- **Q1: What is an operating system?**
  - The Operating System (OS) is a fundamental layer of software that manages the hardware of a machine and helps "translate" it into standardized software abstractions, allowing user programs to run without directly handling low-level details.
  - It is the only piece of running software that interacts with hardware directly with full permissions, controlling resources such as the processor, memory, storage, and input/output devices.
  - All other software relies on the OS, as its intermediary, to make use all available hardware resources: For example, when running a program, users do not need to decide how much processor time their program will get from each processor, in a multiprocessor system; how much physical memory will use and at what ranges; or, how much storage and at which disk blocks.

- **Q2: Why do we need an operating system?**
  - Multiple users, not necessarily sane, wish to build and run different applications simultaneously on the same (shared) hardware.

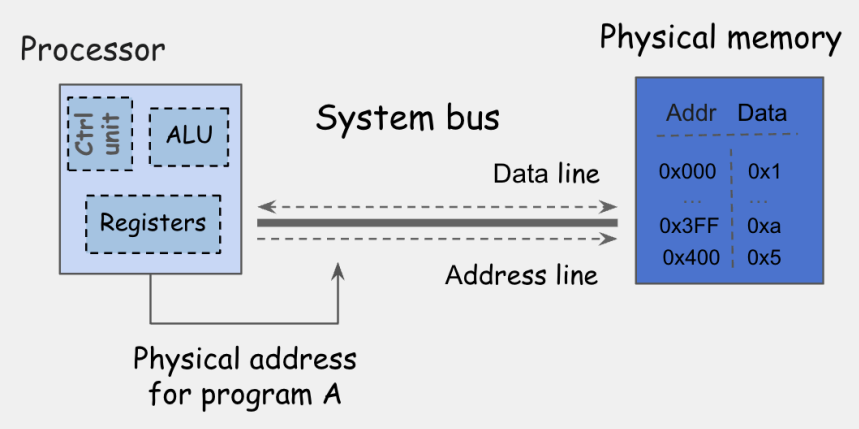
- **Q3: Why learning about operating systems?**
  - Understanding how and why the things you use work will help you build better software.
  - A line of code is not just a line of code: A myriad of things happen under the hood.
  - Increasingly, code is being developed with only a superficial understanding.
    - No free lunch: Amateur codebases won't fix themselves in production.
    - The requirement for software engineers who seriously understand how systems work at their core will soon be at an all-time high.
  - Learning how to navigate and contribute to a complex system with a large codebase, such as an OS kernel, will help you become a fearless software engineer.

- **Q4: What desirable properties should an operating system have?**
  - **Security**: Does the system operate as expected in an adversarial context?
  - **Reliability**: Does the system operate as expected, given benign failures?
  - **Portability**: Is it easy for developers to build /maintain apps?
  - **Performance and responsiveness**: Is it "pleasant" to use? [We'll elaborate more on this later.]

- **Q5: What principles should we follow in designing an operating system?**
  - **Fault isolation:** Each user program must be **confined** to run within its own region of physical memory. (Although it's given the illusion that it has the complete memory for itself.) That is, load/ store/ jump instructions of a running program cannot read, write, or enter another program's memory regions. Such a design, therefore, provides **fault isolation** between any number of user programs running on the same hardware simultaneously.
  - **Principle of Least Privilege (PoLP)**: A*ny system entity should only have the minimum privileges necessary to perform its function—no more, no less.* In that spirit, there must be privilege separation between the OS—which is the only trusted entity with full access to perform any operation involving the hardware—and user programs that do not get full access and are, thus, forced to request from the OS to perform privileged operations on their behalf.
  - **Preemption**: The OS needs to maintain the ability to periodically take control of the processor regardless of what user programs are executing.

- **Q6: What is the basic hardware model assumed when building an OS?**
  - **Physical Memory (PM):** Stores data addressable on a byte granularity. The memory addresses in physical memory are called **Physical Addresses (PA)**.
  - **Processor**: Reads data from memory to its registers, performs computations (e.g., arithmetic and logical operations) with the data that is held in its registers, and writes data from its registers back to physical memory.
  - **System bus**: The interconnection between the physical memory and the processor, which consists of an address and a data line. The processor places addresses on the address line to either (i) read data from memory via the data line, or (ii) write data to memory via the data line.

A basic model of hardware



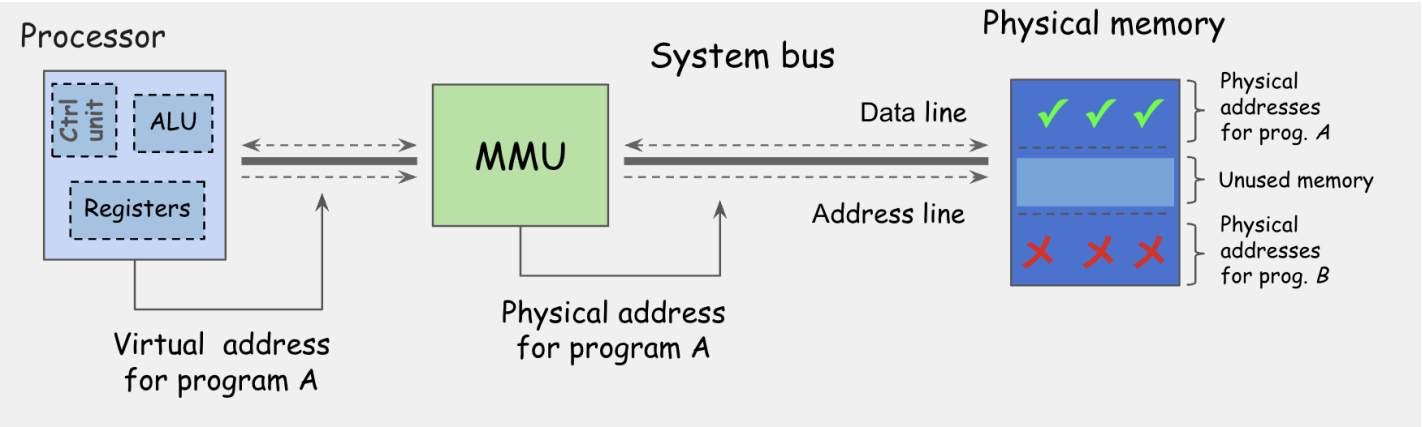
- The above simplistic model is very powerful and can be used to express and perform any computation. **Why?** Obvious correspondence to a Turing Machine.
  - A tape divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. [Akin to data in physical memory.]
  - A head that can read and write symbols from/to the tape, and move over the tape left and right. [Akin to the system bus.]
  - A state register that stores the state of the machine, one of finitely many. [Akin to a processor's registers.]
  - A table of instructions that, given (i) the current state of the machine, and (ii) the tape symbol currently under its head, tells the machine to do one of the following:
    - Either erase the symbol under its head or write a symbol where its head is at.
    - Move the head one step left or right, or stay at the same place.

- **Q7: How do we implement fault isolation and preemption in line with the PoLP?**
  - **Fault isolation**: To give each running program its own range of addresses (address space) in physical memory and confine it within, we utilize a key hardware component—called the Memory Management Unit (MMU)—which sits between the processor and the physical memory.
    - Any address produced by the processor while running a program is treated as a **Virtual Address (VA)**—i.e., one that does not really exist in physical memory, but the processor imagines that it does.
    - When the processor generates a virtual address, it's passed to the MMU, and the MMU translates it to the corresponding physical address (i.e., one that really does exist in physical memory).

- To convert virtual addresses to physical addresses, the MMU uses an index of address translations (called the **page table**) maintained and updated by the OS for each running program.

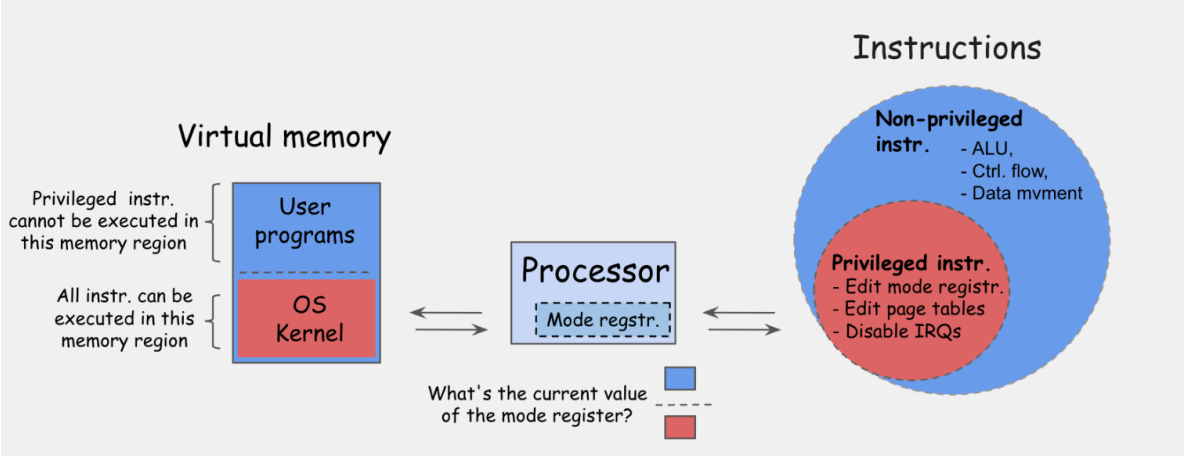
With the above design, regardless of what addresses programs calculate given their perception of **Virtual Memory (VM)**—i.e., the range of all the VAs that the processor may generate while executing them—the OS redirects and confines each program to its isolated physical memory range. Therefore, their faults (e.g., a corruption of data stored in a memory address due to a miscalculation) are isolated.

Hardware support for virtual memory



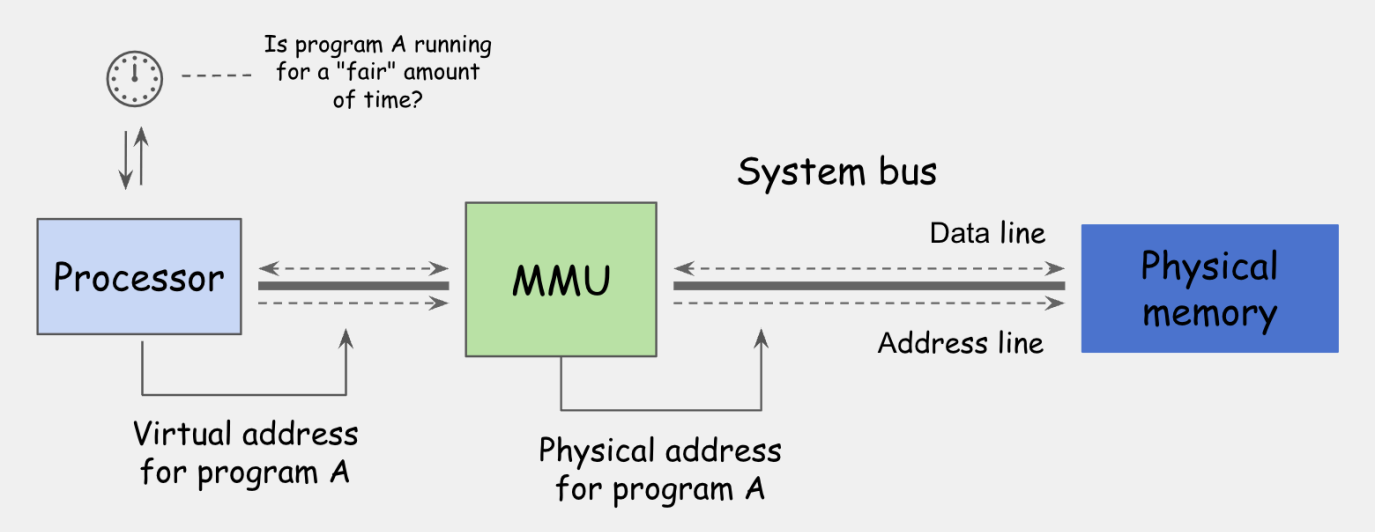
- **Principle of Least Privilege (PoLP):** To enforce the desired privilege separation between the OS (trusted entity) and user programs, we need hardware support for **dual-mode of execution.** That is, some instructions must be privileged and only allowed to be executed by the OS kernel (i.e. in **kernel mode** of execution) but not by user programs (i.e., in **user mode** of execution).
  - A mode register keeps track of the current mode of execution. [e.g., on Linux x86_32, the 2 bits of the %cs control register are used to define 4 execution modes—also called "rings" or Current Privilege Levels (CPLs): kernel mode CPL is ring-0 and user mode CPL is ring-3.]
  - The value of the mode register cannot be changed in user mode.
  - The base of the page table index (e.g., kept in %cr3 control register in Linux x86_32) can only be written in kernel mode and, thus, user programs cannot change their address space.
  - Any attempt to write to the register holding the base address of the page table index in user mode will trigger a General Protection Fault (GPF), Exception #13. [Q: Why is reading this register OK in user mode?]
  - User programs can only ask the OS to perform privileged operations for them. There is a structured way of doing so via **system call (syscall)** interfaces.
    - The user program should express what its request is (e.g., by pointing to the appropriate system call interface), prepare the arguments for it, and notify the OS of its request.
    - When the OS takes control, it can switch the mode register to get full hardware access, and because the OS is trusted, we are sure that it will do all appropriate checks for each syscall to assess that the requested operation is sane and rational before performing it.

Hardware support for dual-mode execution



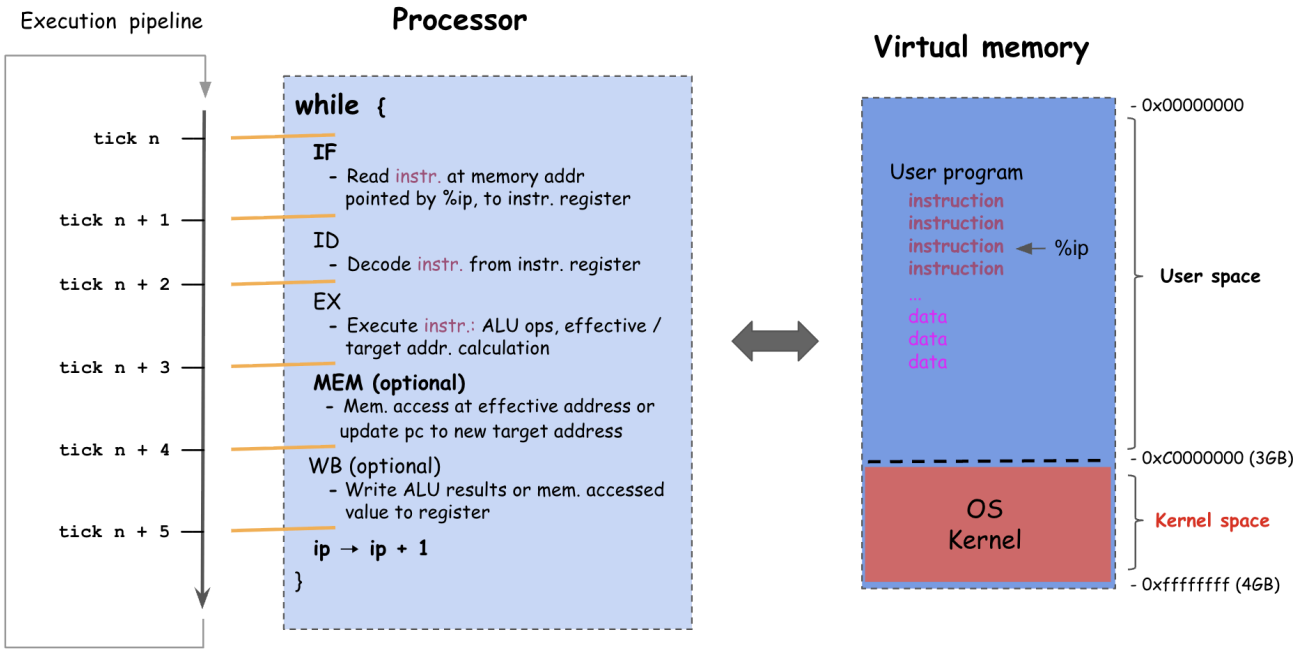
- **Preemption**: To understand how we implement preemption, we need a bit of background on what interrupts are and how the hardware handles them as well as background on time-sharing and scheduling. But at a high level:
    - Assume each processor has hardware support for letting us set a timer that periodically expires on a predefined interval. (Each timer interrupts one processor; therefore, a multiprocessor CPU will usually have one hardware timer for each core.)
    - After completing the execution of each instruction, the processor checks if its timer has expired.
    - If the timer has expired, the processor transfers control to the OS.
    - The OS decides if the user program that was just running has hijacked the processor for too long a time, or is behaving fairly based on statistics it maintains for each running program. (The part of the OS concerned with such decisions is the scheduler subsystem—a core component for which we will talk extensively later on.)
    - The user program resumes execution, or it is penalized for its behavior and the OS selects another program (if any) to give access to the processor.

Hardware support for preemption



- **Q8: How does the processor execute programs?**
  - Recall the simple model of computation from Q4: "The processor reads data from memory to its internal state, performs computations, and may write data back from its internal state to memory." In reality, for each instruction to be executed, there is a more complicated **execution pipeline**, which consists of five basic stages. (In modern processors, pipelines are considerably deeper because each of the five basic stages is further subdivided.)
    - **Instruction Fetch (IF)**: The processor fetches the instruction at the memory address pointed by the **Instruction Pointer (IP)**.
    - **Instruction Decode (ID):** The fetched instruction is decoded, and the processor determines what operation to perform.
    - **Execution (EX):** In this stage, the instruction is executed using the processor's registers, the Arithmetic Logic Unit (ALU), and so on.
      - If the instruction involves memory access (load/store), the calculation of the source or destination address happens here as well.
      - If the instruction involves conditional branches, this stage determines if the condition for branching is met.
    - **Memory Access (MEM):** If the instruction is a memory-related operation, in this stage, the necessary memory access is performed.
      - For load instructions (reading from memory), the data from the source memory address is fetched.
      - For store instructions (writing to memory), the data is written to the destination memory address.
    - **Write-Back (WB):** If the instruction is a load operation, in this stage, the fetched data is written back to the destination register.
  - Since memory stores both data and instructions, the processor needs to interact with memory in two of the above five pipeline stages:
    - **Instruction Fetch:** The processor reads instructions from memory to its internal state–registers, instruction queue, buffers,
    - **Memory Access:** The processor reads or writes data from/to memory as a result of executing a fetched instruction.

The stored program model



- Instruction classes

- Data movement instructions (e.g., load, store).
- ALU instructions (e.g., add, sub, and, or, shl).
- Control flow instructions (e.g., jmp, jne).
- Stack and subroutine instructions (e.g., call/ret, push/pop).
- Privileged instructions (e.g., hlt, iret).

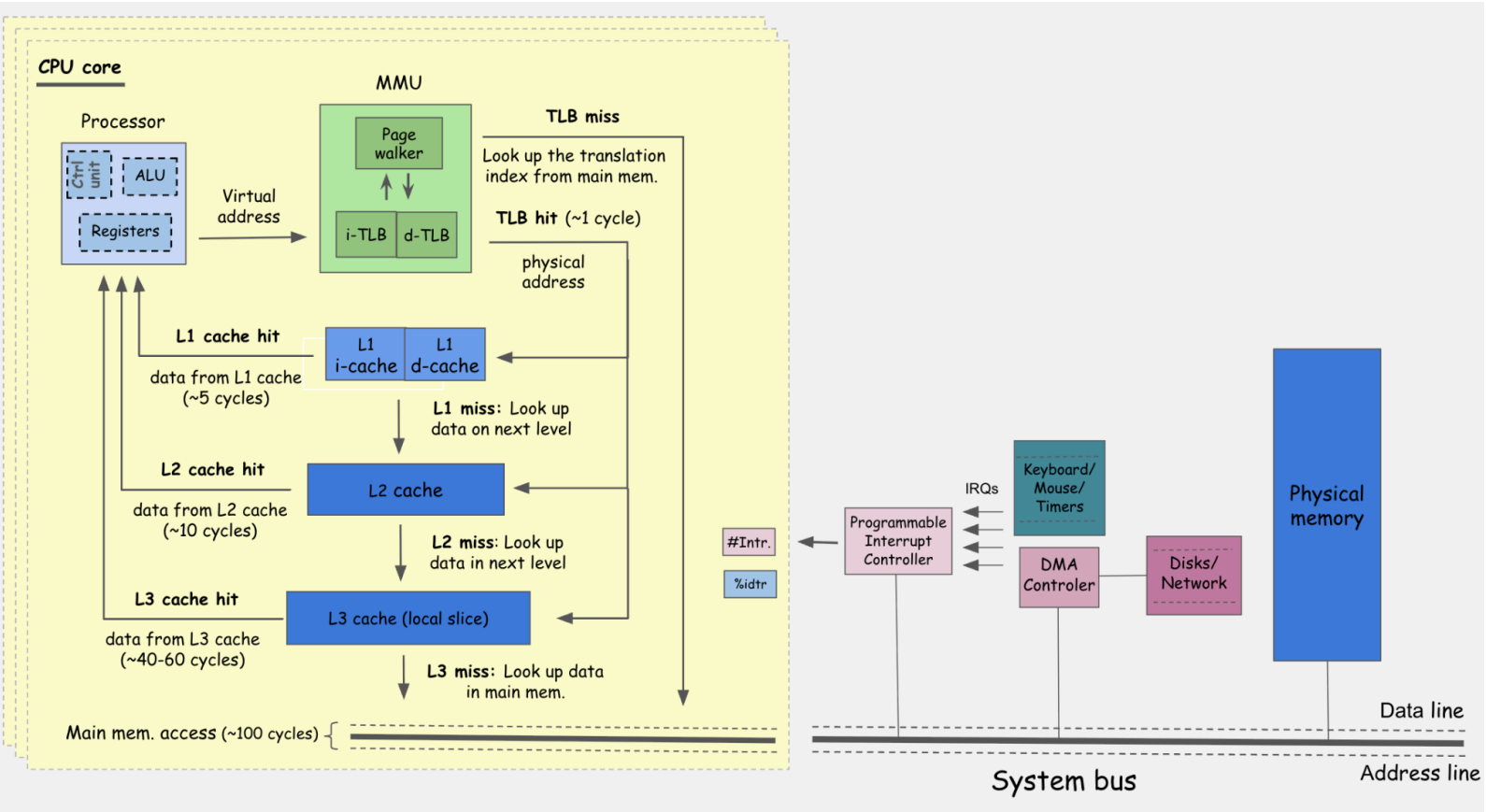
- Example *registers*

- Instruction pointer: %eip (x86_32); %pc (referred to as %r15 on aarch32**).**
- Stack-related: %esp (top of stack on x86_32), %ebp (frame pointer on x86_32**)**; %sp (top of stack on aarch32, referred to as r13), %fp (frame pointer on aarch32, referred to as %r11).
- General-Purpose: %eax, %ebx, %ecx (x86_32**)**; %r0–%r12 (aarch32**).**
- Control: %cr2 (stores the faulting virtual address in case of a page fault on x86_32), %cr3 (holds the base address of the page table on x86_32).

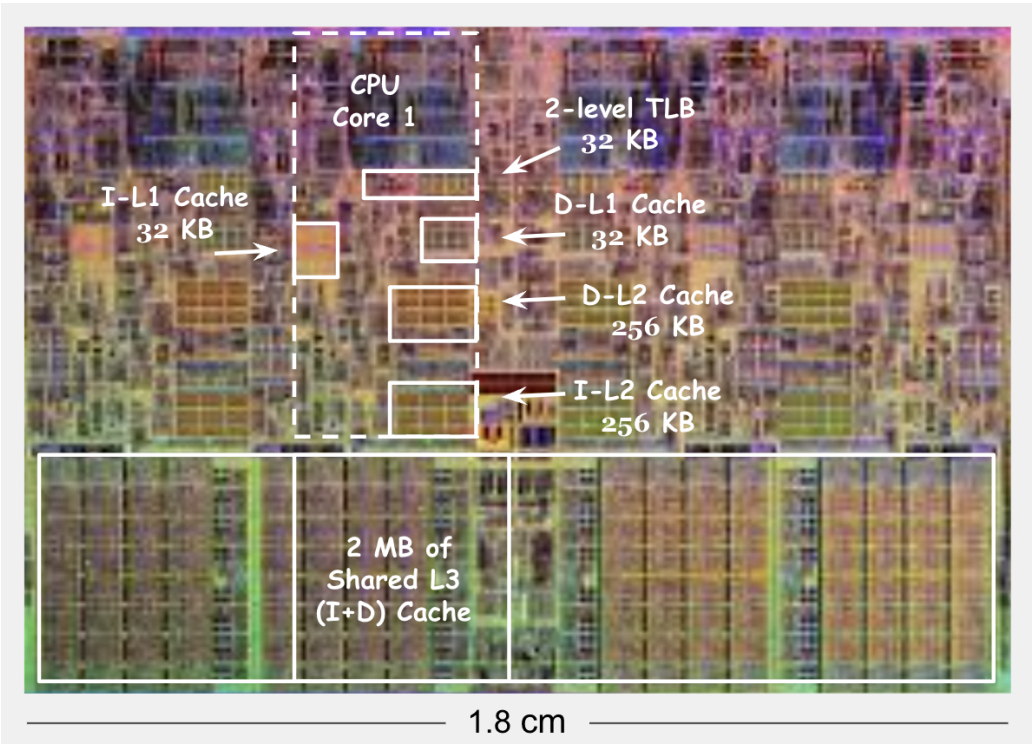
- Q9: What are the most basic hardware components?

- The MMU consists of an address translation cache, called **Translation Lookaside Buffer (TLB)**—which caches the most recent virtual-to-physical address translations— and a **Page walker**—which, in case a TLB miss occurs, it automatically walks the page table stored in PM and updates the missing translation; unless the translation is also missing from the table in PM, or it has invalid permission, and then a page fault exception is raised.
- A per-core **L1 cache**, which caches a subset of memory contents closer to the processor and helps avoid the latency of accessing physical memory. The L1 cache usually sits pre-MMU; it is virtually indexed and physically tagged, and is usually accessed in parallel with the TLB.
- A per-core **L2 cache,** which acts as a larger but slightly slower cache than L1. It is accessed post-MMU and is, therefore, physically indexed and physically tagged. [An **L3 cache** (not shown) is also commonly shared across all processor cores.]
- A **System bus**, which is the interconnection between the PM and the processor, with an address and a data line. The processor sends addresses on the address line and either gets back data from memory via the data line, or writes data to memory via the data line.
- A **Programmable Interrupt Controller (PIC),** which is responsible for notifying the processor when devices have **Interrupt Requests (IRQs).** The PIC translates IRQs to interrupt numbers based on which device requests service from the OS, and raises the interrupt to the processor. [The processor uses the #Intr. to index and execute the appropriate **Interrupt Service Routine** (**ISR**) in the **Interrupt Descriptor Table** (**IDT**) pointed by the **%idtr register**. [PIC can prioritize multiple requests, and it's possible to disable interrupts. Will explain later.]
- A **Direct Memory Access (DMA)** controller allows certain hardware subsystems to access PM independently of the processor. The processor initiates the transfer, offloads it to the DMA, performs other operations while the transfer is in progress, and finally receives an interrupt from the DMA controller when the operation is done.

A more complete picture



Nehalem Intel® Core i7 (Nov. 2008)



Access times

- TLB: ~1 clock cycle (~0.5 ns)
  - Typical hit rate: > 99%
- L1 cache: ~5 clock cycles
  - Typical hit rate: > 95%
- L2 cache: ~10 clock cycles
  - Hit rate (general workloads): 70-95%
- L3 cache: ~40–60 clock cycles
  - Hit rate (general workloads): 30%-80%
- Physical memory access: ~100 clock cycles (same channel), ~150 clock cycles (different channel)
- SSD access: ~100 μs
- HDD access: ~1 ms

- Q10: How does the system boot?

When the system powers on

- The processor is in "real mode". That is, the MMU is disabled and therefore physical and virtual addresses are the same.
- The first 1MB of physical memory, 0x000f0000–0x00100000, is directly mapped by the motherboard to ROM.
- ROM contains the BIOS: a piece of software that knows how to load the bootloader from the disk into RAM.
- The IP points at address 0x000ffff0, and starts executing the BIOS code.
- The BIOS loads the bootloader from disk to memory.
- The user selects which OS kernel to load (or the default is chosen).
- The bootloader loads the OS kernel from disk to memory.
- The kernel flips the processor's mode from real to protected mode (by setting the first bit of the %cr0 register), sets up its page table entries, and enables virtual memory.

Booting Linux on x86_32