Computer Systems

IN BA4

Notes by Ali EL AZDI

September 11, 2024

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

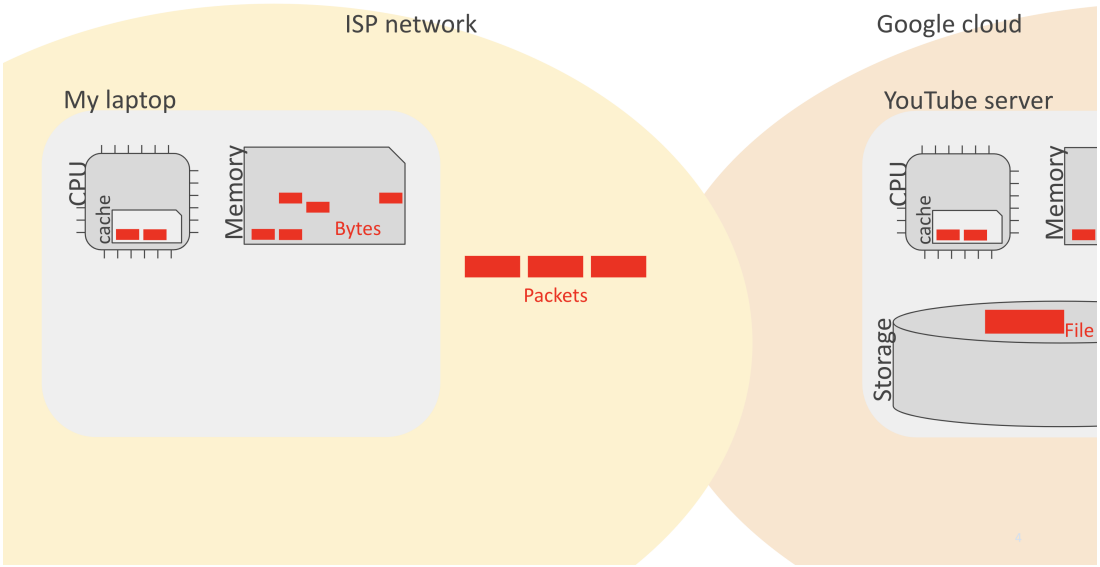
Chapter 1

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

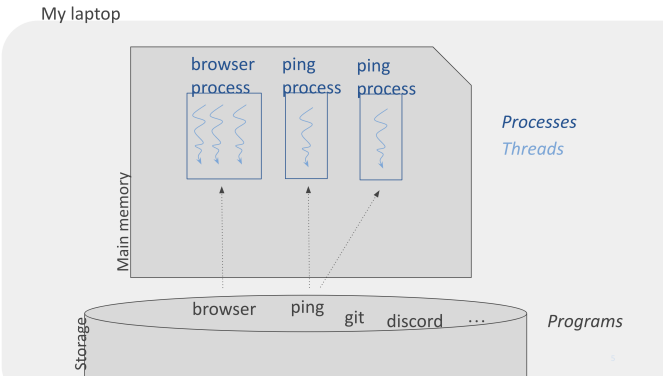
To illustrate these ideas, consider the journey of a YouTube video. The video begins its existence as a file stored on a storage device, is loaded into memory as bytes, transmitted as packets over the Internet, and finally rendered on your screen.



1.1.1 Start of the Journey: Inside the Laptop

The journey begins on your laptop.

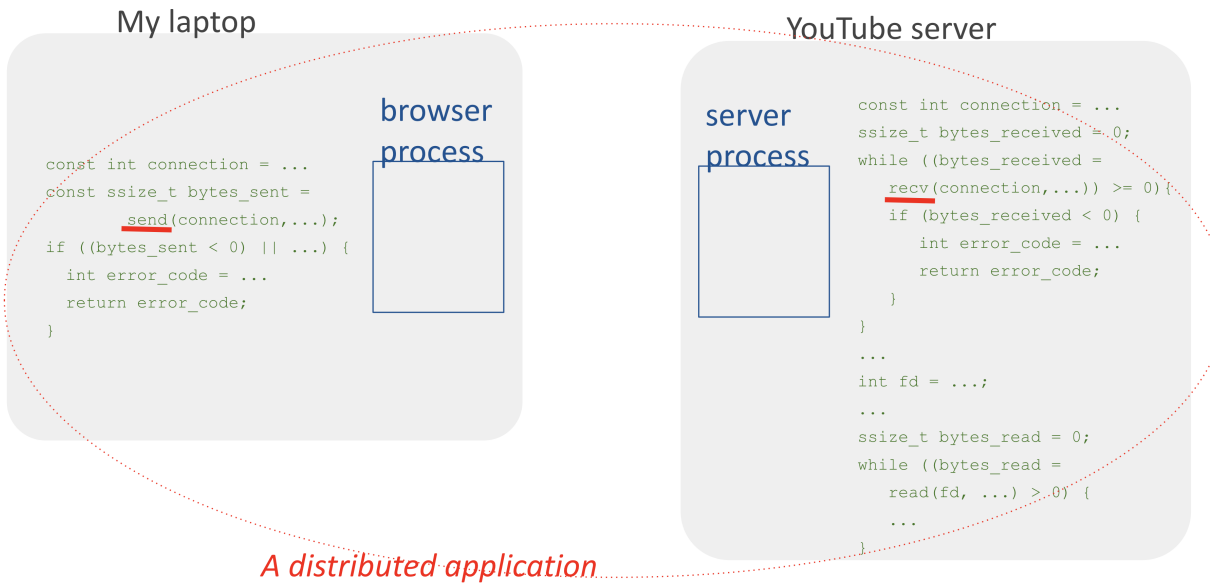
A computer hosts many different programs (e.g., a web browser, a ping utility, a git client). These programs, stored as files on disk, are invoked by user actions such as clicking an icon or typing a command. When a program is invoked, the computer creates a new *process* in main memory. A process represents 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*.

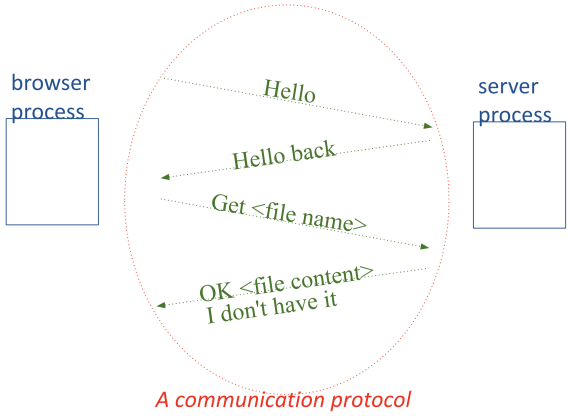


1.1.3 Communication Protocols

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:

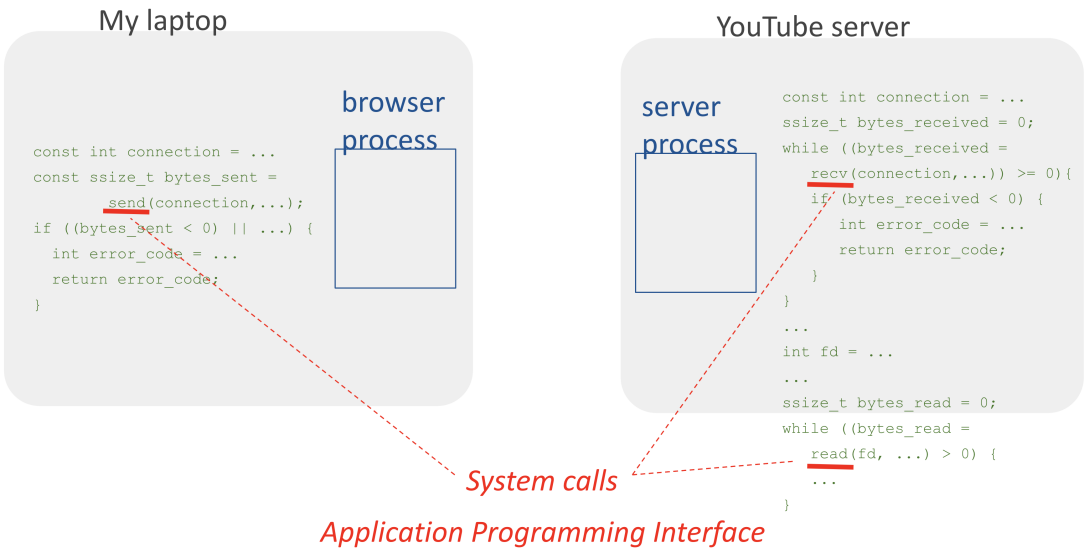
- One process sending "hello" and waiting for a "hello back."
- A subsequent request for a specific file (e.g., `xyz`) with the server responding with the file or an error message.

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



1.1.4 Distributed Applications and APIs

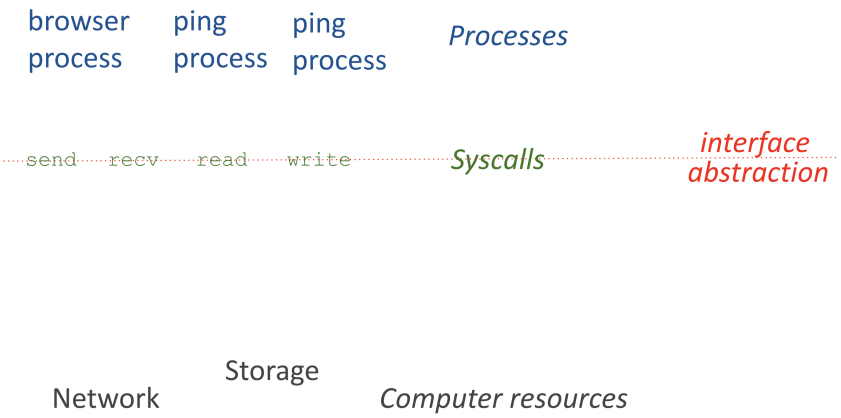
Distributed applications consist of separate pieces of code running as processes on different machines but working toward a common goal. These processes exchange messages over the Internet by following communication protocols. To simplify the development of these applications, developers use *system 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.*

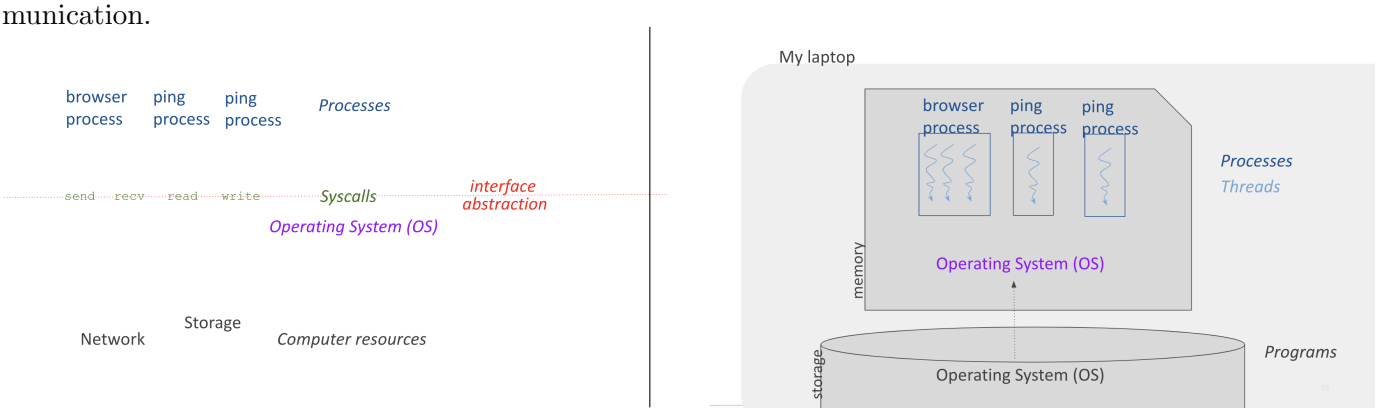
1.1.5 System Calls (Syscalls)

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.



1.2 The Operating System

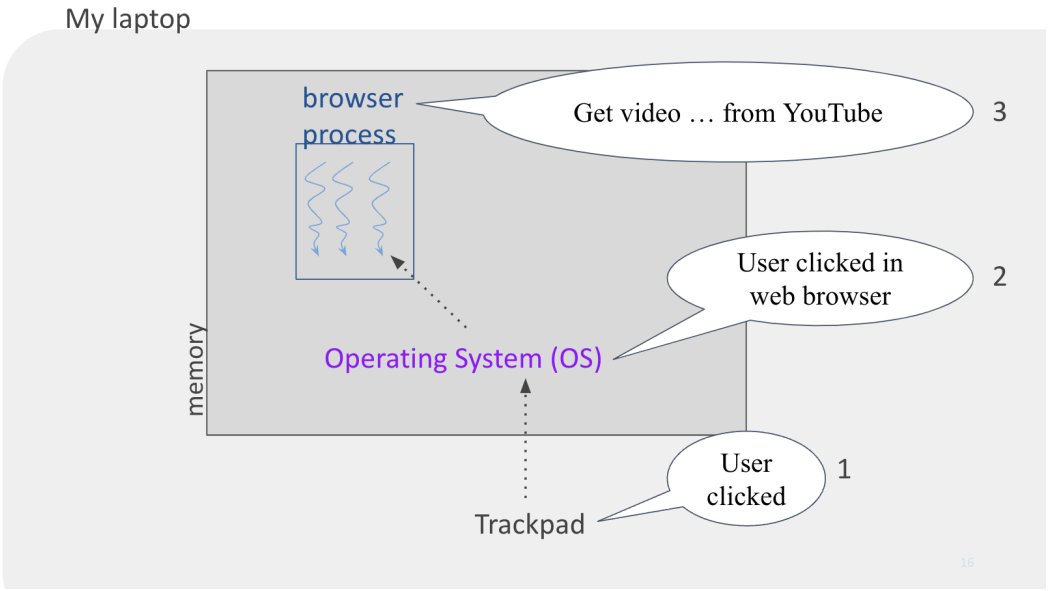
Conceptually, the OS sits between running processes and the underlying hardware resources. It provides the syscall interface and handles tasks such as file system management and network communication.



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.

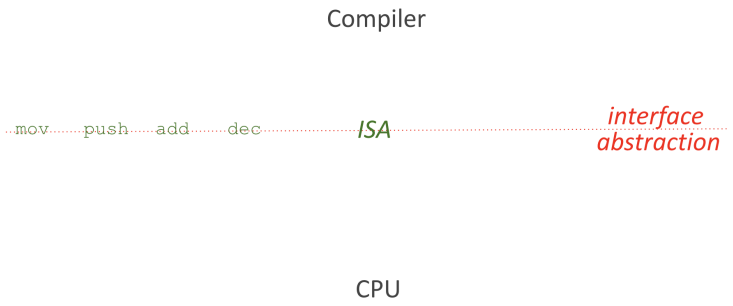


1.3 Program and ISA

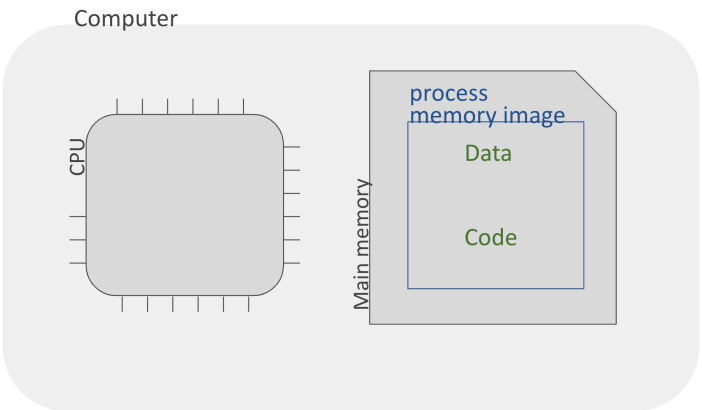
1.3.1 Program

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).

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.*

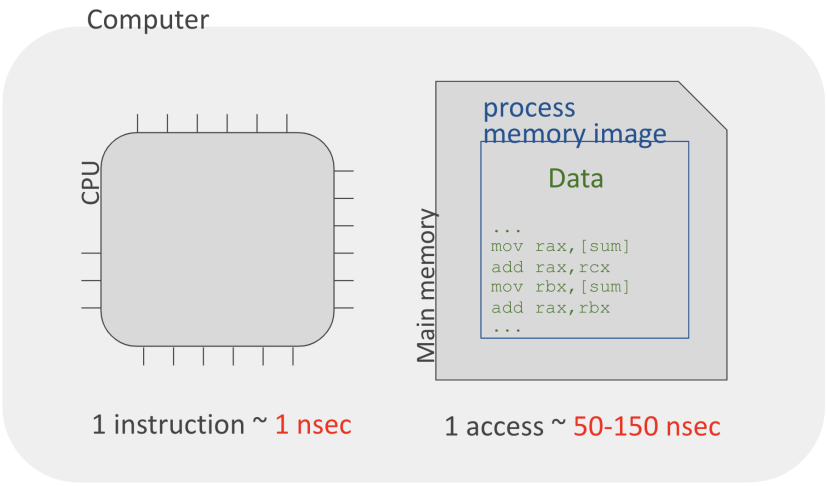
Question: *What is the meaning of the Hz metric?*

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.

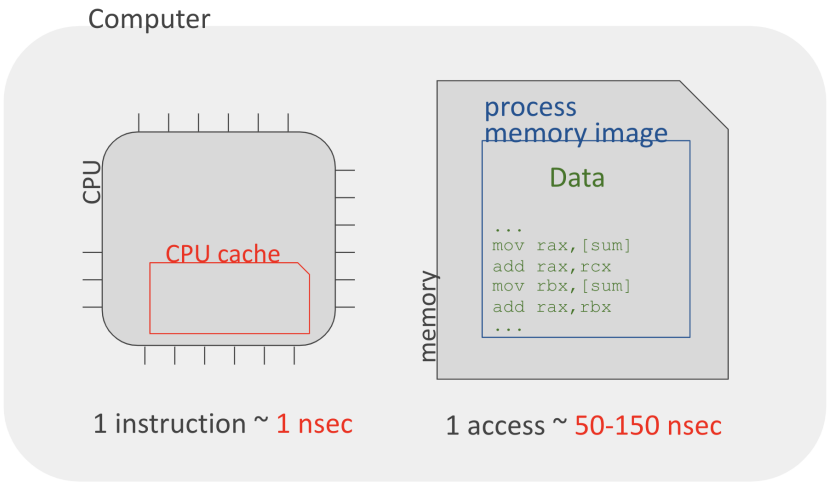


Question: How can we improve the efficiency of a system with such a frequency imbalance?

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.

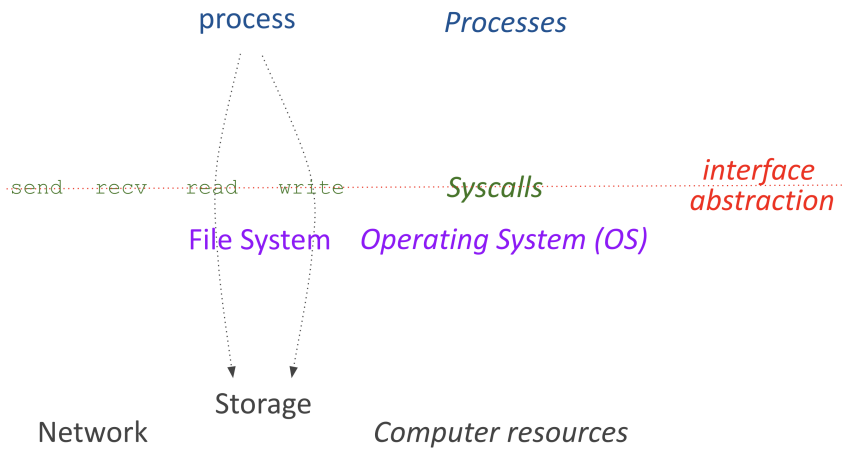
The browser C code → Compiler → *The browser executable*

```
...
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.*

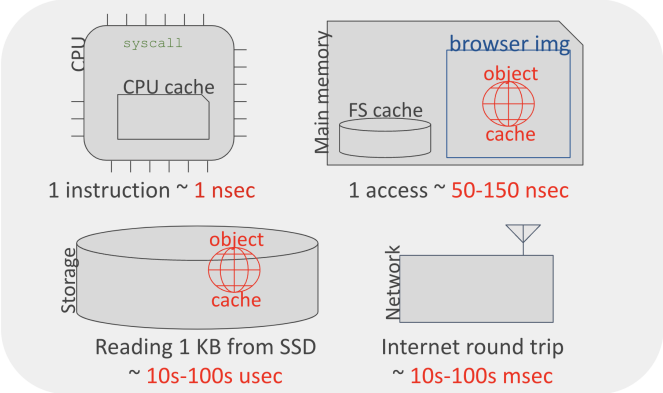
Exam Question: If a program does not create or manipulate any data, will executing this program require reading anything from memory?

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:

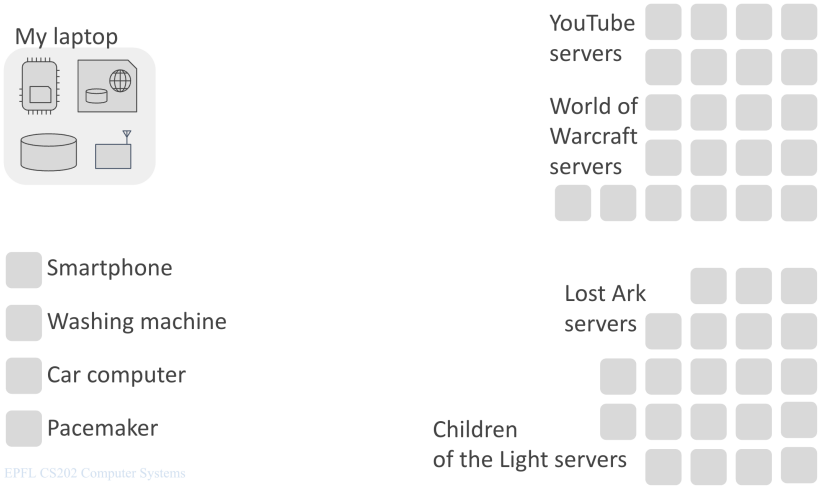
- Simple CPU instructions: ∼1 nsec.
- Main memory accesses: tens to hundreds of nanoseconds.
- Reading 1 KB from an SSD: tens to hundreds of microseconds.
- Requesting data over the Internet: several to hundreds of milliseconds.



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.

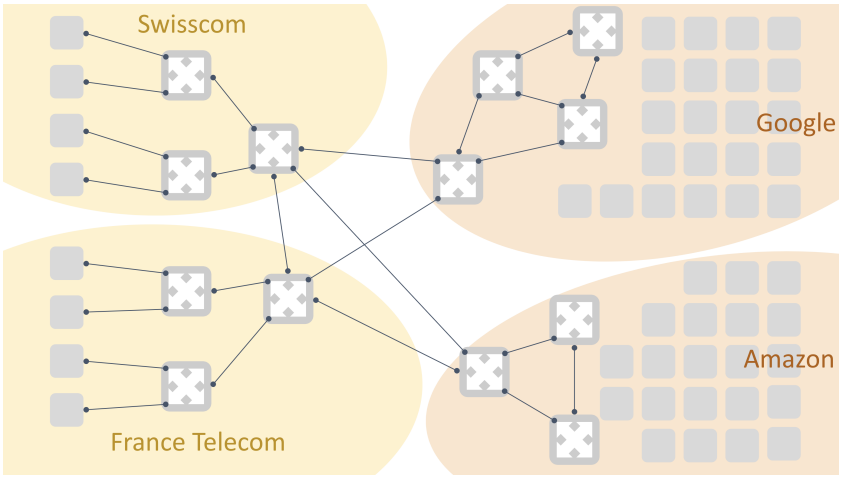


1.6.2 Packet Switches and Network Links

In addition to end systems, the Internet relies on:

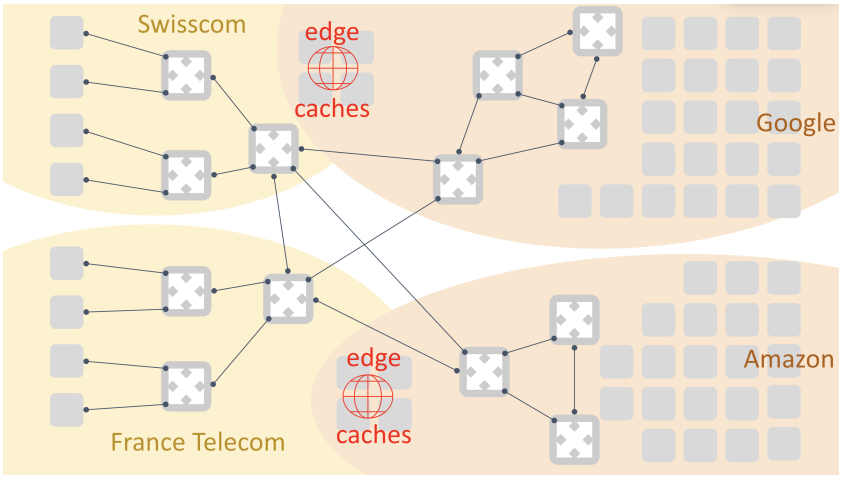
- **Packet Switches**: Devices that route data between end systems.
- **Network Links**: Physical connections that interconnect packet switches and end systems.

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.

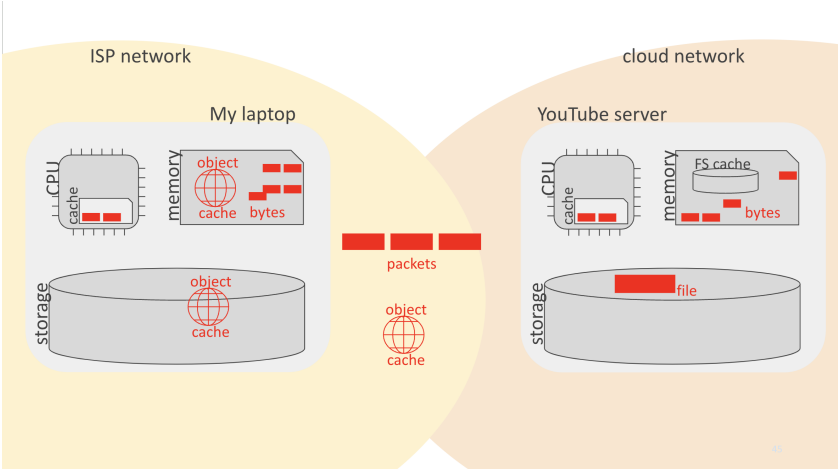


1.7 Summary

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).



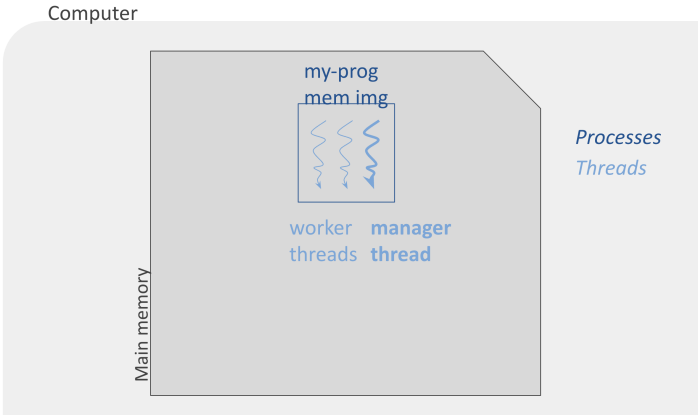
Chapter 2

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

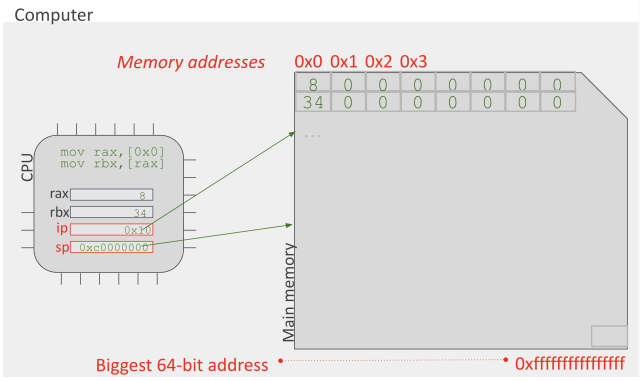
When a program runs, the processor loads it into main memory and creates a thread. In a multithreaded program, there is typically one *manager* thread that delegates work to several *worker* threads. For instance, when computing the sum of a large set of numbers, the workload can be divided into subsets,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

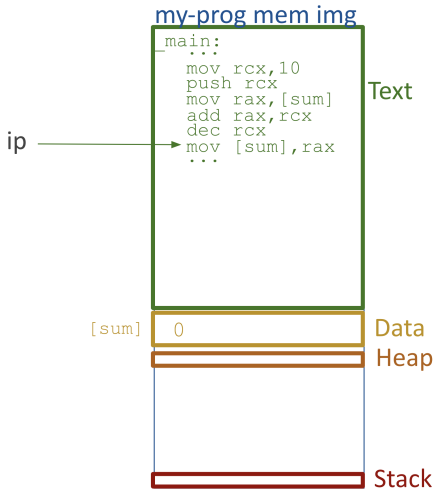
2.3.1 The Stack

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.

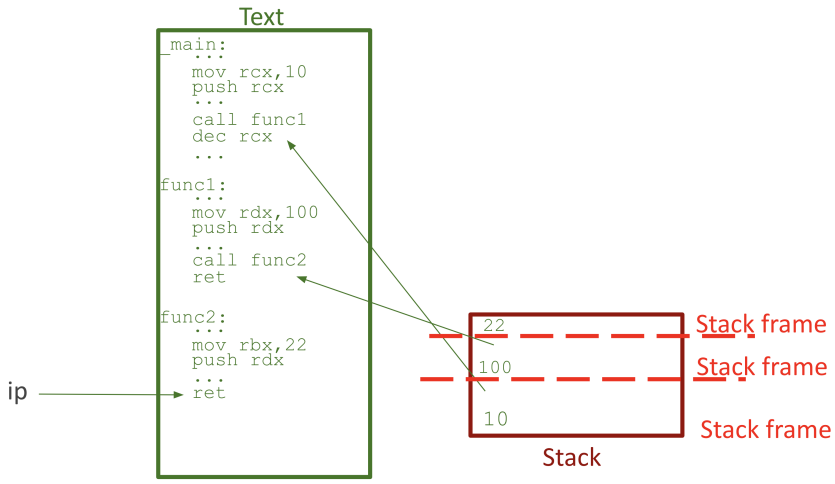
Key Characteristics of the Stack:

- **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.*

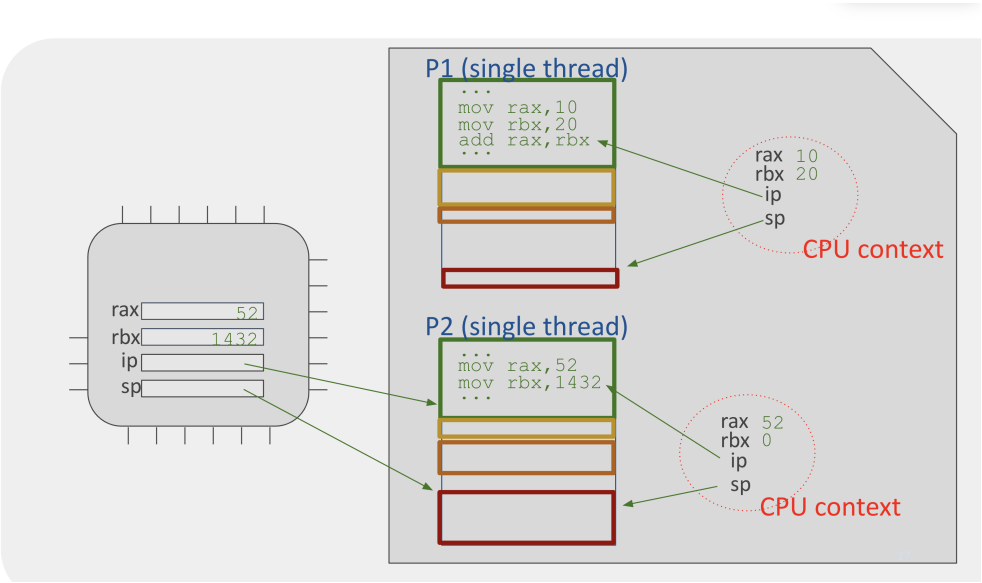
- ***Process ID (PID):*** *A unique identifier assigned to each process.*
- ***Thread ID (TID):*** *A unique identifier for each thread, which may be unique within a process or across the entire system, depending on the operating system.*

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` $\rightarrow$ *Physical* `0x1234AFF8`
- *Process* P_2*: Virtual* `0x400000` $\rightarrow$ *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

L3 - 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 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.

3.1.1 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.

The CPU runs in Limited Direct Execution. Let's see how it's limited···

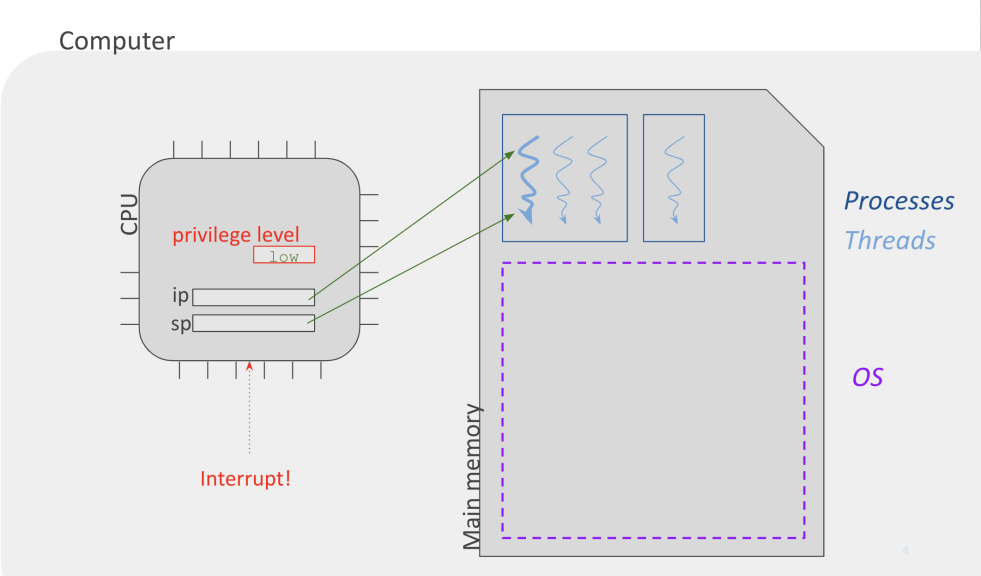
3.1.2 CPU Privilege Levels and Execution Modes

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.

Personal Remark - Kernal code always *run in High Privilege mode, and because of this we say that the OS* may *run in High Privilege mode (kernel is inside of the os), do not confuse both !*

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** (often called 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.

3.1.3 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*.

3.1.4 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.

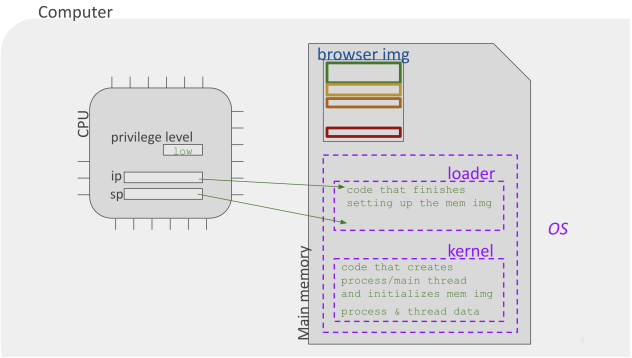
- By switching between kernel mode and user mode, the OS ensures that user applications run securely and do not directly manipulate hardware resources.

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.

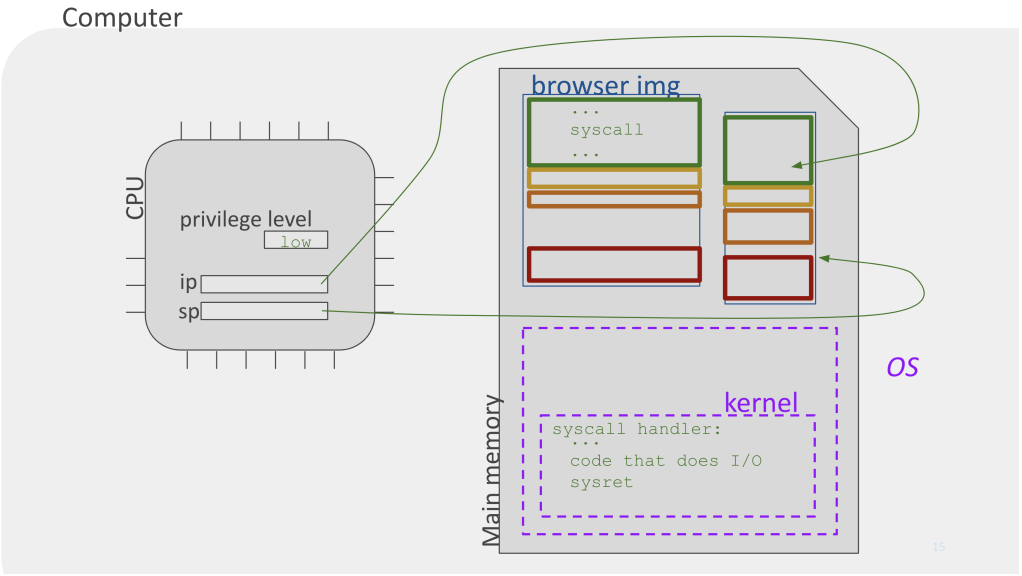


This layered privilege model ensures a secure and efficient execution environment, maintaining stability and protection across all processes within the system.

3.1.5 Syscalls

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

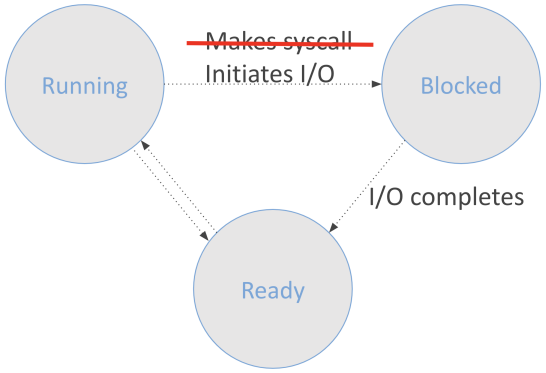
1. The CPU temporarily elevates the privilege level.
2. The control is transferred to the OS to execute the privileged code.
3. If the syscall involves an I/O operation, the process may be blocked while waiting for a response, and the CPU may perform a context switch to another thread.
4. Once the operation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



3.1.6 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.



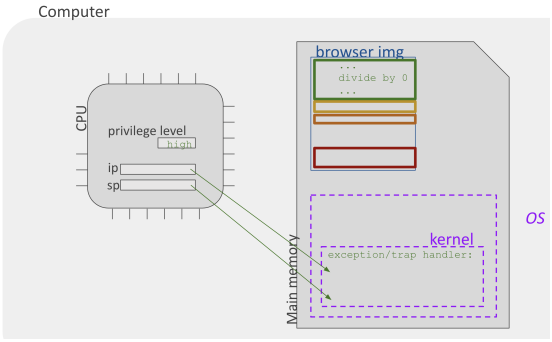
3.2 The Kernel's Job cont.

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.

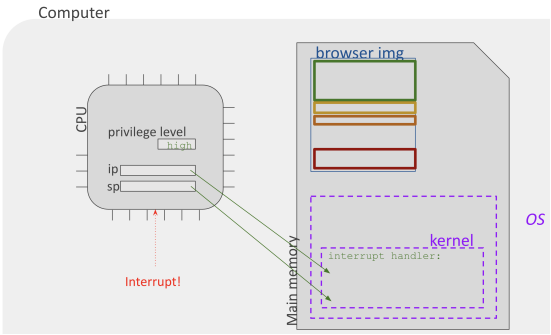
Exception Handling

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.

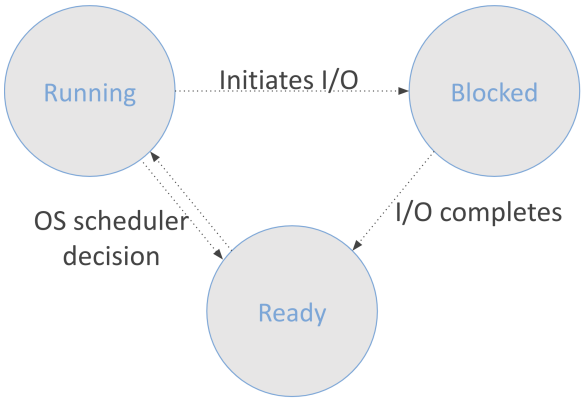


The Timer Interrupt

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.

3.3 Executing Syscalls — Process Management

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

3.3.1 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 .
```

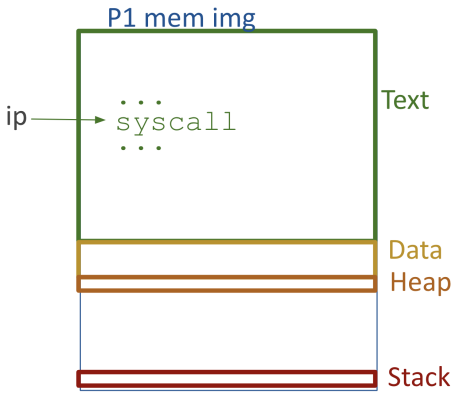


Figure 3.1: Exit Syscall: Code snippet and stack visualization.

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`.*

```
execvp("date", args);
```

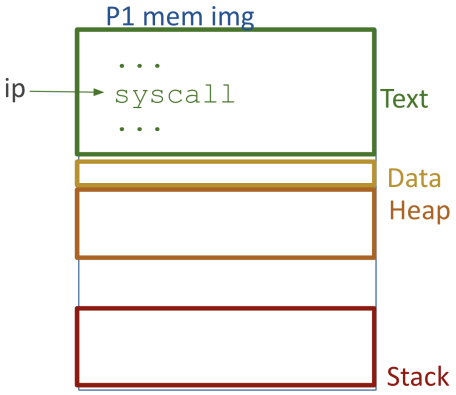


Figure 3.2: Exec Syscall: Code snippet and process image.

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.*

```
int fs = fork();
if (fs == 0) {
   // Child process code
} else {
   // Parent process code
}
```

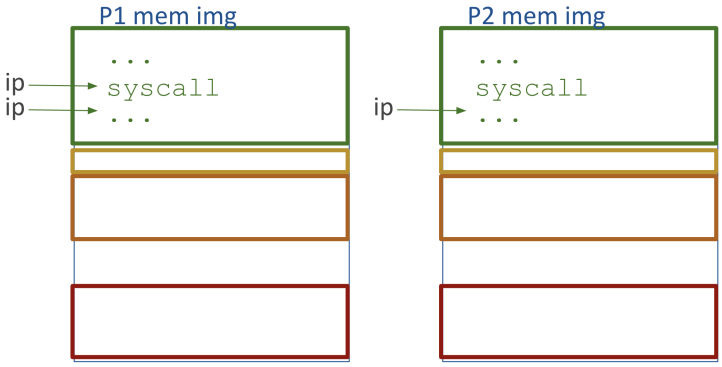


Figure 3.3: Fork Syscall: Example code and the resulting stack layout.

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.*

3.3.2 Process Creation and Cleanup

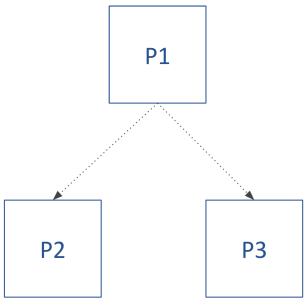
Processes are typically created by combining the `fork` and `exec` syscalls. For example:

```
int fs = fork();
if (fs == 0) {
   execvp("date", args);
} else {
   wait(fs);
}
```

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

3.4 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.



3.5 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.

Chapter 4

L4 - Memory

This chapter covers the fundamentals of main memory, process memory images, memory virtualization, and the CPU's role in managing memory.

4.1 Main Memory

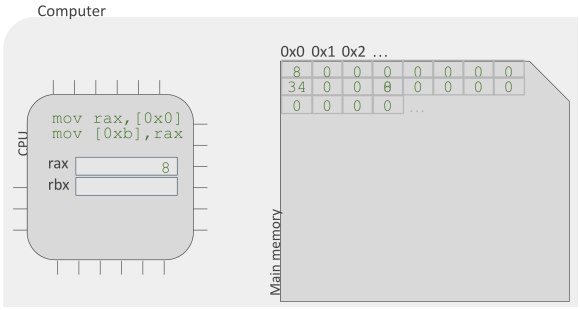
Main memory is conceptualized as a linear array of bytes, where each byte has a unique memory address (e.g., `0x0`, `0x1`, `0x2`, etc.). Each byte can store any value that fits within its 8-bit capacity, and importantly, the value stored in a given byte is independent of its memory address. For instance, the byte at address `0x0` may contain the value 8, 0, or any other valid 8-bit number.

4.1.1 Memory Operations by the CPU

The CPU interacts with main memory by executing specific instructions to read from and write to it. These operations are fundamental to both data processing and code execution:

- **Read Operation:** The CPU issues an instruction to read a block of bytes (for example, 8 bytes starting at address `0x0`) and loads the result into a register (such as `rax`).
- **Write Operation:** The CPU executes an instruction that writes data from a register (e.g., `rax`) into a block of memory (for example, starting at address `0xb`).

Although main memory stores only numbers, the CPU interprets these numbers differently depending on whether they represent data (such as variables) or executable code (such as the instruction `mov rax, [0x0]`).



4.1.2 Instruction Pointer

A key component in the CPU's control mechanism is the *instruction pointer* (IP), in some contexts (ie. FDS, Comparch), this register is also known as the *program counter* (PC), but the term "instruction pointer" more precisely describes its function.

Definition (Instruction Pointer).

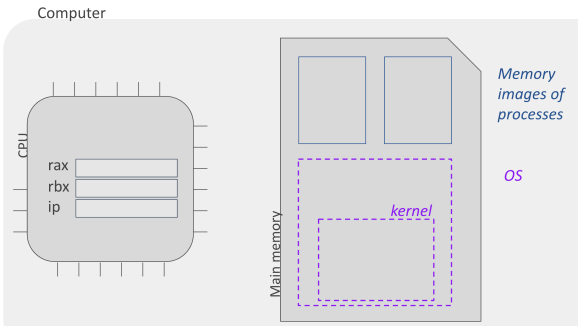
*The **instruction pointer** is a CPU register that holds the memory address of the next instruction to be executed.*

4.1.3 Subparts of Main Memory

Main memory contains not only the memory images of individual processes but also the code and data essential to the operating system (OS). The OS comprises several critical components that ensure the proper operation and usability of the computer. These components include:

- **Process Memory Images:** Every process has its own memory image, typically divided into:
  - **Data Segment:** Stores global variables.
  - **Stack Segment:** Contains local variables, return addresses, and other function call-related data.
  - **Heap Segment:** Holds dynamically allocated memory (e.g., allocated via `malloc`).

- **Operating System Code:** This comprises all the code necessary for the computer's operation and usability. OS code is organized into:
  - *Kernel:* The central component of the OS, running in high-privilege mode. It manages system resources, hardware interactions, and security, ensuring the core functions of the computer operate correctly. It is neither a process nor a library (end of lecture explains).
    - It creates and deletes processes and threads.
    - It initiates I/O.
    - It handles errors and interrupts.
    - It decides which thread will run next.
  - *Processes:* Such as the graphical user interface (GUI) and terminal applications, which provide user-level interaction with the system.
  - *Libraries:* Modules like the standard C library that provide a suite of functions, which are dynamically integrated into processes when called.



4.2 Process Memory Image

Definition (Process Memory Image).

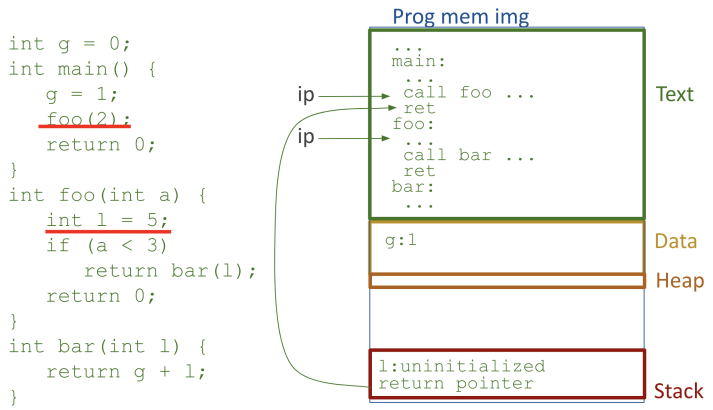
*A **process memory image** is the complete layout of a process's memory, comprising:*

- *The* text segment *for the process's code.*
- *The* data segment *for global variables.*
- *The* stack segment *for local variables and return pointers.*
- *The* heap segment *for dynamically allocated memory. (eg. malloc)*

Exam Question: We provide you with a C program. Your task is to draw the memory image of the corresponding process at different points in the program.

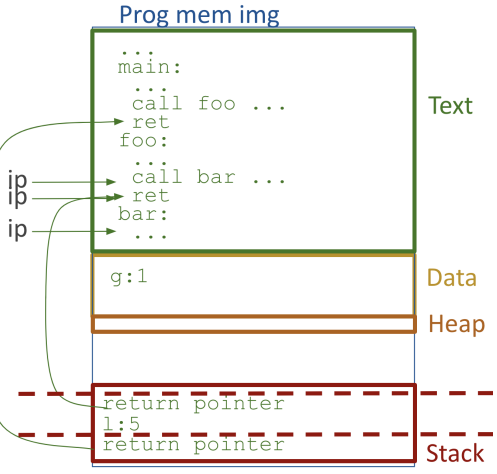
1. Mark each segment, even if it is empty.
2. Draw a schema of the code in the **Text Segment**, including only function names and calls in assembly.
3. Identify global variables and place them in the **Data Segment** (e.g., `g:0`).
4. Simulate each step of the program's execution to fill the **Heap** and **Stack** segments accordingly. This includes local variables, memory allocations, and function calls.

```
int g = 0;
int main() {
   g = 1;
   foo(2);
   return 0;
}
int foo(int a) {
   int l = 5;
   if (a < 3)
      return bar(l);
   return 0;
}
int bar(int l) {
   return g + l;
}
```



Exam Question: Show the memory image and indicate the moment when the stack reaches its maximum size.
For the program shown above, the expected result would be:

```
int g = 0;
int main() {
   g = 1;
   foo(2);
   return 0;
}
int foo(int a) {
   int l = 5;
   if (a < 3)
      return bar(l);
   return 0;
}
int bar(int l) {
   return g + l;
}
```



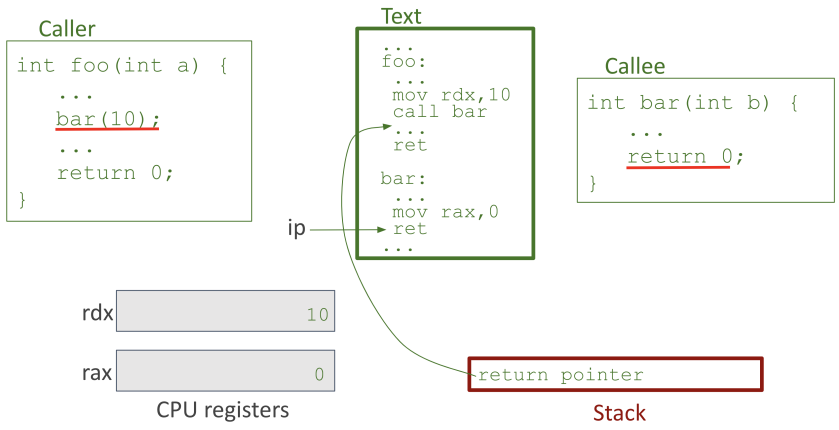
4.2.1 Optional - Stack and Register Functioning

In modern computer architectures, the process of a function call involves a coordinated interplay between CPU registers and the stack.

Definition (Function Call Mechanism).

During a function call:

1. *The **caller** passes arguments to the **callee** by storing values in designated CPU registers.*
2. *The **callee** processes the call and returns a result by placing it in a specific register (for example, the `rax` register).*



Example 4.2.1.1 (Illustrative Function Call)**.** *Consider the scenario where function `Foo` calls function `Bar`:*

1. ***Argument Passing:** `Foo` stores the argument value (e.g., **10**) in a CPU register.*
2. ***Return Value Handling:** An implicit agreement between foo and bar that the return value will be stored in a particular register (eg. rax). `Bar` processes the argument and stores its return value in another register (e.g., `rax`). Later, `Foo` retrieves this value by accessing that register.*

A caller and a callee share common infrastructure by using CPU registers to maintain their context during the call. In addition, the stack is used to preserve register states when necessary:

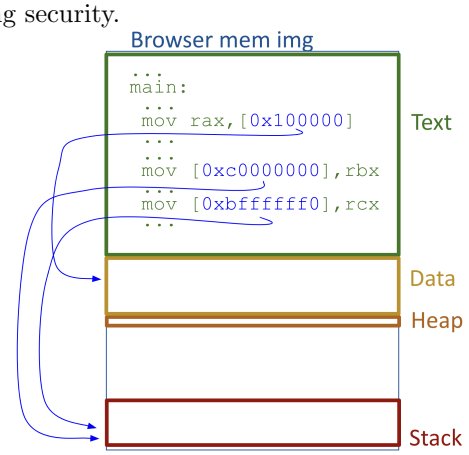
- **Caller-Saved Registers:** The caller saves certain registers to the stack before the call and restores them after the call returns.
- **Callee-Saved Registers:** The callee saves its registers at the beginning of the function and restores them before returning.

The stack, the register the calling conventions form a caller/callee interface.
Adhering to these calling conventions is critical; for instance, if the callee writes into the caller's stack frame, it may lead to stack smashing and compromise program stability.

4.3 Memory Virtualization

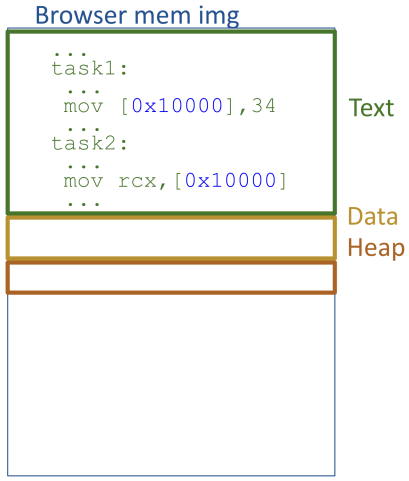
In modern operating systems, each process references memory using virtual addresses. Underneath, these virtual addresses are translated to physical addresses. Importantly, each process has its own virtual address space, which means that two processes may use the same virtual address while referring to entirely different physical locations. This design creates the **safe illusion** that main memory "belongs" exclusively to each process, greatly simplifying program development and enhancing security.

Each address in the image refers to an address within its own stack. The addresses shown are *virtual*, meaning they are process-specific (e.g., address 0 in one process does not necessarily correspond to address 0 in another).



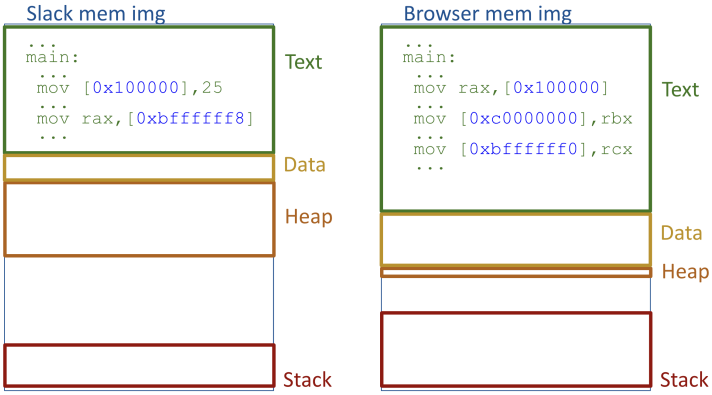
Exam Question: If two memory instructions read the same virtual address, is it the same physical address?

Answer: Yes, when they are translated to the same physical memory address, as they belong to the same virtual address space.



Exam Question: Are these two processes accessing the same memory location?

Answer: No, they are not actually accessing the same physical memory. Although both use the address 0x10000, each process runs in its own virtual address space.



The mechanism of memory virtualization creates a *safe illusion* in which it appears that the main memory is exclusively owned by each process. This design not only simplifies the generation of executable programs but also enforces security by ensuring that a process can only access its own memory image.

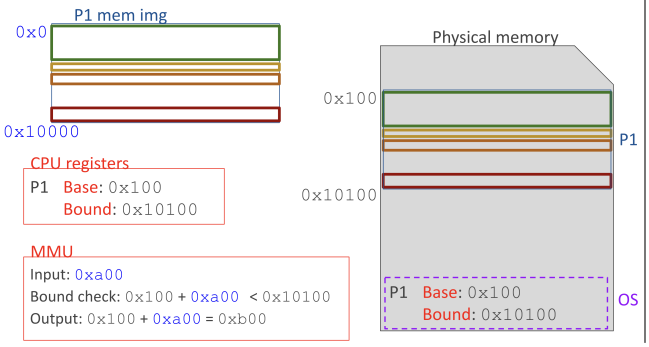
Definition (Contiguous Memory).

Contiguous memory refers to a block of physical memory addresses that are sequentially arranged. In this allocation scheme, the entire memory image of a process is stored in one unbroken segment, simplifying the translation from virtual to physical addresses.

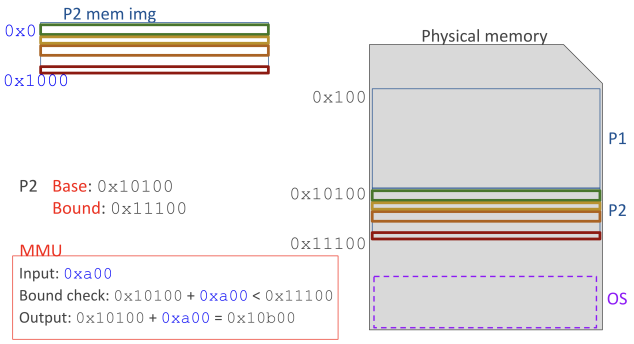
4.3.1 Memory Management Unit — Simple Implementation

The **Memory Management Unit (MMU)** is a specialized piece of *hardware* that translates virtual memory addresses into physical addresses.

For each process, the **OS kernel** sets up *base* and *bound* registers (which are physically stored in the CPU). The MMU then uses these values to ensure that the process's memory image is allocated in a contiguous block of physical memory.



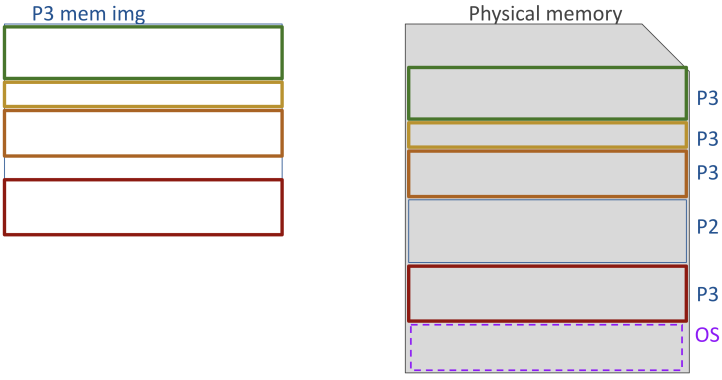
When a second process (P_2) is introduced, the MMU checks its corresponding base and bound registers to determine the physical memory range in which P_2 should be placed.



In this **base–bound** scheme, each process's memory image starts at its base address and extends just before its bound address. This approach is *safe* (preventing a process from accessing memory outside its allocation) and preserves the *illusion* of owning the entire memory.

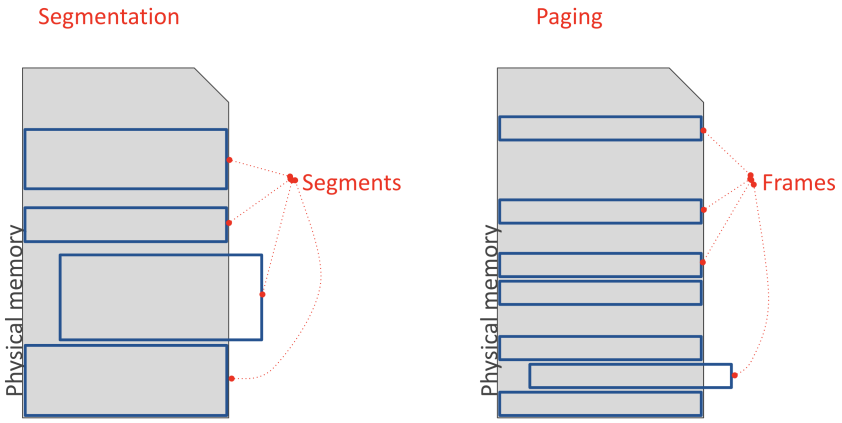
However, because each process must reside in one contiguous memory block, **fragmentation** can occur.

For example, when process P_1 terminates, it might leave a gap that is too small for a new process P_3, even if the total available memory is sufficient.

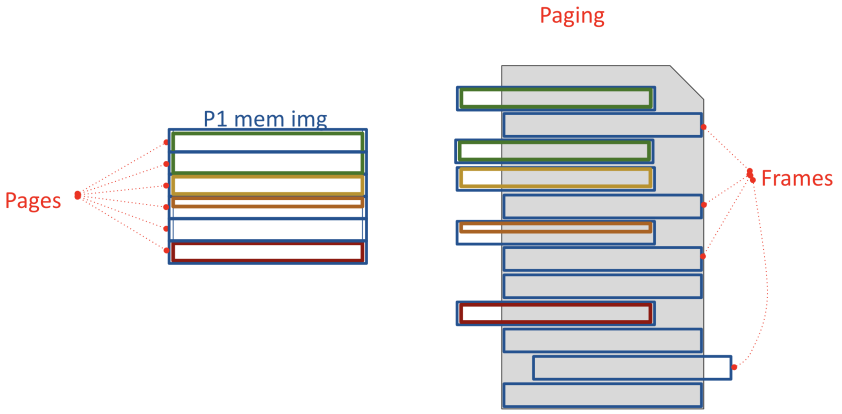


To address the inefficiency of requiring each process to occupy a single contiguous block of memory, an effective strategy is to *divide the process's address space into smaller chunks*, allowing noncontiguous allocation. Two primary techniques for accomplishing this are:

- **Paging:** The address space is split into *fixed-size pages*, which map onto equally sized *physical frames*. This approach can reduce external fragmentation but can introduce *internal fragmentation* if a process does not fully use the last frame of its allocation. Paging is straightforward to manage and scales well for large address spaces.
- **Segmentation:** The address space is divided into *variable-sized segments* (e.g., code, data, stack). This fits naturally with the logical structure of programs and can minimize *internal* waste; however, it can result in *external* fragmentation when segments cannot fit into available gaps in physical memory.



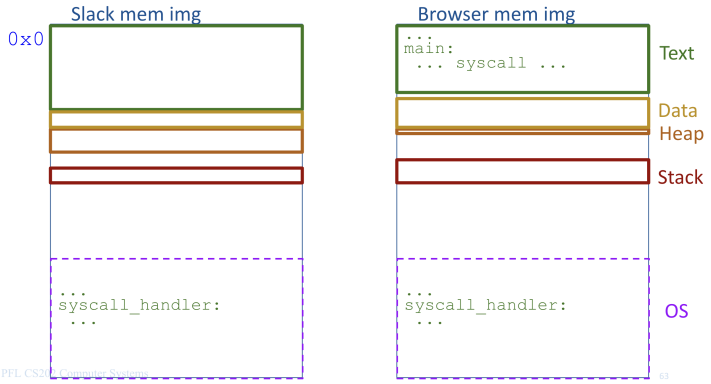
In **paging**, the MMU maintains a *page table* to translate from virtual pages to fixed-size physical frames:



In either scheme, the MMU—*configured* by the kernel with base, bound, or other address-translation structures—ensures each process can access only the memory it has been allocated. This *hardware-based* translation mechanism preserves system safety and helps improve physical memory utilization by allowing noncontiguous allocation.

4.4 Optional - Operating System Mapping in Process Memory

In modern operating systems, the OS is mapped into every process's virtual address space. This design allows a process to make system calls efficiently, as the CPU switches to pre-mapped high-address instructions during such transitions. This integration supports secure and fast interactions between user applications and system-level functions.

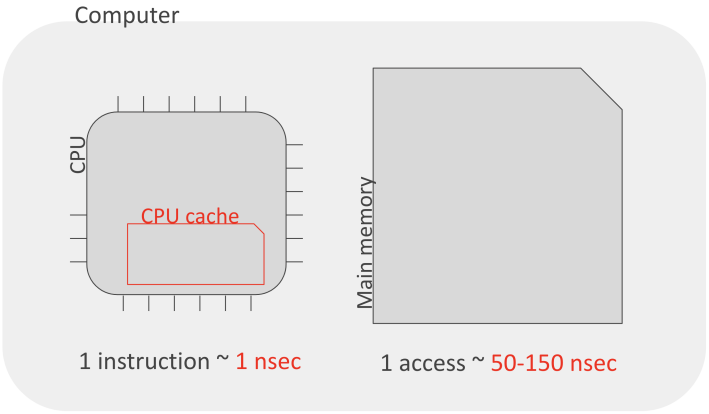


4.5 CPU Caching and Memory Hierarchy

Efficient computation in modern CPUs relies on a well-designed memory hierarchy that mitigates the performance gap between the processor and main memory. Central to this hierarchy is the CPU cache, which stores recently and frequently accessed data.

4.5.1 Overview of CPU Cache

The CPU cache is a small, high-speed memory located close to the processor core. It temporarily holds data and instructions that the CPU is likely to reuse, significantly reducing the latency compared to fetching data from main memory. This approach minimizes delays due to the slower speed of main memory and ensures smoother processor performance.

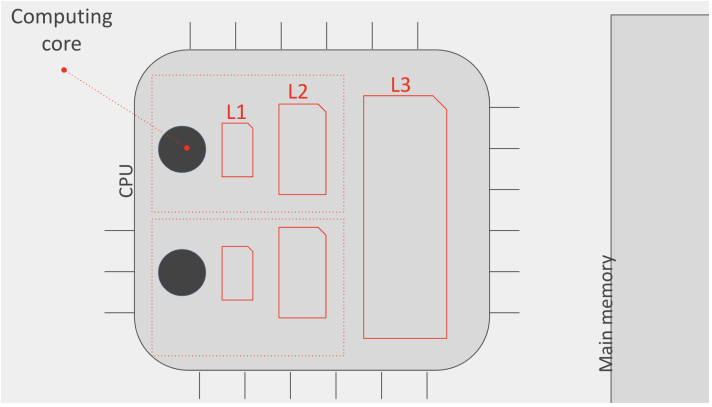


4.5.2 Multi-Level Cache Architecture

Modern CPUs employ a multi-level cache system to balance speed and storage capacity:
Definition (Cache Levels). *The cache hierarchy typically consists of:*

- L_1 ***Cache:*** *The smallest and fastest cache, often divided into separate instruction and data caches.*
- L_2 ***Cache:*** *Larger than L_1 and slightly slower, serving as an intermediary between L_1 and L_3.*
- L_3 ***Cache:*** *The largest and slowest cache, usually shared among multiple cores in multi-core processors.*

The arrangement from smaller and faster (L_1) to larger and slower (L_3) reflects a deliberate trade-off between speed and capacity.



4.5.3 Cache Organization in Multi-Core Processors

Today's processors often include multiple computing cores, each with dedicated L_1 and L_2 caches while sharing a common L_3 cache. This design:

- Provides high-speed access to data for individual cores.
- Balances the overall workload by reducing contention for shared resources.

Without such a hierarchical system, a single cache (e.g., L_1) might evict infrequently used yet critical instructions, thereby degrading performance.

Example 4.5.3.1. *Consider a scenario in which a core with only an L_1 cache continuously evicts a seldom-used, but vital instruction. The presence of additional cache levels (L_2 and L_3) provides extra storage layers, ensuring that even infrequently accessed data remains available when needed.*

4.5.4 Summary of the Memory Hierarchy

The overall memory hierarchy in a modern CPU is structured as follows:

1. **L_1 Cache:** Fastest, smallest, with separate instruction and data caches.
2. **L_2 Cache:** Intermediate in both size and speed.
3. **L_3 Cache:** Largest, slowest, shared among cores.
4. **Main Memory:** Accessed only when data is not found in any cache.

The CPU always accesses the memory hierarchy starting at the fastest level (L_1) and moving downward, ensuring that processing is carried out as efficiently as possible.