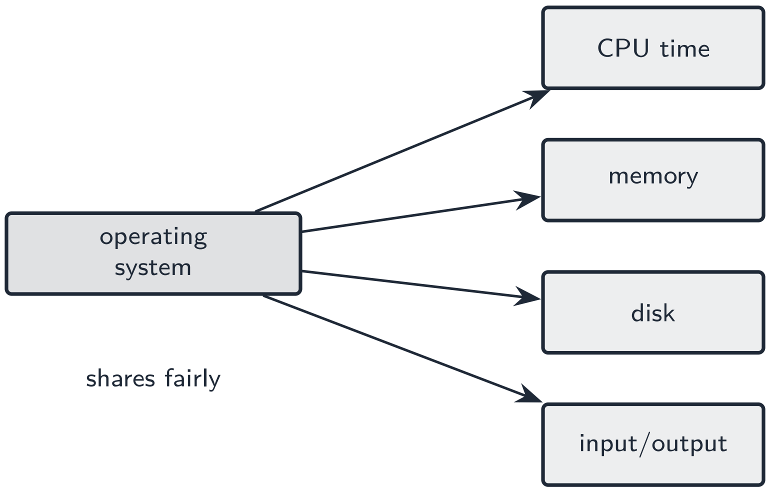
System Software

A-Level Computer Science

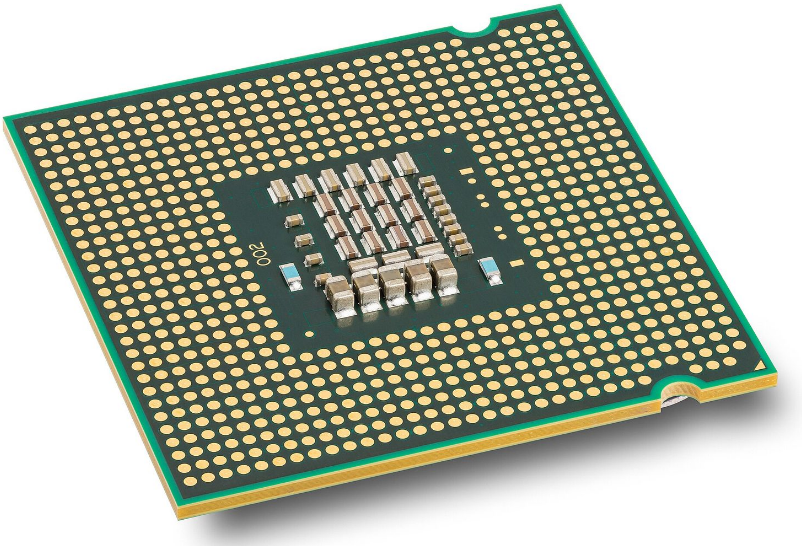
How an OS maximises use of resources

A computer has many resources (CPU time, memory, disk, I/O) and many programs competing for them. The OS **shares them fairly and efficiently** so each is well used and the system stays responsive:



The OS shares the CPU, memory, disk and I/O between programs

- **multi-tasking** 多任务 — switch the CPU quickly between processes so several seem to run at once.
- **memory management** — give each process the memory it needs; use disk **paging** 分页 when **RAM** runs out.
- **spooling** 假脱机 and buffering — print jobs queue on disk so the CPU never waits for the printer.
- **caching** — keep recently-used disk data in **cache** 高速缓存 / RAM.



The processor is a key resource the OS shares between competing tasks

The OS also manages memory (RAM), deciding what to keep in it and what to page out to disk

The user interface

The user interface hides the hardware behind friendly abstractions: the user sees windows, menus and folders, not addresses or sectors. One click on an icon makes the OS find the program on disk, allocate memory, load it and start it. A **CLI** (command line) is powerful and scriptable for experts; a **GUI** (graphical) is easier to learn. Most systems offer both.

Process management

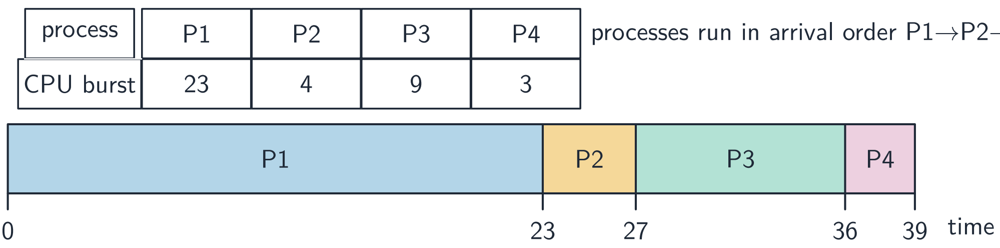
A **process** 进程 is a program in execution — its code, current state, memory and open files.

Scheduling

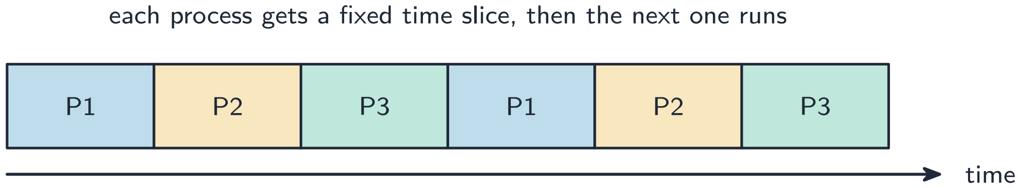
The **scheduler** 调度器 chooses which **ready** process runs next, and for how long:

- **round robin** 轮转 — each process gets a fixed **time slice** 时间片, then goes to the back of the queue.
- first-come-first-served; shortest job first; **shortest remaining time** (run the job with the least work left); priority; multilevel feedback queues.

The trade-off is responsiveness vs throughput vs fairness.



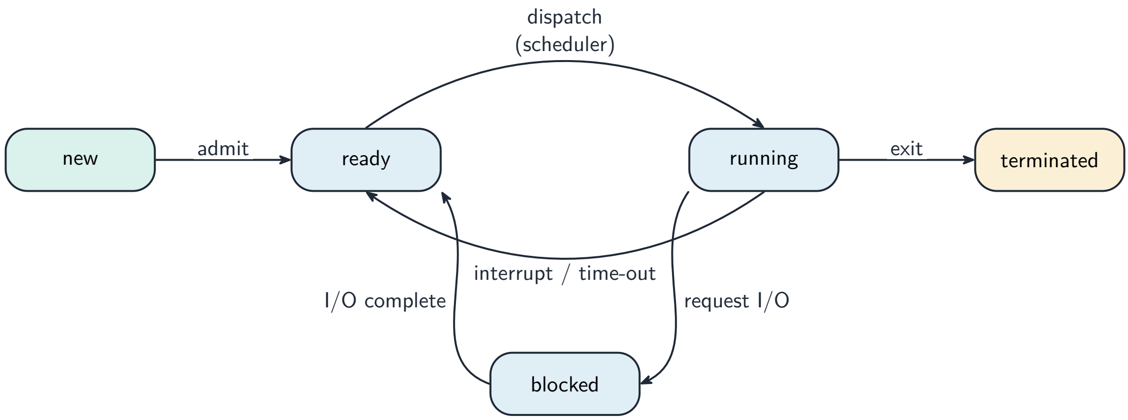
First-come-first-served scheduling of four processes



Round-robin: each process gets a fixed time slice in turn, then the next runs (unlike first-come-first-served)

Process states

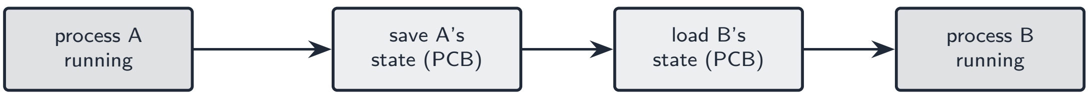
A process is **new**, **ready** (waiting for the CPU), **running**, **blocked** 阻塞 (waiting for I/O or a lock), or **terminated**. When its time slice ends it goes running → ready; when it requests I/O it goes running → blocked; when the I/O finishes it goes blocked → ready.



A process moves between the new, ready, running, blocked and terminated states

Process control block and context switch

For each process the OS keeps a **process control block** 进程控制块 (PCB) — the saved program counter, registers, state and memory info.



A context switch saves one process's state and loads another's

- a **context switch** 上下文切换 suspends one process and starts another: it saves the state into one PCB and restores it from another. This small cost is paid on every switch.
- the **kernel** 内核 (the core of the OS) acts as an **interrupt handler** 中断处理程序. When a device or the timer raises an interrupt, **interrupt handling** 中断处理 saves the running process and runs the right routine — this is what drives low-level scheduling.

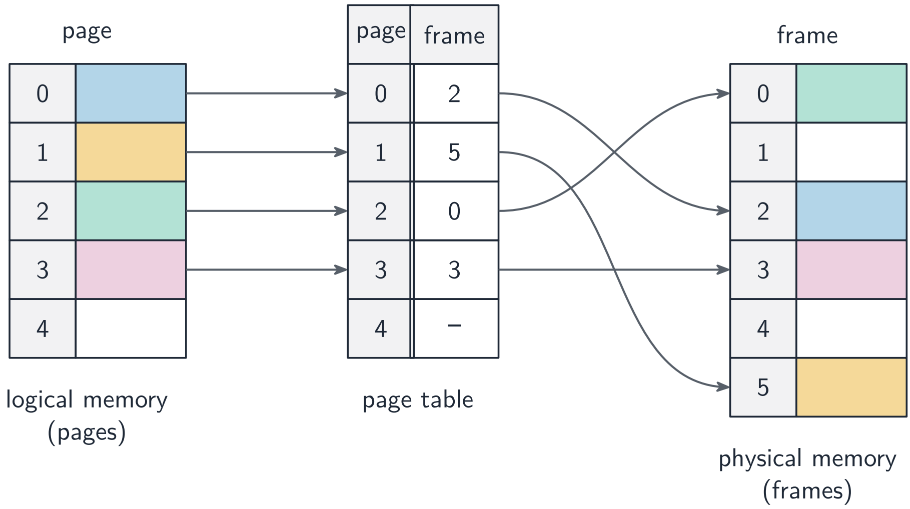
Inter-process communication

Processes are isolated, so the OS provides **inter-process communication 进程间通信**: **pipes 管道** (one program's output feeds another's input), **shared memory 共享内存** (a region several processes can use), and message passing.

Virtual memory, paging, segmentation

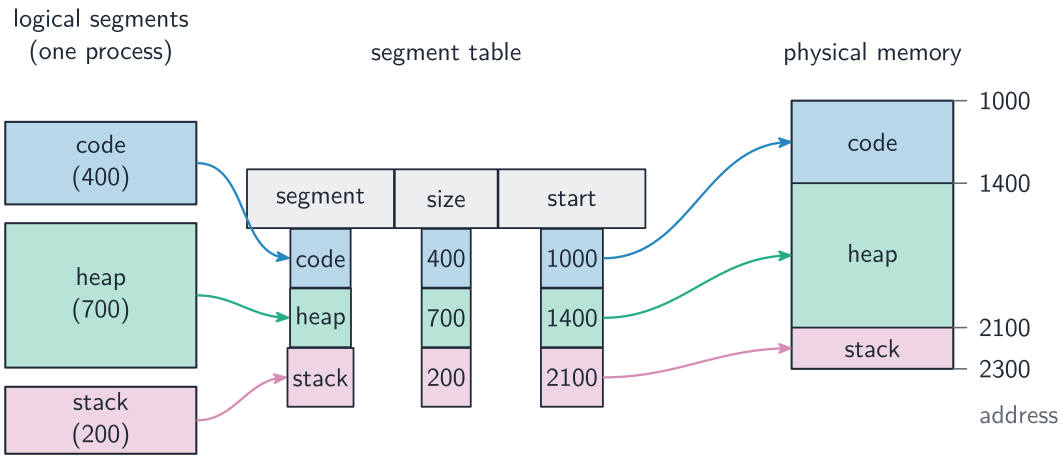
Each process gets its own **virtual address space 虚拟地址空间** — a clean, contiguous range of addresses the OS maps to physical memory. This gives each process a simple space, **protects processes from each other**, and lets the total memory **exceed physical RAM**.

In **paging**, the virtual space is split into fixed-size **pages 页** and physical memory into same-sized **frames 页框**. A page table maps each page to a frame. If an accessed page is not in RAM — a **page fault 缺页** — the OS reads it from the **swap file 交换文件** into a frame, evicting another page if RAM is full. Frequent faults cause **thrashing 抖动** (**disk thrashing**), where the OS spends most of its time swapping pages instead of doing useful work.



Paging maps each page of logical memory to a frame of physical memory

In **segmentation 分段**, memory is split into variable-sized logical segments (code, stack, heap), each with its own permissions. Many systems use paging within segments.



Segmentation maps variable-sized segments using a segment map table

How an interpreter runs a program

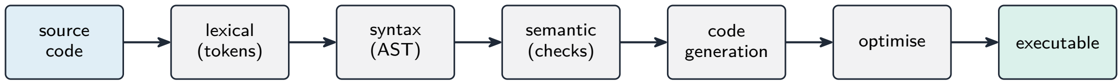
An **interpreter** 解释器 translates and runs the source **at the same time**. For each statement it reads the line, does lexical and syntax analysis, checks types, then **executes** the action, and moves on. Errors are reported immediately and it usually stops; no executable is produced. The translation is redone every run (slower), but it gives fast development feedback and is portable.

Stages of compilation

A **compiler** 编译器 turns source into **machine code** 机器码 in phases:

1. **lexical analysis** 词法分析 — the lexer groups characters into **tokens** 词法单元 (keywords, identifiers, operators, literals), discarding whitespace and comments.
2. **syntax analysis (parsing)** 语法分析 — check the tokens fit the grammar and build an **abstract syntax tree** 抽象语法树. A missing bracket gives a **syntax error** 语法错误.
3. **semantic analysis** 语义分析 — check the program makes sense (variables declared, types match).
4. **code generation** 代码生成 — walk the tree and emit target code, choosing registers and layouts.
5. **code optimisation** 代码优化 — remove redundant work, fold constants, reorder for the pipeline.

The output is an executable.



The phases of compilation, from source code to an optimised executable

Grammar: BNF and syntax diagrams

A **grammar** 文法 says which token sequences are valid programs.

Backus-Naur Form 巴科斯-诺尔范式 (BNF) is textual. A **production rule** 产生式 has the form:

```
<symbol> ::= alternative1 | alternative2 | ...
```

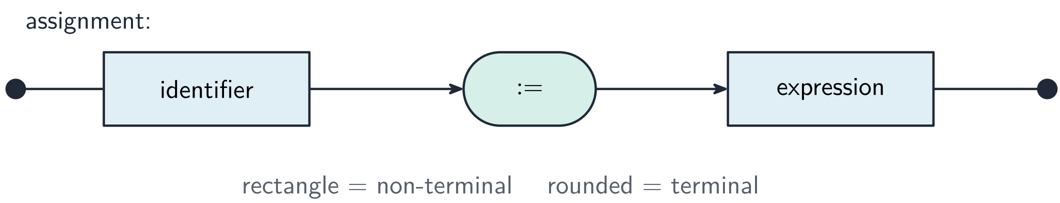
Each alternative is a sequence of **terminal** 终结符 symbols (literal text) and **non-terminal** 非终结符 symbols (other rule names):

```
<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>
```

The recursive third rule expresses ”a letter followed by any number of letters or digits”. An `IF` statement:

```
<if-statement> ::= IF <condition> THEN <statement> ENDIF
                 | IF <condition> THEN <statement> ELSE <statement> ENDIF
```

A **syntax diagram** 语法图 (railroad diagram) shows the same thing graphically: boxes for non-terminals, rounded boxes for terminals, arrows for valid paths, loops for repetition. The two notations are equivalent. The parser uses the grammar to decide whether a program is valid.



A syntax (railroad) diagram for an assignment statement

Reverse Polish Notation (RPN)

In **infix** 中缀 notation the operator sits between its operands (`3 + 4 * 2`), needing brackets and precedence rules. In **Reverse Polish Notation** 逆波兰表示法 (RPN, **postfix** 后缀) the operator follows its operands (`3 4 2 * +`), needing no brackets.

Converting infix to RPN

Use an operator **stack** 栈. Scan left to right: output an operand; for an operator, first pop any stacked operators of **higher or equal precedence** 优先级 to the output, then push it; push `(`; on `)` pop to output until the matching `(`. At the end, pop all operators. Example: `(3 + 4) * 2` → `3 4 + 2 *`.

Evaluating RPN

Use a stack of operands. Scan left to right: push each operand; on an operator, pop the top two, apply it, and push the result. Evaluating `3 4 2 * +`:

Token	**Stack**
`3`	3
`4`	3, 4
`2`	3, 4, 2
`*`	3, 8
`+`	11

Result: 11. RPN needs no brackets at evaluation time and suits a stack machine — which is how the JVM and many **bytecode** 字节码 interpreters work.

Worked example. Convert $(A+B)\times(C-D)$ to RPN, then evaluate $(3+4)\times(5-2)$. Scan left to right using an operator **stack**. Push `(`; output `A`; push `+`; output `B`; on `)` pop back to the matching `(`, giving `A B +` so far. Push `×`, and the second bracket behaves the same way, giving `C D -`. At the end pop the `×`. Result: **`A B + C D - ×`**. To evaluate the numbers, use a stack of **operands**: push 3, push 4; `+` pops both and pushes 7; push 5, push 2; `-` pops both and pushes 3; `×` pops 7 and 3 and pushes **21**. Two things make these reliable: the operands keep their **original order** through the conversion (only the operators move), and every operator acts on the **two values immediately below it** on the stack.

Exam tips

- List the **stages of compilation** (lexical, syntax and semantic analysis, code generation, optimisation) and what each does.
- Explain **virtual memory and paging** (disk used as extra RAM) and its cost (disk thrashing).
- Evaluate **Reverse Polish Notation** with a stack — no brackets are needed.
- Use **BNF or a syntax diagram** to test whether a string is valid.