Week 6

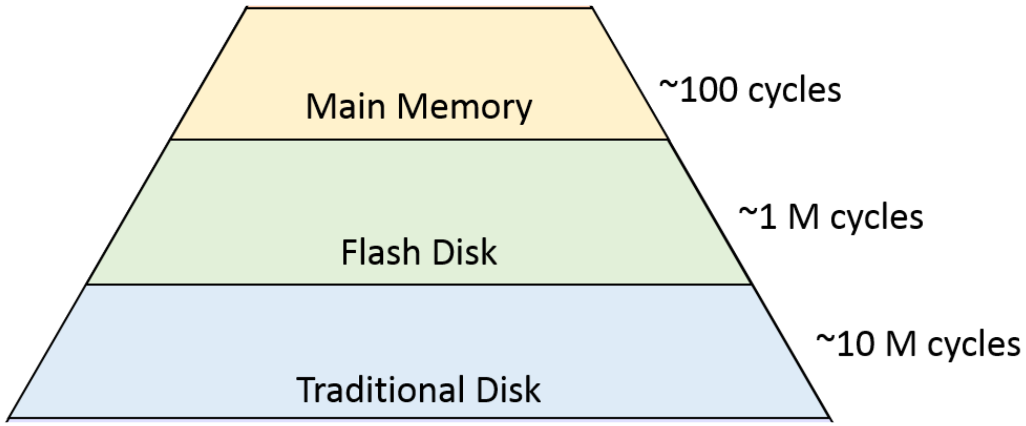
Memory Management: Virtual Memory

Oana Balmau

February 9, 2023

Recap: OS Memory Management

Processor

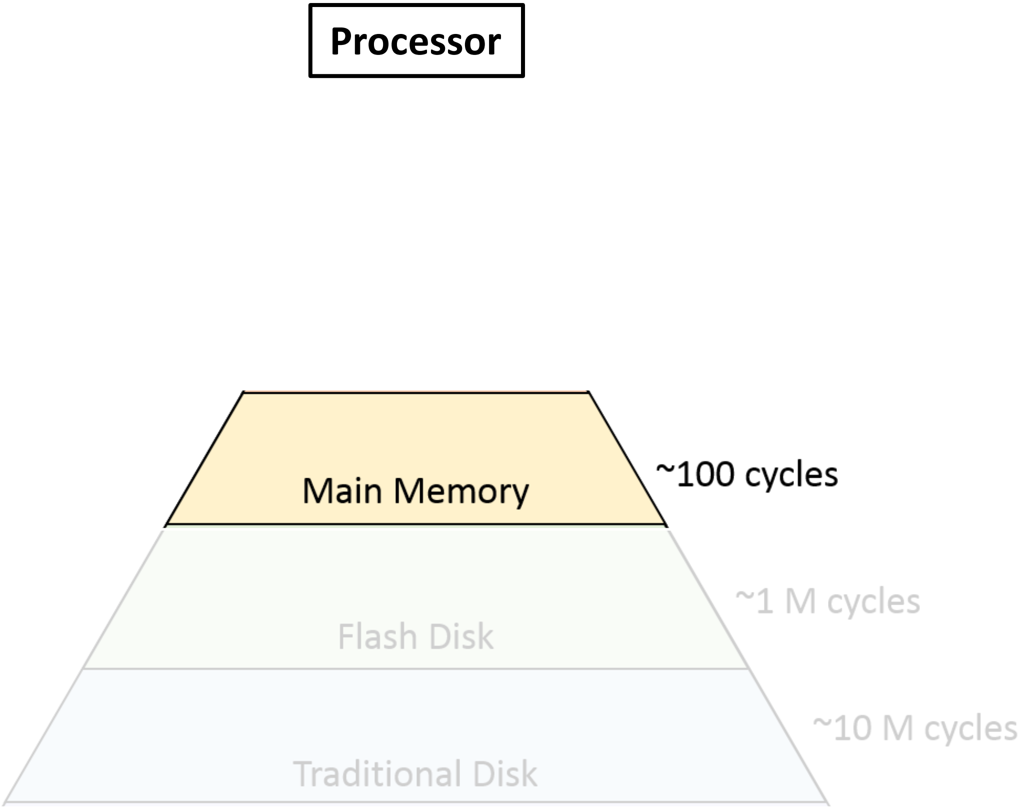


Recap: Simplifying Assumption

So for today:

All of a program must be in main memory

Not concerned with disk

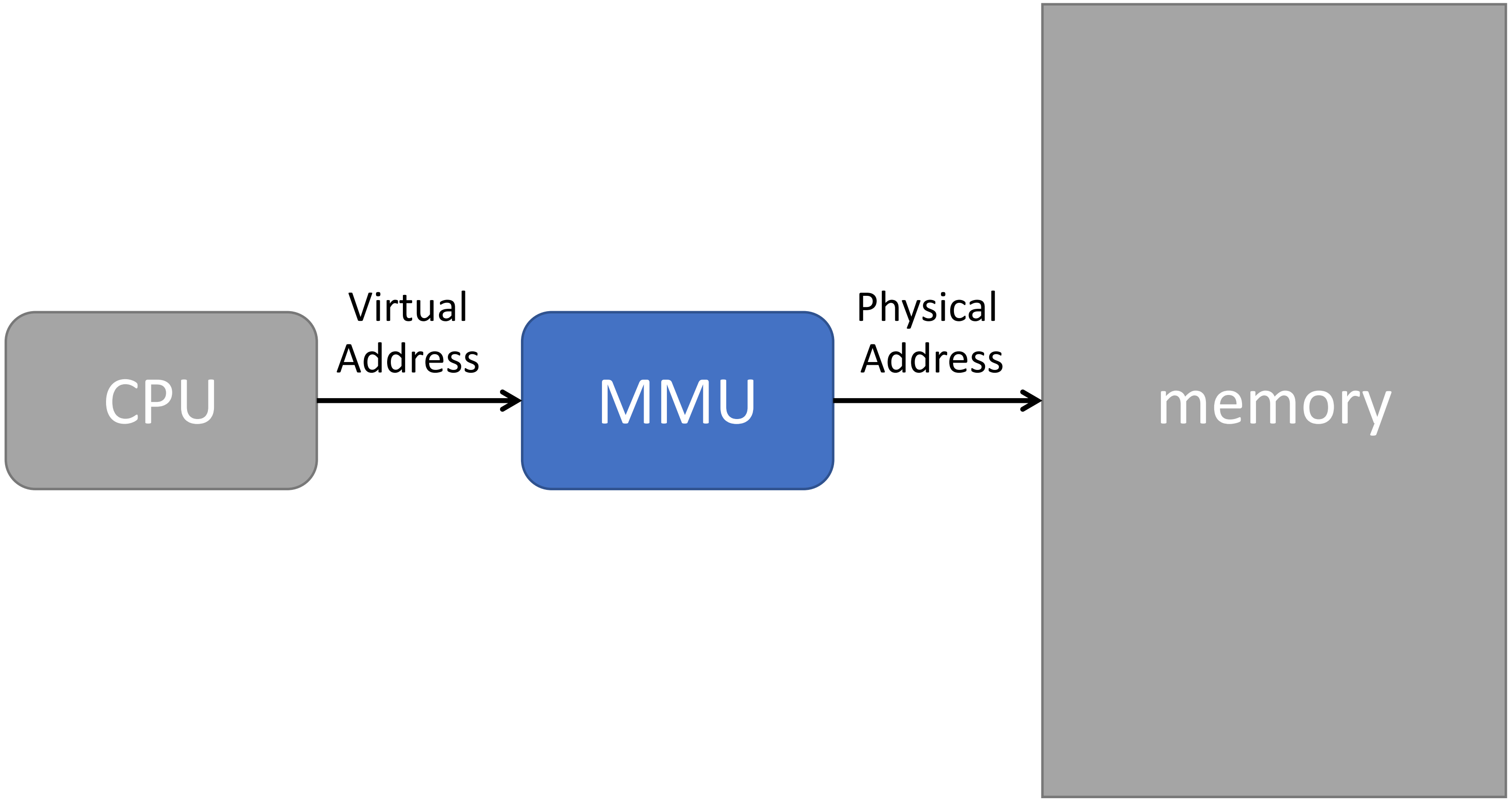


Recap: Virtual vs. Physical address space

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

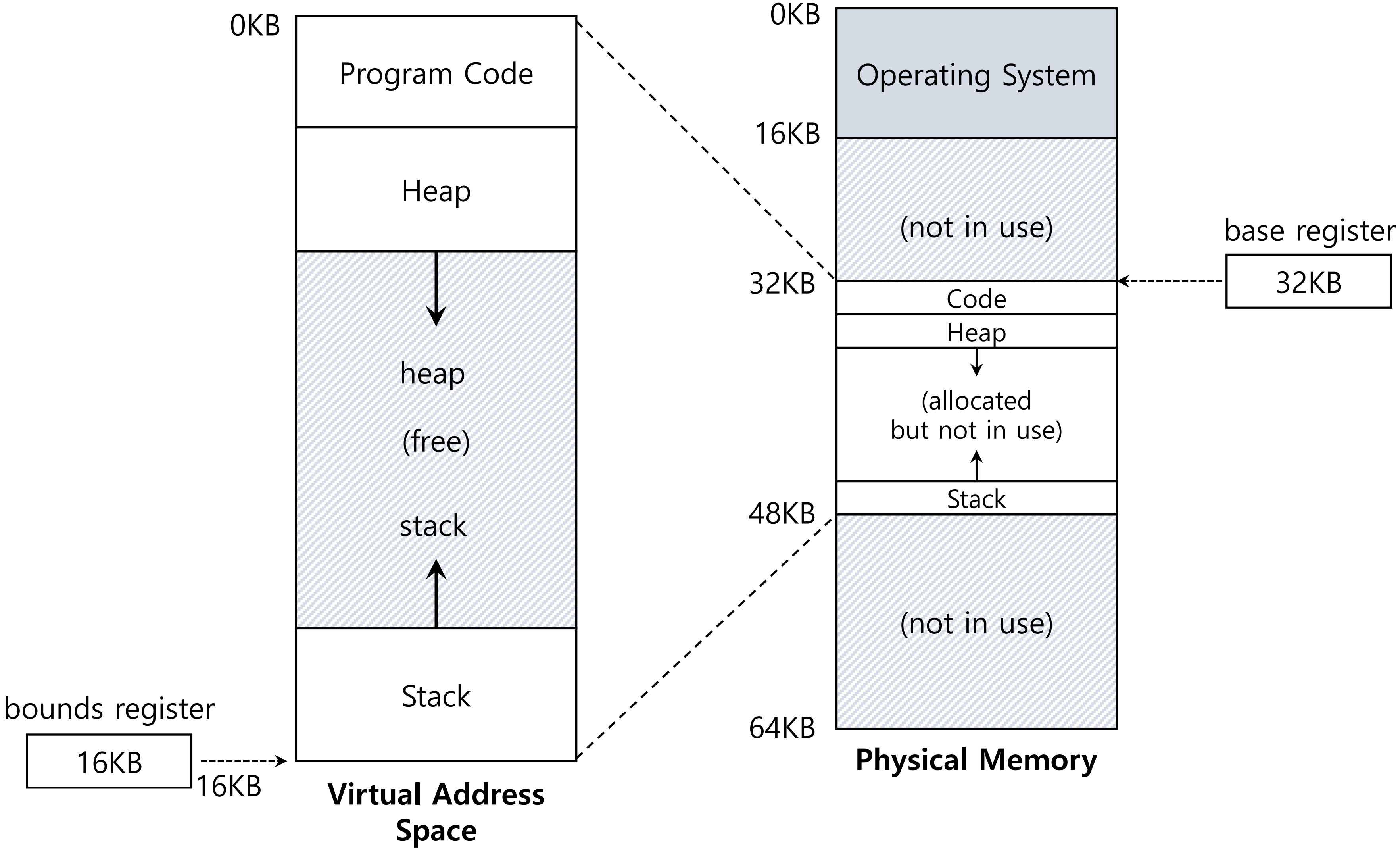
Virtual Address Space

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

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

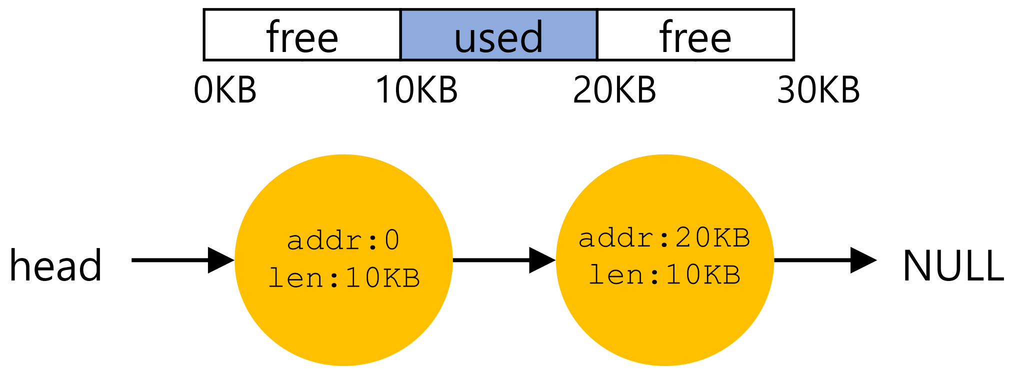
Recap:
Base and Bounds (External) Fragmentation

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.

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

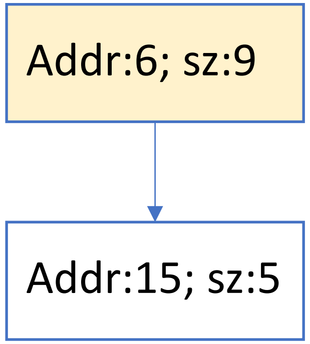
Heap

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

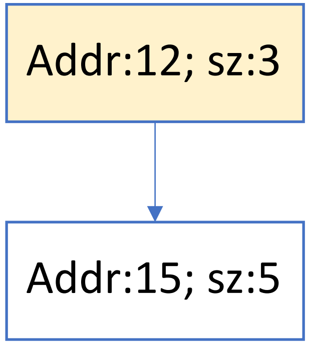
Heap

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

Addr:12; sz:3

↓

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	
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

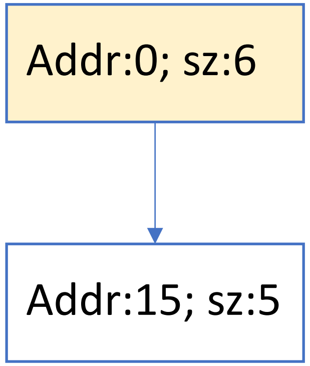
Heap

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



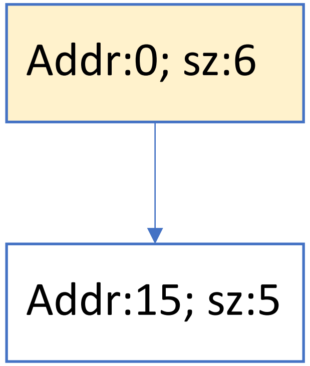
Heap

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



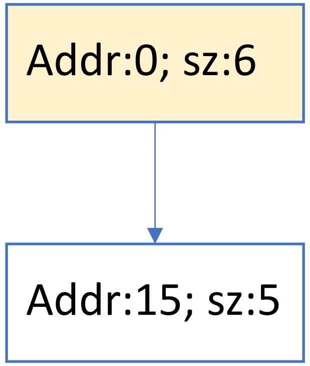
Heap

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



Alloc failed!

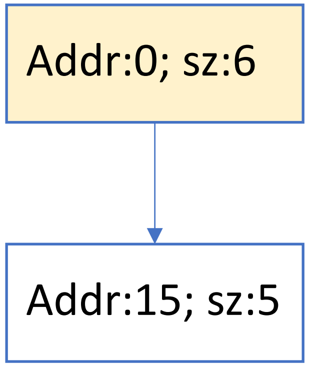
Heap

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



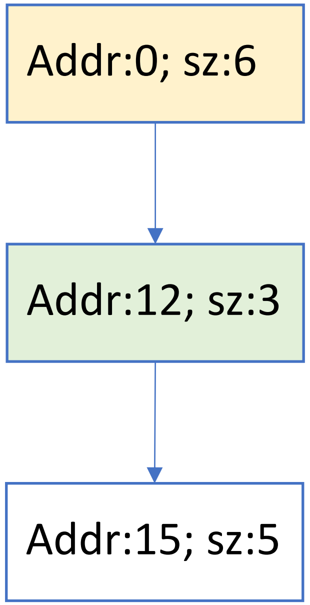
Heap

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

Addr:0; sz:6 → Addr:12; sz:3 → Addr:15; sz:5

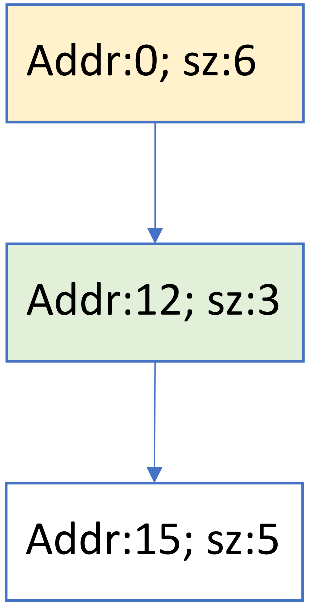
Heap

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!

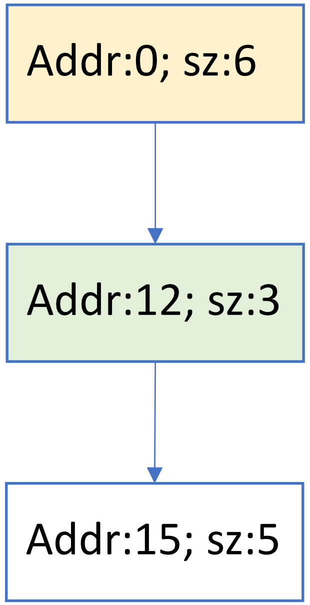
Heap

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



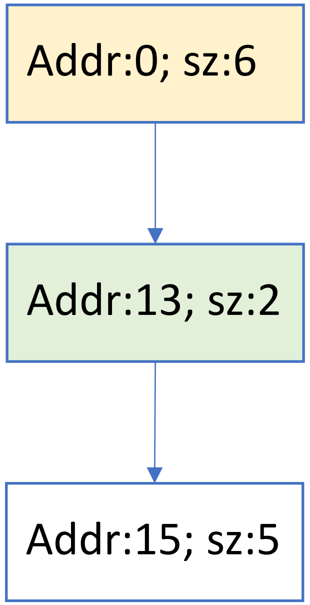
Heap

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	

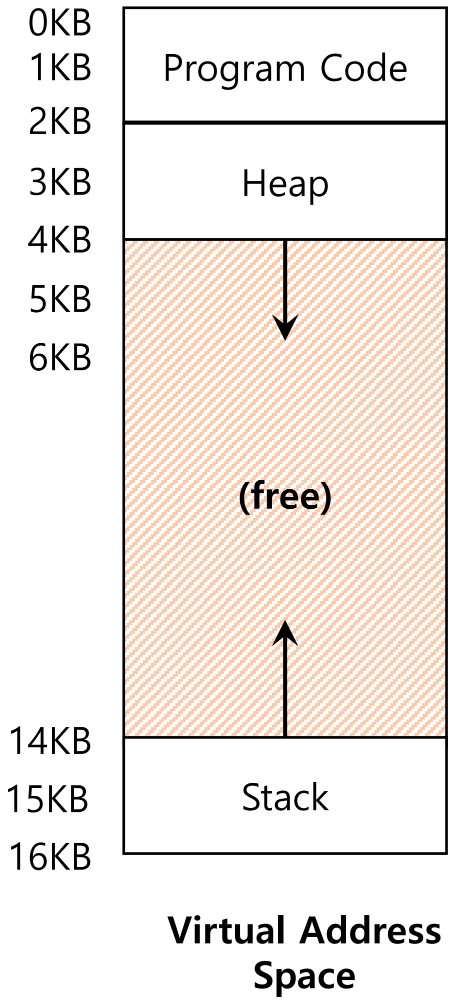
Different Virtual to Physical Mapping Schemes

- Base and bounds

- Segmentation

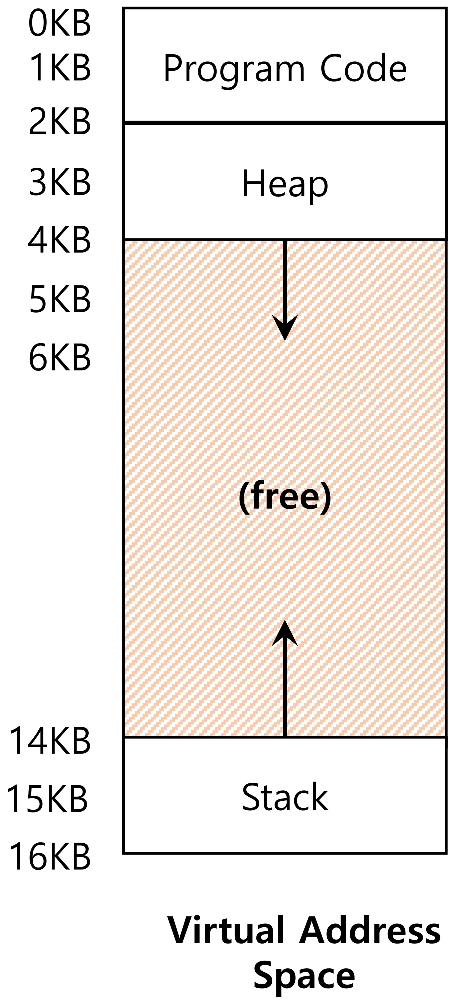
- (Simplified) Paging

Base and Bounds: (Internal) Fragmentation



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

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

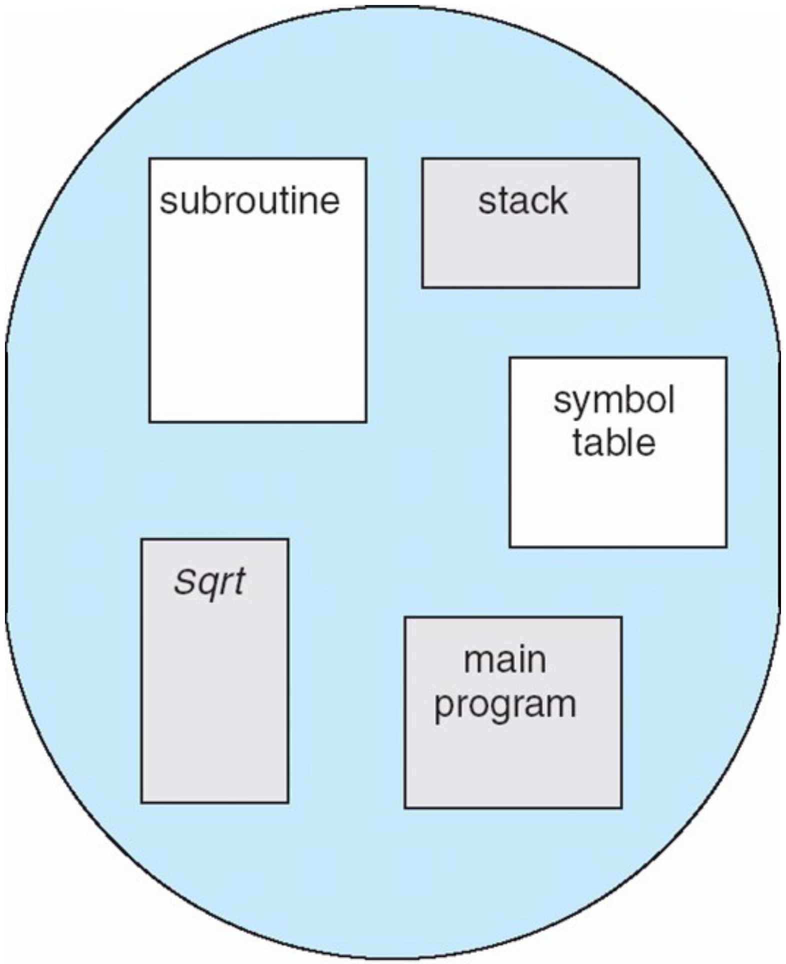
- Two-dimensional
- Set of segments 0 .. n
- Each segment i is linear from 0 to MAX_i

Physical Address Space

- Set of segments, each linear

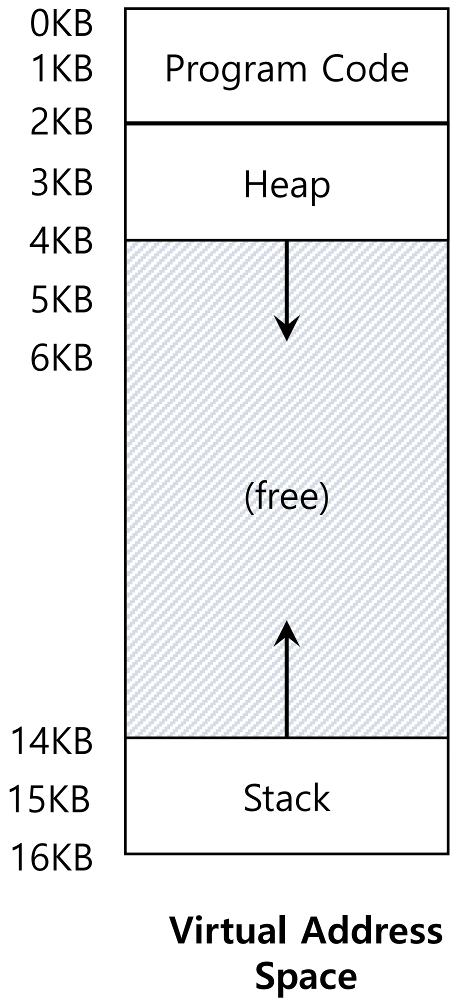
What is a Segment?

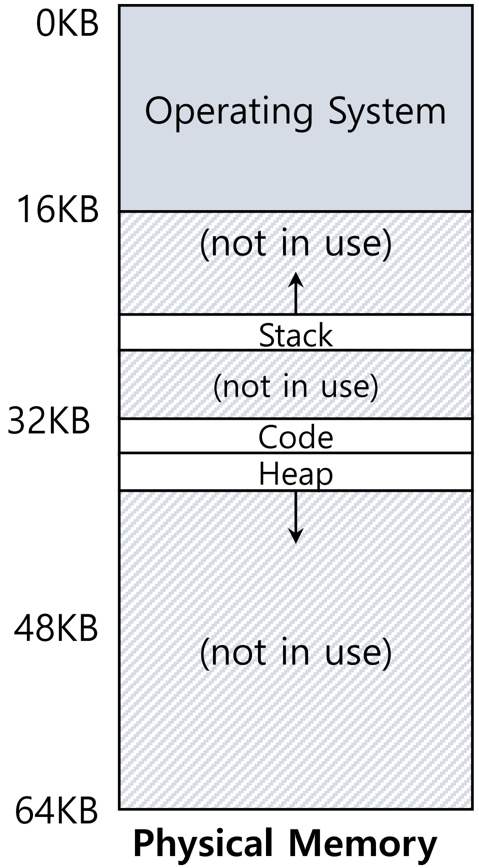
- Anything you want it to be
- Typical examples:
  - **Code**
  - **Heap**
  - **Stack**



Segments

Segmentation Example





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

Segmentation: Virtual Address

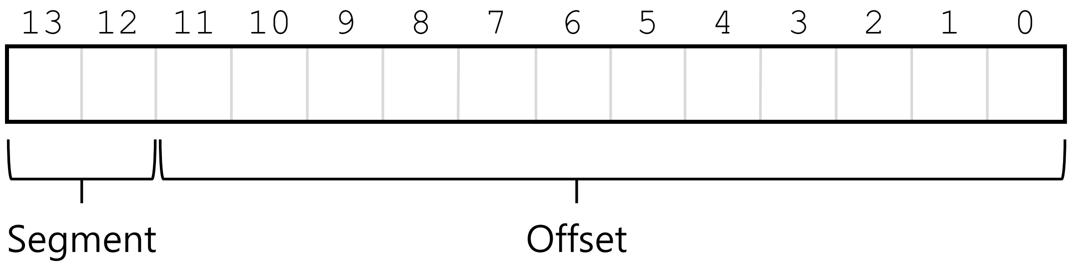
Two-dimensional address:

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

It is like multiple base-and-bounds

Segmentation Virtual Address example

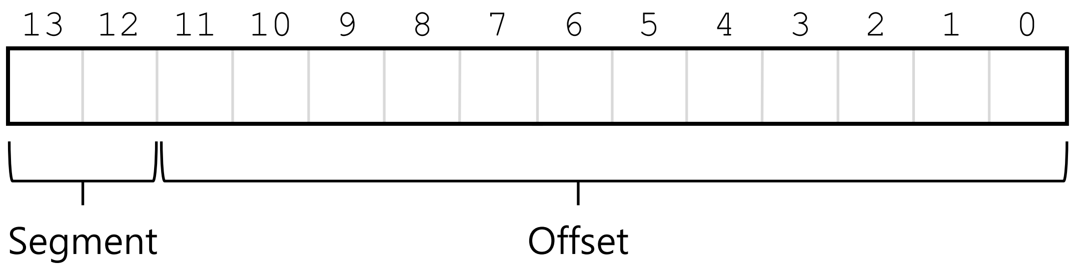
Chop up the address space into segments based on the **top few bits** of virtual address.



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.

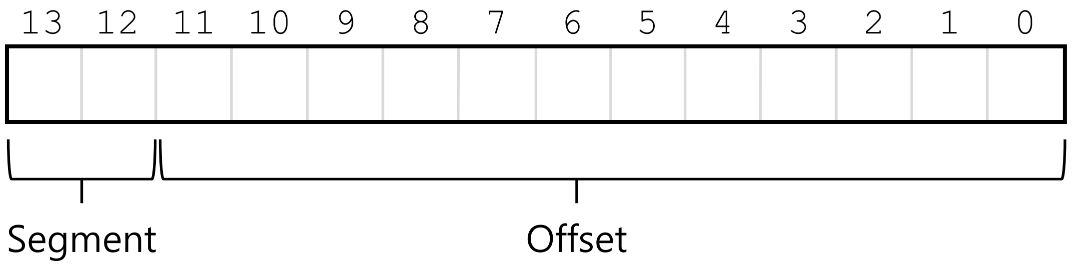


How many segments?
2^2 = 4 segments

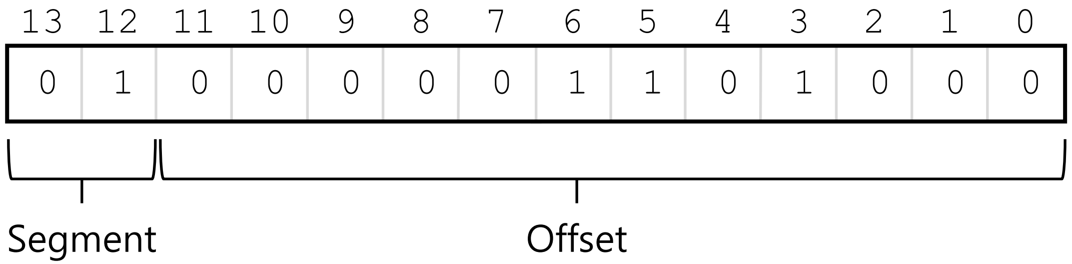
What is the size of each segment?
2^12 = 4KB

Segmentation Virtual Address example

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



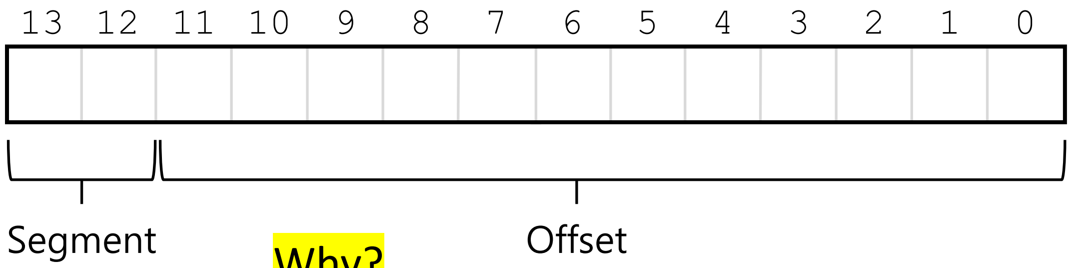
Example: virtual address 4200_{10} (01000001101000_2)



Segment	bits
Code	00
Heap	01
Stack	10
-	11

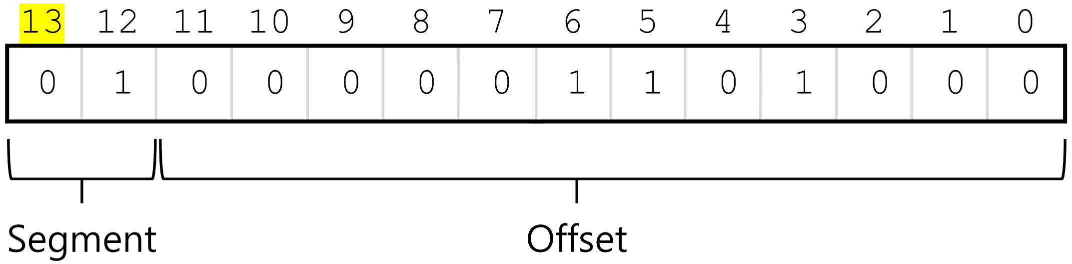
Segmentation Virtual Address example

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



Why?

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



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

Also need:

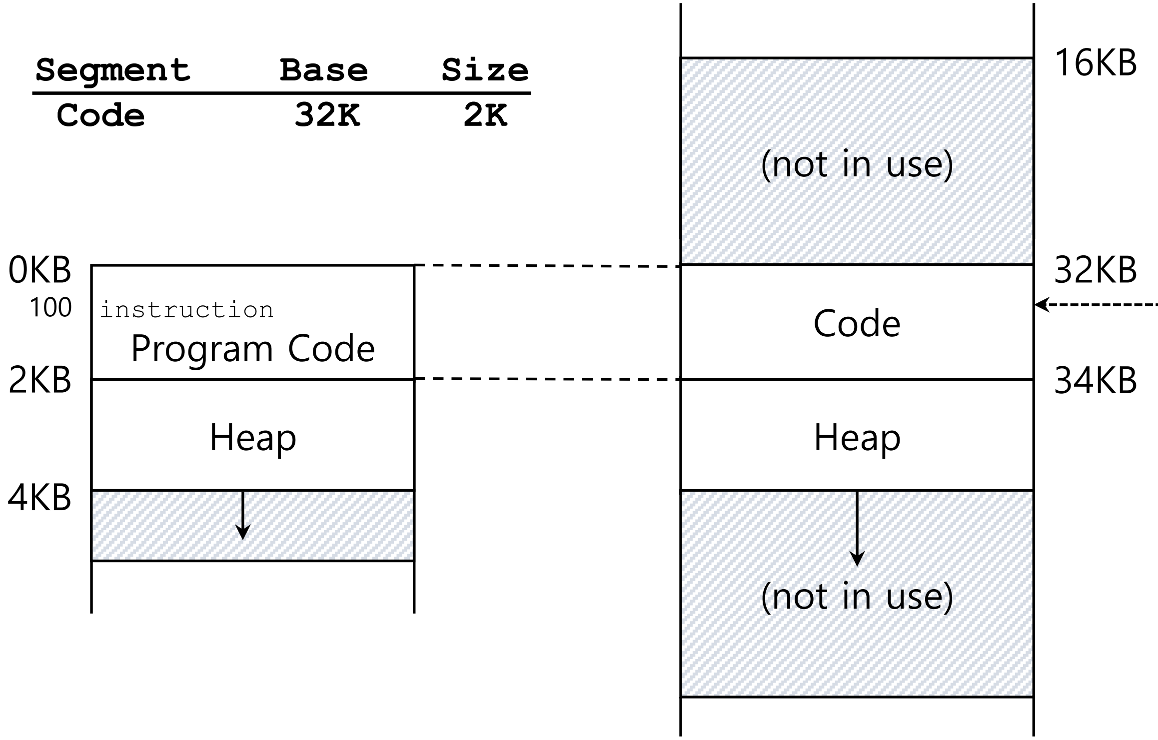
Pointer to segment table in memory

Length of segment table

Segmentation: Address Translation Example

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

Segment	Base	Size
Code	32K	2K



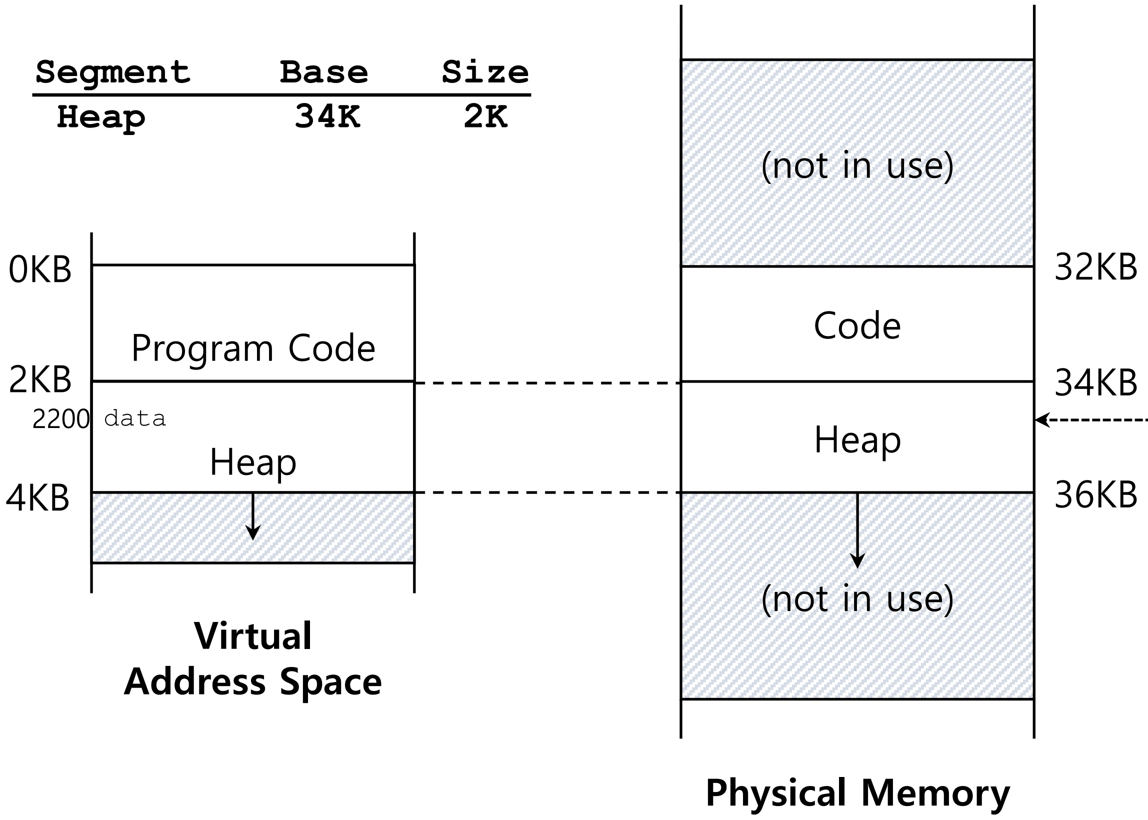
$100 + 32K\ or\ 32868$
is the desired physical address

Segmentation: Address Translation (Cont.)

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

Segment	Base	Size
Heap	34K	2K

152 = 2200 - 2048



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

Let’s practice!

Segmentation

Given a CPU with 3-bit instructions, and 16 bytes byte-addressable of 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}.

*X_{10} means X in base 10 (decimal).

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, because CPU has 3-bit instructions**

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 line in physical memory has 1 byte)**

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.

	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.

	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

- 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	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.

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

4

	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

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			
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	**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

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

VA			
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.

	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

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

6

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			
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	
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

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
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 1

SEG 0

- 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?

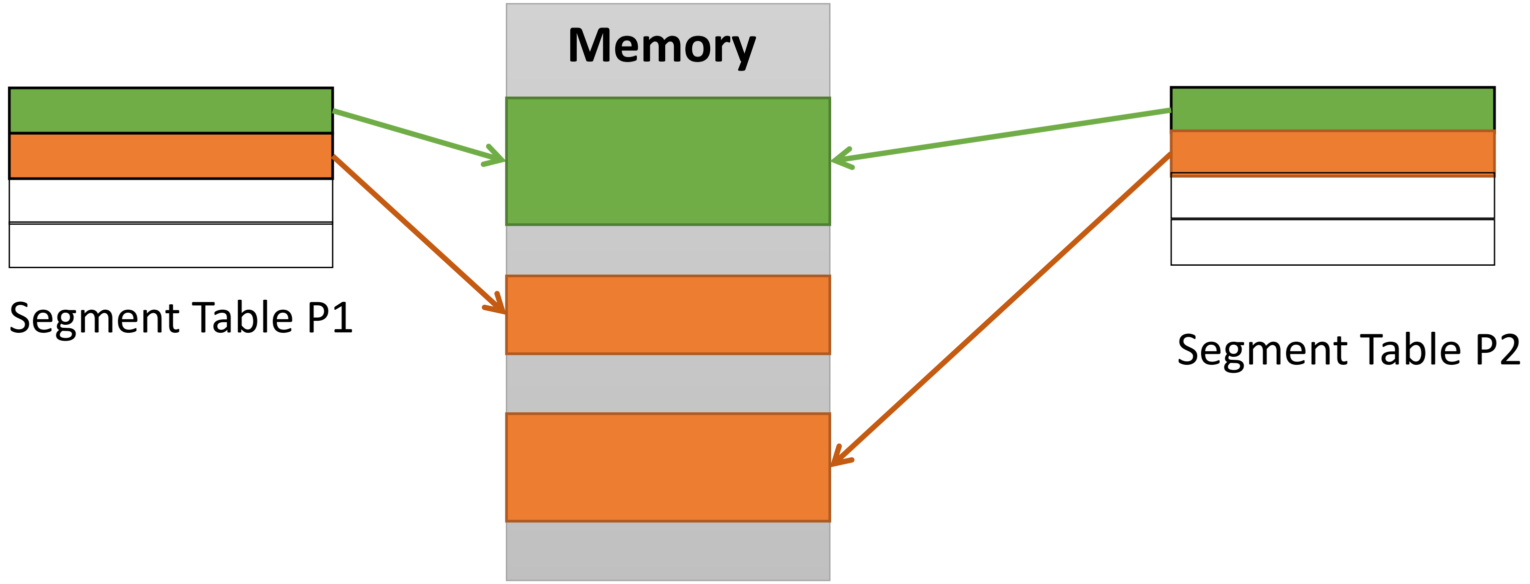
For instance,

- 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

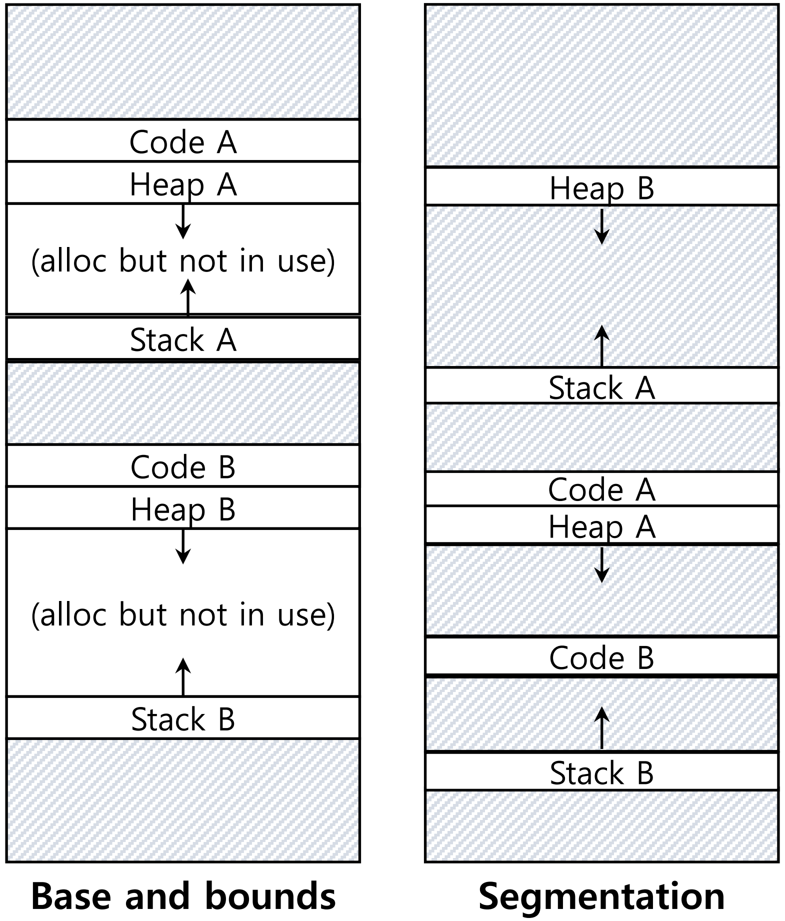
Remember:

Segmentation ~= multiple base-and-bounds

No internal fragmentation inside each segment.

External fragmentation problem is similar.

- Pieces are typically smaller



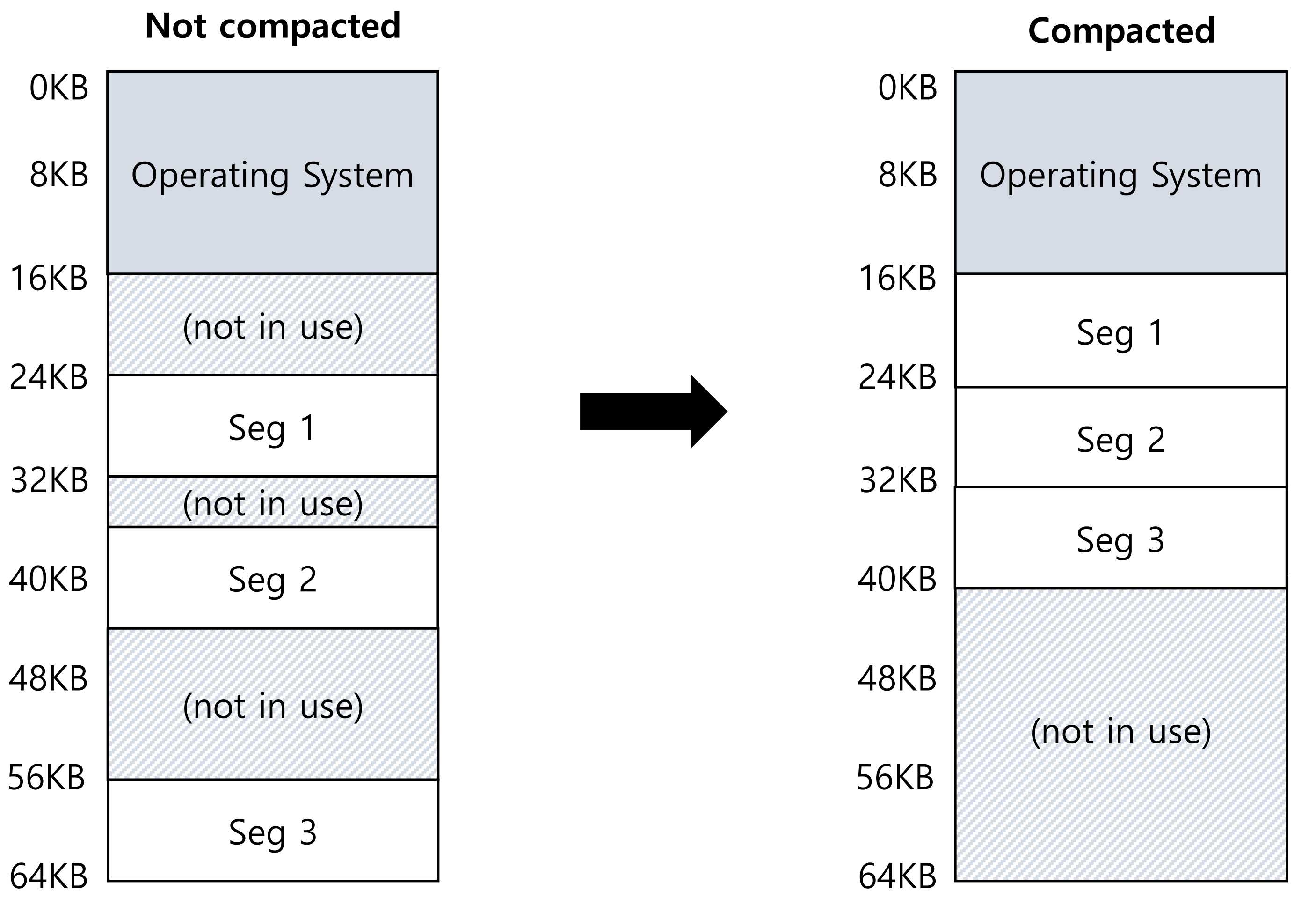
Main memory allocation with Segmentation

Compaction:

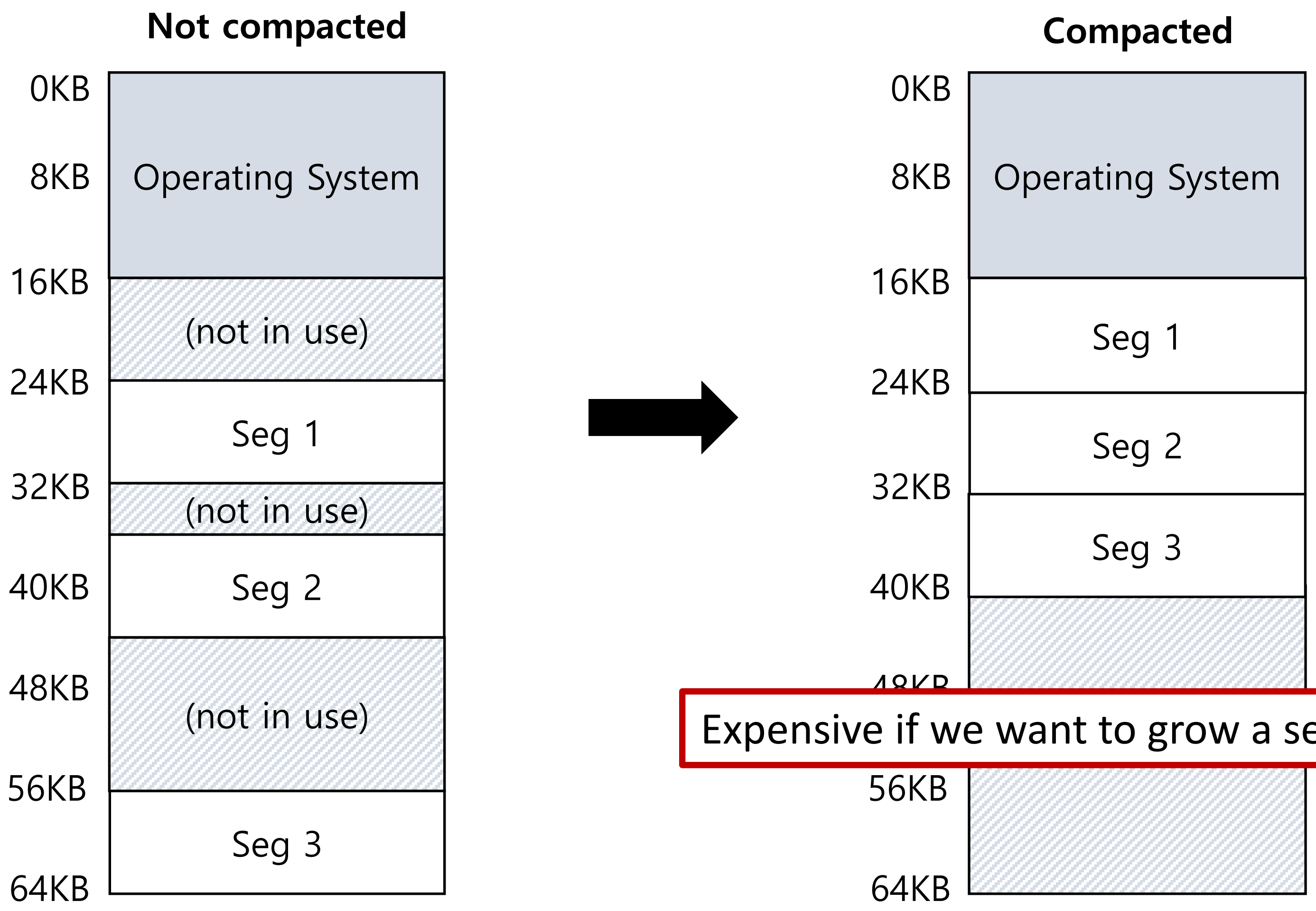
Rearrange segments in physical memory to get rid of “holes”.

- **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	

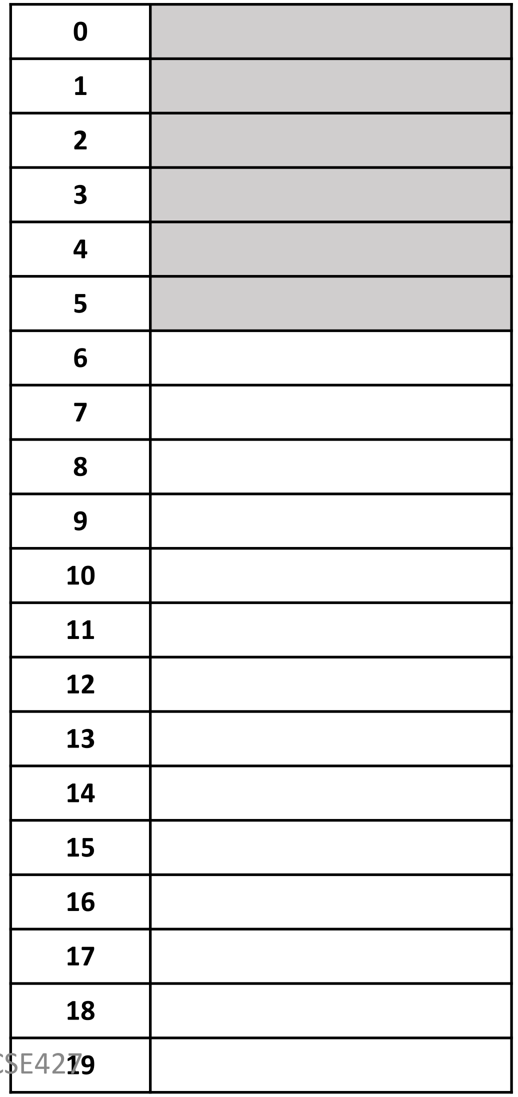
Free Space Management (Part 1)

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

Free List

Addr:0; sz:20

Heap



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

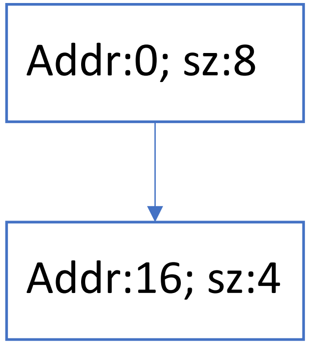
Heap

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



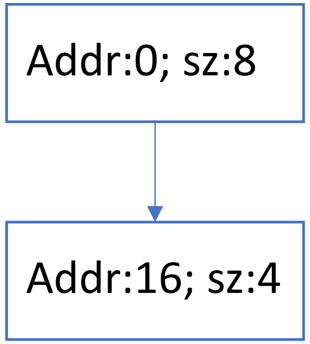
Heap

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



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)

```
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

Advantages/disadvantages of aligned allocation?

Paging (simplified version)

Paging (simplified version)

- **Page:** fixed-size portion of virtual memory
- **Frame:** fixed-size portion of physical memory
- **Page size = frame size**
- Typical size: 4k – 8k (always power of 2)

Paging

Virtual Address Space

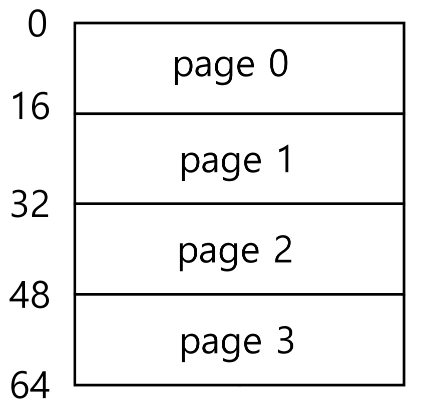
- Linear from 0 up to a multiple of page size

Physical Address Space

- Noncontiguous set of frames, one per page

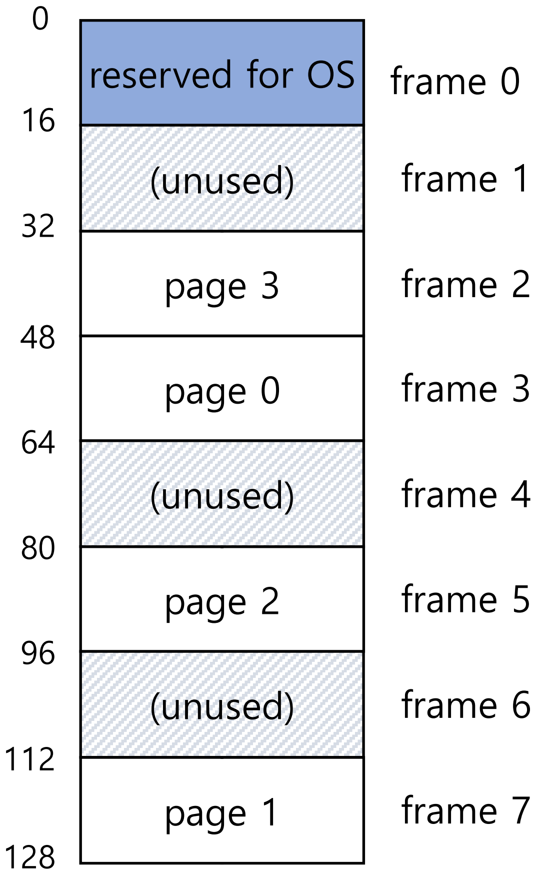
Paging: Example

Contiguous



Virtual Address Space

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



Physical Memory

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

Not
Contiguous

Paging: Virtual Address

Two components in the virtual address

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

VPN		offset			
Va5	Va4	Va3	Va2	Va1	Va0

Paging: Virtual Address Example

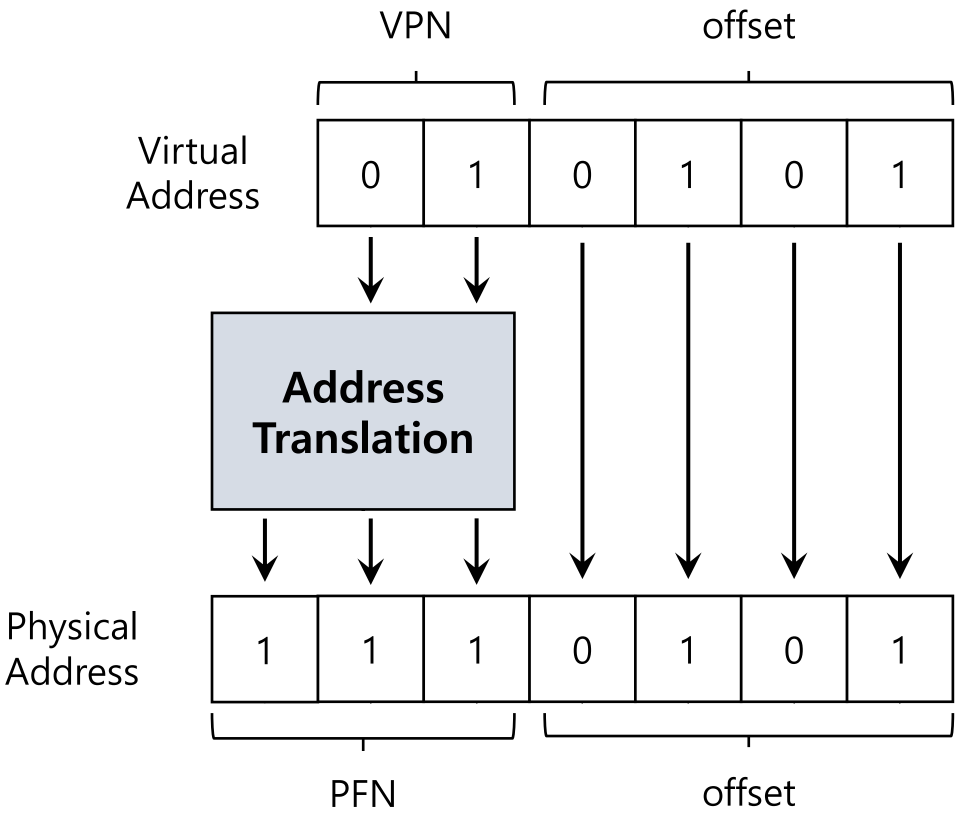
Virtual address 21 (binary 010101) in 64-byte address space

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

4 offset bits → Page size = 2^4 = 16B

$$\overbrace{\boxed{0}\,\boxed{1}}^{\text{VPN}}\,\overbrace{\boxed{0}\,\boxed{1}\,\boxed{0}\,\boxed{1}}^{\text{offset}}$$

Paging: Virtual to Physical Address Example



MMU for Paging

Page table

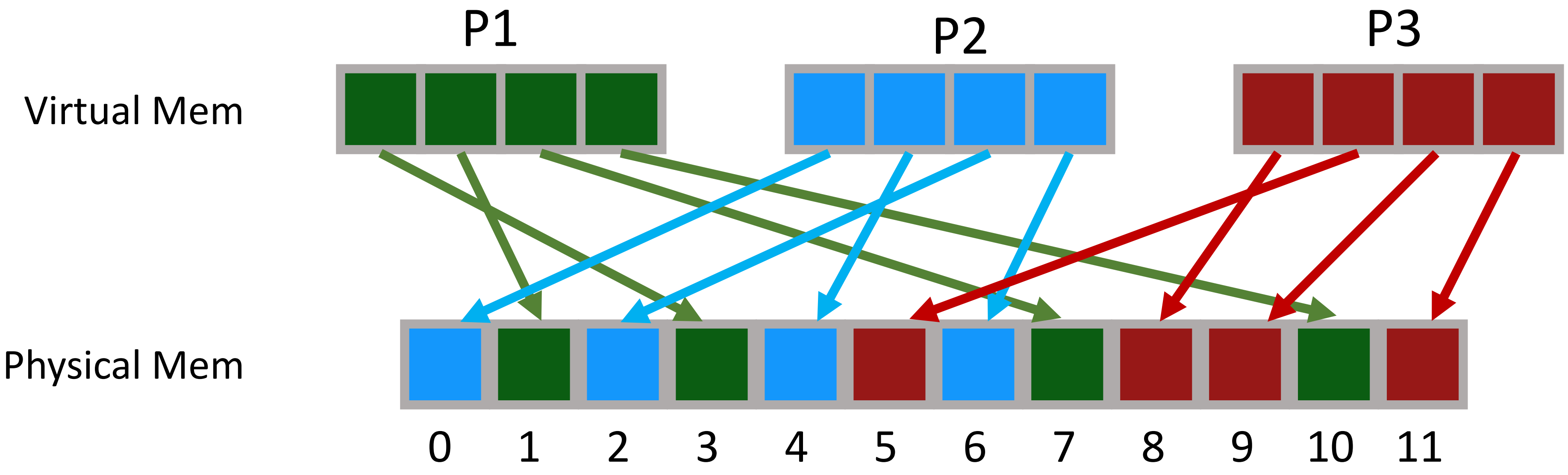
- **Data structure** used to map the virtual address to physical address
- Indexed by page number
- Contains frame number of page in memory
- **Each process has a page table**

Also need:

Pointer to page table in memory

Length of page table

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

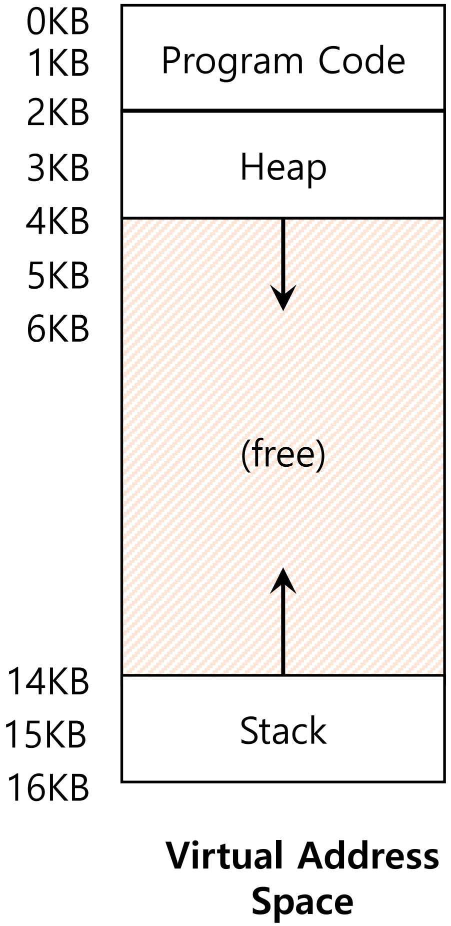
$$4MB = \ 2^{20}\ entries\ \ *4B\ per\ page\ table\ entry$$

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

$$2^{52}\ entries\ \ *4B\ per\ page\ table\ entry = \ Petabyte - scale\ !$$

True, but address space is often sparsely used

Problem?



Address space **sparsely used**

Access to unused portion will appear valid

Would rather have an error

Solution: Valid/Invalid Bit

Page table has length **2^p**

→ Page table does not cover the entire possible virtual address space, only the pages that the process has allocated.

Have valid bit in each page table entry

- Set to valid for used portions of address space
- Invalid for unused portions

→ This is the common approach

Main Memory Allocation with Paging

Logical 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 last page 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).