

# Computer Systems

IN BA4

Notes by Ali EL AZDI

## **Introduction**

This document is designed to offer a LaTeX-styled overview of the Computer Systems course, emphasizing brevity and clarity. Should there be any inaccuracies or areas for improvement, please reach out at ali.elazdi@epfl.ch for corrections. For the latest version of the PDF, you can check the following link: <https://elazdi-al.github.io/compsys/index.html>. Feel free to send a pull request to propose any changes you think might be a useful addition to the course content or a modification.

<https://github.com/elazdi-al/compsys/blob/main/main.pdf>

# Contents

<b>Contents</b>	<b>3</b>
<b>1 Lecture 01: Introduction</b>	<b>5</b>
1.1 The Journey of a YouTube Video . . . . .	5
1.1.1 Start of the Journey: Inside the Laptop . . . . .	6
Definition: Program, Process, Thread . . . . .	6
1.1.2 Accessing a Video: A Distributed Application . . . . .	6
1.1.3 Communication Protocols . . . . .	7
1.1.4 Distributed Applications and APIs . . . . .	7
Definition: Interface . . . . .	8
1.1.5 System Calls (Syscalls) . . . . .	8
1.2 The Operating System . . . . .	9
1.2.1 Example: Execution When Fetching a Video from YouTube . . . . .	9
1.3 Program and ISA . . . . .	10
1.3.1 Program . . . . .	10
Definition: ISA . . . . .	10
Definition: Von Neumann Architecture . . . . .	10
Definition: CPU Frequency . . . . .	11
1.4 Frequency Imbalance and CPU Caching . . . . .	11
1.4.1 CPU Caching . . . . .	12
1.5 Memory Accesses vs. I/O . . . . .	12
1.5.1 Memory Accesses . . . . .	12
1.5.2 Back to YouTube Fetching: System Calls in Action . . . . .	12
1.5.3 Mixing Interfaces . . . . .	13
Definition: Memory Access and I/O . . . . .	13
Definition: I/O . . . . .	13
1.6 Communication Over the Internet . . . . .	14
1.6.1 End Systems . . . . .	14
1.6.2 Packet Switches and Network Links . . . . .	15
1.6.3 Edge Caches . . . . .	15
1.7 Summary . . . . .	15
<b>2 L2 - All About Processes</b>	<b>17</b>
2.1 Multithreading . . . . .	17
2.2 Registers . . . . .	17
Definition: Compiler . . . . .	18
2.3 Memory Organization . . . . .	18
Definition: Memory Segments . . . . .	18
2.3.1 The Stack . . . . .	18
2.3.2 Heap Memory . . . . .	19

2.3.3 Data and Text Segments . . . . .	19
Definition: CPU Registers . . . . .	19
Definition: Process and Thread Identifiers . . . . .	19
Definition: Resource Sharing . . . . .	20
Definition: CPU Sharing . . . . .	20
Definition: Thread's CPU Context . . . . .	20
Definition: Context Switching . . . . .	20
Definition: Process . . . . .	20
Definition: Memory Sharing . . . . .	20
Definition: Virtual and Physical Addresses . . . . .	21
Definition: Virtual Address Space . . . . .	21
Definition: Address Translation . . . . .	21
2.3.4 Stack Smashing . . . . .	21
2.3.5 Summary: CPU and Memory Virtualization . . . . .	21
2.3.6 Conclusion . . . . .	21
<b>3 Sharing the CPU</b>	<b>22</b>
3.1 CPU Privilege Level . . . . .	22
3.1.1 Limited Direct Execution . . . . .	23
3.2 The OS as a Special Program . . . . .	23
3.2.1 CPU Privilege Levels and Execution Modes . . . . .	23
3.2.2 The Kernel: Core Component of the OS . . . . .	23
3.2.3 Process Management and Context Switching . . . . .	23
3.2.4 Syscalls . . . . .	24
3.2.5 Process I/O and Scheduling . . . . .	25
3.3 The Kernel's Job cont. . . . .	25
Definition: Timer Interrupt . . . . .	25
3.4 Executing Syscalls — Process Management . . . . .	26
3.4.1 Syscall Definitions . . . . .	26
Definition: Exit Syscall . . . . .	26
Definition: Exec Syscall . . . . .	27
Definition: Fork Syscall . . . . .	27
Definition: Wait Syscall . . . . .	27
3.4.2 Process Creation and Cleanup . . . . .	27
3.5 The OS Process Graph . . . . .	28
3.6 Key Processes in the OS . . . . .	28

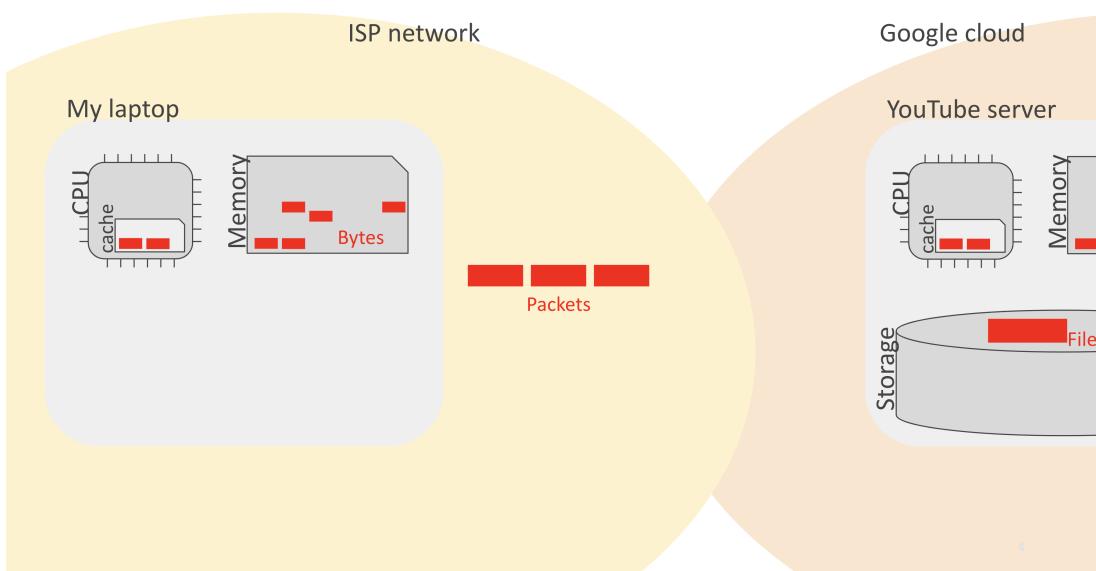
# Chapter 1

## Lecture 01: Introduction

In this lecture we explore the journey of a YouTube video—from its storage as a file to its transformation into bytes, its transmission over networks, and finally, its display on your device. We will introduce key concepts such as processes, threads, distributed applications, system calls, and the role of the operating system in managing hardware resources.

### 1.1 The Journey of a YouTube Video

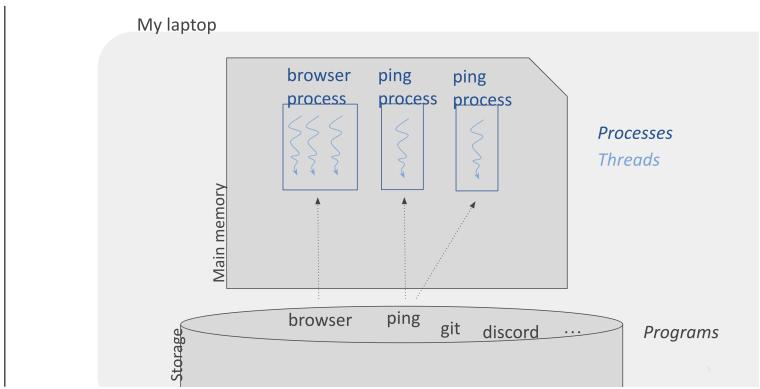
To illustrate these ideas, consider the journey of a YouTube video. The video begins its existence as a file stored on a storage device, is loaded into memory as bytes, transmitted as packets over the Internet, and finally rendered on your screen.



### 1.1.1 Start of the Journey: Inside the Laptop

The journey begins on your laptop.

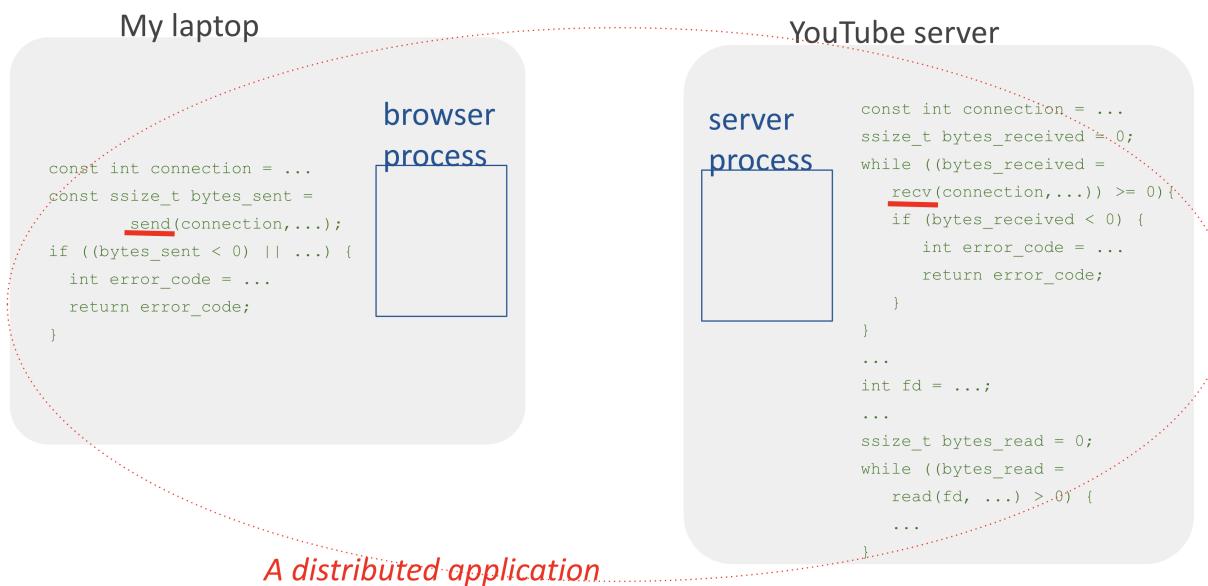
A computer hosts many different programs (e.g., a web browser, a ping utility, a git client). These programs, stored as files on disk, are invoked by user actions such as clicking an icon or typing a command. When a program is invoked, the computer creates a new *process* in main memory. A process represents a running instance of a program and may consist of one or more *threads*—individual units of execution within the process.



**Definition (Program, Process, Thread).** A *program* is a set of instructions stored as a file on disk. When a program is invoked, the computer creates a *process*—a running instance of that program in main memory. A process may consist of one or more *threads*, which are the individual sequences of execution within the process.

### 1.1.2 Accessing a Video: A Distributed Application

When you use your web browser to access a video, the browser sends a message (or request) to a remote YouTube server. The browser process (running on your laptop) and the server process (running on a different computer) work together as parts of a *distributed application*.

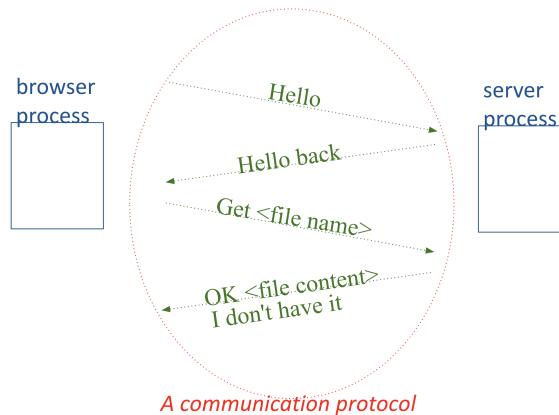


### 1.1.3 Communication Protocols

For two processes running on different devices to work together, they must follow a predetermined set of rules known as a **communication protocol**. For example, a simple protocol might involve:

- One process sending “hello” and waiting for a “hello back.”
- A subsequent request for a specific file (e.g., xyz) with the server responding with the file or an error message.

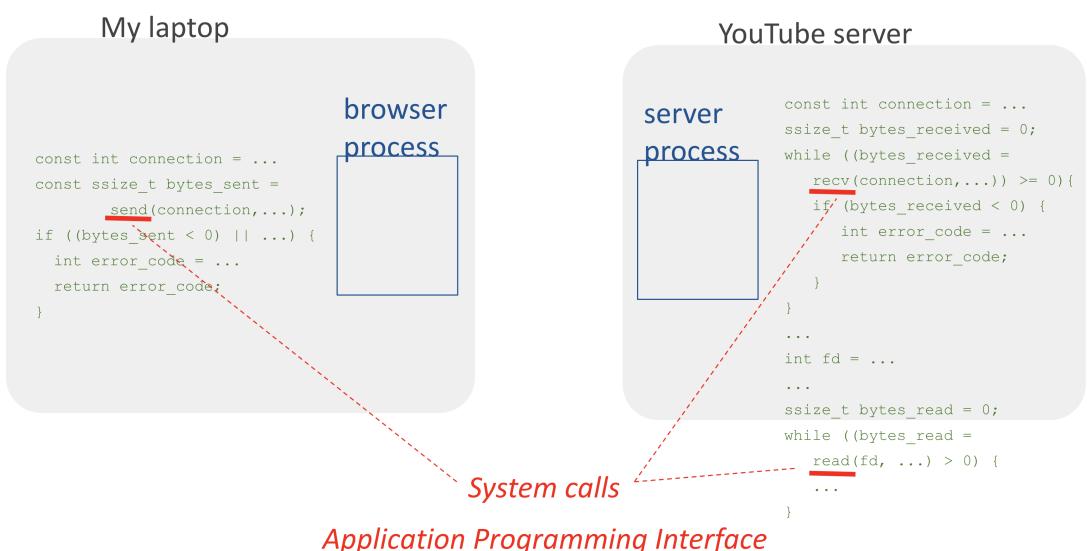
Much like human communication, these protocols ensure that both parties know what to expect, enabling effective interaction.



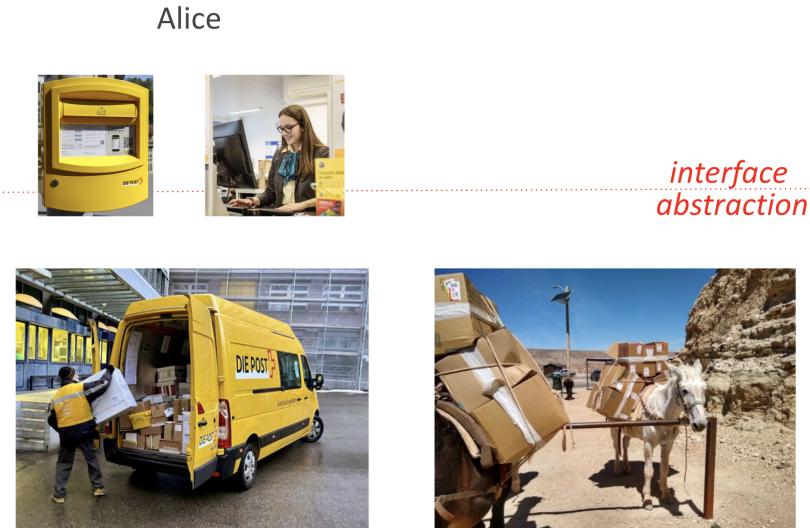
### 1.1.4 Distributed Applications and APIs

Distributed applications consist of separate pieces of code running as processes on different machines but working toward a common goal. These processes exchange messages over the Internet by following communication protocols. To simplify the development of these applications, developers use *system calls* (or **syscalls**). Syscalls are special functions provided by the operating system that allow processes to access resources (e.g., network and storage) without needing to know the low-level details.

The set of syscalls available to an application forms its **Application Programming Interface** (API), abstracting away the complexities of resource management.

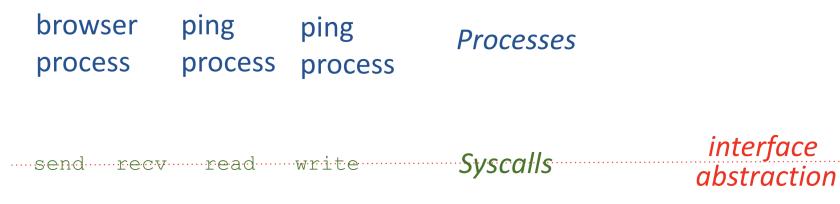


**Definition (Interface).** An *interface* is a set of rules that defines how different components communicate. For instance, when sending a letter via the postal system, one must follow specific rules (e.g., write the address and affix a stamp). This interface abstracts the complexities of the postal system so that users do not need to understand its internal operations.



### 1.1.5 System Calls (Syscalls)

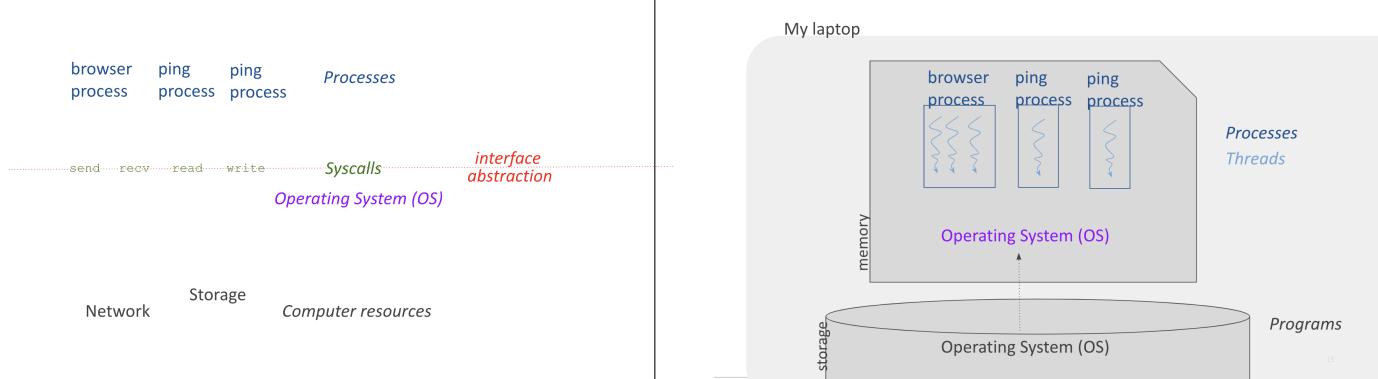
Syscalls form the interface between a process and external resources (like network and storage). They provide an abstraction of these resources, allowing a process to use them without knowing their intricate details. For example, when a process makes a syscall such as `send` or `recv`, the operating system's network stack handles the details of the communication.



Network              Storage  
Computer resources

## 1.2 The Operating System

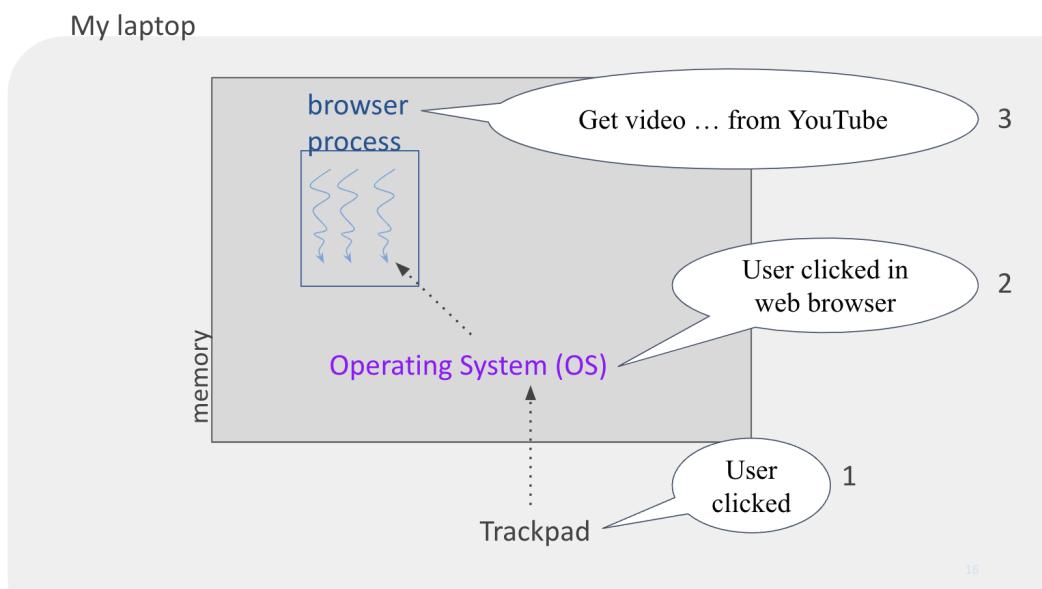
Conceptually, the OS sits between running processes and the underlying hardware resources. It provides the syscall interface and handles tasks such as file system management and network communication.



### 1.2.1 Example: Execution When Fetching a Video from YouTube

When you click a YouTube link, the following sequence of events occurs:

1. Your trackpad detects the click and notifies the OS.
2. The OS identifies that the click occurred within the web browser window and alerts the corresponding process.
3. The browser process initiates a chain of events that eventually fetches and displays the video.



## 1.3 Program and ISA

### 1.3.1 Program

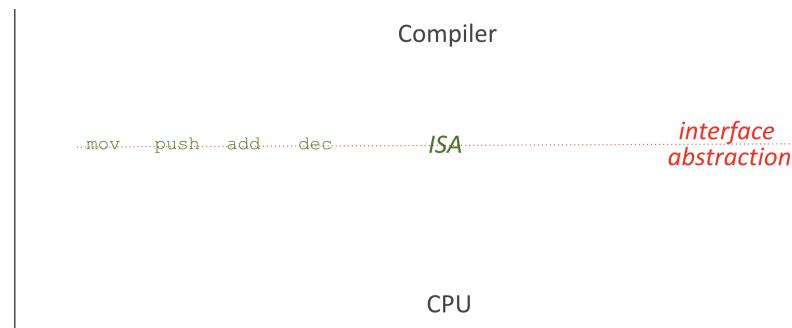
A **program** is a set of instructions written by a human in a high-level programming language (such as C, Java, or Python) that implements an algorithm. When compiled, a program is translated into an *executable* (or binary) that the CPU can run. The executable is expressed in the language defined by the computer's **Instruction Set Architecture** (ISA).

*A C program* ..... ➔ Compiler ..... ➔ *An executable program  
(almost; it's assembly)*

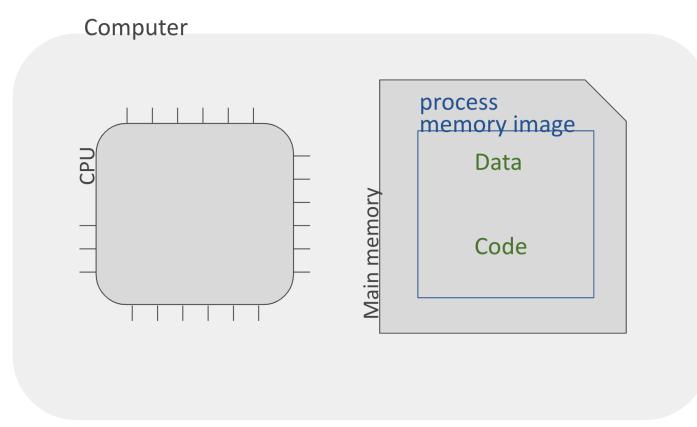
```
int sum = 0;
int main(...) {
    int count = 10;
    ...
    sum = sum + count;
    count = count - 1;
    ...
    return 0;
}
```

```
...
sum resd 1
_main:
...
mov rcx, 10
push rcx
mov rax, [sum]
add rax, rcx
dec rcx
mov [sum], rax
...
```

**Definition (ISA).** *The Instruction Set Architecture (ISA) is the set of all instructions that a CPU can understand and execute. It forms an interface between the compiler (which translates high-level code into machine code) and the CPU.*



**Definition (Von Neumann Architecture).** *The vast majority of computers today follow the Von Neumann architecture, which is characterized by a single main memory that holds both data and instructions.*



*The Von Neumann architecture*

**Definition (CPU Frequency).** A CPU's frequency indicates how many cycles it can perform in one second. For example, a 4.05 GHz CPU performs 4.05 billion cycles per second. In this context, a **cycle** is the minimum time needed for the CPU to complete an operation or for a result to become ready.

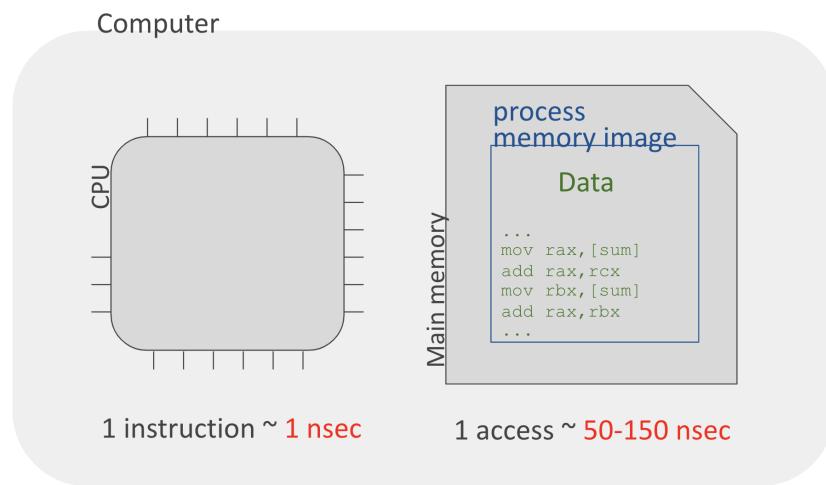
**Question:** What is the meaning of the Hz metric?

**Answer:** Hertz (Hz) measures the number of cycles per second. **Question:** In the context of a CPU, what is a cycle?

**Answer:** A cycle is the minimum unit of time required for the CPU to produce a result from executing an instruction.

## 1.4 Frequency Imbalance and CPU Caching

Modern systems exhibit a *frequency imbalance* between the CPU and main memory. While the CPU might complete an instruction in approximately 1 nsec, accessing data from DRAM typically takes 50–150 nsec. As a result, the CPU can spend a significant amount of time idle, waiting for data.

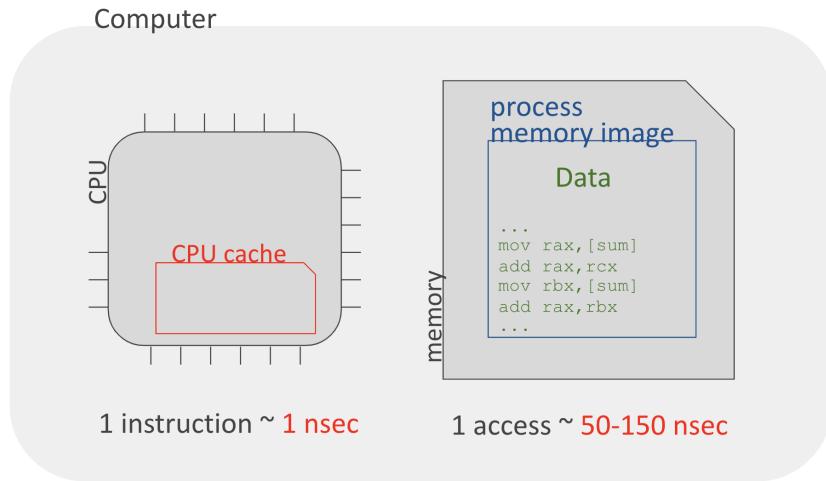


**Question:** How can we improve the efficiency of a system with such a frequency imbalance?

**Answer:** We can improve efficiency by using caching. A small, fast memory (the CPU cache) stores recently accessed data so that subsequent accesses are much faster.

### 1.4.1 CPU Caching

CPU caching adds a small amount of high-speed memory inside the CPU. When data is requested, the cache is checked first. If the data is already in the cache (a cache hit), the CPU retrieves it quickly. Otherwise (a cache miss), the data is fetched from the slower main memory and stored in the cache for future accesses.



## 1.5 Memory Accesses vs. I/O

### 1.5.1 Memory Accesses

When a process reads or writes data in main memory, it uses fast CPU instructions (load/store). These operations are efficient and do not interrupt the normal flow of the process.

### 1.5.2 Back to YouTube Fetching: System Calls in Action

Returning to our YouTube example, when the browser process calls a `send` syscall to request a video, the CPU stops executing the browser's code and switches to executing the more privileged code in the OS. This is because accessing external resources (network or storage) requires a syscall.

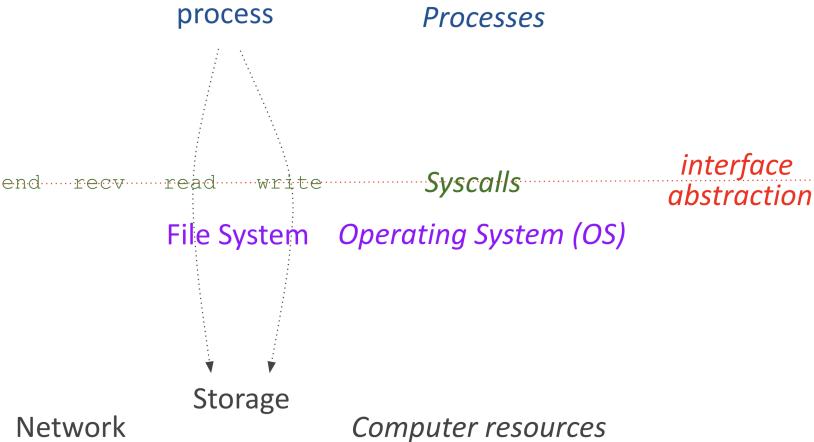
*The browser C code* ..... ➔ Compiler ..... ➔ *The browser executable*

```
...
const int connection = ...
const ssize_t bytes_send =
    send(connection,...);
if ((bytes_sent < 0) || ...) {
    int error_code = ...
    return error_code;
}
...
...
```

```
...
mov rax, 44
mov rdi, [connection]
mov rsi, ...
mov rdx, ...
syscall
...
```

### 1.5.3 Mixing Interfaces

When a process makes a syscall for network communication (such as `send` or `recv`), the CPU transitions from running the process's code to running the OS code associated with the network stack. This is an example of mixing different interfaces: the process interface (its own code) and the OS interface (syscalls).



**Definition (Memory Access and I/O).** *Memory Access* refers to the CPU's direct read/write operations using load/store instructions in main memory. In contrast, **I/O (Input/Output)** involves accessing external devices (such as storage or network) via system calls. I/O operations are generally more expensive because they require the CPU to switch context to execute privileged OS code.

**Exam Question:** How is reading from main memory different from reading from storage or the network?

**Answer:** Reading from main memory uses direct CPU instructions (load/store) and is very fast (tens to hundreds of nanoseconds), whereas reading from storage or the network requires a syscall, which interrupts the process and involves additional overhead (microseconds to milliseconds).

**Definition (I/O).** *I/O (Input/Output)* refers to operations that allow a process to access resources outside of its immediate control, such as storage devices or the network, via system calls. I/O operations are generally slower than direct memory accesses.

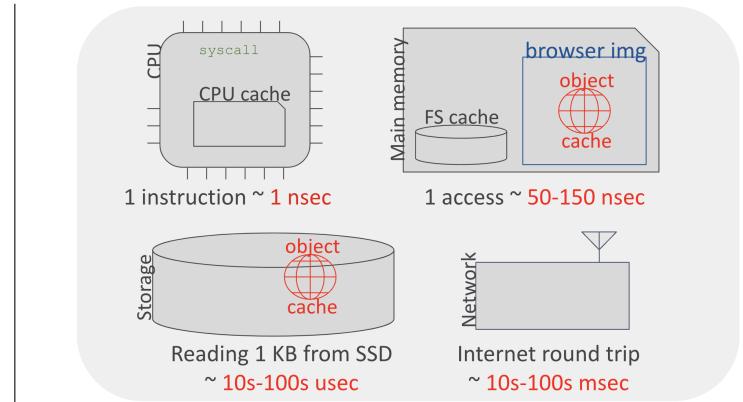
**Exam Question:** If a program does not create or manipulate any data, will executing this program require reading anything from memory?

**Answer:** Yes, executing the program will still require reading something from memory. Even if the program does not create or manipulate any data, the CPU must at least fetch the instructions of the program itself from memory.

## 1.6 Communication Over the Internet

Internet communication involves transferring data over a network where different latencies are encountered:

- Simple CPU instructions:  $\sim 1$  nsec.
- Main memory accesses: tens to hundreds of nanoseconds.
- Reading 1 KB from an SSD: tens to hundreds of microseconds.
- Requesting data over the Internet: several to hundreds of milliseconds.



These delays make network communication expensive in terms of time, which is why caching is critical.

### 1.6.1 End Systems

The Internet is composed of **end systems**—devices that use the network for communication. These include laptops, smartphones, household appliances, connected cars, and even medical devices, as well as large cloud servers.

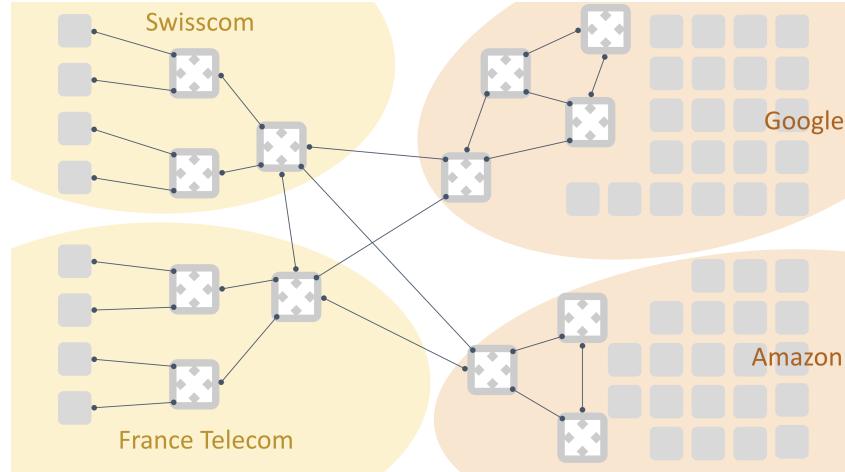


### 1.6.2 Packet Switches and Network Links

In addition to end systems, the Internet relies on:

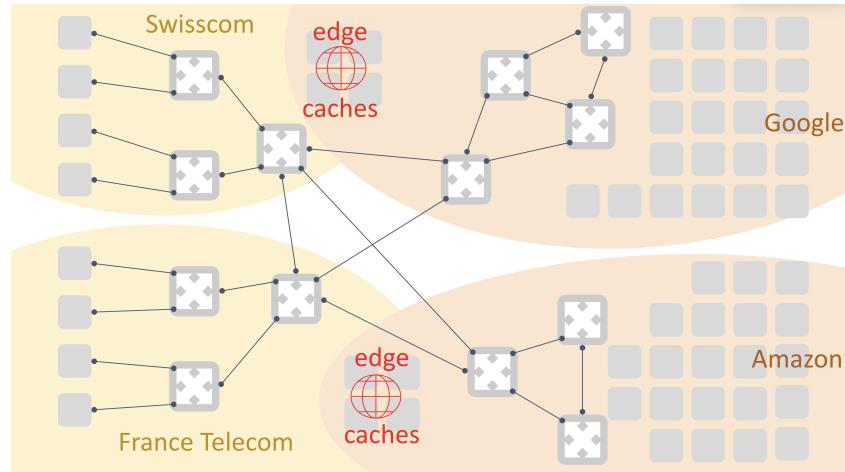
- **Packet Switches:** Devices that route data between end systems.
- **Network Links:** Physical connections that interconnect packet switches and end systems.

These components are managed by Internet Service Providers (ISPs) as well as major cloud providers.



### 1.6.3 Edge Caches

To reduce the load on cloud data centers and improve user performance, large cloud providers often deploy **edge caches** within ISP networks. These caches store frequently accessed content closer to the end-users, reducing latency and network traffic.

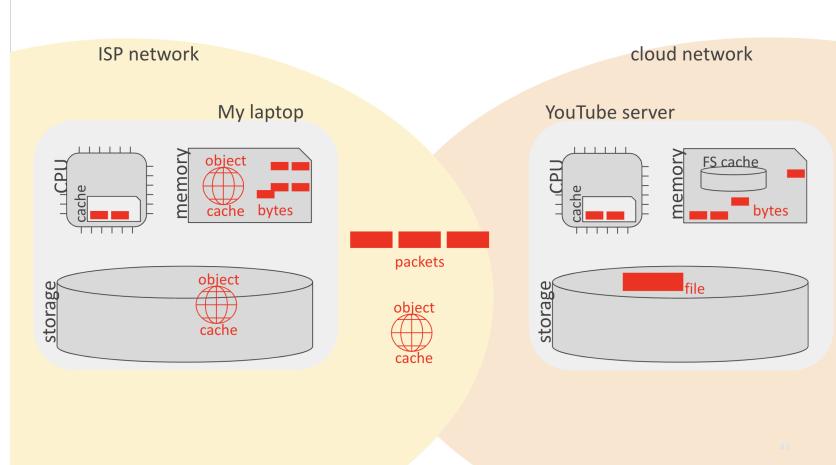


## 1.7 Summary

In this lecture, we traced the journey of a YouTube video and introduced several fundamental concepts:

- **Programs, Processes, and Threads:** Programs stored on disk become processes (and threads) when executed.
- **Distributed Applications:** Different processes communicate over networks using well-defined communication protocols.

- **Interfaces and Abstractions:** System calls, APIs, and caching abstract the complexity of hardware resources.
- **The Operating System:** Acts as an intermediary between processes and hardware resources.
- **Performance Considerations:** Frequency imbalances between the CPU and memory are mitigated by caching at various levels (CPU cache, file system cache, object caches).



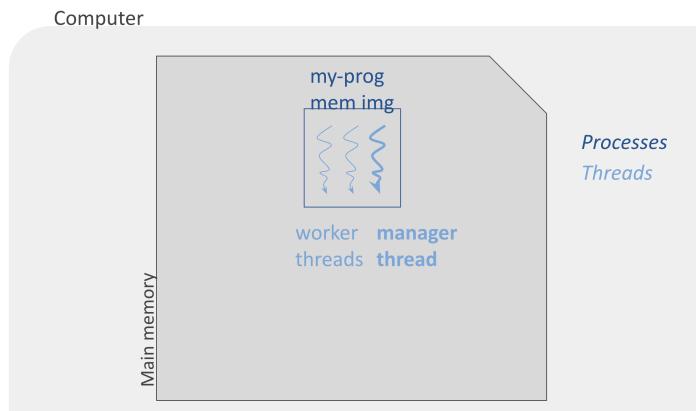
# Chapter 2

## L2 - All About Processes

This chapter provides an overview of the fundamental concepts underlying modern process management and memory organization in computer systems. We discuss multithreading, CPU registers, the role of compilers, and memory organization—including both stack and heap memory—as well as virtualization techniques that allow multiple processes to coexist seamlessly.

### 2.1 Multithreading

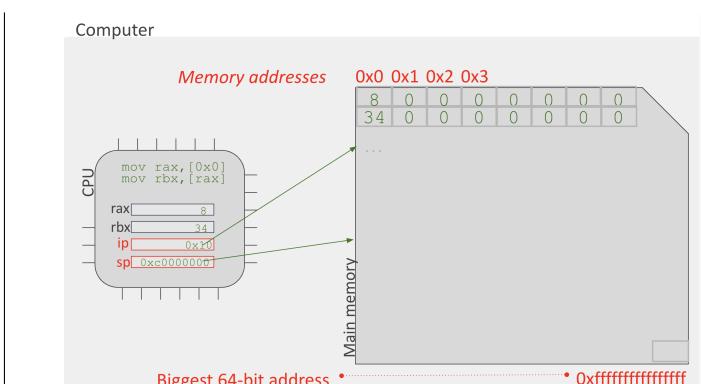
When a program runs, the processor loads it into main memory and creates a thread. In a multithreaded program, there is typically one *manager* thread that delegates work to several *worker* threads. For instance, when computing the sum of a large set of numbers, the workload can be divided into subsets, with each worker thread processing a portion of the data while the manager coordinates the overall computation.



### 2.2 Registers

CPU registers are small storage locations within the processor that hold data and instructions needed during execution. For example, the `mov` instruction might transfer data from one register (or memory location) to another. Key registers include:

- **Instruction Pointer (IP):** Keeps track of the next instruction to be executed.
- **Stack Pointer (SP):** Points to the current top of the stack in main memory.



**Definition (Compiler).** A compiler translates high-level source code (such as C) into low-level executable code (often Assembly language). This translation involves parsing, optimization, and the generation of machine-specific instructions.

## 2.3 Memory Organization

I'll go a little bit deeper for our fellow syscoms, I also recommend understanding how LIFO works before reading this, if you're too lazy for that, it's basically in the name Last In First Out, means that the last item pushed onto the stack is the first one to be removed, just like stacking plates— you take the top plate first before reaching the ones below.

In modern computer architectures, a process's memory is divided into several distinct segments, each serving a specific role during program execution. Understanding these segments is fundamental for effective programming and debugging.

**Definition (Memory Segments).** A process's memory image is typically divided into the following segments:

- **Text Segment:** Contains the executable code and embedded constants. It is usually marked as read-only to prevent accidental modification.
- **Data Segment:** Stores global and static variables. This segment is often subdivided into:
  - **Initialized Data:** Variables explicitly initialized by the programmer.
  - **Uninitialized Data (BSS):** Variables that are declared but not explicitly initialized, and are set to zero by default.
- **Heap Segment:** Used for dynamic memory allocation. Memory here is allocated and deallocated during runtime by the programmer (or automatically via garbage collection in some languages). The heap typically grows upward (from lower to higher memory addresses).
- **Stack:** Manages function calls, local variables, and function parameters. The stack is automatically managed by the CPU, growing downward (from higher to lower memory addresses) as functions are called.

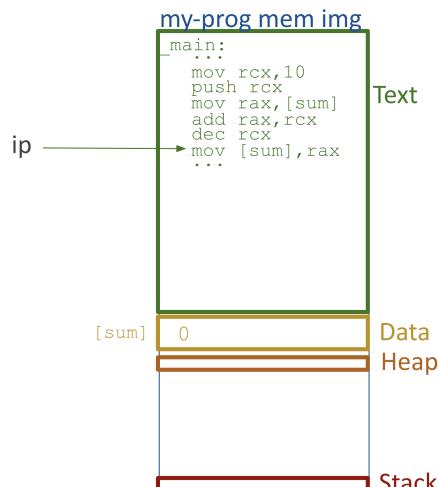
### 2.3.1 The Stack

The stack is a dedicated region of memory that the CPU uses to manage function calls and local variables. When a function is invoked:

1. The CPU executes a `call` instruction, which pushes the return address onto the stack.
2. A new *stack frame* is created to store local variables and function-specific data.
3. Upon function return, the stack frame is removed (or "unwound"), and control returns to the calling function.

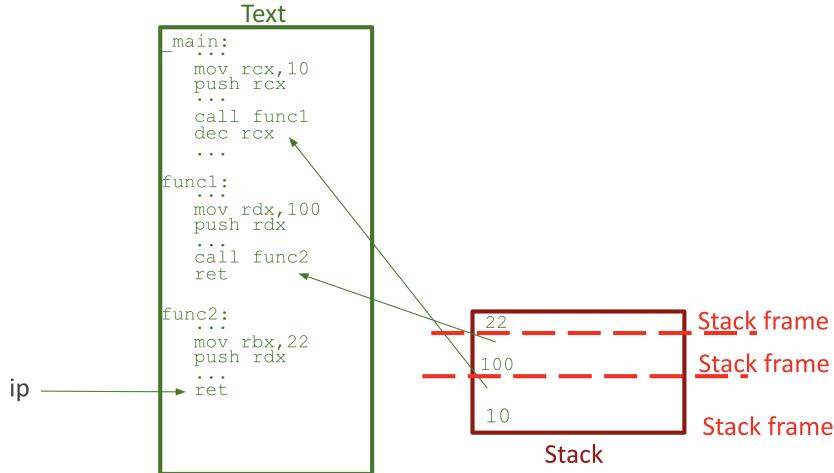
#### Key Characteristics of the Stack:

- **Automatic Management:** The CPU automatically handles pushing and popping of data.
- **Growth Direction:** Grows downward, from higher to lower memory addresses.
- **Contents:** Stores return addresses, local variables, and sometimes function arguments.



## Stack Frames

Each function call creates its own *stack frame*, a self-contained section that isolates the function's local data. This segmentation helps maintain the correct scope and lifetime for local variables and ensures that return addresses are preserved. The following diagram illustrates the organization of stack frames during nested function calls:



### 2.3.2 Heap Memory

The heap is used for dynamic memory allocation, where memory is allocated and deallocated as needed during runtime. Unlike the stack:

- **Manual vs. Automatic Management:** In languages such as C or C++, the programmer is responsible for explicitly allocating (using `malloc` or `new`) and deallocating (using `free` or `delete`) heap memory. In contrast, some modern languages employ automatic garbage collection.
- **Growth Direction:** The heap typically grows upward, from lower to higher memory addresses.
- **Lifetime:** Data allocated on the heap persists beyond the scope of the function that created it, until it is explicitly freed or garbage collected.

### 2.3.3 Data and Text Segments

**Text Segment:** This segment contains the program's executable code and constant values. Its read-only nature helps prevent inadvertent modifications during execution. **Data Segment:** This segment holds global and static variables. It is divided into:

- **Initialized Data:** Variables that have been assigned an initial value at compile time.
- **Uninitialized Data (BSS):** Variables that are declared but not explicitly initialized; these are automatically set to zero at program startup.

**Definition (CPU Registers).** *Two registers are critical for process execution:*

- **Instruction Pointer (IP):** Points to the next instruction in the text segment.
- **Stack Pointer (SP):** Points to the top of the stack.

**Definition (Process and Thread Identifiers).**

*A process is considered to be running if at least one of its threads is executing; otherwise, it is not running.*

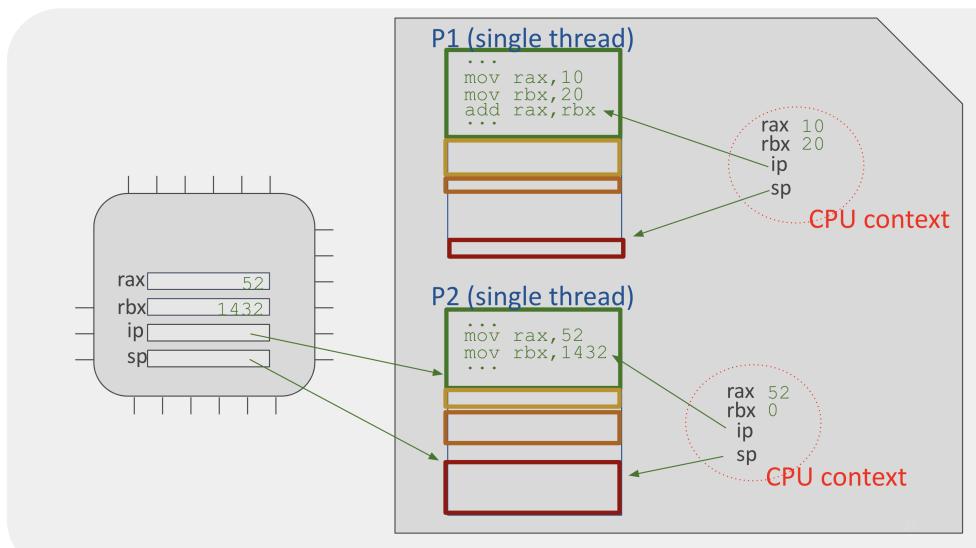
- **Process ID (PID):** A unique identifier assigned to each process.
- **Thread ID (TID):** A unique identifier for each thread, which may be unique within a process or across the entire system, depending on the operating system.

**Definition (Resource Sharing).** Processes and threads share system resources such as CPU and memory. Each thread is given the illusion of having exclusive access to the CPU, and each process appears to have dedicated memory, even though these resources are actually shared.

**Definition (CPU Sharing).** The CPU is time-shared among multiple threads. This virtualization is achieved through context switching, where the CPU rapidly switches between threads, giving each one the impression of exclusive use of the processor.

### Example: Two Programs Running on a Single Core

**Example 2.3.3.1.** Consider two programs running on a single-core processor. Each program is assigned a CPU context, which includes register values such as `rax`, `rbx`, the stack pointer (`SP`), and the instruction pointer (`IP`). When switching between programs, the CPU saves the current context to memory and loads the context of the next program, allowing the programs to resume correctly.



**Definition (Thread's CPU Context).** A thread's CPU context comprises the values of all CPU registers at the moment it was last executing. In a single-threaded process, this context represents the entire process state.

**Definition (Context Switching).** Context switching is the process by which the CPU switches from executing one thread to another. It involves:

1. Saving the current thread's CPU context to memory.
2. Restoring the CPU context of the thread to be executed next.

Each thread has the illusion that it's occupying the CPU **alone**.

This mechanism enables CPU virtualization but introduces performance overhead due to additional memory accesses.

**Definition (Process).** A process is defined by:

- A unique Process ID (PID).
- A memory image that includes the text, data, heap, and stack segments.
- The CPU contexts of each thread within the process.
- Associated resources such as file descriptors.

*Remark: If two threads belong to the same process, does each have its own CPU context, or do they share one? The answer is that each thread should be able to continue its execution independently.*

**Definition (Memory Sharing).** *Memory in a system is space-shared among processes; however, virtualization ensures that each process operates within its own isolated address space. This is achieved through virtual-to-physical address translation.*

**Definition (Virtual and Physical Addresses).** *Virtual addresses allow processes to operate as if they have exclusive access to memory. For example, two processes might both use the virtual address 0x400000; however, these addresses map to different physical addresses:*

- Process  $P_1$ : Virtual 0x400000 → Physical 0x1234AFF8
- Process  $P_2$ : Virtual 0x400000 → Physical 0xABCD5678

**Definition (Virtual Address Space).** *Each process is allocated its own virtual address space, which is shared among all its threads. This abstraction allows developers to ignore the complexities of physical memory allocation.*

**Definition (Address Translation).** *Address translation is the process by which a virtual memory address is converted into a physical memory address. While essential for memory virtualization, this translation incurs a performance cost.*

#### 2.3.4 Stack Smashing

Stack smashing is a type of buffer overflow vulnerability where an attacker deliberately overwrites parts of the memory on the call stack. This typically happens when a program writes more data into a fixed-size buffer than it can accommodate, thereby corrupting adjacent memory areas such as the function's return address.

**How It Works:** When a function is called, a stack frame is created that contains local variables, return addresses, and other control data. If a buffer does not have proper bounds checking, an input exceeding the buffer's capacity can overflow into these critical areas. For example:

1. A fixed-size buffer is allocated on the stack.
2. Excess input data overwrites memory beyond the buffer.
3. The return address (or other control data) is corrupted.
4. On function return, control is transferred to an address chosen by the attacker, potentially executing malicious code.

#### 2.3.5 Summary: CPU and Memory Virtualization

- **CPU Virtualization:** Threads time-share the CPU through context switching, which gives each thread the illusion of exclusive CPU access.
- **Memory Virtualization:** Processes space-share memory via virtual-to-physical address translation, ensuring that each process operates in its own isolated address space.

#### 2.3.6 Conclusion

Modern operating systems are designed to enable multiple programs to share CPU and memory resources seamlessly. Through context switching and address translation, both the CPU and memory are effectively virtualized. This abstraction simplifies development, as compilers and developers can design programs without needing to manage these low-level resource-sharing details directly.

# Chapter 3

## Sharing the CPU

This chapter introduces the mechanisms by which an operating system (OS) manages access to the CPU, ensuring both security and fairness among processes. We discuss CPU privilege levels, the limited direct execution model, and the role of system calls (syscalls) in process management.

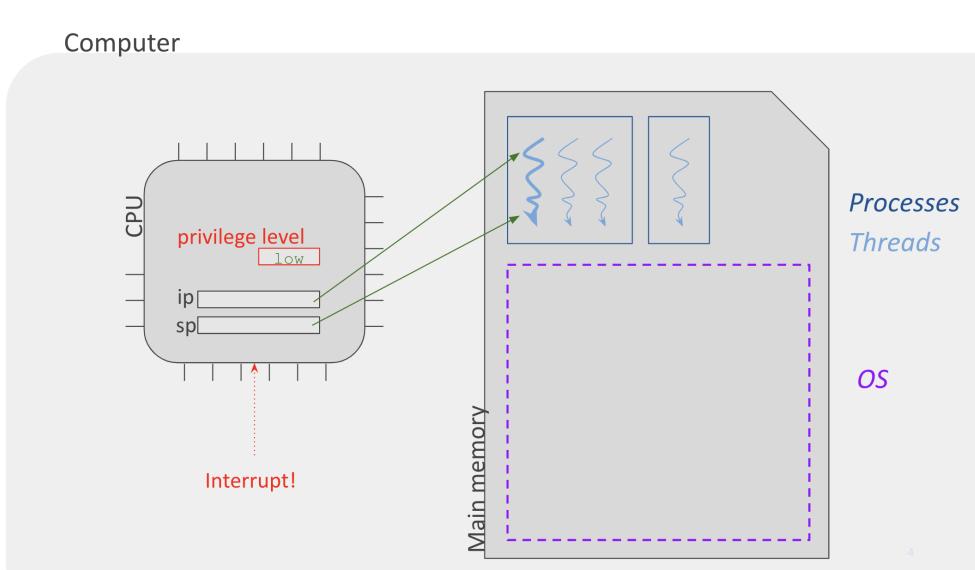
### 3.1 CPU Privilege Level

**Professor Analogy** - Imagine you have children who want to watch TV. You can either keep the remote with you and ask them to call you whenever they need to change the channel, or you can enable Kids Mode on the TV, allowing them limited access without needing to ask for permission each time.

The CPU operates in different modes to regulate access to its instruction set:

- **Low Privilege (User Mode):** A normal process or thread runs in this mode and can execute only a restricted subset of instructions.
- **High Privilege (Kernel Mode):** The operating system runs in this mode and has full access to all instructions and hardware resources.

When a regular thread executes, the CPU remains in low privilege mode. Conversely, OS code runs in high privilege mode, allowing it to perform otherwise restricted operations. This design follows the principle of least privilege, ensuring that processes have only the necessary level of access.



### 3.1.1 Limited Direct Execution

**Professor Analogy cont.** - The children have direct access to the TV but are restricted by Kids Mode. Limited direct execution is a method that allows a thread to run directly on the CPU while enforcing certain restrictions:

- *Direct Execution*: The CPU executes the thread's instructions without any intermediate emulation.
- *Limited*: The thread cannot perform operations that require high privileges; instead, it must request system assistance via syscalls.

This approach minimizes overhead while ensuring that potentially unsafe operations are controlled by the OS.

## 3.2 The OS as a Special Program

The operating system (OS) is a fundamental software layer that manages hardware resources and provides essential services for user applications. Unlike typical applications, the OS operates at different privilege levels and ensures secure and efficient execution of processes and threads.

### 3.2.1 CPU Privilege Levels and Execution Modes

The OS shares the CPU and main memory with normal user processes and threads. However, its execution mode differs based on its role at any given time:

- When the OS executes, the CPU may be in **high privilege mode** (kernel mode), allowing it unrestricted access to all system resources.
- When a normal process or thread executes, the CPU is in **low privilege mode** (user mode), restricting access to critical system resources.

This privilege system exists primarily for security reasons, enforcing the **principle of least privilege**, where each process has only the necessary access rights required to perform its task.

### 3.2.2 The Kernel: Core Component of the OS

A key component of the OS is the **kernel**, responsible for:

- Creating and managing processes and threads.
- Allocating system resources (CPU time, memory, I/O devices, etc.).
- Ensuring that each process has a designated portion of memory, including stack, data, and text segments.
- Enforcing security and isolation between processes.

The kernel always runs in **high privilege mode** (kernel mode), which is why this mode is sometimes referred to as *kernel mode*.

### 3.2.3 Process Management and Context Switching

The OS does not execute continuously but rather runs only when necessary. It performs essential tasks, prepares the environment for the next process, and then switches out, allowing user processes to execute.

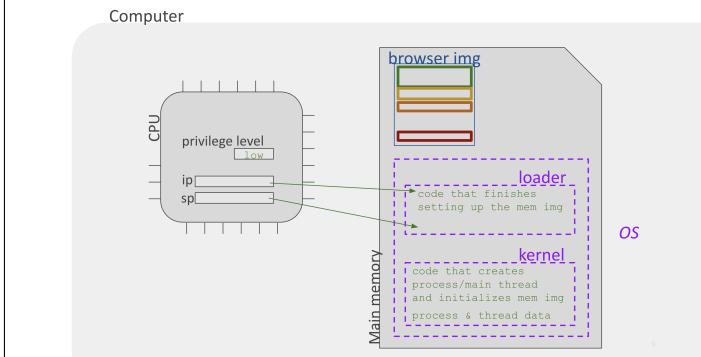
- The OS uses **context switching** to transition between processes efficiently, preserving the state of the current process before switching to another.
- By switching between kernel mode and user mode, the OS ensures that user applications run securely and do not directly manipulate hardware resources.

## The Loader: Setting Up Process Memory

Another critical component of the OS is the **loader**, which is responsible for preparing a program for execution:

- The loader completes the setup of the process's memory image.
- It copies the command-line arguments used to launch the program (e.g., `ls -a`, where `-a` is an argument).
- These arguments are stored in the **stack** of the main thread of the new process to ensure accessibility.

If these arguments were stored in the loader's stack instead, the process would not be able to access them after execution begins.

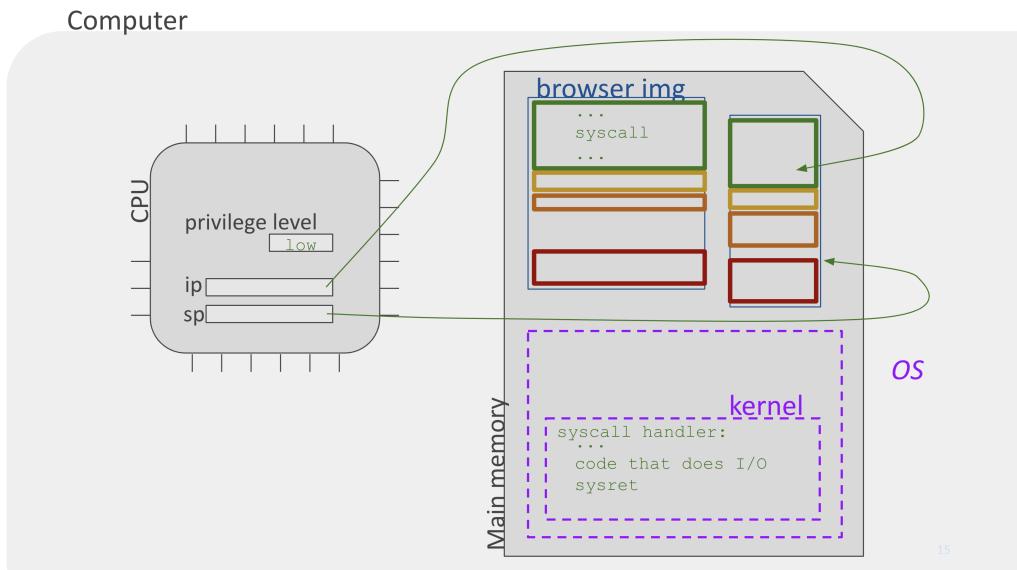


This layered privilege model ensures a secure and efficient execution environment, maintaining stability and protection across all processes within the system.

### 3.2.4 Syscalls

*Personal Remark - Syscalls are NOT needed when executing OS code, they only allow to cross the privileged barrier when running a unprivileged code. A syscall is the mechanism by which a user process requests services from the OS. When a syscall is invoked:*

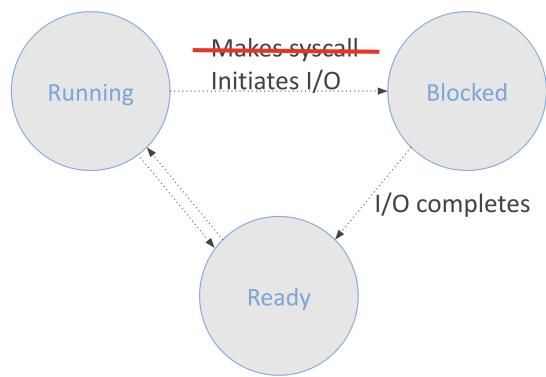
1. The CPU temporarily elevates the privilege level.
2. The control is transferred to the OS to execute the privileged code.
3. If the syscall involves an I/O operation, the process may be blocked while waiting for a response, and the CPU may perform a context switch to another thread.
4. Once the operation is complete, the CPU returns to low privilege.



### 3.2.5 Process I/O and Scheduling

When a process initiates an I/O operation:

- It transitions from *Running* to *Blocked* as it waits for the I/O to complete.
- Once the I/O is complete, the process moves to the *Ready* state.
- The OS scheduler then selects processes from the Ready queue to run, ensuring fair CPU utilization.



## 3.3 The Kernel's Job cont.

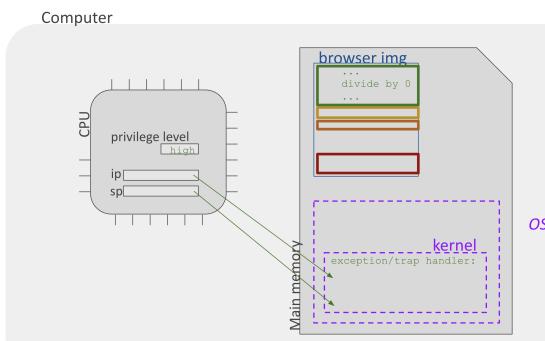
The Kernel handles several types of events:

- **Syscalls:** Requests from running threads for system-level services.
- **Exceptions/Traps:** Synchronous signals generated when a thread executes an illegal or erroneous operation (e.g., division by zero).
- **Interrupts:** Asynchronous signals from external devices (e.g., mouse events, network packets) requiring immediate attention.

### Exception Handling

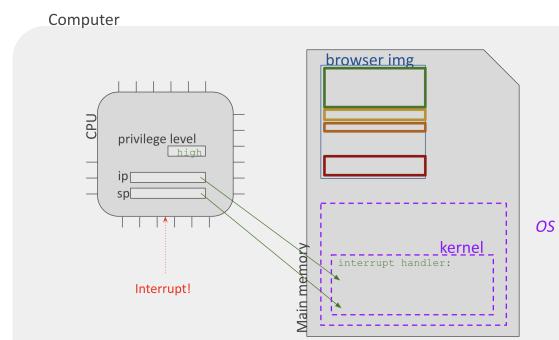
*What happens if the browser executes unauthorized code or encounters an error, such as dividing by zero ?*

When a process executes an illegal operation, the CPU raises an exception. The kernel then takes over to handle the error safely.



### Interrupt Handling

Interrupts are triggered by external events and are managed by both hardware and software. The hardware raises the interrupt, and the kernel (via an interrupt handler) processes it.

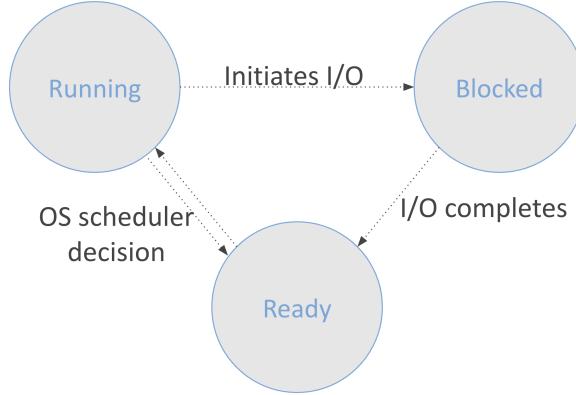


### The Timer Interrupt

**Definition (Timer Interrupt).** *The timer interrupt is raised at regular intervals (typically every few milliseconds). Its handler invokes the OS scheduler to decide which process runs next, ensuring that no process monopolizes the CPU.*

## The OS Scheduler

The OS scheduler manages the state transitions of processes (Running, Blocked, Ready) based on various scheduling algorithms, thereby ensuring equitable CPU access.



## Summary - Limited Direct Execution

- Normal threads execute in low privilege.
- Operations requiring high privilege are performed via syscalls, exceptions, or interrupts, which invoke the kernel.
- Timer interrupts ensure that the OS scheduler periodically gains control, maintaining fairness.

Limited direct execution is essential for safely and efficiently sharing the CPU among multiple processes.

## 3.4 Executing Syscalls — Process Management

Processes are created, modified, and terminated using various syscalls. We now discuss the key syscalls involved in process management.

### 3.4.1 Syscall Definitions

**Definition (Exit Syscall).** *The exit syscall terminates a process. It never returns because, by the time control would return to the calling process, the process no longer exists.*

```

1 _exit(0);
2 .
3 .
4 .
5 .
6 .
7 .
8 .
9 .
10 .
11 .
12 .

```

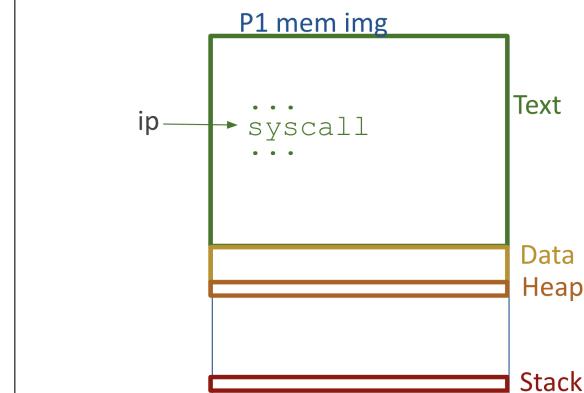


Figure 3.1: Exit Syscall: Code snippet and stack visualization.

**Definition (Exec Syscall).** The *exec* syscall replaces the current process image with a new program. It preserves the process ID and file descriptors while discarding the old program's code, data, and stack. On success, it does not return; on failure, it returns -1.

```

1 execvp("date", args);
2
3
4
5
6
7
8
9
10
11
12
13

```

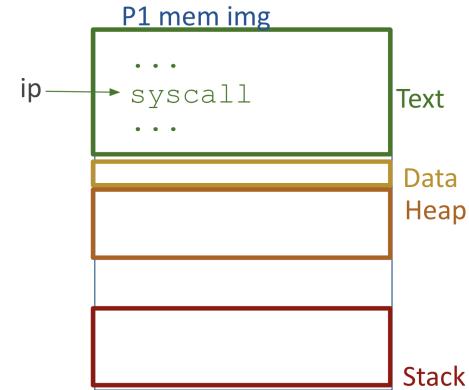


Figure 3.2: Exec Syscall: Code snippet and process image.

**Definition (Fork Syscall).** The *fork* syscall creates a new child process by duplicating the calling process. Both the parent and child continue execution from the point of the fork, but in separate memory spaces. The fork returns 0 to the child and the child's process ID (PID) to the parent.

```

1 int fs = fork();
2 if (fs == 0) {
3     // Child process code
4 } else {
5     // Parent process code
6 }
7
8
9
10
11

```

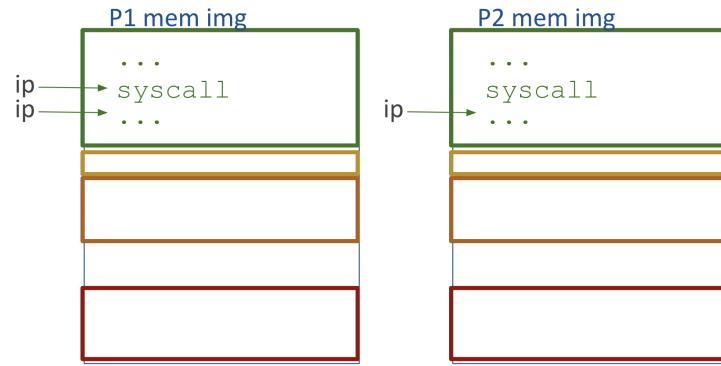


Figure 3.3: Fork Syscall: Example code and the resulting stack layout.

**Definition (Wait Syscall).** The *wait* syscall allows a parent process to block until one of its child processes terminates. If no child process exists, *wait()* returns an error.

### 3.4.2 Process Creation and Cleanup

Processes are typically created by combining the *fork* and *exec* syscalls. For example:

```

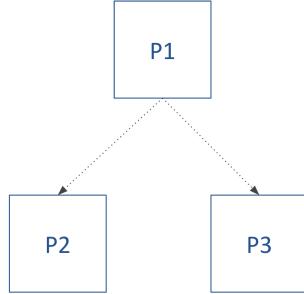
1 int fs = fork();
2 if (fs == 0) {
3     execvp("date", args);
4 } else {
5     wait(fs);
6 }

```

When a parent process calls *wait()*, it is blocked until a child terminates, ensuring proper cleanup of child processes.

### 3.5 The OS Process Graph

The OS maintains a process graph where each square represents a process and each arrow indicates the parent-child relationship. These kind of graphs are crucial for understanding process creation and hierarchy.



### 3.6 Key Processes in the OS

Some critical processes managed by the OS include:

- **GUI Processes:** Manage the graphical user interface.
- **Terminal Processes:** Handle command-line interactions.
- **Init Process:** The first process created by the kernel, responsible for starting system services.
- **Idle Process:** Executes when no other process is runnable.

### Conclusion - The Role of Syscalls

Syscalls provide the interface through which processes access system resources such as storage and networks. They also facilitate self-management operations, including process creation, modification, and cleanup. Through mechanisms such as limited direct execution, exceptions, and interrupts, the OS ensures that the CPU is shared safely and efficiently among all processes.