Operating Systems
Lecture 10: Paging & Locality

Yonghao Lee

June 18, 2025

1 Address Translation in Virtual Memory Systems

1.1 The Translation Process

When the CPU generates a memory request, it produces a **virtual address** that consists of two components:

- **Page Number**: The upper bits that identify which virtual page contains the desired data.
- **Offset**: The lower bits that specify the exact location within that page.

The translation mechanism operates through the following steps:

1. The **page number** serves as an index into the process's page table.
2. The **page table address register** contains the base address of the current process's page table.
3. Each page table entry contains two critical pieces of information:
   - **Valid bit (v)**: Indicates whether the page is currently loaded in physical memory.
   - **Frame number**: Specifies which physical frame contains the page data.

1.2 Translation Outcomes

The address translation process can result in two distinct scenarios:

Successful Translation (Valid bit = 1): When the valid bit is set, the page is present in physical memory. The system extracts the frame number from the page table entry and combines it with the original offset to construct the **physical address**. Memory access proceeds immediately without delay.

Page Fault (Valid bit = 0): When the valid bit is clear, the requested page is not currently in physical memory. This condition triggers a **page fault exception**, transferring control to the operating system. The OS then:

- Locates the page on secondary storage (disk/SSD).
- Loads the page into an available physical frame.
- Updates the page table entry with the new frame number and sets the valid bit.
- Resumes the original memory access instruction.

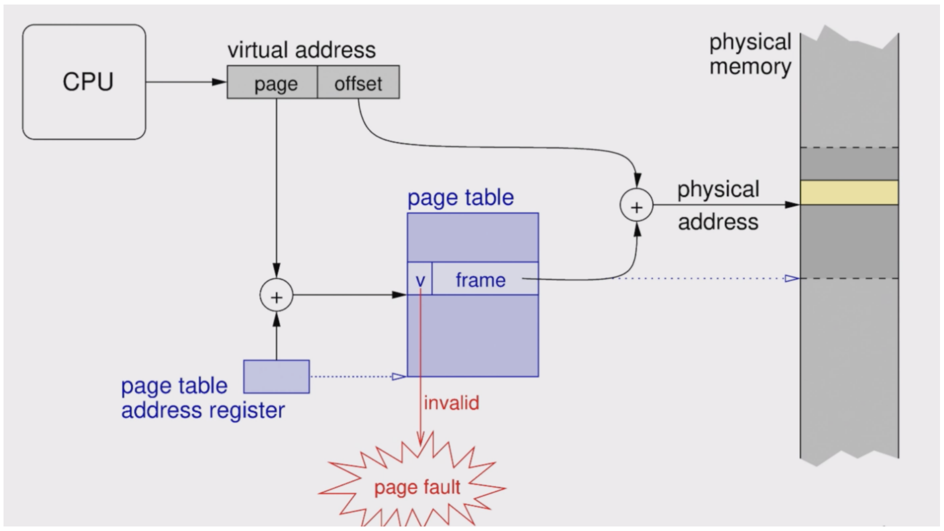


Figure 1: Address Translation Process in Virtual Memory Systems

2 Workloads and Locality Measurement

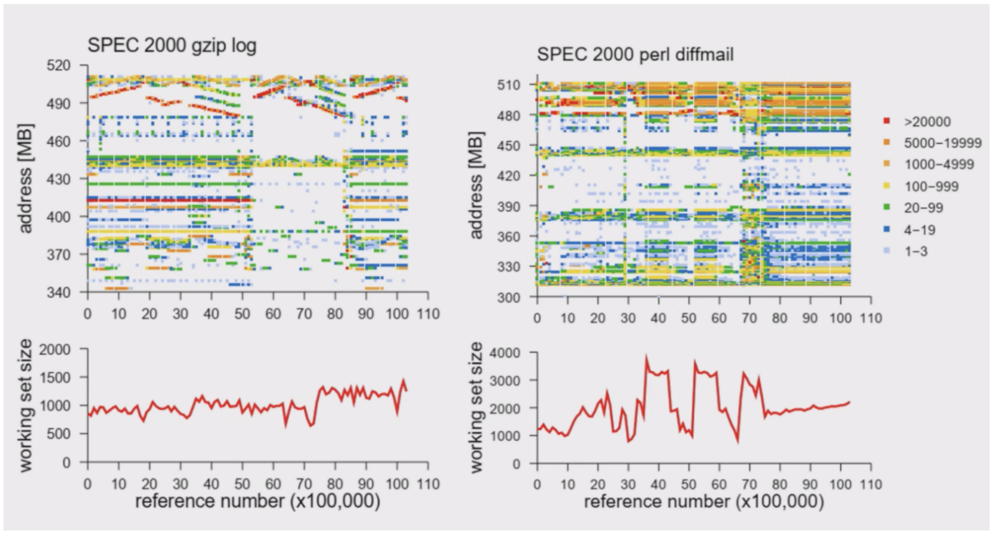


Figure 2: Memory Access Patterns and Working Set Analysis for gzip and perl

Top row - Address Access Patterns:

- X-axis: Program execution time (reference number $\times 100,000$).
- Y-axis: Virtual memory address being accessed (MB).
- Each dot: Individual memory access.
- Colors: Access frequency classification (legend shows ranges 1-3 to >20,000).

Bottom row - Working Set Size:

- X-axis: Program execution time (reference number $\times 100,000$).
- Y-axis: Number of unique addresses actively used.
- Red line: Working set size over time.

2.1 What the Patterns Reveal

The top graphs show *where* in memory the program accesses, while the bottom graphs show *how many different locations* are being used simultaneously.

gzip (left): Clustered dots in the top graph correspond to a stable working set size (∼1000-1500 addresses), indicating good memory locality.

perl (right): Scattered dots across a wider address range correspond to a volatile working set (1000-4000 addresses), indicating poor memory locality.

Having observed these real-world access patterns, we can now examine the theoretical foundations that explain why certain patterns exhibit better locality than others.

3 Temporal Locality: Two Distinct Phenomena

3.1 The Principle of Locality

Definition 3.1: *The principle of locality encompasses two fundamental concepts:*

- ***Temporal Locality****: If we access a memory address, there is a high probability of accessing the same address again in the near future.*
- ***Spatial Locality****: If we access a memory address, there is a high probability of accessing nearby addresses shortly after.*

However, temporal locality actually encompasses two separate memory access behaviors that require different optimization strategies.

3.2 Clustered Accesses

Access Pattern:

```
A A A A A B B B B B C C C C C D D D D D
```

Characteristics:

- Sequential repetition of the same data structure.
- Programs "dig deep" into one data structure, processing it thoroughly before switching to the next.

3.3 Skewed Popularity

Access Pattern:

```
B A B B B B C B B B A B B B B B D B C B
```

Characteristics:

- Frequency-based reuse, where some addresses (like B) dominate.
- Some memory locations are accessed more frequently than others overall.

4 Measuring Locality

4.1 The Challenge

How do we quantify locality in memory access patterns? Two key questions arise:

- How do we know locality really exists?
- How can we characterize the degree of locality for a given address stream?

4.2 Stack Distance Algorithm

Definition 4.1: ***Stack Distance*** *for access i: The number of unique addresses accessed between the current access and the most recent access to the same address. If this is the first access to an address, the stack distance is ∞.*

Algorithm:

1. **Scan the address stream:** For each memory access, maintain a stack-like data structure.
2. **For each address:**
   - Search for the address in the stack from top to bottom.
   - If found: note its depth (stack distance) and remove it from that position.
   - If not found: the stack distance is ∞ (this is its first access).
   - Push the current address to the top of the stack.
3. **Collect depth counters:** Maintain separate counters for each possible distance:

$$\begin{aligned}\text{Hit}[d=1] &= \text{number of accesses found at depth 1}\\ \text{Hit}[d=2] &= \text{number of accesses found at depth 2}\\ &\vdots\\ \text{Hit}[d=\infty] &= \text{number of first-time accesses}\end{aligned}$$

4.3 Example of Stack Distance Calculation

Note: *The following example demonstrates how stack distance counters are maintained as we process each memory access.*

Here is a step-by-step trace of the algorithm for the access stream `1, 3, 1, 1`.

Stack Distance Example

Access	Stack	Distance	Hits

1	1
	
		∞	Hit[1] = 0
			Hit[2] = 0
			Hit[3] = 0
			Hit[∞] = 1

3	3
	1
	
		∞	Hit[1] = 0
			Hit[2] = 0
			Hit[3] = 0
			Hit[∞] = 2

1	1
	3
	
		2	Hit[1] = 0
			Hit[2] = 1
			Hit[3] = 0
			Hit[∞] = 2

1	1
	3
	
		1	Hit[1] = 1
			Hit[2] = 1
			Hit[3] = 0
			Hit[∞] = 2

4.4 Output: Stack Distance Distribution

After processing the entire address stream, the hit counters can be plotted to create a distribution graph:

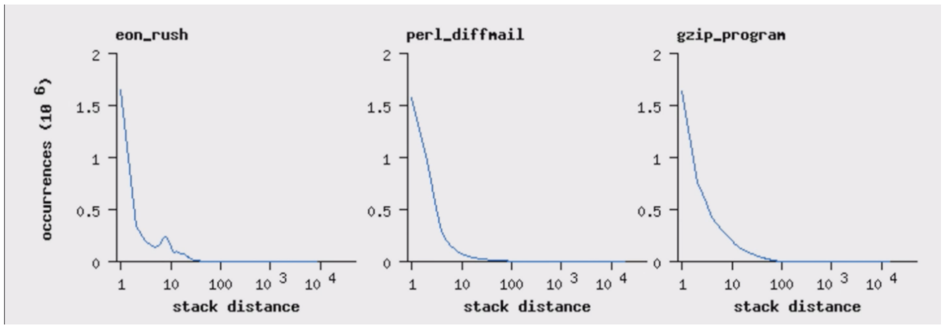


Figure 3: Stack Distance Distribution Showing Locality Patterns

- **X-Axis (Stack Distance):** This axis represents how old a memory access is. A distance of 1 means an address was re-accessed immediately after it was last used (among unique addresses). A distance of d means $d-1$ other unique addresses were accessed in between.
- **Y-Axis (Occurrences):** This axis shows the frequency count for each stack distance, measured in millions (10^6). A high value on this axis indicates that accesses with the corresponding stack distance are very common.

5 Interpreting the Locality Patterns

As shown in Figure 3, all three graphs exhibit a similar, fundamental shape: a very high peak on the left that sharply decays towards the right. This visually confirms the **Principle of Locality**, which states that programs tend to reuse data they have accessed recently. In all cases, small stack distances are far more probable than large ones, predominantly less than 10.

5.1 Using Stack Distance to Evaluate LRU Performance

The stack distance distribution is not merely a theoretical curiosity; it is a powerful and direct tool for evaluating the performance of the **Least Recently Used (LRU)** caching algorithm. The core insight is that a single stack distance profile can predict the miss rate of an LRU cache of any size.

Key Concept 5.1: *Stack distance directly corresponds to LRU cache behavior: an access with stack distance d will be a cache hit if and only if the cache size $k \geq d$.*

The logic is as follows:

1. **Define the Cache:** Assume we have an LRU cache with a capacity to hold exactly k items (pages).
2. **Establish Equivalence:** There is a direct correspondence between the state of the LRU cache and the conceptual stack distance model.
   - By definition, the LRU cache always contains the k most recently used items.
   - The top k positions in the stack distance model also represent the k most recently used items.
   - Therefore, the set of items in an LRU cache of size k is identical to the set of items at the top k positions of the stack.
3. **Define Hits and Misses:** This equivalence allows us to classify every memory access:
   - A **Cache Hit** occurs if the requested item is in the cache. This corresponds to any access with a stack distance $d \leq k$.
   - A **Cache Miss** (or Page Fault) occurs if the item is not in the cache. This corresponds to any access with a stack distance $d > k$.

4. **Calculate Performance:** The probability of a page fault, p, for a cache of size k is therefore the cumulative probability of all accesses having a stack distance greater than k.

$$p(\text{fault}) = P(\text{stack distance} > k)$$

6 Thrashing

6.1 Multiprogramming and CPU Utilization

CPU usage is defined as the fraction of time the CPU spends executing process instructions. Multiprogramming increases CPU utilization by ensuring that when one process becomes blocked (waiting for I/O or other operations), the operating system immediately switches to another ready process rather than letting the CPU remain idle.

However, if the degree of multiprogramming increases excessively—meaning too many processes are competing for system resources—the system can enter a problematic state called **thrashing**. As shown in Figure 4, thrashing occurs when the collective working sets (frequently used pages) of all active processes exceed the available physical RAM.

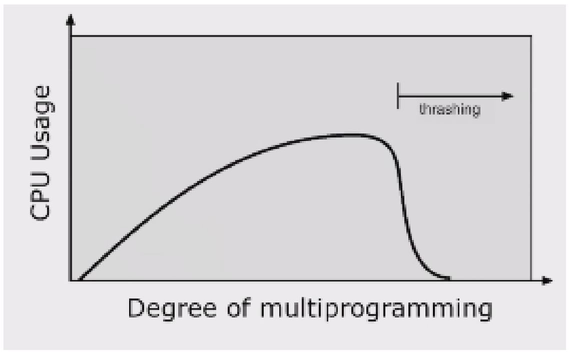


Figure 4: CPU Usage vs. Multiprogramming Degree

6.2 Working Sets and Locality of Reference

Paging is effective because programs exhibit **locality of reference**—they tend to access a small subset of their pages repeatedly over short time periods. This creates a **working set**: the collection of pages a process actively uses during a given time window. Crucially, a process's working set is typically much smaller than its total logical address space. Therefore, we only need to keep the working set in physical memory at any given moment, while the rest can remain on disk.

Since individual working sets are small, multiple processes can simultaneously have their working sets resident in RAM, enabling effective multiprogramming.

6.3 Why Thrashing Occurs

Thrashing occurs when the **combined working sets of all active processes exceed the total available physical memory**. When this happens:

- No process can keep its complete working set in memory
- Each process constantly experiences page faults for pages in its working set
- The system spends more time swapping pages than executing instructions

- CPU utilization paradoxically drops despite high system activity

In this situation, the system becomes trapped in a cycle of constantly evicting pages to disk and loading needed pages back into memory. Rather than executing useful program instructions, the CPU spends most of its time managing memory operations, causing overall system performance to collapse despite high apparent activity.

6.4 Solution: Process Swapping

The solution to thrashing is straightforward but harsh: **limit the degree of multiprogramming**. When the system detects thrashing conditions, it suspends entire processes by swapping their complete working sets from RAM to disk. While this approach is “cruel” to the suspended processes—which make zero progress until resumed—it is necessary to restore system performance. By reducing memory pressure, the remaining active processes can maintain their working sets in RAM and execute efficiently.

We update our diagram of process states as follows:

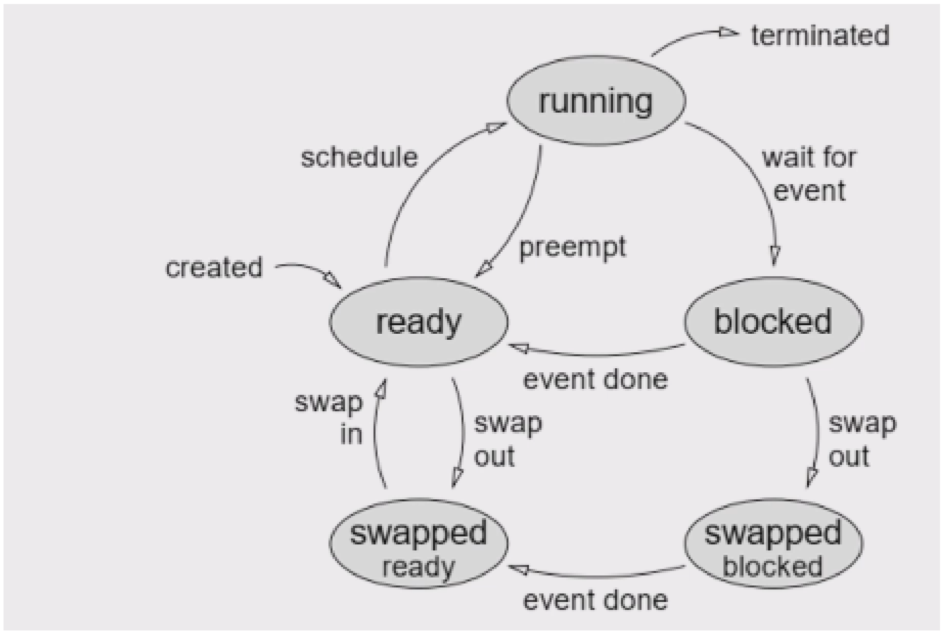


Figure 5: Process State Transitions with Swapping

The diagram illustrates how the traditional process state model is extended to handle memory pressure through swapping. The key additions are the **swapped ready** and **swapped blocked** states, which represent processes whose entire memory images have been moved to secondary storage. When a process is swapped out from the ready state, it enters the swapped ready state and cannot be scheduled until swapped back in. Similarly, blocked processes can be swapped out to free memory, entering the swapped blocked state. Even after their blocking condition is resolved (event done), swapped blocked processes must first be swapped back into memory before returning to the ready state. This swapping mechanism allows the operating system to reduce memory pressure during thrashing by temporarily removing entire processes from physical memory, though at the cost of completely halting their execution until they are swapped back in. Note that there is no way to resume swapped blocked to swap in blocked since there is no use for that.

7 Hierarchical Page Tables

7.1 Page Table Size Problem

Until now, we have discussed only **flat page tables**, where every possible page has an entry in a single table. This approach becomes problematic as address spaces grow larger.

7.1.1 32-bit Systems

For a 32-bit CPU with 4 KB pages:

- Virtual address space: 2^{32} bytes = 4 GB
- Page size: 4 KB = 2^{12} bytes
- Number of pages per process: $\frac{2^{32}}{2^{12}} = 2^{20}$ pages
- Page table entry size: 4 bytes
- Page table size: $2^{20} \times 4 = 4$ MB, which is $\frac{4\text{ MB}}{4\text{ KB}} = 1000$ pages

While 4 MB per process is manageable, the situation becomes dire with 64-bit systems.

7.1.2 64-bit Systems

For a 64-bit CPU, the theoretical address space would be 2^{64} bytes, but this is impractical. Most CPUs limit the logical address space (LAS) to 2^{48} bytes for efficiency.

Consider a system with:

- Logical address space: 2^{48} bytes = 256 TB
- Page size: 4 KB = 2^{12} bytes
- Page table entry size: 8 bytes = 2^3 bytes
- Physical memory: 2^{33} bytes = 8 GB, which is $\frac{2^{33}}{2^{12}} = 2^{21}$ frames

The problem:

- Number of pages: $\frac{2^{48}}{2^{12}} = 2^{36}$ pages
- Page table size: $2^{36} \times 2^3 = 2^{39}$ bytes = 512 GB
- Available physical memory: Only 2^{33} bytes = 8 GB

The page table (512 GB) cannot even fit in the available physical memory (8 GB)! This creates an impossible situation where the page table itself requires 64 times more memory than the system possesses.

Moreover, most processes use only a tiny fraction of their 256 TB virtual address space, making it wasteful to allocate page table entries for unused regions. This necessitates a more sophisticated approach: **hierarchical page tables**.

The HPT breaks up the logical address space into multiple page tables. A simple technique is a two-level page table.

1. **Virtual Address Breakdown**:

   The CPU requests Virtual Address 6, which is `000110` in binary. This address is split into three distinct parts based on the architecture's design:

   - `Pag1` **(Page Directory Index)**: `00`
   - `Pag2` **(Page Table Index)**: `01`
   - `Offs` **(Offset)**: `10`

2. **First Level Lookup (Page Directory)**:

   The MMU (Memory Management Unit) uses the first index, `Pag1` (`00`), to look inside the **Top Level Page Table**. The entry at index 0 in this table contains the frame number 4 (and a valid bit of 1, meaning the entry is valid). This indicates that the required second-level page table is located in **physical memory frame 4**.

3. **Second Level Lookup (Page Table)**:

   Next, the MMU accesses the contents of physical frame 4. It then uses the second index, `Pag2` (`01`), to look inside this second-level page table. The entry at index 1 points to frame number **1**. This tells the MMU that the actual data page is located in **physical memory frame 1**.

4. **Final Physical Address Calculation**:

   The final physical address is calculated by taking the base address of the data frame (frame 1) and adding the offset.

   - The offset `Offs` is `10` in binary, which is 2 in decimal.
   - The calculation is:

   $$\text{Final Address} = \text{Start of Frame 1} + \text{Offset} = 4 + 2 = 6$$

Essentially, instead of holding one table with one million entries, we can hold 1025 tables with 1K entries each.

So, each table is the size of one page, and each virtual address generated by the CPU is split into 3 parts: P_1 represents the entry in the top-level table, P_2 represents the entry in the second-level table, and d represents the offset in the final frame in RAM.

For example, if page 5200 is held in frame 1000, the top-level page table is in frame 17, and the relevant second-level table is in frame 700: we first go to frame 17. If $P_1 = 5$, we go to the 5th entry and find the value 700. This takes us to the second-level table. If $P_2 = 80$, we go to the 80th entry in that table to find the value 1000, which is the frame where page 5200 is held. Finally, we apply the offset d to find the specific address.

7.2 x86/x64 Hierarchies

32-bit (x86)

A 2-level hierarchy is common:

$$\begin{aligned}\text{Address} &= 10(\text{Dir}) + 10(\text{Table}) + 12(\text{Offset}) \text{ bits}\\ \text{Entries per table} &= 2^{10} = 1024\\ \text{Total addressable space} &= 2^{32} = 4\text{GB}\end{aligned}$$

64-bit (x64)

4-level hierarchy: $4 \times 9 + 12$ bits for $2^{48} = 256$ TB of addressable space.

$$\text{PGD} \to \text{PUD} \to \text{PMD} \to \text{PTE}$$

5-level hierarchy: $5 \times 9 + 12$ bits for $2^{57} = 128$ PB of addressable space.

$$\text{PGD} \to \text{P4D} \to \text{PUD} \to \text{PMD} \to \text{PTE}$$

Common x64 properties:

$$\begin{aligned}\text{Entries per table} &= 2^9 = 512\\ \text{Page size} &= 4\text{KB} = 2^{12} \text{ bytes}\end{aligned}$$

The number of entries is fixed because each table is constrained to fit within a single 4KB page (4KB page size / 8 bytes per entry = 512 entries).

Linux x64 naming convention:

- PGD: Page Global Directory
- P4D: Page Level 4 Directory (5-level only)
- PUD: Page Upper Directory
- PMD: Page Middle Directory
- PTE: Page Table Entry

7.3 Benefits of Hierarchical Page Tables

The primary benefit of HPT is its significant memory efficiency, which directly solves the size problem of flat page tables. This is achieved as follows:

- **On-Demand Allocation:** With a hierarchical structure, second-level page tables do not need to be allocated in memory unless the corresponding memory regions are actually being used by the process. These tables are created on-demand as the process accesses new areas of its address space.
- **Size Reflects Actual Usage:** As a result, the total memory consumed by the page table structure reflects the process's *actual* memory footprint, rather than its vast *theoretical* address space. For most processes, which use only a small fraction of their available 256 TB address space, this results in enormous memory savings.

7.4 Page Fault Classification

Page faults in hierarchical page table systems can be categorized into two fundamental types based on what component is missing from memory:

1. **Page Table Structure Fault:** One or more levels of the page table hierarchy are not present in RAM. This occurs when the translation infrastructure itself is missing or incomplete.
2. **Content Page Fault:** The page table structure exists and is complete, but the target data page is not present in RAM. This involves the actual content that the virtual address is meant to access.

On-Demand Page Table Creation

An important characteristic of modern memory management is that intermediate page tables are allocated lazily. Most second-level page tables remain unallocated since processes typically use only a small fraction of their virtual address space. These missing page tables are not stored anywhere—neither in RAM nor on disk—because they would contain only invalid entries.

When a page fault occurs due to a missing page table, the operating system may choose to honor the fault through the following process:

1. **Allocate a new physical frame** for the missing page table
2. **Initialize the frame with zeros**, creating empty page table entries
3. **Update the parent page table entry** to point to this new frame and mark it as present
4. **Handle the original memory access** by populating the appropriate entry in the newly created page table

This lazy allocation strategy significantly reduces memory overhead by creating page table structures only when they are actually needed, rather than pre-allocating the entire hierarchical structure for each process.

7.5 Page Table Entry - Reminder

Recall that when the page table entry has a valid bit equal to 1, it means that the page is loaded into a frame and mapped. If the valid bit is 0 and some internal flags are set, it means that the page is swapped to disk and can be loaded with data from the swap space. If the page table entry is all zeros, no mapping exists.

7.5.1 Null Pointer Dereference Detection

What happens at runtime when we execute `*ptr = 123` if the pointer was initialized as a null pointer? How is this exception implemented at the system level?

The key insight is that the compiler does *not* inject pseudo-code with high overhead to check if `ptr` is null before each dereference. Instead, the detection mechanism relies on a clever operating system design: **page 0 is intentionally left unmapped**.

When a null pointer (address `0x00000000`) is dereferenced:

1. The CPU attempts to access memory address 0
2. The Memory Management Unit (MMU) discovers that page 0 is unmapped in the process's virtual address space
3. The MMU generates a page fault exception
4. The operating system's page fault handler catches this exception
5. Since the access targets the unmapped null page, the OS delivers a `SIGSEGV` signal to the process

7.6 32-bit Memory Management Example

This diagram illustrates the relationship between a program's virtual memory and the computer's physical RAM. The components are as follows:

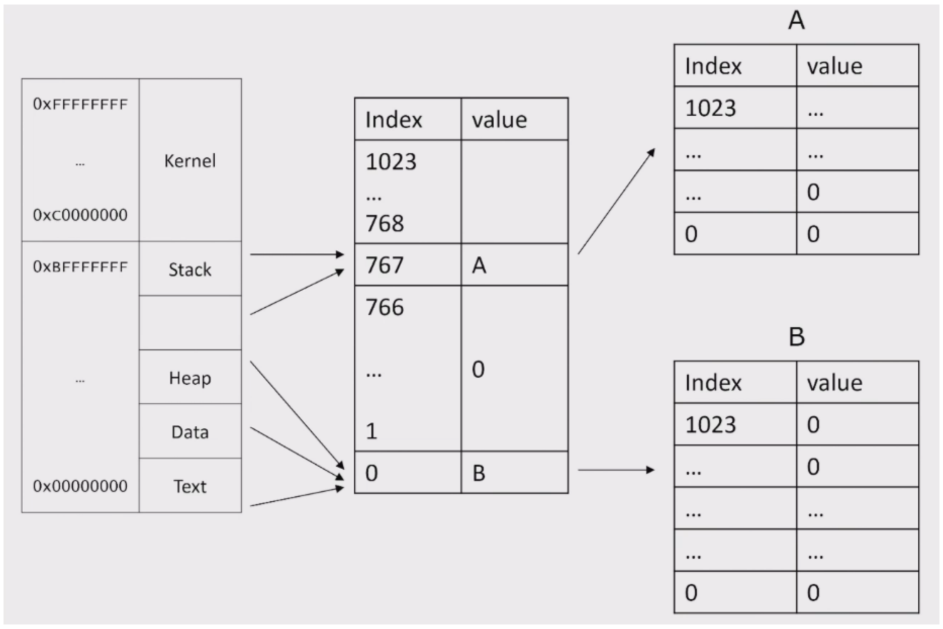


Figure 6: 32-bit Hierarchical Page Table Translation

1. **Virtual Address Space (Left Box):** This represents the private 4GB memory layout as seen by the program. It is organized into distinct segments:
   - `Kernel:` Reserved space for the Operating System.
   - `Stack:` For local variables and function calls.
   - `Heap:` For dynamically allocated memory.
   - `Data:` For global and static variables.
   - `Text:` For the executable code.
2. **Page Table (Middle Box):** This table acts as a translator or map.
   - The *Index* column represents a virtual page number.
   - The *Value* column stores a pointer to where that virtual page is located in physical RAM. For example, index 767 points to frame 'A'.
3. **Physical Pages / Frames (Right Boxes):** The boxes labeled 'A' and 'B' represent actual, fixed-size blocks of physical RAM.
   - These frames hold the real data for the program.

The 32-bit Linux memory management example effectively demonstrates a core principle of modern operating systems: the efficient mapping of a large virtual address space onto a limited physical memory. The use of a hierarchical page table, as illustrated, is crucial to this efficiency. By only creating active mappings for the small subset of virtual pages that are actually in use (represented by frames 'A' and 'B'), the system avoids allocating a massive, mostly empty page table for every process. This "on-demand" mapping mechanism saves significant memory and is precisely what allows a process to operate under the illusion of having a vast, contiguous memory space while only consuming a fraction of that in physical RAM.

8 64-bit Linux Hierarchical Page Table Structure

8.1 Key Differences from 32-bit

The 64-bit Linux system extends the hierarchical page table concept to handle much larger address spaces efficiently.

Address Structure Instead of the 32-bit system's two levels using 10 bits each, the 64-bit system uses **four levels** with **9 bits per level**. The virtual address format becomes [9][9][9][9][12] bits, though only 48 bits of the full 64-bit address space are actually used, providing $2^{48} = 256$ terabytes of addressable space.

Virtual Memory Layout The virtual memory maintains the same conceptual organization as 32-bit systems but with dramatically larger capacity. The kernel space resides at the top addresses, followed by the stack region, heap, data segments, and text segments at the bottom. Each segment can now accommodate vastly larger programs and datasets.

8.2 Four-Level Page Table Hierarchy

The translation process now involves four distinct page table levels:

1. **PGD (Page Global Directory)** – The top-level table containing 512 entries
2. **PUD (Page Upper Directory)** – Second-level tables, also with 512 entries each
3. **PMD (Page Middle Directory)** – Third-level tables with 512 entries each
4. **PTE (Page Table Entry)** – Final level that points to actual physical memory frames

8.3 Address Coverage and Mapping

Each entry in the top-level PGD table now covers an enormous $2^{39} = 512$ gigabytes of virtual address space, compared to just 4 megabytes in the 32-bit system. This means that with only a few active entries in the PGD, we can map the entire virtual address space used by typical applications.

Specific Mappings in the Diagram The diagram shows two key active mappings in the PGD table. Index 255 points to table A, which handles the stack region around virtual addresses starting with `0x7FFF...`. Index 0 points to table B, which manages the text and data regions near virtual address `0x0000...`. All other PGD entries remain zero, indicating unmapped virtual address ranges.

8.4 Multi-Level Translation Process

When translating a virtual address, the system extracts four 9-bit indices and one 12-bit offset. The first 9 bits select an entry in the PGD table. If that entry is valid, it points to a PUD table. The next 9 bits select an entry in that PUD table, which points to a PMD table. The third 9-bit field selects a PMD entry pointing to a PTE table. Finally, the fourth 9-bit field selects the PTE entry containing the physical frame number, which combines with the 12-bit offset to form the final physical address.

8.5 Memory Efficiency Benefits

The hierarchical structure provides massive memory savings for 64-bit systems. A flat page table for the full 2^{48} address space would require 2^{36} entries at 8 bytes each, totaling 512 gigabytes just for the page table. In contrast, the hierarchical approach only allocates page tables for virtual address regions actually in use by the process.

Typical Process Requirements A typical process might only need the top-level PGD table plus a few second, third, and fourth-level tables for the specific virtual address ranges it uses. This could total just tens of kilobytes instead of hundreds of gigabytes, representing a space savings of over 99.99%.

8.6 Sparse Address Space Handling

The 64-bit hierarchical structure excels at handling sparse address spaces where programs use small portions of the available virtual memory scattered across the vast 256-terabyte range. Modern applications can have large gaps between their text, data, heap, and stack regions without wasting physical memory on unused page table entries.

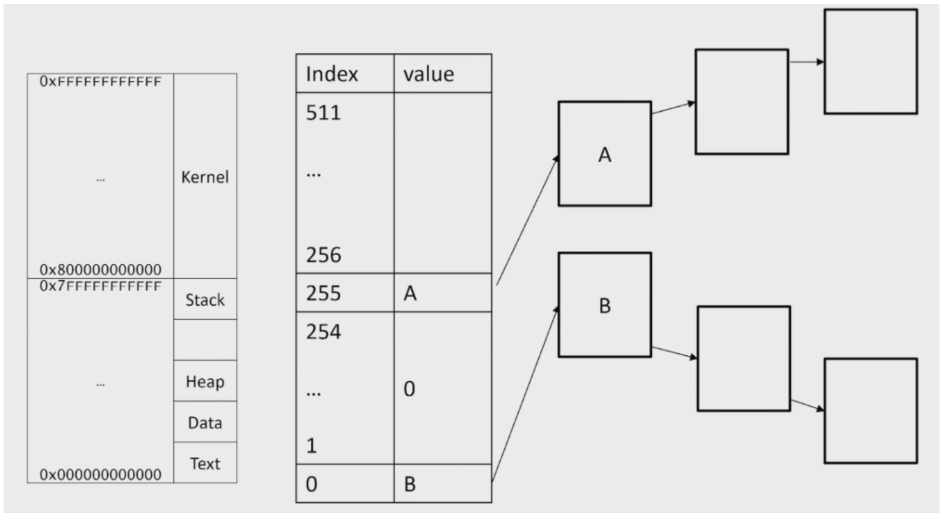


Figure 7: 64-bit Linux Page Table Structure

8.7 Hashed Page Table

Hashed page tables use a hash function to map page numbers to hash table entries, with collision chains to handle multiple pages mapping to the same hash value. This approach is efficient for sparse address spaces.

Algorithm 1 Hashed Page Table Address Translation

```
Require: Logical address $(p, d)$ where $p$ is page number, $d$ is displacement
Ensure: Physical address $(r, d)$ where $r$ is frame number
1: $hash\_index \leftarrow hash(p)$
2: $entry \leftarrow hash\_table[hash\_index]$
3: while $entry \neq NULL$ do
4:     if $entry.page\_number = p$ then
5:         $r \leftarrow entry.frame\_number$
6:         return $(r, d)$
7:     end if
8:     $entry \leftarrow entry.next$
9: end while
10: throw PageFaultException
```

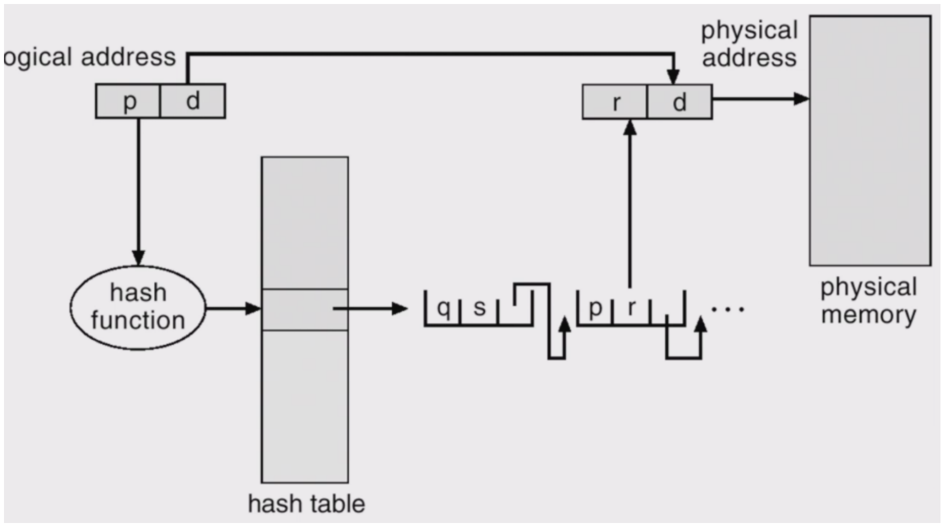


Figure 8: Hashed Page Table Implementation

8.8 Inverted Page Table

Inverted page tables maintain a single system-wide table where each entry corresponds to a physical frame and contains the process ID and page number of the process occupying that frame. This saves memory but requires linear search for address translation.

Algorithm 2 Inverted Page Table Address Translation

Require: Logical address (pid, p, d) where pid is process ID, p is page number, d is displacement

Ensure: Physical address (i, d) where i is frame number

```
1: for $i = 0$ to $size\_of\_IPT - 1$ do
2:     if $IPT[i].pid = pid$ and $IPT[i].page = p$ then
3:         return $(i, d)$
4:     end if
5: end for
6: throw PageFaultException
```

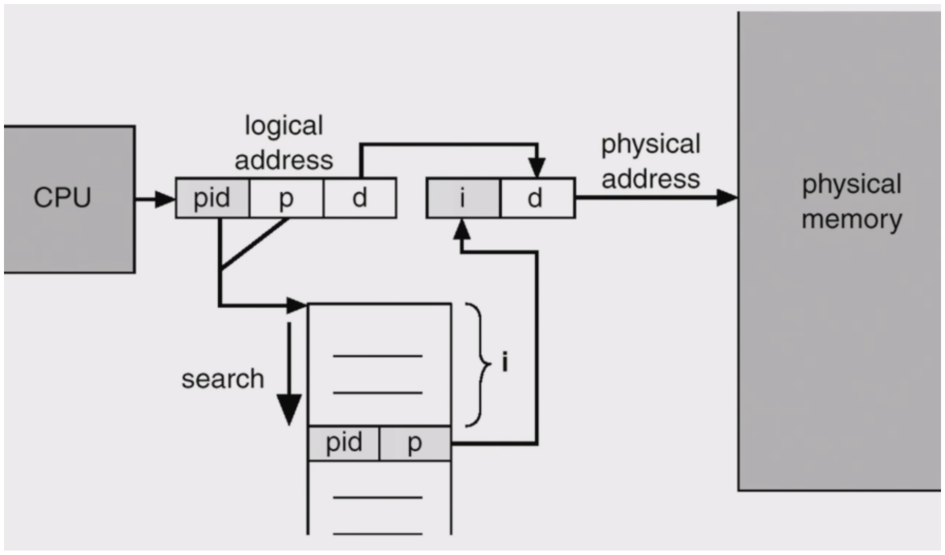


Figure 9: Inverted Page Table Structure

9 A Note About Security

We have discussed three fundamental bits for each page: valid, referenced, and dirty. However, two additional bits are crucial for security: **write** and **execute**.

9.1 Memory Protection Mechanisms

9.1.1 Write Protection

If a page contains code and we make it executable but set the write bit to 0, we prevent attackers from overwriting existing code. This ensures that even if an attacker gains write access to a memory address, they cannot override existing executable code and run malicious instructions.

9.1.2 Execute Protection (NX Bit)

When an attacker can write to a buffer in our process—whether on the heap or stack—setting the executable bit to 0 prevents them from executing any injected code. This creates a fundamental barrier: data areas remain non-executable, preventing code injection attacks.

9.2 Address Space Layout Randomization (ASLR)

Traditional memory layout models present processes with a deterministic structure: text segment at the bottom, followed by data, heap, and stack segments. However, this representation is not entirely accurate for modern operating systems.

In contemporary systems, each process execution loads components non-deterministically. The operating system introduces controlled randomization in the placement of code segments, making memory addresses unpredictable across different program executions.

9.2.1 ASLR Demonstration

The terminal output demonstrates this concept clearly. Multiple executions of the same program show the `main()` function loaded at entirely different virtual addresses:

- Execution 1: 108607404151113
- Execution 2: 107755702083913
- Execution 3: 110972597260617
- Execution 4: 103606596351305
- Execution 5: 101557496613129

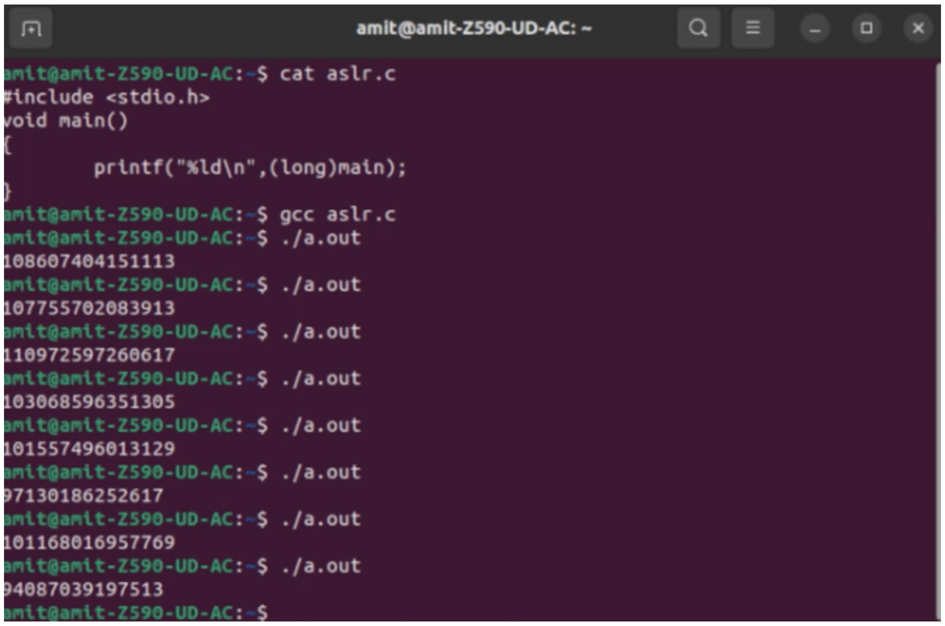
This randomization significantly complicates attack vectors that rely on predictable memory layouts, as attackers cannot reliably determine where specific code or data structures will reside in memory.

9.3 Combined Security Impact

These mechanisms work synergistically to provide robust protection:

1. **Write protection** prevents code modification
2. **Execute protection** prevents code injection
3. **ASLR** prevents predictable targeting

Together, they form a comprehensive defense system that makes successful memory-based attacks significantly more difficult to execute.



```
amit@amit-Z590-UD-AC:~$ cat aslr.c
#include <stdio.h>
void main()
{
        printf("%ld\n",(long)main);
}
amit@amit-Z590-UD-AC:~$ gcc aslr.c
amit@amit-Z590-UD-AC:~$ ./a.out
108607404151113
amit@amit-Z590-UD-AC:~$ ./a.out
107755702083913
amit@amit-Z590-UD-AC:~$ ./a.out
110972597260617
amit@amit-Z590-UD-AC:~$ ./a.out
103068596351305
amit@amit-Z590-UD-AC:~$ ./a.out
101557496013129
amit@amit-Z590-UD-AC:~$ ./a.out
97130186252617
amit@amit-Z590-UD-AC:~$ ./a.out
101168016957769
amit@amit-Z590-UD-AC:~$ ./a.out
94087039197513
amit@amit-Z590-UD-AC:~$
```

Figure 10: ASLR Demonstration Output