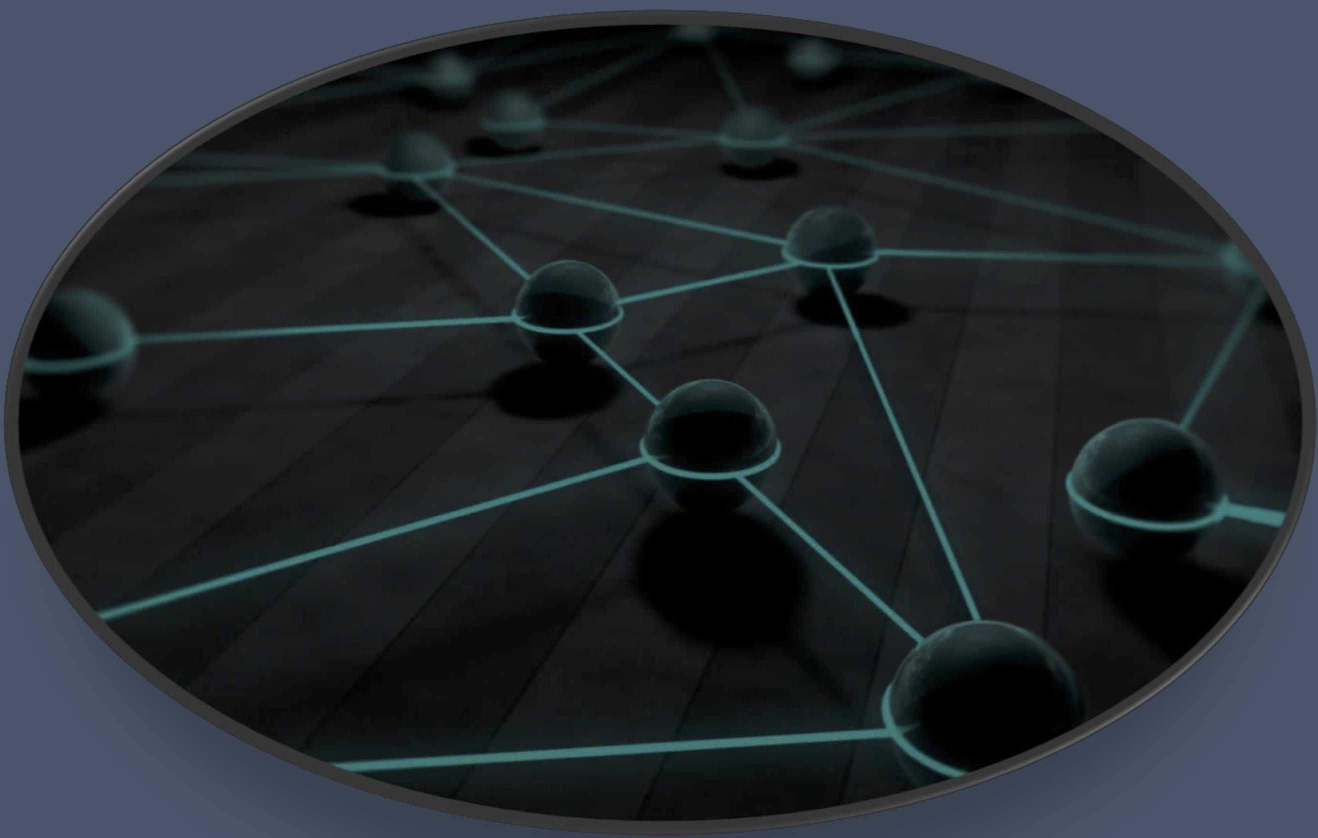
COMP20081

Systems Software

Revision Guide



Matthew Robinson

Contents

1. Introduction

1.1 History of performing computations

1950s

A user may want to perform computations that may have a complexity that is not feasible by a standard calculator.

A description of the computations to be performed could be expressed in a physical form using a perforated card.

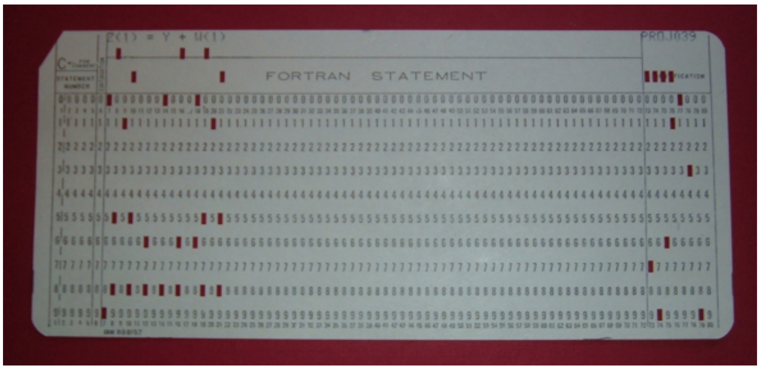


Figure 1.1: Perforated card.

There are multiple lines on the card, each with varying holes. Each line describes a particular instruction that is necessary for the computations to be completed.

The user would describe their computations to someone who is capable of producing the correct perforated card needed for their computations.
↓
The perforated card would be passed to the computer operator. A computer operator is a manager of computational resources that provides common services.
↓
The computer operator would have a job queue from all of the users. These would be processed in a "first in, first out" (FIFO) fashion.
↓
Jobs that are ready to be processed go in to batch processing. Each perforated card is passed through a computer system and the printed results are sent back to the user.

Figure 1.2: Process for performing computations in the 1950s.

>1960s

The process of performing computations after 1960 changed dramatically and required far less human input.

Automation was brought about by the "Operating System Paradigm" in which users could interface with a computer system directly through an operating system (OS).

User → Operating System (OS) → Results

Figure 1.3: Process for performing computations after 1960.

History details

Late 1950s	• Standard subroutines were produced that were loaded at start-up. These contained features similar to those found on an operating system. • Magnetic tapes were used for storage and were later replaced by disks. • Assemblers started to be used. These are programs that takes basic computer instructions and convert them in to machine code; this is a pattern of binary bits (0's and 1's) that the computer system's processor can use to perform its basic operations. • High-level languages, which consisted of more natural and human-readable language, started to be used. For example, FORTRAN is a general-purpose, compiled imperative programming language that is especially suited to numeric computation and scientific computing and was introduced in 1957.
1960s	Automated batch system. • This replaced the computer operator. • Several programs could be loaded in to memory and automatically processed in a "first in, first out" (FIFO) fashion.
1970s	Multiprogramming. • The computer could switch between jobs, which allows processing and input/output (I/O) interaction simultaneously.

1980s	Graphical user interfaces (GUIs). • The interaction between a computer system and a user through the medium of a mouse and keyboard.

1.2 Hardware

External hardware

A **peripheral** is any external hardware device that provides input/output (I/O) for the computer.

For example, a keyboard and mouse are input peripherals, while a monitor and printer are output peripherals. Some peripherals, such as external hard drives, provide both input and output for the computer.

A computer system generally has many internal hardware components and hardware peripherals.

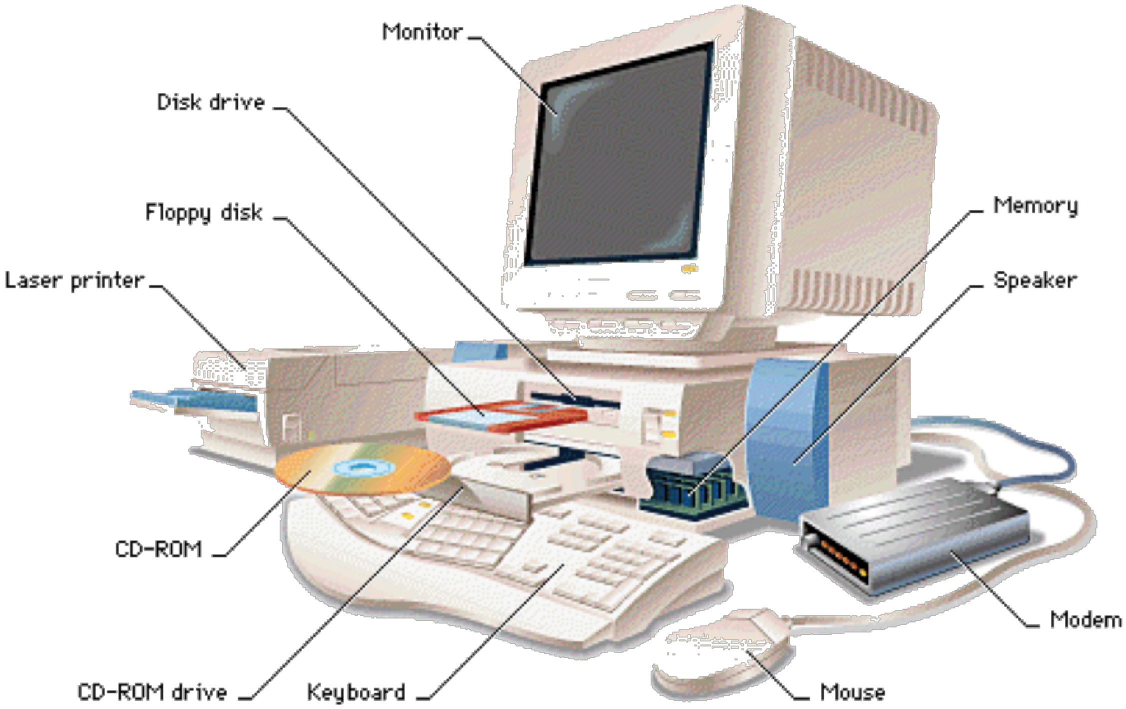


Figure 1.4: External view of computer system.

Internal hardware

A **processor** or **central processing unit (CPU)** is the hardware within a computer that carries out the instructions of a computer program by performing the basic arithmetical, logical, and input/output operations of the system.

A **motherboard** is the main printed circuit board (PCB) in a computer. The motherboard is a computer's central communications backbone connectivity point, through which all components and external peripherals connect.

Inside of a computer system, there are many components connected to the processor via the motherboard.

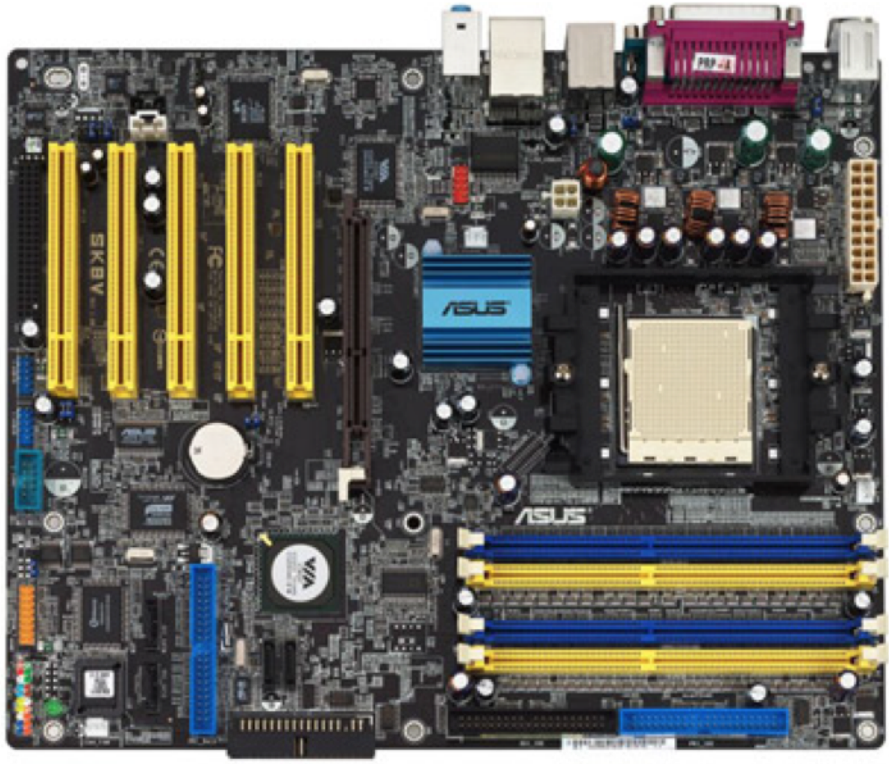


Figure 1.5: Typical computer motherboard.

The motherboard connects:

- all of the internal components via the data bus; and
- the peripherals.

1.3 What is an operating system (OS)?

Definition

An **operating system (OS)** is a collection of system programs that manage the hardware resources and peripherals connected to a computer system. It is also responsible for the graphical user interface or command line interface and all other software running on the computer system.

Purpose of an operating system (OS)

An operating system (OS) is designed to:

- eliminate the need to have hardware knowledge to operate a computer system;

- make the boundary between hardware and software transparent, allowing the user to not be concerned with the technical details; and
- provide a user-friendly environment to execute and develop programs.

These attributes are achieved by layering the computer system such that the user can interface with applications, rather than the operating system (OS) or the hardware directly.

User
↕
Applications
↕
Operating System (OS)
↕
Computer System Hardware

Figure 1.6: Computer system layers.

Structure of an operating system (OS)

The structure of an operating system (OS) can be said to resemble an onion.

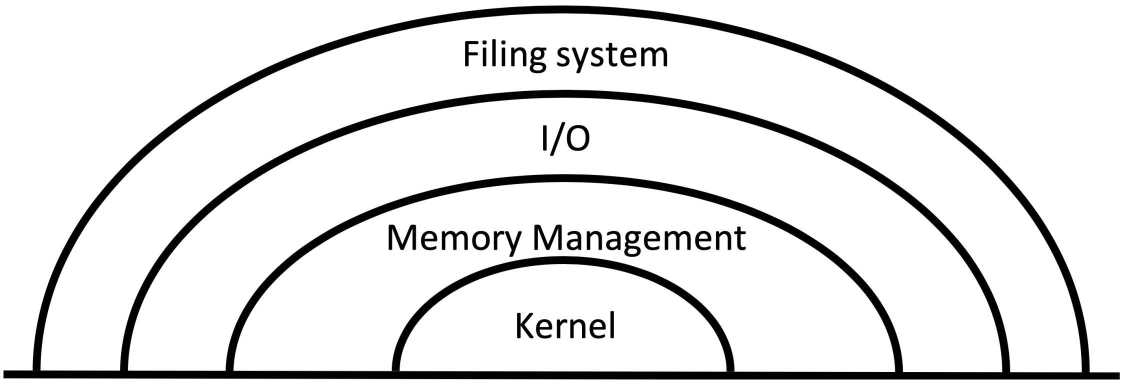


Figure 1.7: Structure of an operating system (OS).

An operating system (OS) has four main components.

The **kernel** hides the complexity of how a computer system works from users. It is responsible for:

- process management;
- CPU scheduling; and
- handling interrupts.

Memory management is responsible for allocating and deallocating memory to processes.

Input/output (I/O) includes any interaction between the internal computer system components and peripherals.

The **filing system** is comprised of file management subsystems.

Each layer in the operating system (OS) structure provides functions to the above layers. Each layer uses facilities provided by layers within and below that layer.

Practical features of current operating systems (OSs)

Concurrency	Allows overlapping input/output (I/O) operations with computations and several programs to be stored in memory at a single time.
Sharing of resources	Sharing hardware and peripherals, such as hard disks and printers.
Access to long term storage	Important for saving important files on mediums such as hard disk drives (HDDs) and solid-state drives (SSDs).
Non-determinacy	The ability to cope with unpredictable events without crashing.

1.4 Functions of an operating system

An operating system (OS) has two main complementary functions:

- resource managing; and
- machine extending.

Resource managing

It manages resources shared among users and user programs and maximises their utilisation of the CPU, RAM and other resources. This is done simultaneously in order to increase the availability.

This is similar to the role of computer operators in the 1950s.

Machine extending

It presents a virtual machine (or extended machine) to users that is much easier to access than the underlying physical machine.

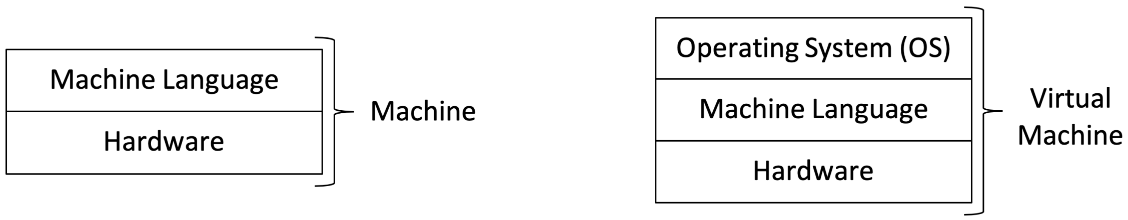


Figure 1.8: Machine extending.

The virtual machine presented to the user provides an abstraction of the computer system. This hides the complexity of the hardware from the user; this means that the user need only be concerned with the details of the hardware if they desire.

This is a way of translating the functions needed by a user from the hardware to a presentable and user-friendly medium. As a result, the operating system (OS) acts as an intermediary layer between the user and machine language.

The benefit of this abstraction can be demonstrated when comparing how computations may be processed with and without an operating system (OS).

Without Operating System (OS)		**With Operating System (OS)**
The instructions written in machine code or assembly language much interface directly with memory hardware. As such, the memory locations to load the two numbers from must be explicitly defined and the memory location to which the result is stored must also be defined. The example below shows a possible assembly code implementation of a computation that is capable of adding two numbers.		The instructions can be written in a high-level language, such as C++.

LDAA $80	(load number at memory location 80)	int a, b, c;
		a = 1;
		b = 2;
		c = a + b;

LDAB $81	(load number at memory location 81)	
ADDB	(add these two numbers)	
STAA &55	(store the sum to memory location 55)	

Figure 1.9: Adding two numbers.

This demonstrates that program development is much more user-friendly with an operating system (OS). This is because without an operating system (OS) the user must have knowledge of the system hardware; in this case, the necessary memory locations.

In addition, it is possible that the machine code or assembly language written may not work on another computer system as that computer system may have a different architecture or the memory locations may be different. For example, in another computer system:

- memory location 81 may not exist as the memory is smaller; or
- memory location 55 contains important data or instructions that should not be overwritten and therefore the computer system may crash.

By contrast, with an operating system (OS), it is possible to perform computations without interfacing directly with hardware. In the example above, variables (a, b and c) can be used to access data in memory rather than addressing memory locations. The only concern here is that the variable c is able to store the result of a + b.

The operating system (OS) provides a unified environment to users to run their computations in different systems. The operating system (OS) is capable of taking high-level code and translating it in to the machine code that can be executed on a particular computer system.

1.5 Current operating system (OS) trends

Hardware evolution

Due to fast rate of hardware evolution, operating systems (OSs) are more wide-spread than just traditional desktop computers. They can be found on hardware such as:

- mobile devices, such as smartphones and tablets; and
- embedded systems.

Desktop	Mobile	Embedded
• Windows • macOS • Linux	• iOS • Android • Symbian OS	• Windows Embedded

Figure 1.10: Popular operating systems (OSs).

Multiprocessor systems

Definiton

A **multiple processor computer system** makes use of two or more processors and has the ability to allocate tasks between them.

Workstations

A single machine may contain multiple processors and therefore have large computing power.

Distributed and network systems

These computer systems share computing power and peripherals.

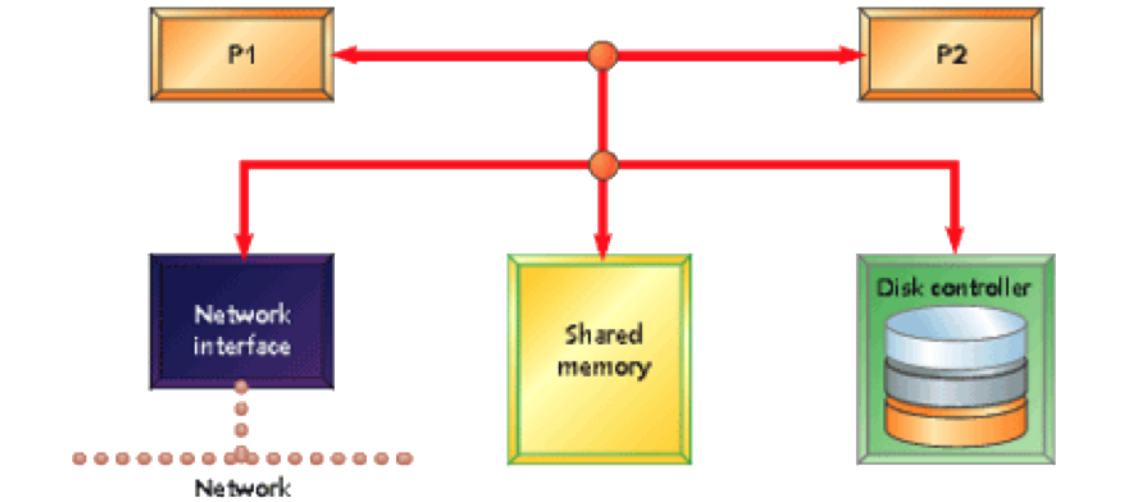


Figure 1.11: Distributed and network systems.

The diagram shows an abstraction of a distributed or network system. P1 and P2 are the connected computer systems that both share memory, access to the disk control and, if the system is a network system, the network interface.

However, there is a distinguishable difference between distributed and network systems.

Similarities	Differences
Consist of multiple systems that are interconnected to exchange information.	In distributed systems, users are not aware of the multiplicity of computer systems available.
	In network systems, users explicitly move/share files, submit jobs for processing and other perform other similar tasks.
	In distributed systems, tasks such as moving/sharing files, submitting jobs for processing and other similar tasks are handled automatically by the operating system (OS).

Evaluation

Advantages	Disadvantages
Increase processor throughput due to the use of parallel processing.	**A more complex operating system is required** in order to be able to interface and manage multiple processor units.
Lower cost than using multiple processors across multiple computer systems because the processors share resources such as the power supply and motherboard.	

Increased reliability because failure of one processor does not affect the other processors and will only slow down the computer system.	

1.6 Operating system (OS) layers

User interfaces

In an operating system (OS), the top layer is the user interface. This is the only layer explicitly visible to the user.

The user interface may be a:

- terminal – text-based command prompt; and/or
- graphical user interface (GUI) – a visual way of interacting with a computer using items such as windows, icons and menus.

History of the graphical user interface (GUI)

The first company to develop a graphical user interface (GUI) was Xerox PARC. They developed the "Alto" personal computer. It had a bitmapped screen and was the first computer to have a "desktop" screen with a graphical user interface (GUI).

The "Alto" personal computer was not a commercial product. However, several thousand units were manufactured and used at Xerox's offices and several universities.

This development was a large influence on the design of personal computers during the late 1970s and early 1980s. Notable examples include:

- Three Rivers PERQ;
- Apple Lisa;
- Apple Macintosh; and
- the first Sun Workstations.

1.7 Kernel mode

Kernel mode vs user mode

Kernel Mode	**User Mode**
Operating systems (OSs) run in kernel mode. This allows: • execution of privileged machine instructions; and • complete access and control of all the hardware.	Other software runs in user mode. In this mode, instructions that affect control of the machine are forbidden. For example: • web browsers; • e-mail software; and • music players.

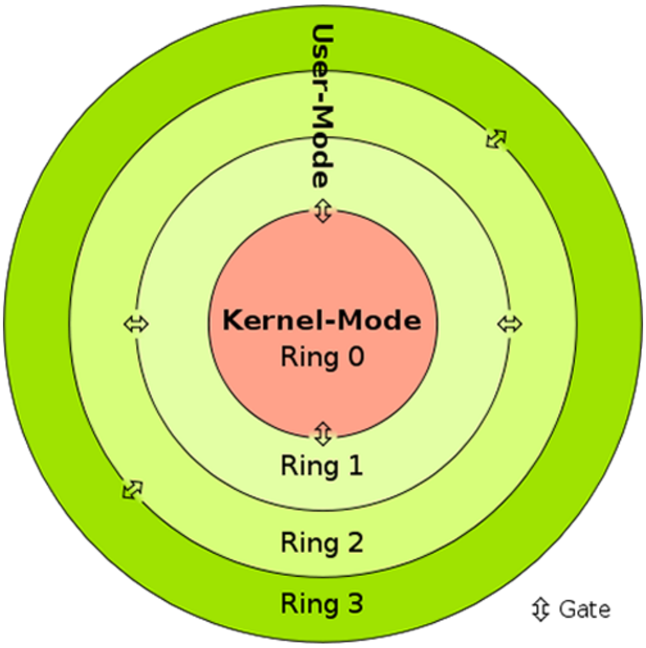


Figure 1.12: Kernel mode and user mode.

Ring 0 represents the kernel mode.

Rings 1-3 represent the user mode.

Kernel mode protection

These rings allow separation between the operating system (OS) and user programs for security and protection purposes.

If user mode had unrestricted access to all of the machine instructions:

- a user could inadvertently obtain a virus or write code that is capable of causing damage to the system, and therefore it is necessary to prevent any instructions from directly controlling the machine;

- a program may use resources unfairly, such as holding the CPU or memory, and therefore harm the execution of other programs; and/or
- sensitive machine instructions could be used improperly which may lead to kernel mode errors.

Kernel mode errors are catastrophic.

Kernel Mode	**User Mode**
A kernel panic represents the operating system (OS) attempting to prevent software causing any harm to the computer system and to recover from a kernel mode error on reboot. An example of a kernel panic is the "Blue Screen of Death" (BSoD) in Microsoft's Windows.	Application errors where an exception was thrown due to an attempt to execute a privileged instruction, this is one that should only be accessed and executed by the kernel mode, represents the operating system (OS) preventing an application from having unrestricted access to all of the machine instructions.

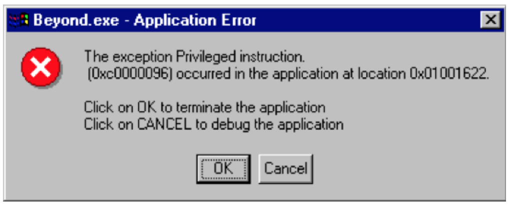


Figure 1.13: Kernel mode errors.

System calls

A **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system on which it is executed. A system call is a way for programs to interact with the operating system.

Some privileged instructions can be called by a programmer via system calls. Operating systems (OSs) contain system calls for which provide a layer of services for user programs to implement some activities/request services. These are usually sensitive or privileged from the kernel.

All interactions with the hardware are implemented via system calls. For example, a system call may occur if an application requires interaction with a peripheral such as a printer.

Invoking a system call is similar to calling a general function. However, the difference is that a general function's code is part of program itself, while the system call code is part of the operating system (OS). Different operating systems (OSs) offer different (limited) sets of system calls.

Call	Description
Process Management	
pid = fork()	Create a child process identical to the parent.
exit(status)	Terminate process execution and return status.
File Management	
n = read(fd, buffer, nbytes)	Read data from a file in to a buffer.
n = write(fd, buffer, nbytes)	Write data to a file
Miscellaneous	
seconds = time(&seconds)	Get the elapsed time since Jan 1. 1970

Figure 1.14: Unix system calls.

2. Process Management

2.1 Programs and processes

Definitions

A **program** is the code written by a programmer.

A **process** (or job/task) shows a program in execution and is a particular instance of a program. These many be shown in a monitoring software, such as Windows Task Manager.

Data are stored values used for the computations by the process.

How it works

A single program may have multiple processes that are currently running.

Each process can share the same code for the program. This is possible as each process uses its own address space, a list of memory locations which the process can read and write. These memory locations contain the program's code instructions and data.

A program is only code however, once it is run, a process is started, and it becomes instructions and data.

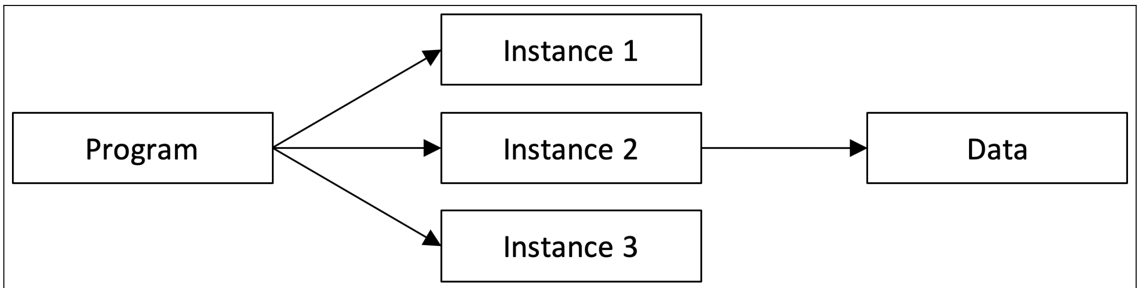


Figure 2.1: Programs and processes.

2.2 Process life cycle

Process creation

A process can be created by:

- a user – a program may be executed by a user, such as via a double-click using a graphical user interface (GUI) or by typing in a command, and "trigger" the processor to load the program's executable file containing the program code; or
- another process – an existing process may create another process by spawning/-forking – the process that creates a new process is called the parent process while the new process is called the child process.

A child process may also spawn a new process forming a tree of processes, as demonstrated in the diagram below.

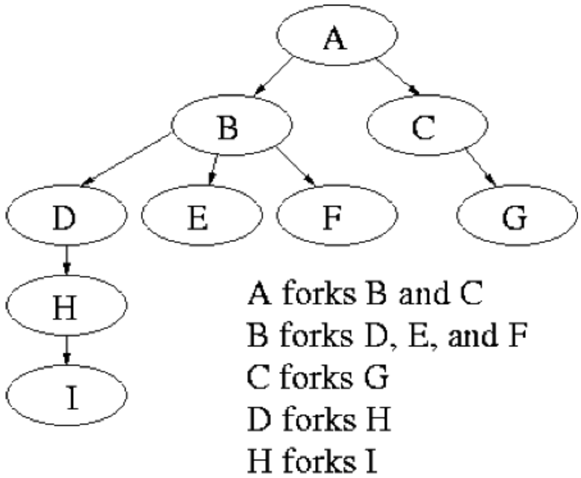


Figure 2.2: Process creation.

Process table and process control block

A **process identification number (PID)** is a unique identifier given to a new process when it is created.

A **process control block (PCB)** holds all of the information about a process. It is created when new process is created.

A **process table** stores the process identification numbers (PIDs) for a process and a pointer to the respective process control block (PCB) for that process. This is managed by the operating system (OS).

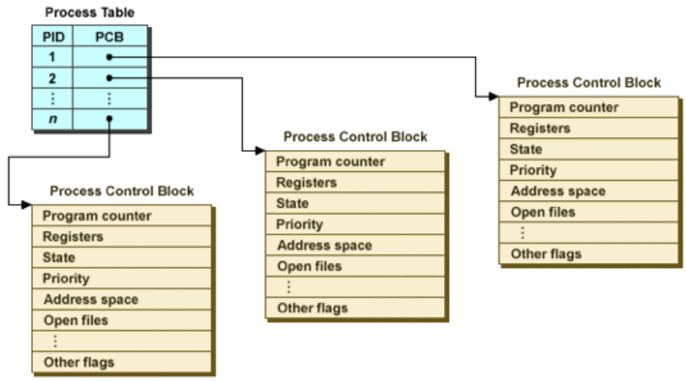


Figure 2.3: Process table with respective process control blocks.

The process descriptor fields in the process control block (PCB) may differ between operating system (OS). An example of possible process descriptor fields is shown by those used by the Minix operating system (OS) are shown below.

Process Management	**Memory Management**	**File Management**
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Effective uid
Time when process started	Process ID	Effective gid
CPU time used	Parent process	System call parameters
Children's CPU time	Process group	Various flag bits
Time of next alarm	Real uid	
Message queue pointers	Effective uid	
Pending signal bits	Real gid	
Process ID	Effective gid	
Various flag bits	Various flag bits	

Figure 2.4: Process descriptor fields in Minix.

Although Minix is a fully-featured operating system (OS), it is a small operating system (OS) and therefore the processor descriptor fields are less complex than other operating systems (OSs).

Three-state model

The **three-state model** shows how a process, once initiated, can be in one of three states. The current state of a process is stored in its respective process control block (PCB).

A process, once initiated, can be in one of three main states:

- running – actually using the CPU to perform a task;
- ready – ready to run but waiting for the CPU as it has not yet had time on the CPU has been temporarily stopped to let another process run; or

- blocked – unable to run until some external event occurs, such as:
  - waiting for an interrupt, this is a message from the hardware saying that a resource is now available to read from – such as waiting for an input/output (I/O) operation to complete; and
  - waiting for another process to finish accessing a shared resources – for example: a file; memory; or an external peripheral, such as a printer.

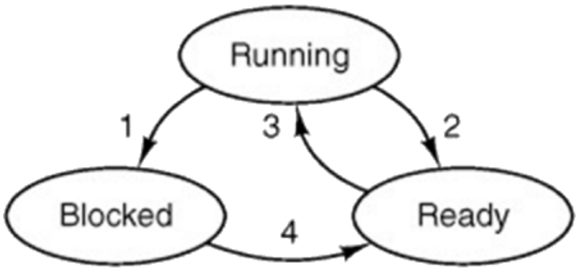


Figure 2.5: Process states.

In reference to the diagram above, transitions can occur between process states.

Transition	**Diagram Number**	**Explanation**
running → blocked	1	**Process blocks for input.** For example, if the process is waiting for some input from I/O.
running → ready	2	**Scheduler picks another process.** The process has had opportunity to run and is flagged as no longer currently running, so that another process can run.
ready → running	3	**Scheduler picks this process.** The next process that is ready to run is set to running to allow access to the CPU.
blocked → ready	4	**Input becomes available.** The process has received input from I/O or the process sends an interrupt and the interrupt service routine (ISR) is executed, the scheduler is called to transition the process from blocked to ready.

Figure 2.6: Transitions between processor states.

The transition between process states is dependent on the scheduling algorithm used. A clock is used to send a signal to stop the current process, move it from running to ready and then run the scheduler to find out what process should be processed next and the next process will be made to running.

State queues

At any time, a process is in only one state.

Running processes

At any time, at most one process is in the running state. This is because a single-core processor is only able to process one instruction coming from a single core at a time. If a computer system has a multi-core processor, this rule applies to each individual core on the processor, rather than the processor as a whole, as they are able to complete parallel execution.

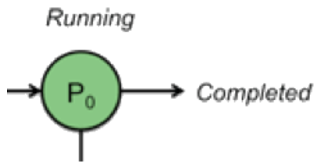


Figure 2.7: A running process.

Ready processes

There may be a queue of processes in the ready state.

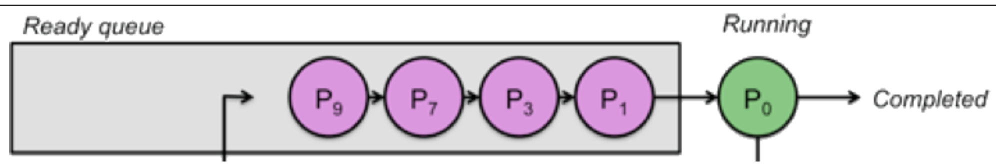


Figure 2.8: Ready processes (purple) and a running process (green).

Blocked processes

There may be several queues of processes in the blocked state, where each queue represents one resource for which processes in that queue are waiting.

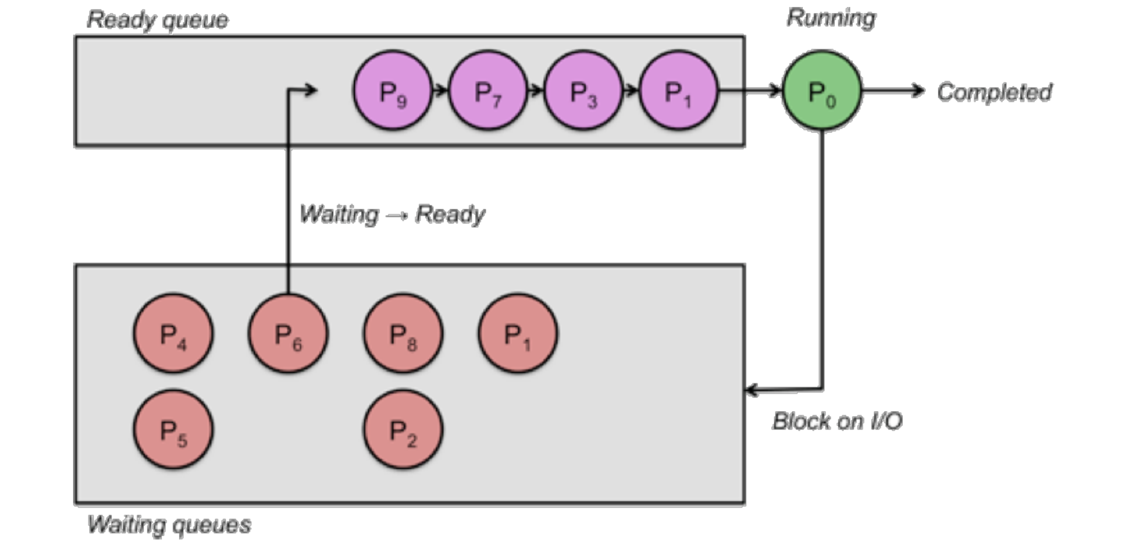


Figure 2.9: Blocked processes (red), ready processes (purple) and a running process (green).

Process termination

Process termination is the end of life for a process, this can occur in two general ways.

Voluntary termination

Voluntary termination of a process represents the end of life for a process where its termination was intended by the user or the programmer.

This can be a:

- normal exit – the process has done its work; or
- error exit – the process itself handles and "catches" an error – for example, some try-catch code is implemented to check if a condition is met, such as if the definition of a variable is present.

Figure 2.10: Voluntary termination due to an error exit.

Involuntary termination

Involuntary termination of a process represents the end of life for a process where its termination as not intended by the user or the programmer.

This can be due to:

- a fatal error – an error is detected by the operating system's (OS's) protected mode – for example, an exception has been thrown due to reference to a non-existent memory location or division by zero; or
- being killed by another process – a process may execute a system call that causes the operating system (OS) to kill another process – this process may have control over the killed process and this may be due to that process being the parent process of the killed process.

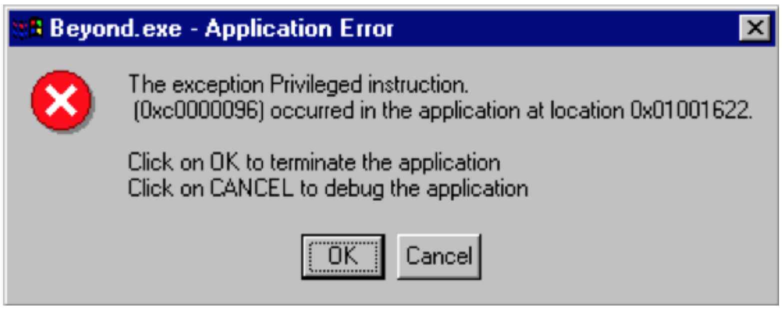


Figure 2.11: Voluntary termination due to an error exit.

2.3 Program execution

The simple fetch-execute cycle

Definition

The **fetch-execute cycle** is an operational process in which a computer system retrieves a program instruction from its memory, determines what actions the instruction dictates, and carries out those actions.

How it works

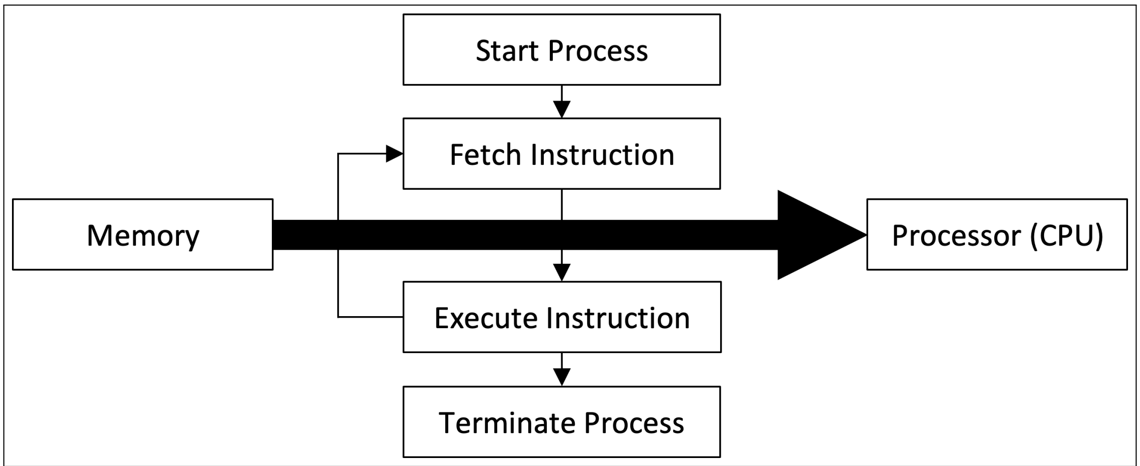


Figure 2.12: Simple fetch-execute cycle.

Once a process is started, instructions are fetched from memory and executed using the CPU. This process will continue until the process is terminated.

This predictable cycle is only feasible to an extent as some processes may be slow or blocked and other may require immediate attention and cannot wait for the current process to terminate. For example, if a process blocks the processor (CPU) because it is waiting for an event to occur, such as a printer to finish its job, or if a high-priority process is supposed to execute as soon as possible. In these cases, interrupts are required.

Interrupts

Definition

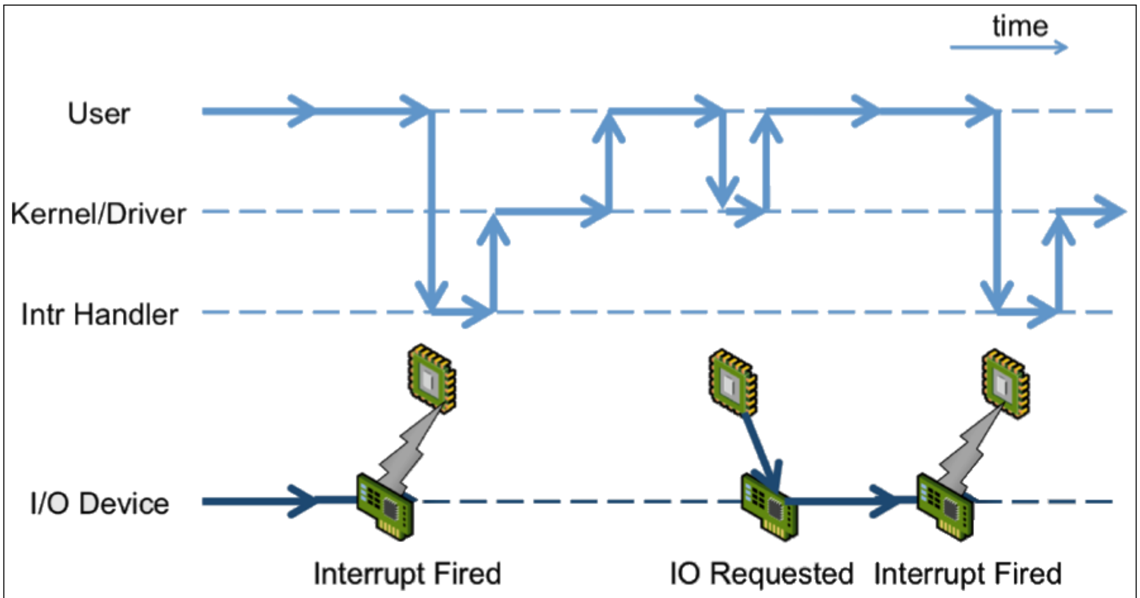
An **interrupt** is a signal sent to the processor indicating that an event caused by hardware or software requires immediate attention.

Types of interrupts

- Input/output (I/O) interrupt – Caused by an input/output (I/O) device to signal completion or an error.

- Timer interrupt – Caused by a processor timer and is used to alert the operating system (OS) at specific instants.
- Program interrupt – Caused by error conditions within user programs or fatal errors.

When do interrupts occur?



A process requiring immediate attention is waiting for the processor (CPU).
For example, an input/output (I/O) event occurs.

↓

The currently running process' execution is suspended.

↓

Processor switches execution to another process.

↓

When input/output (I/O) completes, an interrupt will occur.

↓

The execution is diverted to the interrupt handling routine.
An interrupt handling routine (ISR) contains the operations that are performed to deal with an interrupt. Different types of interrupts can have different routines.

↓

The suspended program may restart, if conditions are met.

Figure 2.13: Interrupt occurring due to input/output (I/O) event.

Interrupts enable operating systems (OSs) to oversee several programs and input/output (I/O) events simultaneously.

This also means that single-core processors can effectively emulate the way in which multi-core processors deal with multiple instructions at a given time by switching between instructions intelligently. Due to the high clock speeds of modern processors, it is easy to give the illusion that true multi-tasking, where two instructions are being processed at once, is taking place on a single-core processor.

Updated fetch-execute cycle

Given the introduction of interrupts, it is now necessary to update the fetch-execute cycle in order to include the possibility of an interrupt.

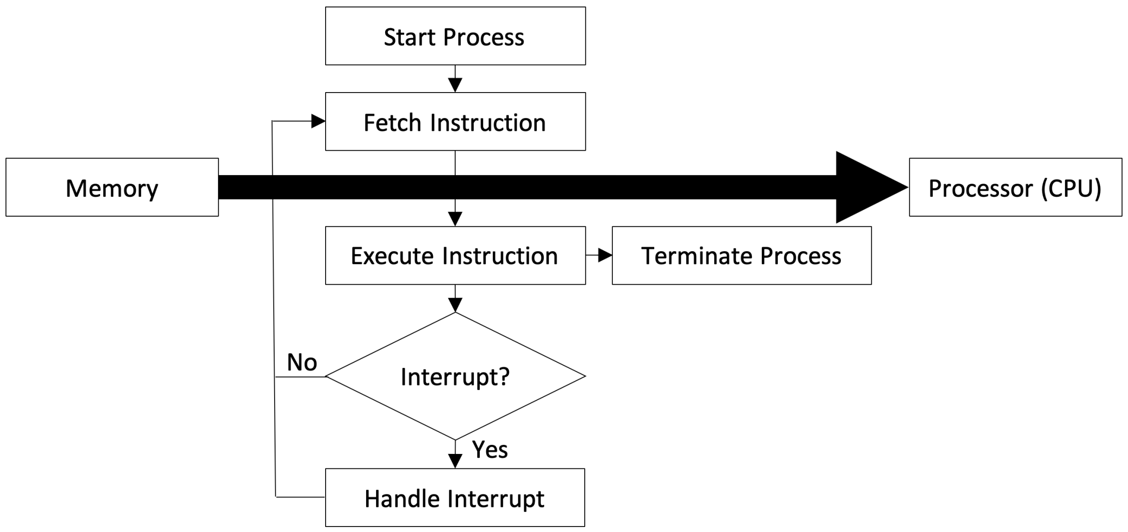


Figure 2.14: Updated fetch-execute cycle.

2.4 Concurrency

Definition

Concurrency describes the ability for a program to be decomposed in to parts that can run independently of each other. This means that tasks can be executed out of order and the result would still be the same as if they are executed in order.

Why is concurrency required

Concurrency allows the processor (CPU) to run several processes. An example of this was previously shown by an interrupt occurring due to an input/output (I/O) event occurring.

Interleaving

Concurrency is able to achieve multitasking, that does not require parallel execution, by performing interleaved execution.

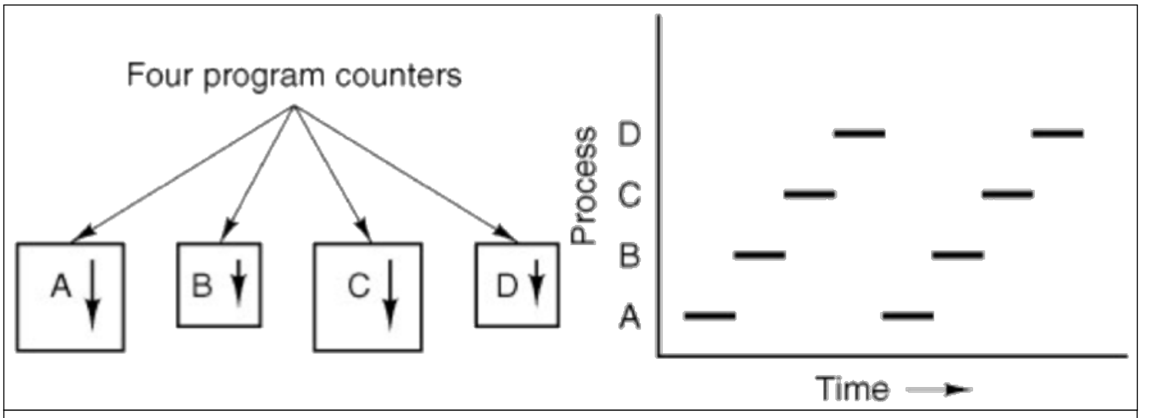


Figure 2.15: Interleaving.

2.5 Process scheduling

The scheduler

Definition

A **scheduler** uses a scheduling algorithm to determine how to share processor time.

How it works

After an input/output (I/O) system call or interrupt handling, control is passed to the scheduler to decide which process to execute next.

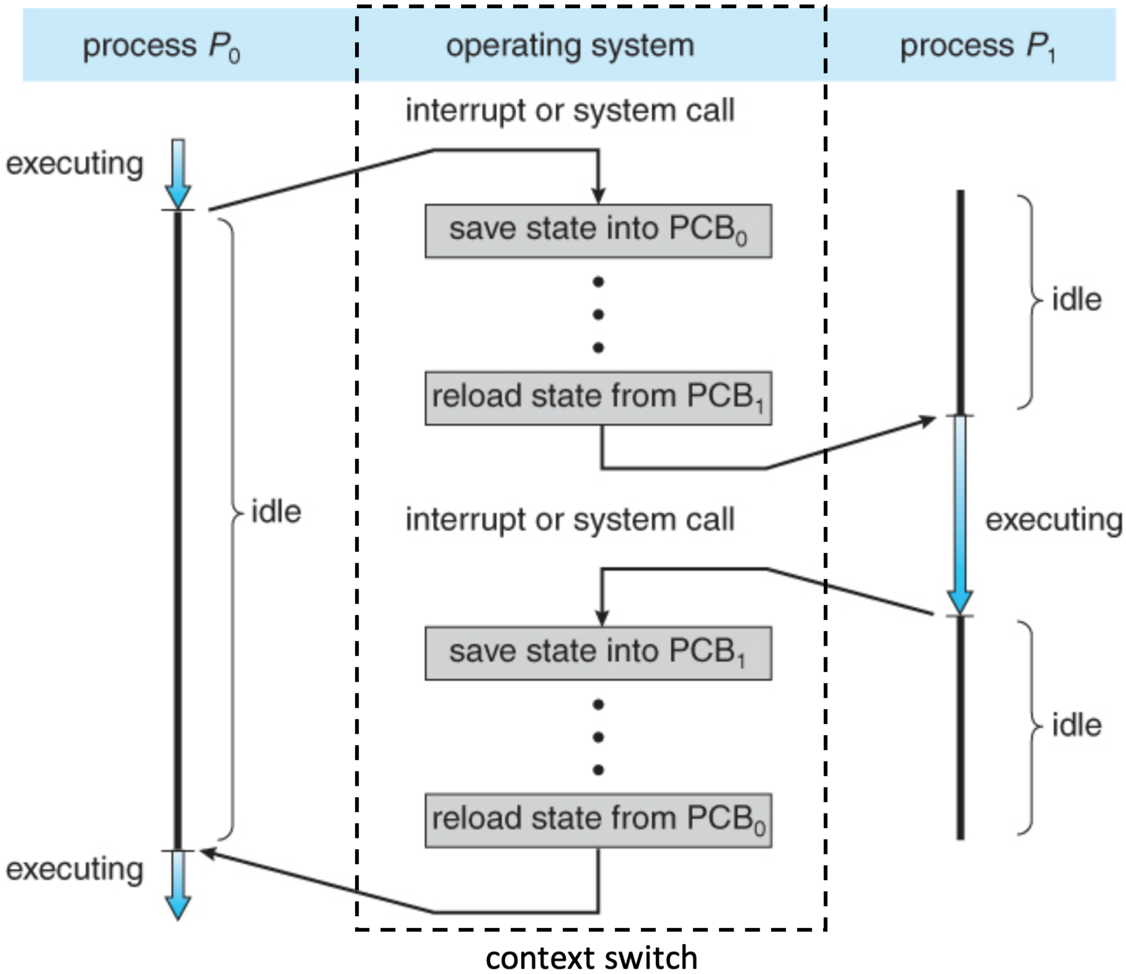


Figure 2.16: The scheduler.

The scheduler checks if the current process is still the most suitable to run at this moment in time. If it is control is returned to the process, otherwise:

- the state of the current process is saved in the process control block (PCB);
- the state of the most suitable process is retrieved from the process control block (PCB); and
- control is transferred to the newly selected process at the point indicated by the restored program counter (PC).

The action of storing the state of the current process and activating another process is called a context switch.

Context switching must be minimised to reduce overheads created by copying the state of processed and the time taken to perform the switch. However, it is still regarded as important to context switch when appropriate to avoid longer waiting times in the event of a blocked process.

Scheduling

Definition

Scheduling is the act of determining the optimal sequence and timing of assigning execution to processes.

Scheduling criteria

Different scheduling criteria may be selected depending on the use case for a given computer system.

- CPU utilisation – Aims to keep the CPU as busy as possible.
- Efficiency – Aims to maximise system throughput.
- Fairness – Aims to be fair to all running processes or to all users on a multi-user operating system (OS).

This means that different policies and algorithms for scheduling will exist to match these criteria.

Scheduling policies

A **non-preemptive scheduling policy** is one that allows processes to run until complete or incurring an input/output (I/O) wait. These scheduling policies can be described as passive.

A **preemptive scheduling policy** is one that allows processes to be interrupted and replaced by other processes, generally through timer interrupts.

Scheduling algorithms

Definitions

Arrival time is the instant at which a process is first created.

Service time is the time that it takes for a process to complete if it is in continuous execution.

The **waiting time** for a process is the sum of time spent in the ready queue during the life of the process. This does not include time that the process is blocked or waiting for input/output (I/O).

In order for a scheduling algorithm to be deemed as fair to all processes, the ratio between waiting time and run time should be about the same for each process.

Case study

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Figure 2.17: Case study.

The case study shows different processes (A-E) that all have different arrival times and different service times.

The following examples of scheduling algorithms will refer to this case study to demonstrate their function.

First come, first served (FCFS) / First in, first out (FIFO) – Non-preemptive

In this algorithm, the first process to arrive is assigned to the processor (CPU) until it is finished. Meanwhile, any other processes that come along are queued up waiting to be processed.

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

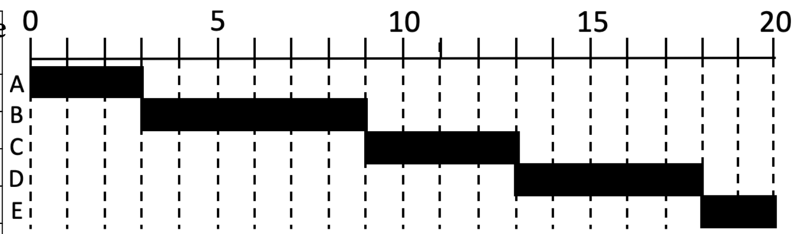


Figure 2.18: Temporal diagram for FCFS / FIFO.

<table>
<tr><th>Advantages</th><th>Disadvantages</th></tr>
<tr><td>Simple to implement.</td><td>Does not consider the priority of a process and therefore the important processes may not be completed quickly.</td></tr>
<tr><td></td><td>Prevents other processes from starting while another is in progress and therefore, if processes are of varying sizes, there may be inefficiencies since a single process may take a long time to complete therefore leaving the user waiting before they can perform any other actions.</td></tr>
<tr><td colspan="2">Favours long processes as all processes are given the opportunity to run until completion and therefore, some shorter processes may not be able to start processing until the longer processes are finished.</td></tr>
</table>

Shortest job first (SJF) – Non-preemptive

In this algorithm, the process with the shortest estimated run time is assigned to the processor (CPU) until it is finished. Meanwhile, any other processes that come along are queued up waiting to be processed.

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

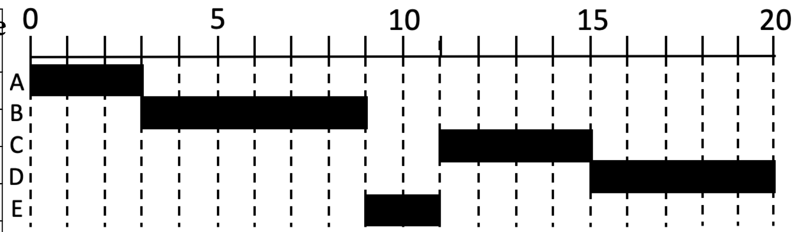


Figure 2.19: Temporal diagram for SJF.

Advantages	Disadvantages
Simple to implement.	**Does not consider the priority of a process** and therefore the important processes may not be completed quickly.
Shorter processes are processed quickly because they take precedence.	**Relies on an estimation** of how long a process will take which could be incorrect.
Minimises the average time taken to complete a process because the shortest processes take precedence.	**Prevents other processes from starting** while another is in progress and therefore, if processes are of varying sizes, there may be inefficiencies since a single process may take a long time to complete therefore leaving the user waiting before they can perform any other actions.
Favours short processes as the processes with the shortest estimated time are given the opportunity to run until completion and therefore, some longer processes may not be able to start processing until there are no more shorter processes currently running or ready to be processed.	

Shortest remaining time (SRT) – Preemptive

In this algorithm, the process with the shortest estimated remaining time is assigned to the processor (CPU). If a process becomes ready that has a shorter remaining time, it will be pre-empted to allow the new process to start and the scheduler is switched to that new process.

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

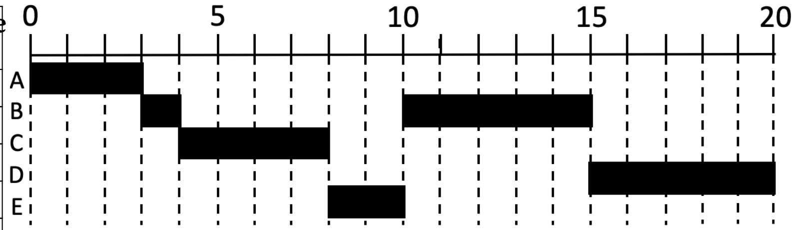


Figure 2.20: Temporal diagram for SRT.

Advantages	Disadvantages
High throughput because the number of processes completed is high due to the shortest processes taking precedence.	**Does not consider the priority of a process** and therefore the important processes may not be completed quickly.
Shorter processes are processed quickly because they take precedence.	**Relies on an estimation** of how long a process will take which could be incorrect.
	Can be inefficient if a large process is in progress and shorter processes are being added to the queue because they will take precedence.
Favours short processes as the processes with the shortest estimated time are given the opportunity to run until a process with a shorter estimated time is ready and therefore, some longer processes may not be able to start processing until there are no more shorter processes currently running or ready to be processed.	

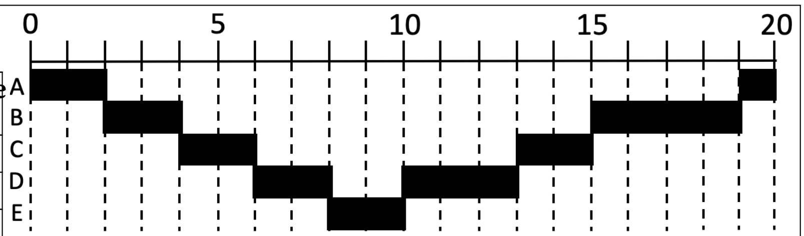
Highest response ratio next (HRRN) – Preemptive

In this algorithm, the process with the highest priority is assigned to the processor (CPU). If a process becomes ready that has a higher priority, it will be pre-empted to allow the new process to start and the scheduler is switched to that new process.

The priority of a process may be based on memory, time and/or any other resource requirement such as:

$$\frac{\text{waiting time} + \text{run time}}{\text{run time}}$$

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



This diagram assumes that the processes are assigned a letter in reverse order of priority, such that E has the greatest priority and A has the lowest priority.

Figure 2.21: Temporal diagram for HRRN.

Advantages	Disadvantages
High priority processes are processed quickly because they take precedence.	**Relies on an estimation** of the priority of a process which could be incorrect.

Round robin (RR) – Preemptive

In this algorithm, each process is dispatched to the processor (CPU) on a "first in, first out" (FIFO) basis with a fixed time quantum.

Each time quantum is typically 10-20ms. Modern processor (CPU) clock frequencies are typically greater than 2GHz, which implies clock periods of 5×10^{-7}ms. This shows that the given time quantum is relatively large compared to a typically processor's (CPU's) clock period.

If a process experiences a timeout, this will mean that it has run over its fixed time quantum. In which case, the process will be interrupted and returned to the back of the queue.

A system designer may wish to choose a time quantum that is most appropriate for a given system. This may be done by measuring the average service time and waiting time for the processes that will be running on the system and design the fixed time quantum around these figures. It may be that a system designer wishes to minimise the number of processes that are interrupted by another process.

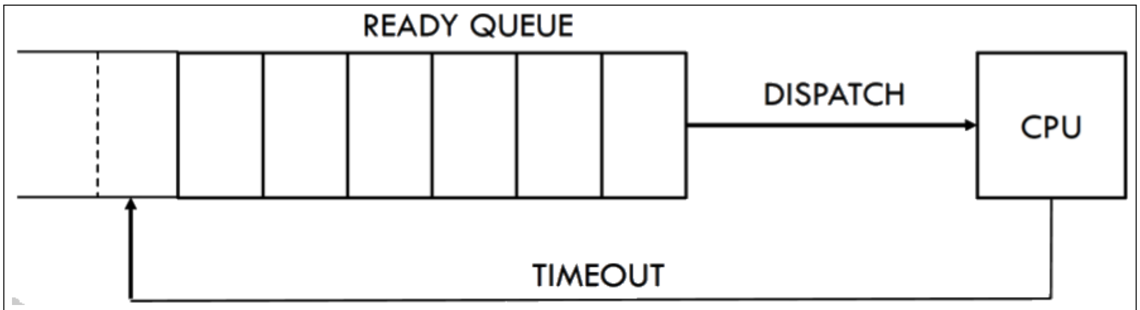


Figure 2.22: RR.

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

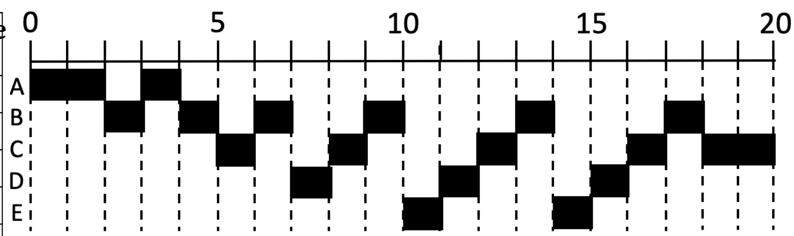


Figure 2.23: Temporal diagram for RR with a fixed time quantum of 1 (q = 1).

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

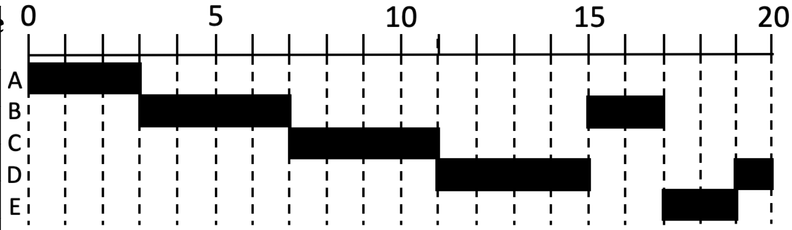


Figure 2.24: Temporal diagram for RR with a fixed time quantum of 4 (q = 4).

Advantages	Disadvantages
Simple to implement.	**Heavy overhead** due to continuous context switches.
Suitable for some types of computer systems, such as those which will be running processes of similar priority and size.	**Does not consider the priority of a process** and therefore the important processes may not be completed quickly.
	Does not consider the size of a process and therefore, if processes are of varying sizes, there may be inefficiencies since a single process may take a long time to complete thus leaving the user waiting before they can perform any other actions.

Multi-level queueing

Definition

Multi-level queueing is a queue with a predefined number of levels which consist of several independent queues.

How it works

Multi-level queueing makes use of other existing scheduling algorithms.

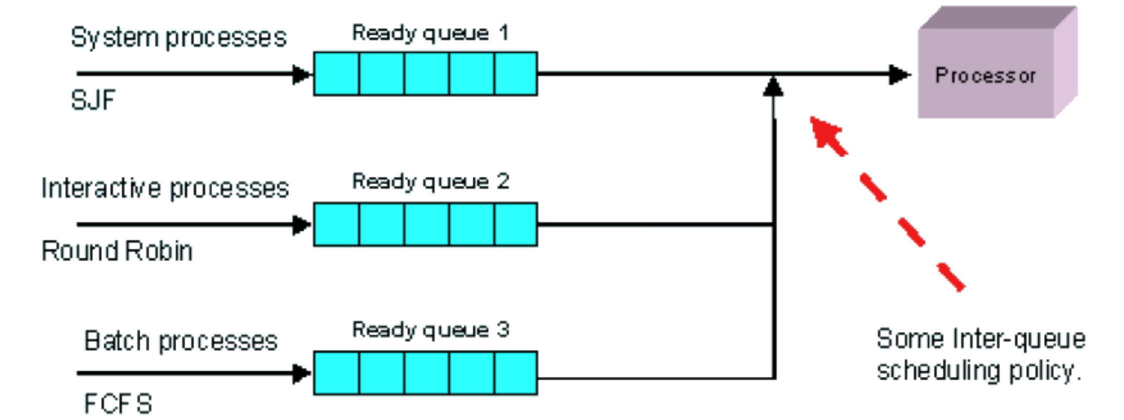


Figure 2.25: Multi-level queueing.

Each queue has a different priority; the top queue has the highest priority and the bottom queue has the lowest priority. Processes are able to move between queues if their priority changes.

There is some form of inter-queue scheduling policy that governs the assignment of processes from each queue to the processor (CPU).

The queues in a multi-level queueing system may differ between operating system (OS). For example, the scheduler used by the Minix operating system (OS) uses multi-level queueing and implements 16 queues.

Advantages	Disadvantages
Allows scheduling optimisation as a system designer may be able to leverage the advantages of a range of different scheduling algorithms.	
Maintains common processes as it is possible to split processes in to different queues depending on their nature. For example, input/output processes could be in one queue while processor (CPU) processes are in another queue.	
Helps to prevent bottlenecks because input/output (I/O) devices are slower than the processor speed and therefore maximising processes involving input/output (I/O) devices ensures that these devices are continuously busy.	

3. Threads and Concurrency

3.1 What are threads?

Definition

A **thread** is an independent path/sequence of execution with in a process and can be managed independently by a scheduler. A process many contain many threads.

Multi-threading

How it works

As mentioned in the previous section, a process (or job/task) shows a program in execution and is a particular instance of a program. By default, these processes are run by means of "single execution thread".

However, different parts of the same process could be parallelised in order to allow multi-threading.

Multi-threading provides a way of improving application performance and therefore improving the efficiency and/or usability of a computer system.

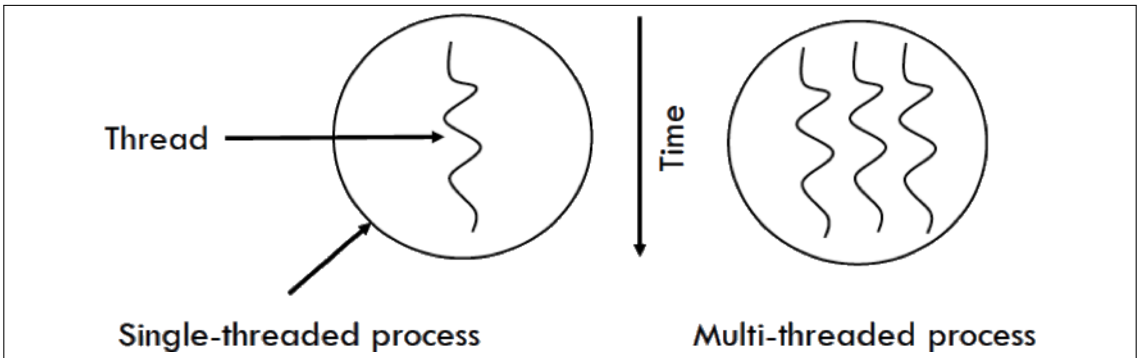
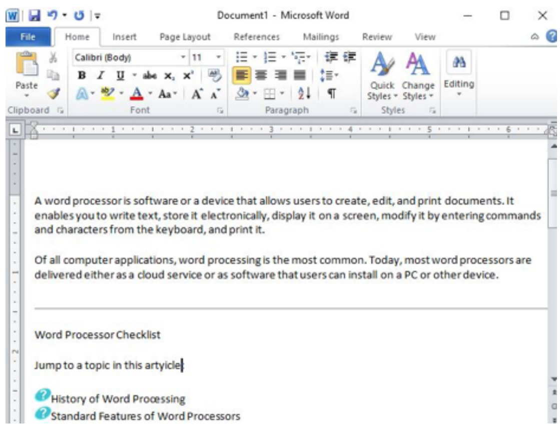


Figure 3.1: Single-threaded vs multi-threaded.

Example



In this example, a thread could be assigned to each of the following tasks:

- user input, such as keyboard and mouse input;
- auto-saving document;
- spell-checking document; and
- printing in the background.

This is possible as all of these tasks can be run in parallel as they do not require information from one another.

Figure 3.2: Example of multi-threading in Microsoft Word.

Threads vs processes

Comparison

Threads and processes share some similarities however, there are also some distinct differences.

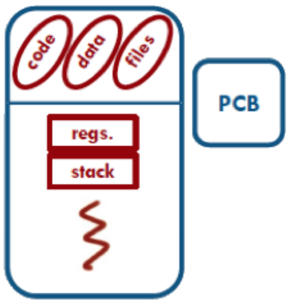


Figure 3.3: Single-threaded process.

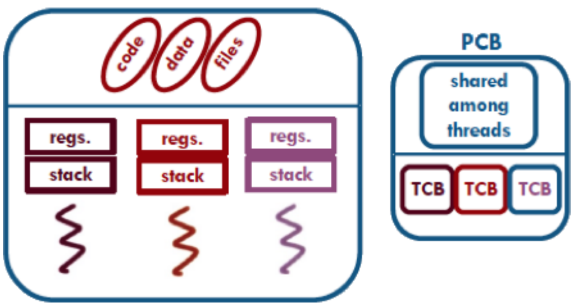


Figure 3.4: Multi-threaded process.

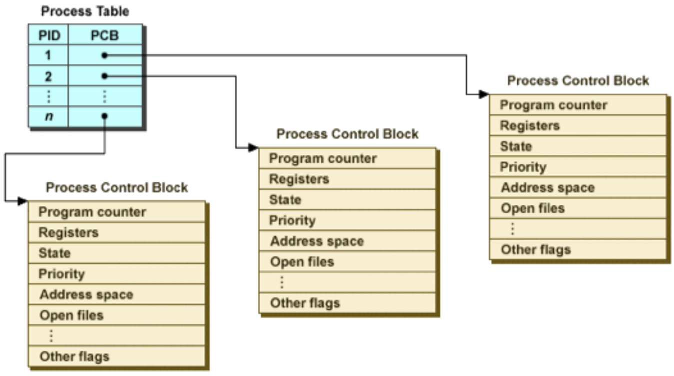


Figure 3.5: Process table and process control blocks.

	Threads	**Processes**
Similarities	Sequential flow of control with start and end.	Sequential flow of control with start and end.
	At any time, a thread has a single point of execution.	At any time, a process has a single point of execution.
	Has its own execution context, stack (history) and program counter stored in a thread control block (TCB).	Has its own execution context, stack (history) and program counter stored in a process control block (PCB).
	Follows the three-state model in which the thread can be running, blocked or ready.	Follows the three-state model in which the process can be running, blocked or ready.
	Context switching can happen for threads.	Context switching can happen for processes.
	A thread can spawn another thread.	A process can spawn another process.
	A thread is often called a lightweight process.	
Differences	A thread cannot exist on its own, instead it exists within a process.	A process does not require a parent entity.
	Usually created and/or controlled by a process.	A process is not typically created and/or controlled by another process.
	Threads can share process properties, including memory and open files.	Processes cannot share process properties with other processes.
	Inexpensive creation and context switching as does not require separate address space.	Expensive creation and context switching as requires separate address space.
	When running multiple threads concurrently, they share an address space.	When running multiple processes concurrently, they share resources, such as memory, disk and printers.

Figure 3.6: Threads vs processes.

Properties

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 3.7: Threads vs processes.

Process properties are shared between threads. Thread properties are local and private to each thread.

3.2 Sequential and concurrent programming

Sequential programming

Definition

Sequential programming is the traditional activity of constructing a computer program using a sequential programming language.

How it works

This involves a programming methodology that assumes statements are executed in order/sequence.

Programs written using sequential programming are assumed to execute on a single-CPU system and have a single thread of control.

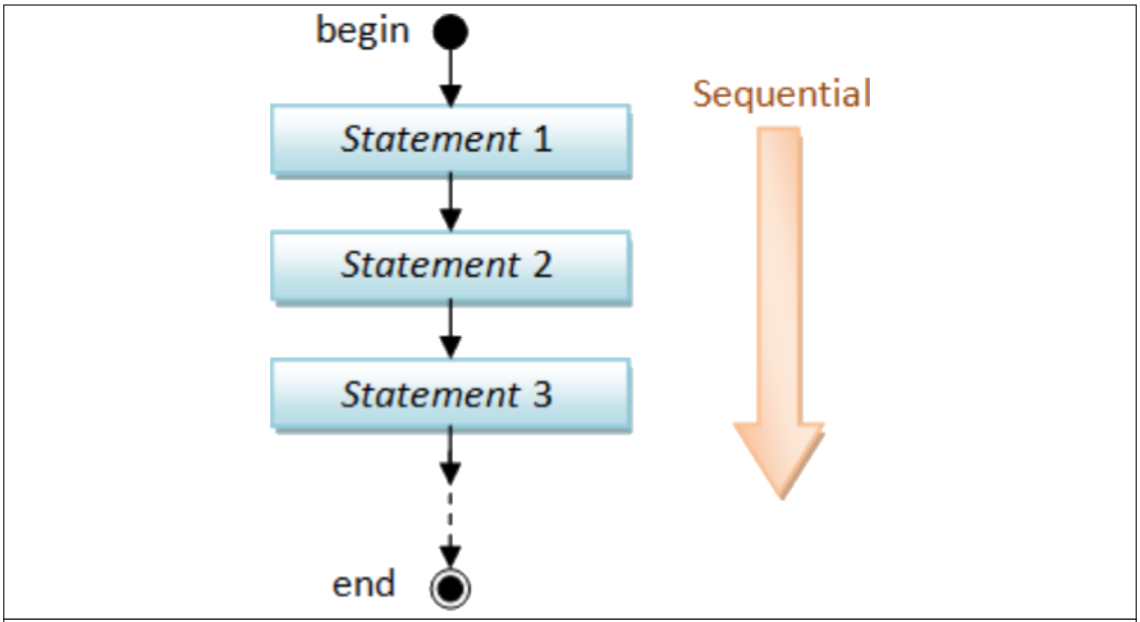


Figure 3.8: Sequential program.

Evaluation

Advantages	Disadvantages
No additional support required from the programming language.	**Lower processor throughput than concurrent programming** as it cannot benefit from multitasking or concurrent processing.
No additional support required from the operating system as most old-school operating systems were generally single-threaded and therefore later generations of operating systems typically inherit this functionality.	**Multiple computer systems that each have their own CPU may yield a higher cost than multi-CPU systems** as more will be spent on resources, such as the power supply and motherboard, that could be otherwise shared in a multi-CPU system.
	Lower reliability than a multi-CPU system because, in the case of the failure of the processor, there is no redundancy.

Concurrent programming

Definition

Concurrent programming is the activity of constructing a computer program that takes advantage of concurrency allowed by the use of multiple threads of control.

How it works

Multiple threads of control allow a given process to perform multiple computations in parallel and to control simultaneous external activities.

The program may be run on both:

- a single-CPU system – where the computer program will take advantage of multi-tasking; and
- a multi-CPU system – where the computer program will take advantage of true parallelism.

Evaluation

Advantages	Disadvantages
Increase processor throughput due to the use of multitasking in a single-CPU system or parallel processing in a multi-CPU system.	**Requires support from the programming language** as it must implement techniques to deal with multitasking on a single-CPU system and/or parallel processing on a multi-CPU system.
A multi-CPU system generally yields a lower cost than using multiple CPUs across multiple computer systems because the processors share resources such as the power supply and motherboard.	**Requires support from the operating system** as it must support multi-threading in order to allow multitasking on a single-CPU system and/or interface and manage multiple CPU units on a multi-CPU system.
Increased reliability in a multi-CPU system because failure of one processor does not affect the other processors, instead the computer system may experience lower performance until fixed.	

3.3 Sequential execution

Definition

Sequential execution is where the execution of threads in a sequential program is executed in sequence/order with no overlapping.

Order and precedence

Explanation

In sequential execution, there is only one possible sequence of execution. This is because a sequential program gives the system strict instructions on the order of executing the statements in the program.

Importance

For example, a simple hypothetical program could be:

P;

Q;

R;

This tells the computer system that the statements must be executed in the order they are written, such that:

- P must precede Q; and
- Q must precede R.

High level

The importance of the order of precedence can be highlighted by demonstrating this idea in a high-level programming language.

Given the following program written in a high-level language:

x = 1;

y = x + 1;

x = y + 2;

it is possible to see that the final values of x and y depend on the order of execution of the statements.

System level

Given the following program written in a high-level language:

x = 1; **P**

y = x + 1; **Q**

x = y + 2; **R**

where each statement is assigned a letter respectively, each statement may be compiled in to several machine instructions.

Statement **P** is treated as a single machine instruction:

- **P1**: store 1 at the memory address of x.

Statement **Q** is broken in to three machine instructions:

- **Q1**: load the value of x in to a CPU register;
- **Q2**: increment the value in this register by 1; and
- **Q3**: store the value in this register at the memory address of y.

Statement **R** is broken in to three machine instructions:

- **R1**: load the value add of y in to a CPU register;

- **R2**: increment the value in this register by 2; and
- **R3**: store the result at the memory address of x.

The nature of sequential execution

The execution of statements **P**, **Q** and **R** at the program level (or high-level) as

$\mathbf{P} \rightarrow \mathbf{Q} \rightarrow \mathbf{R}$

implies that the execution at the system level is as follows

$\mathbf{P1} \rightarrow \mathbf{Q1} \rightarrow \mathbf{Q2} \rightarrow \mathbf{Q3} \rightarrow \mathbf{R1} \rightarrow \mathbf{R2} \rightarrow \mathbf{R3}$,

given that **P** is compiled to a single machine instruction, whilst **Q** and **R** are compiled to three machine instructions.

Sequential execution has the following assumptions:

- total ordering – there is single-threaded computation, and therefore no overlap in the execution of the statements;
- deterministic – the same input will always result in the same output; and
- sequence – users will specify a strict sequence of steps required in order to achieve a desired goal.

However, this does not apply in many practical applications, for which a sequence of steps is not required.

3.4 Introduction to concurrent execution

Definition

Concurrent execution is where the execution of threads in a concurrent program is occurring asynchronously, meaning that the order in which tasks are executed is not predetermined.

The squares example

In this hypothetical example, a person desires to have a list with the results of all of the squares (2^n) from 1 to 100000.

A group of 100000 people are spilt in to heavily uneven teams and assigned the same task to complete all of the calculations in order to achieve the desired result.

It is given that each calculation takes n amount of time.

Team 1		Team 2	
Number of members	1	Number of members	99999
Strategy	One person should complete all of the calculations.	Strategy	Each member is assigned a number between 1 to 100000. Each member should calculate the respective square for the number they are assigned.
Time taken	100000n	Time taken	n

This shows that Team 2 was 100000x faster than Team 1. This was because it was possible to decompose the larger task in to smaller sub-tasks and assign each of those tasks to a separate resource, which in this case is one person.

This example forms that basic concept of concurrent execution.

The nature of concurrent execution

Concurrent execution dismisses many of the assumptions required for sequential execution.

Calculations may be carried out without total ordering. As a result, calculations may be carried out in parallel and overlapping is therefore allowed.

In the example above, each individual person in team 2 carried out their operations in sequence.

In the example above, the operations in the whole computation can be viewed as being in partial order. However, the order does not matter here because there is no dependency between the calculations. This is because the output from any given calculation is not required as an input to any other given calculation.

However, in general, concurrent execution is non-deterministic, and therefore the same input generally means different output due to ordering. This is because there are many cases where the order of operation does matter.

3.5 Interleaving

Why is interleaving required?

Concurrent execution on a computer system with a multi-core CPU or multiple CPUs can make use of parallel processing in order to run threads asynchronously. However, this is not possible on a computer system with a single-CPU that consists of only one core.

As a result, interleaving is used in order to switch execution between threads.

It is important to note that the operations within each thread are strictly ordered, but the interleaving of the operations are not ordered and are interleaved in an unpredictable order.

Calculating interleavings

Formula

It is possible to calculate the number of interleavings given the formula:

$$\text{number of interleavings} = \frac{(t_1 + t_2 + ... + t_n)!}{t_1!t_2!...t_n!}$$

where

- t_n represents the number of statements/operations in each thread.

For example, in a concurrent program that has two threads, the formula may be adjusted to:

$$\text{number of interleavings} = \frac{(n+m)!}{n!m!}$$

where

- n represents the number of statements/operations in the first thread (t_1); and
- m represents the number of statements/operations in the second thread (t_2).

Example

A high-level concurrent program may spawn new threads.

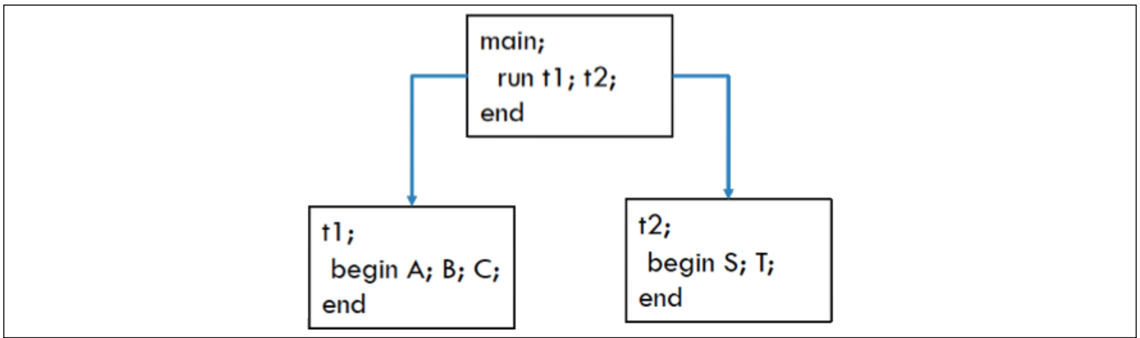


Figure 3.9: Concurrent program spawning threads t_1 and t_2.

In this example, the concurrent program spawns the threads t_1 and t_2 where:

- t_1 has three statements – A, B and C; and
- t_2 has two statements – S and T.

There are two threads and therefore it is possible to use the adjusted formula to calculate the number of interleavings in this concurrent program:

$$\text{number of interleavings} = \frac{(n+m)!}{n!m!}$$
$$\text{number of interleavings} = \frac{(3+2)!}{3! \times 2!}$$
$$\text{number of interleavings} = \frac{6!}{3! \times 2!}$$
$$\text{number of interleavings} = \frac{6 \times 5 \times 4 \times 3 \times 2 \times 1}{(3 \times 2 \times 1) \times (2 \times 1)}$$
$$\text{number of interleavings} = \frac{120}{6 \times 2}$$
$$\text{number of interleavings} = \frac{120}{12}$$
$$\text{number of interleavings} = 10$$

There are 10 possible interleavings, thus yielding 10 possible different execution sequences.

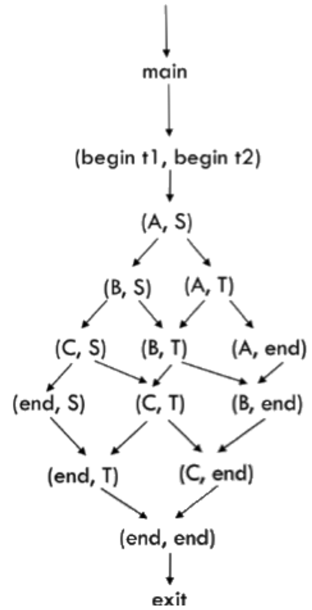


Figure 3.10: Visual representation of the possible execution sequences.

A run of the program corresponds to an interleaving sequence. Each interleaving sequence determines a unique sequence of executing the statements. Repeated runs with the same input will likely trace different interleavings.

Growth of interleavings

The number of interleavings grows extremely quickly given an increase in:

- the number of threads in the concurrent program; or
- the number of statements/operations in one or more of the concurrent program's threads.

This can be demonstrated by increasing the number of operations in the previous example:

- t_1 now has four statements – `A`, `B`, `C` and `D`; and
- t_2 now has five statements – `S`, `T`, `U`, `V` and `W`.

Therefore:

$$\text{number of interleavings} = \frac{(n+m)!}{n!m!}$$
$$\text{number of interleavings} = \frac{(4+5)!}{4! \times 5!}$$
$$\text{number of interleavings} = \frac{9!}{4! \times 5!}$$
$$\text{number of interleavings} = \frac{9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1}{(4 \times 3 \times 2 \times 1) \times (5 \times 4 \times 3 \times 2 \times 1)}$$
$$\text{number of interleavings} = \frac{362880}{24 \times 120}$$
$$\text{number of interleavings} = \frac{362880}{2880}$$
$$\text{number of interleavings} = 126$$

3.6 User and kernel threads

User threads

User threads are created and managed by a user level library, typically without the knowledge of the kernel.

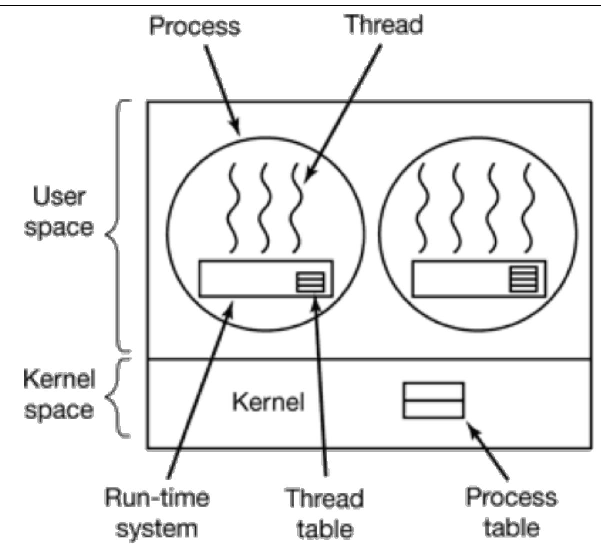


Figure 3.11: User threads.

The diagram shows that:

- all of the threads for a given process are present within the user space; and
- the thread table is present within the process.

User threads are:

- fast to create and manage; and
- portable to any operating system (OS).

If one user thread is blocked, all other user threads in the same process are also blocked. For example, in a word processor application, a thread that handles a printing event would block all other threads and therefore prevent the user from interacting with other aspects of the application.

Multi-threaded applications cannot take advantage of parallel execution on computer systems with a multi-core CPU or computer systems with multiple CPUs.

Kernel threads

Kernel threads are directly managed and supported by the operating system's (OS's) kernel.

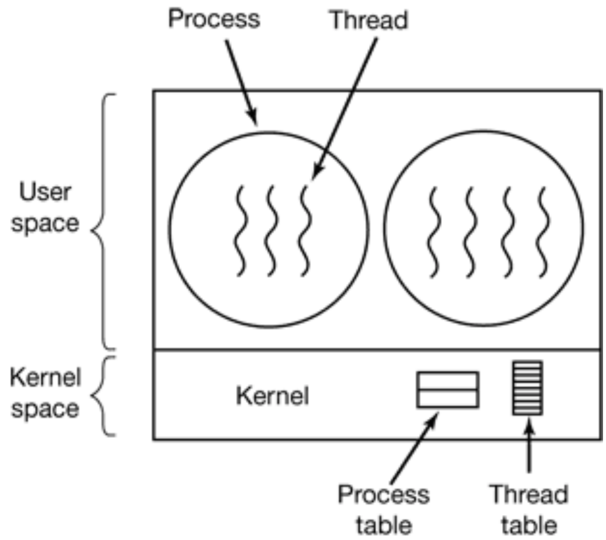


Figure 3.12: Kernel threads.

The diagram shows that:

- all of the threads for a given process are present within the user space; and
- the thread table is present within the kernel space, rather than the process itself or the user space.

Kernel threads are:

- slower to create and manage than user threads; and
- specific to the operating system (OS).

If one kernel thread is blocked, all other kernel threads are scheduled.

Can take advantage of parallel execution on computer systems with a multi-core CPU or computer systems with multiple CPUs.

3.7 Multi-threading models

Why is multi-threading mapping required?

The kernel is generally not aware of the user threads present in a process. Therefore, a thread library must map user threads to kernel threads.

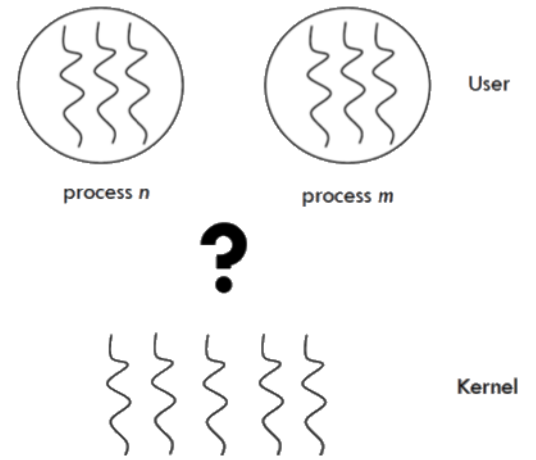


Figure 3.13: Multi-threading mapping.

The diagram shows that there must be some relationship between the user threads and the kernel threads. This relationship may defined using different mappings, including:

- many-to-one;
- one-to-one; and
- many-to-many.

Many-to-one mapping

How it works

All user threads from each process map to one kernel thread.

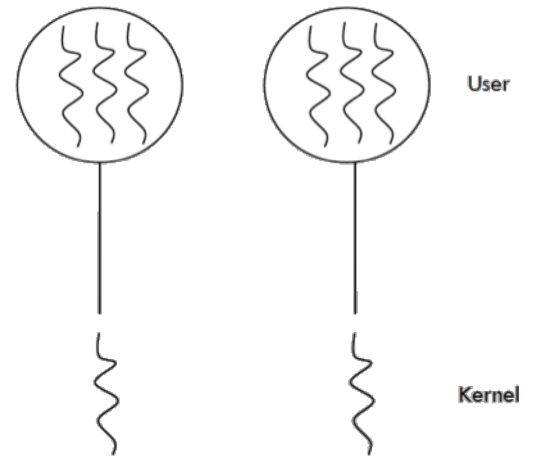


Figure 3.14: Many-to-one mapping.

Evaluation

Advantages	Disadvantages
Portable as there are few system dependencies.	**No parallel execution of threads**.
	No concurrency as all threads in a process are blocked if another thread is blocked, for example if the thread is waiting for an input/output (I/O) interrupt.

One-to-one mapping

How it works

Each user thread maps to a single kernel thread.

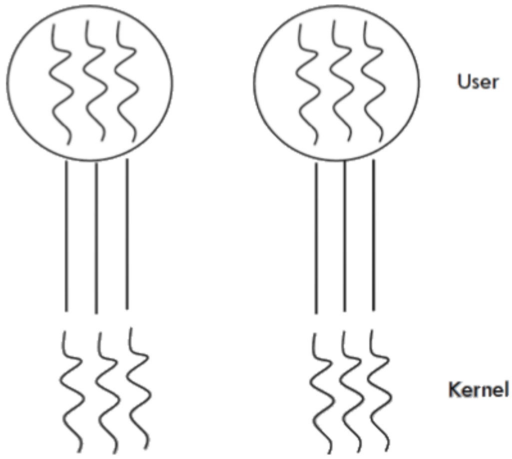


Figure 3.15: One-to-one mapping.

Evaluation

Advantages	Disadvantages
Concurrency as all threads in a process are not blocked if any given thread becomes blocked.	**Slow** as there is management overhead because the kernel is involved for every user thread.
Performance as it can take advantage of multiple CPUs.	**Restricted** as there is typically a limit on the number of threads.
	Creating user threads requires the corresponding kernel support.

Many-to-many mapping

How it works

Many user threads multiplex to an equal or smaller number of kernel threads.

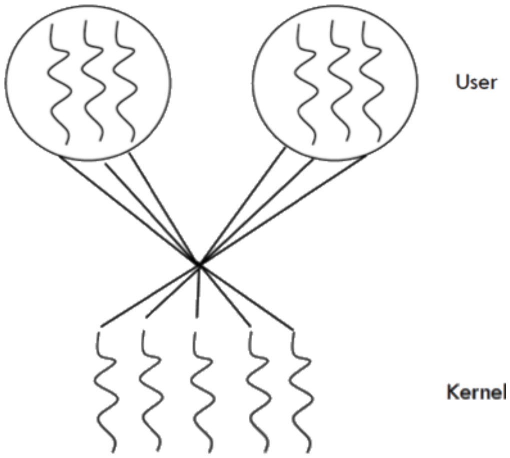
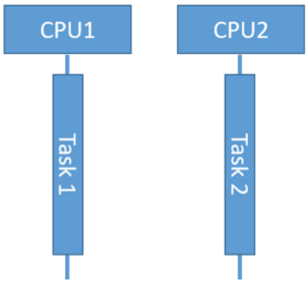
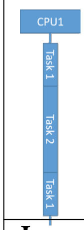


Figure 3.16: Many-to-many mapping.

Advantages	Disadvantages
Performance as it can take advantage of multiple CPUs.	**Complexity** and therefore implementation difficulties.
Flexible as there is no limit on the number of threads.	

3.8 Evaluation of concurrent programming

Advantages	Disadvantages
 **Parallelism.** It improves the efficiency of program execution in computer systems with multiple CPUs by allowing tasks/operations to be split up and executed independently on each CPU.	**Debugging complexity** as concurrent programs are non-deterministic and therefore it can be difficult to trace a problem/bug in the code as the same input will generally not result in the same output.
 **Multi-tasking.** It improves the utilisation of the CPU in a computer system that only has a single CPU. This allows multiple tasks/operations to run alongside each other and appear to be processed simultaneously.	**No protection between threads**.
Increases application responsiveness, for example, in a word processor application one thread could be responsible for responding to user input/output (I/O) while other threads perform tasks in the background.	**Concurrent processes must interact with each other** in order to share resources or exchange data.
Suited to some applications as there are some practical applications that are non-deterministic and concurrent as the order of program operations is determined by other external events. This is useful for applications that need to handle multiple events.	**Synchronisation must be promoted** in order to determine when, how and with what language abstractions computation events can be synchronised in order to eliminate unacceptable outputs.
	Distribution must be taken care of in order to consider how threads can be distributed among a number of CPUs and how one thread is able to interact with another thread on a different CPU.

	Error-prone. Examples of major concurrent programming errors include: • Therac-25 - A computerised radiation therapy machine whose errors contributed to accidents causing deaths and serious injuries. • Mars Rover Pathfinder – Problems with interaction between concurrent tasks caused periodic software resets, thus reducing availability for exploration.

4. Synchronisation and Mutual Exclusion

4.1 Race condition

Definition

A **race condition** describes the competition for resources in a critical section caused by interleaving/thread interference.

Why do race conditions happen?

Race conditions occur due to interleaving/thread interference.

Interleaving/thread interference describes an undesired outcome resulting from non-deterministic, concurrent usage of shared resources.

This happens because, in general, concurrent execution is non-deterministic, and therefore the same input generally means different output due to ordering. This is because there are many cases where the order of operation does matter.

Examples

Racing for memory access

A race condition may occur when two threads attempt to access the same location memory, such as registers or RAM, at the same time.

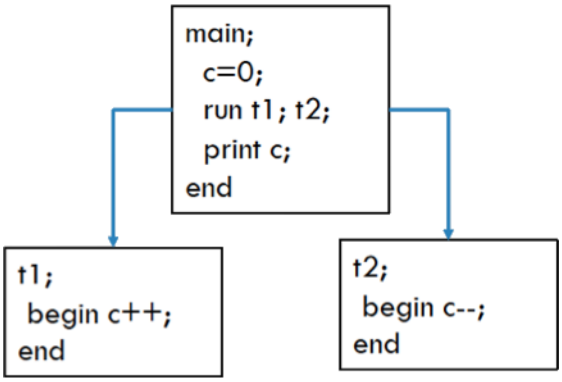


Figure 4.1: Concurrent program spawning threads t_1 and t_2.

In this example, the two threads t_1 and t_2 manipulate the same variable where:

- t_1 increments the variable c; and
- t_2 decrements the variable c.

As seen before (page 40), each statement may be compiled in to several machine instructions.

The increment ($c++$) instruction is broken in to three machine instructions:

- retrieve c;
- increment retrieved value; and
- store result in c.

The decrement ($c--$) instruction is also broken in to three machine instructions:

- retrieve c;
- decrement retrieved value; and
- store result in c.

As a result, one interleaving possibility is as follows:

- t_1: retrieve c;
- t_2: retrieve c;
- t_1: increment retrieved value; (result is 1)
- t_2: decrement retrieved value; (result is -1)
- t_1: store result in c; (c is now 1)
- t_2: store result in c; (c is now -1)

This example shows that the race condition has caused the result from thread t_1 to be lost as it has been overwritten by the result from thread t_2.

Racing for peripheral access

A race condition may also occur when two threads attempt to access the same peripheral, such as a printer spooler directory, at the same time.

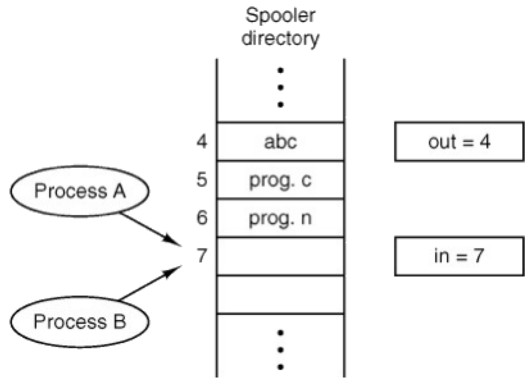


Figure 4.2: Printer spooler directory.

In this example, the two processes A and B attempt to access the printer spooler directory at the same time:

- the next available printer job slot is 7;
- process A and B access printer job slot 7 simultaneously;
- process A reads printer job slot 7 and a timer interrupt occurs that causes a context switch to process B before process A has opportunity to store any data;
- process B reads printer job slot 7 and stores its job data and increments the values; and
- another timer interrupt occurs that causes a context switch to process A that then stores its job at printer job slot 7.

This example shows that the race condition has caused the job data stored in the printer spooler directory by process B to be lost as it has been overwritten by the job data from process A.

4.2 Inter-process synchronisation

Definition

Inter-process synchronisation involves techniques that are designed to prevent race conditions and allows threads/processes to share resources.

Behaviour of threads

Threads in a computer system may behave in two possible ways:

- competing – two or more processes compete for the same computing resource, for example access to a particular memory cell; or
- cooperating – two or more processes may need to communicate with one another, thus causing information to be passed from one to the other.

Inter-process synchronisation is required to manage both threads that are competing and threads that are cooperating.

An operating system (OS) itself contains both threads that are competing and threads that are cooperating.

How mutual exclusion works

The solution to preventing race conditions is by implementing mutual exclusion on any given critical section/region.

The **critical section/region** is code in a process that involves sensitive operations on a shared resource.

Mutual exclusion is the requirement that one thread of execution never enters its critical section/region at the same time that another concurrent thread of execution enters its own critical section/region.

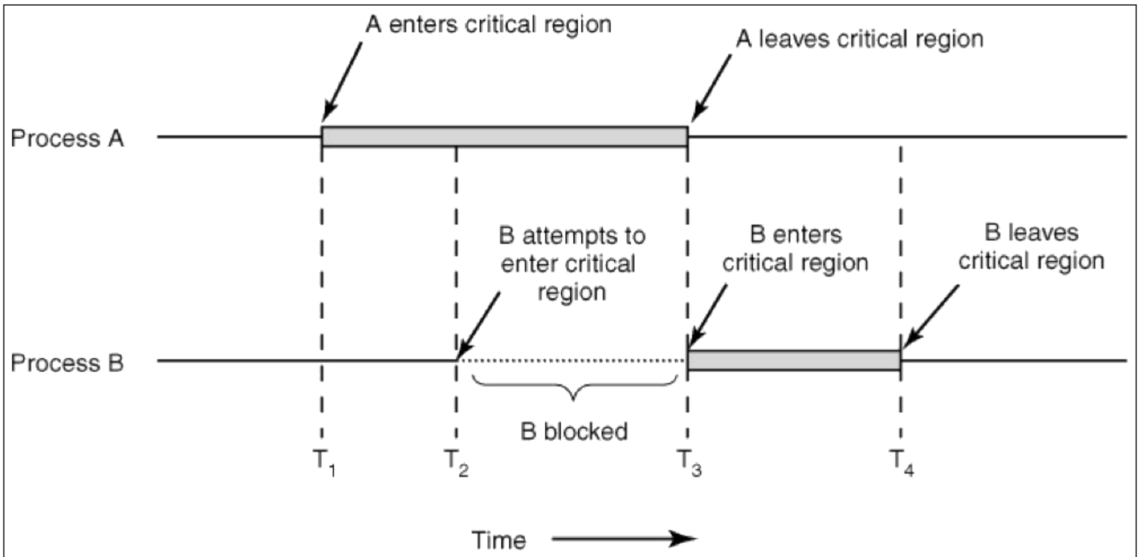


Figure 4.3: Mutual exclusion on critical section/region.

When a thread/process enters its critical section/region, no other thread/process may also enter its critical section/region.

This is demonstrated in the diagram above:

- at time interval T_1, process A enters its critical section/region;
- at time interval T_2, process B attempts to enter its critical section/region;
- process B is blocked from entering its critical section/region until process A leaves its critical section/region;
- at time interval T_3, process A leaves its critical section/region and therefore process B is allowed to enter its critical section/region; and
- at time interval T_4, process B leaves its critical section/region.

This shown that mutual exclusion is enforced as no two threads/processes are simultaneously inside their critical sections/regions.

For mutual exclusion to be effective:

- no assumptions may be made about the speeds or the number of threads/processes;
- no threads/processes running outside its critical section/region may block other threads/processes, such that a thread/process that is not in its critical section/region cannot prevent other threads from entering their critical section/region; and
- no thread/process should have to wait forever to enter its critical section/region.

Mutual exclusion is a major design issue in operating systems (OSs) as consideration must be taken in order to prevent race conditions while maintaining parallelism and efficiency.

How mutual exclusion is implemented

Mutual exclusion is implemented using semaphores.

A **semaphore** is a system tool used for the design of correct synchronisation protocols. This was introduced by Edsger Dijkstra in the 1962/1963. Semaphores are implemented using a variable or abstract data type and are used to control thread/process access to a resource. They are typically integer values that accept only non-negative values.

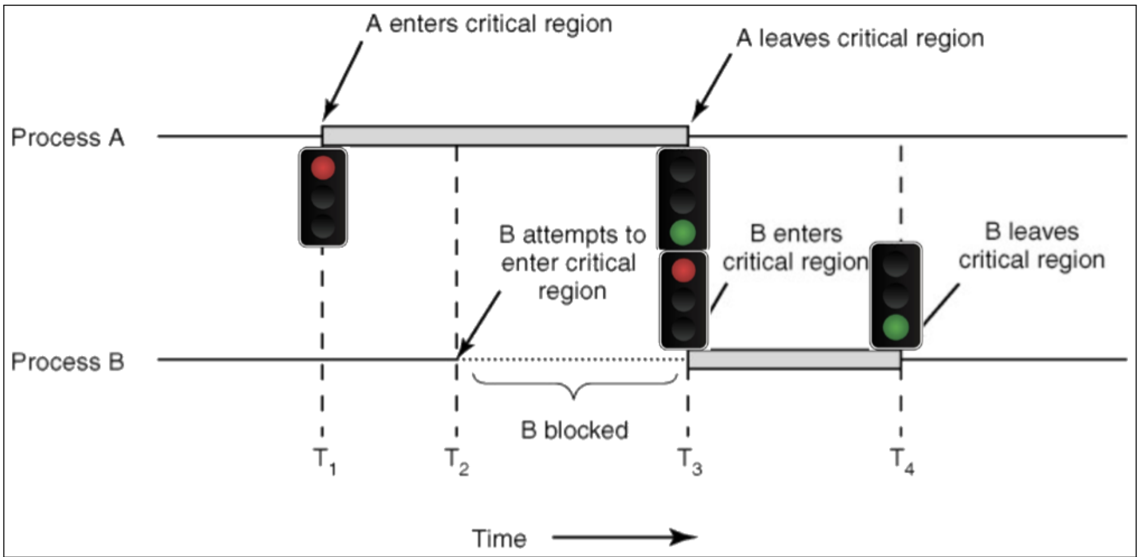


Figure 4.4: Semaphore interaction on critical section/region.

The diagram shows that semaphores allow the CPU to context switch between threads/processes when one becomes blocked.

It is convenient to write entry and exit protocols using a single atomic statement. This statement is atomic and therefore is indivisible, meaning that the statement cannot be interrupted.

As mentioned before, a semaphore, denoted by S, is an integer that takes only non-negative values. Only two atomic (indivisible) statements are permitted, as shown below.

Statement	Statement Implementation	Usage

wait(*s*)	wait(s)
	{
	if (S > 0)
	{
	S--;
	}
	}
		If a thread/process is in its non-critical section/region and wishes to enter its critical section/region, this statement will be performed.
		This means that the thread/process will be blocked until $S = 0$ evaluates to $True$.

signal(*s*)	signal(s)
	{
	S++;
	}
		If a thread/process is in its critical section/region, this statement will be performed. This helps to achieve mutual exclusion as it prevents $S = 0$ from evaluating to $True$ until the thread/process has left its critical section/region.

This is a good solution as there is no possibility for a race condition as these statements will always be enforced due to the fact that they are atomic (indivisible) statements and cannot be interrupted.

4.3 The producer-consumer problem

Problem description

The producer-consumer problem is a classical inter-process communication problem in which:

- a producer repeatedly produces items and places them in to a buffer; and
- a consumer consumes the items one-by-one by taking them from the buffer.

This problem has the following requirements:

- the buffer must be assumed to be first in, first out (FIFO);
- the producer may produce a new item only at a time when the buffer is not full;
- the consumer may consume an item only at a time when the buffer is not empty; and
- the process terminates when all items produced are eventually consumed.

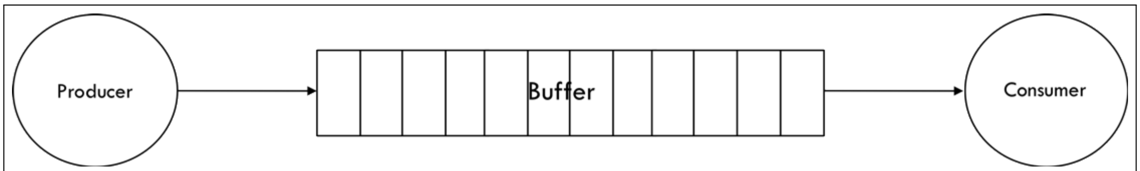


Figure 4.5: The producer-consumer problem.

The problem arises when attempting to devise a method that is able to:

- put the producer to "sleep" when the buffer is full to prevent further items being produced when there is no space in the buffer; and
- "wake" the consumer when the buffer is not empty as there is possibility to consumer when the buffer is not empty.

Possible solution

This problem could be solved by keeping track of the number of items in the buffer.

This could be achieved by implementing loops in the producer class and consumer class.

```
LOOP
{
   Produce item i          //produce item
   if ( itemCount == N )   //end of buffer
   {
      sleep(producer);
   }

   Put item i;             // place item in to buffer
   itemCount++;            // increment buffer count
   if ( itemCount == 1 )   // buffer nearly empty
   {
      wakeup(consumer);
   }
}
```

Producer class.

```
LOOP
{
   if ( itemCount == 0 )     // buffer empty
   {
      sleep(consumer);
   }

   Remove item j;            // remove item from buffer
   itemCount--;              // decrement buffer count
   if ( itemCount == N-1 )   // buffer has space
   {
      wakeup(producer);
   }
   Consume item j;           // consume item
}
```

Consumer class.

The loop in the producer class would be running as one thread and the loop in the consumer thread would be running as another thread. These two threads would be running in parallel. As a result, if the threads in the solution are interleaved, a race condition may occur, which in turn, may cause a deadlock.

Deadlocks

A **deadlock** occurs when two or more threads wait for each other to finish.

Four conditions must be hold simultaneously in order for a deadlock to occur:

- mutual exclusion – a resource can be assigned to, at most, one process at a time;
- hold and wait – processes holding resources are permitted to request and wait for additional resources;
- no pre-emption – resources previously locked cannot be forcefully unlocked by another process, instead they must be released by the holding process; and
- circular wait – there must be a chain of processes, such that each member of the chain is waiting for a resource held by the next member of the chain, as shown in the diagram below.

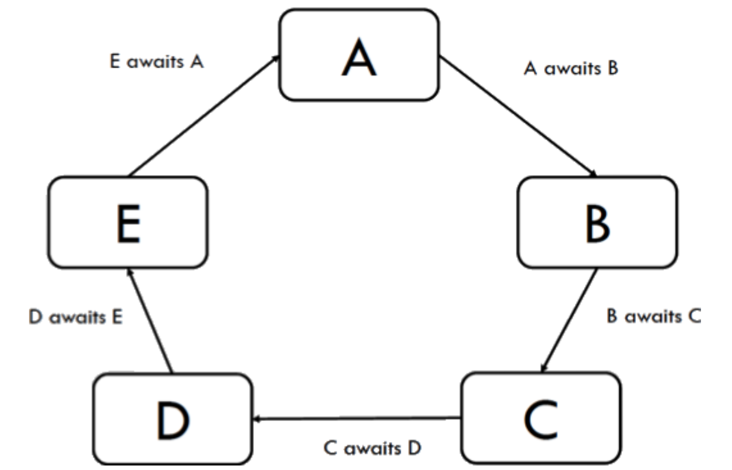


Figure 4.6: Circular wait.

A deadlock may occur in the possible solution described previously.

Consumer reads $itemCount = 0$ and it evaluates to $True$, and therefore $sleep(consumer)$ needs to be called.

↓

Just before $sleep(consumer)$ is called, the consumer is interrupted by a timer interrupt and the producer is resumed.

↓

The producer places an item in to the buffer, such that $itemCount = 1$.

↓

The producer tries to perform $wakeup(consumer)$ however, the consumer is already in "wakeup" mode. As a result, the call to $wakeup(consumer)$ is missed.

↓

When the consumer resumes, it will call $sleep(consumer)$ and get trapped in "sleep" mode.

↓

The producer will continue placing items in the buffer and call $sleep(producer)$ when the buffer is full.

↓

There is now a deadlock as both threads are waiting for a $wakeup$ call from each other.

Figure 4.7: Possible deadlock.

This possible deadlock shows that another solution is required to effectively solve the producer-consumer problem.

Solving the problem using semaphores

It is assumed that:

- `ItemsReady = 0`; and
- `SpacesLeft = N (size of buffer)`.

```
LOOP
{
   Produce item i       // produce item
   Wait(SpacesLeft)     // decrement semaphore

   Put item i;          // place item in to buffer
   Signal(ItemsReady)   // increment
}
```

Producer class.

```
LOOP
{
   Wait(ItemsReady)    // decrement semaphore

   Get item j;         // remove item from buffer
   Signal(SpacesLeft)  // increment semaphore
   Consume item i;     // consume item
}
```

Consumer class.

If this solution uses semaphores correctly, then

$$N = \text{SpacesLeft} + \text{ItemsReady}$$

as the producer will always be placing items in to the buffer when there are spaces available in the buffer.

However, this solution does not consider situations in which there are multiple producers and/or multiple consumers.

The multiple producer-consumer problem

Problem description

The multiple producer-consumer problem is a classical inter-process communication problem in which:

- multiple producer repeatedly produces items and places them in to a buffer; and
- multiple consumer consumes the items one-by-one by taking them from the buffer.

As with the previous producer-consumer problem, this problem has the following requirements:

- the buffer must be assumed to be first in, first out (FIFO);
- the producers may produce a new item only at a time when the buffer is not full;
- the consumers may consume an item only at a time when the buffer is not empty; and
- the process terminates when all items produced are eventually consumed.

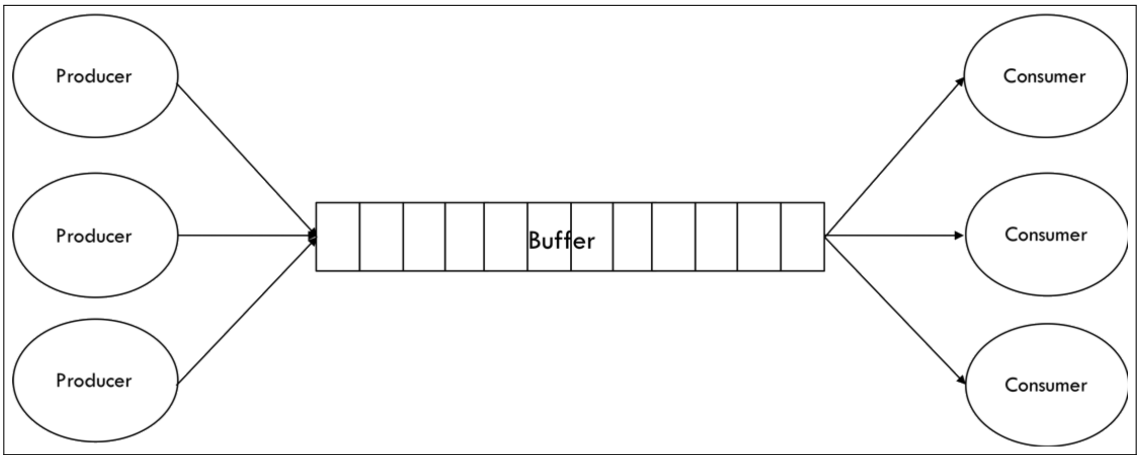


Figure 4.8: The multiple producer-consumer problem.

The problem arises when attempting to devise a method that is able to manage:

- two producers placing items in to the same slot in the buffer; and
- two consumers removing items from the same slot in the buffer.

This is similar to the problem previously discussed in the printer spooler example.

A race condition may also occur when producers attempt to access a variable at the same time.

To demonstrate the race condition, it is necessary to consider the following possible interleaving of the threads/processes:

- two producers access the $SpacesLeft$ variable at the same time, which corresponds to decrementing the semaphore;
- both producers get the same next empty slot in the buffer at the same time; and
- both producers write in to the same slot.

This example shows that the race condition has caused the data stored in the buffer slot by the first producer to be lost as it has been overwritten by the data stored in the buffer slot by the second producer.

In order to ensure mutual exclusion when multiple users are involved, an additional semaphore must be introduced.

Mutex

A **mutex** (or **binary semaphore**) is a semaphore with ownership that can only be released by its owner and is initially set to 1.

Problem solution

It is now possible to construct a solution, using a mutex (or binary semaphore), that will ensure mutual exclusion even when there are multiple producers and/or multiple consumers.

It is assumed that:

- `ItemsReady = 0`; and
- `SpacesLeft = N (size of buffer)`.

```
LOOP
{
   Produce item i      // produce item
   Wait(SpacesLeft)    // decrement semaphore
   Wait(BusyBuffer)    // mutex

   Put item i;         // place item in to buffer
   Signal(BusyBuffer)  // release mutex
   Signal(ItemsReady)  // increment
}
```

Producer class.

```
LOOP
{
   Wait(ItemsReady)    // decrement semaphore
   Wait(BusyBuffer)    // mutex

   Get item j;         // remove item from buffer
   Signal(SpacesLeft)  // increment semaphore
   Signal(BusyBuffer)  // release mutex
   Consume item i;     // consume item
}
```

Consumer class.

The `BusyBuffer` mutex has ownership and therefore can only be incremented/decremented by the same thread/process. The `BusyBuffer` mutex is initially set to one (1).

The order in which semaphores are incremented and decremented is essential. This can be demonstrated by inspecting the effect of switching around two statements in the Consumer class:

```
Wait(ItemsReady)  // decrement semaphore
Wait(BusyBuffer)  // mutex
...
Wait(BusyBuffer)  // mutex
Wait(ItemsReady)  // decrement semaphore
```

Switching these statements may cause a deadlock as calling the `Wait` function on the `BusyBuffer` mutex would prevent any other threads/processes from calling the `Signal` function on the `ItemsReady` semaphore. This would result in a circular wait, as demonstrated in the dining philosophers problem below.

4.4 The dining philosophers problem

Problem description

The dining philosophers problem is a classical inter-process communication problem in which:

- five philosophers are seated around a circular table eating and thinking; and
- each philosopher has a plate of spaghetti that they can eat with forks.

This problem has the following requirements:

- the life of a philosopher consists of only alternating periods of eating and thinking;
- between each pair of plates is one fork;
- each philosopher needs two forks to eat the spaghetti; and
- no two philosophers may hold the same fork simultaneously.

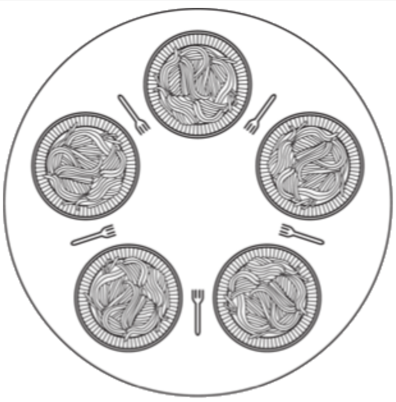


Figure 4.9: Layout of table.

The problem arises when attempting to devise a method that is able to:

- allow each philosopher to have alternating periods of eating and thinking; and
- not result in a deadlock.

It could be said that the problem requirement for each philosopher to need two forks to eat the spaghetti is somewhat artificial. As a result, we can substitute the spaghetti for rice and substitute chopsticks for forks.

The problem now has the following updated requirements:

- the life of a philosopher consists of only alternating periods of eating and thinking;
- between each pair of plates is one chopstick;
- each philosopher needs two chopsticks to eat the rice; and
- no two philosophers may hold the same chopstick simultaneously.

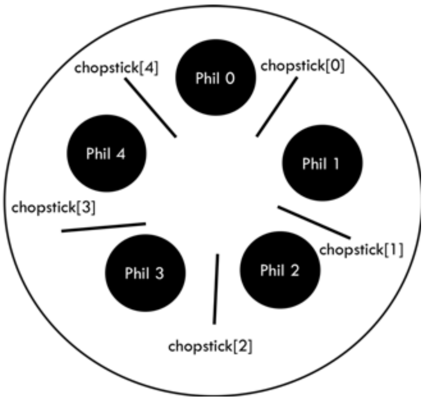


Figure 4.10: Updated layout of table.

Problem solutions

Problem solution 1

This problem can be solved using semaphores, using the following assumptions:

- each philosopher is a different thread with a unique ID;
- one semaphore is implemented per chopstick; and
- chopsticks are identified by using the unique ID of a philosopher.

As chopsticks are identified by using the unique ID of a philosopher, it could be that:

- the chopstick to the left of a given philosopher is $chopstick[i]$; and
- the chopstick to the right of a given philosopher is $chopstick[((i-1)+N)\%N]$.

where i is the ID of the philosopher and N is the number of philosophers.

The identification of the chopsticks works as demonstrated by using Philosopher 1 as an example.

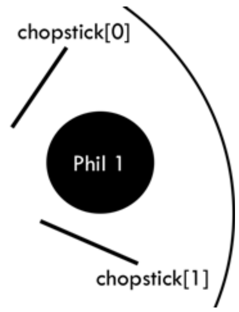


Figure 4.11: Philosopher 1.

The diagram shows that:

- the chopstick to the left of Philosopher 1 is $chopstick[1]$; and
- the chopstick to the right of Philosopher 1 is $chopstick[0]$.

Using this example, it is possible to validate the chopstick identification formulas discussed before.

Given that $i = 1$ for Philosopher 1 and that $N = 5$ as there are five philosophers in total,

$$\begin{aligned}
\text{left} &= \text{chopstick[i]} \\
\text{left} &= \text{chopstick[1]} \\
\text{right} &= \text{chopstick[((i-1) + N) \% N]} \\
\text{right} &= \text{chopstick[((1-1) + 5) \% 5]} \\
\text{right} &= \text{chopstick[(0 + 5) \% 5]} \\
\text{right} &= \text{chopstick[5 \% 5]} \\
\text{right} &= \text{chopstick[0]}
\end{aligned}$$

This shows that the chopstick identification formulas work for Philosopher 1.

```
LOOP
{
   think();
   wait(chopstick[left]);     // take left chopstick
   wait(chopstick[right]);    // take right chopstick
   eat();
   signal(chopstick[left]);   // release left chopstick
   signal(chopstick[right]);  // release right chopstick
}
```

Philosopher class.

However, this solution has the possibility to cause a race condition.

This can be demonstrated in the situation where all five philosophers wish to eat at the same time and therefore all take their left chopsticks:

- Philosopher 0 takes $chopstick[left]$;
- Philosopher 1 takes $chopstick[left]$;
- Philosopher 2 takes $chopstick[left]$;
- Philosopher 3 takes $chopstick[left]$; and

- Philosopher 4 takes $chopstick[left]$.

This causes a race condition as:

- each philosopher will now be waiting to take $chopstick[right]$;
- no philosopher can take $chopstick[right]$ as this chopstick is the subsequent philosopher's $chopstick[left]$ and this has already been taken; and
- no philosopher can perform their *eat* function and therefore no chopsticks will be released as the *signal* functions are only performed after the *eat* function has completed.

This shows that, if the threads in the solution are interleaved, a race condition may occur. In this case, a circular wait is caused and therefore there is a deadlock.

Possible solution 2

As shown in the multiple producer-consumer problem, a mutex (or binary semaphore) can be introduced to prevent this deadlock.

```
LOOP
{
   think();
   wait(busy);                 // mutex
   wait(chopstick[left]);      // take left chopstick
   wait(chopstick[right]);     // take right chopstick
   eat();
   signal(chopstick[left]);    // release left chopstick
   signal(chopstick[right]);   // release right chopstick
   signal(busy);               // release mutex
}
```

Philosopher class.

This solves the deadlock as the mutex (or binary semaphore) signals the critical section/region and prevents other philosophers from attempting to take a chopstick that is currently in use by another philosopher.

Although this solution is deadlock-free, it has a performance bug. The mutex means that only one philosopher can be eating at any instant and, with five chopsticks available, it should be possible to allow two philosophers to eat at the same time. As a result, there are more efficient solutions to this problem that achieve maximum parallelism.

Final revision solution

This solution is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers.

```
#define N          5               // number of philosophers
#define LEFT       (i + N - 1) % N // ID of i's left neighbour
#define RIGHT      (i + 1) % N     // ID of i's right neighbour
#define THINKING   0               // philosopher is thinking
#define HUNGRY     1               // philosopher is trying to acquire
                                   // chopsticks
```

```
#define EATING    2                // philosopher is eating

typedef int semaphore              // semaphores are a special kind of
                                   // integer
int state[N];                      // array to keep track of philosopher'
   s
                                   // state
semaphore mutex = 1;               // mutual exclusion for critical
   regions
semaphore s[N];                    // one semaphore per philosopher

// i: philosopher unique ID, from 0 to N-1
void philosopher (int i)
{
   while (TRUE)                    // repeat indefinitely
   {
      think();                     // philosopher is thinking
      take_chopsticks(i);          // acquire two chopsticks or be blocked
      eat();                       // philosopher is eating
      put_chopsticks(i);           // place both chopsticks back on the
   table
   }
}

// i: philosopher unique ID, from 0 to N-1
void take_chopsticks(int i)
{
   wait(&mutex);                   // enter critical section/region
   state[i] = HUNGRY;              // record fact that philosopher i is
   hungry
   test(i);                        // attempt to acquire two chopsticks
   signal(&mutex);                 // exit critical section/region
   wait(&s[i]);                    // block if chopsticks were not
   acquired
}

// i: philosopher unique ID, from 0 to N-1
void put_chopsticks(int i)
{
   wait(&mutex);                   // enter critical section/region
   state[i] = THINKING;            // record fact that philosopher i is
                                   // thinking
   test(LEFT);                     // check if left neighbour can now eat
   test(RIGHT);                    // check if right neighbour can now eat
   signal(&mutex);                 // exit critical section/region
}

// i: philosopher unique ID, from 0 to N-1
void test(int i)
{
   if (state[i] == HUNGRY && state[LEFT] != EATING
```

```
        && state[RIGHT] != EATING)
    {
        state[i] = EATING;          // record fact that philosopher i is
    hungry
        signal(&s[i]);              // exit critical section/region
    }
}
```

Philosopher class.

This solution uses the array state to keep track of whether a philosopher is:

- eating;
- thinking; or
- hungry (trying to acquire chopsticks).

This represents an array of semaphores and each philosopher has its own *state* array. This enables hungry philosophers to be blocked if the required chopsticks are currently busy.

A philosopher may move in to eating state only if neither neighbouring philosopher is eating. For example, Philosopher 1 cannot enter eating state if:

- Philosopher 0 is currently in eating state; or
- Philosopher 2 is currently in eating state.

5. Memory Management

5.1 Memory hierarchy

Definition

The **memory hierarchy** separates computer storage into a hierarchy based on response time. A computer system is usually composed by a layered memory system

Diagram

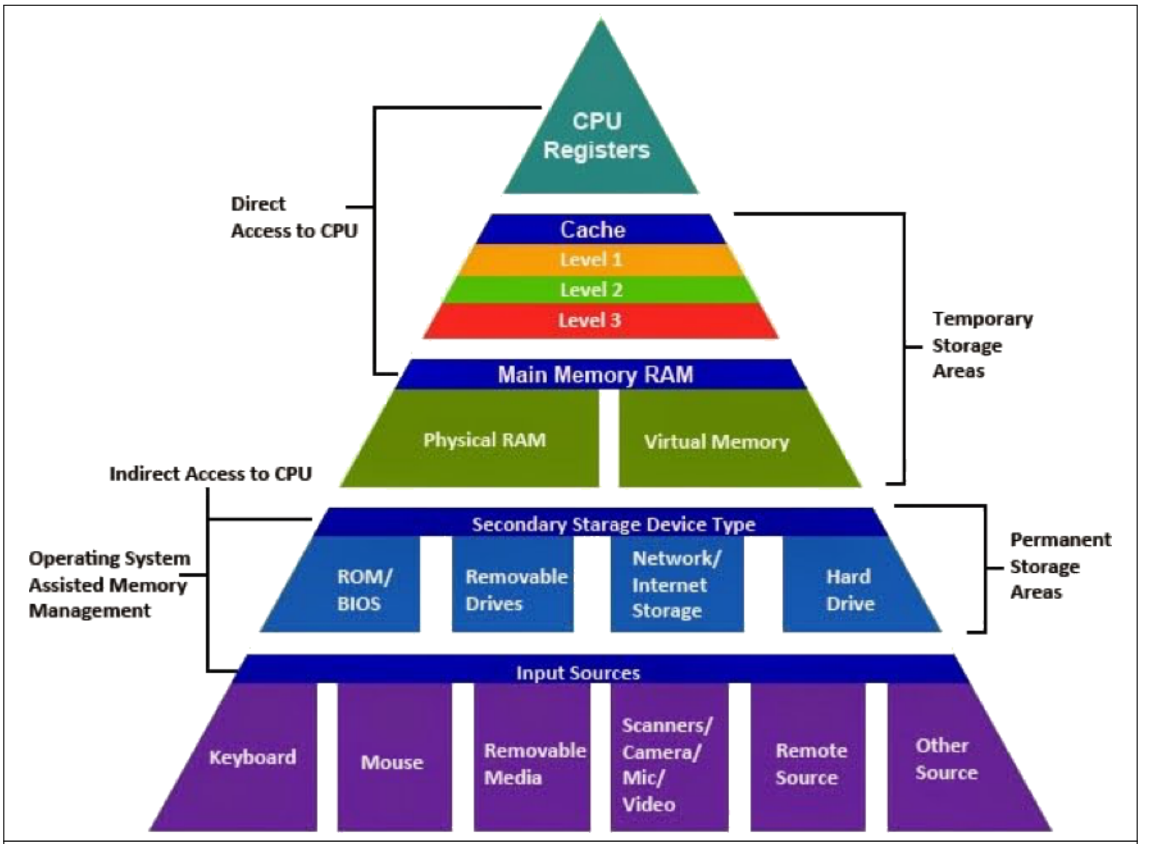


Figure 5.1: Memory hierarchy pyramid.

Higher layers correspond to faster devices that:

- have lower capacity;
- require more power and generate more heat; and
- are more expensive.

Lower layers correspond to slower devices that:

- have higher capacity;
- require less power and generate less heat; and
- are cheaper.

The temporary storage areas can be described as volatile; the data does not persist after power-down.

The permanent storage areas can be described as non-volatile; the data persists after power-down.

5.2 Primary memory

Definition

Primary memory is volatile, meaning that the data contained within does not persist after power-down, and stores data and instructions for use by the CPU in currently running processes.

Registers

Size	Very small.
Speed	Extremely fast.
Purpose	Individual containers for single values. Almost all computers load data from a memory lower in the hierarchy into registers, where it is used for arithmetic operations and is manipulated or tested by machine instructions.
Location	Inside the CPU.

Registers are typically referred to by "name" (i.e. individual identifying code), not by address, as happens with other types of memory.

Figure 5.2: Registers.

There are two main types of registers:

- generic registers – allow the CPU to temporarily store data on which it will perform operations, and consequently data that results from those operations; and
- specialised registers – manipulated in mostly the same fashion as generic registers but are used for specialised purposes.

Specialised registers include:

- Instruction Register (IR) – holds the instruction currently being executed;
- Memory Data Register (MDR) – holds the piece of data that has been fetched from memory;

- Memory Address Register (MAR) – holds the address of the next piece of memory to be fetched;
- Program Counter (PC) – holds the location of the next instruction to be fetched from memory and is automatically incremented between supplying the address of the next instruction and the instruction being executed; and
- Accumulator (ACCU) – used as the default location to store any calculations performed by the arithmetic and logic unit.

Generic registers are available to store any transient data required by the program.

For example, when a program is interrupted its state, the values stored in the specialised registers may be saved into the generic registers, ready for recall when the program is ready to start again.

In general, the more registers a CPU has available, the faster it can work.

Cache memory

Size	Small (KB/MB).
Speed	Fast.
Purpose	Serves as an intermediary between main memory and the registers. When data or instructions are fetched from main memory, they are copied to the cache for further use, and therefore, reduce the average cost (time or energy) to access data from the main memory.
Location	Inside the CPU.

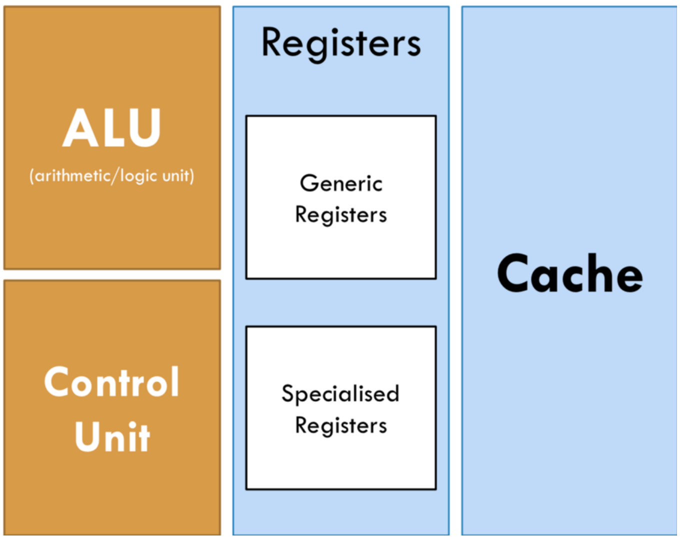


Figure 5.3: Cache memory.

Modern computers typically have a sub-hierarchy of cache memories (L1, L2, L3, L4, etc.), for example:

- level 1 cache (L1) – very fast and small size (2KB - 64KB); and
- level 2 cache (L2) – fast and medium size (256KB – 2MB).

Main memory

Size	Medium (MB/GB).
Speed	Fast.
Purpose	Stores data and machine code currently being used.
Location	Motherboard, connected to the CPU by a bus.

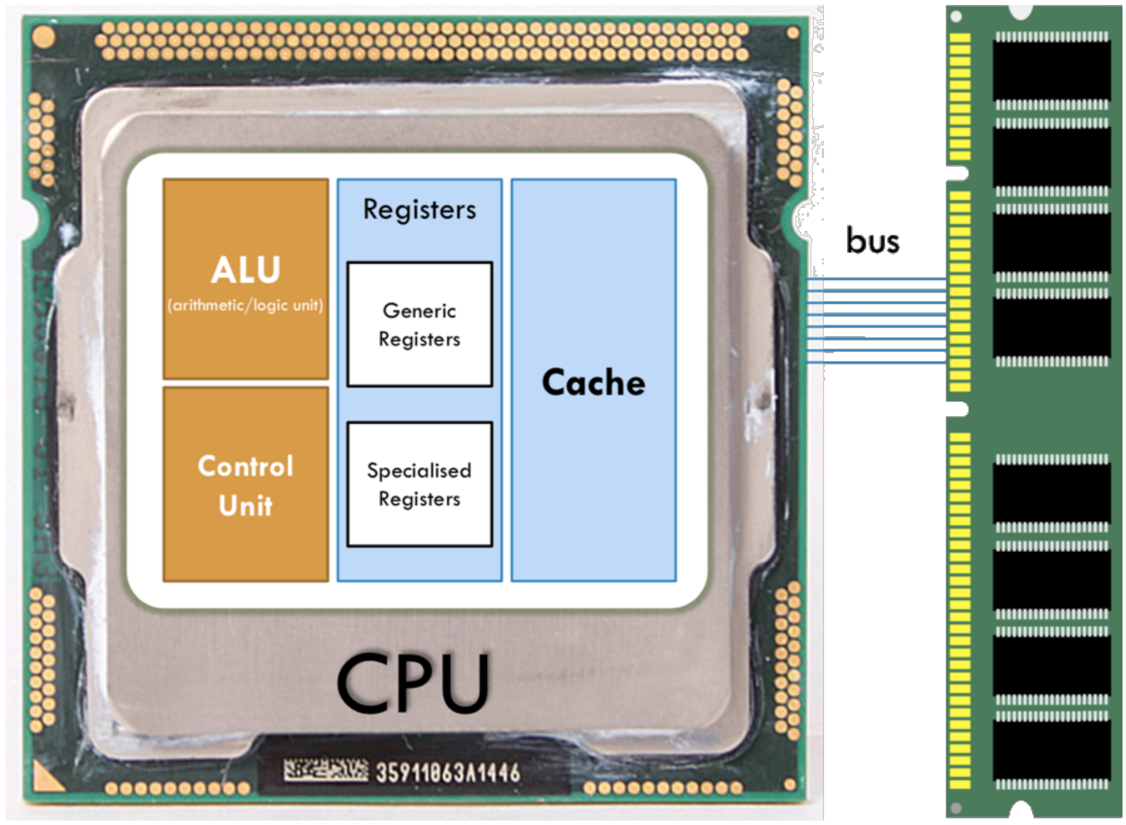


Figure 5.4: Main memory.

Main memory is typically a random-access memory (RAM) device. These devices allow data items to be read from or written to in almost the same amount of time, irrespective of the physical location of the data.

5.3 Secondary memory

Definition

Secondary memory is a non-volatile and persistent store of data, which is used for items such as user files, programs and the operating system.

Magnetic – Hard-Disk Drive (HDD)

Description

A **Hard-Disk Drive (HDD)** contains rotating platters which are coated with magnetic material. This contains ferrous particles, containing iron, which are polarised to become either a north or south state; these states can correspond to either a 0 or a 1.

The platters spin at speeds of up to 10,000 revolutions per minute (RPM) and a drive head moves across the platters to access different tracks and sectors. Data is read from or written to the platters as it passes under the drive head. When no operations are being performed, the drive head is parked to the side of the platters to prevent damage from movement.

There may be several platters and several drive heads.

Size	xxx'GB and x'TB.
Speed	Medium.
Purpose	Consumer desktop computer system secondary storage.
Reliability	Prone to drop damage as it has moving parts and magnetic fields.

Evaluation

Advantages	**Disadvantages**
Large storage capacity.	**Relatively slow read and write speeds** because the disk head must physically move to the location of the data on the platter.
Affordable large capacities, HDDs are available in large storage capacities at relatively low prices.	**Can be loud** because there are moving parts, such as the platters and disk head, which generate noise.
	Low reliability because they are prone to drop damage, due to their moving parts, magnetic fields.
	Relatively large form factor since there must be room to enclose the platters and disk heads.

Magnetic – Magnetic tape

Description

A **magnetic tape** is a reel of tape which is coated with a magnetic material. This contains ferrous particles, containing iron, which are polarised to become either a north or south state; these states can represent either a 1 or a 0.

The data is accessed using serial access; this is where a tape reader starts at the beginning of the tape and "fast forwards" to the portion of the tape containing the required data.

Size	xxx'GB and x'TB.
Speed	Slow.
Purpose	Server backup storage.
Reliability	Prone to magnetic fields.

Evaluation

Advantages	Disadvantages
Large storage capacity.	**Relatively slow read and write speeds** because it uses serial data access.
Affordable large capacities, magnetic tapes are available in high capacities at relatively low prices	**Low reliability** because they are prone to magnetic fields.
	Requires specialist equipment to read the data on the tape, known as a tape reader.

Flash – Solid-State Drive (SSD)

Description

A **Solid-State Drive (SSD)** contains millions of NAND flash memory cells and a controller which manages pages and blocks of memory.

Each NAND flash memory cell delivers a current along the bit and word lines to activate the flow of electrons from the source towards the drain. The current on the word line is strong enough to push a few electrons across an insulated oxide layer into a floating gate. The charge in the flowing gate is measured: if there is some charge, where there are some electrons flowing, a 1 is represented; and if there is no charge, where there are no electrons flowing, a 0 is represented.

Size	xx'GB and xxx'GB.
Speed	Very fast.
Purpose	Portable device secondary storage, such as smartphones and laptops.
Reliability	Less prone to drop damage as there are no moving parts, limited write cycles before damage.

Flash drives and SD cards also use solid-state technologies. They are often used for portable storage; for example, flash drives may be used for transferring documents and SD cards may be used to storage images in a camera.

Evaluation

Advantages	Disadvantages
Fast read and write speeds because the data can be accessed directly on the drive using electrical currents.	**Expensive large capacities**, SSDs are more expensive than other storage options.
Available in different form factors, such as USB memory sticks and SD cards.	**Low durability** because they have a finite number of read and write operations which can be performed before the performance of the drive beings to downgrade due the discharge of its electrons.
High reliability because there are no moving parts.	
Silent operation because there are no moving parts.	

Optical – Optical disk

Description – Optical disk

An **optical disk** works by using:

- a high-powered laser to change the properties of its surface to make them less reflective, by a process called "burning"; and
- a low-powered laser to read the disk by shining light onto the surface and a sensor is used to measure the amount of light that is reflected back.

There are multiple types of optical disks, including:

- CD-ROM;
- CD-RW;
- DVD-RW; and
- Bluray.

Description – CD-ROM

A **CD-ROM (read only Compact Disk)** is pressed during is manufacture and has pits, where the surface has been depressed, and lands, where the surface has not been depressed.

When the low-powered laser is shone on the surface:

- the start or end of a pit, the light is scattered and not reflected very well — this is used to represent a 0; and

- a land, the light is not scattered and reflected normally -– this is used to represent a 1.

Size	xxx'MB.
Speed	Slow.
Purpose	Commercial music distribution.
Reliability	Prone to scratches and cracks.

Description – CD-RW

A **CD-RW (re-writable Compact Disk)** uses a laser and a magnet to heat a spot on the disk. When the spot is heated the magnetic orientation is changed to represent a 0 or a 1 before the magnet is cooled.

Size	xxx'MB.
Speed	Slow.
Purpose	Small file storage such as documents.
Reliability	Prone to scratches and cracks.

Description – DVD-RW

A **DVD-RW (re-writable Digital Versatile Disc)** uses a phase changed alloy that can change between amorphous and crystalline states by changing the power of the laser beam.

Size	x'GB.
Speed	Slow.
Purpose	Standard definition videos.
Reliability	Prone to scratches and cracks.

Description – Blu-Ray

A **Blu-Ray** disk has a higher capacity than a CD-ROM because the laser used has a shorter wavelength which creates smaller pits and lands and therefore, a greater number of pits and lands can be present on the same surface area.

Size	xx'GB.
Speed	Slow.
Purpose	High definition videos.
Reliability	Prone to scratches and cracks.

Evaluation

Advantages	**Disadvantages**
Relatively portable because they can be carried in a disk case.	**Relatively slow read and write speeds** because the disk must spin in the optical drive to read the data.

Immune to magnetic fields meaning that the close proximity of a magnet is not going to corrupt data.	**Low reliability** because they are prone to scratches and cracks.

5.4 Read-only memory (ROM)

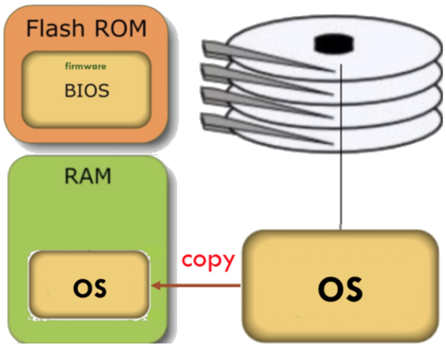
Properties

Size	Medium (MB/GB).
Speed	Fast.
Purpose	Used in the system boot process and, in some cases, stores the operating system in computer systems where the software is not updated.
Reliability	Motherboard.

Use in the system boot process

A **Basic Input/Output System (BIOS)** is a set of instructions which control the boot process of a computer system and the communication between the operating system and peripherals.

A **peripheral** is a piece of hardware outside a computer system that is not required for the computer system to operate.



The motherboard BIOS initialises registers and power management.

↓

The motherboard BIOS performs a self-diagnostic test, known as a Power-On Safe Test (POST).

↓

The motherboard BIOS boots the BIOS for all subsidiary boards and begins communications with basic peripherals.

↓

The motherboard BIOS determines which non-volatile storage device contains the operating system (OS).

↓

The motherboard BIOS initiates the bootstrapping sequence where the secondary bootloader, such as GRUB, LILO or NTLDR etc., is copied to RAM.

↓

The secondary bootloader waits for a few seconds to allow the user to make a choice on the operating system (OS) to be launched.

↓

The secondary bootloader loads the chosen operating system (OS) to RAM and relinquishes control.

Figure 5.5: System boot process.

5.5 Concept and aim of memory management

Concept

Processes share physical memory, as well as the CPU. Main memory (RAM) is one of the most important resources in a computer system as:

- data and instructions are fetched from RAM in to the CPU for execution; and
- memory can store several running processes at once.

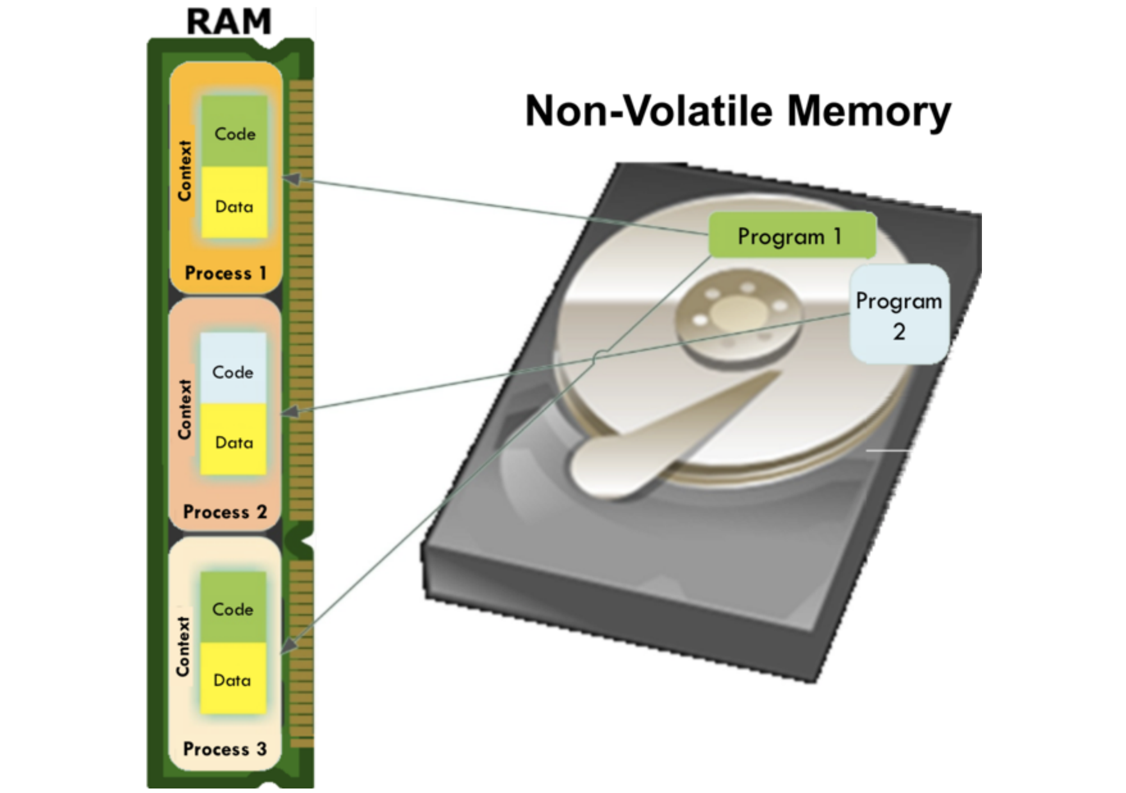


Figure 5.6: Main memory and non-volatile memory.

As seen in the diagram above, RAM can contain the data and instructions for multiple processes. These are loaded in to RAM from non-volatile memory, such as a hard-disk drive (HDD).

Aim

Aims of memory management include:

- providing the memory space for concurrent processes;
- taking advantage of the locality of reference, which is the tendency of a CPU to access the same set of memory locations repetitively over a short period of time, by estimating the data and/or instructions to be used next and transfer them to the most effective place in the memory hierarchy pyramid;
- protect processes from one another;

- relocated memory to new processes; and
- make the addressing of memory space transparent by providing memory abstraction.

5.6 Overview of memory abstraction

Definition

Memory abstraction provides an abstraction layer between the program execution and the memory that provides a different ”view” of a memory location depending on the execution context in which the memory access is made. This is achieved by creating an “abstract memory” to allow for co-existence in physical memory.

Computer system with no memory abstraction

Early mainframe computers had no memory abstraction and every program had access to physical memory.

The memory model presented to the programmer was a set of addresses from zero (0) to max, in which each address corresponds to a cell containing some number of bits.

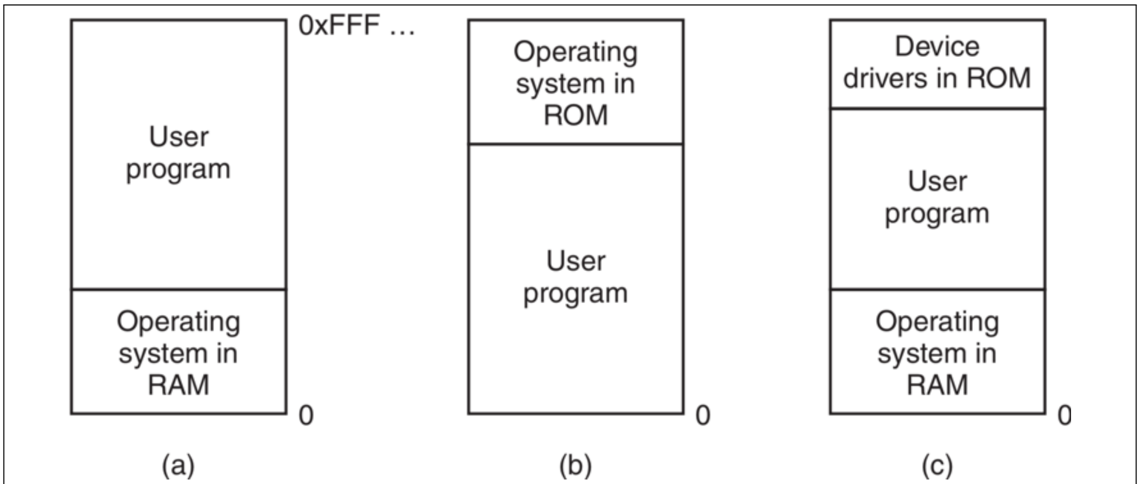


Figure 5.7: Organising memory with an operating system and one user process.

The diagram above shows three different ways of organising memory with an operating system and one user process.

Model	Organisation	Uses
(a)	The operating system may be at the bottom of memory in RAM.	This model was formerly used on mainframes and minicomputers but is rarely used any more.
(b)	The operating system may be in ROM at the top of memory.	This model is used on some handheld computers and embedded systems.

(c)	The device drivers may be at the top of memory in ROM and the operating system may be in RAM at the bottom of memory.	This model was formerly used by early personal computers, such as those running MS-DOS, where part of the operating system (OS) was stored in BIOS.

These models refer to fixed memory addresses and therefore prevent more than one process running at a time.

5.7 Memory abstraction

Definitions

An **address space** is a set of addresses that a process can use to reference memory.

A **physical address** (or **memory address**) identifies a physical location of required data in a memory.

A **logical address** (or **virtual address** / **program address**) is generated by the CPU while a program is running. The logical address is virtual address as it does not exist physically. This address is used as a reference to access the physical memory location by CPU. Logical addresses map to physical addresses.

A **logical address space** is the set of all logical addresses generated by CPU in reference to a program and is viewed from a program's perspective.

Referencing physical memory

How it works

It is possible to reference memory by using the physical address locations.

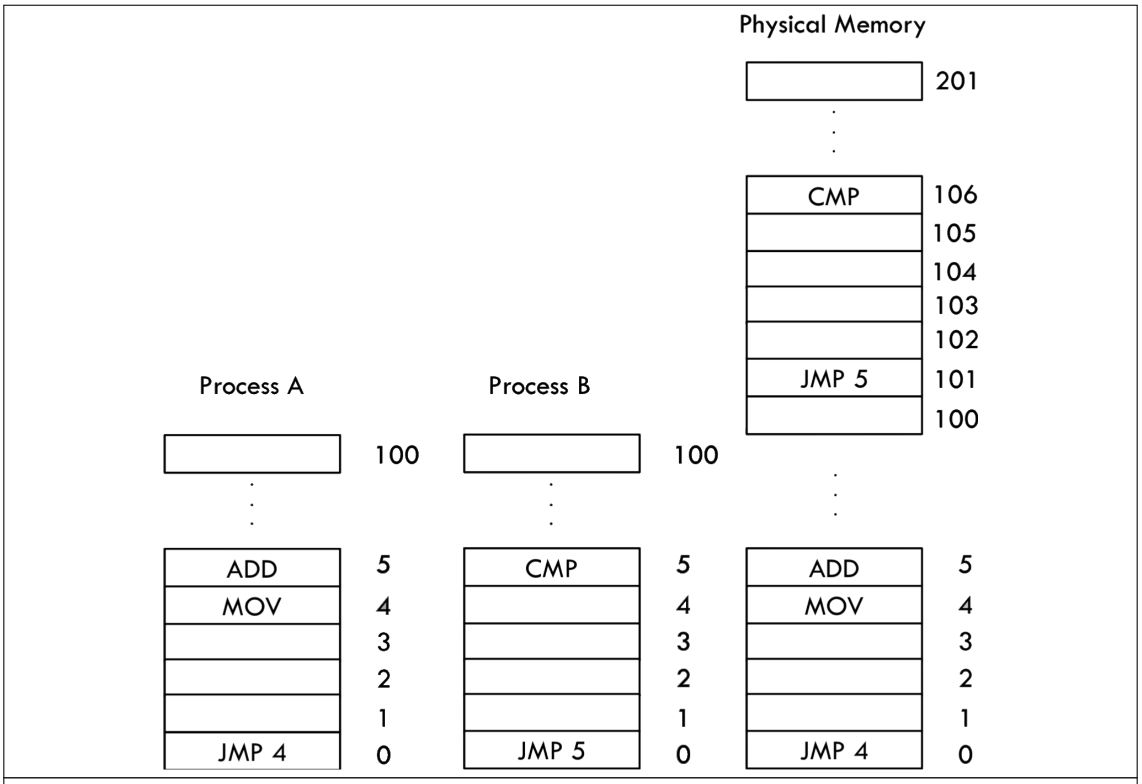


Figure 5.8: Referencing physical memory.

Drawbacks

However, this is not ideal as:

- user programs can address physical memory directly and trash the OS intentionally/accidentally, i.e. models (a) and (c); and
- it is difficult to have multiple programs running at once (via context switch).

For these reasons, users should not be able to reference physical memory directly.

Using logical address spaces

How it works

Programmers can refer to their own address space, the logical address space for a given program.

It is possible for the same logical address, as seen by two different processes, to correspond to different physical addresses. It is important that a distinction is made between the two physical addresses.

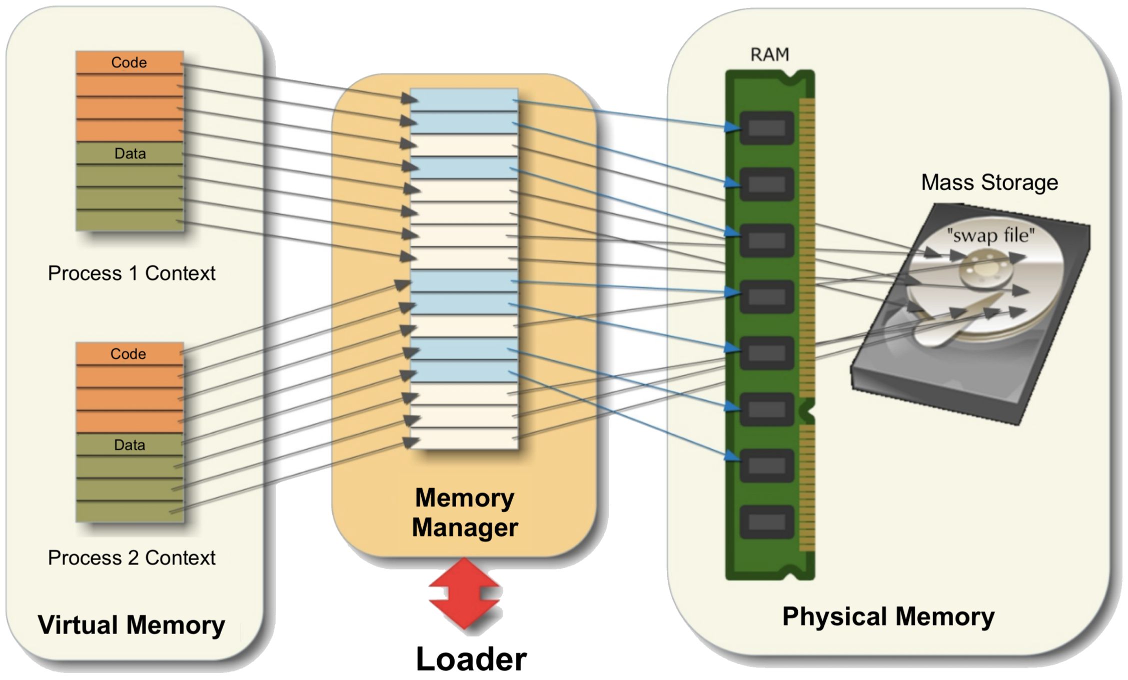


Figure 5.9: Logical address spaces.

When a process is loading, the **loader** is responsible for transferring the program from mass storage to main memory for execution.

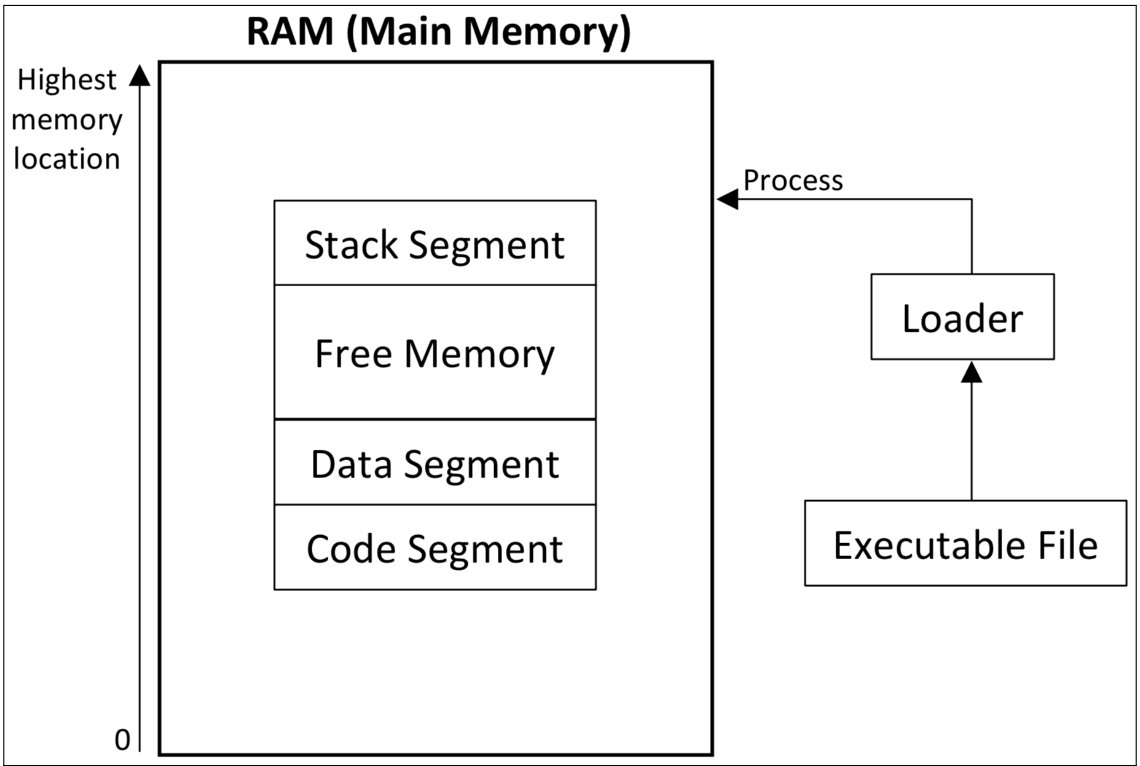


Figure 5.10: Loader.

It is not possible to determine where the program is loaded in physical memory as its location depends on the current contents of memory. As a result, the program's code cannot refer to fixed memory addresses.

As discovered before, machine instructions should not directly access the physical address space and therefore logical addresses are used. These must be converted to physical addresses by the Memory Management Unit (MMU).

5.8 Memory Management Unit (MMU)

Definition

The **Memory Management Unit (MMU)** is responsible for the logical-physical conversion. This is achieved by using registers to record the location of the partition in order to map the logical addresses to the correct physical addresses.

The ultimate aim for the MMU is to perform efficiently such that the CPU does not have to access memory past the cache level. This does not always happen but, in an ideal world, the CPU would only be accessing the faster memory, its registers and cache, as to prevent bottlenecks.

Base and limit registers

The **base register** stores the start address of the partition.

The **limit register** stores the length of the partition.

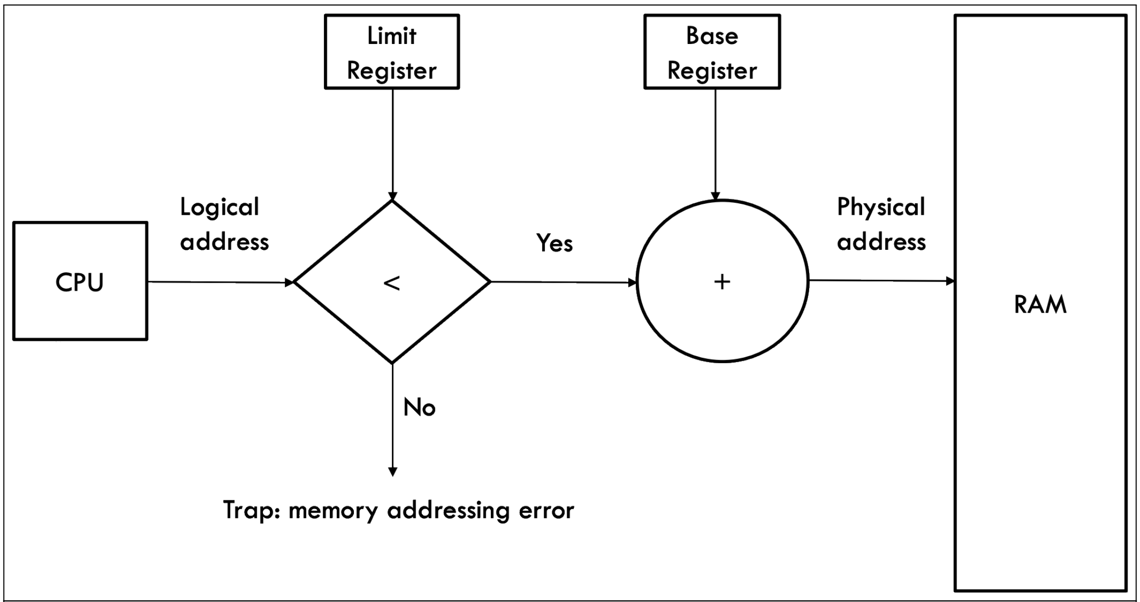


Figure 5.11: Base and limit registers between CPU and RAM.

Memory protection and relocation

Definitions

- **Relocation** – The base register value is added to the logical address in order to map the logical address to a physical address.
- **Protection** – Logical addresses that are greater than the limit register value should point to an invalid memory location. This check ensures that the logical address is within the range of the partition in order to ensure that a process is only accessing its own logical address space.

How it works

The address part of a given machine instruction is used as an offset from the base address.

For example, a machine instruction may have

- a base address of 101; and
- the instruction JMP.

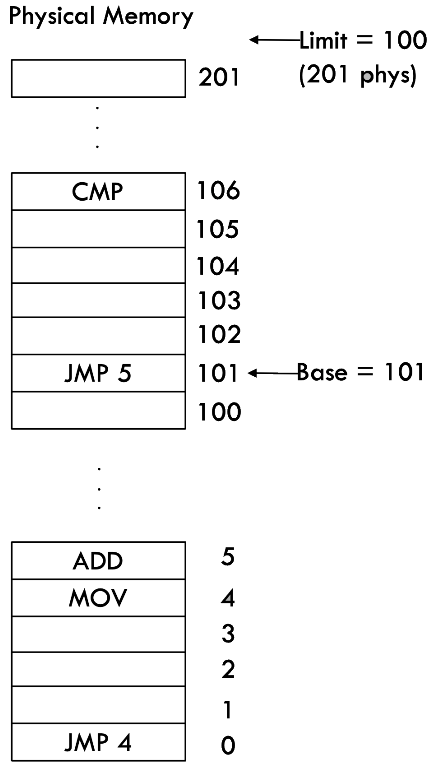


Figure 5.12: Logical addresses.

The operating system (OS) would be responsible for allowing the process to access the memory location desired by the process after the JMP instruction.

The JMP 5 shows a logical address of 5, while the physical address has a base of 101 and therefore the resulting physical address is 106.

In this scenario, if the machine instruction contained `JMP 100` instead, there would be an invalid memory location error. This is because the `JMP 100` instruction would result in a physical address of 201. This violates the logical address limit of 100 and the physical address limit of 201 and therefore this error must be thrown for protection.

5.9 Memory partitioning

Definition

Memory partitioning is the system by which the memory of a computer system is divided into sections for use by the resident programs.

This may be achieved by fixed partitioned memory or variable partitioned memory.

Fixed partitioned memory

How it works

Memory is divided in to several partitions of fixed sizes. The partitions may be different sizes from one another, but remain at the same size once created.

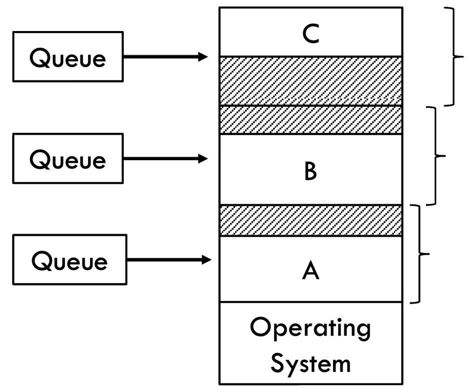


Figure 5.13: Variable partitioned memory.

As shown above, each process (A, B and C) has its own partition.

Evaluation

Advantages	Disadvantages
Multiprogramming, as it is compatible with process co-existence in memory.	**Restricted** as there is a fixed number of possible simultaneous processes.
Easy to implement.	**Internal fragmentation**, as each partition has unused space.
Allows for fast context switching.	**Inefficient space usage** due to internal fragmentation.
	Issues may occur if a process is too large for any partition.

Variable partitioned memory

How it works

Each process is allocated an exact size of memory space. Processes are loaded in to contiguous memory slots until memory is full.

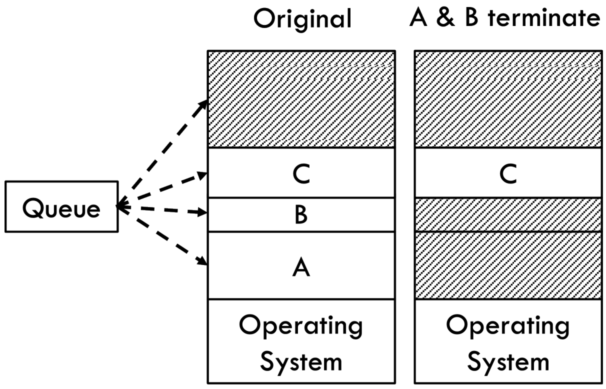


Figure 5.14: Variable partitioned memory.

As shown above, when processes A and B terminate, their memory space is deallocated.

Evaluation

Advantages	Disadvantages
No internal fragmentation.	**Assumes that the memory manager knows how much memory a process requires**.
Efficient space usage as when a process terminates, the space it once occupied in memory is freed up.	**External fragmentation may occur**, where "holes" outside partitions are introduced.
Adjacent fragmentation can be merged.	
The number of parallel processes is not fixed.	

Compaction

Reducing external fragmentation is achieved by compaction.

This is a process whereby processes are physically moved to close the "holes" created by the deallocation of memory that once belonged to processes.

However, this process has a heavy overhead causing slowdowns and therefore, is not commonly performed.

Dealing with processes of varying size

So far, it is been assumed that processes have a fixed size and that the operating system (OS) allocates that memory.

However, it is possible that some processes may try to grow their memory allocation, such as increasing the size of an array.

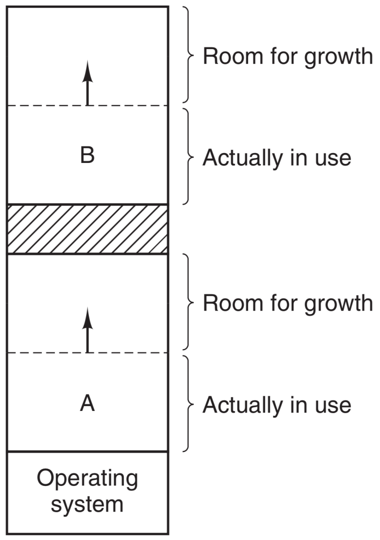


Figure 5.15: Varying process sizes.

As shown in the diagram above, if a hole is adjacent to the process, it can be allocated to the process to allow the process to grow in to that hole.

Otherwise, it is necessary to allocate more memory in a different physical location. If memory is full, swapping may occur or the process needs to be suspended.

5.10 Swapping

Definition

Swapping is a memory reclamation method wherein memory contents not currently in use are swapped to a disk to make the memory available for other applications or processes.

Why is swapping required?

Memory is not an infinite resource, as previously shown in the memory hierarchy. In practice, the total amount of RAM required by all running processes is often larger than the amount available in the computer system.

Idle processes are typically stored on secondary memory as to provide access to more primary memory for processes that are currently in use.

Memory allocations change as processes are loaded and unloaded from memory.

How it works

Swapping shows a simple strategy of loading each process in to memory in its entirety, executing the process for an amount of time and then returning it to secondary memory. This is a similar concept to context switching.

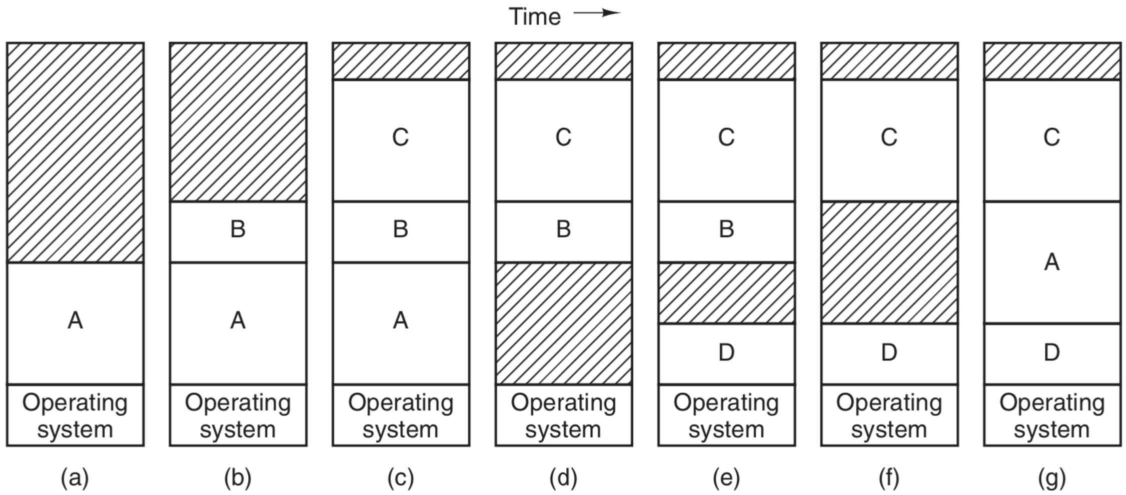


Figure 5.16: Swapping.

However, swapping adds a layer of complexity as the operating system must now manage free space.

5.11 Managing free space

When memory is assigned dynamically via swapping, the operating system must manage it. In general terms, there are two ways to keep track of memory usage: bitmaps and free lists.

Bitmaps

How it works

Memory is divided in to allocation units as small as a few words and as large as several kilobytes.

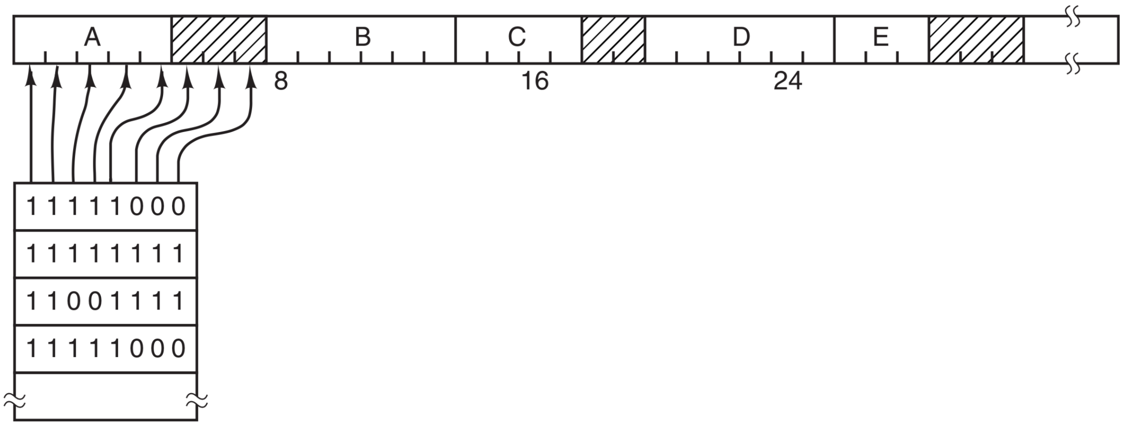


Figure 5.17: Using bitmaps to manage free space.

As shown above, corresponding to each allocation unit is a bit in the bitmap, which is:

- zero (0) if the unit is free; and
- one (1) if the unit is occupied.

Evaluation

Advantages	Disadvantages
Fast as the bitmap array can be accessed directly and the status of the allocation unit can be determined.	**Maintenance overhead** as the status of each allocation unit must be maintained in the bitmap.

Free lists

How it works

A list of processes, denoted as P, and holes, denoted as H, are kept in a list.

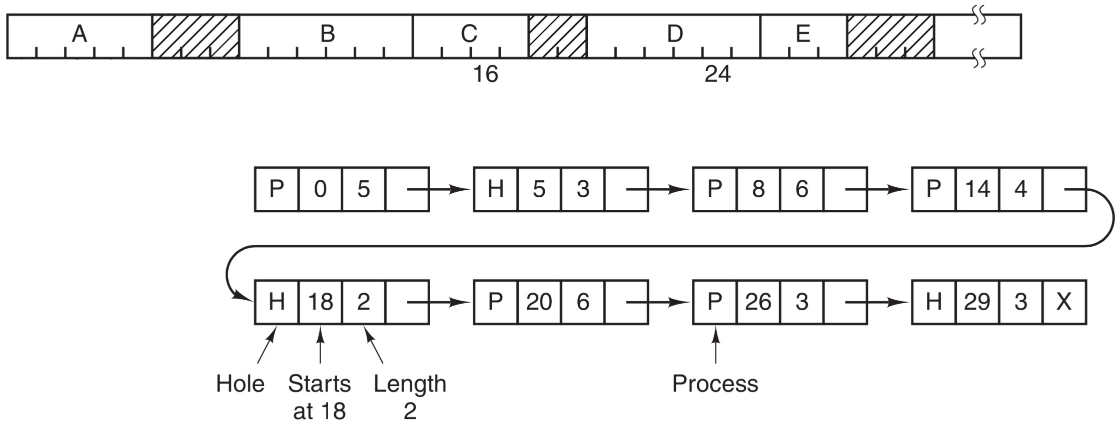


Figure 5.18: Using free lists to manage free space.

In the free list example shown above, there is the following sequence of processes P and holes H in memory:

- a process (P) starting at 0 and occupying 5 spaces;
- a hole (H) starting at 5 and occupying 3 spaces;
- a process (P) starting at 8 and occupying 6 spaces;
- a process (P) starting at 14 and occupying 4 spaces;
- a hole (H) starting at 18 and occupying 2 spaces;
- a process (P) starting at 20 and occupying 6 spaces;
- a process (P) starting at 26 and occupying 3 spaces; and
- a hole (H) starting at 29 and occupying 3 spaces.

Several algorithms, known as placement policies, can be used to allocate memory for new processes.

Evaluation

Advantages	Disadvantages
More compact than using a bitmap as, for example, a sequence of occupied units can be expressed as [P, 0, 5] rather than 11111.	**More complex** as it is more difficult to manage and placement policies are required.

Placement policies

First fit

In first fit, the memory manager scans along the list of segments until it finds a hole that is large enough.

The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit.

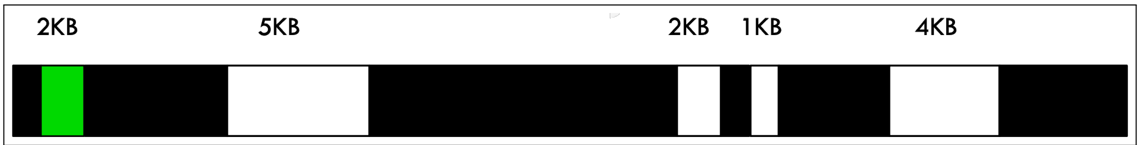


Figure 5.19: Loading a process using first fit with size 2KB.

Figure 5.20: Loading a process using first fit with size 4KB.

Figure 5.21: Loading a process using first fit with size 1KB.

Advantages	Disadvantages
Fast because it searches as little as possible.	**May generate more holes**

Next fit

Next fit is a variation of the first fit policy. It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole.

The next time it is called to find a hole, it starts searching the list from the place where it left off last time, rather than the beginning of the list.

This policy was designed to speed up searching by skipping potential tiny holes that cannot fit a process. However, simulations show that next fit gives slightly worse performance than first fit.

Figure 5.22: Loading a process using next fit with size 2KB.

Figure 5.23: Loading a process using next fit with size 4KB.

Figure 5.24: Loading a process using next fit with size 1KB.

Best fit

In best fit, the memory manager searches the entire list, from beginning to end, and finds the hole that is closest to the actual size required rather than breaking up a larger hole that may be required later.

Figure 5.25: Loading a process using best fit with size 2KB.

Figure 5.26: Loading a process using best fit with size 4KB.

Figure 5.27: Loading a process using best fit with size 1KB.

Advantages	Disadvantages
May reduce memory wastage by using better suited holes for processes.	**May increase memory wastage** as it may generate many tiny holes that are too small to be used by any process. This could be worse than first fit and next fit as they generate larger holes on average that could be used by other processes.
	Slower than first fit and next fit as it requires a search to be completed on the entire list.

Worst fit

Worst fit is the opposite of the best fit policy. It always finds the largest available hole, so that the new hole will be big enough to be useful.

This policy was designed to reduce the number of tiny and unusable holes generated by the best fit policy. However, simulations show that it is not very successful.

Figure 5.28: Loading a process using worst fit with size 2KB.

Figure 5.29: Loading a process using worst fit with size 4KB.

Figure 5.30: Loading a process using worst fit with size 1KB.

Quick fit

In quick fit, the memory manager maintains a table of separate lists for some of the more common sizes requested.

For example, there may be a list for large holes and a list for small holes.

Advantages	Disadvantages
Extremely fast as the lists can be used to quickly find appropriate holes.	**Overhead** as the lists must be maintained.

6. Virtual Memory

6.1 Introduction

Definition

Virtual memory is a memory management technique that provides an idealised abstraction of the storage resources that are actually available on a given machine.

How it works

Virtual memory involves the use of a partition on a computer system's secondary storage device which acts as a form of main memory for the temporary store of data and instructions used in processing by the CPU; this is required when main memory becomes full and is present to prevent crashing.

Parts of processes are stored on secondary storage and loaded in to main memory when required.

This creates the illusion to users of a very large main memory.

Virtual memory and swapping

While swapping allows multiple processes to run whose total size is larger than overall RAM size, virtual memory additionally allows a single process to run whose size is larger than RAM size.

Virtual memory is implemented using paging and segmentation.

6.2 Paging

Definition

Paging is a memory management mechanism that allows the physical address space of a process to be non-contiguous.

Frames and pages

Physical memory is divided in to blocks of equal sizes called **frames**.

In a similar fashion, a process is divided in to blocks of the same size, called **pages**.

The pages from the processes are loaded in to the available frames in physical memory.

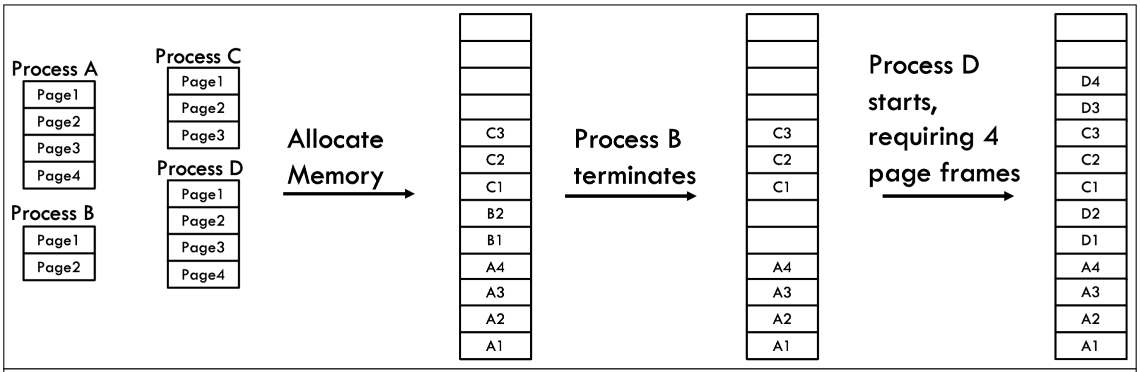


Figure 6.1: Loading process pages in to physical memory frames.

In the diagram above, there are four processes:

- process A which has four pages;
- process B which has two pages;
- process C which has three pages; and
- process D which has four pages.

In the first instance, memory is allocated to processes A-C. Subsequently, process B terminates and its pages are released from the frames.

Later, process D starts. This process has four pages and is therefore split across the two frames available between processes A and C, and after process C.

This is a good solution to address external fragmentation.

Page table

Definition

A **page table** is responsible for mapping virtual pages in to page frames when using paging.

Fields

The layout of a page table is highly machine dependent. However, important common fields of a page table entry include:

- page frame – the most important field that outputs the number of the frame;
- present – records whether the page is present in main memory or secondary memory;
- modified (or dirty bit) – records whether the page has been modified since its last loading, if true then it must be copied back to the disk to be saved;
- protection – includes what kind of access is permitted, read/write/execute; and
- referenced – set by the operating system (OS) when the page is used.

Mapping logical program addresses to physical memory addresses

Each logical address is formed as (p,d) where

- p is the process page number; and
- d is the displacement within that page (offset).

p occupies 4 bits and while d occupies 12 bits.

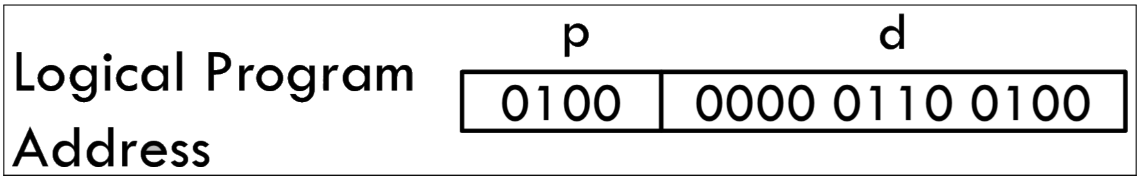


Figure 6.2: Logical program address in paging.

In addition, each process has a page table, which records the memory page frame number for each process page.

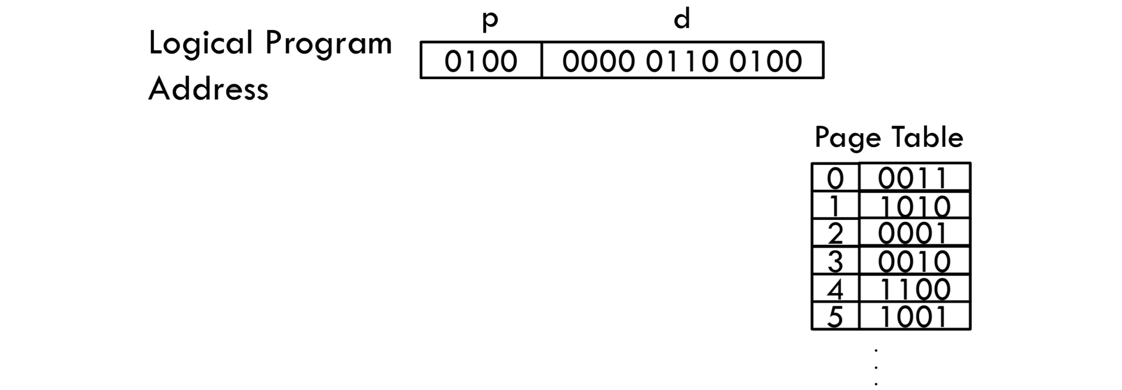


Figure 6.3: Logical program address and respective page table.

The implementation is performed by the hardware, as such the Memory Management Unit (MMU) is responsible for mapping the logical addresses to physical addresses.

Given a 16-bit program address 0100 0000 0110 0100, it can be deduced that:

$$\begin{aligned} p &= & 0100_2 &= & 4_{10} \\ d &= & 0000\ 0110\ 0100_2 &= & 100_{10} \end{aligned}$$

The process page number (p) provides an index to a location in the page table.

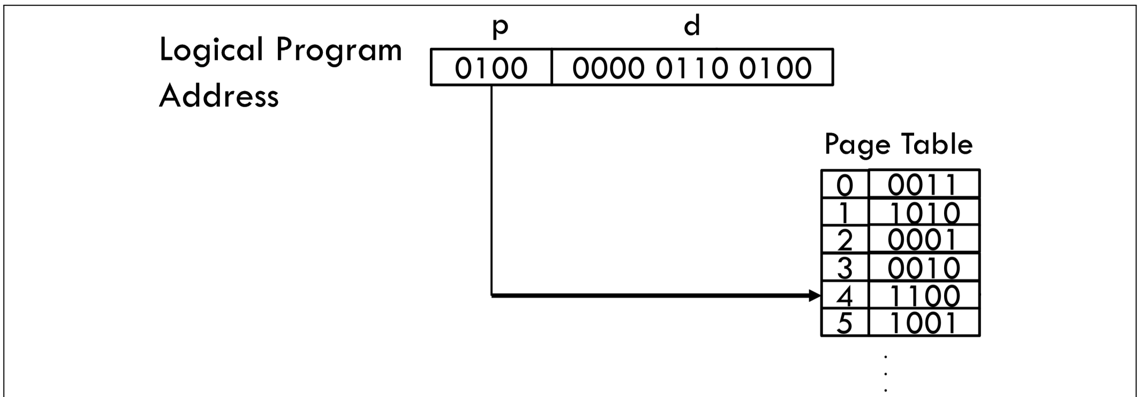


Figure 6.4: Using p as an index to the page table.

The value at index 4 in the page table can then be used as the base (b) for the physical memory address.

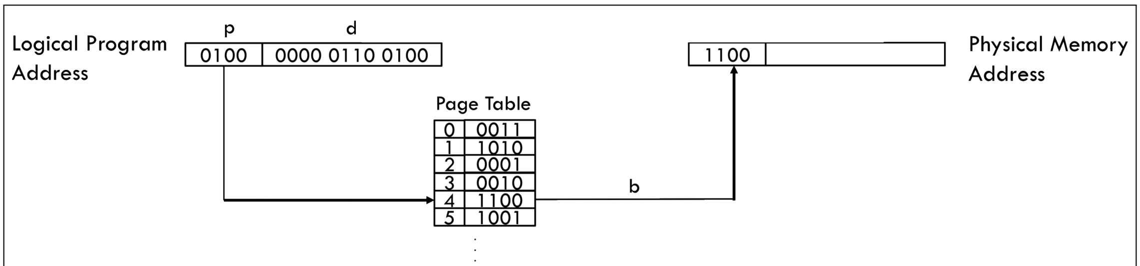


Figure 6.5: Using the value at index 4 as the base (b).

The displacement (d) from the logical program address can then be used to complete the physical memory address.

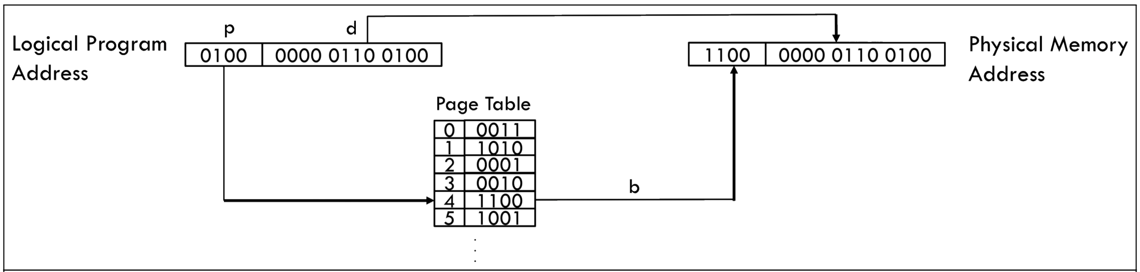


Figure 6.6: Using the displacement value (d) to complete the physical memory address.

This can also be shown by performing the following operations:

$$\text{physical memory address} = \text{page table value at index p} + \text{displacement value (d)}$$
$$\text{physical memory address} = 1100\ 0000\ 0000\ 0000 + 0000\ 0110\ 0100$$
$$\text{physical memory address} = 1100\ 0000\ 0110\ 0100_2 = 104_{10}$$

Page faults

Definition

A **page fault** is a type of exception raised by computer hardware when a running program accesses a memory page that is not currently mapped by the memory management unit (MMU) into the virtual address space of a process.

This means that a specific part of an executing process is not in main memory at the moment it is required and therefore the CPU is not able to access this part of the process.

Page fault handling

When a specific part of an executing process is not in main memory when required, a page fault occurs. ↓ The OS reads out hardware registers to determine which virtual address caused the fault. ↓ The OS computes which page is needed and locates that page on disk. ↓ The OS selects an existing available page frame. ↓ The OS checks if that page frame is modified by inspecting the modified field. ↓ If the page frame has been modified, the OS writes the contents of the page frame back to the disk. ↓ The OS fetches a new page from the disk and replaces the old page in the page frame. This is known as page replacement. ↓ The OS updates the mappings and restarts the trapped instruction by: • marking the virtual page as unmapped by changing the present bit in the page table; and • updating the virtual page address with new translation to physical memory.
Figure 6.7: Page fault handling process

When a process exits, the operating system must release its page table, its pages, and the disk space that the pages occupy when they are on disk. If some of the pages are shared with other processes, the pages in memory and on disk can be released only when the last process using them has terminated.

Page replacement

Definition

Page replacement is required during page fault handling when the OS fetches a new page from the disk and replaces the old page in the page frame.

Page replacement algorithms decide which pages to page out, sometimes called swap out, or write to disk, when a page of memory needs to be allocated.

Objective

Page replacement algorithms aim to decided which pages are to be removed from RAM and placed in mass storage such that there are minimal overheads.

This allows thrashing to be avoided, where the CPU is spending more time swapping pages than actually executing a process.

As such, it must be considered that:

- modified pages must first be saved whereas unmodified pages are just overwritten; and
- it is preferable to not replace an often used page in and out of main memory.

These algorithms use information (bits) from the page table.

Optimal page replacement

The **optimal page replacement algorithm** replaces the page that will not be needed for the longest time in the future.

This is not feasible because:

- it is impossible to know the future of a program; and
- it is impossible to know when a given page will be needed next.

However, this is useful for benchmarking page replacement algorithms a posteriori (i.e. ”what if”).

Not recently used (NRU) page replacement

The **not recently used page replace algorithm** uses the reference and modified bits from the page table to collect useful page usage statistics.

Pages are classified based on the contents of their reference and modified bits in the page table.

Class	**Reference bit**	**Modified bit**	**Meaning**
`Class 0`	0	0	Not referenced and not modified.
`Class 1`	0	1	Not referenced and modified.
`Class 2`	1	0	Referenced and not modified.
`Class 3`	1	1	Referenced and modified.

Figure 6.8: Page classification.

Based on the page classifications, the algorithm removes a page from the lowest numbered non-empty class. For example, should pages exist in `Class 1`, `Class 2` and `Class 3` but not `Class 0`, then a page would be removed from `Class 1`.

A timer interrupt clears the reference bit to distinguish between those pages that have been recently referenced and those which have not been referenced for a given amount of time.

Advantages	Disadvantages
Very low overhead.	**Not optimal** when compared to the optimal page replacement algorithm.
Easy to implement.	

FIFO page replacement

In the **First In, First Out (FIFO) page replacement algorithm**, main memory maintains a list of all pages currently stored in main memory. In which,

- the most recently arrived page is located at the tail; and
- the least recently arrived page is located at the head.

When page replacement occurs, the page at the head is removed and the new page is added to the tail of the list such that the oldest page is removed.

However, this algorithm is rarely used as the oldest page may still be useful.

Second chance page replacement

The **second chance page replacement algorithm** is a variation of the FIFO page replacement algorithm. It attempts to avoid the problem of replacing a heavily used page.

The algorithm checks the reference bit of the oldest page first and performs actions based on the page's state deduced from the reference bit.

Reference bit	Page's state	Actions taken
0	Old and unused.	The page is replaced in the list, in the same manner as the FIFO page replacement algorithm.
1	Old but used.	The page is placed at the end of the list of pages, the load time resets as though it has just arrived in memory and the search continues to the next page.

Figure 6.9: Page classification for the second chance page replacement algorithm.

As shown in the figure above, the second change page replacement algorithm factors in the use of a page as well as its age.

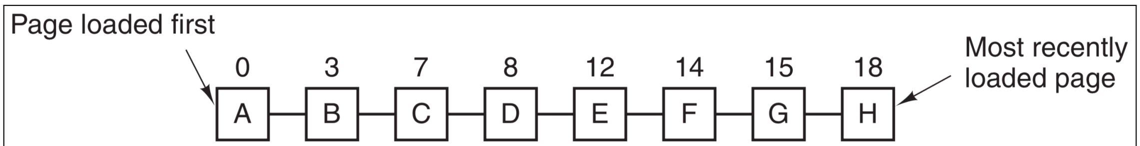


Figure 6.10: Pages shorted in FIFO order.

The figure above shows the pages sorted in FIFO order, with the most recently arrived page at the head of the list of pages and the least recently arrived page at the tail of the list of pages.

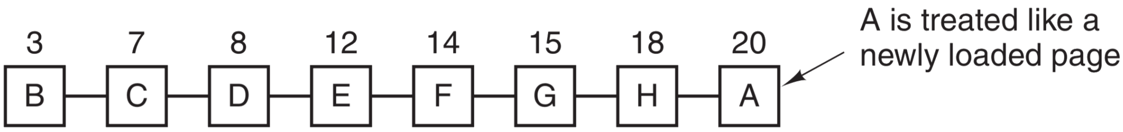


Figure 6.11: "Second chance" for page A.

The figure above shows a "second chance" for page A. If a page fault was to occur at time 20 and page A has a reference bit of 1, the loads times are updated such that it appears that page A has just arrived in memory.

Advantages	Disadvantages
Improvement over the FIFO page replacement algorithm as avoids removing old but useful pages.	**Not optimal** when compared to the optimal page replacement algorithm as unnecessarily moves pages around the list.

Clock page replacement

The **clock page replacement algorithm** is also a variation fo the FIFO page replacement algorithm and is similar to the second chance page replacement algorithm but maintains the pages on a circular list.

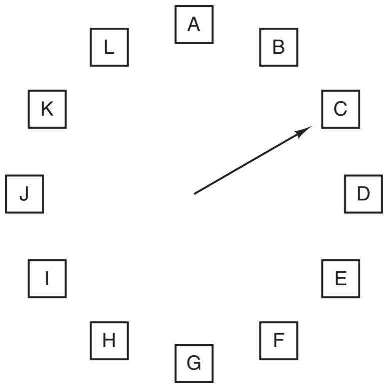


Figure 6.12: Clock page replacement algorithm.

As shown in the figure above, the pages are kept in a circular list in memory, where the hand points to the oldest page.

When a page fault occurs, the page being pointed to by the hand and performs actions based on the page's state deduced from the reference bit.

Reference bit	Page state	Actions taken
0	Old and unused.	The page is replaced in the clock in the same manner as the FIFO page replacement algorithm, and the hand is advanced to the next page.
1	Old but used.	The reference bit for the page is cleared and the hand is advanced to the next page. This is repeated until a page is found with a reference bit of 0.

Figure 6.13: Page classification for the clock page replacement algorithm.

This algorithm is realistic as it avoids the unnecessary movement of pages as seen in the second chance page replacement algorithm.

Least recently used (LRU) page replacement

The **least recently use page replacement algorithm** is based on the ideas that:

- pages that have been heavily used in the last few instructions will probably be heavily used again soon; and
- pages that have not been used for ages will probably remain unused for a long time.

It is necessary for a linked list of all pages to be maintained in memory with:

- the most recently used page at the front; and
- the least recently used page at the back.

As such, the algorithm swaps out pages that have been unused for the longest time.

Advantages	Disadvantages
	Requires time-consuming maintenance of the linked list as the list must be updated on every memory reference such that referenced pages are moved to the front.

Not frequently used (NFU) page replacement

The **not frequently used page replacement algorithm** keeps track of how often a page is used.

A counter is associated with each page and is initially set to 0. At each clock interrupt, the OS will scan all pages in memory. For each page, the reference bit will be added to its associated counter, such that:

- if the reference bit is 0, the counter will remain the same; and
- if the reference bit is 1, the counter will be incremented.

This means that counters can be used to track how often each page has been referenced.

When a page fault occurs, the page with the lowest counter is chosen for replacement.

Summary of page replacement algorithms

Algorithm	**Comment**
Optimal	Not implementable, but useful as a benchmark.
NRU	Very crude approximation of LRU.
FIFO	May throw out important pages.
Second chance	Big improvement over FIFO.
Clock	Realistic.
LRU	Excellent, but difficult to implement exactly.
NFU	Fairly crude approximation to LRU.

Evaluation of paging

Advantages	**Disadvantages**
Almost full utilisation of physical memory.	**Some internal fragmentation** as a process may not use memory in multiples of a page; usually the last page may not use the entire page size.
Can execute programs that have address space larger than physical memory.	**Tuning the page size is tricky** as • a smaller page size leads to less internal fragmentation but more pages are required and therefore the page tables will be larger; and • a larger page size leads to more unused programs to be loaded in to memory but less pages are requires and therefore the page tables will be smaller.
No external fragmentation.	**Large memory consumption** as one page table for each process may consume large amounts of memory.
	Does not support the logical divisions of programs as page sizes are fixed and not based on the actual size of programs.

6.3 Segmentation

Definition

Segmentation is a memory management scheme that supports the logical user-view of programs.

How it works

Each program is split in to variable chunks called segments according to the program's logical structure.

Different compilers for programs may split programs in to different segments.

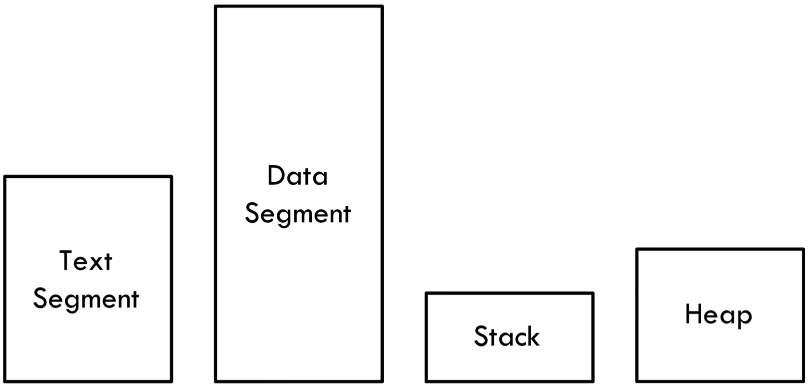


Figure 6.14: Program segments from a C compiler.

As shown in the figure above, a C compiler may generate four segments with different sizes:

- text segment (main code) and libraries;
- data segment;
- the stack; and
- the heap.

The stack and heap are of dynamic size, as they may shrink and grow over the course of the program's lifetime.

Memory is allocated to processes, segment by segment, in non-contiguous areas of physical memory.

Each segment has its own address space.

Segment table

Definition

The **segment table** is responsible for mapping logical program addresses to physical memory addresses when using segmentation.

Mapping logical program addresses to physical memory addresses

Each logical address is formed as (s,d) where

- s is the segment reference; and
- d is the displacement within that segment (offset).

s occupies 4 bits and while d occupies 12 bits.

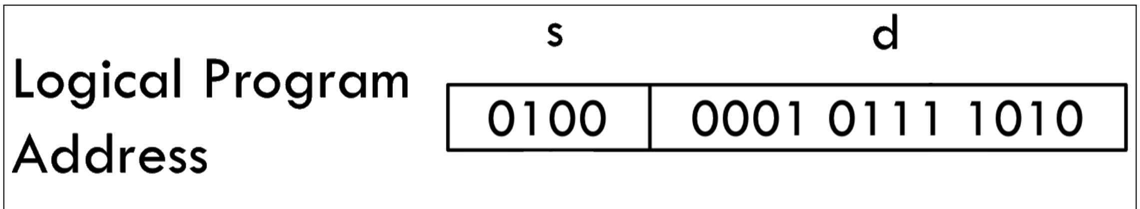


Figure 6.15: Logical program address in segmentation.

In addition, each process has a segment table, which records the base address and the length of the segment.

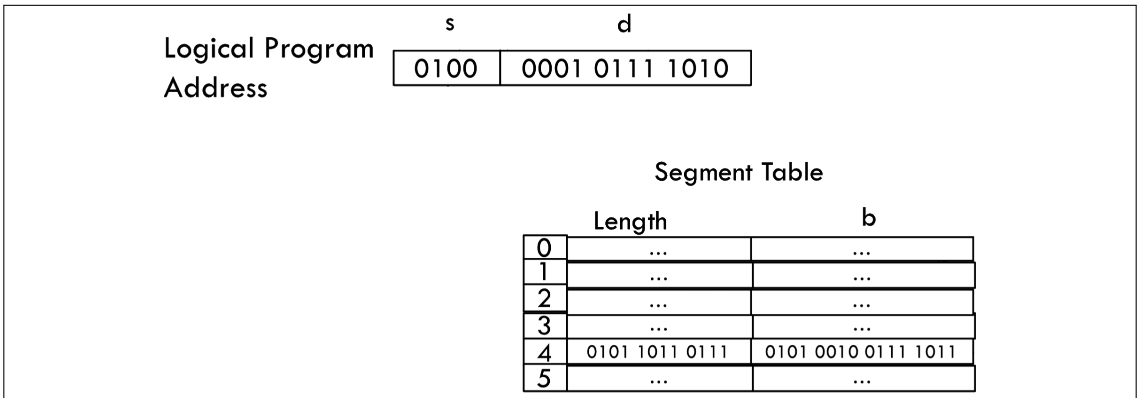


Figure 6.16: Logical program address and respective segment table.

The implementation is performed by the hardware, as such the Memory Management Unit (MMU) is responsible for mapping the logical addresses to physical addresses.

Given a 16-bit program address 0100 0001 0111 1010, it can be deduced that:

$$\begin{aligned} p &= & 0100_2 &= & 4_{10} \\ d &= & 0001\ 0111\ 1010_2 &= & 378_{10} \end{aligned}$$

The segment reference (s) provides an index to a location in the segment table.

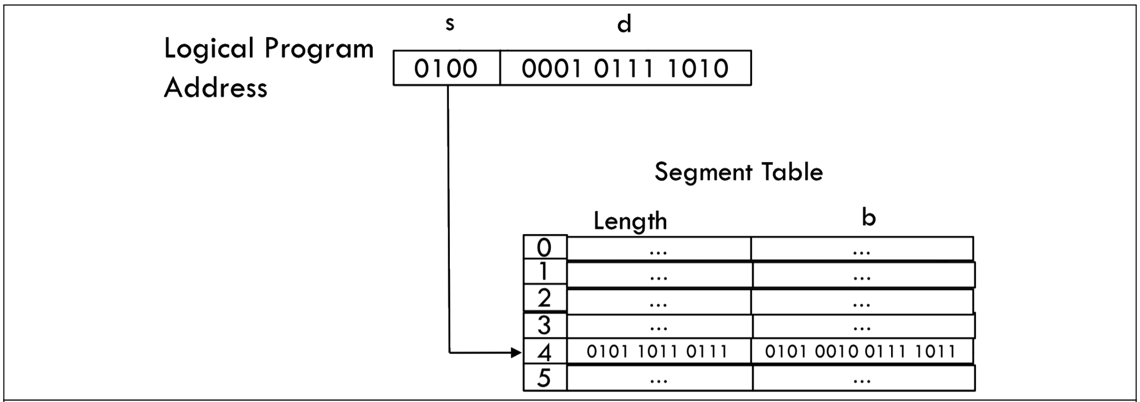


Figure 6.17: Using s as an index to the segment table.

A check is also performed to ensure that the displacement value (d) is less than the segment length recorded in the segment table.

If the check is passed, the sum of the value of the base address (b) at index 4 in the segment table and the displacement value (d) from the logical program address can then be used as the physical memory address.

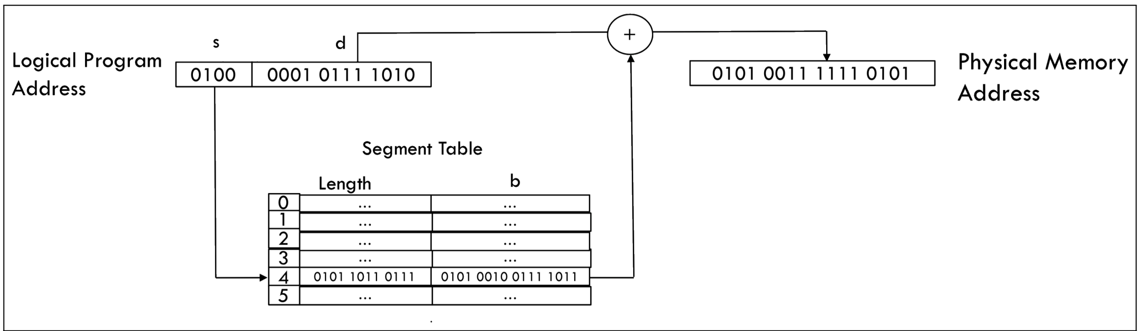


Figure 6.18: Using the value at index 4 as the base (b).

This can also be shown by performing the following operations:

$$\text{physical memory address} = \text{base address (b)} + \text{displacement value (d)}$$
$$\text{physical memory address} = 0101\ 0010\ 0111\ 1011 + 001\ 0111\ 1010$$
$$\text{physical memory address} = 0101\ 0011\ 1111\ 0101_2 = 21493_{10}$$

Protection and sharing

Segmentation aids protection as it:

- assigns different modes, such as read, write and execute, to each segment; and
- checks that memory references do not exceed the segment length, therefore preventing a process accessing another process' segments.

Segmentation aids sharing as a shared segment can be referenced by multiple processes, such as libraries.

Evaluation of segmentation

Advantages	Disadvantages
Support the logical divisions of programs as segments are based on the program's attributes.	**Wasted space** as segments are usually much larger than pages.
Segments can grow and shrink dynamically and independently, such as the stack and heap.	**External fragmentation**.
Aids sharing and protection.	

6.4 Comparison of paging and segmentation

Attribute	Paging	Segmentation
Uses	Useful to have more address space without having to purchase more physical memory.	Useful to allow programs to be broken up in to independent logical address spaces.
Memory division	Physical division of memory which is transparent to the user.	Logical division of memory which is visible to the user.
Size	Fixed size as each page is a pre-determined fixed size.	Variable size as segments can be dynamic, such as stack and heap.
Fragmentation	No external fragmentation.	Generates memory holes.
Address space	The total address space can exceed the size of physical memory.	

6.5 Paging segments

Paging segments is a method of combining paging and segmentation by assigning each segment with a page table.

Segments are typically larger than pages and, in the case where a segment does not fit in physical memory, this method must be used.

This allows the advantages of both memory management schemes to be present, such that this method:

- avoids external fragmentation;
- aids sharing and protection; and
- supports the user-view of programs.

Paging segments is roughly the method used in modern systems, such as Intel-based systems.

7. File Management

7.1 Non-volatile storage

Definition

Non-volatile storage is a type of computer memory that can retrieve stored information even after having been power cycled.

Why is it used?

It is desirable that data should persist for later use and further processing. This should happen even after:

- power-down;
- critical errors;
- crashes; and
- forced shut-downs.

Solely relying on RAM would present issues as:

- information in the process address space would be lost when the process terminates; and
- for most applications it is far too small to contain all of the required data and instructions at any given time.

7.2 Filing system

Definition

A **filing system** is a method of organising and retrieving files from a storage medium, such as a hard drive.

Objectives

A filing system aims to:

- facilitate the long-term storage of data;
- allow the creation and deletion of files with automatic management of secondary storage;
- allow for file reference using symbolic names;
- provide access control to protect files against unauthorised access and allow sharing of files when required; and
- protect files against system failure.

7.3 Files

Definition

A **file** is a uniform logical unit of information created by a process.

Characteristics

Files also have an address space. However, the mapping in to a mass storage unit rather than RAM. As such, files can be described as a named collection of related information that is recorded on secondary storage, such as:

- the set of lines in a program; or
- the set of words in a text document.

Files are used for storing large amounts of data in the long-term.

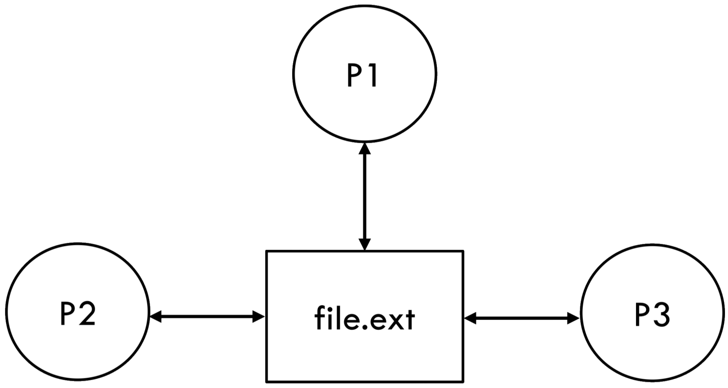


Figure 7.1: Processes P1, P2 and P3 accessing a file.

As shown in the figure above, processes are able to access a file's data concurrently.

File naming

Motivation

File naming removes the need for a user to provide numerical addresses to access files. Instead, files can be accessed using a user-friendly name.

File naming conventions

Different OSs enforce different file naming conventions but most follow a common pattern.

Many OSs, such as Windows and Unix-based OSs, support up to approximately 260 characters for file names.

OSs place restrictions on which characters can be used in file names. For example:

- in Windows, the question-mark character (?) is not allowed in file names; while
- in Unix-based OSs, the question-mark character (?) is valid.

Some OSs distinguish between upper-case and lower-case in file names. For example:

- in Windows, files named `ABC`, `Abc` and `abc` would all represent the same file; while
- in Unix-based OSs, files name `ABC`, `Abc` and `abc` would all represent different files as the OS is case sensitive.

File extensions

A **file extension** is a string of characters attached to a filename, usually preceded by a full stop and indicating the format of the file.

file.ext

Figure 7.2: Base name and extension.

The figure above shows the format for a typical file name, consisting of:

- a base name – often indicative of the file's contents; and
- a file extension – informs the user and the OS what type of data the file contains.

In MS-DOS, only three (3) characters were allowed for file extensions. While, in Unix-based OSs, the length of a file extension is down to the discretion of the user or process.

In Unix-based OSs, file extensions are not enforced by the OS. However, for example, a C compiler would require a file with the extension `.c` to compile.

GUI-based OSs typically attach meanings to extensions and associate applications to file extensions. For example, opening a file with the `.docx` extension may launch Microsoft Word.

However, this poses an issue as it is easy to ”trick” and corrupt files by modifying extensions. This is addressed in macOS, which examines the contents of files in order to determine the file's type.

Extension	**Meaning**
`.bak`	Backup file.
`.c`	C source program.
`.gif`	Graphical Interchange Format image.
`.html`	World Wide Web HyperText Markup Language document.
`.iso`	ISO image of a CD-ROM (for burning to a CD).
`.jpg`	Still picture encoded with the JPEG standard.
`.mp3`	Music encoded in MPEG layer 3 audio format.
`.mpg`	Movie encoded with the MPEG standard.
`.o`	Object file (compiler output, not yet linked).
`.pdf`	Portable Document Format file.
`.ps`	PostScript file.
`.tex`	Input for the TEX formatting program.
`.txt`	General text file.
`.zip`	Compressed archive.

Figure 7.3: Common file extensions.

The figure above shows some common file extensions. However, this list is not exhaustive.

File attributes

Definition

File attributes are metadata associated with computer files that define file system behaviour. Each attribute can have one of two states:

- set; and
- cleared.

Common attributes

Attributes are considered distinct from other metadata.

Attribute	Meaning
`Protection`	Who can access the file and in what way.
`Password`	Password needed to access the file.
`Creator`	ID of the person who created the file.
`Owner`	Current owner.
`Read-only flag`	0 for read/write and 1 for read-only.
`Hidden flag`	0 for normal and 1 for do not display in listings.
`System flag`	0 for normal file and 1 for system file.
`Archive flag`	0 for has been backed up and 1 for needs to be backed up.
`ASCII/binary flag`	0 for ASCII file and 1 for binary file
`Random access flag`	0 for sequential access only and 1 for random access.
`Temporary flag`	0 for normal and 1 for delete file on process exit.
`Lock flags`	0 for unlocked and `non-zero` for locked.
`Record length`	Number of bytes in a record.
`Key position`	Offset of the key within each record.
`Key length`	Number of bytes in the key field.
`Creation time`	Date and time the file was created.
`Time of last access`	Date and time the file was last accessed.
`Time of last change`	Date and time the file was last changed.
`Current size`	Number of bytes in the file.
`Maximum size`	Number of bytes the file may grow to.

Figure 7.4: Common file attributes.

The figure above shows some common file attributes. However, this list is not exhaustive.

File structure

Files can be structured in any of several ways. Three common methods are:

- byte sequence;
- record sequence; and
- tree.

Byte sequence

In the **byte sequence** method, the OS considers a file to be an unstructured sequence of bytes.

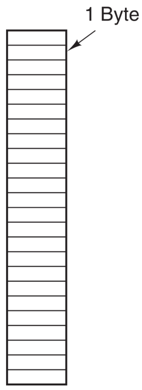


Figure 7.5: Byte sequence for a file.

The OS is not concerned with the file's contents. Instead, it is presented with bytes of data. Any meaning must be imposed by user-level programs.

Both UNIX-based OSs and Windows use this approach.

Record sequence

In the **record sequence** method, a file is a sequence of fixed-length records, each with some internal structure.

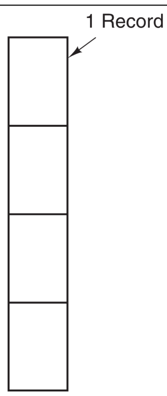


Figure 7.6: Record sequence for a file.

Central to the idea of a file being a sequence of records is the idea that:

- the read operation returns one record; and
- the write operation overwrites or appends one record.

No modern general-purpose system uses this model as its primary file system. However, this was useful when 80-column punched cards and 132-character line printer paper were used on mainframe computer systems.

Tree

In the **tree** method, a file consists of a tree of variable-length records, each containing a key field in a fixed position in the record.

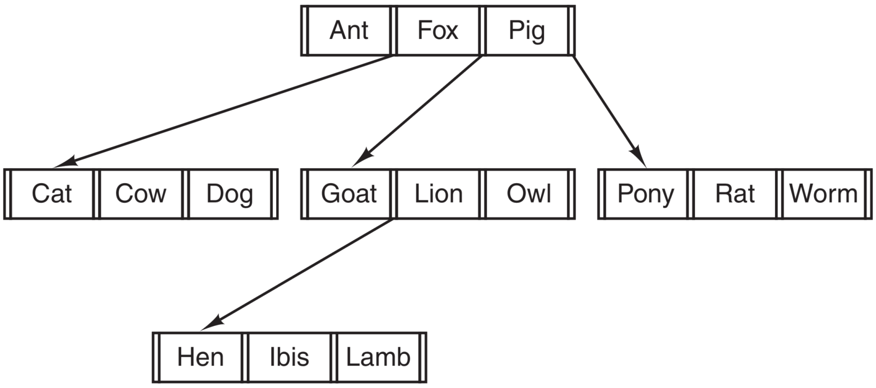


Figure 7.7: Tree for a file.

The tree is sorted on the key field, to allow rapid searching for a particular key.

This type of file is different from the unstructured byte streams used in Windows and Unix-based OSs but is used on some large mainframe computers for commercial data processing and database systems.

File access

Definition

File access refers to the way in which the files stored may be accessed.

Sequential access

In **sequential access** a process can read all the bytes or records in a file in order, starting at the beginning.

It is not possible to skip around and read them out of order. However, sequential files can be rewound so they could be read as often as needed.

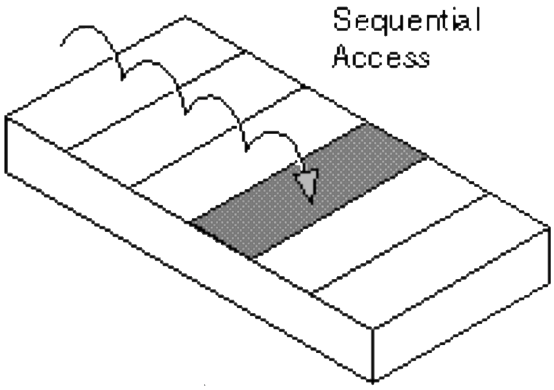


Figure 7.8: Sequential access of a file.

As shown in the figure above, sequential access allows the operations:

- read next; and
- write next.

Sequential files were convenient when the storage medium was magnetic tape rather than disk, as magnetic tapes are accessed sequentially. However, this method of access is still interesting today due to the locality principle (the tendency of a processor to access the same set of memory locations repetitively over a short period of time).

Random (direct) access

In **random (direct) access** a process can read the bytes or records of a file out of order and access records by key rather than by position.

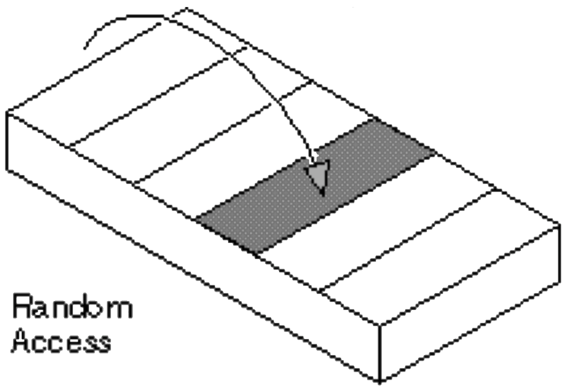


Figure 7.9: Random (direct) access of a file.

As shown in the figure above, random (direct) access allows the operations:

- read n; and
- write n.

n represents a relative block number.

Random (direct) access is essential for most modern applications.

File types

Many OSs support several types of files, as shown below.

File type	Description
Regular	Ordinary files that contain data.
Executable	files that contain program code that can be executed.
Directory / folder	System file containing references to other files.
Character special	Not true files, rather directory entries which refer to character devices and allow communication with I/O devices such as printers.
Block special	Not true files, rather directory entries which refer to block devices and are used for communication with storage devices such as disks and memory.

Figure 7.10: File types

Typical file operations

OSs offer several system calls for file management, as shown below.

Operation	Description
`Read`	Data are read from file.
`Write`	Data are written to an existing file.
`Append`	Restricted form of write. Data can only be added to the end of a file, while maintaining the existing contents of the file
`Seek`	Used for random access files. Repositions the file pointer to a specific location in the file.
`Get attributes`	Retrieve the attributes of the file. For example, used by the C compiler.
`Set attributes`	Some attributes can be set by the user, such as protection-mode.
`Rename`	Change the name of an existing file.

Figure 7.11: Typical file operations.

7.4 Directories

Definition

A **directory** (or **folder**) is a file system cataloging structure which contains references to other computer files, and possibly other directories.

Characteristics

Most filing systems allow files to be grouped together in to directories, resulting in a more logical organisation.

Directories allow:

- operations to be performed in bulk on groups of files, such as copying files or setting one of their attributes; and
- different files to have the same file name provided that they are in different directories.

File descriptor table

Each directory is managed using a special file containing a file descriptor table.

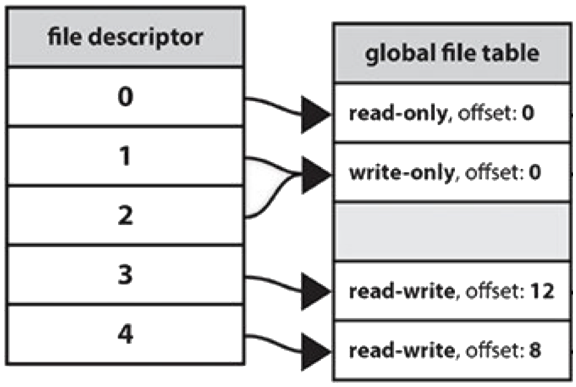


Figure 7.12: File descriptor table.

As shown in the figure above, the file descriptor table contains descriptors for each file under that directory which correspond to specific entries on the global file table.

Directory structure

Definition

The **directory structure** determines how the OS organises the files and directories in the file system.

Single-level directory systems

Single-level directory systems use one directory for all files in the file system.

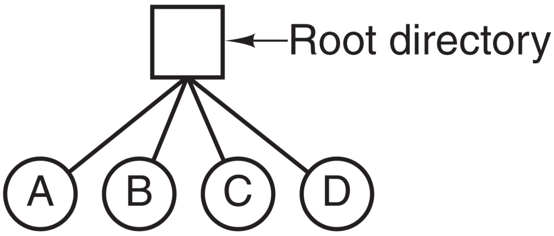


Figure 7.13: Single-level directory system.

The figure above shows a single-level directory system with four files owned by three different users A, B and C.

Advantages	Disadvantages
Simplicity as the root directory is the only directory to be implemented.	**Limited grouping capability** as the root directory is the only directory present in the file system.
Ability to quickly find files as only the root directory must be searched.	**Naming issues** as the root directory is the only directory present in the file system and so no two files can have the same name.

Two-level directory systems

Two-level directory systems use a separate directory for each user.

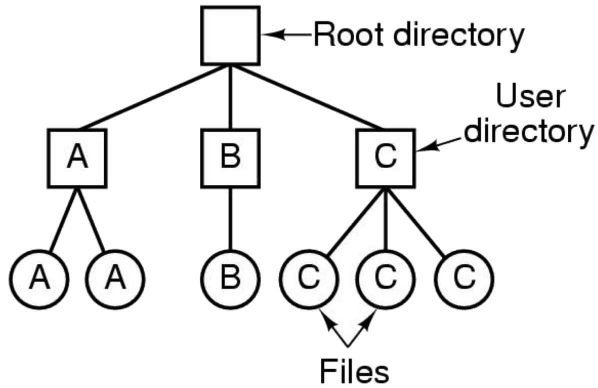


Figure 7.14: Two-level directory system.

The figure above shows a two-level directory system with directories for the users A, B and C.

Advantages	Disadvantages
Flexible naming as files can have the same name provided that they are in different user directories.	**Limited grouping capability** as the user directory is the only directory present in the file system.

Hierarchical directory systems

Hierarchical directory systems maintain directories in a tree-like structure.

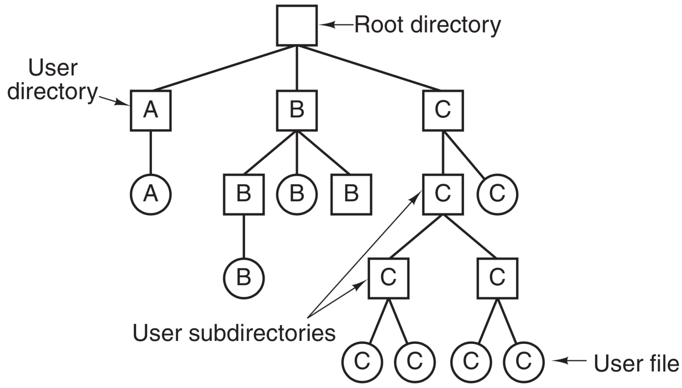


Figure 7.15: Hierarchical directory system.

The figure above shows a two-level directory system with directories for the users `A`, `B` and `C`.

Advantages	Disadvantages
Grouping capabilities as directories can be created by users to organise files.	**Some complexity** as a method for browsing and locating files must be implemented.
More flexible naming as files can have the same name provided that they are in different directories.	

The exact implementation may differ between OSs.

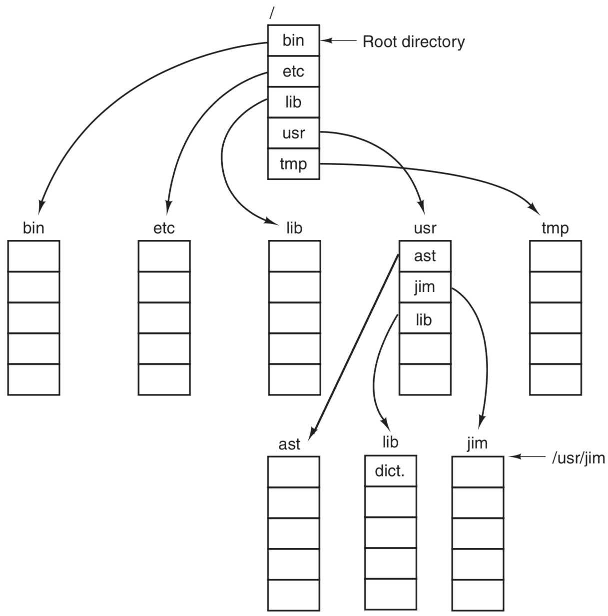


Figure 7.16: Unix directory tree.

The figure above shows the directory tree used by Unix-based OSs.

Directory paths

Browsing files

Two common methods are used for browsing files: absolute path name; and relative path name.

An **absolute path name** is relative to the root directory and is unique. For example:

- in MS-DOS and Windows – `\NTU\syssoftware\demo.c`; and
- in Unix-based OSs – `/NTU/syssoftware/demo.c`.

A **relative path name** is relative to the current directory (or present working directory). For example:

- in MS-DOS and Windows – if the current working directory is `\NTU \syssoftware`, the the file whose absolute path is `\NTU\syssoftware\demo.c` can be referenced as `demo.c`; and
- in Unix-based OSs – if the current working directory is `/NTU/syssoftware`, the the file whose absolute path is `/NTU/syssoftware/demo.c` can be referenced as `demo.c`.

Special enteries

Some special entries exist to provide shortcuts:

- `.` (dot) – shortcut for current directory; and
- `..` (dot dot) – shortcut for parent directory.

Example Unix directory commands

Given that it is required to backup the `dictionary` file located in the `lib` directory:

- if the current directory is `/` (root directory), then the command to perform this action would be `cp /usr/lib/dictionary /usr/lib/dictionary.bak`; and
- if the current directory is `/usr/lib/`, then the command to perform this action would be `cp dictionary dictionary.bak`.

Given that it is required to move the `dictionary` file located in the `lib` directory to the `ast` directory:

- if the current directory is `/` (root directory), then the command to perform this action would be `mv /usr/lib/dictionary /usr/ast/dictionary`; and
- if the current directory is `/usr/ast/`, then the command to perform this action would be `mv dictionary /usr/ast/dictionary`.

Directory file operations

OSs offer several system calls for directories, as shown below.

Operation	Description
`Create`	An empty directory is created.
`Delete`	An existing directory can be deleted. Non-empty directories on many Unix-based OSs will require further clarification by using a "`force`" command.
`Opendir`	Directories can be read. For example, when listing all of the files in the directory.
`Closedir`	When a directory has been read, it should be closed to free up space.
`Rename`	Change the name of an existing directory.
`Link`	A technique that allows a file to appear in more than one directory.
`Unlink`	If a file is unlinked, it is only normally present to one directory.

Figure 7.17: Typical directory operations.

There are more systems calls depending on the OS used.

8. File System Implementation

8.1 File system layout

File systems are stored on drives. Most disks can be divided up into one or more **partitions/volumes**, each holding an independent file system.

The terms ”drives” and ”disk” can be used interchangeably despite referring to different types of mass storage

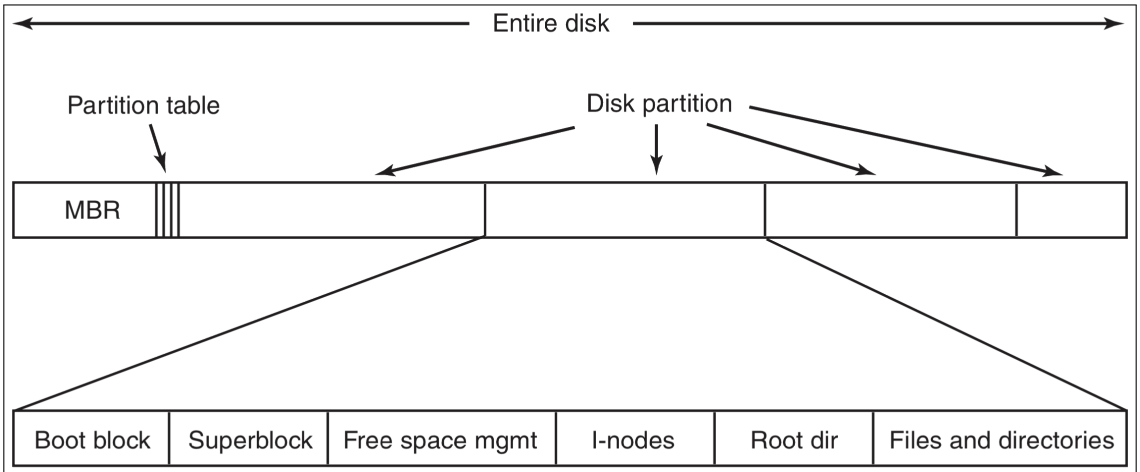


Figure 8.1: File system layout.

`Sector 0` of the disk is the **Master Boot Record (MBR)**. This is used to boot the computer via a **boot block** from a specified partition, from which the OS is loaded.

The MBR program locates the active partition.

↓

The MBR reads in the first block in the active partition, called the boot block.

↓

The program in the boot block is executed and loads the OS contained in that partition.

Figure 8.2: Process for booting an OS from a disk.

The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition. Every partition starts with a boot block, even if it does not contain a bootable operating system. Besides, it may contain one in the future.

The **superblock** contains key parameters about the file system on the partition and is read in to memory when the computer is booted or the file system is first touched. Typical information in the superblock includes:

- a magic number to identify the file system type;
- the number of blocks in the file system; and
- other key administrative information.

8.2 File block allocation methods

Objective

File block allocation methods aim to keep track of which disk blocks go with which file.

Various methods are used in different OSs, including:

- contiguous allocation;
- linked list allocation (non-contiguous);
- linked list with file allocation table; and
- i-nodes.

Contiguous allocation

How it works

In **contiguous allocation**, disks are split in to blocks of fixed size.

This method is often used for CDs.

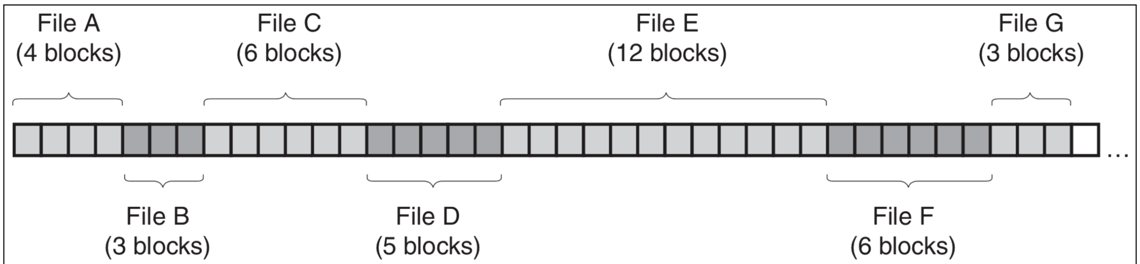


Figure 8.3: Contiguous allocation for files A-G.

The figure above shows how a different number of cotiguous blocks may be allocated to different files. For example:

- if the fixed size of the blocks was 1KB, a file with size 50KB would be assigned to 50 consecutive blocks; and
- if the fixed size of the blocks was 2KB, a file with size 50KB would be assigned to 25 consecutive blocks.

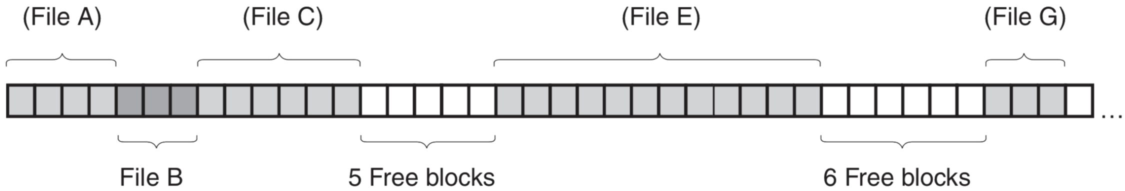


Figure 8.4: Contiguous allocation for files A-G, where files D and F are removed.

As shown in the figure above, removing files frees up the blocks that were once allocated to the deleted file.

Evaluation

Advantages	Disadvantages
Simple implementation as it is only required to store the first block address and its length.	**Overhead** as it is required to track the size of the files when initially created.
Good performance.	**Files cannot grow** as their number of blocks is fixed and the size of the blocks is fixed. For example, it would not be possible to edit an existing document and introduce new data.
Allows easy random access.	**Internal fragmentation** as the data inside a file block may be smaller than the fixed block size.
Resilient to drive faults as damage to a single block results in only localised loss of data.	**External fragmentation** as when files are deleted, holes may be generated. This can be overcome by performing compaction, however this increases the system overhead.

Linked list allocation (non-contiguous)

How it works

In **linked list allocation**, files are stored as linked lists of blocks.

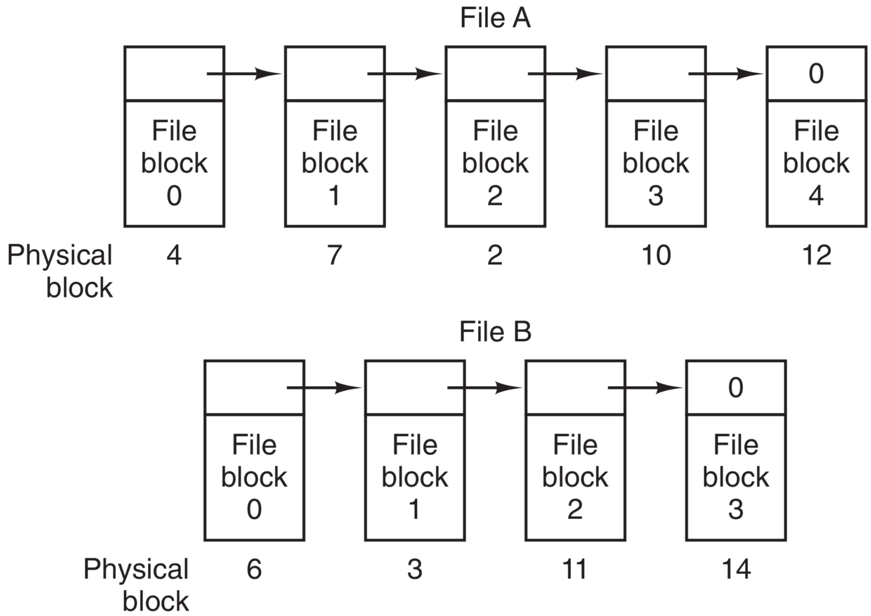


Figure 8.5: Linked list allocation.

As shown in the figure above, each block contains:

- a pointer to the next block (occupies the first word of each block); and
- data.

The final block contains a null pointer, usually denoted by a zero (0) as no further blocks for the file exist.

Evaluation

Advantages	Disadvantages
Every block can be used.	**Does not support random access** as it is very slow due to the necessity to traverse through the chain of blocks.
File size does not have to be known beforehand as files can grow because new blocks can be easily added by changing the null pointer on the last block to point to the newly added block.	**Wasted space** as some space is lost for useful data within each block due to storage of the pointer.
No external fragmentation.	
No internal fragmentation, except for the last block.	

Linked list with file allocation table (FAT)

When using a **linked list with file allocation table (FAT)**, the FAT is located in main memory and stores the pointer of all of the blocks.

This method is used by MS-DOS and older versions of Windows.

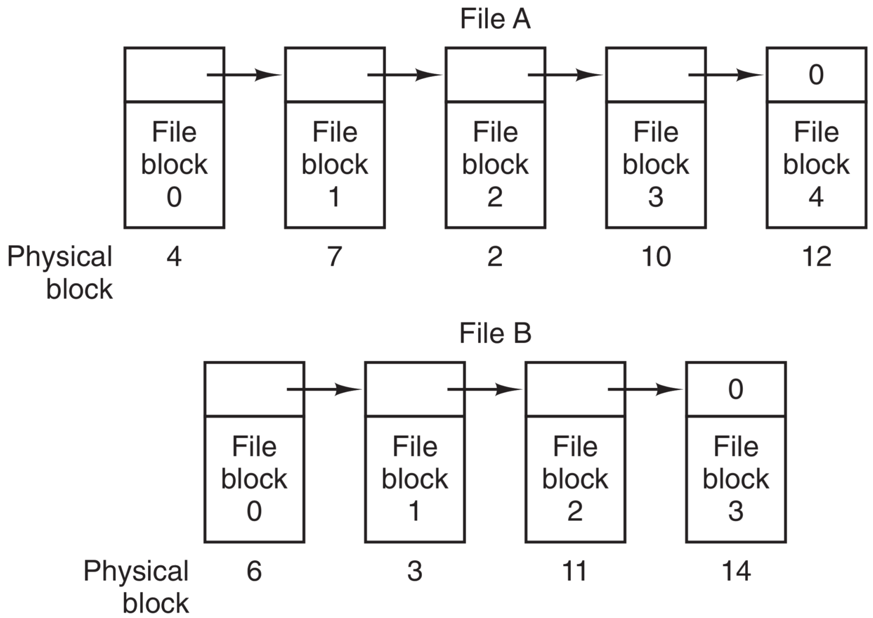


Figure 8.6: Linked list allocation for files A and B.

The figure above shows the linked list allocation for files A and B:

- file A uses disk blocks 4, 7, 2, 10, and 12; and
- file B uses disk blocks 6, 3, 11, and 14.

The pointer word from each block is put in to the FAT in main memory.

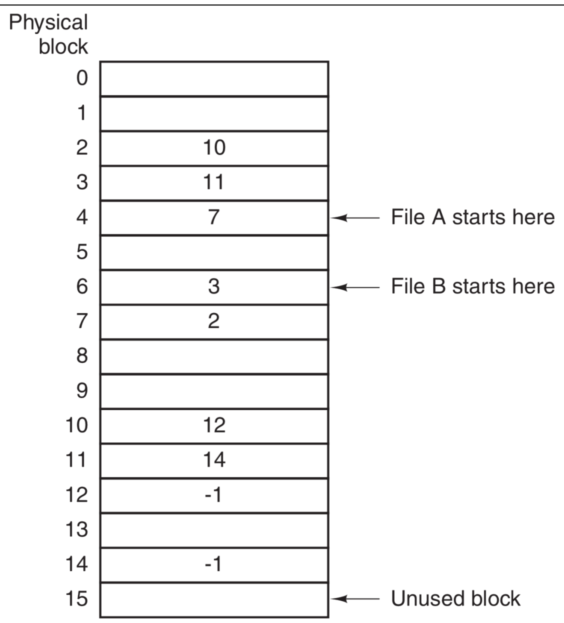


Figure 8.7: Linked list using a FAT in main memory.

For example, using the FAT in the figure above:

- for file A, it is possible to start with block 4 and follow the chain to the end at 12; and
- for file B, it is possible to start with block 6 and follow the chain to the end at 14.

Both chains are terminated with a special marker, usually denoted as -1, that is not a valid block number. This is used to signal the end of the blocks for a given file.

Evaluation

Advantages	Disadvantages
No wasted space as the pointer is not stored in the block, as it was in linked list allocation.	**Does not scale well to large disks** as the FAT may get too large for main memory, especially for drives with large capacities that require more pointers to be stored as there are more files and more blocks.
Random access is much faster than linked list allocation as the FAT is stored in main memory, which is a faster storage medium.	**Serious data loss** can occur should damage be inflicted on the FAT. However, this can be mitigated by storing several backups of the FAT.
No disk references are required when following a chain in the FAT as it is stored in main memory.	

Index-nodes (I-nodes)

When using **index-nodes** (i-nodes), each file is associated with an i-node which lists:

- all of the attributes of the file's blocks; and
- the disk addresses of the file's blocks.

This method is used by Unix-based OSs.

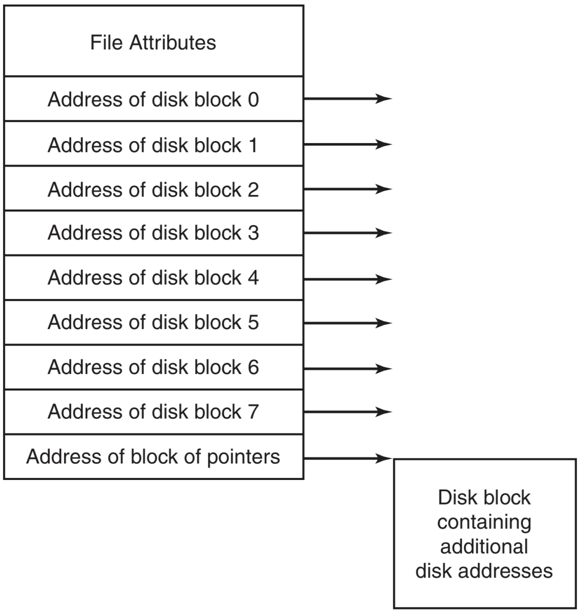


Figure 8.8: I-nodes for files on a disk.

The figure above shows an example of some i-nodes stored for files on a disk.

Given the i-node, it is then possible to find all the blocks of the file.

Evaluation

Advantages	Disadvantages
More efficient memory usage than FAT as only the i-node of the file must be present in main memory and only when the corresponding file is opened.	**Files may grow beyond the limit** of the fixed number of addresses in each i-node. However, this can be mitigated by reserving the last disk address for the address of a block containing more disk-block addresses. This can be further improved, as shown in Unix in the section below.
	List disk addresses must point to an address block instead of a data block.

I-nodes in Unix

In Unix, the i-node contains a number of **direct pointers** to disk blocks. There are typically ten direct pointers.

In addition to the direct pointers, there are three **indirect pointers** that point to further address blocks, which eventually lead to a disk data block.

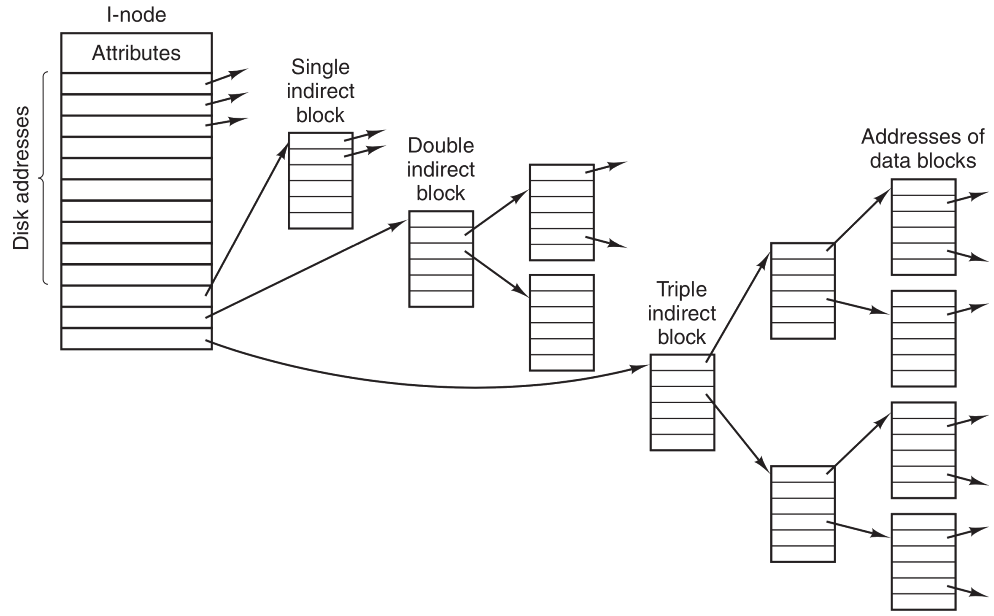


Figure 8.9: An i-node in a Unix-based OS.

As shown in the figure above, the indirect pointer includes:

- a single level of indirections;
- a double indirect pointer; and
- a triple indirect pointer.

These enable the file blocks to be located for a file, without restricting files to a fixed number of addresses.

8.3 Implementation of directories/folders

Directory entries

To open a file, the path name is used to locate its directory entry.

The directory entry provides a mapping from a file name or file descriptor to the disk blocks that contain the data.

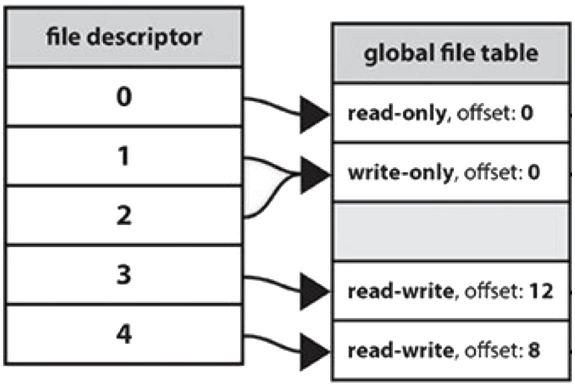


Figure 8.10: File descriptor table.

As shown in the figure above, the file descriptor table contains descriptors for each file under that directory which correspond to specific entries on the global file table.

The directory entry contains all the information needed to find the disk blocks for a given file:

- in contiguous allocation, the directory entry contains the addresses of the entire file;
- in linked list allocations, the directory entry contains the first disk block address; and
- in an i-node implementation, the directory entry contains the file's respective i-node number.

games	attributes
mail	attributes
news	attributes
work	attributes

Figure 8.11: A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry.

The figure above shows the state of the file descriptor table if contiguous allocation or linked list allocations were being used.

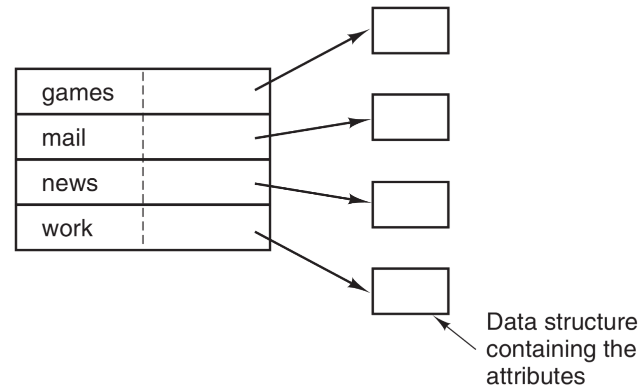


Figure 8.12: A directory in which each entry just refers to an i-node.

The figure above shows the state of the file descriptor table if an i-node implementation was being used.

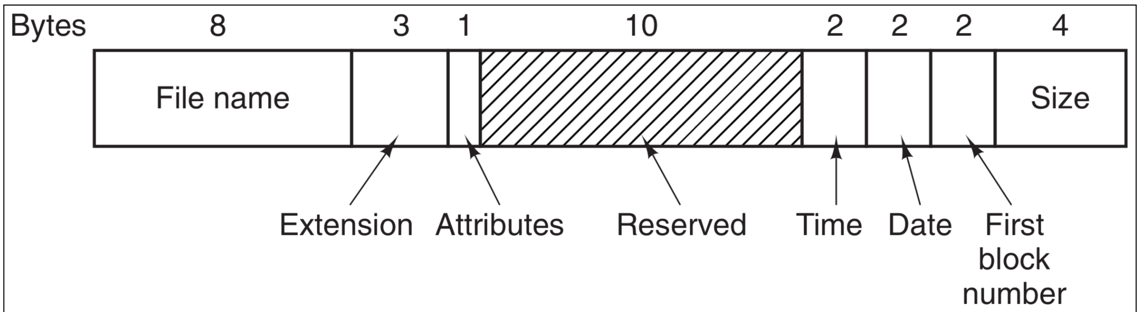


Figure 8.13: MS-DOS directory entry.

As shown in the figure above, in MS-DOS, the directory entry contains the attributes.

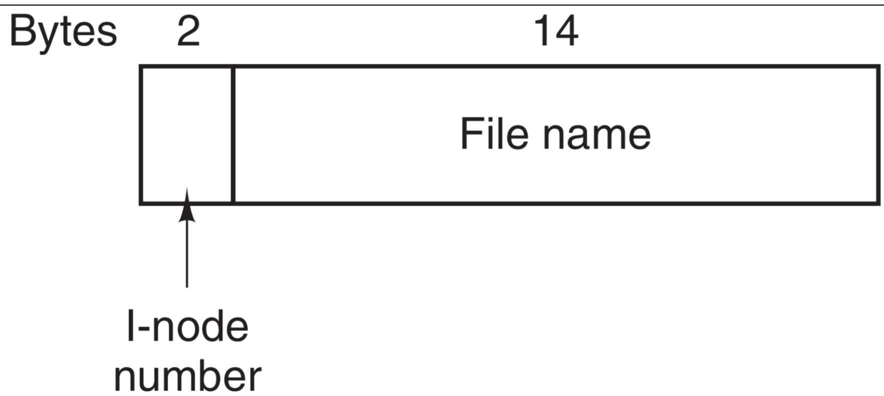


Figure 8.14: Unix directory entry.

As shown in the figure above, in Unix-based OSs, the directory entry contains an i-node number and a file name, where:

- the attributes are stored in the i-node; and
- all i-nodes have a fixed location on the disk, so locating an i-node is simple and fast.

8.4 Locating files using i-nodes

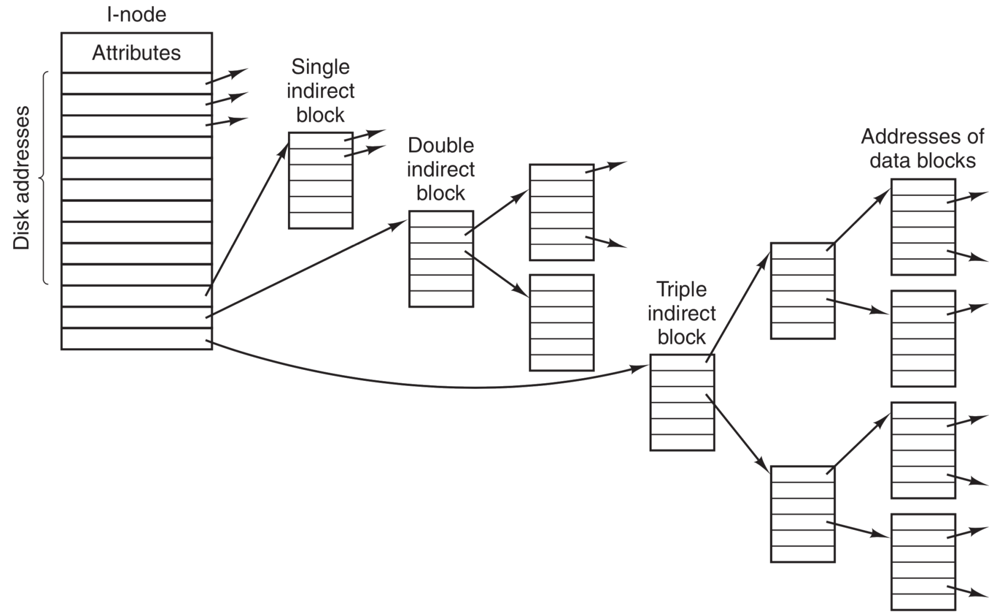
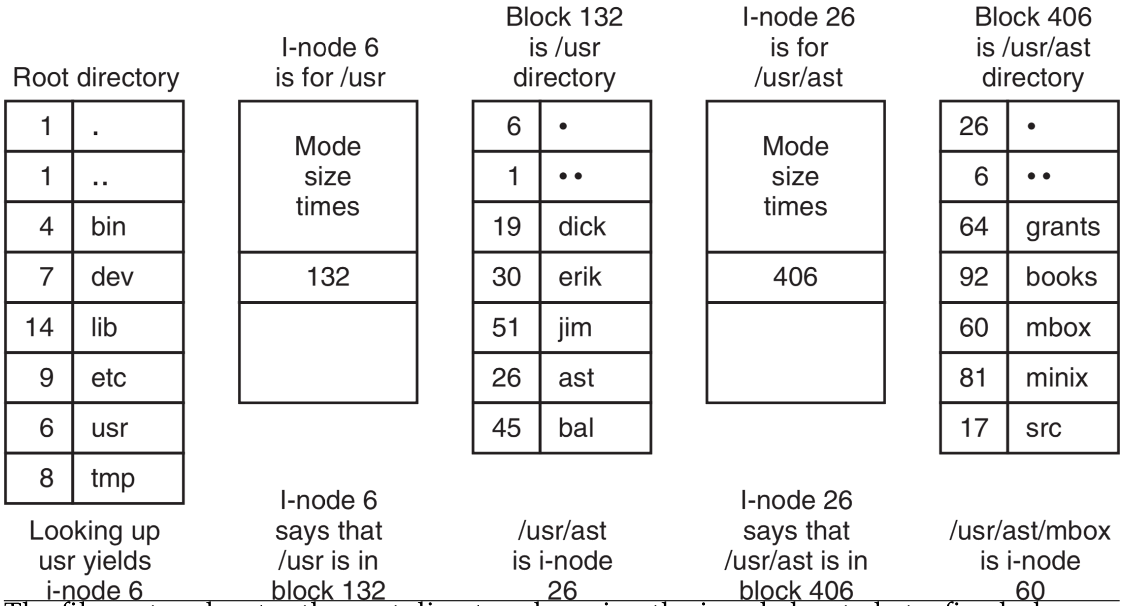


Figure 8.15: An i-node in a Unix-based OS.

As seen before, the figure above shows that the indirect pointer includes:

- a single level of indirections;
- a double indirect pointer; and
- a triple indirect pointer.

These enable the file blocks to be located for a file, without restricting files to a fixed number of addresses, as shown below.



The file system locates the root directory by using the i-node located at a fixed place on the disk.

↓

The file system looks up the first component of the path, `usr`, in the root directory to find the i-node number of the `/usr` file.

↓

From this i-node, the file system locates the directory for `/usr` and looks up the next component of the path, `ast`, inside the i-node.

↓

When the entry for `ast` is found, it has the i-node number for the directory `/usr/ast` and can locate its respective i-node.

↓

From this i-node, the file system can locate the directory itself and lookup the `mbox` file.

↓

The i-node for the `mbox` file is read in to main memory and is kept there until the file is closed.

Figure 8.16: Process for finding file blocks for `/usr/ast/mbox` using i-nodes in a Unix-based OS.

The process for accessing a relative path name is identical, except that the search begins from the current working directory.

Using this process, the i-node of the `mbox` file is kept in main memory until the file is closed. However, locating a file in other hierarchical file systems is similar, lookup is completed component by component.

8.5 Determining block size

All allocation methods require the disk to be split in to fixed-size blocks.

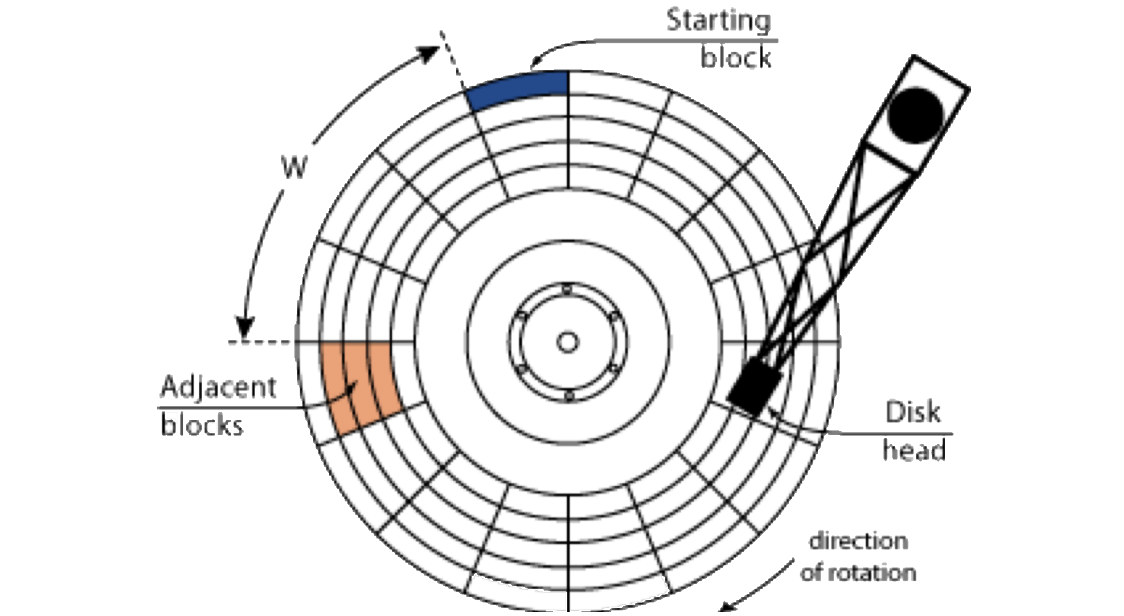


Figure 8.17: Blocks on HDD.

Nearly all modern operating systems use the fixed-sized blocks as shown in the figure above.

There is a similar trade off with block size as there was with page size in memory management. As such, tuning the block size is tricky as:

- **a smaller block size may lead to increased overhead**, as a file may occupy several blocks and therefore there will be a longer access time since more blocks have to be located; and
- **a larger block size may lead to wasted space** as a small file of 1KB would occupy a big chunk, such as 32KB, of the disk block.

Typical block sizes include: 512 Bytes, 1KB and 2KB.

8.6 Tracking free space

Linked list method

The **linked list method** is a way of tracking free space on a disk where some of the free blocks are used to hold free disk numbers.

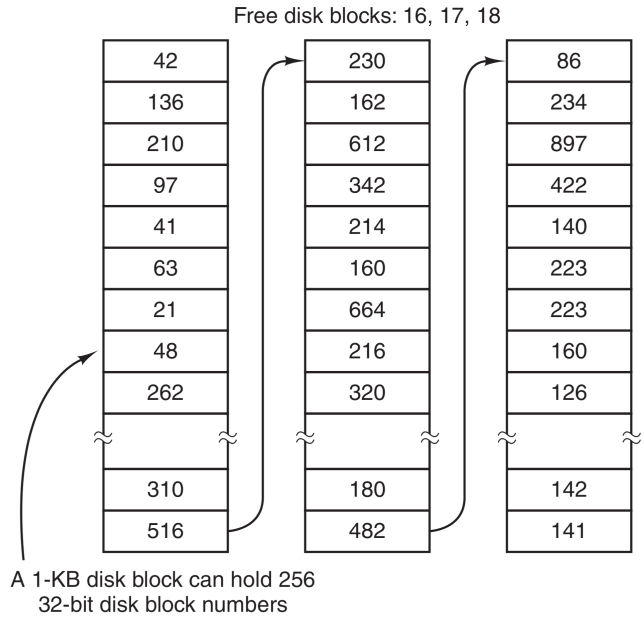


Figure 8.18: Tracking free space using a linked list.

As shown in the figure above, each block holds as many free disk block numbers as possible. The blocks are linked together resulting in a linked list of free blocks, by using one slot in each list for a pointer to the next block.

When considering a 1TB disk, which has about 1 billion disk blocks, storing all these addresses at 32 bit per block requires about 4 million blocks.

Generally, free blocks are used to hold the free list, so the storage is essentially free.

Bitmap method

The **bitmap method** is another way of tracking free space on a disk where a bit map is stored that records whether particular blocks are free or not.

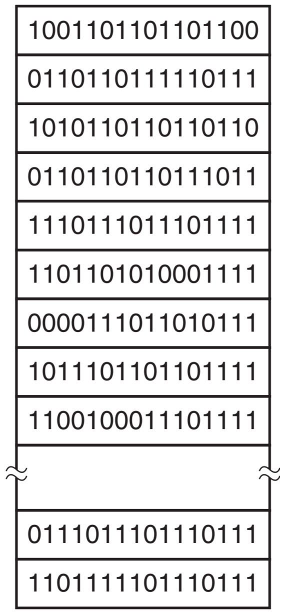


Figure 8.19: Tracking free space using a bitmap.

As shown in the figure above, the bitmap records:

- a zero (0) if the disk block is used; and
- a one (1) if the disk block is free.

A disk with n blocks requires a bitmap with n bits.

When considering a 1TB disk, which requires around 130,000 1KB blocks to store the information, it is clear to see that a bitmap occupies less space than a linked list. Only if the disk is nearly full (i.e. has few free blocks) will the linked list method require fewer blocks than the bitmap.

Comparison of linked list method and bitmap method

Criteria	Linked list method	Bitmap method
Occupied space	32 bits per block.	1 bit per block.
Number of blocks	The number of blocks required to track free space becomes less as the drive gets fuller.	The number of blocks required to track free space is constant.

8.7 Consistency

A file system may be in any one of the following states:

- consistent;
- missing block;
- duplicate block in free list; or
- duplicate data block.

Many file systems read blocks, modify their contents and write them out later.

If the system crashes before all the modified blocks have been written out, the file system can be left in an inconsistent state. This problem is especially critical if some of the blocks that have not been written out are:

- i-node blocks;
- directory blocks; or
- blocks containing the linked list.

To deal with inconsistent file systems, most OSs provide a utility program that checks for file system consistency. For example:

- Windows provides `chkdsk`; and
- Unix-based OSs provide `fsck`.

These utilities can be run whenever the system is booted, especially after a crash.

8.8 Journaling

Definition

A **journaling file system** is a file system that keeps track of changes not yet committed to the file system's main part by recording the intentions of such changes in a data structure known as a ”journal”, which is usually a circular log.

Why is journaling required?

A single file operation may involve multiple actual writes to the disk. For example, in a Unix-based OS, removing a file requires the following three operations to be performed:

- remove the file from its directory;
- release the i-node to the pool of free i-nodes; and
- return all the disk blocks to the pool of free disk blocks.

This process is similar in Windows.

In the event that a system crashes between any one of the three tasks above, the i-nodes and file blocks will not be accessible from any file or available for reassignment. This would require time consuming requires by using the consistency utilities.

How it works

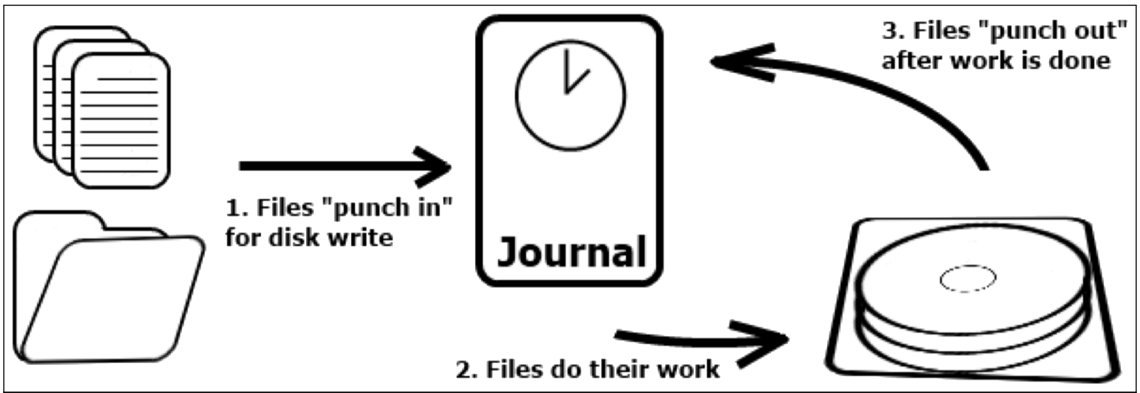


Figure 8.20: Journaling.

The journaling file system uses a special disk area to make a log entry. The log entry lists the actions to be completed.

In the event of failure, the log is used to bring the disk back in to a consistent state by completing all pending actions.

Log entries are erased once the operations complete successfully.

8.9 Backups

Definition

A **backup**, or **data backup**, refers to the process of copying data on the disk in to an archive file so that it may be used to restore the original after a data loss event.

Why are backups used?

Backups help to solve potential problems by allowing:

- recovery from disasters; and
- recovery from mistakes.

Issues

The process for **backing up is slow** as all files must be copied. However, this can be mitigated by performing incremental backups where only those files that have been changed since the previous backup are copied. This reduces the time taken for a backup, apart from the initial backup.

Backups occupy large amounts of space, especially if the stored data on the disk is larger. This can be mitigated by compressing the backup data. However, a corrupted part of the compressed data can render an entire backup unreadable.

The **system must be inactive** while a backup is being performed. This makes it difficult to continue using the system as the file system is actively being backed up.

Must ensure **physical security of backup media** as physical corruption of backups can occur on CD-ROMS, disks and magnetic tapes etc.

8.10 Drives and mount points

In Windows, multiple drives appear with distinct drive letters such as `C`, `D`, `E` etc. When a new drive is attached, it is assigned a unique letter.

In Unix-based OSs, a uniform file system is presented with a single root where new drives usually appear in the `/media/` directory.

8.11 RAID

Definition

Redundant Array of Inexpensive Disks (RAID) is a data storage virtualisation technology that combines multiple physical disk drive components into one or more logical units for the purposes of data redundancy, performance improvement, or both.

RAID provides another layer of abstraction as data is distributed across several physical disks which appear to be a single logical disk.

Why is RAID required?

Disks remain the bottleneck in computer systems as their access speeds are much slower than the CPU's clock speed. As such, RAID is often used for speed enhancements along with other benefits, as seen below.

Types of RAID

There are multiple types of RAID that are suited to different use cases.

Striping, mirroring and parity

Different types of RAID use different techniques including:

- striping – distributing the data across several disks in a way which gives improved speed and full capacity;
- mirroring – uses more than one disk which store the same data; and
- parity – uses an additional disk to provide fault tolerance where a single data bit is added to the end of a data block to ensure the number of bits in the message is either odd or even.

Striping provides no security on its own but gives the best speed and capacity.

Mirroring degrades speed and capacity but provides security.

RAID 0

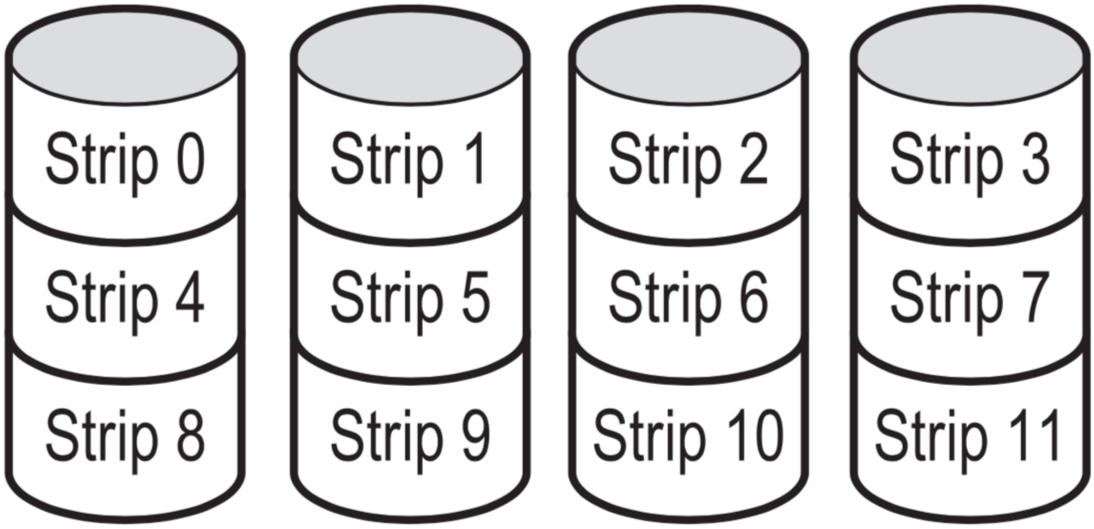


Figure 8.21: RAID 0.

Striping	YES
Mirroring	NO
Parity	NO

The capacity of a RAID 0 volume is the sum of the capacities of the disks in the set. However, striping distributes the contents of each file among all disks in the set and therefore the failure of any disk causes all files to be lost.

RAID 1

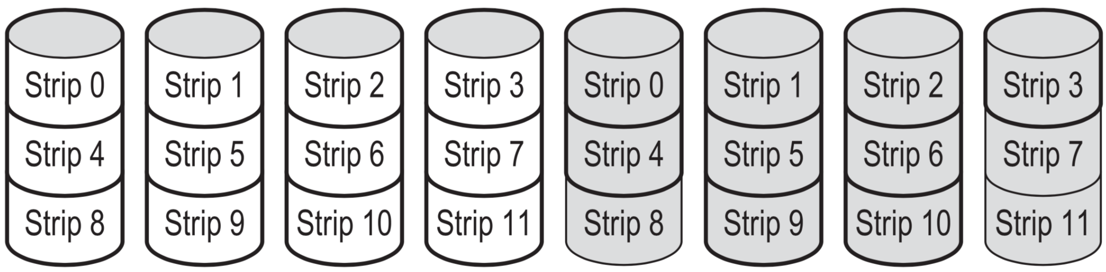


Figure 8.22: RAID 1 (backup and parity drives are shaded).

Striping	NO
Mirroring	YES
Parity	NO

Data is written identically to two drives, thereby producing a ”mirrored set” of drives. Therefore, any read request can be serviced by any drive in the set.

If a request is broadcast to every drive in the set, it can be serviced by the drive that accesses the data first (depending on its seek time and rotational latency), improving performance.

Sustained read throughput, if the controller or software is optimised for it, approaches the sum of throughputs of every drive in the set, just as for RAID 0.

Actual read throughput of most RAID 1 implementations is slower than the fastest drive.

Write throughput is always slower because every drive must be updated, and the slowest drive limits the write performance.

The array continues to operate as long as at least one drive is functioning.

RAID 1+0

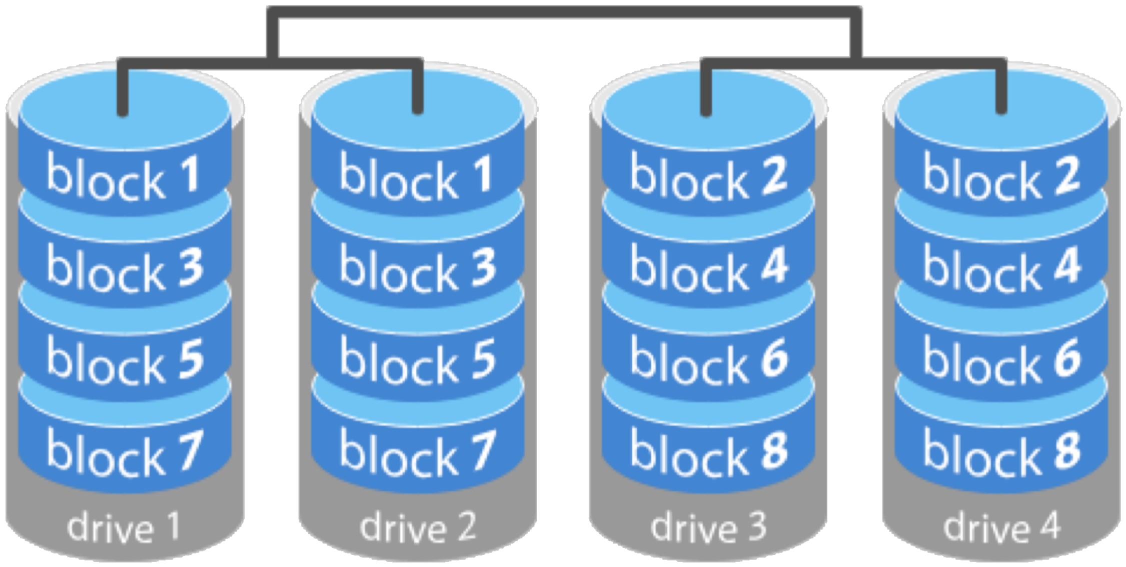


Figure 8.23: RAID 1+0.

Striping	YES
Mirroring	YES
Parity	NO

RAID 1+0 takes advantage of both RAID 0's striping and RAID 1's mirroring. As such, if striping is applied to two disks, an additional two disks are required to mirror the striped drives.

9. Input/Output (I/O)

9.1 Input/Output (I/O) devices

Defintion

An **Input/Output (I/O) device** is a hardware device that has the ability to accept inputted, outputted or other processed data. It also can acquire respective media data as input sent to a computer or send computer data to storage media as storage output.

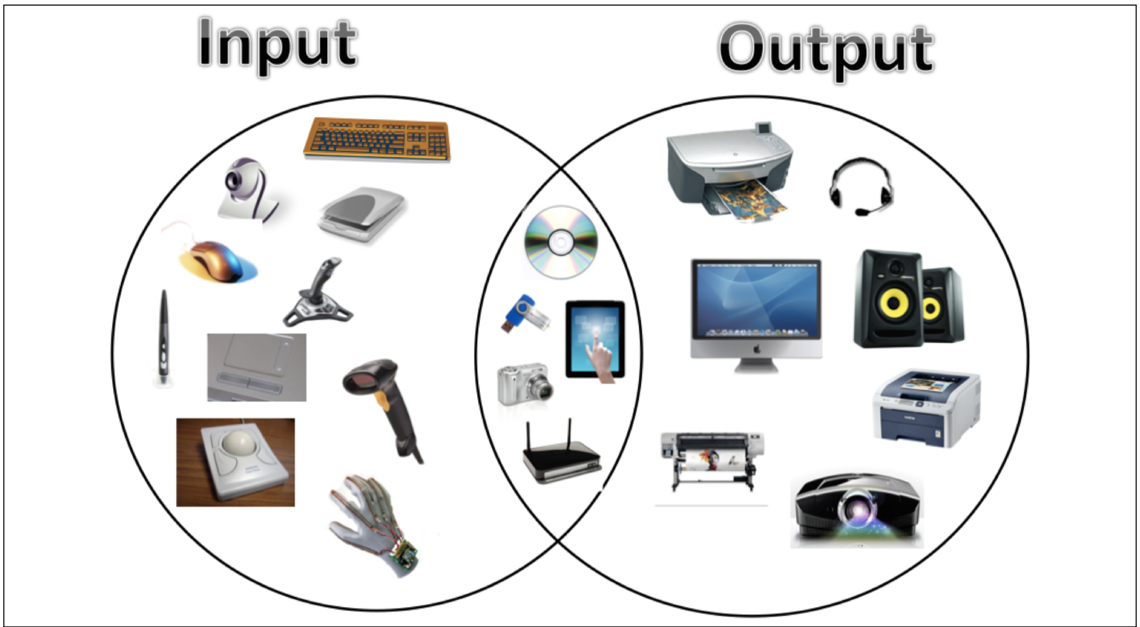


Figure 9.1: Venn diagram of I/O devices.

An I/O device can be responsible for input and output as shown in the cross section in the Venn diagram above.

I/O devices typically consists of:

- the **device proper** – an electro-mechanical component generally located outside the computer, such as a peripheral; and
- the **device controller** – an electrical component often in the form of a chip on the mother board or a card that can be inserted in to an expansion slot, such as on the Peripheral Component Interconnect Express (PCIe) slot.

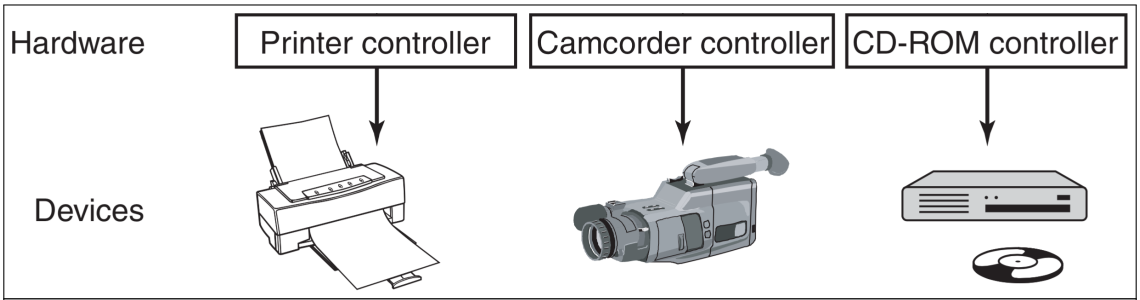


Figure 9.2: I/O device proper connection to device controller.

As shown in the figure above, the I/O device proper is connected using a cable to the device controller for communication to and from the outer world.

I/O controllers

Each I/O controller has a few registers that are used for communicating with the CPU:

- write operation – the OS can command the device to deliver/accept data or switch on or off itself, or perform any other action; and
- read operation – the OS can determine the device's current state, such as whether it is prepared to accept a new command.

Typical characteristics and specifications

Device	**Data rate**
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner at 300dpi	1 MB/sec
Digital camcorder	3.5 MB/sec
4x Blu-ray disc	18 MB/sec
802.11n Wireless	37.5 MB/sec
USB 2.0	60 MB/sec
FireWire 800	100 MB/sec
Gigabit Ethernet	125 MB/sec
SATA 3 disk drive	600 MB/sec
USB 3.0	625 MB/sec
SCSI Ultra 5 bus	640 MB/sec

Single-lane PCIe 3.0 bus	985 MB/sec
Thunderbolt 2 bus	2.5 GB/sec
SONET OC-786 network	5 GB/sec

9.2 Input/Output (I/O) system

Definition

The **Input/Output (I/O) system** includes all the agents that are involved in I/O operations, including:

- the device(s) themselves;
- the device controller(s);
- the OS mediating the operations; and
- the user process requesting the operations.

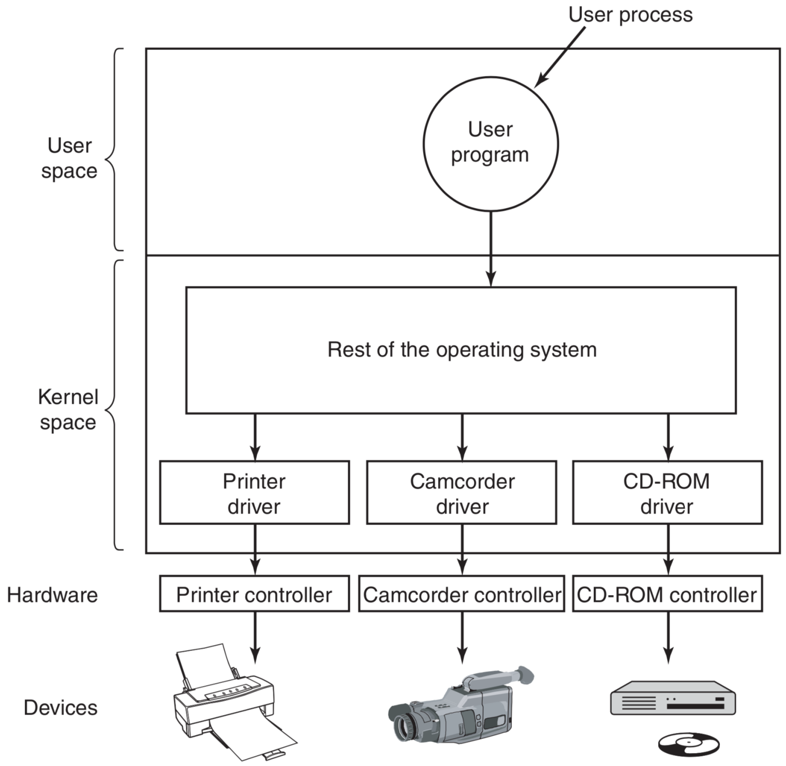


Figure 9.3: I/O system layers.

The figure above shows the several layers on which the I/O system depends, with the agents residing in each layer.

Structure

User program

The user program makes an I/O request in a high level language.

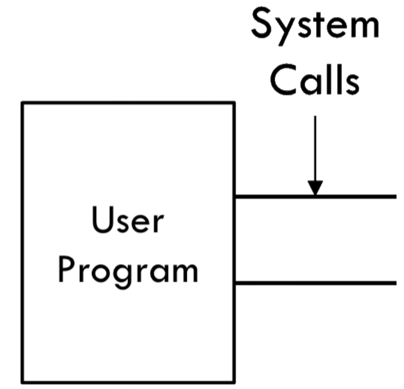


Figure 9.4: User program request translation to system calls.

As shown in the figure above, the request is translated and relayed by system calls.

I/O control system

The I/O control system is a piece of device-independent software that deals with I/O-related system calls.

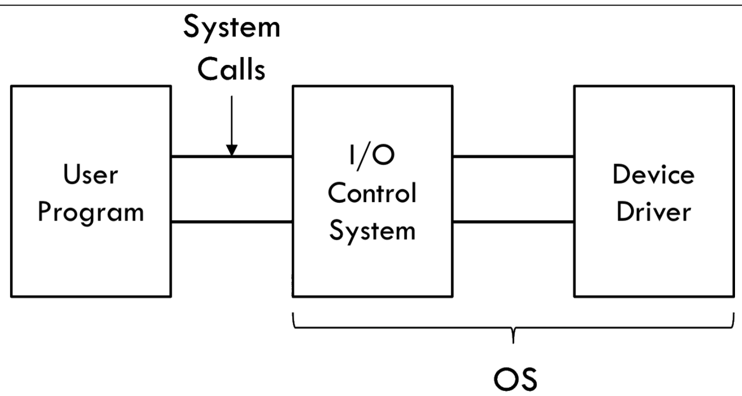


Figure 9.5: I/O control system and device driver.

As shown in the figure above, the I/O control system is a platform for installed device drivers to communicate.

Device driver

A device driver is a software module that manages the communication with a specific I/O device. This is written by the manufacturer of the piece of hardware and is specifically designed for that piece of hardware.

Specific device drivers are provided by hardware manufactures on installation discs or Internet download links. Hardware manufacturers must create device drivers for each operating system for which the hardware must be compatible.

Generic device drivers, or **non-specific device drivers**, are often provided in operating systems. These may not be as good as specific device drivers because they may not be supported by the device or provide all of the necessary features to allow the hardware device to operate properly.

The exact detail of which device driver is required by the operating system is stored in a file and the drivers for the hardware connected are loaded when the computer system is booted.

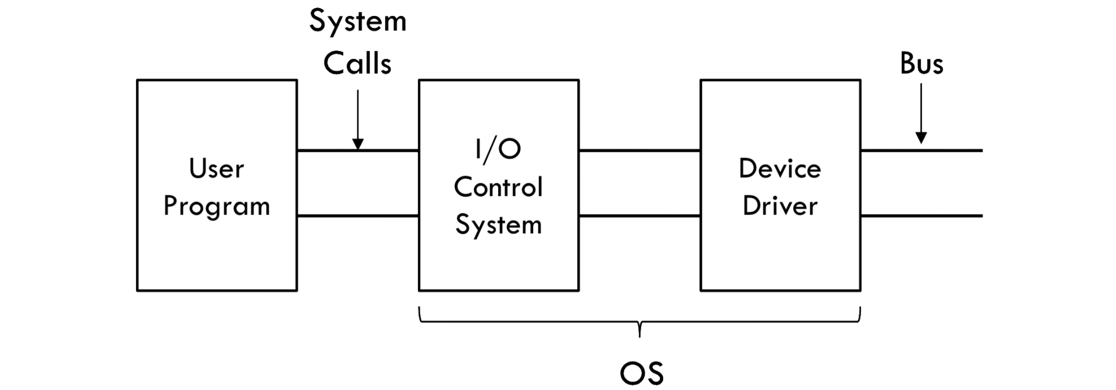


Figure 9.6: Device driver and PCIe bus.

As shown in the figure above, the device driver converts the logical I/O requests in to specific commands directed to that device. This is communicated using the bus, to which the PCIe bus is connected.

A problem arises as many different manufactures create device drivers for their devices.

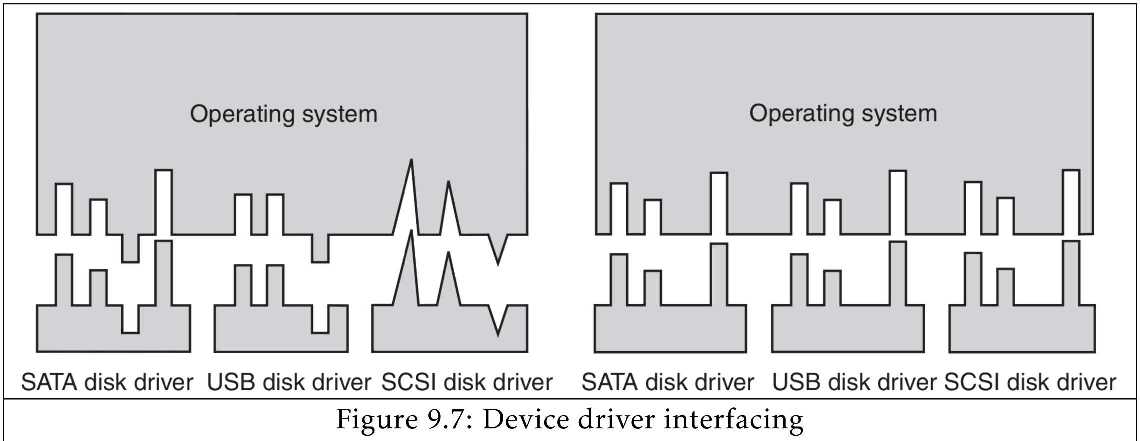


Figure 9.7: Device driver interfacing

In the figure above, the left image shows a case where a standard driver interface is not used. This means that the operating system would need to be modified to be compatible with each device driver after a manufacturer develops a new device. This would yield a high amount of maintenance and code required in the operating system.

In the figure above the right image shows a case where a standard driver interface is used. This allows the operating system to expose an application programming interface (API) to device manufacturers. The device manufacturers must adhere to the API and therefore, all drivers follow this API definition so the operating system code does not have to be modified.

Device controller

The device controller is a hardware unit attached to the bus.

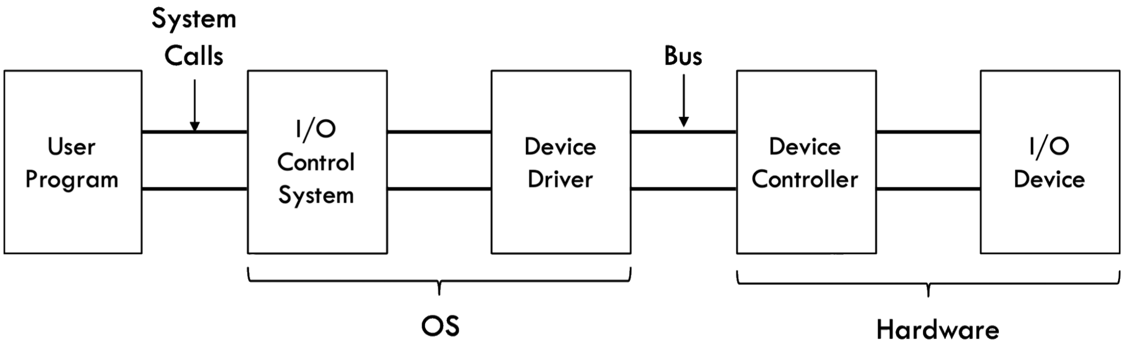


Figure 9.8: Device controller and I/O device.

As shown in the figure above, the device controller provides and interface between the computer system and the I/O device using electrical signals.

I/O device

The I/O device may be:

- a **block device** that transfers data in groups of characters; or
- a **character device** that transfers data one character at a time.

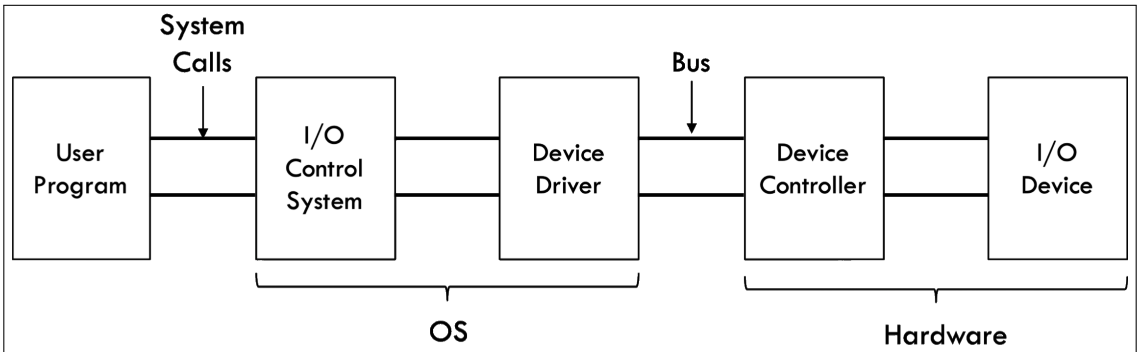


Figure 9.9: Complete I/O system.

The figure above shows the complete I/O system.

Each I/O device has a **device descriptor** containing information about the device. For example:

- the device identification;
- the instructions that operate the device;
- the current status; and
- the current user process (if any).

The information in the device descriptor is used by the I/O controller.

Objectives of the I/O system

The I/O system aims to be **efficient** as:

- I/O devices are much slower than the CPU; and
- I/O devices should be used efficiently as to avoid blocking the CPU.

In addition, the I/O system aims to allow **independence and uniformity** as:

- I/O devices are quite complex and diverse;
- the user should not be concerned with the details of I/O devices;
- the user should be able to treat I/O devices in a uniform way; and
- the user should deal with the virtual devices presented by the OS, rather than the physical devices.

Finally, the I/O system should provide **error handling** that should be:

- handled as close to the hardware as possible; and
- transparent to the user.

I/O interrupts

Definition

An **interrupt** is a signal sent to the processor indicating that an event caused by hardware or software requires immediate attention.

In I/O, interrupts are used as a way of controlling I/O activity in which a peripheral or terminal that needs to make or receive a data transfer.

How it works

Interrupts are very important and are not trivial to handle.

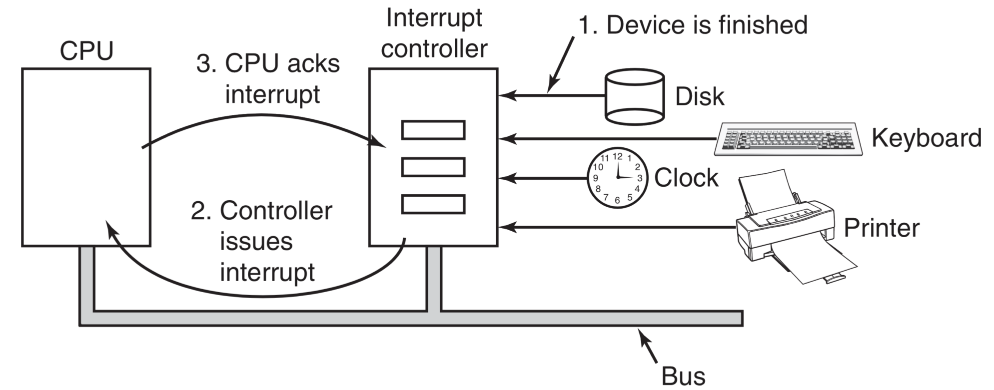


Figure 9.10: An I/O interrupt.

The figure above shows that:

- an I/O interrupt is generated when a particular I/O device is finished with a current process;
- the interrupt controller issues the interrupt to the CPU by asserting a signal on its assigned bus line;
- the CPU's interrupt controller chip detects the signal from the interrupt controller and performs some subsequent actions based on the conditions; and
- the CPU acknowledges the interrupt to the interrupt controller.

The connections between the devices and the controller actually use interrupt lines on the bus rather than dedicated wires.

If no other interrupts are pending, the interrupt controller handles the interrupt immediately. However, if another interrupt is in progress, or another device has made a simultaneous request on a higher-priority interrupt request line on the bus, the device is just ignored for the moment. In this case it continues to assert an interrupt signal on the bus until it is serviced by the CPU.

The CPU stops the current process.

↓

Information including:

- the contents of the program counter (PC);
- registers that are not already saved by the interrupt hardware are saved; and
- the return address.

is saved by pushing to the stack, in order to preserve the state of the current process.

↓

The I/O controller informs the CPU on which interrupt service procedure to use based on the device identification information obtained from the device descriptor.

↓

The number on the address lines is used as an index in to a table called the interrupt vector to fetch a new program counter.

↓

This program counter points to the start of the corresponding interrupt service procedure.

↓

The CPU sets up context for the interrupt service procedure.

↓

The CPU sets up stack for the interrupt service procedure.

↓

The CPU uses the program counter's value to begin the interrupt service procedure.

↓

Shortly after it starts running, the interrupt service procedure acknowledges the interrupt by writing a certain value to one of the interrupt controller's I/O ports. This acknowledgement tells the controller that it is free to issue another interrupt.

↓

The CPU copies the registers from where they were saved to the process table.

↓

The interrupt service procedure extracts information from the device control registers.

↓

The CPU chooses which process to run next.

↓

The CPU sets up the Memory Management Unit (MMU) context for the next process to run.

↓

The CPU loads the new process' registers.

↓

The CPU starts running the new process.

Figure 9.11: Process for handling an I/O interrupt once received by the CPU.

The figure above shows the process for handling an I/O interrupt once it has been received by the the CPU.

Direct Memory Access (DMA)

Definition

Direct Memory Access (DMA) is an alternative method of handling I/O operations that allows an I/O device to send or receive data directly to or from the main memory, bypassing the CPU to speed up memory operations.

How it works

The computer system and peripherals communicate directly with each other using memory buses, removing intervention of the CPU.

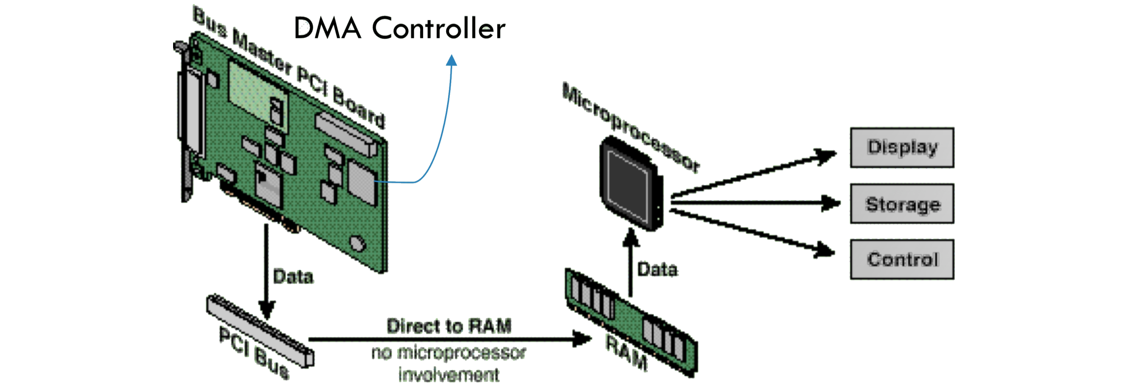


Figure 9.12: DMA Transfer.

During DMA, the CPU is idle and has no control over the memory buses.

The DMA controller takes over the buses to manager the transfer directly between the I/O devices and the memory unit.

Benefits over interrupts

When considering a case where a user process requests sending an eight character string "ABCDEFGH" to a printer peripheral, an interrupt would be caused for each character sent, as shown in the figure below.

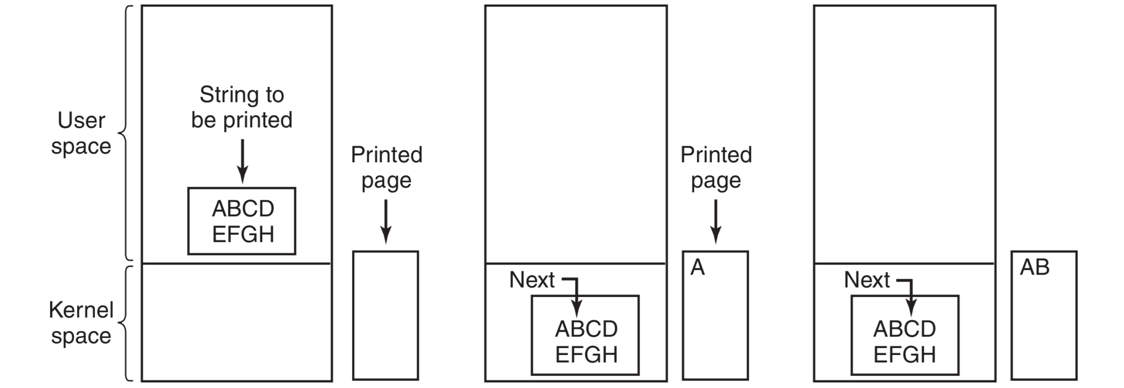


Figure 9.13: Process for printing a string using interrupts.

This is not very efficient as the data transfer between the device and memory unit is limited by the speed of the CPU. Whereas, DMA allows **burst transfer** where a block sequence of words in memory is transferred in a continuous burst by the DMA controller.

Buffering

Definition

A **buffer** is an interim memory area that holds data in transit between process's working RAM and the I/O device.

Why is buffering required?

If a process does repeated I/O, then it will be repeatedly waiting for the I/O to complete. For example, in a a disk read operation:

- the process issues some system calls;
- the process waits to read the first block of data to memory; and
- the process performs some processing on the data.

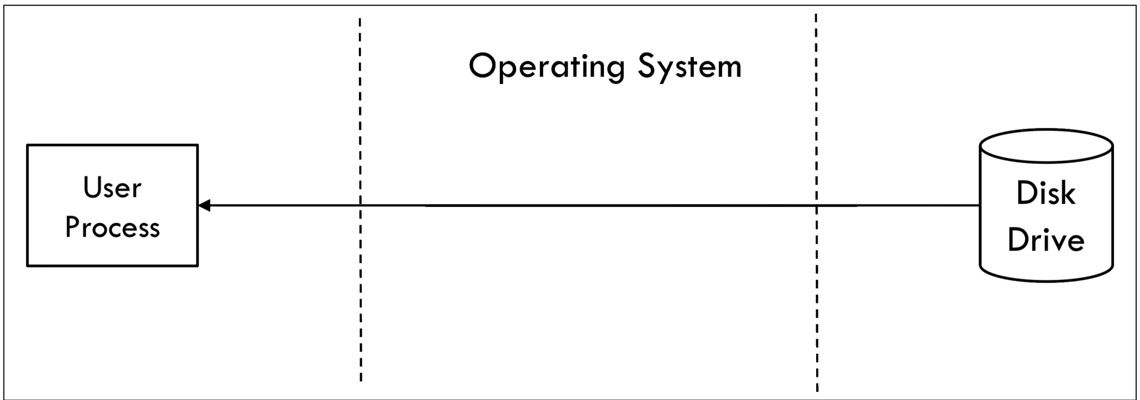


Figure 9.14: Read-processing cycle.

The read-processing cycle shown in the figure above presents some issues as:

- the CPU is idle for most of the time as it is waiting for data transfer to finish and will issue interrupts; and
- the disk drive is not running continuously.

This shows that with no buffering, neither the CPU or disk drive are being used to their maximum capabilities as allowing a process to run many times for short runs is inefficient.

Input and output buffering

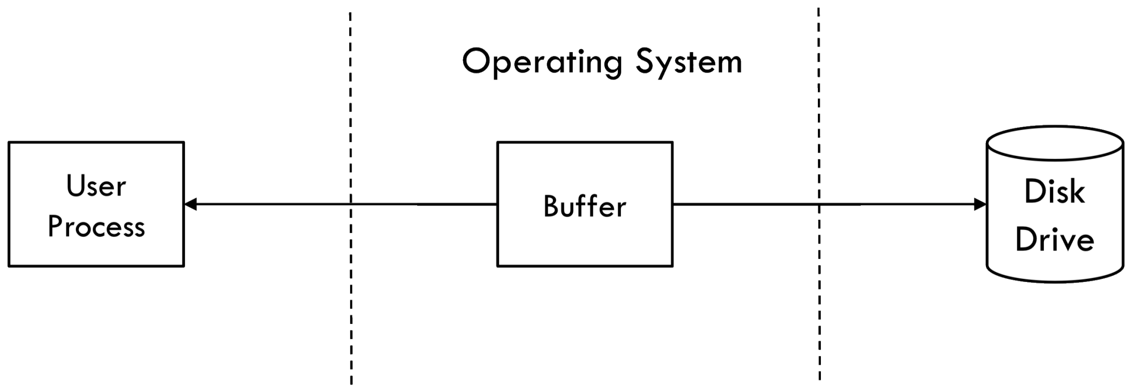


Figure 9.15: Read-processing cycle with a single buffer.

As shown in the figure above, the buffer holds the data in transit between process's working RAM and the I/O device in order to:

- cope with the speed mismatch between the devices as the I/O devices are generally slower than the CPU; and
- perform a checksum validation to ensure that the data is valid.

When a user process is attempting to obtain input data from an I/O device, **input buffering** is used where:

- input data is transferred to an area of memory called the input buffer;
- the process gets data from the input buffer;
- the OS fills the input buffer when it is empty; and
- the process is blocked if it attempts to get data from an empty input buffer.

When a user process is attempting to output data to an I/O device, **output buffering** is used where:

- output data is transferred to an area of memory called the output buffer;
- the OS empties the output buffer when it is full; and
- the process is blocks if it attempts to output before the system has emptied the output buffer.

Single buffering

When using single buffering, blocks are read in to the buffer and then moved to the user's work area.

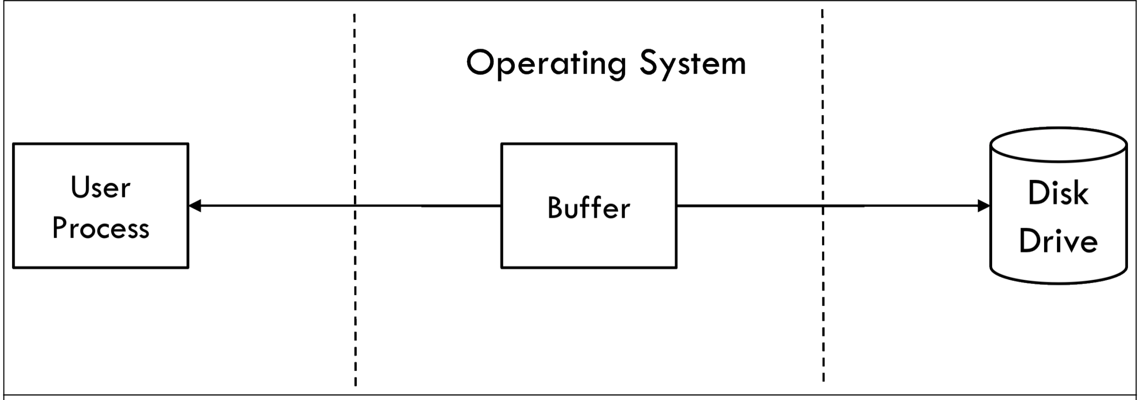


Figure 9.16: Read-processing cycle with a single buffer.

When blocks are moved from the buffer, processing can be completed in parallel with the transfer of the next blocks.

Advantages	Disadvantages
Interrupts are reduced as the CPU is not always waiting for the data transfer to finish.	**The I/O device is idle for some time** as the CPU must wait for the buffer to be filled.

Double buffering

When using double buffering:

- the process reads/writes to one buffer; while
- the OS empties/fills the other buffer.

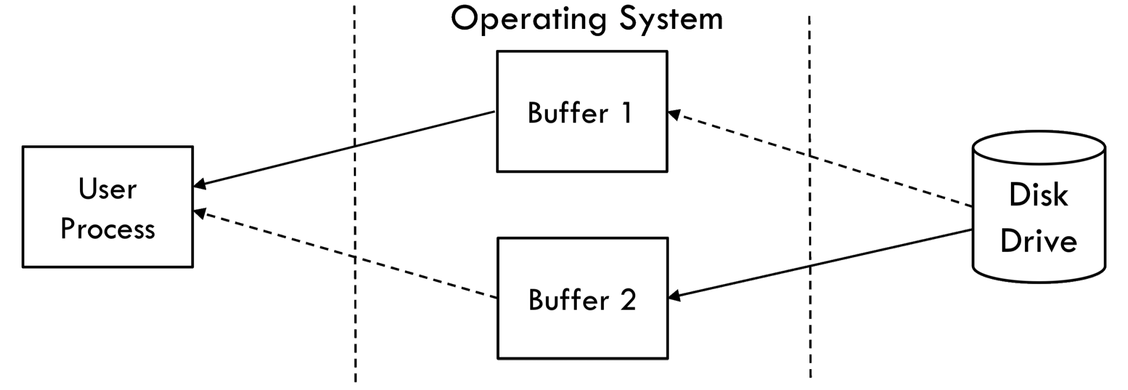


Figure 9.17: Read-processing cycle with double buffers.

Advantages	Disadvantages
Transferring data may be continuous if the processing time is less than the transferring time.	**The I/O device may still be idle for some time** if the transferring time is less than the processing time.
Increases utilisation of the I/O device as it is idle for less time than in single buffering.	

Multiple buffering

In order to achieve continuous I/O, double buffering can be extended to multiple buffering.

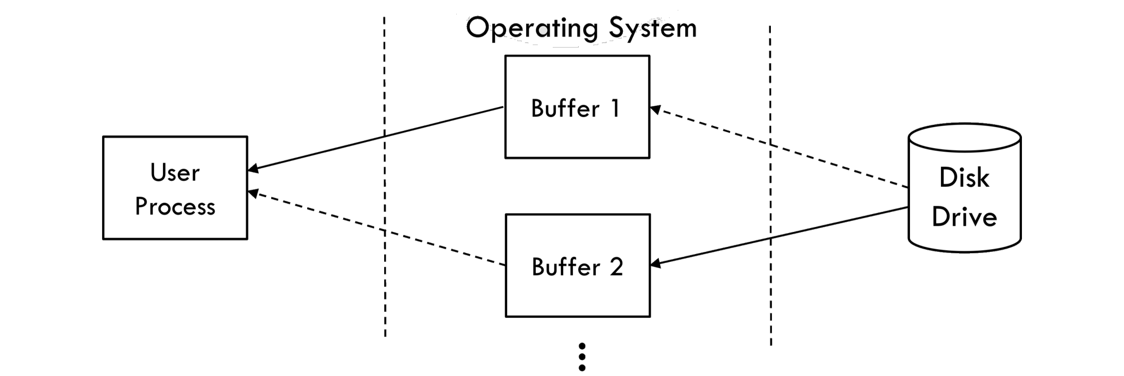


Figure 9.18: Read-processing cycle with multiple buffers.

As shown in the figure above, multiple buffering can be implemented with any number, n, buffers in order to achieve continuous I/O.

Advantages	Disadvantages
Can achieve continuous I/O if enough buffers are used.	**Adds more complexity and overhead** as more buffers must be managed.
Maximum utilisation of the I/O device as there is continuous I/O.	**Increased memory usage** as more buffers are stored in memory.

Overview of the I/O system

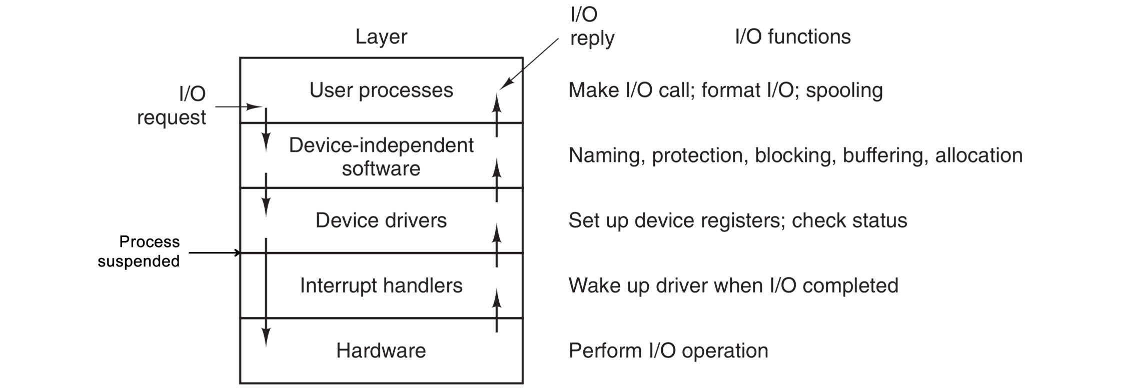


Figure 9.19: I/O system layers.

The figure above shows an overview of the I/O system which contains multiple layers where each layer has a main function.

10. Introduction to Java

10.1 History and philosophy

Java is a high-level programming language, developed by James Gosling in the early 90s, that is:

- designed with a C/C++ syntax style;
- strictly object-oriented; and
- highly portable (slogan: *"Write Once, Run Anywhere"*).

10.2 Code

Machine code

Machine code is the language that is actually used by the processor, and differs between processors.

```
1110001100000000
0101011011100000
0110100001000000
0000100000001000
0001011011000100
0001001001100001
0110100001000000
```

Figure 10.1: Example machine code.

Assembly language

Assembly language is a low-level language that uses mnemonics to refer to instructions for the CPU to process. It requires an assembler to convert the assembly instructions into machine code.

```
CMP AX,97
JL DONE
CMP AX,122
JG DONE
SUB AX,32
```

Figure 10.2: Example assembly language.

The code in assembly language has a one-to-one relationship with the corresponding machine code. The code in assembly language has a many-to-one relationship with the corresponding high-level language code.

High-level languages

A **high-level programming language** is a programming language with strong abstraction from the details of the computer.

```
int x, y, sum;
x = 2;
y = 1;
sum = x + y;
return sum;
```

Figure 10.3: Example code written in a high-level language.

High-level languages may:

- use natural language elements;
- be easier to use; and/or
- automate or hide entirely significant areas of computing systems, such as memory management.

These attributes of high-level languages make the process of developing a program simpler and more understandable than when using a lower-level language.

The amount of abstraction provided defines how "high-level" a programming language is.

High-level languages also provide **portability** as the same code can be ran on different CPUs. This is achieved by using a compiler.

A **compiler** is a program that translates each line of high-level code into machine code and then executes the entire code.

10.3 Java Virtual Machine (JVM)

Definition

A **Java Virtual Machine (JVM)** is a virtual machine that enables a computer system to run Java programs as well as programs written in other languages that are also compiled to Java bytecode.

How it works

All source code is written in plain text with the `.java` extension.

The compilation process generates `.class` files that contains bytecodes.

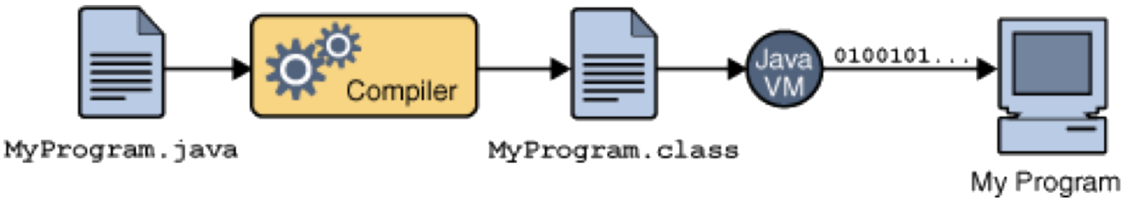


Figure 10.4: Compiler and JVM.

The bytecodes are interpreted by the JVM, which translates the bytecodes in to machine code at runtime (on the fly).

The JVM is available on many different operating systems and platforms.

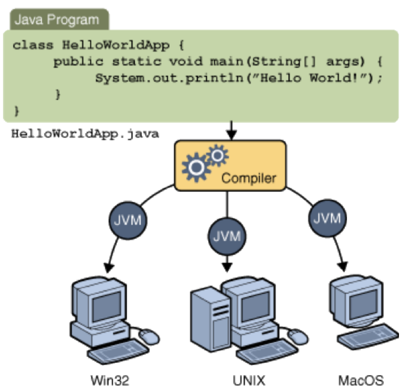


Figure 10.5: Compiler and JVM.

This means that the same `.class` is capable of running on multiple platforms.

10.4 Environment

The Java environment

The Java environment is a software-only platform that runs on top of other hardware-based platforms. It consists of two components:

- a Java Virtual Machine (JVM); and
- Java Application Programming Interface (API) libraries, containing a large collection of ready-made components.

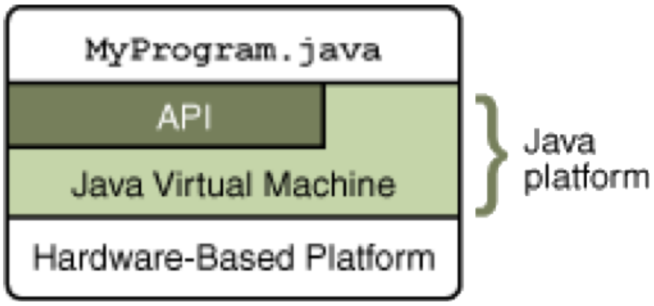


Figure 10.6: The Java environment.

The API is grouped in to libraries and classes. More information about the APIs is available on Oracle's website (`http://docs.oracle.com/javase/8/docs/api/`).

Java environment vs C++ environment

Similarities	Differences
Uses block structure with curly braces (`{ }`) for blocks and normal braces (`()`) for block statements.	**Java's compiler is more sophisticated** as there is no need for a `makefile`.
Both import libraries in a similar manner where C++ uses `#include <>`, Java uses `import <>`.	**Java does not use header files**, while C++ uses header files with the `.h` extension.
	Java does not support pointers, while C++ has inherited support for pointers from C along with other pointer types such as `shared_ptr` included in the standard library.
	The compiler for Java generates bytecode, while the compiler for C++ generates machine code.

It is important to note the **Java is slower than C++** as the bytecode is interpreted at runtime, where as in C++ the machine code can be run directly on the CPU.

Benefits of the Java environment

Benefits of the Java environment include:

- well documented;
- do more with less code;
- automatic garbage collection, meaning that there is no need to deallocate memory after allocation because it is performed automatically;
- no platform dependencies;
- supports multi-threading and inter-networking applications;
- contains a library for GUIs; and
- easy to learn for programmers already familiar with C/C++.

Compiling and executing

Manual method

Java programs can be compiled and executed by using commands on the command-line:

- compile a Java program – `javac MyProgram.java`; and
- execute a Java program – `java MyProgram`.

If a program consists of multiple classes, then compile only the part that has been modified.

Automatic method

Editors that include Integrated Development Environments (IDEs), such as NetBeans and Eclipse, offer automatic compilation and execution tools that are usually triggered by clicking a button on the GUI.

10.5 Simple Java programs

Hello world

```
class HelloWorldApp {
   public static void main(String[] args) {
      System.out.println("Hello␣World!"); //Display the message
   }
   /*implement other methods here*/
}
```

HelloWorldApp.java

From the `HelloWorldApp.java`, it can be seen that:

- single-line comments start with two forward slashes (`//`);
- multi-line comments start with a forward slash and two asterisks (`/**`) and end with an asterisk and a forward slash (`*/`) where each line is preceded by an asterisk (`*`);
- all programs require at least one class that should have the same name as the file name, such as `class HelloWorldApp { }`, with a main function that takes an argument written as `public static void main(String[] args){ }`; and
- the main function should contain statements that control the program's flow, in this case only `System.out.println("Hello world");`.

For loops

```
class HelloWorldApp {
   public static void main(String[] args) {
      for(int i = 0; i < 5; i++) {
         System.out.println("Hello␣World!");
      }
   }
}
```

HelloWorldApp2.java

The `HelloWorldApp2.java` program would print the string ”`Hello world!`” five times.

Using arguments and if statements

```
class HelloWorldApp {
   public static void main(String[] args) {
      String temp = args[0];

      if (temp.equals("1")) {
         System.out.println("Hello␣World!");
      } else if (temp.equals("2")) {
         System.out.println("Hello␣Universe!");
      } else {
         System.err.println("Error");
      }
   }
}
```

HelloWorldApp3.java

The `HelloWorldApp3.java` program shows how the argument passed in to the `main` function can be used to control program flow using a `if-else` statement.

As the argument for the main function is an array of type `String`, the `temp` variable is obtained from the first item in the array by using `temp[0]`.

This program also shows that the normal equality operation (==) used in if statements, and other control blocks, is not to be used for variables of type `String`. Instead, the `equals` method must be called on the variable with the checking criteria as a parameter.

It is also worth noting that errors are written out by using the `System.err.println` function, rather than the `System.out.println` function.

Converting String type to Integer

```
class HelloWorldApp {
   public static void main(String[] args) {
      int temp = Integer.parseInt(args[0]);

      if (temp == 1) {
         System.out.println("Hello␣World!");
      } else if (temp == 2) {
         System.out.println("Hello␣Universe!");
      } else {
         System.err.println("Error");
      }
   }
}
```

HelloWorldApp4.java

The `HelloWorldApp4.java` program is an adaptation of the `HelloWorldApp3.java` program that converts the data stored in `args[0]` from `String` type to `int` before continuing with the if statements. This allows the if statements to use the normal equality operation (==).

11. Java Language Basics

11.1 Java as an object-oriented language

Definitions

Java is a **class-based object-oriented programming (OOP)** language.

An **Object-Oriented Programming (OOP)** language is based around object, which may contain:

- data – in the form of attributes/fields; and
- code – in the form of methods/functions.

A **class** is a blueprint, i.e. the generic definition, for an object which defines its attributes and methods.

An **object** is an instance of a class.

A **method**, or behaviour, defines the functionality of a class.

An **attribute** is a piece of data associated with a class.

Creating a class

```
public class Student {
   String name;

   public void setName(String n) {
      name = n;
   }

   public String getName() {
      return name;
   }
}
```

Student class.

The code above creates the Student class. This class then be instantiated as an object elsewhere in the code.

```
class HelloWorldApp {
   public static void main(String[] args) {
      Student bob = new Student();
      bob.setName("Bob");
      System.out.println(bob.getName());
   }
}

Instantiating the Student object, as shown above, allows access to the
    public methods for the object \texttt{bob}.
```

Instantiation of a Student object.

11.2 Primitive data types

Definition

A **primitive data type** is a data type that is predefined by the language and is named by a reserved keyword.

Java's primitive data types

Overview

Java supports eight primitive data types and uses the same collection of primitive data types as C++, including:

- integer types;
- floating point types;

- Boolean type; and
- characters.

Similarly to C++, Java is a **strictly typed language** and therefore literals must be specifically cast when not using the default type.

However, there are important differences in their operation.

Integer types

Type	Description	Bounds
`byte`	An 8-bit integer.	-2^{-7} to $2^7 - 1$ (-128 to 127)
`short`	A 16-bit integer.	-2^{15} to $2^{15} - 1$ (-32768 to 32767)
`int` (default)	A 32-bit integer.	-2^{31} to $2^{31} - 1$ (-2147483648 to 2147483647)
`long`	A 64-bit integer	-2^{63} to $2^{63} - 1$ (-9.223372e+18 to 9.223372e+18-1)

```
byte eight = 8;          // valid
short sixteen = 16;      // valid
int thirtytwo = 32;      // valid
long sixtyfour = 64L;    // valid
long sixtyfour = 64;     // invalid
```

Using integer types.

Floating point types

Type	Description
`float`	An 32-bit floating point (saves memory).
`double` (default)	A 64-bit floating point.

```
float thirtytwo = 32.0f;    // valid
float thirtytwo = 32.0;     // invalid
double sixtyfour = 64.0;    // valid
double sixtyfour = 64.0d;   // valid
```

Using integer types.

Boolean type

A `Boolean` is an 8-bit value, which can only represent `true` or `false`.

```
boolean flag = 1;      // invalid
boolean flag = true;   // valid
bool flag = true;      // invalid

boolean flag = 0;       // invalid
boolean flag = false;   // valid
bool flag = false;      // invalid
```

Using the Boolean type.

Characters

A `char` is a 16-bit Unicode character. Unicode is a computing standard for encoding, representing and handling text in most writing systems.

```
char letterU = U;    // invalid
char letterJ = 'J';  // valid
char letterB = "B";  // invalid
char letterA = 'A';  // valid
char digit1 = '1';   // valid
char digit0 = '0';   // valid
```

Using the Boolean type.

Java includes some special characters that can be used as escape sequences.

Escape sequence	Meaning
\b	Backspace.
\t	Tab.
\n	New line.
\"	Double quote.
\'	Single quote.
\\	Backslash.

11.3 Strings

Definition

A `String` is a sequence of alphanumeric characters.

Importing String class

A `String` is not a primitive data type, but is instead a class that makes use of an implicit array of characters with operators as methods.

The `String` class is provided by the `java.lang` package.

```
import java.lang.*;
```

Importing the entire java.lang package.

As shown above, the entire `java.lang` package can be imported.

```
import java.lang.String;
```

Importing the String class from the java.lang package.

Alternatively, as shown above, the `String` class can be imported on its own from the `java.lang` package.

Using the String class

Assignment and methods

```
String str1 = "espresso";
String str2 = "espresso";
System.out.println(str1.equals(str2));        // true
System.out.println(str1.toUpperCase());       // ESPRESSO
System.out.println(str1.substring(0,2));      // es
System.out.println(str1.startsWith("o"));     // false
System.out.println(str1.endsWith("o"));       // true
System.out.println(str1.replace('e', 'E'));   // ExprEsso
```

Importing the String class from the java.lang package.

Concatenation

The `String` concatenation operator is +.

It is capable of appending:

- one `String` to another; and
- other data types to a `String`.

```
String name = "Nottingham␣Trent␣University␣";
int year = 2018;

/*part one*/
System.out.println(name);
System.out.println("Nottingham␣" + "Trent␣" + "Univeristy");
/*part two*/
System.out.println("Nottingham\n" + "Trent\n" + "Univeristy\n");
/*part three*/
System.out.println("Nottingham␣Trent␣University␣" + year);
System.out.println(name + year);
System.out.println("Nottingham␣Trent␣University␣" + 2018);
```

Conatenating Strings.

Java distinguishes between the concatenation operator and addition operator through **operand context**, as shown in the examples below.

```
System.out.println(5 + 5);  // 10
```

Conatenating integers

```
System.out.println("5 plus 5 equals " + 5 + 5);  // 5 plus 5 equals 55
```

Conatenating integers and Strings

```
System.out.println("5 plus 5 equals " + (5 + 5));  // 5 plus 5 equals 10
```

Conatenating integers and Strings

11.4 Default operators

Arithmetic operators

Operator	Meaning
+	Addition and String concatenation.
-	Subtraction.
*	Multiplication.
/	Division.
%	Remainder.

Equality and relational operators

Operator	Meaning
==	Equal to.
!=	Not equal to.
>	Greater than
>=	Greater than or equal to.
<	Less than.
<=	Less than or equal to.

Logical operators

Operator	Meaning
&&	Conditional-AND.
||	Conditional-OR.

11.5 Arrays

Definition

An **array** is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key.

Using arrays

Java arrays are considered as a special kind of default objects.

There are two different ways of defining an array, as shown below.

```
int[] anArray {1, 2, 3, 4};
```

Defining an array in one line.

```
int[] anArray = new int[4];  // or int anArray[] = new int[4];

int anArray[0] = 1;
int anArray[1] = 2;
int anArray[2] = 3;
int anArray[3] = 4;
```

Defining an array using index values.

Arrays have a `length` attribute that can be accessed using `.length`, as shown below.

```
int len = anArray.length;
```

Using the array length attribute.

Multi-dimensional arrays

A **multi-dimensional array** is an array with more than one level or dimension.

```
String[][] names = new String[4][7];
```

Defining a multi-dimensional.

The `length` attribute can also be used on multi-dimensional arrays, as shown below.

```
System.out.println(names.length);     // 4
System.out.println(names[0].length)   // 7
```

Defining a multi-dimensional array.

It is important to ensure that the elements in a multi-dimensional array are accessed properly, as shown below.

```
String[][] names = {{"Dr. " , "Mr. " , "Ms. "} , {"Smith" , "Jones"}};

System.out.println(names[0][0] + names[1][0]);  // Dr. Smith
System.out.println(names[0][1] + names[1][0]);  // Mr. Smith
System.out.println(names[0][2] + names[1][1]);  // Ms. Jones
System.out.println(names[0][0] + names[1][2]);  // Exception in thread "main" java.lang.arrayindexoutofboundsexception: 2
```

Accessing elements in a multi-dimensional array.

11.6 Static and non-static

Methods

Definitions

Static methods are known as **class methods** and are used when code exists that is to be shared across all instances of the same class.

Non-static methods are known as **instance methods** and are used when code exists that depends on individual characteristics of its class, such as an attribute.

Examples

```
class Hello {
   /* main method */
   public static void main (String[] args) {
      greetingInstance("Bob");       // illegal
      greetingClass("Bob");          // legal
      Hello test = new Hello();
      test.greetingInstance("Bob");  // legal
      Hello.greetingClass("Bob");    // legal
   }

   /* class method */
   static void greetingClass(String name) {
      System.out.println("Hello " + name);
   }

   /* instance method */
   void greetingInstance(String name) {
      System.out.println("Hello " + name);
   }
}
```

Class containing a static and non-static method.

Attributes

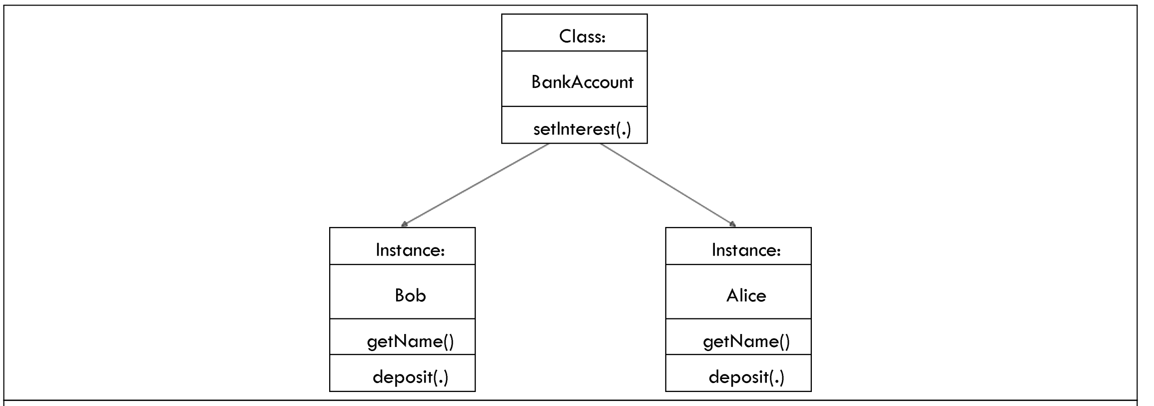


Figure 11.1: Object diagram.

The object diagram above shows that attributes may also be static. In this example, the `setInterest()` method is common to all instances of the class. As a result, the `interestRate` attribute may be a static attribute, as shown below.

```
static int interestRate;
```

Defining a static attribute.

11.7 Constants

Definition

A **constant** is a value that cannot be altered by the program once initialised. A compilation error would be thrown should an attempt be made to change the value of a constant.

Using constants

In Java, constants are defined by using the `final` keyword, as shown below.

```
final int interestRate = 15;
```

Defining a constant.

A constant can also be static, as shown below.

```
final static int interestRate = 15;
```

Defining a static constant.

Why are constants useful?

Constants are useful because:

- they avoid the ”magic number” problem by providing meaning to otherwise unclear literal values, such as using a constant named `MAX_LINES` for defining how many lines can be read from a file; and
- if a constant is used in multiple places, its value need only be modified in one location therefore improving consistency and maintainability.

11.8 Parameter passing

By value/reference

When passing a parameter **by value**, a copy of the value is taken such that any changes made to tbe value inside the method are not reflected in the outer program.

When passing a paramter **by reference**, the memory address that points to the value is taken such that any changes made to the value inside the method are reflected in the outer program.

In Java:

- for standalone data primitive types, only values are passed; and
- only objects are passed by reference.

When standalone primitive data types, such as integer, are passed as parameters, the changes made by the method to the values are not reflected in the outer program. For this to happen, the primitive data types must first be wrapped within an object so that they are passed by reference instead.

Examples

```
class PassingParameters {
    static void increase(int n) {
        // Number before increase: 10
        System.out.println("Number␣before␣increase:␣" + n);

        n++;

        // Number before increase: 11
        System.out.println("Number␣after␣increase:␣" + n);
    }

    public static void main (String[] args) {
        int number = 10;

        increase(number);

        // Number in main method: 10
        System.out.println("Number␣in␣main␣method:␣" + number);
    }
}
```

Passing a primitive data type.

```
class PassingParameters {
    static int increase(int n) {
        // Number before increase: 10
        System.out.println("Number␣before␣increase:␣" + n);

        n++;

        // Number before increase: 11
        System.out.println("Number␣after␣increase:␣" + n);

        return n;
    }

    public static void main (String[] args) {
        int number = 10;

        number = increase(number);

        // Number in main method: 11
        System.out.println("Number␣in␣main␣method:␣" + number);
    }
}
```

Passing a primitive data type and returning the changed value.

12. Introduction to Internetworking

12.1 Networks

What are computer networks?

A **computer network** is a collection of interconnected autonomous computer systems.

Computer systems are said to be interconnected if they can exchange information. They can be connected using a number of different types of physical connections:

- copper;
- fiber optic; and
- infrared etc.

Classification

There are two main classification criteria for computer networks:

- topology/communication; and
- scale.

Topology/communication

Broadcast networks have a single communication channel shared by all machines.

Point-to-point networks consist of many connections between pairs of machines.

Scale

Network	Scale
Personal area networks.	Up to one meter.
Local Area Networks (LANs).	Room, building, campus etc.
Metropolitan area networks (MANs).	City
Wide Area Networks (WANs).	Country, continent.
Internetworks	Any size.

Distributed systems

Definition

Distributed systems are a special case of a computer network-based system.

How they work

In general, applications in networks are aware of other computers connected to the device on which they are running. In distributed systems:

- the fact that several computers exist in the network is masked by the OS, making the system transparent and coherent; and
- the OS can automatically allocate jobs to processors and move files among variable computers without explicit user intervention.

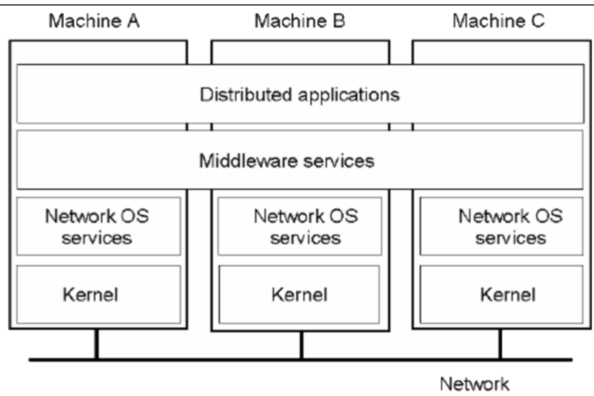


Figure 12.1: A distributed system on a network.

In the figure above, `Machine A`, `Machine B` and `Machine C`:

- are connected via a network;
- share some distributed applications;
- share some middleware services (computer software that provides services to software applications beyond those available from the operating system); and
- each have their own network OS services and kernel.

12.2 Internetworks

Definition

An **internetwork** is a collection of interconnected networks that are connected by gateways/routers.

The Internet

The Internet is the best known example of an internetwork. In the past, the Internet:

- started as the Advanced Research Projects Agency Network (ARPANET);
- was used as the US military network for communications in the late 1960s;
- started with only four computer systems (hosts); and
- used only a single protocol.

The modern internet is currently a multi-protocol network of networks.

Communication protocols

Definition

A **protocol** is a formal standard comprised of rules, procedures and formats that define how communication will take place between two or more devices over a network. Network protocols govern the end-to-end communication between networked devices and promotes timely, secure and managed network communication.

Protocols ensure that the sending network device is transmitting data in the same format that the receiving network device is expecting, including:

- the order of messages; and
- how the information is presented within a message.

If this was not agreed before communication, the application receiving the data would be unable to decipher and display the data.

Layered protocols

Due to the complexity of computer communications, internetworks are arranged in layers.

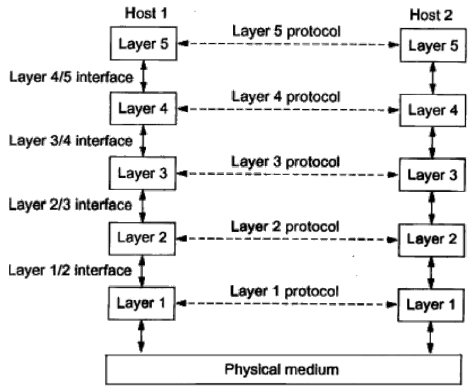


Figure 12.2: Protocol layers.

Different layers handle different aspects of the communication. As such, a **layer** refers to a group of related services that are performed in a given hierarchy.

Layer protocols operate in such a way where:

- each layer provides services to the layer immediately above;
- each layer passes data and controls information to the layer immediately below;
- a given layer n on one machine carries out a conversation with the same layer n on another machine;
- no data are directly transferred from a given layer n to layer n on another machine, instead actual communication occurs over the physical medium; and
-

Each layer provides services to its upper layer.

Services

A **service** is a specified set of primitives/operations available to a user or other entity (layer).

Connection-oriented services:

- are used for reliable transmission;
- have increased overhead due to the need for acknowledgement of proper transmission for each data packet; and
- are analogous to the telephone system.

For example, file transfers use connection-oriented services as they require reliability to ensure that files are not corrupted.

Connectionless services:

- are used for fast transmission;
- are unreliable;
- have decreased overhead as there are no requirements for data packets to be acknowledged; and

- are analogous to the postal system.

For example, audio and video streaming use connectionless services as lost packets are not too noticeable in a audio or video stream but waiting for data package acknowledgement would be highly noticeable.

Overview of internetworks

Protocols establish rules for communication by defining and providing **services**.

Internetworks are generally organised in **layers**, each using its own protocol.

A set of layers and protocols defines the **network architecture**.

12.3 Network architectures

Definition

Network architecture describes the design of a computer network. It is a framework for the specification of a network's:

- physical components and their functional organisation and configuration;
- operational principles and procedures; and
- the communication protocols used.

Open Systems Interconnect (OSI)

Definition

The **Open Systems Interconnection (OSI)** model is a conceptual model that characterises and standardises the communication functions of a computing system without regard to its underlying internal structure and technology. Its goal is the interoperability of diverse communication systems with standard protocols.

How it works

The model partitions a communication system into abstraction layers. The original version of the model defined seven layers.

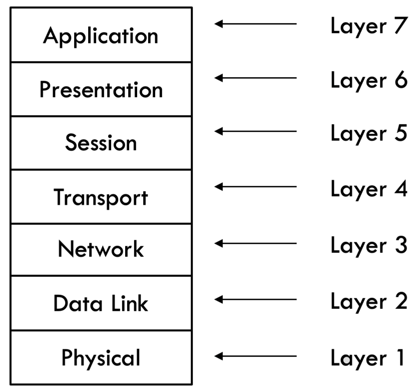


Figure 12.3: OSI network architecture overview.

However, the OSI model is not mandated for networking, yet most protocols and systems adhere to it quite closely.

Transmission Control Protocol / Internet Protocol (TCP/IP)

Definition

The **Transmission Control Protocol / Internet Protocol (TCP/IP) stack** is a set of networking protocols which work together as five connected layers and pass incoming and outgoing data packets throughout the layers during network communication. The TCP/IP stack contains layers, of software and hardware, which are designed to successfully transfer data across a network.

How it works

Packets must travel through the layers in the TCP/IP stack.

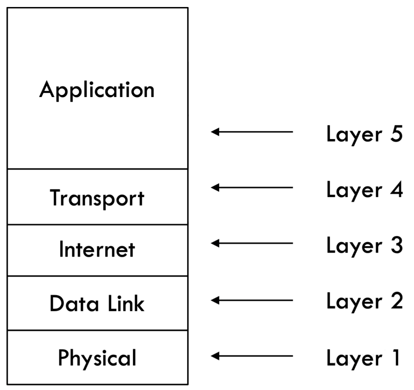


Figure 12.4: TCP/IP network architecture overview.

As shown above, there are five connected layers through which the packet must travel.

When the packet travels through the **sending host**, each layer encapsulates the data with additional information and passes it to the layer immediately below. When the packet is travels through the **receiving host**, each layer dis-encapsulates by removing the respective information and passing the remaining packet to the layer immediately above. This allows communication between respective layers on the sending and receiving hosts.

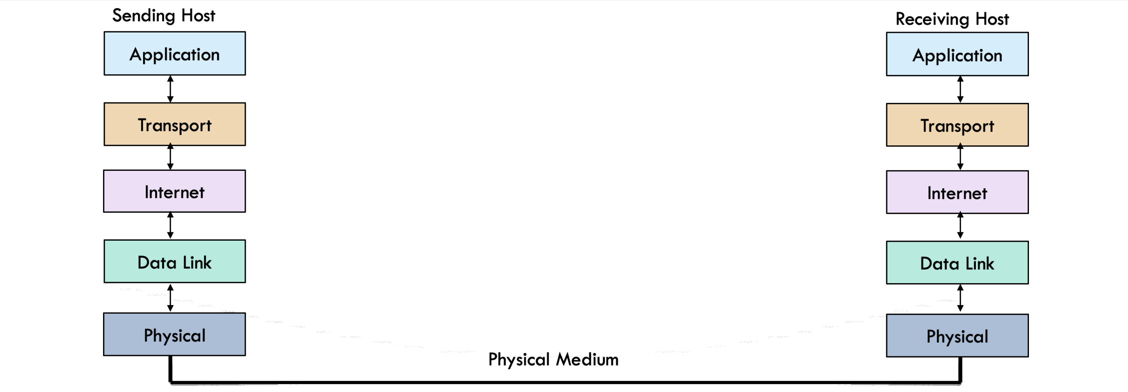


Figure 12.5: Sending host and receiving host.

The TCP/IP stack must transmit a packet from the sending host to the receiving host using the five layers, where the physical layers is connected by some physical medium.

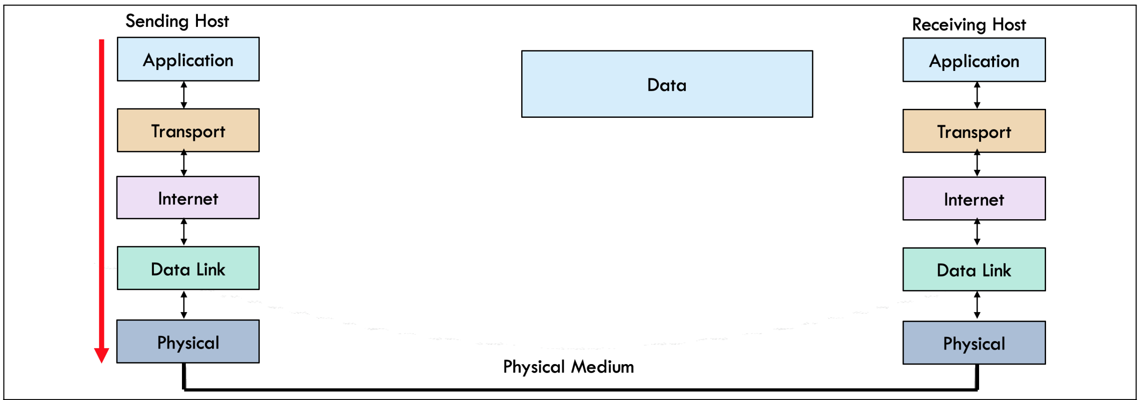


Figure 12.6: Sending host: application layer.

Firstly, at the **application layer** on the **sending host**, the application prepares the data in the message.

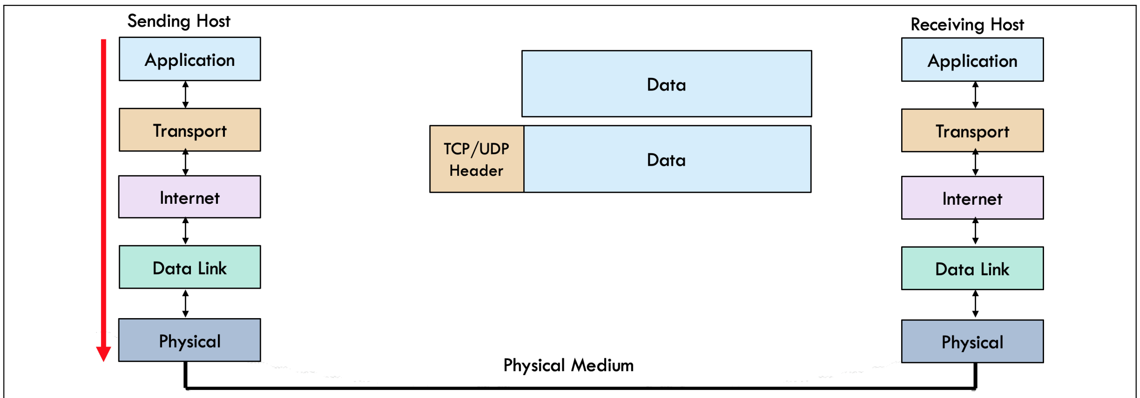


Figure 12.7: Sending host: transport layer.

When the packet arrives at the **transport layer** on the **sending host**, a TCP/UDP header is attached that contains:

- the port number; and
- error checking data.

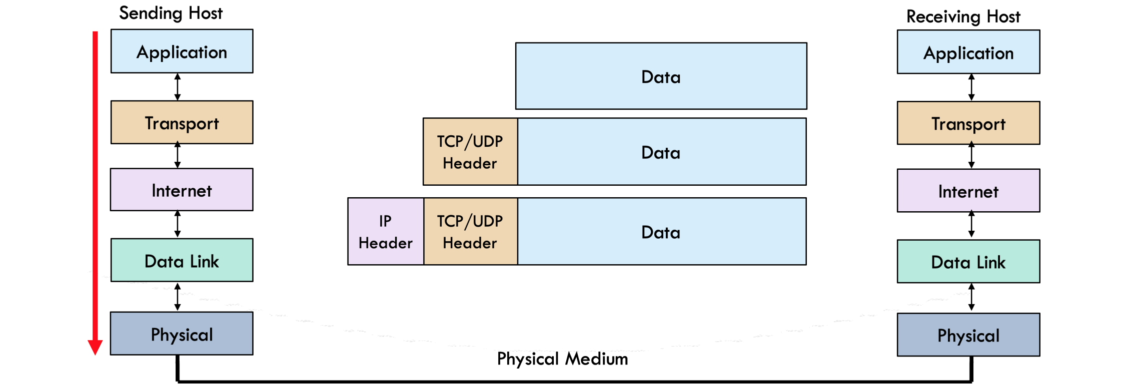


Figure 12.8: Sending host: Internet layer.

When the packet arrives at the **Internet layer** on the **sending host**, an IP header is attached that contains:

- the source IP address; and
- the destination IP address.

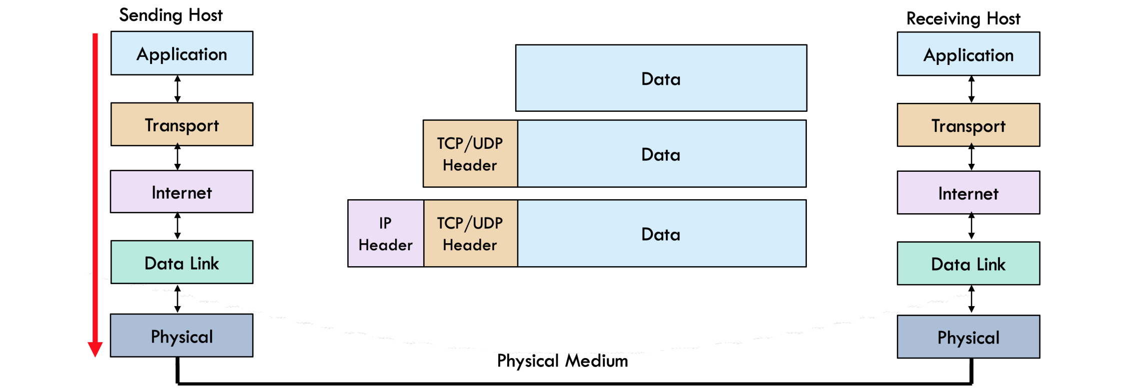


Figure 12.9: Sending host: data link layer.

When the packet arrives at the **data link layer** on the **sending host**, an frame header and frame trailer is attached that assigns the MAC address.

The packet is then sent through the physical medium to the receiving host.

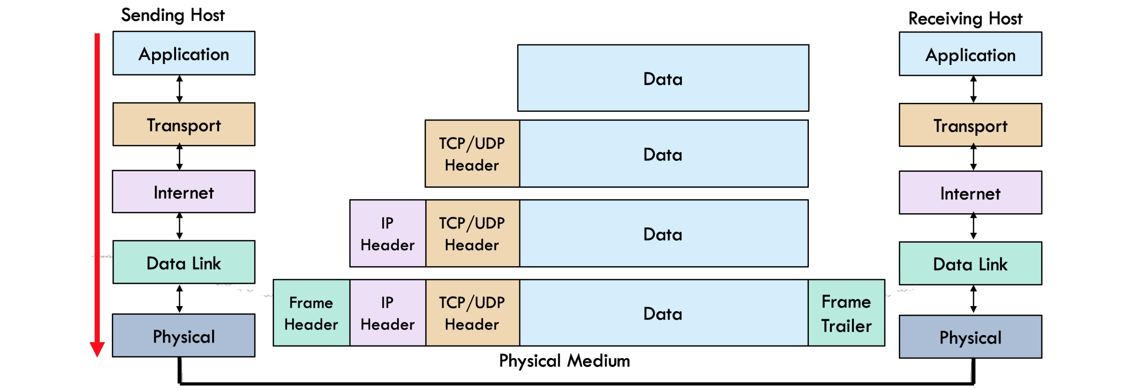


Figure 12.10: Receiving host: data link layer.

When the packet arrives at the **data link layer** on the **receiving host**, the MAC address is removed.

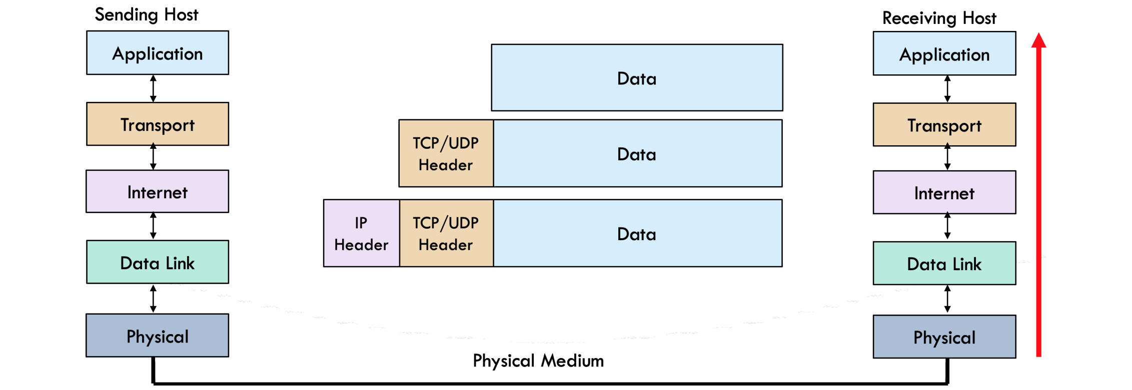


Figure 12.11: Receiving host: Internet layer.

When the packet arrives at the **Internet layer** on the **receiving host**, the IP header is removed providing the ability to:

- verify the source IP address; and
- verify the destination IP address.

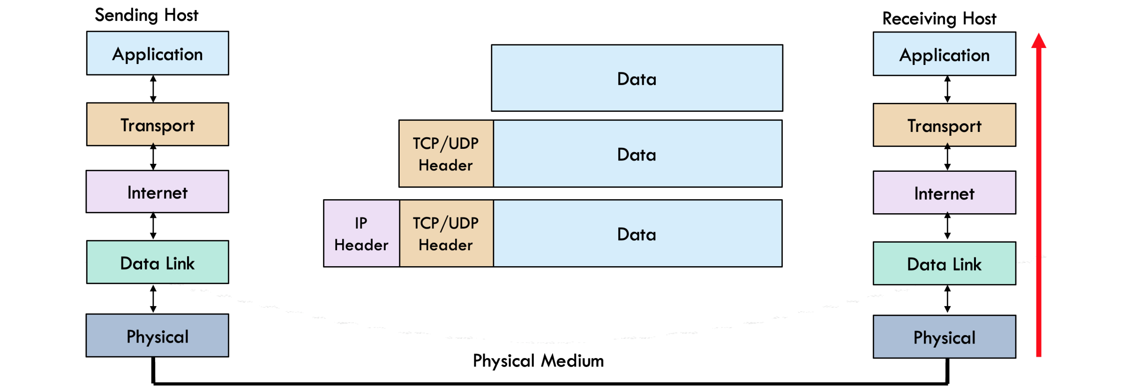


Figure 12.12: Receiving host: transport layer.

When the packet arrives at the **transport layer** on the **receiving host**, the TCP/UDP header is removed providing the ability to:

- direct the packet to the correct application using the port number; and
- perform error checking using the error checking data.

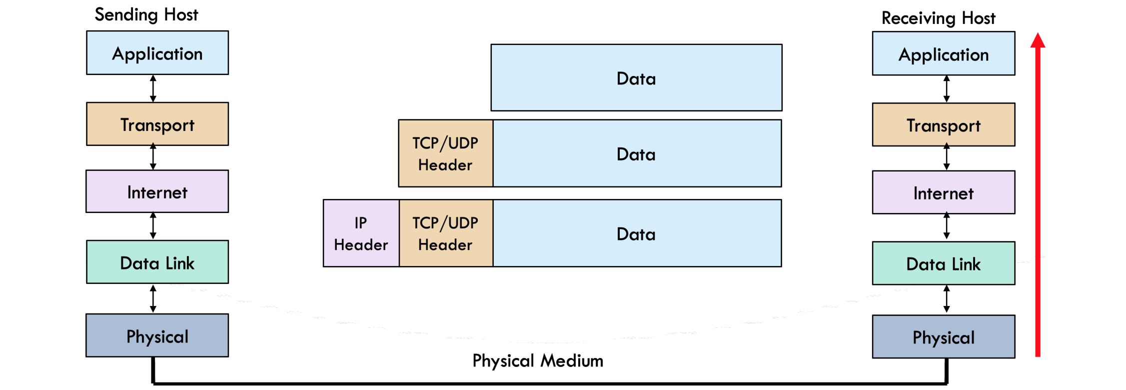


Figure 12.13: Receiving host: application layer.

Finally, when the packet arrives at the **application layer** on the **receiving host**, the data is ready to be used by the application on the receiving host.

Common protocols

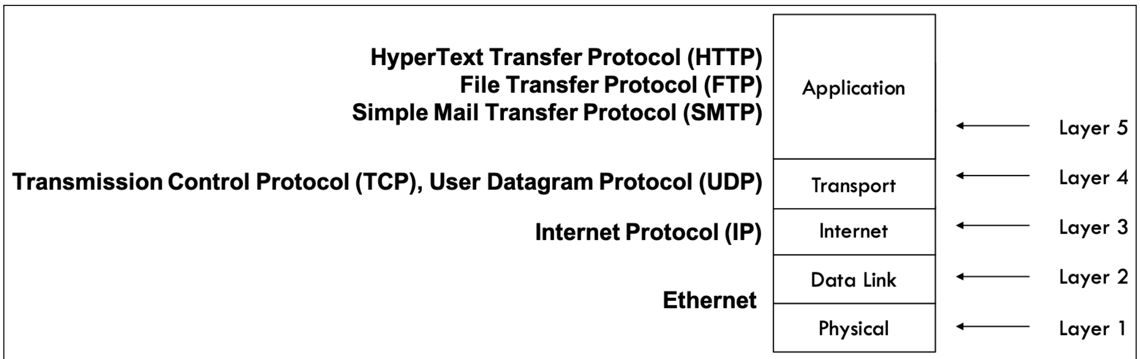


Figure 12.14: TCP/IP common protocols.

The figure above shows some common protocols and their respective layers in the TCP/IP stack.

Internet Protocol (IP)

Definitions

The **Internet Protocol (IP)** is responsible for end-to-end transmission. This is achieved by assigning each computer in a network (host) an IP address.

An **IP address** is a numerical address that is assigned to and uniquely identifies a device on a network. The IP address indicates where a packet of data is to be sent or from where it was sent. Routers can use IP addresses to direct a packet to its destination.

Packets

A **packet**, or **datagram**, is a piece of a message transmitted over a packet-switching network.

A packet can be sized between 1 `byte` to 1500 `bytes`.

As seen before, the internet layer, adds the source and destination IP address. It is important for packets to have this information as long messages may not fit in to a single packet and therefore will be split in to multiple packets, where each packet is routed individually.

Routing

Between two communicating computers on the Internet, there are usually many networking devices.

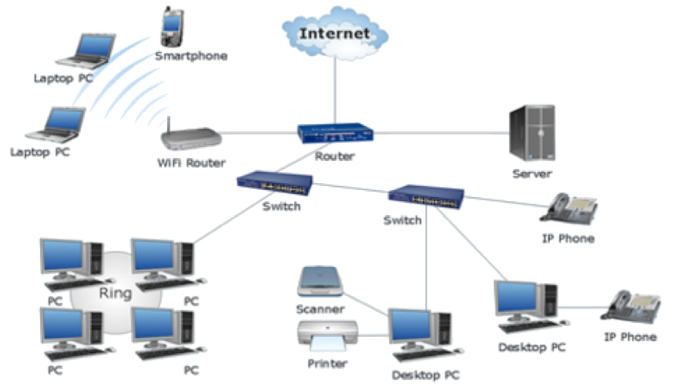


Figure 12.15: Networking devices on the Internet.

Devices that interconnect networks are called **routers/gateways**.

A **router** is a piece of hardware that connects to two or more networks which share the same protocols. They operate on the Internet and link layers of the TCP/IP stack. A router is responsible for:

- accepting data coming in to a network;
- sending data going out of a network;
- determining where to send incoming data; and
- determining where to send outgoing data.

Routers are aware of other routers to which they can send data and have a rough estimation on where is best to next send a piece of data. As such:

- packets are routed by "hopping" between routers; a hop is the act of traversing between one router and another router across a network;
- packets hop between the Internet layer and the link layer when being transmitted between routers connected to the Internet – each time a packet "hops", it is assigned a new MAC address for the next router to which it will be transmitted. This process continues until the packet reaches the router with the destination IP address; and
- each router has a record of other close routers in a routing table and sharing this information allows an algorithm to determine the optimum next path in the route for a packet, such as Dijkstra's Shortest Path algorithm.

This results in a very robust system as, if parts of the system go down, data can still reach its destination by alternative routes.

However, the Internet is a large system and therefore:

- data may get lost or arrive out of order;
- a wire can only carry a limited amount of data and may get congested with traffic if many people are sending data through the wire; and

- the capacity of wires is distributed in the best possible way, but overloading may still cause problems.

As a result, further protocol are required to ensure that packets are transmitted as intended.

Transmission Control Protocol (TCP)

Definition

The **Transmission Control Protocol (TCP)** is built on top of IP and provides a reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications running on hosts communicating via an IP network.

Connection-oriented protocol

TCP can be described as a connection-oriented protocol as:

- it adds packet ordering, sender and recipient information and a checksum;
- data is guaranteed to arrive in the correct order; and
- if any packet is lost, then it will be re-sent.

This means that TCP is reliable but has significant overheads.

Use by applications

Applications from the application layer that use TCP include:

- HTTP (for exchange of website files using a web browser);
- FTP (for file transfer); and
- SMTP (for email communications) etc.

User Datagram Protocol (UDP)

Definition

The **User Datagram Protocol (UDP)** is built on top of IP and allows computer applications to send messages to other hosts on an Internet Protocol (IP) network without prior communications

Connectionless protocol

UDP can be described as a connectionless protocol as:

- it adds packet length and a checksum to identify corrupt packets;
- data is not guaranteed to arrive in the correct order; and

- if any packet is lost, then it will be ignored.

This means that UDP is unreliable but has fast.

Use by applications

Applications from the application layer that use UDP include:

- VOIP (for audio calls);
- streaming services; and
- online gaming protocols.

13. Java Sockets

13.1 Client-server model

Definitions

A **client-server model** is a network application model in which servers and clients communicate over the network.

A **server** is a program that provides a number of services to a client.

A **client** is a program that requests services from the server.

How it works

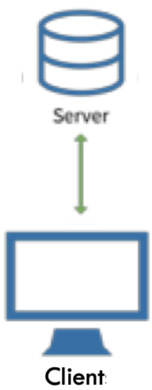


Figure 13.1: Client-server model with a single client.

The server:

- receives a request from a client;
- processes the request; and
- sends the results back to the client.

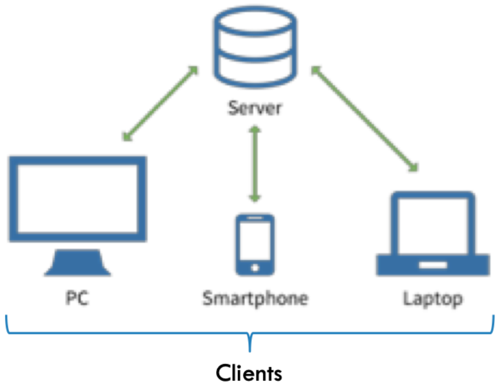


Figure 13.2: Client-server model with multiple clients.

As shown in the figure above, servers are usually designed to handle multiple clients simultaneously, such as email and the servers on the World Wide Web (WWW).

13.2 Sockets

Definition

A **socket** acts as an endpoint instance of a communication connection between two computer processes in a two-way communication link and is identified by a tuple consisting of an IP address and a port number.

Transferring data

The most common function in distributed client-server applications is to transfer data between computer systems (hosts).

As seen before,

- each computer system (host) in a TCP/IP network has an IP address;
- the internet layer in the TCP/IP stack adds the source and destination IP address; and
- routers operate on the data link layer in the TCP/IP and will use the IP addresses to forward the packets to the correct destination.

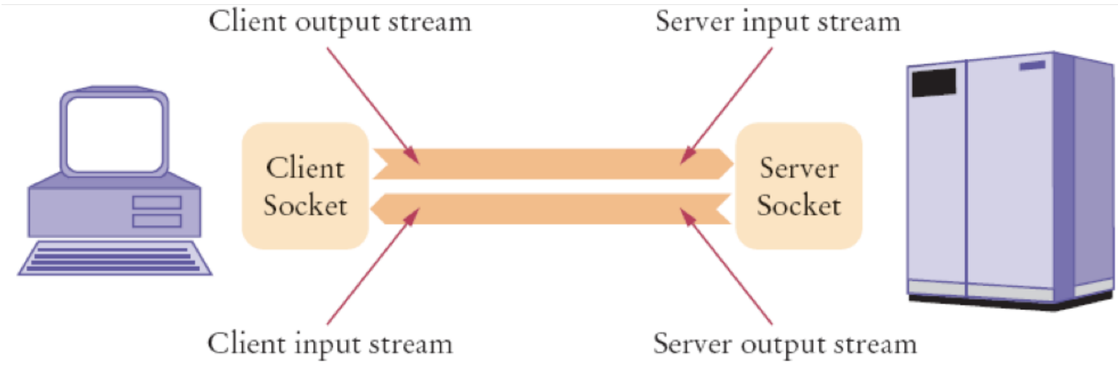


Figure 13.3: Transferring data using sockets.

A socket is formed, using the IP address and port number (e.g. 81.12.23.139:80), which specifies which device the packet must be sent to and the application being used on that device.

This shows that a connection over a TCP/IP can be represented by a pair of sockets (4-tuple).

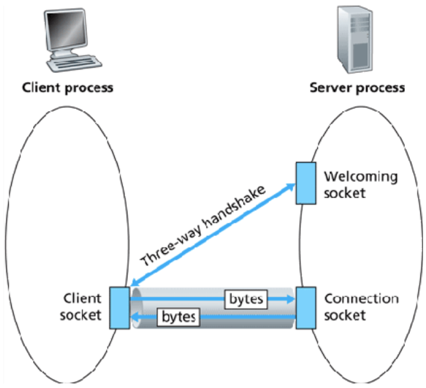


Figure 13.4: Creating sockets.

From a programmer's perspective, a socket pair represents the mechanism to transfer data between computers. As such, a programmer must create sockets to allow their applications to communicate with one another over a network.

13.3 Implementing sockets in Java

Types of sockets

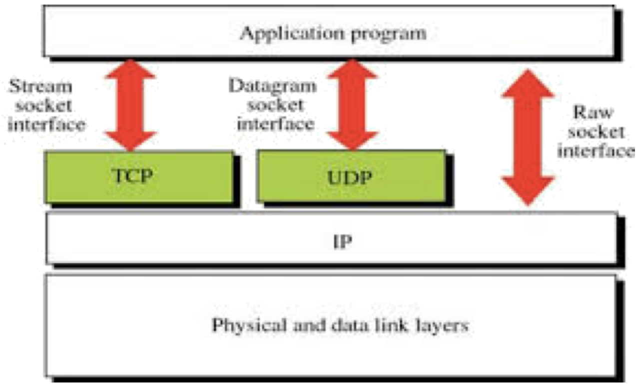


Figure 13.5: Types of sockets.

Java provides an API for two types of socket:

- stream sockets (TCP-based); and
- datagram sockets (UDP-based).

Raw sockets are typically available in routers and other network equipment.

Stream sockets (TCP-based)

Characteristics

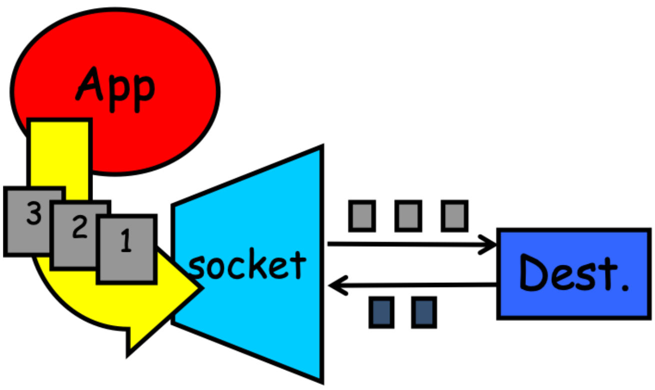


Figure 13.6: A stream socket.

A stream socket:

- is connection-oriented;
- is reliable;
- has overheads;

- sends ordered packets; and
- is bi-directional.

TCP sockets classes

The `java.net` package API provides the `ServerSocket` and `Socket` classes.

A TCP socket can easily be instantiated by calling the constructor of either class.

To set up a TCP server socket, the `ServerSocket` class is used.

```
int port = 9090;

ServerSocket server = new ServerSocket(port);

while(true) {
   System.out.println("Waiting for client...");
   Socket client = server.accept();
   System.out.println("Client " + client.getInetAddress() + " connected."
    );
}
```

Server: setting up a ServerSocket and accepting client connections.

As shown above, for a client to establish a connection to the server, the server must react by:

- accepting the connection set up by calling the `accept()` method; and
- providing a new socket for this connection.

In many cases, a server is set to endlessly wait for incoming connections. The `accept()` method blocks the program flow until a new connection request arrives within an infinite loop.

After a `ServerSocket` is set up on the server, a client can bind to it in order to establish a connection.

```
int port = 9090;
Socket server = new Socket("localhost", port);
System.out.println("Connection to " + server.getInetAddress() + "
   established.");
```

Client: establishing connection to the server.

In the code above both the client and server are running on the same machine, referred to in a relative fashion using the host name `localhost`. However, it is also possible to use the IP address for `localhost`, `127.0.0.1`.

TCP socket methods

The `Socket` class provides the following methods:

- `close()` with no return type; and
- `getInetAddress()` with a return type of `InetAddress`.

The `ServerSocket` class provides the following methods:

- `close()` with no return type;
- `accept()` with no return type;
- `getInetAddress()` with a return type of `InetAddress`.

TCP socket streams

To exchange data over sockets the `Socket` class uses streams:

- `getInputStream()`; and
- `getOutputStream()`.

The `java.io` package has classes for input and output:

- `DataInputStream`; and
- `DataOutputStream`.

These allow:

- different data types to be converted in to byte sequence for simple streams; and
- simple streams to be converted from byte sequence to their respective data types.

For transmitting simple types such as `int`, `long` and `char`, the `DataOutputStream` class provides a write method for each of these types:

- `writeInt()`;
- `writeLong()`; and
- `writeChar()` etc.

For transmitting a `String`, the UTF-8 format is normally used, which transfers the characters in Unicode encoding. The `DataOutputStream` class provides the write method `writeUTF()` for this operation.

Similarly, the `DataInputStream` class exists with corresponding methods:

- `readInt()`;
- `readLong()`;
- `readChar()`; and
- `readUTF()`.

Designing the server and client

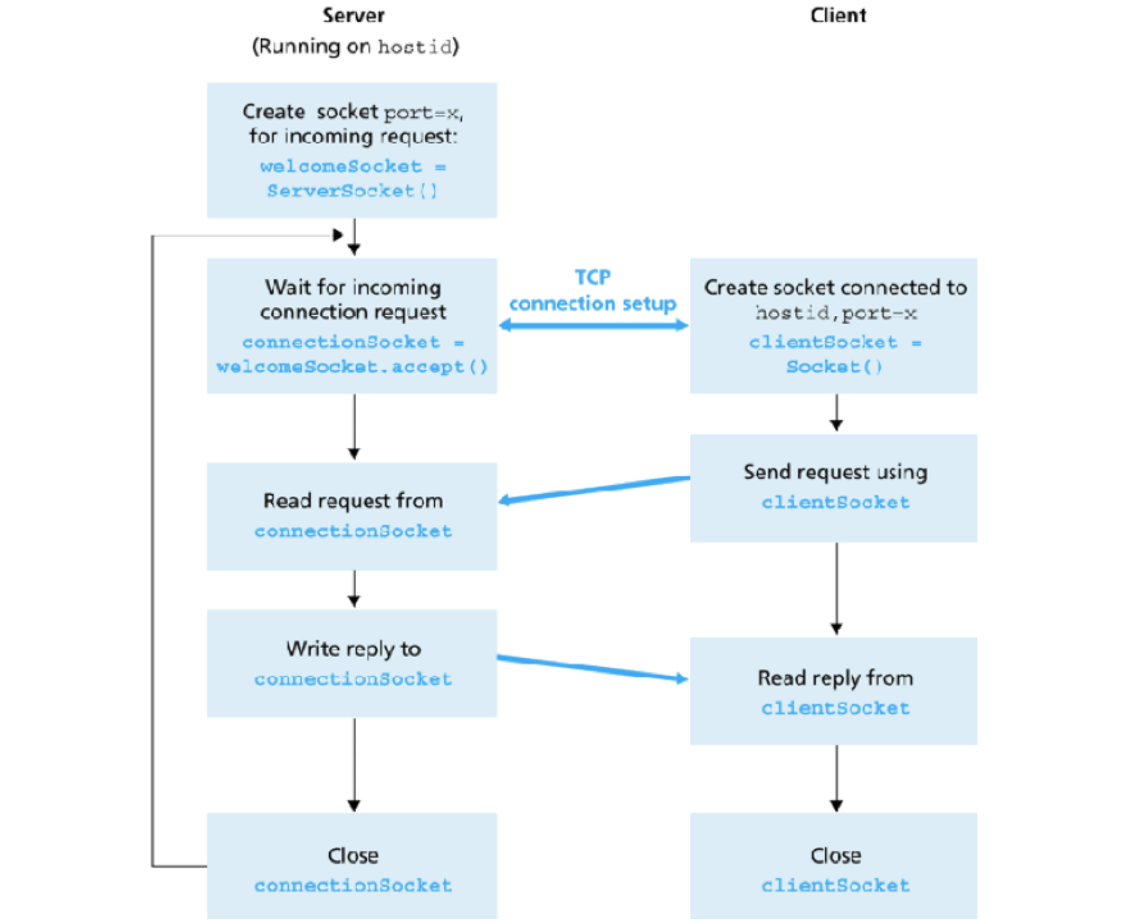


Figure 13.7: Server and client set up using TCP sockets.

In order to allow the set up seen in the figure above, the server must:

- have a ServerSocket set up;
- listen and accept connection;
- set up I/O streams; and
- send data to and receive data from the client.

In addition, the client must:

- have a Socket set up;
- set up I/O streams; and
- send data to and receive data from the server.

```java
import java.net.*;
import java.util.*;
import java.io.*;

public class TimeServer {
   public static void main(String[] args) throws IOException {
      ServerSocket server = new ServerSocket(9090);

      while(true) {
         System.out.println("Waiting...");

         // establish connection to client
         Socket client = server.accept();
         System.out.println("Connected to: " + client.getInetAddress());

         // create I/O streams
         DataInputStream inFromClient =
            new DataInputStream(client.getInputStream());
         DataOutputStream outToClient =
            new DataOutputStream(client.getOutputStream());

         // get any data from client
         System.out.println(inFromClient.readUTF());

         // send date to client
         Date date = new Date();
         outToClient.writeUTF(date.toString());

         // close client connection
         client.close()
      }
   }
}
```

TimeServer class.

```java
import java.net.*;
import java.io.*;

public class TimeClient {
   public static void main(String[] args) throws IOException {
      // establish connection to server
      Socket server = new Socket("localhost", 9090);
      System.out.println("Connected to: " + server.getInetAddress());

      // create I/O streams
      DataInputStream inFromServer = new DataInputStream(server.
   getInputStream());
      DataOutputStream outToServer = new DataOutputStream(server.
   getOutputStream());
```

```
        // send Unicode string to server
        outToServer.writeUTF("Time");

        // read data from server
        String data = inFromServer.readUTF();
        System.out.println("Server␣response:␣" + data);

        // close server connection
        server.close();
    }
}
```

TimeClient class.

Datagram sockets (UDP-based)

Characteristics

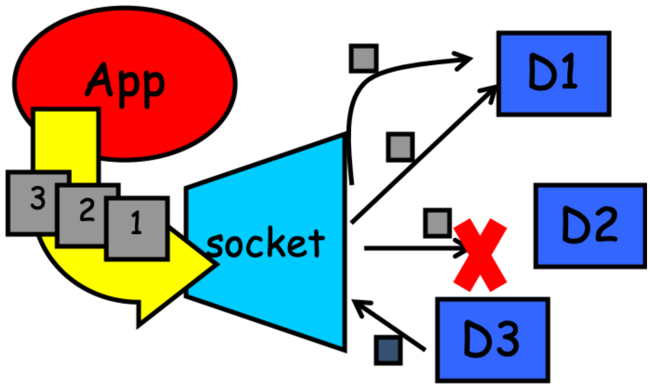


Figure 13.8: A datagram socket.

A datagram socket:

- is connectionless;
- is unreliable;
- has little overheads and makes the best effort without;
- sends unordered packets; and
- is uni-directional.

UDP socket classes

The `java.net` package API provides the `DatagramSocket` and `DatagramPackets` classes.

A UDP socket can easily be instantiated by calling the constructor of either class.

To set up a UDP socket, the `DatagramSocket` class is used and to accept packets the `DatagramPackets` class is used.

UDP socket methods

The DatagramSocket class provides the following methods:

- connect(InetAddress a, int port) with no return type;
- close() with no return type;
- recieve(DatagramPacket p) with no return type; and
- send(DatagramPacket p) with no return type.

The DatagramPacket class provides the following methods:

- getData() with a return type of byte[]; and
- setAddress(InetAddress a) with no return type;
- setPort(int port) with no return type; and
- getAddress() with a return type of int.

Converting bytes to data

UDP packets do not have input/output stream support as seen with TCP packets. This means that data must be converted to and from byte arrays.

```
String message = "Some message;
byte[] data = message.getBytes();
DatagramPacket sendPacket = new DatagramPacket(data, data.length);
/* send DatagramPacket via DatagramSocket */
```

Client: sending a DatagramPacket.

```
byte[] data = new byte[1024];
DatagramPacket recievedPacket = new DatagramPacket(data, data.length);
/* recieve DatagramPacket via DatagramSocket */
data = recievedPacket.getData();
String message = new String(data);
```

Server: receiving a DatagramPacket.

Designing the server and client

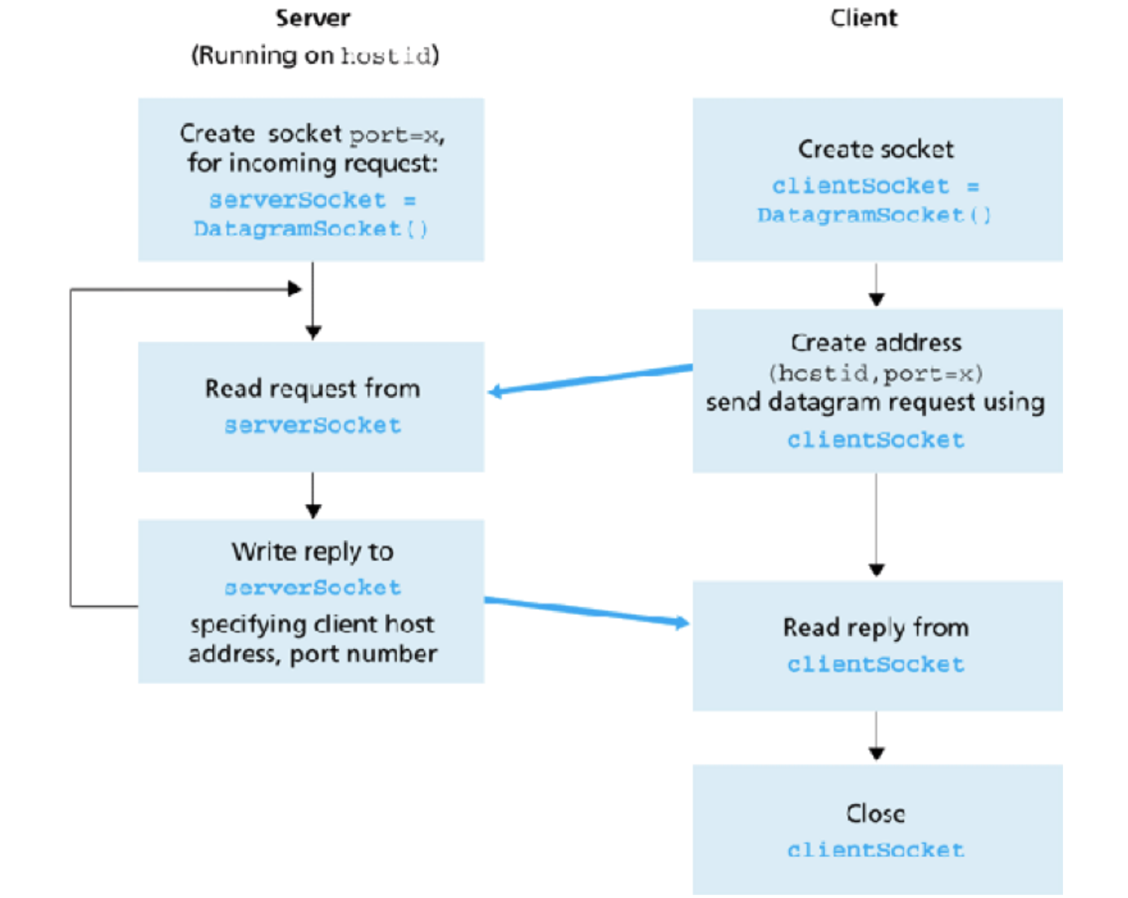


Figure 13.9: Server and client set up using TCP sockets.

In order to allow the set up seen in the figure above, the server must:

- have a DatagramSocket set up;
- create an empty byte array; and
- fill the array with the DatagramPacket recieved from the client.

It is also possible for the server to send packets back to the client using information regarding the client from the packet.

In addition, the client must:

- have a DatagramSocket set up;
- create byte data and DatagramPacket;
- add server address and port to the DatagramPacket; and
- send the DatagramPacket to the server.

```
int port = 9090;
DatagramSocket = new DatagramSocket(port);

while(true) {
   System.out.println("Waiting for packets...");
```

```
    // create an empty byte array and fill it
    byte[] data = new byte[1024];
    DatagramPacket recievedPacket = new DatagramPacket(data, data.length);
    socket.recieve(recievedPacket);

    // get data from packet
    String message = new String(recievedPacket.getData());
    System.out.println("Recieved␣from␣client␣" + recievedPacket.getAddress
     () + ",␣the␣message:␣" + message);
}
```

Server: setting up a DatagramSocket and accepting packets.

As shown above, for a server to receive packets from a client, the server must by:

- have a `DatagramSocket` set up; and
- recieve packets from the socket using the `recieve()` method.

In many cases, a server is set to endlessly wait for incoming connections. The `receive()` method blocks the program flow until a new connection request arrives within an infinite loop.

After a `DatagramSocket` is set up on the server, a client can send a `DatagramPacket` to the server without establishing a connection.

```
DatagramSocket socket = new DatagramSocket();

// convert a string to bytes
byte[] data = "This␣is␣the␣message".getBytes();

// put byte array in to a packet
DatagramPacket packet = new DatagramPacket(data, data.length);

// get IP address from hostname
InetAddress dest = InetAddress.getByName("localhost");
int port = 9090;

packet.setAddress(dest); // add IP address
packet.setPort(port);    // add port

socket.send(packet)      // send the UDP datagram
socket.close();          // close socket
```

Client: sending a DatagramPacket to the server.

14. Java Support for Multithreading

14.1 Recap

As seen in Chapter 3: Threads and Concurrency:

- threads allow for concurrent execution;
- sequential execution involves total ordering; and
- concurrent execution allows flexibility and introduces complexity.

In addition, concurrent programming is used in:

- single-processor multi-tasking systems using interleaving; and
- multi-processor/multi-core systems allowing maximum parallelism using overlapping.

As seen in Chapter 4: Synchronisation and Mutual Exclusion, semaphores and mutexes were used to solve the Producer-Consumer and Dining Philosophers problems.

14.2 Concurrency in Java

JVM and package support

The JVM allows an application to have multiple threads running concurrently.

There is a `Thread` class provided in the `java.lang` package and there are two ways to create a new thread:

- by **extending** the `Thread` class; or
- by **implementing** the `Runnable` interface.

When extending from a class:

- a subclass extends a base class;
- the subclass type *"is a"* base class type; and

- the behaviour of an established method in the base class can be redefined by overriding.

When implementing an interface:

- a class implements an interface;
- the class type implementing the interface *"has a"* of the interface type; and
- the class implements the methods present as abstract methods in the interface.

Methods for creating threads

Extending the `Thread` class

The `thread` class includes the following methods:

- `start()` – causes the thread to begin execution as the JVM will call the `run()` method of the thread;
- `run()` – call the run method of the thread directly;
- `sleep(long ms)` – cease the execution of the current thread for `ms` milliseconds;
- `wait()` – blocks the thread's execution; and
- `notify()` – notifies other threads to unblock.

```
public class ExampleThread extends Thread {
   int parameter;

   ExampleThread(int p) {
      parameter = p;
   }

   public void run() {
      // what should the thread do?
   }
}
```

ExampleThread class extending the Thread class.

As shown above, the `ExampleThread` class is a subclass of `Thread`, therefore inheriting the attributes and methods from the `Thread` class.

The `ExampleThread` class overrides the `run()` method from the `Thread` class as this must contain the code unique to a particular thread's behaviour.

The `ExampleThread` class can then be started by creating an object.

```
ExampleThread t = new ExampleThread(10);
t.start();
```

Creating an instance of ExampleThread.

As shown above, once an `ExampleThread` object has been created, the `start()` method (inherited from `Thread`) can be called to begin the thread's execution.

```
public class ThreadID extends Thread {
   int id;

   ThreadID (int _id) {
      id = _id;
   }

   public void run() {
      System.out.println("This␣is␣thread␣" + id);
   }
}

public static void main(String[] args) {
   System.out.println("Main␣thread␣starts");

   ThreadID t1 = new ThreadID(1);
   ThreadID t2 = new ThreadID(2);
   ThreadID t3 = new ThreadID(3);

   t1.start();
   t2.start();
   t3.start();

   System.out.println("Main␣thread␣ends");
}
```

Example of extending the Thread class.

Implementing the `Runnable` interface

```
public class ExampleThread implements Runnable {
   int paramter;

   ExampleThread(int p) {
      parameter = p;
   }

   public void run() {
      // what should the thread do?
   }
}
```

ExampleThread class implementing the Runnable interface.

As shown above, the `ExampleThread` class implements the `Runnable` class, therefore inheriting the abstract methods from the `Runnable` interface.

The `ExampleThread` class implements the `run()` method from the `Runnable` interface as this must contain the code unique to a particular thread's behaviour.

The `ExampleThread` class can then be started by creating a `Thread` object using the `ExampleThread`.

```
ExampleThread r = ExampleThread(10);
Thread t = new Thread(r);
t.start();
```

Creating an instance of a Thread by using ExampleThread.

As shown above, once an `ExampleThread` object has been created and the `Thread` object uses the `ExampleThread` object as a parameter in its constructor, the `start()` method can be called to begin the thread's execution.

```java
public class ThreadID implements Runnable {
   int id;

   ThreadID (int _id) {
      id = _id;
   }

   public void run() {
      System.out.println("This is thread " + id);
   }
}

public static void main(String[] args) {
   System.out.println("Main thread starts");

   ThreadID t1 = new ThreadID(1);
   ThreadID t2 = new ThreadID(2);
   ThreadID t3 = new ThreadID(3);

   Thread thr1 = new Thread(t1);
   Thread thr2 = new Thread(t2);
   Thread thr3 = new Thread(t3);

   thr1.start();
   thr2.start();
   thr3.start();

   System.out.println("Main thread ends");
}
```

Example of implementing the Runnable interface.

Comparison of methods for creating threads

Both methods allow threads to be created and have the same resulting behaviour.

Method	Advantages	Disadvantages
Extending `Thread`	**Less overhead when setting up threads** as the class that extends `Thread` can be instantiated as an object that acts as a thread straight away.	**May be less convenient** as Java (JDK < v1.8) does not permit multiple inheritance and therefore the class extending `Thread` cannot extend any other class.
Implementing `Runnable`	**More overhead when setting up threads** as the class that implements `Runnable` must be instantiated as an object and then passes as a parameter to the `Thread` constructor.	
	May be more convenient as it allows the restriction where Java (JDK < v1.8) does not permit multiple inheritance to be mitigated as the class that implements `Runnable` can extend another class.	

14.3 Synchronisation and mutual exclusion in Java

Synchronisation and mutual exclusion in Java can be demonstrated by considering the passenger problem.

Description of the passenger problem

This problem considers an air plane with two doors, one at the front and one at the rear.

The passenger problem considers:

- a air plane with two doors;
- one door is located at the front of the air plane; and
- another door is located at the rear of the air plane.

This problem has the following requirements:

- passengers can enter the air plane;
- passengers may not leave the air plane; and
- to determine the number of passengers in the air plane at any time, a concurrent system is implemented which is connected with some sensors at each door.

Implementation of the passenger problem

Simplifying the problem

In the implementation of this problem:

- each door is handled by a concurrent thread;
- a global variable is used to represent the current number of passengers in the air plane; and
- the global variable is updated by a door thread when a passenger enters the front or rear door.

The problem can be simplified for this implementation where:

- 200 passengers are expected;
- 100 passengers are simulated to enter the front door; and
- 100 passengers are simulated to enter the rear door.

As a result, when the experiment finishes, the system should show that there are 200 passengers in the air plane.

Code

```
public class Door implements Runnable {
   Airplane ap;

   Door(Airplane _ap) {
      ap = _ap;
   }

   public void run() {
      for(int i = 1; i <= 100; i++) {
         try {
            ap.increment();

            // cease execution of the thread for 0.5sec on every arrival
            Thread.sleep(500);
         } catch (InterruptedExecption e) {} // do nothing
      }
   }
}
```

Door class.

```
public class Airplane {
   public int count = 0;

   void increment() {
      count = count + 1;
      System.out.println("There␣are␣" + count + "␣passengers");
   }

   public static void main(String[] args) {
      Airplane ap = new Airplane();

      Door front = new Door(ap);
      Door read = new Door(ap);

      Thread t1 = new Thread(front);
      Thread t2 = new Thread(rear);

      t1.start();
      t2.start();
   }
}
```

Airplane class.

Concurrent execution issue

The code above has a concurrent execution issue as the two doors behave in parallel and the order of their events is unpredictable. For example, an situation may arise where both threads are competing for the same resource.

Synchronisation and mutual exclusion must be implemented to prevent this situation.

A shared object, typically data items, can be accessed by multiple threads. When threads access shared data, they can:

- perform read operations;
- perform write operations; and
- perform read+write operations.

It is necessary to synchronise the shared object to ensure that the two threads are not competing for access to the shared resource. Java uses the `synchronized` keyword in the definition of the shared method that accesses the shared resource, as shown below.

```
synchronized void increment() {
   count = count + 1;
   System.out.println("There␣are␣" + count + "␣passengers");
}
```

Revised increment() method in the Airplane class.

14.4 Concurrent client-server model in Java

Premise

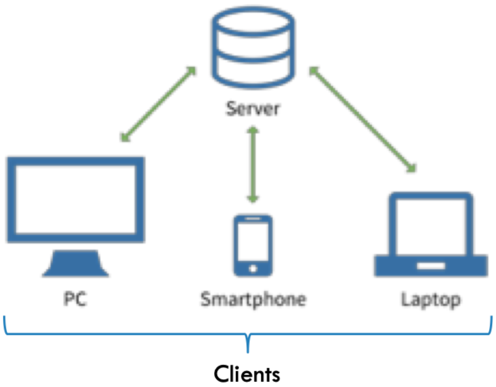


Figure 14.1: Client-server model with multiple clients.

When a server is required to server many clients, as shown in the figure above, an architecture is required that does not simply serve one client while making other clients wait their turn.

Instead, the server should:

- already remain available; and
- accept several clients in parallel.

Handlers

Creating a `Handler` class

In order to construct the concurrent client-server model, the server does not need to perform its tasks on its own. Instead, the tasks should be delegated to a thread, typically defined as the `Handler` class, as shown below.

```
public class Handler implements Runnable {
   public Handler(Socket client) {
      // ...
   }

   public void run() {
      // an entire task can be implemented here
   }
}
}
```

Handler class.

Each new client can then be assigned to a thread, as shown below.

```
public class MultiThreadedServer {
   public static void main(String[] args) {
      // set up server
      ServerSocket server = new ServerSocket(9090);

      while(true) {
         System.out.println("Waiting...");

         // accept client connections
         Socket client = server.accept();
         System.out.println("Connected " + client.getInetAddress());

         // assign each client to a thread
         Handler t = new Handler(client);
         Thread th = new Thread(t);
         th.start();
      }
   }
}
```

MultiThreadedServer class.

Revising the `TimeServer` class

```
import java.net.*;
import java.util.*;
import java.io.*;

public class TimeServer {
   public static void main(String[] args) throws IOException {
      ServerSocket server = new ServerSocket(9090);

      while(true) {
         System.out.println("Waiting...");

         // establish connection to client
         Socket client = server.accept();
         System.out.println("Connected to: " + client.getInetAddress());

         DataOutputStream outToClient =
            new DataOutputStream(client.getOutputStream());

         // send date to client
         Date date = new Date();
         outToClient.writeUTF(date.toString());
      }
   }
}
```

TimeServer class.

The code above shows the original output stream capability of the `TimeServer` class explored in Chapter 13: Java Sockets.

Handlers can be used to revise this `TimeServer` class, as shown below, in order to allow multiple clients concurrently.

```
import java.net.*;
import java.util.*;
import java.io.*;

public class TimeServer {
   public static void main(String[] args) throws IOException {
      ServerSocket server = new ServerSocket(9090);

      while(true) {
         System.out.println("Waiting...");

         // establish connection to client
         Socket client = server.accept();
         System.out.println("Connected to: " + client.getInetAddress());

         // assign the client to a handler
         TimeHandler th = new TimeHandler(client);
         Thread t = new Thread(th);
         t.start();
      }
   }
}
```

Revised TimeServer class.

The revised `TimeServer` shown above requires the `TimeHandler` class shown below.

```
import java.net.*;
import java.util.*;
import java.io.*;

public class TimeHandler implements Runnable {
   Socket client;
   DataOutputStream outToClient;

   public TimeHandler(Socket _client) throws IOException {
      client = _client;
      outToClient = new DataOutputStream(client.getOutputStream());
   }

   public void run() {
      try {
         // return date
         Date data = new Date();
         outToClient.writeUTF(date.toString());  // send date to client
      } catch (IOException e) ()  // do nothing
   }
}
```

TimeHandler class.

As seen above, the `DataOutputStream` has been moved from the `TimeServer` class to the `TimeHandler` class to allow each client to have its own output stream and therefore prevent any client from waiting for another client.

15. GUIs in Java

15.1 Introduction to GUIs

Definition

A **Graphical User Interface (GUI)** allows users to interact with electric devices primarily through the use of graphical icons and visual indictors. This is in contract to text-based user interfaces which allow user interaction through typed command labels and/or text navigation.

Why are GUIs used?

GUIs are used because they:

- allow event-driven programming techniques;
- allow large amounts of information to be presented neatly;
- provide easier methods for the user to interact with the application;
- provide easier methods for the user to control the execution of the application; and
- are generally more visually appealing than text-based user interfaces.

15.2 GUI construction in NetBeans

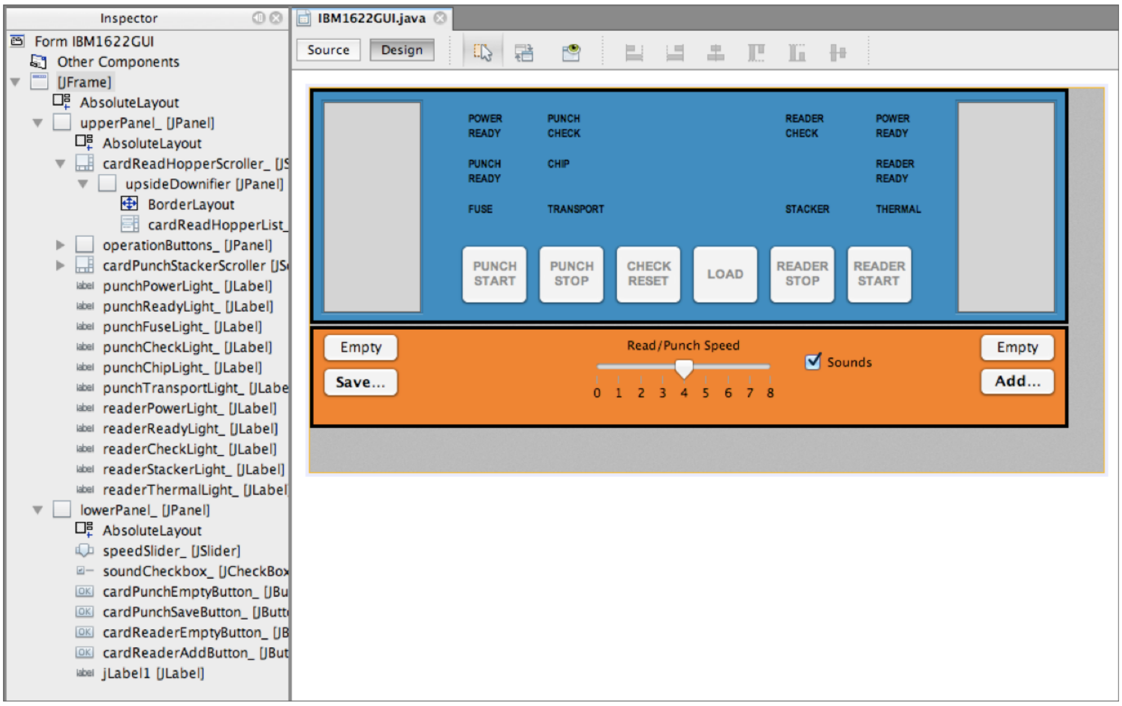


Figure 15.1: NetBeans GUI editor.

As seen in the figure above, the NetBeans IDE allows components such as buttons, sliders and text to be ”dragged” on to the window. Subsequently, the properties of these components can be changed by right-clicking the components.

15.3 Importing Java GUI libraries

`swing` is a package available to efficiently design and develop Java applications with a GUI.

The `swing` package is built on top of the abstract window toolkit (`awt`). Therefore, some components in the `swing` package still use some properties from the `awt`.

```
import javax.swing.*;
```

Importing the entire javax.swing package.

As shown above, the entire `javax.swing` package can be imported.

```
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
```

Importing the JFrame and SwingUtilities components from the javax.swing package.

Alternatively, as shown above, the `JFrame` and/or `SwingUtilities` components can be imported on their own from the `javax.swing` package.

In addition, most listeners are used from the `awt` and therefore these must be imported alongside `swing`.

```
import java.awt.event.*;
```

Importing the entire java.awt.event package.

As shown above, the entire `java.awt.event` package can be imported.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

Importing the ActionEvent and ActionListener components from the java.awt.event package.

Alternatively, as shown above, the `ActionEvent` and/or `ActionListener` components can be imported on their own from the `java.awt.event` package.

For more complex GUIs, other components of `java.awt` and `javax.swing` may also be required to be imported.

15.4 JComponent

Overview

The `javax.swing` package provides `JComponent` for a variety of components to be used in GUI building.

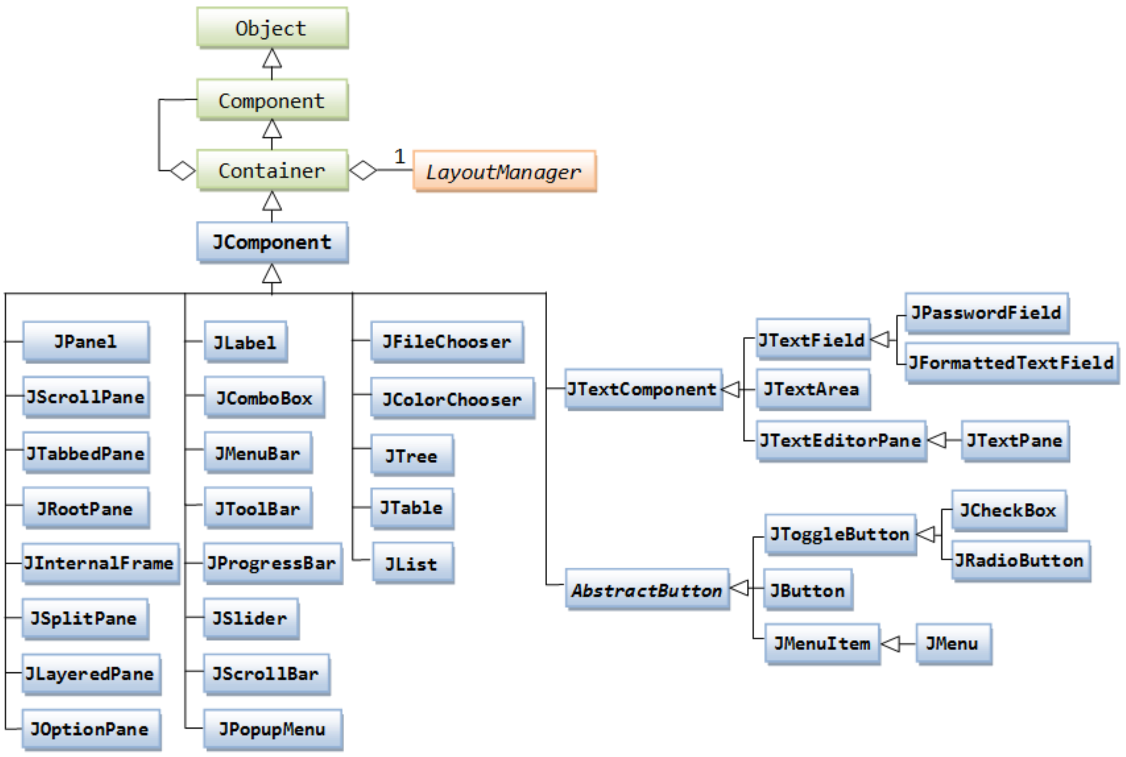


Figure 15.2: JComponent class diagram.

The figure above shows a class diagram containing the components available from `JComponent`.

Construction

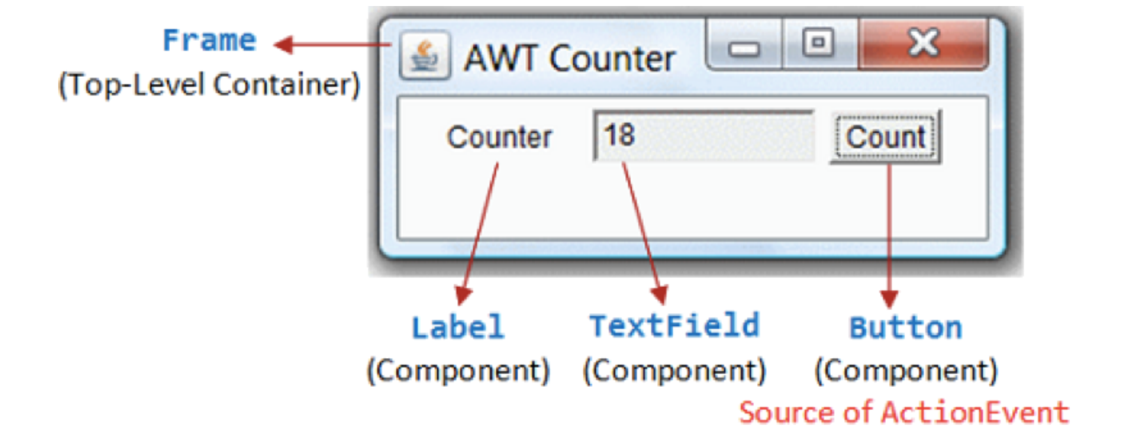


Figure 15.3: Window layout.

When creating a GUI:

- a top-level container (window) must be created to display components, usually a `JFrame`;
- the `setLayout(LayoutManager m)` method is used to specify a layout for the `JFrame`;
- some **components** are created, such as buttons, text fields and check boxes; and
- the components are added to the display area according to the selected **layout manager**.

Next, **listeners** must be written and attached to the components to enable interaction with components. A listener gets a message when an event occurs and executes some code.

Finally, the `JFrame` must be displayed by using `setVisible(true)`.

15.5 JFrame

Incorporating the JFrame

As seen before, the top-level container is typically of a `JFrame` type.

There are two methods of incorporating the `JFrame` with a Java program:

- creating the `JFrame` in a class; or
- extending the class to `JFrame`.

```
class GUIclassExample {
   JFrame jf = new JFrame();
   jf.setTitle("Text to put in the title bar");
}
```

Creating a JFrame in a class.

```
class GUIclassExample extends JFrame {
   setTitle("Text to put in the title bar");
}
```

Extending a class to JFrame.

JFrame methods

Visibility

Call the `setVisible(true)` method to make the generated `JFrame` appear on the screen.

Close

Call the `setDefaultCloseOperation(int operation)` method with the parameter value `EXIT_ON_CLOSE` to ensure that the application exits using the `System exit` method when the window is closed.

Size

Call the `setSize(int width, int height)` method to give a fixed size to the `JFrame`.

Alternatively, the `pack()` method can be called to auto-resize the `JFrame` under control from the frame layout manager by adjusting the size to platform dependencies and other factors that affect component size.

Window type

Call the `setType(Window.Type type)` method with the parameter `Type.NORMAL` or `Type.UTILITY` to set the buttons in the title bar.

Figure 15.4: `NORMAL` window type.

The `NORMAL` window type contains the minimise, maximise and close buttons is generally used for main application windows.

Figure 15.5: `UTILITY` window type.

The `UTILITY` window type contains only a close button is generally used for dialogue boxes.

Content pane

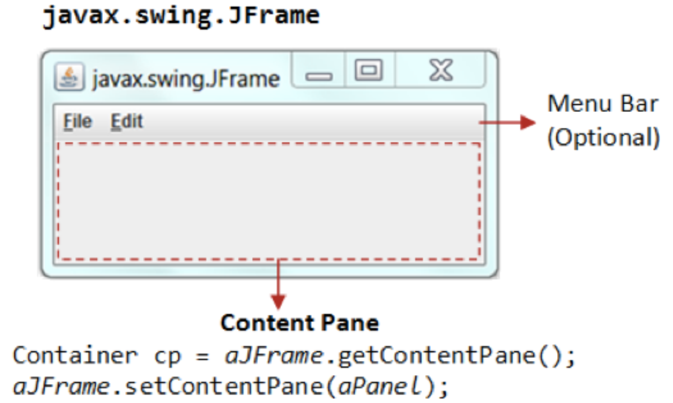


Figure 15.6: JFrame content pane.

Call the getContentPane() method to return the current JFrame object. This is useful with dialogs because they generally close after being used.

Refresh

Call the validate() method to refresh the layout should it be modified.

15.6 Creating and adding components

Creating components

Figure 15.7: Available components.

The figure above shows some of the available components when building a GUI that can be created using the code below.

```
JButton button = new JButton("Click␣me!");
JLabel label = new JLabel("This␣is␣a␣JLabel");
JTextField textField1 = new JTextField("This␣is␣the␣initial␣text");
JTextArea textArea1 = new JTextArea("This␣is␣the␣\n␣initial␣text");
JCheckBox checkbox = new JCheckBox("Label␣for␣checkbox");
JRadioButton radioButton1 = new JRadioButton("Label␣for␣button");
```

Creating components

Adding components

Once components have been created, they must be added to the `JFrame` to be visible.

As seen before, a `JFrame` can be incorporating the `JFrame` in to a Java program by:

- creating the `JFrame` in a class; or
- extending the class to `JFrame`.

There are slight differences in the methods for adding components based on how the `JFrame` was incorporated.

```
JFrame jf = new JFrame();
JButton b = new JButton("OK");

jf.add(b);
jf.setSize(300, 300);
jf.setVisible(true);
jf.setType(Type.UTILITY);
jf.setDefaultCloseOperation(EXIT_ON_CLOSE);
```

Adding a button component in a class where the JFrame was created.

```
JButton b = new JButton("OK");

add(b);
setSize(300, 300);
setVisible(true);
setType(Type.UTILITY);
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

Adding a button component in a class that extends from JFrame.

15.7 Layout managers

The most important layout managers include:

- `BorderLayout`;
- `FlowLayout`;
- `GridLayout`; and
- `BoxLayout`.

BorderLayout

Characteristics

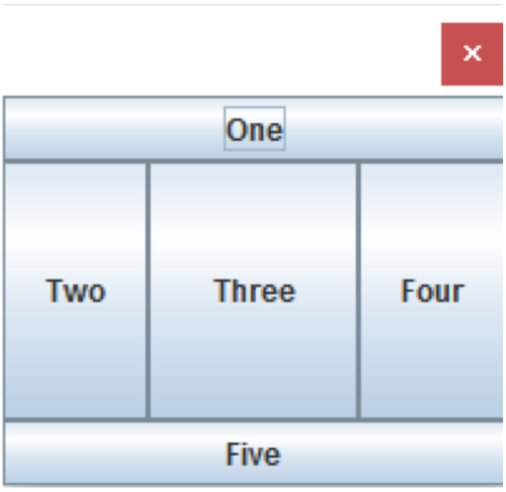


Figure 15.8: BorderLayout example.

The BorderLayout layout manager:

- provides five areas where components can be added; and
- is the default layout manager for a JFrame.

Code

```
public class BorderLayoutExample extends JFrame {
   public BorderLayoutExample() {
      setLayout(new BorderLayout());

      add(new JButton("One"), BorderLayout.NORTH);
      add(new JButton("Two"), BorderLayout.WEST);
      add(new JButton("Three"), BorderLayout.CENTER);
      add(new JButton("Four"), BorderLayout.EAST);
      add(new JButton("Five"), BorderLayout.SOUTH);
   }
}
```

A BorderLayout class.

FlowLayout

Characteristics

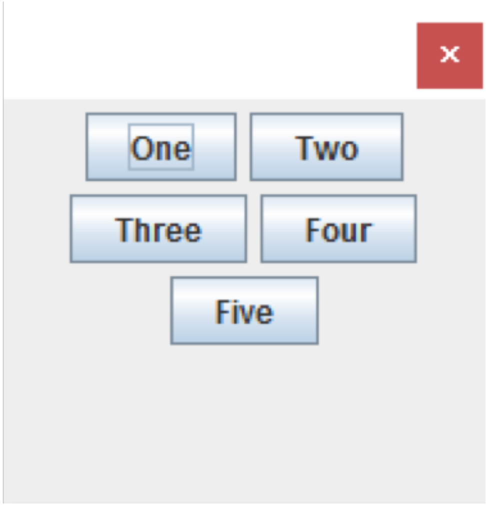


Figure 15.9: FlowLayout example.

The FlowLayout layout manager allows components to be added to left to right and top to bottom.

Code

```
public class FlowLayoutExample extends JFrame {
   public FlowLayoutExample() {
      setLayout(new FlowLayout(FlowLayout.CENTER));

      add(new JButton("One"));
      add(new JButton("Two"));
      add(new JButton("Three"));
      add(new JButton("Four"));
      add(new JButton("Five"));
   }
}
```

A FlowLayout class.

GridLayout

Characteristics

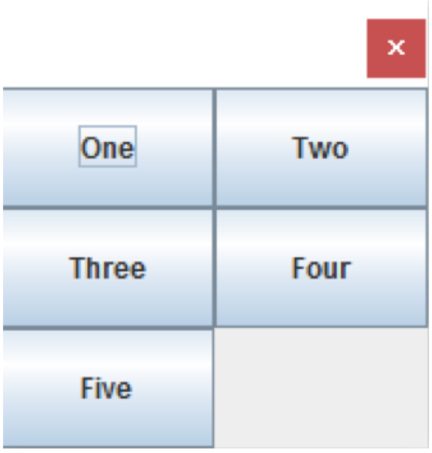


Figure 15.10: GridLayout example.

The GridLayout layout manager:

- allows components to be added in a rectangular grid;
- has areas of all the same size and shape.

Code

```
public class GridLayoutExample extends JFrame {
   public GridLayoutExample() {
      setLayout(new GridLayout(3, 2));

      add(new JButton("One"));
      add(new JButton("Two"));
      add(new JButton("Three"));
      add(new JButton("Four"));
      add(new JButton("Five"));
   }
}
```

A GridLayout class.

BoxLayout

Characteristics

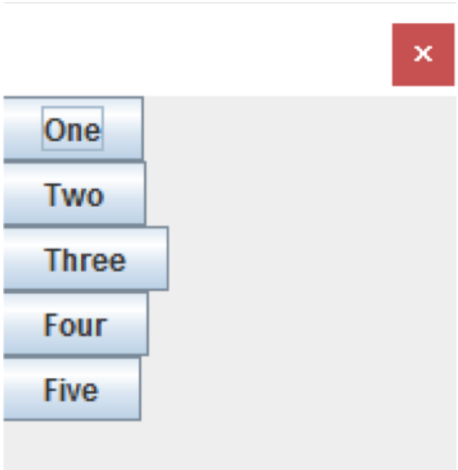


Figure 15.11: BoxLayout example.

The BoxLayout layout manager allows a horizontal row or vertical column of components to be created.

Code

```
public class BoxLayoutExample extends JFrame {
   public BoxLayoutExample() {
      setLayout(new BoxLayout(getContentPane(), BoxLayout.Y_AXIS));

      add(new JButton("One"));
      add(new JButton("Two"));
      add(new JButton("Three"));
      add(new JButton("Four"));
      add(new JButton("Five"));
   }
}
```

A BoxLayout class.

Nested layouts using `JPanels`

How are `JPanels` used?

A `JPanel` is both a container and component:

- as it is a container, other components can be placed within; and
- as it is a component, other containers can be placed within.

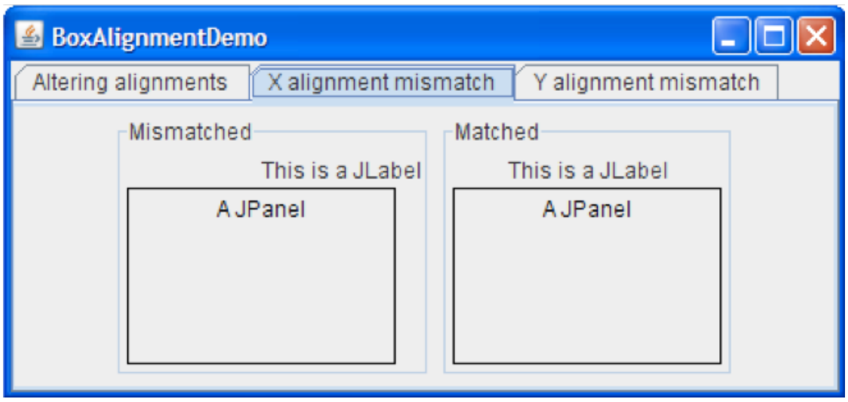


Figure 15.12: JPanel example.

Most Java GUIs are built by:

- creating several JPanels;
- arranging the JPanels; and
- adding components to the JPanels.

The JPanels are then added in to the JFrame (container).

As shown in the code below, each JPanel may have a different layout manager.

```
public class NestedLayoutExample extends JFrame {
   public NestedLayoutExample() {
      // JPanel using the BorderLayout layout manager
      JPanel p1 = new JPanel();
      p1.setLayout(new BorderLayout());
      p1.add(new JButton("A"), BorderLayout.NORTH);
      // code for more buttons B, C, D, E

      // JPanel using the GridLayout layout manager
      JPanel p2 = new JPanel();
      p2.setLayout(new GridLayout(3,2));
      p2.add(new JButton("F"));
      // code for more buttons G, H, I, J, K

      // JPanel using the BoxLayout layout manager
      JPanel p3 = new JPanel();
      p3.setLayout(new BoxLayout(p3,BoxLayout.Y_AXIS));
      p3.add(new JButton("L"));
      // code for more buttons M, N, O, P

      // Add each JPanel to the JFrame
      setLayout(new BorderLayout());
      add(p1, BorderLayout.CENTER);
      add(p2, BorderLayout.SOUTH);
      add(p3, BorderLayout.EAST);
   }
}
```

A class that uses nested layouts.

The code above is able to produce a GUI with:

- a `JFrame` that contains multiple `JPanel`s; and
- multiple `JPanel`s each with a different layout manager.

The resulting GUI produced by this code is shown below.

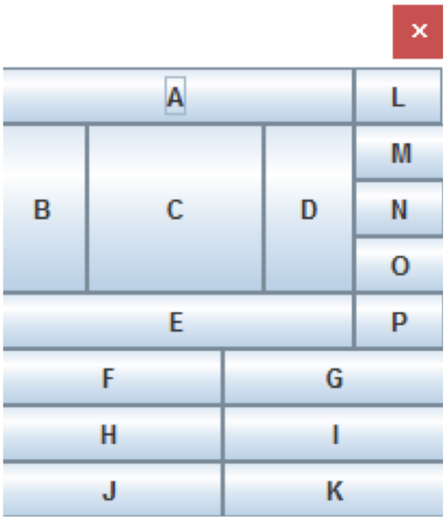


Figure 15.13: `JPanel` example.

15.8 Action listeners

Creating action listeners

Action listeners are supported from the `java.awt.event` package.

Creating and attaching action listeners to the components allows actions to be performed after user interaction with the GUI. They react to events from the components.

There are two methods of creating an action listener:

- creating a class that implements the `ActionListener` interface; or
- creating an anonymous inner class.

Once the action listener has been created, it must be added to the component using the `addActionListener` method.

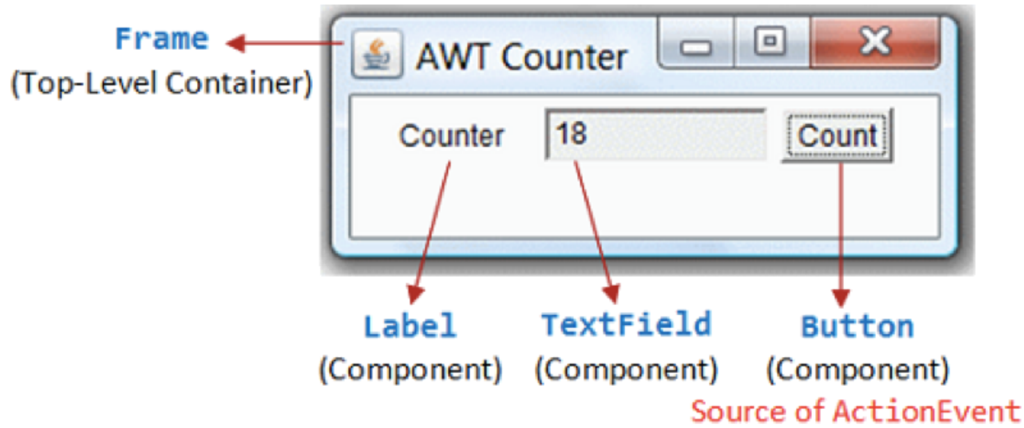


Figure 15.14: JFrame with the Count button.

```
class MyCountListener implements ActionListener{
   public void actionPerformed(ActionEvent event) {
      // some code to handle countButton click
   }
}

JButton countButton = new JButton("Count");
countButton.addActionListener(new MyCountListener());
```

Creating an action listener by creating a class that implements the ActionListener interface.

```
JButton countButton = new JButton("Count");
countButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
         // some code to handle countButton click
      }
   }
)
```

Creating an action listener by creating an anonymous inner class.

Setting and getting values

Some user actions simple set values or options that will be used later, such as:

- entering text;
- setting a checkbox; and
- setting a radio button.

It is possible to read these values out of the components by using the component's methods, as shown in the code below.

```
String myText = myJTextField.getText();
```

Reading value from a text field.

```
boolean checked = myJCheckBox.isSelected();
```

Reading value from a checkbox.

```
boolean selected = myJRadioButton.isSelected();
```

Reading value from a radio button.

15.9 Dialogs

The most important dialogs include:

- message dialogs;
- confirm dialogs;
- input dialogs;
- load file dialogs; and
- save file dialogs.

Message dialog

Figure 15.15: Message dialog.

```
JOptionPane.showMessageDialog(getContentPane(), "This is a \" message \" dialog.");
```

Creating a message dialog.

As shown in the code above:

- `showMessageDialog` is a static method of `JOptionPane`; and
- the first argument for the `showMessageDialog` method is the container (leaving this `null` would cause the message to be displayed outside of the window); and

- the first argument for the `showMessageDialog` method is the text to be displayed as the message.

Message dialogs are often used to inform the user of an occurring process or error.

Confirm dialog

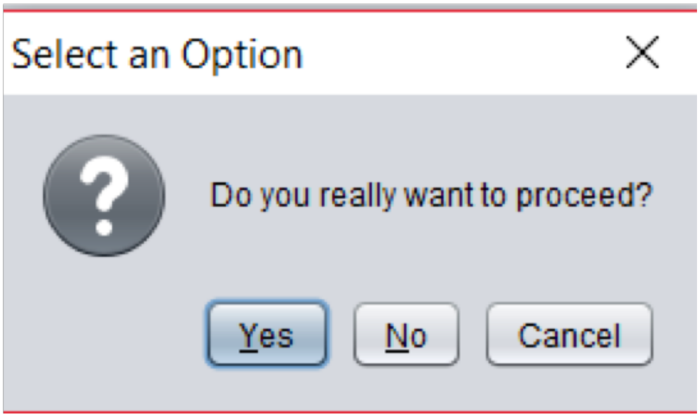


Figure 15.16: Confirm dialog.

```
int YesNo = JOptionPane.showConfirmDialog(getContentPanel(), "Do you really want to proceed?");
```

Creating a confirm dialog.

As shown in the code above:

- `showConfirmDialog` is a static method of `JOptionPane`; and
- the first argument for the `showConfirmDialog` method is the container (leaving this `null` would cause the message to be displayed outside of the window); and
- the first argument for the `showConfirmDialog` method is the text to be displayed as the accompanying message to the options.

In addition, the `showConfirmDialog` returns:

- 0 if Yes is clicked; and
- 1 in any other case.

This means that subsequent control of execution can be informed by the returned value from the `showConfirmDialog` method, as shown below.

```
if (YesNo == JOptionPane.YES_OPTION)
{
   // code to execute if user clicked 'Yes'
} else {
   // code to execute if user did not click 'Yes'
}
```

Using return value from showConfirmDialog.

Confirm dialogs are often used to inform the user of an occurring process or error and ask if they wish to proceed.

Input dialog

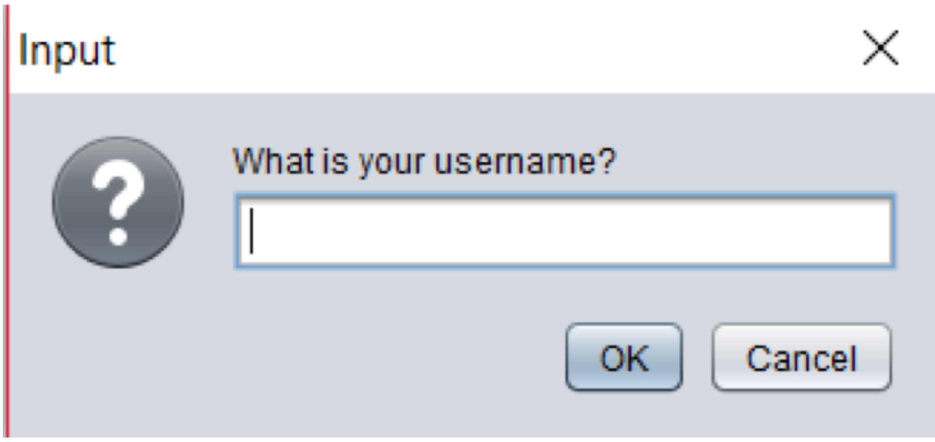


Figure 15.17: Input dialog.

```
String userName = JOptionPane.showInputDialog(getContentPanel(), "What is
    your username?");
```

Creating an input dialog.

As shown in the code above:

- showInputDialog is a static method of JOptionPane; and
- the first argument for the showInputDialog method is the container (leaving this null would cause the message to be displayed outside of the window); and
- the first argument for the showInputDialog method is the text to be displayed as the accompanying message to the input box.

In addition, the showInputDialog returns a String containing the user's input.

Input dialogs are used to obtain input data, such as a username, from a user.

Load file dialog

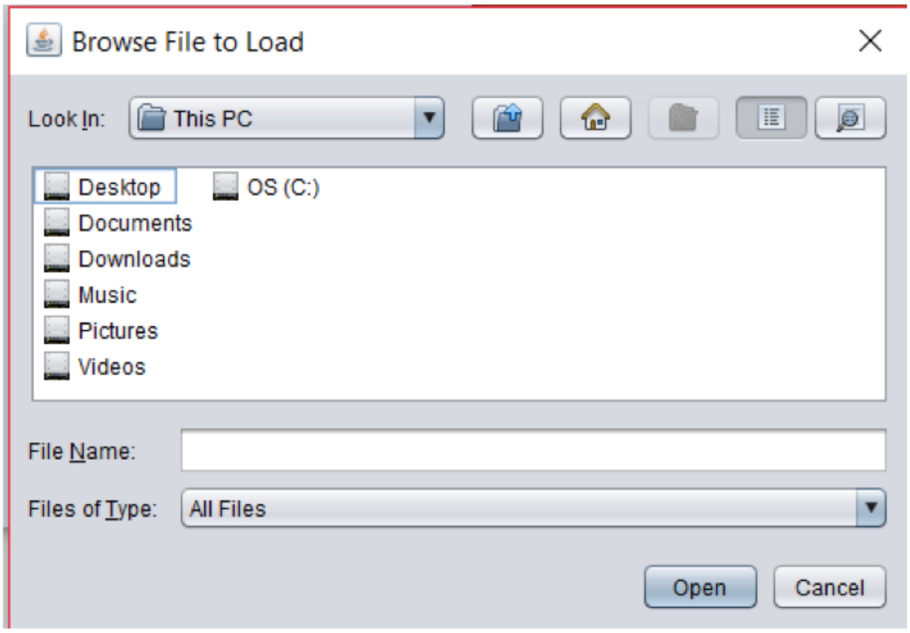


Figure 15.18: Load file dialog.

```
JFileChooser ch = new JFileChooser();
ch.setDialogTitle("Browse␣File␣to␣Load");

int result = ch.showOpenDialog(getContentPane());

if (result == JFileChooser.APPROVE_OPTION) {
   File file = ch.getSelectedFile();

   String fileName = file.getName();
   String filePath = file.getPath();

   // Code to load the file
}
```

Creating a load file dialog.

As shown in the code above:

- the `JFileChooser` component is used to create a load file dialog;
- the `showOpenDialog` is used to create a load file dialog using the file chooser dialog;
- the `getSelectedFile()` method returns a `File` object; and
- the `File` object has methods such as `getName()` and `getPath()` etc.

File open dialogs are often used to allow the user to select a local file to be opened later in the code.

Save file dialog

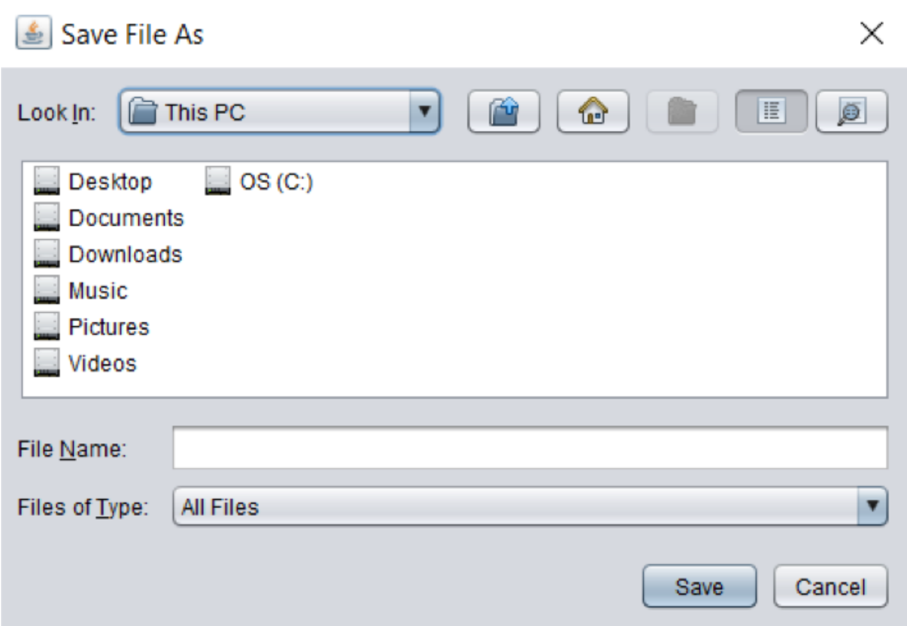


Figure 15.19: Save file dialog.

```
JFileChooser ch = new JFileChooser();
ch.setDialogTitle("Save File As");

int result = ch.showSaveDialog(getContentPane());

if (result == JFileChooser.APPROVE_OPTION) {
   File file = ch.getSelectedFile();

   String fileName = file.getName();
   String filePath = file.getPath();

   // Code to save to the file
}
```

Creating a save file dialog.

As shown in the code above:

- the `JFileChooser` component is used to create a file chooser dialog;
- the `showSaveDialog` is used to create a save file dialog using the file chooser dialog;
- the `getSelectedFile()` method returns a `File` object; and
- the `File` object has methods such as `getName()` and `getPath()` etc.

File open dialogs are often used to allow the user to select a local file to be opened later in the code.

16. Application Layer Protocols

16.1 Introduction

The **application layer** is the fifth layer in the TCP/IP stack and uses protocols to format the data in a way which allows it to be used by the receiving application.

Different protocols exist, such as DNS, HTTP, FTP, SMTP and TELNET, to support different applications such as:

- electronic mail system;
- World Wide Web;
- multimedia; and
- remote desktop.

Most protocols resort to the client-server model.

16.2 Domain Name System (DNS)

Definition

A **Domain Name System (DNS)** provides the rules for assigning domains names to IP addresses by structuring the domain names into a hierarchy of smaller domains, the smaller domains are written as a string and separated by full stops.

URLs

Definition

A **Uniform Resource Locator (URL)** is the complete address of an Internet resource.

Structure

Figure 16.1: URL structure

The figure above shows that a URL is composed of a:

- method – the protocol used to connect;
- host – the resources that are being used;
- location – the specific resource being requested; and
- element – a portion of the requested resource.

Why are URLs useful?

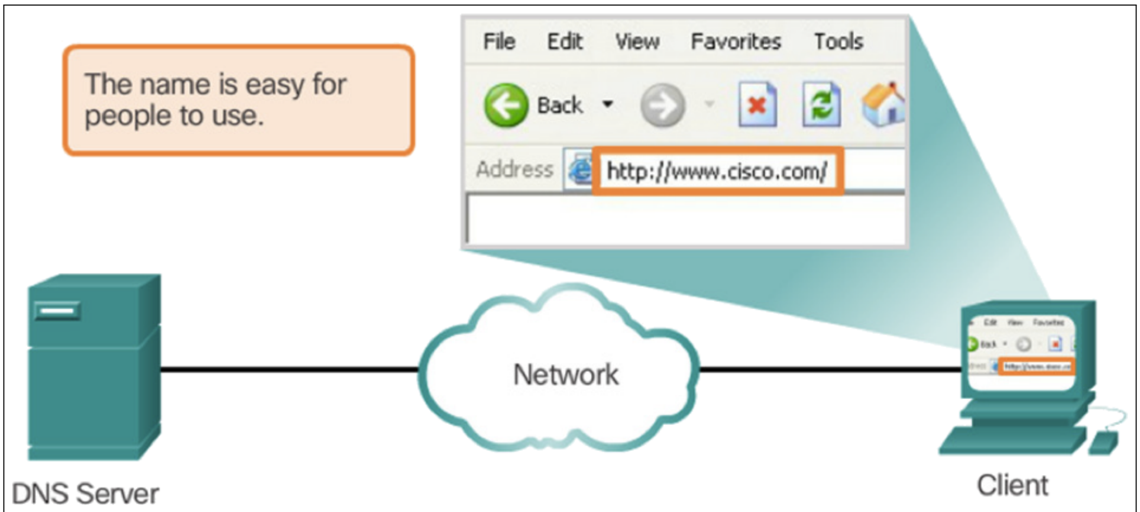


Figure 16.2: Human-readable URL.

Networks refer to hosts by IP addresses, such as 198.133.219.25. However, users would rather use ASCII strings.

Internet registrars

An **Internet registrar** holds records of all existing website names and the details of those domains that are available to purchase. They act as resellers for domain names and must be accredited by their governing registry.

Internet registries

Internet registries are five global organisations which are governed by the Internet Corporation for Assigned Names and Numbers (ICANN) which host worldwide databases containing the records of all the domain names currently issued to individuals and companies and the individual's or company's details.

The details kept by internet registries include:

- name;
- type — individual or company;
- registered mailing address;
- registrar which sold the domain; and
- date of registry.

Internet registries also allocate IP addresses and keep records of which address or addresses a domain name is associated with as part of their DNS.

How it works

Aim

DNS aims to convert URLs to IP addresses. (This can be seen by performing `nslookup` on a command line.)

Hierarchy

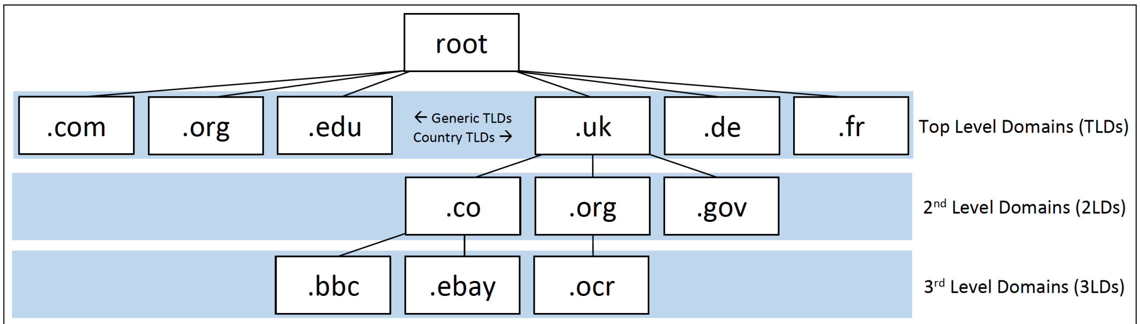


Figure 16.3: DNS hierarchy.

As seen in the figure above, the DNS name space is hierarchical; forming a tree of domains.

At the top level, there are two types of domains:

- generic; and
- international.

Generic domains include:

- `com` (commercial);
- `edu` (educational);
- `gov` (U.S government);
- `mil` (U.S military);

- org (non-profit organisations); and
- net (network providers) etc.

International domains include:

- uk;
- gr; and
- jp etc.

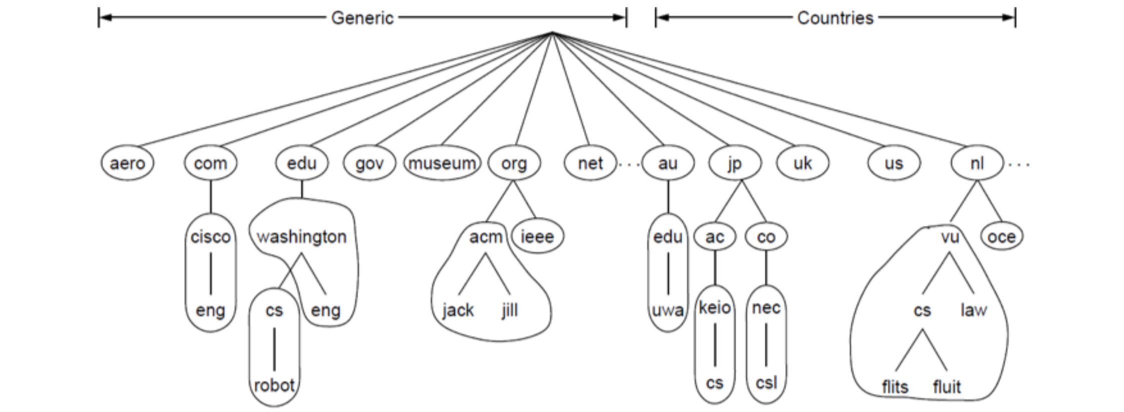


Figure 16.4: DNS hierarchy zones.

As seen in the figure above, the DNS name space is divided in to zones. Each zone covers part of the tree and is associated with name servers.

Name servers

A **name server** contains a database of corresponding domain names and IP addresses.

Each domain name has one or more equivalent IP addresses. A DNS server must then retrieve the correct corresponding IP address.

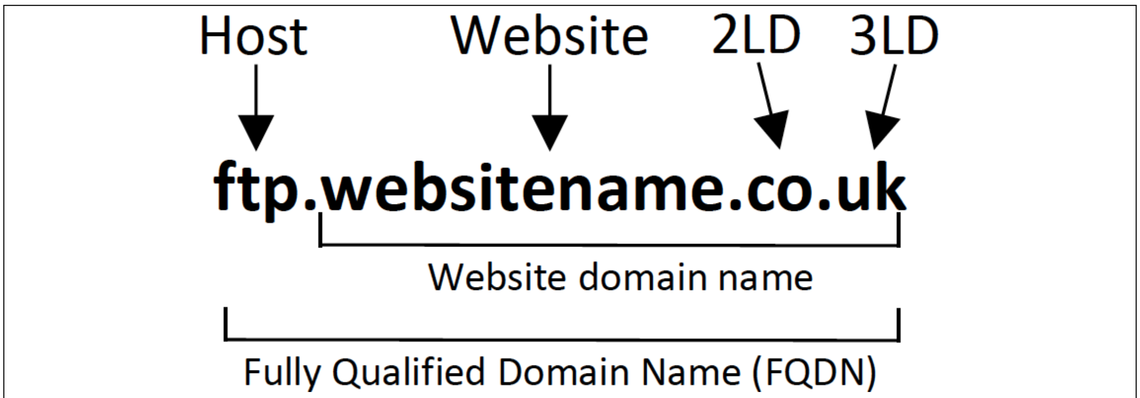


Figure 16.5: Domain name structure.

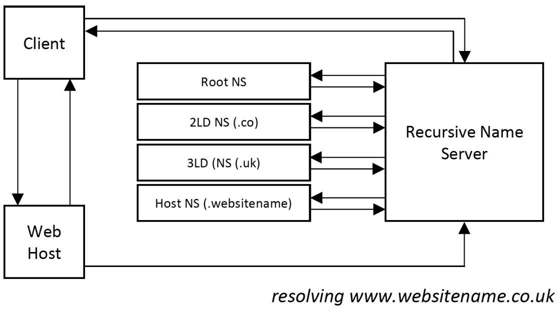
A **resolver** is a library procedure used to map a DNS name to an IP address.

When an DNS name is used:

- the application calls the resolver with the DNS name as a parameter;

- the resolver sends a packet to a name server;
- the name server looks up the name and returns the IP address to the resolver;
- the resolver returns the IP address to the application; and
- the application can then use the IP address for communication.

This can be applied to requesting a web page, as seen in the figure below.



- Client inputs a domain name into their web browser.
- Web browser cache is checked for matching web page.
- If web page is not in the cache, domain is transferred to the recursive name server, which is owned by the Internet Service Provider (ISP) and contains the domain names and corresponding IP addresses for web pages previously requested.
- If the web page is not in the recursive name server, the recursive name server will perform queries on the root, 2LD name server, 3LD name server and the web host's name server until a match is found and the domain has been resolved.
- When the domain has been resolved, the name server containing the domain will return the IP address to the recursive name server that returns the IP address to the client's web browser.
- The client's web browser makes a data request to the web host server, if the page is found, the web elements, such as: HTML; CSS; and JavaScript, will be transmitted back to the client's web browser and displayed to the client.

Figure 16.6: Requesting a web page at www.websitename.co.uk.

16.3 World Wide Web (WWW)

Definition

The **World Wide Web (WWW)** is software which is governed by the HyperText Transfer Protocol (HTTP) and comprises of web pages that reside on computers connected to the Internet. It is dependent on the Internet as a service to communicate the information contained within these pages. This is accessed through a web browser.

This is different to the **Internet**, which is the physical connection between computer systems and networks and comprises of network of computers, copper wires, fibre-optic cables and wireless networks. The Internet is independent of the World Wide Web (WWW).

Background

The WWW was invented in 1989 by English scientist Tim Berners-Lee.

The first prototype for the WWW was created in late 1990 and is still available at `http://info.cern.ch/`.

The modern WWW contains a vast collection of electronic documents/web pages.

Nature

Pages are viewed with a program called a **browser**. Each page may contain links to other pages anywhere in the world.

Mosaic was first graphical interface browser was launched in 1993. Followed by Netscape Communication in 1994.

Current widespread browsers include:

- Chrome;
- Internet Explorer;
- Firefox; and
- Safari.

16.4 HyperText Transfer Protocol (HTTP)

Definitions

HyperText Transfer Protocol (HTTP) defines how messages are formatted and transmitted, and what actions web servers and browsers should take in response to various commands.

Websites using HTTP have a URL beginning with ”`http://`”.

HyperText Markup Language (HTML) is a markup language which uses tags to define elements and how the elements are presented on a web page, such as text, images and hyperlinks.

Features of HTTP

Browsers interpret the content of web documents and display the results to the user. Web documents are written in HTML.

HTTP has two main features:

- markup commands – describe how the text should appear on the user’s screen, for example `<b>Hello World</b>` would display the text ”Hello World” in a bold font; and
- hyperlinks – when a user clicks on the corresponding test, the browser will follow the link, for example `<a href="http://www.ntu.ac.uk">NTU</a>` will produce the text ”NTU” and clicking on this will cause the browser to load the web page http://www.ntu.ac.uk/.

Process for requesting a web page

As seen before, URLs are entered by users or present in links and reference a web server. The URL allows the web browser to establish a connection to the web server.

Given the URL `http://cisco.com/index.html`, its parts can be determined:

- the protocol or scheme – `http`;
- the server name – `www.cisco.com`; and
- the specific file name requested – `index.html`.

The process for requesting this web page in a browser using HTTP is as follows.

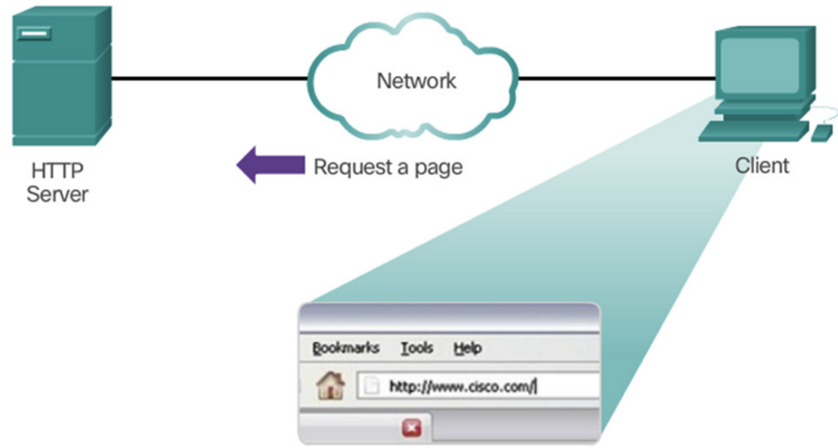


Figure 16.7: Translate URL to IP address.

As seen before, using DNS, the server name portion of the URL is translated to the associated IP address before the server can be contacted.

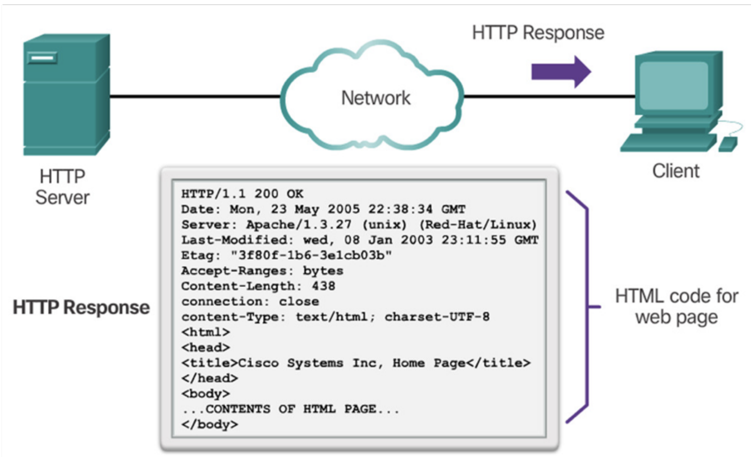


Figure 16.8: GET request.

Next, the browser must establish a TCP connection to port 80 on the server. Once the connection has been established, the browser and server can then exchange data. Commands from the browser include:

- GET – requests to read a web page;
- HEAD – requests to read a web page's header; and
- PUT – requests to store a web page.

The server's response will include a status line, and optional lines such as the requested page itself. The status lines can include various codes, as shown below.

Status Code	Message	Explanation
200	OK	Successful request, the webpage exists.
400	Bad request.	
401	Unauthorised.	Unauthorised request, authentication required.
403	Forbidden	Access to this page or directory is not permitted.
404	Not found.	Page does not exist.
500	Internal server error.	Often caused by an incorrect server configuration.

Figure 16.9: HTTP status codes.

The browser sends a GET request to the server's IP address and requests the `index.html` file that was specified in the URL.

If the request was successful (status code 200), the server then sends the requested file to the client, this contains the HTML code for this web page.

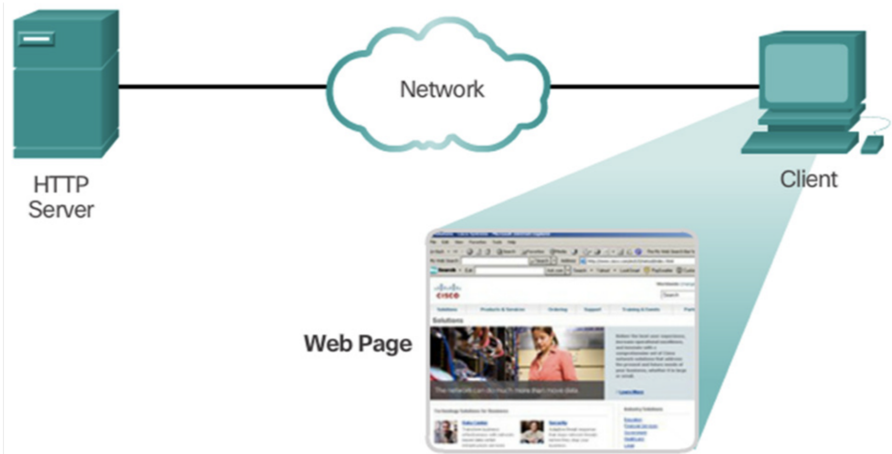


Figure 16.10: GET request.

Lastly, the browser processes the HTML code and formats the web page for the browser window based on the code in the file.

HTTP request

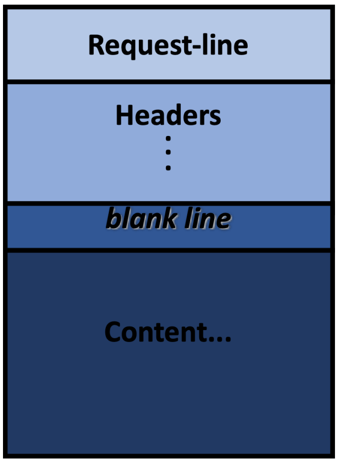


Figure 16.11: HTTP request.

A HTTP request is composed of:

- a request-line;
- headers;
- a blank line; and
- the content.

```
<method><SP><resource id><SP><HTTP version><crlf>
[<Header> : <value>] <crlf>
// ...
[<Header> : <value>] <crlf>
            blank line <crlf>


[Entity body]

GET /path/file.html HTTP/1.0
Accept: text/html
User-Agent Mozilla/40
```

Example HTTP request.

HTTP response

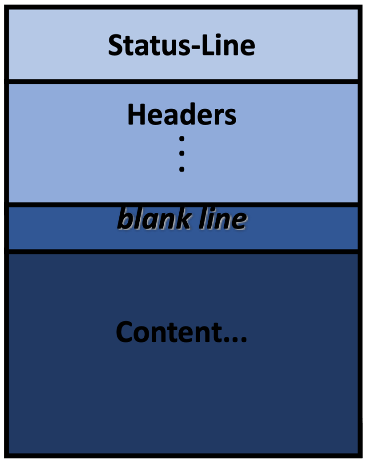


Figure 16.12: HTTP response.

A HTTP response is composed of:

- a status-line;
- headers;
- a blank line; and
- the content.

```
<HTTP Version><SP><result code><SP><explanation><clrf>
[<Header> : <value>] <crlf>
// ...
[<Header> : <value>] <crlf>
            blank line <crlf>


[Entity body]

HTTP/1.0 200 OK
Server: Apache/1.17
Content-Type: text/html
Content-Length :2000

<HTML>
   // ...
</HTML>
```

Example HTTP response.

HTTPS

HyperText Transfer Protocol Secure (HTTPS) is defines how messages are formatted and transmitted in a **secure connection**, and what actions web servers and browsers should take in response to various commands.

Websites using HTTPS have a URL beginning with "`https://`".

Modern web servers have more features such as authentication and access control, provided by HTTPS.

16.5 TELNET

Definition

TELNET is a protocol that provides a bidirectional interactive text-oriented communication facility using a virtual terminal connection.

How it works

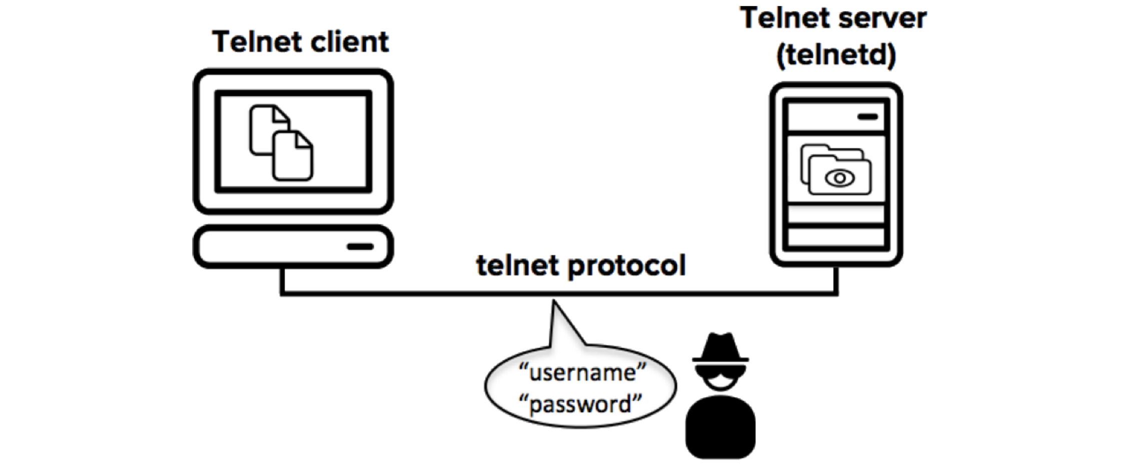


Figure 16.13: TELNET client-server connection.

TELNET provides a remote desktop feature that often requires a username and password, as shown in the figure above.

Remote login applications are general-purpose client-server applications that allow operations such as:

- transferring files; and
- sending messages.

Using TELNET

Windows has a program called telnet that supports the TELNET protocol.

Connecting to a remote host using the telnet program can be achieved by:

- using the domain name – telnet <domain name>; or
- using the IP address – telnet <IP address.

Evaluation

Advantages	Disadvantages
Remote access to software that is only available on the remote host.	**Insecure** over an open network such as the Internet.
Remote access to hardware that is only available on the remote host.	

Network Virtual Terminal (NVT)

A **Network Virtual Terminal (NVT)** is a communications concept describing a variety of data terminal equipment (DTE), with different data rates, protocols, codes and formats, accommodated in the same network.

For example, each OS on the computer systems may accept different keystrokes to perform actions, such as the EOF action:

- Ctrl+Z in MS-DOS; and
- Ctrl+D in Unix-based OSs.

TELNET implements an NVT. This means that when keystrokes are sent by the user, they are first converted to a standard format, as seen in the figure below.

The user sends keystrokes to the terminal where the local operating system accepts the characters but does not interpret them.

↓

The characters are sent to the TELNET client.

↓

The TELNET client transforms the characters to a universal character set called **NVT characters**.

↓

A TCP connection is established on port 23 between the TELNET client and TELNET server.

↓

The stream of characters is sent from the TELNET client to the TELNET server.

↓

The remote host's OS receives the characters and sends them to the TELNET server.

↓

NVT formats the characters to the correct characters that can be understood by the OS.

Figure 16.14: Process converting keystrokes to a standard format.

The TELNET protocol specifics several NVT characters to control the details of the interaction between the TELNET client and TELNET server.

Name	Decimal code	Meaning
Interpret as Command (IAC)	255	Precedes any other command to inform the system that the character should be treated as a command.
Interrupt Process (IP)	244	Abort the process to which the NVT is connected.
Abort Output (AO)	245	Allows the current process to run to completion but do no send its output to the client.
Are You There (AYT)	246	Check to see if the system is still running.
Erase Character (EC)	247	The receiver should delete the last character sent.
Erase Line (EL)	248	Delete all characters in the current line.

Figure 16.15: NVT characters for control commands.

The figure above shows the available control commands provided by NVT characters. It must be noted that all commands must be preceded by the Interpret as Command (IAC) character.

16.6 Email protocols

Email clients and servers

Email servers communicate with other mail servers to transport messages from one domain to another.

Email clients do not communicate directly when sending emails.

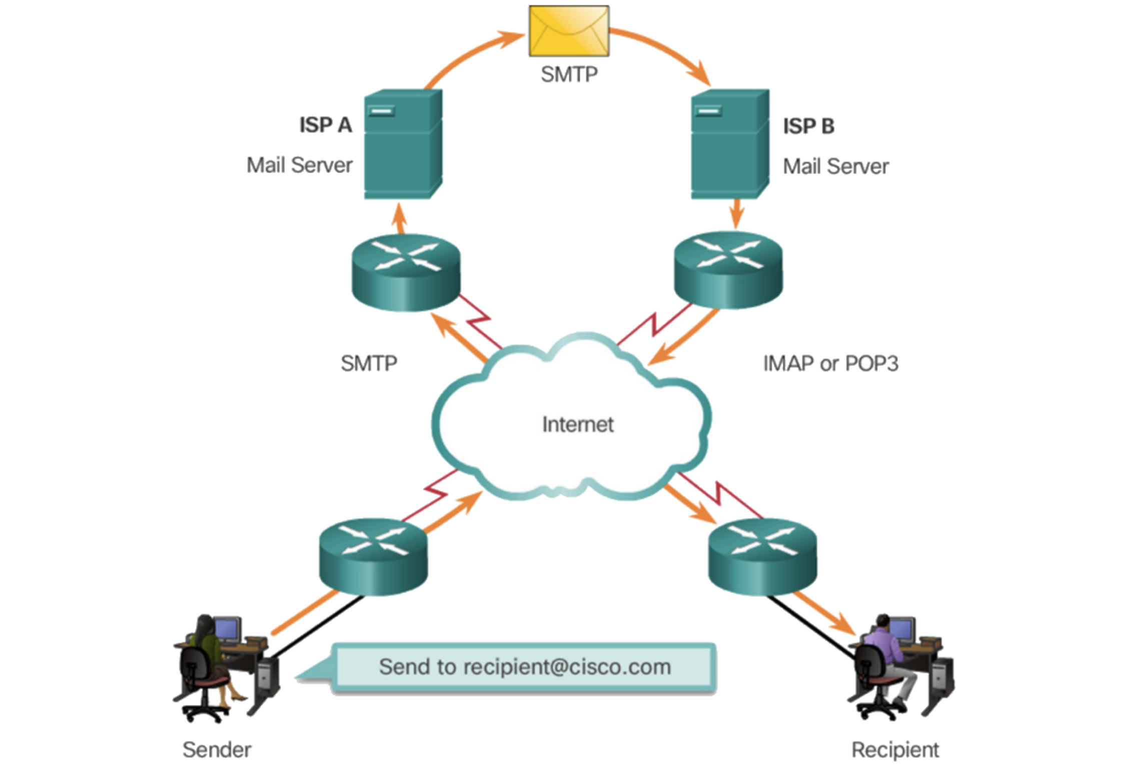


Figure 16.16: Email clients and servers.

As shown in the figure above, email relies on three separate protocols for operations:

- sending – SMTP;
- receiving – POP or IMAP.

SMTP

Definition

Simple Mail Transfer Protocol (SMTP) is used by the mail server to send and receive emails from other mail servers. When a client is sending an email, the email is transferred from the email client's computer system to the mail server. An outgoing email is transferred from one mail server to another.

How it works

A SMTP message format requires:

- a message header – must have a properly formatted recipient email address and a sender email address; and
- a body – can contain any amount of text.

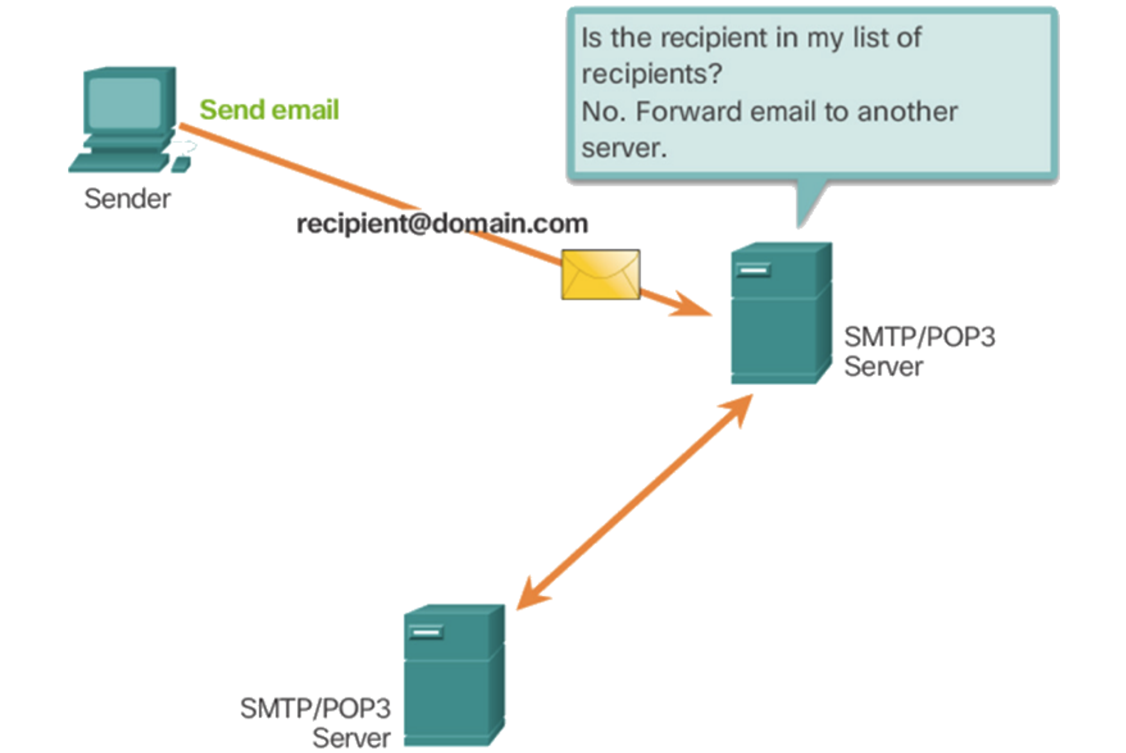


Figure 16.17: Sending an email using SMTP.

As shown in the figure above, an SMTP client connects to a SMTP server to send an email message. This process is outlined in the figure below.

The SMTP client establishes a TCP connection to the SMTP server on port 25. ↓ The SMTP server receives the message and stores it in a local mailbox or relays the message to another mail server. ↓ Users use email clients to retrieve messages stored on the server.
Figure 16.18: Process for sending an email using SMTP.

The email clients discussed in the figure above commonly use either the IMAP or POP protocol to retrieve messages.

Evaluation

Advantages	Disadvantages
Simple for the email client since the only required inputs is the recipient's email address.	**Little security** because SMTP is text-based and any text can be spoofed, therefore it may be necessary to setup and maintain firewalls.
Quick email delivery because SMTP is developed from a simple platform.	
Reliable because if an email fails to be delivered successfully, the email will be marked as temporarily failed and the send operation will be repeated until there is a success or time out.	

POP

Definition

Post Office Protocol v3 (POP3) allows emails from a mail server to be downloaded, deleted or viewed offline by an email client. When an email is received, they are transferred to a local computer system and deleted from the server.

How it works

POP allows for email messages to be downloaded to the client's device and removed from the server.

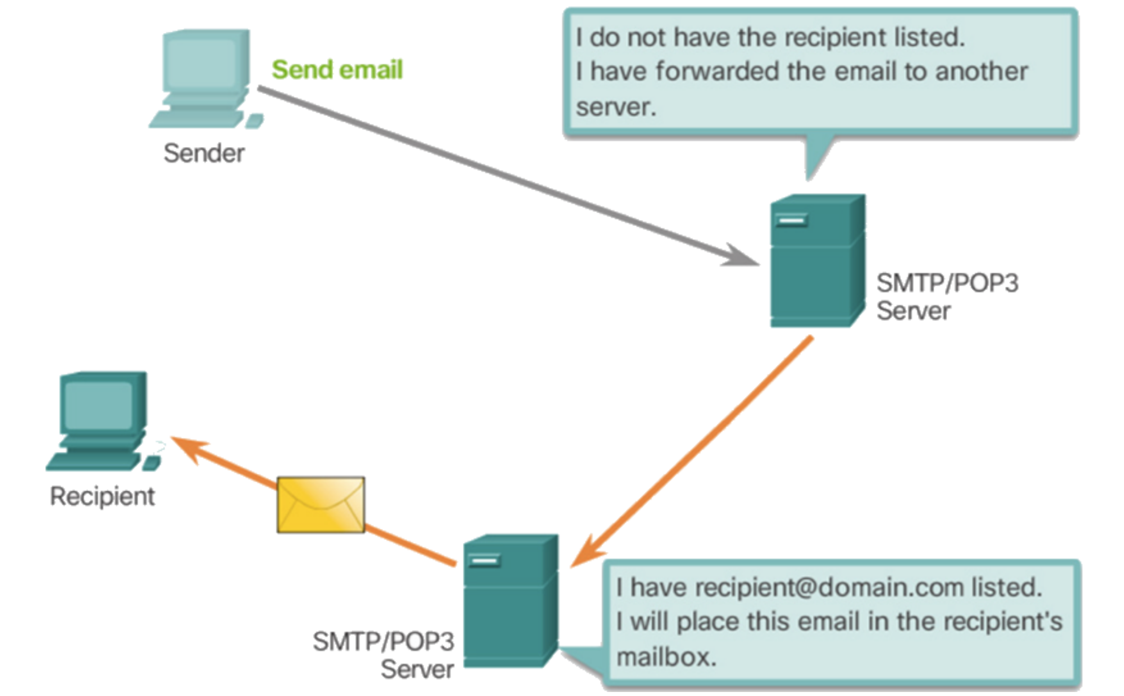


Figure 16.19: Receiving an email using POP.

As shown in the figure above, there is no centralised location where email messages are kept. Instead, a downloaded email message resides on the device that triggered the download. This process is outlined in the figure below.

The POP server listens on TCP port 110 for client requests.

↓

Email clients direct their POP requests to mail servers on TCP port 110.

↓

The POP client and server exchange commands and responses until the TCP connection is closed or aborted.

Figure 16.20: Process for receiving an email using POP.

Evaluation

Advantages	Disadvantages
Emails can be viewed offline because they are downloaded locally once received.	**Can only manage one mailbox**.
Size limit only determined by the size of the computer system's secondary storage since there is no reliance on web-based storage.	**Emails are not synchronised across multiple devices** because they are removed once downloaded.
	Does not support complex searching of emails on the server.

IMAP

Definition

Internet Message Access Protocol (IMAP) allows emails from a mail server to be downloaded, deleted or viewed offline by an email client. When an email is received, they are transferred to a local computer system but also remain on the mail server.

How it works

POP allows for email messages to be displayed to the user rather than downloaded.

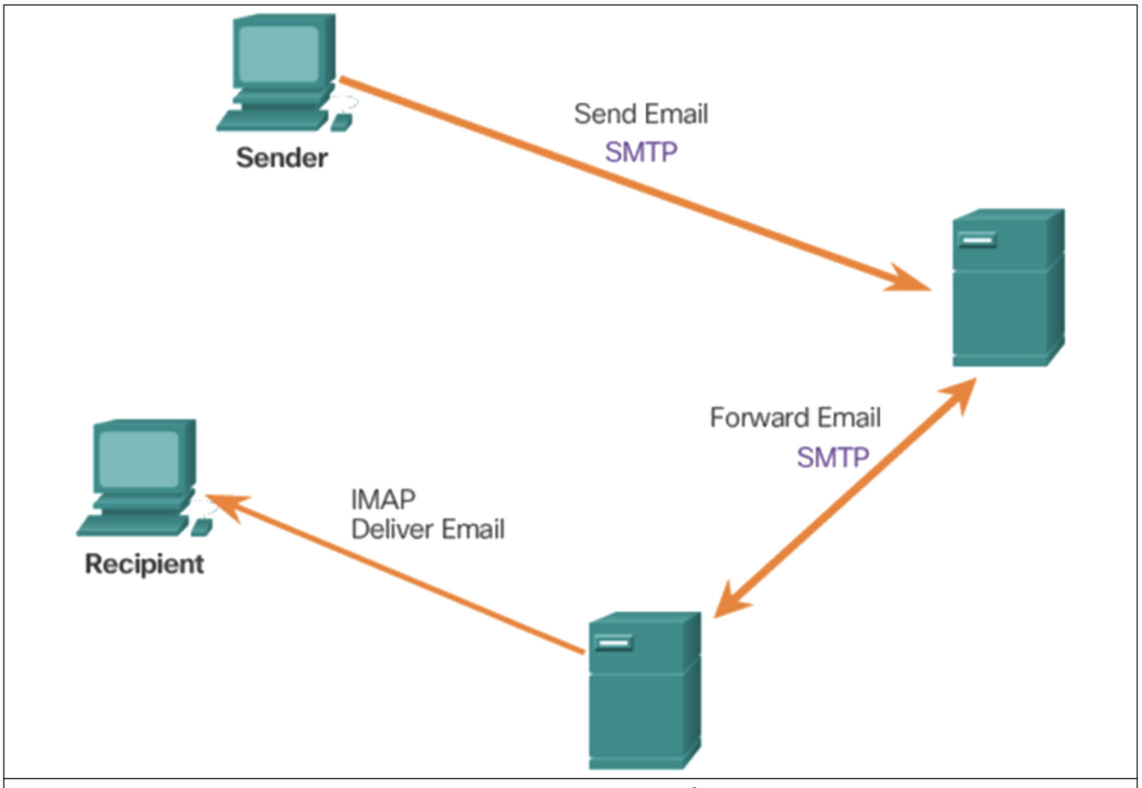


Figure 16.21: Receiving an email using IMAP.

As shown in the figure above, the original email messages reside on the IMAP server until manually deleted by the user and the user views copies of the messages in their email client software.

Users can create a folder hierarchy on the server to organise and store mail. That file structure is displayed on the email client. When a user decides to delete a message, the server synchronises that action and deletes the message from the server.

Evaluation

Advantages	Disadvantages
Can manage more than one mailbox.	**Size limit determined by secondary storage on the mail server**.
Emails are synchronised across multiple devices because they are kept on the mail server and flags are set based on email status, such as read or deleted.	**Difficult to maintain** because it requires use of the host's secondary storage space, storage quotas may be needed to restrict email client's storage usage.
Email clients can choose to download headers, full body or attachments.	
Supports complex searching, such as criteria matching headers, full body or attachments.	

17. Transport Layer Protocols

17.1 Introduction

The **transport layer** is the fourth layer in the TCP/IP stack uses protocols to provide efficient, reliable and cost-effective services to the processes in the application layer.

These services can be TCP (connection-oriented) or UDP (connectionless) and offer functions such as:

- sequencing;
- error-checking; and
- flow and congestion control.

17.2 Ports

Definition

A **port** is a 16-bit number that represents a location where data can be sent.

How they work

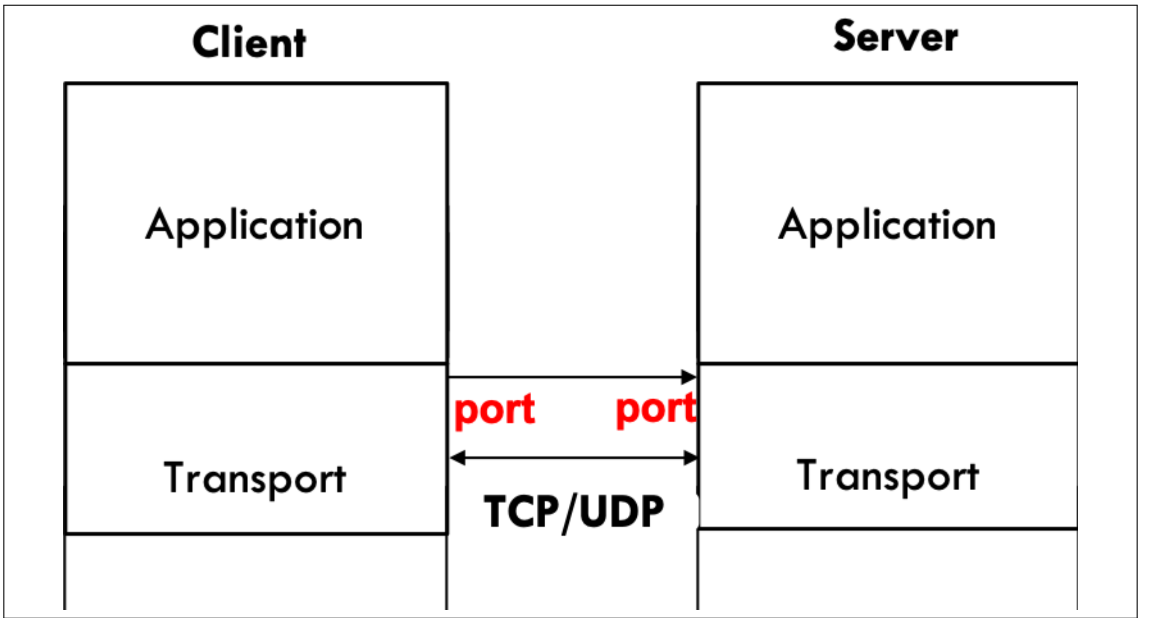


Figure 17.1: Ports in client-server configuration.

As shown in the figure above, clients and servers both use ports to distinguish between the processes transferred data area associated with.

Types of ports

Source/Destination

Port type	Location	Nature	Range
Destination ports	Server machine.	Pre-determined and known to the client.	0–1023 are reserved for well-known services defined by the Internet Assigned Numbers Authority (IANA).
Source ports	Client machine.	Determined dynamically.	Typically a random number greater than 1023.

Registered/Private/Dynamic

Registered ports range from 1024 to 49151.

Private or dynamic ports range from 49152 to 65535.

IANA reserved destination ports

Port number(s)	Protocol	Use
20, 21	FTP	File transfer.
22	SSH	Remote login.
23	TELNET	Remote login.
25	SMTP	Send an email.
53	DNS	Domain name mapping.
69	TFTP	File transfer.
80	HTTP	World Wide Web (WWW).
110	POP	Remote email access.
123	NTP	Clock synchronisation
143	IMAP	Remote email access.
443	HTTPS	Secure World Wide Web (WWW).

17.3 Transport layer headers

TCP and UDP each have their own respective packet headers.

TCP header

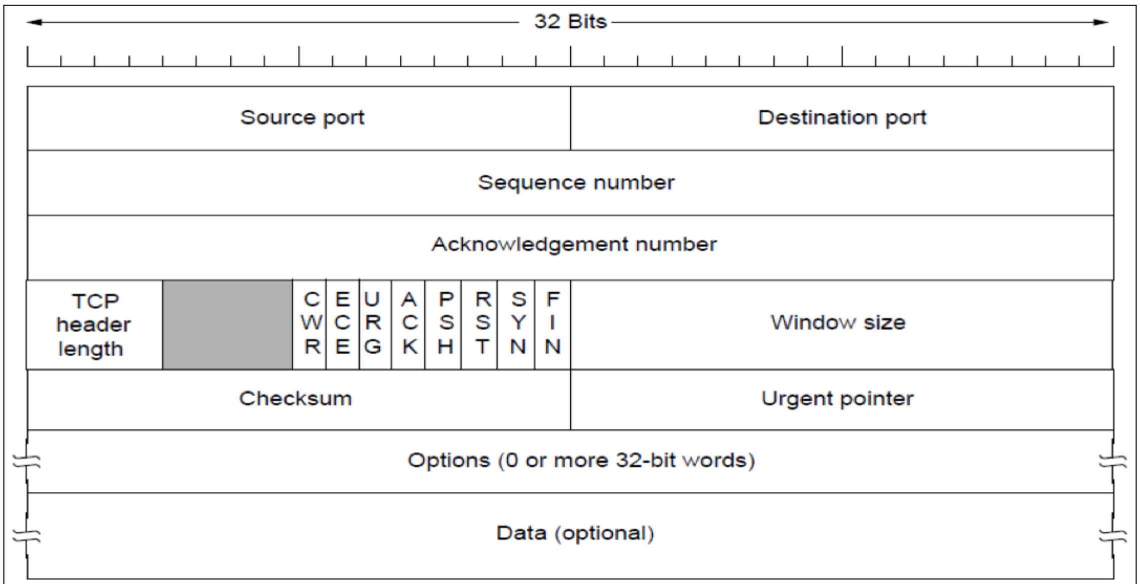


Figure 17.2: TCP header.

As shown in the figure above, the TCP header contains the following fields:

- **source port** – the port number of the local machine;
- **destination port** – the port number of the remote machine;
- **sequence number** – defines the first byte of a segment in a sequence, used to determine the position of the packet in the series of packets;

- **acknowledgement number** – defines the byte number that the receiver of the segment is expecting to receive, this is the sequence number of the next byte the sender expects to receive and therefore provides an acknowledgement of receiving all bytes prior to that;
- **TCP header length** – defines the length of the segment, including the header and options (if any);
- **options fields** – provide a methods for adding extra facilities not satisfied in the regular header, followed by **padding** which is located after the options fields to identify the end of the header;
- **reserved** – 4 bits reserved for future use should the TCP protocol be extended;
- **window size** – shows how many bytes may be sent starting at the byte acknowledged;
- **checksum** – allows error-checking to be performed on the TCP header and the data; and
- **urgent pointer** – used to indicate a byte offset from the current sequence number.

In addition, the TCP header also contains the following bits:

- **CWR bit** – signals to a TCP receiver that the sender has slowed down the transfer rate;
- **ECE bit** – signals to a TCP sender to slow down the transfer rate;
- **URG bit** – enables the *urgent pointer* field;
- **ACK bit** – indicates that the acknowledgement number is valid;
- **PSH bit** – indicates that the receiver requested to deliver the data to the application unbuffered;
- **RST bit** – used to reset a connection that has become confused to due a crash on the host;
- **SYN bit** – used to establish a connection; and
- **FIN bit** – used to release a connection.

UDP header

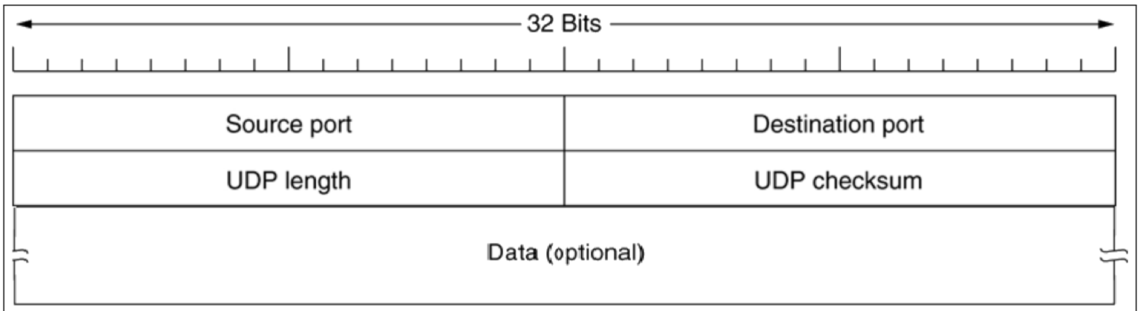


Figure 17.3: UDP header.

As shown in the figure above, the UDP header contains the following fields:

- **source port** – the port number of the local machine;
- **destination port** – the port number of the remote machine;
- **UDP length** – the number of bytes of the segment, including the header and data; and
- **UDP checksum** – used to verify possible data corruption.

17.4 TCP connection sequence

Establishing a connection

Establishing a TCP connection requires a **three-way handshake**. A handshake is an automated process that sets the parameters for communication between two communicating computer systems before packets are transferred.

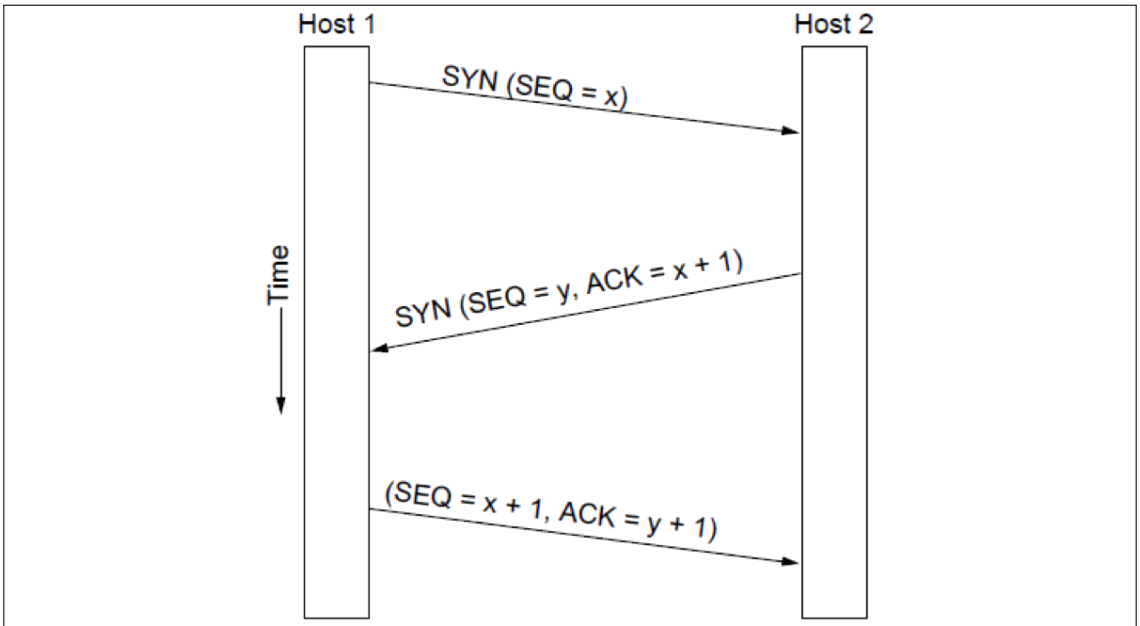


Figure 17.4: Establishing a connection in TCP.

A three-way handshake between `Host 1` and `Host 2` can be seen in the figure above. This process is outlined in the figure below.

Assume that `Host 1` wishes to cease communication with `Host 2`.

↓

`Host 1` sends a connection request (`CR`) to `Host 2`:

- implying that the `SYN` bit is 1; and
- given that the sequence number for `Host 1` is `Host 1`'s initial sequence number.

Given these conditions, it is observed that:

`CR -> SYN=1, SEQ=ISN`

↓

`Host 2` sends an acknowledgement to `Host 1`:

- implying that the `SYN` bit is 1;
- given that the sequence number (`SEQ`) for `Host 1` is now `x+1`; and
- given that the sequence number (`SEQ`) for `Host 2` is now `y`.

Given these conditions, it is observed that:

`CR -> SYN=1, SEQ = y, ACK = x + 1`

↓

`Host 1` sends its first data to `Host 2`:

- given that the sequence number (`SEQ`) for `Host 1` is now `x+1`; and
- given that the sequence number (`SEQ`) for `Host 2` is now `y+1`.

Figure 17.5: Process for establishing a TCP connection.

The process described in the figure above is guaranteed to create a synchronised connection as the `SYN` bit is used to ensure that both `Host 1` and `Host 2` are ready to communicate before any actual data is sent.

As a result, should a delayed packet send a connection request (`CR`), the initial sequence numbers (`ISNs`) will most probably not match and therefore the request will be rejected.

Releasing a connection

A TCP connection can not be released until both hosts have ceased their transmission of data.

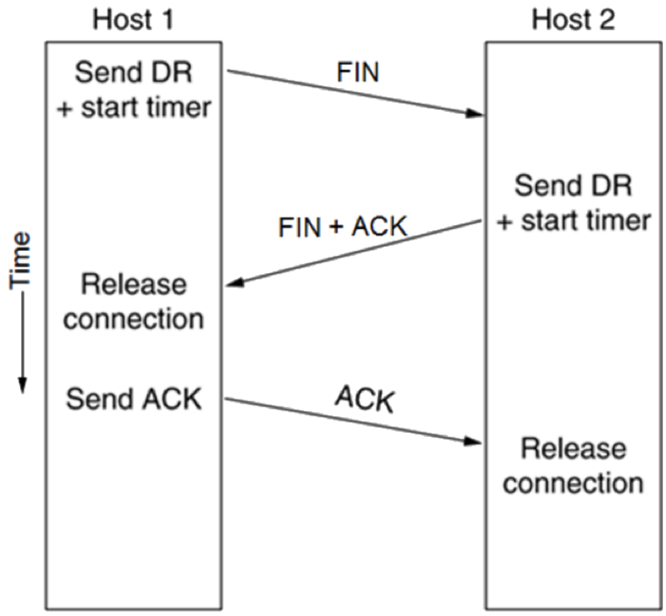


Figure 17.6: Releasing a connection in TCP.

The release of a connection initiated by `Host 1` can be seen in the figure above. This process is outlined in the figure below.

Assume that `Host 1` wishes to cease communication with `Host 2`.

↓

`Host 1` sends a disconnect request (`DR`) to `Host 2`:

- implying that the `FIN` bit is 1; and
- given that the sequence number (`SEQ`) for `Host 1` is `x`.

Given these conditions, it is observed that:

`DR -> FIN=1, SEQ = x`

↓

`Host 1` starts a timer.

↓

`Host 2` acknowledges the disconnect request (`DR`) (`DR`):

- implying that the `FIN` bit is 1;
- given that the sequence number (`SEQ`) for `Host 1` is now `x+1`; and
- given that the sequence number (`SEQ`) for `Host 2` is now `y`.

Given these conditions, it is observed that:

`FIN+ACK -> SEQ=y, ACK=x+1`

↓

`Host 2` also starts a timer.

↓

`Host 1` sends an acknowledgement to `Host 2`:

- given that the sequence number (`SEQ`) for `Host 1` is now `x+1`; and
- given that the sequence number (`SEQ`) for `Host 2` is now `y+1`.

Given these conditions, it is observed that:

`SEQ=x+1, ACK=y+1`

↓

The sequence number (`SEQ`) and acknowledgement number (`ACK`) are now both incremented and therefore the connection can be released.

Figure 17.7: Process for releasing a TCP connection.

The process described in the figure above is guaranteed to release a synchronised connection as `Host 1` and `Host 2` both use individual timers to ensure that both `Host 1` and `Host 2` are able to release the connection in a timely manner.

As a result, should a segment get lost:

- if `Host 1` does not receive an acknowledgement from `Host 2` for its disconnect request (`DR`), then `Host 1` will timeout and resend the disconnect request (`DR`);
- if `Host 1` observes multiple timeouts while waiting for acknowledgement from `Host 2`, the connection will be dropped; and
- if `Host 2` does not receive an acknowledgement from `Host 1`, then `Host 2` will timeout and release the connection.

17.5 Communication in TCP connection

The following example assumes that:

- the three-way handshake has already been completed and therefore the TCP connection has been established; and
- the TCP connection is between a client and server.

```
// Begin communication.

Client: SEQ: 8001, ACK: 15001, Length: 72bytes
Server: SEQ: 15001, ACK: 8001+72=8073, Length: 60bytes
Client: SEQ: 8073, ACK: 15001+60=15061, Length: 156bytes
Server: SEQ: 15061, ACK: 8073+156=8229, Length: 152bytes

// ...

// Release communication.
```

Example TCP communication between a client and server.

As seen above, the client and server relay messages to one another. Each message contains:

- the sequence number `SEQ`;
- the acknowledgement number `ACK`; and
- the length of the message.

The acknowledgement number `ACK` returned is calculated by:

$$\text{ACK} = \text{SEQ} + \text{length}$$

This allows both the client and the server to ensure that no packet loss has occurred.

Packet loss occurs when one or more packets of data travelling across a computer network fail to reach their destination. Packet loss is can be caused by:

- errors in data transmission;
- patchy wireless connections; and

- network congestion.

Checking for this packet loss is important as many mobile devices, such as laptop and smartphones, are likely to use less-reliable wireless connections such as 3G/4G and Wi-Fi. In addition, this is also important because network congestion is common on the Internet.

If packet loss occurs, the affected packets are retransmitted. Retransmission is implemented using timers and ensures that the data transfer is:

- ordered; and
- error-free.

17.6 Limits on data rate in TCP connection

A limit may be placed on the rate of data transmission in a TCP connection should there be a low:

- receiver capacity; or
- network capacity.

Low receiver capacity

When does this occur?

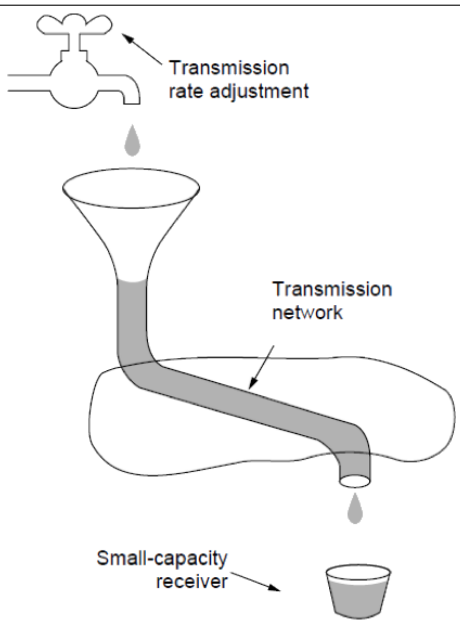


Figure 17.8: Low receiver capacity.

The **receiver capacity** causes a bottleneck, shown circled in the figure above, if a fast network is feeding a low-capacity receiver.

Limiting solution

In the case of a low receiver capacity, the data rate must be limited by **flow control**.

In flow control, buffers are used where the receiver advertises how must data it can receive such that data loss is prevented.

Flow control is managed using the concept of a **sliding window**, in which:

- the receiver informs the sender how much free buffer size it has available, known as the **advertised window** where the window is the amount of buffer space available at any time; and
- when the window is 0, the sender is blocked and may not normally send segments unless the URG flag is on and the running process on the receiver's machine has to be killed.

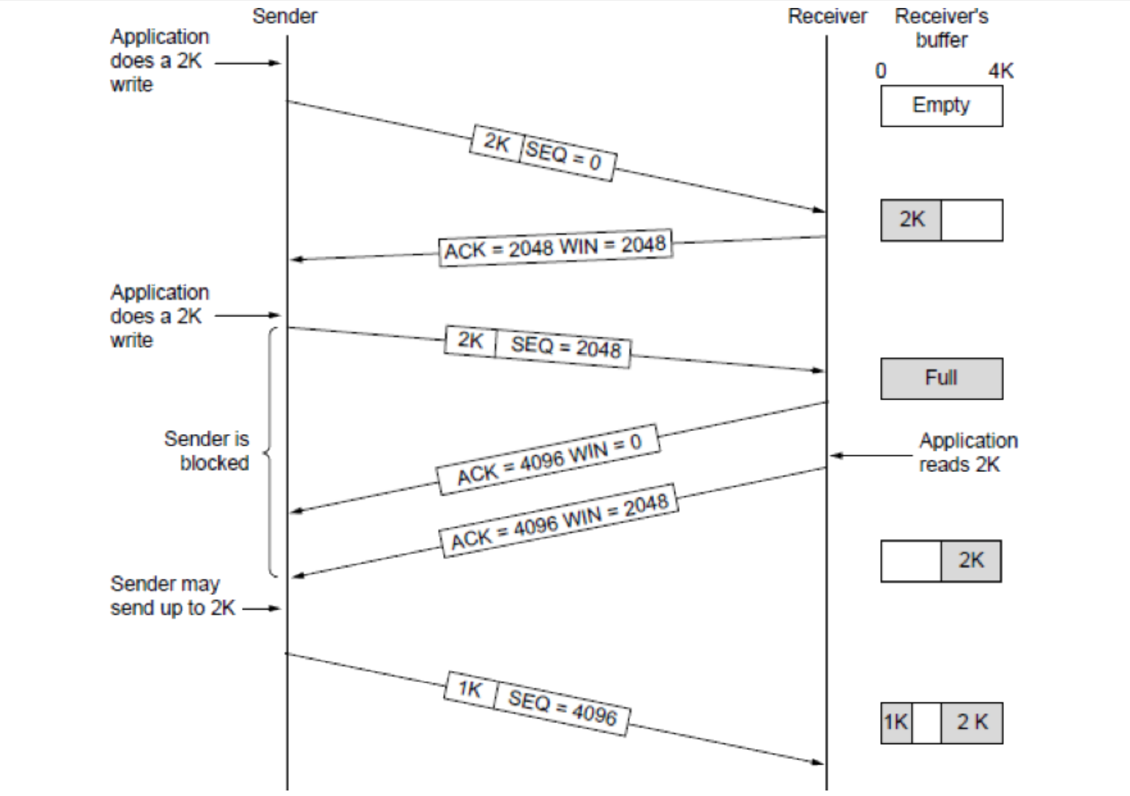


Figure 17.9: Flow control.

The example in the figure above shows the sliding window where the receiver has a buffer size of 4K.

Low network capacity

When does this occur?

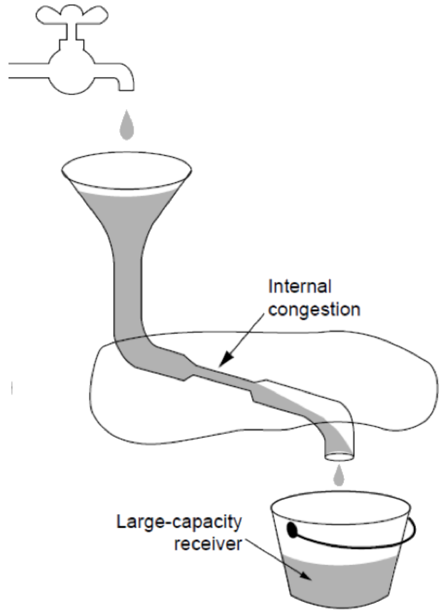


Figure 17.10: Low network capacity.

The **network capacity** causes a bottleneck, shown circled in the figure above, if a slow network is feeding a high-capacity receiver.

Limiting solution

In the case of a low network capacity, the data rate must be limited by **congestion control**.

In congestion control, the sender must control the rate at which it sends the data such that overloading is prevented.

Timeouts caused by packet loss may occur if:

- there is noise on the transmission line; or
- the packet is discarded at a congested route.

However, as modern long-haul lines are very reliable, if a TCP connection times out, it is assumed that the cause was congestion and therefore the data rate must be reduced. This is achieved by using the **Internet Congestion Algorithm**.

The Internet Congestion Algorithm has the following pre-requisites:

- the sender maintains a **receiver window** and **congestion window (CW)**;
- **the maximum segment size** (number of bytes to send) corresponds to the size of the smallest window; and
- a third parameter, known as the **threshold** is used, which is initially **32KB**.

Given these pre-requisites, the Internet Congestion Algorithm follows the process outlined in the figure below.

When a connection is set up, the sender sets the **congestion window (CW)** to the maximum segment size.

↓

The sender transmits data at a size of the maximum segment size.

↓

- If a transmission is successful and the congestion window (CW) is smaller than the threshold, it is doubled on each transmission.
- If a transmission is successful and the congestion window (CW) is larger than the threshold, it is increased by one maximum segment size per transmission.
- If a timeout occurs, the threshold is set to half the current congestion window (CW) and the congestion window (CW) is reset to one maximum segment size.

↓

This process repeats by the sender transmitting more data at a size of the maximum segment size until the connection is released.

Figure 17.11: Process for the Internet Congestion Algorithm.

The process outlined in the figure above shows that the congestion window (CW) can grow and shrink depending on the success the transmissions.

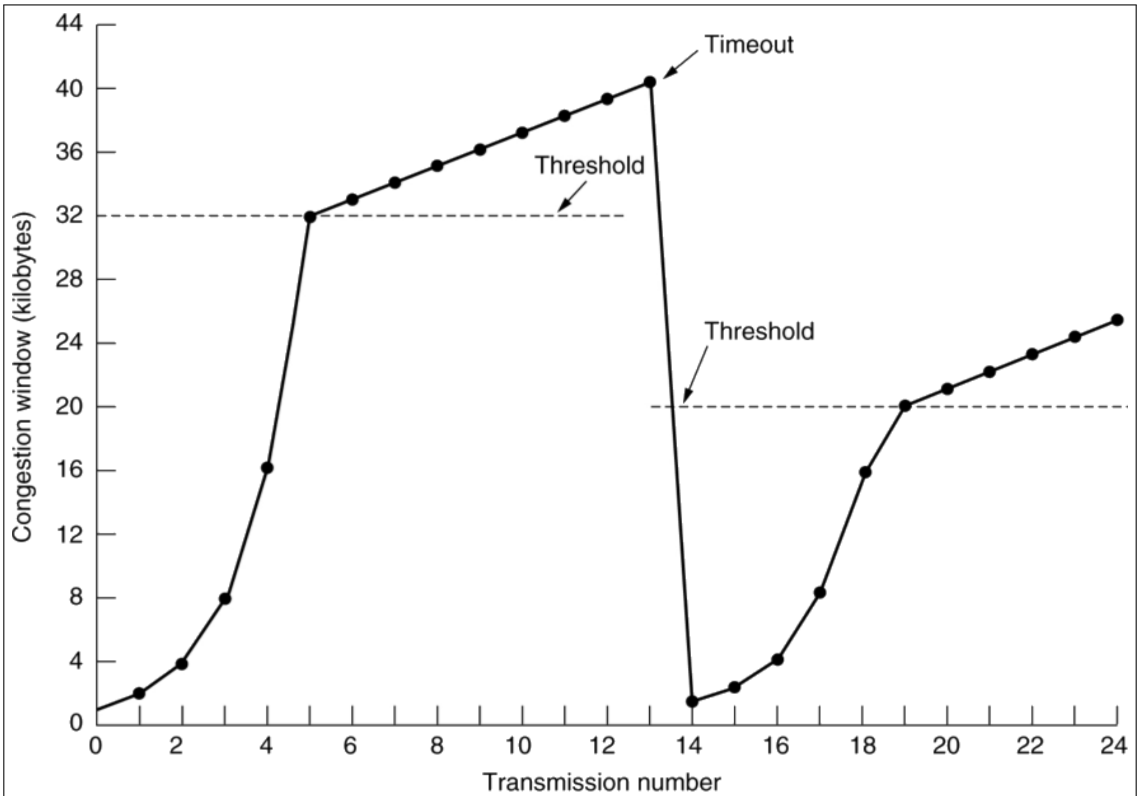


Figure 17.12: Congestion window (CW) growth in the Internet Congestion Algorithm.

As seen in the figure above:

- the growth of the congestion window (CW) starts slow;
- the congestion window (CW) begins to grow exponentially;

- when the threshold is reached, the congestion window (CW) continues to grow linearly; and
- when a timeout occurs, the threshold is decreased and the growth pattern of the congestion window (CW) already observed repeats itself.

A later improvement on the Internet Congestion Algorithm introduced fast recovery in which the congestion window (CW) is not decreased as much.

18. Internet Protocol

18.1 Introduction

The **Internet Protocol** resides in the **Internet** layer of the TCP/IP stack.

The **Internet layer** is the third layer in the TCP/IP stack and uses protocols to transmit packets from source hosts to destination hosts irrespectively of whether the hosts machines are on the same network.

These services are UDP (connectionless best-effort). The TCP services in the transport layer (seen in Chapter 17: Transport Layer Protocols) remain responsible for ensuring that TCP connections remain connection-oriented despite the data being transferred using UDP at the Internet layer.

These services offer functions such as:

- universal addressing;
- fragmentation/re-assembly of packets; and
- internetwork routing (as seen in Chapter 12: Introduction to Internetworking, routers store the topology of the area they cover in a routing table and compute the shortest paths for packets).

18.2 Sending datagrams

A **datagram**, or **packet**, is a piece of a message transmitted over a packet-switching network.

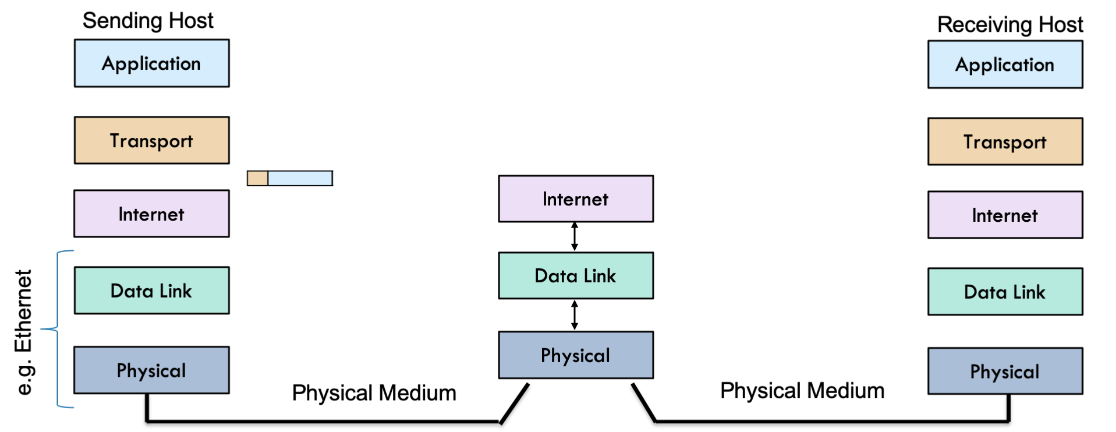


Figure 18.1: Data stream at the transport layer in the sending host.

Firstly, the transport layer:

- takes data streams;
- breaks the data streams in to datagrams (typically `1500 bytes`); and
- sends the datagrams to the Internet layer.

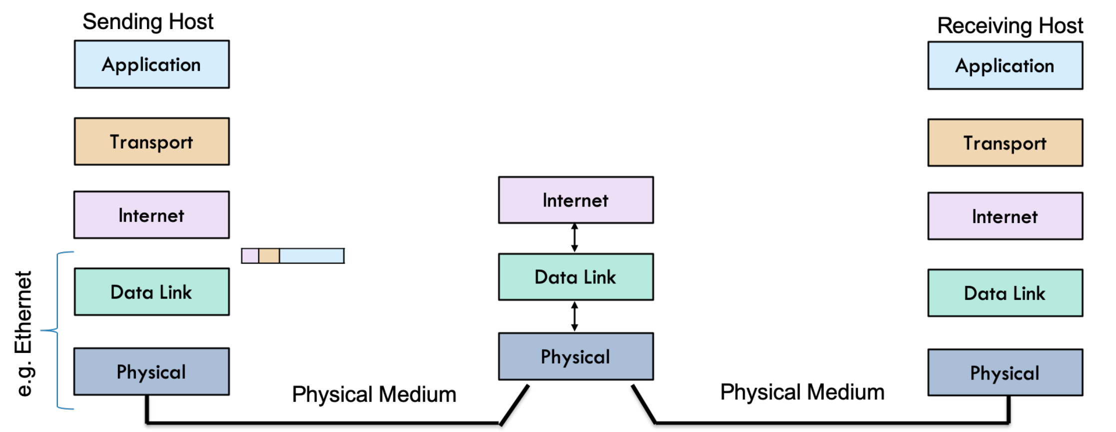


Figure 18.2: Datagram at the Internet layer in the sending host.

The Internet layer transmits the datagrams across the Internet, via the data link and physical layers, orchestrated by routing as seen in Chapter 12: Introduction to Internetworking.

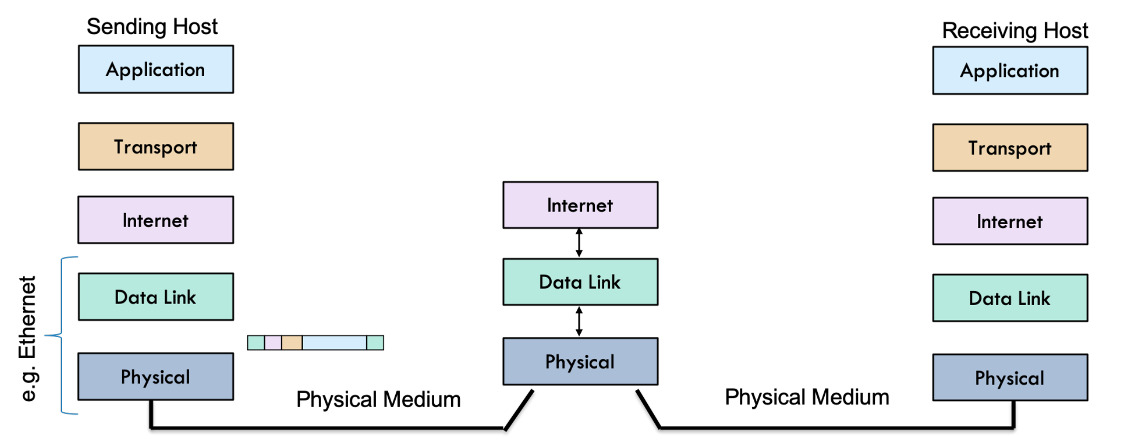


Figure 18.3: Datagram at the data link and physical layers in the sending host.

The physical medium that composes the Internet transmits the signals representing the datagrams.

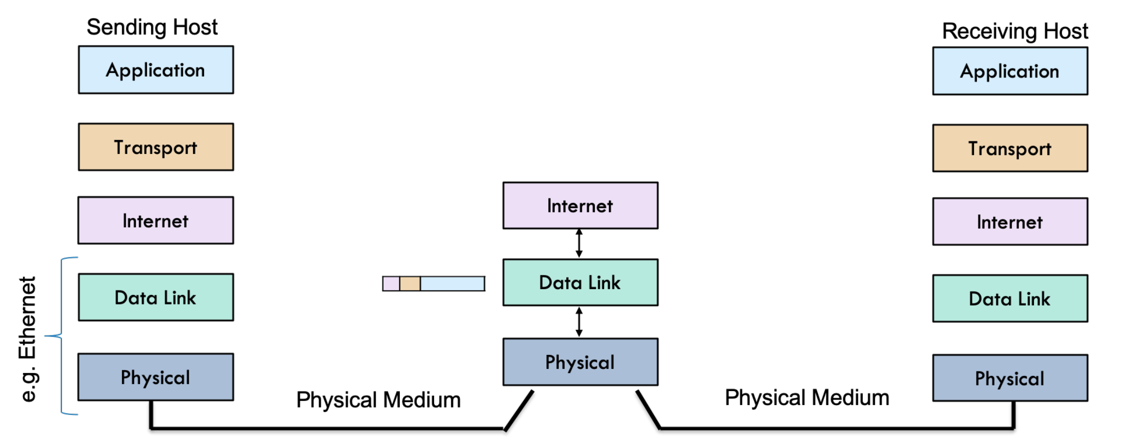


Figure 18.4: Datagram at the data link and physical layers in some router(s).

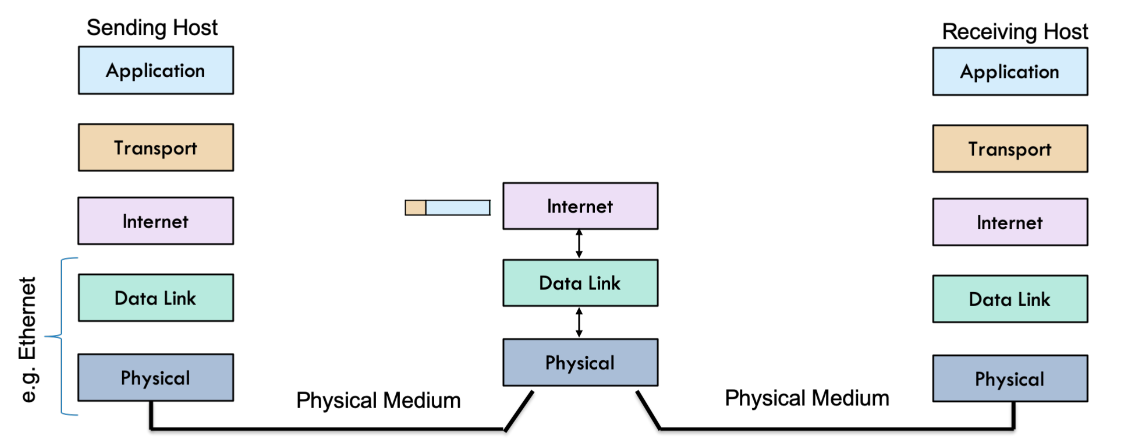


Figure 18.5: Datagram at the Internet layer in some router(s).

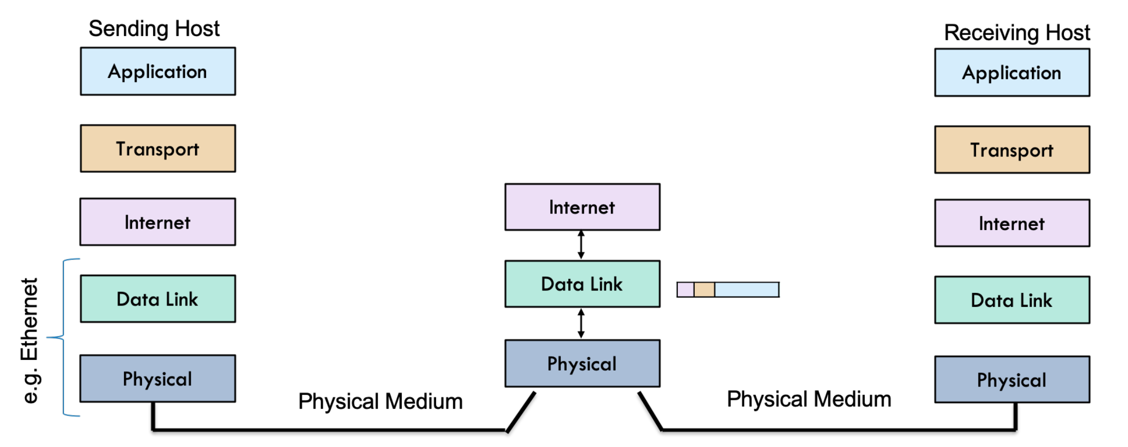


Figure 18.6: Datagram at the data link layer in some router(s).

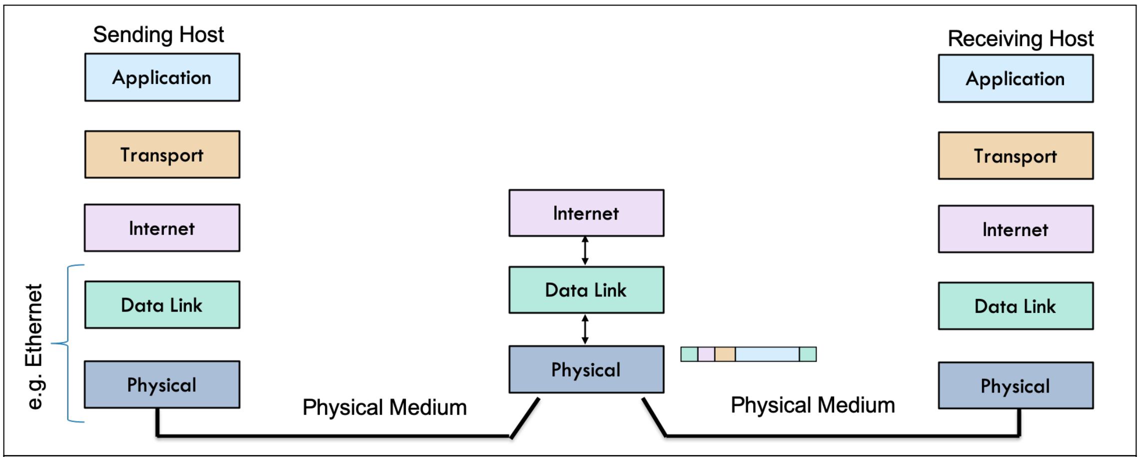


Figure 18.7: Datagram at the physical layer in some router(s).

Each datagram may be fragmented, possibly by a router, if an intermediate network can only handle a smaller packet size therefore causing the datagram to be split.

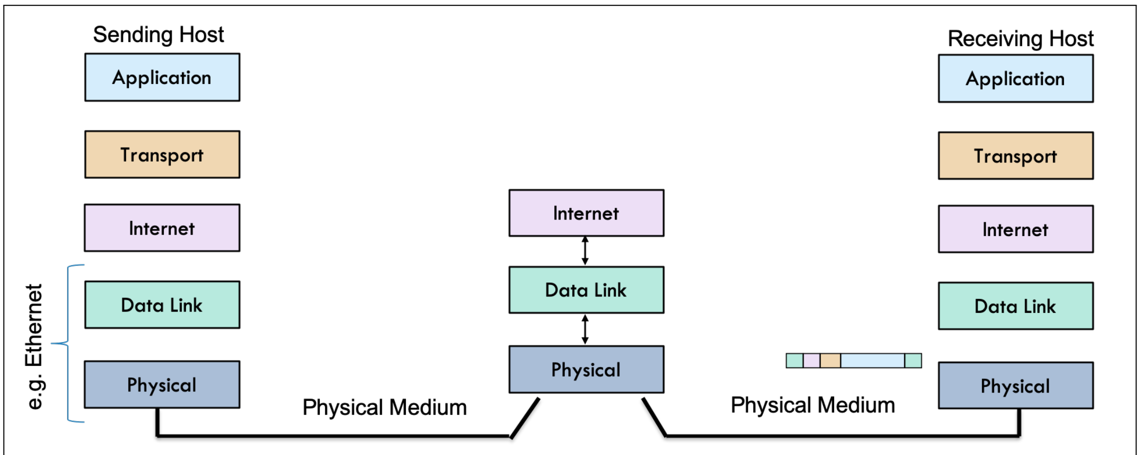


Figure 18.8: Datagram at the physical layer at the receiving host.

Each datagram is reassembled at the destination.

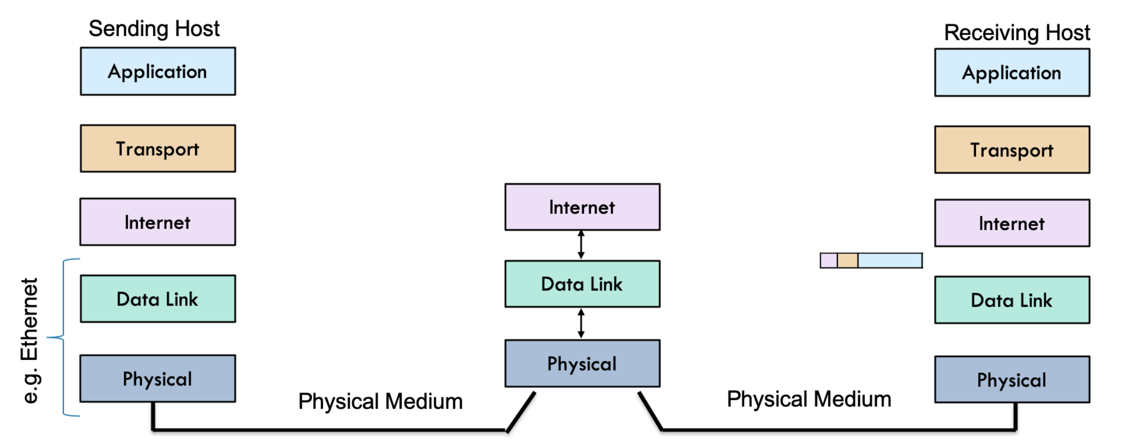


Figure 18.9: Datagram passed from data link layer to Internet layer at the receiving host.

The datagrams are passed to the transport layer, which reassembles them in to a data stream for the application.

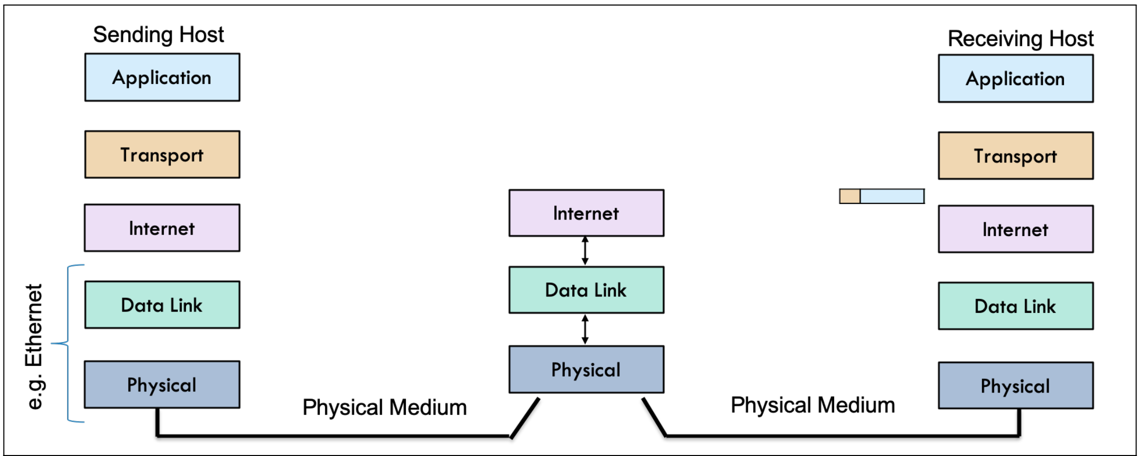


Figure 18.10: Datagram passed from Internet layer to the transport layer at the receiving host.

18.3 IP addresses

Definition

An **IP address** is a universal numerical address that is assigned to and uniquely identifies a device on a network. The IP address indicates where a packet of data is to be sent or from where it was sent. Routers can use IP addresses to direct a packet to its destination.

An IP address does not refer to a host directly but to a network interface. As such, if a host is on two networks, it must have two IP addresses.

IP addresses are managed by the Internet Corporation for Assigned Names and Numbers (ICANN).

IPv4

Definition

Internet Protocol v4 (IPv4) allows each device on a network to have a unique 32-bit binary number address. The binary number is broken down into octets, which are chunks of 8 digits.

Representation

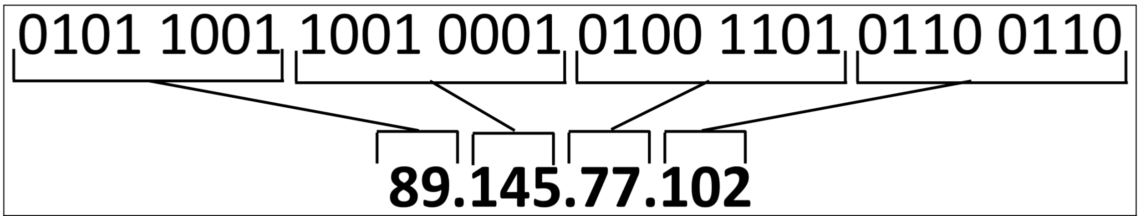


Figure 18.11: An IPv4 address.

As shown in the figure above, IPv4 addresses are often represented using two prevalent notations:

- dotted-decimal notation – for example, `89.145.77.102`; and
- binary (8-bit) notation – for example, `01011001 10010001 01001101 01100110`.

Each block in an IPv4 address:

- can range from 0 to 255 (0 to $2^8 - 1$); and
- must not have a leading zero if represented in binary notation.

Conversions between these notations can be seen in the examples below.

10000001 00001011 00001011 11101111
10000001 = 129
00001011 = 11
00001011 = 11
11101111 = 239
∴ 10000001 00001011 00001011 11101111 = 129.11.11.239

Figure 18.12: Example IPv4 conversion from dotted-decimal notation to binary (8-bit) notation.

11000001 10000011 00011011 11111111
11000001 = 193
10000011 = 131
00011011 = 27
11111111 = 255
∴ 11000001 10000011 00011011 11111111 = 193.131.27.255

Figure 18.13: Example IPv4 conversion from dotted-decimal notation to binary (8-bit) notation.

111.56.45.78
111 = 01101111
56 = 00111000
45 = 00101101
78 = 01001110
∴ 111.56.45.78 = 01101111 00111000 00101101 01001110

Figure 18.14: Example IPv4 conversion from binary (8-bit) notation to dotted-decimal notation.

221.34.7.82
221 = 11011101
34 = 00100010
7 = 00000111
82 = 01010010
∴ 111.56.45.78 = 11011101 00100010 00000111 01010010

Figure 18.15: Example IPv4 conversion from binary (8-bit) notation to dotted-decimal notation.

It is important to note that IPv4 addresses must follow all of the representation standards in order to be valid, as seen below.

IP address	Validity	Explanation
11100010.23.14.45	Invalid.	Mixes binary and dotted-decimal notation.
221.34.5.6.70	Invalid.	There can be no more than four blocks of numbers.
111.56.041.76	Invalid.	There can be no leading zero in any of the blocks.
75.45.301.14	Invalid.	All blocks must be in the range 0 to 255 (0 to $2^8 - 1$).
240.65.75.0	Valid.	Conforms to all of the representation standards.

Classes

IP addresses are hierarchical and the IPv4 address space consists of two parts, as shown in the table below.

Part	Contents	Location
Prefix	Network ID.	Most Significant Bits (MSBs).
Suffix	Host ID.	Least Significant Bits (LSBs).

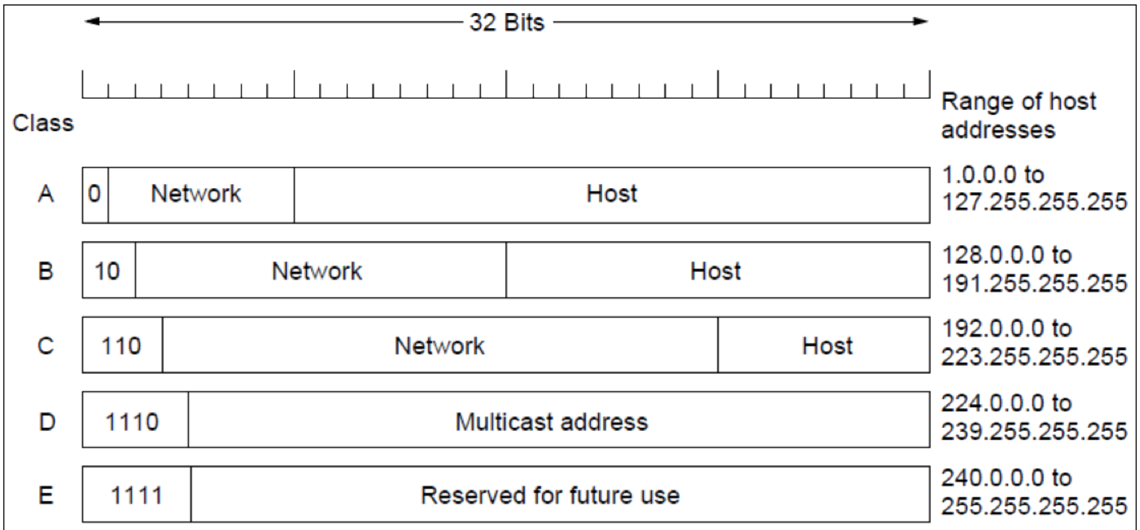


Figure 18.16: IPv4 classes.

As shown in the figure above, IPv4 has five classes. These are described below.

IPv4 addresses with class A:

- have 120 **maximum networks**;
- have 16 million **maximum hosts**;
- are **prefixed** by 0 in their binary (8-bit) notation;
- contain the **network ID** in bits 0–8 in their binary (8-bit) notation; and
- contain the **host ID** in bits 9–32 in their binary (8-bit) notation.

IPv4 addresses with class B:

- have 16384 **maximum networks**;
- have 65536 **maximum hosts**; and
- are **prefixed** by 10 in their binary (8-bit) notation;
- contain the **network ID** in bits 0–16 in their binary (8-bit) notation; and
- contain the **host ID** in bits 17–32 in their binary (8-bit) notation.

IPv4 addresses with class C:

- have 2 million **maximum networks**;
- have 256 **maximum hosts**; and
- are **prefixed** by 110 in their binary (8-bit) notation;
- contain the **network ID** in bits 0–24 in their binary (8-bit) notation; and
- contain the **host ID** in bits 25–32 in their binary (8-bit) notation.

Class D:

- is **not defined** as it is reserved for multicasting (a multicast is group communication where data transmission is addressed to a group of destination computers simultaneously);
- would have IPv4 addresses that are **prefixed** by 1110 in their binary (8-bit) notation.

Class E:

- is **not defined** as it is reserved for future use, should the IPv4 protocol be extended, and experimental purposes;
- would have IPv4 addresses that are **prefixed** by 1111 in their binary (8-bit) notation.

It is possible to:

- determine the **class** of a valid IPv4 address;
- determine the **network ID** of a valid IPv4 address with class A, B or **C**; and
- determine the **host ID** of a valid IPv4 address with class A, B or **C**.

This process is shown in the examples below.

00000001 00001011 00001011 11101111
starts with a 0
$\therefore$ 00000001 00001011 00001011 11101111 has class A
Network ID contained in the first byte
$\therefore$ Network ID = 00000001 = 1
Host ID contained in the last three bytes
$\therefore$ Host ID = 000010110000101111101111 = 11.11.239

Figure 18.17: Example determination of a class from an IPv4 address.

11000001 10000011 00011011 11111111
starts with 110
$\therefore$ 11000001 10000011 00011011 11111111 has class C
Network ID contained in the first three bytes
$\therefore$ Network ID = 110000011000001100011011 = 193.131.27
Host ID contained in the fourth byte
$\therefore$ Host ID = 11111111 = 255

Figure 18.18: Example determination of a class from an IPv4 address.

14.23.120.8
14 = 00001110
23 = 00010111
120 = 01111000
8 = 00001000
$\therefore$ 14.23.120.8 = 00001110 00010111 01111000 00001000
starts with 0
$\therefore$ 14.23.120.8 has class A
Network ID contained the first byte
$\therefore$ Network ID = 00001110 = 14
Host ID contained in the last three bytes
$\therefore$ Host ID = 000101110111100000001000 = 23.120.8

Figure 18.19: Example determination of a class from an IPv4 address.

252.5.15.111
252 = 11111100
5 = 00000101
15 = 00001111
111 = 01101111
$\therefore$ 252.5.15.111 = 11111100 00000101 00001111 01101111
starts with 1111
$\therefore$ 252.5.15.111 has class E

Figure 18.20: Example determination of a class from an IPv4 address.

Special addresses

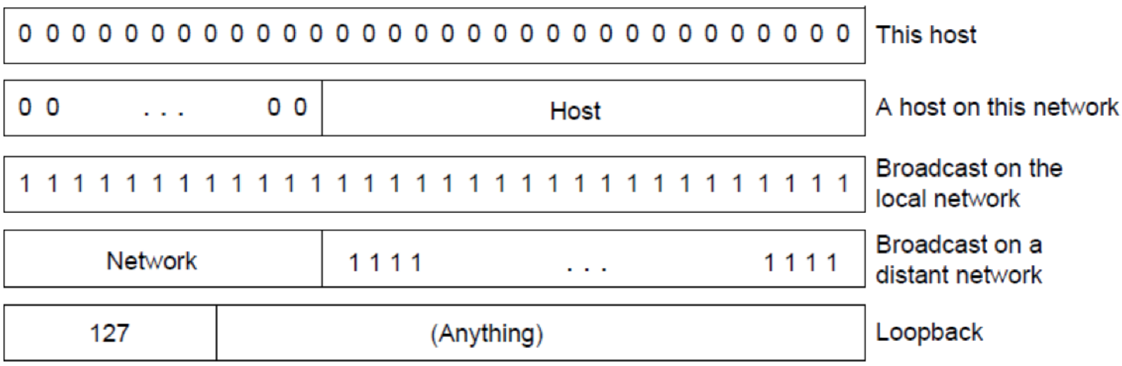


Figure 18.21: IPv4 special addresses.

As shown in the figure above, an IPv4 address:

- containing all zeros (0s) is reserved and identifies the host of origin;
- containing all ones (1s) is reserved and allows broadcasting on the local network (i.e. "send to all neighbours");
- beginning with 127 are reserved for loopback (i.e. send from node to itself), often used for network testing;
- with a network ID containing all zeros (0s) is reserved and identifies a host; and
- with a host ID containing all ones (1s) is reserved and allows broadcasts on a distant network.

Header

An IPv4 header has:

- a 20-byte fixed part; and
- a variable-length `Options field`.

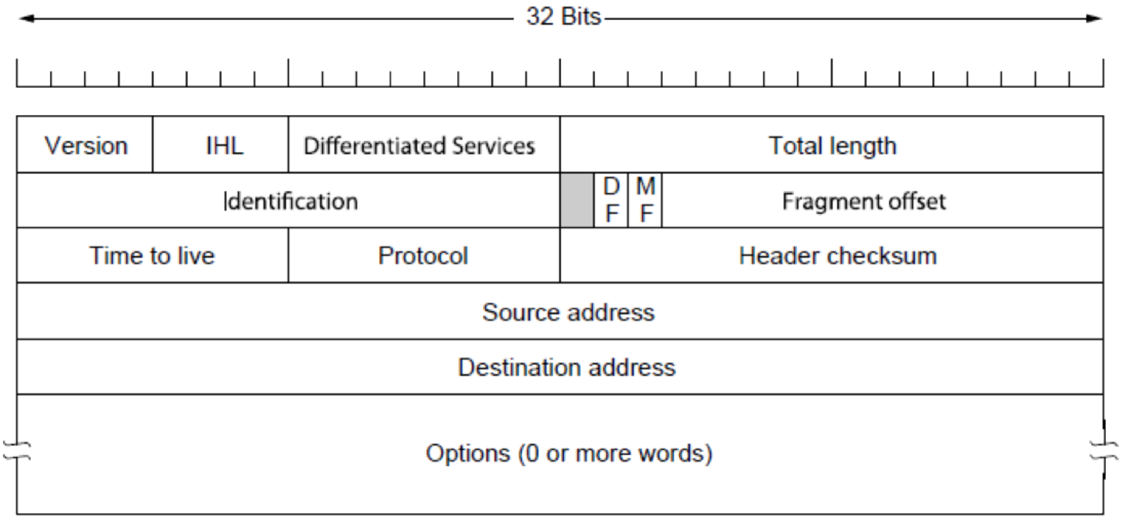


Figure 18.22: An IPv4 header.

As shown above, the IPv4 header contains the following fields:

- **version** – keeps track of the version of IP (IPv4/IPv6) to which the packet belongs;
- **IHL** (Internet Header Length) – indicates the length of the header;
- **differentiated services** – identifies the quality of service;
- **total length** – indicates the length of the packet, including the header and data;
- **identification** – determines to which packet a fragment belongs;
- **reserved** – 1-bit empty for future use, should the IPv4 protocol be extended;
- **DF control flag** – "Don't Fragment", when an IP datagram has its DF flag set to one 1, intermediate devices are not allowed to fragment it so if it needs to travel across a network with a Maximum Transmission Unit (MTU) smaller that datagram length the datagram will have to be dropped (normally, a "ICMP Destination Unreachable message" is generated and sent back to the sender);
- **MF control flag** – "More Fragments", set to indicate the receiver that the current datagram is a fragment of some larger datagram (when set to zero (0), it indicates that the current datagram is either the last fragment in the set or that it is the only fragment);
- **fragment offset** – where in the current packet this fragment belongs;
- **time to live** – counter used to limit packet lifetimes such that, once the counter expires, the packet is discarded or revalidated;
- **protocol** – indicates which transport service to use (TCP/UDP);
- **header checksum** – allows error-checking to be performed;
- **source address** – indicates the IP address of the source network interface;
- **destination address** – indicates the IP address of the destination network interface;
- **options** – allows for additional, non-standard options not covered in the other fields.

Assignment capabilities

IPv4 can assign addresses for up to 4.3 billion computer systems. However, there are a growing number of computer systems and therefore more addresses will be required to allow all devices to be connected to the internet – IPv6 was introduced to solve this problem.

18.4 IP fragmentation

Explanation

An IP datagram with size n bytes must be fragmented according to the **Maximum Transmission Limit (MTL)**. For Ethernet networks, the MTL is 1500 bytes.

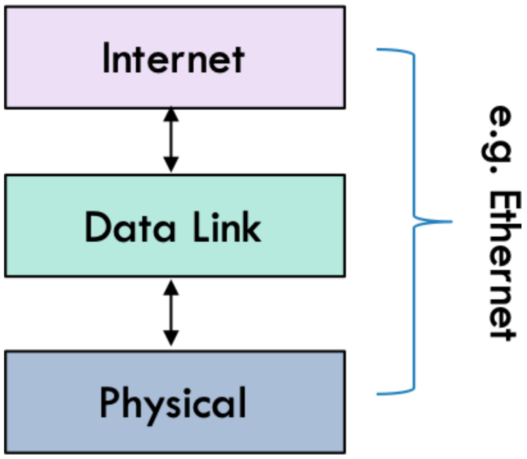


Figure 18.23: IP fragmentation.

As seen before in the Sending datagrams section, in between the data link, physical and Internet layers, each datagram may be fragmented, possibly by a router, if an intermediate network can only handle a smaller packet size therefore causing the datagram to be split.

Example

The following example assumes that:

- the IP datagram being transmitted has size 4400 bytes; and
- the MTL is 1500 bytes.

This means that the input to the router(s) (output from the sending host) would be *one large datagram* with:

- **header length** = 20;
- **total length** = 4400;
- **identification** = 19;
- **DF flag** = 0;
- **MF flag** = 0; and
- **fragment offset** = 0.

It is important to note that the **fragment offset** field:

- must be a multiple of eight; and
- is 16-bit but when a datagram is fragmented, it only has 13 bits to represent the offset (although if the data is ¡64KB, this is sufficient).

As a result, this means that the output from the router(s) (input to the receiving host) would be split, causing *three smaller datagrams* knowns as **fragments**:

Fragment 1 with: • **header length** = 20; • **total length** = 1500; • **identification** = 19; • **DF flag** = 0; • **MF flag** = 1; and • **fragment offset** = 0.	**Fragment 2** with: • **header length** = 20; • **total length** = 1500; • **identification** = 19; • **DF flag** = 0; • **MF flag** = 1; and • **fragment offset** = (1500-20)/8 = 185.	**Fragment 3** with: • **header length** = 20; • **total length** = 20+20+1400 = 1440; • **identification** = 19; • **DF flag** = 0; • **MF flag** = 0; and • **fragment offset** = ((1500-20) + (1500-20)) / 8 = 370.

The **MF flag** is:

- set to zero (1) on the first two fragments as subsequent fragments are to follow; and
- set to (0) on the third fragment as this is the last fragment.

The **total length** is:

- set to the MTL (1500 bytes) for the first two fragments as this is the maximum allowed size, this maximum size is used as to use the least number of fragments possible without exceeding the MTL; and
- calculated by the following formula for the last fragment:

$$\text{total length} = \text{sum of all previous fragment header lengths} + \text{bytes left over}$$

where

$$\begin{aligned}&\text{bytes left over}\\&= \text{total length of the one large datagram} - \text{sum of all previous fragment total lengths}\end{aligned}$$

The **fragment offset** for a fragment n is calculated by:

$$\text{fragment offset} = \frac{\text{total length} - \text{header length}}{8}$$

The division by eight is requested as it has been previously determined the that fragment offset must be a multiple of eight.

IPv6

Definition

Internet Protocol v6 (IPv6) allows each device on a network to have a unique 128-bit binary number address. The binary number is arranged into groups of hexadecimal numbers separated by colons..

Why does IPv6 exist?

IPv6 aims to fix a number of shortcomings presented by IPv4 by:

- using 16-byte addresses;
- paying more attention to different types of services used, such as streaming media;
- adding the possibility to support mobile hosts, by allowing portable computers to obtain the same IP address wherever they are connected; and
- simplifying headers for routers, by using less fields and removing checksums, as to allow faster processing.

Representation

2001:0db8:85a3:0000:0000:8a2e:0370:7334

Figure 18.24: An IPv6 address.

Header

An IPv6 header has:

- a fixed-size header;
- a 16-byte source address; and
- a 16-byte destination address.

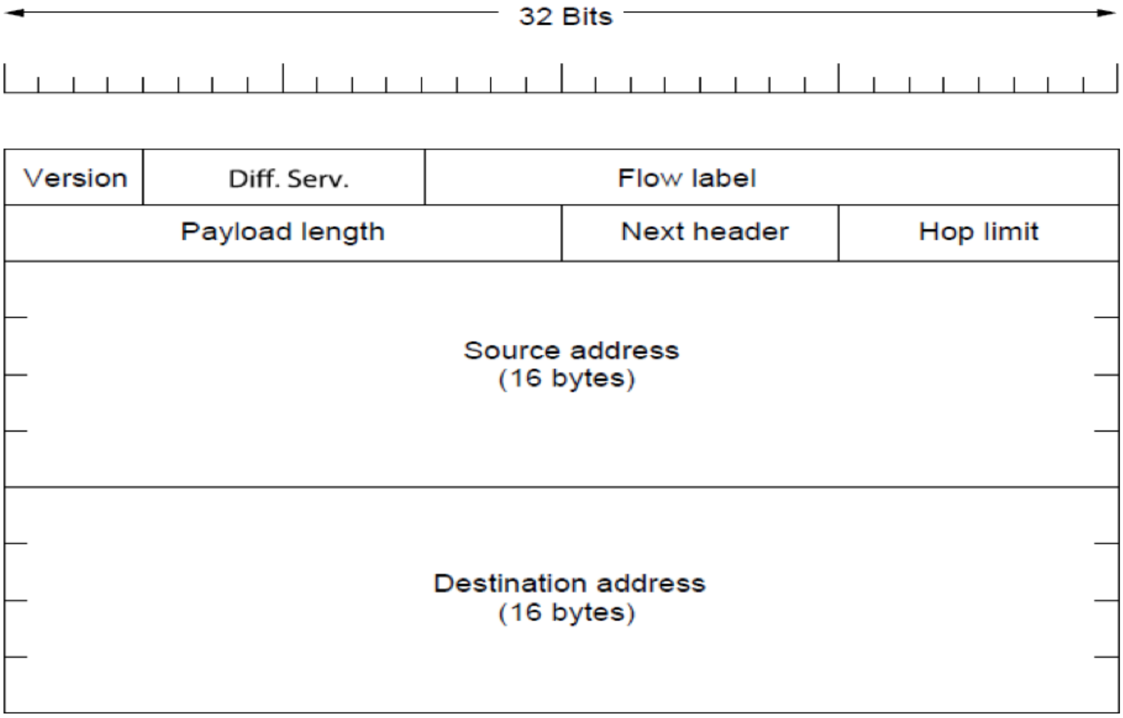


Figure 18.25: An IPv6 header.

As shown above, the IPv6 header contains the following fields:

- **flow label** – maintains the flow and order of packets that belong to the same communication;
- **payload length** – indicates how many bytes follow the header;
- **next header** – indicates whether additional extension headers are used;
- **hop limit** – similar to ”time to live” in IPv4, however the lifetime of a packet is limited by the number of hops it can make between routers because it must be discarded or revalidated;
- **source address** – same as in IPv4, indicates the IP address of the source network interface; and
- **destination address** – same as in IPv4, indicates the IP address of the destination network interface;

Assignment capabilities

IPv6 can assign addresses for up to 2^{128} computer systems.

IPv4 and IPv6 can both coexist and therefore both be used by computer systems on the Internet.

Public and private IP addresses

An IP address can be either:

- public – used to assign a router to the Internet and can be accessed through the Internet; or
- private – used to assign a computer system to a route and cannot be accessed directly through the internet.

18.5 Internet routing

The Internet consists of a large number of independent networks, called **Autonomous Systems (AS)**.

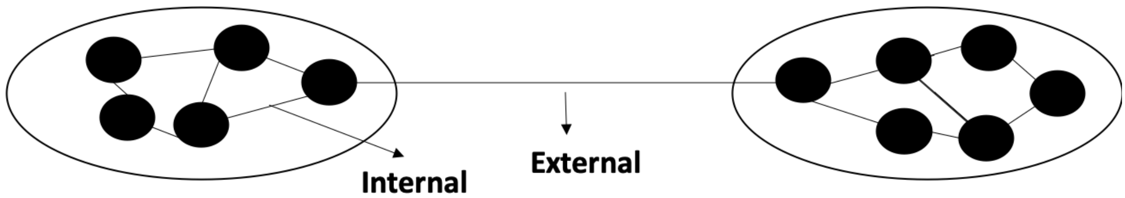


Figure 18.26: Internet routing.

As shown in the figure above:

- routers within an AS use an **Interior Gateway Protocol** to exchange routing information, where a **backbone area** connects all other areas; and
- a router in an AS uses an **Exterior Gateway Protocol** to exchange information with a router from another AS, where **boundary routers** are linked together.

18.6 Tunneling

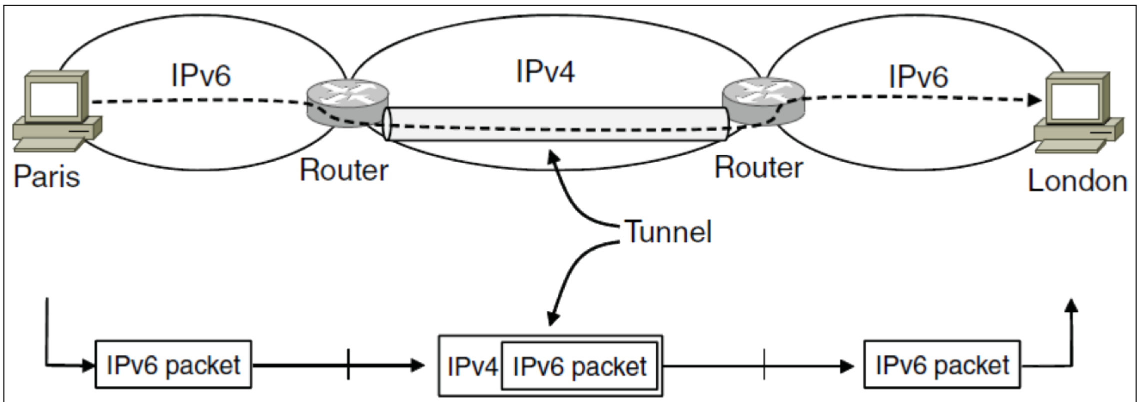


Figure 18.27: Example of tunneling.

The figure above shows a scenario where a company uses IPv6 networks in both Paris and London.

In this scenario the two locations in Paris and London both sending and receiving IPv6 packets. However, the router infrastructure in the data link and physical layers between the two locations can only accommodate for IPv4 packets. In order to circumvent this limitation, the IPv6 packets can be:

- encapsulated inside an IPv4 packet when it has arrived at the boundary router, in the data link and physical layers, from the sending host; and
- de-encapsulated from the IPv4 packet when it has arrived at the boundary router, in the data link and physical layers, to be sent to the receiving host.

The routers themselves are responsible for this encapsulation.

19. Addressing

19.1 Addressing with classes

Addressing with classes is implemented using subnet masks.

Recap

As seen before, IP addresses are hierarchical and the IPv4 address space consists of two parts, as shown in the table below.

Part	**Contents**	**Location**
Prefix	Network ID.	Most Significant Bits (MSBs).
Suffix	Host ID.	Least Significant Bits (LSBs).

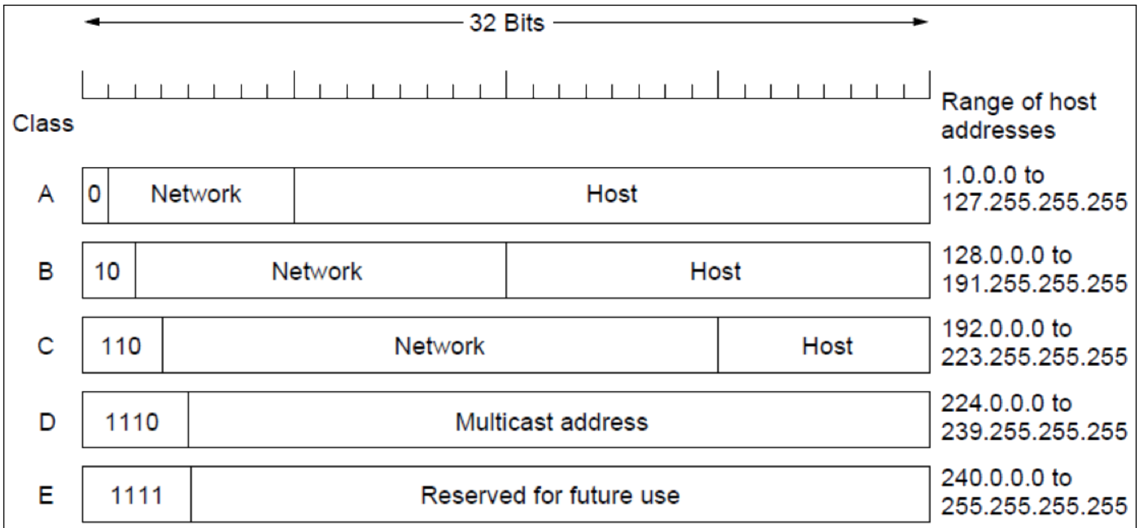


Figure 19.1: IPv4 classes.

As shown in the figure above, IPv4 has five classes. These are described below.

IPv4 addresses with class A:

- have 120 **maximum networks**;
- have 16 million **maximum hosts**;
- are **prefixed** by 0 in their binary (8-bit) notation;
- contain the **network ID** in bits 1–8 in their binary (8-bit) notation; and
- contain the **host ID** in bits 9–32 in their binary (8-bit) notation.

IPv4 addresses with class B:

- have 16384 **maximum networks**;
- have 65536 **maximum hosts**; and
- are **prefixed** by 10 in their binary (8-bit) notation;
- contain the **network ID** in bits 2–16 in their binary (8-bit) notation; and
- contain the **host ID** in bits 17–32 in their binary (8-bit) notation.

IPv4 addresses with class C:

- have 2 million **maximum networks**;
- have 256 **maximum hosts**; and
- are **prefixed** by 110 in their binary (8-bit) notation;
- contain the **network ID** in bits 3–24 in their binary (8-bit) notation; and
- contain the **host ID** in bits 25–32 in their binary (8-bit) notation.

Class D:

- is **not defined** as it is reserved for multicasting (a multicast is group communication where data transmission is addressed to a group of destination computers simultaneously);
- would have IPv4 addresses that are **prefixed** by 1110 in their binary (8-bit) notation.

Class E:

- is **not defined** as it is reserved for future use, should the IPv4 protocol be extended, and experimental purposes;
- would have IPv4 addresses that are **prefixed** by 1111 in their binary (8-bit) notation.

Subnet mask

Definition

A **subnet mask** identifies the length of an IP address.

How it works

A subnet mask is 32 bits in length, the same as an IP address.

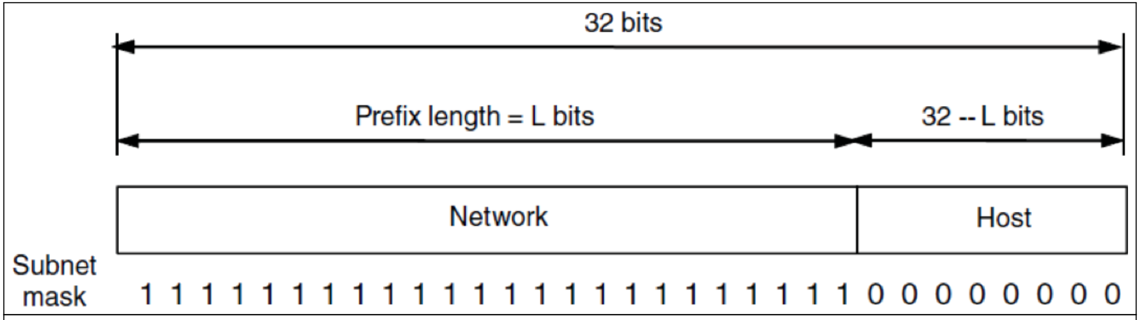


Figure 19.2: Subnet mask.

The subnet mask can be used to extract information for an IP address:

- the IP address is bitwise ANDed with the subnet mask to extract the network ID; and
- the IP address is bitwise ANDed with an inverted version of the subnet mask to extract the host ID (inverting the subnet mask involves flipping all of the bits).

The subnet mask for a class A IP address is 255.0.0.0 as:

- the first byte contains the network ID; and
- the last three bytes contain the host ID.

The subnet mask for a class B IP address is 255.255.0.0 as:

- the first two bytes contain the network ID; and
- the last two bytes contain the host ID.

The subnet mask for a class C IP address is 255.255.255.0 as:

- the first three bytes contain the network ID; and
- the last byte contains the host ID.

This is shown in the examples below.

00000001 00001011 00001011 11101111
starts with a 0
$\therefore$ 00000001 00001011 00001011 11101111 has class A

$$\begin{aligned} \therefore \quad & \text{subnet mask} = 255.0.0.0 \\ & \text{subnet mask} = 11111111\ 00000000\ 00000000\ 00000000 \end{aligned}$$

$$\begin{array}{lcl} \text{Network ID} & = & 00000001\ 00001011\ 00001011\ 11101111 \\ & \& & \underline{11111111\ 00000000\ 00000000\ 00000000} \\ & = & 00000001\ 00000000\ 00000000\ 00000000 \end{array}$$

network ID = 00000001 00000000 00000000 00000000
network ID = 1

$$\begin{array}{lcl} \text{Host ID} & = & 00000001\ 00001011\ 00001011\ 11101111 \\ & \& & \underline{00000000\ 11111111\ 11111111\ 11111111} \\ & = & 00000000\ 00001011\ 00001011\ 11101111 \end{array}$$

host ID = 00000000 00001011 00001011 11101111
host ID = 11.11.239

Figure 19.3: Example determination of network ID and host ID from a class A IP address using subnet masks.

10001001 10000010 00001011 11001111
starts with 10
$\therefore$ 10001001 10000010 00001011 11001111 has class B

$$\begin{aligned} \therefore \quad & \text{subnet mask} = 255.255.0.0 \\ & \text{subnet mask} = 11111111\ 11111111\ 00000000\ 00000000 \end{aligned}$$

$$\begin{array}{lcl} \text{Network ID} & = & 10001001\ 10000010\ 00001011\ 11001111 \\ & \& & \underline{11111111\ 11111111\ 00000000\ 00000000} \\ & = & 10001001\ 10000010\ 00000000\ 00000000 \end{array}$$

network ID = 10001001 10000010 00000000 00000000
network ID = 137.130

$$\begin{array}{lcl} \text{Host ID} & = & 10001001\ 10000010\ 00001011\ 11001111 \\ & \& & \underline{00000000\ 00000000\ 11111111\ 11111111} \\ & = & 00000000\ 00000000\ 00001011\ 11001111 \end{array}$$

host ID = 00000000 00001011 00001011 11101111
host ID = 11.11

Figure 19.4: Example determination of network ID and host ID from a class B IP address using subnet masks.

11000001 10000011 00011011 11111111
starts with 110
∴ 11000001 10000011 00011011 11111111 has class C

$$\begin{aligned} \therefore \quad &\text{subnet mask} = 255.255.255.0 \\ &\text{subnet mask} = 11111111\ 11111111\ 11111111\ 00000000 \end{aligned}$$

$$\begin{aligned} \text{Network ID} \quad = \quad & \ 11000001\ 10000011\ 00011011\ 11111111 \\ & \&\ 11111111\ 11111111\ 11111111\ 00000000 \\ \hline & =\ 11000001\ 10000011\ 00011011\ 00000000 \end{aligned}$$

network ID = 11000001 10000011 00011011 00000000
network ID = 193.131.27

$$\begin{aligned} \text{Host ID} \quad = \quad & \ 11000001\ 10000011\ 00011011\ 11111111 \\ & \&\ 00000000\ 00000000\ 00000000\ 11111111 \\ \hline & =\ 00000000\ 00000000\ 00000000\ 11111111 \end{aligned}$$

host ID = 00000000 00000000 00000000 11111111
host ID = 255

Figure 19.5: Example determination of network ID and host ID from a class C IP address using subnet masks.

Issues

To explore the issues associated with addressing with classes, a generic university can be considered where:

- a class B IP address was used by the Computer Science department;
- later, the Engineering Department requested access to the Internet; and
- later, the Art Department also requested access to the Internet.

When addressing with classes, two new class B IP addresses would have to be purchased for the Engineering Department and Art Department respectively.

This shows that addressing with classes is:

- very expensive; and
- not flexible.

Addressing with classes can be said to be not flexible because, depending on the class used, there may be:

- too many network IDs and not enough host IDs; or
- too many host IDs and not enough network IDs.

Alternatively, classless addressing can be used.

19.2 Classless addressing

Classless addressing is implemented using subnetting.

Subnetting

Definition

Subnetting is a method used in classless addressing.

How it works

In subnetting, the subnet number is added which provides a third level of the hierarchy. This allows the block of addresses to be split **subnets**. These are several parts that are designed for internal use, such as multiple networks, but continue to act as a single network to the "outside world".

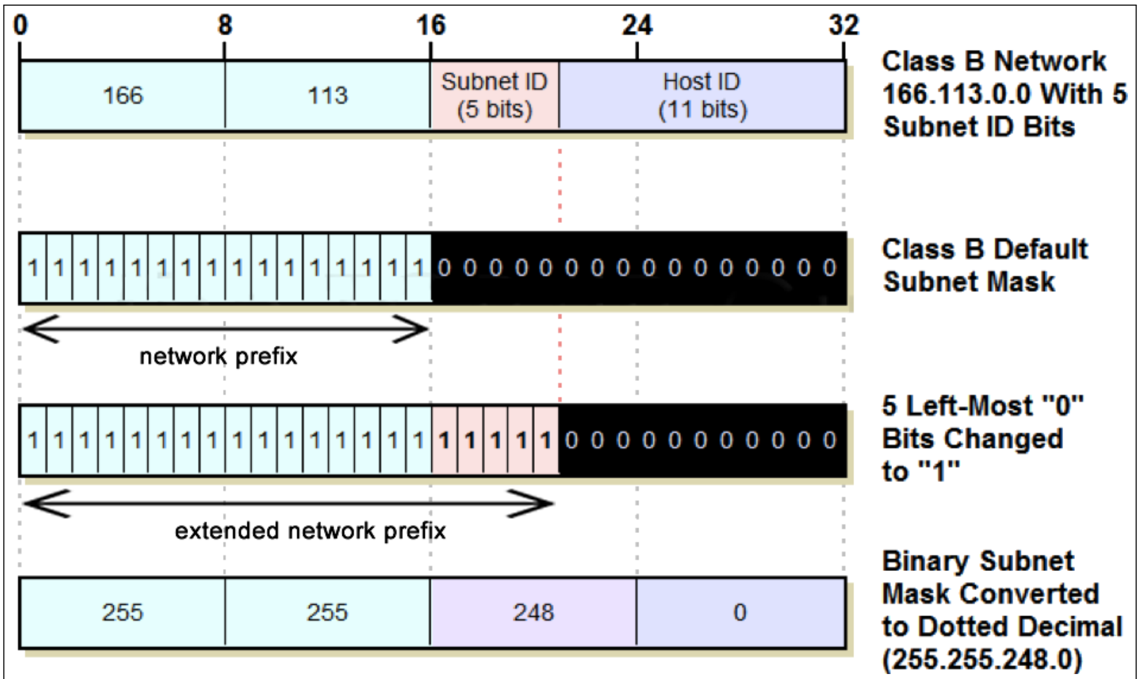


Figure 19.6: Subnetting for class B.

As shown in the figure above, for a class B IP address, the default subnet mask contains a network prefix. However, when using subnetting:

- the network prefix is extended by five bits;
- these five bits are flipped (changed from 0 to 1);
- these five bits now become part of an extended network prefix and therefore can be used to represent the subnet mark within the internal network.

It is important to note that:

- the number of subnets must always be a power of 2; and
- the number of hosts will always be a power of 2.

As a result, the number of subnets available can be calculated by:

$$\text{subnets available} = 2^{X-Y}$$

$$\begin{array}{lccl} \text{where} & X & = & \text{the number of bits in extended network prefix,} \\ & Y & = & \text{the number of bits in network prefix in default subnet mask.} \end{array}$$

As such, the number of hosts available can be calculated by:

$$\text{hosts available} = 2^{\text{subnets available}-(X-Y)}$$

$$\begin{array}{lccl} \text{where} & X & = & \text{the number of bits in extended network prefix,} \\ & Y & = & \text{the number of bits in network prefix in default subnet mask.} \end{array}$$

This is implemented using CIDR, discussed in the next section.

Classless Inter-Domain Routing (CIDR)

Definition

Classless inter-domain routing (CIDR) is a set of Internet protocol (IP) standards that is used to implemented subnetting when using classless addressing.

How it works

The CIDR notation extends the dotted-decimal notation by adding a forward-slash (/) followed by the number of ones (1s) in a subnet mask.

When using the original default subnet masks in addressing with classes, it is easy to identify the network address because it is fixed, as shown in the examples below.

128.208.0.0/16
$\therefore$ 16 bits are used for the network prefix

$$\begin{array}{ll} \therefore & \text{subnet mask} = 11111111\ 11111111\ 00000000\ 00000000 \\ & \text{subnet mask} = 255.255.0.0 \end{array}$$

Figure 19.7: Example determination of subnet mask from a class B IP address using original default subnet masks.

198.208.80.0/24 $\therefore$ 24 bits are used for the network prefix $\therefore$ subnet mask = 11111111 11111111 11111111 0000000 subnet mask = 255.255.255.0
Figure 19.8: Example determination of subnet mask from a class C IP address using original default subnet masks.

However, when using the extended network prefix in classless addressing, groups in the binary (8-bit) representations of the subnet masks may have mixed zero's ('s) and one's (1's), as shown in the examples below.

given that the IP address 131.46.56.0 is class B
$\therefore$ subnet mask = 255.255.0.0
$\therefore$ 16 bits are used for the network prefix
$\therefore$ CIDR = 131.45.56.0/16

given now that the subnet mask is 255.255.128.0, determine the CIDR
255.255.128.0 = 11111111 11111111 10000000 00000000
$\therefore$ 17 bits are used for the extended network prefix
$\therefore$ CIDR = 131.46.56.0/17

subnets available = 2^{X-Y}
where X = the number of bits in extended network prefix,
Y = the number of bits in network prefix in default subnet mask.

$$\begin{aligned}\therefore\quad &\text{subnets available} = 2^{17-16}\\ &\text{subnets available} = 2^{1}\\ &\text{subnets available} = 2\end{aligned}$$

given now that the subnet mask is 255.255.248.0, determine the CIDR
255.255.248.0 = 11111111 11111111 11111000 00000000
$\therefore$ 21 bits are used for the extended network prefix
$\therefore$ CIDR = 131.46.56.0/21

subnets available = 2^{X-Y}
where X = the number of bits in extended network prefix,
Y = the number of bits in network prefix in default subnet mask.

$$\begin{aligned}\therefore\quad &\text{subnets available} = 2^{21-16}\\ &\text{subnets available} = 2^{5}\\ &\text{subnets available} = 32\end{aligned}$$

given now that the subnet mask is 255.255.192.0, determine the CIDR
255.255.192.0 = 11111111 11111111 11000000 00000000
$\therefore$ 18 bits are used for the extended network prefix
$\therefore$ CIDR = 131.46.56.0/18

subnets available = 2^{X-Y}
where X = the number of bits in extended network prefix,
Y = the number of bits in network prefix in default subnet mask.

$$\begin{aligned}\therefore\quad &\text{subnets available} = 2^{18-16}\\ &\text{subnets available} = 2^{2}\\ &\text{subnets available} = 4\end{aligned}$$

Figure 19.9: Example determination of subnet mask from a class B IP address using the extended network prefix.

given that the IP address 192.168.128.0 is class C
$\therefore$ subnet mask = 255.255.255.0
$\therefore$ 24 bits are used for the network prefix
$\therefore$ CIDR = 192.168.128.0/24

---

given now that the subnet mask is 255.255.255.192, determine the CIDR
255.255.255.192 = 11111111 11111111 11111111 11000000
$\therefore$ 26 bits are used for the extended network prefix
$\therefore$ CIDR = 192.168.128.0/26

subnets available = 2^{X-Y}
where X = the number of bits in extended network prefix,
Y = the number of bits in network prefix in default subnet mask.

$\therefore$ subnets available = 2^{26-24}
subnets available = 2^2
subnets available = 4

---

given now that the subnet mask is 255.255.255.224, determine the CIDR
255.255.255.224 = 11111111 11111111 11111111 11100000
$\therefore$ 27 bits are used for the extended network prefix
$\therefore$ CIDR = 192.168.128.0/27

subnets available = 2^{X-Y}
where X = the number of bits in extended network prefix,
Y = the number of bits in network prefix in default subnet mask.

$\therefore$ subnets available = 2^{27-24}
subnets available = 2^3
subnets available = 8

---

given now that the subnet mask is 255.255.255.254, determine the CIDR
255.255.255.254 = 11111111 11111111 11111111 11111110
$\therefore$ 31 bits are used for the extended network prefix
$\therefore$ CIDR = 192.168.128.0/31

subnets available = 2^{X-Y}
where X = the number of bits in extended network prefix,
Y = the number of bits in network prefix in default subnet mask.

$\therefore$ subnets available = 2^{31-24}
subnets available = 2^7
subnets available = 128

Figure 19.10: Example determination of subnet mask from a class C IP address using the extended network prefix.

The examples in the figures above show how classless addressing is more flexible than addressing with classes as:

- when using the original default subnet masks in addressing with classes, only one subnet mask is available; and
- when using subnetting in classless addressing, a high number of subnets (power of 2) is available based on the number of bits in extended network prefix and the number of bits in network prefix in default subnet mask.

Designing subnets

Fixed Length Subnet Masks (FLSM)

The issues with the addressing with classes approach were explored by considering a generic university where:

- a class B IP address was used by the Computer Science department;
- later, the Engineering Department requested access to the Internet; and
- later, the Art Department also requested access to the Internet.

When addressing with classes, two new class B IP addresses would have to be purchased for the Engineering Department and Art Department respectively.

This showed that addressing with classes is:

- very expensive; and
- not flexible.

Addressing with classes was said to be not flexible because, depending on the class used, there may be:

- too many network IDs and not enough host IDs; or
- too many host IDs and not enough network IDs.

However, in the classless addressing solution, the extended network prefixes and subsequent subnet masks are used to identify the subnet and forward packets accordingly.

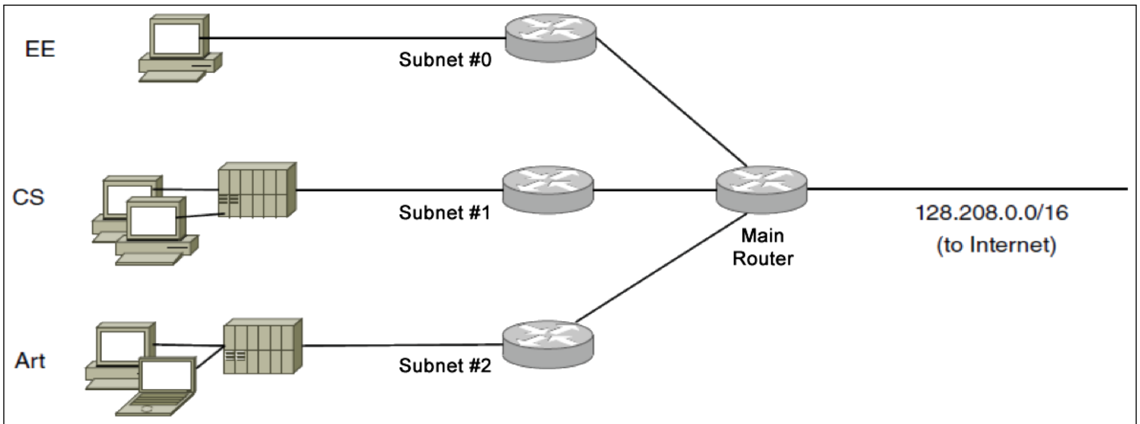


Figure 19.11: Example of subnetting.

The example above shows the three university departments: Computer Science; Engineering; and Art. Three subnets are designed from the class B IP address.

In order to determine the length of the extended network prefix, it is necessary to observe the determined rule that the number of subnets must always be a power of 2:

- $2^0 = 1 - 1$ is not a power of 2;
- $2^1 = 2 - 2$ is not a power of 2; and
- $2^2 = 4 - 4$ is a power of 2.

It has now been determined that 2^2 gives a power of 2 and therefore two additional binary bits are required in the extended network prefix: CIDR = ¡IP address¿/18

$\therefore$ subnet mask = 11111111 11111111 11000000 00000000
subnet mask = 255.255.192.0

The subnets can then be defined by exhausting all possible combinations of the additional bits in the extended network prefix in numerically ascending order until the required amount of subnets is reached (in this case, three subnets):

Subnet #0:	10000000 11010000 **00000000** 00000000	= 128.208.0.0/18
Subnet #1:	10000000 11010000 **01000000** 00000000	= 128.208.64.0/18
Subnet #2:	10000000 11010000 **00000000** 00000000	= 128.208.128.0/18

The range of hosts that can be assigned for each subnet can then be determined:

- **Subnet #0**: 128.208.0.1 – 128.208.63.254;
- **Subnet #1**: 128.208.64.1 – 128.208.127.254; and
- **Subnet #2**: 128.208.128.1 – 128.208.191.254.

Similar to IPv4's special addresses, as seen in Special addresses (Chapter 18), a host ID in any given subnet:

- containing all zeros (0s) is reserved and identifies the network address of the subnet, such as 128.208.0.0 for Subnet #0;
- containing all ones (1s) is reserved and allows broadcasting on the subnet (i.e. "send to all neighbours"), such as 128.208.63.255 for Subnet # 0;

When a packet arrives in to the main router, the subnet mask is used to check where the packet should be routed, as seen in the example below.

A packet arrives with the destination address 128.208.2.130.

↓

It is determined that the subnet mask is 255.255.196.0 for all subnets.

↓

The destination address is bitwise ANDed with the subnet mask to determine the subnet:

128.208.2.130 = 10000000 11010000 00000010 10000010
255.255.196.0 = 11111111 11111111 11000100 00000000

$$\begin{array}{lcr} \text{subnet} & = & 10000000\ 11010000\ 00000010\ 10000010 \\ & \& & 11111111\ 11111111\ 11000100\ 00000000 \\ \hline & = & 10000000\ 11010000\ 00000000\ 00000000 \end{array}$$

subnet = 128.208

Figure 19.12: Example process for routing a packet to a subnet using CLSM.

In this example:

- the total number of allocated addresses is 65536;
- the number of allocated addresses to each department is as follows ...
  - Computer Science department (CS) – 16384 (with extended prefix /18);
  - Engineering department (EE) – 16384 (with extended prefix /18); and
  - Art department (Art) – 16384 (with extended prefix /18).
- the total number of unallocated (available) addresses is 16384.

The number of allocated addresses determines the number of devices that can be connected to that subnet. However it is possible that a specific department requires more addresses than another, as seen in the next section.

Variable Length Subnet Masks (VLSM)

Suppose that:

- the Computer Science department (CS) requires double the current number of allocated addresses (32768);
- the Art department (Art) requires half the current number of allocated addresses (8192); and
- the Engineering department (EE) is satisfied with the current number of allocated addresses (16384).

A similar process is observed as in CLSM however, the destination address is bitwise ANDed with all subnet masks individually.

A packet arrives with the destination address 128.208.2.130.

↓

It is determined that:

- the subnet mask is 255.255.128.0 for the Computer Science department (CS);
- the subnet mask is 255.255.196.0 for the Engineering department (EE); and
- the subnet mask is 255.255.244.0 for the Art department (Art).

↓

The destination address is bitwise ANDed with the each subnet mask individually to determine the subnet.

↓

For the Computer Science department (CS):
128.208.2.130 = 10000000 11010000 00000010 10000010
255.255.128.0 = 11111111 11111111 10000000 00000000

$$\begin{aligned} \text{subnet} = & \quad 10000000\ 11010000\ 00000010\ 10000010 \\ \& & \quad 11111111\ 11111111\ 10000000\ 00000000 \\ \hline = & \quad 10000000\ 11010000\ 00000000\ 00000000 \end{aligned}$$

subnet = 128.208

↓

For the Engineering department (EE):
128.208.2.130 = 10000000 11010000 00000010 10000010
255.255.196.0 = 11111111 11111111 11000100 00000000

$$\begin{aligned} \text{subnet} = & \quad 10000000\ 11010000\ 00000010\ 10000010 \\ \& & \quad 11111111\ 11111111\ 11000100\ 00000000 \\ \hline = & \quad 10000000\ 11010000\ 00000000\ 00000000 \end{aligned}$$

subnet = 128.208

↓

For the Art department (Art):
128.208.2.130 = 10000000 11010000 00000010 10000010
255.255.244.0 = 11111111 11111111 11110100 00000000

$$\begin{aligned} \text{subnet} = & \quad 10000000\ 11010000\ 00000010\ 10000010 \\ \& & \quad 11111111\ 11111111\ 11110100\ 00000000 \\ \hline = & \quad 10000000\ 11010000\ 00000000\ 00000000 \end{aligned}$$

subnet = 128.208

Figure 19.13: Example process for routing a packet to a subnet using VLSM.

Advantages of classless addressing

Classless addressing benefits from the advantages of subnetting, including:

- **improves the efficiency** of IP addresses by not consuming an entire address space for each physical network;
- **allows better organisation** of resources internally;
- **reduces the complexity of external routers** because they only have to store subnet masks, rather than all of the hosts;
- **better performance** because a destination address will be checked against the hosts of the subnet, rather than all hosts;
- **less expensive** as new IP addresses do not need to be purchases should more devices require addressing; and
- **more flexible** as addresses can be allocated as required using Variable Length Subnet Masks (VLSM).

Subnetting exercises

An ISP owns a network with the IP address 201.70.64.0. The ISP wants to create six subnets and advertise them to customers.

What class is the IP address the ISP owns?

201.70.64.0 = 11001001 01000110 01000000 00000000
starts with 110
$\therefore$ IP address has class C

What is the default subnet address of the IP address?

IP address has class C
$\therefore$ subnet mask = 255.255.255.0

What is the CIDR notation of the IP address?

IP address has default subnet address 255.255.255.0
255.255.255.0 = 11111111 11111111 11111111 00000000
$\therefore$ 24 bits are used for the extended network prefix
$\therefore$ CIDR = 201.70.64.0/24

How many additional binary digits are needed to design six subnets? How many hosts does this allow?

subnets available = 2^{X-Y}
where X = the number of bits in extended network prefix,
Y = the number of bits in network prefix in default subnet mask

when using one additional binary digit,
number of bits in extended network prefix = 24 + 1
number of bits in extended network prefix = 25
number of bits in network prefix in default subnet mask = 24
subnets available = 2^{25-24}
subnets available = 2^1
subnets available = 1
∴ one additional binary digit is insufficient

when using two additional binary digits,
number of bits in extended network prefix = 24 + 2
number of bits in extended network prefix = 26
number of bits in network prefix in default subnet mask = 24
subnets available = 2^{26-24}
subnets available = 2^2
subnets available = 4
∴ two additional binary digits are insufficient

when using three additional binary digits,
number of bits in extended network prefix = 24 + 3
number of bits in extended network prefix = 27
number of bits in network prefix in default subnet mask = 24
subnets available = 2^{27-24}
subnets available = 2^3
subnets available = 8
∴ three additional binary digits are sufficient

three additional binary digits gives 8 subnets hosts available = $2^{\text{subnets available}-(X-Y)}$

where X = the number of bits in extended network prefix,
Y = the number of bits in network prefix in default subnet mask

hosts available = $2^{8-(27-24)}$
hosts available = 2^{8-3}
hosts available = 2^5
hosts available = 32

What is the extended network prefix and the subnet mask?

the number of bits in extended network prefix is 27
$\therefore$ extended network prefix = /27

the number of bits in extended network prefix is 27
$\therefore$ subnet mask = 11111111 11111111 11111111 11100000
subnet mask = 255.255.255.224

Express the subnets to advertise in binary notation and dotted-decimal notation.

the subnets can be defined by exhausting all possible combinations of the additional bits in the extended network prefix in numerically ascending order until the required amount of subnets is reached (in this case, six subnets):

- Subnet #0 – 11001001 01000110 01000000 **00000000** = 201.70.64.00/27;
- Subnet #1 – 11001001 01000110 01000000 **00100000** = 201.70.64.32/27;
- Subnet #2 – 11001001 01000110 01000000 **01000000** = 201.70.64.64/27;
- Subnet #3 – 11001001 01000110 01000000 **01100000** = 201.70.64.96/27;
- Subnet #4 – 11001001 01000110 01000000 **10000000** = 201.70.64.128/27; and
- Subnet #5 – 11001001 01000110 01000000 **10100000** = 201.70.64.164/27.

What are the ranges of host addresses for each subnet?

Subnet #0 – 201.70.64.00/27
$\therefore$ range of host addresses for subnet #0: 201.70.64.00 - 201.70.64.31

Subnet #1 – 201.70.64.32/27
$\therefore$ range of host addresses for subnet #1: 201.70.64.32 - 201.70.64.63

Subnet #2 – 201.70.64.64/27
$\therefore$ range of host addresses for subnet #2: 201.70.64.64 - 201.70.64.95

Subnet #3 – 201.70.64.96/27
$\therefore$ range of host addresses for subnet #3: 201.70.64.96 - 201.70.64.127

Subnet #4 – 201.70.64.128/27
$\therefore$ range of host addresses for subnet #4: 201.70.64.128 - 201.70.64.163

Subnet #5 – 201.70.64.164/27
$\therefore$ range of host addresses for subnet #5: 201.70.64.164 - 201.70.64.255

subnet #0 range starts at subnet 201.70.64.00 and subnet #5 range ends at 201.70.64.255
$\therefore$ total number of allocated addresses (granted hosts) = 256

How many hosts are still available for future use by the ISP?

total number of allocated addresses (granted hosts) = 256
the number of allocated addresses (assigned hosts) = 6×32
the number of allocated addresses (assigned hosts) = 192
$\therefore$ the total number of unallocated (available) addresses = 256 - 192
the total number of unallocated (available) addresses = 64 (spanning across the two unused subnets)

20. The Ethernet Protocol

20.1 Introduction

The **Ethernet protocol** resides in the **data link layer** and **physical layer** of the TCP/IP stack.

The **data link layer** is the second layer in the TCP/IP stack and uses protocols to achieve reliable and efficient communication between two adjacent machines. It offers functions such as:

- framing;
- error detection;
- channel access control; and
- addressing.

The **physical layer** is the first layer in the TCP/IP stack and uses protocols to transmit raw bits reliably between machines by:

- specifying the mechanical and electrical interfaces to the network; and
- encoding the binary digits in the frames from the data link layer in to signals.

20.2 The physical layer

Physical transmission media

Guided and unguided transmission media

The signals in the physical layer are transmitted/received across a transmission media.

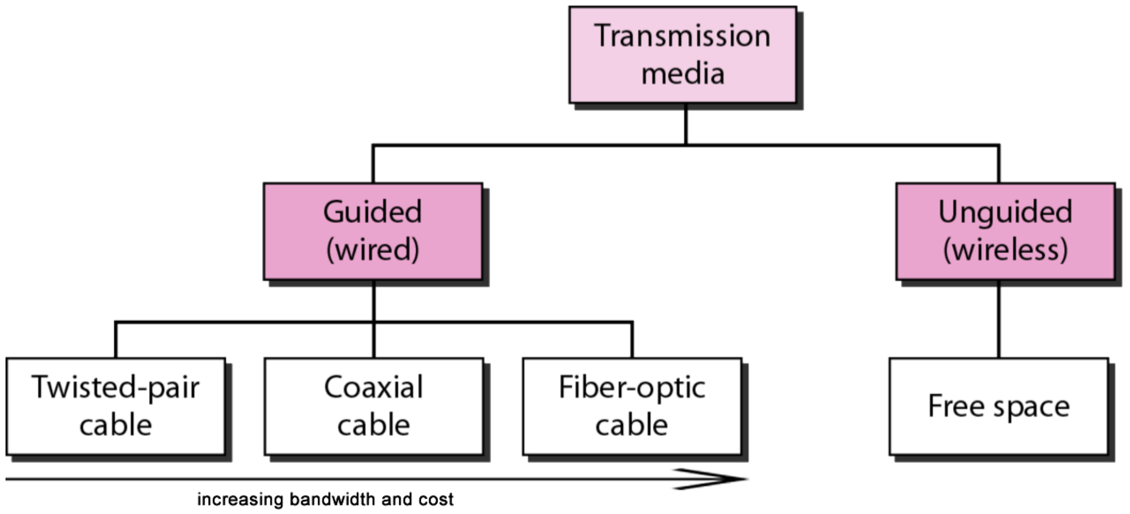


Figure 20.1: Types of physical transmission media.

The figure above shows that the transmission may be

- guided, such as twisted-pair cables, coaxial cables and fibre-optic cables; or
- unguided, such as radio wave through the air.

The figure also shows the difference in bandwidth and cost for guided transmission media.

Signals that represent particular bit patterns are used to indicate the start and the end of a frame.

Twisted-pair cable

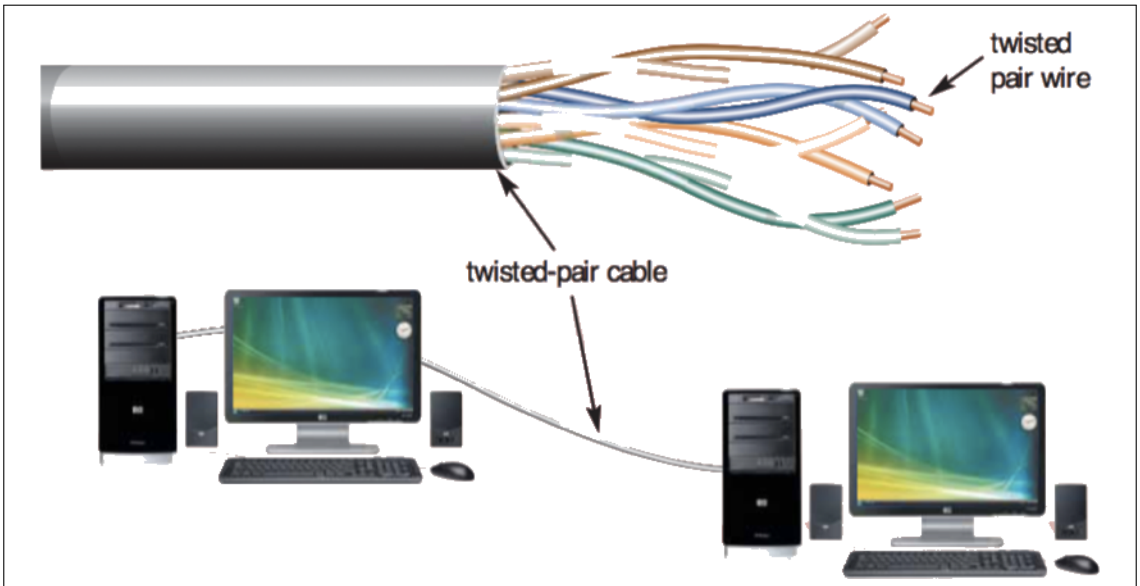


Figure 20.2: Twisted-pair cables.

Twisted-pair cables run for several kilometres but for longer distances, the signal becomes weak.

They support bandwidth for several speeds (Mbps):

- for lower speeds, two out of the four pairs are used where each pair corresponds to each direction; and
- for higher speeds, all four pairs are used in both directions (duplex) simultaneously.

Twisted-pair cables are:

- low cost; and
- widely used as all wired telephones use these cables.

Coaxial cable

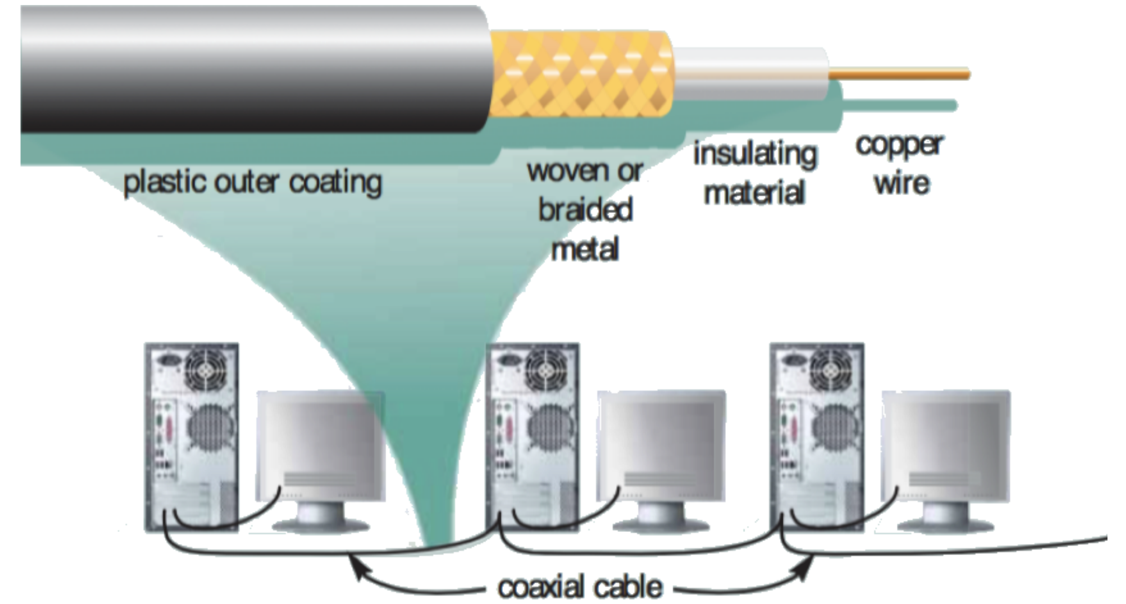


Figure 20.3: Coaxial cables.

Coaxial cables use copper and can run for longer distances at higher speeds than twisted-pair cables.

Coaxial cables are:

- more costly than twisted-pair cables;
- more protected than twisted-pair cables; and
- widely used for cable television.

Fibre-optic cable

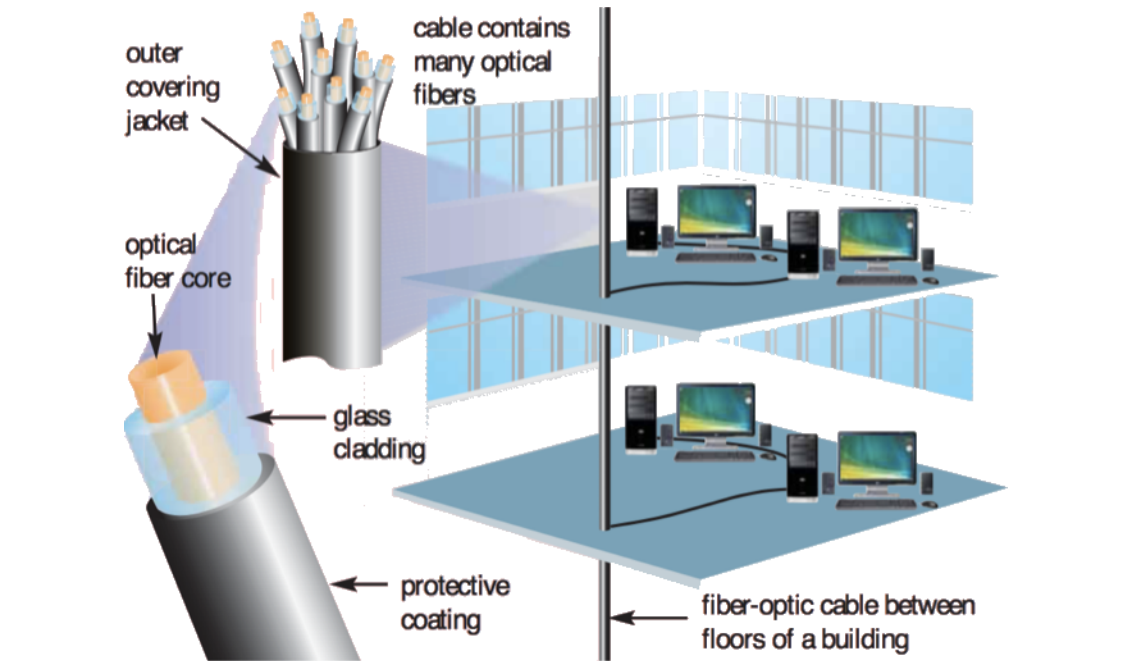


Figure 20.4: Fibre-optic cables.

Fibre-optic cables are similar to coaxial cables as they can run for long distances. However, use glass to propagate light allowing very high speeds.

Fibre-optic cables are:

- more costly than coaxial cables; and
- widely used for transmission in network backbones or high-speed networks.

IEEE standards

The Institute or Electrical and Electronic Engineers (IEEE) started a protect to set a standard to enable intercommunication among equipment from a variety of manufacturers.

This project was called ”802” and includes:

- ”802.3” for Ethernet; and
- ”802.11” for Wi-Fi.

20.3 Ethernet history and evolution

Definition

Ethernet is the most commonly used protocol in local area networks (LANs) and defines standards at:

- the software level (data link layer protocols); and
- the hardware level (physical layer protocols).

History

Classic Ethernet

Bob Metcalfe created the Ethernet in 1976 at Xerox.

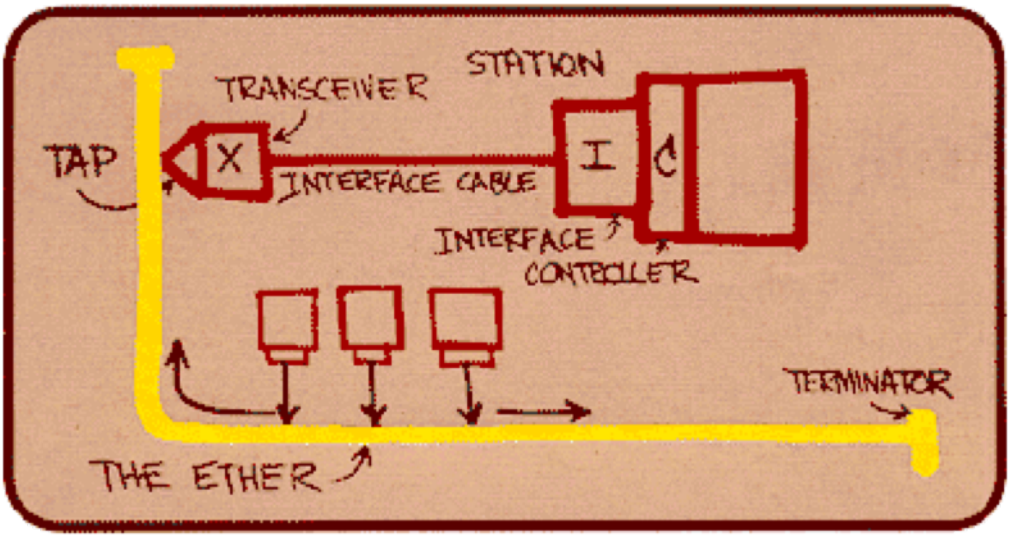


Figure 20.5: Classic Ethernet.

As shown in the figure above, **Classic Ethernet** consisted of a single long cable to which all computers were attached.

In 1983, Ethernet became the IEEE 802.3 standard.

Thin Ethernet

The first variety of Ethernet to be used in practice was **thick Ethernet (10 Base-5)**. This was succeeded by **thin Ethernet (10 Base-2)**.

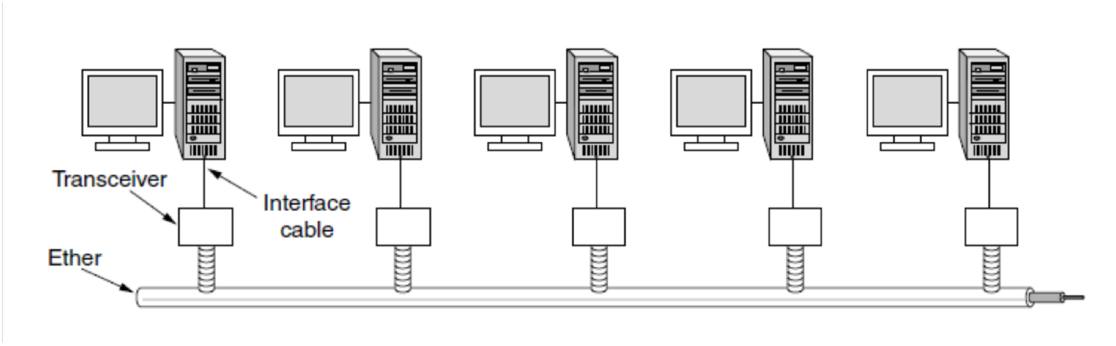


Figure 20.6: Thin Ethernet (10 Base-2).

As shown in the figure above, thin Ethernet (10 Base-2) consists of a bus network topology connected via a half-duplex coaxial cable.

A **bus network topology** involves a single cable connecting all networked devices, with a terminator at the end of the bus. The terminator is a piece of hardware which prevents signals in the cable from echoing, where they bounce back and forth.

This topology is **half-duplex**, meaning that data and information cannot travel in opposite directions at the same time.

This yielded transmission between of 3-10Mbps.

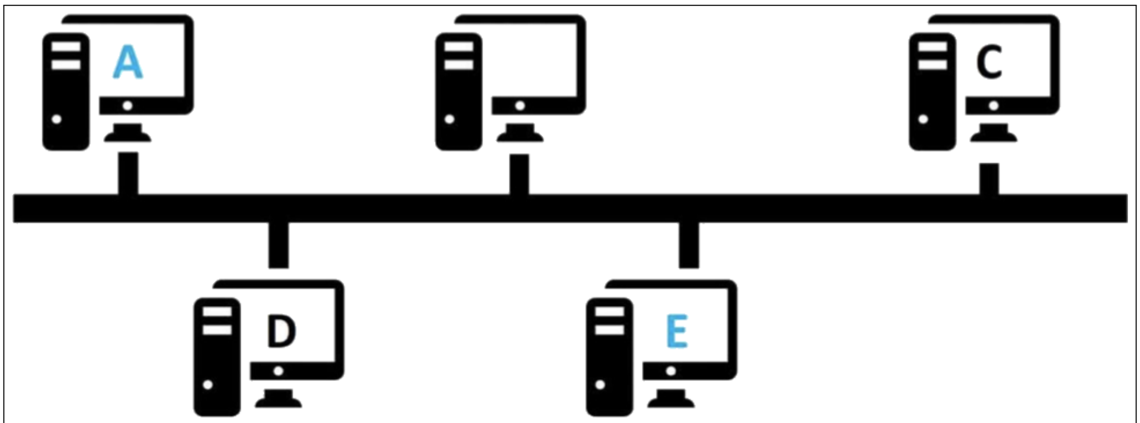


Figure 20.7: Shared bus in bus network topology.

As shown in the figure above, the bus network topology used in Ethernet consists of a shared bus.

As the machines share the same medium, collision may occur as attempts at transmission may be made simultaneously.

Therefore, it is necessary to implement **Carrier Sense Multiple Access/Collision Detection (CSMA/CD)**.

In CSMA/CD:

- if a machine wishes to transmit along the medium, it ”listens” (electronically testing the medium) to the medium and waits for it to become free;
- if the channel is busy, the machine waits until it becomes idle; and

- when the channel becomes idle, data transmission begins immediately but continues to ”listen” to the medium.

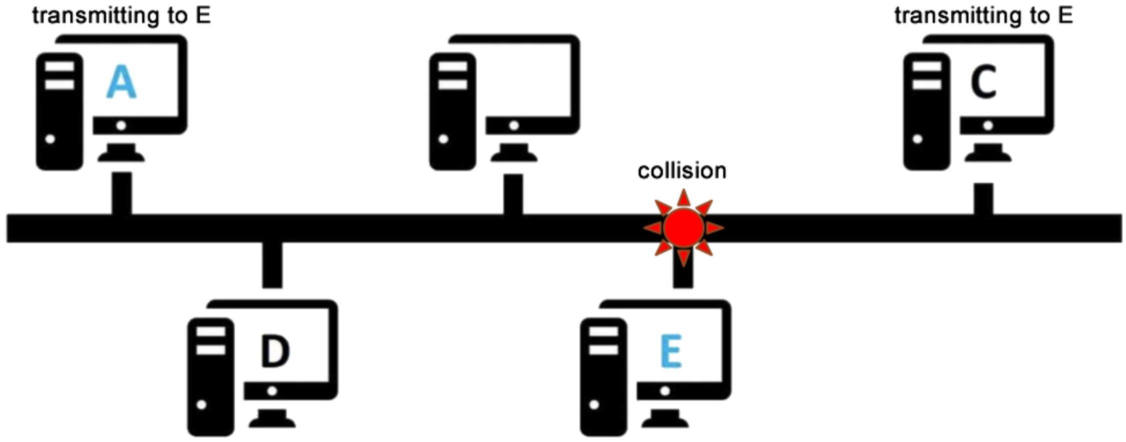


Figure 20.8: Collision on a shared bus.

As shown in the figure above, a collision occurs if two machines sense that the channel is idle and begin transmitting simultaneously. If a collision is detected, a machine stops transmission and waits a random amount of time before re-transmitting, known as the back-off time. The back-off time can be random as there is a very low probability that two machines would calculate the same random number.

Due to these issues, Ethernet evolved away from this single long cable architecture.

Evolution

Switched Ethernet

The evolution of Ethernet, and its departure from the single long cable architecture, involved the introduction of switched Ethernet.

A **switch** is a piece of hardware which connects multiple devices together in a network using a physical connection. When a packet is received, the destination address in the packet's header is examined and the packet is only re-broadcast to the device to which it was intended to be sent.

In **switched Ethernet**, each machine has a dedicated cable running from its **Network Interface Card (NIC)** to a physical port on a central device called the **switch**, as shown in the figure below.

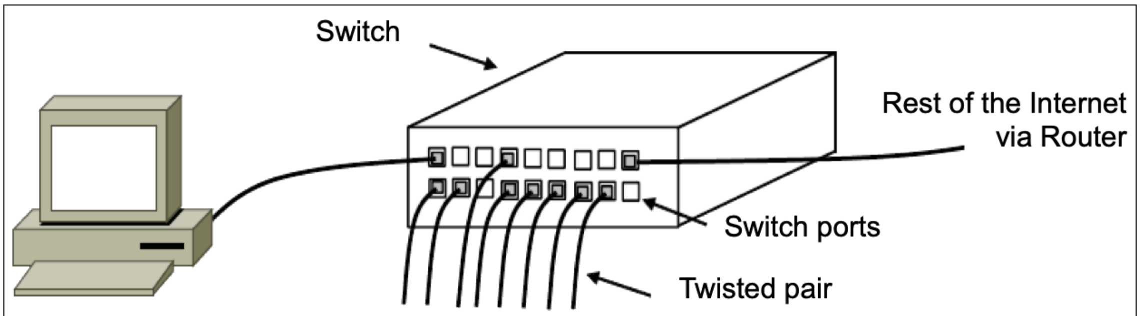


Figure 20.9: Switched Ethernet.

In this method:

- packets (or frames) are passed to a physical port on the switch;
- the destination address in the packet's header is examined;
- the packet is re-broadcast to the correct host connected to the switch.

As such:

- switches only output frames to the ports for which those packets (or frames) are destined; and
- hosts on the network need not be concerned with the switch as it is not an important factor to their method of sending and receiving data on the network, instead host on the network need only be concerned with using their network interface card (NIC) to send and receive on the network.

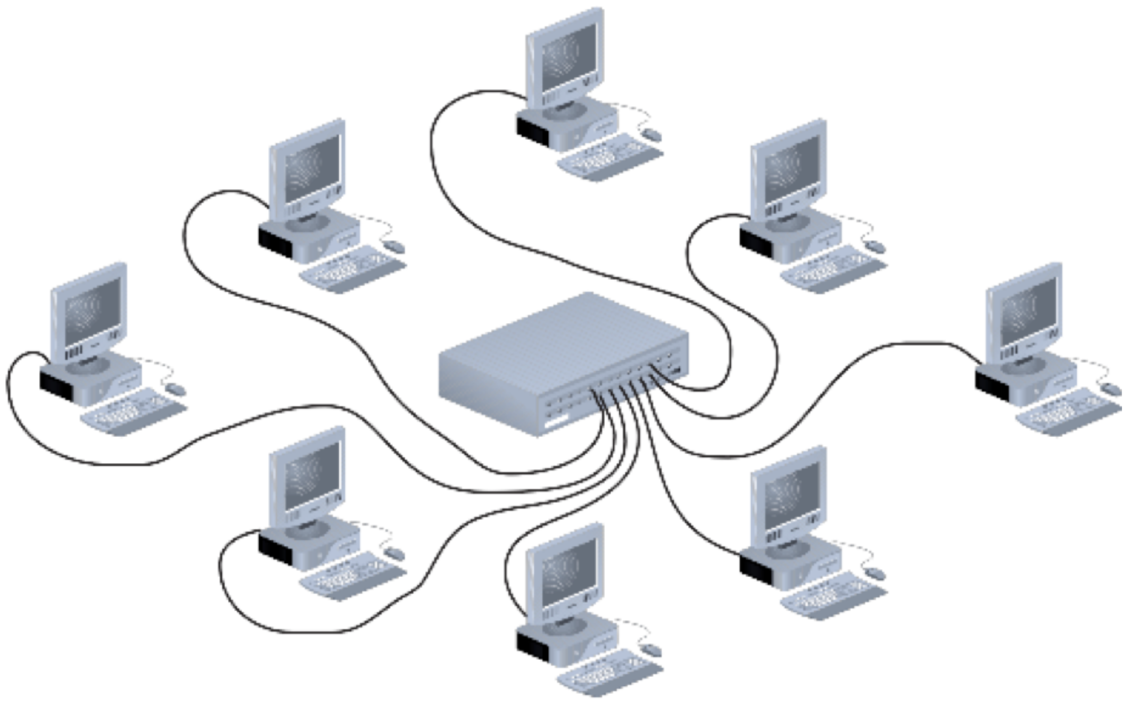


Figure 20.10: Star network topology in switched ethernet.

As shown in the figure above, this method uses a star network topology on a twisted pair cable (other implementations also support fibre-optic cables).

A **star network topology** involves a central node, such as a switch or computer acting as a router, which keeps a record of the unique MAC addresses on the network and can route data to the correct computer system.

This topology is **full-duplex**, meaning that data and information can travel in opposite directions at the same time.

The full duplex nature of the star network topology removes the need for CSMA/CD as used in thin Ethernet.

Timeline

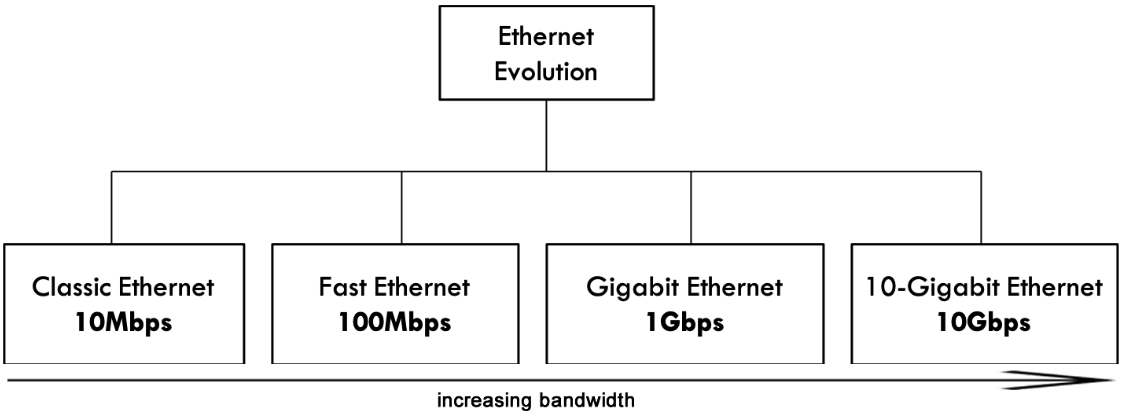


Figure 20.11: Ethernet evolution timeline.

As shown in the figure above, the bandwidth permitted by Ethernet increased as the protocol evolved.

20.4 MAC address

Definition

A **Medium Access Control (MAC) address** is a hardware identification number that uniquely identifies each device on a network.

How it works

Each device on an Ethernet network has its own Network Interface Card (NIC).

Figure 20.12: MAC address sticker.

As shown in the figure above, each NIC has a 6-byte physical address, known as the MAC address.

MAC addresses are:

- expressed in hexadecimal notation; and
- managed by IEEE.

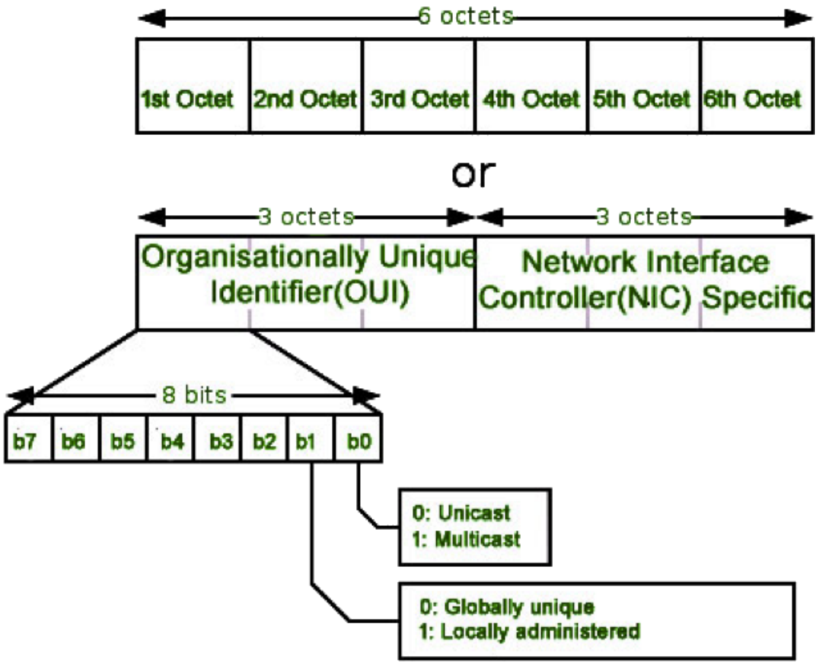


Figure 20.13: MAC address contents.

As shown in the figure above:

- the first three bytes identify the **Organisation Unique Identifier (OUI)**, this is assigned by the IEEE and identifies the vendor;
- the last three bytes identify the NIC specific identifier, assigned by the vendor to each NIC and identifies the device.

The MAC address is stored in ROM to allow each device to be aware of its own ID.

The **Address Resolution Protocol (ARP)** is used to map the IP address (logical) to the MAC address (physical).

Example

In the MAC address `00:60:2F:00:2A:4F`:

- the OUI is `00:60:2F`; and
- the NIC specific identifier is `00:2A:4F`.

20.5 Ethernet frame

Overview

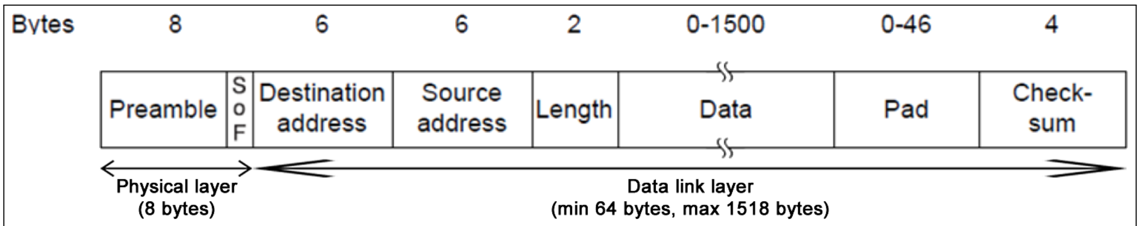


Figure 20.14: Ethernet frame.

The figure above shows that:

- the physical layer is 8 bytes; and
- data link layer has a minimum size of 64 bytes and a maximum size of 1518 bytes.

In addition:

- the fields before the data constitute the **Ethernet header**; and
- the fields after the data constitute the **Ethernet trailer**.

Ethernet frame at the physical layer

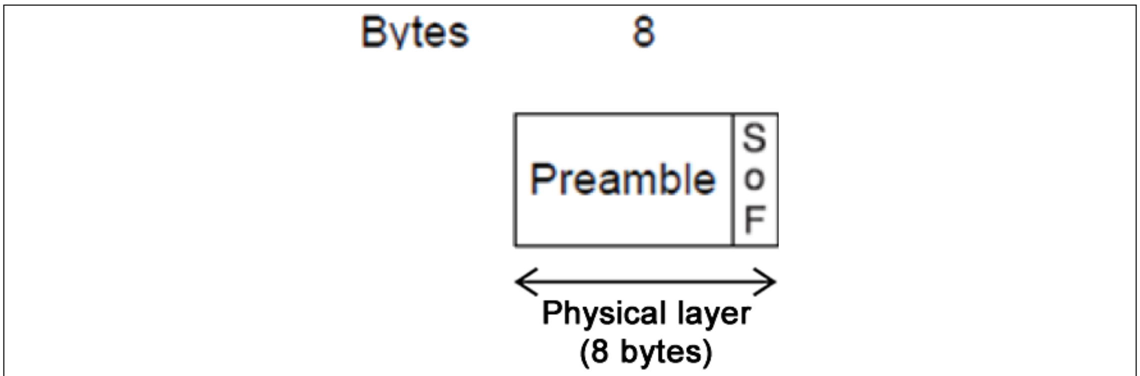


Figure 20.15: Ethernet frame - physical layer.

As shown in the figure above above, the Ethernet frame at the physical layer contains:

- **preamble** – contains 7 bytes of alternating 0s and 1s to enable synchronization ; and
- **Start of Frame (SOF)** – contains 1 byte (10101011) which signals the beginning of the frame.

Ethernet frame at the data link layer

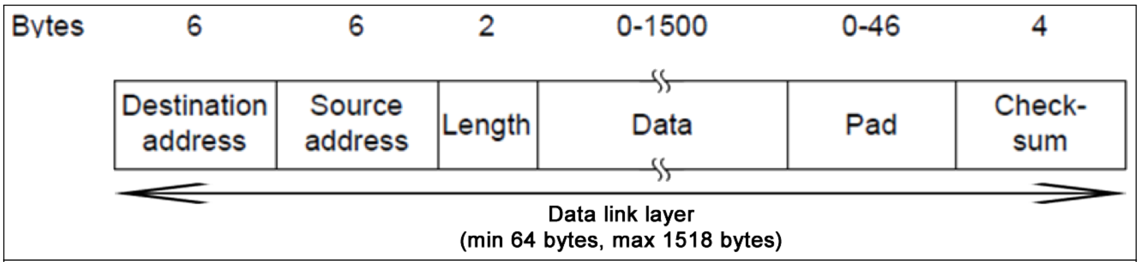


Figure 20.16: Ethernet frame - data link layer.

As shown in the figure above above, the Ethernet frame at the data link contains:

- **destination address** – indicates the physical address (MAC address) of the destination;
- **source address** – indicates the physical address (MAC address) of the sender;
- **length** (or **type**) – used to define the number of bytes in the data field (or describes internet protocol used);
- **data** – carries the data from upper layers in the TCP/IP stack;
- **pad** – padding used to fill our the frame to the minimum size of the **data** is less than 64 `bytes`
- **checksum** – allows error-checking to be performed.

20.6 Advantages of the Ethernet protocol

Ethernet has been established for over 30 years and has no serious competitors due to the following advantages:

- although Ethernet evolved in terms of speed, **no additional equipment is required**;
- **greater scalability** can be achieved through switching;
- **easy to maintain** because there is no software to install other than the drivers of the NIC;
- **failures are very rare**;
- **cheap** as twisted pair cables are not expensive; and
- **Ethernet interworks easily** with TCP/IP (the Internet).