Amirkabir University of Technology
(Tehran Polytechnic)

Operating Systems

Multiprogramming, Dual-mode and System Calls

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

Part1

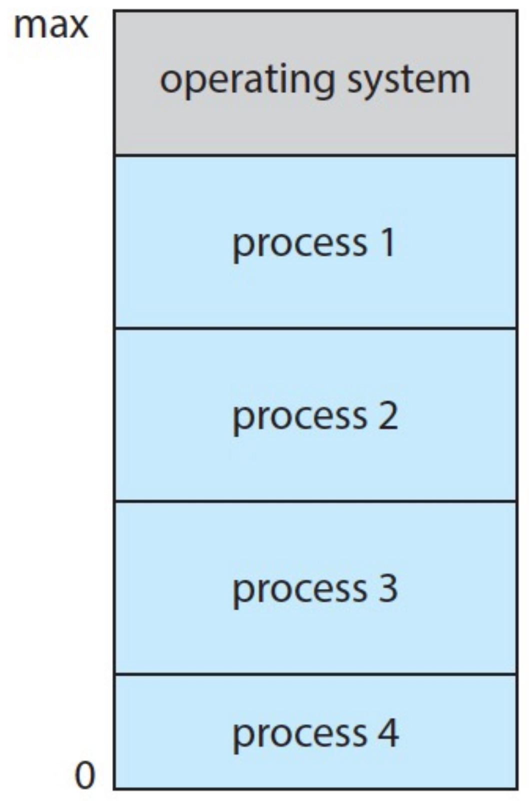
MULTIPROGRAMMING AND DUAL-MODE

Multiprogramming (Batch System) (cont.)

- Multiprogramming organizes multiple jobs (code and data) -->
  - CPU always has one to execute.
- A subset of total jobs

  in system is kept in memory.
- One job selected and run via job scheduling.
- When job has to wait (I/O for example),

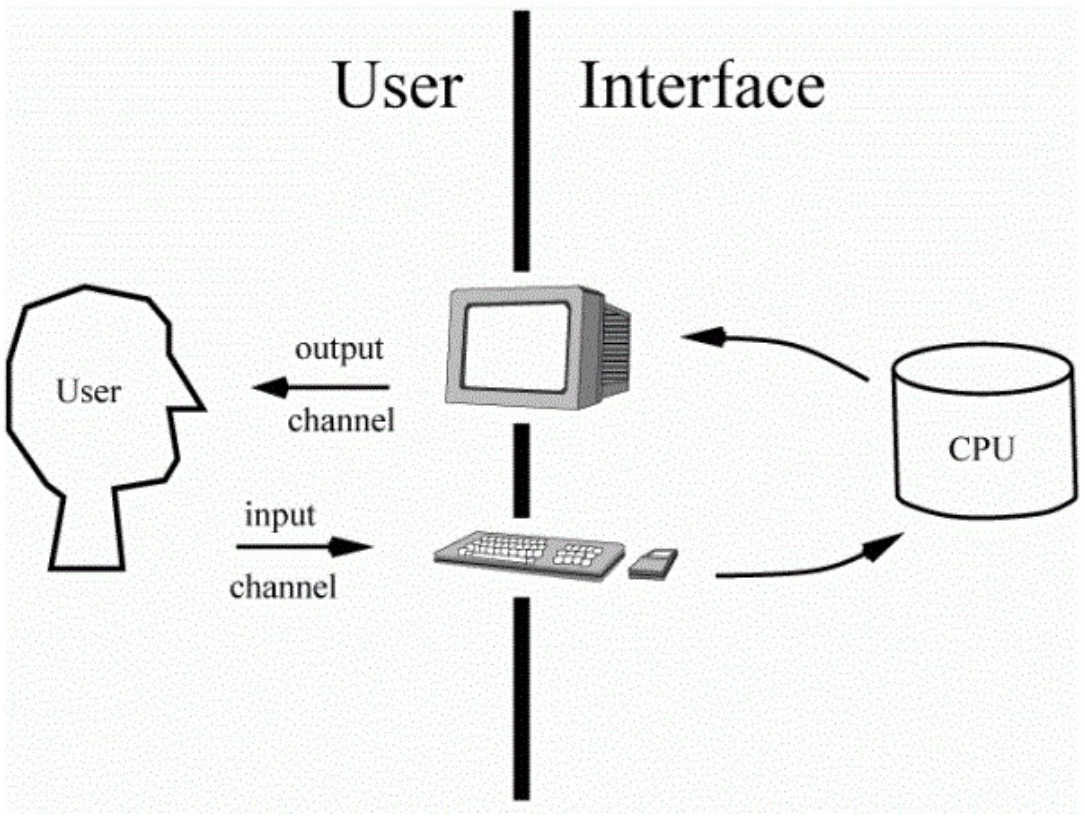
  OS switches to another job.



Memory layout for a multiprogramming system

Multiprogramming (Batch System)

- Single user/program cannot always keep CPU and I/O devices busy.



Multiprogramming (Batch System) (cont.)

- Single user/program cannot always keep CPU and I/O devices busy.
- Examples

Program	CPU-intensive	Memory-intensive	I/O-intensive
Random Number Generator	?	?	?
Microsoft word	?	?	?
QuickTime Player (a long 4K video)	?	?	?

Multitasking (Timesharing)

- A logical extension of Batch systems
- The CPU ***switches jobs so frequently*** that users can interact with each job while it is running, creating **interactive** computing.
  - Response time should be < 1 second.
  - Each user has at least one program executing in memory ⇨ process.
  - If several jobs ready to run at the same time ⇨ CPU scheduling.
  - If processes don't fit in memory, swapping moves them in&out to run.
  - Virtual memory allows execution of processes not completely in memory.

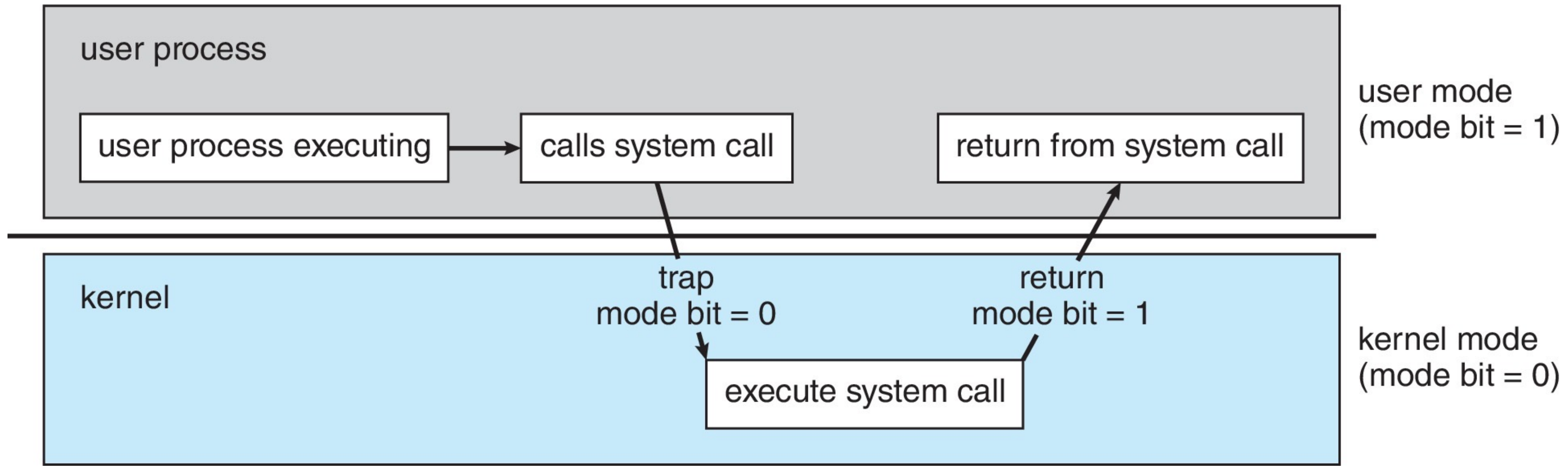
https://www.geeksforgeeks.org/difference-between-job-task-and-process/

Dual-mode Operation

- **Dual-mode** operation allows OS to protect itself and other system components.
  - **User mode** and **kernel mode**
- **Mode bit** provided by hardware
  - Provides ability to distinguish when system is running user code or kernel code.
  - When a user is running ⇨ mode bit is “user”.
  - When kernel code is executing ⇨ mode bit is “kernel”.

Dual-mode Operation (Cont.)

- How do we guarantee that user does not explicitly set the mode bit to “kernel”?
  - System call changes mode to kernel, return from call resets it to user.



Types of Instructions

- Instructions are divided into two categories:
  - The ***non-privileged instruction*** instruction is an instruction that ***any application or user can execute***.
  - The ***privileged instruction*** is an instruction that ***can only be executed in kernel mode***.
- Instructions are divided in this manner because privileged instructions ***could harm the kernel***.

http://web.cs.ucla.edu/classes/winter13/cs111/scribe/4a/

Examples of instructions

Instruction	Type
Reading the status of Processor	?
Set the Timer	?
Sending the final printout of Printer	?
Remove a process from the memory	?

Examples of non-privileged instructions

- Reading the status of Processor
- Reading the System Time
- Sending the final printout of Printer

https://www.geeksforgeeks.org/privileged-and-non-privileged-instructions-in-operating-system/

Examples of privileged instructions

- I/O instructions and halt instructions
- Turn off all Interrupts
- Set the timer
- Context switching
- Clear the memory or remove a process from the memory
- Modify entries in the device-status table

https://www.geeksforgeeks.org/privileged-and-non-privileged-instructions-in-operating-system/

Privileged instructions

If an attempt is made to execute a privileged instruction in user mode

↓

The hardware ***does not execute the instruction*** but rather treats it as ***illegal*** and ***traps*** it to the ***operating system.***

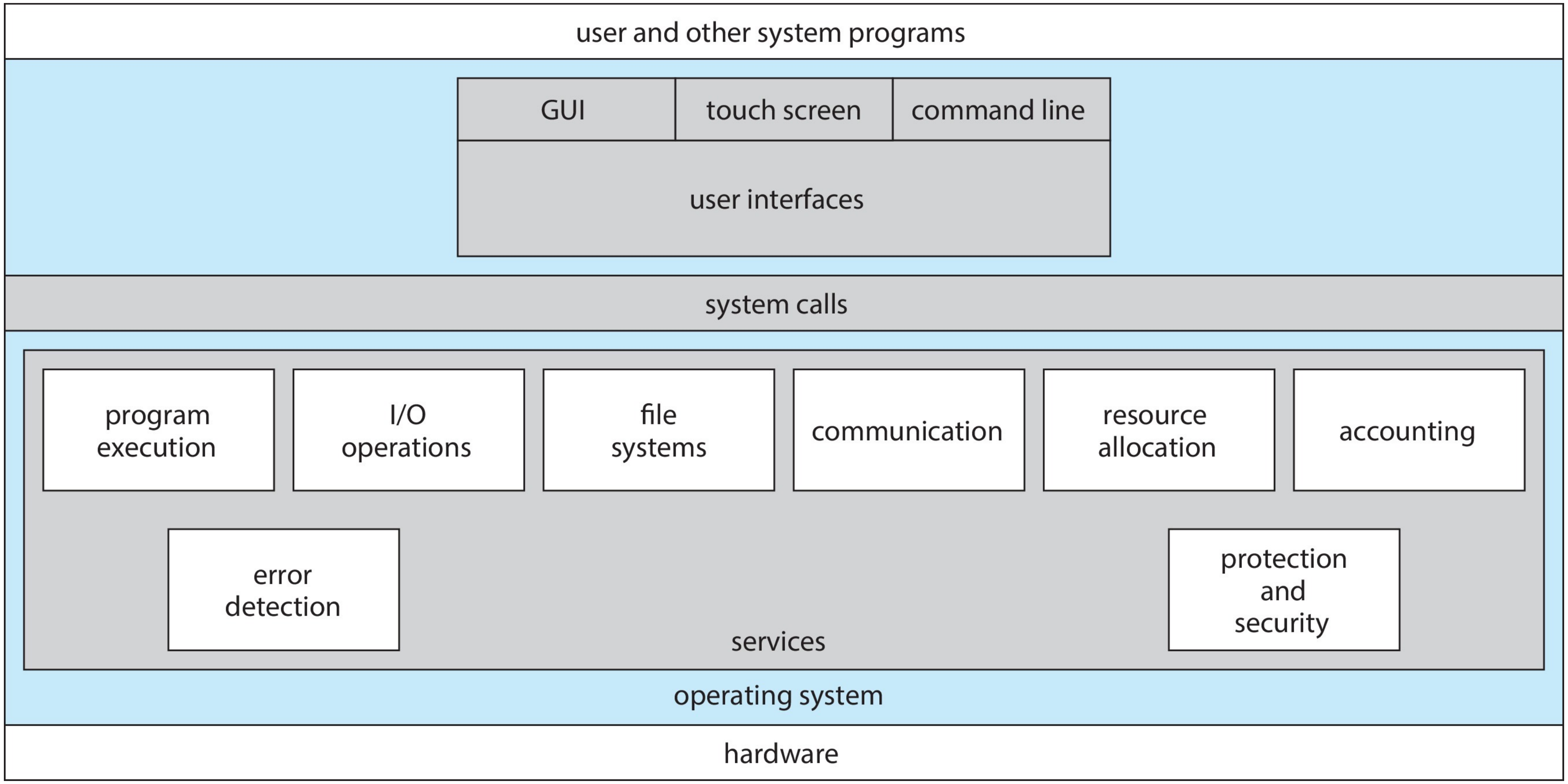
Questions?

Part2

SYSTEM CALLS

System Calls

- Programming interface to the services provided by the OS.



System Calls (cont.)

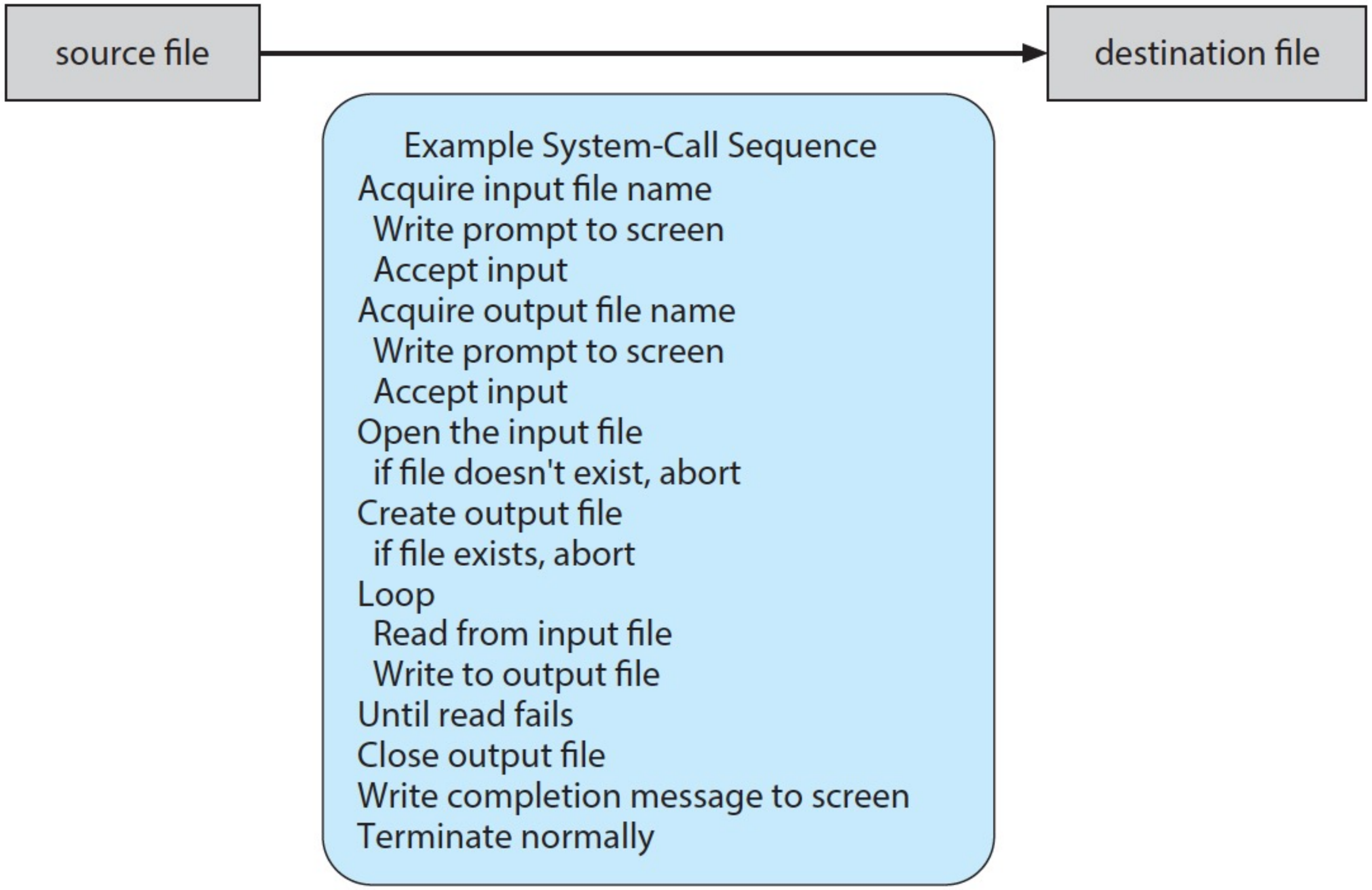
- Typically written in a high-level language (C or C++ or Assembly).
- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use.
- Three most common APIs are:
  - Win32 API for Windows (Win API)
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM).

Note that the system-call names used

throughout this text are generic.

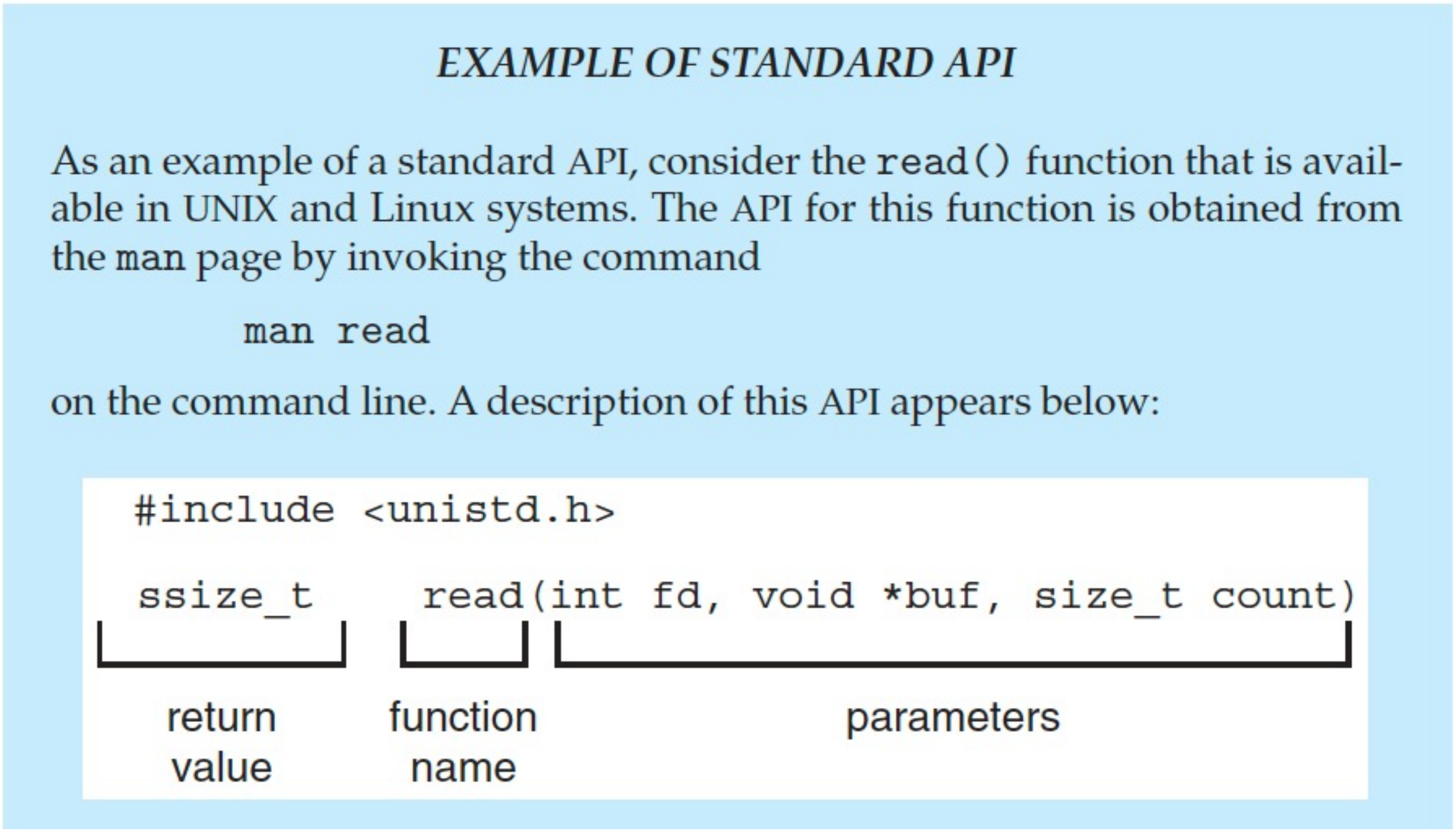
Example of System Calls Seq.

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



```
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



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

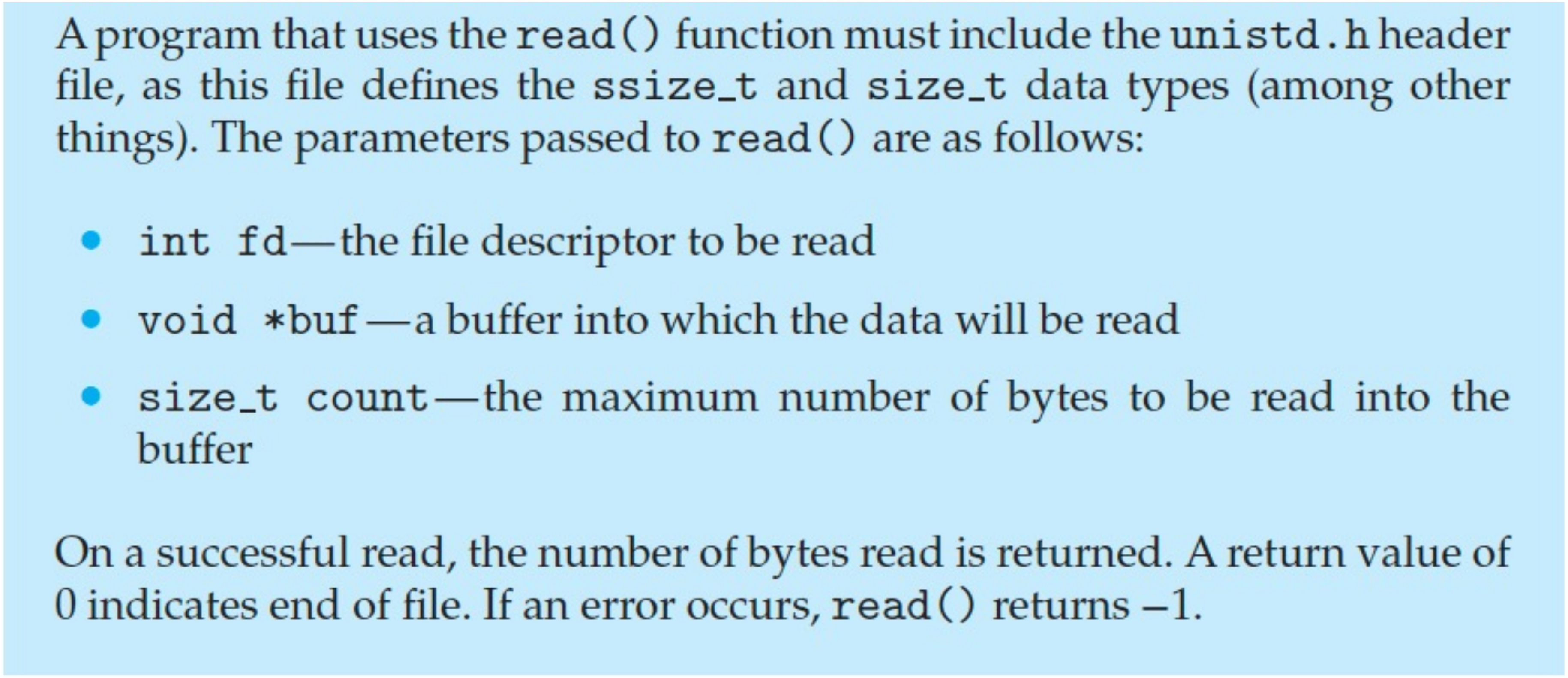
```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>
```

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

Example of Standard API (Cont.)



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 into which the data will be read
- `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.

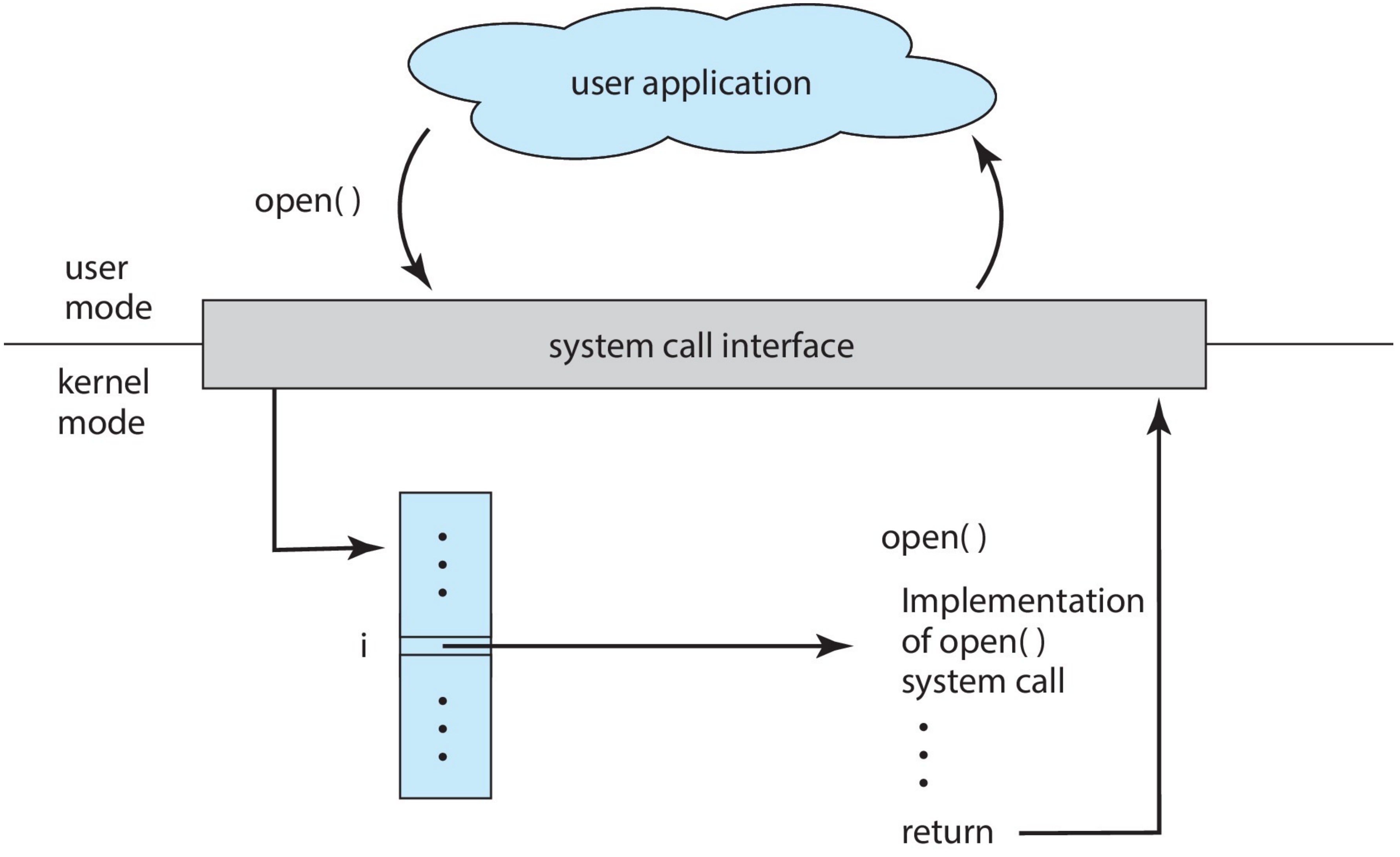
System Call Implementation

- Typically, a number is 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



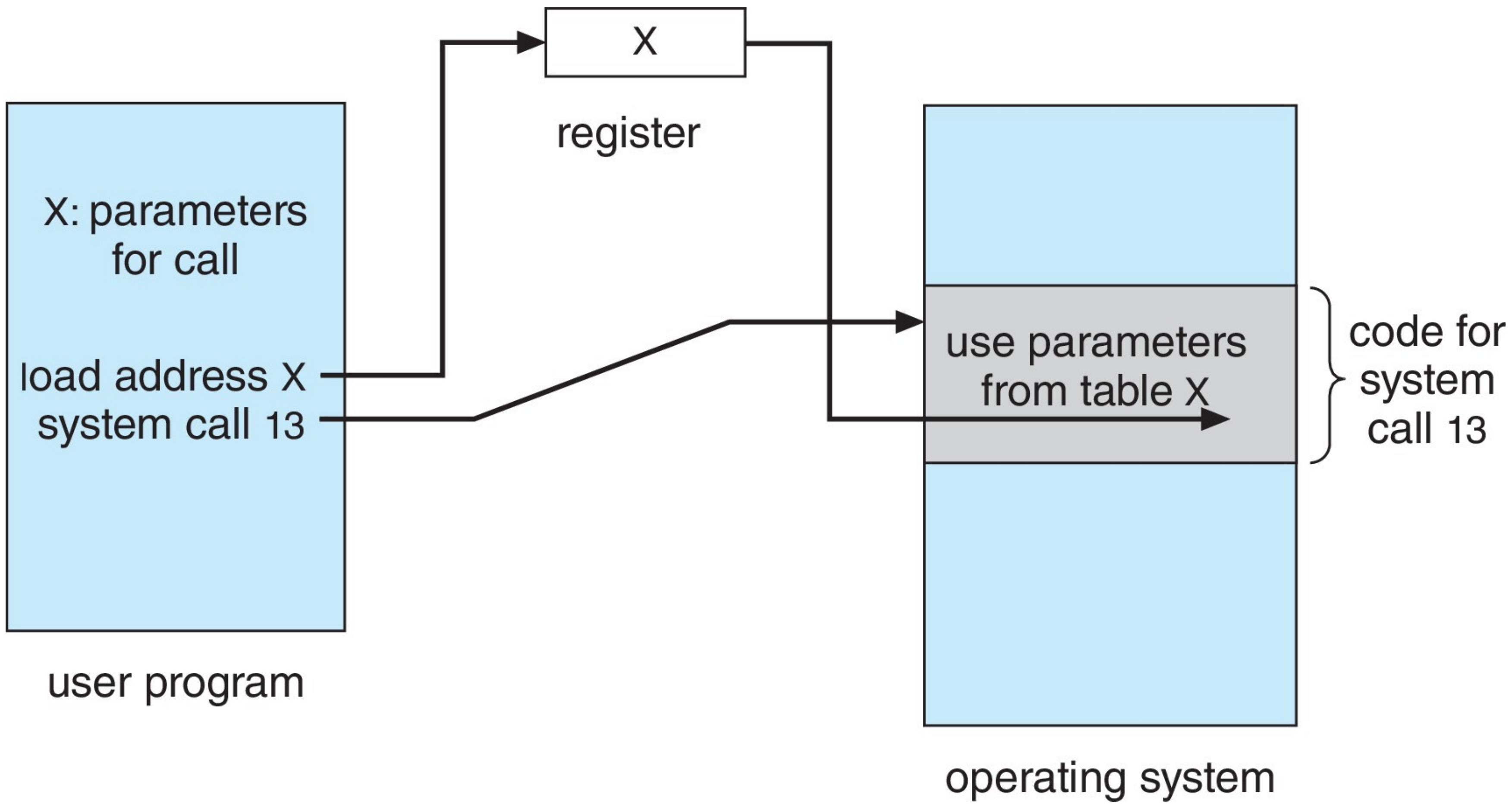
System Call Parameter Passing

- **Parameter Passing**
  - Register
  - Register pointer to mem. Block
  - Stack (Push, Pop)
- Often, more information is required than simply identity of desired system call.
- Exact type and amount of information vary according to OS and call.

System Call Parameter Passing--Methods

- **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.

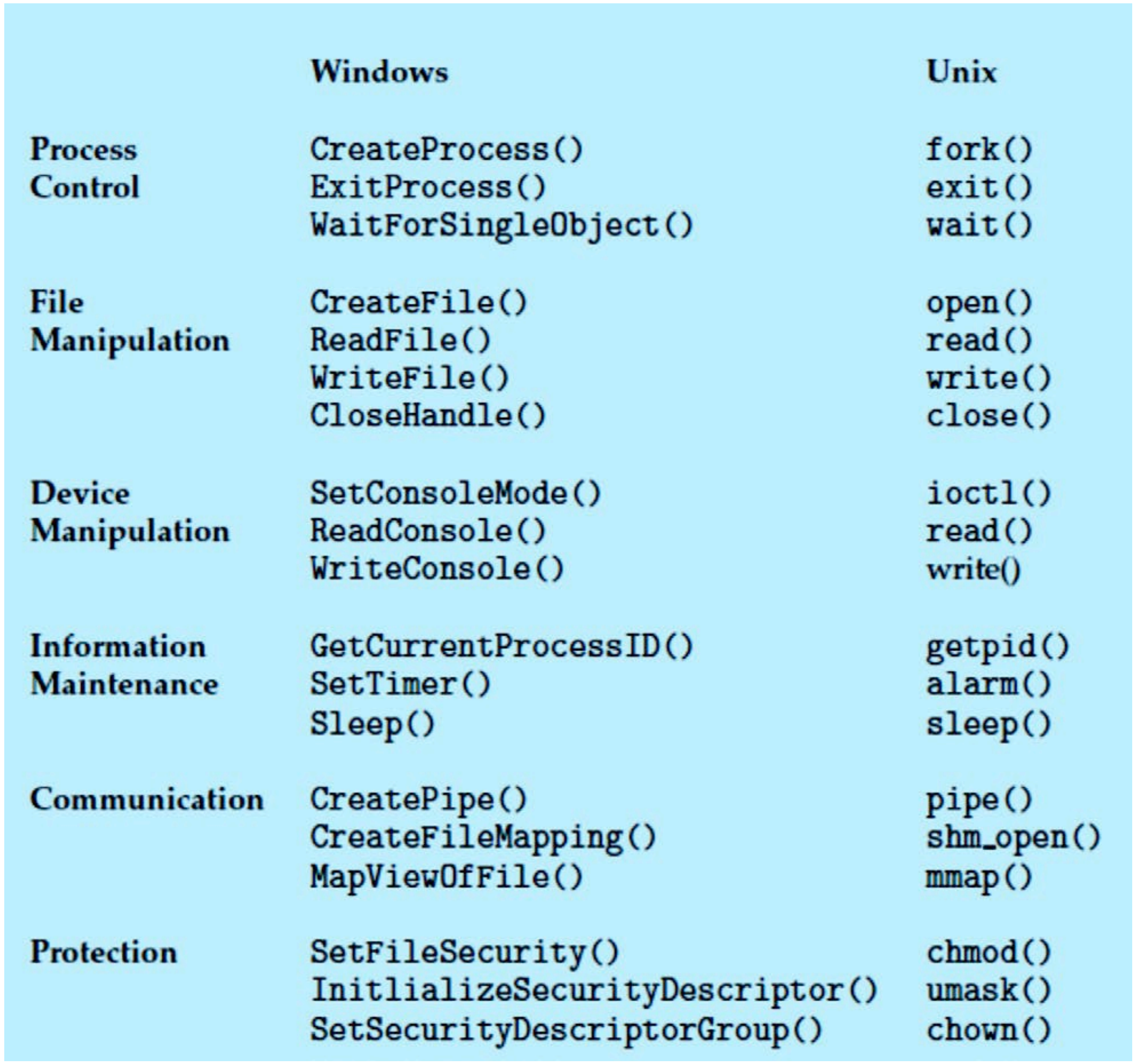
Parameter Passing via Table



Types of System Calls

- **Process control**
  - Create process, terminate process
  - …
- **File management**
  - create file, delete file
  - …
- **Device management**
  - request device, release device
  - …
- **Please study the reference book for more details**

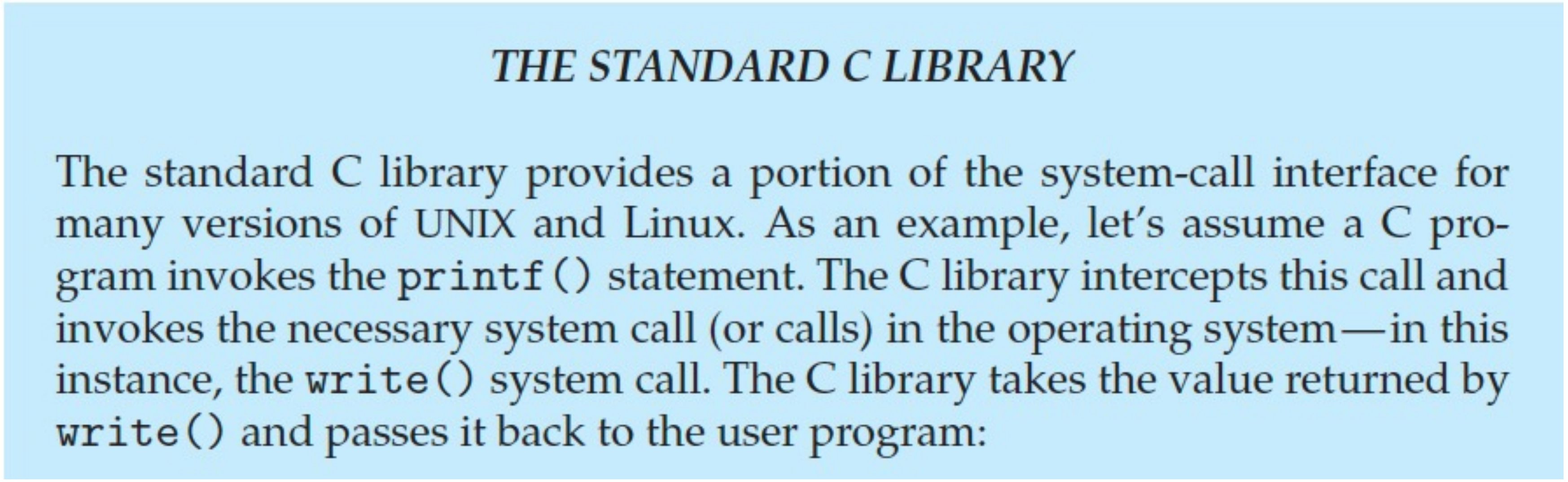
Types of System Calls (Cont.)



	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()	shm_open()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

Standard C Library Example

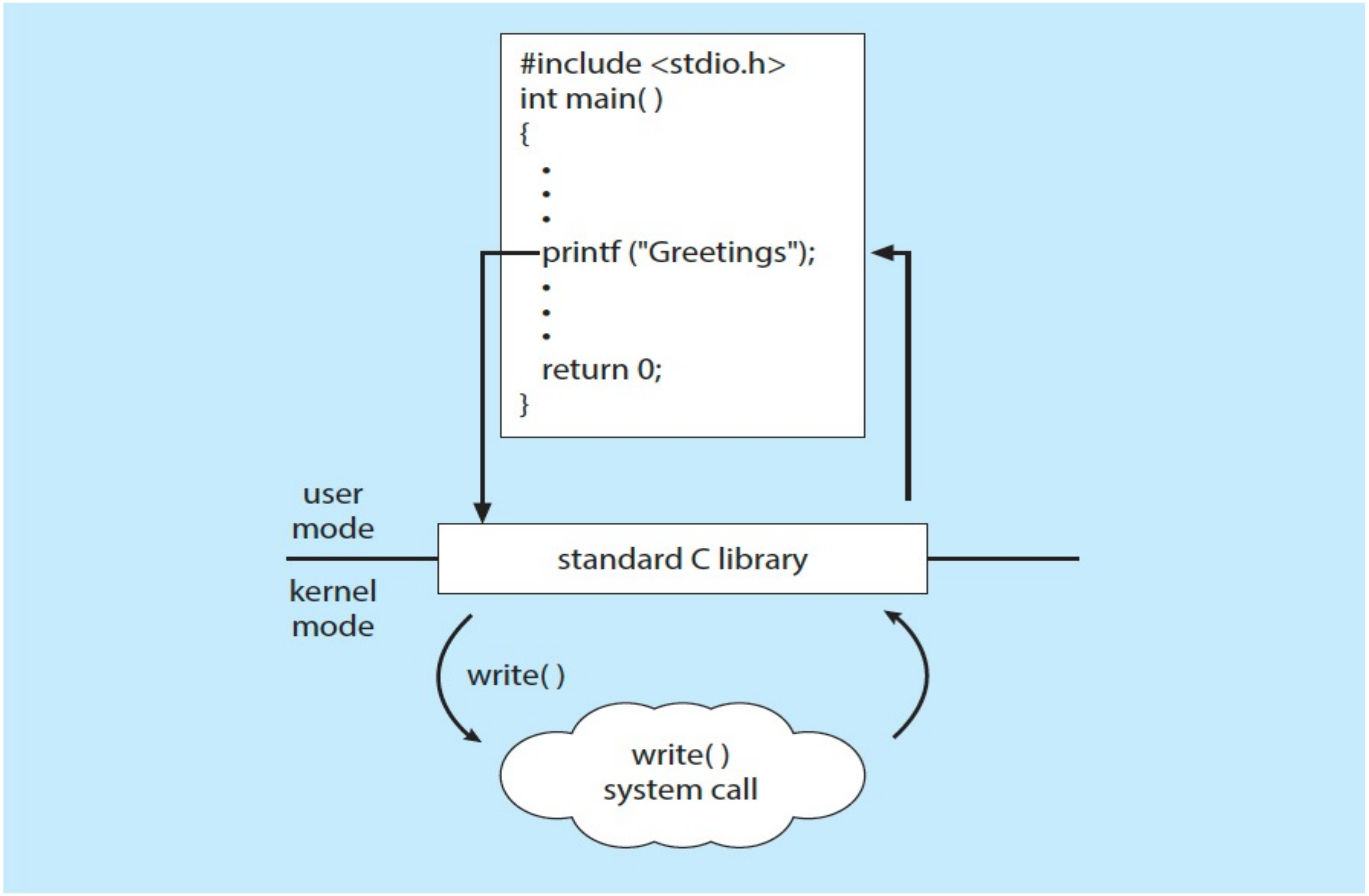
- C program invoking printf() library call, which calls write() system call



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:

Standard C Library Example (Cont.)



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

Why Applications are Operating System Specific

- Apps compiled on one system usually not executable on other OSs.
- Each OS provides its own unique system calls
  - Own file formats, etc.
- Apps can be multi-operating system
  - Written in interpreted language like Python, Ruby, and interpreter available on multiple OSs.
  - App written in language that includes a VM containing the running app (like Java).
  - Use standard language (like C), compile separately on each operating system to run on each.

Questions?