Week 6

Memory Management: Virtual Memory

Oana Balmau

February 7, 2023

Second assignment released

Announcements

First assignment due next Monday!!

Week 6 **Memory Management**	feb 6 C Review: C files	feb 7 **Virtual Memory (1/2)** Optional reading: OSTEP Chapters 12 – 18	feb 8 Scheduling Assignment Released	feb 9 **Virtual Memory (2/2)** Scheduling Assignment Overview -- with Jiaxuan	feb 10
Week 7 **Memory Management**	feb 13 **OS Shell Assignment Due** C Review: Working with pthreads I	feb 14 **Demand Paging (1/3)** Optional reading: OSTEP Chapters 19 – 22	feb 15	feb 16 **Demand Paging (2/3)**	feb 17
Week 8 **Memory Management**	feb 20 C Review: Working with pthreads II	feb 21 **Demand Paging (3/3)** Optional reading: OSTEP Chapters 19 – 22 Practice Exercises Sheet: Memory Management	feb 22	feb 23 Mid-semester Q&A – not recorded • Graded Exercises Sheet Released • Grades released for OS Shell Assignment	feb 24
Week 9	feb 27	feb 28	mar 1	mar 2	mar 3

→ This week, Oana’s office hours happen **today 1pm**, in MC113N. No office hours on Thursday.

Key Concepts

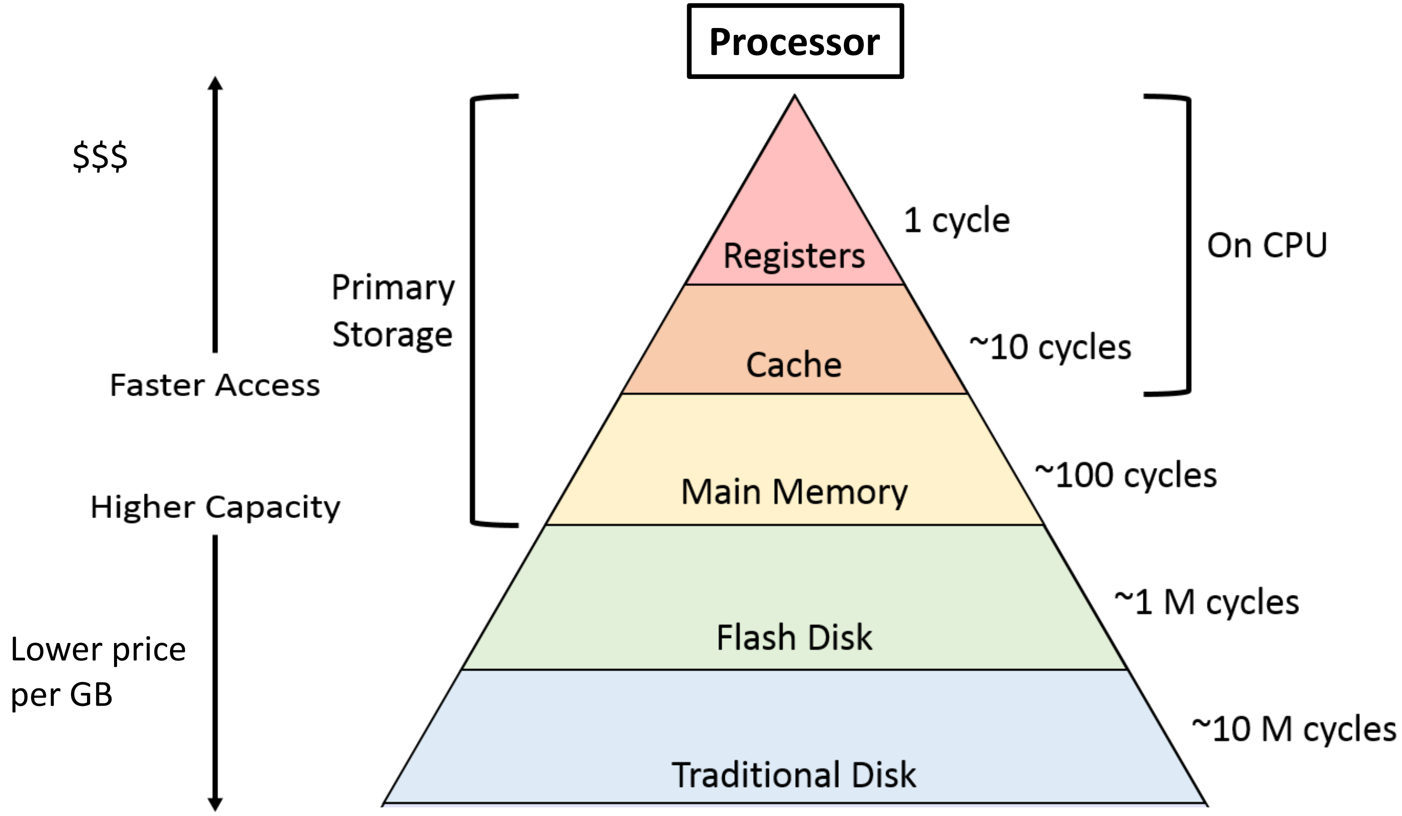
- Virtual and physical address spaces
- Mapping between virtual and physical address
- Different mapping methods:
  - Base and bounds, Segmentation, Paging
- Sharing, protection, memory allocation

Memory: the Dream

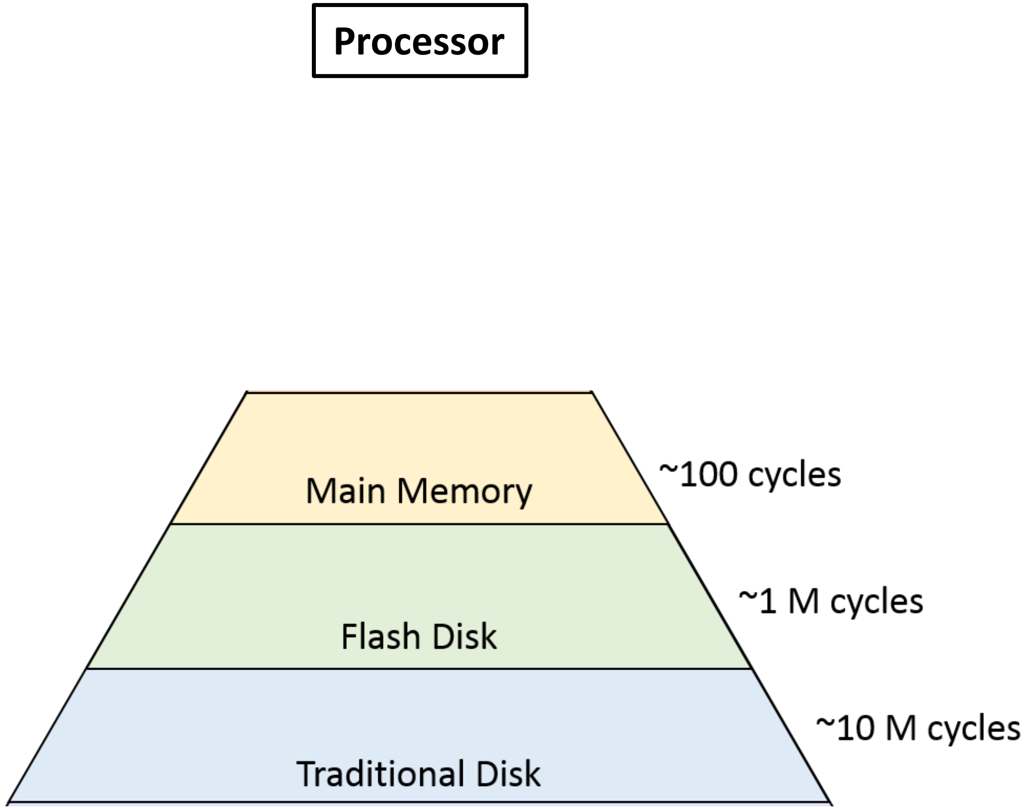
What every programmer would like is a

- private,
- infinitely large,
- infinitely fast,
- nonvolatile, and
- cheap memory

Real world: Memory Hierarchy



OS Memory Management

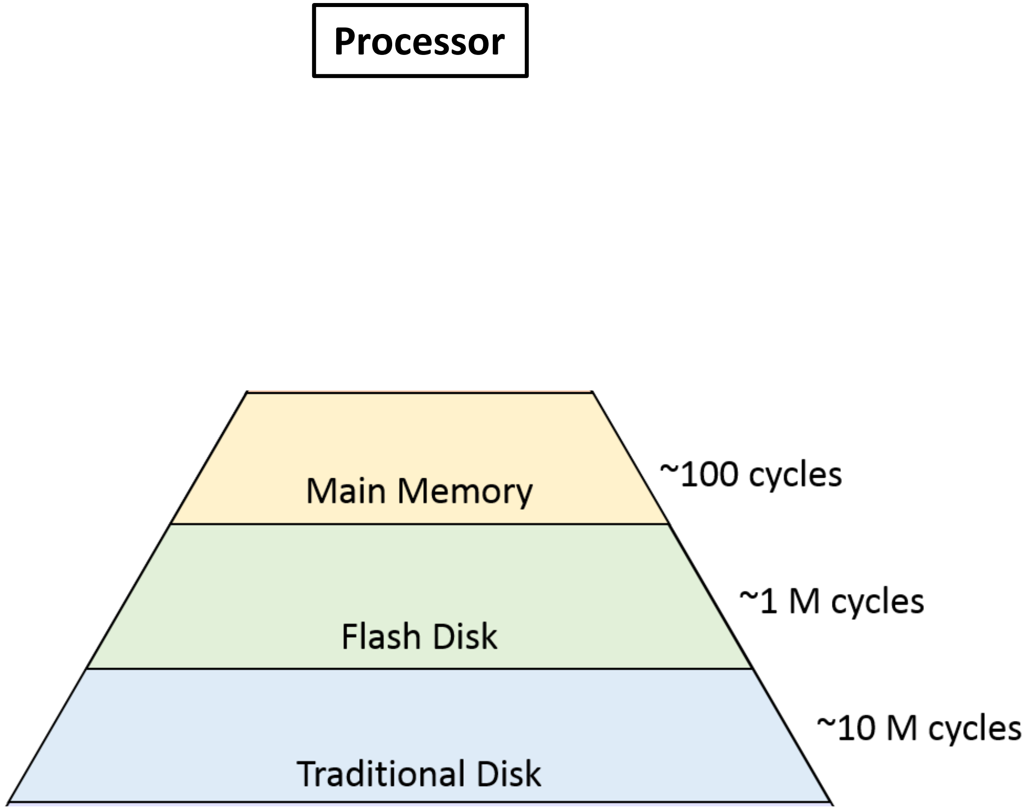


Simplifying Assumption

For this week's lecture only:

All of a program must be in main memory

Will revisit assumption next week

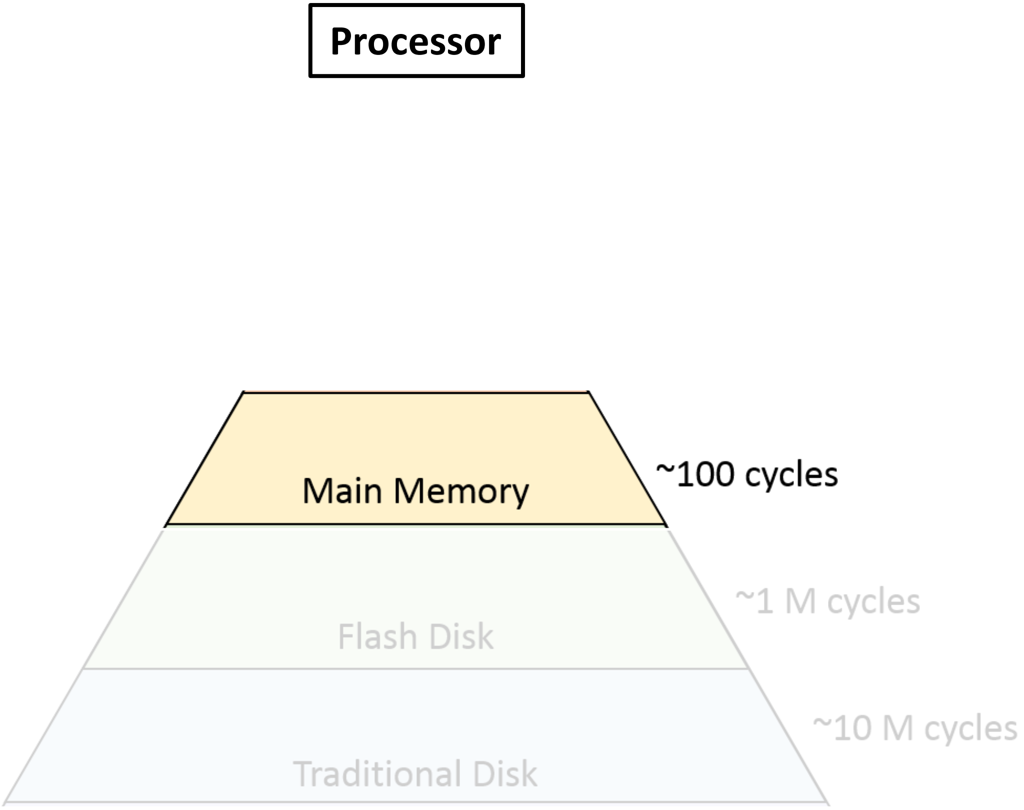


Simplifying Assumption

So for today:

All of a program must be in main memory

Not concerned with disk



Goals of OS Memory Management

Main memory allocation

- Where to locate the kernel?
- How many processes to allow?
- What memory to allocate to processes?

Protection

- Cannot corrupt OS or other processes
- Privacy: Cannot read data of other processes

Transparency

- Processes are not aware that memory is shared
- Works regardless of number and/or location of processes

Goals of OS Memory Management

Main memory allocation *We will return to this topic later today*

- Where to locate the kernel?
- How many processes to allow?
- What memory to allocate to processes?

Protection

- Cannot corrupt OS or other processes
- Privacy: Cannot read data of other processes

Transparency

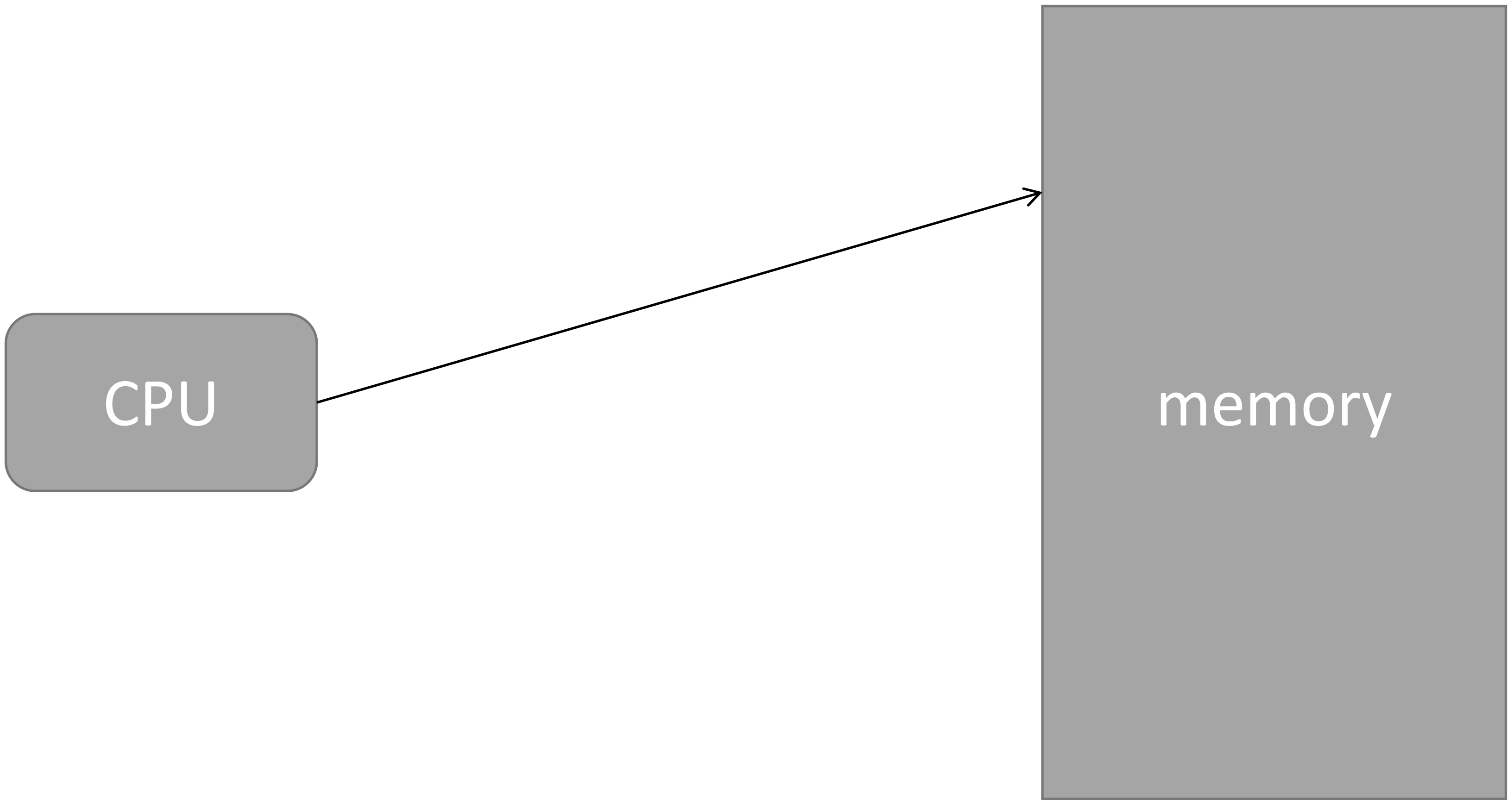
- Processes are not aware that memory is shared
- Works regardless of number and/or location of processes

Protection

One process must not be able to read or write the memory

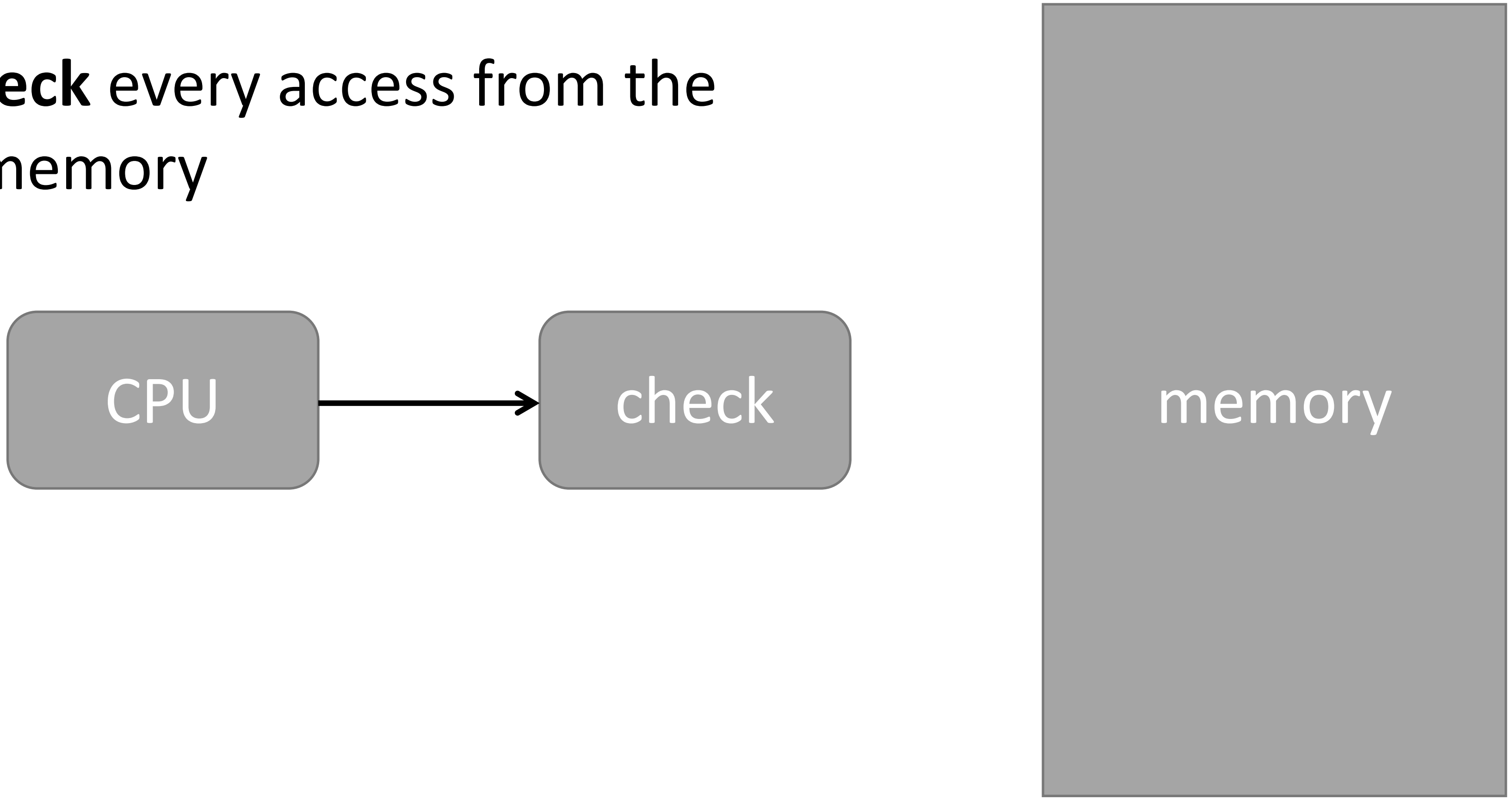
- of another process
- of the kernel

Unprotected Access

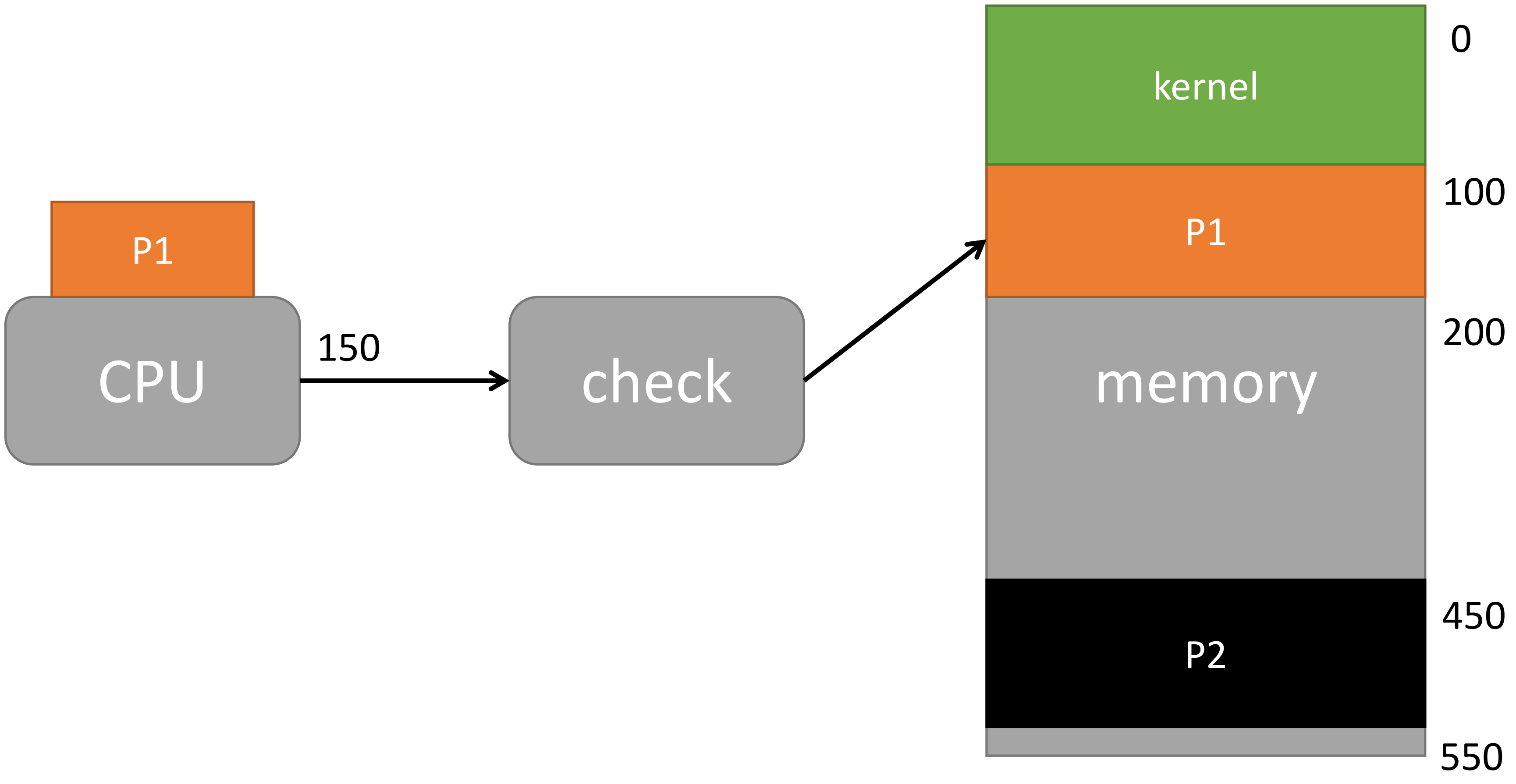


Protected Access

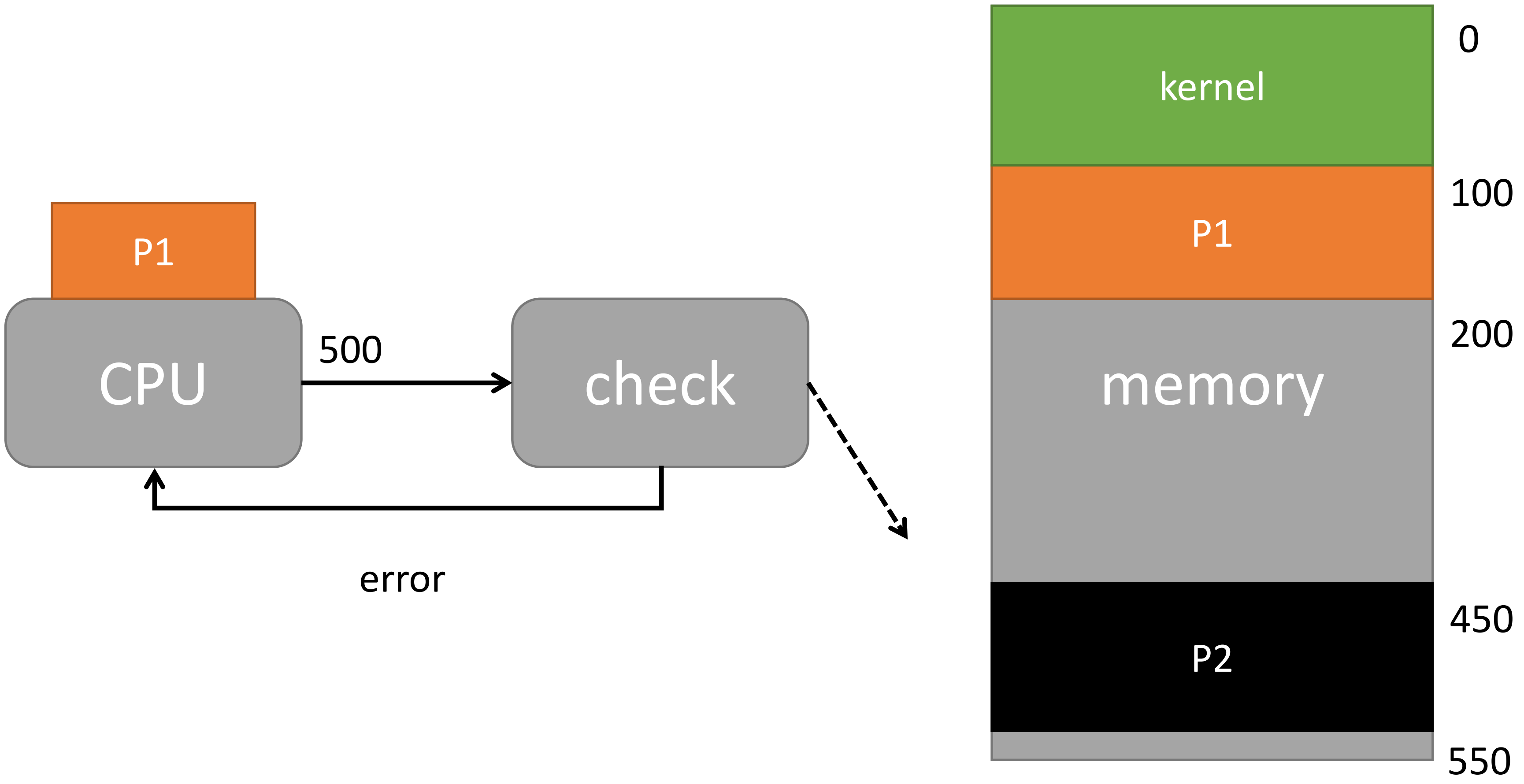
Must check every access from the CPU to memory



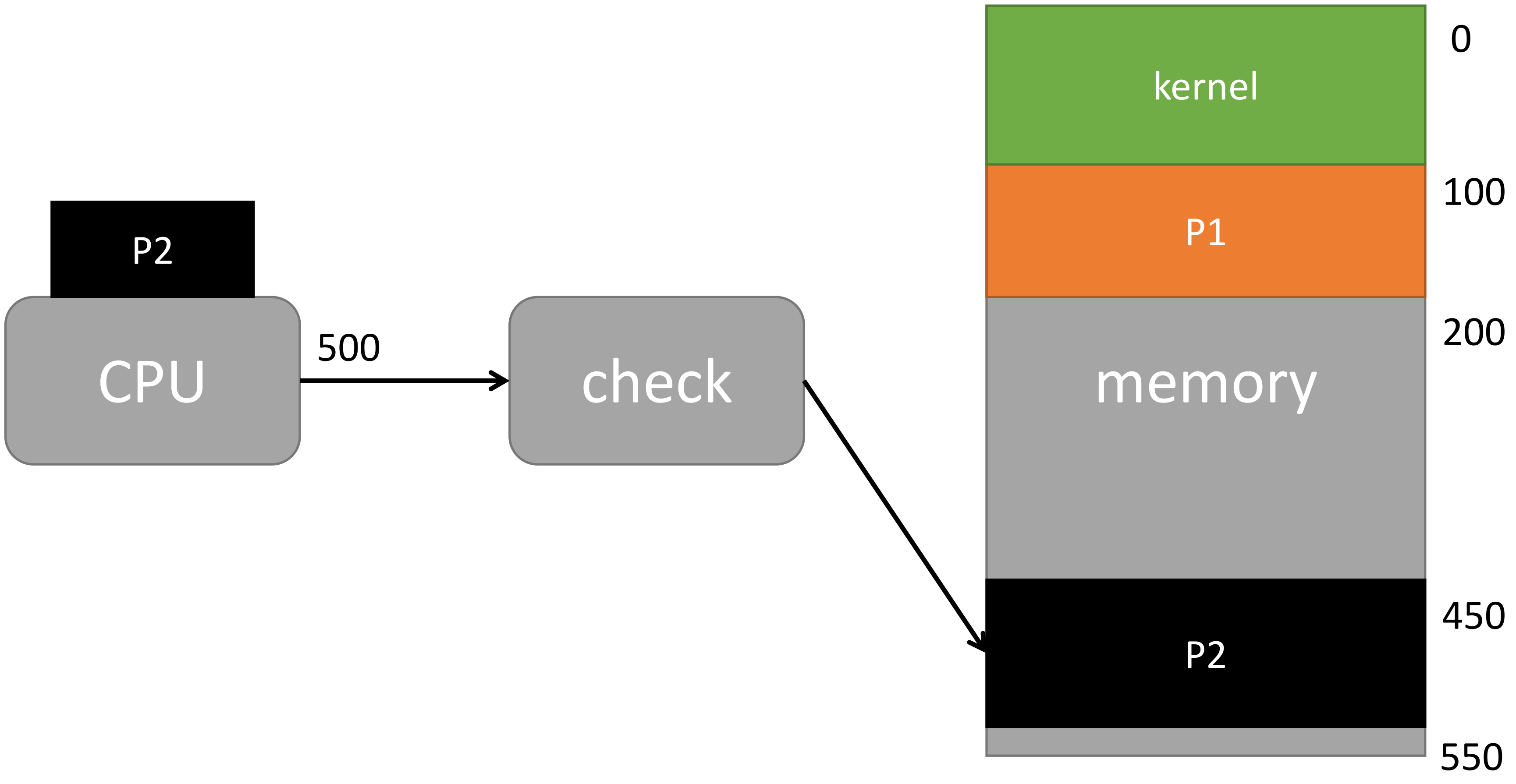
Protection: Examples



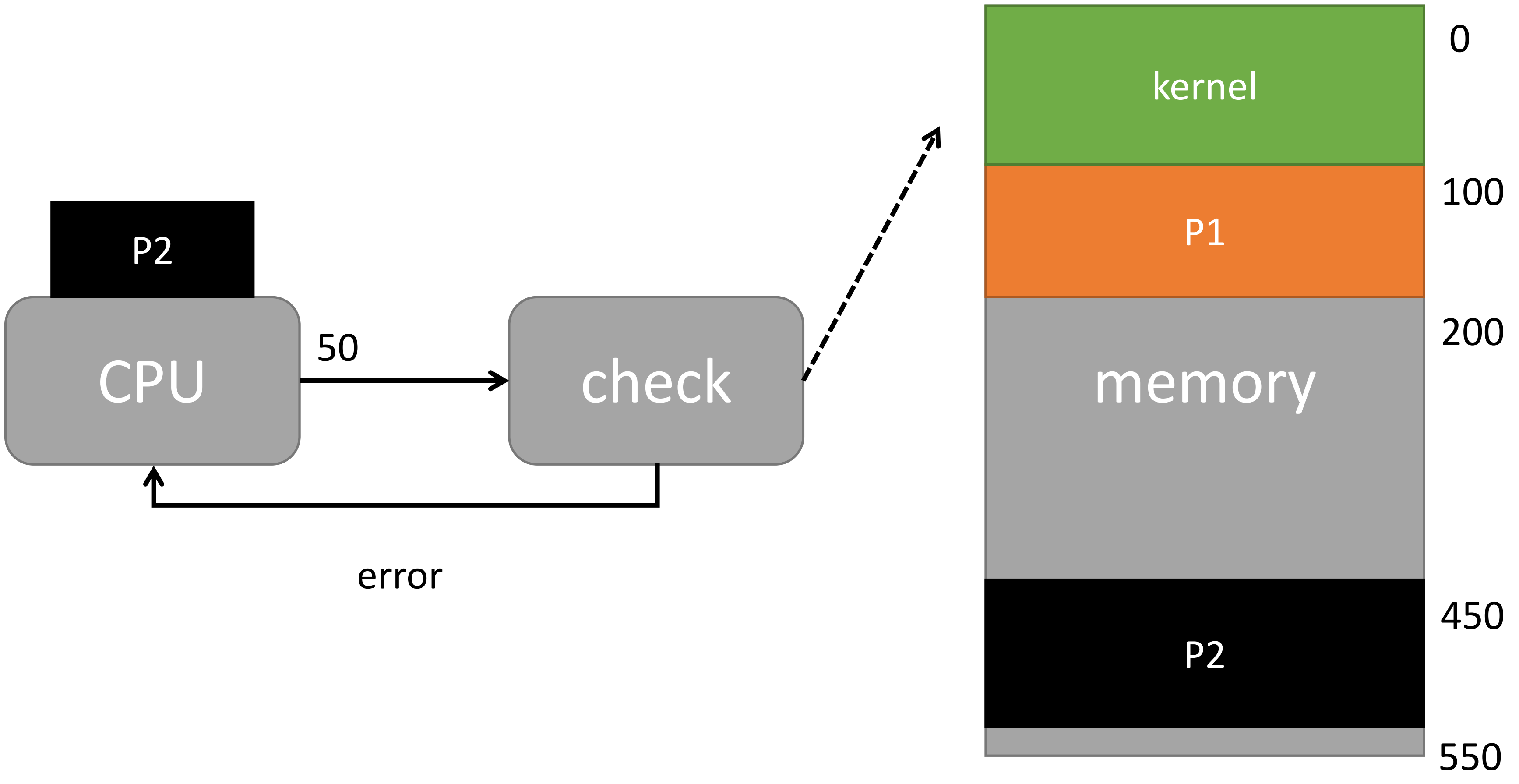
Protection: Examples



Protection: Examples



Protection: Examples



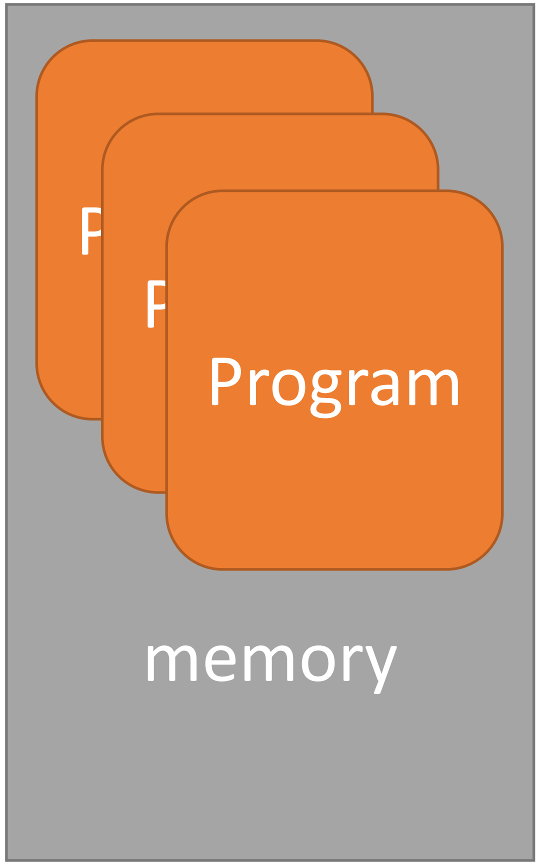
Transparency

Programmer should not have to worry

- where their program is in memory
- where or what other programs are in memory

Transparency

Program can be Anywhere in Main Memory



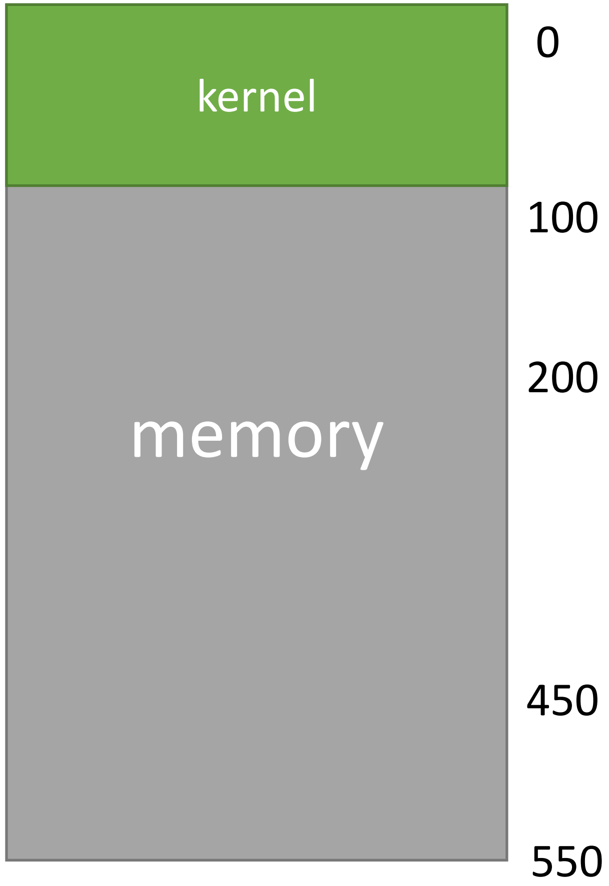
Main Memory Allocation

- Where to locate the kernel?
- How many processes to allow?
- What memory to allocate to processes?

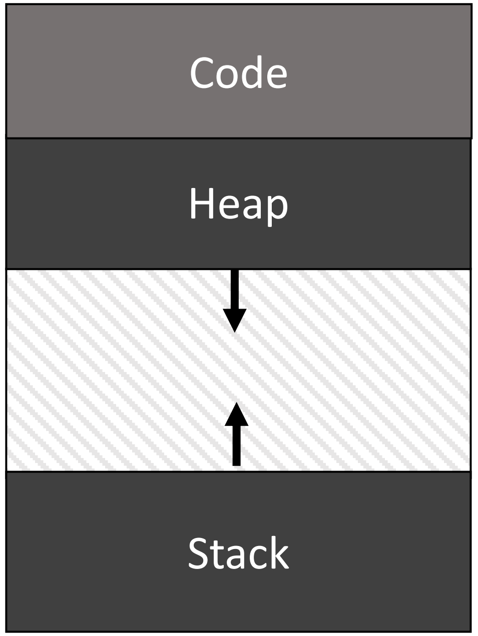
Allocating Main Memory for Kernel

Almost always in low memory

Why? Interrupt vectors are in low memory

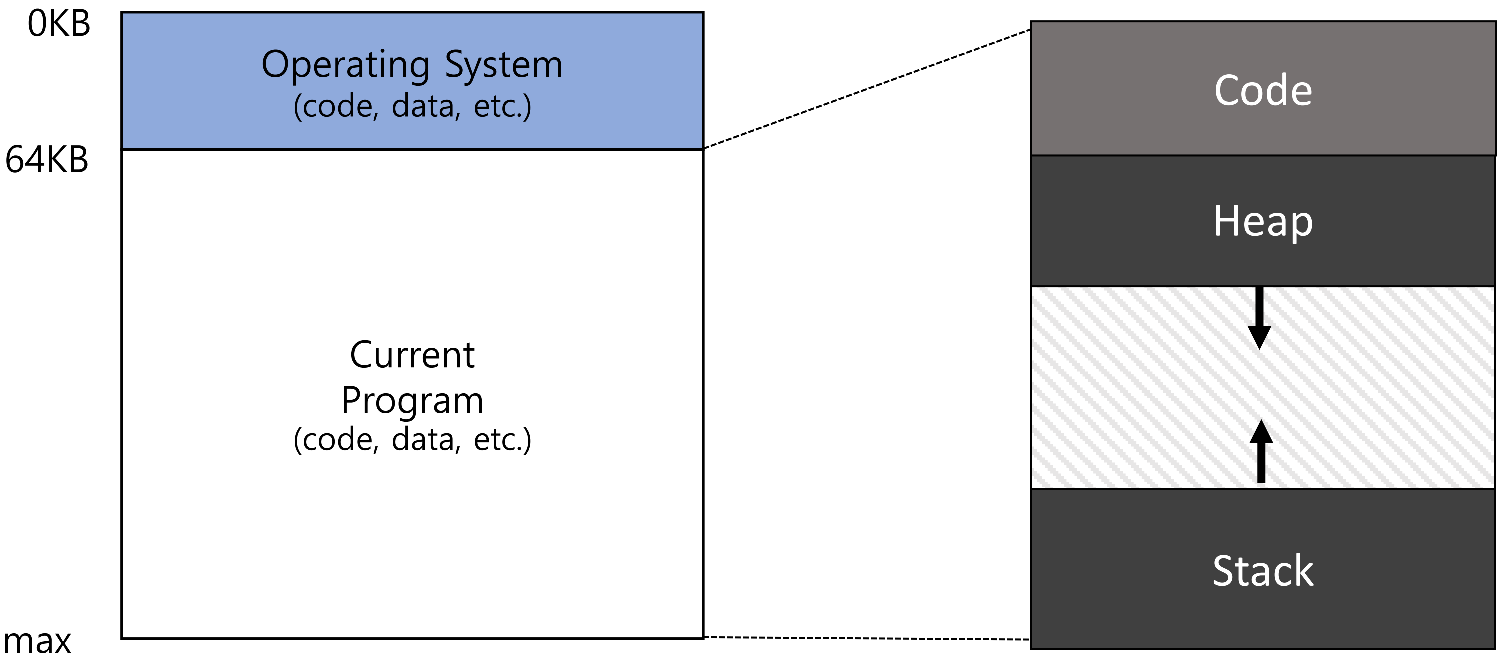


Main Memory Allocation for Processes



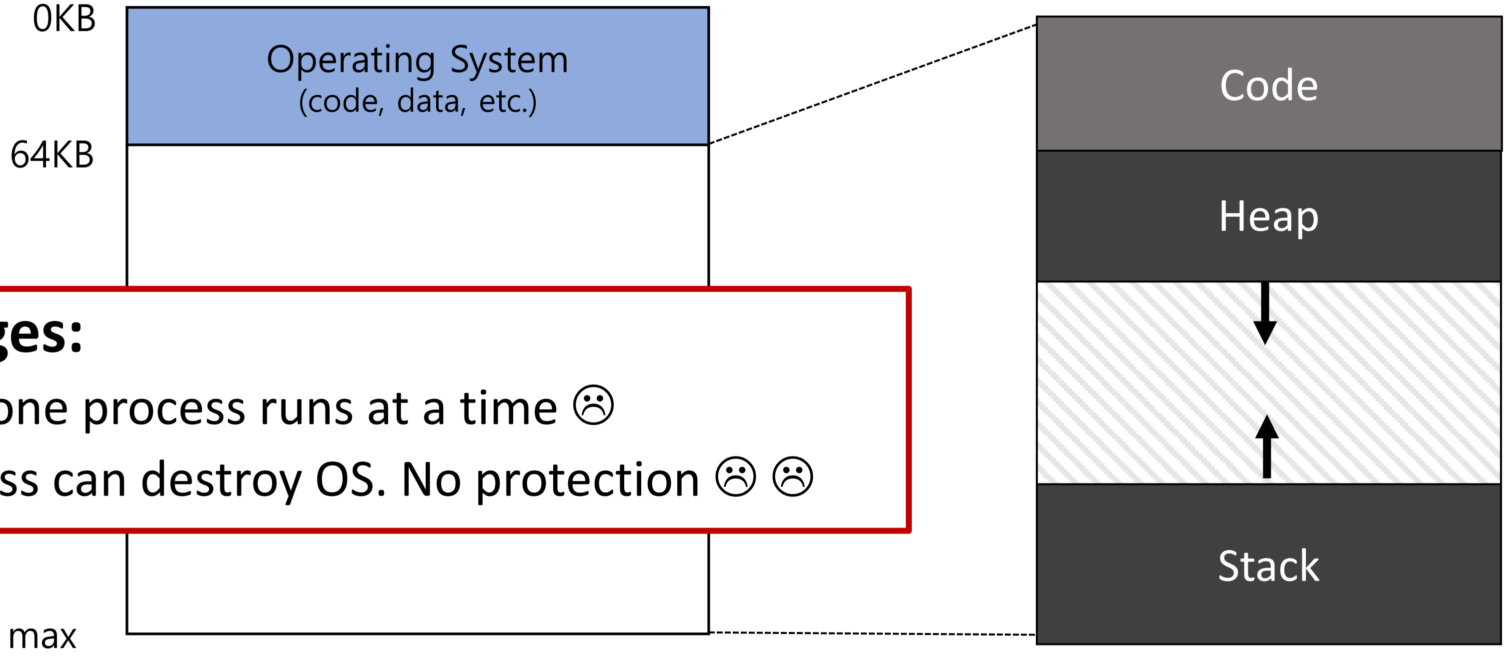
Early days: Uniprogramming

- One process runs at a time. Process ”sees” physical memory.



Early days: Uniprogramming

- One process runs at a time. Process ”sees” physical memory.



Disadvantages:

- Only one process runs at a time ☹
- Process can destroy OS. No protection ☹ ☹

The Crux: Virtualizing memory

How can the OS give **the illusion** of a
private, potentially **large address space**
for **multiple running processes**
(all sharing memory)
on top of a single, physical memory?

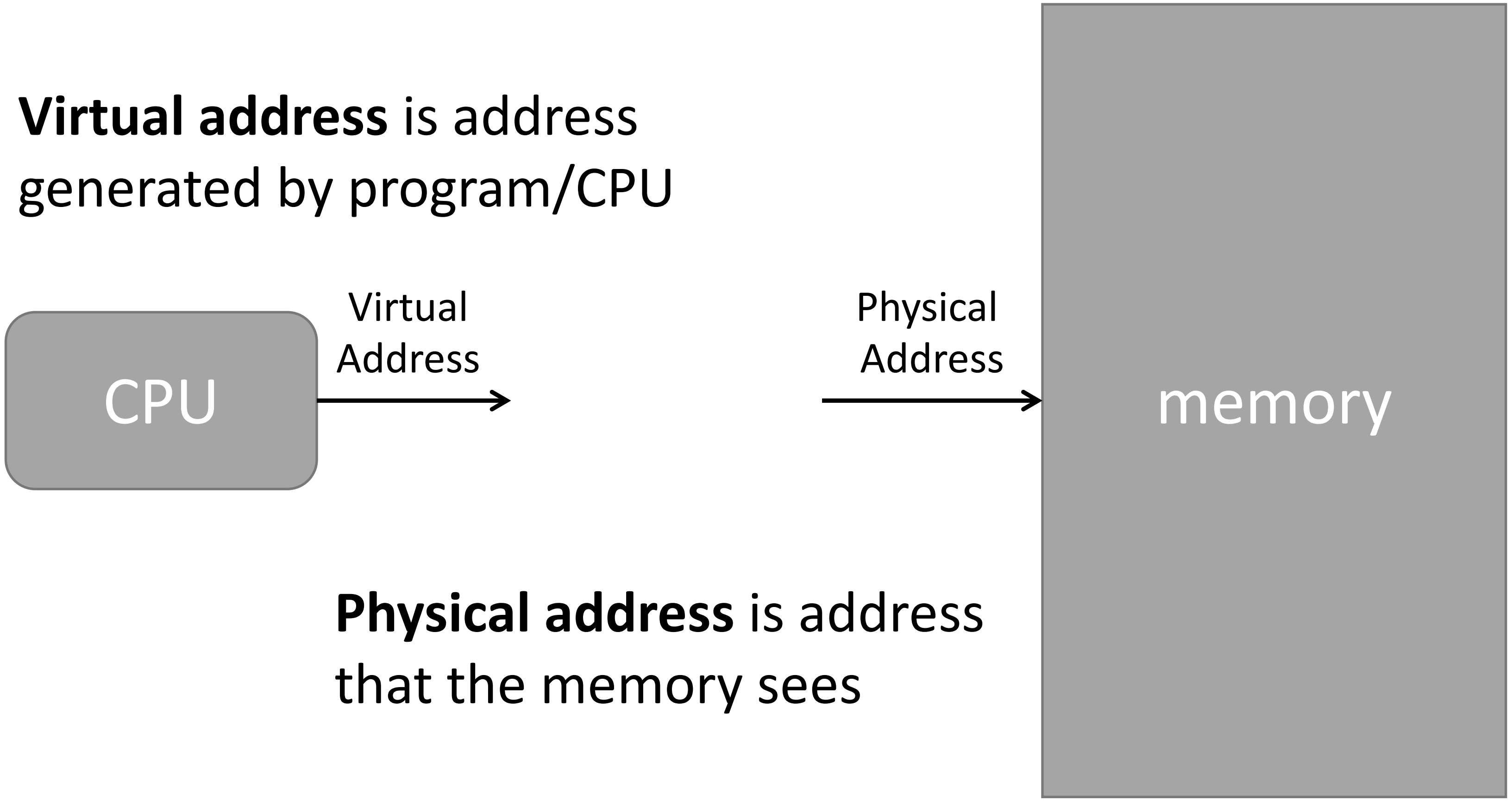
Virtual vs. Physical address space

Virtual/logical address space = What the program(mer) thinks is its memory

Physical address space = Where the program actually is in physical memory

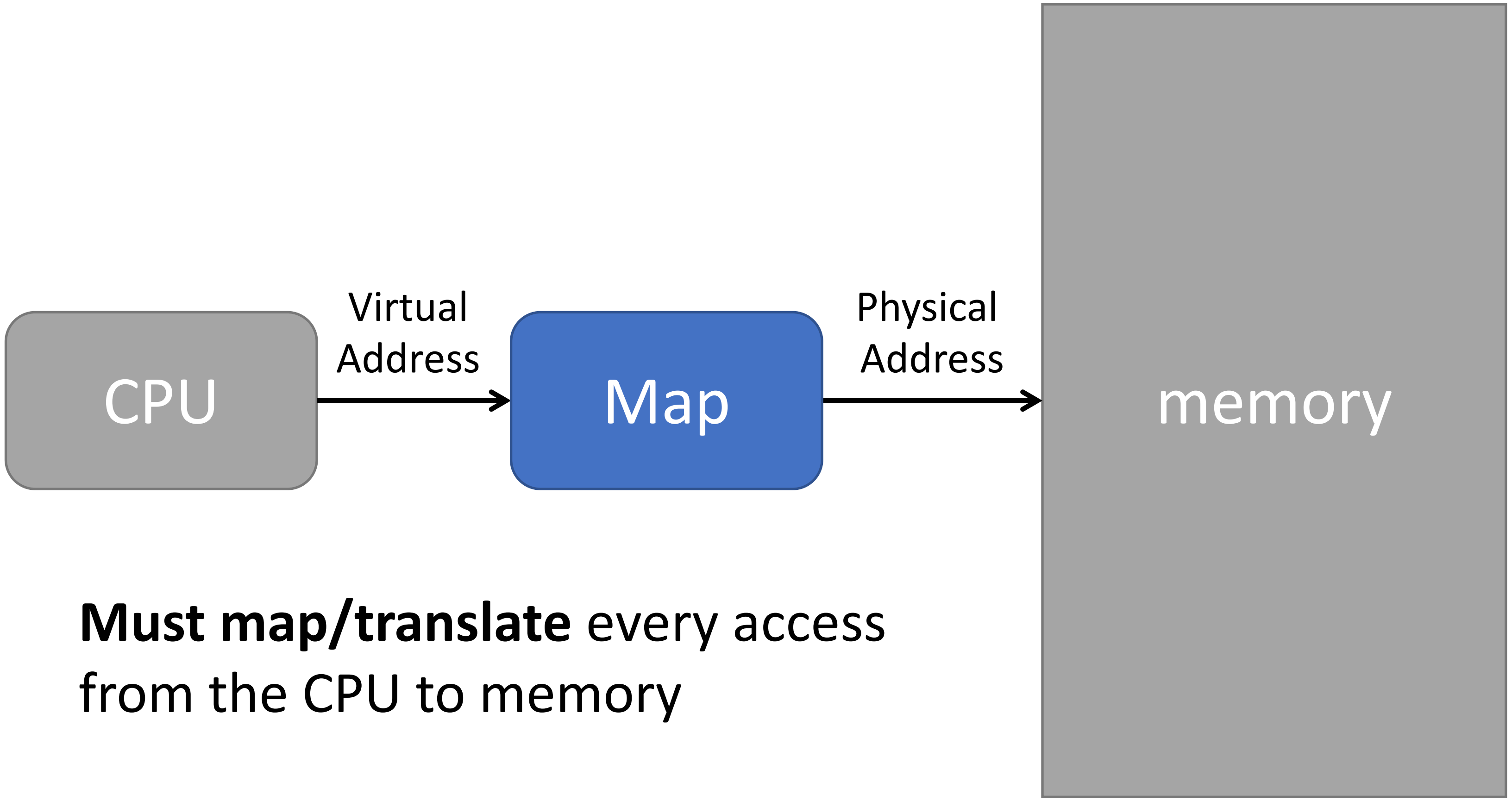
Virtual vs. Physical

Virtual address is address generated by program/CPU



Physical address is address that the memory sees

Translating Virtual to Physical

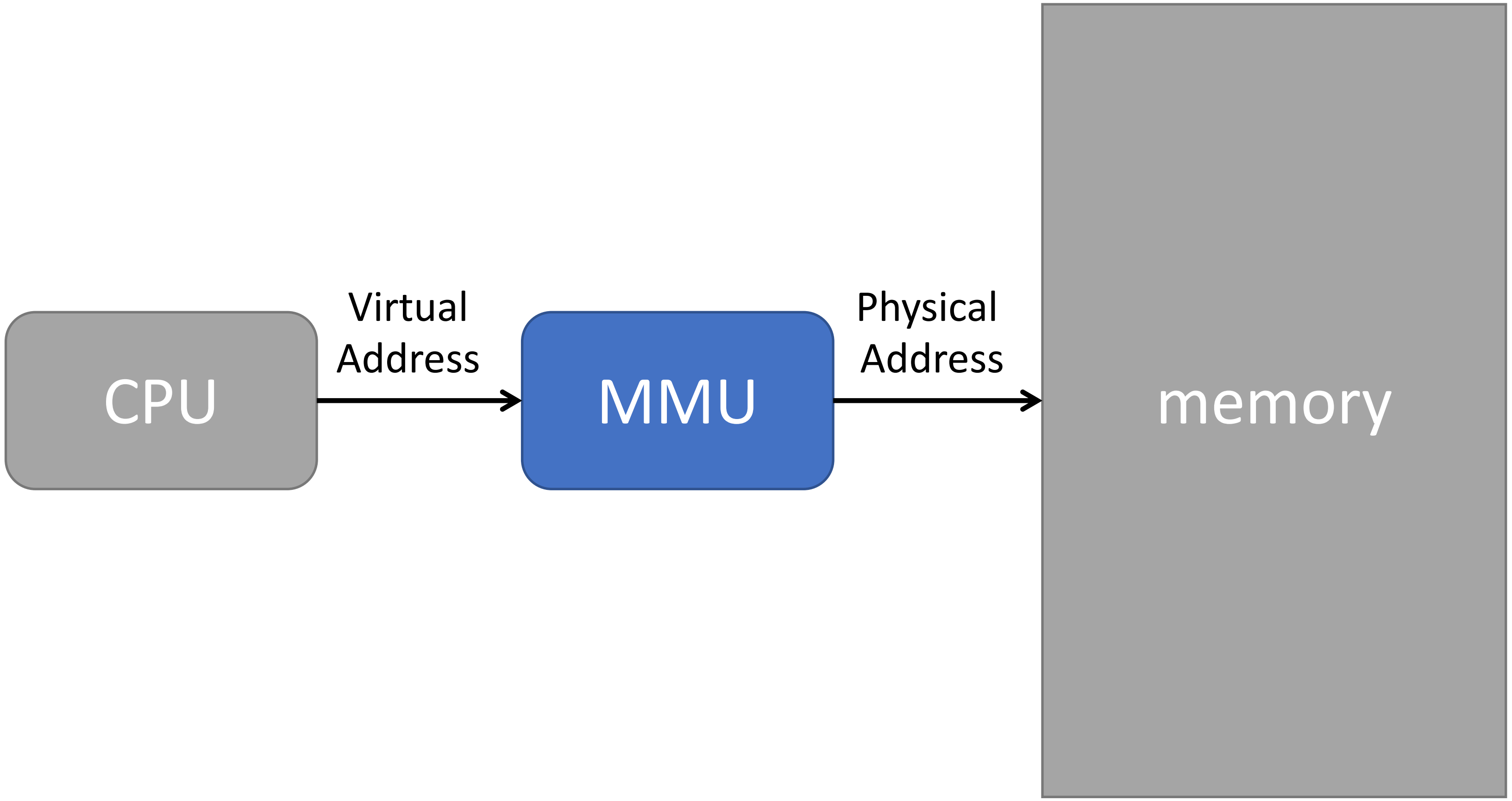


Must map/translate every access from the CPU to memory

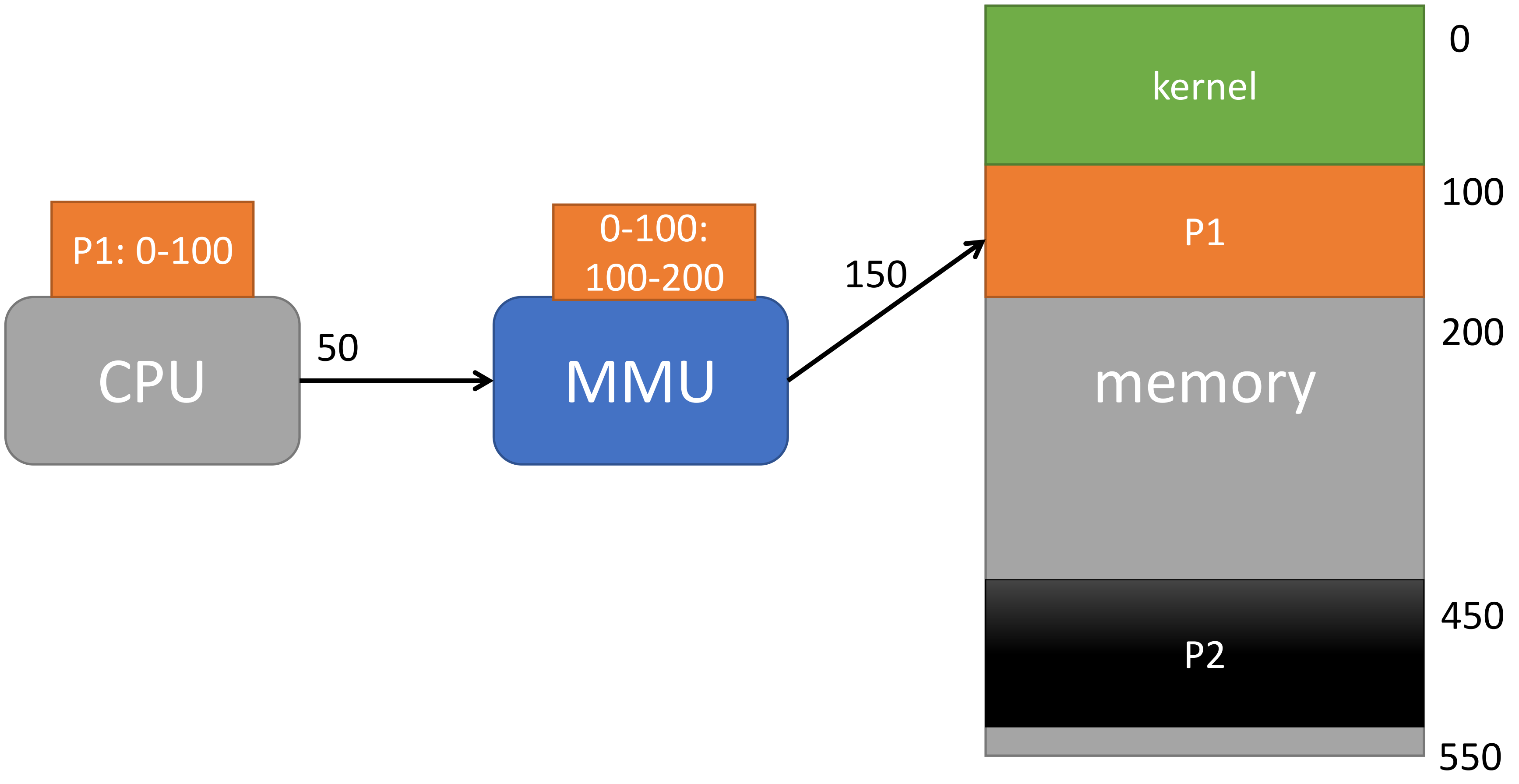
Memory Management Unit (MMU)

- Provides **mapping** virtual-to-physical
- Provides **protection** at the same time
- **Hardware!**

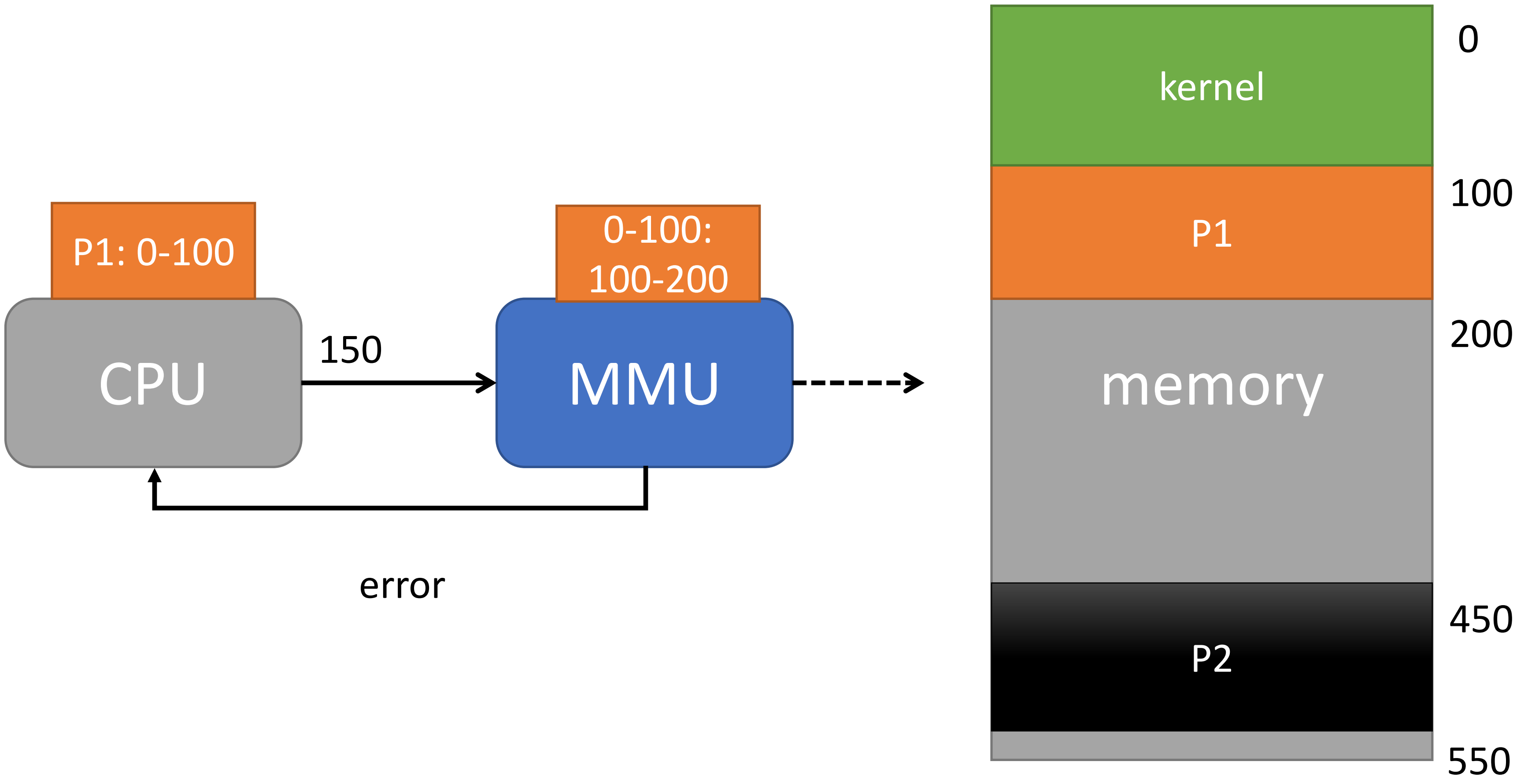
MMU: Virtual to Physical



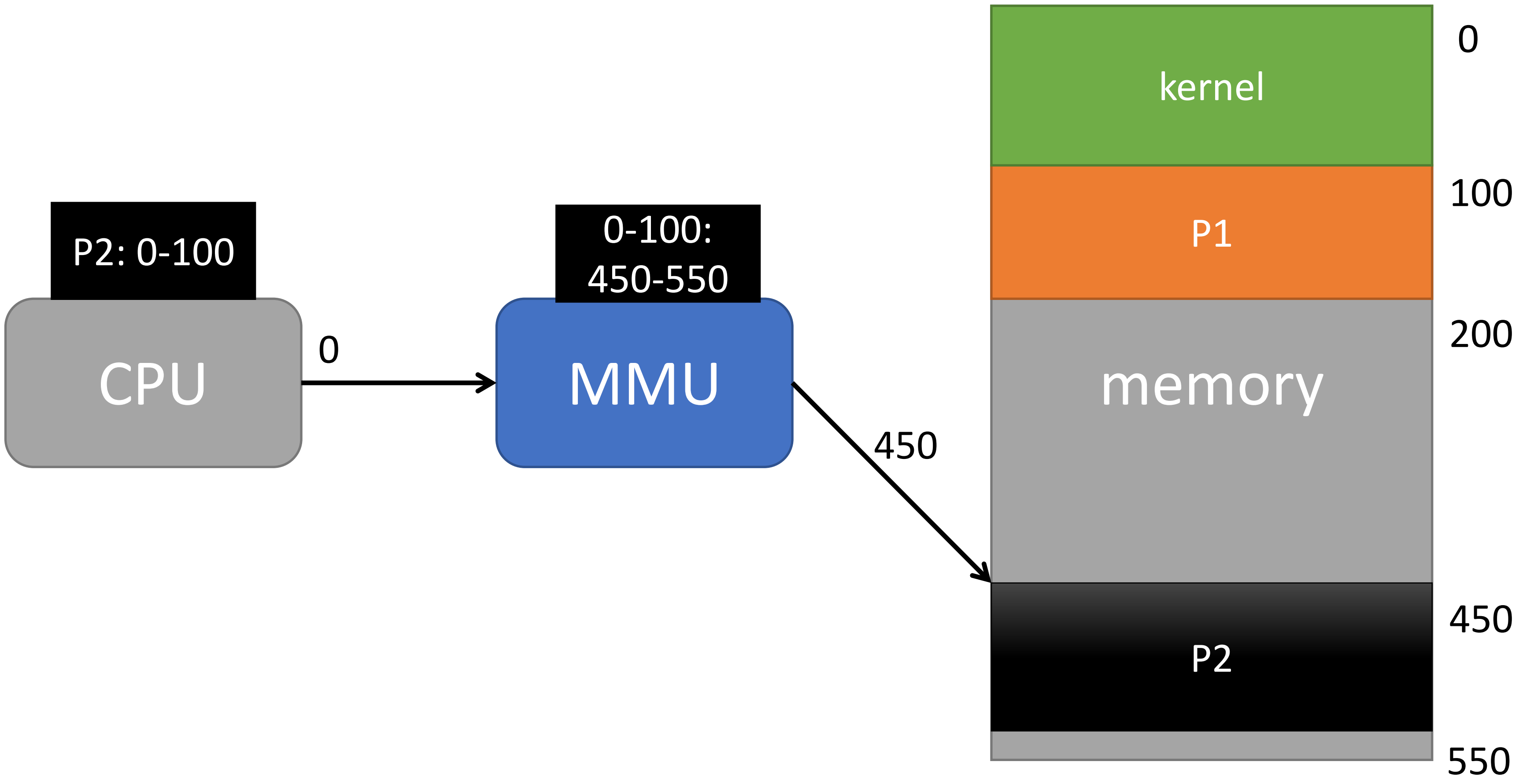
Mapping Example



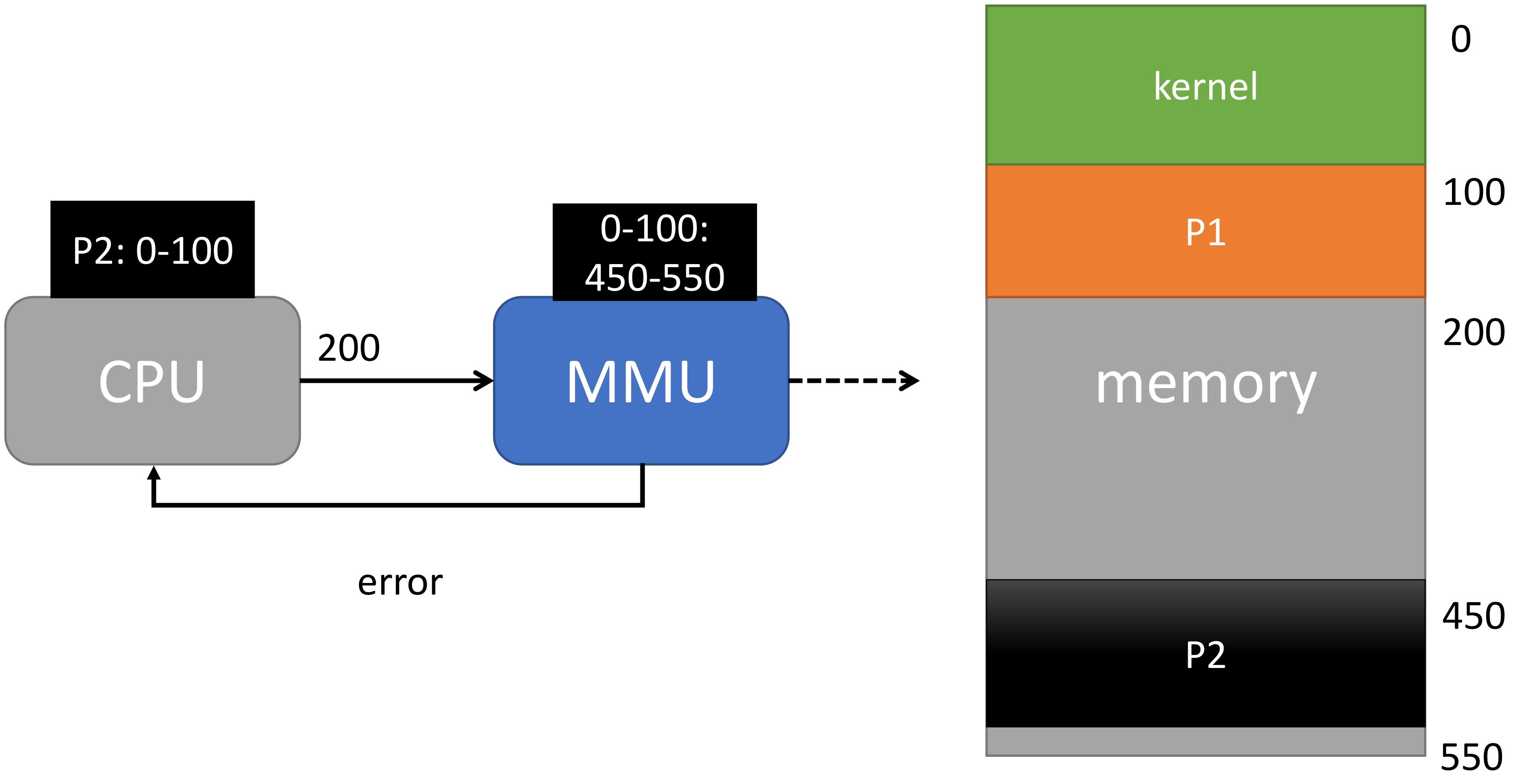
Mapping and Protection Example



Mapping Example 2



Mapping and Protection Example 2



C Code Example

```
void func() {
        int x = 3000;
        ...
        x = x + 3; // this is the line of code we are interested in
}
```

- **Load** a value from memory
- **Increment** it by three
- **Store** the value back into memory

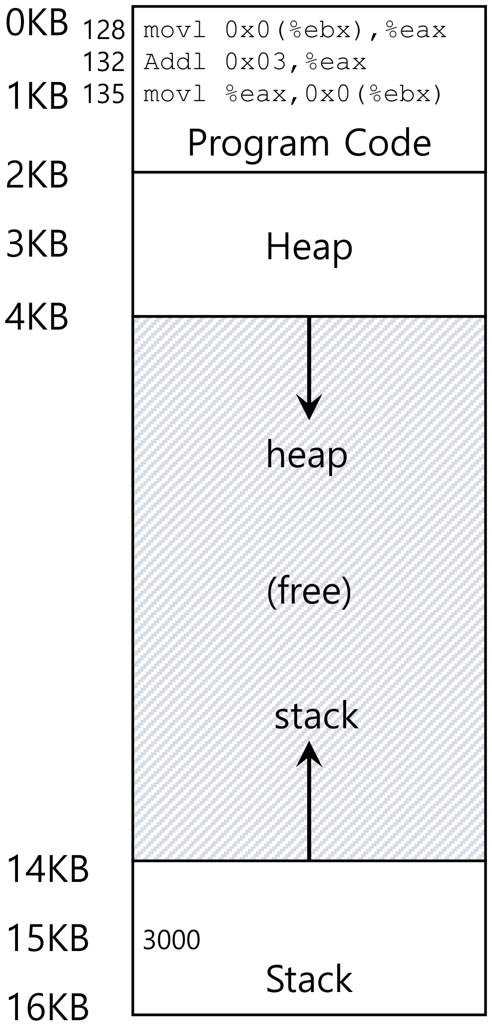
C Code Example

C → Assembly for `x = x + 3`

```
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
```

- Assume that the address of `x` was placed in `ebx` register.
- **Load** the value at that address into `eax` register.
- **Add** 3 to `eax` register.
- **Store** the value in `eax` back into memory.

Code in Virtual Memory

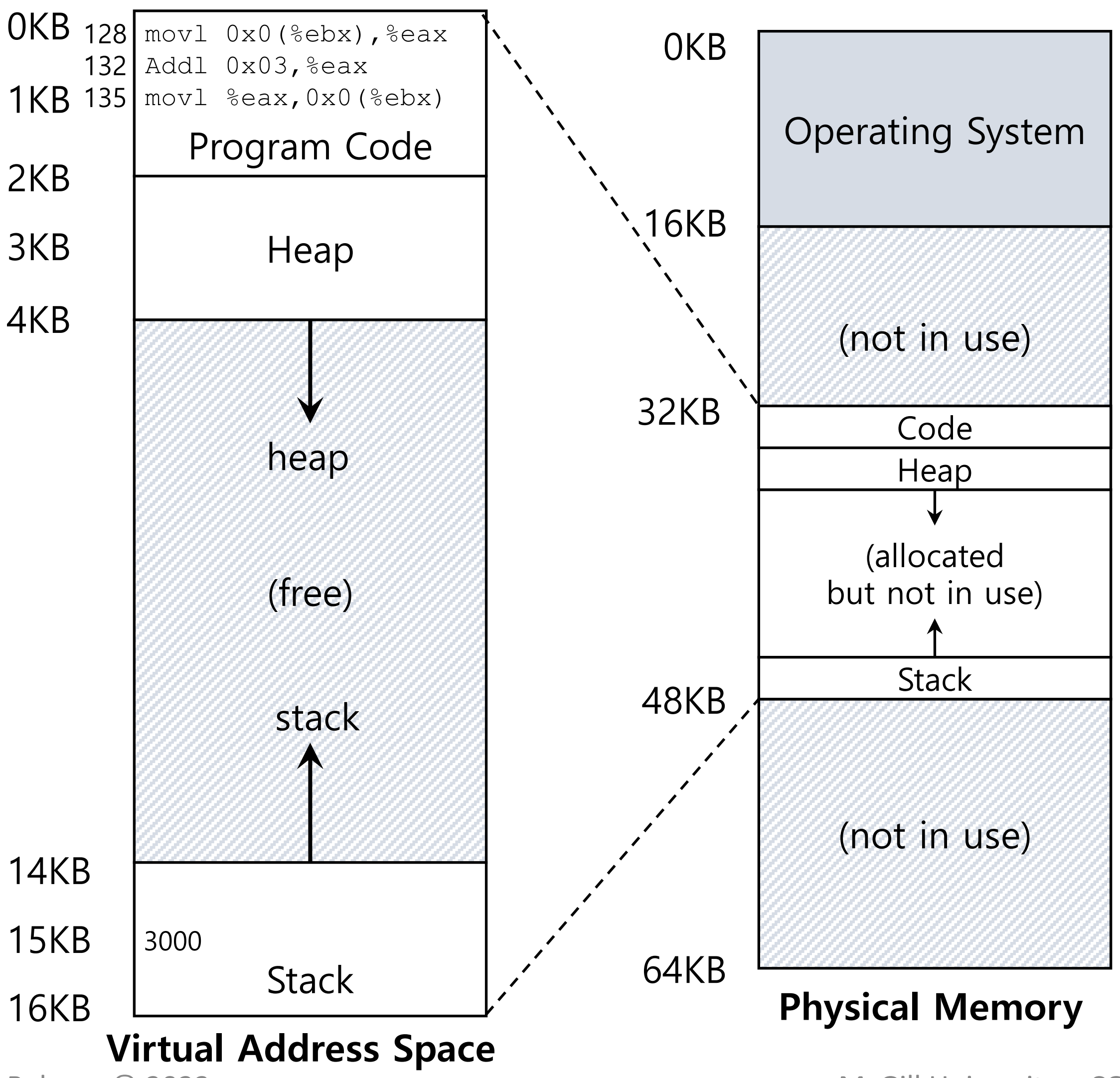


Virtual Address Space

`x = x + 3`

- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

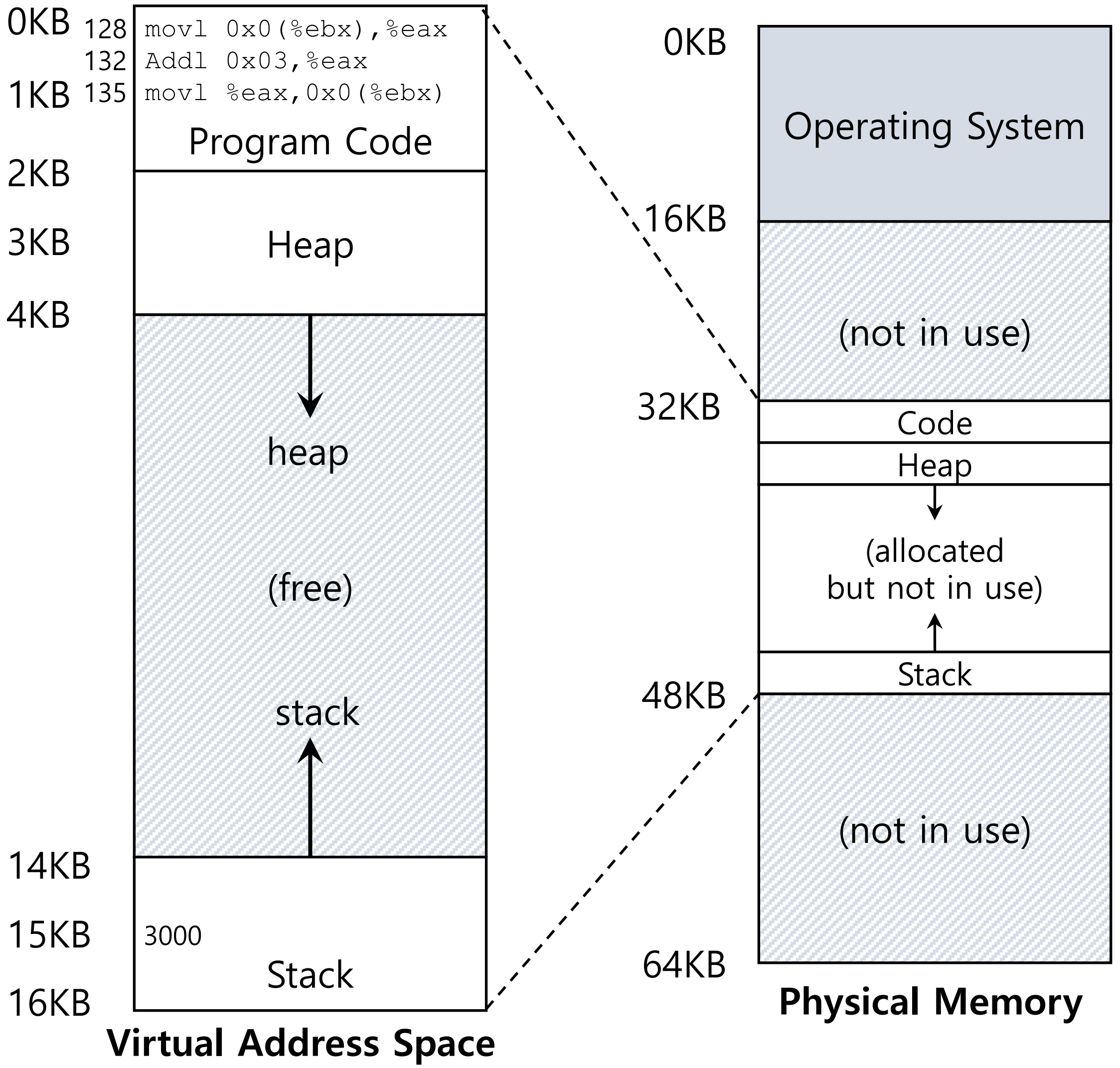
Code in Virtual Memory




```
x = x + 3
```

- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

Code in Virtual Memory

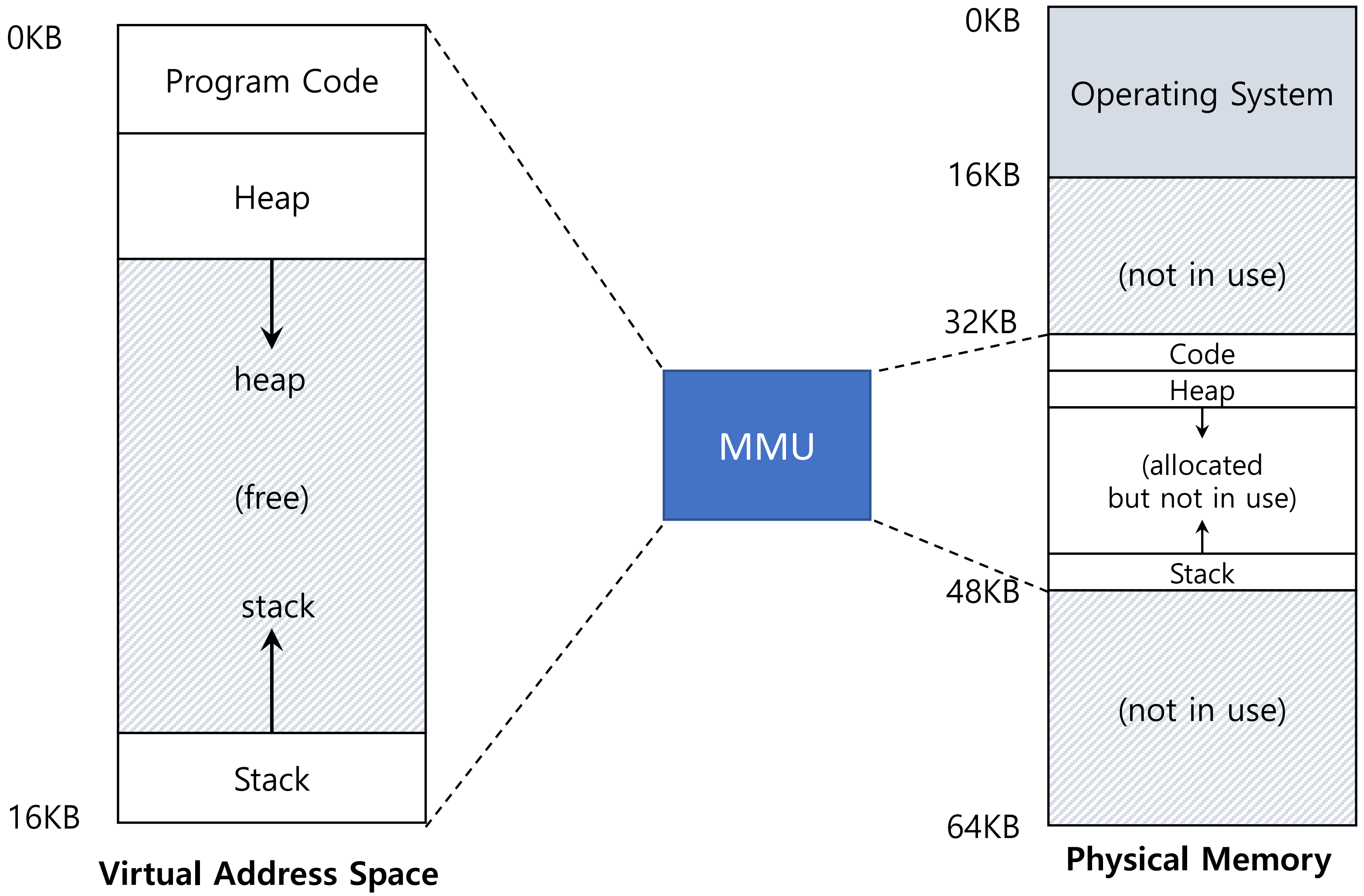


`x = x + 3`

- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- ~~Execute this instruction (no memory reference)~~
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

All these steps go through MMU

Virtual vs Physical Address Space



Size of Address Spaces

- **Maximum virtual address space size**
  - Limited by address **size of CPU**
  - Typically 32 or 64 bit addresses
  - So, 2^32 = 4 GB, or 2^64 = 16 Exabytes (BIG!)

- **Physical address space size**
  - Limited by **size of memory**
  - Nowadays, order of **tens/hundreds of GB**

Size of Virtual Address Spaces

32-bit address space

2^32 (4 GB)

64-bit address space

2^64 (16 Exabyte – big!)

Different Virtual to Physical Mapping Schemes

- Base and bounds
- Segmentation
- Paging

For each scheme

- Virtual address space
- Physical address space
- Virtual address
- MMU

Base and Bounds

Base and Bounds

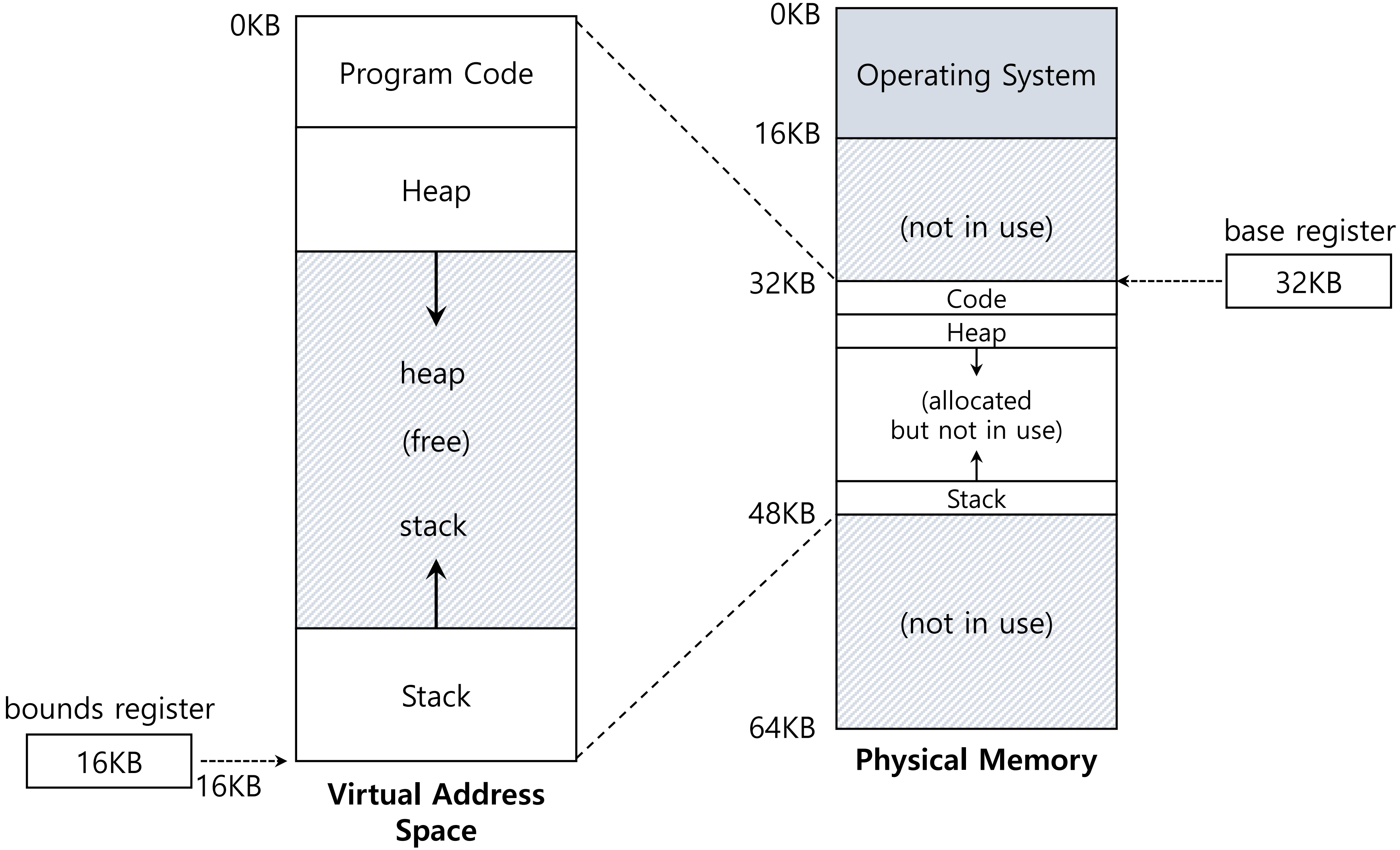
Virtual Address Space

- Linear address space : from 0 to MAX

Physical Address Space

- Linear address space: from BASE to BOUNDS=BASE+MAX

Base and Bounds



MMU for Base and Bounds

MMU

Relocation register: holds the base value

Limit register: holds the bounds value

When a program starts running, the OS decides **where** in physical memory a process should be **loaded (**i.e., what the **base value** is**)**.

Check for valid address: $0 \leq \mathit{virtual\ address} < \boldsymbol{bound}\ (\boldsymbol{in\ limit\ register})$

Address translation: $\mathit{physical\ address} = \mathit{virtual\ address} + \boldsymbol{base}\ (\boldsymbol{in\ relocation\ register})$

Base and Bounds: Example

- C - Language code

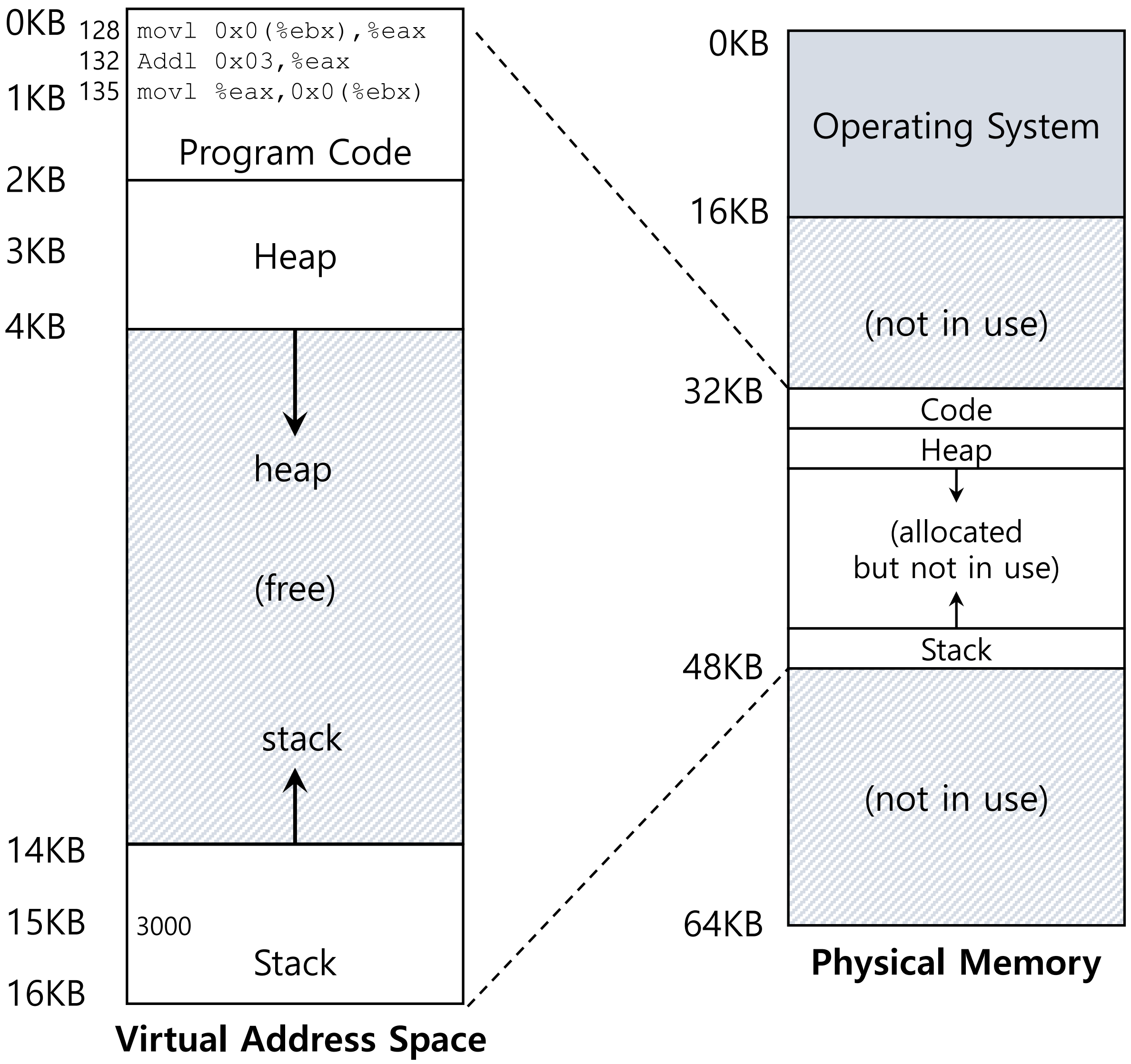
```
void func()
        int x = 3000;
        ...
        x = x + 3; // this is the line of code we are interested in
```

- Assembly

We'll look at this line

```
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
```

Base and Bounds: Example



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

- **Fetch** instruction at address 128

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

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

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

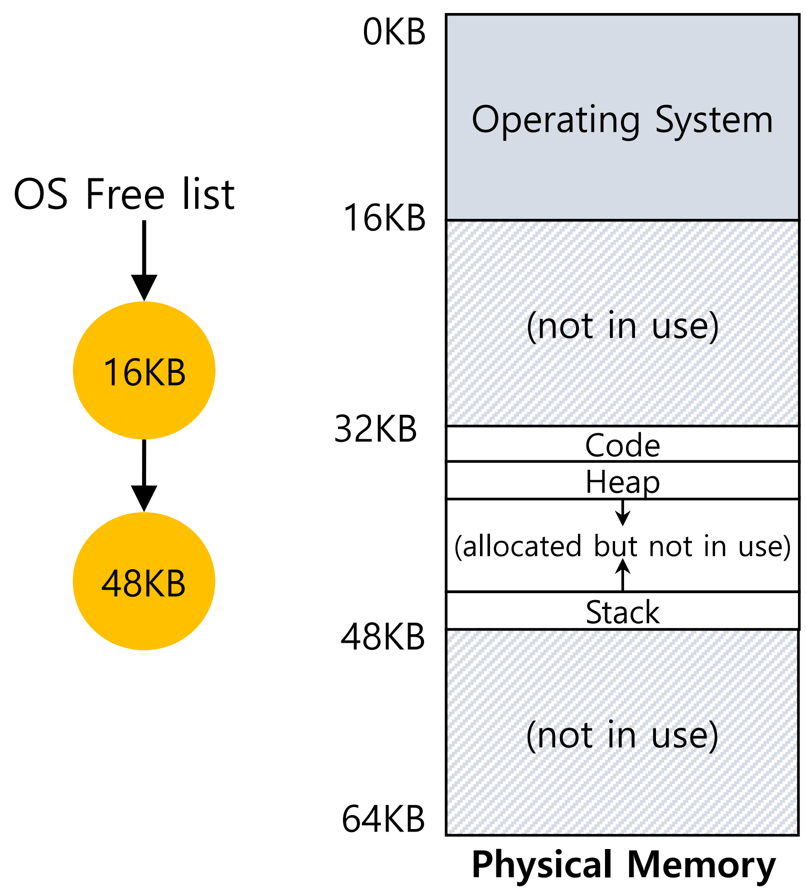
Base and Bounds: Main Memory Allocation

Main memory:

- Regions in use
- “Holes”, regions not in use
- New process needs to go in “holes”

Free list:

- A list of the range of the physical memory not in use.



Base and Bounds: Which “hole” to pick?

First-fit

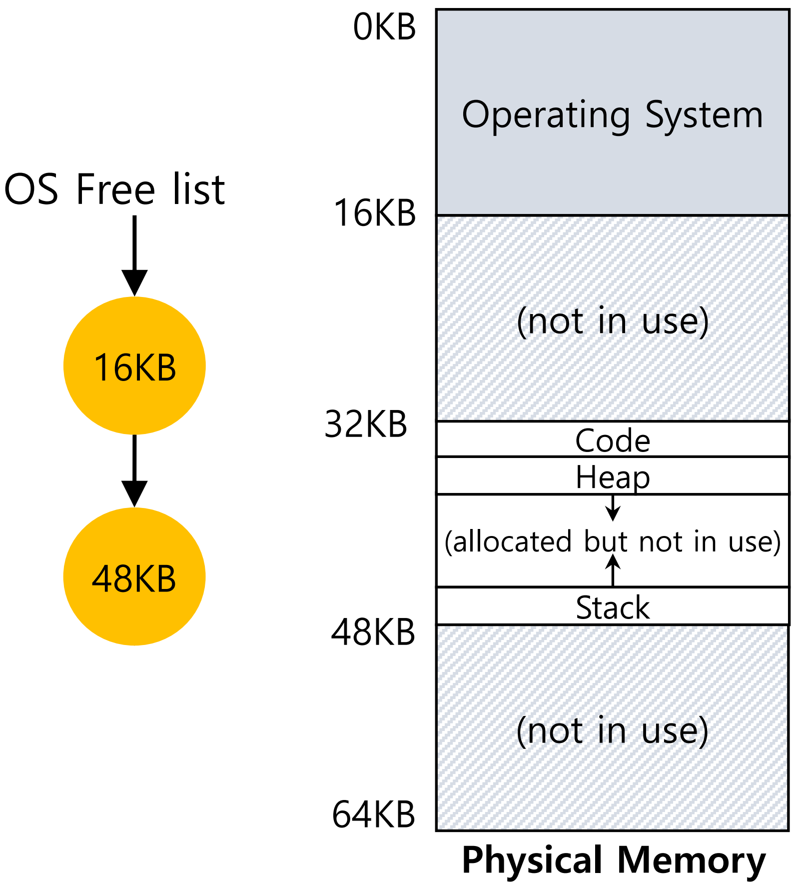
- Take first hole bigger than requested
- Easy to find

Best-fit

- Take smallest hole bigger than requested
- Leaves smallest hole behind

Worst-fit?!

- Takes largest hole
- Leaves biggest hole behind



Base and Bounds: (External) Fragmentation

Small holes become unusable

Part of memory cannot be used

Serious problem ☹

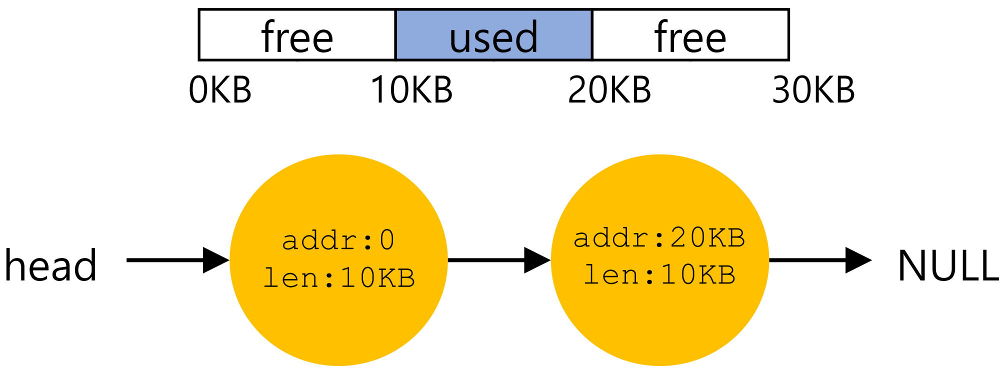
Base and Bounds: (External) Fragmentation

Small holes become unusable

Part of memory cannot be used

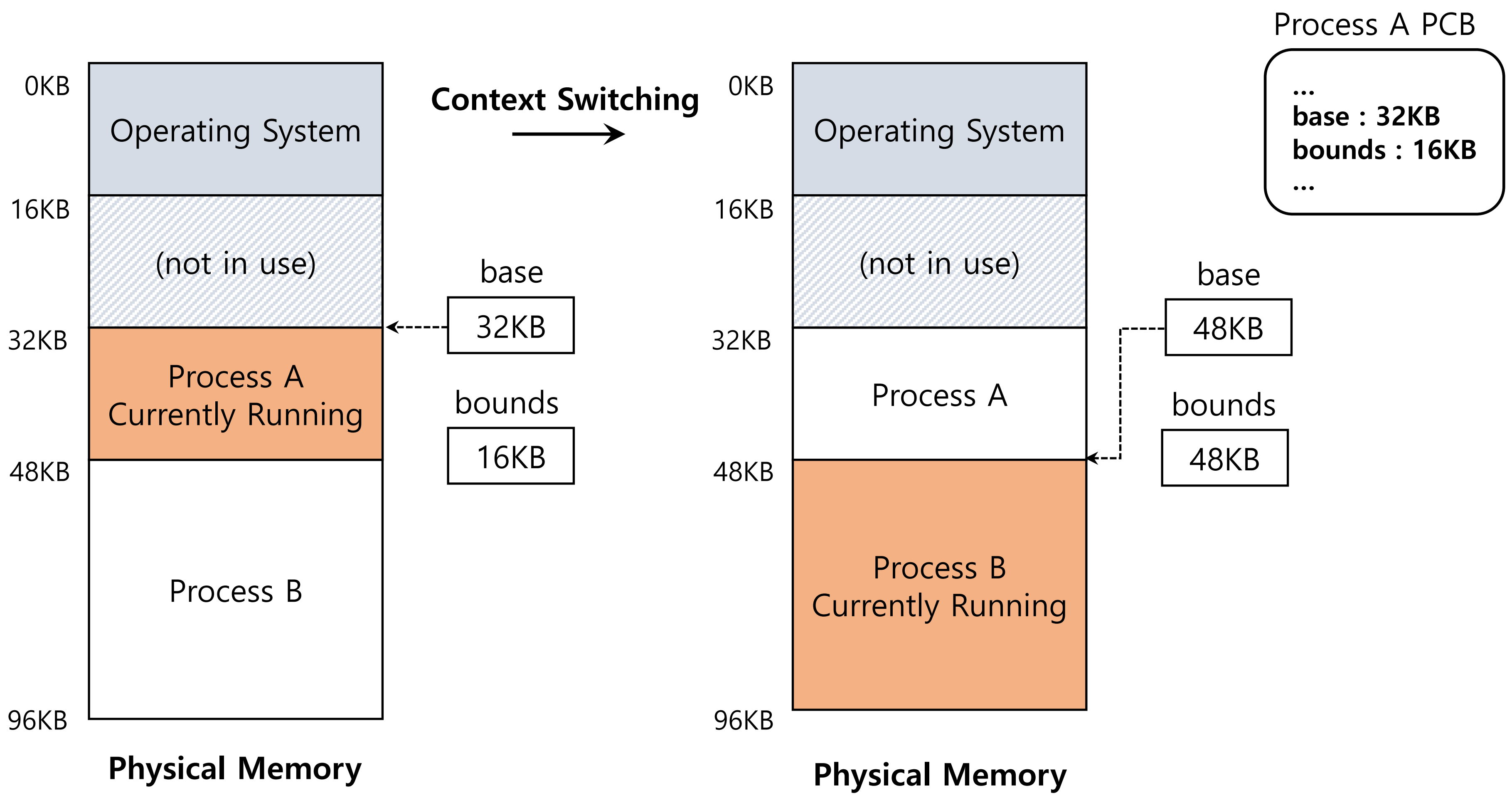
Serious problem ☹

Example:



Cannot allocate a 20KB chunk, even if there are 20KB that are free in memory.

Base and Bounds: Context Switch



Base and Bounds: Context Switch

OS (kernel mode)	Hardware	Program (user mode)
To start process A: alloc entry in process table **alloc memory for process** **set base/bound registers** return from trap (into A)		

Base and Bounds: Context Switch

OS (kernel mode)	Hardware	Program (user mode)
To start process A: alloc entry in process table **alloc memory for process** **set base/bound registers** return from trap (into A)		
	restore registers of A move to **user mode** jump to A's (initial) PC	

Base and Bounds: Context Switch

OS (kernel mode)	Hardware	Program (user mode)
To start process A: alloc entry in process table **alloc memory for process** **set base/bound registers** return from trap (into A)		
	restore registers of A move to **user mode** jump to A's (initial) PC	
		Process A runs: fetch instruction

Base and Bounds: Context Switch

OS (kernel mode)	Hardware	Program (user mode)
To start process A: alloc entry in process table **alloc memory for process** **set base/bound registers** return from trap (into A)		
	restore registers of A move to **user mode** jump to A's (initial) PC	
		Process A runs: fetch instruction
	translate virtual address perform fetch	

Base and Bounds: Context Switch

OS (kernel mode)	Hardware	Program (user mode)
To start process A: alloc entry in process table **alloc memory for process** **set base/bound registers** return from trap (into A)		
	restore registers of A move to **user mode** jump to A's (initial) PC	
		Process A runs: fetch instruction
	translate virtual address perform fetch	
		execute instruction

Base and Bounds: Context Switch (Cont’d)

OS (kernel mode)	Hardware	Program (user mode)
	if load/store: **ensure address is legal** **translate virtual address** perform load/store	

Base and Bounds: Context Switch (Cont'd)

OS (kernel mode)	Hardware	Program (user mode)
	if load/store: **ensure address is legal** **translate virtual address** perform load/store	(A runs…)

Base and Bounds: Context Switch (Cont'd)

OS (kernel mode)	Hardware	Program (user mode)
	if load/store: **ensure address is legal** **translate virtual address** perform load/store **Timer interrupt** move to **kernel mode** jump to handler	(A runs…)

Base and Bounds: Context Switch (Cont'd)

OS (kernel mode)	Hardware	Program (user mode)
	if load/store: **ensure address is legal** **translate virtual address** perform load/store	(A runs…)
	Timer interrupt move to **kernel mode** jump to handler	
Handler timer decide: stop A, run B save regs(A) **including base and bounds** to PCB_A restore regs(B) from PCB_B **including base and bounds** return from trap (into B)		

Base and Bounds: Context Switch (Cont'd)

OS (kernel mode)	Hardware	Program (user mode)
	restore registers to B move to **user mode** jump to B's PC	

Base and Bounds: Context Switch (Cont’d)

OS (kernel mode)	Hardware	Program (user mode)
	restore registers to B move to **user mode** jump to B’s PC	
		Process B runs execute bad load

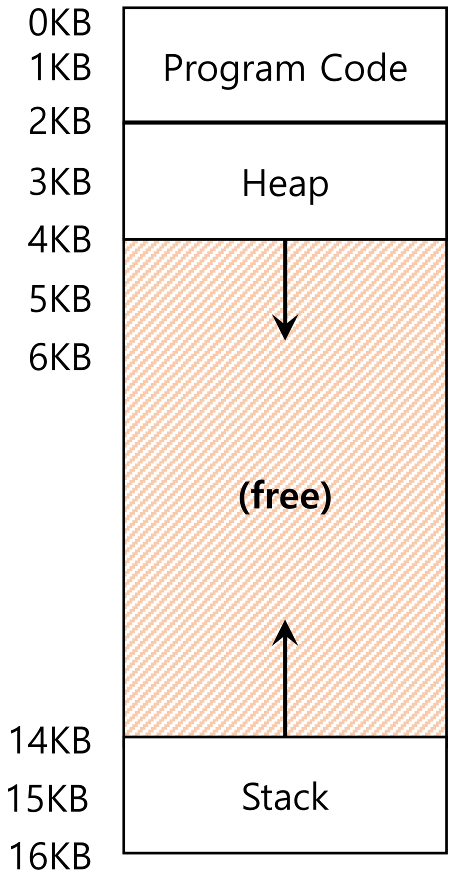
Base and Bounds: Context Switch (Cont’d)

OS (kernel mode)	Hardware	Program (user mode)
	restore registers to B move to **user mode** jump to B’s PC	
		Process B runs execute bad load
	load is out-of-bounds move to **kernel mode** jump to trap handler	

Base and Bounds: Context Switch (Cont'd)

OS (kernel mode)	Hardware	Program (user mode)
	restore registers to B move to **user mode** jump to B's PC	
		Process B runs execute bad load
	load is out-of-bounds move to **kernel mode** jump to trap handler	
Handle the trap decide: kill B deallocate B's memory free B's entry in process table		

Base and Bounds: (Internal) Fragmentation



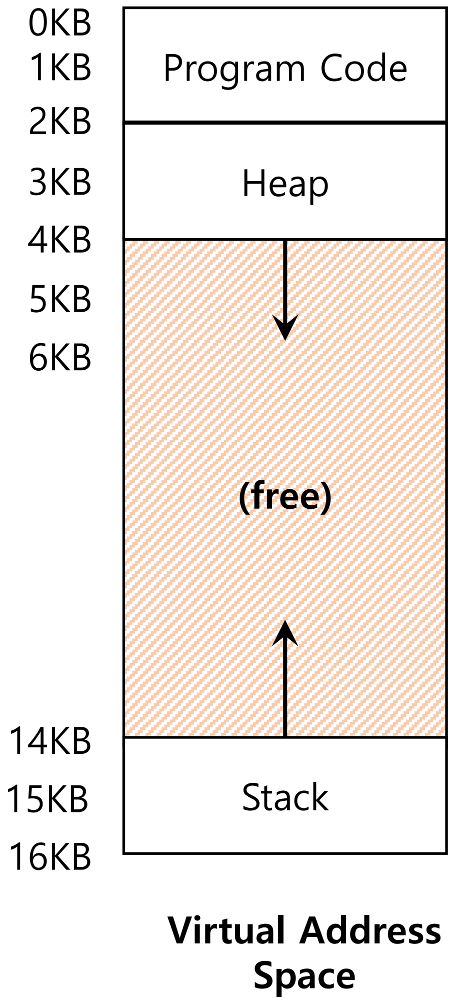
Virtual Address Space

- **Big chunk of "free"** space
- "free" space **takes up** physical memory.
- Inefficient
- (Internal) memory fragmentation

Different Virtual to Physical Mapping Schemes

- Base and bounds
- Segmentation
- (Simplified) Paging

Base and Bounds: (Internal) Fragmentation



- **Big chunk of “free”** space
- “free” space **takes up** physical memory.
- Inefficient
- (Internal) memory fragmentation

Segmentation

Segmentation

Virtual Address Space

- Two-dimensional
- Set of segments 0 .. n
- Each segment i is linear from 0 to MAX_i

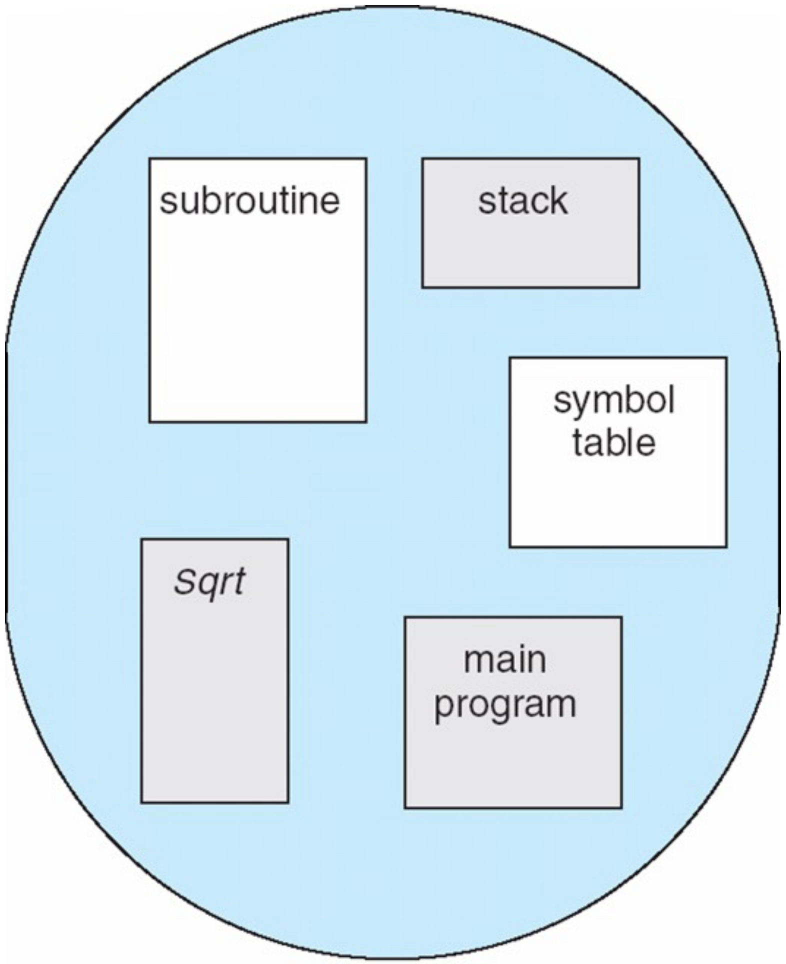
Physical Address Space

- Set of segments, each linear

What is a Segment?

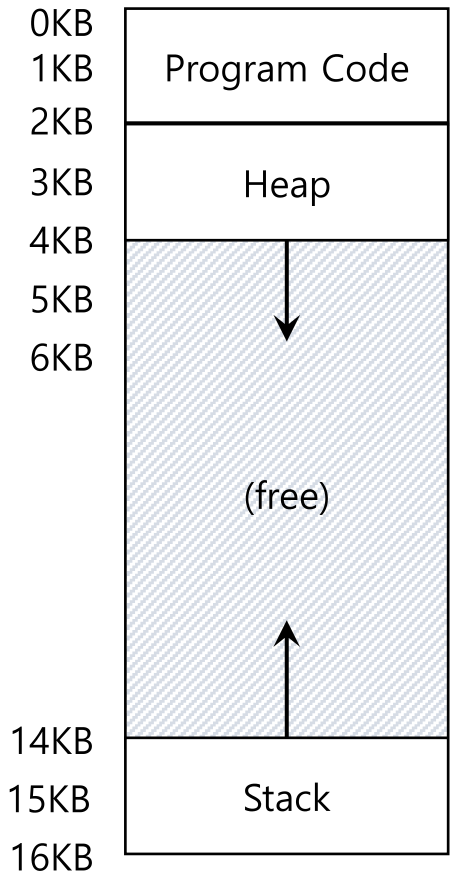
- Anything you want it to be

- Typical examples:
  - **Code**
  - **Heap**
  - **Stack**

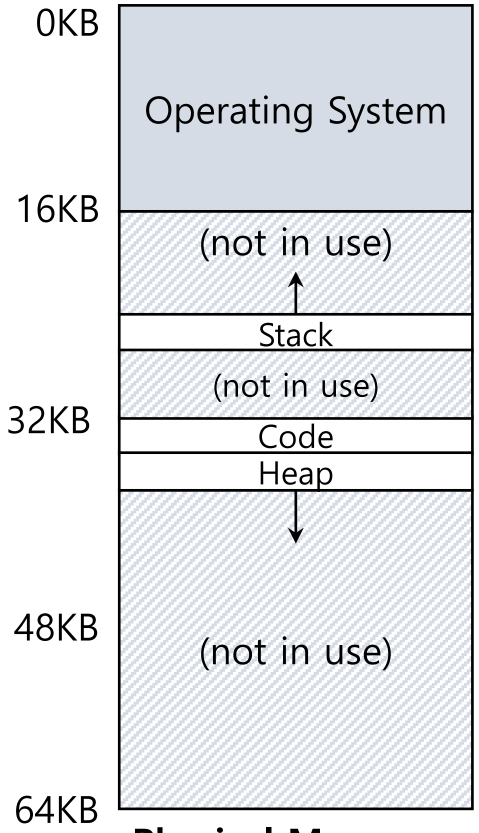


Segments

Segmentation Example



Virtual Address Space



Physical Memory

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

Segmentation: Virtual Address

Two-dimensional address:

- Segment number s
- Offset d **within segment** (starting at 0)

It is like multiple base-and-bounds

Segmentation: Virtual Address

Two-dimensional address:

- Segment number s
- Offset d **within segment** (starting at 0)

It is like multiple base-and-bounds

Further Reading

Operating Systems: Three Easy Pieces by R. & A. Arpaci-Dusseau

Chapters 12–18

https://pages.cs.wisc.edu/~remzi/OSTEP/

Credits:

Some slides adapted from the OS courses of Profs. Remzi and Andrea Arpaci-Dusseau (University of Wisconsin-Madison), Prof. Willy Zwaenepoel (University of Sydney), and Prof. Youjip Won (Hanyang University).