Sistemi Operativi

Corso di Laurea in Informatica

Sapienza Università di Roma

a.a. 2026/2027

Marco Raoul Marini

marini@di.uniroma1.it

Gabriele Tolomei

tolomei@di.uniroma1.it

Lecture 2.6: Virtualization VI

TLB Acceleration and Multi-Level Paging

Opening Thought

> *"A TLB makes virtual-memory-based address translation viable. Without one, each memory reference would require at least an extra memory access, making programs run at least twice as slowly. With one, programs run nearly as fast as if address translation didn't exist."*
> *— Remzi & Andrea Arpaci-Dusseau, OSTEP Ch. 19*

Paging elegantly eliminates external fragmentation, but introduces two costs: (i) the time cost of two memory accesses per operation, and (ii) the space cost of enormous page tables for 64-bit address spaces.
Today we conquer both — with the TLB and with multi-level page tables — and in doing so complete the picture of how modern CPUs perform memory translation at nearly zero overhead.

Lecture Roadmap

This lecture is organized into **9 major sections**:

1. **Recap & The Two Costs of Paging** — The problems we must solve
2. **The TLB: Concept** — A hardware cache for page table entries, and why locality makes it work
3. **TLB: Control Flow and Hardware vs. Software Managed** — Two design philosophies
4. **TLB Performance: Coverage, Thrashing, and Huge Pages** — When the TLB helps and when it does not
5. **Multi-Level Page Tables: Motivation and Core Idea** — A tree of page tables
6. **Two-Level Page Tables: Mechanics** — Step-by-step translation
7. **Beyond Two Levels: 64-bit Systems** — The x86-64 four-level walk (RISC-V and five-level in the appendix)
8. **OS Responsibilities and Advanced Topics** — On-demand page-table allocation and huge pages (inverted tables, THP and security in the appendix)
9. **Worked Examples, Summary & Takeaways** — Exam preparation

> *Extended material (derivations, extra worked examples, real systems, exam practice) is in the **Appendix** (A.1–A.9): each section divider points to its appendix counterpart.*

Prerequisite Check

To follow this lecture, ensure you are comfortable with:

- **Paging basics** (Lecture 2.5): VPN, offset, PTE fields (valid, PFN, R/W/X), flat page table layout.
- **Address translation** (Lecture 2.5): `PA = PFN × page_size + offset`.
- **CPU cache principles** (Lecture 1.1): hit/miss, hit rate, AMAT formula.
- **Context switch** : saving/restoring hardware registers including PTBR.
- **Privilege modes** (Lecture 1.2): kernel-only instructions, privileged registers.
- **Locality of reference** (Lecture 1.1): temporal and spatial locality.
- **Hex, shifts (`>>`) and masks (`&`):** if these feel rusty, Appendix A.2 opens with a three-slide refresher.

If any of these feel uncertain, review the corresponding lecture before proceeding.

§ 1 — Recap & The Two Costs of Paging

📎 *Extended material → Appendix A.1*

Where We Left Off: Paging Solves Fragmentation

Lecture 2.5 showed that paging is the right long-term answer for memory virtualization:

- **No external fragmentation:** any physical frame can hold any virtual page.
- **Fine-grained protection:** per-page R/W/X bits.
- **Flexible sharing:** two processes can map different VPNs to the same PFN.
- **Foundation for swapping:** individual pages can be moved to disk on demand.

But paging introduced **two new costs** that, left unaddressed, would make it completely impractical:

```
Cost 1 (Time):  Every memory access requires an extra memory access
                to look up the page table → programs run 2× slower.

Cost 2 (Space): A flat page table requires entries for every virtual page,
                whether used or not → up to 512 GB per process on 64-bit.
```

Cost 1 — Time: Two Memory Accesses Per Data Access

Without any optimization, every single memory operation in a paged system requires **two physical memory accesses**:

```
Program executes:  movl (%rbx), %eax    ← load from virtual address in %rbx

Step 1: Translate VA → PA
  CPU reads the page table entry from RAM:
    PA_of_PTE = PTBR + VPN × sizeof(PTE)
    memory_access_1: RAM[PA_of_PTE] → PTE (reads PFN + flags)

Step 2: Access the actual data
  CPU uses the PFN from the PTE to compute PA:
    PA = PFN × page_size + offset
    memory_access_2: RAM[PA] → value loaded into %eax

Total: 2 RAM accesses for 1 logical load operation.
```

Every load, every store, every instruction fetch — **doubled**. For a program doing millions of memory accesses per second, this is catastrophic.

Cost 2 — Space: The Flat Page Table Is Too Large

Lecture 2.5 calculated page table sizes for a **flat** (single-level) table:

```
32-bit address space, 4 KB pages, 4-byte PTEs:
  Entries = 2^32 / 2^12 = 2^20 = 1,048,576
  Size    = 1,048,576 × 4 = 4 MB per process

  For 100 processes: 400 MB just for page tables.
  Manageable on a 32-bit system, but not ideal.

64-bit address space (48-bit VA used), 4 KB pages, 8-byte PTEs:
  Entries = 2^48 / 2^12 = 2^36 ≈ 68.7 billion
  Size    = 68.7 × 10^9 × 8 = 549 GB per process

  Impossible. A single process's page table would not fit
  in any existing server's RAM.
```

A flat page table is fundamentally unscalable beyond 32-bit address spaces.

Roadmap: Two Solutions

Problem 1 (Time: 2× memory accesses)

Solution: Translation Lookaside Buffer (TLB)

→ A small, fast hardware cache of recent VPN→PFN mappings.

→ On a TLB hit: translation takes ~1 ns (no RAM access needed).

→ Hit rates >99% in practice → effectively eliminates the time cost.

Problem 2 (Space: enormous flat page table)

Solution: Multi-Level Page Table

→ A tree structure: only allocate page table pages for active regions.

→ A sparse process (small code, small stack, small heap) uses only a few KB of page table memory, not 4 MB or 549 GB.

→ Combined with TLB: the extra levels add only rare extra accesses.

§ 2 — The Translation Lookaside Buffer (TLB): Concept

📎 *Extended material → Appendix A.2*

Definition: The TLB as a Hardware Cache for PTEs

> ***The Translation Lookaside Buffer (TLB) is a small, fully associative hardware cache inside the MMU that stores a subset of recently used page table entries.***

Key properties:

- **Part of the MMU:** on-chip, accessed before any RAM lookup.
- **Fully associative:** any TLB entry can hold any VPN→PFN mapping (no conflict misses).
- **Small:** typically 32–1024 entries, but very fast (1–4 ns access time).
- **Transparent:** the program never knows the TLB exists.
- **Managed by hardware or OS** depending on architecture (see § 3).

When the MMU needs to translate a virtual address, it checks the TLB first. If the entry is there (**hit**), no RAM access is needed. If not (**miss**), the page table in RAM must be consulted.

Anatomy of a TLB Entry

Each TLB entry contains the information needed for one page translation:

```
TLB Entry (one row):

┌──────────┬──────────┬──────────┬─────────┬─────────┬─────────┬─────────┬─────┐
│   ASID   │   VPN    │   PFN    │ Valid   │ Dirty   │  Prot   │ Global  │ ... │
│  8 bits  │ 20 bits  │ 20 bits  │ 1 bit   │ 1 bit   │ 3 bits  │ 1 bit   │     │
└──────────┴──────────┴──────────┴─────────┴─────────┴─────────┴─────────┴─────┘

Fields:
  ASID  – Address Space ID: which process owns this entry (avoids flush on ctx switch)
  VPN   – Virtual Page Number: the tag used to look up the entry
  PFN   – Physical Frame Number: the result of a successful lookup
  Valid – 1 if this entry contains a valid translation; 0 if empty/evicted
  Dirty – 1 if the page has been written to (used by page replacement)
  Prot  – Protection bits: Read, Write, Execute permissions
  Global– 1 if shared across all address spaces (kernel entries)
```

TLB Hit: The Fast Path

When the CPU generates a virtual address and the TLB **contains** the corresponding entry:

```
CPU generates VA = 0x00401ABC

MMU extracts:
  VPN    = 0x00401ABC >> 12 = 0x401 = 1025
  Offset = 0x00401ABC & 0xFFF = 0xABC = 2748

  In plain words: "drop the last 3 hex digits (12 bits = the offset), what's left (0x401) is the VPN". Double-check this in decimal:
    0x00401ABC = 4,201,148 (decimal)
    VPN    = 4,201,148 ÷ 4,096 = 1,025  (integer division, remainder dropped)
    Offset = 4,201,148 − (1,025 × 4,096) = 2,748  (the remainder)

TLB lookup (parallel search across all entries):
  Found: entry with VPN=1025, ASID=current process → HIT ✅

From TLB entry:
  PFN   = 200
  Valid = 1
  Prot  = R-X ✅ (instruction fetch is allowed)

PA = 200 × 4096 + 2748 = 819,200 + 2748 = 821,948

→ CPU accesses physical address 821,948 directly.
   Total translation cost: ~1–4 ns (TLB access only, no RAM needed).
```

TLB Miss: The Slow Path

When the TLB **does not contain** the translation for the requested VPN:

```
CPU generates VA → VPN extracted → TLB lookup → MISS ❌

Two possible responses (depending on architecture):

Hardware-managed TLB (x86-64):
  1. CPU hardware automatically walks the page table in RAM.
  2. Finds the PTE for VPN.
  3. Loads the PTE into the TLB (evicting another entry if full).
  4. Re-executes the memory access → now a TLB hit.
  Cost: 1 (or more, for multi-level) extra RAM access(es).

Software-managed TLB (MIPS):
  1. CPU raises a TLB-miss exception.
  2. OS TLB miss handler runs in kernel mode.
  3. Handler walks the page table, finds PTE.
  4. OS inserts the PTE into the TLB using a privileged instruction.
  5. Handler returns; CPU re-executes the faulting instruction.
  Cost: trap overhead + 1 RAM access + return-from-trap.
```

TLB Hit Rate and Effective Memory Access Time (EMAT)

The **hit rate** (h) is the fraction of TLB lookups that result in a hit.

Effective Memory Access Time (EMAT) — analogous to AMAT from cache theory:

$$\text{EMAT} = h \cdot (T_{\text{TLB}} + T_{\text{mem}}) + (1 - h) \cdot (T_{\text{TLB}} + T_{\text{PT}} + T_{\text{mem}})$$

Where:

- T_{TLB} = time to access the TLB (paid on every access, hit or miss) ≈ 1–4 ns
- T_{PT} = time to access the page table in RAM on a miss ≈ 60–100 ns
- T_{mem} = time to access the actual data in RAM ≈ 60–100 ns
- h = TLB hit rate (0.0 to 1.0)

Simplified (assuming $T_{\text{TLB}} \approx 0$ for clarity):

$$\text{EMAT} \approx T_{\text{mem}} + (1 - h) \cdot T_{\text{PT}}$$

Calculating EMAT: Worked Numerical Example

Given:

- T_{TLB} = 1 ns (TLB access time)
- T_{mem} = 100 ns (RAM access time)
- T_{PT} = 100 ns (page table RAM access — same as data)
- Hit rate h = 0.99 (99%)

EMAT calculation:

$$\begin{aligned}\text{EMAT} &= 0.99 \cdot (1 + 100) + 0.01 \cdot (1 + 100 + 100) \\ &= 0.99 \times 101 + 0.01 \times 201 \\ &= 99.99 + 2.01 = \mathbf{102}\ \text{ns}\end{aligned}$$

Without TLB (two RAM accesses every time):

$$\text{cost} = 100 + 100 = \mathbf{200}\ \text{ns}$$

With TLB at 99% hit rate: only **2% slower** than if paging didn't exist (100 ns ideal). With 95% hit rate, EMAT ≈ 106 ns — still only 6% overhead. The TLB is extraordinarily effective.

Why the Hit Rate Is So High: Locality Applied to the TLB

Why is a hit rate of 99%+ achievable with only ~64 TLB entries?

Temporal locality in action:

```
int sum = 0;
for (int i = 0; i < 1000; i++) {
    sum += array[i];    // sum is on the stack, same page every time
}
```

- Variable `sum`: on the stack, accessed 1000 times → same VPN → TLB loaded once, hit 999 times.
- `i`: same stack page → hit every time after the first miss.
- `array[i]`: spatially local — 1000 integers fit in 1-2 pages → only 1-2 TLB miss for 1000 accesses.

Total misses for this loop: 3-4 (2 stack misses + 1-2 array misses).

Total accesses: ~3000+ (instruction fetches + data accesses).

Hit rate: > 99.9%.

✅ Checkpoint 1 — Recap & The TLB Concept (§ 1–2)

Test your understanding:

1. Without a TLB, how many physical memory accesses does a single `load` instruction require under paging? Explain each access.
2. Calculate the size of a flat page table for a 32-bit address space with 4 KB pages and 4-byte PTEs. For 64 running processes, how much RAM do page tables alone consume? Why is the same design infeasible for a 48-bit virtual address space?
3. What is the TLB? Where is it located and what does it cache? List at least five fields of a TLB entry.
4. Given T_{TLB} = 2 ns, T_{mem} = 80 ns, T_{PT} = 80 ns and hit rate = 0.98, calculate the EMAT. How does it compare to the cost without a TLB?

Write down your answers before moving to Section 3.

§ 3 — TLB: Control Flow and Hardware vs. Software Managed

📎 *Extended material → Appendix A.3*

Hardware-Managed TLBs: The CPU Walks the Page Table

On **hardware-managed TLB** architectures (e.g., x86, x86-64, ARM), the CPU itself handles TLB misses:

```
On TLB miss:
  1. Hardware MMU reads the PTBR (e.g., CR3 on x86-64).
  2. Hardware walks the page table in RAM, level by level.
  3. Finds the PTE for the missing VPN.
  4. If PTE.valid = 1:
       Load translation into TLB (evict an old entry if full).
       Re-execute the faulting instruction → now a hit.
  5. If PTE.valid = 0:
       Raise a page fault exception → OS handles it.

The OS is never involved in a TLB miss for a valid page.
The entire miss handling is done by CPU microcode.
```

Advantage: fast (no OS trap overhead for valid pages).

Disadvantage: the hardware must understand the exact page table format the OS uses — tightly coupled.

Software-Managed TLBs: The OS Handles Misses

On **software-managed TLB** architectures (e.g., MIPS R2000/R3000, early SPARC), the OS handles TLB misses via a trap:

```
On TLB miss:
  1. Hardware raises a TLB-miss exception (a trap to the OS).
  2. CPU switches to kernel mode, jumps to the OS TLB miss handler.
  3. OS handler walks the page table in software.
  4. If PTE.valid = 1:
       OS inserts the translation into the TLB using a privileged
       "load TLB entry" instruction (e.g., TLBWI on MIPS).
       OS executes return-from-exception → CPU re-executes instruction.
  5. If PTE.valid = 0:
       OS raises a page fault; handler loads page from disk or terminates.
```

Advantage: OS can use any page table format it likes — full flexibility.

Disadvantage: every TLB miss incurs a full trap/return-from-trap overhead (~100+ cycles).

Comparing Hardware vs. Software Managed TLBs

Property	Hardware-Managed (x86-64)	Software-Managed (MIPS)
Miss handler	CPU microcode	OS kernel code
Miss latency	Low (~10–50 cycles)	Higher (~100–200 cycles)
Page table format	Fixed by hardware spec	Flexible — OS chooses
OS involvement	Only on page fault	On every TLB miss
Portability	Tied to one page table format	OS can change page table freely
Typical use	x86, x86-64, ARM (v7+)	MIPS, early SPARC, some RISC
Complexity	In hardware (transistors)	In software (easier to update)

Modern high-performance architectures (x86-64, ARMv8) use hardware-managed TLBs. Research and RISC-V systems often support both modes.

The ASID: Avoiding Full TLB Flushes on Context Switch

The **Address Space Identifier (ASID)** is a tag that identifies which process a TLB entry belongs to:

Without ASID:

- On context switch: flush entire TLB (all entries → invalid).
- After switch to Process B: TLB is cold → many misses.
- **Cost:** several hundred ns of extra memory latency per switch.

With ASID:

- Each TLB entry tagged with the ASID of its process.
- On context switch: OS loads new process's ASID into a CPU register.
- TLB lookup: only match entries where `entry.ASID == current_ASID`.
- Old process's entries are STILL in the TLB — just ignored. When old process runs again: its entries are still warm!

TLB can simultaneously hold entries for Process A (ASID=3), Process B (ASID=7), Process C (ASID=12), etc.

§ 4 — TLB Performance: Coverage, Thrashing, and Huge Pages

📎 *Extended material → Appendix A.4*

TLB Coverage: How Much Address Space Is Mapped

TLB coverage is the total amount of virtual address space that can be mapped simultaneously by all TLB entries:

$$\text{TLB coverage} = \text{TLB entries} \times \text{page size}$$

```
Example: 64-entry TLB, 4 KB pages:
  Coverage = 64 × 4,096 = 262,144 bytes = 256 KB
```

This means: if a program's working set fits in 256 KB, it will (roughly) experience zero TLB misses after warm-up.

Most small programs: easily fit in 256 KB working set → excellent TLB performance.

Large databases or in-memory analytics with GB working sets → Frequent TLB misses → performance problem.

TLB Thrashing: When the Working Set Exceeds TLB Coverage

TLB thrashing occurs when a program accesses more unique pages than the TLB can hold simultaneously:

```
TLB has 64 entries (4 KB pages → 256 KB coverage).

Program accesses 100 different pages (400 KB working set):

  Access page 0:   TLB miss → load entry
  Access page 1:   TLB miss → load entry
  ...
  Access page 63:  TLB miss → load entry (TLB now full)
  Access page 64:  TLB miss → load entry (evict page 0's entry)
  Access page 65:  TLB miss → load entry (evict page 1's entry)
  ...
  Access page 0 again: TLB miss (evicted earlier!) → reload
```

If the access pattern cycles through all 100 pages repeatedly, EVERY access is a TLB miss → hit rate ≈ 0% → 2× slowdown. **This is TLB thrashing.**

Increasing Coverage: Huge Pages / Superpages

The most effective way to increase TLB coverage without adding entries: **use larger pages**:

$$\text{TLB coverage} = \text{TLB entries} \times \text{huge page size}$$

```
x86-64 page size options:
  Standard: 4 KB    → 64-entry TLB covers 256 KB
  Large:    2 MB    → 64-entry TLB covers 128 MB  (512× improvement!)
  Huge:     1 GB    → 64-entry TLB covers 64 GB   (262,144× improvement!)
```

Linux

Explicit huge pages (`hugetlbfs`) and Transparent Huge Pages (THP, automatic) allow the OS to back contiguous memory regions with 2 MB or 1 GB pages.

- **Benefit:** database applications, scientific computing, JVM heaps see 10–30% speedups from THP due to reduced TLB pressure.
- **Cost:** internal fragmentation up to 2 MB per huge page allocation.

✅ Checkpoint 2 — TLB Control Flow, ASIDs & Performance (§ 3–4)

Test your understanding:

1. Describe the sequence of events on a TLB miss for a **hardware-managed** TLB (x86-64) and for a **software-managed** TLB (MIPS). When does the OS get involved in each case?
2. What is an ASID? How does it let entries from different processes coexist in the TLB, and what does it save on a context switch?
3. Calculate the TLB coverage for (a) 64 entries × 4 KB pages and (b) 64 entries × 2 MB huge pages. What is the ratio?
4. Define TLB thrashing. Under what condition does it occur?

Make sure you can reproduce the coverage calculation before moving on.

§ 5 — Multi-Level Page Tables: Motivation and Core Idea

📎 *Extended material → Appendix A.5*

The Space Problem: Flat Page Tables Allocate for ALL Pages

The flat (single-level) page table's fundamental flaw:

```
32-bit process, 4 KB pages → 2^20 = 1,048,576 PTE entries
Each PTE = 4 bytes → Page table = 4 MB

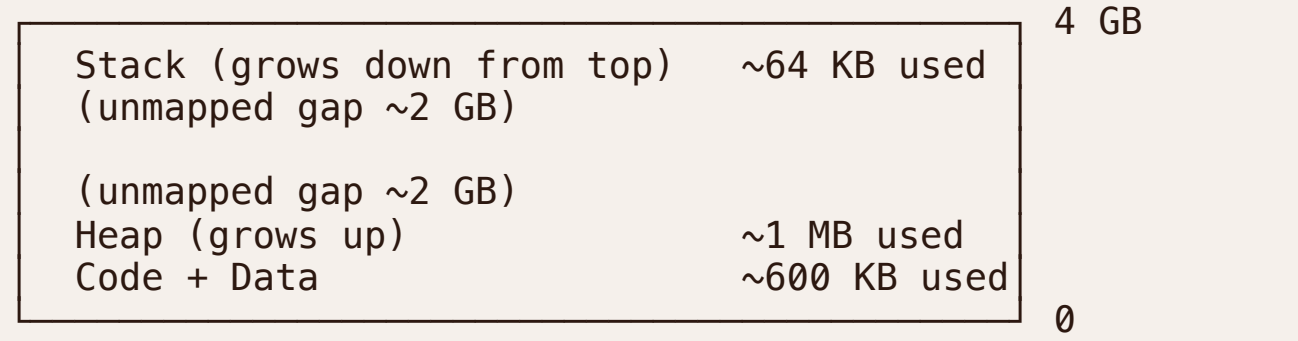
BUT: how many pages does a typical process actually use?

  Code:       ~500 KB → ~125 pages
  Data:       ~100 KB → ~25 pages
  Heap:       ~1 MB   → ~256 pages
  Stack:      ~64 KB  → ~16 pages
  TOTAL:      ~422 pages actively used

The flat page table allocates 1,048,576 entries for 422 used pages.
99.96% of the page table entries are INVALID (valid bit = 0).
99.96% of 4 MB = ~4 MB wasted on invalid PTEs.
```

Key Observation: Most PTEs Are Invalid

The vast majority of a process's virtual address space is **empty** — never mapped to any physical frame:



Only ~1.7 MB / 4 GB = 0.04% of the address space is used.
Flat page table stores 1M entries; only ~422 have valid=1.

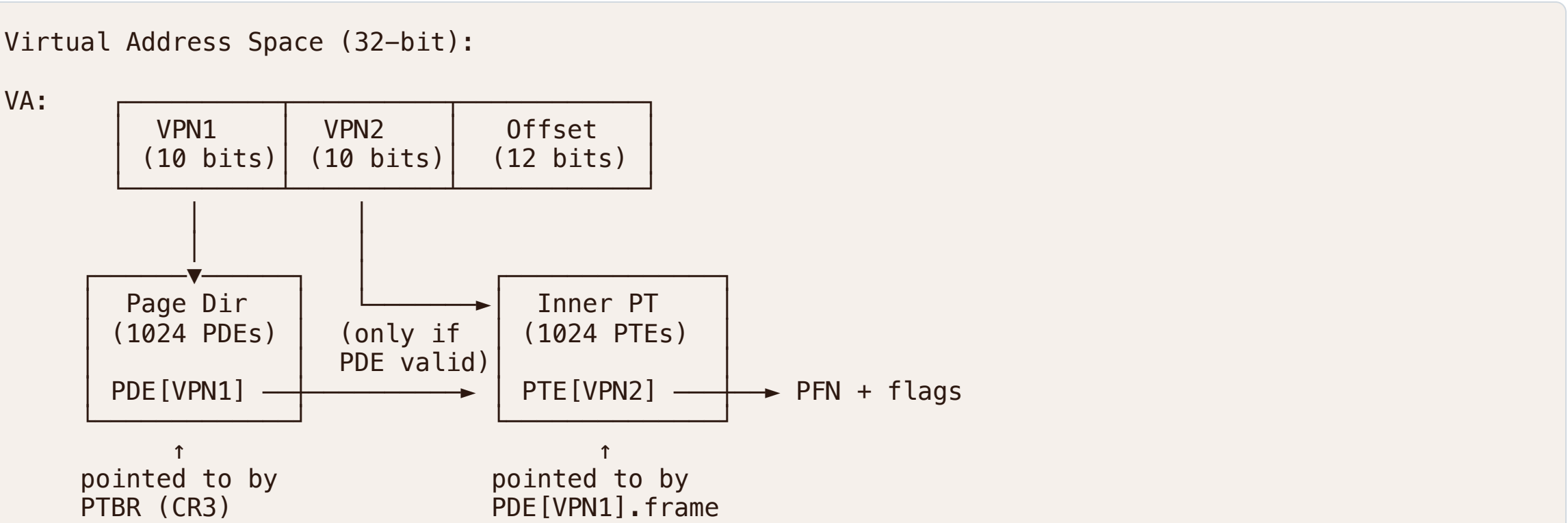
Page Directory Entries (PDEs) vs. Page Table Entries (PTEs)

It is important to distinguish the two levels:

	PDE (Page Directory Entry)	PTE (Page Table Entry)
Level	First (root)	Second (leaf)
Points to	Physical frame containing an inner page table	Physical frame containing actual program data
Valid=0 means	Entire virtual region is unmapped → no inner table	This one virtual page is unmapped
Valid=1 means	Inner page table exists at the given frame	Data page exists at the given frame
Size	Same as PTE (4 or 8 bytes)	4 or 8 bytes
Count	1024 (for 32-bit, 4 KB pages)	1024 per inner table

The PDE's `valid` bit is a tree-node existence bit. The PTE's `valid` bit is a leaf-node existence bit.

Two-Level Page Table Structure Diagram



Space Savings: Two-Level vs. Flat (Worked Example)

Virtual address space: 4 GB (32-bit). **Page size:** 4 KB. PTE/PDE: 4 bytes.

Process uses: code (0–2 MB), heap (4–6 MB), stack (4094–4096 MB).

```
Flat page table:
  Entries = 4 GB / 4 KB = 1,048,576
  Size = 1,048,576 × 4 = 4,194,304 bytes = 4 MB

Two-level page table:
  Page directory: 1 × 1024 × 4 = 4,096 bytes = 4 KB
  Inner page tables needed:
    Code/heap: occupies 2 PD regions → 2 inner page tables
    Stack:     occupies 1 PD region  → 1 inner page table
  3 inner tables × 1024 × 4 = 12,288 bytes = 12 KB

Total two-level: 4 KB + 12 KB = 16 KB

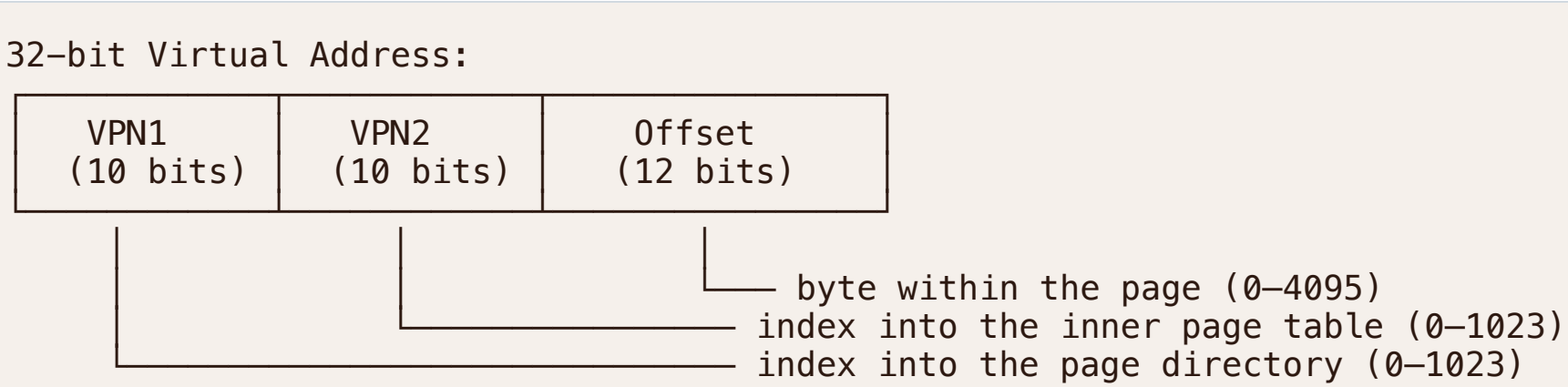
Savings: 4 MB → 16 KB (256× smaller!)
```

§ 6 — Two-Level Page Tables: Mechanics

📎 *Extended material → Appendix A.6*

Virtual Address Structure with Two Levels

For a 32-bit address space with 4 KB pages and a two-level page table, the VA is split into three fields:



Why 10+10+12?
- offset = $\log_2(4096)$ = 12 bits
- remaining = 32 − 12 = 20 bits split evenly: 10 + 10
- Each 10-bit index → 2^10 = 1024 entries per level
- 1024 entries × 4 bytes/entry = 4 KB per table page
  - → each page table page is exactly one physical frame in size ✅

The Two-Level Translation Algorithm: Step by Step

```
Input: VA (32 bits), PTBR (physical address of page directory)

Step 1: Extract fields
  VPN1   = VA[31:22]   (top 10 bits)
  VPN2   = VA[21:12]   (middle 10 bits)
  Offset = VA[11:0]    (bottom 12 bits)

  In shift/mask form (same thing, just how the CPU computes it):
    Offset = VA & 0xFFF                  (keep the last 12 bits)
    VPN2   = (VA >> 12) & 0x3FF          (drop 12, then keep the next 10)
    VPN1   = VA >> 22                    (drop 22 — nothing left to mask,
                                            since VPN1 is the top field)
  In plain words: peel the offset off the bottom, then peel VPN2 off
  what remains, and whatever is left at the top is VPN1.

Step 2: Level-1 lookup (page directory)
  PDE_addr = PTBR + VPN1 × sizeof(PDE)
  PDE = RAM[PDE_addr]                    ← Memory access #1
  if PDE.valid == 0: raise PageFault

Step 3: Level-2 lookup (inner page table)
  PT_base  = PDE.frame × 4096
  PTE_addr = PT_base + VPN2 × sizeof(PTE)
  PTE = RAM[PTE_addr]                    ← Memory access #2
  if PTE.valid == 0: raise PageFault
  if access_type violates PTE.prot: raise ProtectionFault

Step 4: Compute physical address
  PA = PTE.frame × 4096 + Offset         ← Memory access #3 (actual data)
```

Final PA Computation: PFN + Offset

Step 4 — the physical address is assembled from the PFN and the original offset:

```
PFN    = PTE.frame = 150
Offset = VA[11:0]  = 0xABC = 2748  (from original VA decomposition)

PA = PFN × page_size + Offset
   = 150 × 4096 + 2748
   = 614,400 + 2748
   = 617,148
   = 0x96ABC

The CPU accesses physical address 0x96ABC.
The data bytes at that address are read/written.       ← Memory access #3
```

Total memory accesses without TLB: 3 (PDE + PTE + data).

With TLB hit: 1 (data only) — the 2-level overhead disappears.

Time Cost: Three Memory Accesses (Without TLB)

The price of multi-level indirection:

```
Without TLB:
  Two-level page table: 3 memory accesses per data access.
  (PDE + PTE + data)
  vs. flat page table: 2 memory accesses.
  vs. base/bounds: 1 memory access.

Three-level page table: 4 memory accesses per data access.
Four-level page table (x86-64): 5 memory accesses per data access!

This would be catastrophic WITHOUT the TLB.

With TLB (hit rate 99%):
  99% of accesses: 1 memory access (TLB hit → skip all page table levels).
  1% of accesses:  5 memory accesses (TLB miss → walk all 4 levels + data).
  EMAT ≈ 1.04 × T_mem — barely 4% overhead.

The TLB makes multi-level page tables practical.
```

✅ Checkpoint 3 — Multi-Level Page Tables (§ 5–6)

Test your understanding:

Setup: 32-bit VA, 4 KB pages, 2-level PT (10-10-12). PTBR = 0x00005000.
Page directory at frame 5: PD[0] → frame 20 (valid), PD[2] → frame 30 (valid), all other PDEs invalid.
Inner PT at frame 20: PTE[0] → PFN 100 (RW-), PTE[1] → PFN 101 (R-X), others invalid.
Inner PT at frame 30: PTE[5] → PFN 200 (RW-), others invalid.

1. Why does a flat page table waste so much memory for a typical 32-bit process, and what does a PDE with `valid = 0` save?
2. A process uses 2 MB of code, 2 MB of heap and 2 MB of stack, in three different 4 MB regions of a 4 GB address space. Compute the flat and the two-level page-table size.
3. Translate VA = `0x00000A10`, then VA = `0x00400000`. Show VPN1, VPN2, offset, PDE, PTE and PA, or where the walk fails.
4. How many physical memory accesses does the translation of `0x00000A10` need (a) without a TLB, (b) with a TLB hit?

> ***Hint:*** *convert to decimal: Offset = VA mod 4,096; VPN2 = (VA ÷ 4,096) mod 1,024; VPN1 = VA ÷ 4,194,304.*

§ 7 — Multi-Level Page Tables Beyond Two Levels

📎 *Extended material → Appendix A.7*

Why Two Levels Are Not Enough for 64-Bit

Two-level page tables work for 32-bit address spaces. For 64-bit, the math breaks down:

```
48-bit VA (x86-64 standard), 4 KB pages, two-level attempt:

  offset  = 12 bits
  remaining = 48 - 12 = 36 bits to split between two levels.

  If split 18+18: each level has 2^18 = 262,144 entries.
  Page directory: 262,144 × 8 bytes = 2 MB per process. Still large.
  Each inner PT: 262,144 × 8 bytes = 2 MB.
  For a sparse process: 2 MB (dir) + a few MBs (inner) = still multi-MB.

  With 4-level (9-9-9-9-12): each table is exactly one 4 KB page.
  4 levels × 9 bits: 512 entries per table × 8 bytes = 4 KB per table ✅
  Sparse process: 4 KB (PML4) + 4 KB (PDPT) + 4 KB (PD) + 4 KB (PT)
                = 16 KB for the page table structure.
```

Four-Level Page Tables: The x86-64 Standard

Intel x86-64 uses a **four-level page table** for 48-bit virtual addresses:

```
48-bit Virtual Address:

┌──────────┬──────────┬──────────┬──────────┬──────────┐
│  PML4    │  PDPT    │   PD     │   PT     │  Offset  │
│ (9 bits) │ (9 bits) │ (9 bits) │ (9 bits) │ (12 bits)│
└──────────┴──────────┴──────────┴──────────┴──────────┘
  indices:    0–511      0–511      0–511      0–511

Level names:
  PML4  = Page Map Level 4 (root, pointed to by CR3)
  PDPT  = Page Directory Pointer Table
  PD    = Page Directory
  PT    = Page Table (leaf — contains PFNs)

Each level: 512 entries × 8 bytes = 4 KB = exactly one physical frame.
Translation accesses: 4 (PML4 + PDPT + PD + PT) + 1 (data) = 5 total (no TLB).
```

x86-64 Page Table Walk Diagram: CR3 → PML4 → PDPT → PD → PT → Data

```
CR3 ──► [ PML4 table ]
        PML4[VPN1] ──► [ PDPT table ]
                       PDPT[VPN2] ──► [ PD table ]
                                      PD[VPN3] ──► [ PT table ]
                                                   PT[VPN4] ──► [ Physical Page ]
                                                                 + Offset = PA

Each arrow = one memory access (if TLB misses all 4 levels).
CR3 write (context switch) = flush TLB (or update ASID).

Memory accesses per translation (no TLB): 4 + 1 = 5
Memory accesses per translation (TLB hit): 1

This is why the TLB is not optional for x86-64 — it is existential.
Without TLB, a 4-level walk would slow every memory access by 5×.
```

§ 8 — OS Responsibilities and Advanced Topics

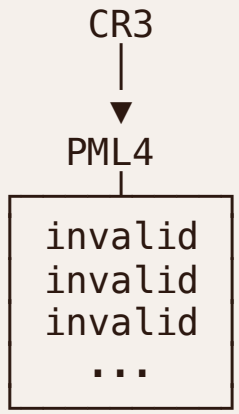
📎 *Extended material → Appendix A.8*

OS Responsibilities: Page Tables Are Allocated On Demand (1/2)

The OS does not normally allocate the entire multi-level page-table hierarchy when a process is created.

(1/2) At process creation:

1. Allocate the root page-table page (PML4 on x86-64).
2. Initialize it with invalid entries.
3. Set CR3 to point to the root page.



OS Responsibilities: Page Tables Are Allocated On Demand (2/2)

(2/2) When a page is first needed:

1. The CPU performs the page-table walk.
2. At some level, it encounters a non-present entry.
3. A page fault (#PF) occurs.
4. The OS's page-fault handler determines that the access is valid and needs a new mapping.
5. The OS allocates any required intermediate page-table pages.
6. Each newly allocated page-table page is initialized and linked into its parent.
7. The OS allocates a physical page for the actual data, if necessary, and installs the final PTE.
8. The faulting instruction is resumed.
9. The CPU retries the translation, which can now succeed.

```
PML4 ⟶ PDPT ⟶ PD ⟶ PT ⟶ Physical Page
```

Page table structure grows **on demand** — exactly when needed.

Superpages / Huge Pages: 2 MB and 1 GB Pages in x86-64

x86-64 supports three page sizes simultaneously:

```
4 KB pages (standard):
  PT leaf entry → points to a 4 KB data frame.
  Most common; finest granularity.

2 MB large pages:
  PD leaf entry (bit 7 "PS"=1) → points to a 2 MB data frame.
  Skips the PT level entirely.
  TLB coverage per entry: 2 MB (512× a 4 KB entry).
  PD entry must be aligned to a 2 MB boundary.

1 GB huge pages:
  PDPT leaf entry (bit 7 "PS"=1) → points to a 1 GB data frame.
  Skips both PD and PT levels.
  TLB coverage per entry: 1 GB.
  Requires hardware support (Intel Nehalem+ / AMD Barcelona+).
```

Summary of OS Page Table Management Responsibilities

Event	OS Action
Process creation	Allocate root page table (PML4); set CR3 in PCB
Page fault (valid page)	Allocate frame; fill PTE; retry instruction
Page fault (new PT page)	Allocate frame for inner PT; set parent PDE; retry
Context switch	Save CR3 to PCB; load new process CR3; (TLB flush or ASID update)
`mmap()` / `brk()`	Allocate PTE region; may pre-allocate inner PT pages
`munmap()` / process exit	Invalidate PTEs; free frames; free PT pages; update free list
Page eviction (swapping)	Set PTE.valid=0; record disk location in PTE; TLB shootdown
Huge page promotion	Collapse 512 contiguous PTEs into one 2 MB PD entry
TLB shootdown	IPI to all cores; each core invalidates affected TLB entries

✅ Checkpoint 4 — Beyond Two Levels & OS Responsibilities (§ 7–8)

Test your understanding:

1. An x86-64 process accesses VA = `0x0000_7FFF_FFFF_1ABC`. Extract the PML4, PDPT, PD and PT indices and the page offset.
2. Without a TLB, how many physical memory accesses does a four-level x86-64 walk require? With a TLB hit?
3. Why does x86-64 use 9-bit indices at each level (instead of 10-bit as in the 32-bit two-level scheme)? What is the size of each page-table page?
4. A process touches a virtual address whose inner page table has never been allocated. Describe, step by step, what the CPU and the OS do.

Check your index extraction by rebuilding the address from the indices.

§ 9 — Worked Examples & Summary

📎 *Extended material → Appendix A.9*

Capstone Worked Example: Two-Level Page Table Translation (Full Trace)

Setup: 32-bit VA, 4 KB pages, 10-10-12 split.
PTBR = 0x00002000 (page directory at physical address 0x2000).

Physical memory content:

- Frame 2 (0x2000): PD — PDE[0]=frame 10 valid, PDE[1]=frame 11 valid, all others invalid.
- Frame 10 (0xA000): PT — PTE[3]=PFN 50 valid RW-, PTE[5]=PFN 60 valid R-X, others invalid.
- Frame 11 (0xB000): PT — PTE[0]=PFN 70 valid RW-, others invalid.

Translate VA = `0x00003B50`:

```
VPN1   = 0x00003B50 >> 22 = 0
VPN2   = (0x00003B50 >> 12) & 0x3FF = 3
Offset = 0x00003B50 & 0xFFF = 0xB50 = 2896

Step 1: PDE[0] = frame 10, valid=1 → PT at 0xA000
Step 2: PTE[3] = PFN 50, valid=1, prot=RW- ✅
Step 3: PA = 50 × 4096 + 2896 = 204,800 + 2896 = 207,696 = 0x32B50
```

✅ Checkpoint 5 — Putting It All Together (§ 9)

Test your understanding:

1. Using the setup of the capstone example, translate VA = `0x00005010` for (a) a read and (b) a write. What happens in each case?
2. A four-level system has T_{TLB} = 1 ns, T_{mem} = 100 ns, a TLB hit rate of 0.99, and every table access costs T_{mem}. Compute the EMAT, counting a full four-level walk on every miss.
3. In one sentence each: which problem does the TLB solve, which problem do multi-level page tables solve, and why do we need both?

If you can do these three, you are ready for Lecture 2.7.

The Combined Solution: TLB + Multi-Level Makes Paging Practical

```
Problem 1 (Time: 2× accesses)
  → Solved by TLB: 99%+ hit rate → ~1 access on average.

Problem 2 (Space: enormous flat page table)
  → Solved by multi-level page tables: only active regions consume RAM.

Together:
  Multi-level page table: provides compact space representation.
  TLB: hides the extra translation levels (5 accesses → 1 effective).

Result:
  A process with a sparse 48-bit VA space uses ~16–64 KB of page table memory
  and experiences near-zero translation overhead at runtime.
  This is how modern 64-bit operating systems provide
  efficient, protected, large address spaces to every process.
```

Summary: Chapter 19 — The TLB

Core mechanism:

- Small hardware cache (32–1024 entries) of recent VPN→PFN translations.
- TLB hit: translation in ~1–4 ns, no RAM access needed.
- TLB miss: consult page table in RAM (hardware or OS walk); load entry; retry.
- ASID: tags entries with process ID, allows coexistence of multiple processes' translations without full flush on context switch.
- Global bit: kernel entries survive context switches.
- Hit rates >99% in typical workloads due to temporal and spatial locality.

$$\text{EMAT} = h \cdot (T_{\text{TLB}} + T_{\text{mem}}) + (1 - h) \cdot (T_{\text{TLB}} + T_{\text{PT}} + T_{\text{mem}})$$

Summary: Chapter 20 — Multi-Level Page Tables

Core mechanism:

- Tree structure: root table (page directory) → inner tables → leaf PTEs.
- PDE valid=0 → entire virtual sub-region unmapped → no inner table allocated.
- Only page table pages for **active virtual regions** are allocated.
- Sparse process: page table size proportional to mapped pages, not VA space.

x86-64 four-level (PML4 → PDPT → PD → PT):

- 48-bit VA, 9-9-9-9-12 split, each table = one 4 KB frame.
- Without TLB: 5 RAM accesses per translation. With TLB: 1.

RISC-V Sv39: three-level, 9-9-9-12 (→ Appendix A.7).
Five-level (PML5): 57-bit VA for >128 TB processes (→ Appendix A.7).

Recommended Exercises

Before Lecture 12, practice:

1. **EMAT:** T_{TLB}=1 ns, T_{mem}=120 ns, T_{PT}=120 ns. Calculate EMAT for hit rates of 0.90, 0.95, 0.99, 1.00.
2. **Two-level translation:** 32-bit VA, 4 KB pages, 10-10-12 split. PTBR=0x3000. Walk VA=`0x00403010`. Assume PD at frame 3, PD[1]=frame 7 valid, PT at frame 7, PT[3]=PFN 200 valid RW-. Give PA.
3. **x86-64 VA decomposition:** Decompose VA=`0x0000_2001_0030_00A4` into PML4, PDPT, PD, PT indices and page offset.
4. **Space savings:** A 32-bit process uses only 3 virtual pages. Compare flat vs. two-level page table sizes.
5. **TLB coverage:** A program's working set is 1 MB. TLB has 64 entries. (a) With 4 KB pages: is there TLB thrashing? (b) With 2 MB huge pages?

Lecture 2.6 Complete

Recommended Reading

- **OSTEP Ch. 19:** *Paging: Faster Translations (TLBs)*
- **OSTEP Ch. 20:** *Paging: Smaller Tables*
- Available free at: https://ostep.org

Contact & Q&A

- **Lecturer:** Gabriele Tolomei
- **Email:** tolomei@di.uniroma1.it

- **Office Hours:** Drop me a message, and we'll arrange a time to meet, either in person or online! 🙂

Thank you for your attention!

§ Appendix

Appendix Map

Section	Extends	What you will find	Slides
A.1	§ 1	The time cost in numbers; locality refresher	2
A.2	§ 2	Hex/shift/mask toolkit; temporal and spatial locality applied to the TLB	5
A.3	§ 3	Control-flow diagrams, TLB entry fields, global bit, flushes, ASID details, shootdowns	7
A.4	§ 4	TLB size, array vs. linked-list traversals, real hit rates, OS policies	5
A.5	§ 5	The crux, the sparse-matrix analogy, the page directory, space comparisons	5
A.6	§ 6	Toy warm-up and the translation, level by level	4
A.7	§ 7	48-bit x86-64 addresses, five-level tables, RISC-V Sv39, space and time analysis, paged page tables	7
A.8	§ 8	Inverted and hashed page tables, Transparent Huge Pages, page-table security	5
A.9	§ 9	Problems 1–5, strategy, common mistakes, key formulas, self-check bank	14

Slides are numbered `A.section.slide`*. Checkpoints, prerequisites and summaries live in the main part.*

📎 A.1 — Recap & The Two Costs of Paging

A.1.1 · Quantifying the Time Cost

Let's put numbers to the overhead:

```
Modern DRAM latency ≈ 60–100 ns.

Without paging (base/bounds):
  1 memory access per load/store = 60 ns.

With flat paging (no TLB):
  2 memory accesses per load/store = 120 ns.
  Effective slowdown: 2×.

A CPU running at 3 GHz executes ~3 billion instructions/second.
Many instructions access memory. A 2× memory latency penalty
translates directly into a 2× or worse overall program slowdown.

Real-world impact: a program that takes 1 second to run
would take 2+ seconds with naive paging. Completely unacceptable.
```

A.1.2 · The Principle of Locality: Why a Cache Can Help

Before solving these problems, recall why locality makes caches so effective (Lecture 1.1):

Temporal locality: if a memory location is accessed, it is likely to be accessed again soon.

- Implication for TLB: if we translate VPN → PFN once and cache it, the same VPN will likely be looked up again soon.

Spatial locality: if a memory location is accessed, nearby locations are likely accessed soon.

- Implication for TLB: sequential code execution and array traversal mean consecutive addresses (and thus the same page) are accessed many times before moving to the next page.

Both forms of locality mean that a **small cache of recent translations** (the TLB) will eliminate the vast majority of page table lookups.

📎 A.2 — The TLB: Concept

A.2.1 · Toolkit: Reading Hex Addresses

Throughout this lecture we write addresses in **hexadecimal (base 16)** because binary is too long and decimal hides the byte structure. You don't need to be fast at hex — you only need one skill: converting a hex digit to 4 binary bits.

```
Hex digit:   0    1    2    3  ...  9    A    B    C    D    E    F
Decimal:     0    1    2    3  ...  9   10   11   12   13   14   15
Binary:   0000 0001 0010 0011 ... 1001 1010 1011 1100 1101 1110 1111
```

Key fact: each hex digit = exactly 4 bits. So a 32-bit address is always **8 hex digits**, and a 12-bit page offset is always **3 hex digits**.

```
Example:  0x1A3
  1 → 0001,  A → 1010,  3 → 0011
  0x1A3 = 0001 1010 0011 (binary) = 419 (decimal)
```

Whenever a slide uses hex, you can always fall back to this digit-by-digit table.

A.2.2 · Toolkit: What Does `>>` (Shift) Actually Do?

A **right shift** `x >> n` simply means: "*throw away the last n bits of x.*" In decimal terms, this is the same as **integer division by** 2^n (drop the remainder).

```
Example in decimal (easier to see the idea):
  1234 >> "divide by 10, drop remainder" = 123   (this is how >> works, but for base 10)

Example in binary, which is what the CPU actually does:
  x = 0b10110101   (= 181 in decimal)
  x >> 4  means "drop the last 4 bits"
        0b1011 0101
               └────┘ ← these 4 bits are thrown away
        result = 0b1011 = 11
```

Why the OS uses this: a virtual address is `VPN | Offset`. If the offset is 12 bits wide, then `VA >> 12` throws away exactly the offset and leaves the VPN sitting in the low bits — no matter how big VA is. This is just "peel off the last 12 bits," nothing more exotic.

A.2.3 · Toolkit: What Does `&` (Mask) Actually Do?

The **AND mask** `x & mask` means: "*keep only the bits where mask has a 1; zero out everything else.*" It is used to **isolate a middle group of bits** after a shift has already removed the bits below them.

```
x    = 0b1 1010 0110 1011     (some address, already shifted)
mask = 0b0 0000 0011 1111     (0x3F — keep only the lowest 6 bits)
       -------------------
x&m  = 0b0 0000 0000 1011     (everything above bit 6 is zeroed)
```

Common masks you'll see in this lecture:

Mask (hex)	Binary	Keeps this many low bits	Use
`0xF`	`1111`	4 bits	one hex digit
`0x3FF`	`11 1111 1111`	10 bits	one 32-bit page-table index (1024 entries)
`0x1FF`	`1 1111 1111`	9 bits	one x86-64 page-table index (512 entries)
`0xFFF`	`1111 1111 1111`	12 bits	a 4 KB page offset

Putting shift and mask together: `(VA >> 12) & 0x3FF` reads as: "*drop the last 12 bits (the offset), then keep only the next 10 bits (this level's index).*" Every VPN-extraction formula in this lecture is just this recipe, repeated with different shift amounts.

A.2.4 · Temporal Locality and the TLB

Temporal locality: the same page is accessed repeatedly over a short time window.

```
Program counter advances through a function:
  instructions at VA 0x401000–0x4010FF → all on page VPN=0x401
  → TLB loaded once for VPN=0x401 → all subsequent fetches: HIT

Stack is heavily reused:
  local variables and function arguments → few stack pages
  → same VPN accessed thousands of times per function call
  → TLB loaded once per function call → all subsequent accesses: HIT
```

The TLB exploits temporal locality: by caching the most recently used translations, it ensures that the most-accessed pages are translated without any RAM lookup.

A.2.5 · Spatial Locality and the TLB

Spatial locality: consecutive addresses within the same page are accessed together.

```
Array traversal:
  int arr[1024];   // 1024 × 4 = 4096 bytes = exactly 1 page

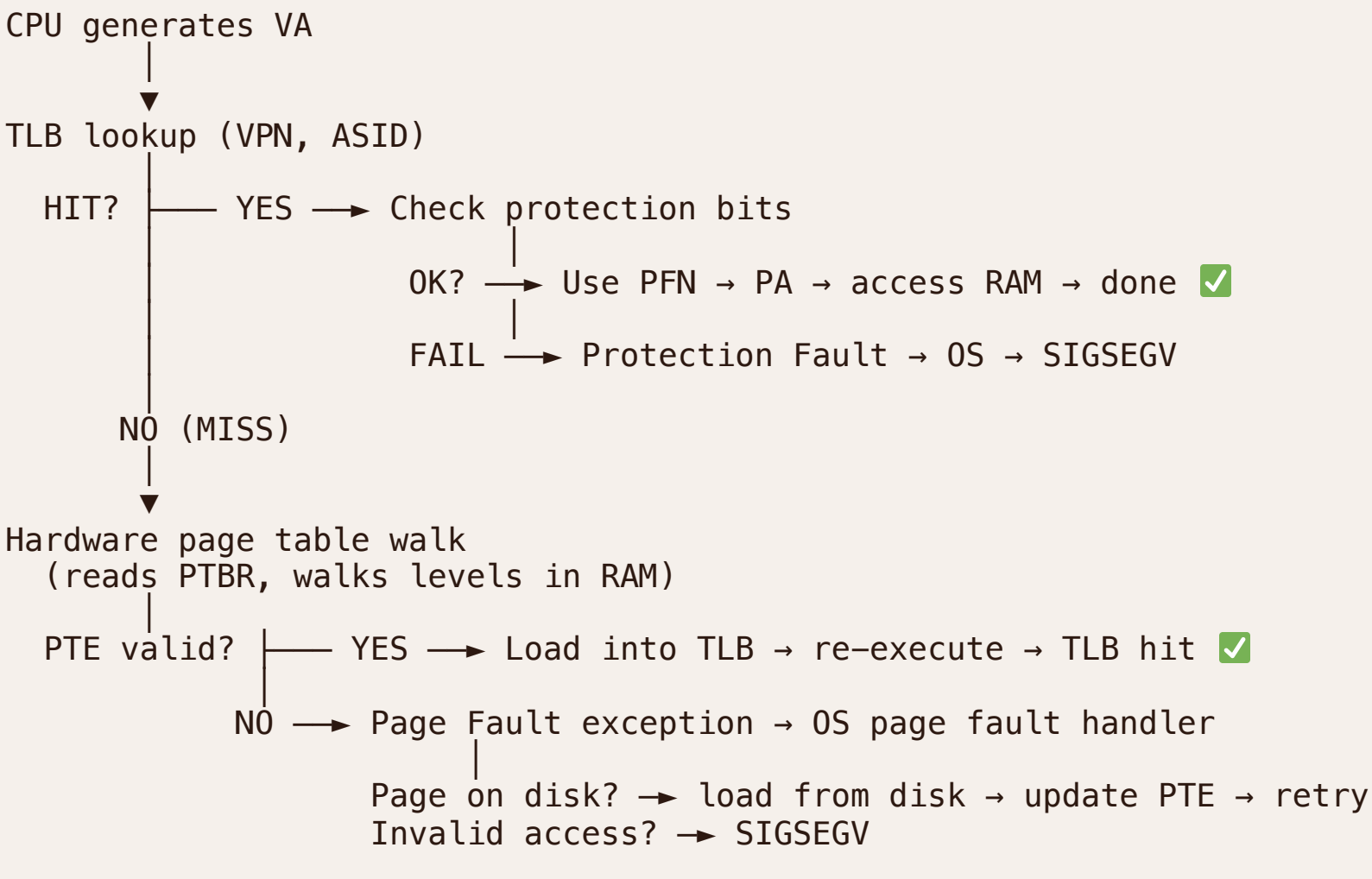
  for (int i = 0; i < 1024; i++) {
      arr[i] = 0;
  }

  All 1024 accesses → same physical page → same TLB entry.
  Cost: 1 TLB miss (first access) + 1023 TLB hits.
  Hit rate: 99.9%.

For larger arrays (multiple pages):
  arr[0]–arr[1023]: page 1 → 1 TLB miss
  arr[1024]–arr[2047]: page 2 → 1 TLB miss
  arr[2048]–arr[3071]: page 3 → 1 TLB miss
  Total misses = number of pages touched (not number of accesses).
```

📎 A.3 — TLB: Control Flow and Hardware vs. Software Managed

A.3.1 · Hardware-Managed TLB: Control Flow Diagram



A.3.2 · Software-Managed TLB: Control Flow Diagram



A.3.3 · TLB Entry Fields: A Complete Inventory

Beyond VPN and PFN, a TLB entry carries several important flags:

Field	Bits	Meaning
VPN	20–52	Virtual Page Number (the lookup key / tag)
PFN	20–40	Physical Frame Number (the result)
Valid	1	1 = entry is live; 0 = empty or invalidated
Dirty	1	1 = page has been written (must write back before evict)
Prot	3	Read / Write / Execute permission bits
User	1	0 = kernel-only; 1 = accessible from user mode
Global	1	1 = shared across all ASIDs (kernel entries)
ASID	8–16	Address Space ID — identifies which process owns this entry

Total size per entry: typically 64–128 bits. With 64 entries: 512–1024 bytes total TLB storage.

A.3.4 · The Global (G) Bit: Kernel Pages

The **global bit** marks TLB entries that are valid for **all processes**:

Without global bit:

- Every context switch must flush ALL TLB entries (expensive!).
- Even entries for kernel code (same for every process) are flushed.

With global bit:

- Kernel page table entries have G=1 (these entries survive context switches.)
- Only per-process (G=0) entries are invalidated on a context switch.

Example:

```
kernel code at VA 0xFFFF8000_00000000
  TLB entry: VPN=0xFFFF8..., PFN=kernel_frame, G=1
  → Remains in TLB across all process switches.
  → Kernel functions called via syscall never suffer TLB misses
    for kernel code (after the first load).
```

A.3.5 · TLB Flush on Context Switch: The Naive Approach and Its Cost

Without ASID, every context switch must flush (invalidate) the entire TLB:

```
Context switch cost without ASID:

1. OS saves Process A's state → PCB_A.
2. OS loads Process B's state ← PCB_B.
3. OS writes Process B's PTBR to CR3.
4. Writing CR3 automatically flushes the entire TLB (x86 behavior).
5. Process B starts with a cold TLB.
6. First few hundred accesses: TLB misses → extra RAM lookups.

Cost of TLB flush:
  - Direct: ~100 cycles to invalidate all entries.
  - Indirect: TLB cold start → dozens to hundreds of extra RAM accesses.
  - Total: effectively adds 500–2000 ns of extra latency per context switch.

On a system doing 1000 context switches/second:
  1000 × 2000 ns = 2 ms/second wasted on TLB warm-up — measurable!
```

A.3.6 · ASID-Tagged TLB: Coexistence of Multiple Processes

With ASID support, TLB entries from different processes coexist:

```
TLB state (64 entries) after running processes A(ASID=3), B(ASID=7):

| ASID | VPN     | PFN | Valid | Prot |
|------|---------|-----|-------|------|
|  3   | 0x00401 | 100 |   1   | R-X  |  ← Process A's code
|  3   | 0x00001 | 200 |   1   | RW-  |  ← Process A's heap
|  7   | 0x00401 | 350 |   1   | R-X  |  ← Process B's code
|  7   | 0x7FFFF | 400 |   1   | RW-  |  ← Process B's stack
|  0   | 0xFFFFF |   5 |   1   | R-X  |  ← Kernel (G=1, ASID=0)
| ...  |   ...   | ... |  ...  |  ... |

When Process A is current (ASID=3): only ASID=3 entries match.
When Process B is current (ASID=7): only ASID=7 entries match.
Kernel entries (G=1): always match regardless of current ASID.
```

A.3.7 · ASID Exhaustion and TLB Shootdowns on Multi-Core

ASID exhaustion: ASIDs are typically 8–16 bits (256–65536 values). When all ASIDs are in use:

```
Solution: ASID rollover
  1. OS picks a new ASID generation number.
  2. Flushes all non-global TLB entries.
  3. Reassigns ASIDs from 0 (new generation).
  Frequency: rare on 16-bit ASID systems (65536 processes before rollover).
```

TLB shootdowns on multi-core: when a page table entry is changed (e.g., page unmapped, permissions changed), all CPUs that might have the old entry in their TLBs must be notified:

```
Core 0 unmaps VPN X:
  1. Core 0 invalidates VPN X in its own TLB.
  2. Core 0 sends an inter-processor interrupt (IPI) to all other cores.
  3. Each core invalidates VPN X in its own TLB.
  4. Core 0 proceeds only after all cores acknowledge.
This is called a TLB shootdown — expensive (hundreds of cycles).
```

📎 A.4 — TLB Performance: Coverage, Thrashing, and Huge Pages

A.4.1 · TLB Size: Small but Mighty

Modern TLBs are intentionally small, for good reasons:

```
Typical TLB sizes (data TLB, L1):
  Intel Core i7:   64 entries (4-way set associative)
  ARM Cortex-A72:  32 entries (fully associative)
  RISC-V (SiFive): 32 entries

Typical TLB sizes (instruction TLB, L1):
  Intel Core i7:   128 entries
  ARM Cortex-A72:  32 entries

Why so small?
  Fully associative hardware: every lookup checks ALL entries in parallel.
  Each entry needs a comparator circuit → area and power grow linearly.
  At 64 entries: compact, fast (1–2 cycles), low power.
  At 1024 entries: large, slower (3–5 cycles), high power, diminishing returns.
```

Many CPUs have a **two-level TLB** (L1 TLB: small/fast; L2 TLB: larger/slower) analogous to the CPU cache hierarchy.

A.4.2 · Worked Example: Array Traversal — TLB Hit and Miss Analysis

Setup: Array of 4096 integers (16 KB = 4 pages). TLB has 64 entries. Cold start.

```
int arr[4096];
for (int i = 0; i < 4096; i++) {
    arr[i] = i;
}
```

```
Access arr[0]:    VPN = arr_base_VPN + 0 → TLB miss (cold)   → load page 0 entry
Access arr[1]:    same VPN → TLB hit ✅
Access arr[2]:    same VPN → TLB hit ✅
...
Access arr[1023]: same VPN → TLB hit ✅  (4096/4 = 1024 ints per page)
Access arr[1024]: new VPN (page 1) → TLB miss → load page 1 entry
...
Access arr[4095]: VPN = page 3 → TLB hit ✅ (loaded earlier)

Total TLB misses: 4  (one per page)
Total accesses:   4096
Hit rate:         (4096 - 4) / 4096 = 99.9% 🎉
```

A.4.3 · Worked Example: Linked-List Traversal — TLB Unfriendly

Setup: Linked list of 4096 nodes. Each node randomly placed in memory (from `malloc()`).

```
struct Node { int val; struct Node *next; };
// Nodes scattered across hundreds of random pages

struct Node *p = head;
while (p) {
    process(p->val);
    p = p->next;
}
```

```
Node 0 at page 100: TLB miss (cold page)
Node 1 at page 350: TLB miss (different page)
Node 2 at page 22:  TLB miss (different page)
Node 3 at page 100: TLB hit ✅ (page 100 still in TLB)
Node 4 at page 678: TLB miss (new page)
...

Worst case: all 4096 nodes on different pages → 4096 TLB misses.
Hit rate: near 0% if working set >> TLB coverage.
```

Linked lists are inherently TLB-unfriendly. Arrays and contiguous data structures perform far better.

A.4.4 · Real TLB Hit Rates in Modern Workloads

Empirical observations from modern system workloads:

Workload	Typical TLB Hit Rate
Integer benchmarks (SPEC CPU)	99.5% – 99.9%
Floating-point benchmarks	99.0% – 99.8%
Web server (Apache, nginx)	98.0% – 99.5%
In-memory database (Redis)	97% – 99%
JVM workloads (Java server)	95% – 99%
Graph analytics (large graphs)	85% – 95%
Random pointer chasing	70% – 90%

Most common workloads achieve >99% hit rates — justifying the TLB as a highly effective optimization. Only highly irregular, pointer-heavy workloads suffer significant TLB pressure.

A.4.5 · OS Policies to Improve TLB Hit Rate

The OS can influence TLB performance through several mechanisms:

Huge Pages

Map large contiguous allocations with 2 MB or 1 GB pages. Same TLB entries cover 512× or 262,144× more address space.

Memory Layout Optimization

Place frequently co-accessed data structures on the same pages. Reduce the number of distinct pages in the working set.

NUMA-Aware Allocation

On multi-socket systems, allocate memory on the same NUMA node as the CPU accessing it. Reduces latency for TLB miss page table walks.

TLB Pre-warming (Software Prefetch)

Some architectures allow software to pre-load TLB entries before a performance-critical loop begins, eliminating cold-start misses.

📎 A.5 — Multi-Level Page Tables: Motivation and Core Idea

A.5.1 · The Crux: How to Avoid Paying for Invalid PTEs

> *"How do we get rid of all those invalid regions in the page table instead of keeping them all in memory?"*
> *— OSTEP Ch. 20*

Two observations lead directly to multi-level page tables:

1. **Most PTEs are invalid** → their only contribution is wasted RAM.
2. **Used pages cluster** → code near 0x0, stack near the top, heap in the middle. They form a small number of contiguous ranges, not random scatter.

The multi-level idea: represent the page table as a **tree**. Allocate inner nodes only when the corresponding virtual address region is actually used. Empty virtual regions have no corresponding tree nodes — they take zero physical RAM.

A.5.2 · Analogy: Sparse Matrix as a Tree vs. Dense Array

Multi-level page tables are analogous to compressed sparse matrix representations:

```
Dense array (flat page table):
  1M entries, 99.96% zeroes, 4 MB regardless of actual use.

Sparse tree (multi-level page table):
  Root node (page directory): 1024 entries, one per 4 MB region.
  Most directory entries: INVALID (their subtree is empty → no allocation).
  Valid directory entries: point to inner page tables (one per used 4 MB region).
  Each inner page table: 1024 entries covering 4 MB of virtual address space.

For our example process using 1.7 MB:
  Root: 1024 entries → only ~2–3 need valid inner tables.
  Inner tables: ~3 × 1024 × 4 bytes = ~12 KB.
  vs. flat: 4 MB.

Savings: 4 MB → ~12 KB — a 333× reduction!
```

A.5.3 · The Page Directory: The First-Level Table

The **page directory** is the root of the multi-level tree:

```
Page Directory (1024 entries, each 4 bytes):
```

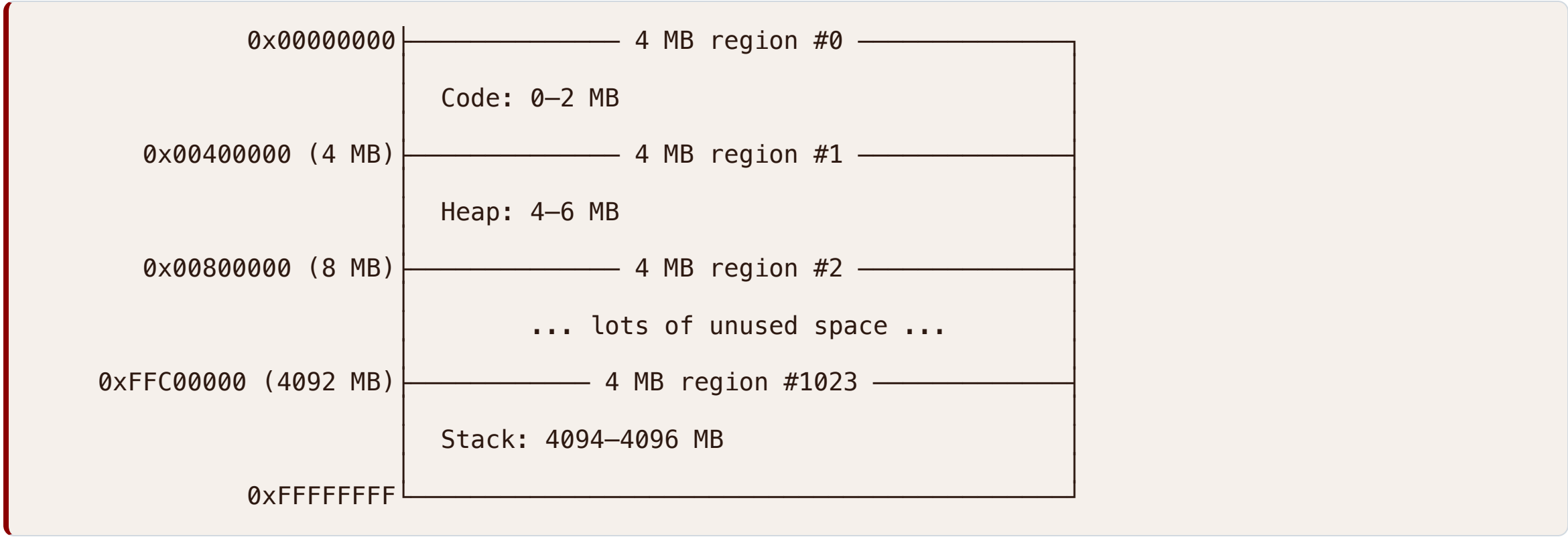
PDE	Contents
0	valid=1, PT_frame=5 ← code/data region (0–4 MB)
1	valid=0 ← empty (no inner table)
2	valid=0 ← empty
...	valid=0 ← empty (most of address space)
1022	valid=1, PT_frame=9 ← heap region
1023	valid=1, PT_frame=12 ← stack region

A **Page Directory Entry (PDE)** with `valid=0` means: the entire 4 MB virtual region covered by this entry is unmapped. **No inner page table exists.** Zero RAM allocated for that region.

A.5.4 · Space Savings: Region Layout for the Worked Example

Virtual address space: 4 GB (32-bit). **Page size:** 4 KB. PTE/PDE: 4 bytes.

Process uses: code (0–2 MB), heap (4–6 MB), stack (4094–4096 MB).



A.5.5 · Space Cost: Two-Level vs. Flat — Detailed Comparison

Scenario	Flat Page Table	Two-Level Page Table
Process uses 0–4 MB only	4 MB	4 KB (dir) + 4 KB (1 inner) = **8 KB**
Process uses 0–4 MB + 4092–4096 MB	4 MB	4 KB (dir) + 2 × 4 KB = **12 KB**
Process uses all 4 GB	4 MB	4 KB (dir) + 1024 × 4 KB = **4 MB + 4 KB**
Typical 32-bit process	4 MB	~16–64 KB

The two-level table uses more memory only when the address space is **fully dense** (all pages mapped). For sparse spaces — the common case — it is dramatically smaller.

📎 A.6 — Two-Level Page Tables: Mechanics

A.6.1 · Warm-Up: The Same Idea With Easy Numbers

Before using real 32-bit addresses, let's see the *idea* with a tiny toy example — a made-up 6-bit address space, 4-entry tables, split as `2 bits | 2 bits | 2 bits`.

```
Toy VA = 0b10 11 01   (decimal 45)

Read it left to right, in three 2-bit groups:
  VPN1   = 10 (binary) = 2   ← which "directory" entry
  VPN2   = 11 (binary) = 3   ← which "inner table" entry
  Offset = 01 (binary) = 1   ← which byte inside the page

No shifting/masking needed here — just read the groups directly,
because we wrote the address already split into groups.
```

The real 32-bit case works **identically** — the only difference is that a real address is written as one long hex number instead of pre-grouped binary, so we need `>>` and `&` to *do the grouping for us*. Keep this toy example in mind as you read the formulas below.

A.6.2 · Level-1 Lookup: Indexing into the Page Directory

Step 2 in detail:

```
PTBR = 0x00010000   (page directory lives at physical frame 16)
        (in decimal: 0x00010000 = 65,536 = 16 × 4,096 — frame 16, as stated)
VPN1 = 1            (from VA decomposition)

PDE address = PTBR + VPN1 × 4
            = 0x00010000 + 1 × 4
            = 0x00010004     (decimal: 65,536 + 4 = 65,540)

Memory read: PDE = RAM[0x00010004]

PDE contents (example):
  bit 0 (valid) = 1   ← inner page table exists ✅
  bits 31–12 (frame)  = 25  ← inner PT is at physical frame 25
                            (physical address = 25 × 4096 = 0x19000)

Proceed to Level-2 lookup using PT at frame 25.
```

A.6.3 · Validity Check on PDE: If Invalid, Fault Immediately

If the PDE's valid bit is 0, there is no inner page table — the entire 4 MB virtual region is unmapped:

```
PDE = RAM[PDE_addr]

if PDE.valid == 0:
  → No inner page table exists.
  → The process accessed an unmapped virtual region.
  → Hardware raises a Page Fault exception.
  → OS page fault handler runs.
  → Since no page was ever allocated here, handler sends SIGSEGV.
  → Process terminated.

This check happens BEFORE we try to access the inner page table.
No second memory access is wasted on an invalid region.
```

A.6.4 · Level-2 Lookup: Indexing into the Inner Page Table

Step 3 in detail, continuing from the previous example:

```
Inner PT base = PDE.frame × 4096 = 25 × 4096 = 102,400 = 0x00019000
VPN2          = 3  (from VA decomposition)

PTE address = 0x00019000 + 3 × 4
            = 0x00019000 + 12
            = 0x0001900C     (decimal: 102,400 + 12 = 102,412)

Memory read: PTE = RAM[0x0001900C]

PTE contents (example):
  bit 0  (valid)    = 1   ← page is present in physical RAM ✅
  bit 1  (writable) = 1   ← page is writable
  bit 2  (user)     = 1   ← accessible in user mode
  bits 31–12 (PFN)  = 150 ← data page is at physical frame 150
```

📎 A.7 — Beyond Two Levels: 64-bit Systems

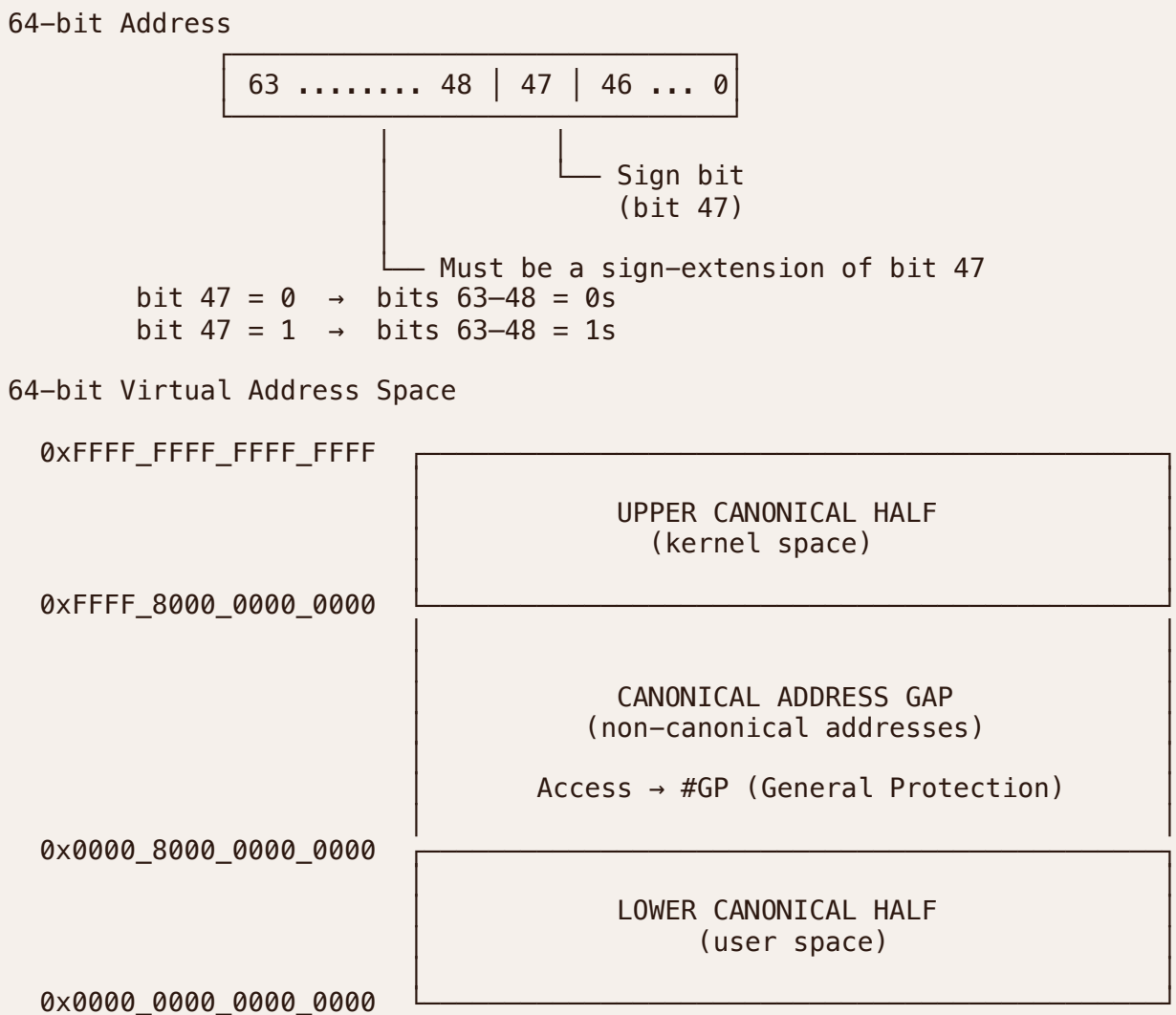
A.7.1 · x86-64 48-bit Virtual Address Structure (1/2): Reading the Fields

The idea is exactly the one you just used for the 32-bit two-level case — the only change is that we now peel off **four** 9-bit groups instead of two 10-bit groups, because there are 4 levels:

```
Reading a 64-bit address, left to right, in groups:
  bits 63–48 : "sign extension" — just 16 copies of bit 47 (explained below)
  bits 47–39 : PML4 index  (9 bits → 0–511) ── 1st table to visit
  bits 38–30 : PDPT index  (9 bits → 0–511) ── 2nd table to visit
  bits 29–21 : PD index    (9 bits → 0–511) ── 3rd table to visit
  bits 20–12 : PT index    (9 bits → 0–511) ── 4th (leaf) table to visit
  bits 11–0  : Page offset (12 bits → 0–4095) ── byte inside the final page
```

Why "sign extension"? x86-64 chips only wire up 48 real address bits, but a register is 64 bits wide. To keep the unused upper 16 bits from causing confusion, the hardware requires them to simply *copy* whatever bit 47 is (all 0s, or all 1s); it is just a validity rule the hardware checks.

A.7.2 · x86-64 48-bit Virtual Address Structure (2/2): Canonical Addresses



A.7.3 · Five-Level Page Tables: x86-64 PML5 (Linux 5.14+)

As applications demand more than 128 TB of virtual address space, x86-64 added a fifth level:

```
Five-level page table (PML5, Intel "Ice Lake" 2019+):
  57-bit virtual address space (vs. 48-bit for 4-level).

  VA structure:
```

PML5 (9 bits)	PML4 (9 bits)	PDPT (9 bits)	PD (9 bits)	PT (9 bits)	Offset (12 bits)

```
  Total VA bits: 5 × 9 + 12 = 57 bits
  Virtual address space: 2^57 = 128 PiB (petabytes)

  Memory accesses per translation (no TLB): 5 + 1 = 6.
  With TLB: still just 1 (TLB completely hides the extra level).

  Enabled in Linux kernel 5.14+ with CONFIG_X86_5LEVEL.
  Useful for: very large in-memory databases, HPC applications.
```

A.7.4 · RISC-V Sv39: Three-Level Page Tables

RISC-V defines multiple virtual memory schemes. **Sv39** (39-bit VA, used by Linux on RISC-V):

```
39-bit Virtual Address (Sv39):

┌──────────┬──────────┬──────────┬──────────────┐
│  VPN[2]  │  VPN[1]  │  VPN[0]  │    Offset    │
│ (9 bits) │ (9 bits) │ (9 bits) │  (12 bits)   │
└──────────┴──────────┴──────────┴──────────────┘
  indices: 0–511      0–511      0–511

  Virtual address space: 2^39 = 512 GiB
  (Split: lower 256 GiB user, upper 256 GiB kernel)

  Translation: satp register → root page table → 3 levels → PA.
  Memory accesses per translation (no TLB): 3 + 1 = 4.

RISC-V also defines:
  Sv48: 4-level, 48-bit VA, 256 TiB address space.
  Sv57: 5-level, 57-bit VA (same as x86-64 PML5).
```

A.7.5 · Space Analysis: Four-Level Page Table for a Sparse Process

How much RAM does an x86-64 process's page table occupy?

```
Minimal process (code + stack only):
  PML4 table:  1 × 4 KB = 4 KB   (always 1, pointed to by CR3)
  PDPT tables: 1 × 4 KB = 4 KB   (one for user code region)
  PD tables:   1 × 4 KB = 4 KB
  PT tables:   1 × 4 KB = 4 KB   (one page table page covers 2 MB)
  Total: 4 levels × 4 KB = 16 KB

  + 1 more PT for the stack (different region):
  Total: ~20–32 KB for a minimal process.

Compare to flat table for 48-bit VA:
  2^36 entries × 8 bytes = 512 GB ← impossible.
  Four-level: ~16–32 KB ← trivially manageable.
```

A.7.6 · Time Analysis: Multi-Level Walks and the TLB

The critical relationship between page table depth and TLB necessity:

Architecture	Levels	Accesses (no TLB)	Accesses (TLB hit)
Base/Bounds	0	1	1
Flat paging	1	2	1
Two-level	2	3	1
Three-level (RISC-V Sv39)	3	4	1
Four-level (x86-64)	4	5	1
Five-level (x86-64 PML5)	5	6	1

With a TLB hit rate of 99%, the effective number of accesses is always close to 1, regardless of the number of levels. The TLB completely absorbs the translation overhead.

A.7.7 · Page Table Pages Themselves Can Be Paged

An elegant recursive property: the page table pages live in physical memory and are themselves subject to memory management.

The OS manages physical frames with a frame free list. When a new inner page table is needed:

- OS allocates one physical frame from the free list.
- OS zeroes the frame (all PTEs = invalid).
- OS sets the parent PDE to point to this new frame.

Page table pages can be evicted to disk (swapped out):

- If the OS is under memory pressure, even page table pages can be swapped to disk — as long as the OS knows how to bring them back when a translation miss occurs.

This creates a recursive dependency:

- To access a page, we need the page table.
- To access the page table, we need it to be in physical memory.
- The OS must ensure page table pages are pinned (non-swappable) or handle the recursive fault carefully.

📎 A.8 — OS Responsibilities and Advanced Topics

A.8.1 · Inverted Page Tables: A Different Approach

Standard page tables are **per-process**: one table per process, indexed by VPN.

Inverted page tables flip this: one **global** table indexed by physical frame number:

```
Inverted Page Table (one entry per physical frame):
```

Frame	PID	VPN	Flags	
0	OS	0	R-X	← OS kernel uses frame 0
1	3	0x401	R-X	← Process 3, VPN 0x401
2	7	0x001	RW-	← Process 7, VPN 0x001
...	...	...	...	

```
Size: one entry per physical frame → proportional to RAM size, not VA size.
For 4 GB RAM, 4 KB frames: 1M entries × 16 bytes = 16 MB (total, shared).
Compare: per-process flat table for 64-bit = 512 GB each!
```

Used by: IBM PowerPC (older), HP-PA RISC, MIPS (some variants).

A.8.2 · Inverted Page Tables: Lookup

To translate VPN → PFN with an inverted page table:

```
Input: (PID, VPN)

Search: scan the inverted table for an entry where
        entry.PID == current_PID AND entry.VPN == VPN.

Found:  row index = PFN → PA = PFN × page_size + offset ✅
Not found: page fault.

Problem: linear search is O(n) where n = number of physical frames.
  For 4 GB RAM: 1M frames → 1M comparisons → far too slow!

Solution: hash the (PID, VPN) pair → hash table into the inverted table.
  Most lookups: O(1) average case (with collision chaining for rare conflicts).

Used with TLB: TLB handles the common case (hit).
Inverted PT only consulted on TLB miss (rare).
```

A.8.3 · Hashed Page Tables for Sparse 64-Bit Spaces

An alternative to multi-level tables for very sparse 64-bit address spaces:

Hashed Page Table:

```
  Hash function: h(VPN) → bucket index.
  Each bucket: linked list of (VPN, PFN, flags) tuples.
```

Advantages:

```
- Size proportional to number of MAPPED pages (not VA space size).
- Good for extremely sparse 64-bit processes.
```

Disadvantages:

```
- Collision chains can degrade worst-case lookup time.
- Less cache-friendly than a linear array lookup.
- More complex OS management.
```

Used by: Solaris (with hardware support for hashed PTEs). Less common than multi-level tables in modern mainstream systems.

A.8.4 · Transparent Huge Pages (THP) in Linux

Linux can automatically use 2 MB pages for large contiguous anonymous mappings:

```
User program calls: mmap(NULL, 512*1024*1024, PROT_RW, MAP_ANON, -1, 0)
  → Requests 512 MB of anonymous memory.

Without THP:
  OS allocates 131,072 × 4 KB frames.
  TLB must hold 131,072 entries to cover the region (impossible — TLB has ~64).
  Many TLB misses during access.

With THP (enabled by default on Linux):
  OS backs the region with 256 × 2 MB huge pages.
  Only 256 TLB entries needed to cover the entire 512 MB!
  Dramatic reduction in TLB misses.

Control in Linux:
  /sys/kernel/mm/transparent_hugepage/enabled
  echo always  > ...   # use THP when possible
  echo madvise > ...   # use only when program hints (madvise MADV_HUGEPAGE)
  echo never   > ...   # disable THP
```

A.8.5 · Page Table Security: SMEP, SMAP, NX

Modern x86-64 CPUs provide hardware security features in the page table:

NX (No-Execute) Bit

- PTE bit 63: if set, the page cannot be executed.
- Prevents stack/heap buffer overflow exploits that inject shellcode.
- Linux: data pages (`RW-`) always have NX set.

SMEP (Supervisor Mode Execution Prevention)

- CR4.SMEP: if set, the kernel cannot **execute** code from user-mode pages.
- Prevents privilege escalation by redirecting kernel execution into user-controlled pages.

SMAP (Supervisor Mode Access Prevention)

- CR4.SMAP: if set, the kernel cannot **read/write** user-mode pages (except with explicit `STAC` / `CLAC` instructions).
- Prevents kernel code from being tricked into reading attacker-controlled data.

📎 A.9 — Worked Examples & Exam Practice

A.9.1 · Problem 1: TLB EMAT Calculation

Given:

- TLB access time: T_{TLB} = 2 ns
- RAM access time: T_{mem} = 100 ns
- TLB miss penalty (page table walk in RAM): T_{PT} = 100 ns (one level)
- TLB hit rate: h = 0.97

Calculate EMAT and compare to the no-TLB baseline.

$$\begin{aligned}\text{EMAT} &= h \cdot (T_{\text{TLB}} + T_{\text{mem}}) + (1-h) \cdot (T_{\text{TLB}} + T_{\text{PT}} + T_{\text{mem}}) \\ &= 0.97 \times (2 + 100) + 0.03 \times (2 + 100 + 100) \\ &= 0.97 \times 102 + 0.03 \times 202 = 98.94 + 6.06 = \mathbf{105}\ \text{ns}\end{aligned}$$

No-TLB baseline: $T_{\text{PT}} + T_{\text{mem}}$ = 200 ns.

Overhead: 105/100 = 5% slowdown vs. ideal (100 ns), yet 1.9× faster than no-TLB (200 ns).

A.9.2 · Problem 1: Sensitivity Analysis

Let's see how EMAT varies with hit rate:

Hit Rate h	EMAT (ns)	Overhead vs. ideal (100 ns)
1.00 (perfect)	102	2%
0.99	103	3%
0.97	105	5%
0.95	107	7%
0.90	112	12%
0.80	122	22%
0.50	152	52%
0.00 (no TLB)	202	102% (2× slower)

The payoff is steep: going from 90% to 99% hit rate drops overhead from 12% to 3%. This is why CPU architects invest heavily in maximizing TLB hit rates.

A.9.3 · Problem 2: Two-Level Translation — Additional Addresses

Setup (same as the Capstone Worked Example in § 9): PTBR = `0x00002000`; PDE[0] → frame 10, PDE[1] → frame 11; PT at frame 10: PTE[3] → PFN 50 (RW-), PTE[5] → PFN 60 (R-X); PT at frame 11: PTE[0] → PFN 70 (RW-); everything else invalid.

Translate VA = `0x00005000`:

```
VPN1   = 0x00005000 >> 22 = 0
VPN2   = (0x00005000 >> 12) & 0x3FF = 5
Offset = 0x00005000 & 0xFFF = 0

Step 1: PDE[0] = frame 10, valid=1 → PT at 0xA000
Step 2: PTE[5] = PFN 60, valid=1, prot=R-X ✅
Step 3: PA = 60 × 4096 + 0 = 245,760 = 0x3C000
```

Translate VA = `0x00400010`:

```
VPN1   = 0x00400010 >> 22 = 1
VPN2   = (0x00400010 >> 12) & 0x3FF = 0
Offset = 0x00400010 & 0xFFF = 0x010 = 16

Step 1: PDE[1] = frame 11, valid=1 → PT at 0xB000
Step 2: PTE[0] = PFN 70, valid=1, prot=RW- ✅
Step 3: PA = 70 × 4096 + 16 = 286,720 + 16 = 286,736 = 0x46010
```

A.9.4 · Problem 3: Space Savings — Two-Level vs. Flat (1/2)

Scenario: A 32-bit process uses the following virtual regions:

- Code: VA 0x00000000 – 0x00100000 (1 MB = 256 pages)
- Heap: VA 0x00200000 – 0x00280000 (512 KB = 128 pages)
- Stack: VA 0xFFF00000 – 0xFFFFFFFF (1 MB = 256 pages)

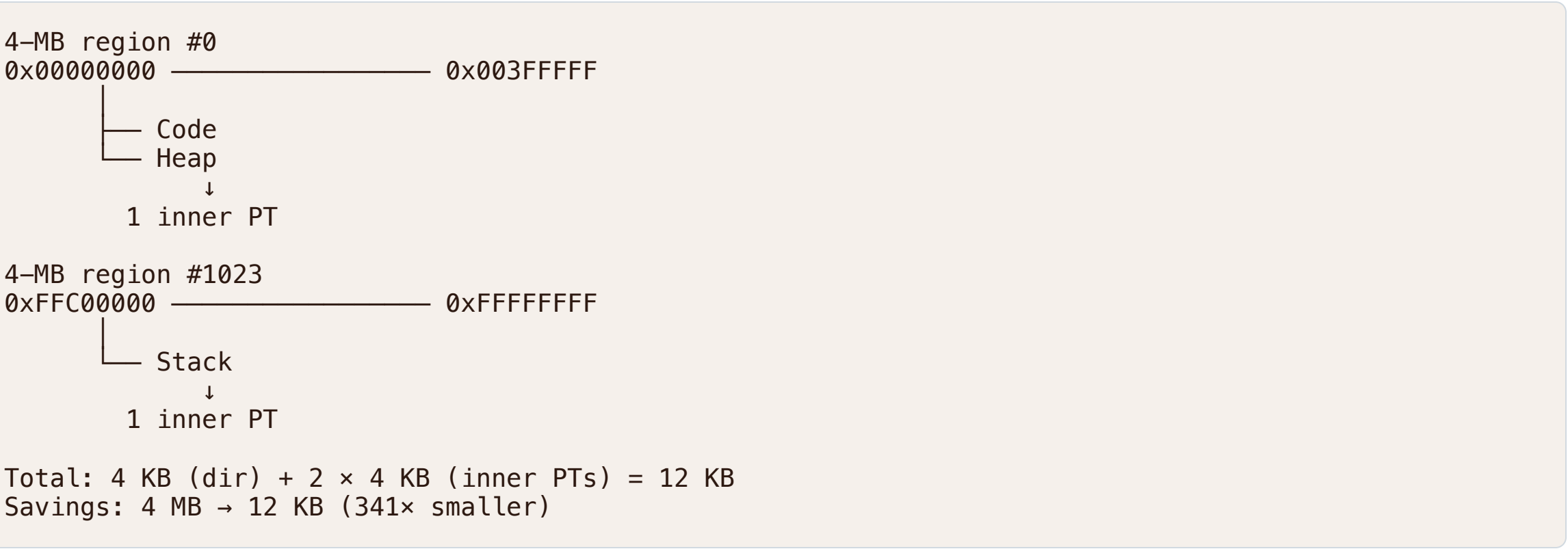
Flat page table:

$$\text{entries} = 2^{20} = 1,048,576 \quad \text{size} = 1,048,576 \times 4 = 4\ \text{MB}$$

Two-level page table (10-10-12):

```
Page directory: 1 × 4 KB = 4 KB
How many inner PTs are needed? Remember: one inner PT covers 4 MB of VAS
```

A.9.5 · Problem 3: Space Savings — Two-Level vs. Flat (2/2)



A.9.6 · Problem 4: x86-64 Four-Level Page Walk (1/3)

Given: VA = `0x0000_7FFF_FFFF_1234`. CR3 = physical address of PML4.

Step A — convert hex to binary, one digit at a time (4 bits each), using the toolkit table from the start of the lecture:

```
Hex digit:   0    0    0    0    7    F    F    F    F    F    F    F    1    2    3    4
Binary:   0000 0000 0000 0000 0111 1111 1111 1111 1111 1111 1111 1111 0001 0010 0011 0100
```

A.9.7 · Problem 4: x86-64 Four-Level Page Walk (2/3)

Step B — group those 64 bits according to the field widths from the previous slide (16 | 9 | 9 | 9 | 9 | 12):

Field	Bits taken	Binary	Decimal index
Sign-extend	top 16	`0000 0000 0000 0000`	(all 0 — valid, lower canonical half)
PML4	next 9	`0 1111 1111`	255
PDPT	next 9	`1 1111 1111`	511
PD	next 9	`1 1111 1111`	511
PT	next 9	`1 1111 1111`	511
Offset	last 12	`0001 0010 0011 0100`	0x234 = 564

*(Tip: it's easiest to mark off the groups starting from the **right** — offset first, then PT, then PD, and so on — since the field widths are given from the bottom up.)*

A.9.8 · Problem 4: x86-64 Four-Level Page Walk (3/3)

Step C — walk the tables using those indices:

```
PML4[255] → PDPT frame
PDPT[511] → PD   frame
PD  [511] → PT   frame
PT  [511] → data frame (PFN)
PA = PFN × 4096 + 564
```

A.9.9 · Problem 5: TLB Miss Rate for Array Traversal

Setup:

- Array of 32,768 integers (128 KB = 32 pages of 4 KB)
- TLB has 16 entries (cold start)
- Array accessed sequentially twice (two passes)

Pass 1 (cold TLB):

```
32 unique pages accessed → 32 TLB misses (one per new page)
Total accesses = 32,768 (one per element)
Misses = 32
Hit rate = (32,768 − 32) / 32,768 = 99.9%
```

Pass 2 (TLB may still be warm):

```
TLB has 16 entries. Array spans 32 pages.
TLB holds only 16 of the 32 page translations.
Pages 16–31 were evicted during pass 1 (FIFO eviction assumed).
→ Pass 2: 16 misses (pages 0–15 still in TLB, pages 16–31 evicted).
Total pass 2 misses = 16.
Hit rate pass 2 = (32,768 − 16) / 32,768 = 99.95%
```

A.9.10 · Exam Strategy: Multi-Level Translation Problems

For any multi-level page table exam question, use this systematic approach:

```
Step 1: Determine the VA decomposition.
  - Identify address bits, page size → offset bits = log₂(page_size).
  - Identify number of levels and bits per level.
  - Write VA as: VPN1 | VPN2 | ... | VPNk | Offset.

Step 2: Extract each index from the VA.
  - Use bit shifts and masks:
    VPN1 = VA >> (offset_bits + (k-1) × level_bits) & mask
  - Work from the top down.

Step 3: Perform the walk, level by level.
  - For each level: compute table entry address, read entry, check valid bit.
  - If any valid=0: stop → Page Fault (note which level failed).

Step 4: Compute PA.
  - PA = PFN × page_size + Offset.
  - Double-check: PA bits = PFN bits ‖ Offset bits.

Step 5: Count memory accesses.
  - Without TLB: k level accesses + 1 data access = k+1 total.
  - With TLB hit: 1 data access only.
```

A.9.11 · Common Mistakes to Avoid

Mistake	Correct Approach
Confusing PDE and PTE	PDE is in the page directory (level 1); PTE is in the inner table (leaf level)
Using `VA / page_size` for VPN1	For multi-level: VPN1 = VA >> (offset + lower_level_bits)
Adding instead of concatenating PFN + offset	PA = PFN × page_size + offset (not PFN + offset)
Forgetting to check all valid bits	Check every level's valid bit; any 0 → Page Fault
Claiming 2 memory accesses for 2-level	Two-level = 3 accesses (PDE + PTE + data); k levels = k+1 accesses
Forgetting ASID in multi-process scenarios	Each process has its own ASID; TLB entries are tagged
Saying TLB flush happens on every memory access	TLB flush happens only on context switch (without ASID) or explicit invalidation

A.9.12 · Key Formulas

EMAT with TLB:

$$\text{EMAT} = h \cdot (T_{\text{TLB}} + T_{\text{mem}}) + (1 - h) \cdot (T_{\text{TLB}} + T_{\text{PT}} + T_{\text{mem}})$$

TLB coverage:

$$\text{Coverage} = \text{TLB entries} \times \text{page size}$$

VA decomposition (k-level, equal split):

$$\text{offset bits} = \log_2(\text{page size}), \quad \text{VPN}_i \text{ bits} = \frac{\text{VA bits} - \text{offset bits}}{k}$$

Page table size (flat, for reference):

$$\text{Size} = \frac{2^{\text{VA bits}}}{\text{page size}} \times \text{PTE size}$$

Physical address assembly:

$$\text{PA} = \text{PFN} \times \text{page size} + \text{offset}$$

A.9.13 · Self-Check Bank (1/2): Questions from § 1–4

Extra questions beyond the in-class checkpoints:

§ 1–2 — TLB concept

- Explain how **temporal locality** makes a TLB effective. Give a code example with strong temporal locality.
- A loop accesses 10,000 elements of an array spanning 10 pages. With a cold TLB, how many TLB misses occur? What is the hit rate?
- Explain how **spatial locality** reduces TLB misses for array traversal. Which property of arrays is responsible?

§ 3 — Control flow and ASIDs

- A system has 8-bit ASIDs (256 values). After 256 processes have been assigned ASIDs, a 257th is created. What must the OS do?
- Explain **TLB shootdowns**. Why are they necessary on multi-core systems when a PTE changes?
- What is the purpose of the **Global (G) bit**? Give an example of when it is 1.

§ 4 — Performance

- A program accesses a 2 MB array sequentially. How many TLB misses occur with 4 KB pages? With 2 MB

A.9.14 · Self-Check Bank (2/2): Questions from § 5–8

§ 5–6 — Multi-level tables. Setup as in Checkpoint 3 (PTBR = `0x00005000`; PD[0] → frame 20, PD[2] → frame 30; PT at frame 20: PTE[0] → PFN 100 RW-, PTE[1] → PFN 101 R-X; PT at frame 30: PTE[5] → PFN 200 RW-).

- Explain the "sparse matrix" analogy. What is the analogue of (a) a non-zero element, (b) a zero element, (c) the row-index structure?
- As a process grows its heap to 100 MB, how does the two-level table grow? Compare with the flat table.
- Translate VA = `0x00001000`. Is a write to it allowed?
- Translate VA = `0x00805030`. Show all steps.
- A process allocates 1 MB of heap starting at VA = `0x01000000`. How many new inner page tables must the OS allocate? How many new PTEs?

§ 7–8 — Beyond two levels

- A RISC-V Sv39 process has one code page and one stack page. How many physical frames does its page-table structure consume?
- What is the *canonical address gap* in x86-64? What happens if a user program accesses it?
- Why could page-table pages theoretically be swapped to disk? What complication does this create?