Chapter 1

L4 - Memory

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

1.1 Main Memory

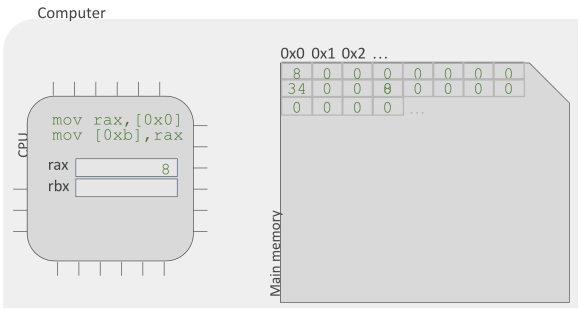
Main memory is conceptualized as a linear array of bytes, where each byte has a unique memory address (e.g., `0x0`, `0x1`, `0x2`, etc.). Each byte can store any value that fits within its 8-bit capacity, and importantly, the value stored in a given byte is independent of its memory address. For instance,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

1.1.1 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]`).



1.1.2 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).

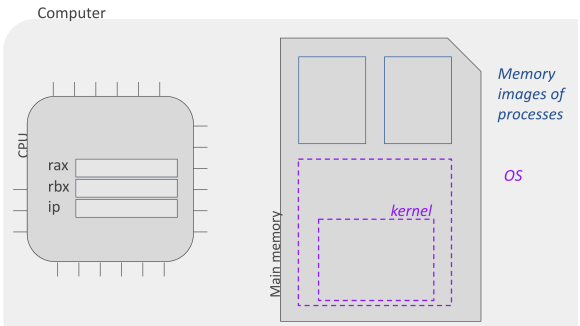
*The **instruction pointer** is a CPU register that holds the memory address of the next instruction to be executed.*

1.1.3 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.



1.2 Process Memory Image

Definition (Process Memory Image).

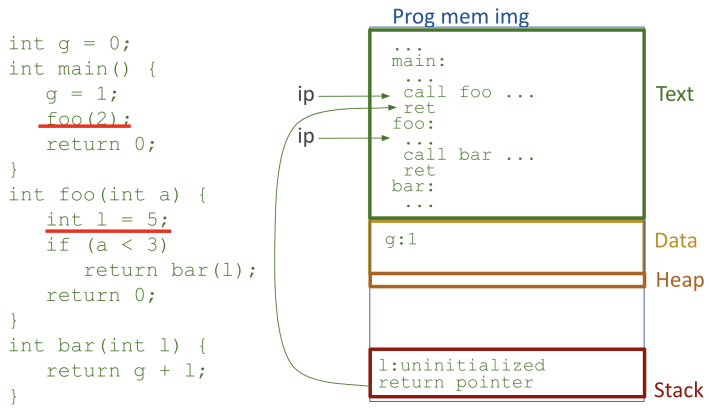
*A **process memory image** is the complete layout of a process's memory, comprising:*

- *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.

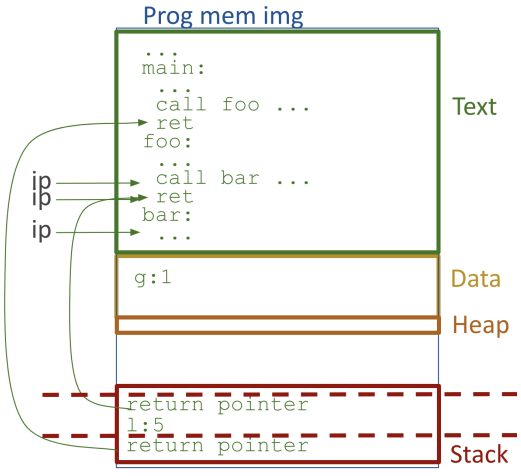
1. Mark each segment, even if it is empty.
2. Draw a schema of the code in the **Text Segment**, including only function names and calls in assembly.
3. Identify global variables and place them in the **Data Segment** (e.g., `g:0`).
4. Simulate each step of the program's execution to fill the **Heap**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



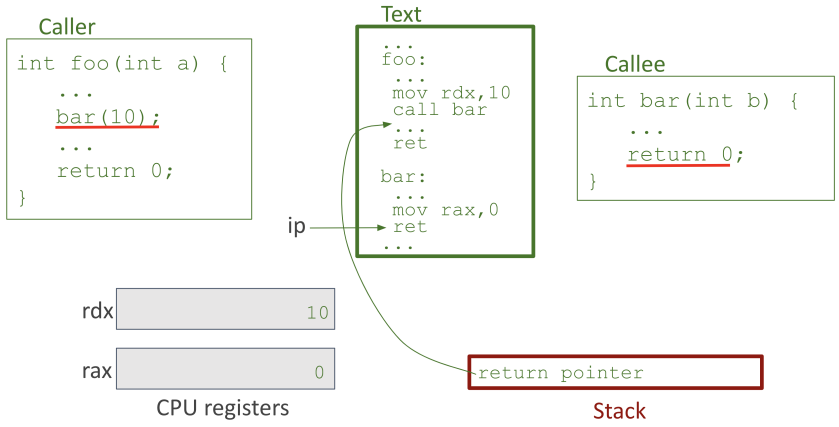
1.2.1 Optional - Stack and Register Functioning

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 1.2.1.1 (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:

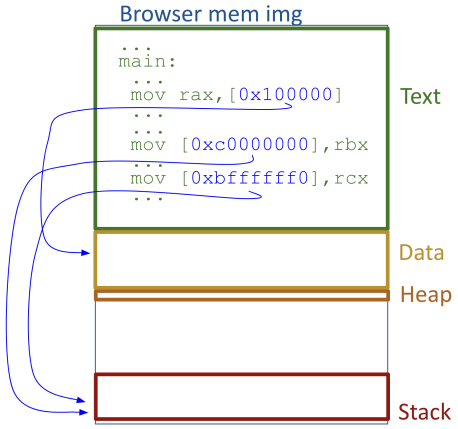
- **Caller-Saved Registers:** The caller saves certain registers to the stack before the call and restores them after the call returns.
- **Callee-Saved Registers:** The callee saves its registers at the beginning of the function and restores them before returning.

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.

1.3 Memory Virtualization

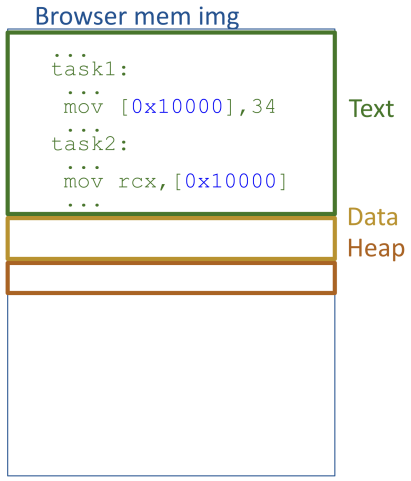
In modern operating systems, each process references memory using virtual addresses. Underneath, these virtual addresses are translated to physical addresses. Importantly, each process has its own virtual address space, which means that two processes may use the same virtual address while referring to 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).



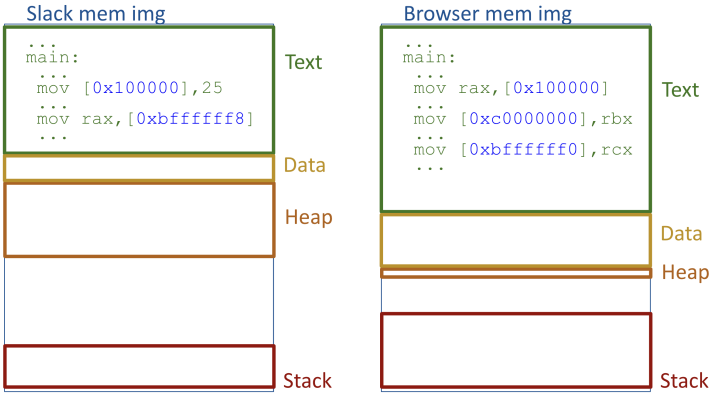
Exam Question: If two memory instructions (in the same process) read the same virtual address, is it the same physical address?

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.

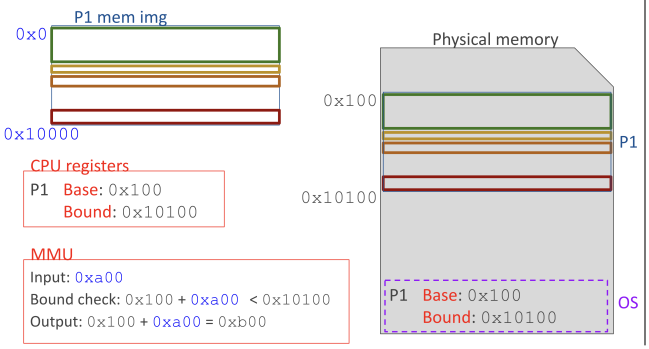
Definition (Contiguous Memory).

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.

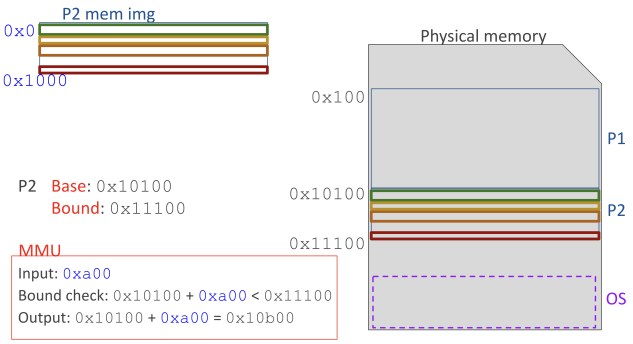
1.3.1 Memory Management Unit — Simple Implementation

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.



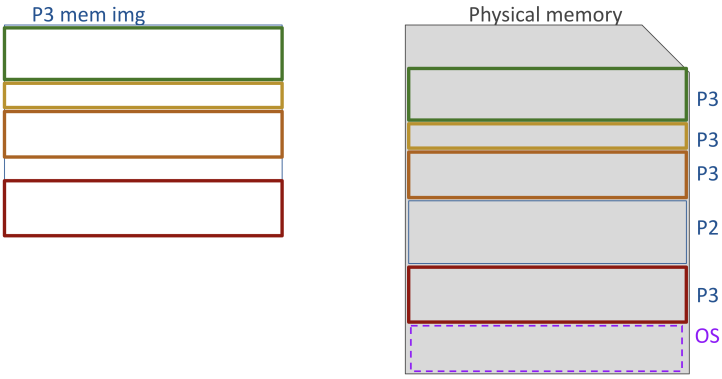
When a second process (P_2) is introduced, the MMU checks its corresponding base and bound registers to determine the physical memory range in which P_2 should be placed.



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.

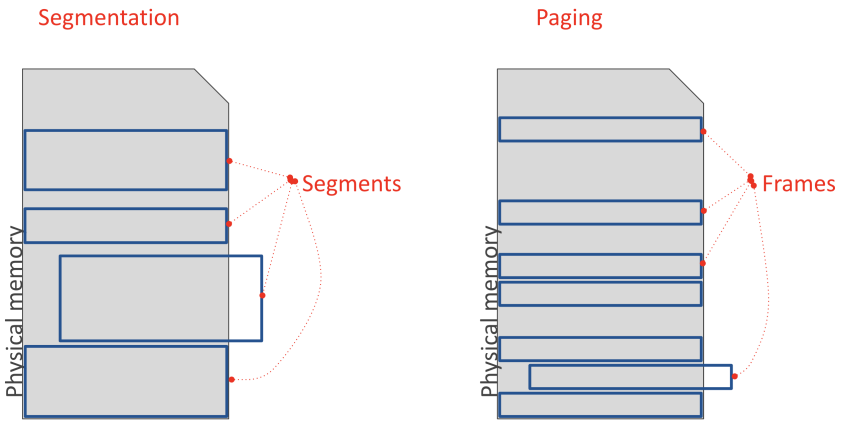
However, because each process must reside in one contiguous memory block, **fragmentation** can occur.

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.

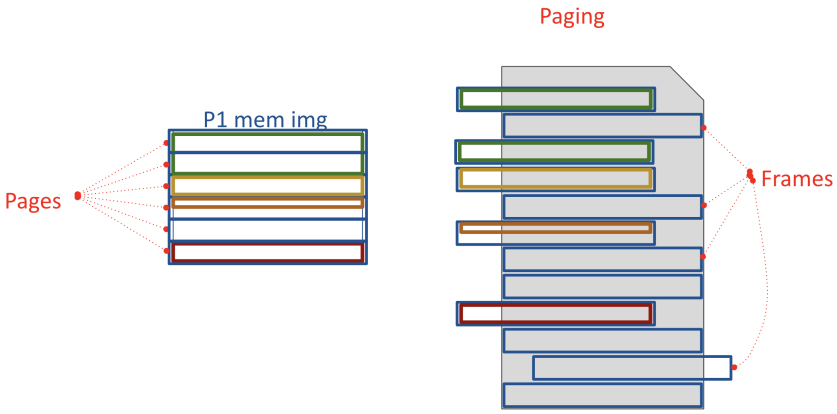


To address the inefficiency of requiring each process to occupy a single contiguous block of memory, an effective strategy is to *divide the process's address space into smaller chunks*, allowing noncontiguous allocation. Two primary techniques for accomplishing this are:

- **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.



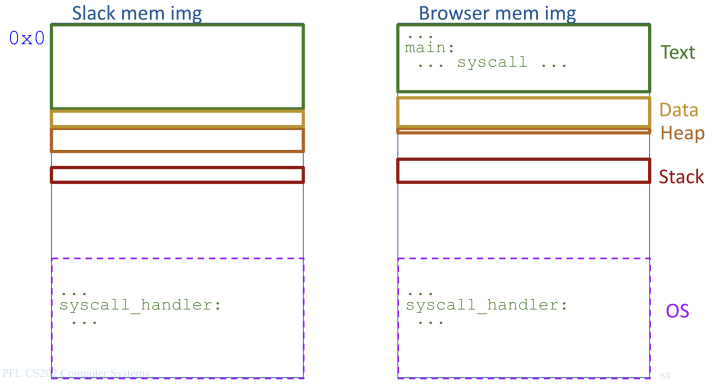
In **paging**, the MMU maintains a *page table* to translate from virtual pages to fixed-size physical frames:



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.

1.4 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.

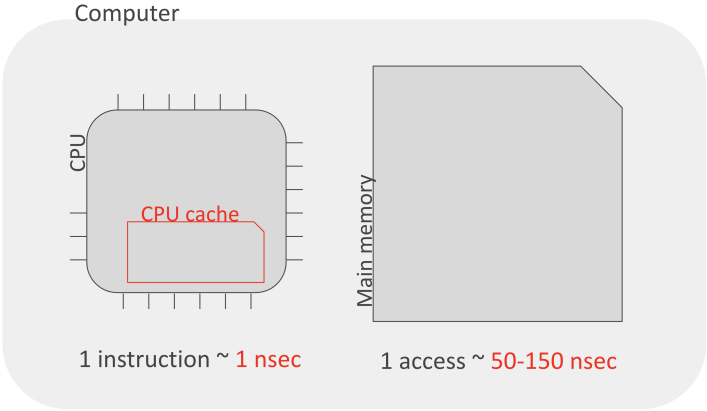


1.5 CPU Caching and Memory Hierarchy

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.

1.5.1 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.



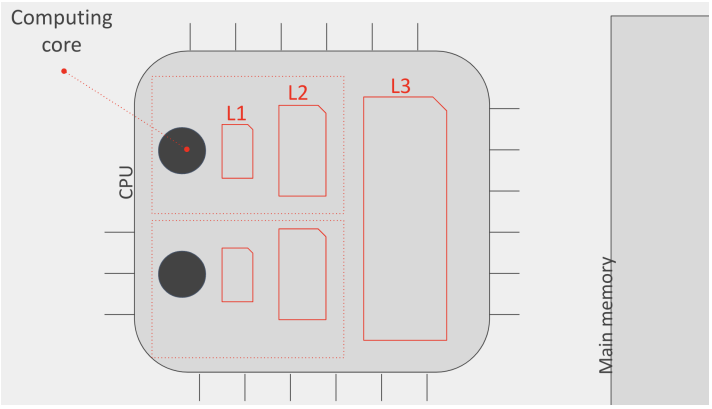
1.5.2 Multi-Level Cache Architecture

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.



1.5.3 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 1.5.3.1. *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.*

1.5.4 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.