
Self-Learning Material (SLM)
MCA 104 Operating System and Information
Technology



Madhya Pradesh Bhoj (Open) University

(Established Under an Act of State Assembly in 1991)

मध्य प्रदेश भोज (मुक्त) विश्वविद्यालय

TABLE OF CONTENTS

Chapter 1 — Operating Systems: Foundations, Structure, and Process Scheduling

1.1 Operating Systems: Evolution and Generations	1
1.2 Types of Operating Systems	4
1.3 Different Views of the Operating System	8
1.4 Operating System Concepts and Structure	10
1.5 The Process Concept	13
1.6 The Process Control Block	15
1.7 Operating System Services for Process Management	17
1.8 Process Scheduling Algorithms.....	19
1.9 Performance Evaluation of Scheduling Algorithms	27
1.10 Summary	29
1.11 Glossary	30

Chapter 2 — Memory Management

2.1 Memory Management Without Swapping or Paging.....	33
2.2 Concepts of Swapping and Paging	37
2.3 Page Replacement Algorithms	41
2.4 Modelling Paging Algorithms.....	48
2.5 Design Issues for Paging Systems	50
2.6 Segmentation	54
2.7 Segmentation with Paging (Segmented Paging and Paged Segmentation)	57
2.8 Summary	59
2.9 Glossary	60

Chapter 3 — Inter-Process Communication, Synchronization, and Deadlocks

3.1 The Need for Inter-Process Synchronization	63
3.2 Mutual Exclusion and Hardware Support.....	66
3.3 Semaphores	71
3.4 Critical Sections, Critical Regions, and Monitors.....	75
3.5 Classical Problems in Concurrent Programming	79
3.6 Deadlocks: Concepts and Characterization	86
3.7 Deadlock Prevention and Avoidance	89
3.8 The Banker's Algorithm	92
3.9 Deadlock Detection and Recovery	96
3.10 Summary	99
3.11 Glossary	100

Chapter 4 — File Systems and Input/Output Management

4.1 File Systems and Directories.....	103
4.2 File System Implementation and Protection	108
4.3 Principles of I/O Hardware	114
4.4 Principles of I/O Software.....	116

4.5 Disks and Disk Scheduling	120
4.6 Clocks	125
4.7 Summary	127
4.8 Glossary	128

Chapter 5 — Information Security, Mobile Commerce, and Cloud Computing

5.1 Introduction to Information Security	131
5.2 Malicious Programs	132
5.3 Cryptography	134
5.4 Digital Signatures	138
5.5 Firewalls.....	141
5.6 User Identification and Authentication.....	142
5.7 Security Awareness and Policies.....	144
5.8 Application Areas Requiring Information Security.....	146
5.9 Introduction to Mobile Commerce.....	146
5.10 Growth of Mobile Commerce.....	147
5.11 Success Stories of Mobile Commerce	148
5.12 Technologies for Mobile Commerce	149
5.13 M-Commerce in India.....	151
5.14 Digital Marketing	152
5.15 History of Centralized and Distributed Computing.....	154
5.16 Overview of Distributed Computing.....	155
5.17 Cluster Computing.....	157
5.18 Grid Computing.....	159
5.19 Introduction to Cloud Computing.....	161
5.20 Cloud Issues and Challenges.....	163
5.21 Properties and Characteristics of Cloud Computing.....	164
5.22 Cloud Service Models	165
5.23 Cloud Deployment Models	167
5.24 Cloud Resources: Network and API	170
5.25 Virtual and Physical Computational Resources	171
5.26 Data Storage in the Cloud	173
5.27 Summary	174
5.28 Glossary	175

Chapter 6 — Internet of Things, Virtual Reality, and Artificial Intelligence

6.1 Introduction to the Internet of Things	179
6.2 Characteristics of IoT	182
6.3 Physical Design of IoT.....	183
6.4 Logical Design of IoT	188
6.5 Functional Blocks of IoT	193
6.6 Home Automation	195
6.7 Industrial IoT and Industry Applications.....	198
6.8 Surveillance Applications of IoT	202

6.9 Other IoT Application Domains	205
6.10 Introduction to Virtual Reality	207
6.11 Smart Systems.....	212
6.12 Embedded Systems	215
6.13 Concept of Artificial Intelligence.....	221
6.14 Machine Learning	223
6.15 Artificial Neural Networks	227
6.16 Robotics.....	229
6.17 Natural Language Processing.....	232
6.18 Expert Systems	234
6.19 Fuzzy Logic Systems	236
6.20 General Applications of Artificial Intelligence.....	238
6.21 Summary	240
6.22 Glossary	241

LIST OF FIGURES

Figure 1.1	Three kernel structures: monolithic, layered, and microkernel	11
Figure 1.2	Process state-transition diagram	14
Figure 1.3	Structure of a Process Control Block	16
Figure 1.4	Multilevel feedback queue structure	25
Figure 2.1	Fixed versus variable partitioning.....	36
Figure 2.2	Address translation via a page table and TLB.....	40
Figure 2.3	The Clock (second-chance) algorithm.....	45
Figure 2.4	Address translation via a segment table	56
Figure 2.5	Segmentation with paging: two-level translation	58
Figure 3.1	Interleaved execution producing a lost update	65
Figure 3.2	Structure of a monitor	77
Figure 3.3	A bounded circular buffer (producer-consumer).....	81
Figure 3.4	The Dining Philosophers seating arrangement.....	83
Figure 3.5	A resource-allocation graph containing a cycle.....	88
Figure 4.1	Absolute and relative paths in a directory tree	107
Figure 4.2	Contiguous, linked, and indexed file allocation	110
Figure 4.3	CPU, memory, and a controller sharing a system bus.....	115
Figure 4.4	The layered structure of I/O software	117
Figure 4.5	Disk geometry: platters, tracks, sectors, and a cylinder.....	120
Figure 5.1	Symmetric-key encryption and decryption	135
Figure 5.2	Public-key (asymmetric) encryption and decryption.....	137
Figure 5.3	Creating and verifying a digital signature.....	139
Figure 5.4	A firewall between a trusted network and the internet.....	143
Figure 5.5	Cluster computing architecture	158
Figure 5.6	Roles in a grid computing network	160
Figure 5.7	Cloud computing architecture.....	162
Figure 5.8	The cloud service model stack	166
Figure 5.9	Public, private, hybrid, and community cloud deployment models	168
Figure 6.1	The conceptual pipeline of an IoT system	178
Figure 6.2	Generic block diagram of an IoT device	182
Figure 6.3	The IoT protocol stack compared with the TCP/IP reference model	184
Figure 6.4	The four IoT communication models	187
Figure 6.5	The six IoT deployment levels.....	190
Figure 6.6	The six functional blocks of an IoT system.....	191
Figure 6.7	Architecture of a home automation system.....	194
Figure 6.8	The edge-fog-cloud architecture of an IIoT deployment.....	197
Figure 6.9	Architecture of a VR system: the tracking-rendering-display loop.....	207
Figure 6.10	The sense-think-act loop, with feedback	211
Figure 6.11	Block diagram of an embedded system	214
Figure 6.12	How embedded systems, IoT, smart systems and VR interlock.....	218

Figure 6.13	A decision tree for predicting fitness from age and exercise frequency.....	223
Figure 6.14	Structure of a simple feedforward artificial neural network	225
Figure 6.15	Core components of a robot and how they interact.....	228
Figure 6.16	The five levels of NLP processing.....	231
Figure 6.17	Architecture of an expert system	233
Figure 6.18	Architecture of a fuzzy logic system.....	235

LIST OF TABLES

Table 1.1	OS capability by hardware generation	3
Table 1.2	OS categories: what each switch between.....	7
Table 1.3	Four perspectives on an operating system	9
Table 1.4	Four kernel structuring approaches.....	12
Table 1.5	Key fields stored in a Process Control Block	16
Table 1.6	Representative system calls for process management	18
Table 1.7	FCFS worked example	20
Table 1.8	SJF (non-preemptive) worked example	21
Table 1.9	SRTN worked example (same input as Table 1.8)	22
Table 1.10	Round Robin worked example, quantum = 2	23
Table 1.11	HRRN worked example	24
Table 1.12	MLFQ worked example	26
Table 1.13	Standard criteria for evaluating CPU scheduling	27
Table 1.14	Trade-offs among the five scheduling algorithms	28
Table 1.15	Matching workload type to a scheduling approach	29
Table 2.1	Internal fragmentation under 8 KB fixed partitions	35
Table 2.2	External fragmentation after Process B terminates	35
Table 2.3	Page table for the address-translation example	39
Table 2.4	FIFO trace over the reference string (3 frames)	42
Table 2.5	Optimal replacement trace (3 frames)	42
Table 2.6	LRU trace (3 frames).....	43
Table 2.7	MRU trace (3 frames)	44
Table 2.8	Clock (second-chance) trace (3 frames)	45
Table 2.9	Belady's anomaly: FIFO faults at 3 versus 4 frames	46
Table 2.10	Fault-count comparison across five algorithms (3 frames).....	47
Table 2.11	Distance string and fault counts for 1-5 frames	49
Table 2.12	Single-level, multilevel, and inverted page tables compared	51
Table 2.13	Segment table for the worked example	55
Table 2.14	Advantages and disadvantages of segmentation	56
Table 2.15	Segment table for segmented paging.....	58
Table 2.16	Page table for Segment 1 (data segment)	58
Table 3.1	A race condition on a shared counter	64
Table 3.2	Mutual exclusion approaches compared	70
Table 3.3	Busy-waiting versus blocking semaphore implementation.....	73
Table 3.4	Message-passing variants compared.....	79
Table 3.5	The four classical synchronization problems compared.....	86
Table 3.6	Deadlock prevention strategies and their cost.....	90
Table 3.7	Banker's Algorithm worked example (time T0)	94
Table 3.8	Deadlock detection worked example	97
Table 4.1	File attributes maintained by the operating system	104

Table 4.2	File access methods compared	106
Table 4.3	Allocation methods compared	110
Table 4.4	Free-space management techniques compared	111
Table 4.5	Access-control lists versus permission bits.....	112
Table 4.6	Block devices versus character devices	114
Table 4.7	I/O software layers and their responsibilities.....	118
Table 4.8	Head movement of six disk-scheduling algorithms.....	123
Table 4.9	Memory-mapped terminals versus serial-line terminals.....	127
Table 5.1	The CIA triad	131
Table 5.2	Types of malicious programs	133
Table 5.3	Documented malware incidents	133
Table 5.4	Symmetric-key algorithms compared.....	136
Table 5.5	Public-key algorithms compared	137
Table 5.6	Advantages and limitations of cryptography	138
Table 5.7	What a digital signature guarantees	140
Table 5.8	Applications of digital signatures	140
Table 5.9	Scope of firewall protection	142
Table 5.10	The three authentication factors	143
Table 5.11	Common security awareness topics.....	144
Table 5.12	Domains where information security is critical	146
Table 5.13	Key features of m-commerce platforms	147
Table 5.14	Drivers of m-commerce growth, by layer	148
Table 5.15	Selected Indian m-commerce success stories	149
Table 5.16	Mobile payment technologies.....	150
Table 5.17	Benefits of m-commerce adoption in India	152
Table 5.18	Core digital marketing channels.....	153
Table 5.19	Characteristics of centralized systems.....	154
Table 5.20	From centralization to decentralization: a timeline	154
Table 5.21	Characteristics of distributed systems.....	156
Table 5.22	Advantages and limitations of distributed systems.....	157
Table 5.23	Types of clusters, by purpose	158
Table 5.24	Advantages and disadvantages of cluster computing	159
Table 5.25	Advantages and disadvantages of grid computing.....	160
Table 5.26	Cloud computing: issues and challenges	163
Table 5.27	General properties of cloud computing.....	164
Table 5.28	The five essential characteristics of cloud computing (NIST)	165
Table 5.29	Infrastructure as a Service (IaaS).....	166
Table 5.30	Platform as a Service (PaaS).....	166
Table 5.31	Software as a Service (SaaS)	167
Table 5.32	Comparing IaaS, PaaS, SaaS, and FaaS.....	167
Table 5.33	Comparing cloud deployment models	169
Table 5.34	Factors in choosing a cloud deployment model	169

Table 5.35	Core components of cloud networking.....	170
Table 5.36	Benefits of cloud networking.....	171
Table 5.37	Virtual machines compared with containers.....	172
Table 5.38	Types of cloud storage.....	173
Table 5.39	Requirements of cloud storage	173
Table 5.40	Benefits and challenges of cloud storage	173
Table 6.1	IoT distinguished from related concepts.....	179
Table 6.2	Characteristics of IoT and the design consequence each produces	181
Table 6.3	Sensor performance parameters	183
Table 6.4	Link-layer technologies and their operating envelopes	184
Table 6.5	Comparison of IoT application-layer protocols	186
Table 6.6	Comparison of the four IoT communication models.....	188
Table 6.7	REST methods and their properties	189
Table 6.8	REST-based versus WebSocket-based communication APIs.....	190
Table 6.9	The six IoT levels.....	190
Table 6.10	Comparison of home automation communication protocols.....	195
Table 6.11	Consumer IoT compared with Industrial IoT.....	196
Table 6.12	Comparison of maintenance strategies	298
Table 6.13	Industrial IoT applications by sector.....	299
Table 6.14	Traditional CCTV compared with IoT-based surveillance	200
Table 6.15	Summary of principal IoT application domains	204
Table 6.16	Virtual, Augmented, Mixed and Extended Reality distinguished.....	205
Table 6.17	Key performance parameters of a VR system.....	208
Table 6.18	Embedded systems, IoT systems and smart systems compared.....	212
Table 6.19	Classification of embedded systems by functional requirement.....	213
Table 6.20	Microprocessor compared with microcontroller	214
Table 6.21	Embedded software architectures compared.....	215
Table 6.22	Representative embedded platforms	217
Table 6.23	Disciplines contributing to Artificial Intelligence.....	219
Table 6.24	Types of machine learning compared	222
Table 6.25	Eight widely used machine learning algorithms.....	222
Table 6.26	Advantages and limitations of machine learning.....	224
Table 6.27	Representative applications of machine learning	224
Table 6.28	Advantages and limitations of artificial neural networks	227
Table 6.29	Representative applications of artificial neural networks.....	227
Table 6.30	Robots compared with conventional AI programs	228
Table 6.31	Advantages and limitations of robotics	229
Table 6.32	Representative applications of robotics.....	229
Table 6.33	Core NLP terminology	231
Table 6.34	Advantages and limitations of natural language processing.....	232
Table 6.35	Representative applications of natural language processing	232
Table 6.36	Advantages and limitations of expert systems	234

Table 6.37	Representative applications of expert systems	234
Table 6.38	Rule base for the fuzzy air-conditioning example	235
Table 6.39	Advantages and limitations of fuzzy logic systems	236
Table 6.40	Representative applications of fuzzy logic	236
Table 6.41	General applications of Artificial Intelligence across domains.....	237

How to Study This SLM

This book covers six chapters that move outward in stages. Chapters 1 to 4 build a complete picture of an operating system: what it is, how it structures itself internally, how it manages running processes, how it manages memory, how it lets processes coordinate safely, and how it manages files and the devices attached to a computer. Chapter 5 turns to information security, mobile commerce, and cloud computing: the concerns that arise once a system is connected to a network and to other people. Chapter 6 looks outward again, to the Internet of Things, virtual reality, and artificial intelligence: fields where computing meets the physical world and, increasingly, tries to reason on its own.

The chapters build on one another. Chapter 2's paging and segmentation assume the process concept introduced in Chapter 1. Chapter 3's synchronization problems assume the scheduling and process-state ideas from Chapter 1. Chapter 4's file and I/O management draw on the process and memory concepts of the three chapters before it. Chapter 5 and Chapter 6 are more self-contained, but still assume the general comfort with computer systems that Chapters 1 to 4 build. It is best to study the chapters in order the first time through, even though later revision can jump straight to whichever chapter an exam or assignment calls for.

Within each chapter, work through every **Check Your Progress** exercise before moving on. These are short diagnostic checkpoints placed right after the material they test, not optional extras, and answering them honestly is the closest a self-learner can get to the feedback a classroom would normally provide. Each chapter also closes with a **Summary** that recaps its main ideas in a few paragraphs and a **Glossary** that collects its key terms in one place for quick revision. A combined **Index** at the end of the book lists important terms with the page on which each is introduced, and a **References** section lists the standard textbooks this material draws on, useful if a topic needs a second, deeper explanation.

Because this is self-learning material, written to be studied without a teacher present, definitions are given before a term is used, technical claims are backed by a worked example wherever one helps, and each chapter tries to explain not just *what* a mechanism is but *why* it exists and *what problem it solves*. Reading a chapter once, working its Check Your Progress boxes, and then re-reading its Summary a day or two later is a more effective way to retain this material than reading it once carefully and moving on.

Terminology and Notation

Certain terms are used precisely and consistently throughout this book, even where everyday usage is looser. For example, **process** and **program** are distinct (a program is a static set of instructions; a process is a program in execution), and **concurrency** and **parallelism** are distinct (concurrency means multiple tasks make progress over overlapping time periods; parallelism means multiple tasks execute at literally the same instant on different processors). Each such term is defined at first use. Algorithms are presented first in plain-language step form and, where standard, in pseudocode; numeric examples use a consistent notation for time (milliseconds, denoted ms) and memory addresses (given in decimal unless hexadecimal is explicitly noted with a 0x prefix).

CHAPTER 1

Operating Systems: Foundations, Structure, and Process Scheduling

1.1 OPERATING SYSTEMS: EVOLUTION AND GENERATIONS

An **operating system (OS)** is the program that sits between a computer's user (or a user's application programs) and the raw hardware, so that neither the user nor the application programmer needs to deal directly with the electrical and mechanical details of the machine. It is, in the most literal sense, an intermediary: application programs ask the operating system for services such as reading a file, allocating memory, printing a page, or waiting for a network packet, and the operating system translates these requests into the specific sequence of hardware operations needed to satisfy them.

An operating system pursues three goals simultaneously. First, it must make the computer **convenient to use**: a user should be able to run a word processor or a browser without knowing anything about disk sector layouts or interrupt vectors. Second, it must use the underlying hardware **efficiently**: CPU cycles, memory, and I/O bandwidth are finite and expensive resources, and a good operating system keeps them busy doing useful work rather than sitting idle. Third, it must allow the system to **evolve**: new hardware, new application demands, and new usage patterns must be accommodated without forcing every existing program to be rewritten.

To meet these goals, an operating system plays three closely related roles at once:

- **Resource allocator.** Many programs may compete for the CPU, memory, disk space, and I/O devices at the same time; the OS decides who gets what, when, and for how long.
- **Control program.** The OS supervises the execution of user programs and the operation of I/O devices, preventing one program's error (or malice) from corrupting another program or the system itself.
- **Kernel.** The OS is, in a strict sense, the one program that is always running on the machine. Every other piece of software, including the shell or desktop environment the user interacts with, is itself just another program that the kernel schedules and supervises.

IMPORTANT — Definition — Operating System

An **operating system** is system software that manages computer hardware and software resources and provides common services for application programs, acting as an intermediary between the user (and user programs) and the computer hardware, with the twin objectives of convenience for the user and efficient use of the hardware.

The capabilities an operating system could offer at any point in history were tightly constrained by the hardware of the day. Studying the evolution of operating systems generation by generation, alongside the corresponding hardware generation, makes clear *why* operating systems look the way they do today: almost every major OS concept, from batch processing and multiprogramming to time-sharing, virtual memory, and graphical interfaces, was invented specifically to overcome a limitation of the hardware available at the time it was invented.

1.1.1 First Generation (1940s – Early 1950s): Vacuum Tubes, No Operating System

The earliest electronic computers (such as ENIAC) used vacuum tubes for switching circuitry and, later, magnetic drums for memory. These machines were enormous, extremely expensive, consumed huge amounts of power, and were individually programmed by hand. There was no operating system in any recognisable sense. A single user (or a small team) had exclusive access to the entire machine for a scheduled block of time, and programs were entered directly in machine

language, sometimes literally by rewiring plugboards or by feeding in a stack of punched cards, with no software layer mediating between the programmer and the hardware. Input and output were handled manually, and a large fraction of the machine's expensive time was wasted on setup, loading, and manual switching between jobs, since the CPU sat completely idle whenever a human was configuring the next run.

1.1.2 Second Generation (Mid-1950s – Early 1960s): Transistors and Batch Processing

The invention of the **transistor** replaced bulky, unreliable, heat-generating vacuum tubes, producing computers that were smaller, more reliable, and considerably cheaper to run. Magnetic core memory became the standard form of main memory. These hardware gains made it worthwhile to invest in software that reduced the wasted, manual setup time of the first generation, and this is exactly what the first true operating systems did.

The key software innovation of this generation was **batch processing**. Jobs (programs together with their data and control instructions) were collected onto magnetic tape by an operator, and a resident "monitor" program (the ancestor of the modern operating system) read, loaded, and ran them one after another automatically, with no human intervention between jobs. This removed the costly human-speed gap between one job finishing and the next one starting. IBM's early batch monitors and systems such as the General Motors/North American Aviation input/output system (GM-NAA I/O) for the IBM 704 are representative of this era. The obvious limitation was that a job still had exclusive use of the CPU from start to finish: if it was waiting on a slow card reader or tape drive, the (very expensive) CPU sat idle waiting for it, exactly as before, only now the queue of waiting jobs was managed automatically instead of manually.

1.1.3 Third Generation (Mid-1960s – Early 1970s): Integrated Circuits, Multiprogramming, and Time-Sharing

Integrated circuits (ICs) replaced individual transistors, packing many switching elements onto a single chip. This further shrank machines, cut costs, and, crucially, made it economical to build machines with enough memory to hold *several* programs at once. This single hardware advance made possible the two most important OS concepts of the era.

Multiprogramming exploited the newly abundant memory by keeping multiple jobs resident in memory simultaneously. Whenever the job currently using the CPU had to wait for an I/O operation to complete, the operating system switched the CPU to a different resident job instead of leaving it idle. This directly attacked the second generation's central weakness: the CPU sitting idle during I/O waits. It dramatically raised CPU utilisation and system throughput.

Time-sharing took the same underlying idea (overlap the CPU time of several programs) and applied it to interactive users rather than batch jobs. By rapidly switching the CPU between many users' programs in tiny slices, a time-sharing system gave each of many simultaneously connected users (typically at terminals) the illusion that they had the machine to themselves, with acceptably fast response to their commands. IBM's OS/360 (an ambitious, if famously complex, attempt at a single operating system for an entire family of mainframes) and **Multics** (Multiplexed Information and Computing Service, a joint MIT/Bell Labs/General Electric project) are the landmark systems of this generation; Multics in particular pioneered ideas that would later shape UNIX, including hierarchical file systems, dynamic linking, and fine-grained protection.

1.1.4 Fourth Generation (1970s – Circa 2000s): Microprocessors, Personal Computing, and Graphical Interfaces

Large-Scale Integration (LSI) and later **Very Large-Scale Integration (VLSI)** technology packed thousands, then millions, of transistors onto a single chip, giving rise to the **microprocessor**, an entire CPU on one chip. This made computers cheap enough, for the first time, to be owned by a single individual rather than shared by an organisation, and the operating systems of this generation evolved in step with that shift.

- **1970s: UNIX** was developed at Bell Labs, introducing (and popularising) a hierarchical file system, built-in multitasking, and multi-user capability in a design that was, unusually for the time, portable across different hardware because it was written mostly in the C programming language rather than assembly.
- **1980s:** The arrival of the IBM PC brought **DOS (Disk Operating System)** to a mass market of single-user personal machines with no memory protection between programs, followed by the early releases of **Microsoft Windows** as a graphical shell running on top of DOS.
- **1990s: Windows 95** brought a genuinely graphical user interface (GUI), pre-emptive multitasking, and plug-and-play hardware support to the mainstream consumer market, while **Linux**, a free, open-source, UNIX-like kernel, emerged as a serious alternative, especially for servers and for programmers who wanted to study and modify the source code themselves.
- **2000s:** Windows continued to dominate the desktop, Linux became the operating system of choice for a large share of internet servers, and **Mac OS X** (built on a UNIX-derived core, later renamed macOS) gave Apple's desktop line a modern, protected-memory, multitasking foundation.
- **2010s: iOS** and **Android** made mobile operating systems, purpose-built for touch interaction and battery-constrained hardware, into the most widely used operating systems on Earth by device count, while the rise of large-scale internet services pushed innovation toward distributed and cloud-oriented operating environments running across thousands of networked machines rather than on any single desktop.

1.1.5 Fifth Generation and Emerging Trends: Distributed, Mobile, and Intelligent Operating Systems

The generation of operating systems now emerging is shaped less by a single new circuit technology and more by new *deployment contexts*: continued miniaturisation down to the nanoscale, the early stages of practical quantum computing, and computing that is distributed across the network by default rather than as an afterthought. Present and emerging operating system work places heavy emphasis on **distributed and cloud-based systems**, where the "operating system" a user experiences is really a coordinated layer spanning many physical machines; on **real-time operating systems (RTOS)** for the ever-growing population of Internet of Things (IoT) devices, which must react to physical-world events within strict, predictable deadlines; on progressively stronger built-in **security and isolation** mechanisms, since networked systems are now permanently exposed to attack; and on early **AI-assisted or AI-integrated** operating system components that manage complex, partly autonomous tasks such as adaptive power management, predictive resource scheduling, and workload placement across heterogeneous hardware (CPUs, GPUs, and specialised accelerators).

Table 1.1 — OS capability by hardware generation

Generation	Approx. period	Defining hardware	What the OS could newly do
First	1940s – early 1950s	Vacuum tubes, magnetic drums	Nothing — programs run by hand, machine language only
Second	Mid-1950s – early 1960s	Transistors, magnetic core memory	Automatic batch processing (jobs run one after another without operator intervention)
Third	Mid-1960s – early 1970s	Integrated circuits	Multiprogramming (overlap CPU and I/O across jobs) and time-sharing (interactive multi-user access)
Fourth	1970s – circa 2000s	Microprocessors, LSI/VLSI	Personal, single-user computing; portable multi-user UNIX; graphical, pre-emptively multitasked desktop systems
Fifth (present/emerging)	2010s – present	Multi-core mobile SoCs, cloud data centres, early quantum hardware	Mobile-first and touch-oriented OS design; distributed/cloud operating environments; real-time OS for IoT; AI-assisted resource management

CHECK YOUR PROGRESS 1.1

1. State the three goals an operating system tries to achieve simultaneously.
2. Why was there no operating system, in any real sense, during the first generation of computers?
3. What single hardware advance of the third generation made both multiprogramming and time-sharing possible, and how does time-sharing differ from multiprogramming in what it optimises for?
4. Name two operating systems associated with the fourth generation and one defining feature of each.
5. What kind of scheduling requirement distinguishes an operating system built for IoT devices from a general-purpose desktop operating system?

1.2 TYPES OF OPERATING SYSTEMS

The terms **multiprogramming**, **multitasking**, **timesharing**, **multithreading**, and **real-time** are among the most frequently confused vocabulary in an introductory operating systems course, largely because they are not mutually exclusive categories: a single modern operating system (Windows, Linux, Android) is simultaneously multiprogrammed, multitasking, time-shared among multiple logged-in users, and supports multithreaded applications. Each term names a *different design question* the operating system answers, and the confusion usually comes from not keeping those questions separate. This section defines each term precisely, in the specific way each differs from the others, before discussing real-time operating systems as a distinct category built around timing *guarantees* rather than concurrency per se.

1.2.1 Multiprogramming Operating Systems

Multiprogramming answers the question: *how do we keep the CPU busy when a program is waiting for I/O?* A multiprogramming operating system keeps several jobs resident in main memory at the same time. Whenever the job currently holding the CPU issues an I/O request and must wait for it to complete, the OS does not leave the CPU idle; instead, it switches the CPU to a different resident job that is ready to run. The historical motivation was purely one of hardware economics: in early systems, the CPU was far more expensive than the peripherals it drove, and a CPU that sat idle waiting for a slow card reader or tape drive was wasted, costly capacity.

IMPORTANT — Definition — Multiprogramming

Multiprogramming is the technique of keeping multiple programs resident in memory simultaneously and switching the CPU between them so that it is given useful work whenever the currently running program is blocked waiting for I/O, thereby maximising CPU utilisation.

Multiprogramming, by itself, says nothing about *how fast* the switching happens or whether a human user notices it. A multiprogramming batch system from the 1960s might run for minutes between switches. That is still multiprogramming, because the goal is CPU utilisation, not responsiveness. This is precisely the distinction that separates multiprogramming from multitasking.

Advantages: higher CPU utilisation and system throughput than running one job to completion at a time; better use of expensive hardware.

Limitations: requires memory large enough to hold multiple jobs at once; requires a scheduler and memory-protection mechanisms that did not exist in earlier systems; a job with very little I/O (a CPU-bound job) still monopolises the CPU for long stretches, which is fine for batch throughput but poor for interactivity.

Typical use: the classic mainframe batch-processing environment, and, conceptually, it is the substrate on which every later type of OS in this list (multitasking, time-sharing) is built.

1.2.2 Multitasking Operating Systems

Multitasking answers a different question: *how do we give the illusion that several programs are running at the same time on one CPU?* A multitasking OS rapidly switches ("time-slices") the CPU among multiple processes so quickly that, from a human's perspective, the processes appear to execute simultaneously, even though, on a single CPU core, only one instruction stream is actually executing at any given instant. Where multiprogramming's switching is triggered by a job blocking on I/O and is invisible to any user, multitasking's switching is deliberately frequent (milliseconds) and is specifically aimed at *user-perceived* concurrency and responsiveness: a user typing in one window while music plays in another and a file download in the background.

NOTE — Multiprogramming vs. Multitasking — the exact difference

Both keep the CPU switching between programs, but they optimise for different things. **Multiprogramming** optimises for CPU utilisation in a batch environment, and switches are triggered by a job blocking on I/O; the delay between switches can be long, and no interactive user is watching. **Multitasking** optimises for the *appearance* of simultaneous execution to an interactive user, and switches happen many times per second on a fixed, short time-slice regardless of I/O behaviour. In short: multiprogramming is about not wasting the CPU; multitasking is about making the CPU feel like it belongs to everyone at once. Every multitasking OS is also multiprogrammed, but not every multiprogrammed system needs to be multitasking (a batch system need not slice time finely, since no one is waiting for instant feedback).

There are two forms of multitasking. In **pre-emptive multitasking**, the operating system itself can forcibly interrupt a running process and hand the CPU to another process, whether the running process agrees or not; this is the approach used by every modern general-purpose OS (Windows since Windows 95/NT, Linux, macOS), because it prevents one badly behaved or hung program from freezing the whole system. In **cooperative multitasking**, a process must voluntarily give up the CPU; if a process fails to yield, because it is buggy, stuck in a loop, or simply badly written, every other process, and often the entire system, can hang. Cooperative multitasking was used in early consumer operating systems such as Windows 3.x and the Classic Mac OS before pre-emptive scheduling became standard.

Advantages: substantially improved productivity and user experience, since a user can run a browser, a document editor, and a media player at once; efficient use of otherwise-idle CPU time; supports background tasks (backups, updates, indexing) without freezing the foreground application.

Limitations: the extra bookkeeping (saving and restoring process state on every switch) is pure overhead compared to running one program at a time; pre-emptive multitasking requires more sophisticated hardware and OS support (timers, memory protection) than cooperative multitasking.

Typical use: essentially every general-purpose desktop, laptop, and mobile OS in use today.

1.2.3 Timesharing Operating Systems

Time-sharing takes the *same* mechanism as multitasking (rapid CPU switching in small time slices, or **quanta**) and applies it specifically to giving **multiple simultaneous users**, typically connected via separate terminals or remote sessions, the impression that each of them has the whole machine to themselves. Where multitasking is usually described from the point of view of one user running several programs, time-sharing is described from the point of view of several *different users*, each submitting work to a shared, central machine. Historically, this is exactly what it sounds like: a

mainframe's processing time was literally "shared" out, quantum by quantum, among dozens of people typing at terminals.

Advantages: dramatically more cost-effective than giving every user a dedicated computer, since expensive hardware is shared; each user still gets acceptably fast, interactive response as long as the system is not overloaded; naturally supports collaborative, multi-user work on shared data.

Limitations: the scheduling, memory-management, and synchronisation machinery needed to keep many users' work both fair and isolated from one another is considerably more complex than in a single-user system; more users sharing one system increases the attack surface and the consequences of a security failure, since a flaw can potentially expose another user's data; heavy contention among many active users can degrade everyone's response time.

Typical use: early UNIX and Multics installations, and, in modern form, any server that many users log into concurrently (a university's shared Linux server, a cloud virtual machine shared among a department).

1.2.4 Multithreading Operating Systems

Multithreading answers a question at a finer grain than any of the above: *how do we allow several independent streams of execution to progress within a single process?* A **thread** is the smallest unit of CPU scheduling, a single sequential flow of control with its own program counter, register set, and stack. Unlike a process, though, a thread shares its code, data, and open files with every other thread belonging to the same process. Multithreading lets a single process (say, a word processor) run several threads concurrently: one thread repaginating the document in the background while another remains responsive to keystrokes, and a third performs an auto-save, all inside the same process and sharing the same in-memory document data without the overhead of separate processes or explicit inter-process communication.

Threads are commonly implemented at three levels.

User-level threads are created and scheduled entirely by a library in user space, without the kernel's knowledge; they are extremely cheap to create and switch between, but the kernel, seeing only a single process, cannot run two user-level threads of the same process on two different CPU cores simultaneously, and if one thread makes a blocking system call the whole process can block.

Kernel-level threads are created and scheduled directly by the operating system kernel; they are more expensive to create and switch (each operation may require a system call) but can be scheduled independently, including onto different CPU cores, and a blocking call in one thread does not freeze the rest.

Hybrid (many-to-many) models multiplex a number of user-level threads onto a smaller or equal number of kernel-level threads, aiming for the low overhead of user-level threads together with the true concurrency of kernel-level threads.

NOTE — How multithreading differs from multitasking and multiprogramming

Multitasking and multiprogramming are both about switching the CPU between **processes**: independent programs with their own separate memory space. Multithreading is about switching (or truly running in parallel, on a multi-core machine) between **threads that belong to the same process** and therefore already share memory. This is why creating a thread and switching between threads is normally much cheaper than creating a process and performing a full context switch between processes: there is far less state to duplicate and swap, since the address space, open files, and most other resources are already shared.

Advantages: much lower creation and context-switching overhead than separate processes, since address space and resources are shared; a natural way to keep an application responsive while heavy work happens in the background; directly exploits multi-core hardware, since independent threads of the same process can run on different cores at the same instant.

Limitations: shared memory between threads means one buggy or malicious thread can corrupt data used by every other thread of that process, unlike the memory isolation between separate processes; programming with threads introduces genuinely hard problems that single-threaded or purely process-based designs avoid, including race conditions, deadlocks, and the need for explicit synchronisation (semaphores, mutexes, monitors); multithreaded programs are typically harder to debug because failures can depend on the precise, non-repeatable timing of thread interleaving. **Typical use:** virtually all modern interactive and server applications, including web browsers (one thread per tab or task), web servers (one thread per connection), and any application that must remain responsive while doing background work.

1.2.5 Real-Time Operating Systems

A **real-time operating system (RTOS)** answers yet another, orthogonal question: not "how many things run at once" but "*can we guarantee a task finishes within a specific deadline?*" An RTOS is built for applications that process events from the physical world as they occur, and where the *correctness* of a computed result depends not only on its value but on the moment it is delivered. Being fast on average is not enough; an RTOS must be **deterministic**: it must bound, precisely and predictably, the worst-case time between an event occurring and the system's response to it.

RTOS designs are split into two categories based on the consequence of missing a deadline:

- **Hard real-time systems** guarantee that critical tasks complete within a strict deadline; missing that deadline is treated as a total system failure, potentially with catastrophic real-world consequences. Examples: an automotive airbag controller, which must trigger within milliseconds of a crash being detected; a pacemaker, which must deliver an electrical pulse at the exact required moment.
- **Soft real-time systems** give scheduling priority to time-sensitive tasks but tolerate an occasional missed deadline as a degradation in quality rather than a catastrophic failure. Examples: video streaming, where an occasional dropped or delayed frame causes a visible stutter but not a disaster; audio processing, where a rare glitch is undesirable but recoverable.

Advantages: highly predictable, reliable behaviour that general-purpose operating systems (which are optimised for average-case throughput, not worst-case latency) cannot guarantee; minimal, tightly bounded interrupt- and task-switching latency; efficient, deterministic use of CPU, memory, and I/O resources tailored to the application.

Limitations: achieving determinism generally means sacrificing the flexibility and raw average-case throughput of a general-purpose OS; RTOS-based systems and their specialised hardware are often significantly more expensive to design, certify, and validate than commodity systems; an RTOS is usually built and tuned for a narrow class of applications and is not a good general-purpose substitute for a desktop or server OS.

Typical use and representative systems: embedded control in automotive systems (engine control units, advanced driver-assistance systems), industrial automation and robotics, aerospace and defence (avionics, guidance systems), and medical devices (infusion pumps, pacemakers). Representative RTOS products include VxWorks, FreeRTOS, QNX (built on a microkernel architecture), and real-time variants of Linux.

Table 1.2 — OS categories: what each switches between

Type	What it actually switches between	What it optimises for	Representative example
Multiprogramming	Whole processes, switched when one blocks on I/O	CPU utilisation in a batch environment	Early IBM mainframe batch systems
Multitasking	Whole processes, switched on a short, regular time-slice	Apparent simultaneous execution for one interactive user	Windows, Linux, macOS desktop

Type	What it actually switches between	What it optimises for	Representative example
Timesharing	Whole processes belonging to many different users	Fair, responsive access for many simultaneous users on one machine	A shared UNIX/Linux server, a multi-user mainframe
Multithreading	Threads within the same process (shared memory)	Low-overhead concurrency and responsiveness inside one application	A browser (one thread per tab), a web server
Real-time (RTOS)	Tasks, chosen strictly by deadline/priority	Guaranteed, predictable response time, not raw throughput	VxWorks, FreeRTOS, pacemaker firmware

CHECK YOUR PROGRESS 1.2

1. Multiprogramming and multitasking both involve switching the CPU between programs. What, precisely, does each optimise for, and what event typically triggers a switch in each?
2. Explain, in one sentence, how timesharing differs from ordinary (single-user) multitasking.
3. Why is creating and switching between threads generally cheaper than creating and switching between whole processes?
4. Distinguish a hard real-time system from a soft real-time system, with one example of each.
5. Explain why a modern smartphone OS can correctly be described, at the same time, as multiprogrammed, multitasking, and multithreaded.

1.3 DIFFERENT VIEWS OF THE OPERATING SYSTEM

"What is an operating system?" does not have one single answer, because the same piece of software looks completely different depending on who is looking at it. A person browsing the web, a programmer calling a library function, and an engineer writing the kernel's memory manager are all, in a sense, dealing with "the operating system," but they care about entirely different aspects of it. Describing all three perspectives is the standard way an OS course frames the subject before diving into internals.

1.3.1 The User's View

From an ordinary **user's** perspective, the operating system *is* the interface: the desktop, icons, windows, and menus (or, on a phone, the home screen and app switcher) through which the user launches applications, manages files, and configures settings. A user judges an operating system almost entirely by usability criteria that have nothing to do with its internal design: is the interface intuitive and easy to learn; does the range of compatible applications cover what the user needs; does the system feel fast and responsive; and does it keep the user's data and privacy reasonably safe. Whether the underlying kernel is monolithic or layered, whether scheduling uses Round Robin or a multilevel feedback queue, is invisible and, from this viewpoint, irrelevant. The user's view is defined entirely by *what the system lets them accomplish*, not by *how* it accomplishes it.

1.3.2 The System Programmer's View

The **system programmer** (sometimes called the systems view, or the resource-manager view) sees the operating system as the entity responsible for allocating and managing every shared hardware and software resource on the machine: the CPU, main memory, storage, and I/O devices. Concretely, this view is organised around the OS's major management responsibilities: creating, scheduling, and terminating **processes** (including handling multitasking, multithreading, and the synchronisation needed when they run concurrently); allocating and reclaiming **memory**, including virtual memory, paging, and segmentation; organising, storing, and retrieving data

through **file system management**; and controlling physical hardware through **device management**, mediated by drivers. Where the user's view stops at the interface, the system programmer's view is precisely everything *behind* that interface: the operating system, seen this way, is a resource manager whose job is to arbitrate demand for scarce shared resources among many competing processes, fairly and efficiently.

1.3.3 The (Applications) Programmer's View

Closely related to, but distinct from, the systems view is the view of an ordinary **applications programmer** writing software that runs *on top of* the operating system rather than software that implements the operating system itself. To this programmer, the OS is an **abstraction layer**: it hides the bewildering, hardware-specific complexity of real disks, network cards, and CPUs behind a small, stable, well-documented set of **Application Programming Interfaces (APIs)**: functions for opening a file, allocating memory, creating a thread, or sending a network packet, that behave the same way (or nearly so) across many different underlying hardware configurations. The programmer's view also includes the tools an OS or its ecosystem supplies to build software in the first place: compilers, linkers, debuggers, and libraries, together with concurrency primitives (semaphores, mutexes, condition variables) that the OS exposes so that programmers do not need to implement thread scheduling or memory protection themselves.

1.3.4 The Internal (Kernel) View

Finally, there is the view from *inside* the operating system itself: the perspective of the people who actually build the kernel. Here, the operating system is neither an interface nor an API but a concrete piece of engineering with its own internal architecture: a specific way of dividing the kernel's own code into components, deciding which parts run with full hardware privilege and which do not, and defining precisely how those internal components call one another. This internal, structural view is the subject of Section 1.4, where it will become clear that "the operating system" can be built as a single tightly-integrated block of code (a monolithic kernel), as a stack of increasingly abstract layers, or as a small, minimal core surrounded by services running outside it (a microkernel). These are three fundamentally different engineering answers to the same question of *how the OS should be built*, and all of them can present an *identical* user's view and an *identical* programmer's view on the outside.

Table 1.3 — Four perspectives on an operating system

View	Who holds it	What "the operating system" means from this view	Primary concern
User's view	An ordinary end user	The interface — desktop, apps, files, settings	Ease of use, responsiveness, application support, security of personal data
System programmer's view	Whoever manages/administers the system's resources	A resource manager for CPU, memory, storage, and I/O	Efficient, fair allocation of shared resources among competing processes
(Applications) programmer's view	Someone writing software that runs on the OS	An abstraction layer exposed through APIs, libraries, and dev tools	A stable, hardware-independent interface for building applications
Internal (kernel) view	Whoever builds the operating system itself	A specific internal architecture (monolithic, layered, microkernel, ...)	Correctness, performance, and maintainability of the kernel's own code

CHECK YOUR PROGRESS 1.3

1. Why might two people looking at the same operating system give completely different answers to "what is an operating system"?
2. From the system programmer's view, name the four major categories of resource the OS manages.
3. What does an API provide to an applications programmer, and why does this matter for portability across hardware?
4. Explain why the internal (kernel) view can differ completely between two operating systems while their user's view looks almost identical.

1.4 OPERATING SYSTEM CONCEPTS AND STRUCTURE

1.4.1 The Core Services an Operating System Provides

Regardless of how it is internally engineered, every general-purpose operating system provides broadly the same set of core services, corresponding closely to the "system programmer's view" of Section 1.3.2:

- **Process management.** Creating, scheduling, suspending, resuming, and terminating processes and threads; providing mechanisms for inter-process communication and synchronisation so that concurrently running programs can cooperate safely. This is the focus of the remainder of this unit (Sections 1.5–1.9).
- **Memory management.** Allocating and reclaiming main memory among competing processes, and, in most modern systems, presenting each process with the illusion of a large, private, contiguous address space through **virtual memory**, implemented via techniques such as **paging** (fixed-size pages) and **segmentation** (variable-sized, logically meaningful segments). (Covered in depth in a later unit.)
- **File system management.** Organising persistent data into files and directories, and providing operations to create, read, write, delete, and control access to them, on top of concrete file system formats such as FAT32, NTFS, ext4, or APFS. (Covered in depth in a later unit.)
- **Device management.** Providing a uniform interface to a huge variety of physical hardware through **device drivers**, and using techniques such as **buffering** (temporarily holding data to smooth out speed differences between a fast producer and a slow consumer, or vice versa) and **spooling** (queuing jobs, such as print jobs, until a device is ready to accept them) to manage devices efficiently. (Covered in depth in a later unit.)
- **Security and protection.** Verifying who a user or process claims to be (**authentication**), deciding what an authenticated identity is permitted to do (**authorisation**), and enforcing that decision through mechanisms such as access control lists, capabilities, and encryption, so that processes and users cannot interfere with resources they have no right to touch.

Unit I focuses in depth on process management, since a clear understanding of what a process *is* and how the OS schedules it underlies almost everything else an operating system does; memory, file, and device management are treated fully in the units that follow.

1.4.2 Structuring the Operating System Itself

The services above describe *what* an operating system does. A separate, equally important question is *how the OS's own code should be organised internally* to provide those services reliably and maintainably. Four structuring approaches are commonly taught; the syllabus for this unit focuses on the first two, **layered** and **monolithic**, as the two classical extremes, with microkernel and modular/hybrid designs presented as the modern alternatives that grew out of their respective strengths and weaknesses.

Monolithic systems. In a **monolithic kernel**, essentially the entire operating system (process management, memory management, the file system, device drivers, the networking stack, and the system call interface) runs together as one large program in a single address space, at the highest hardware privilege level. Any part of the kernel can call any other part directly, through ordinary function calls, because everything shares the same memory space.

IMPORTANT — Definition — Monolithic Kernel

A **monolithic kernel** is an operating system architecture in which the entire kernel, meaning every core service the OS provides, executes as a single program in one address space, with components communicating through direct function calls rather than message passing across a protection boundary.

Because components call one another directly, with no message-passing or context-switch overhead between them, a well-implemented monolithic kernel can be very fast. Its design is also, in one sense, simpler to reason about than a system split across many independently running components, since everything lives in one place. The Linux kernel is the most widely deployed modern example (traditional UNIX and the early Windows 9x kernels are historical examples). The cost of this design is that it sacrifices **modularity**: because every component can directly touch every other component's data, a bug in one driver can corrupt memory belonging to a completely unrelated subsystem, or crash the entire kernel outright; the tight coupling between components makes large sections of the kernel harder to maintain, test in isolation, and safely extend as new hardware and features are added; and because the kernel is one indivisible unit, it is comparatively difficult to replace or reconfigure just one piece of it without touching the whole.

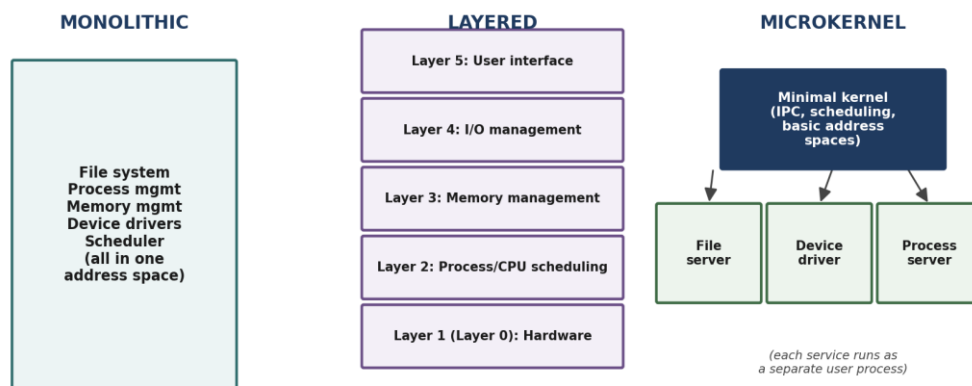


Figure 1.1 — Three kernel structures: monolithic, layered, and microkernel

Layered systems. A **layered operating system** is built as a stack of layers, numbered from layer 0 (closest to the hardware) up to the highest layer (closest to the user), where each layer is built using only the services provided by the layer immediately beneath it, and in turn offers its own services to the layer immediately above it. A typical layered decomposition, from bottom to top, might look like: a **hardware layer** (the physical CPU, memory, and I/O devices); a **hardware abstraction layer (HAL)** that hides device-specific detail behind a uniform interface; a **kernel layer** handling core scheduling, memory management, and basic I/O; a **device driver layer**; a **system call interface** through which user applications request OS services; a **system services layer** offering higher-level functionality (file management, networking); and, at the top, a **user interface layer** (a GUI or a command shell).

IMPORTANT — Definition — Layered Operating System

A **layered operating system** is an operating system structured as a hierarchy of layers, where each layer offers services built exclusively from the services of the layer directly below it, so that

a layer's internal implementation can be understood, changed, and debugged without needing to understand the implementation of every layer above or below it.

The layered approach is essentially a large-scale application of information hiding and abstraction to kernel design, and it delivers the benefits typical of that discipline: strong **modularity**, since each layer has one well-defined role and hides its internal details from the layers above it; easier **debugging and maintenance**, because a fault can usually be localised to a specific layer rather than searched for across the entire kernel; and better **fault isolation**, since a well-designed layer boundary limits how far a bug in one layer can propagate. Dijkstra's early **THE** operating system, built from six strictly ordered layers, is the classic teaching example; Windows NT's hybrid kernel is a modern system that borrows heavily from the layered idea (a hardware abstraction layer beneath a kernel beneath a set of subsystems) while not being purely layered throughout. The costs of strict layering are **performance overhead** (a request from the top layer may have to pass through several layers of indirection before it reaches the hardware, and each layer crossing has some cost), the **design difficulty of choosing the right layer boundaries** so that a layer never needs a service only a *higher* layer can provide, and a certain **rigidity**, since a strict hierarchy can make it awkward to optimise a case that naturally needs to skip layers.

NOTE — Layered vs. Monolithic — the essential trade-off

A monolithic kernel gives every component direct, cheap access to every other component, at the cost of tangled dependencies where a fault anywhere can compromise everything. A layered kernel enforces strict, one-directional dependencies (each layer uses only the layer below it), trading some run-time performance and design flexibility for much stronger modularity, easier debugging, and better fault containment. Neither is strictly "better": the choice is a trade-off between raw speed and simplicity of internal calling on one hand, and structural discipline and maintainability on the other.

Microkernel and modular/hybrid structures. Because both extremes have real costs, most systems in active use today do not sit purely at either end. A **microkernel** design pushes the layered idea further: it keeps only the most essential mechanisms (basic inter-process communication, minimal memory management, and basic scheduling) running in privileged kernel mode, and moves everything else (file systems, device drivers, networking) out into separate processes running in ordinary user space, which communicate with the tiny kernel and with each other via message passing. This maximises fault isolation (a crashing file-system server does not take down the whole kernel) and security, at the cost of the message-passing overhead every service request now incurs; QNX and Minix are representative microkernels. A **modular** (or hybrid) kernel, the approach used by the modern Linux and Windows kernels among others, keeps a large privileged core for performance-critical work but organises it internally as a set of dynamically loadable modules (such as device drivers) that can be inserted and removed from the running kernel without the tight, hard-wired layering of a strictly layered design or the message-passing cost of a true microkernel. It is an attempt to capture the maintainability benefits of modularity without abandoning the performance of direct, in-kernel function calls.

Table 1.4 — Four kernel structuring approaches

Structuring approach	Where core services run	Coupling between components	Typical strength	Typical weakness	Example systems
Monolithic	All in one privileged address space	Tight (direct function calls)	High performance, simple internal calling	Fault in any part can crash the whole kernel; hard to isolate or replace one piece	Linux, traditional UNIX

Structuring approach	Where core services run	Coupling between components	Typical strength	Typical weakness	Example systems
Layered	Privileged, but organised as strict hierarchy of layers	Strict, one-directional (each layer only uses the layer below)	Strong modularity, easier debugging and fault isolation	Overhead of crossing layers; rigid to redesign	Dijkstra's THE OS; Windows NT (partially)
Microkernel	Minimal core privileged; most services in user-space processes	Loose (message passing across processes)	Excellent fault isolation and security	Message-passing overhead reduces raw performance	QNX, Minix
Modular / hybrid	Large privileged core with pluggable, loadable modules	Mixed — tight within the core, modular at the edges	Performance close to monolithic with more maintainability	Still shares some of monolithic design's coupling risk within the core	Modern Linux, modern Windows

CHECK YOUR PROGRESS 1.4

1. List the five core services every general-purpose operating system provides.
2. In a monolithic kernel, how do components communicate with one another, and what is the main risk this creates?
3. In a layered operating system, what rule governs which services a given layer is allowed to use?
4. What specific problem does a microkernel design solve that a monolithic kernel does not, and at what cost?
5. How does a modular (hybrid) kernel try to combine the benefits of the monolithic and microkernel approaches?

1.5 THE PROCESS CONCEPT

1.5.1 Process vs. Program

A **program** is a passive entity: a file sitting on disk, containing executable instructions, that does nothing by itself until someone runs it. A **process**, by contrast, is an *active* entity, a program's instructions in the middle of being executed, together with all the additional state that execution requires: the current value of the program counter, the contents of the CPU's registers, and a private area of memory holding the program's data and its call stack. A process comes into being at the moment a program's executable file is loaded into memory and execution begins (whether triggered by a user double-clicking an icon, typing a command at a shell, or one running program calling a system function to launch another).

IMPORTANT — Definition — Process

A **process** is a program in execution. It comprises the program's code (the **text section**), its current activity as represented by the value of the **program counter** and the contents of the **processor registers**, a **stack** containing temporary data such as function parameters, return addresses, and local variables, a **data section** containing global variables, and a **heap** containing memory allocated dynamically while the program runs.

This distinction (passive program versus active process) also explains why a single program can correspond to *many* simultaneous processes. If three different users each launch the same text editor, there is exactly one copy of the editor's executable file on disk, but three separate processes are created, each with its own program counter, its own register values, and its own private

memory for data and stack, so that the three users' editing sessions do not interfere with one another in any way.

1.5.2 Process Memory Layout

The memory a process uses while it runs is conventionally divided into four regions. The **text section** holds the compiled program instructions and is typically read-only once loaded (so that a process cannot accidentally, or maliciously, modify its own code while running). The **data section** holds global and static variables, whose values persist for the life of the process. The **heap** is a region for memory the program allocates and frees dynamically at run time (for example, memory obtained via a language's dynamic-allocation function); it grows and shrinks as the program's needs change. The **stack** holds the transient information needed by function calls — parameters, return addresses, and local variables — and grows and shrinks automatically as functions are called and return. This memory layout, tracked per-process, is one of the pieces of information the operating system must record for every process. This becomes important in Section 1.6, when the Process Control Block is introduced as the structure that records it.

1.5.3 Process States and the State-Transition Diagram

As a process executes, it moves through a small, well-defined set of **states**, reflecting what the operating system is currently doing with it, or what it is currently waiting for:

- **New.** The process is being created: the OS is doing the initial work of setting up its data structures, but the process is not yet ready to compete for the CPU.
- **Ready.** The process has everything it needs to run and is waiting only for the CPU to become available; it sits in a **ready queue** maintained by the scheduler.
- **Running.** The process's instructions are actually being executed by the CPU. On a single-core system, at most one process can be in this state at any instant.
- **Waiting (or blocked).** The process cannot continue until some event occurs, most commonly the completion of an I/O operation, but also, for example, waiting to receive a signal or acquire a lock, and so it is set aside, out of contention for the CPU, until that event happens.
- **Terminated.** The process has finished executing (normally, by completing its final instruction and calling `exit`, or abnormally, due to an error), and the operating system is in the process of reclaiming the resources it held.

A process moves through these five states over its lifetime

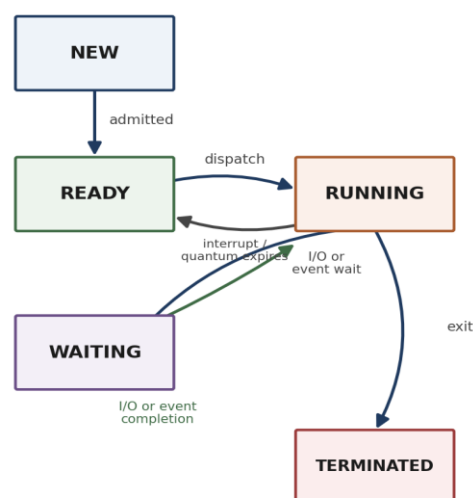


Figure 1.2 — Process state-transition diagram

A process moves between these states according to specific, well-defined transitions, never arbitrarily: **New** -> **Ready** once the OS finishes setting the process up; **Ready** -> **Running** when the scheduler dispatches the process onto the CPU; **Running** -> **Waiting** when the process issues a blocking request such as an I/O read; **Waiting** -> **Ready** once the event the process was waiting for occurs; **Running** -> **Ready** when the process is pre-empted (its time quantum expires, or a higher-priority process becomes ready) before it has finished or blocked; and **Running** -> **Terminated** when the process completes or is aborted. Notice, in particular, that a process can never move directly from Waiting back to Running. It must always pass back through Ready and wait for the scheduler to choose it again, because being unblocked only makes a process *eligible* to run, not immediately running.

1.5.4 Context Switching

Whenever the CPU switches from executing one process to executing another, whether because the running process blocked, was pre-empted, or terminated, the operating system must perform a **context switch**: it saves the complete execution state of the process being suspended (program counter, register contents, and other per-process scheduling and memory-management information) into that process's Process Control Block (Section 1.6), and then loads the previously saved state of the next process to run from *its* Process Control Block, so that the newly scheduled process resumes exactly as it was when it was last suspended.

A context switch is pure overhead: the CPU does no useful work for the process while it is busy saving and restoring state, so time spent context switching is time not spent executing any user's instructions. This overhead is one of the central reasons the details of a scheduling algorithm matter so much (Sections 1.8–1.9): an algorithm that switches processes too frequently (an unnecessarily small-time quantum in Round Robin, for instance) can lose a meaningful fraction of total CPU capacity purely to switching overhead, even though every individual switch is fast.

CHECK YOUR PROGRESS 1.5

1. Explain the difference between a program and a process, and explain how one program can correspond to several simultaneous processes.
2. List the five standard process states and state one event that causes a transition into each.
3. Why can a process never move directly from the Waiting state to the Running state?
4. What information must a context switch save and restore, and why is a context switch considered pure overhead?

1.6 THE PROCESS CONTROL BLOCK

1.6.1 What the PCB Stores, and Why

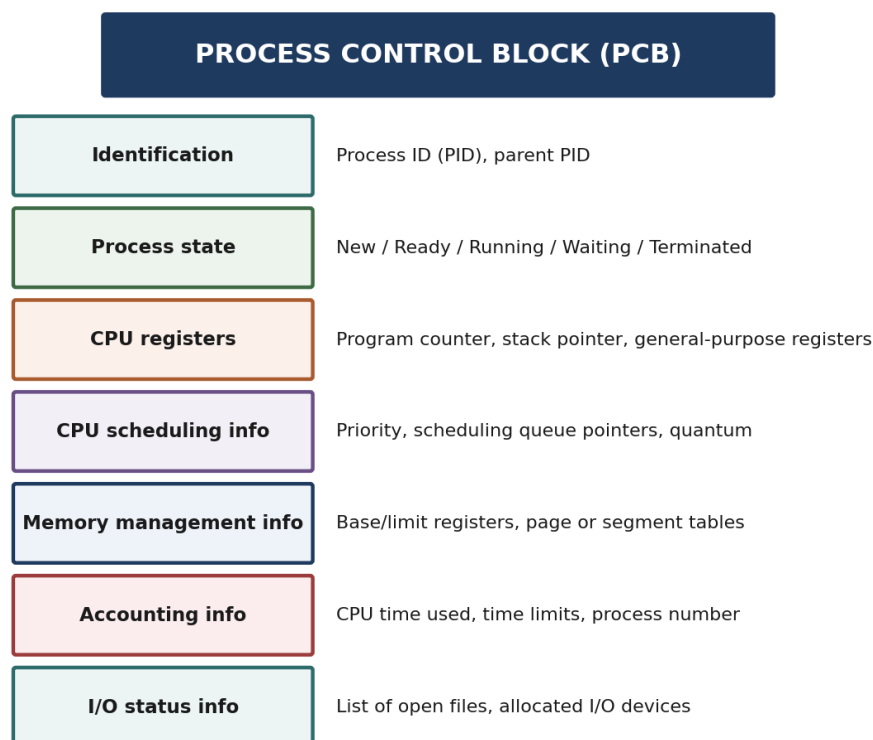
Every fact the operating system needs to know about a process, such as which state it is in, where it should resume execution, or what resources it holds, has to live *somewhere* the OS can find it instantly, for every process, at every context switch. That "somewhere" is the **Process Control Block (PCB)**: a single data structure, maintained by the operating system, that acts as the process's complete administrative record for as long as it exists.

IMPORTANT — Definition — Process Control Block

The **Process Control Block (PCB)** is a data structure maintained by the operating system that stores all the information needed to manage a particular process: its identity, its current state, its saved execution context, its scheduling and memory-management information, and its resource usage, so that the process can be suspended and later resumed exactly where it left off.

Table 1.5 — Key fields stored in a Process Control Block

Field	What it holds	Why the OS needs it
Process ID (PID)	A unique identifier for the process	Lets the OS (and other processes) unambiguously refer to this specific process among all others
Process state	New, Ready, Running, Waiting, or Terminated	Tells the scheduler which queue the process belongs in and what it is currently doing
Program counter	The address of the next instruction to execute	Lets execution resume from exactly the right point after a context switch
CPU registers	Saved contents of all general-purpose and status registers	Restores the exact computational state the process had when it was last suspended
CPU scheduling information	Priority, scheduling queue pointers, and similar scheduling parameters	Let's the scheduler decide when and for how long this process should next run
Memory management information	Base/limit registers, page tables, or segment tables, depending on the memory scheme in use	Tells the OS exactly which memory this process is entitled to use
Accounting information	CPU time consumed, wall-clock start time, limits, and similar statistics	Supports billing, performance monitoring, and scheduling decisions based on past behaviour
I/O status information	The list of open files and any pending I/O requests or allocated devices	Let's the OS track and eventually reclaim every resource the process is using



Every field the OS needs to suspend a process and resume it later, exactly where it left off

Figure 1.3 — Structure of a Process Control Block

1.6.2 The PCB as the Mechanism Behind Everything Else

Every process-related concept discussed so far in this unit is, underneath, an operation on a PCB. A **context switch** (Section 1.5.4) is literally the act of saving the running process's registers and program counter into its PCB and loading the next process's saved registers and program counter from its PCB. A process's **state transition** (Section 1.5.3) is a change to the state field of its PCB, together with moving a pointer to that PCB from one scheduling queue (say, the waiting queue for

a particular device) to another (the ready queue). The **scheduler**, when it decides which process to run next (Section 1.8), is really just choosing a PCB from the ready queue according to whatever algorithm is in force, and the specific field of the PCB it consults (priority, remaining burst time, arrival time) depends on which algorithm is being used. Without a PCB to persist a process's exact state between the moments it is switched out and switched back in, none of these operations would be possible, because the OS would have no reliable, structured place to remember what it needed to.

1.6.3 The Systems Programmer's View of Process Manipulation

From the systems programmer's perspective introduced in Section 1.3.2, a running operating system is, in a very concrete sense, a collection of PCBs linked together in various queues, and much of process-management code is code that moves PCBs between those queues or updates fields within them. Internally, the kernel typically keeps a **process table** (an array or linked structure of all PCBs currently known to the system) together with separate queues (often implemented as linked lists threaded through the PCBs themselves) for processes that are ready to run, processes waiting on each specific device or event, and so on. Implementing **process creation** (Section 1.7.1) means allocating a fresh PCB, populating its fields, and inserting it into the ready queue; implementing **process termination** means removing the process's PCB from whatever queue it occupies, releasing the memory and other resources recorded in it, and finally deallocating the PCB itself; implementing a **scheduling algorithm** means deciding, using the fields of the PCBs currently in the ready queue, which one to dispatch next; and implementing **inter-process communication or synchronisation primitives** typically means updating the state field of the PCBs of processes that must block waiting for a resource, and moving them back to the ready queue once that resource becomes available. Seen this way, the PCB is not just a passive record. It is the fundamental unit that essentially all of the operating system's internal process-management logic operates on.

CHECK YOUR PROGRESS 1.6

1. What is a Process Control Block, and why must the OS maintain one for every process?
2. Name any four fields stored in a PCB and explain what each is used for.
3. Explain how a context switch is, underneath, really just an operation on two PCBs.
4. From the systems programmer's view, what does "process creation" concretely involve doing to a PCB?

1.7 OPERATING SYSTEM SERVICES FOR PROCESS MANAGEMENT

1.7.1 Process Creation

A new process is normally created by an *existing* process, giving rise to a natural **parent-child** relationship and, over the lifetime of a running system, a whole tree of processes descending from one or a few initial processes started at boot time. On UNIX-family systems, process creation conceptually decomposes into two separate steps. **`fork`** creates a new process (the child) that is, at the instant of the call, an almost exact copy of the calling process (the parent): same code, same data, a duplicate of the parent's memory image, differing only in a fresh PID and a few other process-specific fields. **`exec`** then replaces the calling process's memory image with a *different* program, loading fresh code and data over the top of whatever was there before, without creating a new process. Combining the two (fork a copy of the shell, then have the child exec the requested program) is exactly how a command-line shell launches every command a user types. A single umbrella term, **`spawn`**, is often used (particularly outside the UNIX tradition) to describe creating a new process running a specified program directly, without necessarily separating it into a distinct fork-then-exec sequence.

1.7.2 Process Termination

A process ends its life in one of two ways. Normal termination happens when a process finishes its work and voluntarily calls `exit`, at which point the OS reclaims every resource recorded in its PCB (memory, open file handles, allocated devices) and removes the PCB itself. Abnormal termination happens when the OS itself forces a process to stop, typically through an `abort`-style operation, because the process has violated a protection rule, exceeded a resource limit, or encountered a fatal, unrecoverable error. When a parent process terminates, many operating systems also terminate (or reassign) its children, a policy called **cascading termination**, since a child process, particularly one that shares resources set up by its parent, may not be able to run correctly, or may not be wanted at all, once its parent no longer exists.

1.7.3 Process Scheduling as a Service

Deciding *which* ready process gets the CPU next, and for how long, is itself a continuous service the OS provides; it is the subject of Sections 1.8 and 1.9 in detail. As a service, scheduling has to reconcile several system-wide objectives that can pull in different directions: minimising the time an interactive user waits for a response, maximising the number of jobs completed per unit time (throughput), keeping the CPU as busy as possible (utilisation), and ensuring the system continues to support multiprogramming (several jobs resident and progressing at once) rather than starving some processes of the CPU entirely.

1.7.4 Support for Inter-Process Communication (IPC)

Since concurrently running processes frequently need to exchange data or coordinate their actions, the OS provides standard **inter-process communication** mechanisms rather than leaving every application to invent its own. **Shared memory** lets two or more processes map the same region of physical memory into their respective address spaces, so that once set up, communication is as fast as an ordinary memory access, at the cost of the OS no longer being able to protect one process's data from another's mistakes in that shared region. **Message passing** (via pipes, message queues, or explicit send/receive calls) instead routes data through the kernel, which copies it between processes; this is slower than shared memory, since every message is copied, but keeps the two communicating processes cleanly isolated from each other's memory otherwise. Closely related, **synchronisation primitives** (semaphores, mutexes, and monitors) are provided by the OS so that cooperating (or competing) processes and threads can safely coordinate access to any resource they share, avoiding the race conditions that would otherwise result from unsynchronised concurrent access.

1.7.5 System Calls: The Interface Through Which These Services Are Requested

None of the process-management services above are accessed by an application directly reaching into kernel memory; they are requested through **system calls**: controlled entry points that let a user-mode program ask the kernel, running in a more privileged mode, to perform an operation on its behalf. Conceptually, `fork`, `exec`, `exit`, and `wait` are the system calls that implement process creation and termination; a kill-style call requests that another process be sent a signal or terminated; and calls such as `pipe`, `shmget` (shared memory) or `msgsnd/msgrcv` (message queues) implement IPC. This is the concrete mechanism that connects the abstract "services" described above to the "programmer's view" of the OS discussed in Section 1.3.3. From an applications programmer's perspective, the operating system's process-management services *are*, quite literally, this set of callable functions.

Table 1.6 — Representative system calls for process management

Category	Representative system calls (conceptual)	Purpose
Process creation	<code>fork</code> , <code>exec</code> , <code>spawn</code>	Create a new process; load a new program into a process's memory

Category	Representative system calls (conceptual)	Purpose
Process termination	exit, abort	End a process normally or forcibly, reclaiming its resources
Waiting/synchronising on children	wait	Let a parent block until a specified child terminates
Signalling	kill (send signal)	Notify or terminate another process
Shared-memory IPC	shmget, shmat	Set up and attach a memory region shared between processes
Message-passing IPC	pipe, msgsnd, msgrcv	Send and receive data through a kernel-managed channel

CHECK YOUR PROGRESS 1.7

1. Explain the difference between what fork does and what exec does, and how a shell typically uses both together to run a command.
2. What is cascading termination, and why might an OS terminate a process's children when the process itself terminates?
3. Compare shared memory and message passing as two approaches to inter-process communication, in terms of speed and isolation.
4. What is a system call, and why can't an ordinary application simply perform process-management operations directly without going through one?

1.8 PROCESS SCHEDULING ALGORITHMS

1.8.1 Scheduling Fundamentals

A process alternates between two kinds of activity throughout its life: bursts of computation on the CPU (**CPU bursts**) separated by periods spent waiting for I/O (**I/O bursts**). The **CPU scheduler** is the part of the operating system responsible for choosing which of the processes currently sitting in the **ready queue** gets the CPU next, whenever the CPU becomes free; the **dispatcher** is the (much simpler) component that actually performs the resulting context switch, loading the chosen process's saved state from its PCB and transferring control to it.

Scheduling algorithms are classified as **non-preemptive** or **preemptive**. A non-preemptive algorithm, once it has given the CPU to a process, lets that process run until it either finishes or voluntarily blocks (for I/O); the scheduler is not consulted again until the process gives up the CPU by itself. A preemptive algorithm can forcibly take the CPU away from a running process, because its time quantum expired or because a process with a stronger claim (say, a shorter remaining burst, or a higher priority) has just become ready, and hand it to a different process instead. Every algorithm discussed below is explicitly non-preemptive or preemptive, and this distinction is one of the most exam-relevant facts about each of them.

Throughout this section, the following notation is used for every worked example:

AT = arrival time,

BT = CPU burst (service) time,

CT = completion time,

TAT = turnaround time ($TAT = CT - AT$, the total time from a process's arrival to its completion),

WT = waiting time ($WT = TAT - BT$, the portion of turnaround time spent *not* running), and

RT = response time (the time from arrival until the process *first* gets the CPU, $RT = (\text{first time scheduled}) - AT$).

For any algorithm that is non-preemptive, response time and waiting time are identical for every process, because once a process starts running it runs straight through to completion without

interruption; for a preemptive algorithm the two can differ, since a process may be interrupted and resumed more than once. A full, precise treatment of *why* these particular five criteria matter, and how the algorithms below compare against one another on each, is given in Section 1.9; this section's job is to define each algorithm exactly and demonstrate it on a fully worked, arithmetically verified example.

1.8.2 First Come First Serve (FCFS)

FCFS is the simplest possible scheduling algorithm: processes are served strictly in the order they arrive in the ready queue, with no regard to how long each one needs the CPU for. It is **non-preemptive**: once a process starts running, it runs to completion (or until it blocks for I/O) before the next process in line is considered.

IMPORTANT — Definition — FCFS Scheduling

First Come First Serve (FCFS) scheduling dispatches the process at the head of the ready queue (the process that arrived earliest and has not yet run) and allows it to run to completion before considering the next process; ties among simultaneous arrivals are broken by an arbitrary but fixed order (commonly process ID).

Worked example. Consider four processes with the following arrival and burst times:

Process	Arrival Time (AT)	Burst Time (BT)
P1	0	4
P2	1	3
P3	2	1
P4	3	2

Since all four have arrived by the time P1 finishes at $t = 4$, FCFS simply runs them in arrival order thereafter:

0	4	7	8	10
P1	P2	P3	P4	

Table 1.7 — FCFS worked example

Process	AT	BT	CT	TAT (CT-AT)	WT (TAT-BT)	RT
P1	0	4	4	4	0	0
P2	1	3	7	6	3	3
P3	2	1	8	6	5	5
P4	3	2	10	7	5	5

Average TAT = $(4 + 6 + 6 + 7) / 4 = 23 / 4 = 5.75$.

Average WT = $(0 + 3 + 5 + 5) / 4 = 13 / 4 = 3.25$.

Since FCFS is non-preemptive, Average RT = Average WT = 3.25 as well.

Advantages: trivially simple to implement (a plain queue); intuitively fair in the narrow sense that whoever asks first is served first.

Limitations: the algorithm takes no account of how long a process needs the CPU, so a single long process arriving early forces every later, possibly very short, process to wait behind it, a phenomenon known as the **convoy effect**.

This is visible in the example above, where P3, needing only 1 unit of CPU time, still waits 5 units purely because it arrived just after the long-running P1. FCFS is consequently a poor choice for

interactive systems, where short, latency-sensitive requests routinely get stuck behind long-running ones.

1.8.3 Shortest Job First (Non-Preemptive) and Shortest Remaining Time Next (Preemptive)

"Shortest Run Time Next" names a *family* of two related algorithms that both select, from among the processes competing for the CPU, the one with the least CPU work remaining. They differ, though, in one crucial respect: whether a newly arriving process with an even shorter burst can interrupt a process already running.

Shortest Job First (SJF), non-preemptive, selects, whenever the CPU becomes free, the process in the ready queue with the smallest *total* burst time, and then runs it to completion without interruption, exactly like FCFS once the choice is made; the only difference from FCFS is the rule used to pick *which* waiting process runs next.

IMPORTANT — Definition — SJF (Non-Preemptive)

Shortest Job First (SJF) scheduling selects, each time the CPU becomes idle, the ready process with the smallest total CPU burst time and runs it to completion; ties are broken by arrival order (or process ID). It requires the scheduler to know (or accurately estimate) each process's burst time in advance.

Worked example (SJF, non-preemptive).

Process	AT	BT
P1	0	7
P2	2	4
P3	4	1
P4	5	4

At $t = 0$, only P1 has arrived, so it must run first regardless of its length: 0-7.

At $t = 7$, P2 (BT 4), P3 (BT 1), and P4 (BT 4) have all arrived; the shortest is P3, so it runs 7-8.

At $t = 8$, P2 and P4 remain, tied at BT 4; the tie is broken by arrival order, so P2 (which arrived at $t = 2$, before P4's $t = 5$) runs 8-12, and P4 runs last, 12-16.

0 7 8 12 16

P1	P3	P2	P4
----	----	----	----

Table 1.8 — SJF (non-preemptive) worked example

Process	AT	BT	CT	TAT	WT	RT
P1	0	7	7	7	0	0
P2	2	4	12	10	6	6
P3	4	1	8	4	3	3
P4	5	4	16	11	7	7

Average TAT = $(7 + 10 + 4 + 11) / 4 = 32 / 4 = 8$.

Average WT = $(0 + 6 + 3 + 7) / 4 = 16 / 4 = 4$.

As with any non-preemptive algorithm, Average RT = Average WT = 4.

Shortest Remaining Time Next (SRTN), the *preemptive* counterpart, re-evaluates its choice every time a new process arrives: if the newly arrived process's burst time is shorter than the *remaining* burst time of the process currently running, the running process is immediately preempted (returned to the ready queue with its remaining burst time recorded) and the new arrival is dispatched instead.

IMPORTANT — Definition — SRTN (Preemptive SJF)

Shortest Remaining Time Next (SRTN) is the preemptive version of SJF: whenever a new process arrives, its burst time is compared against the *remaining* burst time of the currently running process, and the CPU is switched to whichever process, running or newly arrived, now has the least remaining work.

Worked example (SRTN, preemptive), using the same processes as above.

At $t = 0$, P1 begins (nothing else has arrived yet).

At $t = 2$, P2 arrives with BT 4; P1's remaining burst is $7 - 2 = 5$, which is greater than P2's 4, so P1 is preempted and P2 runs.

At $t = 4$, P3 arrives with BT 1; P2's remaining burst is $4 - 2 = 2$, greater than P3's 1, so P2 is preempted and P3 runs. P3 needs only 1 unit and completes at $t = 5$.

At $t = 5$, P4 also arrives with BT 4; comparing everyone now waiting — P1 (remaining 5), P2 (remaining 2), P4 (remaining 4, brand new) — P2 has the least remaining work, so P2 resumes and runs to completion at $t = 7$.

At $t = 7$, only P1 (remaining 5) and P4 (remaining 4) are left; P4 is shorter, so P4 runs to completion at $t = 11$. Finally, P1, the only process left, runs its remaining 5 units, 11-16.

0 2 4 5 7 11 16

P1	P2	P3	P2	P4	P1
----	----	----	----	----	----

Table 1.9 — SRTN worked example (same input as Table 1.8)

Process	AT	BT	CT	TAT	WT	RT
P1	0	7	16	16	9	0
P2	2	4	7	5	1	0
P3	4	1	5	1	0	0
P4	5	4	11	6	2	2

Average TAT = $(16 + 5 + 1 + 6) / 4 = 28 / 4 = 7$.

Average WT = $(9 + 1 + 0 + 2) / 4 = 12 / 4 = 3$.

Average RT = $(0 + 0 + 0 + 2) / 4 = 2 / 4 = 0.5$.

Notice that, on *identical* input, SRTN achieves a lower average waiting time (3, versus 4 for non-preemptive SJF) and a much lower average response time, precisely because it can react to a short process the moment it arrives rather than waiting for the current process to finish.

Advantages (both forms): SJF/SRTN is provably **optimal for minimising average waiting time** among all scheduling algorithms, for a given, fixed set of processes and known burst times: no other algorithm can do better on that specific metric.

Limitations: the algorithm requires knowing each process's CPU burst time *before* it runs, which is rarely possible exactly in a real, general-purpose system (the OS can only estimate future bursts from past behaviour); and, more seriously, a continuous stream of short processes can indefinitely delay a long process, a problem known as **starvation**. In the SRTN example above, notice how P1, the very first process to arrive, is the *last* to finish, because shorter processes kept being preferred over it. Section 1.8.5 (HRRN) exists specifically to address this starvation risk.

1.8.4 Round Robin (RR)

Round Robin is the classic scheduling algorithm for time-sharing and interactive systems (Section 1.2.3). Every ready process is placed in a circular queue, and each is given the CPU for at most a

fixed **time quantum**; if a process has not finished by the time its quantum expires, it is preempted and placed at the back of the ready queue, and the process now at the front is dispatched next.

IMPORTANT — Definition — Round Robin

Round Robin (RR) scheduling assigns the CPU to each ready process in turn for a fixed time slice, the **quantum**; a process that does not complete within its quantum is preempted and moved to the back of a circular ready queue to await its next turn.

The choice of quantum size is the single most important tuning parameter of Round Robin. A very large quantum makes RR behave almost exactly like FCFS (each process tends to finish, or nearly finish, within its first turn), reintroducing the convoy effect; a very small quantum makes context-switch overhead dominate, since the OS spends a large fraction of its time switching between processes rather than running them, even though each individual process is served very frequently. If there are n processes in the ready queue and the quantum is q , then, in the steady state, no process waits longer than roughly $(n - 1) * q$ time units for its next turn. This is the essential property that makes RR fair and gives it bounded, predictable response time for interactive use.

Worked example. Quantum $q = 2$.

Process	AT	BT
P1	0	5
P2	1	4
P3	2	2
P4	4	1

Tracing the ready queue turn by turn: $t=0$, only P1 is ready; it runs 0-2 (remaining 3) and is requeued behind P2 (which arrived at $t=1$) and P3 (which arrives exactly at $t=2$, added just before P1 is requeued): queue becomes [P2, P3, P1(3)]. P2 runs 2-4 (remaining 2); P4 arrives at $t=4$ and is added before P2 is requeued: queue becomes [P3, P1(3), P4, P2(2)]. P3 runs 4-6 and, needing exactly 2 units, completes. P1 runs 6-8 (remaining $3-2=1$), requeued: [P4, P2(2), P1(1)]. P4 runs 8-9 and, needing only 1 unit, completes. P2 runs 9-11 and, needing exactly its remaining 2 units, completes. Finally, P1 runs its last remaining unit, 11-12, and completes.

0 2 4 6 8 9 11 12

P1	P2	P3	P1	P4	P2	P1
----	----	----	----	----	----	----

Table 1.10 — Round Robin worked example, quantum = 2

Process	AT	BT	CT	TAT	WT	RT
P1	0	5	12	12	7	0
P2	1	4	11	10	6	1
P3	2	2	6	4	2	2
P4	4	1	9	5	4	4

Average TAT = $(12 + 10 + 4 + 5) / 4 = 31 / 4 = 7.75$.

Average WT = $(7 + 6 + 2 + 4) / 4 = 19 / 4 = 4.75$.

Average RT = $(0 + 1 + 2 + 4) / 4 = 7 / 4 = 1.75$.

Response time here is noticeably better than FCFS achieves on similar workloads, precisely because RR guarantees every process a turn within one cycle of the queue rather than making it wait behind every earlier arrival's *entire* burst.

Advantages: fair in a strong, quantitative sense: every ready process is guaranteed a turn within a bounded time, regardless of how long other processes need; simple to implement; requires no advance knowledge of burst times, unlike SJF/SRTN.

Limitations: average waiting time can be worse than SJF/SRTN for a given workload, since RR pays no attention at all to how much work a process actually needs; performance is highly sensitive to the choice of quantum, as discussed above; every preemption, even one that saves almost no time, costs a full context switch, so a small quantum trades responsiveness for raw throughput.

1.8.5 Highest Response Ratio Next (HRRN)

SJF and SRTN can starve long processes indefinitely if short processes keep arriving; **HRRN** is a non-preemptive algorithm specifically designed to prevent this, without needing to preempt anything, by letting a process's *priority for selection* grow the longer it has already waited.

IMPORTANT — Definition — HRRN Scheduling

Highest Response Ratio Next (HRRN) scheduling is non-preemptive: whenever the CPU becomes free, it computes, for every process currently waiting, a **response ratio** $R = (W + S) / S$, where W is the time, the process has already spent waiting and S is its (total) burst time, and dispatches the process with the highest ratio. A process with a small burst time naturally tends to have a high ratio (behaving somewhat like SJF), but a process that has been waiting a long time also sees its ratio climb even if its burst is long, guaranteeing it eventually becomes the most attractive choice.

Worked example.

Process	AT	BT
P1	0	3
P2	1	6
P3	2	4
P4	3	5

At $t = 0$, only P1 has arrived, so it runs 0-3 (no choice is possible yet).

At $t = 3$, P2, P3, and P4 have all arrived and are waiting; their response ratios are computed as $R = (W + S) / S$ using $W = 3 - AT$:

- P2: $W = 3 - 1 = 2$, so $R = (2 + 6) / 6 = 8 / 6 = 1.333$
- P3: $W = 3 - 2 = 1$, so $R = (1 + 4) / 4 = 5 / 4 = 1.25$
- P4: $W = 3 - 3 = 0$, so $R = (0 + 5) / 5 = 5 / 5 = 1.0$

P2 has the highest ratio and runs 3-9.

At $t = 9$, P3 and P4 remain: P3's $W = 9 - 2 = 7$, $R = (7+4)/4 = 11/4 = 2.75$; P4's $W = 9 - 3 = 6$, $R = (6+5)/5 = 11/5 = 2.2$. P3 wins and runs 9-13. Only P4 is left, running 13-18.

0 3 9 13 18

P1	P2	P3	P4
----	----	----	----

Table 1.11 — HRRN worked example

Process	AT	BT	CT	TAT	WT	RT
P1	0	3	3	3	0	0
P2	1	6	9	8	2	2
P3	2	4	13	11	7	7

Process	AT	BT	CT	TAT	WT	RT
P4	3	5	18	15	10	10

Average TAT = $(3 + 8 + 11 + 15) / 4 = 37 / 4 = 9.25$.

Average WT = $(0 + 2 + 7 + 10) / 4 = 19 / 4 = 4.75$.

As HRRN is non-preemptive, Average RT = Average WT = 4.75.

Observe that even though P4 starts with a response ratio of exactly 1.0 (the lowest possible, since $W = 0$) and is passed over at $t = 3$, its ratio keeps climbing every unit of time it continues to wait. This is precisely the mechanism that guarantees no process, however long its burst, waits forever.

Advantages: generally, outperforms plain SJF on average waiting time when waiting times are factored in, because it gives some credit to processes that have already waited; systematically prevents starvation without needing preemption at all, since a long-waiting process's ratio eventually becomes the highest in the system; tends to improve throughput by still favouring short jobs when nothing has been waiting long.

Limitations: like SJF, it needs to know (or estimate) each process's burst time S in advance, which is often impractical in a real, general-purpose OS; computing every waiting process's response ratio every time the CPU frees up is more computational overhead than the simple queue lookups FCFS or RR require.

1.8.6 Multilevel Feedback Queue (MLFQ) Scheduling

All the algorithms above assume the scheduler somehow knows each process's total or remaining burst time. **Multilevel Feedback Queue (MLFQ)** scheduling is designed for exactly the realistic case where it does *not* know this in advance, yet still wants to approximate the good average-waiting-time behaviour of SJF/SRTN by *learning*, from a process's own observed behaviour, whether it looks CPU-bound or I/O-bound (short-burst).

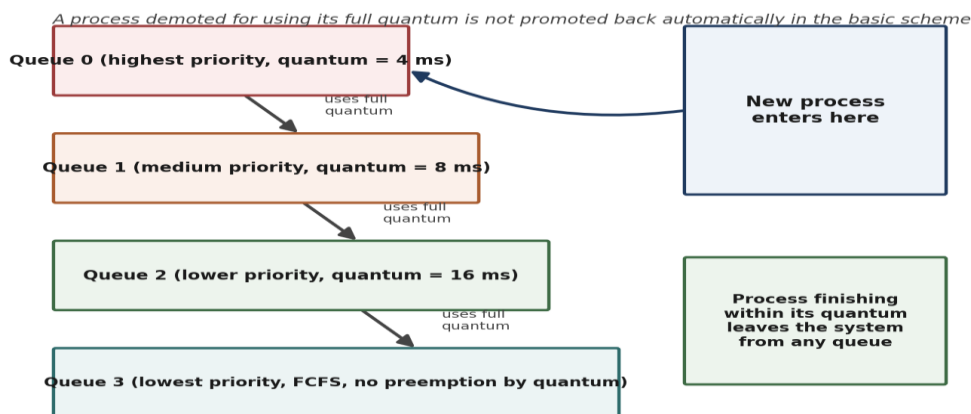


Figure 1.4 — Multilevel feedback queue structure

IMPORTANT — Definition — Multilevel Feedback Queue

A **Multilevel Feedback Queue** scheduler maintains several ready queues arranged in priority order, each with its own scheduling rule (commonly Round Robin with an increasingly generous quantum at lower priority levels, or plain FCFS at the lowest level). A new process enters the highest-priority queue. If it uses its *entire* time quantum without finishing, it is judged to be CPU-intensive and is **demoted** to the next lower-priority queue (which typically grants a longer quantum, since long jobs are given bigger, less-frequent slices rather than many small interruptions); if it finishes, or voluntarily gives up the CPU, before its quantum expires, it is treated as short or I/O-bound and is allowed to stay at (or, in some variants, be promoted back

toward) its current priority. Whenever a higher-priority queue holds a ready process, it always preempts the CPU away from anything running in a lower-priority queue.

Worked example. Three queues: Q0 (highest priority, Round Robin, quantum 2), Q1 (Round Robin, quantum 4), Q2 (lowest priority, plain FCFS, run to completion). All three processes below arrive at $t = 0$ and enter Q0.

Process	AT	BT
P1	0	3
P2	0	6
P3	0	9

Q0 serves P1, P2, P3 in that order, each for up to 2 units. P1 runs 0-2 (used its full quantum, still has $3-2=1$ remaining, so it is *demoted* to Q1). P2 runs 2-4 (remaining $6-2=4$, demoted to Q1). P3 runs 4-6 (remaining $9-2=7$, demoted to Q1). Q0 is now empty, so the scheduler moves to Q1, where the quantum is 4. P1 (remaining 1) runs 6-7 and, needing less than a full quantum, *completes* — no further demotion, since it finished. P2 (remaining 4) runs 7-11 and, needing exactly its remaining time, also completes precisely as its quantum ends. P3 (remaining 7) runs 11-15, uses its *entire* Q1 quantum without finishing (remaining $7-4=3$), and is demoted again, to Q2. Q2 is FCFS, so P3 simply runs to completion: 15-18.

0	2	4	6	7	11	15	18
P1	P2	P3	P1	P2	P3	P3	

Table 1.12 — MLFQ worked example

Process	AT	BT	CT	TAT	WT	RT	Queue path
P1	0	3	7	7	4	0	Q0 -> Q1 (completes)
P2	0	6	11	11	5	2	Q0 -> Q1 (completes)
P3	0	9	18	18	9	4	Q0 -> Q1 -> Q2 (completes)

Average TAT = $(7 + 11 + 18) / 3 = 36 / 3 = 12$.

Average WT = $(4 + 5 + 9) / 3 = 18 / 3 = 6$.

Average RT = $(0 + 2 + 4) / 3 = 6 / 3 = 2$.

Notice the key result the example is meant to illustrate: P1, the shortest job, finishes first (at $t = 7$), and P3, the longest job, finishes last and is the only one demoted all the way to the FCFS queue. MLFQ has, in effect, *discovered* which process was shortest purely by observing that it finished within a quantum, without ever being told any burst time in advance.

Advantages: the most flexible scheduling algorithm covered in this unit, since its many parameters (number of queues, each queue's quantum, its promotion/demotion rule) can be tuned to a specific system's needs; approximates the good response-time behaviour of SJF/SRTN for short, interactive jobs without needing to know burst times ahead of time; naturally favours I/O-bound and interactive processes (which tend to give up the CPU before their quantum expires and so are never demoted) while still eventually servicing CPU-bound processes at a lower priority instead of starving them outright.

Limitations: by far the most complex algorithm to implement and tune correctly among those in this unit; a poorly chosen set of parameters can behave badly (too few queues loses the fine-grained separation between job types; badly chosen quanta can reproduce the very problems, like the convoy effect, MLFQ was meant to avoid); because a process's priority is determined entirely by the system's own observed rules rather than by an administrator or user request, MLFQ offers no direct way to express "this specific process is important" the way a priority-scheduling scheme can.

CHECK YOUR PROGRESS 1.8

1. Define FCFS scheduling and explain what the "convoy effect" is.
2. What is the one difference between non-preemptive SJF and preemptive SRTN? Using the definition of response ratio, explain why HRRN cannot starve a long process indefinitely.
3. In Round Robin, what happens to system performance if the time quantum is set far too large, and what happens if it is set far too small?
4. In the MLFQ worked example, why was P3 eventually moved into Q2 while P1 and P2 were not?
5. Compute, for the FCFS worked example in this section, the waiting time of P4 and show the arithmetic.

1.9 PERFORMANCE EVALUATION OF SCHEDULING ALGORITHMS

1.9.1 Standard Evaluation Criteria

No single number captures whether a scheduling algorithm is "good." Different systems, and different kinds of workload, care about different things, sometimes in direct conflict with one another. Six standard criteria are used, in every OS textbook and in practice, to evaluate and compare scheduling algorithms:

Table 1.13 — Standard criteria for evaluating CPU scheduling

Criterion	What it measures	Usual goal
CPU utilisation	The percentage of time the CPU is actually executing a process rather than sitting idle	Maximise
Throughput	The number of processes completed per unit of time	Maximise
Turnaround time (TAT)	The total time from a process's arrival to its completion ($CT - AT$)	Minimise
Waiting time (WT)	The total time a process spends sitting in the ready queue, not running ($TAT - BT$)	Minimise
Response time (RT)	The time from a process's arrival until it <i>first</i> gets the CPU	Minimise (especially for interactive systems)
Fairness	How evenly the CPU is shared among competing processes, and whether any process is starved indefinitely	Maximise (or: avoid extreme unfairness/starvation)

These criteria matter to different audiences for different reasons. A batch-processing operator, billed by throughput, cares most about CPU utilisation and throughput. An interactive user typing a command cares almost entirely about response time: a system can have excellent throughput and still feel unusable if every keystroke takes a second to echo. A process that submits one long batch job cares about its own turnaround time, and, more specifically, whether it is treated fairly relative to other jobs sharing the machine. Crucially, several of these goals actively conflict: minimising average waiting time (which SJF/SRTN achieves optimally, Section 1.8.3) can come at the direct cost of fairness, since it is precisely optimal *because* it happily makes some processes (the long ones) wait far longer than others; conversely, an algorithm engineered to be maximally fair (strict Round Robin) generally does not achieve the lowest possible average waiting time. There is no algorithm that is simultaneously best on every criterion at once. Every scheduling policy is, explicitly or implicitly, a choice about which of these criteria matters most for the system it is designed for.

1.9.2 Comparative Analysis of the Five Algorithms

Table 1.14 — Trade-offs among the five scheduling algorithms

Algorithm	Preemptive?	Needs burst time known in advance?	Avg. waiting time (relative)	Response time	Starvation risk	Overhead	Best suited for
FCFS	No	No	Can be high (convoy effect)	Poor for late arrivals behind a long job	Low (everyone eventually runs, but can wait a very long time)	Very low (simple queue)	Batch systems with no interactivity requirement
SJF (non-preemptive)	No	Yes	Optimal for non-preemptive algorithms	Good for short jobs, poor for long ones	High for long jobs	Low, but must estimate/know burst times	Batch systems where burst times are known or reliably estimable
SRTN (preemptive SJF)	Yes	Yes	Optimal overall (lowest possible)	Excellent for short jobs	High for long jobs (worse than non-preemptive SJF, since short jobs keep cutting in)	Higher (frequent preemption/context switching)	Systems where minimising average waiting time matters more than fairness
Round Robin	Yes	No	Moderate; depends heavily on quantum	Bounded and predictable for all processes	None (every process gets a bounded-wait turn)	Depends on quantum; can be high if quantum is too small	Interactive and time-sharing systems
HRRN	No	Yes	Better than plain SJF once waiting is counted	Moderate; long jobs eventually served	None (ratio guarantees eventual selection)	Moderate (ratio recomputed at each choice point)	Batch/interactive mixes needing both efficiency and fairness, without preemption
MLFQ	Yes	No	Approaches SJF-like performance without needing burst times	Excellent for short/interactive jobs	Low to moderate (long jobs eventually run at lowest level, though slower)	Highest (most bookkeeping: multiple queues, promotion/demotion tracking)	General-purpose systems mixing interactive and batch/CPU-bound workloads

A few specific comparative points are worth stating explicitly, since they are common exam questions. FCFS is simple but suffers from the convoy effect: a single long process at the front of the queue delays every later, possibly much shorter, process behind it, dragging down average waiting time. Round Robin fixes the *fairness* problem FCFS has, guaranteeing every process a turn within a bounded time, but at the cost of average waiting time that is typically worse than SJF/SRTN would achieve on the same workload; its performance is also extremely sensitive to the quantum, degenerating toward FCFS-like behaviour if the quantum is too large and toward excessive context-switch overhead if it is too small. HRRN can be understood as a *fix for SJF/SRTN's starvation*

problem: it keeps most of SJF's benefit (favouring short jobs) while guaranteeing, through the response-ratio formula, that a job's attractiveness rises the longer it waits, so no process is neglected forever, and it achieves this without any preemption at all. MLFQ, finally, can be understood as an attempt to *approximate* SJF/SRTN's excellent average-waiting-time behaviour in the realistic case where the OS does not, and cannot, know a process's burst time in advance: instead of being told which jobs are short, MLFQ *discovers* this by observing which processes finish within a single quantum, and rewards them with continued high priority, while processes that repeatedly use their full quantum are inferred to be long-running and are progressively demoted.

1.9.3 Choosing an Algorithm for a Given Workload

Because no algorithm dominates on every criterion, the "right" choice depends on the kind of system being built:

Table 1.15 — Matching workload type to a scheduling approach

System type	Priorities	Typically favoured approach
Batch processing, no interactive users	Throughput, CPU utilisation, turnaround time	FCFS (if simplicity matters most) or SJF/HRRN (if burst times are known and average turnaround matters more)
Interactive / time-sharing desktop or server	Response time, fairness among users	Round Robin, or a multilevel feedback queue in practice (most general-purpose OS schedulers, including those in Linux and Windows, are MLFQ-like in spirit)
Real-time (hard or soft)	Deterministic, bounded worst-case latency for high-priority tasks	Fixed-priority preemptive scheduling (a specialised relative of priority scheduling, tuned for guaranteed deadlines rather than average-case fairness)
Mixed general-purpose workload (some interactive, some batch/CPU-bound)	Good response time for interactive work without starving batch work	Multilevel feedback queue, which is precisely designed for this mixed case

CHECK YOUR PROGRESS 1.9

1. List the six standard criteria used to evaluate a scheduling algorithm, and state whether each is normally maximised or minimised.
2. Explain, with reference to SJF/SRTN and Round Robin, why "minimise average waiting time" and "be fair to every process" can be conflicting goals.
3. Why is SRTN's average waiting time never worse than non-preemptive SJF's, on the same input?
4. Which scheduling algorithm in this unit is specifically designed to prevent the starvation that SJF/SRTN can cause, and how does it achieve this without using preemption?
5. Why do most modern general-purpose operating systems use an MLFQ-like scheduler rather than pure FCFS, pure SJF, or pure Round Robin?

1.10 SUMMARY

An operating system sits between the user, application programs, and the raw hardware, making the machine easier to use while keeping its hardware efficient. What it can do has always depended on the hardware it ran on: vacuum-tube machines had no OS at all, transistorised machines introduced batch processing, integrated-circuit machines had enough memory for multiprogramming and time-sharing, microprocessors brought personal and graphical desktop computing, and today's systems are distributed, cloud-based, and increasingly AI-assisted.

We told apart five types of operating system by asking what each one switches between. Multiprogramming switches whole processes so the CPU stays busy during I/O waits. Multitasking

switches processes fast enough that one user feels several things are running at once. Timesharing does the same thing across many users instead of one. Multithreading switches between threads inside a single process, which costs less than switching whole processes. Real-time systems aren't about switching at all; they're built around meeting deadlines, strict or best-effort.

The same OS looks different depending on who's looking at it: a user sees an interface, a system programmer sees a resource manager, an applications programmer sees an API, and a kernel developer sees internal structure. That structure can be monolithic (everything in one address space, fast but fragile), layered (strict hierarchy, easier to debug but slower), microkernel (most services pushed out to user space for isolation), or modular, which is what most systems actually run today, blending a monolithic core with loadable pieces.

A process is a program actually running, not just sitting on disk, and every process moves through the same states: new, ready, running, waiting, terminated. Everything the OS knows about a process, its registers, program counter, memory info, scheduling data, lives in one structure, the Process Control Block. Creating, scheduling, or killing a process really just means updating its PCB, and all of it reaches the application only through system calls.

Deciding who gets the CPU next is scheduling, and we worked through five algorithms by hand. FCFS is simple but suffers from the convoy effect. SJF and its preemptive cousin SRTN give the best average waiting time but need to know burst times in advance and can starve long jobs. Round Robin is fair to everyone through a fixed time slice, though its performance depends heavily on getting that slice size right. HRRN fixes SJF's starvation problem by letting a process's priority climb the longer it waits. MLFQ does something cleverer: it watches how a process actually behaves and adjusts its queue accordingly, without needing to know burst times up front.

No algorithm wins on every measure; we compared them on CPU utilisation, throughput, turnaround time, waiting time, response time, and fairness, and each one trades some of these off against the others. The thread running through this whole chapter is simple: an OS exists to share the CPU, memory, and other resources fairly among competing processes, and scheduling is exactly how it decides who gets the CPU when more than one process wants it.

1.11 GLOSSARY

Term	Definition
Batch processing	Running a queue of jobs automatically, one after another, with no operator intervention between them
Burst time (BT)	The amount of CPU time a process needs to complete a given burst of computation
Context switch	The act of saving a running process's execution state and loading another process's previously saved state, so the CPU can switch between them
Convoy effect	A performance problem in which a long process at the head of a queue forces many shorter processes to wait behind it
CPU scheduler	The OS component that selects which ready process gets the CPU next
CPU utilisation	The percentage of time the CPU is executing a process rather than sitting idle
Dispatcher	The OS component that performs the actual context switch chosen by the scheduler
Distributed operating system	An operating system whose services are coordinated across multiple networked machines rather than confined to one
Fairness	A scheduling criterion concerned with whether the CPU is shared evenly, and whether any process is starved
FCFS (First Come First Serve)	A non-preemptive scheduling algorithm that serves processes strictly in arrival order
HRRN (Highest Response Ratio Next)	A non-preemptive scheduling algorithm that selects the waiting process with the highest response ratio $(W+S)/S$, preventing starvation without preemption
Kernel	The core part of an operating system that is always resident and running, supervising every other program

Term	Definition
Layered operating system	An OS structured as a strict hierarchy of layers, each built only from the services of the layer below it
Message passing	An IPC mechanism in which the kernel copies data between processes, keeping their memory isolated
Microkernel	A minimal, privileged kernel core that pushes most OS services out into separate user-space processes
MLFQ (Multilevel Feedback Queue)	A scheduling algorithm using several priority queues, demoting a process to a lower-priority queue if it uses a full time quantum without finishing
Monolithic kernel	An OS structured as a single large program running entirely in one address space
Multiprogramming	Keeping several programs resident in memory and switching the CPU to a different one whenever the current one blocks on I/O
Multitasking	Rapidly switching the CPU among processes so that they appear to an interactive user to run simultaneously
Multithreading	Allowing multiple threads, sharing the same process's memory, to execute concurrently
PCB (Process Control Block)	The data structure that stores all information the OS needs to manage a specific process
Preemptive scheduling	Scheduling in which the OS can forcibly remove the CPU from a running process before it finishes or blocks
Priority ratio / response ratio	In HRRN, the value $(w+s)/s$ used to rank waiting processes for selection
Process	A program in execution, together with its program counter, registers, stack, and data
Process state	One of New, Ready, Running, Waiting, or Terminated, describing what the OS is currently doing with a process
Program	A passive file of executable instructions stored on disk
Quantum	The fixed time slice a process is allowed to run before being preempted in Round Robin (or a given MLFQ level)
Ready queue	The set of processes that are ready to run and are waiting only for the CPU
Response time (RT)	The time from a process's arrival until it first receives the CPU
Round Robin (RR)	A preemptive scheduling algorithm that gives every ready process, in turn, a fixed time quantum
RTOS (Real-Time Operating System)	An operating system built to guarantee that tasks meet strict (hard) or best-effort (soft) timing deadlines
SJF (Shortest Job First)	A non-preemptive scheduling algorithm that selects the ready process with the smallest total burst time
SRTN (Shortest Remaining Time Next)	The preemptive version of SJF, which can interrupt a running process if a shorter-remaining-time process arrives
Starvation	A situation in which a process is perpetually passed over by the scheduler and never gets the CPU
System call	A controlled entry point through which a user-mode program requests a service from the kernel
Thread	The smallest unit of CPU scheduling; a single sequential flow of control that shares memory with other threads of the same process
Throughput	The number of processes completed per unit of time
Time-sharing	Dividing CPU time into slices shared among multiple simultaneous interactive users
Turnaround time (TAT)	The total time from a process's arrival to its completion ($CT - AT$)
Waiting time (WT)	The total time a process spends in the ready queue, not running ($TAT - BT$)

CHAPTER 2

Memory Management

2.1 MEMORY MANAGEMENT WITHOUT SWAPPING OR PAGING

Every running program needs somewhere to keep its instructions and data while the CPU executes it, and that "somewhere" is main memory (RAM). **Memory management** is the part of the operating system responsible for deciding which parts of memory are in use, by whom, and for granting, tracking, and reclaiming memory as processes are created and destroyed. The schemes covered in this section share one defining restriction: each process, once loaded, occupies a single, complete, contiguous block of physical memory for its entire lifetime: there is no mechanism for running a process that is only partially resident in RAM, and no automatic transfer of memory contents to and from disk while a process runs. These are the simplest possible memory-management strategies, and although real systems moved beyond them decades ago, understanding their strengths and (especially) their failure modes is what motivates every more sophisticated scheme covered later in this unit.

2.1.1 Monoprogramming Without Swapping or Paging

The simplest possible arrangement, historically the first to exist, is **monoprogramming**: at any instant, only one user program occupies memory alongside the operating system itself. A common layout places the operating system permanently in low memory (or in read-only memory, on some very early or embedded designs) and the single user program in the remaining space above it; some systems instead placed device drivers in high memory, in ROM, sandwiching the user program between the OS below and the drivers above. Because only one program is ever present, there is no need for relocation hardware (the program can simply always be loaded at the same fixed address) and no need for protection between user programs (there is only ever one). The entire machine, including all of memory above the resident OS, belongs to that one program until it finishes or is terminated.

Monoprogramming is easy to implement and has essentially zero memory-management overhead, but it is extremely wasteful of the CPU: whenever the running program performs I/O (reading a file, waiting for a keystroke, writing to a printer), the CPU has nothing else to do and sits idle, because no other program is resident to be scheduled in its place. On early machines, where I/O devices were slow relative to the CPU, this idle time could dominate total execution time. This single limitation, an idle CPU during every I/O wait, is precisely what motivated **multiprogramming**: keeping more than one process in memory at once so that the CPU can be switched to a different, ready process whenever the current one blocks on I/O. Multiprogramming, in turn, immediately raises a question monoprogramming never had to answer: since several programs must now share the same physical memory, exactly how is memory divided among them?

2.1.2 Fixed Partitioning

The most direct way to give multiprogramming a place to put multiple processes is **fixed partitioning** (also called static partitioning): main memory is divided, once, into a fixed number of partitions, whose sizes and boundaries are decided at system-generation or boot time and do not change afterwards. Each partition can hold exactly one process; a process is loaded into whichever partition is free and large enough for it, and it remains in that partition until it terminates.

Two well-known variants exist. In **equal-size fixed partitioning**, every partition is the same size, which is simplest to manage but forces every process, regardless of how small it actually is, to occupy an entire partition, and rejects outright any process larger than the (single) partition size. In **unequal-size fixed partitioning**, partitions of several different sizes are created up front (small, medium, and large partitions, say), and an incoming process is placed into the smallest available

partition that is still large enough to hold it, which reduces waste somewhat by letting small jobs occupy small partitions rather than large ones. Either variant typically maintains one input queue per partition (a job waits for "its" size class to become free) or, in a slightly more flexible design, one common queue from which any waiting job is assigned to the first suitable free partition. The common-queue design keeps a large partition from sitting idle while jobs are still queued only for a smaller one.

Fixed partitioning is simple to implement, requires only a small, static partition table maintained by the OS, and each partition can be protected from the others using a pair of bounds (base and limit) registers, described in Section 2.1.5. Its central weakness is that memory allocated to a partition but not used by the process it holds is wasted for as long as that process occupies the partition. This waste is called **internal fragmentation**, examined numerically in Section 2.1.4. A second, structural weakness is that the number of partitions fixes an upper limit on the degree of multiprogramming: if all partitions are occupied, no new process can be admitted no matter how much total free space those partitions happen to contain, and a program larger than the largest available partition simply cannot run at all, however much *total* memory the machine has.

2.1.3 Variable Partitioning and Contiguous Allocation

Variable partitioning (dynamic partitioning) removes the fixed boundaries: partitions are created on the fly, each sized exactly to the process it will hold, at the moment that process is loaded. Because a partition is carved out of memory to match its process precisely, variable partitioning eliminates internal fragmentation almost entirely: a process is given exactly the bytes it needs, no more.

Regardless of whether partitioning is fixed or variable, allocating memory to a process this way is called **contiguous memory allocation**: the process's entire address space, code, data, and stack together, must fit into one unbroken run of physical memory addresses. This is what makes address translation trivial under these schemes: the hardware need only add a single base address (Section 2.1.5) to every address the program generates.

Under variable partitioning, the operating system must actively track which regions of memory are free (called **holes**) and which are occupied, typically with a linked list or a bitmap recording the start address and length of every free hole. When a new process arrives, the OS must choose *which* hole to place it in, using one of several classic placement strategies:

- **First fit.** Scan the list of holes from the beginning and allocate the process to the first hole that is large enough. Fast, and the simplest to implement.
- **Best fit.** Scan the *entire* list and choose the smallest hole that is still large enough for the process, minimising the leftover sliver of unused space in that hole. Tends to leave behind many tiny, largely useless holes over time.
- **Worst fit.** Scan the entire list and choose the *largest* available hole, on the reasoning that the leftover remainder will itself be large enough to be useful for a future process. In practice this tends to perform worse than first fit or best fit.
- **Next fit.** Like first fit, but resumes scanning from the point in the hole list where the previous search left off rather than always restarting from the beginning, which spreads allocations more evenly across memory.

Whichever strategy is used, variable partitioning's price for eliminating internal fragmentation is **external fragmentation**: as processes of different sizes are loaded and later terminate, memory becomes checkerboarded with holes of many different (often small) sizes, none of which may be large enough, on its own, to satisfy a new request, even though the holes' sizes might add up, in total, to comfortably more than the new process needs. One OS-level remedy is **compaction**: periodically pausing execution and shuffling the resident processes so that all the free holes are merged into a single large block at one end of memory. Compaction works, but it is expensive (every relocated process's memory must physically be copied, and every one of its internal pointers and

addresses must be adjusted or dynamically relocated) and is not practical to run continuously, so external fragmentation is a genuine, persistent problem for any purely contiguous-allocation scheme, and one of the strongest practical motivations for paging (Section 2.2), which sidesteps external fragmentation altogether by not requiring contiguity at all.

2.1.4 Internal and External Fragmentation: Worked Examples

Internal fragmentation is memory wasted *inside* an allocated block, because the block is larger than what the process actually needed. **External fragmentation** is memory wasted *between* allocated blocks, in holes that are individually too small to be useful even though their combined size is not. It is worth working through one fully numeric example of each, since exam questions on this topic almost always ask for the exact wasted quantities.

EXAMPLE — Internal fragmentation under fixed partitioning

Suppose an operating system uses fixed, equal-size partitions of 8 KB each, and three processes arrive with the following memory requirements:

- Process A requires 5 KB
- Process B requires 7 KB
- Process C requires 4 KB

Because every partition is a full 8 KB regardless of how much of it a process actually uses, each process is given a whole 8 KB partition:

Table 2.1 — Internal fragmentation under 8 KB fixed partitions

Process	Memory requested	Partition size	Memory allocated	Internal fragmentation
Process A	5 KB	8 KB	8 KB	$8 - 5 = 3 \text{ KB}$
Process B	7 KB	8 KB	8 KB	$8 - 7 = 1 \text{ KB}$
Process C	4 KB	8 KB	8 KB	$8 - 4 = 4 \text{ KB}$

The total internal fragmentation across these three partitions is $3 + 1 + 4 = 8 \text{ KB}$: exactly one whole partition's worth of memory is sitting unused inside partitions that appear, from the partition table's point of view, to be fully occupied.

EXAMPLE — External fragmentation under variable partitioning

Suppose a system has 16 KB of RAM, allocated (using variable partitioning, so each process gets exactly the space it asked for) as follows: Process A takes 4 KB, Process B takes 6 KB, and Process C takes 4 KB, leaving 2 KB free at the end. Memory therefore looks like [A: 4 KB] [B: 6 KB] [C: 4 KB] [free: 2 KB].

Now suppose Process B terminates and releases its 6 KB. Memory becomes [A: 4 KB] [free: 6 KB] [C: 4 KB] [free: 2 KB]: two separate free regions, of 6 KB and 2 KB, with Process C's 4 KB sitting physically between them.

Table 2.2 — External fragmentation after Process B terminates

State	Region 1	Region 2	Region 3	Region 4
Before B terminates	A (4 KB, used)	B (6 KB, used)	C (4 KB, used)	free (2 KB)
After B terminates	A (4 KB, used)	free (6 KB)	C (4 KB, used)	free (2 KB)

Total free memory after Process B terminates is $6 + 2 = 8 \text{ KB}$. If a new Process D now arrives needing 8 KB, it cannot be loaded, even though exactly 8 KB of free memory exists in the system, because that 8 KB is split into two non-contiguous holes (6 KB and 2 KB), neither of which is large enough by itself, and contiguous allocation cannot place a process across a gap occupied by Process

C. This is external fragmentation: memory is wasted not because any single block is oversized, but because the free space is scattered into pieces too small individually to satisfy the request.

The two problems are, in a sense, opposites: fixed-size allocation (fixed partitioning, and later, fixed-size paging) trades external fragmentation for internal fragmentation, since every block is a uniform size and cannot be scattered into oddly-shaped leftover holes, but an allocation is rarely an exact fit. Variable-size allocation (variable partitioning, and later, segmentation) does the reverse: it fits each allocation exactly, eliminating internal fragmentation, but leaves behind a scattered, ever-changing pattern of holes that themselves become the wasted space. No scheme based on unmodified, exact-fit-or-fixed-size contiguous allocation avoids both at once; each family of memory-management technique studied in this unit represents a different trade-off between them.

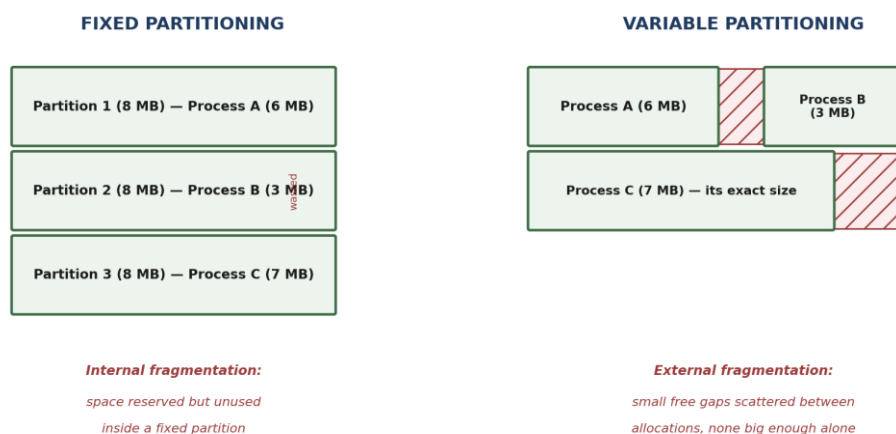


Figure 2.1 — Fixed versus variable partitioning

2.1.5 Relocation and Protection: Base and Limit Registers

Two further problems appear the moment more than one process shares physical memory. First, **relocation**: a program is typically compiled and linked as though it will be loaded starting at address 0, but in a multiprogrammed system it may actually be loaded starting at some other physical address depending on which partition or hole happens to be free at load time. Under variable partitioning, that starting address is not even known until the moment of loading, and could change if the process is later moved (say, by compaction). Second, **protection**: with several independent processes resident at once, one process's bug or malicious code must not be allowed to read or overwrite another process's memory, or the operating system's own memory.

Both problems are solved together, cheaply, with a pair of hardware registers maintained per running process:

IMPORTANT — Definition — Base and Limit Registers

A **base register** holds the starting physical address of the process's partition. A **limit register** holds the size (length) of that partition. For every memory address a running process generates (called a *logical* or *virtual* address, since it is meaningful only relative to that process's own view of its address space), the hardware automatically checks the address against the limit register and, if it passes, adds the base register to it to produce the actual *physical* address sent to the memory unit.

Formally, for a logical address a generated by the CPU: if $a \geq \text{limit}$, the hardware immediately traps to the operating system (a **protection fault**) without ever touching memory, because the

process is attempting to reach outside its own allocated partition; otherwise, the physical address delivered to the memory unit is $\text{physical address} = \text{base} + a$. Because this check and addition happen in hardware, on every single memory reference, they add negligible overhead while providing a hard, un-bypassable guarantee that a process can never construct an address that lands inside another process's partition or inside the operating system's own memory.

Base-and-limit registers also solve relocation automatically: the same program image can be loaded into any partition, at any physical starting address, simply by loading that address into the base register when the process is dispatched. The program's own logical addresses (which it always believes start at 0) never need to be rewritten. Only the two register values, updated by the OS on every context switch, need to change.

EXAMPLE — A relocation-and-protection worked example

Suppose Process P is loaded into a partition with $\text{base} = 20000$ and $\text{limit} = 5000$. The process itself generates logical addresses believing it owns a private address space running from 0 to 4999.

- If the process references logical address 4300: since $4300 < 5000$, the reference is valid, and the hardware computes $\text{physical address} = \text{base} + a = 20000 + 4300 = 24300$.
- If the process references logical address 6000: since $6000 \geq 5000$, the reference lies outside the process's own partition; the hardware traps immediately, the memory access never happens, and the operating system typically terminates the offending process with a protection-violation (segmentation fault) signal.

Notice that the process itself never needed to know it was physically located at address 20000. It only ever dealt with addresses 0 through 4999, and the base register transparently translated every one of them.

Base-and-limit registers are the simplest possible **memory protection and relocation mechanism**, and although modern systems use far richer hardware (multi-level page tables and segment descriptors, covered later in this unit), the fundamental idea, a hardware-enforced boundary check followed by an address translation invisible to the running program, recurs, in more elaborate form, in every memory-management scheme covered in the rest of this unit.

CHECK YOUR PROGRESS 2.1

1. Why does monoprogramming waste CPU time, and what OS technique was introduced specifically to fix this waste?
2. Distinguish fixed partitioning from variable partitioning in terms of when partition boundaries are decided and which type of fragmentation each is more prone to.
3. A system uses 16 KB fixed partitions. Three processes need 10 KB, 6 KB, and 15 KB respectively. Compute the internal fragmentation for each.
4. Explain, with reference to a concrete numeric example, why a system can have enough total free memory to run a new process yet still be unable to load it.
5. What two pieces of information does a base register and a limit register each hold, and how do they together provide both relocation and protection?

2.2 CONCEPTS OF SWAPPING AND PAGING

The schemes of Section 2.1 share a hard limit: the sum of the memory demands of all resident processes can never exceed the size of physical RAM, because every process must be entirely present, in one contiguous block, before it can run at all. Real workloads routinely violate this limit: more processes want to run than can simultaneously fit, or an individual process's own address space is larger than the RAM installed on the machine. **Swapping** and **paging** are the two classical

techniques operating systems use to keep running processes that, together, do not physically fit in RAM at once, and paging in particular is the foundation on which virtual memory (one of the most important ideas in all of operating systems) is built.

2.2.1 Swapping

Swapping is a memory-management technique in which an entire process is moved, as one unit, out of main memory and onto secondary storage (traditionally a hard disk, and on modern systems typically a dedicated swap partition or swap file on an SSD), and later moved back into main memory to continue running. A process currently resident in RAM is said to be **swapped out**; a process later returned to RAM is **swapped in**. The operating system uses secondary storage this way purely as an overflow area (a much larger, if far slower, extension of RAM), so that the total memory demand of processes the system wants to keep alive is not bounded by the amount of physical RAM actually installed.

EXAMPLE — Swapping in action

Suppose a system has 8 MB of RAM. Process A (3 MB) and Process B (2 MB) are resident, leaving 3 MB free. A new Process C, needing 4 MB, now wants to run, but only 3 MB is free, which is not enough.

The operating system chooses a resident process to swap out, say Process B (2 MB), which is currently blocked waiting on I/O and therefore not immediately needed in RAM. Once Process B's memory image has been written out to disk, $2 + 3 = 5$ MB is free, which is enough to swap in Process C's 4 MB, leaving 1 MB free. When Process B is ready to run again, the operating system can swap out some other, less urgently needed process (perhaps Process C, once it in turn becomes idle) to make room, and swap Process B's saved image back into RAM from disk.

Swapping lets a system support more total process memory than it has physical RAM, and it is conceptually very simple: an entire process, unmodified, is copied out and copied back. Its major drawback is speed: disk access is orders of magnitude slower than RAM access, and moving an entire process (which might be many megabytes) takes a proportionally long time, during which that process cannot run at all. A system that swaps processes in and out too aggressively spends most of its time waiting on disk transfers rather than doing useful computation, a problem that becomes acute with **thrashing**, discussed as a design issue in Section 2.5.

2.2.2 Paging: The Basic Idea

Paging attacks the same underlying problem (supporting more total memory demand than physical RAM can hold at once) from a fundamentally different angle: instead of moving whole processes in and out of memory, it moves small, fixed-size *pieces* of a process's address space, and it never requires a process's memory to be contiguous in the first place.

IMPORTANT — Definition — Paging

Paging is a memory-management scheme that divides a process's logical address space into fixed-size blocks called **pages**, and divides physical memory into blocks of the *same* fixed size called **frames**. A process's pages need not occupy contiguous frames, and not all of a process's pages need be resident in physical memory at the same time; the operating system maintains, per process, a **page table** that records, for every page, which physical frame (if any) currently holds it.

Because pages and frames are the same fixed size, any free frame can hold any page: there is no need to search for a hole of a particular size, and, since a page always occupies exactly one whole frame, there is no external fragmentation whatsoever (a free frame is either entirely free or entirely

occupied; there are never awkward, oddly-sized leftover slivers of physical memory). The only fragmentation paging can suffer from is a small amount of *internal* fragmentation in the very last page of a process, if the process's size is not an exact multiple of the page size. On average, this wastes only half a page per process, which is far less severe than the fragmentation problems of Section 2.1.4.

Paging also breaks the requirement, central to every scheme in Section 2.1, that a process must be *entirely* resident before it can run: because address translation happens per-page, a process can begin executing as soon as the pages it is actually using are in memory, while its remaining pages stay on disk until (or unless) they are actually referenced. This is the essential mechanism behind **virtual memory**: a process can behave as though it has access to an address space far larger than physical RAM, with the operating system transparently fetching pages from disk on demand and evicting other pages to make room, a process examined in depth through page-replacement algorithms in Section 2.3.

2.2.3 The Page Table and Address Translation

Every logical address a process generates is interpreted by the paging hardware as two components: a **page number**, which selects an entry in the process's page table, and a **page offset**, which is the position of the referenced byte *within* that page (and, since pages and frames are the same size, also its position within whichever frame the page currently occupies). If the page size is 2^d bytes, the low-order d bits of the logical address are always the offset, and the remaining, higher-order bits are always the page number. This is precisely why page sizes are chosen to be powers of two: the split between page number and offset then requires no arithmetic division at all, just extracting two ranges of bits.

The **page table** itself is simply an array, indexed by page number, in which entry p records the frame number currently holding page p of the process (together with bookkeeping bits: a *valid/invalid* bit saying whether the page is actually in memory right now, and typically protection bits, discussed in Section 2.5). Translating a logical address into a physical one proceeds in three mechanical steps: split the logical address into a page number and an offset; look up that page number in the page table to obtain a frame number; and combine the frame number with the (unchanged) offset to form the physical address.

EXAMPLE — Worked address translation using a page table

Consider a system with a page size of 4096 bytes ($4\text{ KB} = 2^{12}$, so the offset is the low-order 12 bits of every logical address), a logical address space of 32 KB (8 pages, numbered 0-7), and 16 KB of physical memory (4 frames, numbered 0-3). Suppose the process's page table currently looks like this:

Table 2.3 — Page table for the address-translation example

Page number	Frame number	Valid?
0	2	Yes
1	0	Yes
2	—	No (not in memory)
3	1	Yes
4	3	Yes
5	—	No (not in memory)
6	—	No (not in memory)
7	—	No (not in memory)

Now translate the logical address 13000 (decimal). Since the page size is 4096 bytes, dividing gives the page number and offset directly: $13000 / 4096 = 3$ remainder 712; that is, $3 * 4096 = 12288$, and $13000 - 12288 = 712$. So page number $p = 3$ and offset $d = 712$.

Looking up page 3 in the page table gives frame number $f = 1$. The physical address is then $\text{physical address} = (f * \text{page size}) + d = (1 * 4096) + 712 = 4096 + 712 = 4808$.

So logical address 13000 translates to physical address 4808. If, instead, the process had generated a logical address on page 5, 6, or 7 (marked invalid, not currently in a frame), the hardware would raise a **page fault**, trapping to the operating system so it could bring the required page in from disk before the reference could be completed; page faults, and the algorithms that decide which resident page to evict to make room for the incoming one, are the subject of Section 2.3.

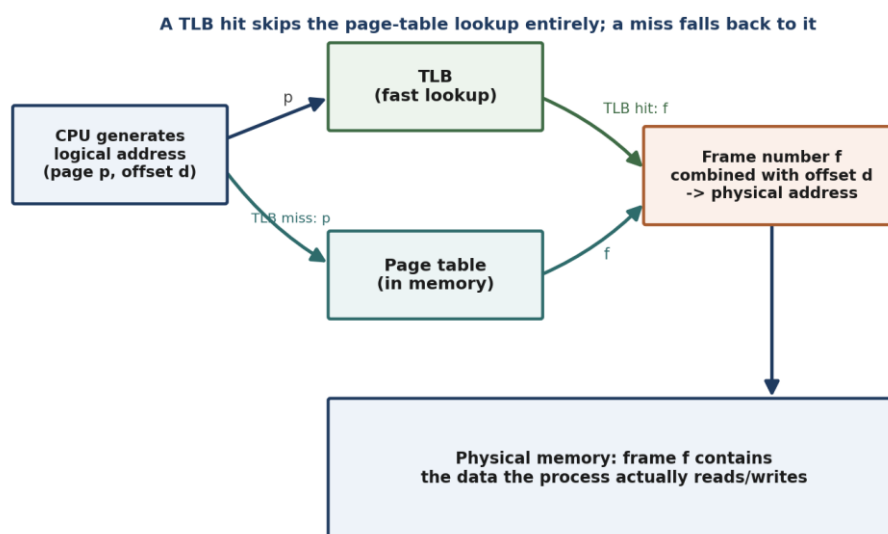


Figure 2.2 — Address translation via a page table and TLB

2.2.4 The Translation Lookaside Buffer (TLB)

The address-translation procedure above requires one extra memory access (the lookup in the page table) *before* the actual memory reference the program wanted can be carried out at all; since the page table itself normally lives in main memory, a naive implementation would double the effective cost of every single memory access. This overhead would make paging far too slow to be practical, so real hardware avoids it with a small, extremely fast, fully associative hardware cache called the **Translation Lookaside Buffer (TLB)**, built directly into the memory-management unit (MMU) alongside the CPU.

The TLB stores a small number of the most recently used page-number-to-frame-number mappings. On every memory reference, the hardware checks the TLB first, in parallel with (or just before) consulting the page table:

- **TLB hit.** The page number is found in the TLB. The corresponding frame number is available immediately, at essentially CPU-cache speed, and the slower page table in main memory is never consulted at all for this reference.
- **TLB miss.** The page number is not in the TLB. The hardware (or, on some architectures, OS software) must look up the mapping in the page table in memory, incurring the extra memory access, and then loads that mapping into the TLB (evicting some existing TLB entry, typically by an LRU-like policy, if the TLB is full) so that a subsequent reference to the same page will hit.

Because most programs exhibit strong **locality of reference**: they tend to access a relatively small set of pages repeatedly over any short interval, both because of loops (temporal locality) and because of sequential access to arrays or instruction streams (spatial locality), the TLB achieves a very high hit ratio in practice (typically well above 90%), which is what makes paging's extra translation step affordable.

EXAMPLE — Effective memory access time with a TLB

Suppose a TLB lookup takes 20 ns, a main-memory access takes 100 ns, and the TLB hit ratio is 80%. On a TLB hit, a reference costs one TLB lookup plus one memory access for the data itself: $20 + 100 = 120$ ns. On a TLB miss, the page table (in memory) must be consulted first, and only then can the actual data be fetched: $20 + 100 + 100 = 220$ ns (TLB lookup, then page-table lookup in memory, then the data access itself).

The **effective access time (EAT)** is the probability-weighted average of these two cases:

$$\text{EAT} = (\text{hit ratio} * \text{hit time}) + (\text{miss ratio} * \text{miss time}) = (0.8 * 120) + (0.2 * 220) = 96 + 44 = 140 \text{ ns}$$

Without any TLB at all, every reference would cost $100 + 100 = 200$ ns (a page-table access followed by the data access); with the TLB and an 80% hit ratio, the effective cost drops to 140 ns, a substantial saving that grows further as the hit ratio improves. This is exactly why hit ratio (driven by locality of reference and TLB size) is such a central performance concern in real processor design.

A further practical wrinkle is that every process has its own page table, so a TLB entry that is valid for one process is meaningless (or actively dangerous) for another; naively, the entire TLB would need to be flushed on every context switch, destroying much of its benefit. Many modern processors avoid this by tagging each TLB entry with an **address-space identifier (ASID)**, so that entries belonging to several different processes can coexist safely in the TLB at once, and a context switch does not force a full flush.

CHECK YOUR PROGRESS 2.2

1. Why does an OS need swapping if paging can already handle a process larger than physical memory? (Hint: think about what unit of data each technique moves.)
2. Define page and frame, and explain why they must be the same fixed size.
3. A system has a page size of 2 KB. Compute the page number and offset for logical address 9000.
4. What is a TLB, and why is it essential for paging to be practical rather than merely possible?
5. In the TLB worked example, if the hit ratio improved from 80% to 95% (holding all other timings the same), would the effective access time increase or decrease? Explain why, without recomputing the exact number.

2.3 PAGE REPLACEMENT ALGORITHMS

When a running process references a page that is not currently in a physical frame, the hardware raises a **page fault**. If a free frame exists, the operating system simply loads the required page into it from disk. But once all frames are occupied (the normal state of affairs on a busy system), bringing in the new page means some *other*, currently resident page must be evicted first, and written back to disk if it has been modified since it was loaded (this is tracked with a per-frame **dirty** or **modified** bit, so that a page which has not been changed need not be written back at all: it can simply be discarded, since an identical copy already exists on disk). A **page-replacement algorithm** is the policy that decides *which* resident page to evict. Since disk access is vastly slower than RAM access, the goal of every page-replacement algorithm is the same: minimise the number

of page faults over the lifetime of a program, because every single fault costs a disk-access delay of many thousands of CPU cycles.

To compare algorithms meaningfully, this section traces all five required algorithms (**FIFO**, **Optimal**, **LRU**, **MRU**, and **Clock (second-chance)**) against the *same* reference string with the *same* number of frames, so that their fault counts can be directly compared. The reference string used throughout is the classic sequence

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

(12 references to 5 distinct pages), with **3 physical frames** available, unless stated otherwise. Every trace below has been verified step by step; a reader working through any trace by hand should arrive at exactly the fault/hit sequence shown.

2.3.1 First In First Out (FIFO)

Selection rule: evict the page that has been resident in memory the *longest*, regardless of how recently or how often it has been used. Pages are tracked in a simple queue, ordered by the time they were loaded, and a fault always evicts the page currently at the front (oldest) of that queue.

Table 2.4 — FIFO trace over the reference string (3 frames)

Step	Reference	Frames after this step (oldest -> newest)	Result
1	1	1	Fault
2	2	1, 2	Fault
3	3	1, 2, 3	Fault
4	4	2, 3, 4 (evict 1)	Fault
5	1	3, 4, 1 (evict 2)	Fault
6	2	4, 1, 2 (evict 3)	Fault
7	5	1, 2, 5 (evict 4)	Fault
8	1	1, 2, 5	Hit
9	2	1, 2, 5	Hit
10	3	2, 5, 3 (evict 1)	Fault
11	4	5, 3, 4 (evict 2)	Fault
12	5	5, 3, 4	Hit

Total: 9 faults, 3 hits. FIFO is simple to implement (a single queue) and requires no extra hardware support beyond load-order bookkeeping, but the trace shows its central weakness plainly: at step 10, page 1 is evicted purely because it happens to be the oldest resident page, even though it was just used two references earlier (step 8). FIFO has no notion of how recently or how often a page has actually been used, only of when it was loaded, so it can and does evict pages that are about to be needed again.

2.3.2 Optimal Page Replacement (OPT / MIN)

Selection rule: when a fault occurs and a frame must be freed, evict the page that will not be referenced again for the *longest time into the future* (or, if some resident page will never be referenced again at all, evict that page in preference to any page that will be used again eventually).

Table 2.5 — Optimal replacement trace (3 frames)

Step	Reference	Frames after this step	Result (eviction reasoning)
1	1	1	Fault
2	2	1, 2	Fault
3	3	1, 2, 3	Fault
4	4	1, 2, 4	Fault — evict 3 (next used at step 10, the farthest away of 1, 2, 3)

Step	Reference	Frames after this step	Result (eviction reasoning)
5	1	1, 2, 4	Hit
6	2	1, 2, 4	Hit
7	5	1, 2, 5	Fault — evict 4 (next used at step 11, farther than 1's step 8 or 2's step 9)
8	1	1, 2, 5	Hit
9	2	1, 2, 5	Hit
10	3	2, 5, 3	Fault — evict 1 (never referenced again in the string)
11	4	5, 3, 4	Fault — evict 2 (never referenced again in the string)
12	5	5, 3, 4	Hit

Total: 7 faults, 5 hits. This is the fewest possible faults achievable by *any* replacement algorithm on this reference string with 3 frames: a mathematical guarantee, not a coincidence of this particular trace. The catch, and the reason Optimal is never actually deployed, is that it requires knowing the *entire future* reference sequence of the program in advance, which is generally impossible while a program is actually running. Optimal is used exclusively as a theoretical yardstick: every practical algorithm's fault count is judged by how close it comes to Optimal's.

2.3.3 Least Recently Used (LRU)

Selection rule: evict the page that has gone the *longest* without being referenced. LRU approximates Optimal by assuming that a page's recent past usage predicts its near future usage (temporal locality), replacing "will not be used for longest" (which requires the future) with "has not been used for longest" (which requires only the past, and is thus actually implementable).

Table 2.6 — LRU trace (3 frames)

Step	Reference	Frames after this step (most- to least-recently used)	Result
1	1	1	Fault
2	2	2, 1	Fault
3	3	3, 2, 1	Fault
4	4	4, 3, 2 (evict 1)	Fault
5	1	1, 4, 3 (evict 2)	Fault
6	2	2, 1, 4 (evict 3)	Fault
7	5	5, 2, 1 (evict 4)	Fault
8	1	1, 5, 2	Hit
9	2	2, 1, 5	Hit
10	3	3, 2, 1 (evict 5)	Fault
11	4	4, 3, 2 (evict 1)	Fault
12	5	5, 4, 3 (evict 2)	Fault

Total: 10 faults, 2 hits. Notably, LRU performs *worse* here than plain FIFO (10 faults versus 9), a useful reminder that this particular reference string was deliberately constructed to expose FIFO's specific weakness (Section 2.3.6), not to reflect LRU's typical advantage. On real, everyday reference patterns (which usually exhibit much stronger locality than this adversarial 12-reference string), LRU consistently outperforms FIFO in practice; the guarantee LRU actually provides is not "fewest faults on every possible string" (only Optimal has that guarantee) but immunity to Belady's anomaly, discussed in Section 2.3.6. A real implementation must timestamp or otherwise track every reference to maintain true LRU ordering, which is nontrivial in hardware; the Clock algorithm (Section 2.3.5) is the practical approximation almost every real operating system uses instead.

2.3.4 Most Recently Used (MRU)

Selection rule: evict the page that was referenced *most* recently: the exact opposite policy to LRU. This sounds perverse at first, but it is well matched to access patterns where the page just used is precisely the one *least* likely to be needed again soon, for example, repeatedly scanning sequentially through a large array or file that is bigger than the available frames, where once a page has been processed, the program moves permanently on to the next one and will not revisit it for a very long time (if ever), while older pages further back may be revisited sooner in a looping or cyclic access pattern.

Table 2.7 — MRU trace (3 frames)

Step	Reference	Frames after this step (most- to least-recently used)	Result
1	1	1	Fault
2	2	2, 1	Fault
3	3	3, 2, 1	Fault
4	4	4, 2, 1 (evict 3, the most recently used)	Fault
5	1	1, 4, 2	Hit
6	2	2, 1, 4	Hit
7	5	5, 1, 4 (evict 2, the most recently used)	Fault
8	1	1, 5, 4	Hit
9	2	2, 5, 4 (evict 1, the most recently used)	Fault
10	3	3, 5, 4 (evict 2, the most recently used)	Fault
11	4	4, 3, 5	Hit
12	5	5, 4, 3	Hit

Total: 7 faults, 5 hits. On this particular reference string, MRU happens to tie Optimal exactly, a coincidence of this specific access pattern rather than a general property of MRU (unlike LRU and Optimal, MRU is *not* a stack algorithm and is, in general, just as susceptible to Belady's anomaly as FIFO). MRU is a specialised policy, useful chiefly for the cyclic-scan access patterns described above; used on a workload with genuine temporal locality (where recently touched pages really are the ones most likely to be touched again soon), MRU performs poorly, since it deliberately discards exactly the pages LRU-style reasoning would keep.

2.3.5 Clock (Second-Chance) Page Replacement

Implementing true LRU requires hardware support (a timestamp or a full ordering of every page on every single reference) that most real machines simply do not provide, because it is too expensive to maintain at memory-reference speed. The **Clock algorithm** (also called **second-chance**) is the standard practical compromise: it approximates LRU's *behaviour* using only a single **reference bit** per frame (a bit the hardware sets to 1 automatically whenever that frame is referenced), arranged as a circular list (hence "clock") with one pointer indicating where the next replacement scan will begin.

Selection rule: on a page fault, the clock pointer examines the frame it currently points to. If that frame's reference bit is 0, that page is evicted immediately (loaded page is placed there, reference bit set to 1, and the pointer advances past it). If the reference bit is 1, the page is given a **second chance**: its reference bit is cleared to 0 (as if it had not been used recently) and the pointer advances to the next frame, repeating the same check. A page can only actually be evicted once its reference bit has been found already at 0, meaning it survived at least one earlier sweep of the pointer without being referenced again in between.

Working through the same reference string with 3 frames (numbered slot 0, 1, 2 arranged in a circle), tracking each frame's page:reference-bit:

Table 2.8 — Clock (second-chance) trace (3 frames)

Step	Reference	Slot 0	Slot 1	Slot 2	Result
1	1	1 : 1	—	—	Fault (loaded into empty slot 0)
2	2	1 : 1	2 : 1	—	Fault (loaded into empty slot 1)
3	3	1 : 1	2 : 1	3 : 1	Fault (loaded into empty slot 2)
4	4	4 : 1	2 : 0	3 : 0	Fault — scan clears bits of 1, 2, 3 in turn, then evicts 1 (now bit 0)
5	1	4 : 1	1 : 1	3 : 0	Fault — slot 1 (page 2, bit 0) evicted immediately
6	2	4 : 1	1 : 1	2 : 1	Fault — slot 2 (page 3, bit 0) evicted immediately
7	5	5 : 1	1 : 0	2 : 0	Fault — scan clears bits of 4, 1, 2 in turn, then evicts 4 (now bit 0)
8	1	5 : 1	1 : 1	2 : 0	Hit (reference bit of slot 1 set to 1 again)
9	2	5 : 1	1 : 1	2 : 1	Hit (reference bit of slot 2 set to 1 again)
10	3	5 : 0	3 : 1	2 : 0	Fault — scan clears bits of 1, 2, 5 in turn, then evicts 1 (now bit 0)
11	4	5 : 0	3 : 1	4 : 1	Fault — slot 2 (page 2, bit 0) evicted immediately
12	5	5 : 1	3 : 1	4 : 1	Hit (reference bit of slot 0 set to 1 again)

Total: 9 faults, 3 hits, identical to FIFO's fault count on this string, which is typical: Clock behaves like FIFO for pages that are never re-referenced (they get evicted on the pointer's first pass, exactly as FIFO would evict the oldest page) but, unlike FIFO, gives a recently re-referenced page a second chance to survive a sweep, which is what lets it approximate LRU's behaviour far more cheaply, using a single hardware bit per frame plus a circulating pointer, instead of a full recency ordering.

Frames with R=0 are evicted on the hand's pass; frames with R=1 get a second chance (R reset to 0)

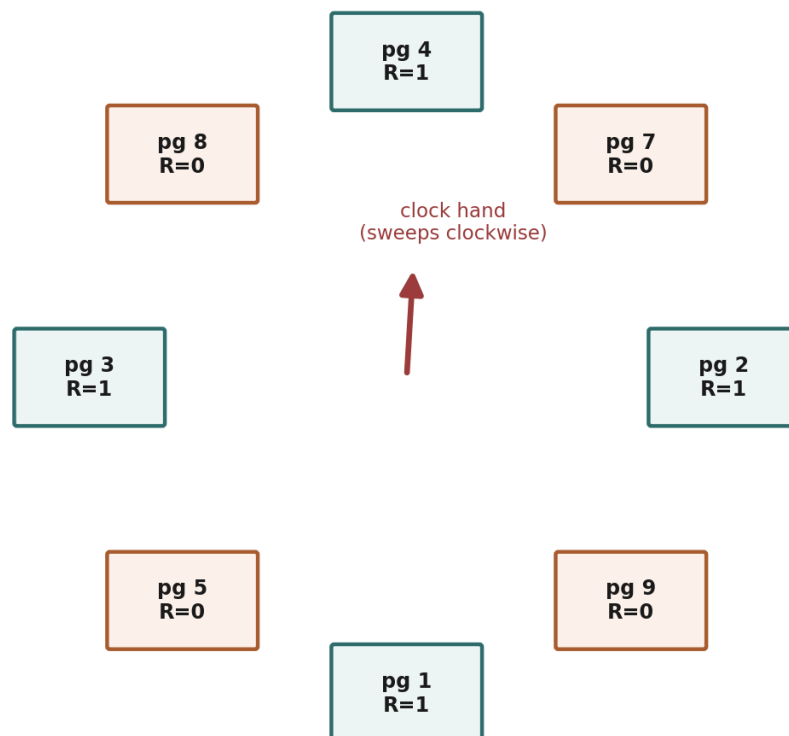


Figure 2.3 — The Clock (second-chance) algorithm

2.3.6 Belady's Anomaly

For most page-replacement algorithms, giving a process more physical frames can only help (or at worst make no difference). Intuitively, more room to keep pages resident should mean fewer faults, never more. **Belady's Anomaly**, named after László Bélády, is the surprising, counter-intuitive

discovery that for *some* page-replacement algorithms (most famously, plain FIFO), the number of page faults can actually *increase* when the number of available frames increases. This is demonstrated with the very same reference string used throughout this section.

EXAMPLE — Belady's Anomaly: FIFO with 3 frames versus 4 frames

The 3-frame FIFO trace was already worked out in full in Table 2.4, giving **9 total faults**. Now repeat the identical reference string, 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, with FIFO given a *fourth* frame:

1. 1 -> [1, -, -, -] : Fault
2. 2 -> [1, 2, -, -] : Fault
3. 3 -> [1, 2, 3, -] : Fault
4. 4 -> [1, 2, 3, 4] : Fault (all four frames now full)
5. 1 -> [1, 2, 3, 4] : Hit
6. 2 -> [1, 2, 3, 4] : Hit
7. 5 -> [2, 3, 4, 5] : Fault (evict 1, the oldest)
8. 1 -> [3, 4, 5, 1] : Fault (evict 2, the oldest)
9. 2 -> [4, 5, 1, 2] : Fault (evict 3, the oldest)
10. 3 -> [5, 1, 2, 3] : Fault (evict 4, the oldest)
11. 4 -> [1, 2, 3, 4] : Fault (evict 5, the oldest)
12. 5 -> [2, 3, 4, 5] : Fault (evict 1, the oldest)

Total faults with 4 frames: **10**.

Table 2.9 — Belady's anomaly: FIFO faults at 3 versus 4 frames

Number of frames	Total page faults (FIFO)
3	9
4	10

Giving FIFO *one additional frame* (strictly more memory to work with) produced *one more* page fault, not fewer. This is Belady's Anomaly.

2.3.7 Stack Algorithms and Immunity to Belady's Anomaly

The property that shields an algorithm from Belady's Anomaly has a name: an algorithm is a **stack algorithm** if, at every point in a reference string, the set of pages it would keep resident with n frames is always a **subset** of the set of pages it would keep resident with $n+1$ frames. If this containment property holds at every instant, adding a frame can only ever *keep* a page that would otherwise have been evicted; it can never cause a page to be evicted that would otherwise have survived. It follows immediately that the fault count can never increase as frames are added. Stack algorithms are therefore, by construction, immune to Belady's Anomaly.

LRU is a stack algorithm. With n frames, LRU always keeps resident exactly the n most-recently-referenced *distinct* pages (this is precisely what the recency ordering in Table 2.6 tracks). The $n+1$ most-recently-referenced distinct pages obviously always include, as a subset, the n most-recently-referenced distinct pages. The extra frame simply keeps one additional, slightly-less-recently-used page around, never displacing a page that a smaller allocation would have kept. This containment holds at every reference, so LRU qualifies as a stack algorithm.

Optimal is also a stack algorithm, by an exactly analogous argument: with n frames, Optimal always keeps resident the n pages that will be needed *soonest* in the future (the smallest "time to next use," which is the mirror image of LRU's "time since last use" looking backward instead of forward). The $n+1$ pages needed soonest necessarily include, as a subset, the n pages needed

soonest, so Optimal's resident set also grows monotonically as frames increase, and Optimal is likewise immune to Belady's Anomaly (in addition to already being provably optimal in fault count for any given number of frames).

FIFO is not a stack algorithm. FIFO's resident set is determined by *load order* alone, with no reference to recency of use at all, and load order interacts with frame count in a way that provides no such containment guarantee. A page evicted early with n frames (because the queue filled up sooner) may end up loaded at a completely different point in the reference string when $n+1$ frames are available, shifting every subsequent eviction decision unpredictably rather than simply adding one extra retained page on top of the smaller-frame-count set. Table 2.9 is exactly a demonstration of this failure: the 4-frame run does not simply keep everything the 3-frame run kept, plus one more page. It evicts and reloads pages on an entirely different schedule, which is precisely how it manages to end up with *more* faults despite having more room. (MRU and Clock are, for the same underlying reason, also not stack algorithms in general, and can in principle also exhibit Belady's Anomaly on some reference string, even though it did not occur for either of them on the specific string traced in this section.)

IMPORTANT — Definition — Stack Algorithm

An algorithm is a **stack algorithm** if, for every reference string and at every point within it, the set of pages resident under an allocation of n frames is a subset of the set of pages resident under an allocation of $n+1$ frames. Stack algorithms (LRU and Optimal are the two classical examples) are, by this containment property alone, immune to Belady's Anomaly; FIFO is the classic counter-example that is *not* a stack algorithm and can suffer from it.

2.3.8 Comparing the Five Algorithms

Bringing every trace in this section together, over the same 12-reference string with 3 frames:

Table 2.10 — Fault-count comparison across five algorithms (3 frames)

Algorithm	Total page faults	Total hits	Fault ratio
FIFO	9	3	$9/12 = 0.75$
Optimal (OPT)	7	5	$7/12 = 0.583$
LRU	10	2	$10/12 = 0.833$
MRU	7	5	$7/12 = 0.583$
Clock (second-chance)	9	3	$9/12 = 0.75$

Optimal, as guaranteed by theory, achieves the fewest faults of all five (no algorithm can beat it on any reference string, for a given frame count); every other algorithm's fault count is at or above Optimal's. Clock closely tracks FIFO here, as expected of a cheap hardware approximation to a recency-based scheme built on FIFO's own circular structure. The apparently poor showing of LRU and the apparently strong showing of MRU on this *particular* string is a reminder that this specific 12-reference sequence was chosen because it stresses FIFO's specific weakness (Section 2.3.6), not because it is representative of typical program behaviour; across realistic reference strings with genuine temporal locality, LRU is consistently one of the strongest practical algorithms, and MRU is a niche policy that performs well only for the specific access patterns (large sequential scans) it targets.

CHECK YOUR PROGRESS 2.3

1. State the selection rule used by each of FIFO, LRU, and Optimal in one sentence each.
2. Why is the Optimal algorithm never actually used in a real operating system, despite guaranteeing the fewest possible page faults?

3. What is Belady's Anomaly, and which of the five algorithms covered in this section is the classic example of it?
4. Define "stack algorithm" and explain, using the containment property, why LRU is immune to Belady's Anomaly while FIFO is not.
5. Why does the Clock algorithm use only a single reference bit per frame instead of a full recency ordering like true LRU, and what does it give up by doing so?

2.4 MODELLING PAGING ALGORITHMS

Section 2.3 compared five algorithms by simulating each one separately against the same reference string with a fixed frame count. That approach works, but it is expensive if the goal is to understand how an algorithm's fault count behaves *across a whole range* of frame counts (which is exactly what a system designer needs, to decide, for example, how many frames a given workload should be allocated): naively, the reference string would have to be re-simulated once per candidate frame count. **Modelling page-replacement algorithms** is the general name for techniques that let a single trace of a program's memory references be analysed once, and used to evaluate replacement policy behaviour systematically, rather than requiring the workload to actually be re-run under every candidate policy and every candidate frame count.

2.4.1 The Reference String as an Abstract Model

The starting point for any such technique is to record a program's **reference string**: the ordered sequence of page numbers it touches over the course of its execution (exactly the kind of sequence used throughout Section 2.3), captured once, independent of any particular replacement algorithm or frame count. Because the reference string is just data (a list of page numbers), it can be fed, after the fact, to any number of candidate algorithms and frame counts, entirely offline, without ever having to re-run the original program. This is what makes reference-string modelling useful for OS design and research: real programs can be traced once, and the resulting reference strings used repeatedly to evaluate and compare replacement policies.

2.4.2 The Distance-String Technique for Stack Algorithms

For the important special case of **stack algorithms** (Section 2.3.7; LRU and Optimal are the two classical examples), there is a particularly elegant one-pass technique, due to Mattson, Gecsei, Slutz, and Traiger, that goes further still: it evaluates a stack algorithm's fault count for *every possible frame count at once*, from a single pass over the reference string, rather than needing one simulation per frame count.

The technique works by maintaining, conceptually, an *unbounded* LRU stack: one with room for every distinct page ever referenced, not limited to any particular physical frame count. Each time a page is referenced, its current position (depth) in this unbounded stack is recorded as that reference's **stack distance**, and the page is then moved to the top of the stack (exactly as ordinary LRU would do). A page's *first* reference (before it has ever appeared in the stack at all) is given a stack distance of infinity, since there is no bounded number of frames small enough to have already been holding it. Recording this distance for every reference in the string produces a second sequence, the same length as the original reference string, called the **distance string**.

The payoff is this: because LRU (and, by an analogous forward-looking construction, Optimal) is a stack algorithm, a reference incurs a fault under an m -frame LRU allocation *if and only if* its stack distance is greater than m (a first-ever reference, with infinite distance, is always a fault, for any finite m ; this is a **compulsory** or **cold** miss that no replacement policy can avoid). So once the distance string has been computed (in a single pass over the reference string), the fault count for *any* frame count m can be read off immediately, just by counting how many entries in the distance

string exceed m . There is no need to re-simulate the reference string separately for each candidate m .

EXAMPLE — Building a distance string and reading off fault counts for every frame count

Take the same reference string used throughout Section 2.3: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. Build the unbounded LRU stack, recording each reference's distance (its depth in the stack *before* this reference moves it to the top) before moving on:

Time	Reference	Stack distance	Unbounded stack after (top = most recent)
1	1	infinite (first appearance)	1
2	2	infinite (first appearance)	2, 1
3	3	infinite (first appearance)	3, 2, 1
4	4	infinite (first appearance)	4, 3, 2, 1
5	1	4	1, 4, 3, 2
6	2	4	2, 1, 4, 3
7	5	infinite (first appearance)	5, 2, 1, 4, 3
8	1	3	1, 5, 2, 4, 3
9	2	3	2, 1, 5, 4, 3
10	3	5	3, 2, 1, 5, 4
11	4	5	4, 3, 2, 1, 5
12	5	5	5, 4, 3, 2, 1

The resulting distance string is: infinite, infinite, infinite, infinite, 4, 4, infinite, 3, 3, 5, 5, 5: five infinite entries (the five distinct pages' first appearances), two entries of distance 3, two of distance 4, and three of distance 5.

Now, $\text{faults}(m)$ for any frame count m is simply the count of distance-string entries strictly greater than m (with infinite always counted, for every finite m):

Table 2.11 — Distance string and fault counts for 1-5 frames

Frames (m)	Entries with distance $> m$	Total faults with m frames
1	all 12 (every finite distance here is at least 3, so all exceed 1)	12
2	all 12 (every finite distance is at least 3, so all exceed 2)	12
3	5 infinite + 2 (dist. 4) + 3 (dist. 5) = 10	10
4	5 infinite + 3 (dist. 5) = 8	8
5	5 infinite only (no finite distance exceeds 5)	5

Notice that $\text{faults}(3) = 10$ here matches exactly the 10 faults counted directly, step by step, for LRU with 3 frames in Table 2.6. The distance-string method and direct simulation necessarily agree, since they are two ways of computing the same quantity, but the distance-string method obtained the fault count for *every* value of m from a single pass over the reference string, whereas direct simulation would need five separate passes (one per candidate frame count) to produce the same table. This is precisely the efficiency gain the model buys: an OS designer plotting a curve of "fault rate versus number of allocated frames" for LRU-style replacement needs only one trace and one pass over it, not one simulation per point on the curve.

Because FIFO, MRU, and Clock are *not* stack algorithms (Section 2.3.7), the distance-string shortcut does not apply to them directly. A change in frame count can reorder or reschedule their evictions in ways a simple stack-position measurement cannot capture, so they generally must still be

simulated separately for each frame count of interest, exactly as done by hand in Section 2.3. This is one more concrete, practical consequence of the stack-algorithm property introduced in Section 2.3.7: it is not just a theoretical guarantee against Belady's Anomaly, but also what makes an entire class of efficient, single-pass evaluation techniques possible in the first place.

CHECK YOUR PROGRESS 2.4

1. What advantage does modelling a page-replacement algorithm with a recorded reference string give an OS designer, compared to only ever running the actual program?
2. Define "stack distance" for a given reference in a reference string.
3. Why is a reference's stack distance recorded as infinite the first time a particular page appears in the string?
4. Using the distance string infinite, infinite, 2, infinite, 3, 2, compute the total number of page faults for $m = 2$ frames.
5. Why does the one-pass distance-string technique apply cleanly to LRU and Optimal but not to FIFO?

2.5 DESIGN ISSUES FOR PAGING SYSTEMS

The mechanics of paging (Section 2.2) and the choice of a replacement algorithm (Section 2.3) are only part of building a real paging system. Several further design decisions have a large effect on performance and are worth studying individually.

2.5.1 Page Size Selection

Choosing the page size is a genuine trade-off with no single right answer, because several costs move in opposite directions as page size changes.

- **Internal fragmentation.** A larger page size wastes more memory, on average, in the last (partially used) page of every process, up to almost a whole page per process. Smaller pages reduce this waste.
- **Page table size.** A smaller page size means more pages per process (and per address space overall), which means a larger page table, consuming more memory purely for translation bookkeeping and adding more entries the TLB must try to cache.
- **I/O efficiency.** Reading or writing a page to disk carries a largely fixed per-transfer overhead (seek and rotational delay, for a disk) on top of the transfer itself; larger pages amortise this fixed overhead over more data per transfer, improving I/O throughput, whereas very small pages generate a disproportionate number of separate, costly I/O operations for the same total amount of data moved.

EXAMPLE — Page size and internal fragmentation

A process requires 10 KB of memory. With a 4 KB page size, it needs $\text{ceil}(10 / 4) = 3$ pages, occupying $3 * 4 = 12$ KB of physical memory, 2 KB more than it actually needs, all internal fragmentation. With a 1 KB page size, the same process needs exactly 10 pages, occupying exactly 10 KB with no internal fragmentation at all, but at the cost of a page table with more than three times as many entries to track.

The page-table-size effect compounds badly for large address spaces: a 32-bit address space (4 GB) with 4 KB pages requires $2^{32} / 2^{12} = 2^{20}$, or roughly one million, page-table entries per process; shrinking the page size to 1 KB (2^{10}) would multiply that to $2^{32} / 2^{10} = 2^{22}$, roughly four million entries: four times as much memory spent purely on translation bookkeeping, for every single process. In practice, operating systems settle on page sizes in the low kilobytes to a few kilobytes range (4 KB has long been the most common default on desktop and server systems,

though modern processors increasingly also support much larger "huge pages," measured in megabytes, specifically to shrink page table size and reduce TLB pressure for very large, memory-hungry applications such as databases).

2.5.2 Page Table Structure for Large Address Spaces

A single, flat, linear page table (one entry per page, exactly as used in the worked example of Section 2.2.3) becomes impractical once the address space is large, precisely because of the page-table-size problem just described: a single flat table for a full 32-bit or 64-bit address space would itself require megabytes to (on 64-bit systems) far more than all of physical memory, per process, even though any one process typically uses only a small, scattered fraction of its full address space at a time. Two structures address this.

A **multilevel (hierarchical) page table** breaks the page number itself into two or more parts, each indexing a separate level of a tree of smaller page tables, so that entire branches of the tree corresponding to *unused* regions of a process's address space never need to be allocated at all. A common two-level scheme splits a 32-bit logical address into an outer index, an inner index, and an offset: for example, a 10-bit outer index selecting one of 1024 entries in an *outer* page table, each entry pointing to a separate *inner* page table (also with 1024 entries), and a 12-bit offset within the eventual 4 KB page. Only inner page tables actually covering address ranges the process uses need ever be created; large, entirely unused regions of the address space (the common case for most real programs, which use only a small fraction of their nominal address space) cost nothing beyond a single null entry in the outer table. Address translation simply walks the tree, one level at a time, adding one extra memory reference per additional level. This is exactly why the TLB (Section 2.2.4) matters even more with multilevel tables, since a TLB hit skips the entire multi-level walk.

An **inverted page table** takes the opposite structural approach: instead of one table per process, indexed by *virtual* page number (with one entry for every possible page in that process's address space, used or not), the system keeps a *single* table, shared across all processes, with exactly one entry *per physical frame*, recording which process, and which of that process's virtual pages, currently occupies that frame. Because its size is tied to the amount of physical memory (which is fixed and comparatively small) rather than to the size of virtual address spaces (which, especially on 64-bit systems, can be astronomically larger than physical RAM), an inverted page table's total size does not grow at all as address spaces get larger. Its cost is on the lookup side: since the address translation now has to search the table for the entry matching a given (process, virtual page) pair (the "wrong direction" for a simple array index), a naive scan of every physical frame's entry would be far too slow, so real inverted-page-table implementations add a hash table over the (process, virtual page) pair to bring lookup back down to roughly constant time.

Table 2.12 — Single-level, multilevel, and inverted page tables compared

Structure	Size scales with	Lookup cost	Typical use
Single-level (flat) page table	Size of the virtual address space	One memory reference (direct index)	Small address spaces only; impractical for 32-bit/64-bit spaces
Multilevel (hierarchical) page table	Actual (used) portion of the virtual address space, not its full nominal size	One memory reference per level (unused branches skipped entirely)	Most general-purpose 32-bit and 64-bit systems
Inverted page table	Size of physical memory, not virtual address space	Hash lookup (needed because translation goes physical-frame-to-virtual-page in structure)	Systems prioritising minimal, memory-bounded translation-table overhead

2.5.3 TLB Management

Because address translation walks through however many page-table levels a design uses, and because an inverted page table's lookup involves a hash computation, the TLB (Section 2.2.4) is even more important to real performance than the basic hit/miss analysis in Section 2.2.4 suggested. A TLB hit skips the *entire* underlying structure, whatever it is, in one step. Managing the

TLB well raises further design questions beyond simply having one: how large it should be (larger TLBs cover more distinct pages but cost more chip area and, if not fully associative, take longer per lookup); what replacement policy to use when it is full (typically an LRU-like policy, mirroring Section 2.3.3); and how to avoid flushing it needlessly on every context switch, which is why address-space identifiers (ASIDs, Section 2.2.4) tagging each entry with its owning process are now standard. On multiprocessor systems, a further complication called **TLB shutdown** arises: when one CPU core invalidates or changes a page-table entry (say, because a page was evicted), every other core that might have cached the old, now-stale translation in its own local TLB must also be signalled to invalidate that entry, or it will keep translating to a physical frame that no longer holds the page it thinks it does.

2.5.4 Working Sets and Thrashing

A process's **working set** is, informally, the set of pages it has referenced within a recent window of time (its most recent Delta references, or the most recent Delta units of virtual time): the pages it is, in a practical sense, "actively using" right now, as distinct from the full set of pages that make up its complete address space. The **working-set model** uses this idea directly for admission and frame-allocation decisions: the operating system estimates each resident process's current working-set size, and only admits (or keeps resident) as many processes as the sum of their working sets can be comfortably held in available physical frames; if a process's working set will not fit, it is better, in this model, not to run that process at all right now than to run it starved of the frames it actually needs moment to moment.

Thrashing is what happens when this balance fails: if the combined working sets of all resident, competing processes exceed the number of physical frames available, each process is constantly missing pages that its own recent behaviour says it needs, faulting almost continuously, evicting other processes' (or even its own) still-needed pages to make momentary room, which promptly fault again moments later. The system spends nearly all of its time servicing page faults (moving pages to and from disk) and almost none of it executing actual instructions. CPU utilisation and system throughput can collapse dramatically even though every resident process is, technically, "running." Perversely, the operating system's naive response to falling CPU utilisation, admitting *more* processes on the reasoning that more resident processes should mean more work available to schedule, makes thrashing worse, not better, since it shrinks the average number of frames available per process still further.

IMPORTANT — Definition — Thrashing

Thrashing is a state in which a system spends the great majority of its time paging (servicing page faults) rather than executing useful instructions, because the combined memory demand of the currently resident processes exceeds the physical frames available to satisfy it.

Two closely related design responses address thrashing directly. The **working-set model** itself (described above) actively limits the degree of multiprogramming to whatever the current sum of working sets allows, rather than admitting processes purely because CPU utilisation looks low. The related **page-fault frequency (PFF)** approach monitors each process's actual fault rate directly: if a process's fault rate climbs above an acceptable upper threshold, it is judged to be starved and is given more frames (or, if none are free, some other process is suspended and swapped out entirely to free frames for it); if a process's fault rate falls below a lower threshold, it is judged to have more frames than it currently needs, and some can be reclaimed for a more frame-hungry process elsewhere in the system.

2.5.5 Separation of Policy and Mechanism

A recurring, general design principle in operating systems, not unique to paging but particularly well illustrated by it, is to keep **mechanism** (the low-level machinery that actually carries an action

out) cleanly separate from **policy** (the decision of *when* and *on what* to use that machinery). In a paging system, the *mechanism* is the hardware- and kernel-level machinery for loading a page into a frame, updating a page-table entry, setting or clearing reference and dirty bits, and raising a page fault when a needed page is absent; the *policy* is the choice of *which* page to evict when a frame must be freed: precisely the question every algorithm in Section 2.3 answers differently.

Keeping these separate lets the same underlying mechanism support several different, independently swappable policies: a system can be reconfigured to use FIFO, LRU-approximating Clock, or a working-set-based policy without touching the mechanism that actually performs a page-in or page-out at all. On some systems, it also lets policy decisions be made outside the kernel entirely (some research and production microkernel designs allow an application-level or user-space "external pager" to make replacement decisions tailored to that specific application's own access patterns, while the kernel mechanism underneath remains generic and unchanged). This separation is what allows page-replacement research (comparing algorithms, as in Sections 2.3 and 2.4) to proceed largely independently of the lower-level engineering of the paging hardware and kernel data structures themselves.

2.5.6 Frame Allocation and the Scope of Replacement

Two further, closely related questions sit underneath every replacement algorithm discussed in Section 2.3: how many frames should each process be given in the first place, and, when a process faults, is the operating system allowed to evict a *different* process's page to make room, or only one of its own?

Frame allocation policy decides how the total pool of physical frames is divided among competing resident processes. An **equal allocation** simply divides frames evenly among all resident processes, regardless of how large or active each one actually is: simple, but clearly wasteful for a tiny process and starving for a large one. A **proportional allocation** instead gives each process a share of frames proportional to its own size (or, more usefully, to its actual working-set size, Section 2.5.4), so a process that genuinely needs more memory receives more frames than one that does not. Either allocation can additionally be made **dynamic**, adjusting a process's frame count while it runs (typically driven by working-set or page-fault-frequency feedback, Section 2.5.4) rather than **fixed** at admission time and never revisited.

Separately, and orthogonally, a system must decide the **scope of replacement**: under **local replacement**, a faulting process may only evict one of its *own* currently resident pages, so one process's own poor locality can never degrade another, better-behaved process's performance, but a process is then strictly limited to whatever frame allocation it was already given, however that was decided. Under **global replacement**, a faulting process may evict *any* resident page system-wide, including one belonging to a completely different process. This lets frames flow naturally toward whichever process needs them most at a given moment, but it also means one process with poor locality (or one that is already thrashing) can steal frames from, and thereby degrade, every other process on the system, which is exactly the runaway dynamic working-set and page-fault-frequency policies (Section 2.5.4) exist to keep in check.

2.5.7 Shared Pages

When several processes need access to the same, unchanging code or data, paging allows them to literally **share** the same physical frame rather than each keeping a separate, redundant copy, a saving that pure contiguous allocation (Section 2.1) cannot offer nearly as cleanly, since sharing part of one contiguous block among several independently relocated processes is awkward at best. The classic example is a shared library or a common, reentrant piece of program code: if several users are simultaneously running the same editor, or several processes are linked against the same shared C library, one single copy of that code's pages can be loaded into physical memory once, with *every* process's own page table simply pointing its relevant page-number entries at those same shared frames. This is only safe for **reentrant** code: code that never modifies itself and holds no

per-process state internally, since anything one process's execution wrote into a shared frame would otherwise leak into or corrupt every other process sharing it. Read-only data can be shared just as safely as reentrant code, for the same reason.

Shared *writable* data is more delicate, and is normally handled with **copy-on-write**: two processes are initially given page-table entries pointing at the very same physical frame (marked read-only in both page tables, even though the underlying data is logically writable), and only if and when one of them actually attempts to write to it does the hardware trap, and the kernel makes a genuine private copy of just that one page for the writing process before letting the write proceed. The other process's page-table entry is left untouched, still pointing at the original, unmodified frame. This lets an operating system defer, and very often entirely avoid, the cost of copying data that both processes only ever read, while still guaranteeing each process a fully private, correct view of its "own" memory the moment either of them actually writes to it.

CHECK YOUR PROGRESS 2.5

1. Explain, using a numeric example, why a smaller page size can both reduce internal fragmentation and increase page table size at the same time.
2. Compare multilevel and inverted page tables in terms of what determines each one's size and the cost of a lookup in each.
3. Distinguish local page replacement from global page replacement, and explain why global replacement can let one process's poor locality degrade another process's performance in a way local replacement cannot.
4. Define thrashing, and explain why simply admitting more processes when CPU utilisation looks low can make thrashing worse rather than better.
5. What is the difference between "mechanism" and "policy" in the context of page replacement, and why is keeping them separate a useful design principle?

2.6 SEGMENTATION

Paging divides a process's address space into fixed-size pages purely for the convenience of physical-memory allocation, with no regard whatsoever for the *logical* structure of the program itself. A single page may straddle the boundary between a function's code and the start of a completely unrelated data structure, purely because of where 4096-byte boundaries happen to fall. **Segmentation** takes the opposite starting point: it divides a program's address space along its *natural logical boundaries* (code, global data, the run-time stack, and so on) into variable-sized units the programmer (or compiler) actually thinks in terms of, and lets each such unit be placed independently in physical memory.

2.6.1 The Concept of Segmentation

IMPORTANT — Definition — Segmentation

Segmentation is a memory-management scheme that divides a process's address space into a number of **segments**: logically distinct, independently sized units such as the code segment, the (global/static) data segment, and the stack segment, each of which is allocated its own contiguous region of physical memory, though different segments belonging to the same process need not be contiguous *with each other*.

A typical program has, at minimum, a code (text) segment holding its compiled instructions; a data segment holding global and static variables; and a stack segment holding the run-time call stack (local variables, return addresses, and saved register state for each active function call). A more elaborate segmentation scheme might further separate a heap segment (for dynamically allocated

memory) and, for larger or more modular programs, one segment per logically distinct module, library, or major data structure. Unlike a page, which is a fixed-size, logically arbitrary slice of an address space, a segment's size is meaningful and can vary enormously from one segment to the next (a small stack segment for a simple program, a large data segment for one that processes a big in-memory table) and, for a growable segment like the stack or the heap, can even change size over the process's lifetime.

2.6.2 Segment Tables and Logical Addressing

Because each segment is placed independently in physical memory, and segments vary in size, the operating system maintains, per process, a **segment table**, directly analogous in purpose to a page table: for every segment, it records that segment's **base address** (where the segment currently begins in physical memory) and its **length** (or **limit**, how large the segment is), together with protection information (for instance, marking a code segment execute/read-only, protecting it even from the process's own accidental writes).

A logical address in a segmented system is naturally expressed as a pair, (**segment number**, **offset**): the segment number selects an entry in the segment table, and the offset gives the position of the referenced byte within that segment, conceptually very similar to a page's (page number, offset) pair (Section 2.2.3), except that here the "size unit" a segment number indexes into is not a uniform, fixed quantity but a variable one, read directly out of the segment table entry itself, and used to bounds-check the offset before translation is allowed to proceed (exactly the limit-register check of Section 2.1.5, now applied per segment rather than once per whole process).

2.6.3 Address Translation Worked Example

EXAMPLE — Translating a segmented logical address

Consider a process with three segments, described by the following segment table:

Table 2.13 — Segment table for the worked example

Segment number	Segment	Base address	Length (limit)
0	Code	1000	400
1	Data	1400	300
2	Stack	1700	200

To translate a logical address (segment number, offset) into a physical address, the hardware first checks $\text{offset} < \text{length}$ for that segment's entry (a protection check, exactly as with a limit register); if the check passes, $\text{physical address} = \text{base} + \text{offset}$.

- (0, 200): Code segment, offset 200. Since $200 < 400$, the reference is valid: $\text{physical address} = 1000 + 200 = 1200$.
- (1, 150): Data segment, offset 150. Since $150 < 300$, valid: $\text{physical address} = 1400 + 150 = 1550$.
- (2, 50): Stack segment, offset 50. Since $50 < 200$, valid: $\text{physical address} = 1700 + 50 = 1750$.
- (1, 350): Data segment, offset 350. Since $350 \geq 300$ (the data segment's length), this reference falls *outside* the data segment entirely: the hardware traps immediately with a protection violation, and the physical address $1400 + 350 = 1750$ (which would, incidentally, land inside the *stack* segment's physical region) is never actually formed or used. The bounds check happens strictly before the addition, precisely so that one segment's overflow can never silently wander into another segment's memory.

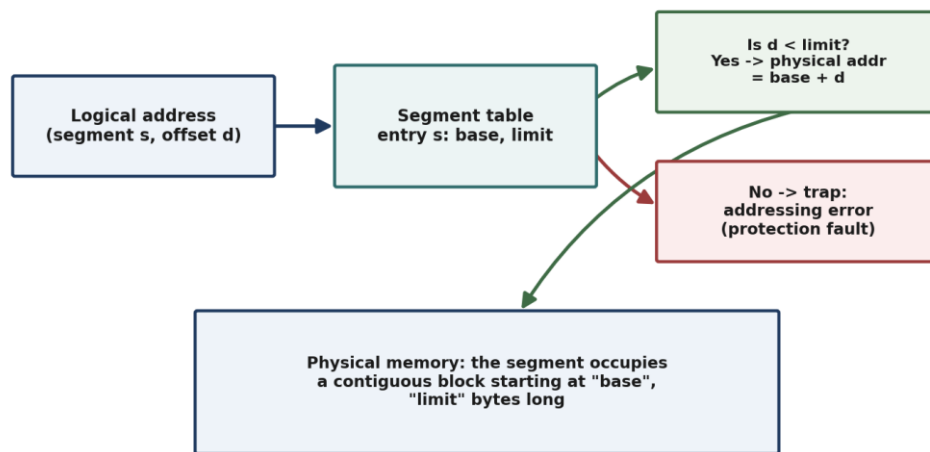


Figure 2.4 — Address translation via a segment table

2.6.4 Advantages and Disadvantages of Segmentation

Table 2.14 — Advantages and disadvantages of segmentation

Advantages	Disadvantages
Logical organisation: segments map directly onto the natural units a programmer or compiler already thinks in (code, data, stack), simplifying reasoning about a program's structure	External fragmentation: because segments are variable-sized and placed contiguously, freeing and reallocating segments over time scatters free memory into holes, exactly as with variable partitioning (Section 2.1.3)
Independent protection and sharing per segment: a code segment can be marked execute-only and safely shared read-only across multiple processes running the same program, while each process's own private data and stack segments remain fully isolated	Complexity: managing variable-sized segments, segment tables, and per-segment bounds checks is more involved than the uniform, fixed-size bookkeeping paging requires
Dynamic growth: a segment such as the stack or heap can grow (or shrink) independently of every other segment, simply by extending its recorded length, without disturbing any other segment's placement	Fitting variable sizes: finding a large-enough contiguous hole for a newly grown or newly created segment reintroduces the same first-fit/best-fit/worst-fit placement problem described in Section 2.1.3

Segmentation's strengths and weaknesses are, in a sense, a mirror image of paging's: paging is logically arbitrary but physically tidy (fixed-size, no external fragmentation); segmentation is logically meaningful but physically messy (variable-sized, prone to external fragmentation). This complementary trade-off is exactly what motivates combining the two, which is the subject of Section 2.7.

CHECK YOUR PROGRESS 2.6

1. What is the key structural difference between a page and a segment?
2. What two pieces of information does a segment table entry hold for each segment, and what is each used for?
3. Using the segment table in Table 2.13, translate the logical address (2, 150) to a physical address, showing the bounds check.
4. Name one advantage of segmentation that paging (as described in Section 2.2) does not naturally provide.
5. Why does segmentation suffer from external fragmentation while pure paging does not?

2.7 SEGMENTATION WITH PAGING

Section 2.6 ended by noting that paging and segmentation have complementary strengths: paging avoids external fragmentation but ignores logical program structure; segmentation respects logical structure but reintroduces external fragmentation. **Segmentation with paging** (a hybrid scheme, historically pioneered by the MULTICS operating system) combines both: a process's address space is still divided into logically meaningful **segments** (as in Section 2.6), but each individual segment is now, in turn, divided internally into fixed-size **pages**, and it is those pages, not whole segments, that are placed into physical memory frames.

2.7.1 A Note on Terminology: "Segmented Paging" versus "Paged Segmentation"

Both of these terms appear throughout OS textbooks and course material, and in the great majority of treatments, including the classical MULTICS-derived scheme described in this section, they are used simply as **synonyms** for the same hybrid mechanism: divide the address space into segments for logical structuring and protection, then divide each segment into pages for physical placement, avoiding external fragmentation. A logical address in this scheme is a triple, (**segment number, page number, offset**), and translation proceeds through *two* levels of table lookup (a segment table, then a page table specific to that segment), described in full in Section 2.7.2.

A few sources do draw a sharper, structural distinction between the two names, based on which structure is treated as *primary*: under this less common reading, "paged segmentation" keeps the segment as the primary logical unit, with paging used purely as an internal implementation detail for placing each segment's contents in memory (which is exactly the scheme worked through below); "segmented paging," under this alternative reading, instead treats the *entire address space* as paged first, with segments introduced afterwards merely as a way of *grouping* a process's pages for protection and sharing purposes, without segments being independently sized, logically primary units in their own right. Because this distinction is not applied consistently across textbooks, and many treat the two names as fully interchangeable, a student should expect either term to appear referring to the mechanism below, and should not assume a fixed, universal rule distinguishing them without first checking how a particular textbook or examiner defines its own terms.

2.7.2 Structure: Segment Table Pointing to Per-Segment Page Tables

In this scheme, the **segment table** no longer stores a segment's base physical address directly (as it did in pure segmentation, Section 2.6.2); instead, each segment table entry stores the base address of a **separate page table for that segment**, together with that segment's length, expressed in pages. Each such per-segment page table then works exactly like the page table of Section 2.2.3: indexed by page number (within that segment), each entry gives the physical frame number currently holding that page.

A logical address is therefore a triple, (s , p , d): segment number s , page number p *within* segment s , and offset d *within* page p . Translation proceeds in two stages: first, s indexes the segment table to find the base address (and length) of segment s 's own, private page table; second, p indexes *that* page table to find the physical frame number holding page p ; finally, the frame number and the (unchanged) offset d combine exactly as in ordinary paging: $\text{physical address} = (\text{frame number} * \text{page size}) + d$.

Each segment has its own page table, so within a segment, memory is still paged

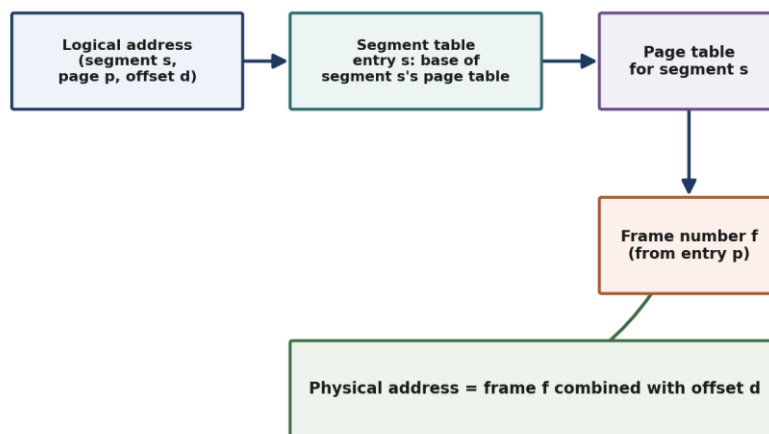


Figure 2.5 — Segmentation with paging: two-level translation

2.7.3 Worked Address Translation Example

EXAMPLE — Translating a segmented-paging logical address

Consider a process with three segments, Code (segment 0), Data (segment 1), and Stack (segment 2), where every segment is divided into pages of size 1 KB (1024 bytes). The segment table records, for each segment, the base address of *that segment's own page table* and the segment's length in pages:

Table 2.15 — Segment table for segmented paging

Segment number	Segment	Page table base address	Length (pages)
0	Code	2000	3
1	Data	3000	3
2	Stack	3500	2

Segment 1's own page table (found starting at address 3000, per the segment table above) is:

Table 2.16 — Page table for Segment 1 (data segment)

Page number (within segment 1)	Frame number
0	5
1	8
2	11

Now translate the logical address (1, 2, 100): segment 1, page 2, offset 100.

- Segment lookup.** Segment number $s = 1$. Segment 1's length is 3 pages; page number $p = 2$ is valid, since $2 < 3$. The segment table gives the base address of segment 1's page table: 3000.
- Page lookup.** Look up page $p = 2$ in segment 1's page table (Table 2.16): frame number $f = 11$.
- Physical address.** Combine the frame number with the offset: physical address = $(f * \text{page size}) + d = (11 * 1024) + 100 = 11264 + 100 = 11364$.

So the logical address (1, 2, 100) translates to physical address **11364**. Note that this required *two* table lookups (segment table, then page table) before the final address could be formed: one more level of indirection than pure paging or pure segmentation each require on their own, which is exactly the overhead this hybrid scheme trades for gaining both of their advantages at once.

2.7.4 Advantages and Disadvantages

Segmentation with paging inherits paging's core strength, no external fragmentation, since physical memory is still allocated in uniform, fixed-size frames, while restoring segmentation's logical structuring: a program's code, data, and stack remain independently sized, independently protected, and independently shareable segments, exactly as in Section 2.6, but each is now internally paged, so a segment's own size need not be a contiguous run of physical memory and does not by itself create external fragmentation the way a purely segmented scheme's segments do. The cost is added complexity and overhead: two levels of table (a segment table and, per segment, its own page table) must be maintained and walked on every translation not already served by a TLB hit, and the bookkeeping for growing or shrinking a segment now involves adding or removing page-table entries rather than simply changing one length field.

CHECK YOUR PROGRESS 2.7

1. What two structures does a logical address in segmentation-with-paging pass through, in order, before a physical address is produced?
2. In the worked example, why is (1, 2, 100) valid but (1, 3, 100) would not be?
3. Using Table 2.15 and Table 2.16, translate the logical address (1, 0, 50) to a physical address.
4. Explain, in one or two sentences, why "segmented paging" and "paged segmentation" are treated as synonyms by most textbooks, and what alternative, less common distinction some sources draw between them.
5. What specific weakness of pure segmentation (Section 2.6) does adding paging inside each segment directly fix?

2.8 SUMMARY

Without swapping or paging, a process has to live entirely in one contiguous block of memory, either a fixed partition set once at configuration or a variable one carved out to fit at load time. Fixed partitions are simple but waste space to internal fragmentation; variable partitions fit tightly but scatter free memory into external fragmentation as processes come and go. Base and limit registers solve relocation and protection cheaply: the hardware adds a base address to every logical address a process generates and checks it against a limit, all invisible to the program. The same idea, translating an address through a table, comes back again and again for the rest of the chapter.

Swapping and paging both let a system support more memory than it physically has, but very differently. Swapping moves a whole process between RAM and disk at once, simple but slow. Paging splits a process into fixed-size pages and memory into matching frames, so pages can scatter across memory and even sit partly on disk. A page table tracks where each page lives, and translating an address means splitting it into a page number and offset, looking up the page number, then combining the frame it points to with the unchanged offset. Since doing that lookup on every access would be expensive, hardware caches recent translations in a TLB, and because real programs reuse the same few pages over and over, the TLB's hit rate stays high enough to make paging fast in practice.

When memory fills up and a new page needs to come in, something has to be evicted, and we compared five ways to choose. FIFO evicts whatever's been resident longest. Optimal evicts the page needed furthest in the future; it's the best any algorithm could do, but it needs knowledge of the future, so it only exists as a benchmark. LRU evicts whatever hasn't been used in the longest time, and MRU does the opposite, which actually helps with sequential scans. Clock approximates LRU cheaply with one reference bit per frame and a moving pointer. Run against the same twelve-reference string with three frames, FIFO gave 9 faults, Optimal 7, LRU 10, MRU 7, and Clock 9, showing plainly that no cheap heuristic beats Optimal, or even beats every other heuristic, on every workload. FIFO also showed something stranger, Belady's Anomaly: giving it a fourth frame on that

same string actually raised its fault count to 10. LRU and Optimal never do this, because whatever they keep resident with n frames is always a subset of what they'd keep with $n+1$; FIFO has no such guarantee, since what it keeps depends purely on load order.

Section 2.4 generalised this comparison: record one reference string, and for algorithms with that subset property, compute a distance string in a single pass that gives the fault count for every possible frame size at once, instead of rerunning a separate simulation for each one.

A few design questions round out how paging works in real systems. Page size trades internal fragmentation against page-table size and I/O efficiency. Multilevel and inverted page tables both keep large address spaces from needing huge tables, just by different means, one skipping unused branches, the other sizing the table to physical memory instead of virtual address space. The working-set model and page-fault-frequency monitoring exist to catch thrashing before it collapses a system's throughput. And keeping the replacement policy separate from the paging mechanism means one can be swapped out without touching the other.

Segmentation takes a different approach entirely, splitting an address space along logical lines like code, data, and stack, with a segment table recording each segment's base and length. This matches memory structure to program structure and lets each segment be protected or shared independently, but it brings back the external fragmentation paging was built to avoid, since segments still have to sit contiguously. Segmenting each segment's own pages, what most textbooks call segmented paging or paged segmentation interchangeably, fixes that by translating addresses through two levels: a segment table pointing to a page table, and only then to a physical frame.

Every scheme in this chapter is really weighing the same three costs against each other: how efficiently memory gets used, how much fragmentation it creates, and how much translation overhead it adds to every memory access. Contiguous allocation keeps overhead low but risks heavy fragmentation; paging nearly eliminates fragmentation but adds translation cost; segmentation restores logical structure at the price of fragmentation again; and combining the two buys both benefits at the price of both overheads. Underneath all of it is one problem that never goes away, memory demands that are unpredictable in size and timing but must still be kept safely apart from each other, and paging and segmentation are best seen as two different, complementary answers to it.

2.9 GLOSSARY

Term	Definition
Base register	A hardware register holding the starting physical address of a process's (or partition's) memory, added to every logical address the process generates.
Belady's Anomaly	The counter-intuitive result, seen with FIFO, in which the number of page faults <i>increases</i> when the number of available frames increases.
Clock (second-chance) algorithm	A practical, cheap approximation to LRU that uses a single reference bit per frame and a circulating pointer, giving a recently used page a "second chance" before eviction.
Compaction	Relocating resident processes in memory to merge scattered free holes into one large contiguous block, remedying external fragmentation at the cost of copying overhead.
Contiguous memory allocation	Allocating a process's entire address space as one unbroken run of physical memory addresses.
Copy-on-write	A technique in which two processes share a read-only physical page until one attempts to write to it, at which point the kernel makes that process a private copy.
Dirty (modified) bit	A per-frame bit indicating whether a page's contents have changed since being loaded, so an unmodified page can be discarded rather than written back to disk.
Distance string	The sequence of stack distances produced by scanning a reference string through an unbounded LRU stack, used to evaluate a stack algorithm's fault count at every frame count in one pass.

Term	Definition
External fragmentation	Wasted memory that exists as scattered free holes between allocated blocks, each individually too small to satisfy a new request even though their combined size might be sufficient.
FIFO (First In First Out)	A page-replacement algorithm that evicts the page that has been resident the longest, regardless of how recently it was used.
Fixed partitioning	Dividing memory into a fixed number of partitions, sized once at configuration time, each holding at most one process.
Frame	A fixed-size block of physical memory, the same size as a page, into which exactly one page can be loaded.
Internal fragmentation	Wasted memory that exists inside an allocated block, because the block is larger than what its occupant actually needs.
Inverted page table	A single, system-wide page table with one entry per physical frame (recording which process and virtual page occupies it), sized by physical memory rather than by virtual address space.
Limit register	A hardware register holding the size of a process's (or partition's) allocated memory, used to bounds-check every logical address before translation.
Locality of reference	The tendency of a running program to repeatedly access a relatively small set of memory locations over any short interval, the property that makes caches (including the TLB) effective.
LRU (Least Recently Used)	A page-replacement algorithm that evicts the page that has gone unused for the longest time in the past.
Monoprogramming	Keeping only a single user program resident in memory (alongside the operating system) at any one time.
MRU (Most Recently Used)	A page-replacement algorithm that evicts the page used most recently, well suited to large sequential-scan access patterns.
Multilevel (hierarchical) page table	A page table split into two or more levels indexed by successive parts of the page number, so that unused regions of an address space need no table entries at all.
Optimal (OPT/MIN) page replacement	A page-replacement algorithm, unrealisable in practice, that always evicts the page that will not be used for the longest time into the future; provably yields the fewest possible faults for a given frame count.
Page	A fixed-size block into which a process's logical address space is divided for paging.
Page fault	A hardware trap raised when a process references a page that is not currently resident in any physical frame.
Page-fault frequency (PFF)	A frame-allocation policy that monitors each process's fault rate directly, granting or reclaiming frames as that rate rises above or falls below set thresholds.
Page replacement algorithm	The policy that decides which resident page to evict when a page fault occurs and no free frame is available.
Page table	A per-process data structure recording, for every page, which physical frame (if any) currently holds it.
Paging	A memory-management scheme dividing logical memory into fixed-size pages and physical memory into same-sized frames, allowing non-contiguous placement and partial residency.
Protection fault	A hardware trap raised when a process attempts to access a memory address outside its authorised bounds.
Reentrant code	Code that never modifies itself and holds no per-process internal state, and can therefore safely be shared, in a single physical copy, among multiple processes.
Reference bit	A per-frame bit, set by hardware whenever that frame is referenced, used by the Clock algorithm to approximate recency of use.
Reference string	The ordered sequence of page numbers a process references over some interval of execution, used to drive and compare page-replacement simulations.
Segment	A logically distinct, variable-sized unit of a process's address space (such as code, data, or stack) allocated independently in memory.

Term	Definition
Segment table	A per-process data structure recording, for every segment, its base address (or, in segmented paging, its page table's base address) and length.
Segmentation	A memory-management scheme dividing a process's address space into logically meaningful, variable-sized segments rather than fixed-size pages.
Segmentation with paging	A hybrid scheme (also called paged segmentation) in which each segment is itself divided into fixed-size pages, combining segmentation's logical structuring with paging's freedom from external fragmentation.
Stack algorithm	A page-replacement algorithm for which the set of pages resident under n frames is always a subset of the set resident under $n+1$ frames, guaranteeing immunity to Belady's Anomaly.
Swapping	Moving an entire process, as one unit, between main memory and secondary storage to free or reclaim memory.
Thrashing	A state in which a system spends the great majority of its time servicing page faults rather than executing instructions, because resident processes' combined memory demand exceeds available frames.
Translation Lookaside Buffer (TLB)	A small, fast, associative hardware cache of recent page-number-to-frame-number translations, avoiding a full page-table lookup on most memory references.
Variable partitioning	Dividing memory into partitions created dynamically, sized exactly to each process, at the moment it is loaded.
Working set	The set of pages a process has referenced within a recent window of time, used to estimate its current active memory demand.

CHAPTER 3

Inter-Process Communication, Synchronization, and Deadlocks

3.1 THE NEED FOR INTER-PROCESS SYNCHRONIZATION

3.1.1 Independent and Cooperating Processes

Every process running under a multiprogramming operating system falls into one of two categories. An **independent process** cannot affect, and cannot be affected by, any other process executing in the system: it neither shares data with another process nor depends on another process's output. A **cooperating process**, by contrast, can affect or be affected by other processes, typically because it shares data with them through a common region of memory or through files and messages.

Operating systems deliberately support cooperating processes, because process cooperation brings several concrete benefits:

- **Information sharing.** Several users or programs may legitimately need concurrent access to the same piece of information, such as a shared file or a shared in-memory data structure.
- **Computation speed-up.** A task can often be split into subtasks that execute in parallel on separate CPUs or cores, finishing sooner than a single sequential execution would.
- **Modularity.** A large system is easier to design, build, and maintain when its functions are divided among separate cooperating processes or threads rather than crammed into one monolithic program.
- **Convenience.** A single user routinely wants to run several tasks at once: editing a document, printing another, and compiling code, all "at the same time."

None of these benefits comes for free. The moment two or more processes touch the same shared data, the operating system must guarantee that their accesses happen in a controlled, predictable order. This coordination problem is **process synchronization**. Without it, the *timing* of how instructions from different processes happen to interleave on the CPU can silently corrupt an otherwise correct program.

3.1.2 Race Conditions

IMPORTANT — Definition — Race Condition

A **race condition** is a situation in which two or more processes (or threads) access and manipulate the same shared data concurrently, and the final outcome of that data depends on the particular, unpredictable order in which their instructions happen to be interleaved by the scheduler.

The word "race" captures the essence of the problem: the processes are racing to read and write the same memory location, and whichever one happens to finish — or get interrupted — at a particular instant determines what value survives. Because the CPU scheduler can preempt a process at essentially any machine instruction, the very same program, run twice with the very same input, can produce two different results. This makes race conditions notoriously difficult to work with:

- The outcome is **non-deterministic**: nothing about the source code tells you which of many possible outcomes will occur on a given run.

- The number of possible interleavings grows extremely quickly with the number of shared statements, making the fault **hard to reproduce and hard to test for**: a race condition might occur once in ten thousand runs.
- Because inserting debugging code (e.g., a print statement) itself changes the timing of a program, a race condition can appear to vanish the moment you try to observe it. Such a fault is informally called a "**Heisenbug**", in contrast to an ordinary, always-reproducible "**Bohr bug**."

EXAMPLE — Two processes incrementing a shared counter

Suppose two processes, P1 and P2, both execute the high-level statement `count = count + 1` on a shared integer variable `count`, which currently holds the value 5. At the machine level, this single line is not atomic: it compiles into (at least) three separate instructions:

1. `register = count` (load the shared variable into a CPU register)
2. `register = register + 1` (increment the register)
3. `count = register` (store the register back into the shared variable)

If P1 runs to completion before P2 starts, the two increments happen safely one after another and `count` correctly ends up at 7. But the scheduler is free to interleave the two processes' instructions at the machine-instruction level. Consider this interleaving:

Table 3.1 — A race condition on a shared counter

Step	Process	Instruction	Register value	count in memory
1	P1	<code>reg1 = count</code>	<code>reg1 = 5</code>	5
2	P1	<code>reg1 = reg1 + 1</code>	<code>reg1 = 6</code>	5
3	<i>(context switch to P2)</i>		—	5
4	P2	<code>reg2 = count</code>	<code>reg2 = 5</code>	5
5	P2	<code>reg2 = reg2 + 1</code>	<code>reg2 = 6</code>	5
6	P2	<code>count = reg2</code>	<code>reg2 = 6</code>	6
7	<i>(context switch back to P1)</i>		—	6
8	P1	<code>count = reg1</code>	<code>reg1 = 6</code>	6

Both processes believed they were incrementing `count`, and both individually executed correct code, yet the final value of `count` is 6 instead of the correct 7. P1's update was silently overwritten by P2, because P1 had already read the *old* value of `count` into its register before P2 finished its own read-increment-store cycle. This particular failure mode is called a **lost update**. Note that this is only one of several possible interleavings; another interleaving of the very same code produces the correct answer of 7. That unpredictability, correct on some runs and silently wrong on others, is exactly what makes race conditions dangerous and hard to catch through ordinary testing.

Two processes executing counter++ without synchronization

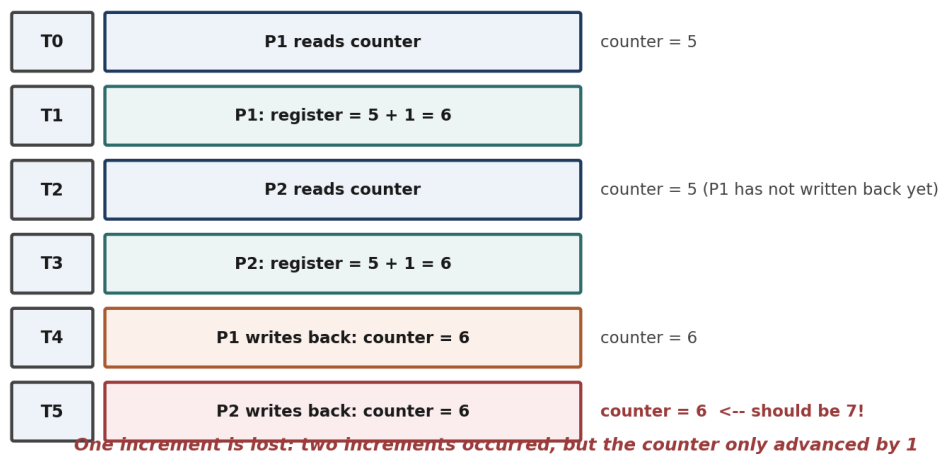


Figure 3.1 — Interleaved execution producing a lost update

3.1.3 The Critical Section

The segment of a process's code that accesses shared data, and whose execution must therefore not be interleaved with another process's access to the same data, is called the **critical section** (sometimes critical region, though Section 3.4 gives that term a more specific, technical meaning). In the counter example above, the three-instruction sequence that reads, increments, and writes back count is the critical section.

Every process that contains a critical section has the same general structure:

1. **Entry section:** the code that requests and is granted permission to enter the critical section.
2. **Critical section:** the code that actually accesses the shared resource.
3. **Exit section:** the code that signals that the process has finished with the critical section, releasing it for others.
4. **Remainder section:** all the rest of the process's code, which does not touch the shared resource.

Schematically, a process containing a critical section always has the shape entry section -> critical section -> exit section -> remainder section, repeated in a loop for as long as the process runs.

The **critical-section problem** is the problem of designing the entry and exit sections so that, no matter how the scheduler interleaves instructions from different processes, at most one process is ever executing inside its critical section at a time, and so that the resulting solution is also fair and does not lock the system up. A correct solution to the critical-section problem must simultaneously satisfy three formal requirements, which Section 3.2 states precisely: **mutual exclusion**, **progress**, and **bounded waiting**. Every synchronization mechanism discussed in this unit, from raw hardware instructions through semaphores and monitors to message passing, is at bottom a different way of solving this one underlying problem.

NOTE — Why "critical"?

A section of code is called *critical* not because it is important in the everyday sense, but because it is the point at which two independently correct pieces of code become mutually dependent: get the entry and exit protocol wrong, and the correctness of the entire program becomes hostage to the scheduler's timing decisions.

CHECK YOUR PROGRESS 3.1

1. Distinguish an independent process from a cooperating process.
2. List any three reasons an operating system provides mechanisms for process cooperation.
3. Define a race condition in your own words, and explain why it is described as "non-deterministic."
4. In the shared-counter example, explain precisely why the final value of count came out as 6 instead of 7.
5. What are the four parts of the general structure of a process containing a critical section?

3.2 MUTUAL EXCLUSION AND HARDWARE SUPPORT

3.2.1 The Three Requirements of a Correct Solution

A solution to the critical-section problem must satisfy three requirements simultaneously:

1. **Mutual exclusion.** If process P_i is executing in its critical section, then no other process may be executing in its critical section at the same time.
2. **Progress.** If no process is currently executing in its critical section, and one or more processes wish to enter theirs, then only processes that are not in their remainder section may participate in deciding which one enters next, and that decision cannot be postponed forever.
3. **Bounded waiting.** There must exist a bound on the number of times other processes are allowed to enter their critical sections after a process has requested entry to its own, and before that request is granted.

Mutual exclusion alone is easy to achieve (a single lock that is never released achieves it trivially); the difficulty of the critical-section problem lies in achieving mutual exclusion *together with* progress and bounded waiting, so that the system never deadlocks and no process starves. Two further background assumptions are standard: each process executes at some nonzero, but otherwise arbitrary and unpredictable, speed; and the underlying memory hardware guarantees that simultaneous accesses to the *same* memory location are automatically serialized (they do not corrupt each other at the bit level), even though the operating system provides no serialization above that.

3.2.2 Why a Naive Turn Variable Fails

The simplest imaginable idea is to let two processes, P_0 and P_1 , take strict turns using a single shared variable `turn`, initialized to 0, with P_i 's critical section executed only when `turn == i`:

1. `while (turn != i) { /* busy wait */ }`
2. CS: critical section
3. `turn = j` (hand the turn to the other process)
4. RS: remainder section

This achieves mutual exclusion (only one value of `turn` exists at a time, so only one process can pass the while-loop), but it **fails the progress requirement**: it forces strict alternation between the two processes regardless of which one actually wants to enter its critical section. If P_0 finishes its critical section and enters a long remainder section while P_1 has no interest in entering its critical section at all, P_0 still cannot re-enter its critical section a second time until P_1 has been through its critical section once, even though P_1 may never ask. A process is thus made to wait for a process that does not even want the resource, which directly violates progress.

A second naive attempt uses one boolean `flag[i]` per process, set to `true` when P_i wants to enter, with each process waiting for `flag[j]` to become `false`. This still satisfies mutual exclusion but can now **deadlock outright**: if both processes set their flags to `true` before either checks the other's

flag (an entirely possible interleaving), each will wait forever for a flag that will never fall, violating progress in an even more severe way. These two failures motivate the search for a solution, Peterson's, that gets all three requirements right simultaneously by combining both ideas.

3.2.3 Peterson's Solution (Two Processes)

IMPORTANT — Definition — Peterson's Solution

Peterson's solution is a software-only algorithm that solves the critical-section problem for exactly two processes, P_0 and P_1 , correctly satisfying mutual exclusion, progress, and bounded waiting using only two shared variables: a boolean array `flag[2]` and an integer `turn`.

The two processes share:

- `boolean flag[2]`, both initially false: `flag[i] == true` means P_i wants to enter its critical section.
- `int turn`: indicates whose turn it is to enter if both want in at once.

To enter its critical section, process P_i (with P_j denoting the other process) executes:

1. `flag[i] = true`: "I want to enter."
2. `turn = j`: "but you may go first if you also want to."
3. `while (flag[j] && turn == j) { /* busy wait */ }`
4. CS: critical section
5. `flag[i] = false`: "I'm done."
6. RS: remainder section

The key insight is step 2: P_i politely yields the turn to P_j *before* checking anything. If both processes attempt to enter at almost the same instant, both will execute `turn = j` (from their own perspective), but only one assignment to the single shared variable `turn` can be the one that survives: whichever process wrote `turn` last. That final value of `turn` decides who proceeds first.

- **Mutual exclusion** holds because both P_0 and P_1 being simultaneously inside their critical sections would require both `flag[0]` and `flag[1]` to be true, and `turn` to equal both 0 and 1 at once, which is impossible for a single integer variable.
- **Progress** holds because a process P_i can only be stuck in its while-loop if `flag[j] == true` *and* `turn == j`. If P_j is not interested (`flag[j] == false`), P_i proceeds immediately. If P_j is also interested and both are looping, exactly one of them has `turn` in its favour (since `turn` holds only one value) and that one proceeds.
- **Bounded waiting** holds because once P_i exits its critical section it sets `flag[i] = false`, letting P_j in immediately if P_j was waiting; and if P_i immediately tries to re-enter, it must again execute `turn = j`, which hands priority back to P_j , so P_i can enter at most once more before P_j is guaranteed a turn. No process can be overtaken more than once by the other.

NOTE — Peterson's solution and modern hardware

Peterson's solution is logically elegant and is a standard teaching example, but modern multiprocessor systems with compilers and CPUs that aggressively reorder instructions and cache shared variables in per-core caches are **not guaranteed to execute ordinary loads and stores in the order the source code implies**. Without an explicit memory barrier, `flag[i] = true` and the later read of `flag[j]` on another core may appear reordered from another core's point of view, silently breaking the proof above. This is precisely why real systems fall back on the hardware-guaranteed atomic instructions described in Section 3.2.5 rather than relying on pure software algorithms built from ordinary reads and writes.

3.2.4 The Bakery Algorithm (n Processes)

Peterson's solution only works for two processes. The **Bakery algorithm**, due to Leslie Lamport, generalizes the idea to any number n of processes, using an analogy to the numbered-ticket system used in a bakery: each customer takes a number on arrival, and the customer holding the lowest number is served next.

Shared data:

- `boolean choosing[n]`, all initially `false`: `choosing[i] == true` while P_i is in the process of computing its own number.
- `int number[n]`, all initially `0`: `number[i]` is the ticket number P_i currently holds (`0` means "not currently requesting").

Because two processes may end up computing the same number (an ordinary assignment statement is itself not atomic when several processes read and update `number [ ]` concurrently), ties are broken by process ID: process P_i is treated as holding the pair $(\text{number}[i], i)$, and pairs are compared lexicographically: $(a, i) < (b, j)$ if $a < b$, or if $a == b$ and $i < j$. Process P_i executes:

1. `choosing[i] = true`
2. `number[i] = 1 + max (number [0], number [1], ..., number[n-1])`
3. `choosing[i] = false`
4. For every other process P_k ($k = 0$ to $n-1, k \neq i$):
 - a. `while (choosing[k]) { /* wait until  $P_k$  finishes choosing its number */ }`
 - b. `while (number[k] != 0 && (number[k], k) < (number[i], i)) { /* wait for  $P_k$  to finish if it has priority */ }`
5. CS: critical section
6. `number[i] = 0`: release the ticket
7. RS: remainder section

The Bakery algorithm satisfies all three requirements: **mutual exclusion** holds because if P_i is in its critical section and P_k has already chosen a number, then $(\text{number}[i], i) < (\text{number}[k], k)$ is guaranteed to hold (by the order in which numbers and the entry test were evaluated), so P_k cannot also be in its critical section at the same time. **Progress and bounded waiting** both follow because processes are admitted to their critical sections strictly in increasing order of $(\text{number}[i], i)$, that is, in first-come-first-served order, so no process can be perpetually overtaken by newly arriving processes.

EXAMPLE — Why the `choosing[]` flag is necessary

Suppose P_i and P_j compute their numbers in an interleaved fashion: both read the current maximum before either writes back its own new number, so both end up with the same number value, say 5. Without the tie-break rule of comparing $(\text{number}[i], i)$ as a pair, both processes could believe they hold the "lowest" number and both enter their critical sections together. Similarly, `choosing[i]` must be checked because if P_k observes P_i 's `number[ ]` slot *while P_i is in the middle of computing it*, P_k might read a stale or partially updated value and make an incorrect ordering decision. Both details exist specifically to survive interleaving at the level of individual instructions, exactly like the race condition of Section 3.1.

The Bakery algorithm's chief drawback in practice is that it still relies on **busy waiting** (as does Peterson's solution) and does not scale gracefully: every process must, in principle, examine every other process's ticket, an $O(n)$ cost on every entry to the critical section.

3.2.5 Hardware Support for Mutual Exclusion

Software-only algorithms such as Peterson's and the Bakery algorithm are intellectually important, but as noted in Section 3.2.3, they are not dependable on modern hardware, and they are also awkward for application programmers to use directly. Real operating systems instead build their locking primitives on **hardware support**: either by disabling interrupts, or by using special machine instructions that are *atomic*, meaning the hardware guarantees they execute as one indivisible, uninterruptible step, even in the presence of multiple CPUs sharing memory.

Disabling interrupts. On a uniprocessor system, the critical-section problem can be solved trivially by disabling interrupts before entering the critical section and re-enabling them on exit. With interrupts disabled, the currently running sequence of instructions cannot be preempted, so no other process can be scheduled to run and interfere with the shared variable. This is exactly the strategy many non-preemptive kernels use internally for very short critical sections. It does not extend well to multiprocessors, however: disabling interrupts on one CPU does nothing to stop a process running concurrently on a *different* CPU from touching the same shared data, and broadcasting an interrupt-disable message to every processor is slow and delays every processor's response to real hardware events (including the system clock), making the approach unattractive as a general-purpose lock.

Atomic hardware instructions. Multiprocessor-safe hardware instead provides special instructions that perform what looks like two actions (a test and a modification) on a single memory word as one atomic, uninterruptible unit, with the memory hardware itself enforcing that only one CPU's atomic instruction can touch a given memory location at a time. Two classic examples are Test-and-Set and Swap (also called Exchange).

Test-and-Set(lock) atomically reads the current value of lock, sets lock to true (or 1), and returns the value that lock held *before* it was set:

```
1. boolean TestAndSet(boolean *lock) {
2.   boolean old = *lock;
3.   *lock = true;
4.   return old;
5. }
```

Crucially, steps 2–3 are guaranteed by the hardware to happen as a single, uninterruptible operation: no other processor's Test-and-Set on the same memory word can be interleaved in the middle. With a shared boolean lock, initialized to false, mutual exclusion is achieved as:

```
1. while (TestAndSet(&lock)) { /* busy wait */ }
2. CS
3. lock = false
4. RS
```

The first process to execute Test-and-Set finds `lock == false`, gets back false from the call (so its while-loop condition is false and it proceeds immediately), and simultaneously sets lock to true: locking out everyone else, whose own calls to Test-and-Set will now read back true and keep looping. This plain version satisfies mutual exclusion and progress, but **not bounded waiting**: when the lock is released, *any* of the waiting processes may be the next one to succeed at Test-and-Set, purely by the luck of scheduling. In principle, one unlucky process could be starved indefinitely while others repeatedly win the race.

Swap (a, b) (or Exchange) atomically swaps the contents of two memory words:

```
1. void Swap (Boolean *a, Boolean *b) {
2.   Boolean temp = *a;
3.   *a = *b;
4.   *b = temp;
}
```

5. }

With a shared lock (initially false) and a local variable key per process:

1. key = true
2. while (key == true) {Swap (&lock, &key);}
3. CS
4. Swap (&lock, &key) (equivalently, lock = false)
5. RS

The reasoning is symmetrical to Test-and-Set: the only process that can proceed past the while-loop is the one whose Swap happens to read lock == false and, in the same atomic step, sets lock to true. Like plain Test-and-Set, plain Swap guarantees mutual exclusion and progress but not bounded waiting on its own.

Adding bounded waiting. A small amount of extra bookkeeping fixes the starvation problem. Using an additional shared array waiting[n] (all initially false) and the shared lock, process P_i executes:

1. waiting[i] = true
2. key = true
3. while (waiting[i] && key) {key = TestAndSet(&lock);}
4. waiting[i] = false
5. CS
6. j = (i + 1) % n
7. while (j != i && ! waiting[j]) {j = (j + 1) % n;}
8. if (j == i) {lock = false;} else {waiting[j] = false;}
9. RS

On exit, P_i scans the other processes in circular order starting just after itself and hands the critical section directly to the next process that is actually waiting (by clearing that process's waiting [] flag), rather than simply releasing lock for an arbitrary winner. This guarantees that any process waiting for the critical section is admitted within at most n - 1 turns of other processes, satisfying bounded waiting.

IMPORTANT — Why atomicity is the crucial ingredient

Software algorithms such as Peterson's and the Bakery algorithm build correctness out of ordinary loads and stores: plain memory reads and writes that a compiler or CPU is, in general, free to reorder or delay relative to each other on modern multiprocessor hardware. Hardware instructions such as Test-and-Set and Swap sidestep this problem entirely: the CPU and memory subsystem *guarantee*, as a hardware contract, that the read-and-modify sequence inside the instruction happens as a single indivisible step with respect to every other processor in the system. That hardware-enforced atomicity is exactly the property that a sequence of separate, ordinary instructions cannot reliably provide once instruction reordering, caching, and true simultaneity across multiple cores enter the picture. This is why every higher-level synchronization tool in this unit (semaphores, monitors, locks) is, underneath, ultimately implemented using one of these atomic instructions or their modern equivalents (such as compare-and-swap).

Table 3.2 — Mutual exclusion approaches compared

Approach	Number of processes	Busy-waits?	Bounded waiting guaranteed?	Works on modern multiprocessors without extra care?
Naive turn variable	2	Yes	Yes (but fails progress)	No

Approach	Number of processes	Busy-waits?	Bounded waiting guaranteed?	Works on modern multiprocessors without extra care?
Peterson's solution	2	Yes	Yes	Not guaranteed (ordinary loads/stores can be reordered)
Bakery algorithm	n	Yes	Yes (FCFS order)	Not guaranteed (same reason)
Disabling interrupts	Any (uniprocessor only)	No	N/A (no other process can run)	No (impractical on multiprocessors)
Test-and-Set / Swap (plain)	Any	Yes	No	Yes (hardware-atomic)
Test-and-Set with waiting[] array	Any	Yes	Yes	Yes (hardware-atomic)

CHECK YOUR PROGRESS 3.2

1. State the three requirements a correct solution to the critical-section problem must satisfy.
2. Why does a simple shared turn variable fail the progress requirement?
3. In Peterson's solution, explain why setting $turn = j$ (rather than $turn = i$) in the entry section is essential to the algorithm's correctness.
4. Why is disabling interrupts an unattractive general solution on a multiprocessor system?
5. Explain, with reference to the definition of Test-and-Set, why it must be implemented as a single atomic hardware instruction rather than as separate read and write instructions.

3.3 SEMAPHORES

3.3.1 Definition and Basic Operations

The hardware-based solutions of Section 3.2 work, but they are low-level and error-prone for application programmers to use directly: every critical section in a program would need its own hand-written busy-wait loop built around a raw atomic instruction. **Semaphores**, introduced by Edsger Dijkstra, package this idea into a higher-level, reusable synchronization tool.

IMPORTANT — Definition — Semaphore

A **semaphore** S is an integer variable that, apart from its initialization, may be accessed only through two standard atomic operations: $wait(S)$ (also historically called P , from the Dutch *proberen*, "to test") and $signal(S)$ (also called V , from *verhogen*, "to increment").

Conceptually, $wait(S)$ is the operation a process performs to *request*, and if necessary, wait for, a unit of whatever resource the semaphore protects, and $signal(S)$ is the operation it performs to *release* that unit back. The essential property of both operations is that they are **atomic**: once one process begins executing $wait$ or $signal$ on a semaphore, no other process can interleave its own $wait$ or $signal$ on that same semaphore in the middle. This atomicity is itself typically built on the hardware instructions of Section 3.2.5 (or, in a single-processor kernel, on briefly disabling interrupts).

A simple, spin-lock-style definition of the two operations, ignoring for a moment how blocking is implemented, is:

$wait(S)$:

1. while ($S \leq 0$) { /* no-op – busy wait */ }
2. $S = S - 1$

signal(S):

1. $S = S + 1$

The while loops and decrement inside wait must be executed as a single atomic block (an interleaved read-test-decrement across two processes would reopen exactly the race condition semaphores exist to prevent); signal's single increment is trivially atomic on its own.

3.3.2 Binary Semaphores and Counting Semaphores

Semaphores come in two flavours, distinguished by the range of values they are allowed to take and, correspondingly, by the kind of problem each is suited to:

- A **binary semaphore** can only ever hold the value 0 or 1. It behaves exactly like a mutual-exclusion lock: 1 means the single protected resource (or critical section) is free, and 0 means it is currently held. Binary semaphores are the natural tool for enforcing straightforward mutual exclusion over one critical section, and are often given the name *mutex* for exactly this reason.
- A **counting semaphore** can hold any non-negative integer value and is used to manage access to a *pool* of k identical resource instances (for example, k free buffer slots, or k available printer connections). Initializing the semaphore to k allows up to k processes to proceed past wait() without blocking; the $k + 1$ -th process to call wait() finds the semaphore at 0 and must wait until some process that is already using a unit calls signal(). Setting $k = 1$ reduces a counting semaphore to the binary case, which is why binary semaphores are sometimes treated as a special case rather than a wholly separate mechanism.

EXAMPLE — Choosing binary versus counting

Guarding a single shared linked list against concurrent modification calls for a **binary semaphore** *mutex*, initialized to 1: exactly one process may be inside the list-modifying critical section at a time. Managing a pool of five identical database connection handles calls for a **counting semaphore** *connections*, initialized to 5: up to five processes may each hold one connection simultaneously, and a sixth process calling wait(*connections*) blocks until one of the five is released via signal(*connections*).

More generally, for an n -process critical section that should admit at most k processes simultaneously, a single semaphore *mutex* initialized to k handles it directly: every process executes wait(*mutex*) before, and signal(*mutex*) after, its critical section, and the semaphore's own value transparently tracks how many "slots" remain.

3.3.3 Busy-Waiting versus Blocking (Queuing) Implementations

The definition of wait(S) given in Section 3.3.1 has a serious practical drawback: a process that finds $S \leq 0$ sits in a tight loop, continuously re-testing S , and burns CPU cycles doing no useful work at all while it waits. This is called **busy waiting**, and the lock built this way is called a **spinlock**. Busy waiting is tolerable only when the expected wait is very short (shorter than the cost of two context switches), because it wastes CPU time that could otherwise run a different, ready process. On a uniprocessor, it can even prevent the very process holding the semaphore from ever running again to release it.

The practical alternative, and the way semaphores are actually implemented inside most real operating-system kernels, is a **blocking (queuing) implementation**. Instead of looping, a process that cannot immediately proceed is *put to sleep* (moved out of the ready queue into a blocked state) and placed on a waiting queue that belongs to that specific semaphore. The CPU is freed for other ready processes to use while the blocked process waits. When another process calls signal(), if any process is waiting on that semaphore's queue, exactly *one* of them is woken up and moved back

to the ready queue. The semaphore's value itself may not even need to change, because the waiting process is handed the resource directly.

IMPORTANT — Definition — Blocking (Queuing) Semaphore Structure

A semaphore implemented for blocking is represented as a **record** containing an integer value and a **list (queue) of process control blocks (PCBs)** waiting on that semaphore:

```
1. typedef struct {
2.   int value;
3.   struct process *queue; // list of waiting PCBs
4. } semaphore;
```

wait(S) and signal(S) are each executed as a single atomic (indivisible) block, typically by briefly disabling interrupts, or by protecting the semaphore's own internal state with a hardware atomic instruction, so that no two processes can simultaneously manipulate the same semaphore's value or queue:

wait (semaphore *S):

```
1. S->value = S->value - 1
2. if (S->value < 0)
   {
3.   add this process to S->queue;
4.   block this process (remove it from the CPU, giving up control to the scheduler);
   }
```

signal (semaphore *S):

```
1. S->value = S->value + 1
2. if (S->value <= 0)
   {
3.   remove a process P from S->queue;
4.   wakeup(P); // move P to the ready queue
   }
```

Under this implementation, S->value is allowed to go **negative**, and when it is negative its magnitude directly counts the number of processes currently blocked on that semaphore's queue: for instance, S->value == -3 means three processes are sitting on S->queue waiting to be woken. This is a useful mental model even though the exact same public interface (wait/signal) is used regardless of whether value is implemented to go negative or to bottom out at zero with a separate waiting-process count.

Two subtleties are worth being precise about. First, signal(S) wakes **exactly one** waiting process (not all of them); the specific process chosen depends on the queuing discipline, most commonly first-in-first-out (FIFO), though this is not part of the semaphore's formal contract and some texts leave the choice unspecified. Second, the operating system, not the calling process, decides *when* the blocked process actually resumes running (it becomes ready, but must still be scheduled), which is precisely why this approach avoids wasted CPU cycles: a blocked process consumes no CPU time at all until it is woken and scheduled.

Table 3.3 — Busy-waiting versus blocking semaphore implementation

Aspect	Busy-waiting (spinlock) implementation	Blocking (queuing) implementation
What a waiting process does	Repeatedly re-tests S in a tight loop	Is removed from the CPU and placed on S's own waiting queue

Aspect	Busy-waiting (spinlock) implementation	Blocking (queuing) implementation
CPU usage while waiting	Wastes CPU cycles continuously	None: CPU is free for other ready processes
Underlying data needed	Just the integer <i>S</i>	Integer value plus a linked list/queue of blocked PCBs
Best suited to	Very short expected waits, multiprocessor low-level locks	General-purpose OS/application synchronization, potentially long waits
Who wakes a waiting process	The process notices the change itself, by polling	<code>signal()</code> explicitly wakes exactly one waiting process from the queue

3.3.4 Using Semaphores to Solve the Critical-Section Problem

To protect an *n*-process critical section, the processes share one binary semaphore `mutex`, initialized to 1. Each process wraps its critical section as:

1. `wait(mutex)`
2. CS
3. `signal(mutex)`
4. RS

Because `mutex` starts at 1, the first process to call `wait(mutex)` succeeds immediately (decrementing it to 0) and proceeds into its critical section; every other process calling `wait(mutex)` while the first is still inside finds the semaphore at 0 or below and blocks. When the process inside calls `signal(mutex)`, the semaphore is incremented and, if any process was waiting, exactly one of them is woken and allowed to proceed. This directly satisfies mutual exclusion; correctly implemented (with atomic `wait/signal` and a fair queue discipline), it also satisfies progress and bounded waiting, all while hiding the underlying hardware-level atomic instruction entirely from the application programmer.

NOTE — A discipline, not a guarantee

Semaphores are a powerful and convenient tool, but they enforce nothing by themselves. Correctness depends entirely on every process in the program calling `wait` and `signal` correctly and in matching pairs. A single omitted `signal()` (for example, on an exceptional early-return path) permanently locks every other process out of that critical section; a spurious extra `signal()`, or a `wait()` called on the wrong semaphore, can just as easily destroy mutual exclusion altogether. This fragility, timing-dependent bugs that appear only under specific, rare interleavings, is exactly what motivates the more structured, compiler-enforced constructs of Section 3.4: critical regions and monitors.

CHECK YOUR PROGRESS 3.3

1. Define a semaphore and name its two atomic operations.
2. Distinguish a binary semaphore from a counting semaphore, and give one situation naturally suited to each.
3. Explain what "busy waiting" means, and why it is wasteful.
4. Describe the two fields that make up a blocking (queuing) semaphore's internal structure, and explain what a negative value represents.
5. In the blocking implementation, precisely how many waiting processes does a single call to `signal(S)` wake up?

3.4 CRITICAL SECTIONS, CRITICAL REGIONS, AND MONITORS

3.4.1 From Critical Sections to Critical Regions

Section 3.1 introduced the **critical section** as a general concept: any stretch of code that touches shared data and must not be interleaved with another process's access to the same data. Semaphores (Section 3.3) let a programmer *implement* a critical section, but they place the entire burden of correctness on the programmer's discipline: every entry point into shared data needs its own matching wait/signal pair, written by hand, with no help from the compiler if a pair is mismatched or omitted.

The **critical region** is a language-level construct designed to remove exactly that burden, by making the compiler itself responsible for inserting the entry and exit synchronization. Rather than writing raw wait/signal calls around arbitrary code, the programmer declares which variable is shared and syntactically brackets the code that touches it:

```
region v do S
```

Here *v* is a shared variable (declared once, with the compiler tracking who is allowed to touch it), and *S* is a statement (or block) that operates on *v*. The compiler guarantees, without any further work from the programmer, that no two processes ever execute their `region v do S` blocks over the same *v* at the same time: the entry and exit protocol (equivalent to the wait/signal pair around a semaphore) is generated automatically. This is a genuine improvement over raw semaphores: it is syntactically impossible to forget the exit step, and the compiler can verify that a shared variable *v* is *only* ever touched inside a `region v` block, catching accidental unprotected accesses at compile time that a programmer using bare semaphores could easily miss.

3.4.2 The Conditional Critical Region

A plain critical region only ever asks, "is *v* currently free?" Many real synchronization problems, however, need a process to wait not merely for exclusive access to a shared variable, but for that variable to satisfy some additional **condition** before it is safe to proceed. For example, a consumer must not merely acquire exclusive access to a shared buffer; it must also wait until the buffer is *non-empty*. The **conditional critical region** extends the plain critical region with exactly this capability:

```
region v when B do S
```

Here *B* is a boolean condition, expressed in terms of (typically) the shared variable *v*. A process attempting to enter this region is granted entry only when *both* of the following hold: no other process is currently inside any region on *v* (the ordinary mutual-exclusion requirement), *and* the condition *B* currently evaluates to true. If *B* is false when the process attempts entry, the process is blocked (not busy-waiting: it is removed from contention for the CPU) until some other process's exit from a region on *v* causes *B* to become true, at which point the compiler-generated runtime re-evaluates *B* and admits the waiting process if it now holds.

IMPORTANT — Distinguishing the two constructs

A plain **critical region** (`region v do S`) only ever tests "is *v* currently available?"; it has no notion of a condition beyond simple mutual exclusion. A **conditional critical region** (`region v when B do S`) adds an explicit boolean guard *B* that must additionally hold before entry is granted, and blocks the requesting process (rather than merely queuing it for the lock) if *B* is currently false. Every plain critical region is trivially a conditional critical region with *B* fixed to true; the reverse is not true.

EXAMPLE — Bounded buffer with conditional critical regions

A producer wishing to insert an item into a shared bounded buffer `buf` of capacity `k` would write region `buf` when `(count < k)` do { `insert item; count = count + 1;` }, while a consumer would write region `buf` when `(count > 0)` do { `remove item; count = count - 1;` }. Neither process needs to write any explicit wait/signal logic, nor manage a separate "how many free slots" semaphore by hand: the compiler-generated runtime evaluates the guard condition and blocks or admits the process automatically.

Despite this elegance, conditional critical regions have practical drawbacks that limited their adoption and motivated the design of monitors. Re-evaluating the boolean condition `B` every time *any* process exits a region on the same variable `v` can be computationally expensive, especially when many processes are contending for the same variable with different conditions. The construct also requires compiler support that was uncommon and difficult to retrofit onto the languages of the time; and because a condition `B` can reference any part of the shared state, the implementation cannot easily know in advance which specific event might make a given waiting process's condition become true, forcing an inefficient blanket re-check of every waiting process's condition on every exit. These practical costs are exactly what led language designers to look for a construct that kept the compiler-enforced mutual exclusion of the critical region, but let the *programmer*, who best knows which specific event a waiting process cares about, decide explicitly when to wake a particular waiter. That construct is the **monitor**.

3.4.3 Monitors

IMPORTANT — Definition — Monitor

A **monitor** is a high-level synchronization construct, an abstract data type, that encapsulates a set of shared data together with the procedures that are the *only* means of operating on that data, with the guarantee that at most one process may be actively executing inside any of the monitor's procedures at any given time. This mutual exclusion is enforced implicitly by the monitor itself; the programmer never writes an explicit lock, semaphore, or wait/signal pair to obtain it.

The general syntactic structure of a monitor is:

```
1. monitor MonitorName {
2.   // shared variable declarations – the monitor's private state
3.   procedure P1(...) { ... }
4.   procedure P2(...) { ... }
5.   ...
6.   procedure Pn(...) { ... }
7.   initialization code(...) { ... }
8. }
```

The variables declared inside the monitor can be accessed only by the monitor's own procedures (no process outside the monitor can touch them directly), and those procedures, in turn, can access only the monitor's own local variables and their own parameters. This encapsulation, combined with the implicit mutual exclusion the monitor construct itself guarantees at its boundary, means that once code has been correctly written *inside* a monitor's procedures, no process anywhere else in the program can accidentally violate its invariants by an unsynchronized access. That is a much stronger, more localized correctness guarantee than semaphores can offer.

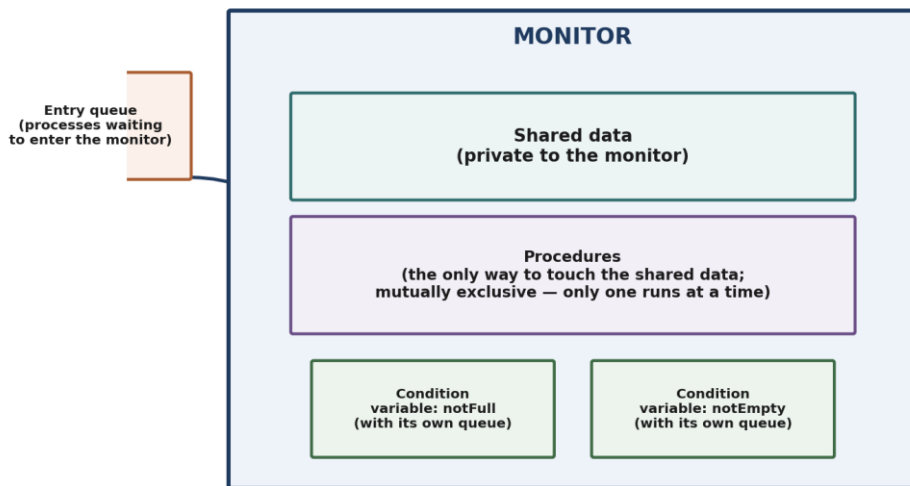


Figure 3.2 — Structure of a monitor

Condition variables. Implicit mutual exclusion at the monitor's boundary is not, on its own, expressive enough for every synchronization need: a process legitimately inside the monitor may still need to wait for some internal condition to become true (for example, a bounded-buffer insert procedure needs to wait until the buffer is not full). Monitors provide this via explicitly declared **condition variables**:

```
condition x, y;
```

A condition variable supports exactly two operations, `wait()` and `signal()`, whose meaning is deliberately different from the semaphore operations of the same name:

- `x.wait()`: the calling process is immediately suspended (removed from the monitor, freeing it for other processes) and placed on a queue associated specifically with condition `x`, until some other process calls `x.signal()`.
- `x.signal()`: resumes **exactly one** process suspended on `x`, if one exists. If no process is currently waiting on `x`, `signal()` has **no effect whatsoever**: the condition variable retains no memory of the signal, unlike a semaphore's `signal()`, which always increments the semaphore's value regardless of whether anyone was waiting.

Because the monitor allows only one process to be *active* inside it at a time, a subtlety arises the moment a process `P` calls `x.signal()` while some process `Q` is already suspended on `x`: both `P` and `Q` cannot simultaneously be active inside the monitor, so a choice must be made about who proceeds first.

- **Signal-and-wait.** `P`, having just called `signal`, itself waits (either until `Q` leaves the monitor, or on some other condition), and `Q` resumes immediately.
- **Signal-and-continue.** `Q` remains suspended, and `P` continues executing inside the monitor; `Q` will only resume once `P` (and possibly others) eventually leaves the monitor or waits.

Both choices are used in practice, and each has a reasonable justification: signal-and-continue lets the already-running process `P` finish its work without an extra context switch, while signal-and-wait guarantees that the condition `Q` was waiting for still holds exactly when `Q` resumes (under signal-and-continue, `P`'s subsequent actions before it itself leaves or waits could, in principle, invalidate the very condition that made `signal()` correct to call). A well-known compromise, adopted in Concurrent Pascal, has the signalling process leave the monitor immediately upon calling `signal`, so the woken process resumes right away with no ambiguity.

EXAMPLE — Bounded buffer solved with a monitor

A monitor `BoundedBuffer` encapsulates the buffer array, an insertion index, a removal index, and a count of items currently stored, together with two condition variables, `notFull` and `notEmpty`:

```

1. monitor BoundedBuffer {
2.   item buffer[k]; int count = 0, in = 0, out = 0;
3.   condition notFull, notEmpty;
4.   procedure insert(item x) {
5.     if (count == k) notFull.wait();
6.     buffer[in] = x; in = (in + 1) % k; count = count + 1;
7.     notEmpty.signal();
8.   }
9.   procedure remove(item &x) {
10.    if (count == 0) notEmpty.wait();
11.    x = buffer[out]; out = (out + 1) % k; count = count - 1;
12.    notFull.signal();
13.  }
14. }

```

A producer simply calls `BoundedBuffer.insert(item)`; a consumer calls `BoundedBuffer.remove(x)`. Neither writes a single `wait(mutex)/signal(mutex)` pair: the monitor's implicit mutual exclusion guarantees only one caller is ever active inside `insert` or `remove` at once, and the two condition variables handle exactly the "buffer full" and "buffer empty" waiting cases that a semaphore-based solution (Section 3.5.2) must instead manage by hand with two extra counting semaphores.

Monitors have been adopted, in various forms, by several real languages: Concurrent Pascal, Mesa, Java (where every object can act as a monitor via synchronized methods and `wait/notify`), and C#, among others.

3.4.4 Message Passing

Every mechanism discussed so far (semaphores, critical regions, monitors) assumes processes communicate through **shared memory**: a common region of memory that all cooperating processes can read and write. This is not always possible or convenient, particularly when the cooperating processes do not share an address space at all (for example, processes on different machines connected by a network). **Message passing** provides an alternative model in which processes communicate and synchronize purely by exchanging messages, using two primitive operations:

- `send(destination, message)`
- `receive(source, message)`

Direct versus indirect communication. In **direct communication**, each process names the other explicitly: `send(P, message)` sends directly to process P, and `receive(Q, message)` receives directly from process Q. This requires every communicating process to know the identity of its partner in advance, which is simple but inflexible: the sender and receiver are tightly coupled by name. In **indirect communication**, processes instead send to, and receive from, a shared intermediate object called a **mailbox** (or **port**): `send(mailbox, message)` and `receive(mailbox, message)`. Two processes can communicate only if they share access to the same mailbox, but neither needs to know the other's identity directly, and a mailbox may be shared by more than two processes, giving considerably more flexibility (a mailbox can even be created and later destroyed,

or reassigned to a different pair of communicating processes, entirely independently of the processes themselves).

Synchronous versus asynchronous communication. Message passing may additionally be **blocking (synchronous)** or **non-blocking (asynchronous)**, independently for both send and receive:

- A **blocking send** does not return until the message has actually been received by the destination (or, in a common variant, until it has at least been safely placed in the destination's mailbox); a **blocking receive** does not return until a message is actually available, suspending the calling process in the meantime. When both send and receive block, the two processes are said to **rendezvous**: synchronization and data transfer happen together, at a single well-defined point in each process's execution.
- A **non-blocking send** returns to the caller immediately, without waiting for the message to be received: the sending process simply resumes its work, and the message is buffered until the receiver asks for it. A **non-blocking receive** returns immediately as well, either with a message if one is available or with an explicit indication that none is currently available (rather than waiting).

Table 3.4 — Message-passing variants compared

Dimension	Variant A	Variant B
Naming	Direct: sender/receiver name each other explicitly	Indirect: sender/receiver communicate through a shared mailbox/port
Timing (send)	Blocking send: sender waits until the message is received	Non-blocking send: sender continues immediately, message is buffered
Timing (receive)	Blocking receive: receiver waits until a message arrives	Non-blocking receive: receiver returns immediately, with or without a message

Message passing and shared memory are not mutually exclusive design choices within one operating system. Many real systems provide both, using shared memory for high-volume, low-latency communication between cooperating processes on the same machine, and message passing for communication that must cross machine or protection-domain boundaries, or where the simplicity of not managing a shared address space directly is worth the added system-call overhead.

CHECK YOUR PROGRESS 3.4

1. What extra capability does a conditional critical region provide over a plain critical region?
2. State two practical drawbacks of conditional critical regions that motivated the design of monitors.
3. How does the meaning of `signal()` on a monitor condition variable differ from `signal()` on a semaphore, when no process is currently waiting?
4. Distinguish `signal-and-wait` from `signal-and-continue` as ways of resolving a monitor's `signal()` call.
5. Distinguish direct from indirect message-passing communication, and blocking (synchronous) from non-blocking (asynchronous) communication.

3.5 CLASSICAL PROBLEMS IN CONCURRENT PROGRAMMING

A handful of synchronization problems have become the standard "test cases" against which every newly proposed synchronization mechanism is measured, because between them they exercise nearly every kind of coordination a real system needs: bounded resource pools, competing readers and writers, and the circular resource dependency that produces deadlock. This section presents

all four in the classical order, with a correct semaphore-based (or monitor-based, where clearer) solution for each.

3.5.1 The Bounded-Buffer (Producer-Consumer) Problem

The problem. A **producer** process generates data items one at a time and places them into a shared buffer; a **consumer** process removes items from the same buffer, one at a time, and processes them. The buffer has a fixed capacity of k slots. The producer must not be allowed to insert into a full buffer, and the consumer must not be allowed to remove from an empty buffer.

What goes wrong without synchronization. If the producer and consumer manipulate the buffer's shared index variables (and the item count) without coordination, two distinct failures are possible. First, an ordinary race condition on the shared count (or in/out index) variables, exactly as in Section 3.1, can silently corrupt the count, causing the consumer to read stale or already-consumed data, or the producer to overwrite an item that has not yet been consumed. Second, even with the count protected by mutual exclusion alone, the producer needs to know *when* the buffer is full (to stop producing) and the consumer needs to know *when* the buffer is empty (to stop consuming); ordinary mutual exclusion says nothing about this, so a naive solution either busy-waits wastefully or, worse, allows a producer to overwrite unread data.

Semaphore-based solution. Three semaphores are used together: `mutex` (binary, initialized to 1) protects the buffer's shared indices from concurrent modification; `empty` (counting, initialized to k) counts the number of currently-empty slots; `full` (counting, initialized to 0) counts the number of currently-occupied (consumable) slots.

Producer process:

1. repeat
2. produce an item `nextProduced`
3. `wait(empty)`: wait for a free slot
4. `wait(mutex)`: enter critical section on the buffer
5. add `nextProduced` to the buffer
6. `signal(mutex)`
7. `signal(full)`: one more consumable item now exists
8. forever

Consumer process:

1. repeat
2. `wait(full)`: wait for an item to consume
3. `wait(mutex)`: enter critical section on the buffer
4. remove an item into `nextConsumed`
5. `signal(mutex)`
6. `signal(empty)`: one more free slot now exists
7. consume `nextConsumed`
8. forever

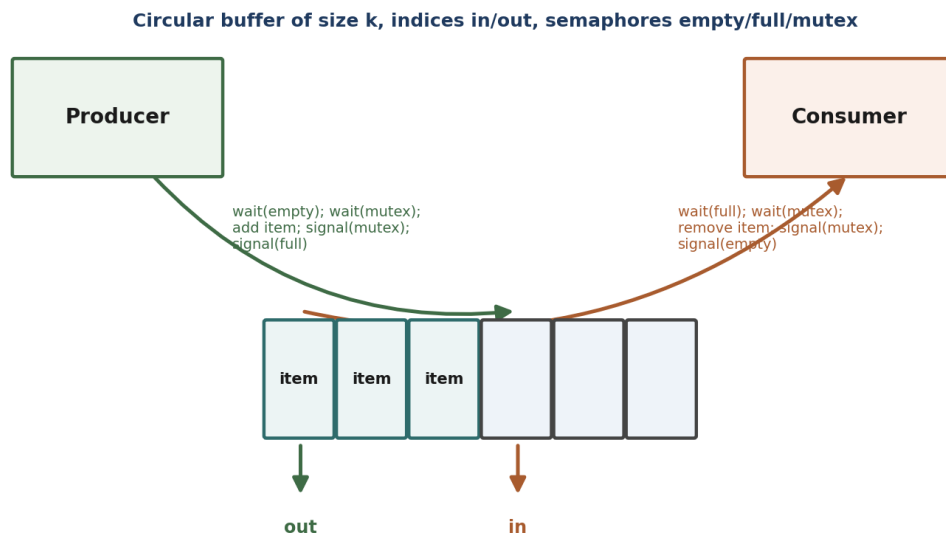


Figure 3.3 — A bounded circular buffer (producer-consumer)

The order of the two `wait()` calls in each process is essential to correctness: the consumer must call `wait(full)` *before* `wait(mutex)`. If this order were reversed, a consumer arriving when the buffer is empty would acquire `mutex` first and then block on `wait(full)`, but it would still be holding `mutex`, permanently locking the producer out of the buffer and deadlocking the system, since the producer can never get `mutex` to add an item and thereby eventually `signal(full)`. The same reasoning requires the producer to call `wait(empty)` before `wait(mutex)`.

3.5.2 The Readers-Writers Problem

The problem. A database (or any shared data object) is accessed by two kinds of processes: **readers**, who only read the data, and **writers**, who read and modify it. Any number of readers may safely access the data simultaneously, since none of them changes it. But a writer requires **exclusive access**: while a writer is active, no other process, reader or writer, may access the data at all.

What goes wrong without synchronization. Without any coordination, a writer could modify the data while a reader is in the middle of reading it, causing the reader to see a data structure that is only half-updated (for example, a reader could see a data structure with a pointer updated but the data it points to not yet updated, following pointers into garbage); or two writers could interleave their modifications and corrupt the data entirely, exactly as in the counter race condition of Section 3.1.

The first (readers-preference) readers-writers problem requires that no reader be kept waiting unless a writer has *already* obtained access to the shared object. In other words, a reader should never have to wait behind a writer merely because that writer arrived and is waiting, if other readers are already in. Shared data:

- semaphore `mutex` = 1: protects the readcount variable.
- semaphore `wrt` = 1: held by a writer for the entire duration of writing, and by the *first* reader to arrive (on behalf of all subsequent readers) until the *last* reader leaves.
- int `readcount` = 0: the number of readers currently reading.

Writer process:

1. `wait(wrt)`
2. writing is performed
3. `signal(wrt)`

Reader process:

1. `wait(mutex)`

2. `readcount = readcount + 1`
3. `if (readcount == 1) wait(wrt):` the first reader locks out writers
4. `signal(mutex)`
5. reading is performed
6. `wait(mutex)`
7. `readcount = readcount - 1`
8. `if (readcount == 0) signal(wrt):` the last reader lets writers back in
9. `signal(mutex)`

Only the *first* reader to arrive (when `readcount` goes from 0 to 1) actually executes `wait(wrt)`, locking writers out on behalf of the whole group of readers that follow; subsequent readers simply increment `readcount` and proceed, without touching `wrt` at all. Symmetrically, only the *last* reader to leave (when `readcount` returns to 0) executes `signal(wrt)`, reopening the resource to writers. This first-readers-preference solution can **starve writers** indefinitely if a steady stream of readers keeps arriving before `readcount` ever drops back to zero.

The second (writers-preference) readers-writers problem flips the priority: once a writer is ready to write, it should be allowed to do so as soon as possible, and no *new* reader may begin reading while a writer is waiting; this can instead **starve readers** if writers arrive frequently enough. Because either strict-preference solution can starve one class of process, practical systems frequently implement a **fair (or "third") variant** that admits readers and writers in roughly the order they arrive (for example, using a FIFO queue that a reader must also check before proceeding, even if `wrt` happens to be free), preventing indefinite starvation of either kind. Real reader-writer locks, provided by many operating systems and languages, generalize this idea directly: a process explicitly acquires the lock in *read mode* (letting it share access with other concurrent readers) or *write mode* (requiring exclusive access), and are especially useful in code with far more readers than writers, since the extra concurrency of allowing simultaneous reads outweighs the greater bookkeeping overhead of a reader-writer lock compared to a plain mutual-exclusion lock.

3.5.3 The Dining Philosophers Problem

The problem. Five philosophers sit around a circular table, alternating between thinking and eating. In the centre of the table is a bowl of rice, and exactly five chopsticks are laid on the table, one between each adjacent pair of philosophers. To eat, a philosopher needs **both** of the two chopsticks adjacent to her (the one to her left and the one to her right); she may pick up only one chopstick at a time, and obviously cannot pick up a chopstick that a neighbour is already holding. Once she has both, she eats without releasing them; when finished, she puts both chopsticks down and resumes thinking.

The dining philosopher's problem matters not because computer scientists particularly enjoy this scenario, but because it is a compact model of a genuinely important and common situation: several processes competing for exclusive access to a fixed pool of shared resources, where poor coordination can leave every process permanently stuck.

Five philosophers, five chopsticks: each philosopher needs both neighbouring chopsticks to eat

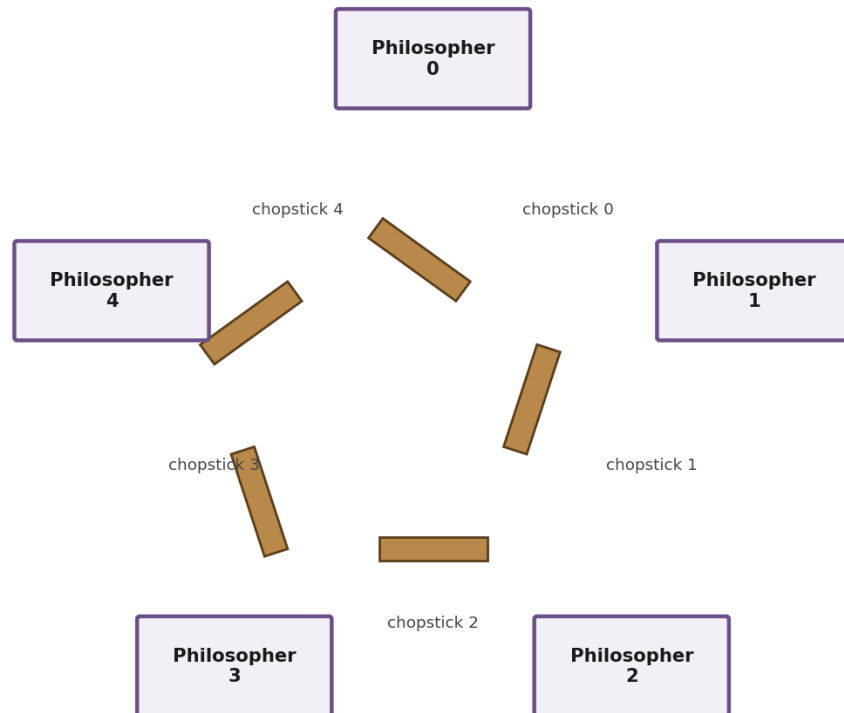


Figure 3.4 — The Dining Philosophers seating arrangement

A naive semaphore solution. Represent each chopstick as a binary semaphore, all initialized to 1:
 semaphore chopstick [5]; (each initialized to 1)

Philosopher i (numbered 0 through 4) executes:

1. repeat{
2. wait(chopstick[i]): pick up left chopstick
3. wait(chopstick [(i + 1) % 5]): pick up right chopstick
-
4. //eat
-
5. & signal(chopstick[i]): put down left chopstick
6. & signal(chopstick [(i + 1) % 5]): put down right chopstick
-
7. //think
-
8. forever

Why this deadlocks. This solution does guarantee that no two neighbouring philosophers ever eat at the same instant (each shares a chopstick with each neighbour, and a chopstick's semaphore admits only one holder). But it must be rejected outright, because it can **deadlock**: suppose all five philosophers become hungry at the same moment and each successfully picks up her *left* chopstick first (each wait(chopstick[i]) succeeds, since every philosopher is requesting a *different* chopstick as her "left" one). Now every chopstick's semaphore has been decremented to 0, and every philosopher's next call, wait(chopstick[(i+1) % 5]) for her right chopstick, blocks forever, because that chopstick is exactly the left chopstick already held by her neighbour, who is itself blocked waiting for its own right chopstick. All five processes now hold one resource each while waiting for another resource held by a neighbour: a **circular wait**, and every one of the four necessary conditions for deadlock (formally defined in Section 3.6) is present at once.

Standard fixes. Several standard remedies break this circular-wait pattern:

1. **Allow at most $n - 1$ philosophers to sit at the table simultaneously** (four, in the five-philosopher case). With only four philosophers able to attempt to eat at once, at least one chopstick is guaranteed to be free among the five, so it is impossible for all four seated philosophers to be simultaneously blocked holding one chopstick each and waiting for another.
2. **Require that a philosopher pick up both chopsticks together, inside a single critical section**, only if both are simultaneously available, never one at a time while holding the first. This removes the possibility of holding one resource while waiting indefinitely for the other.
3. **Asymmetric (odd/even) ordering.** Break the symmetry that causes the circular wait by having philosophers with an **even** index pick up their **left** chopstick first, then their right, while philosophers with an **odd** index pick up their **right** chopstick first, then their left:

Philosopher i , asymmetric version:

1. repeat
 - {
2. if (i is even)
 - {
 - wait(chopstick[i]);
 - wait(chopstick[($i+1$) % 5]);
 - }
3. else
 - {
 - wait(chopstick[($i+1$) % 5]);
 - wait(chopstick[i]);
 - }
4. eat
5. signal(chopstick[i]);
- signal(chopstick[($i+1$) % 5]);
6. think
 - }
7. forever

With five philosophers (an odd number), this pairs each even-numbered philosopher's "first pick" with an odd-numbered neighbour's "first pick" on the *same physical chopstick* somewhere around the table, breaking the perfectly symmetric all-grab-your-left pattern that caused deadlock: it can be shown that at least one philosopher is always guaranteed to obtain both chopsticks and eat, since the resource-acquisition order is no longer identical (a full cycle) for every process.

Even a deadlock-free solution to this problem does not automatically guarantee fairness. A scheme can be deadlock-free yet still allow one unlucky philosopher to be starved indefinitely if her neighbours simply eat more often than she does; a fully starvation-free solution needs additional care (for example, tie-breaking based on how long a philosopher has been waiting), which is beyond what the semaphore solution above addresses.

3.5.4 The Sleeping Barber Problem

The problem. A barbershop has one barber, one barber's chair, and a fixed number, n , of waiting chairs. If no customers are present, the barber sleeps in the barber's chair. An arriving customer must wake the barber if he is asleep; if the barber is busy cutting hair and a waiting chair is free, the new customer sits and waits; if the barber is busy and every waiting chair is already occupied, the customer leaves without a haircut.

What goes wrong without synchronization. Without careful coordination, several races are possible: two arriving customers could both see an "empty" waiting-room count and both try to occupy the same last free chair; a customer could try to wake a barber who is already awake and busy (or, conversely, an arriving customer's wake-up signal could be lost if the barber has not yet gone to sleep, leaving the customer waiting on a barber who never realizes anyone arrived); and the barber could finish a haircut and check for waiting customers at the exact moment a new customer is in the middle of incrementing the count, missing that customer entirely.

Semaphore-based solution. Three semaphores and one shared counter suffice:

- semaphore customers = 0: counts waiting customers; the barber sleeps by blocking on this.
- semaphore barber = 0: signals that the barber is ready to cut hair; a customer waits on this before sitting in the barber's chair.
- semaphore mutex = 1: protects the shared waiting counter.
- int waiting = 0: number of customers currently in the waiting room (not counting the one in the barber's chair).

Barber process:

1. repeat
2. wait(customers): sleep until a customer arrives
3. wait(mutex)
4. waiting = waiting - 1: this customer is leaving the waiting room for the chair
5. signal(barber): tell this customer the barber is ready
6. signal(mutex)
7. cut hair
8. forever

Customer process:

1. wait(mutex)
2. if (waiting < n)
 - {
 - 3. waiting = waiting + 1
 - 4. signal(customers): wake the barber (or register as one more waiting customer)
 - 5. signal(mutex)
 - 6. wait(barber): wait for the barber to be free for *this* customer
 - 7. get haircut
 - 8. }
- else
 - {
 - 9. signal(mutex): waiting room full; leave without a haircut
 - }

The counting semaphore customers neatly plays the role of both "wake the sleeping barber" and "count how many customers are waiting for a turn" at once: the barber's wait(customers) blocks (sleeps) precisely when no customers exist, and is unblocked the instant signal(customers) is called by an arriving customer, with no wake-up ever being lost, because the semaphore's internal state (not a fragile boolean flag) records exactly how many pending "wake the barber" signals remain outstanding. The mutex semaphore ensures that the waiting counter itself, checked and updated by potentially many customers arriving at once as well as by the barber, is never corrupted by a race condition of the kind described in Section 3.1.

3.5.5 Comparing the Four Problems

Table 3.5 — The four classical synchronization problems compared

Problem	Core resource being coordinated	Key synchronization idea	Central risk if unsynchronized
Bounded buffer (producer-consumer)	A fixed-size buffer of k slots	Two counting semaphores (empty, full) plus one mutex	Overwriting unread data, or reading stale/garbage data
Readers-writers	A shared database/data structure	Reader count (readcount) gates a single wrt semaphore shared with writers	A writer modifying data while a reader (or another writer) is mid-access
Dining philosophers	A shared pool of n identical exclusive resources (chopsticks)	Ordering discipline (asymmetric acquisition, or n-1 seating) to break circular wait	Deadlock: the classic four-condition circular wait
Sleeping barber	A single server (barber) plus a bounded waiting room	Counting semaphore doubles as both a wake signal and a waiting-customer count	Lost wake-up signals, or races on the shared waiting-room count

CHECK YOUR PROGRESS 3.5

1. In the bounded-buffer solution, why must the consumer call `wait(full)` before `wait(mutex)`, rather than the other way around?
2. State the first readers-writers problem's requirement in one sentence, and name the class of process it can starve.
3. Explain precisely how the dining philosophers can deadlock if every philosopher picks up her left chopstick first, and name one standard fix.
4. In the sleeping barber problem, what role does the semaphore customers play, and why does a plain boolean "customer waiting" flag risk losing a wake-up signal that a counting semaphore does not?
5. Which of the four classical problems is specifically a model of the risk of deadlock, as opposed to a risk of a race condition or lost update?

3.6 DEADLOCKS: CONCEPTS AND CHARACTERIZATION

3.6.1 The Resource-Allocation Model

In a multiprogramming system, many processes compete for a finite pool of resources: CPU cycles, memory, files, printers, and other I/O devices. Resources are grouped into **resource types**, each with some fixed number of identical **instances** (for example, a system with two CPUs has a resource type "CPU" with two instances; a system with five printers has a resource type "printer" with five instances). Because instances of the same type are, by definition, interchangeable, any request for "a printer" is satisfied by allocating *any* free printer instance.

Under the normal mode of operation, a process must use a resource through exactly three steps:

1. **Request.** The process requests the resource. If it cannot be granted immediately (because the resource, or all its instances, are currently held by other processes), the requesting process must wait.
2. **Use.** The process operates on the resource (for example, prints on the printer it was granted).
3. **Release.** The process releases the resource back to the system.

IMPORTANT — Definition — Deadlock

A set of processes is in a **deadlocked state** when every process in the set is waiting for an event (specifically, the release of a resource) that can only be caused by another process that is itself

in the same waiting set. No process in a deadlock can ever proceed, because each one is waiting on a resource permanently held by another process that is, in turn, equally stuck.

EXAMPLE — A minimal two-process deadlock

Process P1 has been allocated a printer and is now attempting to open file F. Process P2 already holds exclusive access to file F and now requests the printer. Neither process can proceed: P1 is waiting for F, which only P2 can release, and P2 is waiting for the printer, which only P1 can release. Each process holds exactly the resource the other one needs, and neither will ever voluntarily give up what it holds before it acquires what it is waiting for.

3.6.2 The Four Necessary Conditions

A deadlock can arise only if **all four** of the following conditions hold **simultaneously**. This is an important and often-tested point: none of the four conditions alone is sufficient to cause a deadlock, and removing even one of the four is enough to make deadlock impossible in that system. That is exactly what deadlock *prevention* exploits in Section 3.7.

1. **Mutual exclusion.** At least one resource must be held in a non-sharable mode: only one process at a time can use it. (A read-only file, by contrast, can be shared by many readers simultaneously and can never be part of a deadlock's resource chain on its own.)
2. **Hold and wait.** A process must be holding at least one resource while simultaneously waiting to acquire additional resources that are currently held by other processes.
3. **No preemption.** Resources cannot be forcibly taken away from the process holding them; a resource can only be released voluntarily, by the process that holds it, after that process has finished using it.
4. **Circular wait.** There must exist a set of waiting processes $\{P_0, P_1, \dots, P_n\}$ such that P_0 is waiting for a resource held by P_1 , P_1 is waiting for a resource held by P_2 , and so on, until P_n is waiting for a resource held by P_0 : closing the circle.

These four conditions are not fully independent of one another: the first three are individually necessary but not, by themselves, sufficient for deadlock, whereas the fourth condition, circular wait, actually **implies** the first three (a genuine circular wait cannot exist unless the resources involved are held exclusively, are being held while further resources are awaited, and cannot be preempted away) and is therefore both **necessary and sufficient**. In practice, however, it is usually easier to reason about, and to attack, the first three conditions individually (as Section 3.7 does), because each corresponds to a specific, identifiable design decision a system can make, whereas "does a circular wait exist" is a global, whole-system property that is comparatively expensive to check directly.

3.6.3 The Resource-Allocation Graph

Deadlocks can be represented precisely using a directed graph called the **system resource-allocation graph (RAG)**. The graph's vertex set V is split into two disjoint kinds of vertices:

- $P = \{P_1, P_2, \dots, P_n\}$: one vertex (conventionally a circle) for every process in the system.
- $R = \{R_1, R_2, \dots, R_m\}$: one vertex (conventionally a rectangle, with one dot inside it per instance of that resource type) for every resource type in the system.

Two kinds of directed edges appear in the graph:

- A **request edge**, $P_i \rightarrow R_j$, means process P_i has requested an instance of resource type R_j and is currently waiting for it.
- An **assignment edge**, $R_j \rightarrow P_i$, means an instance of resource type R_j has already been allocated to process P_i .

When a process requests a resource, a request edge is inserted; the instant that request is granted, the request edge is instantaneously transformed into an assignment edge (from a specific instance's dot to that process); and when the process later releases the resource, the assignment edge is removed.

P1 holds R1 and wants R2; P2 holds R2 and wants R1 — a cycle, hence deadlock

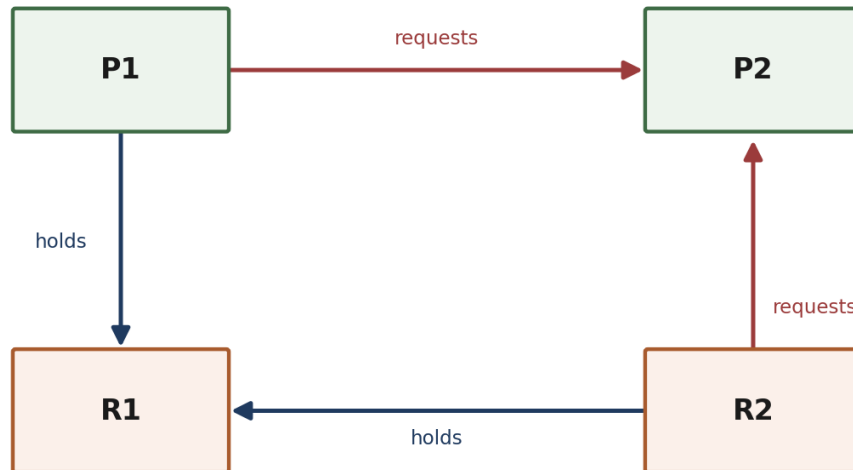


Figure 3.5 — A resource-allocation graph containing a cycle

The cycle-detection rule.

- If the resource-allocation graph contains **no cycle**, the system is **not** deadlocked. This direction of the rule holds unconditionally, for any number of instances per resource type.
- If the graph **does** contain a cycle, the conclusion depends on how many instances each resource type in the cycle has:
 - If **every resource type involved in the cycle has exactly one instance**, a cycle is both necessary and **sufficient** for deadlock: the system is definitely deadlocked.
 - If **some resource type involved in the cycle has two or more instances**, a cycle is necessary but **not sufficient**: the system *may* be deadlocked, but is not guaranteed to be, because it remains possible for a process outside the cycle to release an additional free instance of that multi-instance resource type, which can then satisfy one of the waiting processes in the cycle and break it.

EXAMPLE — A cycle without a deadlock (multi-instance resource)

Suppose resource type R2 has two instances. P3 holds one instance of R2 and is waiting for R1 (held by P1); P1 holds R1 and is waiting for R2 (currently, both instances are held, one by P3 and one by P4); P4 holds the second instance of R2 but is *not* waiting for anything further. A cycle P1 → R2 → P3 → R1 → P1 exists in the graph, yet the system is **not** deadlocked: P4 will eventually finish and release its instance of R2 (with no wait of its own to prevent this), at which point that freed instance can be given to P1, breaking the cycle. This illustrates precisely why a cycle is only necessary, not sufficient, once any resource type in the cycle has more than one instance.

IMPORTANT — Cycle versus circular wait

A **cycle** in the resource-allocation graph is a *necessary* condition for deadlock. A **circular wait** (Section 3.6.2's fourth necessary condition) is a *necessary and sufficient* condition for deadlock. The two coincide exactly when every resource type on the cycle has a single instance; with multi-instance resource types, a graph cycle does not, by itself, guarantee that an actual circular wait exists, one where every process on the cycle is genuinely stuck and none can ever be satisfied by a resource from outside the cycle.

CHECK YOUR PROGRESS 3.6

1. Name the four necessary conditions for deadlock, and state which one is, on its own, both necessary and sufficient.
2. Why can a read-only file never be part of the mutual-exclusion condition that leads to deadlock?
3. In a resource-allocation graph, what does a request edge represent, as opposed to an assignment edge?
4. If a resource-allocation graph contains a cycle, under what precise circumstance can you conclude the system is definitely deadlocked, and under what circumstance is deadlock merely possible?
5. Explain the difference between a "cycle" in the resource-allocation graph and a genuine "circular wait."

3.7 DEADLOCK PREVENTION AND AVOIDANCE

There are, broadly, three strategies for dealing with deadlock: **ensure it can never occur** (either by prevention or avoidance, covered in this section), **let it occur and then detect and recover from it** (Section 3.9), or simply **ignore the problem**, on the pragmatic grounds that deadlocks are rare enough in practice, and the cost of preventing or detecting them high enough, that most general-purpose operating systems (historically including UNIX) accept the small residual risk and rely on a reboot if it ever actually happens. This section covers the first strategy's two variants.

3.7.1 Deadlock Prevention

Deadlock prevention works by guaranteeing that at least one of the four necessary conditions from Section 3.6.2 can *never* hold, by placing structural constraints on how processes are allowed to request resources. Because all four conditions must hold simultaneously for deadlock to occur, permanently ruling out even one is enough to make deadlock structurally impossible. But each of the four comes with a real practical cost, since the same conditions being denied are often exactly what makes a system flexible and efficient.

- **Attacking mutual exclusion.** In principle, this condition is denied by never requiring exclusive access to any resource. In practice this is usually **not achievable**, because many resources (a printer, a write-lock on a file) are intrinsically non-sharable: the underlying hardware or correctness requirement itself demands exclusive use, and no scheduling policy can change that. (Sharable resources, such as a read-only file, never participate in a deadlock's mutual-exclusion condition in the first place, so there is nothing to "prevent" for them.)
- **Attacking hold-and-wait.** This can be denied in one of two ways: require every process to request and be granted *all* the resources it will ever need before it begins execution (so it never holds one resource while waiting for another); or require that a process may only request resources when it is currently holding none at all (forcing it to release everything

before asking for more). Both variants have a real cost: resource utilization can be low, because a process may hold resources it does not currently need for long stretches (in the all-at-once variant) or may repeatedly release and immediately need to reacquire the same resource (in the second variant); and **starvation** becomes possible, since a process needing many popular resources may wait indefinitely for all of them to become simultaneously available.

- **Attacking no preemption.** This condition is denied by allowing resources to be forcibly taken from a process: if a process holding some resources requests another that cannot be immediately granted, all resources currently held by that process are preempted (taken away) and added to the list of resources it is waiting for; the process is only restarted once it can reacquire both its old resources and the newly requested one. This works well for resources whose state can be easily saved and restored (such as CPU registers or memory pages) but is impractical or impossible for many resources (you cannot cleanly "preempt" a half-finished print job or a file mid-write without real cost or complexity).
- **Attacking circular wait.** This condition is denied by imposing a **total ordering** on all resource types (numbering every resource type from 1 to m) and requiring that every process request resources only in strictly increasing order of that numbering: a process holding an instance of resource type R_i may only subsequently request resource types numbered higher than i . Since a circular wait would require some process in the cycle to request a lower-numbered resource type while holding a higher-numbered one, and the ordering rule forbids exactly that, a circular wait becomes structurally impossible. The practical cost is a loss of flexibility: application code must be written (or restructured) to respect a system-wide numbering it may not naturally follow, and a process that needs a low-numbered resource type only later in its logic may still be forced to acquire it earlier than it would prefer, holding it needlessly long.

Table 3.6 — Deadlock prevention strategies and their cost

Necessary condition attacked	Typical prevention technique	Practical cost
Mutual exclusion	Make resources sharable	Rarely feasible: most contended resources are intrinsically non-sharable
Hold and wait	Request all resources upfront, or hold none while requesting	Low resource utilization; starvation possible
No preemption	Preempt a waiting process's held resources when a request cannot be granted	Only workable for easily saved/restored resource state; costly or impossible otherwise
Circular wait	Impose a total ordering on resource types; request in increasing order	Loss of programming flexibility; resources may be held longer than needed

3.7.2 Deadlock Avoidance

Deadlock avoidance takes a fundamentally different approach from prevention. Rather than permanently ruling out one of the four necessary conditions by structural constraint (accepting reduced resource utilization or programming flexibility as the price), avoidance requires additional *information* about how each process will request resources in the future, and uses that information to dynamically decide, request by request, whether granting a particular request is safe. This allows much higher resource utilization than prevention typically permits, at the cost of requiring processes to declare their future resource needs in advance.

The simplest and most widely used model requires every process to declare, in advance, the **maximum** number of instances of each resource type it may ever need over its lifetime. Given this information, together with the current allocation state (which resources are currently held by which process) and the currently available resources, a deadlock-avoidance algorithm dynamically

examines the resource-allocation state before granting any request, and only grants it if doing so cannot possibly lead to a circular wait later on.

IMPORTANT — Definition — Safe State

A resource-allocation state is a **safe state** if there exists at least one **safe sequence**: an ordering of *all* currently existing processes, $\langle P_1, P_2, \dots, P_n \rangle$, such that for every process P_i in the sequence, the resources P_i may still request can be satisfied using the resources currently available **plus** the resources currently held by all processes P_j that precede it in the sequence ($j < i$). A state that is not safe is called an **unsafe state**.

The logic behind a safe sequence is straightforward: if P_i 's remaining resource needs are not immediately available, P_i can simply wait until every process before it in the sequence (P_1 through P_{i-1}) has finished, released its resources, and terminated. At that point P_i is guaranteed to have enough available resources to complete, since the sequence was constructed to guarantee exactly that. Once P_i itself finishes and releases its resources, P_{i+1} is, by the same reasoning, guaranteed to be able to complete next, and so on down the sequence. As each process terminates and releases what it was holding, the pool of available resources only grows (or, at worst, stays the same before growing again), so a process that could be satisfied at its position in the sequence remains satisfiable when its turn actually arrives.

A crucial and frequently misunderstood pair of relationships holds between safe states and deadlock:

- **A safe state is never a deadlocked state.** By definition, a safe state guarantees every process can eventually complete via the safe sequence.
- **A deadlocked state is always an unsafe state** (the converse fails; see below).
- **Not every unsafe state is a deadlock.** An unsafe state only means the *system itself* can no longer guarantee that deadlock will be avoided. It may still be that no deadlock actually happens, if the processes involved happen to request and release resources in a lucky order. But the system can no longer be sure of this, because there exists at least one bad sequence of future requests that would lead to deadlock.

Deadlock avoidance works, therefore, by having the operating system grant a resource request only if doing so would leave the system in a state that is still provably safe, and making the process wait otherwise, even if the resources requested happen to be currently available. This is strictly more conservative than merely checking "are the resources free right now," which is exactly why avoidance requires the extra maximum-claim information that prevention does not.

The resource-allocation graph algorithm extends the RAG of Section 3.6.3 with a new kind of edge for exactly this purpose, but only when every resource type has a single instance. A **claim edge**, $P_i \dashrightarrow R_j$ (drawn dashed), indicates that P_i may request R_j at some point in the future; when P_i actually makes that request, the claim edge is converted into a solid request edge, and when the request is granted, into an assignment edge (converting back to a claim edge once released). All of a process's claim edges must be declared before it starts running. A request can be granted only if converting its request edge into an assignment edge would **not create a cycle** in the resulting graph. This is checked using an ordinary cycle-detection algorithm, which requires on the order of n^2 operations for n processes. If granting the request would create a cycle, the system instead makes the requesting process wait, precisely because granting it would move the system into an unsafe state (with single-instance resources, recall from Section 3.6.3, a cycle among single-instance resources is both necessary and sufficient for actual deadlock, so refusing to ever create one is exactly equivalent to always staying in a safe state).

NOTE — Prevention versus avoidance, contrasted

Prevention constrains *how* requests may be made at all (for example, forcing a total order on resource acquisition), permanently ruling out one necessary condition regardless of the actual current state of the system. **Avoidance** instead constrains *when* an otherwise-permitted request is granted, by dynamically checking, using declared maximum future needs, whether granting it right now would leave the system in a safe state. A request for a currently-available resource might still be refused (the requesting process kept waiting) purely because granting it would be unsafe, even though prevention's structural rules would have allowed it outright.

The resource-allocation graph algorithm does not extend directly to resource types with multiple instances, since a cycle among multi-instance resources is not, by itself, either necessary or sufficient for deadlock (Section 3.6.3). Handling the general, multi-instance case is exactly the job of the **Banker's Algorithm**, covered in full in Section 3.8.

CHECK YOUR PROGRESS 3.7

1. Name the three broad strategies for dealing with deadlock at the level of overall system policy.
2. Which necessary condition does imposing a total ordering on resource types attack, and how does it prevent a circular wait?
3. Define a safe state, and explain the relationship between a safe state, an unsafe state, and a deadlocked state.
4. Contrast deadlock prevention with deadlock avoidance in terms of what information each requires and when each restricts a process.
5. In the resource-allocation graph algorithm for deadlock avoidance, what does a claim edge represent, and under what condition is a pending request actually granted?

3.8 THE BANKER'S ALGORITHM

3.8.1 Overview and Data Structures

IMPORTANT — Definition — The Banker's Algorithm

The **Banker's Algorithm** is a **deadlock-avoidance** algorithm applicable to resource systems with **multiple instances** of each resource type. It is named by analogy with a banker who must decide whether to extend a loan without ever committing so much cash that the bank could no longer, in principle, satisfy every customer's maximum possible demand. The algorithm never grants a resource request that would leave the system unable to guarantee every process can eventually finish.

Every new process entering the system must declare the **maximum** number of instances of each resource type it may ever request, and this maximum may not exceed the total number of instances the system actually has. Whenever a process requests a set of resources, the system uses the algorithm to check whether granting the request would leave the system in a **safe state** (Section 3.7.2); only if it would, are the resources actually allocated. Otherwise the requesting process must wait, even if the resources it asked for happen to be free right now.

Let n be the number of processes and m the number of resource types. Four data structures describe the system's state at any moment:

- **Available:** a vector of length m . $\text{Available}[j] = k$ means k instances of resource type R_j are currently free (unallocated).

- **Max:** an $n \times m$ matrix. $\text{Max}[i][j] = k$ means process P_i may request at most k instances of resource type R_j over its entire lifetime.
- **Allocation:** an $n \times m$ matrix. $\text{Allocation}[i][j] = k$ means process P_i currently holds k instances of resource type R_j .
- **Need:** an $n \times m$ matrix, *derived* from the other two rather than given directly: $\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$, the number of additional instances of R_j that P_i may still request to complete its task.

3.8.2 The Safety Algorithm

The **safety algorithm** determines whether a given resource-allocation state is safe, by attempting to construct a safe sequence (Section 3.7.2). It underlies the Banker's Algorithm proper (Section 3.8.3) as the actual test of safety used after pretending to grant a request.

1. Let *Work* be a vector of length m , and *Finish* a vector of length n . Initialize *Work* = *Available*, and *Finish*[i] = false for all $i = 0, 1, \dots, n-1$.
2. Find an index i such that both:
 - a. *Finish*[i] == false, and
 - b. $\text{Need}[i] \leq \text{Work}$ (every component of process P_i 's *Need* row is less than or equal to the corresponding component of *Work*).

If no such i exists, go to step 4.

3. $\text{Work} = \text{Work} + \text{Allocation}[i]$ (reclaim P_i 's currently-held resources, since P_i is now assumed to run to completion and release them); set *Finish*[i] = true; go back to step 2.
4. If *Finish*[i] == true for all i , the system is in a **safe state**, and the order in which processes were marked finished is a valid safe sequence. If any *Finish*[i] remains false, the state is **unsafe**.

3.8.3 The Resource-Request Algorithm

The **resource-request algorithm** is the Banker's Algorithm proper: it decides, for an actual incoming request from a process P_i , whether the request can be safely granted immediately. Let *Request_i* be the request vector for P_i , where *Request_i*[j] = k means P_i wants k more instances of resource type R_j .

1. If $\text{Request}_i \leq \text{Need}_i$, proceed to step 2. Otherwise, raise an error: the process has requested more than the maximum it originally declared.
2. If $\text{Request}_i \leq \text{Available}$, proceed to step 3. Otherwise, P_i must wait, since the resources it wants are not currently free.
3. **Tentatively** grant the request, updating the state as if it had been allocated:
 - a. $\text{Available} = \text{Available} - \text{Request}_i$
 - b. $\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$
 - c. $\text{Need}_i = \text{Need}_i - \text{Request}_i$
4. Run the **safety algorithm** (Section 3.8.2) on this new, tentative state.
 - If the resulting state is **safe**, the request is granted for real, and the new state (from step 3) becomes the system's actual current state.
 - If the resulting state is **unsafe**, the request **cannot** be granted yet: P_i must wait, and the system is rolled back to the state it was in *before* step 3 was applied.

Steps 1 and 2 are simple consistency checks and are not, by themselves, the safety test; the real decision of whether to grant the request is made entirely in step 4, by re-running the full safety algorithm on the hypothetical post-grant state.

3.8.4 A Fully Worked Numeric Example

Consider a system with **five processes**, P₀ through P₄, and **three resource types**: A (10 instances total), B (5 instances total), and C (7 instances total). The snapshot of the system at time T₀ is:

Table 3.7 — Banker's Algorithm worked example (time T₀)

Process	Allocation (A,B,C)	Max (A,B,C)	Need (A,B,C) = Max – Allocation
P ₀	0, 1, 0	7, 5, 3	7, 4, 3
P ₁	2, 0, 0	3, 2, 2	1, 2, 2
P ₂	3, 0, 2	9, 0, 2	6, 0, 0
P ₃	2, 1, 1	2, 2, 2	0, 1, 1
P ₄	0, 0, 2	4, 3, 3	4, 3, 1

The total instances allocated are A: $0+2+3+2+0 = 7$, B: $1+0+0+1+0 = 2$, C: $0+0+2+1+2 = 5$. Since the system totals are A=10, B=5, C=7, the **Available** vector is $(10-7, 5-2, 7-5) = (3, 3, 2)$.

Step 1 — Is the T₀ state safe? Run the safety algorithm with Work = Available = (3, 3, 2) and all Finish[i] = false:

1. Check P₀: Need[P₀] = (7,4,3). Is (7,4,3) ≤ (3,3,2)? No (7 > 3). Skip.
2. Check P₁: Need[P₁] = (1,2,2). Is (1,2,2) ≤ (3,3,2)? Yes. Grant conceptually: Work = (3,3,2) + Allocation[P₁](2,0,0) = (5,3,2). Mark Finish[P₁] = true.
3. Check P₂: Need[P₂] = (6,0,0). Is (6,0,0) ≤ (5,3,2)? No (6 > 5). Skip.
4. Check P₃: Need[P₃] = (0,1,1). Is (0,1,1) ≤ (5,3,2)? Yes. Work = (5,3,2) + Allocation[P₃](2,1,1) = (7,4,3). Mark Finish[P₃] = true.
5. Check P₄: Need[P₄] = (4,3,1). Is (4,3,1) ≤ (7,4,3)? Yes. Work = (7,4,3) + Allocation[P₄](0,0,2) = (7,4,5). Mark Finish[P₄] = true.
6. Check P₂ again: Need[P₂] = (6,0,0). Is (6,0,0) ≤ (7,4,5)? Yes. Work = (7,4,5) + Allocation[P₂](3,0,2) = (10,4,7). Mark Finish[P₂] = true.
7. Check P₀ again: Need[P₀] = (7,4,3). Is (7,4,3) ≤ (10,4,7)? Yes. Work = (10,4,7) + Allocation[P₀](0,1,0) = (10,5,7). Mark Finish[P₀] = true.

All five processes now have Finish[i] = true. **The T₀ state is safe**, with safe sequence **<P₁, P₃, P₄, P₂, P₀>**. (As a sanity check, the final Work vector, (10,5,7), correctly equals the system's total resource counts, since every process has hypothetically finished and released everything it held.)

Step 2 — Can P₁'s request for (1, 0, 2) be granted? Check the resource-request algorithm:

1. Is Request[P₁](1,0,2) ≤ Need[P₁](1,2,2)? Yes (1≤1, 0≤2, 2≤2).
2. Is Request[P₁](1,0,2) ≤ Available(3,3,2)? Yes (1≤3, 0≤3, 2≤2).
3. Tentatively grant:
 - Available = (3,3,2) - (1,0,2) = (2,3,0)
 - Allocation[P₁] = (2,0,0) + (1,0,2) = (3,0,2)
 - Need[P₁] = (1,2,2) - (1,0,2) = (0,2,0)
4. Re-run the safety algorithm on this new state. Work = (2,3,0):
 - P₁: Need = (0,2,0) ≤ (2,3,0)? Yes. Work = (2,3,0)+(3,0,2) = (5,3,2). Finish[P₁]=true.
 - P₃: Need = (0,1,1) ≤ (5,3,2)? Yes. Work = (5,3,2)+(2,1,1) = (7,4,3). Finish[P₃]=true.
 - P₄: Need = (4,3,1) ≤ (7,4,3)? Yes. Work = (7,4,3)+(0,0,2) = (7,4,5). Finish[P₄]=true.
 - P₀: Need = (7,4,3) ≤ (7,4,5)? Yes. Work = (7,4,5)+(0,1,0) = (7,5,5). Finish[P₀]=true.

- P2: Need = (6,0,0) <= (7,5,5)? Yes. Work = (7,5,5)+(3,0,2) = (10,5,7).
Finish[P2]=true.

All processes finish: the state is **safe**, with sequence `<P1, P3, P4, P0, P2>`. **P1's request is therefore granted**, and the system's actual state is now updated to reflect it: Available = (2,3,0), Allocation[P1] = (3,0,2), Need[P1] = (0,2,0).

Step 3 — Can P4's subsequent request for (3, 3, 0) be granted? (Evaluated against the state reached after P1's request above, where Available = (2,3,0) and Need[P4] = (4,3,1).)

1. Is Request[P4](3,3,0) <= Need[P4](4,3,1)? Yes (3<=4, 3<=3, 0<=1).
2. Is Request[P4](3,3,0) <= Available(2,3,0)? **No**, the first component fails: 3 > 2.

Since step 2 fails, P4 **must wait**; the request cannot be granted regardless of what the safety algorithm would say, simply because the system does not currently hold enough free instances of resource A to satisfy it.

Step 4 — Can P0's request for (0, 2, 0) be granted? (Evaluated against the same state as Step 3: Available = (2,3,0), Need[P0] = (7,4,3), Allocation[P0] = (0,1,0).)

1. Is Request[P0](0,2,0) <= Need[P0](7,4,3)? Yes.
2. Is Request[P0](0,2,0) <= Available(2,3,0)? Yes (0<=2, 2<=3, 0<=0).
3. Tentatively grant: Available = (2,3,0)-(0,2,0) = (2,1,0); Allocation[P0] = (0,1,0)+(0,2,0) = (0,3,0); Need[P0] = (7,4,3)-(0,2,0) = (7,2,3).
4. Re-run the safety algorithm. Work = (2,1,0):
 - P0: Need(7,2,3) <= (2,1,0)? No.
 - P1: Need(0,2,0) <= (2,1,0)? No, second component fails (2 > 1).
 - P2: Need(6,0,0) <= (2,1,0)? No.
 - P3: Need(0,1,1) <= (2,1,0)? No, third component fails (1 > 0).
 - P4: Need(4,3,1) <= (2,1,0)? No.

No process's Need can be satisfied by the current Work: every one fails, and no further progress is possible. **This tentative state is unsafe. P0's request for (0, 2, 0) must therefore be refused for now**, and the system is rolled back to Available = (2,3,0), exactly as it was before this request was considered; P0 must wait, even though the resources it asked for were, in isolation, currently available.

EXAMPLE — Why P0's request is refused despite the resources being free

This last case is the one students most often get wrong: it is tempting to reason "Available had enough, so grant it." But the resource-request algorithm's whole point is to look *beyond* the immediate request and check whether granting it could strand the system with no safe way forward. Here, granting P0's modest-looking request for (0, 2, 0) would consume exactly the B-instances that P1 (in this state) still needs to finish, while leaving no process able to proceed at all: a genuinely unsafe state, even though no actual deadlock exists *yet*. Deadlock avoidance is deliberately conservative: it refuses any request that could possibly lead to deadlock later, not merely requests that would deadlock the system immediately.

3.8.5 Limitations of the Banker's Algorithm

Despite its correctness, the Banker's Algorithm has real practical limitations that limit its use in general-purpose operating systems:

- It assumes a **fixed number of resource instances**; it does not gracefully handle resources that appear or disappear dynamically at run time (a hot-swappable USB device being connected or removed, for instance).

- It assumes a **fixed number of processes**; most interactive, general-purpose multiprogramming systems have processes constantly being created and terminated, which the algorithm does not natively accommodate.
- It assumes that the system will grant every legitimate request, and that every process will eventually return all its resources, **within a finite time**: an assumption that does not always hold in practice (a buggy or malicious process could hold resources indefinitely).
- It requires every process to **declare its maximum resource needs in advance**, which is often unrealistic: many real programs cannot predict their peak resource usage before they run, and requiring this declaration constrains how software can be written.

These limitations are exactly why deadlock avoidance via the Banker's Algorithm, while an important theoretical and exam topic, is comparatively rare in general-purpose commercial operating systems, which more commonly either accept the small risk of deadlock (the "ignore it" policy) or use targeted prevention techniques for specific resource classes.

CHECK YOUR PROGRESS 3.8

1. What is the Banker's Algorithm, and to what class of resource-allocation systems does it apply that the resource-allocation graph algorithm cannot handle?
2. Define the four data structures used by the Banker's Algorithm (Available, Max, Allocation, Need), and state the formula relating Need to the other two.
3. In the safety algorithm, what does it mean for $\text{Need}[i] \leq \text{Work}$ to hold, and what happens to Work once such a process is found?
4. Using the T0 snapshot in this section, what is the Available vector, and what is one valid safe sequence?
5. Explain, in your own words, why P0's request for (0,2,0) is refused in the worked example even though the individual resource amounts requested were all currently available.

3.9 DEADLOCK DETECTION AND RECOVERY

If a system employs neither deadlock prevention (Section 3.7.1) nor deadlock avoidance (Section 3.7.2 and 3.8), a deadlock may genuinely occur. In this case the system needs two additional capabilities: an **algorithm to detect** that a deadlock has actually happened, and a **method to recover** from it once detected.

3.9.1 Detection with a Single Instance per Resource Type

When every resource type has exactly one instance, deadlock detection reduces to cycle detection in a simplified graph called a **wait-for graph**: a graph whose vertices are only the processes (the resource nodes are eliminated), with an edge $P_i \rightarrow P_j$ drawn whenever P_i is waiting for a resource currently held by P_j . This wait-for graph is obtained directly from the resource-allocation graph by collapsing each $P_i \rightarrow R_k \rightarrow P_j$ path (request edge followed by assignment edge) into a single direct edge $P_i \rightarrow P_j$. The system periodically invokes a cycle-detection algorithm (costing on the order of n^2 operations for n processes) on this graph; a cycle in the wait-for graph is, in this single-instance case, both necessary and sufficient for deadlock, exactly matching the resource-allocation graph rule of Section 3.6.3.

3.9.2 Detection with Multiple Instances of a Resource Type

For the general case of multiple instances per resource type, the wait-for graph approach does not apply, and a detection algorithm structurally very close to the Banker's Algorithm's safety check (Section 3.8.2) is used instead. It uses three data structures:

- **Available**: a vector of length m : the number of currently free instances of each resource type.

- **Allocation:** an $n \times m$ matrix: the number of instances of each resource type currently allocated to each process.
- **Request:** an $n \times m$ matrix: $\text{Request}[i][j] = k$ means process P_i is *currently* requesting k more instances of resource type R_j right now.

Note carefully that **Request** here plays a role analogous to, but distinct from, the **Max** matrix of the Banker's Algorithm: Max records a process's total declared *maximum future* claim over its lifetime, whereas Request here records only its **current, outstanding** request at this instant, a strictly less demanding and more immediately verifiable piece of information.

The detection algorithm:

1. Let Work be a vector of length m , and Finish a vector of length n . Initialize $\text{Work} = \text{Available}$. For each process i : if $\text{Allocation}[i]$ is not the zero vector, set $\text{Finish}[i] = \text{false}$; otherwise (the process holds nothing), set $\text{Finish}[i] = \text{true}$.
2. Find an index i such that both:
 - a. $\text{Finish}[i] == \text{false}$, and
 - b. $\text{Request}[i] \leq \text{Work}$.

If no such i exists, go to step 4.

3. $\text{Work} = \text{Work} + \text{Allocation}[i]$; set $\text{Finish}[i] = \text{true}$; go back to step 2.
4. If $\text{Finish}[i] == \text{false}$ for some i , the system is in a **deadlocked state**, and every process P_i with $\text{Finish}[i] == \text{false}$ is one of the deadlocked processes.

This algorithm requires on the order of $m \times n^2$ operations to determine whether the system is deadlocked, where m is the number of resource types and n the number of processes.

EXAMPLE — Worked detection example

Consider five processes, P_0 through P_4 , and three resource types: A (7 instances), B (2 instances), C (6 instances). At time T_0 :

Table 3.8 — Deadlock detection worked example

Process	Allocation (A,B,C)	Request (A,B,C)
P_0	0, 1, 0	0, 0, 0
P_1	2, 0, 0	2, 0, 2
P_2	3, 0, 3	0, 0, 0
P_3	2, 1, 1	1, 0, 0
P_4	0, 0, 2	0, 0, 2

Total allocated: A: $0+2+3+2+0=7$, B: $1+0+0+1+0=2$, C: $0+0+3+1+2=6$, matching the system totals exactly, so $\text{Available} = (0,0,0)$.

Running the detection algorithm with $\text{Work} = (0,0,0)$:

1. P_0 : $\text{Request}(0,0,0) \leq (0,0,0)$? Yes. $\text{Work} = (0,0,0) + \text{Allocation}[P_0](0,1,0) = (0,1,0)$. $\text{Finish}[P_0]=\text{true}$.
2. P_2 : $\text{Request}(0,0,0) \leq (0,1,0)$? Yes. $\text{Work} = (0,1,0) + \text{Allocation}[P_2](3,0,3) = (3,1,3)$. $\text{Finish}[P_2]=\text{true}$.
3. P_3 : $\text{Request}(1,0,0) \leq (3,1,3)$? Yes. $\text{Work} = (3,1,3) + \text{Allocation}[P_3](2,1,1) = (5,2,4)$. $\text{Finish}[P_3]=\text{true}$.
4. P_1 : $\text{Request}(2,0,2) \leq (5,2,4)$? Yes. $\text{Work} = (5,2,4) + \text{Allocation}[P_1](2,0,0) = (7,2,4)$. $\text{Finish}[P_1]=\text{true}$.
5. P_4 : $\text{Request}(0,0,2) \leq (7,2,4)$? Yes. $\text{Work} = (7,2,4) + \text{Allocation}[P_4](0,0,2) = (7,2,6)$. $\text{Finish}[P_4]=\text{true}$.

All five processes finish, via the sequence $\langle P0, P2, P3, P1, P4 \rangle$: **no deadlock exists** at time $T0$. Now suppose $P2$ makes one additional request: one more instance of resource type C , so $\text{Request}[P2]$ becomes $(0, 0, 1)$. Re-running the algorithm from $\text{Work} = \text{Available} = (0, 0, 0)$:

1. $P0$: $\text{Request}(0, 0, 0) \leq (0, 0, 0)$? Yes. $\text{Work} = (0, 0, 0) + (0, 1, 0) = (0, 1, 0)$. $\text{Finish}[P0] = \text{true}$.
2. $P1$: $\text{Request}(2, 0, 2) \leq (0, 1, 0)$? No.
3. $P2$: $\text{Request}(0, 0, 1) \leq (0, 1, 0)$? No, third component fails ($1 > 0$).
4. $P3$: $\text{Request}(1, 0, 0) \leq (0, 1, 0)$? No, first component fails ($1 > 0$).
5. $P4$: $\text{Request}(0, 0, 2) \leq (0, 1, 0)$? No.

No further process can be found with $\text{Finish}[i] == \text{false}$ and $\text{Request}[i] \leq \text{Work}$. The algorithm terminates with $\text{Finish}[P1] = \text{Finish}[P2] = \text{Finish}[P3] = \text{Finish}[P4] = \text{false}$. **The system is deadlocked**, and the deadlocked processes are exactly $P1, P2, P3$, and $P4$ (only $P0$'s resources could be reclaimed, and reclaiming them was not enough to satisfy anyone else's outstanding request).

3.9.3 When and How Often to Invoke Detection

Running the detection algorithm has a real computational cost, so a system must decide how often to invoke it. Two factors drive that decision: **how often a deadlock is likely to occur** in the first place (a system where deadlocks are extremely rare gains little from frequent checking), and **how many processes will need to be rolled back** once one is found, potentially one process for each disjoint cycle in the underlying wait relationship. Detection invoked too rarely lets a deadlock persist longer, wasting more resources and CPU time on processes that can never complete; detection invoked too often (for example, once per resource request) wastes CPU time on the detection algorithm itself. A further practical complication is that if detection is invoked at an arbitrary point after several deadlocks have independently accumulated, the resulting wait-for structure can contain multiple cycles simultaneously, making it hard to attribute the deadlock to any one specific "culprit" request.

3.9.4 Recovery from Deadlock

Once a deadlock is detected, the system (or, in some designs, a human operator informed of the situation) must break it. There are two broad families of recovery techniques: terminating one or more processes, or preempting resources from one or more processes.

Process termination. Two variants are possible. **Abort all deadlocked processes** is simple and guaranteed to break the deadlock completely, but at high cost: every one of the deadlocked processes loses all its work. **Abort one process at a time**, re-running the detection algorithm after each termination, until the deadlock cycle is eliminated. This is more expensive in overhead (detection must be re-run repeatedly) but avoids destroying more work than strictly necessary. Choosing *which* process to terminate first (the **victim**) typically weighs several factors: the process's priority, how long it has already run and how much longer it needs to finish, how many and which resources it holds, how many more resources it still needs to complete, how many total processes will need to be terminated, and whether the process is interactive or a batch job (killing an interactive process a user is actively waiting on is usually more disruptive than killing an unattended batch job).

Resource preemption. Rather than killing a process outright, the system can instead take resources away from one or more deadlocked processes and give them to others, repeating until the cycle is broken. Three distinct issues must be addressed for this to work correctly:

1. **Selecting a victim.** As with process termination, the order in which resources (and from which processes) are preempted should minimize overall cost, commonly weighing the number of resources a process holds and the amount of execution time it has already invested.

2. **Rollback.** A process that has a resource forcibly taken away cannot simply continue running as though nothing happened. It must be rolled back to some earlier, safe state and restarted from there. Since precisely characterizing "how far back is enough" is often difficult, the simplest and most common approach is a **total rollback**: abort the process entirely and restart it from scratch, even though a partial rollback (only as far back as necessary to release the preempted resource) would waste less work, at the cost of requiring the system to track much more detailed state about every running process.
3. **Starvation.** If victim selection is driven purely by a fixed cost formula, the same low-cost process can end up being selected as the victim over and over again, and never make forward progress, a form of starvation. The standard remedy is to fold the **number of times a process has already been rolled back** into the cost function used for victim selection, guaranteeing that a process can only be picked as a victim a small, bounded number of times before other factors make it a less attractive choice than some other process.

NOTE — A combined approach in practice

Real systems rarely commit to a single deadlock-handling strategy across every resource in the system. A common practical compromise partitions resources into classes (for example: internal kernel resources such as process control blocks; central memory; job-level assignable devices such as tape drives; and swappable backing-store space) and applies whichever strategy (prevention via resource ordering, prevention via preemption, avoidance, or simple pre-allocation) best fits the characteristics of each class, rather than forcing one uniform policy onto every kind of resource in the system.

CHECK YOUR PROGRESS 3.9

1. Under what condition (regarding resource instances) is a wait-for graph an adequate deadlock-detection tool, and what does a cycle in it signify?
2. In the multi-instance deadlock detection algorithm, how does the Request matrix differ in meaning from the Max matrix used in the Banker's Algorithm?
3. Using the worked detection example in this section, which processes remain deadlocked after P2's additional request for one more instance of C, and why can P0's resources not help resolve it?
4. Name the two broad families of deadlock recovery techniques, and one drawback of each.
5. Why is it important to include the number of prior rollbacks in the cost function used to select a preemption victim?

3.10 SUMMARY

Processes that share data need to coordinate, because letting them touch that data at the same time without coordination produces race conditions, timing-dependent errors where the final result depends purely on the luck of how the scheduler happened to interleave things. The part of a process's code that touches shared data is its critical section, and every synchronisation tool in this chapter is really a different way of solving the same problem: guaranteeing mutual exclusion, progress, and bounded waiting all at once.

Simple software tricks with a shared turn variable or unchecked flags fail one of those three requirements, or deadlock outright. Peterson's solution handles two processes correctly, and the Bakery algorithm extends the same idea to any number of processes using ticket numbers, but neither holds up on modern multiprocessor hardware that can reorder memory accesses. Hardware support, disabling interrupts on a single processor, or an atomic instruction like Test-

and-Set on any system, gives the indivisible read-modify-write step that plain software can't guarantee on its own, and everything built on top relies on it.

A semaphore is an integer touched only through atomic wait and signal operations. Binary semaphores give simple mutual exclusion, counting semaphores manage a pool of several identical resources, and a blocking implementation, where a waiting process is parked on the semaphore's own queue instead of spinning, is what real kernels actually use.

Semaphores leave it entirely up to the programmer to pair every wait with its signal correctly. Critical regions, and their extension, conditional critical regions, move that pairing into the compiler, but turned out expensive to implement well. Monitors do it better: they bundle shared data with the only procedures allowed to touch it, enforce mutual exclusion automatically at the boundary, and add condition variables for finer control, exactly as the bounded-buffer example showed. Message passing offers yet another option, letting processes coordinate by sending and receiving messages instead of sharing memory at all.

We tested these tools against four classic problems. Bounded buffer needs two counting semaphores and a mutex. Readers-writers can starve either side depending on which one you favour. Dining philosophers can deadlock outright if everyone reaches for their left fork at once, fixed by breaking the symmetry. Sleeping barber uses one counting semaphore as both a wake-up signal and a customer count.

A deadlock happens when a group of processes are each waiting on a resource that only another process in that same group holds, and it needs all four conditions, mutual exclusion, hold-and-wait, no preemption, and circular wait, to be true at once. The resource-allocation graph makes this visible: no cycle means no deadlock, and a cycle among single-instance resources means deadlock is guaranteed.

Prevention rules out one of the four conditions permanently, usually by ordering resource types so circular wait can't happen. Avoidance takes a different route: it checks, before granting anything, whether the system would still be in a safe state afterward, one where every process could still eventually finish. The Banker's Algorithm does this check formally, and the worked example showed something students often find surprising: a request can be refused even when the resources it's asking for are sitting right there available, because granting it would leave no safe path forward.

When a system uses neither prevention nor avoidance, it needs a detection algorithm to notice a deadlock after the fact, and a recovery strategy, killing processes or taking back resources, to break it.

This chapter's tools form one continuum, from raw hardware instructions up through semaphores, monitors, and messages, all built to stop the same problem: processes stepping on each other's shared data. Deadlock is the related but separate failure that can still happen even after every race condition has been eliminated, when processes end up waiting on each other forever instead.

3.11 GLOSSARY

Term	Meaning
Race condition	Outcome of concurrent access to shared data that depends on the unpredictable order of interleaving
Critical section	The part of a process's code that accesses shared data and must not overlap with another process's access to it
Mutual exclusion	The requirement that no two processes execute in their critical sections at the same time
Progress	The requirement that a decision on who enters the critical section next cannot be postponed indefinitely
Bounded waiting	The requirement that there is a limit on how many times other processes may enter before a waiting process's turn comes

Term	Meaning
Peterson's solution	A two-process software algorithm for mutual exclusion using a shared turn and flag array
Bakery algorithm	An n-process software mutual-exclusion algorithm based on ticket numbers, with lexicographic tie-breaking
Test-and-Set	An atomic hardware instruction that reads and sets a memory word in one indivisible step
Swap (Exchange)	An atomic hardware instruction that exchanges the contents of two memory words in one indivisible step
Semaphore	An integer variable manipulated only through the atomic wait/signal operations
Binary semaphore	A semaphore restricted to values 0 and 1, used for simple mutual exclusion
Counting semaphore	A semaphore that can take any non-negative integer value, used to manage a pool of resource instances
Busy waiting (spinlock)	A waiting process repeatedly re-testing a condition in a loop, wasting CPU cycles
Blocking (queuing) semaphore	A semaphore implementation that puts a waiting process to sleep on the semaphore's own queue instead of busy-waiting
Critical region	A language construct (region $\forall$ do S) that automates entry/exit synchronization around a shared variable
Conditional critical region	A critical region extended with a boolean guard (region $\forall$ when B do S) that must hold before entry
Monitor	A high-level construct encapsulating shared data and procedures, with implicit mutual exclusion
Condition variable	A monitor-internal variable supporting wait/signal, used for fine-grained waiting inside a monitor
Signal-and-wait / signal-and-continue	The two ways of resolving who runs next when a monitor's signal() wakes a waiting process
Message passing	Communication and synchronization between processes via send/receive rather than shared memory
Direct communication	Message passing where sender and receiver name each other explicitly
Indirect communication (mailbox)	Message passing where sender and receiver communicate through a shared mailbox/port
Synchronous (blocking) communication	Send or receive that does not return until the message is transferred
Asynchronous (non-blocking) communication	Send or receive that returns immediately, without waiting
Bounded-buffer (producer-consumer) problem	The classical problem of coordinating a producer and consumer around a fixed-size buffer
Readers-writers problem	The classical problem of allowing concurrent readers but exclusive writers on shared data
Dining philosophers problem	The classical problem modelling deadlock risk in acquiring a shared pool of exclusive resources
Sleeping barber problem	The classical problem modelling a single server and a bounded waiting room, without lost wake-ups
Deadlock	A state where every process in a set is waiting for a resource held only by another process in the same set
Mutual exclusion (deadlock condition)	At least one resource must be held in a non-sharable mode
Hold and wait	A process holds at least one resource while waiting for additional resources held by others
No preemption	A resource can only be released voluntarily by the process holding it
Circular wait	A cycle of processes, each waiting for a resource held by the next
Resource-allocation graph (RAG)	A directed graph of process and resource-type vertices, with request and assignment edges
Claim edge	A dashed RAG edge indicating a process may request a resource in the future (used in avoidance)

Term	Meaning
Safe state / safe sequence	A state from which a process ordering exists guaranteeing every process can eventually finish
Deadlock prevention	Structurally denying one of the four necessary conditions so deadlock cannot occur
Deadlock avoidance	Dynamically granting requests only when the resulting state is provably safe
Banker's Algorithm	A deadlock-avoidance algorithm for multiple resource instances, using Available/Max/Allocation/Need
Safety algorithm	The Banker's Algorithm component that tests whether a given state is safe
Resource-request algorithm	The Banker's Algorithm component that decides whether to grant an actual resource request
Deadlock detection	An algorithm that determines, after the fact, whether the system is currently deadlocked
Wait-for graph	A simplified graph (processes only) used for deadlock detection with single-instance resources
Deadlock recovery	Breaking an actual deadlock, by process termination or resource preemption
Rollback	Restoring a preempted process to an earlier safe state, up to and including a full restart

CHAPTER 4

File Systems and Input/Output Management

Every earlier unit in this course dealt with abstractions the operating system builds inside the computer: processes, threads, address spaces, scheduling queues. This unit deals with the point where those abstractions meet the physical world: the disks, controllers, terminals, and clocks that actually store and move information, and which are, without exception, slower, more heterogeneous, and more failure-prone than the CPU that must work with them. A **file system** is the operating system's answer to persistent storage: it turns a raw sequence of addressable disk blocks into named, organised, protected objects that survive after a program exits and the machine is switched off. **Input/output (I/O) management** is the operating system's answer to device diversity: it turns a bewildering variety of hardware (spinning disks, keyboards, printers, network cards, timers) into a small number of clean, uniform software interfaces that application programs can use without caring which specific device sits behind them. This unit studies both halves of that story: how files and directories are organised and implemented on disk (Sections 4.1 and 4.2), how the operating system talks to I/O hardware in principle and structures its I/O software in layers (Sections 4.3 and 4.4), and how the two most important physical devices for this unit, disks and clocks, actually work and are scheduled (Sections 4.5 and 4.6).

4.1 FILE SYSTEMS AND DIRECTORIES

4.1.1 The File Concept

Computers store information on a variety of physical media: magnetic disks, solid-state drives, optical disks, magnetic tape. Each of these media has its own physical organisation: sectors, tracks, blocks addressed by physical location. If every application program had to deal with these physical details directly, writing software would be tedious and fragile, and no two devices would be programmed alike. The operating system solves this by defining a single **logical** unit of storage, the **file**, and presenting every device to application programs through this one uniform abstraction, regardless of what physical medium actually holds the data.

IMPORTANT — Definition — File

A **file** is a named collection of related information, recorded on secondary storage, that the operating system treats as the smallest logical unit of persistent storage a program can create, read, write, or delete. A file may hold a program's source code, its compiled executable image, numeric or textual data, an image, or any other sequence of bytes an application chooses to interpret.

A file is generally organised internally as a sequence of **records**, and a record as a sequence of **fields**. A **field** is a single elementary item of data, such as a student's name or a single subject mark. A **record** is a collection of related fields treated as one unit by an application, for instance one student's name together with the marks obtained in each subject and a pass/fail indicator. A **file** is then a collection of similar records, for instance the marks file for an entire class, one record per student. This record/field view is how application programs (and, historically, file-processing languages such as COBOL) think about file content; the file system itself, at its lowest level, only guarantees that a file is a named, ordered sequence of bytes or blocks. How that sequence is subdivided into records and fields is left to the application, except where the operating system additionally supports a specific record-structured file organisation.

4.1.2 File Attributes

Every file the operating system manages carries a fixed set of properties, independent of its content, that the file system stores and uses to manage the file. These are the file's **attributes**.

Table 4.1 — File attributes maintained by the operating system

Attribute	Description
Name	The human-readable string by which the file is identified and referred to by users; unique only within its own directory
Identifier	A unique internal tag (usually a number) that identifies the file within the file system, used internally instead of the name for speed
Type	The kind of file this is, commonly signalled by an extension on its name (for example .exe, .obj, .txt)
Location	A pointer to the device the file resides on and to the location of the file's data on that device
Size	The file's current size, in bytes, words, or blocks, and possibly the maximum size permitted
Protection	Access-control information determining who may read, write, execute, or otherwise access the file
Time, date, and user identification	Information kept for creation time, last modification, and last use, typically together with the identity of the user who performed each action, useful for security monitoring and for deciding whether a cached copy of the file is still current

All of this attribute information is kept in the file's **directory entry** or, on many systems, in a dedicated on-disk structure such as a UNIX **inode**; Section 4.2 discusses how this is implemented.

4.1.3 File Operations

The operating system provides a small set of system calls through which every higher-level file operation (opening a document, appending a log line, deleting a temporary file) is ultimately built.

1. **Create.** Creating a file is a two-step process: the file system first checks that sufficient space is available, and then makes a new entry for the file in the appropriate directory, recording its name and location.
2. **Write.** To write to a file, the caller supplies the file's name (or an open handle to it) together with the data to be written. The system locates the file via the directory and maintains a **write pointer** that records the location within the file at which the next write is to occur; each write advances this pointer.
3. **Read.** To read from a file, the system similarly locates it via the directory and maintains a **read pointer** to the location within the file from which the next read is to occur; each read advances this pointer. Many systems combine the read and write pointers into a single **current-file-position pointer** shared by both operations.
4. **Reposition within a file (seek).** The directory is searched for the file's entry, and the current-file-position pointer is repositioned to a specified value without any data actually being transferred. This operation is also called a **file seek**.
5. **Delete.** The directory is searched for the named file; once found, the file's space is released back to the free-space pool and its directory entry is removed.
6. **Truncate.** The contents of the file are erased while its attributes (name, identifier, protection information) are left unchanged. This is useful when a program wants to reuse a file's identity while starting its content over.

Other operations (**append**, **rename**, **copy**) are frequently provided as well, but each can be expressed as a combination of the six primitive operations above.

4.1.4 File Types and File Structure

A file's name is conventionally split into two parts: the name proper, and an **extension**, separated by a period. The extension is a convention, not something the file system is forced to enforce, but most operating systems and applications use it to infer the file's **type**: the kind of content the file holds and, often, which application should be used to open it. Table 4.2 gives common examples.

Extension	File type
.exe, .com, .bin	Executable: ready-to-run machine code
.o, .obj	Object: compiled machine code not yet linked
.c, .java, .py	Source: program text in a high-level language
.txt, .doc	Text/document: human-readable or word-processed text
.zip, .tar	Archive: a container bundling multiple files together

A file's type can also imply an **internal structure**. The operating system requires an executable file to have a specific internal structure so that it knows where in memory to load the file and what its first instruction's address is. Some operating systems go further and enforce structure on all files (for example, a fixed set of record types with defined lengths), but doing so increases the size and complexity of the file system, since it must contain the code to interpret and enforce every structure it supports. At the opposite extreme, **UNIX and UNIX-like systems (including Linux) treat every file as simply an unstructured stream of bytes**, individually addressable by its offset from the beginning of the file; any structure a program wants to impose (records, fields, headers) is a convention enforced entirely by the application, not the operating system. This minimal approach keeps the file system itself small and general while still allowing any structure an application needs to be layered on top.

Whatever the logical structure, the file system must eventually **map** the file's bytes onto physical disk blocks of some fixed size (say, 512 bytes or 4 KB). If a file's logical size is not an exact multiple of the block size, the last block allocated to it is only partially used; the unused remainder of that final block is wasted space known as **internal fragmentation**. For example, with a 512-byte block size, a file of 1,949 bytes requires 4 blocks (2,048 bytes total), wasting the last 99 bytes. Every block-based file system suffers from some internal fragmentation; the larger the block size chosen (for efficiency of large-file transfer), the greater the average internal fragmentation across many small files.

4.1.5 File Access Methods

Once information is stored in a file, it must be read back into memory in some order. The operating system supports several distinct **access methods**, and the choice of method affects both how simply a program can be written and how efficiently the underlying device can be used.

Sequential access. Information in the file is processed strictly in order, one record after another, exactly as it was written. A read operation automatically advances a position pointer past the record just read; a special rewind operation resets the pointer to the beginning. Sequential access is the natural (and, on magnetic tape, the only practical) access method, because tape itself can only be read or written in physical sequence. If a file of 100 records has its read/write position currently at record 45 and the 75th record is wanted, every intervening record from 46 through 74 must be passed over before 75 is reached.

Direct (random) access. Also called **relative access**, this method allows records to be read or written in any order, without regard to their position relative to one another, because the underlying storage (a disk) permits any block to be addressed directly by its block number. A file organised for direct access is viewed as a numbered sequence of fixed-size logical records or blocks, and a program can request block 14 and then block 192 with no penalty for the "jump" beyond the seek itself. For example, a disk of 256 blocks with the read/write head currently positioned at block

95 can move to service block 250 directly, without having to pass through blocks 96–249 first; similarly, a music CD holding 10 tracks lets a listener jump from track 3 straight to track 10.

Indexed sequential access. This method addresses the main weakness of pure sequential access, that locating a specific record deep in a large file requires scanning past every record before it, by building an **index** on top of a sequential file. Records are still stored in sequence, ordered by a chosen **key field**, but a separate index file maps each key (or each range of keys) to the physical location of the corresponding record, so a lookup can jump almost directly to the record wanted rather than scanning from the start. Large indexes are frequently organised in two levels: a small **master index** that identifies which block of a larger **secondary index** to search, and the secondary index that in turn points to the actual record's location. For example, a file of 60,000 records might have a master index dividing the file into 6 blocks, each pointed to by one master-index entry; each of those blocks might hold a secondary index of 10,000 entries, each entry consisting of a key field and a pointer field giving the exact location of the corresponding record. This method gives most of the flexibility of direct access while still allowing efficient sequential processing of the whole file when that is what is wanted.

Table 4.2 — File access methods compared

Access method	How records are reached	Best suited to	Typical medium
Sequential	Strictly in written order, one after another	Batch processing, streaming reads, tape backup	Magnetic tape, log files
Direct (random)	Any block addressed directly by block number	Databases, random lookups, paging files	Magnetic/solid-state disk
Indexed sequential	An index narrows the search, then a short sequential scan finishes it	Large keyed files needing both fast lookup and ordered scans	Disk-based database files

4.1.6 Directory Structures

A file system holding millions of files cannot leave them all in one undifferentiated pool; something must organise them so that files can be found, and so that many users or many applications can each manage their own files without stepping on one another's names. This organising structure is the **directory**: a special file (or set of on-disk structures) that maps file names to the information (location, attributes) needed to find and use each file. The set of operations a directory must support includes searching for a file, creating a new entry, deleting an entry when a file is removed, listing all entries, renaming an entry, and traversing the whole structure (for example, to back up every file in the system).

Directory structures have evolved through several designs, each solving a specific limitation of the one before it.

Single-level directory. The simplest possible scheme: one directory holds the entries for every file in the system, with no further subdivision, sometimes called the **root directory**. Its advantages are simplicity and speed of lookup. Its fatal weakness appears the moment more than one user (or more than one unrelated application) shares the system: because names must be unique within the one directory, if user 1 creates a file called `sample` and user 2 later creates a file of the same name, user 2's file silently overwrites user 1's. This makes a single-level directory unworkable on any multi-user system.

Two-level directory. To remove the name-collision problem, each user is given a private directory of their own, called a **user file directory**, and a master directory maps each user name to their private directory. Names chosen by one user no longer interfere with names chosen by another, since uniqueness is now required only within each user's own directory rather than system-wide. The remaining weakness: a single user with many files of their own still has only one flat directory to organise them in, which does not scale well once that user accumulates hundreds or thousands of files across different projects.

Tree-structured directory. A two-level directory eliminates name conflicts between users but is unsatisfactory for a user with a large number of files, since it still offers no way to group related files together within one user's space. The tree-structured directory solves this by allowing any directory to contain both files and further **sub-directories**, so a user (or the system as a whole) can create as many levels of directory as convenient, grouping files by project, type, or purpose. Every file is located by a **path**: the sequence of directory names leading from some starting point down to the file.

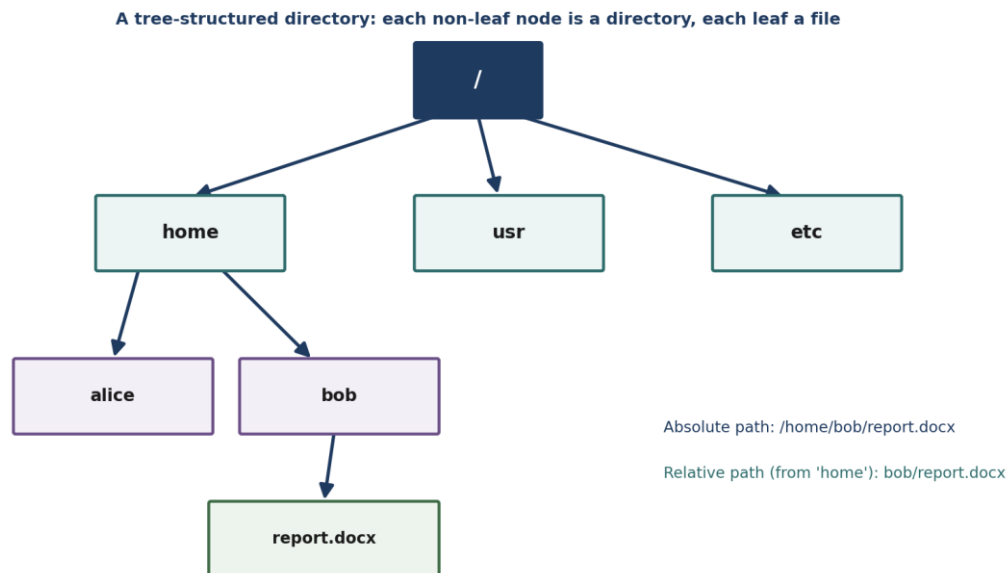


Figure 4.1 — Absolute and relative paths in a directory tree

There are two ways to express a path through a tree-structured directory:

- An **absolute path** begins at the tree's root directory and names every directory along the way down to the file, giving a full, unambiguous description of the file's location regardless of where the user currently is in the tree (for example, /home/student/reports/unit4.docx).
- A **relative path** begins at the user's **current working directory** rather than at the root, and is generally shorter to type, but its meaning depends on which directory is current at the time it is used (for example, reports/unit4.docx, used while the current directory is /home/student).

Acyclic-graph directory. A pure tree forces every file to have exactly one directory entry (one parent), which becomes a problem the moment two or more users need to genuinely **share** a file or sub-directory, for instance several students collaborating on one project's files. An acyclic-graph directory permits a single sub-directory or file to be referenced from more than one place in the directory structure, so a shared sub-directory can be declared common to multiple users' directories; changes any one of them makes are then visible to all the others, because there is really only one underlying file being pointed to from multiple locations. The graph is required to remain **acyclic** (containing no cycles), so that the structure still behaves sensibly for operations like traversal, but a file may now legitimately have more than one absolute path leading to it. This creates a new bookkeeping problem at deletion time: if one user deletes a shared file through one of its directory entries, the file's data must not actually be freed while another directory entry (another path) still points to it. Every pointer to the file must be tracked, and the underlying storage released only once every reference to it has been removed, typically via a **reference count** on the file (also called **link count**) that is decremented each time an entry pointing to the file is deleted, with the physical data reclaimed only when the count reaches zero.

General graph directory. If arbitrary links are permitted between directories with no restriction, cycles can be introduced into the structure (a sub-directory can end up, directly or indirectly,

containing a link back to one of its own ancestors), turning the acyclic graph into a fully general graph. This is more flexible for sharing but creates two serious correctness problems that a general-graph directory implementation must explicitly solve:

- **Traversal.** An algorithm that walks the whole directory tree (to search every file, or to compute total disk usage, or to back up the file system) can loop forever if it does not detect that it has returned to a directory it has already visited; the traversal algorithm must therefore actively perform **cycle detection**, for example by marking each directory visited and refusing to descend into an already-marked directory a second time.
- **Deletion and garbage collection.** With arbitrary cycles, a simple reference count is not sufficient to decide when a file's storage can safely be reclaimed: a group of directories can reference only one another in a cycle, with a combined reference count that never drops to zero even though nothing outside the cycle can reach any of them any more, so the whole group is effectively unreachable garbage that pure reference counting will never collect. Correctly reclaiming such storage requires a full **garbage-collection** pass: traversing the entire directory structure from a known root, marking every file and directory that is actually reachable, and then treating everything left unmarked as free, regardless of what its own internal reference count says.

Because a general graph creates these two problems for comparatively little practical benefit over an acyclic graph, most production file systems (including UNIX/Linux and Windows) permit controlled sharing similar to an acyclic graph. For example, UNIX **hard links** are kept strictly acyclic by the operating system for regular files, while **symbolic links**, which can point anywhere including into a cycle, are treated as ordinary data that is resolved (and can fail with a "too many levels of symbolic links" error) rather than as structural links that a traversal must silently follow.

NOTE — Hard links versus symbolic links (UNIX/Linux)

A **hard link** is a second directory entry pointing at the same underlying inode as an existing file: the file's reference count is incremented, and the two names are indistinguishable; the file's data is freed only when the last hard link to it is removed. A **symbolic link** (symlink) is a small, separate file that merely stores the *path* to another file; it does not increase the target's reference count, can point across file systems (hard links cannot), and becomes a "dangling" link if its target is later deleted.

CHECK YOUR PROGRESS 4.1

1. Distinguish a file's **name** from its **identifier**, and explain why the file system needs both.
2. Which file operation removes only a file's contents while leaving its attributes unchanged, and how does it differ from delete?
3. Explain, with an example, why indexed sequential access is faster than pure sequential access for locating one specific record in a large file.
4. What specific problem does a two-level directory solve that a single-level directory cannot, and what problem does it fail to solve?
5. Why does a general graph directory structure require active cycle detection during traversal, while a tree-structured directory does not?

4.2 FILE SYSTEM IMPLEMENTATION AND PROTECTION

Section 4.1 described files and directories from the point of view of a program using them: names, attributes, operations, access methods. This section looks underneath that interface, at how the file

system actually places a file's data onto disk blocks, tracks which blocks are free, implements the directory itself, and protects files against both careless and malicious misuse.

4.2.1 Allocation Methods

A disk is divided into fixed-size **blocks** (typically 512 bytes to a few kilobytes), and the central implementation question of any file system is: **which blocks belong to which file, and in what order?** The mapping from a file's logical blocks (numbered 0, 1, 2, ... n-1 from the file's own point of view) to physical blocks on the disk is called the file's **allocation**, and three classical strategies are used to record it.

Contiguous allocation. Each file occupies a set of physically contiguous blocks on the disk, that is, consecutive block addresses, so that the file's directory entry needs only record the starting block address and the length (in blocks). Locating any logical block i of the file is then pure arithmetic: physical block = start + i , with no indirection at all.

- *Advantages:* excellent performance for both sequential access (blocks are read in physical order, minimising seek time) and direct access (any block's physical address is computed instantly); simple to implement.
- *Disadvantages:* suffers from **external fragmentation**: as files are created and deleted, free space becomes broken into scattered pieces too small individually to satisfy a new file's request, even though the *total* free space may be more than enough; a file also cannot grow easily beyond the space originally allocated to it unless a fortuitously-placed free block happens to lie immediately after it, which usually forces the file to be copied elsewhere if it outgrows its slot; and the file system must decide the file's full size in advance, or grow it by relocating it, which is costly.

Linked allocation. Each file is a linked list of disk blocks scattered anywhere on the disk; the directory entry stores only a pointer to the file's first block, and every block, besides its portion of the file's data, stores a pointer to the next block in the file (the last block's pointer is a null/end marker).

- *Advantages:* no external fragmentation, since any free block anywhere can be used; a file can grow simply by linking in one more free block wherever one is available, with no need to relocate anything.
- *Disadvantages:* purely sequential access is efficient, but **direct access is very inefficient**, since reaching logical block i requires following i pointers from the start of the file, one disk access at a time; reliability is fragile, since a single corrupted pointer breaks the chain and strands every block after it; and each block wastes some of its space storing the next-block pointer rather than data. A common refinement, **file-allocation-table (FAT)** style linking, moves every block's "next" pointer out of the data blocks themselves and into one dedicated table held in memory, so following the chain becomes a fast in-memory table lookup rather than a disk access per block, and the full data block is available for actual content.

Indexed allocation. Each file is given its own **index block**, a block that itself contains nothing but an array of pointers to the file's actual data blocks, so the directory entry need only record the address of this one index block, and logical block i is located by looking up the i -th pointer in the index block: a single extra level of indirection rather than a chain to be followed one block at a time.

- *Advantages:* supports genuinely efficient direct access, since any block's address is one indirection away, not a chain traversal; no external fragmentation, since blocks can be scattered freely; a file can grow simply by writing to a new block and recording its address in the index block, up to the index block's own capacity.
- *Disadvantages:* some space is spent on the index block itself, which is pure overhead for a very small file (a one-block file still needs a whole index block); and a single index block has a fixed capacity, limiting the maximum file size unless a further mechanism is added. Real file systems address this last problem with **multilevel index blocks** (an index block whose

entries point to further index blocks rather than directly to data, extending reach at the cost of an extra indirection per level) and/or a scheme mixing a handful of direct pointers for a file's first few blocks with one or more indirect pointers for larger files: the design used, for instance, by the classical UNIX inode.

Table 4.3 — Allocation methods compared

Method	Direct access	File growth	Fragmentation	Overhead
Contiguous	Fast (direct arithmetic)	Difficult once initial space is used	External fragmentation	Minimal (start + length only)
Linked	Slow (must follow the chain)	Easy: link in any free block	None (external); some internal	One pointer per block
Indexed	Fast (one indirection via index block)	Easy up to index capacity	None (external); index block itself is overhead	One index block per file (more for very large files)

CONTIGUOUS



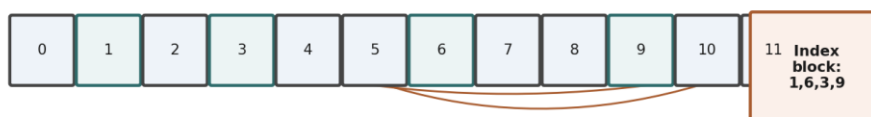
File F: blocks 2-4 (start = 2, length = 3) -- one run of consecutive blocks

LINKED



File F: block 1 -> 6 -> 3 -> 9 -- each block stores a pointer to the next

INDEXED



File F: one index block lists all its blocks directly

-- no chaining needed to find any block

Figure 4.2 — Contiguous, linked, and indexed file allocation

EXAMPLE — Choosing an allocation method

A video file that will be read start-to-finish for playback benefits most from **contiguous allocation**: sequential reads dominate, and the whole file's size is usually known when it is created. A log file that grows unpredictably in small increments over a long time suits **linked allocation** well, since it never needs to be relocated as it grows. A database table file that both grows over time and is accessed at arbitrary offsets (indexed lookups, random updates) is best served by **indexed allocation**, since it combines easy growth with fast direct access to any block.

4.2.2 Free-Space Management

Whichever allocation method is used, the file system must also track which blocks are currently **free**, so it knows where a new file (or a growing one) may be given space. Three common techniques are used.

- **Bit vector (bitmap).** One bit is kept for every block on the disk: 1 if the block is free, 0 if it is allocated (or vice versa, by convention). Finding a free block is then a matter of scanning the bitmap for a 1 bit, which can be done efficiently a whole computer word at a time; the bitmap itself is compact (one bit per block) and simple to reason about, and is the technique used by most modern file systems.
- **Linked list of free blocks.** Every free block holds a pointer to the next free block, forming a free list threaded through the free space itself, headed by a pointer kept in a fixed location. This needs no separate bitmap storage, but finding a large run of contiguous free blocks (useful for contiguous allocation) requires walking the list, and the list itself occupies exactly the free blocks it describes, so it costs nothing extra in steady state but is slower to search than a bitmap.
- **Grouping.** A variant of the linked-list idea that improves its performance: the first free block stores the addresses of the next n free blocks (rather than a pointer to just one), and the last of those n addresses actually points to another block that itself holds a further n addresses, and so on. This lets a large number of free-block addresses be found quickly with far fewer disk accesses than following a plain one-at-a-time linked list.

Table 4.4 — Free-space management techniques compared

Technique	How it works	Main strength	Main weakness
Bit vector (bitmap)	One bit per block, marking it free or allocated	Compact; fast word-at-a-time scanning	Whole bitmap may need scanning to find a large free run
Linked list	Each free block points to the next free block	No separate storage needed beyond the list pointer	Slow to search; finding contiguous runs requires walking the list
Grouping	The first free block holds addresses of the next n free blocks	Finds many free blocks per disk access	Slightly more complex bookkeeping than a plain list

4.2.3 Directory Implementation

The directory itself must be stored somewhere and searched efficiently, and two implementation approaches are common.

- **Linear list.** The directory is a simple list of entries, each holding a file name together with a pointer to its attribute/allocation information (or holding the attributes directly). It is straightforward to implement, but a linear search is required to find, create, or delete any entry, which becomes slow as the number of files in a directory grows large: an operation whose time is proportional to the number of files present.
- **Hash table.** A hash table is built over the linear list, keyed by file name: computing a hash of the name gives (almost always) a direct index into the table, giving near-constant-time lookup, insertion, and deletion regardless of how many files the directory holds. The main complication is handling **collisions** (two different names that hash to the same slot), usually resolved by chaining. Most production file systems use a hash-table (or similar tree-based) structure for any directory expected to hold more than a small number of files.

4.2.4 Protection: Access Control

Once a file system can create, locate, and store files, it must also decide **who is allowed to do what to each file**: the discipline of **protection**. Two related but distinct mechanisms are used across real operating systems.

An **access-control list (ACL)** attaches to each file (or directory) an explicit list of (user or group, permission set) pairs, stating exactly which permissions each individual user or group holds for that specific object. Conceptually, this is one row of a system-wide **access matrix**, whose rows are subjects (users or processes) and whose columns are objects (files, devices, other resources), with each cell holding the set of operations that subject may perform on that object; an ACL is simply the access matrix stored **column-by-column**: one list per object, naming every subject with any right to it. ACLs are expressive (arbitrary users can be granted arbitrary combinations of rights on any one file) but can become long and expensive to store and check when many distinct users need distinct rights on many distinct files.

The alternative, simpler model, used by UNIX/Linux and inherited by most modern systems as their baseline scheme, reduces the subject side to exactly three categories per file: the file's **owner**, the owner's **group**, and **others** (everyone else), and reduces the permission side to a small fixed set (traditionally **read**, **write**, and **execute**). Each file stores nine permission bits: read/write/execute for owner, group, and others respectively. This owner/group/other model is dramatically cheaper to store (a handful of bits per file, versus a list that can grow arbitrarily long) and fast to check (three category comparisons, not a list search), at the cost of expressiveness: it cannot directly grant one specific additional user a right without putting them in a group, which is why many modern UNIX-like systems layer optional ACLs on top of the base permission bits for the cases where finer-grained control is genuinely needed. Table 4.5 summarises the trade-off.

Table 4.5 — Access-control lists versus permission bits

Property	Access-control list (ACL)	Owner/group/other permission bits
Granularity	Per-user, arbitrary combinations, per object	Three fixed categories per object
Storage cost	Grows with number of distinct grants	Fixed, small (a handful of bits per file)
Lookup cost	Must search the list for the requesting subject	Constant-time category check
Typical use	Fine-grained sharing (e.g., a shared project file for a named set of collaborators)	The default, general-purpose model on most files

4.2.5 Authentication and One-Time Passwords

Access control decides what a *known, verified* user may do; **authentication** is the separate, prior problem of establishing that a user really is who they claim to be, before any access-control decision is even applied. Operating systems commonly authenticate users by one (or a combination) of three means: something the user **knows** (a username and password), something the user **has** (a card, a hardware security key, or a code generated by a key-generator device), or something the user **is** (a biometric attribute such as a fingerprint or retina pattern).

A plain reusable password has an inherent weakness: if it is ever intercepted (over a network, by a keystroke logger, or by an observer) it can be reused by the attacker at any later time, since the system has no way of telling a legitimate login from a replayed one. **One-time passwords (OTPs)** are designed specifically to defeat this **replay attack**: a fresh, unique password is required for every single login attempt, and once used, that particular value is permanently invalidated by the system, so an attacker who intercepts one OTP gains nothing. By the time they could reuse it, it is already spent. One-time passwords are implemented in several ways: a physical card printed with a grid of numbers, where the system challenges the user for the value corresponding to a randomly chosen coordinate each time; a hardware token or authenticator app that generates a new numeric code (often time-based) synchronised with the server; or a code sent over a separate channel, typically SMS or e-mail, to a registered device, which the user must enter within a short validity window.

IMPORTANT — Definition — One-Time Password (OTP)

A **one-time password** is an authentication credential valid for exactly one login session or transaction, generated fresh each time it is needed and rendered permanently unusable

immediately after that single use, which defeats replay attacks that would succeed against a static, reusable password.

4.2.6 Threat Model: Program Threats and System/Network Threats

Protection and authentication exist because an operating system must assume it will be targeted, deliberately, by threats of two broad kinds.

Program threats arise when a user program is made, deliberately or through a hidden flaw, to perform actions its user never intended or authorised.

- A **Trojan horse** is a program that appears to perform a legitimate, wanted function while secretly also performing a malicious one, classically capturing a user's login credentials and transmitting them to an attacker.
- A **trap door** (or backdoor) is a hidden entry point deliberately left in an otherwise correctly functioning program, allowing whoever knows of it to bypass normal access checks.
- A **logic bomb** is code that behaves normally until a specific triggering condition is met (a date, an event, an absence of a particular employee's login), at which point it performs its malicious action; its dormancy makes it especially hard to detect in advance.
- A **virus** is a fragment of code that embeds itself in other programs or files and replicates as those files are executed or shared, potentially corrupting or deleting data and rendering a system unusable.

System (and network) threats misuse system services and network connections rather than tricking a single program into misbehaving.

- A **worm** is a self-replicating process that consumes system resources (CPU, memory, network bandwidth) so aggressively, by continually spawning further copies of itself, that it starves every other process of the resources it needs, potentially enough to degrade or shut down an entire network. Unlike a virus, a worm does not need to attach itself to a host program to spread.
- **Port scanning** is a reconnaissance technique in which an attacker systematically probes a target system's network ports to discover which services are running and listening, in order to identify a specific vulnerability to exploit next; it is usually a precursor to an attack rather than an attack in itself.
- A **denial-of-service (DoS)** attack aims not to steal or alter data but to prevent legitimate use of a system or service altogether, typically by overwhelming it with a volume of requests or traffic it cannot cope with.

This threat model, a program threat exploiting one process's trust and a system/network threat exploiting shared infrastructure, is precisely why an operating system's protection responsibilities span both the fine-grained access control of Section 4.2.4 (limiting the damage any one program can do, even if compromised) and system-wide vigilance (resource limits, network filtering, and authentication) against threats that never pass through any single program's own permission checks at all.

CHECK YOUR PROGRESS 4.2

1. Explain why direct access to a file's block *i* is fast under indexed allocation but slow under linked allocation.
2. Which allocation method suffers from external fragmentation, and why do the other two not?
3. What is the essential difference between an access-control list and the owner/group/other permission-bits model, and what does each trade off?

4. Explain, in your own words, why a one-time password resists a replay attack while a reusable password does not.
5. Give one example each of a program threat and a system threat, and explain the key difference between the two categories.

4.3 PRINCIPLES OF I/O HARDWARE

4.3.1 I/O Devices: Block and Character Devices

The devices attached to a computer are extraordinarily varied (disks, keyboards, printers, network cards, sensors), yet the operating system manages to present them through a small number of uniform models. The most fundamental distinction it draws is between **block devices** and **character devices**.

A **block device** stores and transfers information in fixed-size, independently addressable blocks, each identified by its own address; any block can be read or written independently of the others, and, critically, a block device generally supports **direct (random) access**: it is always possible to seek to any block and read or write it, without reference to which block was accessed immediately before. A magnetic or solid-state disk is the archetypal block device: it is always possible to seek to another cylinder and wait for the required block to rotate under the read/write head, regardless of where the arm currently happens to be.

A **character device** delivers or accepts a stream of characters (bytes) with no block structure at all, and it is not independently addressable: there is no notion of "seeking" to a particular position within the stream, since the device has no such positions to seek to. Keyboards, mice, printers, and network interfaces are the common examples of character devices.

Table 4.6 — Block devices versus character devices

Property	Block device	Character device
Unit of transfer	Fixed-size, addressable block	Unstructured stream of characters/bytes
Random access	Yes: any block directly addressable	No: no seek operation exists
Examples	Hard disk, SSD, CD/DVD	Keyboard, mouse, printer, network interface

4.3.2 Device Controllers and Communication with the CPU

An I/O unit typically consists of two parts: a mechanical component (the device itself: the spinning platter, the print head, the keyboard switches) and an electronic component that drives it, called the **device controller**. The controller is the interface between the physical device and the rest of the computer: any device is connected via a plug and socket to its controller, and the controller in turn is what the CPU and memory actually communicate with over the system bus. A single controller can often manage several similar devices (for example, one disk controller managing several drives), and its job includes converting between the device's own serial bit stream and the block-oriented, byte-addressable world the CPU expects, as well as performing basic error detection and correction on the data it moves.

The CPU communicates with a device controller through the controller's **device registers**: a small set of addressable locations, often called **I/O ports**, used to issue commands, supply data, and read back status. There are two ways these registers can be exposed to software, corresponding to two of the three communication approaches described next: they can occupy their own, separate address space (accessed only by special I/O instructions), or they can be mapped into the same address space as ordinary memory (accessed by ordinary load/store instructions).

4.3.3 Programmed I/O, Memory-Mapped I/O, and Direct Memory Access

Three general approaches exist for the CPU to exchange data with a device controller, differing chiefly in how much of the transfer the CPU itself must actively perform.

Special (programmed) instruction I/O. The CPU uses instructions specifically dedicated to I/O (distinct from ordinary memory-access instructions) to send data to, or read data from, a device's registers, addressed in a separate I/O address space. Each unit of data (often a single byte or word) is moved by the CPU executing an explicit instruction, and the CPU typically **polls** the device's status register in a loop to know when it is ready for the next transfer, hence this is also called **polling** or **programmed I/O**. This keeps the CPU fully occupied for the entire duration of the transfer, since it must issue every individual transfer instruction and check status between each one; it is simple to implement but wasteful of CPU time for anything beyond very small or very slow transfers.

Memory-mapped I/O. Here, device registers are mapped into the *same* address space as main memory, so the same load/store instructions the CPU already uses to access memory can be used to read or write a device's registers. No special I/O instructions are needed at all, and any instruction capable of touching memory can manipulate the device. The operating system sets aside a buffer in memory and tells the device to use that buffer for the transfer; the device then operates asynchronously (independently of the CPU) and signals completion with an interrupt when it is done, rather than requiring the CPU to poll continuously. Because it lets ordinary instructions manipulate I/O and supports higher throughput than pure programmed instruction I/O, memory-mapped I/O is used for most high-speed devices such as disk controllers and communication interfaces.

CPU, memory, and a device controller share the system bus; DMA moves data without CPU involvement

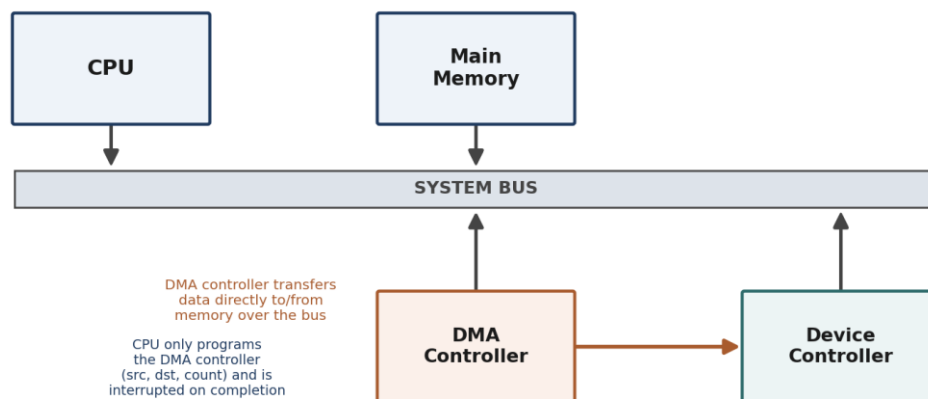


Figure 4.3 — CPU, memory, and a controller sharing a system bus

Direct Memory Access (DMA). Both methods above still require the CPU's attention for every unit transferred (or, at best, for one interrupt per unit in a purely interrupt-driven scheme), which becomes wasteful for high-speed devices moving large blocks of data: a slow device such as a keyboard can be allowed to interrupt the CPU once per character without much overhead, but if a fast device such as a disk generated an interrupt for every single byte, the operating system would spend most of its time simply handling interrupts. **DMA** solves this by delegating the entire transfer to a dedicated piece of hardware, the **DMA controller (DMAC)**, which moves data directly between the device and main memory without the CPU being involved in each individual transfer step at all.

IMPORTANT — Definition — Direct Memory Access (DMA)

Direct Memory Access (DMA) is a hardware mechanism that allows an I/O controller to transfer a block of data directly to or from main memory, bypassing the CPU for the body of the

transfer; the CPU is involved only to initiate the transfer and is interrupted only once, after the entire block has completed, rather than once per unit of data.

The operating system uses DMA hardware through the following sequence of steps:

1. The CPU **programs the DMA controller** with the source address (or device), the destination address in memory, and the count of bytes (or words) to transfer, together with the direction of transfer (device-to-memory or memory-to-device).
2. The CPU then continues executing other work; it is no longer involved in the transfer itself.
3. The **DMA controller arbitrates for the system bus** and transfers the data directly between the device and memory, one unit at a time, incrementing its own address and decrementing its own count register as it goes, entirely independently of the CPU.
4. Once the full count has been transferred, the DMA controller **raises a single interrupt** to notify the CPU that the transfer is complete.

Because the CPU is involved only at the start (programming the controller) and at the very end (handling one completion interrupt), DMA frees the CPU to execute other processes for the entire duration of a large transfer, in sharp contrast to programmed I/O, where the CPU is tied up issuing and monitoring every individual transfer step. This is precisely why DMA, not programmed I/O, is the standard mechanism used for high-volume transfers such as disk reads and writes.

CHECK YOUR PROGRESS 4.3

1. What distinguishes a block device from a character device, and give one example of each.
2. What is a device controller, and how does it relate to a device driver?
3. Why is memory-mapped I/O able to use ordinary load/store instructions to control a device, while programmed instruction I/O cannot?
4. List, in order, the four steps by which the operating system carries out a DMA transfer.
5. Explain why DMA reduces CPU overhead compared with programmed I/O, specifically for a fast device such as a disk.

4.4 PRINCIPLES OF I/O SOFTWARE

4.4.1 Goals of I/O Software

Section 4.3 described the hardware mechanisms by which data physically moves between memory and a device. I/O *software* exists to organise these mechanisms into something an application programmer can use without needing to know which specific hardware sits behind a given request. Well-designed I/O software aims for several goals simultaneously:

- **Device independence.** A program that reads a file should be able to read it from a floppy disk, a hard disk, a network drive, or a CD-ROM without being rewritten for each device: the same `read()` call should work regardless of the underlying hardware.
- **Uniform naming.** Devices (and the files on them) should be named in a consistent way that does not expose physical details such as a specific controller number or bus address to ordinary programs.
- **Error handling.** Errors should, wherever physically possible, be handled as close to the hardware as they occur. For instance, a controller that detects a read error should retry the operation itself before ever troubling higher software layers, which should only need to know whether the operation ultimately succeeded or failed.
- **Synchronous versus asynchronous transfers.** Most application-level I/O calls are **synchronous** (blocking): the calling process is suspended until the transfer completes,

giving the programmer a simple sequential model to reason about. The hardware transfer underneath, however, is usually **asynchronous** (interrupt-driven), and it is the job of the I/O software layers to convert this asynchronous hardware reality back into the blocking illusion the application sees.

- **Buffering.** Data arriving from, or destined for, a device frequently cannot go directly to or from its final resting place (a user process's address space, another device with a different transfer size) and must instead be temporarily held in a **buffer** while the transfer is staged.

4.4.2 The Layered Structure of I/O Software

To meet these goals without duplicating logic across every device, I/O software is conventionally organised into four layers, each hiding a specific kind of complexity from the layer above it. A single request travels *down* through all four layers to reach the hardware, and the hardware's eventual completion signal travels back *up* through them in reverse.

Each layer hides the details of the layer below it from the layer above

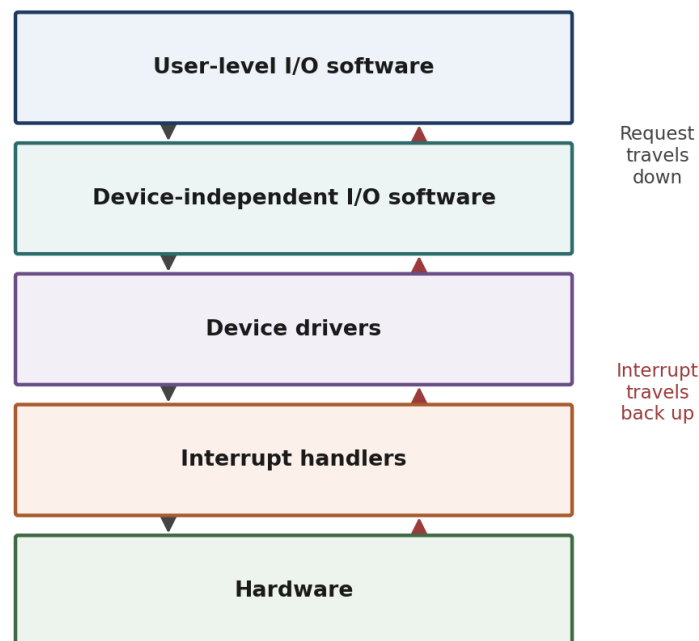


Figure 4.4 — The layered structure of I/O software

1. Interrupt handlers (the bottom layer). When a device completes an operation (or needs attention), it signals the CPU with a hardware **interrupt**. An **interrupt handler**, also called an **interrupt service routine (ISR)**, is the piece of software whose execution is triggered by the reception of a specific interrupt. The interrupt mechanism itself works by presenting the CPU with an address, a number that selects one specific handler out of a small set, usually as an offset into a table called the **interrupt vector table**, which holds the memory address of every registered interrupt handler. When an interrupt occurs, its handler runs, does whatever is needed to service the interrupt (acknowledging the device, updating data structures), and wakes up whichever process was waiting for that particular I/O operation to finish. The interrupt handler's entire purpose in this layered scheme is to hide the mechanics of interrupt handling (saving and restoring processor state, dispatching to the correct handler, restarting the interrupted process) from every layer above it, so that no higher-level software ever has to know an interrupt occurred at all; it only sees that a request it made has now completed.

2. Device drivers. A **device driver** is a device-specific software module, generally written by the device's manufacturer, that is plugged into the operating system to handle exactly one kind (or one closely related family) of device. Everything about a particular piece of hardware that is genuinely unique to it (the exact sequence of register reads and writes its controller requires, its particular

command set, its particular error codes) is encapsulated inside its driver, and nowhere else in the system. A driver's job is threefold: to accept requests from the device-independent layer above it, translated into whatever the specific controller needs; to interact with the device controller to carry the request out, performing whatever device-specific error handling is required; and to report back, once the controller signals completion (typically via the interrupt handler below it), whether the request succeeded. When a request arrives for a driver that is currently idle, it is started immediately; if the driver is already busy servicing another request, the new one is placed on a queue of pending requests for that device. By presenting a **standard interface upward**, regardless of what idiosyncrasies its specific hardware has underneath, the driver is what actually makes device independence in the layer above possible.

3. Device-independent I/O software. Above the drivers sits a layer of code that is genuinely common across *all* devices, so that it need be written only once rather than duplicated inside every individual driver. Its responsibilities include: providing a **uniform interface** to every device driver, so that the layers above never need driver-specific code; **device naming**, mapping mnemonic device names to internal major/minor device numbers that identify which driver, and which specific unit under that driver, a request is destined for; **device protection**, checking that the calling process is permitted to use the device at all; presenting a **device-independent block size** upward, even though different physical devices use different native block sizes underneath; **buffering**, since data coming off a device generally cannot be delivered straight into its final destination; **allocating and releasing dedicated devices**, for devices (such as a tape drive or printer) that can only serve one requesting process at a time; and **error reporting**, translating whatever error a specific driver returned into a standard form the rest of the system understands.

4. User-space I/O software (the top layer). Not all I/O software runs inside the kernel. A substantial amount of it consists of ordinary **library procedures**, linked into the application itself, that present an even simpler, richer interface on top of the system calls the kernel actually exposes — for example, the C standard I/O library (`stdio`), whose familiar `printf()`, `scanf()`, `getchar()`, and `putchar()` functions are themselves implemented in terms of lower-level system calls. One notable exception to the "just a library" pattern is **spooling**: a technique for dealing with a dedicated I/O device (classically, a printer) in a multiprogramming system, where output destined for the device is first written to a disk buffer (a "spool"), letting an application finish quickly and move on, while a separate spooling process or daemon feeds the spooled data to the physical device one job at a time, in order, without the applications that produced it needing to coordinate directly with one another or with the device itself.

Table 4.7 draws the four layers together with their key responsibility and the direction of information flow.

Table 4.7 — I/O software layers and their responsibilities

Layer	Runs in	Key responsibility
User-space I/O software	User space (application/library)	Simple, rich interfaces (<code>stdio</code>) and spooling for shared dedicated devices
Device-independent I/O software	Kernel	Uniform interfacing, naming, protection, buffering, allocation of dedicated devices, error reporting
Device drivers	Kernel	Device-specific command sequences behind a standard upward interface
Interrupt handlers	Kernel (invoked asynchronously)	Hide interrupt mechanics; wake the process/driver waiting on completion

4.4.3 Tracing One Request Through the Layers

It is worth following a single request end to end to see why this layering earns its complexity. Suppose an application calls a library `read()` function to read from a file. The **user-space library** translates this into the appropriate system call and traps into the kernel. The **device-independent**

layer checks protection, maps the request onto the correct device via its device number, arranges a buffer to receive the data, and hands the (now device-specific) request down to the correct **device driver**. The driver either starts the operation immediately or queues it if the device is already busy, and issues the actual commands (via programmed I/O, memory-mapped registers, or by programming a DMA controller, as described in Section 4.3) that the physical hardware needs. The calling process is normally put to sleep at this point (a synchronous, blocking call), freeing the CPU for other work while the transfer proceeds. When the device eventually finishes, it raises an interrupt; the corresponding **interrupt handler** runs, informs the driver that the operation is complete, and wakes the sleeping process. The driver reports success or failure up to the device-independent layer, which delivers the data (now sitting in its buffer) to the user-space library, which finally returns it to the application exactly as an ordinary function return value, with the entire asynchronous, hardware-specific, interrupt-driven journey underneath completely invisible to the application that made the original call. This single trace is also the pattern Section 4.6 reuses when it discusses a clock as simply another interrupt-driven device fitting the same four layers.

4.4.4 Kernel I/O Subsystem Services

The device-independent layer of Section 4.4.2 is often described, particularly in modern kernels, as part of a broader **kernel I/O subsystem** that provides several closely related services on top of individual device drivers, regardless of which specific device a request targets:

- **I/O scheduling.** Beyond the disk-specific scheduling algorithms studied in Section 4.5, the kernel more generally decides a good order in which to service a set of pending I/O requests across possibly several devices, reordering a device's request queue to improve overall throughput and the average response time applications experience, rather than necessarily serving requests in the order they were issued.
- **Buffering,** as already introduced, holds data in memory temporarily while it moves between two entities with different speeds or different transfer-size requirements, decoupling a fast producer from a slow consumer (or vice versa) so that neither has to wait, instruction by instruction, for the other.
- **Caching** keeps a copy of recently used data in a region of fast memory, distinct in purpose from buffering: where buffering exists because two sides of a transfer are not directly compatible, a cache exists purely because accessing the cached copy is faster than repeating the original (usually disk) access: a request that hits in the cache avoids touching the disk at all.
- **Spooling and device reservation,** discussed at the user-space layer in Section 4.4.2, is frequently implemented with kernel support as well: a spool buffer holds output for a device that cannot accept interleaved streams from multiple sources, and the kernel is often what actually reserves a dedicated device for one requesting process at a time.
- **Error handling.** An operating system that runs with protected memory can detect and contain many hardware and application errors (an invalid device response, a corrupted buffer, an out-of-range request) before they can propagate into data corruption or a system-wide failure, reporting a clean error back to the requesting layer instead.

These are best understood not as a fifth layer, but as the concrete set of jobs the device-independent layer of Section 4.4.2 actually performs in a real kernel.

CHECK YOUR PROGRESS 4.4

1. State three distinct goals that good I/O software design tries to achieve.
2. Name the four layers of I/O software, from the hardware upward, and give one responsibility of each.
3. Why does the device-driver layer, rather than the device-independent layer, contain the exact sequence of register reads and writes a controller needs?

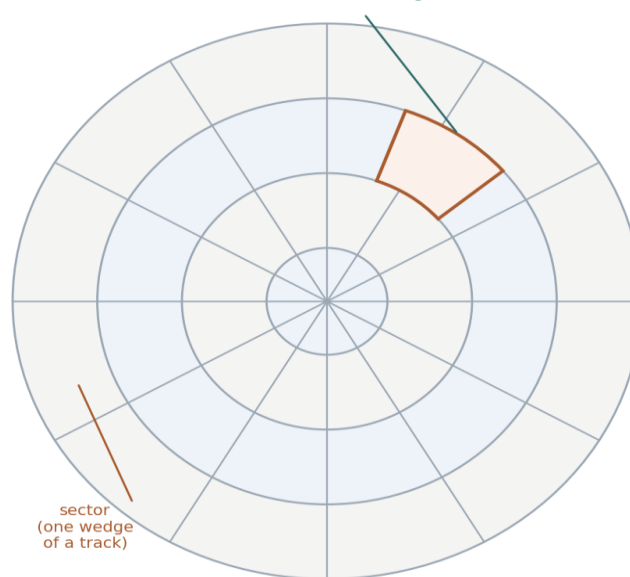
4. What problem does spooling solve, and at which layer does it belong?
5. Trace, in your own words, what happens to a blocked process between the moment its `read()` call is issued and the moment it is woken up.

4.5 DISKS AND DISK SCHEDULING

4.5.1 Disk Hardware

A magnetic disk consists of one or more rigid circular **platters**, coated with a magnetic recording material, mounted on a common spindle that spins continuously at high speed. Each platter's surface is divided into concentric rings called **tracks**, and each track is further divided into fixed-size arcs called **sectors**: the smallest unit the disk can read or write in one operation. Data is read and written by a **read/write head**, mounted on an arm that can move radially across the platter to position itself over any track; where a disk has multiple platters, the set of tracks at the same radial distance on every surface, taken together, is called a **cylinder**, since all the heads (one per surface, all mounted on the same arm assembly) move in and out together and are simultaneously positioned over one cylinder at a time.

Disk geometry: platters divided into concentric tracks, each track into sectors



cylinder = the same track number on every platter surface, stacked

Figure 4.5 — Disk geometry: platters, tracks, sectors, and a cylinder

Servicing any single disk request takes time made up of three components, which together determine the disk's total **access time**:

- **Seek time**: the time for the read/write arm to move from its current track to the track containing the requested data. This is usually the dominant, and most variable, component of access time, since it depends entirely on how far the arm must travel, which is exactly why disk scheduling (Section 4.5.2) focuses on minimising it.
- **Rotational latency**: once the arm is positioned over the correct track, the time spent waiting for the platter to rotate until the desired sector arrives under the head. On average this is half the time for one full revolution, though it can range from zero (the sector happens to already be underneath the head) to a full revolution.

- **Transfer time:** the time actually spent reading or writing the requested data once the head is correctly positioned over it, determined by the amount of data requested and the disk's data transfer rate.

Total access time for one request is therefore, approximately, seek time plus rotational latency plus transfer time; since transfer time for one block is typically small and fixed, and rotational latency averages out over many requests, the effective way an operating system can improve overall disk performance is by minimising the **total seek time**, i.e., the total distance the arm must travel, across a whole queue of pending requests. That is exactly the job of a disk-scheduling algorithm.

4.5.2 Disk Scheduling Algorithms

If the desired disk is idle when a request arrives, it is serviced immediately; otherwise the request is placed on a queue of pending requests, and when the disk becomes free, the operating system must choose which pending request to service next. Because seek time dominates access time, and seek time depends directly on how far the arm must move, the objective of every disk-scheduling algorithm below is the same: **minimise the total distance the read/write arm travels** while servicing a given batch of requests, without introducing unacceptable unfairness or starvation.

EXAMPLE — A common worked example

To compare all six algorithms on equal terms, the same request queue and the same starting head position are used throughout this section: a disk with 200 tracks, numbered 0 to 199. The pending request queue, in the order the requests arrived, is:

95, 180, 34, 119, 11, 123, 62, 64

The read/write head is currently positioned at track 50. Where an algorithm needs an initial direction of travel, the head is taken to be moving toward higher-numbered tracks (that is, "up," toward track 199). A seek rate of 5 ms per track is used wherever a seek *time* (not just head movement in tracks) is wanted.

First Come First Served (FCFS). The simplest possible policy: requests are serviced strictly in the order they arrived in the queue, with no reordering at all. Because arrival order bears no relationship to physical position on the disk, FCFS can force the arm into long, wasteful swings between the innermost and outermost tracks.

Servicing 95, 180, 34, 119, 11, 123, 62, 64 in that exact order from a starting position of 50:

$$|95-50| + |180-95| + |34-180| + |119-34| + |11-119| + |123-11| + |62-123| + |64-62|$$

$$= 45 + 85 + 146 + 85 + 108 + 112 + 61 + 2 = 644 \text{ tracks.}$$

At 5 ms per track, seek time = $644 \times 5 = 3,220$ ms.

Shortest Seek Time First (SSTF). At every step, SSTF selects whichever *pending* request lies closest to the arm's current position, regardless of arrival order, and moves there next, a greedy strategy that generally gives a much lower total seek distance than FCFS, at the risk of **starving** a request that happens to lie far from wherever the arm's recent activity has clustered, since a steady stream of closer requests can keep pushing it to the back indefinitely.

From 50, the nearest pending request is 62 (distance 12); from 62, the nearest remaining request is 64 (distance 2); from 64, the nearest remaining request is 34 (distance 30, versus 31 to 95); from 34, the nearest remaining request is 11 (distance 23); from 11, the nearest remaining request is 95 (distance 84); from 95, the nearest remaining request is 119 (distance 24); from 119, the nearest remaining request is 123 (distance 4); and finally 180 (distance 57) is all that is left.

Sequence: 50 -> 62 -> 64 -> 34 -> 11 -> 95 -> 119 -> 123 -> 180

$12 + 2 + 30 + 23 + 84 + 24 + 4 + 57 = 236$ tracks.

At 5 ms per track, seek time = $236 \times 5 = 1,180$ ms, dramatically better than FCFS on this queue.

SCAN. Also called the **elevator algorithm** because it behaves exactly like a building lift: the arm moves in one direction, servicing every pending request it passes on the way, continues all the way to the last track at that end of the disk even if no request lies there, then reverses direction and services requests on the way back, again travelling in one direction only until it reaches the opposite end of the disk (or until all requests are exhausted).

Moving up from 50: service 62, 64, 95, 119, 123, 180 in that order, continue to the disk's upper boundary at track 199, then reverse and service the remaining requests, 34 and 11, on the way down.

Sequence: 50 -> 62 -> 64 -> 95 -> 119 -> 123 -> 180 -> 199 -> 34 -> 11

$12 + 2 + 31 + 24 + 4 + 57 + 19 + 165 + 23 = 337$ tracks.

At 5 ms per track, seek time = $337 \times 5 = 1,685$ ms.

Circular SCAN (C-SCAN). A variant of SCAN designed to give more uniform waiting times across the whole disk. Like SCAN, the arm sweeps in one direction all the way to the disk's boundary, servicing every request it passes; but instead of reversing and servicing requests on the way back, it jumps immediately back to the *opposite* boundary (these returns jump services no requests) and then resumes sweeping in the *same* original direction. This treats the disk as logically circular, so that tracks near either physical end are not systematically favoured or disadvantaged the way they can be under plain SCAN.

Moving up from 50: service 62, 64, 95, 119, 123, 180, continue to 199, then jump back to 0, then continue upward servicing 11 and 34.

Sequence: 50 -> 62 -> 64 -> 95 -> 119 -> 123 -> 180 -> 199 -> [jump] 0 -> 11 -> 34

$(12 + 2 + 31 + 24 + 4 + 57 + 19) + 199 + 11 + 23 = 149 + 199 + 11 + 23 = 382$ tracks.

At 5 ms per track, seek time = $382 \times 5 = 1,910$ ms. Note the return jump (199 tracks) is counted as real head movement here, since the arm physically travels that distance, even though it services no request while doing so.

LOOK. A practical refinement of SCAN: rather than travelling all the way to the physical end of the disk regardless of whether any request is waiting there, the arm reverses direction as soon as it has serviced the *last actual pending request* in the current direction. It only goes as far as it needs to "look" for more work, never further.

Moving up from 50: service 62, 64, 95, 119, 123, 180 (the last request in the upward direction, so no need to continue to 199), then reverse and service 34 and 11 on the way down.

Sequence: 50 -> 62 -> 64 -> 95 -> 119 -> 123 -> 180 -> 34 -> 11

$(12 + 2 + 31 + 24 + 4 + 57) + 146 + 23 = 130 + 146 + 23 = 299$ tracks.

At 5 ms per track, seek time = $299 \times 5 = 1,495$ ms.

C-LOOK. The circular counterpart of LOOK, combining both refinements: the arm travels only as far as the last pending request in the current direction (as in LOOK), then jumps directly to the *closest* pending request at the other end (rather than all the way to the disk's physical boundary, as C-SCAN does) before resuming its sweep in the original direction.

Moving up from 50: service 62, 64, 95, 119, 123, 180 (the last request going up), then jump directly to the lowest remaining request, 11, and continue upward to 34.

Sequence: 50 -> 62 -> 64 -> 95 -> 119 -> 123 -> 180 -> [jump] 11 -> 34

$(12 + 2 + 31 + 24 + 4 + 57) + 169 + 23 = 130 + 169 + 23 = 322$ tracks.

At 5 ms per track, seek time = $322 \times 5 = 1,610$ ms.

4.5.3 Comparing the Six Algorithms

Table 4.8 collects the total head movement computed above for all six algorithms, on the identical request queue and starting position, so their relative performance can be compared directly.

Table 4.8 — Head movement of six disk-scheduling algorithms

Algorithm	Sequence of tracks visited	Total head movement (tracks)	Seek time at 5 ms/track
FCFS	50-95-180-34-119-11-123-62-64	644	3,220 ms
SSTF	50-62-64-34-11-95-119-123-180	236	1,180 ms
SCAN	50-62-64-95-119-123-180-199-34-11	337	1,685 ms
C-SCAN	50-...-180-199-jump-0-11-34	382	1,910 ms
LOOK	50-62-64-95-119-123-180-34-11	299	1,495 ms
C-LOOK	50-...-180-jump-11-34	322	1,610 ms

A few observations follow directly from these figures. **SSTF gives the lowest total movement** of all six on this particular queue, but it achieves this by always chasing the nearest request, which in general can starve a distant request indefinitely if closer ones keep arriving. That is a real fairness risk FCFS does not have (FCFS is fair by construction, though slow). **SCAN and LOOK are cheaper than their circular counterparts (C-SCAN and C-LOOK)** on this queue, because the circular variants pay the cost of a long return jump across the whole disk to enforce more uniform waiting times across all regions of the disk, a fairness benefit that costs extra total movement. **LOOK improves on SCAN, and C-LOOK improves on C-SCAN**, in both cases by exactly the same idea: never travel further than the last actual pending request demands, rather than always running out to the disk's physical boundary. In practice, LOOK (and its circular form) is the algorithm of choice in most real operating systems, since it captures SCAN's protection against SSTF-style starvation while avoiding SCAN's wasted travel to empty tracks at the disk's edges.

DID YOU KNOW? — Does disk scheduling still matter on an SSD?

A solid-state drive (SSD) has no read/write arm, no platters, and no seek time in the mechanical sense, so the classical goal of minimising arm movement does not apply to it directly. This does not make scheduling irrelevant, however: SSD controllers still benefit from reordering and batching requests, for example through **Native Command Queuing (NCQ)**, which lets the drive accept multiple outstanding requests and internally reorder them for its own efficiency (spreading writes evenly across flash cells, grouping operations to the same internal channel). The specific algorithms of Section 4.5.2 were designed for mechanical disks and remain essential wherever such disks are still used (and as a foundational model for understanding scheduling generally), but a modern system's default I/O scheduler is often configured differently, or disabled in favour of the drive's own internal reordering, when the underlying device is solid-state.

4.5.4 Error Handling on Disks

Disks are mechanical, physical devices, and they fail: a design consideration that scheduling alone cannot address. Three complementary techniques are used to make disk storage reliable despite this.

- **Bad-sector remapping.** When a disk's controller detects that a particular sector has become physically unreliable (it consistently fails to read back correctly), it can mark that sector as bad in a reserved table and transparently substitute one of a small number of spare sectors set aside for exactly this purpose, so the logical block address the operating system uses continues to work correctly even though the physical sector behind it has silently changed.

- **Checksums and error-correcting codes (ECC).** Every sector stores, alongside its data, a small amount of redundant information computed from that data (a checksum, or a stronger error-correcting code) when it is written. When the sector is later read back, the same computation is repeated on the retrieved data and compared against the stored value; a mismatch reveals that the data has been corrupted, and, depending on the strength of the code used, may allow the original data to be reconstructed without needing to re-read the sector at all.
- **Retry on failure.** Many read or write errors are transient rather than permanent, caused by a momentary vibration, a power fluctuation, or dust, and simply retrying the operation a small number of times often succeeds where the first attempt failed. Only after a fixed number of retries have all failed does the controller (and, above it, the operating system) treat the error as a genuine, permanent failure and report it upward rather than continuing to retry indefinitely.

4.5.5 Track-at-a-Time Caching

Because seeking to a track is one of the most expensive operations a disk performs, and because requests for one sector of a track are frequently followed shortly afterward by requests for a *nearby* sector on that same track (sequential file reads being the common case), many disk controllers implement **track-at-a-time caching**: once a seek has brought the head to a given track to satisfy one request, the controller reads the *entire* track into an on-controller cache buffer, not merely the one sector that was actually asked for, since the rotational latency to reach the rest of the track has effectively already been "spent" once the head is there. If a subsequent request turns out to want another sector from the same track, it can be satisfied instantly from the cache, with no further seek and no further wait for the platter to rotate around, since the data is already sitting in fast on-controller memory. This technique costs nothing extra in seek time (the head was going to pass over the whole track regardless, on the way to and past the wanted sector) and can turn what would have been a second full seek-plus-rotation operation into a cache hit answered essentially immediately.

4.5.6 RAM Disks

A **RAM disk** is a portion of main memory (RAM) that the operating system presents to the rest of the system as if it were an ordinary block storage device, complete with a block-addressed interface that a file system can be built on top of exactly as it would on a physical disk.

IMPORTANT — Definition — RAM Disk

A **RAM disk** is a region of main memory configured and accessed through the same block-device interface used for a physical disk, giving programs disk-like storage semantics while the data itself actually resides in much faster, electronically addressed RAM rather than on a mechanical or flash storage medium.

Because RAM has no seek time, no rotational latency, and no mechanical parts at all, a RAM disk is dramatically faster than a physical disk for both reads and writes, often by one or more orders of magnitude, which makes it attractive for storing data that is accessed intensively and does not need to survive a reboot. Its defining trade-off is **volatility**: unlike a physical disk, the contents of a RAM disk are lost the moment the system loses power or is reset, since ordinary RAM (unlike the disk's magnetic or flash medium) does not retain its state without continuous power. RAM disks are therefore used specifically where this trade-off is acceptable or even desirable: as fast, throwaway **scratch or temporary storage** for intermediate results a program does not need to keep; as a **build or compilation cache** where speed matters far more than persistence; and, historically and still today in some systems, as an **early-boot storage environment** (an "initial RAM disk" or `initrd/initramfs`) that holds just enough of a minimal file system and drivers in memory to let

the kernel finish booting and mount its real, persistent root file system, before the RAM disk's temporary contents are discarded.

CHECK YOUR PROGRESS 4.5

1. Define seek time, rotational latency, and transfer time, and state which of the three usually dominates total disk access time.
2. Using the worked example's queue and starting position, which algorithm gave the lowest total head movement, and what fairness risk does that algorithm carry?
3. Explain the one difference between SCAN and LOOK, and the one difference between C-SCAN and C-LOOK.
4. Name the three techniques used to handle disk errors, and briefly explain what each does.
5. Why is a RAM disk much faster than a physical disk, and what is its central limitation?

4.6 CLOCKS

Clocks (also called **timers**) are, in one sense, the simplest I/O devices a computer has: they accept no commands beyond being told when next to interrupt, and they produce no data at all. Yet almost every service an operating system depends on for correct time-based behaviour (pre-emptive scheduling, the wall-clock time reported to applications, timeouts on network connections, alarms an application sets for itself) ultimately rests on this one, unglamorous device. Clocks fit into the same layered I/O software model developed in Section 4.4, and this section closes the unit by showing exactly how.

4.6.1 Clock Hardware

The hardware clock found in essentially every computer is a **programmable interval timer**: a chip that can be loaded with a count, and that decrements this count at a fixed rate (driven by a crystal oscillator), generating a hardware **interrupt** the instant the count reaches zero. The timer can typically be programmed in one of two modes: a **one-shot** mode, where it counts down once and then stops until reprogrammed, or a **square-wave (periodic)** mode, where it automatically reloads its starting count and continues generating an interrupt at a fixed, repeating interval indefinitely, with no further intervention needed from software.

IMPORTANT — Definition — Clock (Interval Timer)

A **clock**, in the operating-system sense, is a programmable interval timer that generates a hardware interrupt after a specified count of oscillator ticks has elapsed, either once or repeatedly, and whose interrupts the operating system uses to drive scheduling, maintain the time of day, and implement software timers.

The periodic clock interrupt is the heartbeat behind several distinct operating-system responsibilities, all driven from the very same underlying hardware signal:

- **Driving the scheduler's timeslice (quantum).** In a pre-emptive, time-shared system, the clock is typically set to interrupt at the end of each scheduling quantum; the clock's interrupt handler is what actually gives the operating system a chance to run and decide whether the currently executing process should be pre-empted in favour of another, exactly the mechanism that makes round-robin and similar pre-emptive scheduling policies possible in the first place.
- **Maintaining the time of day.** Each clock tick can increment a software counter of elapsed ticks; combined with a count of ticks per second and an initial reference time (read from a battery-backed real-time clock at boot, or obtained over a network), this lets the operating

system maintain and report the current wall-clock date and time to applications, without needing a separate device for that purpose.

- **Implementing software timers and alarms.** Applications and kernel subsystems frequently need to be woken up, or to have some action performed, after a specified delay: a network protocol needing to retransmit a packet if no acknowledgement arrives within a timeout, or an application setting an alarm to fire in ten seconds. These are **software timers**, and they are built entirely on top of the one hardware clock, not by giving each requester its own physical timer device.

4.6.2 Supporting Many Software Timers from One Hardware Clock

A real system may have dozens or hundreds of pending software timers active at once, yet the machine typically has only one hardware clock generating interrupts. The standard technique for reconciling this is to keep all currently pending timeouts in a single data structure, usually a **sorted list (or a similar priority-ordered structure) of pending timer events, ordered by absolute expiry time**, and to reprogram the one hardware clock so that its *next* interrupt occurs exactly when the *soonest* pending timer in that list is due to expire, rather than ticking at some fixed, unrelated rate.

When the clock interrupt fires, the handler removes the (now-expired) timer at the head of the sorted list, performs whatever action that timer represents (waking a sleeping process, retransmitting a packet, invoking a callback), and then looks at the new head of the list, the next-soonest remaining timer, and reprograms the hardware clock to interrupt again exactly at that timer's expiry time. In this way, a single physical interval timer can transparently support an arbitrary number of independent logical timers, with the clock interrupt handler doing the bookkeeping of always waking up exactly when the nearest deadline in the whole system is reached, and never any sooner than necessary.

EXAMPLE — Two pending timers sharing one clock

Suppose, at time T , one software timer is due to expire at $T + 40$ ms (a network retransmission timeout) and another at $T + 15$ ms (an application alarm). The sorted list holds the $T + 15$ entry at its head, so the hardware clock is programmed to interrupt at $T + 15$. When that interrupt fires, the handler processes the alarm, removes it from the list, and finds the retransmission timer ($T + 40$, now the new head) is the next one due; it reprograms the hardware clock to interrupt at $T + 40$ next, and the cycle continues, one hardware interrupt per expiring timer, in strict chronological order.

4.6.3 Memory-Mapped Terminals and Serial-Line Terminals

The term **memory-mapped terminal** describes a specific, historically important way of driving a text-mode display, and it is best understood by contrasting it with the alternative, interrupt-driven approach used for a serial terminal.

A **memory-mapped terminal** (the approach used by a PC's classic text-mode video display, among others) works by dedicating a region of the address space to a **video buffer** that the display hardware continuously and independently scans out to the screen. Writing a character to the appropriate location in this buffer is enough to make it appear on the screen: the display hardware reads directly from that memory region on its own, with no CPU interrupt and no device driver call needed for each character shown. Output to such a terminal is therefore extremely cheap: it is just an ordinary memory store instruction, and no separate I/O transaction has to be arranged at all.

A **serial-line terminal**, in contrast, is a genuinely separate physical device connected over a communication line, typically managed through a **UART** (universal asynchronous receiver/transmitter) chip. Because the UART transmits and receives one character at a time over the line, at a rate far slower than the CPU, output to a serial terminal cannot simply be written into

memory the way a memory-mapped display's buffer can: each character must be handed to the UART's transmit register, and, following exactly the layered, interrupt-driven pattern of Section 4.4, the UART typically raises an interrupt once it is ready to accept the *next* character (or once it has received one), with the interrupt handler and device driver cooperating to feed characters out (or collect characters in) one at a time as the hardware becomes ready for each.

Table 4.9 — Memory-mapped terminals versus serial-line terminals

Property	Memory-mapped terminal	Serial-line terminal
Output mechanism	Ordinary store into a shared video buffer	Character-at-a-time transfer via a UART
CPU involvement per character	None: display hardware reads the buffer independently	One interrupt (or poll) typically needed per character
Typical use	Local text-mode console/display	Remote or externally connected terminal, modem link

4.6.4 Clocks Within the I/O Software Layering

A clock fits the same four-layer I/O software structure introduced in Section 4.4 more simply than almost any other device, precisely because it accepts no commands and returns no data. It only ever needs a bottom-layer **interrupt handler**, invoked each time the hardware timer's count reaches zero, which does the sorted-timer-list bookkeeping described in Section 4.6.2, decides what (if anything) needs to be woken up or invoked, and reprograms the hardware for its next interrupt. Above that, a thin **driver-like layer** exposes a small set of kernel services ("wake me in *n* milliseconds," "tell me the current time," "decrement this process's remaining quantum") to the rest of the operating system, and higher software (the scheduler, networking code, an application's own alarm call) simply consumes those services without ever needing to know that a hardware interval timer, ticking away in the background, is what makes every one of them possible. A clock is, in this sense, the cleanest illustration in the whole unit of how the layered I/O software structure hides one small piece of asynchronous hardware behind a simple, synchronous-looking service that the rest of the system can just call.

CHECK YOUR PROGRESS 4.6

1. What is a programmable interval timer, and what are its two typical operating modes?
2. Name three distinct operating-system services that are all driven from the same underlying hardware clock interrupt.
3. Explain how a single hardware clock can support many independent software timers without needing one physical timer per pending timeout.
4. Distinguish a memory-mapped terminal from a serial-line terminal in terms of how a character actually reaches the display.
5. Which single layer of the I/O software model (Section 4.4) does a clock's interrupt handler belong to, and why does a clock need comparatively little software above that layer?

4.7 SUMMARY

A file is the operating system's basic unit of persistent storage: a name, a type, a size, permissions, timestamps, and a handful of operations like create, read, write, and delete. Programs can read a file sequentially, jump straight to a position, or use an index that gives them both. Files sit inside directories, and directory structures have grown more flexible over time, from one flat list, to separate folders per user, to a tree with proper paths, to graphs that allow real sharing, at the cost of needing reference counting or cycle detection to delete things safely.

Underneath that interface, a file system has to decide how to actually place data on disk. Contiguous, linked, and indexed allocation each make a different trade-off between fast access, easy growth, and

fragmentation. Free space gets tracked with bit vectors, linked lists, or grouping, and directories themselves are usually implemented as either a simple linear list or a hash table. Protection ranges from detailed access-control lists down to the simpler, cheaper owner/group/other permission bits, backed up by authentication methods like one-time passwords and an awareness of the usual threats, from Trojans and logic bombs to worms and denial-of-service attacks.

On the hardware side, devices are either block or character devices, talking to the CPU through a controller, and the CPU can reach them through programmed I/O, memory-mapped I/O, or DMA, which frees the CPU from moving data one word at a time. The software side mirrors this in layers: interrupt handlers at the bottom hide the mechanics of an interrupt, device drivers hide what's specific to one device, device-independent software provides services every device needs, and user-space libraries sit on top, with a single read request shown travelling down through all four layers and the answering interrupt travelling back up.

We looked at disks in detail too. Seek time, rotational latency, and transfer time make up how long a request actually takes, and we ran all six classic scheduling algorithms, FCFS, SSTF, SCAN, C-SCAN, LOOK, and C-LOOK, on one identical example to see the trade-offs directly: SSTF moves the arm the least but can starve far-away requests, while LOOK and C-LOOK improve on SCAN and C-SCAN by not bothering to travel all the way to the disk's edge. Error handling relies on remapping bad sectors and checksums; track-at-a-time caching reuses a seek that's already been done; and RAM disks trade persistence for raw speed.

Clocks closed out the chapter: one hardware timer, through a sorted list of pending timeouts, can support scheduling, wall-clock time, and every software timer a system needs, and memory-mapped terminals turned out to be the simplest possible example of the same I/O layering used everywhere else in this chapter.

Put together, this chapter is about the point where an operating system's clean internal abstractions, processes, addresses, scheduling, finally meet hardware that is slow, mechanical, and prone to failure. Every layer covered here, from a well-organised directory tree down to a disk-scheduling algorithm, exists for one reason: so that an application programmer can just open a file or write a byte to a terminal, while the operating system quietly absorbs everything difficult underneath.

4.8 GLOSSARY

Term	Definition
Access-control list (ACL)	A per-object list of (subject, permission) pairs stating exactly which rights each user or group holds on that object
Access matrix	A conceptual matrix with subjects as rows and objects as columns, each cell holding the rights that subject has over that object
Acyclic-graph directory	A directory structure allowing a file or sub-directory to be referenced from more than one place, provided no cycles result
Allocation (file)	The mapping from a file's logical blocks to the physical disk blocks that actually hold its data
Bad-sector remapping	Transparently substituting a spare sector for one the controller has detected as physically unreliable
Bit vector (bitmap)	A free-space management technique using one bit per disk block to mark it free or allocated
Block device	A device that stores and transfers information in fixed-size, independently addressable blocks, supporting random access
C-LOOK	A disk-scheduling algorithm that services requests up to the last one in the current direction, then jumps to the closest remaining request at the other end
C-SCAN	A disk-scheduling algorithm that sweeps to the disk's boundary, then jumps back to the opposite boundary before resuming in the same direction
Character device	A device that delivers or accepts an unstructured stream of characters, with no addressable blocks and no seek operation

Term	Definition
Cylinder	The set of tracks at the same radial position across every surface of a multi-platter disk
Denial of service (DoS)	An attack aimed at preventing legitimate use of a system or service, typically by overwhelming it with requests
Device controller	The electronic component of an I/O unit that acts as the interface between the physical device and the rest of the computer
Device driver	Device-specific software presenting a standard interface upward while handling one device's particular hardware detail
Direct memory access (DMA)	A hardware mechanism letting a controller transfer data directly to/from memory, interrupting the CPU only on completion
Directory	A structure mapping file names to the information needed to find and use each file
External fragmentation	Free disk space broken into scattered pieces too small individually to satisfy a request, despite adequate total free space
FCFS (disk scheduling)	Servicing pending disk requests strictly in the order they arrived
File	A named collection of related information recorded on secondary storage, treated as the smallest logical unit of persistent storage
General graph directory	A directory structure permitting arbitrary links, including cycles, requiring cycle detection and garbage collection
Indexed allocation	A file allocation method using one index block per file holding pointers to that file's data blocks
Indexed sequential access	A file access method combining a sequential ordering with an index that narrows a search before a short scan
Internal fragmentation	Wasted space in the last (partially used) block allocated to a file
Interrupt handler (ISR)	The bottom-layer routine invoked when a specific hardware interrupt occurs, hiding interrupt mechanics from higher layers
Linked allocation	A file allocation method chaining a file's blocks together via a next-block pointer stored in each block
LOOK	A disk-scheduling algorithm that reverses direction as soon as the last pending request in the current direction is serviced
Memory-mapped I/O	A communication scheme mapping device registers into the same address space as memory, usable with ordinary load/store instructions
Memory-mapped terminal	A display driven by writing characters into a shared video buffer that the display hardware reads directly
One-time password (OTP)	A password valid for exactly one login, defeating replay attacks that would succeed against a reusable password
Port scanning	Probing a target system's network ports to discover running services and identify vulnerabilities
Programmed (special instruction) I/O	A communication scheme using dedicated CPU instructions and active polling to transfer data to/from a device
RAM disk	A region of main memory presented to the OS as a block storage device, fast but volatile
Reference count (link count)	A count of how many directory entries point to a file, used to decide when its storage can safely be reclaimed
Rotational latency	The time spent waiting for the desired sector to rotate under the read/write head once the arm is on the correct track
SCAN	A disk-scheduling algorithm (the "elevator" algorithm) that sweeps to the disk's physical boundary before reversing direction
Seek time	The time for the disk arm to move from its current track to the track holding the requested data
Sector	A fixed-size arc of a disk track, the smallest unit the disk can read or write in one operation
Sequential access	A file access method processing records strictly in order, one after another
Spooling	Buffering output for a dedicated device on disk so a producing application need not coordinate directly with the device

Term	Definition
SSTF	A disk-scheduling algorithm always selecting the pending request closest to the arm's current position
Track	A concentric ring on a disk platter's surface, divided into sectors
Track-at-a-time caching	Caching an entire track after a seek, so nearby subsequent requests on the same track are served without a further seek
Trap door (backdoor)	A hidden entry point left in a program that bypasses normal access checks
Trojan horse	A program that performs a hidden malicious action while appearing to perform a legitimate one
UART	A chip that transmits and receives characters one at a time over a serial communication line
Worm	A self-replicating process that consumes system resources so aggressively it starves other processes or the network

CHAPTER 5

Information Security, Mobile Commerce, and Cloud Computing

5.1 INTRODUCTION TO INFORMATION SECURITY

5.1.1 What is Information Security?

Every organisation, and increasingly every individual, depends on information stored and processed by computers: bank balances, medical records, examination results, trade secrets, private conversations. **Information security**, often abbreviated **InfoSec**, is the discipline concerned with protecting this information (wherever it is stored, however it is processed, and whenever it travels across a network) from being seen, changed, or destroyed by anyone who should not have that power.

IMPORTANT — Definition — Information Security

Information security is the practice of protecting information and information systems from unauthorised access, disclosure, modification, or destruction, in order to preserve their confidentiality, integrity, and availability.

Information needs protecting for the same reason anything of value does: it attracts people who want to acquire, alter, or destroy it without permission, whether for financial gain, competitive advantage, political motive, or simple malice. Unlike a physical asset, though, information can be copied without the owner's knowledge, altered without visible damage, and attacked from anywhere in the world. This combination of high value and low physical protection is what makes information security a distinct discipline rather than an extension of physical security.

5.1.2 The CIA Triad

Nearly every information security objective can be traced back to three properties, together known as the **CIA triad** (not to be confused with any intelligence agency).

Table 5.1 — The CIA triad

Property	What it means	What fails without it
Confidentiality	Information is disclosed only to those authorised to see it	An unauthorised party reads sensitive data (for example, a competitor obtains a company's pricing strategy)
Integrity	Information remains accurate and is modified only by authorised parties, in authorised ways	Data is altered, whether maliciously or accidentally, and the alteration goes undetected (for example, a bank balance is silently changed)
Availability	Information and the systems that provide it are accessible to authorised users when needed	A legitimate user cannot access a system or data when required (for example, a hospital's patient-record system is knocked offline during an emergency)

EXAMPLE — Applying the CIA triad to one scenario

Consider an online examination portal. **Confidentiality** is violated if a student obtains another student's answers before submission. **Integrity** is violated if a student's submitted score is altered after grading. **Availability** is violated if the portal crashes during the examination window, preventing legitimate submissions. A complete security design for the portal must address all three properties. Strengthening only one leaves the system exposed through the others.

Two further properties are frequently discussed alongside the CIA triad and matter directly to Sections 1.4 and 1.6 of this unit: **authentication** (confirming that a user or system is who it claims to be) and **non-repudiation** (ensuring that a party cannot later deny having performed an action, such as sending a message or authorising a transaction).

5.1.3 Threats, Vulnerabilities, and Risk

Information security decisions are best understood through three related terms that are often, incorrectly, used interchangeably.

- A **threat** is any potential cause of harm: a hacker, a malicious program, a natural disaster, or even an untrained employee.
- A **vulnerability** is a weakness in a system that a threat can exploit: an unpatched operating system, a weak password policy, or a firewall rule that is too permissive.
- **Risk** is the likelihood that a given threat will exploit a given vulnerability, combined with the resulting impact if it does. Formally, risk is often expressed as a function of threat, vulnerability, and impact: reducing any one of the three reduces overall risk.

Risk management is the ongoing process of identifying threats and vulnerabilities, estimating the resulting risk, and applying controls (technical, procedural, or physical) to reduce that risk to a level the organisation finds acceptable. Risk can rarely be reduced to zero; the aim of information security is to manage it to an acceptable, cost-justified level, not to eliminate it entirely.

THINK ABOUT IT — Think About It

A company stores customer data on a server that has not been patched in two years (a vulnerability) and is a known target for ransomware gangs (a threat). Before reading further in this unit, list two different kinds of control, one technical and one procedural, that would each reduce the resulting risk without eliminating the vulnerability outright.

5.1.4 The Scope of This Part

Part A builds outward from these foundations. Section 5.2 looks at the threats themselves: malicious programs. Sections 1.3 and 1.4 cover cryptography and digital signatures, the mathematical controls that protect confidentiality, integrity, authentication, and non-repudiation. Section 5.5 turns to firewalls, a network-level control, and Section 5.6 explains how systems verify identity. Section 5.7 deals with the human and organisational side of security, awareness and policy, without which technical controls alone are insufficient. Section 5.8 then surveys the domains where all of this matters most in practice.

CHECK YOUR PROGRESS 5.1

1. Define information security and name the three properties of the CIA triad.
2. Distinguish a threat from a vulnerability, and explain how the two combine to produce risk.
3. Why can risk rarely be reduced to zero, and what is the realistic goal of risk management instead?
4. Explain, with an example, why availability is as much a security property as confidentiality.

5.2 MALICIOUS PROGRAMS

5.2.1 What is Malware?

Malware (a contraction of *malicious software*) is any software deliberately written to damage, disrupt, or gain unauthorised access to a computer system, network, or device, without the informed consent of the system's owner. Malware is the most direct embodiment of a "threat" as defined in Section 5.1.3: it is code written specifically to exploit vulnerabilities and cause harm.

Malware is classified chiefly by two independent properties: **how it spreads** (does it need a host file, a network vulnerability, or a deceived human?) and **what it does once installed** (does it destroy data, steal information, extort money, or merely annoy?). For examination and practical purposes, understanding both properties for a given piece of malware is more useful than memorising definitions in isolation.

5.2.2 Types of Malicious Programs

Table 5.2 — Types of malicious programs

Type	How it spreads	What it does
Virus	Attaches itself to a legitimate file or program; spreads when the infected file is shared or executed	Replicates to other files; may corrupt data, steal information, or disable system functions
Worm	Self-replicates and spreads across a network on its own, exploiting protocol or software vulnerabilities, without needing to attach to a host file	Consumes network bandwidth and system resources; often carries a separate malicious payload
Trojan (Trojan horse)	Disguises itself as legitimate, desirable software, tricking the user into installing it voluntarily	Performs hidden malicious actions: stealing data, installing further malware, or giving the attacker remote access
Ransomware	Typically delivered via a Trojan, a malicious email attachment, or an exploited vulnerability	Encrypts the victim's files, or locks them out of the system, and demands payment (usually in cryptocurrency) for restoration
Spyware	Often bundled with other software, or installed via a Trojan	Silently monitors user activity (keystrokes, browsing habits, login credentials) and exfiltrates the collected data
Adware	Often bundled with free software	Displays unwanted advertisements; not always malicious in intent, but frequently compromises privacy by tracking behaviour

NOTE — Virus versus worm

The distinction most often confused in examinations is virus versus worm. A **virus** needs a host: a file or program to attach to, and typically some human action (opening the file) to spread. A **worm** needs no host and no human action; it spreads autonomously by exploiting a vulnerability across a network. This is why worms can spread far faster than viruses.

5.2.3 Real-World Examples

Studying documented, real incidents makes the abstract categories in Table 5.2 concrete, and such incidents are a common source of examination questions.

Table 5.3 — Documented malware incidents

Malware	Year	Type	Notable impact
WannaCry	2017	Ransomware worm	Spread to over 150 countries by exploiting a Windows vulnerability; encrypted files across hospitals, businesses, and government systems and demanded Bitcoin ransoms
Stuxnet	2010	Worm (targeted, state-linked)	Sabotaged industrial control systems at Iran's uranium enrichment facilities by causing centrifuges to malfunction;

Malware	Year	Type	Notable impact
			among the first publicly documented examples of malware causing physical damage
Mirai	2016	Botnet malware	Infected poorly secured IoT devices (routers, cameras, DVRs) using default credentials, then used them to launch large-scale distributed denial-of-service (DDoS) attacks
Zeus (Zbot)	2007 onward	Banking Trojan	Stole banking credentials via keystroke logging, distributed largely through phishing emails; one of the most persistent and widely distributed pieces of financial malware in history

DID YOU KNOW? — A recent, costly example

In 2025, a ransomware attack on the automotive manufacturer Jaguar Land Rover forced a shutdown of production across its plants for over a month, disrupting the wider supply chain. The UK's Cyber Monitoring Centre assessed its economic impact at close to £2 billion, describing it as one of the most costly cyber incidents in British corporate history. It is a reminder that ransomware's damage today extends well beyond the computers it directly infects, into physical manufacturing and entire supply chains.

5.2.4 General Defences Against Malware

While specific controls (firewalls, authentication) are covered later in this Part, certain defences apply generally against malware of every type in Table 5.2:

- **Keeping software patched.** Most worms and many Trojans exploit *known* vulnerabilities for which a patch already exists; timely patching closes this route before it can be exploited.
- **Least privilege.** Running everyday tasks with a non-administrator account limits the damage a successful infection can do.
- **Backups, kept offline or immutable.** Since ransomware's threat depends on the victim having no other copy of their data, a regularly tested backup that ransomware cannot also encrypt removes most of its leverage.
- **User education.** Since Trojans and much ransomware rely on a human being deceived into running them, informed users are themselves a control (this theme is developed further in Section 5.7).
- **Anti-malware software and network monitoring.** Signature- and behaviour-based detection tools catch known and, increasingly, previously unseen malware before it executes.

CHECK YOUR PROGRESS 5.2

1. Distinguish a virus from a worm in terms of how each spreads.
2. Which type of malware from Table 5.2 relies most directly on deceiving the user, rather than exploiting a technical vulnerability?
3. Explain why regular, offline backups are an effective defence specifically against ransomware.
4. Name one malware incident from Table 5.3 and state, in one sentence, why it is significant beyond its immediate technical effect.

5.3 CRYPTOGRAPHY

5.3.1 What is Cryptography?

Cryptography is the study and practice of techniques that allow information to be exchanged securely even when the communication channel itself is not trustworthy: even when an adversary can observe, and sometimes alter, everything sent over that channel. Cryptography achieves this by transforming a readable message, called **plaintext**, into an unreadable form, called **ciphertext**, using a mathematical procedure called an **encryption algorithm** together with a piece of secret information called a **key**. The reverse transformation, recovering the plaintext from the ciphertext, is **decryption**, and requires the appropriate key.

IMPORTANT — Definition — Cryptography

Cryptography is the practice of using mathematical algorithms and keys to transform data so that it can only be read, verified, or trusted by parties possessing the appropriate key, thereby protecting confidentiality, integrity, authentication, and non-repudiation.

Cryptography is needed wherever information must cross an untrusted medium, which, for practical purposes, means almost every network in use today. Without cryptography, anyone able to intercept network traffic (an attacker on the same Wi-Fi network, a compromised router along the path, an internet service provider) could read or alter every password, payment, and private message that passed through it.

A cryptographic system's security should depend entirely on the secrecy of its **key**, not on the secrecy of its algorithm. This principle, known as **Kerckhoffs's principle**, exists because an algorithm is difficult to keep secret over the long term (it is embedded in widely distributed software and hardware), whereas a key can be generated freshly, kept secret, and changed if compromised. Every algorithm named in this section is public and has been extensively studied by the cryptographic research community; the security of a system built with it comes entirely from protecting its keys.

5.3.2 Symmetric-Key Cryptography

In **symmetric-key cryptography**, the same secret key is used for both encryption and decryption. Both the sender and the receiver must possess this shared key before communication begins, and it must never be disclosed to anyone else.

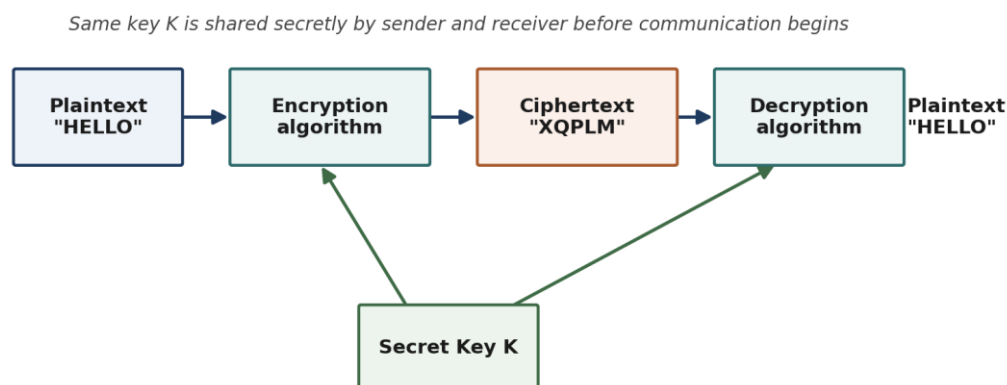


Figure 5.1 — Symmetric-key encryption and decryption

EXAMPLE — A simplified illustration

Suppose the shared key is the number 3, and "encryption" means shifting each letter of the plaintext forward by that many positions in the alphabet (a Caesar cipher, far too weak for real use but useful for building intuition). The plaintext HELLO encrypts to the ciphertext KHOOR. Decryption simply reverses the shift: knowing the key (3) and the direction, the receiver recovers HELLO from KHOOR. Real symmetric algorithms replace this simple shift with mathematically complex, extensively analysed transformations, but the essential structure (one shared secret key, used for both directions) is identical.

The main advantage of symmetric-key cryptography is speed: symmetric algorithms are computationally far cheaper than the public-key algorithms described in Section 5.3.3, which is why they are used to encrypt the *bulk* of data in most real systems. Its main disadvantage is **key distribution**: the two parties must somehow agree on a shared secret key before they can communicate securely, and doing this over an untrusted network is itself a hard problem, one that public-key cryptography was invented to solve (Section 5.3.4).

Table 5.4 — Symmetric-key algorithms compared

Algorithm	Key length	Status (2026)
DES (Data Encryption Standard)	56 bits	Obsolete: breakable by brute force with modern hardware; retained only for historical/educational reference
3DES (Triple DES)	Effectively 112 bits	Deprecated: being phased out of new systems in favour of AES
AES (Advanced Encryption Standard)	128, 192, or 256 bits	Current standard; used almost universally for symmetric encryption today

5.3.3 Cryptographic Hashing

Before introducing public-key cryptography, one more building block is needed, because it underlies both authentication (Section 5.6) and digital signatures (Section 5.4). A **cryptographic hash function** takes an input of any size and produces a fixed-size output, called a **hash value**, **digest**, or **message digest**, with three properties: it is *deterministic* (the same input always produces the same digest), it is *one-way* (it is computationally infeasible to reconstruct the input from the digest alone), and it is *collision-resistant* (it is computationally infeasible to find two different inputs that produce the same digest). Widely used hash functions include the SHA-2 family (such as SHA-256) and the newer SHA-3; the older MD5 and SHA-1 are now considered broken for security purposes because practical collisions have been demonstrated against them.

Hashing is not encryption. There is no key, and no way to "decrypt" a hash value back into its input, because a hash function is a many-to-one mapping rather than a reversible transformation. Its usefulness lies precisely in this one-way property, since a hash value can act as a compact, unforgeable "fingerprint" of a much larger piece of data.

5.3.4 Public-Key (Asymmetric) Cryptography

Public-key cryptography, also called **asymmetric cryptography**, solves the key-distribution problem of Section 5.3.2 by using a mathematically related *pair* of keys instead of one shared key: a **public key**, which can be freely published, and a **private key**, which must be kept secret by its owner alone. Data encrypted with one key of the pair can only be decrypted with the *other* key of the same pair.

The key pair is mathematically related but the private key cannot feasibly be derived from the public key

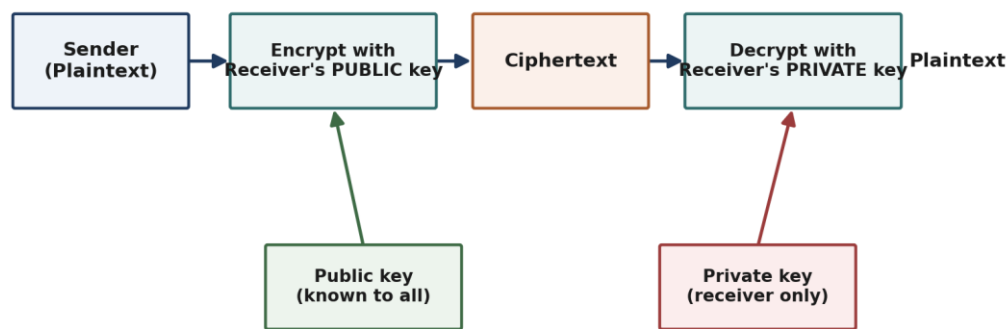


Figure 5.2 — Public-key (asymmetric) encryption and decryption

This gives public-key cryptography two distinct uses, depending on which key encrypts:

- **Confidentiality.** If Alice wants to send Bob a confidential message, she encrypts it with **Bob's public key**. Only Bob, holding the matching private key, can decrypt it. Even Alice cannot decrypt her own ciphertext once sent.
- **Authentication and signatures.** If Bob encrypts something with his **own private key**, anyone holding Bob's public key can decrypt it, which proves the message could only have come from Bob, since only Bob possesses that private key. This use is the mathematical basis of digital signatures (Section 5.4).

IMPORTANT — A worked example — the arithmetic idea behind RSA

RSA, the most widely known public-key algorithm, rests on the fact that multiplying two large prime numbers is computationally easy, while factoring their product back into the original primes is, for sufficiently large primes, computationally infeasible with current classical computers. A tiny (and therefore insecure) numeric example illustrates the mechanism:

Choose two primes $p = 3$ and $q = 11$. Compute $n = p * q = 33$ and $\phi(n) = (p-1)(q-1) = 2 * 10 = 20$. Choose a public exponent e that shares no common factor with $\phi(n)$; here $e = 3$ works. Compute the private exponent d such that $(d * e) \bmod \phi(n) = 1$; here $d = 7$, since $7 * 3 = 21 = 1 \bmod 20$. The **public key** is $(e, n) = (3, 33)$ and the **private key** is $(d, n) = (7, 33)$.

To encrypt a plaintext number $m = 4$: ciphertext $c = m^e \bmod n = 4^3 \bmod 33 = 64 \bmod 33 = 31$. To decrypt: $m = c^d \bmod n = 31^7 \bmod 33 = 4$, recovering the original plaintext. Real RSA uses primes hundreds of digits long, making factorisation infeasible; the arithmetic relationship demonstrated here, at a scale that can be checked by hand, is exactly the same relationship used in practice.

Public-key cryptography is far more computationally expensive than symmetric-key cryptography, so real systems rarely use it to encrypt bulk data directly. Instead, most secure communication protocols (such as HTTPS) use a **hybrid approach**: public-key cryptography is used only to securely exchange a randomly generated symmetric key, after which the much faster symmetric algorithm encrypts the actual data for the rest of the session.

Table 5.5 — Public-key algorithms compared

Algorithm	Typical use	Basis of security
RSA	Encryption, digital signatures	Difficulty of factoring the product of two large primes
Diffie-Hellman	Secure key exchange over an untrusted channel	Difficulty of the discrete logarithm problem

Algorithm	Typical use	Basis of security
Elliptic Curve Cryptography (ECC)	Encryption, digital signatures, key exchange	Difficulty of the elliptic-curve discrete logarithm problem; offers equivalent security to RSA with much shorter keys

DID YOU KNOW? — Post-quantum cryptography

A sufficiently powerful quantum computer could, in principle, break RSA, Diffie–Hellman, and ECC by solving the factoring and discrete-logarithm problems far faster than any classical computer. That threat does not yet exist at practical scale, but it is taken seriously because encrypted data captured today could be decrypted retroactively once such a computer exists. In August 2024, the US National Institute of Standards and Technology (NIST) finalised the first three **post-quantum cryptography** standards: ML-KEM (for key exchange), ML-DSA, and SLH-DSA (for digital signatures), designed to resist attack by both classical and quantum computers. Security agencies have recommended that organisations with long-lived sensitive data begin planning a transition well before 2030.

5.3.5 Advantages and Limitations of Cryptography

Table 5.6 — Advantages and limitations of cryptography

Advantages	Limitations
Protects confidentiality even over a completely untrusted network	Does not protect against a compromised endpoint: if an attacker controls the sender's or receiver's device, encryption is bypassed
Public-key cryptography solves key distribution without a pre-shared secret	Public-key operations are computationally expensive relative to symmetric ones
Modern algorithms (AES, RSA, ECC) are extensively analysed and, when correctly implemented, computationally very strong	Incorrect implementation (weak random-number generation, poor key management) can undermine even a strong algorithm
Provides a mathematical basis for authentication and non-repudiation (Sections 1.4 and 1.6)	Long-term security must anticipate future threats such as quantum computing

CHECK YOUR PROGRESS 5.3

1. Distinguish symmetric-key from public-key cryptography in terms of how many keys each uses and who holds them.
2. Why is a cryptographic hash function not a form of encryption?
3. In the RSA worked example, if the public key is (3, 33), explain in your own words why an attacker who intercepts the ciphertext cannot easily recover the plaintext without the private key (7, 33).
4. Explain why most secure network protocols use a hybrid of symmetric and public-key cryptography rather than public-key cryptography alone.

5.4 DIGITAL SIGNATURES

5.4.1 What is a Digital Signature?

A handwritten signature on a paper document serves two purposes: it identifies who signed the document, and it signals that the signer agreed to its contents at the time of signing. A **digital signature** provides the electronic equivalent of both purposes for digital documents and messages. Because it is built from the cryptographic tools introduced in Section 5.3, it does so with

mathematical guarantees that a handwritten signature cannot offer: a handwritten signature can be forged by copying its shape, but a digital signature cannot be produced without possessing the signer's private key.

IMPORTANT — Definition — Digital Signature

A **digital signature** is a cryptographic value, computed from a message and the sender's private key, that allows any recipient holding the sender's public key to verify both the identity of the sender and the integrity of the message.

Digital signatures combine two ideas already introduced in this unit: cryptographic hashing (Section 5.3.3), which produces a compact, tamper-evident fingerprint of the message, and public-key cryptography (Section 5.3.4), specifically its authentication use: encrypting with a private key so that anyone with the matching public key can verify the result.

5.4.2 How Digital Signatures Work

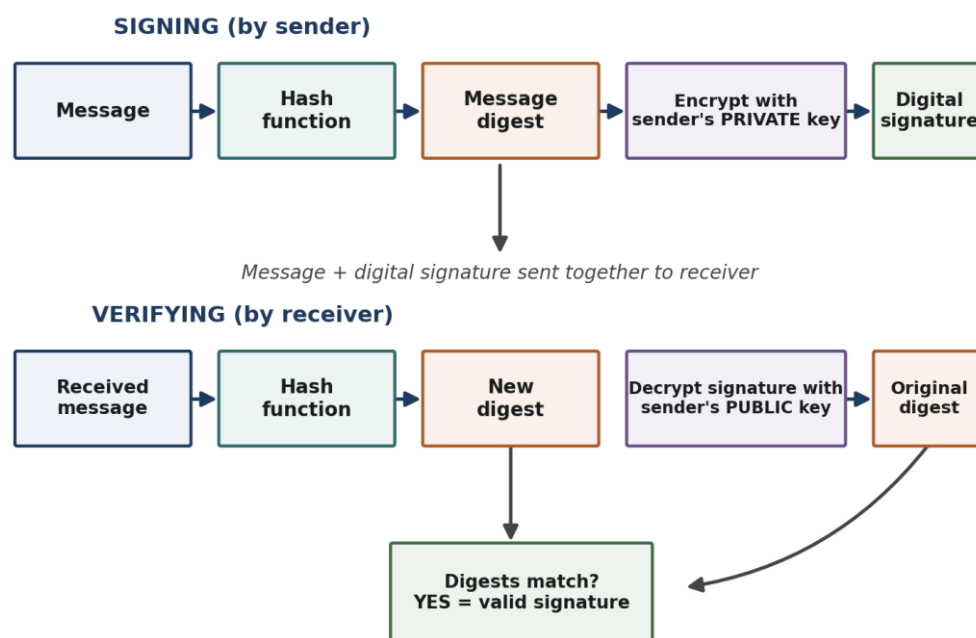


Figure 5.3 — Creating and verifying a digital signature

Signing (performed by the sender):

1. **Compute the message digest.** The sender applies a cryptographic hash function to the message, producing a fixed-size digest that uniquely represents its content.
2. **Encrypt the digest with the private key.** The sender encrypts this digest using their own private key. The result is the digital signature.
3. **Attach and send.** The sender transmits the original message together with the digital signature.

Verification (performed by the recipient):

4. **Decrypt the received signature.** The recipient uses the sender's public key to decrypt the signature, recovering the digest the sender originally computed.
5. **Recompute the digest.** The recipient independently applies the same hash function to the received message, producing a second digest.

6. **Compare.** If the decrypted digest (step 4) matches the freshly computed digest (step 5), verification succeeds: the message has not been altered since signing, and it could only have been signed by the holder of the corresponding private key.

EXAMPLE — Why tampering is detected

Suppose an attacker intercepts a signed message and alters even a single character of its content before forwarding it. Because a cryptographic hash function is extremely sensitive to its input (changing even one bit produces a completely different digest), the digest the recipient computes in step 5 will not match the digest recovered in step 4. Verification fails, and the tampering is detected, without the attacker needing to know anything about the private key at all.

5.4.3 What Digital Signatures Guarantee

Table 5.7 — What a digital signature guarantees

Property	What it means here
Authentication	Only the holder of the private key could have produced a signature that verifies correctly with the matching public key
Integrity	Any change to the signed message, however small, causes verification to fail
Non-repudiation	Because producing a valid signature requires the sender's private key, the sender cannot later credibly deny having signed the message

Note what a digital signature does *not* provide on its own: **confidentiality**. Signing a message does not hide its content: the message in the example above travels in readable form alongside its signature. A system that requires both secrecy and signing (as most secure communication does) applies encryption and signing as two separate steps, often in combination with the hybrid approach described in Section 5.3.4.

NOTE — Digital signature versus digital certificate

A digital signature proves that a specific private key was used to sign a specific message. It does not, by itself, prove *whose* private key that is. That binding (associating a public key with a verified real-world identity) is the job of a **digital certificate**, issued by a trusted third party called a **Certificate Authority (CA)**. Certificates are what allow a browser to trust that a website's public key genuinely belongs to that website, and are the mechanism underlying HTTPS.

5.4.4 Applications of Digital Signatures

Table 5.8 — Applications of digital signatures

Application	How digital signatures are used
Software distribution	Publishers sign software releases so users can verify the software has not been tampered with since release
Financial transactions	Signed instructions authenticate the originator of a fund transfer and provide non-repudiation for disputes
Legal and government documents	E-signature laws in many countries (including India's Information Technology Act, 2000) give digitally signed documents legal standing equivalent to handwritten signatures
Secure email (S/MIME, PGP)	Signed emails let recipients verify the sender's identity and detect tampering in transit
Blockchain and cryptocurrency	Every transaction is digitally signed by the sender's private key to authorise the transfer of funds

CHECK YOUR PROGRESS 5.4

1. Name the two cryptographic building blocks combined to create a digital signature.
2. List, in order, the three steps a sender performs to sign a message.
3. Explain why a digital signature does not, by itself, provide confidentiality.
4. What problem does a digital certificate solve that a digital signature alone does not?

5.5 FIREWALLS

5.5.1 What is a Firewall?

Cryptography (Section 5.3) and digital signatures (Section 5.4) protect the content of specific messages. A **firewall** works differently: it protects an entire network by controlling *which* traffic is allowed to cross its boundary in the first place, regardless of that traffic's content.

IMPORTANT — Definition — Firewall

A **firewall** is a network security system, implemented in hardware, software, or both, that monitors and controls incoming and outgoing network traffic based on a defined set of security rules, and sits at the boundary between a trusted internal network and an untrusted external network such as the internet.

A firewall is needed because a network connected to the internet is, by default, reachable from anywhere in the world; without a control point at the boundary, every device on the internal network would be directly exposed to every scanning, probing, and attacking system on the internet. A firewall reduces this exposure to only the traffic explicitly permitted by policy.

5.5.2 How Firewalls Work

Firewalls have grown more sophisticated over time; a modern firewall typically combines several of the following techniques.

- **Packet filtering.** The firewall examines each packet's header (source and destination IP address, port number, and protocol) and compares it against a set of rules, called an **Access Control List (ACL)**, that specify which combinations are permitted or denied. Packet filtering is fast but stateless: it evaluates each packet independently, without knowledge of the broader conversation it belongs to.
- **Stateful inspection.** A stateful firewall tracks the state of each active connection (for example, that a particular outgoing request has already been sent, and that a reply is therefore expected). This allows it to permit legitimate reply traffic automatically while still blocking unsolicited traffic that does not correspond to any connection the internal network initiated, a significant improvement over stateless packet filtering.
- **Application-layer filtering.** Rather than examining only headers, an application-layer (or "next-generation") firewall inspects the actual content of traffic for a specific application protocol, for example inspecting HTTP traffic for known web-attack patterns, or blocking specific file types in email attachments. This allows far more granular policy than IP-address- or port-based rules alone.

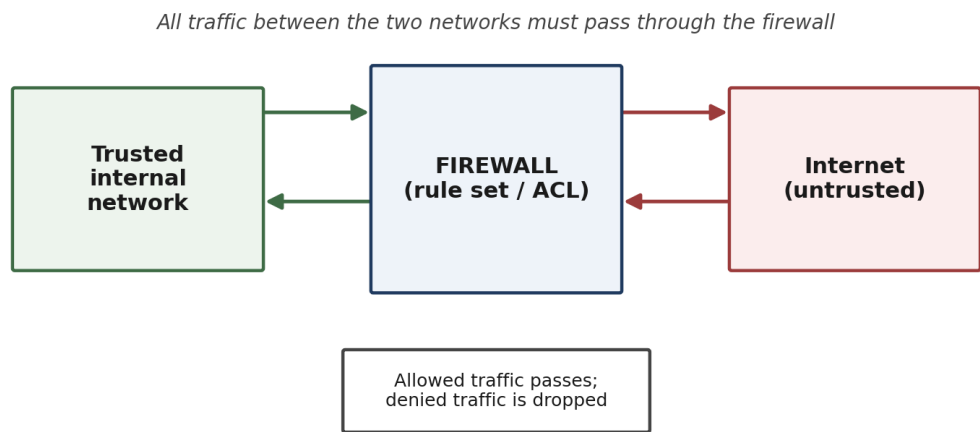


Figure 5.4 — A firewall between a trusted network and the internet

EXAMPLE — A simple ACL rule in plain language

"Permit any traffic from the internal network (192.168.1.0/24) to any external destination on port 443 (HTTPS)." "Deny all traffic from any external source to the internal network on port 3389 (Remote Desktop) unless it originates from the head office's fixed IP address." A firewall evaluates such rules, typically in order, against every packet crossing the boundary, applying the first matching rule.

5.5.3 What Firewalls Do and Do Not Protect Against

Table 5.9 — Scope of firewall protection

Firewalls are effective against	Firewalls are not effective against
Unsolicited inbound connection attempts from the internet	Malware already inside the network (for example, carried in on a USB drive or an infected laptop)
Traffic to or from known-malicious IP addresses or ports	Attacks that use permitted traffic in a malicious way, unless application-layer filtering is also configured
Enforcing an organisation's traffic policy at a single, auditable boundary	A user being socially engineered into voluntarily disclosing credentials (Section 5.7)

This is why a firewall is described as *one layer* of a broader security architecture, often summarised as **defence in depth**, rather than a complete solution on its own. Cryptography, authentication, firewalls, and user awareness each address a different category of threat, and a realistic security design deploys all of them together.

CHECK YOUR PROGRESS 5.5

1. Define a firewall and explain the boundary at which it typically sits.
2. Distinguish stateless packet filtering from stateful inspection.
3. Give one example of a threat a firewall cannot defend against, and explain why.
4. What is meant by "defence in depth," and how does a firewall fit into that idea?

5.6 USER IDENTIFICATION AND AUTHENTICATION

5.6.1 Identification versus Authentication

Two related but distinct steps determine who is allowed to use a system. **Identification** is the act of a user claiming an identity, typically by presenting a username, user ID, or email address. **Authentication** is the act of verifying that the claimed identity is genuine, before granting access. A username by itself proves nothing: anyone can type someone else's username. Authentication is what confirms the person typing it is who they claim to be.

NOTE — Authentication versus authorisation

A third, frequently confused term is **authorisation**: once a user's identity is authenticated, authorisation determines *what* that specific, verified user is permitted to do: which files they can read, which actions they can perform. Authentication answers "who are you?"; authorisation answers "what are you allowed to do?" A system can authenticate a user correctly and still deny them access to a specific resource, if their authorised permissions do not extend to it.

5.6.2 Authentication Factors

Every authentication mechanism relies on one or more of three categories of evidence, called **factors**.

Table 5.10 — The three authentication factors

Factor	Description	Examples
Knowledge factor ("something you know")	Information only the legitimate user should know	Password, PIN, passphrase, answer to a security question
Possession factor ("something you have")	A physical or digital item only the legitimate user should possess	Smart card, hardware security token, one-time password (OTP) sent to a registered phone
Inherence factor ("something you are")	A biological trait unique to the legitimate user	Fingerprint, iris scan, facial recognition, voice pattern

Combining factors from *different* categories produces stronger authentication than using multiple items from the *same* category, because an attacker who compromises one category (for example, guessing a password) still lacks the others.

- **Single-Factor Authentication (SFA)** relies on only one factor, most commonly a password. It is the weakest form, since compromising the single factor (through guessing, phishing, or a data breach elsewhere) is sufficient to defeat it entirely.
- **Two-Factor Authentication (2FA)** requires two factors from different categories, for example a password (knowledge) plus an OTP sent to a registered device (possession). Even if a password is stolen, the attacker still lacks the second factor.
- **Multi-Factor Authentication (MFA)** requires two or more factors, and is used generically to describe any scheme stronger than a single factor; 2FA is the most common special case of MFA.

EXAMPLE — Why 2FA stops most credential-theft attacks

A phishing email tricks a user into typing their password into a fake login page. Under single-factor authentication, the attacker now has full access. Under two-factor authentication with an OTP sent to the user's registered phone, the attacker still cannot log in without also possessing that phone. The stolen password alone is insufficient. This is why banks, email providers, and most sensitive services now require or strongly encourage 2FA.

5.6.3 Cryptographic Techniques in Authentication

Cryptography (Section 5.3) is used extensively to strengthen authentication systems, in three main ways:

- **Password hashing.** Systems should never store passwords in plaintext. Instead, a cryptographic hash of the password (Section 5.3.3) is stored; at login, the entered password is hashed and compared against the stored hash. If the password database is ever stolen, the attacker obtains only hash values, not the original passwords. Weak or common passwords, though, remain vulnerable to precomputed "dictionary" attacks, which is why passwords are typically combined with a random per-user value called a **salt** before hashing.
- **Encryption in transit.** Login credentials and session tokens are encrypted (typically via the hybrid approach described in Section 5.3.4) while travelling over the network, preventing interception.
- **Digital signatures in authentication protocols.** Some authentication schemes use digital signatures (Section 5.4) to let a server verify that an authentication request genuinely originated from a specific, previously registered device or key, without the corresponding secret ever being transmitted.

CHECK YOUR PROGRESS 5.6

1. Distinguish identification from authentication, and give an everyday example of each.
2. Distinguish authentication from authorisation.
3. Name the three authentication factors and give one example of each.
4. Explain why storing a *salted hash* of a password is safer than storing the password itself, or even an unsalted hash of it.

5.7 SECURITY AWARENESS AND POLICIES

5.7.1 Why the Human Element Matters

Sections 1.3 to 1.6 covered technical controls: cryptography, digital signatures, firewalls, and authentication. Even the strongest technical controls, however, can be defeated by a single untrained or careless user: a password written on a sticky note, a phishing email trusted and clicked, a USB drive of unknown origin plugged into a secure workstation. Industry incident reports consistently identify human error and social engineering, rather than a failure of cryptography or firewalls, as the most common route by which real breaches occur. This is why **security awareness** and **security policy** are treated as core components of information security, not optional extras.

IMPORTANT — Definition — Security Awareness Programme

A **security awareness programme** is an organised effort to educate an organisation's people about security threats, safe practices, and their own responsibilities, so that they act as an effective line of defence rather than the weakest link.

5.7.2 What Security Awareness Programmes Cover

Table 5.11 — Common security awareness topics

Topic	Why it matters
Recognising phishing and social engineering	Most malware infections and credential thefts (Section 5.2, 1.6) begin with a deceived human, not a broken technical control
Password hygiene	Encourages strong, unique passwords and the use of password managers and multi-factor authentication (Section 5.6.2)

Topic	Why it matters
Safe handling of removable media and unknown files	Prevents malware introduced physically, bypassing network-level defences such as firewalls
Reporting procedures	Ensures that a suspected incident reaches the security team quickly, before it can spread further
Data handling and classification	Teaches staff which data is sensitive and how it must be stored, shared, and disposed of

5.7.3 Security Policies

A **security policy** is a formal, written statement of an organisation's rules, expectations, and procedures regarding information security. Where awareness programmes change individual behaviour through education, policies establish the organisation-wide rules that behaviour is measured against, and give the organisation a basis for enforcement and audit.

Security policies typically address:

- **Acceptable use:** what employees may and may not do with organisational systems and data.
- **Access control:** who may access which systems and data, and under what conditions (linking directly to the authorisation concept in Section 5.6.1).
- **Key and password management:** acceptable cryptographic algorithms, minimum password strength, and rotation requirements.
- **Incident response:** the steps to follow when a security incident is suspected or confirmed, including who must be notified and within what time frame.
- **Regulatory compliance:** how the organisation meets legal obligations such as data-protection law.

EXAMPLE — Awareness and policy working together

An organisation's policy states that all removable media must be scanned before use. An awareness programme ensures employees actually know this rule exists, understand *why* it exists (to catch malware that bypasses network-level defences), and know how to have a USB drive scanned. Policy without awareness is unenforced; awareness without policy has no organisational backing. Neither is sufficient alone.

5.7.4 Why This Matters for Risk, Compliance, and Incident Response

Referring back to Section 5.1.3's risk framework, security awareness and policy reduce risk on the *human vulnerability* side of the equation, complementing the technical controls of Sections 1.3–1.6. They also serve two further purposes: **regulatory compliance**, since many industries are legally required to demonstrate documented security policies and staff training (for example, organisations handling payment card data or health records); and **incident response**, since staff who know the reporting procedure in advance can limit the damage of an incident that does occur, rather than allowing it to spread undetected.

CHECK YOUR PROGRESS 5.7

1. Why is the human element frequently described as the "weakest link" in information security, despite strong technical controls being in place?
2. Distinguish the purpose of a security awareness programme from that of a security policy.
3. Name three topics a typical security policy addresses.
4. Explain, with an example, why a policy without a supporting awareness programme is likely to be ineffective.

5.8 APPLICATION AREAS REQUIRING INFORMATION SECURITY

Every domain examined in this table depends on the same underlying controls covered in Sections 1.2–1.7 (cryptography, authentication, firewalls, and policy), applied to that domain's specific risks and regulatory obligations.

Table 5.12 — Domains where information security is critical

Domain	Sensitive information at stake	Primary security concern
Financial services	Account details, transaction records, credentials	Fraud prevention, integrity of transaction records, regulatory compliance
Healthcare	Medical records, diagnoses, treatment history	Patient confidentiality, integrity of records used for treatment decisions, regulatory compliance (e.g., health-data protection law)
E-commerce	Payment details, purchase history, shipping addresses	Protecting payment data in transit and storage; preventing account takeover and payment fraud
Government	National security data, citizen records, tax information	Protection against espionage, sabotage, and large-scale data breaches affecting citizens
Education	Student records, examination data, research data	Preventing academic fraud, protecting student privacy, safeguarding research intellectual property
Telecommunications	Voice, data, and message traffic for millions of users	Preventing interception, ensuring service availability, lawful-access compliance
Critical infrastructure (energy, water, transport)	Industrial control systems	Preventing physical disruption through cyberattack (Section 5.2.3's Stuxnet example)
Legal services	Client files, case records, privileged communications	Maintaining client confidentiality and legal privilege

THINK ABOUT IT — Think About It

For each domain in Table 5.12, identify which one control from Sections 1.3–1.6 (cryptography, digital signatures, firewalls, or authentication) most directly addresses its stated primary concern. Some domains will draw on more than one; identify the strongest single match first.

5.9 INTRODUCTION TO MOBILE COMMERCE

5.9.1 What is Mobile Commerce?

Part A examined how information is kept secure. Part B examines one of the largest practical applications of that security: buying and selling over mobile devices. **Mobile commerce**, almost universally abbreviated **m-commerce**, is the conduct of commercial transactions (browsing, ordering, paying, and receiving services) using wireless handheld devices such as smartphones and tablets, rather than a desktop computer connected by a fixed line.

IMPORTANT — Definition — Mobile Commerce

Mobile commerce (m-commerce) is the buying and selling of goods, services, and information through wireless handheld devices, using mobile networks and mobile-optimised applications or websites, without requiring a fixed physical connection to complete the transaction.

5.9.2 M-Commerce as a Subset of E-Commerce

M-commerce is best understood as a **specialised subset of e-commerce** (electronic commerce, the broader category of commercial transactions conducted over any digital network), distinguished by three properties that a desktop-based transaction does not share:

- **Ubiquity.** The transaction can be initiated from almost any location, not only from a fixed workstation.
- **Context-awareness.** A mobile device can determine its location and surroundings (Section 5.12.4), letting a transaction be tailored to where the user actually is at that moment.
- **Immediacy.** A mobile device is typically on the user's person and connected continuously, allowing a purchase decision to be acted on the instant it occurs, rather than waiting until the user is next at a desktop.

These three properties explain most of m-commerce's distinctive advantages and its distinctive security requirements (Section 5.6's authentication factors are applied differently on a small touchscreen than on a desktop, and this is one reason biometric and OTP-based authentication is so closely associated with mobile transactions).

5.9.3 Key Features of M-Commerce Platforms

Table 5.13 — Key features of m-commerce platforms

Feature	Purpose
Mobile-optimised websites and dedicated apps	Deliver a usable interface on a small touchscreen
Secure mobile payment gateways	Allow a transaction to be completed and paid for entirely on the device (Section 5.12.3)
Location-based services	Personalise offers and information to the user's current location (Section 5.12.4)
Social media integration	Let users discover and share products through platforms they already use

For businesses, m-commerce presents an opportunity to reach customers directly, on a device the customer carries at all times, and to personalise the shopping experience far more precisely than a physical storefront or even a desktop website can. For consumers, it offers the ability to browse, compare, and purchase from anywhere, at any time, with minimal friction. Sections 2.2 to 2.6 examine, in turn, how this growth happened, which technologies enable it, how it has developed specifically in India, and how digital marketing drives traffic to m-commerce platforms.

CHECK YOUR PROGRESS 5.9

1. Define mobile commerce and state how it differs from e-commerce in general.
2. Name and briefly explain the three properties (ubiquity, context-awareness, immediacy) that distinguish m-commerce from a desktop-based transaction.
3. Give one example of how a mobile device's location can be used to personalise a shopping experience.

5.10 GROWTH OF MOBILE COMMERCE

5.10.1 Drivers of Growth

M-commerce's growth over the past decade has not been the result of any single factor, but of several reinforcing trends occurring together. Table 5.14 organises these drivers by the layer of the system each one affects: the device, the network and payment infrastructure, and the business practices built on top of them.

Table 5.14 — Drivers of m-commerce growth, by layer

Layer	Driver	Effect
Device	Widespread smartphone ownership	Puts a capable shopping platform in most consumers' pockets at all times
Device	Improved mobile hardware (faster processors, better displays)	Makes browsing and transacting on a small screen as smooth as on a desktop
Network and payment	Faster, more reliable mobile internet (4G, 5G)	Removes the speed and reliability gap that once favoured desktop shopping
Network and payment	Secure mobile payment systems (Section 5.12.3)	Builds consumer confidence to enter payment details on a mobile device
Business practice	Mobile-optimised websites and dedicated apps	Removes the usability friction of using a desktop-oriented site on a small screen
Business practice	Personalisation and targeted marketing	Uses data gathered through the app or site to increase relevance and conversion (Section 5.14)
Business practice	Integration with social media	Lets discovery and purchase happen inside platforms users already spend time in

5.10.2 Newer Growth Patterns

Beyond the general drivers in Table 5.14, two more specific patterns have shaped m-commerce's more recent growth:

- **Mobile-only marketplaces.** Some business models exist *only* as mobile apps, with no meaningful desktop equivalent. Ride-hailing and food-delivery platforms are the clearest examples (and are covered as case studies in Section 5.11). These platforms succeed because their core value proposition (summoning a nearby physical service in real time) depends on the location-awareness and immediacy properties described in Section 5.9.2, which a desktop application cannot easily replicate.
- **Social commerce.** The line between social media and shopping has blurred, with shoppable posts, in-app checkout, and live-stream selling letting a purchase happen without the user ever leaving the social app.

THINK ABOUT IT — Think About It

Of the drivers in Table 5.14, which do you think depends most directly on the information-security techniques from Part A, and which specific technique (cryptography, digital signatures, or authentication) is most relevant to it? Consider the "secure mobile payment systems" row before reading Section 5.12.3.

CHECK YOUR PROGRESS 5.10

1. Name one growth driver from each of the three layers in Table 5.14.
2. What distinguishes a "mobile-only" marketplace from a business that simply also offers a mobile app?
3. Explain what "social commerce" means, with an example.

5.11 SUCCESS STORIES OF MOBILE COMMERCE

Studying real platforms illustrates how the abstract drivers of Section 5.10 combine in practice. The five Indian platforms in Table 5.15 each built their success on a different application of the same underlying features (Table 5.13), and are useful reference examples for examination questions asking for real-world illustration of m-commerce concepts.

Table 5.15 — Selected Indian m-commerce success stories

Platform	Sector	Core m-commerce strength
Flipkart	General e-commerce	Personalised product recommendations at scale, multilingual support, and accessibility features (including text-to-speech) that widen its reach across India's linguistically diverse user base
Myntra	Fashion retail	Mobile-first design with personalised style recommendations, fast product search, and simplified returns, reducing the friction points specific to buying clothing without physically trying it on
Ola	Ride-hailing	Real-time location awareness (Section 5.9.2) combined with in-app payment and live journey tracking, plus accessibility features such as voice commands and screen-reader support
Zomato	Food delivery and restaurant discovery	Combines location-based restaurant discovery with integrated ordering and payment in a single app, reducing a multi-step process (searching, calling, paying) to a few taps
Paytm	Digital payments and services	A single app spanning payments, recharges, bill payments, and shopping, illustrating how a payment platform can expand into a broader m-commerce ecosystem once user trust in its security is established

EXAMPLE — What these platforms have in common

Despite operating in entirely different sectors, all five platforms in Table 5.15 depend on the same foundation: secure authentication (Section 5.6) to protect user accounts, secure payment processing (built on the cryptographic techniques of Section 5.3) to protect financial transactions, and location-based services (Section 5.12.4) to make the experience relevant to where the user actually is. A platform that got any one of these three foundations wrong (insecure payments, weak authentication, or a location feature that felt invasive rather than useful) would struggle to retain users regardless of how good its core product was.

CHECK YOUR PROGRESS 5.11

1. Choose two platforms from Table 5.15 in different sectors and identify one m-commerce feature (from Table 5.13) each relies on most heavily.
2. Explain why "accessibility" (multilingual support, screen-reader compatibility) is a meaningful growth strategy for an m-commerce platform, rather than merely a compliance requirement.

5.12 TECHNOLOGIES FOR MOBILE COMMERCE

M-commerce, as defined in Section 5.9, depends on a stack of underlying technologies. This section examines the six categories that matter most, moving from the interface the user directly sees (2.4.1–2.4.2) to the systems working behind it (2.4.3–2.4.6).

5.12.1 Mobile Apps and Mobile Websites

A business reaches mobile users through two, often complementary, channels.

- **Mobile apps** are dedicated software installed on the device, built either as **native apps** (developed separately for iOS and Android, using each platform's own tools, for the best possible performance and access to device features) or as **cross-platform apps** (built once using a framework such as React Native or Flutter, and deployed to both platforms, trading some native performance for faster, cheaper development).

- **Mobile websites** are accessed through the device's browser rather than installed. **Responsive web design** ensures the same website adapts its layout to any screen size, giving users a consistent experience whether they arrive from a phone, tablet, or desktop, without the business needing to build and maintain a separate app.

Apps generally provide a faster, more capable experience and better access to device hardware (camera, GPS, biometric sensors); mobile websites require no installation and are immediately accessible to any user who follows a link. Most substantial m-commerce businesses maintain both.

5.12.2 Progressive Web Apps

A **Progressive Web App (PWA)** is a mobile website built with modern web technologies (such as **Service Workers**, which allow a web page to run background tasks and cache content, and a **Web App Manifest**, which lets a website be "installed" to a home screen) so that it behaves much like a native app (working offline, sending push notifications, and launching from a home-screen icon) while remaining, technically, a website rather than a separately compiled app. PWAs are attractive to businesses that want app-like engagement without the cost of building and maintaining separate native apps for each platform.

5.12.3 Mobile Payment Solutions

Secure, convenient payment is the single technology most directly responsible for converting browsing into a completed transaction, which is why it draws so heavily on Part A's cryptographic foundations.

Table 5.16 — Mobile payment technologies

Technology	How it works	Security techniques used
Digital wallets (e.g., Google Pay, Apple Pay, Paytm)	Store payment credentials on the device and transmit a one-time token, rather than the actual card number, to complete a purchase	Tokenisation, encryption in transit (Section 5.3), biometric authentication (Section 5.6)
Near Field Communication (NFC)	Allows a tap-to-pay transaction between the device and a point-of-sale terminal over a very short range	Encrypted, single-use transaction data exchanged at close range, limiting interception risk
Unified Payments Interface (UPI), used widely in India	Enables instant bank-to-bank transfer using a virtual payment address rather than sharing account or card details directly	Device binding, PIN authentication, and encryption of transaction data
Peer-to-peer payment apps	Allow direct transfer of funds between individual users' linked bank accounts or wallets	Authentication and encrypted transaction processing, as above

5.12.4 Location-Based Services

Location-Based Services (LBS) use a device's determined geographic position, via GPS, Wi-Fi positioning, or short-range **beacon** technology in physical stores, to make an m-commerce experience relevant to where the user actually is. This underlies use cases such as showing nearby store availability, sending a location-triggered promotion as a customer walks past a shop, or (as in Section 5.11's ride-hailing and food-delivery examples) matching a user with the nearest available service in real time.

5.12.5 Augmented Reality, Virtual Reality, AI, and Machine Learning

- **Augmented Reality (AR)** overlays digital content onto a live camera view of the real world, letting a shopper visualise a product (furniture in their own room, a shade of lipstick on their own face) before purchasing, which reduces the uncertainty that leads to returns.
- **Virtual Reality (VR)** goes further, replacing the user's view entirely with a simulated shopping environment they can navigate as if physically present.

- **Artificial Intelligence (AI) and Machine Learning (ML)** power personalised product recommendations, demand-based pricing, and AI-driven chatbots that handle routine customer queries without human staff, drawing on the same underlying techniques as any other machine-learning application (recommendation and classification, as introduced in a general AI context elsewhere in this programme).

5.12.6 How These Technologies Combine

EXAMPLE — A single purchase, several technologies at once

A user opens a retailer's native app (Section 5.12.1), which uses their location (Section 5.12.4) to show nearby store stock. An AR feature (Section 5.12.5) lets them preview a piece of furniture in their living room through the camera. They complete the purchase using a digital wallet secured by tokenisation and biometric authentication (Section 5.12.3, drawing on Section 5.6). A recommendation engine (Section 5.12.5) then suggests a matching product for their next visit. No single technology made this experience possible; it is the combination that defines modern m-commerce.

CHECK YOUR PROGRESS 5.12

1. Distinguish a native app from a cross-platform app, and state one advantage of each.
2. What is a Progressive Web App, and what two web technologies does it rely on?
3. Explain how tokenisation improves the security of a mobile wallet transaction compared with transmitting the actual card number.
4. Give one m-commerce use case each for AR and for location-based services.

5.13 M-COMMERCE IN INDIA

5.13.1 Why India is a Distinctive M-Commerce Market

India is one of the world's largest and fastest-growing m-commerce markets, but its growth path has differed from that of markets where e-commerce first took hold on desktop computers and only later moved to mobile. In India, a large share of first-time internet users came online directly on a smartphone, skipping the desktop era almost entirely, a pattern often called "**mobile-first**" or, for the newest wave of users, "**mobile-only**." This matters for understanding why India's m-commerce technologies (Section 5.12) place unusually heavy emphasis on low-bandwidth design, regional-language support, and mobile-native payment systems such as UPI, rather than adapting desktop-era approaches for a smaller screen.

DID YOU KNOW? — The scale of India's digital payment infrastructure

India's Unified Payments Interface (UPI), the instant, bank-to-bank mobile payment system introduced in Table 5.16, has grown into one of the largest real-time payment systems in the world. According to National Payments Corporation of India (NPCI) data, UPI processed over 19 billion transactions in July 2025 alone, worth roughly ₹25 lakh crore, and continued growing through into 2026. India's internet user base crossed 100 crore (1 billion) subscribers by mid-2025, with wireless tele-density (mobile connections per 100 people) above 80%, according to the Telecom Regulatory Authority of India (TRAI). Treat such figures as *indicative of scale and trend* rather than fixed values, since digital-adoption statistics change from one reporting period to the next.

5.13.2 Benefits Driving Adoption in India

Table 5.17 — Benefits of m-commerce adoption in India

Benefit	Particular relevance to India
Convenience	Removes the need to travel to a physical store, valuable given the distances involved outside dense urban centres
Accessibility	Extends online shopping to rural and semi-urban populations that traditional retail infrastructure has historically underserved
Personalisation	AI-driven recommendations and regional-language interfaces (Section 5.12.5) adapt the experience to a highly diverse user base
Cost savings	Mobile apps are a cheaper customer-acquisition channel for businesses than physical retail, and often pass on savings through discounts
Time efficiency	One-click purchasing and saved payment details (Section 5.12.3) reduce transaction time on a market where price and time sensitivity are both high

5.13.3 Distinctive Challenges

Alongside its scale, India's m-commerce market faces challenges specific to its context: uneven internet quality outside major cities, a highly price-sensitive customer base that makes discount-driven strategies common, the need to support many languages and varying levels of digital literacy, and, returning to Part A's themes, the need to build user trust in digital payment security among a population where large numbers of users are transacting digitally for the first time.

CHECK YOUR PROGRESS 5.13

1. Explain what "mobile-first" (or "mobile-only") internet adoption means, and why it applies particularly to India.
2. Name two figures that illustrate the scale of UPI adoption in India, and state their approximate order of magnitude.
3. Identify one challenge specific to the Indian m-commerce market that would matter less in a market where desktop e-commerce was established first.

5.14 DIGITAL MARKETING

5.14.1 What is Digital Marketing?

Building an m-commerce platform (Sections 2.1–2.4) solves only half of the business problem; the platform still needs customers to find it. **Digital marketing** is the set of strategies and channels used to promote products, services, or brands through digital media (search engines, social media, email, and mobile apps) in order to reach audiences, drive traffic, and convert that traffic into sales.

IMPORTANT — Definition — Digital Marketing

Digital marketing is the practice of promoting products, services, or brands using digital channels and technologies, with the aim of reaching target audiences, driving measurable engagement, and generating leads or sales.

Digital marketing is distinguished from traditional marketing (television, print, billboards) chiefly by **measurability**: nearly every digital channel in Table 5.18 provides detailed data on how many people saw a message, how many acted on it, and at what cost, allowing strategies to be tested and refined continuously in a way traditional media rarely allows.

5.14.2 Core Digital Marketing Channels

Table 5.18 — Core digital marketing channels

Channel	What it does
Search Engine Optimisation (SEO)	Improves a website's organic (unpaid) ranking on search-engine results pages by optimising content, structure, and technical factors for relevant keywords
Search Engine Marketing (SEM) / Pay-Per-Click (PPC)	Places paid advertisements on search engines, with the advertiser paying per click or impression, for immediate (rather than gradually built) visibility
Social Media Marketing (SMM)	Builds brand awareness and engagement through content, advertising, and interaction on platforms such as Instagram, Facebook, and LinkedIn
Content marketing	Attracts and retains an audience by publishing genuinely useful content (articles, videos, guides) rather than direct advertising, building trust over time
Email marketing	Sends targeted, often personalised messages to an existing list of subscribers to nurture leads and retain customers
Influencer marketing	Uses individuals with an established, trusted audience in a specific niche to promote a product to that audience
Affiliate marketing	Pays independent publishers a commission for traffic or sales they generate through referral links, aligning the affiliate's incentive with the business's
Mobile marketing	Reaches users specifically on mobile devices, through mobile-optimised ads, SMS, push notifications, and location-based targeting; the channel most directly tied to Part B's subject matter

5.14.3 Analytics: The Feedback Loop

Every channel in Table 5.18 generates data (impressions, clicks, conversions, cost per acquisition), and **analytics** is the practice of collecting and interpreting that data to judge which channels and campaigns are actually working. This feedback loop is what makes digital marketing measurable in a way traditional marketing is not: a campaign can be adjusted mid-flight based on real performance data, rather than only being evaluated after the fact.

EXAMPLE — Choosing a channel for an m-commerce launch

A new food-delivery app (compare Section 5.11's Zomato example) launching in a single city might prioritise **mobile marketing** with location-based targeting (reaching users physically near partner restaurants), **influencer marketing** (local food influencers building initial trust), and **SEM** (capturing users actively searching "food delivery near me"). A more established, nationwide platform might instead invest more heavily in **SEO** and **content marketing**, which compound in value over a longer time horizon. There is no single correct channel mix; the right combination depends on the stage of the business and the audience it is trying to reach.

CHECK YOUR PROGRESS 5.14

1. Define digital marketing and explain the key property that distinguishes it from traditional marketing.
2. Distinguish SEO from SEM/PPC in terms of how visibility is achieved and paid for.
3. Why is mobile marketing described as the channel most directly tied to m-commerce?
4. Explain why analytics is described as a "feedback loop" rather than simply a reporting tool.

5.15 HISTORY OF CENTRALIZED AND DISTRIBUTED COMPUTING

5.15.1 Centralized Systems

Computing's earliest organising model is the simplest to picture, and provides the baseline against which every later model in this Part is defined by contrast.

IMPORTANT — Definition — Centralized System

A **centralized system** is a computing architecture in which processing and data are concentrated in a single central system, typically one powerful computer or a small, tightly coupled cluster of processors, which one or more client devices access, usually by sending a request and receiving a response.

The definition box above is intentionally terse; its implications are worth spelling out.

Table 5.19 — Characteristics of centralized systems

Characteristic	What it means in practice
Single point of control	One central node coordinates the entire system; clients only ever interact through it
Global clock	Because there is one central node, every operation can be given a single, unambiguous time ordering, a property that, as Section 5.16 will show, distributed systems must work much harder to achieve
Dependent failure	If the central node fails, the entire system fails; there is no other node able to take over its coordinating role
Simplicity	Centralized systems are comparatively easy to design, reason about, and secure, precisely because there is only one place where control and data reside

A single-user desktop computer and a classic mainframe serving many remote terminals are both centralized systems by this definition, differing only in scale: the mainframe has more processors, memory, and storage, and serves many simultaneous users, but both funnel all processing through one coordinating centre.

5.15.2 The Shift Toward Distribution

Centralized systems' greatest strength, one authoritative centre, is also their fundamental limitation: that centre is a single point of failure, and its capacity places a hard ceiling on how far the system can scale. **Distributed computing** emerged specifically to remove this ceiling, and its history is best understood as a series of responses to the limitations of whatever model came immediately before it. Table 5.20 traces this evolution as a timeline; Section 5.16 then examines the resulting distributed-computing model in its own right.

Table 5.20 — From centralization to decentralization: a timeline

Era (approximate)	Development	What problem it solved
1960s	Mainframes with time-sharing, accessed by many simple terminals	Let many users share one expensive computer, at the cost of remaining fully centralized and very expensive
Early 1970s	Cluster computing (groups of similar workstations connected by a fast local network, Section 5.17); early packet-switching and ARPANET	Provided a cheaper alternative to a single mainframe, and enabled message exchange between physically separate machines
1970s–1980s	TCP/IP as a common protocol; growth of the internet and personal computers; the Domain Name System (DNS) replacing unscalable manual host lists	Let independently owned networks interconnect as a single "network of networks," and let that network scale far beyond what a centrally maintained naming scheme could support

Era (approximate)	Development	What problem it solved
1980s–1990s	The World Wide Web (HTTP and HTML, developed by Tim Berners-Lee); remote procedure calls (RPC) over TCP/IP	Gave the internet a standard way to publish and link content, and gave separately running programs a standard way to communicate with one another
1990s–2000s	Peer-to-peer (P2P) networking; Grid computing (Section 5.18); early web services using XML-based, platform-independent messaging	Removed the need for a central coordinator entirely (P2P); let many independent computers jointly tackle one large task (grid); let programs written in different languages, on different platforms, communicate uniformly (web services)
2000s	Practical x86 virtualization	Let one physical machine run multiple independent "virtual" machines, a prerequisite for the cloud computing model in Section 5.19
2000s–2010s	Cloud computing , converging cluster technology, virtualization, and middleware; growth of mobile computing and the Internet of Things (IoT)	Let computing resources be consumed as an on-demand, pay-per-use service over the internet, and extended computing to phones, sensors, and everyday devices
2010s	Fog and edge computing	Moved some processing back closer to where data is generated, reducing the latency and bandwidth cost of sending everything to a distant, centralized cloud data centre
2010s–2020s	Containers (e.g., Docker) and orchestration platforms (e.g., Kubernetes)	Packaged an application with its dependencies so it runs consistently anywhere, without the overhead of a full separate operating system per application, as a virtual machine requires

NOTE — Reading the timeline correctly

This timeline should not be read as each development *replacing* the one before it. Mainframes, clusters, grids, and clouds are all still in active use today for the workloads each suits best. It should instead be read as each development *adding a new option* to what was available before, in response to a specific limitation that the existing options could not address. Cloud computing (Section 5.19) is best understood as the point at which several of these threads (clusters, virtualization, and internet-scale networking) converged into a single, commercially accessible service.

CHECK YOUR PROGRESS 5.15

1. Define a centralized system and name its single greatest structural weakness.
2. Explain why the growth of the internet made a centralized naming system such as HOSTS.TXT unworkable, and what replaced it.
3. Name the three technological threads that converged to produce cloud computing, according to Table 5.20.
4. Explain, in one sentence, what problem fog and edge computing solve that a purely centralized cloud model does not.

5.16 OVERVIEW OF DISTRIBUTED COMPUTING

5.16.1 What is a Distributed System?

Section 5.15 traced *why* computing moved away from pure centralization. This section defines the resulting model precisely.

IMPORTANT — Definition — Distributed System

A **distributed system** is a collection of independent computers, connected by a network and communicating by passing messages, that appears to its users as a single, coherent system, working together to achieve a common goal without sharing main memory or a single central clock.

The phrase "appears to its users as a single system" is doing important work in that definition: a user of a well-designed distributed system (checking email, using a search engine, streaming a video) is typically unaware that their request was handled by any particular one of possibly thousands of physically separate machines. This property, hiding the fact of distribution from the end user, is called **transparency**, and is one of the central design goals of distributed systems engineering.

5.16.2 Characteristics of Distributed Systems

Table 5.21 — Characteristics of distributed systems

Characteristic	What it means
Resource sharing	Nodes share hardware, software, or data over the network, but not main memory or disks directly
Concurrency	Multiple nodes can process different (or the same) tasks simultaneously
Scalability	Capacity grows by adding more nodes to the network, rather than by upgrading one central machine
Fault tolerance	The failure of one node need not bring down the whole system, since responsibility is spread across many nodes
Transparency	A user or program interacting with the system need not know which specific node is handling a given request
Absence of a global clock	Unlike the centralized system of Section 5.15.1, there is no single clock every node is synchronised to, which makes ordering events across the system a genuinely hard problem

5.16.3 Types of Distributed Databases

Distributed systems that manage shared data are further classified by how uniform their software is across sites:

- **Homogeneous distributed databases** run the same database software and schema at every site, with data merely partitioned across sites. Their design goal is to present users with what looks like a single, unified database, while hiding the details of how data is actually distributed.
- **Heterogeneous distributed databases** run different software, and potentially different schemas, at different sites, often because the sites' databases already existed independently before being integrated. Their design goal shifts from hiding distribution to making genuinely different systems interoperate usefully.

5.16.4 Components and Architectures

Every distributed system, regardless of scale, is built from two kinds of component: **nodes** (individual computers, mobile devices, or other computational units) and **communication links** (the network connecting them, cabled, wireless, or a mix). How nodes relate to one another defines the system's architecture:

- **Peer-to-peer**, in which all nodes have equal standing and cooperate toward a shared goal without a designated coordinator (Section 5.15's P2P entry, and Section 5.18's grid computing, both build on this idea).

- **Client-server**, in which some nodes act as servers, providing a specific coordinating or resource-providing role, while others act as clients that consume that role.
- **N-tier**, in which an application's logic is split across several distinct layers or nodes (for example, a presentation layer, an application-logic layer, and a data layer), each of which may itself be distributed, working together to serve the end user.

5.16.5 Advantages, Limitations, and Disadvantages

Table 5.22 — Advantages and limitations of distributed systems

Advantages	Limitations and disadvantages
Scalability: capacity grows by adding nodes	Design complexity: algorithms must account for the absence of a global clock and varying network latency between nodes
Fault tolerance: the system can continue operating despite individual node failures	Difficult consensus: getting many independent nodes to agree on a single fact or action is a genuinely hard distributed-computing problem
Improved performance through parallel processing across nodes	Synchronisation challenges: keeping data consistent across nodes accessed concurrently requires complex protocols
Lower latency than routing every request through one distant central server, since work can be handled by a geographically nearer node	Security challenges: every additional node and network link is a potential point of attack, widening the attack surface relative to a centralized system
Cost-effectiveness: systems can be built from many inexpensive, off-the-shelf machines rather than one very powerful (and very expensive) central machine	Higher development and operational cost for algorithm design and ongoing maintenance, despite cheaper hardware

THINK ABOUT IT — Think About It

Table 5.22 lists "cost-effectiveness" as an advantage and "higher development and operational cost" as a limitation. These are not contradictory; they refer to different costs. Before continuing, identify which cost each row is actually referring to (hint: consider hardware cost separately from the cost of the specialised engineering skill needed to design correct distributed algorithms).

CHECK YOUR PROGRESS 5.16

1. Define a distributed system, and explain what "transparency" means in this context.
2. Distinguish a homogeneous distributed database from a heterogeneous one.
3. Name the three architectures by which nodes in a distributed system may relate to one another.
4. Why does the absence of a global clock make consensus difficult in a distributed system?

5.17 CLUSTER COMPUTING

5.17.1 What is Cluster Computing?

Where Section 5.16 described distributed systems in general, **cluster computing** is one specific, tightly organised form of them, distinguished by how closely and uniformly its nodes are coupled.

IMPORTANT — Definition — Cluster Computing

A **cluster** is a group of tightly connected computers (typically similar hardware, running the same operating system, joined by a high-speed local-area network) that work together so closely that they can be managed and used as though they were a single, more powerful system.

Cluster computing matters because it offers most of the raw processing power of a large mainframe or supercomputer, assembled instead from many comparatively inexpensive, similar machines. This makes high-performance computing economically accessible to organisations that could not otherwise afford a purpose-built supercomputer, while still providing better scalability and availability than any single machine could offer alone.

5.17.2 Types of Cluster, by Purpose

Table 5.23 — Types of cluster, by purpose

Type	Purpose	Typical use
High-Performance (HP) clusters	Combine the processing power of many nodes to solve one large computational problem, with nodes communicating closely during the computation	Scientific simulation, large-scale data analysis
Load-balancing clusters	Distribute incoming requests across several nodes running identical software, so no single node is overwhelmed	Web hosting, high-traffic application serving
High-Availability (HA) clusters	Maintain redundant nodes ready to take over instantly if an active node fails, prioritising uninterrupted service over raw computational throughput	Business-critical databases, e-commerce platforms

Clusters are also classified by how exposed their nodes are to the outside network: an **open cluster** requires every node to have its own IP address reachable over the internet, which is more flexible but increases the security exposure of each node; a **closed cluster** hides its nodes behind a single gateway node, requiring fewer public IP addresses and offering better protection, at the cost of some flexibility.

5.17.3 Cluster Architecture and Components

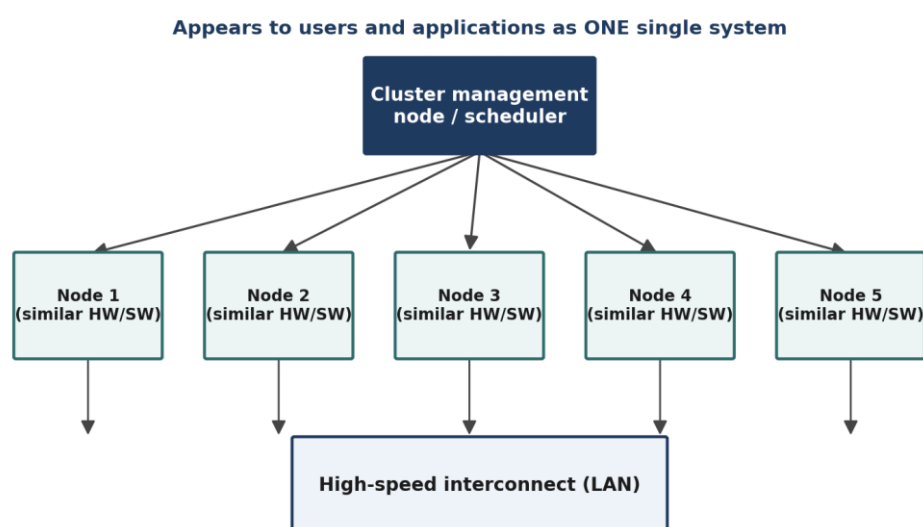


Figure 5.5 — Cluster computing architecture

A cluster's architecture connects two or more **nodes** (each a single- or multi-processor computer with its own memory and I/O) either directly to one another or, more commonly, through a local-

area network, so that they operate collectively as a single integrated computing resource. Four components recur across virtually every cluster implementation:

- **Cluster nodes:** the individual computers performing the actual computation.
- **Cluster operating system (or middleware):** software that presents the collection of nodes as a single, manageable system, scheduling work across them.
- **Node interconnect (switch):** the high-speed network hardware connecting nodes to one another.
- **Network switching hardware:** the broader networking infrastructure that carries traffic to and from the cluster as a whole.

5.17.4 Advantages, Disadvantages, and Applications

Table 5.24 — Advantages and disadvantages of cluster computing

Advantages	Disadvantages
Higher performance than a comparable single mainframe, at lower cost	High upfront hardware and design cost relative to a single machine of modest capacity
Easy to expand by adding further nodes (scalability)	Diagnosing which specific node or component has failed can be difficult
Improved availability: other nodes continue serving if one node fails	Requires more physical space, power, and cooling as more nodes are added

Cluster computing is applied wherever a computational problem is both large and separable into pieces multiple nodes can work on together: weather forecasting, image rendering, earthquake simulation, petroleum-reservoir simulation, and large-scale e-commerce platforms that need both raw throughput and high availability.

CHECK YOUR PROGRESS 5.17

1. Define cluster computing and explain what distinguishes it from a general distributed system (Section 5.16).
2. Distinguish a High-Availability cluster from a High-Performance cluster by their differing priorities.
3. Distinguish an open cluster from a closed cluster in terms of security exposure.
4. Name the four components common to most cluster implementations.

5.18 GRID COMPUTING

5.18.1 What is Grid Computing?

Grid computing is a form of distributed computing (Section 5.16) that, unlike a cluster (Section 5.17), does not require its participating machines to be similar, co-located, or purpose-built for the task. Instead, it pools the *spare, otherwise idle* computing resources of many independently owned machines, potentially spread across many different physical locations, to jointly tackle a task too large for any one of them.

IMPORTANT — Definition — Grid Computing

Grid computing is a form of distributed computing in which a network of independently owned computers contributes spare processing power and storage to a shared resource pool, coordinated by middleware, so that the pool functions collectively as a single virtual supercomputer.

The analogy the field's founders used is deliberate: just as an electrical power grid lets a consumer draw electricity on demand without knowing or caring which specific power station generated it, a computing grid lets a task draw computing power on demand without knowing or caring which specific machine actually executes it. This idea, described by Ian Foster and Carl Kesselman in 1999, is also a direct conceptual ancestor of cloud computing (Section 5.19), though grid computing itself remains a distinct and still-used model, particularly in scientific computing.

5.18.2 How Grid Computing Works

A grid computing network is built from three roles, which a single machine can move between over time:

- **Control node:** usually a server (or a small group of servers) that administers the network, keeps track of available resources, and decides how requests for resources are matched to providers.
- **Provider:** a computer contributing its otherwise-idle resources (processing power, storage) to the shared pool.
- **User:** a computer that is currently consuming resources from the pool to run its task.

Independently owned, geographically distributed resources pooled as a virtual supercomputer

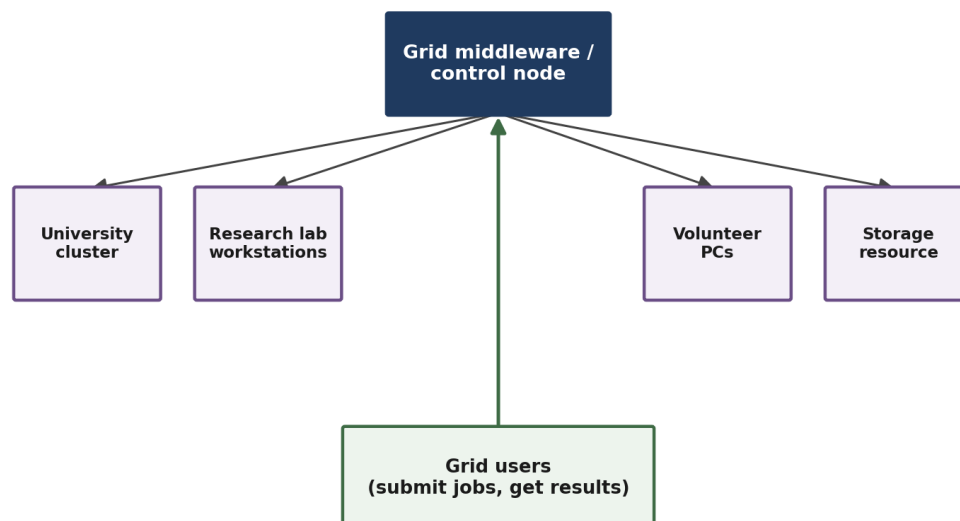


Figure 5.6 — Roles in a grid computing network

Coordination is handled by **middleware**: software that authorises which processes may run on the network, ensures no single provider is overloaded with work, and manages the network's resources on the control node's behalf. Middleware's authorisation role matters particularly because a grid, by its nature, allows a user's task to run on hardware it does not own; without careful control by the middleware, this openness would be a significant security risk (recall Section 5.1.3's framing of vulnerability and risk).

A grid is called **homogeneous** if all its machines run the same operating system and platform, and **heterogeneous** if they run a mixture. The ability to usefully combine heterogeneous machines is one of the properties that most clearly distinguishes grid computing from cluster computing, where uniformity is the norm.

5.18.3 Advantages and Disadvantages

Table 5.25 — Advantages and disadvantages of grid computing

Advantages	Disadvantages
No dedicated central processing server is required, only a lightweight control node for coordination	Grid middleware and standards are less mature and less standardised than cluster or cloud tooling

Advantages	Disadvantages
Can combine heterogeneous machines running different operating systems into one network	Requires very fast interconnects between resources to be effective, which is not always available across geographically dispersed providers
Tasks execute in parallel across many physical locations, often at no direct monetary cost to the user, since idle capacity is being reused	Software licensing that assumes a fixed number of machines can become complicated or prohibitively expensive across many providers
	Organisations are frequently reluctant to share their spare computing resources with others, limiting the pool's practical size

NOTE — Grid versus cluster versus cloud, at a glance

A **cluster** (Section 5.17) uses similar, co-located, purpose-dedicated machines under one administrative owner. A **grid** uses potentially dissimilar machines, often in different locations and under different owners, contributing spare capacity. **Cloud computing** (Section 5.19) takes the resource-pooling idea further still, adding commercial, on-demand self-service provisioning on top of virtualized infrastructure that a single provider owns and operates. Each is a legitimate answer to "how do I get more computing power than one machine provides?" They differ in ownership, uniformity, and how resources are provisioned.

CHECK YOUR PROGRESS 5.18

1. Define grid computing and explain the "power grid" analogy used to describe it.
2. Name the three roles a machine can occupy in a grid computing network, and explain how a single machine can move between them.
3. What is the role of middleware in a grid computing system, and why does that role matter for security?
4. Distinguish a homogeneous grid from a heterogeneous grid.

5.19 INTRODUCTION TO CLOUD COMPUTING

5.19.1 What is Cloud Computing?

Section 5.15's timeline described cloud computing as the point where cluster technology, virtualization, and internet-scale networking converged. This section defines the result precisely and examines its architecture.

IMPORTANT — Definition — Cloud Computing

Cloud computing is a model for delivering computing resources (processing power, storage, networking, and software) as an on-demand service over the internet, from a shared pool of configurable, virtualized infrastructure, rather than from hardware the user owns and manages directly.

The defining shift cloud computing makes, relative to every model examined so far in Part C, is **commercial and self-service**: a cluster or grid is typically assembled and administered by the organisation that uses it, whereas cloud computing lets a user provision computing resources instantly, pay only for what is actually used, and leave the underlying hardware entirely to the provider. A user of cloud computing can, within minutes, obtain storage, run a new server, or deploy an application, without purchasing, installing, or maintaining any physical equipment themselves.

Typical operations performed through cloud computing include storing and backing up data, delivering software on demand rather than through local installation, developing and testing new applications on infrastructure that can be created and destroyed as needed, and streaming audio and video at a scale that would be impractical to serve from a single organisation's own servers.

5.19.2 Cloud Computing Architecture

Cloud computing architecture is organised into three parts, corresponding to what the user directly interacts with, what the provider operates behind the scenes, and how the two are connected.

A customer's request travels front-end to back-end and the response travels back, over the network

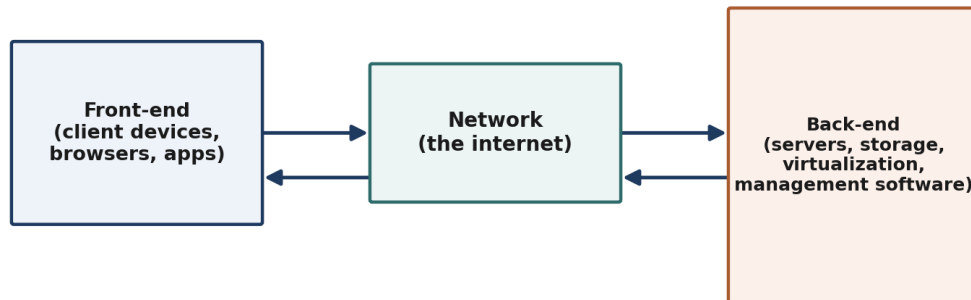


Figure 5.7 — Cloud computing architecture

- **Front end.** The interface through which the user accesses cloud services: a web browser, a mobile app, or a dedicated client application. Front ends are broadly grouped into **thin clients**, which do minimal local processing and rely on a browser for a lightweight, portable experience, and **fat clients**, which perform more processing locally for a richer, more capable experience.
- **Back end.** The provider's infrastructure that actually does the work: servers that run application logic, and storage systems that hold data. This is the part of cloud computing built most directly on the virtualization and physical infrastructure discussed in Section 5.25.
- **Network.** The connectivity binding front end to back end: the public **internet** for general access, an organisation's own **intranet** for internal-only access, and, where a task spans multiple cloud providers, an **intercloud** connection enabling interoperability between them.

EXAMPLE — Tracing a request through the architecture

A user opens a cloud-based email application in their phone's browser (front end, thin client). Their request travels over the mobile internet (network) to the provider's data centre, where a server retrieves the requested emails from a storage system (back end) and sends the result back along the same path. The user perceives none of this, only that their inbox appeared. This is the same *transparency* property introduced for distributed systems generally in Section 5.16.1, now delivered commercially, at global scale, by a cloud provider.

Sections 3.6 to 3.12 examine, in turn, the challenges cloud computing introduces, the properties and characteristics that formally define it, the different ways its services and deployments are packaged, and the specific resources (network, computation, and storage) that make it work.

CHECK YOUR PROGRESS 5.19

1. Define cloud computing and identify the single word that best captures how it differs from a cluster or a grid.
2. Name the three parts of cloud computing architecture and what each is responsible for.
3. Distinguish a thin client from a fat client.
4. What role does the "intercloud" play in cloud computing's network layer?

5.20 CLOUD ISSUES AND CHALLENGES

Cloud computing's benefits (examined formally in Section 5.21) come paired with a specific set of challenges, most of which follow directly from the same property that makes the cloud attractive in the first place: control over infrastructure passes from the user's organisation to an external provider. Recognising these challenges is essential both for examination purposes and for making informed real-world decisions about when and how to adopt cloud services.

Table 5.26 — Cloud computing: issues and challenges

Challenge	What it involves
Security and privacy	Data leaves the organisation's own premises and is processed on infrastructure shared with other, unrelated customers (multi-tenancy); a breach or misconfiguration at the provider can expose data belonging to many customers at once
Data governance and sovereignty	Many regulations require certain categories of data (financial, health, government) to remain within specific national borders; a global provider's default data placement may not satisfy this without deliberate configuration
Vendor lock-in	A provider's proprietary services and data formats can make it difficult and expensive to migrate to a different provider later, reducing the customer's future bargaining power and flexibility
Availability and downtime	Even large, well-run providers experience outages; an organisation depending entirely on one provider inherits that provider's downtime as its own
Cost management	The pay-per-use model (Section 5.21) is efficient when usage is well understood, but can produce unexpectedly large bills when usage scales up unpredictably or is not actively monitored
Latency and performance	Processing data in a distant data centre, rather than locally, introduces network delay that can matter for time-sensitive applications: the same limitation that motivated fog and edge computing in Section 5.15's timeline
Compliance and auditability	Demonstrating compliance with regulations (Section 5.7.3) is more complex when the organisation does not directly control the infrastructure being audited
Interoperability and portability	Cloud services from different providers rarely follow identical standards, making it difficult to combine services from multiple providers or move workloads between them without redesign

IMPORTANT — Shared responsibility

Cloud security is commonly described using a **shared responsibility model**: the provider is responsible for the security *of* the cloud (the physical data centres, the underlying virtualization and network infrastructure), while the customer remains responsible for security *in* the cloud (how they configure access controls, encrypt their own data, and manage user accounts, Sections 1.3 and 1.6). A large share of real-world cloud security incidents result from customer-side misconfiguration (for example, a storage bucket left publicly accessible) rather than a failure of the provider's own infrastructure. This shows that moving to the cloud changes an organisation's security responsibilities rather than eliminating them.

THINK ABOUT IT — Think About It

Choose any two challenges from Table 5.26 and explain, using concepts from Part A, one concrete control that would help mitigate each. (For example: which of cryptography, digital signatures, firewalls, or authentication mitigates the "security and privacy" risk of multi-tenancy?)

CHECK YOUR PROGRESS 5.20

1. Explain why multi-tenancy is a specific security concern for cloud computing that does not arise in the same way for infrastructure an organisation owns outright.
2. What is vendor lock-in, and why does it reduce a customer's future flexibility?
3. Describe the shared responsibility model, and identify one security task that remains the customer's responsibility even after moving to the cloud.
4. Explain why cost management is listed as a challenge, given that pay-per-use pricing is also cited as a benefit of cloud computing.

5.21 PROPERTIES AND CHARACTERISTICS OF CLOUD COMPUTING

5.21.1 General Properties of Cloud Computing

At a general level, cloud computing can be recognised by a small set of properties that, together, distinguish it from the models examined earlier in this Part.

Table 5.27 — General properties of cloud computing

Property	Explanation
Virtualized	Resources are delivered as virtual machines, containers, or virtual storage (Section 5.25), abstracted away from the specific physical hardware underneath
Service-oriented	Capability is consumed as a service (Section 5.22) rather than as owned equipment
Shared (multi-tenant)	The same underlying physical infrastructure serves many customers simultaneously, with virtualization keeping each customer's resources logically isolated
Elastic and on-demand	Capacity can be requested, provisioned, and released automatically as needs change, without manual procurement
Metered and pay-per-use	Usage is measured, and customers are billed for what they actually consume, much as a utility such as electricity is billed

These general properties explain the practical, business-facing benefits of cloud adoption: because infrastructure is virtualized and shared, the provider achieves economies of scale that let it offer resources more cheaply than most individual customers could provision for themselves; because it is metered, customers avoid large upfront capital expenditure on hardware and instead pay only for what they use; and because the provider operates the physical facilities, customers benefit from professional physical security and maintenance without staffing it themselves.

5.21.2 The Five Essential Characteristics (NIST Model)

While Section 5.21.1's properties describe cloud computing informally, the **U.S. National Institute of Standards and Technology (NIST)** definition, the model most widely cited in both academic and industry contexts, formalises cloud computing as having exactly five essential characteristics. A service that does not exhibit all five is arguably not "cloud computing" in the strict sense, however cloud-like it may appear.

Table 5.28 — The five essential characteristics of cloud computing (NIST)

Characteristic	Definition
On-demand self-service	A user can provision computing capabilities (server time, storage) automatically, without requiring human interaction with the provider
Broad network access	Capabilities are available over the network and accessed through standard mechanisms usable from diverse client platforms (laptops, mobile phones, tablets)
Resource pooling	The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned according to demand
Rapid elasticity	Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward with demand; to the consumer, capacity often appears unlimited
Measured service	Resource usage is automatically monitored, controlled, and reported, providing transparency for both the provider and the consumer

EXAMPLE — Applying the five characteristics to a familiar service

A streaming video platform hosted on a public cloud illustrates all five characteristics at once: a developer can spin up a new server instance through a web console with no phone call to the provider (**on-demand self-service**); the service is reachable from a smart TV, a phone, and a laptop alike (**broad network access**); the same physical servers simultaneously serve many unrelated customers of the cloud provider (**resource pooling**); server capacity automatically increases during a popular new release and decreases afterward (**rapid elasticity**); and the streaming platform's cloud bill itemises exactly how much storage and processing it consumed that month (**measured service**).

NOTE — Properties versus characteristics — why both are taught

Section 5.21.1's properties describe cloud computing's general nature in accessible, business-oriented language; Section 5.21.2's five characteristics are the precise, formally agreed technical definition used to decide whether a given service qualifies as cloud computing at all. An examination question asking to "list the characteristics of cloud computing" is almost always asking specifically for the five NIST characteristics in Table 5.28, not the general properties in Table 5.27, though a strong answer can note that the general properties follow naturally from the five formal characteristics.

CHECK YOUR PROGRESS 5.21

1. Name the five essential characteristics of cloud computing according to the NIST model.
2. Explain "resource pooling" in your own words, and connect it to the concept of multi-tenancy from Section 5.20.
3. Distinguish "rapid elasticity" from "on-demand self-service." Both involve a form of automatic provisioning, but they answer different questions.
4. Why might a service that lacks "measured service" (for example, a flat-fee, unlimited-usage plan) still be a legitimate cloud service, while arguably not satisfying the strict NIST definition?

5.22 CLOUD SERVICE MODELS

5.22.1 The "As a Service" Spectrum

Cloud services are organised into layers, each handling progressively more of the technology stack on the customer's behalf. Moving up the stack trades away control for convenience: at the lowest layer the customer manages almost everything except the physical hardware, while at the highest layer the customer manages almost nothing except how they use a finished application.

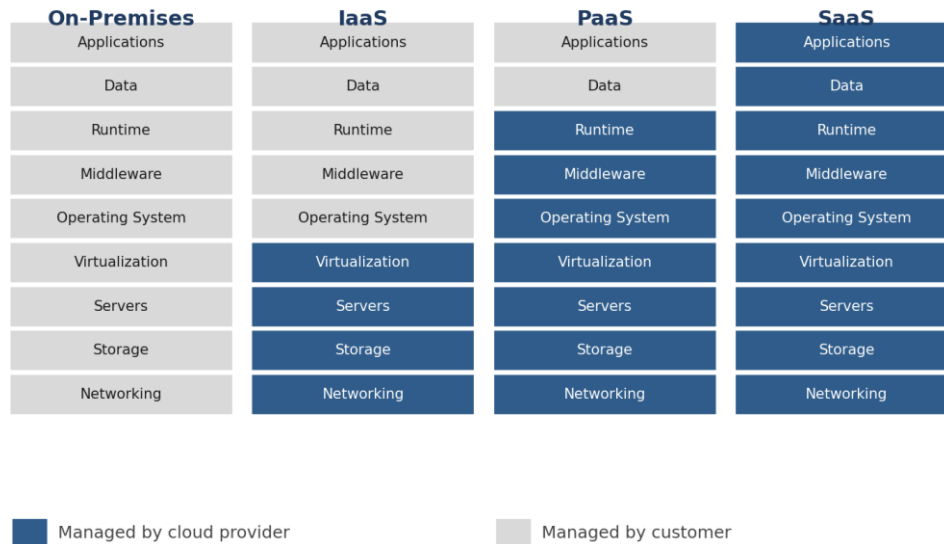


Figure 5.8 — The cloud service model stack

5.22.2 Infrastructure as a Service (IaaS)

IaaS provides the most fundamental building blocks of computing (virtual machines, storage, and networking) as a service, leaving the customer responsible for the operating system, middleware, and application software running on top.

Table 5.29 — Infrastructure as a Service (IaaS)

Property	Detail
What the provider manages	Physical servers, storage, networking, virtualization
What the customer manages	Operating system, runtime, middleware, application, data
Key benefit	Maximum flexibility and control over the software environment, without owning physical hardware
Typical use	Hosting custom applications with specific operating-system or configuration requirements

5.22.3 Platform as a Service (PaaS)

PaaS provides a ready-to-use platform (operating system, runtime environment, and development tools) on which a customer deploys their own application code, without needing to manage the underlying servers or operating system at all.

Table 5.30 — Platform as a Service (PaaS)

Property	Detail
What the provider manages	Everything IaaS manages, plus the operating system, runtime, and middleware
What the customer manages	Application code and data only
Key benefit	Developers focus entirely on application logic; the platform handles scaling and infrastructure maintenance automatically
Typical use	Rapid application development and deployment without infrastructure overhead

5.22.4 Software as a Service (SaaS)

SaaS delivers a complete, ready-to-use application over the internet, typically accessed through a web browser, with the provider managing every layer beneath the application itself.

Table 5.31 — Software as a Service (SaaS)

Property	Detail
What the provider manages	Everything: infrastructure, platform, and the application itself
What the customer manages	Only their own data and account configuration within the application
Key benefit	No installation, maintenance, or infrastructure knowledge required; updates and security patches are applied automatically by the provider
Typical use	Email, office productivity suites, customer-relationship-management software, accessed directly by end users

5.22.5 Function as a Service (FaaS) and the Serverless Extension

A newer model, **Function as a Service (FaaS)**, often called **serverless computing**, takes the "as a service" idea further still: the customer supplies only individual pieces of application code (**functions**), which the provider runs automatically in response to specific events, and for which the customer is billed only for the actual execution time consumed, down to fractions of a second. "Serverless" does not mean there are no servers; it means the customer never provisions, sees, or manages any server at all.

5.22.6 Comparing the Service Models

Table 5.32 — Comparing IaaS, PaaS, SaaS, and FaaS

Model	Analogy	Customer manages	Best suited to
IaaS	Renting an empty plot of land	OS, middleware, application, data	Custom infrastructure requirements
PaaS	Renting a fully built kitchen	Application code and data	Rapid application development
SaaS	Ordering a finished meal	Only account settings and data entered	End users who need a working application, not a platform to build on
FaaS	Paying only for the exact ingredients used in one dish	Only the function's code	Event-driven tasks with unpredictable or intermittent demand

CHECK YOUR PROGRESS 5.22

1. Order IaaS, PaaS, and SaaS from least to most managed by the provider, and explain what shifts at each step.
2. A company wants to build a custom application but does not want to manage servers or operating systems, only write code. Which service model fits best, and why?
3. Explain what "serverless" actually means, given that servers are, in fact, still involved.
4. Why is SaaS the model an end user (rather than a developer or system administrator) is most likely to interact with directly?

5.23 CLOUD DEPLOYMENT MODELS

5.23.1 What a Deployment Model Defines

Where Section 5.22 asked *how much* of the technology stack the provider manages, a **deployment model** asks a different question: *who owns and who may access* the underlying infrastructure. This

determines the balance a customer strikes between the cost efficiency of sharing infrastructure and the control of not sharing it.

5.23.2 The Five Deployment Models

Public cloud. Infrastructure is owned and operated by the provider and made available to any customer over the internet, typically on a pay-per-use basis. Multiple unrelated customers share the same underlying physical infrastructure (Section 5.21's multi-tenancy), with virtualization keeping their resources logically separated.

Private cloud. Infrastructure is dedicated to a single organisation, whether hosted on that organisation's own premises or by a third party on its behalf. No other customer shares the same infrastructure, giving the organisation the greatest control over configuration, security, and compliance, at the cost of losing the economies of scale a shared public cloud offers.

Hybrid cloud. Combines public and private cloud infrastructure, with data and applications able to move between the two as needed. For example, an organisation might run routine workloads on a private cloud for control, while bursting to public cloud capacity during periods of unusually high demand.

Community cloud. Infrastructure is shared by several organisations with common concerns (a shared industry, regulatory environment, or objective), typically managed by one of the participating organisations or a third party on their behalf. It combines some of a private cloud's targeted control with some of a public cloud's shared cost.

Multi-cloud. An organisation deliberately uses more than one public cloud provider simultaneously, rather than combining public and private infrastructure as hybrid cloud does. This mitigates the risk of a single provider's outage or lock-in (Section 5.20), and lets the organisation select each provider's particular strengths for different workloads.

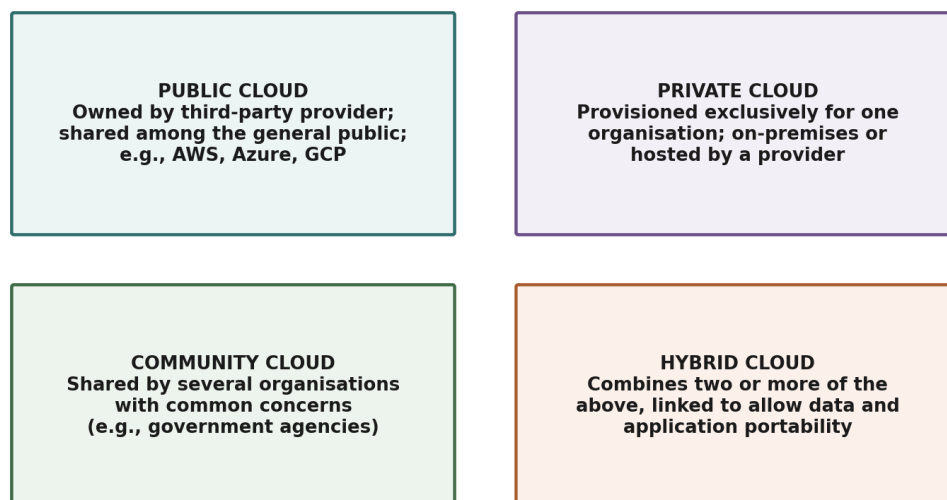


Figure 5.9 — Public, private, hybrid, and community cloud deployment models

DID YOU KNOW? — The public cloud market

As of early 2026, three providers (Amazon Web Services, Microsoft Azure, and Google Cloud) together account for roughly two-thirds of global public cloud infrastructure spending, according to Synergy Research Group, with AWS holding the largest individual share. Total quarterly spending on public cloud infrastructure services has continued to grow at over 30% year-on-year, driven substantially by demand for AI workloads. This shows how large a share of

the industry's infrastructure now runs on a small number of public cloud platforms rather than on organisations' own private infrastructure.

5.23.3 Comparing the Deployment Models

Table 5.33 — Comparing cloud deployment models

Factor	Public	Private	Community	Hybrid
Initial setup	Simple: no infrastructure to build	Complex: requires dedicated infrastructure and a skilled team	Complex: shared infrastructure still needs coordinated setup	Complex: must integrate two environments
Scalability	High	High, but limited to the organisation's own provisioned capacity	Limited: fixed by the participating organisations' shared capacity	High
Cost	Most cost-effective (shared infrastructure)	Most costly (dedicated infrastructure for one customer)	Cost shared among community members	Between public and private
Data security and privacy	Lowest (shared, open to any customer)	Highest (dedicated, fully controlled)	High (shared only among a defined, trusted group)	High for the private portion; depends on data placement

5.23.4 Choosing a Deployment Model

No single deployment model is universally "best." The right choice depends on the specific organisation's priorities, weighed against the factors in Table 5.33 and Table 5.34.

Table 5.34 — Factors in choosing a cloud deployment model

Factor to weigh	Question it answers
Cost	How much can the organisation afford to pay for the infrastructure it needs?
Scalability	How much, and how unpredictably, does demand vary?
Ease of use	How much in-house expertise exists to manage the chosen model?
Compliance	What regulations govern where and how the organisation's data may be stored (Section 5.20)?
Privacy	How sensitive is the data involved, and how much isolation from other customers does it require?

EXAMPLE — Matching a scenario to a deployment model

A hospital handling patient records, subject to strict health-data regulation (Sections 1.8, 3.6), is likely to favour a **private** or **community** cloud (shared, perhaps, only with other regulated healthcare providers) for the sensitive core system, while a **public** cloud may still be appropriate for its low-sensitivity public website. A start-up building a consumer app with unpredictable, potentially viral growth is far more likely to choose a **public** cloud, prioritising scalability and low upfront cost over the maximum control a private cloud would offer.

CHECK YOUR PROGRESS 5.23

1. Distinguish a public cloud from a private cloud in terms of ownership and access.
2. Explain how hybrid cloud differs from multi-cloud, since both involve more than one type or instance of cloud infrastructure.
3. Which deployment model would you recommend for an organisation with strict data-sovereignty requirements but a need to share infrastructure costs with similar organisations? Justify your answer using Table 5.33.

4. Name two of the five factors an organisation should weigh when choosing a deployment model.

5.24 CLOUD RESOURCES: NETWORK AND API

5.24.1 What is Cloud Networking?

Every cloud service described in Sections 3.8 and 3.9 depends on one resource operating correctly beneath all of them: the network connecting virtualized infrastructure, applications, and users. **Cloud networking** is the design, implementation, and management of network architecture within a cloud computing environment, delivering network services and connectivity from a pool of virtualized infrastructure rather than from fixed, manually configured physical hardware.

Traditional, pre-cloud networking meant physically installing and cabling routers and switches in a data centre, a slow, capital-intensive process. Cloud networking replaces this with **virtual networks** that can be created, modified, or removed through software in minutes, without touching any physical equipment.

5.24.2 Core Components of Cloud Networking

Table 5.35 — Core components of cloud networking

Component	Role
Virtual networks	Software-defined networks that direct traffic between cloud resources without the physical constraints of traditional network hardware
Cloud routers	Manage and direct data across the virtual network, ensuring it reaches its intended destination
Load balancers	Distribute incoming requests across multiple servers so that no single server is overwhelmed, directly supporting the load-balancing cluster concept introduced in Section 5.17.2
Virtual Private Cloud (VPC)	An isolated network environment within a public cloud, letting an organisation define its own private IP address ranges, subnets, and access policies while still running on shared physical infrastructure
Direct connect service	A dedicated, private network connection between an organisation's own premises and the cloud provider, improving performance and security compared with a connection over the public internet

Underlying all of these is **Software-Defined Networking (SDN)**, a technology that separates the network's **control plane** (the logic deciding how traffic should be routed) from its **data plane** (the hardware and software that actually forwards traffic). This separation is what allows a cloud provider to reconfigure an organisation's entire virtual network through software commands, rather than requiring someone to physically rewire equipment.

5.24.3 The Role of APIs

An **Application Programming Interface (API)** is a defined set of rules that lets separate software applications or components communicate with one another. In cloud computing specifically, APIs are the mechanism through which almost everything described in this Part is actually done: a customer's own application requests a new virtual machine, uploads a file to storage, or scales a service up or down, by calling the provider's API, not by any human clicking through a web console. Because a provider's entire service catalogue is exposed through APIs, automation (scripting the creation and destruction of infrastructure as demand requires) becomes possible at a scale manual administration could never match.

EXAMPLE — Why APIs made modern cloud automation possible

A retailer's application detects a surge in traffic during a flash sale. Rather than a system administrator manually creating new server instances, the application itself calls the cloud provider's API to request additional capacity, and calls it again afterward to release that capacity once traffic falls. This kind of automated, unattended scaling, a practical expression of the "rapid elasticity" characteristic from Section 5.21.2, depends entirely on the provider exposing its services through a well-designed, reliable API.

5.24.4 Benefits of Cloud Networking

Table 5.36 — Benefits of cloud networking

Benefit	Explanation
Scalability and flexibility	Network resources scale up or down on demand, without upfront investment in physical hardware
Cost-efficiency	Pay-as-you-go pricing means an organisation pays only for the networking capacity it actually consumes
Performance and reliability	Traffic is served from a globally distributed set of data centres, and redundancy across them reduces the risk of downtime
Security and compliance	Providers invest heavily in network-level security controls (encryption, intrusion detection; Sections 1.3 and 1.5) and support compliance with regulatory standards

CHECK YOUR PROGRESS 5.24

1. Explain, in one sentence, how cloud networking differs from traditional, pre-cloud networking.
2. What does a Virtual Private Cloud (VPC) provide that using the public cloud's shared network directly would not?
3. What separation does Software-Defined Networking (SDN) introduce, and why does that separation matter?
4. Explain, using the flash-sale example, why APIs are essential to achieving the "rapid elasticity" characteristic from Section 5.21.2.

5.25 VIRTUAL AND PHYSICAL COMPUTATIONAL RESOURCES

5.25.1 The Physical Layer

Beneath every virtual machine, container, and API call examined so far in Part C sits real, physical hardware: servers, processors, memory, storage drives, and networking equipment, housed in a cloud provider's **data centres**. These facilities are deliberately distributed across many geographic locations, both to keep services close to users worldwide (reducing latency, as discussed in Section 5.20) and to provide redundancy, so that a disaster affecting one location does not take the entire service offline. Providers invest heavily in the physical security of these facilities (restricted access, surveillance, environmental controls) precisely because, as Section 5.20's shared responsibility model notes, this remains the provider's responsibility even after a customer's data moves to the cloud.

5.25.2 Virtualization: Turning One Machine into Many

Virtualization is the technology that lets a single physical machine be divided into multiple, independent **virtual machines (VMs)**, each behaving as though it were its own separate computer, complete with its own operating system, isolated from the others sharing the same physical hardware.

IMPORTANT — Definition — Virtualization

Virtualization is the creation of a virtual (rather than physical) version of a computing resource, such as a server, storage device, or network, using software called a **hypervisor** to abstract and share the underlying physical hardware among multiple independent, isolated instances.

A **hypervisor** is the software layer that makes this possible, allocating each virtual machine its share of the physical machine's processor, memory, and storage, and keeping each VM's activity isolated from the others. Hypervisors are classified into two types:

- **Type 1 (bare-metal) hypervisors** run directly on the physical hardware, with no separate host operating system beneath them, and are the type used in most cloud data centres because of their performance and strong isolation between VMs.
- **Type 2 (hosted) hypervisors** run as an application on top of an existing host operating system, which is simpler to set up but adds an extra layer of overhead. This design is more common for running a VM on a personal desktop computer than in a production data centre.

Virtualization is what makes cloud computing's resource pooling and multi-tenancy (Section 5.21) practical: instead of dedicating one entire physical server to one customer's workload (which would waste most of that server's capacity most of the time), a provider runs many customers' independent virtual machines on the same physical hardware, allocating actual physical resources dynamically as each VM's demand changes.

5.25.3 Containers: A Lighter-Weight Alternative

Containers package an application together with everything it needs to run (libraries, configuration, dependencies) into a single, portable unit that runs consistently on any host with a compatible operating system kernel. Unlike a virtual machine, a container does not include its own full copy of an operating system; it shares the host machine's kernel, which makes containers significantly lighter-weight and faster to start than a comparable VM.

Table 5.37 — Virtual machines compared with containers

Property	Virtual Machine	Container
Includes its own OS	Yes, a full guest operating system	No, shares the host's OS kernel
Isolation	Strong: separate OS per VM	Good, but less isolated than a full VM, since the kernel is shared
Startup time	Slower: an entire OS must boot	Fast: typically seconds
Resource overhead	Higher	Lower
Typical tooling	Hypervisors (e.g., VMware, Hyper-V)	Docker (for building and running containers), Kubernetes (for orchestrating many containers across a cluster)

Containers and virtual machines are not mutually exclusive in practice: cloud providers commonly run containers *inside* virtual machines, combining the strong isolation of virtualization at the infrastructure level with the lightweight portability of containers at the application level.

CHECK YOUR PROGRESS 5.25

1. Define virtualization and explain the role of a hypervisor.
2. Distinguish a Type 1 (bare-metal) hypervisor from a Type 2 (hosted) hypervisor.
3. Explain the key architectural difference between a virtual machine and a container, and why that difference makes containers faster to start.
4. Why do cloud providers still invest heavily in physical data-centre security, given that most of what a customer interacts with is virtual?

5.26 DATA STORAGE IN THE CLOUD

5.26.1 What is Cloud Storage?

Cloud storage is a cloud computing model in which data is stored on infrastructure owned, hosted, and managed by a third-party provider, rather than on the customer's own on-premises hardware, and is accessed over the internet from anywhere. Cloud storage is one of the most widely adopted cloud services precisely because storage needs grow predictably and continuously for almost every organisation, making its cost-effectiveness and scalability especially valuable.

5.26.2 Types of Cloud Storage

Cloud storage systems are classified by the architecture they use to organise data, and different applications are matched to the type best suited to how they need to read and write that data.

Table 5.38 — Types of cloud storage

Type	How data is organised	Best suited to
Object storage	Unstructured data stored as discrete objects within flat containers called "buckets," each with associated metadata	Media files, backups, and large volumes of unstructured data needing flexible, scalable access
File storage	Data organised hierarchically, in folders and files, much like a conventional file system	Shared documents and applications that need to browse and access files by path
Block storage	Data divided into fixed-size blocks, each stored and addressed independently, offering high-performance read and write	Databases, virtual machine disks, and other workloads needing fast, low-latency access

5.26.3 Requirements of a Viable Cloud Storage Solution

Table 5.39 — Requirements of cloud storage

Requirement	Why it matters
Resilience	Storage must survive hardware failure, natural disaster, and data loss without losing customer data. Providers achieve this chiefly through redundancy, keeping multiple copies of data across physically separate locations
Accessibility	Data must be retrievable remotely, from multiple devices and regions, without requiring physical proximity to the storage hardware
Security	As sensitive data, stored data must be encrypted (Section 5.3) at multiple layers and accessible only after proper authentication (Section 5.6)

5.26.4 Benefits and Challenges

Table 5.40 — Benefits and challenges of cloud storage

Benefits	Challenges
Cost-effectiveness: avoids the capital expense of in-house storage infrastructure	Business downtime: reliability depends on the provider's own infrastructure and the customer's internet connection
Scalability: capacity can be increased on demand as data volume grows	Choosing the right provider: pricing, bandwidth, and industry-specific expertise vary significantly between providers
File sharing and synchronisation: enables multiple users, in different locations, to work on the same data with automatic consistency	Ill-defined Service Level Agreements (SLAs): vague commitments on uptime, support, and resolution time create ambiguity when problems occur
Disaster recovery: provides an off-site backup that survives a local hardware failure	Data mobility: migrating large volumes of data from one provider to another can be slow, costly, and technically difficult

Benefits	Challenges
Compliance support: providers implement controls that help customers meet regulatory data-handling requirements (Section 5.7.3, Section 5.20)	Bandwidth limitations: retrieving very large volumes of data can be constrained by available network bandwidth and, in some pricing models, its cost

IMPORTANT — Service Level Agreements (SLAs)

A **Service Level Agreement (SLA)** is a formal, documented commitment between a cloud provider and a customer, specifying measurable service expectations such as guaranteed uptime percentage, data-recovery time objectives, and support response times. A well-defined SLA is what gives a customer a concrete, enforceable basis for holding a provider accountable, closing precisely the ambiguity a poorly defined SLA leaves open, as noted in Table 5.40.

CHECK YOUR PROGRESS 5.26

1. Distinguish object storage, file storage, and block storage by how each organises data.
2. Name the three basic requirements every viable cloud storage solution must meet.
3. What is a Service Level Agreement, and why does a poorly defined one create risk for the customer?
4. Explain why "data mobility" is listed as a challenge, given that switching cloud providers might otherwise seem as simple as copying files from one place to another.

5.27 SUMMARY

Part A built information security on three properties that any secure system must preserve: confidentiality, integrity, and availability. Threats exploit vulnerabilities to create risk, and malicious programs, whether viruses, worms, Trojans, ransomware, spyware, or adware, are the most direct expression of threat, each one distinguished by how it spreads and what it does once installed. Cryptography protects confidentiality and underpins authentication in two complementary forms: symmetric-key algorithms, which are fast but need a pre-shared key, and public-key algorithms, which solve key distribution at the cost of speed. Cryptographic hashing supplies the tamper-evident fingerprint that both digital signatures and password authentication rely on, and digital signatures themselves combine hashing with public-key cryptography to deliver authentication, integrity, and non-repudiation. Firewalls control traffic at the network boundary, and identification and authentication (strengthened by combining knowledge, possession, and inherence factors) confirm who is on the other end of a connection before granting access. None of this technology works on its own, though: security awareness and policy address the human element that technical controls cannot reach, and the stakes of getting all of it right show up across every sector that handles sensitive information, from finance and healthcare to critical infrastructure.

Part B turned to mobile commerce, a security-dependent subset of e-commerce set apart by ubiquity, context-awareness, and immediacy. Its rapid growth has come from smartphone adoption, improved mobile hardware, and payment systems people trust enough to use, and Indian platforms such as Flipkart, Myntra, Ola, Zomato, and Paytm illustrate how these factors play out in practice. A specific stack of technologies makes m-commerce work: apps and mobile websites, progressive web apps, mobile payment solutions, location-based services, and AR, VR, and AI. India stands out as a mobile-first market, one whose scale is easiest to grasp through UPI's transaction volumes. Digital marketing, spanning channels from search engine optimisation to influencer marketing, brings in the traffic that converts into sales, with analytics closing the loop by showing which channels are actually working.

Part C traced computing's path from centralized systems through distributed computing in general to two specific forms of it: cluster computing, where similar machines are tightly coupled under one owner, and grid computing, where independently owned, often dissimilar machines pool spare capacity. That path arrives at cloud computing, where clustering, virtualization, and internet-scale networking converge into a commercial, self-service model. Five essential characteristics, on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service, formally define what counts as cloud computing, while its issues and challenges (security, vendor lock-in, cost management, and more) mostly trace back to control passing from the customer's organisation to an external provider under a shared responsibility model. Service models (IaaS, PaaS, SaaS, FaaS) settle how much of the technology stack the provider manages; deployment models (public, private, hybrid, community, multi-cloud) settle who owns and may access the infrastructure. Underneath both sit cloud networking and APIs, the virtualization and container technology that lets one set of physical hardware serve many customers at once, and the object, file, and block storage systems that hold the data itself.

One thread runs through all three parts: scale, whether of threats, of transactions, or of computing demand, cannot be handled safely or efficiently by ad hoc, manual methods. Cryptographic protocols, mobile platform architectures, and cloud computing models are each a considered, engineered answer to the same underlying problem: operating securely and efficiently at a scale no single person or machine could manage by hand.

5.28 GLOSSARY

Term	Meaning
Information security (InfoSec)	Protecting information and systems from unauthorised access, disclosure, modification, or destruction
CIA triad	Confidentiality, integrity, and availability: the three core properties information security preserves
Threat / vulnerability / risk	A potential cause of harm; a weakness a threat can exploit; the resulting likelihood and impact if it does
Malware	Software deliberately written to damage, disrupt, or gain unauthorised access to a system
Virus / worm / Trojan	Malware attached to a host file; malware that self-replicates across a network unaided; malware disguised as legitimate software
Ransomware	Malware that encrypts or locks a victim's data and demands payment for its restoration
Cryptography	Using algorithms and keys to protect the confidentiality, integrity, authenticity, and non-repudiation of data
Symmetric-key cryptography	Encryption and decryption using the same shared secret key
Public-key (asymmetric) cryptography	Encryption and decryption using a mathematically related public/private key pair
Cryptographic hash function	A one-way function producing a fixed-size, tamper-evident digest of any input
Digital signature	A cryptographic value proving a message's authenticity, integrity, and the signer's non-repudiation
Digital certificate	A trusted third party's binding of a public key to a verified real-world identity
Firewall	A system controlling network traffic across a boundary according to security rules
Access Control List (ACL)	A firewall's set of rules specifying which traffic to permit or deny
Identification / authentication / authorisation	Claiming an identity; verifying that claim; determining what a verified identity may do
Multi-Factor Authentication (MFA)	Authentication requiring two or more factors (knowledge, possession, inherence)
Security policy	An organisation's formal, written rules governing information security

Term	Meaning
Mobile commerce (m-commerce)	Buying and selling conducted through wireless handheld devices
Progressive Web App (PWA)	A mobile website built to behave like a native app (offline use, push notifications, home-screen install)
Location-Based Services (LBS)	Services that use a device's determined location to personalise content or offers
Digital marketing	Promoting products or brands using digital channels such as search, social media, and email
UPI (Unified Payments Interface)	India's instant, bank-to-bank mobile payment system
Centralized system	An architecture in which processing and data are concentrated in a single central node
Distributed system	Independent, networked computers that appear to users as a single system working toward a common goal
Transparency (distributed systems)	Hiding the fact of distribution from the end user
Cluster computing	Tightly connected, similar computers managed and used as a single system
Grid computing	Independently owned computers pooling spare resources, coordinated by middleware, into a virtual supercomputer
Middleware	Software that coordinates and authorises activity across a distributed, grid, or cluster network
Cloud computing	On-demand delivery of computing resources as a service over the internet from virtualized, shared infrastructure
Multi-tenancy	Multiple customers sharing the same underlying physical cloud infrastructure, kept logically isolated
Shared responsibility model	The division of security duties between a cloud provider (security of the cloud) and customer (security in the cloud)
Vendor lock-in	Difficulty migrating away from a provider due to proprietary services or data formats
On-demand self-service	A NIST cloud characteristic: provisioning resources without human interaction with the provider
Rapid elasticity	A NIST cloud characteristic: capacity scaling automatically with demand
Measured service	A NIST cloud characteristic: automatic monitoring, control, and reporting of resource usage
IaaS / PaaS / SaaS / FaaS	Infrastructure / Platform / Software / Function as a Service: cloud service models by how much of the stack the provider manages
Public / private / hybrid / community / multi-cloud	Cloud deployment models, differing in who owns and may access the underlying infrastructure
Virtual Private Cloud (VPC)	An isolated network environment within a public cloud
Software-Defined Networking (SDN)	Separating a network's control plane from its data plane to enable software-based management
Application Programming Interface (API)	A defined set of rules letting separate software components communicate
Virtualization	Creating a virtual version of a computing resource using a hypervisor to share physical hardware
Hypervisor (Type 1 / Type 2)	Software creating and managing virtual machines; bare-metal versus hosted-on-an-OS
Container	A lightweight, portable unit packaging an application with its dependencies, sharing the host OS kernel
Object / file / block storage	Cloud storage organised as flat objects with metadata; as a hierarchical file system; as fixed-size, independently addressed blocks
Service Level Agreement (SLA)	A formal, measurable commitment between a provider and customer on service expectations

CHAPTER 6

Internet of Things, Virtual Reality, and Artificial Intelligence

6.1 INTRODUCTION TO THE INTERNET OF THINGS

6.1.1 What is the Internet of Things?

IMPORTANT — Definition — Internet of Things

The **Internet of Things (IoT)** is a global infrastructure of physical objects, or "things," each of which has a unique identity, is equipped with sensing and/or actuation capability and with embedded computing and communication capability, and is connected to a network so that it can exchange data with other things, with services, and with applications, in most cases without direct human intervention.

Four clauses carry the entire weight of that definition.

"Physical objects." The elements of an IoT system are not documents or web pages but *material* objects: a valve, a lamp, a pallet, a tractor, a human body wearing a device. Because the endpoints are physical, IoT systems inherit the awkward properties of the physical world: objects move, corrode, run out of battery, and sit inside metal enclosures that block radio signals.

"Unique identity." Every *thing* must be individually addressable and distinguishable. If a factory has four hundred temperature sensors, a reading of 78 °C is meaningless unless the system knows *which* sensor produced it. Identity is provided in layers: a network address (typically IPv6), a device-level identifier (MAC address, EPC code, serial number), and an application-level name such as plant1/lineB/motor3/bearing-temp.

"Sensing and/or actuation." A **sensor** converts a physical quantity into an electrical signal that can be digitised: the *input* direction. An **actuator** converts an electrical signal into a physical effect: the *output* direction. A device with sensing only observes; one with actuation only acts; one with both can close a control loop by itself.

"Without direct human intervention." This clause produces most of IoT's economic value and most of its engineering difficulty. Because no human is in the loop, the system must handle its own failures, configuration, security and decision-making. A web application can display an error and wait for the user; a soil sensor two kilometres from the nearest road cannot.

NOTE — Sensor versus transducer

A **transducer** is any device converting energy from one form to another. A sensor is a transducer used for *measurement*; an actuator is a transducer used for *control*. In everyday usage the words are interchanged, but the distinction above is the correct one for an examination answer.

6.1.2 Why IoT emerged when it did

The idea itself is not new. The phrase is usually credited to Kevin Ashton, who used it in a 1999 presentation about RFID in supply chains. What changed was not the idea but its economics. Five cost curves fell far enough, at roughly the same time, to make large-scale deployment viable: **MEMS manufacturing** made sensors cheap; **32-bit microcontrollers** with integrated radios and deep sleep modes became inexpensive; **connectivity** became ubiquitous through Wi-Fi indoors and NB-IoT, LTE-M and LoRaWAN outdoors; **cloud platforms** made storage and computation elastic and pay-per-use; and **IPv6**, with its 128-bit addresses, removed the objection that there were not enough addresses in the world to give every object one.

DID YOU KNOW? — The scale of the thing

Industry analysts estimated approximately **21.1 billion connected IoT devices worldwide at the end of 2025**, growing about 14% year-on-year, with projections of roughly **39 billion by 2030**; Wi-Fi, Bluetooth and cellular together accounted for around 80% of connections. Treat such figures as *indicative orders of magnitude*: different analysts count different things, and the numbers change every year.

6.1.3 The conceptual model: from thing to insight

Every IoT system, from a two-rupee temperature logger to a national smart-grid deployment, performs the same five-stage journey.

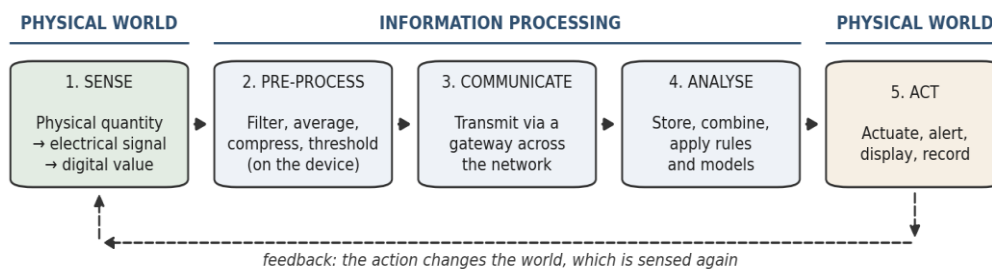


Figure 6.1 — The conceptual pipeline of an IoT system

1. **Sense.** A physical phenomenon is converted into an electrical signal by a sensor and digitised by an analog-to-digital converter.
2. **Pre-process locally.** The embedded computer filters, averages, compresses or thresholds the raw signal. This stage matters enormously: transmitting every raw sample is usually the largest single consumer of energy in the whole system.
3. **Communicate.** The processed data travels, usually via a gateway, to a server or cloud platform.
4. **Process and analyse.** The platform stores the data, combines it with historical and contextual data, and applies rules, statistics or machine-learning models to extract meaning.
5. **Act and present.** The conclusion becomes an outcome: an actuator is commanded, an alert raised, a dashboard updated, or a record written.

THINK ABOUT IT — Think About It

Stages 1 and 5 touch the physical world; stages 2–4 are pure information processing. Now ask: for a system that must detect a gas leak and shut a valve within two seconds, where should the decision logic live, in the cloud or on the device? Hold your answer; Sections 6.5.4 and 6.8.2 return to exactly this question.

6.1.4 IoT compared with related concepts

Table 6.1 — IoT distinguished from related concepts

Concept	Core idea	Endpoints	Relationship to IoT
Conventional Internet	Interconnecting computers to exchange information for human use	Servers, PCs, phones operated by people	Same underlying network, but IoT adds non-human endpoints and machine-initiated traffic
M2M (Machine-to-Machine)	Direct, usually point-to-point communication between two machines for one purpose	A device and one dedicated server, often over a proprietary non-IP link	IoT is the generalisation: IP-based, many-to-many, open, integrated with Internet services
WSN (Wireless Sensor Network)	Spatially distributed sensor nodes cooperating to monitor a phenomenon	Homogeneous, resource-constrained nodes, multi-hop to a sink	Usually a <i>subsystem</i> of an IoT deployment: the sensing tier feeding a gateway
CPS (Cyber-Physical System)	Tight, often real-time integration of computation with physical processes under feedback control	A physical process and its controllers	Overlaps heavily. CPS emphasises <i>control and timing correctness</i> ; IoT emphasises <i>connectivity, scale and services</i>
Ubiquitous computing	Computing embedded everywhere, disappearing into the background	Any computational artefact in the environment	A vision rather than an architecture; IoT is a principal means of realising it

IMPORTANT — Examination point

"Differentiate between IoT and M2M" is frequently asked. The three discriminators that earn marks are **addressing** (M2M often non-IP and proprietary, IoT IP-based), **topology** (M2M point-to-point between one device and one application, IoT many-to-many with data shared across applications) and **openness** (M2M closed vertical silos, IoT open standards and web-service integration).

6.1.5 Four essential components

Regardless of scale or domain, four components are always present. Verifying that all four are properly specified is a good first move when you are asked to design or critique an IoT system.

1. **The device (thing)** with its sensors and/or actuators, built around an embedded computer.
2. **Connectivity:** the communication technology and protocol stack, frequently including a **gateway** that translates between a constrained local protocol and the Internet.
3. **Data processing and storage:** the server-side or edge-side software that receives, stores, validates and analyses.
4. **The user (and machine) interface:** dashboard, mobile application, alert channel or API through which results are consumed and commands issued.

EXAMPLE — A minimal but complete example

A cold-storage warehouse must keep vaccines between 2 °C and 8 °C. **Device:** a battery-powered node with a digital temperature sensor reporting every five minutes. **Connectivity:** BLE to a corridor gateway, which forwards readings over the Internet using MQTT. **Processing:** a cloud platform stores every reading; a rule engine alerts if two consecutive readings fall outside the band. **Interface:** a dashboard of live temperatures, an SMS to the duty pharmacist on alert, and a monthly PDF compliance report.

Remove the gateway and the node cannot reach the Internet; remove the rule engine and data accumulates but nobody is warned; remove the interface and the alert never reaches a human who can open the door and check.

CHECK YOUR PROGRESS 6.1

1. Define the Internet of Things, and state the four clauses that distinguish it from the conventional Internet.
2. Distinguish between a sensor and an actuator, giving one example of each.
3. Name the five stages of the IoT conceptual pipeline in order.
4. Give two precise points of difference between IoT and M2M communication.
5. Name the four essential components of any IoT system.

6.2 CHARACTERISTICS OF IoT

An IoT system is not merely "a network with sensors attached". Each of its characteristics forces a specific design decision, and it is the *pairing* of characteristic with consequence that turns a list into usable engineering knowledge. Table 6.2 is the summary; the notes below add what the table cannot carry.

Unique identity. Every thing carries an identity persisting over its lifetime, because data without provenance is useless. Consequently, **provisioning** (how a factory-fresh device acquires its identity and credentials) becomes a first-class engineering problem rather than an afterthought.

Heterogeneity. A deployment routinely mixes devices from different manufacturers, on different platforms, speaking different protocols, because no single vendor supplies everything and deployments grow over years. This is the primary argument for **middleware**: a well-designed platform exposes a uniform data model and hides the difference between a Modbus-speaking industrial sensor and a BLE wearable behind adapters.

Connectivity and integration. Things are members of a wider information system, not isolated data loggers. IoT data must therefore be *exposed*, not merely *stored*: this is why well-documented APIs matter as much as the sensing hardware. Commercial value usually appears only when sensor data is joined with other data; sensor readings joined with maintenance records; energy readings joined with tariff schedules.

Dynamic behaviour and self-adaptation. Devices move, join, leave, sleep and fail. A surveillance camera lowers its frame rate and switches to infrared as light falls; a fleet tracker reports less often when stationary. Applications must therefore tolerate missing, late and out-of-order data. Any design assuming "a reading from every device every minute, on time" will fail in the field.

Self-configuration. Consider a utility deploying two hundred thousand smart meters: if configuring each required an engineer's visit, the deployment would be impossible. IoT platforms therefore need automated onboarding, service discovery, and **firmware over-the-air (FOTA)** update with rollback. FOTA is not a convenience: without it, a vulnerability discovered after deployment can never be patched.

Interoperable protocols. Things must communicate using open, mutually intelligible protocols. This is the force behind IEEE 802.15.4, 6LoWPAN, CoAP, MQTT and, in the smart home, Matter.

Resource constraint. End devices are constrained in energy, processing, memory and bandwidth; a node may run for years on a coin cell with tens of kilobytes of RAM. This single characteristic explains more IoT design decisions than any other: lightweight binary protocols instead of verbose HTTP, aggressive duty-cycling, local pre-processing, and careful selection of cryptographic operations.

IMPORTANT — The energy hierarchy

For a battery-powered wireless node, **transmitting** a bit costs vastly more energy than **computing** on it, which costs more than **sleeping**: transmitting one bit can cost as much as executing hundreds to thousands of instructions. This is precisely why local pre-processing

(send the average, or only threshold violations) extends battery life so dramatically: you spend cheap computation to avoid expensive transmission. Expect this trade-off to be examined.

Intermittent connectivity. Devices are frequently *not* connected: they sleep, move out of coverage, and use lossy networks. Protocols must tolerate this through broker queueing, store-and-forward buffering, retained messages, last-will notification, and idempotent command semantics so a command delivered twice does no harm.

Massive scale. Endpoint counts and data volumes are orders of magnitude beyond conventional IT, requiring horizontally scalable event-driven backends, time-series rather than general-purpose relational storage, retention and down-sampling policies, and automation, because manual administration does not scale to a million devices.

Context awareness. A reading of 40 °C means one thing from a server-room sensor at 3 a.m. and something quite different from an outdoor wall in Nagpur in May. The number is identical; the meaning is not. Metadata (location, calibration date, units, device type) must therefore travel and be stored with every measurement, or the data becomes uninterpretable within months.

Security and privacy as intrinsic concerns. Because IoT senses and acts on the physical world, a security failure has physical consequences (an unlocked door, a manipulated process) and the data collected is frequently personal. Security must be designed in at every layer, as Section 6.5 develops.

Table 6.2 — Characteristics of IoT and the design consequence each produces

Characteristic	Principal design consequence
Unique identity	Layered addressing and naming; secure provisioning of identity and credentials
Heterogeneity	Middleware, abstraction layers, protocol adapters, common data models
Connectivity and integration	Well-documented open APIs; data exposed as a service, not siloed
Dynamic and self-adapting	Applications tolerant of missing, late and out-of-order data
Self-configuration	Automated onboarding, service discovery, firmware over-the-air updates
Interoperable protocols	Adoption of open standards (802.15.4, 6LoWPAN, CoAP, MQTT, Matter)
Resource constraint	Lightweight binary protocols, duty cycling, local pre-processing
Intermittent connectivity	Queueing, store-and-forward, retained messages, idempotent commands
Massive scale	Horizontally scalable event-driven backends; time-series storage; automation
Context awareness	Metadata carried and stored with every measurement
Security and privacy	Defence in depth: secure boot, mutual authentication, encryption, access control

CHECK YOUR PROGRESS 6.2

1. List any six characteristics of IoT.
2. Using the energy hierarchy, explain why an IoT node should transmit an average rather than every raw sample.
3. What is FOTA, and why is it essential rather than optional?
4. Why does context awareness require metadata? Give an example where one numerical reading has two meanings.
5. State the design consequence that follows from intermittent connectivity.

6.3 PHYSICAL DESIGN OF IoT

6.3.1 What "physical design" means

The **physical design** of IoT concerns the tangible elements of a system: what a thing is physically made of, what components it contains, and what protocols actually run on the wire and in the air. Contrast it with the topic of the next section:

- **Physical design** answers *what hardware is present, and what protocols does it speak?* In short, things and protocols.
- **Logical design** (Section 6.4) answers *how is the system organised, abstractly, in terms of components, interactions and responsibilities?* In short, functional blocks, communication models and APIs, deliberately independent of hardware.

NOTE — An analogy that helps

In software engineering, a class diagram is a logical design and the compiled binary on a particular server is a physical realisation. In IoT, "the device publishes readings and a subscriber consumes them" is logical design; "an ESP32 transmits an MQTT PUBLISH packet over 802.11 b/g/n at 2.4 GHz to a broker" is physical design. Both describe the same system at different levels of abstraction.

6.3.2 Things in IoT: the generic block diagram

Almost every IoT device is built from the same seven functional blocks. When asked to draw "the block diagram of an IoT device", this is the expected answer.

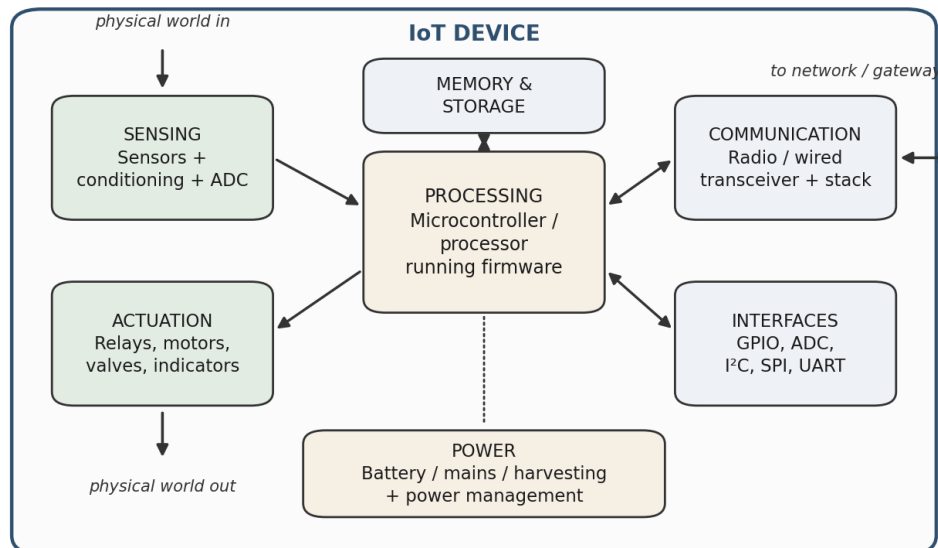


Figure 6.2 — Generic block diagram of an IoT device

1. **Sensing:** sensors, signal-conditioning circuitry, and the analog-to-digital converter (ADC) that turns a continuous voltage into a number.
2. **Actuation:** relays, motor drivers, valves, indicators. Many devices contain none.
3. **Processing:** the embedded computer running the firmware that samples sensors, applies local logic and drives the communication stack (Section 6.12).
4. **Memory and storage:** program flash, working RAM, and a small non-volatile store for configuration and for buffering readings during a network outage.

5. **Communication:** the radio or wired transceiver (Wi-Fi, BLE, 802.15.4, LoRa, cellular, Ethernet) with its protocol stack.
6. **Power:** battery, mains or energy harvesting, plus regulation and power management. In battery designs this block dictates almost every other decision.
7. **Interfaces:** I/O pins (GPIO, ADC/DAC) and device-to-device buses (I²C, SPI, UART).

NOTE — Interface buses you should recognise

UART: asynchronous serial, two wires, point-to-point; common for debug consoles and GPS/GSM modules. **I²C:** two wires (clock and data), addressed, multi-device, moderate speed; ideal for several low-rate sensors on one bus. **SPI:** four wires, master/slave, no addressing (a chip-select line per device), much faster; used for displays, SD cards and high-rate sensors. A common question asks for the I²C/SPI distinction: *I²C uses fewer wires and addresses devices on a shared bus; SPI uses more wires and is faster.*

6.3.3 Sensors and actuators

A **sensor** converts a physical quantity into a measurable electrical signal. **Analog sensors** (an LM35, a light-dependent resistor) produce a continuously varying voltage and need an ADC; **digital sensors** (a DHT22, a BMP280) contain their own conversion circuitry and deliver a value over I²C or SPI, making them easier to interface and less noise-prone on long wires. **Active sensors** emit energy and measure what returns (ultrasonic, radar, lidar); **passive sensors** merely receive existing energy (thermistor, microphone, PIR detector).

Table 6.3 — Sensor performance parameters

Parameter	Meaning
Range	Minimum and maximum values the sensor can measure
Resolution	Smallest change the sensor can detect
Accuracy	Closeness of a measurement to the true value
Precision (repeatability)	Closeness of repeated measurements to one another
Sensitivity	Ratio of change in output to change in the measured quantity
Response time	How quickly the output settles after a step change in input
Drift	Slow change in output over time at constant input; corrected by recalibration
Power consumption	Energy per reading and in standby, decisive for battery nodes

IMPORTANT — Accuracy is not precision

A sensor that always reads exactly 3 °C above the true temperature is **precise** (repeatable) but **inaccurate** (biased). One whose readings scatter randomly around the true value is on average accurate but imprecise. A systematic bias can be corrected by calibration; random scatter can only be reduced by averaging. This distinction is a standard examination question.

Sensor types commonly deployed include temperature (thermistor, RTD, IC), humidity, pressure, proximity (inductive, capacitive, ultrasonic), PIR motion, MEMS accelerometer and gyroscope, light, gas, current (Hall effect or CT clamp), soil moisture, CMOS image sensors and GNSS receivers.

An **actuator** converts an electrical control signal into a physical effect. The categories are electromechanical (relays and contactors for switching high-power loads, solenoids for linear motion and locks), motors (DC for continuous rotation, servo for precise angular position, stepper for precise increments), hydraulic and pneumatic where large forces are needed, thermal (heaters, Peltier devices), and optical or acoustic indicators that actuate on the *human* rather than on a machine.

NOTE — Why actuation is riskier than sensing

A compromised sensor produces false data, which is serious, but recoverable. A compromised actuator produces false *action*: an opened lock, a stopped pump, a released valve. Authentication and authorisation requirements for actuation are therefore always stricter than for telemetry, and safety-critical actuation should retain an independent physical interlock that software cannot override.

6.3.4 The IoT protocol stack

The IoT protocol stack follows the familiar TCP/IP layering but adds, at each layer, protocols designed for constrained devices and lossy networks.

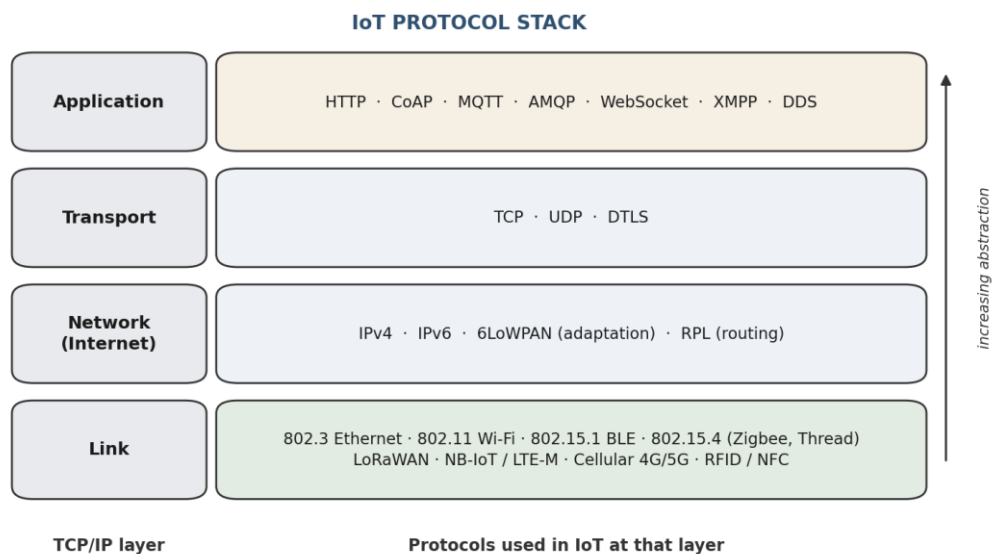


Figure 6.3 — The IoT protocol stack compared with the TCP/IP reference model

(a) Link layer. This determines how bits cross the medium and is generally the single most consequential physical-design decision, because it fixes the range, data rate and power budget of the whole system.

Table 6.4 — Link-layer technologies and their operating envelopes

Technology	Standard	Typical range	Typical data rate	Characteristic use
Ethernet	IEEE 802.3	~100 m cable	10 Mbps – 100 Gbps	Fixed industrial equipment, gateways
Wi-Fi	IEEE 802.11 a/b/g/n/ac/ax	20–100 m indoors	1 Mbps – several Gbps	Mains-powered devices where infrastructure exists
Bluetooth Low Energy	Bluetooth SIG	10–100 m	~125 kbps – 2 Mbps	Wearables, beacons, phone-connected devices
Zigbee, Thread	IEEE 802.15.4	10–100 m per hop; mesh extends	250 kbps	Home and building automation meshes
LoRaWAN	LoRa Alliance	2–15 km	0.3 – 50 kbps	Wide-area very low-power sensing, unlicensed spectrum
NB-IoT / LTE-M	3GPP	Cellular coverage	~30 kbps – 1 Mbps	Licensed-spectrum wide-area sensing, metering
Cellular 4G/5G	3GPP	Cellular coverage	Mbps – Gbps	Video, vehicles, high-bandwidth mobile assets
RFID / NFC	ISO/IEC 18000, 14443	cm to a few m	Low	Identification, ticketing, access control, inventory

IMPORTANT — The fundamental trade-off

Range, data rate and power cannot all be optimised at once. LoRaWAN achieves kilometres at milliwatts precisely *because* it sends only a few hundred bits per second; Wi-Fi achieves megabits *because* it spends hundreds of milliwatts. When asked to select a technology, identify which two of the three the application actually needs and accept the loss on the third.

Worked selections. *500 water meters across a town, ten-year battery, hourly reading* → **LoRaWAN or NB-IoT**: range large, data trivial, power severe. *A mains-powered smart bulb responding in a fraction of a second* → **Wi-Fi, or Thread/Zigbee with a hub**. *A perimeter camera streaming 1080p* → **Ethernet if cabling is possible, else Wi-Fi or 4G/5G**; LoRaWAN is off by four orders of magnitude and is not a candidate.

(b) Network (Internet) layer. IPv4's 32-bit address space is exhausted and cannot give every thing a globally routable address; IPv6's 128-bit addresses can, and its stateless autoconfiguration supports the self-configuration characteristic. **6LoWPAN** (*IPv6 over Low-power Wireless Personal Area Networks*) is an **adaptation layer** providing (i) **header compression**, exploiting the predictability of much of the IPv6 header within a local network, and (ii) **fragmentation and reassembly** of larger IPv6 datagrams across multiple frames. **RPL** (*Routing Protocol for Low-power and Lossy Networks*) builds a **Destination-Oriented Directed Acyclic Graph (DODAG)** rooted at a border router, giving each node a preferred parent for upward forwarding.

NOTE — Why not simply run ordinary IPv6 on sensor nodes?

Because the numbers do not fit. An IEEE 802.15.4 frame carries at most **127 bytes**, while the IPv6 header alone is **40 bytes** and IPv6 requires a minimum MTU of **1280 bytes**. Adding an 8-byte UDP header consumes 48 of the 127 bytes before any application data or link-layer security. 6LoWPAN's compression can reduce that IPv6 header to a handful of bytes where addresses are derivable from link-layer information. This is a classic examination question, and the answer is *frame size*.

(c) Transport layer. TCP is connection-oriented and reliable, at the cost of handshake overhead, per-connection state, and retransmission behaviour that copes poorly with long, lossy links. UDP is connectionless with negligible state, the natural choice for small, frequent, individually non-critical readings, with reliability added by the application protocol above. DTLS provides TLS-equivalent security over UDP and is the standard means of securing CoAP.

(d) Application layer. This is where IoT departs most visibly from the conventional web.

HTTP: request-response, text-based, over TCP. Universally understood and trivially integrated with web infrastructure, but verbose and a poor fit for the most constrained devices.

CoAP (Constrained Application Protocol, RFC 7252), designed by the IETF for constrained devices. It deliberately mirrors HTTP's REST model (the same GET, POST, PUT, DELETE on URI-identified resources) so gateway translation is straightforward, but is **binary**, has a compact 4-byte fixed header, and runs over **UDP**. It offers optional reliability through *confirmable* messages, and an **observe** option letting a client register interest in a resource and receive change notifications, effectively adding publish-subscribe behaviour to a request-response protocol. It suits **device-to-device** interaction on constrained networks.

MQTT (Message Queuing Telemetry Transport): a lightweight **publish-subscribe** protocol over TCP. Clients publish to a hierarchical **topic** (for example plant1/lineB/motor3/temperature) on a central **broker**; subscribers receive matching messages, never addressing publishers directly and never needing to be online at the same time. It defines three **quality-of-service** levels: QoS 0 ("at

most once"), QoS 1 ("at least once", acknowledged, may duplicate) and QoS 2 ("exactly once", a four-step handshake); plus **retained messages** (the broker keeps the last message on a topic for new subscribers) and a **last will and testament** (published on the client's behalf if it disconnects unexpectedly). Both are direct answers to intermittent connectivity. MQTT suits **device-to-cloud telemetry at scale**.

AMQP: a heavier enterprise messaging protocol with queues, exchanges, sophisticated routing and transactional guarantees, used at the *back end* between platform and enterprise applications rather than on constrained devices. **WebSocket** provides a full-duplex persistent connection over a single TCP connection, upgraded from HTTP, suiting live dashboards and interactive control. You should also recognise **XMPP** (XML-based, presence and messaging patterns), **DDS** (broker-less peer-to-peer publish-subscribe with rich real-time QoS, used in defence, robotics and industrial control), and the industrial protocols **Modbus** and **OPC UA** (Section 6.7).

Table 6.5 — Comparison of IoT application-layer protocols

Feature	HTTP	CoAP	MQTT	AMQP
Interaction model	Request-response	Request-response (+ observe)	Publish-subscribe	Publish-subscribe, queues
Transport	TCP	UDP	TCP	TCP
Message encoding	Text	Binary	Binary	Binary
Header size	Hundreds of bytes typically	4 bytes fixed	2 bytes minimum	Larger
Reliability	Provided by TCP	Optional (confirmable messages)	Three QoS levels (0/1/2)	Strong, with transactions
Security	TLS (HTTPS)	DTLS	TLS	TLS/SASL
Broker required	No	No	Yes	Yes
One-to-many delivery	Poor	Multicast possible	Native	Native
Suits constrained devices	Poorly	Very well	Well	Poorly
Typical placement	Device-to-web, management APIs	Device-to-device on constrained networks	Device-to-cloud telemetry	Backend/enterprise integration

IMPORTANT — How to answer "Which protocol would you use?"

Never answer with a name alone. Name the deciding constraint first, then the protocol: *"The devices are battery powered behind a lossy link, and the same reading is needed by three applications. Publish-subscribe decouples producers from consumers, and MQTT's retained messages and last-will cope with intermittent connectivity, so MQTT over TLS is appropriate."* An answer structured this way demonstrates understanding; a bare protocol name does not.

CHECK YOUR PROGRESS 6.3

1. Draw and label the generic block diagram of an IoT device.
2. Distinguish between accuracy and precision. Which can be corrected by calibration?
3. Why is 6LoWPAN necessary? State the specific numerical mismatch it resolves.
4. State the three MQTT quality-of-service levels and the practical difference between QoS 1 and QoS 2.
5. A farmer must monitor soil moisture at fifty points across a 40-hectare farm, on five-year batteries, reporting twice daily. Recommend a link-layer technology and an application-layer protocol, justifying each.

6.4 LOGICAL DESIGN OF IoT

The **logical design** of an IoT system is an abstract representation of its entities, responsibilities and interactions, expressed *without* committing to particular hardware or vendors. Its value is that it survives implementation change: "the temperature entity publishes readings that any interested service may consume" remains true whether the sensor is a DS18B20 on Zigbee or a BMP280 on Wi-Fi. Logical design is studied under four headings: functional blocks (Section 6.5), communication models, communication APIs, and IoT levels.

6.4.1 IoT communication models

Four models are standard, each defined by the entities involved and the coupling between them.

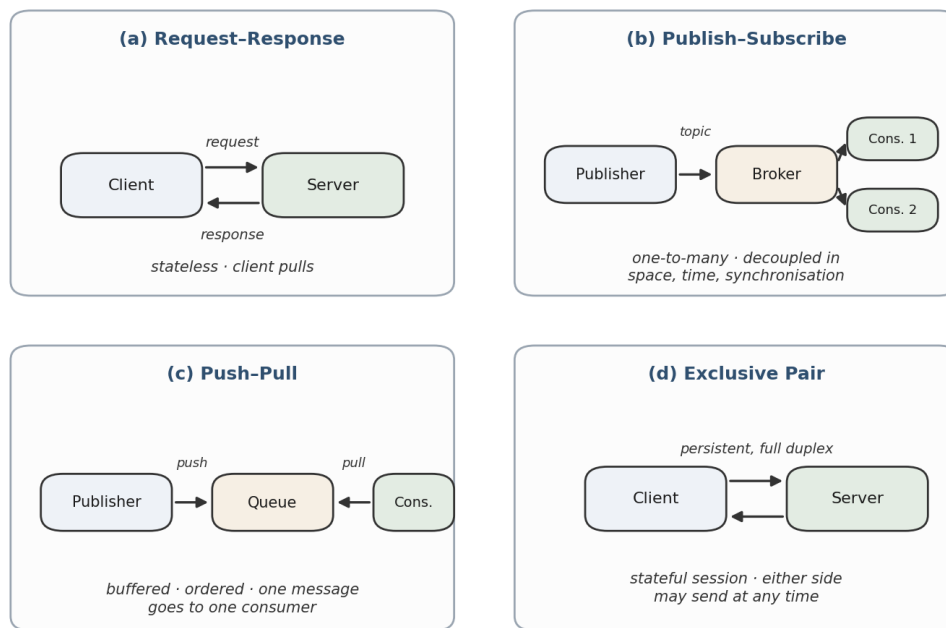


Figure 6.4 — The four IoT communication models

(a) Request-Response. A **client** requests a resource; a **server** retrieves or computes it and responds. The client *pulls* data at a moment of its own choosing. The key property is **statelessness**: each request carries all information needed to service it, so any server instance can handle any request, which is what allows the model to scale. It fits configuration reads and writes, on-demand queries and management operations, and is implemented by HTTP and CoAP. Its limitation is continuous monitoring: an application needing a reading every second must *poll* every second, and most responses return unchanged data, wasting bandwidth and battery.

(b) Publish-Subscribe. A **publisher** sends messages labelled with a **topic** to a **broker**; **consumers** register interest in topics and receive matching messages. Publishers and consumers never know of each other. The model decouples them in three distinct senses (naming all three earns marks): **space** (neither needs the other's address), **time** (they need not be connected simultaneously, as the broker can hold messages) and **synchronisation** (neither blocks waiting for the other). It fits one-to-many telemetry distribution (one sensor reading consumed simultaneously by a dashboard, an alerting service and an analytics pipeline), and MQTT is the canonical implementation. Its limitation is that the broker is a central component that must be scaled and made highly available.

(c) Push-Pull. Publishers *push* messages into a **queue**; consumers *pull* them when ready, each message normally going to exactly one consumer and then being removed. The queue absorbs rate

mismatch: if sensors generate faster than processing can absorb, the queue grows rather than data being lost. It fits ingestion pipelines distributing work across a pool of workers.

(d) Exclusive Pair. Client and server establish a persistent, **full-duplex** connection open for the session; either may send at any time without being asked. Unlike request–response it is **stateful**. It fits interactive control and live monitoring, and WebSocket is the standard implementation. Because the server holds an open connection and state per client, it does not scale to very large device populations as gracefully as publish-subscribe.

Table 6.6 — Comparison of the four IoT communication models

Aspect	Request–Response	Publish–Subscribe	Push–Pull	Exclusive Pair
Entities	Client, server	Publisher, broker, consumer	Publisher, queue, consumer	Client, server
Initiated by	Client	Publisher	Publisher	Either party
Connection	Typically transient	Persistent to broker	Persistent to queue	Persistent, full duplex
Delivery	One-to-one	One-to-many (all subscribers)	One-to-one (one consumer per message)	One-to-one
Coupling	Tight	Loose in space, time, synchronisation	Loose; buffered	Tight, session-based
State	Stateless	Broker holds subscriptions	Queue holds messages	Stateful session
Message ordering	Not applicable	Not guaranteed across topics	Generally preserved	Preserved
Typical protocol	HTTP, CoAP	MQTT	AMQP, message queues	WebSocket
Best used for	On-demand queries, configuration	Telemetry distribution at scale	Load levelling, work distribution	Interactive real-time control

IMPORTANT — Publish–subscribe versus push–pull

This distinction is examined frequently. In **publish–subscribe** a message reaches *every* interested subscriber: a broadcast to interested parties. In **push–pull** it reaches *one* consumer, which removes it: work distribution. Queues also generally preserve message **order**, and consumers control the rate at which they receive work.

THINK ABOUT IT — Think About It

A smart building has 2,000 occupancy sensors, and three systems need their data: the lighting controller (immediately), the HVAC optimiser (every five minutes) and a monthly utilisation report. Which model serves this best, and why is request–response poor here? *The answer turns on one-to-many delivery: with publish–subscribe each sensor publishes once and all three consumers receive it independently; with request–response each system polls all 2,000 sensors separately: 6,000 polling relationships instead of 2,000 publications.*

6.4.2 IoT communication APIs

REST (REpresentational State Transfer) is not a protocol but an **architectural style**: a set of constraints which, if obeyed, yield scalability, simplicity and evolvability. It was formulated by Roy Fielding in 2000 and underlies the modern web. Everything of interest is a **resource** identified by a URI, manipulated through a small fixed set of methods.

Table 6.7 — REST methods and their properties

Method	Meaning	Safe?	Idempotent?
GET	Retrieve the representation of a resource	Yes	Yes
POST	Create a subordinate resource, or submit data for processing	No	No
PUT	Create or completely replace a resource at a known URI	No	Yes
DELETE	Remove a resource	No	Yes

NOTE — Safe and idempotent

A method is **safe** if it does not modify server state (GET). It is **idempotent** if performing it several times has the same effect as once (PUT, DELETE, but not POST). This matters acutely in IoT: on a lossy network a request may be retried after a lost acknowledgement, so idempotent design ensures an accidental repeat does not, for instance, dispense a dose of medicine twice.

The six architectural constraints of REST, which should be reproduced accurately:

1. **Client-server**: separation of concerns between the client (interface, request initiation) and the server (storage, business logic), each free to evolve independently.
2. **Statelessness**: every request contains all information needed to service it; the server keeps no client session context between requests. This improves scalability at the cost of larger requests.
3. **Cacheability**: responses declare themselves cacheable or not, allowing clients and intermediaries to reuse them. Significant in IoT, where a sensor value may remain valid for many seconds.
4. **Layered system**: a client cannot tell whether it is connected to the end server or to an intermediary, which lets gateways translate protocols (CoAP to HTTP) transparently.
5. **Uniform interface**: all resources are accessed through the same small standard operation set and identified by URIs. This is what gives REST its generality.
6. **Code on demand (optional)**: the server may extend client functionality by transferring executable code. The only optional constraint, and rarely used in IoT.

EXAMPLE — A RESTful IoT interface

GET /api/v1/buildings/3/rooms/204/temperature → {"value": 24.6, "unit": "C", "ts": "2026-09-06T10:15:00Z"}

PUT /api/v1/buildings/3/rooms/204/setpoint with body {"value": 23.0} → sets the target temperature

DELETE /api/v1/buildings/3/rooms/204/alerts/17 → removes alert rule 17

Note how the URI expresses a *hierarchy of resources* and the *method* (not the URI) expresses the *action*. A design using /getTemperature?room=204 and /setTemperature?room=204&v=23 is not RESTful, because it encodes the verb in the resource name and abandons the uniform interface.

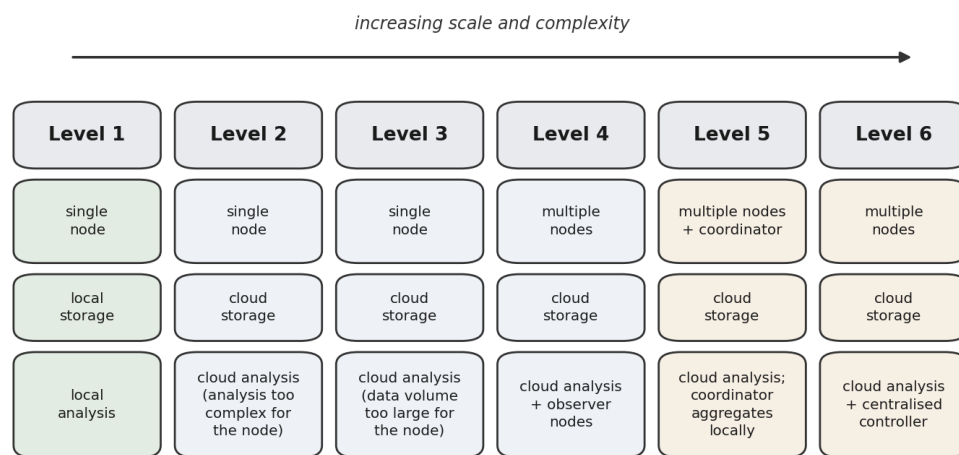
A **WebSocket-based API** implements the exclusive-pair model: the client sends an HTTP request with an Upgrade: websocket header and, if accepted, the connection becomes persistent and full-duplex. This matters in IoT because with a REST API a server needing to push a command must wait for the device to poll (costing energy and latency), whereas over a WebSocket the command is delivered the instant it is issued. The costs are held connection state per device and the loss of native one-to-many distribution.

Table 6.8 — REST-based versus WebSocket-based communication APIs

Aspect	REST-based API	WebSocket-based API
Underlying model	Request–response	Exclusive pair
Connection	Transient; new request per interaction	Persistent, full duplex
State	Stateless	Stateful
Server-initiated messages	Not possible without polling	Native
Overhead per message	HTTP headers on every request	Small frame header after handshake
Scalability to many devices	Very high	Limited by open connections held
Best suited to	Configuration, queries, web integration	Live dashboards, real-time control

6.4.3 IoT levels

An **IoT level** classifies a complete deployment by where its data resides and where analysis is performed. The taxonomy converts the vague question "how big is this system?" into a specific one: *how many nodes, where does the data live, and where does the analysis run?*



Levels 1–3: one node. Levels 4–6: many nodes. The distinguishing feature of Level 6 is centralised control of the device population.

Figure 6.5 — The six IoT deployment levels

Table 6.9 — The six IoT levels

Level	Nodes	Data stored	Analysis	Distinguishing feature and example
1	Single	Local	Local	Entirely self-contained: a standalone smart light sensing ambient brightness and switching itself on
2	Single	Cloud	Cloud	Analysis too heavy for the node: a soil-moisture node whose readings run against cloud crop models
3	Single	Cloud	Cloud	Data volume is the driver: a turbine vibration sensor sampling at kilohertz rates for spectral analysis
4	Multiple	Cloud	Cloud	Multiple independent nodes and observer nodes: a city noise-and-air-quality network
5	Multiple + coordinator	Cloud	Cloud	Coordinator aggregates and pre-processes locally: forest-fire detection clusters relaying via one mast
6	Multiple	Cloud	Cloud	A centralised controller commands the whole population: city-wide smart street lighting

IMPORTANT — Distinguishing Level 2 from Level 3

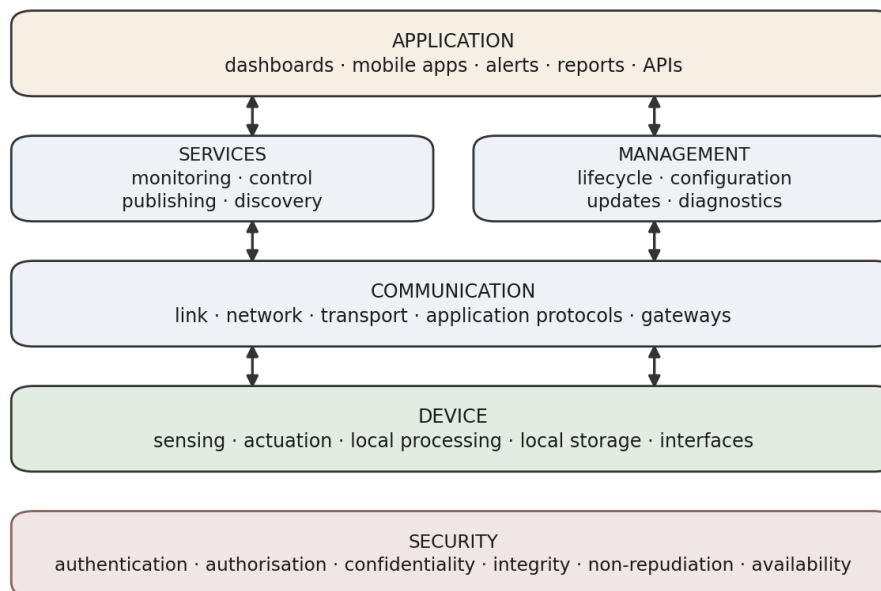
Students routinely confuse these; both are single-node, cloud-analysed systems. The difference is *why* the cloud is used. In **Level 2** the driver is **computational complexity**: the node could store the data but cannot analyse it. In **Level 3** the driver is **data volume**: the node cannot even store what it produces. State the driver explicitly in an examination answer.

CHECK YOUR PROGRESS 6.4

1. State the difference between the physical and the logical design of an IoT system.
2. Name the four communication models and the entities involved in each.
3. Explain the three senses in which publish–subscribe decouples publishers from consumers.
4. List the six architectural constraints of REST. Which one is optional?
5. Classify by IoT level, with a one-line justification: (a) a home weather station uploading readings for cloud forecasting; (b) a national network of 50,000 smart meters with central load-shedding control; (c) a fitness band counting steps and displaying them on its own screen.

6.5 FUNCTIONAL BLOCKS OF IoT

A **functional block** groups related capabilities that an IoT system must provide. The functional-block view answers "*what must this system be able to do?*" without prescribing how or where those capabilities are implemented. It is therefore an architectural checklist: a design missing any one block has a gap that will surface later as a defect.



Security is not a layer added at the end: it applies to every block above it.

Figure 6.6 — The six functional blocks of an IoT system

Device. Interaction with the physical world: the things themselves, with their sensors, actuators, embedded processing, local storage and physical interfaces. Everything the system knows about the world enters here, and everything it does to the world leaves here.

Communication. Moving data between devices, gateways, servers and applications: the entire protocol stack of Section 6.3.4, plus gateways translating between constrained local protocols and Internet protocols, and the addressing and routing that let a message find its destination.

Services. The capabilities offered over devices and data, in four families: **device monitoring** (state, health, availability), **device control** (commands to actuators, configuration changes), **data publishing** (making data available to applications, with transformation, aggregation and enrichment) and **device discovery** (letting applications learn what devices exist, what they can do and where they are).

Management. Governing the system over its operational lifetime. Management is distinct from services: services act on the *world*, management acts on the *system*. It covers device lifecycle (onboarding, commissioning, decommissioning), configuration, firmware and software updates, fault detection and diagnostics, performance monitoring, and accounting. In a deployment of a hundred thousand devices, this block determines whether the system remains operable at all.

Security. Protecting the system, its data and its users through **authentication** (establishing identity), **authorisation** (determining permissions), **confidentiality** (protecting data in transit via TLS/DTLS and at rest via encryption), **integrity** (detecting alteration), **non-repudiation** (preventing denial of having sent a message) and **availability** (resisting denial of service).

Application. Delivering value to the end user through dashboards, mobile applications, alerting channels, reports and APIs. However sophisticated the infrastructure, this is the only part most users will ever see, and a deployment whose data is never presented in an actionable form delivers nothing.

NOTE — Two different lists, both called "functional blocks"

Some textbooks give the six blocks above; others give Sensor/Actuator, Connectivity, Data Processing, Application, Security and Management. The lists overlap almost entirely: "Sensor/Actuator" is Device, "Connectivity" is Communication, and "Data Processing" is the analytical part of Services. Follow your prescribed textbook's list if one is specified; what matters is that you can explain the *responsibilities*, not that you memorise particular labels.

IMPORTANT — Why security is not a layer added at the end

The assets requiring protection exist in *every* block: keys and secure boot in the device block, encrypted channels in communication, access control in services, secure update in management. A single perimeter cannot protect a device against firmware tampering, nor protect data at rest, nor stop an authenticated but unauthorised user issuing a command.

EXAMPLE — Case study — overhead water tank monitoring for a housing society

Four tanks; residents complain of dry taps and the operator sometimes runs the pump dry.

Device: an ultrasonic level sensor in each tank lid measuring distance to the water surface, with a temperature sensor for compensation (the speed of sound varies with temperature); at the pump house, a current sensor detecting pump draw and a relay to switch it. Nodes buffer 24 hours of readings if the network fails.

Communication: tank nodes over Wi-Fi to a gateway, forwarded to the cloud by MQTT over TLS; the pump controller on wired Ethernet, because reliability of the control path matters more than installation convenience.

Services: level per tank and pump state; start/stop and threshold setting; a daily consumption figure published to the society's billing sheet; automatic discovery of the four tanks by the mobile app.

Management: over-the-air firmware twice a year; a fault raised if a node is silent for thirty minutes or reports a physically implausible level (indicating a fouled sensor); backup-battery monitoring.

Security: a unique certificate per node issued at commissioning; only the secretary and the pump operator may issue a pump command, and every command is logged with the issuer's identity; all residents may view levels.

Application: live gauges, a push notification below 20%, an operator start/stop control, and a monthly consumption report.

Note the interlock: a float switch wired directly into the pump contactor cuts power if the sump runs dry, regardless of what the software commands, applying the principle from Section 6.3.3.

Organising a system this way yields four practical benefits, each a standard short-answer question.

Scalability: blocks under load scale independently: doubling the device count grows communication and services, but the application block may not change. **Interoperability:** block boundaries become interfaces, so a new vendor's device is admitted provided it satisfies the device–communication interface. **Modularity:** each block can be implemented, tested, replaced or upgraded independently; migrating cloud platforms should leave the device block untouched. **Deployment flexibility:** the same logical architecture runs on-premises, in a public cloud, or hybrid, with latency-sensitive processing at the edge and long-term analytics centrally.

CHECK YOUR PROGRESS 6.5

1. Name the six functional blocks and state each responsibility in one line.
2. Distinguish the services block from the management block, with one example capability of each.
3. List the six components of the security block.
4. Explain why security cannot be treated as a single layer added at the end of a design.
5. For a smart parking system in a shopping mall, identify briefly what belongs in each of the six blocks.

6.6 HOME AUTOMATION

6.6.1 What home automation is

Home automation (the *smart home*) applies IoT within a residence to monitor and control domestic systems automatically, remotely, or in response to sensed conditions, improving convenience, safety, security, energy efficiency and accessibility.

The distinction between automation and mere remote control is the substance of the topic. **Remote control** means a human issues a command from a distance: a phone application switching a light. **Automation** means the system issues the command itself from sensed conditions and rules: a light switching on because motion was detected *and* ambient light was below a threshold *and* the time is after sunset. The value of a smart home rises sharply as it moves from the first to the second, because the human effort per action falls to zero.

NOTE — Why accessibility matters more than convenience

Home automation is often presented as a luxury. For elderly residents and people with mobility or sensory impairments it is frequently the opposite: voice-controlled lighting, automated locks, fall detection and medication reminders can be the difference between independent living and dependence on a carer. Evaluate a design against this use case, not only against convenience.

6.6.2 Architecture and components

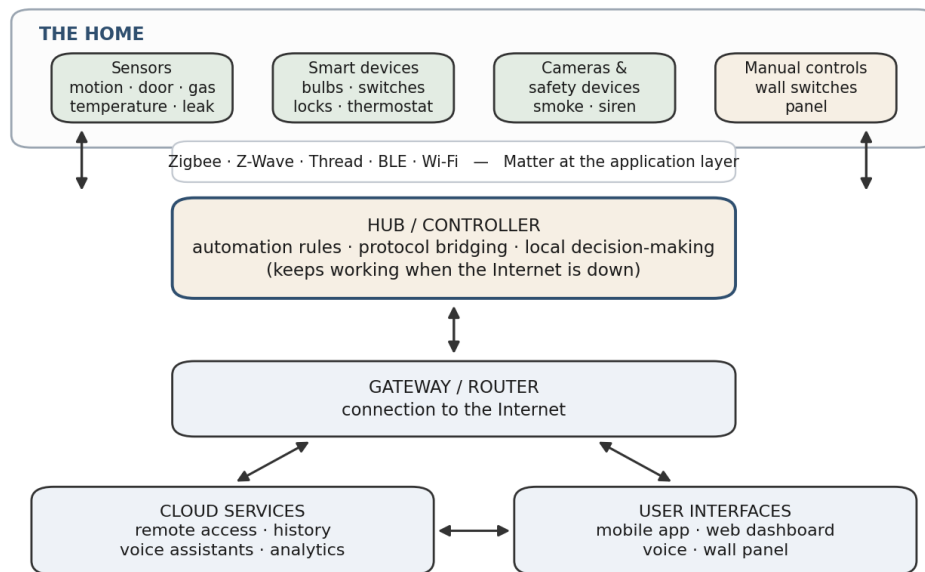


Figure 6.7 — Architecture of a home automation system

A home automation system decomposes into three parts, all of which must be right.

Hardware: the sensors, actuators, controllers, hubs and gateways physically installed. Hardware fixes what the system can sense and do, and is the most expensive part to change once cabling and in-wall devices are installed.

Software, of three distinct kinds that students often conflate: **device firmware**, which reads sensors, drives actuators, speaks the protocol and applies local logic (and must support over-the-air update, since a smart lock that cannot be patched is a permanent vulnerability); **hub or controller software**, which holds the automation rules, coordinates devices from different vendors, and keeps working when the Internet is down; and the **user-facing application**: mobile app, wall panel or voice assistant.

Communication protocols: because a home accumulates devices from many vendors over many years, protocol choice determines whether the home remains a coherent system or a collection of unrelated apps.

The components deployed are **sensors** (motion/PIR, door and window contacts, temperature, humidity, smoke, gas, water leak, light level, camera), **actuators and smart devices** (bulbs, switches, dimmers, plugs, motorised blinds, locks, valves, thermostats, sirens), a **controller/hub** running the rules and bridging protocols, a **gateway** to the Internet (often in the hub or router), **cloud services** for remote access, history and voice, a **middleware or integration layer** reconciling protocols, and **user interfaces**.

IMPORTANT — Why local control matters

A well-designed system must keep working when the Internet fails. If "turn on the light when motion is detected" is evaluated in a distant cloud, a broadband outage leaves the resident in a dark corridor. Systems evaluating rules on a **local hub** degrade gracefully: remote access is lost, but the home still works. This is the general IoT principle (place time-critical or safety-relevant logic as close to the physical process as possible), and it is the same question posed in Section 6.1.3.

Table 6.10 — Comparison of home automation communication protocols

Protocol	Band	Topology	Power	Notable characteristics
Wi-Fi	2.4 / 5 GHz	Star, via router	High	No hub needed; high bandwidth; unsuitable for battery devices; router congestion at scale
Bluetooth LE	2.4 GHz	Star, mesh	Very low	Direct phone connection; short range; suits wearables and locks
Zigbee	2.4 GHz (also sub-GHz)	Mesh	Low	802.15.4 based; self-healing mesh; mature ecosystem; needs a hub
Z-Wave	Sub-GHz (region specific)	Mesh	Low	Less Wi-Fi interference, better wall penetration; strong interoperability certification
Thread	2.4 GHz	Mesh	Low	802.15.4 based but IPv6-native via 6LoWPAN; no single point of failure
Matter	Over Wi-Fi, Thread, Ethernet	Application layer	Depends on transport	Not a radio: an application-layer interoperability standard

NOTE — Matter is not a competitor to Zigbee or Wi-Fi

Matter, developed under the Connectivity Standards Alliance, solves the *interoperability* problem: before it, a bulb from one vendor and a hub from another frequently could not work together. It defines a common data model and commissioning process and runs *on top of* existing transports: Wi-Fi and Ethernet for mains-powered devices, **Thread** for low-power battery devices, with BLE used during commissioning. Its 1.5 revision (November 2025) added cameras using WebRTC, a modular treatment of closures such as blinds and garage doors, soil sensors, and expanded energy management including tariff information and bidirectional EV charging.

Understand the layering: **Thread is the network; Matter is the language**. They are complementary, not alternatives.

6.6.3 Applications

Lighting: occupancy- and daylight-based switching, dimming, scheduling and scenes; typically automated first, because the payback in convenience and energy is immediate. **Climate control:** learning thermostats, zone conditioning, and integration with window sensors so an open window does not cool the street. **Security and access:** smart locks with per-user credentials and audit logs, contact and glass-break sensors, video doorbells, alarm integration. **Safety monitoring:** smoke, carbon monoxide and LPG detection, and water-leak sensors under sinks and washing machines; these have the clearest life-safety justification. **Energy management:** per-circuit and per-appliance metering, load shifting to off-peak tariffs, and integration of rooftop solar and battery storage. **Water management:** tank level monitoring, pump control with dry-run protection, leak detection, and irrigation driven by soil moisture and weather. **Appliance automation, entertainment and ambience:** cycle-completion reporting, remote preheating, multi-room audio, automated blinds and scene control. **Assistive applications:** voice control, fall detection, medication reminders, and activity monitoring that alerts a relative when a lone resident's daily pattern changes unexpectedly.

EXAMPLE — Case study — automating a three-bedroom apartment

A working couple with an elderly parent wants occupancy-responsive lights, gas and smoke alerting, assurance the geyser is not left running, notification if the door opens while nobody is home, and simple voice control for the parent, whose eyesight is poor.

Sensing: PIR occupancy in corridor and living room; ambient light; contacts on main and balcony doors; an LPG detector near the hob; a corridor smoke detector; a current sensor on the geyser circuit.

Actuation: smart switch modules behind the existing wall switches, a smart plug on the geyser, a siren, and a solenoid on the gas line.

Protocols: battery sensors on **Thread**, mains switches on **Wi-Fi**, with a **Matter**-capable hub so both families interoperate and future devices can be added.

Rules, evaluated on the hub: R1: corridor motion AND light < 50 lux → corridor light on for 3 minutes. R2: gas detected → close solenoid, sound siren, notify all residents. R3: geyser current > 0 for 45 minutes → switch off and notify. R4: main door opens while mode = "away" → capture a snapshot and notify.

Three deliberate choices. Rules run *on the hub*, so the flat works during a broadband outage. The switch modules sit *behind* existing switches, so a visitor who knows nothing about the system can still use the home. The gas solenoid is paired with a manual valve a resident can operate directly.

6.6.4 Benefits and limitations

The benefits are convenience, measurable energy and water savings, earlier detection of gas, smoke and leaks, security with audit trails, accessibility for elderly and disabled residents, and remote assurance. Against these: **interoperability** gaps remain despite Matter, and households accumulate incompatible ecosystems; **security and privacy** exposure is physical as well as informational, since occupancy data reveals when a house is empty and cameras and voice assistants operate in intimate spaces; **dependence on connectivity and vendors** can render cloud-tied devices inoperable if a manufacturer discontinues the service, a phenomenon common enough to be named *cloud abandonment*; **cost and complexity** of installation and fault diagnosis exceed most residents' skills; and **reliability expectations** are severe: a light switch is expected to work every time for twenty years, a standard far above that applied to a phone application.

CHECK YOUR PROGRESS 6.6

1. Distinguish remote control from automation in a smart home, with an example of each.
2. Name the three parts of a home automation system and the three distinct kinds of software involved.
3. Explain the relationship between Thread and Matter. Why is it wrong to call them competitors?
4. Why should safety-critical rules be evaluated on a local hub rather than in the cloud?
5. State three limitations of home automation, one relating to security or privacy.

6.7 INDUSTRIAL IoT AND INDUSTRY APPLICATIONS

6.7.1 What Industrial IoT is, and how it differs

The **Industrial Internet of Things (IIoT)** applies IoT to industrial assets and processes (machinery, plant, infrastructure, vehicles and supply chains) to increase efficiency, reduce unplanned downtime, improve quality and improve safety. It is not simply "IoT in a factory": the operating environment imposes requirements consumer IoT never faces.

Table 6.11 — Consumer IoT compared with Industrial IoT

Dimension	Consumer IoT	Industrial IoT
Consequence of failure	Inconvenience	Production loss, equipment damage, injury, environmental harm
Reliability requirement	Best effort	High availability, often measured in nines of uptime
Timing requirement	Rarely critical	Frequently hard real-time, deterministic latency

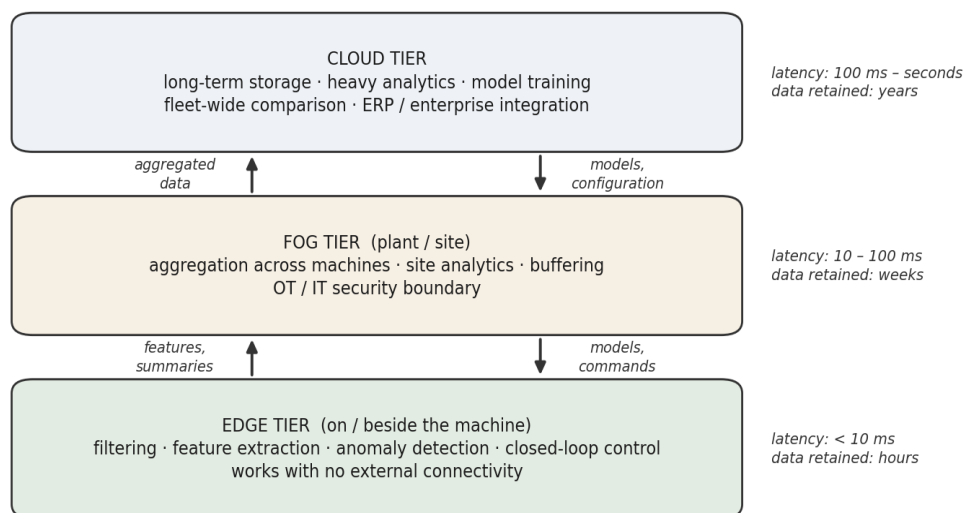
Dimension	Consumer IoT	Industrial IoT
Equipment lifetime	2–5 years	15–30 years; must integrate with legacy equipment
Environment	Benign (homes, offices)	Heat, vibration, dust, moisture, chemicals, electromagnetic noise
Data scale	Modest	Very large: high-rate vibration, current and vision data
Security consequence	Data loss, privacy breach	Physical damage, safety incident, infrastructure risk
Standards	Matter, Bluetooth	OPC UA, Modbus, PROFINET, TSN
Operated by	The consumer	Operations technology (OT) staff, with practices distinct from IT

NOTE — IT and OT convergence

Industrial plants have run **operational technology (OT)** (PLCs, SCADA, industrial buses) for decades in networks deliberately isolated from corporate **IT**. IIoT requires the two to be joined so plant data reaches analytics platforms, and this is the central organisational challenge: OT prioritises *availability and safety* above all, IT prioritises *confidentiality and rapid patching*. A patch IT would apply immediately may be unacceptable to OT if it requires stopping a furnace. Successful projects manage this explicitly, through a segmented architecture with a controlled boundary.

6.7.2 The edge–fog–cloud architecture

Because industrial data is voluminous and industrial decisions are often time-critical, processing is distributed across three tiers rather than sent wholesale to a distant cloud.



Going up: data volume decreases, latency increases, and independence from external connectivity decreases.

Figure 6.8 — The edge–fog–cloud architecture of an IIoT deployment

The **edge tier** computes on or immediately beside the machine (on the sensor node, the PLC, or a small computer in the same cabinet) with latency in milliseconds, performing filtering, feature extraction, anomaly detection and closed-loop control. Its defining property is that it keeps working when the network to the outside does not. The **fog tier** sits at plant or site level, aggregating data from many machines, running site-wide analytics, buffering for the cloud, and enforcing the OT/IT security boundary. (*Edge* and *fog* are used loosely and sometimes interchangeably; where a distinction is drawn, edge means on or beside the device and fog means at the site level.) The **cloud tier** provides centralised storage and heavy computation: long-horizon analysis, model training, fleet-wide comparison across plants, and integration with enterprise systems.

IMPORTANT — Why edge processing is not optional in IIoT

Three independent arguments converge, and a good answer gives all three. **(i) Latency:** a loop that must respond within 10 ms cannot wait for a cloud round trip. **(ii) Bandwidth and cost:** a vibration sensor sampling at 20 kHz produces far more data per day than is economic to transmit, so features must be extracted locally and kilobytes sent instead of gigabytes. **(iii) Availability:** a plant must keep running when its Internet link fails, so any function whose loss would stop production must not depend on a remote service.

6.7.3 Why industry adopts IIoT

Continuous visibility into machine state lets bottlenecks be found and **Overall Equipment Effectiveness (OEE)** (the standard metric combining availability, performance and quality) be measured from actual machine data rather than estimated from manual logs. **Reduced unplanned downtime** is usually the single largest financial benefit, since an unplanned line stoppage costs lost output, restart costs and missed deliveries. **Cost savings** accrue from lower maintenance expenditure, extended asset life, lower energy use, less scrap and smaller spares inventory, set against substantial initial expenditure on sensors, connectivity, platforms and, often the largest item, integration and change management. **Safety** improves through continuous monitoring of gases, temperatures, structural loads and worker location, and through remote operation of hazardous tasks. **Quality** improves because in-line measurement during production, rather than sampling inspection afterwards, allows a drifting process to be corrected within minutes. Finally, connected products enable **new business models: servitisation**, in which a compressor manufacturer sells compressed air per cubic metre rather than compressors, taking on maintenance risk because it can now see and predict machine condition.

6.7.4 Predictive maintenance: the flagship application

Table 6.12 — Comparison of maintenance strategies

Strategy	Basis for intervention	Weakness
Reactive (run to failure)	Repair after breakdown	Maximum downtime; secondary damage; unsafe
Preventive (scheduled)	Service at fixed intervals of time or usage	Servicing healthy machines unnecessarily; still misses early failures
Predictive (condition-based)	Intervene when measured condition indicates approaching failure	Requires sensing, data history and analysis capability

Predictive maintenance on a motor-driven pump proceeds in six steps. **(1) Instrument:** accelerometers measure vibration in three axes, a thermocouple or infrared sensor measures bearing temperature, a current transformer measures motor current. **(2) Extract features at the edge:** the raw vibration signal, sampled at kilohertz rates, is transformed (typically by an FFT) into a frequency spectrum, because specific fault modes produce energy at characteristic frequencies: bearing defects at frequencies set by the bearing geometry, imbalance at the rotational frequency, misalignment at twice it. Only these features, not the raw waveform, are transmitted. **(3) Baseline:** the healthy signature is recorded over a period of normal operation. **(4) Detect deviation:** an anomaly model or engineering thresholds flag departure from the baseline. **(5) Estimate remaining useful life** where enough failure history exists. **(6) Act:** a work order is raised, scheduling repair at the next planned break with the correct spare already ordered.

EXAMPLE — Quantifying the case

A packaging line stops for eight hours when a gearbox fails, and the plant loses ₹4,00,000 of production per hour. Two unwarned failures a year cost ₹64,00,000. Condition monitoring costs ₹3,50,000 to install and ₹80,000 a year to run, and detects deterioration early enough to convert

both events into planned two-hour interventions during an existing shift change, at effectively zero production loss. *Annual benefit* $\approx$ ₹63,20,000; *payback* well under a month.

Read this critically. It assumes monitoring detects *every* failure with sufficient warning and produces no false alarms; in practice detection is imperfect and false alarms have their own cost. The example illustrates the *structure* of the business case (avoided downtime versus monitoring cost), not a guaranteed outcome. State your assumptions explicitly, as has been done here.

6.7.5 Applications by sector

Table 6.13 — Industrial IoT applications by sector

Sector	Representative applications
Manufacturing	Real-time machine monitoring giving live OEE and identifying the line constraint; predictive maintenance on motors, pumps, compressors and robots; in-line vision and dimensional inspection of every unit rather than a sample; digital twins for simulating process changes without disturbing production; RFID and BLE tracking of tools, work-in-progress and finished goods
Agriculture	Precision farming from soil sensors (moisture, temperature, pH, nutrients) combined with weather stations and drone imagery; automated irrigation driven by soil moisture; livestock collars monitoring location, activity, rumination and temperature to detect illness early; grain-silo and cold-store monitoring to cut post-harvest loss
Energy and utilities	Smart grids giving real-time visibility of load and power quality, enabling demand response, faster fault localisation and reduced technical and commercial losses; instrumented wind turbines and solar arrays reporting generation, soiling and component condition; transformer and transmission-line condition monitoring
Transport and logistics	Fleet telematics reporting position, fuel, harsh braking and diagnostic codes for route optimisation and vehicle predictive maintenance; smart warehousing with RFID/BLE and automated guided vehicles; cargo tracking combining GNSS with temperature, humidity, shock and tilt sensing for chain-of-custody evidence
Healthcare	Central monitoring of imaging equipment, ventilators and infusion pumps across a hospital group; predictive maintenance minimising unavailability of critical devices; continuous environmental logging in pharmacies, laboratories, theatres and the vaccine cold chain with audit-quality records; location tracking of mobile equipment
Mining	Remotely operated haul trucks, drills and loaders removing workers from hazardous locations; real-time tracking of vehicles, equipment and personnel for productivity and emergency response; continuous methane, carbon monoxide, air-flow, dust and ground-movement monitoring with automatic evacuation alerting

NOTE — What a digital twin is

A **digital twin** is a continuously updated virtual model of a physical asset or process, fed by live sensor data. Unlike a conventional simulation model, which represents a design in the abstract, a twin tracks the *specific individual asset in its current actual state*, so a proposed change can be tested against how that machine is behaving today, not how machines of its type behave in general. It is also the principal meeting point between IIoT and Virtual Reality (Section 6.10.5).

6.7.6 Challenges in adoption

Legacy integration is often the largest part of a project: equipment with a decades-long service life predates connectivity and exposes at best a serial Modbus interface, and retrofitting sensors and gateways without invalidating safety certification is difficult. **Interoperability** is addressed by **OPC UA** (Open Platform Communications Unified Architecture), the principal vendor-neutral industrial standard, which supplies not merely transport but a rich information model describing what the data *means*, a significant advance over protocols like Modbus that transmit bare register

values. **Security** requires network segmentation, strict boundary control, device authentication, monitoring for anomalous control traffic, and a patching regime compatible with production constraints, because connecting control systems designed for isolated networks expands the attack surface against targets where an attack has physical consequences. **Determinism** is supplied by **Time-Sensitive Networking (TSN)**, a set of IEEE 802.1 amendments adding bounded-latency behaviour to standard Ethernet so control and IT traffic can share one network safely. **Data quality and skills** matter because analytics is worthless if sensors are poorly sited, uncalibrated or mislabelled, and because staff who understand both process engineering and data engineering remain scarce. Finally, **justifying the investment** stalls many projects after a successful pilot: those that succeed usually begin with a single measurable problem, one line, one failure mode, rather than an attempt to instrument everything at once.

CHECK YOUR PROGRESS 6.7

1. Give four points of difference between consumer IoT and Industrial IoT.
2. Explain the edge, fog and cloud tiers and state what determines which tier a computation belongs in.
3. Give three independent reasons why processing must occur at the edge in an industrial deployment.
4. Distinguish preventive from predictive maintenance, and outline the six implementation steps on a rotating machine.
5. What is a digital twin, and what does it enable that a conventional simulation model does not?
6. Name three IIoT adoption challenges and one concrete measure addressing each.

6.8 SURVEILLANCE APPLICATIONS OF IoT

6.8.1 From CCTV to IoT surveillance

Surveillance is the systematic observation of people, places, assets or environmental conditions for security, safety, compliance or operational awareness. **IoT-based surveillance** replaces the closed, human-watched systems of the past with networked, sensor-rich systems in which analysis is performed largely by software.

Table 6.14 — Traditional CCTV compared with IoT-based surveillance

Aspect	Traditional CCTV	IoT-based surveillance
Camera	Analog, dumb; sends a video signal	Networked, with on-board processing
Transmission	Coaxial cable to a local recorder	IP network, wired or wireless
Storage	Local digital video recorder	Local, edge and/or cloud; policy-driven retention
Analysis	A human watching a monitor, or reviewing afterwards	Automated analytics, largely at the edge, in real time
Sensing	Video only	Video, audio, motion, thermal, environmental, access control, fused
Response	Human decides and acts	Automated alerting and actuation (lock, siren, light)
Scalability	Limited by human attention	Limited by processing capacity, not attention

IMPORTANT — The attention argument

The strongest technical argument for automated surveillance is a human-factors one. Studies of vigilance tasks consistently show that a person monitoring multiple feeds for rare events loses a substantial proportion of their detection ability within tens of minutes. A hundred cameras

cannot be watched effectively by any realistic number of operators. Automated analytics does not tire, and its role is not to replace human judgement but to *direct human attention* to the small fraction of footage that warrants it.

6.8.2 Architecture

The **sensing tier** is multi-modal. Visible-light cameras, fixed or pan-tilt-zoom, are the most prominent, but **thermal and infrared cameras** detect heat rather than light and therefore work in complete darkness, through smoke and through light foliage; **motion sensors** (PIR, microwave, or dual-technology units requiring both to trigger) reduce false alarms; **acoustic sensors** detect glass breakage or abnormal machine noise; **access-control readers** supply the identity dimension cameras alone cannot; **environmental sensors** extend surveillance from security to safety; and **structural sensors** (strain gauges, tiltmeters, accelerometers) monitor bridges, dams and buildings.

The **edge processing tier** is what makes modern surveillance distinctive. A camera or small appliance runs computer-vision models locally to perform motion and intrusion detection with defined zones and tripwires; **object detection and classification**, distinguishing a person from a vehicle from an animal, which eliminates the great majority of false alarms produced by naive motion detection; **Automatic Number Plate Recognition (ANPR)**; loitering, abandoned-object and crowd-density detection; personal-protective-equipment compliance checking; and, where lawful, face recognition, the most privacy-sensitive capability and the most heavily regulated.

NOTE — Why analytics moved to the edge

Three reasons, recognisable from Section 6.7.2. **Bandwidth**: a single 1080p stream at 25 fps consumes roughly 2–4 Mbps, so a hundred cameras streamed continuously need a link most organisations do not have. **Latency**: an alert that must trigger a door lock cannot wait for a cloud round trip. **Privacy**: transmitting the event "*person detected at gate 3 at 14:22*" rather than continuous video means far less personal data leaves the premises, a pattern known as *privacy by design*, and increasingly a legal requirement.

The **communication tier** favours wired Ethernet with Power over Ethernet for fixed cameras, since one cable carries both data and power, with wireless used where cabling is impractical at some cost in reliability. The **storage and platform tier** runs a Video Management System that records, indexes, enforces retention and provides search; retention policy is a genuine design decision with legal weight, since keeping footage longer than necessary increases both cost and exposure. The **response tier** covers control-room alerts, mobile notifications, automated actuation and integration with incident management.

6.8.3 Application domains

Public safety and smart cities: city camera networks integrated with traffic management and emergency response; automated violation detection; crowd-density monitoring at large gatherings to prevent crush incidents. **Critical-infrastructure protection**: perimeter intrusion detection at substations, refineries and water-treatment plants, where a breach has consequences far beyond theft. **Retail**: shrinkage reduction, queue-length monitoring and footfall analysis, though the last is a marketing application wearing the clothes of security and should be recognised as such. **Transport**: station and airport monitoring, ANPR for tolling and parking, driver-fatigue monitoring in commercial fleets, track-side monitoring. **Healthcare and elder care**: ward monitoring, fall detection in care homes, restricted-area monitoring; camera-free alternatives such as radar or thermal sensors that detect a fall without capturing an identifiable image are increasingly preferred precisely because they are less invasive. **Environmental and structural surveillance**: forest-fire

detection, flood monitoring, structural health monitoring of bridges and dams, pollution and illegal-dumping monitoring. **Agricultural and wildlife surveillance:** camera traps for census and anti-poaching, and crop-intrusion monitoring.

EXAMPLE — Case study — perimeter surveillance for a substation

A 220 kV substation suffers repeated copper theft. Guards cannot patrol 1.2 km continuously, and the existing cameras generate so many false alarms (stray dogs, moving vegetation, insects on the lens, headlight glare) that operators have stopped responding.

Design. Eight fixed thermal cameras give overlapping fence-line coverage, with four PTZ visible-light cameras for verification and evidence, a fibre-optic vibration sensor along the fence, and switchable lighting. Each camera group is served by an edge appliance running a person/vehicle classifier. An alarm is raised only when the fence vibration sensor *and* the thermal camera classify a human-sized heat signature in the same zone within a five-second window: a **sensor-fusion rule** that suppresses the dominant false-alarm sources, since a dog does not produce a human-sized signature and wind on the fence produces no thermal signature at all. On alarm, the nearest PTZ slews to the zone, floodlights switch on, a voice announcement plays, and a verified clip reaches the control room and the on-call supervisor. The site uplink carries only alarm clips and metadata, never continuous video. Retention is seven days at low resolution and ninety days for alarm clips.

What made the difference was not better cameras but better *fusion and classification*. The original system failed for a human-factors reason (alarm fatigue), and the fix raised the precision of alarms rather than their number.

6.8.4 Privacy, ethics and the legal dimension

Surveillance is the IoT application where the tension between capability and rights is sharpest. The same system that detects an intruder records every employee's arrival time, every visitor's face, and the movements of everyone in range; capability proportionate for one purpose is disproportionate for another. Responsible deployments follow seven principles: **purpose limitation** (define before deployment what the system is for, and do not repurpose safety data into performance management without a fresh basis); **data minimisation** (collect and retain the least that achieves the purpose: prefer an event to a video stream, a radar fall-detector to a camera); **retention limits** (indefinite retention is a liability, not an asset); **transparency** (notify people that an area is under surveillance, and by whom); **access control and audit** (restrict who may view footage and log every access, since most real-world misuse is by insiders rather than intruders); **security of the data itself** (unsecured cameras are among the most commonly compromised IoT devices, and large botnets have been assembled from cameras left on default credentials); and **proportionality** (face recognition in a workplace canteen is unlikely to be proportionate; ANPR at a secure site gate probably is).

IMPORTANT — The Indian legal context

In India the **Digital Personal Data Protection Act, 2023 (DPDP Act)** governs the processing of digital personal data. Surveillance footage identifying individuals is personal data, so its processing attracts obligations on lawful basis, notice, purpose limitation, security safeguards and breach reporting, with sector-specific rules also applying in banking, telecommunications and healthcare. For any real deployment obtain current legal advice: statutes and rules change, and this unit states the position in general terms only, as engineering-relevant orientation rather than as legal advice.

THINK ABOUT IT — Think About It

A hospital proposes cameras with face recognition in its wards, to detect when a patient at risk of falling gets out of bed unattended. The clinical benefit is real. But could the same benefit come from a pressure mat under the mattress, or a ceiling radar sensor that detects posture and records no image? If it could, is the camera-based design defensible? This is *data minimisation* applied to a concrete decision, and the reasoning that distinguishes a system designer from an integrator.

The practical limitations are **false alarms**, which remain the limit on usefulness since a system that cries wolf is switched off by its own operators; **environmental dependence**, as visible-light cameras degrade in fog, rain, glare and darkness while thermal cameras lose contrast when ambient temperature approaches body temperature; **algorithmic bias**, since face and person recognition have demonstrated materially different error rates across demographic groups, which is a serious defect wherever a false identification has consequences for an individual; **cost of bandwidth and storage** at scale, the reason edge processing became standard; and **security of the surveillance system itself**, which if compromised hands an attacker exactly the reconnaissance capability the system was installed to prevent.

CHECK YOUR PROGRESS 6.8

1. Give four points of difference between traditional CCTV and IoT-based surveillance.
2. State three reasons why video analytics is performed at the edge rather than in the cloud.
3. What is sensor fusion, and how did it reduce false alarms in the substation case study?
4. List five principles a responsible surveillance deployment should follow.
5. Explain data minimisation and give an example of a design choice applying it.

6.9 OTHER IoT APPLICATION DOMAINS

For each domain below, note the pattern: *what is sensed, what is decided, and what changes as a result*. That triad is the analytical structure to apply to any new application you meet.

Smart cities apply IoT to urban infrastructure and services. **Smart street lighting** dims and brightens with ambient light and presence, and is typically a city's first project because the energy saving alone funds it. **Smart parking** guides drivers to free bays, reducing the circling traffic that is a significant share of central-city congestion. **Waste management** plans collection routes by bin fill level rather than by fixed schedule. **Traffic management** adapts signal timing to real-time flow and gives priority to emergency vehicles. **Water distribution** localises leaks from pressure and flow sensing, critical where non-revenue water losses are high. **Air-quality monitoring** supplements a few reference-grade stations with distributed low-cost sensors, giving spatial resolution reference stations alone cannot.

Healthcare and the Internet of Medical Things (IoMT) covers connected devices used clinically: **remote patient monitoring** of heart rate, blood pressure, glucose, oxygen saturation and ECG at home, letting chronic conditions be managed without admission and deterioration be caught earlier; **wearables**, where the boundary between consumer wellness and regulated medical device is a live regulatory issue; **smart implants and connected drug delivery**, such as an insulin pump closing a loop with a continuous glucose monitor; **medication adherence** through instrumented dispensers; and the hospital operations of Section 6.7.5.

IMPORTANT — Safety-critical constraints in IoMT

Medical IoT carries constraints beyond ordinary IoT: regulatory approval of the device, clinical validation of any algorithm influencing treatment, fail-safe behaviour when connectivity is lost, and stringent protection of health data. An insulin pump must have a safe autonomous mode; it

cannot simply stop working because a network is unavailable. This class of constraint is what distinguishes a postgraduate answer on IoMT from a list of gadgets.

Wearables are IoT devices worn on the body: fitness bands, watches, smart glasses, hearables, connected clothing, medical patches. Their distinguishing constraints are extreme limits on size, weight and battery, skin-safe materials, and the sensitivity of continuous personal data. Most use BLE to a smartphone, which acts as gateway to the Internet, the gateway pattern of Section 6.5 at personal-area scale, adopted because a direct Wi-Fi or cellular radio would consume far more energy than a wearable battery can supply.

Retail and supply chain uses item-level RFID for inventory visibility, cold-chain monitoring for food and pharmaceuticals, smart shelves detecting low or misplaced stock, contactless checkout fusing weight, vision and RFID, and end-to-end shipment visibility joining container tracking with condition and customs status.

Smart energy extends beyond the grid applications of Section 6.7.5 to **demand-side management**: appliances deferring operation to periods of low tariff or high renewable generation, rooftop solar and home batteries coordinated as a **virtual power plant** (an aggregation of distributed resources dispatched as though it were a single generating unit), and electric-vehicle charging scheduled to avoid stressing local distribution, with bidirectional vehicle-to-grid charging turning parked cars into distributed storage.

Environment and disaster management covers flood monitoring with automated warning dissemination, forest-fire detection, landslide monitoring, and **earthquake early warning**, where dense accelerometer networks detect the fast P-wave and issue a warning seconds before the destructive S-wave: enough time for automated actions such as stopping trains, closing gas valves and halting lifts at the nearest floor, even though it is far too little for human evacuation.

Connected and autonomous vehicles are among the most complex IoT endpoints. **Vehicle-to-everything (V2X)** communication spans vehicle-to-vehicle, vehicle-to-infrastructure, vehicle-to-pedestrian and vehicle-to-network links, supporting collision warning, cooperative adaptive cruise control, signal-phase advisory and emergency-vehicle priority: applications with latency requirements in tens of milliseconds and safety consequences that place them firmly in cyber-physical-systems territory.

Table 6.15 — Summary of principal IoT application domains

Domain	Typically sensed	Typical connectivity	Principal outcome
Smart city	Light, occupancy, flow, fill level, air quality	LoRaWAN, NB-IoT, city Wi-Fi	Lower energy and operating cost; better services
Healthcare / IoMT	Physiological signals	BLE via phone, Wi-Fi	Earlier detection; care outside hospital
Wearables	Motion, heart rate, temperature	BLE	Personal health and activity insight
Retail / supply chain	Identity, location, temperature	RFID, BLE, cellular	Inventory accuracy; product integrity
Energy	Current, voltage, generation, tariff	Cellular, PLC, Wi-Fi	Load shifting; renewable integration
Environment	Water level, seismic, gas, particulates	LoRaWAN, satellite, cellular	Early warning; regulatory monitoring
Vehicles / V2X	Position, speed, proximity, vehicle state	Cellular V2X, DSRC	Collision avoidance; traffic efficiency

CHECK YOUR PROGRESS 6.9

1. Name any five smart-city applications, stating for each what is sensed and what changes.

2. What is IoMT? State two constraints applying to medical IoT devices but not to ordinary consumer devices.
3. Why do most wearables use BLE to a smartphone rather than connecting to the Internet directly?
4. What is a virtual power plant, and which IoT capability makes it possible?
5. Explain how earthquake early warning is useful even with only a few seconds of notice.

6.10 INTRODUCTION TO VIRTUAL REALITY

6.10.1 Definition and characterising properties

IMPORTANT — Definition — Virtual Reality

Virtual Reality (VR) is the use of computer modelling and simulation to generate a three-dimensional, computer-synthesised environment with which a user can interact in real time, in such a way that the user's sensory input is substantially replaced by the simulated environment, producing a subjective sense of *being present* within it.

Two clauses carry the meaning. "**Sensory input is substantially replaced**" distinguishes VR from a three-dimensional game on a monitor: in VR, turning your head changes what you see as it would in the real world, because the display moves with your head and occupies your whole field of view, so your visual system receives evidence consistent with being in the virtual place. "**Interact in real time**" means the environment responds with imperceptible delay: a pre-rendered 360° video is *immersive* but not *interactive*, and is therefore only partially VR.

A widely used framework, due to Grigore Burdea, characterises VR by three properties. **Immersion** is the *objective, technical* degree to which the system's displays surround and engage the senses (field of view, resolution, refresh rate, tracking accuracy, spatial audio, haptics), and is in principle measurable. **Interaction** is the extent and naturalness of the user's ability to act upon and modify the environment; reaching out and grasping a virtual object with a tracked hand is more interactive, and more natural, than pressing a key. **Imagination** is the application of the technology to conceive and represent things that do not or cannot exist (a molecule the size of a room, a building not yet constructed, the interior of a working jet engine), emphasising that VR's value comes from what is *represented*, not only from how well it is displayed.

NOTE — Immersion and presence are not the same thing

Presence is subjective: the user's psychological sense of "being there", of responding to the virtual environment as though it were real: a user who ducks to avoid a virtual object experiences presence. High immersion makes presence more likely but does not guarantee it: excellent hardware with a poorly designed environment can fail to produce presence, while a simple environment with a compelling task can produce it on modest hardware. **Presence is broken** by tracking errors, latency, low frame rate, and by any event that reminds the user of their real surroundings.

Table 6.16 — Virtual, Augmented, Mixed and Extended Reality distinguished

Term	What the user sees	Relationship to the real world	Example
Virtual Reality (VR)	Entirely computer-generated environment	Real world replaced and excluded	Enclosed head-mounted display; flight simulator
Augmented Reality (AR)	Real world with computer-generated information overlaid	Real world dominates; the overlay adds information but does not interact with it	Navigation arrows on a phone camera view; heads-up display

Term	What the user sees	Relationship to the real world	Example
Mixed Reality (MR)	Real and virtual content coexisting, virtual objects <i>anchored to and aware of</i> real geometry	Virtual objects occlude behind, rest on, and are lit by real surfaces	A virtual screen fixed on a real wall as you walk around it
Extended Reality (XR)	—	Umbrella term covering VR, AR and MR collectively	Used in standards work such as OpenXR

IMPORTANT — AR versus MR — where the boundary lies

The discriminator is *spatial awareness*. In AR the overlay is drawn on top of the camera image; walk behind a real pillar and the virtual object remains drawn in front of it, because the system does not model the pillar. In MR the system has a geometric model of the room, so the pillar correctly **occludes** the virtual object. Occlusion, physical interaction with real surfaces, and consistent lighting are the practical tests. The **virtuality continuum** of Milgram and Kishino places these on a single axis from the entirely real to the entirely virtual environment, showing that they are points on a spectrum rather than four discrete categories.

6.10.2 Types of VR system

Non-immersive (desktop) VR renders a three-dimensional environment on an ordinary monitor, controlled by keyboard, mouse or joystick, leaving the user fully aware of the real environment. It is cheapest, needs no special hardware, causes no cybersickness and is easily shared with onlookers; it suits CAD, architectural walkthroughs and training where full immersion is unnecessary.

Semi-immersive VR uses large-screen or multi-projector systems filling much of the visual field without enclosing the user. The classic example is the **CAVE** (*Cave Automatic Virtual Environment*): a room whose walls, floor and sometimes ceiling are rear-projection screens, with the user's head tracked so perspective is rendered correctly for their viewpoint. Several people can share the experience and see their own bodies and colleagues, which matters for collaborative design review, and no headset is worn; against this, it is very expensive, fixed in place, and correct in perspective only for the one tracked user.

Fully immersive VR uses a head-mounted display (HMD) occluding the real world entirely, with head tracking, hand controllers or hand tracking, and spatial audio. It gives the highest sense of presence at relatively modest cost, but is isolating, susceptible to cybersickness, and limited in session duration by weight and heat.

Collaborative and networked VR places multiple users, each with their own system, in one shared environment represented to one another by avatars, for distributed design review, remote training and social applications. Here the engineering problem shifts from rendering to *state synchronisation* across a network with latency.

6.10.3 Components of a VR system

A VR system is best understood not as a collection of parts but as a **closed loop** that must complete many times per second: the user moves → the movement is sensed → the scene is recomputed from the new viewpoint → the new image is displayed → the user perceives it and moves again. Everything in the architecture exists to make that loop fast and accurate.

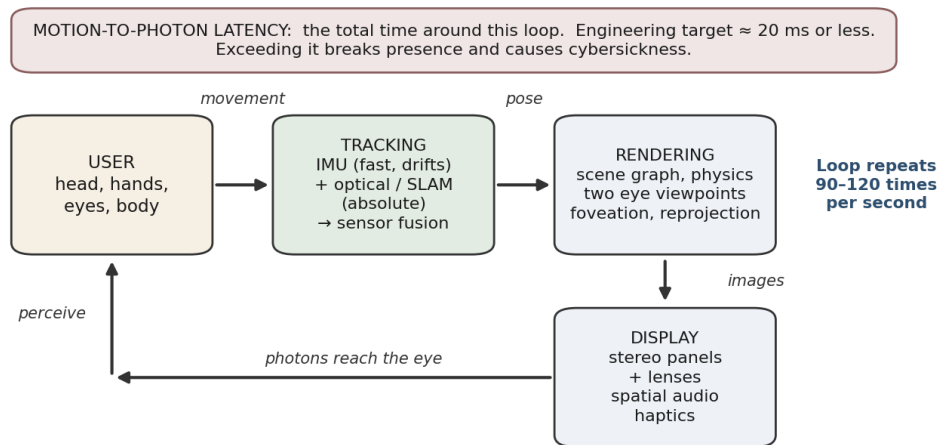


Figure 6.9 — Architecture of a VR system: the tracking-rendering-display loop

Input and tracking. **Degrees of freedom (DoF)** is the key term: **3DoF** tracks orientation only (pitch, yaw, roll), so the user can look around but not move; **6DoF** adds position along three axes, letting the user lean, crouch and walk. 6DoF is required for genuine presence, because the human visual system expects **motion parallax** (nearby objects shifting more than distant ones as the head moves). **Outside-in tracking** uses external base stations fixed in the room: accurate, but requiring installation and confining the user to the tracked volume. **Inside-out tracking** uses cameras on the headset to observe the environment and compute its own pose, typically using **SLAM** (Simultaneous Localisation and Mapping); it needs no external hardware and now dominates consumer devices. MEMS accelerometers, gyroscopes and magnetometers in an **inertial measurement unit (IMU)** give very high-rate orientation data but **drift**, while optical tracking is slower but absolute, so **sensor fusion** uses the IMU for responsiveness and the optical system to correct drift, the same pattern seen in the surveillance case study of Section 6.8.3. Other input includes tracked hand controllers, camera-based hand tracking, data gloves, eye tracking, body tracking, voice, and locomotion devices such as omnidirectional treadmills.

Output. The **visual display** is two small panels, one per eye, viewed through lenses, presenting a **stereoscopic** pair: two slightly different images from which the visual system derives depth. **Audio** is **spatialised**, filtered by a head-related transfer function so sounds appear to originate from points in space; it contributes far more to presence than most students expect and is comparatively cheap to implement well. **Haptics** provides feedback to touch, from controller vibration through force feedback that resists motion. **Motion platforms** in high-end simulators apply real accelerations to the body, resolving the sensory conflict that otherwise causes sickness.

Processing and rendering. The engine maintains the scene graph, computes the geometry visible from each eye, applies lighting and shading, and outputs two images per frame, which is why VR is computationally expensive. Two optimisations are worth knowing. **Foveated rendering** exploits the fact that visual acuity is high only in a small central region: if eye tracking reveals where the user is looking, the periphery can be rendered at much lower resolution without perceptible loss. **Reprojection (timewarp)** geometrically warps the previous frame according to the latest tracking data when a new frame cannot be rendered in time, preserving responsiveness when the frame rate falters: a critical comfort mechanism.

Software and standards. **Unity** and **Unreal Engine** dominate application development. **OpenXR**, an open royalty-free Khronos Group standard, defines a common API between applications and XR devices; before it, an application had to be ported separately to each vendor's SDK, and version 1.1 (2024) consolidated widely adopted extensions into the core. This is the same interoperability problem, and the same kind of solution, that Matter provides in the smart home. **WebXR** delivers

VR and AR through a web page without installing an application, and **glTF** and **USD** exchange scene and model data between tools.

6.10.4 Performance parameters and human factors

Table 6.17 — Key performance parameters of a VR system

Parameter	Meaning	Why it matters
Field of view (FOV)	Angular extent of the visible scene, typically 90–120° horizontal	Wider FOV increases immersion; human binocular FOV is roughly 200°, so all current devices are restricted
Resolution per eye	Pixels presented to each eye	Determines sharpness; pixels per degree (PPD) is more meaningful than raw pixel count because it accounts for FOV
Refresh rate	Frames per second, typically 90–120 Hz	Below about 90 Hz motion judders and discomfort increases sharply
Motion-to-photon latency	Time from head movement to the corresponding photons reaching the eye	The single most important comfort parameter; the cited engineering target is $\leq \sim 20$ ms
Tracking accuracy and jitter	Error and instability in reported pose	Errors break presence and, if the world appears to move when the head is still, cause sickness
Weight and ergonomics	Mass and balance of the headset	Determines how long a session can comfortably last
IPD adjustment	Matching lens spacing to the user's eyes	Mismatch causes eye strain and blurred stereo

IMPORTANT — Cybersickness — cause and mitigation

Cybersickness is nausea, disorientation, eye strain and headache experienced by some VR users. The principal explanation is **sensory conflict**: the visual system reports self-motion while the vestibular system in the inner ear reports that the body is stationary, and the brain's response to contradictory evidence about self-motion includes nausea. **Latency compounds it**: if the image lags head movement the world appears to swim, producing a second and more acute conflict, which is why the ~ 20 ms budget is a hard engineering constraint rather than a target. Mitigations, in approximate order of effectiveness: keep latency low and frame rate high and *stable*; prefer **teleportation** over smooth artificial locomotion; when smooth movement is unavoidable, reduce the field of view briefly during motion (**vignetting**); avoid accelerations, especially rotational ones, the user did not initiate; provide a stable visual reference such as a cockpit; and keep early sessions short. A complete answer names sensory-conflict theory, identifies latency as an aggravating factor, and lists at least three mitigations.

DID YOU KNOW? — How current hardware compares

A mainstream standalone headset such as the Meta Quest 3 offers roughly 2064×2208 pixels per eye on LCD panels, 90–120 Hz, about 110° horizontal FOV, on a mobile-class processor, at around 515 g and a consumer price. A premium device such as the Apple Vision Pro uses micro-OLED at roughly 3660×3200 per eye with far higher pixel density (approximately 34 pixels per degree against roughly 18) at 90–96 Hz, 600–650 g, and several times the cost. The comparison illustrates the fundamental trade-off: **pixel density, field of view, weight, battery life and cost cannot be optimised simultaneously**. Products change rapidly; the trade-off structure does not, and it is the trade-off that is examinable.

6.10.5 Applications of VR

Education and training is VR's most substantiated application, and its advantages are specific: it allows practice of tasks that are *dangerous* (firefighting, working at height, hazardous materials),

expensive (aircraft, cranes, process plant), *rare* (emergency procedures, low-frequency faults) or *irreversible* (surgery). A trainee can fail safely and repeat immediately.

Healthcare and therapy uses VR for surgical planning and rehearsal on patient-specific anatomy reconstructed from CT or MRI; **exposure therapy** for phobias and post-traumatic stress, in which the patient meets a controlled, graduated version of the feared stimulus; **distraction analgesia**, where immersion measurably reduces pain during burn dressing changes; and motor rehabilitation after stroke, where VR gives engaging, repeatable, precisely measured exercise.

Engineering, design and architecture allows full-scale review before anything is built: walking through a building, sitting in a vehicle interior, checking whether a technician can physically reach a component. Errors found here cost orders of magnitude less than errors found during construction, and collaborative review lets separated teams inspect one model together.

Manufacturing and the digital twin is where VR meets IIoT: an operator walks through a virtual representation of a running plant with live sensor data mapped onto the corresponding virtual equipment (seeing, for instance, a bearing glowing red because its measured temperature is elevated), and rehearses process changes before applying them to the real line.

Defence and aerospace uses flight, vehicle and mission simulators, maintenance training on equipment too scarce to take out of service, and mission rehearsal in accurate terrain reconstructions. **Retail** offers virtual showrooms and try-on; **tourism and heritage** allows virtual site visits and reconstruction of destroyed or inaccessible sites; **entertainment** is commercially the largest segment and funds the hardware development on which all other applications depend; and **remote collaboration** provides shared workspaces around a three-dimensional model, of particular value where the subject is inherently spatial.

EXAMPLE — Case study — VR training for substation maintenance

A utility must train technicians to isolate and earth a 220 kV feeder. Live training is impossible: the feeder cannot be de-energised for teaching, and a mistake on a real installation is unacceptable.

A 6DoF application reconstructs the substation to scale from engineering drawings. The trainee performs the full procedure (verifying the single-line diagram, operating the isolator in sequence, testing for dead, applying earths, fitting locks and tags), with the system enforcing real constraints (a padlock must be fitted before the next step is permitted) and recording every action with a timestamp.

Why VR fits here: all four training advantages apply at once: the task is dangerous, the equipment cannot be taken out of service, the emergency variants are rare, and errors are irreversible. Assessment is objective, because the system records the exact sequence performed rather than relying on an instructor's observation.

Limitations that must be acknowledged: the simulation does not reproduce the weight of the earthing equipment, the stiffness of a corroded isolator handle, or the noise and heat of a real substation. VR here *supplements* supervised on-site training; a design claiming otherwise would be unsafe.

The **advantages** of VR are safe repeatable practice of dangerous tasks, access to environments that are inaccessible, destroyed, unbuilt or physically impossible, objective automatic measurement of performance, reduced prototyping cost and time, better spatial understanding at full scale, and remote collaboration on spatial problems. The **limitations** are cybersickness affecting a significant minority; the cost of high-end systems and, more significantly, of producing content, which is usually the dominant expense; **fidelity gaps**, since haptics remain crude and weight, texture, resistance and smell are largely unrepresented, so skills depending on these transfer poorly; ergonomic constraints of weight, heat, hygiene of shared headsets and prescription lenses;

isolation and physical safety, since an immersed user cannot see their surroundings and needs a cleared area; accessibility limits for users who cannot perceive stereo depth or who have balance disorders; and **evidence quality**, since learning gains are sometimes asserted more strongly than the evidence supports: transfer of training should be measured, not assumed.

CHECK YOUR PROGRESS 6.10

1. Define Virtual Reality and explain the three "I"s that characterise it.
2. Distinguish clearly between immersion and presence.
3. What is the difference between Augmented Reality and Mixed Reality? State the practical test.
4. Explain 3DoF and 6DoF tracking, and why 6DoF is necessary for presence.
5. What is cybersickness? Explain the sensory-conflict theory and give three mitigation techniques.
6. Why is motion-to-photon latency the most important comfort parameter, and what is the target value?
7. Name four VR application domains, giving for each one reason VR is preferable to conventional methods.

6.11 SMART SYSTEMS

6.11.1 What makes a system "smart"?

IMPORTANT — Definition — Smart system

A **smart system** is an engineered system that **senses** its environment or its own state, **processes** what it senses to reach a decision, and **acts** on that decision to change its environment or its own behaviour, autonomously, and adapting as conditions change.

The test is simple: *does the system change what it does in response to what it observes, without a human deciding each time?* A conventional wall switch is not smart: it does exactly what it is told. A remotely operated switch is not smart either: it still does exactly what it is told, merely from further away. **Connectivity alone is not intelligence.** A switch that turns a light on when it detects occupancy *and* darkness is minimally smart; a system that learns a household's routine and pre-cools the bedroom before the occupants usually retire is smarter still, because it *adapts* rather than applying a fixed rule.

NOTE — "Smart" is a spectrum, not a category

It is more useful to speak of *degrees of smartness*. **Level 0 (Controlled)**: does what it is told (a switch). **Level 1 (Monitored)**: reports its state (a logging meter). **Level 2 (Rule-based)**: acts on fixed conditions (IF temperature > 30 THEN start fan). **Level 3 (Adaptive)**: adjusts its own parameters from experience. **Level 4 (Predictive)**: anticipates future states and acts in advance. **Level 5 (Autonomous/optimising)**: pursues a goal, choosing among strategies and coordinating with other systems. Locating a system on this ladder produces a far better answer than a yes-or-no judgement.

6.11.2 The sense–think–act loop

Every smart system implements the same cycle, and this loop is the single most important idea in the section.

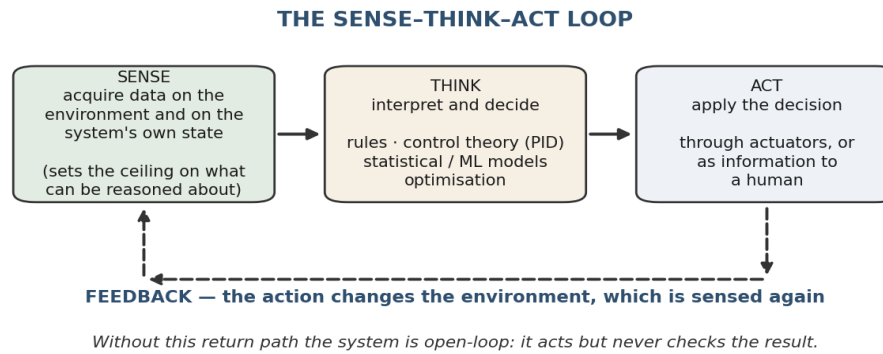


Figure 6.10 — The sense–think–act loop, with feedback

Sense acquires data about the environment and the system's own state. This stage sets the *ceiling* on how smart the system can be: it cannot reason about what it cannot observe. **Think** interprets and decides, and ranges enormously in sophistication: **rule-based reasoning** (explicit conditional logic, transparent and verifiable, which is why safety-critical functions almost always use it); **control theory**, most commonly the **PID (Proportional-Integral-Derivative) controller**, which computes a corrective action from present error, accumulated past error and rate of change of error, and remains the workhorse of industrial control; **statistical and machine-learning methods**, powerful but hard to guarantee on inputs unlike the training data; and **optimisation**, choosing the action that best satisfies an objective under constraints. **Act** applies the decision through actuators, or through information delivered to a human. **Feedback** closes the loop: the action changes the environment, which is sensed again. This is what distinguishes a *closed-loop* control system from an *open-loop* device: a washing machine running a fixed 40-minute cycle regardless of how clean the clothes become is open loop; one measuring water turbidity and continuing until the water runs clear is closed loop.

EXAMPLE — The loop at three levels of sophistication

The task: keep a room at 24 °C.

Rule-based (Level 2). IF temp > 24.5 THEN cooling on; IF temp < 23.5 THEN cooling off. Simple and reliable, but the temperature oscillates within a band and the compressor cycles frequently.

Control-theoretic (Level 3). A PID controller modulates compressor speed from the error, its integral and its derivative, holding temperature close to setpoint with far less oscillation and mechanical wear.

Predictive and optimising (Level 4–5). The system knows the afternoon will be hot, that a meeting begins at 15:00 and that the tariff rises at 14:00. It pre-cools before 14:00 on cheaper power, then coasts. Same objective, radically different behaviour: obtained not from better actuators but from more information and a better decision procedure.

The architecture of a smart system maps onto the IoT functional blocks of Section 6.5, which is unsurprising since most modern smart systems *are* IoT systems: a sensing layer, a data and communication layer, an **intelligence layer** holding the decision logic wherever it resides, an actuation layer with its interlocks, and an interaction layer, including the crucial ability for a human to *understand and override* what the system is doing. Four technologies have most enlarged what such systems can do: **machine learning**, letting decision rules be *learned* rather than written by hand, which is essential where no engineer can state the rule explicitly (recognising an incipient bearing failure from a vibration spectrum); **edge computing**, placing intelligence close to the process; **low-power embedded hardware**, including microcontrollers running compact neural

networks (a field known as **TinyML**), putting inference on a battery-powered device; and **digital twins**, allowing an action to be simulated before being applied.

Table 6.18 — Embedded systems, IoT systems and smart systems compared

	Embedded system	IoT system	Smart system
Defining feature	A dedicated computer inside a larger device	Things connected to a network, exchanging data	Autonomous sense–think–act behaviour
Must it be connected?	No	Yes, by definition	Not necessarily
Must it be intelligent?	No	Not necessarily	Yes, by definition
Typical example	A washing-machine controller	A connected water meter reporting reading	A thermostat that learns occupancy
Relationship	The <i>building block</i>	The <i>connectivity paradigm</i>	The <i>behavioural goal</i>

IMPORTANT — How the three ideas relate

The cleanest formulation, and a good examination answer: **embedded systems provide the capability, IoT provides the connectivity, and "smart" describes the resulting behaviour.** A device may be embedded but neither connected nor smart (a basic microwave timer); embedded and connected but not smart (a meter that only reports); or embedded and smart but not connected (a camera autofocus, which senses, decides and acts entirely on its own). The most capable modern systems are all three, which is why these topics appear together in this unit.

6.11.3 Examples, and the limits of autonomy

The **smart grid** senses load, voltage, frequency and generation across a network; decides how to dispatch generation, reconfigure feeders and shed load; and acts through breakers, tap changers and demand-response signals, exhibiting the highest levels of the ladder. **Smart traffic management** senses vehicle presence and queue lengths, decides signal phase and duration, and coordinates adjacent junctions to create progression along a corridor. **Smart irrigation** senses soil moisture at several depths together with weather and forecast, and its intelligence lies in *not* watering when rain is forecast, a decision a timer cannot make. **Advanced driver-assistance systems** sense the road scene through cameras, radar and lidar and act by braking or steering, in a setting where the "think" stage must be not merely accurate but *verifiably* so.

Four design considerations follow from autonomy. **Autonomy must be bounded:** a smart system acts without asking, so the consequences of a wrong action must be limited by physical interlocks, restrictions on permitted actions, and a clear always-available human override. **Explainability** matters wherever decisions materially affect people, which is a strong argument for rule-based logic in safety functions even where a learned model performs marginally better on average. **Graceful degradation** requires that when a sensor fails or connectivity is lost the system detects the failure and falls back to a defined safe behaviour; continuing to act confidently on a stuck sensor reading is among the commonest causes of real incidents. And **security** applies with full force, because a system that acts autonomously is a system an attacker can cause to act.

CHECK YOUR PROGRESS 6.11

1. Define a smart system and state the test distinguishing a smart device from a merely connected one.
2. Describe the sense–think–act loop and explain what feedback adds.
3. Distinguish open-loop from closed-loop control, with an example of each.

4. Complete and explain: "Embedded systems provide the ____, IoT provides the ____, and smart describes the ____."
5. Give three design considerations specific to autonomous smart systems, explaining why each matters.

6.12 EMBEDDED SYSTEMS

6.12.1 Definition, characteristics and classification

IMPORTANT — Definition — Embedded system

An **embedded system** is a computer system designed to perform one or a few dedicated functions, built into a larger mechanical or electrical device, usually operating under real-time constraints and with limited resources, and generally not perceived by its user as a computer at all.

Contrast this with a **general-purpose computer**, designed to run arbitrary programs chosen by its user. The distinction is not processing power (a modern embedded processor may outperform a general-purpose computer of a decade ago) but *purpose*: an embedded system's software is fixed at manufacture and defines what the product *is*.

Its defining characteristics are that it is **application-specific** (a microwave controller will never run a spreadsheet, and hardware and software are optimised for exactly that task); **tightly coupled to hardware**, interacting directly with pins, timers and converters with little operating-system abstraction; **real-time constrained**, so correctness depends not only on the right answer but on delivering it by the deadline; **resource-constrained** in memory, processing and often energy; **highly reliable over long unattended operation**, possibly running for years without restart in a physically inaccessible location; **cost- and power-sensitive**, since high volumes make a few rupees per unit significant and a milliampere of standby current may determine battery life; and **not user-programmable**: the user operates the product, they do not install software on it (manufacturer firmware *update* is a different matter, and is essential).

NOTE — Real-time does not mean fast

This is the most persistent misconception in the subject. A **real-time system** is one whose correctness depends on meeting timing deadlines, not one that is merely quick. A system that must respond within one second, and always does, is real-time. A system that usually responds in one microsecond but occasionally takes ten milliseconds is *not* real-time for a one-millisecond deadline, however fast it is on average. **Real-time means predictable, bounded response: determinism, not speed.**

Table 6.19 — Classification of embedded systems by functional requirement

Class	Description	Consequence of a missed deadline	Example
Stand-alone	Operates independently; no network dependence	—	Digital camera, microwave oven, calculator
Hard real-time	Deadlines are absolute	System failure, possibly catastrophic	Airbag deployment, anti-lock braking, pacemaker, flight control
Soft real-time	Deadlines matter but occasional misses are tolerable	Degraded quality of service	Video streaming buffer, router queueing, digital dashboard
Networked	Depends on a network connection to function	Loss of function	IP camera, smart meter, point-of-sale terminal

Class	Description	Consequence of a missed deadline	Example
Mobile	Portable and battery operated; size, weight and power dominate	—	Fitness band, portable medical monitor

By scale, **small-scale** systems use an 8- or 16-bit microcontroller with a few kilobytes of memory, programmed in C or assembly without an operating system; **medium-scale** systems use a 16- or 32-bit microcontroller or DSP, possibly with a real-time operating system; and **sophisticated** systems use multi-core 32- or 64-bit processors, sometimes with FPGAs or ASICs, running an embedded operating system such as Linux.

6.12.2 Hardware organisation

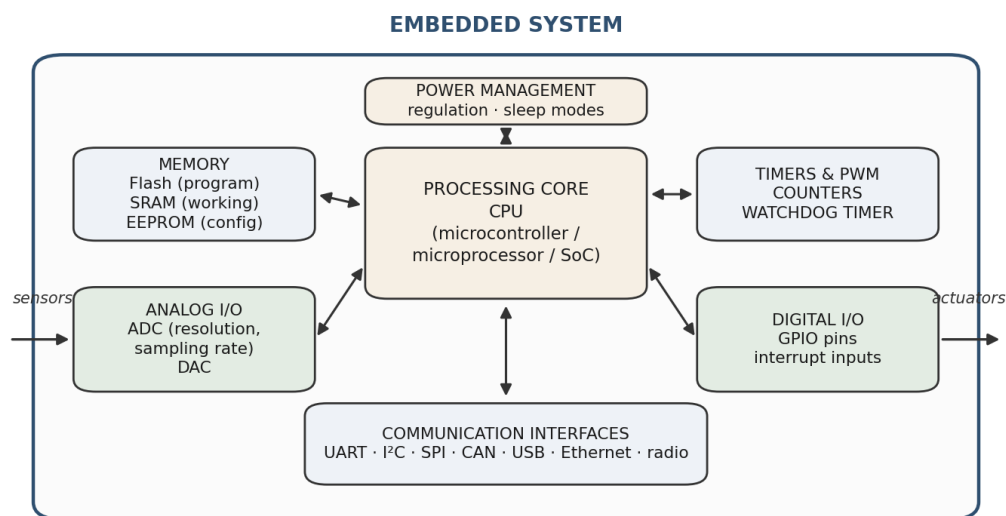


Figure 6.11 — Block diagram of an embedded system

The central choice is between a microprocessor and a microcontroller, and this distinction is examined in almost every paper on the subject.

Table 6.20 — Microprocessor compared with microcontroller

Aspect	Microprocessor	Microcontroller
Contents	CPU only	CPU + memory (RAM and flash) + peripherals on one chip
External components required	RAM, ROM, I/O, timers: all external	Few or none
Board complexity and size	Higher	Much lower
System cost	Higher	Lower
Power consumption	Higher	Lower; extensive sleep modes
Processing power	Higher	Lower
Flexibility	High: memory and peripherals chosen freely	Fixed at chip selection
Typical use	General-purpose computers, high-end embedded systems	Dedicated control applications
Examples	Intel Core series, ARM Cortex-A family	8051, AVR (ATmega), PIC, ARM Cortex-M, ESP32

A third option is the **System on Chip (SoC)**, integrating processor, graphics, memory controller, radio and peripherals on one die, the basis of smartphones and of capable IoT devices such as the

ESP32. A **Digital Signal Processor (DSP)** is architecturally specialised for signal-processing arithmetic, and an **FPGA** provides reconfigurable hardware for tasks needing true parallelism or deterministic timing beyond what software offers.

Memory comprises **ROM/Flash** for non-volatile program storage (reprogrammable, which is what makes firmware update possible), **RAM** for variables and stack (typically the scarcest resource, at tens of kilobytes on a microcontroller), **EEPROM** for configuration and calibration constants that must survive a power cycle, and external storage where data logging is required.

The **peripherals** are the interface to the physical world. **GPIO** provides individually controllable digital pins for switches, LEDs and relays. The **ADC** converts an analog sensor voltage into a number, characterised by **resolution** (bits, hence distinguishable levels: a 10-bit ADC resolves 1024) and **sampling rate**; a **DAC** does the reverse. **Timers and counters** measure intervals, generate periodic interrupts and produce **PWM (Pulse Width Modulation)** outputs, by which a digital pin controls the average power delivered to a motor, heater or LED. **Communication interfaces** cover UART, I²C and SPI locally, and USB, CAN (dominant in automotive), Ethernet and radio more widely. The **watchdog timer** is a counter that must be periodically reset by correctly functioning software; if the software hangs, the watchdog expires and forces a hardware reset: the standard mechanism by which an unattended system recovers from a software fault without human intervention, and a favourite examination topic. The **interrupt controller** lets an external event suspend the running program and invoke an **interrupt service routine (ISR)**, which is how embedded systems respond promptly without wasting energy polling.

Power management covers regulation, battery management and, critically for battery-powered IoT nodes, **low-power modes**. A microcontroller may draw tens of milliamperes active and a few microamperes in deep sleep, and since a typical sensor node is awake for well under one per cent of the time, deep-sleep current dominates average consumption. Designing an efficient node is largely the art of staying asleep.

EXAMPLE — Why sleep dominates battery life — a worked calculation

A node wakes once a minute, is active for 200 ms drawing 40 mA, and sleeps the remaining 59.8 s drawing 5 μ A, powered by a 2000 mAh battery.

Charge per active period: $40 \text{ mA} \times 0.2 \text{ s} = 8 \text{ mA}\cdot\text{s} = 0.00222 \text{ mAh}$

Charge per sleep period: $0.005 \text{ mA} \times 59.8 \text{ s} = 0.299 \text{ mA}\cdot\text{s} = 0.000083 \text{ mAh}$

Per hour: $\approx 0.1385 \text{ mAh} \rightarrow \text{battery life: } 2000 \div 0.1385 \approx 14,440 \text{ hours} \approx \mathbf{1.65 \text{ years}}$

Now suppose the designer neglects sleep mode and the node idles at 8 mA: *per hour* $\approx 8 \text{ mA} \rightarrow \text{battery life} = 250 \text{ hours} \approx \mathbf{10.4 \text{ days}}$.

Same hardware, same task, same battery, and a factor of roughly **58×** in service life, decided entirely by whether the firmware sleeps. This is the quantitative justification for the claim in Section 6.2 that resource constraint drives more IoT design decisions than any other characteristic. (*The calculation ignores battery self-discharge and radio transmission energy during the active window; a real budget must include both.*)

6.12.3 Software organisation

Firmware is the software resident in non-volatile memory. The term reflects its intermediate position: it is software, but bound to specific hardware and not routinely changed by the user.

Table 6.21 — Embedded software architectures compared

Architecture	How it works	Advantages	Disadvantages	Suitable for
Bare-metal (super-loop)	main() runs an infinite loop polling and servicing tasks;	Minimal footprint; no OS overhead; fully predictable	Hard to manage many concurrent tasks;	Simple, small systems

Architecture	How it works	Advantages	Disadvantages	Suitable for
	interrupts handle urgent events		timing guarantees grow fragile	
RTOS	A small kernel schedules tasks by priority, with pre-emption and synchronisation primitives	Deterministic scheduling; clean separation of concerns; scales to many tasks	Some memory and processing overhead; care needed with shared data	Medium systems with several concurrent timing-sensitive activities
Embedded general-purpose OS	A full OS with processes, virtual memory, file systems and drivers	Rich functionality, networking, standard libraries, familiar tooling	Large memory requirement; non-deterministic latency without a real-time kernel	Gateways, cameras with analytics, infotainment

An RTOS introduces several concepts worth knowing precisely. A **task** is an independent unit of execution with its own stack and priority. The **scheduler** decides which ready task runs; **pre-emptive priority scheduling** (the highest-priority ready task always runs, interrupting a lower-priority one if necessary) is the norm because it makes response time analysable, and **rate-monotonic scheduling**, giving shorter-period tasks higher priority, is a standard way of assigning those priorities for periodic task sets. A **context switch** saves one task's processor state and restores another's. **Semaphores** and **mutexes** protect shared resources while **queues** pass data safely between tasks.

IMPORTANT — Priority inversion

A hazard specific to priority-based systems: a high-priority task is blocked waiting for a resource held by a low-priority task, which is itself pre-empted by a medium-priority task, so the medium-priority task effectively delays the high-priority one, inverting the intended order. The remedy is **priority inheritance**: while the low-priority task holds a resource a higher-priority task is waiting for, it temporarily inherits that higher priority, so it cannot be pre-empted by intermediate tasks and releases the resource promptly. This is a classic examination question, and the failure mode has caused real, well-documented incidents in deployed systems.

An **ISR must be short**, because it runs with other interrupts potentially disabled and delays everything else. The standard discipline is: in the ISR do only the minimum (acknowledge the hardware, capture the data, set a flag or post to a queue), and defer processing to a normal task. Variables shared between an ISR and the main program must be declared **volatile** in C, so the compiler does not optimise away reads it cannot see being modified, and access to multi-byte shared data must be protected against interruption midway.

6.12.4 Development workflow and platforms

Embedded development differs from application development in one structural respect: **the program is written on one machine and runs on another**, and this shapes the whole workflow. After **requirements** (behaviour, deadlines, power budget, environment, unit cost, regulatory needs) comes **hardware/software partitioning**, deciding which functions are implemented in hardware (faster, costlier, fixed) and which in software (flexible, cheaper, slower), then **component selection**, a decision difficult to reverse. **Development** is typically in C, for its predictable mapping to machine behaviour and minimal runtime, with C++ in larger systems, assembly for a few critical routines, Rust increasingly where memory safety is valued, and MicroPython for prototyping. **Cross-compilation** follows: the toolchain runs on the host PC but emits machine code for the target architecture, with a linker script describing the target's memory map so code goes to flash and variables to RAM. The binary is then **flashed** through a programmer or bootloader. **Debugging** uses on-chip JTAG or SWD for breakpoints and memory inspection. But because a debugger halts the processor and destroys real-time behaviour, timing problems are usually investigated instead with a logic analyser or by toggling a spare GPIO pin and observing it

externally, a technique every embedded engineer uses. **Testing** spans host-side unit tests, hardware-in-the-loop testing, environmental testing across temperature and voltage extremes, power measurement and, where required, EMC and radio conformance certification. Finally, **deployment and maintenance** cover manufacturing programming and firmware over-the-air updates through the product's life.

Table 6.22 — Representative embedded platforms

Platform	Processor	Distinguishing feature	Typical use
8051	8-bit	Historic architecture, still taught and shipped in low-cost devices	Simple control, education
AVR / Arduino	8-bit ATmega	Very accessible tooling and large community	Prototyping, education, light commercial
PIC	8/16/32-bit	Wide family range, long production availability	Industrial and consumer control
ARM Cortex-M (STM32, nRF)	32-bit	The mainstream of modern microcontrollers; efficient, well supported	Commercial products of all kinds
ESP32 / ESP8266	32-bit with Wi-Fi and Bluetooth	Radio on the same chip at very low cost	IoT nodes, smart-home devices
Raspberry Pi	ARM Cortex-A running Linux	A full computer at low cost; <i>not</i> a microcontroller	Gateways, edge processing, vision

NOTE — Raspberry Pi is not a microcontroller

A frequent confusion. A Raspberry Pi (excluding the Pi Pico, which *is* a microcontroller board) is a single-board *computer*: it runs Linux, boots from an SD card in tens of seconds, has no on-chip firmware flash, cannot reach microampere sleep currents, and offers no hard real-time guarantees. An Arduino or STM32 board starts executing within milliseconds of power-up, sleeps at microamperes, and behaves deterministically. Use the former where you need networking, a file system and processing power; the latter where you need low power, fast start-up and predictable timing. Many IoT products contain both.

Embedded design is finally an exercise in balancing metrics none of which can be optimised freely: **unit cost** (dominant at high volume), **non-recurring engineering cost** (the one-time design cost, amortised over volume), **performance** (throughput and, more importantly in real-time systems, worst-case response), **power and energy**, **size and weight**, **reliability and safety**, **time to market** (often decisive commercially: a technically superior product that arrives late may fail), and **maintainability**, including the capacity to receive security updates for the product's life.

THINK ABOUT IT — Think About It

Consider a smart water meter that must run ten years on a non-rechargeable battery, report once a day, cost under ₹1,200 to manufacture, and survive installation in a flooded pit. Which metrics conflict? *Energy versus reporting frequency and radio range; unit cost versus battery capacity and enclosure sealing.* Which single decision constrains all the others? *The radio technology: it fixes the energy per transmission, which fixes the battery size, which fixes much of the cost and physical volume.* This is the reasoning an examiner wants when a question asks you to "design" an embedded IoT device: identify the dominant constraint, follow its consequences, and state the trade-offs you accepted.

6.12.5 Why embedded systems are the foundation of IoT

Every point in this section connects back to the rest of the unit. The **device functional block** of Section 6.5 *is* an embedded system. The **resource-constraint characteristic** of Section 6.2 is a property of embedded hardware, and is why lightweight protocols such as CoAP and MQTT exist at all. The **self-configuration characteristic** depends on the firmware-update mechanisms

described here: without reprogrammable flash and a bootloader, no deployed device could ever be patched. **Edge intelligence**, in IIoT (6.8.2), in surveillance analytics (6.9.2) and in smart systems (6.12.2), is embedded software on constrained hardware. And the **local decision logic** that keeps a home automation system working during an Internet outage (6.7.2) runs on an embedded controller.

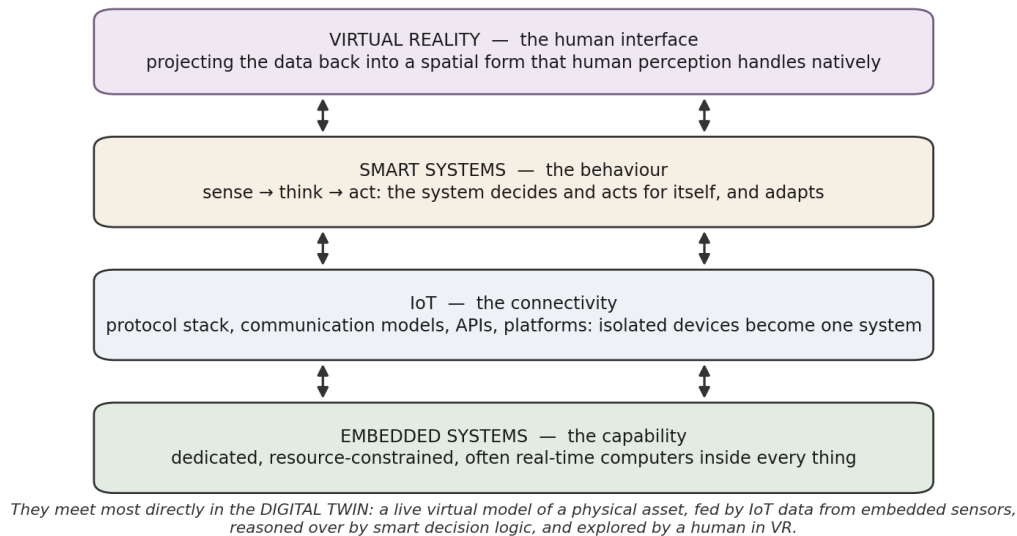


Figure 6.12 — How embedded systems, IoT, smart systems and VR interlock

The four topics of this unit are usually taught separately, which can leave the impression that they are unrelated. They are not. **Embedded systems are the substrate**: every IoT thing, every smart system's controller and every VR headset's tracking subsystem is built on one, and the constraints of embedded design (energy, memory, determinism, cost) propagate upward to explain most decisions in the other three topics. **IoT is the connectivity layer** that turns isolated embedded devices into a system: a thousand sensors that cannot communicate are a thousand separate instruments, but connected they yield aggregate insight no individual device possesses. **"Smart" describes what such a system does with that connectivity**: connectivity alone yields reporting, intelligence yields action, and the sense–think–act loop recurs identically in predictive maintenance, surveillance analytics and a home automation rule. **VR is the human interface to all of it**, projecting data back into a spatial form human perception handles natively; the **digital twin** is precisely where the two meet.

An IoT deployment can be no better than the embedded systems it is built from. A cloud platform cannot compensate for a node whose sensor is uncalibrated, whose firmware cannot be updated, or whose battery is exhausted in a month.

CHECK YOUR PROGRESS 6.12

1. Define an embedded system and state five of its defining characteristics.
2. "Real-time means fast." State whether this is correct and justify your answer.
3. Distinguish hard from soft real-time systems, with one example of each.
4. Give five points of difference between a microprocessor and a microcontroller.
5. What is a watchdog timer and what problem does it solve?
6. Explain priority inversion and the mechanism used to prevent it.
7. State the two rules of good ISR design and why each is necessary.
8. Why is cross-compilation necessary in embedded development?
9. A designer proposes a Raspberry Pi for a battery-powered soil-moisture sensor reporting twice a day for three years. Explain why this is inappropriate and state what you would use instead.

6.13 CONCEPT OF ARTIFICIAL INTELLIGENCE

6.13.1 What is Artificial Intelligence?

Computing systems have always been built to *execute instructions*: to carry out, exactly and quickly, a procedure that a human has already worked out in full. A payroll program does not decide how tax should be computed; it applies a formula that a person supplied in advance. **Artificial Intelligence (AI)** is the branch of Computer Science concerned with a different and harder goal: building systems that can decide, learn, or reason for themselves in situations the designer did not fully anticipate in advance.

IMPORTANT — Definition — Artificial Intelligence

Artificial Intelligence is the branch of Computer Science concerned with building machines and software that can perform tasks which, when performed by a human being, are understood to require intelligence, such as perceiving, learning, reasoning, planning, and deciding under uncertainty.

John McCarthy, who coined the term in 1956, described AI as "the science and engineering of making intelligent machines, especially intelligent computer programs."

A useful way to hold this definition is: AI is what allows a computer, a computer-controlled robot, or a piece of software to behave in a way that would be called *intelligent* if a human did it. That behaviour is produced by studying how people themselves perceive, learn, decide, and solve problems, and then designing algorithms and data structures that reproduce the *useful outcome* of that process, not necessarily the biological mechanism itself.

Goals of AI. Work in AI is generally directed at two related but distinct goals:

- **Building expert-level problem solvers.** Systems that exhibit intelligent behaviour in a specific domain (diagnosing a fault, recommending a product, translating a sentence) and can explain or justify their output well enough for a human to trust it.
- **Reproducing general human intelligence in machines.** A more ambitious and still largely unrealised goal: systems that understand, learn, and behave across many domains the way a human being does, rather than in one narrow task.

Nearly every system described in this unit (a spam filter, a chess engine, a voice assistant) serves the first goal. The second goal, often called **Artificial General Intelligence (AGI)**, remains an open research problem and is mentioned here only so that you can correctly place today's systems, which are all **narrow AI** (competent in one task or a closely related family of tasks), against that longer-term ambition.

6.13.2 What Contributes to AI?

AI is not a single technique but a meeting point of several older disciplines, each contributing a piece of the puzzle. Table 6.23 summarises the contribution of each.

Table 6.23 — Disciplines contributing to Artificial Intelligence

Contributing discipline	What it contributes to AI
Mathematics and statistics	Probability, optimisation, and linear algebra, which underlie learning algorithms and reasoning under uncertainty
Computer Science and engineering	Data structures, algorithms, and the hardware and software platforms on which AI systems run
Psychology and cognitive science	Models of how humans perceive, remember, and solve problems, used as inspiration for algorithm design
Linguistics	The structure of human language, needed to build systems that read, understand, or generate text and speech

Contributing discipline	What it contributes to AI
Biology and neuroscience	The structure of the brain and nervous system, which inspired neural networks (Section 6.15)
Philosophy	Formal logic and theories of knowledge, reasoning, and mind

No single project draws on all six; a speech recognition system leans heavily on linguistics and statistics, while a walking robot leans on mechanical engineering and control theory. What unifies these efforts under the name "AI" is the shared goal described in Section 6.13.1, not a shared technique.

6.13.3 AI Technique: Organising Imperfect Knowledge

Every AI system must represent and use **knowledge**: facts, rules, and relationships about its domain. In practice, that knowledge is rarely convenient to work with. It is typically:

- **Voluminous**: a medical diagnosis system may need to reason over thousands of diseases and symptoms;
- **Poorly organised**: knowledge from textbooks, experts, and observed data rarely arrives in a single consistent format; and
- **Incomplete and constantly changing**: new facts are discovered, and old ones are revised or found to be exceptions rather than rules.

An **AI technique** is a way of organising and using this imperfect knowledge so that a system built with it remains useful despite these problems. A good AI technique has three properties: it should be understandable to the people who supply the knowledge (so that errors can be found), it should be easy to correct or extend (so that a single wrong fact does not require rebuilding the whole system), and it should still produce useful results even when the underlying knowledge is incomplete or slightly inaccurate. The branches introduced later in this unit (rules and inference in expert systems, learned patterns in machine learning, degrees of truth in fuzzy logic) are, at heart, four different answers to the same question: how should imperfect knowledge be organised so that a machine can use it well?

THINK ABOUT IT — Think About It

A conventional database also stores knowledge (customer records, inventory counts), yet nobody calls a database an AI system. What is missing from a plain database that an AI technique is expected to supply? (Hint: consider what happens when the database is asked a question its schema was not designed to answer.)

CHECK YOUR PROGRESS 6.13

1. Define Artificial Intelligence in your own words, and state the two broad goals AI research pursues.
2. Distinguish narrow AI from Artificial General Intelligence (AGI).
3. Name any four disciplines that contribute to AI, and state what each contributes.
4. List the three properties a good AI technique should have.

6.14 MACHINE LEARNING

6.14.1 What is Machine Learning?

Instead of being told the rules, a machine learning system is given *data* (examples, past observations, or recorded outcomes) and is expected to work out the rules, or at least a useful approximation of them, for itself.

IMPORTANT — Definition — Machine Learning

Machine Learning is a branch of Artificial Intelligence that gives computer systems the ability to improve their performance on a task by learning from data, without being explicitly programmed with the rules for that task.

Why is this needed? Many real tasks are ones for which nobody can write down a complete rule. No engineer can enumerate every rule that separates a photograph of a cat from a photograph of a dog, or every pattern that distinguishes a fraudulent bank transaction from a genuine one; yet a system shown enough labelled examples of each can learn to tell them apart. Machine learning is the AI technique of choice precisely where the *pattern* is real but the *rule* cannot be written down by hand.

At a high level, an ML system is built by: (1) collecting data that is representative of the task, (2) choosing a mathematical model with adjustable internal parameters, (3) running a **learning algorithm** that adjusts those parameters so the model's output matches the observed data as closely as possible, and (4) using the trained model to make predictions on new, previously unseen data. Step 3 is what "learning" means in this context: automatically improving the parameters using data, in contrast to a programmer manually setting them.

6.14.2 Types of Machine Learning

Machine learning problems are conventionally divided into three categories, based on what kind of feedback the learning algorithm receives.

Supervised learning. The training data is *labelled*: for every input, the correct output is already known, much as a teacher supervises a student by providing worked answers. Formally, given input variables x and a known output y for each training example, the algorithm learns an approximate mapping function $y = f(x)$, so that when a new, unlabelled input x arrives, it can predict the corresponding output y . Supervised problems are further divided by the kind of output they predict:

- **Classification:** the output is one of a small set of categories, such as "spam" or "not spam", or "malignant" or "benign".
- **Regression:** the output is a continuous numeric quantity, such as the price of a house or the distance a vehicle will travel on a litre of fuel.

Unsupervised learning. The training data carries no labelled output at all: only the inputs x . There is no teacher and no known correct answer; the algorithm's task is to discover structure that is already present in the data. The two most common kinds of unsupervised problem are:

- **Clustering:** grouping data points so that points in the same group are more similar to each other than to points in other groups, for example grouping retail customers by purchasing behaviour.
- **Association:** discovering rules that describe how items in the data co-occur, for example finding that customers who buy bread frequently also buy butter.

Reinforcement learning. Here the system, called an **agent**, is not given labelled examples at all. Instead, it is placed in an environment, takes actions, and receives a numeric **reward** or **penalty** as feedback. Through repeated trial and error, the agent learns a strategy (a **policy**) that maximises

its cumulative reward over time. Reinforcement learning is the natural fit for problems that are sequences of decisions rather than single predictions, such as game-playing or robot control, and its mathematical framework is the **Markov Decision Process**.

Table 6.24 summarises the three types.

Table 6.24 — Types of machine learning compared

Type	Feedback available	Typical question answered	Example task
Supervised	Labelled input-output pairs	"What is the correct output for this new input?"	Predicting whether a loan will default
Unsupervised	Inputs only, no labels	"What structure already exists in this data?"	Segmenting customers into groups
Reinforcement	Reward/penalty signal from an environment	"What sequence of actions maximises long-term reward?"	Training a game-playing agent

EXAMPLE — Recognising the type from the problem statement

"Given ten years of a hospital's patient records, each labelled with whether the patient was later readmitted, predict whether a new patient will be readmitted" is **supervised classification**: the outcome (readmitted or not) is known for every historical record. "Given the same records, but with no readmission label at all, group patients into clinically meaningful subpopulations" is **unsupervised clustering**: there is no known correct grouping to learn from, only structure to discover.

6.14.3 Common Machine Learning Algorithms

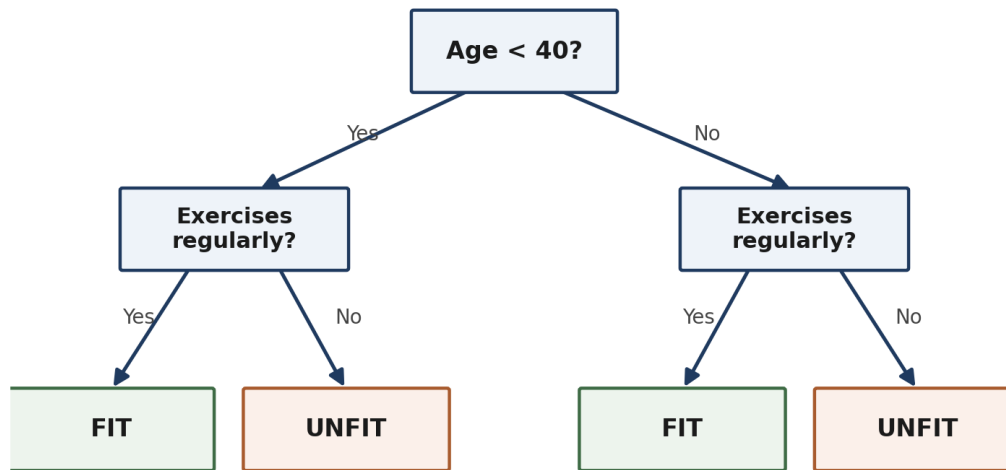
A very large number of learning algorithms exist; Table 6.25 introduces eight of the most widely used ones, organised by the type of problem (Section 6.14.2) each is normally applied to.

Table 6.25 — Eight widely used machine learning algorithms

Algorithm	Type	Core idea
Linear regression	Supervised, regression	Fits the straight line (or hyperplane, for more than one input) that best predicts a continuous output from the input variables
Logistic regression	Supervised, classification	Despite its name, a classification method: estimates the probability that an input belongs to one of two classes
Decision tree	Supervised, classification/regression	Repeatedly splits the data on the input variable that best separates the outcome, building a tree of yes/no questions
Random forest	Supervised, classification/regression	Trains many decision trees on random subsets of the data and combines their predictions by majority vote or averaging
Support Vector Machine (SVM)	Supervised, classification/regression	Finds the boundary that separates two classes with the widest possible margin
Naive Bayes	Supervised, classification	Applies Bayes' theorem, assuming input features are independent of one another given the class
K-Nearest Neighbours (KNN)	Supervised, classification/regression	Classifies a new point by majority vote among its k closest points in the training data
K-Means	Unsupervised, clustering	Partitions data into k clusters by repeatedly assigning points to the nearest of k centroids and recomputing the centroids

Two of these repay a closer look, because they illustrate two very different styles of "learning a mapping from data."

Decision tree, worked through. A decision tree answers a classification question by asking a short sequence of simple questions, one per internal node, until it reaches a **leaf** that names the predicted class. Consider a small dataset used to predict whether a person is medically fit, based on age and exercise frequency.



Internal nodes ask a question; leaves (FIT / UNFIT) give the predicted class.

Figure 6.13 — A decision tree for predicting fitness from age and exercise frequency

The tree is built by choosing, at each step, the question that best separates fit from unfit individuals in the training data (typically measured using a statistic such as information gain); the process repeats on each resulting subgroup until the groups are pure enough, or another stopping condition is reached. To classify a new person, start at the root and follow the branch matching their answer at each node until a leaf is reached. Decision trees are popular in SLM-style examinations because their reasoning can be read off directly as a set of IF-THEN rules, an advantage most other algorithms in Table 6.25 do not share.

Naïve Bayes, worked through. This algorithm classifies using Bayes' theorem:

$$P(A | B) = (P(B | A) \times P(A)) / P(B)$$

Here A is the class (for example, "spam") and B is the evidence (for example, the words present in an email). The algorithm is called *naïve* because it assumes every piece of evidence is independent of every other piece, given the class: an assumption that is rarely exactly true, but which keeps the computation simple and, in practice, still gives good results for tasks such as spam filtering and document categorisation.

NOTE — Why "naïve"?

Treating the words of an email as independent ignores the fact that some words genuinely tend to occur together ("machine" and "learning"). The assumption is a simplification, not a claim about how language actually works: hence *naïve*.

CHECK YOUR PROGRESS 6.14

1. Distinguish supervised, unsupervised, and reinforcement learning by the kind of feedback each receives.
2. A dataset labels each email as "spam" or "not spam". Is predicting this label a classification or a regression problem? Justify your answer.
3. Name the algorithm in Table 6.25 that builds many decision trees and combines their predictions, and explain briefly why combining several trees tends to be more accurate than using one.
4. State Bayes' theorem and explain what "naïve" refers to in Naïve Bayes.

6.14.4 Advantages and Limitations of Machine Learning

Table 6.26 — Advantages and limitations of machine learning

Advantages	Limitations
Can capture patterns too complex to write down as explicit rules	Needs a large volume of representative, good-quality training data
Performance improves automatically as more data becomes available	Learned patterns can be difficult to explain, especially for complex models
A single algorithm can be reused across many different problems by retraining on new data	A model trained on one context often performs poorly if the underlying data changes ("data drift")
Reduces the manual effort of hand-crafting rules for every situation	Can silently reproduce or amplify biases already present in the training data

6.14.5 Applications of Machine Learning

Table 6.27 — Representative applications of machine learning

Application area	How machine learning is used
Fraud and anomaly detection	Flagging transactions or network activity that deviate from a learned pattern of normal behaviour
Recommendation systems	Predicting which products, movies, or articles a user is likely to prefer, based on past behaviour of that user and similar users
Medical diagnosis support	Learning to associate patterns in clinical data or medical images with the presence of disease
Credit scoring	Predicting the likelihood that a loan applicant will repay, from historical repayment records
Predictive maintenance	Learning the signature of an impending equipment failure from sensor readings, so that maintenance can be scheduled before a breakdown
Search ranking	Learning which results users find useful for a given query, and ranking future results accordingly

6.15 ARTIFICIAL NEURAL NETWORKS

6.15.1 What is an Artificial Neural Network?

Machine learning algorithms such as those in Section 6.14 are, mathematically, quite simple functions (a straight line, a set of yes/no splits, a distance calculation). Many real-world patterns, however, are far too complex for such simple functions to capture well: recognising a face in a photograph regardless of lighting or angle, or understanding the meaning of a spoken sentence. **Artificial Neural Networks (ANNs)** are a family of learning models loosely inspired by the structure of the biological brain, designed to represent and learn exactly this kind of complex, non-linear pattern.

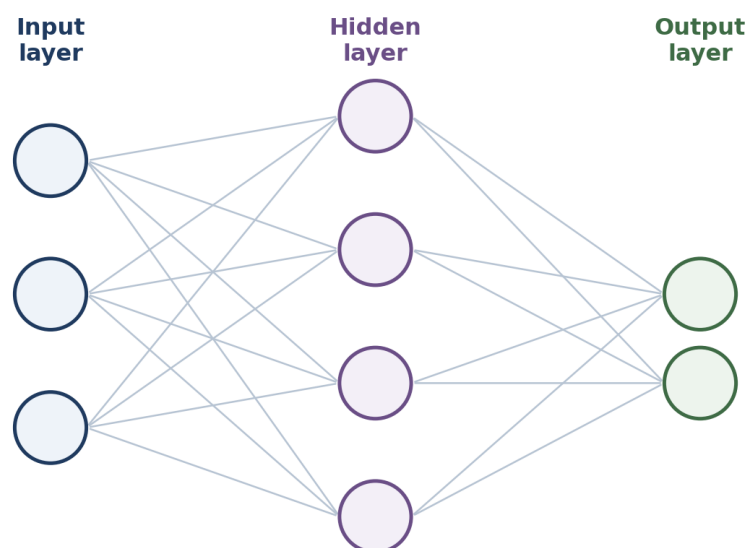
IMPORTANT — Definition — Artificial Neural Network

An **Artificial Neural Network** is a computing system composed of a large number of simple, interconnected processing units called **neurons** (or **nodes**), organised in layers, which learns to perform a task by adjusting the strength of the connections between neurons in response to training data.

The inspiration is biological. The human brain contains roughly 100 billion **neurons**, each connected to thousands of others through fibres called **axons** and **dendrites**. A neuron receives electrical signals from many others and, if the combined signal is strong enough, fires a signal onward to the neurons it is connected to. Learning, in the biological brain, corresponds to strengthening or weakening particular connections through experience. An ANN reproduces the *structure* of this process in software (many simple units, densely connected, each combining its inputs and passing on a result) without claiming to reproduce the brain's biology in any literal sense.

6.15.2 Structure of an ANN

An ANN is built from neurons arranged in **layers**: an **input layer** that receives the raw data, one or more **hidden layers** that transform it, and an **output layer** that produces the final result. Every connection between two neurons carries a numeric **weight**, which determines how strongly the signal from one neuron influences the next. Each neuron combines its weighted inputs and passes the result through an **activation function** (a rule that decides how strongly the neuron itself should "fire") before passing its output, called its **activation**, to the next layer.



Each connection carries a weight; each neuron combines its weighted inputs through an activation function and passes the result forward.

Figure 6.14 — Structure of a simple feedforward artificial neural network

Learning in an ANN means adjusting the weights so that, for a given set of inputs, the network's output matches the desired output as closely as possible. The standard algorithm for doing this in a supervised setting is **backpropagation**: the network is run forward on a training example, its output is compared with the known correct answer to compute an **error**, and that error is then propagated backward through the network, layer by layer, to determine how much each weight contributed to the mistake and adjust it accordingly. Repeating this process over many training examples gradually reduces the overall error.

6.15.3 Types of ANN Topology

ANNs are classified by how information is allowed to flow through their connections.

- **Feedforward networks.** Information flows in one direction only (from input layer, through the hidden layers, to the output layer), with no connection ever sending information backward. There are no loops. Feedforward networks have fixed inputs and outputs and are the most common topology, used for pattern classification and recognition tasks such as those in Section 6.14.
- **Feedback (recurrent) networks.** Connections are permitted to loop back to earlier layers or to the same layer, so that a neuron's own past output can influence its future input. This makes feedback networks suitable for tasks with an inherent notion of sequence or memory, such as content-addressable memory, where a partial or noisy input pattern should be completed or corrected using previously learned associations.

6.15.4 Learning Strategies in ANNs

ANNs can be trained under any of the three learning paradigms introduced in Section 6.14.2, applied specifically to a network of weighted connections:

- **Supervised learning.** The network is shown an input together with the correct output (as in pattern recognition), compares its own guess with the correct answer, and adjusts its weights, via backpropagation, to reduce the difference.
- **Unsupervised learning.** No correct output is provided. The network instead discovers structure on its own (for example, clustering similar inputs together) based on regularities present in the input data alone.
- **Reinforcement learning.** The network interacts with an environment and adjusts its weights based on a reward or penalty signal received after acting, rather than from a labelled example of the correct action.

NOTE — Backpropagation networks and pattern recognition

Backpropagation is specifically a *training algorithm*, not a network topology. It is most commonly applied to feedforward networks and is well suited to pattern recognition and input-to-output mapping tasks, where a large set of correctly labelled examples is available for training.

CHECK YOUR PROGRESS 6.15

1. Define an Artificial Neural Network and identify the three types of layer present in a typical ANN.
2. Explain the role of a weight and an activation function in a single neuron.
3. Distinguish a feedforward ANN from a feedback (recurrent) ANN, and give one task suited to each.
4. Briefly describe what backpropagation does when a network's output does not match the desired output.

6.15.5 Advantages and Limitations of ANNs

Table 6.28 — Advantages and limitations of artificial neural networks

Advantages	Limitations
Can learn highly complex, non-linear relationships that simpler ML algorithms cannot represent well	Typically require large amounts of training data and significant computing power
Continue to perform reasonably even with some noisy or incomplete input	Behave as a "black box": it is difficult to explain exactly why a trained network produced a given output
The same basic architecture generalises across very different tasks: vision, speech, text	Training can be slow, and a poorly chosen architecture may fail to learn the pattern at all

6.15.6 Applications of Neural Networks

Neural networks are particularly effective at tasks humans perform effortlessly but which are difficult to reduce to explicit rules: recognising a face, understanding speech, or judging the quality of a signal.

Table 6.29 — Representative applications of artificial neural networks

Domain	Representative applications
Vision and image processing	Face and object recognition, optical character recognition, medical image analysis
Speech and audio	Speech recognition, speech synthesis, noise filtering in hearing aids
Finance	Credit scoring, fraud detection, algorithmic trading, currency value prediction
Aerospace and automotive	Autopilot systems, fault detection, autonomous driving assistance
Industry	Quality inspection, process control, predictive maintenance
Medicine	Analysis of ECG/EEG signals, cancer cell analysis, prosthetic design
Time-series forecasting	Stock price and demand forecasting, weather-related prediction

6.16 ROBOTICS

6.16.1 What is Robotics?

Robotics is the branch of AI concerned with designing, constructing, and programming **robots**: artificial agents that perceive and act in the physical world, rather than a purely computational one. Robotics is inherently multidisciplinary, drawing on mechanical engineering for the robot's body, electrical engineering for its power and control circuitry, and computer science and AI for the software that decides what the robot should do.

IMPORTANT — Definition — Robot

A **robot** is an artificial, programmable agent capable of sensing its physical environment and acting on it (moving, manipulating objects, or otherwise producing a physical effect) in order to carry out a task, typically one that would otherwise require a human to perform it.

The purpose of building robots is generally one of three things: to perform tasks in environments too dangerous or inaccessible for humans (bomb disposal, deep-sea or space exploration); to perform repetitive tasks with greater precision, speed, and consistency than a human, freeing people from work that would otherwise be tedious or exhausting; or to perform tasks that require physical strength or endurance beyond human capability.

6.16.2 Robots versus Conventional AI Programs

A robot differs from a purely software-based AI program (such as a chess engine or an expert system) in a fundamental way: it must operate in, and be affected by, the real physical world rather than a symbolic or simulated one.

Table 6.30 — Robots compared with conventional AI programs

Aspect	Conventional AI program	Robot
Operating environment	A symbolic or simulated world inside a computer	The real physical world
Input	Symbols, numbers, and rules	Analogue signals from sensors: sound waveforms, camera images, force readings
Hardware requirement	A general-purpose computer	Specialised hardware: sensors, actuators, and a mechanical body

6.16.3 Components of a Robot

Every robot, regardless of its purpose, is built from the same four categories of component.

- **Mechanical structure:** the physical form and construction that suits the robot to its task, from a jointed arm to a wheeled chassis.
- **Power supply:** the energy source driving the robot's motion and computation, typically batteries, but sometimes solar, hydraulic, or pneumatic power.
- **Actuators:** devices that convert stored energy into physical movement. Common types include electric motors (for rotational movement), pneumatic "air muscles" (which contract when compressed air is admitted), and piezoelectric or ultrasonic motors (used in industrial robots requiring fine, precise motion).
- **Sensors and a controller:** sensors provide the robot with real-time information about its task environment (vision sensors for depth and shape, tactile sensors that imitate the touch receptors in a human fingertip, and sensors for light, temperature, sound, or pressure), while an onboard controller (an embedded computer) decides what the robot should do next based on that sensor information and its program.

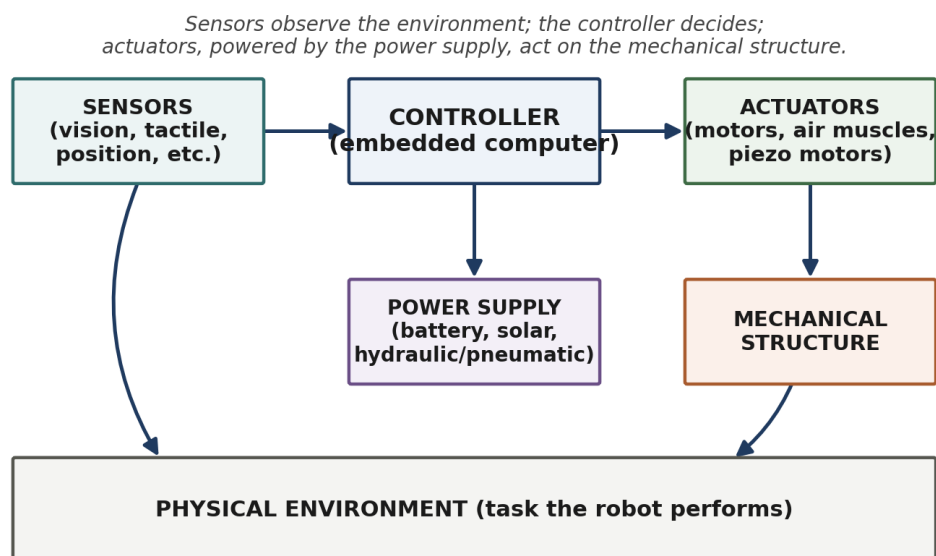


Figure 6.15 — Core components of a robot and how they interact

6.16.4 Robot Locomotion

Locomotion is the mechanism by which a robot moves through its environment. The main forms of locomotion are legged, wheeled, a combination of the two, and tracked (slip/skid) locomotion.

Legged locomotion suits rough or irregular terrain, where wheels would lose traction, but it is mechanically complex, requires more motors, and consumes more power than wheeled locomotion because the robot's body must be actively balanced at every step. A robot's legs may number one, two, four, or six; with more than one leg, the legs' movements must be coordinated, or **gaited**, so that the robot remains stable while walking.

EXAMPLE — Counting possible leg events

For a robot with k legs, one commonly used way to count the distinct lifting-and-releasing events available to it is $N = (2k - 1)!$. For a two-legged robot ($k = 2$): $N = (2 \times 2 - 1)! = 3! = 6$. These six events are: lifting the left leg, releasing the left leg, lifting the right leg, releasing the right leg, lifting both legs together, and releasing both legs together. For a six-legged robot ($k = 6$), the same formula gives $N = 11! = 39,916,800$, illustrating, in concrete numbers, why the control complexity of a legged robot grows very sharply with the number of legs.

Wheeled locomotion requires fewer motors, is mechanically simpler, and is far more power-efficient than legged locomotion, at the cost of being less capable on rough terrain. Robots may use standard wheels (rotating about a single axle), castor wheels (which also rotate about an offset steering joint, as in a shopping trolley), or omnidirectional wheels (which can move sideways as well as forward), depending on the maneuverability required.

Tracked (slip/skid) locomotion, as used in a tank, steers by driving the two tracks at different speeds. The large contact area between track and ground gives excellent stability and traction, at some cost in efficiency and precision of turning.

6.16.5 Computer Vision as a Sensing Capability

Many robots rely on **computer vision**: the capability, built from cameras and image-processing algorithms, that allows a machine to extract meaningful information from images. Computer vision underlies tasks such as object recognition (identifying items on a production line or in a supermarket), face detection, optical character recognition (converting a scanned or photographed document into editable text), and estimating the position of an object relative to the camera, as is required for a robot to reach for and pick up an item accurately.

6.16.6 Advantages and Limitations of Robotics

Table 6.31 — Advantages and limitations of robotics

Advantages	Limitations
Performs dangerous, repetitive, or physically demanding tasks without fatigue, boredom, or risk to human life	High initial cost of hardware, sensors, and programming
Achieves precision and consistency difficult for a human to sustain over long periods	Limited adaptability: a robot designed for one task or environment often cannot be redeployed easily to another
Can operate in environments closed to humans: vacuum, extreme heat, radiation, deep water	Depends on the reliability of its mechanical parts and sensors; a single component failure can halt the whole task

CHECK YOUR PROGRESS 6.16

1. Define a robot, and give two reasons robots are built to perform tasks rather than leaving those tasks to humans.
2. List the four categories of component present in every robot.
3. Compare legged and wheeled locomotion on power efficiency and suitability to terrain.
4. Using $N = (2k - 1)!$, verify that a four-legged robot has 5,040 possible lifting-and-releasing events.

6.16.7 Applications of Robotics

Table 6.32 — Representative applications of robotics

Domain	Representative applications
Industry	Material handling, welding, spray painting, drilling, and polishing on production lines
Military and defence	Reaching hazardous or inaccessible zones; India's DRDO-developed robot <i>Daksh</i> is used to safely handle and destroy explosive devices

Domain	Representative applications
Medicine	Robot-assisted surgery, rehabilitation of patients with disabilities, high-throughput clinical testing
Exploration	Planetary rovers for space exploration; underwater drones for ocean exploration
Entertainment	Animatronics and robotic characters used in film-making and theme parks

6.17 NATURAL LANGUAGE PROCESSING

6.17.1 What is Natural Language Processing?

Natural Language Processing (NLP) is the branch of AI concerned with enabling computers to work with human ("natural") language (such as English or Hindi) in the form of written text or speech, as opposed to the formal, rigid syntax of a programming language. NLP is needed wherever a person should be able to instruct or query a system in ordinary language: telling a voice assistant to set an alarm, typing a question to a chatbot-based expert system, or having a document automatically summarised or translated.

IMPORTANT — Definition — Natural Language Processing

Natural Language Processing is the field of Artificial Intelligence that develops methods for computers to understand, interpret, and generate human language, in either written or spoken form.

6.17.2 Components of NLP

NLP is conventionally divided into two complementary components, distinguished by the direction in which information flows between language and meaning.

- **Natural Language Understanding (NLU)**, the harder of the two, takes language as input and produces a structured, machine-usable representation of its meaning. It involves mapping the input sentence into a useful internal representation and analysing the input at several linguistic levels (word structure, sentence structure, meaning, and context), each described in Section 6.17.3.
- **Natural Language Generation (NLG)** takes a structured internal representation and produces natural language as output: the reverse direction. NLG proceeds through **text planning** (deciding what content, drawn from a knowledge base, should be communicated), **sentence planning** (choosing words, forming grammatical phrases, and setting an appropriate tone), and **text realisation** (converting the planned sentence structure into the final, grammatically correct sentence).

NLU is harder than NLG chiefly because human language is genuinely **ambiguous**: the same words can carry more than one meaning, and NLU must choose correctly among them, whereas NLG starts from a meaning that is already fixed and only needs to express it well.

EXAMPLE — Three kinds of ambiguity

Lexical ambiguity occurs at the level of a single word: in "board the plane" versus "the wooden board", the word *board* is a verb in one sentence and a noun in the other. **Syntactic ambiguity** occurs at the level of sentence structure: "He lifted the beetle with the red cap" could mean either that he used the cap to lift the beetle, or that the beetle he lifted had a red cap. **Referential ambiguity** occurs when a pronoun's referent is unclear: in "Rima went to Gauri. She said, 'I am tired,'" it is not clear from the sentence alone whether *she* refers to Rima or Gauri. Resolving these ambiguities correctly, using context, is a central difficulty of NLU.

6.17.3 Levels of NLP Processing

A complete NLP system typically processes an input sentence through five progressively deeper levels of analysis.

Input sentence is analysed in five progressively deeper stages



Words -> Grammar -> Literal meaning -> Context of nearby sentences -> Intended meaning

Figure 6.16 — The five levels of NLP processing

1. **Lexical analysis.** The input text is divided into paragraphs, sentences, and individual words, and each word is identified against the vocabulary (lexicon) of the language.
2. **Syntactic analysis (parsing).** The words of each sentence are analysed against the grammar of the language, and their grammatical relationship to one another (subject, verb, object) is determined. A sentence that violates the grammar, such as "The school goes to boy," is rejected at this stage even though every individual word is valid.
3. **Semantic analysis.** The literal, dictionary meaning is extracted from the syntactically valid sentence, and the sentence is checked for meaningfulness in the real world. A grammatically correct but meaningless sentence such as "hot ice-cream" is flagged as anomalous at this stage.
4. **Discourse integration.** The meaning of a sentence is interpreted in the context of the sentences immediately before and after it, since a single sentence's meaning is often incomplete on its own, as in the referential-ambiguity example above, where "she" can only be resolved using the preceding sentence.
5. **Pragmatic analysis.** What was literally said is re-interpreted in light of what the speaker actually meant, drawing on real-world knowledge beyond the text itself, for example, correctly interpreting "Can you pass the salt?" as a request rather than a question about physical ability.

6.17.4 Essential NLP Terminology

Table 6.33 — Core NLP terminology

Term	Meaning
Phonology	The study of how sounds are organised systematically in a language
Morphology	The study of how words are built from smaller meaningful units
Morpheme	The smallest unit of meaning in a language
Syntax	The rules governing how words combine to form grammatically valid sentences
Semantics	The study of meaning: of individual words and of the phrases and sentences they form
Pragmatics	The study of how context affects the interpretation of an otherwise fixed sentence
Discourse	The study of how meaning is affected by the sentences surrounding a given sentence

CHECK YOUR PROGRESS 6.17

1. Distinguish Natural Language Understanding from Natural Language Generation, and explain why NLU is generally considered the harder problem.
2. Identify the type of ambiguity present in "I saw the man with the telescope."
3. List the five levels of NLP processing in order, from the shallowest to the deepest.
4. Define morphology and syntax, and explain how they differ.

6.17.5 Advantages and Limitations of NLP

Table 6.34 — Advantages and limitations of natural language processing

Advantages	Limitations
Lets a non-technical user interact with a system in ordinary language instead of a rigid command syntax	Genuine ambiguity in human language means no NLP system achieves perfect understanding
Can process very large volumes of text far faster than manual reading	Performance often degrades on informal language, slang, code-switching, or low-resource languages
Automates tasks (translation, summarisation, sentiment analysis) that would otherwise require extensive manual effort	Struggles with sarcasm, humour, and meaning that depends on shared real-world context

6.17.6 Applications of Natural Language Processing

Table 6.35 — Representative applications of natural language processing

Application	Description
Machine translation	Automatically translating text or speech from one language to another
Sentiment analysis	Determining the emotional tone (positive, negative, neutral) expressed in a piece of text, widely used to analyse product reviews and social media
Chatbots and virtual assistants	Understanding spoken or typed requests and responding appropriately, as in Siri, Alexa, or a customer-service chatbot
Grammar and writing assistance	Detecting grammatical errors and suggesting stylistic improvements, as in tools such as Grammarly
Text summarisation	Automatically producing a shorter version of a document that preserves its key information
Information extraction	Automatically pulling structured facts (names, dates, relationships) out of unstructured text

6.18 EXPERT SYSTEMS

6.18.1 What is an Expert System?

An **Expert System** is an AI program that captures the specialised knowledge of a human expert in a particular subject area, or **domain**, and uses that knowledge to solve problems or offer advice at a level of competence approaching that of the human expert. Edward Feigenbaum of Stanford University, one of the field's founders, defined an expert system as an intelligent program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solution.

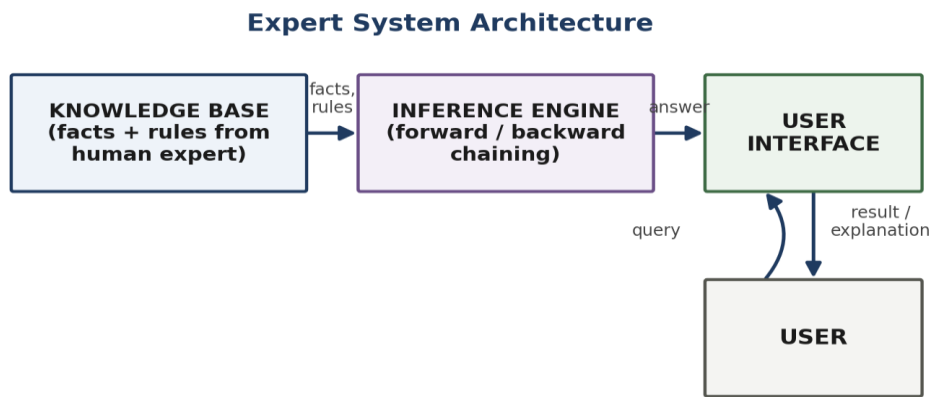
IMPORTANT — Definition — Expert System

An **Expert System** is a knowledge-based AI program that stores domain-specific facts and rules gathered from a human expert, and uses an inference procedure over that knowledge to answer questions, diagnose problems, or recommend actions within its domain.

Expert systems are also called **knowledge-based systems**, because their competence depends less on a clever general-purpose algorithm and more on the quality and completeness of the domain knowledge stored within them: knowledge that may be drawn directly from a human expert, or from textbooks, manuals, and other documented sources.

6.18.2 Architecture of an Expert System

An expert system is built from three principal components, working together as shown in Figure 6.17.



Forward chaining: facts -> rules -> conclusion ("what will happen?")
 Backward chaining: hypothesis -> rules -> supporting facts ("why did this happen?")

Figure 6.17 — Architecture of an expert system

- **Knowledge base.** The core of the system: a structured repository of domain facts and rules, gathered from the human expert through a process called knowledge acquisition. It typically contains both **empirical knowledge** (established facts of the domain) and **heuristic knowledge** (the expert's own rules of thumb, built from experience).
- **Inference engine.** The component that reasons over the knowledge base to reach conclusions, in much the way a human expert would reason from the same facts and rules. The inference engine typically uses one of two reasoning strategies:
 - **Forward chaining** starts from known facts and repeatedly applies matching rules to derive new facts, moving forward until a conclusion is reached: answering the question "what will happen, given what I currently know?" It suits prediction tasks, such as forecasting a likely stock-market movement from current indicators.
 - **Backward chaining** starts from a candidate conclusion (a hypothesis) and works backward, checking whether the facts needed to support that hypothesis are actually true: answering the question "why did this happen, or is this hypothesis correct?" It suits diagnostic tasks, such as determining the cause of a patient's stomach pain by testing hypotheses in turn.
- **User interface.** The component that manages interaction between the (typically non-expert) user and the system: accepting a query in a usable form, such as a menu or form, and presenting the system's conclusions, along with an explanation of how they were reached, in a way the user can act on.

EXAMPLE — Forward versus backward chaining

Given the facts "the patient has a fever" and "the patient has a cough," a **forward-chaining** system might apply the rule "fever + cough -> suspect respiratory infection" to arrive, step by step, at a diagnosis. A **backward-chaining** system, instead, starts from the hypothesis "the patient has a respiratory infection" and works backward to check whether fever and cough (the facts required to support that hypothesis) are actually present.

CHECK YOUR PROGRESS 6.18

1. Define an expert system and explain why such systems are also called "knowledge-based systems."
2. Name the three main components of an expert system's architecture.
3. Distinguish forward chaining from backward chaining, and state which one is better suited to a diagnostic task.
4. Distinguish empirical knowledge from heuristic knowledge within a knowledge base.

6.18.3 Advantages and Limitations of Expert Systems

Table 6.36 — Advantages and limitations of expert systems

Advantages	Limitations
Makes scarce expert knowledge available consistently, on demand, and without the expert being present	Building and maintaining the knowledge base is time-consuming and depends on the availability of a cooperative human expert
Can explain the reasoning behind its conclusion, which builds user trust	Struggles outside its defined domain, and cannot exercise the "common sense" a human expert brings to an unusual case
Provides consistent answers, unaffected by fatigue, mood, or memory lapses that affect human experts	Needs to be manually updated as domain knowledge changes, or it can give confidently wrong advice

6.18.4 Applications of Expert Systems

Table 6.37 — Representative applications of expert systems

Domain	Representative application
Medicine	Diagnostic support systems that suggest likely diseases from reported symptoms and test results
Finance	Credit approval and investment-advisory systems that apply an institution's lending or investment rules
Manufacturing	Fault-diagnosis systems that identify the likely cause of an equipment failure from observed symptoms
Customer support	Systems that guide a support agent, or a customer directly, through a structured troubleshooting procedure
Agriculture	Crop-advisory systems that recommend fertiliser, irrigation, or pest control based on soil and weather data

6.19 FUZZY LOGIC SYSTEMS

6.19.1 What is Fuzzy Logic?

Conventional ("crisp") logic, the kind a computer naturally works with, takes precise input and produces a definite output: true or false, 0 or 1. Human reasoning does not usually work this way. Asked whether the weather today is "hot," a person rarely answers with a flat yes or no; the honest answer usually lies somewhere between "certainly yes" and "certainly no." **Fuzzy Logic**, introduced by Lotfi Zadeh, is a form of reasoning designed to formalise exactly this kind of graded, human-like judgement inside a computer system.

IMPORTANT — Definition — Fuzzy Logic

Fuzzy Logic is a method of reasoning that allows a variable to take a *degree* of truth between completely false and completely true, rather than being restricted to the two extreme values used in conventional logic, and it uses this graded truth to produce a definite, usable output from input that may itself be imprecise or incomplete.

Fuzzy logic is useful precisely because so much of the input available to real systems is imprecise, noisy, or only approximately known: a room's temperature "feels" warm rather than measuring exactly 27.4°C in the way a control decision needs to reason about it. A **fuzzy logic system (FLS)** is designed to take such input and still produce an acceptable, definite output.

6.19.2 Architecture of a Fuzzy Logic System

A fuzzy logic system has four main components, applied in sequence.

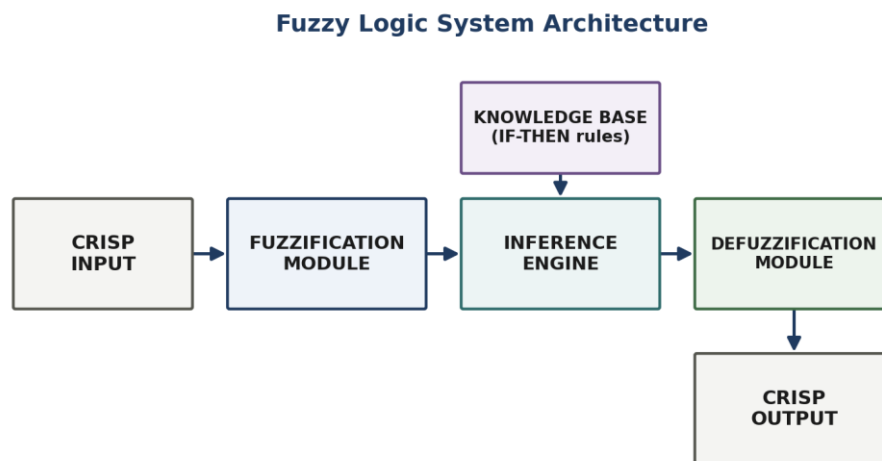


Figure 6.18 — Architecture of a fuzzy logic system

1. **Fuzzification module.** Converts precise ("crisp") numeric inputs into **fuzzy sets**: that is, it decides the degree to which the input belongs to each of several linguistic categories, such as *Large Positive*, *Medium Positive*, *Small*, *Medium Negative*, and *Large Negative*.
2. **Knowledge base.** Stores the IF-THEN rules supplied by domain experts, expressed in terms of the same linguistic categories used by the fuzzification module.
3. **Inference engine.** Simulates human reasoning by applying the IF-THEN rules from the knowledge base to the fuzzified inputs, combining the results of every rule that applies.
4. **Defuzzification module.** Converts the fuzzy result produced by the inference engine back into a single, crisp, usable output value.

6.19.3 Membership Functions

A **membership function** is the mechanism that performs fuzzification: it assigns every possible input value a **degree of membership**, a number between 0 and 1, in a given fuzzy set. A value of 0 means the input does not belong to that fuzzy set at all; a value of 1 means it belongs completely; values in between express partial membership. Several membership functions, one per linguistic category, are typically used together to fuzzify a single numeric input. The **triangular** shape is the most commonly used membership function, though trapezoidal, Gaussian, and other shapes are also used, since simple shapes are usually sufficient and add little practical accuracy over more complex ones.

6.19.4 A Worked Example: Fuzzy Logic Air Conditioner

Consider an air conditioner that adjusts room temperature by comparing the current room temperature with a target temperature, using the linguistic terms *Very Cold*, *Cold*, *Warm*, *Hot*, and *Very Hot*.

Step 1: Define linguistic variables. The variable *Temperature* is defined over the set {Very Cold, Cold, Warm, Hot, Very Hot}; each of these five terms covers part of the overall temperature range.

Step 2: Construct membership functions. A membership function is defined for each of the five linguistic terms, describing the degree to which a given numeric temperature belongs to that term.

Step 3: Construct the rule base. A table of rules specifies the required action for every combination of room temperature and target temperature.

Table 6.38 — Rule base for the fuzzy air-conditioning example

Room Temp / Target	Very Cold	Cold	Warm	Hot	Very Hot
Very Cold	No change	Heat	Heat	Heat	Heat
Cold	Cool	No change	Heat	Heat	Heat

Room Temp / Target	Very Cold	Cold	Warm	Hot	Very Hot
Warm	Cool	Cool	No change	Heat	Heat
Hot	Cool	Cool	Cool	No change	Heat
Very Hot	Cool	Cool	Cool	Cool	No change

These are then written as IF-THEN-ELSE rules, for example: *IF room temperature is (Cold or Very Cold) AND target is Warm, THEN heat.*

Step 4: Fuzzify, infer, and defuzzify. The current room temperature and target temperature are converted into degrees of membership in the five linguistic terms (fuzzification). The rule base is then evaluated: fuzzy AND is computed as the minimum of the two memberships involved, and fuzzy OR as their maximum; every rule that applies contributes to the result, and these contributions are combined into a single fuzzy output. Finally, this fuzzy output is converted back into a single, crisp adjustment to the compressor or heater (defuzzification).

6.19.5 Advantages and Limitations of Fuzzy Logic

Table 6.39 — Advantages and limitations of fuzzy logic systems

Advantages	Limitations
Underlying mathematics is comparatively simple	No single systematic method exists for designing a fuzzy system from scratch
Easy to modify: rules can be added or removed without redesigning the whole system	Becomes hard to understand and validate once the rule base grows large
Tolerates imprecise, noisy, or incomplete input gracefully	Best suited to problems that do not demand very high numerical accuracy
Can be implemented in hardware, software, or a combination of both, across systems from small microcontrollers to large workstation-based control systems	

CHECK YOUR PROGRESS 6.19

1. Explain, in your own words, how fuzzy logic differs from conventional ("crisp") logic.
2. Name the four components of a fuzzy logic system's architecture, in the order a signal pass through them.
3. What does a membership function of 0.7 mean for a given input and fuzzy set?
4. State one advantage and one limitation of fuzzy logic systems.

6.19.6 Applications of Fuzzy Logic

Table 6.40 — Representative applications of fuzzy logic

Domain	Representative applications
Automotive systems	Automatic gearboxes, four-wheel steering, cabin climate control
Consumer electronics	Camera autofocusses and exposure, washing machine wash-cycle selection, hi-fi sound tuning
Domestic appliances	Microwave ovens, refrigerators, vacuum cleaners
Environmental control	Air conditioners, heaters, and humidifiers that adjust gradually rather than switching abruptly on and off

6.20 GENERAL APPLICATIONS OF ARTIFICIAL INTELLIGENCE

Sections 1.2 to 1.7 examined what each branch of AI does and where that branch specifically is applied. In practice, most real products combine several branches at once: a single voice assistant uses NLP to understand speech, machine learning to personalise its responses, and often a rule-based component resembling an expert system to handle well-defined commands reliably. Table 6.41 surveys AI's reach across everyday domains, cutting across the branch-specific views already given.

Table 6.41 — General applications of Artificial Intelligence across domains

Domain	How AI is typically used	Illustrative example
E-commerce	Recommendation engines analyse browsing and purchase history to suggest products	Online retailers recommend items based on a customer's browsing and purchase history
Education	Adaptive learning platforms tailor content and pacing to an individual learner's strengths and weaknesses	Online learning platforms use AI to recommend courses and personalise feedback
Navigation and transport	Route-planning systems combine live traffic, weather, and historical data to suggest routes and estimate arrival times	Mapping applications calculate the fastest route using real-time traffic conditions
Automobiles	Driver-assistance and autonomous-driving systems process sensor and camera data to keep lane, maintain distance, and detect obstacles	Modern driver-assistance systems combine lane-keeping, adaptive cruise control, and automated parking
Healthcare	AI analyses medical images and patient records to support diagnosis and personalise treatment	Diagnostic-support systems flag likely conditions from medical imaging and clinical data
Finance	Algorithms analyse transaction and market data for fraud detection, credit scoring, and investment advice	Robo-advisory platforms build and rebalance investment portfolios automatically
Human resources	Automated screening matches resumes against job requirements and schedules interviews	Recruitment platforms rank candidate resumes against a job description automatically
Agriculture	Sensors and drones monitor crop health and soil conditions to guide irrigation and fertilisation decisions	Precision-agriculture platforms combine drone and sensor data to guide farming decisions
Data security and cybersecurity	Systems monitor network traffic in real time to detect and respond to anomalous or malicious activity	AI-based security platforms flag unusual network behaviour that may indicate an intrusion
Astronomy	Algorithms process telescope and satellite data to detect and classify celestial objects automatically	Space agencies use AI to identify candidate exoplanets from telescope brightness data
Generative AI	Machine learning models generate new text, images, audio, or video from a prompt, rather than merely classifying or predicting from existing data	Conversational systems such as ChatGPT, Claude, and Gemini generate human-like text for writing assistance, summarisation, and coding help
Entertainment	Streaming platforms use viewing and listening history to personalise recommendations	Music and video streaming services build personalised playlists and watch-lists from user history

THINK ABOUT IT — Think About It

Pick any one row of Table 6.41 and identify which branch, or combination of branches, from Sections 1.2 to 1.7 most plausibly powers it. A recommendation engine, for instance, is chiefly machine learning (Section 6.14); a diagnostic-support system in healthcare typically combines machine learning with an expert-system-style rule base (Section 6.18).

CHECK YOUR PROGRESS 6.20

1. Name three domains, other than those used as examples in this section, where AI is applied in everyday life.
2. Explain what distinguishes generative AI from the earlier machine learning applications discussed in Section 6.14.
3. For the "Healthcare" row of Table 6.41, identify which branch of AI (Sections 1.2–1.7) is most directly responsible for the application described.

6.21 SUMMARY

For most of computing history, the network carried traffic between people: someone typed a request, and a distant machine answered. That is no longer true. Most devices connected to networks today are not operated by a person at all. A soil-moisture probe buried in a field, a vibration sensor bolted to a pump housing, an electricity meter on a building, a heart-rate monitor on a patient's wrist: each produces data continuously, without being asked, and often acts on that data without anyone being told. This shift from a network of people to a network of things is the Internet of Things (IoT), and it is where this chapter began.

Three related ideas followed. Virtual Reality (VR) turned out to address the reverse problem: IoT brings the physical world into the computer as data, while VR sends the computed world back out to human senses. Smart systems proved to be the general category IoT belongs to: any engineered system that senses its environment, reasons about what it senses, and acts on its conclusions. Embedded systems sit beneath all of this as the enabling layer: the small, dedicated, resource-constrained computers built into the "things" themselves. Without an embedded system there is no IoT; it is where a deployment physically begins.

The chapter's second half took up Artificial Intelligence: building machines and software that perform tasks normally understood to require human intelligence, from the narrow, task-specific competence every system in this chapter shows, to the still-unrealised long-term goal of Artificial General Intelligence. AI's real difficulty is organising imperfect, incomplete, constantly changing knowledge so it stays useful, and six branches were covered, each answering that difficulty differently. Machine Learning learns patterns from data instead of explicit rules, through supervised, unsupervised, and reinforcement learning. Artificial Neural Networks build on this with layered, brain-inspired structures that learn more complex, non-linear patterns, trained mainly through backpropagation. Robotics puts AI into physical agents that sense and act in the real world, built from a mechanical structure, power supply, actuators, and sensors, and moved by legs, wheels, or tracks. Natural Language Processing lets computers understand and generate human language, splitting the task into understanding (NLU) and generation (NLG) across five progressively deeper levels of analysis. Expert Systems store a human expert's domain knowledge in a knowledge base and reason over it with an inference engine, either chaining forward toward a conclusion or backward from a hypothesis. Fuzzy Logic formalises graded, human-like judgement, reasoning in degrees of truth rather than strict true or false, through fuzzification, a rule base, inference, and defuzzification.

None of these six branches tends to work alone. The AI applications people actually encounter (recommendation engines, voice assistants, autonomous vehicles, diagnostic support systems) usually combine several of them at once, and that combination matters more for the rest of the course than any single technique on its own.

6.22 GLOSSARY

Term	Meaning
6LoWPAN	IPv6 over Low-power Wireless Personal Area Networks; an adaptation layer providing header compression and fragmentation so IPv6 can run over IEEE 802.15.4
Activation function	The rule within a neuron that determines how strongly it "fires" given its combined inputs
Actuator	A device converting an electrical signal into a physical effect
ADC	Analog-to-Digital Converter; characterised by resolution and sampling rate
AI technique	A way of organising and using imperfect, incomplete, or changing knowledge so that a system remains useful despite it
Ambiguity (lexical/syntactic/referential)	A property of natural language in which a word, sentence structure, or pronoun reference admits more than one interpretation
AR (Augmented Reality)	Computer-generated information overlaid on the real world, without spatial anchoring to real geometry
Artificial General Intelligence (AGI)	A still-unrealised, longer-term AI goal: general human-like intelligence across many domains, rather than one narrow task
Artificial Intelligence (AI)	The branch of Computer Science concerned with building systems that perform tasks understood to require intelligence when performed by humans
Artificial Neural Network (ANN)	A learning model composed of layered, weighted, interconnected processing units ("neurons"), loosely inspired by the brain
Backpropagation	The standard supervised training algorithm for ANNs, which adjusts weights by propagating output error backward through the network
Backward chaining	Reasoning from a hypothesis backward to check whether supporting facts hold
Bare-metal	Embedded software running directly on hardware without an operating system, typically a super-loop with interrupts
Broker	The intermediary in publish-subscribe that receives published messages and forwards them to subscribers
CAVE	Cave Automatic Virtual Environment; a room-sized semi-immersive VR system using projection
Classification	A supervised learning problem whose output is one of a fixed set of categories
Clustering	An unsupervised learning problem: grouping similar data points together
CoAP	Constrained Application Protocol (RFC 7252); binary, RESTful, UDP-based, with an <i>observe</i> option for change notification
Computer vision	The capability that allows a machine to extract meaningful information from images
Context awareness	Interpreting data in the light of situational information such as location, time and device state
CPS	Cyber-Physical System; computation tightly integrated with physical processes under feedback control, with timing correctness central
Cross-compilation	Compiling on a host machine to produce executable code for a different target architecture
Cybersickness	Nausea, disorientation and eye strain in VR, attributed chiefly to visual-vestibular conflict
Decision tree	A supervised learning model that classifies by asking a sequence of simple questions, one per node
Defuzzification	Converting a fuzzy result back into a single, crisp, usable output
Digital twin	A continuously updated virtual model of a physical asset or process, fed by live sensor data
DoF	Degrees of Freedom; 3DoF is orientation only, 6DoF adds position
DTLS	Datagram Transport Layer Security; TLS-equivalent security over UDP, used to secure CoAP

Term	Meaning
Edge computing	Processing performed on or immediately beside the device rather than in a central cloud
Embedded system	A computer built into a larger device to perform dedicated functions, typically resource-constrained and often real-time
Expert system	A knowledge-based AI program that reasons over domain knowledge gathered from a human expert
Feedback (recurrent) network	An ANN in which connections may loop back to earlier or the same layer
Feedforward network	An ANN in which information flows in one direction only, with no loops
Firmware	Software resident in an embedded system's non-volatile memory
Fog computing	Processing at the local network or site level, between edge and cloud
Forward chaining	Reasoning from known facts toward a conclusion
FOTA	Firmware Over-The-Air; remote updating of device firmware over a network
Foveated rendering	Rendering the periphery of a VR image at reduced resolution using eye tracking, saving computation without perceptible loss
Fuzzification	Converting a precise numeric input into fuzzy-set membership degrees
Fuzzy logic	Reasoning that allows degrees of truth between completely false and completely true
Gateway	A device connecting a local, often constrained, network to a wider network, frequently translating protocols
Generative AI	AI that creates new text, images, audio, or video, rather than only classifying or predicting from existing data
GPIO	General Purpose Input/Output; individually controllable digital pins
Hard real-time	A system in which missing a deadline constitutes failure
IIoT	Industrial Internet of Things
Immersion	The objective, technical degree to which a VR system engages and surrounds the senses
IMU	Inertial Measurement Unit; accelerometer, gyroscope and often magnetometer, giving high-rate motion data subject to drift
Inference engine	The component of an expert system that reasons over the knowledge base to reach conclusions
IoMT	Internet of Medical Things
IoT level	A classification of a deployment by node count and by where storage and analysis occur; six levels are defined
ISR	Interrupt Service Routine; the function executed on a hardware interrupt, kept short by design
Knowledge base	The component of an expert system storing domain facts and rules
Locomotion	The mechanism by which a robot moves through its environment (legged, wheeled, or tracked)
LoRaWAN	A long-range, very low-power, low-data-rate wide-area technology in unlicensed spectrum
Machine Learning (ML)	AI that learns to perform a task from data, rather than being programmed with explicit rules for it
Matter	An application-layer smart-home interoperability standard running over Wi-Fi, Ethernet and Thread
Membership function	A function assigning a degree of membership (0 to 1) of an input in a fuzzy set
Microcontroller	A single chip integrating CPU, memory and peripherals, for dedicated control applications
Microprocessor	A CPU requiring external memory and peripherals, for general-purpose computing
Motion-to-photon latency	Delay between a user's movement and the corresponding change reaching their eyes; the principal VR comfort parameter

Term	Meaning
MQTT	A lightweight publish–subscribe protocol over TCP with three QoS levels, retained messages and last-will notification
MR (Mixed Reality)	Real and virtual content coexisting, with virtual objects anchored to and occluded by real geometry
Naive Bayes	A classification algorithm based on Bayes' theorem, assuming input features are independent given the class
Narrow AI	AI competent in one task or a closely related family of tasks; the form of every AI system in current, everyday use
Natural Language Generation (NLG)	The NLP component that produces natural language from a structured internal representation
Natural Language Processing (NLP)	AI concerned with enabling computers to understand and generate human language
Natural Language Understanding (NLU)	The NLP component that maps language into a structured, machine-usable meaning
NB-IoT	Narrowband IoT; a 3GPP low-power wide-area cellular technology in licensed spectrum
OEE	Overall Equipment Effectiveness; a manufacturing metric combining availability, performance and quality
OPC UA	Open Platform Communications Unified Architecture; a vendor-neutral industrial standard with a rich information model
OpenXR	A Khronos Group open standard defining a common application interface to XR devices
Predictive maintenance	Maintenance triggered by measured evidence of deteriorating condition rather than by schedule or failure
Presence	The subjective sense of "being there" in a virtual environment
Priority inversion	A high-priority task delayed by a lower-priority one holding a needed resource; addressed by priority inheritance
PWM	Pulse Width Modulation; controlling average delivered power by varying the duty cycle of a digital output
QoS	Quality of Service; in MQTT, 0 at most once, 1 at least once, 2 exactly once
Regression	A supervised learning problem whose output is a continuous numeric value
Reinforcement learning	Learning a strategy through trial and error, guided by reward and penalty signals from an environment
REST	Representational State Transfer; an architectural style defined by six constraints, with URI-identified resources and a uniform method set
Robot	An artificial, programmable agent that senses and acts in the physical world
RPL	Routing Protocol for Low-power and Lossy Networks; builds a destination-oriented directed acyclic graph rooted at a border router
RTOS	Real-Time Operating System; a small kernel providing deterministic priority-based scheduling
Sensor	A device converting a physical quantity into a measurable electrical signal
Sensor fusion	Combining data from multiple sensors for a result more accurate or reliable than any one provides
SLAM	Simultaneous Localisation and Mapping; building a map of an unknown environment while tracking one's position in it
Smart system	A system that autonomously senses, decides and acts, adapting its behaviour to conditions
Soft real-time	A system in which occasional missed deadlines degrade quality but do not constitute failure
Stereoscopy	Presenting slightly different images to each eye to produce a perception of depth
Supervised learning	Learning from data in which the correct output is already known for every training example

Term	Meaning
Thread	An IPv6-based low-power mesh protocol on IEEE 802.15.4; the network layer beneath many Matter devices
TSN	Time-Sensitive Networking; IEEE 802.1 amendments adding bounded-latency behaviour to standard Ethernet
Unsupervised learning	Learning from data with no known correct output, aimed at discovering structure
V2X	Vehicle-to-Everything communication, encompassing V2V, V2I, V2P and V2N
Virtual Reality (VR)	A computer-synthesised, interactive three-dimensional environment substantially replacing the user's sensory input
Watchdog timer	A timer that resets the system if not periodically cleared by correctly functioning software
WebSocket	A protocol providing a persistent, full-duplex connection over TCP, established by upgrading an HTTP connection
XR	Extended Reality; umbrella term covering VR, AR and MR
Zigbee	A low-power mesh protocol on IEEE 802.15.4, widely used in home and building automation

REFERENCES

Operating Systems (Chapters 1–4)

1. Silberschatz, A., Galvin, P. B., & Gagne, G. *Operating System Concepts* (10th ed.). Wiley.
2. Tanenbaum, A. S., & Bos, H. *Modern Operating Systems* (4th ed.). Pearson.
3. Stallings, W. *Operating Systems: Internals and Design Principles* (9th ed.). Pearson.
4. Dhamdhere, D. M. *Operating Systems: A Concept-Based Approach* (3rd ed.). McGraw Hill Education.
5. Godbole, A. S., & Kahate, A. *Operating Systems* (3rd ed.). McGraw Hill Education.

Information Security, Mobile Commerce, and Cloud Computing (Chapter 5)

6. Stallings, W., & Brown, L. *Computer Security: Principles and Practice* (4th ed.). Pearson.
7. Pfleeger, C. P., Pfleeger, S. L., & Margulies, J. *Security in Computing* (5th ed.). Prentice Hall.
8. Kahate, A. *Cryptography and Network Security* (4th ed.). McGraw Hill Education.
9. Kalakota, R., & Robinson, M. *M-Business: The Race to Mobility*. McGraw Hill.
10. Turban, E., King, D., Lee, J. K., Liang, T. P., & Turban, D. C. *Electronic Commerce: A Managerial and Social Networks Perspective* (8th ed.). Springer.
11. Erl, T., Puttini, R., & Mahmood, Z. *Cloud Computing: Concepts, Technology & Architecture*. Prentice Hall.
12. Rittinghouse, J. W., & Ransome, J. F. *Cloud Computing: Implementation, Management, and Security*. CRC Press.
13. Mell, P., & Grance, T. *The NIST Definition of Cloud Computing*. National Institute of Standards and Technology, Special Publication 800-145.

Internet of Things, Virtual Reality, and Artificial Intelligence (Chapter 6)

14. Bahga, A., & Madiseti, V. *Internet of Things: A Hands-On Approach*. Universities Press.
15. Greengard, S. *The Internet of Things*. MIT Press.
16. Sherman, W. R., & Craig, A. B. *Understanding Virtual Reality: Interface, Application, and Design* (2nd ed.). Morgan Kaufmann.
17. Russell, S., & Norvig, P. *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
18. Rich, E., Knight, K., & Nair, S. B. *Artificial Intelligence* (3rd ed.). McGraw Hill Education.
19. Goodfellow, I., Bengio, Y., & Courville, A. *Deep Learning*. MIT Press.
20. Jurafsky, D., & Martin, J. H. *Speech and Language Processing* (3rd ed. draft). Pearson.
21. Siegwart, R., Nourbakhsh, I. R., & Scaramuzza, D. *Introduction to Autonomous Mobile Robots* (2nd ed.). MIT Press.
22. Ross, T. J. *Fuzzy Logic with Engineering Applications* (4th ed.). Wiley.