Week 6

Memory Management: Virtual Memory

Max Kopinsky

11 February 2025

Key Concepts

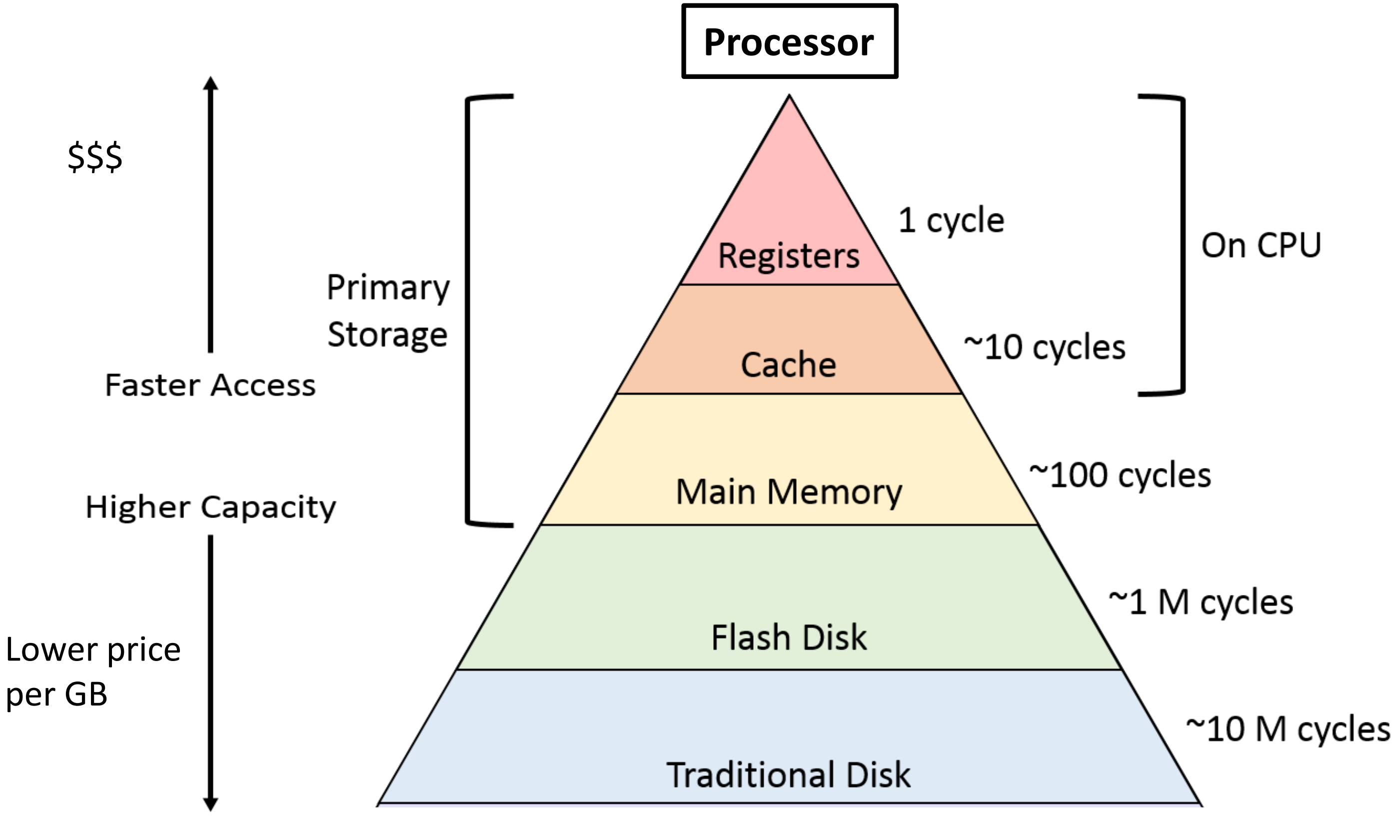
- Virtual and physical address spaces
- Mapping between virtual and physical address
- Different mapping methods:
  - Base and bounds, Segmentation, Paging
- Sharing, protection, memory allocation

Memory: the Dream

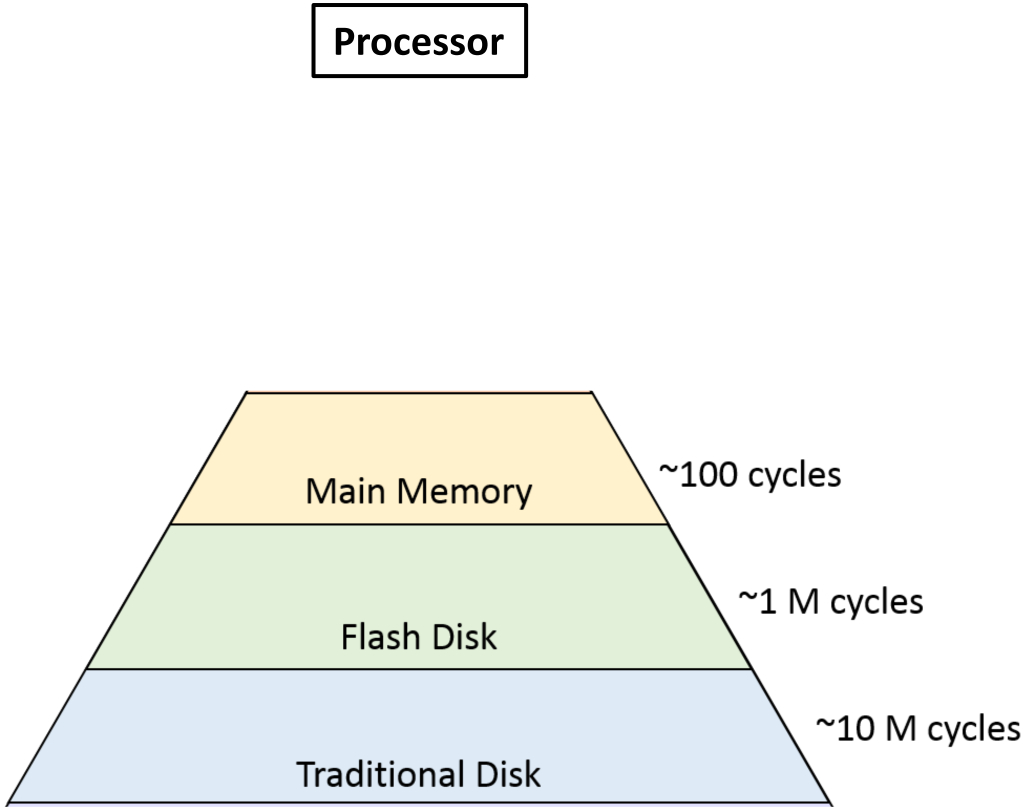
Memory that every programmer wants:

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

Real world: Memory Hierarchy



OS Memory Management

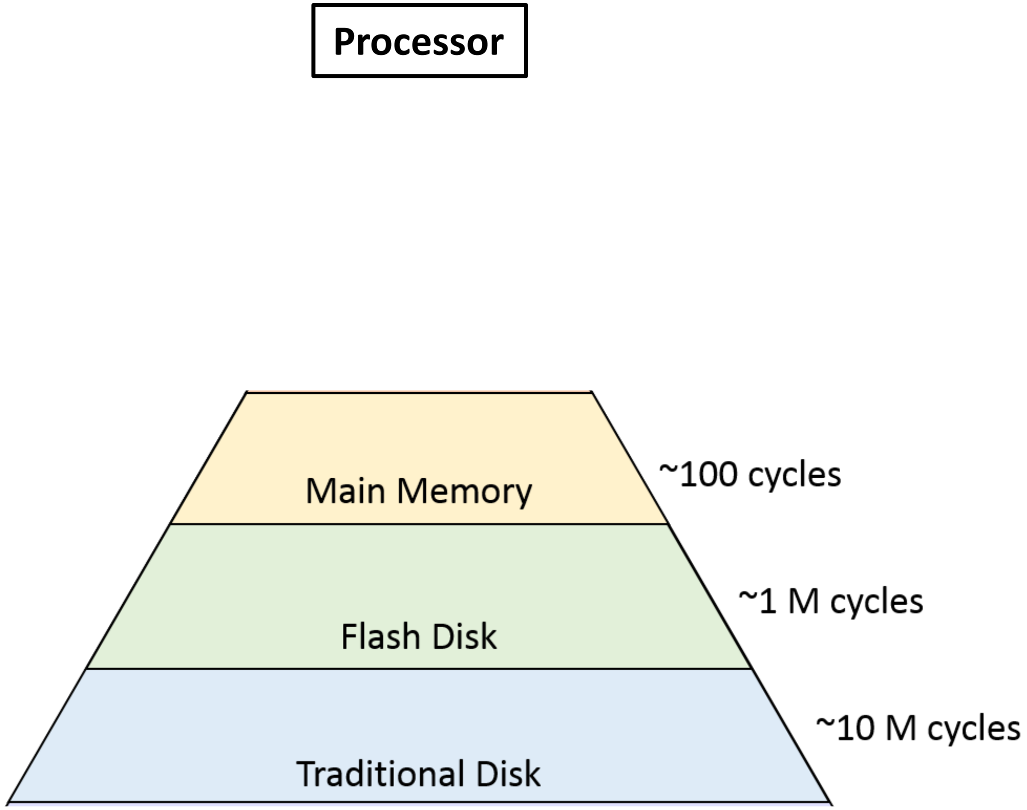


Simplifying Assumption

For this week's lecture only:

All of a program must be in main memory

Will revisit assumption next week

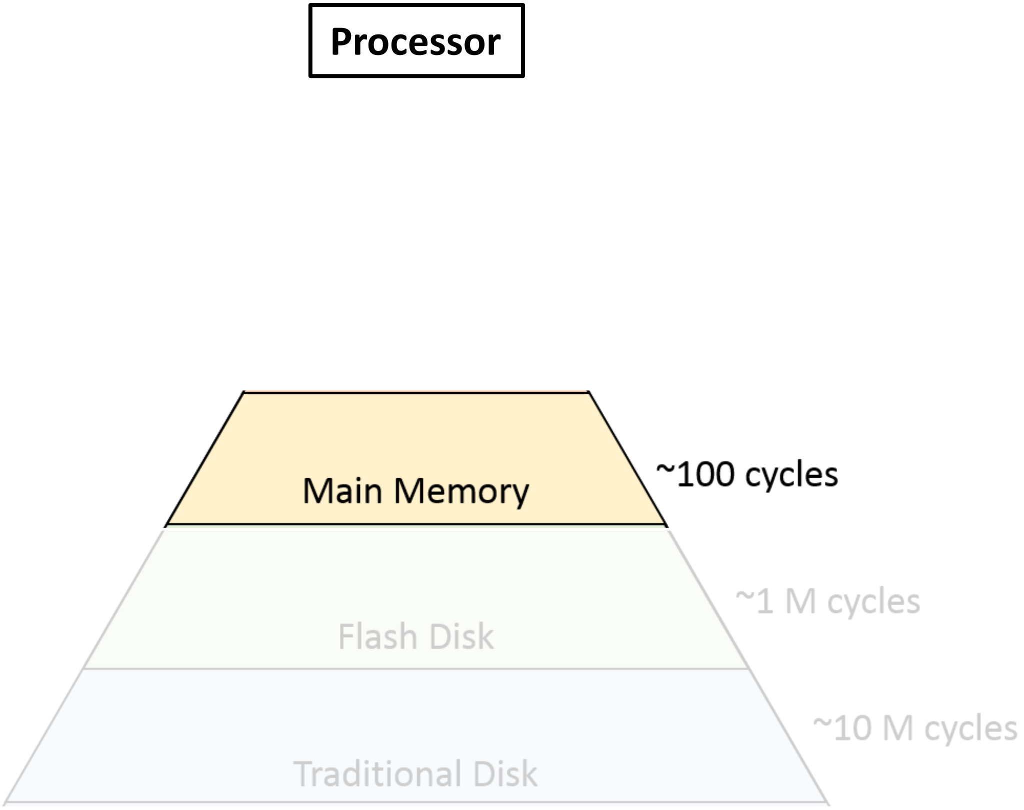


Simplifying Assumption

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

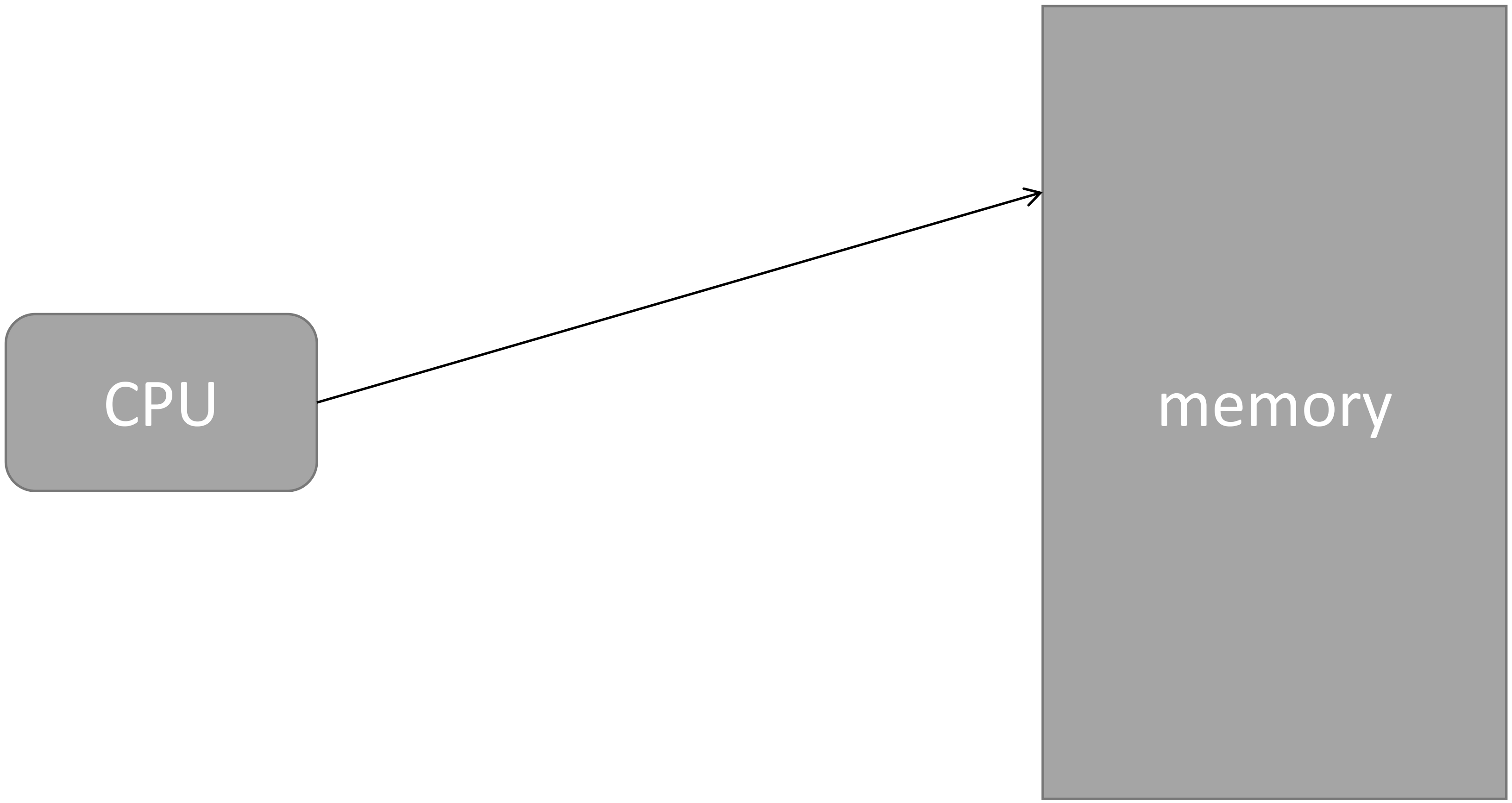
- Processes are not aware that memory is shared
- Works regardless of number and/or location of processes

Protection

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

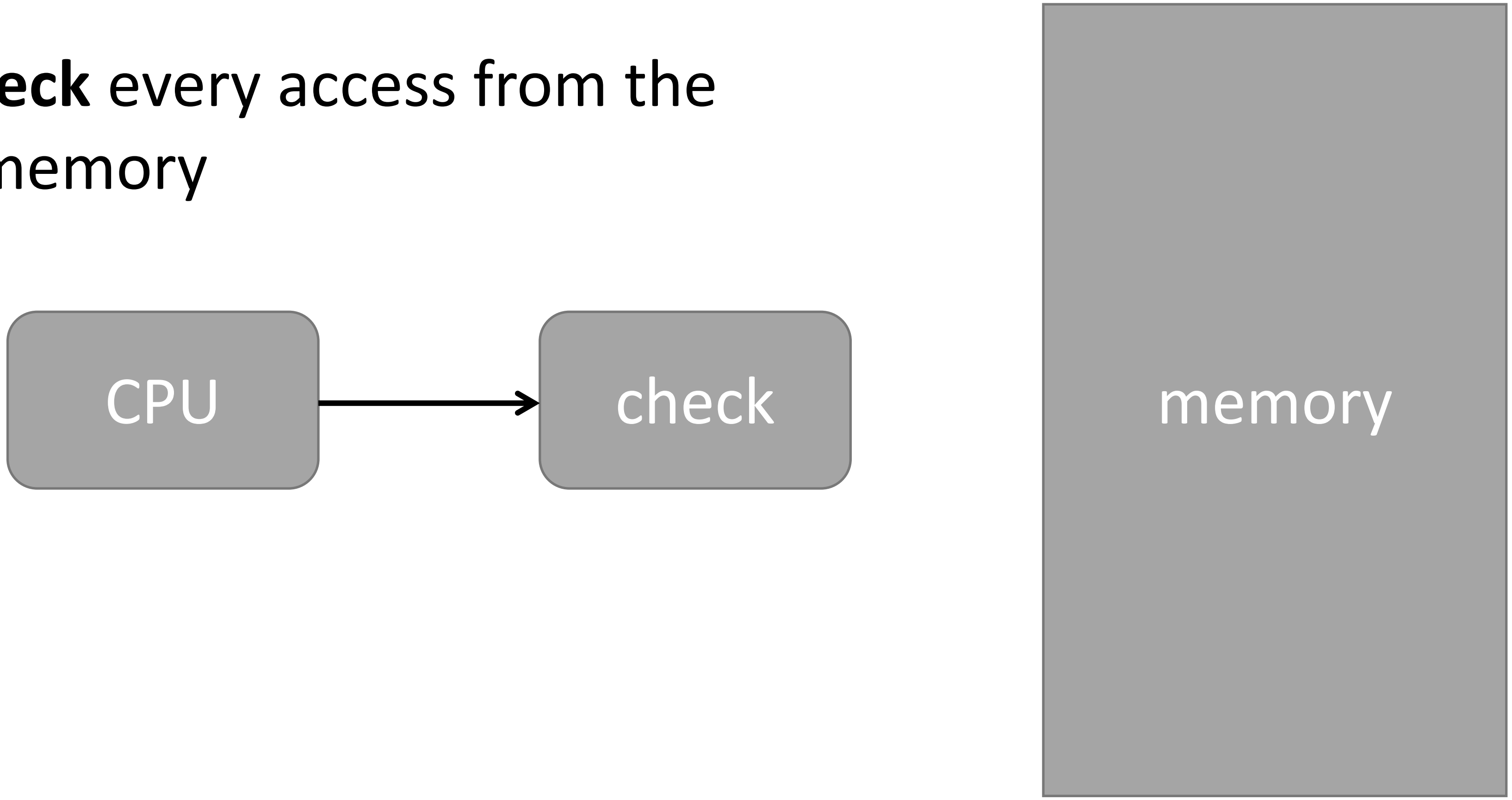
- of another process
- of the kernel

Unprotected Access

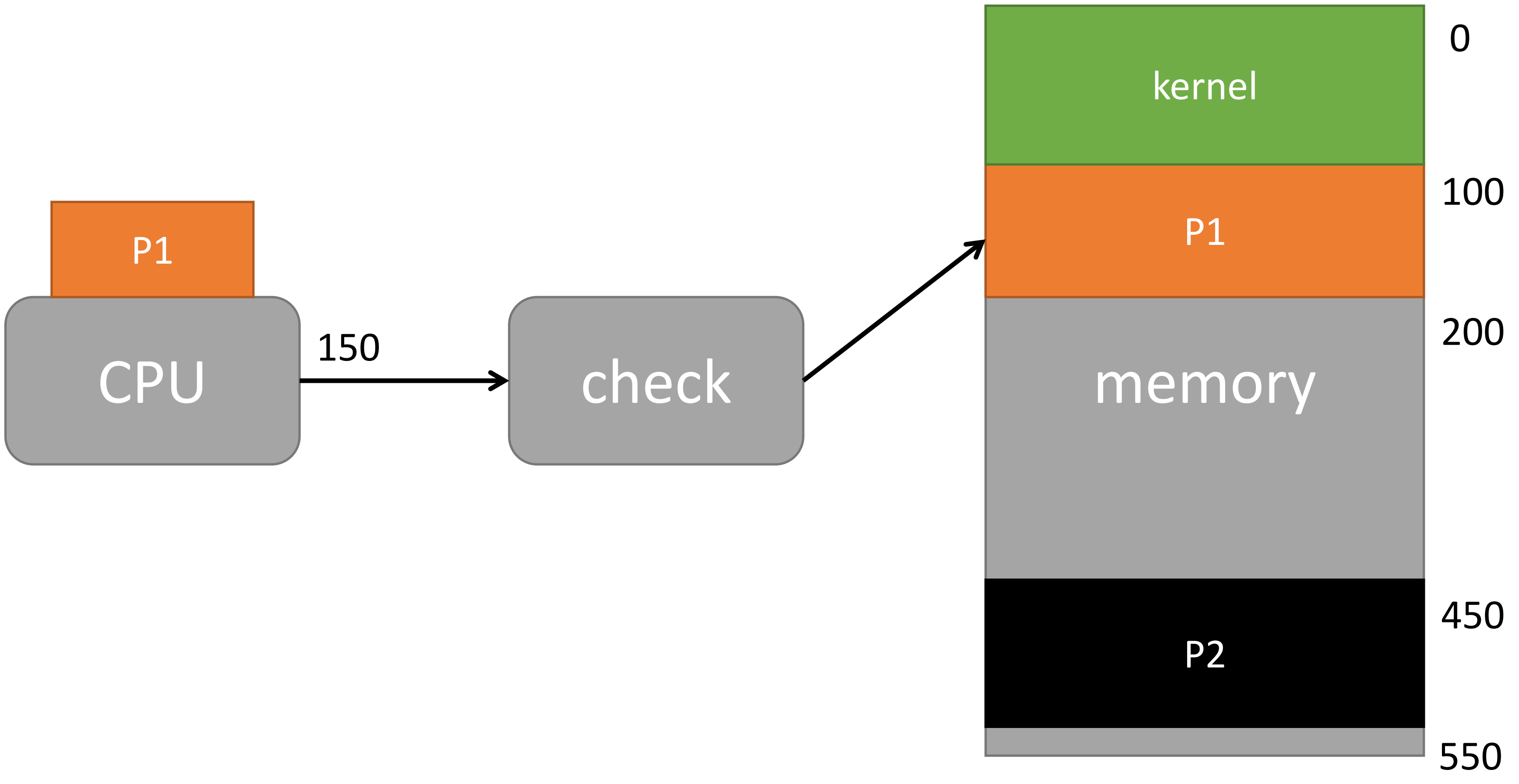


Protected Access

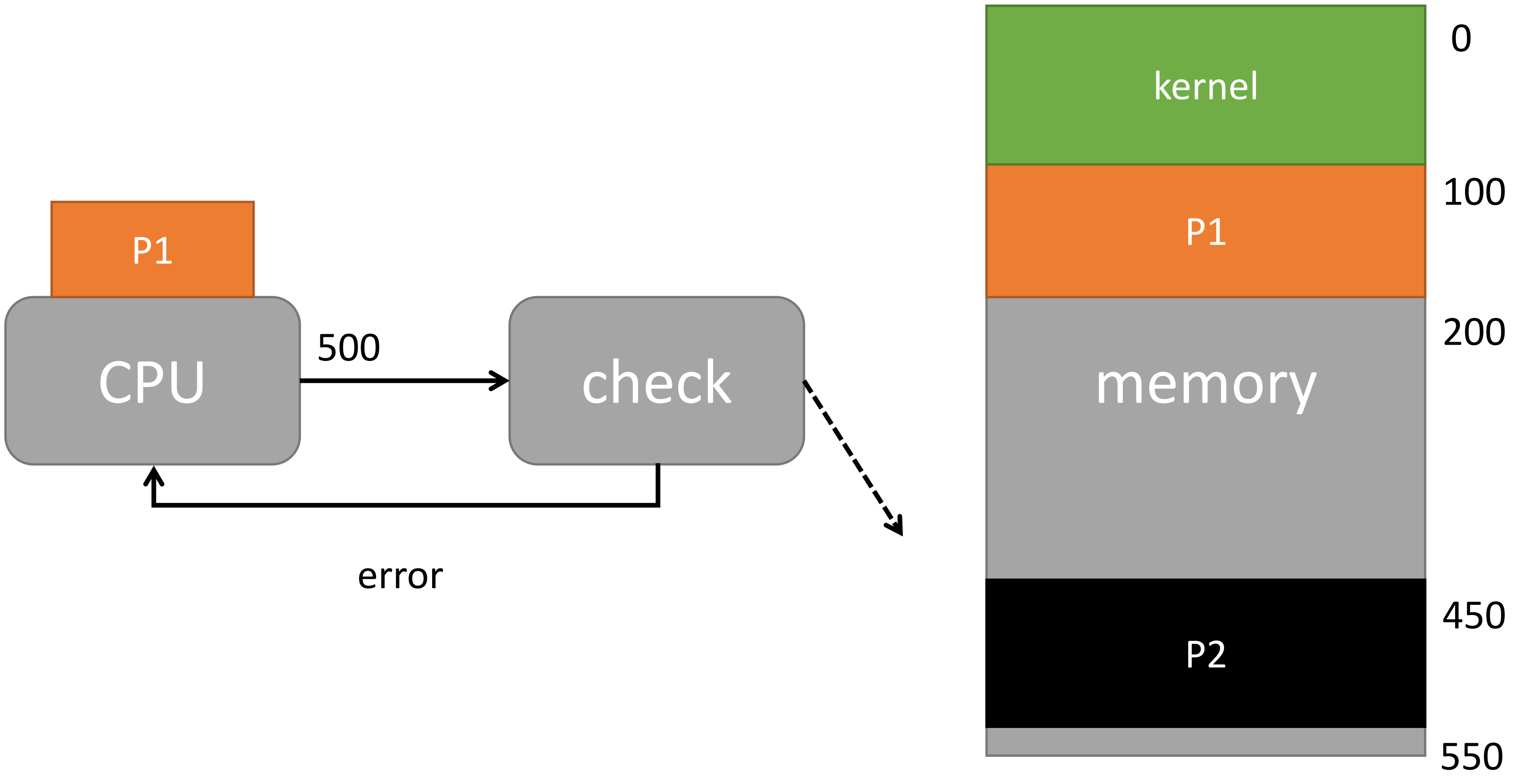
Must check every access from the CPU to memory



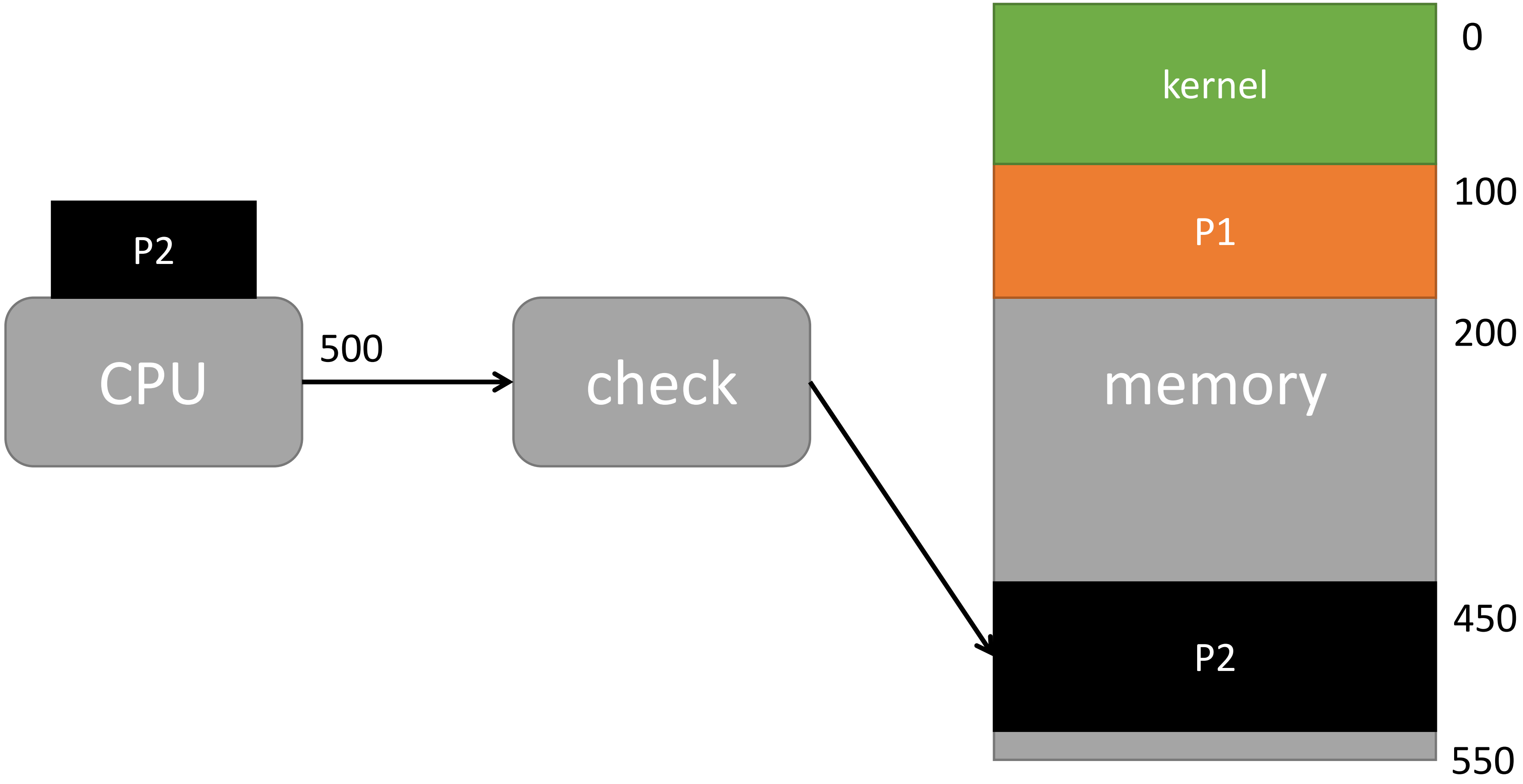
Protection: Examples



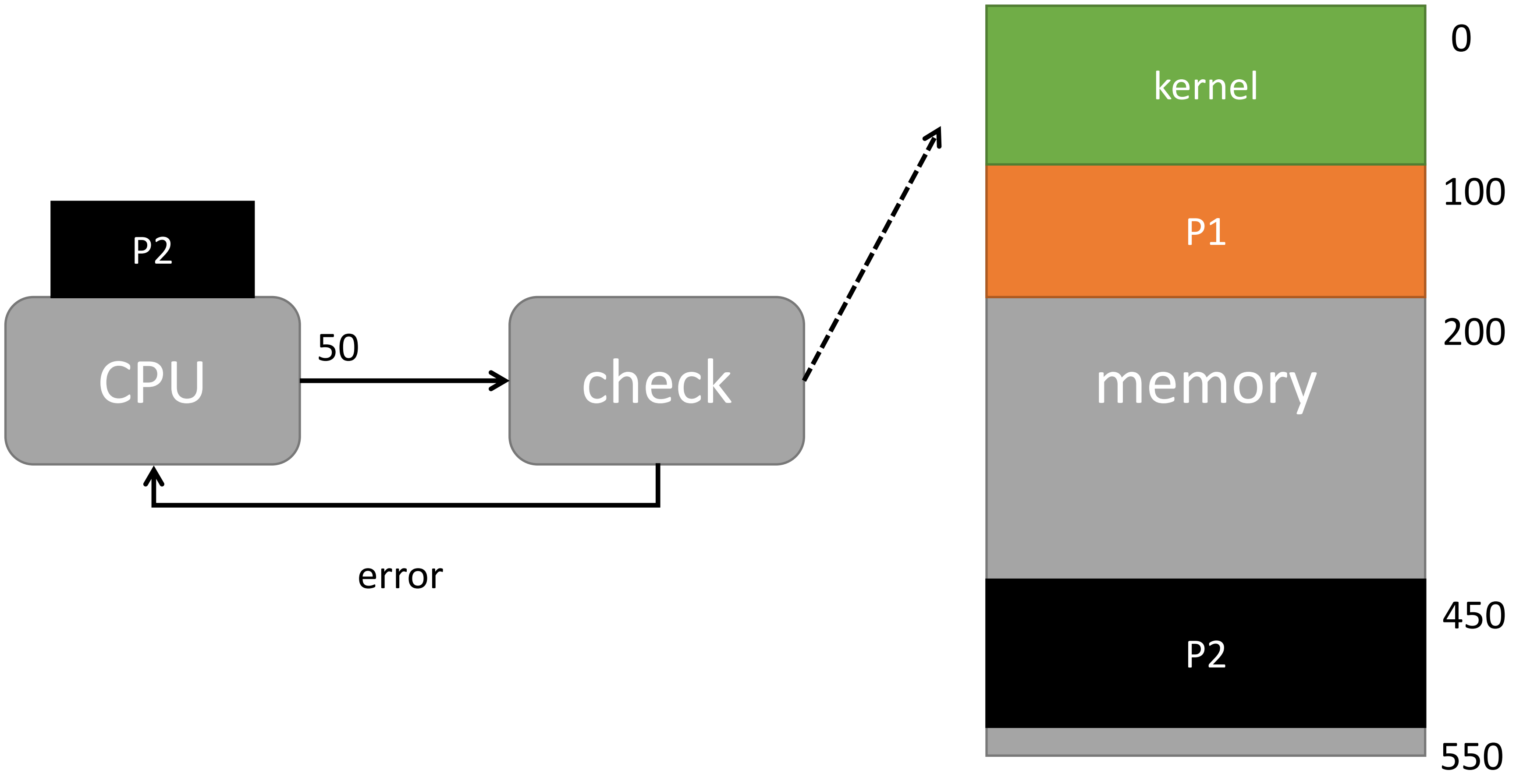
Protection: Examples



Protection: Examples



Protection: Examples



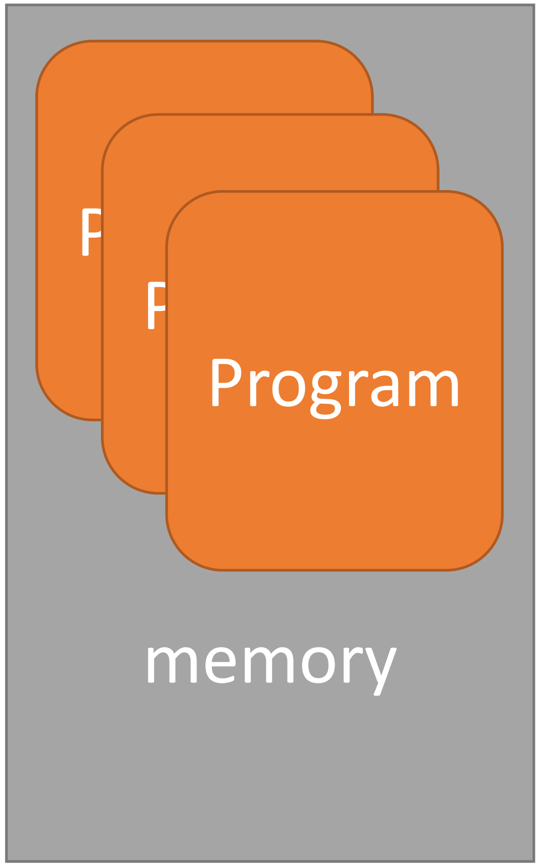
Transparency

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

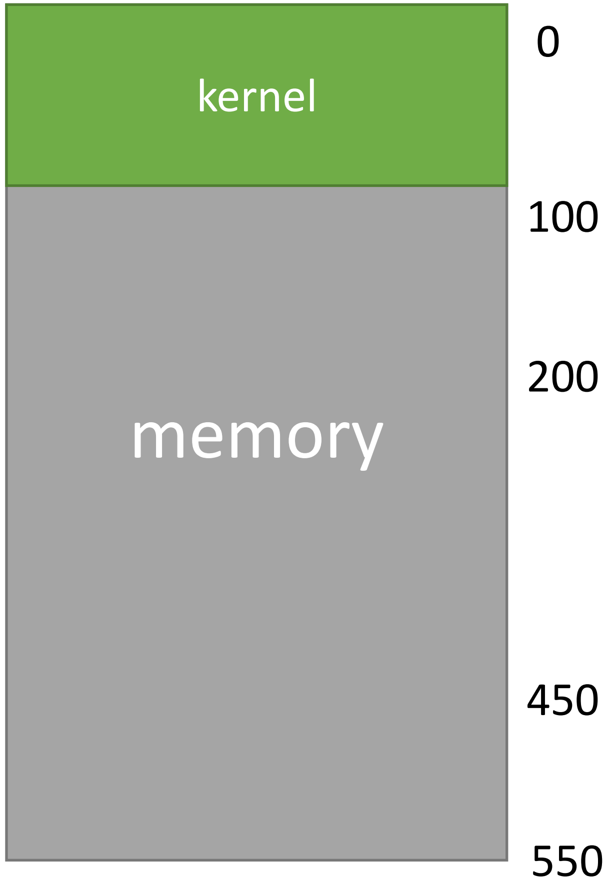
- Where to locate the kernel?
- How many processes to allow?
- What memory to allocate to processes?

Allocating Main Memory for Kernel

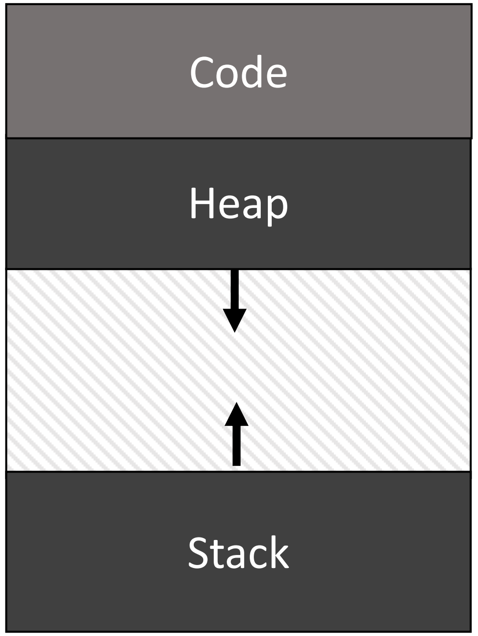
Almost always in low memory

Why? (x86) Interrupt vectors are in low memory

Now it's just convention.

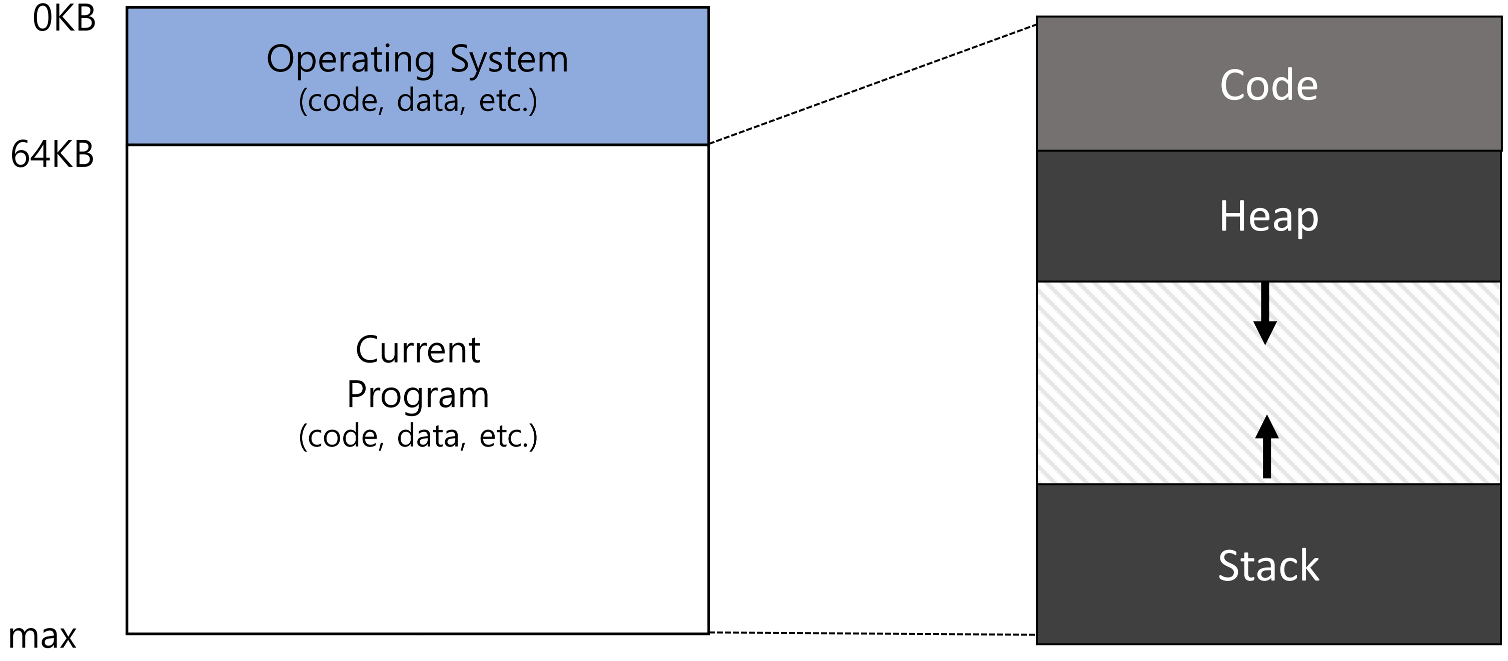


Main Memory Allocation for Processes



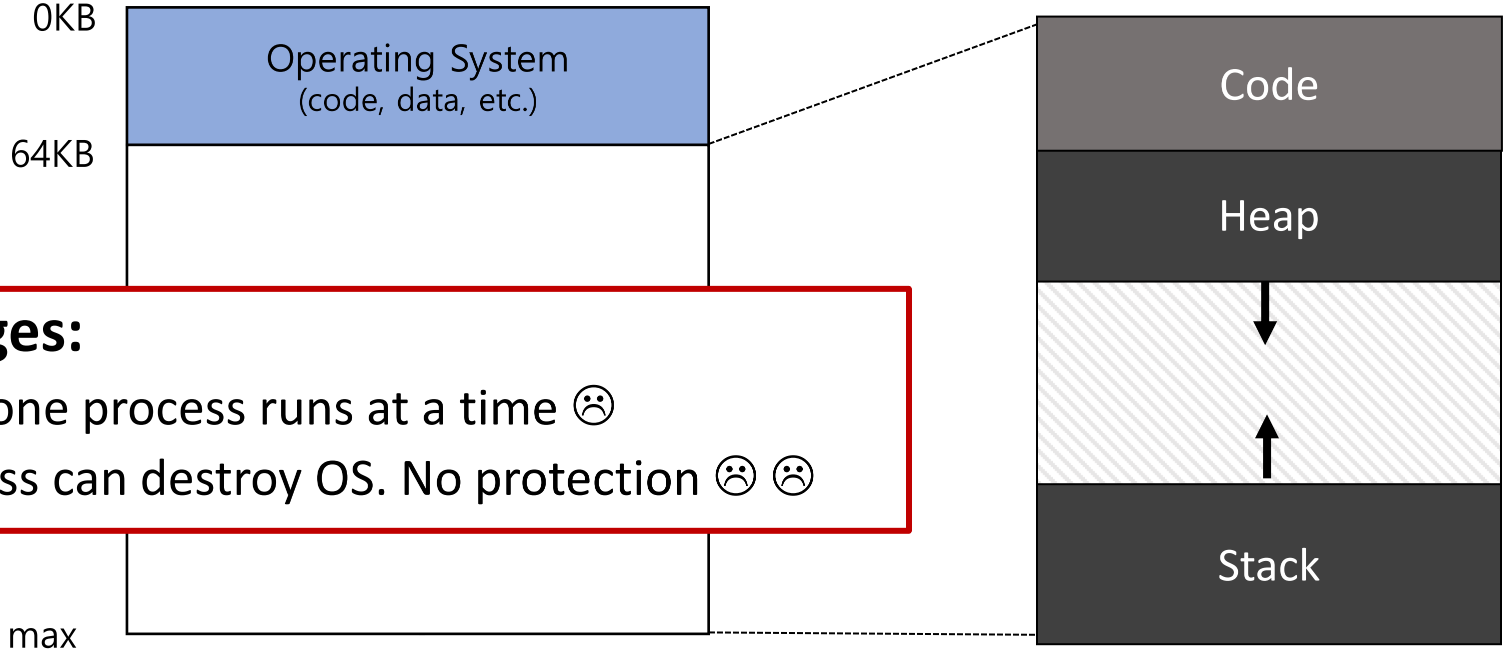
Early days: Uniprogramming

- 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?

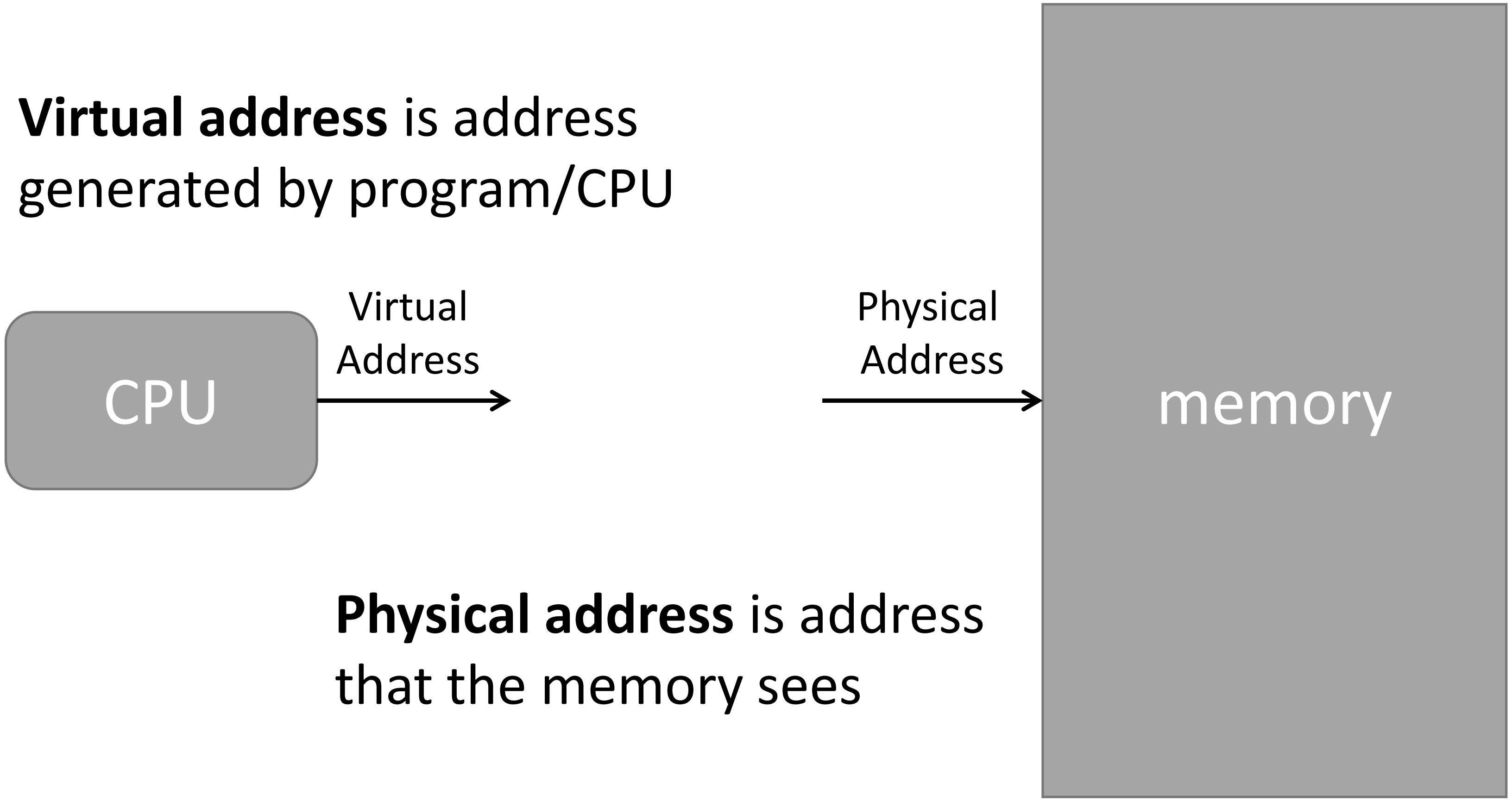
Virtual vs. Physical address space

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

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

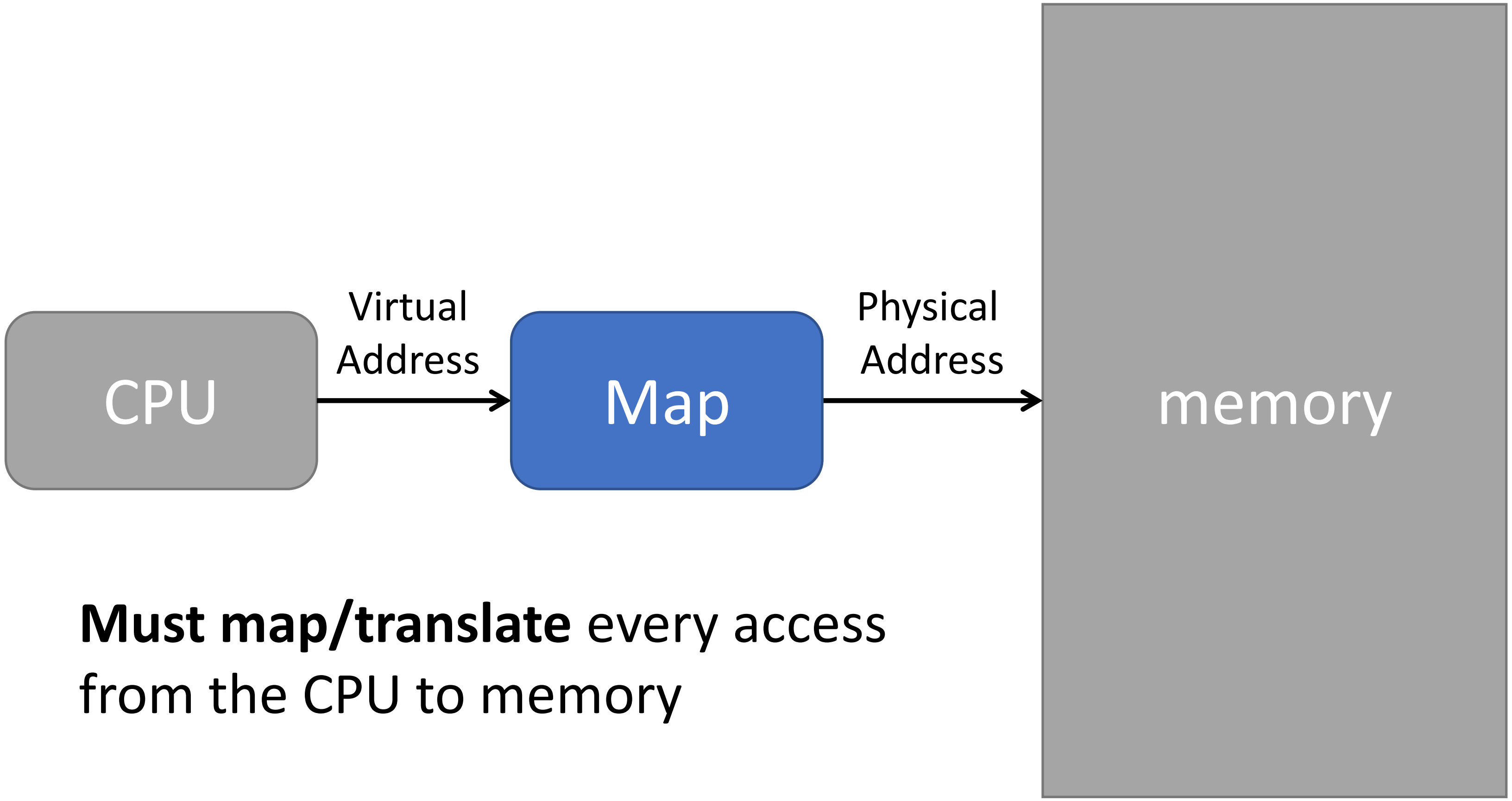
Virtual vs. Physical

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

Software Translation

- Add small circuit between CPU and RAM
- Circuit causes interrupts **whenever** user program accesses RAM
- Operating System handling:
  - Determine (virtual) address desired by program
  - Translate to physical address
  - Access correct physical address

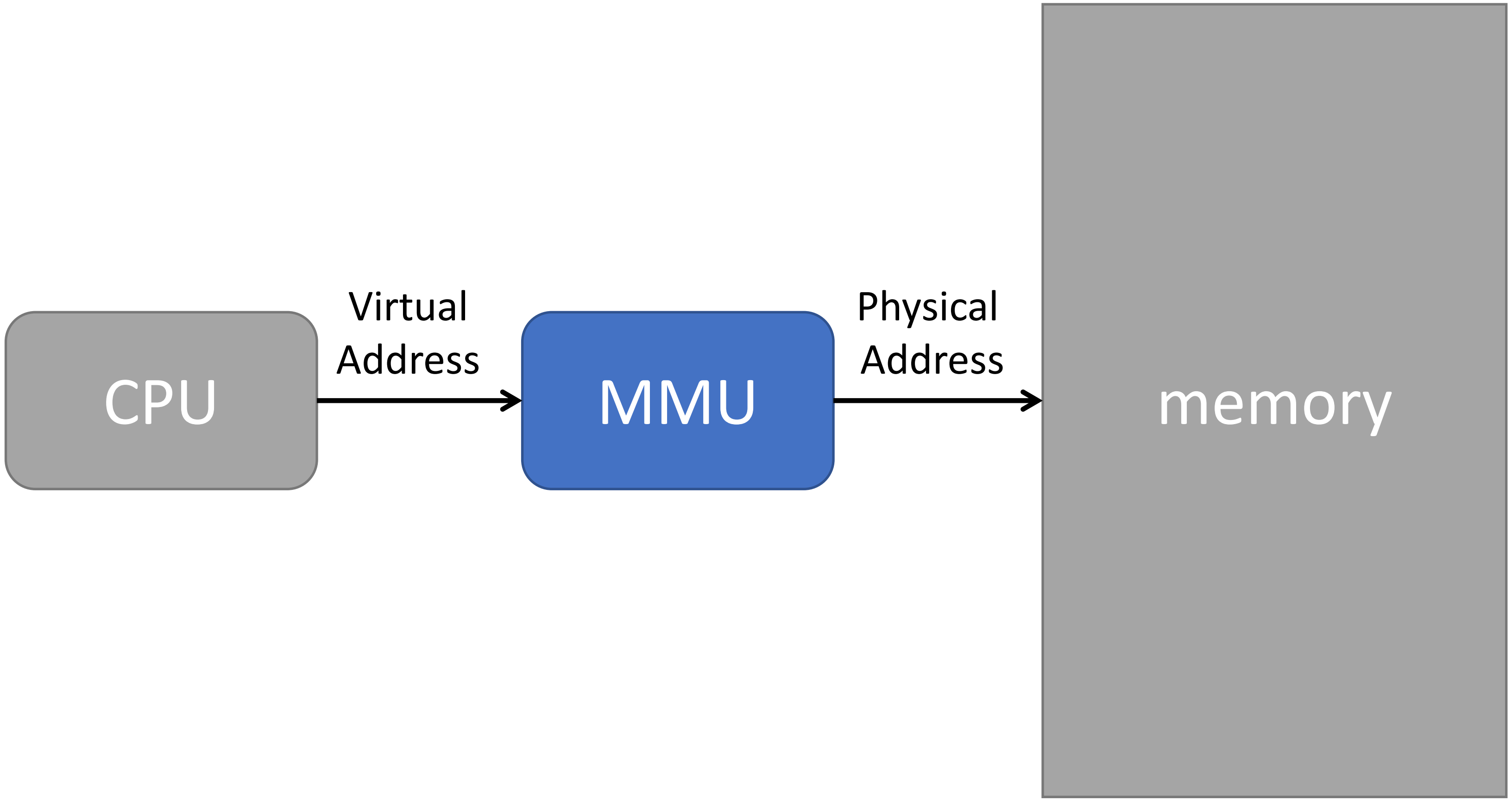
- Operating system gets to implement protection too!

- No modern operating systems work this way.

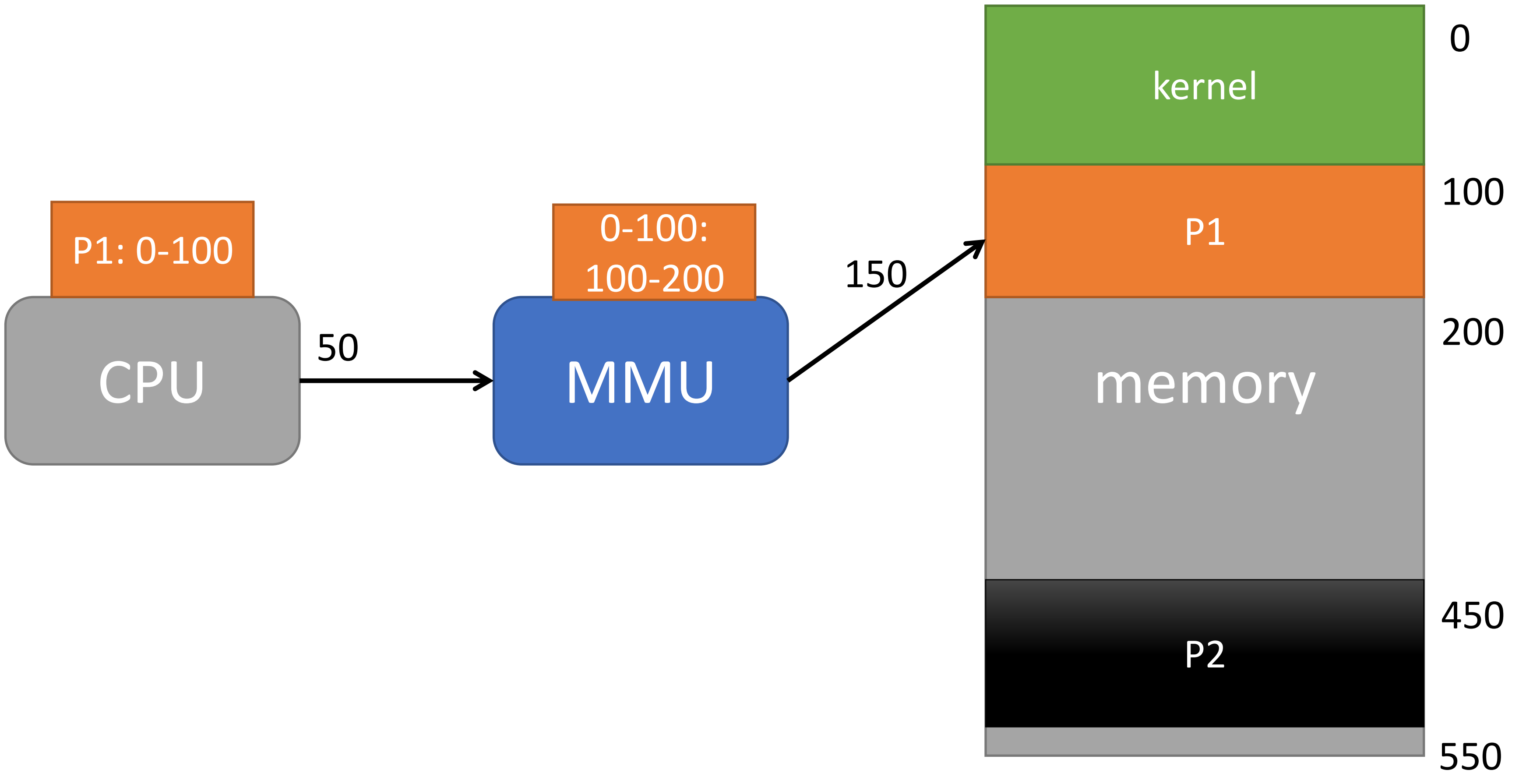
Memory Management Unit (MMU)

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

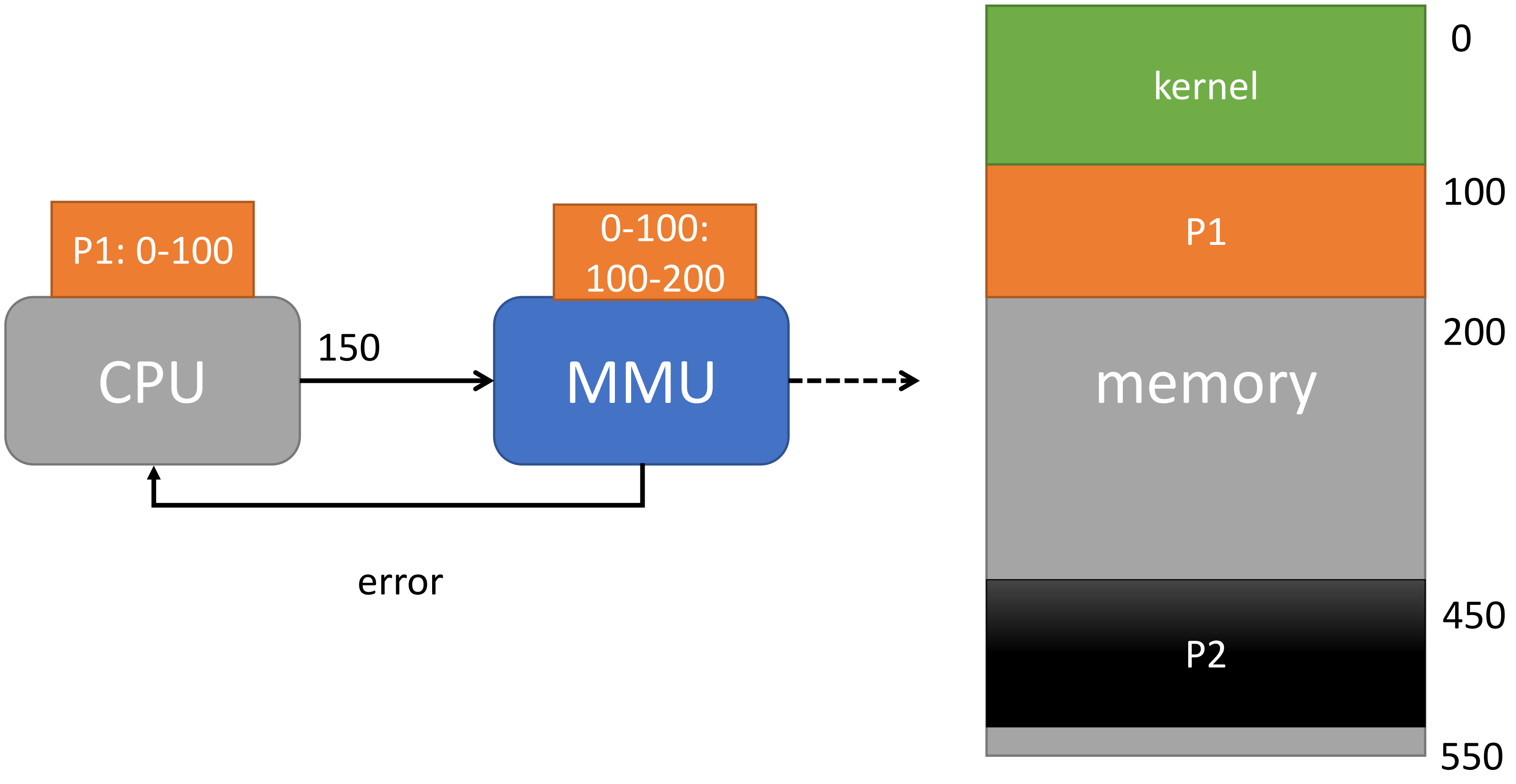
MMU: Virtual to Physical



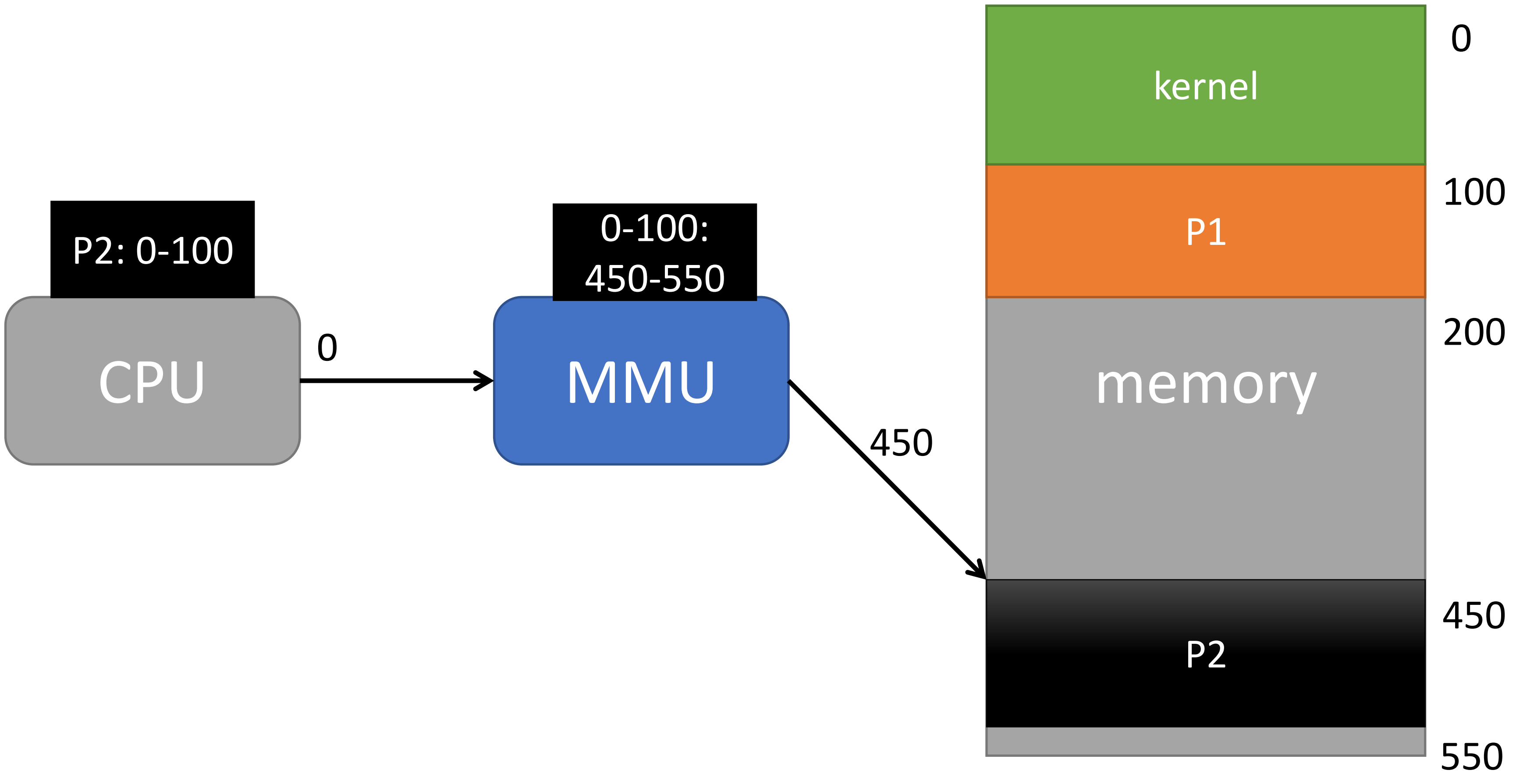
Mapping Example



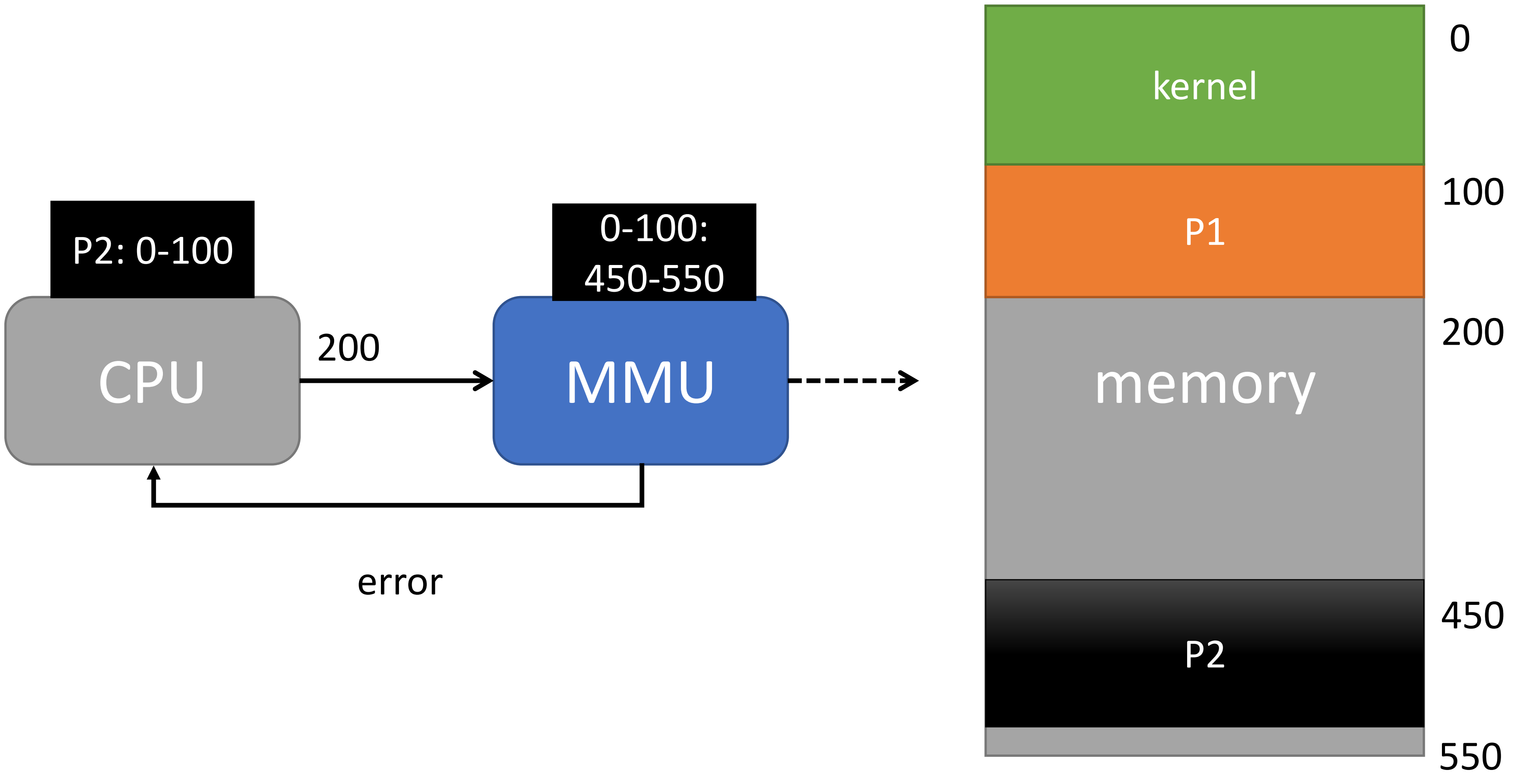
Mapping and Protection Example



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

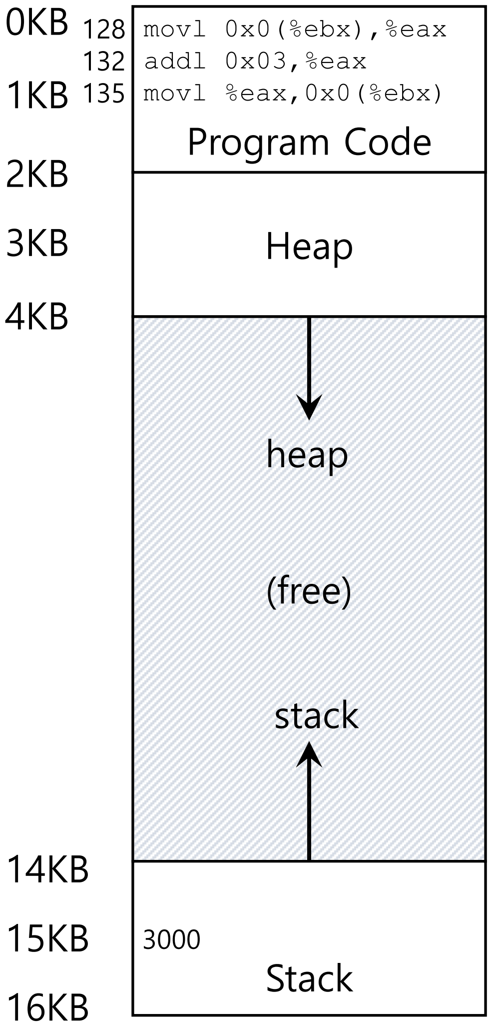
C Code Example

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

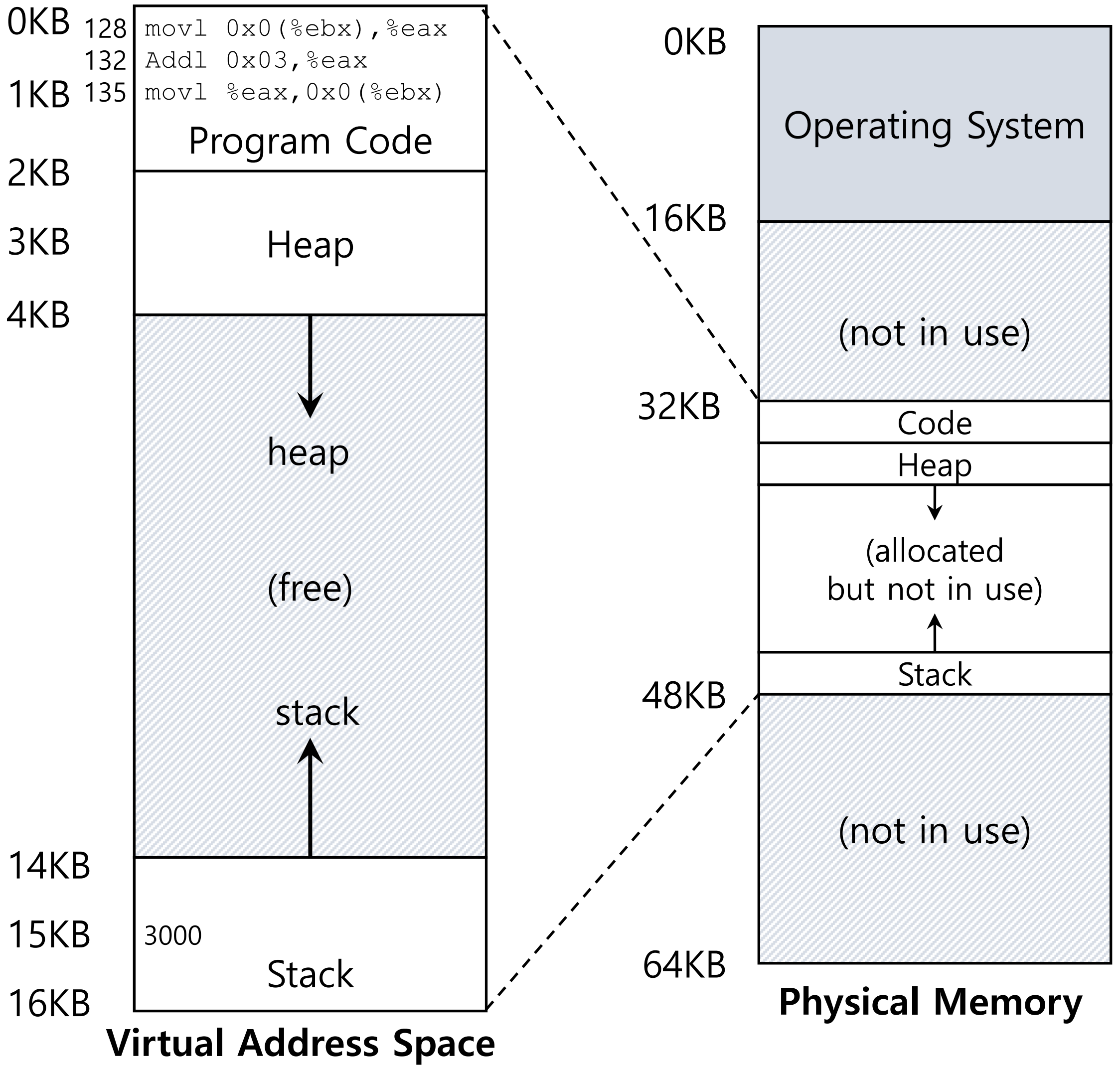


Virtual Address Space

```
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)

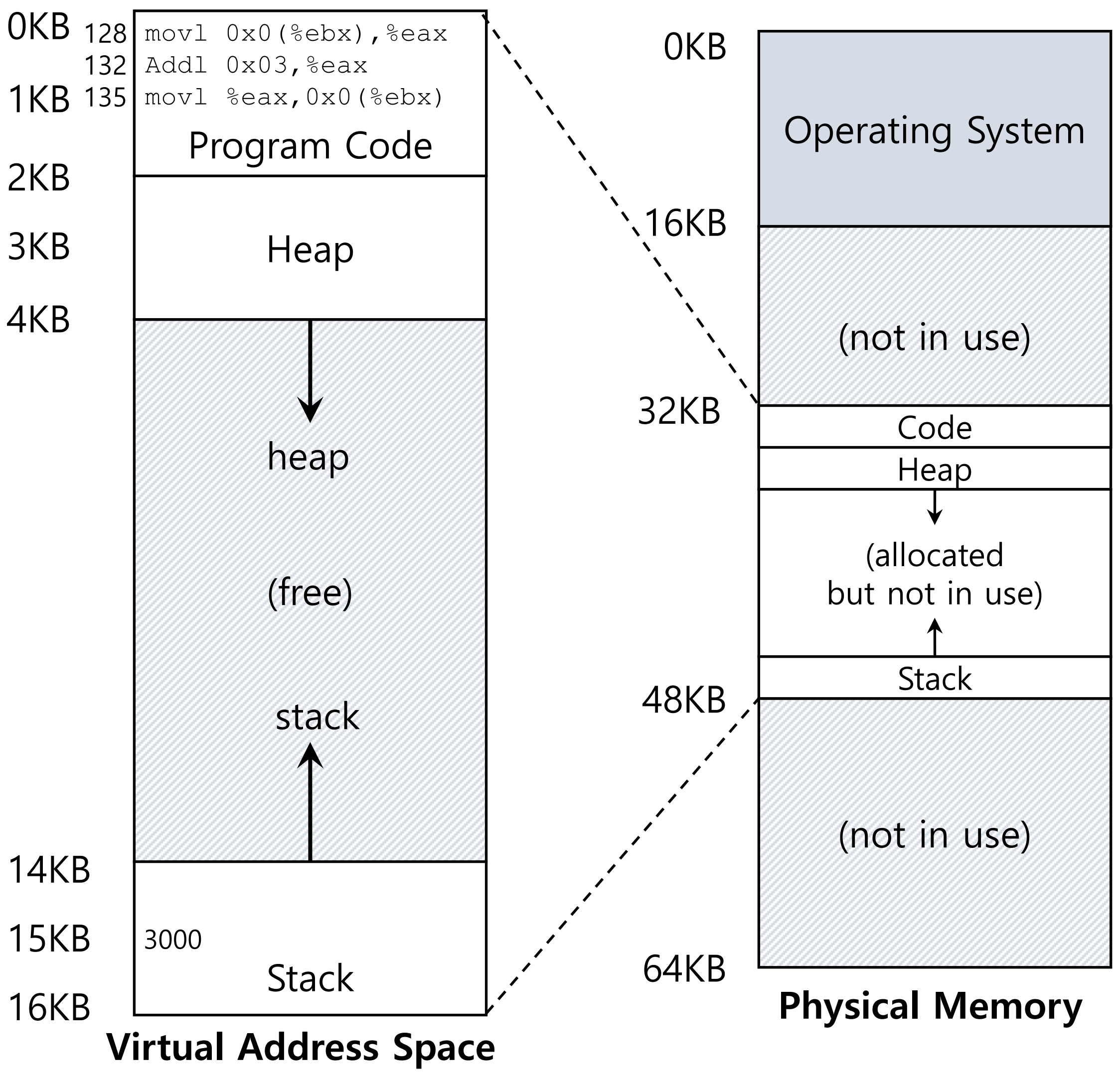
Code in Virtual Memory




```
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

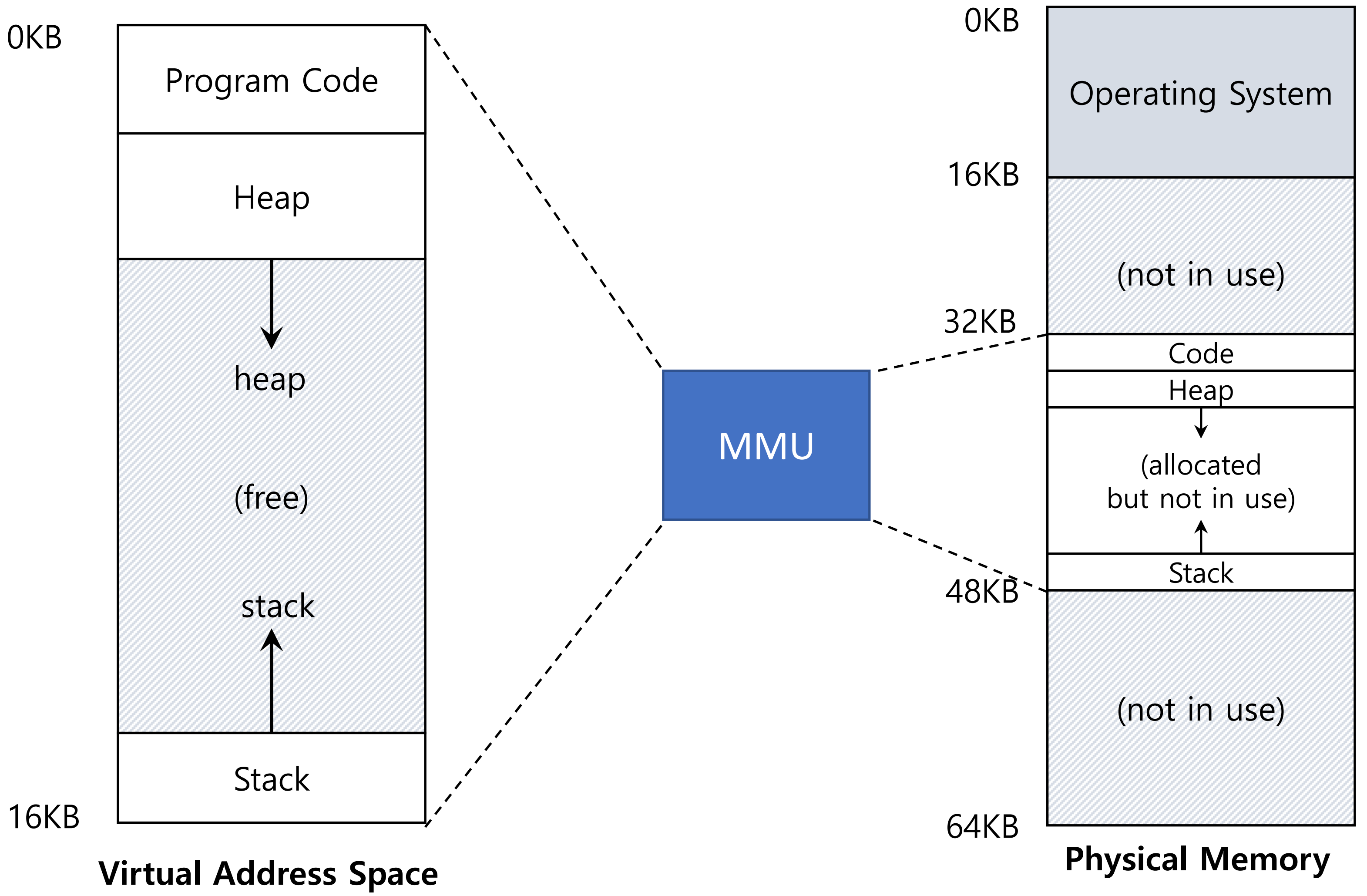


`x = x + 3`

- 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!)
  - Also limited by how MMU interprets bits
    - “Page table” structure is modern limit
    - We will see why on Thursday
  - 2^48 or 2^57 for x86 machines in last 10 years
  - 2^48 or 2^52 for ARM machines

Size of Address Spaces

- **Physical address space size**
  - Limited by **size of memory (RAM)**
  - Nowadays, order of **tens/hundreds of GB**
  - Also limited by outgoing bits of MMU
  - But MMU capability exceeds RAM sizes
    - high-end modern RAM: 39 bits
    - Standard modern MMU: 52 bits

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 – every modern system

For each scheme

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

Base and Bounds

Base and Bounds

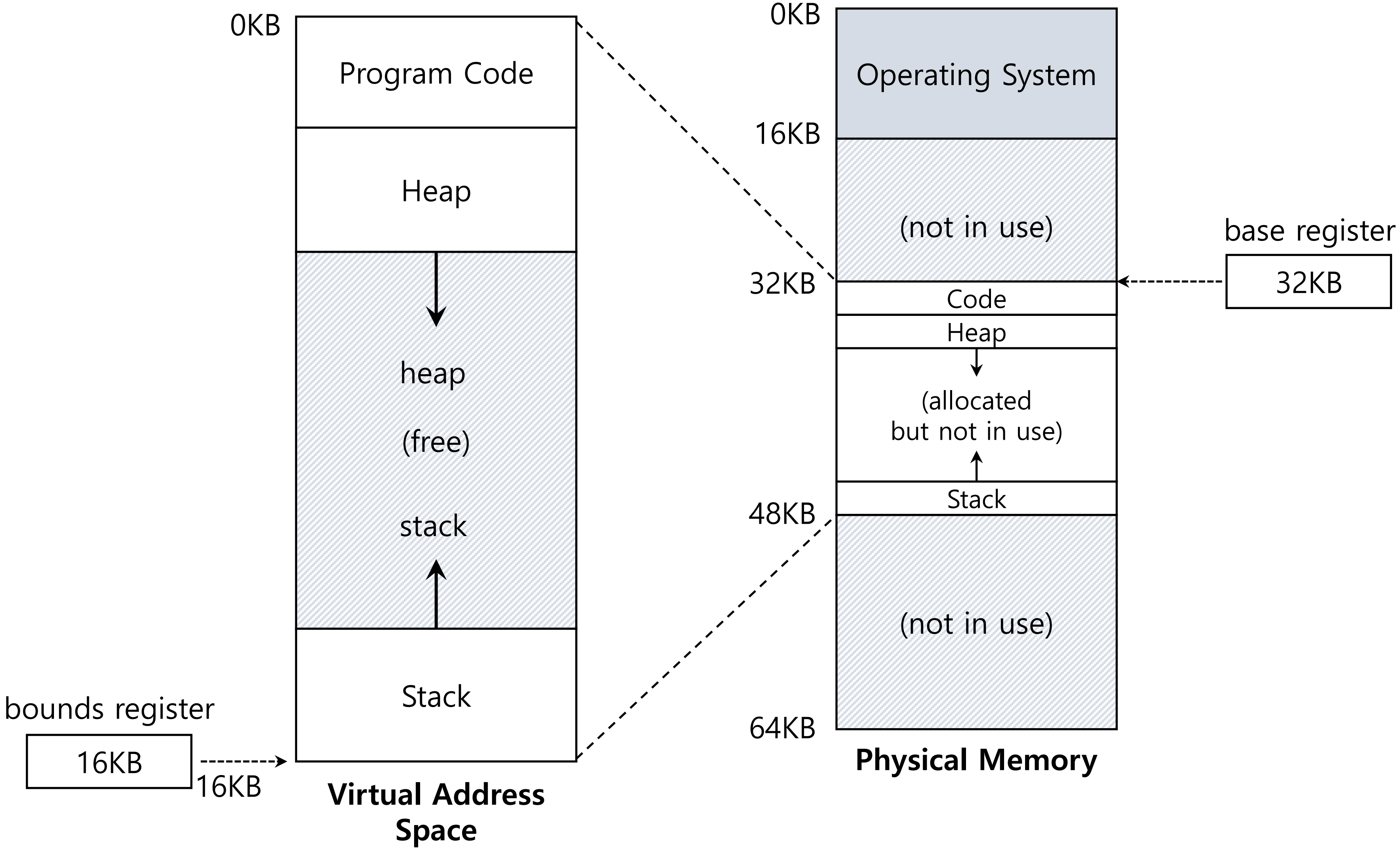
Virtual Address Space

- 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: $\boxed{0 \leq \textit{virtual address} < \boldsymbol{bound}\ (\boldsymbol{in\ limit\ register})}$

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

Base and Bounds: Example

- C - Language code

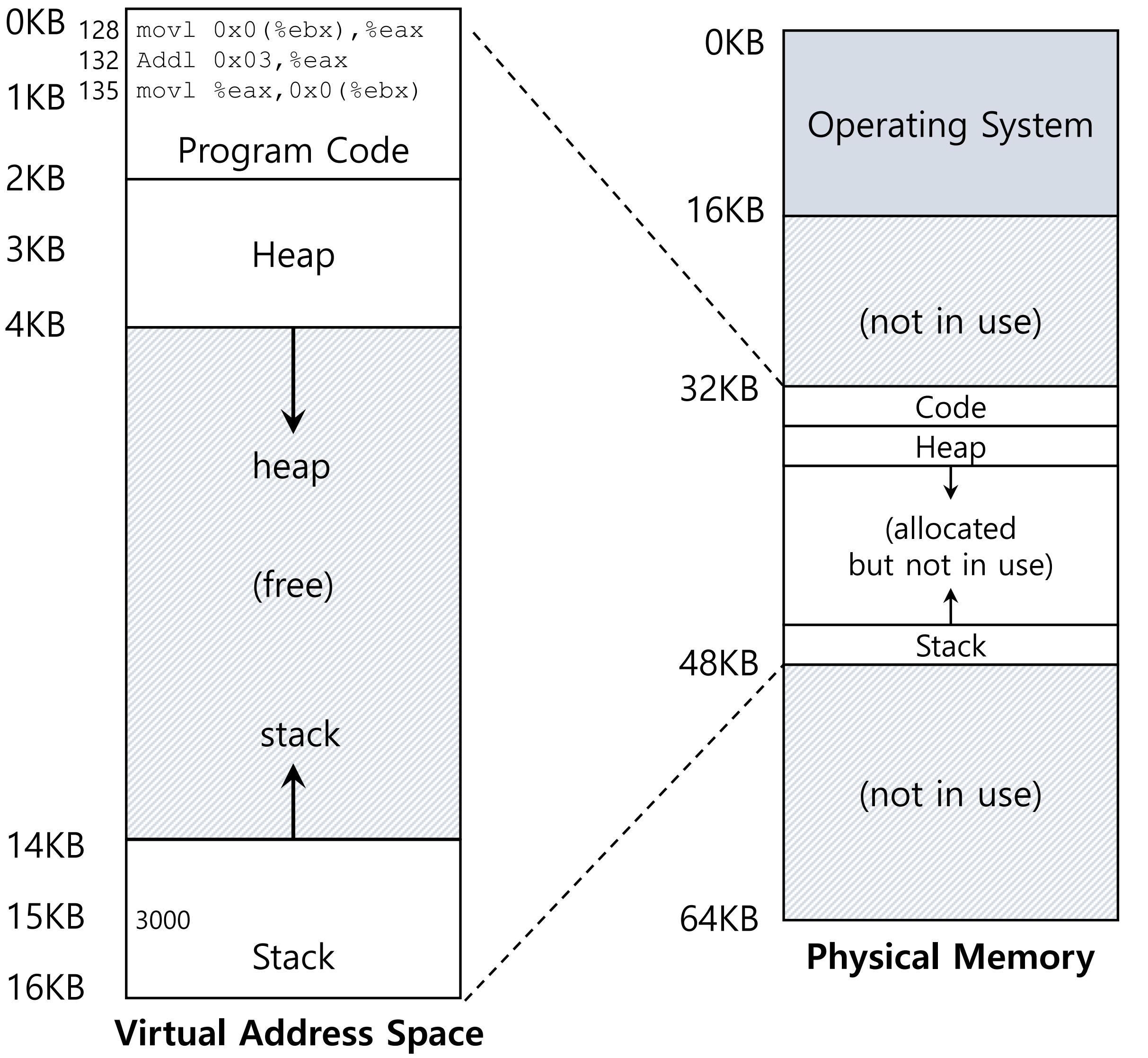
```
void func()
        int x = 3000;
        ...
        x = x + 3; // this is the line of code we are interested in
```

- 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

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

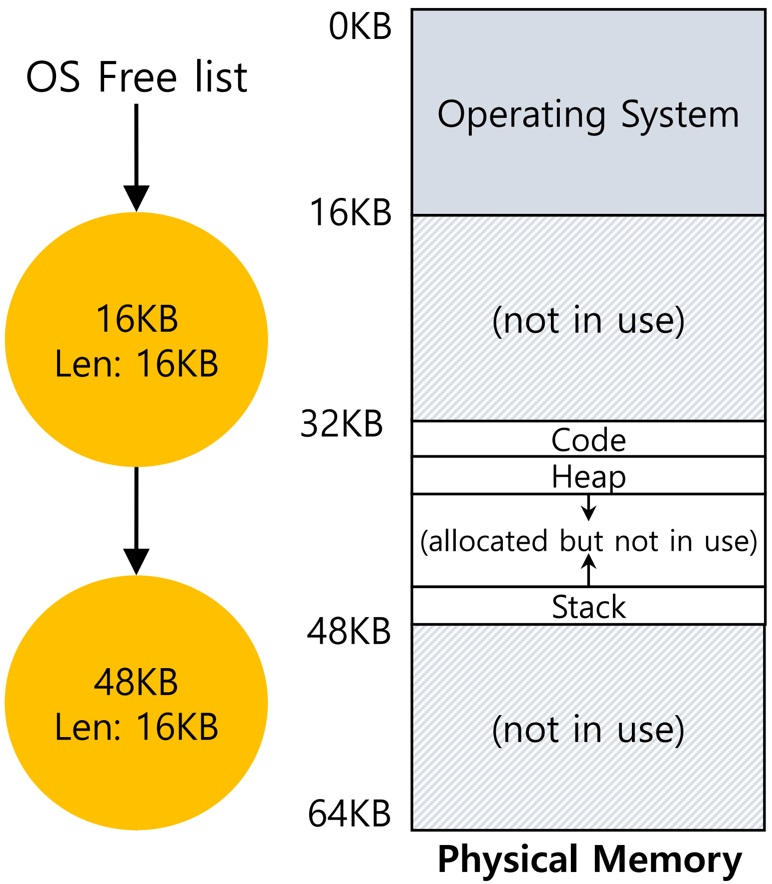
Base and Bounds: Main Memory Allocation

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

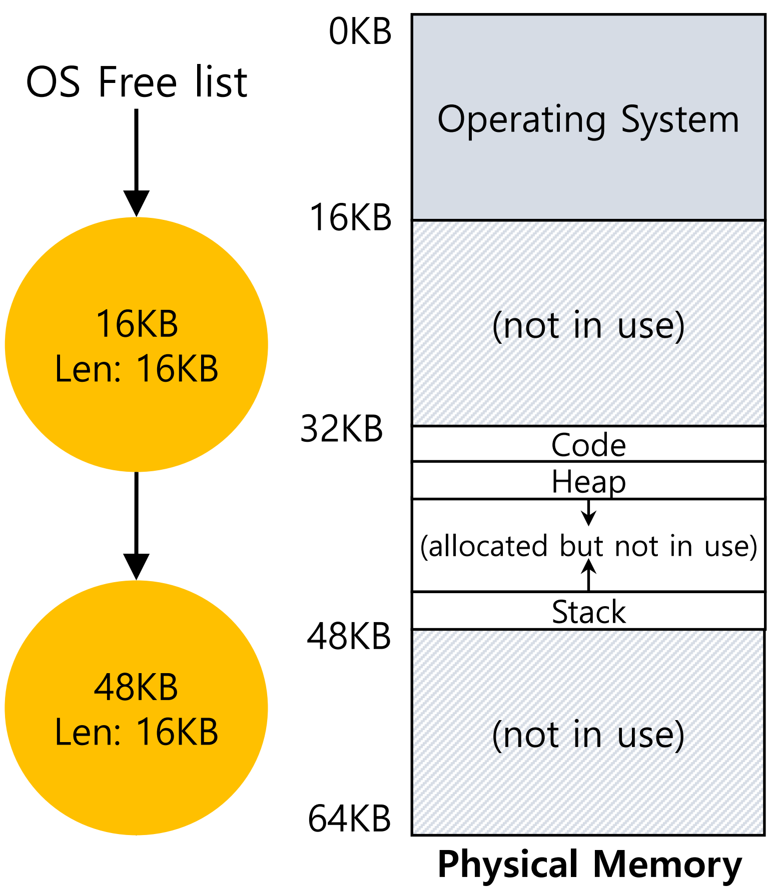
- Take first hole bigger than requested
- Easy to find

Best-fit

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

Worst-fit?!

- Takes largest hole



Base and Bounds: (External) Fragmentation

Small holes become unusable

Part of memory cannot be used

Serious problem ☹

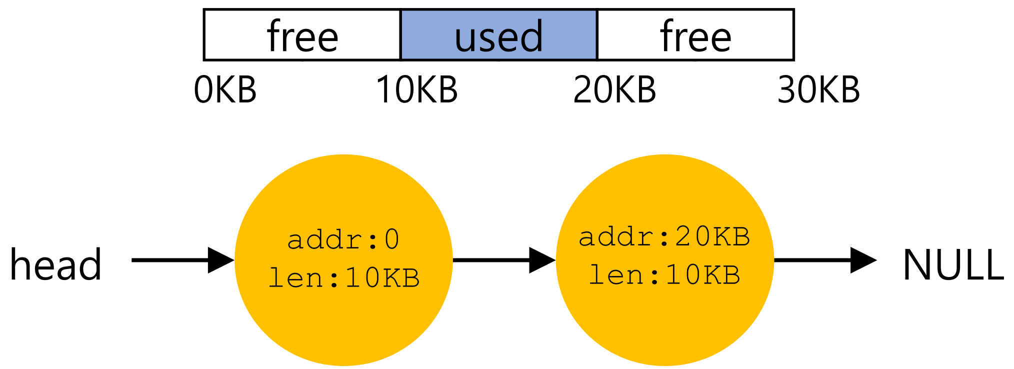
Base and Bounds: (External) Fragmentation

Small holes become unusable

Part of memory cannot be used

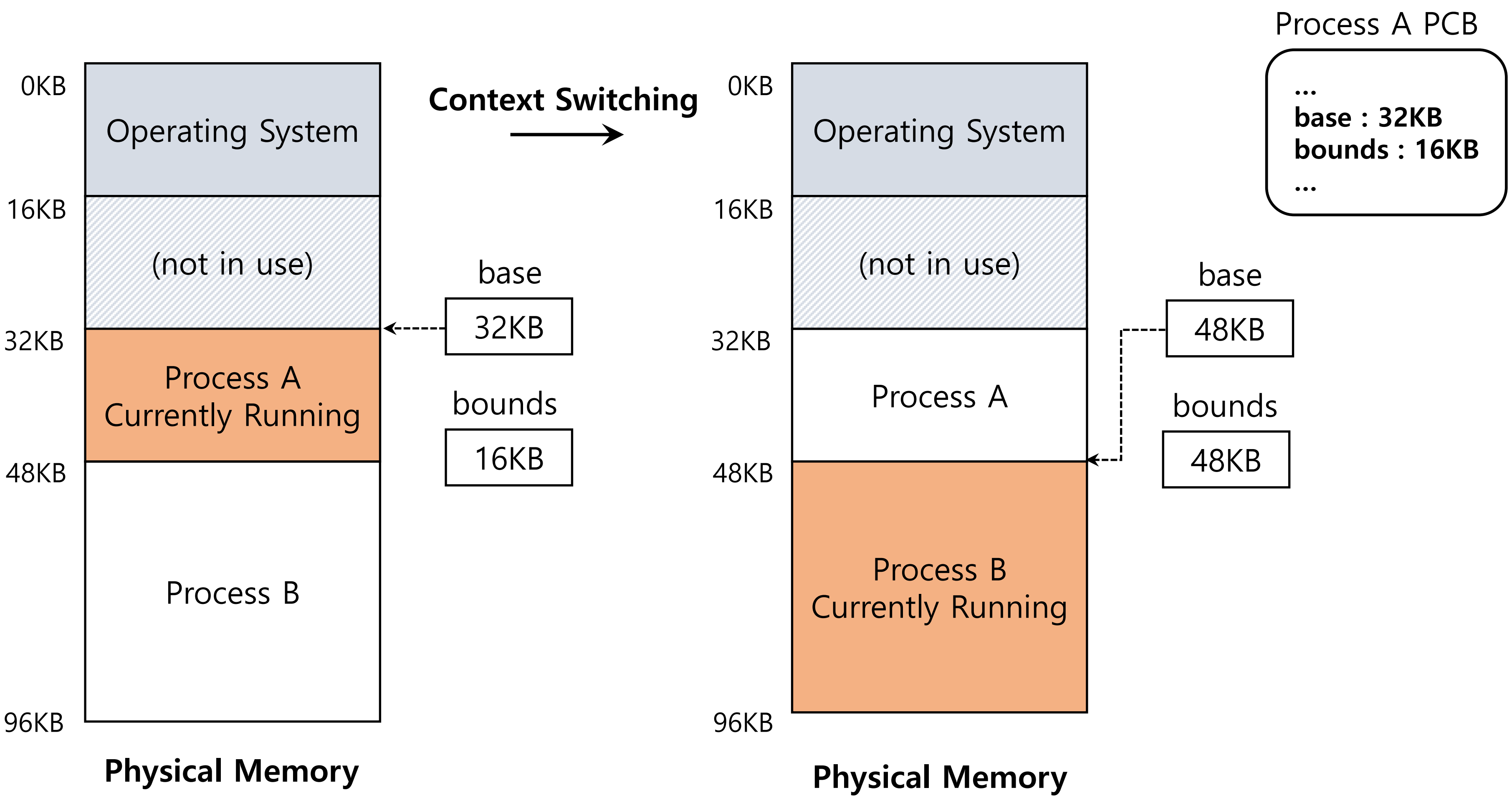
Serious problem ☹

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		

Let’s practice!

Free Space Management (Part 1)

In this exercise we will look at memory allocation/free operations on the heap and **draw the heap and the free list at each step**. The simple memory allocator has 2 operations:

```
P = Alloc(n) // allocates n bytes to pointer P
Free(P) // frees memory that was allocated to pointer P
```

The heap of size is 20 bytes, starting at address 0. The free list is kept ordered by address (increasing). Finally, the allocator has a "best fit” free-list searching policy. The operations are:

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
7. P4 = Alloc(9) ;
8. Free(P3);
9. P5 = Alloc(7);
10. P6= Alloc(1);
```

Free Space Management (Part 1)

`Initialization`

Free List

Addr:0; sz:20

Heap

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
```

Free List

Addr:0; sz:20

Heap

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
```

Free List

Addr:0; sz:20

Heap

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
```

Free List

Addr:6; sz:14

Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
```

Free List

Addr:6; sz:14

Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
```
`2. P1 = Alloc(9);`

Free List

Addr:15; sz:5

Heap

0	**P0**
1	**P0**
2	**P0**
3	**P0**
4	**P0**
5	**P0**
6	**P1**
7	**P1**
8	**P1**
9	**P1**
10	**P1**
11	**P1**
12	**P1**
13	**P1**
14	**P1**
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
```

Free List

Addr:15; sz:5

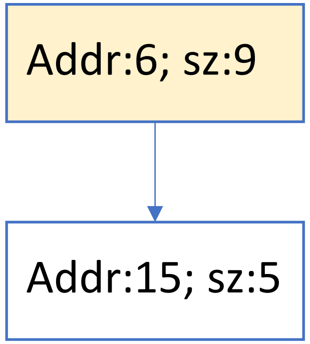
Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	P1
7	P1
8	P1
9	P1
10	P1
11	P1
12	P1
13	P1
14	P1
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
```

Free List



Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
```

Free List

Addr:6; sz:9

Addr:15; sz:5

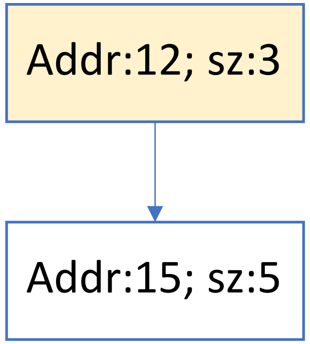
Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
```

Free List



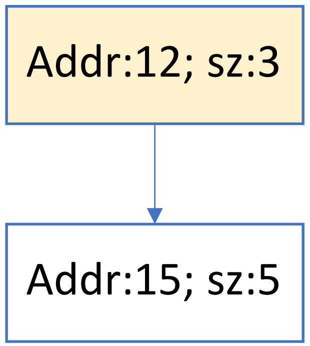
Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	P2
7	P2
8	P2
9	P2
10	P2
11	P2
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
```

Free List



Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	P2
7	P2
8	P2
9	P2
10	P2
11	P2
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
```

Free List

Addr:15; sz:5

Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	P2
7	P2
8	P2
9	P2
10	P2
11	P2
12	P3
13	P3
14	P3
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
```

Free List

Addr:15; sz:5

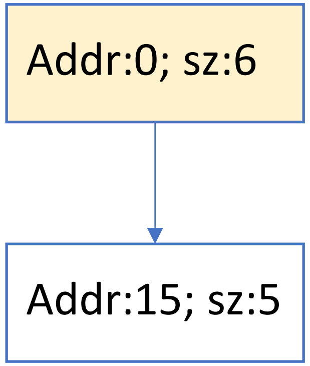
Heap

0	**P0**
1	**P0**
2	**P0**
3	**P0**
4	**P0**
5	**P0**
6	**P2**
7	**P2**
8	**P2**
9	**P2**
10	**P2**
11	**P2**
12	**P3**
13	**P3**
14	**P3**
15	
16	
17	
18	
19	

Free Space Management (Part 1)

1. `P0 = Alloc(6);`
2. `P1 = Alloc(9);`
3. `Free(P1);`
4. `P2 = Alloc(6);`
5. `P3 = Alloc(3);`
6. **`Free(P0);`**

Free List



Heap

0	
1	
2	
3	
4	
5	
6	P2
7	P2
8	P2
9	P2
10	P2
11	P2
12	P3
13	P3
14	P3
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
```
`7. P4 = Alloc(9) ;`

Free List

Addr:0; sz:6

↓

Addr:15; sz:5

Heap

0	
1	
2	
3	
4	
5	
6	P2
7	P2
8	P2
9	P2
10	P2
11	P2
12	P3
13	P3
14	P3
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
7. P4 = Alloc(9) ;
```

Free List

Addr:0; sz:6

↓

Addr:15; sz:5

Alloc failed!

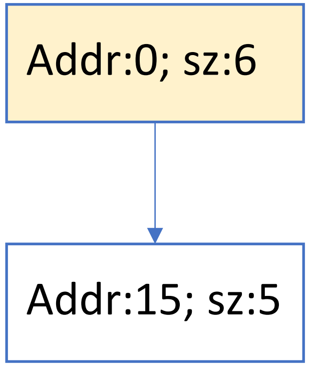
Heap

0	
1	
2	
3	
4	
5	
6	P2
7	P2
8	P2
9	P2
10	P2
11	P2
12	P3
13	P3
14	P3
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
7. P4 = Alloc(9) ;
```
`8. Free(P3);`

Free List



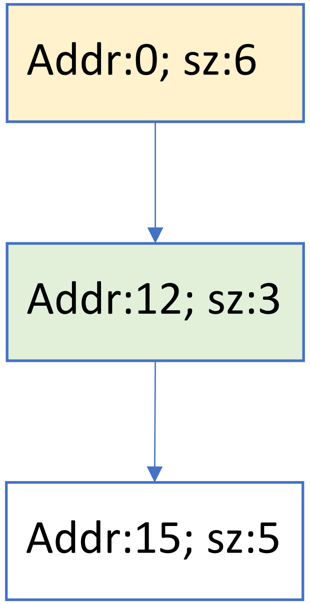
Heap

0	
1	
2	
3	
4	
5	
6	P2
7	P2
8	P2
9	P2
10	P2
11	P2
12	P3
13	P3
14	P3
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
7. P4 = Alloc(9) ;
8. Free(P3);
```

Free List



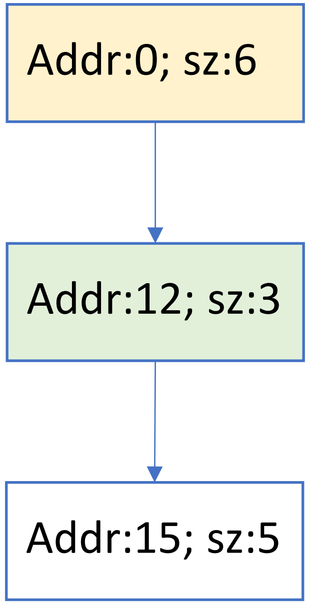
Heap

0	
1	
2	
3	
4	
5	
6	P2
7	P2
8	P2
9	P2
10	P2
11	P2
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2.  P1 = Alloc(9);
3.  Free(P1);
4.  P2 = Alloc(6);
5.  P3 = Alloc(3);
6.  Free(P0);
7.  P4 = Alloc(9) ;
8.  Free(P3);
9.  P5 = Alloc(7);
```

Free List



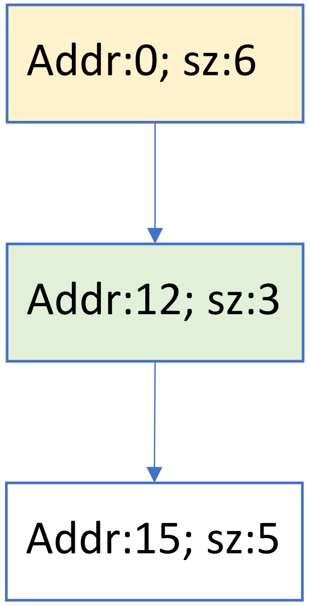
Heap

0	
1	
2	
3	
4	
5	
6	P2
7	P2
8	P2
9	P2
10	P2
11	P2
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2.  P1 = Alloc(9);
3.  Free(P1);
4.  P2 = Alloc(6);
5.  P3 = Alloc(3);
6.  Free(P0);
7.  P4 = Alloc(9) ;
8.  Free(P3);
9.  P5 = Alloc(7);
```

Free List



Alloc failed!

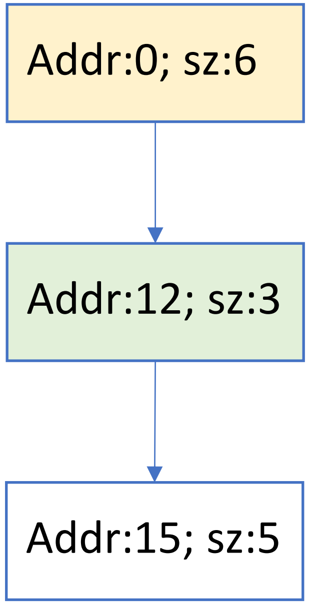
Heap

0	
1	
2	
3	
4	
5	
6	P2
7	P2
8	P2
9	P2
10	P2
11	P2
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
7. P4 = Alloc(9) ;
8. Free(P3);
9. P5 = Alloc(7);
10. P6= Alloc(1);
```

Free List



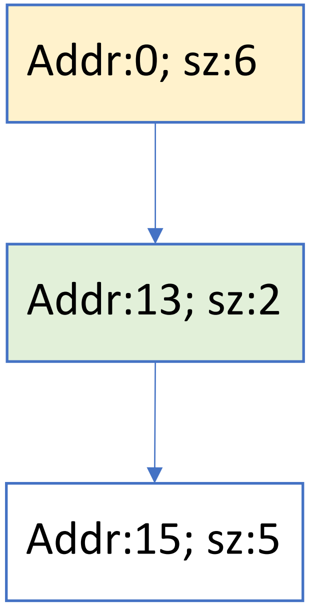
Heap

0	
1	
2	
3	
4	
5	
6	P2
7	P2
8	P2
9	P2
10	P2
11	P2
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
7. P4 = Alloc(9) ;
8. Free(P3);
9. P5 = Alloc(7);
10. P6= Alloc(1);
```

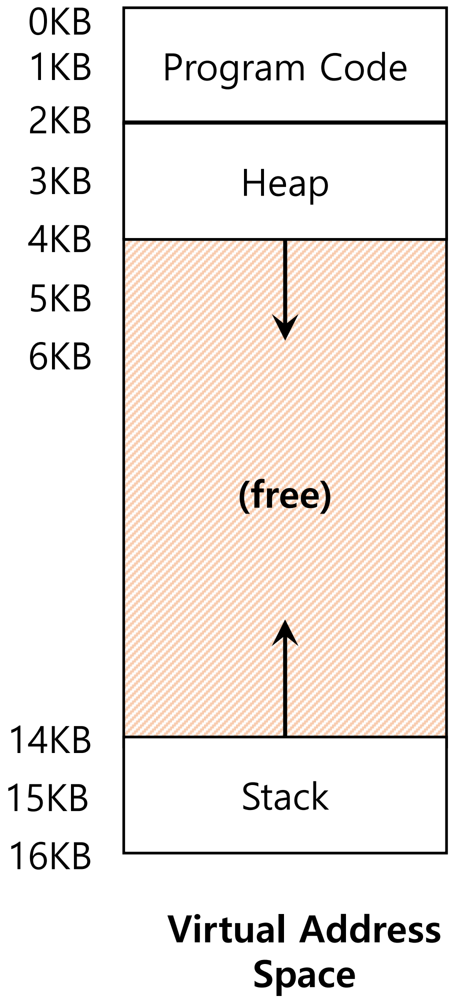
Free List



Heap

0	
1	
2	
3	
4	
5	
6	P2
7	P2
8	P2
9	P2
10	P2
11	P2
12	P6
13	
14	
15	
16	
17	
18	
19	

Base and Bounds: (Internal) Fragmentation



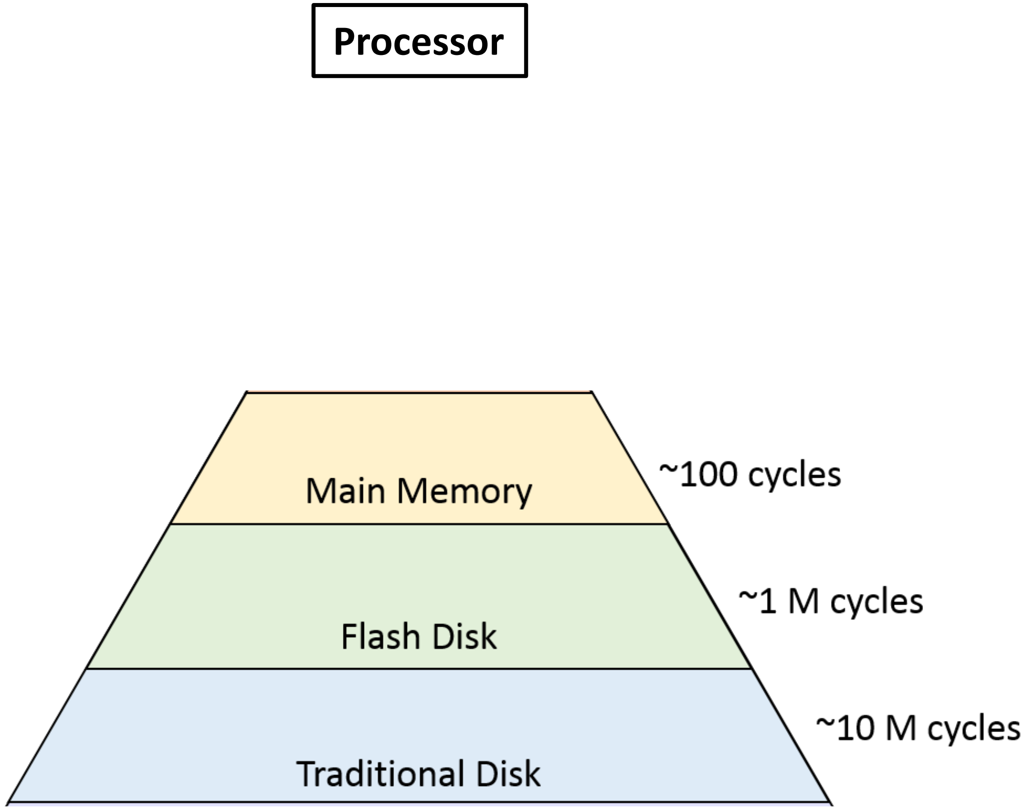
- **Big chunk of “free”** space
- “free” space **takes up** physical memory.
- Inefficient
- (Internal) memory fragmentation

Week 6

Memory Management: Virtual Memory

Max Kopinsky
13 February 2025

Recap: OS Memory Management

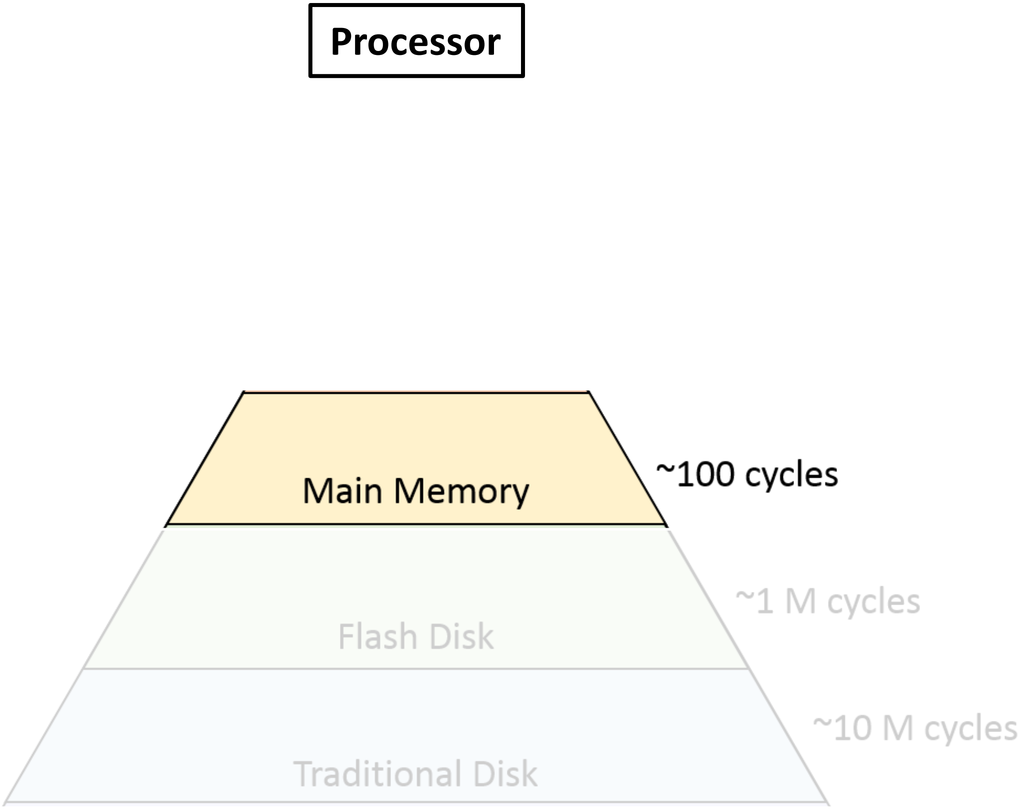


Recap: Simplifying Assumption

So for today:

All of a program must be in main memory

Not concerned with disk

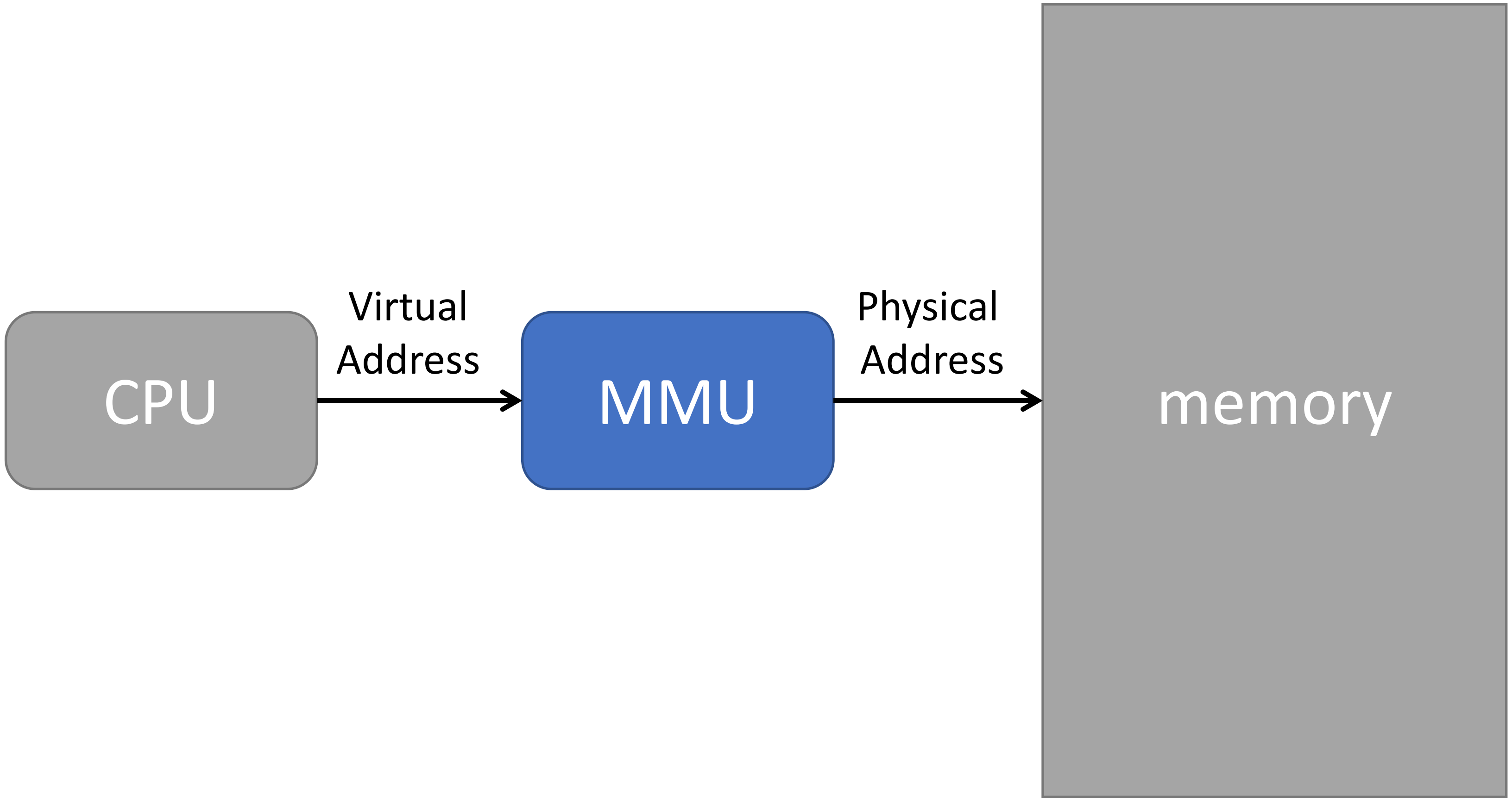


Recap: Virtual vs. Physical address space

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

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

Recap:
MMU: Virtual to Physical



Recap: Base and Bounds

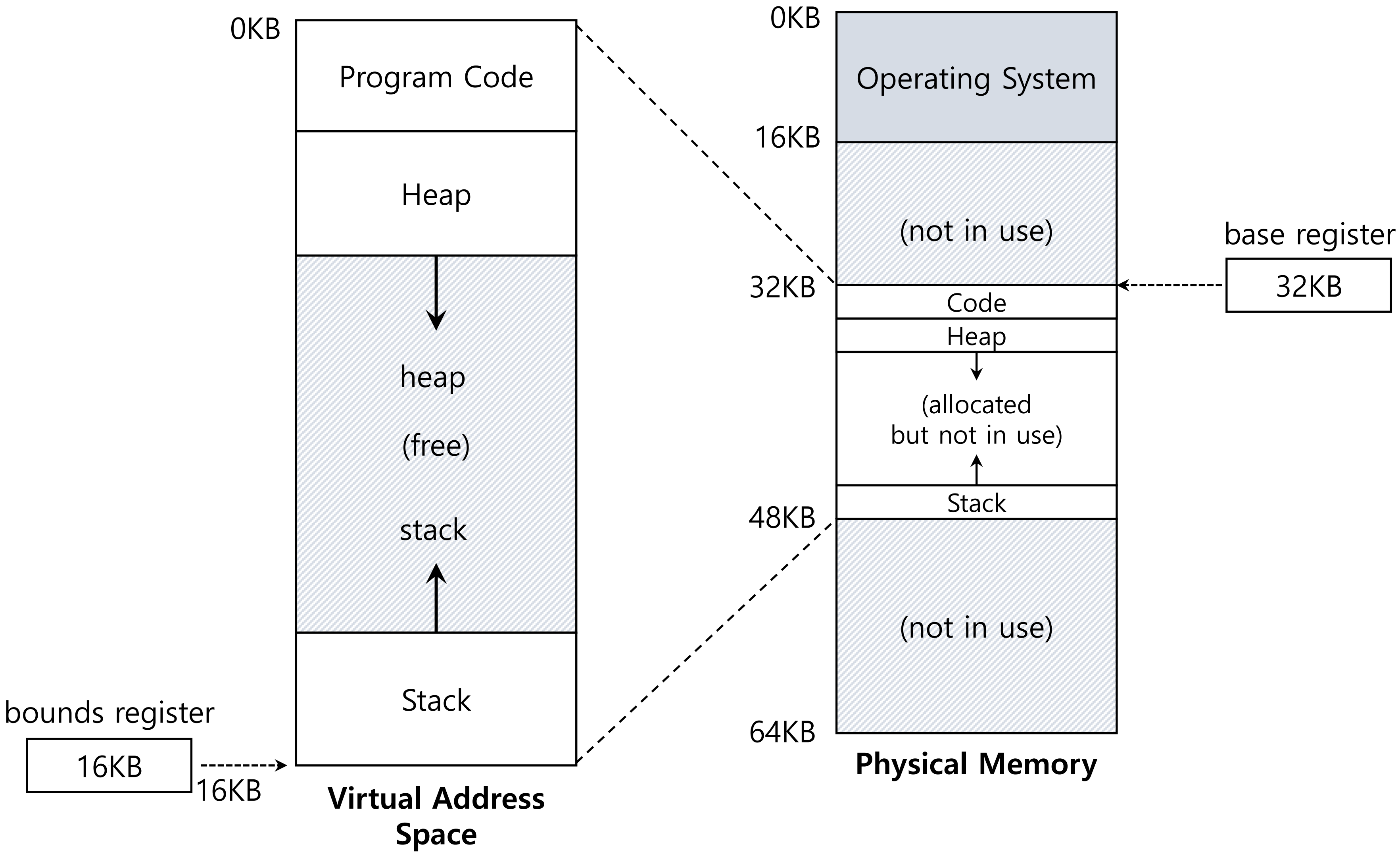
Virtual Address Space

- Linear address space : from 0 to MAX

Physical Address Space

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

Recap: Base and Bounds



Recap: 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: $0 \leq \ virtual\ address < \boldsymbol{bound}\ (\boldsymbol{in\ limit\ register})$

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

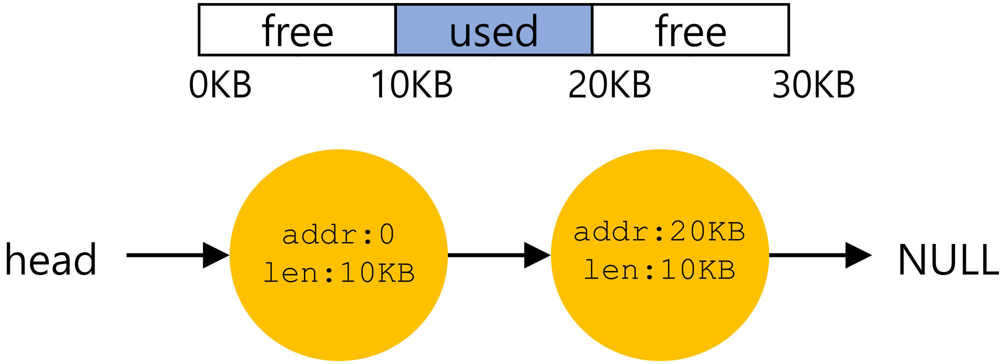
Recap: Base and Bounds (External) Fragmentation

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.

Different Virtual to Physical Mapping Schemes

- Base and bounds
- Segmentation
- (Simplified) Paging

Segmentation

Segmentation

Virtual Address Space

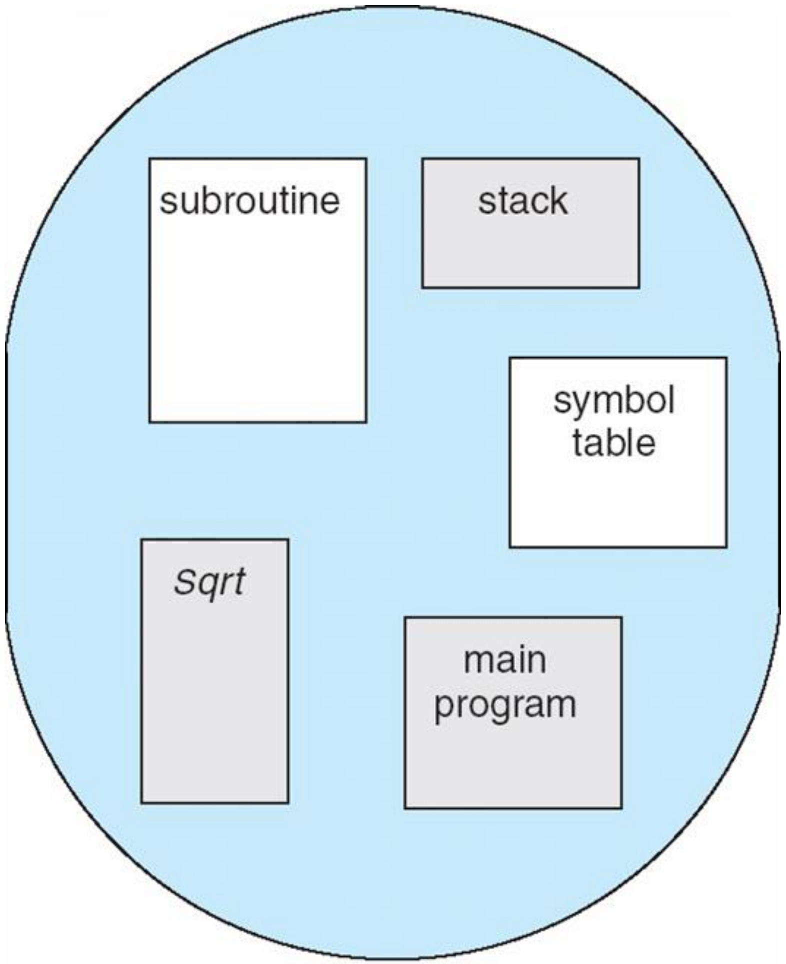
- Two-dimensional
- Set of segments 0 .. n
- Each segment i is linear from 0 to MAX_i

Physical Address Space

- Set of segments, each linear

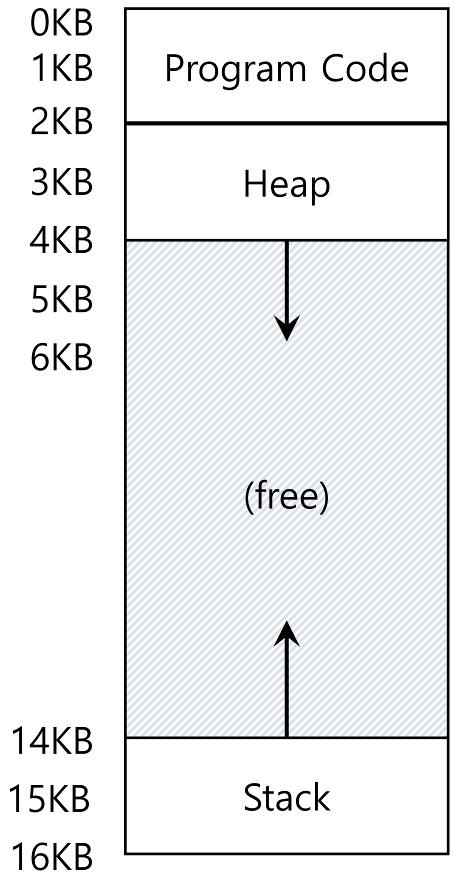
What is a Segment?

- Anything you want it to be
- Typical examples:
  - **Code**
  - **Heap**
  - **Stack**

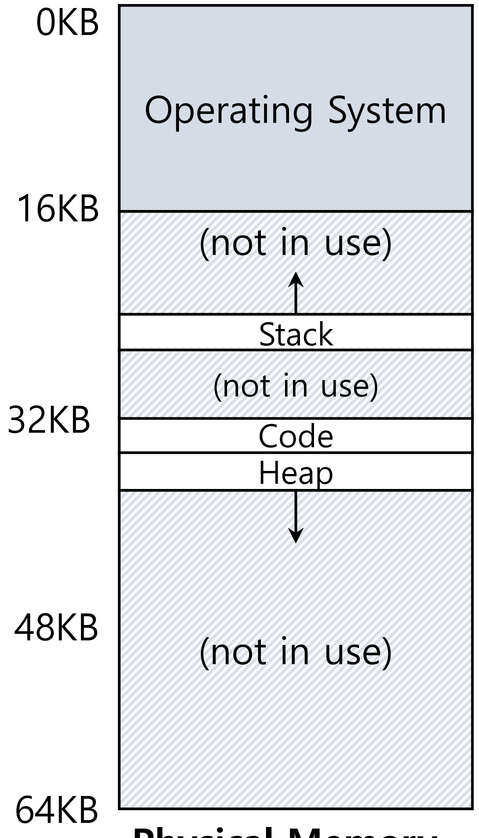


Segments

Segmentation Example



Virtual Address Space



Physical Memory

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

Segmentation: Virtual Address

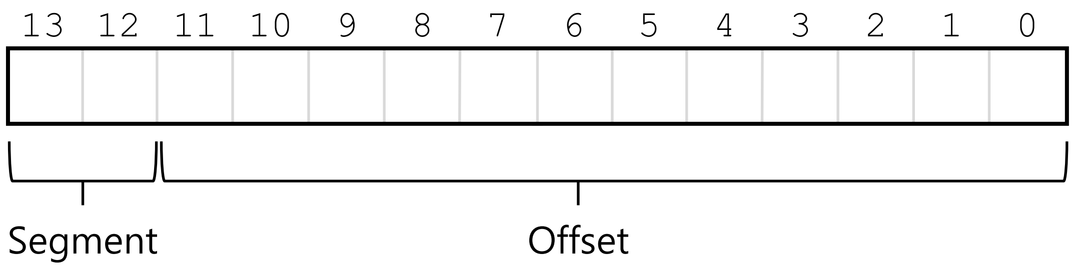
Two-dimensional address:

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

It is like multiple base-and-bounds

Segmentation Virtual Address example

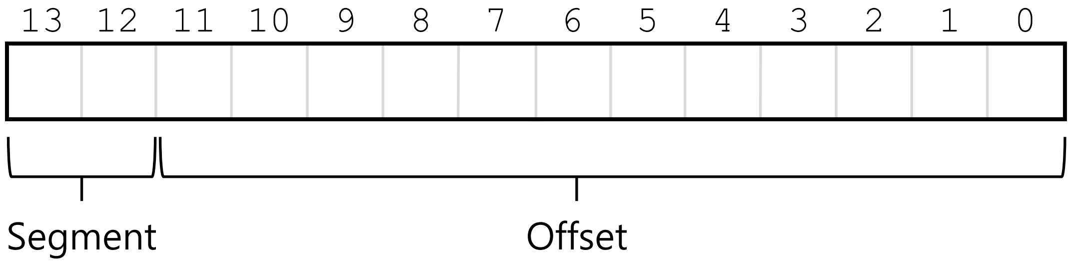
Chop up the address space into segments based on the **top few bits** of virtual address.



How many segments? What is the size of each segment?

Segmentation Virtual Address example

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

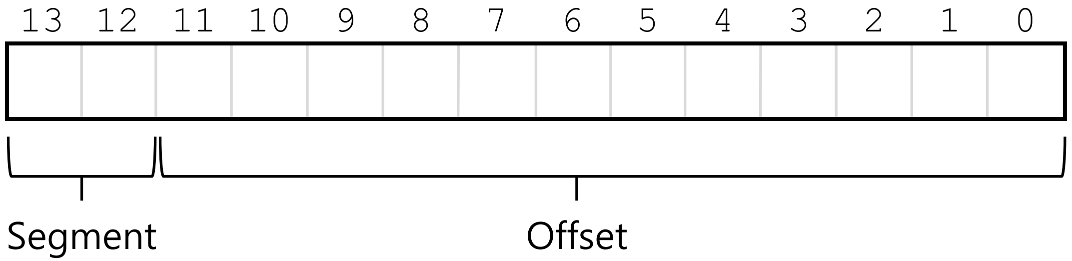


How many segments?
2^2 = 4 segments

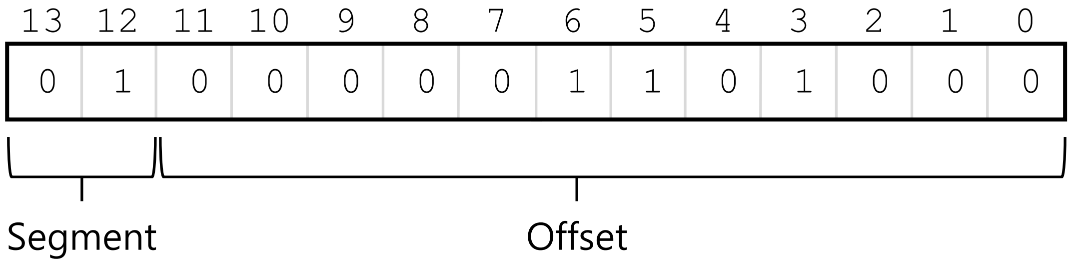
What is the size of each segment?
2^12 = 4KB

Segmentation Virtual Address example

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



Example: virtual address 4200_{10} (01000001101000_2): which segment?



Segment	bits
Code	00
Heap	01
Stack	10
-	11

MMU for Segmentation

Segment table

Indexed by segment number

Contains **(base, limit) pair**

- Base: physical address of segment in memory
- Limit: length of segment

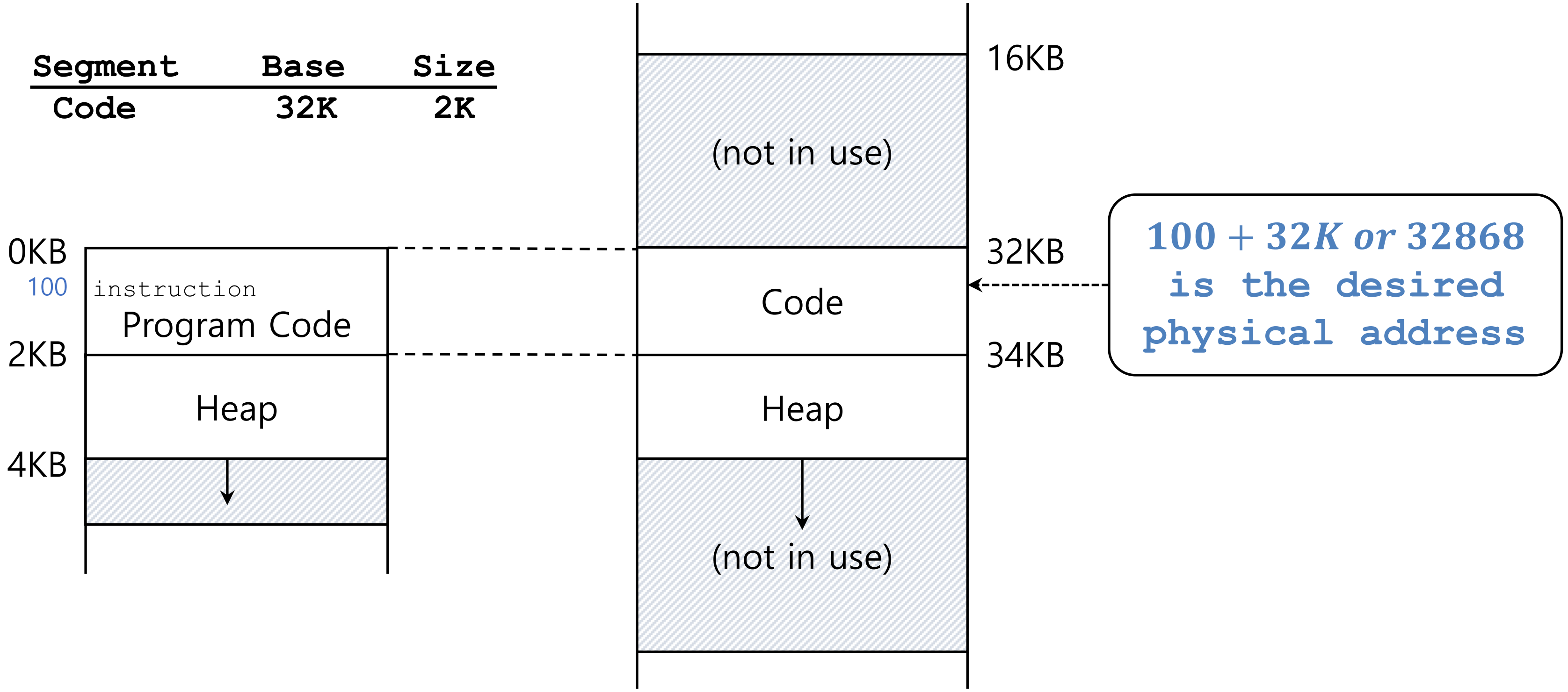
If segment table is in memory, rather than MMU hardware:

- *Need pointer to table in memory*
- *Need length of table*

Segmentation: Address Translation Example

$$physical\ address = offset\ in\ segment + segment\ base$$

Segment	Base	Size
Code	32K	2K

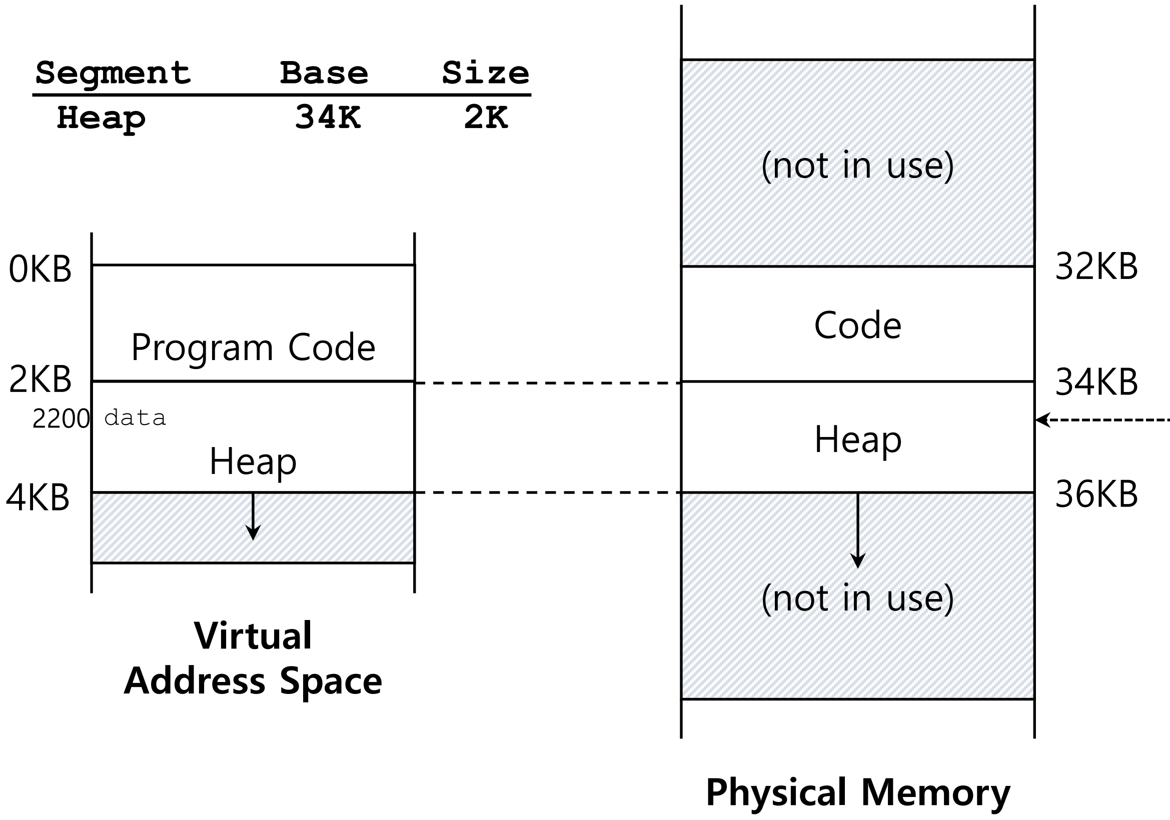


Segmentation: Address Translation (Cont.)

$$physical\ address = offset\ in\ segment + segment\ base$$

Segment	Base	Size
Heap	34K	2K

152 = 2200 – 2KB



$152 + 34K$ or 34968 is the desired physical address

Let’s practice!

Segmentation

Given a CPU with 16 bytes of byte-addressable physical memory, with the following 2 segments in main memory:

SEG 0 (base address: 7_{10}, bound: 3_{10})

SEG 1 (base address: 3_{10}, bound: 2_{10})

Compute the virtual to physical address translations (or Segmentation Faults), for the following virtual addresses:

4_{10}, 5_{10}, 6_{10}, 0_{10}, 3_{10}.

(Not enough info just yet!)

Segmentation

CPU with 3-bit instructions, 16 bytes physical memory
SEG 0 (base address: 7_{10}*, bound: 3_{10})
SEG 1 (base address: 3_{10}, bound: 2_{10})

Preliminaries:

1. How many bits does a virtual address have? What is the size of the virtual address space?

2. How many bits does a physical address have? What is the size of the physical address space?

3. What is the structure of the virtual address?
   - How many bits for the segment?
   - How many bits for the offset?
   - What is the maximum size of each segment?

Segmentation

CPU with 3-bit instructions, 16 bytes physical memory
SEG 0 (base address: 7_{10}*, bound: 3_{10})
SEG 1 (base address: 3_{10}, bound: 2_{10})

Preliminaries:

1. How many bits does a virtual address have? What is the size of the virtual address space?

   → **3 bits, with a virtual address space of 8 addresses**

2. How many bits does a physical address have? What is the size of the physical address space?

   → **4 bits, with a phys address space of 16 addresses, because 16 =2^4**

3. What is the structure of the virtual address?
   - How many bits for the segment? → **1 bit**
   - How many bits for the offset? → **2 bits**
   - What is the maximum size of each segment? → **4 Bytes (each segment has max length 4 and each address denotes 1 byte of physical memory)**

Segmentation

CPU with 3-bit instructions, 16 bytes physical memory
SEG 0 (base address: 7_{10}*, bound: 3_{10})
SEG 1 (base address: 3_{10}, bound: 2_{10})

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

Segmentation

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Physical Address Space

- SEG 0 (base address: 7_{10}*, bound: 3_{10})
- SEG 1 (base address: 3_{10}, bound: 2_{10})

Segmentation

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

- SEG 0 (base address: 7_{10}*, bound: 3_{10})
- SEG 1 (base address: 3_{10}, bound: 2_{10})

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Physical Address Space

Segmentation

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

- SEG 0 (base address: 7_{10}*, bound: 3_{10})
- SEG 1 (base address: 3_{10}, bound: 2_{10})

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	**SEG 0**
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Physical Address Space

Segmentation

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	**SEG 1**
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	**SEG 0**
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Physical Address Space

- SEG 0 (base address: 7_{10}*, bound: 3_{10})
- SEG 1 (base address: 3_{10}, bound: 2_{10})

Segmentation

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

4

0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	**SEG 1**
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	**SEG 0**
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Physical Address Space

- SEG 0 (base address: 7_{10}*, bound: 3_{10})
- SEG 1 (base address: 3_{10}, bound: 2_{10})

Segmentation

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

4

VA	**1**	**0**	**0**

Virt addr 4_{10}
Is valid in SEG 1
Phys addr 3_{10}

0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	**SEG 1**
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	**SEG 0**
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Physical Address Space

- SEG 0 (base address: 7_{10}*, bound: 3_{10})
- SEG 1 (base address: 3_{10}, bound: 2_{10})

Segmentation

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

5

0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	**SEG 1**
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	**SEG 0**
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Physical Address Space

- SEG 0 (base address: 7_{10}*, bound: 3_{10})
- SEG 1 (base address: 3_{10}, bound: 2_{10})

Segmentation

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

5

VA	**1**	**0**	**1**

Virt addr 5_{10}
Is valid in SEG 1
Phys addr 4_{10}

0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	**SEG 1**
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	**SEG 0**
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Physical Address Space

- SEG 0 (base address: 7_{10}*, bound: 3_{10})
- SEG 1 (base address: 3_{10}, bound: 2_{10})

Segmentation

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

6

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	**SEG 1**
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	**SEG 0**
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Physical Address Space

- SEG 0 (base address: 7_{10}*, bound: 3_{10})
- SEG 1 (base address: 3_{10}, bound: 2_{10})

Segmentation

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

6

VA			
VA	**1**	**1**	**0**

Virt addr 6_{10}
Is out of bounds

0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	**SEG 1**
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	**SEG 0**
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Physical Address Space

- SEG 0 (base address: 7_{10}*, bound: 3_{10})
- SEG 1 (base address: 3_{10}, **bound: 2_{10})**

Segmentation Fault!

Offset is not smaller than Bound register for SEG 1.

Segmentation

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

0

0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	**SEG 1**
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	**SEG 0**
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Physical Address Space

- SEG 0 (base address: 7_{10}*, bound: 3_{10})
- SEG 1 (base address: 3_{10}, bound: 2_{10})

Segmentation

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

0

VA **0** **0** **0**

Virt addr 0_{10}
Is valid in SEG 0
Phys addr 7_{10}

0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	**SEG 1**
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	**SEG 0**
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Physical Address Space

- SEG 0 (base address: 7_{10}*, bound: 3_{10})
- SEG 1 (base address: 3_{10}, bound: 2_{10})

Segmentation

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

3

0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	**SEG 1**
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	**SEG 0**
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Physical Address Space

- SEG 0 (base address: 7_{10}*, bound: 3_{10})
- SEG 1 (base address: 3_{10}, bound: 2_{10})

Segmentation

Virtual Addresses (decimal): 4, 5, 6, 0, 3.

	s	offset	
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Virtual Address Space

3

VA	**0**	**1**	**1**

Virt addr 3_{10}
Is out of bounds

0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	**SEG 1**
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	**SEG 0**
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Physical Address Space

- SEG 0 (base address: 7_{10}*, **bound: 3_{10})**
- SEG 1 (base address: 3_{10}, bound: 2_{10})

Segmentation Fault!

Offset is not smaller than Bound register for SEG 0.

Sharing Memory between Processes

Why would we want to do that?

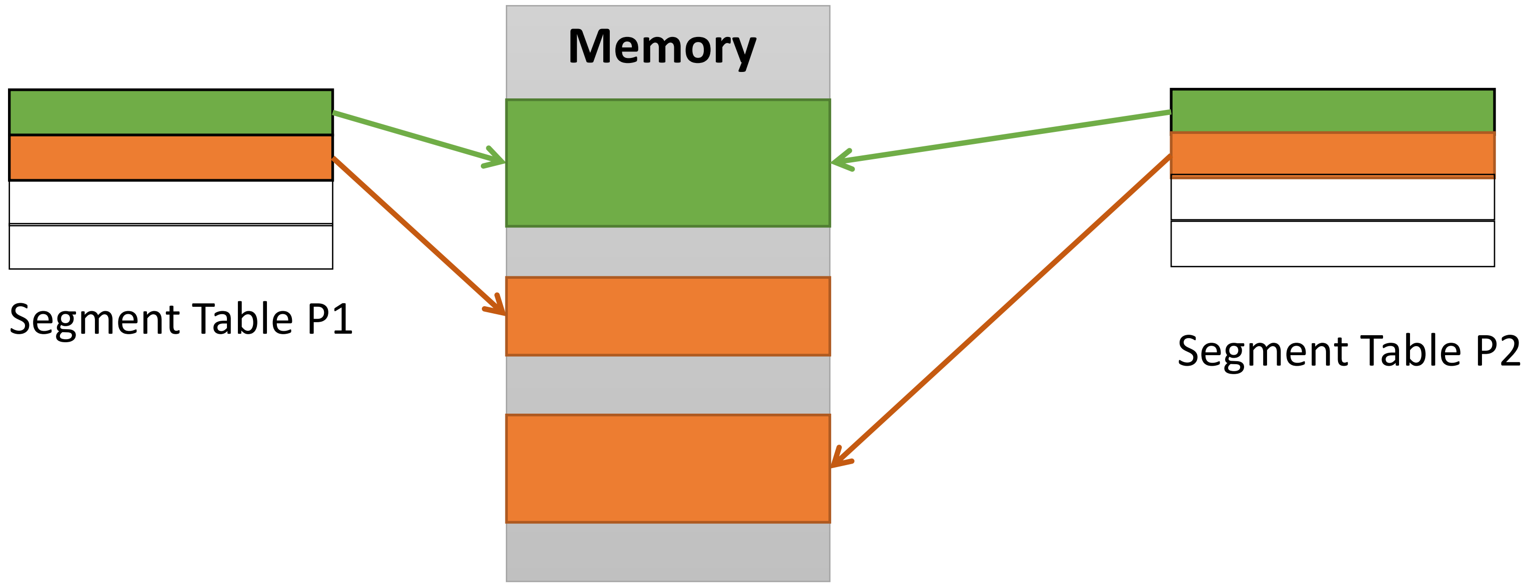
For instance,

- Run twice the same program in different processes
- May want to share code
- Read twice the same file in different processes
- May want to share memory corresponding to file

Sharing not possible with base and bounds, but is possible with segmentation

Segmentation Provides Easy Support for Sharing

- Create segment for shared data
- Add segment entry in segment table of both processes
- Points to shared segment in memory



Segmentation Provides Easy Support for Sharing

Extra **hardware** support is need for form of **Protection bits.**

- **A few more bits** per segment to indicate **permissions** of **read,** write and **execute**.

Example Segment Register Values(with Protection)

Segment	Base	Size	Protection
Code	32K	2K	Read-Execute
Heap	34K	2K	Read-Write
Stack	28K	2K	Read-Write

Main memory allocation with Segmentation

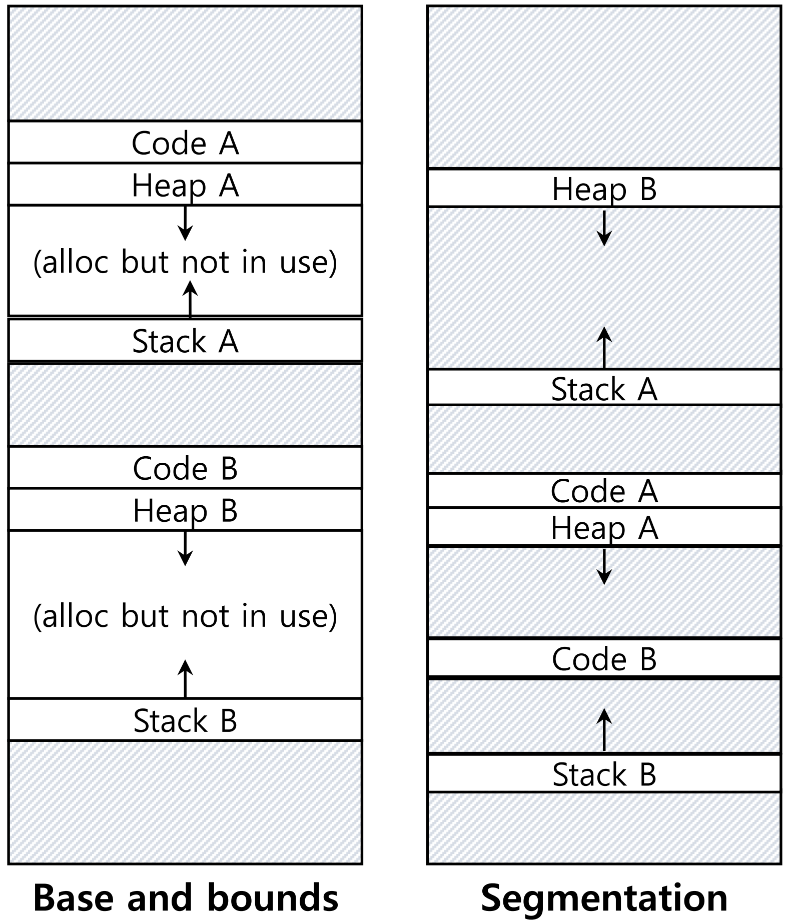
Remember:

Segmentation ~= multiple base-and-bounds

No internal fragmentation inside each segment.

External fragmentation problem is similar.

- Pieces are typically smaller



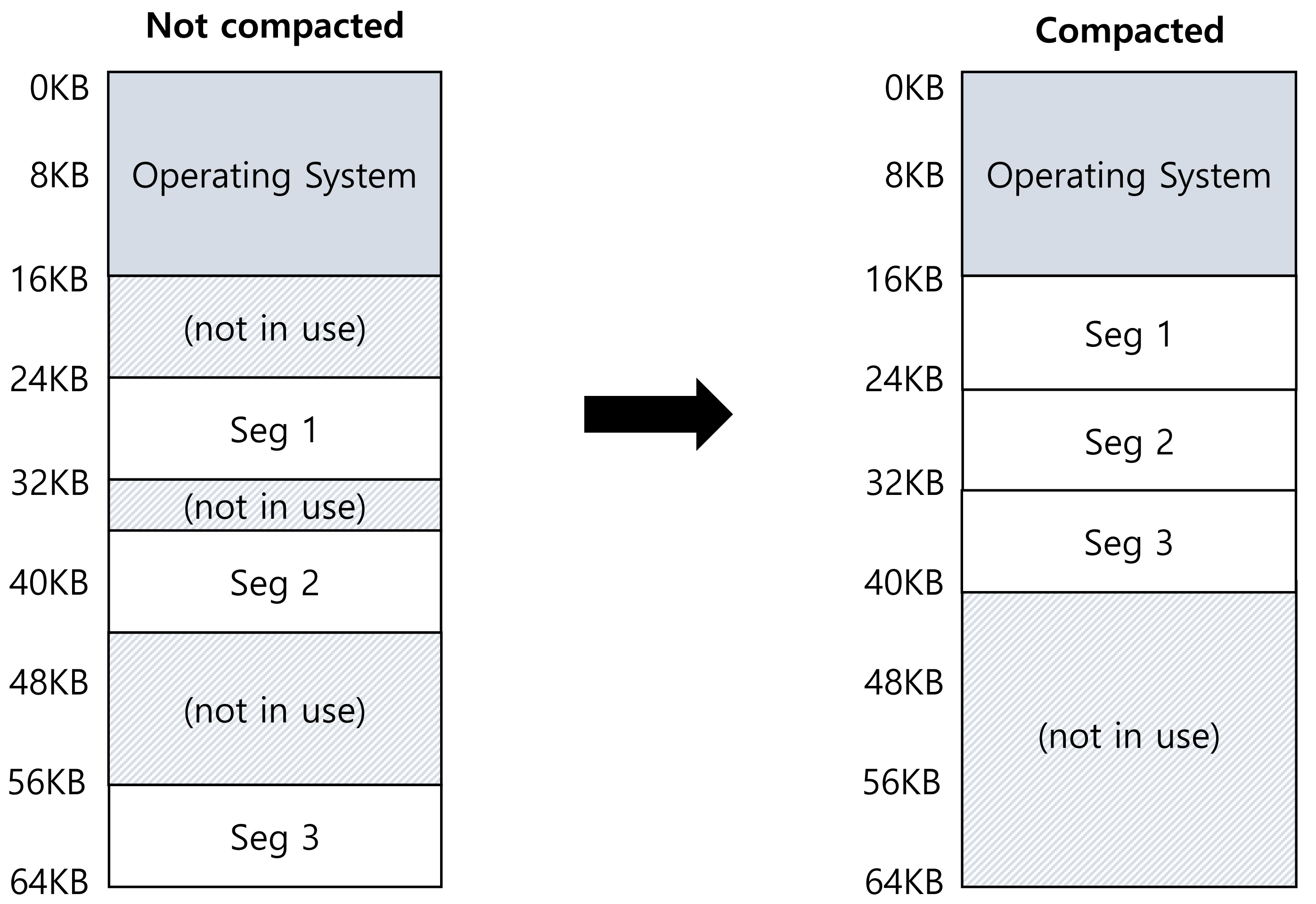
Main memory allocation with Segmentation

Compaction:

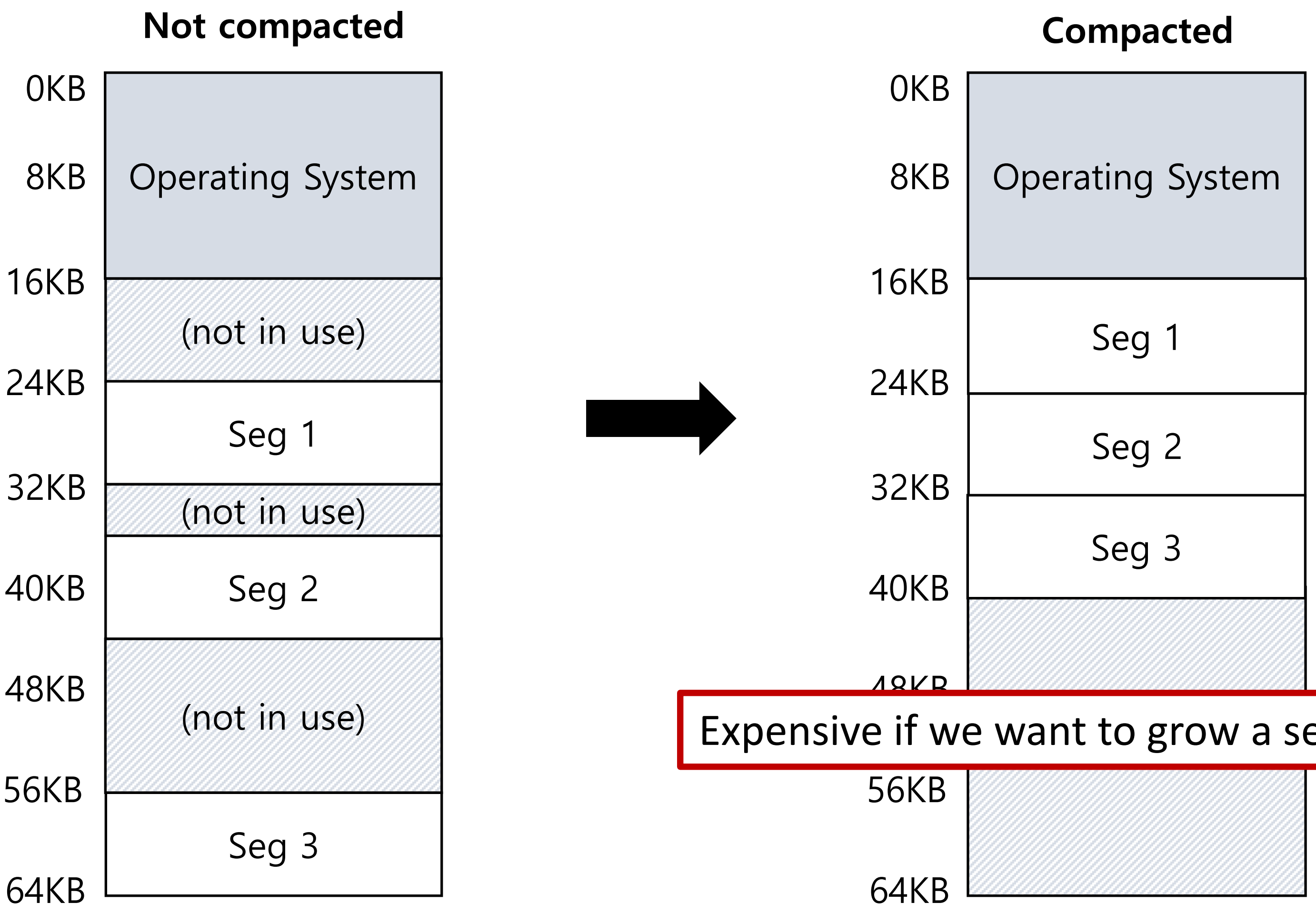
Rearrange segments in physical memory to get rid of “holes”.

- **Stop** running process. Inefficient! ☹
- **Copy** data to somewhere. Expensive! ☹☹
- **Change** segment register value.

Memory Compaction Example



Memory Compaction Example



Expensive if we want to grow a segment after compaction.

More practice!

Free Space Management (Part 2)

Consider the same memory allocator as in Part 1, but this time **the allocator is 4-byte aligned.** This means that each allocated space rounds up to the nearest 4-byte free chunk in size. Draw the heap and the free list for each step.

Same as in part 1, the operations are:

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
7. P4 = Alloc(9) ;
8. Free(P3);
9. P5 = Alloc(7);
10. P6= Alloc(1);
```

Reminder:

- `P = Alloc(n)` // allocates n bytes to pointer P
- `Free(P)` // frees memory that was allocated to P
- The heap of size is 20 bytes, starting at address 0.
- The free list is kept ordered by address (increasing).
- "best fit” free-list searching policy.

Free Space Management (Part 2)

Free List

Heap

```
Initialization
```

Addr:0; sz:20

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
```

Free List

Addr:0; sz:20

Heap

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
```

Free List

Addr:8; sz:12

Heap

0	**P0**
1	**P0**
2	**P0**
3	**P0**
4	**P0**
5	**P0**
6	**P0**
7	**P0**
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
```

Free List

Addr:8; sz:12

Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	P0
7	P0
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 1)

```
1. P0 = Alloc(6);
```

Free List

Addr:0; sz:20

Heap

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
```
`2. P1 = Alloc(9);`

Free List

Addr:-1; sz:0

Heap

0	**P0**
1	**P0**
2	**P0**
3	**P0**
4	**P0**
5	**P0**
6	**P0**
7	**P0**
8	**P1**
9	**P1**
10	**P1**
11	**P1**
12	**P1**
13	**P1**
14	**P1**
15	**P1**
16	**P1**
17	**P1**
18	**P1**
19	**P1**

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
```

Free List

Addr:-1; sz:0

Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	P0
7	P0
8	P1
9	P1
10	P1
11	P1
12	P1
13	P1
14	P1
15	P1
16	P1
17	P1
18	P1
19	P1

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
```

Free List

Addr:8; sz:12

Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	P0
7	P0
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
```

Free List

Addr:8; sz:12

Heap

0	**P0**
1	**P0**
2	**P0**
3	**P0**
4	**P0**
5	**P0**
6	**P0**
7	**P0**
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Free Space Management (Part 2)

1. `P0 = Alloc(6);`
2. `P1 = Alloc(9);`
3. `Free(P1);`
4. **`P2 = Alloc(6);`**

Free List

Addr:16; sz:4

Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	P0
7	P0
8	P2
9	P2
10	P2
11	P2
12	P2
13	P2
14	P2
15	P2
16	
17	
18	
19	

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
```

Free List

Addr:16; sz:4

Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	P0
7	P0
8	P2
9	P2
10	P2
11	P2
12	P2
13	P2
14	P2
15	P2
16	
17	
18	
19	

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
```

Free List

Addr:-1; sz:0

Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	P0
7	P0
8	P2
9	P2
10	P2
11	P2
12	P2
13	P2
14	P2
15	P2
16	P3
17	P3
18	P3
19	P3

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
```
5. P3 = Alloc(3);
6. Free(P0);

Free List

Addr:-1; sz:0

Heap

0	P0
1	P0
2	P0
3	P0
4	P0
5	P0
6	P0
7	P0
8	P2
9	P2
10	P2
11	P2
12	P2
13	P2
14	P2
15	P2
16	P3
17	P3
18	P3
19	P3

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
```

Free List

Addr:0; sz:8

Heap

0	
1	
2	
3	
4	
5	
6	
7	
8	P2
9	P2
10	P2
11	P2
12	P2
13	P2
14	P2
15	P2
16	P3
17	P3
18	P3
19	P3

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
```
`7. P4 = Alloc(9) ;`

Free List

Addr:0; sz:8

Heap

0	
1	
2	
3	
4	
5	
6	
7	
8	**P2**
9	**P2**
10	**P2**
11	**P2**
12	**P2**
13	**P2**
14	**P2**
15	**P2**
16	**P3**
17	**P3**
18	**P3**
19	**P3**

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
```
`7. P4 = Alloc(9) ;`

Free List

Addr:0; sz:8

Alloc failed!

Heap

0	
1	
2	
3	
4	
5	
6	
7	
8	**P2**
9	**P2**
10	**P2**
11	**P2**
12	**P2**
13	**P2**
14	**P2**
15	**P2**
16	**P3**
17	**P3**
18	**P3**
19	**P3**

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
7. P4 = Alloc(9) ;
```
8. Free(P3);

Free List

Addr:0; sz:8

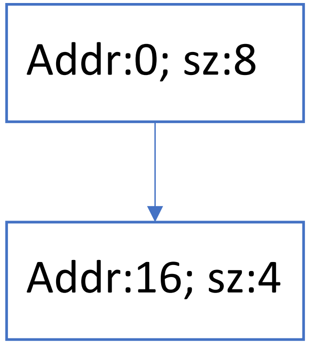
Heap

0	
1	
2	
3	
4	
5	
6	
7	
8	P2
9	P2
10	P2
11	P2
12	P2
13	P2
14	P2
15	P2
16	P3
17	P3
18	P3
19	P3

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
7. P4 = Alloc(9) ;
```
8. Free(P3);

Free List



Heap

0	
1	
2	
3	
4	
5	
6	
7	
8	P2
9	P2
10	P2
11	P2
12	P2
13	P2
14	P2
15	P2
16	
17	
18	
19	

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
7. P4 = Alloc(9) ;
8. Free(P3);
9. P5 = Alloc(7);
```

Free List

Addr:0; sz:8 → Addr:16; sz:4

Heap

0	
1	
2	
3	
4	
5	
6	
7	
8	P2
9	P2
10	P2
11	P2
12	P2
13	P2
14	P2
15	P2
16	
17	
18	
19	

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
7. P4 = Alloc(9) ;
8. Free(P3);
9. P5 = Alloc(7);
```

Free List

Addr:16; sz:4

Heap

0	P5
1	P5
2	P5
3	P5
4	P5
5	P5
6	P5
7	P5
8	P2
9	P2
10	P2
11	P2
12	P2
13	P2
14	P2
15	P2
16	
17	
18	
19	

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
7. P4 = Alloc(9) ;
8. Free(P3);
9. P5 = Alloc(7);
```

`10.P6 = Alloc(1);`

Free List

Addr:16; sz:4

Heap

0	P5
1	P5
2	P5
3	P5
4	P5
5	P5
6	P5
7	P5
8	P2
9	P2
10	P2
11	P2
12	P2
13	P2
14	P2
15	P2
16	
17	
18	
19	

Free Space Management (Part 2)

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
7. P4 = Alloc(9) ;
8. Free(P3);
9. P5 = Alloc(7);
```
`10.P6 = Alloc(1);`

Free List

Addr:-1; sz:0

Heap

0	P5
1	P5
2	P5
3	P5
4	P5
5	P5
6	P5
7	P5
8	P2
9	P2
10	P2
11	P2
12	P2
13	P2
14	P2
15	P2
16	P6
17	P6
18	P6
19	P6

Free Space Management (Part 2)

Free List

Addr:-1; sz:0

```
1. P0 = Alloc(6);
2. P1 = Alloc(9);
3. Free(P1);
4. P2 = Alloc(6);
5. P3 = Alloc(3);
6. Free(P0);
7. P4 = Alloc(9) ;
8. Free(P3);
9. P5 = Alloc(7);
10.P6 = Alloc(1);
```

Advantages/disadvantages of aligned allocation?

Heap

0	P5
1	P5
2	P5
3	P5
4	P5
5	P5
6	P5
7	P5
8	P2
9	P2
10	P2
11	P2
12	P2
13	P2
14	P2
15	P2
16	P6
17	P6
18	P6
19	P6

Paging (simplified version)

Paging (simplified version)

- **Page:** fixed-size portion of virtual memory
- **Frame:** fixed-size portion of physical memory
  - Usually confusingly called a “page frame”
- **Page size = frame size**
- Typical size: 4kb
  - 8kb sometimes, but always power of 2

Paging

Virtual Address Space

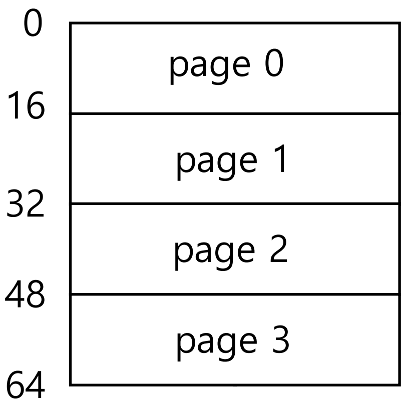
- Linear from 0 up to a multiple of page size

Physical Address Space

- Noncontiguous set of frames, one per page

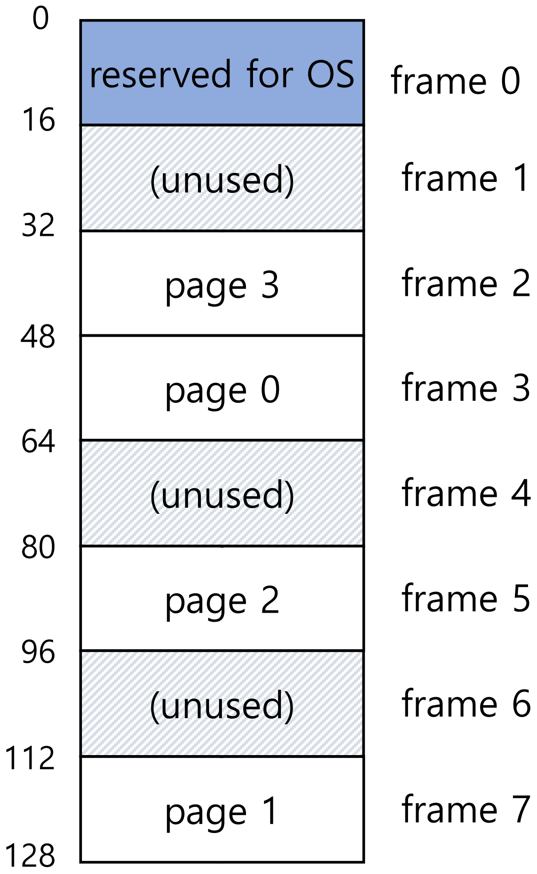
Paging: Example

Contiguous



Virtual Address Space

64B address space with four **16B pages**



Physical Memory

128B physical memory with eight **16B frames**

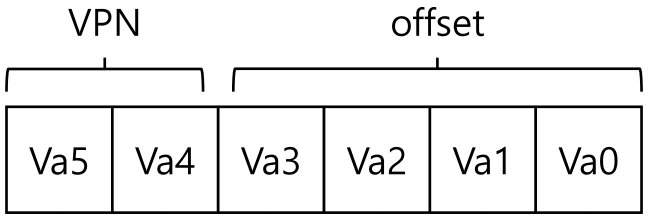
Not contiguous

Not even in order

Paging: Virtual Address

Two components in the virtual address

- VPN: virtual page number
- Offset: offset within the page; Page size = 2^{offset}

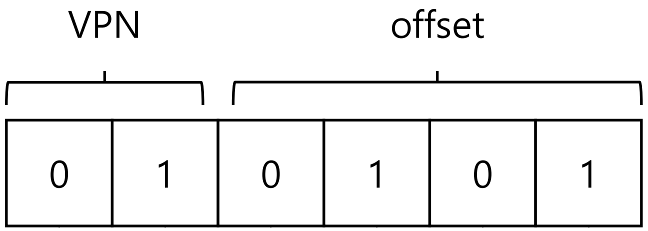


Paging: Virtual Address Example

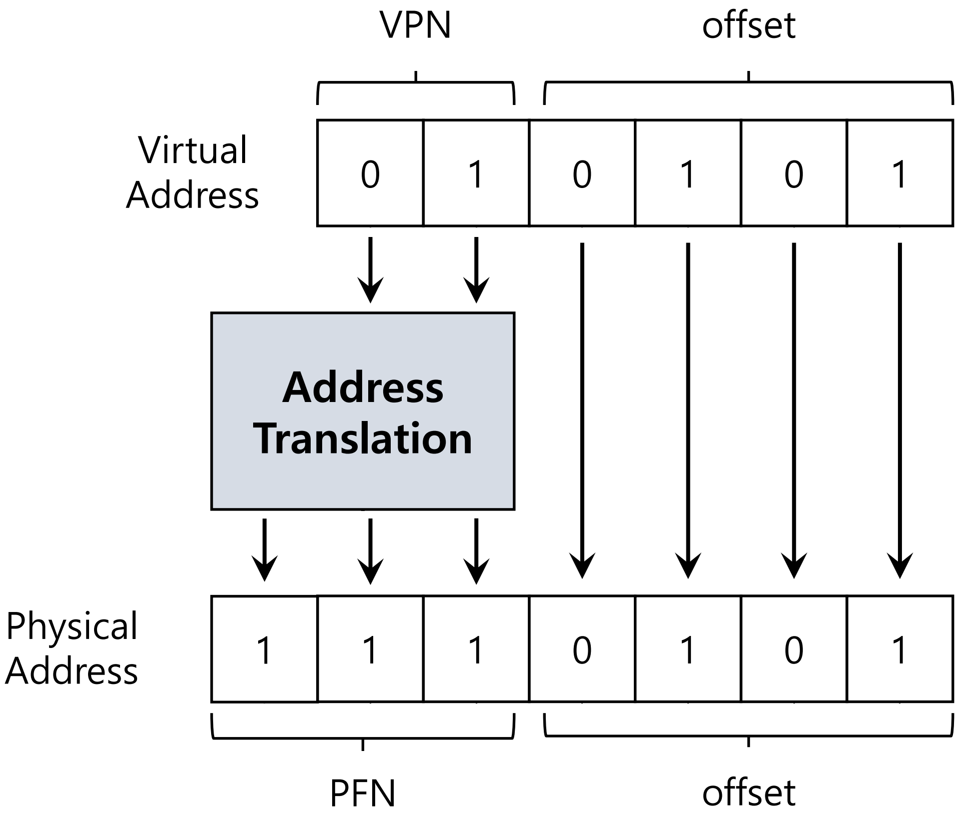
Virtual address 21 (binary 010101) in 64-byte address space

2 VPN bits → Number of pages = 2^2 = 4 pages

4 offset bits → Page size = 2^4 = 16B



Paging: Virtual to Physical Address Example



MMU for Paging (simplified)

Page table

- **Data structure** used to map the virtual address to physical address
- Indexed by page number
- Contains frame number of page in memory
- **Each process has a page table**

MMU also needs:

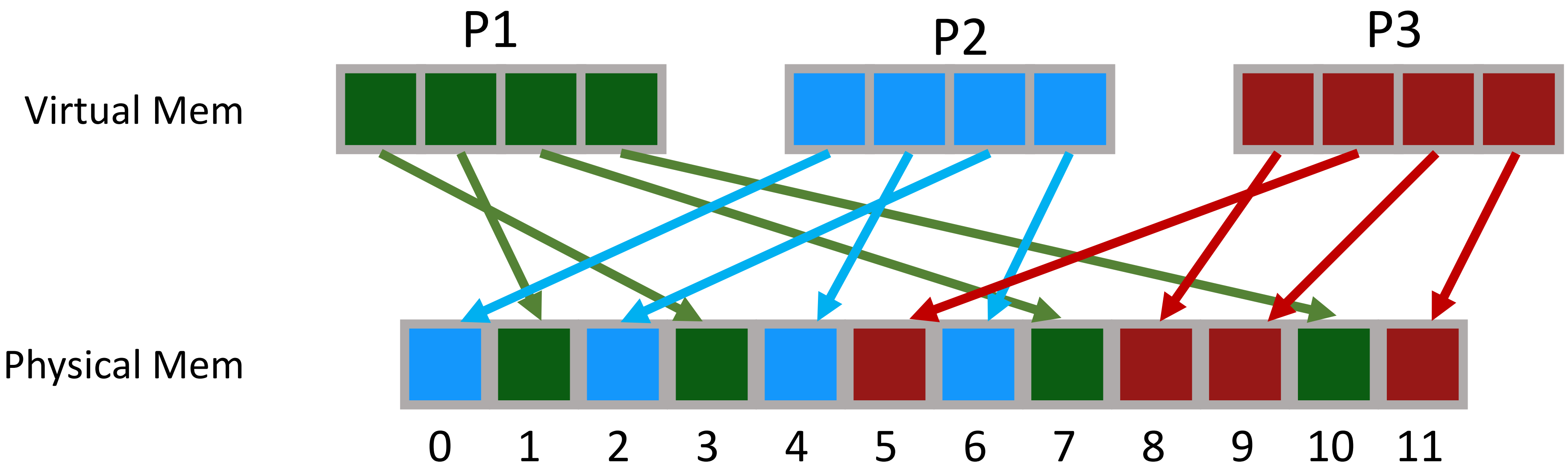
Pointer to page table in memory

Length of page table

Operating system also needs:

Reverse mapping of frame numbers to which pages they hold for which processes

Quiz: Fill in Page Table



Page Tables:

P1	P2	P3
3	0	8
1	4	5
7	2	9
10	6	11

Page Tables can get large

32-bit address space with 4-KB pages, 20 bits for VPN

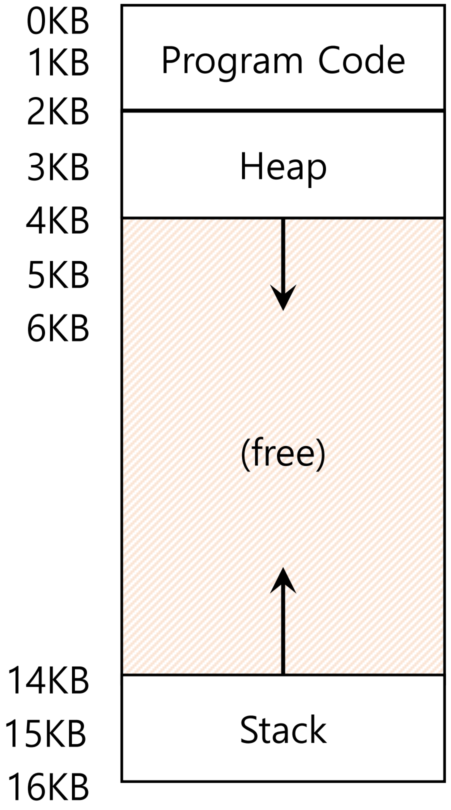
$$4MB = 2^{20}\ entries\ * 4B\ per\ page\ table\ entry$$

64-bit address space with 4-KB pages, 52 bits for VPN

$$2^{52}\ entries\ * 8B\ per\ page\ table\ entry = Petabyte - scale\ !$$

True, but address space is often sparsely used

Problem?



Virtual Address Space

Address space **sparsely used**

Access to unused portion will appear valid

Would rather have an error

Unused portion could be given to other procs

Partial Solution: Valid/Invalid Bit

Have valid bit in each page table entry

- Set to valid for used portions of address space
- Invalid for unused portions
- Some pages might be “swapped”
- Meaning: they are used, but not currently in RAM (will see more of this!)
- We call those invalid (or “absent”) as well

→ This is the common approach

But What About Huge Tables?

- Valid/Invalid bits don't solve the petabyte-scale table problem
  - Because they change the information in the table, not its size

- Real solution: **Multi-Level Page Tables**
- Page table entries point to *other page tables*
  - Need fewer page table entries per table
  - Outer level invalid bit means the next level table doesn't exist.
  - Now we really are saving (massive amounts of) space!
- Will get into detail next week.

Main Memory Allocation with Paging

Virtual address space: fixed size pages

Physical address space: fixed size frames

New process

- Find frames for all of process's pages

Easier problem ☺

- **Fixed size**

(Internal) Fragmentation in Paging

With paging

- Address space = multiple of page size

Part of one page of each region (code, stack, heap) may be unused

With reasonable page size, not a big problem

Summary – 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

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).