

Computer Architectures and Operating Systems (CAOS)

Study Notes

Dr Chris Crispin-Bailey

Revised Summer 2022

Disclaimer: All trademarks used herein are property of their respective owners, and are neither endorsed or subject to any claim of ownership by the author or publisher. Microsoft, Windows, MS-DOS, Apple, Mojave, Android, Chrome, INTEL, ARM, AMD, ASUS, Fujitsu, Cray, IBM, Google, Samsung, Hisilicon, Huawei, Qualcomm, Transcend, Crucial, Infineon, and Seagate, are registered trademarks of their respective owners.

Use of these marks does not imply any affiliation or endorsement between any of these entities and the Course provider. Wherever possible, such marks have been capitalised to identify them. However, the reader should make their own enquiries regarding the status of such instances of brand names and trade-marks that may appear within this text, capitalised or otherwise.

The use of imaginary companies and references may be used occasionally in this text for the purposes of illustration and embellishment of case-study material. Any similarity to actual companies past or present is entirely coincidental. Though information in this text has been carefully checked and is believed correct at the time of publication. No liability is accepted, and no warranty implied or given for any information contained within this document.

Copyright: This material is subject to copyright. Any unauthorised reproduction of content in any form, or storage in a retrieval system, electronic or otherwise, is prohibited without express permission of the author.
The University of York, UK, has express permission to scan and digitally reproduce this material for teaching of students registered on undergraduate and postgraduate degree programmes at the University of York.

Acknowledgments: This document was typeset, in Lato and Lucida Sans typeface, with the help of KOMA-Script^a and Latex^b using the kaobook class^c.

^a KOMAScript: <https://sourceforge.net/projects/koma-script/>

^b LATEX: <https://www.latex-project.org/>

^c Kaobook: <https://github.com/fmarotta/kaobook/>

^d Manuscript Ref. 2.05

Detailed Contents

Detailed Contents	iii
1 Introduction	1
1.1 Introduction	1
1.1.1 Important Notes	1
2 The Fundamentals	3
2.1 Some initial observations	4
2.2 Important units of measurement and specification	5
2.2.1 Kilo, Mega, Giga, and more	6
2.3 A conceptual computer system	7
2.3.1 von Neumann model	8
2.3.2 The von Neumann Bottleneck	9
2.4 The software viewpoint	10
2.5 Putting electronics in the picture	12
2.5.1 Speed versus complexity	14
2.6 Summary	16
2.7 Terminology used in this chapter	17
3 The Heart of The Machine	19
3.1 Processors: past, present, and future	20
3.2 So what does a CPU do?	21
3.2.1 Heat and power	22
3.2.2 What exactly is computation?	24
3.2.3 Toward a better definition	25
3.3 The performance barrier	26
3.3.1 Processor frequency	33
3.3.2 The power of multiples	34

3.3.3	Guesswork: prediction and speculation	35
3.3.4	Branch prediction	36
3.3.5	Speculative execution	38
3.4	CPU internal architecture	38
3.4.1	An instruction 'in flight'	41
3.4.2	A more advanced microarchitecture	43
3.5	A little bit of programming	44
3.6	Extending the Instruction Set	47
3.7	More on instruction microsequences	48
3.8	Summary	49
3.9	Terminology introduced in this chapter	50
4	Computers Everywhere	51
4.1	A machine with many facets	52
4.2	The general purpose computer	52
4.3	General purpose computers	53
4.3.1	The computer in your hand	54
4.3.2	The hidden computer	55
4.4	Being specific	56
4.5	The impact on processor technology	58
4.6	Mainframes and supercomputers	58
4.7	The internet of things	60
4.8	Portable, wearable, implantable	61
4.9	Parallel processing	62
4.9.1	Speedup vs Amdahl's Law	63
4.10	Summary	65
4.11	Terminology	66
5	Making Memories	67
5.1	Memories, past and present	68
5.1.1	Getting physical	70
5.2	Non-volatile memories	71
5.3	Volatile memories	73
5.3.1	Static Random Access Memory (SRAM)	74
5.3.2	Dynamic Random Access Memory (DRAM)	74
5.4	Memory performance strategies	75
5.4.1	Access time and cycle time	76
5.4.2	Memory access protocol	76
5.4.3	Memory width	77

5.4.4	Single-cycle versus Burst-mode access	77
5.4.5	Page mode access	80
5.5	Cache memory	82
5.5.1	Multilevel cache	85
5.6	Cache management policies	87
5.6.1	Mapping policies	87
5.6.2	Replacement policies	89
5.6.3	Write-back policies	90
5.7	Avoiding errors	92
5.8	Programming considerations	93
5.9	The future of memory devices	95
5.10	Summary	97
5.11	Terminology introduced in this chapter	99
6	Building The System	101
6.1	Putting it all together	102
6.2	The system bus vs the dedicated bus	104
6.2.1	Concurrency in bus architecture	106
6.3	Bus standards	107
6.4	A generic system bus	108
6.5	PCI: a very successful Bus Standard	110
6.6	Data rate matching and buffers	113
6.7	USB: Universal Serial Bus	113
6.7.1	Plug-and-Play	116
6.7.2	Hubs and ports	116
6.8	More common bus standards	116
6.9	Industrial and embedded standards	117
6.9.1	I2C (Inter-Integrated Chip) bus standard.	118
6.9.2	CAN bus	120
6.10	Rack-mount, hot-swap, and servers	121
6.11	IO device mapping and IO servicing	122
6.12	Summary	123
6.13	Terminology introduced in this chapter	125
7	Data Storage	127
7.1	Storing Data	128
7.2	Disks and data storage	128
7.2.1	From tapes to disks	128
7.2.2	Modern disks in detail	131

7.2.3	A disk performance example	134
7.2.4	The cost of fragmentation	137
7.3	Disk cache	138
7.4	Solid State Drives (SSD)	139
7.4.1	SSD memory wear	141
7.4.2	Storage density	142
7.4.3	Buyer beware	143
7.5	Storage reliability and fault tolerance	144
7.6	Remote storage	145
7.7	Summary	146
7.8	Terminology introduced in this chapter	148
8	A World of Peripherals	149
8.1	Everyday peripherals	150
8.1.1	Input devices	150
8.1.2	Output devices	152
8.2	Novel peripheral interfaces	155
8.3	Networking connectivity	156
8.3.1	Wired networks	156
8.3.2	Wireless networks	158
8.3.3	Network protocols and overheads	159
8.4	Summary	162
8.5	Terminology introduced in this chapter	163
9	Operating Systems Concepts	165
9.1	The idea of an Operating System	166
9.1.1	A 'bare-metal' system	166
9.1.2	BIOS	167
9.1.3	Low-level input/output drivers	167
9.2	The operating system hierarchy	168
9.3	Choices, choices	170
9.4	The purpose of the kernel	172
9.4.1	Abstraction	173
9.5	The OS as taskmaster	174
9.6	General vs specific, once again	176
9.7	Summary	178
9.8	Terminology introduced in this chapter	179

10 Workload Management	181
10.1 The task in hand	182
10.2 Process ID and housekeeping	183
10.3 Thread management models	184
10.4 Task scheduling	185
10.5 Scheduling algorithms	186
10.5.1 Task scheduling queues	188
10.6 Pre-emptive scheduling	188
10.6.1 Multiple queues	190
10.6.2 Interrupts and polling	191
10.7 Achieving priority	193
10.8 Advanced CPU architecture OS support	194
10.8.1 From scalar to superscalar	194
10.8.2 Multicore	195
10.8.3 Hyperthreading and SMT	196
10.9 Summary	199
10.10 Terminology introduced in this chapter	200
11 Memory Management	201
11.1 Memory management	202
11.2 Memory allocation	202
11.2.1 Memory access control	204
11.3 Virtual memory	205
11.3.1 Paging cost penalties	208
11.4 Inter-Process Communication	209
11.4.1 Pipes	209
11.4.2 Message queues	210
11.4.3 Shared memory	212
11.5 Privileges and restrictions	212
11.6 Summary	216
11.7 Terminology introduced in this chapter	217
12 File System Management	219
12.1 Foreword	220
12.2 The concept of a file system	220
12.3 A file-system example	222
12.4 The boot sector	222
12.5 Volumes and partitions	223
12.6 The root directory	224

12.7	File-system hierarchy	224
12.8	Special file cases	225
12.9	File attributes	227
12.10	The variety of file systems	229
12.11	File compression	230
	12.11.1 File compression example	230
12.12	Automatically compressed file systems	232
	12.12.1 File encryption	233
12.13	File-system resilience	234
12.14	Internal file formats	235
	12.14.1 File structures and encodings	236
	12.14.2 File format tradeoffs: an example	239
12.15	File management	241
12.16	Summary	243
12.17	Terminology introduced in this chapter	244
13	Networks: Being Connected	245
13.1	Foreword	246
13.2	Putting modern networks in context	246
13.3	The OSI reference model	247
	13.3.1 TCP/IP: Another example of layer abstraction	249
	13.3.2 Security	250
13.4	Network structures	250
	13.4.1 Bridging the gap	251
	13.4.2 Switching things around	253
13.5	Routing	254
13.6	Network node identification	256
13.7	Network services	259
	13.7.1 Sockets Interface	260
13.8	Distributed systems	261
13.9	A web-based client-server system	262
13.10	Cloud computing	265
	13.10.1 Traditional in-house infrastructure	265
	13.10.2 The cloud computing option	266
13.11	Compute servers and virtual machines	267
13.12	Virtualisation	269
13.13	Summary	272
13.14	Terminology introduced in this chapter	273

14 Keeping It Safe	275
14.1 Foreword	276
14.2 Security threats	276
14.3 Viruses	277
14.3.1 Virus checkers	277
14.3.2 Virus examples	278
14.4 Architectural exploits	279
14.5 Firewalls	280
14.6 Encryption and validation	281
14.7 Data Resilience	282
14.8 Resilient systems	285
14.9 UPS systems	287
14.10 Resilience and safety	288
14.11 Summary	289
14.12 Terminology Introduced in this chapter	290
Appendices	291
A Glossary of Terminology	293
A.1 Glossary of terms used in this book	293
Alphabetical Index	317

Figures

1.1 Electronic Radio Valve	1
2.1 Simple Computer System Block Diagram	8
2.2 Standard Logic Gates, symbols, and input/output 'truth tables'	13
2.3 Simples examples of Speed and Complexity	14
3.1 Electronic components, ancient and modern	21
3.2 A processor with heat sink	23
3.3 Simplified microarchitecture of a CPU	39
3.4 A typical micro-instruction sequence	41
3.5 Non-pipelined Instruction Microsequence	42
3.6 Pipelined Instruction Microsequence	43
3.7 A Superscalar Microarchitecture	44
3.8 3-Way Multiport Register File	44
3.9 Variations of instruction microsequences.	48
4.1 CPU Examples	55
4.2 NAT Miniature EEG sensor	56
4.3 Supercomputers	60
5.1 Core-store Memory Modules	69
5.2 SIMM and DIMM memory modules	70
5.3 DRAM single read operation	78
5.4 DRAM burst read operation	79
5.5 DRAM fast-page-mode read operation	81
5.6 A scalar microarchitecture with dual on-chip cache hierarchy	86
5.7 Cache Mapping Schemes	89
5.8 Memory Map Example	94
5.9 DRAM memory module capacity trends	96
6.1 Motherboard example	103
6.2 System diagram with more comprehensive bus architecture	105
6.3 A simple hypothetical bus example	108
6.4 Data transfer efficiency on PCI bus	112
6.5 Some additional commonly used bus standards	117

6.6 I2C bus system example	119
6.7 Differential signalling concept	121
7.1 Punched card Program Instruction	128
7.2 Tape System Conceptual operation	129
7.3 Drum Storage concept	130
7.4 Disk unit concept	132
7.5 Modern HDD and SSD units compared	143
8.1 A system with typical Peripherals	151
8.2 Golf Ball Printer	154
8.3 Network Efficiency and Latency	161
9.1 Operating System Hierarchy	169
9.2 Examples of OS types	172
9.3 A desktop OS Process monitoring tool	176
10.1 CPU workload process Monitor	183
10.2 Thread management models	185
10.3 SMT example	198
11.1 Virtual memory page Swapping	207
11.2 Memory usage versus physical memory	208
12.1 Volumes and partitions Example	224
12.2 Example hierarchical file system	226
12.3 Linked files example	227
12.4 Example of file attributes	229
13.1 OSI reference model	248
13.2 TCP/IP model	249
13.3 Simple network example	250
13.4 Bridged network example	252
13.5 Switched network example	254
13.6 Remote central heating network application	263
13.7 Client-Server Heatweb application	264
13.8 Typical virtualisation models	271
14.1 Parity checking Examples	284
14.2 Gaussian MTBF curve	285

1

Introduction

1.1 Introduction

In researching these course notes, it was observed that the topics of computer architecture and operating systems are often taught in two strands, as if they exist in different environments entirely. Architectures for engineers, operating systems for programmers, seems to be the path more often taken. However, these two aspects are more closely interrelated than is often given suitable recognition.

Another observation that one can make, is that often, dividing these two themes is necessary in order to present them as in-depth topics, with the fullest technical detail possible for the hardware or software engineer respectively.

These course notes provide a different approach - we will not pursue every issue in the same depth that separate specialist textbooks might do, but instead attempt to gain a fair grounding in each subject, and understand the relationships between them as they co-exist.

You are encouraged to read independently on all of the presented topics, in order to gain additional insights, context, and depth of understanding.

1.1.1 Important Notes

- ▶ Where possible, example calculations have been simplified for readability and to present methods straightforwardly. This means that some rounding may have been used at intermediate stages in calculations. Reproducing these examples on a calculator with full precision may give slightly different final values. You should not be concerned about this.
- ▶ Throughout these topic notes, and to remain in alignment with the broader course convention, the definition of Kilobytes, Megabytes, Gigabytes, etc as binary rather than decimal quantities, has been adhered to. Further discussion is provided in section 2.2.

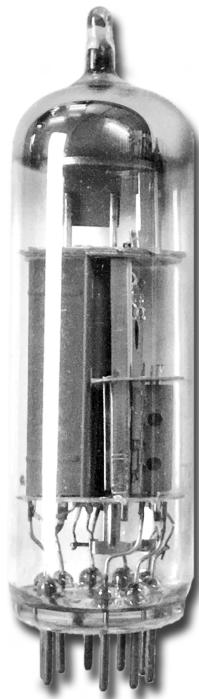


Figure 1.1: Electronic Radio Valve. Some of the earliest computers were built from a few radio valves. We find its modern equivalent, the transistor, on silicon chips, sometimes over 10 billion on one piece of silicon.

2

The Fundamentals



Partial Layout of a 65nm CMOS ASIC design. The image shows the multi-layer metal track patterns, as generated by a computer aided design tool, in preparation for generating manufacturing data for the fabrication of the silicon chip. Large lateral lines represent power grids, smaller lines are interconnections to individual transistors. The area shown covers a region of about 5x5 microns.

Photo Chris Crispin-Bailey 2019.

2.1 Some initial observations	4
2.2 Important units of measurement and specification	5
Kilo, Mega, Giga, and more	6
2.3 A conceptual computer system	7
von Neumann model	8
The von Neumann Bottleneck	9
2.4 The software viewpoint	10
2.5 Putting electronics in the picture	12
Speed versus complexity	14
2.6 Summary	16
2.7 Terminology used in this chapter	17

2.1 Some initial observations

In order to make progress with learning the principles of computer architecture, organisation, systems, and operating principles, there are many concepts that are often assumed to be as good as common knowledge. This can be confusing for the beginner, and of course this is whom this text is aimed at. So we should take some time to establish a few basic ideas before jumping in to details.

By the end of this text, we might wish to be able to understand and perhaps evaluate computer systems in terms of their components, their behaviour, and their performance. Inevitably we will need to simplify some concepts and details, and it is for the enthusiastic reader to go beyond that level of basic understanding, and fill in the greater level of expertise that is possible with further study.

Let us begin by asking what we think a computer system is: It is undoubtedly a collection of hardware and software. But that only gives us the thinnest veneer of a definition. So let us probe a little deeper. Exactly what do we mean by 'hardware'? We can attempt to define hardware in terms of a system of components and characteristics, perhaps along the following lines:

Key elements and characteristics of Hardware

- ▶ It includes **electronics**, **chips**, **circuit boards**, **cables** and **connectors**.
- ▶ But also, by extension, it includes **peripherals**: the things we plug into our computers to build larger and more complex configurations.
- ▶ Some peripherals are all but essential. Without a **keyboard** and **mouse**, or at least a **touch-screen**, most computer systems have limited usability for the average user.
- ▶ All electronic devices consume **power** and generate **heat**, and most include at least some **digital electronic** circuits.
- ▶ Digital electronics operate on the level of **binary numbers** and **binary signals**.
- ▶ Many of the performance limitations of computers relate to the **speed** at which circuits can manipulate these binary signals, and many others relate to how quickly binary information can be **moved** from one place to another.

Software is also a highly complex composition of elements, with varied purposes, capabilities, and characteristics:

Key elements and characteristics of Software

- ▶ **Applications** that you buy and **install** on your laptop/phone/- computer.
- ▶ **Utilities** that are part of the operating system, be it WIN-DOWS, Linux, or another option.
- ▶ The **operating system** itself, and software embedded within peripherals, typically known as **firmware**.
- ▶ Low level software that is part of the standard computer motherboard, known as **BIOS (basic input/output system)**.
- ▶ Software that runs as part of a **web-page**.
- ▶ Software that **compiles** (converts) program **source code** into processor **machine code**.
- ▶ Software that **interprets** code written in one style into code in another style, to permit it to run on a given system.
- ▶ Software programs that you write yourself and then run on the system, referred to as **user programs**.
- ▶ Software that we **do not want**, such as **viruses**, that interfere with normal system operation.
- ▶ Software that is installed to **detect and prevent viruses** from interfering with normal operations.

Clearly, there are many aspects to both hardware and software, and we will need to bring all of these things together to give a complete picture of a modern **computer system**. We will begin to sketch out the basics in this first chapter, and then build toward a compete picture.

2.2 Important units of measurement and specification

We will encounter a number of possibly unfamiliar units of measurement when we begin to study computer systems, and we will also encounter parameters that are widely used to specify systems behaviour and requirements. It is worth reviewing a few of these right away, to get a fair background appreciation of some of these before we begin to use them in earnest.

[1] Standard scientific units for power in computer systems are typically defined as watts (W), milliwatts (mW) (1×10^{-3}), microwatts (uW) (1×10^{-6}), and occasionally even nanowatts (nW) (1×10^{-9}).

Power: Electrical power consumed by a computer component or system, often measured in watts (W), milliwatts (mW), microwatts (uW), and even nanowatts (nW). [1].

Frequency: A measure of how often a given activity happens in any one second. In a computer system, typically, this refers to a basic operating cycle called a clock cycle. The basic measure of **frequency** is **Hertz (Hz)**. Most computer systems operate at very high frequencies, and therefore MegaHertz (MHz), and GigaHertz (GHz) are normal units of measurement.

Clock Cycle: The frequency of activities is typically regulated by a **clock signal**. This is a repetitive pulse that dictates each new event cycle, and operates at a frequency as mentioned above.

Data Capacity: The number of data items a device or system can manage, or store. Typically, data capacity is quoted in large multiples of bits or bytes, and typically using binary measurement units rather than decimal (see below).

Data Rate: The quantity of data that can be transferred from one place to another in a given amount of time, usually one second. For example a connection between two components may have a data rate of 100 million bytes per second (or about 95.3 binary Megabytes/sec).

2.2.1 Kilo, Mega, Giga, and more

Defining terms is important, as is recognising their standard notations. A **bit** is a single binary value and a **byte** is a group of 8 bits. Traditionally, a **kilobyte** (KB) is 1024 bytes, a **megabyte** (MB) is 1024×1024 bytes, and a **gigabyte** (GB) is 1024 megabytes. Computer systems even have capacities as high as 1024 gigabytes: this is known as a **terabyte** (TB). Exceptionally, and in particular for super-computer systems, we may have many hundreds or thousands of terabytes of data memory, or storage. A further subtlety in notation, to be well noted, is the use of **MB** to denote **megabytes**, and **Mb** to denote **megabits**. Read and write these units with care whenever possible. Generally in this material we will use megabytes, megabits, or perhaps Mbits for brevity, to avoid opportunities for such misunderstandings whilst we learn together.

This terminology can cause initial confusion for non-computer scientist practitioners, since the standard scientific units of kilo, mega, giga, etc are more generally based upon **decimal** powers 10^3 , 10^6 , 10^9 , etc. This has become more of a problem since the introduction of new base-10

definitions of Kilo, Mega, Giga, etc^[2], in preference to the still widely used 'deprecated' binary definitions. This new standard has not yet become universally prevalent, and much of the computer systems world, including older documentation and huge volumes of legacy software and hardware, is still based upon the older definitions.

However, the **binary** version of these units correspond to 2^{10} , 2^{20} , 2^{30} , giving slightly different values. A binary megabyte for example is 1,048,576 bytes, and a million bytes is 95.3 *binary* Megabytes^[3]. For a more complete picture of new and old standards, see TUTORIAL ??.

IMPORTANT !

Note that in this material we default to using the well established approach of using binary megabytes, gigabytes, etc, unless stated otherwise. This is because they are more relevant to physical hardware structures such as memory devices, disk storage sectors, and so on, as we will find in later chapters. When base-10 units are appropriate, for instance where we discuss bus/network data rates, terms such as 'millions of bytes per second' will be used where possible.

Where you find other texts or online sources referring to kibi, mebi and gibi, these are the modern reference terms for binary kilo, mega, and giga quantities used in this material.

[2] In particular the definitions made under IEC 80000, where kilo= 10^3 bytes, mega= 10^6 bytes, giga= 10^9 bytes, and so on.

[3] To convert from millions of bits or bytes into binary megabytes, simply divide the subject figure by 1.024 twice, and for billions of bytes to gigabytes, divide three times.

2.3 A conceptual computer system

The basic model of a computer system is traditionally represented by a block diagram showing the most important components. Of course every real-world system is different, and may have additions, novel variations to the normal practice, etc. This is especially so when computers are **application-specific** rather than **general-purpose**. General purpose computers are designed to perform adequately across a wide range of uses, but they are rarely optimal in any of those cases. Compromises are made to make the computer flexible and widely applicable, and potentially more cost effective. Where a computer system needs to be highly efficient at one well defined task, then application-specific computer designs can achieve very high performance in that one niche area.

So what does a general purpose computer consist of? Let us take a look, starting with the block diagram model given in Figure 2.1. A very simple computer system, as a first-order model, might be presented thus. It

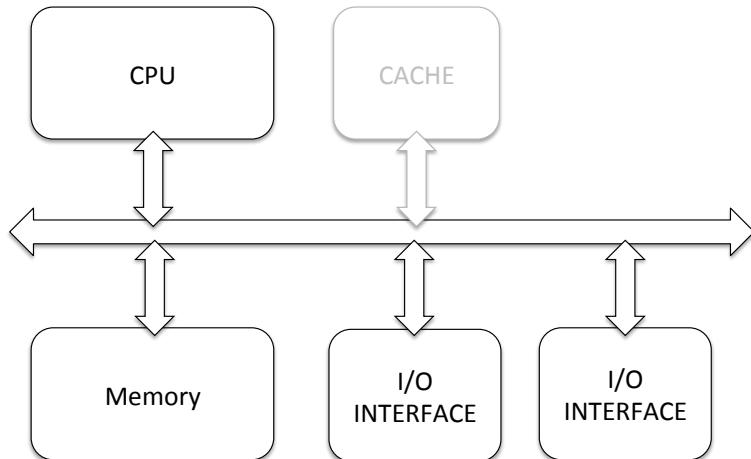


Figure 2.1: Block diagram of a general purpose computer system. Diagram shows CPU: Central Processing Unit (top left), cache memory unit (top right), Main memory (bottom left), and several IO interfaces (bottom right).

consists of a **Central Processing Unit (CPU)**, often simply referred to these days as a **processor**, sharing a connection to a **system bus** with a number of additional modules. These include the essential **memory** and **input/output interface(s)**.

2.3.1 von Neumann model

As far back as 1945, the computer scientist **John von Neumann** defined a computer architecture as consisting of a CPU, a memory, and input/output devices. This has been the most frequently used model for computer systems ever since. It is amazing to think that this is a definition that is rapidly approaching its 80th year. Connecting the individual modules of our computer system is a **System bus**. This may well be a new concept for the reader, so let us take a moment to define what a bus is before we continue:

Definition 2.3.1 *Bus: Simple Definition*

A bus generally consists of a group of wires, working together to achieve a communication of information between several devices in a coordinated fashion.

Busses allow many devices to share one communication route for information. However, it is important to note that only one device can control the bus at a time (the **bus master**). This is a limitation that we need to

be aware of, as it has significant impact for performance. A system bus is thus just a particular type of bus, in which some of the wires designate an **address**, some designate **data**, and some designate essential **control signals**. We will find out what these groups of wires represent later in detail.

The system bus allows the CPU to communicate with other essential modules. These include: the memory, where data and instructions are stored, and suitable electronic circuits to permit IO devices to be connected to the system. An **IO device** is an input/output device, for example, a **keyboard** is an input device, and a **video screen** is an output device. We will study these devices and their related **electronic interfaces** in more detail in a later chapter.

There is one more module in our diagram: the **cache**. Cache is a special kind of memory, which is much faster than normal memory. This was not explicitly defined in von Neumann's original model, but is worth mentioning at this point in passing. We will see later that this special fast memory allows a computer system to boost its performance by exploiting the way programs and data are accessed in the memory space the computer uses.

So, we can say at this stage that, in the simplest computer system, all devices connect to a system bus, the CPU is always the bus master, and all devices appear as if they are part of the visible memory of the computer system. This allows for very straightforward programming models to access devices and modules within the system, though not necessarily the most optimal performance.

If we analyse any modern computing system, it will have at least the same minimum features as our simple **von Neumann model**. A system without a CPU cannot do computations, a system without memory cannot store a program, and a computer that has no input/output capability cannot accept data to process or give out its results. So, all of these components are regarded as essential building blocks.

2.3.2 The von Neumann Bottleneck

Although the von Neumann model is perhaps one of the most successful concepts in computer design, it does have some drawbacks. Consider what happens when the CPU is busy **fetching** an **instruction** from memory. All of the bus wires will have signals present, and therefore they cannot be used by another device in the system.

[4] This is indeed possible in some systems. The concept is known as direct memory access, or DMA. DMA capable IO devices are able to take over bus control and act like a master bus controller in order to coordinate data transfers between two or more bus entities.

If a CPU wishes to fetch data from memory, it cannot also fetch an instruction from memory at the same time. Likewise, if an IO device was capable of accessing memory directly^[4] then it would similarly be prevented from doing so whenever the CPU was already accessing memory. This constraint, due to many devices wanting to use one shared bus, is known as the **von Neumann bottleneck**.

In an attempt to alleviate the von Neumann bottleneck, designers invented an alternative architecture, known as the **Harvard Architecture**. In this system, there are two busses, one for program memory, and a second bus for data memory and IO. This permits instructions and data to be accessed **simultaneously**, permitting a speed gain.

However, in a modern computer system there are actually a wide variety of options for multiple bus systems, and covering all of them at this point may be a distraction.

There are many more fundamental questions we have yet to ask: what exactly is a CPU?, how does memory work?, how does cache give a performance boost?, how do IO devices connect to computers?, and how do they behave? We will be turning our attention to these questions in the following chapters.

2.4 The software viewpoint

As we discovered earlier, the concept of software covers a very diverse set of possibilities. Understanding some of the more fundamental concepts will help us in our journey toward a fuller understanding. In this text we will explore many, if not all aspects of the major software elements of a computer system. But let us start by familiarising ourselves with some basics.

Incredibly, some of the earliest examples of software, if we stretch the definition to its limit, include **punched cards** used in automated weaving machines, dating back to the 1700's. The **Jacquard loom**, used such cards to designate the weave pattern for each line of the cloth being woven. In effect, each set of holes punched in a card roll acted as an **instruction** to a weaving machine. The punched card rolls allowed a rather complex set of weaving instructions to be performed by an automated machine, resulting in accurate reproduction of a complex pattern. There is a lot to see here in a modern definition of software, where we would arguably accept that software is a series of instructions relating to a complex set of actions to be performed by a digital circuit (the

CPU), and rather than manipulating threads of cotton, we manipulate data values.

As we have noted, software can exist in many forms. At the simplest level, we can divide software into **source code**, which is in some way intelligible to a human, and **machine code**, which is nothing more than a series of binary numbers, and rarely understood by visual inspection by a programmer. The first systems had to be **programmed** in machine code, because nothing else existed. For most purposes this was a laborious and undesirable task, and made the creation of highly complex programs quite difficult. As a result, programming techniques had to evolve.

Programming languages began to emerge in the late 1950s and evolved rapidly over the next 30 years. Early languages such as **ALGOL** and **COBOL** are remembered fondly by ageing programmers, but the diversity of languages available now is huge by comparison^[5]. Most modern computing languages, such as **Java**, **C**, **Python**, and so on, are founded upon principles of these first languages in some form or another.

In modern computing situations we use two types of so-called **high level languages (HLLs)**. The first type performs **compilation**, using a software utility called a **compiler**. A compiler automatically converts high level language statements into sequences of low-level machine code. It is the machine code that is executed on the computer, not the high level language statements typed in by the programmer. If the compiler is very good, then it will generate very efficient code.

A second approach is to perform **interpreted execution**. An **interpreter** (another kind of utility program) takes the programmer's program statements, or a condensed version of them, and reads through that sequence of high level language statements. It then executes a predefined piece of low level code for each statement it recognises.

Meanwhile, we still have a third option: to write programs in low level machine code, using a textual representation of each instruction, known as **assembly language**.

But didn't we just say that low level programming is laborious and usually undesirable? This is true, but there are situations where the very compact and highly efficient programming style of assembly language is still useful, especially where very low level activities such as directly operating IO device signals at very high speeds is required.

[5] It is hard to know if there is a definitive answer to this. Some have suggested there are well over 500 languages in existence that have been used widely enough to be considered as candidates, but of course this number increases every year.

2.5 Putting electronics in the picture

In this text, we try to stay away from deep electronics knowledge. A full understanding of the physics of **transistors**, **semiconductors**, **logic gates**, and the challenges of designing very high speed circuits, would require significant engineering knowledge. However, we cannot have a complete picture of a computer system without some basic understanding of how digital circuits achieve computations. Let us therefore review a few key concepts here too, in preparation for our journey of discovery.

[6] Transistors have many properties, and are useful for amplifying and manipulating electronic signals of all kinds. When used in one particular mode, they operate in a way that approximates an on-off switching function.

- ▶ **Transistors:** Invented in the late 1950's, a transistor is a kind of switch, at least as far as digital electronics is concerned^[6], and when properly used they can operate in a binary signalling regime (on and off, one and zero).
- ▶ **Transistor Operation:** A simplified description of transistor behaviour in switching mode might be as follows: A typical transistor uses a control signal (**the Gate**) to control the flow of a signal from the input (**the Source**) to the output (**the Drain**). However, multiple transistors can be combined to create logical switching functions.
- ▶ **Logical Operation:** Imagine that our Gate control signal can switch from zero to one, and by combining two transistors in the right way, we can create a switching arrangement that produces corresponding outputs one and zero. That is to say, the output is the opposite of the input. This is known as an **inverter**, or a logical NOT operation: the simplest logic gate. The output is NOT the input, if you wish to think of it in those terms.
- ▶ **AND and OR:** We can also combine transistors in such a way that they create gates with two inputs. Then the circuit can be arranged such that a signal output can only be binary one when both inputs are one (the **AND gate** function, where input-a AND input-b must both be one at the same time to generate an output of one). Likewise, a slightly different arrangement causes the circuit to generate a one if either input is one (generates one if input-a **OR** input-b are one), creating an **OR gate**.

What we have just described is the behaviour of three very simple **logic functions** with the definitions of AND, OR, and NOT. These circuits are referred to as **logic gates**. So we typically talk about the **AND gate**, the **OR gate**, and the **NOT gate** (or just an inverter). A further gate, the **XOR gate** performs an **exclusive-or** function, where an output is one only when any one (but not both) inputs are one. These components have diagrammatic symbols, and their behaviours can be defined by tables of

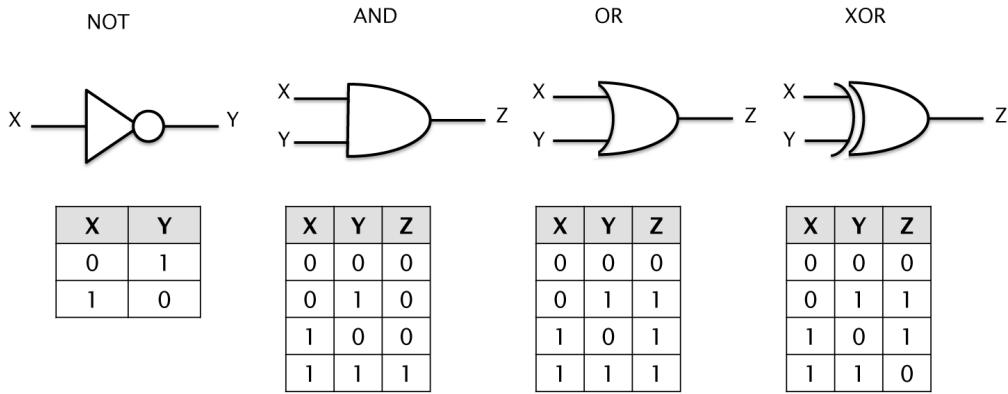


Figure 2.2: Standard Logic Gates, symbols, and input/output 'truth tables'. Diagram shows NOT gate (Left), XOR gate (right), AND gate (centre left) and OR gate (centre right). Each gate has one or more inputs, labelled x,y, etc., and a single output, z. Output signal behaviour is defined by the combination of input(s) and the type of gate.

inputs and outputs known as **truth tables**. Figure 2.2 shows the symbols and truth tables for our four logic gates.

Any of these gates can be combined with an inverter to give an opposite function. AND gate becomes **NAND gate**, OR gate becomes **NOR gate**, XOR gate becomes **NEXOR gate**^[7], and an inverter becomes a **buffer** (a gate that has no logical effect^[8]). Without going into too much detail, mixtures of these eight logic gates can also be combined into more complex circuits, including circuits that add binary bits together (arithmetic), or perform other mathematical operations such as multiplying, comparing, and so on. These circuits are called **combinational circuits**.

Meanwhile, logic gates can also be used to build **counters** and other **control circuits** to permit a computer chip to control sequences of events. These circuits require a time pulse to control their repetitive behaviour: this is the **clock signal**. A clock is simply a repetitive on-off-on-off signal, like a clock tick-tock-tick-tock. Because these circuits use a clock to synchronise their behaviour, they are known as **synchronous circuits**.

Finally, by using logic gates in a particular way, such that some of their outputs feed back as input signals to the same circuit, it is possible to have a signal circulating in a loop as part of the electronic behaviour of the circuit. This allows information to be retained, and the result is sometimes referred to as a **data latch**, or a **flip-flop**, which can store a zero or one. A single bit of storage is not particularly useful, but if we combine eight latches, for example, then we have created an **8-bit**

[7] NEXOR also often referred to as XNOR

[8] This may seem pointless, but it permits certain capabilities, such as delaying a signal, among a few other purposes.

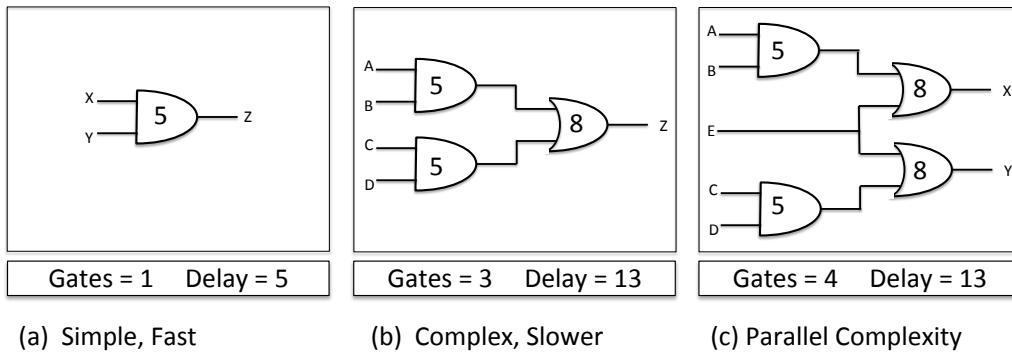


Figure 2.3: Simple Examples of Speed and Complexity. Figure (a) shows a simple fast circuit. Figure (b) shows a more complex and slower circuit. Figure (c) even more complex, but **not** slower. The numbers indicate the signal delay caused by each gate. For instance, '5' might represent 5 nanoseconds.

register, a component that can store an 8-bit binary number: one byte of data. This binary storage principle is the basic embodiment of **computer memory**. At this stage, if you are not familiar with the idea of **Binary** and **Hexadecimal** number systems, then this is something to address now. Do some research, or read the provided material in Tutorial ??.

Hopefully from this description it should be apparent that transistors allow us to create logic gates, and logic gates allow us to create circuits that can perform arithmetic, store information, and control sequences of events. These will all appear to be very relevant in the next chapter.

[9] The idea that more gates means slower circuits is based upon the idea that logic gates are cascaded together, such that their delays accumulate from end to end. Actually, some gates can operate in tandem (side by side) so this is not an entirely concrete truth, though in general terms it is likely.

2.5.1 Speed versus complexity

Nothing is free. All of these logic gates and logic circuits we are talking about **consume power** and **generate heat**. The more gates that are combined, the more power is consumed and the slower the circuit becomes^[9].

Slower circuits are revealed when we look at the **clock frequency** of a system. A clock should pulse as fast as the circuit can tolerate, to make it work as fast as it can. But if the circuit is slow then we must also have a slower clock. This is one reason why so many different CPUs have very different clock frequencies. What we have just described is the dilemma of **speed versus complexity**: more complex circuits might be better at performing a particular function. But, if they are also slower, then does this actually mean they do any more useful work?

Figure 2.3 illustrates, in a very simplistic way, the main concepts of speed versus complexity. Consider Figure 2.3(a), where a single gate is undoubtedly a simple circuit. Suppose that this logic gate has a **signal delay** of 5ns, then we can say that the complexity is low and speed is high^[10]. Now consider Figure 2.3(b) which has three gates, so the complexity has increased. Meanwhile, any path through the circuit from input to output requires 13ns. In other words, it is slower. Finally, Figure 2.3(c) has four gates (even more complex) but, surprisingly, it still has a delay of only 13ns (more complex but *not* slower).

Circuit (b) is more complex, and slower, than circuit (a). Intuitively you might also expect circuit (c) to have increased delay compared to (b). After all, circuit (c) is certainly more complex than circuit (b). Whilst it is true that increased complexity often, even typically, causes increased delay, it is sometimes possible to have more gates but not slow down the circuitry. Indeed, in circuit (c) what we have done is use parallelism, because there are actually two separate circuits operating side by side. This is a key principle underlying the idea of multicore processors, a topic we will explore in later chapters.

Speed versus complexity is one of the big arguments in computer architecture, and one that has not yet reached a conclusion. It has had a major influence on processor design, not the least of which was the **RISC versus CISC** debate. During the 1980's, processor designs were heading toward increasing complexity, following the concept of **Complex Instruction Set Computers (CISC)**. However, it was observed that many of these instructions were rarely used effectively. This spurred a new concept: **Reduced Instruction Set Computers (RISC)**, where a streamlined but very efficient set of instructions was implemented in the processor design, meaning less complex circuits and higher clock speeds.

The same argument has influenced the idea of **multicore processors**: the idea of putting many CPUs on a single chip, to increase performance. Instead of one processor with ten times the complexity, why not have ten processors just like the ones you already have? (ten circuits operating in parallel). As the number of transistors available on a chip at a given price^[11] continues to increase, these arguments will continue to evolve too.

[10] When we talk about speed, we think about frequency (MHz, GHz, etc.), and delay is expressed as time (microseconds, nanoseconds, etc.). These are related, since $f=1/t$ and $t=1/f$. Therefore, if $t=5\text{ns}$, $f=200\text{MHz}$, compared to $f=77\text{mHz}$ if $t=13\text{ns}$.

[11] transistor count is one measure, another term often used is 'silicon real-estate', and silicon real-estate per dollar is therefore a measure of cost versus complexity.

2.6 Summary

We have just explored a few basic ideas relating to computer systems. We have considered what the scope of hardware and software encompasses, and we have seen that those two words innocently hide a very large field of knowledge. We have also learned some basic concepts of computer systems architecture, explored some simple aspects of digital electronic circuits, and defined some key terms of reference.

With our beginner's approach, we should not attempt to cover all of these aspects in expert depth, and that indeed is not our purpose. However, in the following chapters we will expand many of these ideas to gain deeper knowledge, hopefully without information overload!

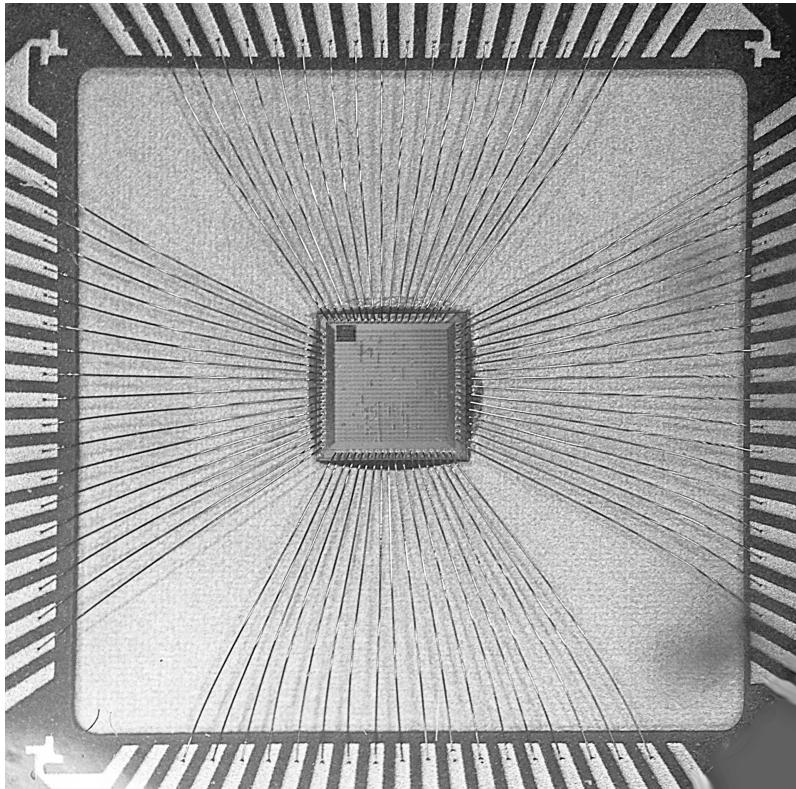
2.7 Terminology used in this chapter

Address	Application-specific
Applications	Assembler
Assembly language	BIOS
Cache	Clock signal
Combinational circuit	Compiler
Control signal	CPU
Data	Data capacity
Data latch	Data rate
Firmware	Flip-flop
Frequency	General-purpose
Hardware	High level language
IO device	Interpreter
Inverter	Logic gate
Logical AND	Logical NOT
Logical OR	Logical XOR
Machine code	Operating system
Peripherals	Power
Register	Software
Source code	Storage element
Synchronous circuit	System bus
Transistors	Truth table
Utilities	Virus

These terms are defined in the glossary, Appendix A.1.

3

The Heart of The Machine



65nm CMOS ASIC, NOMAD Microprocessor, mounted in glass-covered test package. Photo: Christopher Crispin-Bailey 2019.

3.1	Processors: past, present, and future . . .	20
3.2	So what does a CPU do?	21
	Heat and power	22
	What exactly is computation?	24
	Toward a better definition	25
3.3	The performance barrier	26
	Processor frequency	33
	The power of multiples	34
	Guesswork: prediction and speculation.	35
	Branch prediction	36
	Speculative execution	38
3.4	CPU internal architecture	38
	An instruction 'in flight'	41
	A more advanced microarchitecture	43
3.5	A little bit of programming	44
3.6	Extending the Instruction Set	47
3.7	More on instruction microsequences	48
3.8	Summary	49
3.9	Terminology introduced in this chapter	50

3.1 Processors: past, present, and future

In this chapter we will review the origins of electronic computer systems, and discover the key concepts behind the workings of the processor: the heart of the machine.

Since the mid 1970's, the vast majority of computer systems have been built around a key component: the single-chip **microprocessor**. Prior to this time, computer systems were almost always built from a large number of **discrete components**. These were often individual transistors in small metal canisters, or very simple digital logic components implemented individually as very simple logic chips^[13] also known as **integrated circuits**. But even further back, computer circuits were built from **radio valves** filling hundreds of racks, and a multitude of cabinets.

In terms of dimensions, a radio valve^[14] is many times larger than a transistor, and a transistor is many times larger than a logic IC. A microprocessor is many many times more complex than a logic IC, yet of similar size. This is illustrated nicely in Figure 3.1 where some of these components are shown together at the same scale.

Consequently, the march of progress in computer systems has been characterised by unrelenting miniaturisation: from valves to transistors, from transistors to IC's, and finally from IC's to microprocessors. Throughout this evolution of these important fundamental building blocks, the concept of a central processing unit (CPU) has been a constant feature.

A CPU in a 1950's computer might have consisted of many yards or metres of shelves full of radio valve racks, all wired together into a complex circuit, consuming kilowatts of power and generating more heat than an electric fire. In contrast, a modern day CPU is represented by a single microprocessor IC, may consume only watts or tens of watts, and yet such a chip is unimaginably more complex.

Whilst early CPU's might contain several thousand radio valves (in some way analogous to a few thousand transistors), a modern microprocessor may contain several thousand million transistors, and yet represent much more than just a primitive central processing unit. It is even possible for a complete computer system to fit on a single chip, with little else needed but a keyboard and screen.

Interestingly, heat and power are still a problem for modern CPU's, and this has a major impact on the design of modern computer systems and

[13] The most successful of these was the TTL logic chip family, which is still available and still used in digital electronics.

[14] As its name suggests, a radio valve was a component originally designed for driving radio amplifiers, but was also found useful in a variety of electronics applications, including implementing simple boolean (binary) logical functions.

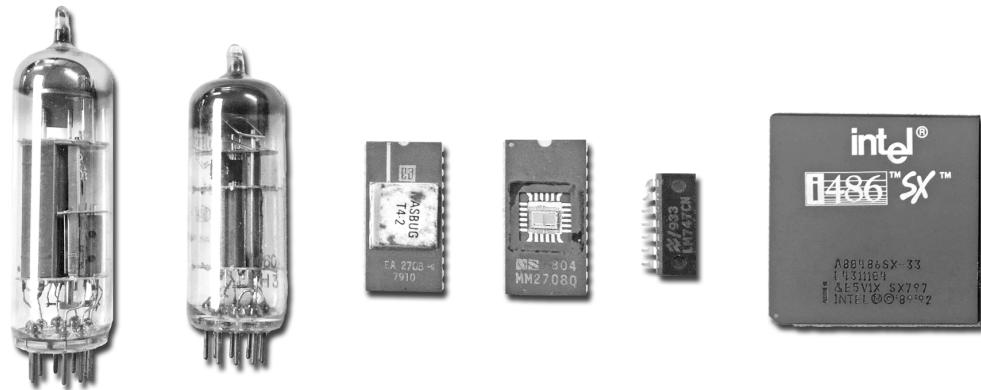


Figure 3.1: Electronic components, ancient and modern. Showing (left) radio valves roughly equivalent to one or two transistors, (middle) integrated logic chips with potentially of the order of 1 million transistors, (middle right) small IC chip package typical of simple logic gate chips containing typically 100's of transistors, and (far right) one of the first widely used one-million transistor microprocessors (1998).

their components, whilst influencing many of the trends that processor design is now following.

3.2 So what does a CPU do?

One of the most fundamental questions in computer architecture is: how do we compute information? The simplest answer we can give is: **the CPU performs computations**. But how does this happen?, why does it consume power, why does it take time? These are just a few of the questions we would like to know more about. So let us begin with a conceptual description of a CPU:

Definition 3.2.1 CPU: Simplified Definition

A **CPU** is a digital electronic circuit, usually built from thousands or millions of transistors. It can hold a small amount of temporary data and perform mathematical operations upon them. A CPU can also sequence operations according to a program it reads from memory, step by step.

So, a CPU performs a sequence of (often mathematical) operations upon temporary data it holds, and this sequence is dictated by a program found in memory.

We can expand upon this description quite considerably, and we will do so later, but let us start with this simple case and understand what it means for computer design and operation. In particular we will examine design goals and design problems, and see how these are addressed in modern computer designs.

3.2.1 Heat and power

It is important to appreciate that all digital circuits consume electrical power. Just as a light-bulb requires power to generate light, a digital circuit requires power to perform its function. The amount of power consumed depends upon several factors, not least of which is the kind of **silicon technology** the chip is manufactured with.

Another factor is how fast the circuit operates. Just like a car engine, the faster it runs, the more energy it consumes. Every time a transistor switches on or off, it consumes some power, known as **dynamic power**. The faster the on-off transitions occur, the more power is consumed. And what happens to all of this power? Classical physics tells us that energy cannot be destroyed, only transferred, therefore when a circuit consumes energy, one of the unwanted by-products is heat^[15]. Incidentally, silicon chips also consume power when the transistors are idle, like a car motor ticking over whilst waiting at a road junction. This is known as **static power**.

[15] There are however other by-products. For example circuits generate radio noise as well as heat. This can be undesirable, and theoretically it can even been exploited as a security risk.

[16] Moore's Law: For a more detailed discussion try Wikipedia 'Moore's Law'.

[17] A micron is one millionth of a metre, or one thousandth of a millimetre

When digital circuits are manufactured, they make use of miniaturisation of transistors to pack many components into a tiny chip, often only a few millimetres in size.

As **Moore's Law**^[16] has predicted successfully for many decades, the number of transistors on a given chip area doubles every two years. This can only be achieved by halving the dimensions of transistors at each step, and therefore reducing chip area per transistor by four. Only 20 years ago the critical dimension for transistor design was still as large as a 1 or 2 microns (1-2um), whereas these days the figure can be as small as 0.01 microns, or 10 nanometres (10nm)^[17]. This dimension is known as the **feature size** and we refer to each incremental advance in silicon technology as a **technology node**.

Observing the state of processor technology (in 2019), it can be seen that processors are manufactured at a range of technology nodes, including 90nm, 65nm, 45nm, 22nm, being quite frequently used across the semiconductor industry at the time of writing.

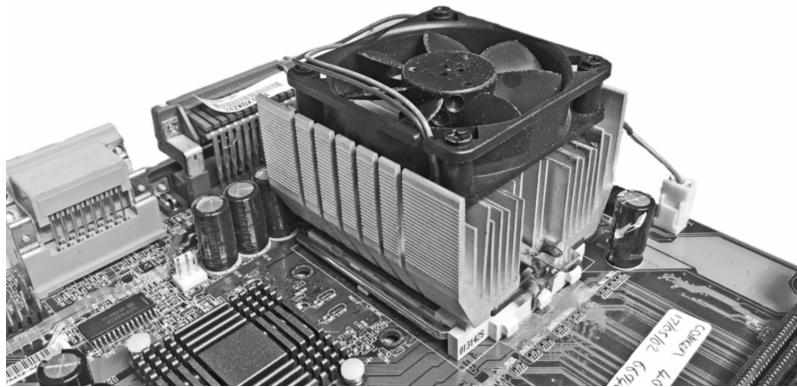


Figure 3.2: A processor with heat sink. Showing main CPU top-centre, with large aluminium multi-finned heatsink and forced air cooling fan. Lower left also shows smaller passive heatsink providing thermal dissipation for an auxiliary controller chip.

The power consumed by a transistor at each technology node reduces with size, so we might imagine that power is a diminishing problem for a digital circuit such as a processor. However, that is not the case. Why is this? The answer is that although transistors get smaller and smaller, designers want to use more and more transistors per chip, and those transistors operate faster too. The complexity of processors is thus an ever increasing feature of computing.

Another consequence of this trend is that heat generated by circuits increases too. Where we have smaller circuits consuming more power and thus generating more heat we can say that **power density** and then **thermal density** increases. This is recognised as a serious problem. This is why a modern processor system is often cooled by a large **heat-sink**, a module designed to draw heat away from the processor so that it does not overheat. An example of a heatsink is shown in Figure 3.2, where the example CPU requires a substantial **metal-finned heatsink** supplemented with a **cooling fan**. Some heatsinks are further assisted by liquid cooling systems.

Let us consider an example to put this in perspective: An INTEL Core i7 8700 3.2GHz chip is designed to run at up to 67 Watts with the recommended heatsink. So 15 of these processors could consume as much power as a 1kilowatt electric fire, and generate a substantial proportion of the same heat. That is a lot of power to pay for and a lot of heat to deal with.

The problem of heat also dictates some considerable design limitations. **Server farms**, huge warehouses full of racks of computers, produce so

much heat that they must be cooled by advanced cooling systems, and these in turn consume power. Therefore, the total power consumed, and thus the running cost of the servers may be much higher than just the CPU power consumption itself.

Other systems are physically constrained within modules: a laptop case, a system rack, inside a handheld phone or device. These systems cannot generate more than a certain amount of heat because they are not able to get rid of it faster than a certain rate. You may have noticed your smartphone getting warm when heavily used. This might well be the limiting factor in performance, and not how powerful the CPU itself is, or the size of the battery.

3.2.2 What exactly is computation?

Now that we have established that digital circuits are what makes a CPU work, but at the cost of heat and power, what exactly are these resource-hungry computations doing?

A typical CPU will perform a range of different computational operations, also known as **instructions**. These operations act upon data in different ways, such that if multiple instructions are combined into a **program sequence**, almost any computational task can be performed. Let us survey some common instruction characteristics:

Some characteristics of instructions

- ▶ Data is held in temporary storage components within the CPU, called registers. Instructions operate upon these register contents.
- ▶ Many operations are directly mathematical: add, subtract, multiply, divide, and so on.
- ▶ Some of these instructions operate only on integers (whole numbers) whilst other operations may be able to act upon floating point or fixed point numbers. Not all processors support both kinds of computation.
- ▶ Some instructions are logical operations. These perform boolean logic operations on data.
- ▶ Some instructions transfer data to and from memory. These are known as Load/Store operations.
- ▶ Some instructions test conditions and choose which part of the program to execute next.

We have just introduced some new concepts here: for example, what does **execute** mean? A program consists of a number of instructions, and a CPU **executes** (in other words, performs) each instruction in turn. In order to execute an instruction, the CPU must first **fetch** it from memory, where our program is usually stored. Fetching an instruction, and then executing it, is known as the **fetch-execute cycle**. The CPU repeats this over and over indefinitely (or until such time as it reaches a halt instruction to tell it to stop).

If a CPU simply executed one instruction after another in a continuous list, we would have what is known as a **linear program sequence**. However, very few programs can be written like this in their entirety. Actually most programs include choices and decisions which affect which part of the program is executed next.

Consider the comment that 'some instructions test conditions and choose what part of the program to execute next'. These are known as **program-flow instructions**. The flow of a fetch-execute cycle sequence, as it moves through memory, is generally **non-linear** (more accurately it is linear for short sequences), interspersed with a number of jumps to other parts of the memory where the next short linear sequence is located. Program-flow instructions are therefore frequently encountered.

3.2.3 Toward a better definition

We have gathered quite a lot of information about our initial question. So let us try to answer this again: What is a CPU?

Definition 3.2.2 CPU: Expanded Definition

A **CPU** is a digital electronic circuit, manufactured from a silicon technology defined as being at one particular technology node, consuming a certain amount of power, and generating a certain amount of heat during its operation. A CPU fetches program instructions from memory using a concept known as the fetch-execute cycle, and as each instruction is fetched, it is executed, performing one of a number of possible operations upon data held within the CPU registers. These operations include mathematical and logical operations, and operations that change the flow of the program to execute parts chosen by the result of testing conditions encountered during a program sequence.

This is quite a detailed description, and hopefully one that makes sense given the preceding subsections. We should be able to appreciate now

[18] Heat is nearly always a nuisance. However, some researchers have used CPU's to heat water to power central heating systems using waste energy from computations.

that processors can operate at different speeds, and have varying degrees of power consumption, and they create heat, which is usually not desirable^[18], but by slowing down processors, they can consume less power and create less heat at the cost of reduced performance.

3.3 The performance barrier

We will return to the idea of instructions shortly, and explore further. There are however some higher level concepts that we must understand a little better, not least because they have a major impact upon performance of modern computer systems.

What is performance?: performance is the achievement of certain goals whilst consuming certain resources. Another way to express this is cost versus benefit or **cost-benefit**. We sometimes refer to this as a **performance metric**. Let us consider some cost-benefit examples in very loose terms, making some assumptions:

- ▶ A given CPU can execute one million instructions per second (meaning it has a throughput of one **mips**)
- ▶ The CPU may well require 1 Watt/hour of power to do this.
- ▶ Assume that one watt of electricity costs 1/10th cent,
- ▶ Assume the CPU costs \$100, and is used for 1 yr continuously.
- ▶ The CPU dimensions are 20x20x5mm

Now what can we say about this CPU?, it has a raw performance of 1 mips. This is a measure of how much work it can do in a given amount of time. But how about a cost-benefit analysis:

This CPU performs **1 mips per watt** (because we use 1W of power and get 1 mips in return).

Use of the CPU for one year requires the purchase price (100 dollars), plus the cost of 1 year of electricity at the stated rate of consumption. One year equates to $365 \times 24 = 8760$ hours, and therefore $8760 \times 1\text{W} = 8.76 \text{ kWh}$. At 0.1 cent per watt-hour, this electricity will cost 8760 cents, or \$87.60. Therefore, the total cost of owning and running this CPU for one year is approximately \$188.

Why is this important? The reason we want to know about such things is because we want to be able to predict the **cost-effectiveness** of a given computer system, firstly as a singular case, but secondly in terms of comparison between two or more systems. Consider, for example, the list of performance characteristics given in Table 3.1.

	mips	W/hr	Dimensions	Cost of CPU
CPU A	10	40	20x20x5 mm	\$280
CPU B	5	11	15x15x5 mm	\$150
CPU C	5	15	25x25x6 mm	\$80
CPU D	8	4	25x25x5 mm	\$200

Table 3.1: Various imaginary processors and some of their attributes.

Given this data for multiple hypothetical processors, can we work out from this table, the following information?

- ▶ Which CPU can do the most processing in one hour?
- ▶ Which CPU is most cost efficient for power consumed?
- ▶ Which CPU gives the most performance per dollar?
- ▶ Which CPU is cheapest to buy and run for one year at 1 mips?
- ▶ Which CPU has the worst (highest) power density at peak mips?

Each of these is a performance measure often known as a **performance metric**. Try and answer these questions before reading on, and see how much you can answer correctly.

Most Processing Per Hour

We can say immediately that CPU A is the best choice for this objective. At 10 mips, it performs more work than any other processor in the list in a given period. We could calculate the total instructions executed per hour if we wish, but the same conclusion will be obtained.

Instructions executed in one hour

CPU A: $10 \text{ mips} \times 3600 \text{ seconds} = 36,000 \text{ million instructions.}$

CPU B: $5 \text{ mips} \times 3600 \text{ seconds} = 18,000 \text{ million instructions.}$

CPU C: $5 \text{ mips} \times 3600 \text{ seconds} = 18,000 \text{ million instructions.}$

CPU D: $8 \text{ mips} \times 3600 \text{ seconds} = 28,800 \text{ million instructions.}$

It is important to realise that mips can only measure raw performance. It does not automatically reflect the true performance of processors in general terms, since each process has a different instruction set.

Most Cost efficient for power

We can determine this using our mips-per-watt measure, or watts per mips. We can see that the most power efficient processor is the one with the lowest watts per mip, or the highest mips per watt, CPU D:

Most Power Efficient

given Mips per Watt = $\frac{\text{mips}}{\text{watts}}$ and Watts per Mip = $\frac{\text{watts}}{\text{mips}}$

CPU A: $10/40 = 0.25$ mips per watt, and 4 watts per mips.

CPU B: $5/11 = 0.45$ mips per watt, and 2.2 watts per mips.

CPU C: $5/15 = 0.33$ mips per watt, and 3 watts per mips.

CPU D: $8/4 = 2.0$ mips per watt, and 0.5 watts per mips.

We might wonder why we need two measures. Actually either of these metrics is sufficient to answer the question posed. However, if we want for instance to run a computer system on a battery with only 50 watts available before the battery charge runs out, then we can use watts per mips to quickly determine how much work any one of the processors can perform, or for how long a particular task can be performed (if we know the mips requirement).

Now, if we were to have a battery system that provides a maximum of 50 watts per charge, and we need 1 mips to run a particular task, CPU D can be seen to be able to run for 100 hours on one battery charge. [19]

[19] With a 50 W charge, 1 mips required to sustain the computational task, and 0.5 Watt per mips, therefore expends 50 Watts at 1×0.5 Watts per hour, thus 100 hours of operation is possible.

Most performance per dollar

This is another simple calculation, but having the same caveat as performance per hour. We are only measuring raw performance per dollar. CPU C has the best dollar per mips or mips per dollar:

Most performance per dollar

CPU A: $280/10 = \$28$ dollars per mips.

CPU B: $150/5 = \$30$ dollars per mips.

CPU C: $80/5 = \$16$ dollars per mips.

CPU D: $200/8 = \$25$ dollars per mips.

Least Expensive to buy and run for one year at 1 mips

For this calculation we must combine the cost of power consumed, with the cost of purchasing the processor. The least expensive processor to buy may not be the least expensive overall, conversely the one that consumes the least power per mips may not be the best either, depending upon the lifespan of the intended use. Indeed we see CPU C is the best choice here even though it is one of the poorer performing processors for power efficiency alone.

Most Cost effective to own and run

At 0.1 cents per watt-hour, a kWh^[20] costs 1000×0.1 cents = \$1.

CPU A: 4 watts per mips, therefore $365 \times 24 \times 4$ watt-hours = 35.04kWh, costing \$34.94.

CPU B: 2.2 watts per mips, therefore $365 \times 24 \times 2.2$ watt-hours = 19.21kWh. costing \$19.22.

CPU C: 3 watts per mips, therefore $365 \times 24 \times 3$ watt-hours = 26.21kWh. costing \$26.21.

CPU D: 0.5 watts per mips, therefore $365 \times 24 \times 0.5$ watt-hours = 4.37kWh. costing \$4.37.

And then, total costs (to nearest dollar):

CPU A: \$280 + \$34.94 = \$315

CPU B: \$150 + \$19.22 = \$169

CPU C: \$80 + \$26.21 = \$106

CPU D: \$200 + \$4.37 = \$204

[20] kWh : Kilowatt Hour, the typical measure for electricity cost calculations.

It is important to recognise, as with some earlier metrics, that there are additional factors not considered here. For example, a computer system contains a power supply module, cooling fans, and much more hardware. This calculation only considers the CPU in isolation. The benefit of choosing CPU C may thus be far less significant in overall terms.

Worst and best power density at peak mips For a complete processor chip in its packaging, power density is a function of volume and power consumption^[21]. CPU C has the worst power density:

approximate power density

CPU A: Volume = $20 \times 20 \times 5$ mm = 2000 cubic mm. at 40W this gives 0.02 W/mm^3 or 20 mW/mm^3 .

CPU B: CPU A: Volume = $15 \times 15 \times 5$ mm = 1125 cubic mm. at 11W this gives 9.8 mW/mm^3 .

CPU C: Volume = $25 \times 25 \times 6$ mm = 3750 cubic mm. at 11W this gives 4.0 mW/mm^3 .

CPU D: Volume = $25 \times 25 \times 5$ mm = 3125 cubic mm. at 11W this gives 1.3 mW/mm^3 .

[21] There are other conventions for power density worth highlighting. In chip design, the actual chip die is effectively a 2-dimensional slab of silicon, of the order of 10s to 100s of square millimetres. In this domain power density is typically quoted in watts/cm^2 in relation to die surface area. Silicon engineers may even calculate power density in small regions of the chip where power can be very high (known as hot spots).

[22] Indeed, taking this to the full extent, server cabinets full of 'rack-mount' circuit boards can be rated in power capacities of the order of 20-30kW, and power densities calculated accordingly given the cabinet volume. This is used to indicate how much compute capability can be hosted in a given space. This is a critical concern in server farms and data-hosting centres, for example.

We can conclude that CPU A has the worst power density at peak mips and CPU D has the best. Power density can be important. Where power density is very high, special cooling requirements may arise such as heat-sinks and CPU fans, or even liquid cooling in some extreme cases^[22]. In less demanding situations, devices that are hand held or wearable, and therefore very compact, may also have difficulties even with more moderate power densities.

3.3.0.1 Power/Frequency Scaling

Given that power is such a significant concern, and indeed power consumed ultimately turns into heat creating a further problem, we would expect that CPU designers have spent much effort on trying to deal with these issues.

Unfortunately the ideal solution, of endlessly making transistors more and more efficient, thus consuming less power, is simply beyond the realms of physics as we understand them today. Circuits and chips are becoming more efficient, but nowhere near as rapidly as we would wish. Designers currently turn instead to optimisation strategies at another level in order to make the best of the systems as they are able to be fabricated.

Processors operate at particular clock frequencies, and each clock pulse equates to a cycle in which some work is performed, some power consumed, and some waste heat generated^[23]. A simple solution to power consumption is to reduce clock frequencies when the CPU does not have much work to do, and to 'ramp up' the frequency when work suddenly becomes demanding. This may well be labelled with various conceptual terminology such as *speed-scaling*, *speed-step*, *CPU throttling*, and so-on. It might be considered analogous to turning a tap on more or less to moderate the flow of water.

If we take the frequency scaling idea to its extreme, then a CPU that typically runs at 100's of MHz might well only be clocking at 10's of KHz. In this mode the CPU is effectively in what is known as **sleep mode** and can only do as much as is required to keep an eye on what is happening in the system so it is ready to 'wake up' and ramp up the clock rate when needed. Sleep modes are often a little more sophisticated than this, turning off some parts of the CPU entirely for periods of time, might have multiple options, and can achieve very high power savings. However, as we always tend to discover in computer architecture, things are rarely as simple as turning a dial and getting a linear response. Remember, as

[23] In actuality, not just heat. Processors also radiate radio frequency energy as part of the 'waste' energy output, though it is less often considered a problem.

mentioned earlier, a CPU has two kinds of power consumption, **dynamic** and **static** power.

Dynamic power is the power that is consumed for each operation performed, such that the more operations you may execute per second, the more power is consumed. This is a more or less linear relationship for our purposes. Ten times as fast a clock rate means ten times the (average) dynamic power consumed^[24]. On this basis, setting the clock rate to zero ought to reduce power consumption to zero too.

Static power is power that is consumed continuously even when the CPU does nothing, even when its clock rate is set to zero (which many CPU's do not permit in any case). structures like register files, on-board memories, and other circuits, will consume power even when idle and not even clocked by a clock signal. This static power is like a baseline, upon which dynamic power is then added.

[24] Provided it is ten times the same kind of work. The precise amount of dynamic power consumed depends upon the exact mix of instructions being executed , with an average being an approximation.

CPU complex power Example

Suppose a CPU has the following power data from the manufacturer's data sheet, and a maximum clock rate of 100MHz:

Standby (sleep) mode power: 0.1 Watt.

Static Power, P_S when active : 20 watt.

Dynamic power, P_D when active: 2 Watt per MHz ($P_d=2W$).

Assuming that Power = $P_S + P_D$, where $P_D = P_d \times F$ MHz,

Then, we can see that this processor, when active, will consume up to 220 Watts at maximum clock rate of 100 MHz:

$$\text{Total Power} = 20W + (100 \times 2W) = 220W$$

But we can also see that if we halve the clock rate, the power is then:

$$\text{Total Power} = 20W + (50 \times 2W) = 120W$$

Notably, the total power has not halved, due to the residual static power that is always present. Importantly, total power is not linear with respect to frequency.

Now what about sleep mode? Suppose the CPU is clocked at full speed for 50% of the time and is in sleep mode otherwise:

At 100MHz, P_{tot} is 220W , but for 50% time , thus 110W.

And in standby mode: $0.1W \times 50\% = 0.05W$.

Thus total power is 110.05W, better than simply scaling the clock.

What we have observed is that power can be moderated in different ways using CPU design features. In practice we have not considered all of the factors, there are drawbacks and penalties for moving in and out of standby mode or for changing clock frequencies 'on the fly'. Such factors vary between CPU designs, and these may well make marginal cases less worthwhile to consider. However, we do see that when a CPU has no useful work to do, or very little, it is potentially valuable to use standby modes.

3.3.0.2 More about Mips

A further word about **mips** at this point might be useful. What exactly do we mean by mips? In its simplest possible definition, mips is simply millions of instructions per second, a simplistic measure of the **computational throughput** a processor is capable of. However, in a more realistic situation, we would know what sort of instructions we are executing, how often, and so on.

Every program is different, and every processor has a slightly different instruction set, so a simple mips measure is not really a true representation. We may also come across the term **peak mips**, which measures the maximum possible mips a processor can achieve under the best possible conditions. Often when we talk of billions of instructions per second, it is common to use GOPS and GFLOPS (Giga-operations per second, and Giga floating-point operations per second), and these can also be peak measures in some cases.

To measure anything other than peak mips or peak gflops, and thus perhaps compare several processors with realistic code behaviours, we would need to have a common test program to run to reproduce the same workload on each processor in turn. These test programs are known as **benchmarks**.

A well known example, **Scimark** is a scientific computing benchmark which can be executed on any processor, and then used to measure identical work across many systems. There are different kinds of benchmarks for different kind of computing tasks, such as artificial intelligence, scientific computing, gaming and multimedia, embedded systems, and other domains. The lesson here is that a performance metric should be chosen carefully to test the behaviour that is relevant to the problem.

In a simplistic example, consider a program that multiplies two 3x3 matrices together. Suppose our earlier CPU A requires 1000 instructions to complete this task, and CPU B requires 800 instructions to complete the

	Frequency	CPI	Peak mips
CPU X	3 GHZ	1	3,000 mips
CPU Y	5 GhZ	2	2,500 mips
CPU Z	5 GHz	0.8	6,250 mips

Table 3.2: Hypothetical CPUs with clock rates in GHz, peak mips, and CPI.

same work (but using its own slightly different instruction set). We might think CPU B is best. But remember that CPU A performs at 10 mips (peak) and CPU B performs at 5 mips (peak). Will 1000 instructions at 10 mips complete more quickly than 800 instructions at 5 mips? Almost certainly yes in this case, but without running the two CPU programs (effectively a simple benchmark) we cannot be certain.

3.3.1 Processor frequency

Another key factor, relevant to performance measurement, is processor **clock frequency**, which is a measure of how fast a processor can perform a key internal circuit operation^[25]. We measure frequency in **cycles per second**, or **Hertz**. Since most processors and digital circuits run at incredibly fast speeds, we typically refer to processors running in the megaHertz (MHz) or gigaHertz (GHz) range.

This is one area where we must be particularly cautious: internal circuit operating frequency, or **core clock frequency**, is not directly proportional to mips. An instruction might require one clock cycle, or two, or three, or perhaps many more. The same identical operation on another processor might require more, less, or the same number of clock cycles.

Therefore, a higher clock frequency on its own is not a guarantor of higher mips or higher benchmark scores.^[26] To unravel this potential confusion we need to know another important parameter: **clocks per instruction**, or **CPI**.

Consider the data in Table 3.2. If we consider frequency alone, then CPU Y and CPU Z appear to have identical performance, and CPU X is not as good. But if we also consider the CPI rating of each processor, then we see that CPU Z is the best choice, and CPU Y is actually the worst choice for peak mips.

In order to make sense of this, we again face the problem of exactly which instructions we are referring to when we use CPI: Clocks per instruction?, yes, but clocks related to which instructions? Again, benchmarks help

[25] This is usually the slowest circuit path within the design, known as the critical path.

[26] An analogy is a car engine, where frequency is represented by revolutions per minute (rpm). Here rpm is not an indicator of speed on its own, we also need to know about the gearbox or transmission ratio, and even the diameter of the wheels, to determine how fast a given vehicle can move. So an identical rpm (frequency) could give easily very different results in two different vehicles.

us out here and a credible CPI measure would ideally be based upon a suitable benchmark executed on each CPU we wish to compare, using an appropriate mix of different instructions. This is in fact how many computer systems are rated for performance capabilities, and the benchmarks might be general purpose, scientific workload, multimedia focused, or based upon more specialised requirements.

3.3.2 The power of multiples

It may well appear that to some extent the obvious path for processors is to push for smaller and smaller transistors, more and more complex CPUs, which are faster and faster, leading to higher and higher clock frequencies, and greater and greater processing throughput, be it mips, scimark score, or any other performance metric. However, this trend, which has indeed been sustained for several decades prior to the 2010's, has a downside.

[27] This is known as the Law of diminishing returns. Successive application of equal effort delivers progressively smaller benefits.

As transistors get smaller, their power consumption becomes more and more of a problem, design issues can cause CPI to increase, and the ability of more and more complexity to exploit the most frequently occurring cases in workloads becomes less and less beneficial^[27]. As a result of this, CPU's have shifted through several eras of new design thinking, often referred to as **paradigm shifts**. Whilst we do not need to have a deep understanding of these ideas here, it is very useful to understand the terminology and general principles, as these are ideas that come to the fore regularly in computer systems design and use.

Early processors were able to execute one instruction at a time, and would be classed as having a **scalar execution model**. Here, the execution of instructions was serial, meaning that one instruction would complete before the next began. This seems fairly logical, if programs are sequences. However, it was recognised early on in the history of processor design that an instruction need not always be completed before the next one begins. This concept of **overlapped instruction execution**, a form of a technique known as **pipelining**, allowed more instructions to be executed within fewer clock cycles, meaning that the inherent CPI decreases: usually an indication of better performance capacity.

[28] When one instruction relies on the result of another, we call this a dependency. Only non-dependent instructions are potentially able to execute in parallel.

This kind of optimisation is only possible where the two instructions do not depend upon one another: the next instruction clearly cannot use the result of the previous instruction unless it waits for the preceding instruction to finish^[28]. This potential dependency is known as a **pipeline hazard** or **data hazard**, and with care, and using a concept known

as **data-flow analysis**, code can be written to avoid many of these problems.

Once it was understood that instructions could overlap frequently enough for a benefit to be gained, the next progression was to realise that multiple instructions could start at the same time. This is referred to as **multiple issue**: issuing multiple instructions per clock cycle, and is the basis of the **superscalar execution model**.

In superscalar execution, instructions can be issued in groups, perhaps 2, 3, or 4 instructions at a time^[29] and these instructions finish at different times (**out of order completion**). Of course, this means that instructions are issued in order but may complete out of order. Again, the consequences of instructions relying upon preceding results leads to data hazards of various kinds, but careful program construction or clever compilers will avoid these problems^[30].

Eventually, superscalar processor issue width also hit a limiting point in its evolution. Moving from an issue width of 2 to 4 yielded good benefits but moving from 4 to 8 yielded a diminished return. The concept could not exploit the ever-increasing complexity available for chip designers, because there was not enough non-dependency in the instruction sequences. Therefore, another shift in architectural thinking was required.

The idea of placing multiple processors on one chip was the only obvious next step to use the larger transistor counts being offered. This resulted in the creation of the current generation of **multicore processors**, where perhaps between 4 to 8 large complex processor cores are integrated onto a single chip. As we will see later, this multiplicity of processor cores has relevance to modern operating systems, and introduces some additional advantages.

Hopefully what we can learn from this is that a superscalar processor might operate at a lower clock rate than a scalar one, yet execute more instructions, and a multicore processor could easily outperform a single core even when running much more slowly.

3.3.3 Guesswork: prediction and speculation.

As we have observed earlier, a program is typically non-linear in program flow. In other words, rather than instructions being executed entirely consecutively from memory locations 0,1,2,3,4,5 and so on, we find that a program sequence might be linear for a relatively short run, and then jump to another part of the program where a further linear sequence

[29] This parameter is known as **issue width**. Sometimes processors are referred to as being 2-way or 4-way superscalar, which means they have an issue width of 2 or 4 instructions per clock cycle.

[30] There is a substantial branch of computer science devoted to instruction scheduling, and this is a topic with many avenues to read upon further. See also Tutorial ??.

[31] This is often related to program structure in the high level language: such as loops and if-then-else statements.

of instructions is encountered^[31]. Often these non-linear transitions, or **jumps**, are due to a **conditional evaluation** (in other words, a computation resulting in some form of comparison with a true/false outcome).

For example if we tested if some data value 'X' is greater than zero, there are only two possible outcomes: true or false. A program may decide to jump to a new program section if the comparison is true or just continue linearly to the next instruction if the result is false. In this way, program sequences can become conditional upon the results of other program sequences.

As a result of this **conditional decision point**, the location of the next instruction to be executed is not always known in advance, meaning that we cannot fetch the next instruction whilst the current one is executing. Remember how we observed that early scalar processors progressed by introducing overlapped instruction execution: a simple form of pipelining, to achieve a performance gain? When linear code sequences are executed, overlapped instruction fetching is fine, but when a condition is encountered and a jump occurs, this presents us with a temporary roadblock, and time is wasted waiting for the comparison instruction to complete, costing the system some performance. This is known as a **branch penalty**.

[32] In computer architecture a 'Branch' is just another name for a 'jump'.

3.3.4 Branch prediction

We can deal with the problem of unknown jump conditions using a technique called **branch prediction**^[32]. This can be very simple or rather complex.

In the simplest case the CPU designer could simply assume that certain jumps are always true and others are always false. This fixed branch prediction scheme, known as **static branch prediction**, can be designed into the CPU circuitry or **hardwired**. It would rely upon the fact that, statistically, certain branches behave in certain ways. Of course this is primitive, and is only as good as the assumptions made in the first place.

The static branch prediction strategy may not work well for all programs. An alternative is to use **dynamic branch prediction**. In this scheme, the processor keeps track of how often particular branches are found to be true or false via a **branch history table**. Then, when the CPU encounters a branch, it uses the most up-to-date information in that table to decide if the branch should be taken or not. Call it an educated guess, based on the way the program is behaving at that point in time.

Clearly this scheme has some advantages: it would be expected to adapt to the behaviour of individual programs, and even particular parts of programs, and can therefore be more accurate than less sophisticated schemes^[33].

So what does branch prediction allow us to do? If we guess that a branch is taken or not taken, then we can begin fetching the next instruction immediately, with overlapped instruction fetching preserved, and performance kept at peak level. This may typically save one or two clock cycles. If the prediction is correct then we carry on and everything is wonderful! On the other hand, if the prediction is wrong, then that fetched instruction has to be discarded and the CPU then switches to the correct location (which by now it will know). This is known as a **branch mis-prediction penalty**, and this may be of the order of a few clock cycles typically.

What we are encountering here is a **performance tradeoff**. If a correct branch prediction saves one clock cycle and an incorrect prediction suffers a 2 clock cycle penalty, then how do we know we are getting a good deal? We can evaluate this if we know the branch prediction **hit rate**, which is the success rate of the prediction algorithm:

Hit rate impact example

Suppose for example we have the following information:

- ▶ The CPU predicts the branch correctly 80% of the time,
- ▶ For this particular CPU this saves 2 clock cycles,
- ▶ Meanwhile a mis-prediction also costs 2 clock cycles.

What is the net benefit?

Gains: $80\% \times 2 \text{ clock} = + 1.6$

Costs: $20\% \times 2 \text{ clocks} = - 0.4$

Combined: $1.6 - 0.4 = 1.2$

Overall = +1.2 clocks (a saving of 1.2 clock cycles per branch).

So it is apparent that, in this example, we gain 1.2 clock cycles per jump/branch if the hit rate is 80%.

Now, can you determine what are the net gains or losses if the hit rate is 40%, 60% and 90%? Try and calculate these based on the example preceding*.

[33] There are actually a whole variety of dynamic branch prediction algorithms, some better than others.

* Answers: 40% (-0.4), 60% (0.4), 90% (1.6).

We may well wonder at this point: is a saving of 1.2 clock cycles really such a big deal? It doesn't sound like much when we talk about processors running at 100's of millions of instructions per second. However, it turns out that branches are very common (as many as one in 6 instructions can be branches). So for a processor executing 1 million instructions per second, we might save:

$$1.2 \times 1,000,000 / 6 = 200,000 \text{ clock cycles per sec}$$

This is a lot of computation time and resource saved. You may wonder why branches are so frequent. This is because they are used heavily in program structures such as loops, which are a fundamental workhorse of programming practice.

3.3.5 Speculative execution

Speculative execution is an extension of branch prediction, particularly useful in superscalar processors. Here, rather than simply predicting the branch so we can fetch the next instruction, the CPU allows execution to continue on some way beyond the predicted branch, even though we do not yet know if that is the correct course of action at that point.

This means that any instructions executed after that point are **speculative**, and they must be capable of being discarded and their effects reversed if the branch turns out to be mis-predicted. It follows that the gains here are much larger than simply predicting a branch, and the penalties for **mis-speculation** are larger too. As this topic requires quite a lot more knowledge of the internal architecture of the CPU in question, we will not delve any deeper in this section.

3.4 CPU internal architecture

So far we have examined CPU's more or less as self-contained modules, and we have understood some of the capabilities and limitations of processors in general use. Although a full understanding of the internals of modern processors is a major study effort in its own right, we will here try to establish some basic ideas to build upon at a later date.

We often use the term **microarchitecture** to refer to the internal organisation of a processor. This refers to the internal components of the processor, as far as can be described at a level that is relevant to the programmer, in order to understand how the processor operates and how

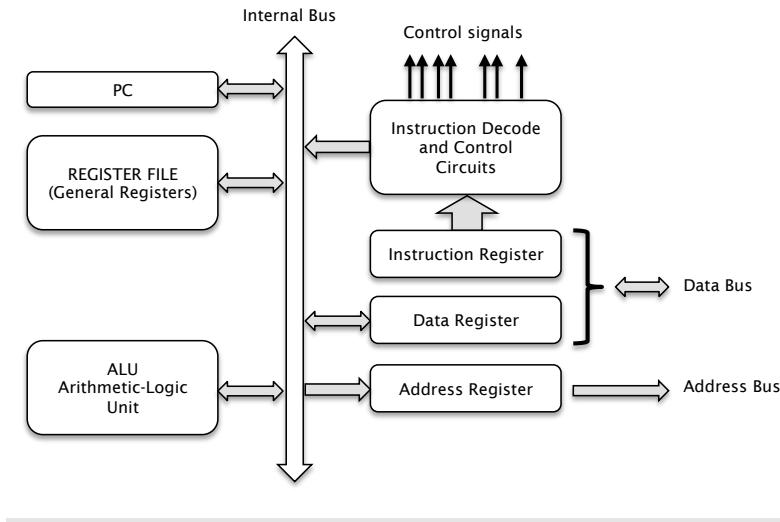


Figure 3.3: Simplified microarchitecture of a CPU. Showing internal components of a processor, including program counter, instruction, data and address registers, arithmetic logic unit, register file, and instruction decoder.

it can be programmed at the machine level. Figure 3.3 shows a typical, albeit simplified, microarchitecture.

Now let us attempt to understand how this microarchitecture operates: We already know that a CPU fetches instructions from memory, and these instructions tell the CPU to perform an arithmetic or logical operation on data held in registers. The **arithmetic logic unit (ALU)** does the actual computation.

Registers are simply binary storage elements that can hold numbers of a certain size. We often talk about a register being 8-bit, 16-bit, 32-bit, and so on. What we mean is that a single register can contain a binary number with that quantity of bits.

We can see in Figure 3.3 that there are a number of different registers, and these have different functions:

The **PC register** is also known as the **program counter**, and keeps track of where the next instruction is in memory, so we can fetch it when needed. As we noted earlier, if a branch occurs, the value of the PC may jump to a completely different location in memory. Otherwise it simply increments to the next instruction location in a linear fashion.

The programmer also has access to general registers, which are usually organised into a block known as the **register file**. How many registers are available will vary widely depending upon the processor in question. Some have as few as two, some have hundreds of registers, though 8 or 16 registers is much more typical.

The **data-register**, **address-register**, and **instruction-register** are all needed by the CPU to operate, but are not normally visible to the programmer. Think of them as operating 'behind the scenes', to make sure information gets to the right places at the right times.

When the CPU needs to access a memory location, it will place an **address** in the address register, and then perform a **read** or **write** operation.

If it performs a memory read operation, the resulting value fetched from memory will be placed in the data register, or the instruction register, depending upon the kind of read: an **instruction fetch** reads instructions, a **data fetch** reads data values.

The instruction register holds the current instruction, also known as an **opcode**, whilst the control unit examines it and **decodes** it.

If a memory write operation occurs, then whichever value, previously having been placed in the data register, is transferred to memory. So we can see that registers allow data or instructions to be transferred to and from memory, and to dictate which locations should be accessed.

In order to coordinate these activities, and decide what should happen and when, the CPU has a **control unit**, a digital circuit that is capable of generating sequences of control signals to tell the internal components of the CPU what to do, and when. It operates as follows:

- ▶ An instruction is read from memory into the instruction register.
- ▶ The instruction is decoded (converted into control signals)
- ▶ The control unit provides the correct sequence of control signals during one or more clock cycles.

Exactly what control signals are generated will depend upon the instruction being decoded. There can be hundreds of control signals in a modern processor, so instruction decoding can be quite a major design requirement.

One of the things that the control unit is responsible for is to tell the ALU (arithmetic-logic Unit) what kind of operation to perform on data. The ALU operates as follows:

- ▶ The **Control Unit** sends one or two register contents to the **ALU** by performing register read operations from the source registers.
- ▶ The **ALU** performs the chosen operation
- ▶ The result is sent back to a destination register by a **register write** operation, sometimes referred to as a **register write-back** operation.

ADD R0, R1, R5 (what the programmer sees) What the CPU does:	CPU Activity
<ul style="list-style-type: none"> • Copy the PC register to the Address register • Perform a memory Read to the instruction register • Decode the instruction in the instruction register • Copy Register 0 to the ALU • Copy Register 1 to the ALU • Tell the ALU to ADD • Write Back the ALU result to register 5 	<ul style="list-style-type: none"> • Instruction Fetch • Instruction Decode • Register Read • Register Read • Execute • Register Write

Figure 3.4: A typical micro-instruction sequence. Shows the key stages of an instruction life-cycle: instruction fetch, instruction decode, register reads, execute phase, and register write-back.

In a straightforward microarchitecture, any register can be read or written by the control unit. The control unit can also generate **constant data values** and send these to the ALU or registers. What is also quite clear is that even the simplest instruction visible to the programmer causes a potentially complex sequence of operations to be performed within the machine. As a result of this, it is quite possible that a single instruction might take several clock cycles to be completed, hence a CPI of 2, 3 or more might be observed.

3.4.1 An instruction 'in flight'

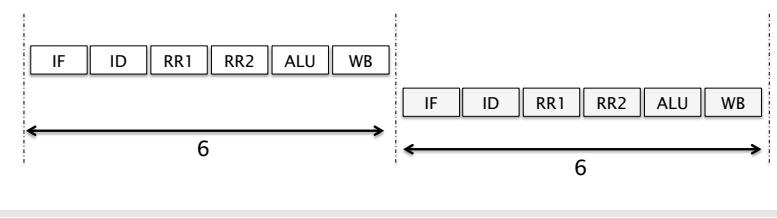
We sometimes refer to instructions as being **in flight**. What we mean is that it is in progress and being processed through its various stages of life. Let us take a very simple example. We wish to add two registers together, and this sequence of events is represented by Figure 3.4.

So we can see that any single instruction from the programmer's point of view, is actually a sequence of **micro-instructions** as far as the CPU is concerned internally. So programs are made of instructions, and instructions are made of micro-instructions.

We can also see that we can perform more complex operations. For example, we could add two register contents together, send the result to the address register, perform a memory read, and then write the result to a third register. This would be performing something rather complex, but also very useful. This particular scenario is used to access elements of arrays or data tables efficiently. We could devise micro-instruction sequences for quite a few cases in this way, which is exactly what the CPU designer has to do, and in the case of CISC processors is taken to the full range of possibilities. The designer then needs to design a control

Figure 3.5: Non-Pipelined Instruction Microsequence.
 Showing IF: instruction fetch, RR1: register read 1, RR2: register read 2, ALU: ALU operation, WB: register write-back.

[34] For those familiar with digital logic design, or wanting to know more, a control unit is often just a variation of something known as a state machine. There are several techniques for implementing them, including Mealy and Moore models, as well as ROM-based designs.



unit circuit that generates the right internal signals at the right times to make those **microsequences** happen^[34].

So how many clock cycles does our ADD instruction take? According to our micro-instruction sequence, there are six steps, and this might be envisaged to require at least six clock cycles, implying a CPI of 6.0, which is quite slow. We can see this visualised as a time-line in Figure 3.5, which shows the same instruction repeating.

However, we might observe, from our micro-instruction sequence, that the first two steps are related to what we have referred to previously as instruction fetch and decode. As we noted earlier, it is possible with careful control unit design to overlap instruction fetch, decode, and instruction execution activities. Actually, this also applies to register read and write-backs. It is also possible to read two registers and also write a third register all at the same time, using a circuit known as a multi-port register file.

This being the case, the ADD instruction could appear to take as little as one clock cycle, and CPI=1.0, because almost all activities for an instruction can overlap with a preceding instruction. As Figure 3.6 shows, a new instruction can complete after every clock cycle using the overlapping model. We can see in this diagram that this is possible because different things are happening in different time slots, and they 'zip' together like a zipper quite neatly in this example.

One thing we can observe in Figure 3.6 however, is that by the time the third instruction is overlapping with the first two, we reach a point where its two register reads overlap with the register write-back of instruction one. This is now a potential data-hazard. There are two possibilities here: either delay the register reads to avoid any preceding write-backs, and lose some performance, or make sure that none of the registers being read are the same as the one being written back. This is the basis of an optimisation technique called instruction scheduling.

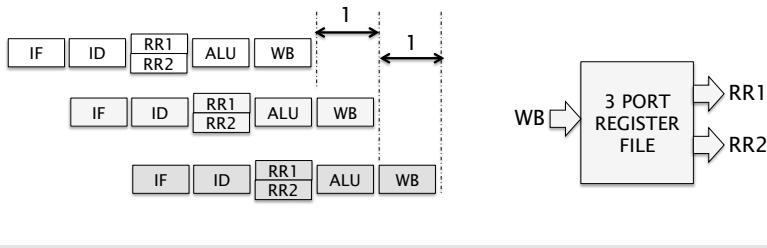


Figure 3.6: Pipelined Instruction Microsequence, showing IF: instruction fetch, RR1: register read 1, RR2: register read 2, ALU: ALU operation, WB: register writeback. To the far right: a 3-port register file, allowing two reads and one writeback simultaneously.

3.4.2 A more advanced microarchitecture

Now that we have established some basic understanding of microarchitecture, we should in theory be able to look at one of the many processor architecture diagrams available from manufacturers that describe the internals of the machine, and understand at least the basic aspects of what is going on.

Now let us look at a slightly more advanced processor model, such as the one in Figure 3.7.

How does this differ from the previous case?

Microarchitecture Differences

- ▶ There are now **two ALUs**, and these have been re-designated as integer ALUs.
- ▶ A **third Arithmetic unit** has been added, but this one is **floating point**, so it can work with decimal values.
- ▶ We have added a **branch prediction unit**, which can work out what the next value of the PC will be, based upon a **branch prediction algorithm**.
- ▶ Finally, because there are more ALU's and other units, **the Register File has more connections**.

A significant difference here is that this new microarchitecture can conceivably execute up to four instructions at a time. Of course they cannot be just *any* arbitrary combination of instructions. This processor can execute two integer ALU operations, alongside one floating point operation, and also process a branch instruction by predicting its behaviour.

This can only happen when the combination of instructions is just right. We refer to this as **peak execution performance** or **peak mips**. In reality, due to the way programs are structured, it is more likely that we

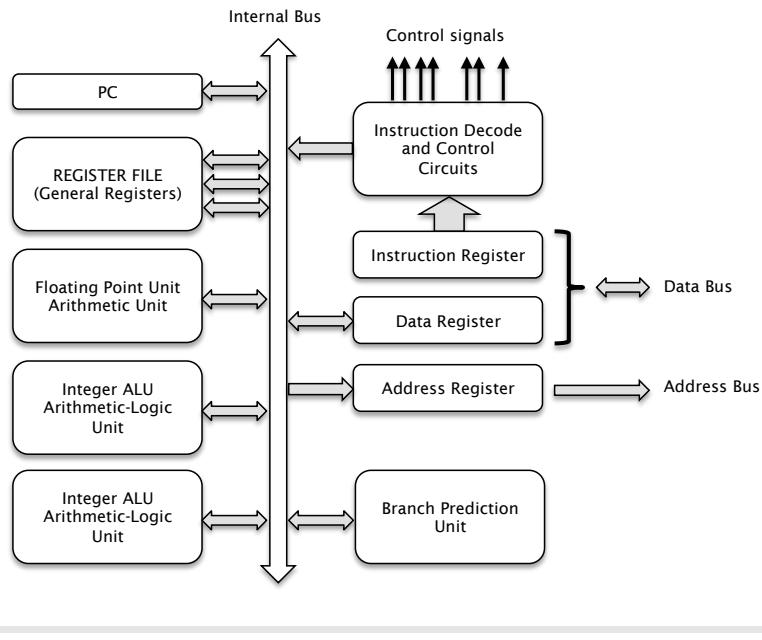


Figure 3.7: A Superscalar Microarchitecture. Showing internal components, including multiple integer ALU's, a floating-point ALU, branch prediction unit, instruction decode, register file, program counter, and additional registers.

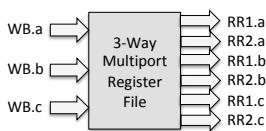


Figure 3.8: 3-way multiport. Given, for example, a three-way superscalar processor microarchitecture, we must then have a three-way multiport register file for peak performance. Such a component would require six read ports and three write-back ports, all capable of operating simultaneously. This implies some quite complex and power-hungry circuitry, and register files are well known as a challenging aspect of modern CPU chip design.

will achieve perhaps 3 instructions per clock cycle on average, but this is still a desirable CPI of 0.33 nonetheless. This is an important point to remember: in computer architecture and systems, four times the potential does not always deliver four times the actual benefit. Additionally, this extra complexity is costly in terms of hardware, and may result in lower clock frequencies, requiring more complex components such as that shown in Figure 3.8. There is no such thing as a 'free lunch' in computer architecture.

3.5 A little bit of programming

We should now have a good understanding of the general internal behaviour of a relatively simple processor. At this point it should be possible to start to write simple programs, assuming of course that we have a list of instructions, an **instruction set**, that we can understand and use. Since our processor examples were just imagined for the purpose of learning the basic principles, we have the advantage of being able to imagine an instruction set as well.

Of course for these instructions to be valid, they must be capable of being performed using the microarchitecture. We will not worry about this right now, but a good exercise would be to look at each instruction

Table 3.3: Instruction reference sheet.

INSTRUCTION SET REFERENCE SHEET	
Instruction form	Function and description
ADD Rx, Ry, Rz	Add Register x and register y together and store the result in register z.
SUB Rx,Ry, Rz	Subtract register x from register y, and put the result in register z.
MUL Rx,Ry, Rz	Multiply register x by register y, and put the result in register z.
DIV Rx,Ry, Rz	Divide register x by register y, and put the result in register z.
LDI Rx, nn	Put the number nn in register x
LOAD [Rx], Ry	Copy contents of memory location pointed to by Rx into register Ry

definition and test it against the microarchitecture to check that they are actually valid.

Let us suppose that the instructions are available as given in our 'instruction reference sheet' given in Table 3.3 and the aim is to write a program to get three numbers from memory and calculate their average, as shown in Table 3.4. In order to make the program behaviour more understandable, we sometimes use a technique called a **program trace**, as shown in Figure 3.5. This shows the step-by-step effects on the processor of each instruction as it is executed, including the program counter and any registers modified. We can see that if we assume the initial memory contents are 6, 8 and 3, then the average of these numbers is 5 (excluding any decimal remainder, since the arithmetic instructions are integer operations).

What we can see here is that the process of writing programs in the lowest level machine code (also known as machine language), that which the CPU understands, is somewhat tedious and long-winded. This is why the vast majority of programs are written in high-level languages such as Java, Python, and C. By using high-level languages, the programmer can focus on what they want to do at a high level of abstraction and not worry so much about what the CPU does at the machine level. A software tool called a compiler or an interpreter then does the rest.

We will not delve any deeper into programming at the machine-level at this point. This is another of those topics that can fill a whole textbook

Table 3.4: Example program.

PROGRAM		(data values in hex)
LDI	R7, 1000	Load the integer value 1000 in R7
LOAD	[R7], R0	Load a value from the address held by R7 into R0
LDI	R7, 1001	Load the integer value 1001 into R7
LOAD	[R7], R1	Load a value from the address held by R7 into R1
ADD	R0, R1, R0	ADD R0 and R1 together, put result in Register zero
LDI	R7, 1002	Load R7 with Integer 1002
LOAD	[R7], R1	Load a value from the address held by R7 into R1
ADD	R0, R1, R0	Add R1 onto the R0 and store back n R0
LDI	R2, 0003	Load the integer value 3 into R2
DIV	R0, R2, R0	Divide value in R0 by value in R2, put the result in R0

Table 3.5: Program Trace.
 This shows the state of the processes step by step, including the program counter, the opcode and data values it fetches from memory, the equivalent instruction, and the register content written back at each stage. We have assumed we know the contents of memory addresses 'M1000' 'M1001' and 'M1002'. Note how the addresses are represented by two bytes in this table. Results 0Eh and 11h are equivalent to 14_{10} and 17_{10} (decimal), respectively. So 17 divided by 3 gives 5 (as the result is an integer with no rounding capability).

Assumptions: M1000 = 6, M1001=8, M1002=3

PC	OPCODE	DATA	DATA	PROGRAM CODE	REGISTERS
0000	LDI			LDI R7, 1000	R7=1000
0003	LOAD	10	00	[R7], R0	R0=6
0004	LDI			LDI R7, 1001	R7=1001
0007	LOAD	10	01	[R7], R1	R1=8
0008	ADD			ADD R0, R1, R0	R0=14
0009	LDI			LDI R7, 1002	R7=1002
000C	LOAD	10	02	[R7], R1	R1=3
000D	ADD			ADD R0, R1, R0	R0=17
000E	LDI	00	03	R2, 3	R2=3
0011	DIV			R0, R2, R0	R0=5

in its own right, and there are plenty of resources available to study this subject for the more inquisitive reader.

3.6 Extending the Instruction Set

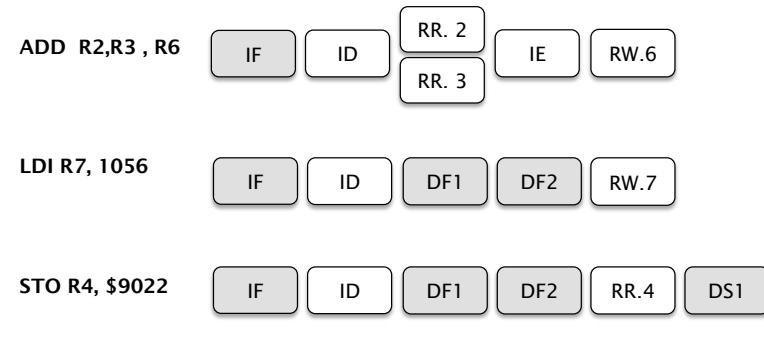
In the preceding sections we have considered some very fundamental and essential machine-code instructions. Of course there are many more possibilities and we could add numerous instructions to those already defined.

A major limitation of the existing instruction set, as presented in Table 3.3, is that it allows only unconditional program execution. In other words, we can specify any number of program steps, but they are executed strictly one after another. Another way of describing this is a **linear execution path**. In order for programs to achieve complex behaviours, there needs to be the ability to execute instructions conditionally. We consider this the essence of 'decision making' within program algorithms, and elements such as loops and 'if-then-else' programming structures would be impossible otherwise. To achieve **conditional execution** we would need to be able to test the state of registers, and then jump to another part of the program depending upon the result. The use of **CMP** (compare) and **JMP** (Jump) instructions is therefore an almost universal expectation for an instruction set. We explore this a little further in Tutorial ??.

Another reason to include additional instructions is improved code efficiency. Consider for example: if we had a new instruction **INC Rx**, which simply increments the specified register by the value of one each time it is executed. Could we then rewrite our program to use it? Do you think it would be more efficient? Would it use less instructions ?

Importantly, although there are many such cases, where instruction set additions can improve the optimisation and/or utility of the code, the addition of extra instructions is not cost free. Remember that instructions require circuits, and more instructions potentially require more circuitry. The result may be a processor with more instructions, but where each instruction is slower, rather than a lean processor that is fast but has limited instruction variety. This is the root of the well known RISC versus CISC computer architecture trade-off (see Section 2.5.1). Consequently, the balance between important and desirable is a fine judgement in CPU design.

Figure 3.9: Variations of instruction microsequences.
 Three examples are given here, with memory events shaded. The topmost case is an **add** instruction, which we have already encountered, and which operates on registers only. The middle case presents a memory **load** instruction, which needs to fetch data values from memory (the **data fetch** or **DF** stages). And, finally, the bottom case shows a **store** instruction, which both fetches data (**DF**) and also transfers data to memory using **data store** (**DS**) stages. Here we assume **DF** and **DS** events are single byte operations.



3.7 More on instruction microsequences

So far we have seen some simpler cases of microsequences, where instructions are fetched from memory, and data is manipulated within the register file. However, this is not the only possibility. Indeed, there are many possible variations of instruction microsequences. A few extra examples are shown in Figure 3.9.

It can be observed in Figure 3.9 that three instruction microsequences are presented. An **add** instruction (using only registers), a **load** instruction (which fetches data from memory), and a **store** instruction (which sends data to a memory address).^[35] We can see that all three instructions require an instruction fetch **IF** stage, but two of the instructions also use memory for data fetch operations (**DF**), and one also uses memory for a data store (**DS**). This means that memory is needed for both instructions and data values, but ideally at different times. Note that these examples assume that each data fetch or store cycle is a single byte read or write.

[35] Here we can also see that in order to store data at the given address, that address must first be fetched from memory first via **DF** stages. So, in this case, we have a microsequence that uses three different kinds of memory access within the same one instruction.

It could also be noted that, until instruction decode **ID** has been completed, we cannot start to perform any **DF** or **DS** operations. It is only *after* decoding the instruction that the microarchitecture knows what is required. Because of the need for both instruction and data accesses to use memory, you may be able to see that pipelining of these more complex instructions is also a more complex task in itself. Indeed, we

cannot have an instruction fetch and a data access going to the memory bus at the same time, a principle we know well as the **von Neumann bottleneck**. You will be able to see this if you attempt to sketch out a few of these instructions in random orders, and try and pipeline them as we did with earlier examples.

3.8 Summary

In this chapter we have explored some key concepts relating to processors, primarily the considerations relating to the concept of a CPU. As we shall see later, a processor, as we commonly know it, may well combine additional components onto the same chip to create a more highly integrated circuit system.

We also discovered that a CPU is responsible for fetching program instructions and executing them, either one at a time or sometimes in small groups, and that this can be exploited to gain higher performance. We also found that there are many ways to measure a CPU's behaviour and performance. Circuit complexity, driven by Moore's Law and the ever increasing miniaturisation of transistors, has pushed us toward higher clock rates, higher power density and higher throughput per CPU, but at a significant penalty: heat and power consumption. When it comes to computer systems, and processors in particular, our options for achieving the best outcome with the least resource are always a key concern.

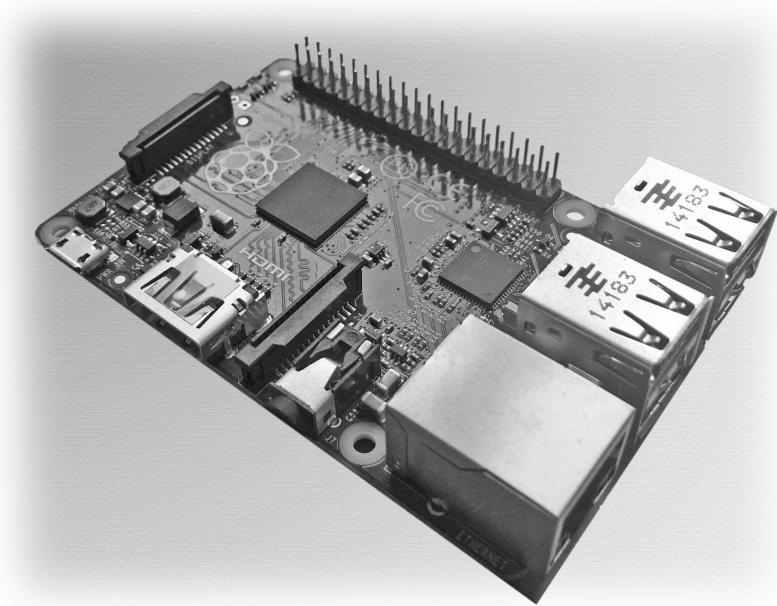
3.9 Terminology introduced in this chapter

Address register	Fetch-execute cycle
Mips	ALU
Floating point	Mis-speculation
Arithmetic Logic Unit	Heatsink
Moores Law	Benchmark
Hit-rate	Multicore
Branch	In-order issue
Non linear	Branch misprediction penalty
Instruction	Opcode
Branch prediction	Instruction issue
Out-of-order completion	Clocks per instruction
Instruction register	Overlapped instruction execution
Cost-benefit	Instruction set
CPI	Integer
Data hazard	Integrated circuit
Data register	Issue width
Data-flow analysis	Jump
Decision point	Kilowatt
Destination register	Linear program sequence
Dynamic branch prediction	Load
Execute	Logic chip
Feature size	Logical operation
Machine code	

These terms are defined in the glossary, Appendix A.1.

4

Computers Everywhere



A Raspberry Pi Single Board computer, approximately 3 x 2 inches in size. Photo: Christopher Crispin-Bailey 2019.

4.1	A machine with many facets	52
4.2	The general purpose computer	52
4.3	General purpose computers	53
	The computer in your hand.	54
	The hidden computer	55
4.4	Being specific	56
4.5	The impact on processor technology.	58
4.6	Mainframes and supercomputers	58
4.7	The internet of things	60
4.8	Portable, wearable, implantable	61
4.9	Parallel processing	62
	Speedup vs Amdahl's Law	63
4.10	Summary	65
4.11	Terminology	66

4.1 A machine with many facets

When we talk about computer systems in general terms, we overlook an important aspect of their design. There is no one single kind of computer system that perfectly suits all possible uses. Indeed, we may not even be aware that computers systems are being used in some situations at all. For instance, if you were asked where to find a computer system in your kitchen, would you immediately think of a microwave oven? The answer is probably no, and yet many everyday items contain computers of some kind within them. But, clearly, these are not the same computers as our laptops or our company data servers.

This underlines an important point: there are many computer systems, with many applications, and using the right kind of system in the right kind of scenario is eminently sensible. Not only that, but it most likely represents the most cost-effective way to solve that particular problem, and cost is always an issue. In this chapter we will try to define some terms of reference and attempt to understand some of the rich variety of computers, and indeed processors, that make up this spectrum of possibilities. To help distinguish between these differences, we often group computers into **general purpose** and **application-specific** domains.

4.2 The general purpose computer

When the first computers were built, back in the days of radio valves, and racks full of spaghetti wiring, computers were built with quite particular purposes in mind. Some of the first digital electronic computers were developed to speed up the calculation of physics calculations, relating to such things as nuclear bomb design, aircraft aerodynamics, cryptography, and only later, as systems became more commercially practical, they were used for processing stock data, financial accounts, and the beginnings of the wider applications we might be familiar with today.

As computers became more sophisticated and powerful, their uses expanded to many more tasks. At this point, many computers began to be designed with the general market in mind: one machine that can do many things for many customers was more cost effective to mass-produce and sell.

But, at the same time, these general-purpose computer systems became less efficient at very specific tasks. There is a saying 'Jack of all trades, and master of none'. This is exactly what general purpose computers are,

and therefore another class of specialist computing systems also emerged at this point.

Let us not be too demanding however. Modern computer systems are now so powerful that many specialised tasks can be performed quite adequately on a general purpose computing system, or even a desktop computer. To some degree this is dependant upon the expectations of the user.

It may, for instance, be possible to run a nuclear reaction simulation in a few minutes, and the programmer or program user may not care that it could be done in 10 seconds on a more specialised machine. Nonetheless, there are times when pure computational athleticism in one discipline is the key requirement, and if the choice is between waiting ten months or ten days for a program to complete, then there is an obvious issue with system capability.

At the simplest level of abstraction, computer systems can therefore be divided loosely into two groups. The **general-purpose** machines, as we have just briefly described, and those **application-specific** systems that are engineered to the last detail to be highly efficient at the one, or few, tasks they are designed for.

We shall examine some of the parameters and attributes of these systems in a little more detail in the next sections. There are further divisions of course, and we will highlight these along the way.

4.3 General purpose computers

It is safe to say that almost everyone has encountered a general purpose computer at some point. The classical examples of these systems include the desktop computer and the laptop. These machines typically have the characteristics shown in the panel following.

We may wonder what 'a large amount of memory' actually is. Well, this is not so easy to answer, as memory capacities have been steadily increasing for many decades, so this is not a 'stand-still' number. Currently (2021) a computer system with 8 Gigabytes of memory is considered fairly ordinary. Some laptop and desktop users may have as much as 32 Gigabytes of main memory, where they have particular needs for certain kinds of software (in fact they are customising their general purpose computers to be more application specific).

Features of General Purpose Computers

- ▶ A large amount of memory (large in terms of the everyday user), that allows a wide range of tasks to be performed,
- ▶ A large amount of local disk storage (again, large in everyday terms),
- ▶ Built from readily available, off-the-shelf, mass-market components,
- ▶ Can run a variety of operating systems,
- ▶ Can run a wide variety of software,
- ▶ Has lots of different kinds of connections for peripheral devices.
- ▶ Is relatively inexpensive.

As far as disk storage is concerned, there is a similar story. Laptops and desktop computer systems with 1 Terabyte of disk storage are becoming increasingly the norm, and certainly 500 Gigabytes would be considered very standard at this moment in time.

A key point here is 'relatively inexpensive'. These computer systems could of course have much bigger memories and much larger disk capacities, but at a significant cost, which is not justified for general purpose computing.

4.3.1 The computer in your hand.

These days, it is not just desktops and laptops that are general purpose computers. In recent years, **smartphones** and **tablet** computers have become widely popular and powerful enough to be used in similar ways: these are also general purpose computers. The interface is somewhat specialised (a **touchscreen** rather than keypad and mouse), and the size, weight and power source place some constraints on the design, but in overall terms these are very much general purpose devices, if perhaps perhaps optimised toward media-rich applications.

Ask yourself how often you make an actual voice-call on your smartphone? Does this amount to as much as 5% of the time you use it? One might suspect possibly even less. Meanwhile, you may well use the same device for games, diaries, home banking, news, social media, web-surfing, navigation, shopping, and much more. These are certainly general purpose devices in the modern sense.

The only difference here is that the kind of circuitry required for these devices must be very compact and very power efficient. As a result of

this, the emergence of mobile processors has been observed in the chip industry in recent years. These processors are typified by integrating many functions onto a single chip^[37] alongside the usual CPU hardware (functions that in a desktop may be provided by physically separate chips or circuit boards). Meanwhile, the power consumption of these chips is optimised for the kind of uses that a hand-held device most needs.^[38]

4.3.2 The hidden computer

Another type of computer you may well encounter frequently in everyday life is a computer which we refer to as an **embedded system**. These may be the most numerous computer systems in the world, possibly even exceeding mobile processors in smartphones and devices, though often using similar processor families.

Embedded systems are effectively computer systems embedded in devices in such a way that they perform a set of functions, without a person having full access to the computer system as a computer user. The example given earlier was the microwave oven. This common kitchen utility frequently has a digital display, a keypad, and a relatively simple processor chip inside the control panel, to operate all of the internal parts of the system that make the oven work. It follows algorithms in order to perform cooking tasks according to the settings provided via the keypad, and it displays information on the display. The processor will also have some memory (generally a small memory built into the same chip). So, referring back to Chapter 2, your microwave oven may well include a von Neumann architecture (CPU, Memory, Input, and Output).

However, it is not just microwaves. Consider the following list, which is certainly only the tip of the iceberg:

Typical places to find embedded systems:

- In a microwave oven, washing machine, dishwasher,
- On the digital lock on your office door,
- Inside an office printer,
- At the self-service supermarket checkout,
- Inside your car engine control unit, and key-fob
- In a central heating control dial,
- In an electric toothbrush,
- Inside your flat-panel tv.

[37] Often referred to as a system on a chip or SOC.

[38] Some examples of these include **A11 Bionic** (APPLE), **Snapdragon 845** (QUALCOMM), **Exynos 9810** (SAMSUNG) and **Hisilicon kirin 970** (HUAWEI).



Figure 4.1: A few of the many examples of devices typically containing some form of CPU, some less obvious than others.

That is quite a list! Yet our list barely scratches the surface. It was once claimed that the average home would probably never require a computer, and indeed it would be hard to envisage any family having more than one. Yet we each seem to have tens of computers in our homes, and we don't even know it. Naturally there is a downside to all of this: all of these devices consume power, create radio emissions, and many are potentially hackable under the right conditions.

Apart from these 'every-day' cases, there are also embedded systems that are highly engineered for very demanding applications, some extremely small in size as in Figure 4.2. These systems have to be designed and programmed in particular ways to ensure that they operate reliably under all relevant conditions. This includes being tolerant of fault conditions (**fault tolerance**), having the ability to revert quickly to a correct state after detecting a fault, being able to respond within very tight timescales to specific events (known as **real-time computing**), and being resilient to security threats such as hacking and viruses.

Examples of this include aircraft control systems, missile guidance, self-driving cars and driver assist capabilities, medical devices such as insulin pumps, medical measurement devices, Dental X-ray machine, and many other 'safety-critical' cases (where any incorrect behaviour simply cannot be tolerated due to the consequences).

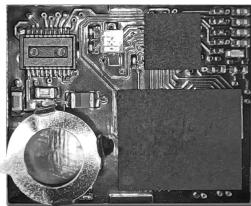


Figure 4.2: The NAT (Neural Activity Tracker). This is a miniature EEG sensor measuring only 22x18x8mm. (Cybula Ltd. and University of York).

4.4 Being specific

When a computer system has enough specialisation to cease to be considered a general purpose system it becomes application specific. As one might expect, there are many examples of this. Many embedded systems are to some extent application specific. They may be designed to fulfil a particular task well at low cost (microwave oven controller), they may be designed to fulfil some very demanding requirements: self-driving cars and safety-critical scenarios for example. Other systems may simply be designed to perform a particular kind of computational task very efficiently. Examples of computationally specialised systems include systems designed to solve particular scientific problems.

Let us consider some examples:

- ▶ **Weather forecasting:** requires a huge number of complex mathematical computations to be performed in a very tight timescale. There is no point in forecasting tomorrow's weather if it takes three days to get an answer. This kind of system would likely have a large

number of processors to handle the volume of computations, and a large memory distributed amongst those processors.

- **Web-search Engines:** Designed to handle high volumes of small simple requests to deliver blocks of data (web page content), these systems have lots of hard disk space, high performance connectivity to networks, and so on.
- **Artificial Intelligence (AI):** A relatively new computing demand, artificial neural networks (ANNs) have been around for a long time as a computing concept. However, processors designed specifically to process ANN tasks are a recently maturing field. A few companies have developed dedicated Neural Network processor architectures, or 'AI engines', which require very specialised systems to be built around them. An example is the GOOGLE TPU designed for large-scale artificial neural network systems^[39].
- **Digital Signal Processors:** This is a class of processor that is very efficient at performing specific mathematical operations on signals. They can be used in sensor systems, radar, biomedical imaging, audio processing (noise cancelling headphones for instance), and other applications. Often these processors have specialised instruction sets^[40]. DSP's are also optimised to permit pipelining of complex instructions to maximise algorithmic throughput.
- **Space Flight:** Computer systems designed for space hardware such as satellites, space probes, and manned spacecraft are a very specialised form of computer system. They have to be very robust, in particular having resilience to radiation and cosmic rays that are encountered in space^[41], and can easily crash a computer system or corrupt its data memory. They also often need to adhere to very precise safety-critical design requirements. There is no room for error in space.
- **Nuclear Applications:** Many issues experienced in space-flight, satellites, etc, are also relevant to the nuclear industry. Building robots that can operate in the harshest nuclear environments, such as the Chernobyl nuclear site, is extremely demanding on processor and electronics reliability.
- **Graphics, media, and gaming:** Many computer systems can be adapted to be high performance gaming machines. However, in order to really perform at the top of their game (no pun intended), they need some rather specialised system components. They often use special kinds of hard disks (solid state drives^[42]) that are currently quite expensive, but very fast. They also use specialised graphics modules, known as graphics cards, which incorporate specialised processors known as GPUs (Graphical Processing Units). In

[39] GOOGLE TPU: Tensor Processing Unit, a specialised AI processing engine.

[40] An example is the Multiply Accumulate operation, which performs the equivalent of $A = A + (B \times C)$. This is a frequent operation in signal processing. A well known example is $(n \times k) + (m \times j)$, which is used to mix two signals together. This can be achieved by two MAC operations.

[41] These are sometimes referred to as 'rad-hard' systems.

[42] Solid state drives, or SSD's are becoming more common as a general system component. However, typical capacity versus cost is still nowhere near that of traditional disk units. This will change in time.

effect, the graphics task is handled by a very specialised computer system all of its own (the graphics card).

4.5 The impact on processor technology.

What we have observed, in the preceding sections, is that there are many different kinds of computer system. And they have quite varied requirements. As a result we find that processors, being the heart of the machine, are also quite varied in their capabilities, attempting to match processor architectures to computer system requirements. There are far too many processors in the marketplace at present to list them all, but we can make some attempt to list a few examples of each kind, examples of which are given in Table 4.1.

4.6 Mainframes and supercomputers

There was a period, probably between the late 1960s and throughout the 1970's, when any computer system was still a considerable investment for a large company. Consider an insurance company, where there might be 100 employees, all wishing to use computer resources from time to time on a daily basis. The cost of putting a computer on every desk would have been prohibitive in those days, and the desks would need to be very large! Instead, the concept of a mainframe, and mini-mainframe computer had already become established to solve this requirement economically.

The concept was straightforward: multiple users could access the same computer (probably set up in a convenient basement in the company tower block), as if at the same time. Typically access would be via a terminal (a screen and keyboard with very minimal hardware^[43]). These were often known as **dumb terminals** since they could do nothing on their own. Instead, they were simply remote interfaces to the 'real' computer system. This was also an early form of networked computing infrastructure.

Consequently, any user could run their application and perform a task right at their desk. In order to give the impression that everyone had equal access, the concept of time-slicing was utilised. Effectively, if every user had one millisecond of CPU time every second, and the terminals handled the local keyboard and display tasks, the impression was given that all 100 users were using the computer's resources simultaneously.

[43] The VT100 terminal is considered a classic example, and is still emulated in some software terminals.

PROCESSOR	TYPE	COMMENTS
INTEL CORE I7-9700K	GenPurp	Desktop/Laptop PC applications
AMD Ryzen 7 2700X	GenPurp	Desktop/Laptop PC applications
AMD Opteron™ X3421	Servers	Servers
INTEL XEON PLATINUM 8180	Servers	Servers (28 cores on one chip)
APPLE A7	Mobile	Used in iphones
Cortex-A57	Mobile	Mobile devices, touch-pads.
EXYNOS 9810	Mobile	Mobile phones
ARM Cortex-R8	Real-time	Designed for real-time uses
SIGMA DSP ADAU1787	DSP	Audio Processing
GEFORCE MX110	GPU	Optimised for laptops
AMD Polaris GPU	GPU	Used in some Radeon Graphics cards
GOOGLE TPU	AI	ANN accelerator Chip
IBM True North	AI	ANN Accelerator Chip
ATMEGA2560	Embedded	General use – e.g. microwave, heating
INFINEON XMC4000	Embedded	General purpose Embedded control

Table 4.1: A variety of processors with specific/-generic application categorisations.

Actually, a similar principle applies today, although the terminology and technology have changed. We talk about cloud-computing, thin-clients (the modern equivalent of a dumb terminal), and servers (akin to a mainframe). We will find out more about these later.

Going in the opposite direction, there were some computing domains where sharing out small slices of compute time to lots of general purpose users was not the primary goal. Instead, the demand was to run hugely complex computational tasks, with massive amounts of data, and to do so as fast as possible. A system designed to fulfil that kind of function is known as a supercomputer. These are the giants of computer systems, with disk storage measured in Petabytes [44] , hundreds if not thousands of processors, and terabytes of memory. Of course, such a precious resource can be in demand, and there may still be a small group of users competing for access. This requires a job-scheduling strategy.

It may be worthwhile defining what exactly we mean by fast, in the context of supercomputers. From the user perspective, the problem is to complete a hugely complex simulation or calculation within a reasonable time frame. For a PhD researcher, who has only three years to complete a PhD, a month may be a long time for one experiment. However, for an astrophysicist working on mathematical problems related to the physics of black holes, three or 6 months might be acceptable. For a weather centre, running a simulation in a few hours is probably the maximum that can be tolerated, otherwise the weather forecast would be issued the day after it has happened!. Some problems are so demanding that they can take several years to run a complete processing task, even on

[44] 1000s of Terabytes, or more accurately 1 Binary Petabyte equates to 1024 Terabytes or 2^{50} bytes.

Figure 4.3: Supercomputers. Photos showing (left) an older IBM mainframe and (right) late 1980s/90s CRAY EL98. With thanks to The Jim Austin Computer Museum (2019). The Cray EL98 had an entry level price of around \$340,000, and achieved up to 1 Gi-gaFlops.



some of the world's most powerful supercomputers. And of course this consumes huge amounts of power, and requires huge amounts of cooling, deep pockets, and patience.

[45] SETI: Search for Extra Terrestrial Intelligence, www.seti.org

[46] web URL:
<https://www.top500.org>
 This site also has regular news and technical features on the latest developments.

[47] GPU's have a class of computational capabilities that are highly mathematical, and often have high degrees of parallelism in terms of their computational units. This makes it possible to map complex scientific problems onto their hardware and achieve high throughput.

Because supercomputers are so expensive, some groups, such as the SETI Institute^[45] have started initiatives where compute resources are crowd-sourced: By taking a little bit of spare CPU capacity from every participant's desktop, often when those computers are sitting idle, they can aggregate huge amounts of computing power and achieve computing tasks that would otherwise require a dedicated supercomputer. Of course this relies upon goodwill and participant enthusiasm. For interest, Figure 4.3 shows a 1970's mainframe system, and a 1990's era supercomputer.

In some ways, supercomputers are seen as the pinnacle of current computational achievement, certainly for scientific computing tasks, and there is strong commercial competition to be top. In order to keep track of this progress, the Top-500 supercomputer index is updated regularly and lists all of the world's fastest supercomputers^[46].

Not surprisingly, the kind of processors used in these systems has also become very specialised. Specialised data processors known as vector processors, processors that contain thousands of simple number crunching elements (cellular arrays), and the adoption of graphic processing units^[47] for scientific computation are all examples. Often these chips can be very expensive.

4.7 The internet of things

A recurring theme over the past 30 years has been the idea of a world in which computers are everywhere, performing all kinds of functions, and

yet they are quietly working away behind the scenes. The latest embodiment of this idea is referred to as the **internet-of-things**. However, this has also been known widely in past times as **pervasive computing**, **ubiquitous systems**, and **the disappearing computer**. The common thread in these very similar ideas is the concept of embedding processors in a multitude of everyday things and environments, in order to achieve the perceived benefits of large amounts of operational data and interactivity.

Often, these days, such ideas are coupled with the concept of **machine learning**^[48]. The difficulty in crafting all-encompassing algorithms using hand-engineered algorithms has made the alternative of **Artificial Intelligence** (AI) very attractive. AI has the unique ability to aggregate large amounts of apparently subtly related data into coherent inferences, and then to create reactions that appear intelligent.

For example, an intelligent shower that learns a home-owner's regular habits, can pre-heat water at just the right time to use low-cost energy and provide hot water when needed. The shower may also inform the central heating system that the downstairs heating system should come online, ensuring that the rooms are warm even if the resident gets up and showers earlier than normal. Linking together lots of relatively simple systems makes a more complex holistic system with a higher degree of awareness. Of course they also present security risks, as most computing systems do.

In order to achieve this kind of functionality, there are several key requirements. An embedded processor is required in each device, a network connection is required, typically wireless for convenience of installation and maintenance, and this then leads to further considerations: security, privacy, resilience, and so on. To meet this need, processor manufacturers have developed low power processors with inbuilt wireless communications and cryptographic functions, along with the other expectations of an embedded processor, such as particular IO interfaces, non-volatile memory, display controllers, support for sensors, and so on.

4.8 Portable, wearable, implantable

Taking the principles of pervasive computing to the extreme, not only do everyday devices have the potential to be computerised, but people too. Wearable computers already come in many guises. Smart watches,

[48] The idea of machine learning is far from new. It dates back to the 1960's. However, the lack of cheap and powerful computing resources has meant that it has taken a long time to begin to reach its potential.

medical monitoring devices, digital hearing aids, specialist environmental monitors (such as radiation and gas detection), and of course our smartphones: probably the most successful computing device ever designed. The ultimate step for some may be **implantation**, and so far this has been generally used quite conservatively, examples being pacemakers, glucose pumps, very specialised brain implants, and so on. More ambitious researchers have experimented with implanting simple devices under the skin for non-medical applications. The barriers here are as much ethical as technical. It is difficult to predict how this will develop over the next 20 years.

In all of these cases, very low power devices are the star players: the ability to do complex computing tasks with biological data for example, yet consume minute amounts of power. Ultimately, the potential for devices to be so low in power consumption that they can collect power from their environments and not have a replaceable battery, will make some devices possible that are currently impracticable. This concept, known as **power harvesting**, can take a number of forms, including power from light (**photovoltaic**), power from heat (**Peltier** devices for example), power from motion (miniaturised dynamos), power from vibration (the **Piezoelectric** effect for instance), and power from radio waves occurring naturally in the background (**RF power harvesting**).

4.9 Parallel processing

As we found out in Chapter 3 it is possible for multiple instructions to execute together simultaneously, in other words, instruction level parallelism. However, there are limits to this method of increasing performance. Fortunately, it is also possible to have multiple distinctly separate processors in a single system (a multiprocessor system). Indeed, where servers are concerned, it is not unusual for rack-mount systems to host tens or hundreds of processors^[49]. However, these systems as a whole would typically be described as **loosely-coupled systems**. These are systems where the processors work fairly independently in assigned tasks and primarily communicate with other processors via the operating system. Such systems can be used to achieve **parallelism** easily, where a task needs to be repeated many times, with the same algorithm applied to a number of different data-sets for instance. Ten CPU's running ten jobs each will get the work done faster than one CPU doing 100 jobs.

A **tightly-coupled system**, on the other hand, is one in which processors work closely together, often using **dedicated communication links**^[50]

[49] In a modern system a single physical processor chip might have 4, 6, or 8 CPU cores on the same chip. Consequently, a rack-mounted module with 8 processor chips could easily have 64 cores.

between them in order to exchange information without the full baggage of the operating system. This is the kind of system we generally think of when we talk about parallel processing arrays: many processors working together in close cooperation to achieve a particular highly complex computational task, such as nuclear physics or weather simulations. Here, a loosely-coupled system would not be able to cope well with the inter-dependence of the work as divided across processors, because the **data-flow** between processors would be too demanding.

[50] This can be via a dedicated communications link, but could also be via a **shared memory**.

4.9.1 Speedup vs Amdahl's Law

An important point about the type of computing jobs applied to parallel processing systems, is the **achievable speedup** obtainable with a given workload, as compared to the best possible speedup implied by the hardware.

Although we might imagine that 1000 processors will do a job 1000 times faster than a single processor, there is an important 'fly in the ointment' to be aware of here: For a given workload, there will be a portion of that workload, **P**, that is parallelisable, and a remaining portion (**1-P**) that is serially constrained. This is the basis of **Amdahl's Law**.

Definition 4.9.1 Amdahl's Law

Amdahl's Law states the following:

*For a workload with proportion **P** capable of optimisation, maximum speedup is:*

$$S = \frac{1}{(1 - P)}$$

So if **P** was 50%, then maximum speedup would be 2.0, no matter how much the parallel part of the program can be speeded up. In a sense this is obvious: if the parallel part was speeded up ad infinitum, and then requires zero time, we are still left with the serial part (half of the total in this case), and can never overcome that. Of course in a real system, even the part being optimised will always still take some remaining time, even after optimisation.

One of the consequences of this is that parallel processors will almost certainly not deliver a speedup of **n**, when **n** processors are used. Consider

Table 4.2: Amdahl's Law applied to a parallel processing system. This table compares speedup versus number of processors 'n', and proportion of task that can be parallel (P).

	Number of processors 'n'			
	1	10	100	1000
P = 1.0	1.00	10.00	100.00	1000.00
P = 0.9	1.00	5.26	9.17	9.91
P = 0.5	1.00	1.82	1.98	2.00
P = 0.1	1.00	1.10	1.11	1.11
P = 0.0	1.00	1.00	1.00	1.00

the following formula, which uses the principle of Amdahl's Law to give us the speedup for a specific parallel processing system.

Definition 4.9.2 *True Speedup*

Actual Speedup for n processors optimising a portion P of a problem, by a perfect division of effort:

$$S_n = \frac{1}{(1 - P) + (\frac{P}{n})}$$

Here, **P** is the proportion of the workload that can be parallelised (a value between 0.0 to 1.0), and **n** represents the number of processors dividing the work up.

When this data is calculated for various values of **n**, we get a set of data and find that this can be tabulated as detailed in Table 4.2

What we see is that even when P=90%, the best speedup available with 1000 processors is only 9.9 times speedup, which is very disappointing. Even with P=99%, the speedup for 1000 processors is only 91 times. This seems far less benefit than might be expected for the implied cost.

An important lesson here is that high degrees of speedup are only possible when a workload has a very high degree of parallelisable code content. Examples include repetitive tasks on successive sets of data with little or no dependencies between data sets (because dependencies necessitate the serialisation of workloads).

Parallelism, however, is not a problem that always yields up solutions easily. In many cases, there is a dependency between parallel parts of a problem, or some contention for a resource such as a block of data in memory or on disk. This problem is generically known as **interference**, and can occur in many different ways.

4.10 Summary

In this chapter we have explored the diversity of ways in which computer systems might be utilised, and formed an overview of how widely they are indeed used in practice. One thing that is very apparent is that a single processor or computer design can never be efficient for all cases: there is no 'one-size-fits-all' solution. Instead, we need many different kinds of processors, ranging from very simple microcontrollers though to highly specialised application specific processing engines, designed to perform specific tasks with high efficiency.

So, a key role of a computer engineer must be understanding the computational demands of a problem, and selecting, or designing suitable components to fulfil those expectations. Only then can computer designers meet the many competing demands of power, cost, size, and capability.

4.11 Terminology

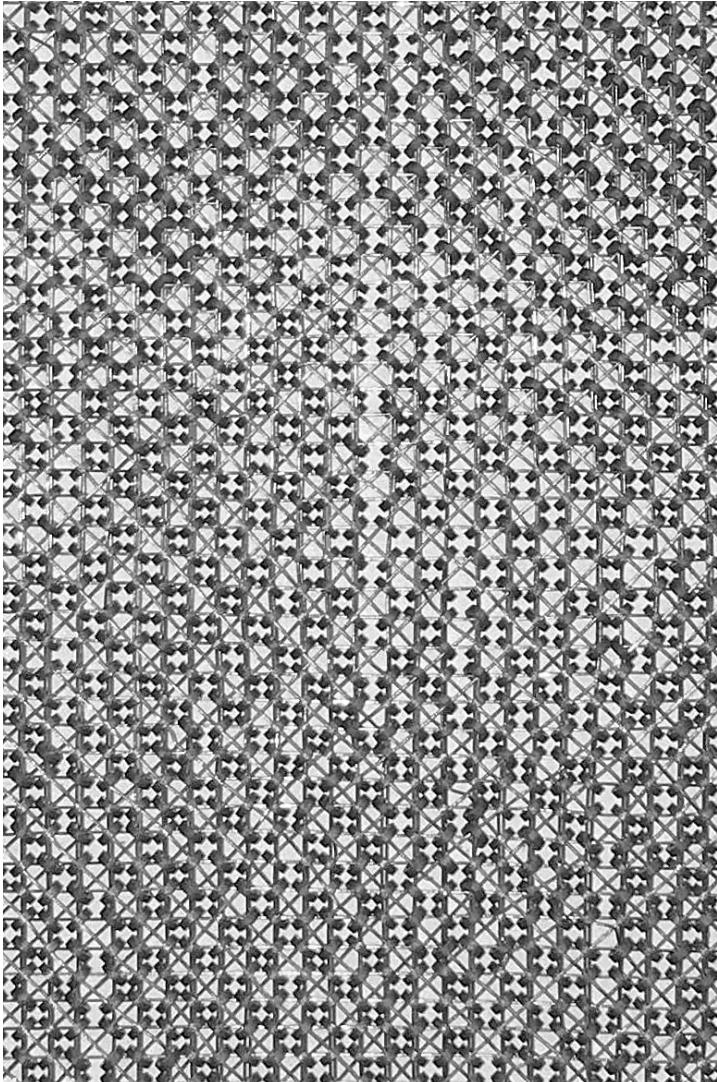
Terminology introduced in this chapter. You may wish to review this list and ensure that you can describe in writing the meaning of each of these terms for improved familiarity and recall.

Amdahl's Law	Application-specific
Artificial intelligence	CPU time
Data-flow	Digital signal processor
Dumb terminal	Embedded-system
Fault-tolerance	General-purpose
Hackable	Internet-of-things
Job-scheduling	Loosely-coupled system
Machine learning	Mainframe
Mobile-processor	Parallelism
Peltier device	Pervasive computing
Photovoltaic cell	Piezo cell
Power harvesting	Real-time computing
Resilience	RF power harvesting
Safety-critical	Smartphone
Speedup	Supercomputer
Superscalar	Tablet
Tightly-coupled system	Touchscreen
Ubiquitous system	Web-Search-Engine

These terms are defined in the glossary, Appendix A.1.

5

Making Memories



Close-up image of Magnetic core-store Memory array (1960s). Photo: Chris Crispin-Bailey.

5.1	Memories, past and present	68
	Getting physical	70
5.2	Non-volatile memories	71
5.3	Volatile memories	73
	Static Random Access Memory (SRAM)	74
	Dynamic Random Access Memory (DRAM)	74
5.4	Memory performance strategies	75
	Access time and cycle time	76
	Memory access protocol	76
	Memory width	77
	Single-cycle versus Burst-mode access	77
	Page mode access	80
5.5	Cache memory	82
	Multilevel cache	85
5.6	Cache management policies	87
	Mapping policies	87
	Replacement policies	89
	Write-back policies	90
5.7	Avoiding errors	92
5.8	Programming considerations	93
5.9	The future of memory devices	95
5.10	Summary	97
5.11	Terminology introduced in this chapter	99

5.1 Memories, past and present

Computer memory systems have evolved very significantly in the past 60 years. In some ways, they have changed beyond recognition, but in other aspects, for example, the way memory is organised and used, they still owe much to the earliest days of computer system design and construction.

In the simplest definition, a computer memory is a component that **retains information** (data) for a period of time. This should not be confused with the concept of a storage device, which we will examine later. Storage devices are secondary data storage mediums, often (but not always) capable of being removed from the computer system, and are very typically **non-volatile**. By non-volatile, we mean that data is retained even when the system is disconnected from the computer and has no power source attached. A computer memory, on the other hand, is typically a permanent part of the computer system, and can be either volatile or non-volatile.

Early computer systems used some fairly exotic methods to retain data in the form of a data memory. In the days when computer systems were still being built from radio valves, it was of course possible to build logic gates and perhaps also to build storage elements by using radio valves in a similar way to transistors. However, the size of a single bit of storage would then be quite large, a whole byte even larger, and a 'reasonable' amount of computer memory, let us say a puny 256 bytes, would fill shelves full of such electronics. Clearly, if memories were to get any larger, some new technologies were going to be required.

One system, the **mercury delay line**, used sound waves travelling through a tube of mercury to represent binary zeros and ones as pulses of acoustic energy. By capturing the sound waves at the other end and then feeding them back into the start, the pulses could travel along the delay line indefinitely, at least in theory. This meant that data bits were held within the system as long as power was maintained, and nothing interfered with normal operation. One delay line could hold of the order of 500 bits of data. Though more reliable than radio valve logic, the downside was the need for 20kg of mercury for each delay line. Be glad that computers do not still use this technology... a modern laptop with even a small 4Gigabyte memory would weigh over 1.3 million metric tonnes!

Another system, known as **magnetic core-store**, used very small magnetic rings threaded onto a **row and column matrix** of wires, to form a

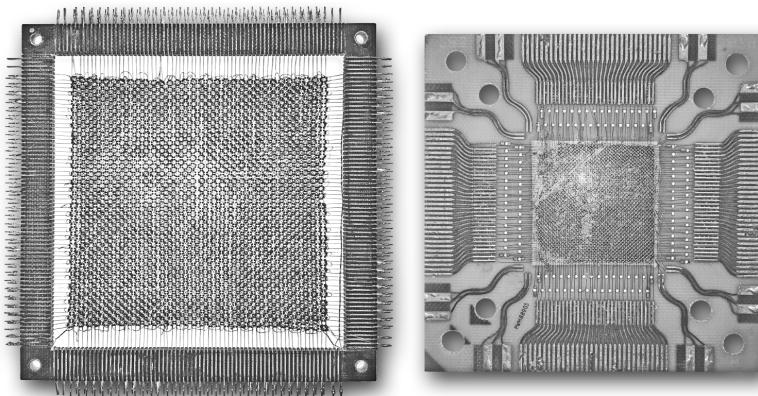


Figure 5.1: core-store Memory Modules. Showing a 1kbit module (left) and a later, more compact, double-sided card module (right). The term 'core-dump' is still used to refer to dumping memory content to a diagnostic-file when an error occurs.

grid of binary storage elements. Indeed, we still refer to memory organisation as rows and columns internally. An example of core-store is given in Figure 5.1. Core-store was relatively compact. A 1 kilobit module might be 10 x 10cm across and 1cm deep, and weigh 40 or 50 grams.

This memory system was still terribly slow by modern standards. Even so, several variations of core-store memory were used to build the Apollo guidance computers^[52], and managed to get men to the moon and back. Without core-store it simply would not have been possible. One small step for man, one giant leap for computer memory!

Needless to say, computer systems engineers were not satisfied with the memory technologies they had to make use of, and it was not long before all-electronic memories were starting to be developed.

As we learned briefly in Chapter 2, transistors can be combined into logic gates, and logic gates can be combined into simple data latches with the capability of storing a single binary bit. However, as 6 transistors are needed for one storage cell (often referred to as a **6T bit-cell**), this was a bulky form of memory. One bit could easily occupy a few square centimetres of circuit board space. Later, the advent of **integrated circuits** allowed many transistors to be squeezed onto a single chip of similar size, and before long, computer memory chips with many kilobytes of data were beginning to become available.

If we fast-forward to modern day computer systems, we find that individual chips, of the same size as those early memory chips, have capacities as high as 10's of giga-bits. And contain billions of transistors. When a few of these chips are combined on a small circuit board, we get a

[52] The Apollo-11 Guidance computer (AGC) used both erasable core-store arrays, and a permanent memory version of core-store known as **rope memory**, in which the guidance programs were stored.

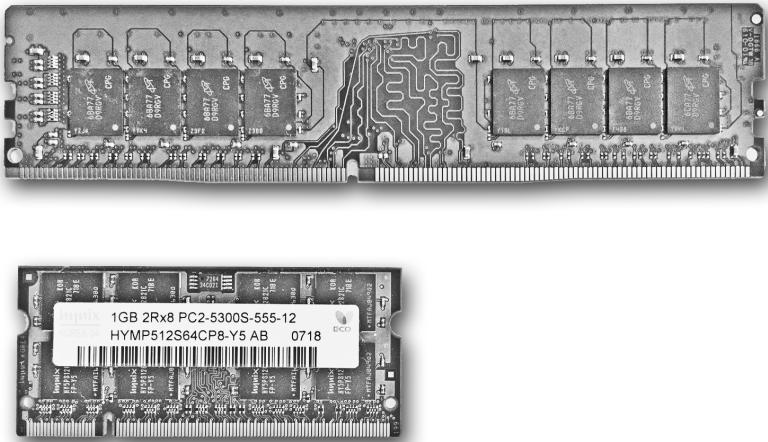


Figure 5.2: Examples of **DIMM modules**: Dual In-line Memory Module. **DDR DRAM** (top), and **SDRAM** (bottom).

module known as a **SIMM** or **DIMM**. Some DIMMs are shown in Figure 5.2. These modules typically come in capacities of 8, 16, 32 or even 64 Gigabytes. Their cost is very low considering how much data capacity they have.

5.1.1 Getting physical

[53] Small? Bill Gates, co-founder of **MICROSOFT**, and inventor of **MS-DOS**, was once claimed to have said 640 Kilobytes would be more than enough for most users. Never make predictions about computer technology!

It is noticeable that a computer system that perhaps has only 4 Gigabytes of memory can still run a large number of programs or applications simultaneously, and each of those programs potentially seems to have a lot of data in play: perhaps much more than the memory capacity. So how come we can fit so much into a relatively small memory?^[53] The answer is that the hardware of the system, the actual memory chips, are only part of the equation.

The electronic memory chips that make up the system memory are sometimes referred to as **physical memory**. This is to reflect the fact that for every byte of physical memory, there is some hardware, some transistor, some chip, where that data is physically held whilst the computer operates. These are known as the physical memory. It also helps to differentiate that memory from a concept known as **virtual or logical memory**. We will say more about this later, as this is actually more of an operating systems concept, and not so much to do with the hardware of the system. For now we can just say that virtual memory is an illusion of having a much bigger physical memory, when in reality most of that data is stored on a storage device such as a disk unit and moved into

the real physical memory only when it is needed. The result is that a system with 4 Gigabyte of physical memory may appear just like one with 32 Gigabyte of physical memory. The only noticeable difference is that things may run much slower. We will come back to this point later.

5.2 Non-volatile memories

In actual fact, there is not one kind of computer memory in current use, but a wide variety of them. We can divide these into categories that help us to understand what their properties and uses are. One major category is **non-volatile memory**, and there are many sub-types of this technology:

Non-Programmable (Read-Only) Memory:

- ▶ Permanently manufactured with a particular data content.
- ▶ Manufactured as an Integrated circuit (IC) with fixed content.
- ▶ Expensive to design, and to update (needs redesign).
- ▶ Cheap when mass produced.
- ▶ Sometimes referred to as Read-Only Memory (ROM).
- ▶ Data content is never lost.

One-Time Programmable Memory

- ▶ Manufactured with an array preset to a blank state.
- ▶ Can then be programmed to a particular data content.
- ▶ Once data is programmed it can never be erased.
- ▶ Cheaper to manufacture.
- ▶ Changes to content just require changes to the data being 'burned' into the memory.
- ▶ Data content is never lost.
- ▶ Often known as PROM (Programmable Read Only Memory).

Programmable Non-Volatile Memory.

- ▶ Can be erased and reprogrammed many times.
- ▶ Data is retained even when no power is present.
- ▶ Erasure and reprogramming can be done in several ways:
 - **EPROM:** Erasable Programmable ROM, is erased by UV light, and programmed by electrical signals.
 - **EEROM:** Electrically Erasable ROM, erased by an electrical signal, programmed by electrical signal.

- **EAROM:** Electrically Alterable ROM, is reprogrammable by electrical signals.
- **Flash Memory:** a variation of EEROM, has large capacity, cheap, relatively fast to read and write data.

These represent the main types of non-volatile electronic memory in common use currently. The obvious question is, why are there so many kinds of memory? There are many reasons, but a key one is the requirements of the application for which the memory is to be used.

A true **ROM** (i.e. one that is manufactured as an IC with a fixed content) is used where a permanent data content is required, and this is to be mass produced in very large numbers. ROM is used where content must never be altered. For example, a program in an implanted medical device is safety critical and must never be able to be corrupted or deliberately hacked. ROM is a safe solution here.

However, it is very expensive to manufacture a ROM from start to finish. The chips may only cost \$5 but the design cost could be \$250,000. To avoid such large up-front costs, we often use PROM instead.

A **PROM** is very similar to a ROM in use, but comes from the factory **un-programmed** (or, in other words, blank). The customer can then buy thousands of chips and only program a few hundred chips at a time, thus allowing for revisions of the data content at a later date. Once the data is **programmed** into the chip it is as permanent as ROM and can never again be modified.

For a range of less safety critical applications, the desire to have memory chips that act as if they are permanent, but can be altered under tightly controlled conditions, has led to the development of **EPROM**. An EPROM is like a ROM that is supplied blank, can be programmed, and then acts very much the same. However, at some later date, the chip can be placed in a **UV light-box** and **erased**, ready to be reprogrammed. This allows for modifications of devices 'in the field', provided an engineer can be physically there. It is not possible for EPROM to be erased by a remote hacker simply by sending a virus or a rogue command, so it is considered relatively secure.

Other types of non-volatile memory have different categories of use. All of the remaining types can be erased by some form of **electrical signal**, and therefore are less secure in some respects, if the system they are built into allows control of those signals to be accessed. This is often the case, since the purpose of these type of memories is to allow **semi-permanent** data storage but also to permit firmware updates to be possible.

For instance, a product might be configured to have certain factory settings, and during its use after being deployed to a customer, those settings might be updated. So-called 'Smart-TV's' are a good example. They contain non-volatile memories, where their software (effectively firmware) is stored. Data broadcast over the TV radio spectrum or via internet connection is received by smart TVs and used to overwrite this firmware as and when updates are released by the manufacturer. Of course this can be done entirely remotely, and therefore presents a theoretical risk of a hacker doing this in an unauthorised fashion.

Most **electrically erasable** non-volatile memories are relatively slow to read and write. However, one particular type of memory, known as **flash-memory**, is able to perform at reasonably high speeds (though nowhere near as fast as volatile memories). These memory chips are used in **USB flash drives**, also known as **memory sticks**, because they are relatively fast and have high capacities.

However, we have strayed into the domain of storage devices! Yes, it is true that a flash memory stick is a memory, but it is also a type of storage device (it is removable, non-volatile). As we shall see later, the way storage devices are built is beginning to overlap with how memory operates, but they should still be viewed as two distinct and different things in practical use.

[54] This is not entirely true. EPROMS can lose their programmed data over very long time periods, of the order of decades. Some other types of memory can become corrupted over very long time scales. Often we see fit to ignore this problem; after all computers are often obsolete so quickly that this problem may never be observed. However, it is one that cannot be forgotten entirely.

5.3 Volatile memories

Just as non-volatile memories can retain data indefinitely, even without power^[54], the **volatile memory** can only retain data when powered up and operated correctly. There are currently two major types of volatile memory: the **Static Randomly Access Memory (SRAM)** and the **Dynamic Random Access Memory (DRAM)**^[55]. Both of these memory technologies are found in most general purpose computers, and the reasons for having both will become clearer as we discover more about them.

Certainly **RAM**, **ROM**, **PROM**, **EPROM**, all appear to be very similar, if not identical, in terms of how data can be read from them. Other non-volatile memories may have certain read and write constraints, which we will not dig into at this point.

[55] Incidentally, the idea of RAM being **randomly accessible** is a bit of a 'red herring'. Almost all modern memory devices allow data to be accessed in a random fashion. In early systems such as mercury delay lines, data had to be accessed in a sequential pattern. The term 'randomly addressable' came about to highlight the new non-sequential capabilities of newer and (then) more advanced memories.

5.3.1 Static Random Access Memory (SRAM)

The SRAM, as we mentioned earlier, is based upon an arrangement of transistors, the 6T bit-cell. There are other variations which have specialised properties, but the standard SRAM storage element is based upon this idea.

As this is not an electronics or a digital engineering course specifically, we will not delve into precisely how six transistors can store one bit of data, but this can easily be researched by interested readers.

One of the main principles of static RAM, is that it operates using purely digital circuitry, and the 6T bit-cell will remain in a zero or one state for as long as power is provided. However, even a momentary loss of power is enough to lose some or all of the memory content, and leave it **corrupted**^[56]. For this reason, most robust computer systems have power supplies that maintain a smooth delivery of power, even if momentary glitches might occur on the supply to the building.

SRAM is typically very fast, but because of the circuit design, it occupies a lot of chip space per bit, at least when compared to DRAM. Another way to express this is to say that SRAM has lower **storage density** than DRAM. This means it is more expensive per bit for a given piece of silicon containing an array of bit-cells, and one megabyte of SRAM may be many times more expensive than one megabyte of DRAM. Therefore, if you wish to have a huge memory capacity, then DRAM appears to be attractive in terms of cost and space occupied on the system motherboard.

5.3.2 Dynamic Random Access Memory (DRAM)

[57] A capacitor is almost like a mini-battery, but it loses its charge quickly.

[58] This is due to the capacitive bit-cells they use, which lose their charges over a period of milliseconds. Thus they must be topped up (refreshed) frequently in order to retain their data values.

Because the 6T SRAM cell occupies a relatively large amount of silicon space, another competing technology has also become widely used. DRAM uses a different way of retaining a binary one or zero. By using a small **chargeable component** known as a **capacitor**^[57], an electrical charge corresponding to a binary '1' can be represented when the capacitor is charged, and when discharged this represents a zero. Such **capacitive bit-cells** are smaller and therefore more bit-cells can be packed into a DRAM chip than an SRAM chip, leading to higher storage density and higher overall capacity. Of course there must be a downside, otherwise SRAM would have become obsolete long ago. DRAM is relatively slow compared to SRAM, and has certain additional circuit operating requirements, known as a **refresh cycle**^[58]. These factors add to the

complications of engineering a system using DRAM. But because it is very inexpensive, it is widely used for what is known as **bulk memory**; in other words, the main provider of physical memory in the computer system.

DRAM's come in a number of formats, the most popular of which are small circuit boards known as **SIMM** and **DIMM modules**. These can be slotted into a socket on a motherboard of a computer system, and just as easily removed, making the system easily configurable and upgradeable.

However, because DRAM **access time** (the time taken to read a data value) is slower, this presents a problem for designers. Memory speed is typically slower than CPU speed in terms of technology progression over time. DRAM access speeds are often anywhere between five and twenty times slower than contemporary processor clock frequencies, and SRAM can be one to three times slower. A one-to-one speed rating is good because it means the CPU never has to wait more than one clock cycle for SRAM to respond, thereby maximising performance.

This potential slow relative speed of memory means that a CPU could spend a lot of time simply waiting for DRAM to respond to a request for data instead of working on it, and as a result the CPU will not deliver its peak performance capability. To get around this problem there are multiple solutions, with various cost implications. We will explore some of these in the next section.

5.4 Memory performance strategies

As we have just highlighted, memory speed, which so far we have defined only in terms of access time, is a critical factor for computer performance, and unfortunately for us, large and inexpensive memories are relatively slow. Consider a CPU operating at 3.9 GHz, and a DRAM operating at a relatively fast 1330MHz. There is a three-to-one ratio between CPU speed and memory speed if we optimistically assume that one clock cycle yields one memory access. This already seems to be a significant problem, but there are more factors to consider to get the whole picture. We probably need to learn a little more about memory before we can see the whole picture from an informed viewpoint.

5.4.1 Access time and cycle time

Both SRAM and DRAM (indeed any IC-based memory device) have several potentially different timing properties: **Access time** is the time it takes to read a data value onto the system bus after completing the process of requesting that data from the memory. Ideally this will require fewer nanoseconds than the length of one CPU clock cycle period, such that the CPU can request data during one clock cycle and receive it before the start of the next clock cycle (when it wants to use it). Any delay longer than that results in whole extra clock cycles having to be spent waiting for memory to catch up^[59]. These are known as **wait-states** or **wait-cycles**. If we have a **zero-wait-state memory** system, then we are in good stead, because the CPU gets the data within one clock cycle.

[59] We can have 1, 2, 3 clock cycles, but we cannot have fractions of clocks. If access time is even a picosecond too slow, then an entire extra clock cycle will be needed.

Cycle time is another critical property of memory. Whereas it may take a particular amount of time to access data, the memory cycle time represents the time required for that memory chip to set up a request, access the data, and then also recover internally from its effort and be ready for the next access. The cycle time will therefore typically be much longer than the access time, and even more so after a write operation.

5.4.2 Memory access protocol

Taking a read operation as an example, in order for a memory device to provide a data value, it must first be told what address to read. This process requires the CPU to place an address on address lines of the system bus, and to assert the correct timing signals to tell the memory device to perform a read operation. This required set of actions is called the **memory access protocol**. We should be able to determine from this that a standard read operation must therefore require at least two clock cycles, but there are reasons why this is likely to be more, as we will see shortly.

Consider the following scenario:

So, let us suppose the CPU requests data from DRAM, and gets a response in three clock cycles. But, on further investigation, it is found that the DRAM has a full cycle time of 5 clock cycles. Now we can see that the CPU cannot request another data item until a further two clock cycles have elapsed after data access. This means that the CPU can only read a new data item once in every 5 clock cycles in this case.

The lesson here is that we cannot always rely upon access time to determine system performance. It is cycle time that ultimately limits the use of memory bandwidth.

Suppose that a CPU has a clock frequency of 1000MHz, and it encounters the memory system we have just described, with a memory cycle time of 5 clock cycles, this means it can only access 200 million data items in memory per second. For peak performance we might want this to be 1000 million data access per second. Fortunately some techniques exist to try to augment this disparity.

5.4.3 Memory width

One thing we have not yet considered is **memory bus width**. If a memory location contains 8 bits, in other words, 1 byte, then 200 million memory accesses per second means 200 Million bytes of data can be accessed per second (or 190 Binary Megabytes/sec). We refer to this measure as **memory bandwidth**^[60].

It might be obvious therefore that to increase memory bandwidth, we could simply make the memory bus wider. Why not have a 16-bit or 32-bit wide memory instead of 8 bits?^[61] A 32-bit memory with a cycle time of 5 clocks would deliver 800 Million bytes/sec, instead of 200 Million bytes/sec.

In recent years we have seen computer systems moving toward 64bit system architectures, partly for the reason that it delivers wider memory busses and increased memory bandwidth. In-fact, all devices connected to the system bus will enjoy improved bandwidth, and we sometimes refer to the total bandwidth available as the **system bandwidth**. However, there are limits: simply increasing memory width to 256 bits, for example, won't necessarily give the gains that might be expected^[62]. Such schemes are also likely to complicate the circuit board design of the system considerably, demand more expensive system components, and so on, and so it is not an unlimited panacea.

5.4.4 Single-cycle versus Burst-mode access

In the single-read DRAM **access policy**, as observed in Figure 5.3, the procedure required to perform a memory access begins when the CPU sends an address as a **row address** and a **column address**, taking one clock cycle for each transfer to the DRAM. The DRAM then spends

[60] Unless specifically stated we will use decimal megabytes here, and for bandwidth and data rate calculations generally, as we are taking about data transfer on a clocked system. Memory capacity, on the other hand, is almost always specified in binary megabytes.

[61] Just as adopting a four-lane highway rather than a one-lane road will potentially increase traffic flow.

[62] One of the reasons is that, as data width gets larger, the probability that all of the data bits will be useful for each and every read starts to diminish, and the system may then simply end up frequently reading unwanted data.

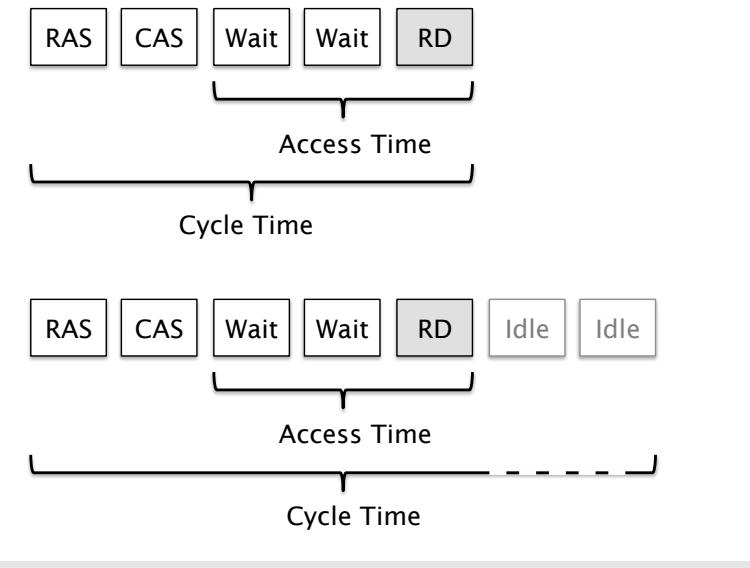


Figure 5.3: DRAM single read operation. Showing actions performed in successive clock cycles, assuming an access time of 3 clock cycles. The top case represents the simple scenario, whilst the second case shows the possibility of idle cycles being added at the end of a memory transaction.

[63] There are various reasons why this may be the case. The CPU may simply not be ready to start a new read, there may be additional circuit effects to consider, and the memory itself may have constraints. Manufacturer datasheets, for example, often have several complex design parameters to consider when building a real system. In particular, the requirement to issue an 'RAS precharge' command, when access switches to another row address, can create delays of multiple cycles before the DRAM is able to properly start a new memory access.

time looking up its data (internal readout), and then provides the result. There is sometimes also a period of extra 'idle' time whilst the DRAM completes its additional cycle time requirements^[63]. Only then can the CPU start a new request.

However, given the structure of programs and the way data is organised in memory, we typically find that a CPU may be accessing multiple memory locations in a linear sequence. Therefore, sending a new and updated address for every one of these successive accesses, and then also waiting for the DRAM to complete its cycle time, is very wasteful.

Burst mode is a very useful innovation in DRAM technology that boosts memory performance, in part overcoming the penalties of cycle time, and certain other factors that limit performance.

In burst mode, as shown in Figure 5.4, a DRAM can be instructed to read a number of **consecutive** memory contents out onto the bus one after another. It is then only necessary to provide the first row/col address, since the other data items follow directly afterward in a linear sequence, and because the DRAM is still actively reading the same block of memory during this burst sequence, any additional end of cycle delays are only encountered one time, at the end of the burst.

Now let us compare these examples, making some assumptions such that, for both cases, the access time assumes two wait states before a read,

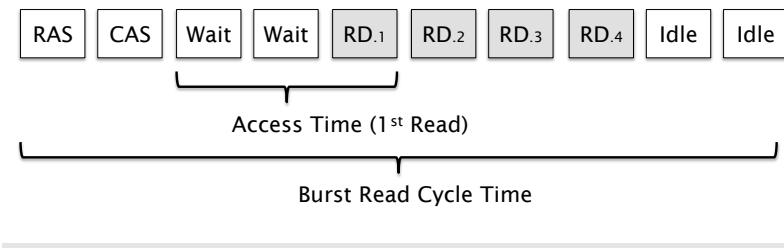


Figure 5.4: DRAM burst read operation. Showing actions performed in successive clock cycles, assuming a burst length of 4.

and also assuming that the cycle time includes two idle clock cycles after the read.

It can be observed that in the first case, with the assumptions as shown in Figure 5.3, and using the single-read DRAM memory access protocol, then one random read operation requires a total of 7 clock cycles if idle cycles are assumed present (or 5 without any). We would therefore require 28 cycles to read four consecutive values, repeating the single-read process four times. The figure of 7 clocks per access represents a rather poor level of performance, especially as a processor might desirably want a new memory value available in every clock cycle.

However, the second example, shown in Figure 5.4, uses a burst-mode access protocol, and can read all four locations in only 10 clock cycles, with the same assumptions. This translates to an average of 2.5 clocks per access, and clearly much improved performance. Indeed, this speed is approaching three times faster than the simple random access case. There are several reasons for this. We may note first of all that only the first read access requires the longer delay with wait states, and successive reads in the burst are then single-cycle operations. Secondly, if (as in this case) the idle cycles are assumed to be needed at the end of a read, they are incurred only once, at the end of the burst, rather than potentially four times^[64].

Useful though burst-mode is, it cannot be used continuously in real programs however. This is because there will be sections of the program where burst mode is heavily used: primarily linear code sequences, and linear accesses to data sequences, in successive memory locations, but then the memory addressing pattern will reach a point where it will jump to a different part of memory, forcing the current burst to terminate early, and a new memory access to begin. If this happens frequently then the average length of a burst access may be less than the maximum supported burst length. In practice, what we will get is a mixture of the multiple cases, linear and non-linear patterns, based upon the kind of program and the kind of data we are dealing with.

[64] Why 'potentially'? If you appreciated the earlier comments relating to RAS Precharge, you might see that four consecutive reads could remain within the same row, and therefore the idle cycles are incurred only once in the simple case too. That would mean only 22 cycles for four reads, an average of 5.5 cycles per read. However, taking this into consideration requires a much deeper knowledge of the memory system and specifications than we wish to pursue at this point in our study.

Now let us evaluate a burst mode scenario with the same access protocol just discussed, but in this case we will assume that the memory access pattern is a mixture of single cycle access mode and burst mode accesses, as and when they are possible:

A Mixed Burst Mode Scenario

Suppose we find that burst mode can be used 40% of the time on average, with our memory system as already discussed. What will be the average data rate of the memory if it is 32 bits wide and the system clock rate is 1000 MHz?

Average rate of access:

$$\text{Burst} = (40\% \times 2.5) = 1.00$$

$$\text{Simple} = (60\% \times 7.0) = 4.2$$

$$\text{Total average} = 1.0 + 4.2 = 5.2$$

Memory width = 4 bytes, therefore **average per byte = 5.2/4.0**

Therefore, we can access one byte every 1.3 clock cycles.

There are 1000 million clock cycles per second in this system, so we can access 1000 million divided by 1.3 = **769 Million bytes/sec**.

Thus, the achievable memory data rate of this scenario is about **733 MB/sec (using binary megabytes per second)**.

A further problem with using burst-mode is that the von Neumann architecture uses a single bus for both instructions and data. Therefore, we will find that the system bus is flitting between at least two sets of addresses at very different locations in memory. When we introduce the idea of instruction fetch, and overlapped instruction fetch, the estimates of memory behaviour become quite complex, and predicting exact performance is a challenge.

5.4.5 Page mode access

Page mode access, also known as fast page mode (FPM) DRAM, is another way of speeding up the slow standard DRAM access protocol, and as with burst-mode DRAM, this attempts to exploit the program and data addressing behaviour of the system favourably.

With burst-mode access, the system supplies one row and one column address, and then the DRAM will read consecutive locations. So the CPU

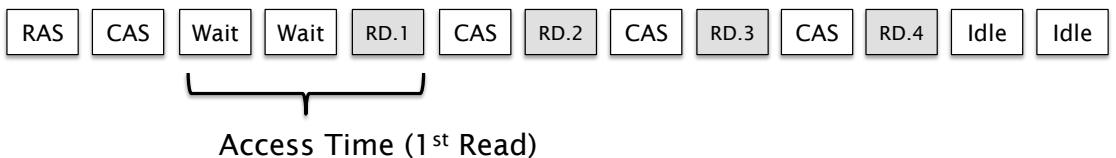


Figure 5.5: DRAM Fast Page Mode read operation. Showing each successive column address being provided in order to read the word within the row already activated by the first access within the page.

might request Row 6524, Column 122, and then it will receive data values at column 122,123,124, and 125 if the burst is four items in length.

A saving is achieved there by not having to send the same row and col values four times when the CPU knows it is not changing, and also by not having to complete a cycle time period after every read, just at the end of the burst sequence. However, if the whole burst sequence is not usefully exploited, then this efficiency starts to reduce quickly.

Fast-Page-Mode DRAM uses another variation of this idea, whereby the row address is transmitted the first time a new row address is encountered, and then only the column addresses are subsequently provided by the CPU, with the assumption that all of the columns in this page sequence are in the same row. Duplicating the scenario above, the CPU would request Row 6524, but then send column address 122, column address 123, column address 124, and column address 125. So a series of four reads, as in the previous examples, would look like Figure 5.5, with a total of 13 cycles for four reads (an average of 3.25 cycles per read).

This may initially seem to be just a less efficient version of burst-mode access. However, there is an advantage: the CPU can send **any column** address, in **any order**. So it could instead send Row address 6524, then column address 55, 12, 77, 78, and so on. It might be apparent that this is a more random ordering of addresses, yet within the same row (or page as it is known here), which is more in keeping with the semi-random nature of program code and data access patterns often encountered.

It would seem that FPM DRAM access mode is likely to be more frequently able to be used than burst-mode for many programs, but at the cost of reduced efficiency that can result from having to send a unique column address for every data item accessed.

There are several other variations of DRAM access protocols, indeed new schemes are being invented every year or two as DRAM design concepts

evolve. You may wish to research these and familiarise yourself with their mode of operation, but we will not discuss them further at this point. We will instead investigate the concept of cache memory, which has a lot to offer in smoothing over many of the problems just discussed.

5.5 Cache memory

Since we have established that a memory system composed of DRAM is far from perfect, even though it is relatively inexpensive, we are fortunate to have a further card to play in delivering a suitably high performance from our memory system. This is the concept of **cache**.

As we noted earlier, SRAM is smaller, more expensive, but much faster than DRAM. But DRAM is low cost, and allows us to have very large memories, even in everyday computers. Computer system engineers play a very clever trick to deliver both high performance **and** low cost from these memory systems, and as in the previous section, it exploits the behaviour of programs and data in the way they access memory.

It has been observed over many years of experience, that in very general terms at least, there is concept that 10% of program code is accessed 90% of the time^[65]. This is very much due to the way programs are structured: subroutines and loops being a major contributor. Often many of these program accesses relate to near or adjacent locations (**spatial locality**), or are used multiple times within short periods of time (**temporal locality**).

A similar story can be told for data held in memory, but this is much more dependant upon which kind of data it is, what is being done with it, and how the programmer has chosen (wisely or not) to organise the data in memory^[66].

For example, let us suppose that a program searches through a huge list of perhaps 10 million data values to find one particular item. Then, it may be the case that the CPU needs to access every data item exactly once. There is no opportunity to reuse these data values, so storing them in cache would be wasted effort.

On the other hand, a program performing mathematical manipulation of an image consisting of 10 million pixels (each one a data item) may need to access 10% of the pixels 10 times, and the rest of the pixels only twice.

[65] This is of course quite dependant upon the actual program or programs being studied, it might be 20/80, 15/85, or another ratio, but the principle that a small **subset** of the total program is used the majority of the time is still very prevalent.

[66] For example, a programmer who knows much about the underlying hardware might organise data accesses to exploit the system in ways that allow more optimal memory traffic capacities.

The behaviour of these two cases is quite different. The second case is much more amenable to exploiting cache than the first case. Indeed, the first scenario may actually make system performance worse by wasting cache resources without any benefit.

Let us keep the story simple, and assume that data and program code both appear to follow the case where 10% of the memory content is accessed 90% of the time. Incidentally, we usually refer to the 10% portion of memory content that we wish to access frequently (and therefore efficiently) as the **working set**. So how does cache exploit this principle?

The concept of cache is to provide a small amount of fast SRAM alongside a large amount of slow DRAM. Indeed, we included cache as a system block in our von Neumann system diagram (Figure 2.1). When considered together, the cost of the whole memory system is still relatively low. However, we shall see that this can boost performance considerably for that small extra investment.

Imagine we have 512 Mbyte of SRAM-based cache, and 4 Gigabyte of DRAM, a good ratio (8 to 1) if we expect cache to keep a copy of 10% of memory contents. The SRAM has single-cycle access time. The DRAM has an initial access time of 3 clock cycles, and both memories also require row and column address cycles^[67].

Each time memory is read, SRAM is accessed. If the data is present, a cache hit then delivers a read in one clock cycle in this case. If a cache miss occurs, the DRAM also then needs to be read, a further 5 cycles. During a miss, the cache controller (a special control circuit) copies that data into SRAM so that next time it is accessed it can just get the data from SRAM (a cache hit).

Now, suppose that we have a hit-rate of 90% based on our assumptions of program behaviour as mentioned earlier: what will be the average read time?

[67] In practice, this means that the SRAM read time is 3 cycles (2+1) and the DRAM read time is 5 cycles (2+3).

Cache Example

90% of Memory Access with a cache Hit:

This uses SRAM, therefore read time = 3 clock x 90%

10% of memory access with a cache miss:

This uses DRAM, therefore read time = 5 clock x 10%

But a cache miss also reads SRAM = 3 clock x 10%

Total average read time:

$$= (90\% \times 3) + (5 \times 10\%) + (3 \times 10\%) = 3.5 \text{ clocks}$$

[68] As always, if we have more detailed information than that known in our simpler examples, we can refine our evaluations. The miss penalty of a cache might well require fewer cycles than a cache hit. Consider what the same calculation would derive if the miss penalty was only two cycles for instance. Likewise, in the absence of such information we should assume the pessimistic case of miss and hit being of equal cost (as we did in this case).

So we see that whilst DRAM requires at least 5 clock cycles to deliver every data item, the SRAM/DRAM cache example delivers the same data in only 3.5 clock cycles on average^[68].

What we have just confirmed is that small amounts of cache memory can speed up memory access by significant amounts. When a complex memory system employs wider memory busses, burst-mode and page-mode access, and cache together, the memory bandwidth can reach quite high levels of average performance.

Now, what about cost? Let us also make some assumptions here: We could get suitable data from researching the actual components if we wished, but some arbitrary values will do for this example.

So let us suppose that 1 Gigabyte of SRAM costs \$200 and 1 Gigabyte of DRAM costs \$50. There are now three possibilities for our 4 Gigabyte memory system, each with a different cost:

Option 1: Only Use DRAM (least expensive option)

DRAM \$50 per Gigabyte, so total cost = $4 \times \$50 = \200

So in this case we get a system where a memory access takes 5 clocks. This is the lowest cost option, but also the slowest.

Option 2: Only use SRAM (most expensive option)

SRAM is \$200 per Gigabyte, so total cost = $4 \times \$200 = \800

This option is very expensive, but we get the fastest memory accesses which means optimal memory speed.

Option 3: Use SRAM and DRAM

SRAM is \$200 per gigabyte, so cost for 512MB = \$100

DRAM is \$50 per gigabyte, so cost for 4GB = \$200

Therefore, Total cost = \$300

This option is near to the less expensive option but performs very close to the fastest. It is therefore very cost effective.

It should be clear now that cache provides a very useful compromise: we get most of the performance benefit of an optimal memory configuration,

but at a cost that is almost the same as the least expensive option. This is an example of what we mean by a strong cost-performance tradeoff.

5.5.1 Multilevel cache

Another aspect of cache that you may come across in computer systems architecture is the idea that there can actually be **multiple caches** in the system. This enhances the benefits we have already highlighted for immediate memory read times, but also helps to alleviate some of the issues relating to the von Neumann bottleneck, as we shall see.

The concept of **multi-level cache** is the most common case we might encounter. Put in simple terms, a processor may have an external bus and external DRAM memory chips, and alongside those DRAM chips there will be an external SRAM cache. However, at the same time that this is happening, the CPU chip can also have cache built into its internal circuit layout (known as **on-chip cache**). And furthermore, this cache may be split into two parts (**a dual cache**), these being **instruction cache** and **data cache**. An example microarchitecture is shown in Figure 5.6.

We can see that the von Neumann bottleneck is reduced in this machine, since both instructions and data can be fetched at the same time, provided at least one of the dual caches scores a hit. Indeed, the bottleneck is only incurred here when both caches miss simultaneously and then both have to compete for access to the main system bus and external memory. If the miss rate for each cache was 10%, then the probability of both caches missing simultaneously would only be $10\% \times 10\%$, which is only 1%.

When this example processor is used in a system where the external memory already has a cache, it would be referred to as a **two-level cache hierarchy**, where the on-chip cache represents level one and the external cache represents level two.

Some processors take this a step further, having the split instruction and data cache on chip, then also a combined cache (the **unified cache**) also on chip, and finally the main memory cache^[69].

There are several benefits of using on-chip cache:

- ▶ The on-chip cache can be optimised to work with the CPU circuitry to hide the address setup clock cycles, using overlapping activities (pipelining) to make these appear to take zero time. This means that the on-chip cache can deliver a new data item for every clock cycle.

[69] The INTEL Core i7 processor for example, has such a hierarchy. The i-cache and d-cache operate in parallel at **level-1**, and connect to the unified cache at **level-2**, both on chip. Finally, the external memory bus will likely have a cache of its own, forming **level-3 cache**.

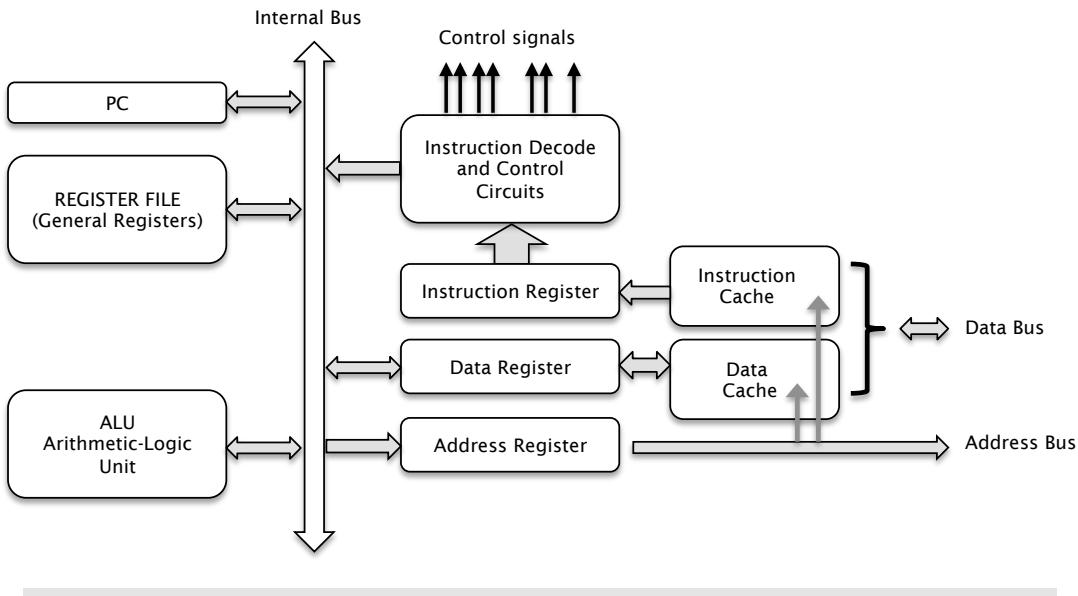


Figure 5.6: A scalar microarchitecture with dual on-chip cache hierarchy. Showing separate data and instruction caches on chip, providing independent memory channels.

- ▶ The width of the on-chip cache can be fairly arbitrary, matched to the internals of the CPU rather than external memory, and potentially organised in ways that maximise cache performance for that specific processor under specific conditions.
- ▶ The instruction cache and data cache can be accessed simultaneously, and it is rare that both caches have a miss at the same time, meaning that even if one cache misses, it can use the external memory bus whilst the other cache continues as if nothing has happened.
- ▶ The split cache gives the impression that the memory bus no longer has the von Neumann bottleneck. On-chip at least, it is much closer to a Harvard architecture.

Importantly, all of these advantages come as part of the CPU chip, and do not need radical changes to the design of the system as a whole.

Now consider our previous cache example again. Remember that we discovered that we were able to reduce memory read time from 5 clock cycles to 3.5 clock cycles. Now let us consider what happens if the on-chip cache can also achieve a 90% hit rate, and requires only one clock cycle per access:

Multilevel Cache Calculation

90% hit rate: Internal (on-chip) access:

Average main memory read time = $90\% \times 1$ clock cycle

10% miss rate: External read time:

Average main memory read time = $10\% \times 3.5$ clock cycles

Cache miss penalty = $10\% \times 1$ clock

Total average read time:

$$(90\% \times 1) + (10\% \times 3.5) + (10\% \times 1) = 1.35 \text{ clock cycles.}$$

Let us reflect upon what we have now achieved: The average memory read time in this multi-level cache system is now getting reasonably close to 1 clock cycle; in other words, very close to optimal performance.

But, there is much better news hidden in the data: Actually, this result may be duplicated for each cache, operating in parallel. So now, with the ability to operate the CPU such that it can access both instruction and data cache simultaneously, we could reasonably argue that **average access time** is actually half this amount, or 0.68 clock cycles^[70].

If we were again to assume the processor/system clock frequency is 1000 MHz, then we can also say that instruction bandwidth for the multilevel system with split cache is $1000/0.68 = 1470$ Million reads/sec, and the same for data bandwidth. Compared to the original figure of 200 Million reads/sec, from our first memory example, in Section 5.4.1, this is over seven times the performance of that original case, but with very small investment^[71].

[70] This is evident, because if we can read two values at a time with an average access time of 1.35 clocks, then the average per single value is 1.35 divided by 2, giving 0.68 clocks per access.

[71] For a 32-bit internal/external memory width assumption, this would imply nearly 6 billion bytes per second.

5.6 Cache management policies

In the simplest terms, cache is merely a much faster memory, where the most useful parts of the main memory content are kept for quick access. The definition of what is useful and how this should be determined, to exploit temporal locality, is the basis of a concept called **cache replacement policy**. Meanwhile, the way the data is organised within the cache, to enhance spatial locality, is known as the **cache mapping policy**.

5.6.1 Mapping policies

For a cache to maintain copies of selected memory locations within its faster memory, it needs to keep track of which locations it has cached,

[72] This is known as the Valid bit, and indicates if the cache entry has been read in from memory already.

[73] This is known as the Dirty bit, and indicates if data in the cache needs to be written back to memory at some point.

[74] The simple case is a fully associative cache with block size of one

[75] This would be described as a fully associative cache with block size 16.

as well as their content. In the simplest possible scheme, a cache could therefore be organised as a table of values, where the first n bits designate the whole address of the item being cached, and the remaining m bits represent the copied data. If a memory system is 32-bits wide, then m must be 32 bits. If the address space is also 32 bits, then n will also be 32 bits, and thus the total size of each cache entry would need to be $n+m=64$ bits. We also typically need at least one bit for each entry to determine if the data is actually in cache^[72], and another to indicate if the cache content is also resident in memory^[73].

This simplest case (known as a **fully associative cache**^[74]) results in a lot of circuitry and bit storage flip-flops. Since multiple adjacent locations are often within the same working set (spatial locality), then having a separate address field for every single location is very inefficient. Consequently, caches are usually organised into **cache lines**, **cache blocks**, and **cache sets**, in an attempt to optimise hardware cost and performance.

In cache terminology, a **tag** or **tag field** is simply an address or more typically a portion of an address. A **line** is simply a tag and one or more data words related to that address or **locality**. If a line contains more than one data word then those words represent a block. A **block** might be 4, 8, 16, 32, or any useful group of consecutive data items relating to a 2^n numerical word count.

Thus, to improve our simple case, we might decide that each address relates to a block of sixteen words, or memory locations. We would then have sixteen words per address field, resulting in the address **tag** shrinking by 4 bits and becoming a **partial address**. In hardware terms, we end up with a line consisting of sixteen 32-bit words plus a single shared address tag of 28 bits, a total of 540 bits, or about 34 bits per word^[75], requiring much fewer storage bits, and much less circuitry and significant reductions in power consumption. Adding a valid and dirty bit for each word would take the total number of bits to 572 and an average of around 36 bits per word, which still compares very favourably to the 64 bits per word in the simplest case given earlier.

Now what about **sets**? The idea of a **set** is that a fixed part of the address range has a set of lines associated with it. If we designed our cache to have 16 sets, with a single line in each set, then the already smaller tag field can again be reduced by a further four bits (since $2^4=16$). By doing this we would end up with the **directed mapped cache** scheme shown in Figure 5.7(a). Here there are 16 sets and each set has a size of one line. Note that if we had the opposite (only one set but lots of lines

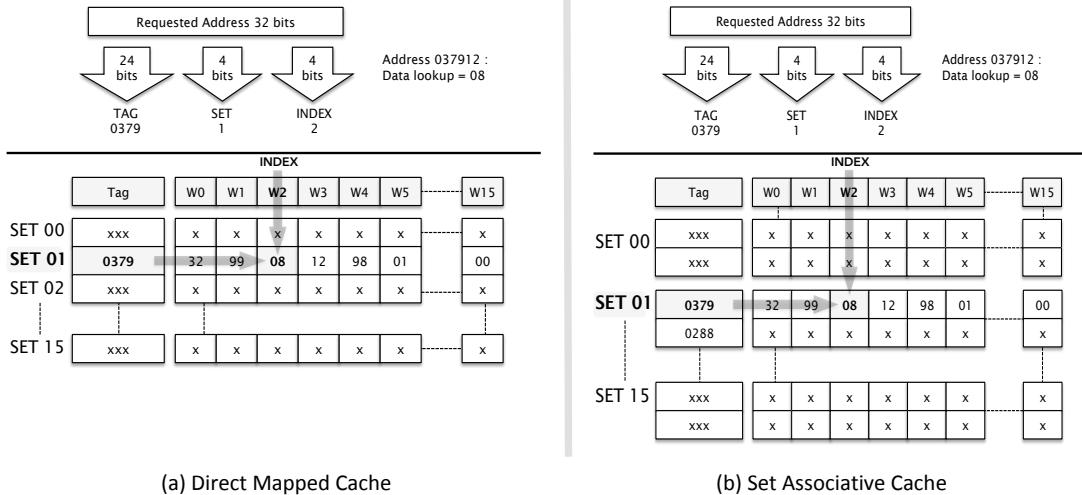


Figure 5.7: Cache Mapping Schemes. Item (a) shows a **Direct Mapped** cache, which has multiple sets and single lines per set. Item (b) shows a **Set Associative** cache, which has multiple sets and multiple lines per set. Compare these cases to a fully associative cache, where there is only one set but multiple lines.

in that set), then we have ended up back with a fully-associative cache once again.

From this point it is easy to extend the concept of direct mapped cache to **set associative cache**. Actually, directed mapped cache is simply a special case of set associative cache where the set size is one line per set. So adding **multiple lines to each set** gives us set-associative cache mapping as shown in Figure 5.7(b). This final scheme has some advantages beyond simply saving on cache storage hardware cost, but this is beyond the scope of this introductory text. Advanced readers might wish to read further on the subject however.

5.6.2 Replacement policies

A cache **replacement policy** is an algorithm used by a cache controller circuit to decide what should be in the cache and what should not. Remember that cache is typically small compared to main memory, so it can only hold a subset of the whole memory content. For cache to be successful at speeding up memory access and memory bandwidth, it needs to make sure it contains the most useful content. As a consequence, there will come a time when, to bring something new into cache, some

existing content must be discarded to make way. Out with the old, in with the new. Deciding what should be replaced is the basis of **cache replacement policies**.

There are a number of policies that can be used. These are worth reviewing briefly, if not in advanced detail.

Least Recently Used (LRU) replacement policy: In this case, the data item in the cache that was least recently used (not necessarily the oldest item) is replaced by the new item. The cache must keep track of when items were added to its content, to support this policy, which exploits temporal locality.

Least Frequently Used (LFU) replacement policy: In this case, the data item in the cache that was least often used is replaced by the new item. The cache must keep a tally of the use of each item in cache in order to support this policy.

First In First Out (FIFO): replaces the oldest item in the cache with the new item regardless of how often that item has been used. Circuits are required to track the age of the items in the cache.

Random Replacement Policy: In this case, the data item in the cache to be replaced is chosen at random. This has the advantage that there is no need for circuits to keep track of the use of items in the cache.

There are many other algorithms, many of which are variations on those just mentioned. These are rather specialised and need more advanced knowledge of computer architecture to fully appreciate. Also, it should be noted that when we talk about replacing an item, this may well be a single word, or a line, or even a set, depending upon the cache design and replacement policy.

5.6.3 Write-back policies

There is one further important consideration for cache memory, which relates particularly to writing data to memory rather than reading data from it. This is the concept of cache write-back policy. This is also important when understanding cache design. There are two significant variations of this concept:

Delayed write-back: This will allow the cache to capture any memory writes generated by the CPU, without involving main memory. Repeated writes will go to the cache, which is much faster. Only when an item

is finally discarded from the cache, during a replacement action, will its latest new value be written back to main memory.

In computer architecture, the idea of a **dead store**, an item written to memory and then overwritten again before it is ever read, indicates that those memory writes are redundant. Delayed write-back captures this optimisation very well.

However, this policy also means that discarding a cache item necessitates a main memory write operation (but nowhere near as many repeated writes as are being directed to the cache for that same memory location). This potentially has a speed penalty, as well as advantages.

Cache Write-Through policy: In contrast, a **write-through** policy ensures that every write to a location held in cache is also written at the same time to main memory. This means that every cache write has a main memory speed penalty associated with it.

This model is less efficient, but it does ensure that main memory content is always as up-to-date as possible. Nothing can reside in cache that is not also in main memory.

You may wonder why there are two policies. Delayed write-back seems more efficient than write-through, so why not just use that? The answer is to do with a concept known as **memory coherency**. When the memory content is identical to the content the processor sees, when 'viewing memory through the cache' so to speak, we say that the memory is **fully coherent** (or in sync with cache content if you like). This is always the case with write-through cache, since the memory is always updated by a write that also updates the cache. On the other hand, write-back cache does not immediately update main memory when it receives a write, and therefore the memory coherence in such a system will not be 100% all of the time.

Does memory coherence matter? This very much depends upon the system. Often the answer is no. However, where a system has a number of devices connected to the system bus, and these devices have access to parts of the main memory, then the fact that they may be looking at out-of-date memory content might well be important.

Likewise, if a device in the system changes the content of main memory, and the cache is not aware of this, then it could keep providing an out of date value to the CPU when it performs a read (known as a stale cache item). This is one aspect of the related problem of cache coherency.

Although it is beyond the scope of this text to get into more detailed discussions of replacement and write-back policies, it is clear that they affect the way the CPU sees the content of memory, and for some applications and some systems designs, this could be a problem if not taken into account.

These hierarchical internal cache schemes, and the choice of replacement and write-back policy, will dictate performance for a system, and can sometimes be chosen specifically for certain kinds of computational workload. A weather simulation is very different from a video editing package, even though both may deal with large volumes of data. Where a system is built with a specific application in mind, the cache and memory architecture can be customised to maximise performance based upon knowledge of how the system behaves in terms of patterns of code and data usage^[76].

[76] The interested reader can read further on this topic: in particular the concept of associative cache policies.

5.7 Avoiding errors

Whilst most memory systems are utilised without **error detection** and **error correction** in mind, there are some memory systems where this is a major consideration. A bit error in a large memory system could be caused by a number of things, including faulty transistors in the memory chips themselves, potentially damaged due to voltage surges, or excessive heat due to insufficient cooling, momentary electrical interference, or even radiation. There are situations where this cannot be tolerated, in particular in safety-critical systems as mentioned earlier.

There is thus sometimes a need for memory systems that detect and even correct errors. One mechanism is to use a specialised memory controller, and another is to use a slightly modified version of RAM known as **ECC memory**. ECC is an abbreviation of Error Correcting Code, and there are potentially several algorithms that are used to implement ECC capabilities, typically derived from a mathematical technique known as Hamming codes. Simpler techniques include parity checking, a topic which we will look at further in Section 14.7.

Power-on self test is another mechanism that helps to reduce memory errors. When a computer turns on, it may perform a startup checklist, including writing test data patterns to every memory location in its physical memory. If the correct response does not occur when reading back the data it has written, then it knows that this location of memory is damaged (or potentially the connections to the chip). A lot of systems

will, at this point, just stop with an error message, and request that you resolve the problem. In some cases, however, it may be possible for the operating system or BIOS to record the location of faulty memory and mark this as unusable during system operation. This is particularly common where flash memory is used.

5.8 Programming considerations

We have focused up to this point on the hardware aspects of memory systems. At this point it is worth introducing some additional points, particularly relating to memory architecture in the programming domain.

The concept of a **memory map** is often used to represent the structure of the memory as it may be accessed by the programmer. For example, a 16-bit ranged memory map may show some of the following aspects:

- 0000-07FFh** : BIOS ROM
- 1000-37FFh** : FLASH RAM
- 4000-7FFFh** : RAM
- 8000-EFFFh** : UNUSED
- F000-FFFFh** : IO Mapping

This can be translated into a diagram as shown in Figure 5.8. Most of these items should make sense as blocks of memory. However, the final block of memory (**IO Mapping**) is unusual. What this memory map suggests is that **IO devices** are mapped into memory addresses. This **memory mapped IO** makes some sense if we remember that IO devices (or at least their interface circuits) are often connected to the system bus alongside memory and CPU and are therefore potentially visible to the CPU as if they are memory spaces. In practice, what happens is that a given IO device will occupy a small number of uniquely allocated locations, relating to individual control addresses within that IO device. For example, a keyboard and mouse might have the following address designations:

- F000h** : Main Key Value
- F001h** : Status of Caps Lock key (on/off)
- F002h** : Caps Lock Light Control (on/off)
- - -
- F020h** : Left Button Status
- F021h** : Right Button Status
- F022h** : Motion Register

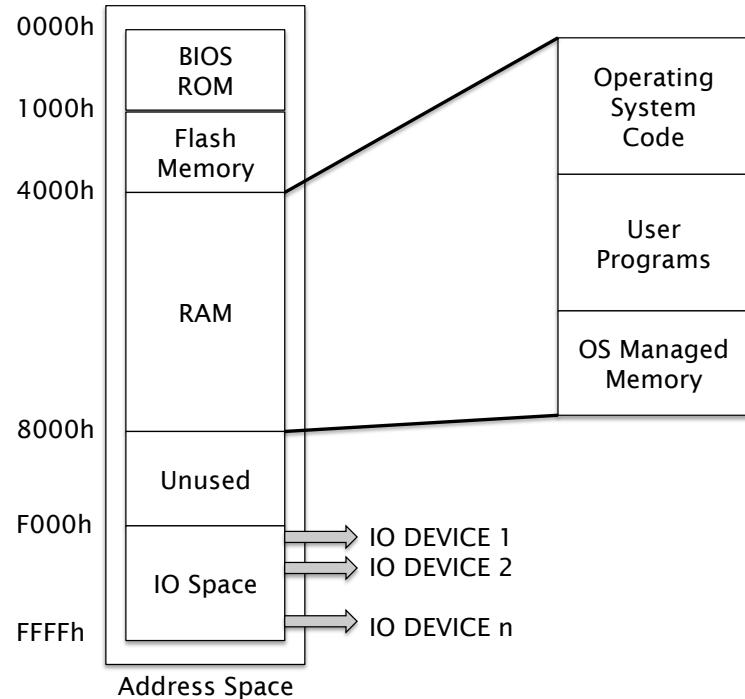


Figure 5.8: Memory Map Example. Showing BIOS in low memory, flash memory for reconfigurable firmware, RAM, and IO device mapping space.

[77] In practice, mapping an IO device with 'n' IO registers to a very specific set of n corresponding memory addresses would require lots of logic circuitry, with little benefit. Consequently it is normal to map, for instance, eight IO registers to a larger memory block. Thus, our keyboard example might repeat the IO register addresses across the range F000h to F01Fh, and the mouse register addresses might occupy F020h to F03FH. Registers simply appear to repeat within this block. For the programmer, there is no disadvantage.

These are typically known as **IO addressable registers**. Each IO device will have a set of such register addresses, relating to internal functions and data. The assignment of IO addresses is normally controlled by the operating system in a modern computer, either via the low-level (BIOS) or the installed OS^[77].

One important thing that we might note is that if IO registers look like memory addresses, then they could, in theory, end up in cache. In the case of our keyboard, if we keep reading a key value from cache, we will not see new key-presses (only the first one to go into cache). This is a case where memory content should not be cached.

Another scenario is where two processors share access to the same memory area (a shared memory system). Clearly, if either processor is caching the content of the shared memory block, it will be unaware of any changes made by the other processor, which defeats the point of having shared memory. This cache coherency issue is a well known problem.

To combat some of the problems just highlighted, some operating systems permit areas of memory to be designated as **non-cacheable**, and

configure the memory controller and/or cache controller appropriately.

Many processors also support the idea of IO having a separate address space (**Direct IO Addressing**), which allows it to exist in a memory space outside of the scope of the caching hierarchy. Memory maps can be very useful for illustrating these kinds of systems and making such problems and solutions visible in the design process, and in the system documentation.

Finally, memory maps also provide a useful way to visualise how memory is utilised by the software and operating system components. In Figure 5.8 for example, we see that the RAM is organised by the operating system into different functionalities.

5.9 The future of memory devices

Memory devices have progressed almost incomprehensibly over the past 40 years. The first memory chips, manufactured in the 1970s, had capacities measured in kilobytes. Modern memory chips have capacities which are vastly superior. This advance has been driven primarily by the advances in silicon chip manufacture. Early chips had component dimensions (**feature size**) measured in **microns** (millionths of a metre), whereas today's technologies have components measured in the **nanometre** (billionths of a metre) scale.

Even in the past ten years, chip technology has progressed from 90nm down to 10nm, an almost ten-fold reduction in feature size, and potentially a 100-fold reduction in silicon area per bit-cell. This translates to memory chip capacities increasing by over 80 times in a decade. As of 2019, 16 Gigabytes on a single DRAM chip is considered to be fairly advanced, especially if combined with high-speed data access techniques. Figure 5.9 shows recent memory capacity trends over recent years.

It would be fortuitous if feature size continued to shrink every year, far into the future, resulting in ever increasing memory chip capacities. Unfortunately, **VLSI**^[78] technology is hitting a number of roadblocks at the present time. It was once considered doubtful that circuits with feature sizes of 10nm were capable of being reliably manufactured, yet this has been achieved, and it is not inconceivable that this may continue a little further. However, there are basic laws of physics that prevent traditional electronics circuits from operating properly at ever smaller sizes. We are rapidly reaching the point where transistor devices are so small that effects of individual atoms and electrons become significant enough to

[78] VLSI: Very Large Scale Integration, representing the current chip fabrication era.

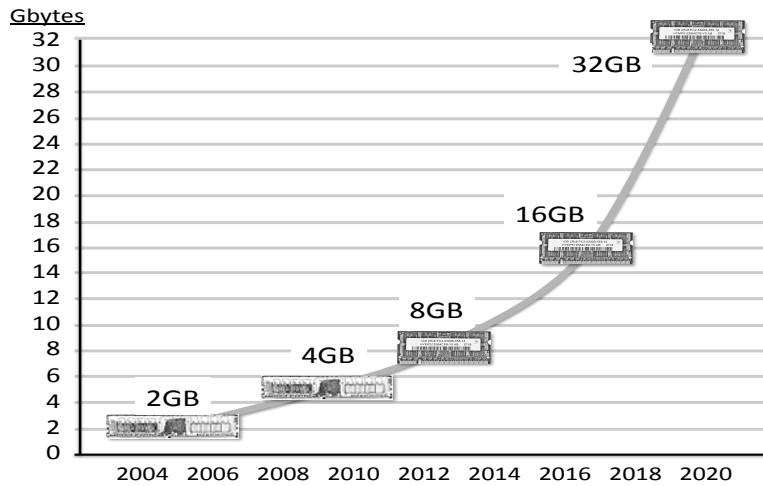


Figure 5.9: DRAM memory module capacity trends. Showing plug-in memory module growth in storage capacity over the past 15 years.

spoil the party. It is notable that memory chip capacities were doubling roughly every two or so years, but more recently this has slowed, as these effects become harder to design around.

A further problem is the concept of **yield**. When chips are manufactured, some of them are faulty. With one billion transistors on a chip, there is a small probability that at least one of these might be substandard or just non-functional. This could mean the whole chip is useless and has to be discarded^[79]. The remaining chips, with no faults, are the ones that can be sold. If 70% of chips in a batch are good, then the yield is 70%. This impacts upon cost, since the manufacturer still has to make 100 chips in order to get 70 that work. It then follows that every time the number of transistors on a chip increases, the risk of a fault increases too (and thus yield drops, then costs potentially go up). This problem only gets worse as we head down toward ever smaller feature sizes.

[79] There are a few tricks that can be played to reduce this impact. For example, if a 4 Gbyte DRAM chip is manufactured with four identical blocks of 1 Gigabyte inside its chip layout, and one block is found to be faulty, then that chip might be reconfigured to disable two blocks (including the faulty one) and then be sold as a 2Gbyte chip. This helps to keep costs lower by making effective yield better across several product lines.

To bypass the worst of the yield problem, and side-step the physics of very small components, manufacturers have recently taken a slightly different route. Rather than making a RAM chip that is twice as complex, they take two chips manufactured with existing technologies and stack them one on top of the other (di stacking), using specialised micro-manufacturing techniques. We can quite happily stack four or perhaps even eight chips into a stack, and as a result, the size of the chip does not increase much (since most of the physical chip size is related to the external package and not the small piece of silicon inside).

A limiting factor here is power and heat. As more chips are stacked and

packed into a small chip package, they consume more power in a smaller volume (the **power density** increases). This in turn leads to higher heat concentration within the package (thermal density).

Stacking also makes it harder for heat to escape from the silicon chips. If chips get too hot then they become unreliable, and ultimately get damaged. It is not unusual to see high performance memory modules having their own heatsinks to help cool the modules down when running at peak activity. Ultimately, there is a point where it is not possible to pack more chips into a given module, unless feature size, power, and heat are all reduced further. This barrier to progress is sometimes referred to as the **power wall**.

Another area of interest, apart from data capacity, is of course speed. By organising internal memory structures in the internal memory chip design, there are ways to accelerate the access of data within the chip. In terms of external data transfer from chip to processor, there are also new methods of speeding up data transfer, by altering bus protocols, and methods of clocking data from one device to another.

Various **DDR (Double Data Rate)** schemes already exist; for example, combining dual clock-edge data transfer and burst mode data transfers (for example **DDR2** and **DDR3**), and these have helped to push data transfer speeds across busses to high levels.

In the more distant future, new electronics and physics discoveries will help to provide yet more memory advancement. The recent discovery and implementation of the **memristor**, a new kind of electronic component, has promised to allow new innovations in memory technology. It may take a decade for these leading-edge technology concepts to mature to mainstream products, so we shall see how things develop.

5.10 Summary

Memory is a fundamental and extremely important component of the working computer system. In this chapter we have found that rather than memory being a single entity, it is actually an entire family of related but differing component technologies, each with specialist capabilities.

A particular observation made in this chapter relates to performance: memory is imperfect and creates delays. Delays make processors wait for data when they should be working. Therefore, memory is a limiting factor on both processor and system performance.

Finally, we discovered that to overcome these limitations, it is necessary to make carefully considered compromises in memory system design, balancing cost against ideal performance. As with most solutions in computer architectures, introducing a solution, such as cache for example, brings new problems. The designer must be aware of these, at the hardware level, software level, user programming level, and also appreciate the consequences for the operating system too.

5.11 Terminology introduced in this chapter

Burst-Mode cache coherency	Cache hit
Cache line	Cache memory
Cache miss	Cache replacement policy
Cache write-back	Cache Write-Through
Capacitive bit cell	Column
Column address	Computer memory
Cycle time	Data bandwidth
Data cache	DDR Double-Data Rate
DIMM	DRAM
Dynamic Ram	EAROM
ECC memory	EEROM
Effective memory bandwidth	EPROM
Erasable Memory	Error correction
Error detection	Fast-Page-Mode
Feature size	Flash Drive
Flash Memory	Heatsink
Hit-rate	Instruction bandwidth
Instruction cache	Internal readout
Least Frequently used replacement	Least recently Used replacement
Linear sequence	Magnetic core-store
Memory bandwidth	Memory bus width
Memory capacity	Memory coherency
Memory data rate	Memory module
Memory stick	Memristor
Mercury delay line	Micron multilevel cache nanometre
Non-linear sequence	Non-programmable memory
Non-volatile	One-Time Programmable
Physical memory	Power density
Power wall	Power-on self-test
PROM	Random Replacement
Read-Only memory	Refresh cycle

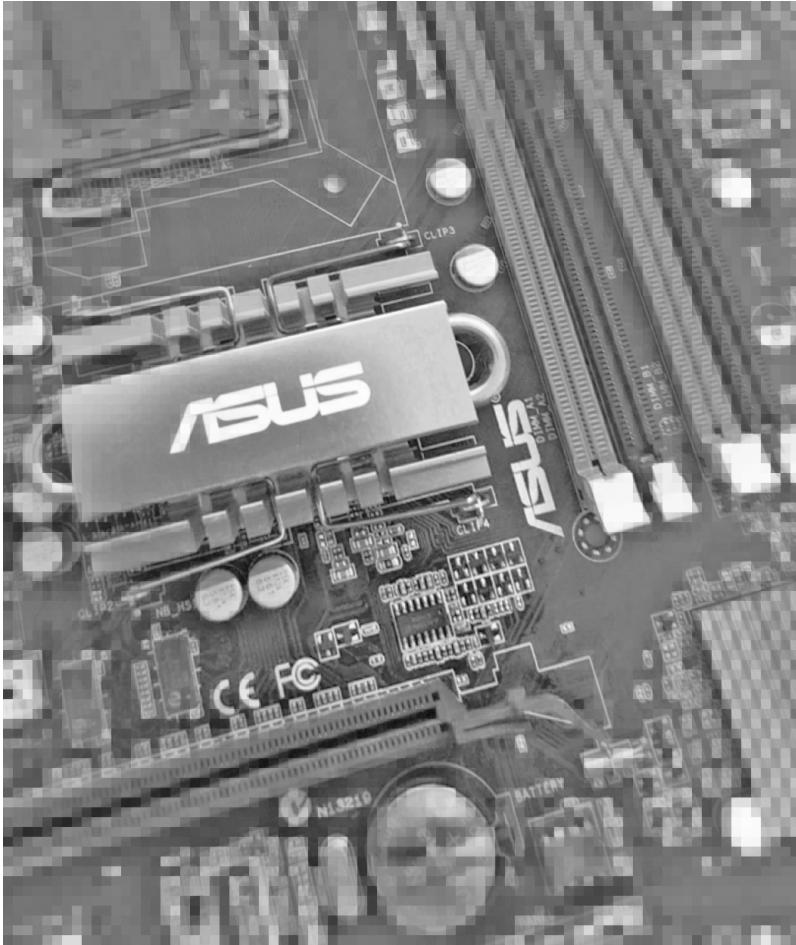
List continued overleaf ...

ROM	Row
Row address	Safety-critical system
SIMM	SRAM
Stale cache item	Static Ram
Storage density	Storage device
System bandwidth	Thermal density
Unified cache	Virtual memory
Volatile	Working set
Yield (chip)	

These terms are defined in the glossary, Appendix A.1.

6

Building The System



Portion of an ASUS (AsusTek Computer Inc) Motherboard.

6.1 Putting it all together	102
6.2 The system bus vs the dedicated bus	104
Concurrency in bus architecture	106
6.3 Bus standards	107
6.4 A generic system bus	108
6.5 PCI: a very successful Bus Standard	110
6.6 Data rate matching and buffers	113
6.7 USB: Universal Serial Bus	113
Plug-and-Play	116
Hubs and ports	116
6.8 More common bus standards	116
6.9 Industrial and embedded standards	117
I2C (Inter-Integrated Chip) bus standard	118
CAN bus	120
6.10 Rack-mount, hot-swap, and servers	121
6.11 IO device mapping and IO servicing	122
6.12 Summary	123
6.13 Terminology introduced in this chapter	125

6.1 Putting it all together

We have already covered quite a large number of concepts in computer architecture, and introduced some key terminology. However, looking back on the past few chapters, we have taken a system level view of the way computer systems work. Block diagrams do not tell the whole story. On the other hand, this is not a digital electronics text in the engineering sense, and it is not aimed at advanced computer science or computer engineering students.

In order to understand how things connect, link, and build, into a complete computer system, we will now explore things in more detail but try to avoid too much electronics, and too much physics. The place to start with this task is perhaps most appropriately the **motherboard**.

In a modern computer system, certainly a general purpose one, there will likely be a self-contained circuit board: a **mainboard** or **motherboard** as it is sometimes known. This motherboard has all of the circuit connections, chip-sockets, busses, and miscellaneous additional chips^[81] to allow a complete high performance computer system to be built, simply by plugging in and/or connecting the appropriate modules and chips to configure the system. Figure 6.1 shows a typical motherboard with some of the most important connections labelled.

[81] The term often used for various minor chips and components needed to tie things together is 'glue logic', in computer parlance.

[82] These are known as printed circuit board tracks, and being metal, they have capacitive and resistive properties. When capacitance and resistance are combined, they create a signal attenuating effect.

[83] Early motherboards could easily be larger than 300 x 300mm, though in recent years these have shrunk significantly as far as desktop style PC's are concerned: A micro-ATX board may be about 240 x 240mm, and nano-ITX boards are about 120 x 120mm

These circuit boards are designed with some very demanding constraints, often relating to the way in which electronic signals behave in wires. On a circuit board, wires are formed by metal tracks built onto or into the material making up the board^[82]. As we decided not to get too far into physics, it is enough to say that the length of these wires can cause high frequency signals to travel through them quite poorly. A very short wire will begin to lose signals only at very very high frequencies (let us say of the order of 10GHz for the sake of argument). As wires get longer, the frequencies that are **attenuated** get lower. Starting at perhaps 8GHz for extremely short wires, then 5GHz, and then when wires are just a bit longer than that we find that 3GHz or 4GHz frequencies are affected. Now, this is where we start to encounter design issues, because many modern CPU's operate in this frequency range. So a processor running at 3.8GHz will find it very difficult to send signals at those speeds across a wire track on a motherboard, unless that wire is very short.

But motherboards are actually quite big^[83], so the location of the chips, the CPU, the memory DIMMs and SIMMS, and other devices must be carefully considered. Even so, it may prove impossible to make motherboard wires operate at frequencies of 5-10GHz.

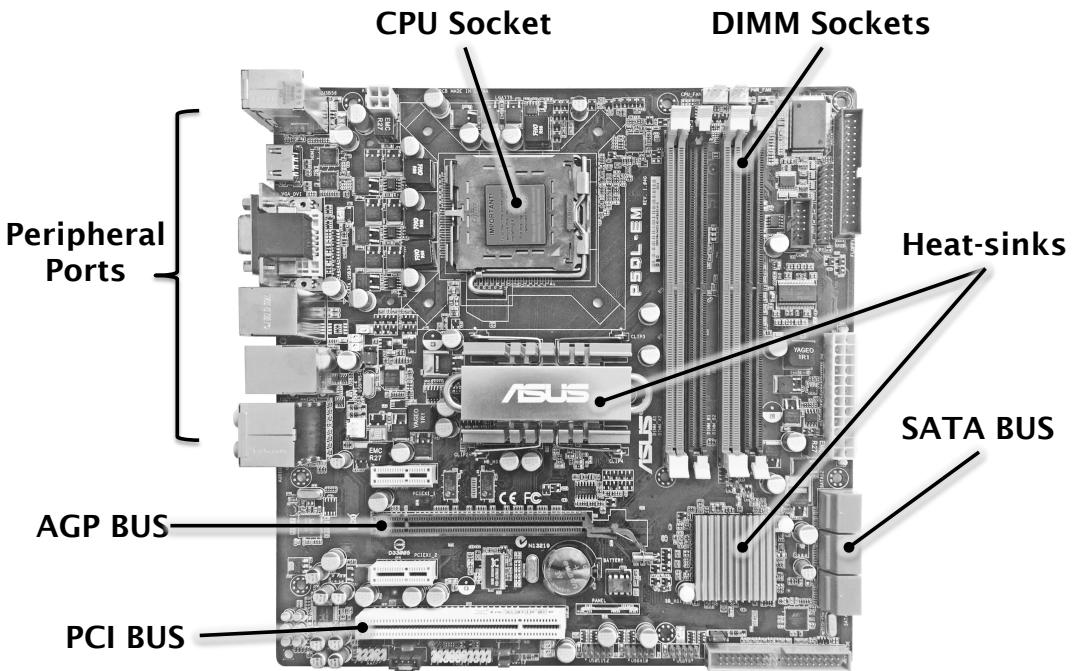


Figure 6.1: Motherboard Example. Showing peripheral ports (top left), DIMM memory sockets (top right), CPU socket (top centre), PCI BUS (bottom left), AGP BUS (centre left), SATA busses (bottom right), and examples of heat-sinks (centre right).

In order to overcome the limitations of circuit board design we have just mentioned, there is an answer of sorts. We can simply send signals at lower frequencies, and accept that this means data will be transferred more slowly. Whereas a system can certainly be designed to rely mainly upon a system bus, operating upon the same frequency as the processor, this would force the whole system to run at the lowest device speed in the system. Instead, many modern bus systems operate at their own local bus frequencies (that is, at a different speed to that of the processor), and many of these busses also operate according to individually selected protocols, known as **bus standards**. As a result of this fundamental design compromise, the idea of a bus has evolved far beyond a simple group of wires connecting parts together, with a single simple clock regime, as we first envisaged in our rudimentary system block diagram (Figure 2.1 of Chapter 2). We will now explore this idea further.

6.2 The system bus vs the dedicated bus

As we have already mentioned in earlier chapters, the system bus is, in a simplistic view, just a collection of signal wires capable of connecting various basic system components together and allowing them to exchange information. At the very least, we would expect CPU and memory to communicate via this bus, and also some form of Input/Output device interface to be visible on the same bus (the basic von Neumann Model). This system bus is also sometimes referred to as the **host bus** (where the CPU is the host), but also somewhat less intuitively the **local bus** and the **front-side bus**, a **memory bus**, or the **main bus**.

It is also useful to remind ourselves that the definition of a bus isn't just a group of wires connecting devices together, but connecting **multiple devices** together. A bus allows multiple devices to engage in data transfers. In the simplest mode, the data transfer is between any two devices connected to that bus, but there are scenarios where multiple devices can receive data from one or more other devices sending that data (known as a **broadcast or multicast**).

In both of these scenarios there is the concept of a **bus master** and a **bus slave** device (or devices). The master is in control of the data transfer, and decides when it begins^[84], what is transferred, and when to end the process. The start, middle, and end of the data transfer process may be referred to as a **bus transaction**.

Apart from the system bus, there may be **secondary busses** in our computer system. These can connect to the system bus via an interface chip (known as a **bus bridge** or **auxiliary bus controller**), or even connect directly to the CPU using a **dedicated bus** in its own right. In a very high performance system, this latter case has a number of advantages. We shall see examples later.

We might now envisage a block diagram of a system, evolved from our first computer architecture in Chapter 2, but advanced enough to represent the main features we have just discussed. One possible case is shown in Figure 6.2.

Figure 6.2 is only one possible combination of busses in a computer system or, in other words, one particular **bus hierarchy**. This is a term which reflects the fact that the bus architecture has a tiered structure, such that the CPU has a host bus or system bus at the top level, and there are auxiliary busses that are subordinate to that. We could have

[84] Although the master may initiate a transfer, it may sometimes be doing so in response to a request from a slave device. Therefore, it may control the transfer but not the whole chain of events.

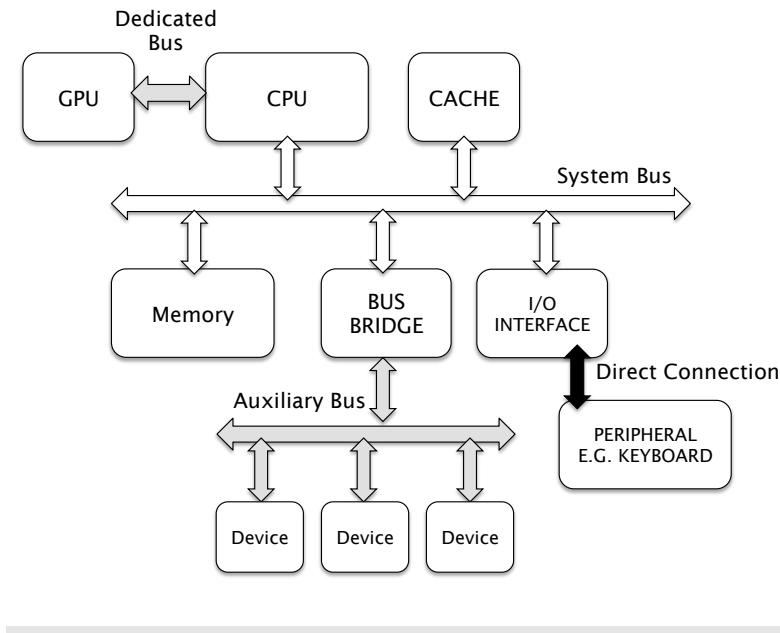


Figure 6.2: System diagram with more comprehensive bus architecture. System includes a bus bridge hosted on the main bus, extending an auxiliary bus to host three additional IO devices.

further levels of busses connected in even lower tiers if we wish to do so.

In this example we can see several important aspects of the system that need explanation:

First of all, the system bus permits several components to connect to the CPU, including a standard IO interface, which then connects to a keyboard by a direct connection in this example. If this direct connection can only connect **point to point**, then it is not a bus in the standard definition. We may well have a bus in which only two devices exist because that is all that we have added to the system, but that is not classically a direct point-to-point connection either.

Also connected to the system bus is a component we have already mentioned, the bus bridge. We may be more specific about this component if we know which kind of bus it supports: a **PCI bridge** for example, supports a **PCI bus**. The bus bridge creates a connection that the CPU can recognise on its host bus, but which also allows the CPU to communicate with devices on the other side of the bridge (those on the auxiliary bus). This can be the same type of bus, but is possible even if they use a different interface standard and bus protocol. A bus bridge is often just a particular chip on the motherboard. The advantage of this is that a CPU need not provide every possible bus connectivity that might be

selected by a designer; instead the system designer simply supplies the appropriate bridge component.

A further observation is that devices on the auxiliary bus can communicate with each other without necessarily involving the CPU, but on occasions can also communicate with the CPU via the bridge. Therefore, individual busses can potentially operate independently of each other. A bridge permits both compartmentalisation and cooperation.

The CPU in this example also has a dedicated bus connecting to the GPU component. This is a highly specialised bus connection, going directly between the GPU and the CPU (not via the system bus). This allows it to operate continuously without interfering with the system bus bandwidth, and without system bus transactions interfering with its own behaviour. In this case, the device connected is a GPU and dedicated busses are often used to permit high performance GPU communication to the CPU (among a number of other scenarios).

6.2.1 Concurrency in bus architecture

Looking at our example system, there are clearly multiple buses, as well as other kinds of connections. This results in some quite important advantages, and in particular the concept of **concurrency** becomes apparent.

It should be possible to see that in our example system, the system bus can operate any one transaction at a time between a set of possibilities including CPU and memory, cache and memory, CPU and the IO interface, and so on. However, whilst this is happening, the auxiliary bus can also be performing transactions between its local devices too.

In other words, the host bus and the auxiliary bus can run **concurrently**^[85], provided they do not want to cooperate in a data transfer between their two buses. Equally, whilst these two busses are operating concurrently, the dedicated bus can also be operating. These three busses may not interfere with each other at all during these operations and therefore will not hinder each other's performance.

With concurrency, the system has effectively been subdivided, or partitioned into more or less independent parts that can work simultaneously on different things. The only exception to this is if the CPU decided to communicate with an auxiliary device (or vice versa), via the bus bridge, in which case both busses would be involved in the transaction and neither bus could be used by any other devices at that time.

[85] The term 'concurrently' means several entities operating (e.g. transferring data) at the same time.

This is an important observation, because being able to do three things at once means more work done in less time. It can also be measured in another way:

Suppose the system bus has a data transfer bandwidth of 200 Megabytes/sec, the auxiliary bus has a data transfer bandwidth of 500 Megabytes/sec, and the dedicated bus has a data transfer bandwidth of 1 Gigabyte/sec.

Then we can determine that the system has an effective transfer bandwidth of 1.7 Gigabytes/sec. This is because we can potentially add all of the independent bus capacities together (in the best case scenario), giving us $1000 + 200 + 500 = 1700$ Megabytes/sec.

If we wished to boost performance further, we could add multiple bus bridges to the host bus and have other independent groups of auxiliary bus devices, each adding a further 500 Megabytes/sec of system bandwidth.

But of course, this optimistic evaluation requires all available busses to be used in near optimal fashion simultaneously (perhaps a rare occurrence). Therefore, when a system has heavy work to do in particular categories of activity, it can be made more efficient by using a concurrent bus hierarchy, provided that the right kind of busses and devices exist, and their behaviours are compatible with efficient **bus utilisation**.

6.3 Bus standards

The development of **Bus standards** has several important purposes. First of all, a well defined standard, maintained by a recognised organisation, provides precise electrical and functional operating principles for that bus, allowing any manufacturer to develop a device that is **bus-compatible** with that standard. This ensures technology is interchangeable, and where possible, open to competition^[86].

There are a large number of standards, some of which are less frequently used, and considered semi-obsolete. We refer to these semi-obsolete standards as **legacy bus standards**, and computer systems either do not support them any more, or provide support primarily for convenience to allow older technologies to continue to be used for a period of time. A good example of this is the **VESA** bus standard, which was largely replaced by the **PCI** bus in the 1990s.

We cannot cover all bus standards in detail. However, covering a few will give us enough understanding of how busses work such that given

[86] The vast majority of standards are defined to allow open interchangeability of products. However, a few standards are designed specifically to provide protected intellectual property, via patents for example, or simply for the purposes of branding. These are known as proprietary standards and can be tightly controlled by the owner.

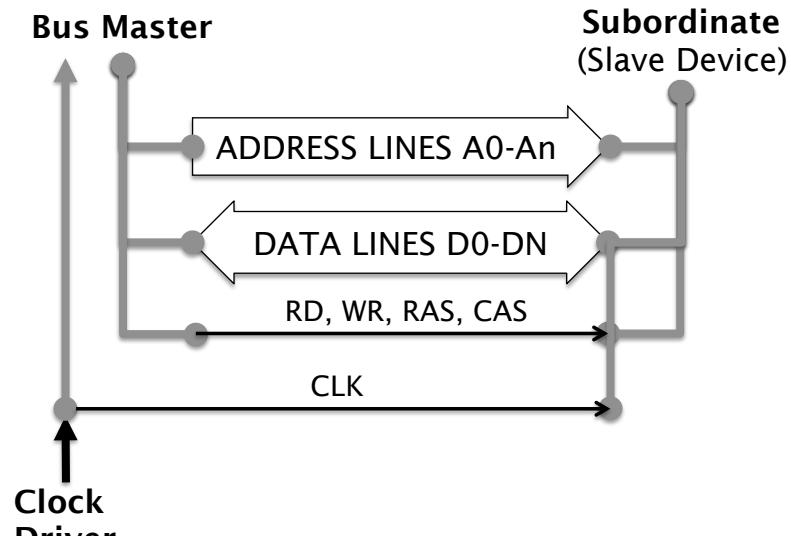


Figure 6.3: A simple hypothetical bus Example. Showing Address lines, data lines, and several control lines, plus the clock.

the documentation for another bus standard we have not met before, we should be able to grasp the basic concepts of the specification. Therefore, we will start with a generic case: the system bus as a point of comparison, and move on from that foundation.

6.4 A generic system bus

By generic, we mean in this case, a very standard example of a system bus system without any specialisations. The features of this example will be found in almost any system bus in a modern computer system. An illustration of this case is given in Figure 6.3.

This diagram represents the origin, destination, and the flow of information in the bus, which you may remember is simply a group of wires linking multiple devices together.

Looking at Figure 6.3, we can see that there is a special component, the **clock driver**, which we have not yet mentioned. This is the circuit that generates the regular on/off signal pulse that represents the **system clock**^[87].

All devices in the system use this same clock, even the master device. The clock signal acts as a synchronisation mechanism and determines

[87] The accuracy of a system clock is important, and the precision of the clock signal is guaranteed by the use of a crystal-oscillator component precisely shaped and trimmed to meet the specification that permits it to oscillate at a precise frequency.

when each event begins and ends on the bus.

Next we have **address lines**. These are a group of wires that represent the unique numerical address of a memory location with which the master wishes to interact. In a system with 10 address lines, these would normally be named **A0** through to **A9**. The number of address lines is related to the addressable memory of the system.

Because addresses are binary, the relationship is determined by the base-2 number system, such that given a number of address lines, we can determine the address range and number of locations as follows:

Definition 6.4.1 Available address range

Where $a = \text{number of address lines}$:

then there are

2^a addressable locations

For example, if $a = 10$, then there are $2^{10} = 1024$ locations.

We can easily see that to address 1 Megabyte of memory, where each byte is in its own location, then we have to have 20 address lines. Computer systems with large address spaces can easily have 32 address lines, and billions of addressable locations.

Similarly, the data lines, **D0-Dn**, represent the size of data values that can be transferred across the bus. This might be a byte, in which case we would expect to see data lines **D0** to **D7** being represented. However, many computer systems have 16, 32, or perhaps even 64 data lines.

We can see already that a computer system with 32 address lines and 64 data lines has accumulated a requirement for 96 bus signal lines, and these are implemented as metal tracks on a circuit board. This reinforces what we said earlier, relating to the difficulties of designing motherboards to run at high speeds. There are alternatives that can reduce this problem however.

There are a few additional signals present in our diagram. **RAS** and **CAS** relate to **Row-Address-Select**, and **Column-Address-Select** signals. These are required by memory chips to allow them to be given the row and column information required to access individual memory locations. Some processors generate these signals directly, whilst others make use of an additional chip called a **memory controller**.

Because memory can be addressed in rows and columns, we can potentially reduce the number of address lines. If, for example, memory is

arranged into a 16-bit row, and a 16-bit column address scheme, then we can send the two parts one after another (i.e. row then column) and then the address lines require only 16 lines in total. This is known as **multiplexed addressing**, because we multiplex (i.e. switch) between row and column address content on those lines. Of course this also means that it now takes two clock cycles to send a complete address, instead of only one cycle without multiplexing, so this reduction in address lines does not come for free.

Two further important signals are also needed: **RD (read)** and **WR (write)**. These signals are used to control which direction the data is transferred: if the master sends data to the slave it is a **write transaction** and the WR signal is enabled. On the other hand, if the master receives data from the slave, it is performing a **read transaction** and RD is enabled instead. So RD and WR indicate if the slave device is being read from or written to.

It follows that any device wishing to act as a bus master must be capable of generating all of the required signals on the system bus (except for the clock). Devices which cannot do this can only operate as slave devices (most memory chips fall into this category as they never act as bus master).

6.5 PCI: a very successful Bus Standard

We will now look at a more complex bus standard, an example of an auxiliary bus architecture, and one in very common use in modern computer systems: The **PCI** Bus Standard. Some of the key features of the original PCI bus standard are as follows:

- ▶ 33 MHz Bus Frequency,
- ▶ 32 bit Data Width,
- ▶ 32 bit Address Range,
- ▶ Bus Arbitration requires 5 clock cycles,
- ▶ Bus Turnaround requires 1 clock cycle,

The PCI bus employs bus multiplexing, but not in the same way as we encountered earlier for DRAM addressing. Instead of subdividing the address into two parts, the PCI bus uses a single bus to transfer address and data values. This is known as the **address-data bus**. This way, 32 address lines and 32 data lines can actually be the same wires, just at different times. Due to the way the PCI bus works, address and data do

not need to be transmitted at the same time. This is therefore another form of multiplexing (data/address multiplexing).

We now have quite a lot of information, and thus might attempt to estimate some kind of initial performance measures for this bus standard:

Bus bandwidth estimate

Suppose that every single clock cycle allows a data transfer, then we transfer :

$$\begin{aligned} & 33 \text{ million} \times 32 \text{ bits, or } 33 \text{ million} \times 4 \text{ bytes} \\ & = \mathbf{132 \text{ Million bytes/sec}} \end{aligned}$$

So bus bandwidth appears to be about **126 Megabytes/sec.**

This is the correct answer in the absence of any other information, but is also a somewhat simplistic assumption. What this calculation tells us is the **raw bandwidth** of the bus, but not the whole picture. Remember the specification: there are 5 clock cycles for arbitration, and one clock cycle for turnaround. These are **protocol** requirements of the functionality of the PCI bus that must be respected.

Bus arbitration is a process whereby a controller (in this case a PCI controller chip), decides which one among the devices in the system is allowed to be **granted ownership** of the bus at the next opportunity. It then spends some clock cycles setting up the data transfer - a total of 5 clock cycles.

Once a device has finished with the bus, it must relinquish or **release ownership** of the bus, so that another device can use it. This requires 1 clock cycle under normal conditions (the **turnaround cycle**).

So we see that actually, to transfer one 32-bit data word on the bus requires seven clock cycles (five for arbitration, one for the actual data, and finally one for turnaround).

This **protocol overhead** impacts performance significantly. If it takes seven clock cycles to transmit one data word, then our 132 Million bytes/sec is never going to be achieved. Indeed, we can only achieve 1/7th of this peak bandwidth, resulting in data transfer rates of around 18 to 19 Million bytes/sec, a very poor substitute for our original expectations.

In order to avoid the worst impact of bus overheads, the data transfers we perform on busses can often be performed as block transfers, similar

Figure 6.4: Data transfer efficiency on PCI bus with various block sizes. It can be observed that as transaction length is increased, transfer efficiency increases too.

Transaction Length	1	4	8	16	32	64
Clock cycles	7	10	14	22	38	70
Efficiency	14%	40%	57%	73%	84%	91%
Transfer rate MB/s	18	50	72	92	106	115

to the burst-mode of the DRAM we looked at in Chapter 5. But for bus transactions, these blocks are often much longer.

As an example of how this improves performance, if we transfer 10 data words in a single transaction, rather than one, then the cost is 16 cycles (5 arbitration, 10 data, 1 turnaround). This equates to a data transfer efficiency of around 62% ($10/16$), and a data transfer rate of over 78 Megabytes/sec ($62\% \times 126$ Megabytes/sec).

Interestingly, if we calculate the efficiency of the bus at a range of different data transfer block sizes, we see a trend emerge, as illustrated in Table 6.4.

From Table 6.4 we can see that as the number of words transferred per transaction increases, the efficiency of the data transfer also increases. This is because we only ever have one arbitration and turnaround cost but many data transfers. So it would seem that the answer to our problems is to increase data transfer block lengths to large numbers, and enjoy data transfer rates close to the maximum possible. This is indeed part of the solution. However, there are problems with this idea:

- ▶ We do not always want to transfer data in large blocks, so there will always be cases where short and low efficiency transactions will occur.
- ▶ The longer the block size, the longer other devices have to wait until their next turn. This impacts upon fairness, and ability to respond quickly to events.

In conclusion, we can see that with this very shallow view of the PCI interface, there are limitations on performance due to the bus protocol. However, all bus protocols have some kind of requirements and overheads, so there is nothing particularly inferior about PCI; indeed, it is one of the most widely used bus standards for internal module connectivity in general purpose computer systems.

6.6 Data rate matching and buffers

One aspect that might be apparent from studying the concept of data transfer on a bus, is that the capabilities of two devices may differ, even though they may share the same bus connection. This can also happen at a bridge, where a device on a fast bus is trying to send a block of data to a device on a slower bus on the other side of the bridge. In this situation we consider the idea of a **producer** and a **consumer**, which are in effect the sender and the receiver during a bus transaction, or succession of transactions between the two entities.

When a producer creates data faster than a receiver can consume that data, the receiver must **buffer** that data locally, until it can work through the backlog. This is something like an in-tray piling up in an office. As long as the buffer never gets completely full, the producer can continue to send data as fast as the bus will permit.

Meanwhile, if a receiver consumes data faster than it is being sent, then the receiver may spend time idle when it would be better doing useful work. This producer-consumer dependency reduces efficiency if the two entries are not well matched. We will see later that operating systems attempt to deal with programs wasting time waiting for an IO event, by using various resource management techniques.

Meanwhile, in the case of a bus bridge, incoming ports of the bridge can have local buffering, to allow block transfers to complete on the producer side of the bridge, and then for that data to be forwarded to the consumer bus device at a slower rate a little later. This means that the producer does not have to stall its data transfers.

6.7 USB: Universal Serial Bus

We will now look at an example of another kind of bus technology, with different applications.

It is more than likely that you have heard of **USB**, or more specifically the **Universal Serial Bus** (USB). Whereas PCI is ideal for **internal** devices such as modules plugged into a computer system motherboard, the USB system is optimised for convenient **external** peripheral connectivity (in other words, USB is a **peripheral bus standard**).

These days you will find that, most often, peripherals use a USB connector, or at least provide the option, and keyboard, mouse, printer,

[88] This is certainly true for the original USB standards, where power in the range of a few hundred milli-amps is usual. However, there are many variations, and some USB standards support up to 3 Amps, which is quite a considerable amount of power.

auxiliary storage, flash memory sticks, and many other devices will use this standard by default as the interface of choice.

One of the reasons why USB has become so widely used is the capability to provide power to devices, albeit relatively small amounts^[88], to the devices plugged into it. This is very useful for devices such as a computer mouse, touchpad, etc, which have low power consumption, but also allows other devices to eliminate the need for a separate power cable. Consequently, USB has allowed many power supplies and wires to be eliminated from the computer desk: undoubtedly a key advantage given the menagerie of external peripherals we find plugged into our modern computer systems. We will examine USB further, assuming for now that we are referring to the original USB 1.0 specification (there have been several revisions since this was first launched in 1996).

USB1.0 has some key features that are worth highlighting:

USB1.0 Key Features

- ▶ The USB bus can supply limited power to a connected device. Data is transmitted in packets, along with a header, and an error check code.
- ▶ Data rates of up to 12 Mbits/sec are possible for USB 1.0.
- ▶ Packets can contain anywhere from 0 to 8192 **payload** data bits^[89].

[89] Sending zero bytes may seem to defeat the purpose of having a packet to send data. Zero length packets are however used for certain special purposes within the USB system.

As we might expect, having investigated the limitations of the PCI bus, the true data rate possible for a USB bus is limited by the need to perform its specific protocol related signal activities, as well as to moderate the length of data transfers. There are particular points to note:

- ▶ Every USB transmission begins with a SYNC signal sequence to indicate the start of a new packet. This a fairly short pattern (roughly equivalent to about 8 bits of data), but it uses up bus transaction time for every packet sent.
- ▶ Every packet also includes a packet identifier (PID), which is also equivalent to 8 bits.
- ▶ Every packet also ends with an end of packet (EOP) bit sequence, though this is only equivalent to three bits.
- ▶ Every packet also includes an error check (a 16 bit CRC code).

We can see from this that every packet contains the equivalent of at least 35 bits before we even send a single data bit. This is the overhead due to the bus protocol.

There is a further complication with USB protocol, and this relates to an idea known as **bit-stuffing**. Because of the way USB operates, it has no bus clock. Instead, the data bits themselves allow a clock to be recovered, using a special circuit which detects the zero-one and one-zero transitions in the bit stream being received.

However, we have no control over what data the application is sending; as far as USB is concerned it is random. If too many consecutive bits are the same (all zeros or all ones), then the clock recovery circuit will eventually lose synchronisation. To avoid this, the USB hardware has to insert an extra bit to create an alternate transition when this situation is approached. It does this after observing 6 consecutive unchanged bits in the data sequence. Consequently, a data transfer may take up to 1/6th longer than expected, based purely on the number of bits inserted into the payload, in a worst-case^[90].

One of the consequences of bit-stuffing is that, even if every packet transmitted has the same payload length, the length of the stuffed packet, and its transmission time, can vary, or '**jitter**', by a certain degree^[91]. This can impact upon the predictability of system behaviour.

A USB 1.0 packet can contain anywhere from zero to 1024 bytes (0 to 8192 bits). If we assume the least data we want to send is 1 byte, or 8 bits, then it could take up to 43 bit periods to transmit 8 data bits due to the overhead (a data transfer efficiency of only 18%). On the other hand, if we use the longest packet size, 8192 bits can be transmitted in 8227 bit periods, an efficiency of 99.5%.

So, as in the case of PCI bus, it seems that the bus is more efficient if we send larger blocks, but the same negatives also apply: we don't always want to send large blocks of data in one operation, and the longer the data packets are, the longer other devices have to wait for their turn at using the USB^[92].

The data rate of USB 1.0 is rather slow, certainly by modern standards. Fortunately, USB 2.0 and USB 3.0 have been defined and released as updated bus standards, and also incorporate new features. USB 2.0 has a raw data rate of up to 480 Mbits/sec, and USB 3.1 operates at up to 10 Gigabits/sec.

When plugging a USB1 device into a USB2 bus system, things still work. This is called **legacy compatibility**. This is because USB releases are backward compatible with older bus versions, but they are then forced to run at the slower speed of the older bus. This is another example of legacy bus support, as mentioned earlier.

[90] How often will this happen? Bit-stuffing events will happen fairly often, but it is rare for an entire packet to require continuous bit-stuffing. Therefore, the 1/6th figure is the worst possible case, not a typical one. Nonetheless, it is worst-cases that dictate minimum service guarantees in system design.

[91] Jitter is conceptually a variation in some characteristic, and has many causes. This particular effect might be referred to as jitter in packet transmission time, or more loosely as **payload jitter** or **packet jitter**.

[92] Achieving good responsiveness and high efficiency is a balance, and this is often referred to as Quality Of Service (QOS), especially when it becomes apparent at a user level (e.g. jerky versus smooth video playback).

6.7.1 Plug-and-Play

Another important aspect of USB, is the concept of **plug-and-play**. It would not be a good idea to open up your desktop PC and start unplugging modules from the PCI bus with the power switched on. At the very least your system will probably crash, you may corrupt data, but at worst you may damage parts of your computer. These modules are not designed to be removed and replaced during system operation. On the other hand, a system where this is allowed is known as a **hot-swappable** system architecture, and sometimes as a **plug and play** system.

The USB standard is designed to facilitate fast and convenient connection and disconnection of peripherals, and sometimes these are plugged in and unplugged whilst the system is running. Hence the plug-and-play label. This does however require hardware and operating system functionality working together to achieve seamless operation. Nonetheless it allows the user to plug in a flash drive, unplug a printer, plug in a network adapter, unplug the memory stick, and so on, and all with the system happily carrying on with no problems at all. This, plus the ability to send power to peripherals over the same connection, is what has made USB so successful.

6.7.2 Hubs and ports

On a practical aside, each USB connector on the computer is known as a USB port. Typically, a general purpose computer will have 4 ports, though it depends (laptops often have two), some desktops may be configured to have more. However, any USB port can connect to a USB hub, rather than a peripheral, and this provides expansion capability.

For example, one USB port might plug into a 4-way USB hub, providing 4 new ports. Hubs can be cascaded, up to the point where the total number of devices in the system is a maximum of 127 devices. This is far more than the vast majority of users will need.

6.8 More common bus standards

A few other common bus and interconnect standards are of interest to computer systems design. Some of these have arisen from particular manufacturer initiatives, and others are widely used standards in particular applications. Table 6.5 shows a few of these.

Bus	Max Data Rates	Typical Uses
Firewire400	49 MBytes/sec	Firewire is a brand-named version of IEEE-1394 serial bus standard. Permits up to 63 devices, plug-and-play, used for video, storage, and high bandwidth peripherals.
Thunderbolt	5 GBytes/Sec (Thunderbolt 4)	Branded bus standard, up to 6 devices per connector port, high speed devices.
IDE/PATA	100 MBytes/sec	Integrated Drive Electronics, an early Disk storage interface bus.
SATA	600 MBytes/sec	Serial 'AT' Attachment, frequently used for disk interfacing in modern general purpose systems.
SCSI	2400 MBytes/sec	Small Computer System Interface, primarily for high performance and high-resilience disk systems (in conjunction with RAID configurations).

Figure 6.5: Some additional commonly used bus standards.

Thunderbolt in particular has become very widespread due to the prevalence of certain touchpads and smartphones, whilst **Firewire** is also widely used in APPLE products such as laptops and desktop machines. For disk storage systems, there are a whole class of bus standards, including **SCSI**, **IDE**, **SATA** (but also Firewire to some extent, particularly for external drives), and these provide various kinds of high speed data channels for disk storage units.

Often these disk storage interfaces use a concept known as **daisy-chaining**, whereby a series of individual connectors loop from one device to the next, forming a bus in the process. This reduces the need for multiple wires from devices to the central device. We will explore disk storage in more detail in the next chapter so we will not say more about this now.

6.9 Industrial and embedded standards

Whilst general purpose bus systems are designed primarily for high data rates, and convenience, there are other application domains where bus systems are used, and where requirements are quite different. A good example of this is in the **embedded systems**, and **industrial computing** domains.

In industrial settings, computer systems have to have resilience, fault-tolerance, immunity to electrical noise caused by heavy machinery, and

operate on power levels that are unnecessary or undesirable in office or domestic scenarios. As in the previous section, there are just too many systems to cover them all in detail. However, the following subsections will give an insight with a few particular examples.

6.9.1 I2C (Inter-Integrated Chip) bus standard.

A typical application example of the **I2C bus** system is shown in Figure 6.6 where our familiar simple computer system has been augmented by the addition of an I2C interface, typically a dedicated I2C interface chip, and a number of devices connected to the I2C bus which that chip supports. In some systems the I2C interfacing might be built directly into the CPU chip. In this case, the I2C interface would still be observed as a device with the same kind of functionality, but by incorporating it with the processor core it reduces the number of chips required to build a system.

In the example, the I2C bus has multiple devices connected, an LCD display driver chip, an EEPROM for non-volatile data storage, a temperature sensor, and a keypad interface chip. This system could easily be the starting point for a central heating controller. It has all of the requirements, except for a way to communicate with the heating boiler, but this could also be accomplished by an I2C link if desired. Note that only two wires are required for an I2C bus: a very small hardware cost, making system assembly easier, and also meaning that the numerous interfaced devices need only two pins to connect to a system, allowing their electronics to be kept modest.

The I2C system uses a very straightforward protocol, the **SCL (Serial Clock Line)** provides synchronisation, and is always generated by the bus master, so that the devices on the bus which are receiving data will know when data bits are present. The **SDA line (Serial Data)** provides the means to transmit data bits, one bit per clock pulse.

Any device on the I2C bus can be master; however, in the example given, only the CPU is capable of doing so. There is no overall controller for the bus. Any device may attempt to become master, and as a result, several devices might try to do this at the same time. However, the protocol is able to deal with this using a conflict resolution algorithm. The following is an example of how the system ensures only one master is active at a time:

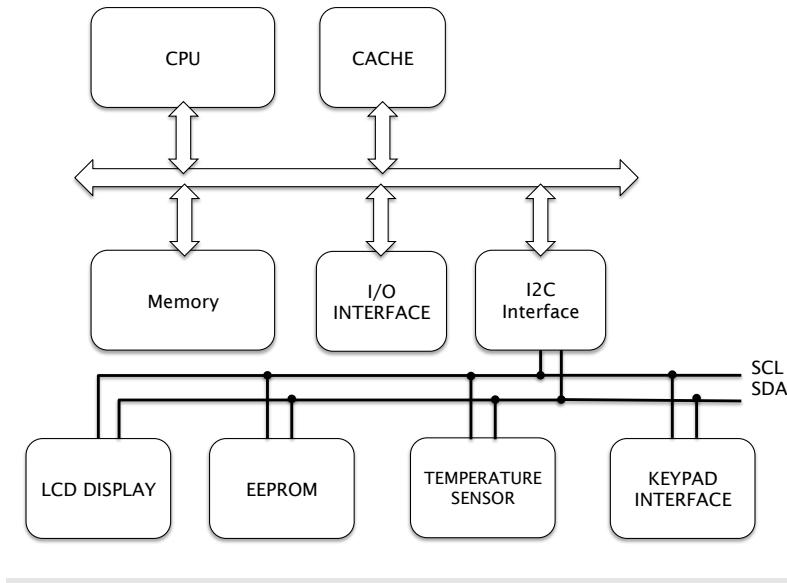


Figure 6.6: I²C bus added to our standard von Neumann architecture. Showing a system scenario in which an LCD display, a keypad, a temperature sensor, and a non-volatile memory are linked to the I²C bus.

Bus Master resolution protocol

- ▶ Potential Master waits if SDA appears to be active already (it cannot be master if another device is already busy).
- ▶ When SDA is inactive, the master attempts to start a transmission.
- ▶ The would-be master monitors the state of SDA, and if it is not as expected, then this means another device is attempting to be master too. The device then backs off and stops trying to be master.
- ▶ Otherwise, if no conflict is observed, the would-be device assumes master status, and proceeds.

This is an example of a **back-off-and-retry** arbitration mechanism. These are often used where there is no central controller for a bus system. In most situations, even if two or more masters attempt to gain possession of the bus, only one will succeed. This is based on the assumption that two masters will not communicate with the same slave device simultaneously (and by simultaneously, we mean exactly at the same time down to the first bit of transmission happening in the same clock cycle for both masters). In the rare case where this actually happens, problems can occur. Therefore, I²C systems are normally designed such that two masters never access the same slave, or never without some form of mutual agreement^[93].

[93] The concept of a semaphore as a software gate-keeper between two competing entities is one potential solution; another is some form of token exchange, or a third option is a time-division protocol.

The full specification of I2C transmission capabilities is beyond the scope of this text, but it is well defined in additional reading sources. There are also variations of this standard such as INTEL's **SMBus**, which implement subsets of the full I2C standard, and have become popular in certain systems applications.

6.9.2 CAN bus

CAN Bus (Controller Area Network) was originally designed for the automotive industry, to allow many electronic devices distributed around the chassis and bodywork of a vehicle to link together via a two-wire interface. The alternative was for every device to have a bundle of wires leading back to the central control module of the vehicle, and these 'wiring looms' as they are referred to, were complex and difficult to maintain. Having a single two-wire bus that can run around the vehicle from one device to another, is a much more streamlined solution in terms of wiring.

An important feature of CAN bus data transfer is the idea of **differential signalling**. Rather than having a single wire transmitting zeros and ones, the CAN bus system uses two wires, transmitting mirror image data patterns, such that if the first line transmits 0-1-0 as zeros and positive voltages, the second line would transmit zeros and negative voltages.

This duplication may seem like a waste of effort, but when the problems of electrical noise and interference are added to an ideal signal, the result is degraded reliability. However, the differential (mirror image) signal levels allow this noise to be cancelled out, thus making the bus very resilient to noisy situations that occur in factories, near heavy machinery, and of course in automotive products. In industrial situations, this can be an essential requirement.

The concept of differential signalling is shown in Figure 6.7 where the top case shows a standard one-wire signal level, affected by noise. This can cause data bits to be misread in severe cases: something that could be catastrophic in an industrial control system for example. The second panel shows differential signalling, where **D+** represents the normal signal, including noise, and **D-** represents the inverted signal, but with the same noise (not the inverse noise)^[94]. When these two signals are combined in the right way, the difference between them preserves the binary transitions, but cancels out almost all of the noise. This allows systems to work very reliably in electrically noisy environments.

[94] Because this noise is coming from an external source and affects both lines in virtually the same way.

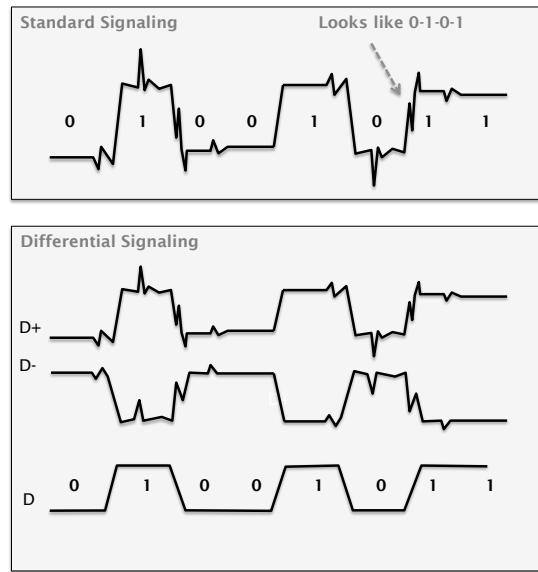


Figure 6.7: Differential signalling concept. Showing standard signal (top panel), and differential signalling (bottom panel).

6.10 Rack-mount, hot-swap, and servers

Whilst a typical desktop or laptop computer has a motherboard, and is built primarily for one user, the concept of a mainframe system has persisted into modern computing practice, if somewhat reborn in new guise.

The server platform is the modern equivalent of a mainframe, providing computing resources to many users as if simultaneously. To achieve this, servers are built with particular specifications, often based on racks full of motherboards (a **rack-mount** system), each with processors housing multiple processing cores, large disk arrays, and large amounts of memory. Single server racks could quite easily support several hundred processor cores. In some systems the compute resources are racked in one section and disk storage arrayed together in another section. This provides a more convenient way to organise and maintain modules.

Sometimes these systems are designed to be fault tolerant, and with automated backup systems, to ensure a users' data can be recovered in the event of a major system breakdown, and can even switch in a spare processor card to service a customer's program load in the event of a particular CPU failing. This often takes only a matter of seconds, so that services appear to be unaffected.

Such systems also rely upon the ability for engineers to connect and disconnect modules such as hard-disk units which may have failed, without having to turn off the computer system. This is known as **hot-swapping**, and is one of the design requirements of many server systems. As we have noted, some bus systems, such as USB and SATA permit this, whilst other bus systems do not.

6.11 IO device mapping and IO servicing

In Section 5.8 we learned about the idea of memory maps as representations of how devices, primarily memory modules, are mapped into the physical address range of the computer system. We also noted that a special case of memory mapping is the idea of an IO device appearing to be part of the address space.

Consequently, any suitable IO device or interface chip can appear as a group of memory addresses. This is known as **device mapping**. The CPU, or indeed any other device connected to the bus, can see these IO addresses, and potentially read from or write to them.

An important concept in **IO servicing** (the process of ensuring IO devices get the information they need when they need it), is the idea that a device can gain a suitable response from the system when a particular IO event occurs.

In terms of IO devices notifying the CPU that something of interest has occurred, there are two options. **Polling** requires the CPU to regularly read the status register of the IO device (via its IO address space), and check its status. **Interrupts** provide an alternative where the CPU never checks, but instead receives a special trigger signal (an interrupt) when an event arises. Using interrupts means that the CPU does not waste time checking lots of devices frequently.

Once a device has gained the attention of the CPU, the CPU will typically then control a data transfer, from the device to an area of memory, or vice-versa. This is known as a **CPU driven data transfer**. Again, this involves a lot of CPU effort, but this can be avoided if a device uses a technique known as **DMA**.

Direct memory access (DMA) allows an IO device to take over control of the system bus (become bus master) and transfer data to the chosen destination without CPU involvement. This could conceivably be

a device-to-memory transfer, a memory-to-device transfer, or a device-to-device transfer.

DMA is typically highly efficient as it often uses large block transfers, and often the CPU will continue to run its programs simultaneously, by relying upon its own on-chip cache rather than memory accessed via the system bus. Some interfaceable devices and chipsets support highly automated DMA. For example, an audio chipset may be able to fetch blocks of data from memory continuously by way of repeated DMA block transfers, without CPU involvement. This allows capabilities such as audio streaming to be smoothly executed without placing demand on the CPU.

One aspect that must be considered carefully with DMA is the impact this has on other devices in the system. If multiple devices want to perform DMA, then there needs to be some care taken to ensure that devices that need to maintain a minimum data-rate (such as audio playback) are getting enough bus bandwidth to fulfil their service requirements, even though other devices may want to perform DMA at the same time.

Therefore, there is a trade-off between having very long DMA transfers, causing other devices to wait a long time to get their turn at using the bus (service latency), versus much shorter transfers that allow all devices to get frequent use of the bus in turns, but with the penalty of lower data transfer efficiency.

6.12 Summary

In this chapter we have explored the nature of computer systems from the point of view of connecting components into working cooperative systems. We have explored the basic ideas of a motherboard, and the mechanisms that such a circuit board embodies as a hierarchical system of busses and ports.

The role of bus standards, providing clear definitions of bus protocols and operating principles helps us to understand the exact requirements for a bus performing a data transfer transaction, and the cost of having such specifications. We discovered that this protocol overhead has a performance impact, and that this can be estimated and evaluated, whilst the configurations and choices relating to issues such as data payload size will impact upon both the efficiency of the bus system and the latency - that is, the time delay between successive bus transactions, and the subsequent effect this has on user service levels.

Ultimately, with suitable design choices, we can design systems that utilise busses as mechanisms to divide and conquer data bandwidth and deliver superior performance.

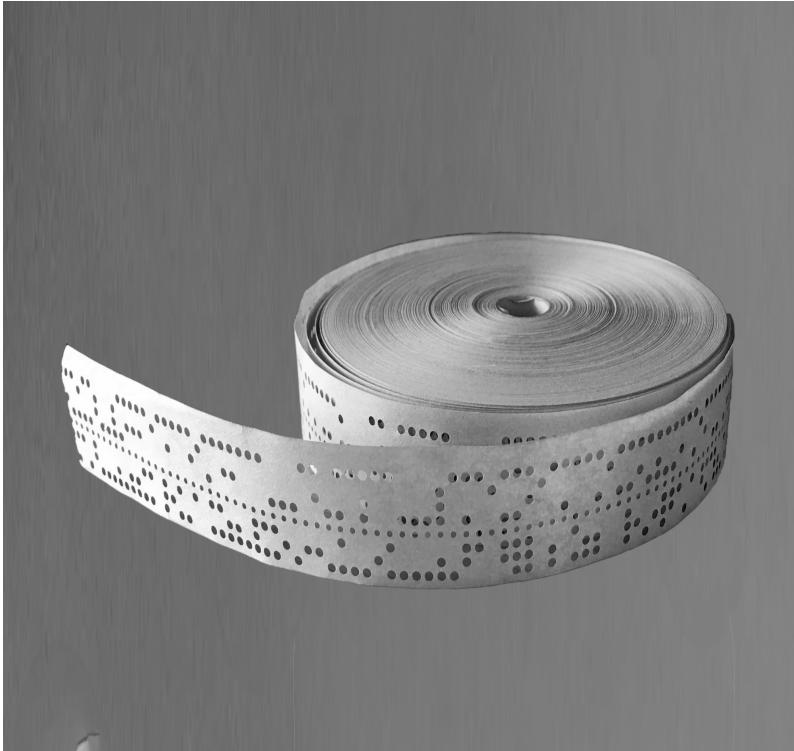
6.13 Terminology introduced in this chapter

Address line	Address-data bus
Addressable locations	Attenuation
Auxiliary bus	Available address range
Back-off-and-retry	Bit-stuffing
Broadcast	Bus arbitration
Bus bridge	Bus hierarchy
Bus master	Bus protocol
Bus slave	Bus standard
Bus transaction	Bus utilisation
CAN Bus	CAS
Clock driver	Column-Address-Select
Concurrency	CPU socket
Daisy-chaining	Data line
Dedicated bus	Device mapping
Differential signalling	DIMM socket
Direct Memory Access	DMA
Embedded system	Firewire
Front-side-bus	Host bus
I2C bus	IDE
Interrupt	IO servicing
Legacy bus standard	Local bus
Main bus	Mainboard
Memory bus	Memory controller
Motherboard	Multicast
Multiplexed addressing	PCI bridge
PCI bus	Plug-and-play
Point-to-point	Polling
Printed circuit board	Peripheral bus standard
Producer-consumer	Rack-mount
RAS	Raw bandwidth
Read-transaction	Row-Address-Select
SATA	SCL
SCSI	SDA
Secondary bus	Serial clock line (I2C)
Serial Data line (I2C)	Turnaround cycle
Universal Serial Bus	USB
Write-transaction	

These terms are defined in the glossary, Appendix A.1.

7

Data Storage



Punched-Hole Paper Tape, 1960/70s era, data storage system. Holes punched in the tape represented binary data values. This system was used as a basic storage medium until magnetic media became less expensive and more convenient toward the mid 1970s.

7.1 Storing Data	128
7.2 Disks and data storage	128
From tapes to disks	128
Modern disks in detail	131
A disk performance example.	134
The cost of fragmentation	137
7.3 Disk cache	138
7.4 Solid State Drives (SSD)	139
SSD memory wear	141
Storage density . . .	142
Buyer beware . . .	143
7.5 Storage reliability and fault tolerance	144
7.6 Remote storage . .	145
7.7 Summary	146
7.8 Terminology introduced in this chapter	148

[96] Indeed, punched cards as a data storage medium were used throughout most of the 1900's, following Herman Hollerith's inventions of the 1890's. A punched card is shown in Figure 7.1



Figure 7.1: A punched card and job card from a FORTRAN program.

7.1 Storing Data

In this chapter we will examine a key additional component that is needed to make a computer system complete. This will include a significant investigation of data storage, one of the most important components not directly supported on a computer motherboard, and which comes in a number of forms.

7.2 Disks and data storage

As we have already observed, computer systems rely upon several different kinds of data memory, which we refer to as the physical memory of the computer system. However, the data storage capabilities of memory are often volatile, and rarely big enough to store all of the data the system will ever use. To overcome this limitation, computer systems have long utilised the concept of **backing store**, or in modern day terms, **disk storage**.

Computer systems of the 1960s and 1970s relied upon fairly antiquated technologies to store data in non-volatile fashion. **Punched cards** and then **punched paper tapes** were used to store data using holes in paper and card to represent storage symbols, which could then be read by a paper tape or punched card reader^[96]. Later, **magnetic tapes** become available, and the natural next step was to replace holes in paper tape with magnetically manipulated data signals. Some time later, the concept of **magnetic disks** became feasible, and until recently this has been the basis for most non-volatile but rewritable data storage systems.

7.2.1 From tapes to disks

The original magnetic tape systems, which you may only have seen in old sci-fi films running in the background in some 1960's science lab, made use of the concept of a **magnetic recording head**, not that different to those used for recording audio tapes. In a fairly well developed tape storage system, the tape was wide enough to accommodate multiple recording heads across its width, as shown in Figure 7.2, in such a manner that a number of bits could be recorded at each tape position. These adjacent bits were known as **tracks**. Then, as the tape was moved (**spooled**) through the **read/write head assembly**, data could be read or written. There is an obvious problem with this system, and

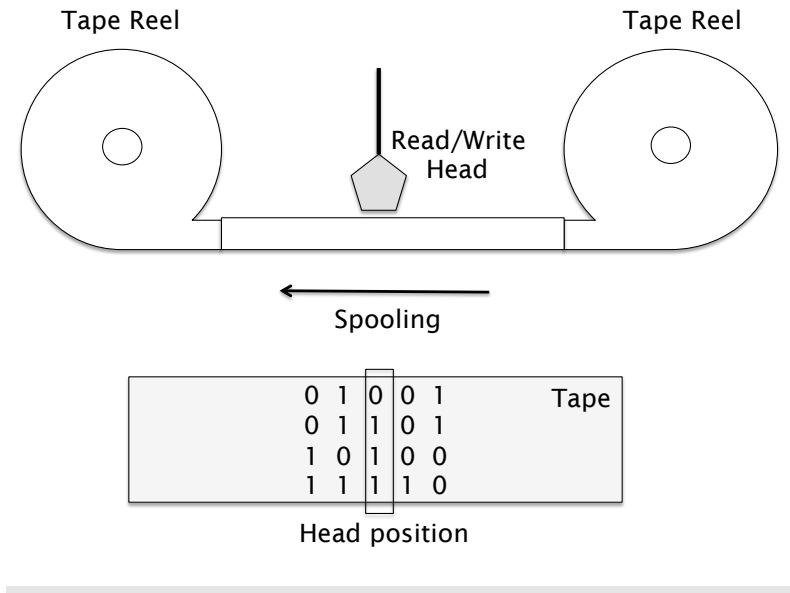


Figure 7.2: Tape System Conceptual operation. The tape reel allows tape to spool from one reel to another, in either direction. The read-write head typically has multiple positions creating multiple bits per tape region. These bits are then read or written in parallel.

that is the issue of **serialisation** of data storage. If we were to start at the beginning of a tape, and wanted to access the data at the other end, then the tape machine would have to spool the tape at high speed until we reached the point on the tape we wanted to access. This often takes far longer than actually reading the data. Therefore, there are two time factors for a tape storage system, the **seek time**, and the read (or write) **data transfer time**.

This observation is quite important: data transfer rates can be quite high with tape systems, once the data location has been reached by spooling to the correct place, but there is a disproportionate delay in reaching that position.

Magnetic tapes (in cartridge form) can still be used for emergency data backups due to their large capacity and robustness. However, it soon became apparent that significant speed gains could be achieved by rearranging the magnetic surface into a two-dimensional storage medium (as opposed to tape which is as good as one dimensional).

Although tape systems are seen as somewhat of an old technology, they have not disappeared. Tape systems have continued to evolve, primarily for the purposes of providing high density back-up storage for large quantities of data, for data security, or archiving^[97]. Indeed, companies such as SONY, IBM, and FujiFilm Corp, and others, are continuing to push the limits of tape storage using advanced materials and magnetics

[97] For example, an automated robotic tape storage library can store hundreds of petabytes of data.

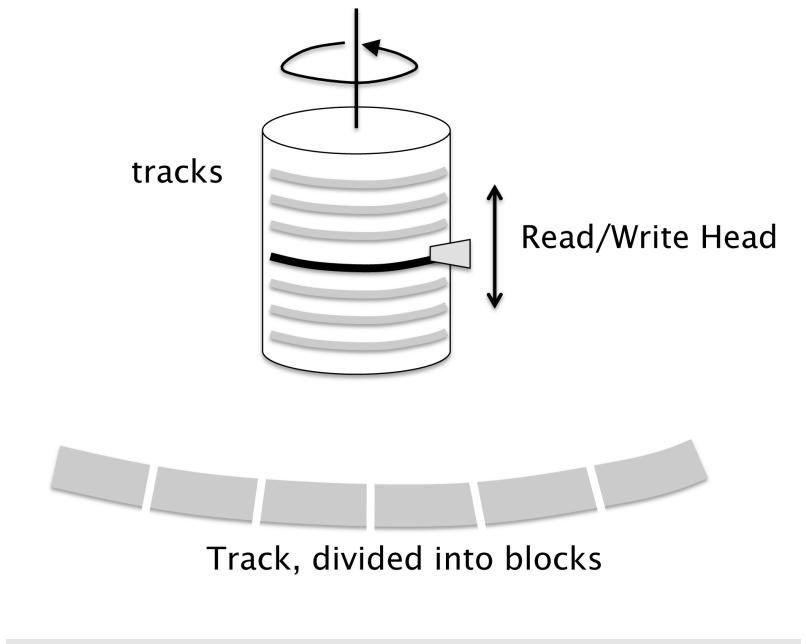


Figure 7.3: Drum Storage concept. The drum is divided into single tracks, like many short sections of tape. Instead of spooling a very long tape, a drum can jump rapidly between tracks (seek) to get to the data of interest, making access time faster.

science. Typical tape capacities have grown by about 40 times in the past ten years, and are likely to continue to improve well into the 2020's.

The first system to move to a true two-dimensional storage medium used a rotating **drum**, which was simply a cylinder upon which magnetic material was deposited to create a recording surface. Such a system is shown in Figure 7.3. Conceptually it was little more than a very wide piece of tape formed into a loop. The advantage of this system was that the drum effectively had hundreds of tracks, and rather than having to spool through every piece of data, the drum reader could jump to the track of interest and then rotate the cylinder by a small amount to get to the particular data block. Thus the seek time was much faster than tape. Later, for reasons of compactness, drums were replaced with **disks**.

A disk (or disc) is also a 2-dimensional magnetic recording surface, and just like the drum, it can rotate, and the read-write head can jump quickly to the track of interest and then rotate a little way until the data within the track is found.

7.2.2 Modern disks in detail

Since disk storage is the only major storage medium used almost universally in computer systems at present we will focus upon this system in detail. We will find that the same issues of data transfer rate and seek time still apply, though they are much more efficient in a modern disk system. At this point it would help to look at a disk unit conceptual diagram, and get to know some of the key terminology. This is shown in Figure 7.4.

As we can see in the diagram, a modern **Hard Disk Drive (HDD)**^[98] often consists of multiple disks, known as **platters**. These can also be double-sided, allowing data to be stored on both surfaces of each platter.

In the example of Figure 7.4, there are three platters, so there could be up to six recording surfaces^[99]. Each surface can be accessed via a read-write head. In most systems these are mounted on a single **armature**, but some systems have been designed with heads that can move independently of each other^[100].

In order to locate data, the disk unit control circuit translates the request for a particular data block into a platter location^[101], and then a particular track. The drive then locates a particular sector of that track, within which the data of interest can be found. In a disk unit the time taken for an armature to move to the required track is the **seek time**. Finding the sector requires rotation, and hence incurs **rotational latency**.

This description highlights an important point about disk storage systems: the smallest data block that can be read or written is based on the **sector size**. Sector size might, for example, be 512 bytes. As a result, a file^[102], which can be an entirely arbitrary size, has to fit into an integer number of sectors. For example, a small text file might be 300 bytes long but it will still occupy (or at perhaps more precisely 'reserve') an entire sector, since we cannot store data in smaller blocks. A larger file, say 5000 bytes, needs ten sectors, and so on.

You may notice, then, that it is very unlikely that a file ever occupies exactly 'n' number of sectors, and there is always some wasted space on the disk for a given file. If the sector size is 512 bytes, then the worst case **sector wastage** per file must be 511 bytes, and the best case wastage is zero. So on average, each file wastes about 0.5 sectors of storage space (256 bytes in the case of a 512 byte sector).

[98] The Hard disk is a term used to refer to internal drives that rely upon metal platters, often stacked as mentioned. Floppy disks, on the other hand, are single platter removable disks, where the platter is made of a plastic material. These are low storage density storage disks designed to be removed and transferred from one computer to another. These days, floppy disks are rare, as USB stick drives are far more convenient.

[99] However, one of these is usually reserved for special purposes within the drive system, for example to help align heads accurately, so in actual fact, three platters translates to five usable data surfaces.

[100] This idea has recently seen a resurgence of interest as drives have become much larger in capacity whilst disk caches have not kept pace. Drive manufacturer SEAGATE is a leader in this area currently.

[101] This results in **controller delay**. However, this is a very small time period compared to other factors, and is generally ignored.

[102] A file is simply a block of data of a particular size, stored in memory or on disk.

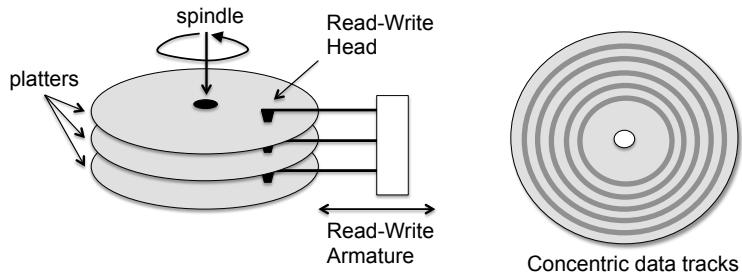


Figure 7.4: Disk Unit Conceptual Diagram. Showing the circular disk layout, and an arrangement with multiple platters on a spindle.

Now, it may not matter much if a file wastes a few hundred bytes, if the files are typically multi-megabytes in size. However, systems with large sector sizes are not efficient for applications where large numbers of small files are to be used, because the wastage ($0.5 \times$ sector size) is significant compared to a small file. This should not be overlooked when trying to determine file storage requirements. Indeed, when copying data between two disks of the same size, but different sector sizes, it is entirely possible that the data will not fit on the second disk: a somewhat confusing situation if the reason is not appreciated!

Disk systems are organised to try and minimise seek time, maximise data transfer time, and achieve the highest possible data storage capacities, whilst remaining reliable. This is a balancing act between a number of parameters. Seek time can be reduced by having more tracks per disk, and by having more platters to choose from, but this demands read-write heads to work on smaller and smaller scales (storage density is increased). Modern disk units use very high precision head positioning systems to achieve this goal.^[103] Likewise, packing more data bits into a sector requires physically smaller read-write heads, and smaller magnetic areas on the disk representing individual bits (known as domains).

At the same time, the **rotational speed** of the disk, measured in revolutions per minute or RPM, dictates how fast the required bits can be rotated into position and read/written. Typical speeds are of the order of 5400rpm to 7200rpm in drive technologies at the present time (2019), but higher speeds can lead to shorter drive lifetimes, more power consumption, and so on. Nonetheless, some high performance disk systems use rotational speeds up to 15000 rpm, reducing seek time, but at a cost of higher power consumption. The impact on seek time contributed by rotational latency (the time taken for the platter to perform a maximum of one rotation) is highlighted in Table 7.1, where it is clear that low rpm leads to large contributions to seek time.

[103] Techniques have included **high-precision servos**, **stepper motors**, and **voice-coils**

Rotations per minute	Rotations per Second	Time for 1 rotation
5400	90	11.1 milliseconds
7200	120	8.3 milliseconds
15000	250	4.0 milliseconds

Table 7.1: Examples of rotational speed and contribution to total access time.
Faster rotations mean lower access latencies.

When the rotational latency is added to the time taken for the head armature to move to the correct track, that is the seek time, then we arrive at a figure for the total access time for a data access. Rotational latency is also related to the **sustainable data rate** achievable by a disk unit. Whilst it represents a delay in terms of access time, because of the time taken to rotate to the start of the sector of interest, it also represents the time taken to read one whole track once the data is in the correct position. The faster that the disk rotates, the faster that data can be read, and the more densely the bits can be packed into the tracks.

If we know the rotational latency, the number of sectors per track, and the number of bytes per sector, then we can calculate data rate per read-head:

Data rate Example

Suppose a track contains 100 sectors, and each sector contains 1Kbyte of data. Let us also assume that we have an 8.3 ms rotational latency at 7200 RPM ($60\text{s}/7200\text{rpm}=8.3\text{ms}$). Thus, 100 sectors can be read in **8.3 ms**, assuming a single read head.

Therefore, if 100 sectors can be read in **8.3 milliseconds**, then 1 sector can be read in **83 microseconds**.

If one sector contains 1Kbyte, (1024 bytes) then we can read 1024 bytes in **83 us**.

Therefore, we can read one byte in $83 \times 10^{-6} / 1024 = 81$ nano seconds. And, if one byte requires **81ns**, then:

$$\text{Data rate} = 1 / 81 \times 10^{-9} = \mathbf{12.3 \text{ million bytes/s}} = \mathbf{(11.8 \text{ MB/s})^{[104]}}$$

A complication here is that modern drives use the concept of **zoned bit recording**, such that the number of sectors on the outermost tracks is higher than the number of sectors on the innermost track. This is

[104] We need to be careful here, as one million bytes in decimal is not the same as a binary megabyte (1024×1024 bytes). Our decimal megabyte figure would be around 12.3 Mbytes/sec, but in binary megabytes it is 11.8 million bytes/sec. This illustrates an important point: sometimes a figure is driven by an underlying binary (base-2) property, such as sector size in this case. Therefore, we need to take care with unit definitions.

because the outer tracks are longer and can accommodate more bits at near-optimal bit-density. Rather than making sectors have different numbers of bits, ZBR allocates more fixed-size sectors per track as we move from inner to outer tracks. Data transfer rates on the outer tracks may also be higher than those for the inner tracks, since the same period of rotation delivers more bits to the head. Therefore, a very fast rotating disk with very high storage density per square millimetre of disk surface, will deliver high sustainable data read rates.

We should be able to realise from the above example that the data transfer rate can be increased simply by doing one or more of the following:

- ▶ Rotating the disk faster,
- ▶ Packing more bytes into a sector,
- ▶ Having more sectors per track.
- ▶ Reading tracks in parallel (multiple read heads)

It is obvious then, that manufacturers will benefit from developing new technologies that allow more and more bits to be stored on a disk per square millimetre of magnetic surface area, and indeed this is exactly what has happened for the past 20-30 years.

The science of miniaturising the recording of data bits magnetically on a disk surface has become quite complex, relying upon very fine precision engineering, and some fairly deep exploitation of the physics of materials. We will not go into all of those details here, but there are many articles about this quest for ever larger and faster storage systems for your own research to follow.

7.2.3 A disk performance example.

At this point, we shall try to put some of our understanding of disk mechanics to good use, and attempt to predict realistic performance expectations. We will use rpm to determine average, best, and worst case access time, data transfer rate based on sectors per track, sector size, and rpm, and calculate data read time for a 3 KiloByte block with an example.

In this case we will use some arbitrary figures for drive performance, not far from a reasonable set of assumptions. For a specific hard-disk you would of course be able to obtain the technical data from the manufacturer's web-site. Datasheets are supplied for most components in computer systems, and give a lot of insight into capabilities and limitations of particular components.

An example: estimating file access time. A drive has a rotational speed of 5400 rpm, a data read rate of 100 Megabytes/sec, and a sector size of 2 KB. What are the worst, best, and average times to access our 3 KB data?

Let us assume that deciding which platter, and which track to access is almost instantaneous (it is an internal control circuit operation taking microseconds at most).

The drive now needs to move the head to the correct track. Let us assume this seek time requires a maximum of 2 milliseconds (this would be known from the datasheets provided by the manufacturer^[105]).

The disk rotates at 5400rpm, or 90 times per second, so it completes one rotation in 11.1 ms. A 3Kbyte file occupies 2 sectors of 2 Kbyte, so the disk unit needs to read 4 Kbytes in order to complete its read activity for that block of data^[106]. At 100 Megabytes per sec, this requires about 40 microseconds.^[107] Now let's try some performance estimates:

Worst case:

Head movement (seek time) 2 ms, plus one full rotation, 11.1 ms,

Total access time therefore = $11.1 + 2\text{ms} = 13.1\text{ms}$

Then a further 40us to read the data (transfer time).

total read time = 13.14ms.

This implies a **peak data transfer rate** of about **0.3 Megabytes/sec**, if we assume two sectors provides 4096 bytes of data, but only 0.23 MegaBytes/sec for the actual 3 KByte file size in question.

[105] We can actually estimate this, if we know the rotational latency of the drive and the manufacturer quotes total worst-case access time, then seek time is the difference between rotational latency and total access time.

[106] Unfortunately we cannot assume that the disk controller will release the data before reading an entire sector, even if only part of it is used. For example, error checking, and other low-level considerations, may operate on a sector as an invisible unit.

[107] To be exact, 0.039 milliseconds, though in reality the true data transfer rate might depend upon the location of the track (inner, middle, outer, etc), therefore we use an average figure here.

Best Case

Head Movement 0 ms (head by chance is in just the right place),

no rotation required (by chance, the disk is in just the right rotational position),

Total access time = 0ms

Then read data 40us, and total read time = **0.04ms**.

This implies a **peak data transfer rate of 97.7 Megabytes/sec**. Again, for the actual 3 KiloByte file it is actually lower, at around 73 Megabytes/sec.

Average Case

From the worst and best cases, we can determine a simple average case: $(13.14 + 0.04) / 2 = \mathbf{6.59 \text{ ms}}$.

This implies a **data transfer rate** of around **0.60 Megabytes/sec.**

Note, it is important to appreciate that this only estimates the time taken for the process of getting the data from the magnetic surface into the memory buffer of the disk unit. The transfer from disk to CPU memory potentially requires further time and effort, and depends upon the bus interface used to connect the disk unit.

In this example we also observe that small files are dominated by the physics of the disk access mechanics, whereas larger files would be less impacted, and therefore are potentially able to deliver higher effective data transfer rates than smaller ones. Indeed, the actual useful data read in each case was only 3 Kilobytes, so we could have used that figure to calculate data rates and got an even more pessimistic figure. For the case evaluated, using a very small file test case, we see that average performance is about 150 times lower than the best case performance of the drive.

Whilst a simple average performance can be obtained by assuming that files are spread equally over the disk (as we did in this example), the reality is that files may be clustered together around a heavily used portion of the disk, whilst other areas might be unused or lightly used. This means that average performance in live systems might be better than this estimate. What those systems cannot do is **guarantee** that performance **will** be better, not unless they deliberately arrange files on the disk in a way that enhances this effect^[108].

It is also important to appreciate what this means for effective data transfer rates. The disk manufacturer may quote a data transfer, let us say 500 Megabytes/sec, meaning the ability to transfer data between disk and CPU system bus. However, this does not mean that we can read data off the disk at that rate in a realistic file system, or indeed at the rotational speed of that particular disk. A more useful figure quoted by drive manufacturers is **maximum sustained transfer rate**. Yet we see in our example that even that figure is not telling the full story.

If we know how big our files are typically, then we can make fairly accurate estimates of how much data a hard disk can deliver under peak load. We could alternatively use a benchmarking tool (which tests the disk with various file sizes and combinations) to give a reasonable measure.

[108] This is of course possible: with the right file system policies, or by using suitable utilities to reorganise the disk content to optimise this behaviour with varying degrees of sophistication.

However, since this behaviour is often a run-time characteristic, it may need to be evaluated through profiling of real systems during their use in order to derive suitable optimisation insights.

Importantly, you should appreciate that the physical disk unit impacts upon wider performance. If we had a system that operated quite nicely with a particular disk unit, and then this was upgraded to a larger disk, it may actually reduce the system performance, due to different disk mechanics, and other aspects of its internal organisation. This could be a serious issue. Consider, for example, a situation where a back-office server system is handling enquiries from a customer ordering system at an airline office, just about coping with the workload, and then suddenly the system appears to be taking longer to do everyday tasks because perhaps the RPM of the new disk is lower after an upgrade.

7.2.4 The cost of fragmentation

If we were to repeat the previous example, but assume a much larger file size, then the data read rate and read time would eventually become dominant. However, this perhaps reveals another over-optimistic view of how files are stored on disk. In reality, the possibility of finding many sectors of a file next to each other on the same track is actually not 100%. Instead, file data is often spread across the disk, with sectors scattered here and there. This is known as **fragmentation**.

Imagine you work in a shop, and the customer asks for five items, so you go to the store room, and the first shelf you come to has the exact same five items sitting right next to each other on the shelf. That's great: it takes you no time at all to access the items and bring them to the customer. Another customer comes in, and also asks for five items, but this time, each item is on a different shelf, and in a different position on the shelf. It will take much longer to do the fetching this time.

This is the principle of file fragmentation. When all of the sectors of a file just happen to be on the same track, and in just the right order, this is known as a **contiguous file** (it has no gaps, and appears like a continuous series of data blocks). When a file does not have all of its sectors on the same track, in just the right order, and with no gaps, then this is known as a **fragmented file**.

A fragmented file with n sectors could have up to n separate fragments. On the other hand, it may have none, or only have one or two discontinuities in its series of sectors. In general then we could say that any file will have between zero and n fragmentations. But is this a problem?

Let us consider our previous file access example again. We had a 3 Kilobyte file split into two 2 Kilobyte sectors. In our previous calculations we assumed (without realising it) that the file was contiguous. However,

if those two sectors were in different places on the disk, then things get a lot worse. Consider two cases, one being our worst case read time from the previous example (the contiguous case), and the other being the same case but with a fragmented file.

Fragmentation impact estimate

Contiguous Worst Case:

$$2\text{ms} + 11.1\text{ms} + (2 \times 0.02 \text{ ms}) = 13.14 \text{ ms}$$

Fragmented Worst Case: (with one fragment)

$$(2\text{ms} + 11.1\text{ms} + 0.02\text{ms}) + (2\text{ms} + 11.1\text{ms} + 0.02\text{ms}) = 26.24 \text{ ms}$$

So what has happened is that the access time for head and rotation mechanics has to be duplicated for each incidence of a sector that is fragmented. In general, every fragmentation of the file could result in a worst case access time per sector read. The best case could still be close to zero, so the average is now around 13ms. Clearly if the file has lots of fragmentations, then the read time will become quite slow, even to the degree that it is noticeable to the user to an extent that is undesirable.

One solution to this problem is to use a **defragmentation utility**, which is a program that iteratively reads and re-writes the entire disk content in such a way that it attempts to make every file contiguous: all sectors placed next to each other on the same or immediately neighbouring track, and in the right order, for example. This significantly impacts on system performance, theoretically maximising the transfer rate for each file.

7.3 Disk cache

There are additional tricks that will improve disk performance, and our old friend, the cache, pops up again as a solution. Most modern disk units have an inbuilt cache memory module, which acts as a data buffer and as an optimising function. Consider the following examples that might be relevant:

- ▶ A computer program reads the same file repeatedly at intervals.
- ▶ A computer program begins to read data from a file, stops part way through and then resumes reading
- ▶ A computer program reads multiple files in an intermixed fashion (jumping from one file to another, known as interleaving)
- ▶ Files are fragmented.

We can see here, if we understand how cache works, that these very common cases can be helped by cache. Where the same file is read repeatedly, then keeping that data in a cache means we no longer have to go through the long-winded process of accessing data on the disk. We eliminate all of the mechanical access time costs associated with that file.

Where a program stops and starts during file reading, each resumption of reading would require a new access process, potentially adding up to a full 13ms each time, as calculated in our example. If, on the other hand, the cache is able to read the whole file in an uninterrupted flow, it can appear as if the next part of the file is instantly available when the program resumes reading, with zero apparent delay.

Here the cache can potentially read the whole file in one operation, with the expectation that it will be needed later. Of course if this turns out to be a bad assumption, then electrical power has been wasted reading the extra data only for it to be ignored, and this means there is a cost-penalty as well as a benefit.

Where a program is jumping around between multiple files, the ability to cache some or all of these files will also eliminate the undesirable **interleaved seek operations** that this entails. Again, performance will potentially be much better when this happens.

Finally, in the case of file fragmentation, a badly fragmented file, once read into cache, no longer suffers the consequences of fragmentation.

Note that whilst disk cache is a valuable enhancement to general performance, it cannot make the initial file access faster, only those successive accesses to the file. Also, just like memory cache, there is limited capacity and content can be discarded from the cache to make way for new data, so we cannot assume that a file once read will always be in disk cache.

For more on this topic, see CASE-STUDY ??.

7.4 Solid State Drives (SSD)

A relatively recent development in data storage is the **Solid State Drive**, or **SSD**. This technology is still developing, and so is relatively costly compared to the much more mature state of magnetic disk system hardware. Even so, the SSD is much faster.

We have already encountered the idea of non-volatile memory in Section 5.2, and we learned that certain memory systems, such as **flash memory**, allow data to be read and written electronically, yet retained after power off. This is essentially a storage device in the same way that an HDD is a non-volatile storage device.

In theory, therefore, we could use flash memory in place of disks. This is essentially what an SSD does. They contain large amounts of flash memory, organised alongside a memory controller that converts disk access commands into memory reads and writes. The result is that the SSD looks identical to an HDD as far as the CPU and operating system is concerned. However, the SSD has no mechanics, and therefore none of the complications of head seek, sector rotation, or magnetic read/write constraints.

All of this makes SSD very fast, even when files are non-contiguous, and even where programs interleave between different files frequently. Indeed, whereas a typical HDD might manage a 100 Megabytes/sec sustained data transfer during a file read, an SSD could potentially achieve 20 times this performance^[109], and without any mechanical access time delaying the start of data transfer and without making small files inefficient. The only negative is that write speed is a lot lower, due to the way flash memory has to be programmed. Flash reads are very fast but flash writes are fairly slow.

Because SSD's are still multiple times the price of magnetic storage for a given capacity, they have to be used sparingly, unless money is no object, and if the purpose is biased heavily toward read performance then it maximises the benefit. A good example would be to store operating system and application code to make everyday systems run faster and more smoothly. The operating system and installed software packages do not change often but they are read heavily during use. Likewise, a video server could deliver semi-permanent data content at high speed to many viewers at the same time, and the time taken to load new videos onto the SSD would not be a big issue as it would likely be done during a quiet period, such as the middle of the night.

SSD's can also be used for accelerating disk swapping in virtual memory, or for acting as a temporary work space in very file intensive data processing tasks. However, costs will reduce in time, and in all likelihood, we will see SSD become the standard system component over the next decade, particularly if faster write speeds can be achieved. HDD systems are then likely to be phased out, just as magnetic tape was.

[109] Indeed, the main restriction on SSD read performance is often the system bus interface rather than the SSD memory chips themselves, which can be organised in many ways to boost data read rates if desired.

Interestingly, at the time of writing the first edition (2019) a recent trend in hybrid drive technology (SSHD) had begun to be seen, with very large magnetic disks combined with fairly small SSD memory technology to create a more cost effective but large format drive technology. As with all of these disk technologies, there are niche areas where each may perform well, but they are not perfect in all settings.

7.4.1 SSD memory wear

In all current flash-based memories, the ability to overwrite data cells in the structure is limited to a certain number of reliable write cycles over its operable lifetime. When a memory cell is near its write-limit, it may begin to lose data integrity; typically this happens over periods of years. Read data capacities, on the other hand, are effectively unlimited.

An SSD bit-cell can also theoretically become entirely non-functional with excessive use^[110]. In order to avoid this becoming an issue, a drive will attempt to write to different blocks of physical memory each time a new block is stored. This distributes the write activity across the whole SSD memory space rather than creating excessive use in small areas. This method is known as **wear-levelling**.

The wear issue may not be as problematic as it might seem: A 2 Terabyte drive, rated at 300 Terabyte total guaranteed writes, doesn't sound like much. However, the problem does not affect read cycles, and files are rarely overwritten repeatedly in their entirety at high frequency. Rather, only small parts of files are usually updated. Secondly, remember that drives also have inbuilt cache, and if a write-back cache is used, then the vast majority of repeated writes will hit cache, and much more rarely the actual SSD storage flash array.

Even if we consider a case where new files are being generated daily, the wear-out of bit cells may not be quite the worry it might appear to be.

Consider a 2 Terabyte (2 TB) SSD unit, where a user fully updates 10 Gigabytes of data files every day (quite a heavy user by typical standards): How long should this drive last if the storage cells are guaranteed for a total write capacity of 300 TBW ?

[110] For example, the SAMSUNG EVO 850 Pro series SSD has a wear rating of 300TBW (MZ-7KE1T0, 1,024 GB), and 450 TBW (MZ-7KE2T0 2,048 GB) respectively, where TBW is TeraByte Writes. Interestingly, the doubling of drive size doesn't yield a doubling in write capacity in this example.

Wear Capacity Example (using decimal megabytes)

At 10 Gbyte per day, the user would write **3.65 Terabyte per year**.

At this rate it would take **$300/3.65 = 82$ years** to wear out the drive.

Most SSD units are only guaranteed for 3 to 5 years anyway, so there is no prospect of this user wearing out the drive during its planned lifetime, even at ten times the estimated use.

Now consider a few other examples of user data-write requirements:

- ▶ **Operating System:** maybe 10 Gigabytes of storage, perhaps 10% of files updated every 3 months, impact 4 Gigabyte/year
- ▶ **Computer Games:** typical game 10 Gigabyte, install one new game per month : 120 Gigabyte/year
- ▶ **Document Editing:** typical document 5 Megabyte, change per day 5%, impact 0.25 Megabyte/day, about 65 Megabyte per year for a 5-day working week.

So unless a user is doing something very demanding, the problem of wear is unlikely to be encountered^[111]. In such extreme cases, the use of a highly optimised HDD configuration would be preferred. Meanwhile, if a choice of drives comes down to paying a high premium for a relatively marginal increase in write-wear rating, think carefully about the extra money: will it really deliver a justifiable benefit?

7.4.2 Storage density

Storage density is a measure of how much data is able to be stored in a given space. In the past 40 years, magnetic storage devices have changed from modules the size of a washing-machine, to something smaller than a cigarette packet in the case of so-called 2.5 inch drives.

Meanwhile the capacity of these systems has grown from perhaps 10 Megabytes to a point approaching up to 10 Terabytes. Solid state memories are likely to take this an order of magnitude further; indeed, SSD units typically started out following the same form-factor as 2.5 inch HDDs, but even this is already being surpassed by newer, even more compact, SSD design styles. Figure 7.5 shows examples of recent HDD and SSD modules for size comparison.

[111] A little care is needed here however. If most of the SSD is occupied by static file content, in other words files that are rarely/never updated, then any other frequently changing files will focus their wear effects on a smaller portion of the SSD, causing wear-out to occur more quickly. Fortunately, this is a recognised problem and 'static wear-levelling' has been developed to counter this by occasionally moving static file content around the SSD sectors, to rebalance the wear effects. This is somewhat analogous to rotating tyre locations on a vehicle, to even out tread wear.



Figure 7.5: Modern HDD and SSD units compared. Showing a 3.5 inch 80GB HDD (left) a 2.5 inch 500GB HDD (middle), and a 500GB SSD (right).

Drive	Sustained Read Mbytes/Sec	Sustained Write Mbytes/Sec	Price
CRUCIAL MX500 1TB SSD	483	429	£95
CRUCIAL P1 3D NVMe PCIe M.2 1TB SSD	1419	1338	£99
TRANSCEND SSD370 1TB SSD	473	392	£352
SEAGATE Barracuda 1TB HDD (2016)	170	146	£32
SEAGATE SkyHawk 1TB HDD(2016)	140	138	£48

Table 7.2: A few example HDD and SSD disk storage options (data as reported from hdd.userbench.com 2019)

7.4.3 Buyer beware

When considering suitable components for a system, it is important to realise that not all components are created equal. Performance and prices vary for components quite widely. This isn't just a question of up-front cost, speed, and capacity, but might impact upon power consumption, and reliability.

This is certainly the case for storage HDDs and SSDs. A few examples are given in Table 7.2. Although these drives may be of different release dates, this serves to illustrate that performance is not always directly linked to price. There may be other reasons why a drive with apparently lower read/write performance is more expensive. The key thing is to ensure that, when specifying system components, plenty of effort is put into researching the options. Small increases in price might yield big gains for system performance, and indirectly that may actually mean the deployment of that system is more cost effective.

7.5 Storage reliability and fault tolerance

When it comes to disk data storage, there are many applications where data integrity is critically important. This could include such applications as financial record keeping, medical data scans, safety critical systems, and so on. In order to avoid errors in data storage, systems have been developed to deal with these situations.

There are actually several faults that may arise, which would be undesirable. The first is a **mechanical failure** of an HDD during use, where the HDD just stops working altogether, and thus makes all of its data unavailable. Partial recovery of data on such a disk is possible, with the assistance of a specialist, but it cannot be done in timescales of hours, more likely days or weeks. SSDs do not have mechanical parts, and therefore offer one way to avoid such problems. That is not to say that an SSD is entirely failure proof however.

The second scenario is **file corruption**, caused by a number of things, and all are generally unpredictable.

Typically, the probability of bit errors in read operations on disk units is vanishing small - of the order of one-hundred trillion to one according to some drive manufacturers. For most applications, almost too low to be a major concern, though not if your HDD is controlling a nuclear power station perhaps.... Because there are different levels of severity of consequences, there are also different levels of dealing with these problems:

For simple bit errors in the storage system, data blocks can be stored with additional information, called an error correction code, or ECC^[112]. This can help to identify errors, and in some cases correct them. At the very least, the software will be able to know that the file it is reading is corrupted, and take appropriate action.

For disk unit failures, we can use **redundant storage arrays**. In simple terms, two or more disks can store the same data, and if one fails, another is still available. However, a less costly option may be just to frequently back up the HDD content so that a recent version of the data is always recoverable. This very much depends upon how quickly it is required to get back to an operational state: days, hours, or minutes?

For high degrees of data integrity, we can use more complex disk system configurations, involving duplicate resources and data storage. The **RAID** disk system (redundant array of independent disks) provides a number of different versions of disk duplication (redundancy) to provide

[112] There are a variety of methods, CRC: cyclic redundancy check, for instance. Of course these are not cost free - the codes use some of the available storage space, making files larger by a degree that is related to how substantial the ECC scheme is. There is also a small cost in terms of CPU or disk-controller effort and time required to perform the ECC functions.

for various scenarios and levels of resilience against data loss. RAID can also be used to improve performance, by combining multiple disks to make them appear like one faster disk. The full range of possibilities defined by the RAID standards (such as RAID0, RAID1, RAID2 ...) are rather complicated, and best left to the motivated reader to investigate. A couple of brief examples are worthwhile however:

RAID0: simply combines multiple drives, using a technique known as **disk striping**, to appear like one faster disk (because they can be read in parallel to multiply the sustained data read rate). The downside of this is that there are more drives to fail, so failure rate may be higher: we will see more about this in Section 14.8.

RAID1: uses **disk mirroring**. In this case, two or more drives have identical content. The advantage is that (a) if one drive fails, the others remain accessible with the same data, and (b) the drive that can read the data first will always deliver the data ahead of the others, and therefore there is a probability that average access time of the whole RAID system will be reduced, meaning faster data access and improved average data transfer rates.

So, we have observed that where data integrity is important, there are methods available to improve data resilience, and these systems can (at a higher cost) be quite complex and sophisticated in their methods of preventing data loss.

7.6 Remote storage

Another form of data storage that is increasingly popular is the idea of **remote storage**. To fully understand this concept we will of course need to understand the idea of networks, which we will examine in more detail in Chapter 13.

Let us assume for now that a typically computer system has a network connection of some kind, allowing the computer to communicate with the internet. Somewhere else in the world, another computer also has a network connection to the internet. Consequently, the two computers can communicate and share data if they wish to do so.

This brings about the concept of **remote data storage**, including the idea of **cloud storage** and **cloud computing**. Cloud computing is another way of saying that a computational task is performed on some remote computer (the **compute server**), usually for a fee, and the results

will subsequently be sent for local display on the user's own computer (the **client machine**). Cloud data storage is simply a variation of this where the server is primarily providing stored data on request rather than computed results.

Modern operating systems support the idea of a **network drive** or remote drive, and allow users to view files that are stored on a remote server, and in a way which looks almost the same as a local set of files. The user may see little or no difference, and may not even be aware that their files are stored remotely.

With the correct connectivity to the internet, the delay in sending a request to a server via a network connection can be of the order of milliseconds, and perhaps much less for local office networks. The additional time involved in accessing a file on a remote server (**server latency**) is often small, provided the data transfer rates of the network are also high. This works particularly well for relatively small files where transfer time is not noticeably long, or for data that needs to be accessed in a streamed^[113] fashion such as video, where all of the data doesn't need to arrive as a whole in a short period.

Increasingly, **enterprise systems** - that is systems deployed across large organisations - are based upon servers and cloud data and cloud computing resources. It is often cheaper to lease time on servers than to buy and maintain systems in-house. All of the benefits of rapid response to failures, automatic data back-up, and the ability to suddenly increase capacity for short periods, or to scale up as a business grows, make these solutions highly desirable.

7.7 Summary

In this chapter we have reviewed the evolution of data storage devices, from early primitive systems, to modern disk based HDD equivalents, and finally through to the latest solid-state memory-based mass storage devices such as the SSD. We have highlighted the performance characteristics of these devices, and explained how these parameters relate to the mechanisms of their systems, and understood the implications for system performance.

Meanwhile, we explored the idea of fragmentation, and data organisation on disk, and the impact this has on performance, along with techniques such as disk caching to hide these effects and boost performance.

[113] File streaming, particularly for audio or video, involves sending a file across a network in sections, which are collected and buffered at the receiving computer. When done efficiently, the data arrives just quickly enough to keep the buffer modestly full of data, and therefore not using more network bandwidth than immediately necessary. When data doesn't arrive quickly enough, video or audio will freeze momentarily, which can happen when a network is being heavily used by too many users at once.

Looking toward the future, SSD technology is beginning to make significant inroads into general system design, and in high performance computing, if used carefully in context. The wear characteristics of SSDs, and their capacity versus cost, will undoubtedly continue to improve, and it is possible that HDD technology will be phased out in all but some highly specialised applications in the next ten years. Who can guess what the next storage technology will be? It is probable that SSD will not be the last step in the storage revolution.

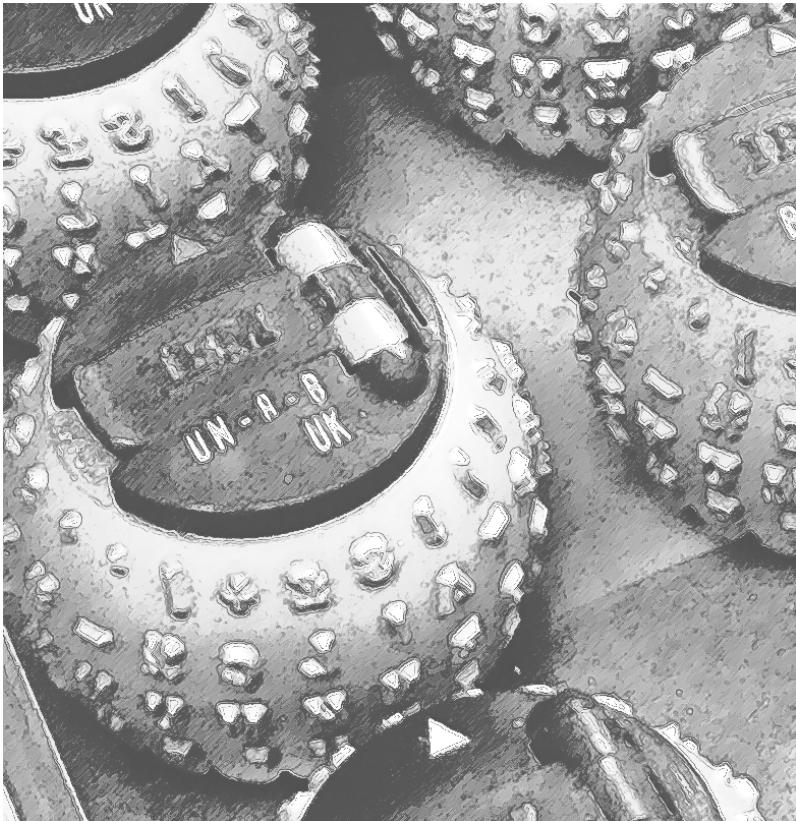
7.8 Terminology introduced in this chapter

Armature	Backing store
Client machine	Cloud computing
Cloud storage	Compute server
Contiguous file	Controller delay
Data transfer time	Defragmentation
Disk	Disk mirroring
Disk striping	Disk track
Enterprise system	Error correction code (HDD)
File access time	File corruption
Flash memory	Fragmentation
Hard disk drive	HDD
Interleaved seek	Magnetic disk
Magnetic recording head	Magnetic tape
Network drive	Paper tape
Platter	Punched card
RAID	Read/write head
Rotational latency	Sector
Sector size	Sector wastage
Seek time	Server latency
Solid state drive	Spooling
Stepper motor	Storage density
Sustainable data rate	Sustained transfer rate (disk)
Wear-levelling	Zoned bit recording

These terms are defined in the glossary, Appendix A.1.

8

A World of Peripherals



Golf-Ball Print head.

8.1	Everyday peripherals	150
	Input devices	150
	Output devices	152
8.2	Novel peripheral interfaces	155
8.3	Networking connectivity	156
	Wired networks	156
	Wireless networks	158
	Network protocols and overheads	159
8.4	Summary	162
8.5	Terminology introduced in this chapter	163

8.1 Everyday peripherals

Peripherals are devices that plug into the main computer system to extend its capabilities in specific ways. Many peripherals can be classified as input devices or output devices, though as we will observe later, the lines are becoming blurred as new variations of peripherals are being developed.

A very typical system may be found to be similar to that shown in Figure 8.1. We can see that even at this level, a computer system can have multiple types of connectivity to communicate with a variety of external devices. Some key peripherals are listed in the following subsections.

8.1.1 Input devices

Input peripherals are usually designed to support user interaction, either information input or control over computer functionality and behaviour. Some common input devices are listed in Table 8.1 which shows common devices, interface standards widely used at present, and likely data rates of such a device.

Keyboard: Almost everyone is familiar with this peripheral. Keys are wired into a signal matrix, a little like a row and column grid, and every time a key is pressed, a unique combination of row and column data is generated. A decoder chip can then work out which unique key has been pressed. Some keyboards also include indicator lights and other features, technically making them output devices too. Data rates for keyboards are very low: if you could type twenty characters per second (cps), then this is potentially of similar magnitude to how many bytes are being transferred to the computer per second too^[115].

Mouse: The computer mouse was initially invented to allow movement of an on-screen text cursor, and later became indispensable for on-screen pointer/cursor movement in graphically based operating systems. Mice use a variety of techniques to detect movement on a desk or mat, sometimes mechanical, sometimes optical. Every movement of a mouse is broken down into many much smaller measurements of movement, and each of these results in data transmission to the computer. So a mouse may be sending 100's of bytes per second for a single typical mouse movement event.

[115] In practice, whilst the operating system may appear to present convenient forms of keyboard input such as one-byte **ASCII** character codes, at a lower level most keyboards rely upon **scancodes**, which can be one to three bytes per keyboard event typically. There are a number of scancode 'sets' and these offer much more information than simply indicating which character is typed after a key is pressed.

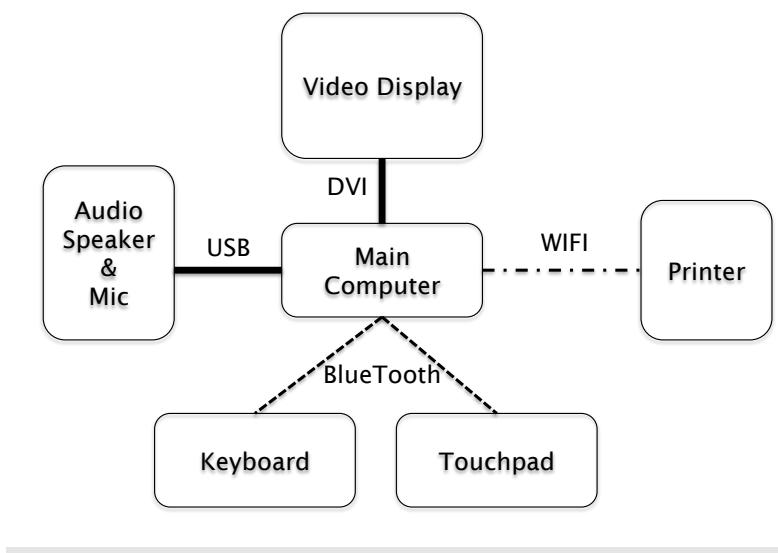


Figure 8.1: A System with typical Peripherals and common connection methods. Different peripherals may use different connection methods.

Touchpad/Trackpad: The touchpad is a touch-sensitive surface, somewhere around the size of a drinks coaster, which detects the motion of a finger or fingers, and translates these into mouse-like motion data streams, which the computer software treats as mouse motion. This is an alternative method of controlling on-screen cursors and pointers.

Touchpads permit more sophisticated inputs than a standard mouse. For example, two fingers can be used to indicate rotation, zoom, and so on (**multi-touch interface**). Users of APPLE Macintosh computers, and some smartphones, will be familiar with this concept.

TouchScreen: A touch screen is really just a touchpad capability overlaid on top of a visible screen panel, whereby the touchpad capability is transparent and does not obscure the view of the screen. Users can interact directly with screen content in this system, which is currently used heavily in tablet computers and smartphones. Generally, this capability is integrated into the device rather than being a true peripheral.

Tablet Stylus: Similar to the concept of mousepad and touch-screen, a stylus (a pen-like device) can be used to interact with the surface of the device. Where a mouse-pad technology is used this is normally called a graphics tablet, where a touchscreen is used it is just a tablet computer used with a stylus.

Microphone: Microphones are used for various purposes, including recording audio for audio production purposes. However, with suitable software,

Table 8.1: Standard Input Devices

Device	Typical Interfaces	Typical data rates
Keyboard	USB/Bluetooth	very low data rates
Mouse	USB/Bluetooth	very low data rates
Touchpad	USB/Bluetooth	very low data rates
Tablet Stylus	USB/Bluetooth	very low data rates
Microphone	USB/Bluetooth	typical data rates 24/48/96 Kilobyte/sec
Camera	USB/Bluetooth/WiFi	10Kilobyte/sec – 10Megabyte/sec
TouchScreen	Integrated	Relatively low data rates.

[116] Video-Conferencing is a process by which two way (or more) communication of audio and video can be established between parties on different computers.

[117] Examples include reading QR codes and barcodes for example.

they are also used for detecting voice commands, for audio dictation to text, and for audio connectivity during video-conferencing^[116].

Camera: As with audio, the obvious use is for capturing pictures or video for production purposes, but similarly, in a typical user context, a camera can be used for still-frame capture, video-conferencing, and potentially for activities such as motion tracking, gesture recognition, face recognition, and a few other applications^[117]. In embedded applications, cameras have a much wider variety of uses. Recent advances in camera technology have resulted in 3D image capture systems, an example of which is the Kinect system used in gaming systems, but having a variety of other applications.

8.1.2 Output devices.

Output devices are a little less numerous, since the primary human communication mediums are vision and sound. We could add touch, taste, and smell, and indeed, experimental systems exist for all of these as human-computer interfaces. However, they are not yet widely used. Examples of more traditional output devices are listed in Table 8.2.

The major output devices are video, audio, and print. They have different requirements in terms of data volumes, and formats. It is also the case that print content, video, and audio can be quite demanding on file storage space, and techniques have been developed to deal with this, as we will see later.

Table 8.2: Standard Output Devices.

Device	Typical Interfaces	Typical data rates
Video Display	VGA,DVI,HDMI	very high data rates
Video Projector	VGA,DVI,HDMI	very high data rates
Printer	USB/Bluetooth/WiFi	Large data quantities (may/not be fast)
Audio card	USB/Bluetooth	Similar to microphones for simple cases, perhaps 5-10 times more for complex audio such as Dolby 5.1
3D printer	USB	Medium Data rates

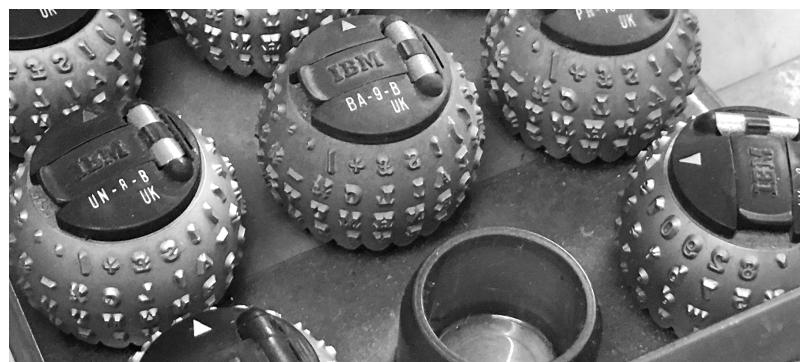
Video: Video displays these days are primarily based upon flat-panel display systems, very close cousins of flat-panel TVs. The variety of technologies for flat-panel displays is ever expanding. At the time of writing (2019) there are multiple coexisting technologies, LCD, TFT, OLED, QLED, each with particular capabilities. All of these systems use the concept of generating minute spots of colour (pixels) to build large scale images. Each pixel can have 24-bits of data associated with it, sometimes more. Therefore, a single frame of a video image might contain perhaps 1000×800 pixels, at 3 bytes per pixel = 2.4 million bytes of data. At 50 frames per second, data rates could easily approach around 120 million bytes/sec, and ultra-high definition images might contain five or ten times as many pixels. As a result, video displays require specialised connectors such as HDMI.

Video projectors: essentially similar in functionality to a flat-panel display, but images are projected via lenses onto surfaces, such as walls, screens, etc. Technologies for generating projected images include shining light through a flat-panel display matrix, or using arrays of micro-mirrors (**Micro-Mirror Array**) to create projected pixels individually.

Video Cards and GPUs: It may occur to the reader that generating such large volumes of video data at such high data rates is a rather demanding requirement for a computer system. One method of dealing with this is to move the burden to a separate processor, known as a GPU (graphical processing unit) which not only manages image generation but has many extra features to accelerate graphics creation, such as line drawing, shapes and shading algorithms, etc, all built into hardware circuits.

A standard CPU running old-school graphics routines simply could not manage the performance required for this task, and also do everything

Figure 8.2: Print heads from a golf-ball printer, which used a typeface mounted on a spherical print head. To change fonts, the operator had to manually change the print head with another one in the set. (photo with thanks to Jim Austin Computer Museum.)



else the CPU is needed for. When GPU's are mounted on plug-in modules they are referred to as graphics cards or GPU cards. Because these have very high demands on system resources, they often have their own dedicated connection to the system and CPU (the advanced graphics port: AGP, which we encountered briefly in Section 6.2).

Printers: Printers allow us to generate hard-copy or print-on-paper output. They operate in several ways. One method, the laser-printer, uses a laser to scan a drum in order to create an image using powdered ink (toner); another method uses a movable head to scan a page and spray dots of ink onto paper (ink-jet). There are other, older technologies, which we will not explore further here, but you may be interested in reading about pen-plotters, dot-matrix, and line printers if you like really old technologies. An example of a now fairly obscure printing system is shown in Figure 8.2.

Both laser and ink-jet printers have evolved to the point where they can generate very high quality monochrome or colour images, at relatively low cost. Laser printers are seen as the top-level for quality in most general purpose applications, though highly specialised printers for artwork may use very refined ink-jet technologies.

As for video images, a page, just like a video frame, is made up of millions of pixels (in print we refer to dots). A high quality image contains 600x600 dots-per-inch. An A4 page might contain over 34 million coloured dots. In order to represent image content, data is usually transferred in a format known as a PDF (Portable Document Format) file or a PS (postscript) file. Due to the way these files are defined, often a printed page will be much smaller than many tens of megabytes. After all, most printed pages are largely filled with 'white space' and only a fraction of the page is ink.

Since printers are connected via USB, Bluetooth, WiFi or direct Ethernet connection, then the process of printing usually involves transferring a PDF or PS file to the printer via that connection. This is known as spooling. In organisations where printers are shared between users, there may be a central printer management application, called a print spooler, which manages these jobs in a particular way, perhaps managing priority, permission to use various formats such as colour, payment logging, quotas, and so on.

Audio: Audio generation is another task that is generally left to a separate chip, usually an audio DSP (Digital Signal Processor). In the simplest form, audio can be monophonic, and have only one channel, and have data rates of a few thousand bytes per second. More typically, CD-quality audio requires data rates of around 84 Kilobytes/sec.^[118] However, this is the data rate needed for the final stage of audio generation and the file containing the stored audio representation can make use of various techniques to reduce the size of the file without any noticeable audio quality impact.

Like the white space on a page, there are elements within audio that can be removed without any noticeable difference. One of the most successful of these techniques is the **MP3 (MPEG audio Layer 3)** file format, which can easily compress audio by a factor of 10 or 20, making storage of the audio file 10 or 20 times smaller than the raw (uncompressed) audio data. Audio cards are nowhere near as sophisticated as GPU's. However, they can still do sophisticated things like filtering, audio effects, encoding and decoding mp3 in hardware, and so on.

Typically, an audio card will plug into a PCI bus as it does not have huge data transfer requirements, and some audio modules can even operate via USB or Bluetooth quite satisfactorily.

[118] Standard CD audio has a sample rate of 44100 samples per second, where each sample is 16 bits (2 bytes). However, professional recording systems often use 96 Kilosamples/sec per channel and 24-bits per sample.

8.2 Novel peripheral interfaces

There are several human-computer interaction modes that are as yet still more experimental than mainstream, but will no doubt be more prevalent in computing in the future. It is therefore worth briefly highlighting a few of these.

Haptics is the domain of human-computer interaction involving touch sense. We have already identified a few examples of this: a mouse, and a touchscreen, for example. However, there are others.

The dataglove is one technology that is emerging, particularly for virtual reality applications. This is, as it suggests, a glove worn on the hand but able to detect motions and joint positions as they move, allowing a user's hands to be projected into a 3-dimensional model of space, usually supported with a VR headset.

Another model is to use video tracking of limbs, hands, fingers, etc, to track motions, and gestures. This relies upon a number of technologies, both 2D and 3D camera technology, mathematical algorithms and artificial intelligence.

In order to make these systems more interactive, haptic feedback systems have been developed. The simplest case is a gaming handset with vibration to simulate some form of event-related feedback. However, more sophisticated concepts have been developed, including pressure-based sensation, or ultrasonic stimulation, to mimic the sensation of a finger in free space actually being in contact with an object that is not really there. When combined with vision and sound, these can be very convincing.

8.3 Networking connectivity

Network connectivity is an almost essential component of a modern computer system. The vast majority of computers in use today, with the exception of deeply embedded single board computers, have some form of internet or local network connectivity. Even fancy lightbulbs can be controlled by a smartphone app these days, and this is achieved via networks too.

Network connectivity is achieved by a network interface component, often a plug-in device, or a chip built into the computer motherboard. There are also different classes of network connectivity:

- ▶ Wired network connections, via cables and sockets,
- ▶ Wireless connectivity via WiFi^[119] technologies.

[119] WiFi is actually a trademark, and doesn't really stand for anything technical.

8.3.1 Wired networks

By far the most common wired network connection is **Ethernet**, a standard developed to permit many computer systems to connect to a single shared network cable (in effect, Ethernet is like a bus system, at least in the simplest modes of operation).

Another system, **ATM (Asynchronous Transfer Mode)** is less popular but used in certain scenarios. We will not go into deep technical detail about how these systems differ. More about networks will be covered in later chapters. However, Ethernet uses the concept of competitive use of the shared Ethernet bus, such that devices may end up wanting the bus at the same time, and potentially making service levels unpredictable. ATM reserves a fixed time-slice of the network bus for each device, guaranteeing quality of service (often referred to as QOS).

An Ethernet controller is effectively a chip or digital module of a chip, which performs the Ethernet data transfer protocol. This controller may even be integrated into a processor chip to minimise extra chips on a circuit board and possibly achieve tighter coupling of Ethernet to system memory. This is important where (a) CPU effort dealing with networks needs to be minimal to save power, or (b) where data transfer between memory and network is expected to be very high.

Ethernet controllers utilise a technique known as **Carrier Sense Multiple Access / Collision Detection, (CSMA/CD)**. In this system, devices can start to transmit at any time, without any central control. Of course this results in the possibility of **collisions** (several devices attempting to transmit at the same time). To minimise this, the CSMA/CD protocol is used as follows:

- ▶ Any device seeking to use the Ethernet bus must first check to see if the bus is already in use (this is the carrier sensing part).
- ▶ If the Ethernet bus is not busy, then any number of devices can attempt to transmit (multiple access).
- ▶ During transmission, the transmitting devices monitor the state of the bus to check that it is correct. If it is not, then this indicates that another device tried to transmit at just the same moment, and this is known as collision detection.
- ▶ If a collision is detected, all devices attempting to transmit will stop and wait a random amount of time before trying again (known as back-off-and-retry).

You may notice that this system has some similarities with I2C, which we examined in Chapter 6.9.1. This approach is quite common where there is no centralised arbiter of bus access. The negative side of this system is that there is no guarantee that a device will ever get to transmit its data. It could randomly back off and fail at each new attempt, for an indefinite period (the odds are minute, but they are never zero). On this basis, standard Ethernet is not considered predictable in its behaviour

(it is non-deterministic), and might be inappropriate in a safety critical system, unless a suitably modified standard is used.

Data transfer rates are clearly going to be a key concern for networks. On wired networks these are limited by the physical capabilities of the cabling system, the length of the cables and how many devices are competing for network bandwidth. Ethernet standards vary from 10Mbit/sec to multi-giga-bit/sec data rates, and there are many variants of the basic definition, each with different cable technologies, applications, and constraints (literally tens, if not hundreds, of variations).

8.3.2 Wireless networks

Wireless networks, sometimes referred to as **WiFi** networks, use radio frequency data transmissions in place of physical wires. This has a number of advantages, primarily the removal of the need for cabling infrastructure in a building, and the ability to mobilise devices such as laptops, smartphones, etc without having to plug them into a network in order to access internet or local network functionality. If there is a downside, it is that radio signals are prone to interference, and propagation of signals in buildings to give good network availability can sometimes be a problem.^[120]

Wireless Ethernet is simply a variation of Ethernet that permits use of radio channels in place of wires, and shares much of the same functionality. We will be familiar with this in our homes and perhaps our offices, where broadband routers allow a wireless network to exist in that space, and connect that space to the internet. These systems typically work well over tens of metres, but can be boosted and extended in various ways, in a large office for example.

Another popular wireless data networking standard is **Bluetooth**. Bluetooth is generally used over short distances, a few metres being typical. Many peripherals can operate using Bluetooth: keyboard and mouse for example can connect to a computer system using Bluetooth wireless links. However, Bluetooth is not designed for high bandwidth in the same class as wireless Ethernet, and where devices such as shared data servers, laser printers, and so on are being connected to a system, it is wireless Ethernet that is the primary standard of choice.

As with Ethernet, there are many Bluetooth standards, some designed for long distances (100 metres perhaps, though these are rarely used) and others for low power and relatively low data rates (e.g. 2 Megabits/sec, 5-10 metres). Some Bluetooth specifications allow data transfer rates

[120] Equally, the unconstrained availability of WiFi signals is also a security risk. Third parties can detect WiFi signals and attempt to eavesdrop on data content.

of 24 Megabits/sec, though others are much lower, and also dependant upon the distance between the two devices (data rates are lowered when the link is too distant for a particular data rate to operate reliably^[121]).

Because of the lower data rates, compared to wireless Ethernet, and the design of the radio protocol, Bluetooth can operate a relatively lower power. This makes Bluetooth attractive for mobile devices and devices powered by batteries (e.g. smartphones, wireless keyboards, and personal wearable devices).

[121] Although systems can detect bit errors in transmissions, if this results in a block of data having to be retransmitted, then the available bandwidth reduces because some of it is wasted with duplicate data. It may be more efficient to reduce the data rate and have fewer errors.

8.3.3 Network protocols and overheads

Although the raw electrical behaviour and frequency response of a network connection, be it wired or wireless, will dictate the maximum possible bit rate of that physical link, in practice, this is rarely achievable.

Firstly, the low-level hardware requirements of a network, such as CS-MA/CD mechanism may introduce the need for additional bits or clock cycles to be incorporated into the transmission of data. Secondly, the transmission of data may be packaged into distinct portions, typically referred to as packets. Such packets may include an information block, known as a packet header, as well as the actual data of interest, known as the payload.

Consequently, if a particular network requires a header of m bits, then transmitting n bits of data will require $m+n$ transmission bit periods. We can define the data transfer efficiency of a network protocol as follows:

Definition 8.3.1 Data Transfer Efficiency

$$\text{Efficiency} = \frac{\text{PayloadSize}}{\text{PacketSize}}$$

Where $\text{PacketSize} = \text{PayloadSize} + \text{HeaderSize}$.

So, as an example, if a network had a payload of 128 bytes, and a header of 8 bytes, the packet size will be 136 bytes for a payload of 128 bytes. Then, its efficiency will be 94%, since $128/136 = 0.94 = 94\%$.

Now let us take Ethernet as an example. The most common Ethernet format, 'Frame Type-II', has the format parameters outlined in the following summary detail.

Ethernet Frame Type-II info

► **HEADER**

6 bytes MAC Source address
6 bytes MAC Destination address
2 bytes Control Field

► **PAYOUT**

Min 46 bytes, Max 1500 bytes

► **CRC**

4 byte CRC error checksum

Now, in this case, there are 18 bytes of protocol information (header and tail-end CRC values) alongside n bytes of payload data. The worst and best case efficiency would therefore be:

Ethernet example calculations

Worst Case : 46 payload bytes, 14 header bytes, and 4 CRC bytes,
packet size = 64
efficiency = $46/64 = 72\%$

Best Case: 1500 payload bytes, 14 header bytes, and 4 CRC bytes
packet size = 1518
efficiency = $1500 / 1518 = 98.8\%$

[122] Again, remember for transfer bandwidth we generally use decimal megabytes here.

[123] At least in a simple case. Total network latency might involve routers and other factors that accumulate multiple delay components.

What does this mean in practice? Well, a network with 1 Gigabit/sec raw bandwidth and a protocol yielding 72% efficiency can only transfer a maximum of 720 million bits/sec of actual data (686 Mbits/sec)^[122].

Whilst it may therefore seem desirable to have longer payloads, and higher efficiency, this also means that network devices have to wait longer for each opportunity to use the network due to the time taken for the currently active device to complete a packet transfer, known as transmission delay.^[123] Therefore, simply having very big payloads is not always a good system optimisation.

A useful analogy here is a crossroads with traffic lights. If the traffic switches between north-south to east-west once per minute, and the lights take 10 seconds to change, then the efficiency of traffic flow (the bandwidth) is $50/60 = 83\%$, and the maximum time a driver has to wait is 1 minute (the service latency).

We could decide to have the lights change every 10 minutes, and have a traffic flow efficiency of $590/600 = 98\%$, but one can imagine the drivers

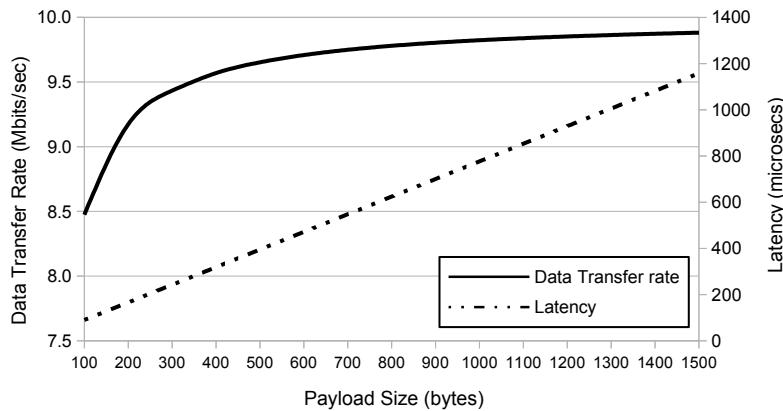


Figure 8.3: Network Efficiency and Latency (delay) tradeoff. Assumes a simple 10 Megabit raw network bandwidth, a header size of 18 bytes and a single network segment.

being rather annoyed at the 10 minute wait for each light change! The message here is that quality of service is a combination of bandwidth and latency.

Returning to our network scenario, we can attempt to represent this efficiency versus payload-size tradeoff via a graph. We can see in Figure 8.3 that as we increase payload size, efficiency of data transfer improves, approaching the maximum bandwidth, but at the same time, delay increases too.

In this example we can see that above about 500 bytes per payload, the transfer rate only improves marginally, but at the cost of rapidly increasing latency penalty. A payload size of between 500 and 1000 bytes would be a good compromise here if bandwidth was important but delay was also a secondary concern. In that case we would achieve nearly all of the peak data transfer rate potential of the system, but with much lower than maximum latency.

If, on the other hand, delay was more important than bandwidth, then a good compromise might be a payload size of no more than 500 bytes, as at this point bandwidth is not dropping off sharply but delay is much lower than the first case mentioned.

There are further subtleties here too. Consider the limitation that a packet cannot be used by a receiving processor until fully received. CRC cannot be verified until the whole packet is visible, for example. Instead of sending 1000 bytes in a single payload, sending 100 bytes ten times in a row might allow the CPU to work on the first 100 bytes whilst the next 100 is being sent (in a sense, a form of pipelining). This is even more

pertinent if a multi-core processor was assigning successively received packets to different CPU cores in turn for processing.

Likewise, rapidity of information updates may be more important than flow rate. In online gaming, players often complain of **lag**: they do something in the game but it responds too slowly, spoiling the feeling of real-time interactivity. Here, shorter packets will allow a finer grain of responsiveness even though total bandwidth is reduced.

8.4 Summary

Peripherals are, in some senses, the most important parts of the computer system. They allow us to customise its configuration, interact with the computer via input and output devices, and to produce video, audio and printed documents. However, the wide variety of devices, and the variations in the ways they connect to the computer system have resulted in many connectivity options being developed, be they wireless or wired connection, high speed busses, low speed point-to-point wires, and so on.

Although we have not explored in detail how peripherals work, for example how a mouse detects movement, we have explored how these devices demand certain degrees of data bandwidth to function sensibly, and how the connection choices and the underlying mechanisms of those connections (such as Ethernet protocol) will lead to performance constraints and tradeoffs that need careful design consideration.

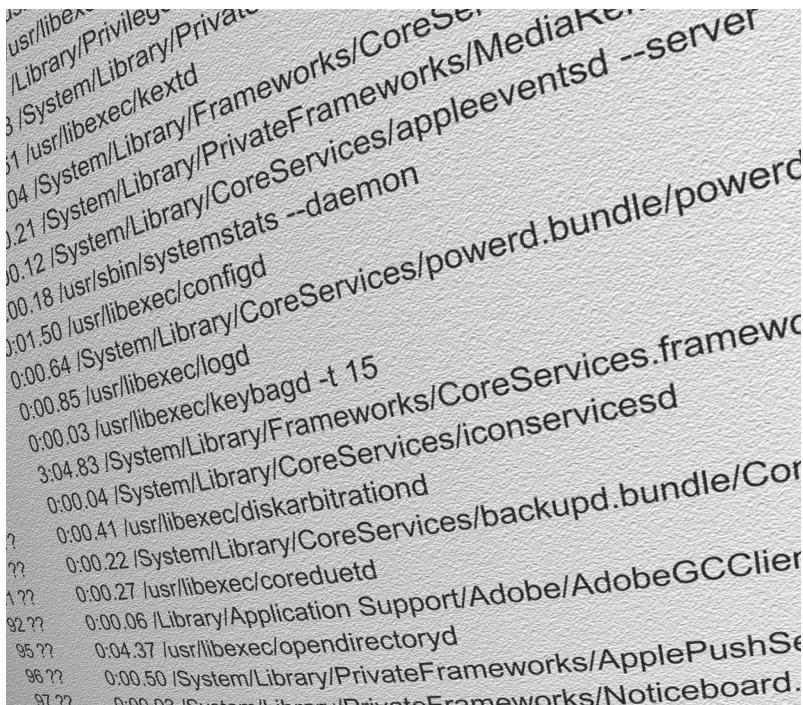
As with topics mentioned in previous chapters, things will continue to evolve. New connection standards will emerge every year or two for the foreseeable future, not least because new peripheral technologies continue to be developed (the maturing capabilities and decreasing costs of VR headsets is a good recent example, driving new sets of requirements). Will we recognise computer systems in twenty years time? They may be very different.

8.5 Terminology introduced in this chapter

ATM network	Bluetooth
CSMA/CD	Dots-per-inch
Ethernet	Golf-ball print head
Lag (network)	Micro-mirror Array
Multi-touch interface	Packet collision
Pixel	Tablet stylus
WiFi	Wireless network

These terms are defined in the glossary, Appendix A.1.

Operating Systems Concepts



9.1	The idea of an Operating System	166
	A 'bare-metal' system.	166
	BIOS	167
	Low-level input/output drivers	167
9.2	The operating system hierarchy	168
9.3	Choices, choices ...	170
9.4	The purpose of the kernel	172
	Abstraction	173
9.5	The OS as taskmaster	174
9.6	General vs specific, once again	176
9.7	Summary	178
9.8	Terminology introduced in this chapter	179

9.1 The idea of an Operating System

The topic of the operating system (OS) can be quite daunting for beginners. A very simple definition of an operating system might be:

Definition 9.1.1 *Operating System: Simplified Definition*

Operating System: a collection of software operating at different levels, to hide the true hardware structure from programs and users.

But why do we need this? Perhaps a good way to understand what an operating system does is to start with a computer system in its naked state, without any kind of operating system, and then see how far it can get toward a fully functional system. We may consequently understand how and why an operating system steps in to bridge the gap.

Let us start with a very simple scenario: suppose that we have a motherboard, we have installed a CPU, and DRAM, we have a hard disk unit, a keyboard, and a video display. Now, what happens when we plug in the power, and turn this system on? Let us explore what happens step by step, and then try to understand why an operating system is needed, and what it does.

9.1.1 A 'bare-metal' system.

At startup, the first thing that will happen is that the CPU will reset its program counter to zero, and then begin to read instructions from memory.

If the memory contains only DRAM or SRAM, which is volatile, then the content at this point will simply be random values- effectively, garbage. The CPU will start to fetch these values and try to execute them as instructions. The computer will crash almost immediately. This is what will happen in a **bare-metal system**^[125]. Without any software, the system cannot function. So what does this tell us?

Clearly the area of memory starting at location zero must have some permanent (non-volatile) memory content, containing valid instructions, otherwise the computer will never be able to function.

This was one of the fundamental reasons for operating systems to begin to be developed in the early days of computers. Whilst punched cards or paper tape could provide a program sequence to the computer, the computer first needed another program to allow it to read the tape. If

[125] The term 'bare-metal system' refers to a system without any supporting software, other than the code it was programmed specifically to execute - in other words, no operating system.

there was no other option, this code had to be entered manually when the machine was switched on. Naturally, this was replaced as soon as possible by some form of initial non-volatile program storage, even if only a few hundred bytes of memory.^[126] In a sense, this was the first echo of modern operating systems coming into existence.

9.1.2 BIOS

We can solve the problem of startup code requirements by using one of the ROM, PROM, EPROM, EEPROM, or other options we discussed in Chapter 5. The instructions found in that memory chip will tell the CPU what to do immediately after power is switched on. The most important early task in this instruction sequence will be to initialise any important registers or components in the system, to ensure that their behaviour is under some kind of normal CPU control. The next thing that this instruction sequence will do is most likely to be to perform a check for the existence of DRAM or SRAM. If present, it would then typically perform a power-on self-test on that memory, and any other key components, to make sure they are present and functioning as expected.

So, we have an important piece of program code, stored in a non-volatile memory, which the CPU starts to execute immediately after power-on, and this is known as **BIOS (Basic Input Output System)**. This is sometimes referred to as a **bootloader** or **bootstrap program**. BIOS not only performs some essential tasks as just described, but BIOS typically also provides the ability for a user to halt the startup procedure and then change various important system settings (if they know what they are doing).

Once the basic startup procedure has been completed, the BIOS will then look for a storage device, typically a hard-disk, in order to find out what it has to do next. Now we have another problem: how does the computer even know how to read a disk?

9.1.3 Low-level input/output drivers

The BIOS is usually provided with a capability to detect the existence of a disk unit. At this stage, our computer knows nothing about its higher level identity, or even what kind of disk is present. It must therefore talk to the disk unit and obtain enough information to then read information from the disk.

[126] The process of triggering the loading of the initial code from storage memory was known as bootstrapping. Hence the idea that we boot computers when we turn them on.

It is important at this point to understand a little more about disk storage (and SSD by extension). A disk unit is a physical system with storage capabilities. It is manufactured to be agnostic in terms of its end-use. A disk unit can be configured to operate with a variety of file systems and operating systems. The BIOS therefore has to first of all read the first block of data from the disk, known as the **boot sector**, and then decode that information, which then allows the BIOS to understand what that file system is. This is typically achieved by the boot sector containing a short program, which is executed by the BIOS.

Hopefully at this stage, our BIOS will be able to pass control over to this boot sector code, it begins to run, and this ends the involvement of BIOS in the initial boot procedure. Of course this is not the end of the whole system startup process, just the beginning. What happens next is usually that the boot sector code will begin to read the file system, looking for particular named files that control the more comprehensive operating startup procedures. These files are the root functionality of the operating system proper.

9.2 The operating system hierarchy

So at this stage, we have already encountered three levels of software in our system: the BIOS, the boot-sector code, and the as yet unexplored operating system files. It should start to be apparent that the whole software structure of the computer system is actually a number of parts or layers working together. We can visualise this as a block diagram, as shown in Figure 9.1.

There are a number of components to consider here in more detail:

Hardware: we are hopefully familiar with this, the base system containing all of the electronics and physical system components. The hardware might include components manufactured by different manufacturers, and these may affect how the operating system has to respond.

BIOS: the initial startup code that the CPU starts to run when power is switched on. This is the program that checks for attached storage devices and attempts to find a boot sector program.

Boot-Code: the program instructions stored in the boot-sector of the first device which the BIOS locates, and which contains one (there can be several in theory). This code allows the CPU to access the file system of the higher operating system.

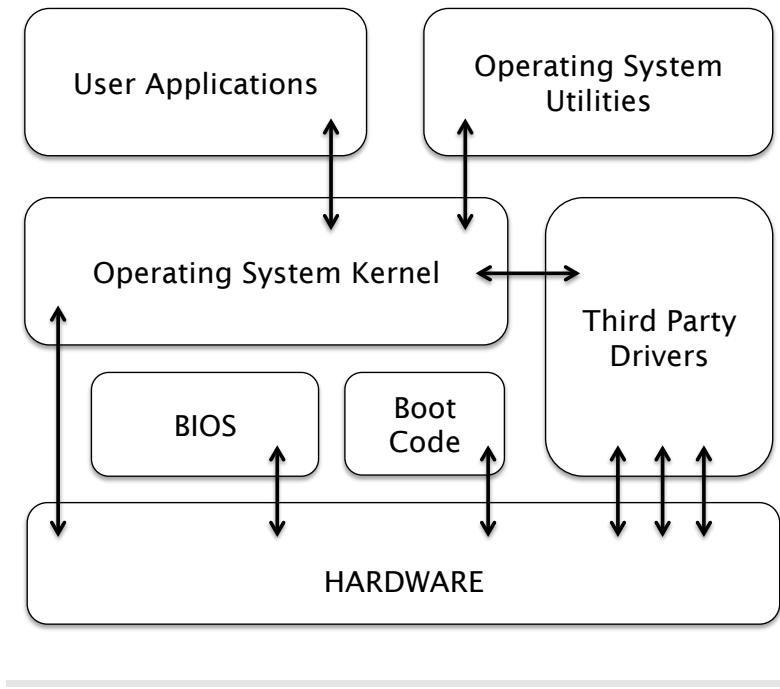


Figure 9.1: Operating System Hierarchy. Showing multiple layers of functionality with hardware being abstracted by the operating system, alongside third-party drivers. Thus, applications normally do not access hardware directly.

Operating System Kernel: the most important subset of the operating system, a collection of code modules that permit access to the main functionality of the computer system, such as keyboard, IO modules, and so on, but also includes specialist code functions to deal with issues such as how memory is used, how events are managed in time, how programs start and run, and what access any program has to the low-level machine. In a sense it can act as an overseer for everything else that wants to run on the system.

Third-Party Drivers: the operating system quite often contains a variety of auxiliary code modules, known as device drivers, some of which are selectively installed during the set-up of the computer system, and which help the operating system to communicate correctly with devices such as mouse, keyboard, network interface, disk unit, video display, and so on.

Because new devices come along frequently, it is not unusual for the device manufacturers to provide their own (third-party) device drivers, which may always be needed, or might eventually be replaced by drivers written by the operating system vendor.

Operating System Utilities: A collection of programs written by the

operating system vendor to provide useful capabilities, often relating to basic system configuration tasks, file management, and often providing basic convenient applications such as a text editor. Some operating systems provide a richer variety of utilities and applications that have a lot more functionality.

User Applications: These are either programs the user has written themselves to run on their machine, or purchased or obtained from an application vendor. Strictly speaking, they sit on top of the operating system. There are a huge variety of applications available, but all developed with some key things in common: They utilise the functionality of the operating system to coexist on the computer where they run, and typically access low-level hardware capabilities only via drivers and kernel functions managed by that operating system.

Most applications do not need to access hardware or low-level system functions directly, and indeed many operating systems attempt to ensure that this can never happen: we will find out more about how and why this is achieved later.

9.3 Choices, choices ...

In some ways, the world of computers would be so much easier if there was only one operating system. One OS that fits all needs, all of the time. Of course this is simply not the case, and for various reasons could never be achieved. There are some good reasons for this:

- ▶ The marketplace, and the diversity of proprietary systems: there are many operating systems because there are many developers.
- ▶ A counterbalance to proprietary developers are so-called open standards and open source platforms, where no one has commercial ownership of an OS.
- ▶ Historical divergences of systems models in the early days of computers.
- ▶ The need for efficiency, and application-specific versus general purpose operating system capabilities.

So, the reality is that there are many OS options available for a computer system to use, and for many reasons. Because of this, the problem of compatibility of software and file systems across platforms has always existed, and is a major issue.

It is also worth noting at this point that we can divide OS categories into two major differentiated standards of user interactivity:

Command line operating systems: use text-based commands, and rely entirely on textual interfaces to human users and operators. The first operating systems were based on this model, simply because there were no feasible alternatives. Very widely used examples of these include MS-DOS^[127] and Unix. Often the command line capability is referred to as the shell.^[128]

Graphical operating systems: use graphical representations of file systems and functionality to provide a graphically interactive user experience. Widely used examples of these include the MICROSOFT WINDOWS OS releases, Ubuntu, ANDROID, APPLE IOS, and numerous Linux^[129] based graphical desktops.

Graphical operating systems are hugely popular and increasingly sophisticated for the end-user due to the power of modern computer graphics systems, and the inherent simplicity of the user experience. A next step in this evolution has been the arrival of powerful and portable touch-screen and tablet computer systems.

Examples of these two cases are shown in Figure 9.2. It is worth highlighting that due to the way operating systems have evolved from command line and then into graphical systems, it is quite common for graphical operating systems to be built on top of command line OS kernels, and indeed it is possible to bring up a command line window on many graphical desktops and interact with the OS as a command-line system. A significant proportion of users never need to do this of course: if they are primarily only interested in running their third-party applications, then it is not necessary.

However, where a more direct interaction with the OS kernel is required, for example, for system administration and maintenance of software issues, this provides the route to doing so.

Another scenario where command line is advantageous is where a complex sequence of actions is required to be defined and repeated on demand. The use of a command line to create and run a shell script (a file containing a sequence of commands for execution in a shell) can be quite convenient.

For example, a script could be written to find all image files that have been on the computer disk for more than 7 days, and delete them. This could be run once a week to keep the hard disk system free of old files of that type and save storage space from being gradually eaten up.

[127] Microsoft Disk Operating System (MS-DOS).

[128] A shell or command line is actually a program in its own right: it is a way of parsing commands typed by the operator and then initiating subroutines or separate programs as requested. There are numerous different shells.

[129] Linux is effectively a version of a UNIX-like command-line operating system.

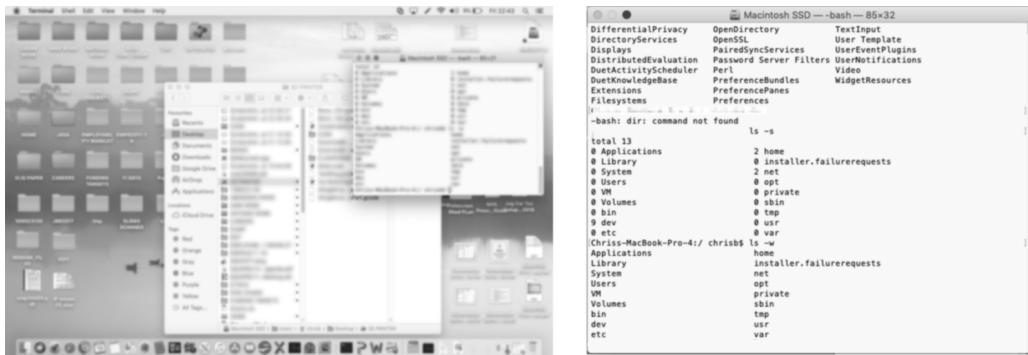


Figure 9.2: Examples of OS types. Showing graphical OS APPLE OSX (left) and command line (right).

Performing this in a windowed OS mode could be quite laborious and prone to errors.

9.4 The purpose of the kernel

As we have already mentioned, the kernel has a few very important tasks to perform. Before moving on to a more in-depth look at operating systems, let us consider what the kernel is responsible for in further detail:

Controls access to low-level hardware: The kernel provides program sequences (sometimes referred to as **functions**, **routines**, or **modules**) that permit low-level hardware capabilities of the system to be accessed by programs wishing to do so. There are several reasons for doing this:

- ▶ To provide one reliable way of doing so, rather than relying upon many software vendors reinventing the capability, and perhaps in erroneous ways.
- ▶ To ensure that programs can only access hardware when it is desirable, for security and reliability.

Manages the allocation of memory to programs wishing to run on the system: Ensuring that there is enough memory, that memory is allocated and released correctly by programs, that programs only access the memory they have permission for, and a few other issues, including the idea of virtual memory, which we have mentioned previously, and will return to later.

Manages the allocation of CPU time: Because many programs may wish to run, either one by one or side by side, the OS kernel will provide methods to ensure that this work is fairly allocated to available CPU resources.

Manages fault conditions gracefully: When something goes wrong, the kernel attempts to maintain stability in the system, by ensuring failing programs do not affect other programs also running.

Controls the rights to access functions of the system on a user by user basis: This allows system administrators more control over the OS than the general user, and separates the use of the system between users. Different users have different access rights and permissions. These restrictions typically also apply to any programs these users run.

Manages how the computer deals with IO devices that require a response from the computer system: This includes plug-and-play scenarios where devices may appear and disappear as they are presented or removed from the system whilst it is running (during 'run-time'). This also includes the need to manage the set of device drivers required at any one moment in time,. Devices indicate the need for responses by using interrupts. The kernel therefore manages the servicing of interrupts, which cause device drivers to interact with devices.

9.4.1 Abstraction

Together with device drivers, the kernel achieves a concept known as **abstraction**. This refers to the idea that accessing a specific resource, which might vary materially from one system to another, is done via the drivers or kernel acting as an intermediary. This makes all of those devices appear to behave in identical fashion. For example, from the programmer point of view, an SCSI hard disk, an SATA SSD, or a USB flash drive, may all look the same as part of the file system, even though the underlying hardware components are quite different, and are accessed by different routines within the kernel.

This process of abstraction is quite important because programs can be written without knowing precisely what the low-level behaviour of devices is. The kernel hides those differences and provides a known point of interface for software to interact with. Programmers can then write software that can be used widely across variations of a similar computer system design without having to write and test lots of versions of their software.

An example of abstraction in the real world is a doorbell. It is simply a button which you press: that is the abstraction. However, when you press the button, this could cause a buzzer to sound, a bell to ring, or a light to flash, and this could be done via a wire or perhaps a wifi signal. You only need to know one action in order to ring a doorbell, regardless of how many ways that button results in an outcome. Abstraction in software is there for the same principle.

9.5 The OS as taskmaster

A major responsibility of the OS kernel is to manage the running of programs on the system. As we have already noted, the OS sits on top of the lowest level code entities (BIOS, Boot Code), so it has no responsibility for them. However, drivers, utilities and applications, and also the various sub-components of the OS are managed by the OS kernel. We will see more about exactly what that management process entails in a later section. However, at this point it is helpful to introduce some terminology and basic concepts.

Whenever a software entity, such as a **user program**, is required to run, an OS will take responsibility for managing that program. We might ask why this is necessary? Why not just let the program run and finish off its own accord? The answer is that in modern computer systems, many programs can, and indeed must, run **concurrently** on the system in order for things to function correctly. We refer to these concurrent programs as **processes**; a looser term is **task**, which can refer to a variety of things not just including processes.

It is important to realise that concurrency (multiple events occurring simultaneously) is a fairly loose definition here. In truth, some computer systems may actually switch between processes at a very high speed, executing a little bit of each process in turn, in such a way that it appears that all of the processes are running together. This is known as **task-switching**.

In some other systems, where multiple processors or multicore processors are installed, multiple processes can run with true concurrency, in theory at least. An ideal scenario would be a separate CPU core for every process. However, this is generally not feasible, or efficient, as a modern system may actually be running hundreds of process-related program modules, some of which rarely do any work, but must be present. Task-switching allows the sharing of CPU resources to make the best of the available

CPU's at a given acceptable level of cost (8 CPUs may be affordable, 80 may not be). Where an OS maintains a policy to decide which task should be executed next and how much time it is allowed, this is known as **task-scheduling**.

A further level of hierarchy is typically present in the structure of programs. If a program is a **process**, then a process can also contain **threads**. Threads are effectively mini-programs that run concurrently as part of the same program. This is why a system may appear to have hundreds of tasks running, be they processes or threads within processes. And, of course, every one of these processes and threads requires allocation of CPU time, memory resources, and access to various hardware elements, all potentially managed via the OS kernel.

We can investigate what is going on in most operating systems with respect to processes, by making use of suitable utilities, either desktop or command line versions.

An example is given in Figure 9.3, which shows a portion of the complete process list for an APPLE-MacOS desktop scenario. We can see that there are numerous user applications running: MICROSOFT Word, MICROSOFT Powerpoint, Chrome Web Browser, Open Office, and others. Meanwhile another process 'kernel_task' is also running. Each process potentially has threads, the kernel_task has 149 all by itself, but there could be thousands. This corroborates the point made earlier: there could be literally hundreds of concurrent programs (or at least threads within processes), apparently running simultaneously. Many of the kernel-related threads and processes will however be quietly working away with low effort in the background. These are often related to services, such as network capabilities for example. Such special purpose tasks are sometimes referred to as **daemons** or **service daemons**.

Even within applications, some threads have minimal work to do. For instance, the autosave feature in a word processor only has to do one thing - every ten minutes, check if the user has changed anything, and if they have, then save the document to disk. We can safely say that this thread spends 99.9% of its time doing nothing, but it must remain present continuously whilst the word processor application process is running.

An inquisitive user can find out quite a lot about the workload the computer is managing with the use of such a tool, including how long a process has been running, how much memory it is using, how much CPU time it has used up, how much it is using at the present moment, how much disk activity and network activity a process is generating, and

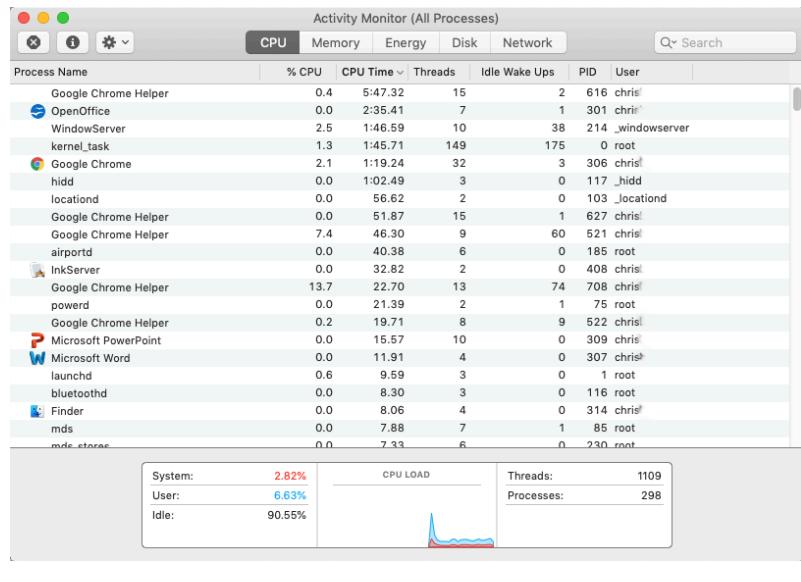


Figure 9.3: Example of a desktop OS Process monitoring tool. Example shows APPLE MOJAVE OS process monitor.

even the estimated energy being used. Where a system needs to be optimised for any of these performance requirements, this information can be quite useful to identify particularly wasteful and inefficient processes and perhaps find alternatives or configure them to behave differently.

As a result of the development of the concept of processes, threads, task-switching, and scheduling, a number of developments have been driven by these attributes of OS design, including **multi-threaded programming languages** (a good example of which is Java), **multi-user systems**, **virtualisation** of systems, and many other features, including related innovations in processor architecture and design. We will explore these further in later chapters.

9.6 General vs specific, once again ...

Just as we learned in earlier chapters that a computer system can be general purpose or application-specific, the same is true for operating systems. A large number of widely used operating systems are either general purpose or strongly leaning in that direction.

As an example, the ANDROID operating system, which is highly popular for mobile devices, including tablets, and smartphones, is clearly geared toward some of the constraints of those kind of systems, yet they support

a hugely varied range of user applications and uses. Here are a few examples of general and specific OS categories:

General-Purpose: Examples of such operating systems include Linux, MICROSOFT WINDOWS, Ubuntu, APPLE Mac OS variants. These systems aim to provide every reasonable capability for the general computer user: the home PC, the office desktop, the server-based computing environment, and so on. They are optimised to try to deliver the best performance across a range of uses.

Mobile: For smartphones, tablet computers, the most prevalent operating systems in use currently (2019) are APPLE IOS and ANDROID. Variations of MS WINDOWS for tablet PC's are also available. These OS versions, particularly ANDROID and IOS are heavily optimised for mobile devices, delivering smooth user interface experiences, saving battery power, minimising the OS code footprint (the size of its storage requirements) and providing efficient connectivity to mobile networks.

RTOS (Real-Time Operating Systems): These are somewhat more specialised toward deployment in specific applications, where the demands of an industrial or a control-system related problem are a priority. An example might be an operating system used in an automotive vehicle, or in a medical system, where safety features must be validated and legally proven to operate correctly under all possible operating conditions. Here there is a tight demand for ensuring that certain things happen within certain timescales. An emergency braking system cannot be allowed to be interrupted by another process that then behaves unpredictably, as this makes the braking system unpredictable too.

RTOS systems are very different from the general purpose OS standards we are used to on our desktops. They may lack many familiar features, simply because they are not relevant to that application (often an embedded system scenario), and also because providing those features would defeat the deterministic nature (i.e. predictability) of the system. These OS variants typically also have features within them to ensure that when faults occur, the system can always recover to a known-good state rapidly, and indeed within a known maximum timescale (e.g. a system may have to reset and reboot to an operational state within 1ms in a particular application).

Examples of RTOS include specialised versions of familiar systems such as RTLinux and 'WINDOWS Embedded', but also systems designed from more generic starting points such as OSE and QNX.

9.7 Summary

In this chapter we have attempted to gain a broad oversight of the diversity of aspects that are encompassed within the idea of an operating system.

In particular we followed a path from the basic low-level startup event of a power-on reset in a computer system, to the initial execution of the BIOS code, the boot-sector process, and then finally the loading of the operating system core - the kernel. Building upon this we have seen that an operating system is actually a symbiotic system of many software elements.

The role of the operating system as a task overseer and memory manager was highlighted. This is an area we will expand upon in the next chapter. At this point we have gained an appreciation of the concept of tasks, processes, threads, and the concept of an operating system managing these factors to deliver resources to applications with control, ensuring stability and security of resources.

We will see in the next few chapters that the operating system foundations can be built upon in terms of task management, memory organisation and management, and finally the role of the file system. We will then have a complete overview of the operating system and how it interacts with the underlying computer architecture we have introduced in earlier chapters.

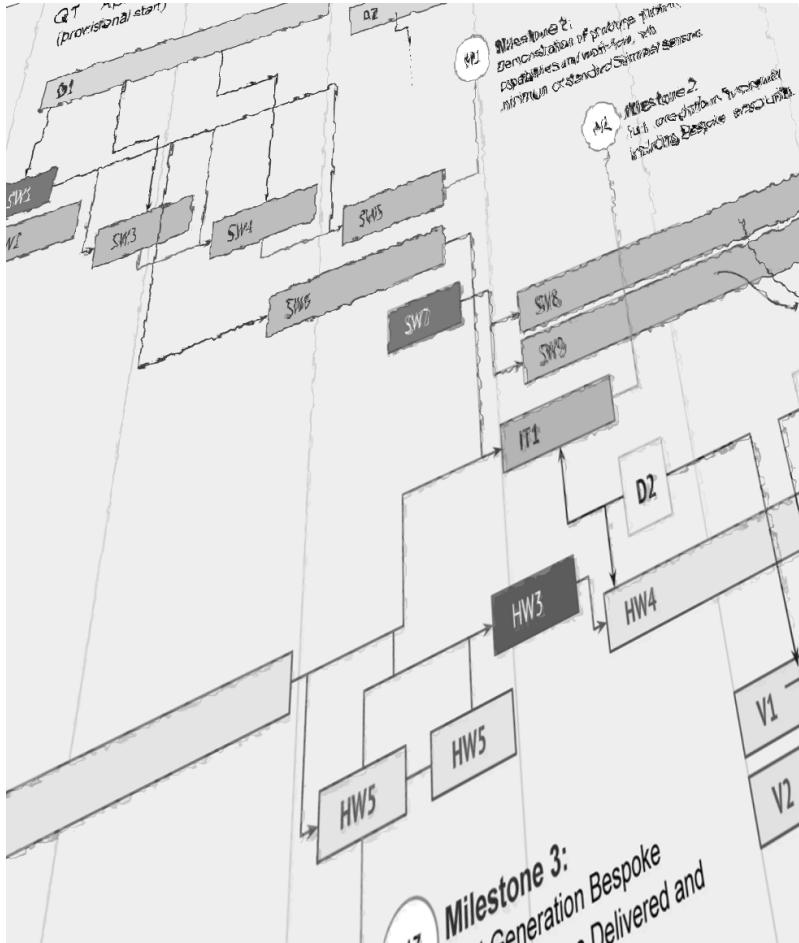
9.8 Terminology introduced in this chapter

Abstraction (OS)	Bare-metal system
BIOS	Boot sector
Bootloader	Bootstrap program
Command line	Daemon
Driver	Graphical operating system
Kernel	Multi-threaded program
Multi-user system	Operating system
Power-on reset	Process
Real-time operating system	RTOS
Service daemon	Task
Task-scheduling	Task-switching
Thread	Virtualisation

These terms are defined in the glossary, Appendix A.1.

10

Workload Management



10.1	The task in hand	182
10.2	Process ID and house-keeping	183
10.3	Thread management models	184
10.4	Task scheduling	185
10.5	Scheduling algorithms	186
	Task scheduling queues	188
10.6	Pre-emptive scheduling	188
	Multiple queues	190
	Interrupts and polling	191
10.7	Achieving priority	193
10.8	Advanced CPU architecture OS support	194
	From scalar to super-scalar	194
	Multicore	195
	Hyperthreading and SMT	196
10.9	Summary	199
10.10	Terminology introduced in this chapter	200

10.1 The task in hand

From previous sections, we have established that operating systems manage tasks, and that tasks can be classified as processes or threads. Some operating systems have their own terminology for these concepts which we will avoid for the moment. Instead, let us consider what these two types of tasks are allowed to do and what they are not.

Processes: are programs that run under control of the operating system, such that when a program begins, the OS allocates it a memory space that is private to that process. It also adds the program to a task-list, to allow it to be given regular intervals of CPU time to perform its activities.

One consequence of a process having its own private portion of memory space is that other processes cannot, in theory at least, access or alter that memory. This means that the concept of compartmentalisation of processes is achieved. This has many benefits, including privacy and security of data within a task, and resilience to being affected if another process crashes. Imagine, for example, if your banking app could be spied upon by another app simply by it looking at the other app's memory. A critical purpose of the operating system is to prevent this from happening.

Threads: Within a process there may be many threads. These are sub-modules, or mini-programs that each perform a purpose in their own right and which are required by the program designer to operate concurrently in order to perform a collective outcome associated with that process. Threads do not exist on their own, they must be created within a process.^[131]

[131] Threads are sometimes referred to as lightweight processes, but they are not processes in the true sense.

A thread example might be encountered in a process which reads data from a microphone into memory, whilst also copying the resulting audio data from memory to disk. In order to get smooth and uninterrupted recording, it may be necessary for the process to run the microphone function as one thread, and the disk writing function as a second thread. Then both threads get a slice of CPU time at frequent intervals, and if correctly set up, it will allow recording and disk transfer to happen as if concurrently. Additional threads can be introduced, for example one that monitors and updates a GUI, without upsetting the operation of the first two threads.

Importantly, because threads are like siblings within the main process, they all use the same memory resources allocated to that process on

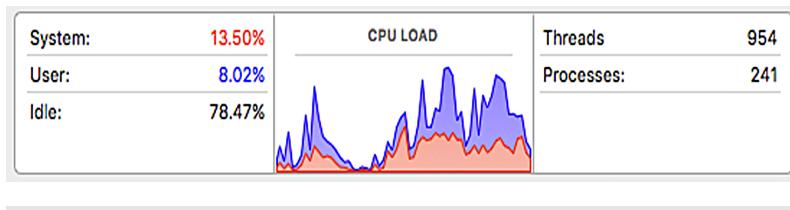


Figure 10.1: CPU workload process Monitor. This view shows a MAC OS performance monitor view, with both the kernel/system load (bottom trace) and the user load (upper trace).

equal terms. They can access blocks of memory within that process, and share information. If this happens inside a process then it is potentially advantageous, since it allows easy sharing of key data between threads. Generally, threads tend to work together on some level, therefore sharing data and memory accesses with the threads of their mutual encapsulating process makes sense.

In a typical desktop or laptop, a large number of processes may be running simultaneously, and each many have many associated threads. The work these processes and threads are doing can vary from one moment to another. This is observable in Figure 10.1 where we can see 241 processes are running, and 954 threads! This is all on a single laptop computer. Notice also that this monitoring tool displays system and user workloads as two distinct categories, and varying over a time window.^[132] In this case the OS kernel is using more CPU time than the user on average, but overall, the CPU is idle nearly 80% of the time in this example, suggesting it is not being very heavily used by the user at this particular point.

[132] In this case the time window was of the order of a few minutes.

10.2 Process ID and housekeeping

In order to keep track of processes as they execute, the operating system makes use of two important concepts. The first is the idea of a **process identifier**, or **PID**. This is simply a unique number that is assigned to any process when it is started. This means that every process on the system has a unique PID.

Once a PID is assigned, the operating system can then keep a more detailed record of the process's key information, known as a **process control block** or **PCB**. A PCB contains a variety of key information about its associated process, including:

- ▶ Process State (starting, ready, waiting, etc).
- ▶ Information about its parent process,
- ▶ Task priority and privileges,

- ▶ IO attribute related information,
- ▶ General info such as CPU state data, held whilst the task is suspended.

Together, these pieces of information allow a special task management function, known as a **scheduler**, to manage each process according to the rules and configurations of the operating system, and the particular task attributes assigned to the process and its owner.^[133]

[133] A process could belong to one of several users of the same system, or different login identities operated by the same user.

10.3 Thread management models

Although threads belong to processes, there are several models under which a thread can be created and managed. Figure 10.2 illustrates these two models, which can be described as follows:

User-level threads exist when multiple user processes exist in an operating system which are scheduled by the kernel, but in which the threads belonging to those processes are scheduled within the process code itself (the kernel does not see those threads as separate entities). This is achieved by a process having its own scheduler built into its software. The advantage here is that the threads within that process can be scheduled and managed in entirely custom fashion, within the boundaries of that process.

Kernel-level threads require processes to request that the kernel creates and manages threads on that process's behalf, and then allow them to be managed by the kernel scheduler. The advantage here is that processes do not need to include their own custom thread scheduling code as part of their software, but must also accept all of the constraints of the kernel scheduler, which may vary from one operating system to another.

In theory both schemes could coexist. The operating system has no visibility of user-level threads, it cannot see them, and therefore even in a kernel-level thread model, processes could also have their own local user threads too. However, in general, the idea is that where kernel level threading is available, programmers should utilise the operating system features accordingly. This might be an enforced requirement in some systems, to ensure all threads are managed according to a given policy.

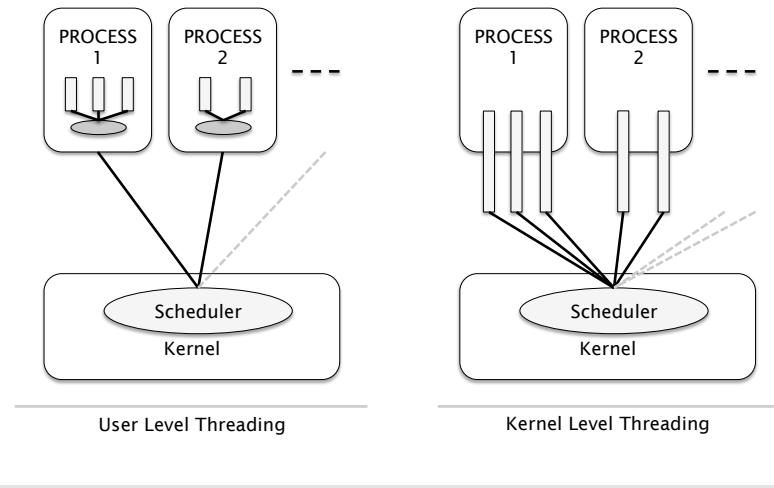


Figure 10.2: Thread management models. User threads are created and managed by additional schedulers that exist only within each process. Kernel level threads are created and managed by the kernel on behalf of processes.

10.4 Task scheduling

We have now gained an overview of how tasks can coexist in a machine's operating system, and how they appear to all intents and purposes to execute in parallel with each other. We have also learned that in actual fact this may be an illusion, and the vast majority of processes are executed by a method of interleaving small portions of the code execution of each task in turn.

A useful analogy is that of a reader who buys five books to take on holiday. They could take the conventional approach, and read one book at a time, completing each book before starting the next. This is analogous to serial workload management. In early computer systems, the computer could only process one job at a time, and could only run one process, and no threads. Users would submit their jobs to an operator who would schedule them manually for loading and running one by one.

As computer systems improved, and early operating systems progressed, it became possible to run multiple jobs using time-slicing, such that part of the CPU time was allocated to each job. This is essentially the same as task scheduling as we know it today, though the granularity has become much finer.

Returning to our analogy, the book reader could instead chose to read one chapter of each book in turn, such that all five books appear to be being read at the same time. This is yet another form of multiplexing, a concept we have already come across.

Now, imagine that the reader gets somewhat over-enthusiastic about this new method of reading, and decides to read a single page of each book in turn, and then just a single paragraph. The reader would spend more and more time physically swapping between books, replacing bookmarks, and so on, and less and less time actually reading. This is an example of the impact of the **task-switching overhead**. And what this tells us is that there is a limit to how fast tasks can be switched, and how fine the granularity of task switching can be, whilst maintaining reasonable efficiency.

In the worst case, any task switching will result in the jobs or tasks being executed in more absolute time than if they were executed serially without the switching overheads. However, there are reasons why this is not always true, and indeed it can be the case that task switching achieves much higher throughput (work done in a given amount of time) than the serial case. Of course it is also the case that we generally also want many tasks to run in concurrent fashion in a modern operating system, and therefore the option of serial execution is simply not feasible in a modern computing environment.

Regardless of how fine the granularity of the task switching, it must be managed by some sort of control function in the operating system kernel. This is known as the **task scheduler**, and this task scheduler follows some well-defined policy with respect to how it manages the tasks. These policies are known as **scheduling algorithms**.

10.5 Scheduling algorithms

The concept of task scheduling can be quite complex, with a number of fairly sophisticated algorithms. However, ultimately all of these approaches attempt to fulfil one basic purpose: to ensure that all processes receive an appropriate share of resources. If this can also be done in such a way that the system's resources are optimally utilised, then this is the ideal situation.

Let us first consider a very rudimentary task scheduler, an imaginary case that will allow us to grasp the concepts a little better. We will call it 'SimpleSched'.

This task scheduler has a list of currently running tasks, known as a **task list**. Each time a new program is started, it is added to the list, and each time a program ends, it is removed. The list can therefore grow and shrink dynamically as the system goes about its business.

SimpleSched will use a simple **round-robin** model of scheduling. This algorithm simply goes through the list one task at a time, and executes each task for a fixed period, say 1ms. Then, when it reaches the end of the list, it goes back to the beginning and starts over again, repeating indefinitely.

When the next task is picked from the list, the current running task is **suspended**, and the next task is **resumed**. This is like pressing pause and play on your TV recorder with different recordings. This switching mechanism is known as a **context switch**. Among other things, this involves copying the paused CPU state into the process control block (PCB) and restoring the resuming process's paused CPU state from its PCB back to its active condition in the processor.

We can see that SimpleSched has some notable properties. First of all, every task gets an equal amount of CPU time. Also, every task is executed once per full pass through of the list.

On one hand this algorithm represents fairness, which is an aspect of task scheduling that we may find desirable. However, there is a problem here that would become apparent in a real system fairly quickly. Because every task has to wait its turn in the list, important tasks are made to wait whilst unimportant ones have their turn at utilising the CPU. This means that SimpleSched has no sense of prioritisation. **Prioritisation** or **task priority** is the idea that some tasks are more important than others and might be given more CPU time, and/or executed more often.

There is a further problem with SimpleSched. Whilst it seems fair to give every task the same amount of CPU time, it is entirely possible that this is a huge waste of resources. Consider that one task may be very intensively processing data, and will use every clock-cycle of its allocated CPU time usefully, but another task may have very little to do, and it will effectively be idle for much of its allocated time.

Yet another task may want to do lots of work, but actually spends a lot of time waiting for something outside its control, such as data requested from a hard drive, which may well need 5 or 10 ms to become available. This kind of task is waiting for an IO event.

We might argue that actually, having equal time-slices is not fair after all: If some of those tasks do not need that much of the CPU's resources then why deprive other tasks from using that capability for no reason?

Fortunately, with the right task scheduling policies, we can attempt to overcome some of these limitations quite successfully. However, we first need to understand another scheduling concept: queues.

10.5.1 Task scheduling queues

In our first example, SimpleSched, we described a group of tasks organised as a list. Whilst this is a valid scenario for a scheduler, it doesn't conveniently encapsulate some of the ideas we wish to develop, and which more complex scheduling algorithms use. Instead, we need to make use of the concept of a **queue**.

Imagine that SimpleSched was modified so that instead of having a list that the scheduler works through one by one, it instead has a queue. As each task is suspended, it is placed at the back of the queue, and the item at the front of the queue is resumed for a period of time. Eventually every task in the queue will be resumed, and then suspended and pushed back into the queue to come around again. New tasks are fed into the end of the queue to join the group, and terminating tasks are simply dropped from the queue and not sent back into the queue for a further scheduling cycle.

In its simplest case, this is a **FIFO** queue (**First-In First-Out**). You might note that the net effect of this model appears to be identical to SimpleSched. It still sees each task executed once per full cycle of the queue. However, we will see shortly that adopting a queue model will give us new capabilities for scheduling. At this point, it is worth noting that we have actually described a basic model of scheduling, known as a **pre-emptive scheduler**.

10.6 Pre-emptive scheduling

So it turns out that if SimpleSched is modified to use a queue rather than a list, it is effectively a pre-emptive task scheduler as would be recognised by most operating systems.

A non-preemptive scheduler in contrast, would go through the queue or list, resuming each task in turn, and allowing it to run until that task yields control back to the scheduler. In effect, the tasks decide how long they get. But this means that tasks have to be very well-behaved in order to achieve a fair and sensible scheduling outcome.

The idea of pre-emptive scheduling is that a regular time-keeping element triggers the switch between the current and the next process. This is often achieved by a system component called a **Repetitive Interrupt Timer (RIT)** which sends a signal to the CPU after a certain amount of time has elapsed. In the simplest case, this time period is uniform for all

tasks. However, there is no reason why this must be the case, and some tasks may be allowed a longer period of execution than others.

If a task is assigned a rating, known as a **priority level**, the scheduler can apply different rules to that task, as compared to a task with another priority rating. For instance, it might decide if it should be given a short period of CPU time, a medium period, or a long period, for example.

Another feature that can be introduced into a scheduling policy is the idea of processes voluntarily yielding control back to the scheduler prematurely. Imagine a process that is resumed, only to decide that it has no work to do right now. It could use the time slice up by simply waiting. On the other hand, it could send an instruction back to the scheduler to say it has nothing to do: in other words, it **yields control** back to the scheduler, which can then immediately suspend the task and resume another one.

This concept allows the scheduler to use all of the otherwise unused time in idle tasks to service other tasks in the queue.^[134]

An important thing has just happened here, and it is worth noting. When a task scheduler can allocate fixed times per task, even if the amount of time is different for each task, then this allows the task scheduling to be **deterministic**. In other words, we can always predict how long a particular task has to wait before it next gets to be executed, and how much total CPU time it is able to utilise.

However, if we introduce the concept of tasks being able to yield control, then there is no easy way to know how long any task time-slice will last, and the determinism of the system becomes very weak, if not altogether eliminated.

In this case, whilst we still know the worst-case time between a task's successive time-slices, we do not know the best-case or even the average. This is a very important issue for some applications, such as real-time and safety-critical systems, where it will result in execution times of particular activities varying widely (a problem known as **jitter**).

We can consider an example with some numbers here. Suppose a simple scheduling algorithm requires 5ms per task switch, and allocates 20ms per time-slice. And consider that there are 100 tasks in the queue. How much time is available per thread ? How often do threads repeat?

[134] This is a point for the programmer to consider carefully when writing an application. Most programming languages provide support for this: for example `pthread_yield()` in the C language, and `yield()` in Java.

Timeslices Example (simple case)

Each task requires a total of 25ms. There are 100 tasks so a total of $25 \times 100 \text{ ms} = 2500 \text{ ms}$ is needed for all tasks to execute once.

A task receiving 20 ms of CPU time every 2500 ms has $20/2500 = 0.008$ (0.8%) of CPU time.

We can also see that each task repeats once every 2.5 seconds, or a frequency of $1/2.5 = 0.4$ times per second.

This is quite slow, and would not give very smooth system performance.

We can also consider what happens if **yield** is used to skip over inactive tasks in the queue. Suppose 50 of the tasks yield as soon as they are resumed, what is the result?

Timeslices Example (with yield)

Each active task requires a total of 25ms. Each inactive task yields as soon as the 5ms task-switch is completed. So we have $(50 \times 25) + (50 \times 5) = 1500 \text{ ms}$ total time

An active task receiving 20 ms of CPU time every 1500 ms has $20/1500 = 0.0133$ (1.33%) of CPU time.

We can also see that each task repeats once every 1.5 seconds, or a frequency of $1/1.5 = 0.66$ times per second.

Still relatively slow, but quicker, and CPU time per thread has increased.

10.6.1 Multiple queues

Given the scenario described, where a task may decide to yield control because it has no work to do, there is an opportunity to improve the scheduling model further by introducing multiple queues.

Consider what happens with the single queue: A task is resumed, it has nothing to do, so it yields control back to the scheduler and is put back in the queue for another round. So the scheduler could spend an

appreciable amount of time resuming processes that simply yield control back to the scheduler again straight away. This is costly: remember the task switching overhead.

A better approach, which helps to alleviate this problem, is having two or perhaps more queues. Then tasks can be placed in a queue that best represents their situation at that moment.

Let us consider a scenario where several queues are defined as follows:

- ▶ **Ready Queue:** A queue containing all tasks that are currently ready to be executed again.
- ▶ **IO Wait Queue:** A queue containing all tasks waiting for an IO activity to complete.

In this system, with two queues, the scheduler can move tasks to the **IO Wait Queue** when they yield and signal that they have an IO wait condition. This means that the task scheduler can suspend and resume tasks in the **Ready Queue**, knowing that they are not going to simply yield control again immediately after being resumed, or waste time waiting during their time-slice. Also, since all of those extra and unnecessary task-switch overheads can be eliminated, this leaves a little more CPU time for servicing ready tasks.

Meanwhile, the tasks that have ended up in the IO Wait Queue are made to wait in line, until they achieve their IO objectives, at which point such a task is moved back to the ready queue.

10.6.2 Interrupts and polling

We have just encountered the idea of a task being able to be designated as waiting for an IO event. In a sophisticated task scheduler, such a task will be moved to the IO-Wait Queue and will stay there until the IO event is completed. This raises the question of how the scheduler knows this is the case, and introduces the idea of polling versus interrupts.

IO-Polling is an algorithm in which a task repeatedly reads the status of a device to see if it has completed the requested task. For example, if a disk access is required, a thread could request a disk access, then repeatedly read the status of the device at short intervals. Of course the slowness of the disk access (of the order of 5-10ms) means that a lot of CPU time will be spent getting the answer 'no i am not ready yet'.

This approach is wasteful. As noted earlier, if we have a single-queue scheduler, every task must be resumed at a frequent interval, in order

for them to update their polling context, and some of those tasks will simply be spending their time-slices reading device statuses and getting the 'I am not ready yet' response.

Interrupts are signals used to indicate that events have occurred in a computer system. They can be driven by timers to create repetitive events, or to create delays, and also by IO devices and other hardware resources, to indicate occurrences such as a disk access being completed. Therefore, interrupts can be very useful in task scheduling.

In the more sophisticated task scheduler, we described moving tasks to the IO-Wait Queue when they were waiting for an IO event. Now if we add interrupts into the model we can see how this works. A thread wishing to perform an IO operation will set up the request such that when it is completed, an interrupt is generated. The thread can then put itself into a suspended (sleep) state until the interrupt is received.

In the case of a single queue model, the sleeping thread could simply yield its execution as soon as it is resumed, and this will continue to happen until it receives an interrupt.

In the case where an IO-Wait Queue is used, the interrupt causes the thread waiting in the IO-Wait queue to request that the scheduler moves the thread back to the ready queue, and this means that the waiting threads do not waste any CPU time at all whilst waiting for an event.

It is worth noting that events can be caused by disk units, by input devices such as cameras, by network interfaces, and many other IO devices and peripherals. For example, a thread might initiate a disk data read, and then have nothing else to do until the disk has read the data, transferred it to a designated area of memory using DMA, and generated an interrupt.

One of the consequences of having an interrupt-driven IO policy is that many tasks can share CPU time in such a manner that when tasks are stalled waiting for IO, other tasks can use the spare CPU capacity for something useful. These tasks may even belong to different users, meaning that other users get to use the spare system capacity. This approach should ensure that the utilisation of resources is as efficient as reasonably possible.

10.7 Achieving priority

Priority is an important concept in task scheduling. It can be achieved in several ways. The idea of priority is that some tasks are more important than others, and need to be executed more often and/or have more CPU time.

In the simplest schedulers, tasks have no priority, and simply get serviced as they are reached in the queue (first-come first-served algorithm).

The idea of priority is managed by the operating system by default, but can be influenced by the user or the applications they run. For example, in Linux, tasks can have a priority level running from -20 to 0 and from 0 to +19. Zero is the default priority, larger positive numbers mean a lower priority, and larger negative numbers mean a higher priority. So we might decide to allocate 'MyBirthdayReminder' a priority of 19 because it doesn't need to be serviced very often at all. A video camera recording app may require higher than normal priority, perhaps -5.

In a more complex system, perhaps a server that supports multiple capabilities in a company, priority may be set low for tasks that are not time-sensitive. For example, running a daily stock report does not need to be done in 30 minutes, it can take 6 hours and just as easily complete its purpose on time. Such a task might be assigned low priority so it doesn't reduce the quality of service for users running more interactive or urgent tasks on the server (e.g. processing live customer orders).

What do these numbers actually do? In practice, the priority grading in a particular operating system works in a way that is likely to be specific to that operating system. However, there are two general principles that the scheduler might use:

One option might be to change the length of time allocated to each time-slice given to a particular task. So some tasks might have time-slices of 1ms and others might have 0.9ms, 0.5ms, 0.1ms, and so on. In practice, this may be difficult to achieve.

Another approach is to keep the time-slices uniform, but to skip over tasks in the queue so they get time-slices less often. This is achieved by having a queue in which higher priority tasks can be placed ahead of lower priority tasks.

Another variation is to have several queues, high, low, and medium priority queues, for example, and to manage the issuing of time-slices to threads by placing them in the appropriate queue.

In a specialised operating system, priority models might be managed in different ways from those of a general purpose OS. For example, in an RTOS it is not unusual to have scheduling models that rely upon **Earliest Deadline First** (EDF) priority. This makes sense, since the most urgent task must surely be given priority. In a more general purpose operating system, the idea of explicit deadlines for completion of tasks is less relevant than other considerations.

10.8 Advanced CPU architecture OS support

As the nature of computer operating systems, and their uses, have evolved, so have the demands placed upon processors, and the solutions devised to meet them. If this was not the case, then computer systems would still be using single-core scalar processor architectures.

However, modern processors exhibit a number of specialised capabilities which have only become possible with recent increases in circuit complexity, from millions of transistors per chip in the 1990s, to billions of transistors in the 2020's. The relationship between operating systems and the desire to utilise the extra complexity offered by these new chips has resulted in the processor technologies we see today, and of course they are still evolving.

10.8.1 From scalar to superscalar

As we discovered earlier, in Chapter 3, the limitation of executing one instruction at a time (scalar execution) was quickly recognised as a constraint that could potentially be overcome. The idea stems from the observation that code contains fine-grain parallelism, known as **instruction-level parallelism** (ILP). In other words, neighbouring instructions in a program sequence can often execute independently of each other, whilst some others are dependent upon their neighbouring predecessors, or slightly more distant predecessors. Therefore, a code sequence generally contains a mixture of dependent and non-dependent groups of instructions. As a result, it is possible to opportunistically execute non-dependant instructions simultaneously.

The ability to start multiple instructions simultaneously is known as superscalar instruction issue, and the ability to execute them simultaneously is called superscalar execution. This allows ILP to be exploited as far as is reasonable. However, it was eventually demonstrated that there are

limits to how much ILP can be exploited easily in any given part of a program, even with clever code optimisation algorithms and compilers. Generally, a single program sequence (typically a thread in operating system terms) will rarely be able to execute as many as 4 instructions in parallel for each and every clock cycle. The average over a long program sequence may well be between 2 and 3 instructions per clock (IPC).

Compilers attempt to maximise ILP, and their ability to succeed is certainly influenced by the kind of program being executed. However, there are limits, and the prospects of ILP achieving something like 8 instructions being executed in every clock cycle is unlikely to be obtainable in most general purpose computing contexts.

10.8.2 Multicore

It was eventually realised that investing more and more transistors to deliver higher levels of ILP would not deliver the desired outcome. Extra hardware is costly, and if it delivers a gain of 1% or 2% at best then it may well be a poor design choice. When heat and power are added into the equation, there may even be a case to say that excessive ILP hardware will make performance worse in those terms. Obtaining a 1% performance gain at a cost of a 20% increase in power consumption doesn't sound like a good deal.

For this reason, among others, processor designers began to turn away from ILP as the mainstay of performance increments in next-generation processors. Most current superscalar processors have settled into a 4 to 6 instruction ILP model, but at the same time, processor manufacturers have begun to move toward the idea of using extra available transistors to duplicate CPU cores to run threads in parallel (Thread Level Parallelism or TLP).

Whilst one model was to simply have multiple separate processor chips (multiprocessor system) as we discussed briefly in Chapter 3, this does not exploit the ever-increasing integration possible with VLSI. Designers then began to wonder: why have one over-complex processor on a chip, when we can have 4 more modest and streamlined processors?

This is the basis of multicore processors. The idea of on-chip processing arrays is not new. Indeed, chips have been produced even with thousands of cores for quite a while. However, these tend to be very simple processors. A full-blooded desktop PC processor is a more complex thing. Therefore, most multicore systems currently used in general purpose systems have perhaps 4 to 8 processor cores on a chip.

Because the operating system can be modified to recognise the existence of multiple cores, the task scheduler in particular can run multiple tasks on multiple processors. No longer does the scheduler have to run one thread at a time: it can run 4 or even 8 threads simultaneously with true concurrent operation. This idea can be extended further. In actual fact each of the cores can run a scheduling queue independently, so that we may have a number of threads being multi-tasked on each core. The workload can then be spread across the cores to maximise performance.

Obviously this boosts performance considerably in theory. But remember that a single chip with 8 processor cores still has one system bus, one physical memory and one set of IO resources, and that these must be shared among the 8 cores.

This memory access bottleneck can be augmented by using suitable cache design. If each core has its own instruction and data cache, and the whole chip also has a unified cache to tie together these eight cores to memory, then the pressure on the memory bus could be reduced very significantly.

Therefore, a typical multicore architecture will have a structure that includes multi-level cache hierarchy, methods of dealing with memory coherence, which we encountered in earlier sections, and some capability to exchange information rapidly between cores.

10.8.3 Hyperthreading and SMT

Given that a multicore processor allows an operating system to run a separate scheduling queue on each core, there is an initial boost in performance that could be expected to be up to 'n' times more performance when 'n' cores are used.^[135] However, the act of scheduling and switching of threads on those individual cores is still no more efficient than it is on a single core.

Remember the example in Chapter 8, where a reader tries to read five books at once, and the granularity of their reading is reduced incrementally. Reading one chapter of each book in turn, then one page, and then one paragraph, leads to less and less reading being done, and more and more time being spent swapping between books, putting in bookmarks, etc.

This principle nicely illustrates the problem with task scheduling and task switching. This is partly due to the on-chip **state** of a processor core. Processor state relates to things like the contents of registers, buffers,

[135] In practical terms the gain will never be quite as much as n, due to overheads in the operating system, even if all of the other resources are in unlimited supply. Amdahl's Law once again!

the execution pipeline hardware, and other items. Switching all of that state information when a task switch occurs is costly, taking tens or hundreds of CPU clock cycles per task switch.

We also know that when a thread hits a limit in terms of its instruction level parallelism, it needs to wait until the processor can clear the backlog of waiting instructions. Fine-grain parallelism in a single thread has unavoidable limitations.

Hyperthreading/SMT^[136] is a technique that attempts to optimise these problems. The idea is that instead of having a single set of registers, cache, etc, for each core, and then swapping that state content to and from memory in a coarse fashion, when a task switch occurs, it is possible to duplicate the state related circuits of the core for several threads. So there may be perhaps 2 or 4 thread states held actively in hardware, but still only one set of computational hardware for computation, memory access, cache, etc.

As a result of this approach, multiple threads can coexist, and each thread can attempt to issue instructions at the same time. They will of course compete over the limited resources available for computation, such as integer and floating point ALU's, and memory ports. But this is potentially quite a good thing as it ensures all of these resources are heavily utilised.^[137] The result is that each thread will be able to progress small pockets of instructions at a time, intermixed with the work of other threads.

An example of this is shown in Figure 10.3. Here, if a particular thread gets **stalled**, because it is waiting for memory for instance, then the other threads can use those resources more than normal, and make more progress. In this way, threads can exploit parallelism not just via ILP within an isolated thread but in terms of a fine level of granularity across threads.

Without hyperthreading/SMT, the stalled thread would just wait, wasting CPU time, because only one thread is active at a time. We can see in Figure 10.3 that the SMT approach allows the same work to be completed more quickly: in other words, the performance of the system has been increased significantly, as has the utilisation of CPU resources.

As before, the fact that 4 threads can coexist will not necessarily deliver 4 times the performance. But if 4 threads can deliver 2 times the performance that is still a big gain. Ultimately, in a system with 8 cores and 4-way hyperthreading, the best possible performance gain might be

[136] In practice, Hyper-threading is the same conceptually as SMT (Simultaneous Multithreading). Strictly speaking, Hyper-threading is an INTEL branded SMT mechanism, but the term has caught on in general use as a byword for SMT.

[137] Maximum performance occurs when all CPU resources are used all of the time, known as peak utilisation.

Figure 10.3: SMT versus traditional multithreading. (a) shows two independent threads being switched back and forth via traditional multithreading. (b) shows two threads operating as simultaneous multi-threads, both attempting to progress in a piecemeal fashion in competition with the other.

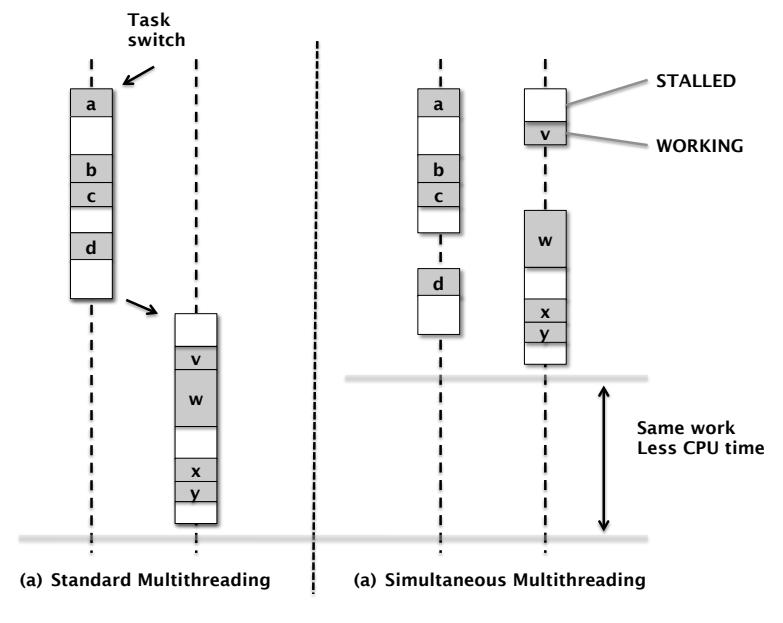


Table 10.1: Examples of hyperthreaded multicore processors. Benchmark scores via cpu.userbenchmark.com (64-core-point test).

	Price £	Cores	Threads	Benchmark
AMD Ryzen TR 2920X	388	12	24	1925
AMD Ryzen TR 2990WX	1490	32	64	4142
INTEL Core i7-9700KF	360	8	8	1062
INTEL Core i9-9980XE	1850	18	36	2851

imagined to be 32 times the performance. But this would never be achievable in practice. However, a real performance gain of say 8 could well be achievable with the right workload.

It is important to note that whilst a core might only support 4 hyper-threads, these can still also be individually switched via task switching, as well as running simultaneously with others. In practice, a hyperthread may actually be a group of alternating threads running in parallel with another group of hyperthreads alongside and doing the same thing.

Examples of the performance impact of cores and hyperthreading support are given in Table 10.1, to illustrate this point. For the particular benchmark used (of which there are many), we can see that a processor with nearly three times as many cores and threads (AMD 2990WX vs 2920X) achieves about 2 times the performance. Meanwhile a processor with around double the number of cores but four times the threads (INTEL 9700KF vs 9980XE) gains almost three times the performance.^[138]

[138] We should be cautious here of course, one benchmark may reflect particular capabilities and not a general level of performance. We should always choose benchmarks carefully to reflect the intended use of the component.

An important lesson here, therefore, is that increasing resources doesn't automatically translate into equal increases in performance; the situation is far more subtle.

10.9 Summary

Exploring the nature of operating systems, from the workload management perspective, we should have been able to see ultimately that a modern computer system is a cleverly coordinated collection of software elements. Rather than a few desktop packages running in their respective windows, the reality is that we can have literally hundreds of individual software parts running side by side, each playing an essential part in some aspect of managing the smooth running of the system, and your applications.

Not only do many software elements have to operate together in the same workspace, but they also have to communicate information to each other, or preserve their information privacy. This is achieved via the operating system too. Indeed, some of these aspects are essential for maintaining security and reliability in computer systems.

Finally, we learned that the illusion of multi-tasking is often just that. Even though we perceive many processes, applications and threads running in parallel, it is often the case that these components are actually being sliced up into small parts, and serviced piece by piece, in an interleaved fashion. The time-slicing, and the scheduling of these actions, lies at the heart of the modern operating system.

Looking toward the future we have observed how the current generation of processor architectures have begun to evolve toward supporting these demanding requirements. The emergence of powerful hyperthreaded architectures and their continued development will undoubtedly shape the next ten years of computer systems architecture.

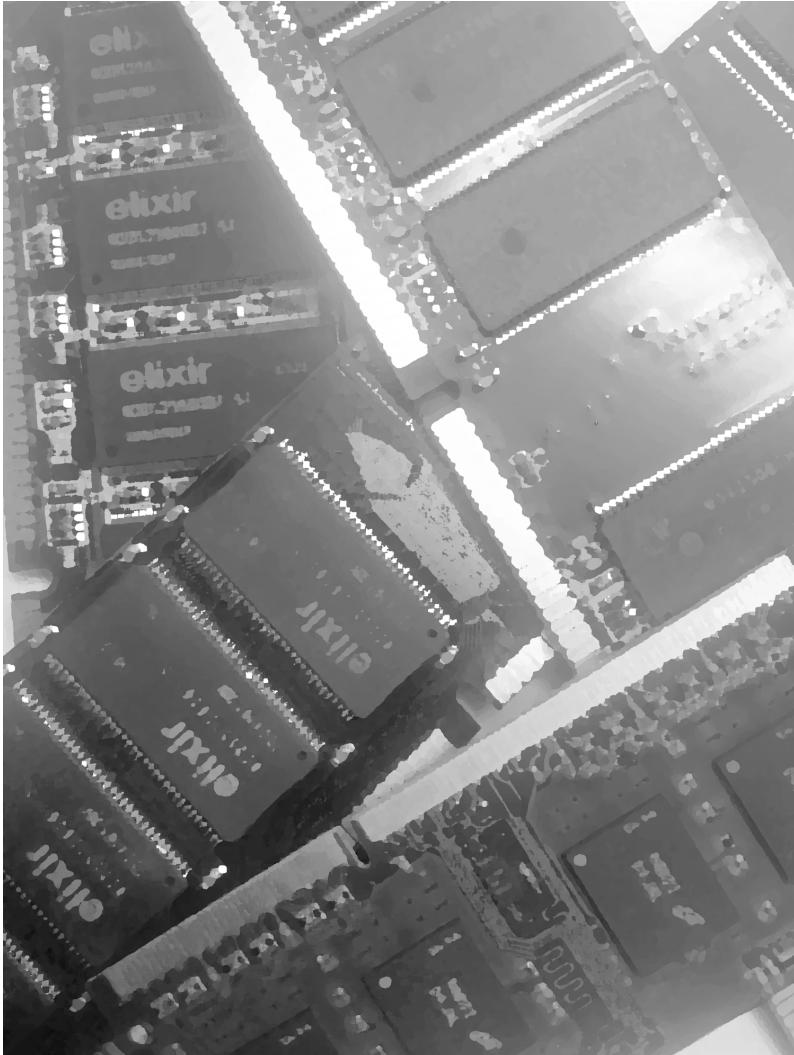
10.10 Terminology introduced in this chapter

Context switch	Deterministic
FIFO	Hyperthreading
ILP	Instruction-Level parallelism
IO Wait Queue (task)	IO-Polling
Jitter (task)	Kernel level thread
PCB	PID
Prioritisation	Priority level
Process control block	Process identifier
Processor state	Read Queue (task)
Repetitive Interrupt Timer	Resumed task
RIT	Scheduler
Scheduling algorithm	Simultaneous Multithreading
SMT	Stalled thread
Suspended task	Task list
Task priority	Task queue
Task scheduler	Task-switching overhead
User level thread	Yield control (task)

These terms are defined in the glossary, Appendix A.1.

11

Memory Management



11.1	Memory management	202
11.2	Memory allocation	202
	Memory access control	204
11.3	Virtual memory	205
	Paging cost penalties	208
11.4	Inter-Process Communication	209
	Pipes	209
	Message queues	210
	Shared memory	212
11.5	Privileges and restrictions	212
11.6	Summary	216
11.7	Terminology introduced in this chapter	217

11.1 Memory management

Alongside workload management, another important responsibility of any operating system is management of memory resources. Recall that we learned that a computer system has a physical memory, made up of memory chips, reaching a certain physical memory capacity. However, if an operating system was limited to executing only the number of tasks that would fit into this physical memory constraint, then the operating system would have a significant limitation, and could quickly run into trouble.

A given application might start up in a state where it uses a relatively small amount of memory, but during its use it may require more memory capacity. An example may be an image editor, where by default the application is initialised with a small and low resolution default image canvas. The user then opens a file representing a large and very high resolution image, demanding much more memory. What then happens if the physical memory does not have enough space left to satisfy this request? One solution is to simply display an error message 'sorry you've run out of memory'. However, this is not a very helpful way to manage the problem.

We will see that there is a better solution to this problem shortly. But first let us dig a little deeper into exactly how memory comes to be allocated in the first place.

11.2 Memory allocation

Memory allocation is the process by which a task obtains temporary ownership over some memory space within the computer system. In terms of a programming perspective, a software engineer writing an application for example, there are two types of explicit memory allocation that are likely to be encountered:

[140] The C programming language, for example, provides the `calloc`, `malloc`, and `free` function calls to support this.

Static memory allocation occurs when a program is written in which it allocates a block of memory as soon as it is invoked, and it retains that fixed amount of memory for as long as the program is running.

Dynamic memory allocation occurs when a program decides at some point during its operation that it needs a particular block of memory, and then requests it via a suitable mechanism supported by the operating system. This mechanism is usually a kernel library function, invoked via a function or alias provided within by the programming language^[140].

Another type of memory is also available to the running task, and this is a standard 'default' memory allocation used during its operation, provided to all tasks. This relates to small amounts of data storage allocated during the execution of the code and stored in a region known as the **stack**. The stack must be provided with a sufficient amount of memory space to reside within, and this is again a block of memory that is ultimately allocated by the operating system when the program starts.

A further important point is that any memory allocated to a task must at some point be de-allocated; generally we refer to this as **freeing** or **releasing** memory. This ensures that when a task is finished with a piece of memory, or is terminating, any memory it was using can be released to be reused elsewhere.^[141]

We can see therefore that a program may statically allocate some memory when it starts up and optionally might dynamically allocate many blocks of memory during its execution. We also see that dynamically allocated blocks of memory are also likely to be released or **freed** by the program in a dynamic fashion when no longer needed. This is also typically done via an operating system call.

As a consequence of memory being repeatedly allocated and freed, and not necessarily freed in the same order as allocated, the memory allocated to an application or task can become **fragmented**. In other words, a program can have many blocks of memory, scattered all over the memory space of the computer system. This **memory fragmentation** is not unlike file system fragmentation, which we encountered earlier, and it has a performance cost that can lead to memory allocation becoming slow.

So far we have talked about programs requesting memory statically or dynamically. One might assume that this only relates to data structures, and this is certainly one case. However, the allocation of memory can also relate to the actual program code itself. For example, when a task is initiated it must be loaded into a block of memory, and the operating system must decide where and how much to allocate for that program code sequence. Indeed, program code can also be allocated memory space statically and dynamically in a fashion similar to data.

Code-space memory allocation is influenced by a process known as **linking**, which is part of the code generation process undertaken when a program is converted from source code into an executable machine code sequence^[142]. There are two types of linking:

[141] When this fails to happen, often due to a bug, memory can remain reserved without an owner, reducing the available memory. This is known as a **memory leak**.

[142] When programs are converted from source code into executable machine code, this is known as compilation. Linking is a further step in this process, in which any functions used by the programmer that are provided as a standard part of the language are linked to the main program to make a complete unit.

Static Linking is a process whereby all parts of the program (the various sub-modules it is built from) are combined together into a single self-contained block of program code, which can then be loaded in its entirety into memory whenever the program is executed. The advantage here is that everything is present from the start of program execution.

Dynamic Linking, on the other hand, is a process whereby the core of the program is loaded into memory when the program is started, but any additional modules are only loaded into memory as and when needed. They can also be discarded once they become surplus to requirements. The advantage of this approach is that the **initial memory footprint** of the program is much smaller, and it only grows as large as required to perform the parts of the program that are needed. Additionally, libraries of dynamically linked code can be provided by developers and used to reduce coding effort. These are known as **dynamic link libraries** (DLLs) on some operating systems.^[143]

[143] It is also often the case that multiple applications can run on a system and all use the same DLLs. This means using a shared code-base, and can improve efficiency and consistency.

Consequently, a statically linked program is likely to be more efficient in terms of consistent execution speed, since all modules of the program are immediately available in memory when needed. However, dynamic linking means that the total breadth of program code and functionality can be much larger, without wasting large amounts of memory that may never be used. Consider a program in which the user only uses 5% of its features in any one session; a statically linked application would waste 95% of the memory space every time it is used. And indeed, some applications may be so large in their entirety that it is impractical to fit the whole application into physical memory as a statically linked program.

Although dynamic linking allows much larger applications to use memory very efficiently, it also comes at a cost, since any new feature invoked by the user for the first time in that session must be loaded from disk on demand. This can be observed in many desktop applications, where using a particular feature causes a short delay whilst the hard disk spins into action and loads the new module of code into memory, especially if the system has been allowed to go idle for a while and the hard disk has shut down to save power. We might conclude therefore that dynamic linking is not very desirable in systems where fast, real-time, and predictable behaviours are important.

11.2.1 Memory access control

One of the consequences of memory being allocated as processes are created, and even during their operation, is that the operating system

must ensure that processes and threads do not interfere with each other's memory areas. Controlling the range of memory addresses that each processor is entitled to access is part of this responsibility.

Theoretically, no process or thread should be able to arbitrarily access memory belonging to another process or thread, unless the operating system is knowingly facilitating this. This restriction is essential for **security** and **resilience**, particularly for the following reasons:

- ▶ Rogue processes should not be able to corrupt the memory of other threads when they go haywire, which ensures that even if one application crashes, the others will carry on as normal.
- ▶ Likewise, a rogue program that is attempting to corrupt memory or steal data from another program cannot do so if it cannot gain access to other process memory areas.

Of course, making memory private also has downsides. There are times when two or more tasks do legitimately want to share some memory content. The operating system therefore also has to facilitate this capability when needed, but with careful control. We learn more about this in Section 11.4

11.3 Virtual memory

Having established that memory, be it code or data related, can be allocated by the operating system both at application startup and during its operation, we now have to return to the question of what happens when the physical memory is not large enough to provide for all of the memory required by all of the tasks running on the system at a given point in time.

As we noted earlier, an operating system could simply report an error and refuse to continue with that task, but this is far from satisfactory. Fortunately there is an alternative: **Virtual Memory**.

Virtual memory is an important innovation, developed to solve the problem of physical memory limitations. No matter how much physical memory a system has, there might be a scenario where we want more. To overcome this problem, a modern operating system will have not only a **physical memory** but also a **logical memory**. There are important differences between these two concepts:

The **logical address space** (the virtual memory) can be much larger than the physical memory. The bulk of this memory content is stored on a

secondary storage medium, typically an HDD or SSD storage drive. Since HDD/SSD storage capacities are potentially huge, the virtual memory can also be extremely large if required.

The **physical memory**, consisting of actual memory chips installed on the computer motherboard, holds a portion of the logical address space at any moment in time, but this content can be swapped between HDD and physical memory on demand. Physical memory is expensive so is usually limited by economics.

Therefore, in simple terms, the virtual memory is a large expanse of notional memory content, held primarily on the disk storage media of a system. Portions of this virtual memory can be loaded into physical memory whenever a particular application is being used and needs that data.

This is illustrated in Figure 11.1, where the example shows several co-existing processes, each occupying part of the physical memory, with memory allocated in **pages** (small blocks of memory).

In the example of Figure 11.1, several pages are being transferred in or out of physical memory, and the number of pages held in physical memory is limited compared to the total set of pages allocated in the much larger virtual memory.

Since the memory is often already holding valid data, the action of bringing new virtual memory content into physical memory usually requires that some of the current physical memory content is dumped back onto the storage disk to make space for the new content. This process of flipping the memory content of applications between virtual and physical memory is known as **disk swapping**, or sometimes as **paging**.

The operating system module responsible for managing this virtual memory allocation process is the **paging supervisor**. Often the storage space allocated to this purpose is known as the **swap file**, and the disk used for this can be referred to as the **swap disk**.

In many systems, the swap disk resides on the same disk as many other files, in order to minimise overall system cost. However, a dedicated swap disk might be used in a high performance system where a very fast disk unit might be used solely for virtual memory purposes, instead of competing for the same disk bandwidth as other file-system requirements.

As mentioned, in practice, the blocks of memory swapped back and forth between physical memory and secondary storage are organised into smaller units, referred to as virtual memory **pages**. These might be 512

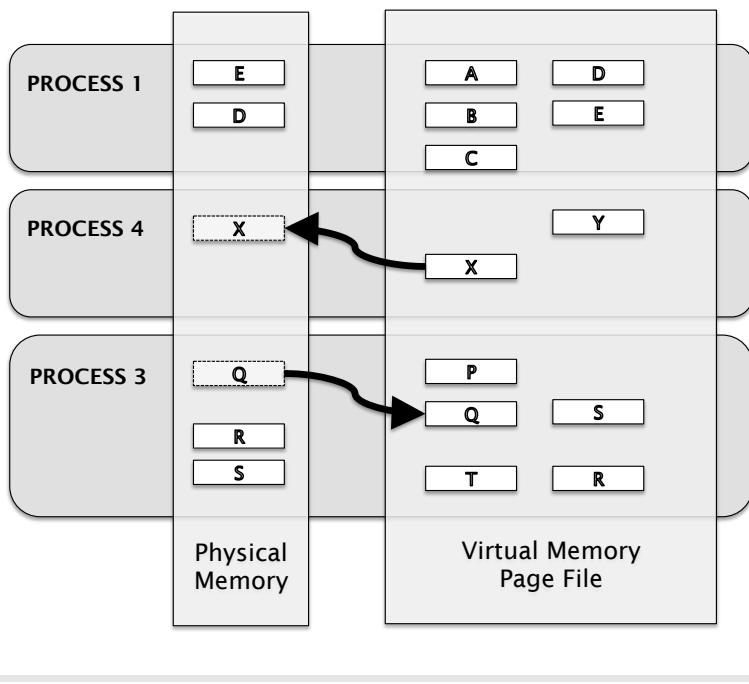


Figure 11.1: Virtual memory example. Showing selected pages belonging to processes held in physical memory, with the complete set of memory content also held in virtual memory on disk. Here, Process 4 has triggered a page fault, and Page X is replacing Page Q belonging to Process 3, which is being paged out in order to make room. This may be because it has not been accessed recently or frequently, and is therefore considered less important.

bytes in size for example, but 1 Kilobyte, 4 Kilobyte, or 8 Kilobyte are also common.

The paging supervisor keeps track of the pages of virtual memory allocated to a task, and when a task requires access to that memory page, if it is not already in physical memory^[144], it is moved to physical memory, and some other content is moved back to secondary storage. This is more efficient than swapping an entire application in or out of memory as one event, and is known as paging.

Importantly, it is necessary to realise that each time a page is swapped in to physical memory, the physical address allocated is likely to be different. This is because the paging supervisor looks for the next unused block of memory to allocate, and this will be changing dynamically all of the time the system is operating.

As a result of this, the paging supervisor and the operating system must keep track of which memory addresses are allocated to a given page. An address lookup capability is therefore needed to allow threads to find the correct physical address for their data in a lookup table known as a page table. This may be managed by hardware structures such as memory management units in order to achieve satisfactory performance.

[144] When a page is required, but it is not already found to be in physical memory, this is known as a **page fault**. It is somewhat similar to a cache miss. This triggers the page transfer process.

Figure 11.2: Memory usage versus physical memory. Example from Mac OS with 8 GB physical memory, and only 3.7 Gbyte actually allocated by the OS.

MEMORY PRESSURE		Physical Memory: 8.00 GB	App Memory: 2.03 GB
		Memory Used: 3.76 GB	Wired Memory: 1.29 GB
		Cached Files: 3.11 GB	Compressed: 447.0 MB
		Swap Used: 75.8 MB	

[145] But not so large as to be disproportionately expensive of course.

Paging makes sense, since a number of tasks will coexist in an operating system, but each will only need to use part of its total allocated virtual memory at a given time, therefore paging those portions of virtual memory into physical memory makes efficient use of the physical memory. It is feasible that this subset of all virtual memory might fit into physical address space with only infrequent paging, provided the physical memory is large enough.^[145] In a sense, it is not unlike the concept of cache, which we explored earlier. The physical memory caches a subset of the much larger virtual memory that is needed and most frequently used.

11.3.1 Paging cost penalties

The paging concept is clearly going to incur performance costs, since any movement of data to and from main memory will require disk access, and even with an SSD this takes an amount of time that is significant when compared to the processor execution speed of programs.

In unfortunate scenarios, an operating system may find itself swapping and/or paging memory content back and forth frequently as two or more tasks compete for the limited resources of physical memory, resulting in a problem known as **thrashing**. In this scenario the system spends much of its time waiting for the disk unit to catch up with data transfer requests, and less time actually doing useful work.

An obvious solution to thrashing is to increase the size of physical memory, in order to capture more of the critical pages required for the competing applications to run without resorting to paging too often. This must of course be cost effective. If it doubles the cost of the system, it might not be such a good design choice.

An example of memory allocated in a running system is given in Figure 11.2. Here, the amount of memory needed for applications is much less than physical memory, and consequently there will be minimal disk swapping.

Other options might be to install a much faster hard disk or SSD to mask the impact of thrashing (though this is just hiding the problem). However, it may simply be that the workload needs to be distributed across several computers in order to achieve good performance.

A further degree of optimisation of paging behaviour is possible with **pinned pages**. These are effectively pages that are designated as always in physical memory, and the paging supervisor knows that they should not be discarded to make way for new pages. This might be used to ensure for instance that certain operating system functionality is always resident in physical memory, to ensure consistent performance. This is of course at the cost of increasing the amount of paging encountered by other less important processes.

11.4 Inter-Process Communication

We have just examined the scenario where threads can share data and memory, but processes cannot. However, there are occasions where two processes may want to exchange data. In order for this to be possible, the operating system must support the concept of **inter-process communication**. This allows two processes to communicate in a way that is controlled by the OS.

11.4.1 Pipes

To deal with the situation of two processes wanting to communicate with each other, and to make this process more efficient, the concept of a **pipe** was invented. A pipe is typically a one-way conduit for sending data from one process to another.^[146] However, a pipe can only be created under operating system supervision.

Although a pipe is typically unidirectional, two pipes can exist side by side, one travelling each way, providing two-way data-flow. Usually these pipes also have a buffer of a few thousand bytes, allowing blocks of data to be sent rather than single bytes. This is much more efficient, especially as the sender and the receiver are tasks that are executing in different time-slices and therefore cannot interact instantaneously with each other in most situations.

One problem with pipes is that if they are created at run-time by a process, then they can only exist between threads or processes that are invoked from the same root process. In other words, they have some

[146] Some operating systems *do* support bidirectional pipes, but this is not universal.

[147] If Process A created a pipe wishing to connect it to Process B, then how would B know unless they could share information? After all, isn't that what the pipe is trying to do? This is the classic chicken-and-egg paradox.

shared contextual knowledge that allows them to both know about the existence of the pipe.

This situation occurs because pipes are created dynamically at run-time, and the creating process has no way of telling unrelated processes that the pipe exists^[147].

To overcome this limitation, the operating system often supports the idea of a **named pipe**. In practice, a named pipe may operate in a very similar way between processes, but because it has a fixed name, decided beforehand, any process can look for it in the OS environment and then connect to that pipe. The pipe may still be created dynamically, but its name will always be the same, which means that any processes wishing to connect to that pipe can be predisposed to look for it in advance.

Another aspect of pipes, named or not, is that data flows through them in a first-in-first-out (FIFO) fashion. This means that the order of the bytes sent down a pipe is preserved at the output connected to the receiving process.

11.4.2 Message queues

An observant reader might realise that whilst pipes provide a convenient mechanism for data flow between two processes, they do not allow data to flow in a more generalised fashion. This is because they are primarily intended to be point-to-point connections, and this does not match some commonly required cases of inter-process communication.

For example, suppose that instead of Process A and B exchanging data on a one-to-one basis, we instead wanted Process A to receive information at various times from Processes B, C, and D. We could of course create named pipes between all of these processes, and have pipes A-B, A-C and A-D. However, this is somewhat cumbersome, and it doesn't allow for the idea of multiple possible destinations. The idea of **producer-consumer** models is useful here:

Imagine a case where Process A creates blocks of data as a **producer**, and any of Processes B, C, and D can **consume** that data. Ensuring that all consumer processes can see the same block of data and yet only one can consume it, could become very messy in terms of programming and coordination between the processes. This is because a pipe is a point-to-point connection, and the scenario we are attempting to implement here is more like a bus/broadcast model (where many processes can see the same data).

If we consider the various ways such a queue might be used, there are actually several models that we might wish to support at different times:

- ▶ Point-to-point: Single producer-consumer,
- ▶ Fan-out: One to many, single producer, multiple consumers,
- ▶ Fan-in: Many to one, multiple producers, single consumer,
- ▶ Collective: multiple producers, multiple consumers.

Only one of these scenarios are readily practical for implementation via pipes. Again, necessity is the mother of invention, and therefore operating systems developers eventually introduced the concept of a **message queue** to deal with situations like this.

In a message queue, any process can **post** information to that queue, not surprisingly in the form of a message. So Processes B, C, and D need not have private pipes to Process A. They simply all need to be able to read the same message queue. Meanwhile, Process A will receive any message posted to that queue by Processes B, C and D.

A consequence of the queue is that any of the consumer processes (B, C and D) can take a message from the queue and deal with it, and because it has been removed, only that process will see it and react to it. This guarantees a unique consumer for each message. However, if necessary, a message can be left in the queue or put back into the queue for other processes to see. This is a choice for the programmer depending upon the exact kind of relationship the consumers have with the messages.

What is a message? In principle, a message could be very simple, even a single byte of data, but typically a message includes a short header followed by a block of fixed length data. The header determines what kind of message is being sent, and that dictates what the data means. It might indicate which process sent it and which process should read it, among other things. Consequently, the programming environment used to develop a program must understand the OS messaging protocols and formats, and processes sending and receiving messages must have a common understanding of what a message of a given format looks like.

As an example, a message called 'MyHealthMessage' might contain four bytes, relating to age, height, weight, and blood pressure. If both the sender and receiver understand this format, then the two processes can interact smoothly and the programmer needs to know very little about how messaging actually works 'under the hood'.

Where such defined messaging standards exist, they are defined in pre-defined code modules, known as libraries or **APIs (Application Pro-**

gramming Interfaces). Many operating systems have message processing functionalities integrated into their interactive desktop interface elements. The WINDOWS operating system for example, has a large number of defined messages relating to events that can happen in its environment, such as mouse click, changing from one window to another, pressing a button on screen, receiving a message from a USB device, and so on. Without messaging, these systems would not be able to work as they do.

11.4.3 Shared memory

There are scenarios in which an operating system will support shared memory being accessible by multiple processes simultaneously. In this case, a block of memory is reserved by a process, and then it creates sub-processes. The sub-processes will then be able to access the shared memory. In practice, this is much faster than using message passing or pipes for some cases. Some care must be taken to ensure that two processes don't attempt to modify the same area of the shared memory at the same time, as the results will be unpredictable. A method to overcome this is known as a **semaphore**: a mechanism that allows processes to tell each other when the memory is available for access and when it is in use. This ensures that threads access the data with the principle of **mutual exclusion**, sometimes referred to as **mutex**.

As an example, consider a very large data array, where processes are writing data into individual rows and columns at arbitrary times. It would be very inefficient if each process had to message other processes each time something was updated either with the entire array or even an indication of what had changed. Instead, a large shared block of memory can act as a central reference area for one array with all of the latest modifications present. However, we would not want process A reading data from an area that process B is updating, as this could result in a mixture of old and new values being retrieved by process A. Therefore, we would need to ensure mutual exclusion is implemented.

11.5 Privileges and restrictions

The concept of task and thread execution with different **privilege levels** is an important concept in operating systems.

At the very least, a user may come across the problem of user privileges when running an installation of a program or trying to execute a particular command on a command line terminal. The system will say that the program cannot be executed because it requires administrator privileges^[148].

This restriction is a deliberate feature of the operating system, intended to prevent unauthorised meddling with the integrity of the computer system, either by deliberate acts or by innocent tinkering by inexperienced users.

However, this concept of access and privilege goes much deeper into the operating system than a simple differentiation between an administrator and a user. Consider that in actuality, a user typically has many programs running on their system, applications, processes, threads, all belonging to their user applications.

If every thread in a system was capable of doing anything it wanted, then there would be no privilege levels in that system. Indeed, some operating systems fall into this category, and it may be acceptable under some circumstances. However, this is a very dangerous situation for a lot of other modern computer systems scenarios, since it means that any thread can access any part of the memory, perform disk access to any part of the virtual memory, execute any instruction that the processor supports, and access any underlying hardware element of the entire system. Who knows what those programs could do, either accidentally or deliberately?

Consider a couple of possibilities:

- ▶ A simple program error that overwrites a random area of memory could overwrite the task scheduler of the operating system or one of its tables, and consequently, the entire system will crash. Bad enough if one user is sitting at the computer when this happens. But what if the computer is a server and twenty people are logged on and running an airline booking system for example?
- ▶ Equally, if a server had 20 clients at different travel-agent companies, all running their software on the same server, and one of the clients could run a program that reads the memory of other users, they could read emails, steal data, corrupt their files, and engage in all sorts of mischief. This is definitely not desirable!

Therefore, in order to prevent the kind of scenarios we have mentioned, and many others, the concept of privileged access in an operating system provides several necessary advantages:

[148] Administrator, Supervisor, and superuser are effectively the same thing in this terminology.

- ▶ It prevents unauthorised access to blocks of memory that do not belong to the thread requesting access. (the thread can only access its own memory or that within its parent process). This is known as **memory protection**.
- ▶ It prevents programs from running, or prevents those programs from running other programs, without operating system control.
- ▶ It prevents the execution of specialised CPU instructions that are only intended for special reserved purposes within the operating system.

The actual definition of operating system privilege levels is related to the operating system in question. For example, in Linux systems we have the following definitions:

Kernel Mode: Also known as system mode, permits code running in this mode to access any area of memory, and potentially execute any CPU instruction. This is of course essential in order for the operating system to be able to control the machine it is running on. Processes running in kernel mode, known as a **kernel processes**, may also have different constraints applied to them with respect to task switching and task scheduling policies.

User Mode: In this mode, many aspects of the system are inaccessible to the process in question. When a process runs in user mode, known as a **user process**, it cannot arbitrarily access blocks of memory at will, and cannot execute certain CPU instructions. Attempting to access a prohibited memory area or execute a prohibited instruction will cause an error and the operating system will block this from proceeding, and may shut down that process.

The constraints applied to a process also apply to all of its threads, and any other processes which that process happens to run.

If a user process needs to interact with a resource that requires kernel mode privileges, it must do so by making a function call to the kernel; in other words, it can only access these resources via a trusted and reliable intermediary (the kernel). The kernel will check that the process/thread/making this request has the right to do so, and will block invalid requests.

Ideally this will ensure that no task can ever perform an incorrect action in the system, and if an attempt is made, it will be stopped, without affecting the continuing execution of all other tasks in the system. All other tasks should continue as if nothing has happened.

Table 11.1: Comparison of some key OS attributes.

	MS Windows	Mac OS	Linux	OpenSolaris
Kernel	Hybrid WM ¹	Hybrid WM	Monolithic WM	Monolithic WM
File System	NTFS, FAT, ISO 9660 ³ , UDF	HFS+, APFS, HFS, UFS, AFP, ISO 9660, FAT, UDF, NFS, SMBFS, NTFS	ext2, ext3, ext4, FAT, ISO 9660, UDF, NFS	HFS+, AFPS, FTP ²
Typical Use	Desktop, Server	Desktop, Server	Desktop, Server	Servers, Desktops
Filesystem Encryption	YES	YES	YES	YES
Firewall	Windows Firewall	IPFW	Netfilter	IPFilter
Resource Control & Isolation	Root Control ⁴ & File Flags, ACL, RBAC ⁶	Root Control & File Flags, POSIX ⁵ , ACL	Root Control, Namespace, &other, POSIX, ACL, MAC	Root Control, Containers ⁷ , Logical Domains ⁸
Notes	1 WM : With Modules 2 FTP, supports remote file access via FTP 3 ISO 9660 File System for Optical Media, e.g. CD , DVD. 4 Typically using chroot, forces a process to see a specified folder path as the root. 5 POSIX – Portable Operating Systems Standard Interface 6 RBAC: Role-based access control 7 Originally a feature of Solaris that managed a set of restrictions for a given process. 8 An example of virtualisation where each virtual machine (logical domain) has its own constraints.			

In other operating systems, similar modes are defined. In MICROSOFT Windows there is privileged mode and user mode for example. Of course, the exact constraints applied to each mode are operating system specific, but since they are defined for the same purpose, their overall philosophy will be the same.

11.6 Summary

In the last few chapters we have covered considerable ground relating to operating systems in terms of processes, and memory. Bringing together most of these ideas allows us to view a few different operating systems in terms of this new understanding. Table 11.1 presents several widely used operating systems as compared across some key features, including type of kernel, typical file system support, and important security features. Hopefully some of this diversity now begins to make sense, based on the fundamentals we have acquired.

Ultimately, the knowledge acquired here about the basics of operating systems will start to make more sense to those who delve further into the topic, particularly from a programmer's perspective, where there is every opportunity to utilise a wide range of operating systems capabilities to good effect.

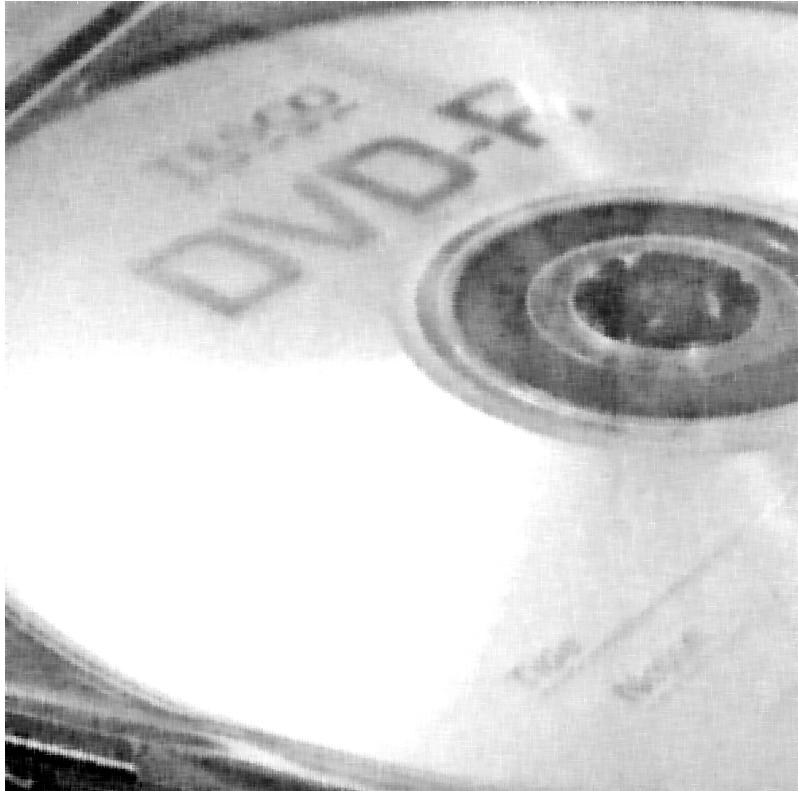
11.7 Terminology introduced in this chapter

API	Application Programming Interface
Disk swapping	Dynamic linking
Dynamic memory allocation	Freeing memory
Inter-process communication	Kernel mode privilege
Linking (compiler)	Logical address state
Logical memory	Memory footprint
Memory fragmentation	Memory protection
Message queue	Named Pipe
Page (virtual memory)	Paging
Paging supervisor	Physical memory
Pipe	Privilege levels (OS)
Physical memory	Releasing memory
Static linking	Static memory allocation
Swap disk	Swap file
Trashing	User mode privilege
User process	Virtual memory

These terms are defined in the glossary, Appendix A.1.

12

File System Management



DVD storage media.

12.1 Foreword	220
12.2 The concept of a file system	220
12.3 A file-system example	222
12.4 The boot sector	222
12.5 Volumes and partitions	223
12.6 The root directory	224
12.7 File-system hierarchy	224
12.8 Special file cases	225
12.9 File attributes	227
12.10 The variety of file systems	229
12.11 File compression	230
File compression example	230
12.12 Automatically compressed file systems	232
File encryption	233
12.13 File-system resilience	234
12.14 Internal file formats	235
File structures and encodings	236
File format tradeoffs: an example	239
12.15 File management	241
12.16 Summary	243
12.17 Terminology introduced in this chapter	244

12.1 Foreword

Though computer systems have experienced many significant changes and evolutionary steps in the past five or six decades, the nature of data has perhaps changed more than any other aspect. We can say this because of the huge changes in data volume being managed by computer systems, the explosion in the diversity of types of data, and the way it is stored and retrieved, both in local systems and via networks and remote resources. As a result, the concept of file management has evolved alongside operating systems technology in tandem. In this chapter we will explore the concept of file systems, the nature of data, and some important aspects of file and file-system management.

12.2 The concept of a file system

In computer systems, as we have discovered in earlier chapters, there is a concept of data storage, typically using disk systems. The hard-disk, and more recently the SSD, provide large scale storage of data in the form of many separate files. We also store files on removable devices such as USB memory sticks. These devices all look the same at the level of the file-system, even though the underlying hardware may differ.

In a file system, each file is a separate data entity. A file might be a document, a program, an image, a piece of audio, a spreadsheet, and so on: the possibilities are almost as limitless as the inventiveness of the programmer and user.

We might imagine that, on one level, every file is simply a sequence of binary values, or a stream of bytes. This is correct, certainly at the low-level structural state of data as stored on a disk unit. However, the arrangement of these bytes within the file dictates a particular file-format, and often there is more organisation to this than first thought might suggest.

However, let us stay with the idea of a file at a low level, as a sequence of bytes, just for the moment. Remember that a disk unit has a concept of storage organised into tracks, and sectors. Also remember that a sector is of a particular fixed size, such as 4096 bytes. Consequently, a single file must consist of one or more data sectors. If by pure good luck a file requires precisely 4096 bytes of storage, it might fully occupy a single sector. If by sheer bad luck a file requires exactly 4097 bytes, then it

must occupy two sectors, even though the second one is almost entirely empty of valid data.

Every file will therefore waste (on average) 0.5 sectors of data storage capacity, which is 2048 bytes for a sector size of 4096 bytes. We can reduce this wastage by making sectors smaller, but this means that the number of sectors on the disk becomes larger, and this has negative implications for performance and other aspects of file management.

Now, if a single file typically consists of a number of sectors, each of which could be anywhere on the disk, then we need a way to keep track of which sectors belong to a given file, and this is achieved using a **file allocation table**. And because there are usually thousands of files, then we also need to keep track of every one of those files and its associated sectors. The smaller the sectors are, the larger the number of sectors that need to be kept track of per file, and in total for the entire disk. This requirement is one of several which drive the need for a well designed file-system.

We can now perhaps begin to define what a file system is, and add a few additional expectations we may wish it to fulfil:

- ▶ It keeps track of the individual identity of files on a disk.
- ▶ It knows which sectors relate to which files.
- ▶ It conforms to a standard to make disks interchangeable across systems.
- ▶ It achieves a degree of efficiency, balancing competing requirements.
- ▶ It has some fault tolerance of disk problems.
- ▶ It (may) provide a degree of security assurance.

So a file-system has to respect **efficiency**, and also the concept of **interchangeability**. The latter point is important: If a disk is configured with a given file system, then it can only be read by an operating system that also understands that file system. This creates the expectation of a **file-system standard**, which allows many computers to utilise the same storage devices. For example, one of the most widely known file systems is the FAT file system, but there are numerous others. One of the problems with standardisation is that there are many standards! However, at least these are well defined and able to be reproduced by software engineers reliably when required.

The aspect of **fault tolerance** is also important. Since disks can be damaged, and may develop faults over time, the file system must be aware of bad sectors and ensure these are never used for storage. This is relatively

easy to achieve, and where there are huge numbers of sectors on a typical disk and rare faults, the impact on the user is usually unnoticeable. Other kinds of faults, such as mechanical failure, can be more catastrophic, but systems exist to try to mitigate these problems too.

We should also remember that the file system not only keeps track of where files are on the disk, but it also facilitates deletion of files and reuse of the space created by doing so. Keeping track of unused sectors is also important, then when a new file is created, the file system will know which sectors to use, and perhaps even which sectors represent the most efficient choices.

12.3 A file-system example

[150] This is sometimes abbreviated to FAT-FS. There are many implementations of FAT-FS, so this term is widely used.

The FAT file system^[150] is, as mentioned, a very widely known file system based on the concept of a **file allocation table**. There are a number of variants of FAT. However, they all have the same concept at their foundation. A file allocation table is simply a look-up table, typically stored on a reserved track of the disk unit, and containing information about every storage sector on the disk. In early systems, the number of sectors was limited to 65536 or even a puny 4096 sectors per disk, whereas modern FAT systems may typically support 4 billion sectors. One of the advantages of having a large number of sectors is that the sector size can be chosen to given optimal file storage and retrieval performance rather than being restricted by the need to keep the number of sectors low.

12.4 The boot sector

As mentioned in earlier sections, the idea of a boot sector, as a portion of the disk containing startup code, is a well established aspect of file systems, including FAT. This is an integral part of the file-system: when a disk is formatted, the disk is configured to conform to a given file-system standard, with certain sector size, and other parameters. In order for the disk to be bootable it must have a valid boot sector. This is set up during a process called **disk formatting**, which prepares a disk for use with a particular file format.

The boot sector is responsible for ensuring that the computer system can start up and begin loading the operating system, which is stored on

the disk using the chosen file format. Note that disks do not have to be bootable, in which case they can only be used as secondary disks, to provide extra storage space, but must exist alongside at least one bootable disk unit also installed on that system.

Note that a non-bootable disk can still contain applications, it just cannot initiate the startup process of the computer. Once the computer has started up by other means, it can happily run programs stored on the secondary non-bootable disk, should it need to do so.

12.5 Volumes and partitions

Sometimes it is convenient to have more than one drive in a system, though having several physical drives may be somewhat cumbersome. It is however possible to divide the total capacity of a drive unit into several subsections, known as **partitions**. Each of these can be formatted to appear as if it is a separate drive, and not necessarily using the same file system as other partitions on the same disk. It is possible for example to create two partitions, for example: one Linux OS, and one Windows OS, and have both of them capable of being bootable, giving the user a choice at startup^[151].

Meanwhile, we can invert the argument we have just presented. Rather than subdividing a drive into smaller partitions, we may want to create the impression that a drive is bigger than a single physical drive, in other words, aggregate several drives into a single entity. This can be done by using the concept of a **spanned volume**.

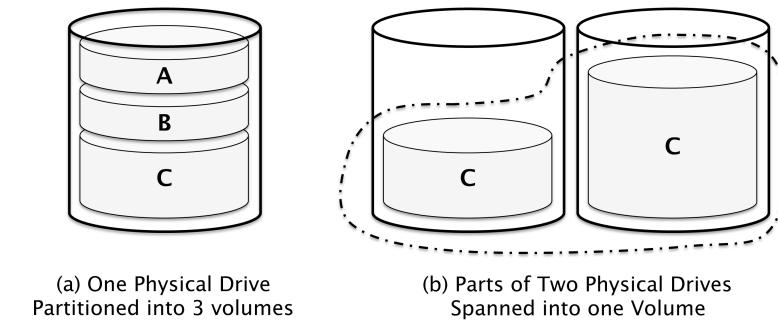
It is worth noting that when a volume is smaller or the same size as a single drive then it is no different than a partition in practice. It is normal to talk about partitioning a drive and configuring it to have several volumes. However, a volume can span more than one physical drive.

So, for example, two 4 TeraByte drives could theoretically be formed into an 8 Terabyte volume that looks like a single drive to the operating system, its users, and their applications. These are sometimes also referred to as **spanned volumes**. One of the benefits of file system and operating system functionality is that the complexities of these issues are hidden: all drives look the same at the file access level.

Examples of a spanned volume and a partitioned drive are illustrated in Figure 12.1, where we can see (a) a single physical drive partitioned into

[151] This is known as a **dual-boot system**, though they can have more than two boot choices. There are a number of software tools available to manage these kind of systems conveniently. APPLE computer users might be familiar with BootCamp utility, for example.

Figure 12.1: Examples of a partitioned drive, volumes and spanned volumes. Example (a) shows one drive partitioned into multiple volumes. Example (b) shows several drives spanned to create one larger volume occupying part or all of the two drives.



three non-spanning volumes, A, B, and C. We also see (b) a spanned volume where parts of two physical drives are combined to create a single larger volume.

12.6 The root directory

All file systems have a root directory. The root directory is the first place the operating system is able to look in order to find files stored on that disk. Simplistically, from the user point of view, this is where all of the files are stored. However, this would not be very convenient if this was all that happens.

Consider a system in which 10,000 customer records have to be stored as separate files, and at the same time the same computer needs to keep 300 spreadsheets with various accounts, stock records, employee records, etc. Add upon this the additional load of 1000's of operating system files containing kernel, code, utilities, and so on. Clearly, if these were all just dumped in the root directory in an entirely arbitrary fashion, things could get messy very quickly: files could get lost, accidentally deleted, mixed up, etc. To avoid this recipe for disaster, we have the concept of a hierarchical file system.

12.7 File-system hierarchy

To ensure some kind of organisational sensibility in the way files are managed, the vast majority of file systems employ the concept of file-system hierarchy. Let us consider again the example above, but impose some order out of the chaos.

Whilst the root directory^[152] notionally contains all of the stored files, it can contain them hierarchically. This is achieved by the concept of sub-directories. If root is the first directory, and we then create further directories within the root, then these will be known as sub-directories. A directory is simply a named container used to hold a group (a subset) of files belonging to the file system. Therefore, in our rather unwieldy example above, we might decide to reorganise things into directories as follows:

OS : the directory containing all of the OS related files. CUST : directory containing all customer records. EMP : directory containing all employee records. STOCK : Directory containing stock spreadsheets ACCT: directory containing account spreadsheets

It should be clear that this is much more sensible already. We can easily find particular groups of files simply by remembering the directory name in question. Files cannot get mixed up easily.

In some operating systems, it is possible to restrict access to certain directories. For example, only the person who maintains the computer system might be allowed to have access to 'OS', so that other users do not accidentally mess up the operating system settings. We could take this a step further, and subdivide the directory structure to another level. We might end up with a structure as represented by the diagram of Figure 12.2. Here the root directory contains three top level directories, and some of those directories contain sub-directories. In this way, file-system content can be organised into a nested structure, with any number of sub-directories that may realistically be needed for most purposes.^[153]

[152] Directory is one term, another more recently popularised term is 'folder'.

12.8 Special file cases

The file-system hierarchy allows for several special capabilities to be implemented, to create the illusion of a particular file configuration, when in fact that file is elsewhere. There are two particular cases worth mentioning at this point:

Symbolic Links: Some operating systems support the concept of linked files and directories. With symbolic links, it is possible to create the illusion of a file or directory existing in a particular place, but the reality hidden from the user is that this link actually points to another place in the system. This is usually done for convenience and consistency in file management arrangements in a given system.

[153] There are of course limits to all systems. However, for the vast majority of uses, this is unlikely to be encountered, especially as disk systems continue to grow in capacity.

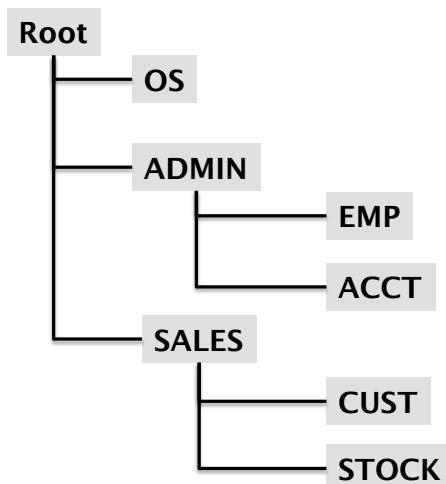


Figure 12.2: Example hierarchical file system directory structure. Where the Root directory contains three sub-directories, several of which have further nested sub-directories.

For example, taking Figure 12.2 as our model again, each of the accountants might find it easier to access the accounts if they had their own folders 'sam', 'joe', 'tina', and so on. Inside each of those folders a sub-directory exists called 'accounts'. However, 'accounts' is actually a symbolic link in this case, which simply links to: root/ADMIN/ACCT, or perhaps even links to a particular set of files that are the responsibility of that particular accountant. An example of this is shown in Figure 12.3

There are some advantages in doing this: for example, accounts can be stored anywhere, and moved around from time to time, and, as long as the symbolic links are kept up to date, the accountant never needs to know where the files are actually stored. They only need to look in their own local folder and the files magically always appear there.

Symbolic links are also necessary from time to time to manage various operating system tweaks that may result from various versions of software coexisting or running with different versions on a system.

Virtual Files: Where computer systems are operating in a networked environment, it is possible for files and directories to exist on one machine, but to be visible on another machine, via a network and suitable operating system enabled capability. In a similar way to a symbolic link, a file or directory on a remote system can be made to appear as if it exists in a local folder on the user's system.

Again, using our company example, this can be useful: the store room of the company may have four separate stock control terminals in different

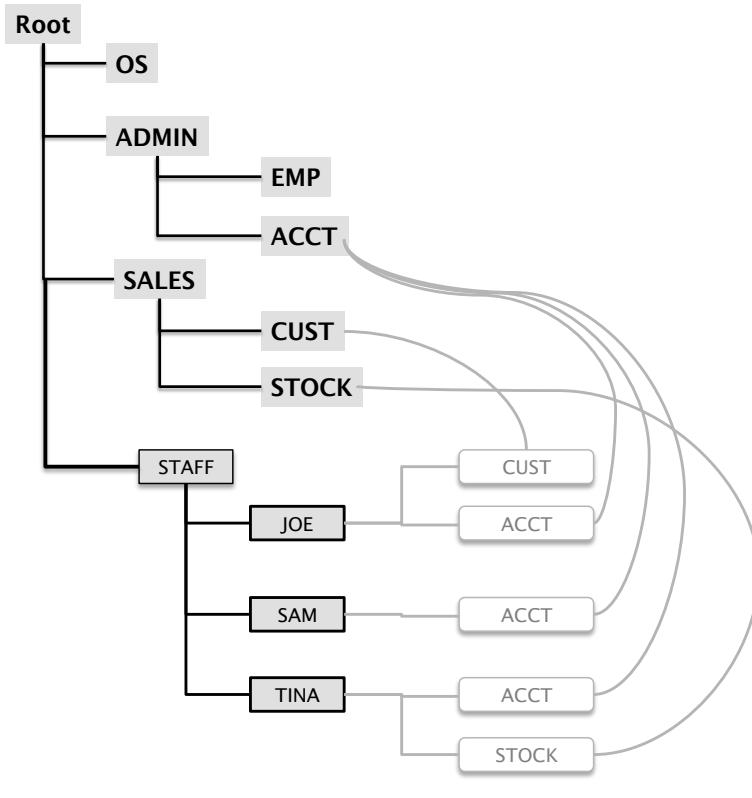


Figure 12.3: An example of a linked file configuration.
Joe, Sam, and Tina each see their own file space, but this is actually part of a larger file system.

parts of the warehouse. Each of these may see the stock files as if they are in a local folder, but in fact that set of files may be elsewhere, accessed via the network and looked after by a system technician at that location who never has to come down to the warehouse, or the four terminals, to manage those files.

12.9 File attributes

Most file systems support the concept of files having attributes. These help to control how files can be used, allow users and administrators to track such things as when a file was created, when it was last modified, last read, and so on. This and other capabilities make file management much easier.

Exactly which attributes are supported is operating system dependent, and these are not universal. Some typical file attributes are described

below:

Hidden: A hidden file is one that is not normally visible to the user. Hidden files can be viewed using particular commands, but not all users may have this capability. Often a hidden file is used to hide unnecessary information from general users, and prevent accidental changes to these files.

Executable: Some operating systems will only execute a program stored in a file if the file has been designated as an executable file. Attempting to execute a file that is not an executable could otherwise potentially cause a program crash. Also, a user that does not have the ability to change a file attribute to executable, is unable to write a program and then run it. This might provide additional security to that system.

System: A file that belongs to the operating system or kernel. Generally only a superuser can alter these files. This protects the operating system from being accidentally or deliberately modified.

Read-Only: A file that is read-only (or write-protected) can be viewed but not modified or deleted. If a user has the right to change a file to remove the read-only status then they can alter it, but only after changing the attribute. This provides a level of protection against accidental removal or overwriting of files even if the user has the right to do so.

Archive: Used in some operating systems to control the backup policy for particular files.

We can view file attributes using command line commands, an example of which is shown in Figure 12.4, or via a window-based file manager.

Some operating systems control some of these features by designating files as readable and writable. For example a file that is readable, but not writable, is read-only. A file that is not readable is effectively a hidden file as far as the user is concerned. The effect is the same, only the terminology is different.

Some operating systems are more sophisticated in file attribute management. For example, in Linux OS, it is possible to define groups of users, and to set permissions for groups to access, read, write, and execute files, according to their group status. So, for example, a group called 'HR' might allow all HR members to view staff record files, whereas 'employees' cannot.

```

chris — bash — 80x24
Desktop           Music          minheap.c
Documents         NetBeansProjects minheap.h
Downloads         Pictures        yimeg_exp2.scr
Google Drive      Public
Library          dynarray.c

total 48
drwx----- 4 chris  staff   128  8 Feb  2017 Applications
drwx-----+ 81 chris  staff  2592 15 Aug 10:19 Desktop
drwx-----+ 35 chris  staff  1120 18 Jul 08:57 Documents
drwx-----+ 37 chris  staff  1184 15 Aug 10:10 Downloads
drwx-----@ 32 chris  staff  1024 15 Apr 2016 Google Drive
drwx-----@ 71 chris  staff  2272 12 Jul 08:23 Library
drwx-----+ 3 chris  staff   96   4 Jan 2015 Movies
drwx-----+ 6 chris  staff  192  23 Aug 2016 Music
drwxr-xr-x  5 chris  staff  160  25 Feb 2016 NetBeansProjects
drwx-----+ 15 chris  staff  480  4 Mar 15:37 Pictures
drwxr-xr-x+ 6 chris  staff  192  8 Oct 2017 Public
-rw-r--r--@ 1 chris  staff  5647 22 Nov 2017 dynarray.c
-rw-r--r--@ 1 chris  staff  1015 22 Nov 2017 dynarray.h
-rw-r--r--@ 1 chris  staff  3414 22 Nov 2017 minheap.c
-rw-r--r--@ 1 chris  staff  579  22 Nov 2017 minheap.h
-rw-r--r--@ 1 chris  staff  707 10 Apr 2018 yimeg_exp2.scr

```

Figure 12.4: Example of file attributes. Example accessed via a Mac-OS command line, showing attribute flags on the left.

12.10 The variety of file systems

There is a wide variety of file systems in use currently, often frequently found being used alongside particular operating systems with which they have co-evolved.

- ▶ **FAT** File Allocation Table , FAT32 most commonly in use currently, very widely used and supported across many operating systems.
- ▶ **NTFS** New Technology File System, an evolution from FAT, which includes capabilities for much larger drives and files (FAT32 limits files to 4Gbyte), and also includes some fault-tolerance features to prevent data loss due to power outages etc.
- ▶ **APFS** APPLE File System, used on APPLE devices such as laptops and desktop machines, smartphones, and so on.
- ▶ **EXT** Extended File System (various versions Ext2, Ext3, Ext4 and so on), used with Linux primarily.

There are many others (tens, if not hundreds), too many to list in detail. Many of these were devised to work with fairly specialised operating systems, many of which are no longer (or perhaps never were) widely used. Since operating systems are often driven by proprietary or open standards, the most widely used are often associated with MICROSOFT Windows systems, Unix and Linux systems, and Mac-OS systems, and their relatives (such as IOS, Android, etc).

12.11 File compression

[154] Does this translate into better performance? This very much depends upon how much CPU effort is required to perform compression and decompression.

File compression is a technique that permits a portion of data to occupy less space when stored on a disk than it requires when loaded into computer memory for immediate use. This concept is widely used, both to reduce storage capacity of files, and also to reduce the amount of data transmitted by network connectivity, and in theory this can also reduce the delays required to transfer information around systems.^[154]

There are a multitude of compression schemes, some very specific to particular data types, whilst others are more general purpose, or certainly used as such. Within the scope of compression, there are also other factors: most importantly the ability to reconstruct the data content in its exact original form (**lossless compression**), as compared to a format that can only regenerate an approximation of the original data (**lossy compression**).

However, to be precise, when we consider file-system compression as it is found integrated into operating system and file storage practices, we are almost certainly talking about lossless methods, specifically to reduce file size in transmission and storage, but without altering the reproducible file content in any way. Data compression in a broader sense is slightly different, because it can apply any method to the original data, before it is stored, and could be lossless or lossy in nature, depending upon what can be tolerated.

[155] Lempel–Ziv–Storer–Szymanski (LZSS) compression algorithm.

[156] File archiving, using TAR or ZIP is first of all a method of collecting together groups of files into single entities. Most archive tools offer compression as an option (though it is nearly always the reason they are used).

12.11.1 File compression example

One of the the most widely used file-system compression methods is known as **Huffman compression**, often used in tandem with **LZSS compression**^[155]. The **ZIP** file format is a file archive format,^[156] with optional compression, which supports many compression methods and options within its range of capabilities (including Huffman-LZSS), and allows (with care) a compression outcome that is particularly good for a given file or data type.

An easy way to understand Huffman compression is as follows. Suppose instead of bytes, we have letters of the alphabet A to Z. Now consider that with 26 letters, and each letter being assigned a code, it will require two digits to represent each letter, (A being 01, B being 02, up to Z being 26). So the word 'EDEN' could be represented by 8 digits:

$$\mathbf{E} \ (05) \ \mathbf{D} \ (04) \ \mathbf{E} \ (05) \ \mathbf{N} \ (14) = \ 05,04,05,14$$

So far, all we have done is assigned an equally sized value to each character, and this offers no benefit. But now suppose that we observe that certain letters occur much more frequently than others in any piece of text, such that E,D,O,S,C etc are very frequent, whilst N,B,X,Z,K etc are much less frequent. In a particular Huffman scheme, the most frequent letters might require only a single digit, less frequent letters require two digits, and the least frequent letters require three, four, perhaps more digits. Consequently we might find that:

- E** might have the code **1**,
- D** might have the code **3**,
- N** might have the code **109**.

Now we can represent our word as follows:

$$\mathbf{E} \ (1) \ \mathbf{D} \ (3) \ \mathbf{E} \ (1) \ \mathbf{N} \ (109) = \ 1,3,1,109$$

We should be able to see that the new encoding requires only 6 digits, a saving of 25%. We can express a compression saving in another way, described as a compression ratio. We can define this as follows:

Definition 12.11.1 Compression Ratio (CR)

$$\text{CompressionRatio} = \frac{\text{OriginalDataSize}}{\text{CompressedDataSize}}$$

Using this formula we find that $\text{CR} = 8/6 = 1.33$. Any compression ratio greater than 1.0 represents a reduction in file size.^[157]

When this method is applied to data in a more general sense, it can be used to achieve significant data compression savings. Often the reason this is possible is that data contains many repetitions, and these can be encoded as frequent cases, using fewer digits to represent them.

It is important to note that there is a cost to using any compression method. It requires CPU time to compress, and then later to decompress such files. Also, we are actually reducing the total information content of the data by applying compression, even though its effective information content is still present. In other words, redundancy within the data representation is reduced by compression. But redundancy helps to improve fault tolerance. If a plain text file is slightly corrupted it may be possible to recover the meaning of the text:

[157] We have used decimal digits here for simplicity. If you prefer an example closer to a real hardware storage example, consider each letter initially as a 4-bit binary code, with values ranging from 0000_2 to 1001_2 , then consider what happens if we replace these with optimal length bit codes such as 01, 110, 11, etc.

Can you still work out what this says?

However, if a compressed text file is corrupted even by a single bit, then it will very likely decompress to a potentially complex mis-decoded and usually unintelligible file content.

12.12 Automatically compressed file systems

[158] This is also known as transparent compression. It may actually apply to an entire disk or to designated parts of the file-system hierarchy, depending upon the sophistication of the system employed.

Because there are some significant benefits in using file compression, some disk units and/or operating systems have been designed to apply this automatically to all files stored on that disk unit.^[158] This is in contrast to the user-directed file compression which is applied on an ad hoc basis by a user only as and when it is felt to be desirable.

There are considerations to be made here, some of which are competing against each other. Consider the following :

- ▶ Compressing a file requires CPU effort,
- ▶ CPU effort uses power and time,
- ▶ Compressed files are smaller so use less disk space
- ▶ Less data means less effort moving it about,
- ▶ Smaller files can be accessed faster and have less fragmentation.

So we see that actually, whilst enabling a compressed file-system to operate continuously will save disk space, and speed up file access, it will also take time to compress files when writing, and to decompress files when reading.

Remembering the estimates we made for disk performance, in Chapter 7, let us suppose for example that we have a disk system in which the average seek time to locate a file is 5 milliseconds, and the average file size is 100 sectors, taking 0.1 milliseconds per sector to read. Then the average file read time is 15 milliseconds: 5 milliseconds for seek time plus 10ms ($100 \times 0.1\text{ms}$) for sector reads.

[159] Compression Ratio is calculated as original size divided by new size. In this case, 100 divided by 60 = 1.66.

Now, suppose that by using file encryption the average file size can be reduced to 60% of its original size, (a compression ratio of 1.66^[159]), and assume also that the decompression effort when reading back the file requires 2 milliseconds. Now we can calculate average file time as seek time 5ms, 60 sector reads of 0.1ms each (6ms) and decompression time 2 ms, a total average access time of 13 milliseconds.

Interestingly, using compression, in this example, we have achieved an increase in file-system read performance of around 15%, whilst also reducing file storage requirements by 40%. In this example, as long as the compression effort takes 4ms or less (the **break-even point**) then we get the reduced storage requirement with no net penalty in speed terms, and potentially a speedup.

Clearly, there are issues here, in that an over-complicated compression algorithm might squeeze a few percent more compression from a file system, but with a large CPU time demand, the question of how much compression is enough is not easy to answer, but so-called **lightweight compression algorithms** appear to gain the best of high speed, low CPU demand, and reasonable compression ratios.

12.12.1 File encryption

Often, in the process of performing file compression, it is beneficial to also apply encryption. This is a technique which transforms the data content into a form, using a mathematical process, that can only be recovered by reversing that mathematical process when in possession of some key information^[160].

[160] Coincidentally, this is often referred to as the encryption key.

The precise methods of encryption vary, and there are multiple algorithms, which we examine further in Chapter 14. Increasingly, those encryption techniques are becoming more and more complex, in order to keep up with the increasing power of computers to break existing weaker encryption methods.

For everyday users, a file encryption is unlikely to be compromised within a short period of time. If data is required to be kept confidential for long time periods, then this is an issue that should be considered more possible. Files may have to be re-encrypted at some point with stronger encryption methods to remain secure.

Some disk systems also support inbuilt encryption, and this is seen in removable hard disks, and particularly in USB flash drives. Because flash drives are easily lost or stolen, they represent a security risk in some scenarios, and therefore using an encrypted drive makes sense.

No encryption is unbreakable, but the amount of effort required to break encryption for the vast majority of data files is simply too much effort for the gain that might be obtained in an average scenario. Only in highly sensitive areas such as governments, security services, etc, are there likely to be the resources to break such file encryptions in a reasonable time

frame. Even so, this time-frame might be hours or days of supercomputer time, and not minutes on a laptop.

12.13 File-system resilience

As mentioned earlier, the physical integrity of data can be protected, to a lesser or greater extent, by using suitable disk hardware configurations, such as RAID storage arrays. However, this is only part of the picture. Ensuring file-system resilience also requires additional aspects of operating system functionality and design.

At a lower level of kernel functionality, the file system needs to have features that ensure data integrity under unexpected conditions. For example, suppose a file is being moved from one place to another, and in the middle of that transfer, the power fails. What happens to the partially created file? What happens to the original file that we thought we were moving? With careful design, the file system and OS kernel can manage these scenarios in ways that allow errors to be recoverable in sensible ways that guarantee no data loss.

For example, a system where overwriting of a file is achieved by creating a duplicate, and only deleting the old version once writing has been completed and fully verified, will ensure that the worst that can happen is the latest version of the file may be lost, but not the previous version. A weakly resilient system might simply overwrite existing sectors of the file, meaning that if the system fails during the process, neither the old or the new file are intact.

This cautious philosophy has to be extended to the effects of virtual memory, file-caching in memory, cache modules within the hardware, and on the disk system, and so on. A fully resilient file system must ensure that any data written to a disk is flushed through to the physical disk as soon as possible, and any data not at that point is tracked so that errors can be flagged after something like a power outage, or component failure.

This is of course quite a difficult task to achieve in a watertight way. In most computer systems, the degree of resilience is a compromise between what is cost-efficient, good for performance, technically achievable, and also tolerable in terms of consequences. Most users know that if they pull out the power plug whilst typing in a document on a desktop PC, they will lose some data. But, they also expect systems to minimise the data

loss by using features such as autosave to regularly update the hard-disk content to be as close as possible to the most recent work typed in.

Meanwhile, the ability of a user (and by extension any task started by that user) to modify files, is controlled by the concept of file attributes introduced earlier. So, if a particular file is read-only for a particular user, then any program that they run will also have that restriction. This ensures that rogue programs cannot start to do unexpected things by overwriting or deleting important files. This ensures that all tasks within the operating system follow the rules (permissions) relating to the right to read, write, or execute the content of any file present on the system.

A further extension of this model is the idea of **root control** as a restriction upon a process or set of processes, whereby the process being started is given a **virtual root directory** in which its own local files are visible, but outside of which it can see nothing of the wider filesystem. For example, if a system had a true root directory containing sub-folders 'OS', 'Users', and 'Private', then a process can be configured to see the root of the system as any desired path^[161], such as **(root)/Users/user3**, meaning in this case that the process can only see files within user3 sub-directory, and not Private, OS, or other colleagues' directories that might exist in 'Users'. Essentially everything 'above' the point designated as virtual root is inaccessible. In this case, the process would be unable to even see the operating system files in OS, let alone do anything to the file content. This prevents deliberate or accidental process misbehaviour damaging unrelated file-system content.

12.14 Internal file formats

File formats are far too numerous to investigate in detail individually in this text. However, there are some basic concepts of file formats that are worth investigating further.

ASCII TEXT FILE: The **ASCII**^[162] text file contains a sequence of bytes, where each byte corresponds to a character, a letter, number, or other printable symbol, along with a few special characters (non-printable symbols). A text file can of course represent many things: the source code of a program, a list of items representing a record of a customer or employee, a simple text document, and so on. A text file can represent a series of words, parameters or commands that allow a data structure to be reconstructed. The format is entirely dictated by the program designer.

[161] This is often achieved using the Linux **chroot** command or equivalent in non-Linux systems.

[162] ASCII: American Standard Code for Information Interchange, a system where typically one byte is used to represent one character or symbol in a data stream or file. ASCII has been the dominant standard for many decades (since 1960 in fact) but is now being challenged by the newer and varied UTF (Universal Transformation Format) encodings, which were first defined in 1992. The first 128 ASCII codes and their corresponding characters are listed below.

0-31 Special Control codes.

32(space)	64 @	96 `
33 !	65 A	97 a
34 "	66 B	98 b
35 #	67 C	99 c
36 \$	68 D	100 d
37 %	69 E	101 e
38 &	70 F	102 f
39 '	71 G	103 g
40 (72 H	104 h
41)	73 I	105 i
42 *	74 J	106 j
43 +	75 K	107 k
44 ,	76 L	108 l
45 -	77 M	109 m
46 .	78 N	110 n
47 /	79 O	111 o
48 0	80 P	112 p
49 1	81 Q	113 q
50 2	82 R	114 r
51 3	83 S	115 s
52 4	84 T	116 t
53 5	85 U	117 u
54 6	86 V	118 v
55 7	87 W	119 w
56 8	88 X	120 x
57 9	89 Y	121 y
58 :	90 Z	122 z
59 ;	91 [123 {
60 <	92 \	124
61 =	93]	125 }
62 >	94 ^	126 ~
63 ?	95 _	127

XML: Extensible Markup Language is a file format that permits a wide range of textually readable parameters and data values to be specified within a file, which is also well defined in its layout in such a way that it can be consistently read by computer programs. An example might appear as follows:

Simple XML example

```
<diary>
    <meeting>
        <date>03.04.19</date>
        <location>Office4</location>
    </meeting>
    <lunch>
        <date>05.04.19</date>
        <location>Luigi's</location>
    </lunch>
</diary>
```

We can see with a bit of effort that a collection of items exists within a group called **diary**, and that these include different types of events such as **meeting**, **lunch**, and anything else we wish to add. A program reading this file would then need to know how to deal with each type of event. Note that the indentations are entirely optional, and are there for human readability.

Binary file: Any file that is a continuous sequence of bytes with no human readable context. The vast majority of file formats contain some binary structure, since this is the most efficient way to store data in compact form: for example, the number 123 requires three ASCII character codes (3 bytes) to be stored as text, but in binary the same number requires only 1 byte at most. A graphical image might be described by a sequence of lines drawn by an artist using a file format that supports descriptions of drawings. However, an image might also simply be a large array of data bytes corresponding to pixels within the image. The latter case is typically a binary file format. Some examples of binary file formats include **BMP** (bitmap), **PNG**, **JPEG**, **WMV**, **MPEG**.

12.14.1 File structures and encodings

As mentioned, file formats vary according to the file type. Usually the file format is indicated by the **file extension** portion of the filename (the

part after the main filename, after the dot). So **dog.bmp** is a BMP file type, and **shopping.txt** is a text file. Most file extensions are so well known that they are used universally only for that file type. However, it is worth cautioning that occasionally file types can be ambiguous because software developers have reused existing, less well-established, file extensions for other purposes.

A file extension is used to tell an application firstly if the file is one it can recognise and read, and secondly which type it is. Many applications accept multiple file types. A graphics editor for example, might desirably be expected to support all possible graphics file types, or at least the majority of widely used ones.

Usually a file has an initial portion of data called a **header**, containing essential information about the file, and this is then followed by the actual encoded file data, which we might call the payload.

A header block may tell the application which version of the file format is in use. A newer version of the file may have features that an older version does not support, or a feature that has been discontinued. It is generally necessary for software to support older **legacy** versions of file formats. It is not usually acceptable to just say 'sorry that is an old file type, it can't be opened'. This concept is also known as **backward-compatibility**.

Let us take an example: The **BMP** file format (known as the **device independent bitmap**), as defined by MICROSOFT. It consist of the following sections:

- ▶ **Header:** Basic information such as file size and other data. Also contains a space for application specific information.
- ▶ **Image Information:** A short block of data defining things like image dimensions, number of colours per pixel, and additional options.
- ▶ **Optional Colour Table:** a table defining colours used in the image if they are not the default ones.
- ▶ **PixelData:** A block of data containing 3 bytes for every pixel in the image.

All of these data values are encoded as binary data, so BMP is not human readable as if it were text. This is important, as pixel data can be compressed into a smaller number of binary bits than the textual equivalents.

The BMP file format has the option of algorithmic compression to further reduce the number of bytes that need to be stored. Many other file

formats also use compression algorithms within their file encodings in order to achieve data compression.

Every file encoding is ideally defined by a well defined and maintained standard, such that any developer wishing to make their software compatible just has to implement their code according to that standard.

Where developers deviate from these standards they create problems for interoperability which may arise in the future. It is therefore important that standards are well defined, updated by a single standard originator, and properly documented. Here are some of the most well known file formats:

- ▶ **BMP**: Bitmap graphics file for 2D images.
- ▶ **CSV**: Comma Separated Values, used to store arrays of data.
- ▶ **TXT, TEX**: Text file, usually ASCII text characters.
- ▶ **GIF**: Graphics Interchange Format, 2D images.
- ▶ **PNG**: Portable Network Graphic.
- ▶ **WAV**: Audio Waveform file.
- ▶ **JPEG**: Joint Photographic Experts Group, lossy compressive graphics file format.
- ▶ **MPEG**: Moving Picture Experts Group, lossy compressive video file format.
- ▶ **DOC**: Document file (typically Microsoft Word Documents).
- ▶ **XLS**: Spreadsheet binary file format (typically MICROSOFT Excel).
- ▶ **XLSX**: As for XLS but newer format that uses XML structure.
- ▶ **ODT**: Open Office Document.
- ▶ **EXE**: Executable file
- ▶ **OBJ**: Object File (intermediate file generated by a compiler).
- ▶ **ASM**: Assembly language file format.
- ▶ **C, CPP, java, py**, HLL program source file formats.
- ▶ **ZIP, TAR**: File archive, multiple files can be grouped together as a single 'tar' or 'zip' file, most often with the option of compression also applied.
- ▶ **PS**: Postscript, printed page content.
- ▶ **PDF**: Portable Document Format, as for PS.
- ▶ **SQL**: Structured Query Language (Database) file.
- ▶ **CGI**: Common Gateway Interface (script file for webservers).
- ▶ **HTML**: HyperText Markup Language (web page content).

We can find details of any of these formats via a web search, and can typically find a standard definition for version(s) of each file type, including detail of the internal structure of the file including header format,

payload sections, etc. A competent programmer should therefore be able to write a program to read or write any of these file formats with suitable documentation and patience.

12.14.2 File format tradeoffs: an example

As we have just discussed, some file formats are very human friendly, but relatively inefficient, whereas others are distinctly unintelligible without program support, but compact and more efficient for the computer system. An example may now help to highlight the pros and cons of these two approaches. Suppose a car dealership wants a computer file to record their car stock. They need to know the *model*, the *year*, the *mileage*, and the *price* of each vehicle. Let us look at how this might work in several different file format choices.

OPTION 1: BINARY Format

Each piece of information may have a fixed length, so we might allow the following number of characters for each item:

MODEL: 8 characters

YEAR: 4 characters

MILEAGE: 6 characters

PRICE: 5 characters.

Now, there are three cars encoded in the fixed length binary format:
(note . represents a blank space here)

```
FORD....2008..9223.5995CHRYSLER
2010100978.8000JAGUAR..2012.9923110995
```

You should be able to work out from the encoded content and the specification of the data fields (though not easily) that the car records are as follows:

MAKE	YEAR	MILEAGE	PRICE
FORD	2008	9223	5995
CHRYSLER	2010	100978	8000
JAGUAR	2012	99231	10995

This file format is reasonably efficient on storage space^[163], and you will be able to see that it requires 23 characters per car - a total of 69 characters. There are other advantages, for example a computer program can skip over car records by skipping over multiples of 23 characters without having to read the content of each entry.

[163] For the sake of simplicity of the example we have actually ignored a more efficient option, where each character of the file can represent an 8-bit byte. In that case, year and price only requires two bytes each (two file characters), and mileage requires 3 bytes. So the car record could use as few as 15 bytes per car.

However, if the record format is ever changed, the entire file format will need to be read and re-written. Consider for example if a new item is added, called 'colour', or if price had an extra digit added. This would require restructuring of the entire file. It may be necessary for a program to support multiple variations of the file format, each with new features. In order to support this it is essential that the file itself includes some information about which version the file format is, thereby necessitating a header section.

OPTION 2: Structured Text File

Suppose that we encode our car data in the following textual format following an XML style

```
<car>
    <make>FORD</make>
    <year>2008</year>
    <mileage>9223</mileage>
    <price>5995</price>
</car>
<car>
    <make>CHRYSLER</make>
    <year>2010</year>
    <mileage>100978</mileage>
    <price>8000</price>
</car>
<car>
    <make>JAGUAR</make>
    <mileage>99231</mileage>
    <year>2012</year>
    <price>10995</price>
</car>
```

With this more structured file format, there are several things to note:

- ▶ The file is human readable, and could even be edited manually as a text file to correct problems.
- ▶ The size of the data fields are arbitrary. They do not have to be the full length, indeed they can be longer without having to modify the file format.
- ▶ Fields do not need to be in the same order each time. Each data item is preceded by a tag (the item in brackets), so the program reading the file will know what it is reading, even if the order changes.

- ▶ New tags can be added to the specification, and not all cars need to have that data field if not relevant. (It is extensible and record content can be customised to each car).
- ▶ Earlier file format versions are easier to support, and new features can be given default values, where older 'legacy' file formats are being read in and such data features are not present.
- ▶ Because each record may be a different length, the program reading the file cannot simply skip over a number of records by skipping a fixed distance. Therefore, the file must be iteratively parsed (read entry by entry) to find a particular record.^[164]

This file format offers more flexibility, but at the cost of a larger file size. The tags waste space, though we could make them slightly shorter at the cost of less readability. If we count up the number of characters we find that this file format uses a total of 301 bytes or characters^[165]. This equates to an average of about 100 characters per car record.

Note that the programmer still needs to create a bespoke program to interpret this file format. However, if standard XML was used, then the XML file structure (which might look not that dissimilar) accommodates the creation of data fields and parameters and hierarchical structure automatically. It would then become possible for any XML reader to read the file and display the content even though the program might not understand what it means. Programming languages often come with XML function libraries to permit rapid development of file content management and formats in this way: this takes away the hard work and provides ready debugged program code to do the job of reading and writing the data format.

[164] For very large sets of records, this becomes impractical, and other methods are used. In particular, **SQL databases**, which organise and access data records in more sophisticated and efficient ways. In effect they reduce unnecessary use of slow data storage systems to give speedier access to data.

[165] This figure includes an extra non-printable character at the end of each line of the text file, which is not used in the binary file-format case.

12.15 File management

Many system users only ever use their file systems via a window-based user interface (such as MICROSOFT Windows, Ubuntu, Android, APPLE OS, etc). The kind of actions they perform on files are usually moving files from one folder to another, copying them, deleting them (and undeleting!), renaming, perhaps zipping and unzipping, and so on.

The more advanced user can use the shell or command line capability of the operating system to type commands directly into the system, and act directly upon files at a low level. In some respects, the capabilities available are much more varied and sophisticated in this scenario, though the operator needs a deeper system knowledge to make good use of it.

If you wish to be highly proficient in use of shell file management, then you must study the particular operating system in question and learn how to use the command line effectively. Often there are no 'undo' options with the shell, so you must know what you are doing. However, with sufficient expertise, it also then becomes possible to combine commands into text files and have these executed one after another (shell scripting).

Let us consider an example. In the Mac/OS command shell, for instance, the following commands will perform a given task:

```
cd fred  
ls  
cp file1 file2  
mv file2 ../  
cd ..
```

These commands perform the following tasks (in order):

CD changes the directory to place the command line context into the directory called fred.

LS performs a listing of the files visible in that directory.

CP copies file1 to a duplicate file called file2.

MV moves file2 to a new location in the directory level above the current one (which is back where we started).

CD moves the command line context back to the original directory (above fred).

So, in entirety, the sequence of commands results in **file1** in the **fred** subdirectory being duplicated, and that copy is then left in the directory where we started off. Clearly there is some learned expertise in doing this: knowing how to use each command allows them to be combined to achieve a particular task. Indeed, this is not the most concise way to perform the action described, we could simply type:

```
CP fred/file1 file2
```

A similar sequence might be achieved in MS-DOS, or Linux, though perhaps with slightly different command keywords and syntax. It is therefore also the case that using command line file management requires knowledge and experience of specific operating systems.

12.16 Summary

In this chapter we have explored the nature of files and file systems in a degree of detail that allows the beginner to appreciate most of the key considerations they are likely to come across. As always, some aspects have been simplified, and some deeper technical knowledge has been avoided.

However, what we should now appreciate is that file systems come in many varieties, and that the way that files are created and managed on disk systems is also quite diverse. File systems imply compatibility issues that need to be understood. The way storage devices are configured will dictate much about how the file system operates, and how efficient the file system is for a given use.

Just as the operating system kernel provides protections for tasks within the memory space, the file system also provides degrees of protection for users within the file space. Privileges, access rights, and file attributes can be used to constrain what users do within the system, and even to ring-fence their own regions of file space that are mutually exclusive of others. This is important for security and resilience, as well as for just making life easier for the users themselves.

12.17 Terminology introduced in this chapter

Archive file attribute	ASCII
Automatic file compression	Backward-compatibility
Disk formatting	Executable file
File allocation table	File header
File system	Hidden file
Huffman compression	Lightweight compression algorithm
Lightweight Process	Lossless compression
Lossy compression	LZSS compression
Partition (disk)	Read-only file
Resilient file system	Root control
Root directory	Spanned volume
Symbolic link	System file
Transparent compression	Virtual file
Virtual root directory	Volume (disk)
XML	

These terms are defined in the glossary, Appendix A.1.

Networks: Being Connected



13.1 Foreword	246
13.2 Putting modern networks in context	246
13.3 The OSI reference model	247
TCP/IP: Another example of layer abstraction	249
Security	250
13.4 Network structures	250
Bridging the gap	251
Switching things around	253
13.5 Routing	254
13.6 Network node identification	256
13.7 Network services	259
Sockets Interface	260
13.8 Distributed systems	261
13.9 A web-based client-server system	262
13.10 Cloud computing	265
Traditional in-house infrastructure	265
The cloud computing option	266
13.11 Compute servers and virtual machines	267
13.12 Virtualisation	269
13.13 Summary	272
13.14 Terminology introduced in this chapter	273

13.1 Foreword

In this chapter we will explore the concepts of **networks** and **distributed systems**, primarily from a systems and operating systems perspective. This means that we will focus on how networks and choices made in network configurations will impact upon performance, and how an operating system manages to support the ways in which networks are utilised. Having already given a brief introduction into network technology in Chapter 8, the aim here is to understand how these components are utilised in delivering services and capabilities with the aid of the operating system.

13.2 Putting modern networks in context

Networking in a modern computer system can cover a large range of possibilities. In the early days of computer networks, these were simply computer systems linked via some form of communication channel, radio, wired, or some other scheme usually operated by one company or organisation. If they were in the immediate area (perhaps the same or adjacent buildings), then this became known as a **local area network (LAN)**. If the network covered more distant locations, it would be known as a **wide area network (WAN)**. However, these days, networks can cover distances of a few centimetres up to the entire size of the earth^[167], and the user may not even know the difference in everyday use of their systems: the power of the operating system is that it can make a thermostat in your living room appear just as accessible as a temperature sensor at the north pole, with no discernible difference to the user. This is of course achieved with some quite sophisticated components of the operating system relating to network capabilities. We will explore some of these in this chapter.

The concept of using a remote computer may not be new, but in the 1960's the computer might have been a mainframe in the basement of a large corporation and the users might have been located in various offices above. Today the same office users may be using computing resources that are 100's of miles away, or even on another continent.

The idea of multiple users utilising one machine in a shared fashion, such that each user appears to have their own resource, is also not new. However, whereas users in the 1960s, 70s, and 80s were likely to see this in the form of multi-user access to particular enterprise applications via terminal systems, it is possible in modern systems to have entire

[167] On the small scale we have smart watches talking to phones in our pockets, and on the large scale even the sky is not the limit: the International Space Station (ISS) now has an internet capability.

computer system environments delivered to each user from a server or servers, and have them sharing the same resources without realising it. This is known as **virtualisation**.

For example, a single well-specified machine might simultaneously support numerous instances of a Linux desktop environment - one physical machine - but multiple users and independent desktops. Or it might run many web-servers configured as separate virtual machines, one per customer. This allows the customer to manage their resources as if they are logged on to a real machine in an office, but this is done remotely by a suitable capability, of which we will find out a little more shortly.

As a result of all of these new developments, it can be understood that networks and distributed systems are a very diverse subject, and many technologies exist. As in earlier chapters, we will gain an overview here, rather than an expert knowledge of every aspect, so that the interested reader can then travel on towards a fuller understanding through additional study and reading.

13.3 The OSI reference model

In order to understand general principles of networks we do not need to have a full understanding of the OSI reference model. Nonetheless, this is a well established definition which defines the principles of computer networks in fine detail, and obtaining a working knowledge of its principles should be helpful.

The Open Systems Interconnection (OSI) reference model, as illustrated in Figure 13.1, defines a network as having a number of layers relating to the software and hardware required to achieve a network. The lowest layer represents the physical network, which could be as simple as a piece of wire with two voltage levels, and the highest layer represents the application that the user interacts with in order to send or receive information via that network. In between these two layers are numerous other layers, which in theory at least, have particular purposes. In practical implementations of OSI some of these layers end up being combined into single entities simply because the simplicity of that network makes this convenient to do so. In other cases, the full OSI model is implemented in a modular fashion.

One of the most important aspects of the OSI model is the **transport layer**. This ensures that rather than a single connection existing, multiple data flows can coexist. It does this by interacting with the **network layer**,

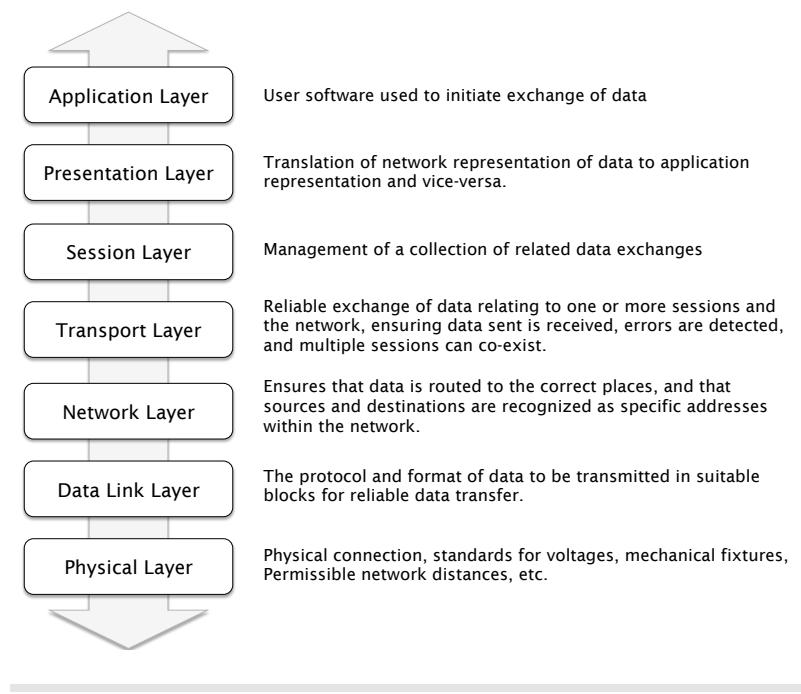


Figure 13.1: OSI reference model. Each layer within the reference model has a defined purpose, ranging from the physical network infrastructure up to the application layer. Each layer provides an abstraction away from the underlying complexity of the lower layers.

where packets are sent and received, on behalf of instances of the **session layer**, which connects to the user and their particular need of use of the network. So in fact, a live network may have multiple Application-Presentation-Session layer stacks, all talking to a local transport layer that manages multiple sessions. This is of course another instance of the concept of multiplexing of a resource, which we have encountered a few times already in other contexts.

Layers below the transport layer are responsible for ensuring that data is transmitted in manageable portions, using packets and frames, ensuring data is guaranteed to arrive after being sent, detecting and dealing with errors, re-sending data after faults, and making sure physical data values are transmitted in suitable ways according to agreed standards.

Because errors are dealt with in these intermediate layers, the user and their applications will ideally never see errors during operation: the lower layers will always detect and correct these before they become visible to the user or higher service layers.^[168] This allows a **reliable service provision** to be established and maintained. User applications can then rely upon this without having any knowledge of the underlying network technology or what is happening with the data being transmitted.

The network layer also ensures that data exchanges between machines

[168] Typical methods of error management include cyclic redundancy checks (CRC) and parity based error detection. We will discuss this in a little more depth in section 14.7.

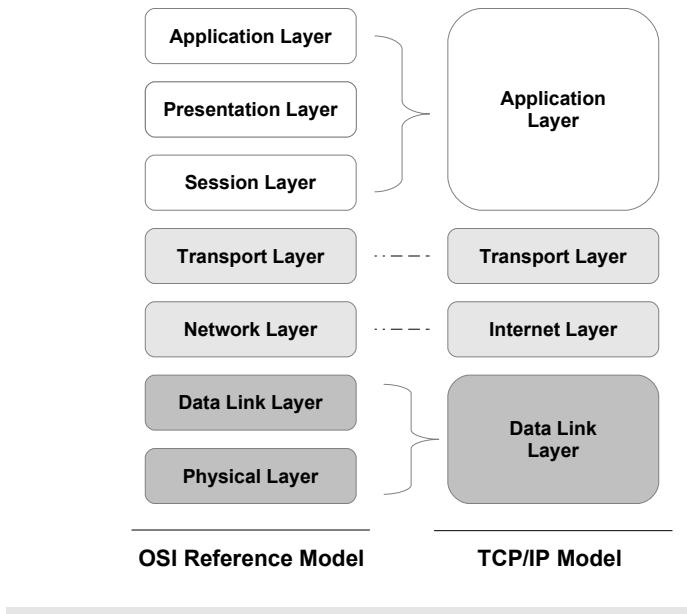


Figure 13.2: TCP/IP model
 The TCP/IP model has four layers, broadly equivalent to the seven layers of the OSI reference model.

are handled in the correct routing and addressing format, such that every device on the network has a unique network ID, and data is sent to the correct place to arrive at that machine.

The process of writing functional modules, relating to OSI layers themselves, requires quite a bit of knowledge of networking, and of the particular language and network functionality it provides. They have their own terminologies and details. Therefore, typically, a pre-existing network standard will be adopted, and suitable software libraries will be available for that purpose. These generally have the advantage of being well tested and regularly updated.

13.3.1 TCP/IP: Another example of layer abstraction

Whilst the OSI model defines a theoretical system of layers, each potentially distinct from each other, the practicalities of real implementations of such systems often result in some of these layers being merged. A good example of such a networking framework is the TCP/IP standard, often referred to in software terms as the TCP/IP stack. This model reduces the implementation model to four layers^[169], as shown in Figure 13.2.

TCP (Transmission Communication Protocol) and IP (Internet Protocol) are actually two standards operating together to form TCP/IP. The

[169] Although there are occasionally variants, with five layers for example.

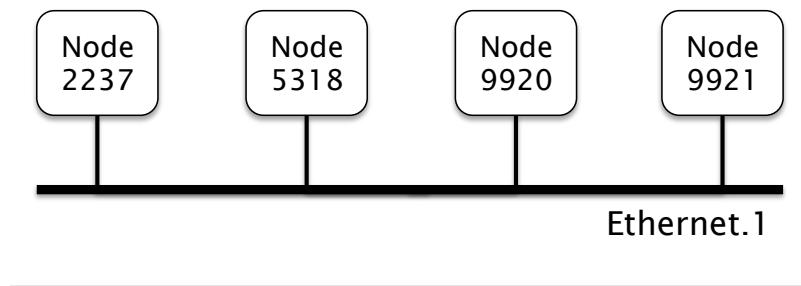


Figure 13.3: Simple network example. All nodes exist on a single network segment and share the network bandwidth.

TCP model was first demonstrated in 1974 and has evolved since then into one of the primary mediums by which modern computer systems communicate.

13.3.2 Security

Security and integrity of data transmission is another aspect that is managed within the OSI model, or equivalent layers of a networking model. We will delve a little deeper into encryption in Section 14.6. However, at this point it is worth highlighting that encryption is not a singular process. Taking the OSI model as a reference point, we can identify multiple levels of encryption at different levels of the OSI hierarchy. Applications can exchange data securely with their own encryption standards, known as **end-to-end encryption**, whilst lower levels of the protocol stack will support other forms of encryption at intermediate and lower levels, at the data-link layer for instance, even if applications do not do so.

13.4 Network structures

The simplest networks consist of nothing more than a shared wireless wifi channel used by multiple devices, or a single shared network cable, operating very much like a standard bus architecture. In these cases, all of the devices, or **network nodes**, are visible on a single physical network domain, and every node can, in theory, see every other node on that network.

An example of this simple case is given in Figure 13.3, where we see four network nodes, each with a unique identity. We will find out more about node identification later; for now we will just assume each node has a unique 4-digit number.

We can see that any node on this network can communicate with any other node on the network, simply by transmitting a data packet with the correct **destination address**. Node 2237 can communicate with node 9920 for example.

This is very simple to implement, but has drawbacks. Because the network is a shared resource, the nodes communicating with each other must compete for network bandwidth. As more and more nodes are connected to the same simple network, there is more and more competition for network bandwidth. If this continued, then eventually no single pair of nodes would be able to obtain enough network bandwidth to maintain their desired quality of service (QOS) level. In other words, this network has particular limitations on its **scalability** to large numbers of nodes.

13.4.1 Bridging the gap

A **bridge** is a network component, often described as a special component that can be used to join together two networks (or indeed two busses, as we discovered earlier). In some cases, joining two existing networks together with a bridge is exactly what is needed, and that is all good. A bridge will allow traffic to flow smoothly between two separate networks, appearing as if they are a single network.

However, we can turn this on its head and also say that a bridge allows us to split one network into two sections, or **network segments**. With this view, we can see how a bridge might well become a valuable performance tool for dividing network utilisation into manageable partitions.

Consider Figure 13.4, which shows our original network, but now split into two segments, named Ethernet.1 and Ethernet.2. Notice that nodes **2237** and **5318** are on one segment, and nodes **9920** and **9921** are on another segment.

The bridge will still allow traffic to flow from nodes on Ethernet.1 to nodes on Ethernet.2. So when node **2237** wishes to communicate with node **9920**, this will be facilitated by the bridge transferring packets between the two segments in the direction required.

However, if node **9920** wants to communicate with **9921**, the bridge will be able to recognise that those data packets do not need to cross the bridge from segment 2 to segment 1. Likewise, communications between nodes **2237** and **5318** do not require traffic to cross from segment 1 to segment 2. This process is known as filtering.

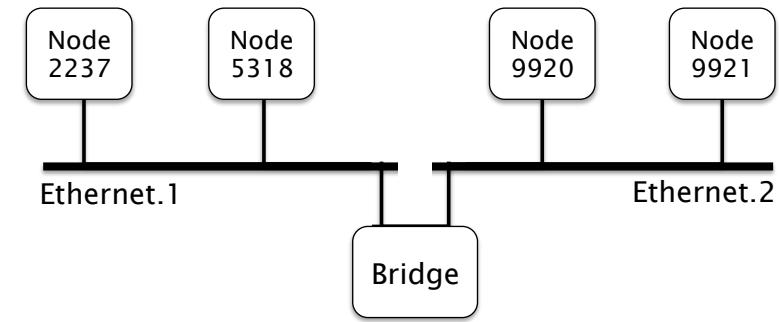


Figure 13.4: Bridged network example. The network is divided into two segments by a bridge, with each being able to be operated independently.

As a result of a bridge being able to filter network traffic, our new network now has two segments that can operate independently or jointly, depending upon the required node-to-node source-to-destination combination. When packets are not crossing the bridge, both segments can operate independently, providing twice the network bandwidth for the system as a whole. The implication is that, if we understand the traffic behaviour in a network, we can reconfigure it to maximise performance by using a technique such as bridging to create network segments that can operate concurrently.

Consider the following example:

Suppose our network has the following traffic flow during operation:

- (a) 2237 to 5318: 10 Megabytes/sec
- (b) 2237 to 9921: 1 Megabyte/sec
- (c) 9920 to 9921: 10 Megabytes/sec

The total traffic flow on the single network given on Figure 13.3 is simply the sum of the components, 21 Megabytes/sec.

[170] **Traffic volume** is how much we actually have happening on the network, as opposed to **traffic capacity**, which is how much we can accommodate.

However, on the segmented network as shown in Figure 13.4, traffic flows (a) and (c) can occur concurrently, because they involve different segments, whilst only (b) involves both segments and must operate cooperatively. The total is thus 11 Megabyte/sec for each network segment. But this does not mean we have 22 Megabyte/sec of network traffic volume^[170]. We have $10+10+1 = 21$ Megabytes/sec total traffic volume, because we cannot count (c) twice.

So, we can observe that in the traffic case given in the example, the network bandwidth on each segment required to perform the described service has been almost halved. If this is a service that is being provided

to end users, this means that we can deliver almost twice as much of this service using the new configuration when compared to the old one.

We can estimate network performance in a slightly different way. Suppose we have a 100 Megabyte/sec network bandwidth, which we then divide into two identical segments, and we find that 10 Megabyte/sec needs to cross the bridge. What will the total network capacity now be:

Network Capacity example

Each network segment has 100 Megabyte/sec available.

Each network segment cooperates in the 10 Megabyte/sec shared data traffic.

Bandwidth remaining on each network is therefore:

$$\mathbf{100-10 = 90 \text{ Megabyte/sec.}}$$

And then, total available traffic volume =

$$\mathbf{90+90+10 = 190 \text{ Megabyte/sec.}}$$

This represents a network capacity improvement of 90%.

So we see that the improvement depends upon the number of segments and the amount of shared versus independent network traffic per segment.

13.4.2 Switching things around

A bridge has a minimum of two network connections (called network ports), one for each side of the bridge. However, a bridge can have more than two ports, and such **multi-port bridges** are more commonly defined as **network switches** or **hubs**, depending upon their capabilities.

A **hub** is one of the least sophisticated network components: simply receiving incoming packets on any connected port and resending them out on all of the other ports. There is no filtering, it is simply a way to connect several devices or networks together reliably so they can communicate jointly.

A **network switch** performs a very similar job to a bridge in practice, but rather than simply acting as a middle-man between two network segments, it acts as a kind of interchange between three or more segments.

A switch will still achieve the traffic filtering effect described earlier, and will allow a network to be split into multiple segments. However, the

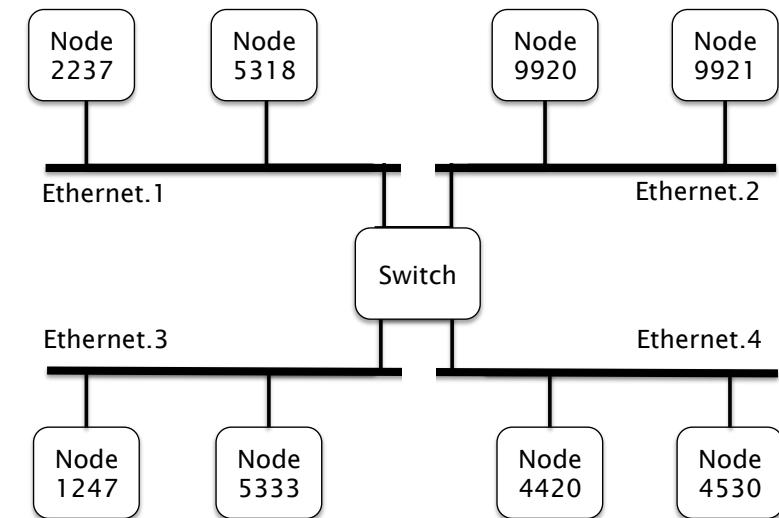


Figure 13.5: Switched network example. Where a switch allows many network segments to be linked according to the traffic flow between segments.

switch must know a little more about the network than a bridge, since it must decide which of its ports provides the segment where the destination node is accessible. It then switches those packets to the correct network segment using a mechanism involving a look-up algorithm based upon a table of destinations, or some other mechanism. An example of network with a switch is given in Figure 13.5.

Once multiple network segments are introduced, with multiple switches, it becomes possible for network segments to link to each other via more than one path. This might be to provide resilience to failures in network segments, or to increase available data transfer capacity, but it also adds to the complexity of the network switching strategy: there may be several routes that data can flow through to get from a to b, some faster than others. To manage this, more complex switching policies can be envisaged.

13.5 Routing

Routing is the process by which network hardware decides which of the multiple available network paths should be presented with network traffic from a source, in order to reach the desired destination.

In its simplest form, routing is already present to some extent in bridges and switches. In the case of a bridge, a packet appearing on one network

segment can either be routed across the bridge to the other segment, or simply ignored because it is already present on the destination segment (also known as a **subnet**). So filtering in this instance could be considered as a form of selective routing.

In the case of a switch, a packet arriving on one network port could potentially be switched to any of the other ports at that switch. The switch decides which port to send the packet to, depending upon its knowledge of which nodes and further 'downstream' switches are connected to each of the other ports and subnets.

The routing of packets can be based upon a fixed set of rules, known as a **static routing algorithm**. This has the advantage that routing decisions are fast and predictable. However, if links in the network fail, there is no fault tolerance, as the routing plan cannot adapt itself.

Another approach is **dynamically adaptive routing**, whereby information about the network is constantly passed to all switches in the network using additional packets, but at the cost of using some network bandwidth to do so. This information then allows switches and routers to make choices based on real-time network behaviour, for example, choosing between a faster route 'a' which is temporarily heavily loaded, versus slower route 'b' which has low traffic levels. Perhaps route b is the better choice at that particular moment? Dynamic routing is also potentially able to work around network faults.

The differences between a switch and a router are fairly esoteric for the every-day user. Since network switches and bridges perform some level of traffic directing, they already incorporate some level of routing within them. A dedicated router is generally designed to connect a local area network to a larger system, such as a connection to the wider internet. The methods of connectivity to the larger network environment can be via a number of options:

- ▶ Dial-up modem over analogue telephony infrastructure,
- ▶ ADSL Asymmetric digital subscriber line, an optimised version of copper-wire telephony infrastructure,
- ▶ Fibre channel connection, via optic fibre cable domestic comms infrastructure, using shared cable infrastructure.

Dedicated 'hard lines' and custom subscriber lines provide high bandwidth connections into the internet system, often private connections rather than being shared with other subscribers^[171].

A good example of this might be a 'big city' trading location, where that company requires high bandwidth, uninterrupted service, and no

[171] Telecommunications providers often share the bandwidth of a physical cable across multiple customers, leasing a proportion of the bandwidth to each customer.

variation in service quality. This cannot always be guaranteed if the line is shared by multiple other users with potentially conflicting needs for bandwidth throughout the day. Also, by cutting out intermediaries and extra hardware, such as switches used to share bandwidth on a common cable, the response time (latency) of the link may also be much faster, which could be critical where automated trading systems make million-dollar decisions on a timescale of seconds or fractions of seconds.

Another example might be a large organisation such as a university, where local telecomms infrastructure is not sufficient to meet the average or peak bandwidth requirements. A university could conceivably have 5,000 simultaneous users connected to the internet. Here, dedicated high bandwidth lines will be configured and leased specifically to connect to the main campus network access points.

13.6 Network node identification

The Internet is the greatest embodiment of computer technology yet realised. It is global, and almost ubiquitous. These days, even people in remote parts of relatively under-developed countries can gain access to some forms of internet connectivity via smartphones.

One can envisage a time where every individual has at least one computing device with a connection to the internet. Currently there are estimated to be in excess of 7 billion internet connected devices in operation, though the balance across the global population is not equitable. If this trend continues, we can imagine 20 or even 50 billion devices ultimately being simultaneously connected to the internet in our lifetimes.

In order for any device to be recognisable on the internet, or indeed even on a local network, it must have a unique identifier. This is often categorized as an IP address, or as a MAC address. There is an important difference to be noted here:

MAC Address (Media Access Control Address): This is a permanently assigned identifier that is preset by the manufacturer of every Network interface card and wifi controller, even those inside your laptop if you have one. In theory this code ensures that no two devices can appear to be duplicates. If this ever happens it may be because someone has deliberately forced a device to appear to be another device for some security-breaching mischief^[172]. MAC addresses, in their current form, consist of six pairs of hexadecimal digits, which provides 256^6 possible unique codes^[173].

[172] There may be other reasons - manufacturers not following the rules, or mistakes for example, so whilst they should be, we cannot safely say that every MAC address is unequivocally unique to one device.

[173] This amounts to around 281 trillion possibilities in total, so they are unlikely to be used up in the foreseeable future.

Internet Protocol (IP) address: This is a dynamic identifier, assigned to any device to allow it to be identified on the internet, regardless of the MAC address of the device. The IP address is assigned to a device that wishes to appear to be visible on the internet. The **IPv4** address standard currently consists of a 4-part number where each part consists of an 8-bit numerical field (256 values each). The IP address therefore supports 256^4 addresses.^[174]

It is interesting to note that the most widely used IP standard at present is the IPv4 model, which as mentioned above supports 256^4 addresses, giving almost 4.3 billion permutations. This numbering system has already been exhausted (after all, there are at least 7 billion devices using the internet already). The more recent **IPv6** identification scheme is beginning to become the new standard at the present time, with a huge number of possible permutations, equivalent to 256^{16} possible addresses.^[175]

As previously mentioned, IP addresses are assigned in a dynamic process. An assigned address may well be retained for a very long period of course, but it is not predetermined when the device is manufactured. Instead, an IP service provider, such as your cable company, will already have acquired a large block of IP addresses for its own use, and it will assign these individually to customers via their routers and server equipment.

Once a device is assigned an IP address, then in theory it can be made publicly visible on the internet. However, to do this conveniently, it is necessary to acquire a **domain name**, or a **subdomain**, and link it to the assigned IP address. When this is done, the relationship between the domain name and the IP address is stored in a globally accessible database, hosted by a machine called a **Domain Name Server (DNS Server)**. The DNS Server allows any textual internet domain name to be translated into an IP address for subsequent direct access.

For example, if a domain name was registered by a company with the name **www.fancydomainnameexample.com** then they (as the owner of that domain) could submit an IP address to the DNS database to create a link to this domain name. The IP address can be any valid address of a device on the internet that they own or have permission to use for that purpose. This is essentially how the internet allows us to enter web addresses (though more typically as URLs^[176]) into web browsers and arrive at a named page. If we actually already knew the IP address we could just type that in instead, and end up at the same place, though this is far less convenient, and offers nothing appealing in terms of domain name branding that we are now so familiar with.

[174] This equates to nearly 4.3 billion addresses.

[175] About 340 trillion trillion permutations (3.4×10^{38} potential addresses).

[176] URL, or Uniform Resource Locator, is a textual string of information consisting of the domain name, followed by any additional locality-specific information, such as the name of a webpage.

It is important to recognise that the IP address is not a permanent identifier, though they can be allocated and remain the same for long periods. The server that provides **www.fancydomainnameexample.com** may be assigned a particular IP address, and that might stay the same for several years. However, if the company then decides to move its domain name to another service provider, with another server, then the IP address may have to change. The users of this website will have no awareness of this, and will not need to be told anything. This is because the DNS lookup data is updated to point to the new IP address at the time of the changeover. This is the beauty of using domain names rather than IP addresses.

A further point to note is that within the internal constraints of a local area network, such as a university-wide network, machines may be assigned IP addresses independently of the global internet IP addressing scheme. This is only used where the machines are to be used internally in a local area network and not individually publicly accessible via a DNS lookup. To maintain some form of standard for this kind of private IP assignment, certain IP address ranges are reserved for this purpose, and organisations setting up IP addressing within their own 'closed' networks would be expected to use one of these ranges as appropriate.

For example, the following IP address ranges are designated as such:

**10.X.X.X
172.16.X.X
192.168.X.X**

Here, 'X' represents any number between 0 and 255 in each digit position. The X's represent the lower portion of the network IP address range (subnet) that is available for custom allocation. So a valid address could be allocated as **172.16.254.32** for example. Whenever a new user connects their computer to a local area network in a workplace, this process takes place dynamically via a system known as **Dynamic Host Configuration Protocol (DHCP)**, or is manually set by an IT engineer on each machine individually.

This scheme is also used for home IP address assignments, which are managed by the cable or broadband router in your home. For example, a laptop on a home network was found to have the IP address 192.168.0.46 assigned to it on the occasion that this was checked. However, on another occasion, after leaving the network and rejoining, the IP address may be different.

Some caution is needed when typing IP addresses directly into a web browser or using them in programming scenarios. For example, an IPV4 address consists of four numbers, but each of these numbers can be typed into a browser as decimal values or octal. Surprisingly, the IP address **192.168.0.46** is not the same as IP address **192.168.0.046**, because the leading zero in the final number tells the web-browser that this number is octal 046, and this is converted to its decimal equivalent, yielding an IPV4 address **192.168.0.38**. This may be confusing to those who aren't aware of this quirk of IP notation. Therefore, one should avoid using leading zeros in IP addressing unless it is intended to use octal notation.^[177]

[177] The Octal system has digits 0 to 7 (a base-8 number system), as compared to decimal (base-10), Hexadecimal (base-16), and of course Binary (base-2).

13.7 Network services

The concept of a network service is simply any capability that operates on a user's computer that facilitates some service via a network. Typically these are user applications that are initiated by the user when needed for a particular purpose, or are running on a remote server to provide some resource when a remote request is received. There are many examples of user applications in this class:

- ▶ Email systems,
- ▶ Web browsers,
- ▶ Remote file systems,
- ▶ Video and audio conferencing,
- ▶ Online gaming,
- ▶ Video streaming,
- ▶ Corporate enterprise systems.

All of these services rely upon some underlying operating system components, or additional modules that interact with the operating system. Some important and widely used components to be aware of are:

- ▶ **SSH (Secure Shell)** used to permit login on remote computers via the network, access to the command shell and execution of commands, and passing back responses to the initiator.
- ▶ **FTP (File Transfer Protocol)** used to facilitate exchange of data files between two systems on a network.
- ▶ **HTTP (HyperText Transfer Protocol)** used to support web-browser services, including queries sent by browsers and responses sent by web-servers.

- ▶ **SMTP (Simple Mail Transfer Protocol)** used to support email transmission and reception in conjunction with mail-servers.
- ▶ **POP (Post Office Protocol)** used to support email transmission and reception in conjunction with mail-servers.
- ▶ **CGI (Common Gateway Interface)** a mechanism by which servers and web-browsers can exchange data.
- ▶ **X11:** Also known as **X-windows**, an essential component for many remote graphical operating system applications. Along with equivalent systems, X11 provides the ability to view a window-based operating system and its applications remotely.

13.7.1 Sockets Interface

[178] Sockets Interface, developed in 1980's at University of California, Berkeley.

Many internet connection schemes now rely upon the idea of **sockets**, as originally developed in the 1980s^[178]. In this model, a service on a remote machine, **the server**, has a **socket ID** assigned to it when it starts up and initiates a connection. A program on the user's computer also has a socket ID assigned when it opens a connection and attempts to link to an application or process on the remote machine. Consequently, if both entities are successful, two machines will be connected, each with an IP address identifying the specific computer, and then a socket ID relating to the specific process or thread possessing that socket. In a sense this is not unlike a pipe between two threads as we encountered earlier, but on different machines in different locations.

For example:

Server 144.32.128.230:**1228** (Socket ID=1228)

Client 144.146.128.200:**3007** (Socket ID=3007)

The implication here is that the client process has assigned port **3007** as a socket for the user process, and the server has assigned port **1228** as a socket for the server's service-providing process. These are unique to those two physical machines and the two specific processes running on them.^[179] Remember that many programs can run on one computer and connect to many other computers doing the same thing. Therefore, it is not enough to simply have the IP addresses alone.

[179] In theory, the client and server could both exist as sockets on the same machine. That might well be useful for a diagnostic purpose for example, but not usual.

Certain socket addresses have reserved purposes (indeed, all socket ID's up to 1023 are reserved for special purposes). This means that connections can be made to machines knowing in advance what service this will be linking to (again, a bit like a named pipe in our earlier reading). For

any other connection, the operating system will assign unique socket IDs between 1024 and 65535 to any processes requiring them.

13.8 Distributed systems

When an application is developed which requires network capability, it is generally built on top of one or more of the services mentioned previously (or similar alternatives). When the functionality of the application is divided between two or more systems at different locations, this is known as a **distributed system**.

A very simple example of a distributed system might operate as follows: Joe has installed a small computer in his house to control the central heating system. The computer can be accessed by a control panel in the house, but it also has an **SSH** capability and runs an operating system that supports command line commands (like an MS-DOS or Linux terminal shell).

At the simplest level, Joe can just open up a command line terminal on a PC at work, activate an SSH connection to his remote computer, and then begin typing commands at the command line as if Joe is sitting at home in front of the machine. This is a basic level of remote access.

However, to make life easier, and the access to heating control more user friendly, Joe writes a simple application program, 'Heatcon', to run on his work computer. This invokes an SSH session on the work PC to connect to his home computer but displays a **graphical user interface (GUI)** to Joe. The application can then transfer commands captured by the GUI to the remote computer to turn the heating on or off, or change its temperature setting, mimicking the commands that Joe would otherwise have to type in manually.

Any responses from the remote computer, which would normally be read by Joe as text in the command shell, are read by the application and processed to understand what information is being sent back. If the application is sophisticated enough it can then convert this information into part of its application display. For example, a graphical temperature gauge could be displayed, driven by a textual response from the remote computer.

Joe has, in fact, implemented a simple remote service application: a remote central heating controller. When someone uses the Heatcon application, they only see a box pop up which requests a heating temper-

ature setting. This is the user interface (UI), or graphical user interface (GUI).

Everything else that happens, operates 'under the hood' in other words, it is hidden from the general user and is carried out automatically behind the scenes: all done via textual information exchange via an SSH connection. This means that a user does not need to know anything about SSH or command line commands at all, they can just point and click options in a GUI, and then see the responses.

After entering a temperature, and clicking 'ok', Joe's application sends the command to set the heating to temperature 'X' and then requests a report back of system status. The remote computer sends the status report back, and the Heatcon app displays this data in some useful way.

This example of a simple distributed system, as illustrated in Figure 13.6, needs two bespoke components to operate – the user interface application program on the client-side, and a utility program running on the remote computer (the server-side) that can send heat instructions to the heating system from the commands it is sent.

What we have described is a simple **client-server** network application with a local component (the Heatcon Client App) and a remote system (the server app). This is a nice simple solution. Many remotely maintained systems use this kind of approach. For example, a remote 3G cellphone tower might have computer equipment that can be accessed by an engineer remotely via SSH, allowing it to be reconfigured rapidly if a fault occurs, regardless of its location or conditions such as severe weather. To make it more efficient, an app might be developed to act as a client and allow common SSH tasks to be automated, and allowing the engineer to deal with problems more quickly.

One drawback of this model is that both the client and server software components may have to be modified and reinstalled whenever a significant change is made to the software at either end of the connection. In a system with many users, this could be a significant planning and implementation effort, and may require a period of **downtime** when the system is not able to operate.

13.9 A web-based client-server system

Whereas Joe's Heatcon application operates as a distributed client-server system, where two tangible components exist on different computers, and

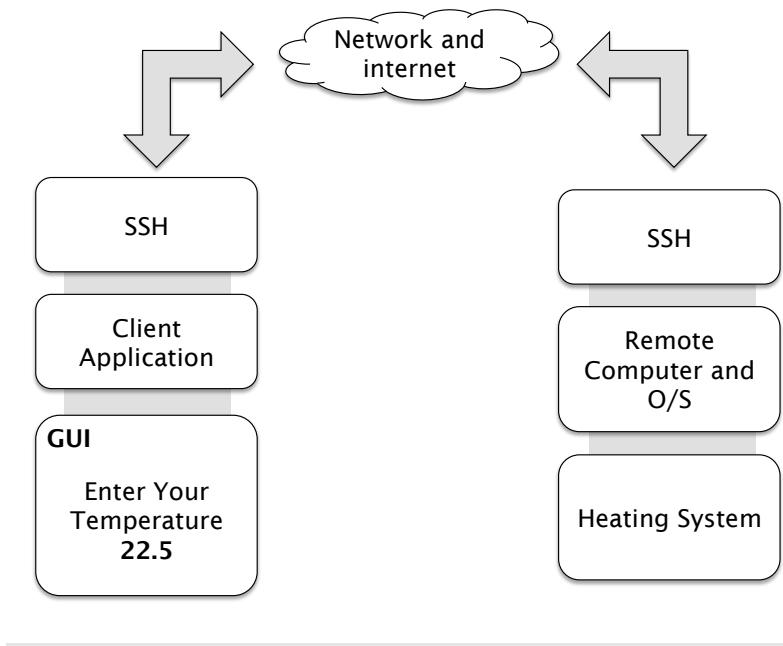


Figure 13.6: Joe's Heatcon remote central heating network application. Bespoke software is required on both the client and server machines.

cooperate to achieve the desired result, Joe could have implemented this system differently.

If the remote computer at Joe's house was set up as a web-server instead of accepting direct commands from the client application, it would be capable of receiving queries from Joe's work computer via a web-browser, and then reacting by sending back responses to be displayed and viewed as further web-pages.

At work Joe could type in a web-address and then see a web-page generated by the remote computer. It might say 'hello, the temperature at home is 20c, please select a new temperature'.

Joe can simply view this page and do nothing, or can select a new temperature via the webpage functionality and press send. The information is then sent back to the remote computer in the form of an **HTTP request**, and the remote machine decodes this message, performs the relevant tasks locally, and then sends a new web-page as a response to say it has completed the request. The new page might simply be a refreshed and updated version of the same page, so that it simply appears to update in response to the user action.

What is interesting about this system, as shown in Figure 13.7 is that there is no custom client component. Any web-browser can act as the

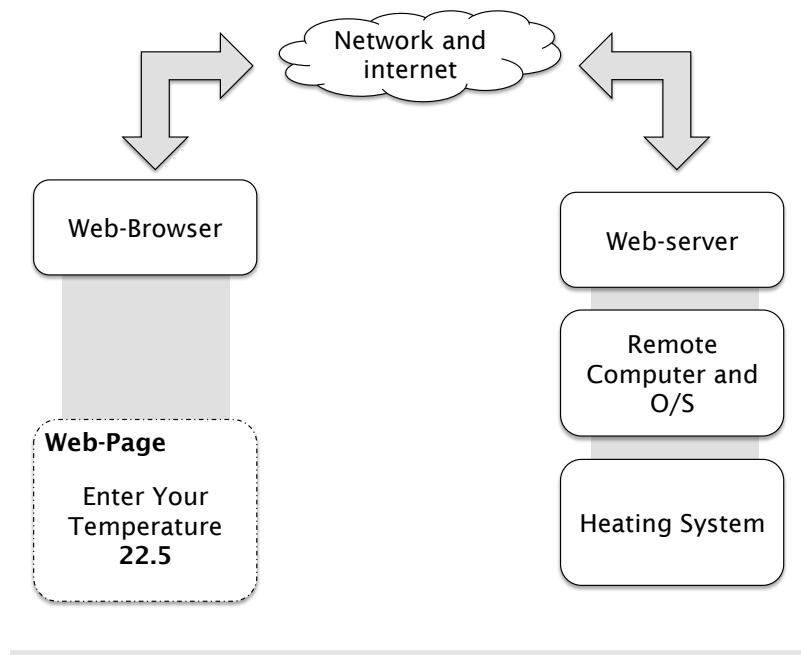


Figure 13.7: Joe's client-server Heatweb application. Only the server requires bespoke software. The client operates entirely via a standard web browser.

[180] Though it may not seem obvious, it is also (importantly) possible to just as quickly revert back to a previous server program version. If a problem arose after updating, then the system can quickly be '**rolled back**' to a known good previous state.

[181] Javascript is effectively a way of sending program code to a web-browser from a server to allow it to perform something more complex than point and click browsing.

access point to communicate with the server, but all of the actual work is done on the remote computer: it serves up the web-pages, interprets the http requests, performs the local commands, and generates new pages as responses.

A consequence of adopting this model is that any maintenance can be done entirely on the server side, and that program version can be changed over in a matter of seconds to an updated version^[180], meaning virtually no interruption of service is observed.

In the distributed system example, Heatcon, as shown in earlier Figure 13.6, the user application had some role in deciding how to respond to user choices (which commands to send etc), but with this new Heatweb system the client (the web-browser) has no knowledge of the remote system.

There are many programming languages specifically designed to deal with client-server scenarios. On the client-side, which is effectively just a web-page, the server could provide a web-page that contains Javascript^[181] for example. This would allow the web-page to have more interactive features and be more user friendly. But note that the GUI aspect that this provides is still being generated by the server: the client needs no prior knowledge or locally installed code to operate, provided that it has

javascript enabled in its browser settings.

Meanwhile, on the server side, a language such as PERL^[182] could manage HTTP requests and generate appropriate responses, whilst being able to initiate running of additional programs such as the utility Joe created to allow the home computer to send commands to the heating controller.

[182] PERL is one of a number of languages that are flexible and convenient in providing CGI (Common Gateway Interface) program modules to manage web-server requests and responses. Such CGI-scripts form the backbone of many web-server systems.

13.10 Cloud computing

Cloud computing is a term used to refer to distributed computing systems, and in particular it relates to the use of remote computing resources by a user or users, to achieve a particular computational outcome. This may take several forms:

- ▶ **Cloud storage:** Primarily a data storage and retrieval provision, used to provide reliable storage capability and/or large scale data storage, without the need for local equipment and maintenance.
- ▶ **Cloud computing:** Primarily a computational service provision, used to provide always-available scalable computing resources at low cost, particularly where a customer has a highly variable workload demand model.
- ▶ **Hybrid:** a combination of storage and compute services used together according to specific need.

Understanding the concept of cloud computing has much to do with the economics and logistics of deploying computer systems and services, and the nature of computer system end-user demand.

Consider, for example, a company that has a small number of staff but has a large data storage requirement, which is likely to grow over time. Let us suppose this is a medical image analysis and archiving service provider. What are the options for this company to provide for its resource requirements?

13.10.1 Traditional in-house infrastructure

One option available would be to have a server installed in their building (the in-house infrastructure option). However, there are drawbacks for this approach:

- ▶ **Cost:** The initial cost of the server might be quite high, skewing the costs of the business toward resource investment rather than day to day running costs.
- ▶ **Staff:** The company must now have a member of staff responsible for maintaining the server, upgrading the hard disks from time to time, to increase storage capacity. This may mean discarding perfectly good resources because they are upgraded.
- ▶ **Data backups:** The company must make regular data backups, and must also have a contingency to ensure that data is always available even if the server fails. Should they have two servers just in case?
- ▶ **Data resilience:** Perhaps it is not enough to simply backup the data? If the data is important enough, it must be backed up onto storage media and stored in a second location to guard against a fire at the main premises for example.
- ▶ **Data security:** Both the local server and the remote backups may need high security. Server security requires specialist IT skills: more staff costs, and remote backup storage may require a secure storage location and incur costs to lease.
- ▶ **Location:** Perhaps the company is one of the new breed of companies where staff are scattered around the country, working from home or at multiple geographical locations. A single server doesn't make much sense without a main office.

If a local infrastructure is preferred, then these are the issues that must be considered. However, there are other options.

13.10.2 The cloud computing option

Another option is to lease remote computer services. (the cloud computing option). Rather than buying and installing a server in the head office, the company leases cloud computing resources. How does this help?

- ▶ **Cost:** The initial cost of the server is zero, the company simply rents services (server time and capacity). It may be slightly more expensive day to day, but up-front costs are much lower.
- ▶ **Staff:** The company does not need any significant IT staff: the maintenance is all done remotely by the cloud service provider, and is factored into the cost.
- ▶ **Data backups:** Regular backups are standard for many cloud computing platforms, and specific backup options can be subscribed to when needed. Major cloud service providers will have distributed

backup capabilities, and that data may be backed up to multiple locations, perhaps not even in the same country. Loss of data is then almost impossible.

- ▶ **Data security:** No system is entirely secure, but a reputable cloud service provider will have security arrangements at least as good as any private company can manage via their own IT staff, and typically much better.
- ▶ **Resilience:** A cloud server could fail, but there are always second, third, and fourth machines waiting to take over, meaning system failures will be rarely noticed and have a short-lived impact.
- ▶ **Location:** The cloud platform can be anywhere, and so can the employees of the company accessing the resources: there is no physical location constraint to consider.
- ▶ **Upgrades:** The company only pays for what it uses. If its storage capacity increases, it will pay more, but it will not have to pay repeatedly for old hard disks to be scrapped and upgraded.

So there are many benefits from using a cloud-based infrastructure, much of them surrounding the consistency and reliability of the service provision, but also in terms of the cost, or the spreading of that cost over a time period.

13.11 Compute servers and virtual machines

The example described in previous sections was based upon the idea that a company needed reliable access to securely stored and backed up images in large volumes. There the main requirement was storage and retrieval.

Suppose that the company now analyses its working practices (its workflow model) and it is observed that each employee's main working activity (repeated many times per day) consists of:

- (a) loading an image from the server,
- (b) applying image processing on a local computer,
- (c) saving the new image and results back to the server.

Clearly, the storage and retrieval is an important aspect of this work, and is catered for by the cloud computing data storage and retrieval capability. However, it is also noted that the analysis techniques are becoming gradually more sophisticated, and the image sizes are growing larger as medical scanners acquire higher imaging resolutions. As a result, the computers used by the staff, and the network they are connected to,

are increasingly becoming a bottleneck: they are becoming too slow to handle the new requirements.

An option here would be to scrap all of the computers and buy new, more powerful ones. But perhaps this will simply mean the problem recurs in a year or two. What should the company do? Should they accept the cost of regular machine upgrades? Or buy very expensive high performance machines to stay ahead of the game for longer periods? Again this suggests all sorts of costs, and maintenance implications. And of course those computers will sit idle for 16 hours a day, when the offices are closed.

Again, cloud computing can provide a different kind of answer. Rather than attempting to perform more and more computation on local machines, the company can instead maintain a set of modestly specified computers for its staff, on their desks. Then, when an image processing task is to be performed, the staff member can run that image analysis task on a remote server (sometimes referred to as a **compute server**). In a sense, the staff desktop computers have become modern 'dumb terminals', like the VT100 terminal we mentioned earlier. There are some valuable advantages of doing this. For example, the remote server can make available many processors for a short period when needed. Instead of one CPU taking 5 minutes, 5 CPUs can do the work in one minute. The working efficiency of the staff is improved instantly as a result.

If the company recognised that cloud computing and cloud storage were both good solutions to their requirements, and even better when working together, they may even be able to organise an integrated solution with a service provider, such that the images are processed remotely without ever being transferred to the staff member, and only the results are viewed. This would result in a lower data traffic requirement: speeding things up further and potentially saving money once again on broadband connection bandwidth costs.

What we have explored in this case is an imaginary company with particular requirements, and the potential to optimise their costs and infrastructure, and reduce risks, by using cloud computing. We can imagine many cases where this could be applied, and of course a few where it may not be the best option.

For more on this topic, see CASE-STUDY ??.

13.12 Virtualisation

In the previous section we explored the idea of a server as a machine that can provide resources to multiple users apparently simultaneously, either as a local resource or a service delivered via the cloud infrastructure route. An important question here is how the user is provided with this service. If 10 employees all use the same server at the same time, how does that work?

In early computer systems, the multi-user paradigm was often managed by having time-slicing between users. Each user would get frequently switched portions of CPU and IO time. Where the operating system and the use of resources was very limited, this could be done with relatively low task-switching overheads. Another approach was for users to submit jobs to a queue and they would then be processed in some order of priority. However, that system does not allow for interactive use of the computer resource (whereas user time-slicing does).

In a modern computer system things are a little different. It is still possible for a server to be set up to accept jobs sent to a queue: for example, image processing requests generated by staff members in the company scenario we discussed. A computer server could be configured to do this quite successfully if there is frequent demand for repetitive processes, and the users are happy to wait for a response for a certain period.

On the other hand, a user may want to have an interactive desktop, running some particular operating system and applications, and an extended level of control over what it is doing. Likewise, cloud computing providers do not want to have a separate physical machine set up in their premises for every customer, especially as their use might be intermittent. In these scenarios, the idea of **virtualisation** becomes quite important as a solution.

Virtualisation is a concept whereby an operating system and user applications can run on an emulation of a real machine. Because a computer can potentially run several instances of these virtual machines (VMs), then one well-specified computer system (i.e. a server) can emulate a number of virtual machines at the same time. A server might support 8 virtual machines for example. There is of course a limit, based upon the performance characteristics of that server. With a particular amount of physical memory, and a particular processor technology, a particular set of disk units, there will be a limit to the number of virtual machines a server can support without noticeably poorer performance for the individual users of those virtual machines.

A virtual machine therefore looks just like a normal operating system including the windowed desktop if required. Here, there are two options:

- ▶ A user can connect to a virtual machine via SSH to connect to the command line of that machine and run suitable commands.
- ▶ A user can connect to the virtual machine via X-Windows (or an equivalent), which duplicates the desktop of that machine on the screen of the user's local machine.

In the case of SSH, the remote virtual machine looks exactly like a local command line computing resource. This might well be all that is needed for running long, complex computational jobs (simply run a script and wait), and this could be the basis of a job-queue. Equally, if the virtual machine is set up to run as a web-server, making web-pages available to internet users, then SSH command line access may be all that is needed for an administrator to alter file content and web-page configurations.

In the case of X11 X-windows and similar, the user will instead see a desktop window on their local machine, containing a live image of the entire desktop of the remote virtual machine. It will look and behave exactly like a normal desktop, though there may be speed and lag issues in the behaviour of the system if the local network and internet connectivity is not suitably high in performance. One particular advantage of this approach is that the user can access a much more powerful machine to occasionally run specialised applications, where a dedicated local machine of similar capability would be just too expensive to justify.

There are several ways in which virtualisation can be supported. Figure 13.8 shows some examples of these virtual machine concepts. Each concept relies upon an intermediary software agent called a **hypervisor**^[183], which coordinates the activities needed to share resources, emulate the virtual machines, etc.

Native Virtualisation: This allows a machine with a specific natively executed **hypervisor** component to provide multiple virtual machines, each appearing to be duplicates of the underlying system, but sharing the machine resources as if each user is the sole user of the machine. The VMs can each have different customisations such as user privileges and installed options etc, but are effectively clones of the same base system.

[183] Not to be confused with hyperthreading, which is a technique that allows multiple threads to co-execute on a processor core simultaneously: See Section 10.8.3.

Hosted Virtualisation: This uses a model whereby a native operating system has a hypervisor installed as an application, and this can be any third-party virtualisation software. This hosted hypervisor can then, in theory, support any virtual machine model, so a MICROSOFT Windows

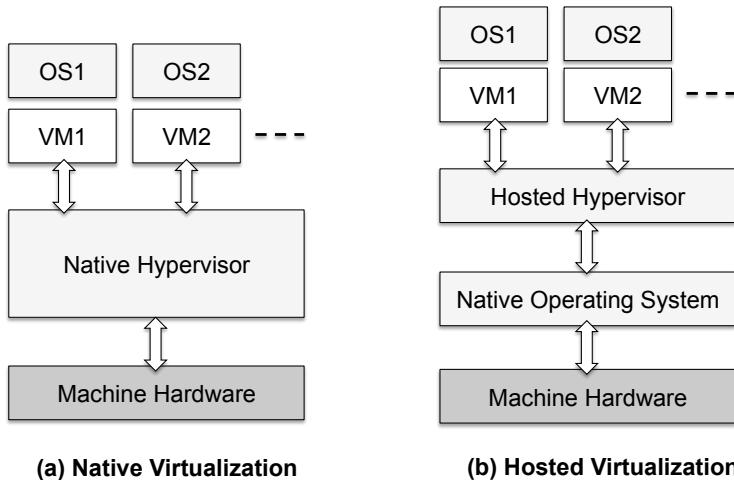


Figure 13.8: Typical virtualisation models. Showing (a) Native virtualisation, and (b) Hosted Virtualisation, where a virtualisation system is hosted as a layer on top of another operating system.

native operating system could host a hypervisor running a Linux virtual machine for example, or vice versa. Indeed, multiple virtual machines can coexist if the hypervisor supports this, and each VM can be identical or entirely different from the others. One of the consequences of this is that a server can be set up to support multiple users (different customers accessing a cloud resource for example) and provide a different resource to each user: different versions of the same operating system, or entirely different operating systems.

Local Virtualisation: Virtualisation can also be used locally, and not just to access resources on remote servers. For example, an APPLE Mac user may need to use Linux OS or Windows OS from time to time, and could do so by using a virtual machine hypervisor tool to allow these operating systems to appear to run on that platform.

This is frequently used where software engineers have to maintain software for multiple platforms, or where there is some legacy requirement to maintain software or to access a particular resource that can only be run under a given operating system. This is far more efficient than maintaining a physical machine configured with that setup just for those occasional cases of use, and removes the need to keep rebooting a machine with a different boot option each time in order to choose the particular setup required.

13.13 Summary

Networking is a topic that is often presented with a high degree of technical knowledge. However, the basic principles are fairly straightforward. Understanding how these relate to hardware certainly helps, however.

In this course material we have sought to explore the idea of a network as a hardware infrastructure, as well as a set of software capabilities and services. Hopefully, with the cases we have highlighted, and the performance issues we have explored, the capabilities and limitations of networked systems will be a little clearer.

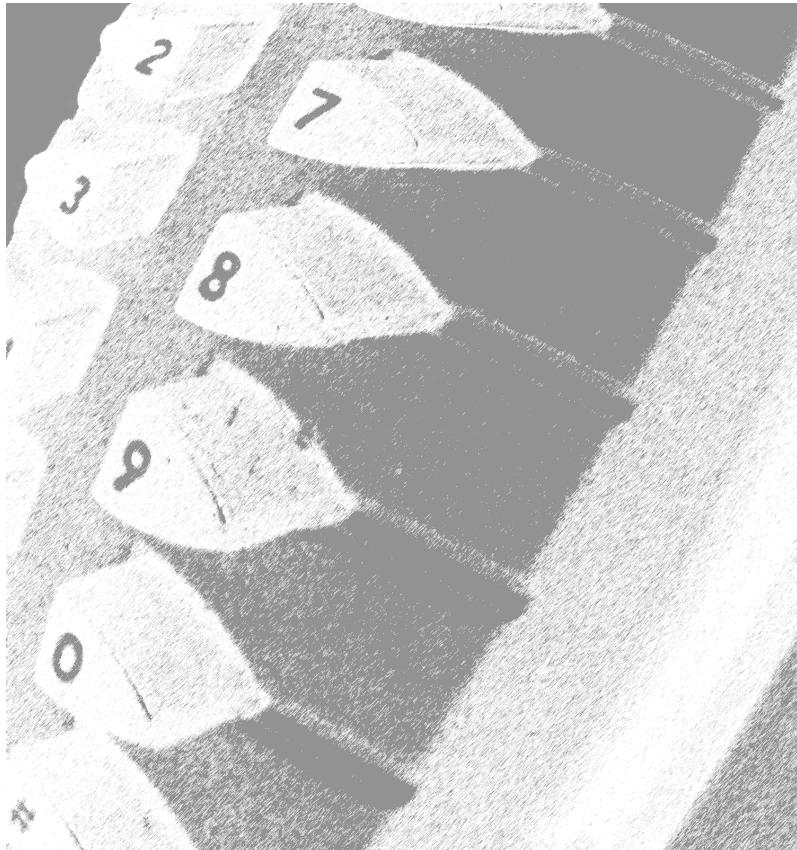
13.14 Terminology introduced in this chapter

Client-server	DCHP
Destination node	Distributed system
Domain Name	Domain Name Server
Dynamically adaptive routing	Graphical User Interface
Hosted Virtualisation	HTTP Request
IP Address	LAN
Local area network	MAC address
Multi-port bridge	Native Virtualisation
Network hub	Network layer (OSI)
Network node	Network routing
Network scalability	Network segment
Network subnet	Network switch
Reliable service provision	Session layer
Socket Interface	Static routing algorithm
Subdomain	Transport layer (OSI)
WAN	Wide area network

These terms are defined in the glossary, Appendix A.1.

14

Keeping It Safe



14.1 Foreword	276
14.2 Security threats . . .	276
14.3 Viruses	277
Virus checkers . . .	277
Virus examples . . .	278
14.4 Architectural ex- ploits	279
14.5 Firewalls	280
14.6 Encryption and valida- tion	281
14.7 Data Resilience . . .	282
14.8 Resilient systems . .	285
14.9 UPS systems	287
14.10 Resilience and safety	288
14.11 Summary	289
14.12 Terminology Introduced in this chapter	290

14.1 Foreword

In this final chapter of From Architectures to Operating Systems, we will explore the implications of computer security, with a particular focus upon the operating system roles and responsibilities, and reflecting upon what we have encountered in earlier chapters where relevant.

We will introduce some basic concepts and issues in computer security, identify some of the ways in which security threats impact upon computer systems, and also highlight how operating systems and their components can prevent some of these issues from arising or reduce their severity.

Of course some problems happen entirely by misadventure, and there are also issues that can arise due to entirely innocent occurrences. We will also consider some of these and look at mitigating strategies.

14.2 Security threats

[185] It is claimed that damage to an Iranian centrifuge facility, where uranium is enriched for nuclear projects, was compromised by an infected flash drive in 2010, by a virus known as stuxnet. This highlights the danger of employees using unauthorised devices at work when systems do not have suitable security blocks.

Security threats to computer systems can only happen when an unsafe entity is introduced to the system. If a computer system is never connected to the outside world, or any device that might be encountered (such as a flash drive)^[185], and only ever has trusted software installed, then security issues driven by exploitation of technology loopholes should be very rare, if not impossible.

However, as soon as a network is introduced into a computer system or users are able to introduce removable storage media, then a security risk emerges.

Security threats can take many forms:

- ▶ Data theft,
- ▶ Identity theft,
- ▶ Deliberate corruption of data,
- ▶ Denial of access to data,
- ▶ Denial of access to a service or system,
- ▶ Misappropriation of computing resources,
- ▶ Imitation
- ▶ and, and, and ...

Therefore, there are very many possible ways in which security can be challenged in a modern computer system, and even this set of categories can be expanded into hundreds of variations on those themes. We will examine this spectrum of threats in the following sections.

14.3 Viruses

Viruses are programs that are written to maliciously access or alter the behaviour of a computer system. As they are programs, then they must be executed in order for them to do damage. Once they are running they could achieve any of the classes of threat mentioned above as part of their purpose.

Commencing execution of a virus typically happens when a user is tricked into running a program, or when, for example, an application such as a web-browser has a security flaw that allows a program to run without the user even being asked permission. There are other means, for example an infected boot sector of a bootable device can execute virus code without the user even being aware or able to intervene. This is known as a **boot-sector virus**.

Closing off some of these opportunities will help to reduce the threat. For example, disabling the ability of a machine to boot from any device other than its own primary hard disk unit, will ensure that boot-sector viruses cannot be activated via a temporary storage unit left plugged into the computer. These settings are usually available in the BIOS configuration menu of the system in question. However, this still leaves the possibility of such software arriving via the internet during the machine's everyday operation by a user.

14.3.1 Virus checkers

Virus Checkers (anti-virus software) are programs that potentially inspect every single piece of information being transferred via relevant IO devices (those with a connection to the real world via networks or plug-and-play devices). By matching received data against a set of known data patterns (virus signatures), the virus checker attempts to spot viruses before they can start running. This is done in as streamlined a way as possible. The entire data is not checked byte for byte, rather particular patterns that match known viruses are checked for. This allows the laborious task to be speeded up, though it is certainly a drag on data transfer rates even given this optimisation.

Whilst this model works fairly well, a key problem is that new viruses are being written every day, and therefore virus checkers must be kept up to date with high frequency. Some viruses, known as polymorphic viruses, even use self-created mutations of their own code structure to

create duplicates that are not identical, making virus checking via known signatures more difficult.

Once a virus is running, it can do many nasty things. It can simply corrupt the data on the machine, causing serious problems for the user, but there are far more sophisticated things that may be done. Here are a few possibilities you might encounter if you are unlucky:

- ▶ It could corrupt file content (as mentioned),
- ▶ Encrypt your data and demand payment for its release (**ransomware**),
- ▶ It could simply sit quietly in the background and log keystrokes and attempt to steal passwords and credit card numbers (**key-loggers**).
- ▶ It could take control of cameras and microphones and record things without the user knowing.
- ▶ It could relay data files to a remote server (**data theft**)
- ▶ It could use the infected computer to send **spam** emails,
- ▶ and more ...

This can be annoying at the very least but potentially catastrophic: imagine an airline whose booking system has been compromised by a keylogger for a month without anyone noticing and all of their booking agents have been typing in the credit card numbers of their customers for that whole time. The customers could be defrauded, and the reputational damage to the airline, or even legal damages claims from customers, might be severe for that company.

14.3.2 Virus examples

According to a leading virus checker software vendor*, these (plus one or two others) are some of the most notable viruses that have been observed since the threat was fully recognised:

CryptoLocker: a ransomware virus that encrypts files on a system making them inaccessible. Files cannot be accessed without paying a sum of money to obtain the decryption key.

ILOVEYOU: spread via email attachments, but once activated, overwrites operating system files as well as corrupting user data.

MyDoom: Among other things, MyDoom took control of computers (usually without the user being aware for some time), and then used

* Sourced from NORTON “The 8 Most Famous Computer Viruses of All Time”,
https://uk.norton.com/norton-blog/2016/02/the_8_most_famousco.html

those computers to send excessive internet traffic to particular servers, taking them offline for normal users. (a **denial of service** attack).

Storm Worm: also spread by email, resulting in machines being taken over and used to send spam emails, including further spreading of the virus. When a computer is taken over in this way it is known as a zombie or bot infection. Storm Worm was a particularly widespread virus that gained user interaction via an email with the subject line of '230 dead as storm batters Europe'. When curious users opened the email, the potential for the virus to become active was initiated.

Sasser: This virus spreads by scanning remote computer network ports, and attempts to exploit security flaws in operating systems, that allow it to trick the computer into downloading the virus and then doing the same to other machines. Whilst the virus does not do anything 'nasty' to the infected machine directly, it uses CPU and network resources, sometimes to the extent that a companies infrastructure can be disrupted.

Stuxnet: allegedly designed by a national security agency as a cyber-attack against Iranian nuclear weapons development facilities. Iranian centrifuges used for uranium enrichment were targeted by messing up their operating parameters, causing equipment failures. It is suggested that this was transferred to the factory via an infected flash-drive whose user was unaware of its content.

14.4 Architectural exploits

The idea that a well-informed malcontent programmer could utilise their knowledge of the underlying hardware, and perhaps elements of the operating system, in order to exploit a security risk, has existed for a very long time. These are generically known as **side-channel attacks**. However, the sophistication of modern processor technology has released a new wave of opportunities for security loopholes to be discovered and exploited.

The **TLBleed** case is a particularly good example, using side-channel attacks to exploit the relationship between virtual memory and hardware and a component known as the **Translation Look-Aside Buffer** (TLB). TLBleed was developed in 2018, primarily to demonstrate that the flaw existed rather than for malicious purposes. This highlights the complexity of modern systems and how various components interact in ways that are very hard to evaluate fully for security loopholes.

The **Spectre** exploit is another good example. This makes use of branch-prediction and the effects this has upon the timing of certain subsequent behaviours of the machine, during speculative execution, in order to deduce useful information that should be entirely private to another task or process. Remember that processes are meant to be entirely self-contained and unobservable to each other. When any information can leak from one process to another, it can then conceivably be exploited for malicious purposes.

Even features of the hardware system such as DMA can be exploited. **DMA side-channel attack** is a method by which a device connected to a system can initiate DMA data transfers, and because the operating system is not involved, it cannot administer the normal memory privacy protections that apply at the software level. If a computer is not properly protected against this attack then a malicious person could plug in something when the machine is unattended, and access all of the computer's memory content, even if the machine is password locked. BIOS settings may help to prevent such activities from being initiated by plug-in devices.

Most recently, the role of hyperthreading in the mix of other issues that relate to these highly complex architectural exploits has led to hyperthreading being disabled on systems without suitable operating system fixes to alleviate these problems.^[186] The sheer complexity of architectures, is, in some ways, making absolute security of a system more difficult to verify, but also offering more and more exotic ways in which hackers can invade the integrity of our computer systems. It seems that there is no likelihood of this kind of problem disappearing anytime soon.

[186] This is more than just an inconvenience. The disabling of key CPU performance features can cause system performance to drop significantly, which could be a serious issue for digital service provision of all kinds.

14.5 Firewalls

Firewalls are filters built into key network infrastructures and/or operating systems that block certain types of data and limit the access to particular resources on the network to particular groups of recognised IP addresses, or perhaps deny access to particular addresses whilst allowing everything else. In theory a firewall can prevent a system from being accessed by a third party trying to access the network from an external connection. This prevents data theft and malicious attacks among other things. Some of these functionalities are often integrated into routers, whilst operating systems can have some of this functionality as standard.

Service attacks, particularly **Denial of Service Attacks**, are malicious behaviours that attempt to overload a server or network connection, such that normal users of the service cannot get their usual services. An online ordering system could be overwhelmed by randomly generated requests that confuse the system and overload the server, meaning that genuine customers cannot place orders because the system is going too slow or is just failing altogether. If this happens to an in-house server, there is not much that can be done except to try to block the offending IP addresses via a firewall so that they do not get to use any system resources by presenting fake requests, but automated attacks can quickly switch to other IP addresses, and therefore a determined attack is hard to mitigate.

With Cloud Computing services, the service providers are more geared up to preventing such attacks, and can switch in extra servers to meet the demand for example, to reduce impact whilst a better solution is found. Specialist software can detect such attacks early and try to automate the IP blocking processes. This is beyond the ability of most small companies, strengthening the case for the value of cloud computing in terms of resilience.

14.6 Encryption and validation

Although data theft is only one part of the security threat spectrum, it is a serious one. This is addressed as far as possible by using encryption. When data is sent via a web-browser, it can be encrypted to a level that most malicious agents would not be able to decode within a reasonable amount of time. It is not a question of *if* an algorithm can be broken, but rather: is the time required and the data obtained really worth the effort? There are multiple encoding/encryption standards in wide use currently, a few common ones are listed here:

AES: Advanced Encryption Standard

RSA: Rivest–Shamir–Adleman algorithm

DES: Data Encryption Standard

SHA: Secure Hash Algorithm^[187].

There are also many variations that use aspects of these encryption algorithms, adapting the main core standards that the most widely used encryption concepts are based upon.

Currently there is no cryptography method that is entirely secure, and which can also be used widely in computer systems. The best hope here

[187] Secure hashing, in its own right, is not necessarily able to deliver encryption, though it is often used alongside it. It might be better described as an encoding that allows data integrity to be validated for the sake of resilience or trust, without the data being able to be recoverable from the SHA itself.

is **quantum cryptography**, but this is nowhere near being deployable for everyday computing uses at the present time.

Because computer processing power is increasing year by year, encryption standards theoretically become weaker over time, as it is easier for an encryption scheme to be cracked with a less expensive computer (or more of them). Therefore, the encryption standards also increase in strength over time. A 64 bit encryption standard is weak compared to a 128 or 256 bit standard for example. Additionally, new nuances and variations are introduced to strengthen existing algorithms. The **3DES** standard is much more secure than **DES** for example.

However, this is also a double-edged sword. The need for more and more complex encryption also demands more and more CPU effort. Encryption is a burden on any system, and is unfortunately part of the cost and delay involved in sending and receiving data via a link that is capable of being observed by a third party. On top of which we also have to suffer the impact of constant virus checking efforts. In order to reduce this burden, modern processors may well have instruction set features and possibly specialised hardware units to accelerate encryption and take the burden away from the main processor. The downside is that some of these resources could also be exploited to accelerate attempts to break encryptions on a hacker's machine.

14.7 Data Resilience

A key aspect of computer systems, communications, the internet, and data storage, is the concept of data resilience. In particular we are concerned with the idea that errors at the bit-level can be detected and, ideally, corrected. We assume that technology deals with errors and consequently we rarely see these in the everyday use of our computer systems.

[188] This may seem like a problem exclusively for space systems engineers, and not everyday computing, but even at the altitudes of commercial jets, the rates of bit flip events increases rapidly (of the order of several hundredfold).

Whilst errors can be very rare, the reality is that no electronic circuit, wifi link, or storage device, is error free, and in some situations such as industrial computing or harsh environments such as space technology, nuclear facilities, and so on, errors are common. For example, in space, cosmic rays can pass randomly through memory circuits in a satellite on-board computer, causing bits to flip state^[188]. If that simply causes a very momentary glitch in an audio conversation on a satellite phone link, then this is no big deal. If, on the other hand, that single bit relates

to a multi-million dollar digital stock trading transaction, or a guidance system command, then it could be very serious.

Nonetheless, error detection and correction come at a cost, and the more resilient one wishes a system to be, the more effort is required. There are of course many ways to make systems more resilient. One method is to hand-shake important data exchanges. So for example, when a signal is sent, it is sent back and the original sender checks that the signal is still the same. Very much like reading back a delivery address when a customer orders a take-away meal. Not foolproof, but likely to greatly reduce the likelihood of an error. However, it is not always feasible to do this, and methods that allow the data receiver to detect errors independently of the data supplier are important. Some of the best known methods include parity checks and CRC (Cyclic Redundancy Checks), and often such schemes are built into hardware directly, such as the case of ECC error detection in DRAM SIMM and DIMM modules.

CRC and parity checking methods can also be applied in many contexts, including data storage, data transmission, and memory resilience, so they are not only found in relevant layers of the OSI reference model for example, but are built into the hardware of disk units and memory systems.

So what is parity? And how can it help to detect errors? Let us consider the example of parity checking as given in Figure 14.1. We can see that the simplest method is to apply a single parity bit to each byte.^[189] In Figure 14.1(a) we see the original data, with parity bits assigned to each row such that the total number of binary '1' digits in a row is even (known as **even parity**). The parity bit is set to zero or one in order to even up the count in each row. The data is then transmitted or stored along with the calculated parity bits.

In Figure 14.1(b) we see that if one bit is corrupted, then the parity bit no longer agrees with the even parity in that row and the receiver/reader of the data can detect this. This is the core principle of parity checking. However, it might be noted that if there are two errors in the same row, it is possible that they cancel each other out (Bit 'A' flips to a one and Bit 'B' flips to zero, for example). As Figure 14.1(c) shows, this can lead to the error not being detected. What we can say is our **horizontal parity check** is resilient for single bit errors, but not for all possible error cases^[190]. Of course single bit errors are typically far more likely than multiple bit errors, so we are at least addressing the most likely case.

[189] In this example we have one parity bit assigned for every 8 bits of data, but we could have any number of bits associated with a parity bit. More data bits per parity bit gives less resilience, as there are more ways an error can occur that one parity bit cannot detect.

[190] You might also have considered here that the parity bit itself could be corrupted. As a single bit error event that would still be detectable.

(a) Initial data with H-parity

Char	ASCII Data	Parity
H	01001000	0
E	01000101	1
L	01001100	1
L	01001100	1
O	01001111	1

(b) Single error is detectable

Char	ASCII Data	Parity
H	01001000	0
U	01010101	1 err
L	01001100	1
L	01001100	1
O	01001111	1

(c) Double error not detectable

Char	ASCII Data	Parity
H	01001000	0
Q	01010001	1 ok
L	01001100	1
L	01001100	1
O	01001111	1

(d) Multiple H-V Parity bits

Char	ASCII Data	Parity
H	01001000	0
E	01000101	1
L	01001100	1
L	01001100	1
O	01001111	1
		<hr/>
	01000010	0

Figure 14.1: Parity checking
Examples of use of parity to detect errors.

In Figure 14.1(d) we see that the parity system has been extended to include both **horizontal** and **vertical** parity bits. Typically this is applied to a whole data block of interest - e.g. a payload in a network data packet, or a disk sector, perhaps. You should be able to see that very few error combinations are undetectable. Repeating the error case from Figure (c) leads to vertical parity bits highlighting the errors for instance. Actually, it requires a minimum of 4 bit errors occurring in the same two rows and overlapping columns to create an error case that may be undetectable, and of course this has a very small probability compared to single bit errors, or multiple bits in a single row.

Of course we can make parity error checking more complex still, but often cyclic redundancy checking is used instead (this is a topic you may wish to research yourself).

Ultimately it is good to remember that no system is 100% error resilient, and there is always a cost. In our example message, there are 40 data bits. But with dual parity checking we have a total of 54 bits in this example. This means we have to transmit or manage around 35% more bits and thus get a corresponding reduction in data transfer or storage capacity.

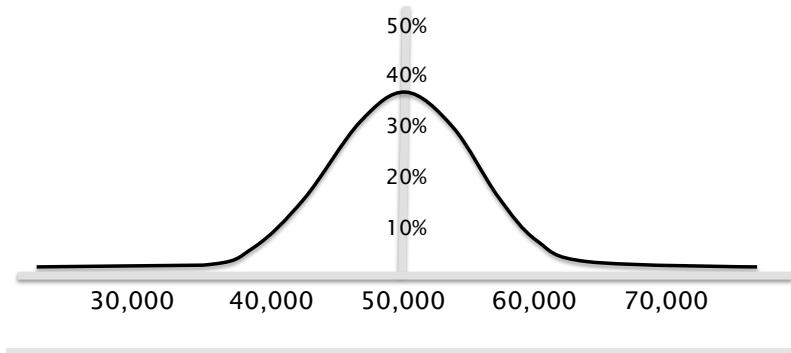


Figure 14.2: Gaussian distribution of some arbitrary hard-disk operating times before failure. The average MTBF is 50,000 hours, with a varying proportion of drives failing at lesser or greater timescales.

The trade-off between having more data bits per parity bit will determine how 'dense' the information is (i.e. transfer or storage capacity) versus how susceptible that data is to bit errors. This has to then be balanced against the severity of impact of a bit error remaining undetected - a matter that is very application dependent.

14.8 Resilient systems

Resilience comes in many forms, and some of these require complex issues for system designers to deal with. All system components can fail. The **Mean-Time-Between-Failure** or **MTBF** is a statistically derived figure that gives us an indication how likely this failure is.

If an HDD manufacturer rates their disks as having an MTBF of 50,000 hours of operation, then this is an average, not a guarantee. Given three drives, one may fail after 40,000 hours, one may fail after 50,000 hours, and one may still be working after 60,000 hours. In actual fact, with the quality of manufacturing, the vast majority of drives will operate within narrow band around the average, and this looks something like Figure 14.2.

A probabilistic curve carries on in both directions indefinitely in theory. It should be possible for example to find that the proportion of HDD units failing after 10,000 hours is, let us say 1%, for a given product. Therefore, in a system that has only one HDD of that type, there is a 1% probability that the system will fail within 10,000 hours.

Using some mathematics, we can work out the probabilities for various different scenarios:

Probability of at least one failure when n drives are installed

The key statement is **at least one** failure: this is the same probability as one minus the probability of no failures at all. And thus if $n=5$:

$$P_{\text{NoFailures}} = P_{\text{works}}^n = 99\%^5 = 0.951 \text{ (95.1\%)}$$

$$P_{\text{OneOrMore}} = (1 - P_{\text{NoFailures}}) = 1 - 0.951 = 0.049 \text{ (4.9\%)}$$

Redundant pairs

Suppose that each drive is duplicated to provide redundancy, and the system can continue as long as no combined pair fails. This means there are now 10 identical drives, and probability of any (i.e. one or more) failures is:

$$P_{\text{none}} = (0.99^{10}) = 0.904 \text{ (90.4\%)}$$

$$P_{\text{AnyFailure}} = (1 - P_{\text{NoFailures}}) = 0.096 \text{ (9.6\%)}$$

But the probability of a drive pair failing or working is:

$$P_{\text{PairFail}} = 1\% \times 1\% = 0.01 \times 0.01 = 0.0001 \text{ (0.01\%) .}$$

$$P_{\text{PairWorks}} = 1 - 0.0001 = 0.9999 \text{ (99.99\%) .}$$

And then finally, the probability of any pair failing is:

$$P_{\text{NoPairsFail}} = (P_{\text{PairFail}})^5 = 0.9999^5 = 0.9995 \text{ (99.95\%)}$$

$$P_{\text{AnyPairFail}} = 1 - (P_{\text{NoPairsFail}}) = 1 - 0.9995 = 0.0005 \text{ (0.05\%)}$$

So what do these scenarios actually tell us in practical terms? First of all, a system made up of various components, each with a failure rate of some percentage per year, period of hours, etc, will have an **independent failure rate** of the individual components. That is, the probability of any one component failing. If we assume that any one failure makes the system faulty as a whole then this is the likelihood that the system will fail: the **critical failure**.

On the other hand, we can reduce the potential for a whole system failure by introducing a degree of redundancy. If we introduced paired redundancy in the drive system, the overall chance of any one component failure increases (because there are more components), but the **critical failure**, which is now the possibility that **two** paired drives fail in the same period, is much lower^[191], and it is only the latter case that causes a whole system failure.

[191] Indeed, in our example cases, we found that the likelihood of a critical failure was reduced by a factor of about 100.

Furthermore, once redundancy is introduced, an engineer could immediately be alerted to the first drive failure in a pair and then very quickly replace it, and then we will never encounter a double failure: the possibility of a second failure within a few hours of the first, in the same pair, and the first being yet to be replaced, is minuscule. In this scenario it is arguable that the chance of critical failure is as good as zero in practice.^[192]

The redundancy effect is the basis of many fault tolerant systems, and indeed some RAID disk array configurations use exactly this principle. If the drive modules are hot-swap compatible^[193], then the engineer's work will be unnoticed, and nobody will ever be aware of the problem.

14.9 UPS systems

Resilience comes in many forms, and some of these are quite complex issues for system designers to deal with. One obvious issue that is often overlooked is the simple fact that without electricity, a system will fail catastrophically within moments, unless suitable mechanisms are put in place to deal with this.

The obvious thing to do here is to ensure that power is not interrupted at all. However, in practice, this is not likely to be feasible for most situations. The best that can be expected is to provide a short term reserve of power to be used for a few minutes whilst the system reacts to the situation, and follows a remediation procedure. Typically a system would aim to flush any uncommitted data held in memory back to disk, close any files, and stop any critical processes, to prevent data loss, and in the event of a safety critical system also place the system into as safe a mode as it is able to be designed to do.

For the general user, the Uninterruptible Power Supply (UPS) is a solution that allows a modest degree of resilience. These systems can cost a few hundred pounds for a small system, and may permit a computer system or small server to run for a short period of perhaps minutes. When the UPS detects a power outage, it sends a message to the computer to tell it that it must follow its shut down procedure. In professional enterprise systems and server farms, UPS systems can be extremely sophisticated, and needless to say, very expensive. They can potentially ensure that limited critical systems continue to operate for hours, whilst less important systems are shut down cleanly.

[192] However, this doesn't mean it should be ignored. In a large installation, such as a data-centre, we still need to predict failure rates to ensure enough engineers are available to do the work and that enough spare components are in stock. So we move from predicting failure to predicting maintenance demands.

[193] Hot-Swap components are components that may be plugged in and removed without halting or powering down the host system.

14.10 Resilience and safety

We have talked already about redundancy in terms of providing duplicate resources to reduce the critical failure rate. However, no system can be fault-free forever. When systems do fail, they may have to do so in a particular way, in order to ensure a safe outcome is reached. Such systems are often encountered where they manage highly critical requirements, and are often referred to as **safety-critical systems**. They can be found in many environments, including industrial, medical, automotive, and aerospace, to name but a few.

An example that is often talked about at the present time is the idea of a self-driving car. There are many obvious safety concerns here, if such a vehicle suffered a technical failure. Let us take the simplest task: ensuring the vehicle remains between two sets of white lines on a road. If this function is determined by a processor running an evaluation algorithm to analyse sensor data, camera feeds, and so on, then this processor is a critical point of failure.

If the processor were to suddenly stop working, then the system would crash: literally! But actually there are other worrying possibilities too. What if the processor continued to operate but started to generate erroneous results: the algorithm would cease to function correctly, and could potentially be even more dangerous than the first case.

In a system like this, a common solution is to introduce redundancy, but in a particular way. For instance, three processors could be installed to independently compute the same algorithm (**triple-redundancy**), and then the three processor outputs would be compared.^[194] Ideally all three agree, but if one processor becomes faulty and starts to disagree, the other two will still agree, and they will determine how to steer the car. This situation would also cause the system to move into a safe-mode, perhaps pulling over at the next safe place on the road within a 30-second emergency stop procedure for example.

This type of redundancy offers high degrees of integrity. The probability of two processors spontaneously failing within the space of 30 seconds before the car can safely pull over is vanishingly small. If this actually happened, it might well be due to some single cause: such as fire or electrical shorts in the control unit and not random failures. These are factors that can also be engineered down toward very low probabilities of occurrence.

[194] This is also often referred to as **triple modular redundancy** or **triple-mode redundancy** (TMR).

An important point was encountered here. When the car suffered a fault, we imagined it moving into a safe mode, reaching a position where it can shut down without danger. This is exactly what we wish to happen in such systems, and this is known as **graceful degradation**. If a sensor fails, the car may decide that it just has to reduce its maximum speed. If two sensors fail, it may have to decide to leave the motorway at the next exit to seek assistance. If a processor fault occurs, it may first of all reboot that specific processor and check if things have reverted to normal, if not then the safe stop-and-park mode is invoked. These are all scenarios that depend upon certain failures and their severity being understood and designed for within the whole system design, and which we will increasingly have to come to know and work with in the future world of computers.

14.11 Summary

We have learned that computer systems are vulnerable to many faults, both the kind that occur due to simple failure of equipment and components, but also those more deliberate acts of intent.

One field that never stands still is computer security. In this chapter we explored some basic security issues and ended up exploring some highly complex ideas such as DMA side-channel attack, which could only be understood with a knowledge of hardware that is often overlooked.

In a sense, many of the easy software-based routes to security loopholes have been thoroughly understood, and increasingly closed off as solutions are deployed.

However, the response to this - an increasing emergence of security exploits that use deep architectural knowledge, such as branch-prediction side effects, means that future security experts may well have to know a lot more about hardware than software to be fully prepared.

14.12 Terminology Introduced in this chapter

Denial of service attack	Encryption
Encryption algorithm	Independent failure rate
Key/logger	MTBF
Ransomware	Security threat
Side-channel attack	Virus
Virus checker	

These terms are defined in the glossary, Appendix A.1.

Appendices

A

Glossary of Terminology

A.1 Glossary of terms used in this book

Abstraction (OS)	Abstraction, in the context of operating systems, is the principle of removing the visible complexity of low level hardware detail from the programmer or application domain.
Address	A numerical value used to indicate a unique position within a memory device that typically contains a data item.
Address line	A signal line used to convey a binary value representing one binary digit of an n-bit address.
Address register	An internal storage element found within a CPU or other device, used to hold an address value.
Address-data bus	A group of bus signal wires used to represent address or data values in complementary clock cycles, in a multiplexed fashion.
Addressable locations	A quantity representing the number of locations within a memory that may be addressed by a CPU or other hardware device. Normally a function of the number of address lines available.
ALU	Arithmetic Logic Unit. The circuit within the CPU that is responsible for performing computational operations on data.
Amdahl's Law	A law defining the maximum possible speedup obtainable when part of a computational task is optimised.
API	Application Programming Interface. A set of software modules that permit an application to conveniently interface with a set of capabilities in hardware or in the operating system. See API.
Application Programming Interface	
Application-specific	Being designed and optimised specifically for a particular purpose.
Applications	Programs written to provide significant software and user functionalities, examples being word processing, spreadsheet, graphics editor, video player, web browser, and so on.
Archive file attribute	An attribute applied to a file to designate it as being archivable. See ALU.
Arithmetic Logic Unit	
Armature	In a disk unit, this relates to a lever capable of moving a read/write head across the surface of a disk area in order to place a head precisely above a particular recording track.
Artificial intelligence	The use of mathematical functions to mimic neural properties, such that a software system may appear to learn analytical capabilities.

ASCII	American Standard Code for Information Interchange. A standard set of numerical values used to represent alphanumeric and other printable characters.
Assembler	A program designed to translate textual representations of machine code instructions into binary values which can be executed directly by a microprocessor.
Assembly language	The textual representation format of a machine code program.
ATM network	Asynchronous Transfer Mode, a well known network standard which guarantees minimum levels of service for each data stream transmitted.
Attenuation (signal)	The reduction of signal strength of some or all signal components by some property of the transmission medium used.
Automatic file compression	A feature of an operating system or disk unit which performs data compression on files automatically.
Auxiliary bus	An additional bus within a computer system, supplementing the primary or main bus.
Available address range	The range of addresses supported in a given system.
Back-off-and-retry	An arbitration mechanism whereby two or more agents attempting to gain access to a resource simultaneously, and subsequently detect a conflict, will stop their requests and retry after a random period.
Backing store	Historical name for data storage systems.
Backward-compatibility	A system which maintains the ability to operate with older equipment, software, or files, even though newer versions are now supported.
Bare-metal system	A system without any operating system installed.
Benchmark	A program, or well-defined suite of programs, designed to perform a fixed set of activities on any system in order to permit that system's performance to be measured consistently and then compared with others.
BIOS	Basic input/output system, a small portion of code permanently present in a computer system to permit a startup procedure to be initiated prior to loading any subsequent software.
Bit-stuffing	A method of modifying the bit sequence of a data stream to ensure it conforms to a particular transmission requirement, but which can be reversed at the receiver to obtain the original intended data.

Bluetooth	A wireless communication standard designed typically for nearby communications of the order of metres to tens of metres.
Boot sector	A sector of a hard disk reserved to contain startup code, which all systems are able to locate and read by default.
Bootloader	A program found in the boot sector of a hard disk and which subsequently initiates a file system and operating system startup sequence.
Bootstrap program	See bootloader.
Branch	A program instruction which causes the current program counter position within a linear program sequence to divert to a new location in a non-linear fashion.
Branch misprediction penalty	The cost associated with failing to predict a conditional branch correctly. Typically refers to time, but can also refer to other related factors.
Branch prediction	The process by which a conditional branch can be predicted in advance of the processor knowing the correct outcome.
Broadcast	A process where data is sent to multiple destinations simultaneously.
Burst-Mode	A protocol used in DRAM memory devices, in which consecutive locations may be read in quick succession, providing higher data read throughput.
Bus arbitration	The process by which a bus device is able to request and be granted ownership of the bus.
Bus bridge	A device acting as an intermediary between two busses of similar or differing bus standards.
Bus hierarchy	A collection of busses operating in a tiered structure.
Bus master	The device currently in possession of the bus.
Bus protocol	The rules and operating sequences necessary for a bus to perform its required operations,
Bus slave	A device being accessed via a bus, under the control of another device (the bus master).
Bus standard	A set of definitions pertaining to a bus, including its bus protocol, and other related issues such as electrical standards and connectors.
Bus transaction	The operation of a data transfer via a bus.
Bus utilisation	The amount of bus capacity currently being used, typically measured in percentage or as a value ranged from zero to one.
Cache	A small fast memory provided to supplement the slower main memory, thereby boosting memory performance.

Cache coherency	The state in which the content of a cache and the content of main memory are the same (coherent) or different (incoherent).
Cache hit	An event whereby a memory access is able to be facilitated by a cache, without needing to access slower main memory.
Cache line	A group of locations held within a cache relating to a particular portion of memory.
Cache memory	See cache.
Cache miss	An event whereby a memory access fails to be facilitated by cache and then has to be serviced by the slower main memory.
Cache replacement policy	A mechanism whereby existing content in a cache is chosen to be replaced by new content, in order to ensure the most important data is held in cache.
Cache write-back	The process whereby an item in cache is only written back to memory during a replacement event.
Cache write-through	The process whereby a memory write to cache is also forwarded to memory, eliminating the need for a later cache write-back.
CAN Bus	An industrial and automotive bus standard used in environments where resilience to noise and interference are important.
Capacitive bit cell	A circuit using a capacitor to retain a charge equivalent to a one or zero in binary signal terms, part of a memory.
CAS	Column Address Select, a signal indicating that the address bus contains a column address.
Client machine	A machine used locally by a user to access a remote service.
Client-server	A remotely located machine set up to provide services to clients.
Clock cycle	A single clock transition sequence, consisting of a zero-one-zero cycle.
Clock period	The amount of time taken for one clock cycle to complete. Numerically the reciprocal of clock frequency.
Clock driver	A circuit designed to generate a precise on-off clock signal repetition at a designated frequency.
Clock signal	A signal consisting of a constant repetitive on-off cycle.
Clocks per instruction	The number of clock cycles required for a single instruction to complete its activities.
Cloud computing	The use of remote computing resources to provide services.
Cloud storage	The use of remote computing resources to provide data storage capabilities.
Column address	One half of the row-column address field of a memory address.
Column-Address-Select	See CAS.

Combinational circuit	A circuit consisting of individual logic gates connected to achieve a more complex function.
Command line	A textual interface allowing a user to give textual commands to an operating system and receive textual responses.
Compiler	A program that translates textual program code (source code) of a high level language into machine code executable by a computer.
Compute server	A remote machine set up to provide computational services.
Computer memory	The circuits within a computer system that retain information.
Concurrency	The simultaneous operation of two or more activities.
Context switch	The instance where a machine moves its frame of reference from one state to another.
Contiguous file	A file which is stored on a disk in such a way that its component parts (sectors) are adjacent and in the correct sequence.
Control signal	A signal performing a function of controlling a particular event in a computer system.
Controller delay (disk)	A delay associated with the time required for a disk unit and/or interface to determine how to service a disk read/write request.
Cost-benefit analysis	The evaluation of a positive outcome balanced against any penalty incurred.
CPI	See clocks per instruction
CPU	Central Processing Unit. The processor, the circuit that performs all of the primary computational actions of a computer system.
Cpu socket	A connection on a motherboard designed to accommodate a processor.
CPU time	A measure of CPU time utilised by a particular task.
CSMA/CD	A protocol used in Ethernet networks.
Cycle time	The time taken for a cycle of some activity to complete. Often used to mean clock cycle time.
Daemon	A task responsible for providing some persistently available service on demand within an operating system.
Daisy-chaining	Linking successive devices together with intermediate connections,
Data bandwidth	The capability and capacity of a component, typically a bus or network, to convey data at a particular rate.
Data cache	A cache specifically reserved for caching data content of memory.
Data capacity	The amount of data that a device can accommodate, typically a storage device.

Data hazard	A conflict in a succession of instruction execution cycles, whereby one instruction relies upon the data values generated by a preceding instruction, and may then be delayed.
Data latch	A storage element capable of storing a single bit.
Data line	A signal wire used to convey a single binary digit of a data value.
Data rate	The rate at which data may be transferred between entities.
Data register	An internal storage element, register, within a CPU or other device which is responsible for holding a data value.
Data transfer time	The time taken to transfer a quantity of data from one place to another.
Data-flow (CPU)	The relationship of data values as they flow through an instruction sequence.
Data-flow analysis	Analytical techniques used to understand and potentially optimise data-flow.
DCHP	A mechanism for allocating IP addresses to devices on a network.
DDR Double-Data Rate	A method of data transfer between DRAM and CPU which uses both a zero and one transition of a clock to synchronise data transfers (as compared to a single transition in single data rate).
Decision point	A point in a program where a condition is evaluated and a choice of program flow is made, via a conditional branch.
Dedicated bus	A bus designed for one specific purpose.
Defragmentation	The process by which a file is made contiguous.
Denial of service attack	A malicious action designed to prevent a server from being accessed normally.
Destination node	In a network, a device receiving data.
Destination register	In programming, a destination register is the register designated to be receiving the result of an instruction.
Deterministic	Predictable, consistent behaviour able to be calculated in advance.
Device mapping	The process of interfacing devices such that they appear to occupy memory addresses within the address space of the machine.
Differential signalling	A signal transfer mechanism in which dual signal wires send opposing voltage states, which are then combined at the destination to create a noise free digital value.
Digital signal processor	A processor designed specifically for the complex processing of digital signals (of the type generated by complex sensors for instance).

DIMM	A memory module which may be plugged into a motherboard to provide memory capacity.
DIMM socket	A socket designed to accommodate a DIMM module.
Direct Memory Access	A process whereby a device may take control of the bus as bus master, in preference to the CPU, and perform data transfers.
Disk	A storage device using circular disk surfaces for magnetic data storage.
Disk formatting	The process by which a disk is configured to a particular file system and prepared for use.
Disk mirroring	A system in which multiple disks contain the same information, to provide redundancy in the event of single unit failures.
Disk striping	A system where multiple disks are used to provide parallel portions of data in order to boost overall disk performance.
Disk swapping	The activity whereby virtual memory pages are moved between memory and disk.
Disk track	A circular zone on a disk, consisting of multiple sectors arranged around its circular region.
Distributed system	A system of computers being remotely located in different places and linked via a network infrastructure in order to perform a collective function.
DMA	See Direct Memory Access
Embedded system	A computer system designed primarily to perform tasks within a closed system, where the user does not interact with its lower level functions.
Domain Name	A textual label used to identify a web-based IP address.
Domain Name Server	A machine responsible for translating domain names into IP addresses.
Dots-per-inch	The density of dots used in printing systems to create the impression of realistic images.
DRAM	Dynamic Random Access Memory.
Driver	A program responsible for ensuring a device is able to communicate with the operating system and applications in a consistent fashion.
Dumb terminal	A minimal computer terminal, keyboard and monitor, that is used to access a remote system with more capabilities.
Dynamic branch prediction	A process whereby branches are predicted based upon information gathered during program execution.
Dynamic linking	A mechanism that allows program modules to be selectively added to memory as and when they are needed by a main program.

Dynamic memory allocation	Allocation of memory by the operating system in a real-time fashion, that is during program execution.
Dynamic Ram	See DRAM
Dynamically adaptive routing	A network routing mechanism that relies upon information gathered during the operation of the network in order to inform the best routing decisions.
EAROM	Electrically Alterable ROM
ECC memory	A memory device which includes Error Correction Code capabilities.
EEROM	Electrically Erasable ROM
Effective memory bandwidth	The bandwidth of memory achieved after taking into account relevant factors, such as cache for example.
Encryption	The process by which data is transferred into a form that is not intelligible by a third party without considerable effort.
Encryption algorithm	An algorithm designed to encrypt data in a particular way.
Enterprise system	A system designed to serve a wide ranging commercial purpose within a medium to large organisation.
EPROM	Erasable Programmable ROM
Erasable Memory	See EEPROM, EAROM, EEROM.
Error correction	A mechanism that uses coding techniques to permit bit errors to be corrected via redundancy of information.
Error correction code (HDD)	A code provided to facilitate error correction.
Error detection	The process by which a bit error can be identified via analysis of redundant reference data (see ECC).
Ethernet	A widely used network standard.
Executable file	A file capable of being executed as a series of instructions on a processor.
Execute	To execute an instruction or program, to cause the processor to perform its constituent instructions.
Fast-Page-Mode	A memory access protocol whereby successive addresses within the same row may be accessed in any order but with a faster than normal rate of access.
Fault-tolerance	The ability of a system to continue to operate satisfactorily when a fault or failure occurs in one or more components.
Feature size	The critical dimension of structures on silicon chips used to design transistors.
Fetch-execute cycle	The process of fetching an instruction from memory, decoding it, and then executing it, repetitively on a processor.
FIFO	First-In First-Out, a queue or buffer policy.

File access time	The time taken to read the content of a given file from a particular storage device.
File allocation table	A table of information indicating which tracks and sectors each file occupy in a storage device.
File corruption	Errors within the content of part or all of a file.
File header	An initial portion of file content containing specific information required to make sense of the remainder of the file.
File system	An organisation and set of procedures relating to files stored on a storage unit or units.
Firewire	A proprietary peripheral connector.
Firmware	Program code stored and available at start up , usually in a semi permanent memory chip. Analogous to BIOS.
Flash Drive	A device using flash memory to provide data storage, as if equivalent to a disk unit.
Flash Memory	A kind of memory that is capable of being written to and then retaining data for very long periods (decades) so essentially non-volatile.
Flip-flop	A circuit that is capable of storing a single binary digit.
Floating point	A number including decimal fractions.
FLOP	A floating Pint operation.
Fragmentation	The state in which files or memory blocks are scattered in such a way as to be non-contiguous.
Freeing memory	The action of releasing a reserved portion of memory back to the operating system for reuse.
Frequency	The number of clock cycles occurring per second, or the number of events of a particular type occurring in a given time period.
Front-side-bus	See main bus.
General-purpose	Something designed to facilitate many different activities without being optimal in any of them.
Golf-ball print head	An antiquated method of printing fonts on paper, consisting of a ball upon which all font characters are available for impact against a ribbon (like a typewriter).
Graphical operating system	An operating system in which a graphical user interface provides most of the facilities and services.
Hackable	A system that is capable of being subject to security breaches.
Hard disk drive	A disk unit for data storage. See disk.
Hardware	Any electronic or physical component of the computer system.
HDD	See hard disk drive.

Heatsink	A metal finned component designed to disperse heat rapidly away from a source , such as a cpu, to maintain satisfactory operating temperatures.
Hidden file	A file that has been designated as non-visible by the file system, but still exists.
High level language	A textual form of computer programming that uses syntax and formulations that are abstracted away from machine level program code.
Hit-rate	The rate at which a particular desired event is achieved, in cache this relates to how often a memory access succeeds in being serviced by a cache memory.
Host bus	See main bus.
Hosted Virtualisation	A method of providing virtual machine(s) via a software layer that sits above (is hosted by) the operating system, potentially a third-party virtualisation utility.
HTTP Request	A message format defined by the HyperText Transfer Protocol which can be sent to a remote HTTP server.
Huffman compression	A method of compressing data by encoding frequently occurring symbols with shorter codes, in exchange for encoding sparsely occurring symbols with longer ones.
Hyperthreading	See SMT.
I2C bus	A bus interface standard using a two-wire protocol to link multiple devices. Often used in embedded systems and industrial systems.
IDE (disk)	Integrated Drive Electronics, an interface standard for hard disks and other storage devices.
ILP	Instruction Level Parallelism.
In-order issue	Commencement of instructions in the same order as they are encountered in a program sequence.
Independent failure rate	The probability of one component failing in a set of two or more.
Instruction	A numerical value designating a computational operation to be performed by a CPU.
Instruction bandwidth	The amount of memory bandwidth available for instruction fetches.
Instruction cache	A cache memory reserved specifically for caching of instruction codes.
Instruction issue	The process of commencing the processing of an instruction.
Instruction register	A storage element, register, used to hold an instruction within a CPU or other device, during its processing.

Instruction set	The list of defined possible instructions a particular processor is capable of performing.
Instruction-Level parallelism	See ILP.
Integer	A number having no decimal part, a whole number.
Integrated circuit	A complex electronic circuit completely implemented on a single piece of silicon.
Inter-process communication	Exchange of information between two processes, normally via a well defined route defined and managed by an operating system.
Interleaved seek	The result of multiple read operations continuing over a period of time from a storage unit, causing the head armature to move repeatedly back and forth between multiple zones of the hard disk, usually causing inefficient data access.
Internal readout (memory)	The action of reading data from an internal storage structure of a memory device, into a temporary buffer for more convenient data access.
Internet-of-things	The idea of many items in everyday life being connected to the internet to provide collective capabilities.
Interpreter	A program which translates high level language statements into sequences of low level function executions.
Interrupt	A signal generated by a device which notifies a CPU that the device requires a servicing action to be performed.
Inverter	A logic gate which converts a binary input into an opposite binary output.
IO device	A device responsible for performing some input or output function in a computer system.
IO servicing	The process of a CPU performing essential actions in order to support an IO device in a required task.
IO Wait Queue (task)	A queue in which threads are held whilst they are waiting for an IO device to respond.
IO-Polling	A method of establishing the status of an IO device by repeatedly reading its status.
IP Address	A numerical identifier used to identify a computer uniquely within a region of a network.
Issue width	The number of instructions a CPU can commence at the same time, in a single clock cycle.
Jitter (task)	The variability of execution time and periodicity of a task due to time-slicing and uneven task scheduling distribution of CPU time.
Job-scheduling	A process whereby a series of jobs are queued in a particular order for processing.
Jump	See branch.

Kernel	The core program modules of an operating system.
Kernel level thread	A thread created and managed with the involvement of the operating system.
Kernel mode privilege	Capabilities only available for processes and tasks operating as part of the operating system kernel, or under its direction.
Key-logger	A program that secretly monitors data values in the keyboard interface in order to record what the user is typing.
Kilowatt	One thousand watts.
Lag (network)	A delay in the response of a network that causes noticeable effects on the quality of service.
LAN	See Local Area Network.
Least frequently used replacement	A replacement policy, typically used in cache, which replaces the least often used cache content with new data.
Least recently used replacement	A replacement policy, typically used in cache, which replaces the least recently used cache content with new data.
Legacy bus standard	A bus standard that is still supported for convenient use of old devices but which is no longer regarded as a standard part of a computer specification due to being superseded by newer options..
Lightweight compression algorithm	A compression algorithm that uses minimal CPU overhead.
Lightweight Process	Another name for a thread.
Linear program sequence	A sequence of instructions that follow one after another in an uninterrupted sequence.
Linear sequence	See Linear program sequence
Linking (compiler)	The process by which a compiler combines predefined code modules (libraries) with the custom code developed by the programmer to make a complete program.
Load (memory)	The action of performing a read operation.
Local area network	A network depending on locally installed infrastructure, and not using any network infrastructure outside of a designated area , such as a building.
Local bus	See main bus.
Logic chip	An IC containing a number of logic gates.
Logic gate	A digital circuit capable of performing one primitive logic function by operating on binary inputs and generating a binary output.
Logical AND	A logic function in which an output is binary 'one' only when both inputs are also one.

Logical memory	An address that exists in virtual memory and not in the physical address space.
Logical NOT	See inverter.
Logical operation	Any binary operation that translates binary inputs into a binary output.
Logical OR	A logic gate that generates a binary one if either input is binary one.
Logical XOR	A logic gate that generates a binary one only when both inputs are opposites, e.g. one/zero or zero/one
Loosely-coupled system	A system where a number of processors or computers cooperate on a collective task, with minimal exchange of information, low dependency on each others results, and using standard connection options such as busses and networks.
Lossless compression	A data compression method where the reconstructed data is identical to the original after subsequent decompression.
Lossy compression	A data compression method where the reconstructed data is an approximation of the original data after subsequent decompression.
LZSS compression	Lempel–Ziv–Storer–Szymanski algorithm used to compress data by creating an expanding set of recurrent symbol sequences that can be encoded by references to previous instances.
MAC address	A supposedly unique identification assigned to any networkable device.
Machine code	The numerical value sequences understood by a processor to be a sequence of instructions.
Machine learning	See Artificial Intelligence.
Magnetic core store	A now obsolete memory storage medium in which each bit was stored in a minuscule magnetic ring, slotted onto an Row-Column grid of wires.
Magnetic disk	A data storage system using a magnetic surface deposited on metal or rigid rotating disks.
Magnetic recording head	A magnetic sensor and emitter capable of magnetising surfaces and changing their magnetic fields.
Magnetic tape	A plastic tape coated with a magnetisable surface material capable of being magnetised in order to store data.
Main bus	The primary bus of the computer system, linking together processor, main memory and IO devices.
Mainboard	See motherboard.
Mainframe	A powerful computer system designed to manage the workloads of many users simultaneously.
Memory bandwidth	The data transfer capacity of a memory device.
Memory bus	See Main Bus, primarily any bus used to connect memory to the system.
Memory bus width	The number of signal lines, and therefore bits, provided for transferring a data value from memory to CPU and vice versa. Typically 8, 16, 32, or 64 bits.

Memory capacity	The amount of data values a memory device or system is capable of storing.
Memory coherency	The state of memory as compared to another representation such as cache, such that the memory content is either the same as cache (coherent) or different (non-coherent).
Memory controller	A chip or circuit responsible for controlling all of the complex physical memory control functions in a system. See memory bandwidth.
Memory data rate	The amount of memory utilised by a given resource, such as a program, utility, driver, or other function.
Memory footprint	The result of dividing memory into individually allocated blocks, and these blocks ending up scattered across memory in diverse locations non-contiguously.
Memory fragmentation	A memory component, such as a DIMM or SIMM.
Memory module	A feature of an operating system that ensures that threads and processes not related to each other are unable to access each other's memory space.
Memory protection	See Flash Drive.
Memory stick	A recently developed electrical component with novel capabilities allowing it to retain information at an atomic level.
Memristor	An obsolete data memory device capable of storing bits by circulating acoustic vibrations through a tube filled with mercury.
Mercury delay line	A queue holding a series of data packets designated as messages, used to permit shared communication between independent entities in one to one, multicast or broadcast mode.
Message queue	A micro-electro-mechanical system (MEMS) providing large 2 dimensional arrays of microscopic mirrors in order to create dynamically modulated light sources. Primarily used in video projectors.
Micro-mirror Array	One millionth of a metre.
Micron	A cache system in which there are multiple layers of cache, existing in different components of the system.
Multilevel cache	One billionth of a metre
Nanometre	Millions of instructions per second.
Mips	

Mis-speculation	The result of attempting to predict a branch and then execute further instructions, but then it transpiring that the prediction was incorrect.
Mobile-processor	A processor designed and optimised for mobile devices such as smartphones.
Moore's Law	The theory that silicon transistor counts on the latest chip fabrication technologies doubles roughly every two years.
Motherboard	The main circuit board in a computer system, housing the processor and major components.
MTBF	Mean time between failure, the timescale on average before a component fails.
Multi-port bridge (Network)	A network bridge with the ability to support multiple segments.
Multi-threaded program	A program that consists of several independently operating parts, which may or may nor exchange information and cooperate in doing so.
Multi-touch interface	A touch-sensitive interface that can detect and interpret multiple simultaneous touch points, for example several fingertips.
Multi-user system	A system that supports simultaneous use by multiple users.
Multicast	A transmission of information to multiple specified destinations.
Multicore	A processor chip containing multiple CPUs.
Multiplexed addressing	A mechanism that permits an address to be split into rows and columns, and reduced the number of address lines required by doing so.
Named Pipe	A communication pipe with a designated identity that persists before and after program execution.
Native Virtualisation	A method of supporting virtual systems that is linked directly to the operating system and provides access to the low level system.
Network drive	An entity that appears to be a disk storage unit but actually connects to a resource accessed via a network connection.
Network hub	A device that permits multiple network segments to be connected to a single host network.
Network layer (OSI)	The software layer of the OSI reference model responsible for moving and routing data packets.
Network node	An entity visible on a network, typically a machine or accessible resource.
Network routing	The process of deciding which network segment(s) a packet should be transmitted across or to.

Network scalability	The ability of a network to grow to a larger level without loss of quality of service of its supported purposes.
Network segment	A subsection of a network separated by a bridge, hub, or network switch.
Network subnet	See network segment.
Network switch	A device capable of routing network packets to one of multiple possible network segments to which it is connected.
Non-linear sequence	A program sequence where the linear instruction execution sequence breaks, and moves to another location in memory.
Non-programmable memory	A memory device that has content that cannot be altered.
Non-volatile	A property of memory that means that it is able to retain data content when power is switched off.
One-Time Programmable	A memory device that can be programmed one time and then becomes unalterable.
Opcode	A numerical value representing a particular processor instruction code.
Operating system	The software and hardware components that comprise the system that manages the smooth operation of the computer system and its resources.
Out of order completion	Completion of instructions in a processor, with results being generated in a different order than the order that the related instructions were issued.
Overlapped instruction execution	Execution of one instruction whilst activities related to another instruction, such as instruction fetch, are being performed at the same time.
Packet collision	An event where two packets are attempting to be transmitted on the same network segment, and as a result neither can be transmitted correctly.
Page (virtual memory)	A memory space larger than physical memory, residing mainly on the data storage system, in which portions of that memory content can be copied into physical memory for fast access as and when needed.
Paging	The process of moving blocks of memory back and forth between physical and virtual memory.
Paging supervisor	The program responsible for paging.
Paper tape	An obsolete storage system in which paper tape has holes punched into it to designate binary values.

Parallelism	Multiple processors working on different parts of the same work at the same time.
Partition (disk)	A subdivision of a physical disk into separately identifiable parts.
PCB (OS)	Process Control Block, a set of information relating to the state and configuration of a task existing in multi-tasking operating system.
PCI bridge	A bus bridge specifically for interfacing to PCI busses.
PCI bus	A Bus conforming to the PCI bus standard.
Peltier device	A device that can convert heat into power or use power to generate cooling.
Peripheral bus standard	A definition describing a bus or connection standard used for connecting peripheral devices.
Peripherals	Devices external to the main computer system that are connected to make the system more usable or to extend its core capabilities.
Pervasive computing	See Internet Of Things.
Photovoltaic cell	An electronic device that can convert light into power.
Physical memory	The actual chips present in a system that provide real memory capacity.
PID (OS)	Process Identifier. A numerical value assigned to a task to allow it to be uniquely identified during its operation.
Piezo cell	An electronic circuit that can convert mechanical stress and vibration into power.
Pipe	A communication mechanism between two software elements.
Pixel	A single image element, of which many combined together side by side would create the impression of an image.
Platter	A circular disk surface existing within a disk unit.
Plug-and-play	The ability for devices to be plugged in and removed from a computer system whilst it is operating.
Point to point	A connection between exactly two devices.
Polling	The process of checking the status of a device.
Power	Electrical power, as consumed in any electronic system.
Power density	The amount of power consumed by a device of given size and volume.
Power harvesting	Methods of obtaining power from the environment, via for example, light, heat, vibration.
Power wall	A performance barrier related to power consumption and related heat.
Power-on reset	A restart of a computer system by removing and reconnecting power.

Power-on self-test	A startup procedure by which a computer system tests key resources immediately after starting up to ensure it is capable of normal operation.
Printed circuit board	A circuit board mass produced with wires embedded into its surface.
Prioritisation	A method of designating entities with a system as having differing importance, typically applied to tasks.
Priority level	A value designating the relative importance of an entity such as a task.
Privilege levels (OS)	Settings determining what level of access a task or entity has to restricted resources within a system or operating system.
Process	A program running on a computer system under operating system control, and potentially having multiple sub parts known as threads.
Process control block	See PCB.
Process identifier	See PID.
Processor state	The essential register values and other related information that represents the processor state at any given moment.
Producer-consumer	A relationship between two entities, typically software, where one produces information and the other consumes it.
PROM	Programmable Read Only Memory
Punched card	A paper card which may be punched by holes to represent and store binary or symbolic data.
Rack-mount	Mechanical rack systems that allow modules of particular standard sizes and dimensions to slot into systems to create easily built arrays of processors and storage devices.
RAID	A disk storage system that supports a variety of redundancy options to facilitate fault tolerance.
Random Replacement	A policy, typically used in cache, to replace an existing content with new content, on a random basis.
Ransomware	A malicious program that encodes or blocks access to a user's data, and then will only release it after a payment.
RAS	Row Address select signal.
Raw bandwidth	The bandwidth of a bus or network before taking into account any packet format or bus protocol overheads.
Ready Queue (task)	The queue of tasks that are currently ready to execute.
Read-only file	A file that may be read but not overwritten.

Read-Only memory	A memory device that can be read but not altered.
Read-transaction	The process of performing a data read from memory.
Read/write head	A device capable of generating or detecting magnetic fields for the purpose of achieving storage or retrieval of data from a magnetic surface.
Real-time computing	Computer systems and programming where the timing of events is critical to the interaction with real-world requirements.
Real-time operating system	An operating system designed specifically to fulfil the needs of real-time computing.
Refresh cycle	An internal operation with DRAM memory chips that requires it to perform a special data amplification operation on stored data.
Register	A set of flip-flops making up an n-bit storage element.
Releasing memory	The process of de-allocating memory previously allocated by dynamic allocation.
Reliable service provision	In networking terms this relates to being able to provide trusted and reliable data exchange that higher level software components can rely upon without having to implement their own error avoidance mechanisms.
Repetitive Interrupt Timer	A counter that is able to repeatedly measure elapsed time with precision, and generate interrupts when those time intervals have elapsed.
Resilience	The ability to resist the effects of faults or failures without the system ceasing to operate at a minimum acceptable level.
Resilient file system	A file system which is designed to reduce the possibility of data loss under undesirable circumstances such as power failure.
Resumed task	A task that was suspended, but has been resumed for a further period of execution.
RF power harvesting	Conversion of ambient radio signals into electrical power.
RIT	See repetitive Interrupt Timer.
Root control	The ability to control which part of the file system appears to be the root of the file system, as perceived by a given task or user.
Root directory	The top level directory of the file system.
Rotational latency	The time required for a disk to rotate once.
Row-Address-Select	See RAS.
RTOS	See Real-Time Operating System.
Safety-critical	A requirement that is essential to correct and safe operation of a system.
SATA	A widely used disk control interface.

Scheduler	The operating system component responsible for managing the time-slicing of tasks.
Scheduling algorithm	An algorithm used by a scheduler to achieve fair and efficient task scheduling.
SCL	Serial Clock - A control line used by I2C BUS.
SCSI	A widely used disk interface standard.
SDA	Serial Data Line - a data line used in the I2C bus.
Secondary bus	An auxiliary bus providing additional connectivity in a computer system.
Sector	A block of bytes, a small section of a track, on a disk platter.
Sector size	The number of bytes stored in a sector.
Sector wastage	The amount of unused bytes where a file does not divide exactly into an integer number of sectors.
Security threat	Any thing that is capable of presenting a threat to the correct operation of a computer system or to the privacy of its data.
Seek time	The time taken for a disk head to be moved to the desired track during a data read operation.
Serial clock line (I2C)	See SCL.
Serial Data line (I2C)	See SDA.
Server latency	The time delay between a server receiving a request for data and that data being sent back to the requester.
Service daemon	See Deamon.
Session layer	An OSI layer responsible for maintaining the continuity of a service for an application whilst it is needed.
Side-channel attack	A malicious attempt to interfere with a computer system by exploiting low level knowledge of its specific hardware behaviour.
Simultaneous MultiThreading	See SMT.
Smartphone	A mobile phone containing a processor and operating system and able to run applications.
SMT	The operating of multiple threads concurrently without needing to switch between them in time-slices.
Socket Interface	A software agent operating on client and server machines to facilitate a unique data flow between tasks existing on those two machines.
Software	Any program capable of running on a machine.
Solid state drive	A flash memory module with large capacity, and control circuits which make it appear like a normal storage drive.
Source code	The textual representation of a program in high level language form.

Spanned volume	A disk space that appears as a single entity but actually comprises several partitions on multiple drives.
Speedup	The ratio of the original versus new performance element being measured.
Spooling (tape)	The process of winding a tape forward or backward until the desired point is reached for data read or write.
Spooling (print)	Sending print jobs to a print server queue.
Stalled thread	A thread which has to stop issuing instructions due to a delay in completing preceding operations and a related dependency.
Static linking	The process of combining predefined program modules with a main program such that it becomes one monolithic block of program code.
Static memory allocation	Allocation of memory at program startup and not during its runtime execution.
Static routing algorithm	A network routing algorithm that uses fixed rules.
Stepper motor	A special motor capable of making very small step motions.
Storage density	The amount of data stored in a given area or volume of space.
Storage element	A device capable of storing one or more bits.
Subdomain	a subdivision of an internet domain.
Supercomputer	A large scale and highly powerful computer system designed to perform demanding computational tasks.
Superscalar	Pertaining to multiple instructions being issued and executed simultaneously.
Suspended task	A task that is temporarily placed in standby whilst something else is executed (usually another task).
Sustainable data rate	The data transfer rate that can be maintained for extended periods, rather than for short bursts.
Swap disk	See disk swapping. The disk where a swap file is stored during disk swapping.
Swap file	A file on a disk that receives virtual memory pages.
Symbolic link	In a file system this is the creation of the visibility of a file in a place where it does not physically exist, but linking it to some other location where the real file is stored.
Synchronous circuit	A circuit that uses a clock to control its behaviour.
System bus	See Main bus.
System file	A file that belongs to the operating system.
Tablet	A computer system that consist of a portable screen and only has a touchscreen for data entry.
Tablet stylus	A small pen like device that allows touch input to a tablet screen.

Task	A process or thread.
Task list	The list of tasks existing on a system, maintained by the operating system.
Task priority	The level of importance assigned to a particular task relative to others.
Task queue	A queue where tasks are held by the operating system for successive time slice allocations.
Task scheduler	See scheduler.
Task-scheduling	The process of allocating time slices to tasks.
Task-switching	The process of switching between two threads at the start of a new time slice.
Task-switching overhead	The cost incurred by task switching.
Thread	A sub-part of a process which is able to operate independently but not without its parent process being present.
Tightly-coupled system	A system in which processors or computers cooperate to perform a piece of work and exchange data frequently in ways that demand optimised and specially engineered connectivity.
Time-slice	A period of time allocated for a task to execute before being suspended.
Touchscreen	A screen that is also sensitive to user touch.
Transistor	An electronic component manufactured on a silicon chip that acts as a switching element.
Transparent compression	See automated file compression.
Transport layer (OSI)	A layer of the OSI model that is responsible for representing the host to host connectivity of services.
Thrashing	In Virtual memory systems the problem of pages being moved back and forth between physical memory and disk storage repeatedly in an inefficient manner.
Truth table	A table of values representing the possible inputs to a logic circuit and the resulting outputs.
Turnaround cycle	A clock cycle required by a bus or network connection that must elapse after a data transaction before a new transfer can begin.
Ubiquitous system	See Internet Of Things.
Universal Serial Bus	see USB
USB	A widely used Peripheral bus interface standard.
User level thread	A thread operating within a process that is responsible for its own internal task switching policy.
User mode privilege	A set of restrictions or rights allocated to a user at a given level in an operating system.
User process	A process running by choice of the user , for example by starting and application.
Utility	A useful program provided to assist with a particular purpose.

Virtual file	A file existing in a remote system but appearing to be local to the user.
Virtual memory	A memory space that resides primarily on the disk storage system, but some of which can be resident in the physical memory of the system, thus giving the impression of a much bigger memory space.
Virtualisation	The concept of making multiple independent machine environments appear to exist simultaneously on the same physical machine.
Virus	A malicious software agent intending to cause damage or inconvenience to the user by interfering with normal system operation.
Virus checker	A software utility that is tasked with detecting and preventing viruses from becoming resident on a system.
Volume (disk)	A designated space on a disk unit, or units, appearing to have a single defined identity in the file system.
WAN	See wide area network.
Wear-levelling	A method used in SSD's to ensure that write operations do not repeatedly used the same data block excessively, as this can cause memory fatigue.
Web-Search-Engine	An server designed to handle requests for information from a user, and which returns results in the form of links to web pages available anywhere on the internet.
Wide area network	A network covering a large area, often more than one organisation, or multiple sites belonging to the same company or organisation.
WiFi	Wireless network, a network created by radio frequency signal transmissions.
Wireless network	See WiFi
Write-transaction	The action of preforming a write operation to memory or storage.
XML	Special file format supporting structured and extensible data records.
Yield control (task)	When a task decides it does not need any further CPU time in its current time-slice and tells the scheduler to suspend it in favour of other threads.
Yield control (IC)	The proportion of integrated circuits in a manufacturing run, which remain, once any faulty chips are excluded.
Zoned bit recording	A mechanism that ensures that outer tracks of a disk (which are physically longer than inner ones) have more sectors per track than the middle or inner tracks. Multiple zones may be formed, for example ten regions each with different sector counts per track.

Alphabetical Index

- 3D image capture, 152
6T bit-cell, 69, 74
8-bit register, 14
- abstraction, 173
access control (memory), 204
access control (OS), 172
access time, 76
access time (storage), 129
address lines, 109
address range, 109
administrator privileges, 212
AES encryption, 281
AFFS file system, 228
Amdahl's Law, 63, 196
AND gate, 12
anti-virus software, 277
apollo guidance computer, 69
application layer (network), 247
applications (OS), 170
arbitration, 118
arbitration (bus), 111
architectural exploits (security), 279
arithmetic logic unit, 39
armature, 131
artificial intelligence, 58, 61
ASCII file format, 235
ATM network, 156
audio compression, 155
audio DSP, 155
audio output, 155
auxiliary bus, 104
- back off and retry, 119, 157
backing store, 128
bandwidth (network), 160
bare-metal system, 166
benchmark, 32
- benchmarking (storage), 136
best case efficiency (network), 160
binary file format, 236
BIOS, 94, 167
bit, 6
bit errors (storage), 144
bit-density (storage), 133
bit-stuffing (usb), 115
block length (bus transfer), 112
Bluetooth, 158
book reader task switching analogy, 185
boot sector, 167, 222
boot sector (file system), 222
boot-code, 168
boot-sector (security), 277
boot-sector virus, 277
bootable disk, 222
bootstrap, 167
branch history table, 36
branch mis-prediction penalty, 37
branch prediction, 36
bridge (bus), 113
bridge (network), 251
buffer, 13
buffer (bus transfer), 113
buffer (pipe), 209
burst mode access, 78
bus, 8, 103
bus bandwidth, 111
bus bridge, 104
bus efficiency, 112
bus master, 8, 104, 118
bus master resolution protocol, 118
bus multiplexing (PCI), 110
bus ownership, 111
bus protocol overhead, 111
bus slave, 104

bus standards, 107
bus transaction, 104
byte, 6

cache, 9
cache (disk), 138
cache block, 88
cache coherency, 91, 94
cache hit, 83
cache line, 88
cache memory, 82
cache miss, 83
cache replacement policy, 89
cache sets, 88
cache tag, 88
camera, 152
CAN bus, 120
capacitive charge, 75
carrier sensing (CSMA/CD), 157
CD-quality audio, 155
cellular array, 60
central processing unit, 20
CGI, 259, 265
chip socket, 102
circuit speed, 14
client-server architecture, 262
clock, 6, 13
clock frequency, 33
clocks per instruction, 33
cloud computing, 59, 265
cloud data storage, 145
cloud storage, 265
collision detection (CSMA/CD), 157
column address select, 109
command line (OS), 171
command shell, 241
compartmentalisation (OS), 182
compiler, 11, 45
compressed file systems, 232
compression ratio, 231
compute server, 267
computer memory, 14
concurrency (bus), 106

concurrency (network), 251
concurrency (OS), 174, 182
conditional decision point, 36
conditional execution, 47
control unit, 40
core-store, 69
cost versus performance (storage), 142
cost-benefit, 26
cost-performance tradeoff, 85
CPI, 41
CPU throttling, 30
CPU time (OS), 183
CRC, 283
CRC (storage), 144
critical path, 33
CSMA/CD protocol, 157
cursor, 150
cycle time, 76
cyclic redundancy check, 144

daisy-chain (bus), 117
data capacity, 6
data fetch, 48
data hazard, 34
data integrity, 145
data lines, 109
data packet (network), 250
data rate, 6
data register, 40
data storage, 128
data storage capacity, 132
data store, 48
data transfer, 104
data transfer bandwidth, 107
data transfer latency (network), 146
data transfer rate (storage), 129
data transfer time (disk), 132
data-flow analysis, 34
data-glove, 155
DDR memory, 97
dead store, 91
dedicated hard-line (network), 255
defragmentation (storage), 138

delayed write-back (cache), 90
denial of service attack, 280
dependency, 34
DES encryption, 281
deterministic (OS), 177
DHCP, 258
di stacking (memory chip), 96
differential signalling, 120
digital signal processing, 58
DIMM, 70
direct IO addressing, 95
direct mapped cache, 88
direct memory access, 10, 122
disk cache, 138
disk formatting, 222
disk mirroring, 145
disk storage, 128, 130
disk striping, 145
disk swapping, 206
distributed systems, 261
DLL, 204
DMA, 122
dma side-channel attack, 280
DNS server, 257
domain name server, 257
doorbell abstraction analogy, 173
dot-matrix printer, 154
dots per inch (printing), 154
drain (transistor), 12
DRAM, 73
drivers (OS), 169
dual on-chip cache, 85
dumb terminal, 58
dynamic allocation (OS), 202
dynamic branch prediction, 36
dynamic frequency scaling, 30
dynamic link library, 204
dynamic linking ((OS), 204
dynamic power, 22, 30
dynamic random access memory, 73
earliest deadline first, 193
EAROM, 72
ECC (memory), 92
ECC (storage), 144
EEROM, 72
effective data transfer rate, 136
electrically erasable memory, 73
embedded system, 55
embedded system (bus), 117
encrypted storage, 233
encryption (security), 281
encryption strength, 282
EPROM, 72
error correction (memory), 92
error detection, 283
error detection(memory), 92
Ethernet, 156
Ethernet Frame, 159
executable file, 227
EXT file system, 228
failure probability, 285
fairness (scheduling), 187
FAT file system, 228
fault-tolerance, 56, 285
fault-tolerance (storage), 143
feature size, 97
fetch-execute cycle, 25
FIFO (pipe), 210
FIFO queue (scheduling), 188
file allocation table, 221
file attributes, 227
file compression, 229
file corruption, 144
file encryption, 233
file extension, 237
file formats, 235
file header, 237
file management, 241
file structures, 236
file system, 220
file-system hierarchy, 224
filtering (network bridge), 251
firewall, 280
Firewire (bus), 117

firmware, 73
first in first out policy (cache), 90
flash memory, 72
flash memry (SSD), 139
floating point operation, 43
fragmentation (memory), 203
fragmentation (storage), 137
frequency, 6
FTP, 259
fully associative cache, 88

gaming, 58
gate (transistor), 12
General Purpose computer, 53
general purpose computer, 52
gesture recognition, 152
gigabyte, 6
golf-ball print head, 154
GPU, 60, 153
graceful degradation, 288
graphical operating system, 171
graphical processing unit, 60, 153
graphics tablet, 151

Hamming codes, 92
haptics, 155
hardware, 4
Harvard Architecture, 10
heat density, 97
heat-sink, 23
heatsinks, 97
hidden file, 225, 227
high level languages, 11
hit rate (branch prediction), 37
hit rate (cache), 83
Hollerith, 128
horizontal parity bit, 283
host bus, 105
hosted virtualisation, 270
hot-swap (bus), 116
hot-swapping, 122
HTTP, 259
HTTP server, 262

hub (network), 253
hub (usb), 116
Huffman compression, 230
hyperthreading, 196
hypervisor, 270

I2C bus, 118, 157
idle task, 187
ILP, 194, 195
ink-jet printer, 154
input devices, 150
instruction, 24
instruction fetch, 40
instruction level parallelism, 194
instruction register, 40
instruction scheduling, 35
instruction set, 44
instructions per clock, 195
integer ALU operations, 43
inter-process communication, 209
interference (parallelism), 64
interleaved seek operation, 139
internet of things, 61
interoperability (file standards), 238
interpreter, 11, 45
interrupts, 122
interrupts (task scheduler), 191
inverter, 12
IO device mapping (memory), 122
IO drivers (BIOS), 167
IO servicing, 122
IO wait queue (scheduling), 191
IP adddress, 256
IPC, 209
issue width, 35

Jacquard loom, 10
javascript, 264
jitter, 115
jitter (task scheduling), 189
jumps, 25

kernel (OS), 168

kernel level threads, 184
kernel mode (OS privileges), 214
kernel processes, 214
key-logger, 278
keyboard, 150
kilobyte, 6

LAN, 246
laser printer, 154
latency (network), 255
law of diminishing returns, 34
least frequently used policy (cache), 90
least recently used policy (cache), 90
legacy compatibility (bus), 115
level-1 cache, 85
level-2 cache, 85
lightweight compression algorithm, 233
lightweight process, 182
linear execution, 47
linear program sequence, 25
linear sequence, 35
linear sequence (memory), 78
linked file, 225
load/store operations, 25
local area network, 246
local virtualisation, 271
logic delay, 14
logic gate, 12, 68
logical address space, 205
logical memory, 205
logical operations, 24
loosely-coupled system, 62
lossless compression, 230
lossy compression, 230
low level programming, 11
LZSS, 230

MAC address, 256
machine code, 11
machine learning, 61
magnetic disk, 128
magnetic drum, 130
magnetic tape, 128

mainboard, 102
mainframe, 58
maximum speedup, 63
mean time between failure, 285
megabyte, 6
memory access protocol, 76
memory allocation (OS), 202
memory bandwidth, 77
memory bus width, 77
memory chip, 71
memory coherency, 91
memory corruption, 74
memory cost, 84
memory de-allocation (OS), 203
memory management (OS), 202
memory map, 93
memory mapped IO, 93
memory speed, 75
memory trends, 95
memory wear (SSD), 141
memristor, 97
mercury delay line, 68
message (IPC), 211
message queue, 210
micro-instructions, 41
micro-mirror array, 153
microarchitecture, 38
microphone, 151
microprocessor, 20
microsequence, 41
microwave oven, 55
mini-mainframe, 58
mips, 26
mis-speculation, 38
mobile operating systems, 177
mobile processor, 55
Moore's law, 22
motherboard, 102
motion tracking, 152
mouse, 150
mouse pointer, 150
mp3, 155

MTBF, 285
multi-level cache, 85
multi-touch, 150
multi-user access (network), 246
multicast, 104
multicore (OS), 195
multicore processors, 15, 35
multiple access (CSMA/CD), 157
multiple issue, 35
multiple recording heads (tape), 129
multiple sets (cache), 89
multiplexed addressing, 110
mutex, 212
mutual exclusion, 212

named pipe, 210
NAND gate, 13
native virtualisation, 270
network capacity, 253
network identification, 256
network interface, 156
network lag (gaming), 162
network node, 249
network segment, 251
network service, 259
network typologies, 249
networked computer, 58
networks, 156, 246
NEXOR gate, 13
noise resilience (bus), 120
non-cacheable, 95
non-deterministic (network), 158
non-linear transitions, 35
non-preemptive scheduling, 188
non-volatile memory, 71
NOR gate, 13
NOT, 12
NTFS file system, 228

operating system, 166
operating system hierarchy, 168
optical mouse, 150
OR gate, 12

OSI reference model, 247
output devices, 152

packet header (network), 159
packet jitter, 115
packet size (network), 159
packets (network), 248
page (virtual memory), 206
page fault, 207
page mode access, 80
page table, 207
paging (virtual memory), 206
paging penalty, 208
paging supervisor, 207
paper tape, 128
parallelism, 62
parity check, 283
partition, 223
payload (network), 159
payload jitter, 115
payload size (network), 159
PCB, 183, 187
PCI bus, 110
PDF, 154
peak mips, 32, 43
Peltier, 62
performance, 26
performance metric, 26, 27
performance tradeoff, 37
peripherals, 150
PERL, 265
pervasive computing, 61
photovoltaic, 62
physical layer (network), 247
physical memory, 71, 202
PID, 183
piezo, 62
pinned page, 209
pipe, 209
pipeline hazard, 34
pipelining, 34, 86
pixels, 152
platter, 131

plug-and-play, 115
point-to-point, 210
point-to-point connection, 105
polling, 122
polling (task scheduler), 191
polymorphic virus, 277
POP, 259
post (message queue), 211
postscript (printing), 154
power, 5, 22
power density, 23, 97
power harvesting, 62
power on sequence (OS), 166
power scaling, 30
power-on self test, 92
powered bus, 114
pre-emptive scheduling, 188
pressure-based feedback, 156
printer, 154
priority (scheduling), 187
priority level (task), 189
priority levels (scheduling), 192
private memory space (OS), 182
privileges (task), 212
process, 175, 182
Process control block, 183
Process ID, 183
process identifier, 183
process monitor utility (OS), 175
process state, 183
producer-consumer (OS, IPC), 210
producer-consumer model, 113
program counter, 39
program trace, 45
program-flow instructions, 25
prom, 71
protocol overheads (network), 159
punched card, 128

QoS, 251
quality of service, 161
quality of service (network), 251
Quality of services QoS, 157

queue (scheduling), 187
radio valve, 68
radio valves, 20
RAID, 144, 234
RAID0, 144
RAID1, 144
random replacement policy (cache), 90
raw bandwidth, 111
raw bandwidth (network), 160
read write control signals, 110
read-only file, 227
read-write head (disk), 130
ready queue (scheduling), 191
real-time computing, 56
real-time operating systems, 177
redundant storage array, 144
refresh cycle, 75
register file, 39
registers, 24
reliability (storage), 143
reliable service provision (network), 248
remote storage, 145
repetitive interrupt timer (scheduling),
 188
reserved socket ID, 260
resilient file-system, 234
resilient system, 285
resource allocation (OS), 172
RISC versus CISC, 15
RIT, 188
rogue process, 205
rom, 71
root control, 235
root directory, 224
rope memory, 69
rotational latency, 132
rotational speed, 132
round-robin scheduling, 186
routing (network), 254
routing algorithm, 255
row address select, 109
RSA encryption, 281

RTOS, 177, 193
safety-critical, 72
safety-critical system, 56, 158, 177, 288
safety-critical systems, 92, 189
samples per second (audio), 155
scalar execution, 34
scheduling, 176
sector, 131
sector (file system), 220
sector size, 131
sector wastage, 131
security threats, 276
seek time, 129
semaphore, 212
serial workload management, 185
server farms, 23
service latency (network), 160
session layer (network), 247
set associative cache, 89
SHA encoding, 281
shading algorithm, 153
shared memory, 63
shared memory (IPC), 212
shared memory (threads), 182
shell script, 171
side-channel attacks (security), 279
signal delay, 14
signal propagation (WiFi), 158
SIMM, 70
simple computer system, 7
Simplesched scheduler example, 186
simultaneous multi-threading, 196
sleep (task), 192
sleep mode, 30
smartphone, 54
SMT, 196
SMTP, 259
SOC, 55
socket ID, 260
sockets (network), 260
software, 4
solid state drive, 139
source (transistor), 12
source code, 11
Space Flight, 58
spam virus, 278
spanned volume, 223
spatial locality, 82, 87
speculative execution, 38
speed versus complexity, 14
speedup in parallel system, 64
split cache, 86
spooling (printer), 154
SRAM, 73
SSD, 139
SSH, 259
SSH application, 261
stack (OS), 202
stalled thread (hyperthreading), 197
static allocation (OS), 202
static branch prediction, 36
static linking (OS), 203
static power, 22, 30
static Random Access memory, 73
storage cell, 69
storage density, 74, 133, 142
storage density (RAM), 75
storage element, 13
stylus (tablet), 151
sub-directory, 224
supercomputer, 58
superscalar (OS), 194
superscalar execution, 35
superuser (OS), 212
supervisor (OS), 212
suspended (task), 192
switch (network), 253
symbolic link (file system), 225
synchronous circuits, 13
system bus, 8, 105
system on a chip, 55
tablet stylus, 151
tape spooling, 129
task resumption, 187

task scheduler, 184
task scheduling, 184
task suspension, 187
task switching, 174
task switching overhead, 186
task-list, 182, 186
TCP/IP, 250
TCP/IP stack, 250
technology nodes, 22
temporal locality, 82, 87
terabyte, 6
thermal density, 23
thrashing, 208
thread, 175
thread level parallelism, 195
thread management, 184
threads, 182
Thunderbolt (bus), 117
tightly-coupled system, 63
time-slicing, 58, 185
TLP, 195
toner (laser printer), 154
top500 Supercomputer index, 60
touchpad, 150
touchscreen, 151
track, 131
traffic light analogy (network), 160
traffic volume (network), 252
transfer efficiency (network), 159
transistor, 12
transistor operation, 12
transmission delay (network), 160
transparent compression, 232
transport layer (network), 248
triple redundancy, 288
truth tables, 12

ubiquitous systems, 61
ultrasonic stimulation, 156
unified cache, 86
Universal Serial Bus, 113
UPS power backup, 287
URL, 257

USB, 113
USB flash drive, 73
usb protocol, 114
user level threads, 184
user mode (OS privileges), 214
utilities (OS), 169

vertical parity bit, 283
video card, 153
video conferencing, 151
video display, 152
video projector, 153
video recording, 152
video-tracking, 155
virtual file, 226
virtual machine, 267, 269
virtual memory, 71, 205
virtualisation, 246, 268
virus, 276
virus checker, 277
virus signature, 277
voice command, 151
voice dictation, 151
volatile memory, 73
volume, 223
von Neumann Bottleneck, 80
von Neumann bottleneck, 9, 49, 85
von Neumann model, 8
VR headset, 155
vt100 terminal, 58

wait-cycle, 76
WAN, 246
wearable computers, 62
weather forecasting, 58
wide area network, 246
WiFi, 158
wired network, 156
wireless network, 156
wireless networks, 158
worst access time, 135
worst case efficiency (network), 160
write-back (register), 41

write-back policy (cache), 90
write-limit (SSD), 141
write-through policy (cache), 91

X11, 259
XML, 240
XML file format, 235
XNOR gate, 13

XOR gate, 12
Xwindows, 259

yield (memory chip fabrication), 96
yield (task scheduling), 188

zero-wait-state, 76
ZIP file, 230
zoned bit recording, 133