System Software

A-Level Computer Science

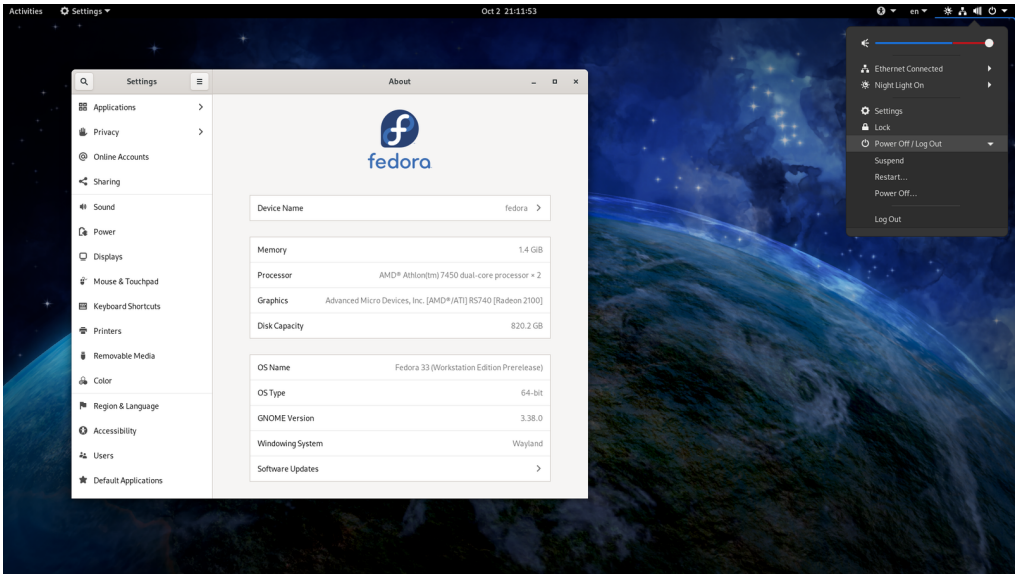
Operating systems

Why a computer needs an OS

Hardware on its own can only fetch and run instructions — it knows nothing about files, programs, networks or users. The **operating system** 操作系统 (OS) is the **software layer** that:

- **manages the hardware** (**processor** 处理器, memory, I/O, storage) for the running programs.
- **provides services** (file system, network, user accounts) through a clear interface, so programs need not talk to the hardware directly.
- **provides a user interface** (command line, GUI, touch).
- **lets several programs share the hardware safely** — each gets fair CPU time and is kept out of the others' memory.

Without an OS, every program would need its own drivers, and only one program could safely run at a time.

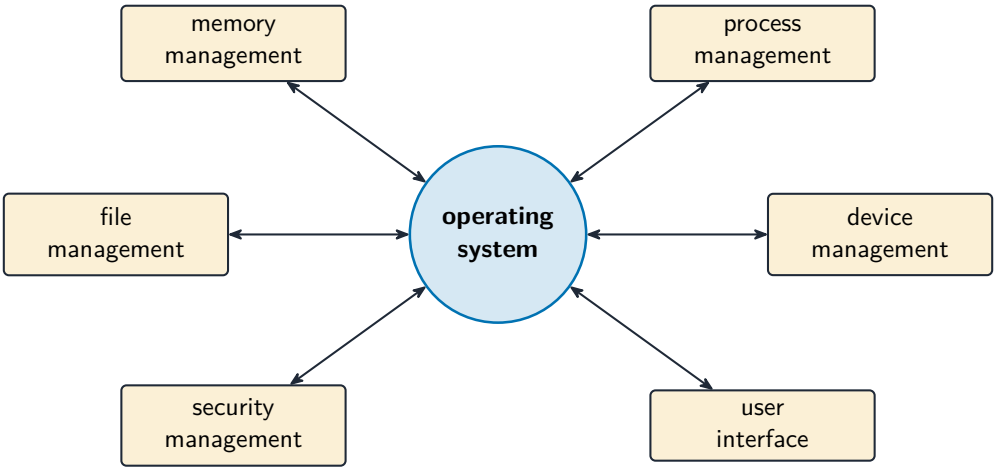


A desktop operating system manages the screen, files and programs for the user

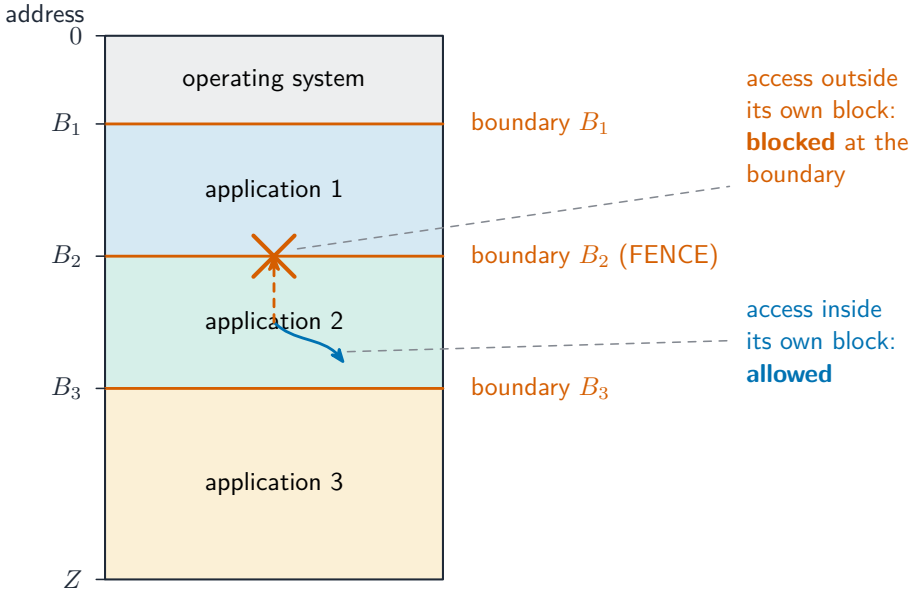
A smartphone runs a mobile operating system such as Android

Key management tasks

- **process management** 进程管理 — load programs, schedule them on the CPU, switch between them, and kill misbehaving ones. (A running program is a **process** 进程.)
- **memory management** 内存管理 — give memory to processes, keep them apart, and use **virtual memory** 虚拟内存 / **paging** 分页 so the working set can exceed physical **RAM** 随机存取存储器.
- **file management** — organise files and folders on **secondary storage** 辅助存储器, control permissions, prevent write corruption.
- **device management** (hardware management) — handle I/O and peripherals through **device drivers** 设备驱动, buffer data, manage interrupts, and give a uniform interface.
- **security management** — accounts, permissions, firewall, encryption.
- **user interface** and **networking**.



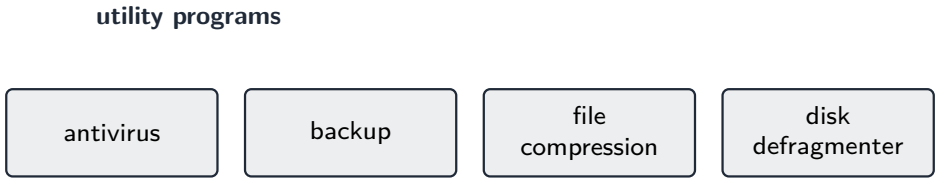
The main jobs the operating system manages



Memory protection keeps each application in its own block of memory

Utility software

Most OSes include **utility programs** 实用程序 to maintain the system:



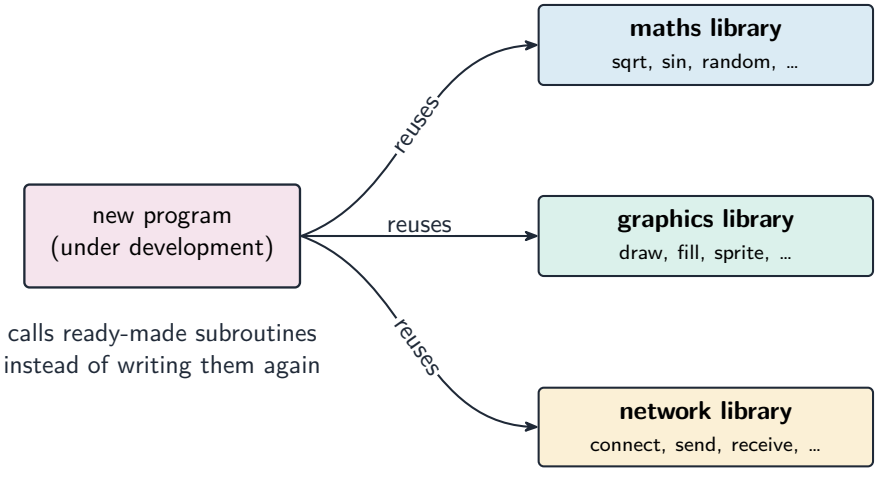
Utility programs: antivirus, backup, compression and defragmenter

- **disk formatter** — a **file / disk management** tool that prepares a disk (sets up its file system); related tools copy, move and delete files.
- **defragmentation software** (**disk defragmenter** 碎片整理) — moves the pieces of fragmented files together on a hard disk to cut seek time (not useful on SSDs).
- **disk contents analysis / disk repair software** — check disk integrity and fix file-system errors and bad sectors.
- **back-up software** (**backup** 备份) — copies user data elsewhere so it can be recovered.
- **virus checker** (**antivirus** 杀毒软件) — scans for malware and quarantines threats.
- **firewall** 防火墙 — filters network traffic by rules.
- **file compression** (**compression** 压缩) / archiving; **system monitor**; **updates**.

Bundling these with the OS saves the user installing each one.

Program libraries

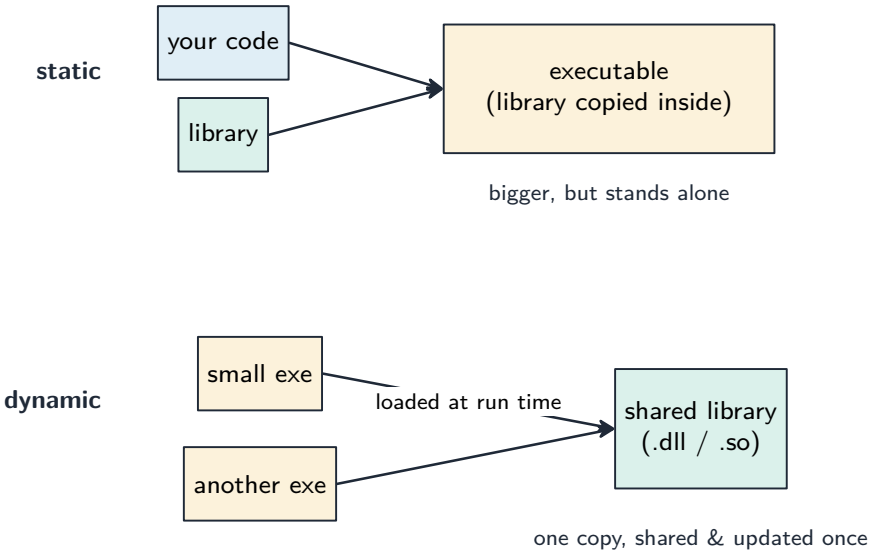
A **program library** 程序库 is pre-written code (**subroutines** 子程序, classes, modules) that programs reuse instead of writing it themselves — e.g. a maths library, a network library, a graphics library.



A new program reusing ready-made routines from libraries

Benefits: **saves time** (off-the-shelf code), **reliable** (well-tested, widely used), and **standardised** (consistent behaviour).

- a **static library** 静态库 is copied into the executable at compile time (stands alone, but larger and needs rebuilding to update).
- a **dynamic library** 动态库 (DLL, Dynamic Link Library; `.so`) is loaded at run time (smaller executables, shared by many programs, updated once for all).



Static: the library is copied into the executable. Dynamic: a shared library file is loaded at run time

Language translators

You write source code; the computer runs **machine code** 机器码. A **translator** 翻译器 converts between them.

Assembler

An **assembler** 汇编器 translates **assembly language** 汇编语言 into machine code, one instruction per instruction. Used for low-level code (embedded systems, drivers).

Compiler

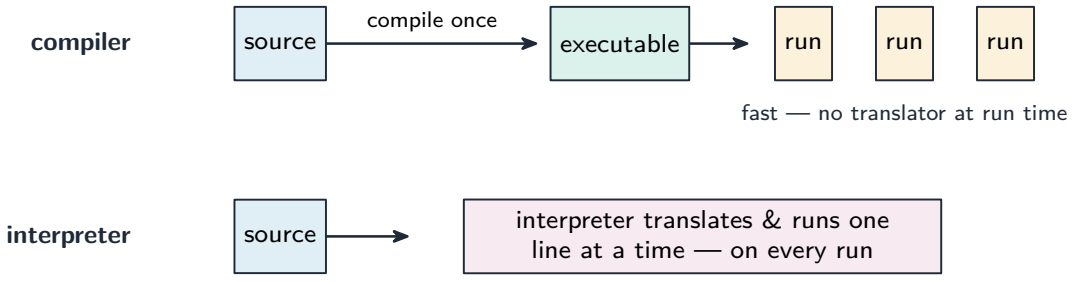
A **compiler** 编译器 translates a high-level program into machine code **once, before it runs**.

- it reports **all** errors at compile time; once clean, it produces a stand-alone **executable** 可执行文件 that runs **without the compiler installed** and can be run many times.
- generally **faster at run time** (no translation while running), but tied to one CPU/OS — recompile for each platform.

Interpreter

An **interpreter** 解释器 translates and runs a high-level program **one line at a time**, producing no executable.

- it reports an error **when it reaches that line**, then stops; you can fix it and continue — good for development.
- the **interpreter must be installed** to run the program; generally **slower** (each run re-translates), but easy to **port** across platforms.



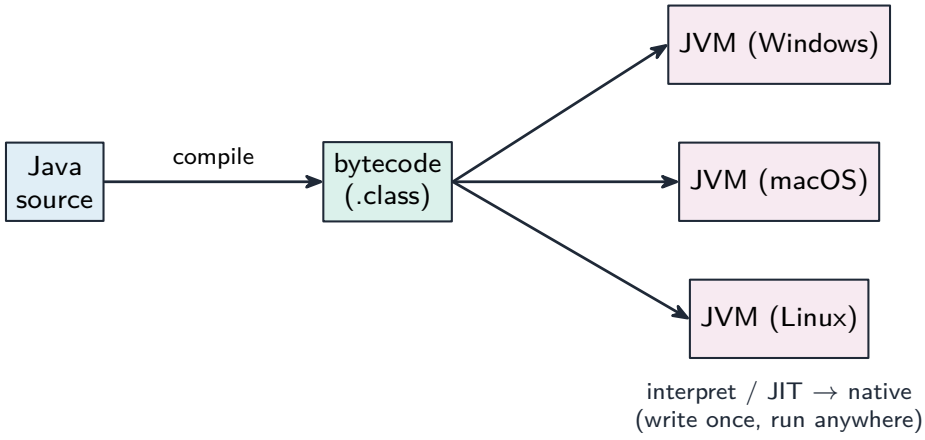
A compiler translates once into a standalone program; an interpreter translates line by line, every run

Choosing between them

Use a compiler when:	Use an interpreter when:
run-time speed matters	you want fast edit–run cycles
distributing to users without dev tools	writing cross-platform scripts
the program runs many times	the program is small or run once
	teaching beginners

Hybrid: Java

Java is compiled into **bytecode** 字节码 (a platform-independent intermediate form), which a **virtual machine** 虚拟机 (the JVM) then interprets — or uses **just-in-time compilation** 即时编译 to turn hot parts into native code. So errors are caught early, the bytecode runs anywhere with a JVM ("write once, run anywhere"), and long-running programs reach near-native speed. C# and Python use similar designs.

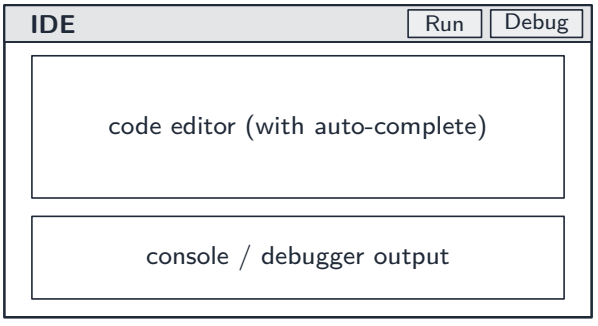


Java compiles to portable bytecode that any JVM runs — write once, run anywhere

Worked example. Java source is compiled to **bytecode**, which a JVM then interprets. Why use both, instead of compiling straight to machine code? A compiler produces machine code for **one** processor and operating system, so a program compiled on one machine will not run on another. Java's compiler instead targets a **virtual** machine, so the bytecode it produces is identical everywhere; each platform then supplies its own **JVM** to interpret that bytecode into its own native instructions. One compiled file therefore runs anywhere a JVM exists - "write once, run anywhere". The price is speed: interpreting bytecode is slower than running native code, which is why a real JVM also uses **JIT** compilation to turn frequently-run bytecode into native code while the program runs. Name **both** sides - the marks are for portability **bought at the cost of speed**.

Integrated Development Environment (IDE)

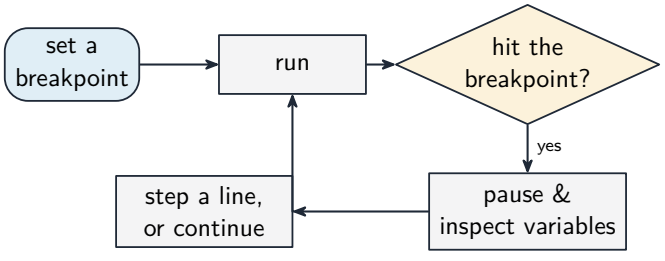
An **integrated development environment** 集成开发环境 (IDE) brings the tools to write, test and debug code into one application:



An IDE bundles the editor, a Run button and a debugger

- **source code editor** with **syntax highlighting** 语法高亮 (keywords, strings, comments in different colours), auto-indent and bracket matching.
- **auto-complete** 自动补全 — suggests names and shows function parameters as you type.
- **translator integration** — compile/run with one keystroke; errors shown inline.
- **debugger** 调试器 — set **breakpoints** 断点 to pause, **step through** line by line, and **inspect variables**.
- **version control** 版本控制 integration (git), **project management**, a help system, **refactoring** 重构 tools (safe renaming), and **unit test** 单元测试 integration.

An IDE speeds development by putting writing → running → debugging → fixing behind one interface. Common IDEs: Visual Studio, PyCharm, Eclipse, VS Code.



A debugger: set a breakpoint, run, then pause to inspect variables and step through the code

Exam tips

- List the OS's jobs (memory, process, file, device and security management) — "manages resources" alone is too vague.
- Compare **compiler vs interpreter vs assembler**: what each translates and when errors are reported.
- Explain what an **IDE** provides (editor, debugger, error diagnostics, auto-complete).