Chapter 2: Operating-System Structures

Neerja Mhaskar

Department of Computing and Software, McMaster University, Canada

Acknowledgements: Material based on the textbook Operating Systems Concepts (Chapter 1)

Operating System Structure

- General-purpose OS is a very large program
- Various ways to structure an OS
  - Simple structure (Monolithic) – e.g., MS-DOS
  - Non-simple structure (Monolithic) – e.g., UNIX
  - Layered – an abstraction
  - Microkernel –Mach
  - Modules

Simple Monolithic Structure -- MS-DOS

- Simplest monolithic structure has little to no structure at all.
- All of the functionality of the kernel (process, memory, file, I/O) is placed into a single, static binary file that runs in a single address space.
- MS-DOS – written to provide the most functionality in the least space. But has some structure, however not divided into modules.
  - Its interfaces and levels of functionality are not well separated.
    - Application programs can access I/O devices!

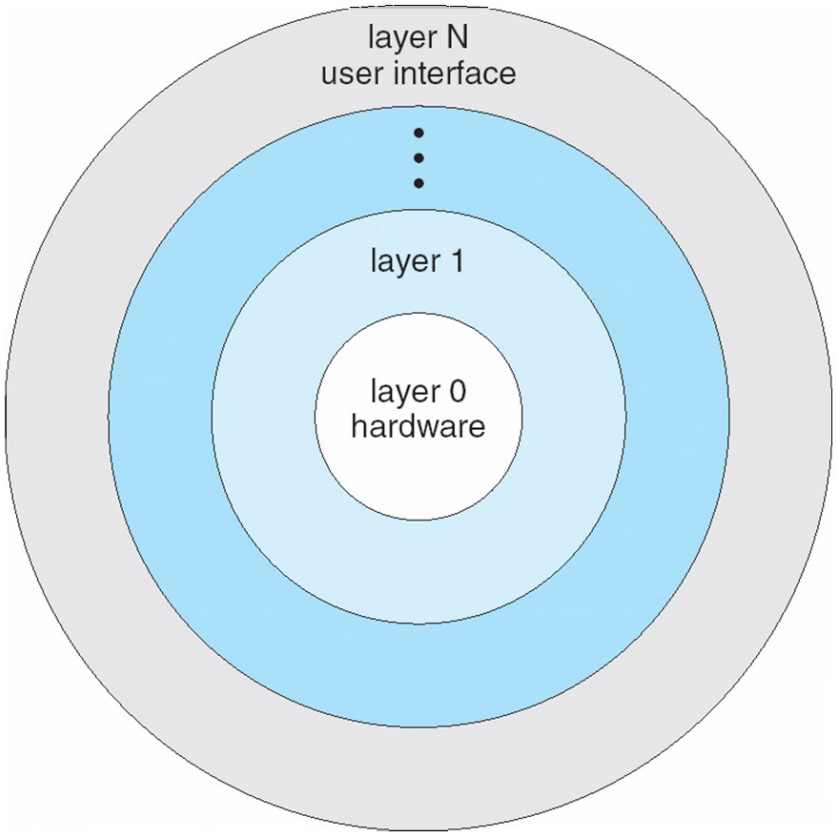
Non-simple Monolithic Structure -- Original UNIX OS

- The original UNIX operating system had limited structuring.
- Like MS-DOS limited by hardware functionality.
- The UNIX OS consists of two separable parts:
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions;
    - A large number of functions for one level

Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers
- Disadvantages:
  - Delineating the layers is tricky.
  - Slow as any user request needs to go through all the layers with correct function calls and parameters.

Layered Approach Contd…



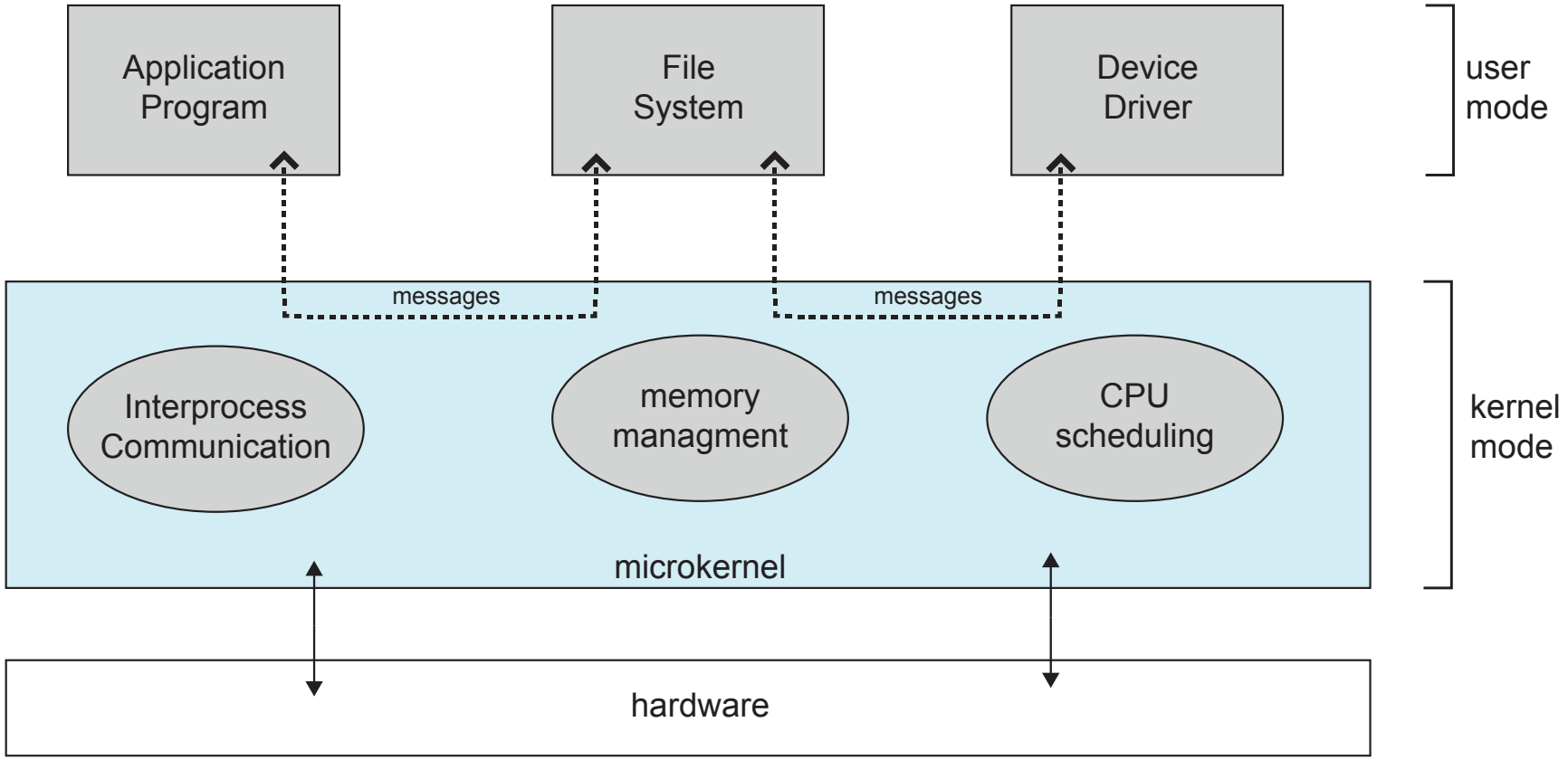
Microkernel System Structure

- As Unix expanded the kernel became large and difficult to manage.
- Microkernel – This approach structures the OS by
  - Removing all the nonessential components from the kernel and
  - Implementing them as user or system-level programs.
- Microkernels provide minimal process and memory management.
- **Mach** is an example of **microkernel**
  - Mac OS X (open source) kernel Darwin partly based on Mach
- Communication between processes takes place between user modules using **message passing** (sharing data by passing messages).

Microkernel – contd…

- Advantages:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable and secure
- Disadvantages:
  - Performance overhead of user space to kernel space communication (Windows NT, first release had a layered microkernel approach.)

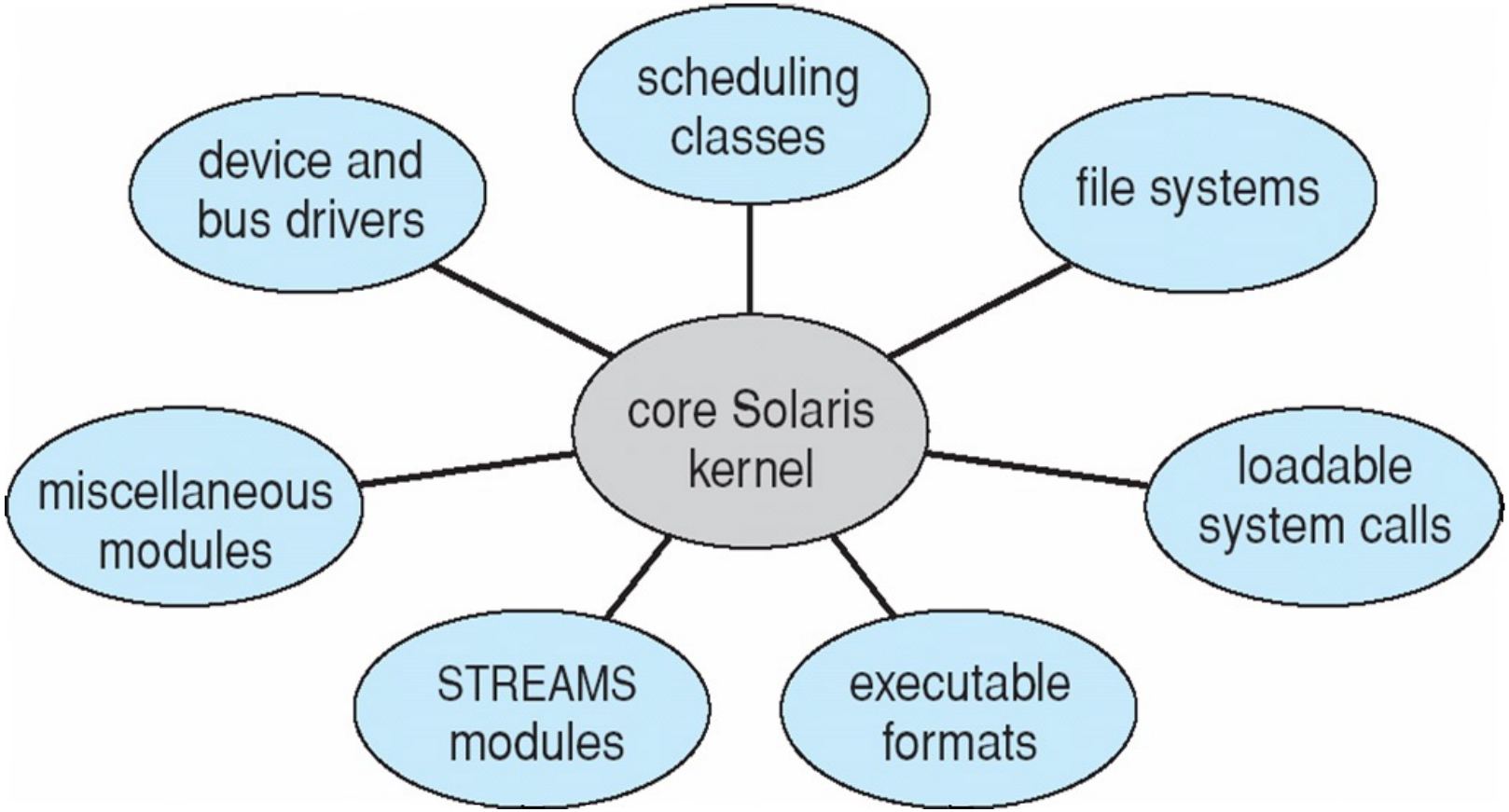
Microkernel System Structure



Modules

- Many modern operating systems (UNIX, Linux, solaris, etc., and Windows) implement **loadable kernel modules** (so far, the best methodology for OS design)
  - Kernel has a set of separate core components (with clearly defined interfaces). Similar to a combination of layered and micro kernel approach.
  - Additional services are linked in via modules (either at boot time or run time.)
  - Each module is loadable as needed within the kernel

Solaris Modular Approach



Hybrid Systems

- Most modern operating systems are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels are:
    - Monolithic for efficient performance.
    - Modular - dynamic loading of functionality

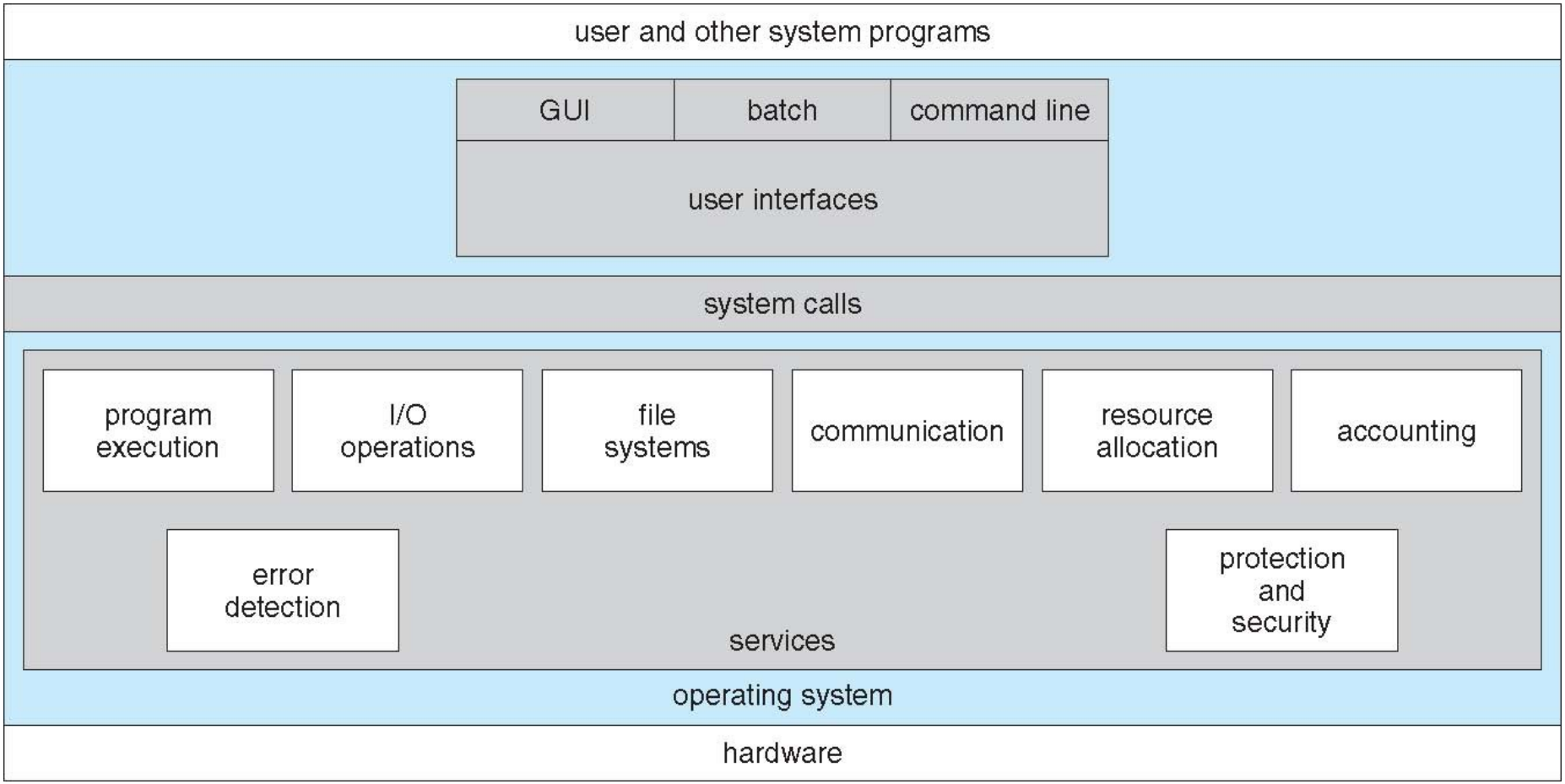
Operating System Services – for user

- Operating systems provide an environment for execution of programs and services to programs and users.
- **Operating-system services for the user**:
  - User interface to interact with OS - e.g., CLI, GUI, touch screen interface.
  - Program execution
  - I/O operations
  - File-system manipulation
  - Communications
  - Error detection

Operating System Services – for system

- **Operating-system services for the system:**
  - Resource allocation
  - Accounting
  - Protection and Security

A View of Operating System Services



User and Operating-System Interface

Command Line Interpreter (CLI) allows direct command entry to OS.

- Windows and Unix treat CLI as a separate program; that is, CLI is not part of the kernel.
- In Unix/Linux they are called **shells** (e.g., bash shell in Linux).
- Primary job is to fetch a command from user and execute it.
- Sometimes the CLI itself contains the code to execute the command.
- Sometimes just names of programs (used by Unix)
  - Adding new features doesn't require shell modification!

Shells in Unix and Linux

- Multiple shells are available (you can write your own shell program!)
- $ echo $0 OR $ echo “$SHELL” – will give you the current shell
- Shells do not have the code to execute the request.
- Eg: rm file.text in terminal → Invokes Shell → Shell searches for file rm → load rm in memory → executes it with file.txt as parameter
- Shell has no idea how ‘rm’ command is implemented and the system call used to process the request.

System Calls

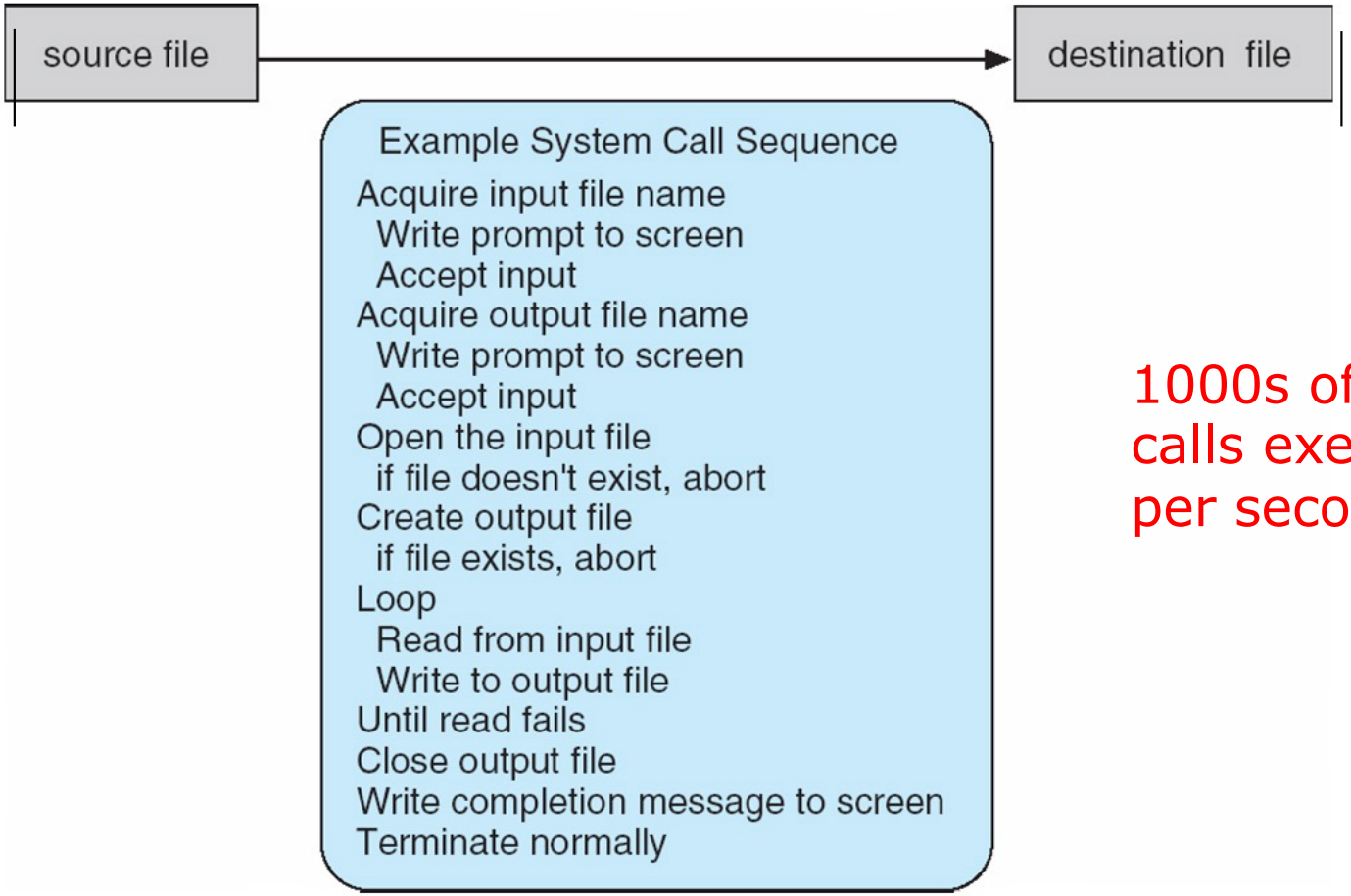
- Provide an interface to OS services
- Are routines mostly written in a high-level language (C or C++). However, lower level task written in assembly.
- Accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use.

API

- An API specifies a set of interfaces (functions) available to the programmer.
- These interfaces can be implemented as single or multiple system calls.
- APIs used as system calls are detailed and difficult to work with.
- Three most common APIs are:
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)

How are System Calls used? - Example

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



1000s of system calls executed per second!

Note: Programmers don't see this level of detail.

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:

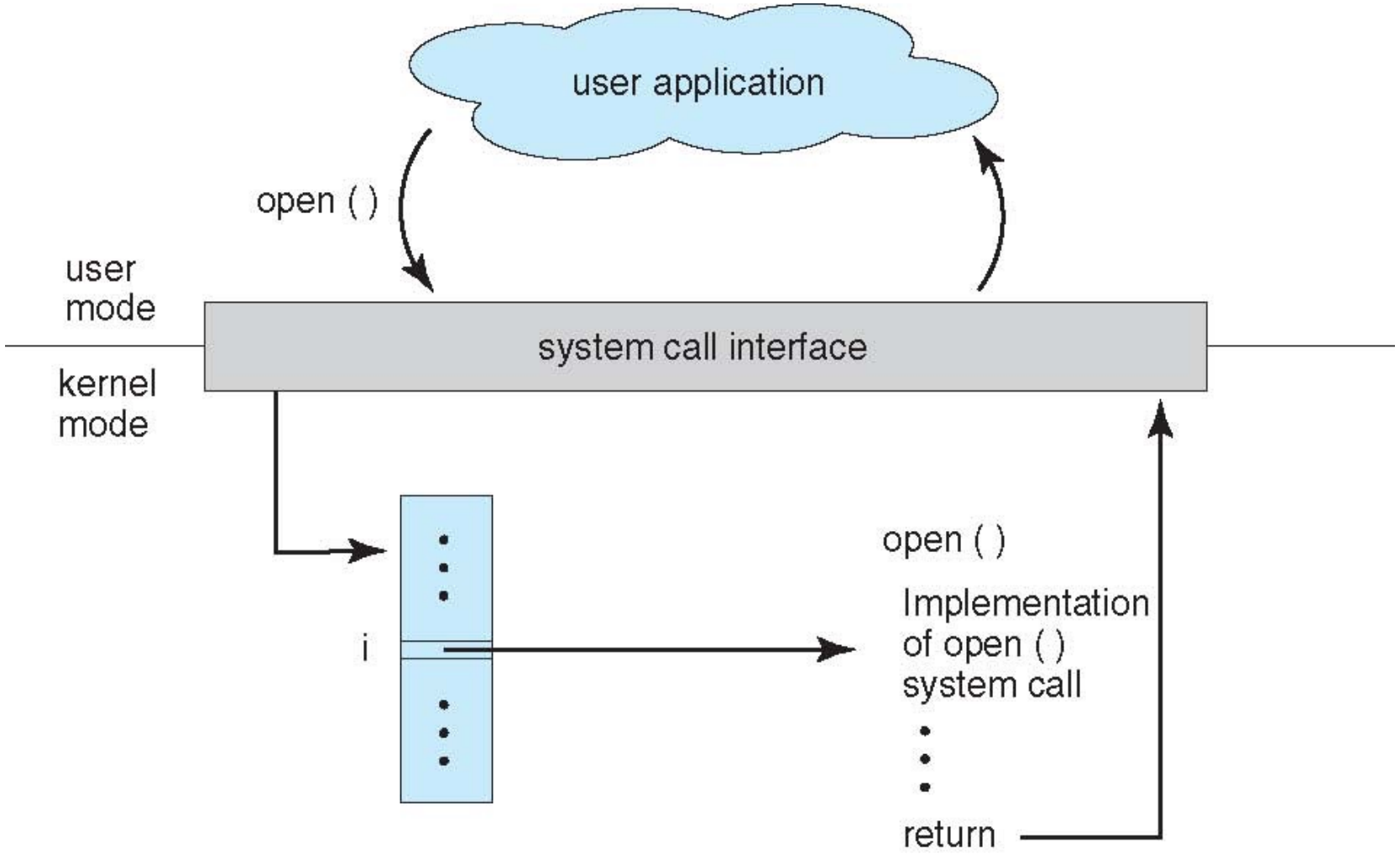
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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.

System Call Implementation

- The API makes the appropriate system calls through the **system call interface**.
- **System call interface** serves as a link to system calls made available by the Operating System.
  - Each system call has a number that uniquely identifies it
  - Maintains a table indexed according to these numbers
- The user program need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result after the system call is invoked.
  - Most details of OS interface hidden from programmer by API

API – System Call – OS Relationship



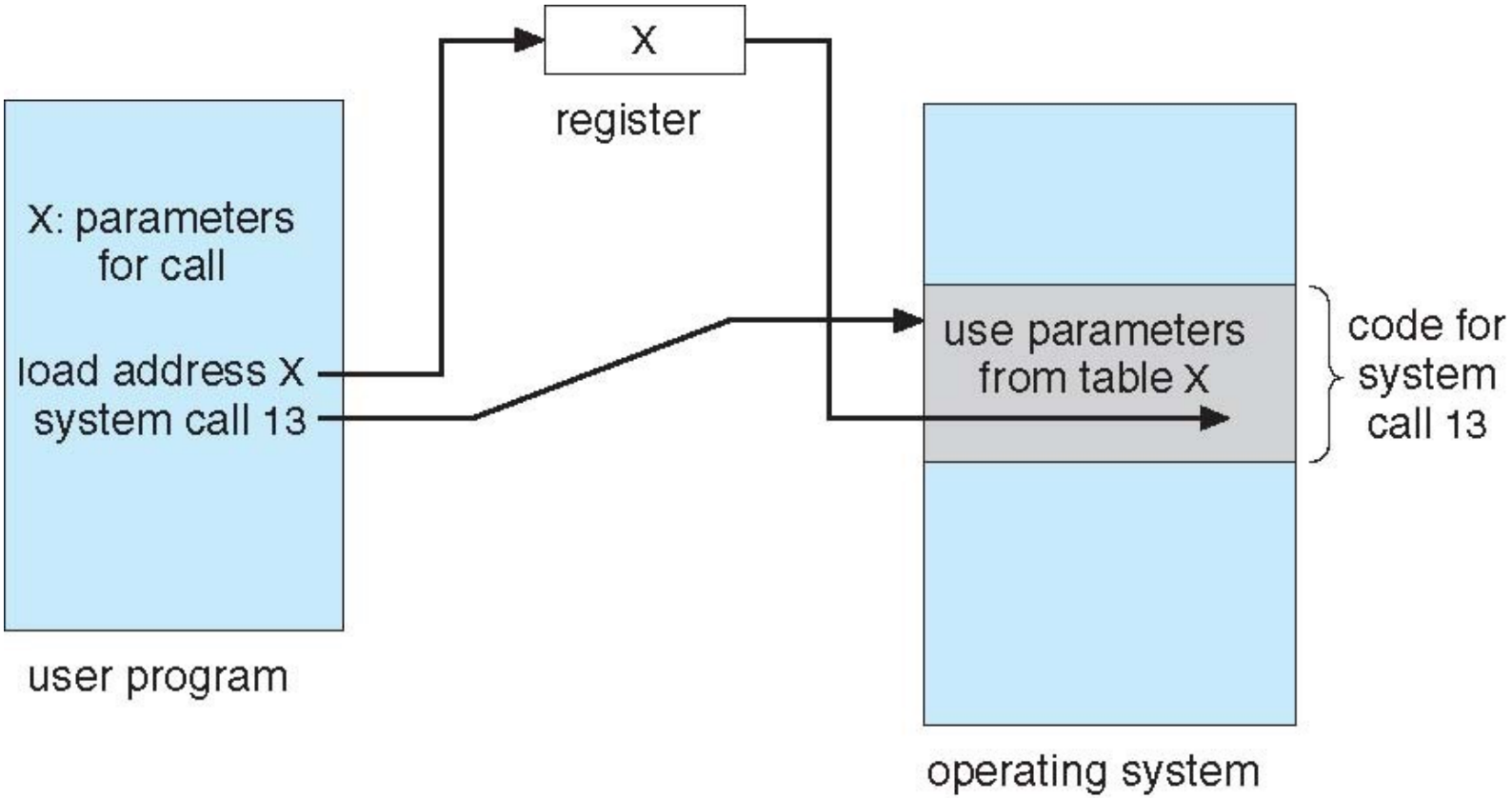
System Call Parameter Passing

- Often, more information is required than simply knowing the identity of the 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

System Call Parameter Passing

- Parameters stored in a block, or table, in memory, and address of **block** is passed as a parameter in a register
  - This approach taken by Linux and Solaris
- Parameters placed, or **pushed**, onto the **stack** 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**
  - For managing processes: e.g. create, terminate, load, execute process.
- **File management**
  - create, delete, open, close, read, write file etc.
- **Device management**
  - request/release device, read, write, reposition, logically attach or detach devices etc.
- **Information maintenance:** e.g. Get/set time or date, get/set system data, get/set process, file, or device attributes
- **Communications:** e.g. create, delete communication connection, attach and detach remote devices etc.
- **Protection:** e.g. Control access to resources, get and set permissions, allow and deny user access etc.

Examples of Windows and Unix 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()

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
- They are also called **system utilities**
- Constantly running system-program processes are known as **services, subsystems, daemons**
- System Programs can be divided into these categories:
  - File management
  - Status information
  - File modification
  - Programming-language support
  - Program loading and execution
  - Communications