CAIE Computer Science IGCSE

4 - Software

Advanced Notes

4.1 Types of software and interrupts

System and application software

Software can be classified into two types:

1. System software
2. Application software

System software

System software provides the services that the computer provides. It manages and controls the computer hardware and acts as a platform to run application software.

Examples:

- Operating Systems (OS): e.g. Windows, macOS, Linux
- Utility programs: e.g. antivirus, backup software
  Utility software are programs designed to help maintain, enhance, and troubleshoot a computer system
- Device drivers: allow OS to communicate with hardware
- Firmware: built-in software controlling hardware (e.g. BIOS)

Application software

Application software provides the services that the user requires.

Examples:

- Word processors (such as Microsoft Word)
- Web browsers (such as Chrome, Firefox)
- Games, media players, email clients

Operating Systems (OSs)

Operating systems, such as Windows, macOS and Linux, are a type of software that manage and control the computer and its resources. Without an operating system, a computer would not be usable. Operating systems have several main functions.

Managing files

File management involves controlling how data is stored, organised, and retrieved on a computer. The operating system handles all the basic tasks involved in dealing with files. This includes creating files, giving them names, saving them to storage, and placing them into folders. The OS also allows users to move, copy, delete, or rename files.

In addition to managing the organisation of files and folders, the OS keeps track of where each file is stored on the disk. It ensures that data is saved correctly and that files do not overwrite each other unless specifically instructed to do so.

Handling interrupts

An interrupt is a signal sent to the processor to indicate that an event needs immediate attention. The operating system manages these signals so that important events are dealt with quickly without disrupting the entire system. For example, when a key is pressed on a keyboard, an interrupt tells the CPU to pause its current task and process the key press. The OS uses an interrupt handler to determine what caused the interrupt and to carry out the correct response before returning to the original task.

Providing an interface

Operating systems provide a user interface that allows users to interact with the computer. There are a few different kinds of user interfaces:

- A graphical user interface (GUI) is a visual way for users to interact with electronic devices. It allows them to interact with the computer by using graphical elements (such as windows, icons, menus, and pointers).

- Command line interfaces are less visual, and the user interacts with the computer by typing in text-based commands.

Managing peripherals and drivers

Peripherals, such as printers, keyboards, and monitors, need special instructions to work with the computer's hardware. These instructions are provided by device drivers. The operating system manages these drivers so that software can use peripherals without needing to know the details of how they work. For example, when a user clicks "Print" in a document editor, the OS uses the printer's driver to send the data in the correct format. This allows different hardware to be swapped or upgraded without changing the application software and allows different manufacturers' hardware to be compatible with a wide variety of devices.

Managing memory

The operating system is responsible for keeping track of all the computer's memory. This includes deciding where programs and data are stored in RAM, ensuring that programs do not overwrite each other's memory space, and moving data between RAM and secondary storage when necessary (virtual memory). By managing memory effectively, the OS ensures that programs can run smoothly and efficiently, even when there is limited physical memory available.

Managing multitasking

Modern operating systems allow more than one program to run at the same time. The OS achieves this by rapidly switching the CPU's attention between tasks, giving the impression that they are running simultaneously. It decides which process should run next, how long it should run for, and how to share resources such as memory and input/output devices between them. This ensures that all running programs remain responsive.

Providing a platform for running applications

Application software relies on the operating system to access hardware and perform basic operations. The OS provides a consistent environment in which applications can run, regardless of the specific hardware in the computer. This means that developers can write software for the OS rather than for each individual hardware configuration. For example, an application can save a file without needing to know whether the data is going to a hard drive or SSD.

Providing system security

The operating system is responsible for protecting the system from unauthorised access and malicious activity. This includes managing user permissions, requiring passwords for accounts, and controlling which programs are allowed to run. The OS may also include security features such as encryption, firewalls, and automatic updates. These measures help keep data safe and ensure the overall stability of the system.

Managing user accounts

User management is another key function of the operating system. Many systems allow multiple users to have separate accounts, each with its own files, settings, and preferences. The OS is responsible for creating and managing these accounts. This includes setting up usernames and passwords, managing login processes, and assigning different levels of access to different users. For example, an administrator account might have permission to install software, while a standard user does not.

Running application software

Hardware, firmware, and an operating system are required to run applications software.

- **Hardware** - The physical components of the computer such as the CPU, RAM, storage, and input/output devices. It can only understand machine code and cannot directly run high-level applications.

- **Firmware** - Permanent software stored in non-volatile memory (such as ROM) that runs as soon as the computer is powered on. It provides low-level control over the hardware and starts the boot process. Part of the firmware is the bootloader, which checks that the hardware is working correctly and then loads the operating system into memory.

- **Operating system** - Software that manages hardware resources and provides a platform for running applications. It allows applications to work without needing to control hardware directly by acting like a bridge between them.

Process of running applications

1. The computer is switched on and the hardware powers up.

2. The firmware runs, performing initial checks and loading the operating system from storage into RAM.

3. The operating system starts and provides services for applications.

4. Applications are launched on top of the operating system, which handles all communication between them and the hardware.

Interrupts

As mentioned above, an interrupt is a signal sent to the processor to indicate that an event needs immediate attention. They allow the CPU to respond quickly to important events.

How an interrupt is generated

- Hardware devices, such as keyboards or mice, can send an interrupt signal to the CPU when an action occurs.
  Examples: Pressing a key on the keyboard, moving the mouse

- Software conditions, such as an error or a special instruction, can also generate an interrupt.
  Examples: division by zero errors, and two processes trying to access the same memory location at the same time.

How interrupts are handled using interrupt service routines

1. The CPU finishes its current instruction.
2. The CPU saves the current state (contents of registers and the program counter) to memory so it can resume later.
3. The CPU identifies the correct interrupt service routine and runs it to deal with the cause of the interrupt.
4. The CPU restores the saved state and continues with the original task.

What happens as a result of interrupts

- Urgent tasks are dealt with promptly without waiting for the CPU to finish all other work.
- System responsiveness improves.
- Some interrupts may change the program flow permanently, such as errors that stop execution.

4.2 Types of programming language, translators and integrated development environments (IDEs)

Low-level vs high-level languages

Programming languages are used to write instructions that computers can execute. They fall into two main categories:

- High-level languages (e.g. Python, Java, C#)
- Low-level languages (e.g. Assembly language, Machine code)

High-level languages

Designed for humans to read and write, with instructions that are similar to English (such as `print` and `while`). Most computer programs are written using high-level languages. Examples: Python, C#.

Advantages	Disadvantages
Easy for humans to understand and debug as the instructions are closer to English.	Slower to execute than low-level languages.
Programs written are machine independent, since they can be translated into machine code for each specific type of processor.	Must be translated into machine code, and this translated machine code can be less efficient than if it was originally written as machine code.

Low-level languages

Closer to machine code (binary). Examples: Assembly language, Machine code

Advantages	Disadvantages
Faster and more efficient to execute (and translate in the case of assembly language).	Hard to read and write.
Gives more control over hardware, as it is directly manipulated.	Not portable - specific to one type of processor.

Assembly language

Assembly language is a form of low-level language that uses mnemonics, which are single executable machine code instructions. Each mnemonic instruction is a single machine code instruction giving it a 1:1 relationship with machine code.

Assemblers

An assembler translates assembly language into machine code. Because each assembly language instruction has a 1:1 relationship to a machine code instruction, translation between the two languages is fairly quick and straightforward. Assembly language is often used to develop software for embedded systems and for controlling specific hardware components. Assemblers are platform specific, meaning that a different assembler must exist for each different type of processor instruction set.

Compilers and interpreters

Compliers

A compiler can be used to translate programs written in high-level languages, like C# and Python, into machine code. Compilers take a high-level program as their source code, check it for any errors and then translate the entire program at once. If the source code contains an error, it will not be translated. Because compilers produce non-portable machine code, they are said to be platform specific.

Once translated, a compiled program can be run without the requirement for any other software to be present. This is not the case with interpreters.

Interpreters

An interpreter translates high-level languages into machine code and executes it line-by-line. Interpreters do not generate machine code directly - they call appropriate machine code subroutines within their own code to carry out statements.

Rather than checking for errors before translation begins (as a compiler does), interpreters check for errors as they go. This means that a program with errors in can be partially translated by an interpreter until the error is reached.

When a program is translated by an interpreter, both the program source code and the interpreter itself must be present. This results in poor protection of the source code compared to compilers which make the original code difficult to extract.

Comparison of compilers and interpreters

Compiler	Interpreter
Checks source code for errors line-by-line before translation. If any errors are found they are provided in a single report.	Translation begins immediately.
Entire source code translated at once, before executing them.	Each line is checked for errors and then translated and executed sequentially.
No need for source code or compiler to be present when the translated code is executed.	Both the source code and the interpreter must be present when the program is executed.
Protects the source code from extraction.	Offers little protection of source code.

Integrated Development Environments (IDEs)

An Integrated Development Environment (IDE) is software that brings together tools needed to write, test, and debug programs in one place, making development faster and easier.

Common functions of an IDE

- **Code editors** - Provide a space to write and edit program code, often with features like syntax highlighting and line numbering to make code easier to read.
- **Run-time environment** - Allows the program to be run and tested within the IDE so developers can see how it behaves without leaving the environment.
- **Translators** - Convert the source code into machine code using a compiler or interpreter so it can be executed by the computer.
- **Error diagnostics** - Identify and display errors in the code, often giving details about the problem and where it occurred to help with debugging.
- **Auto-completion** - Suggests or completes code elements such as variable names or function calls while typing, reducing mistakes and saving time.
- **Auto-correction** - Automatically fixes certain minor coding errors, such as missing brackets or typos in keywords.
- **Prettyprint** - Formats the code neatly with consistent indentation and spacing, making it easier to read and maintain.