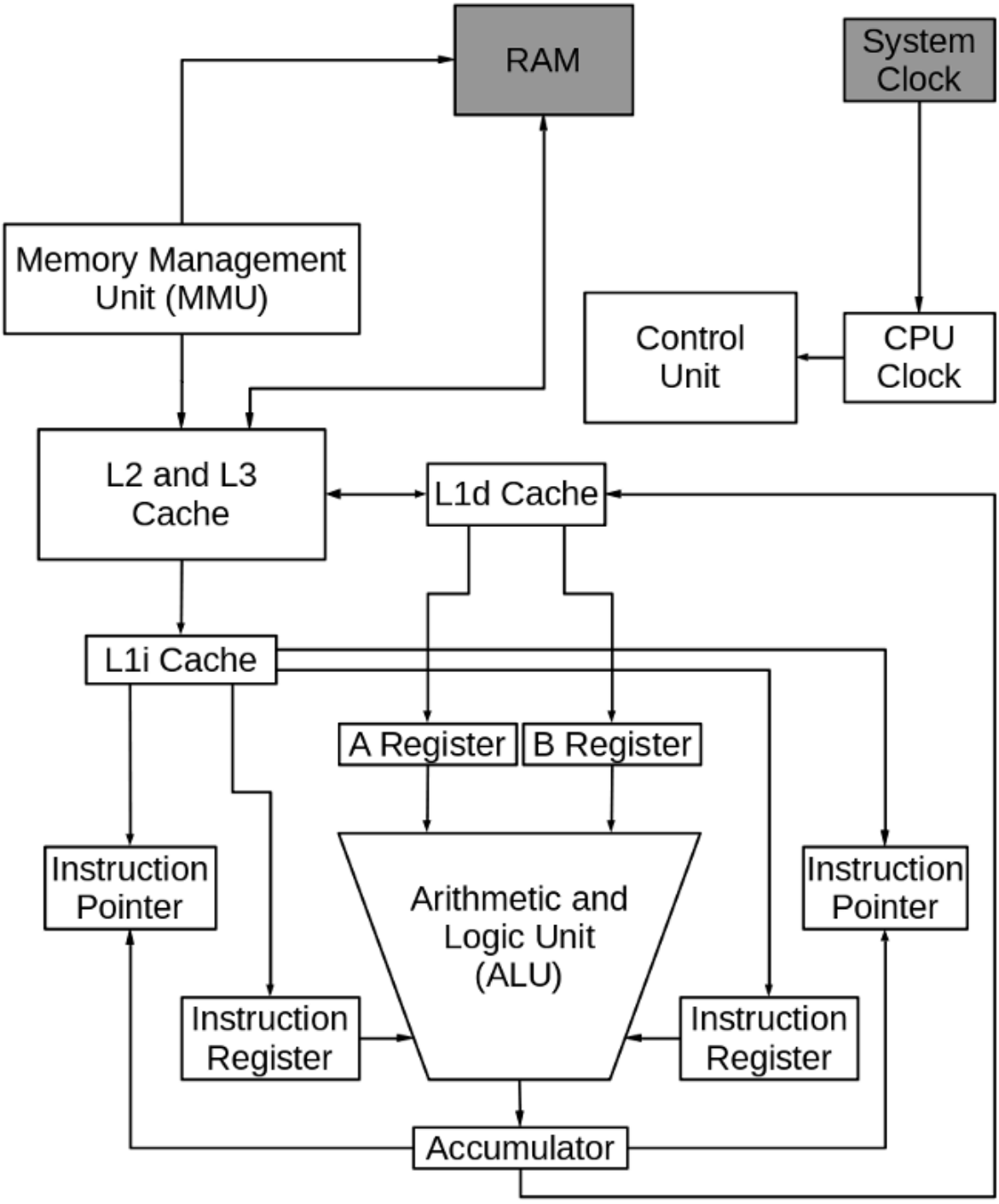
COMPUTER ARCH AND OS INTERVIEW QUESTIONS

Good questions and answers can be found here

General OS/CA Questions

- *What are some of the components of a microprocessor?*
  - **Arithmetic and logic unit**: performs math computations such as division, addition and subtraction and Boolean functions;
  - **Registers**: act as the temporary data holding places of microprocessors;
  - **Control units**: receive signals from the CPU and move data from one microprocessor to another;
  - **Memory caches**: accelerate the computing process, as the CPU doesn't have to use the slower RAM to retrieve data.



- *What is the difference between a thread and a process?*
  - **A process** is an independent program that runs in its own isolated memory space and has its own set of system resources, such as file descriptors and environment variables. Each process is scheduled by the operating system independently, providing a high level of isolation from other processes. This isolation ensures that if one process crashes, it generally does not affect the others, but it also means that inter-process

communication (IPC) is more involved, typically relying on mechanisms like pipes, sockets, or shared memory with proper synchronization.
    - **A thread**, on the other hand, is a smaller unit of execution that runs within a process and shares the same address space and resources as other threads in the same process. Threads enable parallel execution within a single process and allow for efficient communication and data sharing since they access the same memory directly. However, this shared environment also means that threads must be carefully synchronized to avoid race conditions, and a fault in one thread can potentially compromise the entire process. In summary, while processes provide isolation and robustness, threads offer lightweight parallelism and efficient resource sharing within the same application.

- *What is the difference between a thread and a core?*
    - A **core** is a physical execution unit on the CPU chip. Each core has its own execution pipelines, ALUs, registers, often its own L1 caches, and so on. If you have a 4-core CPU with no hyper-threading, the machine can truly execute 4 independent instruction streams at the same time, one per core.
    - A **hardware thread** (what the OS calls a “logical CPU”) is a separately schedulable execution context *inside* a core. With Simultaneous Multithreading (SMT) / Hyper-Threading, a single physical core can expose 2 (or more) hardware threads. They share the core’s physical resources (execution units, caches, etc.), but each has its own architectural register state and can run its own instruction stream. To the OS, each hardware thread looks like a separate “CPU”, even though two of them may be sharing one core.
    - So, for example, a CPU advertised as “4 cores / 8 threads” means: 4 physical cores, each supporting 2 hardware threads, and the OS will see 8 logical CPUs to schedule work onto.
    - On top of that, there are *software threads* (POSIX threads, C++ std::thread, Java threads, etc.), which are just abstractions the OS scheduler maps onto those hardware threads. You can have many more software threads than hardware threads; the OS time-slices them on the available logical CPUs.

- *Can multiple processes run concurrently on the CPU?*
    - In most operating systems, processes are managed through time-sharing, where the CPU is allocated to each process for a short interval in turn. The kernel performs context switches between processes, which means that while only one process is

running on a CPU core at any given instant, the rapid switching between processes creates the illusion that they are running concurrently. This scheduling is managed by the kernel's process scheduler, which determines how much CPU time each process gets and in what order, ensuring that each process appears to run independently.
    - Additionally, on systems with multiple cores or processors, true parallelism is possible when different processes (or threads within processes) run simultaneously on separate cores. However, even in those cases, each core is still executing one thread at a time, and the kernel orchestrates these tasks across cores. In single-core systems, or when more processes are active than there are cores, the kernel's context switching provides the illusion of simultaneous execution by interleaving small time slices of execution for each process.

- *What is the difference between physical and virtual memory?*
    - **Virtual memory** is an abstraction provided by the operating system that gives each process its own contiguous address space, independent of the actual physical memory available. This means that from the perspective of a process, it appears to have a large, continuous block of memory starting at address zero—even if that memory is not physically contiguous in RAM. This mechanism maps virtual addresses to physical addresses through **page tables**, ensuring that each process only accesses its own allocated physical memory. Even if the physical memory is "full," the OS can use techniques like paging (swapping parts of memory to disk) to manage allocations without corrupting memory between processes.
    - **Physical memory**, on the other hand, refers to the actual Random Access Memory (RAM) installed in a computer. It is the hardware resource that stores data temporarily while a system is running. The operating system manages this limited resource by mapping virtual addresses to physical locations, ensuring that the contents of one process do not interfere with another. While virtual memory provides a convenient and flexible abstraction for application developers, physical memory is the underlying resource that ultimately determines a system's real-time performance and capacity.
    - See this video for more info.

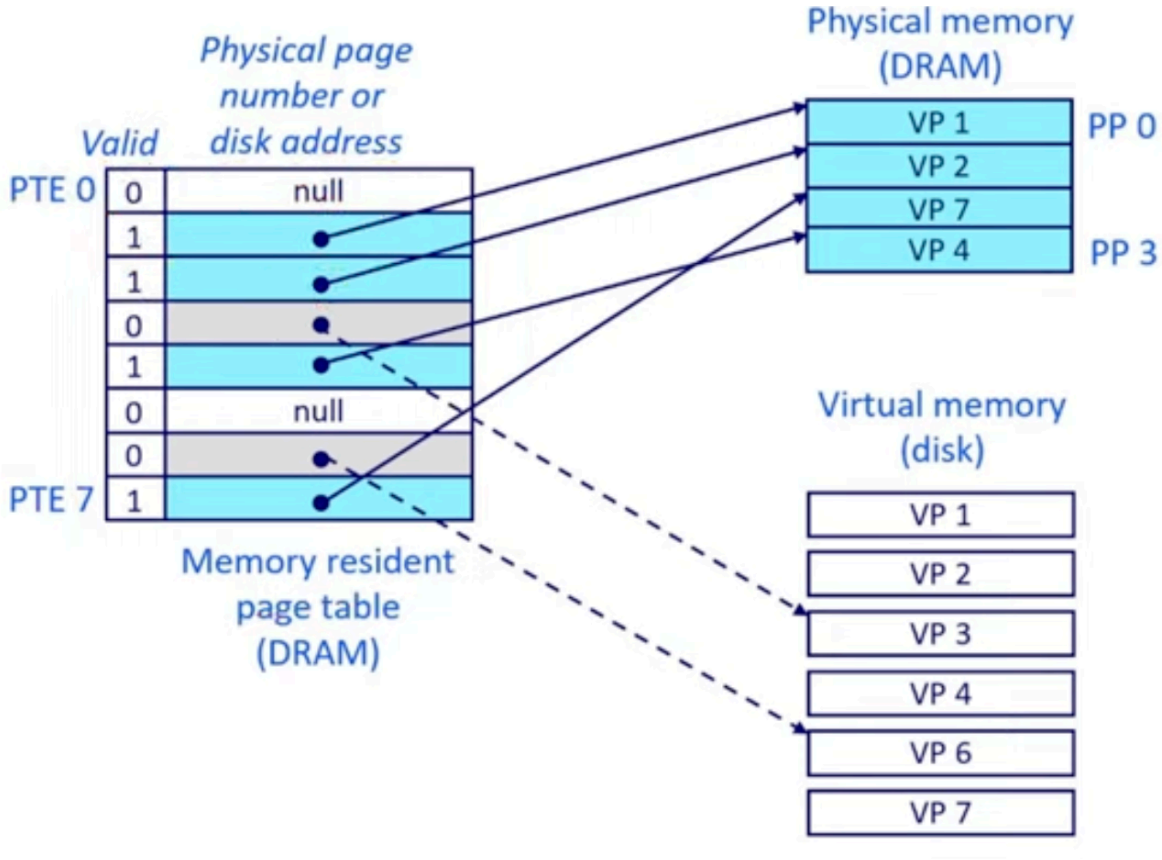
- *What is paging in OS?*
    - First it would be good to explain what is a page and a page table:
        - A **page** is a fixed-size block of virtual memory used as the basic unit for memory management. Each virtual memory page is mapped to a **physical memory frame of the same size**, and these sizes are always a power of two. Virtual memory in modern operating systems is divided into pages—commonly 4 KB in

size, though other sizes are possible—so that memory can be managed in uniform chunks. This division simplifies memory allocation, paging (swapping between physical memory and disk), and protection, since each page can be independently managed. When a process accesses memory, the virtual address is split into a page number and an offset, allowing the system to locate the corresponding block of physical memory.

    - A **page table** is the data structure that maps virtual pages to physical frames in RAM. Each entry in the page table, often called a page table entry (PTE), contains information about a virtual page, such as the physical frame number where the page is stored, access permissions (read, write, execute), and status bits (like present/absent, dirty, or access history). The memory management unit (MMU) uses the page table to translate virtual addresses to physical addresses during program execution.

- **Paging** is a memory management scheme that allows the operating system to map a process's virtual address space to physical memory in a flexible, non-contiguous way. In paging, both the virtual address space and the physical memory are divided into fixed-size blocks—pages and frames respectively (typically 4KB in size, though larger page sizes may be available on some systems). Each page in the virtual address space can be mapped to any available frame in the physical memory, and this mapping is maintained in a data structure called the page table. The hardware memory management unit (MMU) uses this page table to translate virtual addresses into physical addresses on the fly, often with the help of a cache-like structure called the

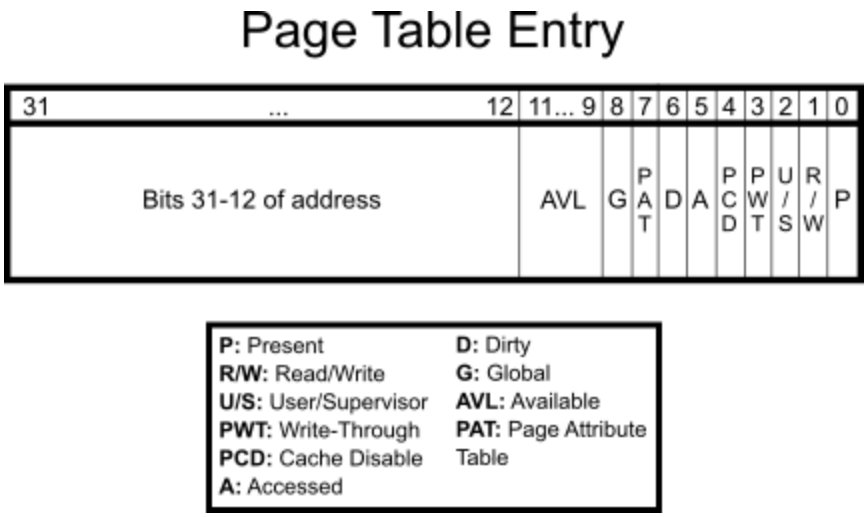
Translation Lookaside Buffer (TLB) to speed up repeated translations.



- See this video for more info.

- *What information is typically contained in a page table entry?*
  - At a high level, a Linux **page table entry (PTE)** either (a) **points to the next page-table level** (a non-leaf entry) or (b) **maps one virtual page to a physical page frame** (a leaf entry). A leaf PTE packs a **physical frame number (PFN)** together with a set of **hardware permission/state bits** and some **Linux-specific software bits**.
  - **What's in a leaf PTE (conceptually across CPUs):**
    - **PFN / physical address bits** — which physical page frame this virtual page maps to.
    - **Present/valid** — whether the mapping is currently valid (otherwise a fault occurs).
    - **Read/Write permissions** — controls stores; write-protect used for COW and write-tracking.
    - **User/Supervisor (privilege)** — whether user mode may access it.
    - **Execute permission** — controls instruction fetch (NX/XP bits).
    - **Accessed/Young** — set by hardware on first use; OS uses it for aging/reclamation.

- **Dirty** — set on first write; used for writeback/eviction decisions.
- **Global / nG** — whether the TLB entry is global (not flushed on ASID/CR3 changes).
- **Cache/memory type** — e.g., write-through vs write-back, device vs normal memory (PAT/AttrIndx), shareability on ARM.
- **Huge-page marker** (at higher levels) — says “this entry maps a **2 MB/1 GB** (x86) or **contiguous/large** page” instead of pointing to the next level.



- *What are multi-levels page tables? Why do we need them?*
  - A page table is the data structure that an operating system uses to translate virtual addresses—what processes work with—into physical addresses in RAM. In its simplest form, a flat page table is just a large array where each entry corresponds to one page of virtual memory; the processor walks this array to find the frame number for any given virtual page. While straightforward, a flat page table for a 32-bit address space with 4 KB pages already requires an array of 2^{20} entries, and for 64-bit systems the size becomes astronomically large. Allocating and managing such huge contiguous tables for every process would waste enormous amounts of memory, most of which is often unused.
  - Multi-level page tables solve this problem by splitting the page-table array into a hierarchy of smaller tables, arranged in a tree whose levels correspond to successive chunks of the virtual-address bits. On a lookup, the CPU uses the topmost bits to index into the first-level table; that entry points to a second-level table indexed by the next bits, and so on until it reaches the final level, which contains the physical-frame number. By only allocating lower-level tables when a process actually maps pages in that region, the OS avoids reserving space for vast areas of unallocated or unused memory. This sparse allocation dramatically reduces per-process memory waste, especially on 64-bit machines.

- Although a multi-level walk incurs several memory references when the translation is not cached, modern processors employ a Translation Lookaside Buffer (TLB) to cache recent virtual-to-physical mappings and eliminate almost all page-table walks on warm hits. Moreover, many architectures include hardware-accelerated page-table walkers that streamline the hierarchical access. The net result is that the occasional extra indirection is far outweighed by the savings in memory footprint and the flexibility to support huge virtual address spaces without blowing out OS overhead.
- Beyond basic two- or three-level schemes, more elaborate variations—such as inverted page tables, hashed page tables, or radix trees—have been developed to optimize for multi-gigabyte or sparse address maps in huge-memory or 64-bit environments. Nevertheless, the core insight remains: by breaking the page table into levels, the operating system can represent a very large address space economically, allocate bookkeeping structures on demand, and keep the performance cost manageable through TLB caching and hardware support.
- See this video and this video for more info. See also this article.

- *Assume we are on a 32-bit system with $4\ \mathrm{KB}$ pages. What would be the number of pages? How much memory would be taken by a flat page table in this context?*
  - A single page is $4\ \mathrm{KB} = 4 * 2^{10}$ bytes in size. Since we are on a 32-bit system, that means we must have $2^{32}/2^{12} = 2^{20}$ pages. Hence, assuming that a PTE is 4 bytes, the total space taken up by a flat page table would be $4 * 2^{20} \approx 4\ \mathrm{MB}$.

- *Assume we are on a 32-bit system with $4\ \mathrm{MB}$ pages. What would be the size of the offset part of a single page address?*
  - A single page is $4\ \mathrm{MB} = 2^{22}\ \mathrm{bytes}$. Hence, the size of the offset part of the memory address is $22\ \mathrm{bits}$, enough to access every bytes in the page. Hence, since the memory address total size is $32\ \mathrm{bits}$, we know that the remaining $32 - 22 = 10\ \mathrm{bits}$ are used as an index to the $2^{10} = 1024$ entries in the page table.

- *When the CPU needs to load/write a virtual address, assuming there is no page fault, will the kernel be involved at all?*
  - **Short answer:** on modern x86-64 and ARMv8 systems, **no**—if the address is valid and there's **no page fault**, the kernel is **not** involved in the load/store. The CPU does the whole translation and access on its own.

- **What happens:** running in user mode, the core looks up the virtual address in the **TLB**. If it hits, it uses the cached translation and performs the load/store (hitting L1/L2/L3, updating the **Accessed/Dirty** bits, etc.). If it **misses in the TLB**, the core performs a **hardware page-table walk** using the page tables the kernel set up earlier; if the walk succeeds (present + permissions OK), the TLB is filled and the access completes—still **no kernel entry**.
- **When the kernel *would* get involved:** only if something exceptional happens—e.g., the PTE isn't present, permissions fail (user tries to write a read-only page or execute NX), or copy-on-write/demand paging needs servicing. Those conditions trigger a **page fault trap** into the kernel. Similarly, kernel participation is needed when setting up or changing mappings, doing TLB shootdowns, handling IO/MMIO, etc.—but **not** for routine, successful data accesses.
- **One historical exception:** on some architectures with **software-managed TLBs** (e.g., classic MIPS or older SPARC), even a TLB miss on a valid mapping traps to the kernel (or a tiny handler) to refill the TLB. That's not the case for mainstream x86-64 and ARMv8 desktops/servers today.
- See this video for more info.

- *Can you explain exactly what happens when the CPU needs to load/store from a virtual address?*
  - Here's the typical fast-path of a **load from a virtual address** on a modern x86-64/ARMv8 system (no page fault):
    1. **Core issues the load (VA in hand).**
       The load/store unit gets a **virtual address (VA)** computed by earlier pipeline stages.
    2. **MMU/TLB lookup.** The **MMU** checks the **TLB** (a cache of page-table translations):
       - **TLB hit:** you immediately get the **physical page number + permissions**; combine with the VA's page offset → **physical address (PA)**.
       - **TLB miss:** do a **hardware page-table walk**:
         - x86-64: walk **PML4** → **PDPT** → **PD** → **PT** (4 or 5 levels), following each entry's PFN and checking **Present, U/S, R/W, NX** bits. Huge-page hits (1 GB at PDPT, 2 MB at PD) can end earlier.
         - ARMv8: similar multi-level walk (TTBR/translation tables) honoring **AP, UXN/PXN, AttrIndx, SH**.
         - If every level is valid, fill the **TLB** with the translation and permissions; else you take a **page fault** (kernel handles it). Still no kernel on a successful walk.

3. **Permission checks.**
   The MMU enforces **user/kernel, read/write, execute-never** against the current CPU mode and the instruction type. If disallowed → **fault**. If allowed, continue.
4. **Derive the line and hit the caches.** Use the PA to index the cache hierarchy:
   - **L1 data cache** (VIPT/PIPT): check the **tag** in the set.
     - **Hit:** return the word/byte to the core (after forwarding from **store buffer** if a recent store wrote the same line).
     - **Miss:** consult **L2**, then **LLC (L3/SLC)**; if found at a lower level, fill back into upper levels.
5. **Miss in last-level cache → memory system.** The cache issues a **read request** to the memory subsystem:
   - **Coherence:** if other cores might hold the line, the **coherence protocol (e.g., MESI/MOESI)** finds an up-to-date copy (maybe in another core's cache) or ensures memory has it.
   - **DRAM:** memory controller fetches the 64-B (typical) **cache line** from DRAM into LLC, then to L2/L1.
   - Core completes the load when the line reaches L1 (often hundreds of cycles after an off-chip miss).
6. **Accessed/Dirty bookkeeping.**
   Hardware sets **Accessed/Young** on first touch; for **stores**, it also sets **Dirty** (and transitions the line to a writable state, see below). These bits live in the PTE/TLB entry and guide the OS's paging decisions later.
7. **Return data to the pipeline.**
   The load's data is now ready; dependent instructions wake and continue.

- **If this were a store instead of a load**
  - Steps 1–3 are identical (translation + permissions).
  - If the cache **has the line but not in a writable state**, the core issues a **Read-For-Ownership (RFO)** to gain **Exclusive/Modified** ownership (coherence invalidates other sharers).
  - With **write-allocate** (usual for CPU caches), a **write miss** fetches the line (RFO) then updates it in **L1**, marking it **dirty** (**write-back** policy).
  - The store retires as soon as it enters the **store buffer**; the cacheline update and writeback to lower levels happen **asynchronously**, respecting the memory model (barriers ensure ordering when you need it).

- *What is the difference between a page and a word?*

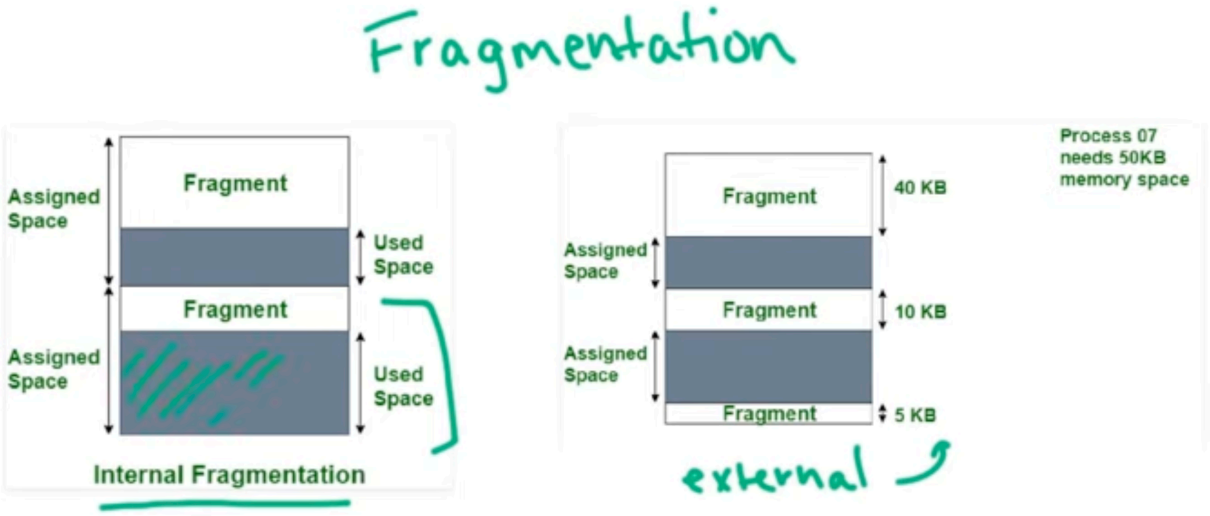
- A **word** is the basic unit of data used by a computer's processor, typically corresponding to the width of the processor's registers (for example, 32 or 64 bits). It represents the size of data that the CPU can process in a single operation, such as reading from or writing to memory. The word size determines many aspects of a computer's architecture, such as its natural data size, arithmetic precision, and addressing capabilities.
- A **page**, on the other hand, is a much larger block of memory used by the operating system for managing virtual memory. Pages are the fixed-size units into which both virtual memory and physical memory are divided, and they are typically measured in kilobytes (commonly 4 KB, though other sizes are possible). The operating system uses pages to map virtual addresses to physical memory via page tables, handle memory protection, and manage swapping of memory between RAM and secondary storage. In summary, while a word is a low-level unit of data manipulation by the CPU, a page is a higher-level unit used by the operating system for organizing and managing memory.

- *How can you relate the concept of 'stack' and 'heap' with virtual and physical memory?*
  - The concepts of "heap" and "stack" in C++ refer to how memory is managed within a process, while virtual memory is an abstraction provided by the operating system that underlies all memory accesses. Both the heap and the stack are regions within the process's virtual address space, which means that although your code allocates memory from the heap or stack, it is actually working with virtual addresses.
  - Virtual memory takes the process's virtual address space—which includes the stack, the heap, global variables, and code—and maps it to physical memory (RAM) using a mechanism called paging. This means that the physical memory backing your stack or heap does not have to be contiguous and can even be swapped out to disk if needed. The OS and hardware work together (through page tables, the MMU, and TLBs) to translate the virtual addresses you use in C++ into actual physical addresses. So while you, as a programmer, think in terms of heaps and stacks, these are just logical regions within the larger, virtualized memory system that the OS manages, providing benefits like memory protection, isolation, and the ability to run programs that require more memory than physically available.

- *What are internal and external fragmentation?*
  - Internal fragmentation occurs when allocated memory blocks contain unused space within them. This happens because memory is often allocated in fixed-size blocks or chunks, and if a process requests a size that does not exactly match these

predetermined sizes, the allocation may end up being larger than necessary. The extra space inside that allocated block—unused by the process—is referred to as internal fragmentation. This type of fragmentation is common in systems where memory allocation strategies round up requests to fit fixed block sizes, resulting in wasted memory that is reserved for the process but never actually used.

- External fragmentation, on the other hand, is the wasted space that exists between allocated memory blocks in the heap or other dynamically managed areas. Over time, as memory is allocated and freed in varying sizes, the free memory becomes divided into many small, noncontiguous blocks. Even if the total free memory is large, it may not be usable for a new allocation request if no single contiguous block is large enough to satisfy it. External fragmentation is a challenge in systems with dynamic memory allocation and can lead to inefficient utilization of memory, as the scattered free spaces are often too small to be practically useful.



- See this video which is a good summary.

- *Are page tables shared between all processes or does each process have its own page table?*
  - Each process generally has its own page tables that define its virtual address space. This means that every process gets its own mapping of virtual addresses to physical memory, which is managed by the operating system. In many systems, while the user space page tables are unique per process, the kernel space mappings may be shared among processes.

- *What happens if you run out of physical memory (RAM) and need to add more memory?*
  - There are several techniques used by a system when it runs out of physical memory

1. **Disk swapping:**
    - When RAM is full, the OS moves inactive memory pages to **swap space** (a reserved area on a hard drive that acts as a virtual extension of a computer's RAM) on the disk (HDD/SSD) to free up RAM for active processes. This allows the system to continue running but significantly slows performance because disk access is much slower than RAM. If too much swapping occurs, the system may enter a state called **thrashing**, where it constantly moves data between RAM and swap, making it nearly unresponsive. **Swapping is extremely slow** compared to RAM (disk access is **100,000x slower**).
2. **Page Reclamation**
    - The OS reclaims memory by **clearing caches, compacting memory pages, or terminating idle processes**. It prioritizes freeing memory occupied by non-essential background processes and cached disk data. While this can temporarily improve performance, aggressive page reclamation can lead to **increased CPU and disk activity**, especially if frequently accessed data is removed from memory.
3. **Out of Memory Killer (OOM)**
    - If memory pressure continues, the OS may **forcibly terminate high-memory processes** to prevent a system crash. On **Linux**, the **OOM Killer** selects processes based on memory usage and kills the least critical ones, while **Windows** prompts the user to close applications. Although this helps restore system stability, it can lead to **data loss and unexpected program termination**.

- *What is TLB?*
    - **A translation lookaside buffer (TLB)** is a type of memory cache that stores recent translations of virtual memory to physical addresses to enable faster retrieval. This high-speed cache is set up to keep track of recently used page table entries (PTEs). Also known as an address-translation cache, a TLB is a part of the processor's memory management unit (MMU).
    - A TLB hit means a PTE is present in the TLB and the processor has found it, given a virtual address. When this happens, the CPU accesses the actual location in the main/physical memory.
    - A TLB miss means a PTE was not found in the TLB. In this case, the page number is used as the index while processing the page table. In other words, the processor accesses the page table in the main memory and then accesses the actual frame in the main memory.

- A TLB is generally designed as a **fully associative cache** because it is a small, specialized cache that must minimize translation misses. In a fully associative structure, any virtual-to-physical translation can reside in any slot, which significantly reduces conflict misses. Since TLBs typically hold a few dozen to a few hundred entries, the hardware can efficiently search all entries in parallel. This design ensures that even if multiple pages map to different parts of memory, their translations can be stored without being forced into a fixed index—improving hit rates and overall performance.
- See this for more info.
- A good recap table for some of the main memory units:

Memory Unit	Size	Location	Speed (Fast → Slow)	Purpose
Registers	8–64 bits	CPU	**Fastest** (1 cycle)	Temporary data for instructions
Word	4–8 bytes	CPU	Fast	CPU's natural processing size
Cache Line	32–128 B	CPU Cache	Very Fast	CPU reads/writes in blocks
Memory Page	4 KB	Virtual Memory	Medium	OS memory management
Page Frame	4 KB	RAM	Medium	Physical counterpart of a page
RAM	GBs	Main Memory	Slower than cache	Active program storage
Swap Space	GBs	Disk (SSD/HDD)	Very Slow	Virtual memory backup
HDD/SSD	TBs	Secondary Storage	Slowest	Permanent storage

- *What does it mean to 'flush' the TLB? What is the cost of flushing the TLB? Are there ways used by modern architectures to prevent TLB flushes?*
  - Flushing the TLB means invalidating all or a subset of its entries, so that cached virtual-to-physical address translations are cleared out. When a TLB is flushed, subsequent memory accesses must perform full page table walks to re-establish these translations, which can significantly slow down performance.

- **In contrast to regular caches, TLB consistency is implemented by the operating system (OS), using a mechanism based on inter-processor interrupts (IPIs); called a TLB shootdown**.
- Good paper here.
- **Cost of Flushing the TLB:**
  - **Increased Latency:** After a flush, every memory access may incur the overhead of a page table walk until the TLB is repopulated.
  - **Performance Degradation:** Frequent flushes, such as during context switches or when updating page table entries, can degrade overall system performance due to increased translation overhead.
- **Techniques to Prevent TLB Flushes:**
  - **Address Space Identifiers (ASIDs) / Process Context Identifiers (PCIDs):** Modern processors tag TLB entries with an identifier corresponding to the address space. This means that when switching between processes, the TLB can retain entries from multiple processes, avoiding a complete flush.
  - **Selective/Partial Invalidation:** Some architectures allow for flushing only those TLB entries affected by a change, rather than flushing the entire TLB.
  - **Hardware Support for Coherency:** Certain processors incorporate mechanisms that automatically update or invalidate TLB entries when page table entries change, reducing the need for an explicit flush.

- *What is a TLB shootdown?*
  - TLBs are maintained separately per core, so they can become inconsistent if one core updates a page table entry while another core still holds an outdated translation. To handle this, operating systems use a mechanism called **TLB shootdown**. When a page table entry changes, the OS issues invalidation instructions (often via inter-processor interrupts) to flush the affected TLB entries on all cores, ensuring that every core has coherent, up-to-date address translations. This mechanism is analogous to cache-coherency protocols used for data caches, though it's implemented at the OS/hardware interface rather than through a hardware coherence protocol.
  - TLB shootdowns causes each affected core to context switch into the kernel and thus causes latency spikes for the process running on the affected cores. It will also cause TLB misses when a address with an invalidated page table entry is subsequently accessed.
  - Note that in NUMA, TLB shootdowns can become a particularly important issue to consider:

- On a NUMA system each processor or group of processors (often on separate nodes) maintains its own TLB. When a page table entry changes, every TLB that might have cached that mapping must be invalidated—a process known as a TLB shootdown. On NUMA architectures, the cost of this becomes particularly high because:
  - **Inter-node Communication Overhead:**
    TLB shootdowns typically use inter-processor interrupts (IPIs) to notify all affected cores. In NUMA systems, cores on remote nodes have higher communication latency, so flushing their TLBs takes longer and incurs more overhead.
  - **Distributed Memory Implications:**
    Since memory is distributed across nodes, stale TLB entries on any node can lead to significant performance issues or even incorrect behavior if not invalidated promptly. This necessitates careful coordination across nodes.
  - **Impact on Performance:**
    Frequent TLB shootdowns can severely degrade performance on NUMA systems due to the increased synchronization cost and latency of invalidating TLBs across multiple nodes.
- See this thread, this video starting at 1h28mins and this video.

- *What is a page fault?*
  - A page fault occurs when a process tries to access a page of memory that is not currently mapped into its address space or isn't present in physical memory. When this happens, the operating system steps in to resolve the fault by mapping the page into memory or loading it from secondary storage.
  - **Hard Page Fault:**
    This type happens when the needed page is not in physical memory at all. The OS must fetch the page from disk (or another backing store), which involves expensive I/O operations and takes considerably more time.
  - **Soft Page Fault:**
    This occurs when the required page is already in physical memory but isn't mapped into the process's address space or isn't in the TLB. This mostly happens in cases involving shared memory: for example when a program loads a shared library, it is mapped into the program virtual memory. When another program loads the same library, the library is already in physical memory but not yet map in the new program own virtual address space, hence a soft page fault occurs. The OS can resolve this fault quickly without disk I/O by simply updating the page table or TLB.
  - See this video and this article for more info.

- *What is a page/table walk? When does it happen?*
  - A **page walk** is the process by which the CPU's Memory Management Unit (MMU) traverses the multi-level page table structures in memory to translate a virtual address into a physical address. Normally, the MMU first checks the Translation Lookaside Buffer (TLB) for a cached translation. When there's a **TLB miss**, the MMU initiates a page walk, reading the page directory, page tables, and other hierarchical structures to locate the page table entry that maps the virtual address to its corresponding physical address.
  - This whole (multi-level) walk is done entirely by hardware (MMU), without involving the kernel, *as long as* the data in the page tables is valid. If the walk fails (no valid PTE, or permission bits deny access), the MMU raises a **page fault exception**. At that point, the CPU traps into **kernel mode**, and now the **kernel** gets involved.
  - This process happens at runtime on a TLB miss and can introduce additional latency, which is why TLBs are crucial for performance.

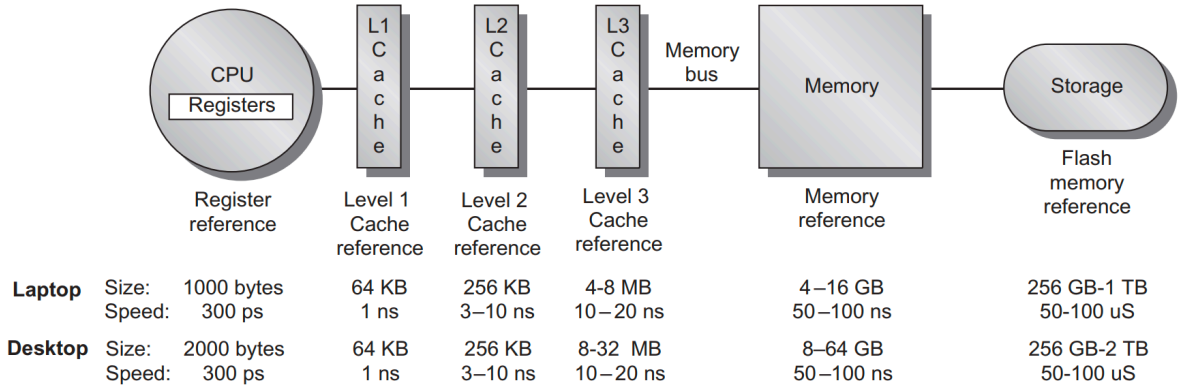
- *What are the typical different level of cache? What are their respective size/latency/associativity?*

  - Modern CPUs almost universally employ a three-level cache hierarchy—L1, L2, and L3—each level trading off size for speed and associativity.
  - The **Level 1 (L1) cache** sits closest to the execution units and is split into an instruction cache (L1i) and a data cache (L1d). A typical core implements 32 KB of L1d and 32 KB of L1i, organized as an 8-way set-associative SRAM (64-byte lines), to keep lookup hardware manageable and collisions low. Because it's on the same silicon slice as the core's ALUs, L1 delivers hits in only about 4 CPU cycles (≈1 ns), making it the fastest buffer in the hierarchy.
  - The **Level 2 (L2) cache** is larger and a bit farther from the core, trading some speed for capacity. Desktop ("Skylake-S") designs often use a 256 KB private L2 per core with 4-way associativity, whereas server parts ("Skylake-SP") ramp that up to 1 MB per core with 16-way associativity. Despite the size increase, L2 hits still complete in roughly 10 cycles (≈3 ns) thanks to pipelined tag checks and high-speed SRAM .
  - The **Level 3 (L3) cache**, or last-level cache (LLC), is shared among multiple cores and provides a unified reservoir before main memory. Capacities range from a few megabytes up to dozens of megabytes—e.g. client CPUs may have ~2.5 MB per core (20-way inclusive on Skylake-S) or server CPUs ~1.375 MB per core (11-way non-inclusive on Skylake-SP), while AMD's Zen 2 CCX design offers 16 MB per four-core cluster with 16-way non-inclusive associativity. Access latencies jump to around 40

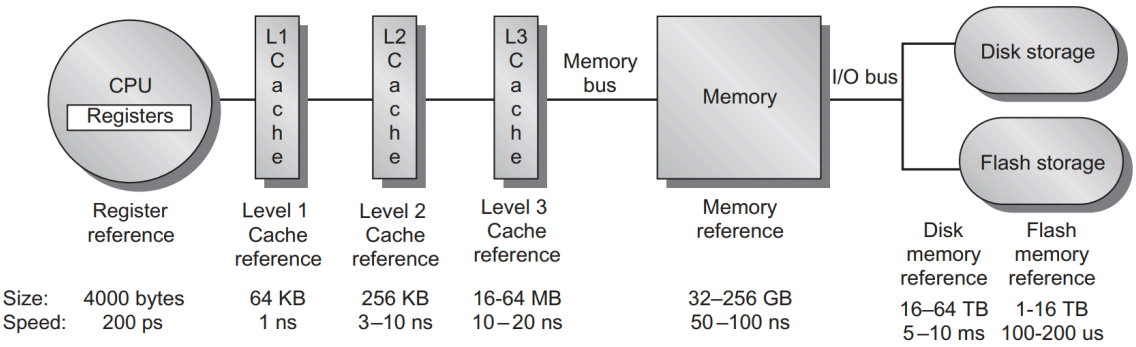
cycles for a local, unshared hit (≈12–21 ns) and as high as 65 cycles if the line resides in another core's slice .

- Beyond L3, very high-end or specialized processors sometimes include an **L4 cache**—often eDRAM on a separate die—with sizes in the tens of megabytes and latencies measured in hundreds of cycles, but three levels remain the industry norm.

Level	Typical Size	Associativity	Latency (cycles)	Latency (ns @3 GHz)
Registers	1000 Bytes	-	~1	~0.3 ns
L1	64 KB (32KB data and 32KB instruction	8-way	~1-3	~1 ns
L2	256 KB (per core)	4-way	~4-10	~3-10 ns
L3 (LLC)	8–32 MB (shared)	16–20-way	~30–40	~10–20 ns
L4 (eDRAM)*	64–128 MB	16-way (or higher)	~80–120	~27–40 ns
Main memory	8–64 GB (per channel)	N/A (flat DRAM array)	~180–300	~60–100 ns
Disk (HDD)	1–10 TB	N/A (sequential device)	~15–30 million	~5–10 ms



(B) Memory hierarchy for a laptop or a desktop



(C) Memory hierarchy for server

- See this video for more info.

- *Why did constructors decide to separate the L1 cache into instruction and data caches?*
  - Modern CPUs *pretend* to be von Neumann (one memory for code and data), but in practice it's faster to keep two tiny, separate L1 caches: one for **instructions** (code) and one for **data**. Code and data usually live in different memory regions and are used in different ways, so they don't really need to share the same small cache.
  - With separate caches, the CPU can in the same cycle:
    - fetch the next instructions from the **instruction cache**, and
    - load/store data from the **data cache**
    without the two fighting over the same memory ports. That gives higher bandwidth and lower latency than a single small unified cache.
  - There's also a bonus: the CPU can store **already decoded** or partially decoded instructions in the instruction side (decoding is slow), which speeds things up even more, especially after branch mispredictions when the pipeline has to be refilled.

- *How do you convert from cpu cyles to nanoseconds given a cpu with let's say 2 GHz?*
  - Use the clock period. If the CPU runs at fff hertz, one cycle takes $T = \frac{1}{f}$ seconds.
  - At **2 GHz** (2×10^9 cycles/s):
    - **1 cycle =** $1/2\ \text{GHz} = 0.5$ **ns**
    - To convert cycles → ns:

      $$\text{ns} = \text{cycles} \times \frac{10^9}{f_{\text{Hz}}} = \frac{\text{cycles}}{f_{\text{GHz}}}$$

      (because 1 GHz = 1 / ns).
  - Examples at 2 GHz:
    - 1,000 cycles → $1000/2 = 500$ ns
    - 3,000 cycles → $3000/2 = 1500$ ns $= 1.5\ \mu s$
  - See this thread for more info.

- *What are the difference between inclusive, exclusive and NINE caches?*
  - Cache inclusion policies govern whether data in a higher-level cache must, must not, or may or may not also reside in a lower-level cache. The three principal schemes—

inclusive, **exclusive**, and **non-inclusive non-exclusive (NINE)**—each trade off capacity, coherence complexity, and latency.

- **Inclusive caches** enforce that every line present in an upper cache (e.g. L1) also exists in the next level down (e.g. L2). On an L1 miss where the line sits in L2, the block is simply fetched into L1, and any evictions from L1 need not touch L2. However, when L2 evicts a line, it must invalidate that line in all higher caches to maintain the inclusion invariant. This simplifies coherence protocols—since the last-level cache (LLC) "knows" about all resident lines—but at the cost of wasted capacity due to duplication.
- **Exclusive caches** invert that relationship: no block appears in more than one cache level at a time. On an L1 miss but L2 hit, the block is moved from L2 into L1; if this displaces another line from L1, that evicted line becomes the fill for L2. Only evictions from L1 ever populate L2—main-memory fetches bypass L2 entirely—so the combined capacity of L1+L2 is effectively larger. This maximizes usable cache space but complicates lookups (requiring tag checks in multiple levels) and can increase miss-penalty latency.
- **Non-inclusive non-exclusive (NINE)** caches place no strict inclusion or exclusion constraints. On an L1 miss that hits in L2, the block is fetched into L1 without any eviction from L2. On a miss in both, a memory-fetched block is placed in both levels; on an eviction from L2, no back-invalidations occur. This flexibility lets architectures balance capacity and coherence overhead: some blocks may be duplicated for faster hits, others not, and coherence invalidations are only needed when true duplicates exist.
- In practice, inclusive hierarchies simplify multi-core coherence at the expense of cache capacity, exclusive hierarchies squeeze out every byte of storage but demand more complex tag searches, and NINE hierarchies sit between, offering a tunable middle ground that modern CPUs can adapt to workload patterns.
- See this video for more info.

- *What are the two types of cache locality?*
  - **Temporal Locality:**
    Temporal locality means that if a memory location is accessed, it is likely to be accessed again soon. Caches exploit this by keeping recently accessed data in a fast, nearby cache so that repeated accesses can be served quickly without going to slower main memory.
  - **Spatial Locality:**
    Spatial locality refers to the tendency to access memory locations that are close to one another. When one memory location is accessed, nearby locations are likely to be

accessed soon after. Caches take advantage of this by loading contiguous blocks of memory (cache lines) into the cache, so that when one element is accessed, the neighboring data is already available.

- See this video for more info.

- *Can you explain what is the difference between latency and throughput in a computer architecture context?*
  - **Latency** is the time it takes to complete one operation end-to-end (e.g., a cache miss taking 80 ns, a disk read taking 8 ms, or a single instruction's data dependency stalling 5 cycles), while **throughput** is the rate of completing many operations over time (e.g., 4 instructions retired per cycle, 100 GB/s memory bandwidth, 10,000 requests/sec).
  - In architecture, techniques like **pipelining**, **superscalar issue**, **out-of-order execution**, **vector/SIMD**, and **multi-core** often **raise throughput** by keeping more work in flight, but they may not improve—and can even slightly worsen—**per-item latency**.
  - Conversely, lowering **latency** usually means shortening the critical path or distance (e.g., hitting L1 instead of DRAM, reducing branch mispredicts, cutting cache miss penalties), which can help throughput indirectly but is a distinct goal.
  - Rule of thumb: latency is **how long one thing takes**; throughput is **how many things per unit time**—and the two can move in different directions (e.g., deeper pipelines: higher throughput, similar or higher latency).

- *How would you check what RAM/Cache latency is?*

```cpp
#include <bits/stdc++.h>
using namespace std;
using clk = chrono::high_resolution_clock;

double latency_ns(size_t bytes) {
    size_t n = bytes / sizeof(uint32_t);
    if (n < 2) return NAN;

    // Build a random "next" array to pointer-chase
    vector<uint32_t> idx(n); iota(idx.begin(), idx.end(), 0u);
    shuffle(idx.begin(), idx.end(), mt19937(1));
    vector<uint32_t> next(n);
    for (size_t i = 0; i + 1 < n; ++i) next[idx[i]] = idx[i+1];
    next[idx.back()] = idx[0];
```

```
    // Warm up
    volatile uint32_t p = 0;
    for (size_t i = 0; i < n; ++i) p = next[p];

    // Time one full lap (n dependent loads)
    auto t0 = clk::now();
    for (size_t i = 0; i < n; ++i) p = next[p];
    auto t1 = clk::now();

    double ns = chrono::duration<double, nano>(t1 - t0).count();
    return ns / n; // ns per load
}

double throughput_gbps(size_t bytes) {
    vector<char> A(bytes), B(bytes);
    // warm-up
    memcpy(A.data(), B.data(), bytes);

    auto t0 = clk::now();
    memcpy(A.data(), B.data(), bytes);
    auto t1 = clk::now();

    double sec = chrono::duration<double>(t1 - t0).count();
    return (bytes / 1e9) / max(sec, 1e-12); // GB/s
}

int main() {
    cout << "Latency (ns per load) as working set grows:\n";
    for (size_t kb = 4; kb <= 262144; kb *= 2) {  // 4 KB .. 256 MB
        double ns = latency_ns(kb * 1024);
        if (!isnan(ns)) cout << setw(7) << kb << " KB : " << fixed <<
setprecision(2) << ns << " ns\n";
    }

    cout << "\nThroughput with memcpy (bigger ~ RAM):\n";
    for (size_t mb : {8ULL, 64ULL, 256ULL, 1024ULL}) { // 8MB..1GB
        double gbps = throughput_gbps(mb * 1024ULL * 1024ULL);
        cout << setw(4) << mb << " MB : " << fixed << setprecision(2) <<
gbps << " GB/s\n";
    }
}
```

- The program measures two different things: **latency** and **throughput**. For latency, it builds a random “pointer chase” through an array and then follows that chain one step at a time. Each load depends on the result of the previous load, so the CPU can’t overlap or prefetch them; this exposes the true “time per load.” By repeating the test

with larger working-set sizes (from a few KB up to hundreds of MB), you see a staircase: fast when it fits in **L1**, slower once it spills to **L2**, slower again for **L3**, and slowest in **DRAM**. The random permutation prevents stride-based prefetchers from hiding latency, and a quick warm-up pass faults pages in so you're timing memory, not page faults. The number the code prints is essentially **nanoseconds per dependent load**, which jumps at the points where the working set no longer fits in the next cache.

- For throughput, it times a big **memcpy** between two large buffers. A sequential copy is perfect for hardware prefetch + vectorization, so it quickly becomes limited by **memory bandwidth** rather than compute. The code reports payload GB/s using the simple accounting "read B + write A = 2×size bytes." As the buffers grow far beyond last-level cache, the measured GB/s approaches your platform's sustainable DRAM bandwidth. Small warm-ups ensure the first measured copy isn't dominated by one-time effects.
- Key takeaways: latency is "**how long one dependent access takes**," revealed by pointer chasing over growing sizes; throughput is "**how many bytes per second we can stream**," revealed by a large memcpy. Expect clear steps in the latency results (L1 → L2 → L3 → DRAM) and a plateau in copy GB/s that reflects your memory subsystem.

- *Can you describe what is thread affinity?*
  - **Thread affinity** refers to the deliberate binding—or "pinning"—of software threads to specific processor cores or hardware threads in a multicore system. By fixing a thread to one core, the operating system's scheduler will always run that thread on the same core, preserving the contents of that core's private caches (L1 and L2) between context switches and avoiding the overhead of migrating the thread's working set to a different core. This can markedly reduce cache-miss penalties, TLB flushes and memory-access latency, especially on NUMA architectures where memory access time varies by the distance between core and memory bank. In performance-critical applications—such as high-throughput servers, real-time simulations or low-latency trading systems—thread affinity helps achieve more predictable timing and higher sustained throughput by maximizing data locality and minimizing inter-core communication.
  - Most operating systems expose APIs to control affinity. For example, on Linux you can use the `pthread_setaffinity_np` call along with a `cpu_set_t` bitmask to specify exactly which cores a POSIX thread is allowed to run on, and `sched_setaffinity` for whole processes. In C++ you might write:

```
cpu_set_t mask;
CPU_ZERO(&mask);
CPU_SET(3, &mask);                // bind to core 3
```

```
pthread_setaffinity_np(pthread_self(),
sizeof(mask), &mask);
```

- This ensures that subsequent scheduling decisions for that thread will confine it to core 3. Similar APIs exist on Windows (via `SetThreadAffinityMask`) and other platforms. By carefully assigning threads in this way—either statically or dynamically based on workload characteristics—developers can exploit cache hierarchy and NUMA topology to squeeze out maximum performance from modern multicore processors.
- See this video for more info.

- *What is the difference between cache affinity and cache coherence?*
  - **Cache coherence** refers to the mechanism that ensures all copies of shared data across multiple processor caches are consistent, meaning if one processor updates a piece of data, all other caches containing that data are updated as well: when more than one processor (or core or hyper-thread) is available and accessing the same memory it must still be assured that both processors see the same memory content at all times. If a cache line is dirty on one processor (i.e., it has not been written back yet) and a second processor tries to read the same memory location, the read operation cannot just go out to the main memory. Instead the content of the first processor's cache line is needed.
  - **Cache affinity** refers to the tendency of a process or thread to be preferentially scheduled on a specific processor core due to the data it frequently accesses already residing in that core's cache, allowing for better performance by minimizing cache misses

---

- *Can you explain how cache coherence is typically implemented? What is the MESI model?*
  - When you have several cores each with its own cache, they can easily disagree about the value of a memory location: core A writes `x = 5` in its cache, but core B still has `x = 3` in its own cache. Cache-coherence protocols fix this by making caches *talk to each other* about who owns which line and whether it's been modified.
  - Typical implementation (snooping idea): every cache line has a small *state* tag, and each core's cache controller "listens" (snoops) to a shared bus or interconnect. When a core reads or writes a line, or when it sees another core do so, it updates that line's state and sometimes sends messages (like "invalidate your copies of this line" or "I have the latest data, I'll supply it").
  - The MESI protocol is a very common way to label those states. Each cache line is always in exactly one of four states:

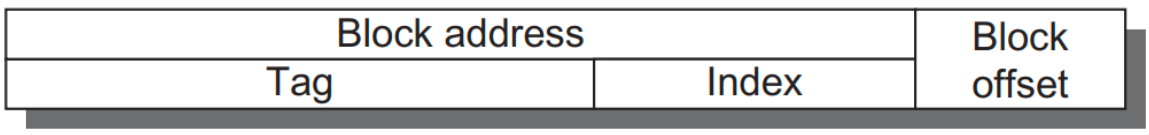
        - **M – Modified**: this core has changed the line; this copy is the only one that exists and is newer than main memory (must eventually be written back).
        - **E – Exclusive**: this core has the only copy, but it is *clean* (same as memory).
        - **S – Shared**: the line is clean and might also be present in other cores' caches.
        - **I – Invalid**: this line isn't valid in this cache (effectively "not present").
    - Very roughly, transitions work like this: lines are initially empty, hence start in **I**. A read miss brings a line in as **E** if nobody else has it, or **S** if others do. A core that wants to write will move its copy to **M** and force others to drop theirs to **I**. If another core later reads a line that's in **M** in your cache, your cache supplies the up-to-date data and both usually end up in **S**.
    - That way, at any moment, either there's one modified copy (M) that everyone agrees is the latest, or all existing copies are clean and identical to memory (E/S), and invalid copies (I) are ignored. This is how hardware keeps a consistent view of memory across multiple cores.
    - See this section 3.3.4 for more info. See this video also.

- *Can you explain what are RFO messages, how it relates to cache coherency and why they can significantly slowdown a program?*
    - An RFO is a **Read For Ownership** (also called "Read with intent to modify"). It is a special cache-coherence request a core sends when it wants to **write** to a cache line that it does not currently own in Modified/Exclusive state.
    - In a MESI-style protocol, a core is only allowed to modify a cache line if it is the **only** owner (state M or E). If a thread on core A does a store and the line is either not in A's cache or is only Shared, A's cache controller sends an RFO on the interconnect. That RFO says: "Give me this cache line and make me the sole owner." The line is then supplied either from main memory or from another core's cache, and all other cores that have that line must invalidate it (go to state I). After the RFO completes, core A has the line in M/E and can safely write.
    - RFOs hurt performance because they are much more expensive than a normal cache hit. They create extra traffic on the shared bus or mesh, they may require a cache-to-cache transfer from another core, and they force other cores to invalidate their copies. If several threads keep writing to different words that share the same cache line (false sharing), ownership of that line "ping-pongs" between cores: each store causes a new RFO, each RFO invalidates the others' copies, and everyone keeps stalling waiting for ownership. Even a simple store to a cold line in a write-allocate cache first has to **read** the whole line via an RFO before updating a few bytes, so the write miss turns into a read+coherence round-trip.

- So an RFO is the mechanism that enforces cache coherency for writes, but when it happens often—especially on contended or falsely shared lines—it can significantly slow a program by adding latency and burning interconnect bandwidth.

- *Since the DRAM channel (memory bus) is typically narrower than a cache line (for e.g. 64bits vs 64bytes) how is the cache populated?*
  - Think of DRAM as a “narrow but fast conveyor belt” and the cache line as a “big box” that has to be filled from that belt.
  - If a cache miss needs a 64-byte line and the external DRAM bus is 64 bits (8 bytes) wide, the memory controller simply issues a **burst** read of 8 transfers on that bus. Each transfer brings 8 bytes, and the cache controller stores them into the right positions in the 64-byte line:
    1. The core misses in L1 on address A.
    2. The cache asks the memory controller for the whole 64-byte line containing A.
    3. The memory controller opens the DRAM row and starts a burst: 8 data beats × 8 bytes = 64 bytes.
    4. As each 8-byte beat arrives, the cache line is filled piece by piece. Once all beats are in, the line is marked valid in the cache.
  - Many CPUs also use **critical-word-first / early-restart**: the 8-byte chunk that actually contains the requested word can be sent to the core as soon as it arrives, while the rest of the line is still being fetched, so the core doesn’t always have to wait for the full 64 bytes.
  - So even though the bus is narrower than a cache line, the cache is populated by a short burst of multiple transfers; hardware hides this detail, making it appear to the core as if an entire 64-byte chunk arrived more or less at once.

- *How are cache lines organized inside a cache?*
  - Caches are organized as **sets of lines** (a line = “block”). An address is split into three fields:



    - The **block-offset** picks a byte inside a line (size = line size, e.g., 64 B → 6 offset bits).
    - The **index** selects **one set** out of all sets in the cache.

    - The **tag** is stored with each line and is compared to decide if the selected set actually holds the requested block.
- In a **set-associative** cache there are N lines per set (the “associativity,” e.g., 2-way, 4-way, 8-way). A cache line maps to exactly one set—computed as

$$\text{set} = (\text{line address}) \bmod (\#\text{sets})$$

  —but it can occupy **any one of the N lines in that set**. On a lookup, the cache reads that set and compares all N tags **in parallel**; on a hit, the matching line supplies the bytes at the given offset. On a miss, the cache fetches the block from the next level and installs it **into some line in that set**, evicting one if needed (policy like LRU/PLRU/random). Each line carries **valid** and usually **dirty** bits (dirty ⇒ modified and must be written back on eviction).
- Two endpoints of this organization:
    - **Direct-mapped**: $N = 1$. Each set holds exactly one line; fastest/cheapest but most conflict-prone.
    - **Fully associative**: one big set with many lines ($N =$ total lines). A block can go **anywhere**; fewest conflicts but most expensive to search.
- Write policies pair with this structure: **write-through** updates lower memory on each write, while **write-back** marks the line dirty and writes it to the next level only on eviction. Both write strategies can use a **write buffer** to allow the cache to proceed as soon as the data are placed in the buffer rather than wait for full latency to write the data into memory.
- A quick sizing example: a 32 KiB cache, 64 B lines, 8-way associativity → $\#\text{sets} = 32\,\text{KiB}/(64\,\text{B} \times 8) = 64$; offset = 6 bits, index = 6 bits, remaining high bits are the tag.

- *What are the advantages/disadvantages of a directly-mapped cache vs an n-associative cache?*
    - **Direct-Mapped Cache:**
        - Advantages:
            - **Simplicity:** Each memory block maps to exactly one location, so the hardware for tag comparison is very simple.
            - **Speed:** Since there's only one candidate per access, tag comparisons and cache lookup are very fast.
            - **Cost & Power:** Less complex circuitry typically results in lower cost and power consumption.
        - Disadvantages:

        - **Conflict Misses:** If multiple frequently used blocks map to the same cache line, they continually evict each other, leading to higher conflict miss rates.
        - **Flexibility:** Limited placement options can degrade performance for certain memory access patterns.
    - **Associative Cache (n-way Set-Associative):**
        - Advantages:
            - **Reduced Conflict Misses:** With n slots (ways) per set, a memory block can be placed in any of the n locations, which helps avoid evicting other blocks that map to the same set.
            - **Improved Flexibility:** Better performance for workloads with non-uniform memory access patterns since blocks have multiple potential locations.
        - Disadvantages:
            - **Complexity:** More complex hardware is required to compare the tags in all n ways simultaneously.
            - **Slightly Slower Access:** The increased hardware complexity (multiple comparisons per lookup) can result in slightly longer access times.
            - **Higher Cost & Power Consumption:** More complex circuitry can lead to higher cost and increased power usage.

- *What are some cache replacement policies?*
    - Cache replacement policies decide which cache line to evict when a new block needs to be brought in but the cache is already full. One of the most common schemes is **Least Recently Used (LRU)**, which evicts the line that has not been accessed for the longest time. True LRU requires tracking exact access order for every cache line, which can be expensive in large associative caches; hardware often implements pseudo-LRU approximations (for example, tree-based bits) or the Clock (second-chance) algorithm, which gives each line a single “reference” bit and cycles through them, clearing bits on the first pass and evicting only when a line’s bit is already zero.
    - **First-In First-Out (FIFO)** is even simpler: it evicts the block that has been in the cache the longest, regardless of how often or how recently it was used. While this policy is trivial to implement with a circular queue, it can suffer from Belady’s anomaly, in which increasing cache size actually increases miss rate for some access streams. Random replacement picks a victim line uniformly at random; despite its simplicity and very low hardware cost, in many real workloads its average performance is competitive with more complex schemes and it never exhibits pathological anomalies.
    - **Least Frequently Used (LFU)** evicts the line with the smallest access count over time, which can be advantageous when certain data are “hot” for long stretches. However,

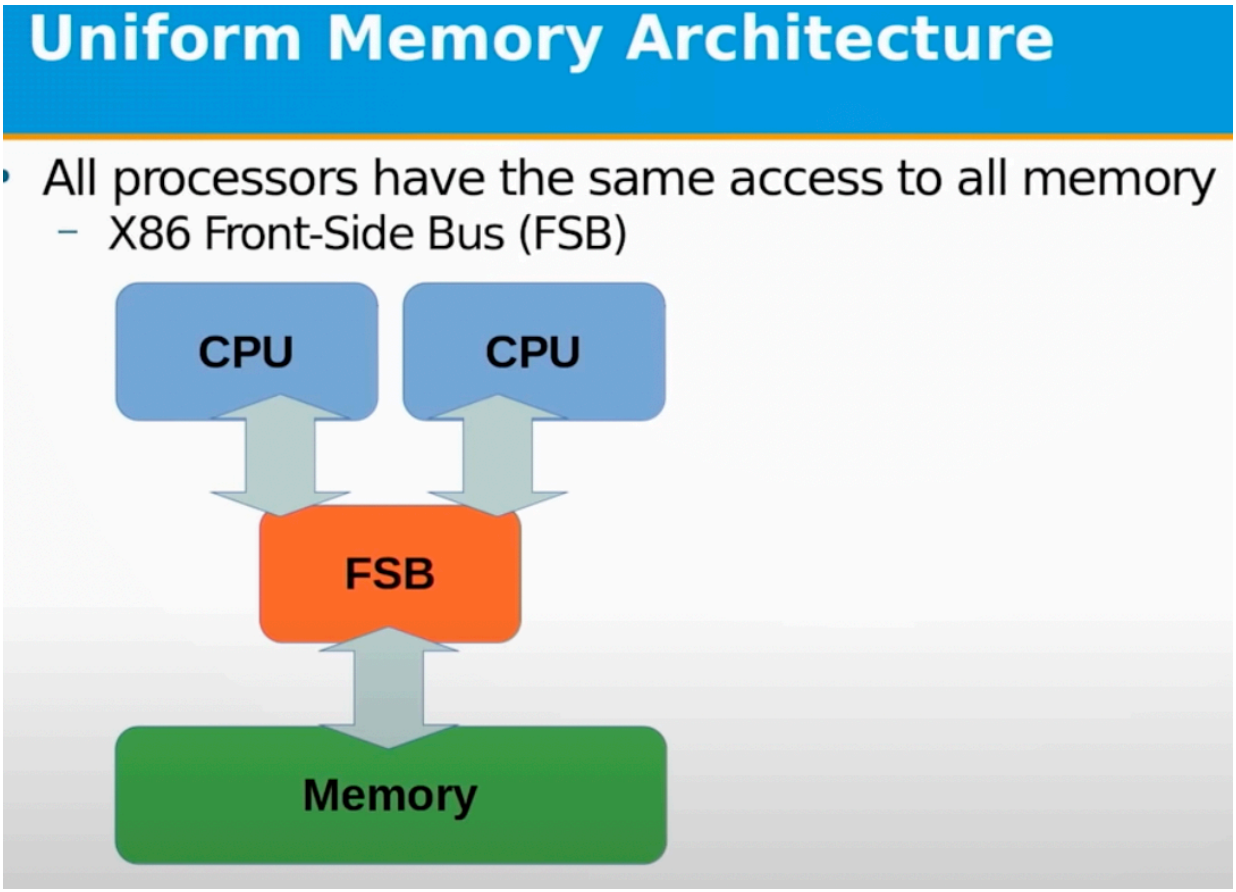
pure LFU can suffer from cache pollution by items that were once popular but are no longer used; sliding-window or “aging” counters are often added to bias the counts toward more recent accesses. Adaptive Replacement Cache (ARC) blends recency and frequency by maintaining two LRU lists—one for recently used lines and one for frequently used lines—and dynamically tuning their relative sizes to match the workload.

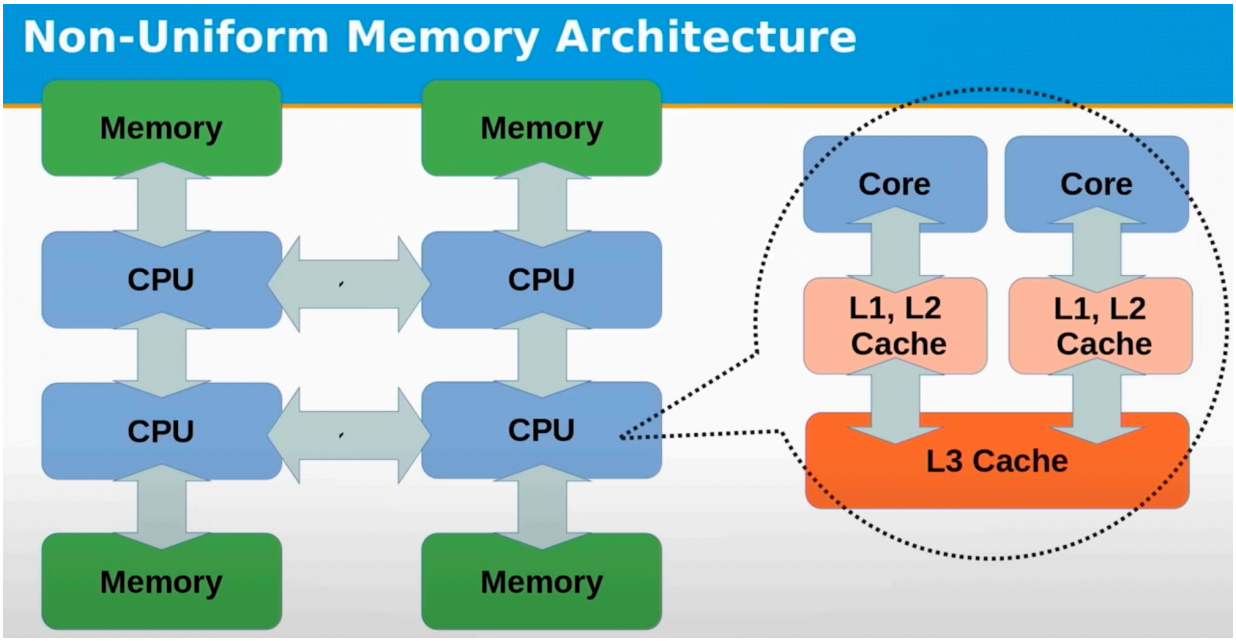
- **Segmented LRU (SLRU)** is a hardware-friendly variant that splits the cache into a small “probationary” segment and a larger “protected” segment: newly loaded lines enter probation, and only after they prove worth (by being re-accessed) do they migrate into the protected pool. This prevents a single burst of accesses from evicting long-lived, valuable lines. On the theoretical end, Belady’s OPT algorithm provides a lower bound on miss rate by evicting the block whose next use lies farthest in the future—but it is only implementable with perfect knowledge of future accesses, making it purely a benchmark for comparing practical policies.
- In modern on-chip CPU caches, designers almost always choose either a pseudo-LRU scheme (for direct-mapped or highly associative caches) or a simple Clock variant, since these strike the best balance between hit-rate efficiency and hardware complexity. Larger software-managed caches—such as those in operating-system buffer managers, database buffer pools, or web-proxy caches—can afford to experiment with LFU, ARC, or hybrid policies, tuning them to the specific access patterns of their applications.
- See this video for more info.

- *What is the NUMA architecture?*
  - **NUMA (Non-Uniform Memory Access)** is a memory architecture designed for multi-processor systems, where each processor (or group of processors) has its own local memory that it can access quickly, while memory attached to other processors (remote memory) can be accessed as well but with higher latency. In contrast, UMA (Uniform Memory Access) architectures provide a single shared memory space where all processors access memory at roughly the same speed regardless of which processor “owns” the memory. Here are some detailed differences and implications:
    - **Memory Latency and Bandwidth:**
      In a UMA system, every processor sees the same memory access latency because all memory is physically shared and managed uniformly. In NUMA systems, however, memory access times vary: accessing local memory is fast (low latency, high bandwidth), while accessing remote memory (memory local to another processor) is slower due to additional interconnect overhead.

- **Scalability:**
  UMA systems are simpler to program but can become bottlenecked as more processors compete for a single shared memory bus. NUMA architectures improve scalability by distributing memory among nodes, reducing contention. However, NUMA requires careful consideration of data placement and task scheduling to optimize performance.
- **Operating System and Application Implications:**
  On NUMA systems, the OS and runtime systems must be NUMA-aware. They try to allocate memory on the node where a thread is running and schedule threads close to the memory they frequently access (this is sometimes called “NUMA locality”). In contrast, UMA systems don’t have this issue because memory access is uniform.
- **Programming Considerations:**
  In UMA, programmers can generally assume that memory access cost is uniform across the system. In NUMA, however, performance can be significantly impacted by where data is placed relative to the executing thread. Poorly optimized NUMA applications might suffer from increased latency if they frequently access remote memory, whereas well-optimized code minimizes remote accesses.

- In summary, while UMA systems simplify memory access with uniform performance across all processors, NUMA systems provide better scalability and performance for larger multiprocessor setups by reducing contention—but they do so at the cost of non-uniform memory access times that require thoughtful software design to fully exploit local memory performance.





- *What are denormals?*
  - **Denormals** (or subnormal numbers) are floating-point values that are too close to zero to be represented in the normalized format. In IEEE 754, when a number is smaller than the smallest normalized value, it is represented in a denormalized (or subnormal) format with less precision.

- **What They Are:**
  In normalized numbers, the floating-point format uses an implicit leading 1 in the significand (mantissa). For very small values (closer to zero than what normalized numbers can represent), this implicit 1 is no longer assumed; instead, the significand starts with 0. This allows representation of numbers closer to zero, but with reduced precision.
- **Negative Impact on Performance:**
  Most modern CPUs have specialized hardware for fast floating-point arithmetic on normalized numbers. However, operations on denormal numbers often require extra handling—sometimes even falling back to slower microcode routines. This can cause significant performance penalties in numerical code, especially in tight loops or high-performance computing scenarios where a few denormals can slow down an entire application.
- **How to Solve It:**
  - **Flush-to-Zero (FTZ):** Many processors offer a mode where denormal numbers are flushed to zero, so instead of performing slow operations on them, the hardware treats them as zero. This mode can often be enabled via control registers (for example, using the SSE/AVX instructions on x86 with `_MM_SET_DENORMALS_ZERO_MODE` and `_MM_SET_FLUSH_ZERO_MODE`) or compiler flags (such as `-ffast-math` in GCC/Clang, though that flag affects many other floating-point behaviors too).
  - **Avoid Producing Them:** In some algorithms, you can restructure your code or add checks to avoid generating denormal values in the first place.
  - **Compiler Options:** Some compilers provide options to automatically flush denormals to zero, improving performance without needing manual changes in the code.

- *What would be the general steps for a CPU to modify a variable in memory (let's say an int)?*
  - When the CPU needs to modify an int in memory, it typically goes through the cache hierarchy rather than directly accessing main memory. Here's a simplified breakdown:
    1. **Cache Line Fetch:**
       The CPU first checks whether the cache line containing that int is already in its cache. If it isn't, the entire cache line is loaded from main memory into the cache. This cache line holds not only the int in question but also adjacent data.
    2. **Modification in Cache:**
       Once the cache line is present, the CPU performs the modification. In many cases, this involves loading the int from the cache line into a register, modifying it (for

example, incrementing it), and then writing the updated value back to the cache line. Alternatively, certain atomic instructions can perform a read-modify-write directly on the cache line without a separate explicit load and store.

3. **Write-Back to Memory:**
   With the modified value in the cache, the cache line is marked as “dirty.” In a write-back cache, this updated cache line will eventually be written back to main memory, while in a write-through cache the new value may be written to both cache and memory simultaneously.

This process leverages the CPU cache to avoid the high latency of main memory and to allow efficient, localized modifications. So while it might seem like the CPU “fetches, modifies, and then writes back,” these operations are optimized by the cache subsystem and the memory hierarchy, with hardware mechanisms (like store buffers and atomic instructions) ensuring correctness and efficiency.

- *What are some CPU cache write policies?*
  - CPU cache write policies determine how and when modified data in the cache is propagated back to main memory, and how the cache allocates space for stores. The two fundamental dimensions are **write strategy** (how writes update memory) and **allocation policy** (whether a store allocates a cache line on a miss).
  - **Write-Through vs. Write-Back**
    In a **write-through** cache, every write to a cached location is immediately sent to main memory (and usually also updates any higher-level caches). This guarantees memory and all caches remain coherent without extra bookkeeping, but it generates high write bandwidth and latency. By contrast, a **write-back** cache buffers writes in the cache and only writes the modified (“dirty”) line back to memory when it is evicted. This reduces memory traffic and can improve performance, but requires `dirty` bits per line and a mechanism (e.g., write-back buffers) to manage outstanding writes and maintain coherence.
  - **Write-Allocate vs. No-Write-Allocate**
    On a cache miss for a store operation, a **write-allocate** (also called fetch-on-write) policy brings the referenced line into the cache and then performs the write there. This is commonly paired with write-back, since subsequent stores to the same line stay in cache. In contrast, **no-write-allocate** (or write-around) writes the data directly to memory without caching that line; the cache remains unchanged. No-write-allocate is sometimes combined with write-through, avoiding pollution of the cache by lines that are only written once.
  - **Hybrid and Advanced Schemes**
    Some architectures mix these policies dynamically. For example, Intel CPUs often use

write-back with write-allocate for general stores but may employ write-through for certain cache levels or specific memory regions (such as device-mapped I/O). A **write-combine** buffer can coalesce multiple adjacent writes into a burst transfer to memory, improving throughput for streaming writes without allocating full cache lines. There are also exotic proposals like **write-once** caches, which write data to cache on the first store and then mark lines as read-only, or schemes that adapt between write-through and write-back based on workload characteristics or contention.

- **Coherence and Ordering Implications**
  These policies interact with cache-coherence protocols and memory ordering models. Write-through simplifies coherence (other cores see updates immediately), while write-back requires explicit snooping or directory mechanisms to handle dirty-line propagation. Likewise, write-allocate can increase coherence traffic if many stores thrash a cache line. Architects choose the combination of these policies to balance performance, complexity, and power: multi-level caches typically use write-back/write-allocate at the L1 and L2 levels for speed, but may adopt write-through at the L3 or memory controller interface to simplify the last-level cache coherence.
- See this video for more info.

- *What is a 'dirty bit'?*
  - A **dirty bit** is a one-bit flag associated with each cache line (or memory page in virtual-memory systems) that indicates whether the copy stored in the cache has been modified (“written to”) since it was loaded from main memory. In a write-back cache, whenever the CPU writes to a cached line, the dirty bit for that line is set. Later, if that line is evicted—because another block needs its space—the hardware checks the dirty bit: if it’s clear, the cache simply discards the line (since main memory already has the up-to-date data); if it’s set, the cache must write the modified data back to main memory before eviction. This mechanism avoids needless memory writes on every store (as would happen in a write-through cache) and dramatically reduces memory traffic and latency. Beyond caches, operating systems also use dirty bits in page tables to track which pages in RAM have been written to, so that only those pages need to be written out to disk when swapping, rather than the entire working set. This simple flag thus plays a crucial role in optimizing both memory-hierarchy performance and I/O efficiency.

- *Are the stack and heap shared between processes? Are they shared between threads within a process?*

- Each process typically has its own **separate virtual address space** that includes both its stack and its heap. The stack is used for function calls and local variables and is private to each thread within the process. The heap is the area for dynamic memory allocation (via functions like malloc or new) and is also private to the process by default. However, if needed, processes can explicitly share memory using inter-process communication mechanisms, but in a standard configuration, the heap is not shared between processes.
- In a multithreaded process, **each thread has its own dedicated stack**. These stacks are not subdivisions of one single, shared global stack; rather, they are independent regions allocated within the process's overall virtual address space. The operating system (or runtime) reserves separate memory areas for each thread's stack, ensuring that local variables and function call information remain private to each thread while all threads share the same heap.
- To go more in details, each thread has its own dedicated stack pointer that points to the top of its individual stack. In a multithreaded process, each thread's stack is allocated as a separate region within the process's virtual address space. When a thread is executing, its stack pointer (stored in a dedicated CPU register, like RSP on x86_64) is used to manage function calls, local variables, and other temporary data on that thread's stack. During a context switch, the CPU saves the current thread's stack pointer along with its other registers and later restores the stack pointer for the thread being resumed. There isn’t one global stack pointer for the whole process; instead, every thread has its own stack and its own stack pointer, ensuring that each thread's call history and local data remain isolated from the others. Although, threads have independent call stacks, the memory in other thread stacks is still accessible and in theory you could hold a pointer to memory in some other thread's local stack frame (though you probably should find a better place to put that memory!).

Per Process Items	Per Thread Items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	Thread state
Pending alarms	
Signals and signal handlers	
Accounting information	

- *What is the difference in speed between stack and heap?*

- From a purely hardware perspective, both stack and heap reside in the same main memory, so reading or writing a given address is equally fast once it's in cache. In practice, though, stack accesses often appear faster because:
  1. **Cache Locality:**
     The stack is typically used in a last-in-first-out pattern with small, localized regions of memory. This predictable usage often leads to better cache locality compared to the more scattered allocations on the heap.
  2. **Allocation Overhead:**
     Accessing an already-allocated heap variable is about the same speed as accessing the stack, but obtaining memory from the heap (e.g., via malloc or new) can involve more overhead than incrementing the stack pointer.
  3. **TLB/Address Layout:**
     The stack is usually in a single contiguous region, whereas heap allocations can be spread out. That can affect how frequently the CPU and operating system must update or look up page mappings.
- Once a variable is allocated and in cache, however, both stack and heap accesses are essentially the same speed at the hardware level. The primary difference comes from allocation patterns, cache usage, and the overhead associated with managing dynamic memory.

- *What data structure is typically used for the heap?*
  - The "heap" used for dynamic memory allocation isn't the same as the heap data structure (like a binary heap). Instead, the heap is a region of memory managed by the operating system's memory allocator. Internally, memory allocators typically use data structures to keep track of free and allocated memory blocks. Common techniques include:
    - **Free Lists:**
      A linked list (often doubly-linked) of free memory blocks. Many allocators use segregated free lists, where blocks are grouped by size into different bins. This makes it faster to locate a block of the appropriate size when a request comes in.
    - **Buddy Allocator:**
      A buddy system divides memory into blocks of sizes that are powers of two. When a block is allocated or freed, adjacent "buddy" blocks are merged or split accordingly. This method can simplify coalescing adjacent free blocks and managing fragmentation.
    - **Tree-based Structures:**
      Some allocators use balanced trees (such as red-black trees) to index free blocks

by size, which can allow for efficient searches when an allocation request is made.

- Different allocators (such as dlmalloc, ptmalloc, jemalloc, or tcmalloc) may use variations or combinations of these structures to optimize performance, minimize fragmentation, and suit their target workloads.

- *How would you measure cache latency on your architecture?*
  - One common way to measure cache latency is to write a microbenchmark that performs pointer chasing through an array, using a high-resolution timer (such as the RDTSC instruction on x86) to measure the number of CPU cycles per memory access. The idea is to construct a pointer-chasing loop that forces the CPU to load data from memory repeatedly and then compute an average latency per access. For example, you could use the following code:

```
#if defined(__i386__) || defined(__x86_64__)
static inline uint64_t rdtsc() {
    uint32_t lo, hi;
    __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
    return ((uint64_t)hi << 32) | lo;
}
#else
#error "This benchmark requires an x86/x86_64 architecture for rdtsc"
#endif

int main() {
    // Choose an array size that fits in the cache you want to test.
    // For example, for L1 cache, you might use a small array.
    const int size = 8 * 1024; // number of elements (adjust
according to cache size)
    std::vector<int> arr(size, 0);

    // Create a pointer-chasing loop: each element points to the next
index
    // and the last element wraps back to the first.
    for (int i = 0; i < size - 1; i++) {
        arr[i] = i + 1;
    }
    arr[size - 1] = 0; // circular list

    // Warm up: run the loop a few times to fill the cache.
    volatile int idx = 0;
    for (int i = 0; i < size; i++) {
        idx = arr[idx];
```

```
    }

    // Now measure latency using a large number of iterations.
    const int iterations = 10000000;
    uint64_t start = rdtsc();
    for (int i = 0; i < iterations; i++) {
        idx = arr[idx];
    }
    uint64_t end = rdtsc();
    double avgCycles = static_cast<double>(end - start) / iterations;
    std::cout << "Average cycles per access: " << avgCycles <<
std::endl;
}
```

- How This Works
  1. **Array Setup:**
     An array is allocated and arranged into a circular pointer-chasing loop. By choosing the size appropriately, you can target different cache levels (for example, a small array for L1 cache or a larger one for L2/L3).
  2. **Warm-Up Phase:**
     The warm-up loop fills the cache with the array data, so subsequent measurements are not skewed by cold cache misses.
  3. **Measurement Loop:**
     The main loop performs a fixed number of pointer chases, and the rdtsc instruction is used to record the number of CPU cycles before and after the loop.
  4. **Latency Calculation:**
     The difference (end - start) divided by the number of iterations gives an average cycle cost per memory access, which reflects the cache latency.
- Otherwise, you could also make two loops, one that access each element of that array randomly (and measure), and one that first iterates over each array elements to warm the cache, the reiterate over the array and measure that second iteration. Compare the results.

- *What is the advantage of 2's complement over 1s' complement in negative number representation in binary number system?*
  - 1's complement flips all the bits of a binary number (0 becomes 1 and 1 becomes 0), while 2's complement is obtained by taking the 1's complement and adding 1
  - The primary advantage of two's complement over ones' complement is that two's complement only has one value for zero. Ones' complement has a "positive" zero and

a "negative" zero.
- Next, to add numbers using one's complement you have to first do binary addition, then add in an end-around carry value.
- Two's complement has only one value for zero, and doesn't require carry values.

- *What is the system's kernel? What is its role?*
  - The kernel is the core component of an operating system, serving as the bridge between software applications and the hardware. It has full control over the system's resources and is responsible for several critical functions:
    - **Resource Management:** The kernel allocates and manages the system's resources—such as CPU time, memory, and input/output devices—to ensure that different processes and applications run efficiently and fairly.
    - **Memory Management:** It handles the distribution of memory for various processes, implements virtual memory, and ensures that one process does not interfere with the memory of another.
    - **Process Scheduling and Management:** The kernel oversees the creation, execution, and termination of processes. It schedules tasks so that each process gets the necessary CPU time to operate.
    - **Device Management:** Acting as an intermediary, the kernel communicates with hardware devices via drivers. This abstraction allows applications to interact with hardware without needing to know the specifics of the hardware details.
    - **System Calls and Security:** Applications interact with the kernel through system calls. The kernel validates these requests to maintain system stability and security by controlling direct hardware access.

- *What are the different segments that typically compose a process memory?*
  - A process's memory is commonly divided into five main segments, each serving a distinct purpose. Here's a detailed breakdown:

1. **Text (or Code) Segment**
   - **Contains:** The compiled machine code or executable instructions of the program.
   - **Details:** This segment is usually marked as read-only to prevent accidental modification or corruption of the code during execution. It may also be shared among processes running the same program, which is efficient for memory usage.
2. **Data Segment**
   - **Contains:** Global and static variables that have explicit initial values defined in the source code.

- **Details:** This segment holds variables that must be initialized to a specific value when the program starts. Unlike the text segment, the data segment is writable since it contains variables whose values can be altered during program execution.

3. **BSS (Block Started by Symbol) Segment**
   - **Contains:** Global and static variables that are declared but not initialized.
   - **Details:** The operating system initializes this segment to zero (or another default value) at runtime. By separating uninitialized variables into the BSS, the program's executable file can be kept smaller because it doesn't need to store large blocks of zeros.
4. **Heap Segment**
   - **Contains:** Dynamically allocated memory during runtime (using calls like `malloc`, `calloc`, or `new` in C/C++ and similar functions in other languages).
   - **Details:**
     - The heap starts at a certain memory location and grows upward as more dynamic memory is allocated.
     - Memory allocated here must be managed manually by the programmer (or automatically via garbage collection in some languages), which introduces risks like memory leaks or fragmentation if not handled properly.
5. **Stack Segment**

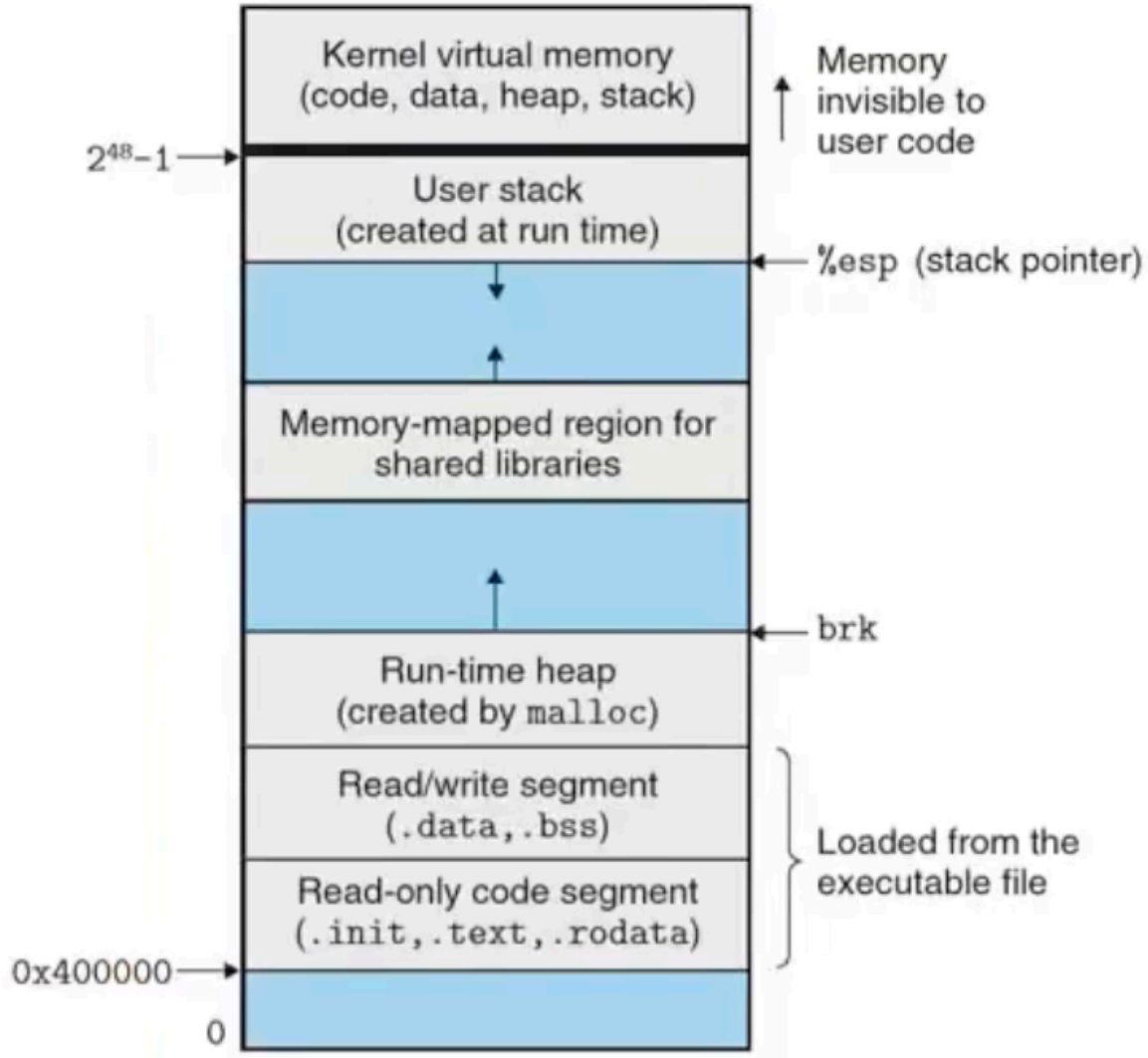
   - **Contains:** Local (automatic) variables, function parameters, return addresses, and control information for function calls.

   - **Details:**

   - The stack is typically organized in a Last-In, First-Out (LIFO) manner to efficiently support nested function calls.

   - It grows and shrinks as functions are called and return.

   - Its size is generally limited by the operating system, which means excessive use (e.g., deep recursion) can lead to a stack overflow.



- *Can you explain the concept of coalescing and splitting in the context of free memory management and allocators?*
  - In dynamic-memory allocators, **splitting** and **coalescing** are complementary strategies used to manage free blocks of memory efficiently and mitigate fragmentation.
  - When a program requests a chunk of heap memory, the allocator searches its free-list or free-map for a block at least as large as the request. If it finds a larger block than needed, it will **split** that free block into two pieces: one exactly the right size to satisfy the allocation and a second, smaller leftover block that remains free. By splitting, the allocator avoids wasting an entire large block on a smaller request, reducing internal fragmentation (wasted space within allocated regions). The leftover piece is typically returned to the free list, possibly coalesced later or used to satisfy future requests.

- Conversely, when the program frees a block, the allocator checks its immediate neighbors (the blocks just before and after in memory). If any of those neighboring blocks are also free, it will **coalesce** them into a single larger block. By merging adjacent free pieces, coalescing prevents the heap from becoming littered with many tiny free fragments that individually can't satisfy larger allocation requests, thus reducing external fragmentation (wasted space between allocations). Some allocators perform coalescing eagerly—immediately on each free—while others defer it or do it incrementally to balance the cost of boundary checks and list adjustments.
- Together, splitting and coalescing let the allocator adapt to varying allocation patterns: splitting supplies right-sized chunks on demand, while coalescing rebuilds larger contiguous runs when they become free. Properly tuned, they help maximize usable memory, minimize both kinds of fragmentation, and keep allocation and deallocation operations efficient.

- *Can you discuss some allocation placement strategies used to manage free space?*
  - Allocation placement strategies determine how an allocator chooses among available free blocks when satisfying a new request. The simplest is **first-fit**, where the allocator scans the free list from the beginning and picks the first block that's large enough. First-fit tends to be fast on average—since it stops as soon as it finds a match—and it leaves large free blocks near the front of the list for big allocations. However, it can lead to **external fragmentation** toward the front, as many small scraps accumulate there over time.
  - A close cousin is **next-fit**, which remembers where the last allocation search ended and begins the next search from that point. By cycling through the free list rather than always restarting at the head, next-fit spreads out fragmentation more evenly and can reduce search time when free blocks are clustered. Its downside is that it may skip over a perfectly good block near the front (because the "hand" has already passed it) and potentially cycle through the list more before finding a fit.
  - **Best-fit** takes the opposite approach to first-fit: it scans the entire free list and picks the smallest block that is still at least as large as the request. By doing so, best-fit aims to minimize wasted space within an allocated block (internal fragmentation), since it leaves the smallest possible leftover. In practice, though, best-fit often creates many tiny unusable fragments that can't satisfy future requests, and the full-list scan makes allocation slower—unless the allocator maintains sophisticated size-indexed data structures to quickly find the "just-big-enough" bin.
  - At the other extreme is **worst-fit**, which finds the largest free block and splits it for the allocation. The leftover tends to be large, so it can service many subsequent allocations without immediately fragmenting into unusable pieces. Yet because worst-

fit consistently splits the largest block, it can leave behind middling-sized holes that don't match typical request sizes, actually increasing fragmentation in realistic workloads. Like best-fit, it generally requires a global scan or a specialized max-heap structure to be efficient.

- In choosing among these strategies, allocator designers must weigh speed, fragmentation, and implementation complexity. First-fit and next-fit are simple and fast but risk uneven fragmentation, while best-fit and worst-fit aim to optimize space usage at the cost of slower searches and often counterintuitive fragmentation behavior. Many modern allocators therefore combine placement strategies with size-segregated free lists or indexed trees—using best-fit within a size class, for instance—to capture the benefits of each approach without their extreme drawbacks.

- *Does a context switch automatically imply a switch from user mode to kernel mode?*
  - **No.** A context switch and a mode switch are different things.
    - **Mode switch** = the CPU changes privilege level (user ↔ kernel). This happens on syscalls, traps, and interrupts—even if the *same* thread keeps running. No context switch is required.
    - **Context switch** = the OS scheduler picks a *different* runnable thread/process and loads its CPU state (registers, PC, stack pointer; possibly a new address space, etc.). This is independent of privilege level.
  - How they relate in common cases:
    - Switching between two **user processes/threads** is an OS context switch and is **initiated in kernel mode** (the scheduler runs in the kernel), so you enter kernel mode to perform it. But after the switch you may return to **user mode** (for a user thread) or stay in **kernel mode** (if the next runnable entity is a kernel thread).
    - You can have a **mode switch without a context switch**: a syscall or interrupt enters the kernel and returns to the same user thread.
    - You can have a **context switch without ending up in user mode**: e.g., switch from one kernel thread to another, staying in kernel mode.
    - User-level ("green") threading libraries can do **user-space context switches** entirely in user mode within one OS thread (saving/restoring registers and stacks themselves).
  - So: a context switch usually involves entering kernel mode to let the scheduler run, but it **does not automatically imply** a user↔kernel mode change as part of the *resulting* execution mode, and a mode switch does not automatically imply a context switch.

- *On a context switch, are the caches (TLB, L1, L2, L3) typically flushed?*
  - Short answer: **no**—ordinary context switches do **not** flush the data/instruction caches (L1/L2/L3) and they usually **don't flush the whole TLB** either. Caches hold *physical* lines and are kept coherent by hardware, so switching from one thread/process to another doesn't require invalidating them; the new task just experiences whatever cache state (hits/misses) is there, which is why context switching causes **cache pollution** but not a flush. For the **TLB**, modern CPUs tag entries with an address-space identifier (ASID/PCID). When the OS switches to a different process, it loads the new page-table pointer and ASID; the CPU then ignores TLB entries from other address spaces without flushing them, avoiding a big performance hit. If ASIDs aren't available or not used, switching CR3 (x86) *does* effectively invalidate most TLB entries, and page-table changes trigger targeted invalidations (e.g., `invlpg` /shootdowns) — but that's about correctness of mappings, not a blanket "flush caches on every switch."
  - **Exceptions/mitigations:** some security hardenings can add selective flushes: e.g., flushing **L1D** on certain kernel/user or VM boundaries on vulnerable CPUs (L1TF/MDS mitigations), or issuing **IBPB**/BTB barriers to clear branch-predictor state between protection domains. Those are policy-driven and relatively rare; they're not the default behavior of a normal timeslice context switch.

- *How does a TLB handle context switches?*
  - When the operating system switches from one process (or address space) to another, most of the virtual-to-physical translations cached in the TLB no longer apply, since they refer to the old process's page tables. Naively, the OS can simply flush the entire TLB—on x86 this happens automatically when you load a new page-directory base into CR3—so that no stale translations remain. This guarantees correctness but imposes a performance penalty, since every new memory access must repopulate the TLB, incurring extra page-table walks and latency.
  - To avoid that full flush, modern CPUs support tagging TLB entries with a small process identifier often called an **Address Space Identifier (ASID)**. Each translation is stored along with the ASID of the process that owns it, allowing entries for multiple processes to coexist in the TLB. On a context switch, the OS simply changes the current ASID; the hardware will automatically ignore entries whose tag doesn't match, effectively providing a fast "switch" without having to clear out the entire cache. Since the number of hardware ASIDs is limited, the OS must manage them carefully—reusing identifiers only after ensuring no live entries remain, or periodically performing a full flush when identifiers wrap around.

- *What are the different types of cache misses?*
  1. *Compulsory miss*: If the cache at level k is empty, then any access of any data object will miss. An empty cache is sometimes referred to as a cold cache, and misses of this kind are called **compulsory misses** or **cold misses**. Cold misses are important because they are often transient events that might not occur in steady state, after the cache has been warmed up by repeated memory accesses.
  2. *Conflict miss*: A **conflict miss** occurs when multiple memory blocks, which could all fit in the cache overall, continuously evict each other because they are forced to map to the same cache location (or small set of locations) by the **cache's placement policy**. For example, if a cache restricts blocks to specific slots using a modulo mapping (like block i maps to slot ($i \mod 4$)), then even though the cache can hold more blocks overall, repeatedly accessing two blocks that map to the same slot will cause each to be evicted when the other is loaded, leading to repeated cache misses.
  3. *Capacity miss*: Programs often run as a sequence of phases (e.g., loops) where each phase accesses some reasonably constant set of cache blocks. For example, a nested loop might access the elements of the same array over and over again. This set of blocks is called the **working set of the phase**. When the size of the working set exceeds the size of the cache, the cache will experience what are known as **capacity misses**. In other words, the cache is just too small to handle this particular working set.
     - To sum up, **compulsory misses** are those that occur in an infinite cache. **Capacity misses** are those that occur in a fully associative cache. **Conflict misses** are those that occur going from fully associative to eight-way associative, four-way associative, and so on.

- *Could you give the approximate number of cpu cycles taken to execute common operations (like +, %, division, etc).*
  - Here's a rough "cheat-sheet" of latencies (in clock-cycles) for common operations on modern x86-64 CPUs (e.g. Intel Haswell/Skylake family). These are **latencies** in a dependency chain; actual throughput can be higher for fully-pipelined ops.

Operation	Approx. Cycles	Notes & Sources
Integer		
Add / Sub	1 cycle	Latency of simple ALU ops is 1 cycle on all recent x86 CPUs superuser.com
Multiply (`IMUL`)	3 cycles	Fully-pipelined on many CPUs (throughput = 1 per cycle) superuser.com

Operation	Approx. Cycles	Notes & Sources
Divide / Modulo (`DIV` / `IDIV`)	~30 cycles (32-bit) 42–95 cycles (64-bit)	Integer division is microcoded and slow: Haswell ≈ 30 cycles; Skylake DIVQ ≈ 42–95 cycles forum.nasm.usstackoverflow.com
Bitwise AND/OR/XOR	1 cycle	Same simple-ALU latencies superuser.com
Shift (`SHL` / `SHR`)	1 cycle	Single-cycle barrel shifter superuser.com
Floating-Point / SIMD		
FP Add / Sub	3 cycles	Typical FADD latency ≈ 3 cycles on Haswell; throughput ≈ 0.5–1 cycle superuser.com
FP Multiply	5 cycles (Haswell) 3 cycles (other)	FMUL has ≈ 5 cycle latency on Haswell, though many older designs list ≈ 3 cycles superuser.com
FP Divide	20–40 cycles	Similar scale to integer divide, though exact depends on width and μarch; microcoded and high-latency
Memory & Caches		
Register access	0 cycles (free)	Values in registers are “free” beyond the ALU latency of the op
L1 load	1-3 cycles	L1 d-cache load-use latency on Haswell & newer is ≈ 4 cycles stackoverflow.com
L2 load	4-10 cycles	L2 cache hit ≈ 10 cycles stackoverflow.com
L3 load	30-40 cycles	L3 local hit ≈ 40 cycles stackoverflow.com
DRAM load (RAM)	~200 cycles	Main-memory latency ≈ 60 ns → ≈ 180–200 cycles at 3 GHz stackoverflow.com
Control Flow		
Branch (correctly predicted)	1 cycle	Predicted branches retire in ≈ 1 cycle stackoverflow.com

Operation	Approx. Cycles	Notes & Sources
Branch (mispredicted penalty)	12–20 cycles	Typical mispredict penalty ≈ 12 cycles (Haswell) up to ≈ 20 on older μarches; can be > 10 cycles stackoverflow.comstackoverflow.com

---

- *Why is division more expensive than multiplication?*
  - Because of how the hardware has to do the work.
  - Multiplication maps well to **parallel** hardware: the multiplier forms many partial products (via Booth/array/Wallace-tree logic) and reduces them in parallel with fast adders. It pipelines easily, so you can get **low latency and high throughput** (one result per cycle once the pipeline is full).
  - Division, by contrast, naturally looks **sequential**. Classic integer/FP dividers (restoring, non-restoring, SRT) determine the quotient **digit by digit**, updating a running remainder each step. That loop creates **data dependence** between steps, which is hard to pipeline deeply; many cores even make `div` **non-pipelined**, so a new division can't start until the previous one finishes. Even when FP division uses a faster scheme (e.g., compute a reciprocal with a table + Newton–Raphson or Goldschmidt iterations), you still need **several dependent multiplies/adds** to converge, so latency stays much higher than a single multiply.
  - Two more costs make division slower:
    - **Normalization and special cases:** IEEE-754 handling (NaN/Inf, subnormals, exact rounding) and integer corner cases (divide by zero, overflow) add logic on the critical path.
    - **Area vs speed trade-off:** Vendors spend silicon on multipliers because they're ubiquitous (MAC/FMA in DSP/ML). Dividers are used less often, so designs accept **longer latencies** to save area/power.
  - The upshot you'll see in practice: on many CPUs, a multiply has ~3–5 cycle latency with 1-per-cycle throughput; a divide often has **tens of cycles** of latency and **low throughput**. That's why compilers and hand-tuned code try to replace `x / c` by a **multiply by a precomputed reciprocal** ("magic" constant) and a shift, or in FP compute `y * rcp(x)` when accuracy constraints allow.
  - See this thread for more info.

- *Can you explain how the kernel allocates memory to user processes?*

- Every user process runs in its own isolated virtual address space that the kernel builds up out of a handful of logical regions—read-only text for code, initialized data for globals, a growable heap, one or more mapped libraries, and a downward-growing stack. None of these regions consumes physical RAM until the process actually touches pages within them.
- When a program calls **malloc** (or adjusts the heap via the **brk**/**sbrk** syscall) or explicitly requests memory via **mmap**, the kernel simply expands its metadata: it records a new “heap” or “mapped” segment in the process’s mm_struct (on Linux) or equivalent. No frames are bound, and page-table entries remain marked invalid.
- The first time the process loads or stores at any virtual address, the MMU traps to the kernel’s page-fault handler. The handler locates the corresponding VMA (a record of the region’s bounds and permissions), allocates a free physical frame, zeroes or populates it (from a file if file-backed), updates the page-table entry with the new frame’s number plus the appropriate permission bits, and returns to user mode. Execution resumes as if the page had always been there.
- Under the hood, free frames are managed by a **frame allocator** (in Linux, the “buddy” allocator, subdivided into zones such as DMA, normal, and high-memory). When asked for a page, it finds a suitably sized block of one or more contiguous frames, splits larger blocks as needed, and hands one off; freed pages are coalesced back up the buddy hierarchy to reduce fragmentation.
- On **fork**, the kernel uses **copy-on-write**: parent and child share the same read-only mappings, with PTEs marked copy-on-write. Only when either process writes to a shared page does a fault trigger allocation of a fresh frame and a byte-for-byte copy, letting both continue independently.
- If physical memory runs low, the kernel’s **page-reclaim** subsystem selects lightly used or “clean” pages to evict (writing them to swap if dirty), unmaps them from their processes, and frees their frames for new allocations. This automatic swapping keeps user processes supplied with pages even under contention, at the cost of increased latency for reclaimed pages.
- Together, these mechanisms—region bookkeeping, demand paging via page faults, a buddy-based frame allocator, copy-on-write for efficient forks, and background reclamation—allow the kernel to present each process with a large, contiguous address space while dynamically mapping and unmapping physical memory as workloads change.

- *Can you explain in details how the mmap() syscall works?*
  - When you call

```
void *addr = mmap(addr_hint, length, prot, flags, fd, offset);
```

your libc wrapper packages the six arguments and issues the `mmap` syscall. Control switches into the kernel's syscall entry point, which unpacks your arguments and invokes the core routine (typically `do_mmap_pgoff()`).

- First, the kernel must pick—or verify—the address range where the mapping will live. If you passed `MAP_FIXED`, it simply checks that the requested region doesn't overlap any existing mappings (otherwise it unmaps or fails). Otherwise it calls `get_unmapped_area()`, which scans the process's existing VM regions (`struct mm_struct` → its red-black tree or list of `vm_area_struct`s) to find a suitably aligned hole large enough for `length` bytes. Once an address is chosen, the kernel allocates and initializes a new `vm_area_struct`: it records the start/end addresses, the protection bits (`PROT_READ/WRITE/EXEC`), the sharing mode (`MAP_SHARED` vs. `MAP_PRIVATE`), and, if file-backed, the pointer to the open `struct file` plus the page-offset. That VMA is then spliced into the process's memory map and the MM's statistics and resource counters are updated.
- Crucially, no physical memory is touched at this point—pages are mapped lazily. When your code first reads or writes anywhere in the new region, the CPU raises a page fault. The fault handler examines the faulting address, looks up the enclosing VMA (to check permissions and ownership), and then either:
  - for an anonymous (`MAP_ANONYMOUS`) mapping, allocates a zeroed frame from the buddy allocator;
  - for a file-backed mapping, grabs the page from the page cache (reading it from disk if necessary);
  - for a private mapping (`MAP_PRIVATE`), may copy the page on write (COW).
    In each case it installs a new PTE with the correct PFN and permission bits, resumes the user instruction as if the page had always been present, and thus hides all of this behind the illusion of a contiguous, backed virtual memory region.
- Flags like `MAP_POPULATE` (prefault all pages now), `MAP_LOCKED` (pin pages in RAM), `MAP_HUGETLB` (use huge pages), or `MAP_SYNC` / `MAP_SHARED_VALIDATE` (synchronize with underlying storage) tweak both the VMA setup and how the fault handler behaves, but the overall pattern is always the same:
  1. Reserve and record the address range and metadata in a VMA,
  2. Delay any heavy work until the first access fault,
  3. Populate PTEs on demand,
  4. And later, when you `munmap()`, tear down the VMA, flush TLB entries, and drop any associated page-cache references.
- This combination of lazy allocation, file-backing vs. anonymous zeroing, copy-on-write, and deferred page-fault handling is what makes `mmap()` both flexible and efficient for

everything from shared libraries to huge sparse data structures.

- On most Unix-style systems, a process's virtual address space is carved into a sequence of distinct regions, each tracked by its own VMA. At the very bottom live the executable's static segments: first the read-only text (.text) for code, then the initialized data (.data), and immediately after that the zero-filled BSS. Following those comes the heap, which starts just above BSS and grows upward as you call brk()/sbrk().
- All of your mmap() allocations then occupy their own VMAs somewhere above the heap. By default the kernel picks a free hole "high up" in the address space—typically between the heap and the stack—and hands you that range. You can have multiple anonymous mappings (and file-backed mappings) interleaved here, each one independently created and destroyed without touching the heap's break pointer.
- Above your mmaps usually sit any dynamically loaded shared libraries (which themselves are implemented via mmap), and finally a guard page, then the process's stack growing downward from the very top of user space. That means in a typical /proc/‹pid›/maps snapshot you'll see, in ascending order:
  – `.text` → `.rodata` → `.data` → `.bss` → `[heap]` → one or more `[anon]` or file-backed VMA's from mmap() → shared-object mmaps → `[stack]`.
- Because heap and stack grow toward each other—and mmap regions can be placed anywhere there's room—you won't normally see your heap and then immediately "the stack." Instead, your anonymous mappings sit in between, giving you maximum flexibility over where and how large each region can be.
- See this and this for more info.

- *What is the difference between brk/sbrk and mmap?*
  - The C library—and by extension most user-level allocators—has two fundamentally different ways to ask the kernel for more memory:
    1. **`brk()` / `sbrk()` (the "heap"):**
       This mechanism moves a single pointer—the program's "break"—up or down to grow or shrink the process's data segment in one contiguous chunk. When you do `sbrk(n)`, the kernel simply records that the heap now extends by `n` bytes; no physical RAM is assigned until you actually touch those addresses. Because it's one big region, you can only ever return memory at the very top (you can't carve out the middle of the heap), and you can't have two separate heap regions—everything lives in that one growable space.
    2. **`mmap()` (memory-mapping):**
       This creates an entirely new virtual-memory area (VMA), which the kernel tracks independently of the heap. You can map anonymous memory (`MAP_ANONYMOUS`) or file-backed pages, at any address (subject to alignment and existing mappings).

Each mapping is its own slice of the address space, so you can create multiple discontiguous regions and later release any of them with `munmap()`. Just as with the heap, no RAM is committed until you access the pages, but unlike `brk`, you get full flexibility over where and how large each mapping is.

3. **Granularity and freeing:**
    - **Heap (`brk`)** grows and shrinks only at its end, so fragmentation within the heap can trap free memory unless it's at the top. Shrinking with `brk` only works if you free everything allocated since the previous break point.
    - **`mmap`** lets you return arbitrary chunks back to the kernel via `munmap()`, so you can avoid long-term fragmentation by giving back exactly the pages you no longer need—even in the middle of your address space.
4. **Performance and use cases:**
    - **`brk`** is very fast for small, frequent allocations because it's just pointer arithmetic in the kernel and cheap bookkeeping in the malloc implementation. It's the default for most small `malloc()` requests.
    - **`mmap`** has more overhead up front (allocating a VMA, updating the process's MM data structures), so it's typically used for large allocations (often sizes above a tunable threshold, say 128 KB or 256 KB) or for special needs like shared memory, file mapping, or huge pages.
5. **Permissions and backing:**
    - **Heap pages** are always anonymous, zero-filled on first touch.
    - **`mmap`** can give you read-only, read-write, or executable regions, and can map files so that modifications either go back to disk (`MAP_SHARED`) or stay private (`MAP_PRIVATE`).

- In practice, a modern allocator will intermingle both: it will steal from the heap for most small requests, but fall back to `mmap` whenever it makes sense to keep large chunks separate, return them promptly, or apply special protections. See this and this for more info and this video, this video and this video.

- *Why is malloc typically implemented with a call to brk for small allocations, and mmap otherwise? Why not always use brk (or always mmap)?*
    - Allocators split work between `brk` and `mmap` because each system call has very different trade-offs. `brk` grows a single contiguous "heap" segment at the program break; the allocator then slices that big region into many small blocks in user space. For small allocations this is fast and cache-friendly because a single `sbrk/brk` can amortize thousands of `malloc` calls with no further kernel crossings, and all the allocator's bookkeeping and reuse (bins, freelists, coalescing) happens locally. **The downside is that memory obtained via `brk` can only be returned to the OS from**

the *top* of the heap; holes in the middle caused by frees cannot be given back, so long-running programs suffer from heap fragmentation and growing RSS even if they free many small blocks. Expansion can also fail if something else has mapped memory right above the heap, because `brk` needs contiguous virtual space.

- `mmap`, in contrast, creates independent mappings anywhere in the address space. Large allocations are a great fit: each big block can be given back precisely with `munmap`, so you avoid bloating the main heap and you can truly return unused pages to the OS, reducing fragmentation and peak RSS. You also get nice properties like page alignment, optional guard pages, and better ASLR dispersion for large chunks. But using `mmap` for *every* small `malloc` would be expensive: each call crosses the kernel, creates a separate VMA with kernel metadata, contends with limits on the number of mappings, and can trigger more TLB/VMA management work. Thousands of tiny `mmap`s are measurably slower than carving from one big `brk` region and can stress kernel bookkeeping.
- That's why mainstream allocators use a hybrid with a size threshold: small requests come from the contiguous `brk` heap to minimize syscall and metadata overhead, while large requests use `mmap` so they can be independently returned and won't fragment the heap. Using only `brk` would fragment and "leak" address space over time; using only `mmap` would make small allocations disproportionately costly and create a huge number of mappings. The split gives you the best of both: speed for small, precise release for large.

- *Can you explain what is the buddy algorithm?*
  - The **buddy algorithm** is a fast, simple way to manage physical (or virtual) memory in power-of-two–sized chunks. The entire addressable memory region is viewed as one large block of size 2^U (the maximum "order"). For each order k (0 ≤ k ≤ U), the allocator maintains a free list of blocks of size 2^k. To satisfy a request of size S, it rounds S up to the next power of two, say 2^k, then searches the free list at order k. If that list is nonempty, it removes and returns one block. If it's empty, it looks at the next larger order (k+1), and so on, until it finds a nonempty list at order m > k.
  - At order m, the allocator removes a block of size 2^m and splits it into two "buddies" of size 2^(m–1). One half is placed back on the free list for order m–1, and the other is either returned (if m–1 == k) or further split. This splitting continues until a block of the desired size 2^k is produced. Because each split divides a block evenly, the two resulting buddies are guaranteed to be aligned on 2^m boundaries and to pair up neatly for recombination later.
  - When freeing a block of size 2^k at address A, the allocator computes its buddy's address by flipping bit k in A's binary representation—that is, buddy_address = A ⊕ (1

<< k). It then checks the free list for order k: if that exact buddy block is present, the allocator removes the buddy from the list, merges the two into a single block of size 2^(k+1) at the lower of the two addresses, and repeats this coalescing process at order k+1. This “merge-and-climb” continues until either the buddy is allocated (or in a different order’s list) or the maximum order is reached.

- Because both allocation and deallocation inspect at most one free list per order—and each order is visited at most twice (once on a split, once on a coalesce)—operations run in O(U) time where U is the number of orders (logarithmic in total memory). The buddy system thus offers a good balance of speed, simplicity, and low fragmentation overhead, at the cost of internal fragmentation when requests aren’t exact powers of two and of keeping separate free lists for each order.
- See this video for more info.

- *What is the difference between multithreading and multiprocessing?*
  - Multithreading and multiprocessing both let you exploit parallelism, but they differ fundamentally in how execution contexts are organized, how resources are shared, and how isolation and communication work.
  - A **process** is an instance of a running program with its own private virtual address space, file descriptors, and kernel resources. In a **multiprocessing** model, each CPU core (or hardware thread) runs a separate process. Because processes don’t share memory by default, they’re isolated—if one crashes or corrupts memory, others stay unaffected. Communication between processes requires explicit kernel-mediated mechanisms (pipes, sockets, shared memory segments, message queues), which carry context-switch and copy-overhead. Creating or destroying a process is relatively heavyweight: the kernel must allocate a new `task_struct`, duplicate or copy-on-write its page tables, file tables, and other resources, and schedule it separately.
  - A **thread** is a lighter-weight execution context within a process: multiple threads share the same address space, open files, and other process-wide resources, but each has its own registers, stack, and thread-local storage. **Multithreading** lets threads run concurrently—often on multiple cores—but without the full isolation of separate processes. Because threads share memory, communication is as simple as reading and writing shared variables, but it requires careful synchronization (locks, condition variables, atomic operations) to avoid races. Creating or switching between threads is cheaper than with processes: the kernel reuses most of the parent process’s data structures and only allocates a small `task_struct` and stack for the new thread.
  - In practice, multiprocessing shines when you need fault isolation, separate privilege domains, or to leverage multiple independent applications in parallel. It’s common in microservices architectures or when running untrusted code. Multithreading excels

when you need fine-grained parallelism within a single application—e.g. handling many network connections in parallel or dividing a computation across cores—because threads can communicate and coordinate with far lower overhead than processes.
- Ultimately, the choice depends on your requirements for isolation versus efficiency, ease of data sharing versus the risk of synchronization bugs, and the cost you're willing to pay for context switches and inter-execution communication.

- *What is the difference between a hardware thread and a software thread?*
  - A **software thread** is an abstraction provided by the operating system (or by a user-level threading library) that represents a single sequence of programmed instructions within a process. Each software thread has its own logical context—its own stack, register set, and thread-local storage—but shares the process's address space, file descriptors, and other resources with its sibling threads. The OS scheduler (or a user-level scheduler) decides when each software thread gets to run on the CPU, preempting and resuming threads as needed to give the illusion of concurrent execution even on a single core.
  - A **hardware thread**, by contrast, is the processor's actual physical (or logical) execution context. In a traditional single-threaded core, there is just one hardware thread: one set of instruction-fetch/decode/execute pipelines plus one register file. Modern CPUs often implement Simultaneous Multithreading (SMT)—Intel's Hyper-Threading or AMD's "SMT"—where each physical core provides two (or more) hardware threads. Each hardware thread has its own program counter and architectural state, allowing the core to keep its execution units busy by quickly switching between ready contexts without involving the OS.
  - In practice, software threads must be scheduled onto hardware threads. If you create more software threads than there are hardware threads, the OS kernel time-slices them on each hardware context, saving and restoring their register state on every switch. If you have at least as many hardware threads as software threads, each can run truly in parallel (subject to core-resource sharing). Thus, software threads are the units of work you program against; hardware threads are the concrete resources that execute that work.

- *Does each CPU core has its own mmu?*
  - Yes—every hardware thread (core, or SMT "hart") comes with its own dedicated MMU logic. In practice, that means each core has its own:

    - **Page-table base register(s):** when the OS switches contexts, it loads the new process's page-directory (or PML4) pointer into the core's register (CR3 on x86), so each core independently knows where to look for translations.
    - **TLB hierarchy and page walker:** instruction- and data-TLBs (and any shared L2 TLB) sit on the core, as does the logic that handles misses by walking the page tables in memory.
  - Because these structures live inside the core, translations and protection checks happen entirely locally, and one core's view of "which pages are mapped where" can't accidentally leak into another core's MMU. When the OS changes a PTE, it has to send invalidations (a "TLB shootdown") to every core that might have cached that mapping. That shootdown sequence flushes or updates each core's MMU state so they all see the new mapping.

- *We know that the way data is stored in caches is in cache lines (which are around 64 bytes). What happens if you try to store an object which size is greater than a cache line (e.g. a large array)? Will it be separated in multiple cache lines? So what happens during a capacity miss? Does this mean that what you are trying to store is bigger than the entire cache as a whole?*
  - Cache hardware only ever moves and stores data in fixed-size lines (commonly 64 B). If you have an object larger than one line—say a big array—that array simply spans multiple consecutive cache lines in memory. On first access to any part of the array, the CPU will load the 64 B line containing that address into the cache. As your code steps through the array past that 64 B boundary, each new line is brought in independently, so the array ends up "striped" across as many cache lines as needed.
  - A **capacity miss** happens when your cache simply isn't large enough to hold the processor's current *working set* of data, not because your object is literally bigger than the entire cache, but because over time you've touched more unique cache lines than the cache can accommodate.
  - Imagine you're streaming through an array whose total size exceeds the cache's total capacity. Even if every access is to a distinct, well-aligned cache line (so there are no conflict or alignment issues), once you've brought in more lines than the cache can store, the oldest lines will be evicted to make room for the new ones. Those evicted lines will cause misses if you revisit them—these are capacity misses, because the cache ran out of space.
  - In contrast, a single object that's larger than the cache will indeed span more lines than the cache can hold, so accesses to that object will incur capacity misses as you move through it. But you can also get capacity misses on a smaller object if your program's

overall working set includes other data or code that, together with that object, exceeds cache capacity.
    - Put simply: capacity misses reflect the mismatch between *how much* data you need close at hand and *how much* the cache can actually store, not a requirement that any one allocation be bigger than the entire cache.

- *What kind of cache replacement policy is typically used for a TLB?*
    - TLBs almost never track exact access counts (so you won't see an LFU scheme) and true LRU is usually too expensive in hardware for even modestly associative—or fully associative—buffers. Instead, most designs use an **approximate LRU** policy (often called pseudo-LRU) or even **random** replacement as a low-overhead alternative.
    - For example, the UltraSPARC-II's 64-entry fully-associative TLB implements a tree-based pseudo-LRU algorithm to pick victims, giving a good balance between recency bias and hardware simplicity. More generally, ARM and Intel cores similarly employ either PLRU variants—where a small number of bits per set track "which half was used last" in a binary tree—or simple random eviction (especially in unified second-level TLBs) rather than counting-based schemes. This keeps the critical TLB lookup path fast and avoids the extra storage and update logic that LFU would require.

- *Why do we need page tables at all?*
    - Page tables exist to give each process the illusion of a large, contiguous address space while actually sharing and protecting real physical memory. Without them, every program would have to be linked to fixed physical addresses, making it impossible to load multiple programs safely (they'd collide in memory) or to isolate one process's data from another's.
    - By breaking both virtual and physical memory into uniformly sized pages and using a page table to map each virtual-page number to a physical-frame number, the OS can:
        - **Relocate** a program at load time (or even after), since all pointers become virtual addresses that the MMU remaps on the fly.
        - **Protect** memory by marking pages read-only or no-execute, and by keeping each process's page tables private so one process can't stomp on another's pages.
        - **Share** code or data (e.g., shared libraries) simply by pointing multiple processes' page-table entries at the same physical frames, avoiding duplication.
    - Page tables also support **sparse** address spaces: you only need to allocate page-table entries (and physical frames) for the parts of the address space a process actually uses. Large gaps (holes) in a program's layout—unused addresses between

segments, or on-demand stacks and heaps—don't waste page-table memory because the OS simply leaves those entries marked invalid until you touch them.
- Finally, page tables are what make advanced features like **copy-on-write** forks, file-backed `mmap()`, guard pages, and fine-grained per-page permissions possible. All of these rely on the OS's ability to change just one entry in the table—mark a page read-only, point it at a file, or install a new frame on fault—without rearranging the rest of memory. In short, page tables are the central data structure that lets the OS virtualize, protect, share, and manage memory at the granularity of pages.

- *Is it up to the OS to decide which process is being run on which physical thread? If a process is changed from one thread to another, does this mean it has to flush the content of the original thread and repopulate the cache of the new thread?*
  - The mapping of software threads (or processes) onto hardware threads (cores or SMT "harts") is entirely the job of the OS scheduler. When a thread becomes runnable, the scheduler picks one of the available hardware threads, saves the outgoing core's architectural state (its registers, program counter, etc.) to that thread's kernel thread-control block, and then restores the incoming thread's saved registers before letting it execute.
  - By contrast, the CPU's caches (L1 instruction/data, L2, etc.) are tied to the physical hardware thread itself and are *not* flushed on every context switch or migration. If you migrate a thread from Core A to Core B, Core B's caches still contain whatever lines were there from whatever ran previously—it doesn't get zeroed out for your thread. Your thread simply starts with a "cold" cache on Core B: as it touches addresses, those lines are loaded (and may evict whatever was there). No explicit flush or "repopulation" step is required—or normally performed—by the OS.
  - The only per-core hardware structure the OS typically invalidates is the TLB (or individual page mappings via shootdowns) when changing address spaces; the data-cache remains intact for performance. In short:
    - **OS decides** which software thread runs on which hardware thread.
    - **Registers** are saved/restored at each context switch, but **caches stay resident**.
  - **Cache "warm-up"** on the new core happens naturally as your code accesses memory; there's no mandatory flush and reload.

- *If a process is currently running, is it guaranteed to stay on the same physical core throughout the run time? Or could the process be migrated to another physical in the middle of the process execution?*

- No—there's no luxury of "pinning" a runnable process to one core unless you explicitly set its CPU affinity. The kernel's scheduler runs on every CPU (invoked on each timer tick or when tasks wake up) and continuously balances work by moving tasks onto idle or less-loaded cores Stack Overflow. By default it also uses "natural" affinity—trying to keep a waking task on the same CPU to benefit from still-warm caches—but when another core is under-utilized it will migrate the task over to even out the load Server Fault.
- If you need a hard guarantee, you can bind your process or thread to specific CPUs (via `sched_setaffinity`, `taskset`, or corresponding APIs). Otherwise any running thread can be rescheduled on a different physical core at the next context-switch point.

- *What is the prefetcher?*
  - The **prefetcher** is a hardware mechanism inside modern CPUs that attempts to predict which memory locations a program will need in the near future and proactively fetches them into cache before the CPU actually requests them. Its purpose is to hide the long latency of main memory accesses, which can take hundreds of CPU cycles, by overlapping computation with memory transfers. When the CPU eventually issues a load instruction for that data, it ideally finds it already in the cache thanks to the prefetcher, resulting in a "prefetch hit" and much faster execution.
  - Prefetchers operate by observing patterns in memory access behavior. The simplest kind is the **sequential prefetcher**, which assumes that memory will be accessed linearly, as in array traversals. More sophisticated designs include the **stride prefetcher**, which detects regular gaps (e.g., accessing every 8th element) and the **correlation prefetcher**, which tries to detect complex, program-specific patterns. Prefetchers can target different cache levels: an **L1 prefetcher** brings data into the first-level cache very aggressively, while **L2 or L3 prefetchers** are more conservative, pulling data into higher levels of the hierarchy to reduce bandwidth waste.
  - While prefetching improves performance when predictions are accurate, it can backfire if the CPU fetches data that is never used, a situation known as **cache pollution**. This wastes cache space and memory bandwidth, potentially displacing useful data. To mitigate this, modern CPUs employ adaptive prefetching strategies, where the prefetcher monitors its accuracy and throttles itself when it generates too many useless requests.
  - In short, the prefetcher is a predictive engine inside the memory hierarchy that speculatively loads data into cache based on access patterns, thereby reducing effective memory latency. It is not visible to the programmer at the instruction set level, but it is critical for sustaining high performance in workloads with predictable memory

access patterns such as scientific computing, multimedia processing, and large-scale data traversal.

- *What happens, in branch prediction, when the CPU predicts the wrong path of execution?*
  - When the CPU's branch predictor makes a wrong guess, the processor has speculatively executed instructions along the incorrect path. These instructions are fetched, decoded, and often partially executed before the actual branch outcome is known. Once the CPU discovers that the prediction was incorrect, usually when the branch instruction itself reaches the point in the pipeline where its condition is resolved, it must undo all the speculative work done along the mispredicted path. This process is called a **pipeline flush**.
  - During a flush, the CPU discards all instructions in the pipeline that belong to the wrong path, and it clears any intermediate results that have not yet been committed to the architectural state (the official register and memory values visible to the program). Modern CPUs maintain mechanisms such as the **reorder buffer** to ensure that speculative execution never alters visible state until it is certain that the instruction is on the correct path. Once the wrong-path instructions are invalidated, the CPU fetches instructions from the correct target of the branch and begins refilling the pipeline.
  - The cost of a misprediction is the time wasted in fetching, decoding, and speculatively executing the wrong instructions, plus the time needed to refill the pipeline with correct ones. On modern deep pipelines, this penalty can be on the order of 10 to 20 cycles, and in very aggressive superscalar or out-of-order processors, it can exceed 30 cycles. This penalty is why accurate branch prediction is crucial for high performance: a high misprediction rate can stall the processor frequently and dramatically reduce instruction throughput.
  - In summary, when a branch is mispredicted, the CPU must flush the wrong instructions, restore a clean architectural state, and restart execution from the correct program counter, incurring a significant performance penalty proportional to pipeline depth and complexity.

- *What is the difference between an interrupt and a trap?*
  - In operating systems, both interrupts and traps are ways to transfer control to privileged kernel code, but they come from different sources and have different timing guarantees.
  - An **interrupt** is an asynchronous, externally generated event—typically from hardware like a NIC, disk, or the APIC timer—that arrives "between" instructions. Because it is

external to the executing instruction stream, the CPU can treat it as happening at clean instruction boundaries: the current instruction completes, a small, well-defined register frame is pushed, the processor vectors through an interrupt table to the appropriate Interrupt Service Routine, and—unless it's a non-maskable interrupt—the OS may defer or prioritize it by masking or by interrupt priority. The hallmark of interrupts is that they are not causally tied to the instruction you just executed; they're about the outside world demanding attention, and they introduce scheduling and jitter concerns because they can preempt anything at almost any time.

- A **trap**, by contrast, is synchronous and internally generated by the CPU as a direct result of executing the current instruction. Classic examples are intentional traps used for system calls and breakpoints (x86 `int 0x80` / `syscall`, ARM `SVC`, a debugger's `int3`), and architectural exceptions raised by the instruction itself (e.g., divide-by-zero, invalid opcode). The key property is causality and timing: the event is detected while executing a specific instruction, so the architecture defines a precise restart point. Traditional terminology further refines synchronous exceptions into faults, traps, and aborts: a *fault* (like a page fault) is reported before the instruction completes and, once the kernel fixes the condition, control returns to re-execute the faulting instruction; a *trap* is reported after the instruction completes and returns to the next instruction (as with breakpoints and many syscalls); an *abort* signals an unrecoverable condition with no defined restart point. Different ISAs use these words with minor variations, but the synchronous (instruction-caused) vs asynchronous (device-caused) split is universal.
- Practically, this means a timer firing or a NIC signaling new packets is an interrupt: it can be masked, prioritized, and handled between instructions with no need to restart anything. A system call or a debugger breakpoint is a trap: the kernel entered because *your* instruction asked it to, and the architecture guarantees a precise place to resume (next instruction for a trap, same instruction for a resolvable fault). In low-latency systems this difference drives design choices: avoiding syscalls reduces trap frequency on the hot path, while techniques like busy-polling or kernel bypass reduce reliance on interrupts and the jitter they introduce.
- In short, interrupts are asynchronous, external, and occur between instructions; traps are synchronous, instruction-caused, and occur at a precise point in the instruction's execution (returning to the next instruction for traps, or re-trying the instruction for faults).
- See this thread for more info.

- *What does the OS when an interrupt occurs?*

- When an **interrupt** occurs, the operating system (OS) temporarily suspends the execution of the currently running process to address the event that triggered the interrupt. This mechanism allows the CPU to respond quickly to urgent signals, such as hardware requests or software exceptions, ensuring efficient multitasking and system responsiveness.
- The OS first saves the current state of the running process, including the program counter and CPU registers, so that execution can later resume exactly where it left off. It then determines the source and type of the interrupt—whether it came from a hardware device like a keyboard, disk, or network card, or from a software event such as a system call or timer. The appropriate Interrupt Service Routine (ISR), also known as an interrupt handler, is then executed to handle the specific event.
- Once the interrupt has been serviced, the OS decides whether to resume the interrupted process or to perform a context switch and schedule another process if the current one is no longer ready to run (for example, if it has been blocked while waiting for I/O).
- In summary, when an interrupt occurs, the OS manages it by saving the current process state, executing the relevant interrupt handler, and then resuming or rescheduling processes as appropriate, thereby maintaining system stability and responsiveness.

- *What is the program counter?*
  - The **program counter** (PC), also known as the **instruction pointer** on some architectures, is a special CPU register that holds the memory address of the next instruction to be executed. During program execution, the CPU repeatedly fetches the instruction stored at the address in the program counter, decodes it, executes it, and then updates the program counter to point to the following instruction. This is what allows a processor to execute instructions sequentially.
  - For most instructions, the PC simply moves to the next address in memory, but for control flow instructions like jumps, branches, function calls, and returns, the program counter is explicitly modified to a new address, allowing the program to loop, branch, or call subroutines. When interrupts or exceptions occur, the CPU saves the current PC so that it can resume execution later. In essence, the program counter keeps track of “where the CPU is” in a program, and by changing it, the CPU changes the flow of execution.

- *What happens with the program counter with an interrupt?*

- When an **interrupt** occurs, the CPU temporarily stops what it's doing, saves the **current program counter (PC)**, and then jumps to a special piece of code called the **interrupt handler** or **interrupt service routine (ISR)**. This allows the processor to handle external events (like I/O, timers, or exceptions) without losing track of where it was in the original program.
- Here's what happens step by step:
  1. **Interrupt signal**: An external device or internal event triggers an interrupt.
  2. **Save PC (and state)**: The CPU automatically saves the current program counter — usually the address of the **next instruction** that was about to be executed — along with other essential processor state (like flags) onto the stack or into special registers.
  3. **Change PC**: The CPU loads a new value into the program counter, corresponding to the entry point of the interrupt handler. This is defined by the system's **interrupt vector table**, which maps interrupt types to handler addresses.
  4. **Run handler**: The CPU executes the interrupt service routine, handling whatever caused the interrupt (e.g., reading from a device).
  5. **Restore PC**: When the handler finishes, it executes a special **return-from-interrupt** instruction (such as `iret` on x86), which restores the saved PC and processor state. Execution then continues from exactly where it left off.
- In short, an interrupt causes the CPU to **save the current PC**, **jump to a handler**, and then **restore the PC** afterward, so that normal program execution resumes seamlessly once the interrupt has been serviced.
- See this video for more info.

- *What are green threads?*
  - **Green threads** are user-level threads managed by a runtime or library instead of the operating system. The runtime implements its own scheduler, stacks, and context switches in user space, so creating, switching, and parking a green thread is much cheaper than doing the same with OS threads. Because the OS doesn't know about them, a single OS thread can host many green threads; the runtime multiplexes them (often called M:N scheduling when many green threads are mapped over a pool of N OS threads).
  - The big wins are low overhead and scalability: you can spawn hundreds of thousands or even millions of green threads, with microsecond-scale context switches and tiny stacks. The main caveat is blocking: if a green thread makes a blocking syscall on an OS thread, it can stall every green thread riding on that OS thread. Runtimes avoid this by using non-blocking I/O, epoll/kqueue/IOCP, async syscalls, or by offloading blocking

work to separate OS threads. Preemption is also runtime-defined: some systems are cooperative (threads yield at await points), others add timer-based preemption.
- Historically, early Java on Solaris used green threads; today, examples include Go's goroutines (green threads scheduled over a worker pool), Erlang/Elixir processes, many coroutine/fiber libraries (Boost.Fiber, Ruby fibers), and async runtimes that schedule "tasks" in user space (Kotlin coroutines, Rust async executors, Python gevent/greenlet). Compared with pure OS threads, green threads offer far higher concurrency at lower cost, but require careful integration with I/O and native code to avoid accidental blocking and to achieve true parallelism across cores.

- *Are green threads used in HFT?*
  - **Rarely in the hot path.** HFT engines care about ultra-low and *predictable* latency. The common design is **one core ↔ one pinned OS thread ↔ one busy-poll loop**, with **no scheduler in the way**, no blocking syscalls, and lock-free SPSC queues between stages. Green threads (goroutines/fibers/coroutines scheduled in user space) add a runtime scheduler and potential pause points; even tiny, they introduce **jitter** and complicate "nothing ever blocks" guarantees. That's why the tick-to-trade path is typically C/C++ (or FPGA), not Go/managed runtimes with GCs.
  - Where you might see them:
    - **Non-critical paths**: monitoring, risk UI, ETL, backtesting, research infra—places where a few µs of jitter doesn't matter. Go/Erlang/etc. can shine here.
    - **Cooperative coroutines/fibers in C++**: sometimes used to structure code, but only if every syscall is non-blocking and the event loop is still a **single, pinned OS thread**. Think "syntax sugar over an event loop," not massive M:N scheduling.
  - Rule of thumb: for the trading hot loop, use **pinned OS threads + busy-wait**; for everything else, pick whatever boosts developer speed—with green threads fine outside the latency-critical core.

- *Can you explain what if hyperthreading and how it differs from multithreading?*
  - Hyper-Threading (HT) is Intel's brand name for simultaneous multithreading (SMT), a hardware feature where a single physical core exposes two (or more) **logical** processors to the OS. The core duplicates only its architectural state (registers, program counters, etc.) but **shares** the heavy execution resources (front end, schedulers, ALUs, caches, load/store units). With HT/SMT, the core can issue instructions from two threads in the **same cycle**, filling pipeline bubbles when one

thread stalls on a cache miss or branch mispredict. It doesn't double throughput because the threads contend for the same execution units and caches; real-world gains are workload-dependent (often 10–30%) and can be negative for resource-hungry code or when cache contention dominates.

- Software **multithreading**, by contrast, is an OS/runtime concept: a process has multiple threads of execution that the scheduler maps onto whatever hardware is available—physical cores, SMT siblings, or time slices on a single core. Multithreading gives you concurrency; actual parallelism depends on hardware. A 4-core CPU with HT shows 8 logical CPUs: the OS can run up to 8 threads "at once," but only 4 can execute on distinct physical cores; the other 4 share resources with a sibling on the same core. You can multithread even without HT: the OS time-slices threads on each core, interleaving them when there aren't enough hardware contexts.
- In short, Hyper-Threading/SMT is **hardware** support that lets one core run multiple threads simultaneously by sharing execution resources; multithreading is a **software** model for structuring programs into threads, which the OS schedules onto cores (with or without SMT). Practical implications: HT can improve utilization and throughput for latency-bound or mixed workloads, but it may hurt for memory-bandwidth-bound or cache-sensitive code; thread pinning and benchmarking both HT on/off are common to find the best setting for a given workload.

- *Why is hyperthreading generally not recommended in HFT?*
  - In HFT you usually care about *worst-case latency and jitter*, not total throughput. Hyper-Threading (SMT) is great for throughput, but it makes latency and determinism worse. That's the core reason it's often disabled.
  - In more detail, the main issues are:
    1. **Resource contention on the core**
       With Hyperthreading, two logical threads share one physical core's execution resources:
       registers, ALUs, caches, load/store units, branch predictors, etc. If your critical HFT path is pinned to one logical core and *anything at all* is scheduled on the sibling logical core, they compete for those resources. That competition adds:
       − extra cycles to retire instructions
       − more cache pollution and misses
       − more variability from run to run
       Even if the *average* latency impact is small, the *tail* (p99, p999) gets fatter, which is exactly what HFT wants to avoid.
    2. **Increased latency *jitter***
       HFT systems are tuned so that the path from "packet hits the NIC" to "order on the

wire" is as consistent as possible. SMT introduces another source of non-determinism: sometimes the sibling thread is busy, sometimes it's not; sometimes it evicts your cache line, sometimes it doesn't. So you get micro-bursts of extra latency that are hard to predict and hard to fully eliminate. Tail latency is more important than mean latency; hyperthreading tends to hurt tails.

3. **Cache and TLB pollution**
   The sibling logical core shares:
   − L1 I/D caches
   − L2 cache
   − TLBs
   If you've carefully warmed your instruction and data caches for a tight critical loop, another thread running on the same core can evict those lines or TLB entries. When that happens, your "hot" path suddenly takes a cache miss or TLB miss and spikes in latency. In HFT, avoiding those rare spikes is worth giving up some theoretical throughput.
4. **Less predictable performance tuning**
   HFT shops do extreme low-level tuning: CPU pinning, cache layout, prefetching, NUMA placement, busy-waiting loops, custom allocators, etc. All those tricks assume a relatively stable relationship between your code and the hardware resources. Hyperthreading complicates that mental model: you now need to consider what's running on the sibling, how the OS scheduler might move background tasks, and how that interacts with your caches and pipelines. It's usually simpler and safer to:
   − disable HT/SMT in BIOS, and
   − dedicate whole physical cores to specific tasks (e.g., one for network RX, one for matching, one for TX).

- *What is a Virtual Memory Area (VMA)?*
  - A **Virtual Memory Area (VMA)** is a contiguous range of virtual addresses within a process that shares the same attributes and backing store. Each VMA describes "what this region is and how it should behave":
    - its start/end addresses, permissions (read/write/execute),
    - whether it's private copy-on-write or shared,
    - and what backs it (anonymous memory like heap/stack or a file mapping via `mmap`).
  - Operating systems—Linux in particular—store a process's address space as a set of VMAs (e.g., code, data, heap, each mapped shared library, stack). VMAs are metadata; they don't hold page contents or page tables themselves. On a page fault,

the kernel finds the VMA covering the faulting address to decide if access is allowed, whether to allocate a zeroed page, read from a file, or trigger copy-on-write.
    - VMAs can be split/merged when mappings change, and are typically organized per process in balanced trees/lists for fast lookup. In short, a VMA is the OS's high-level descriptor for a uniform region of a process's virtual address space, guiding protection, paging, and mapping behavior.
    - See this for more info.

- *What is demand paging?*
    - **Demand paging** is a virtual-memory strategy where the OS loads a page from backing store (disk/SSD) **only when a program actually touches it**—not at process start. Each PTE starts **invalid** (or marked "not present"). When the CPU tries to read/write an unmapped virtual address, it triggers a **page fault** trap; the kernel looks up what should live there (executable file, mapped data, or a zero page), picks or frees a **physical frame** (evicting another page if needed), reads the 4 KB (or huge-page) block from storage into RAM, fixes the PTE (valid, with access bits), and resumes the faulting instruction. Variants include **demand-zero** pages (filled with zeros on first touch), **copy-on-write** (forked pages shared until a write), and memory-mapped files (page-in file blocks on access).
    - Why use it: **fast startups** (don't load everything up front), **lower memory footprint** (only hot pages occupy RAM), and **better sharing** (code/data loaded once, shared across processes). Costs: the first access to a cold page pays a **huge latency** (microseconds–milliseconds) for I/O and kernel work; heavy faulting can cause **thrashing** as pages are constantly evicted and reloaded. Systems mitigate with **prefetching/pre-paging**, **working-set** tuning, and using **huge pages** for large, linear scans.
    - See this video for more info.

- *How does the kernel allocates memory to user processes?*
    - At a high level, Linux gives a process **virtual addresses**, not raw RAM. When your code asks for memory (via `malloc/new`), the C library requests address space from the kernel using **`mmap`** (and sometimes the old `brk` / `sbrk` for small heaps). The kernel records this as **VMAs** (virtual memory areas) in the process but usually **doesn't back it with physical pages yet**—it's *lazy*.
    - When the process **touches** an address for the first time, the CPU raises a **page fault** because the PTE is "not present." The kernel's fault handler looks up the VMA,

decides what should live there (anonymous zero-filled memory, a page from a mapped file, or a copy-on-write page from `fork`), grabs a **physical frame** from its page allocator (the buddy allocator; or reuses one by evicting another page if needed), **zeroes** it for safety (unless it's file-backed), installs a valid **page-table entry** mapping that frame to the virtual page with appropriate permissions, and resumes the instruction. This "demand paging" makes allocation fast up front and only pays for the pages you actually use.

- Large or file-backed regions work the same way: `mmap` of a file creates a VMA whose pages are **paged in on demand**; writes dirty the page and the kernel later **flushes** it back. After `fork`, parent and child share pages as **copy-on-write** until one writes, whereupon the kernel allocates a new frame and updates that process's PTE. For performance, Linux can use **huge pages** (THP 2 MB, or explicit hugetlb 2 MB/1 GB) to reduce TLB pressure; **NUMA** policies can steer page placement to a socket-local node.
- Accounting and limits are separate: the kernel tracks **virtual size** (address space) vs **RSS** (resident physical pages), applies **overcommit** policy to decide whether to allow large reservations, enforces **ulimits**/cgroup memory limits, and reclaims memory under pressure (LRU eviction, swapping, file-cache writeback). Security and robustness come from per-VMA protections (r/w/x), **ASLR**, and **guard pages** around stacks.
- So: user code asks for address space; the kernel maps it virtually; **physical RAM is provided lazily on page faults**, with copy-on-write, file-backed paging, huge pages, NUMA placement, and reclamation machinery handling the details behind the scenes.
- See this video and this video for more info.

- *So if I use malloc in my program and request let's say 32KB of memory, is the kernel directly going to map 8 pages (assuming 4kb pages) to 8 physical frame?*
  - Short answer: **no**. A call to `malloc(32*1024)` does **not** immediately make the kernel map 8×4 KB physical frames for you. `malloc` is a user-space allocator; it usually satisfies your request from its own arenas (**private pools of memory** that it previously obtained from the kernel (via `mmap` or `brk`)). Only when it needs more address space does it ask the kernel—either by extending the heap (`brk/sbrk`) or by `mmap`-ing a region (large/huge allocations). Even then, Linux is **lazy**: it creates *virtual* mappings but typically **doesn't back them with physical RAM yet**. Pages become backed **on first touch** via a **page fault** (demand paging): the kernel allocates a frame, zeroes it (for safety), fills the page table entry, and resumes your code. So for a 32 KB block you might get a pointer into memory the allocator already had (no syscalls at all), you'll fault in **only the pages you actually access**, and those pages need not be physically

contiguous. The only time you'd see immediate page backing is with special flags (e.g., `MAP_POPULATE`, locked memory with `mlock`, or explicit huge/hugetlb pages); otherwise, mapping is virtual-first, physical-on-demand.

- *Explain the difference between write-back and write-through caches? What about write-allocate and no-write allocate?*
  - **Write-through vs. write-back (what happens on a *hit*)**
    - **Write-through:** every store updates the cache *and* immediately writes the new value to the next level (or memory). The cache line stays **clean** (no "dirty" bit).
      Pros: simplest correctness and coherence, easy to snoop, no surprises on eviction.
      Cons: lots of downstream traffic; needs a **write buffer** to avoid stalling the CPU on each store.
    - **Write-back:** a store updates only the cache line and marks it **dirty**; the lower level is updated **later**, when the line is evicted (or explicitly flushed).
      Pros: far fewer writes to lower levels → better bandwidth/energy, allows multiple stores to collapse into one writeback.
      Cons: more complex (dirty tracking, eviction writebacks, coherence), and data isn't in memory until eviction (matters for DMA or power loss unless protected).
  - **Write-allocate vs. no-write-allocate (what happens on a *miss*)**
    - **Write-allocate (a.k.a. fetch-on-write):** on a store miss, bring the target line into the cache, then perform the write (often read-modify-write if partial).
      Why: future accesses are likely; pairs well with **write-back** so subsequent stores hit in cache.
    - **No-write-allocate (write-around):** on a store miss, **do not** fill the cache; send the write directly to the next level/memory.
      Why: avoid polluting the cache for data you won't reuse; often paired with **write-through**.
    - In details, with **write-allocate** (common with write-back caches) the line is fetched on the miss (usually via RFO), updated in cache, and marked dirty. With **no-write-allocate** (typical with write-through or MMIO regions) the store is **written around** the cache directly to the next level; the line is not filled, so later accesses will miss again.
  - **Common combinations & practice**
    - CPU data caches are typically **write-back + write-allocate** (best locality and bandwidth).
    - **Write-through + no-write-allocate** is used for special regions (MMIO, DMA-shared buffers) to keep memory immediately up to date and to avoid cache

pollution.
    - Regardless of policy, modern cores hide latency with **store buffers** and may merge adjacent writes (write combining).
    - Reads on a miss are almost always **allocate** (you fetch the line you're about to read).
- In one line: *write-through vs write-back* decides **when** lower levels see your stores on hits; *write-allocate vs no-write-allocate* decides **whether** a store miss pulls the line into the cache in the first place.

Example Assume a fully associative write-back cache with many cache entries that starts empty. Following is a sequence of five memory operations (the address is in square brackets):

```
Write Mem[100];
Write Mem[100];
Read  Mem[200];
Write Mem[200];
Write Mem[100].
```

What are the number of hits and misses when using no-write allocate versus write allocate?

Answer For no-write allocate, the address 100 is not in the cache, and there is no allocation on write, so the first two writes will result in misses. Address 200 is also not in the cache, so the read is also a miss. The subsequent write to address 200 is a hit. The last write to 100 is still a miss. The result for no-write allocate is four misses and one hit.

For write allocate, the first accesses to 100 and 200 are misses, and the rest are hits because 100 and 200 are both found in the cache. Thus, the result for write allocate is two misses and three hits.

- *Can you explain what is Direct Memory Access (DMA)?*
    - Direct Memory Access (DMA) is a way for hardware devices to read/write main memory **without the CPU copying every byte itself**. Instead of the CPU doing `load/store` loops to move data between a device and RAM, the CPU just *sets up* a DMA transfer, and the device's DMA engine moves the data directly over the memory/bus fabric.
    - The usual flow looks like this: the CPU programs a DMA-capable device (via registers / descriptors) with "source address in memory, destination address (or device FIFO), length, and options". Then it tells the device "go". The DMA engine then transfers the data between device and RAM on its own, possibly as a bus master. While that's happening, the CPU can go do other work. When the transfer finishes (or an error happens), the device typically raises an interrupt or updates some status that the CPU can check.

- The main benefits are:
  1. Much higher throughput than CPU-driven copying, especially for large blocks.
  2. Lower CPU usage: the CPU is not stuck in a memcpy loop.
  3. Better overlap of computation and I/O, since transfers can run in parallel with normal code.
- For a NIC (Network Interface Card), DMA is absolutely central. When a packet comes in from the wire, the NIC **does not** trap into the CPU and ask, “Can you please write these bytes into memory?” That would be far too slow. Instead, the OS and NIC cooperate like this (simplified):
  1. The OS allocates one or more **receive buffers** in RAM and ensures it knows their physical (or DMA-mapped) addresses.
  2. It builds a **ring (queue) of descriptors** in memory. Each descriptor says “here is a buffer at physical address X with size N where you may store one incoming packet”.
  3. The OS programs the NIC with the location of this descriptor ring (again using DMA-capable addresses) and tells the NIC “RX ring is here, you may use these buffers”.
  4. When a packet arrives, the NIC’s DMA engine writes the packet payload **directly into those RAM buffers** using DMA. It also updates the descriptor to say “this buffer is now filled with a packet of length L”.
  5. The NIC then signals the CPU (interrupt, or the CPU polls the ring) that some descriptors are now “complete”.
  6. The kernel networking stack reads metadata from the descriptor, processes the packet, and eventually recycles or frees the buffer.
- So the packet path looks like: **wire → NIC → DMA into RAM**. The CPU does not copy the packet byte-by-byte into memory; it only does control operations: set up rings, inspect headers, update state. This is how a modern NIC can handle line rates of many Gbit/s without pegging a CPU core just doing copies. Similarly, on transmit:
  1. The OS puts outgoing packet data into buffers in RAM.
  2. It fills TX descriptors pointing at those buffers (with addresses and lengths).
  3. It notifies the NIC: “here are descriptors N..M ready to send”.
  4. The NIC DMA-reads the packet data from RAM into its internal transmit queues and puts it on the wire.
  5. When it’s done, it marks those descriptors as free, so the OS can reuse the buffers.
- NICs often support **scatter-gather DMA** (lists of fragments) so you can build a packet from multiple non-contiguous memory chunks without first concatenating them into one buffer, reducing extra copies.

- In modern systems there's also usually an **IOMMU** in the path, which lets the OS control which physical pages a device's DMA engine can access, so a buggy or malicious device cannot DMA into arbitrary kernel or user memory.
- See this video for more info.

Linux-Related OS/CA Questions

- *Can you explain the booting process of Linux, a bare-metal OS, and an RTOS?*
  - **Linux Boot Process**
    • First, the system firmware (BIOS or UEFI) performs hardware initialization and then looks for a bootloader (typically GRUB on PCs).
    • The bootloader loads the Linux kernel image—often accompanied by an initial RAM disk (initramfs) containing drivers and early userspace tools.
    • The kernel decompresses itself, sets up essential subsystems (memory management, device drivers, interrupt handling, etc.), and then mounts the root filesystem.
    • Finally, it transfers control to the first userspace process (traditionally init, or nowadays systemd), which in turn starts all other services and user applications.
    This process is quite elaborate because Linux must handle a wide range of hardware and support a rich set of services and drivers.
  - **Bare-Metal OS Boot Process**
    • A bare-metal OS typically runs with minimal runtime support directly on the hardware.
    • The firmware (or a very simple bootloader) initializes the hardware and then loads the OS image from nonvolatile storage into RAM.
    • The OS kernel immediately takes over—performing only the essential initialization like setting the CPU mode, configuring basic memory management, and initializing minimal drivers needed for its purpose.
    • Once initialized, the OS enters its main loop or scheduler.
    Because the OS is "bare-metal," it usually avoids the layers of abstraction (and complexity) found in a full-featured OS, resulting in a leaner and faster startup tailored for specific hardware or embedded applications.
  - **RTOS (Real-Time Operating System) Boot Process**
    • RTOS booting is designed to be extremely deterministic and fast. The firmware or a lightweight bootloader loads the RTOS image, often from flash memory or ROM, with little overhead.
    • The RTOS kernel initializes by setting up critical hardware resources: it configures clocks, hardware timers (for scheduling), and the interrupt vector table, ensuring that all time-critical services are in place.

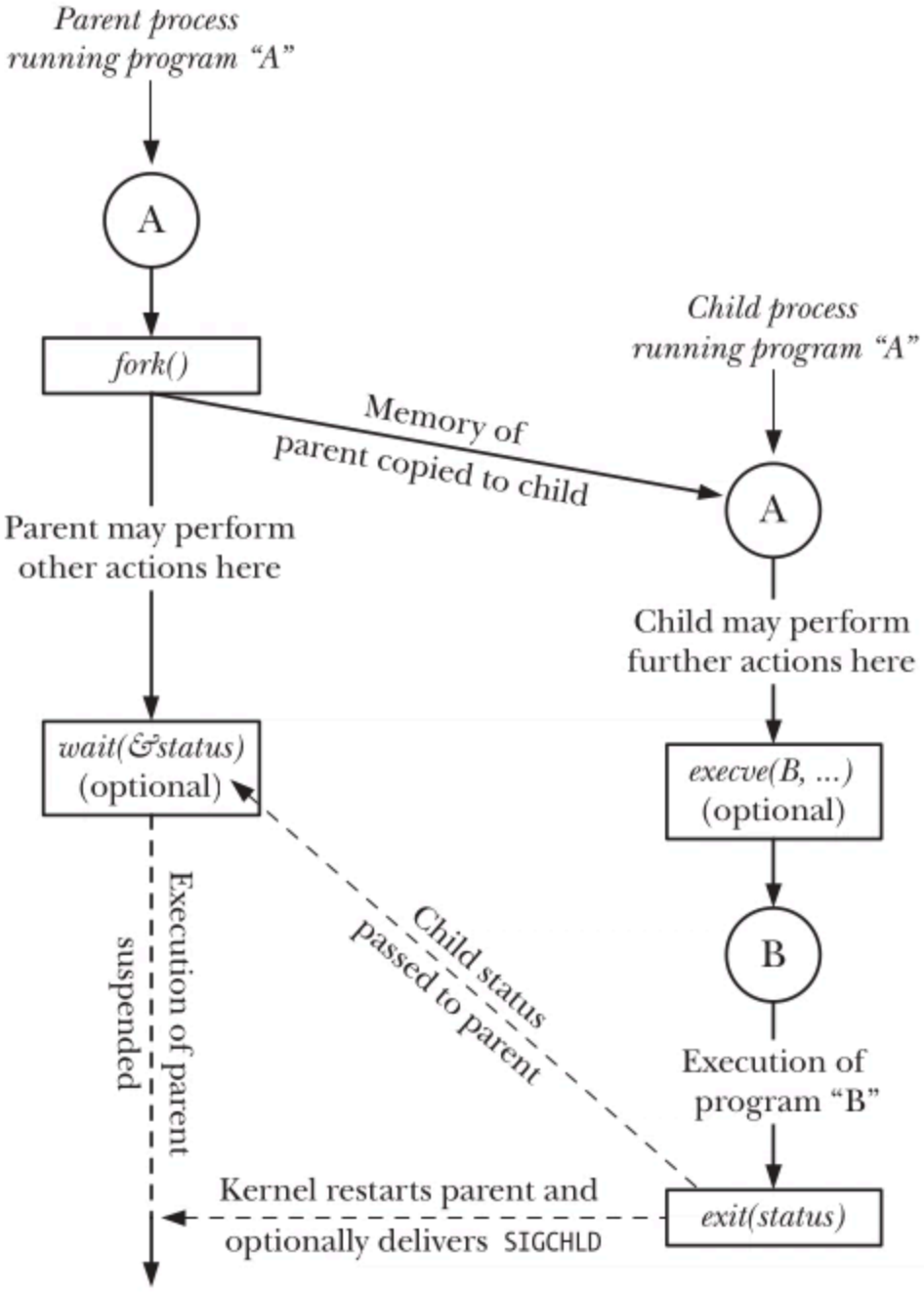
• Unlike Linux, which dynamically loads many services, an RTOS typically uses statically allocated resources and a minimal set of drivers to ensure predictable timing.
• The kernel then initializes its task control structures and starts the real-time scheduler, which immediately begins executing tasks according to strict priority rules. This streamlined process is crucial for applications (such as in industrial or automotive systems) where predictability and low latency are paramount.

- *Can you explain what happens during a context switch (Linux)?*
  - During a **context switch** in Linux, the operating system saves the state of the currently running process or thread and restores the state of another, allowing the CPU to switch from one task to another. This process begins when the scheduler decides that the current task should stop running, either because its time slice has expired, it voluntarily yields, or a higher-priority task becomes runnable.
  - The kernel first **saves the CPU state** of the current task, which includes the contents of general-purpose registers, the **program counter** (so the process can later resume exactly where it left off), the stack pointer, and other processor flags. This state is stored in the task's kernel data structure, typically a `task_struct` and associated kernel stack.
  - Next, the kernel **loads the saved state** of the task being switched to. This involves restoring its registers, stack pointer, and program counter, effectively rewinding the CPU to look exactly as it did when that task was last running.
  - The **memory context** (such as the page tables) is also switched if the new task belongs to a different process, ensuring that the virtual address space corresponds to the correct program.
  - Finally, the kernel updates internal scheduling structures and jumps to the new task's program counter, resuming its execution as if it had never been interrupted. This entire process happens in kernel mode and is designed to be as fast as possible, since frequent context switches are common in multitasking systems.

- *Can you give some commonly use Linux system call and each time give an explanation of what it does?*
  - **`open(pathname, flags, mode)`**
    Opens (or creates) a file specified by `pathname`, returning a file descriptor that refers to it. The `flags` control access mode (read, write, append) and behaviors (e.g., `O_CREAT` to create, `O_EXCL` to error if it exists), while `mode` sets permission bits when a new file is created.

- **`read(fd, buf, count)`**
  Attempts to read up to `count` bytes from the file descriptor `fd` into the buffer `buf`. Returns the number of bytes actually read (which may be less at end-of-file) or 0 on EOF. On error, returns −1 and sets `errno`.
- **`write(fd, buf, count)`**
  Writes up to `count` bytes from buffer `buf` to the file descriptor `fd`. Returns the number of bytes written—useful for handling short writes on non-blocking or interrupted I/O—or −1 on failure.
- **`close(fd)`**
  Closes the file descriptor `fd`, releasing its reference to the underlying file (or socket, pipe, etc.). After this call, `fd` can be reused by future `open`, `socket`, or other descriptor-allocating syscalls.
- **`fork()`**
  Creates a new process by duplicating the calling process. The child inherits copies of the parent's memory, file descriptors, and execution context, but receives a return value of 0 (while the parent gets the child's PID). Used to spawn concurrent execution paths.
- **`execve(path, argv, envp)`**
  Replaces the current process image with a new program loaded from the executable at `path`. Arguments (`argv`) and environment (`envp`) are passed to the new program. On success, this call does not return; on failure, it returns −1.
- **`waitpid(pid, &status, options)`**
  Suspends the calling process until the child specified by `pid` terminates (or changes state). The child's exit status (or signal info) is stored in `status`. Options like `WNOHANG` allow non-blocking checks for child status.

- **`exit(status)`**
  Terminates the calling process immediately, returning `status` to the parent (via `waitpid`) and freeing all of the process's resources (memory, open files, etc.).



- **`mmap(addr, length, prot, flags, fd, offset)`**
  Creates (or maps) a memory region of `length` bytes, optionally backed by a file (`fd`) at byte offset `offset`, or as anonymous zero-filled pages if `fd` is −1. The `prot` bits control read/write/execute permissions, and `flags` such as `MAP_PRIVATE` or `MAP_SHARED` determine visibility of changes. See this video for more info.
- **`munmap(addr, length)`**
  Unmaps the memory region starting at `addr` for `length` bytes, returning the pages to the kernel. Any subsequent access to that region causes a segmentation fault.

- **`brk(addr)` / `sbrk(increment)`**
  Adjusts the end of the process's data segment (the "heap"). `brk()` sets the break to a specific address, while `sbrk()` moves it by `increment` bytes. Typically used by user-level allocators (e.g., `malloc`) to grow or shrink the heap.
- **`socket(domain, type, protocol)`**
  Creates an endpoint for network communication. `domain` selects the address family (e.g., `AF_INET` for IPv4), `type` picks the semantics (`SOCK_STREAM` for TCP, `SOCK_DGRAM` for UDP), and `protocol` is usually zero to select the default. Returns a socket descriptor for use with `bind`, `connect`, `send`, and `recv`.
- **`bind(fd, sockaddr *addr, addrlen)`**
  Associates a local address (IP and port) with a socket descriptor `fd`, preparing it to accept incoming connections or datagrams on that address.
- **`listen(fd, backlog)`**
  Marks a bound socket descriptor `fd` as passive, indicating the kernel should queue up to `backlog` incoming connection requests until they're accepted.
- **`accept(fd, sockaddr *addr, addrlen *addrlenptr)`**
  Extracts the next pending connection from the listen queue of socket `fd`, returning a new descriptor for the established connection and filling in the client's address.
- **`connect(fd, sockaddr *addr, addrlen)`**
  Initiates a connection on a socket descriptor `fd` to a remote address specified by `addr`. For stream sockets, this performs the TCP handshake; for datagram sockets, it sets a default peer.
- **`select(nfds, readfds, writefds, exceptfds, timeout)`**
  Monitors multiple file descriptors to see if they're ready for reading, writing, or have exceptional conditions. Blocks until one or more descriptors become ready or the optional timeout expires.
- **`poll(fds, nfds, timeout)`**
  A more scalable alternative to `select()`, taking an array of file descriptors and event masks, and waiting up to `timeout` milliseconds for events.
- **`pipe(pipefd)`**
  Creates a unidirectional data channel: `pipefd[0]` is the read end, `pipefd[1]` the write end. Data written on one end can be read from the other, useful for parent-child communication.
- **`dup(oldfd)` / `dup2(oldfd, newfd)`**
  Duplicates a file descriptor. `dup()` returns the lowest-numbered unused descriptor referring to the same open file description as `oldfd`; `dup2()` makes `newfd` refer to it (closing `newfd` first if necessary).

- **`kill(pid, sig)`**
  Sends the signal `sig` to the process (or process group) identified by `pid`. Commonly used to request termination (`SIGTERM`) or to send user-defined signals (`SIGUSR1`, etc.).
- See this for more info.

- *Can you explain how a system call works/is handled under Linux?*
  - When a program invokes a system call (for example by calling the libc wrapper for `open()`), the C library places the system-call number in a designated register (on x86-64, `%rax`) and arranges the arguments in the other ABI-specified registers (`%rdi`, `%rsi`, `%rdx`, etc.). It then executes the special instruction (`syscall` on modern x86-64), which causes a **trap** into the kernel: the CPU switches from user mode to privileged mode, saves the user's register state into a trap frame on the kernel stack, and jumps to a fixed entry point in the kernel's trap-handling code.
  - At the top of the kernel's entry path, the trap handler inspects the saved state to extract the system-call number and its arguments. It validates any pointers (ensuring they point to legal user pages), checks permissions, and then dispatches to the appropriate internal routine (for example, the VFS layer for file operations or the memory manager for `mmap()`). If the call must block—say, on disk I/O or waiting for a lock—the kernel may switch out the calling thread and schedule another, but the original context remains parked in the scheduler until it can complete.
  - When the kernel routine finishes, it writes the return value (or a negated error code) back into the trap frame's return register, then executes the **return-from-trap** instruction (`sysret` or `iret` on x86). That instruction restores the saved registers and privilege level from the trap frame, returning control to user mode at the instruction immediately following the original `syscall`. From the program's perspective, the call simply returns like any other function, but behind the scenes it was boxed by a trap into the kernel and a return-from-trap back to user space.
  - See this video for more info.

- *Can you explain why would you want to use huge pages? How do you change the size of your pages on Linux?*
  - You use huge pages (e.g., 2 MB or 1 GB instead of 4 KB) to **cut translation overhead** and make memory access more predictable. Fewer, bigger pages mean **far fewer PTEs** and therefore **fewer TLB entries** needed to cover the same working set—this increases the TLB's effective "reach," reducing costly TLB misses. You also shrink the

page-table footprint and the number of **page faults** needed to populate large regions (heaps, code, buffers, mmap'd files), which lowers kernel bookkeeping and interrupts. In latency-sensitive apps (trading, databases, HPC), this translates into **lower tail latency** and **smoother throughput**; in bandwidth-bound code, it can improve sustained GB/s by avoiding TLB thrash. Huge pages also help **NUMA placement** and large DMA mappings by reducing the count of mappings the kernel and IOMMU must manage.

- Trade-offs: bigger pages raise **internal fragmentation** (you might waste RAM inside pages you don't fully use), page faults can be more expensive, and transparent huge pages (THP) heuristics don't always pick the best spots—so high-performance systems often **explicitly** allocate huge pages (e.g., `hugetlbfs`, `madvise(MADV_HUGEPAGE)` / `MADV_NOHUGEPAGE`) for the **hot, contiguous** parts of their memory while leaving the rest on normal pages.
- You can't "change" the base page size of a running Linux system—4 KB (or 16/64 KB on some builds) is set by the kernel/arch at compile time—so the practical way to use larger pages is to allocate **huge pages**.
- Two mechanisms exist: **Transparent Huge Pages (THP)** and the **hugetlb** API. THP (usually 2 MB pages) is on many distros by default; you can check/toggle it with `cat /sys/kernel/mm/transparent_hugepage/enabled` and `echo always|madvise|never > …/enabled`, and you can opt a region in from code with `madvise(ptr,len,MADV_HUGEPAGE)` (or opt out with `MADV_NOHUGEPAGE`). The hugetlb path gives explicit control and quotas: see the huge page size with `grep -i huge /proc/meminfo`, reserve pages with `echo 1024 > /proc/sys/vm/nr_hugepages` (for 2 MB pages), mount a pool via `mount -t hugetlbfs nodev /mnt/huge`, then allocate from user space using `mmap` with `MAP_HUGETLB | MAP_HUGE_2MB` (or `MAP_HUGE_1GB` if supported) or by opening files on `/mnt/huge`. For 1 GB pages on x86, you typically reserve them **at boot**: add kernel params like `default_hugepagesz=1G hugepagesz=1G hugepages=4` (reserves four 1 GB pages), then allocate with hugetlb as above. In short: base page size is fixed; you "change" effective page size per mapping by enabling THP or explicitly reserving/using hugetlb (2 MB/1 GB) where it helps.
- See this for more info.

- *Can you explain what would be the advantages and drawbacks of using huge pages?*
  - Using huge pages (e.g. 2 MB or 1 GB instead of 4 KB) is mostly about trading memory flexibility for better CPU-side efficiency, especially in TLB behavior. The main advantage is that you need far fewer page table entries to cover the same virtual address range. That means fewer TLB entries are consumed, fewer TLB misses, and

fewer expensive page walks. For large, frequently accessed data structures—order books, risk matrices, pricing grids, or big caches—a reduction in TLB pressure can meaningfully drop tail latency and improve throughput. Huge pages can also reduce page-fault frequency after warm-up and slightly lower page-table memory overhead, since the kernel doesn't need to maintain as many levels of page tables for the same amount of RAM.

- However, there are several drawbacks. The big one is internal fragmentation: if you allocate a 2 MB huge page and only use a small portion of it, you still "burn" the full 2 MB of physical memory. That makes huge pages a poor fit for many small, scattered allocations and can lead to memory waste in systems with lots of variable-size objects. Allocation itself can be more fragile and expensive: the OS needs physically contiguous memory to back a huge page, so it may fail or require compaction if the system is fragmented; in practice, you often need to pre-reserve or pre-fault huge pages at startup and run with enough free memory. There's also operational complexity: on Linux, using hugetlbfs often means special mount points, privileges, and manual configuration of pool sizes; even with transparent huge pages, you can get unpredictable behavior (e.g. unexpected promotions or demotions, compaction overhead) that can introduce latency spikes.
- From a performance-tuning point of view, huge pages can actually hurt if your access pattern is highly sparse or if you pin large regions you barely touch. You increase the cost of a single page fault or TLB miss (because the backing structures are larger), and you reduce the granularity with which the OS can manage and migrate memory across NUMA nodes. For very latency-sensitive workloads, the worst case is accidentally mixing huge pages into a memory map without understanding when they are created or reclaimed: you might see occasional stalls due to compaction or page migration that are much more damaging than the TLB savings you were hoping for. So, the short summary is: huge pages are great when you have large, hot, relatively contiguous data that you know you'll touch repeatedly and keep resident; they're risky when your working set is fragmented, dynamic, or when you can't tightly control how and when the OS allocates them.

- *Can you explain what is a trap table, what is its purpose and when it is used?*
  - A **trap table** (also called an **exception/interrupt vector table**) is a small table the CPU consults to decide **which handler to run** when an exceptional event occurs—like a system call, page fault, divide-by-zero, breakpoint, or a hardware interrupt. Each entry (a *vector*) describes a target routine in the kernel and its attributes (privilege level, gate type, etc.). The OS sets this table up early at boot (often per-CPU), points

the CPU at it (e.g., x86 `lidt` to load the **IDT**, ARM `VBAR` to set the vector base), and then the hardware uses it automatically whenever an event happens.

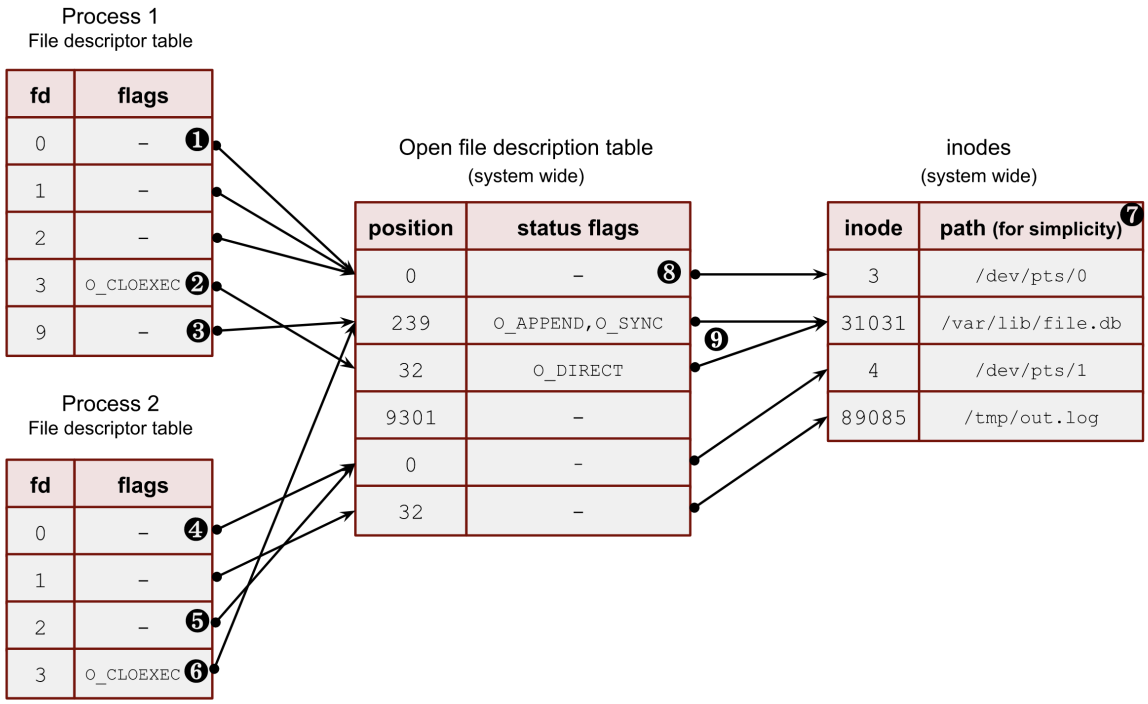
- **Purpose.** It provides fast, safe **dispatch** from arbitrary code to the **correct kernel handler**, while enforcing **privilege transitions** (user → kernel), selecting the right **stack** (e.g., per-CPU kernel stack), and preserving machine state. On entry, the CPU typically saves a **trap frame** (PC/flags/SSP and sometimes an error code), switches to privileged mode, and jumps to the handler address from the table. The handler then services the event (e.g., resolves a page fault, delivers a signal, acknowledges an interrupt, or executes a system call), possibly schedules another thread, and returns with a special instruction (x86 `iretq`, ARM `eret`) that restores the saved state.
- **When it's used.**
  - **Synchronous traps (exceptions):** caused by the current instruction—page fault, divide-by-zero, invalid opcode, breakpoint, system call (`int 0x80` / `syscall` on x86, `svc` on ARM).
  - **Asynchronous interrupts:** from devices or timers—network RX, disk completion, APIC timer tick.
  - **Faults vs traps vs aborts:** a *fault* may be correctable and restarts the faulting instruction (e.g., demand paging); a *trap* reports after the instruction (e.g., debug trap); an *abort* signals severe errors.
- **Concrete mappings:** on **x86**, the trap table is the **IDT** with entries like *interrupt gates* and *trap gates* (differ in how they handle the interrupt-enable flag); on **ARM**, it's the **vector table** at a fixed or configurable base. Regardless of ISA, the idea is the same: the trap table is the CPU's jump directory for exceptional control transfers into the kernel, ensuring the right code runs with the right privilege and saved context.
- See this and this video for more info.

- *What is the difference between static and dynamic libraries in Linux?*
  - **Static libraries** (`.a`) are archives of object files that the **linker copies into your executable at build time**. The resulting ELF is self-contained (no runtime deps on that lib), starts up with no dynamic linker work, but is **larger**, can't share those code pages with other processes, and **won't get security/bug fixes** when the library on the system is updated—you must rebuild. Build/use: `ar rcs libfoo.a *.o`; link with `g++ app.o libfoo.a` or `-lfoo` (and `-static` for a fully static binary).
  - **Dynamic (shared) libraries** (`.so`) are separate files the **dynamic linker loads at runtime** (`ld-linux.so`). Your binary contains references; on start, the loader maps the `.so`s, resolves relocations (PLT/GOT), and many processes **share the same mapped code pages**. Binaries are smaller, you get **updates** by replacing the `.so`, and you can preload/plug-in modules, but you rely on the host's ABI-compatible

libraries and pay a small startup/relocation cost. Build/use: compile with PIC (`-fPIC`), create with `-shared` (`g++ -shared -o libfoo.so *.o`), link with `-lfoo` and control search paths via `rpath` / `LD_LIBRARY_PATH` ; inspect deps with `ldd` .

- Rule of thumb: static for standalone tools/containers/embedded where reproducibility matters; dynamic for normal apps/services where **sharing, updates, and flexibility** matter.

- *What is an Open File Table?*
  - In Unix-like kernels, the **Open File Table** is the kernel's per-open-instance record that sits between a process's **file descriptor** and the filesystem object. Each process has a **descriptor table** (an array mapping `int fd → pointer to open-file entry`). Those pointers reference entries in a **system-wide Open File Table** (often called a *file* or *struct file* in Linux). An open-file entry holds the **current file offset (seek position)**, the **open flags/mode** (read/write/append, O_NONBLOCK, etc.), a **reference count**, and a pointer to the underlying **vnode/inode** (the actual file object) with its metadata and methods. Multiple fds can point to the **same** open-file entry (e.g., after `dup()` or `fork()`), which is why they **share the file offset and status flags**; separate `open()` calls get **distinct** entries and thus independent offsets. When you `read()/write()` the kernel consults the open-file entry (permissions, offset), performs I/O against the vnode (using caches/buffers), updates the offset, and returns. `close()` just **decrements the open-file entry's refcount** and frees it when it hits zero; the inode stays around while other opens exist or the cache retains it. The same machinery is used for regular files, pipes, sockets, devices—only the vnode "ops" differ.



- See this video and this for more info.

- *Can you explain how to handle files in Linux, what system calls should be used, etc.?*
  - **open / openat / openat2**
    You obtain a file descriptor (an integer handle) by opening a path. Use `openat(dirfd, path, flags, mode)` (or legacy `open`) to choose access (`O_RDONLY`, `O_WRONLY`, `O_RDWR`) and behavior (`O_CREAT`, `O_EXCL`, `O_TRUNC`, `O_APPEND`, `O_CLOEXEC`). Prefer the "at" family (`openat`, or `openat2` on newer kernels) so you can open relative to a directory fd for safer, race-free path handling and sandboxing.
  - **close**
    When you're finished with a descriptor, call `close(fd)`. This drops the kernel's reference to the open-file entry; when the last reference goes away, resources are released. Always close fds you own to avoid leaks (and set `O_CLOEXEC` to prevent leaks across `execve`).
  - **read / write**
    `read(fd, buf, n)` and `write(fd, buf, n)` transfer bytes at the current **file offset** and then advance it by the number of bytes processed. They can return short counts—always loop until you've transferred everything or hit EOF/error.
  - **pread / pwrite**
    `pread` / `pwrite` are positional I/O: they operate at a specified offset and **do not change** the file's current offset. Use them for multi-threaded access to the same file without seeking races.
  - **lseek**
    `lseek(fd, offset, whence)` moves the file offset (`SEEK_SET`, `SEEK_CUR`, `SEEK_END`; plus `SEEK_DATA` / `SEEK_HOLE` on filesystems that support sparse files). Pair with `pread/pwrite` when you need precise positioning and offset control.
  - **fsync / fdatasync**
    To make data durable on storage, call `fsync(fd)` (flushes file data **and** metadata) or `fdatasync(fd)` (mostly data). Use this for transactional updates (e.g., write temp file → `fsync` → `rename`), or before reporting success to the outside world.
  - **ftruncate**
    `ftruncate(fd, new_len)` grows or shrinks a file. Growing creates a hole (reads return zeros until written); shrinking discards data beyond the new end.
  - **mmap / msync / munmap**
    `mmap` maps a file into your address space so you can access bytes with loads/stores. It can be private (COW) or shared (writes affect the file). Use `msync` to flush changes when needed and `munmap` to unmap. Great for random access, zero-copy pipelines, and memory-mapped read-only data.

- **stat / fstat / statx**
  These calls return metadata (size, mode, timestamps, inode). Prefer descriptor-based `fstat(fd, &st)` once you already have an fd. `statx` is a richer, modern interface when available. For paths, use `fstatat` / `lstat` as needed (don't-follow-symlink vs follow).
- **Directory listing (opendir / readdir / closedir)**
  Use the POSIX directory stream API to iterate entries. It's simple and portable (backed by `getdents64` under the hood). Each `readdir` returns a name and type; join with `openat` / `fstatat` for robust tree walks.
- **Create / rename / delete**
  Create files atomically with `openat(..., O_CREAT|O_EXCL, mode)`. Replace/update safely via **write to temp + `fsync` + `renameat2`** (or `rename`) so readers never see partial files. Delete with `unlinkat(dirfd, name, 0)` (files) or `unlinkat(..., AT_REMOVEDIR)` (directories).
- **Links and symlinks (linkat / symlinkat / readlinkat)**
  Create hard links with `linkat` (same filesystem) and symbolic links with `symlinkat` (across filesystems). Read symlink targets using `readlinkat`. Use the "at" forms to stay race-safe and relative to a directory fd.
- **Locking (fcntl / flock)**
  Use `fcntl` advisory locks for **byte-range** or record locking (`F_SETLK`, `F_SETLKW`). Use `flock(fd, LOCK_EX|LOCK_SH)` for whole-file, simpler advisory locks. These are cooperative: all participants must follow the convention.
- **dup / dup3**
  Duplicate descriptors with `dup3(old, new, O_CLOEXEC)`. Useful for redirecting stdio or creating independent fd numbers that reference the same open-file entry (thus sharing offset and flags).
- **Temporary files**
  Prefer `openat(dirfd, O_TMPFILE|O_RDWR, 0600)` to create unnamed, already-open files that can later be linked into the filesystem, or `mkstemp()` for a named, unique temp file. Avoid `tmpnam`.
- **Performance knobs**
  `O_DIRECT` can bypass the page cache for large, aligned sequential I/O (requires aligned buffers and sizes). `posix_fadvise(fd, off, len, ADVICE)` gives the kernel hints (`SEQUENTIAL`, `RANDOM`, `WILLNEED`, `DONTNEED`). For `mmap`, use `madvise` similarly. For big sequential writes, **preallocate** with `fallocate` to avoid fragmentation.
- **At-style, race-free paths**
  Wherever you have a path-taking syscall, use the "*at" variant (`openat`, `fstatat`, `linkat`, `renameat2`, `unlinkat`) with a directory fd (or `AT_FDCWD`) to avoid TOCTOU

races and to confine operations to a directory tree—this is the modern, safer style on Linux.
- See this video for more info.

- *The stack is generally 1Mb to 8 Mb. Does that mean that when the program starts, it automatically allocates those Mb? Or does it do it lazily ?*
  - Short answer: **it's lazy.**
  - On Linux, a thread's *stack size* is a **virtual-address reservation**, not pre-allocated RAM. The kernel (for the main thread) or the C library (for `pthread` threads) reserves a contiguous region of virtual memory for the stack (typically **1–8 MiB by default**, platform/distro dependent), plus a **guard page** (`PROT_NONE`) to catch overflows. **Physical pages aren't allocated up front.** As your code pushes frames or uses `alloca` , the stack **grows downward**; each first touch to a new page causes a **page fault**, and the kernel then backs that page with a physical frame. Unused parts of the reserved stack region consume **no RAM**.
  - The maximum size of the main thread's stack comes from the **RLIMIT_STACK** (see `ulimit -s`), and it can auto-expand within that limit. POSIX threads get their stacks from the runtime via `mmap` ; they're also **demand-paged** (physically committed on use), with a default size you can change via `pthread_attr_setstacksize` (and a guard page you can tune with `pthread_attr_setguardsize`). If you exceed the reserved region (past the guard), you'll take a **SIGSEGV**. In short: the "1–8 MiB" figure means "reserved address space," while **RAM is paid as you go** as stack pages are actually touched.
