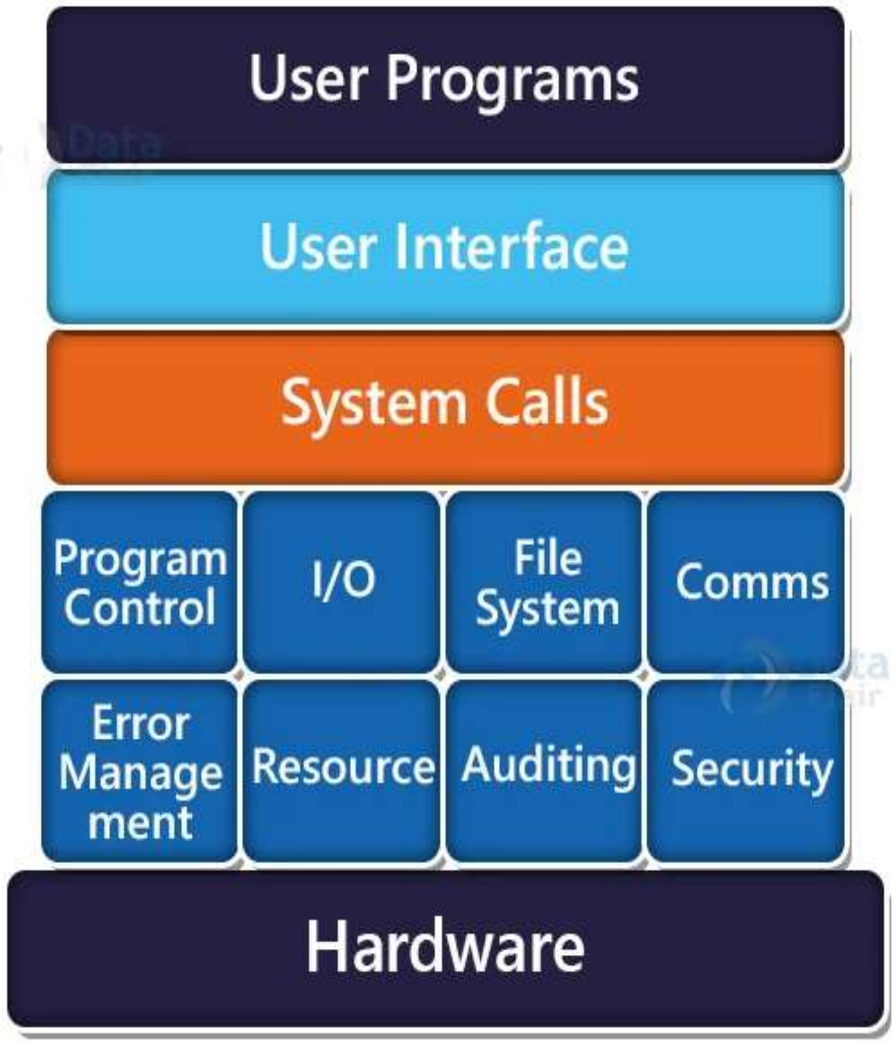
CSE308 Operating Systems

System Calls

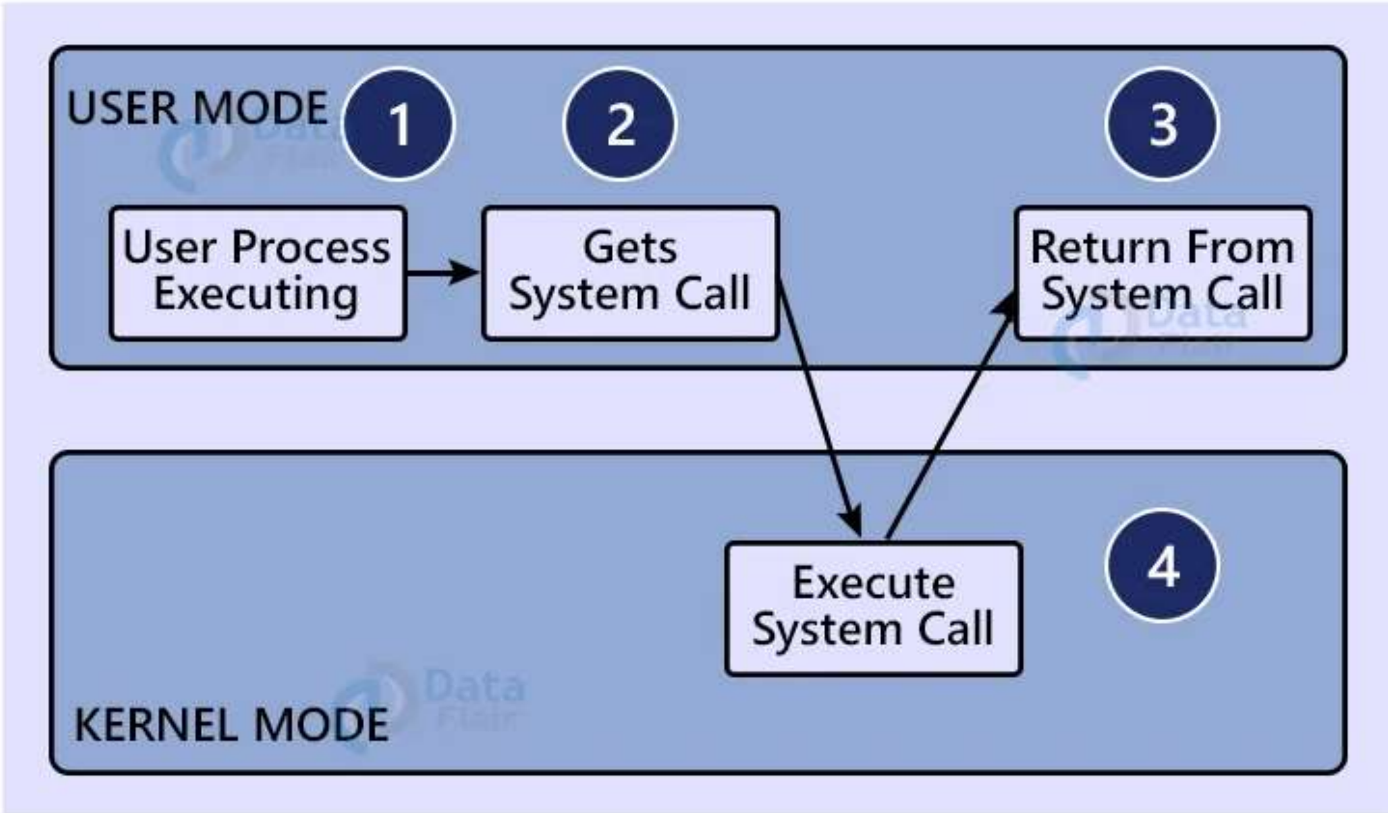
S.Rajarajan
SoC
SASTRA

System Calls

- System calls commonly (abbreviated to **syscall**) provide **an interface to the services** provided by **OS**
- A system call **helps a user program** to **request services** from the **kernel**
- System calls are generally available as routines written in **C and C++,** but certain **low-level tasks** may have to be written using **assembly-language**
- Implementing system calls requires a transfer of control from **user space to kernel space**.
- A typical way to implement this is to use a **software interrupt or trap.**



WORKING OF A SYSTEM CALL



Example for System calls

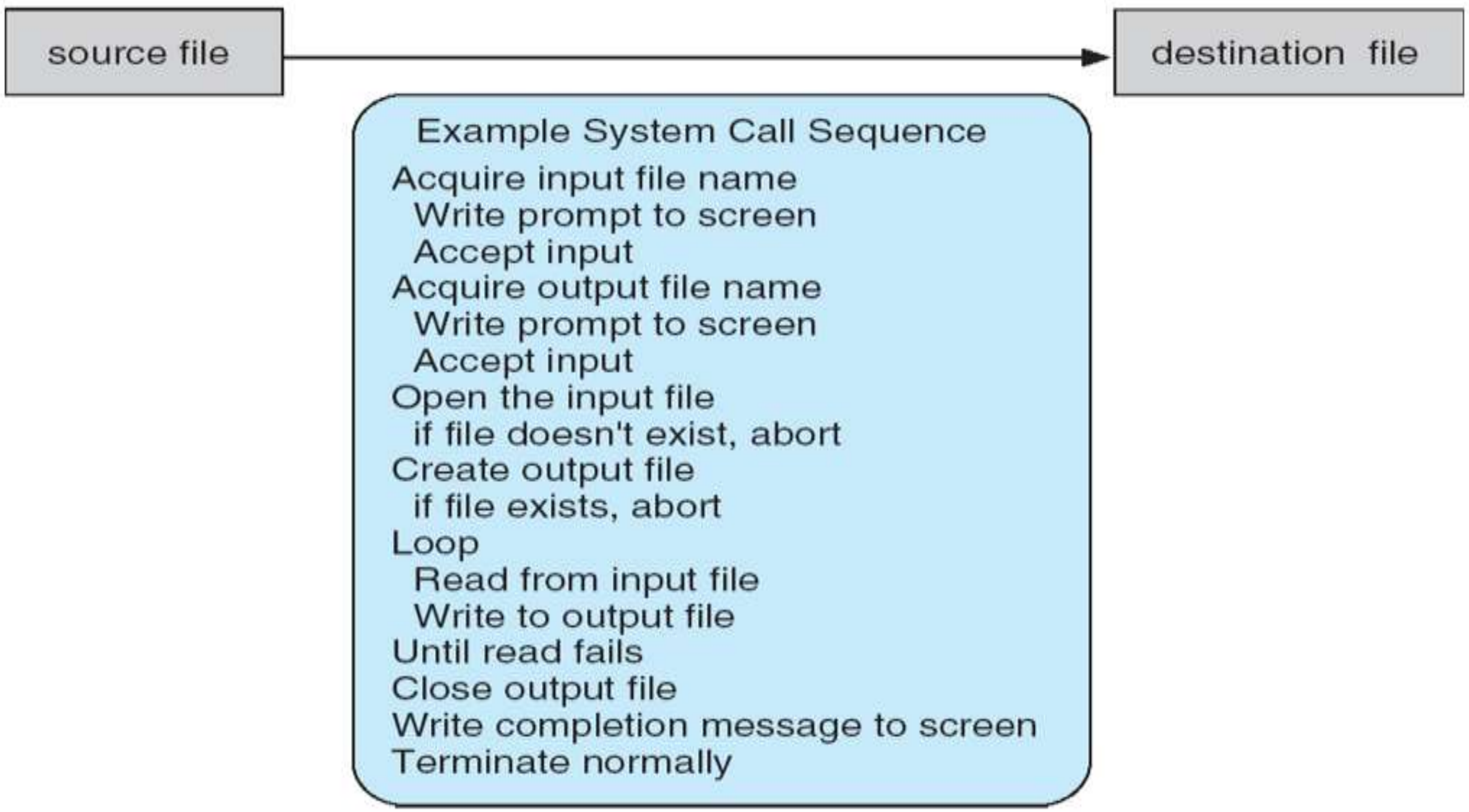
- **Writing a simple program to read data from one file and copy them to another file**
  - First **input** that the program will need is the **names of the two files**: the input file and the output file.
  - This will require a sequence of **I/O system calls**, first to write a **prompting message** on the screen and then to **read from the keyboard** the characters that define the two files.
  - Once the two file names have been obtained, the program must **open the input file** and **create the output file**.
  - Each of these operations requires **another system call.**

- Possible **error conditions** can require additional system calls. When the program tries to open the input file, for example, it may find that there is **no file of that name** or that the **file is protected**
- In these cases, the program should **print a message on the console** (another sequence of system calls) and then **terminate abnormally** (another system call).
- If the input file exists, then we must create a new output file. We may find that there **is already an output file with the same name**.
- This situation may **cause the program to abort** (a system call), or we may **delete the existing file** (another system call) and **create a new one** (yet another system call).
- Another option, **in an interactive system**, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) **whether to replace the existing file** or to **abort the program**

- When both files are set up, we **enter a loop that reads from the input file** (a system call) and **writes to the output file** (another system call).
- Each read and write **must return status information** regarding various possible **error conditions.**
- On input, the program may find that the **end of the file has been reached** or that there was a **hardware failure** in the read (such as a parity error).
- The **write operation may encounter various errors**, depending on the output device (for example, **no more disk space**)
- Finally, after the entire file is copied, the program may **close both files** (another system call), **write a message to the console or window** (more system calls), and finally **terminate normally** (the final system call)

Example of System Calls

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



API

- It is a **software interface** that allows connection between computer programs
- System calls mostly accessed by programs via a **high-level 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)
  - **Java API** for the Java virtual machine (JVM)
- API specifies a **set of functions** that are available to an application programmer, including the **parameters** and the **return values**
- 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()** actually invokes **theNTCreateProcess() system call** in the Windows kernel
- A single API function could make **several system calls**
- **API Vs System calls**
  - The main difference between API and system call is that **API is a set of protocols, routines, functions** that allow **exchanging data among various applications and devices** while a **system call is a method** that allows a **program to request services from the kernel**

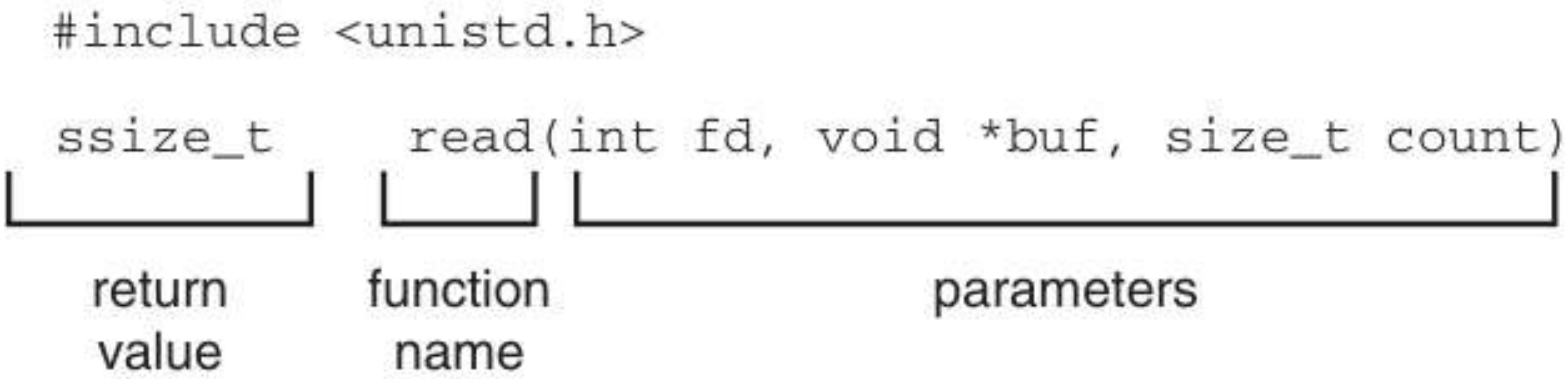
Why use APIs rather than system calls?

- There are several reasons for doing so.
- One benefit is program **portability**.
- Each **operating system** has **its own name** for each system call.
- Application programmer designing a program using an API can expect their programs to **compile and run on any system** that supports the same API
- Furthermore, actual system calls can often be more detailed and **difficult to work with** than the API available to an application programmer – **abstraction**
- Nevertheless, there often exists a strong correlation between a function in the API and its associated system call within the kernel.

- The **caller need know nothing** about how the system call is implemented or what it does during execution.
- Rather, the caller need only **obey the API** and understand what the operating system will do as a result of the execution of that system call.
- Thus, most of the details of the operating-system interface are **hidden from the programmer** by the API and are managed by the **run-time support library**

Example of Standard API

- As an example of a standard API, consider the **read()** function that is available in UNIX and Linux systems.



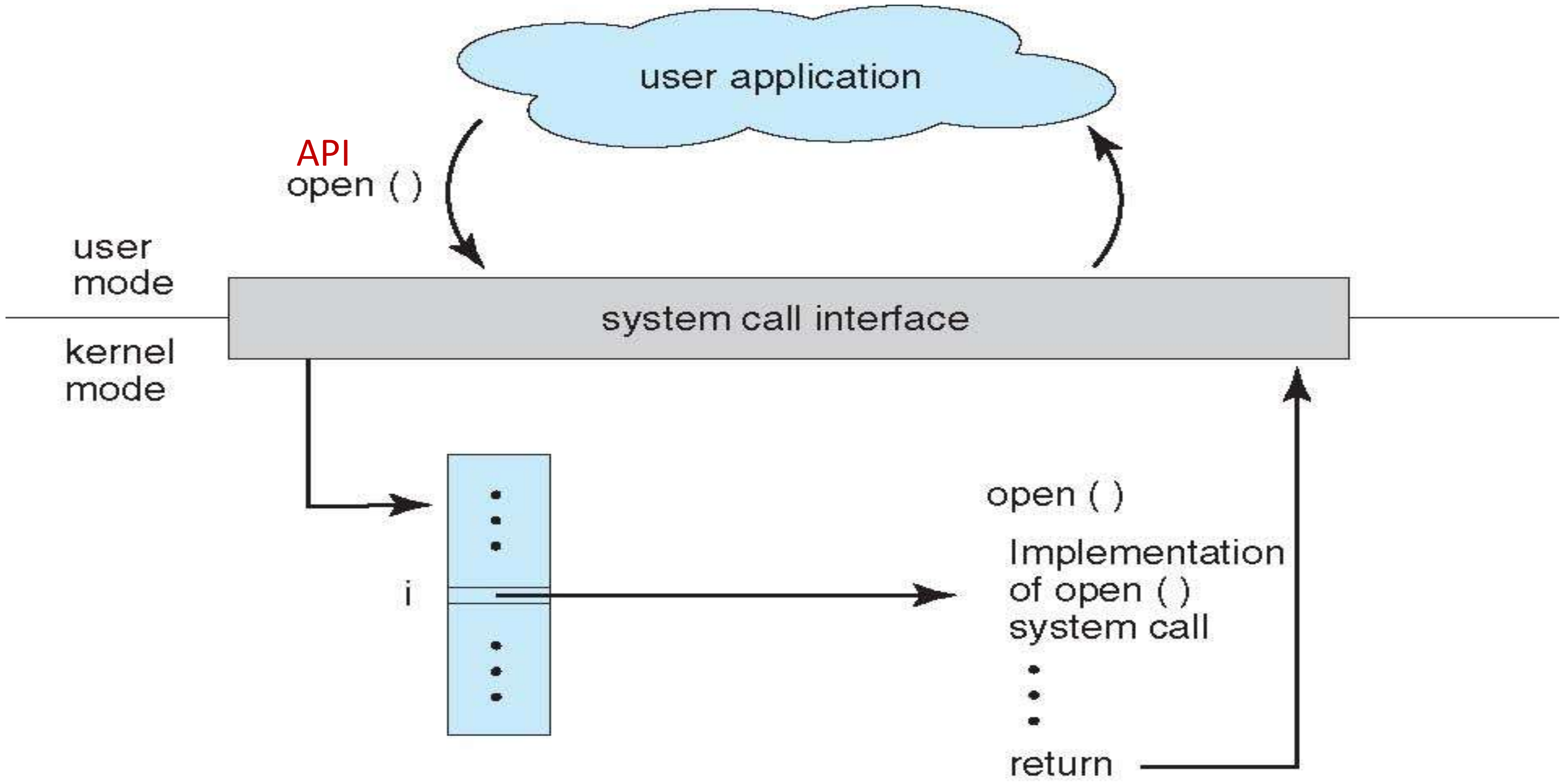
- A program that uses the read() function must include the unistd.h header file, as this file defines the ssize t and size t data types (among other things).

- 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.

API to System call

- For most programming languages, **the run-time support system** provides a **system call interface** that serves as the **link to system calls** made available by the operating system.
- System-call **interface intercepts function calls in the API** and **invokes necessary system call** switching the operating system mode.
- **<u>How the appropriate system call is identified ?</u>**
- Typically, **a number is associated with each system call**, and the system-call interface maintains a **table indexed according to these numbers**
- The system call interface then **invokes the intended system call** in the operating-system kernel and **returns the status** of the system call and **any return values**

API – System Call – OS Relationship

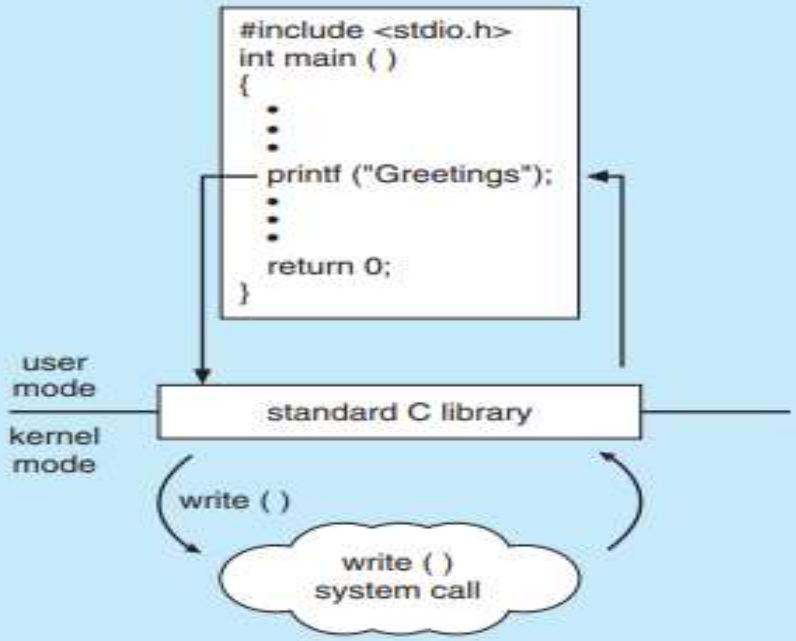


Standard C Library Example

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

EXAMPLE OF STANDARD C LIBRARY

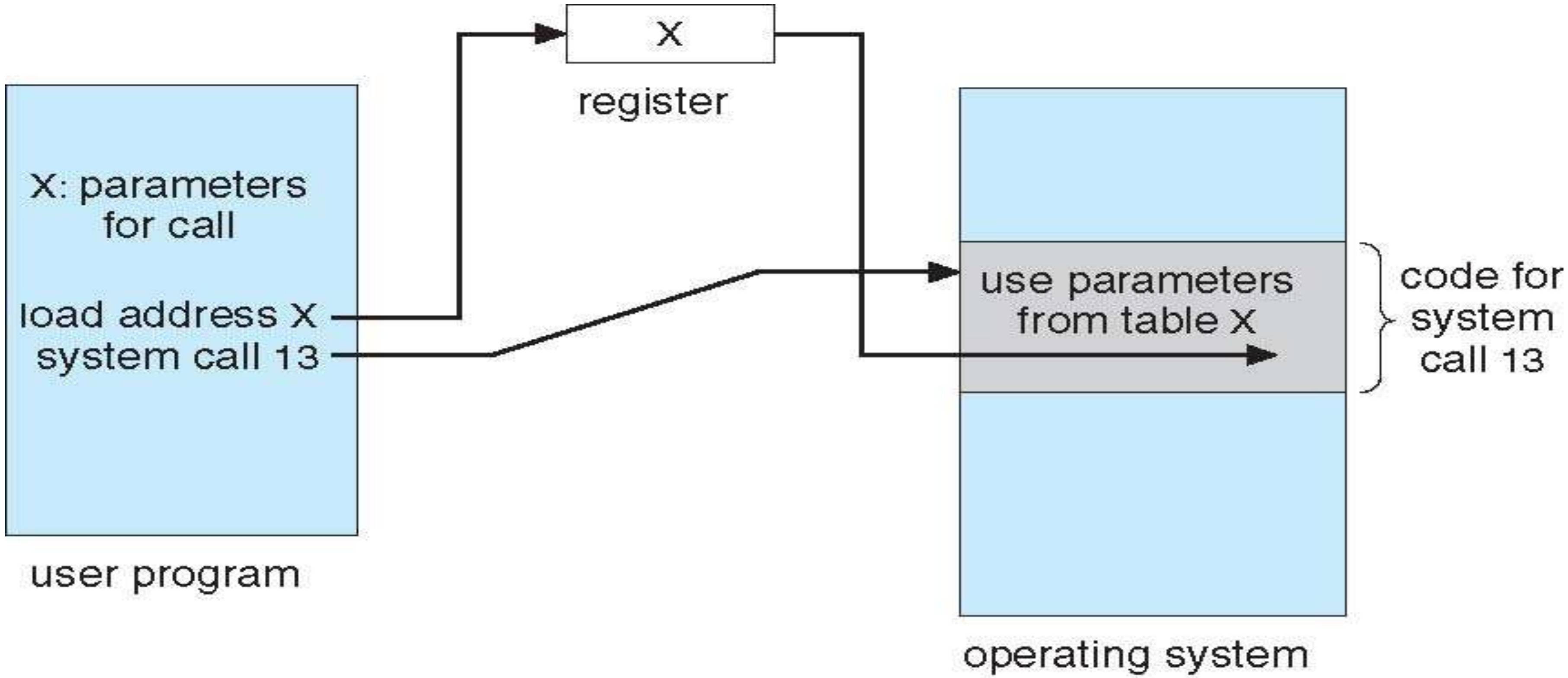
The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance,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



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
  - **Option1:** pass the parameters in ***registers***
    - In some cases, may be more parameters than registers
  - **Option 2:** 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
  - **Option 3:**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 roughly into **six major categories**: process control, file manipulation, device manipulation, information maintenance, communications, and protection
- **Process control**
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- **File management**
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

Types of System Calls (Cont.)

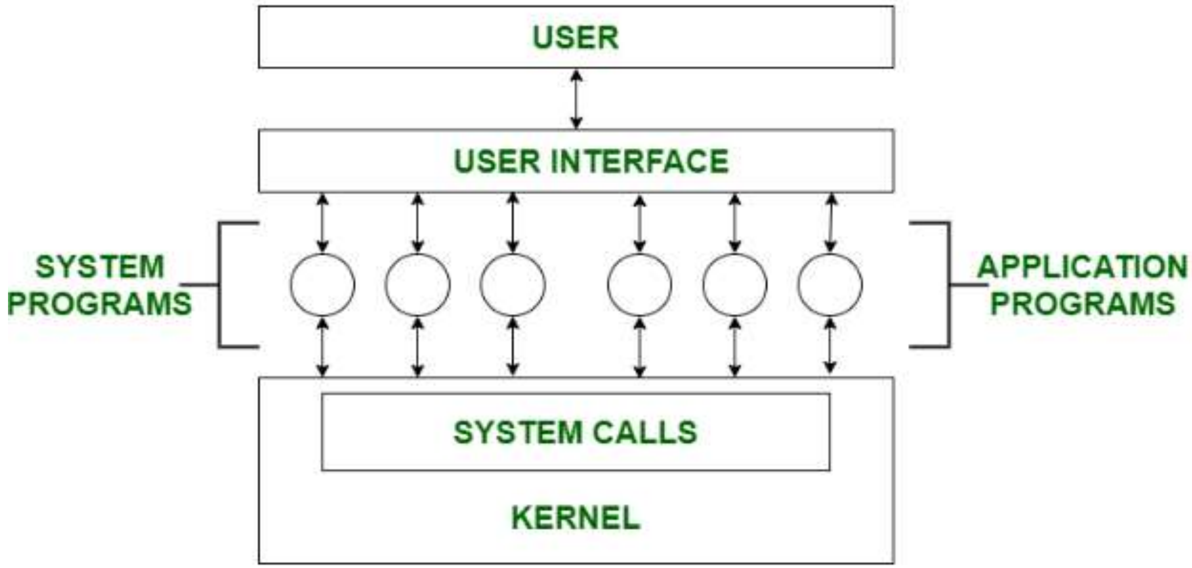
- **Device management**
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- **Information maintenance**
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- **Communications**
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach and detach remote devices

Examples of Windows and Unix System Calls

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

System Programs

- **System Programming** can be defined as the act of building Systems Software using System Programming Languages
- Also known as **system utilities**, provide a **convenient environment** for program **development and execution**
- Some of them are simply user interfaces to system calls.
- Others are considerably more complex.



- System programs can be divided into:
  - **File manipulation-** These programs **create, delete, copy, rename, print, dump, list**, and generally **manipulate files** and directories.
  - **Status information** - Some programs simply ask the system for the **date, time, amount of available memory or disk space, number of users**, or similar status information.
  - Others are more complex, providing **detailed performance, logging**, and **debugging information**

- **File modification** - Several text editors may be available to create and modify the content of files stored on disk or other storage devices
- **Programming language support** - Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and PERL) are often provided with the operating system or
- **Program loading and execution** - lOnce a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders
- **Communications** - These programs provide the mechanism for creating virtual connections among processes, users, and computer systems.
- **Background services-** All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to run until the system is halted – **dameon**s

• Most users' view of the operation system is defined by system programs, not the actual system calls