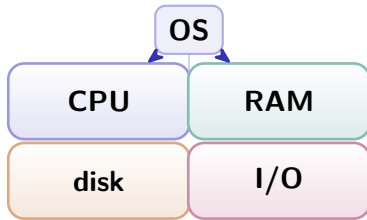


System Software

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



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

switch the CPU quickly so several programs *seem* to run at once

paging

give each process memory; use disk when RAM runs out

spooling

queue print jobs on disk so the CPU never waits

caching

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.



A process is a program in execution

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

round robin

each process gets a fixed **time slice**, then the back of the queue

FCFS

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

SJF / SRT

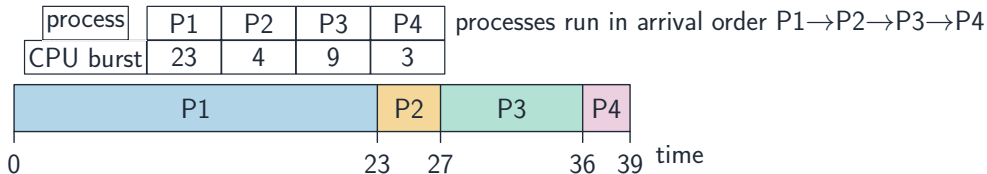
shortest job, or shortest remaining time

priority

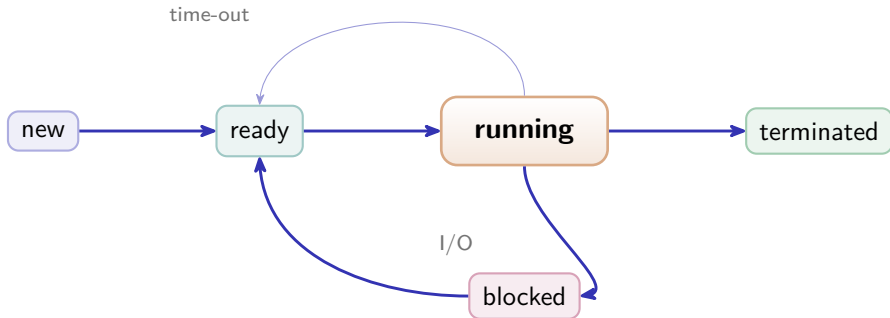
urgent work first; risk of starvation

First come, first served

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



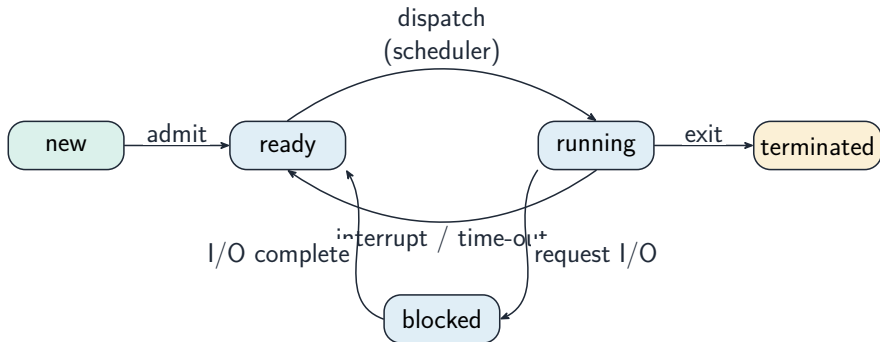
Process states



Time slice ends: running $\rightarrow$ ready. I/O requested: running $\rightarrow$ blocked. I/O done: blocked $\rightarrow$ 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** (PCB): saved program counter, registers, state, memory info. A **context switch** saves one PCB and loads another. A small cost, paid on every switch.

The **kernel** 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** : **pipes** , **shared memory** , message passing.

Process control block and context switch

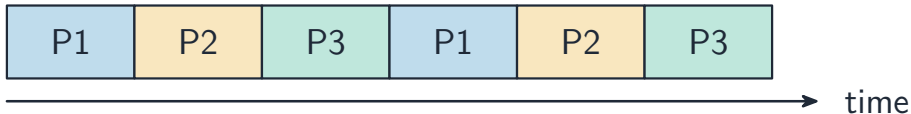


A context switch saves one process's state and loads another's

Round robin

Each process gets a fixed **time slice** ; 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

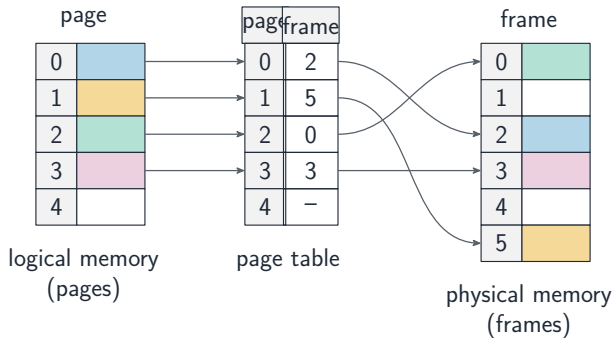
Each process gets its own **virtual address space**, mapped onto physical memory.

- **pages** ↔ **frames** of the same size
- a page table stores the mapping
- a **page fault** loads the page from the **swap file**

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

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

Virtual memory, paging, segmentation



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

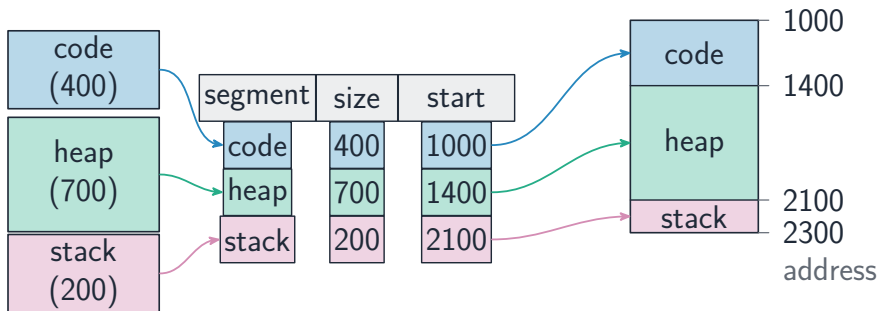
Paging and segmentation

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

logical segments
(one process)

segment table

physical memory





How an interpreter runs a program

An **interpreter** 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

Two more jobs the OS does, and how a compiler reads code

Interrupt handling

Interrupt handling : a device raises a signal, the CPU saves state, runs the handler, and resumes – so no polling loop is needed.

Inter-process communication

Inter-process communication : pipes, shared memory and message queues let separate processes exchange data under the OS's control.

Lexical analysis

Lexical analysis splits the source into tokens and strips comments and whitespace.

Syntax analysis

Syntax analysis (parsing) builds an **abstract syntax tree** from those tokens – and reports a **syntax error** when they fit no rule.

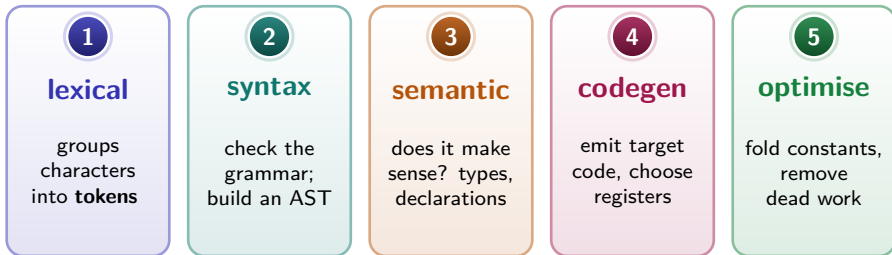
Then

Code optimisation improves the intermediate form, and code generation emits machine code or **bytecode** .

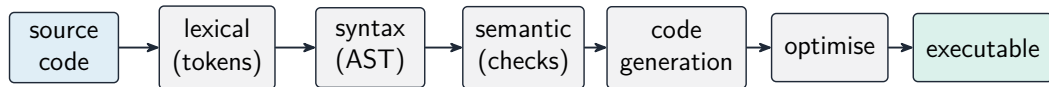
A syntax error is caught before anything runs; a logic error is not caught at all. The compiler can only check the grammar.

Five stages of compilation

A **compiler** turns source into **machine code** .



Five stages of compilation, in detail



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

Syntax, as a diagram

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

assignment:



rectangle = non-terminal rounded = terminal



BNF – a grammar in text

Backus-Naur Form - writes a **grammar** as
production rules: `<digit> ::= 0 | 1 | ... | 9`

`<identifier> ::= <letter>
 | <identifier> <letter>
 | <identifier> <digit> < >` are

non-terminals ; literals are **terminals** .
Recursion expresses “any number of”.

A **syntax diagram** (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** (postfix): operator *follows* its operands.



The diagram illustrates the conversion of an infix expression to postfix notation. On the left, a light blue rounded rectangle contains the infix expression $(3 + 4) \times 2$. A blue arrow points from this box to another light blue rounded rectangle on the right, which contains the postfix expression $3\ 4\ +\ 2\ \times$.

Scan left to right with an operator **stack** : 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.

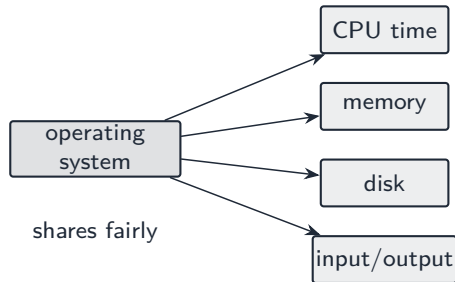
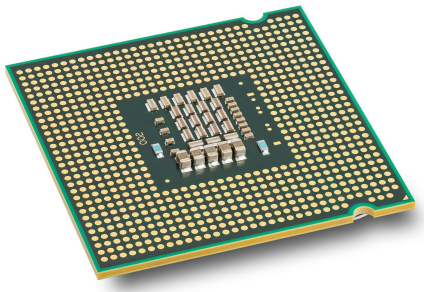
Token	Stack
3	3
4	3, 4
2	3, 4, 2
*	3, 8
+	11

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

Convert: A B + C D - ×

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

How an OS maximises use of resources



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

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

What a scheduler is trading off

responsiveness

how quickly an interactive process gets a turn

throughput

how many processes complete per unit time

fairness

every process getting a share, and none starving

the trade-off

you cannot maximise all three – and each algorithm picks a different corner

first come, first served

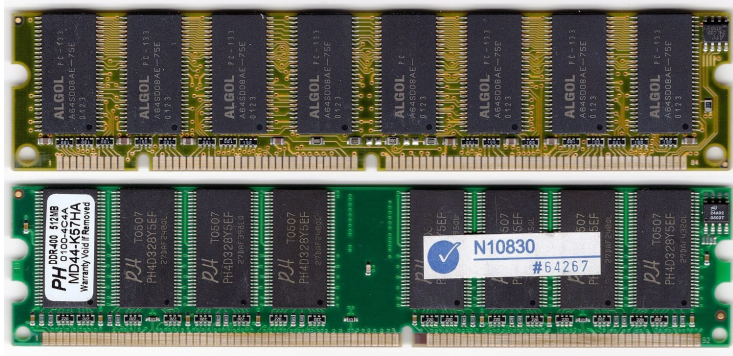
maximum fairness by arrival, and terrible responsiveness behind a long job

round robin

good responsiveness; a short time slice costs throughput in context switching

Naming the trade-off is what turns a list of algorithms into an answer. Say which of the three a given algorithm sacrifices.

The OS also manages memory (RAM)



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

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.



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 $5 + 1 + 2 + 4 \times 3$

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