Operating Systems
Lecture 8: Memory Hierarchy and Caching

Yonghao Lee

May 25, 2025

1 Memory Hierarchy and Caching

In our previous discussions of memory, we treated memory units as simple registers of 32 or 64 bits located within the CPU. These registers are extremely fast, with access times so minimal they appear instantaneous. However, this simplified view doesn't capture the complexity of real memory systems.

Modern computer systems employ a hierarchical memory structure. At the top of this hierarchy are CPU registers, followed by various levels of cache memory, main memory (RAM), and finally storage devices such as hard drives and solid-state drives. Below RAM in the hierarchy, we encounter persistent storage including disks, SSDs, and other storage technologies.

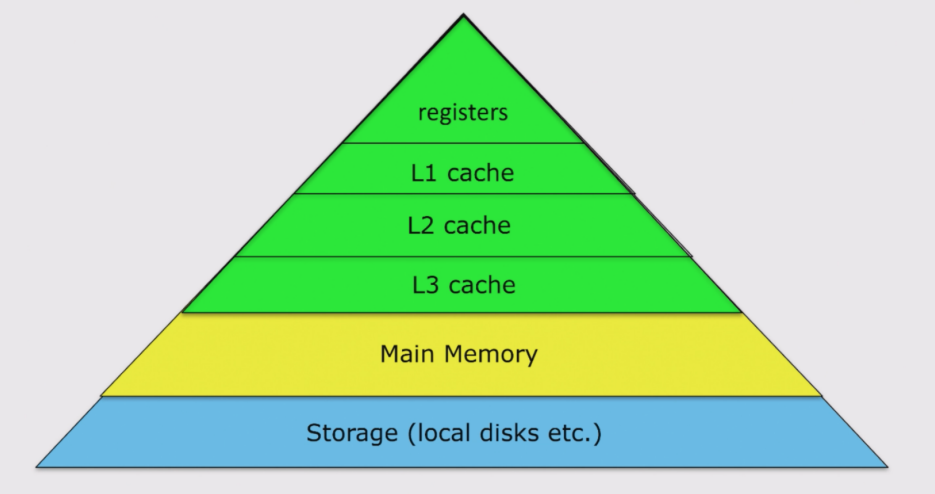

Figure 1: The Memory Hierarchy (representing PDF Figure 1)

In today's computing landscape, cloud storage has introduced an additional layer to this hierarchy. We can now access virtually unlimited storage capacity through cloud services, though this storage is geographically remote. This relationship allows us to conceptualize our local storage as a cache for the larger, cloud-based storage systems.

1.1 Memory Layer Properties

When analyzing each layer of the memory hierarchy, we examine several critical properties:

- **Performance vs. Cost Trade-off:** The relationship between access speed, storage capacity, and economic cost.
- **Caching Relationships:** Which layers serve as caches for other layers in the hierarchy.
- **Implementation Technology:** The underlying hardware technology used to implement each memory layer.
- **Volatility:** Data persistence characteristics—what happens to stored data when power is removed.

- **Core-level Ownership:** Whether memory layers are shared among processor cores or private to individual cores.
- **Management Responsibility:** Which system components (hardware, OS, application) are responsible for managing each layer.

Understanding these properties is crucial for optimizing system performance and making informed decisions about memory usage in both system design and application development.

2 The CPU-DRAM Performance Gap

The significant performance disparity we observe today, particularly between the CPU and main memory (DRAM), was not always so pronounced. In 1980, processor and memory performance levels were relatively balanced, starting from similar baselines. However, over the subsequent decades, processor performance increased at a dramatically faster rate than DRAM performance. This growing divergence became known as the **CPU-DRAM gap**.

By 1989, this performance gap had widened sufficiently that the first cache memories were introduced as a necessary compensation mechanism. The fundamental goal of caching is to ensure that most CPU operations can be performed using fast cache memory, with only occasional direct access to the slower main memory.

2.1 Principles of Caching

Cache memory serves as a small, fast storage layer that acts as an intermediary for larger, slower memory systems. When designing and implementing cache systems, we must address three fundamental questions:

- **Location:** How do we efficiently locate and retrieve items stored in the cache?
- **Placement:** Which items should we prioritize for cache storage?
- **Replacement:** When the cache is full, which items should be evicted to make room for new data?

These same principles and solutions apply broadly throughout computer systems. For instance, the operating system treats RAM as a cache for disk storage, employing similar strategies and facing analogous trade-offs.

2.2 Why Caching Works: The Principle of Locality

The effectiveness of caching relies heavily on the **principle of locality**, which manifests in two distinct forms:

Temporal Locality: If a program accesses a particular memory address, there is a high probability that the same address will be accessed again in the near future. Common examples include:

- Variables being updated repeatedly.

- Loop control variables and counters.
- Frequently called functions or procedures.

Spatial Locality: If a program accesses a particular memory address, there is a high probability that nearby addresses will be accessed soon. This occurs in scenarios such as:

- Sequential array processing.
- Linear program execution (instructions stored consecutively).
- Structure or object member access.

To exploit spatial locality, cache systems store data in **blocks** (also called cache lines) rather than individual bytes. When a memory location is accessed, the entire surrounding block is loaded into the cache, making subsequent accesses to nearby locations much faster.

The fundamental caching strategy is elegantly simple: maintain the data and instructions that the CPU is most likely to need next in fast memory positioned close to the processor. This approach dramatically reduces average memory access time and significantly improves overall system performance.

3 Random-Access Memory (RAM) Technologies

Within the memory hierarchy, Random-Access Memory (RAM) forms a critical component, but not all RAM is created equal. There are two primary types of RAM technologies, each with distinct characteristics that make them suitable for different roles in the memory hierarchy.

3.1 Static RAM (SRAM)

Static RAM represents the higher-performance tier of RAM technology, with several key characteristics:

- **Data Persistence:** Retains stored values indefinitely, as long as power is maintained to the memory cells.
- **Noise Immunity:** Relatively insensitive to electrical disturbances such as electromagnetic interference.
- **Performance and Cost:** Faster access times than DRAM but significantly more expensive per bit of storage.
- **Primary Applications:** Used extensively for CPU registers and cache memory where speed is paramount.

SRAM achieves its speed and stability through a more complex cell design that uses multiple transistors per bit (typically 6 transistors). This design eliminates the need for refresh cycles but results in larger cell sizes and higher manufacturing costs.

3.2 Dynamic RAM (DRAM)

Dynamic RAM serves as the workhorse of main memory systems, offering different trade-offs:

- **Refresh Requirements:** Stored values must be refreshed every 10-100 milliseconds due to capacitive charge leakage.
- **Electrical Sensitivity:** More susceptible to disturbances and noise compared to SRAM.
- **Performance and Cost:** Slower access times than SRAM but much cheaper per bit, enabling large memory capacities.
- **Primary Applications:** Used for main system memory where capacity and cost-effectiveness are prioritized over speed.

DRAM's simpler cell structure (typically 1 transistor and 1 capacitor per bit) allows for much higher density and lower cost per bit. However, the capacitive storage mechanism requires periodic refresh operations to maintain data integrity, which introduces both complexity and performance overhead.

3.3 RAM Technology Trade-offs

The choice between SRAM and DRAM in system design reflects fundamental trade-offs in the memory hierarchy:

- **Speed vs. Capacity:** SRAM provides faster access for smaller amounts of data, while DRAM offers larger capacity for bulk storage.
- **Cost vs. Performance:** SRAM delivers superior performance at premium cost, while DRAM provides adequate performance at economical pricing.
- **Power vs. Complexity:** SRAM consumes more power per bit but requires simpler control logic, while DRAM is more power-efficient but needs refresh circuitry.

This explains why modern processors use SRAM for cache memory (where every nanosecond matters) and DRAM for main memory (where gigabytes of affordable storage are essential). The memory hierarchy leverages both technologies strategically to achieve an optimal balance of performance, capacity, and cost.

4 Memory Hierarchy Properties

Beyond performance and cost considerations, the memory hierarchy exhibits three critical organizational properties that determine system behavior:

4.1 Management Responsibility

Different layers of the memory hierarchy are managed by different system components:

- **Compiler:** Manages register allocation, deciding which variables reside in CPU registers.

- **Hardware:** Automatically manages all cache levels (L1, L2, L3) through built-in cache controllers.
- **Operating System:** Manages main memory allocation and storage access through virtual memory systems.

4.2 Volatility Characteristics

The memory hierarchy has a clear volatility boundary:

- **Volatile Memory:** Registers, all cache levels, and main memory lose data when power is removed.
- **Non-volatile Memory:** Storage devices (disks, SSDs) and remote storage retain data without power.

This volatility boundary between main memory and storage is critical for system design, as it determines where persistent data must be stored and influences backup and recovery strategies.

4.3 Core-Level Ownership

In multi-core systems, memory layers have different sharing characteristics:

- **Private Resources:** Registers and L1/L2 caches are typically private to each CPU core, providing dedicated high-speed access.
- **Shared Resources:** L3 cache, main memory, and storage are shared among all cores, enabling inter-core communication and data sharing.

This ownership structure balances performance (private caches reduce contention) with functionality (shared resources enable coordination between cores).

5 Address Space and Virtual Memory

Understanding how programs access memory requires examining the concept of **address space**—the total amount of memory that can be addressed by a program. This concept is fundamental to modern operating systems and memory management.

5.1 Address Space Fundamentals

The address space available to programs is determined by several key factors:

- **Architecture Dependency:** The total addressable memory depends on the CPU architecture's address bus width.
- **Program Accessibility:** Address space represents all memory locations that a program can potentially reference.
- **Virtual vs. Physical:** Address space is typically virtual, meaning programs see a uniform memory model regardless of actual physical memory configuration.

5.2 Architecture-Specific Limitations

Different CPU architectures provide varying address space capacities:

32-bit Architecture:

- Maximum addressable memory: $2^{32} = 4$GB.
- Address space division varies by operating system:
  - Windows: 2GB reserved for OS, 2GB available for user processes.
  - Linux: 1GB reserved for OS, 3GB available for user processes.

64-bit Architecture:

- Theoretical limit: 2^{64} addresses (impractical to implement fully).
- Current implementations: Typically 48-bit addressing, providing $2^{48} = 256$TB total.
- Typical division: Half for OS, half for user processes (e.g., 128TB for OS, 128TB for user processes).

5.3 Per-Process Address Space

A critical aspect of modern memory management is that **each individual process receives its own complete address space**. This means:

- Every process on a 64-bit system can theoretically access up to its allocated user portion (e.g., 128TB) of memory.
- Processes are isolated from each other—one process cannot directly access another's memory.
- The operating system manages the mapping between virtual addresses (what processes see) and physical addresses (actual RAM locations).
- Multiple processes can run simultaneously, each believing it has exclusive access to the entire user portion of the address space.

This per-process address space model is the foundation of virtual memory systems, enabling memory protection, process isolation, and efficient memory utilization across multiple concurrent programs.

6 Virtual Memory and Address Translation

The concept of per-process address spaces leads directly to one of the most important innovations in modern computing: **virtual memory**. This system creates an abstraction layer between what programs see (virtual addresses) and the actual physical memory locations.

6.1 Process View vs. Physical Reality

Each process operates under the illusion that it has exclusive access to a contiguous, private address space. However, the physical reality is quite different:

Process Perspective:

- Each process sees a linear address space starting from address 0x0000...
- Memory appears to be organized in standard segments: stack, heap, data, and instructions (text).
- Addresses appear contiguous and predictable.
- The process believes it owns the entire user portion of the address space.

Physical Reality:

- Multiple processes share the same physical memory simultaneously.
- Each process's virtual segments are mapped to potentially different and non-contiguous physical memory locations.
- Physical memory may be fragmented.
- Address translation hardware converts virtual addresses to physical addresses transparently.

The gap between these two views necessitates **address translation**—a fundamental mechanism that maps virtual addresses to physical addresses.

6.2 Program Loading and Address Resolution

The journey from source code to execution involves several stages where memory addresses are determined:

Compilation Process:

1. **Source Code:** Written with symbolic names for variables and functions.
2. **Compilation:** Compiler generates object modules with relative addresses.
3. **Linking:** Linkage editor combines object modules and resolves some addresses.
4. **Loading:** Loader places the program in memory and creates the final address mappings.

Address Types Throughout the Process:

- **Symbolic Addresses:** Variable and function names in source code.
- **Relocatable Addresses:** Relative offsets in object modules.
- **Logical Addresses:** Virtual addresses in the loaded program.
- **Physical Addresses:** Actual memory locations in RAM.

6.3 When Are Logical Addresses Created?

Logical (virtual) addresses are established during the **loading phase**, when the operating system's loader:

- Allocates virtual address space for the new process.
- Maps the executable's segments (code, data, stack, heap) to virtual memory regions.
- Establishes the virtual-to-physical address translation tables.
- Creates the process's virtual memory layout with standardized segment organization.

This loading process transforms a static executable file into a running process with its own virtual address space. The operating system maintains translation tables that map each process's virtual addresses to physical memory locations. This enables multiple processes to coexist safely while each believes it has exclusive access to a private, contiguous memory space. The virtual memory system thus provides the critical abstraction that allows modern multitasking operating systems to run multiple programs simultaneously while maintaining memory protection and efficient resource utilization.

6.4 The Memory Management Unit (MMU)

The address translation process is handled by dedicated hardware called the **Memory Management Unit (MMU)**, which serves as the critical bridge between virtual and physical memory:

- **Hardware Location:** The MMU is typically integrated within the CPU package, operating as part of the processor's memory subsystem.
- **Translation Process:** When the CPU generates a virtual address, the MMU automatically intercepts it and translates it to the corresponding physical address.
- **Transparent Operation:** The translation occurs transparently—the CPU operates as if it's directly accessing physical memory.
- **Performance:** Hardware-based translation enables the high-speed address conversion necessary for practical virtual memory systems.

This hardware approach ensures that virtual memory translation doesn't significantly impact system performance, making the abstraction both powerful and practical.

6.5 Virtual Memory System Challenges

Implementing an effective virtual memory system requires addressing three fundamental challenges:

1. **Address Mapping:** How to efficiently map logical process addresses to physical RAM addresses while maintaining fast translation speeds.

2. **Process Isolation:** How to ensure complete separation of process address spaces so that one process cannot access or interfere with another's memory.
3. **Size Mismatch:** How to handle the significant gap between logical address space size (potentially 128TB per process on 64-bit systems) and available physical RAM (typically measured in gigabytes).

These challenges form the core design problems that virtual memory systems must solve, leading to sophisticated techniques like page tables, memory protection mechanisms, and demand paging. These enable modern operating systems to provide the illusion of unlimited, private memory to each process.

7 Memory Allocation Strategies

Virtual memory systems can be implemented using different allocation strategies, each with distinct trade-offs in terms of simplicity, efficiency, and resource utilization.

7.1 Contiguous Allocation

Contiguous allocation represents one of the earliest and simplest approaches to memory implementation. In this scheme, each process's entire address space is mapped to a single, continuous block in physical memory.

Basic Mechanism

- Address translation uses a simple base-and-offset scheme: a logical address x within a process maps to the physical address base + x.
- The MMU implementation can be very simple, requiring only a single base register per process to store the starting physical address of that process's memory block.
- Context switching involves updating this base register to point to the new process's memory block.

Advantages

Advantages of contiguous allocation include:

- Simple and fast address translation.
- Minimal hardware requirements (e.g., a single base register).
- Potentially good cache locality due to the contiguous memory layout of processes.

Limitations and Drawbacks

Despite its simplicity, contiguous allocation suffers from significant limitations:

- **Internal Fragmentation:** Occurs when processes are allocated more space than they strictly need (e.g., for potential stack/heap growth), leading to wasted memory within their allocated block. Internal fragmentation is free memory inside a process's allocation.
- **External Fragmentation:** As processes are loaded and terminate, physical memory becomes divided into non-contiguous free blocks (holes). This can lead to a situation where enough total free memory exists for a new process, but no single hole is large enough. External fragmentation is free memory between processes' allocations. Efforts typically focus on solving external fragmentation.
- **Managing Free Space (Allocation Algorithms):** To manage external fragmentation, an allocation algorithm decides which free block to use for a new process. Inputs include the list of free ranges and the requested size; output is the chosen range. While not detailed in the PDF's contiguous allocation section, common strategies include:
  - **Best Fit:** Scans all free blocks to find the smallest one that is large enough. Aims to minimize leftover space but can be slow and create tiny, unusable fragments.
  - **First Fit:** Selects the first free block encountered that is large enough. Faster, but can lead to fragmentation at the start of memory.
  - **Next Fit:** Starts searching from where the last allocation ended, potentially distributing fragmentation more evenly.
- **Memory Compaction:** To combat severe external fragmentation, memory compaction relocates processes to consolidate free space into a single larger block. However, this is a very costly operation due to extensive data copying.
- **Inflexibility:** Ill-suited for processes with dynamic memory needs or for efficient shared memory.
- **Memory Size Constraints:** Problems arise if no single contiguous block of physical memory is large enough for a process, or if total demand exceeds physical RAM. This relates to the "Size Mismatch" challenge.

Concluding Remarks on Contiguous Allocation

While conceptually straightforward and used in early systems, the significant limitations of contiguous allocation (especially fragmentation and inflexibility) spurred the development of more sophisticated paging-based virtual memory systems for greater flexibility and efficiency.

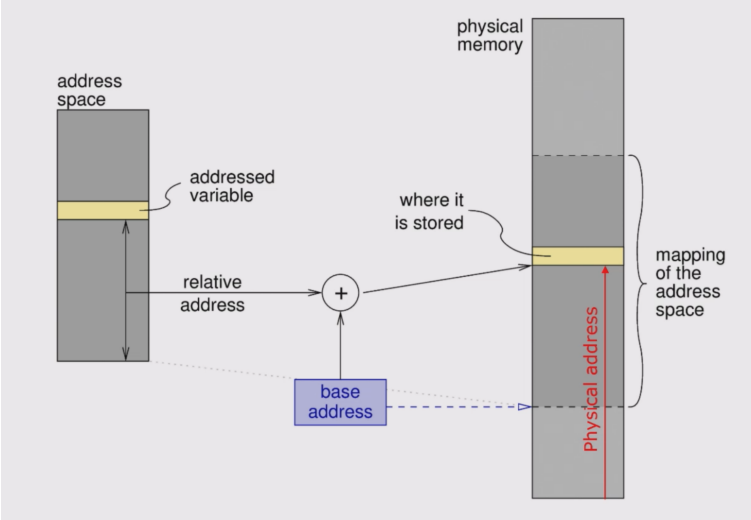

Figure 2: MMU for Contiguous Allocation (representing PDF Figure 2)

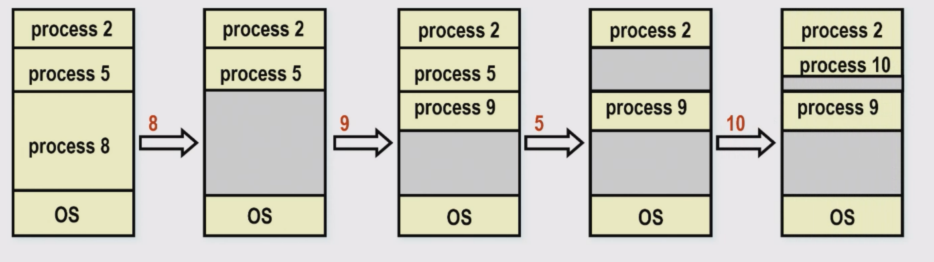

Figure 3: Fragmentation resulting from Allocation Dynamics (representing PDF Figure 3)

8 Paging Concepts

The limitations of contiguous allocation led to the introduction of paging.

8.1 The Concept of Paging

In a paging system, the process address space is divided into fixed-size units called **pages** (typically e.g., 4KB). Physical memory is conceptually divided into fixed-size blocks called **frames**, which are the same size as pages. A key feature is that any virtual page can be mapped to any physical frame, and this mapping is managed by the Operating System and used by the MMU to access memory.

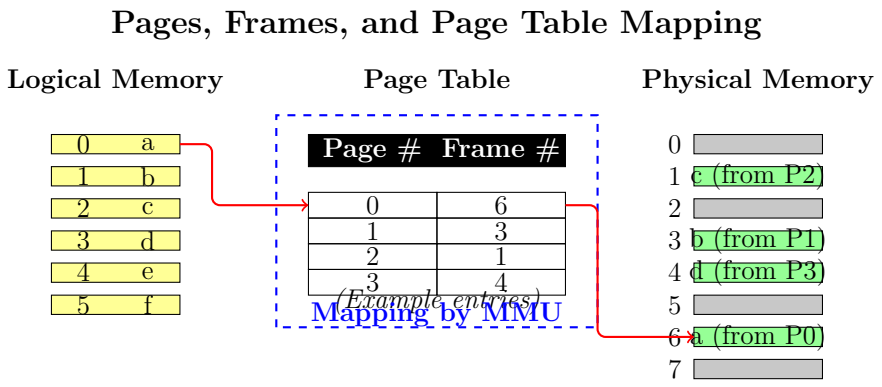

Figure 4: Virtual Memory Paging: Logical pages map to physical frames via a page table (representing PDF Figure 4). Arrows illustrate an example mapping for Page 0.

The Address Translation Architecture (ATA) for paging is visualized as follows:

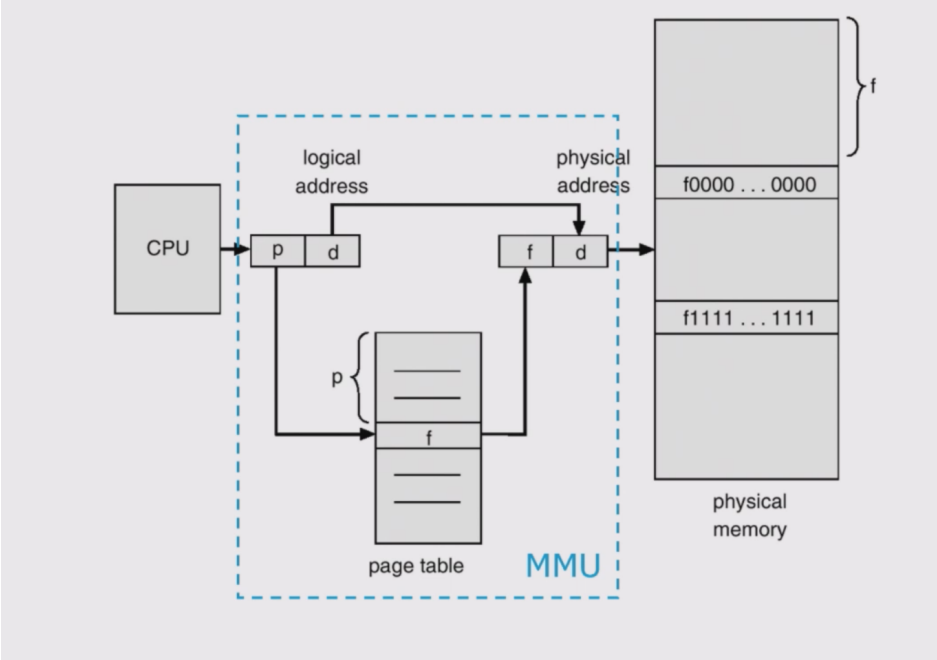

Figure 5: Address Translation Architecture for Paging (representing PDF Figure 5)

The diagram (representing PDF Figure 5) illustrates how a logical (or virtual) address generated by the CPU is translated into a physical address using paging.

8.2 Components Involved in Paging Address Translation

- **CPU (Central Processing Unit):** Generates logical addresses.
- **Logical Address:** Composed of a **Page Number (p)** and a **Page Offset (d)**.
- **Page Table:** A per-process data structure mapping virtual page numbers to physical frame numbers. The page number (p) indexes into this table.

- **Page Table Entry (PTE):** Contains the **Frame Number (f)** for the corresponding virtual page, among other control bits.
- **Physical Address:** Formed by combining the frame number (f) with the page offset (d).
- **Physical Memory (RAM):** Stores the actual data in frames.

The translation process generally follows these steps:

1. The CPU generates a logical address.
2. The logical address is divided into a page number (p) and a page offset (d).
3. The page number (p) is used to index the page table.
4. The PTE at that index provides the physical frame number (f).
5. The frame number (f) is prepended to the page offset (d) to form the physical address.
6. This physical address is used to access physical memory.

This translation is typically performed by the Memory Management Unit (MMU).

8.3 Paging Implementation Details

8.3.1 Optimal Page Size

The choice of page size involves a trade-off:

- **Small pages:** Reduce internal fragmentation but lead to larger page tables.
- **Large pages:** Result in smaller page tables but can increase internal fragmentation.

Definition 8.1 (Page Size Overhead Formula)**:** *If p = page size (bytes), s = process size (bytes), and e = page table entry size (bytes), the total memory overhead is:*

$$Overhead = \frac{se}{p} + \frac{p}{2} \tag{1}$$

The first term, $\frac{se}{p}$, represents the page table overhead. The second term, $\frac{p}{2}$, represents the average internal fragmentation overhead for the last page of a process segment.

Example 8.1 (Page Size Comparison (from PDF))**:** *Consider a 1MB process ($s = 1,048,576$ bytes) with 4-byte page table entries ($e = 4$):*

- ***With 1KB Pages:*** *Total overhead = 4,608 bytes.*
- ***With 64KB Pages:*** *Total overhead = 32,832 bytes.*

To minimize total overhead, the optimal page size can be found by $p_{\text{optimal}} = \sqrt{2se}$.

Note: *The optimal page size depends on both the process size and the page table entry size. Larger processes and larger page table entries favor larger optimal page sizes.*

This overhead (page table size and internal fragmentation) is the cost of implementing paging to gain benefits like eliminating external fragmentation and flexible memory allocation.

8.3.2 Complete Translation Example (32-bit Address)

Example 8.2 (32-bit Address Translation (from PDF))**:** *Consider virtual address* `0x12345ABC` *in a system with 4KB pages (*2^{12} *bytes):*

- *The 32-bit virtual address is split. The bottom 12 bits form the offset (*`0xABC`*).*
- *The top* $32 - 12 = 20$ *bits form the page number (*`0x12345`*).*
- *If the page table entry for virtual page* `0x12345` *indicates it's in physical frame* `0x56789`*.*
- *The physical address is formed by concatenating the frame number and the offset (conceptually, frame number shifted left by offset bits, then offset added):* `0x56789ABC`*.*

8.3.3 Page Table Storage and the Double Memory Access Problem

Page tables themselves are stored in main memory. This can be large (e.g., a 2MB process with 4KB pages needs about 2KB for its page table). A significant issue arises: to access data at a virtual address, the system might first need to access memory to read the page table entry (to find the frame number), and then access memory again to get the actual data. This is the **double memory access problem**, potentially halving memory access speed.

Solution: Translation Lookaside Buffer (TLB) To mitigate this, a specialized hardware cache called the **Translation Lookaside Buffer (TLB)** is used within the MMU.

- The TLB stores recently used page number-to-frame number translations.
- **TLB Hit:** If a translation is in the TLB, the physical address is formed quickly without accessing the page table in memory.
- **TLB Miss:** If the translation is not in the TLB, the page table in memory is consulted, the frame number is retrieved, and the translation is usually then added to the TLB for future use.

The TLB leverages locality of reference for page table entries.

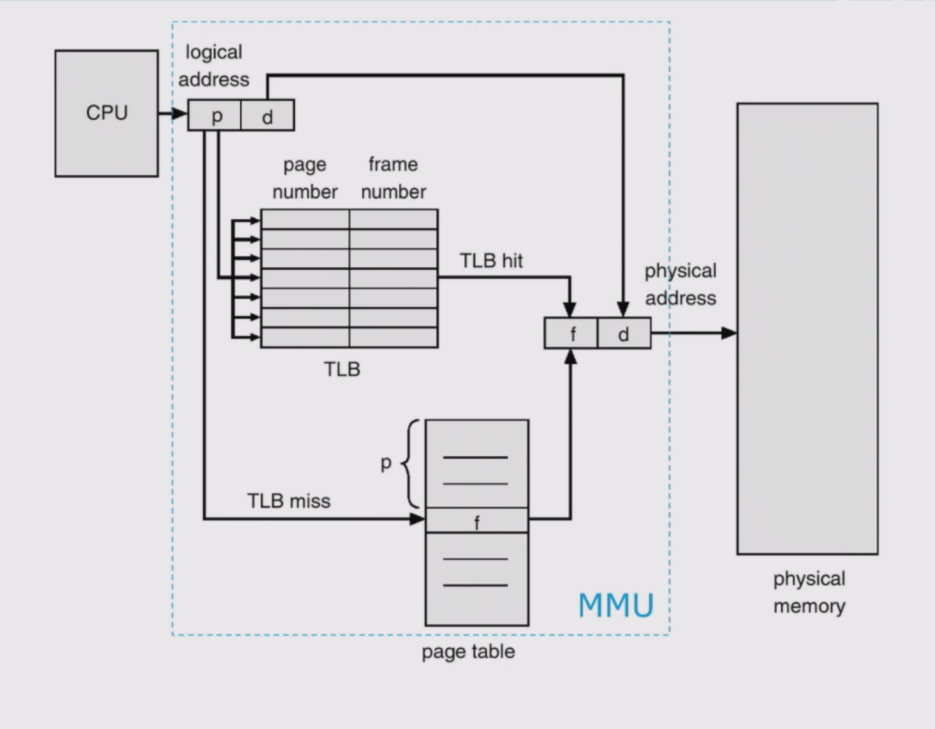

Figure 6: Paging Hardware with TLB (representing PDF Figure 6)

8.3.4 Allocating New Pages

The OS manages a list of free physical frames. When a process needs n new pages (e.g., during creation, 'malloc', stack expansion, or a page fault):

1. The OS assigns n new virtual page numbers to the process.
2. It selects n available frames from its free frame list (these frames need not be contiguous).
3. The allocated frames are removed from the free list.
4. Frame content is initialized (e.g., loaded from disk or zero-filled).
5. The process's page table is updated to map the new virtual pages to their assigned physical frames.

9 Virtual Memory Concepts Concluded

9.1 Logical vs. Physical Memory Distinction

Each process operates under the illusion that it has a large, private address space (logical/virtual memory). The sum of all logical address spaces for concurrent processes can easily exceed the actual physical memory available. However, only a portion of a program's data and instructions needs to be in physical memory at any given time for execution. This allows the logical address space to be much larger than the physical address space, enabling more processes to coexist and improving system efficiency.

9.2 Summary of Key Topics Covered

- **Memory Hierarchy:** (Registers, Caches, RAM, Disk) and its properties.
- **Process Address Space:** Its virtual nature, segments (stack, heap, text, data), and creation timeline.
- **Memory Management Techniques:**
  - **Base+Bound (Contiguous Allocation):**
    - Allocation Algorithms (Best Fit, First Fit, Next Fit as general strategies).
    - Fragmentation (Internal and External) and Compaction.
  - **Paging and Virtual Memory:**
    - Page tables for mapping.
    - Optimal page size considerations.
    - TLB for performance.
    - The concept of virtual memory allowing logical space to exceed physical space.