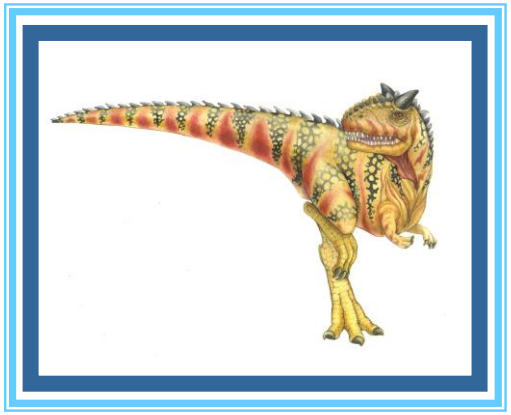
# Chapter 2: Operating-System Structures

Reading: Chapter 2.1-2.3



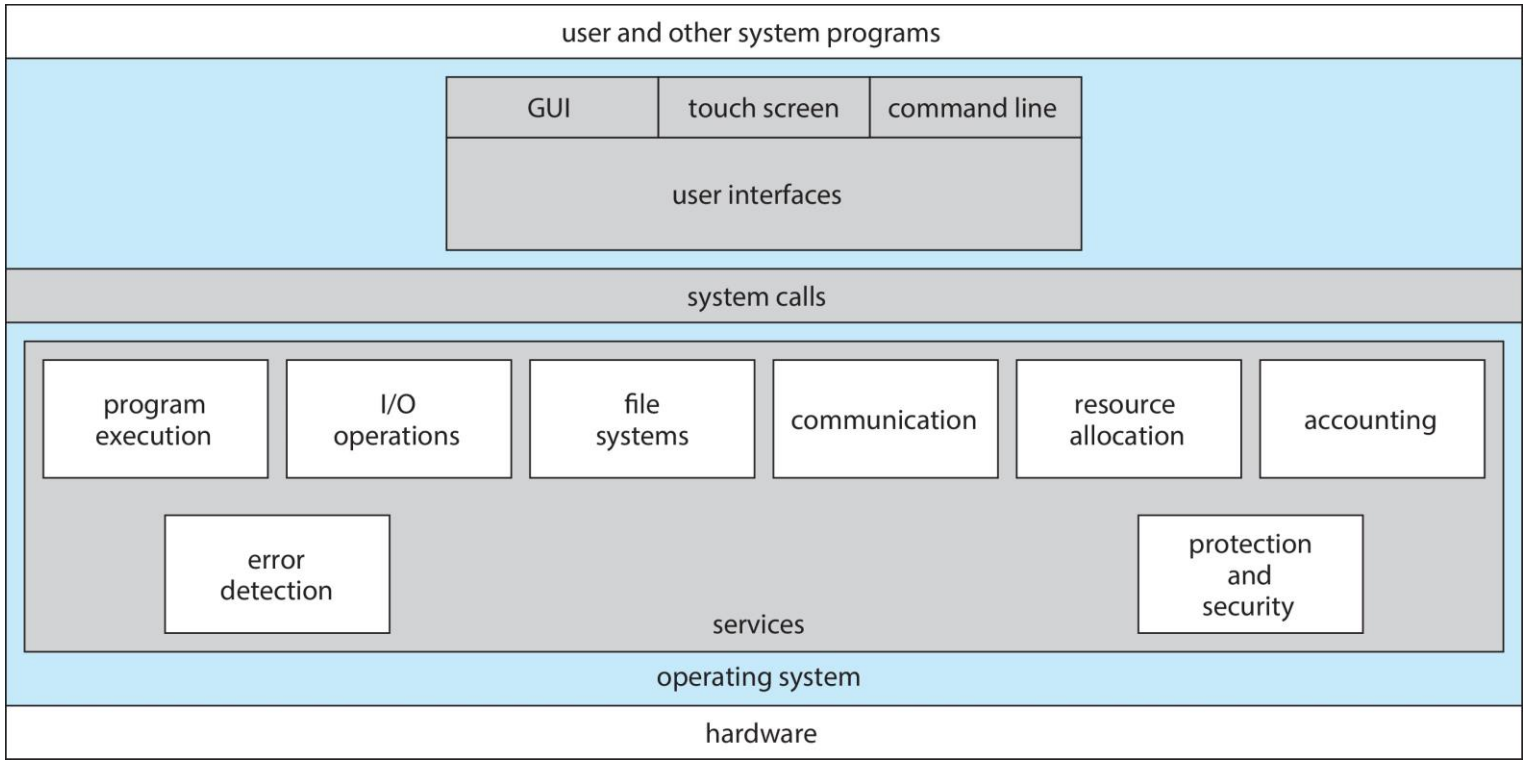
# Operating System Services

- An operating system provides services to programs and users
- One set of OS services provides functions helpful to the user
  - **User interface – command-line interface (or shell), graphical user interface (GUI), touch-screen**
  - **Program execution** - OS must be able to load a program into memory, run that program, and end program execution
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
  - **File-system manipulation** (e.g., read/write files and directories, create/ delete files and directories)
  - **Communications** – Processes may exchange information via shared memory or through message passing
  - **Error detection**
    - Errors may occur in hardware or in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing

# Operating System Services (Cont.)

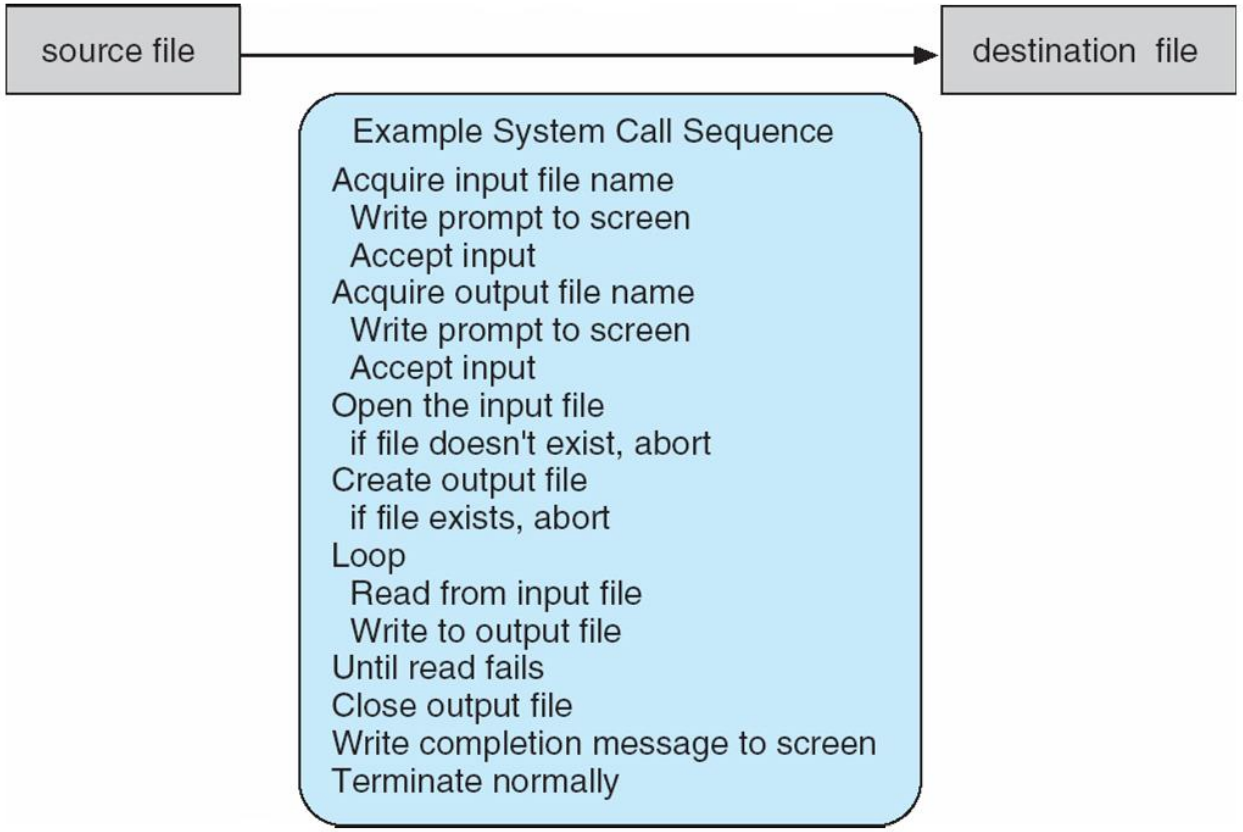
- Another set of OS functions ensure the efficient operation of the system itself
  - **Resource allocation -** When multiple processes run concurrently, resources must be allocated to each of them
  - **Logging -** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security**
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** is to defend a system from external attacks

# A View of Operating System Services




# System Calls

- System calls provide an interface to the services provided by the OS
  - Generally available as routines written in C or C++
- 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)
```

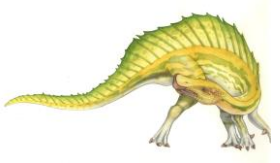
return value — function name — parameters

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

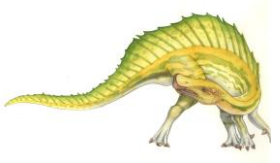
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

Type "man 2 read" to look up read in section 2 of man page



# Application Programming Interface

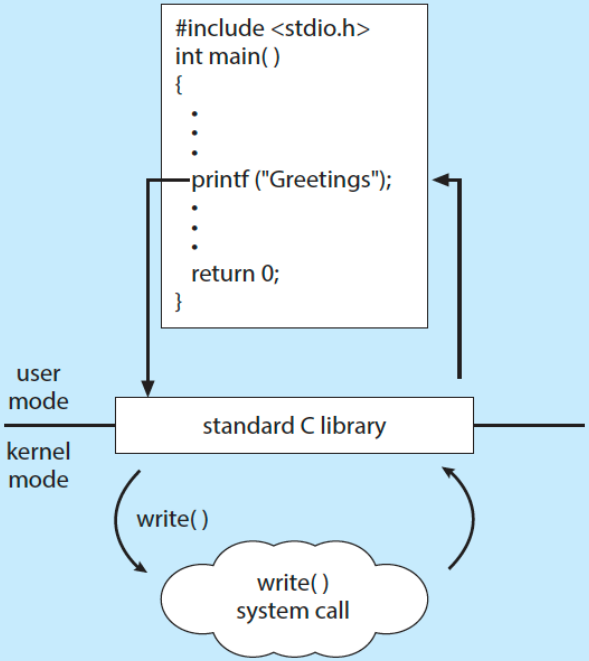
- Programmers mostly access systems calls via a high-level **Application Programming Interface (API)** rather than direct system call use
  - The API specifies a set of library functions available to application programmers
  - The library functions typically invoke system calls
- Three most common APIs
  - Windows API for Windows systems
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and macOS)
  - Java API for programs that run on the Java virtual machine (JVM)



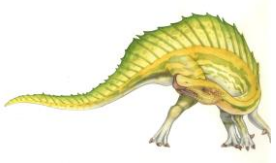
# Standard C Library Example

*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:

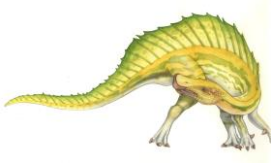



A C program invoking printf() library call, which invokes write() system call

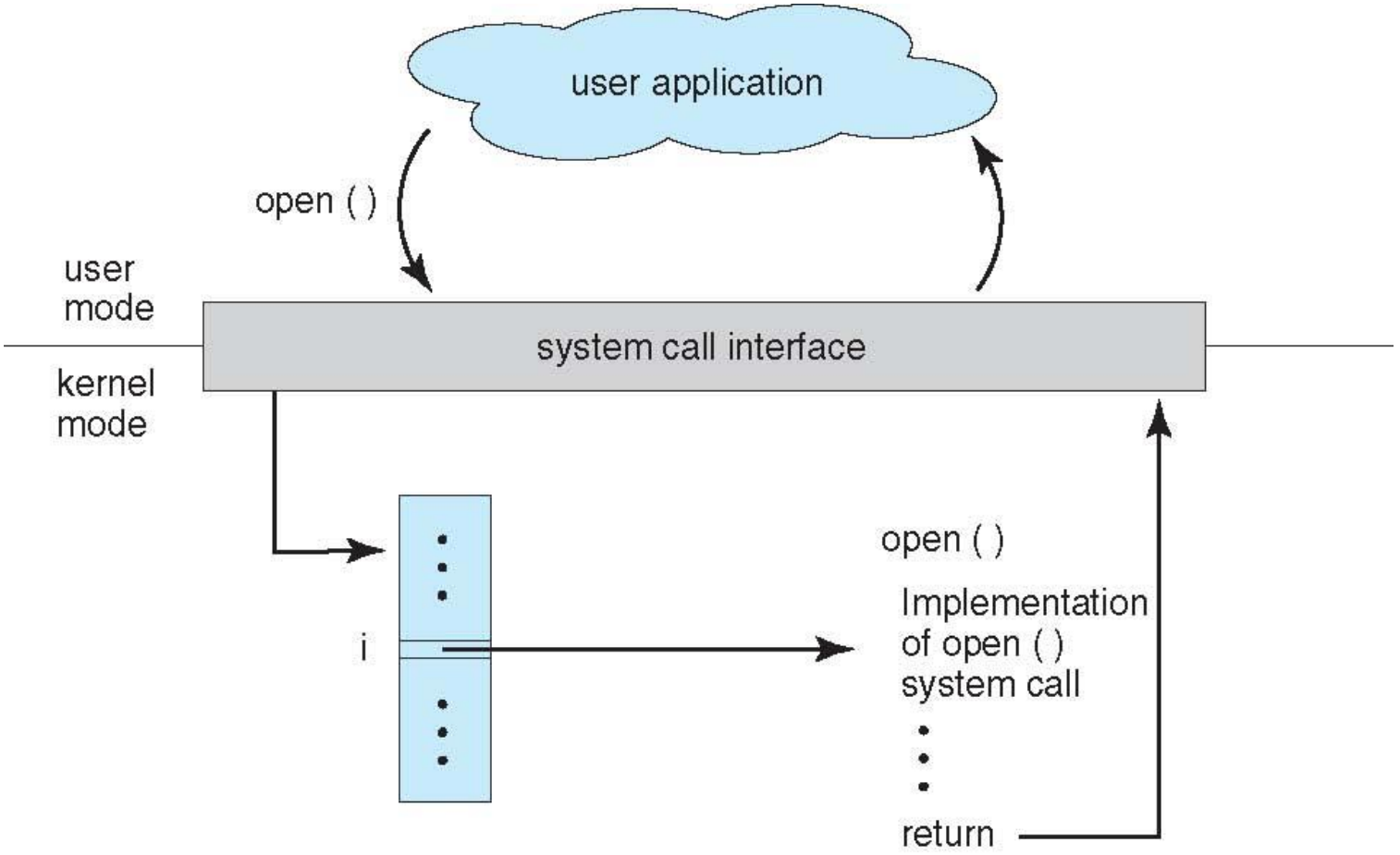


# System Call Implementation

- When a user program makes a system call, it triggers a trap
- Typically, a number is associated with each system call
  - The **run-time environment** of a given programming language provides a **system-call interface** that maintains a table indexed according to these numbers
  - Table stores addresses of system call implementations
- 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 knows nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result of the call

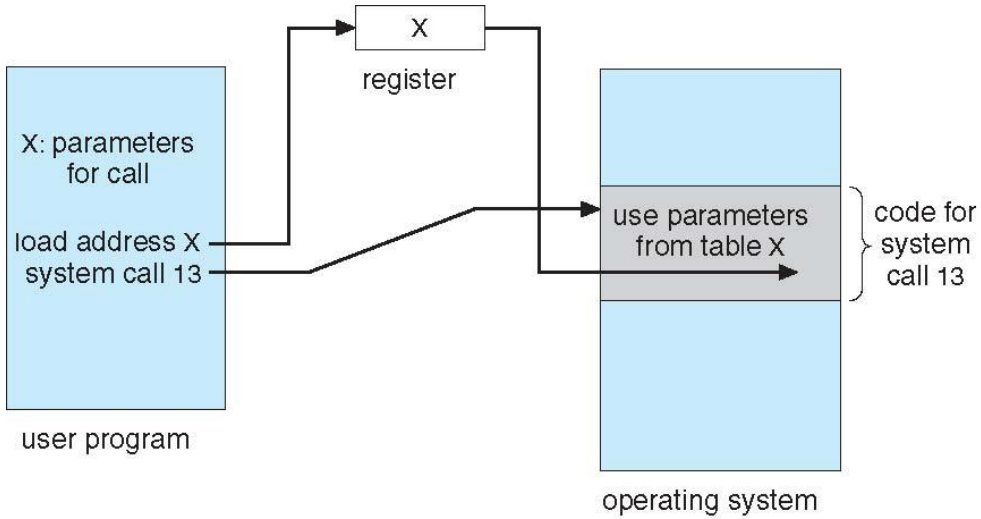


# Example: User Application Invoking open() System Call




# System Call Parameter Passing

- Three general methods are used to pass parameters to the OS
  - Method 1: Pass the parameters in registers
    - There may be more parameters than registers
  - Method 2: Parameters stored in a block in memory, and address of block passed as a parameter in a register




  - Method 3: Parameters **pushed** onto the **stack** by the program and **popped** off the stack by the OS
  - Methods 2 and 3 do not limit the number or length of parameters being passed