Main Memory

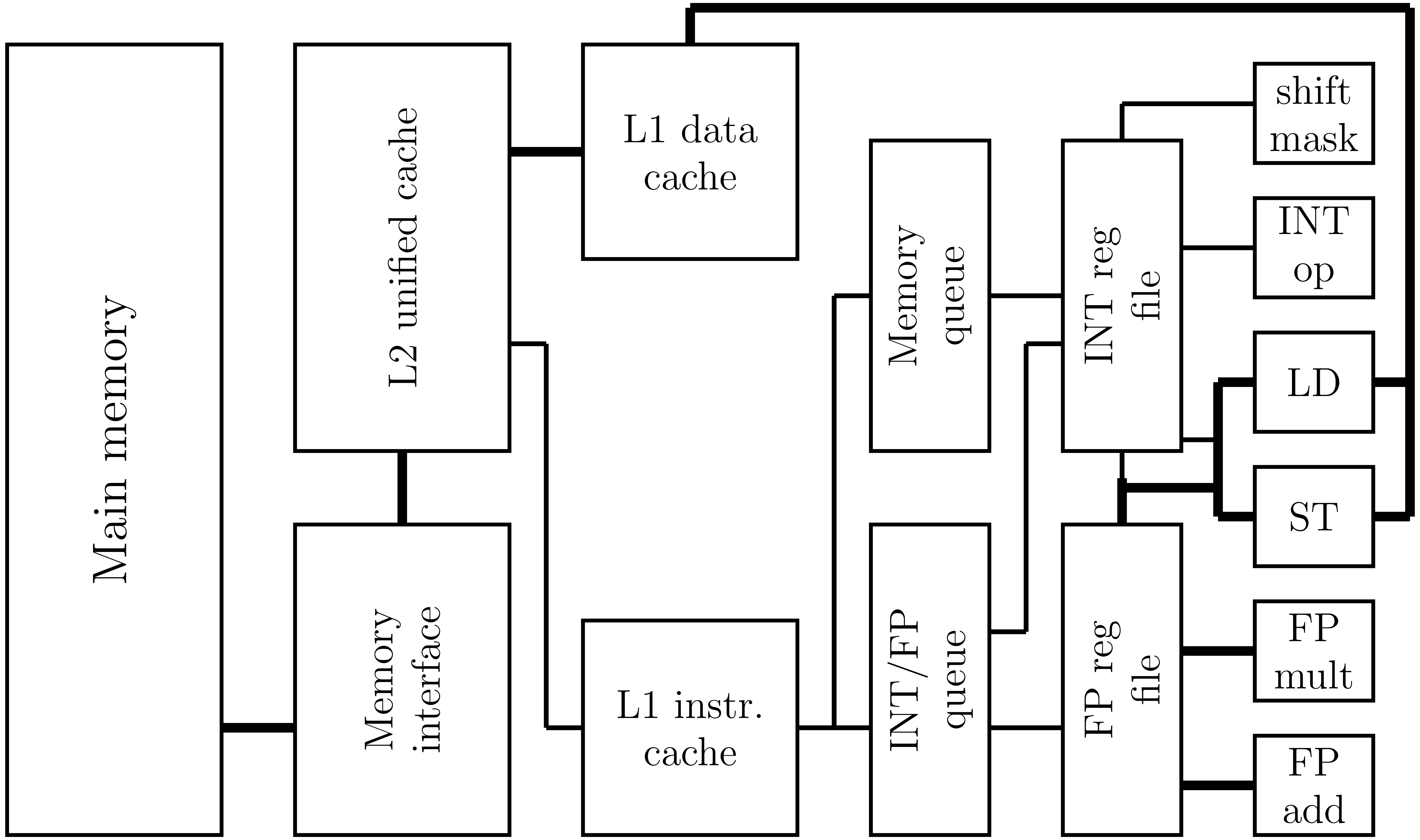
01/31/2023

Professor Amanda Bienz

Background

- Program starts in disk
  - Brought into memory and placed within a process for it to be run
- Register access is done in one CPU clock cycle (or less)
- Main memory can take many cycles (hundreds) - causing a **stall**
- **Cache :** sits between main memory and CPU registers

Cache-Based Microprocessor

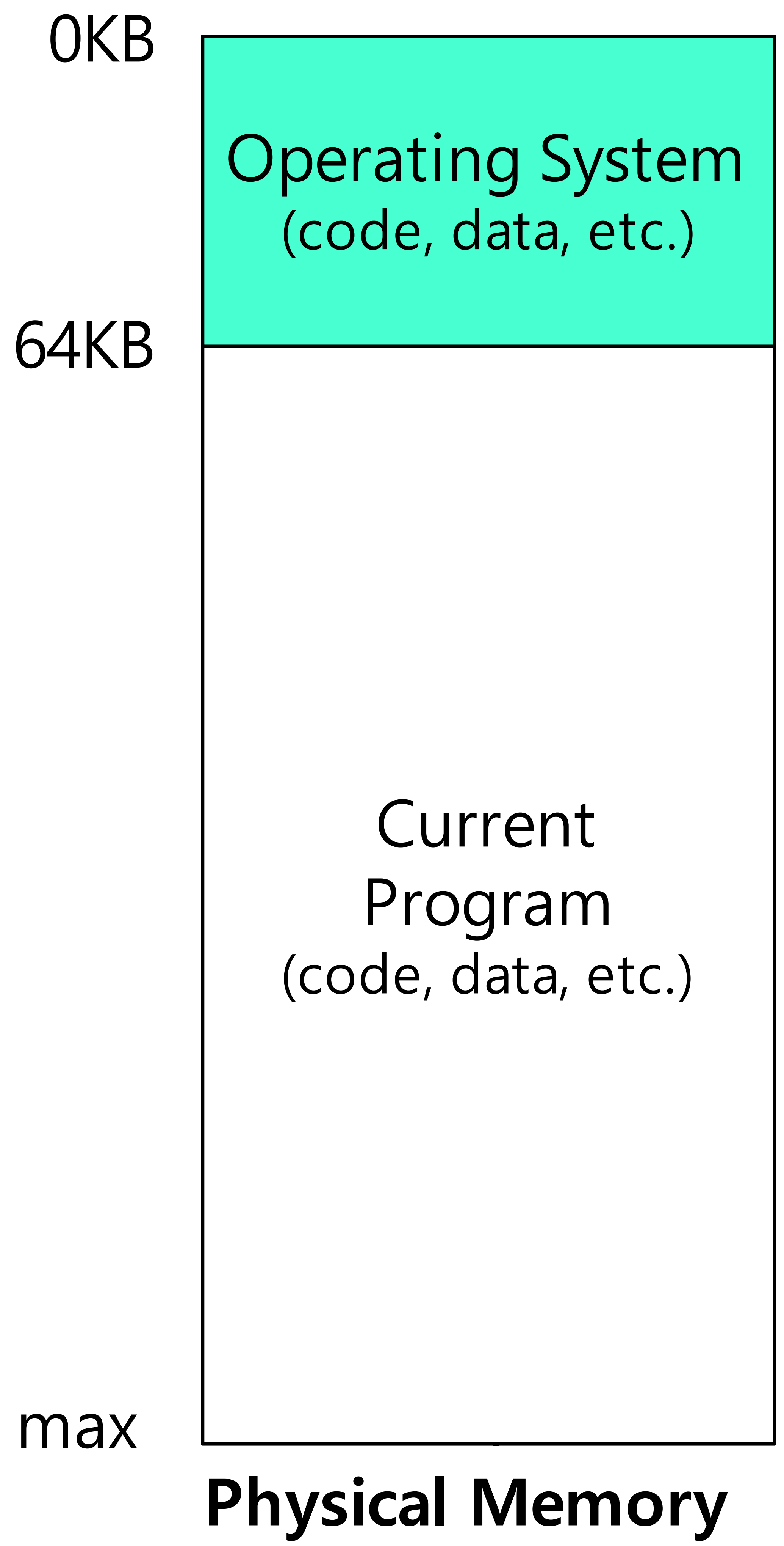


Memory Virtualization

- OS virtualizes physical memory, providing illusion of separate memory space per process
- Seems like each process uses entire memory space
- **Benefits :**
  - Ease of programming
  - Memory efficiency
  - Guarantee isolation for user processes and OS (protection)

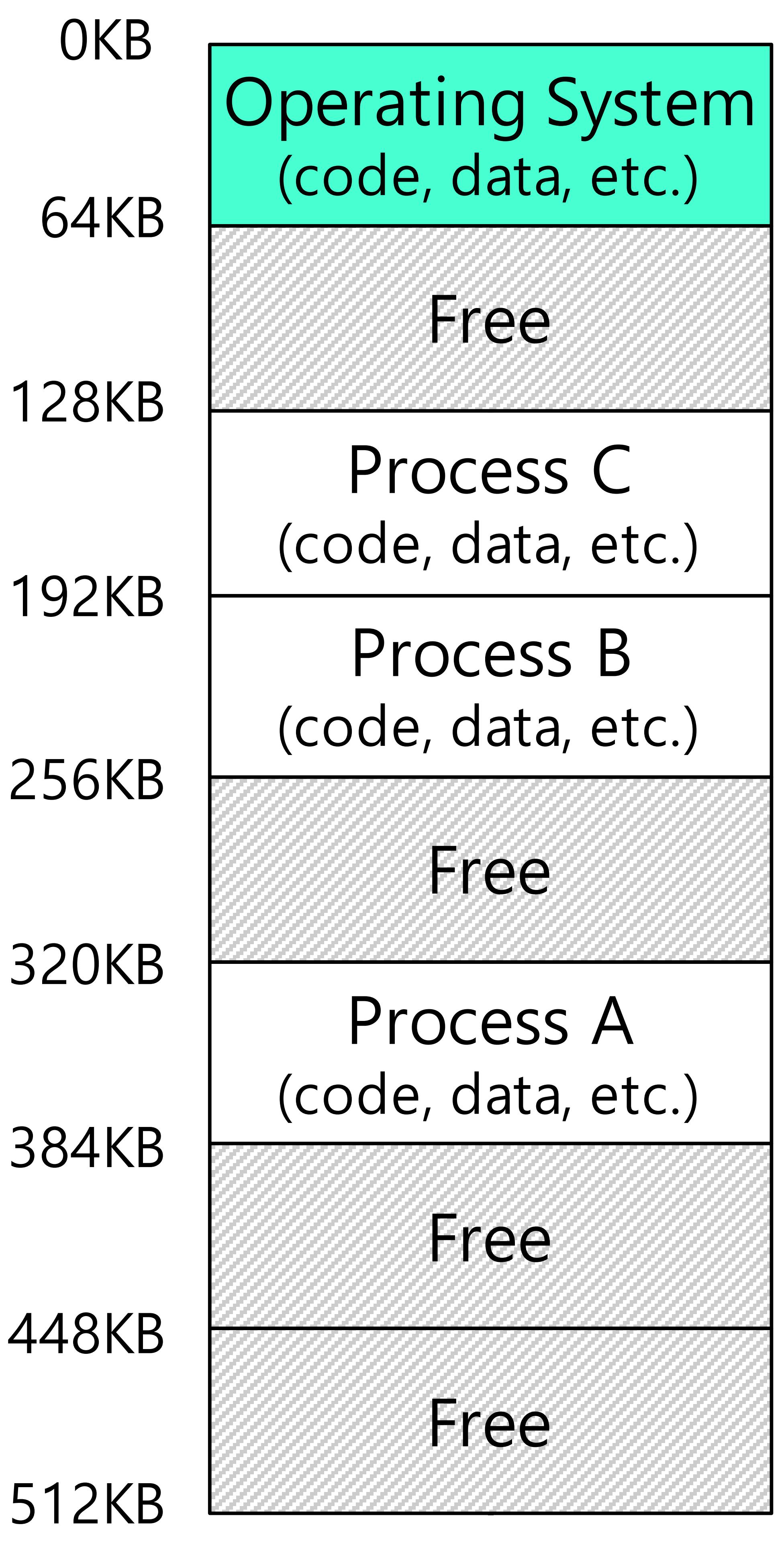
Early Operating Systems

- Load a single process in memory
  - Poor utilization and efficiency



Multiprogramming and Time Sharing

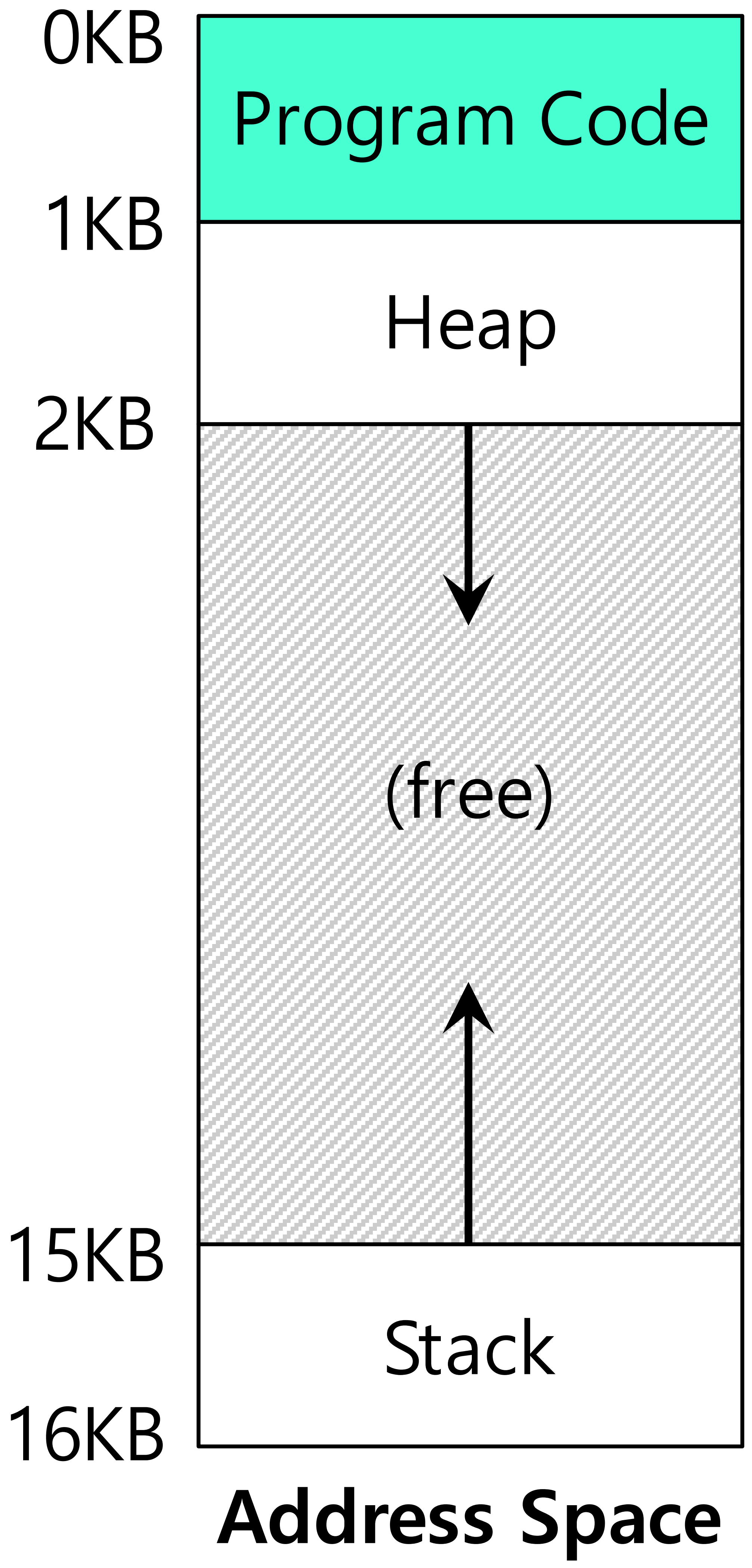
- Load multiple processes in memory
  - Execute one for short time
  - Switch between processes in memory
  - Increase utilization and efficiency
- Protection issue:
  - Errant memory accesses from other processes



Physical Memory

Address Space

- OS creates abstraction of physical memory
  - Address space contains info for a running process
  - Consist of program code, heap, stack, etc
- Text/Data : where instruction and global variables live
- Heap : dynamically allocated memory (malloc or new)
- Stack : return addresses/values, local variables
- We can print out what the addresses are for each



Address Binding

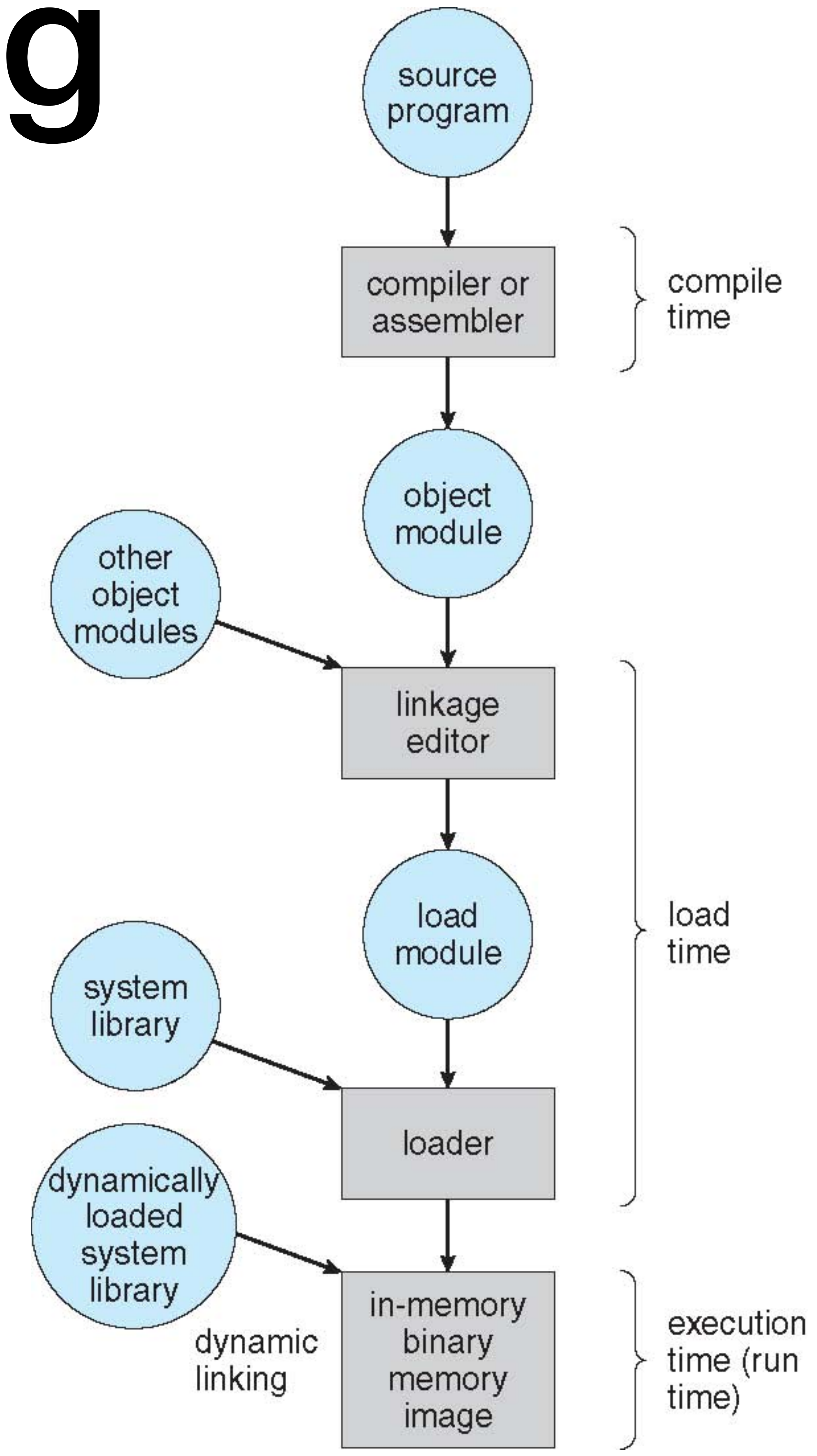
- Program resides on disk as a binary executable file
- Must be brought into memory and placed within the context of a process
    - Then, eligible for execution on an available CPU
- Process executes - reads instructions and data from memory
- When process finishes, memory is reclaimed for other processes

Address Binding

- **Address binding :** mapping from one address space to another
- Addresses in source program are typically symbolic
  - i.e. the variable count
- Compiler binds these symbolic addresses to relocatable addresses
  - i.e. 14 bytes from beginning of this module
- Linker or loader binds relocatable addresses to absolute addresses

Address Binding

- Address binding of instructions and data to memory addresses can happen at three different stages:
  - **Compile time:** If memory location known a priori, **absolute code** can be generated… must recompile code if starting location changes
  - **Load time:** must generate **relocatable code** if memory location is not known at compile time
  - **Execution time:** binding delayed until run time if the process can be moved during its execution from one memory segment to another



Logical vs Physical Address Space

- **Logical address (virtual address) :** generated by CPU
- **Physical address :** address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes
    - Differ in execution-time address binding
- **Logical address space :** set of all logical addresses generated by a program
- **Physical address space :** the group of physical addresses mapped to logical address space

Address Translation

- Address spaces are virtual addresses
  - Must be transparently translated to actual physical memory addresses used by the underlying hardware
- Hardware transforms virtual address to a physical address
- OS must get involved at key points to set up hardware

Example Address Translation

```
void func()
        int x;
        ...
        x = x + 3; // this is the line of code we are interested in
```

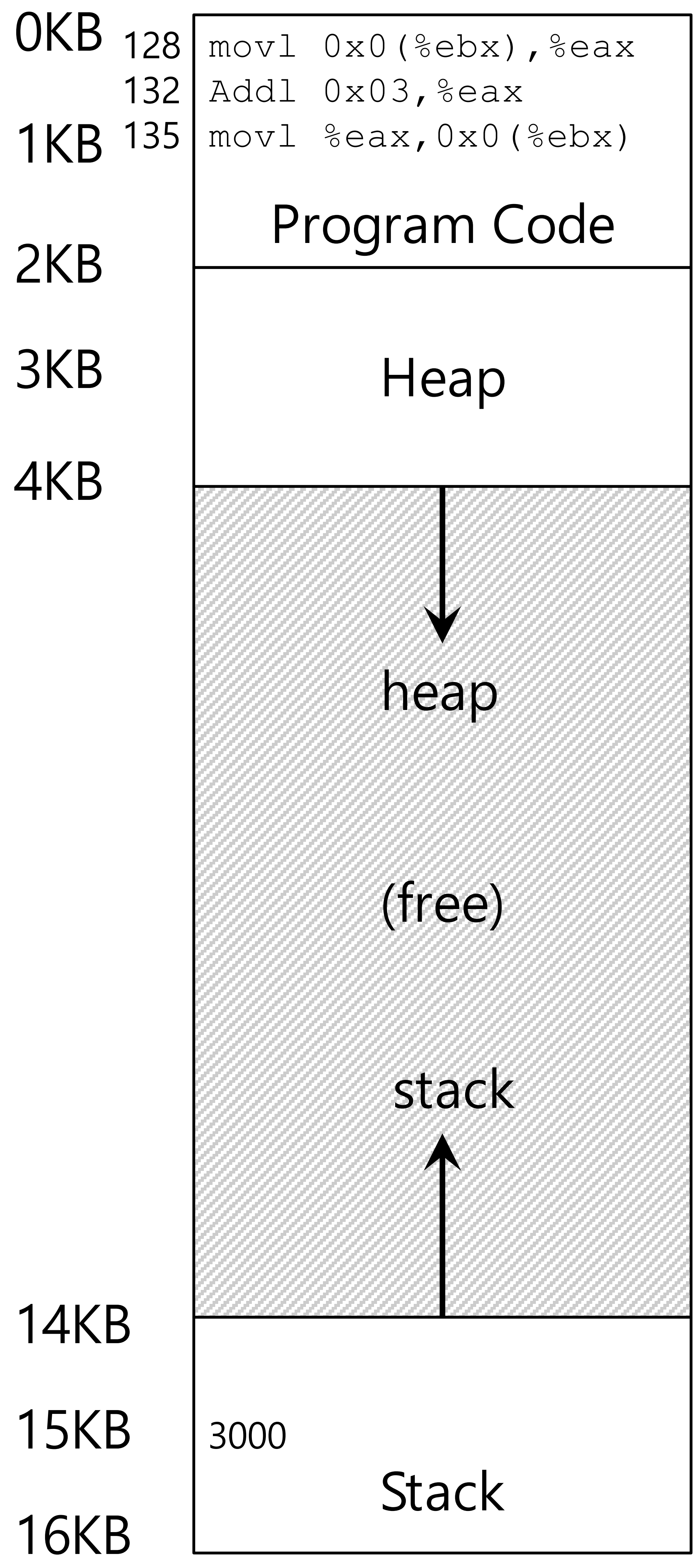
- Example in C :
  - Load a value from memory
  - Increment value by 3
  - Store value back into memory

Example Address Translation

```
128 : movl 0x0(%ebx), %eax          ; load 0+ebx into eax
132 : addl $0x03, %eax              ; add 3 to eax register
135 : movl %eax, 0x0(%ebx)          ; store eax back to mem
```

- Example in assembly :
  - Presume that address of ‘x’ has been placed in ebx register
  - Load value at that address into eax register
  - Add 3 to eax register
  - Store value in eax back into memory

Example Address Translation



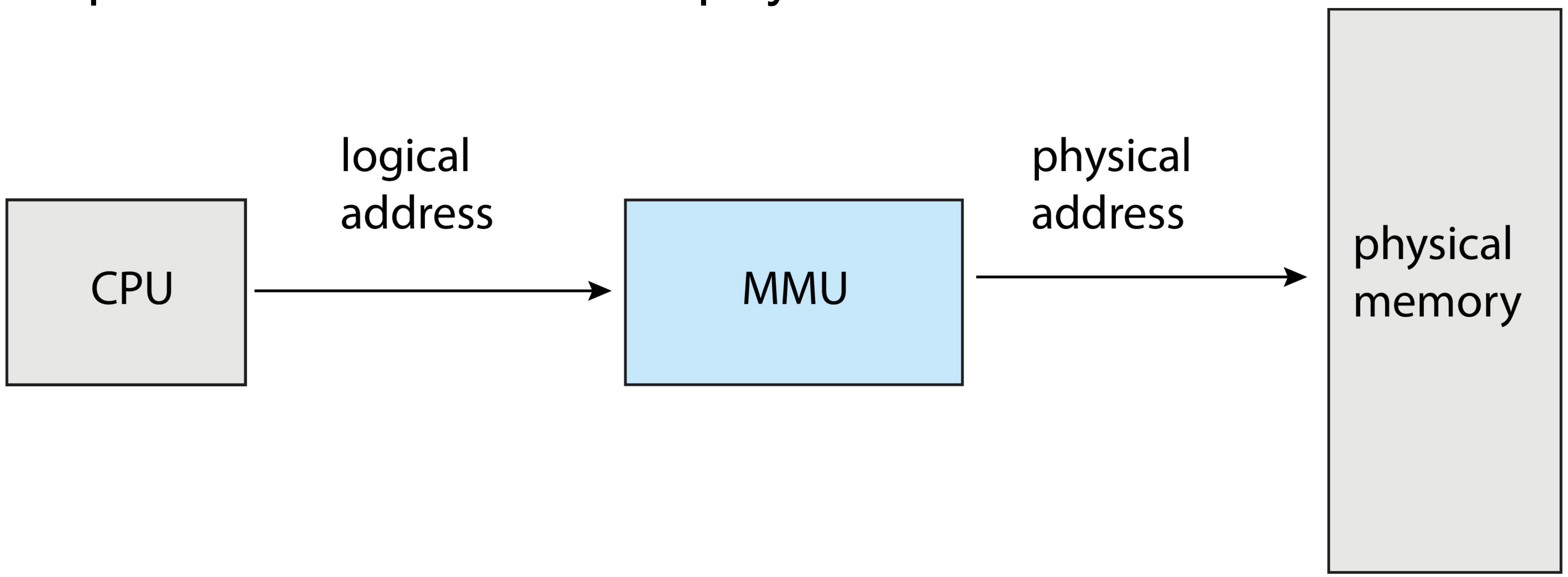
- Fetch instruction at address 128
- Execute this instruction (load from address 15kb)
- Fetch instruction at address 132
- Execute this instruction
- Fetch instruction at address 135
- Execution this instruction (store to address 15kb)

Relocation Address Space

- OS wants to place process somewhere else in physical memory (not at address 0)
- Virtual memory can start at address 0, but correspond to different location in physical memory

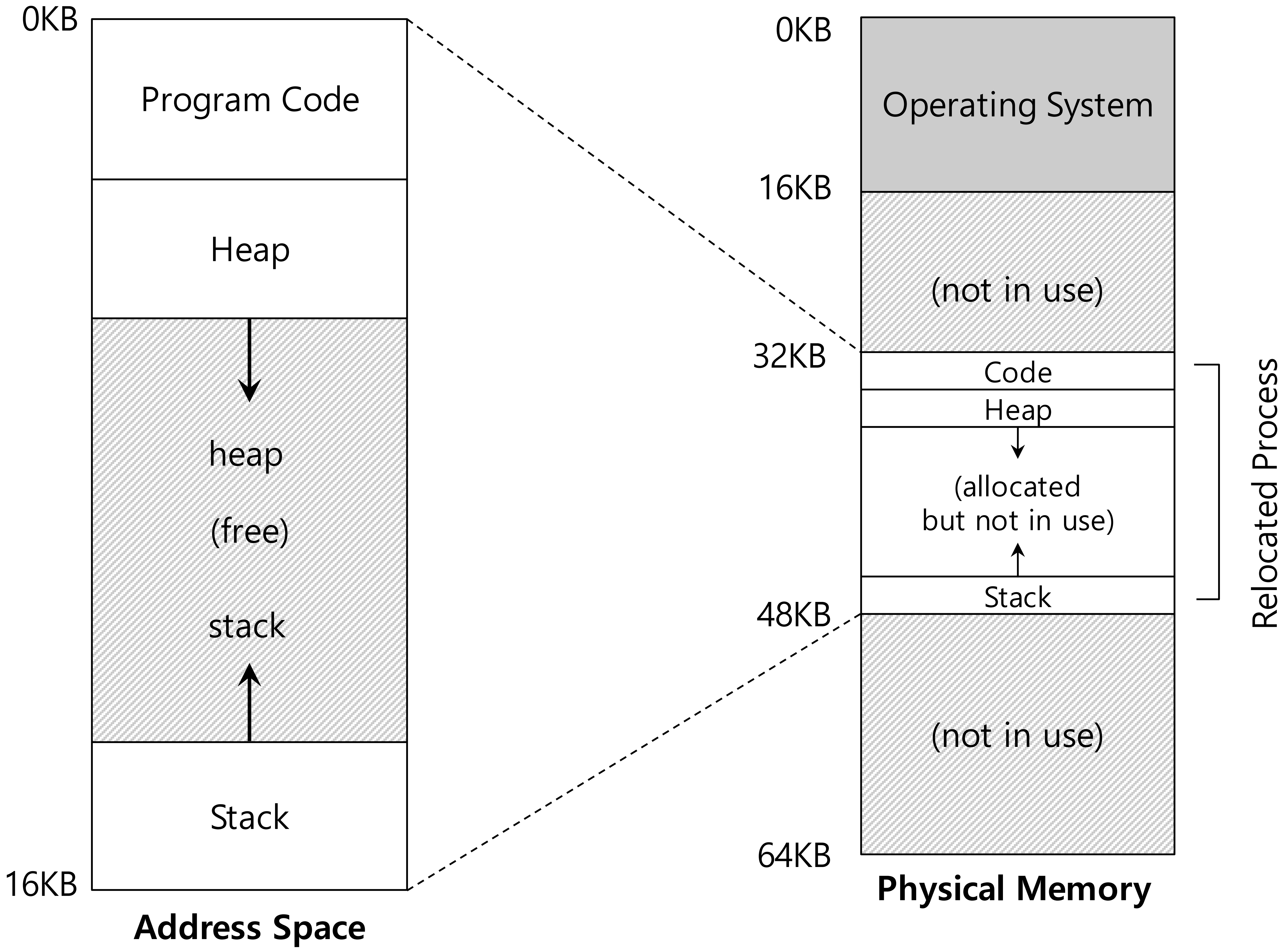
Mapping Physical to Virtual Memory

- **Memory-Management Unit (MMU) :** hardware device that at run time maps virtual addresses to physical address



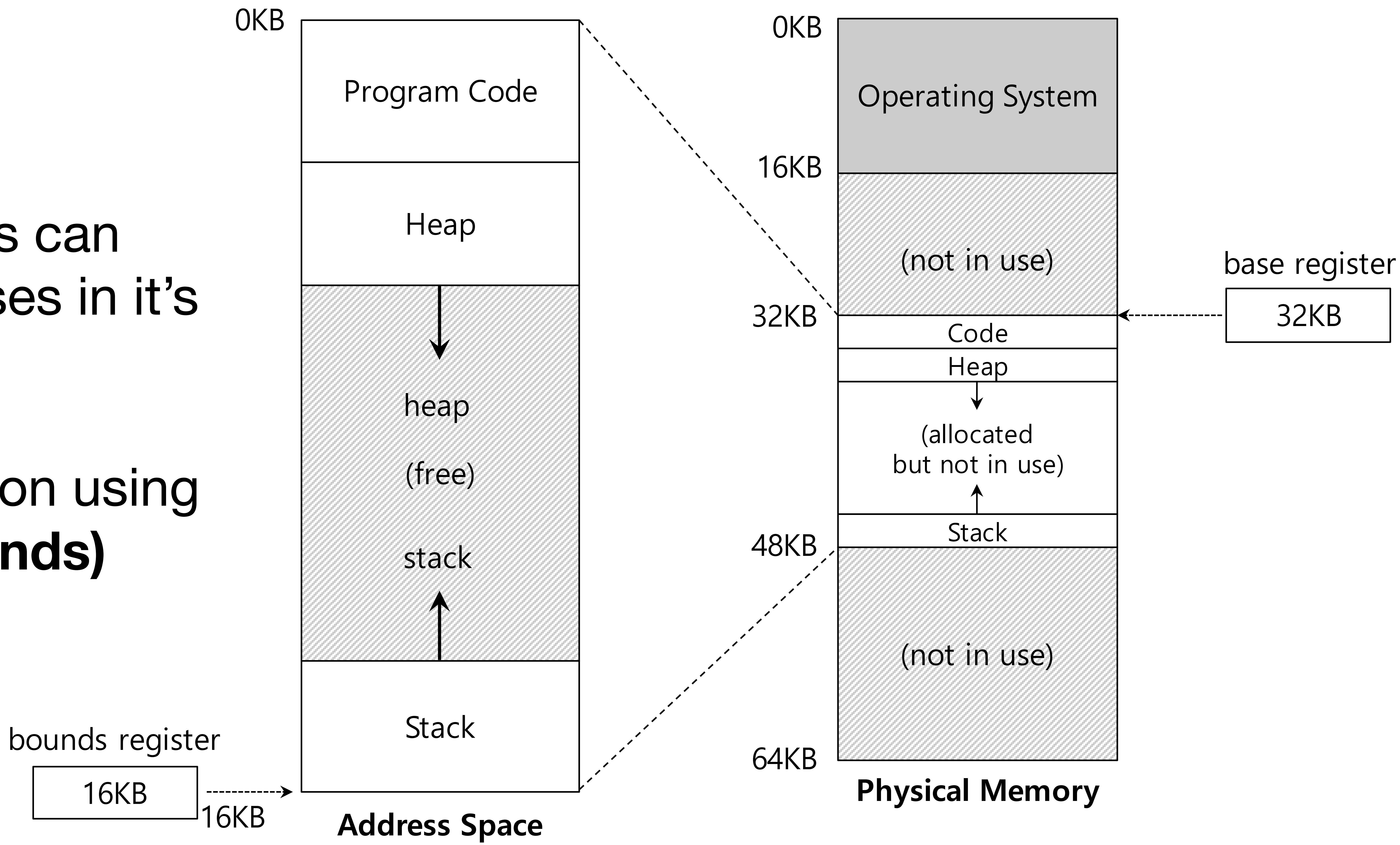
- Many methods possible for doing this

Relocated Process



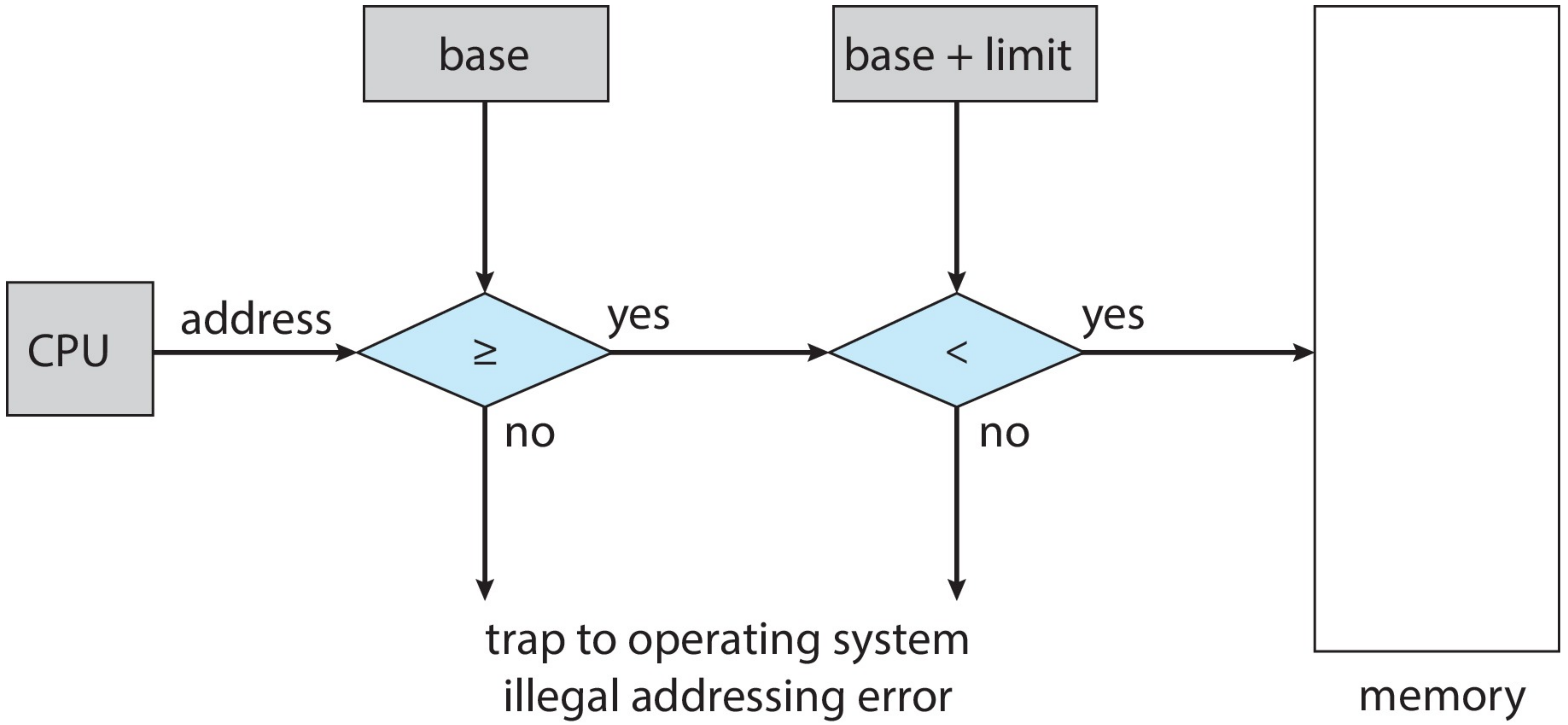
Base and Limit Registers

- Must ensure process can only access addresses in it's address space
- Provide this protection using **base** and **limit (bounds) registers**



Base and Limit Registers

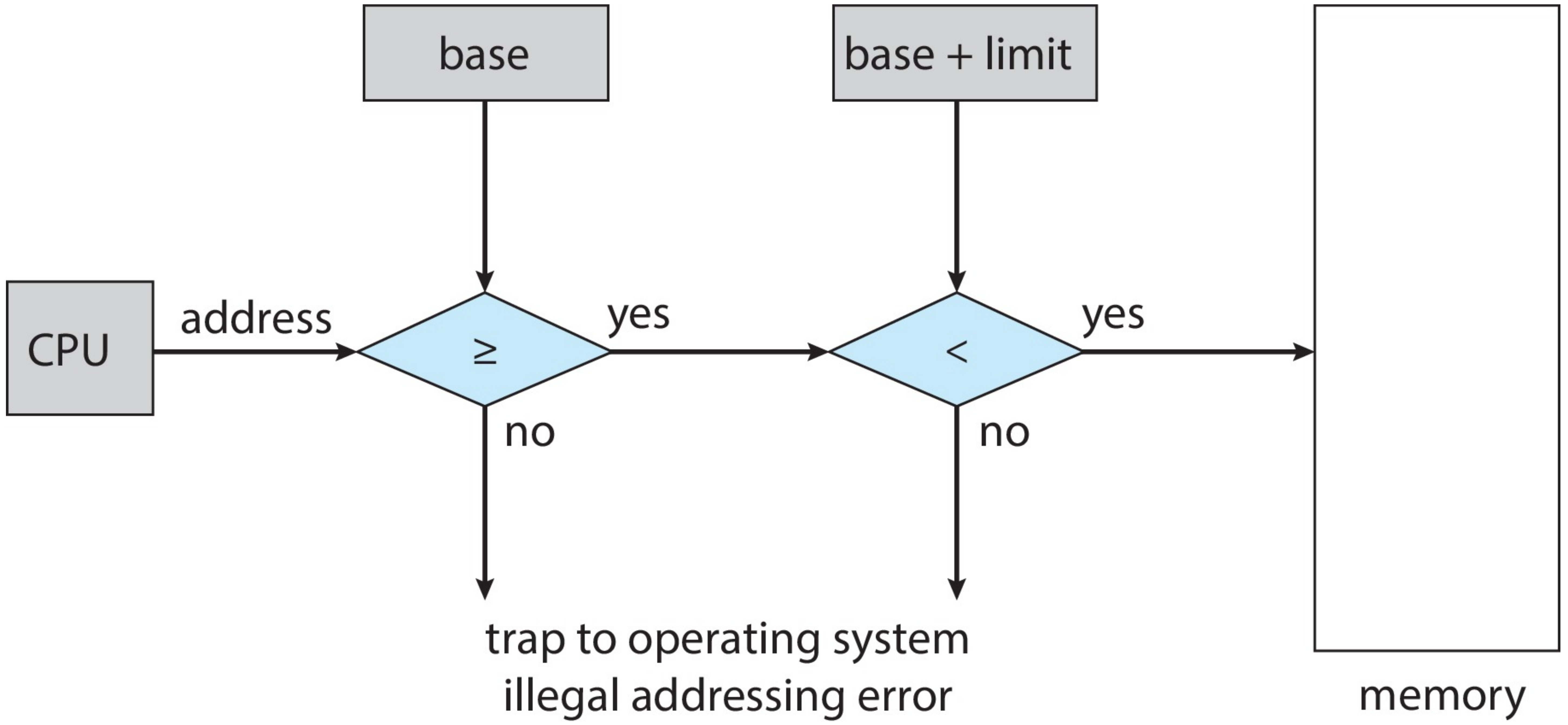
- CPU checks every memory access generated in user mode to ensure it's between base and base+limit



- What operation is privileged (requires kernel mode)?

Base and Limit Registers

- CPU checks every memory access generated in user mode to ensure it's between base and base+limit



- What operation is privileged (requires kernel mode)? **Loading base and limit registers**

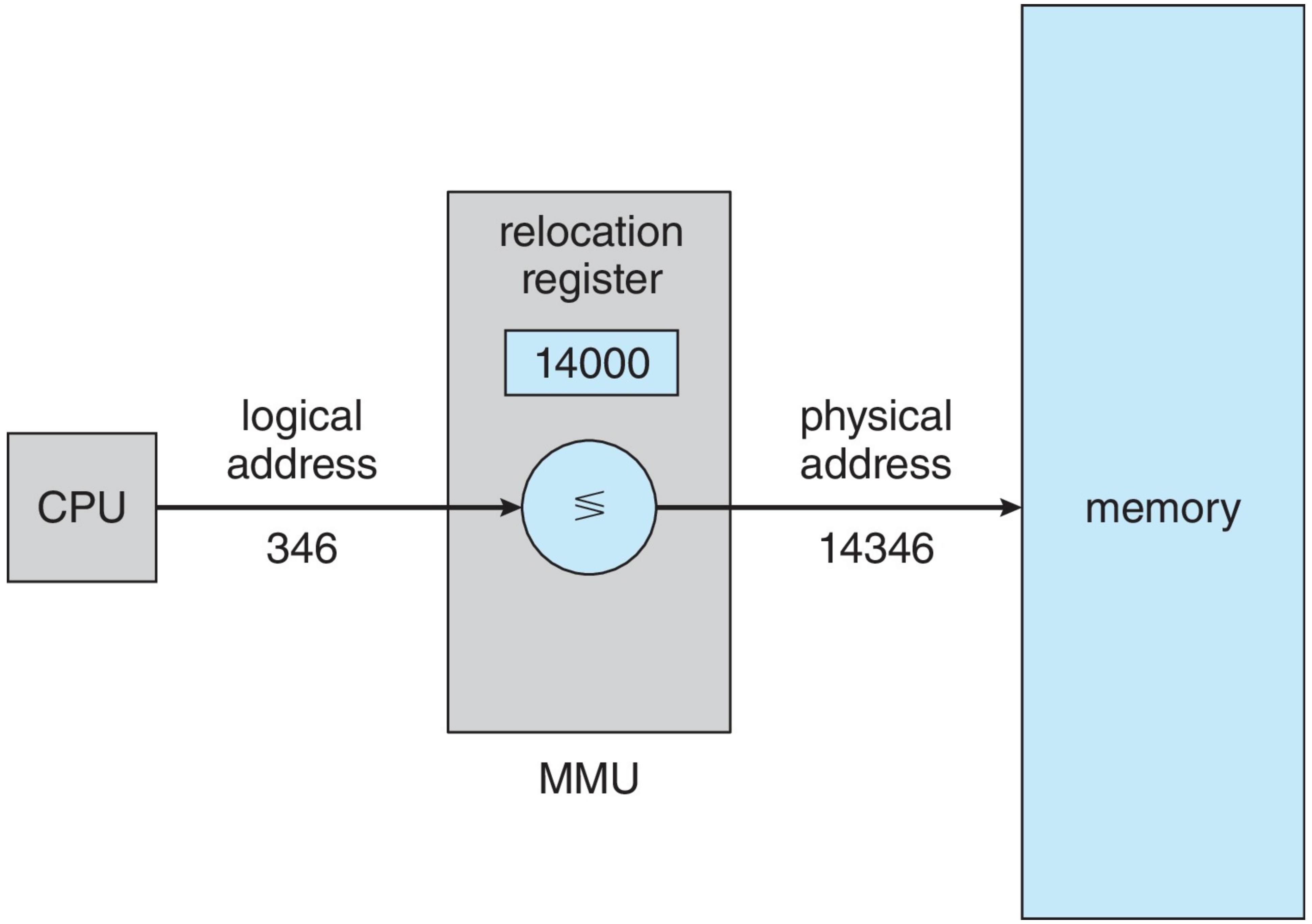
Dynamic (Hardware Base) Relocation

- When a program starts running, OS decides where in physical memory a process should be loaded
  - Set **base** register :
    - physical address = virtual address + base
  - Every virtual address must not be great than bound or negative:
    - 0 <= virtual address < limit

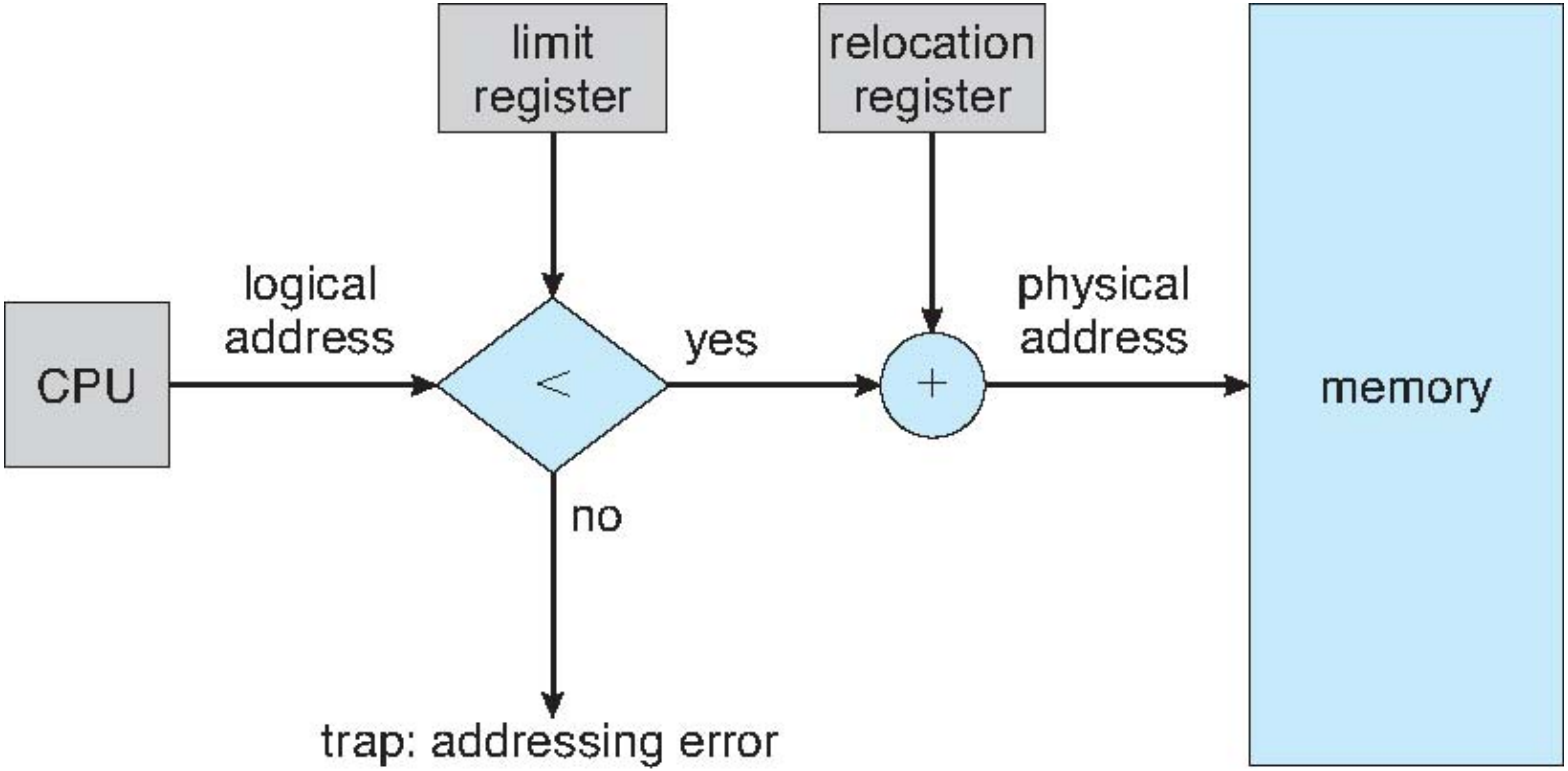
Memory-Management Unit (MMU)

- **Relocation register :** the value is added to every address generated by a user process at the time it is sent to memory

- User program deals with **logical** addresses (never sees real physical addresses)



Hardware Support for Relocation



Relocation and Address Translation

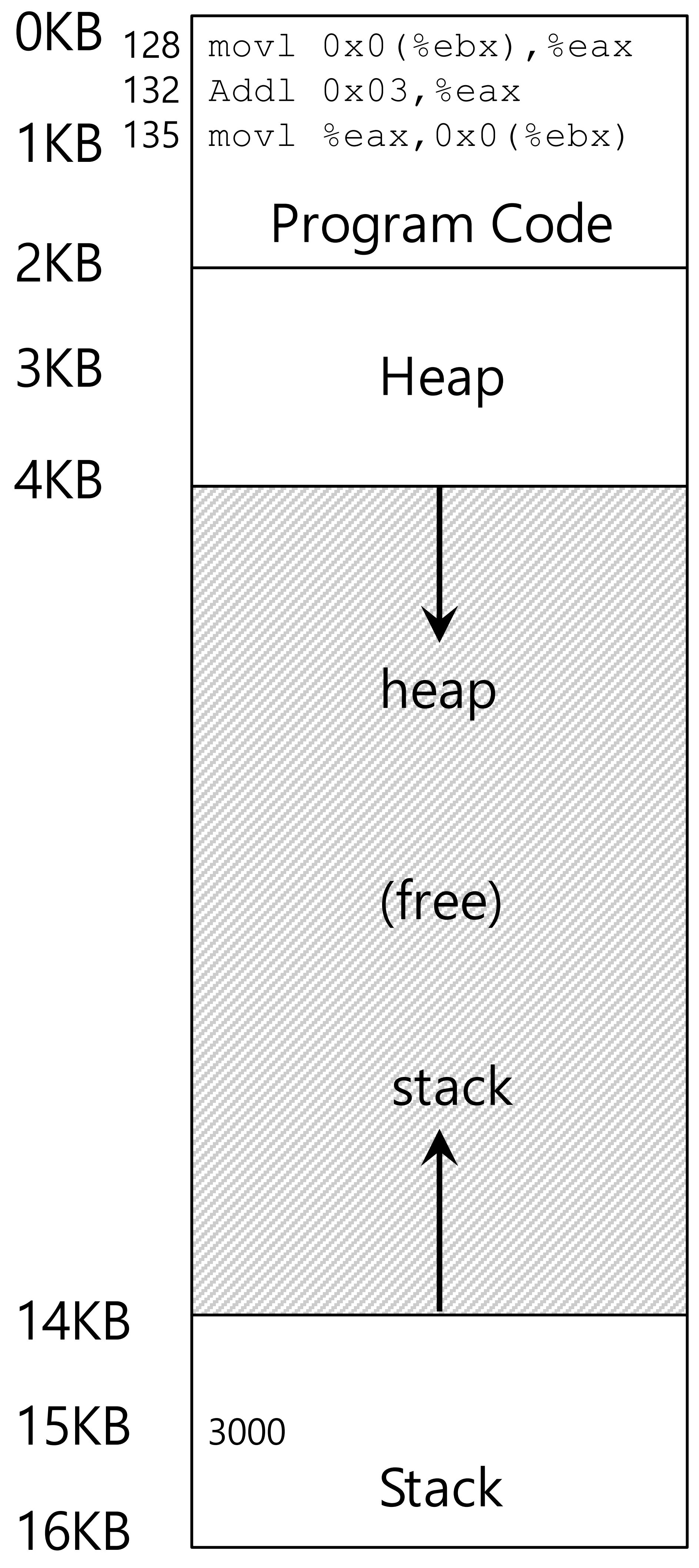
```
128 : movl 0x0(%ebx), %eax
```

- **Fetch** instruction at address 128

$$32896 = 128 + 32KB(base)$$

- **Execute** this instruction
  - Load from address 15KB

$$47KB = 15KB + 32KB(base)$$



Dynamic Loading

- Entire program does not need to be in memory to execute
- Routine is not loaded until it is called
- Better memory-space utilization : unused routine is never loaded
- All routines kept on disk in relocatable load format
- When is this useful?

Dynamic Loading

- Entire program does not need to be in memory to execute
- Routine is not loaded until it is called
- Better memory-space utilization : unused routine is never loaded
- All routines kept on disk in relocatable load format
- When is this useful? **When large amounts of code are needed to handle infrequently occurring cases**

Dynamic Linking

- **Static Linking -** system libraries and program code combined by the loader into the binary program image
- **Dynamic linking -** linking postponed until execution
  - Small piece of code (stub) used to locate the appropriate library routine in memory
  - Stub replaces itself with the address of the routine and executes
  - OS checks if routine is in process's memory (if not, add to address space)
  - **Shared library -** can be shared between many processes
- How is it possible to shared library if it resides in other process's memory?

Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource - must be allocated efficiently… contiguous allocation is one early method
- Main memory : usually into two partitions:
    - Resident operating system, usually held in low memory with interrupt vector
    - User processes : held in high memory
    - Each process contained in single contiguous section of memory

Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing OS code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses - each logical address must be less than the base + limit register
  - MMU maps logical address dynamically

Memory API : malloc()

- Main programmer-level API in C
- Language-level interface (not actual OS interface)
- Allocate a memory region on the heap
  - Argument: **Unsigned Integer**
    - size_t size : size of memory block (int bytes)
  - Return:
    - Success : a void type pointer to the memory block allocated by malloc
    - Fail : a null pointer
- Similar methods : free, calloc, realloc

Many ways to go wrong with memory

- What are some issues you have had with memory in the past?

Many ways to go wrong with memory

- Not copying enough data (terminating nulls)
- Not allocating space for the data copied
- Not freeing data
- Accessing data after its freed
- Freeing an area multiple times
- **What actually happens in each case? Requires understanding of how C API is built on top of OS memory API**

Memory Management System Calls

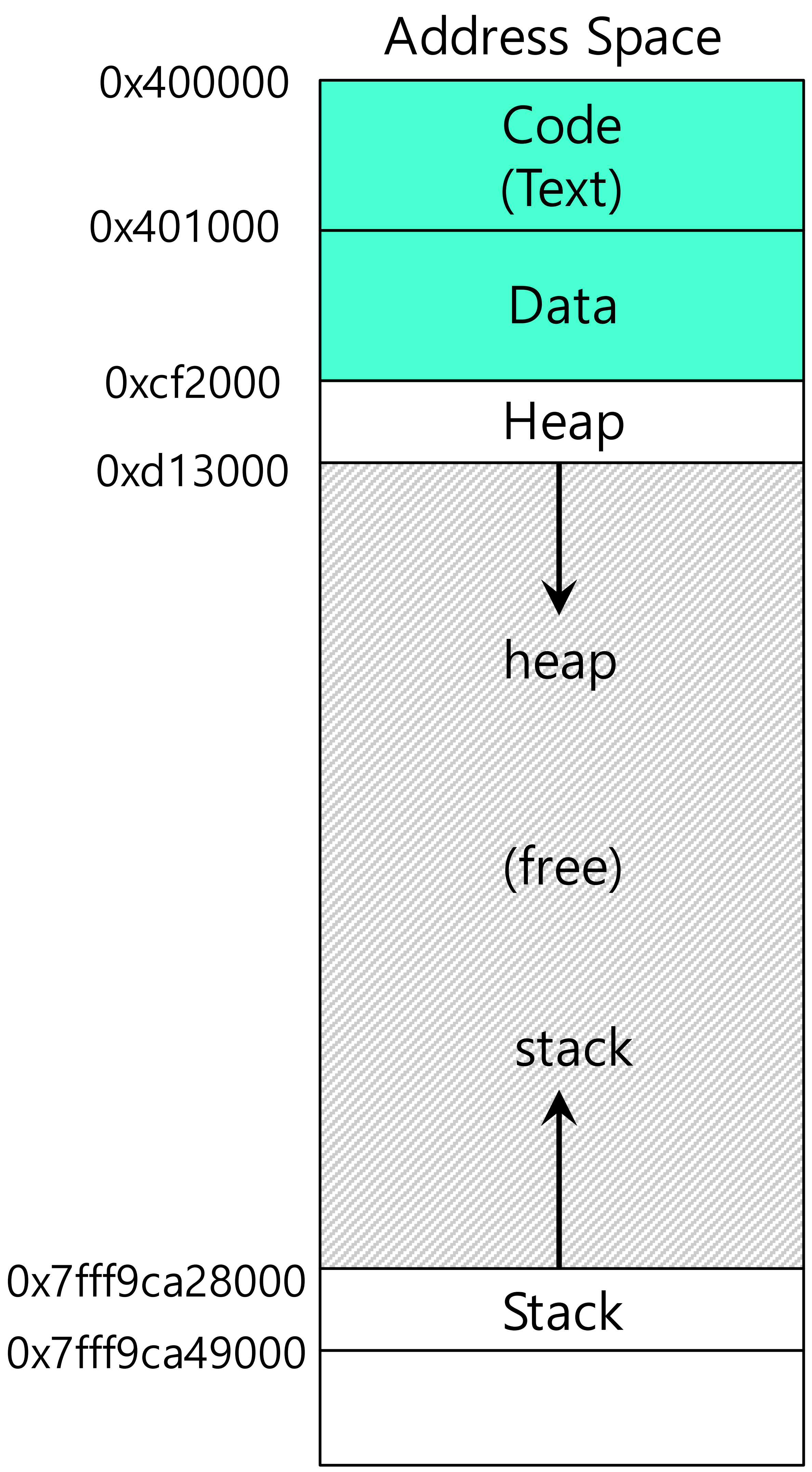
```
#include <unistd.h>

int brk(void *addr)
void *sbrk(intptr_t increment);
```

- Malloc library call uses brk and mmap system calls
    - brk : called to expand the program's break
        - Break : the location of the end of the heap in address space
    - sbrk : additional call similar to brk
    - Programmers should never directly call either brk or sbrk
- What does this method actually do?

About brk and sbrk

- Greyed-out area is not actually allocated
- To use it, OS has to make it available
- The methods brk/sbrk set the location of the boundary between allocated heap memory and unallocated memory



System Calls (Cont.)

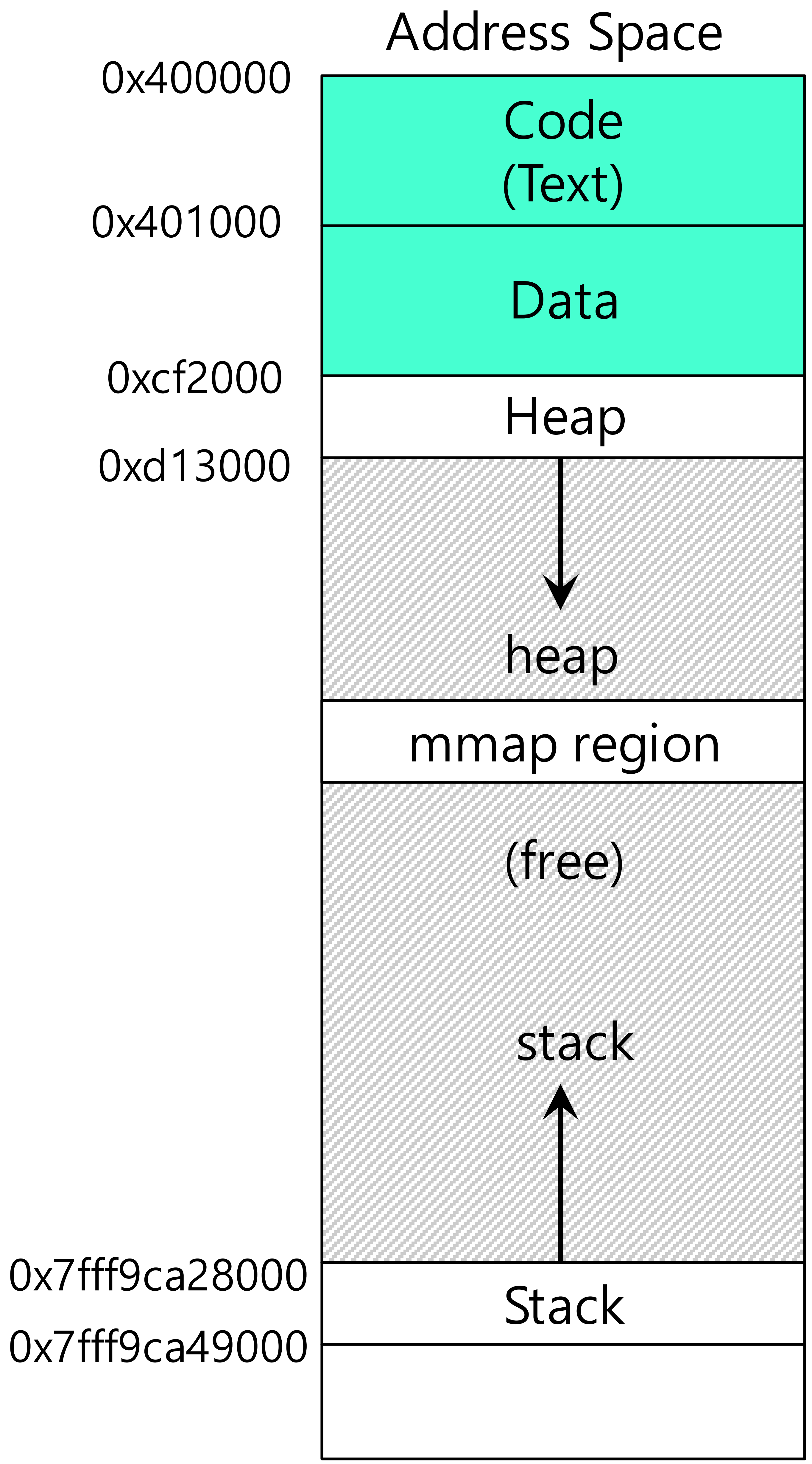
```
#include <sys/mman.h>

void *mmap(void *ptr, size_t length, int port, int flags,
int fd, off_t offset)
```

- The system call *mmap* creates an anonymous memory region

About mmap

- The system call mmap lets the program request fine-grain allocation of parts of its address space
- More than just moving the program break
- Note that the address space is now disjoint
- Mmap can do other methods (implicit file I/O) that we will talk about later



Malloc/New Implementation

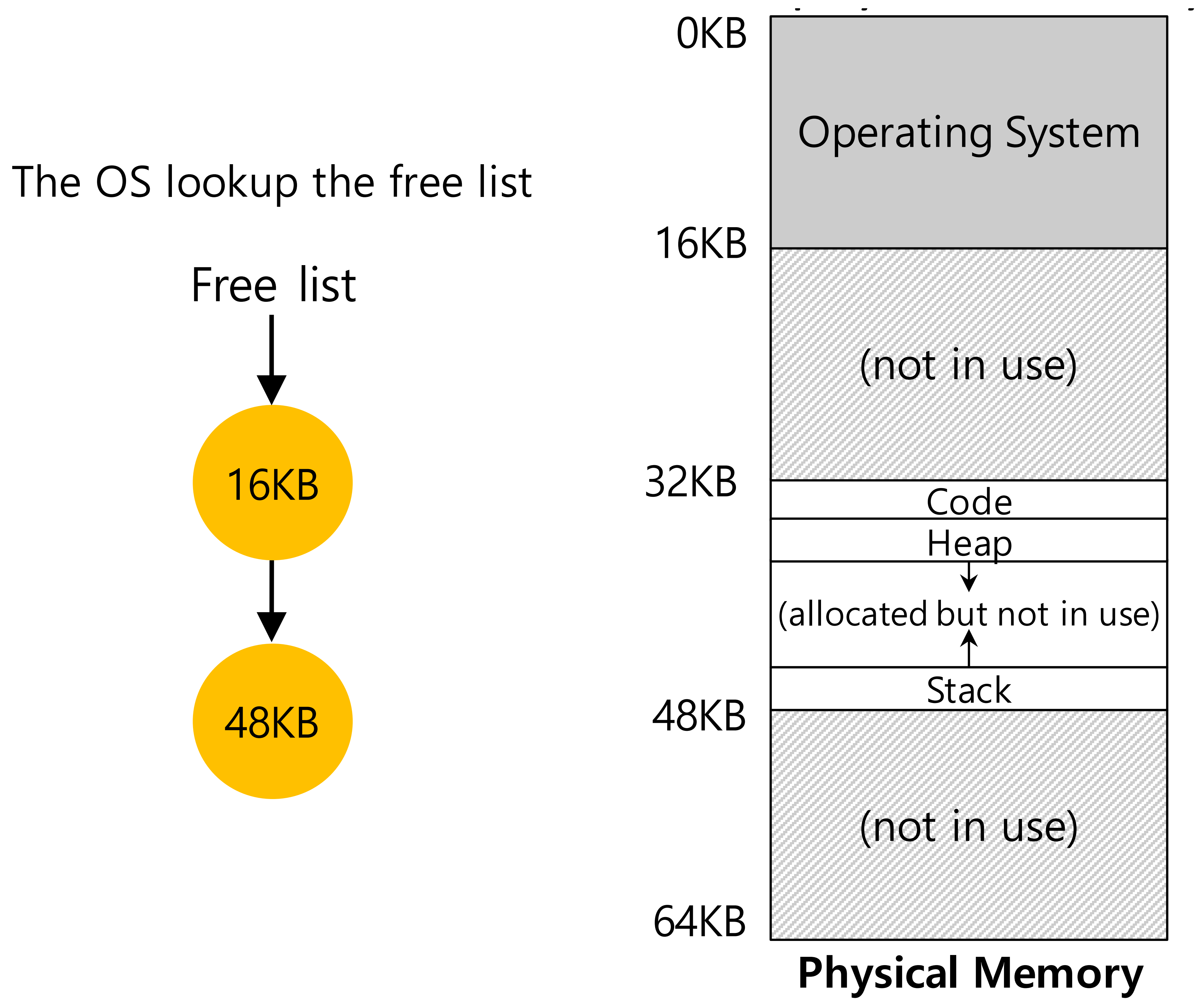
- Language runtime (libc) gets memory in chunks from OS
  - Uses mmap or sbrk to get chunks for 4k bytes or more at a time
  - Why not smaller pieces?
- Language runtime divides these big blocks up to satisfy malloc/new requests
  - Basic data structure is a ‘free list’ (linked list of free chunks of memory)
  - Malloc/new search list to find a chunk that satisfy an allocation request
  - Free returns things to this linked list

OS Issues for Virtualizing Memory

- OS must take action to implement base-and-bounds approach
- Three critical junctures :
  - When a process **starts running :**
    - Finding space for virtual address space in physical memory
  - When a process is **terminated :**
    - Reclaiming the memory for use
  - When **context switch occurs :**
    - Saving and storing base and bounds pair

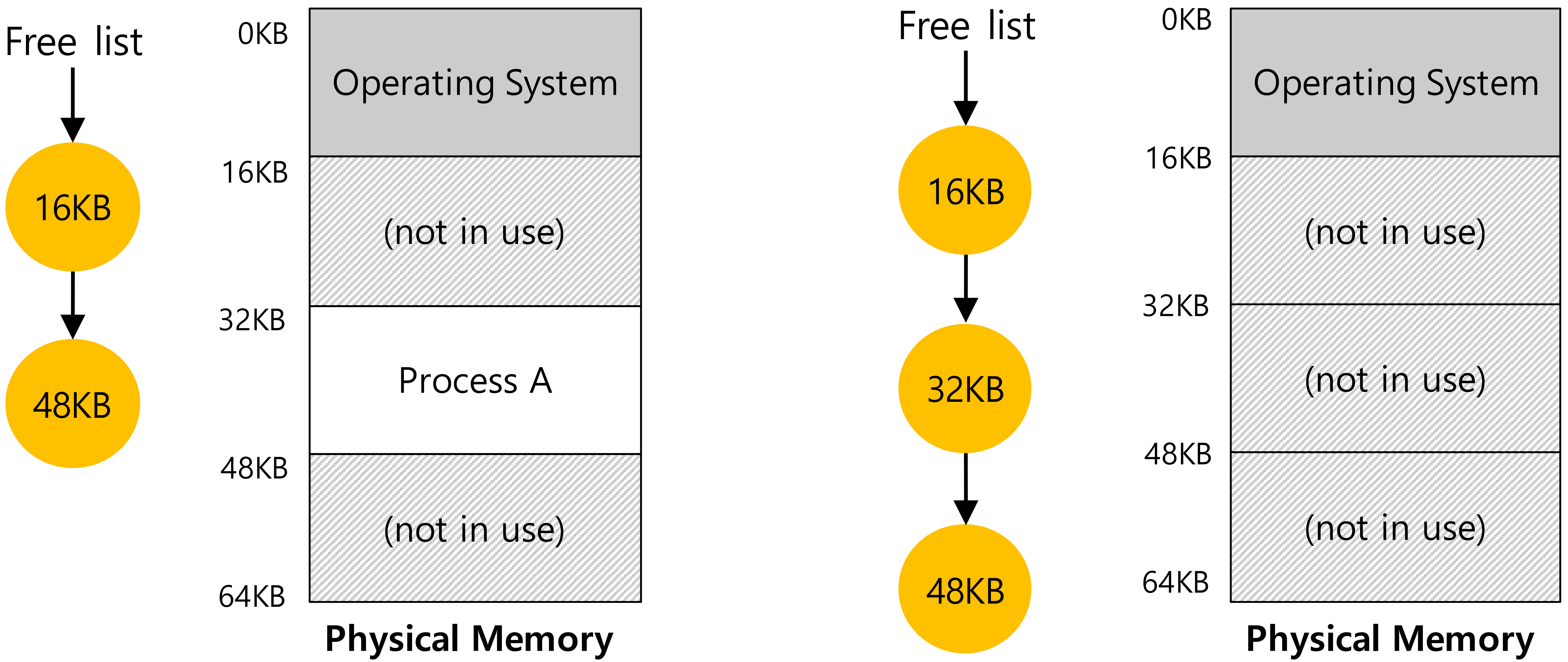
OS Issues : When a process starts running

- OS must find a room for the new address space
  - Free list : a list of the range of physical memory which are not in use



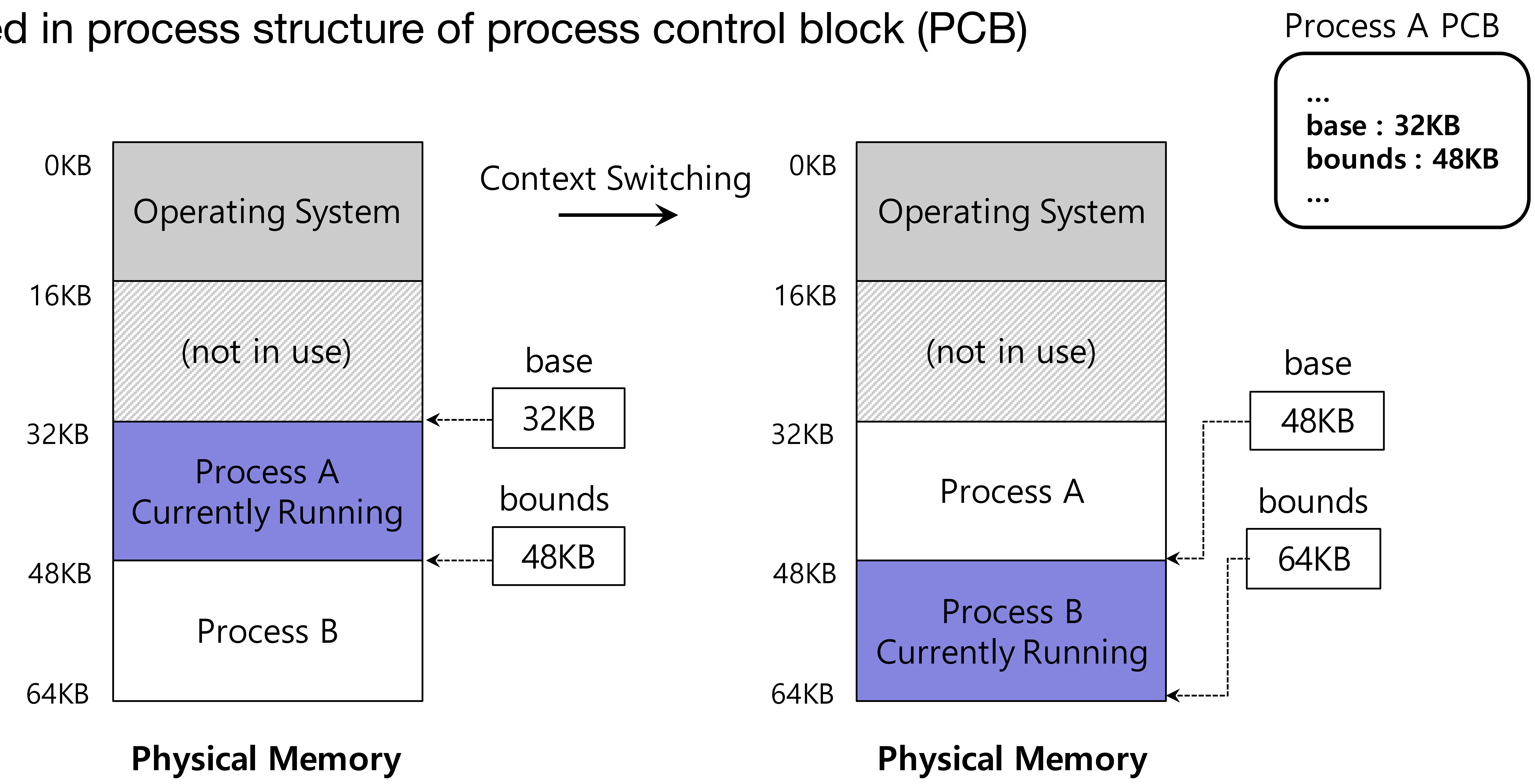
OS Issues : When a Process is Terminated

- The OS must put the memory back in the free list



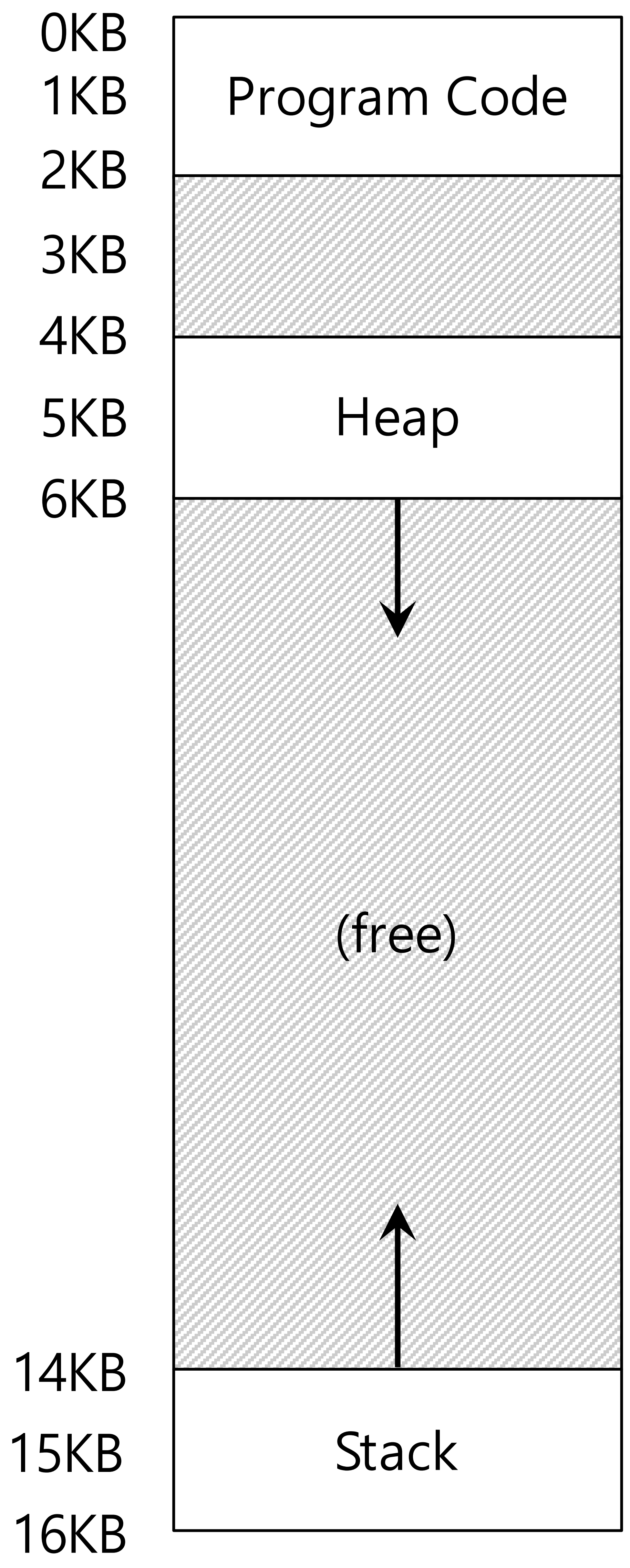
OS Issues : Context Switch

- OS must save and restore base and bounds pair
  - Saved in process structure of process control block (PCB)



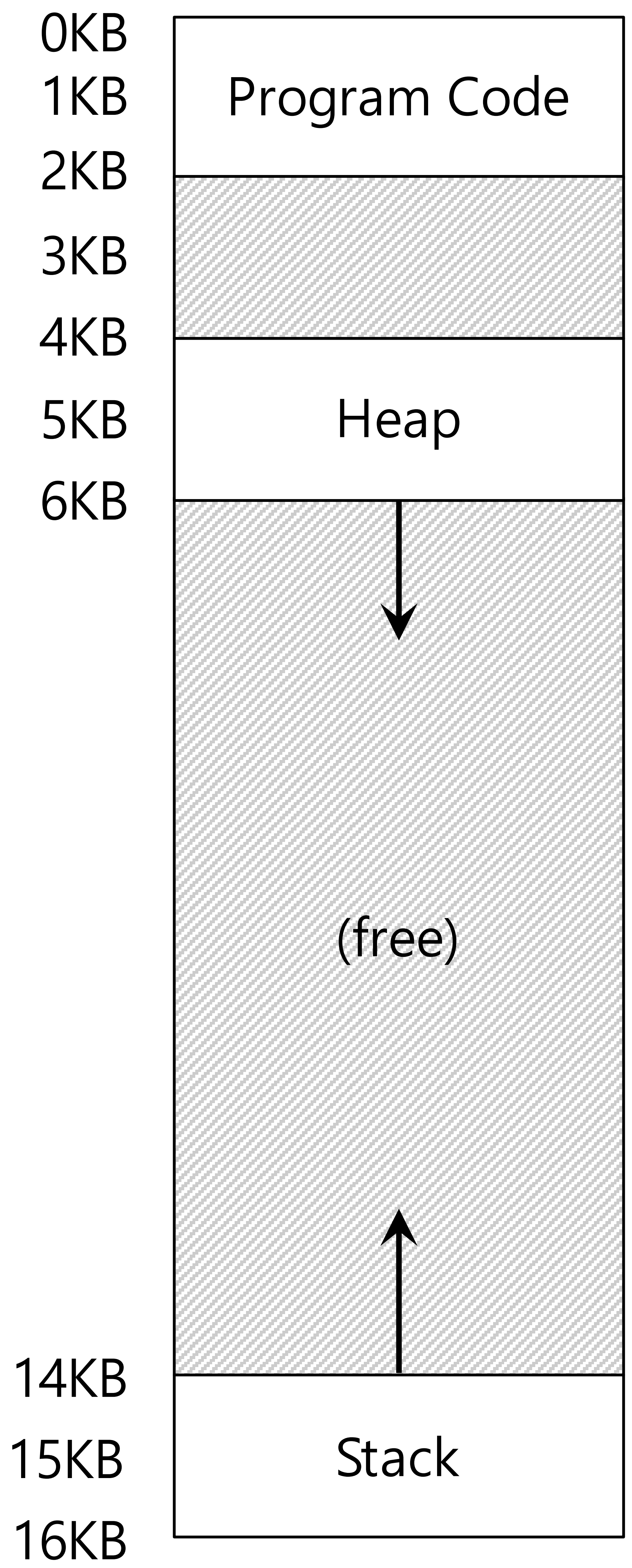
Segmentation

- Why not just base and bound?



Segmentation

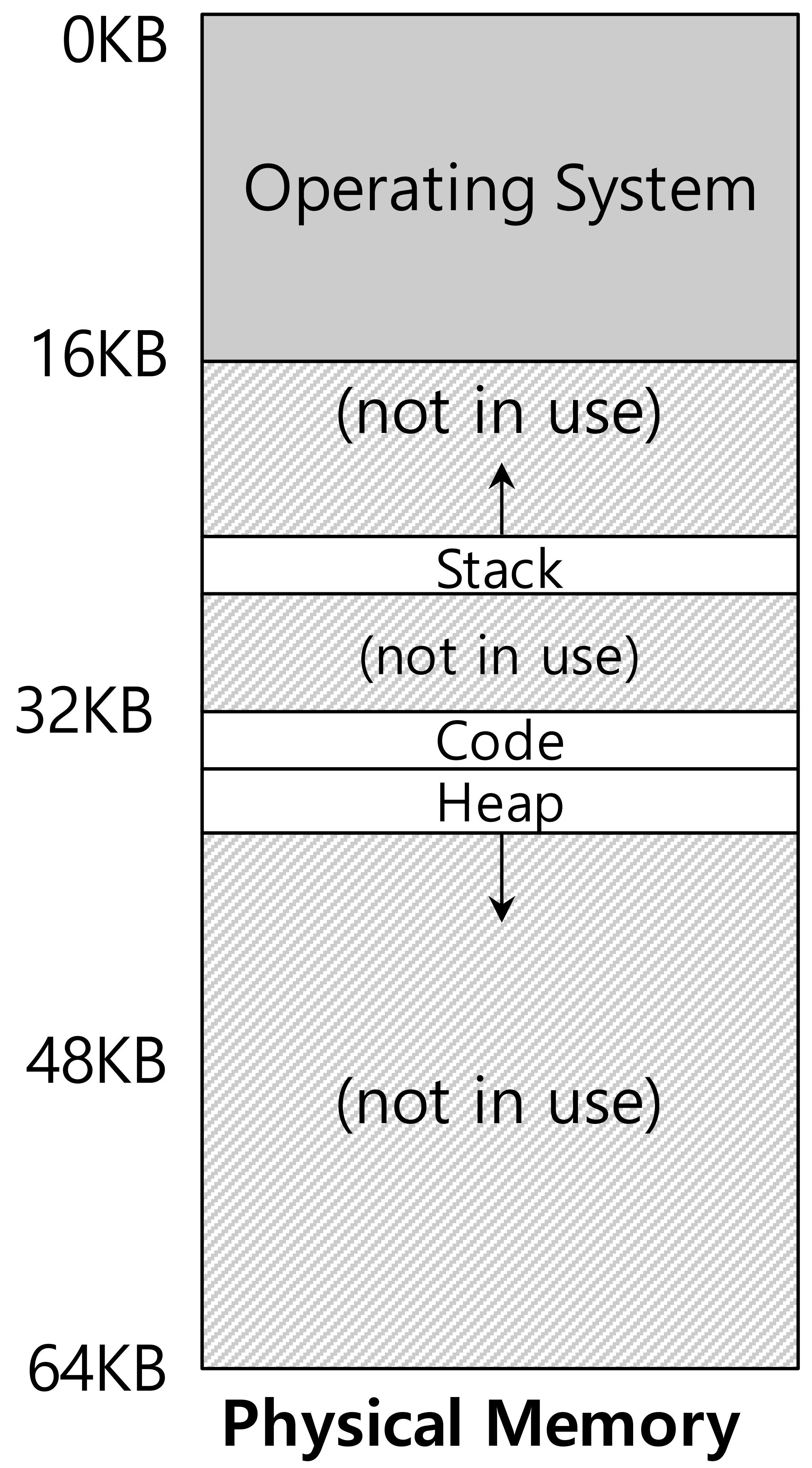
- Why not just base and bound?
  - Large chunk of free space
  - Free space takes up physical memory
  - Hard to run when an address space does not fit into physical memory



Segmentation

- A contiguous portion of the address space of a particular length
  - Logically-different segment : code, stack, heap
- Each segment can be placed in different part of physical memory
  - Base and bounds exist per each segment

Placing Segment in Physical Memory



Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Example

- Consider program that is separated into two parts : code and data
- CPU knows whether it wants an instruction or data
- Two base-limit register pairs, one for each segment
- **Can either of these be read only?**
- **What are the benefits of this scheme?**
- **What are the disadvantages?**

Variable Partition

- Multiple-partition allocation
  - **Variable-partition** sizes for efficiency
  - **Hole -** block of available memory
    - Holes of various size scattered throughout memory
    - When a process arrives, it is allocated memory from a hold large enough to accommodate it
    - Process exiting frees its partition
      - Adjacent free partitions can be combined
- OS maintains info about allocated partitions and free partitions

Dynamic Storage-Allocation Problem

- How to satisfy a request of size n form a list of free holes?
  - **First fit :** allocate the first hole that is big enough
  - **Best fit :** allocate the smallest hole that is big enough
    - Must search entire list unless ordered by size
    - Produces smallest leftover hole
  - **Worst fit :** allocate the largest hole
    - Must search entire list unless ordered
    - Produces largest leftover hole
- Which is best/worst for speed? For storage utilization?

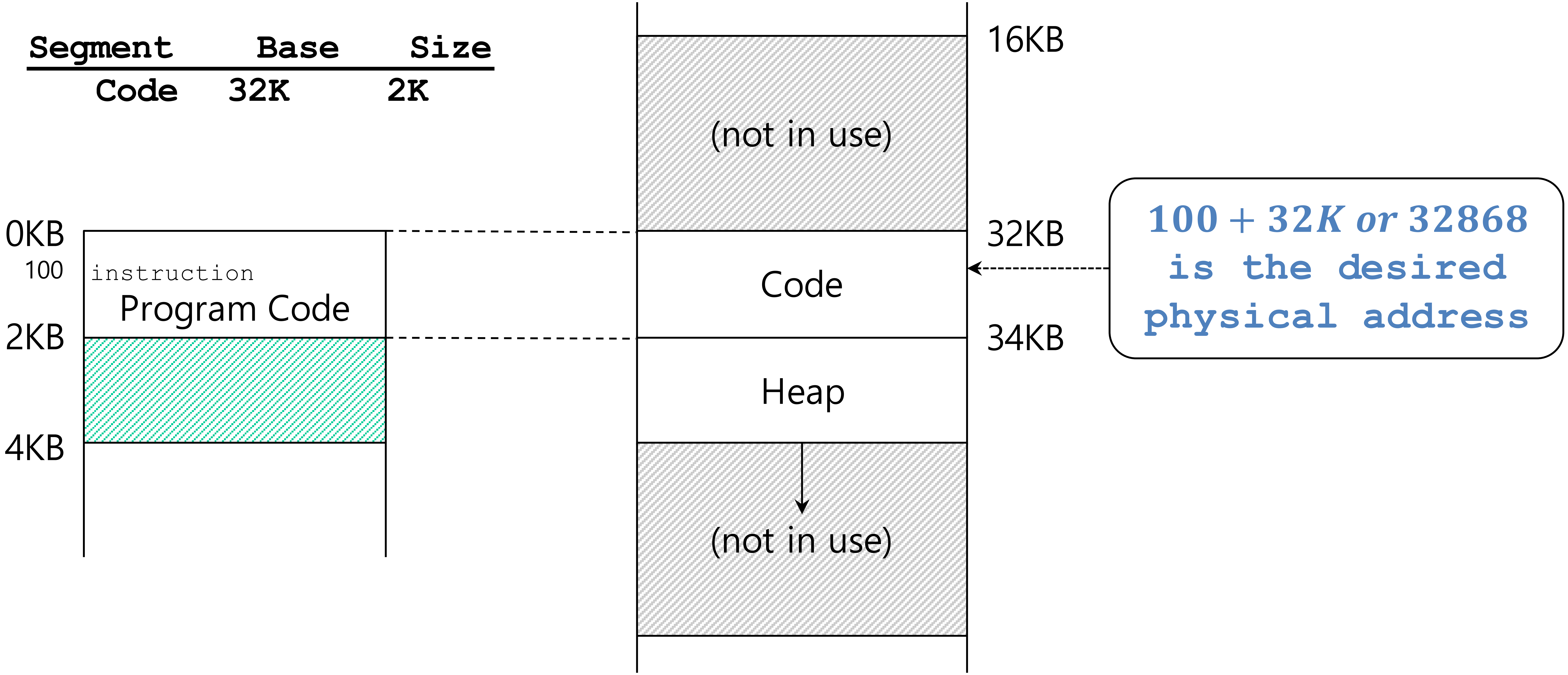
Example

- Six partitions (in order) : 300KB, 600KB, 350KB, 200KB, 750KB, 125KB
- Processes to be placed (in order): 115KB, 500KB, 358KB, 200KB, 375KB

Address Translation on Segmentation

- Physical address = offset + base

Segment	Base	Size
Code	32K	2K



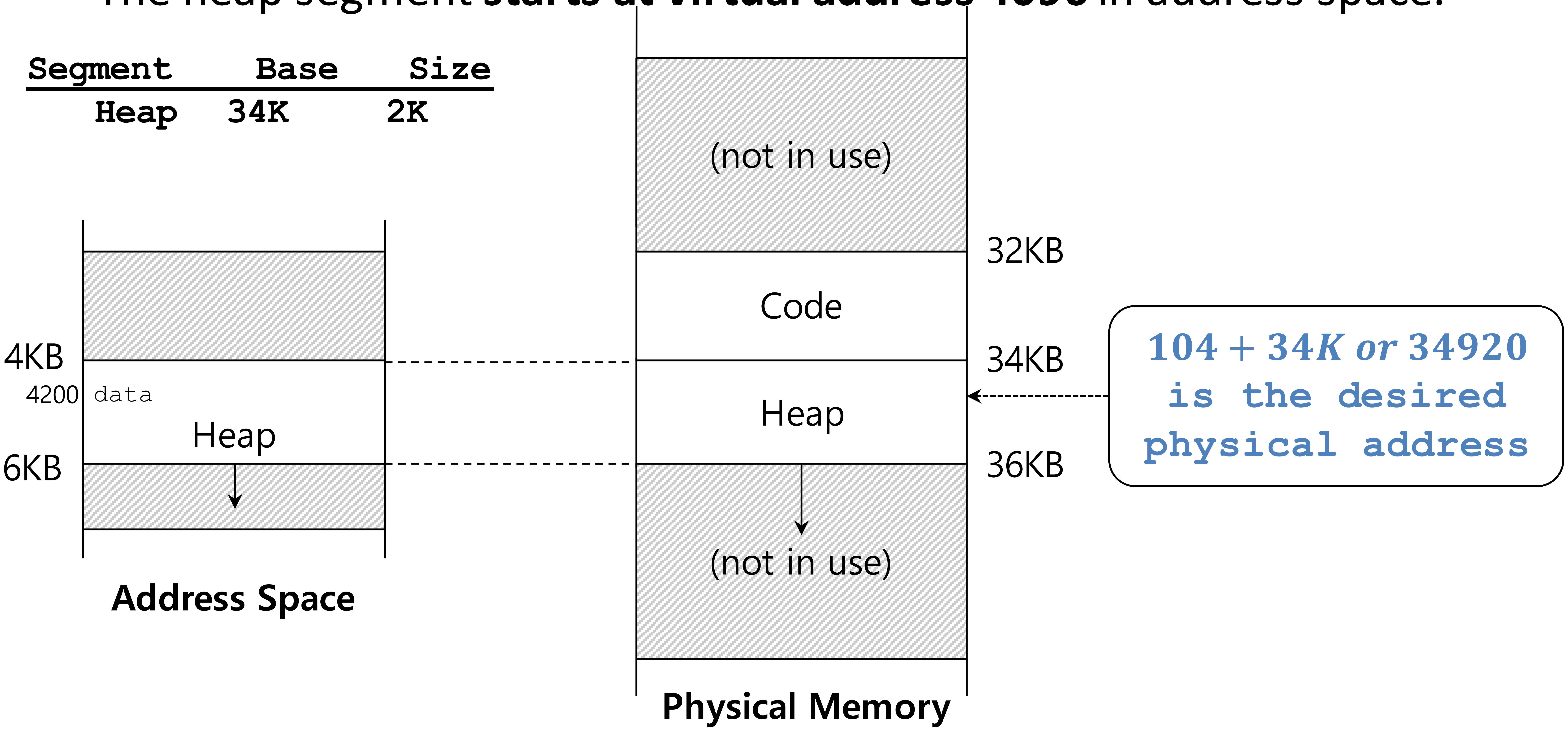
Address Translation on Segmentation

- Is physical address = virtual address + base?

Address Translation on Segmentation

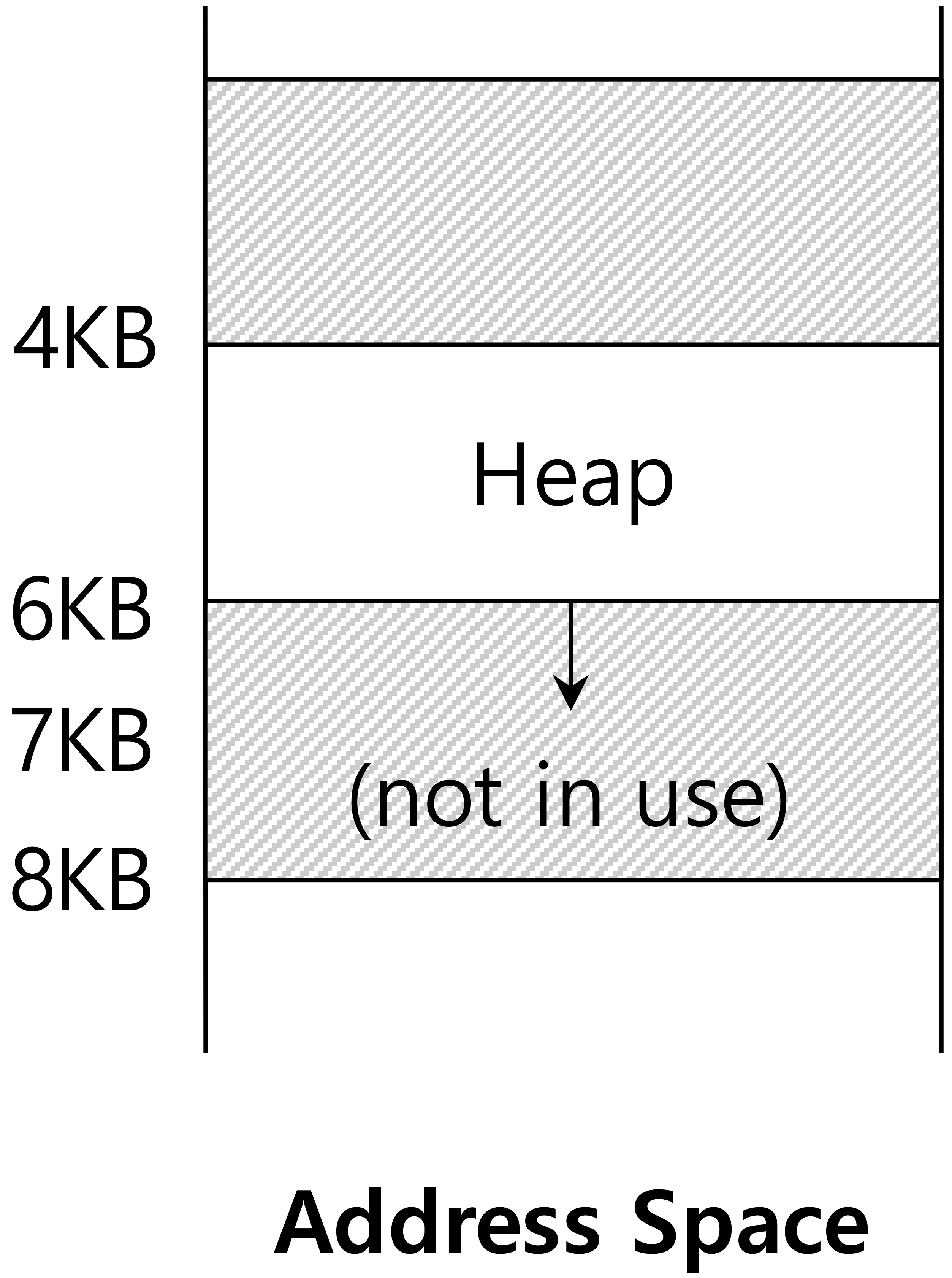
- **The `offset` of virtual address 4200 is `104`.**
  - The heap segment **starts at virtual address 4096** in address space.

Segment	Base	Size
Heap	34K	2K



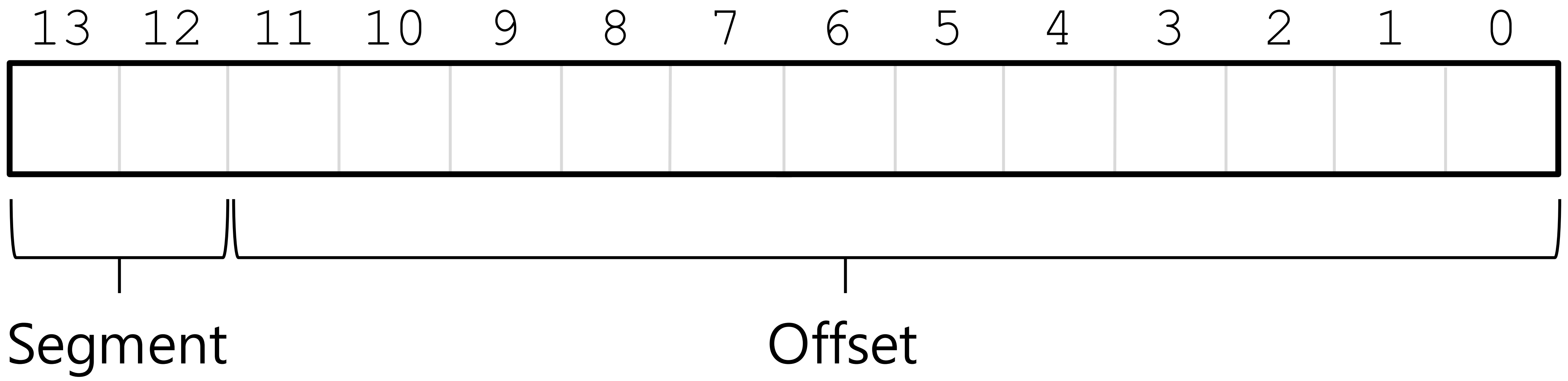
Segmentation Fault

- If an illegal address such as 7KB which is beyond end of heap is referenced, OS occurs segmentation fault
  - Hardware detects address is out of bounds



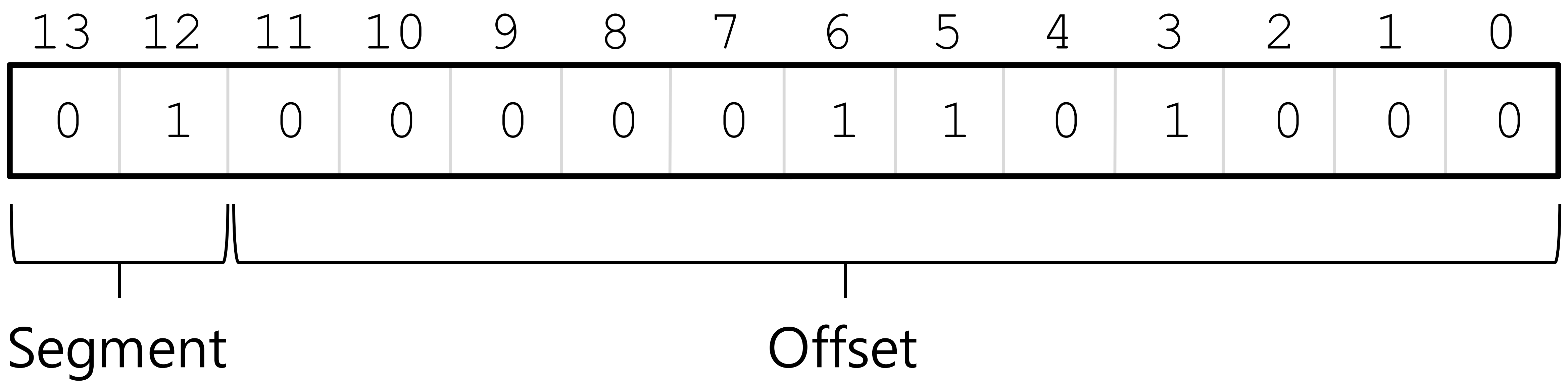
Referring to Segment

- **Explicit approach**
  - Chop up the address space into segments based on the **top few bits** of virtual address.



- **Example: virtual address 4200 (01000001101000)**

Segment	bits
Code	00
Heap	01
Stack	10
-	11



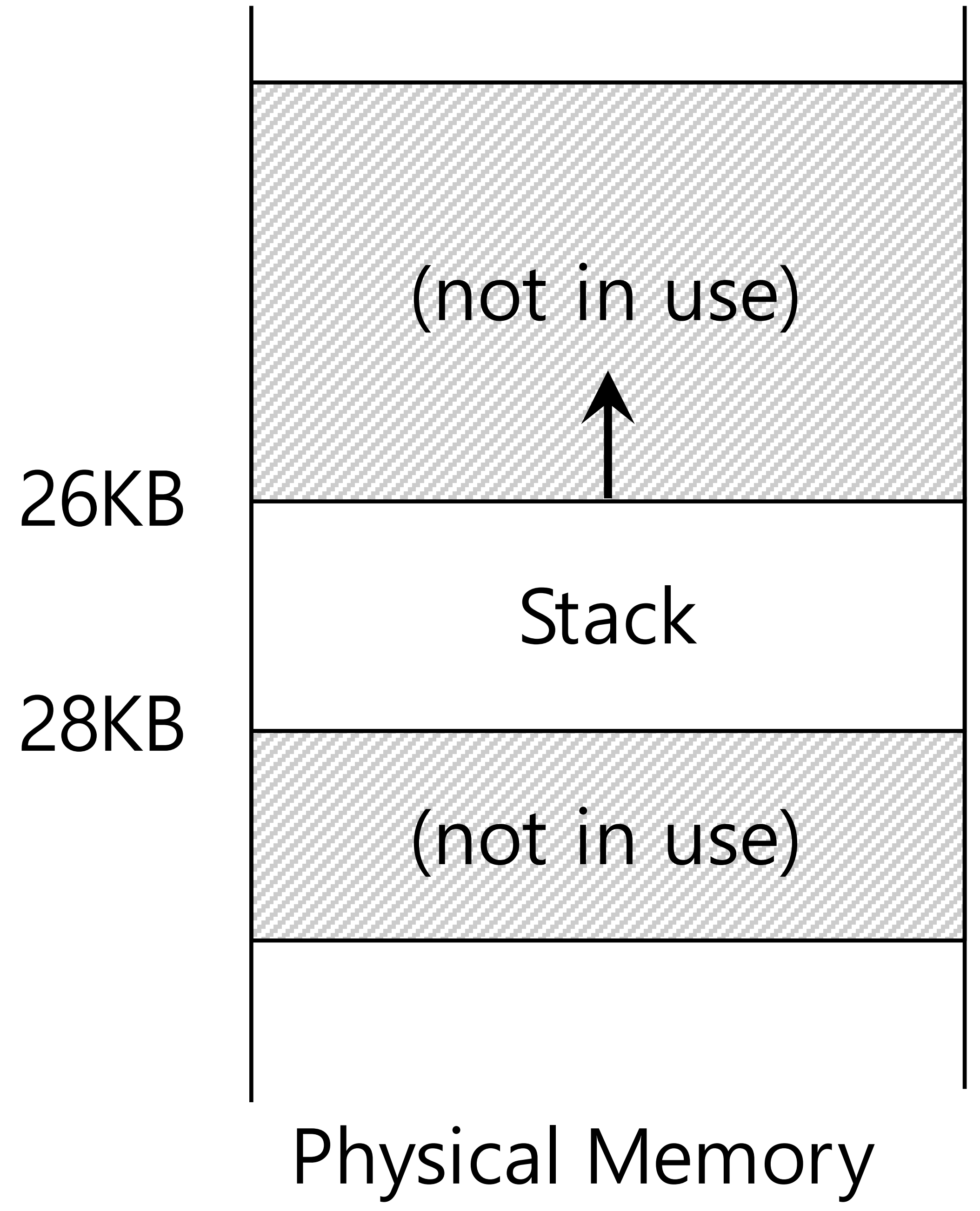
Referring to Segment

```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// now get offset
Offset = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
    RaiseException(PROTECTION_FAULT)
else
    PhysAddr = Base[Segment] + Offset
    Register = AccessMemory(PhysAddr)
```

- SEG_MASK = 0x3000(11000000000000)
- SEG_SHIFT = 12
- OFFSET_MASK = 0xFFF (00111111111111)

Referring to Stack Segment

- Stack grows backwards
- Extra hardware support is needed for this
  - Hardware checks which way segment grows
  - 1 : positive direction, 0: negative direction



Segment Register(with Negative-Growth Support)

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

"Half of Operating Systems is Stupid Memory Management Tricks" - Prof. Bridges

- Now : multiple processes, each with own address space
- Lots of optimization opportunities and subtle questions
  - How many copies of libc exist in memory of system at once?
  - What is we want to run more programs than we have physical memory?
  - Can physical memory be in multiple segments at same time?

Support for Sharing

- Segment can be shared between address spaces
  - **Code sharing** (e.g. shared libraries)
  - Needs extra hardware support
- **Protection bits :** a few extra bits per segment, indicating permissions of read, write, and execute

Segment Register Values(with Protection)

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

- Who maintains these bits?

How many segments should we have?

- Coarse-grained (few segments) means segmentation in a small number of segments
  - Code, head, stack
  - Relatively easy to manage
- Fine grained (lots of segments) allows more flexibility for ‘stupid’ OS tricks
  - OS can do lots of things with lots of segments
    - Map multiple different shared libraries into multiple processes
  - OS has to manage allocation of all these segments
  - Typically supported with a hardware segment table

Fragmentation

- External Fragmentation : unused memory between partitions
  - As processes are loaded and removed from memory, free memory space is broken into pieces
  - Example : There are 24KB free, but not in one contiguous segment
  - OS cannot satisfy 20KB request
- Internal Fragmentation : unused memory that is internal to a partition
- 50 Percent rule : for first fit, given N blocks allocated, 0.5 N blocks lost to fragmentation (1/3 may be unusable)

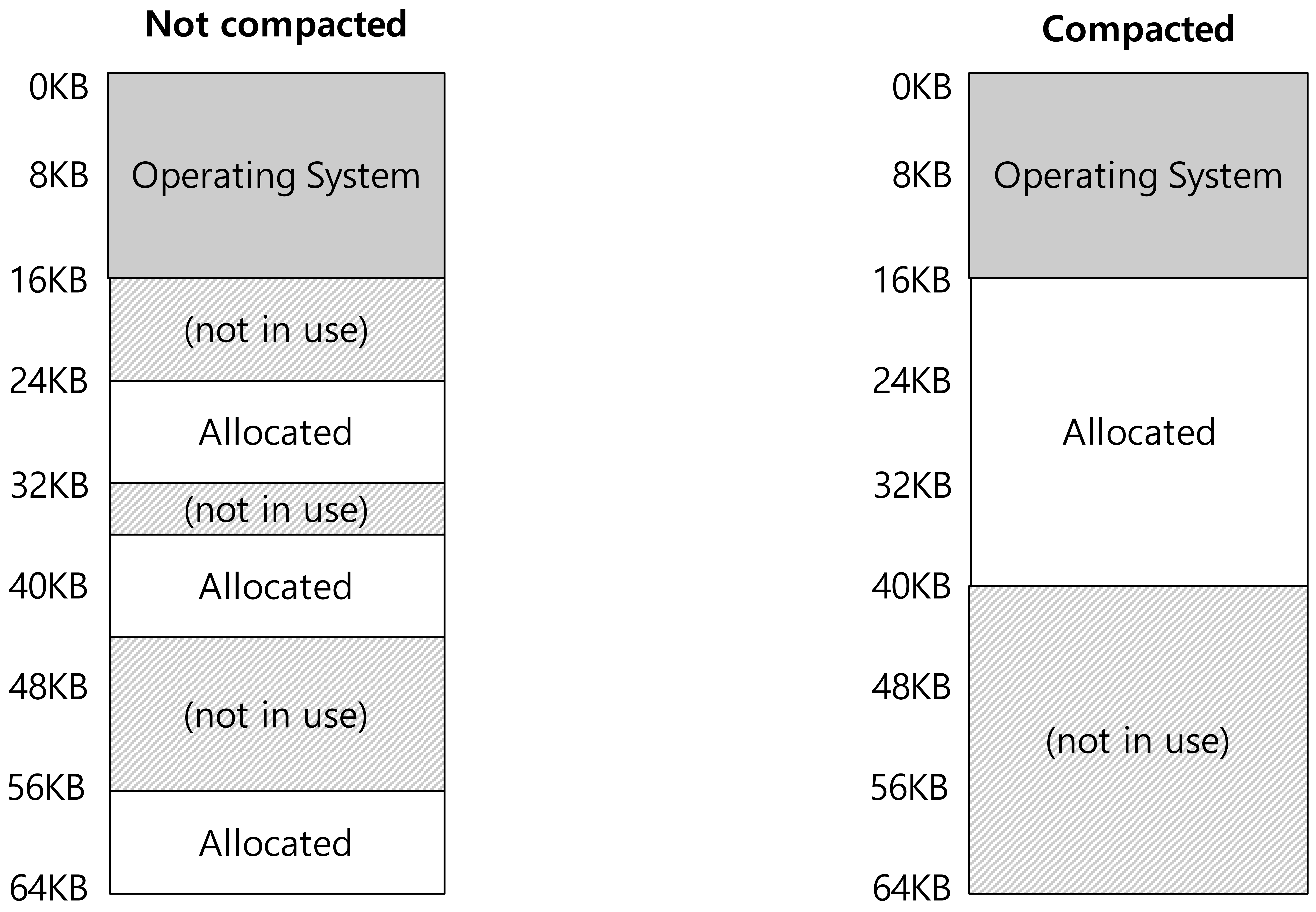
Fragmentation (Cont.)

- Hole size : 18,464 bytes
- Requested allocation size : 18,462 bytes
    - If allocated exact size, hole will be 2 bytes … overhead with keeping track of hole is not worth it
- Solution : break physical memory into fixed-size blocks and allocate memory based on block size

Compaction

- Reduce external fragmentation by :
  - Shuffling memory contents to place all free memory together in one large block
- Only possible if relocation is dynamic and is done at execution time
- Can be expensive to move address spaces to make larger holes
- Could permit logical address space to be noncontiguous (paging)
  - Will discuss this later

Memory Compaction



Whence Segmentation

- Segmentation is variable length allocation
  - Just like malloc free lists, with many of the same problems
  - Useful and flexible, but hard to manage well
  - Particularly hard when lots of segments
- Modern OS make only very limited use of segmentation
  - 32-bit mode x86 : can use segments extensively, but most OS (Windows/Linux) don't
  - 64-bit mode x86 forces most segments to have base address of 0
  - Usually used for thread-specific data