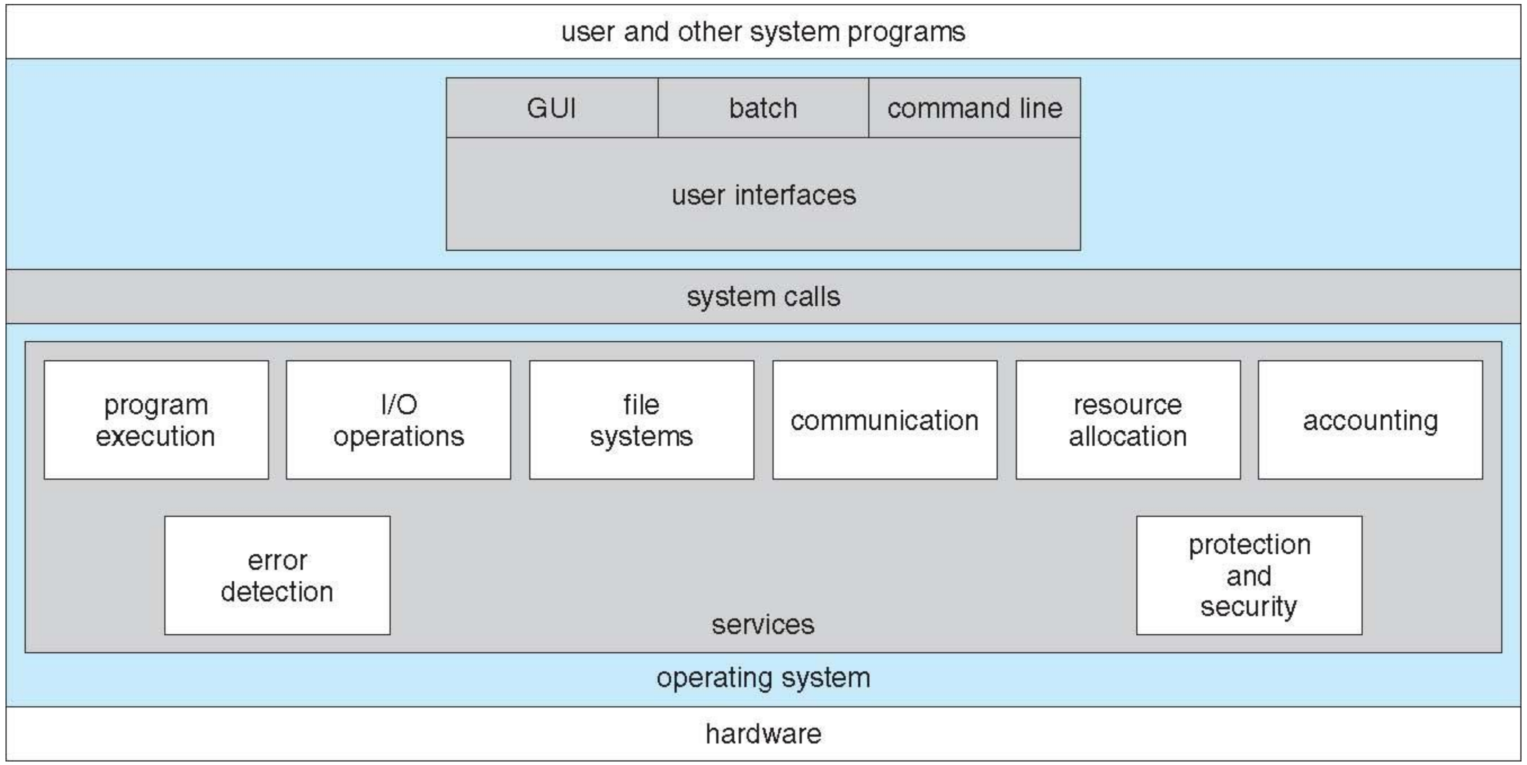


Operating System Services (Cont)

- OS services for efficient operation:
  - **Resource allocation - e.g.** CPU cycles, main memory, and file storage
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security**



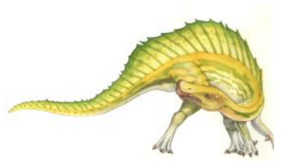
A View of Operating System Services



User Operating System Interface - CLI

- Command Line Interface (CLI) or command interpreter allows direct command entry
    - Sometimes implemented in kernel, sometimes by systems program
    - Sometimes multiple flavors implemented – shells (.bashrc on zeus)
    - Primarily fetches a command from user and executes it
- Many systems now include both CLI and GUI interfaces
    - Microsoft Windows is GUI with CLI “command” shell
    - Apple Mac OS X as “Aqua” GUI interface with UNIX kernel underneath and shells available
    - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

1. How do we get to the command interpreter? Windows? Linux?
2. How do we modify the command interpreter environment for our own specific needs? Linux?
3. Putty to zeus … create an alias … add . to PATH

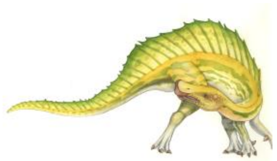


System Calls

- 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 Program Interface (API) rather than direct system call use
- Three most common APIs are:
  - Win32 API for Windows
  - POSIX (Portable Operating System Interface) API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)

1. Why use APIs rather than system calls?

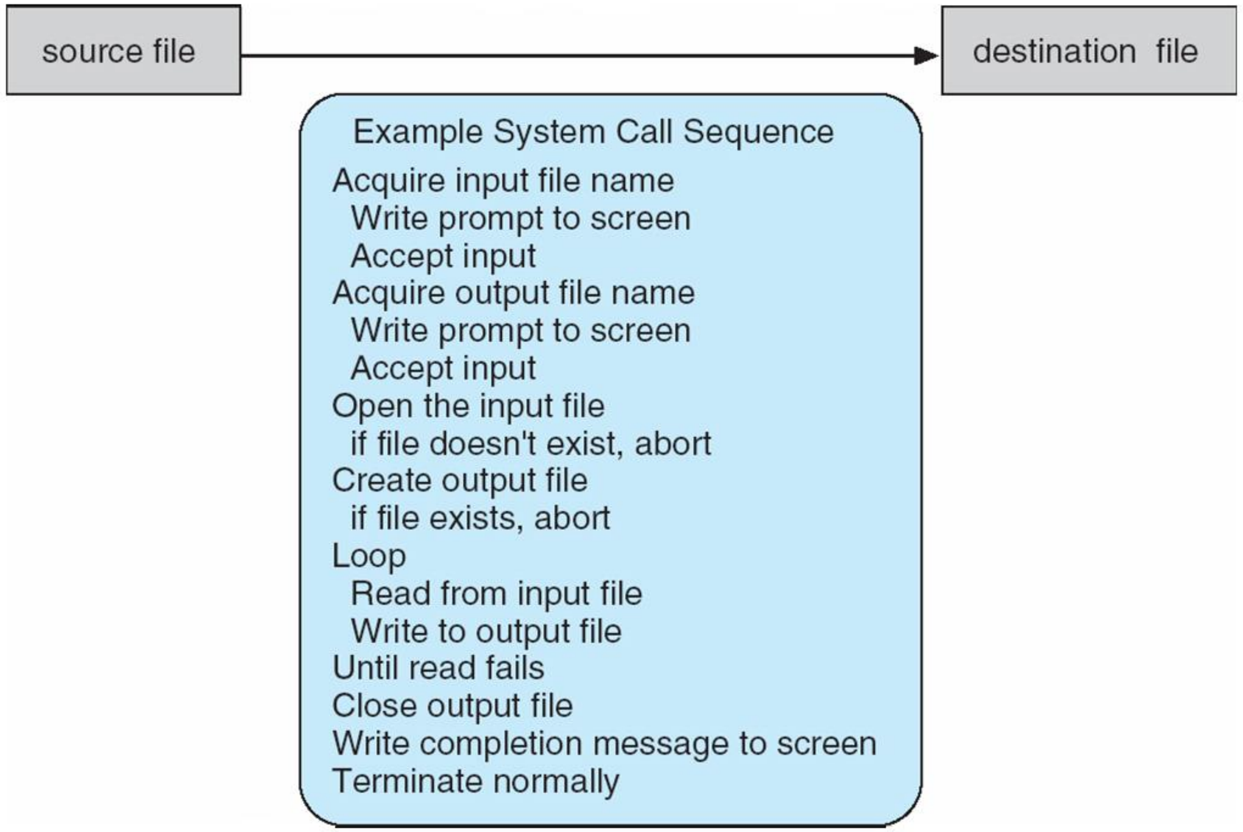
(Note that the system-call names used throughout this text are generic)



Example of System Calls

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

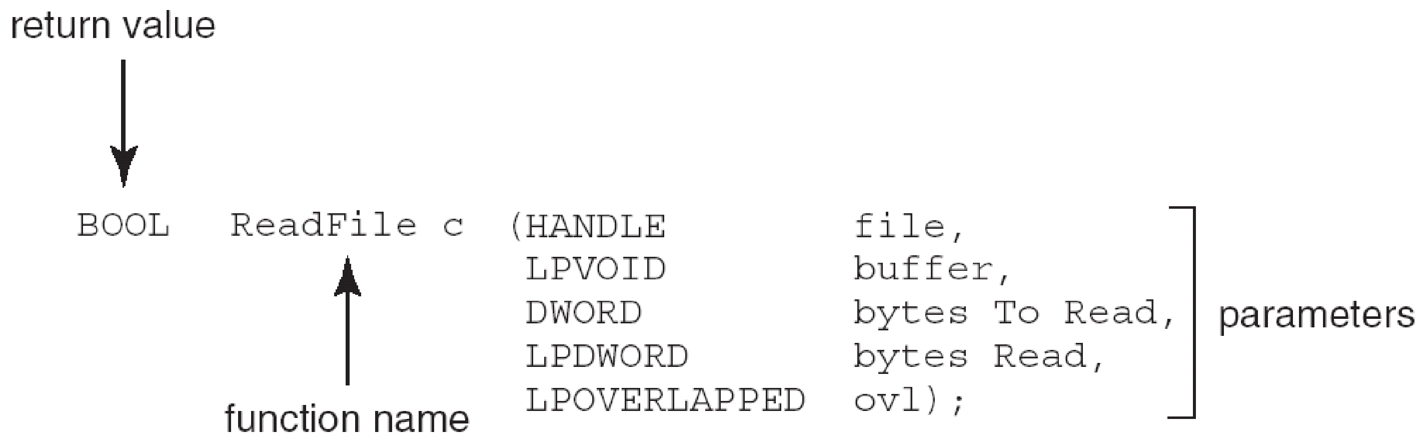
1. Where do the system calls come from?
2. Is any API used? If so, what? If not, why not?



```
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
```

Example of Standard API

- Consider the ReadFile() function in the
- Win32 API—a function for reading from a file

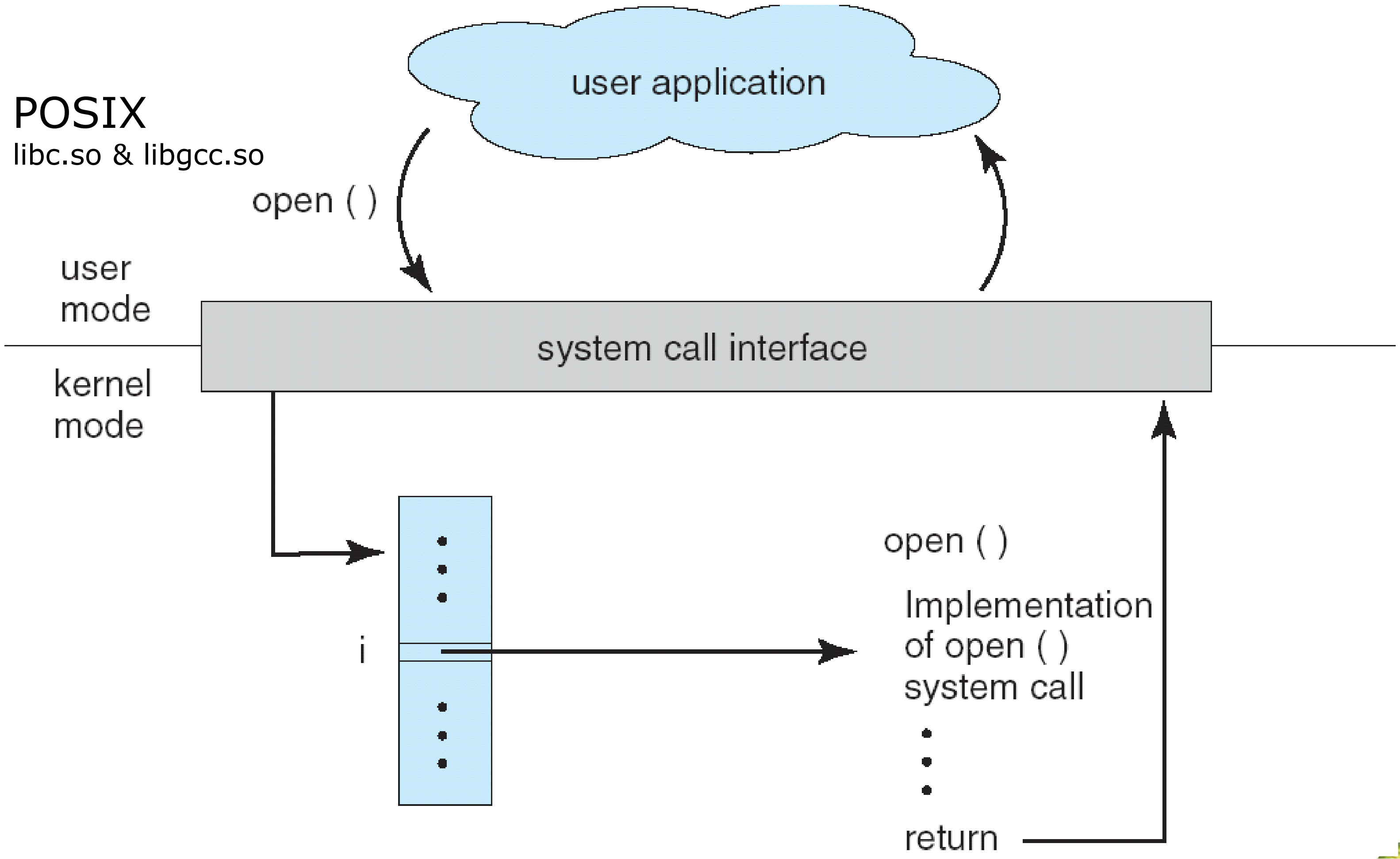


```
BOOL   ReadFile c  (HANDLE         file,
                    LPVOID         buffer,
                    DWORD          bytes To Read,
                    LPDWORD        bytes Read,
                    LPOVERLAPPED   ovl);
```

- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

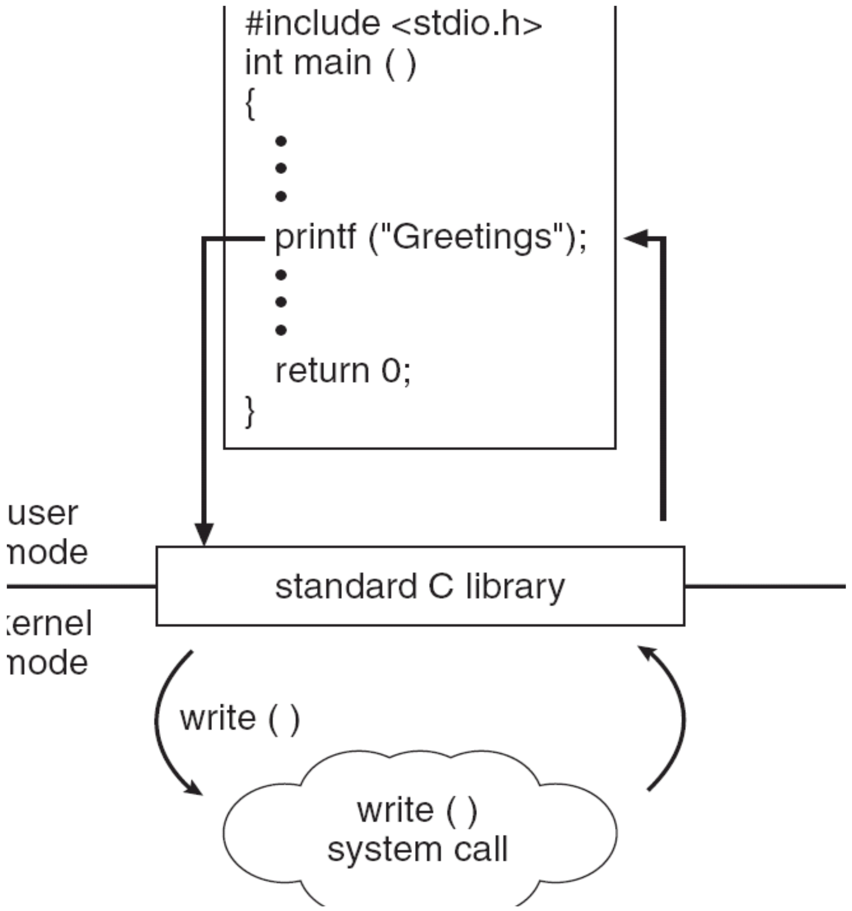
API – System Call – OS Relationship

POSIX
libc.so & libgcc.so

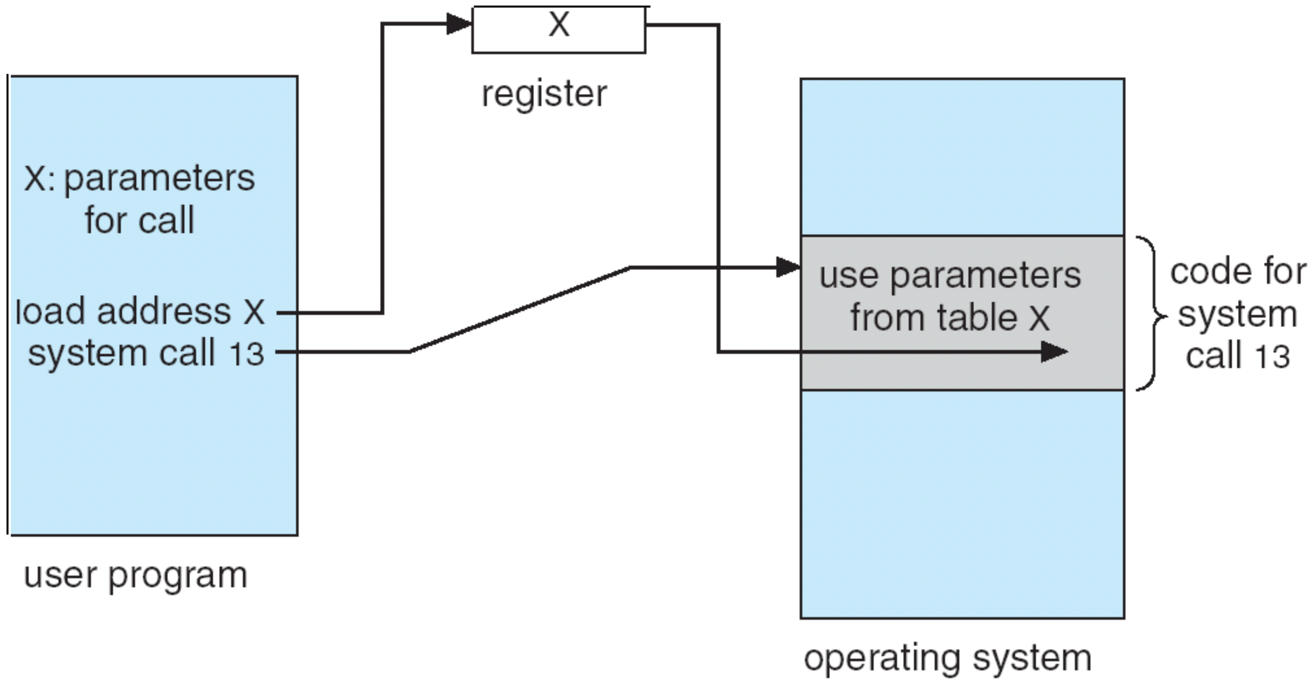


Standard C Library Example

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



Parameter Passing via Table

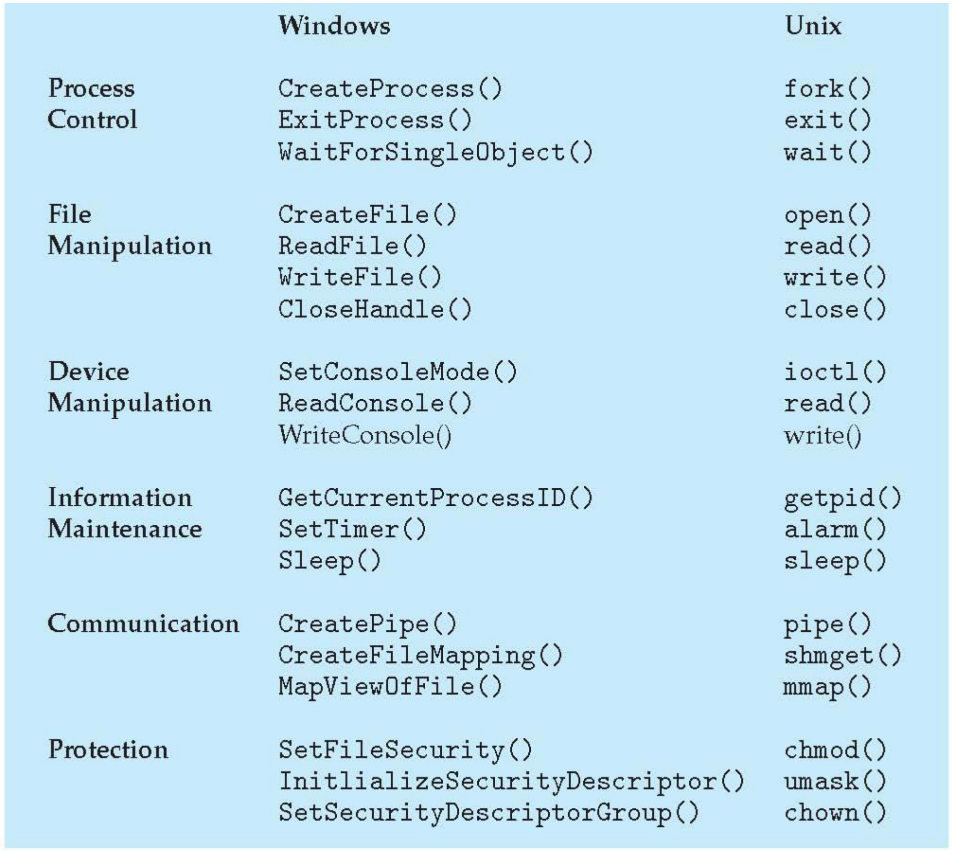


Three parameter passing methods to pass data to a system call:
- Registers -Block of Memory -Stack

1. What are advantages and disadvantages of each?

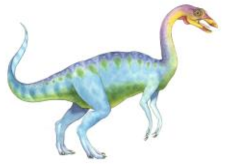
Examples of Windows and Unix System Calls

Types of system calls



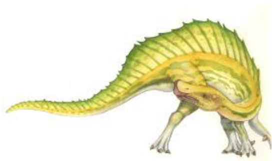
	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

1. What’s a pipe? How would you use a pipe?
2. Has anyone used chmod or chown? How?

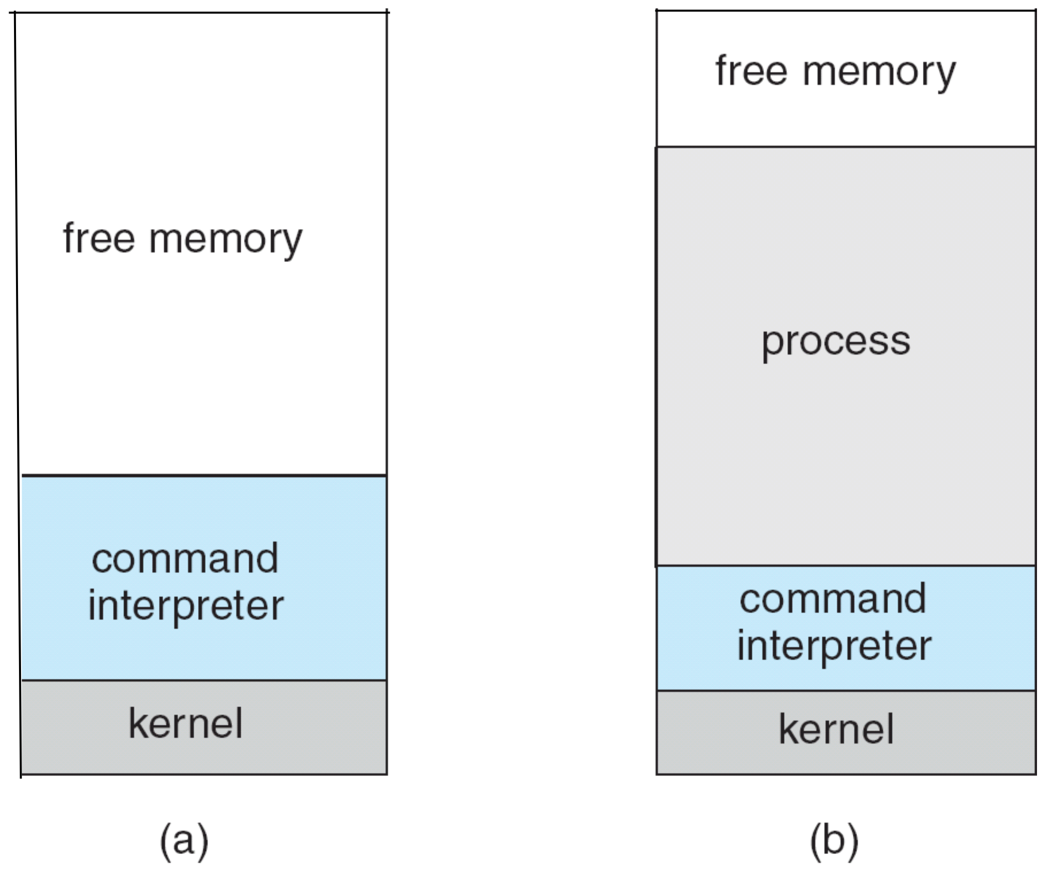


Single-tasking OS

- MS-DOS OS is single-tasking
  1. Command interpreter invoked when computer is started
  2. To run a program:
     a. Load program into memory writing over portion of OS for more space
     b. Program is run until completed or error causes trap
     c. Portion of command interpreter left is executed and reloads OS that was kicked out



MS-DOS execution



(a) At system startup (b) running a program

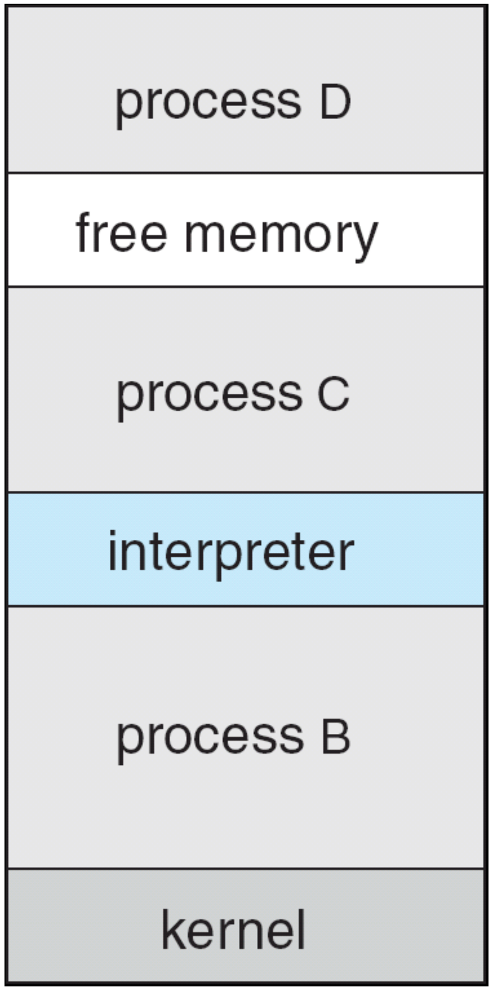
Multitasking OS

- FreeBSD OS is multitasking
  1. User logs on and shell is run
  2. Program is loaded and executed BUT command interpreter can continue running simultaneously
  3. To run a program
     a. fork () a new process
     b. Load program into MEM via exec ()
     c. Program can be run in foreground or background
     d. exit () terminate process

1. What is the meaning of foreground or background?

2. How to run a program in the background?

FreeBSD Running Multiple Programs



System Programs

- System programs:
  - are not necessarily part of the OS
  - provide a convenient environment for program development and execution. They can be divided into:
    - File management (e.g.
    - Status information (e.g.
    - File modification (e.g.
    - Programming language support (e.g.
    - Program loading and execution (e.g.
    - Communications (e.g.
    - Background services (e.g.
- Most users' view of the operation system is defined by system programs, not the actual system calls