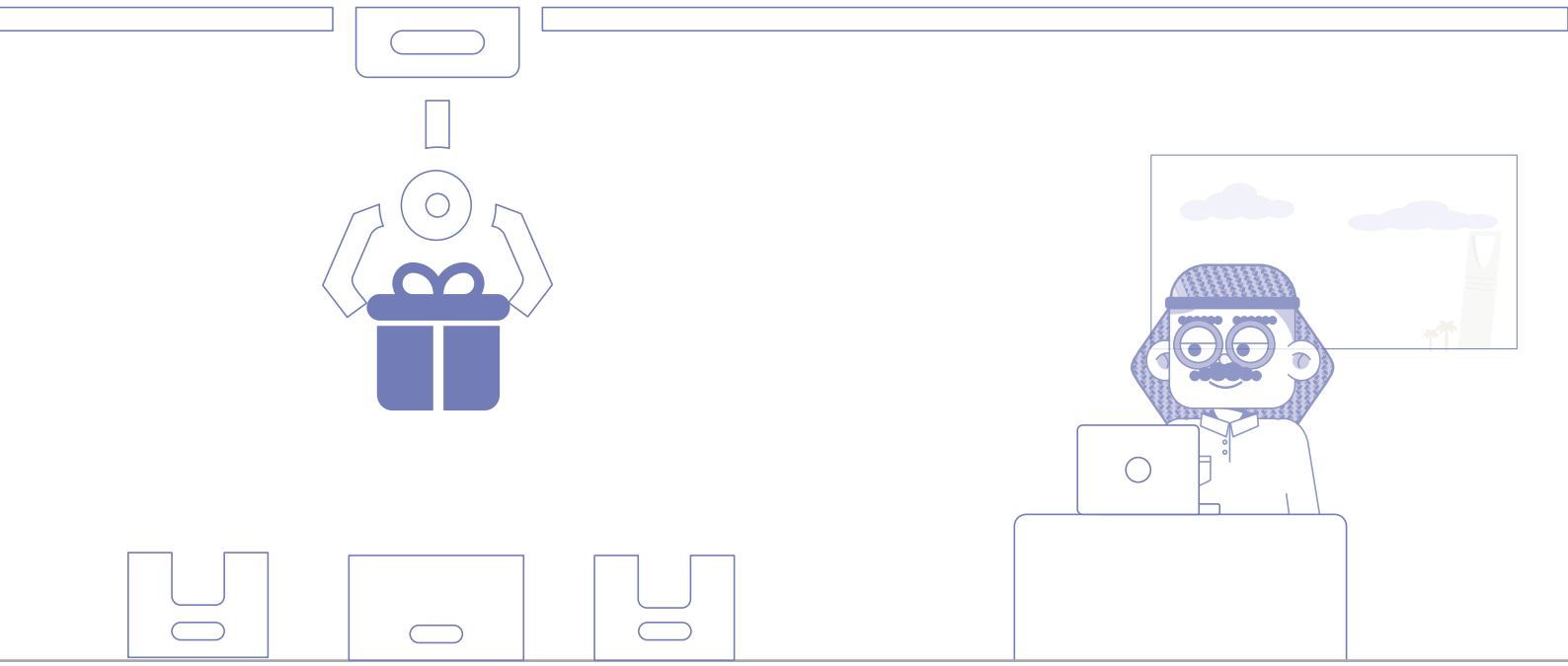
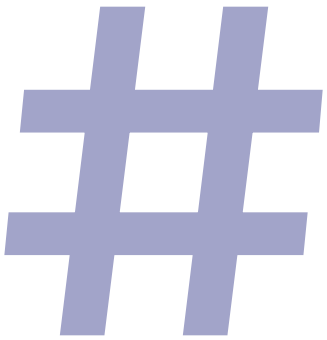
OPERATING SYSTEM

Input and Output Management



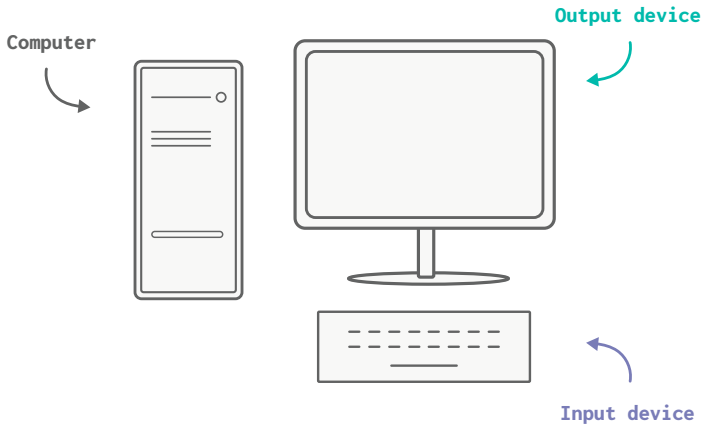


Input and Output Introduction

Step 01 Introduction

Computers are designed to process input and produce output. To achieve this, it needs to get this input from users via an input device and produce an output via an output device. Thus, input and output devices can be thought of as the communication method between the computer and the outside world.

> NOTE
> Both input and output devices are hardware components.



Step 02 What are Input Devices?

Input devices are responsible for delivering the input which can be data or commands from users to the computer. They serve as the interface between the user and the computer, enabling the computer to receive data for processing.

01 Examples

- Microphone
- Keyboard
- Mouse

Step 03 What are Output Devices?

Output devices are hardware components that convey information from the computer to the users. They convert processed data into a form that is understandable to humans depending on the output device type.

01 Examples

- Screen / Monitor
- Speaker
- Projector
- Headphones

Step 04 Device Drivers

An important concept regarding I/O devices is that each device has a specialized type of software called a device driver. This driver explains how the operating system can communicate with the device.

Since the early development of computers, hardware devices have been produced in large quantities. To avoid the complexity of instructing the computer to handle every newly produced device individually, drivers were developed. These drivers allow devices to connect to the computer by following the standards provided by the operating system for communication. This setup enables devices to operate when a system call triggers their execution. It is the responsibility of hardware manufacturers to develop these drivers.

> **NOTE**
> Device drivers translate generic OS commands into device-specific instructions, allowing the OS to communicate with a wide variety of hardware.

Step 05 User and Kernel Levels

A fundamental concept in operating systems is the user level and kernel level and the distinction between them. These levels differ based on the privileges each level has, OS is after all a process that controls other processes and the computer as a whole. To achieve this, it needs complete control over the computer and its resources including memory, CPU, and input/output devices.

User level is where user programs get executed, they don't need to access other processes' data or control the computer. As a result, they operate with limited privileges compared to the operating system (OS runs in the kernel level/space).

> **NOTE**
> The user and kernel space are execution context where the kernel has complete privileges and the user has limited privileges to execute.

01 Why have User and Kernel Levels?

The separation of the execution contexts (user and kernel), helps to secure the computer from malware programs and stabilize it by ensuring a failure in one process will not affect the computer as a whole. Also, it provides abstraction to user programs for easier communication and utilization of computer resources.

Step 06 How do I/O Devices Works?

When a user program needs an I/O device, it will request to access it from the operating system (via syscalls). Then, the execution context will switch from user mode to kernel mode so the operating system can run and access the device for the user program to complete its task.

Once the resource has been used and the OS finishes its task, the control will be given back to the user program after switching to user mode.

Step	Description
Request Access	It all starts when a user program sends a request to the operating system to get access to a specific I/O device. This is done through system calls.
Context Switch	Next, the system switches from user mode to kernel mode. This transition is crucial because it gives the operating system the necessary privileges to handle the hardware.
Device Access	Now, the operating system takes over communication with the I/O device, sending commands and managing the data transfer between the device and the program.
Completion of Task	Once the operation is finished, the operating system wraps up any related tasks, ensuring everything is in order.
Return Control	Finally, the control goes back to the user program by switching back to user mode, allowing it to continue running with the results from the I/O operation.

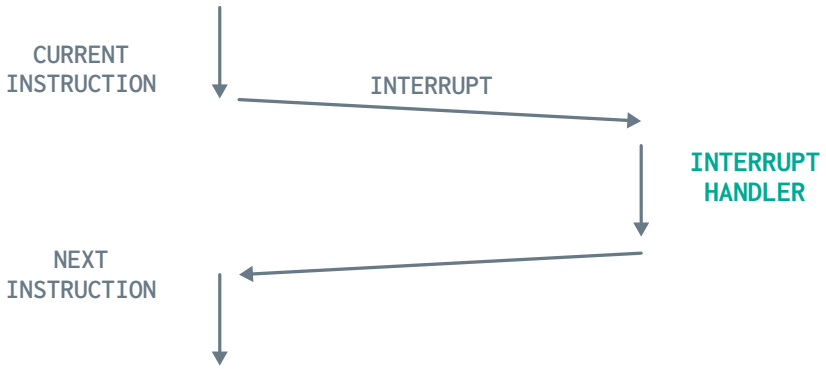
Step 07 How do I/O Requests Handled

To explain how requests are handled, we need to understand the concept of Exceptional Control Flow (ECF), which refers to the control of the execution flow in a computer. Ideally, computers execute instructions sequentially, one after another. However, depending on the computer's state, there are times when we need to stop the current instruction due to an error, a hardware fault, or the need to execute another instruction.

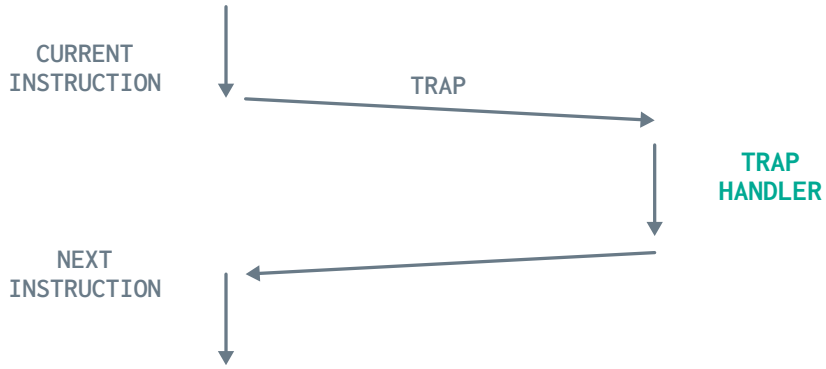
Depending on the reason, control may be transferred from a program running in user mode to the operating system, which runs in kernel mode, to handle the situation.

Exceptional Control Flow can be classified into the following categories:

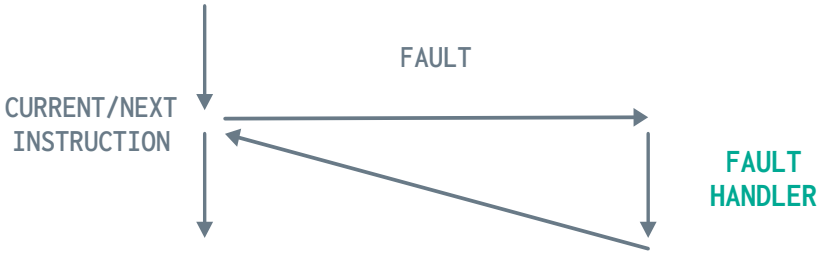
- **Interrupt:** This occurs when an I/O device completes its task by sending an interrupt signal to the processor, signaling that the data is ready.



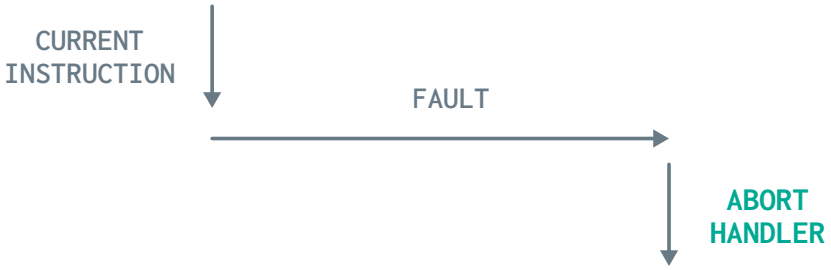
- **Trap (syscall):** A mechanism for a program to request a service from the operating system.

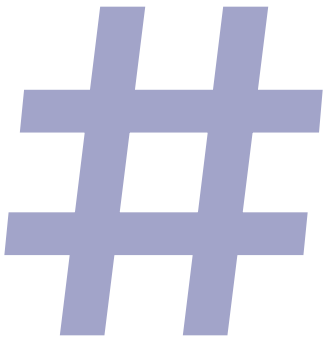


- **Fault:** An unexpected condition that occurs during instruction execution, often related to issues like invalid memory access.



- **Abort:** A severe error that causes the execution to be halted, often requiring a system reset.





I/O Management

Step 01 I/O Abstraction

Most operating systems including Unix-like systems and Windows, handle and see hardware devices integrating with the computer **as files**. This means accessing devices and requesting read/write operations on them is the same as read/write operations done to files. This approach enhances standardization and simplifies communication with hardware devices attached.

Step 02 How does File Abstraction Work on I/O Devices

Integrating a new device into your computer involves several steps. First, you need to plug in the device (e.g., printer, USB drive, external hard drive) into the computer using the appropriate port (USB, HDMI, etc.).

Once the device is connected, the operating system detects it and represents it as a file or a set of files (for example, disk drives appear as file systems). After the device is recognized, applications can access it using standard file operations such as **open**, **read**, **write**, and **close**., allowing for seamless interaction with the device.

Step 03 Conclusion

By treating devices as files, operating systems simplify the interaction between applications and hardware. This abstraction allows developers to use familiar file I/O operations while enabling the OS to manage devices efficiently and securely. This approach enhances usability and promotes consistency across different types of interactions with both physical and virtual devices.

> NOTE
> The name **File I/O** mirrors the concept of handling I/O devices as files since both use the same syscalls.

Step 04 Standard Input and Output

Standard input and output are the default devices set to present output or take input. Typically, they are referred to as standard input (stdin), standard output (stdout), and there is also standard error (stderr).

Normally, each standard I/O stream is connected to a device.

- **stdin**: Usually the keyboard.
- **stdout**: Typically the monitor (console).
- **stderr**: Also typically the monitor (console), used specifically for error messages.

In programming languages, there are functions that read from and write to these standard I/O stream.

- **stdin:** Functions like `scanf()` in C or `input()` in Python are used to read data from stdin.
- **stdout:** Functions like `printf()` in C or `print()` in Python are used to send data to stdout.
- **stderr:** Functions like `fprintf(stderr, ...)` in C or using `print()` with a file object in Python can send data to stderr.

Step 05 Standard I/O

01 stdin

The program prompts the user to enter an integer, which is then read and displayed.

```c
#include <stdio.h>

int main() {
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);
    printf("You entered: %d\n", number);
    return 0;
}
```

02 stdout

The program prints "**Hello, World!**" to the console.

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

03 stderr

An error message is printed to standard error, which can be redirected separately from standard

output.

```c
#include <stdio.h>

int main() {
    fprintf(stderr, “Error: Something went wrong!\n”);
    return 1;  // Return a non-zero value to indicate an error
}
```

04 Reading input from stdin and store it in a file

```c
#include <stdio.h>


int main(){
    char  string[50];
    scanf(“%s”, string);

    FILE* file;
    file = fopen(“new.txt”,”w”);

    if(file == NULL)
        printf(“%s”,”Failed to open the file”);

    fprintf(file, “%s”, string);
    fclose(file);

    return 0;
}
```

05 Reading input from a file and output it to the console (**output device**)

```c
#include <stdio.h>

int main() {
    FILE *file;
 // Buffer to hold each line of text
    char buffer[256];

    // Open the file for reading
    file = fopen(“example.txt”, “r”);
    if (file == NULL) {
        perror(“Error opening file”);
        return 1;
    }

    // Read each line from the file and print it to the console
    while (fgets(buffer, sizeof(buffer), file) != NULL) {
        printf(“%s”, buffer);
    }
    // Close the file
    fclose(file);
    return 0;
}
```