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 2.5: Virtualization V

Segmentation and Introduction to Paging

Opening Thought

> *"Segmentation solves the problem of internal fragmentation but introduces external fragmentation. Paging solves both by using fixed-size units — but at the cost of a new structure: the page table."*
> *— paraphrasing Remzi & Andrea Arpaci-Dusseau, OSTEP Ch. 16–18*

In Lecture 2.4 we built the address space abstraction and saw how base/bounds provides a first, concrete mechanism for memory virtualization. But one base/bounds pair per process wastes enormous physical RAM on the heap-stack gap. Today we fix that — first with segmentation, then with paging — and in doing so, we trace the central arc of the entire memory virtualization story.

Lecture Roadmap

This lecture is organized into **8 major sections**:

1. **Recap & Motivation** — Why base/bounds is not enough
2. **Segmentation: The Core Idea** — One base/bounds pair per segment
3. **Segmentation: Address Translation in Detail** — Offsets, protection, sharing
4. **OS Support for Segmentation** — Creation, context switch, growth, free-space
5. **Limitations of Segmentation** — External fragmentation and the path beyond
6. **Introduction to Paging** — Fixed-size pages and page tables
7. **Paging: Address Translation in Detail** — VPN, offset, PTE fields, costs
8. **Comparison & Summary** — Segmentation vs. Paging, key formulas, next steps

> *Extended material (derivations, extra worked examples, history, reference) is in the **Appendix** (A.1–A.8): each section divider points to its appendix counterpart.*
>
> *Let's begin where Lecture 2.4 left off.*

Prerequisite Check

Before proceeding, ensure you are comfortable with:

- **Base/bounds translation:** `PA = VA + base`, bounds check `VA < bounds`.
- **Address space layout:** code, static data, heap (grows ↑), stack (grows ↓).
- **Virtual vs. physical address:** the fundamental distinction.
- **OS free list:** a linked list of available physical memory ranges.
- **Context switch:** saving/restoring hardware registers (PC, SP, GPRs, base/bounds).
- **Internal vs. external fragmentation:** the two types of memory waste.

If any of these feel uncertain, review Lecture 2.4 before continuing.

§ 1 — Recap & Motivation

📎 *Extended material → Appendix A.1*

Where We Left Off: Base/Bounds Limitations

Lecture 2.4 established that base/bounds (dynamic relocation) works but has four critical limitations:

1. The entire address space must fit **contiguously** in physical memory.
2. The heap-stack gap wastes physical RAM (**internal fragmentation**).
3. Address spaces larger than physical RAM are **impossible**.
4. Code sharing between processes is **impossible** with a single pair.

   *Because every address space occupies an equal-sized slot, free-space management is trivial and there is **no external fragmentation**. It appears only when allocation units become variable-sized — as we are about to see with segmentation.*

We need a mechanism that allocates physical RAM only for the parts of the address space that are actually in use.

The Key Insight: Why One Monolithic Block?

The root cause of internal fragmentation with base/bounds is treating the address space as **one indivisible unit**:

```
One base/bounds pair → one contiguous physical allocation
→ entire virtual range (0 to bounds) must be reserved in physical RAM
→ the heap-stack gap is physically reserved even though it is empty
```

But look at what a process's address space actually contains:

```
Three logically independent regions:
  1. Code segment   — fixed size, read-only
  2. Heap segment   — grows upward from a known base
  3. Stack segment  — grows downward from a known top
```

What if we gave each region **its own independent** base/bounds pair? Then only the active regions occupy physical RAM.

The Generalization: One Base/Bounds Per Segment

The leap from base/bounds to segmentation is conceptually simple:

```
Base/bounds (Lecture 2.4):
  One pair: (base, bounds) → entire address space

Segmentation (this lecture):
  Three pairs:
    Code:  (base_code,  bounds_code)
    Heap:  (base_heap,  bounds_heap)
    Stack: (base_stack, bounds_stack)
```

Each segment is translated and protected independently. The physical locations of code, heap, and stack are completely unrelated to each other. The gap between heap and stack takes up **zero physical bytes**.

What Segmentation Promises to Fix

Problem (with base/bounds)	Fixed by Segmentation?
Internal fragmentation (heap-stack gap)	✅ Yes — gap no longer allocated
Code sharing between processes	✅ Yes — code segment mapped to same physical frames
Address space larger than physical RAM	❌ No — each segment still needs contiguous physical space
Sparse 64-bit address spaces	⚠️ Partially — but not fully

But there is a price: segments are *variable-sized*, so segmentation introduces a **new** problem — **external fragmentation** (§ 4–5).

Segmentation is a major improvement over base/bounds, but it is not the final answer.

✅ Checkpoint 1 — Recap & Motivation

Test your understanding:

1. In base/bounds, a process has a 32 KB address space but only 6 KB of code, 2 KB of heap, and 2 KB of stack in active use. How much physical RAM is wasted?
2. Explain in one sentence why treating the address space as one monolithic block causes internal fragmentation.
3. How many base/bounds pairs does segmentation use per process? What is each one responsible for?
4. Name two problems that segmentation fixes over base/bounds, and one problem it does not fix. What new problem does it introduce?

Write your answers before moving on.

§ 2 — Segmentation: The Core Idea

📎 *Extended material → Appendix A.2*

Definition of a Segment

> ***A segment is a contiguous portion of the address space with a particular length and associated permissions.***

In the classic three-segment model:

- **Code segment:** holds executable instructions. Read-only (or read+execute). Fixed size.
- **Heap segment:** holds dynamically allocated memory (`malloc`). Read+write. Grows upward.
- **Stack segment:** holds function call frames and local variables. Read+write. Grows downward.

Each segment is a **logically cohesive** unit. The hardware treats each one as having its own base and bounds — its own slice of physical memory, completely independent of the others.

The Segment Table

The OS maintains a **segment table** — one entry per segment — in the process's PCB:

Segment Table for Process A:

Seg	Base	Bounds	Protection
Code	32,768	2,048	R-X
Heap	36,864	3,072	RW-
Stack	28,672	2,048	RW-

(base in bytes) (bounds in bytes)

On a context switch, the OS loads this table into the hardware segment registers. On a memory access, the hardware picks the correct row, translates the address, and checks the protection bits.

Segment Types: Code, Heap, Stack

A quick orientation to how each segment is used:

Code Segment

Loaded from the ELF `.text` section. Size fixed at load time. Marked read-only to prevent self-modification. Can be shared among multiple processes running the same binary.

Heap Segment

Starts just above the code/data sections. Grows upward as the program calls `malloc()`. The C runtime calls `sbrk()` → OS syscall to extend the heap segment's bounds.

Stack Segment

Starts at the top of the address space and grows **downward**. Each function call pushes a frame; each return pops it. Special translation logic required (see § 3).

Diagram: Virtual Address Space with Three Segments

A process's virtual address space with explicit segmentation:

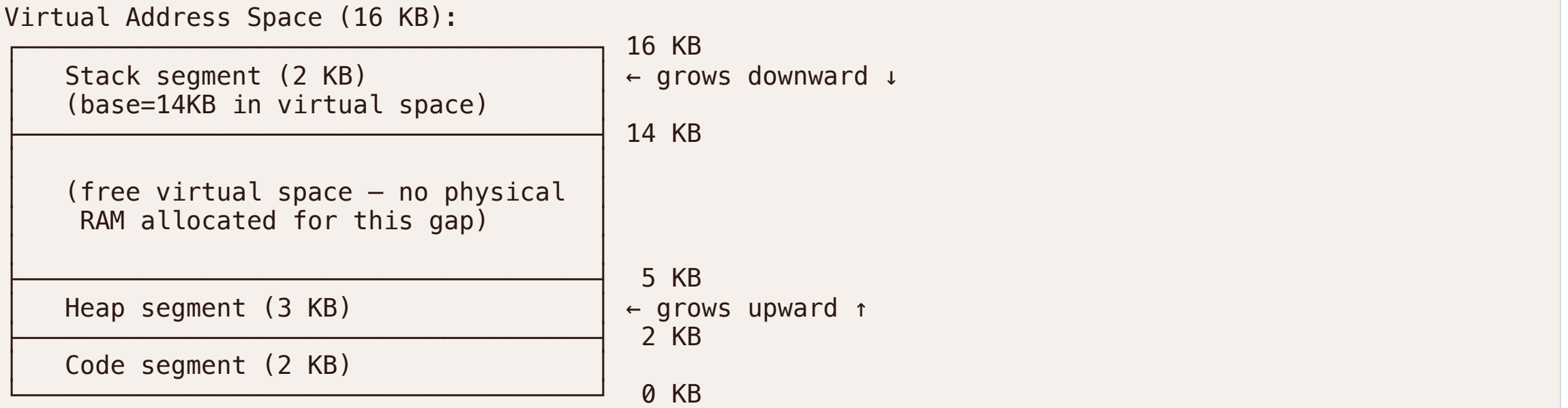
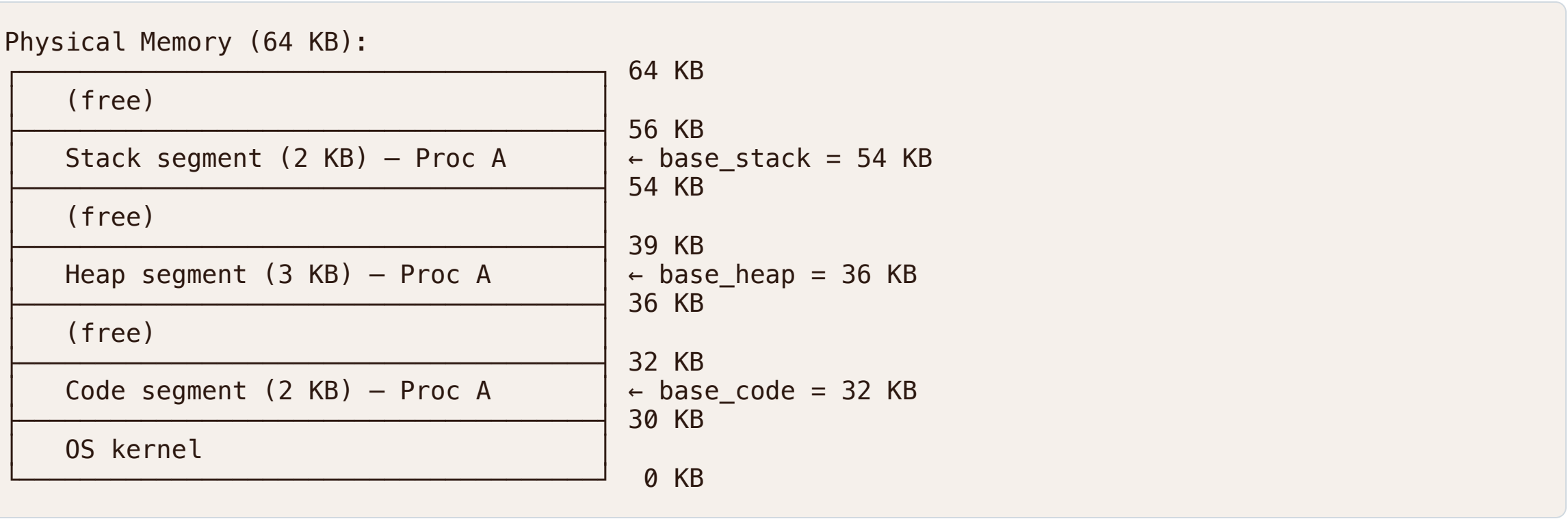


Diagram: Physical Memory Layout — Segments Placed Independently

The three segments land in completely unrelated locations in physical RAM:

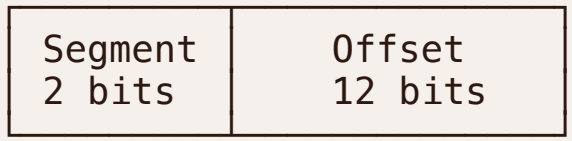


Physical RAM used by Process A: 2 + 3 + 2 = **7 KB** — not 16 KB!

Explicit Approach: Top Bits as Segment Number

With a 14-bit virtual address and 3 segments, we use the **top 2 bits** as the segment selector:

```
Virtual address (14 bits):
```




```
Segment encoding:
  00 → Code  segment
  01 → Heap  segment
  11 → Stack segment
  10 → (unused / invalid)

Example: VA = 01000000001100 = 0x100C
  Segment bits = 01 → Heap
  Offset = 0x00C = 12
  → Look up Heap base, add 12
```

✅ Checkpoint 2 — Segmentation: The Core Idea

Test your understanding:

1. A process has code = 2 KB, heap = 4 KB, stack = 2 KB. Under base/bounds (16 KB address space), how much physical RAM is wasted? Under segmentation?
2. Draw the segment table for a process with code at physical 40 KB (2 KB), heap at physical 20 KB (3 KB), stack at physical 56 KB (2 KB).
3. In the explicit approach, how does the hardware know which segment a virtual address belongs to? How many bits does it use for 3 segments, and what remains for the offset in a 14-bit address?
4. What is stored in each segment table entry?

Write your answers before moving on.

§ 3 — Segmentation: Address Translation in Detail

📎 *Extended material → Appendix A.3*

Address Translation with Explicit Segmentation

With explicit segmentation (top-2-bits segment selector), the translation algorithm is:

```
Given virtual address VA (14 bits):

1. Extract segment number:  seg    = VA >> 12        (top 2 bits)
2. Extract offset:          offset = VA & 0x0FFF     (bottom 12 bits)
3. Look up segment table:   entry  = seg_table[seg]
4. Check validity:          if entry.valid == 0 → fault (segment not present)
5. Check bounds:            if offset >= entry.bounds → fault (out of bounds)
6. Check protection:        if access_type not in entry.prot → fault (protection)
7. Compute physical address: PA = entry.base + offset
```

Steps 4, 5, and 6 are all checked **before** the physical address is used.

The Translation Formula: PA = base[seg] + offset

The core formula for segmentation translation:

$$\text{PA} = \text{base}[\text{seg}] + \text{offset}$$

Subject to the validity checks:

$$\text{offset} \geq 0 \quad \text{AND} \quad \text{offset} < \text{bounds}[\text{seg}]$$

This is identical to base/bounds, except that:

- `base` and `bounds` are **per-segment** (not global).
- The `seg` index selects the correct row from the segment table.
- There is an additional **protection** check.

The translation is still a single addition — implemented in hardware in a few clock cycles.

Worked Example: Code Segment Access

Setup:

```
Segment table:
  Code:  base = 32 KB = 32,768,  bounds = 2,048,  prot = R-X
  Heap:  base = 34 KB = 34,816,  bounds = 3,072,  prot = RW-
  Stack: base = 28 KB = 28,672,  bounds = 2,048,  prot = RW-  direction=down
```

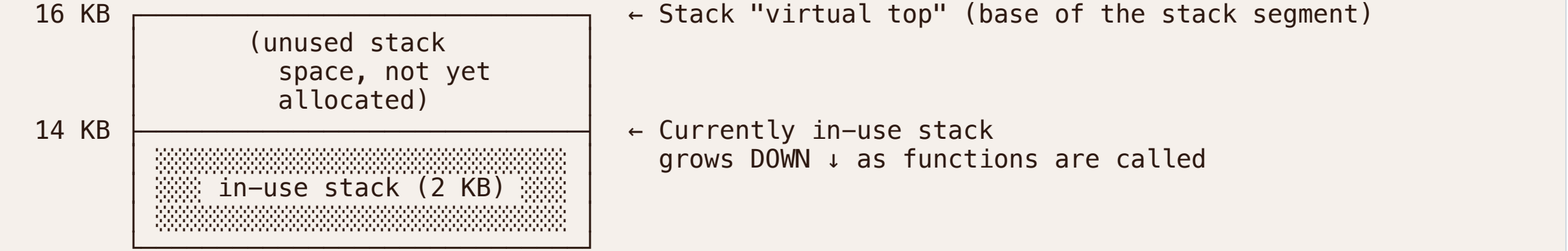
Access: Instruction fetch at virtual address `0x0068` (= 104 decimal).

```
1. seg    = 0x0068 >> 12 = 0b00 → Code segment
2. offset = 0x0068 & 0x0FFF = 104
3. Bounds check: 104 < 2,048 ✅
4. Protection check: fetch = Execute, prot = R-X ✅
5. PA = 32,768 + 104 = 32,872

→ CPU fetches instruction from physical address 32,872.
```

The Stack: A Special Case

Unlike code, heap, or other segments, the stack **grows downward** — toward lower virtual addresses as it's used.



Stack Translation Example: Negative Offsets

For a downward-growing segment, the offset extracted from the virtual address means "*this many bytes below the top of the segment's address range*" — not "*this many bytes above the base.*"

```
Virtual address of a stack access: 0x3C00 = 15,360
Segment's offset field is 12 bits → max_size = 2^12 = 4096

  1. segment  = 15,360 >> 12 = 0b11 → Stack segment
  2. offset   = 15,360 & 0x0FFF = 0xC00 = 3,072

  top of segment
  (virtual 16 KB)
        │
        │  ↓ offset = 3072   (raw bits from VA)
        │
        ▼
  accessed byte
  (virtual 15 KB)

  negative_offset = offset - max_size
                  = 3072 - 4096
                  = -1024   → "1024 bytes below the top"
```

Segment Protection Bits: R, W, X Flags

Each segment table entry carries **protection flags** that restrict what type of access is legal:

Flag	Meaning	Typical segments
R (Read)	Process may read from this segment	Code, Heap, Stack
W (Write)	Process may write to this segment	Heap, Stack
X (Execute)	Process may execute code from this segment	Code only

Example protections:

- Code segment: `R–X` — readable and executable, but **not writable** (prevents self-modification).
- Heap segment: `RW–` — readable and writable, but **not executable** (prevents stack/heap attacks).
- Stack segment: `RW–` — readable and writable, not executable.

Summary: Segmentation Translation Algorithm

Complete pseudocode for the hardware translation:

```
function translate(VA, access_type):
  seg    = extract_segment_bits(VA)   // top N bits
  offset = extract_offset_bits(VA)    // remaining bits

  entry  = segment_table[seg]

  if entry.valid == 0:
    raise SegmentNotPresentFault

  if entry.direction == NEGATIVE:
    neg_offset = offset - max_segment_size
    if abs(neg_offset) > entry.bounds:
      raise SegmentBoundsFault
    PA = entry.base + neg_offset

  else:  // POSITIVE growth
    if offset >= entry.bounds:
      raise SegmentBoundsFault
    PA = entry.base + offset

  if not check_protection(entry.prot, access_type):
    raise ProtectionFault

  return PA
```

✅ Checkpoint 3 — Segmentation: Address Translation

Test your understanding:

Setup: 14-bit VA space, 2-bit segment selector (top bits), 12-bit offset.
Segment table: Code (base=40KB, bounds=4KB, R-X), Heap (base=20KB, bounds=6KB, RW-), Stack (base=56KB, bounds=4KB, RW-, grows down, max=4KB).

1. Translate VA = `0x0080` (code access). Show all steps.
2. Translate VA = `0x1500` (heap write). Show all steps.
3. Translate VA = `0x3F00` (stack read). Use negative-offset formula.
4. VA = `0x0FFF` is a write to the code segment. What happens?
5. VA = `0x1FFF` — is this a valid heap access? What is the physical address?

Work through each translation step-by-step before checking.

§ 4 — OS Support for Segmentation

📎 *Extended material → Appendix A.4*

The OS's Five Segmentation Duties — At a Glance

#	Duty	What the OS does
1	**Process creation**	Find free physical regions for code, heap and stack; fill the segment table
2	**Process termination**	Return the three regions to the free list (coalescing neighbours)
3	**Context switch**	Save/restore the whole segment table (base, bounds, direction, protection)
4	**Segment growth**	Serve `sbrk()`: extend the heap if there is room, otherwise relocate or refuse
5	**Exceptions**	Handle segment-violation faults raised by the hardware (typically `SIGSEGV`)

Duties **3** and **4** are detailed next; duties **1, 2** and **5**, and the free-list algorithms behind them, are in Appendix A.4.

OS Duty 3: Context Switch — Saving/Restoring Segment Table

On every context switch, the full segment table must be swapped:

```
Switching from Process A to Process B:

SAVE Process A's segments (already in PCB — hardware registers → PCB):
  pcb_A->seg[CODE].base   = cpu_seg_reg[CODE].base
  pcb_A->seg[CODE].bounds = cpu_seg_reg[CODE].bounds
  ... (repeat for HEAP, STACK)

LOAD Process B's segments (PCB → hardware registers):
  cpu_seg_reg[CODE].base   = pcb_B->seg[CODE].base
  cpu_seg_reg[CODE].bounds = pcb_B->seg[CODE].bounds
  ... (repeat for HEAP, STACK)
```

All these moves happen in kernel mode. User code never executes between save and load.

The number of registers to save/restore is 3× larger than with base/bounds — a modest overhead.

OS Duty 4: Segment Growth — `sbrk()` and Heap Expansion

When a process calls `malloc()` and the heap is full, the C runtime calls `sbrk(increment)`:

```
sbrk(increment) system call:

1. User process: heap is full; C allocator calls sbrk(4096) (ask for 4 KB more).
2. Kernel receives sbrk system call.
3. OS checks: is there free physical memory adjacent to current heap end?
   → If yes: extend heap bounds by 4096 bytes in segment table.
             Update free list.
   → If no (non-adjacent): try to find a new larger block, copy heap, update base.
             If impossible: return -1 (out of memory).
4. OS updates seg_table[HEAP].bounds += 4096 in the PCB.
5. Returns new heap end address to the C runtime.
6. C runtime carves malloc blocks from the newly acquired space.
```

Free-Space Management: The External Fragmentation Problem

External fragmentation is the unavoidable consequence of variable-size allocation:

```
Physical memory after many process create/terminate cycles:

 ┌──────┬──────┬──────┬──────┬──────┬──────┬──────┬──────┬──────┬──────┬──────┐
 │  OS  │ FREE │      A      │ FREE │      B      │ FREE │      C      │ FREE │
 │ 8 KB │ 2 KB │     7 KB    │ 3 KB │     5 KB    │ 8 KB │     4 KB    │ 1 KB │
 └──────┴──────┴──────┴──────┴──────┴──────┴──────┴──────┴──────┴──────┴──────┘
 ┌──────┬──────┬──────┐
 │  D   │      FREE      │
 │ 3 KB │     23 KB      │
 └──────┴──────┴──────┘
0      8      10      17      20      25      33      37      38      41      64
         ↑               ↑               ↑               ↑               ↑
         └───────────────┘               └───────────────┘               └──────┘
                  37 KB FREE TOTAL

Free list: [(8KB, 2KB), (17KB, 3KB), (25KB, 8KB), (37KB, 1KB), (41KB, 23KB)]
Largest free region = 23 KB
Suppose a new process needs 25 KB contiguous → FAILS (largest = 23 KB)
```

Even though 37 KB is free, no 25 KB contiguous block exists. The OS must either refuse the allocation or perform **compaction**.

✅ Checkpoint 4 — OS Support for Segmentation

Test your understanding:

1. List the five duties the OS has under segmentation. Which two must it perform on *every* context switch and heap growth request, respectively?
2. What exactly must the OS save and restore on a context switch under segmentation, compared with base/bounds?
3. Explain what happens when `malloc()` exhausts the current heap and `sbrk()` is called. Trace the path from user space to the OS and back.
4. In the memory snapshot above, 37 KB are free but a 25 KB request fails. Why? What two options does the OS have?

Make sure you can answer 3 and 4 without looking.

§ 5 — Limitations of Segmentation

📎 *Extended material → Appendix A.5*

External Fragmentation: The Fundamental Unsolved Problem

Segmentation **eliminates** internal fragmentation (no heap-stack gap allocated) but **introduces** external fragmentation (variable-size allocations leave unusable holes):

> *"The general problem that arises is that the free space gets chopped up into little pieces and becomes fragmented; subsequent requests may fail even though the total amount of free memory is sufficient."*
> *— OSTEP Ch. 17*

This is not a bug in the OS design — it is an **inherent mathematical property** of variable-size allocation. Any scheme that allows allocations of different sizes will eventually fragment free space. The only question is *how fast* and *how badly*.

The Crux: We Need Fixed-Size Allocation Units

> *"How do we support a large address space with (potentially) a lot of free space between stack and heap, without all of the complexity of segmentation?"*
> *— OSTEP Ch. 18*

The answer: **stop allocating variable-sized chunks**. Instead, divide both virtual and physical memory into **fixed-size units**. Then:

- Any physical unit can satisfy any virtual request of the same size.
- No external fragmentation — ever.
- No need for compaction.
- No need to find large contiguous blocks.

These fixed-size units are called **pages** (virtual) and **frames** (physical).

✅ Checkpoint 5 — Limitations of Segmentation

Test your understanding:

1. Explain why external fragmentation is an *inherent* property of variable-size allocation, not a bug that can be fixed with a clever algorithm.
2. A memory system has 40 KB free in fragments of 8 KB, 12 KB, 6 KB and 14 KB. A new segment of 15 KB is requested. What are the options for the OS?
3. Why did base/bounds with equal-sized slots *not* suffer from external fragmentation? What did it suffer from instead?
4. What property of paging completely eliminates external fragmentation? Why doesn't segmentation have this property?
5. What key trade-off does paging introduce in exchange?

§ 6 — Introduction to Paging

📎 *Extended material → Appendix A.6*

The Paging Insight: Chop Everything into Fixed-Size Pieces

> *"Paging is the idea of dividing the virtual address space of a process into fixed-size units called pages, and dividing physical memory into fixed-size units of the same size called page frames."*

The two key decisions:

1. **Virtual address space:** divided into fixed-size **pages** (e.g., 4 KB each).
2. **Physical memory:** divided into fixed-size **frames** of the same size.

Since all units are the same size, any frame can hold any page. The OS maintains a **page table** — a per-process map from virtual page number to physical frame number. No contiguity required. No external fragmentation possible.

Definition: Pages and Frames

Term	Definition
Page	A fixed-size unit of the *virtual* address space
Frame	A fixed-size unit of *physical* memory (same size as a page)
Page Table	A per-process array mapping virtual page numbers (VPN) → physical frame numbers (PFN)
VPN	Virtual Page Number — the high-order bits of a virtual address
PFN / PPN	Physical Frame/Page Number — the high-order bits of a physical address
Offset	The low-order bits of an address — same in both VA and PA

The offset bits are **identical** in the virtual and physical address — they index into the page/frame. Only the high-order bits change during translation.

Why Fixed Size Solves External Fragmentation

The fundamental reason paging eliminates external fragmentation:

```
With variable-size segments:
  Segment A needs 7 KB → must find a contiguous 7 KB block.
  After A frees, the hole is 7 KB — only useful for requests ≤ 7 KB.
  Holes accumulate in various sizes → external fragmentation.

With fixed-size pages (4 KB each):
  Process needs 7 KB → allocate 2 pages (8 KB, slight internal fragmentation).
  Each page is placed in any free frame — no contiguity needed.
  After freeing: 2 free frames, each usable by any future 4 KB request.
  No unusable holes — every free frame is equally useful.
```

The price paid: a small amount of **internal fragmentation** (up to one page per allocation), but this is bounded and manageable.

Diagram: Virtual Address Space Divided into Pages

A 64-byte virtual address space with 16-byte pages (OSTEP's running example):

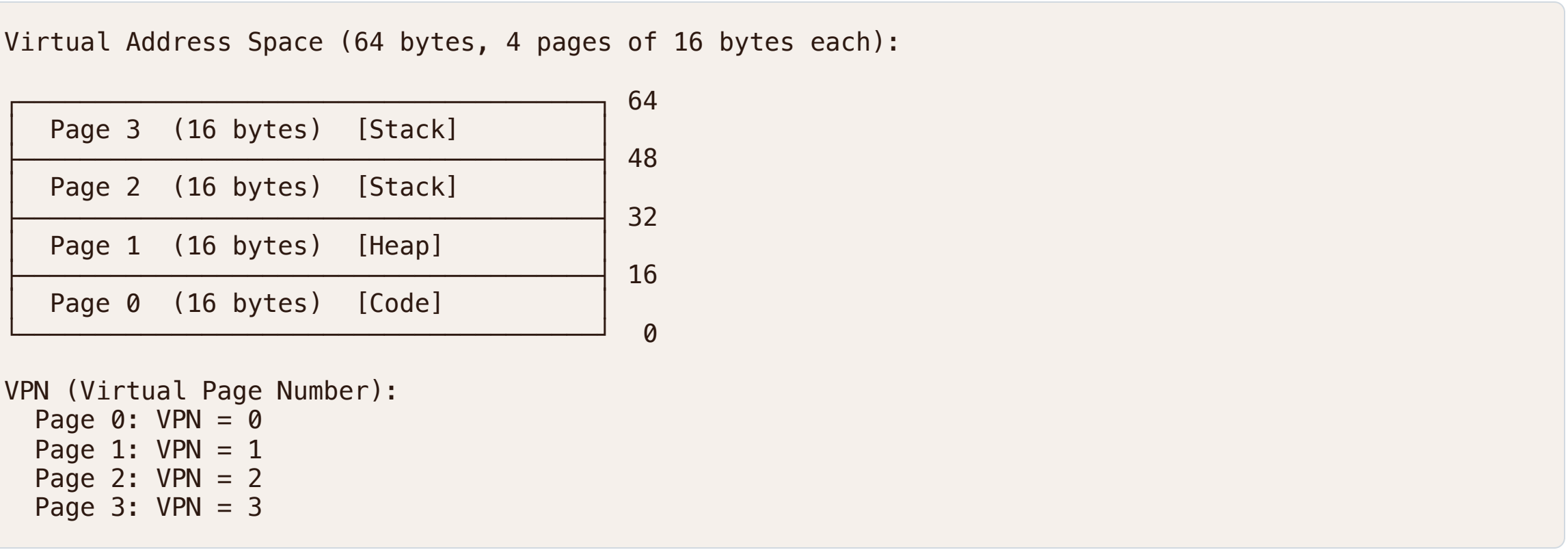
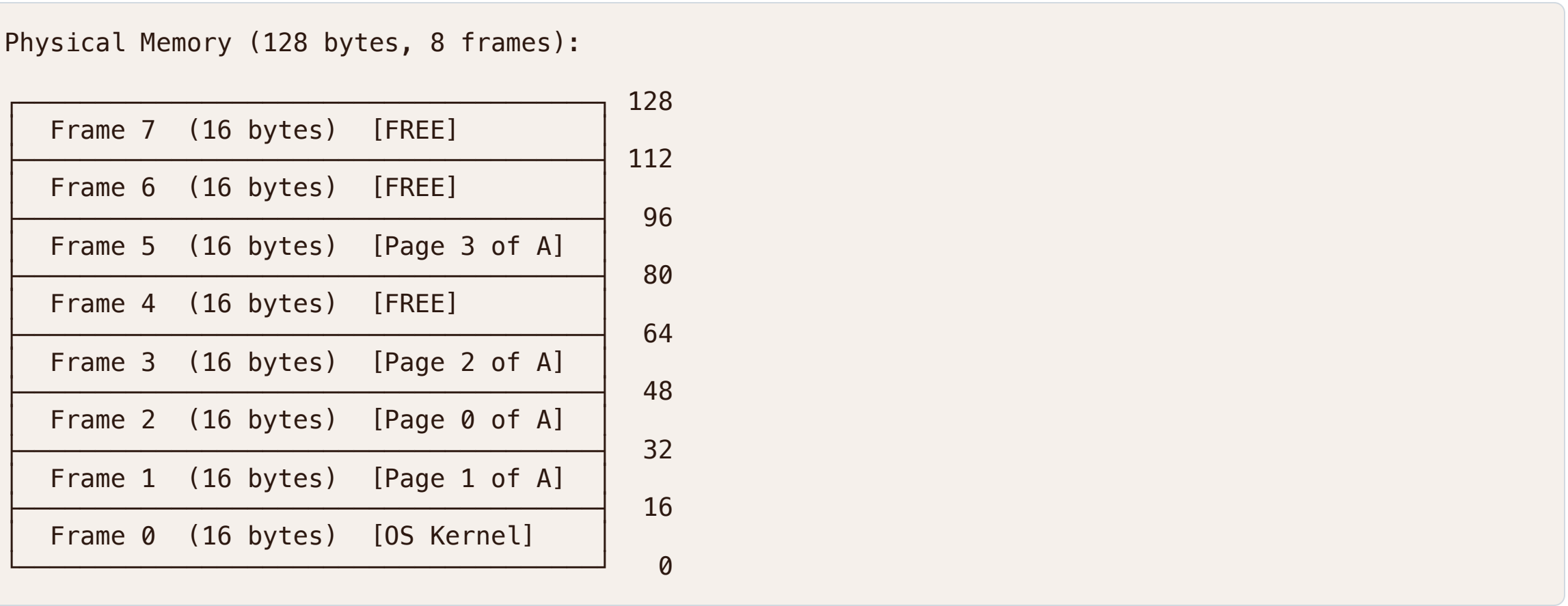


Diagram: Physical Memory Divided into Frames

Physical memory (128 bytes, 8 frames of 16 bytes each):



Process A's 4 pages are scattered across frames 1, 2, 3, 5 — **no contiguity required**.

The Page Table: Mapping VPN to PFN

The **page table** is a per-process array indexed by VPN, storing the PFN and flags:

```
Page Table for Process A (from the diagram above):

┌─────┬──────────────────────────────────┐
│ VPN │   Page Table Entry (PTE)         │
├─────┼──────────────────────────────────┤
│  0  │  PFN = 2,  valid=1, prot=R-X     │   (Code: VPN 0 → Frame 2)
│  1  │  PFN = 1,  valid=1, prot=RW-     │   (Heap: VPN 1 → Frame 1)
│  2  │  PFN = 3,  valid=1, prot=RW-     │   (Stack: VPN 2 → Frame 3)
│  3  │  PFN = 5,  valid=1, prot=RW-     │   (Stack: VPN 3 → Frame 5)
└─────┴──────────────────────────────────┘
```

To translate: look up the row for the VPN, extract the PFN, concatenate with the offset.

Address Translation with Paging: VPN and Offset

A virtual address is split into two parts:

```
Virtual Address (N bits total):

┌──────────────────────┬──────────────────────┐
│  VPN                 │       Offset         │
│  (top N−k bits)      │   (bottom k bits)    │
└──────────────────────┴──────────────────────┘

where:
  k = log₂(page_size)         (number of offset bits)
  N − k = log₂(num_pages)     (number of VPN bits)

For 64-byte VA with 16-byte pages (OSTEP example):
  k = log₂(16) = 4 bits       → offset field = 4 bits
  VPN = 6 − 4  = 2 bits       → up to 2^2 = 4 pages

For 32-bit VA with 4 KB pages (more realistic):
  k = log₂(4096) = 12 bits    → offset field = 12 bits
  VPN = 32 − 12  = 20 bits    → up to 2^20 = 1,048,576 pages
```

The Translation Formula: PA = PFN × page_size + offset

The physical address is constructed by combining the PFN from the page table with the offset from the virtual address:

$$\text{PA} = \text{PFN} \times \text{page_size} + \text{offset}$$

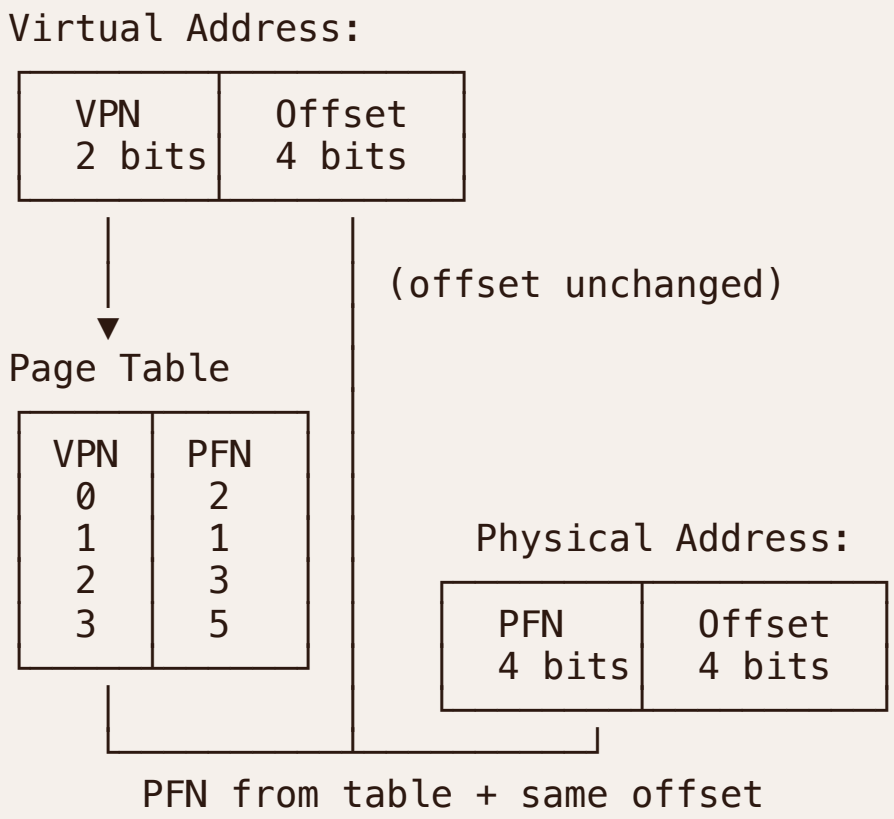
Or equivalently (using bit concatenation, which is how hardware does it):

$$\text{PA} = (\text{PFN} \,\|\, \text{offset})$$

where $\|$ denotes bit concatenation — the PFN bits become the high-order bits and the offset bits stay unchanged as the low-order bits.

This is a key observation: the **offset is preserved** unchanged through the translation. Only the page (VPN → PFN) changes.

Diagram: The Full Paging Translation Pipeline



The Time Cost of Paging: Two Memory Accesses

Paging introduces a time overhead: **every memory access requires two memory accesses**:

```
Without paging (base/bounds):
  Access #1: read/write the actual data → 1 memory access total

With paging (naive — no TLB):
  Access #1: read the PTE from the page table in RAM → 1 memory access
  Access #2: read/write the actual data using the PA → 1 memory access
  Total: 2 memory accesses per operation = 2× slower!
```

This 2× slowdown would be catastrophic in practice. The solution is a hardware cache for page table entries: the **Translation Lookaside Buffer (TLB)**, which we study in Lecture 2.6.

✅ Checkpoint 6 — Introduction to Paging

Test your understanding:

1. A 32-bit address space uses 4 KB pages. How many bits are in the VPN? How many in the offset?
2. Translate VA = `0x00005ABC` with page size = 4 KB and page table entry at VPN 5 giving PFN = 10.
3. A page table entry has valid = 0. What happens when the hardware tries to use it?
4. Calculate the size of a flat page table for a 32-bit address space with 4 KB pages and 4-byte PTEs. Is this feasible?
5. Why does naive paging require two memory accesses per data access? What hardware structure solves this?

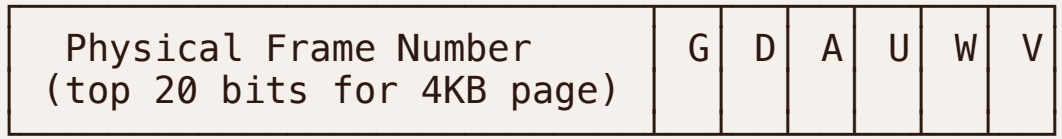
§ 7 — Paging: Address Translation in Detail

📎 *Extended material → Appendix A.7*

Page Table Entry (PTE): Fields

A page table entry encodes more than just the PFN:

```
Typical 32-bit PTE:
```




```
  V = Valid bit     (1 = page in physical RAM; 0 = not present → page fault)
  W = Writable bit  (1 = page is writable)
  U = User bit      (1 = accessible in user mode; 0 = kernel only)
  A = Accessed bit  (1 = page has been read recently; used by page replacement)
  D = Dirty bit     (1 = page has been written; used by page replacement)
  G = Global bit    (1 = shared across all processes, e.g., kernel pages)
```

Page Table Size Calculation

How large is a page table? Let's calculate for a 32-bit address space with 4 KB pages:

```
Number of pages = 2^32 / 2^12 = 2^20 = 1,048,576 pages

Each PTE = 4 bytes (32-bit PTE: 20-bit PFN + 12 bits for flags)

Page table size = 1,048,576 × 4 bytes = 4,194,304 bytes = 4 MB

For 100 processes: 100 × 4 MB = 400 MB just for page tables!
```

A 4 MB page table per process is manageable on 32-bit systems, but consider 64-bit:

```
64-bit VA space (48 bits used): 2^48 / 2^12 = 2^36 ≈ 68 billion pages
Each PTE = 8 bytes
Page table size = 2^36 × 8 = 512 GB per process → IMPOSSIBLE
```

This is why **multi-level page tables** (Lecture 2.6) are essential for 64-bit systems.

Full Worked Example: 4 KB Pages, Realistic Addresses

Setup: 32-bit VA space, 4 KB pages, page table shown below.

```
Page table (selected entries):
  VPN  0: PFN = 100, valid=1, prot=R-X
  VPN  1: PFN = 200, valid=1, prot=RW-
  VPN  2: PFN = 0,   valid=0             ← not present!
  VPN  3: PFN = 150, valid=1, prot=RW-
```

Translate VA = 0x00001204:

```
VPN = 0x00001204 >> 12 = 1,  Offset = 0x204 = 516
PFN = page_table[1].pfn = 200,  valid=1 ✅,  prot=RW-
Access type: write ✅ (W bit set)
PA = 200 × 4096 + 516 = 819,200 + 516 = 819,716 = 0xC8204
```

Translate VA = 0x00002010:

```
VPN = 2,  Offset = 0x010 = 16
PTE: valid = 0 → PAGE FAULT
OS page fault handler invoked.
```

✅ Checkpoint 7 — Paging: Address Translation in Detail

Test your understanding:

Setup: 32-bit address space, 4 KB pages. Page table: VPN 0→PFN 5, VPN 1→PFN 10 (valid), VPN 2→invalid, VPN 3→PFN 8 (read-only).

1. Translate VA = `0x00000A10`. Show VPN, offset, PFN, and PA.
2. Translate VA = `0x00003050`. Show all steps. What happens?
3. Process tries to write to VA = `0x00003004`. VPN 3 is read-only. What happens?
4. How many bits are in the VPN and offset for a 32-bit VA with 4 KB pages?
5. A process has 2 active pages out of a 32-bit address space. How much physical memory does the OS waste storing its flat page table?

§ 8 — Comparison & Summary

📎 *Extended material → Appendix A.8*

Side-by-Side Comparison: Base/Bounds, Segmentation, Paging

Property	Base/Bounds	Segmentation	Paging
Allocation unit	One fixed-size slot (entire AS)	Variable-size segments	Fixed-size pages
Internal fragmentation	High (heap-stack gap)	None (only active segments)	Low (≤ 1 page per segment)
External fragmentation	**No** (equal-sized slots)	**Yes** (variable-size segments)	**No**
Code sharing	No	Yes (per-segment)	Yes (per-page)
Address space > RAM	No	No	**Yes** (with swapping)
Translation cost	1 addition + 1 check	1 addition + 2 checks	1 lookup + 1 addition + 2 checks
OS data structure	1 pair per process	Segment table (3+ entries)	Page table (millions of entries)
Hardware required	Base/bounds registers	Segment registers	MMU + PTBR + TLB

Summary: Chapter 16 — Segmentation

Core mechanism:

- One base/bounds pair **per segment** (code, heap, stack).
- Translation: `PA = base[seg] + offset`.
- Bounds check: `offset < bounds[seg]`, with direction for downward-growing stack.
- Protection: per-segment R/W/X bits.
- Code sharing: multiple processes can map their code segment to the same physical frames.

OS responsibilities: allocate/free 3 regions per process; save/restore full segment table on context switch; handle `sbrk()` for heap growth; compact to fight external fragmentation.

Limitation: external fragmentation is inherent and unsolvable without compaction.

Summary: Chapter 18 — Introduction to Paging

Core mechanism:

- Virtual address space and physical memory both divided into **fixed-size pages/frames** (typically 4 KB).
- **Page table** per process: array indexed by VPN, storing PFN + flags.
- Translation: `PA = PFN × page_size + offset` (offset unchanged).
- Bounds/protection: valid bit, R/W/X/User bits per PTE.

Advantages: no external fragmentation; any frame serves any page; fine-grained protection and sharing; foundation for swapping.

Current costs:

- **Space:** flat page table = 4 MB per 32-bit process (impossible for 64-bit).
- **Time:** two memory accesses per access (page table lookup + data access).

Both costs are addressed in Lecture 2.6 (TLB + multi-level page tables).

Recommended Exercises

Before Lecture 2.6, practice:

1. **Segmentation:** Segment table: Code (base=64KB, bounds=8KB, R-X), Heap (base=16KB, bounds=4KB, RW-), Stack (base=80KB, bounds=4KB, RW-, grows down). Translate: VA=`0x0040`, VA=`0x1010`, VA=`0x3F00`, VA=`0x2FFF` (heap write). Identify faults.
2. **Segmentation — stack:** Using the table above, compute the physical address for a stack access at VA=`0x3800`. Use the negative-offset formula.
3. **Paging:** 32-bit VA, 4 KB pages. Page table: VPN 0→PFN 3, VPN 1→PFN 7, VPN 2→invalid, VPN 4→PFN 2 (read-only). Translate VA=`0x00001080`, VA=`0x00002010` (write), VA=`0x00004001`.
4. **Page table size:** Calculate the flat page table size for a 48-bit address space with 4 KB pages and 8-byte PTEs. Why is this impossible to store in RAM?
5. **Comparison:** Fill in this table for base/bounds, segmentation, and paging: internal fragmentation, external fragmentation, code sharing, space overhead, time overhead.

Lecture 2.5 Complete

Recommended Reading

- **OSTEP Ch. 16:** *Segmentation*
- **OSTEP Ch. 17:** *Free-Space Management*
- **OSTEP Ch. 18:** *Introduction to Paging*
- 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	Fragmentation recap; the natural structure of a process	2
A.2	§ 2	Placement of segments, selectors, implicit addressing, the segment-table structure	5
A.3	§ 3	Heap and stack worked examples, SP/BP vs. base, protection checks, code sharing	11
A.4	§ 4	OS actions 1, 2, 5; free list, allocation policies, buddy system; Chapter 17 summary	8
A.5	§ 5	Compaction cost, free-list complexity, sparse spaces, x86 history	7
A.6	§ 6	Page-size trade-offs, toy examples, valid-bit semantics	4
A.7	§ 7	PTE flags, table sizes, PTBR, VPN/offset procedure, protection, OSTEP example	11
A.8	§ 8	When to use which, hybrid segmented paging, key formulas, self-check bank	6

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

📎 A.1 — Recap & Motivation

A.1.1 · The Two Fragmentation Problems — Recap

Internal fragmentation: wasted space *inside* an allocated region.

```
Process (base/bounds, 16 KB allocated):
  Code:   2 KB  ← in use
  Data:   1 KB  ← in use
  Gap:   12 KB  ← physical RAM allocated but EMPTY (heap/stack haven't grown yet)
  Stack:  1 KB  ← in use
  Wasted: 12 KB of 16 KB = 75% waste!
```

External fragmentation: free space exists but is too scattered for new allocations. It needs **variable-sized** allocations, e.g. segments:

```
Physical RAM: [OS][FREE 4KB][Seg 16KB][FREE 8KB][Seg 16KB][FREE 20KB]
New segment needs 24 KB contiguous → FAILS (largest free = 20 KB)
Total free = 32 KB, but no contiguous 24 KB block exists.
```

With base/bounds and equal-sized slots every free slot fits every process, so this cannot happen: base/bounds suffers only from internal fragmentation.

A.1.2 · The Natural Structure of a Process

A process uses three structurally distinct memory regions with very different behaviors:

Segment	Growth	Permissions	Typical Size
Code	Fixed (no growth)	Read + Execute	Kilobytes to megabytes
Heap	Grows upward ↑	Read + Write	Zero to gigabytes
Stack	Grows downward ↓	Read + Write	Kilobytes to megabytes

Each segment can be placed **independently** anywhere in physical memory. The OS only needs to find space for the *used* portion of each segment — not for the vast virtual gap between heap top and stack bottom.

📎 A.2 — Segmentation: The Core Idea

A.2.1 · How the OS Places Each Segment Independently

On process creation, the OS performs three independent allocations:

```
1. Code segment (2 KB):
   → Search free list for a 2 KB block
   → Found at physical 32 KB
   → Set segment_table[CODE].base   = 32 KB
   → Set segment_table[CODE].bounds = 2 KB

2. Heap segment (3 KB):
   → Search free list for a 3 KB block
   → Found at physical 36 KB
   → Set segment_table[HEAP].base   = 36 KB
   → Set segment_table[HEAP].bounds = 3 KB

3. Stack segment (2 KB):
   → Search free list for a 2 KB block
   → Found at physical 54 KB
   → Set segment_table[STACK].base  = 54 KB
   → Set segment_table[STACK].bounds = 2 KB
```

Three separate free-list allocations — completely independent.

A.2.2 · Only Active Segments Occupy Physical RAM

The crucial advantage over base/bounds:

```
Base/Bounds (Lecture 2.4):
  OS allocates 16 KB physical for the entire virtual range.
  7 KB used + 9 KB wasted (heap-stack gap).
  Physical waste: 56%.

Segmentation (this lecture):
  OS allocates only:
    2 KB (code) + 3 KB (heap) + 2 KB (stack) = 7 KB physical.
  Physical waste: 0% (no gap allocated).
```

As the heap grows (more `malloc()` calls), the OS extends the heap segment's bounds and finds a larger physical block. The stack grows similarly. Only *actual growth* consumes more physical RAM.

A.2.3 · The Segment Register / Segment Selector

The hardware must know, for each memory access, **which segment** the virtual address belongs to.

The approach depends on architecture:

Explicit Segmentation (most common in OS textbooks)

The top bits of the virtual address encode the **segment number**. The hardware extracts these bits to index into the segment table.

Implicit Segmentation

The hardware infers the segment from context:

- Instruction fetch → always Code segment.
- Stack pointer (`%rsp`) references → always Stack segment.
- All other data references → Heap/Data segment.

Intel x86-16/32 used explicit segment registers (`CS`, `DS`, `SS`, `ES`). Modern 64-bit systems largely abandon hardware segmentation in favor of paging.

A.2.4 · Implicit Approach: Infer from Context

In implicit segmentation, the hardware uses context clues:

```
CPU is fetching an instruction:
  → The address came from the PC register
  → Must be in the Code segment
  → Use segment_table[CODE]

CPU is executing PUSH/POP (stack instruction):
  → The address came from the Stack Pointer register
  → Must be in the Stack segment
  → Use segment_table[STACK]

CPU is executing a data load/store:
  → The address is a data reference
  → Must be in the Heap/Data segment
  → Use segment_table[HEAP]
```

Implicit segmentation requires no bits stolen from the virtual address, but is less flexible.

A.2.5 · The Segment Table Data Structure

In hardware and OS, the segment table is a small array of descriptors:

```
// One entry per segment:
struct segment_descriptor {
    uint64_t base;       // physical start address of this segment
    uint64_t bounds;     // size (bytes) of this segment
    uint8_t  prot;       // protection: R=4, W=2, X=1 (bitmask)
    uint8_t  direction;  // 0 = grows up (code/heap), 1 = grows down (stack)
    uint8_t  valid;      // 1 = segment in use, 0 = not present
};

// Segment table for one process (3 segments):
struct segment_descriptor seg_table[3];  // indexed by segment number
```

The OS stores this table in the PCB. At context switch time, the OS loads it into the hardware segment registers (or a dedicated hardware table pointer register).

📎 A.3 — Segmentation: Address Translation in Detail

A.3.1 · Worked Example: Heap Segment Access

Setup: same segment table as previous slide.

Access: `movl %eax, 0x1500` — store to virtual address `0x1500` (= 5,376 decimal).

```
1. seg    = 0x1500 >> 12 = 0b01 → Heap segment
2. offset = 0x1500 & 0x0FFF = 0x500 = 1,280
3. Bounds check: 1,280 < 3,072 ✅
4. Protection check: write = W, prot = RW- ✅
5. PA = 34,816 + 1,280 = 36,096

→ CPU stores to physical address 36,096.
```

Fault example: If we try to *execute* from the heap (`prot = RW-`, no X bit):

```
  Access type = Execute, prot = RW- (no X) → Protection Fault!
  OS delivers SIGSEGV. Prevents data-segment execution attacks.
```

A.3.2 · Stack Segment Table Entry: base, bounds and direction

Segment table entry for the stack:

Field	Value	Meaning
`base`	physical address of the **top** of the stack segment	corresponds to the *highest* virtual address of the segment (virtual 16 KB); the stack's actual bytes live *below* this address
`bounds`	e.g. 2 KB	*maximum* stack size — also the furthest the stack can extend *below* `base`
`direction`	`1` (down)	tells hardware to flip the offset calculation

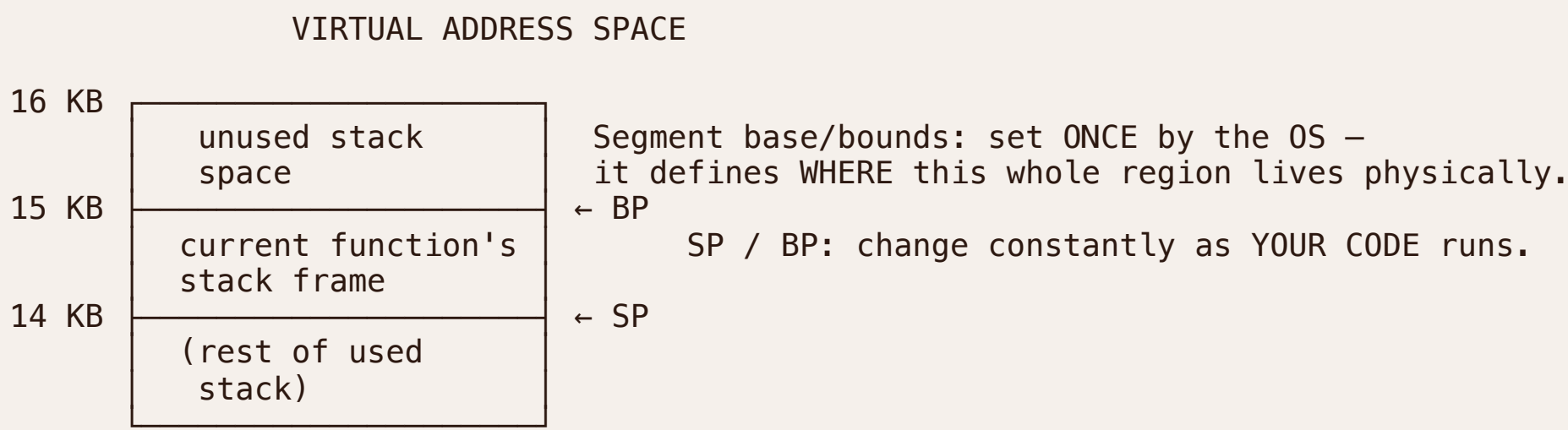
> *Because the stack grows down, an address's offset must be measured from the **top** of the segment, not the bottom.*

A.3.3 · ⚠️ Don't confuse: Segment `base` vs. SP/BP registers

There are **two unrelated things** both called "base" here — keep them separate:

	Segment `base` / `bounds`	Stack Pointer (SP) / Base Pointer (BP)
Layer	Hardware/OS — MMU translation	CPU registers — used by *your* running code
Set by	OS, once, at process/segment setup	Compiler-generated code, constantly (`push` / `pop` , prologue/epilogue)
Purpose	Maps virtual segment → physical memory	Track current stack top (SP) and current frame's anchor (BP)
Visibility	Invisible to the running program	Directly visible/used in assembly
Changes how often?	Fixed once allocated	Every function call / push / pop

A.3.4 · Segment `base` vs. SP/BP registers



SP and BP are just *virtual addresses* — like any pointer — that happen to fall inside the stack segment. They get **translated** using the segment's `base` / `bounds`, exactly like any other stack access would. The segment registers don't know or care whether a translated address came from SP, BP, or `array[i]`.

A.3.5 · Stack Worked Example: Step by Step

Example: Step by step

```
Given:
  Stack segment: base = 28,672 (28 KB), bounds = 2,048, direction = DOWN
  Virtual address accessed: 0x3C00 = 15,360

1. Extract segment bits:
     seg = 15,360 >> 12 = 0b11 → Stack segment ✓

2. Extract raw offset:
     offset = 15,360 & 0x0FFF = 0xC00 = 3,072

3. Since direction = DOWN, convert to a negative offset:
     negative_offset = offset - max_size = 3,072 - 4,096 = -1,024

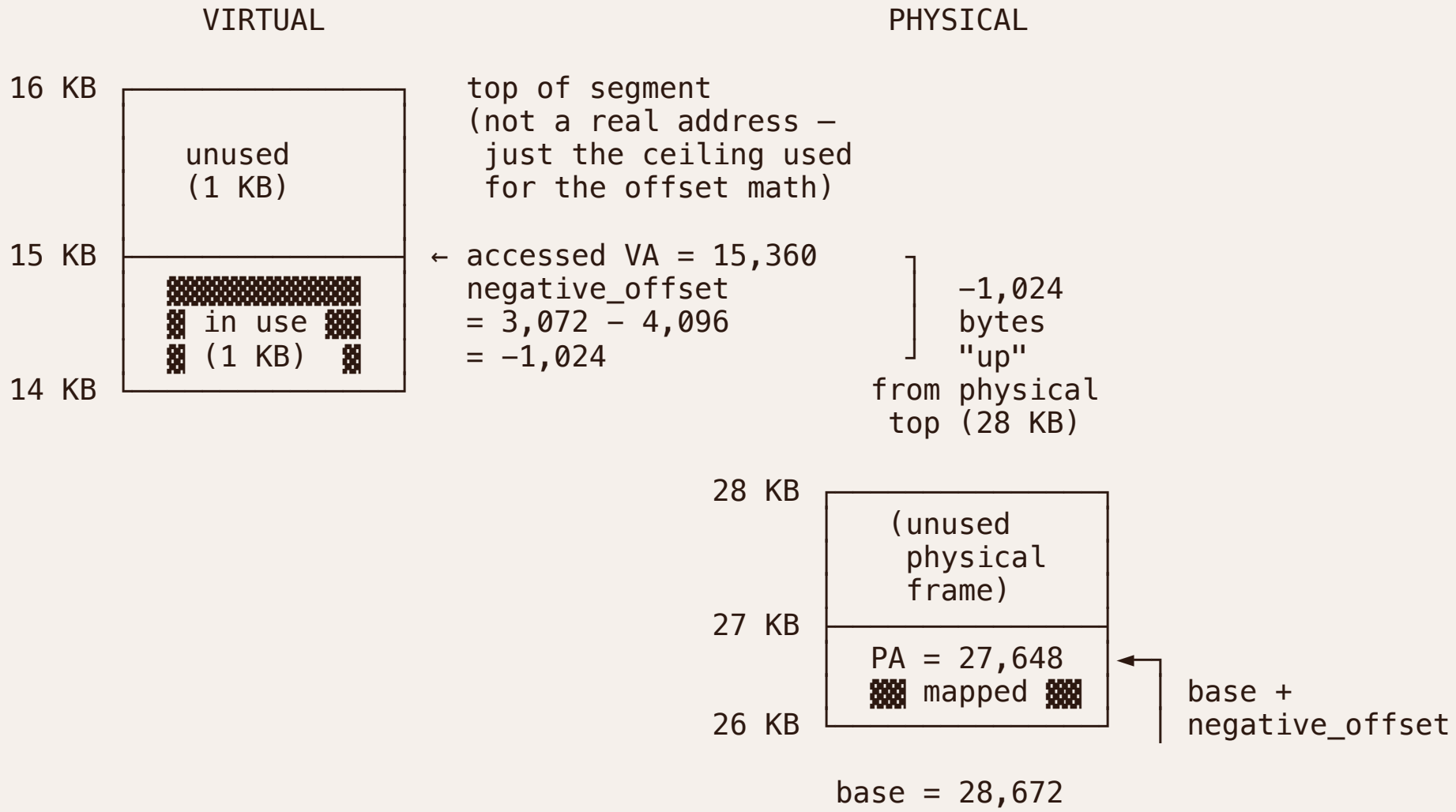
4. Bounds check (compare the *magnitude* against bounds):
     |negative_offset| = 1,024  ≤  bounds (2,048)   ✅ valid

5. Translate:
     PA = base + negative_offset = 28,672 + (-1,024) = 27,648
```

Physical address = 27,648 (27 KB)

> *Intuition: `PA = base − (max_size − offset)`. You're walking backward from the top of the region by however far the access is below that top.*

A.3.6 · Stack Worked Example — Visual Recap



A.3.7 · Stack Worked Example — Summary of the Math

```
segment          = 3            (Stack)
offset           = 3,072
max_size         = 4,096        (2^12, size of the 12-bit offset field)
negative_offset  = 3,072 − 4,096 = −1,024
bounds check      1,024 ≤ 2,048  ✅
PA               = base + negative_offset
                 = 28,672 + (−1,024)
                 = 27,648

Result: virtual 15 KB  →  physical 27 KB (27,648 bytes)
```

A.3.8 · Hardware Protection Check: Enforcing RWX

On every memory access, after the bounds check, the MMU enforces protection:

```
Access types and required flags:
  Instruction fetch    → requires X bit set
  Data load (read)     → requires R bit set
  Data store (write)   → requires W bit set

Protection check algorithm:
  if (access_type == EXECUTE  && !(seg_entry.prot & X)) → fault
  if (access_type == LOAD     && !(seg_entry.prot & R)) → fault
  if (access_type == STORE    && !(seg_entry.prot & W)) → fault

On fault: CPU raises "General Protection Fault" or "Segmentation Fault"
          OS handler terminates the offending process (SIGSEGV)
```

Protection bits are **privileged** — only the OS (kernel mode) can set or change them.

A.3.9 · Code Sharing Between Processes

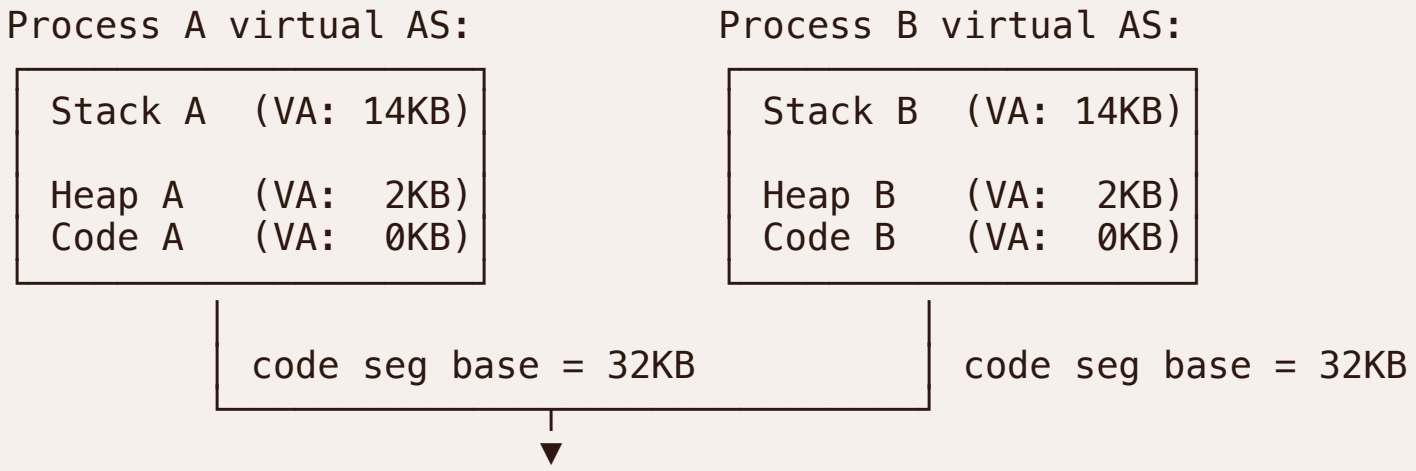
Segmentation enables **read-only code sharing**: multiple processes can map their code segment to the **same physical memory**:

```
Process A segment table:  Code → base = 32 KB, bounds = 2 KB, prot = R-X
Process B segment table:  Code → base = 32 KB, bounds = 2 KB, prot = R-X
                                        ↑
                          SAME physical base!

Physical Memory:
  32 KB: [shared code for both A and B]
  34 KB: [Heap A]
  36 KB: [Stack A]
  38 KB: [Heap B]
  40 KB: [Stack B]
```

The code is stored **once** in physical RAM but appears in both processes' address spaces. Since both mappings are `R-X` (no write), neither process can corrupt the shared code. This is how `bash`, `python`, or any commonly-run program saves physical RAM.

A.3.10 · Diagram: Two Processes Sharing a Code Segment



Physical savings: one copy of code instead of two.
If 100 bash processes are running: code loaded ONCE, not 100 times.

A.3.11 · Full Segment Table with Protection Bits

A complete segment table entry following OSTEP Table 16.1:

Segment	Base (phys)	Bounds	Grows	Prot	Valid
Code	32 KB	2 KB	Positive	R-X	1
Heap	34 KB	3 KB	Positive	RW-	1
Stack	28 KB	2 KB	**Negative**	RW-	1

Key fields:

- **Grows:** direction of growth (positive = upward, negative = downward).
- **Valid:** 1 if the segment is present in physical memory; 0 if not allocated (access → fault).
- **Prot:** bitmask of Read/Write/Execute permissions.

📎 A.4 — OS Support for Segmentation

A.4.1 · OS Action 1: Process Creation

When the OS creates a new process with segmentation:

```
1. Parse ELF binary headers to determine:
   - Code segment size (from .text + .rodata sections)
   - Initial data segment size (from .data + .bss sections)
   - Initial stack size (OS policy, e.g., 8 KB)
   - Initial heap size (0 — grows on first malloc)

2. For each segment:
   a. Search the physical memory free list for a contiguous block of the required size.
   b. If found: remove from free list; record (base, bounds) in segment table entry.
   c. If not found: report out-of-memory error; process creation fails.

3. Load code and data segments from disk into their physical locations.

4. Set stack segment to initial size (zero-filled).

5. Store segment table in the PCB. Place process in READY queue.
```

A.4.2 · OS Action 2: Process Termination

When a process exits or is killed, the OS must reclaim **all three segments** independently:

```
On process termination:

1. Read segment table from PCB:
   Code:  base=32KB, bounds=2KB  → free_list.add(32KB, 2KB)
   Heap:  base=34KB, bounds=3KB  → free_list.add(34KB, 3KB)
   Stack: base=28KB, bounds=2KB  → free_list.add(28KB, 2KB)

2. For each freed region, attempt to coalesce with adjacent free blocks:
   If (28KB, 2KB) is adjacent to (30KB, 2KB) already free:
   → merge into (28KB, 4KB)

3. Destroy PCB, close file descriptors, send SIGCHLD to parent.
```

Three separate free-list insertions — one per segment. Each can be coalesced independently.

A.4.3 · OS Action 5: Exception Handling — Segment Violation Faults

When the MMU raises a segmentation fault:

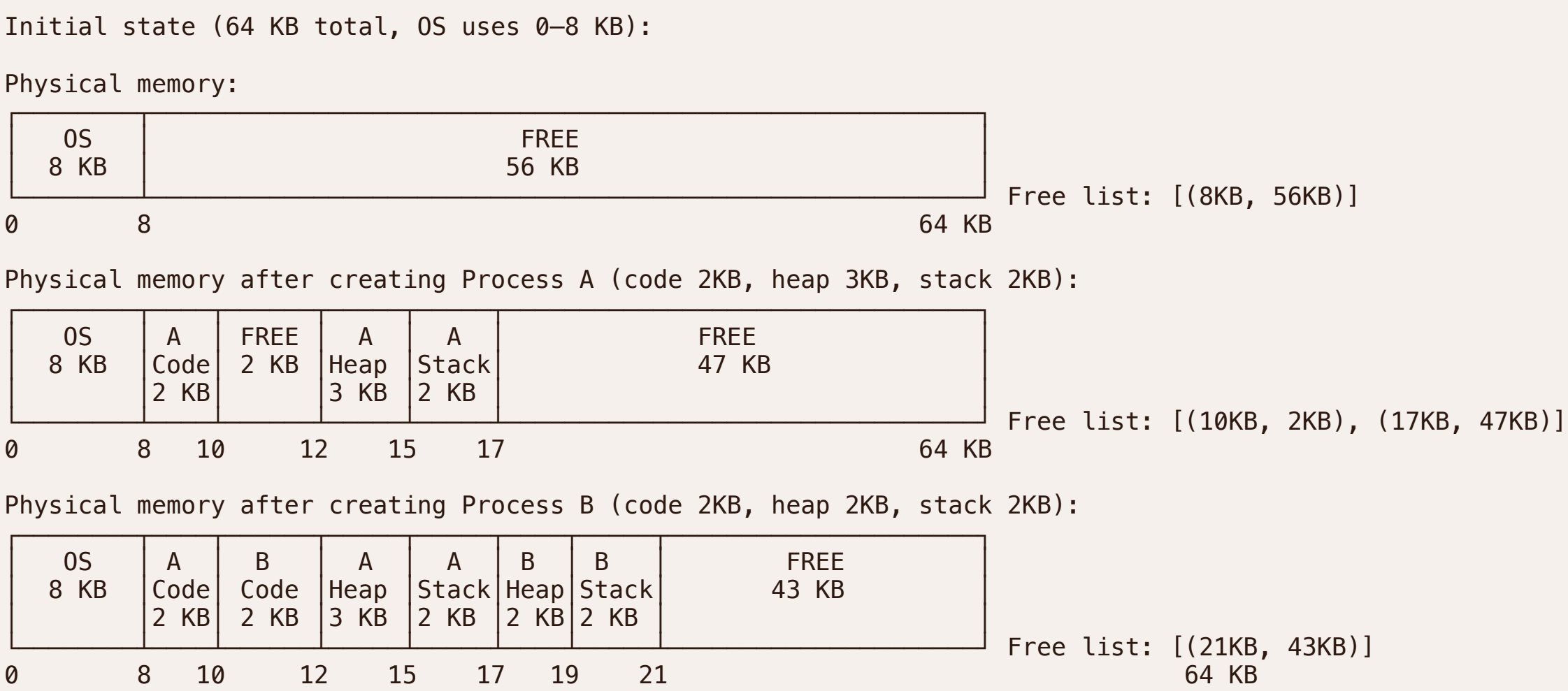
```
Types of segmentation faults:
  1. SegmentBoundsFault   — offset ≥ bounds (out of range)
  2. ProtectionFault      — wrong access type (e.g., write to R-X segment)
  3. SegmentNotPresent    — valid bit = 0 (segment not loaded)

OS exception handler:
  1. CPU switches to kernel mode, jumps to registered fault handler.
  2. Handler reads fault address and fault type from hardware registers.
  3. Identifies the offending process (from current PCB).
  4. Sends SIGSEGV signal to the process (default action: terminate + core dump).
  5. OS cleans up the process (frees all three segments, destroys PCB).
  6. Returns to scheduling loop.

The rest of the system continues running unaffected.
```

A.4.4 · The Free List with Variable-Sized Regions

With segmentation, the OS free list holds variable-sized entries, unlike base/bounds where all slots were the same size:



A.4.5 · Allocation Algorithms: Best-Fit, Worst-Fit, First-Fit

When the OS allocates from a fragmented free list, different policies behave differently:

First-Fit

Scan from the beginning; use the **first** free block that is large enough.

- Fast (no full scan needed).
- Leaves small fragments near the head of the list.

Best-Fit

Find the **smallest** free block that still fits.

- Minimizes leftover fragment per allocation.
- Leaves many tiny, unusable fragments over time.

Worst-Fit

Use the **largest** free block.

- Leaves larger residual fragments that future allocations can use.

A.4.6 · The Buddy System

A practical compromise for managing variable-size free lists:

```
Buddy system: always allocate power-of-two sizes.

Free list organized by size class (each a power of 2):
  [32B]: ——
  [64B]: [block1] → [block2]
  [128B]: [block3]
  [256B]: ——
  [512B]: [block4]
  ...

On allocation of N bytes:
  Round N up to next power of 2.
  Find the smallest size class ≥ N.
  If empty: split a block from the next larger class (creating two "buddies").
  Return one buddy; put the other in the smaller class.

On free:
  Check if the buddy (adjacent same-size block) is also free.
  If yes: merge into one larger block (coalesce upward recursively).
```

The buddy system makes coalescing O(log N) — far better than scanning the full free list.

A.4.7 · OS Responsibilities Summary: Segmentation

Event	OS Action
Process creation	Allocate 3 separate physical regions; initialize segment table
Process termination	Free 3 regions; coalesce free list
Context switch	Save current seg table to PCB; load new process's seg table
`sbrk()` / heap grow	Extend heap segment bounds; update free list
Seg fault	Identify violating process; deliver SIGSEGV; terminate & clean up
Memory pressure	Run compaction to merge free space (expensive)

A.4.8 · Summary: Chapter 17 — Free-Space Management

The problem: variable-size allocations and deallocations create a fragmented free list.

Key algorithms:

- **First-fit:** fast; leaves small fragments at the front.
- **Best-fit:** minimal per-allocation waste; creates many tiny fragments.
- **Worst-fit:** leaves larger residuals; generally performs poorly.
- **Buddy system:** power-of-two sizes; O(log N) coalescing.

External fragmentation is unavoidable with variable-size allocation. Compaction is the last resort: correct but expensive (copy + update + pause).

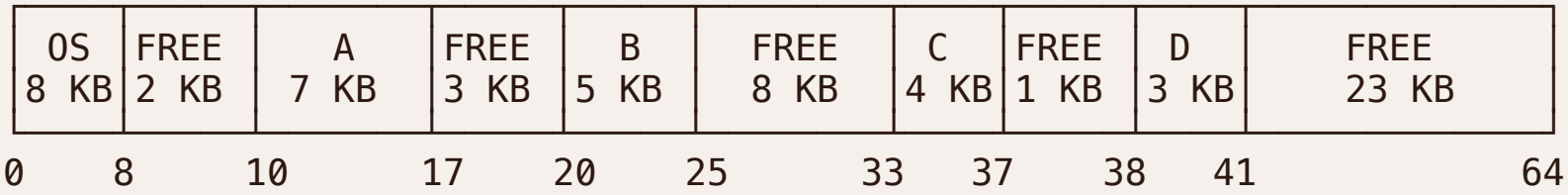
The solution: eliminate variable-size allocation → paging.

📎 A.5 — Limitations of Segmentation

A.5.1 · Compaction: Moving Segments to Merge Free Space

The OS can **compact** memory by relocating active segments:

```
Before compaction:
```




```
OS compacts (moves A, B, C, and D together):
1. Copy segment A immediately after OS.
2. Update seg_table_A[CODE].base (and other affected segments).
3. Copy segment B immediately after A.
4. Update seg_table_B[CODE].base.
5. Copy segment C immediately after B.
6. Update seg_table_C[CODE].base.
6. Copy segment D immediately after C.
7. Update seg_table_D[CODE].base.

After compaction:
```



A.5.2 · Why Compaction Is Expensive (1/2): Costs

Compaction is logically correct but has significant costs:

- **Memory bandwidth:** Moving large segments requires copying potentially gigabytes of data.
- **CPU time:** Compaction must be done with the process **stopped** (or with careful synchronization).
- **Downtime:** Interactive processes experience latency spikes while their memory is being copied.
- **Frequency:** In a busy system, compaction may be needed very often.

Compaction is the OS equivalent of defragmenting a hard disk — correct, but you don't want to do it often.

This is why designers sought a mechanism that **avoids external fragmentation altogether** — leading eventually to paging.

A.5.3 · Why Compaction Is Expensive (2/2): The Options When Allocation Fails

Every time external fragmentation blocks a new allocation, the OS must choose:

- **Fail** the allocation (return `NULL` from `malloc()` — bad for application).
- **Compact** memory (expensive — copy many megabytes, pause processes).
- **Wait** until more processes terminate and free space coalesces naturally.

None of these options is satisfactory for a high-performance, always-on system. The ideal solution would **never create external fragmentation** in the first place. This requires allocating in **fixed-size units**.

A.5.4 · Variable-Sized Segments Complicate Free-List Management

With variable-size segments, the OS free-list data structure itself becomes complex:

- The free list must be sorted (for best-fit).
- Searching the list is O(n) in the number of free entries.
- Coalescing requires checking neighbors — not O(1).
- The buddy system helps but adds implementation complexity.
- In a long-running server, the free list can grow to thousands of entries.

Compare this to what paging will offer: a free list of identically sized 4 KB frames — any frame is interchangeable with any other.

A.5.5 · Sparse Address Spaces: Heap/Stack Collision

Even with segmentation, very large virtual address spaces create challenges:

```
64-bit process:
  Code:   0x0000 0000 0000 – 0x0000 0010 0000  (1 MB)
  Heap:   0x0000 0010 0000 – grows upward ↑
  Stack:  0x7FFF FFFF F000 – grows downward ↓

Heap and stack are 128 TB apart in virtual space.
Under segmentation: the OS still needs to find contiguous physical blocks
for each segment as they grow.
If either grows to many GB: finding a large enough contiguous physical block
becomes increasingly difficult.
```

Segmentation still requires each segment to be **physically contiguous** — a constraint that paging removes entirely.

A.5.6 · Difficult to Support Very Large Heaps or Stacks

A large database server might allocate a 16 GB heap:

```
With segmentation:
  Heap segment: base = some physical address, bounds = 16 GB
  OS must find 16 GB of CONTIGUOUS physical memory for the heap.
  On a 32 GB machine: after other processes, this may be impossible
  even if 20 GB is free — but fragmented.
```

With paging, a 16 GB heap is split into 4,194,304 × 4 KB pages, each placed **independently** anywhere in physical RAM. The OS only needs 4 KB at a time — never a contiguous 16 GB block.

A.5.7 · Historical Context: Intel x86 Segmentation

Intel's 16-bit and 32-bit architectures implemented hardware segmentation:

- **8086 (1978):** 16-bit segments with 16-bit selectors × 16 = 20-bit physical address space (1 MB).
- **80286 (1982):** Protected mode with a segment descriptor table (GDT/LDT). Segment selectors index into a table of base/bounds/protection descriptors.
- **80386 (1985):** 32-bit segments — up to 4 GB per segment.

In practice, operating systems (Linux, Windows) running on x86-32 used segmentation in a **"flat" mode**: all segments (CS, DS, SS) point to the same base=0, bounds=4GB range, effectively disabling segmentation. Real memory protection was done through paging.

x86-64: Segment bases are largely ignored (forced to 0 for CS, DS, SS). Segmentation is a dead end in mainstream architecture.

📎 A.6 — Introduction to Paging

A.6.1 · Page Size: Typical Values and Trade-offs

Modern systems use a **4 KB** page size as the standard, with larger "huge pages" as an option:

Page Size	Pros	Cons
Small (e.g., 512 B)	Less internal fragmentation	Huge page tables; more TLB entries needed
4 KB (standard)	Good balance; matches disk sector size	Some internal fragmentation
2 MB (huge page)	Fewer TLB entries; faster for large allocations	More internal fragmentation
1 GB (gigantic page)	Ideal for large in-memory databases	High internal fragmentation for small processes

The 4 KB standard emerged from a careful balance of page table size, TLB coverage, and internal fragmentation.

A.6.2 · Example: 64-Byte Address Space, 16-Byte Pages

OSTEP Chapter 18 running example: 6-bit VA, 4-bit offset (16-byte pages), 4 pages.

Translate VA = 21:

```
VA = 21 = 0b010101 (6 bits)

VPN    = top 2 bits  = 0b01 = 1
Offset = bottom 4 bits = 0b0101 = 5

Page table lookup: VPN 1 → PFN 1

PA = PFN × 16 + offset = 1 × 16 + 5 = 21

→ Physical address = 21 (Frame 1, byte 5)
```

Check: Frame 1 spans physical addresses 16–31. PA=21 is within frame 1. ✅

A.6.3 · Example: 4 KB Pages, 32-Bit Address Space

More realistic example: 32-bit VA, 4 KB pages, process with page table loaded.

Translate VA = 0x00003204:

```
Page size = 4 KB = 4,096 bytes → 12 offset bits

VPN    = 0x00003204 >> 12 = 0x3 = 3
Offset = 0x00003204 & 0xFFF = 0x204 = 516

Page table lookup: VPN 3 → PFN = 7 (from page table)

PA = 7 × 4,096 + 516 = 28,672 + 516 = 29,188 = 0x7204

→ Physical address = 0x7204
```

Note: the low 12 bits (0x204) are **identical** in VA and PA — only the top bits changed.

A.6.4 · What Happens When the Valid Bit = 0?

If a process accesses a page whose PTE has `valid = 0`:

```
Process accesses VA = 0x0000A000
VPN = 0xA = 10

Page table entry at VPN 10:
  valid = 0  ← page not in physical memory!

Hardware raises: PAGE FAULT exception
  → CPU switches to kernel mode
  → Jumps to OS page fault handler

OS page fault handler options:
  1. Page is on disk (swapped out): load it from disk → update PTE → retry.
  2. Page was never allocated (access to unmapped address): deliver SIGSEGV.
  3. Page is in another process (copy-on-write): allocate new frame, copy, update PTE.

We study swapping and on-demand paging in Lecture 2.7.
```

📎 A.7 — Paging: Address Translation in Detail

A.7.1 · Page Table Entry (PTE) Fields in Detail

A closer look at each PTE flag and when the OS or hardware uses it:

Flag	Set by	Used for
Valid (V)	OS (on page allocation)	Hardware: if 0 → page fault; if 1 → proceed with translation
Writable (W)	OS	Hardware: if access=write and W=0 → protection fault
User (U)	OS	Hardware: if U=0 and CPU in user mode → protection fault
Accessed (A)	Hardware (automatically)	OS page replacement: which pages have been recently used?
Dirty (D)	Hardware (on write)	OS: if D=1, page must be written back to disk before eviction
PFN	OS (on page allocation)	Hardware: high-order bits of physical address

A.7.2 · The Memory Cost of Paging

Paging introduces a new space overhead: the page table itself lives in **physical memory**:

```
Without paging:
  Actual code + data + heap + stack → in physical RAM

With paging:
  Actual code + data + heap + stack → in physical RAM
  PLUS: page table for this process → ALSO in physical RAM

For a 32-bit process:
  Page table = 4 MB (as calculated above)
  This 4 MB must be resident in RAM for the hardware to walk it (Lectures 2.6–2.7 show how this requirement is relaxed)
```

The OS must allocate physical frames for the page table itself. On a machine running 100 processes, 400 MB is consumed by page tables alone — before any program data is loaded.

A.7.3 · The Page Table Base Register (PTBR)

Where does the hardware find the page table for the currently running process?

The OS stores the **physical address of the page table** in a dedicated CPU register:

```
x86-64: CR3 register (Control Register 3)
ARM:    TTBR0 / TTBR1 (Translation Table Base Registers)
RISC-V: satp register (Supervisor Address Translation and Protection)

On context switch:
  Save current process's PTBR to PCB.
  Load next process's PTBR from PCB.
  → Hardware now uses the new process's page table for all translations.
```

Setting the PTBR is a **privileged instruction** — only the OS can redirect the hardware to a different page table.

A.7.4 · Virtual Address Structure: VPN Bits + Offset Bits

The number of VPN and offset bits is fixed by the page size and address space size:

```
General formula:
  offset_bits = log₂(page_size)
  vpn_bits    = address_bits - offset_bits

Examples:
  32-bit AS, 4 KB pages:   offset=12,  vpn=20
  32-bit AS, 64 KB pages:  offset=16,  vpn=16
  64-bit AS, 4 KB pages:   offset=12,  vpn=52 (or 36 for 48-bit VA)
  OSTEP example (64B, 16B pages): offset=4, vpn=2

The offset bits determine WITHIN-PAGE position (unchanged in translation).
The VPN bits determine WHICH PAGE (translated via page table to PFN).
```

A.7.5 · Calculating VPN and Offset From a Virtual Address

Step-by-step procedure:

```
Given: VA = X, page_size = P (must be a power of 2)

Step 1: offset_bits = log₂(P)
Step 2: offset      = X mod P   = X & (P - 1)   (bottom offset_bits bits)
Step 3: VPN         = X / P     = X >> offset_bits  (top bits)

Example: VA = 0x1A3C, page_size = 4 KB = 0x1000
  offset_bits = 12
  offset      = 0x1A3C & 0x0FFF = 0xA3C = 2,620
  VPN         = 0x1A3C >> 12    = 0x1   = 1

Example: VA = 45, page_size = 16 bytes
  offset_bits = 4
  offset      = 45 & 15  = 45 mod 16 = 13
  VPN         = 45 >> 4  = 45 / 16   = 2
```

A.7.6 · Step-by-Step Translation Example (OSTEP Ch. 18)

Setup: 64-byte address space, 16-byte pages, 128-byte physical memory.

```
Page table for Process A:
  VPN 0 → PFN 2 (valid)
  VPN 1 → PFN 1 (valid)
  VPN 2 → PFN 3 (valid)
  VPN 3 → PFN 5 (valid)
```

Translate VA = 21:

```
Step 1: offset_bits = log₂(16) = 4
Step 2: VPN    = 21 >> 4  = 1
        Offset = 21 & 0xF = 5
Step 3: Page table lookup: VPN 1 → PFN 1
Step 4: PA = (PFN 1 << 4) | 5 = 16 + 5 = 21
```

Translate VA = 33:

```
VPN = 33 >> 4 = 2,  Offset = 33 & 0xF = 1
PFN = page_table[2] = 3
PA  = (3 << 4) | 1  = 48 + 1 = 49
```

A.7.7 · Protection Bits in PTEs: Enforcing R/W/X Per Page

Paging provides **per-page** protection — finer-grained than per-segment:

```
Page Table Entry protection enforcement:

  Access: instruction fetch (Execute)
  PTE.prot must include X bit.
  If not: protection fault → SIGSEGV.

  Access: data load (Read)
  PTE.prot must include R bit.
  If not: protection fault → SIGSEGV.

  Access: data store (Write)
  PTE.prot must include W bit.
  If not: protection fault → SIGSEGV.

Real-world use:
  Code pages:  R-X  → readable and executable, not writable.
  Data pages:  RW-  → readable and writable, not executable.
  Copy-on-write pages: R-- initially → write triggers fault → OS makes private copy.
```

A.7.8 · Page Table Size: 32-Bit vs. 64-Bit

Let's calculate page table sizes for realistic architectures:

Architecture	VA bits	Page size	Entries	PTE size	Table size
x86 (32-bit)	32	4 KB	2^{20} = 1M	4 B	**4 MB**
x86 (32-bit)	32	64 KB	2^{16} = 64K	4 B	**256 KB**
x86-64	48	4 KB	2^{36} = 64G	8 B	**512 GB** ← impossible!
x86-64 (multi-level)	48	4 KB	—	8 B	~few KB–MB per process
RISC-V (Sv39)	39	4 KB	2^{27} = 128M	8 B	**1 GB** ← still too large!

The flat page table is feasible for 32-bit systems but completely **infeasible for 64-bit systems**. Multi-level page tables (Lecture 2.6) solve this.

A.7.9 · The Memory Cost of Page Tables

Every process needs its own page table in physical RAM:

```
System: 100 processes, 32-bit address space, 4 KB pages

Page table memory per process: 4 MB
Total page table memory: 100 × 4 MB = 400 MB

If the machine has 4 GB of RAM:
  400 MB / 4,096 MB = ~10% of RAM used JUST for page tables.

With 1000 processes (server workload):
  1000 × 4 MB = 4 GB — ALL of RAM consumed by page tables!
```

This is the "space overhead" of paging. Combined with multi-level page tables (which only allocate table pages for active regions), this becomes manageable.

A.7.10 · Where Is the Page Table Stored?

The page table is stored in **physical memory** (RAM), not in the CPU:

```
CPU state:
  PTBR = physical address of the base of the page table
  (x86: CR3 register)

Physical memory:
  Frame 0:    OS kernel code
  Frame 1:    OS kernel data
  Frame 2:    Page table for Process A  ← pointed to by CR3 when A runs
  Frame 3:    Page table for Process B
  Frame 4:    Process A's actual code
  ...

On context switch:
  OS stores Process A's PTBR in PCB_A.
  OS loads Process B's PTBR from PCB_B → CR3.
  Now all memory accesses use B's page table.
```

A.7.11 · Setting Up: Page Table Base Register

The PTBR is a **privileged register** — only the OS (Ring 0) can write to it:

```
# x86-64 context switch (simplified):
# Save old process's CR3 (PTBR) to its PCB:
mov  rax, cr3          # read current PTBR
mov  [pcb_old + ptbr_offset], rax

# Load new process's CR3 from its PCB:
mov  rax, [pcb_new + ptbr_offset]
mov  cr3, rax          # write new PTBR — this is a privileged instruction!
                       # also flushes the TLB (discussed in Lecture 2.6)
```

Writing CR3 immediately causes the hardware to use the new process's page table for all future translations — effective from the very next instruction.

📎 A.8 — Comparison & Summary

A.8.1 · When Is Segmentation the Right Choice?

Segmentation fits well when:

- The address space has a **small, well-defined number** of logical regions (code, heap, stack, shared library).
- You want **coarse-grained sharing** — sharing an entire code region, not individual pages.
- The system is simple (embedded, microkernel) and external fragmentation is manageable.
- Protection at **segment granularity** is sufficient.

Modern usage: largely **replaced by paging**, but the *concept* lives on in:

- **ELF segments** (PT_LOAD entries map file regions to address ranges).
- **Memory-mapped files** (`mmap()`) — each mapping is logically a segment.
- **Intel SGX enclaves** — defined by base/bounds registers.

A.8.2 · When Is Paging the Right Choice?

Paging fits well for virtually all modern general-purpose OS workloads:

- Large, sparse address spaces (64-bit processes with GB+ virtual ranges).
- **No external fragmentation** — critical for long-running servers.
- **Fine-grained sharing and protection** — per-page read/write/execute bits.
- **Swapping** — individual pages can be moved to disk and back.
- **Copy-on-write** after `fork()` — individual pages copied only on write.
- **Memory-mapped files** — backed by file pages, evicted/loaded on demand.

Every modern general-purpose OS (Linux, Windows, macOS, iOS, Android) uses paging as its primary memory virtualization mechanism.

A.8.3 · Hybrid Approach: Segmented Paging (x86-32)

Intel's 32-bit x86 architecture combined both mechanisms:

```
Virtual address → Segment selector → Segment base + offset → Linear address
Linear address → Page table → Physical address

Step 1 (Segmentation): VA + segment base = linear address
  Each segment (CS, DS, SS) has its own base in the GDT/LDT.
  On Linux x86-32: all segments have base=0 → linear address = VA (flat mode).

Step 2 (Paging): linear address → page table lookup → physical address
```

In practice, Linux and Windows used **flat segmentation** (all bases = 0, all limits = 4 GB), making segmentation a no-op. All real protection and virtualization was done through paging. x86-64 goes further: segment bases are architecturally forced to 0 for CS, DS, SS — segmentation is essentially gone.

A.8.4 · Key Formulas

Segmentation translation:

$$\text{PA} = \text{base}[\text{seg}] + \text{offset}$$

$$\text{valid if: } 0 \leq \text{offset} < \text{bounds}[\text{seg}]$$

Segmentation (stack, grows down):

$$\text{PA} = \text{base}[\text{seg}] + (\text{offset} - \text{max_seg_size})$$

$$\text{valid if: } |\text{offset} - \text{max_seg_size}| \leq \text{bounds}[\text{seg}]$$

Paging translation:

$$\text{VPN} = \lfloor \text{VA}/\text{page_size} \rfloor, \quad \text{offset} = \text{VA} \bmod \text{page_size}$$

$$\text{PA} = \text{PFN} \times \text{page_size} + \text{offset}$$

Page table size (flat):

$$\text{size} = \frac{2^{\text{VA_bits}}}{\text{page_size}} \times \text{PTE_size_bytes}$$

A.8.5 · Self-Check Bank (1/2): Questions from § 2–5

Extra questions beyond the in-class checkpoints:

- What are the two methods for determining which segment a virtual address belongs to (explicit and implicit)? Which one needs extra address bits?
- Why can the code segment be shared between two processes running the same program, but the heap and stack cannot?
- A process terminates. It had code at physical 40–42 KB, heap at 20–23 KB, stack at 56–58 KB. Describe the free-list updates, including any coalescing if 42–44 KB was already free.
- Why must compaction stop the process being relocated? What would happen if it kept running?
- A free list has entries (10KB, 4KB), (20KB, 8KB), (35KB, 6KB). A request for 5 KB arrives. Which block is chosen under first-fit, best-fit and worst-fit?
- Explain the buddy-system coalescing rule. What is its time complexity?
- Why did Linux and Windows effectively disable x86 hardware segmentation and use it in "flat mode"?

A.8.6 · Self-Check Bank (2/2): Questions from § 6–7

- Why does writing to the CR3/PTBR register also flush the TLB? (Think: what does the TLB cache?)
- A 64-byte address space has 16-byte pages and the page table VPN 0→PFN 2, 1→PFN 1, 2→PFN 3, 3→PFN 5. Translate VA = 45.
- Compare the flat page-table sizes of a 32-bit/4 KB-page system (4-byte PTEs) and a 48-bit/4 KB-page system (8-byte PTEs). Which one is feasible?