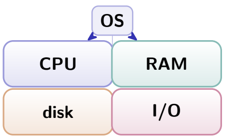
# System Software

A-Level Computer Science ▪ Topic 16

A-Level Computer Science






## What we will cover

1 How an OS shares resources
2 Processes, scheduling, memory
3 Interpreters and compilers
4 BNF, syntax diagrams, RPN
5 Recap

# 1 Sharing resources



## The OS shares four scarce things

Many programs compete for CPU time, memory, disk and I/O. The OS shares them fairly so the machine stays responsive.

**multi-tasking** 多任务
switch the CPU quickly so several programs *seem* to run at once

**paging** 分页
give each process memory; use disk when RAM runs out

**spooling** 假脱机
queue print jobs on disk so the CPU never waits

**caching** 高速缓存
keep recently-used disk data in RAM



## The user interface hides the hardware

The user sees windows, menus and folders – not addresses or sectors. One click on an icon makes the OS find the program, allocate memory, load it and start it.

**CLI**
powerful, scriptable

**GUI**
easier to learn

> Most systems offer both. The interface is an abstraction, not a different OS.

# 2 Processes

## A process is a program in execution

Its code, current state, memory and open files. The **scheduler** 调度器 chooses which ready process runs next, and for how long.

**round robin**
each process gets a fixed **time slice** 时间片, then the back of the queue

**FCFS**
first-come, first-served – simple, can stall behind a long job

**SJF / SRT**
shortest job, or shortest remaining time

**priority**
urgent work first; risk of starvation

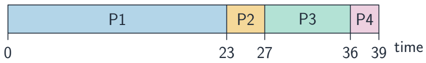


## First come, first served

Simple and fair in arrival order, but one long job delays everything behind it.

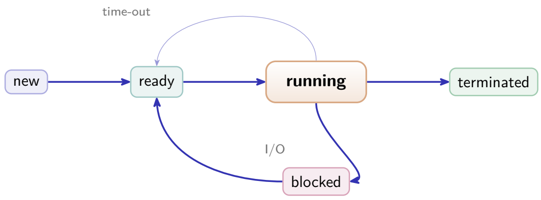
| process | P1 | P2 | P3 | P4 |
|---|---|---|---|---|
| CPU burst | 23 | 4 | 9 | 3 |

processes run in arrival order P1→P2→P3→P4






## Process states

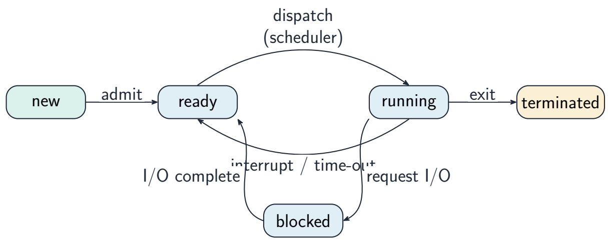



Time slice ends: running → ready. I/O requested: running → blocked. I/O done: blocked → ready.



## The state diagram

Ready, running, blocked — and only the scheduler moves a process between ready and running.






## PCB and context switch

For each process the OS keeps a **process control block** 进程控制块 (PCB): saved program counter, registers, state, memory info. A **context switch** 上下文切换 saves one PCB and loads another. A small cost, paid on every switch.

The **kernel** 内核 is the interrupt handler: a device or the timer raises an interrupt, the running process is saved, and the right routine runs.

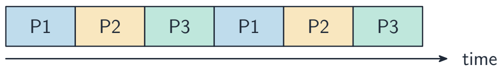
Isolated processes talk via **IPC** 进程间通信: **pipes** 管道, **shared memory** 共享内存, message passing.



## Round robin

Each process gets a fixed **time slice** 时间片; when it expires the scheduler switches to the next.

each process gets a fixed time slice, then the next one runs



## Virtual memory, paging, thrashing

Each process gets its own **virtual address space** 虚拟地址空间, mapped onto physical memory.

- **pages** 页 ↔ **frames** 页框 of the same size
- a page table stores the mapping
- a **page fault** 缺页 loads the page from the **swap file** 交换文件

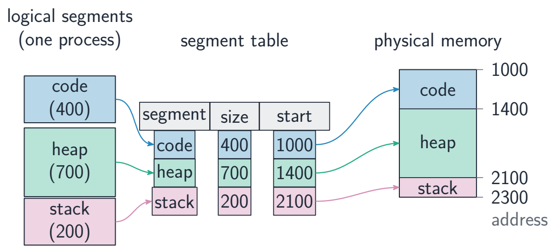
> **Thrashing** 抖动 is when the OS spends most of its time swapping pages instead of running programs – too little RAM for the working set.

**Segmentation** 分段 splits memory into variable-sized logical chunks (code, stack, heap), each with its own permissions.



## Paging and segmentation

Paging splits memory into equal **pages** 页; segmentation splits it into logical, unequal **segments** 段.

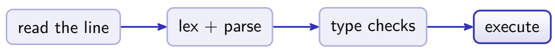



# 3 Translation



## How an interpreter runs a program

An **interpreter** 解释器 translates and runs the source at the same time.

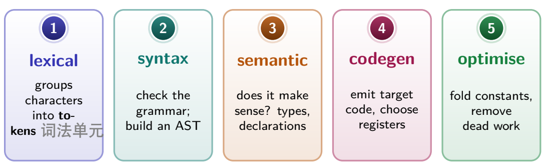



- errors are reported immediately; it usually stops
- no executable is produced – translation is redone every run
- slower, but fast development feedback and portable



## Five stages of compilation

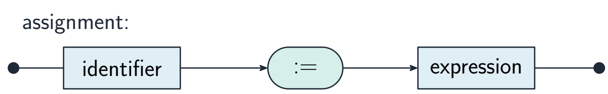
A **compiler** 编译器 turns source into **machine code** 机器码.






## Syntax, as a diagram

A syntax diagram says the same as BNF: follow any path through it and you get a valid construct.




rectangle = non-terminal    rounded = terminal

# 4 Grammar and RPN



## BNF – a grammar in text

**Backus-Naur Form** 巴科斯-诺尔范式 writes a **grammar** 文法 as production rules: `<digit> ::= 0 | 1 | …| 9`

`<identifier> ::= <letter>`
`| <identifier> <letter>`
`| <identifier> <digit>` < > are **non-terminals** 非终结符; literals are **terminals** 终结符. Recursion expresses "any number of".

A **syntax diagram** 语法图 (railroad diagram) shows the same grammar graphically. The parser uses either to decide whether a program is valid.



## Reverse Polish Notation needs no brackets

Infix: operator between operands, so you need brackets and precedence.
**RPN** 逆波兰表示法 (postfix): operator follows its operands.

$(3+4)\times 2 \longrightarrow \boxed{3\ 4\ +\ 2\ \times}$

Scan left to right with an operator **stack** 栈: output operands; pop higher-or-equal precedence operators before pushing; parentheses flush the stack.



## Worked example – evaluate RPN

**Evaluate** $3\ 4\ 2\ *\ +$

Push each operand; on an operator, pop two, apply, push the result.

| Token | Stack |
|---|---|
| 3 | 3 |
| 4 | 3, 4 |
| 2 | 3, 4, 2 |
| * | 3, 8 |
| + | 11 |

$(A+B)\times(C-D)$

Convert: `A B + C D - ×`

Evaluate $(3+4)\times(5-2)$: push 3, 4; $+\to 7$; push 5, 2; $-\to 3$; $\times\to 21$.

# 5 Recap



## Topic 16 – key takeaways

1. **OS** — multi-task, page, spool, cache; PCB / context switch
2. **Memory** — virtual addresses, pages, faults, thrashing
3. **Compile** — lex, parse, semantic, generate, optimise
4. **RPN** — postfix; evaluate with an operand stack

## Check yourself

1. Round-robin: what happens when a time slice ends?
2. What is thrashing?
3. Name the five compilation stages in order.
4. Evaluate $5\ 1\ 2\ +\ 4\ \times\ +$

**Answers** **1.** it goes to the back of the ready queue **2.** constant paging, little useful work **3.** lex, syntax, semantic, codegen, optimise **4.** $5 + (1 + 2) \times 4 = 17$