

# Chapter 2: System Structures





# System Calls

---

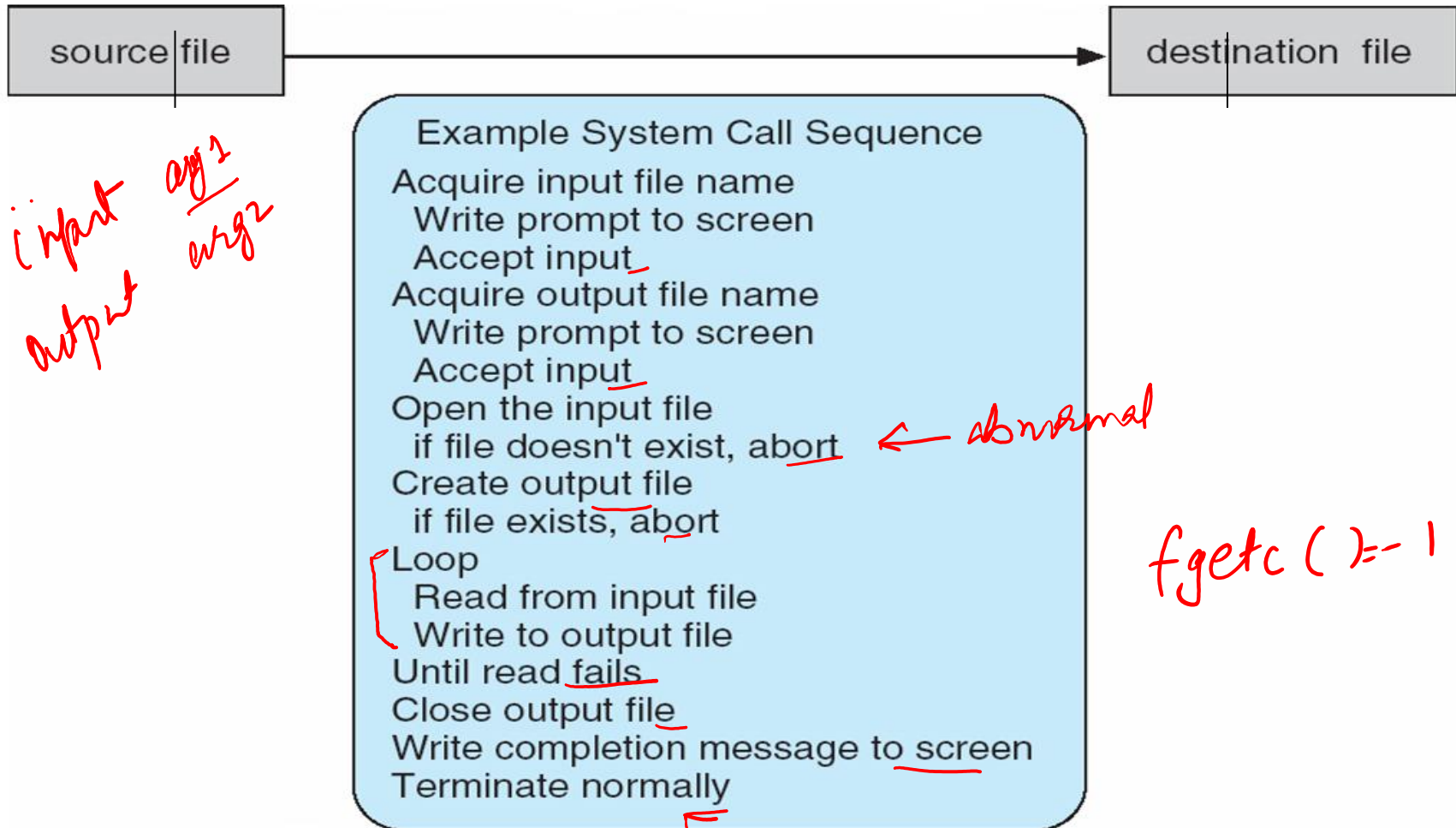
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API 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





# 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<br>value | function<br>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

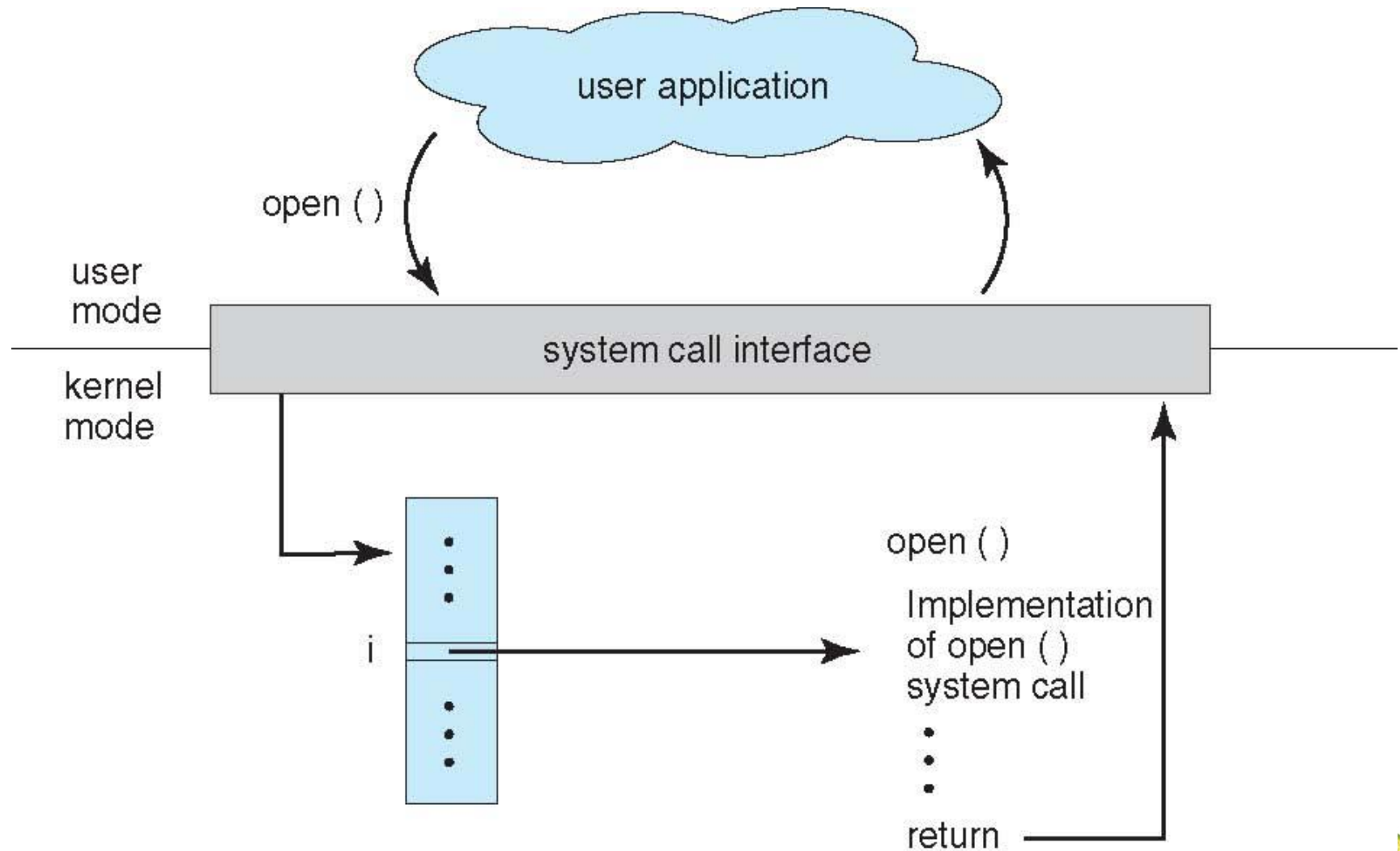
---

- 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





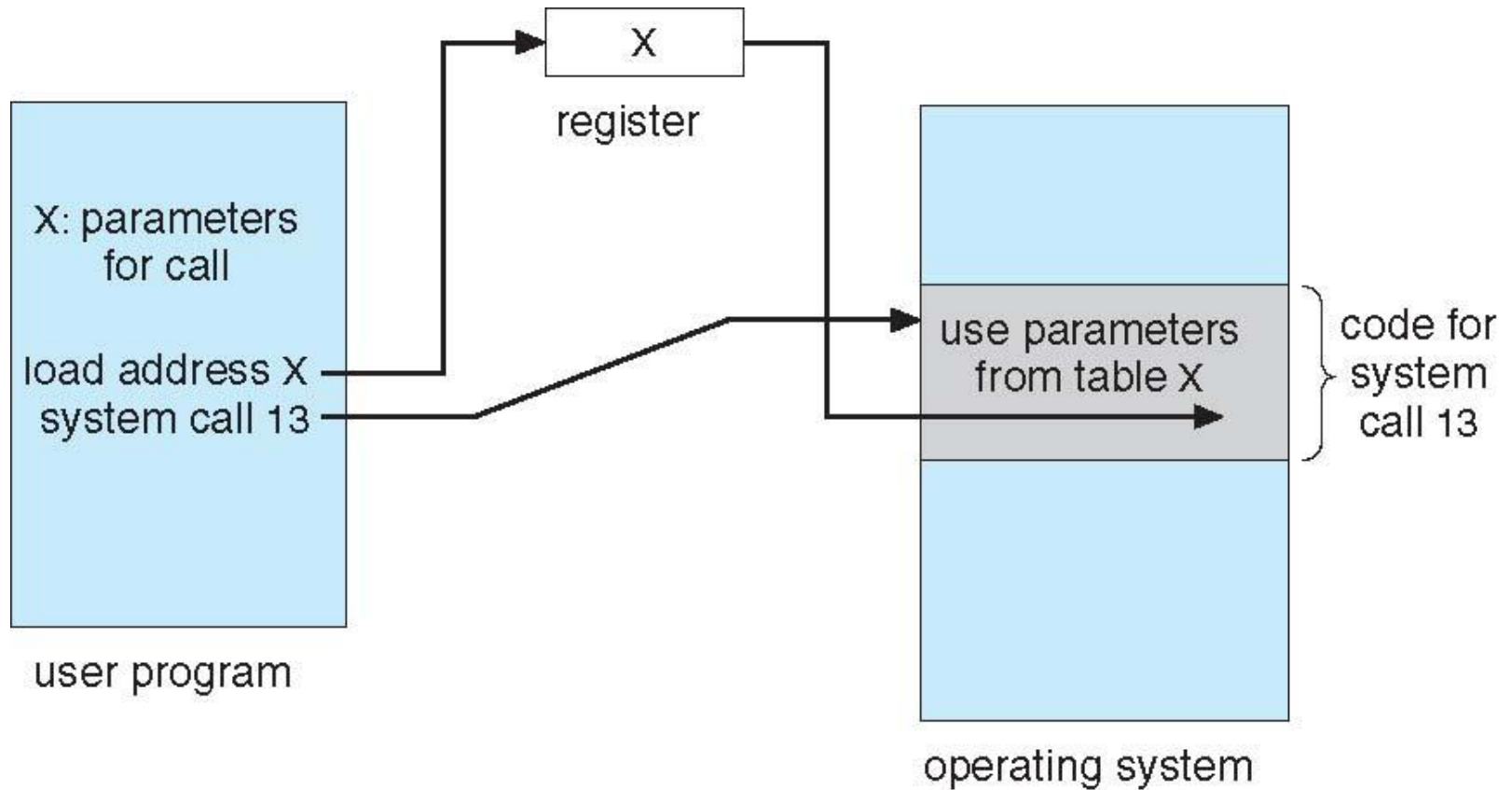
# System Call Parameter Passing

- ❑ 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

---

System Calls can be grouped in SIX major categories:

- ❑ Process control
- ❑ File Management
- ❑ Device Management
- ❑ Information Maintenance
- ❑ Communications
- ❑ Protection





# 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

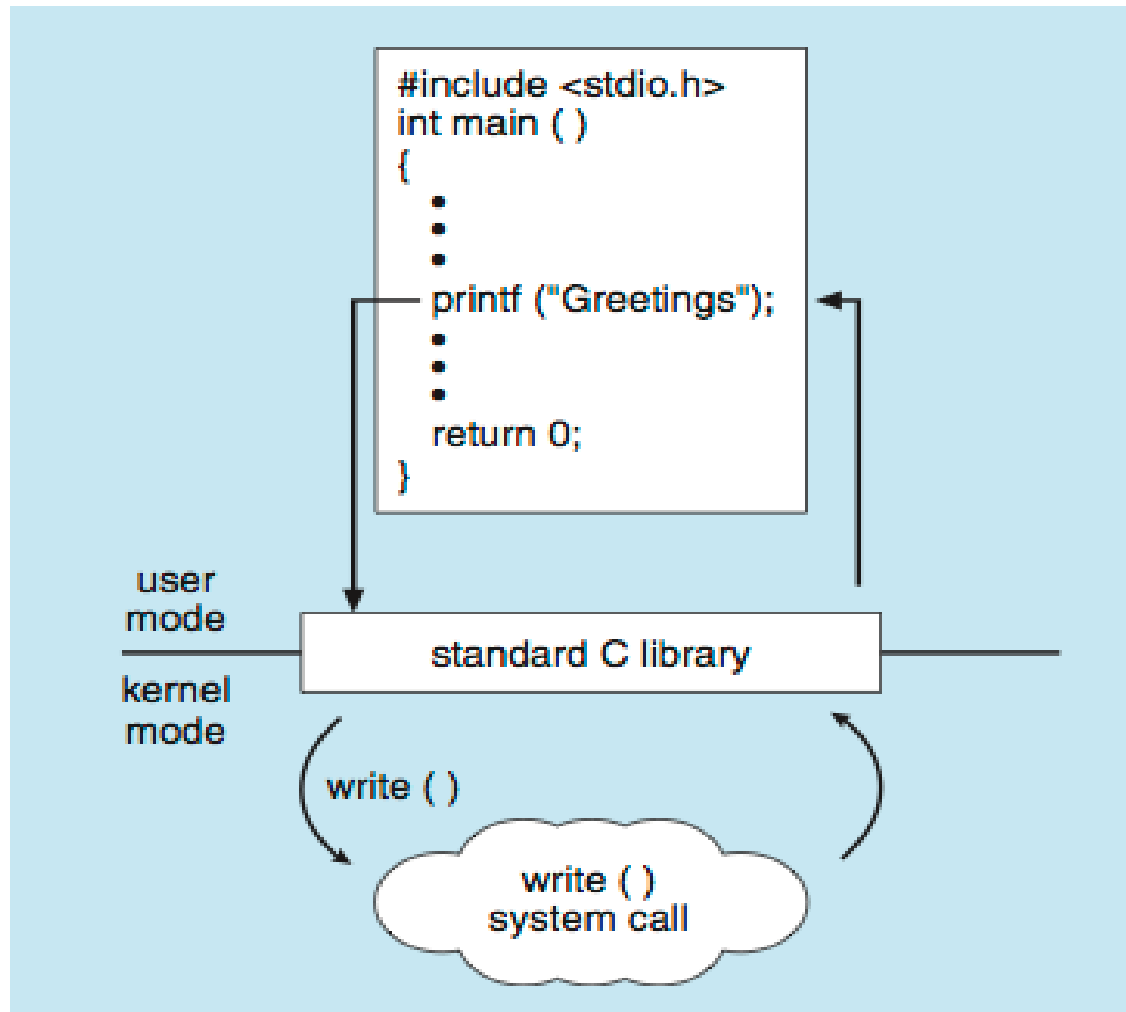
|                                | Windows   | Unix                                   |
|--------------------------------|---|--|
| <b>Process Control</b>         | CreateProcess()<br>ExitProcess()<br>WaitForSingleObject()                           | fork()<br>exit()<br>wait()             |
| <b>File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





# Standard C Library Example

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





# System Programs

---

- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation
  - Status information sometimes stored in a File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls







# System Programs

---

- Provide a convenient environment for program development and execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a **registry** - used to store and retrieve configuration information





# System Programs (Cont.)

---

- ❑ **File modification**
  - ❑ Text editors to create and modify files
  - ❑ Special commands to search contents of files or perform transformations of the text
- ❑ **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- ❑ **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- ❑ **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - ❑ Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another





# System Programs (Cont.)

---

## □ Background Services

- Launch at boot time
  - ▶ Some for system startup, then terminate
  - ▶ Some from system boot to shutdown
- Provide facilities like disk checking, process scheduling, error logging, printing
- Run in user context not kernel context
- Known as **services**, **subsystems**, **daemons**

## □ Application programs

- Don't pertain to system
- Run by users
- Not typically considered part of OS
- Launched by command line, mouse click



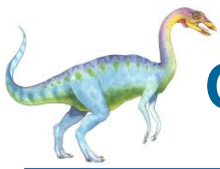


# Operating System Design and Implementation

---

- Design and Implementation of OS 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





# Operating System Design and Implementation (Cont.)

---

- Important principle to separate

**Policy:** *What* will be done?

**Mechanism:** *How* to do it?

- 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)
- Specifying and designing an OS is highly creative task of **software engineering**





# 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





# Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- OS generate **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
  - Sometimes using **trace listings** of activities, recorded for analysis
  - **Profiling** is periodic sampling of instruction pointer to look for statistical trends

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 Generation

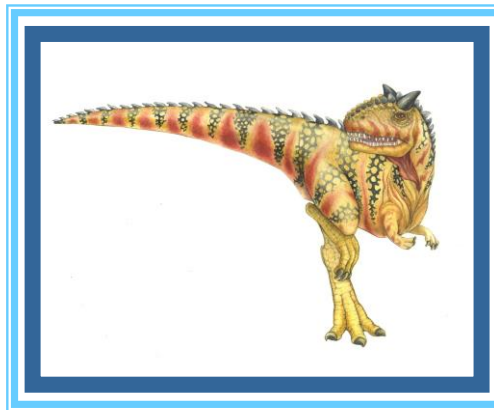
---

- n Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- n **SYSGEN** program obtains information concerning the specific configuration of the hardware system
  - | Used to build system-specific compiled kernel or system-tuned
  - | Can generate more efficient code than one general kernel



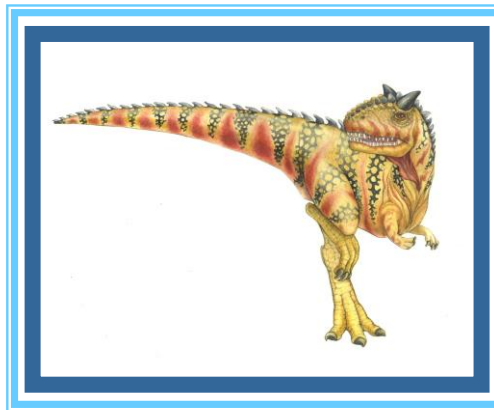
# End of Chapter 2

---



# Chapter 3: Processes

---





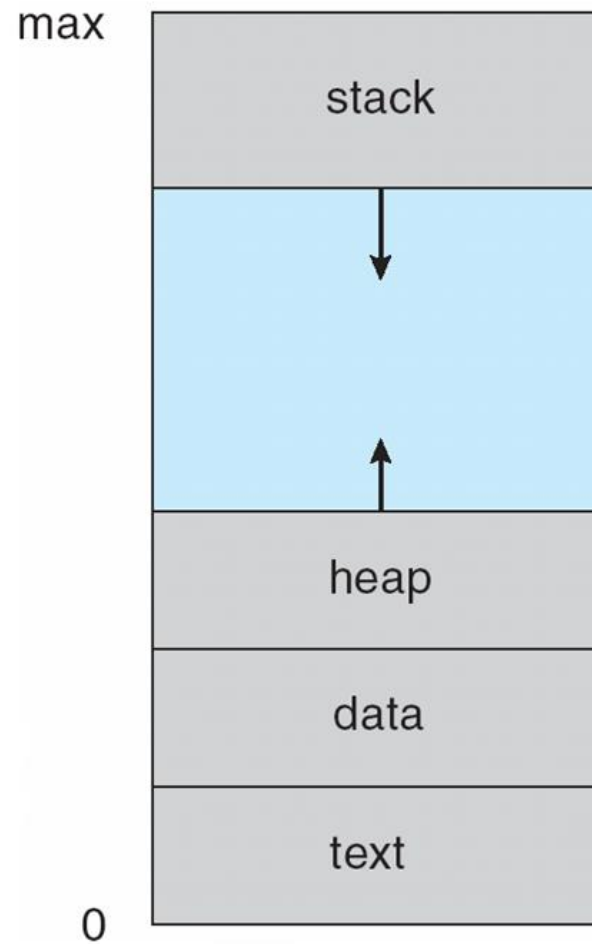
# Process Concept

- ❑ An operating system executes a variety of programs:
  - ❑ Batch system – **jobs**
  - ❑ Time-shared systems – **user programs** or **tasks**
- ❑ Textbook uses the terms **job** and **process** almost interchangeably
- ❑ **Process** – a program in execution; process execution must progress in sequential fashion
- ❑ Multiple parts
  - ❑ The program code, also called **text section**
  - ❑ Current activity including **program counter**, processor registers
  - ❑ **Stack** containing temporary data
    - ▶ Function parameters, return addresses, local variables
  - ❑ **Data section** containing global variables
  - ❑ **Heap** containing memory dynamically allocated during run time





# Process in Memory





# Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**), process is **active**
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program
    - ▶ Web browser is opened in multiple windows
  - Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary.





# Process State

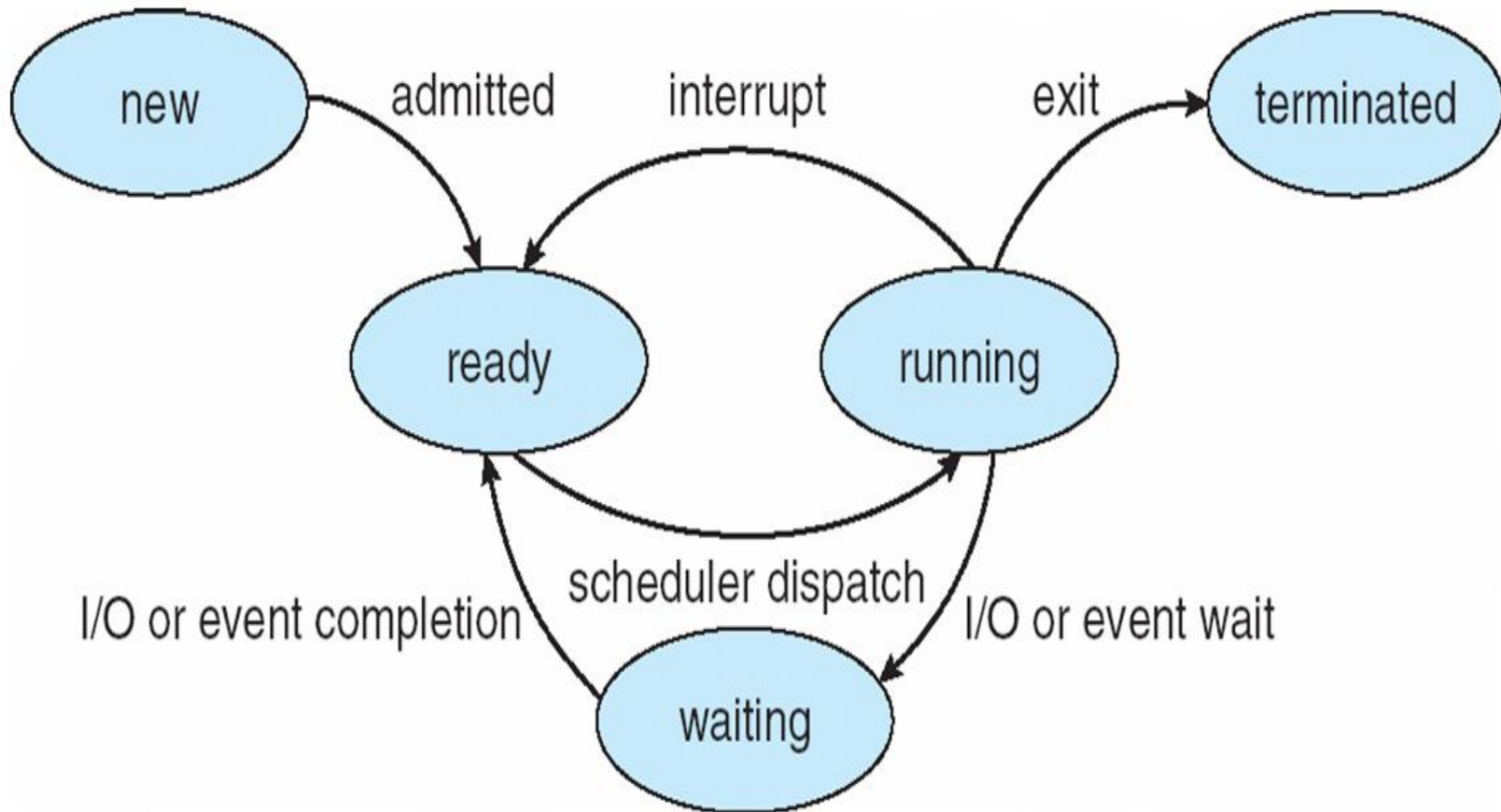
---

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution





# Diagram of Process State



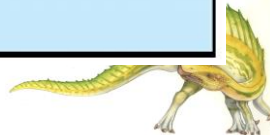
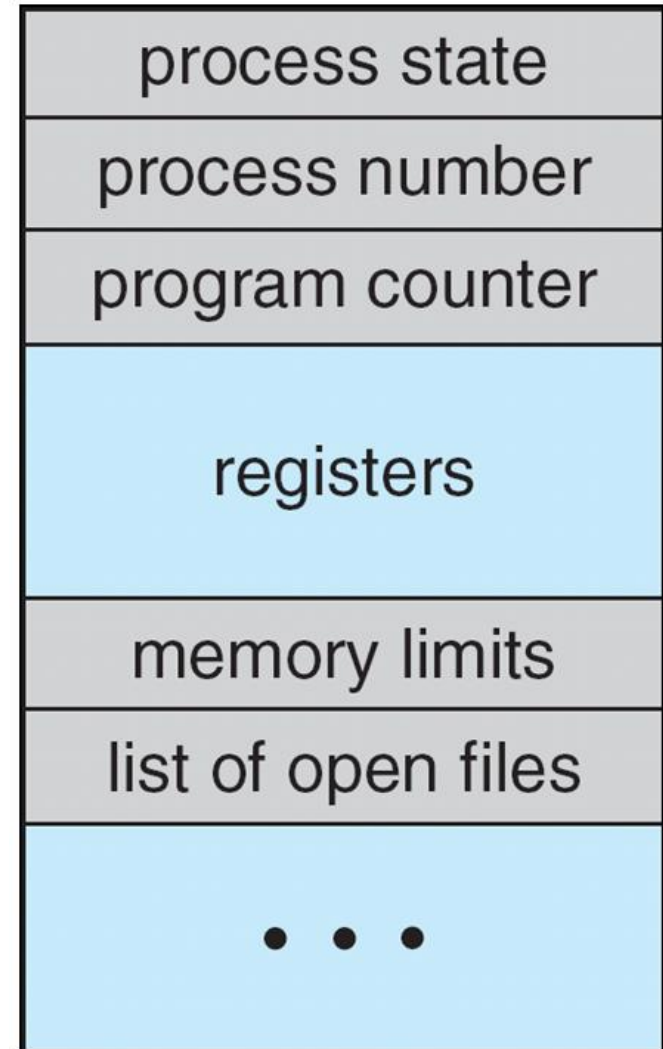




# Process Control Block (PCB)

Information associated with each process  
(also called **task control block**)

- ❑ Process state – running, waiting, etc
- ❑ Program counter – location of instruction to next execute
- ❑ CPU registers – contents of all process-centric registers
- ❑ CPU scheduling information- priorities, scheduling queue pointers
- ❑ Memory-management information – memory allocated to the process
- ❑ Accounting information – CPU used, clock time elapsed since start, time limits
- ❑ I/O status information – I/O devices allocated to process, list of open files





# CPU Switch From Process to Process

