Operating Systems
Lecture 9: Paging and Virtual Memory

Yonghao Lee

June 3, 2025

1 Foundation Concepts

To begin, it is important to remember that each process operates with its own consistent range of logical addresses—its unique perspective on the memory it can access. In reality, physical memory has a different, more complex organization to manage multiple processes and the operating system itself. Because of this discrepancy between the process's view and the actual physical layout, a system for address translation is essential.

The Memory Management Unit (MMU), typically part of the CPU package, is responsible for this translation. The CPU issues logical (or virtual) addresses, which are sent to the MMU. The MMU then translates these into physical addresses. These physical addresses are subsequently used to access the actual RAM (main memory) via the system bus, allowing the CPU to read or write data.

1.1 Paging

The proposed solution is **paging**. This method involves:

- Dividing a process's logical address space into fixed-size blocks called **pages** (usually 4KB).
- Dividing physical memory into blocks of the same size, called **frames**.

A key feature is that any page from the process can be mapped to any available frame in physical memory. This flexibility effectively eliminates external fragmentation. The mappings between these logical pages and physical frames are stored in a data structure called a **page table**. Importantly, it is not necessary to map the entire logical address space of a process into physical memory at once; only the portions currently in use need to be mapped.

Processes, such as those for large AI models, may utilize a significant portion of their logical address space due to:

- **Vast Model Parameters:** Storing billions of weights and biases.
- **Activations Memory:** Holding intermediate computation values, especially during training.
- **Large Datasets:** Requiring substantial memory for training or input data.
- **Optimizer States:** Storing additional data for learning algorithms during training.

1.2 The Page Table

The **page table** is a crucial data structure that maps a process's logical pages to physical frames in memory. Each process has its own separate page table.

The translation process typically works as follows:

- The logical page number, derived from the logical address, is used as an **index** into the page table.
- The entry found at that index in the page table contains the **frame number** where the corresponding page is stored in physical memory.

Key characteristics and functions of the page table include:

- **Context Switching**: Since each process has its own page table, the operating system must switch the page table the MMU is currently using when a context switch between processes occurs.
- **OS Management**: The page table is populated and managed by the operating system. Its contents reflect the OS's decisions about how and where to map a process's pages.
- **Hardware Utilization**: The MMU uses the page table to perform the translation from logical to physical addresses at hardware speed, facilitating efficient memory access.

1.3 Overheads

Page tables introduce significant overheads. They consume a considerable amount of space as they need to be stored in main memory. Furthermore, accessing the page table for a translation itself involves an additional memory (RAM) access, which is inherently slow.

To mitigate this performance bottleneck, a specialized hardware cache called the **Translation Lookaside Buffer (TLB)** is used, typically integrated within the MMU. The TLB stores a small number (for example, around 64) of the most recently used page-to-frame translations. The process is as follows: if a required translation is present in the TLB (a **TLB hit**), it is used directly and very quickly. If the translation is not found in the TLB (a **TLB miss**), the system must then access the main page table in memory to obtain the translation. This translation is then used to form the physical address and is often loaded into the TLB for potential future use.

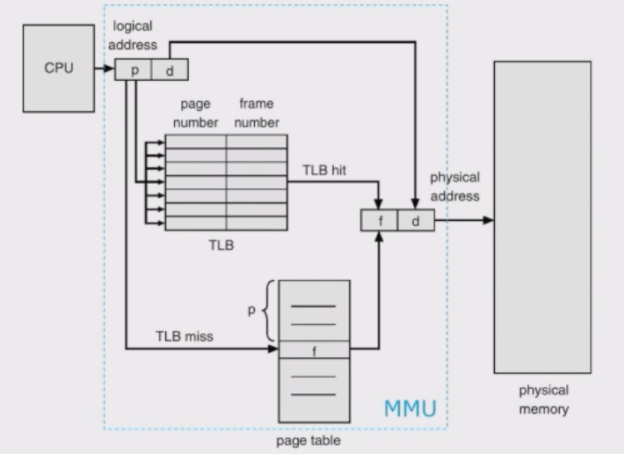

Figure 1: Address Translation with TLB

1.4 Logical Versus Physical Memory

A fundamental question in memory management is whether logical memory or physical memory is typically larger.

Each process operates under the illusion that it runs alone and possesses access to its entire logical memory space. Since the number of processes active on a system is generally

not limited, the total **required logical memory** (which is the sum of the logical address spaces of all processes) is often much larger than the actual **physical memory** available.

This seemingly paradoxical situation is managed by a crucial observation: programs do not utilize their entire address space all the time. For instance:

- During different **program phases**, code such as initialization routines may no longer be needed once that phase is complete.
- **Error handling code** might rarely, or even never, be executed.
- A program might use one data structure extensively and then switch to another, rendering the first one temporarily unused.

Consequently, unused portions of a process's address space do not need to be mapped into physical memory continuously. These parts can be stored on disk and loaded into memory only when actually needed. This approach effectively reduces memory pressure on the physical RAM, allows for more efficient process creation, and enables more processes to run concurrently.

2 Virtual Memory

Virtual memory addresses the discrepancies between physical and logical memory through virtualization, creating the illusion of unlimited memory despite physical memory constraints. This approach decouples processes from the limitations of available RAM, allowing them to operate as if they have access to all the memory they require.

The implementation of virtual memory is typically achieved through **demand paging**, a technique that loads pages from secondary storage into physical memory only when they are actually accessed, and stores them back to disk when they are no longer actively needed.

2.1 Key Concepts and Mechanisms

- **Page Fault:** When the CPU attempts to access a virtual address and the Memory Management Unit (MMU) determines that the corresponding page is not currently resident in physical RAM, a hardware exception called a **page fault** is triggered. This exception transfers control to the Operating System (OS) to handle the missing page.
- **Page Loading:** Following a page fault, the OS locates the required page in secondary storage (e.g., hard disk or SSD) and loads its contents into an available physical frame in RAM. The corresponding page table entry is then updated to reflect the new mapping.
- **Memory Organization:**
  - **Pages** are fixed-size divisions of a process's *virtual address space* (the logical memory layout as seen by the process).
  - Physical **RAM** is partitioned into fixed-size **frames** of the same size as pages.

– A page is considered "resident" or "in memory" when its contents have been loaded from storage into a physical frame. The entire virtual address space need not be in RAM simultaneously—only the actively used portions.

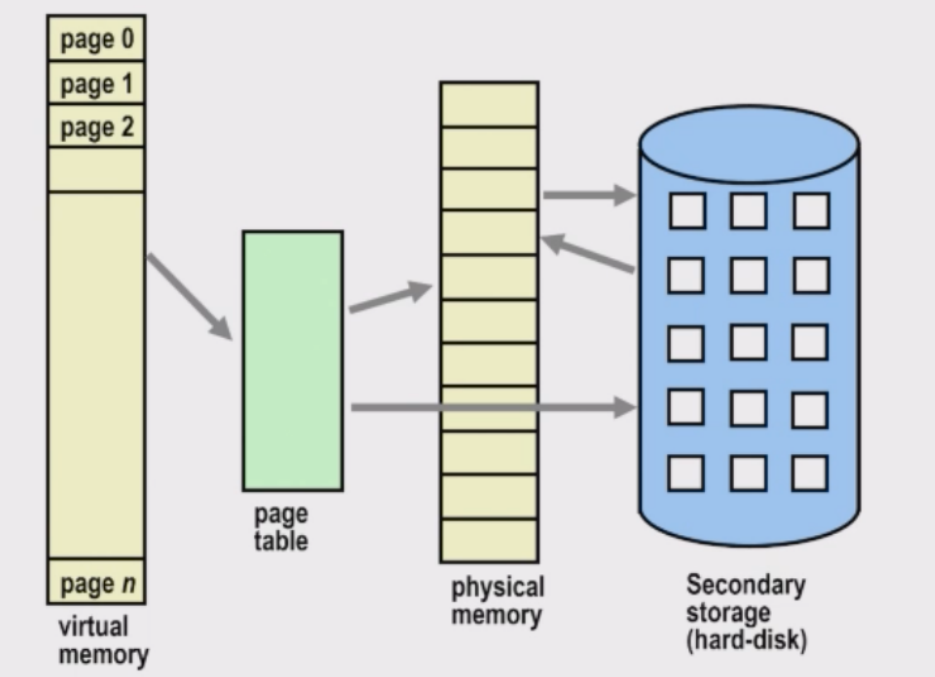

Figure 2: Dynamics of Demand Paging: Virtual-to-Physical Address Translation and Page Management

2.2 Virtual Memory Management Process

The virtual memory system operates through several coordinated mechanisms:

1. **Virtual Address Space:** Each process maintains its own **virtual memory**, providing a logical view of memory divided into fixed-size **pages**. This abstraction isolates processes from physical memory details and from each other.
2. **Address Translation via Page Table:** When the CPU generates a virtual address, the page number portion indexes into the process's **page table**. This table maintains mappings between virtual pages and their corresponding physical memory frames, along with status information (present/absent, permissions, etc.).
3. **Physical Memory Access:** If the page table entry indicates the page is present in **physical memory**, the MMU completes the address translation by combining the frame number with the page offset, allowing the memory access to proceed directly.
4. **Page Fault Handling and Loading:**
   - When the page table entry indicates a page is not resident in RAM, a page fault exception occurs.

- The OS page fault handler locates the required page in **secondary storage**.
- The OS loads the page from disk into an available physical frame.
- The page table entry is updated to reflect the new virtual-to-physical mapping and mark the page as present.
- Execution resumes, and the original memory access completes successfully.

5. **Page Replacement and Eviction:**

- When physical memory is full and a new page must be loaded, the OS employs a **page replacement algorithm** to select a victim page for eviction.
- If the victim page has been modified since loading (indicated by a "dirty" bit), it must be written back to secondary storage to preserve changes.
- Clean pages (unmodified since loading) can be discarded without write-back, as their disk copy remains valid.
- The freed frame becomes available for the incoming page.

2.3 Benefits and Trade-offs

Demand Paging Advantages: Demand paging provides several significant benefits: reduced I/O overhead (only needed pages are loaded), efficient memory utilization (processes need not be entirely resident), faster system response times, and increased system capacity (more processes can be supported concurrently since each uses only a subset of physical memory).

Alternative Approach - Pre-paging: An alternative strategy involves **pre-paging**, where the OS predictively loads pages into memory before they are explicitly accessed, based on spatial locality or other heuristics. While this can reduce page fault frequency when predictions are accurate, incorrect predictions result in wasted I/O bandwidth and memory resources, making demand paging the more commonly adopted approach.

2.4 RAM as a Storage Cache

From a systems perspective, virtual memory implements a classic memory hierarchy where small, fast memory serves as a cache for larger, slower storage. In this arrangement, **RAM functions as a cache for secondary storage** (disk or SSD), with the virtual memory system managing this cache transparently.

The cache management mechanisms mirror traditional cache design principles:

- **Cache Lookup:** Address translation through page tables and the Translation Lookaside Buffer (TLB) serves as the mechanism to locate items in this memory cache, determining whether requested pages are resident in RAM.
- **Cache Population:** Items are stored in the cache using **demand paging**—whenever the CPU attempts to access data not currently in RAM, the system automatically loads the corresponding page from secondary storage into available memory frames.

- **Cache Management Challenge:** The critical question becomes: *which pages should be evicted when RAM is full?* When all physical frames are occupied and a new page must be loaded, the system must select victim pages for removal to make space.

This cache replacement decision is fundamental to virtual memory performance, as poor choices can lead to thrashing (excessive page faults) while optimal selections maintain high cache hit rates and system efficiency.

3 Page Fault Handling and Replacement

3.1 Handling Page Faults: When Pages Are Not in Memory

When the CPU issues a virtual address that references a page not currently present in physical memory (marked as *invalid* in the page table), the requested data cannot be accessed directly. This situation triggers a hardware exception called a **page fault**.

The page fault handling process follows these steps:

1. **Exception Generation:** The MMU detects the invalid page table entry and generates a page fault exception, transferring control to the operating system.
2. **OS Handler Invocation:** The OS page fault handler is invoked through the interrupt vector table, gaining control to manage the missing page situation.
3. **Disk I/O Initiation:** The handler initiates a disk operation to retrieve the required page from secondary storage and load it into an available physical frame using DMA.
4. **Process Blocking:** The faulting process is put to sleep (blocked state) until the requested page arrives from disk, as disk I/O operations are significantly slower than memory access.
5. **CPU Scheduling:** While the page is being loaded, the CPU scheduler runs other ready processes, maintaining system utilization during the I/O wait period.
6. **Completion and Resumption:** When the disk operation completes:
   - The page table entry is updated with the new frame number
   - The valid bit is set to indicate the page is now present
   - The blocked process is awakened and moved to the ready queue
   - The original compiled machine instruction that caused the page fault is re-executed from the beginning, now successfully accessing the loaded page

This transparent handling ensures that page faults are invisible to the application program. The CPU simply re-executes the same machine code instruction that originally triggered the fault, and this time it completes successfully since the required page is now resident in memory.

3.2 The Valid Bit

The **valid bit** is a crucial flag stored in each page table entry that indicates whether the corresponding page is currently present in physical memory. This single bit serves as the primary mechanism for the MMU to determine if address translation can proceed directly or if a page fault must be generated.

Valid Bit States:

- **Valid bit = 1 (set):** The page is currently loaded in physical RAM, and the frame number in the page table entry points to the actual physical location.
- **Valid bit = 0 (clear):** The page is not currently in physical RAM. It may reside on secondary storage, be swapped out, or represent unallocated virtual memory.

Operation During Address Translation: When the MMU encounters a virtual address, it examines the valid bit in the corresponding page table entry. If the valid bit is set (1), address translation proceeds using the frame number. If the valid bit is clear (0), the MMU immediately generates a page fault exception, transferring control to the OS page fault handler.

During page fault resolution, the OS updates both the frame number (to indicate where the page was loaded) and sets the valid bit to 1, enabling future accesses to the page to proceed without faulting.

3.3 Page Replacement: Managing Memory Pressure

The page fault handling process assumes the existence of free physical frames to accommodate incoming pages. However, in practice, physical memory often becomes fully occupied. When this occurs and a new page must be loaded, the system faces a critical resource management decision that requires **page replacement**.

Page replacement involves selecting an existing page currently resident in memory for **eviction**—copying its data to secondary storage (if modified) and using its frame for the new page. The evicted page is referred to as the **victim**.

4 Page Replacement Algorithms

When physical memory is full and a new page must be loaded, the system faces a critical decision: *which existing page should be evicted to make room?* This victim selection process is fundamental to virtual memory performance.

4.1 Implementation Considerations

Most replacement algorithms conceptually operate on frame lists and maintain status bits associated with frames. However, a practical implementation challenge arises: when a frame is selected for eviction, the system must determine *which virtual page currently occupies that frame* in order to invalidate the corresponding page table entry.

This leads to two primary implementation approaches:

- **Frame-to-Page Mapping:** Maintain a reverse lookup structure (such as an inverted page table) that maps each physical frame to the virtual page currently using it. When evicting a frame, this structure immediately identifies which page table entry must be invalidated.
- **Page Table-Centric Approach:** Work directly with page tables by scanning valid entries and maintaining separate free frame lists per process. This approach ignores invalid page table entries and selects victims by examining only currently mapped pages, avoiding the need for reverse lookup structures.

4.2 Theoretical Optimal: Bélády's Algorithm

The **Optimal Algorithm**, also known as **Bélády's Algorithm**, **MIN cache replacement policy**, or the **clairvoyant algorithm**, is a theoretical page replacement strategy that achieves the minimum possible number of page faults.

The algorithm operates on a simple but impossible premise:

Replace the page that will not be used for the longest period of time in the future.

The optimality of this approach can be mathematically proven using exchange arguments. However, this algorithm is infeasible due to the oracle problem—we need to know the future. Therefore, it is used primarily as a benchmark to assess the performance of other algorithms.

4.3 Baseline Comparison: Random Replacement

Random replacement simply evicts any page randomly, representing the other extreme from optimal. While optimal uses full knowledge of everything including the future, random uses no knowledge of anything including the past. This algorithm is also used for comparison—if an algorithm performs worse than random selection, it is clearly suboptimal.

4.4 First-In-First-Out (FIFO)

The FIFO algorithm maintains pages sorted in the order they were loaded to memory, typically implemented as a linked list. The victim is always the first page in this order (the oldest page).

Advantages:

- Simple to implement
- Requires no hardware support
- Independent of access patterns

Disadvantages:

- Pages that have been in memory the longest may actually be used frequently
- Exhibits Bélády's Anomaly

FIFO was historically used in Windows NT due to its simplicity and independence from hardware support.

4.4.1 Bélády's Anomaly

Bélády's Anomaly is a counterintuitive phenomenon where increasing the number of page frames can actually increase the number of page faults with certain algorithms like FIFO.

Expected behavior: Larger memory → fewer page faults
Actual FIFO behavior: Sometimes larger memory → more page faults

4.5 FIFO Demonstration Example

Consider the reference sequence: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4.5.1 Memory with 3 Frames

Time	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Newest	1	2	3	4	1	2	5	5	5	3	4	4
Middle	-	1	2	3	4	1	2	2	2	5	3	3
Oldest	-	-	1	2	3	4	1	1	1	2	5	5
Page Fault?	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	N	N	**Y**	**Y**	N

Table 1: FIFO with 3 frames - **9 page faults**

4.5.2 Memory with 4 Frames

Time	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Newest	1	2	3	4	4	4	5	1	2	3	4	5
Second	-	1	2	3	3	3	4	5	1	2	3	4
Third	-	-	1	2	2	2	3	4	5	1	2	3
Oldest	-	-	-	1	1	1	2	3	4	5	1	2
Page Fault?	**Y**	**Y**	**Y**	**Y**	N	N	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**

Table 2: FIFO with 4 frames - **10 page faults**

This demonstrates the anomaly: adding more memory (4 frames vs 3 frames) resulted in more page faults (10 vs 9), contradicting intuition.

5 Rethinking Replacement Strategy: The Working Set Concept

The discovery of Bélády's Anomaly forces us to reconsider our fundamental assumptions about page replacement. This counterintuitive phenomenon compels us to ask a deeper question: **what are we actually trying to optimize?**

5.1 The Illusion of Future Knowledge

Initially, we might think: *"Ideally, we want to know the future."* After all, Bélády's optimal algorithm achieves perfect performance precisely because it has complete knowledge of future page accesses.

But this raises a critical question: **Why do we want to know the future? What specific information are we seeking?**

The answer lies not in the future itself, but in understanding **which pages the process is currently actively using**.

5.2 The Working Set: The True Objective

What we actually want to identify is the process's **working set**—the collection of pages that constitute the process's current computational focus.

- **Pages in the working set** should be **retained** in memory
- **Pages not in the working set** can be safely **evicted**

This distinction captures the essence of what an optimal page replacement algorithm should achieve.

5.3 Formal Definition

Parametric Definition: The working set $W(t, k)$ at time t is defined as:

$$W(t, k) = \{p : \text{page } p \text{ was accessed in the last } k \text{ memory references}\}$$

where k is a parameter that defines the temporal window of interest.

5.4 The Power of Locality

The working set approach is effective due to the **principle of locality of reference**:

1. **Temporal Locality:** Recently accessed pages are likely to be accessed again soon
2. **Spatial Locality:** Pages near recently accessed pages are likely to be accessed soon

6 Hardware Support for Page Replacement

Memory access occurs at clock speed, necessitating hardware support to track page usage while limiting overhead. Modern systems provide this support through specialized bits in page table entries.

6.1 Hardware-Maintained Status Bits

The MMU maintains additional status information beyond the basic address translation:

- **Reference bit (accessed bit):** Set to 1 when a page is accessed (read or write)
- **Dirty bit (modified bit):** Set to 1 when a page is modified (write operation)

In modern processors, these bits are stored in the page table together with the frame number. In older architectures, they were sometimes stored per physical frame.

6.2 Using Reference Bits: Not Recently Used (NRU)

Since we cannot foresee the future, we attempt to estimate future behavior based on past access patterns using reference bits.

NRU Algorithm:

1. Periodically (on clock interrupts), clear all reference bits
2. When choosing a victim, select randomly among pages with reference bit = 0
3. Logic: Pages with reference bit = 0 have not been accessed since the last clearing

This provides a crude but inexpensive estimation of the least recently used pages.

6.3 Least Recently Used (LRU)

Building on temporal locality principles, LRU selects the page that was accessed furthest in the past for eviction.

Rationale: Pages that were recently used are likely to be used again, so evict the one that was least recently used. This provides a good approximation of the working set and follows the same logic as NRU but with better accuracy.

Implementation Challenges: LRU implementation is expensive and difficult:

- Timestamp maintenance: How many bits are needed for timestamps?
- Finding minimum: Must locate the least recent timestamp on each eviction
- Update overhead: Re-sort data structures on every access?

7 Practical Approximations of LRU

7.1 The Clock Algorithm (Second Chance)

The Clock Algorithm provides a practical approximation of LRU using a circular list structure.

Data Structure: All frames are organized as a circular list with a "hand" pointing to one frame. Each frame has a reference bit.

Algorithm:

1. When eviction is needed, examine the frame pointed to by the hand
2. If the reference bit is set (1):
   - Clear the reference bit (give it a "second chance")
   - Advance the hand to the next frame
   - Repeat from step 1
3. If the reference bit is clear (0):
   - Evict this page (it hasn't been referenced recently)

This algorithm is relatively crude, providing no discrimination between pages with different activity patterns beyond the binary recently-used/not-recently-used distinction.

7.2 Working Set Clock (WSClock) Algorithm

To achieve better approximation of the working set, WSClock tracks time since last reference rather than just binary recent access.

Data Structures:

- Circular list of all frames/pages
- Virtual time variable (per process)
- Each page entry contains: reference bit (R), time of last use, dirty bit

Simplified WSClock Algorithm:
Upon a page fault requiring eviction:

1. Examine the frame pointed to by the hand
2. If R = 1:
   - Set R = 0
   - Update time of last use to current virtual time
   - Advance hand and go to step 1
3. If R = 0 AND (current virtual time - time of last use) ¡ k:
   - Page was referenced within the last k time units (in working set)
   - Advance hand and go to step 1
4. Otherwise: Evict this page

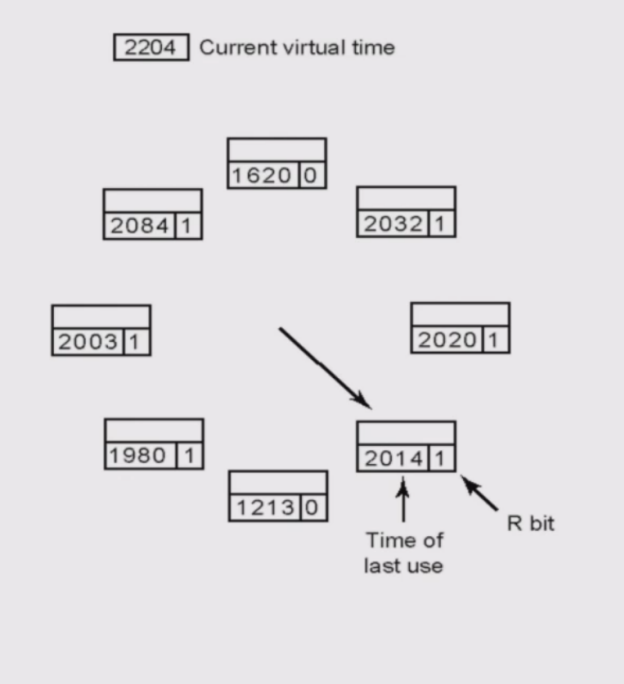

Figure 3: WSClock Algorithm Structure

7.3 Enhanced WSClock: Handling Dirty Pages

The full WSClock algorithm incorporates dirty bit considerations to minimize I/O overhead.

Clean vs. Dirty Pages:

- **Dirty page:** Modified since last disk write; requires write-back before eviction
- **Clean page:** Unmodified; can be discarded immediately (disk copy is current)

Performance Impact:

- Evicting dirty page: Wait for disk write + wait for new page load
- Evicting clean page: Wait only for new page load

Therefore, clean pages are preferred for eviction to minimize I/O latency.

Full WSClock Algorithm:

Upon a page fault requiring eviction:

1. Examine the frame pointed to by the hand
2. If R = 1: Set R = 0, update time of last use, advance hand, go to step 1
3. If R = 0 AND (current virtual time - time of last use) ¡ k: advance hand, go to step 1
4. If dirty bit set: advance hand, go to step 1 (prefer clean pages)
5. Otherwise: Evict this page

Special Cases:

- If dirty but old (age ¿ k): Schedule for background write-back using DMA
- If no candidates found after full scan: Evict the oldest page (clean or dirty)

7.4 Global vs. Local Paging

When process P1 causes a page fault, should the OS choose a victim from P1's pages (local) or from any process (global)?

Local Paging: Victim selection limited to the faulting process's pages

- Provides process isolation and predictable behavior
- May not achieve globally optimal memory utilization

Global Paging: Victim selection from the "best" candidate across all processes

- Achieves better overall system performance
- May lead to unfair resource allocation between processes

Different operating systems adopt different approaches based on their design goals and target workloads.

8 Performance Analysis

Understanding virtual memory performance requires quantitative analysis of the relationship between page fault rates and system performance.

8.1 Key Performance Variables

Let us define the key performance variables for virtual memory systems:

$$p = \text{probability of page fault (disk access)} \quad 0 \leqslant p \leqslant 1 \tag{1}$$

where p represents the fraction of memory accesses that result in page faults.

8.2 Effective Access Time Formula

The total time required for a memory access, accounting for the possibility of page faults, is given by:

Effective Access Time

$$T_{\text{effective}} = p \cdot T_{\text{page fault}} + T_{\text{memory access}} \tag{2}$$

Rationale: After a page fault is resolved, the original memory access instruction must still be re-executed to read the data from RAM.

8.3 Performance Slowdown Metric

The performance degradation due to page faults is quantified by the slowdown factor:

$$\text{Slowdown} = \frac{T_{\text{effective}}}{T_{\text{memory access}}} = \frac{p \cdot T_{\text{page fault}} + T_{\text{memory access}}}{T_{\text{memory access}}} \tag{3}$$

Simplifying:

$$\text{Slowdown} = \frac{p \cdot T_{\text{page fault}}}{T_{\text{memory access}}} + 1 \tag{4}$$

8.4 Contemporary Hardware Performance

Modern systems exhibit the following characteristic timing parameters:

Operation	**Time**
Page fault time (SSD)	40 μs
Memory access time (RAM)	100 ns
Speed ratio	**400:1**

Table 3: Typical performance characteristics

8.5 Performance Case Studies

8.5.1 Case 1: High Page Fault Rate

Consider a system with $p = 0.01$ (1% page fault rate):

$$\begin{aligned} T_{\text{effective}} &= 0.01 \times 40,000 \text{ ns} + 100 \text{ ns} && (5)\\ &= 400 \text{ ns} + 100 \text{ ns} && (6)\\ &= 500 \text{ ns} && (7) \end{aligned}$$

$$\text{Slowdown} = \frac{500 \text{ ns}}{100 \text{ ns}} = 5 \tag{8}$$

Result: The system operates at **5× slower** than optimal performance.

8.5.2 Case 2: Low Page Fault Rate

Consider a system with $p = 0.0001$ (0.01% page fault rate):

$$T_{\text{effective}} = 0.0001 \times 40,000 \text{ ns} + 100 \text{ ns} \tag{9}$$
$$= 4 \text{ ns} + 100 \text{ ns} \tag{10}$$
$$= 104 \text{ ns} \tag{11}$$

$$\text{Slowdown} = \frac{104 \text{ ns}}{100 \text{ ns}} = 1.04 \tag{12}$$