

Operating-System Structures

بُنَى أنظمة التشغيل

Services, interfaces, system calls, and the internal architecture of a modern kernel

PRESENTED BY

Prof. Mostafa Elhosseini

Professor of Smart Systems Engineering

Six fundamental questions

01 WHAT services does the OS provide?

Operating-System Services

02 HOW do users interact with the OS?

GUI, CLI, Shell

03 HOW do applications request OS services?

APIs and System Calls

04 HOW are programs prepared and executed?

Compilers, Linkers, Loaders

05 HOW is the OS organized internally?

Monolithic, Layered, Microkernel, Modular, Hybrid

06 HOW does the OS start, and how can we examine it?

Booting, Debugging, Monitoring

2 . 3

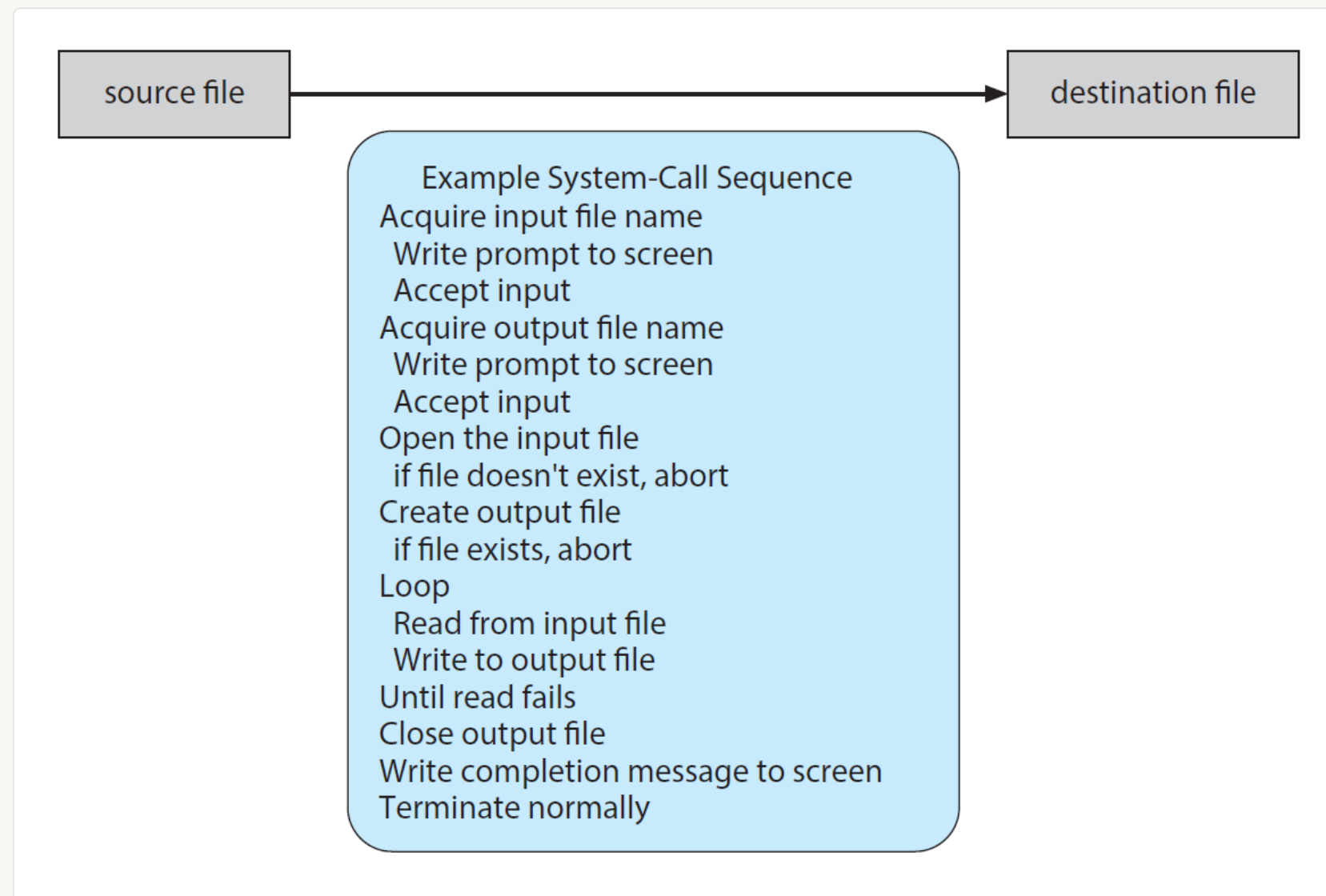
System Calls

استدعاءات النظام

The interface to the services made available by an operating system

Copying one file to another

- How a simple user operation, such as **copying a file**, requires **multiple interactions** between an application and the Operating System Kernel.



```
$cp in.txt out.txt
```

- Every step** — acquiring names, opening, creating, reading, writing, closing, reporting errors — is one or more system calls
- Error conditions must be handled at each call: missing file, protected file, existing output file, end of file, no disk space
- Frequently, systems execute **thousands of system calls per second**

Copying one file to another

- A System Call is a mechanism through which a program requests a service from the Operating System Kernel.
- **Imagine you are in a hotel.**
 - You are a guest, and you need a service from the hotel, such as room cleaning or food delivery.
 - You cannot simply enter the kitchen or staff facilities and perform those operations yourself.
 - Instead, you contact the reception desk and request the service.
- The cp program is a user-level application.

```
$cp in.txt out.txt
```

- Application: Hotel guest.
- System Call: Official request for a service.
- Kernel: Hotel staff responsible for providing the service.
- Hardware: Hotel facilities and equipment.

Programmers code to an API, not to system calls

- API (**Application Programming Interface**) — is a set of functions, rules, and conventions that allows one software component to use the services of another
- In **operating systems**, an API provides functions that applications can use to request services such as **opening files, creating processes, or reading data**.
- Instead of dealing directly with kernel-specific system-call mechanisms, you can use a familiar function such as: **read**
- The **API defines** how to request the operation, what parameters to provide, and what result to expect.
- The underlying implementation handles the necessary interaction with the operating system.

```
</> C  
read(fd, buffer, 100);
```

Programmers code to an API, not to system calls

■ Why an API

- **Simplicity.** System calls involve **low-level details** such as system-call numbers, parameter conventions, and transitions between User Mode and Kernel Mode. APIs hide many of these details.
- **Portability.** When different operating systems support the same API specification, programmers can often use similar source code across those systems.

Windows API

for Windows systems

POSIX API

UNIX, Linux, and macOS

Java API

for the Java virtual machine

code written using POSIX interfaces can often be compiled on Linux, macOS, and other POSIX-compatible environments with relatively few modifications

Programmers code to an API, not to system calls

■ Ordering food

- Imagine visiting different branches of a restaurant.
- The menu is the API: it tells you which services you can request and how to request them.
- The kitchen is like the Kernel: it performs the work.
- Each branch may organize its kitchen differently, but customers use the same menu.
- That's the value of a standardized interface.

Windows API

for Windows systems

POSIX API

UNIX, Linux, and macOS

Java API

for the Java virtual machine

Example of a standard API: `read()`

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)
```

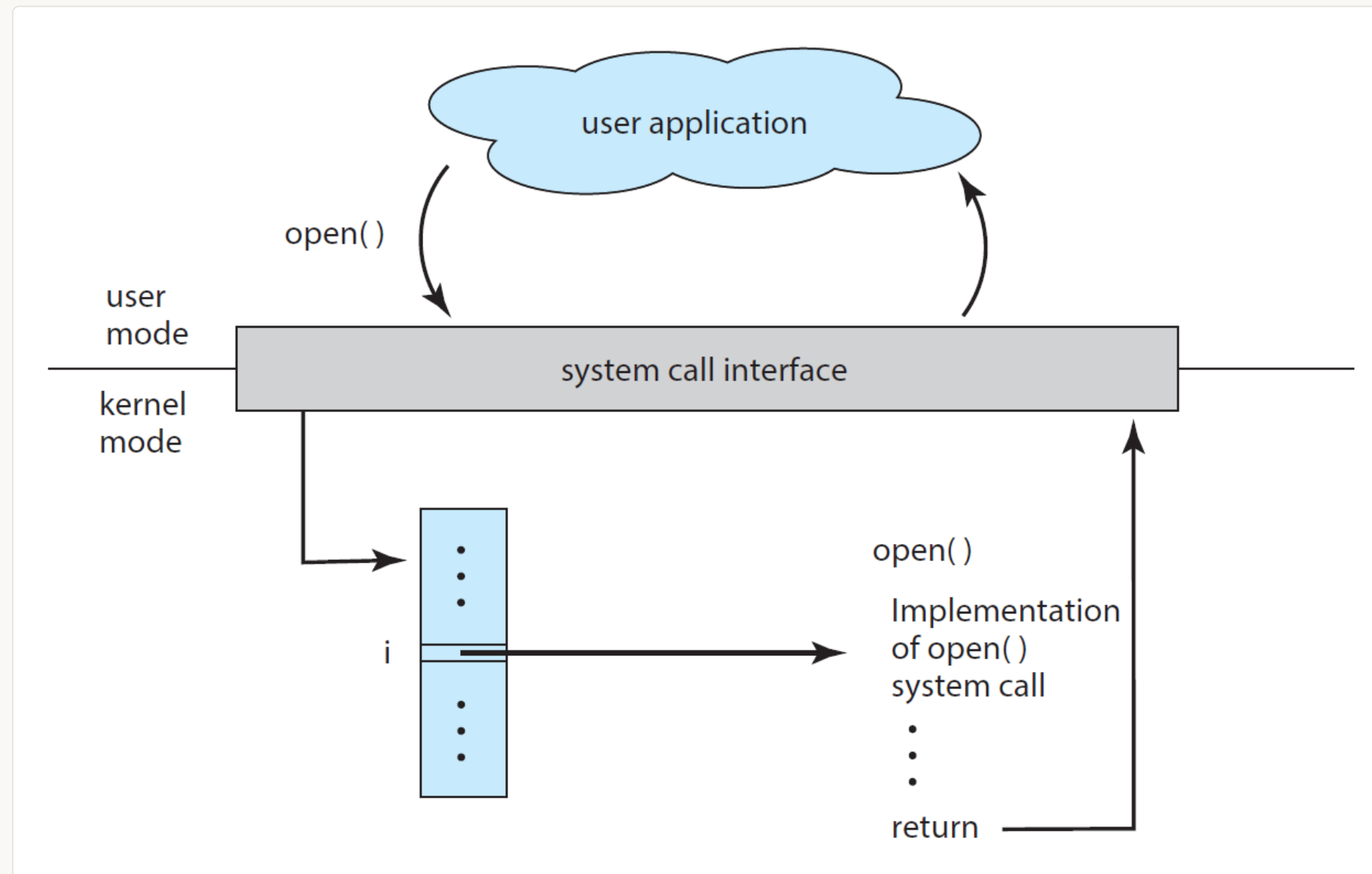
<code>ssize_t</code>	<code>read</code>	<code>(int fd, void *buf, size_t count)</code>
return value	function name	parameters

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 into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

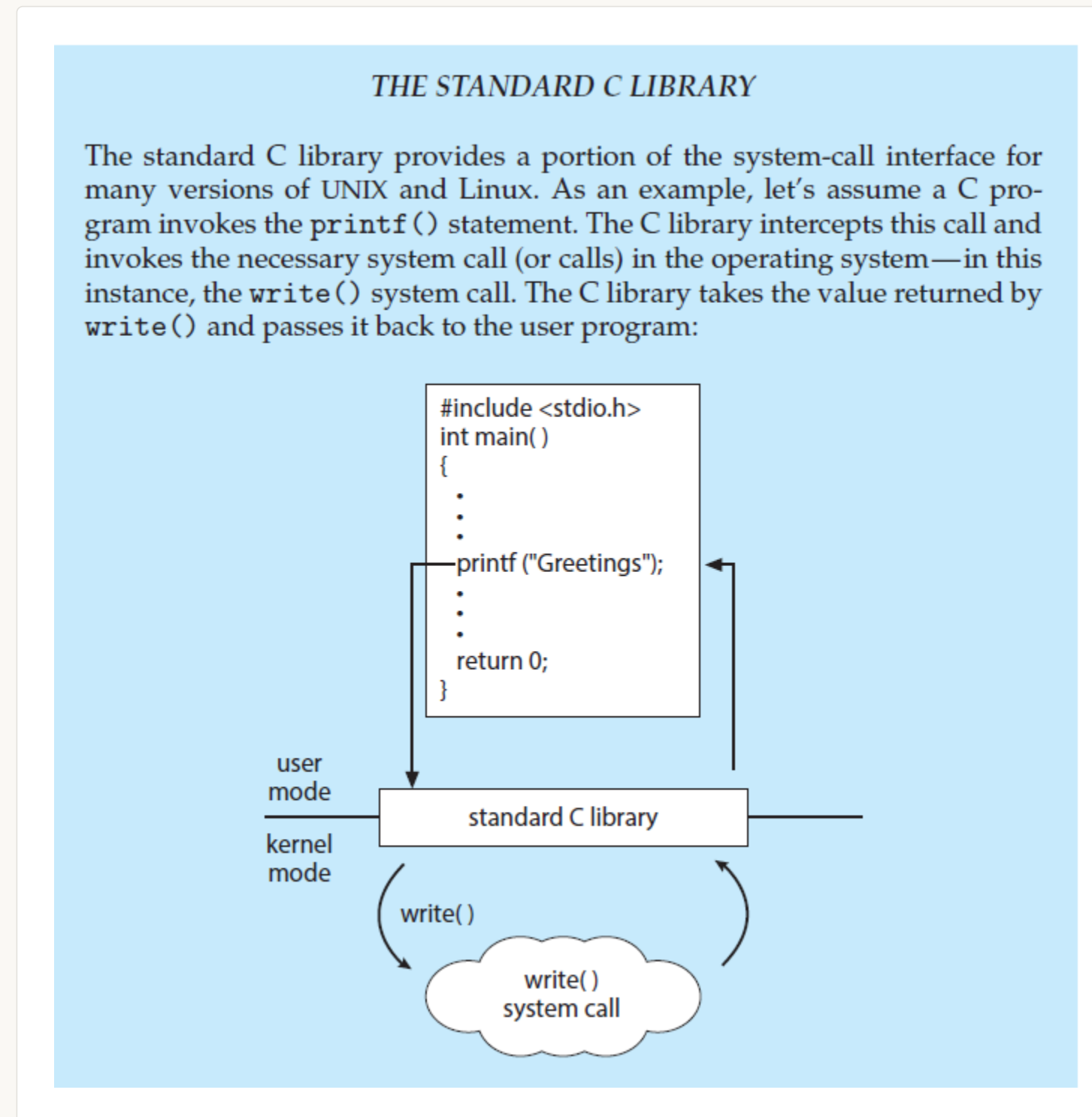
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.

Handling a call to `open()`

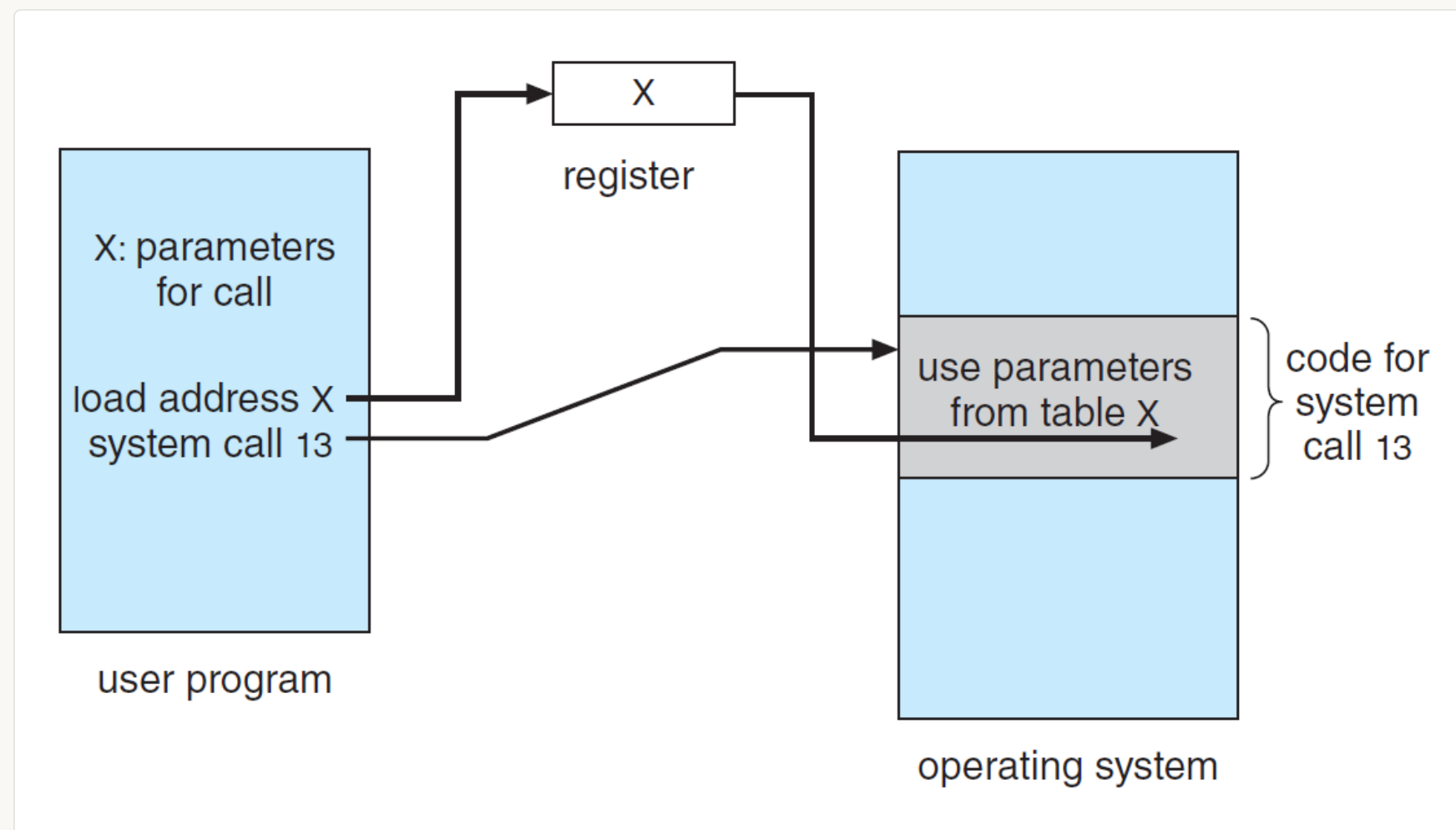


- The system-call interface intercepts the API function and invokes the intended system call in the kernel
- Control crosses from **user mode** into **kernel mode** and back with a status result
- The caller need know nothing about the implementation — only the API contract

From `printf()` to `write()`



Three ways to pass parameters to the kernel



- **Registers** — simplest, but limited in number
- **Block or table** — parameters in memory, the block address passed in a register
- **Stack** — pushed by the program, popped by the operating system

Linux combines them: registers for five or fewer parameters, the block method beyond that

Six categories of system calls

- Process control
 - create process, terminate process
 - load, execute
 - get process attributes, set process attributes
 - wait event, signal event
 - allocate and free memory
- File management
 - create file, delete file
 - open, close
 - read, write, reposition
 - get file attributes, set file attributes
- 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 process, file, or device attributes
 - set process, file, or device attributes
- Communications
 - create, delete communication connection
 - send, receive messages
 - transfer status information
 - attach or detach remote devices
- Protection
 - get file permissions
 - set file permissions

Names in the text are generic; each operating system has its own name for each call

Windows and UNIX system calls compared

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()

**Process control, file management ,
and device management —
Windows on the left, Unix on the right**

Windows and UNIX system calls compared

Communications	CreatePipe()	pipe()
	CreateFileMapping()	shm_open()
Protection	MapViewOfFile()	mmap()
	SetFileSecurity()	chmod()
	InitializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

Information maintenance, communications, and protection — same column order

The Runtime Environment

بيئة التشغيل

The software environment needed to run a program

A SIMPLE QUESTION

You wrote a program and compiled it. Is it completely ready to run on its own?

Not necessarily.

What is a Runtime Environment?

A **Runtime Environment (RTE)** is a collection of software components, libraries, and supporting services that provide the environment a program needs to **execute correctly**.

DURING EXECUTION, A PROGRAM MAY NEED

Libraries

Ready-made functions it calls

Memory support

Allocate, manage, release

A way to talk to the OS

Requests for kernel services

What happens when you run this?

```
#include <stdio.h>

int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- **Loader** — the OS loads the executable into memory and prepares it
- Runtime libraries — the program uses existing libraries, such as **glibc** on Linux
- **API** — `printf()` offers a convenient interface for displaying text
- **System call** — to reach the terminal, the library may use `write()`
- **Kernel** — the OS performs the actual output operation

From `printf()` to the kernel

Application

```
printf("Hello World!")
```

Runtime Library (glibc)

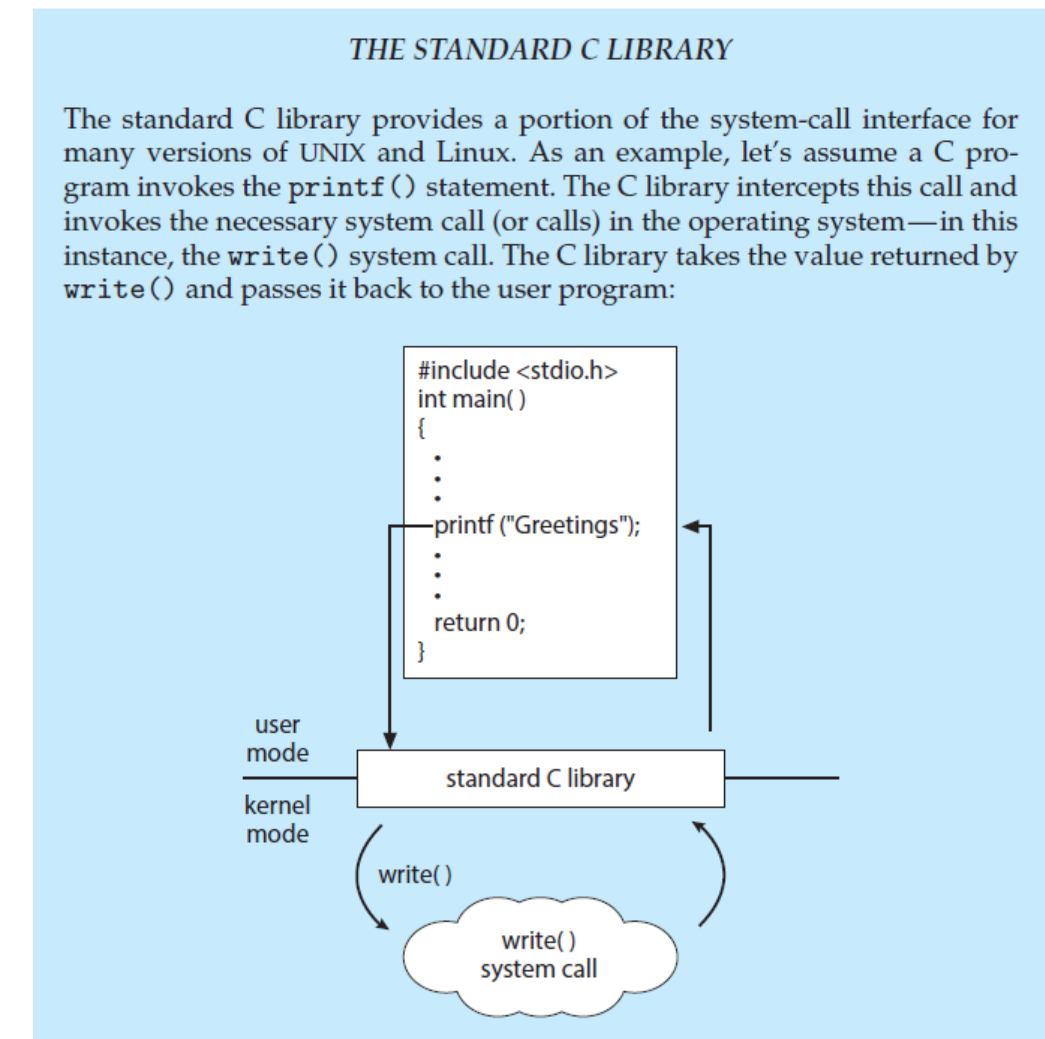
Prepares and manages the output operation

System-Call Interface

Transfers control to Kernel Mode when required

Operating System Kernel

Performs the requested service



C library handling of `write()`

What a Runtime Environment may include

COMPONENT	FUNCTION
Runtime Libraries	Provide reusable functions needed by programs
Loader	Loads executable programs and required shared libraries into memory
Memory Management Support	Helps programs allocate, manage, and release memory
System-Call Wrappers	Provide convenient interfaces for accessing kernel services
Runtime Services	Exception handling, garbage collection, or other language-specific functionality

Build time vs run time

Build time

The **compiler** prepares the program — before it ever runs

Run time

The **runtime environment** supports the program — while it runs

Exception: Java's runtime includes a **Just-in-Time (JIT) compiler** that compiles code **during** execution.

From `Hello.java` to execution

Java Source Code

`Hello.java`

Java Compiler (javac)

Produces bytecode

Bytecode

`Hello.class`

Java Runtime Environment

JVM + libraries — executes bytecode, manages runtime services

Operating System and Hardware

- The **JVM** is a major component of the Java RTE
- It provides memory management, bytecode execution, and interaction with the OS
- **Portability** — the same `.class` file runs on any OS with a compatible Java runtime

A chef needs a kitchen

Application

The **chef** who performs the work

Runtime Environment

The **equipped kitchen** the chef works in

Libraries

Ready-made **tools, ingredients, recipes**

Operating System

The **infrastructure** — power, water, building

KEY IDEA

The application contains the **instructions**; the Runtime Environment provides the **supporting software** needed to execute them.

Related — but not the same

CONCEPT	MAIN PURPOSE
Runtime Environment (RTE)	Provides the supporting software and services needed during program execution
API	Defines how an application requests particular functions or services
System Call	A mechanism for requesting a service from the operating-system kernel
Kernel	Implements the privileged operating-system services

Not every library call reaches the kernel

```
printf()
```

Runtime library → **may** issue `write()` → kernel

CROSSES INTO KERNEL MODE

```
strlen()
```

Counts characters in memory — **no system call** needed

STAYS IN USER MODE

The runtime library decides when an operating-system service is actually required.

Remember three statements

01 · Before

Compiler

Prepares the program for execution

02 · During

Runtime Environment

Provides supporting components and services while the program runs

03 · Underneath

Operating System

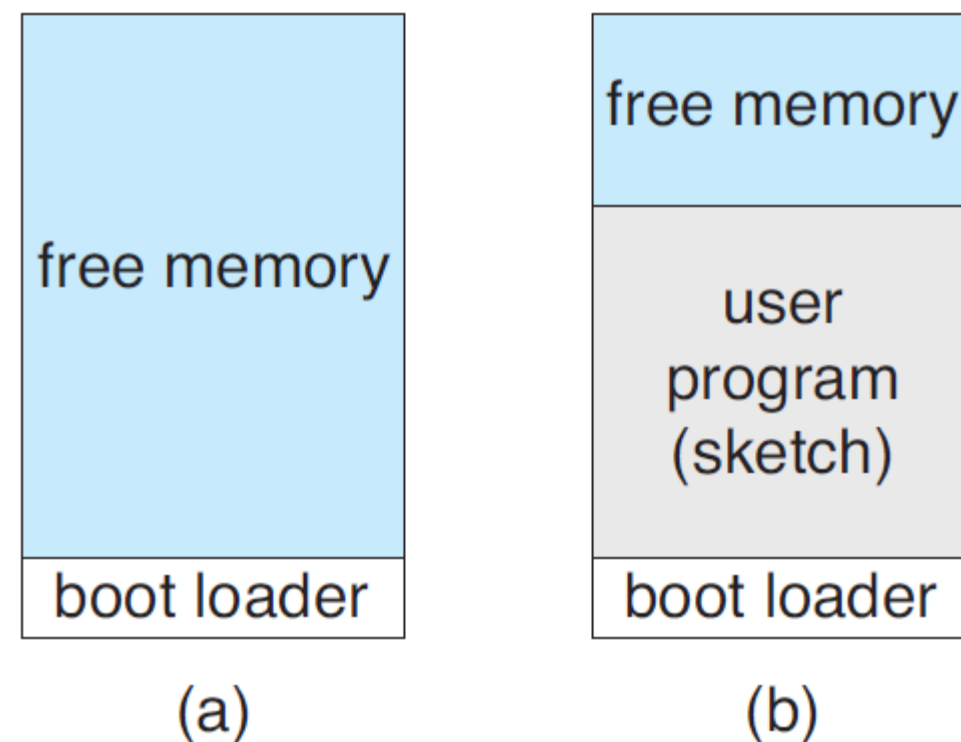
Provides system services and manages the hardware

IN ONE SENTENCE

The **Compiler** prepares the program, the **Runtime Environment** helps it run, and the **Operating System** provides the underlying services.

A single-tasking system: Arduino

- The Arduino program is called a **Sketch**.
- When we compile and upload the Sketch, the **bootloader** helps place the compiled program into Flash memory.
- simple Arduino microcontroller that **does not require a general-purpose operating system**.



- No operating system: a boot loader places the compiled program — a **sketch** — into a fixed region of flash memory
- Only one sketch can be in memory at a time; loading another replaces it. No user interface beyond hardware sensors

- **Not every** computing device needs a general-purpose operating system. A simple embedded controller can execute a program directly

Running a program with no operating system

01

Power ON

The microcontroller starts its startup instructions

02

Bootloader

If present, checks whether a new program needs uploading

03

Sketch execution

The uploaded **Sketch** starts running

04

Hardware control

LEDs, sensors, timers —
no OS system calls

SO WHO MANAGES THE CPU AND MEMORY?

CPU

Executes the Sketch directly

Memory

Managed by the program and its runtime code

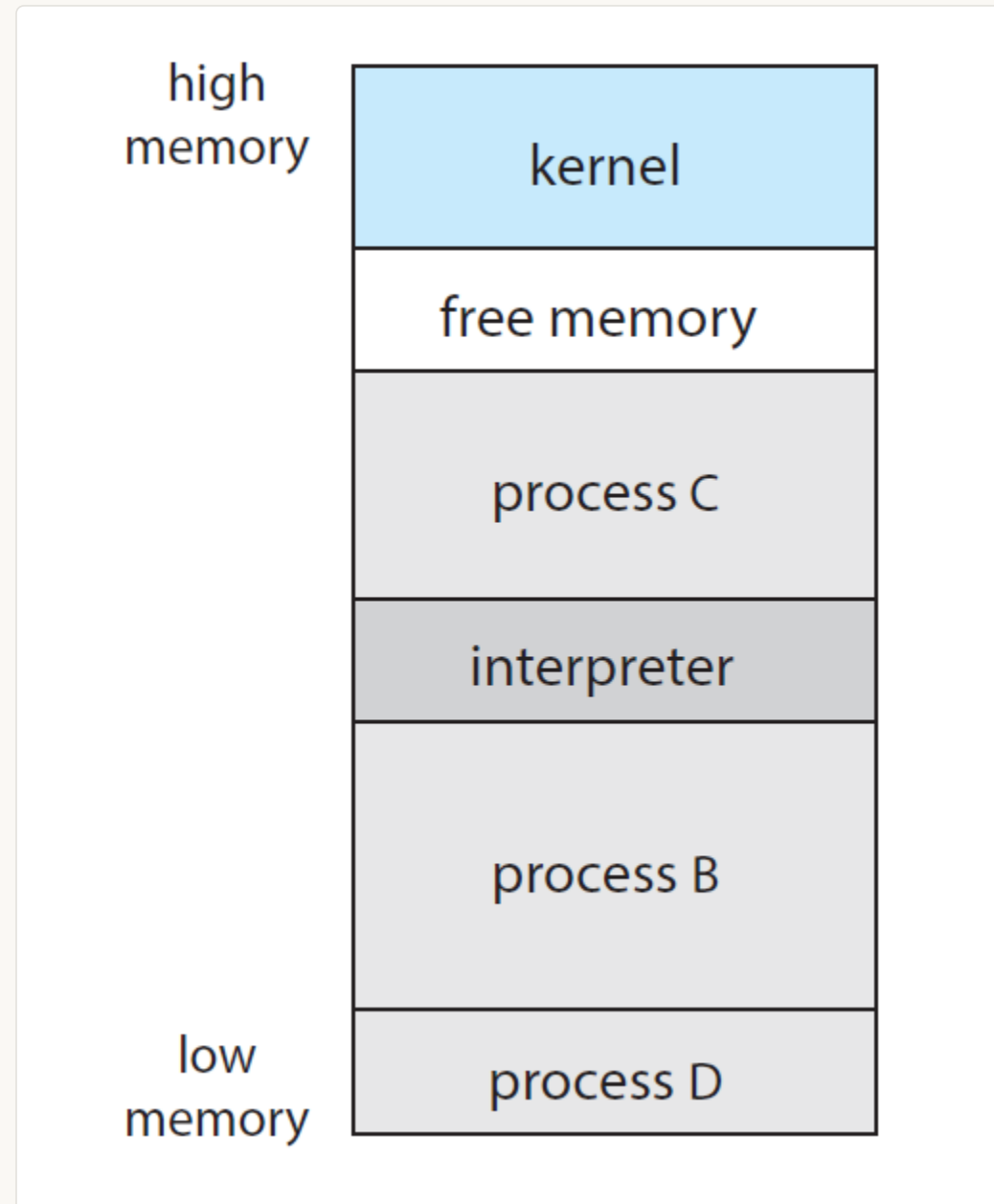
I/O

Through microcontroller registers and libraries

Scheduling

No general-purpose OS scheduler

A multitasking system: FreeBSD



- `fork()` — the shell creates a new process
- `exec()` — the selected program is loaded into memory and executed
- `exit()` — the process terminates, returning status 0 or a nonzero error code
- The shell either waits for the process or runs it “in the background,” where it cannot read the keyboard directly

One worker in a workshop, or a factory with a manager?

SIMPLE ARDUINO	LINUX COMPUTER
No operating system	Has an operating system
Runs one main Sketch	Runs multiple processes
Direct hardware access	Hardware access mediated by the OS
No process scheduler	Kernel schedules processes
Bootloader uploads programs	Bootloader loads the OS kernel

Bootloader ≠ OS

It only starts or loads other software.
Advanced boards can run an **RTOS** such as FreeRTOS.

ANALOGY

Arduino is a **workshop with one worker** . Linux is a **factory** that needs a manager — the OS.

Key takeaway: an OS is not mandatory — it becomes valuable for multiple processes, shared resources, protection, and complex operations.

2 . 4 – 2 . 6

System Services, Linkers, and Loaders

خدمات النظام، الرابط، والمُحمِّل

The software around the kernel — and why programs do not cross between systems