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 2.4: Virtualization IV

Address Space and Base/Bounds

Opening Thought

> *"The OS, with a little hardware support, tries its best to get out of the way of the running program, to deliver an efficient virtualization; however, by interposing at those critical points in time, the OS ensures that it maintains control over the hardware."*
> *— Remzi & Andrea Arpaci-Dusseau, OSTEP Ch. 15*

We have virtualized the CPU — made one physical processor appear as many. Now we turn to the second and equally fundamental pillar: **virtualizing memory**. By the end of this lecture you will understand how the OS creates the illusion of a private, large address space for every process, and the first concrete hardware mechanism that makes this possible.

Lecture Roadmap

This lecture is organized into **7 major sections**:

1. **Why Virtualize Memory?** — The problem and three goals
2. **The Address Space Abstraction** — Code, stack, heap, and the illusion
3. **Interlude: The Memory API** — `malloc`, `free`, and common errors
4. **Early Approach: Static Relocation** — Software-only translation and its limits
5. **Mechanism: Address Translation and Base/Bounds** — Hardware-based dynamic relocation
6. **OS Issues with Base/Bounds** — The OS role in memory management
7. **Limitations and the Path Forward** — Why base/bounds is not enough

> *Let's begin by asking: why is virtualizing memory so important?*

Prerequisite Check

To follow this lecture, you should recall:

- **Process address space layout** (code, data, heap, stack) — from Lectures 1.3 and 1.4.
- **CPU privilege modes** (User/Kernel) — from Lecture 1.2.
- **System calls and traps** — from Lecture 1.2.
- **Context switching** .
- `malloc` / `free` — from Lecture 1.4 (C Programming II).

Memory virtualization builds directly on top of the CPU virtualization infrastructure we already understand.

§ 1 — Why Virtualize Memory?

The Memory Problem

A modern computer runs dozens or hundreds of processes simultaneously, but has **only one physical RAM**:

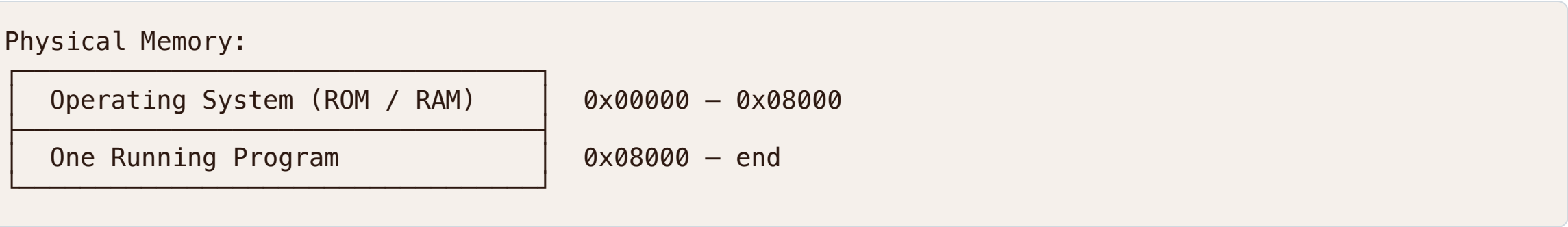
```
Physical RAM (e.g., 16 GiB):
┌──────────────────────────────────────────┐
│  ???  How do we share this among all     │
│       running processes safely?          │
└──────────────────────────────────────────┘
```

Questions the OS must answer:

- Where in physical RAM is each process's code and data stored?
- How does a process address its own data without accidentally overwriting another process?
- What happens if a buggy process writes to address `0x00000000`?

Early Approach: No Abstraction (Monoprogramming)

The very first computers offered **no memory abstraction at all**:



- Only one program at a time shared the machine with the OS.
- The program used **physical addresses** directly.
- If the program wrote to the wrong address it could corrupt the OS.
- **No multiprogramming** — the CPU was idle whenever the program waited for I/O.

The Problem with Monoprogramming

Monoprogramming wastes resources severely:

- **CPU utilization:** A program doing disk I/O for 10 ms (very fast) at 3 GHz means **30 million wasted CPU cycles** per I/O operation.
- **Memory utilization:** Expensive DRAM sits mostly unused — only one program's data is present.
- **User experience:** Only one user can run a program at a time.

The demand for multiprogramming — running multiple programs simultaneously — was the key driver for memory virtualization.

Multiprogramming: The Goal

To improve CPU utilization and support multiple users, systems began loading **multiple programs into memory simultaneously**:

```
Physical Memory:
┌─────────────────────────────┐
│  Operating System           │
├─────────────────────────────┤
│  Process A (text editor)    │
├─────────────────────────────┤
│  Process B (compiler)       │
├─────────────────────────────┤
│  Process C (web server)     │
└─────────────────────────────┘
```

When Process A blocks on I/O, the OS switches to Process B. CPU stays productive.

The Protection Problem

Placing multiple programs in memory immediately creates a **dangerous problem**:

```
Physical Memory (naive layout):

┌──────────────────────────────────┐
│  OS (address 0–64 KB)            │
├──────────────────────────────────┤
│  Process A (64–128 KB)           │
├──────────────────────────────────┤
│  Process B (128–192 KB)          │
└──────────────────────────────────┘

If Process A writes to address 130 KB → corrupts Process B's data!
If Process A writes to address 10 KB  → corrupts the OS!
```

Without protection, any process can accidentally or maliciously destroy another process or the kernel.

Three Goals of Virtual Memory

Every virtual memory system must achieve three fundamental goals:

1. Transparency

The OS should implement virtual memory **invisibly**. The running program behaves as if it has its own private physical memory. It is unaware of sharing.

2. Efficiency

The virtualization must be fast (no slowdown per memory access) and space-efficient (not wasting RAM on OS structures). This requires **hardware support**.

3. Protection

No process must be able to read or write any memory that does not belong to it. Isolation between processes, and between processes and the OS, is guaranteed.

Goal 1: Transparency

> *"The OS should implement virtual memory in a way that is invisible to the running program."*

What transparency means in practice:

- A program compiles and links assuming it starts at address 0.
- It uses addresses like `0x4010a0` for its code and `0x7fff1234` for its stack.
- These are **virtual addresses** — they don't correspond to real physical RAM locations.
- The same virtual address `0x1000` in Process A and Process B maps to completely different physical locations.

The program never knows it is sharing RAM with any other process.

Goal 2: Efficiency

> *"The OS should strive to make the virtualization as efficient as possible, both in terms of time and space."*

Efficiency constraints:

- **Time:** Address translation happens on **every single memory access** — instruction fetch, data load, data store. Even 1 extra nanosecond per access is catastrophic at billions of accesses per second.
- **Space:** The OS data structures that record memory mappings must themselves be small.

Implication: Pure software address translation is too slow. We need **hardware-accelerated** translation — the Memory Management Unit (MMU).

Goal 3: Protection

> *"The OS should protect processes from one another and the OS from processes."*

Protection requirements:

- Process A cannot **read** Process B's private data (confidentiality).
- Process A cannot **write** to Process B's memory (integrity).
- No process can corrupt or take over the OS kernel (security).

Hardware must enforce that every memory access by a process is within the bounds of its own allocated memory. Any violation must trigger an exception.

Without protection, the OS cannot maintain control of the machine.

✅ Checkpoint 1 — Why Virtualize Memory?

Test your understanding:

1. Explain the **protection problem** that arises when multiple processes are placed in physical memory without any abstraction.
2. State the three goals of virtual memory. Which one requires hardware support and why?
3. Why is pure software address translation insufficient for efficiency?
4. If both Process A and Process B reference virtual address `0x0000100`, are they accessing the same physical RAM cell? Explain.

Write down your answers before moving to the next section.

§ 2 — The Address Space Abstraction

Defining the Address Space

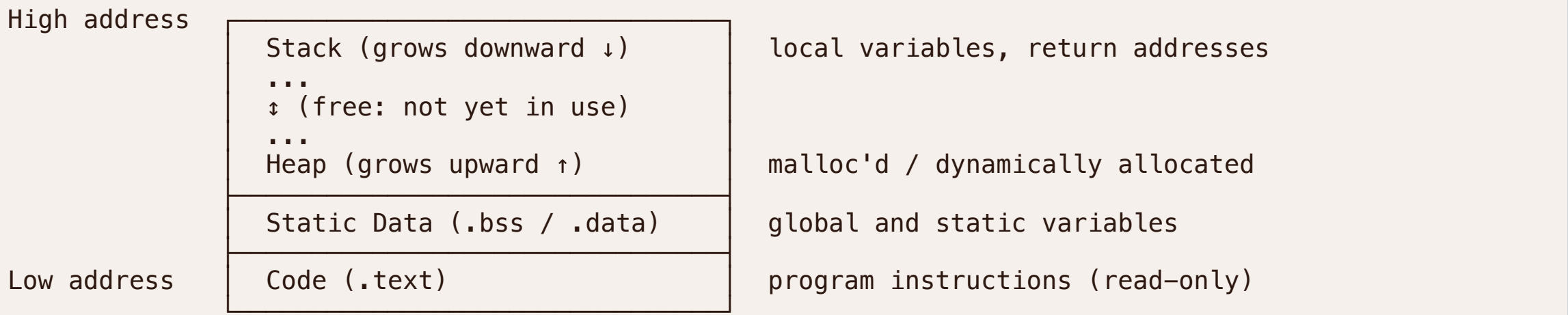
> ***The address space is the OS abstraction of physical memory: it is the running program's view of memory in the system.***

An address space is:

- A contiguous range of **virtual addresses** the program may reference.
- On a 32-bit system: addresses `0x00000000` to `0xFFFFFFFF` (4 GiB).
- On a 64-bit system: a subset of `0x0000000000000000` to `0xFFFFFFFFFFFFFFFF`.
- **Private:** Each process has its own, independent address space.
- **Illusory:** The virtual address range does not correspond directly to physical RAM.

Contents of an Address Space

An address space contains three logically distinct regions:



The Code Segment

The **code segment** (`.text`) contains the compiled machine instructions of the program:

- Located at the **bottom** (low addresses) of the address space.
- **Read-only:** The program must not overwrite its own instructions at runtime.
- **Fixed size:** Determined at compile time; does not grow during execution.
- Shared between threads within the same process (more on this later).
- On modern systems, the OS marks code pages as `R–X` (readable, executable, not writable) in the page table.

```
// This function's machine code lives in .text:
int add(int a, int b) {
    return a + b;
}
```

The Static Data Segment

Between the code and the heap lie two static data sections:

`.data` (Initialized Global/Static Variables)

```
int global_counter = 42;        // stored in .data
static char prefix[] = "LOG";   // stored in .data
```

`.bss` (Uninitialized / Zero-Initialized Global/Static Variables)

```
int uninitialized_array[1000]; // stored in .bss (zero-filled by OS loader)
static double result;          // stored in .bss
```

Both sections have fixed size determined at compile time. The OS zero-fills `.bss` before the program starts.

The Heap Segment

The **heap** holds memory **dynamically allocated at runtime**:

- Located just **above** the static data sections.
- **Grows upward** (toward higher addresses) as the program calls `malloc()`.
- Managed by the C runtime library (`glibc`'s `ptmalloc`, `jemalloc`, etc.).
- The library requests large blocks from the OS using `brk()` or `mmap()` system calls.
- Individual allocations within the block are handled by the library allocator.

```
int *arr = malloc(100 * sizeof(int)); // grows the heap
free(arr);                            // returns memory to allocator
```

The Stack Segment

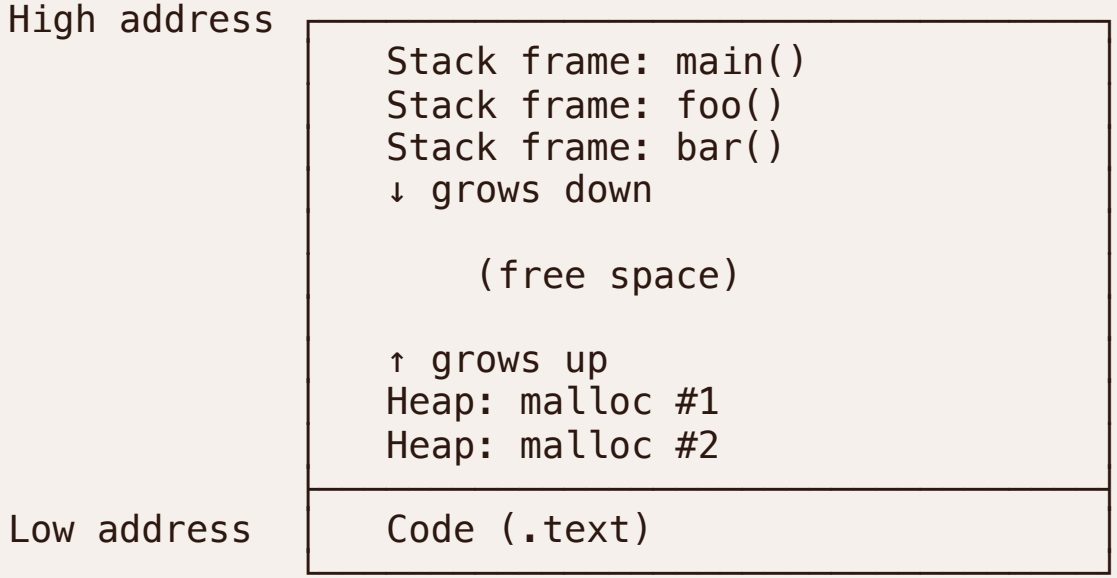
The **stack** holds each function's **activation frame**:

- Located at the **top** (high addresses) of the address space.
- **Grows downward** (toward lower addresses) with each function call.
- Contains: local variables, function parameters, return addresses, saved registers.
- Automatically managed by the compiler and hardware (Stack Pointer register).
- The OS sets an initial stack size (typically 1–8 MB) and may extend it on demand.

```
void foo(int x) {
    char buf[256];  // allocated on the stack — automatic, no malloc needed
    // ...
}  // stack frame popped when foo returns
```

Stack and Heap: Growth Directions

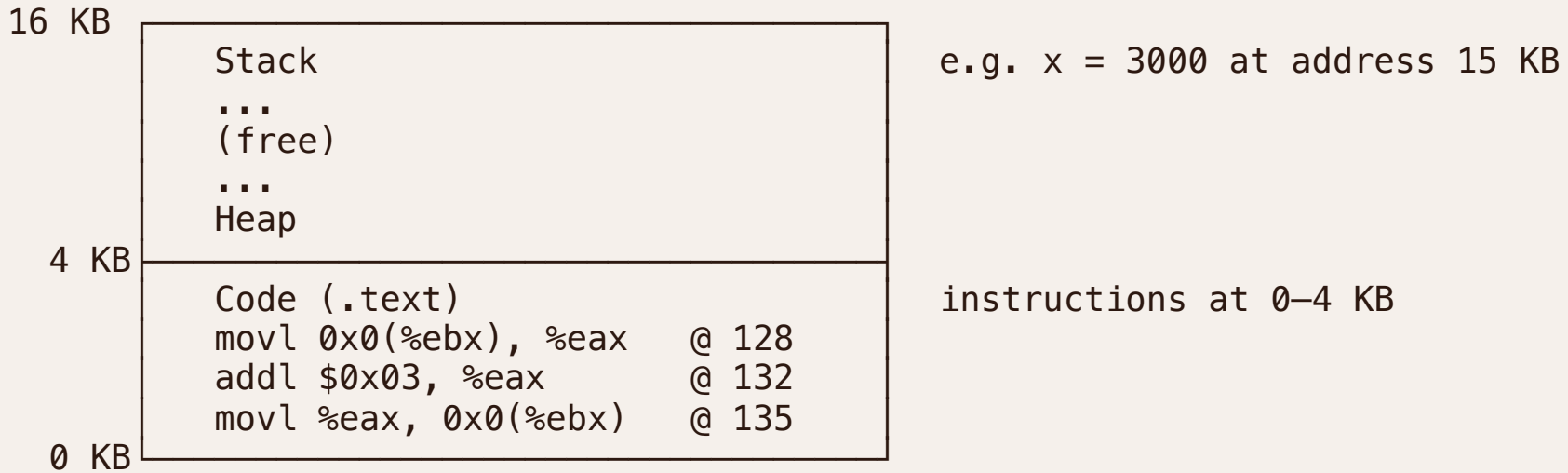
A key structural feature: stack and heap **grow toward each other**:



If they meet: **stack overflow** or **heap exhaustion** → OS kills the process.

A Concrete Example: 16 KB Address Space

Let's trace a small but concrete address space example:



The program believes it runs starting at virtual address 0 and up to 16 KB. It has no idea where in physical RAM it actually sits.

Virtual vs. Physical Address: The Key Distinction

This distinction is perhaps the most important concept in memory virtualization:

	Virtual Address	Physical Address
Generated by	CPU executing program instructions	MMU hardware after translation
Seen by	The program / process	The RAM chips
Range	`0` to `2^N − 1` (per process)	`0` to `RAM_size − 1` (global)
Unique?	Not globally — two processes can have the same virtual address	Yes — one physical address per RAM cell
Example	`0x0000 1000` (always from 0)	`0x8000 1000` (where OS placed it)

Every address a program generates is a virtual address. Physical addresses are never visible to user programs.

Every Address Is Virtual

A common misconception: "my program uses real addresses."

```c
#include <stdio.h>

int x = 42;

int main(void) {
    printf("Address of x: %p\n", (void *)&x);
    // Output: 0x601040 (or similar)
    // This is a VIRTUAL address — not the physical RAM location!
    return 0;
}
```

Even when you print an address in C, you see the **virtual address**. The OS and MMU silently translate it to a physical address before accessing RAM. The program is completely unaware of this translation.

The Illusion: Two Processes, Same Virtual Address

Two processes can both use virtual address `0x00001000` with no conflict:

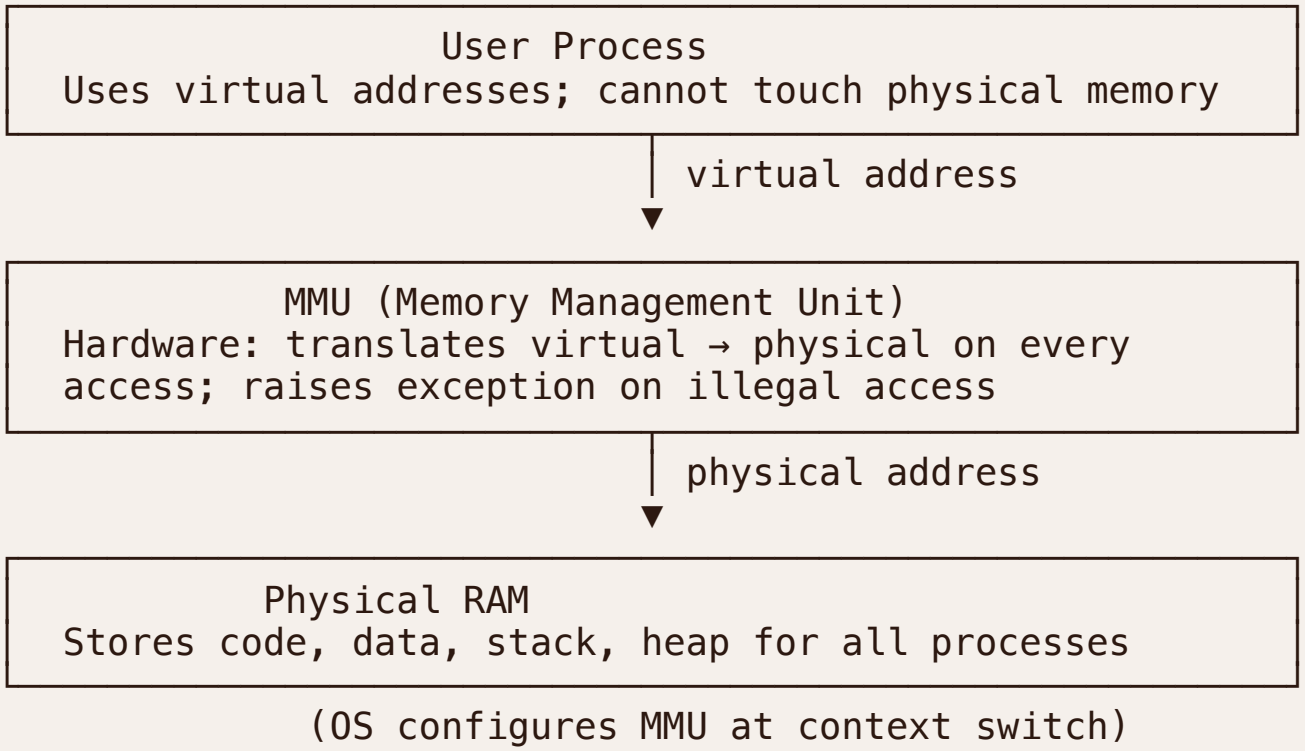
```
Process A (virtual address 0x1000) ──MMU──► Physical address 0x32000
Process B (virtual address 0x1000) ──MMU──► Physical address 0x58000
```

From Process A's perspective: it has the entire 4 GiB virtual address space to itself.
From Process B's perspective: exactly the same.

The OS, via the MMU, ensures translations are completely different and independent. *This is the address space abstraction in action.*

The OS/Hardware Contract

The address space illusion is maintained through a cooperation between OS and hardware:



Address Space Size: 32-bit vs. 64-bit

The size of a virtual address space depends on the CPU architecture:

Architecture	Address Bits	Virtual Address Space Size
16-bit (8086)	20-bit physical	1 MiB
32-bit (x86)	32-bit virtual	4 GiB
64-bit (x86-64)	48-bit virtual (implemented)	256 TiB
64-bit (ARMv8)	48–52 bits	256 TiB – 4 PiB

On a 64-bit Linux system, each process gets a private virtual address space split into:

- Lower half (`0x0000 0000 0000 0000` – `0x0000 7FFF FFFF FFFF`): **user space**.
- Upper half (`0xFFFF 8000 0000 0000` – `0xFFFF FFFF FFFF FFFF`): **kernel space**.

✅ Checkpoint 2 — The Address Space Abstraction

Test your understanding:

1. Draw the layout of a process address space. Label: code, static data, heap, stack, and the directions of growth.
2. Process A and Process B both have a variable `x` at virtual address `0x5000`. Do they share the same value of `x`? Explain.
3. What is the difference between a virtual address and a physical address? Who generates each?
4. On a 64-bit Linux system, a C program prints `Address of global: 0x601040`. Is this a virtual or physical address?
5. What happens if the stack and heap segments collide within an address space?

Sketch the address space diagram from memory before moving on.

§ 3 — Interlude: The Memory API

Types of Memory: Stack vs. Heap Allocation

C programs use memory from two sources, with fundamentally different management:

Stack (Automatic) Allocation

```c
void foo(void) {
    int x = 5;          // on the stack — automatic
    char buf[256];      // on the stack — automatic
}   // x and buf destroyed here, no free() needed
```

Heap (Dynamic) Allocation

```c
void bar(void) {
    int *p = malloc(sizeof(int) * 100);  // on the heap
    // ... use p ...
    free(p);   // must free explicitly
}
```

The OS provides a system call interface (`brk`, `mmap`) that the C runtime uses; the `malloc` family is a **library**, not a system call.

`malloc()` — From an OS Perspective

```c
#include <stdlib.h>
void *malloc(size_t size);
```

What happens beneath `malloc()`:

1. The C runtime allocator checks its free list for a suitable block.
2. If needed, it requests additional virtual memory from the OS:
    - `brk()` / `sbrk()` — adjusts the process's data segment.
    - `mmap(MAP_ANONYMOUS)` — creates an anonymous virtual memory mapping.
3. The OS establishes the virtual memory mapping. Physical memory allocation **may be deferred**.
4. When the process accesses a page that lacks physical backing, a page fault may cause the OS to allocate a physical page and map it into the process's address space.
5. The allocator returns a pointer to the requested block.

```c
int *arr = malloc(10 * sizeof(int));  // → may call brk() under the hood
if (!arr) { perror("malloc"); exit(1); }
```

`free()` and Memory Reclamation

```
void free(void *ptr);
```

When `free(ptr)` is called:

1. The C runtime allocator marks the block as available in its free list.
2. If the freed block is at the top of the heap, the allocator may call `sbrk(-size)` to shrink the heap and return pages to the OS.
3. Otherwise the block stays in the allocator's pool for future `malloc()` calls.

Important distinction:

- `free()` returns memory to the **allocator** (user-space).
- The OS reclaims physical memory only when the allocator decides to release pages, or when the **process terminates**.

On process termination, the OS automatically reclaims all memory — no leaks survive process exit.

Common Memory Error 1: Forgetting to Allocate

```c
// BUG: using a pointer without allocating memory
char *str;
strcpy(str, "hello");  // ⚠ wild pointer — str points nowhere!

// Fix:
char *str = malloc(strlen("hello") + 1);
if (!str) { perror("malloc"); exit(1); }
strcpy(str, "hello");
free(str);
```

The OS will likely deliver a **segmentation fault** (`SIGSEGV`) because `str` contains garbage, pointing to some invalid virtual address, and the MMU rejects the access.

Common Memory Error 2: Buffer Overflow

```c
// BUG: not allocating enough memory
char *str = malloc(strlen("hello"));  // allocates 5, needs 6 (+ '\0')
strcpy(str, "hello");  // ⚠ writes 6 bytes into 5-byte buffer

// Fix:
char *str = malloc(strlen("hello") + 1);  // +1 for null terminator
strcpy(str, "hello");
free(str);
```

Writing past the end of an allocated block **corrupts the allocator's metadata** stored in adjacent memory, leading to crashes during future `malloc()` or `free()` calls — often far from the actual bug.

Common Memory Error 3: Memory Leak

```
// BUG: allocating without freeing
void process_request(void) {
    int *buf = malloc(4096);
    // ... handle request ...
    return;   // ⚠ buf is never freed!
}

// In a server handling 1 million requests:
// 1,000,000 × 4,096 bytes = ~4 GB leaked!
```

Memory leaks do not cause an immediate crash but slowly consume physical RAM until the system becomes unresponsive. The OS reclaims all leaked memory only when the process terminates.

Common Memory Error 4: Dangling Pointer (Use After Free)

```
int *p = malloc(sizeof(int));
*p = 42;
free(p);

// ⚠ p is now a dangling pointer — the memory has been returned
printf("%d\n", *p);   // Undefined Behavior: may print 42, garbage, or crash
*p = 99;              // Undefined Behavior: corrupts heap allocator metadata!

// Fix: always null the pointer after free
free(p);
p = NULL;
```

The allocator may have given the freed block to another `malloc()` call. Writing through a dangling pointer corrupts other live data structures.

Common Memory Error 5: Double Free

```
int *p = malloc(sizeof(int));
free(p);
free(p);   // ⚠ DOUBLE FREE — heap corruption

// The allocator's free list metadata is now corrupted.
// Future malloc/free calls may crash or produce wrong results.
// This class of bug is frequently exploited for security attacks.

// Fix: set to NULL after free — free(NULL) is always a safe no-op
free(p);
p = NULL;
free(p);  // ✅ safe: free(NULL) does nothing
```

Common Memory Error 6: Invalid Free

```c
// BUG: freeing stack memory
int x = 5;
free(&x);   // ⚠ &x is a stack address, not a heap pointer!

// BUG: freeing middle of a block
int *arr = malloc(10 * sizeof(int));
free(arr + 5);   // ⚠ not the start of the allocation!

// BUG: freeing a pointer to a string literal
char *s = "hello";
free(s);    // ⚠ string literals live in .rodata, not the heap!
```

The allocator expects **exactly the pointer returned by** `malloc()`. Any deviation is undefined behavior.

Tools for Detecting Memory Errors

Two essential tools (both seen in Lecture 1.4 from the C programming side; now from the OS perspective):

Valgrind (memcheck)

```
gcc -g -O0 program.c -o program
valgrind --leak-check=full --show-leak-kinds=all ./program
```

Instruments every memory access dynamically. Detects: leaks, use-after-free, invalid reads/writes. Overhead: ~10–50× slower.

AddressSanitizer (ASan)

```
gcc -g -fsanitize=address,undefined -o program program.c
./program
```

Compiler-instrumented. Detects: heap/stack/global buffer overflows, use-after-free, double-free. Overhead: ~2× slower.

Both tools work because they understand the boundary between the process's valid address space and invalid memory.

✅ Checkpoint 3 — The Memory API

Test your understanding:

1. What is the difference between stack and heap memory in terms of allocation and lifetime?
2. When `free(ptr)` is called, is the physical memory immediately returned to the OS? Explain.
3. A server calls `malloc(1024)` on every request but never calls `free()`. After 10 million requests, what happens to the process's memory consumption?
4. Why does a double-free often cause a crash only much later — not immediately at the second `free()`?
5. What is the difference between Valgrind and AddressSanitizer in terms of detection approach and overhead?

§ 4 — Early Approach: Static Relocation

The Static Relocation Concept

Before hardware address translation existed, operating systems used **static relocation** — a purely software technique:

> *The OS loader rewrites all absolute addresses in the program binary **before** it starts executing, adjusting them to the physical location where the program has been placed.*

```
Program binary on disk:    After static relocation (loaded at 3000):
  movl 1000, %eax     →     movl 4000, %eax     (1000 + 3000)
  jmp  500            →     jmp  3500           (500  + 3000)
  movl %eax, 2000     →     movl %eax, 5000     (2000 + 3000)
```

The loader scans every instruction, identifies absolute address references, and adds the base offset.

How the Loader Performs Static Relocation

The static relocation process:



The relocation table (produced by the linker) lists every location that contains an absolute address requiring patching.

Advantages of Static Relocation

Static relocation has a few genuine strengths:

- **No hardware required:** Any CPU, even the simplest, can support it.
- **Zero runtime overhead:** All patching done at load time; running instructions are native addresses already.
- **Simple implementation:** The loader is relatively straightforward.

For very constrained embedded microcontrollers without an MMU, a simplified form of static relocation is still used today.

Critical Disadvantage 1: No Protection

Static relocation provides **no memory protection**:

```
After relocation:
  Process A loaded at  0–16 KB (addresses rewritten to 0–16 KB range)
  Process B loaded at 16–32 KB (addresses rewritten to 16–32 KB range)

Nothing prevents Process A from executing:
  movl $42, 20000    ← writes to address 20000 (inside Process B!)
```

Process A can freely generate any address. There is no hardware check. Static relocation only adjusts the *initial* addresses — once running, the program can compute and use arbitrary addresses.

Critical Disadvantage 2: No Relocation After Load

Once statically relocated and running, **the process cannot be moved**:

- All addresses in the running program's instructions and data are physical addresses.
- If the OS wants to move the process to a different physical location (to compact memory), it would need to:
    i. Stop the process.
    ii. Copy all its memory to a new location.
    iii. Re-scan and re-patch every absolute address in the running image.
    iv. Also patch any addresses currently sitting in CPU registers — impossible.
- In practice: **once placed, it stays there forever**.

This rigidity causes **external fragmentation** — the OS cannot reorganize memory to accommodate new processes.

Static Relocation Is Not Enough

Summary of static relocation's failures:

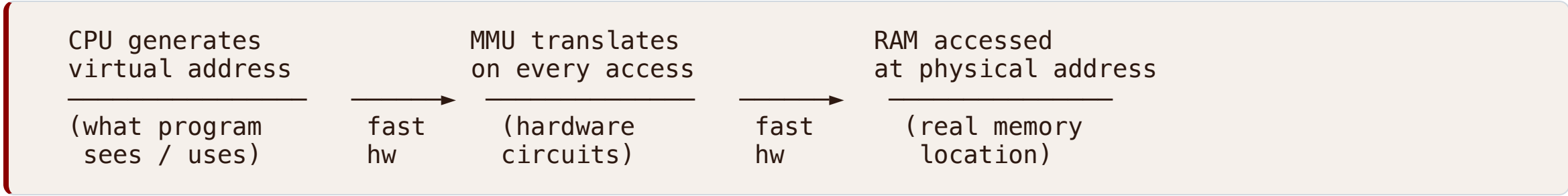
Property	Static Relocation	What We Need
Transparency	✅ Yes (program unaware)	✅
Efficiency	✅ (zero runtime overhead)	✅
Protection	❌ None	❌ Required
Relocatability	❌ Cannot move after load	✅ Required
Hardware needed	None	MMU

The conclusion is clear: **hardware support is essential** for correct, efficient, and safe memory virtualization. This brings us to the first hardware mechanism: **base and bounds**.

§ 5 — Mechanism: Address Translation & Base/Bounds

The Core Mechanism: Hardware-Based Address Translation

With hardware support, we replace static patching with **dynamic, per-access translation**:



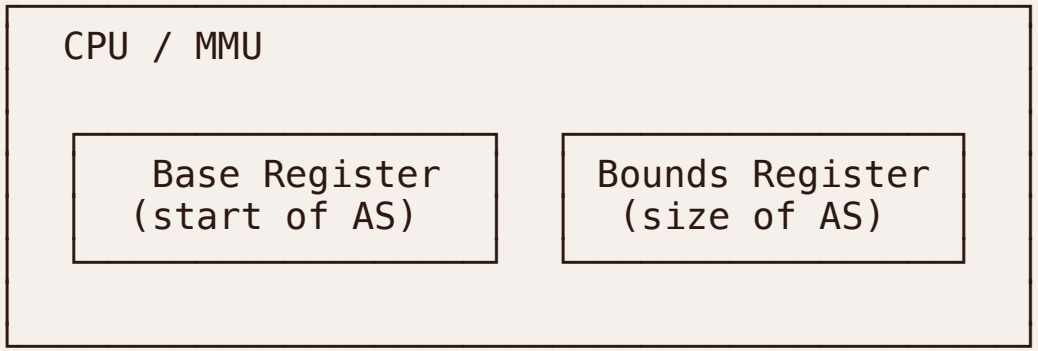
> ***Address translation*** *is the technique whereby the hardware transforms each virtual address generated by the program into the physical address where the data actually resides.*

This translation is **transparent** to the program, **efficient** (done in hardware), and — with proper hardware support — enforces **protection**.

Dynamic Relocation: The Base-and-Bounds Idea

The simplest hardware-based translation is **base and bounds** (also called **dynamic relocation**), introduced in late-1950s time-sharing machines:

The CPU is equipped with **two special registers** inside the MMU:



One pair of base/bounds registers exists **per CPU**. When the OS context-switches, it saves the old pair and loads the new pair for the next process.

The Base Register

The **base register** holds the physical start address of the process's address space:

```
Process A loaded at physical address 32 KB:
  Base register = 32,768 (32 × 1024)

Process B loaded at physical address 48 KB:
  Base register = 49,152 (48 × 1024)
```

When Process A is running: Base = 32,768.

When Process B is running: Base = 49,152.

Every virtual address generated by the running process is offset from this base.

The Bounds Register

The **bounds register** (also called the *limit register*) holds the **size** of the process's address space:

```
Process with 16 KB address space:
  Bounds register = 16,384 (16 × 1024)

This means:
  Valid virtual addresses: 0 to 16,383 (inclusive)
  Any virtual address ≥ 16,384 → ILLEGAL → hardware exception
```

Alternative definition: some architectures store the **end physical address** rather than the size. Both are logically equivalent.

The bounds register enforces **containment**: the process can only access memory within its own address space.

The Translation Formula

On every memory access, the hardware performs:

$$\text{physical address} = \text{virtual address} + \text{base}$$

And immediately checks:

$$\text{virtual address} \geq 0 \quad \text{AND} \quad \text{virtual address} < \text{bounds}$$

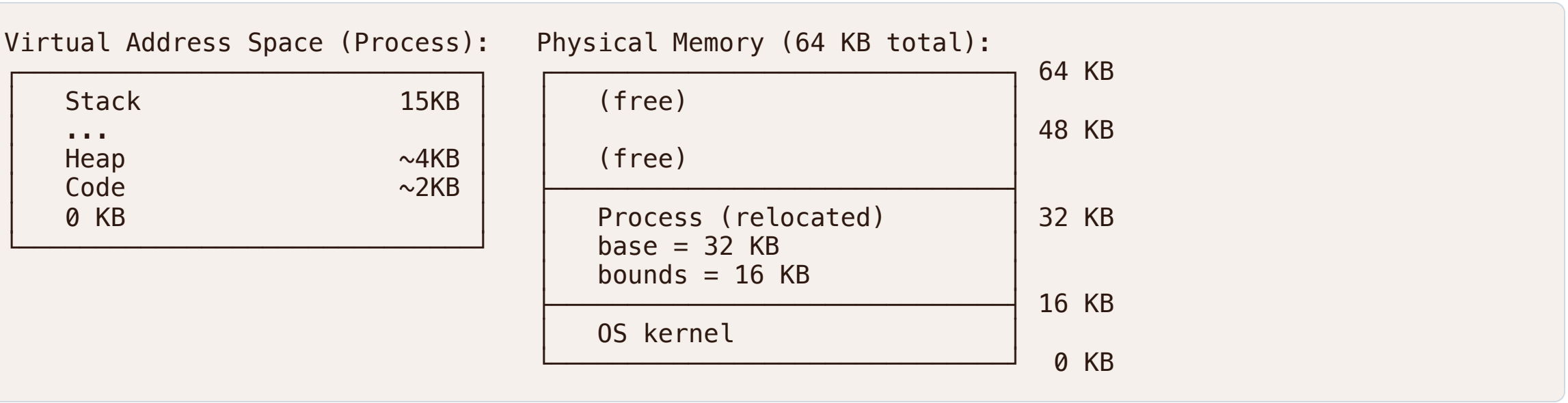
If the check fails → **segmentation fault** exception.

```
Example:
  Base   = 32 KB = 32,768
  Bounds = 16 KB = 16,384

  Virtual address 0     → physical 32,768  ✅
  Virtual address 1 KB  → physical 33,792  ✅
  Virtual address 15 KB → physical 48,128  ✅
  Virtual address 16 KB → ⚠ OUT OF BOUNDS → exception!
```

Virtual Address Space vs. Physical Memory Layout

Let's visualize a concrete example:



The process believes it lives at 0–16 KB. It actually occupies physical addresses 32–48 KB.

Example: Step-by-Step Instruction Trace

Let's trace the code example with base = 32 KB, bounds = 16 KB:

```
// C code (simplified):
int x = 3000;     // stored at virtual address 15 KB (stack)
x = x + 3;       // load x, add 3, store x
```

```
Compiled to three x86 instructions at virtual addresses 128, 132, 135:

1. Fetch instruction @ virtual 128:
   PC = 128 → physical = 128 + 32,768 = 32,896 ✅ → fetched from RAM

2. Execute: movl 0x0(%ebx), %eax  (load from virtual 15 KB = 15,360)
   15,360 < 16,384 ✅ → physical = 15,360 + 32,768 = 48,128 → RAM read

3. Execute: addl $0x03, %eax  (pure register operation — no memory access)

4. Execute: movl %eax, 0x0(%ebx)  (store to virtual 15 KB = 15,360)
   15,360 < 16,384 ✅ → physical = 15,360 + 32,768 = 48,128 → RAM write
```

The Bounds Check: Catching Illegal Accesses

What happens when a process accesses memory outside its bounds?

```
Base = 32 KB, Bounds = 16 KB

Process B attempts: load from virtual address 20 KB (20,480)

Step 1: Check bounds: 20,480 ≥ 16,384 → VIOLATION!
Step 2: Hardware raises an exception (Segmentation Fault).
Step 3: CPU switches to kernel mode, jumps to OS exception handler.
Step 4: OS handler terminates Process B, reports the error.
Step 5: OS continues running other processes normally.
```

Process A is completely unaffected — it was not involved. The hardware enforced the boundary perfectly.

Two Interpretations of the Bounds Register

There are two equivalent ways to define the bounds register:

Interpretation 1: Holds Address Space Size (OSTEP default)

$$\text{valid if: } 0 \leq \text{VA} < \text{bounds}$$
$$\text{physical} = \text{VA} + \text{base}$$

Interpretation 2: Holds End Physical Address

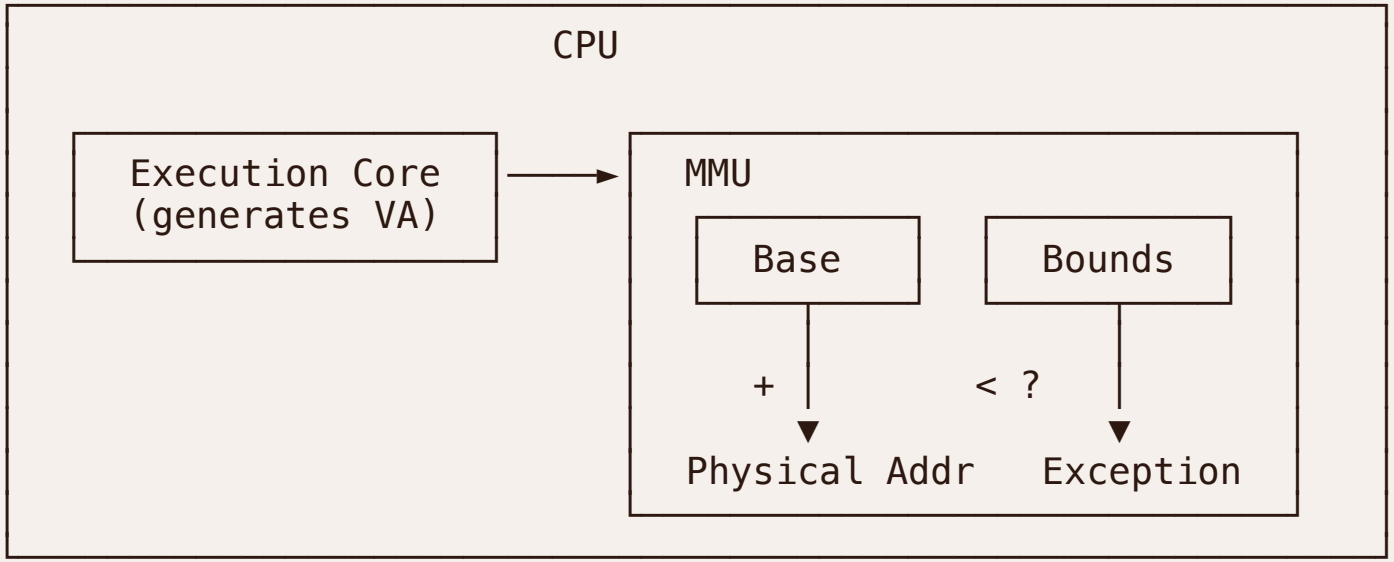
$$\text{valid if: } \text{base} \leq (\text{VA} + \text{base}) \leq \text{bounds}$$

Both are logically equivalent. **Interpretation 1** (size-based bounds) is simpler; **Interpretation 2** (end-address bounds) avoids one addition in the check circuit.

Real hardware implementations vary; the key insight is that the same protection guarantee is achieved.

The Memory Management Unit (MMU)

The hardware component performing these translations is called the **Memory Management Unit (MMU)**:



The MMU is part of the CPU chip. It sits between the execution core and the memory bus, intercepting every memory access.

Hardware Requirements Summary

What the CPU hardware must provide for base-and-bounds to work:

Hardware Feature	Purpose
Privileged mode (User/Kernel)	Prevents user programs from executing privileged instructions
Base register	Holds start address for address translation
Bounds register	Holds size limit for bounds checking
Translation circuitry	Performs `VA + base` on every memory access
Bounds check circuitry	Compares VA against bounds register
Privileged instructions to set base/bounds	Only OS can modify them
Ability to raise exceptions	Notifies OS of out-of-bounds or privilege violations
Exception handler registration	OS tells hardware where handlers are located

Privileged Instructions: Setting Base and Bounds

Setting the base and bounds registers is a **privileged operation** (Ring 0 / Kernel Mode only):

```
# x86-like pseudocode for OS context switch:
# Save old process's base/bounds:
mov [pcb_A + base_offset],   %cr_base    # save base to PCB
mov [pcb_A + bounds_offset], %cr_bounds  # save bounds to PCB

# Load new process's base/bounds:
mov %cr_base,   [pcb_B + base_offset]    # restore base from PCB
mov %cr_bounds, [pcb_B + bounds_offset]  # restore bounds from PCB
```

If a user-space program tried to execute these instructions:

- The CPU checks the current privilege level (CPL).
- If CPL = Ring 3 (User Mode) → General Protection Fault → OS terminates the process.

User programs cannot see or modify their own base/bounds registers.

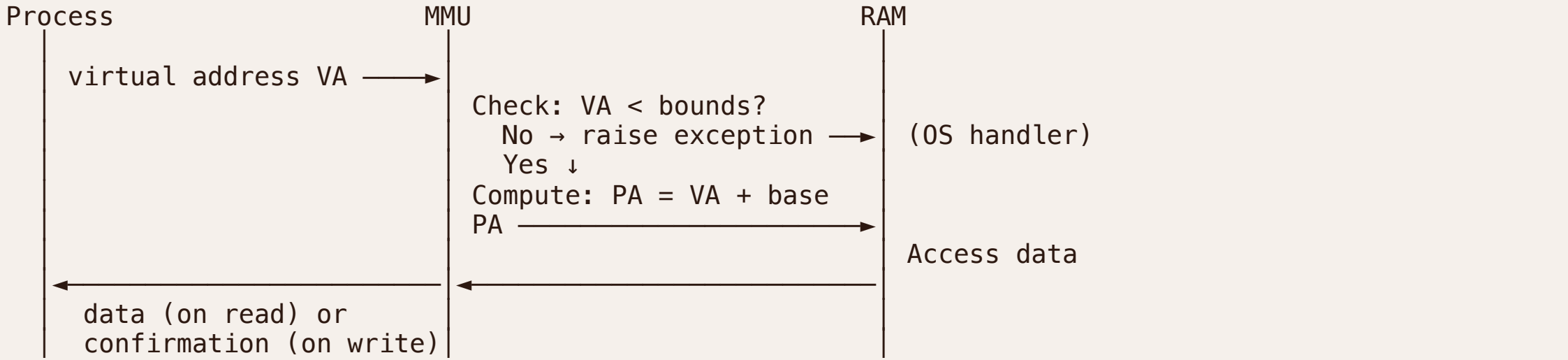
What Happens on an Out-of-Bounds Access

The complete hardware-to-OS flow when a process generates an invalid address:

```
1. Process (Ring 3) generates virtual address VA = 20 KB
2. MMU checks: 20 KB ≥ bounds (16 KB) → VIOLATION
3. Hardware raises "Segmentation Violation" exception
4. CPU:
   a. Saves current PC, SP, flags onto kernel stack
   b. Switches to Ring 0 (Kernel Mode)
   c. Jumps to OS exception handler (registered in IDT)
5. OS exception handler runs:
   a. Identifies the offending process
   b. Sends SIGSEGV signal to the process
   c. Terminates the process
   d. Frees its memory
6. OS returns to scheduling other processes
```

The Full Address Translation Pipeline

Every single memory access (instruction fetch, load, store) follows this path:



This entire pipeline executes in a small number of hardware clock cycles — negligible overhead.

Example Translation Table

Let's work through several translations concretely.

Setup: Base = 16 KB = 16,384. Bounds = 4 KB = 4,096.

Virtual Address	Bounds Check	Physical Address	Result
0	0 < 4,096 ✅	0 + 16,384 = **16,384**	Valid
1 KB = 1,024	1,024 < 4,096 ✅	1,024 + 16,384 = **17,408**	Valid
3,000	3,000 < 4,096 ✅	3,000 + 16,384 = **19,384**	Valid
4,096	4,096 < 4,096 ❌	—	**Fault**
4,400	4,400 < 4,096 ❌	—	**Fault**

Multiple Processes: Each Gets Its Own Base/Bounds Pair

The hardware stores one base/bounds pair **per CPU**. Each process has its own values, saved in the PCB:

```
OS: 0–16 KB
Process A PCB:  base_A = 32 KB,  bounds_A = 16 KB
Process B PCB:  base_B = 48 KB,  bounds_B = 16 KB

Physical Memory:
┌──────────────────┐  64 KB
│    Process B     │
├──────────────────┤  48 KB ← base_B = 48 KB
│    Process A     │
├──────────────────┤  32 KB ← base_A = 32 KB
│     (free)       │
├──────────────────┤  16 KB
│    OS Kernel     │
└──────────────────┘  0 KB
```

Dynamic Relocation: Moving a Process at Runtime

Unlike static relocation, base/bounds allows the OS to **move a process** while it is not running:

1. The OS **deschedules** the process (takes it off the run queue).
2. The OS **copies** the process's entire address space from old physical location to new.
3. The OS **updates the base register** in the process's PCB to the new location.
4. The OS reschedules the process. On next context switch, the new base is loaded.

The process is completely **unaware** it moved. Its virtual addresses are unchanged; only the base register changed.
This is the power of dynamic (hardware-based) relocation.

OS Free List: Tracking Available Physical Memory

The OS must know which physical memory regions are available when creating new processes:

```
Free List (example — 64 KB physical RAM, 16 KB slots):

Slot 0 (0–16 KB):   OS kernel     → IN USE
Slot 1 (16–32 KB):  (empty)       → FREE
Slot 2 (32–48 KB):  Process A     → IN USE
Slot 3 (48–64 KB):  (empty)       → FREE

Free list entries: { [16 KB, 16 KB], [48 KB, 16 KB] }
                     (start, size)    (start, size)
```

When a new process needs memory, the OS searches the free list for a suitable slot, removes it from the list, and records the assigned base/bounds in the process's PCB.

The Free List Data Structure

The OS maintains the free list as a linked list of available memory ranges:

```
// Simplified free list node:
struct free_node {
    uint64_t start;          // physical start address of free region
    uint64_t size;           // size in bytes
    struct free_node *next;  // pointer to next free region
};

// Example free list for 64 KB RAM with OS at 0–16 KB:
// [start=16384, size=16384] → [start=49152, size=16384] → NULL

struct free_node *free_list = &node1;
```

On process creation: remove a node (or split a larger one).

On process termination: add the freed region back, possibly merging adjacent free regions.

✅ Checkpoint 4 — Address Translation & Base/Bounds

Test your understanding:

1. Write the formula for translating virtual address VA to physical address PA given base and bounds registers.
2. A process has base = 48 KB, bounds = 16 KB. Translate virtual addresses: 0, 4 KB, 15 KB, 16 KB.
3. Why must the instructions to modify the base and bounds registers be privileged?
4. Trace the sequence of events when a process generates a virtual address of 20 KB with bounds = 16 KB.
5. Describe in three steps how the OS can move a process to a new physical location without the process knowing.

Work through the translation table before proceeding.

§ 6 — OS Issues with Base/Bounds

OS Responsibilities: Overview

While base/bounds is implemented in hardware, the OS must handle five key scenarios:

```
OS Responsibilities for Base/Bounds Memory Management
1. Process creation  — find space, set base/bounds
2. Process termination — reclaim memory, update free list
3. Context switch — save/restore base/bounds in PCB
4. Process relocation — copy memory, update base in PCB
5. Exception handling — respond to out-of-bounds accesses
```

OS Action 1: Process Creation

When a new process is created (`fork()` + `exec()` or a fresh launch):

```
Steps:
1. OS determines required address space size
   (from ELF binary headers, plus stack/heap allocation)

2. OS searches the free list for a slot:
   → First-fit: first region large enough
   → Best-fit: smallest region that fits
   → Worst-fit: largest available region

3. OS removes the slot from the free list (or splits a larger region)

4. OS loads the program binary into the allocated physical memory
   (static sections: .text, .data, .bss)

5. OS sets base register = physical start address of the slot
   OS sets bounds register = size of the address space

6. OS creates a PCB, stores base and bounds in it

7. OS places process in the READY queue
```

OS Action 2: Process Termination

When a process exits or is killed:

```
Steps:
1. OS removes the process from the run queue

2. OS reads base and bounds from the process's PCB:
   base = 32 KB, bounds = 16 KB

3. OS adds the freed region back to the free list:
   free_list.add(start=32 KB, size=16 KB)

4. OS checks if the freed region is adjacent to other free regions:
   → If yes: merge (coalesce) into a single larger free entry
   → Prevents free list fragmentation

5. OS destroys the PCB, closes file descriptors, etc.
```

Coalescing is critical: without it, the free list accumulates many small fragments, and large contiguous allocations become impossible even when total free memory is sufficient.

OS Action 3: Context Switch — Saving and Restoring Base/Bounds

On a context switch, the CPU's one base/bounds pair must be swapped:

```
Switching from Process A to Process B:

OS actions (in the context switch code):

1. SAVE: Read current base/bounds from CPU registers
   pcb_A->base   = cpu_base_register
   pcb_A->bounds = cpu_bounds_register

2. LOAD: Write B's values into CPU registers
   cpu_base_register   = pcb_B->base
   cpu_bounds_register = pcb_B->bounds

3. Switch stack pointer and restore registers

4. Execute return-from-trap → Process B resumes
   (now with B's base/bounds active in hardware)
```

Between steps 1 and 2, the CPU must be in kernel mode (no user process can generate addresses during the switch).

The PCB with Base/Bounds Fields

The Process Control Block must store the memory context:

```
struct proc {
    int      pid;
    enum     procstate state;

    // CPU virtualization context :
    struct   trapframe *trapframe;  // saved user registers
    struct   context   context;    // saved kernel registers

    // Memory virtualization context (NEW – this lecture):
    uint64_t base;    // physical start address of address space
    uint64_t bounds;  // size of address space in bytes

    // Other fields:
    struct file *ofile[NOFILE];
    char         name[16];
};
```

OS Action 4: Moving a Process

The OS can relocate a stopped process to compact memory or accommodate a large new allocation:

```
Move Process A from physical 32 KB to physical 64 KB:

Precondition: Process A is NOT running (state = READY or BLOCKED)

1. OS copies all of A's memory:
   memcpy(physical_64KB, physical_32KB, A.bounds)

2. OS updates A's PCB:
   pcb_A->base = 64 KB    (was 32 KB)
   pcb_A->bounds unchanged

3. OS marks the old region (32 KB for A.bounds bytes) as FREE

4. OS re-adds the old region to the free list

5. When A next runs, its context switch loads base = 64 KB
   → All virtual-to-physical translations adjust automatically
```

The program is oblivious. Its virtual addresses (0 to bounds) are unchanged.

OS Action 5: Exception Handling

At boot time, the OS registers exception handlers in the Interrupt Descriptor Table (IDT):

```
Boot-time setup:

OS (Kernel Mode):
  initialize trap table
  → register handler for: system call, timer, illegal memory access,
                          illegal instruction

  initialize process table (PCB array)
  initialize free list (all physical memory except OS as free)

  start timer interrupt

  load first process → base/bounds set → jump to user space
```

When an out-of-bounds exception fires at runtime, the OS handler delivers **SIGSEGV** to the offending process and terminates it cleanly.

OS Responsibilities Summary Table

Directly from OSTEP Chapter 15:

OS Responsibility	Specific Actions
Memory management	Allocate for new processes via free list; reclaim on termination; manage free list (split, coalesce)
Base/Bounds management	Set base/bounds when creating processes; save/restore during context switch
Exception handling	Register handlers at boot; terminate processes that access out-of-bounds memory

Free List Management: Allocation Policies (Preview)

When the OS searches the free list for a slot, it applies a policy:

First-Fit

Scan from the beginning; use the first region large enough. Fast, but leaves small fragments near the front.

Best-Fit

Find the smallest region that still fits the request. Minimizes wasted space per allocation, but leaves many tiny fragments.

Worst-Fit

Use the largest available region. Leaves larger residual fragments that may be useful later.

There is no universally best policy — it depends on the workload. We revisit these in the context of free-space management.

Fragmentation: A Critical Problem

With variable-sized allocations and deallocations, **external fragmentation** occurs:

```
Physical Memory after several allocations and deallocations:

┌──────────┬──────────┬──────────┬──────────┬──────────┬──────────┐
│    OS    │   FREE   │  Proc A  │   FREE   │  Proc B  │   FREE   │
│  16 KB   │   8 KB   │  16 KB   │   4 KB   │  16 KB   │  20 KB   │
└──────────┴──────────┴──────────┴──────────┴──────────┴──────────┘

Total free memory: 8 + 4 + 20 = 32 KB
A new process needing 24 KB contiguous space: ❌ FAILS!
(20 KB free region not large enough; 32 KB total free but fragmented)
```

Base/bounds requires each process's address space to be **contiguous** in physical memory — so fragmentation is a fundamental problem.

Internal Fragmentation: Wasted Space Inside

Base/bounds also suffers from **internal fragmentation** — wasted space *inside* the allocated region:

```
Process A: address space size = 16 KB
  - Code:         2 KB  (actively used)
  - Static data:  1 KB  (actively used)
  - Heap:         0 KB  (currently empty — will grow later)
  - Stack:        1 KB  (currently used)
  - Free gap:    12 KB  (between heap top and stack bottom)
```

The OS allocated 16 KB of physical memory for Process A. But 12 KB of that allocation is the heap-stack gap — **wasted physical RAM** that Process A isn't using yet.

This internal waste scales badly with large address spaces.

✅ Checkpoint 5 — OS Issues with Base/Bounds

Test your understanding:

1. List the five OS responsibilities in the base/bounds scheme. For each, state *when* the OS performs it.
2. A process has base = 32 KB, bounds = 16 KB. It terminates. Describe the free list update if the region 16–32 KB is already free.
3. Why is coalescing free regions important? What goes wrong if the OS never coalesces?
4. Explain the difference between external fragmentation and internal fragmentation. Give a concrete example of each.
5. If Base is 32 KB = 32,768 bytes: what specific instructions does the OS execute during a context switch to save and restore this value?

§ 7 — Limitations of Base/Bounds & The Path Forward

Limitation 1: Internal Fragmentation at Scale

The single base/bounds pair allocates the **entire address space** contiguously:

```
Modern process address space (64-bit):
  Code:           ~1 MB   (active)
  Static data:    ~1 MB   (active)
  Heap:          ~10 MB   (active, may grow to GBs)
  Stack:          ~8 MB   (active, may grow with recursion)
  Gap (heap↔stack): VAST — up to hundreds of GBs of virtual space
```

Physical allocation needed: the entire contiguous range from the lowest allocated virtual address to the highest allocated virtual address (i.e., from the code/data region to the current stack region), including the **unused** gaps between them → Potentially hundreds of GB of physical RAM wasted!

A 64-bit process with a few MB of actual data would require hundreds of GB of contiguous physical RAM reserved — clearly impossible.

Limitation 2: Address Space Must Fit in Physical Memory

Base/bounds can only virtualize an address space if the **entire address space fits in physical RAM**:

- Virtual address space = 100 MB.
- Physical RAM = 64 MB.
- **Result:** Cannot run this process at all with base/bounds.

Modern programs routinely use virtual address spaces **larger than physical RAM**. Base/bounds provides no solution for this — the OS cannot "spill" part of the address space to disk.

The solution (swapping / demand paging) requires a more sophisticated translation mechanism.

Limitation 3: No Support for Sparse Address Spaces

Real programs have a very **sparse** address space: active data sits at the extremes (low addresses for code/data, high addresses for stack), with a vast empty middle:

```
Virtual Address Space (16 GB):
0 GB    ├── Code + Data (2 MB active)
        │
        │    (14 GB of empty virtual space)
        │
16 GB   ├── Stack (8 MB active, growing downward)
```

Base/bounds forces the OS to treat this as a single 16 GB block. Paging (upcoming lectures) handles sparse spaces gracefully by only allocating physical pages for the ranges actually in use.

Limitation 4: Difficult Code/Data Sharing

A common optimization: multiple processes running the same program (e.g., 100 users running `bash`) should share the **same physical code pages**:

```
Desired:
  Process A (bash): code at physical 32–34 KB  ← shared
  Process B (bash): code at physical 32–34 KB  ← same pages!
  Process A data:   at physical 48–64 KB
  Process B data:   at physical 64–80 KB
```

With a single base/bounds pair, the entire address space (code + data + stack) must be one contiguous block. You cannot have the code at one physical location and data at another — they all share the same base. **Code sharing is impossible with base/bounds.**

Limitation 5: External Fragmentation

Because every process needs a **contiguous physical allocation**, holes in physical memory cannot be used for processes larger than the hole:

```
After several process creations and terminations:
```

OS 16KB	FREE 4KB	A 16KB	FREE 8KB	B 16KB	FREE 4KB	FREE 20KB

```
                    Total free: 36 KB
                    Largest contiguous: 20 KB
New process needing 32 KB contiguous: ❌ Must compact or fail
```

Compaction (moving all processes together to merge free space) is expensive: requires copying all process data and updating base registers.

Quantifying the Waste: A Concrete Scenario

Consider 4 processes in a 128 KB physical RAM system:

Process	AS Size	Actually Used	Wasted (Internal)
A	16 KB	4 KB	12 KB
B	16 KB	6 KB	10 KB
C	16 KB	3 KB	13 KB
D	16 KB	8 KB	8 KB

Total physical RAM used: 4 × 16 KB = **64 KB** (50% of RAM).
Total actually needed: 4 + 6 + 3 + 8 = **21 KB** (16% of RAM).

Base/bounds wastes 67% of the physical RAM in this scenario due to internal fragmentation alone.

The Crux: How to Support Large Address Spaces

> *"How do we support a large address space with (potentially) a lot of free space between the stack and heap?"*
> *— OSTEP, Chapters 16 and 17*

We need a mechanism that:

- Allocates physical memory **only for the parts of the address space actually in use**.
- Supports address spaces **larger than physical RAM**.
- Allows **sharing** of code pages between processes.
- Reduces or eliminates external fragmentation.

Two major advances address this: **Segmentation** and **Paging** (Lectures 2.5 and 2.6).

Where Base/Bounds Is Still Used Today

Despite its limitations, base/bounds (or close variants) appears in real systems:

- **Embedded microcontrollers:** ARM Cortex-M with MPU (Memory Protection Unit) uses region-based protection similar to base/bounds.
- **Hypervisors (Stage-1 and Stage-2 translation):** The outermost level of nested paging in virtualization (Intel VT-x EPT, ARM Stage-2) can use flat base/offset translation for simple guest configurations.
- **Security enclaves:** Intel SGX uses base/bounds-style registers to define the enclave's memory region.
- **Teaching:** Base/bounds is the ideal starting point for understanding the entire virtual memory hierarchy — from this simple mechanism, all others are derived.

✅ Checkpoint 6 — Limitations of Base/Bounds

Test your understanding:

1. A process has a 4 GB virtual address space. Only 20 MB are in active use. How much physical RAM must the OS allocate under base/bounds? How much is wasted?
2. Explain why code sharing between two processes running the same binary is **impossible** with a single base/bounds pair.
3. Define external fragmentation. Give a scenario where 32 KB of free physical memory exists but a 24 KB process cannot be created.
4. Define internal fragmentation. Give a scenario where it wastes more than 50% of allocated physical memory.
5. Which limitation does **segmentation** primarily address? Which limitation does **paging** primarily address?

§ 8 — Worked Examples & Exam Problems

Problem 1: Basic Address Translation

Given:

- Process A: base = 24 KB = 24,576, bounds = 8 KB = 8,192
- Translate the following virtual addresses:

Virtual Address	Step 1: Check VA < bounds	Step 2: Physical Address
0	0 < 8,192 ✅	0 + 24,576 = **24,576**
2,048 (2 KB)	2,048 < 8,192 ✅	2,048 + 24,576 = **26,624**
4,000	4,000 < 8,192 ✅	4,000 + 24,576 = **28,576**
8,000	8,000 < 8,192 ✅	8,000 + 24,576 = **32,576**
8,192	8,192 < 8,192 ❌	**FAULT**
9,000	9,000 < 8,192 ❌	**FAULT**

Problem 1: Key Observations

From the translation table:

- The boundary case at VA = 8,192 (exactly equal to bounds) **faults** — the valid range is `[0, bounds)`, i.e., 0 to bounds−1 inclusive.
- VA = 8,000 is valid (just inside the last valid byte at 8,191).
- VA = 9,000 is invalid — 808 bytes past the end.

Common exam mistake: forgetting that the bounds register holds the *size*, not the last valid address. The last valid virtual address is `bounds - 1 = 8,191`, not `bounds = 8,192`.

Problem 2: Out-of-Bounds Detection

Scenario: A process is running with base = 0 KB, bounds = 16 KB. Its code at virtual address `0x4000` (16,384) contains the instruction:

```
movl $42, 0x4001     ; store 42 at virtual address 0x4001
```

Question: Will this store succeed or fault?

Solution:

- VA = 0x4001 = 16,385
- Bounds check: 16,385 < 16,384? **No** → FAULT

Discussion: Virtual address `0x4000` (16,384) is also out of bounds (since valid range is 0–16,383). The instruction *fetch* itself would have faulted before the store even executes — the instruction at VA=16,384 is the first invalid address. The OS would deliver SIGSEGV.

Problem 3: Context Switch with Base/Bounds

Scenario: The OS is switching from Process A to Process B.

```
Process A PCB: base = 48 KB, bounds = 16 KB
Process B PCB: base = 16 KB, bounds = 16 KB
CPU base register (current): 48 KB
CPU bounds register (current): 16 KB
```

Step-by-step OS actions:

1. **Save Process A's base/bounds** (already in PCB or copied from CPU registers — verify they match).
2. **Load Process B's values:** CPU base register ← 16 KB; CPU bounds register ← 16 KB.
3. **Complete context switch** (restore B's PC, SP, GPRs from B's trapframe).
4. **Return from trap** → CPU now runs B with base=16 KB, bounds=16 KB.

From this moment: every memory access by B is checked and translated using B's base (16 KB).

Problem 4: Free List Management

Scenario: 64 KB physical RAM. OS uses 0–16 KB. Initial free list: `[(16 KB, 16 KB), (32 KB, 16 KB), (48 KB, 16 KB)]`.

Events:

1. Process A (needs 16 KB) created → allocate first-fit → slot (16 KB, 16 KB).
2. Process B (needs 16 KB) created → allocate → slot (32 KB, 16 KB).
3. Process A terminates → free (16 KB, 16 KB).
4. Process C (needs 24 KB) created → needs contiguous 24 KB.

Free list after step 3: `[(16 KB, 16 KB), (48 KB, 16 KB)]`

Step 4: Largest free region = 16 KB. Process C needs 24 KB. **Fails!** Even though 32 KB total free, no contiguous 24 KB block exists. External fragmentation.

Problem 4: After Compaction

To accommodate Process C, the OS compacts memory:

```
Before compaction:
  0–16 KB:  OS
  16–32 KB: FREE
  32–48 KB: Process B
  48–64 KB: FREE

OS action: move Process B from 32 KB to 16 KB:
  1. Copy 16 KB from physical 32 KB to physical 16 KB.
  2. Update B's PCB: base = 16 KB (was 32 KB).
  3. Free list: [(32 KB, 32 KB)] — now one contiguous 32 KB block!

After compaction:
  0–16 KB:  OS
  16–32 KB: Process B  (moved here)
  32–64 KB: FREE (32 KB contiguous)

Process C can now be allocated at 32 KB.
```

Exam Strategy: Memory Translation Problems

For any base/bounds exam question, follow this template:

```
Step 1: Identify Base and Bounds values from the problem.
Step 2: For each virtual address VA:
  a. Check: Is VA ≥ 0? (always true for unsigned)
  b. Check: Is VA < bounds?
     → If NO:  write "FAULT (out of bounds)" and stop.
     → If YES: continue.
  c. Compute: PA = VA + base.
  d. Verify: Base ≤ PA < Base + Bounds (sanity check).
Step 3: Show your work — examiners award partial marks for correct steps.
```

Common Mistakes to Avoid

Mistake	Correct Approach
Using `VA ≤ bounds` for the check	Correct check is `VA < bounds` (strict less-than)
Forgetting to add base even when check passes	Always compute `PA = VA + base`
Confusing VA with offset into segment	In base/bounds, VA *is* the offset — no segment bits
Stating bounds register holds the end address	In OSTEP's default, bounds = *size*, not end address
Forgetting the OS saves base/bounds on context switch	Base/bounds are part of the process context
Applying one process's base to another's addresses	Each process has its own saved base/bounds in its PCB

§ 9 — Summary & Key Takeaways

Summary: Ch. 13 — The Address Space Abstraction

The **address space** is the OS abstraction that makes memory virtualization possible:

- Contains: code (.text), static data (.bss/.data), heap (↑), stack (↓).
- Every address a program generates is a **virtual address** — never a physical one.
- Goals: **Transparency** (invisible to program), **Efficiency** (hardware-aided), **Protection** (isolation).
- On 64-bit systems: 256 TiB user space + 256 TiB kernel space per process.
- The same virtual address in two processes maps to **different physical locations** — full isolation.

Summary: Ch. 14 — The Memory API

The C memory API (from an OS perspective):

- `malloc()` / `calloc()` / `realloc()` are **library functions**, not system calls — they manage a heap pool, calling `brk()` or `mmap()` when the pool is exhausted.
- `free()` returns memory to the allocator; the OS reclaims physical pages only when the allocator releases them (or process terminates).
- Critical errors: forgetting to allocate, buffer overflow, memory leak, dangling pointer, double-free, invalid free.
- Detection tools: **Valgrind** (runtime instrumentation), **AddressSanitizer** (compiler instrumentation).

Summary: Ch. 15 — Base/Bounds Mechanism

The base/bounds (dynamic relocation) mechanism:

$$\text{physical address} = \text{virtual address} + \text{base}$$

$$\text{check: } 0 \leq \text{VA} < \text{bounds (else: exception)}$$

Component	Role
Base register	Holds physical start address of process's AS
Bounds register	Holds size of AS — enforces containment
MMU	Performs translation + check on every memory access
OS (boot)	Registers exception handlers, initializes free list
OS (creation)	Finds free slot, sets base/bounds in PCB
OS (termination)	Returns slot to free list, coalesces
OS (context switch)	Saves/restores base/bounds to/from PCB

Key Formulas and Facts

Address translation:

$$\text{PA} = \text{VA} + \text{base}$$

Validity check (size-based bounds):

$$\text{valid if: } 0 \leq \text{VA} < \text{bounds}$$

Address space sizes:

$$\text{32-bit VA space} = 2^{32} = 4\ \text{GiB}$$

$$\text{64-bit VA space (implemented)} = 2^{48} = 256\ \text{TiB}$$

Free list: linked list of `(start_address, size)` pairs; grows on termination, shrinks on allocation.

Fragmentation types:

- *Internal:* wasted space inside allocated region (heap-stack gap).
- *External:* free regions too scattered for large contiguous allocations.

Looking Ahead: Lecture 2.5 — Segmentation

We have established that base/bounds is a powerful first step but suffers from internal and external fragmentation.

In Lecture 2.5, we extend the model to **Segmentation**:

- Instead of one base/bounds pair for the entire address space, each *segment* (code, heap, stack) gets its own pair.
- Each segment is relocated independently in physical memory.
- The heap-stack gap consumes **zero** physical RAM.
- This directly addresses the internal fragmentation problem.
- New challenges: variable-sized segments, external fragmentation, segment growth.

Reading: OSTEP Chapter 16 (Segmentation)

Recommended Exercises

Before Lecture 2.5, practice:

1. Given base = 56 KB, bounds = 8 KB, translate: VA = 0, 1 KB, 4 KB, 7,999, 8 KB, 10 KB. Identify faults.
2. Draw the physical memory layout for 3 processes in 64 KB RAM (OS uses 0–8 KB). Show free list.
3. Process B (base=16 KB, bounds=16 KB) terminates. Process A (base=32 KB) needs to grow from 16 KB to 24 KB. Can it? What must the OS do?
4. Write pseudocode for the OS context switch that saves and restores base/bounds.
5. Calculate internal fragmentation for a 32 KB address space where code=2 KB, data=1 KB, heap=0, stack=2 KB are active.

Lecture 2.4 Complete

Recommended Reading

- **OSTEP Ch. 13:** *The Abstraction: Address Spaces*
- **OSTEP Ch. 14:** *Interlude: Memory API*
- **OSTEP Ch. 15:** *Mechanism: Address Translation*
- 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!