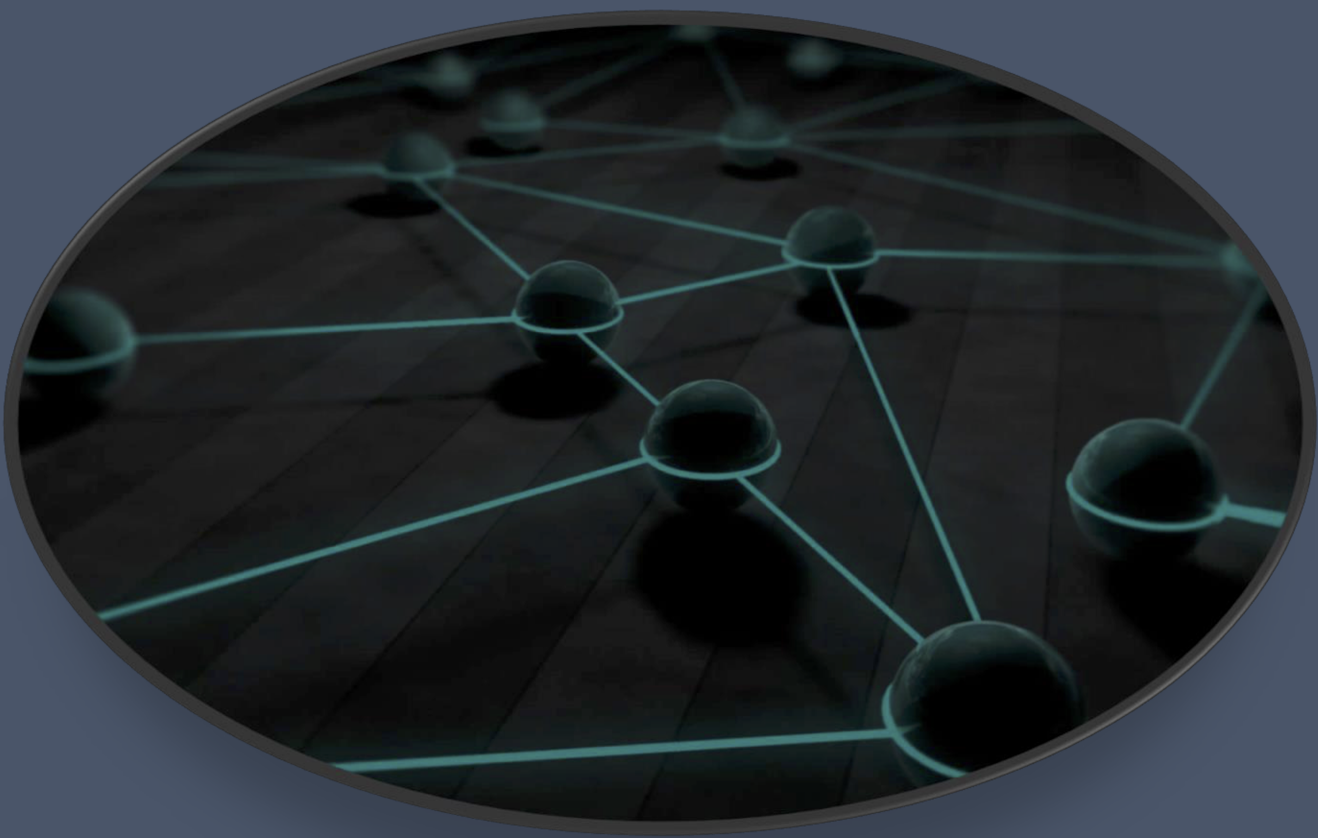
COMP20081

Systems Software

Revision Guide

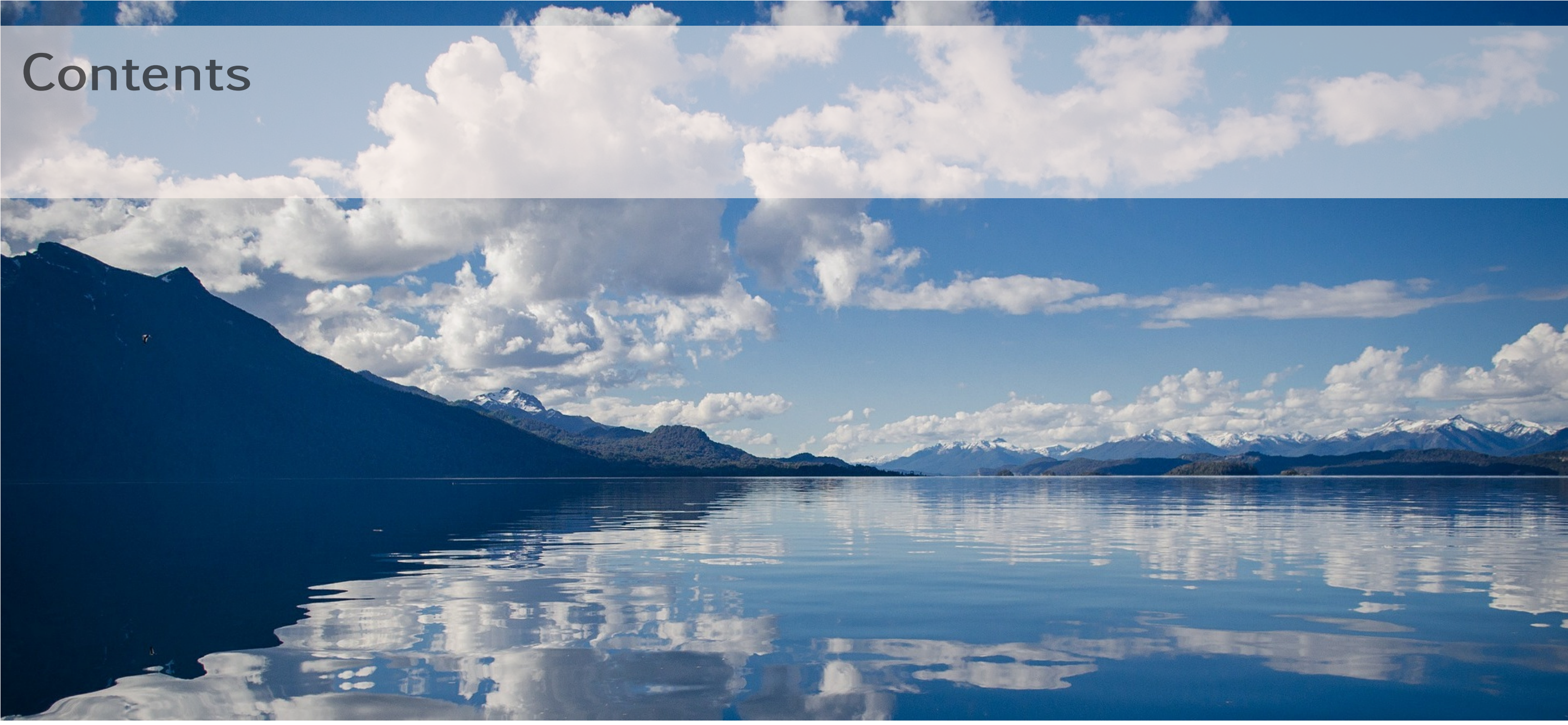


Matthew Robinson

Art. No
ISBN
Edition

Cover design by

Published by
Printed in



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.

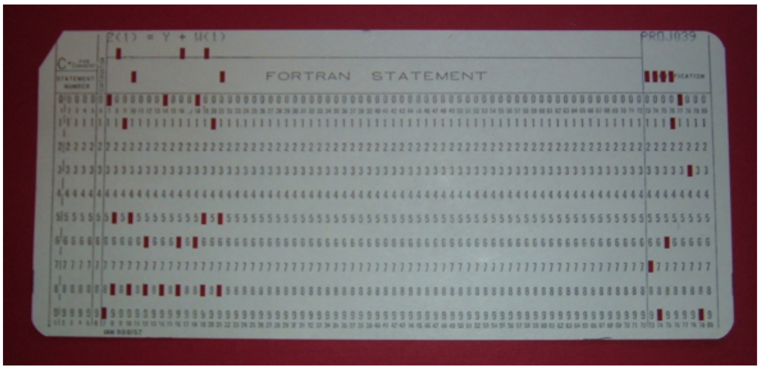


Figure 1.1: Perforated card

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.

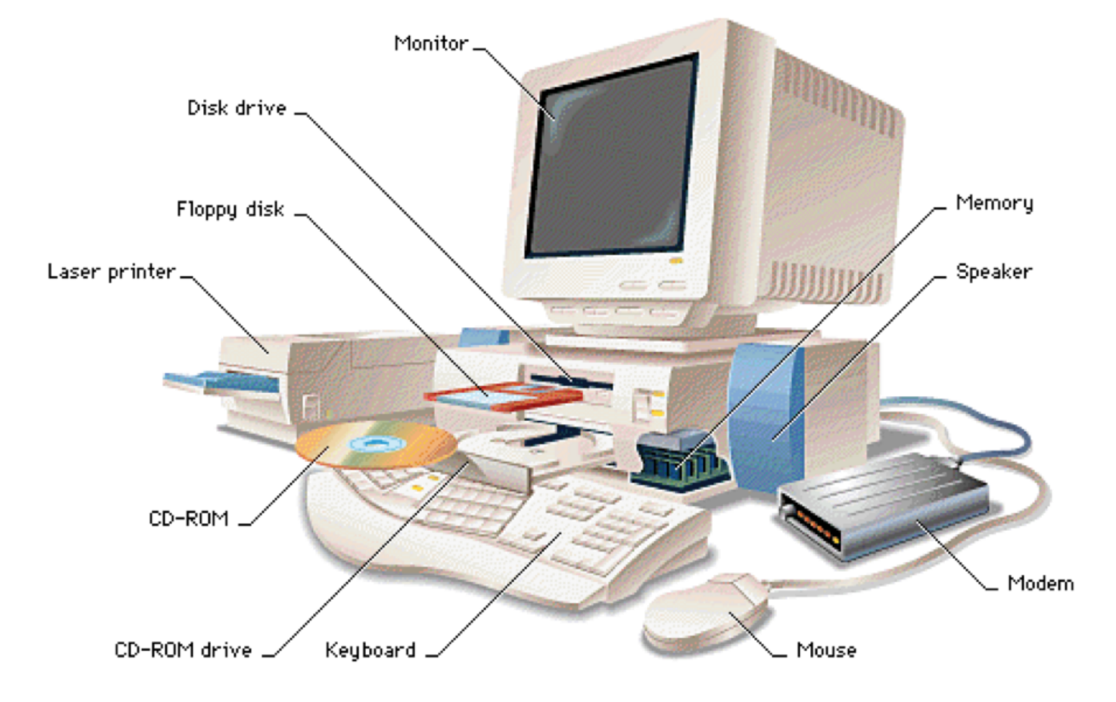


Figure 1.4: External view of computer system

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.

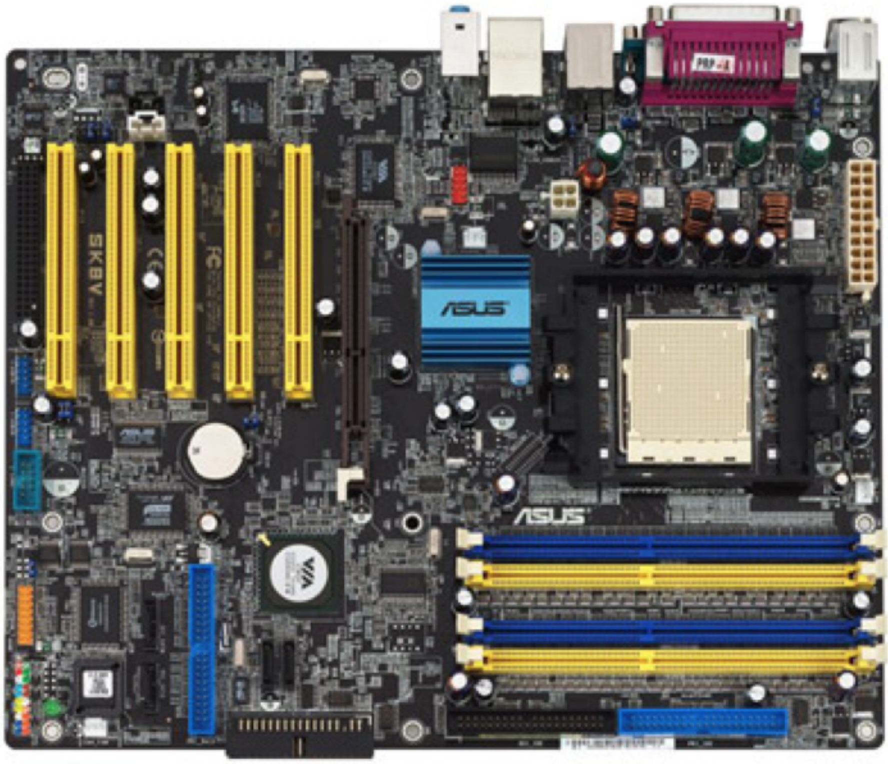


Figure 1.5: Typical computer motherboard

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.

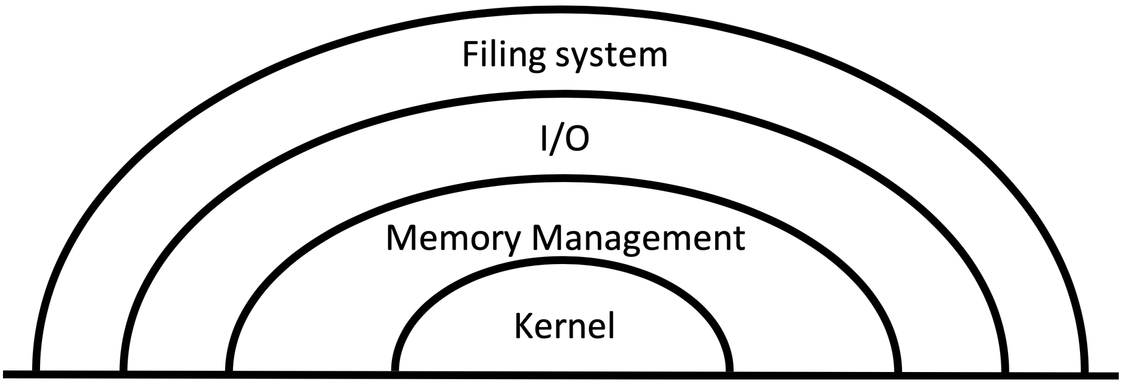


Figure 1.7: Structure of an operating system (OS)

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.

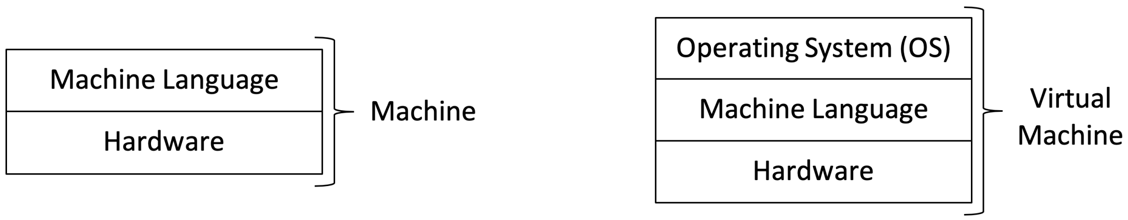


Figure 1.8: Machine extending

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).

<table>
<tr><th>Without Operating System (OS)</th><th>With Operating System (OS)</th></tr>
<tr><td>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.</td><td>The instructions can be written in a high-level language, such as C++.</td></tr>
<tr><td><table>
<tr><td>LDAA $80</td><td>(load number at memory location 80)</td></tr>
<tr><td>LDAB $81</td><td>(load number at memory location 81)</td></tr>
<tr><td>ADDB</td><td>(add these two numbers)</td></tr>
<tr><td>STAA &55</td><td>(store the sum to memory location 55)</td></tr>
</table></td><td>int a, b, c;
a = 1;
b = 2;
c = a + b;</td></tr>
</table>

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.

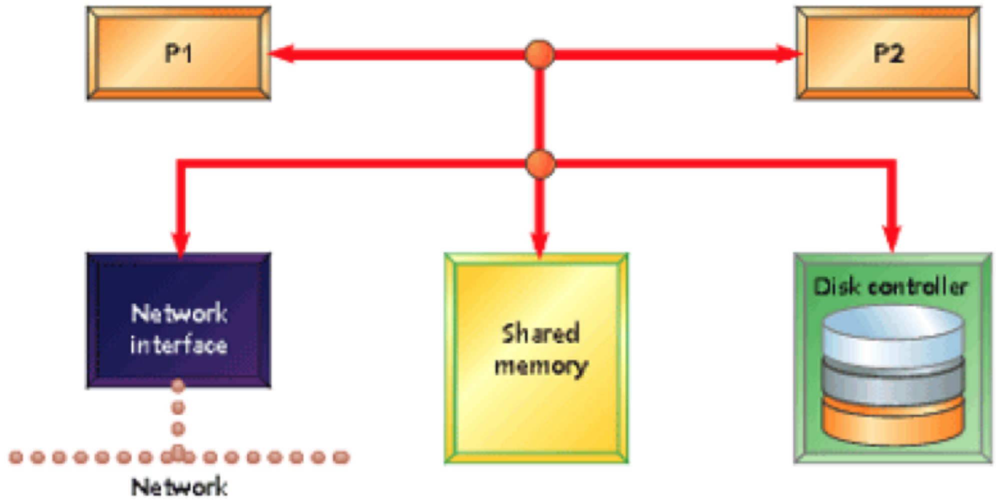


Figure 1.11: Distributed and network systems

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.

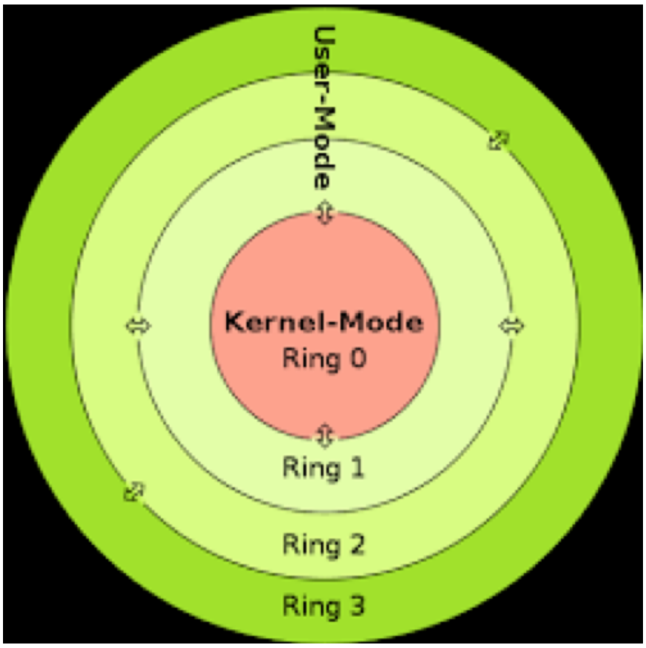


Figure 1.12: Kernel mode and user mode

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.

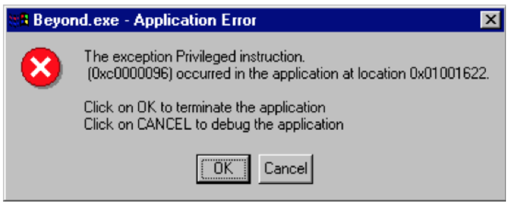


Figure 1.13: Kernel mode errors

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.

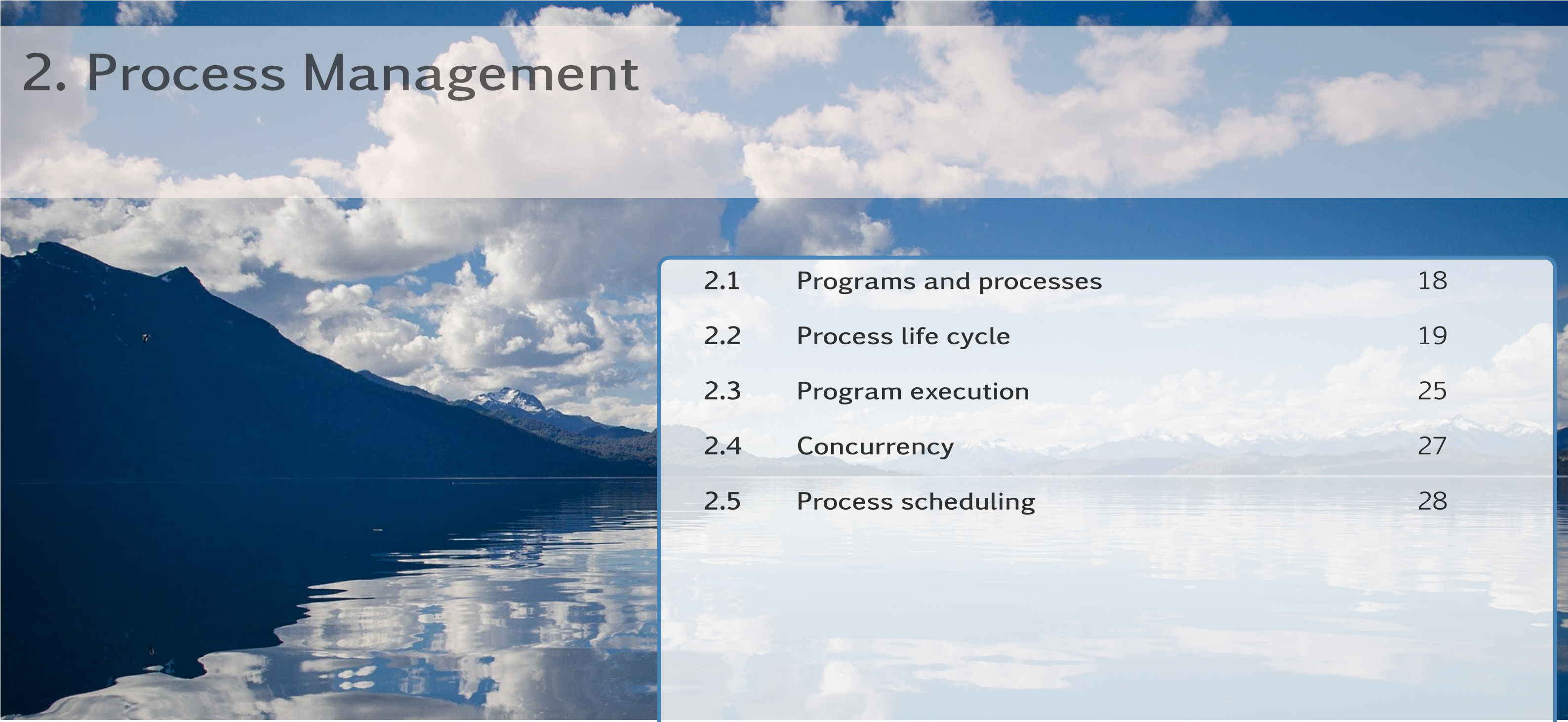
Some privileged instructions can be called by a programmer via system calls. Operating systems (OSs) contain system calls for which provide a layer of services for user programs to implement some activities/request services. These are usually sensitive or privileged from the kernel.

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 Mangament	
pid = fork()	Create a child process identical to the parent.
exit(status)	Terminate process execution and return status.
File Mangament	
n = read(fd, buffer, nbytes)	Read data from a file in to a buffer.
n = write(fd, buffer, nbytes)	Write data to a file
Process Mangament	
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.

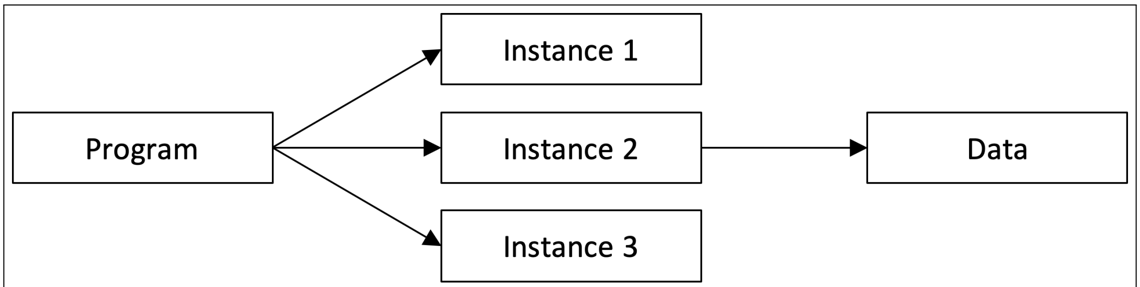


Figure 2.1: Programs and processes

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

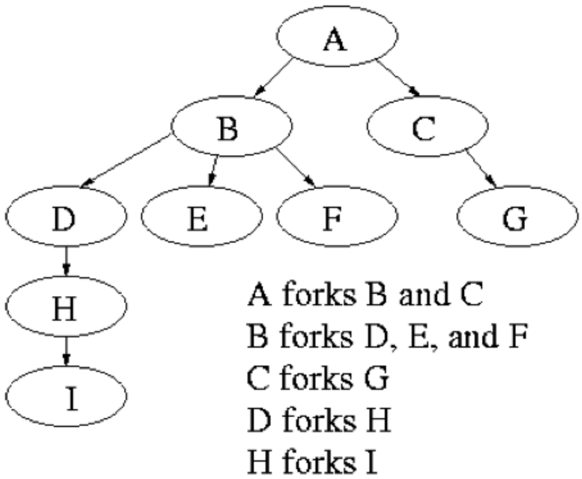


Figure 2.2: Process creation

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).

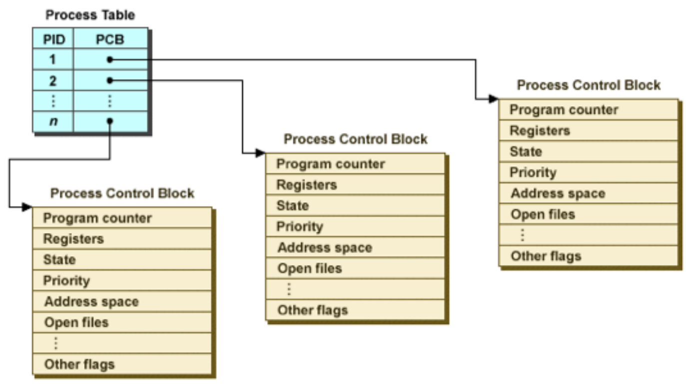


Figure 2.3: Process table with respective process control blocks

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

•ready – 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.

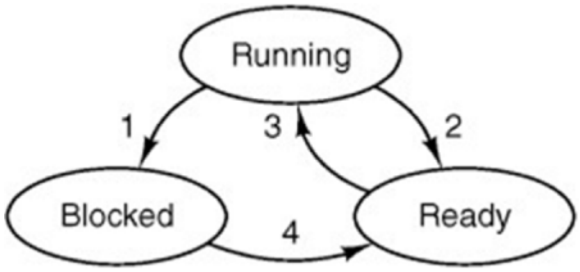


Figure 2.5: Process states

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.

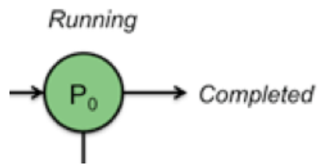


Figure 2.7: Running processes

Ready processes

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

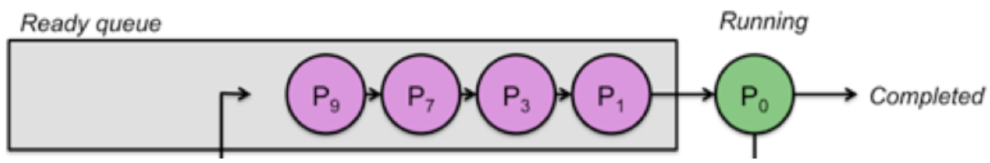


Figure 2.8: Ready processes

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.

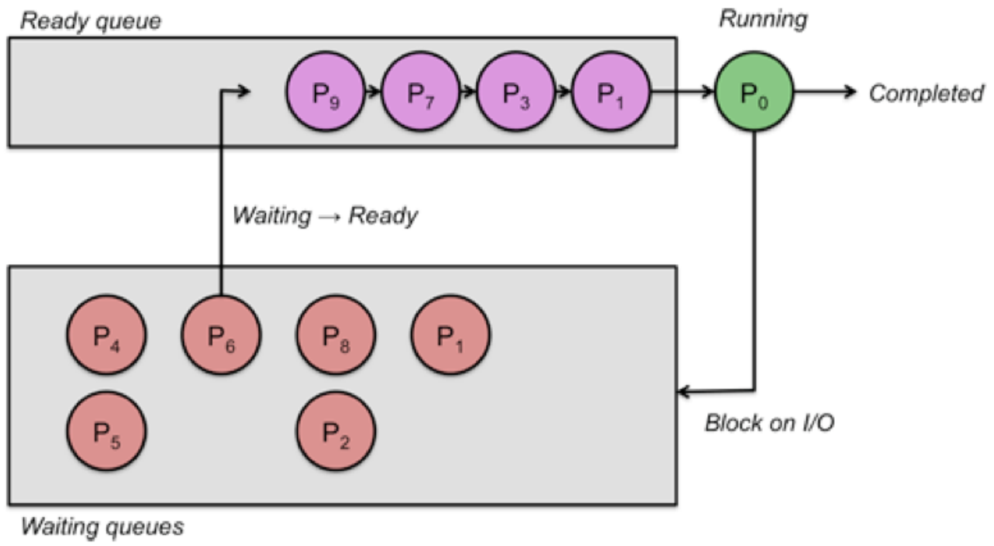


Figure 2.9: Currently running processes

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.

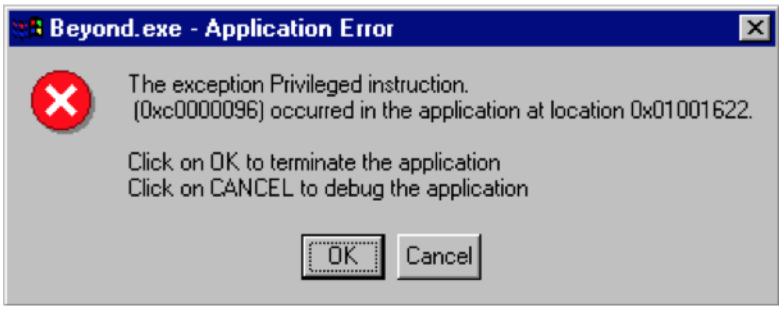


Figure 2.11: Voluntary termination due to an error exit

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

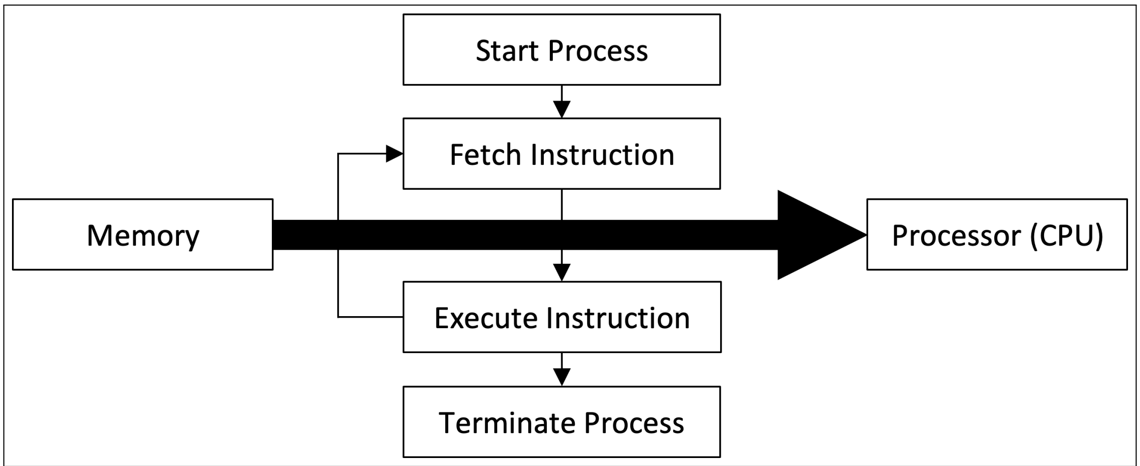


Figure 2.12: Simple fetch-execute cycle

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

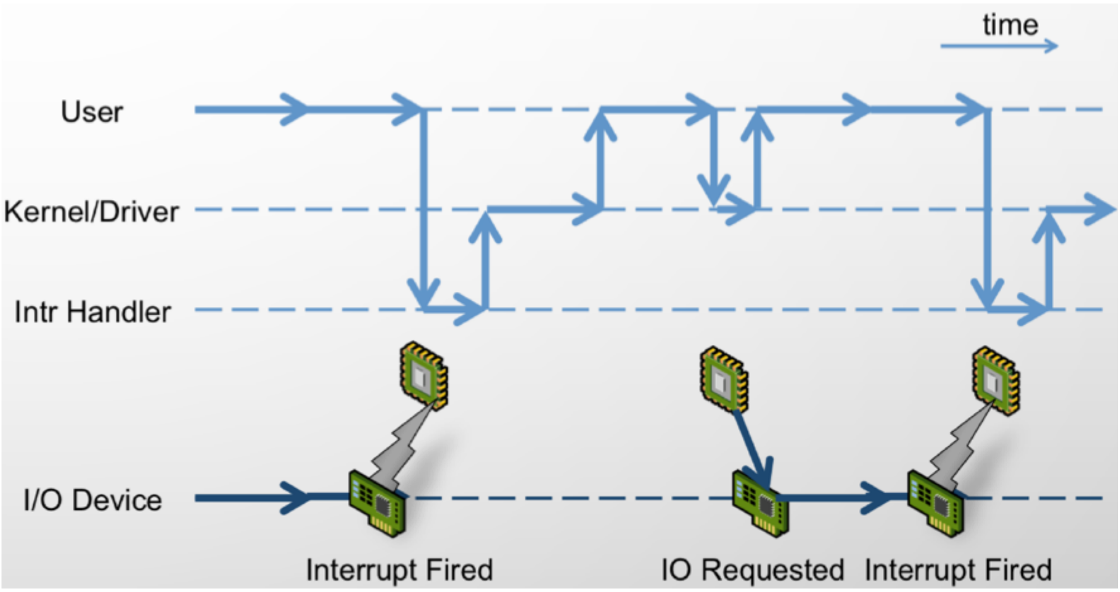
Definition

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.

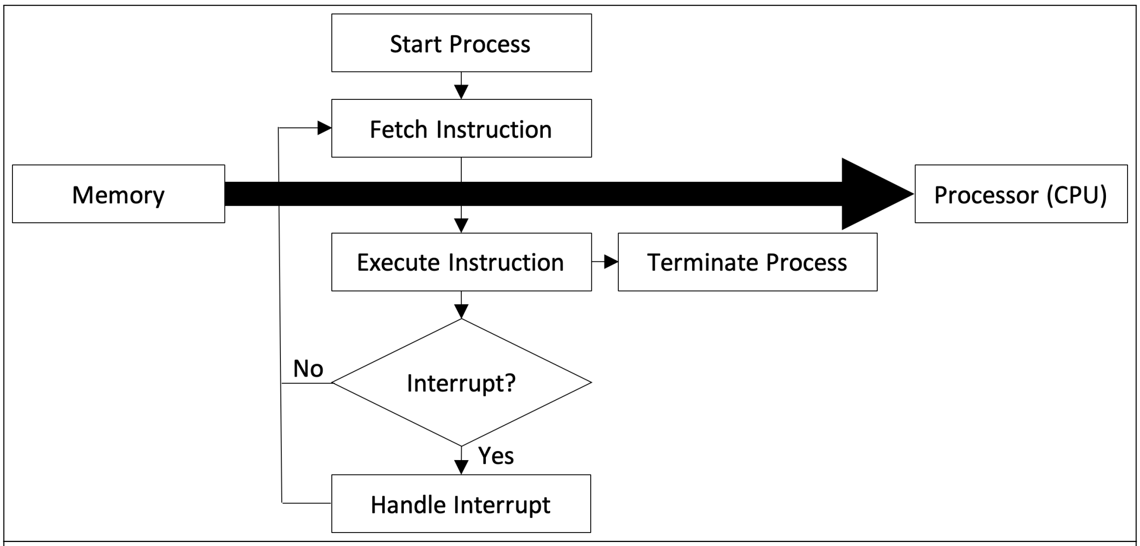


Figure 2.14: Updated fetch-execute cycle

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 is shown by an interrupt occurring due to an input/output (I/O) event occurring (page 24).

Interleaving

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

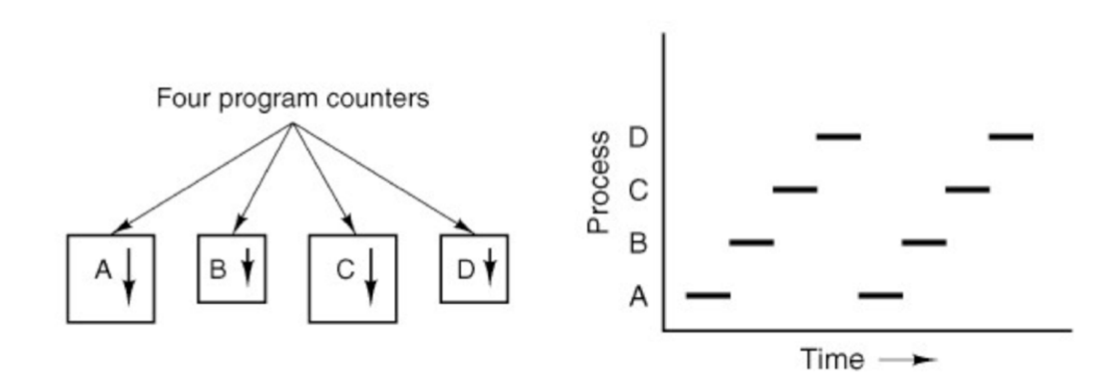


Figure 2.15: Interleaving

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.

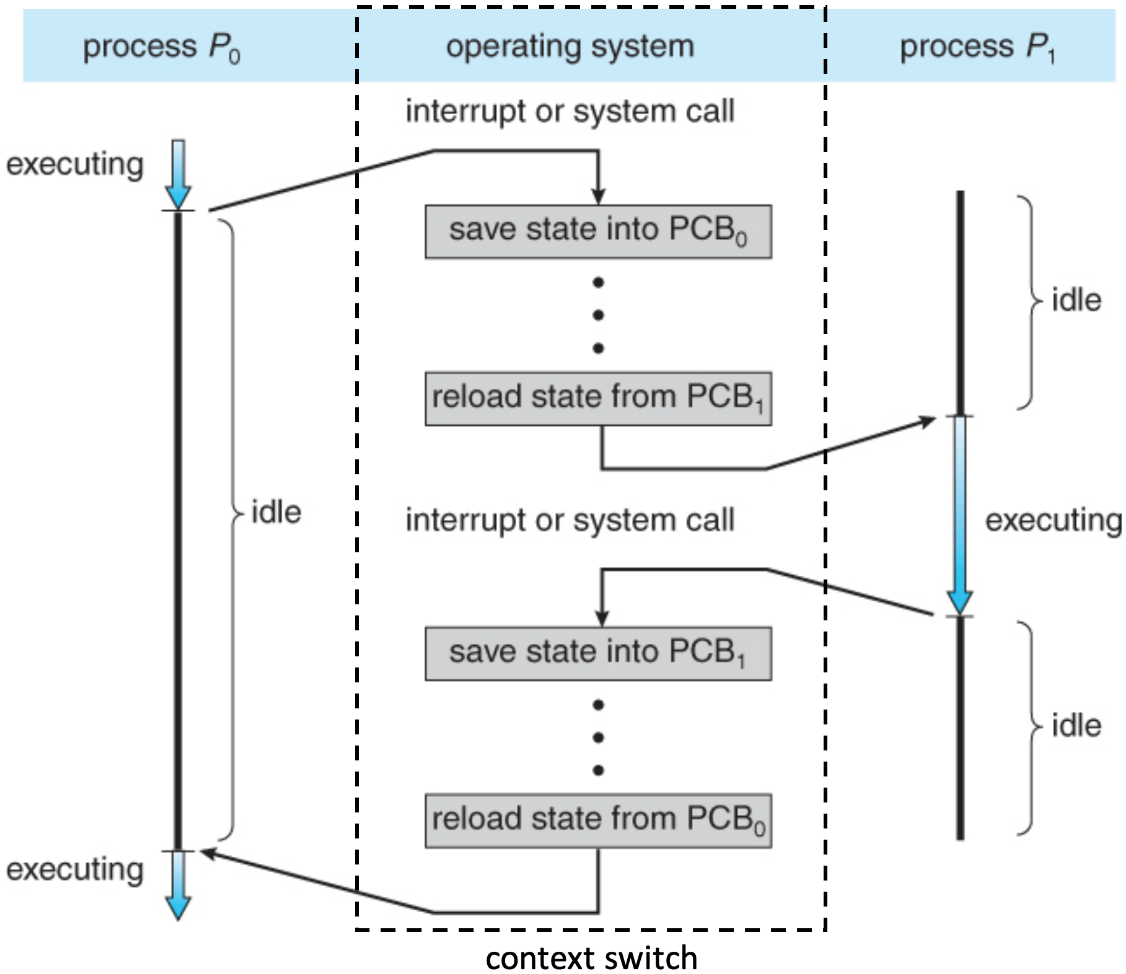


Figure 2.16: The scheduler

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

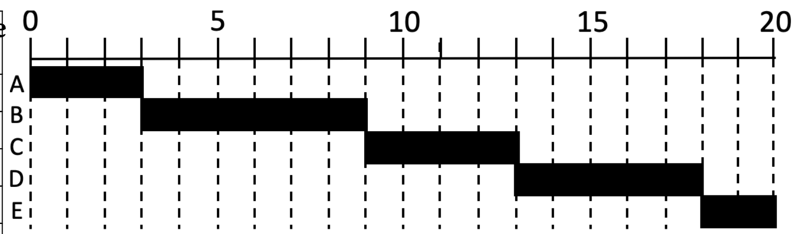


Figure 2.18: Temporal diagram for FCFS / FIFO

Advantages	Disadvantages
Simple to implement.	**Does not consider the priority of a process** and therefore the important processes may not be completed quickly.
	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 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.	

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

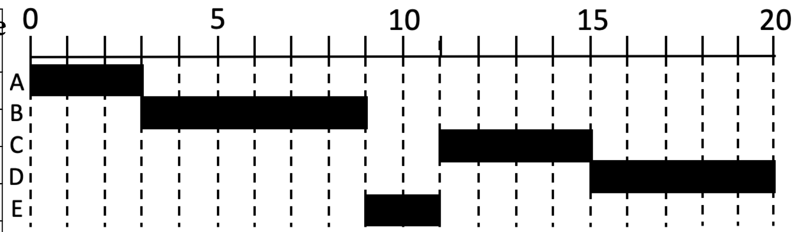


Figure 2.19: Temporal diagram for SJF

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 long 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

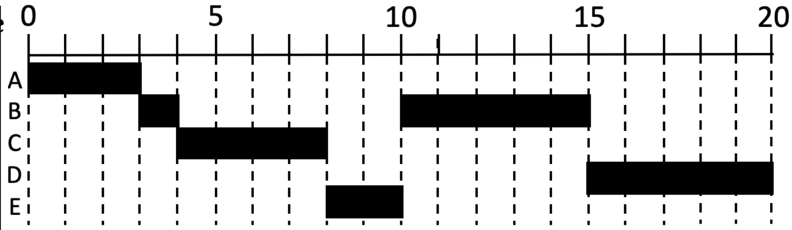


Figure 2.20: Temporal diagram for SRT

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 long 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.	

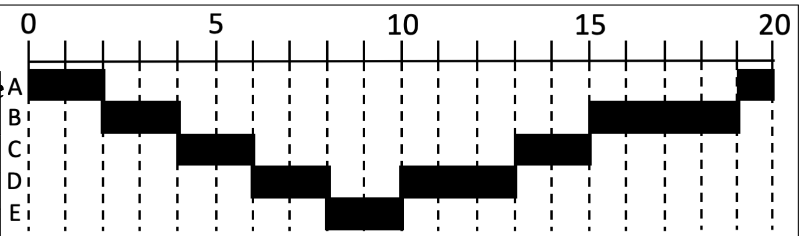
Highest response ratio next (HRRN) – Preemptive

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:

$$\frac{waitingtime + runtime}{runtime}$$

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 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.

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.

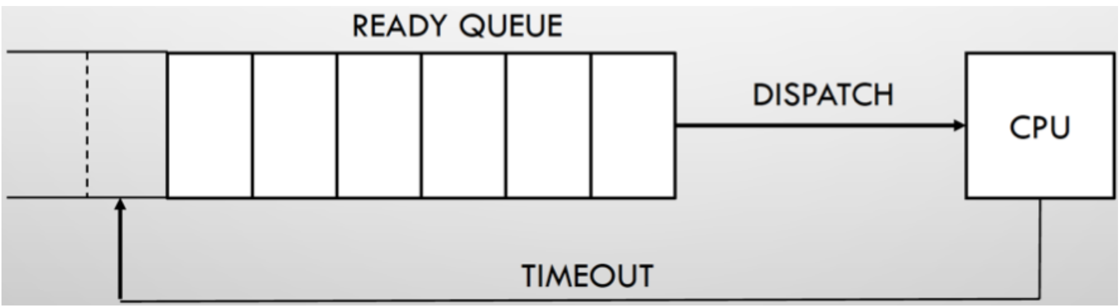


Figure 2.22: RR

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

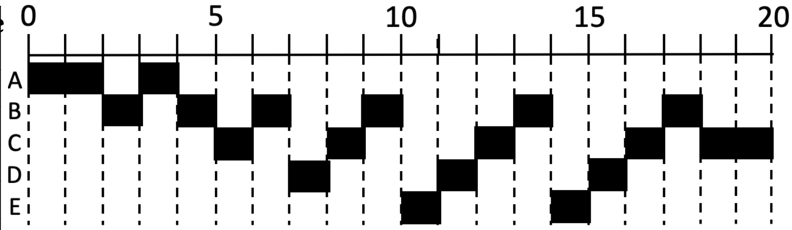


Figure 2.23: Temporal diagram for RR with a fixed time quantum of 1 (q = 1)

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

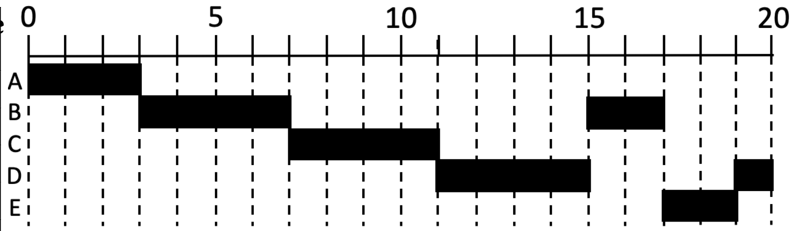


Figure 2.24: Temporal diagram for RR with a fixed time quantum of 4 (q = 4)

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.

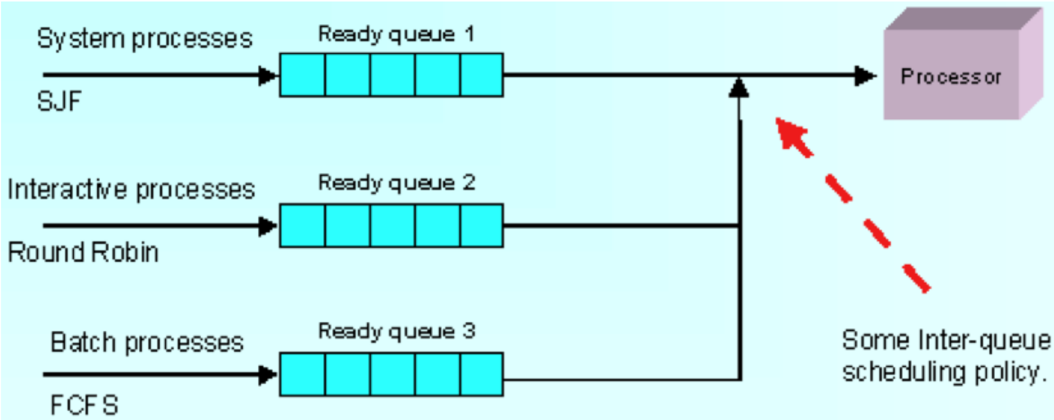


Figure 2.25: Multi-level queueing

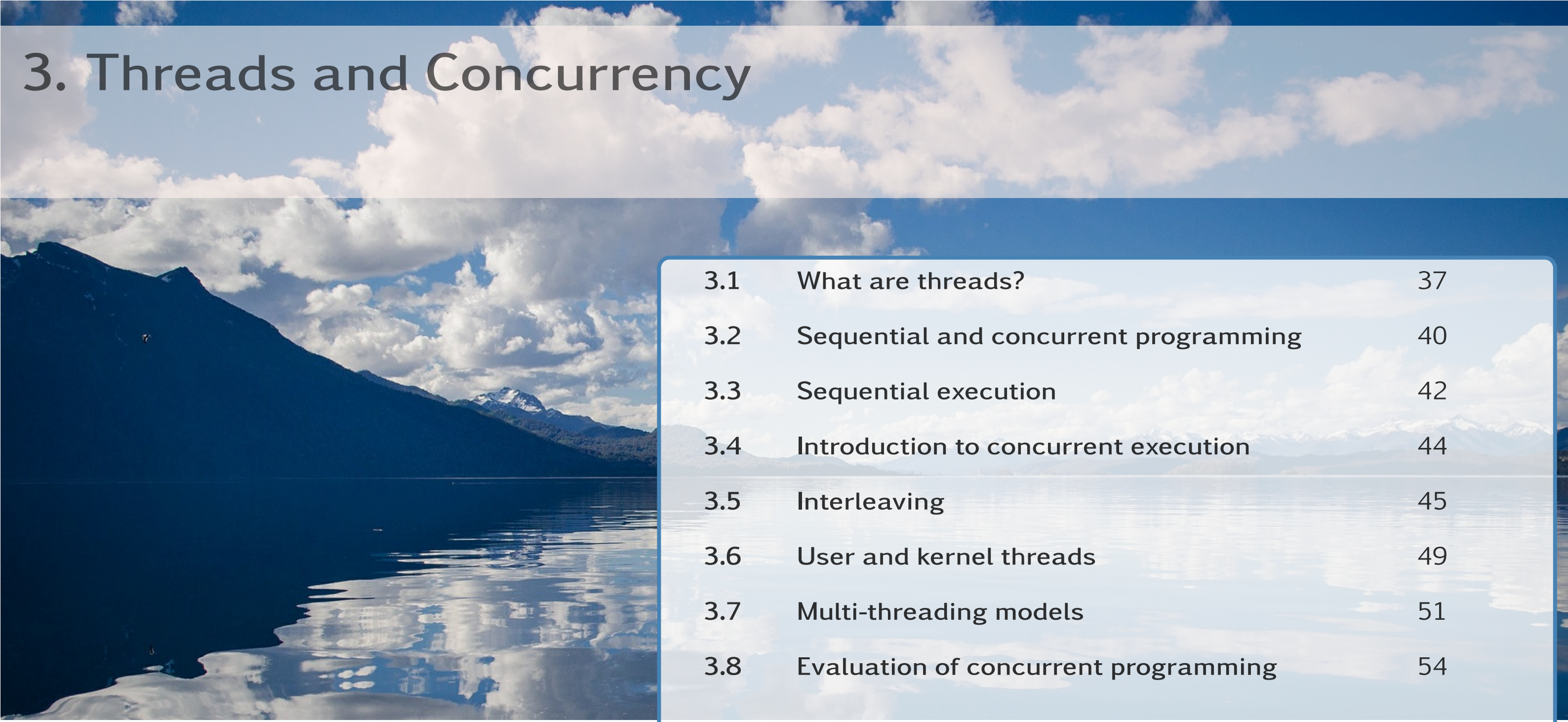
Each queue has a different priority; the top queue has the highest priority and the bottom queue has the lowest priority. Processes are able to move between queues if their priority changes.

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.

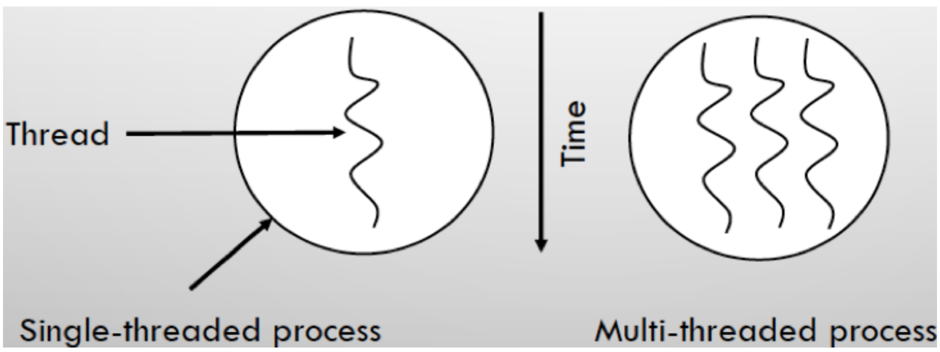
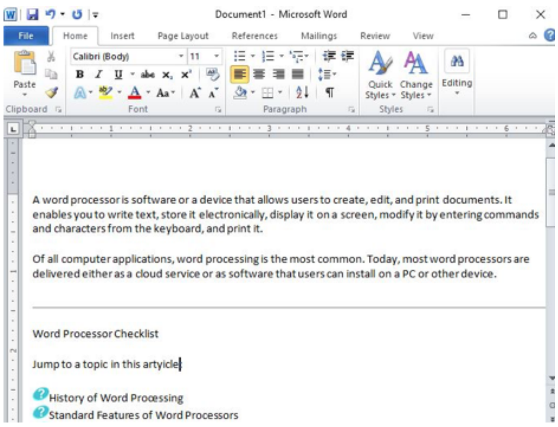


Figure 3.1: Single-threaded vs multi-threaded

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.

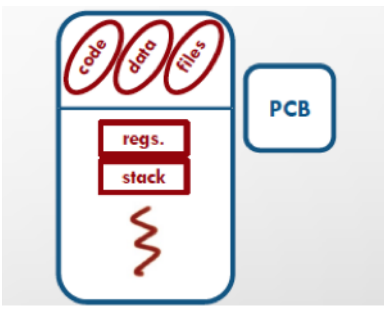


Figure 3.3: Single-threaded process

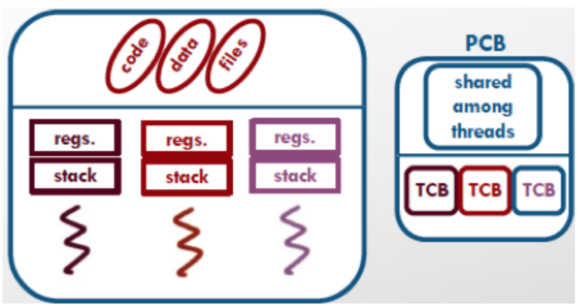


Figure 3.4: Multi-threaded process

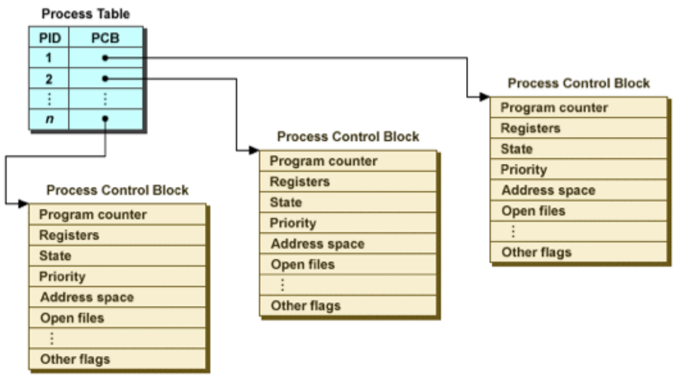


Figure 3.5: Process table and process control blocks

	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 are 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.

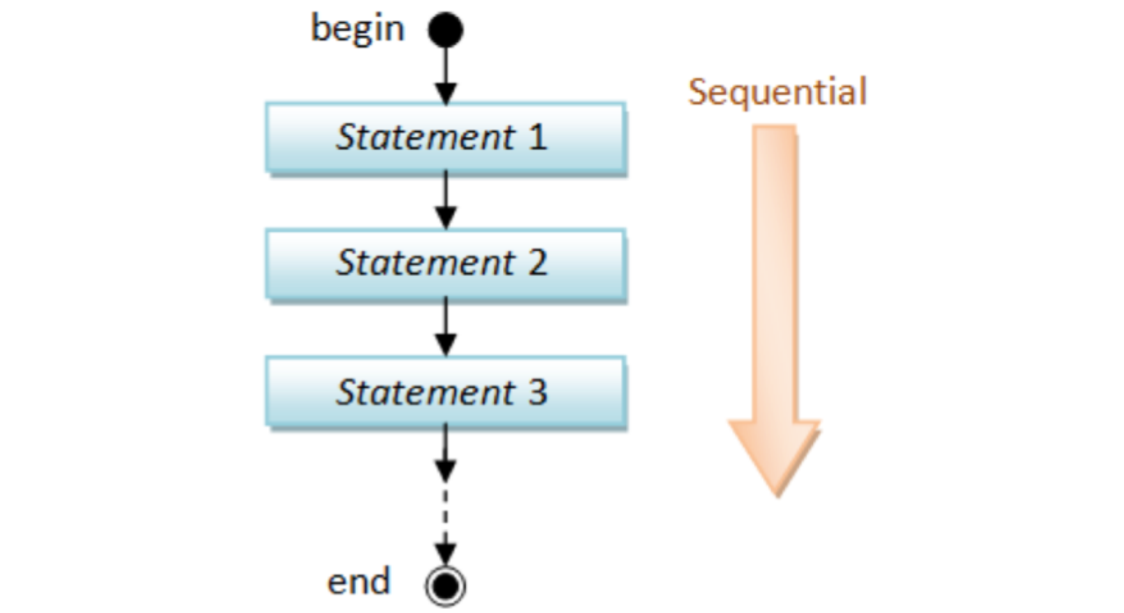


Figure 3.8: Sequential program

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 multitasking; 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

P → **Q** → **R**

implies that the execution at the system level is as follows

P1 → **Q1** → **Q2** → **Q3** → **R1** → **R2** → **R3**,

given that **P** is compiled to a single machine instruction, whilst **Q** and **R** are compiled to three machine instructions – as seen on page 40.

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 (page 41).

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.

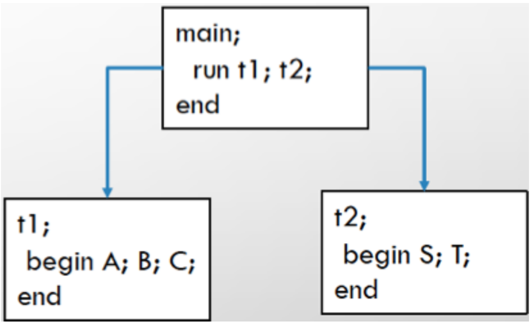


Figure 3.9: Concurrent program spawning threads t_1 and t_2

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.

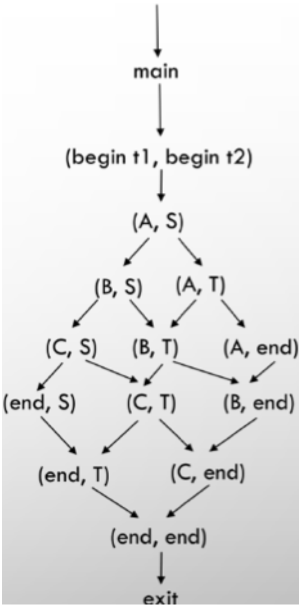


Figure 3.10: Visual representation of the possible execution sequences

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.

and therefore:

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

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.

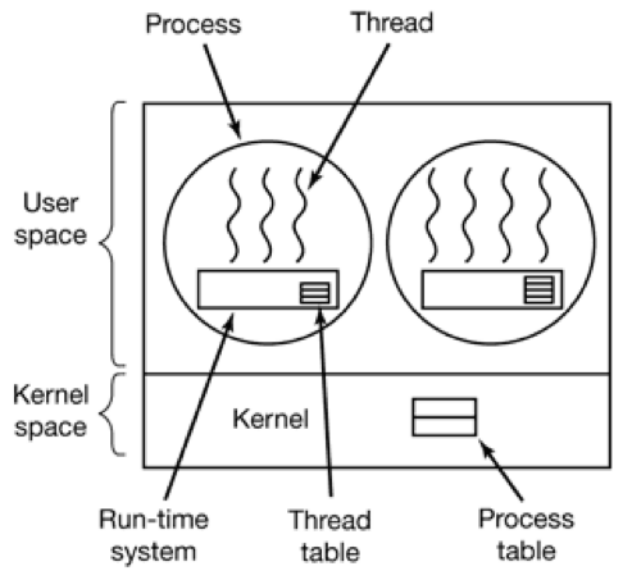


Figure 3.11: User threads

The diagram shows that:

- all of the threads for a given process is 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 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.

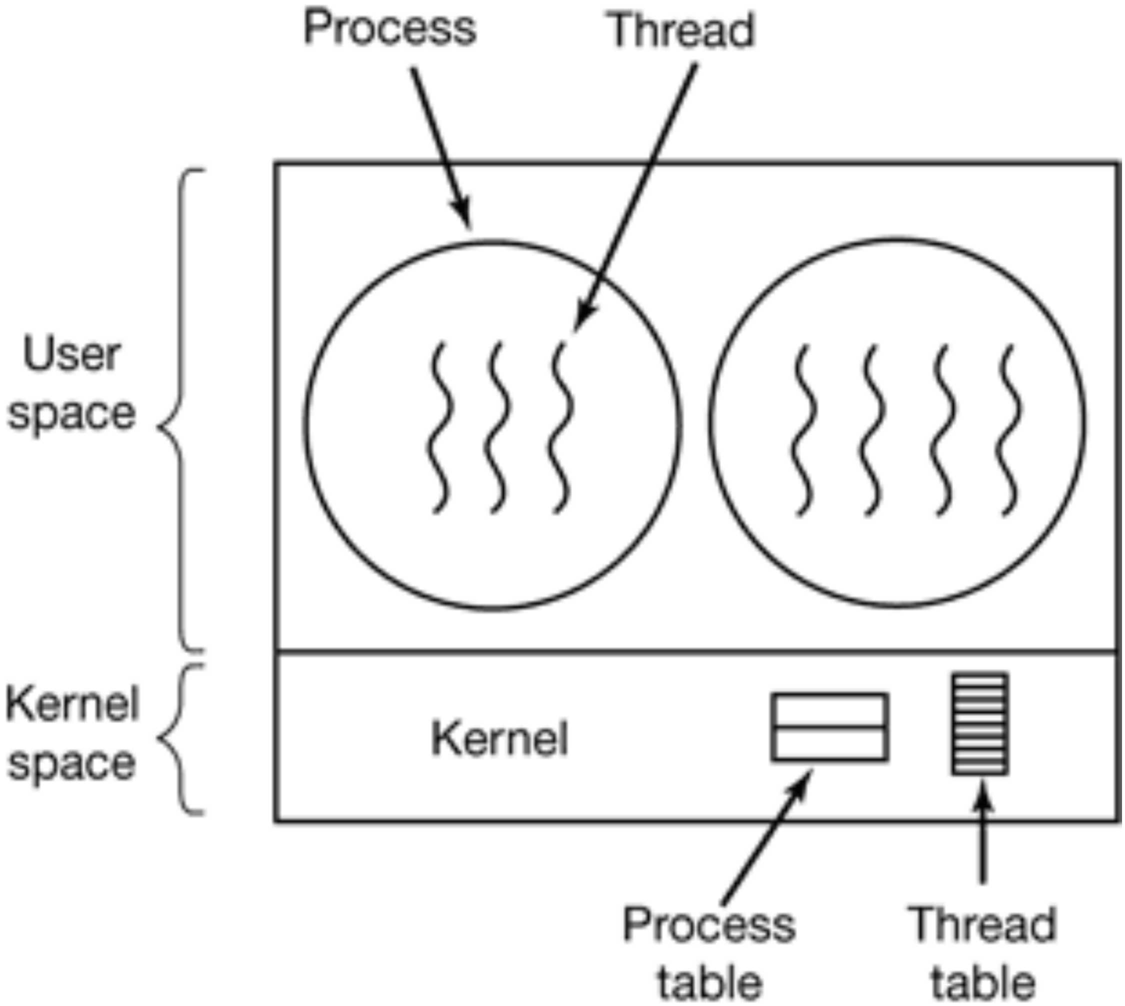


Figure 3.12: Kernel threads

The diagram shows that:

- all of the threads for a given process is 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 user thread is blocked, all other threads in the same process are scheduled and not blocked. For example, in a word processor application, a thread that handles a printing event would no longer block all other threads and therefore would allow the user to interacting with other aspects of the application whilst the printing event occurs.

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.

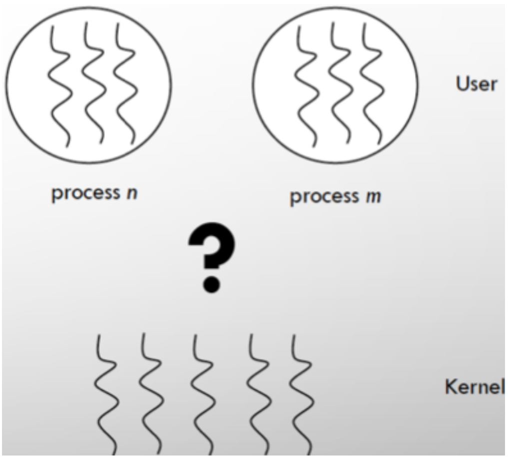


Figure 3.13: Multi-threading mapping

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.

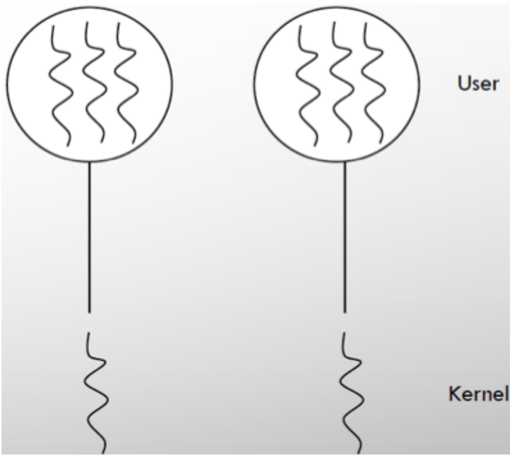


Figure 3.14: Many-to-one mapping

Evalution

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.

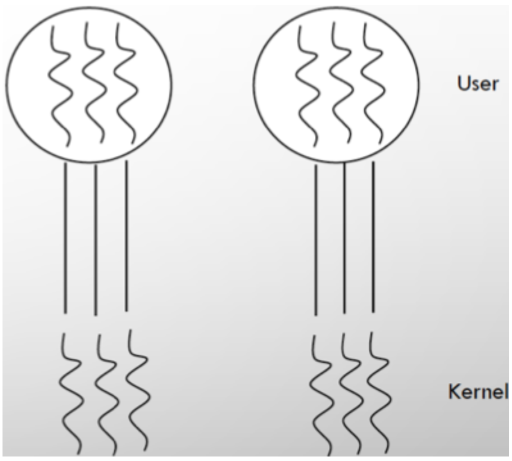


Figure 3.15: One-to-one mapping

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.

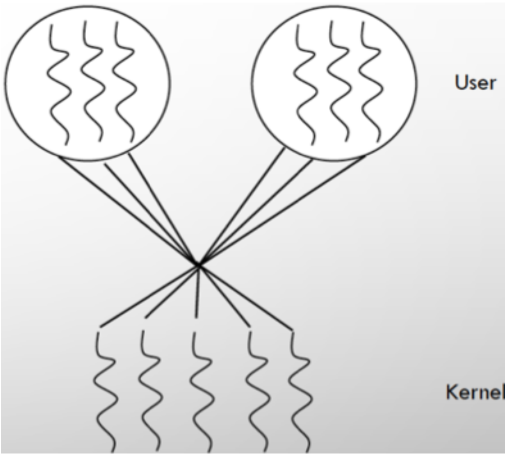
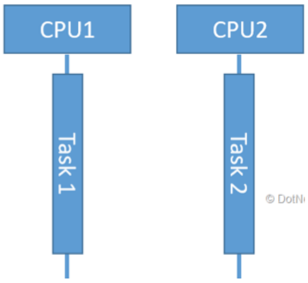
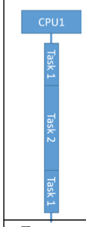


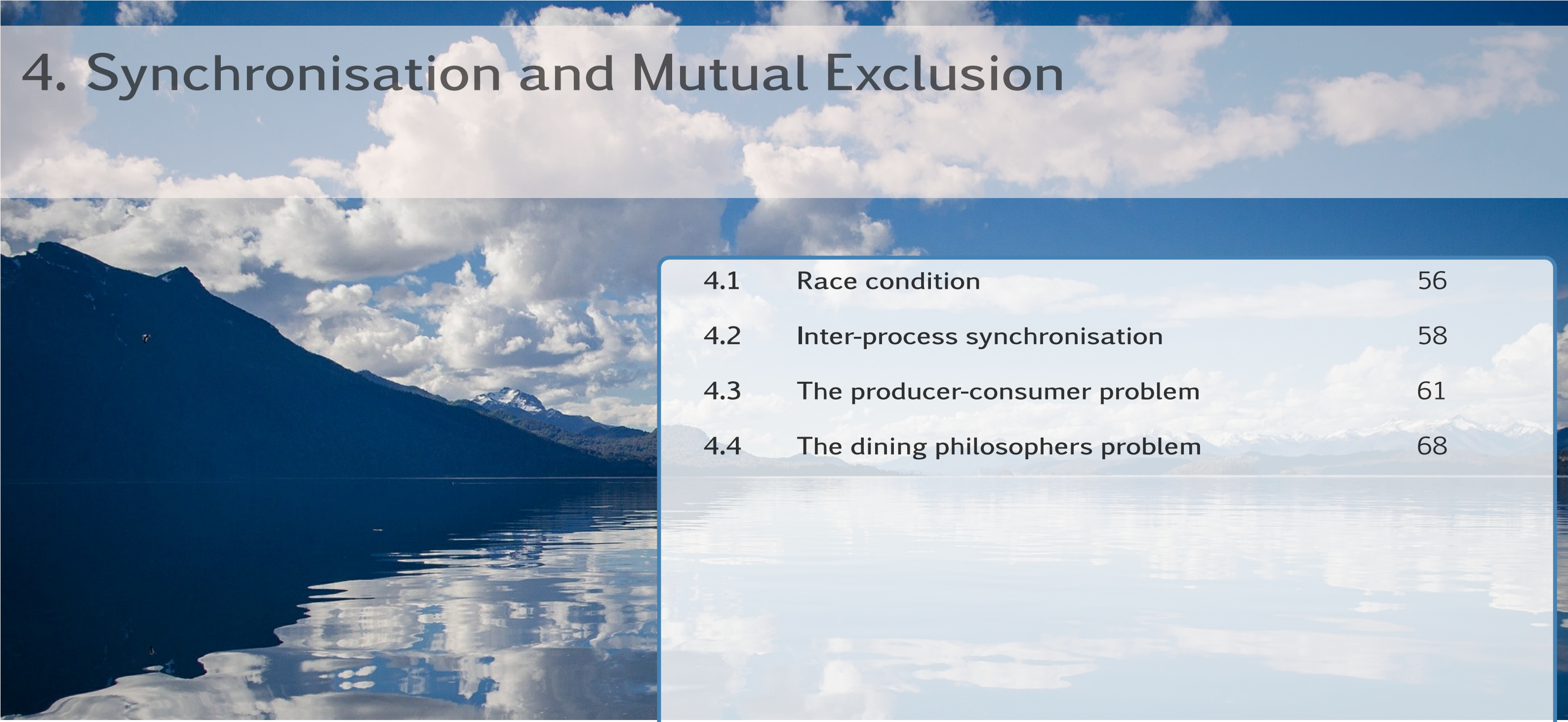
Figure 3.16: Many-to-many mapping

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 allows 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.

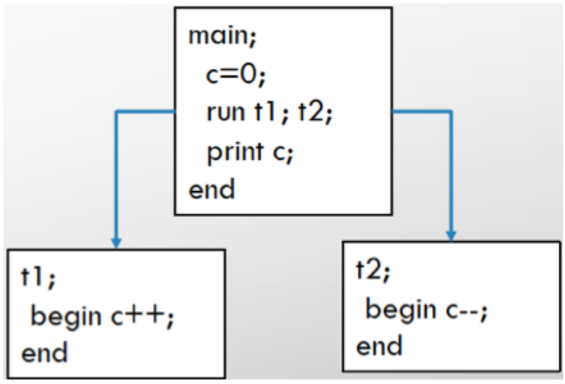


Figure 4.1: Concurrent program spawning threads t_1 and t_2

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.

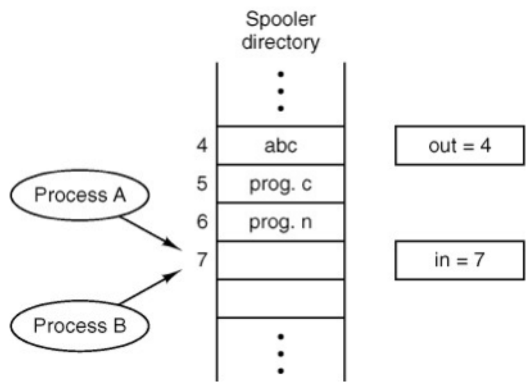


Figure 4.2: Printer spooler directory

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.

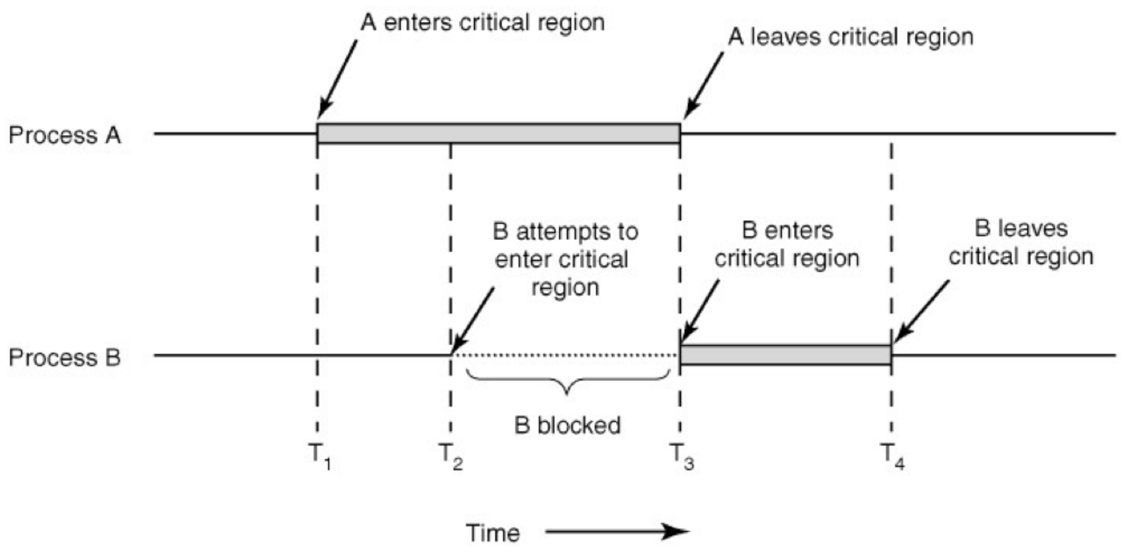


Figure 4.3: Mutual exclusion on critical section/region

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.

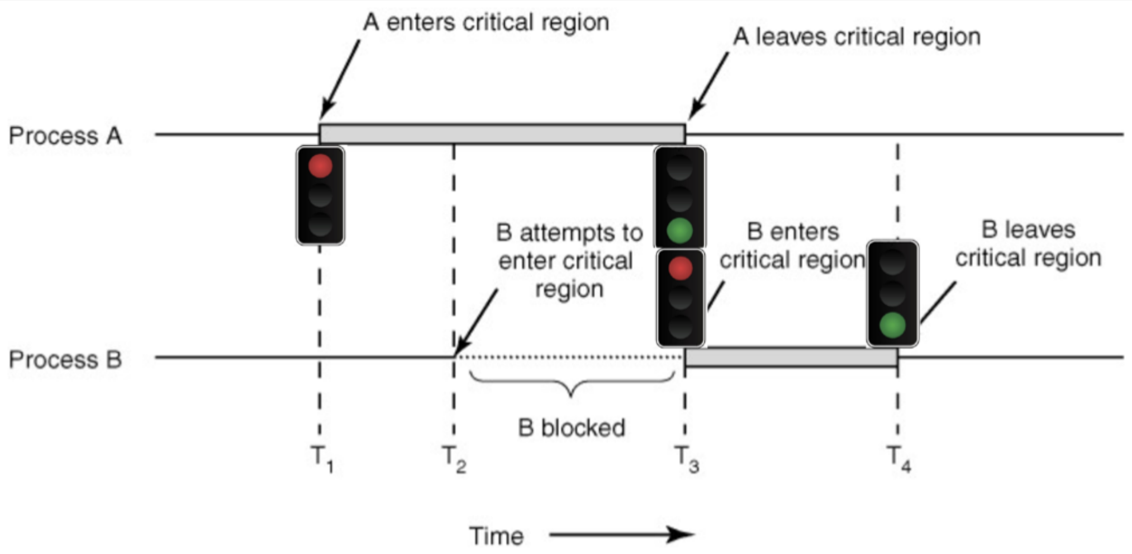


Figure 4.4: Semaphore interaction on critical section/region

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.

<table>
<tr><th>Statement</th><th>Statement Implementation</th><th>Usage</th></tr>
<tr><td>$wait(s)$</td><td><pre>
wait(s)
{
    if (S > 0)
    {
        S--;
    }
}
</pre></td><td>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 $True$.</td></tr>
<tr><td>$signal(s)$</td><td><pre>
signal(s)
{
    S++;
}
</pre></td><td>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.</td></tr>
</table>

This is a good solution as there is no possibility for a race condition as these statements will always be enforced due to the face 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.

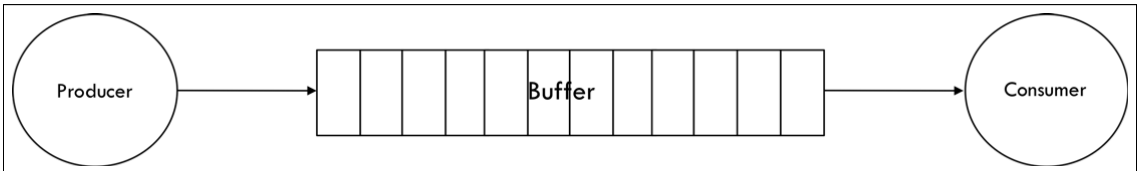


Figure 4.5: The producer-consumer problem

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.

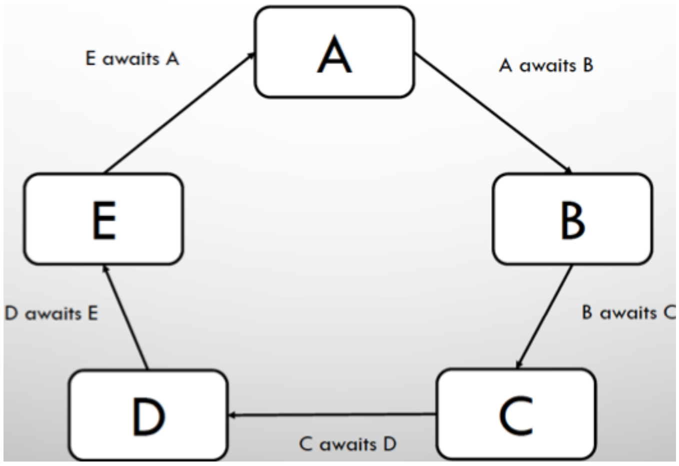


Figure 4.6: Circular wait

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
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.

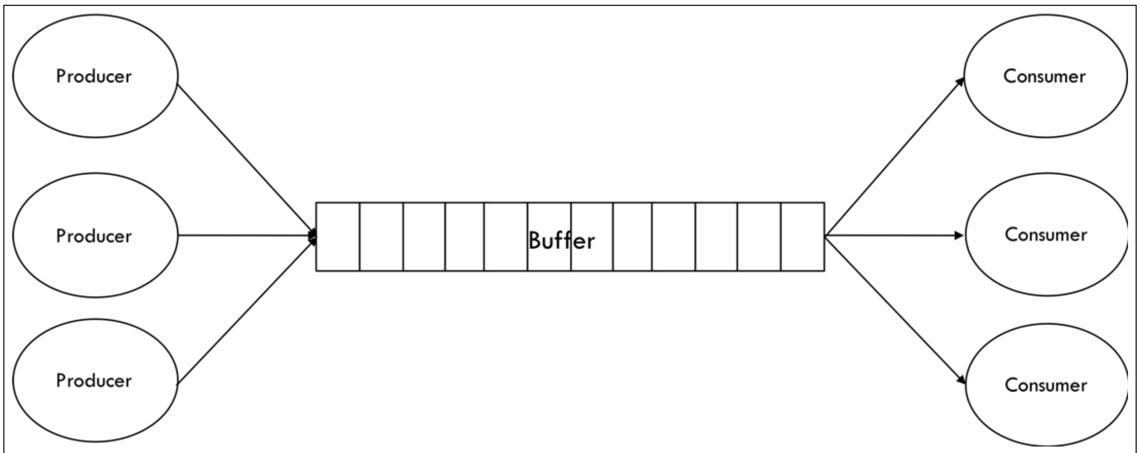


Figure 4.8: The multiple producer-consumer problem

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 discussed in the printer spooler example (page n).

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
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 mutex $BusyBuffer$ has ownership and therefore can only be incremented/decremented by the same thread/process.

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
```

This switching would cause ...

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.

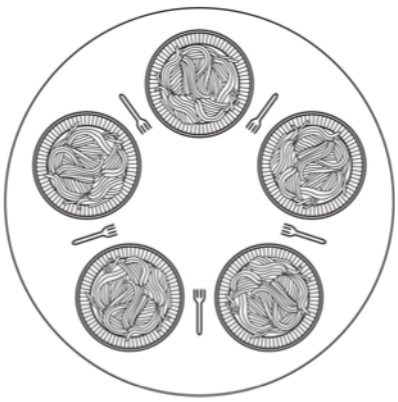


Figure 4.9: Layout of table

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.

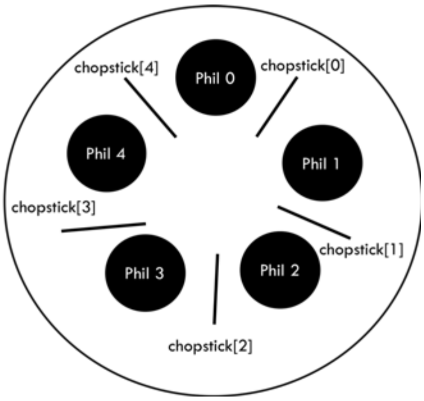


Figure 4.10: Updated layout of table

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.

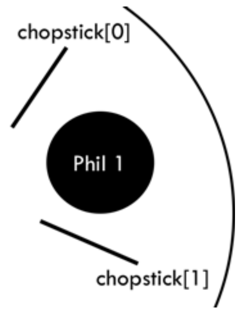


Figure 4.11: Philosopher 1

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
}
```

Producer 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
}
```

Producer 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
   }
}
```

Producer class

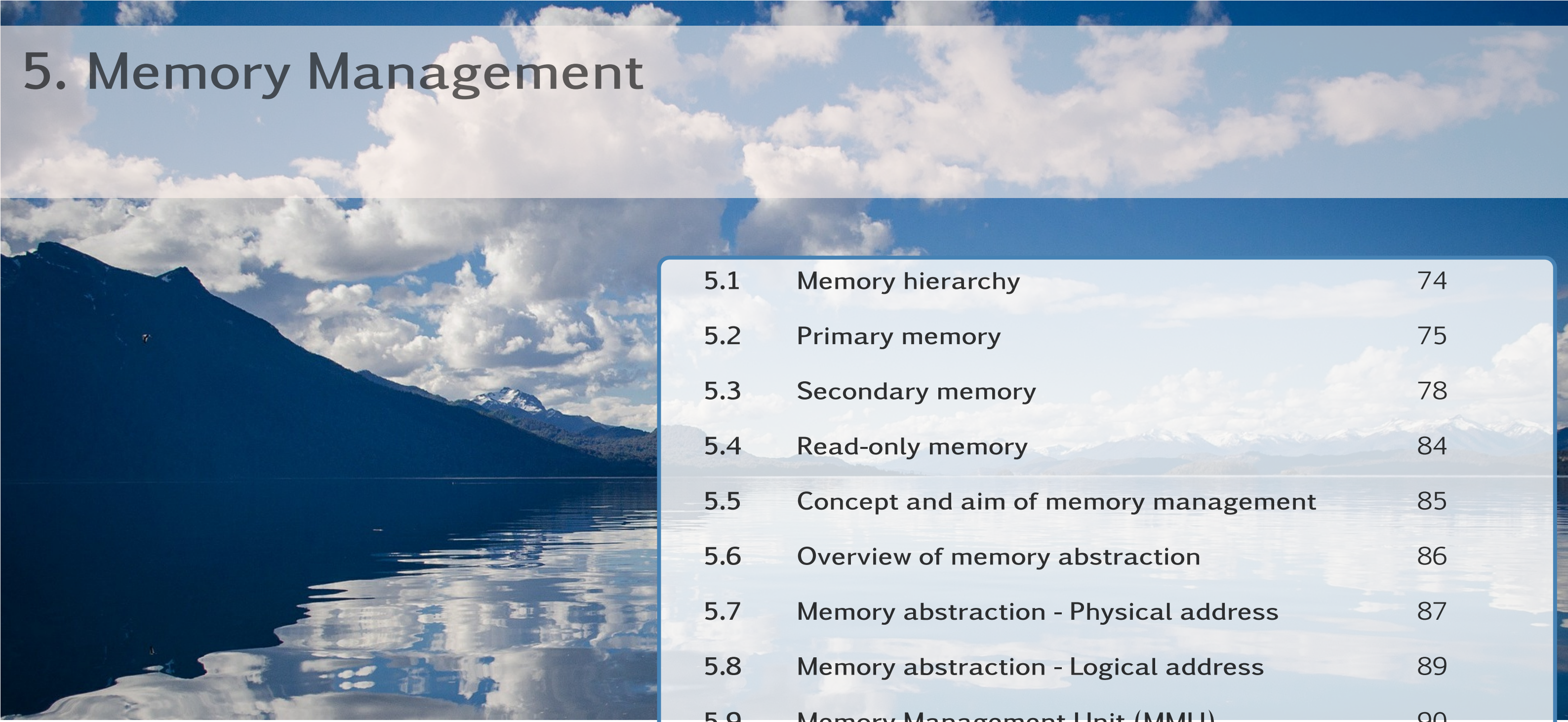
This solution uses the array state to keep track of whether a philosopher is:

- 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

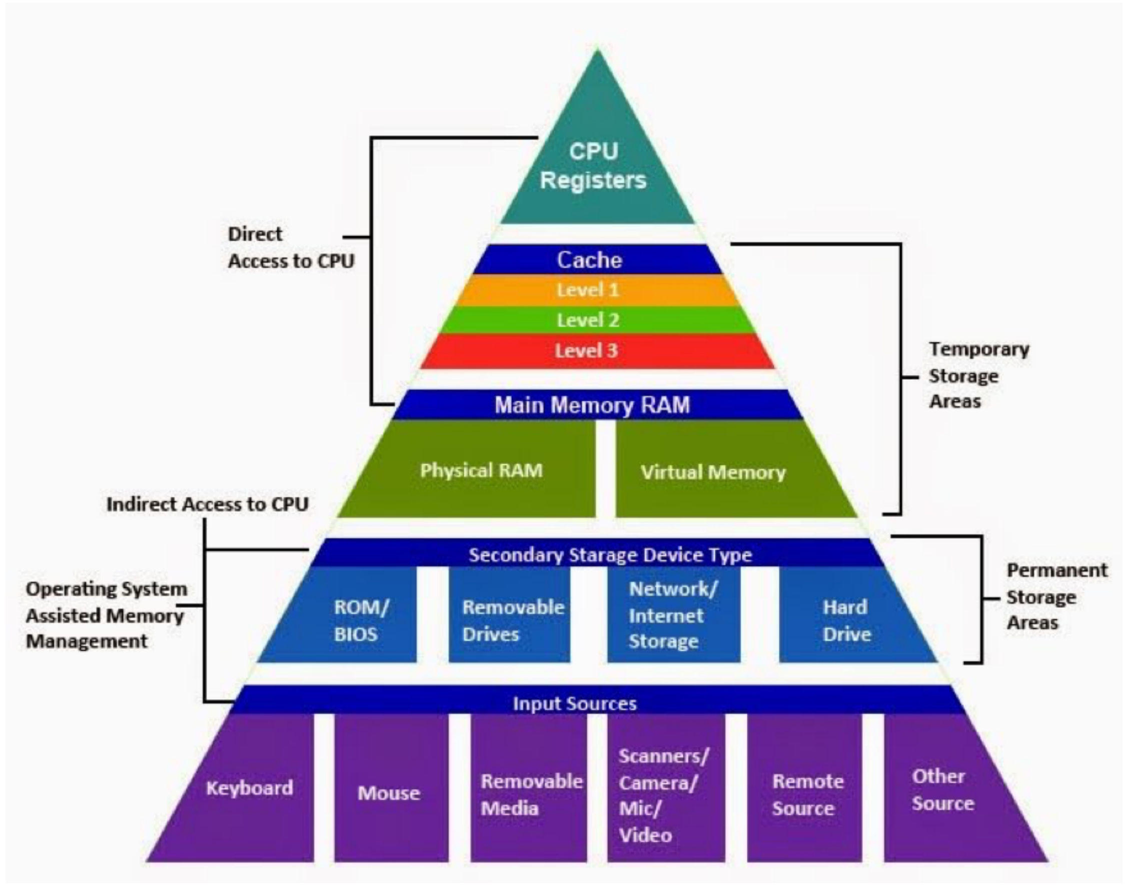


Figure 5.1: Memory hierarchy pyramid

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.

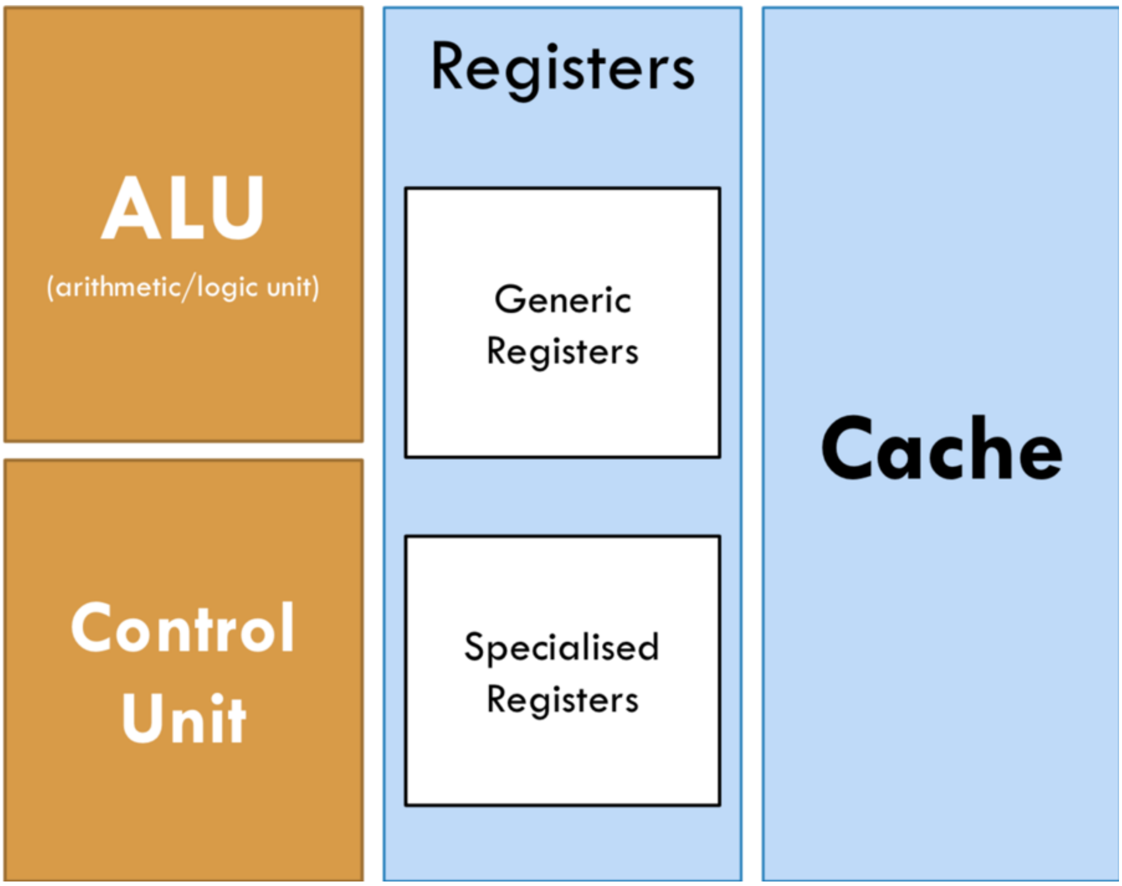


Figure 5.3: Cache memory

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.

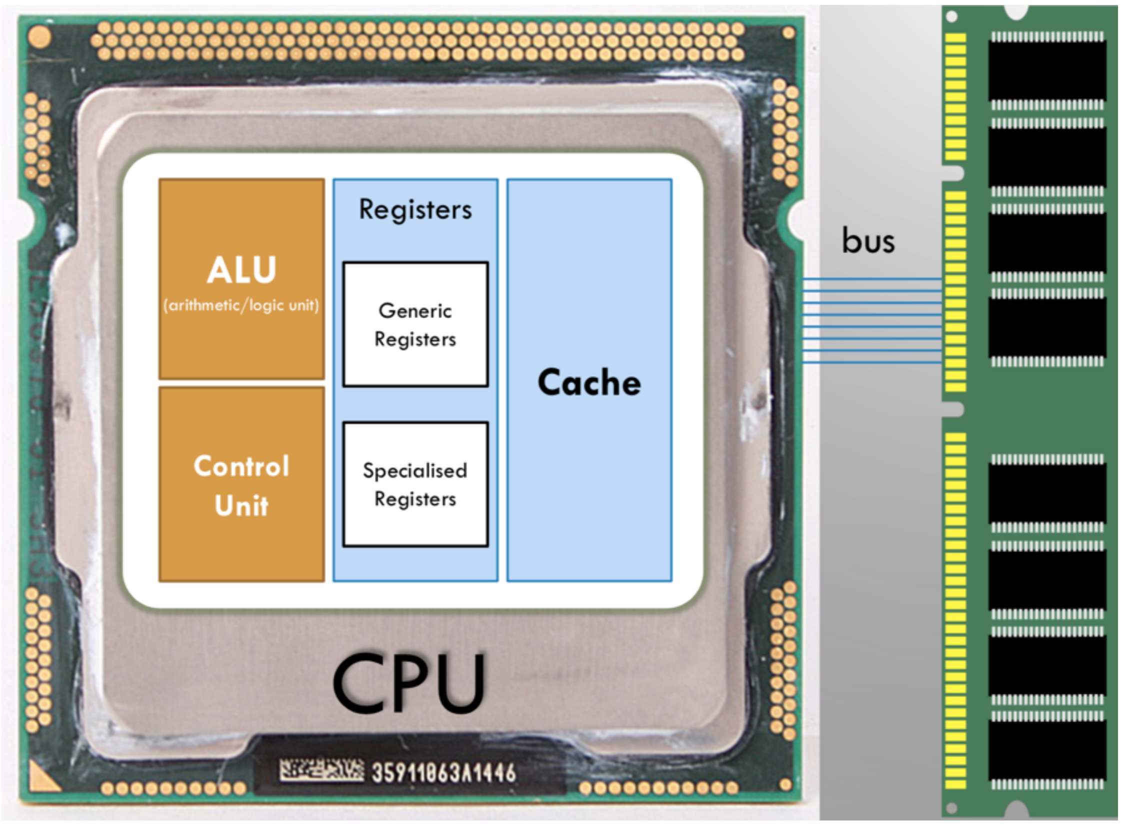


Figure 5.4: Main memory

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

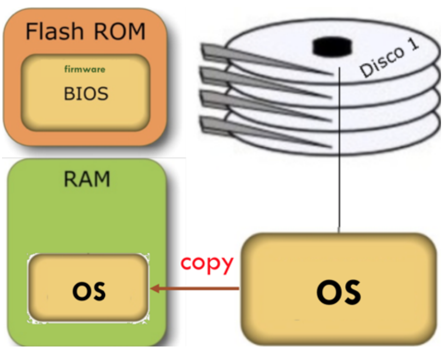
Properties

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.

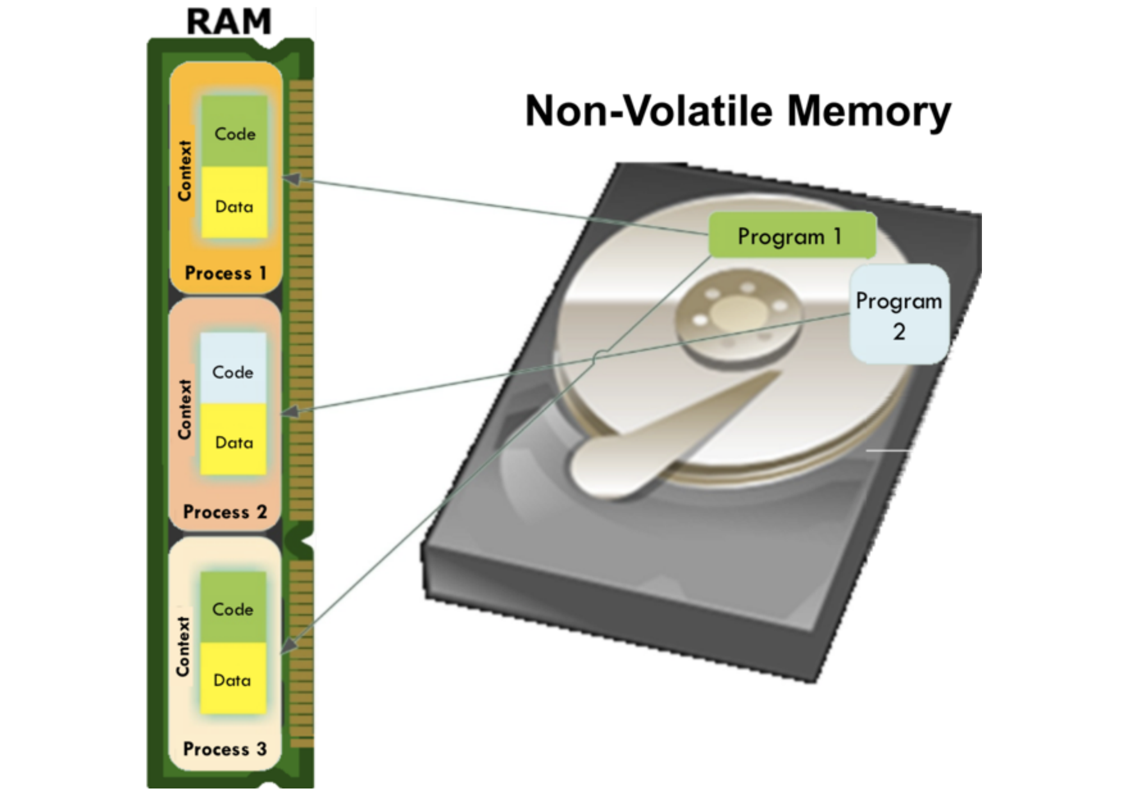


Figure 5.6: Main memory and non-volatile memory

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.

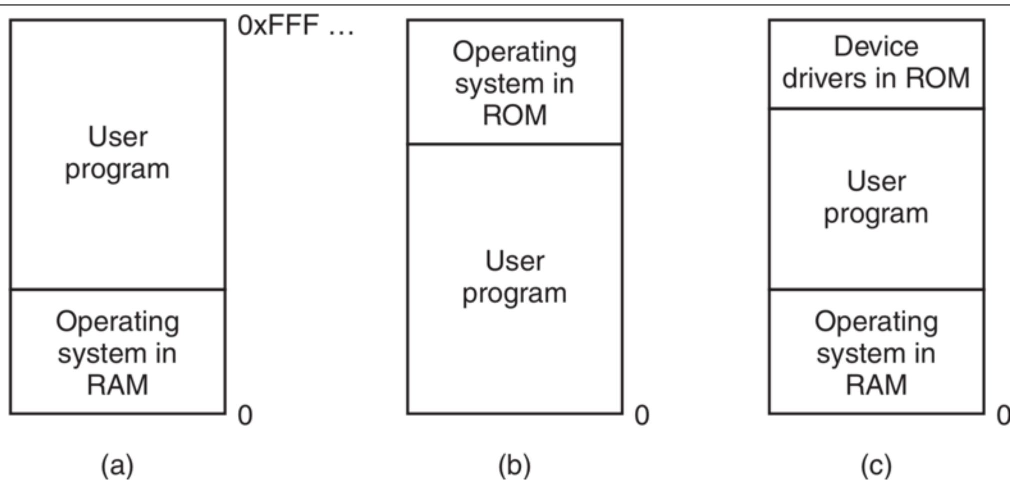


Figure 5.7: Organising memory with an operating system and one user process

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 - Physical address

Definition

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.

Logical address space is the set of all physical addresses corresponding to the logical addresses in a logical address space.

Referencing physical memory

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

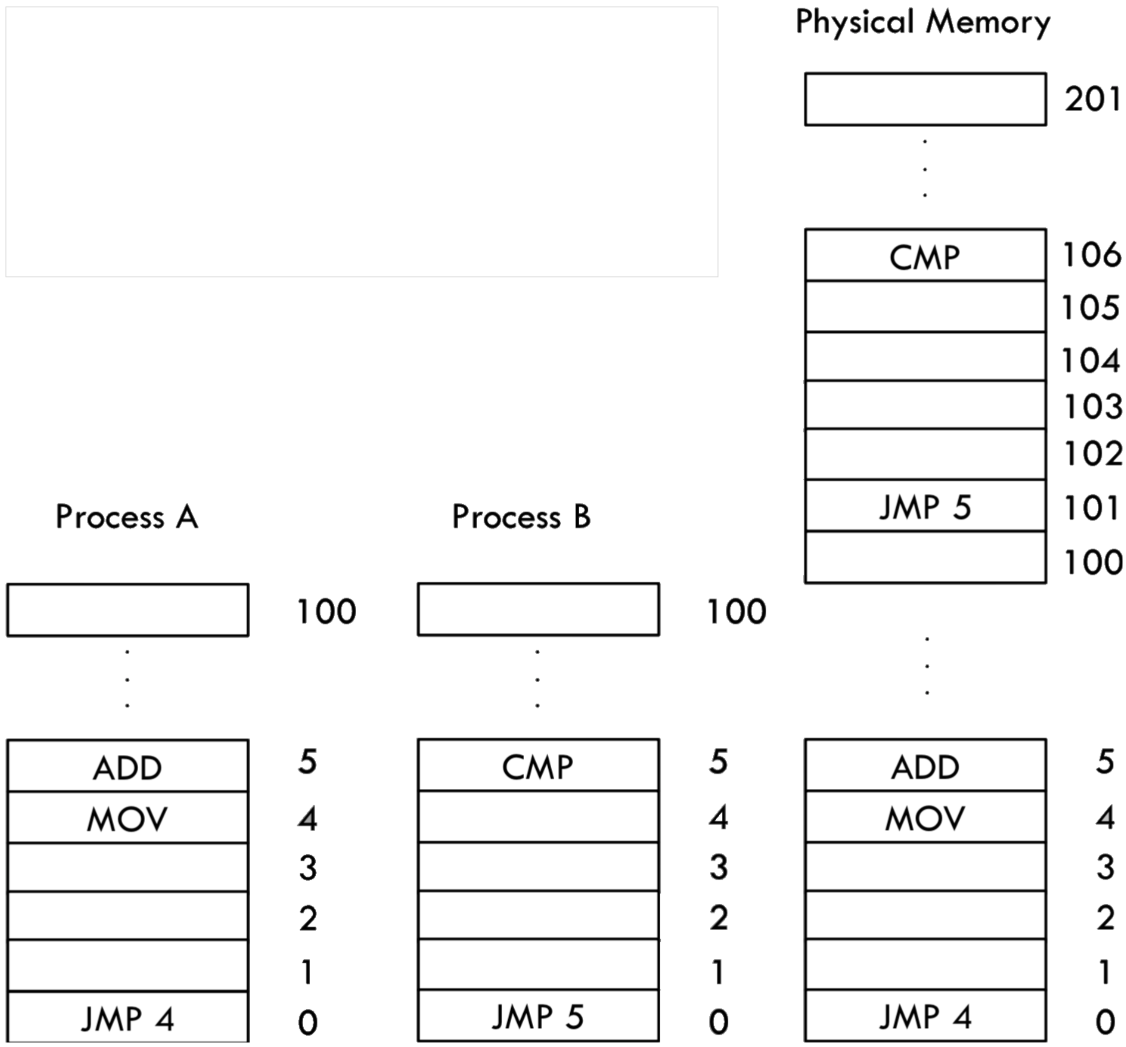


Figure 5.8: Referencing physical memory

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.

5.8 Memory abstraction - Logical address

Definition

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

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

Logical address space is the set of all logical addresses generated by a program's perspective.

Using logical address spaces

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.

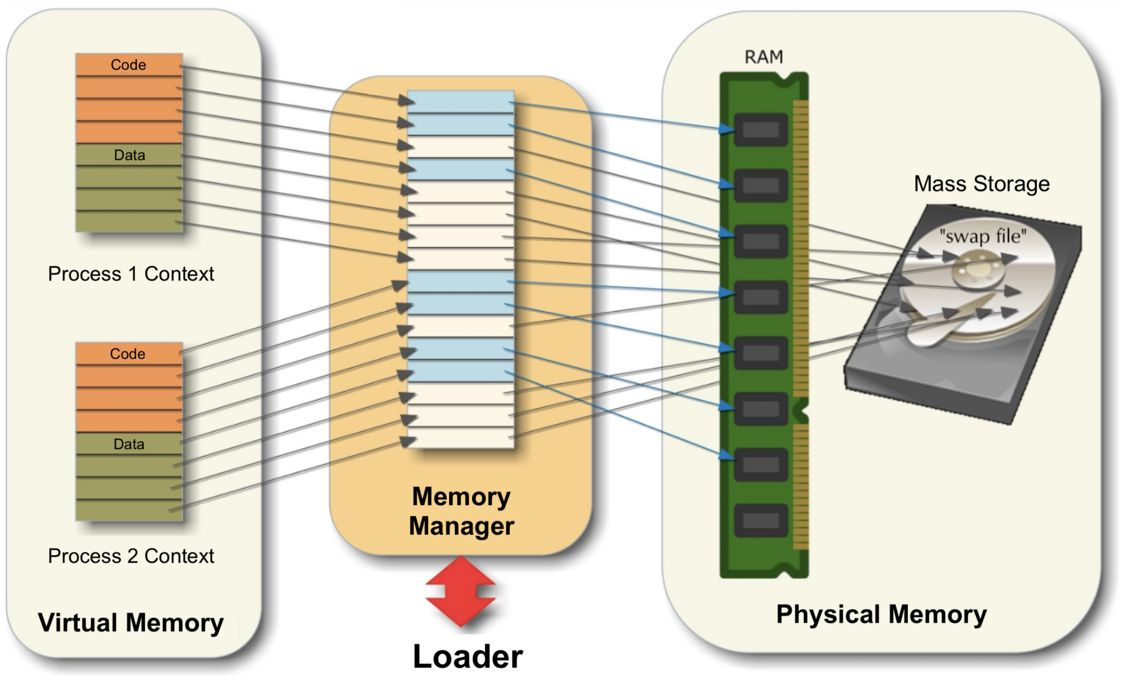


Figure 5.9: Logical address spaces

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

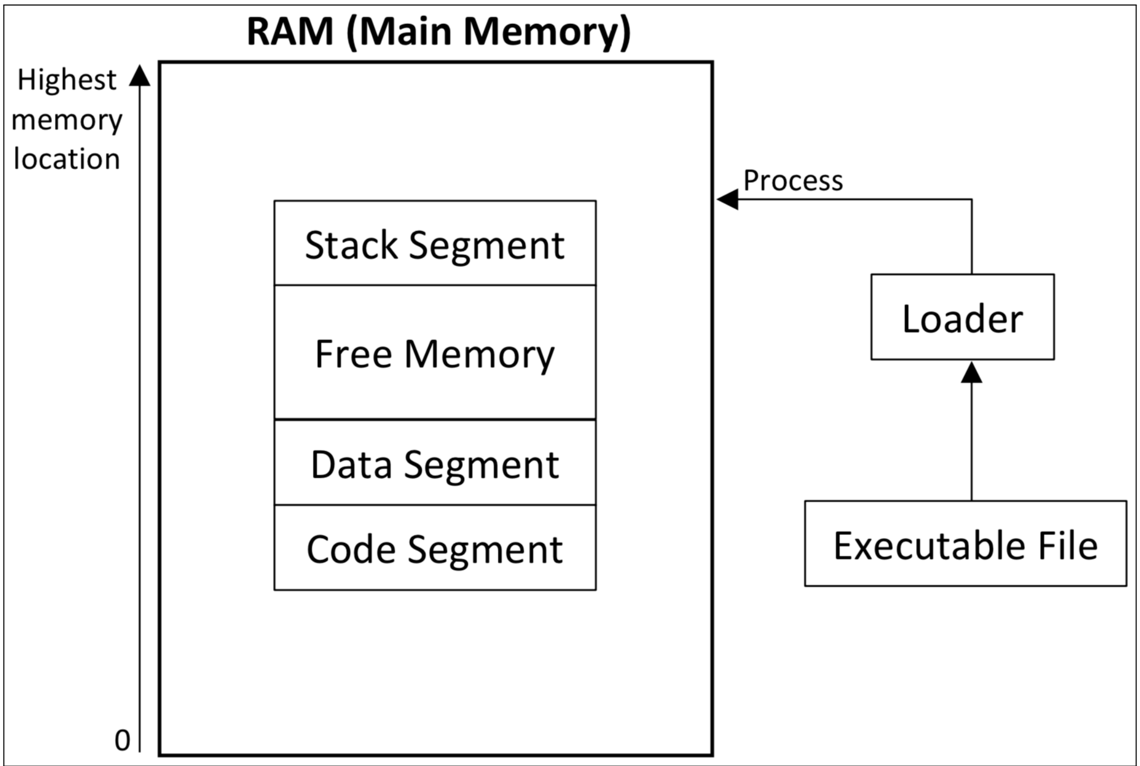


Figure 5.10: Loader

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).

5.9 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.

Base and limit registers

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

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

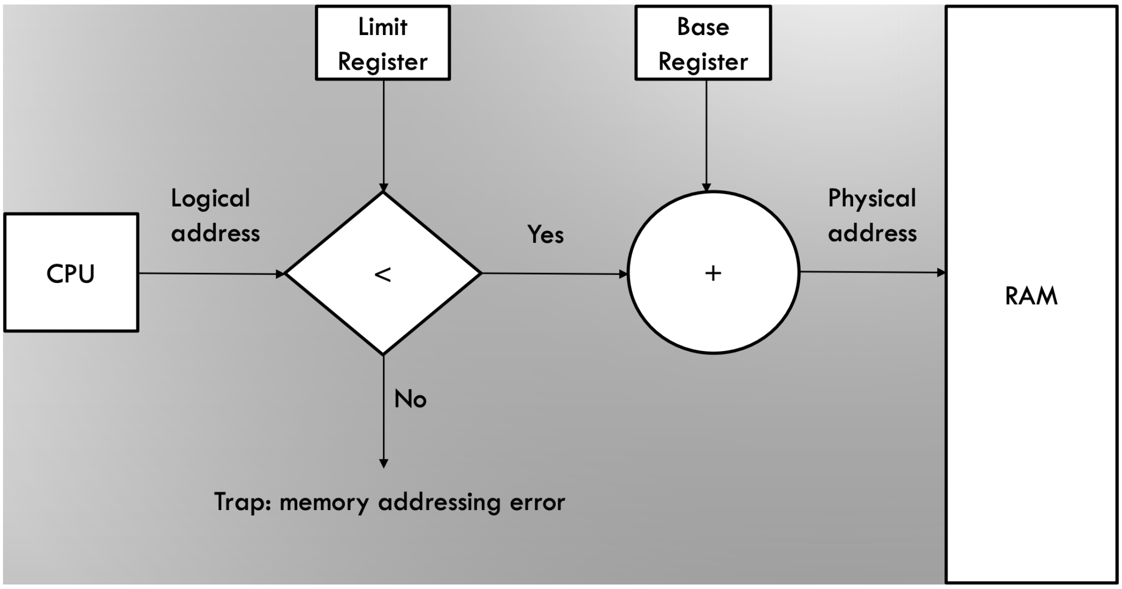


Figure 5.11: Base and limit registers between CPU and RAM

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`.

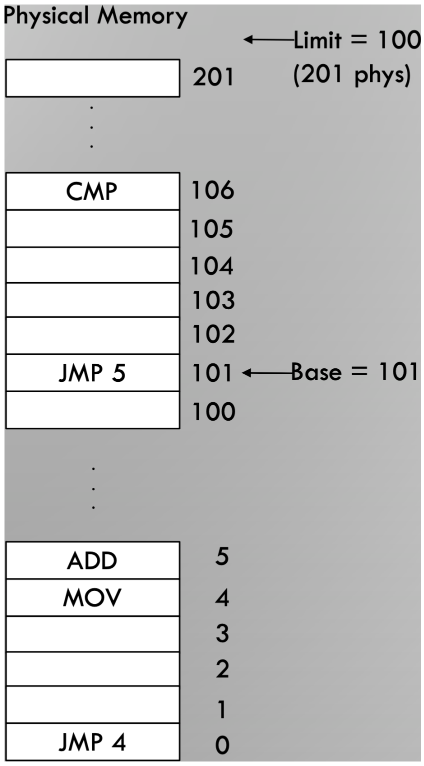


Figure 5.12: Base and limit registers between CPU and RAM

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.

5.10 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.

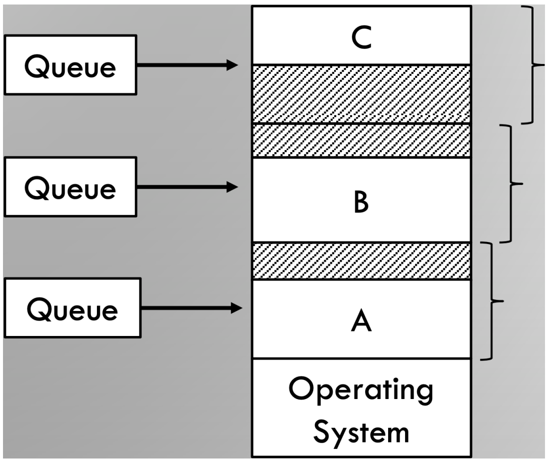


Figure 5.13: Variable partitioned memory

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.

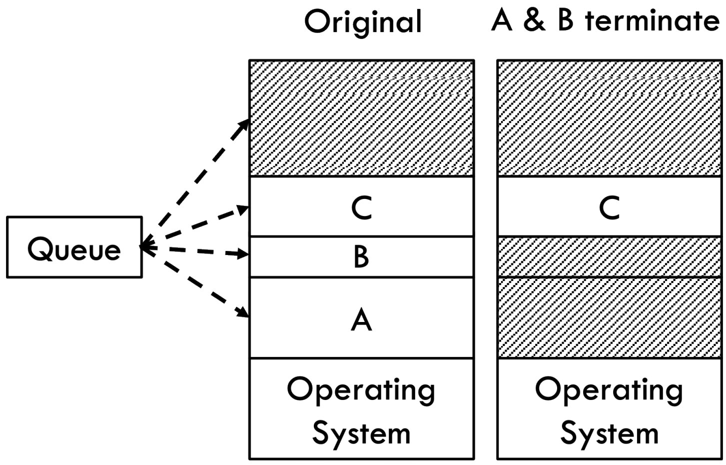


Figure 5.14: Variable partitioned memory

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.

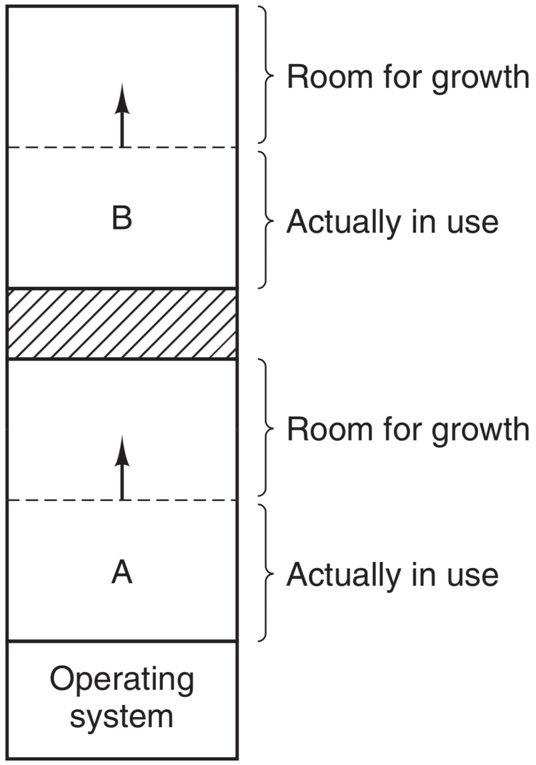


Figure 5.15: Varying process sizes

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.

5.11 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 shown in the memory hierarchy (page n). 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.

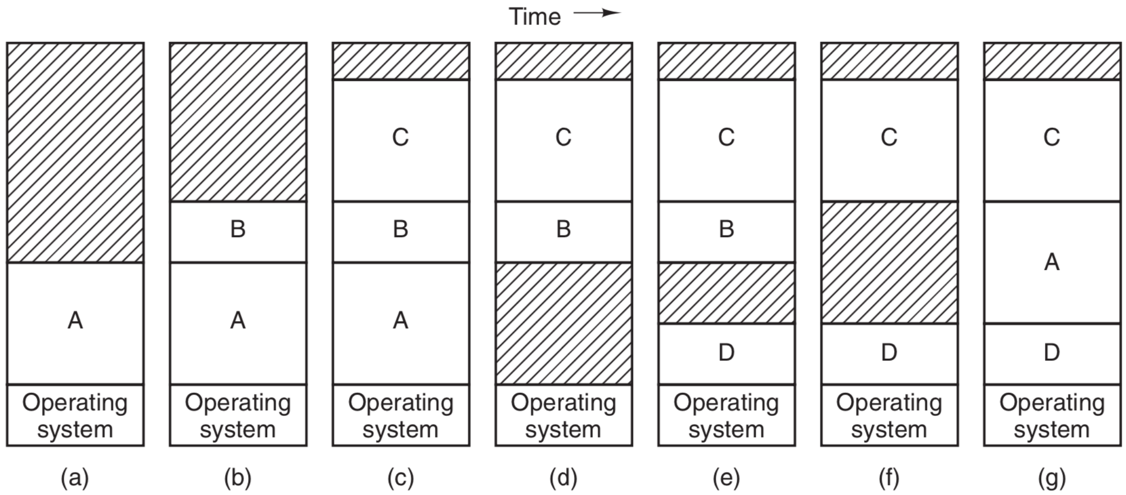


Figure 5.16: Swapping

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

5.12 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 into allocation units as small as a few words and as large as several kilobytes.

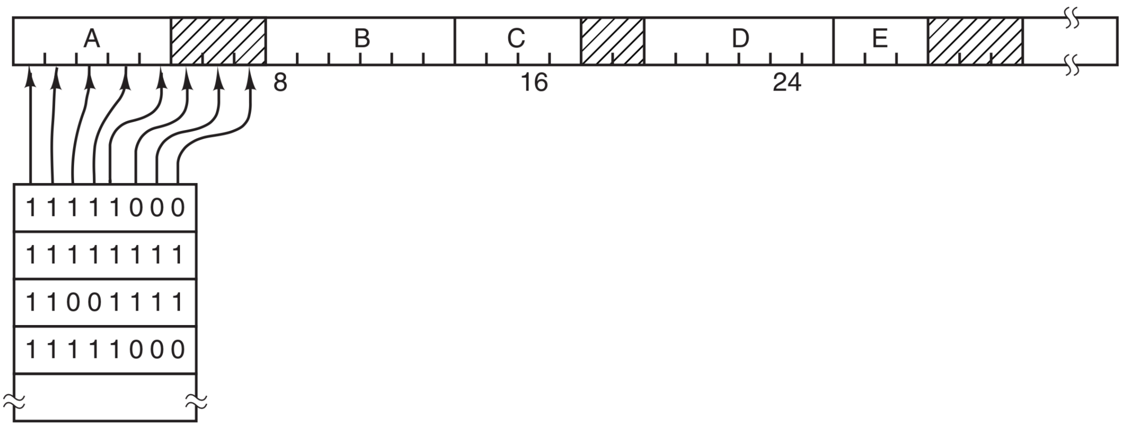


Figure 5.17: Using bitmaps to manage free space

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.

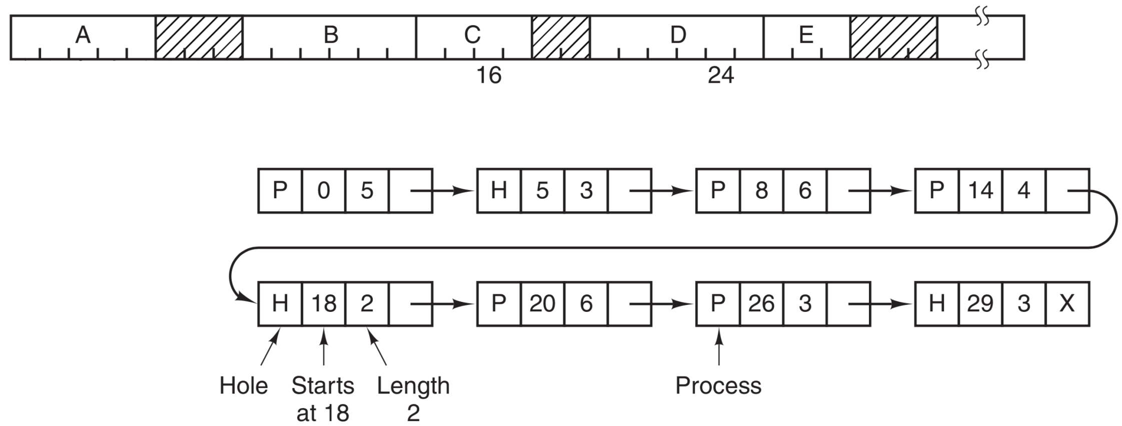


Figure 5.18: Using free lists to manage free space

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.

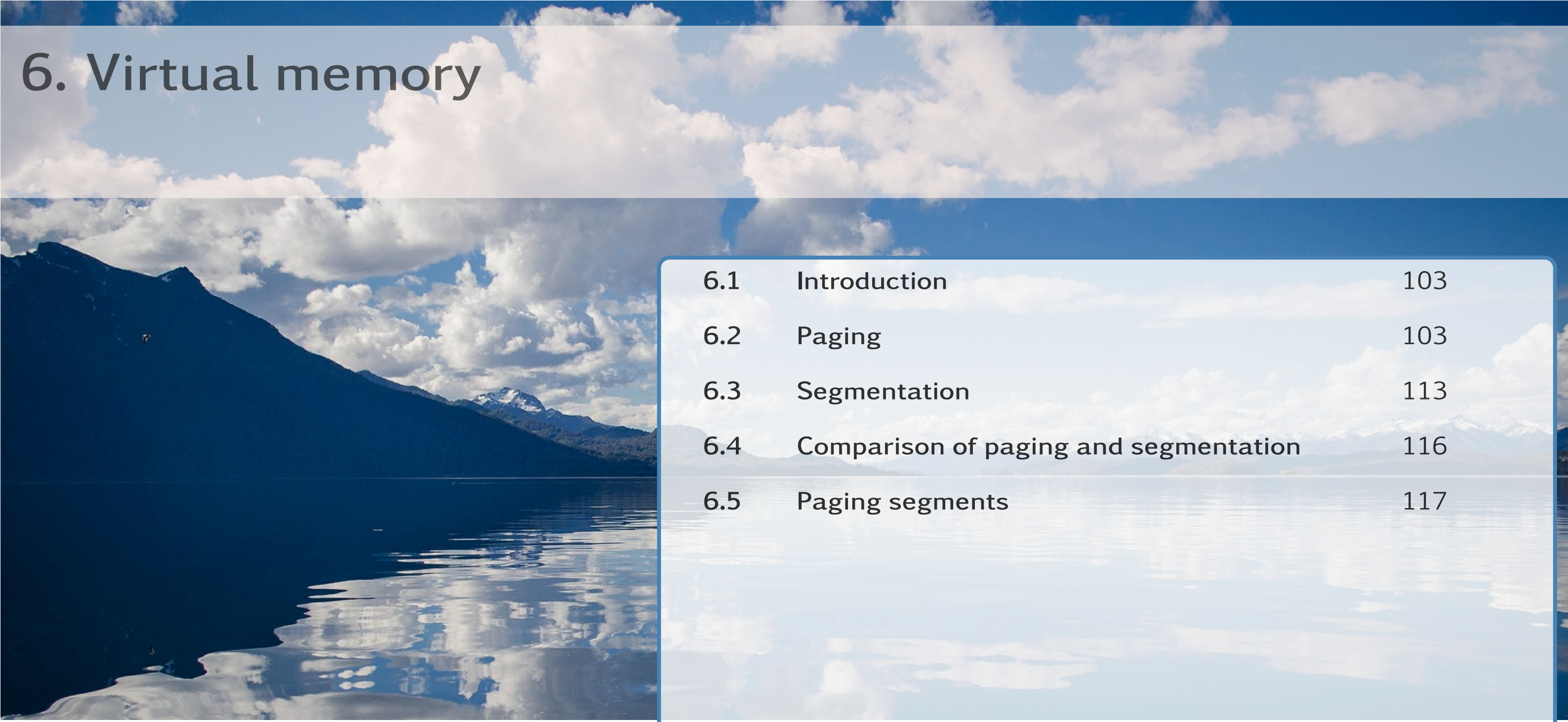
Figure 5.30: Loading a process using worst fit with size 1KB.

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.

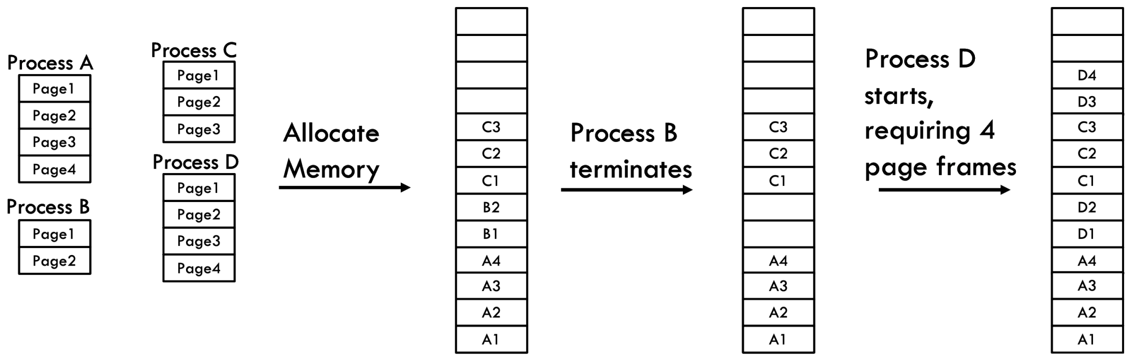


Figure 6.1: Loading process pages in to physical memory frames.

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.

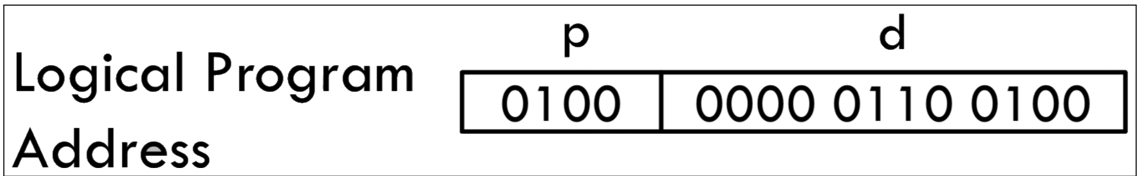


Figure 6.2: Logical program address in paging.

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

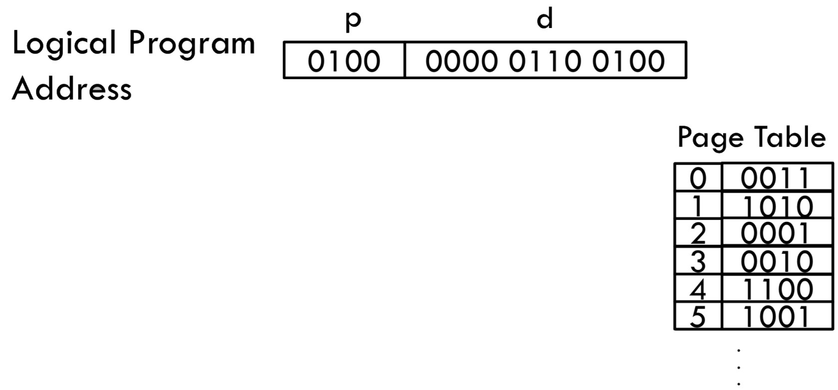


Figure 6.3: Logical program address and respective page table.

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:

$$p = 0100_2 = 4_{10}$$
$$d = 0000\ 0110\ 0100_2 = 100_{10}$$

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

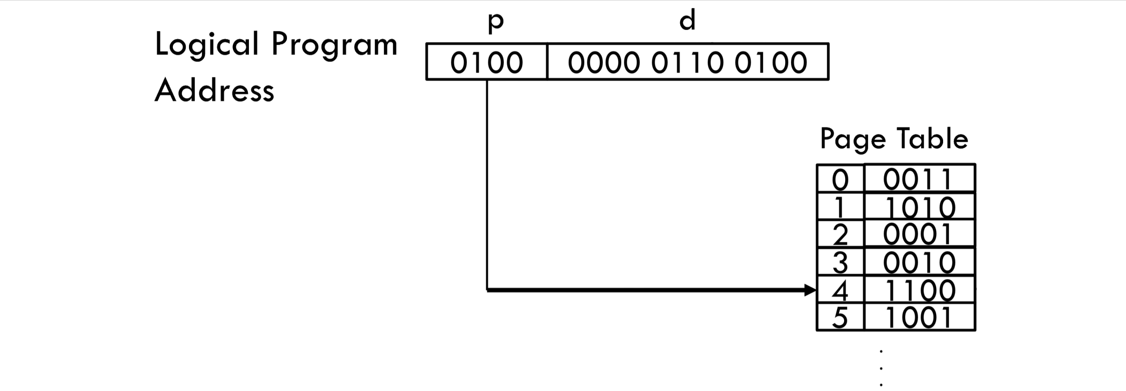


Figure 6.4: Using p as an index to the page table.

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

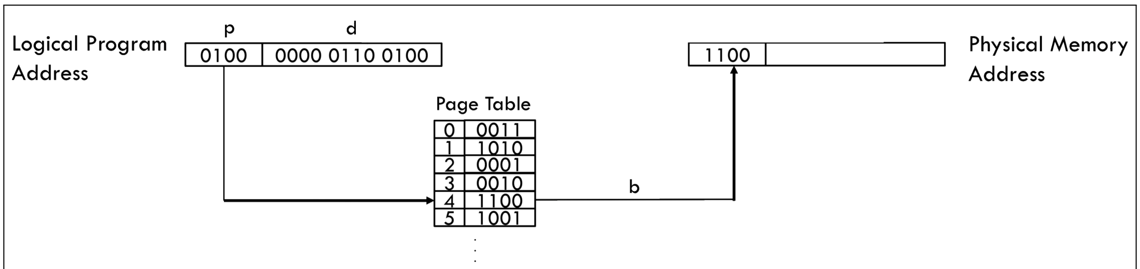


Figure 6.5: Using the value at index 4 as the base (b).

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

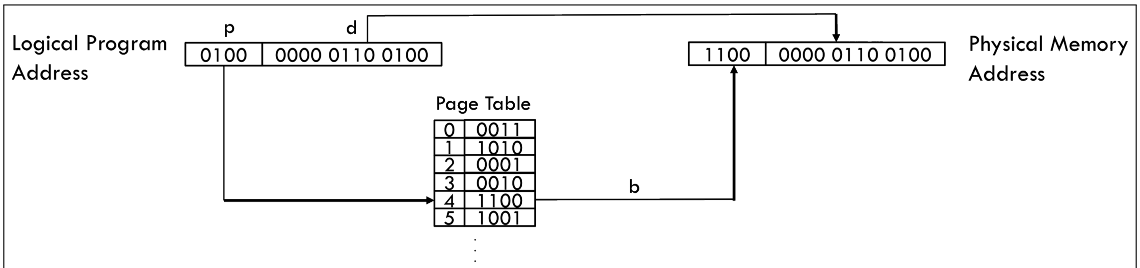


Figure 6.6: Using the displacement value (d) to complete the physical memory address.

This can also be shown by performing the following operations:

physical memory address = page table value at index p + displacement value (d)

physical memory address = 1100 0000 0000 0000 + 0000 0110 0100

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 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	1	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.

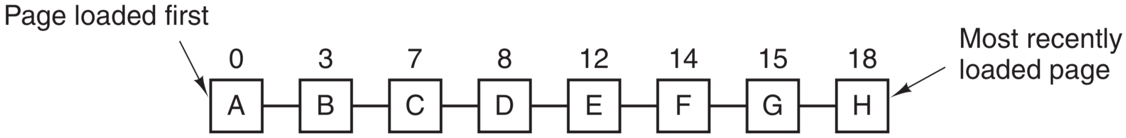


Figure 6.10: Pages shorted in FIFO order.

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.

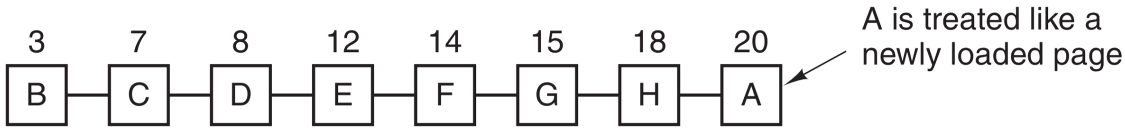


Figure 6.11: "Second chance" for page A.

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.

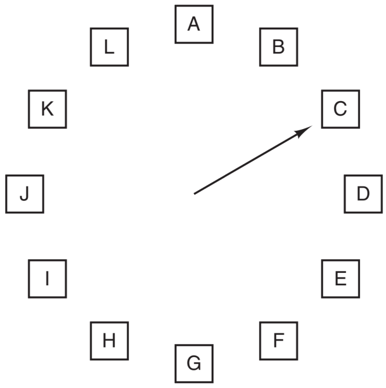


Figure 6.12: Clock page replacement algorithm.

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 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 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.

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.

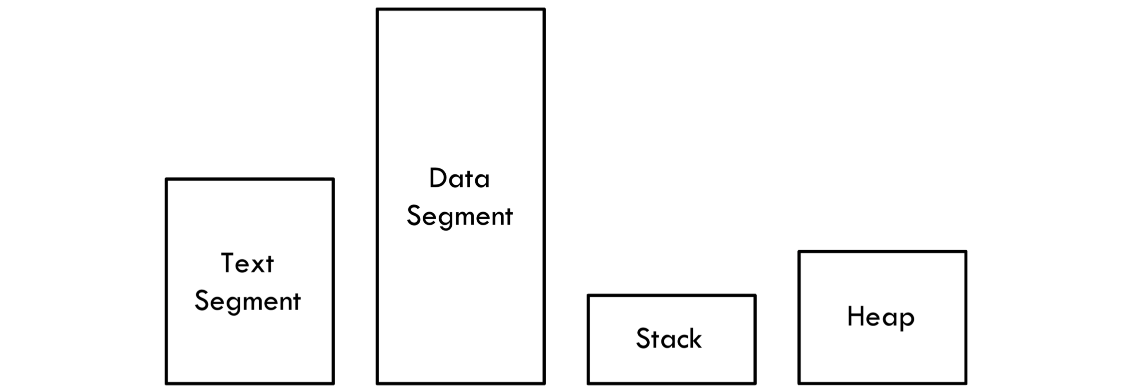


Figure 6.14: Program segments from a C compiler.

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.

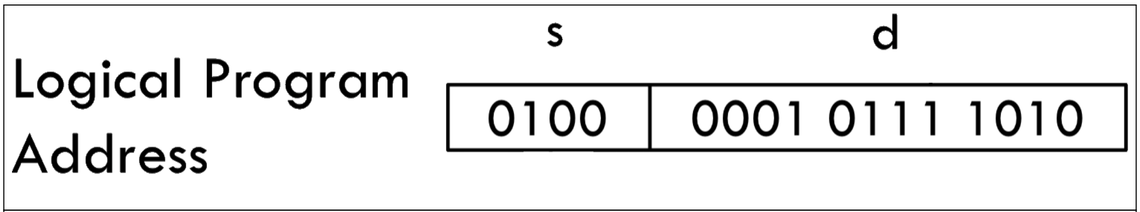


Figure 6.15: Logical program address in segmentation.

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

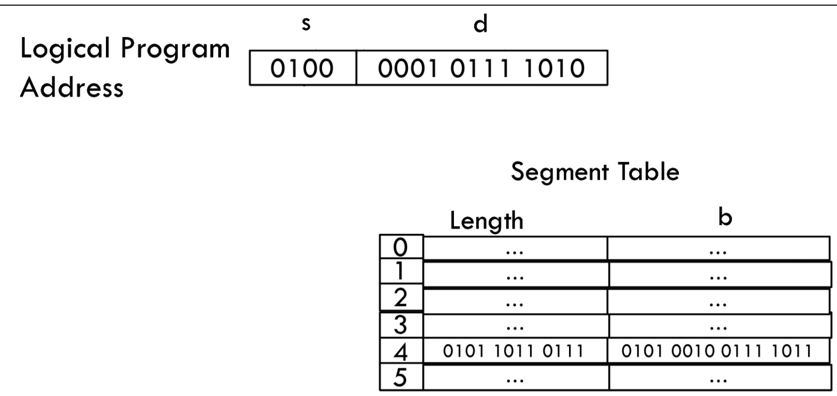


Figure 6.16: Logical program address and respective segment table.

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.

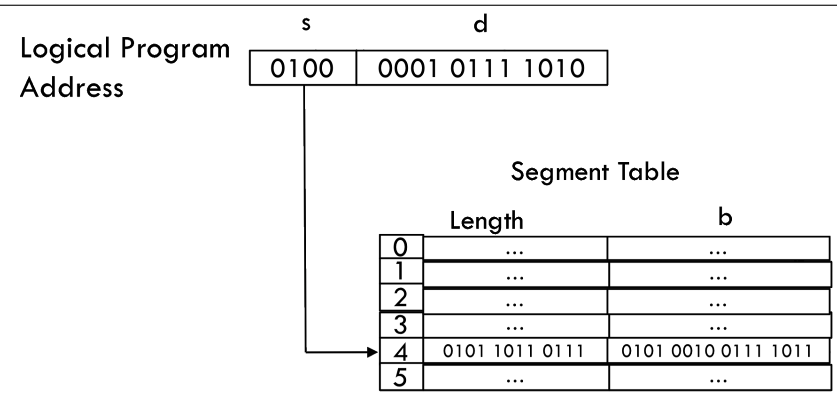


Figure 6.17: Using s as an index to the segment table.

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.

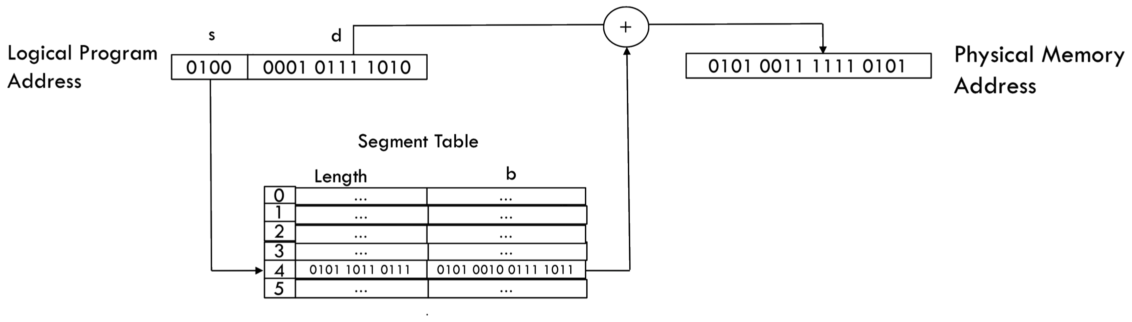


Figure 6.18: Using the value at index 4 as the base (b).

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.