

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

TWO FUNDAMENTAL QUESTIONS

01 How does a program move from *source code* to a *running process* ?

Compiler · Linker · Loader

02 Why can a program compiled for one OS *fail to run* on another?

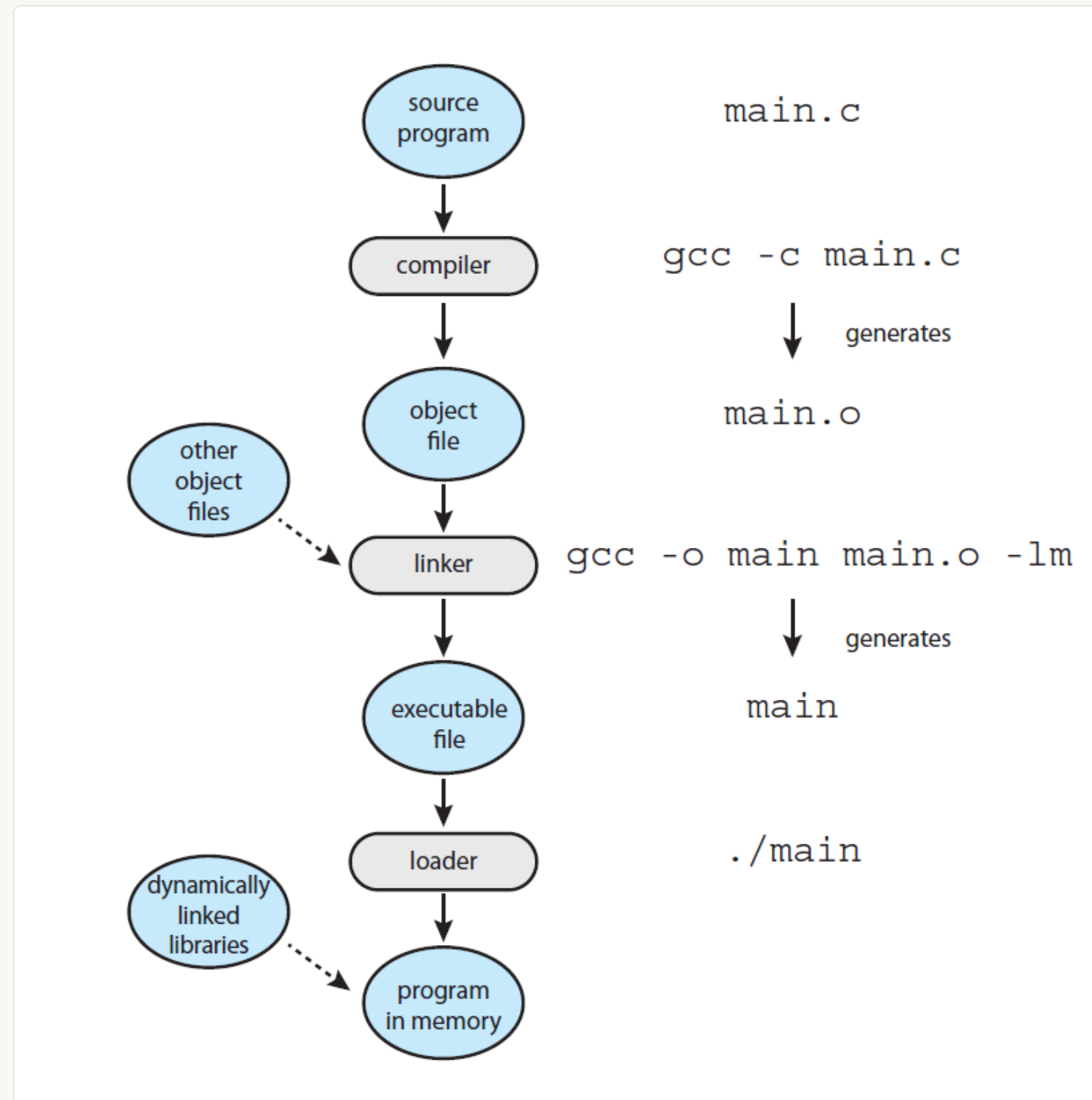
Formats · Instruction sets · System calls · ABI

2 . 4 – 2 . 6

Compiler, Linkers, and Loaders

المترجم، الرابط، والمُحمِّل

From source file to a program in memory



PART 1

From Source Code to Execution

من الكود المصدري إلى التنفيذ

Can the CPU directly execute C code?

Can the CPU directly execute this?

FILE: main.c

```
#include <stdio.h>

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

No.

- The CPU understands **machine instructions** — not C source code
- The program must pass through **several stages** before it can run

Seven stages from `main.c` to a process

01 Source Code

`main.c`

Human-readable C instructions



02 Compiler

`gcc -c main.c`

Translates into object code



03 Object File

`main.o`

Machine code + symbols
+ relocation info



04 Linker

`gcc -o main main.o`

Combines and resolves components



05 Executable File

`main`

Executable stored on disk



06 Loader

`./main`

OS prepares the program in memory



07 Running Process

`CPU executes`

Program becomes active

Building and driving a car

Compiler

Manufactures the parts
from engineering instructions

Source code → object code

Linker

Connects the parts into a
complete car

Object files + libraries → executable

Loader

Prepares the car to operate
on the road

Executable → ready in memory

Building a program and **running** a program are two different activities.

What Does the Compiler Do?

المُترجم

From high-level language to object code

Inside an object file

Translates a **high-level** program into **lower-level** code

```
$ gcc -c main.c
```

`-c` — compile only, **no final linking** → `main.o`

COMPONENT	PURPOSE
Machine Code	Compiled CPU instructions
Symbol Table	Names and info about functions and variables
Relocation Info	References that may need address adjustment
Other Metadata	Useful to the linker and debugging tools

Why `main.o` is not ready to run

```
int calculate(int x);    // elsewhere

int main(void) {
    return calculate(5);
}
```

- The compiler translates the call — the **final address is unknown**
- The object file **records the unresolved reference**
- The linker resolves it later

Relocation

Adjusting address-dependent references **after** components are assigned positions — during linking or loading

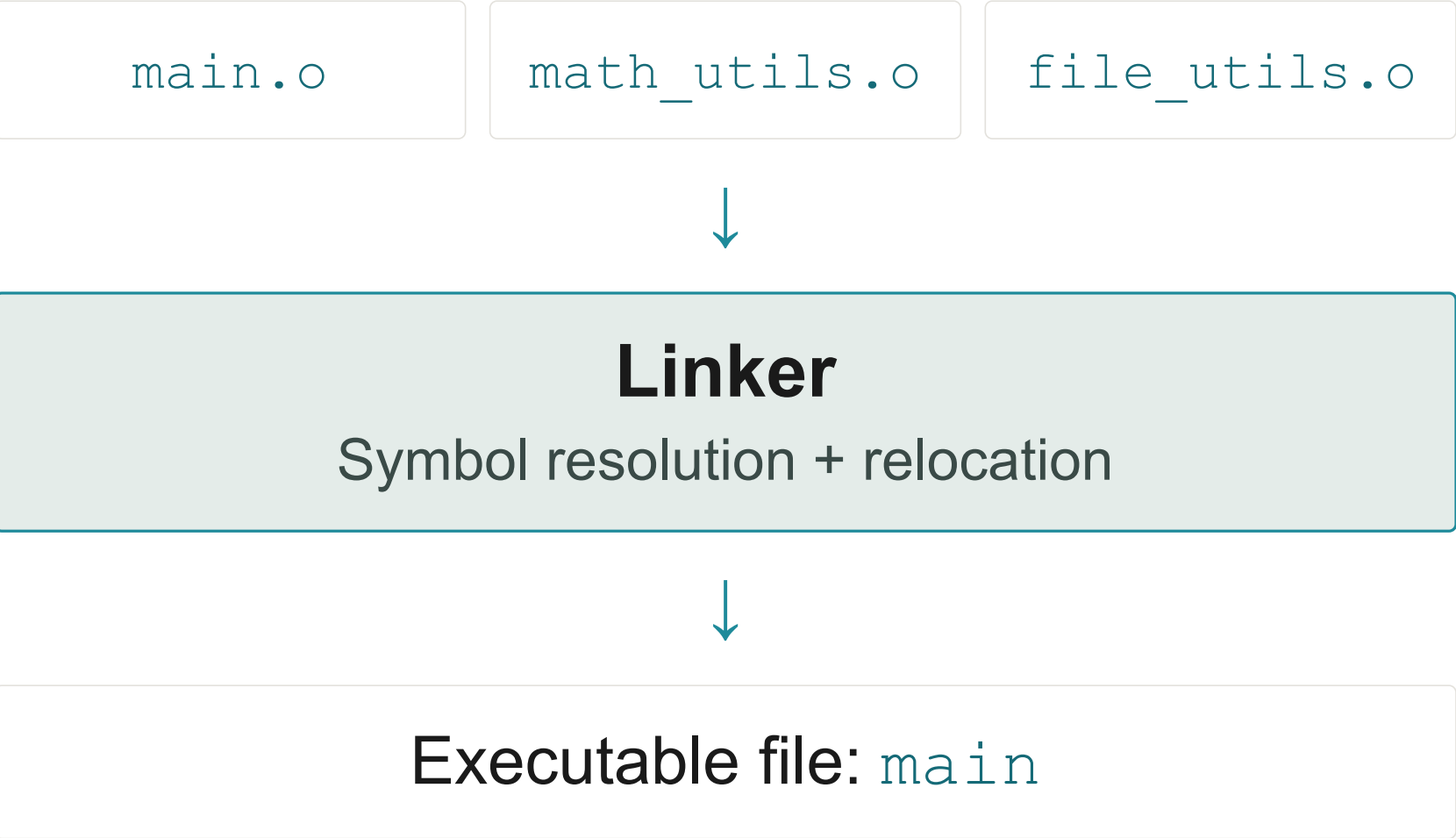
ANALOGY “Send this to the **Finance Department**” — before its room number is assigned. Once the building is organized, the reference finds the right room.

What Does the Linker Do?

الرابط

Combining object files and resolving references

Combine and resolve



```
$ gcc -o main main.o -lm
```

PART	MEANING
gcc	Invokes the toolchain
-o main	Names the executable <code>main</code>
main.o	The compiled object file
-lm	Requests the math library — <code>sin()</code> , <code>cos()</code>

PRACTICAL EXAMPLE

The Linker in Code

الرابط عمليًا

What does the Linker actually do after compilation?

THE CENTRAL QUESTION

If the Compiler already produced machine code — why do we still need a *Linker*?

Compiler

Translates **individual** source files

Linker

Connects compiled parts and **resolves**
references

A simple calculator in two files

FILE: `main.c`

```
#include <stdio.h> // implemented in another file

int main(void) {
    int result = add(10, 20);
    printf("Result = %d\n", result);
    return 0;
}
```

FILE: `math_utils.c`

```
int add(int a, int b) {
    return a + b;
}
```

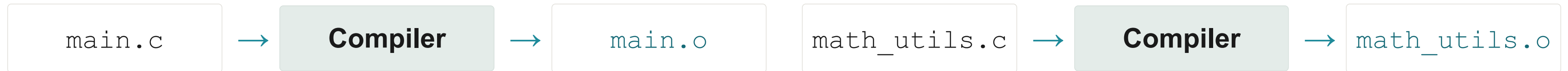
`int add(int a, int b);` tells the Compiler that `add()` **exists** — but its body is **not** in `main.c`

- `main.c` **calls** `add()`
- `math_utils.c` **provides** it

Two object files — no executable yet

```
$ gcc -c main.c $ gcc -c math_utils.c
```

`-c` — compile, but **do not perform the final linking step**



Machine code and linking info — but **not yet a complete executable**

Reading symbols with nm

```
$ nm main.o
                 U add
000000000000000000 T main
                 U printf
```

```
$ nm math_utils.o
000000000000000000 T add
```

Simplified, illustrative output

SYMBOL	MEANING
T main	main() is defined in this object file
U add	add() is referenced — not defined here
U printf	printf() is referenced — not defined here
T add	add() is defined in <code>math_utils.o</code>

U = Undefined *here* — not “doesn’t exist anywhere”

One file needs — another provides

`main.o`

Needs `add()`

`U add`

LINKER

resolves and
combines

`math_utils.o`

Provides `add()`

`T add`

Executable: `calculator`

A linker error

```
$ gcc main.o -o calculator  
undefined reference to `add'  
ld returned 1 exit status
```

- The Compiler knew `add()` was **declared**
- The Linker found **no implementation** in the supplied files or libraries

Compiler error

Something is wrong with the **source code** or its translation

Linker error

Required **definitions** or binary references **cannot be resolved**

Linking succeeds

```
$ gcc main.o math_utils.o -o calculator
$ ./calculator Result = 30
```

GCC also supplies **startup code** and **standard libraries**

1 Sees that `main.o` **requires** `add()`

2 Locates its **definition** in `math_utils.o`

3 **Connects** the reference to the implementation

4 Produces the executable `calculator`

Separately compiled components → **one working executable**

We never wrote `printf()` — so who did?

```
U printf
```

- `printf()` comes from the **C Standard Library** — **glibc** on Linux
- GCC links the standard C library **automatically**
- With **dynamic linking**, the executable records the dependency; the **dynamic loader** resolves it at run time

`add()`

Implemented in **our own** `math_utils.o`

`printf()`

Provided by an **external standard library**

What if a function is missing?

☐ `math_utils.o` NOT included

```
$ gcc main.o -o calculator
```

Linking failed

The Linker cannot find the definition of `add()` .
No executable is produced.

☒ `math_utils.o` included

```
$ gcc main.o math_utils.o \  
-o calculator
```

Linking successful

The Linker finds `add()` and produces
`calculator` → **Result = 30**

Why separate compilation matters

`memory.c`

Memory management

`scheduler.c`

changed → recompile only this

`filesystem.c`

File system

`... hundreds more`

- Recompiling **everything** every time is inefficient
- Compile files **individually**, then **link** the object files
- One file changes → recompile **only that file** , then link again

Separate compilation + linking = **fast builds** for large systems

Building a car

PROGRAMMING	CAR
Source Code	Engineering designs
Compiler	Factory making components
Object Files	Engine, wheels, transmission
Linker	Assembly line
Executable	Completed car
Loader	Preparing it to operate
Running Process	Car on the road

ANALOGY The design needs an **engine** — but the engine is **missing** . Every other part is made, yet the car can’t be finished.

That is a **missing symbol** during linking.

In their own words

Compiler

“I translated the source code into machine code.”

Compilation → object files

Linker

“I connected the compiled components and resolved their references.”

Linking → a complete executable

Loader

“I prepared the executable in memory so the CPU can run it.”

Loading → ready for execution

What Does the Loader Do?

المُحمِّل

From an executable on disk to a running process

What happens on `./main`

```
$ ./main
```

The Shell asks Linux to run it — commonly via `execve()`

Disk

`main` — executable file

OS Loader

Prepares segments and state

Virtual memory

Code · Data · Heap · Stack

CPU

Begins executing

- 1 Identify the format**
Recognize **ELF**, read its loading info
- 2 Create the memory image**
Set up virtual address-space mappings
- 3 Map program segments**
Code and data
- 4 Arrange dynamic linking**
Load and resolve shared libraries
- 5 Initialize execution state**
Stack, arguments, environment, CPU state
- 6 Begin execution**
Jump to the **entry point**

A file on disk is not a process

EXECUTABLE FILE	RUNNING PROCESS
Stored on disk	Executing or ready
Passive	Active execution entity
Code and metadata	Execution state + address space
<code>main</code>	A running instance of <code>main</code>

ANALOGY A **recipe book** on the shelf is the executable; a **chef following it** is the process.

Demand paging

Pages are mapped into virtual memory and loaded into RAM **only when needed**

ANALOGY You don't copy all **500 pages** of a textbook before reading chapter one.

Carry your tools — or share them

Static linking

App code + Library code = Larger executable

- Library code **copied into** the executable
- Usually **larger**; fewer runtime dependencies
- Resolved mainly **at link time**

A chef with a personal toolbox

Dynamic linking

Application → Dynamic loader → Shared library

- Shared libraries **stay separate**
- Often **smaller**; needs compatible libraries
- Some references resolved **at load or run time**

A shared equipment room

Windows

`.dll`

Linux

`.so`

macOS

`.dylib`

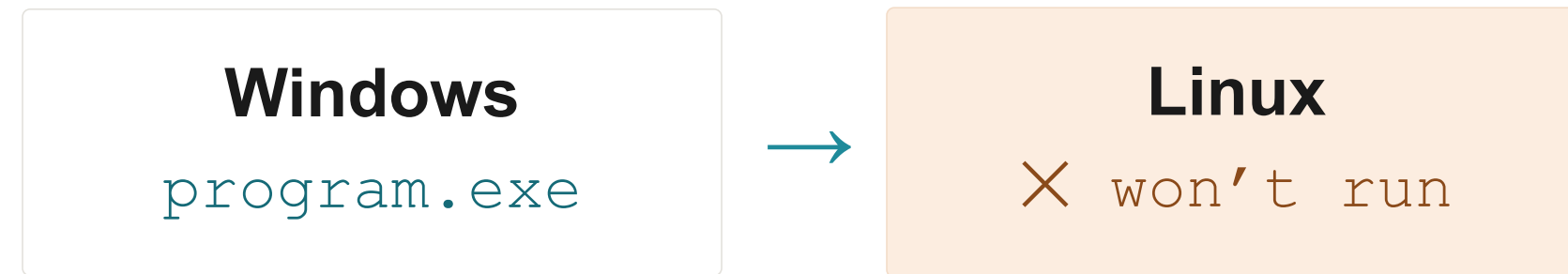
PART 2

Application Portability

قابلية نقل التطبيقات

Why can't a Windows program run directly on Linux?

Copy `program.exe` to Linux — and run it?



Normally, no.

Different operating systems follow different rules for:

- Executable files
- Kernel services
- Binary interfaces

A recipe used in different countries

ROUTE 1

Interpreted language

Python · Ruby

`hello.py`

Send the recipe to a local chef who **understands it**

بنكتب البرنامج بلغة Python، ونقدر نشغل نفس الـ Source Code على Windows أو Linux أو macOS، بشرط وجود Python Interpreter متوافق والمكتبات المطلوبة.

ROUTE 2

Virtual machine

Java bytecode + JVM

`Hello.class`

Same **standardized kitchen** in every country

بنحوّل برنامج Java إلى Bytecode، وبعد كده الـ JVM الموجودة على كل نظام تشغيل بتنفذ نفس الـ Bytecode.

ROUTE 3

Standardized API

POSIX C — recompile

`main.c`

Common recipe, **adapted to local equipment**

نفس Source Code غالبًا مع Recompile لكل نظام

These routes **support** portability — they do not guarantee every program runs unchanged everywhere.

Barriers 1 and 2

01 Executable format

OS	FORMAT
Linux	ELF
Windows	PE
macOS	Mach-O

Tells the **loader** how code, data, and metadata are organized

02 CPU instruction set

x86-64

ARM64

Both may use ELF — but contain **different machine instructions**

x86-64 code cannot normally run on ARM64 without **translation or emulation**

Barriers 3 and 4

03 System calls

POSIX	WINDOWS
<code>open()</code>	<code>CreateFile()</code>
<code>read()</code>	<code>ReadFile()</code>

API-level examples — native kernel mechanisms differ **further**

04 ABI

- How **parameters** are passed
- Which **registers** are used
- How the **stack** is organized
- Data **sizes and alignment**
- How **results** are returned

ANALOGY Same language, different hand-over rules:
right hand or on the desk?

Source-level vs binary-level contract

API	ABI
Application Programming Interface	Application Binary Interface
Source-code-level contract	Binary-level contract
Available functions and behavior	Calling conventions, data layout
Source-code portability	Binary compatibility

API tells the programmer how to call a function. **ABI** tells machine code how that call is performed.

```
int add(int a, int b);
```

API LEVEL
Takes two integers, returns an integer

ABI LEVEL
Which registers hold `a`, `b`, and the result?

Questions 1 – 3

Q1 A C file has been compiled into `main.o` . What is normally needed to combine it with other object files?

A. Loader

B. Linker ✓

C. Shell

D. CPU Scheduler

The Linker resolves external references and combines object files.

Q2 Which component prepares an executable's process image for execution?

A. Compiler

B. Text Editor

C. Loader ✓

D. Assembler

The Loader maps and prepares the executable in memory.

Q3 What is the main advantage of dynamic linking?

A. Every library is duplicated

B. Sharing libraries between applications ✓

C. Eliminating all dependencies

D. Removing the kernel

Compatible shared-library code is reused across applications.

Questions 4 – 5

Q4 Why might an x86-64 ELF binary fail on an ARM64 Linux computer?

A. Different CPU instruction sets ✓

B. Linux cannot load ELF

C. Source code is missing

D. The file has no extension

ELF is a file format; instructions must still match the CPU.

Q5 Which statement correctly distinguishes API from ABI?

A. API is hardware; ABI is software

B. API is source-level; ABI is binary-level ✓

C. API and ABI are identical

D. ABI is only used by Java

APIs define programming interfaces; ABIs define machine-level conventions.

Five concepts to remember

Compiler	Translates source code into object code
----------	---

Linker	Combines object files and resolves references
--------	---

Loader	Prepares an executable for execution
--------	--------------------------------------

API	Defines the source-level programming interface
-----	--

ABI	Defines binary-level compatibility rules
-----	--

THE CENTRAL MESSAGE

The **Compiler** prepares the code, the **Linker** builds the executable, the **Loader** prepares it to run — and the OS's **formats and interfaces** decide whether it can run on a given platform.

Operating-System Design and Structures

تصميم وبنية أنظمة التشغيل

How should we organize an OS so it is fast, reliable, secure, flexible, and easy to maintain?
