Software

System and application software, what an operating system does, how interrupts are handled, high-level and low-level languages, compilers, interpreters and IDEs.

4.1 Types of software and interrupts

Types of Software

System software provides the necessary services and infrastructure that the computer requires to operate effectively. It acts as a bridge between the hardware and the user applications, ensuring that all components of the computer system work together smoothly.

Examples:

- Operating systems, such as Linux Ubuntu
- Utility software, such as a disk defragmenter, antivirus software, or file compression software

Application software is designed to help users perform specific tasks or activities. Unlike system software, which manages and operates the computer itself, application software directly serves the user's needs and preferences.

Examples:

- Productivity software, such as Microsoft Word
- Multimedia software, such as Adobe Photoshop
- Web browsers, such as Firefox
- Games, such as Call of Duty

Note Academy

Operating System

The operating system (OS) is the most crucial software component in a computer system. It acts as an intermediary between the user and the computer hardware, ensuring that the system runs efficiently and effectively. Written below are some of the basic functions of an operating system.

- **File management:** The OS organizes and keeps track of files and directories on storage devices.
- **Interrupt management:** Handling interrupts, which are signals sent by hardware or software to indicate an event that needs immediate attention, is one of the functions of an OS.
- **User interface:** The OS provides a user interface that allows users to interact with the computer. This can be a graphical user interface (GUI) with windows, icons, and menus, or a command-line interface

(CLI) where users type commands.

- **Memory management:** An OS manages the computer's memory, ensuring that each application has enough memory to run efficiently without interfering with other applications.
- **Multitasking:** Operating systems allow multiple programs to run simultaneously by managing the CPU's time and resources. They schedule tasks, giving the illusion that all tasks are running at the same time by quickly switching between them.
- **Application platform:** The OS provides a platform for running application software by offering common services such as file handling, input/output operations, and graphical rendering.
- **System security:** An OS provides various security measures to protect the system from unauthorized access and potential threats.
- **User account management:** The operating system allows for the creation and management of multiple user accounts, each with its own settings and permissions. It ensures that users can personalize their computing environment while maintaining system security by isolating user data and applications.

Hardware, Firmware and OS

- The **hardware** provides the physical foundation for the computer system and performs the actual computational and storage tasks.
- The **firmware**, which includes the bootloader, initializes hardware and loads the OS and provides low-level control and communication with hardware components.
- The **operating system** manages hardware resources and provides a stable environment for applications and offers essential services such as file management, task scheduling, and user interfaces.

Interrupt Handling

An **interrupt** is a signal sent to the CPU by hardware or software indicating that an event needs immediate attention. This signal causes the CPU to temporarily halt its current operations and execute a specific routine to deal with the event.

How an interrupt is generated

Hardware interrupts:

- When a key is pressed, the keyboard sends an interrupt signal to the CPU.
- Mouse movements generate interrupts to update the cursor position on the screen.
- When a printer finishes a print job or is ready to receive data for a new print job, it generates a hardware interrupt to notify the CPU.

Software interrupts:

Note Academy

- A software interrupt is triggered if a program attempts to divide a number by zero.
- Two processes accessing the same memory location can cause a conflict, generating an interrupt for the CPU to resolve the issue.
- If a program tries to access a memory location that it is not allowed to, it triggers a software interrupt to handle the error.

How an interrupt is handled using an Interrupt Service Routine (ISR)

1. When an interrupt occurs, the CPU receives an **interrupt signal** and temporarily pauses its current tasks.
2. The **CPU saves the current state** of its registers and the program counter (which points to the current instruction being executed). This is necessary to resume the interrupted task later.
3. The CPU then jumps to a specific memory location where the **ISR (Interrupt Service Routine)** is stored. The ISR is a special function designed to handle the interrupt. It processes the interrupt, performing tasks such as reading input from a keyboard or handling a division by zero error.
4. After the ISR completes its task, the **CPU restores the saved state** of its registers and program counter.
5. The **CPU resumes executing** the original program from where it was interrupted.

What happens as a result of interrupts

- **Interrupts can be prioritized**, meaning some interrupts are more critical than others. The CPU handles higher-priority interrupts before lower-priority ones.
- Interrupts **allow the system to respond quickly** to important events, such as user inputs or hardware malfunctions, without waiting for the current task to complete.
- By handling events as they occur, **interrupts help in efficiently utilizing CPU resources**, ensuring that critical tasks are not delayed.

4.2 Types of programming language, translators and integrated development environments

High-level and Low-level Languages

High-level languages (HLLs) are programming languages that are more abstract and closer to human languages. They are designed to be easy to read, write, and maintain. Examples include Python, Java, and C++.

Advantages

- HLLs use syntax and structures that are **similar to natural languages**, making them easier to understand and write. A simple loop in Python might look like this:

Note Academy

```python
for i in range(10):
    print(i)
```

- Because the code is easier to read, **errors are easier to find and fix**. Most high-level languages also come with robust debugging tools.
- Programs written in HLLs are generally **machine-independent**. This means the same code can run on different types of computer hardware with minimal modifications.

Disadvantages

- High-level languages **do not allow direct manipulation of hardware or memory**.
- Programs written in HLLs **may run slower** than those written in low-level languages because of the extra layer of abstraction.

Low-level languages are closer to machine code and provide little or no abstraction from a computer's instruction set architecture. Examples include Assembly language and machine code.

Advantages

- Low-level languages allow programmers to write instructions that **directly control the hardware**, providing high performance and efficiency.
- Programs written in low-level languages can be **highly optimized for speed and memory usage**, as they operate directly on the hardware with minimal overhead.

Disadvantages

- Low-level languages are **much harder to read and write** compared to high-level languages. They use mnemonic codes and require detailed knowledge of the computer's architecture. The same loop in Assembly language would be much more complex:

```asm
MOV CX, 10
MOV AX, 0
START:
MOV DX, AX
; code to print DX
INC AX
LOOP START
```

- **Debugging low-level code is challenging** due to its complexity and the detailed understanding required of how the hardware executes the instructions.

Note Academy

- Low-level languages are often specific to a particular type of computer hardware, making the code **non-portable**. Programs must be rewritten or extensively modified to run on different hardware.

Assembly Language

Assembly language is a type of low-level programming language that provides a way to write instructions in a form that is more understandable to humans than machine code. It uses **mnemonics**, which are short, human-readable codes that represent machine-level instructions.

Examples of mnemonics:

- MOV (move data from one place to another)
- ADD (add two values)
- SUB (subtract one value from another)

These mnemonics correspond directly to the binary instructions (machine code) that the computer's processor can execute.

An **assembler** is a special program that translates assembly language code into machine code. The machine code is the binary instructions that the computer's processor can execute directly.

Process of assembly

1. A programmer writes the assembly language code using mnemonics and labels.
2. The assembler reads the assembly language code and translates it into machine code.
3. The assembler produces an output file containing the machine code, which can be executed by the computer's processor.

Translators: Compiler and Interpreter

High-level programming languages need to be translated into machine code that a computer can execute. This translation can be done using either a compiler or an interpreter. Below is an explanation of how a **compiler** works.

1. The compiler checks all code before executing any code.
2. It produces an error report with all errors found for the whole code (before translating/running any of the code).
3. It optimizes the source code.
4. It produces an executable file.

Advantages of a compiler

- The resulting executable file is optimized and runs faster since it's already in machine code.

Note Academy

- Once compiled, the executable can be distributed and run on any compatible system without the need for the source code or compiler.

Disadvantages of a compiler

- The compilation process can be time-consuming, especially for large programs.
- Errors are reported after the entire code is analyzed, which can make debugging more complex since you may need to fix multiple errors before recompiling.

An **interpreter**:

1. Is a translator.
2. Translates high-level language to low-level language.
3. Executes one line at a time.
4. Runs error diagnostics.
5. Produces error messages to tell the user the location of the error.
6. Stops execution when it encounters errors.
7. Continues translating when an error is fixed.

Advantages of an interpreter

- Programs can be run and tested quickly without the need for a separate compilation step.
- Errors are reported as soon as they are encountered, making it easier to identify and fix issues on the fly.

Disadvantages of an interpreter

- Interpreted programs generally run slower than compiled programs because the translation happens simultaneously with execution.
- The source code and the interpreter must be present on the system where the program is to be executed.

Integrated Development Environment (IDE)

An IDE is a software application that provides comprehensive facilities to computer programmers for software development. Listed below are some of the features of an IDE.

- **Code editors** for writing and editing code with features like syntax highlighting and code folding.
- **Run-time environments** for running and debugging code directly within the IDE.
- **Translators** like compilers and interpreters for converting code into executable form.
- **Error diagnostics** for identifying and fixing errors with features like syntax checking and detailed error messages.

Note Academy

- **Auto-completion** to assist in writing code quickly and accurately.
- **Auto-correction** to automatically fix common mistakes.
- **Prettyprint** for maintaining a clean and readable code format.

Note Academy