

# CS 471 Operating Systems

Yue Cheng

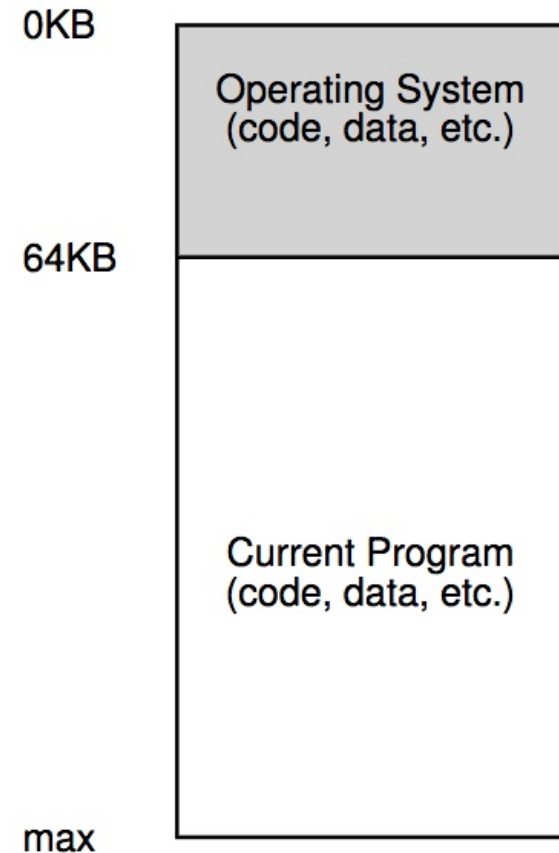
George Mason University  
Spring 2019

# Outline

- Address space
- Virtual memory accesses
- Relocation
- Segmentation

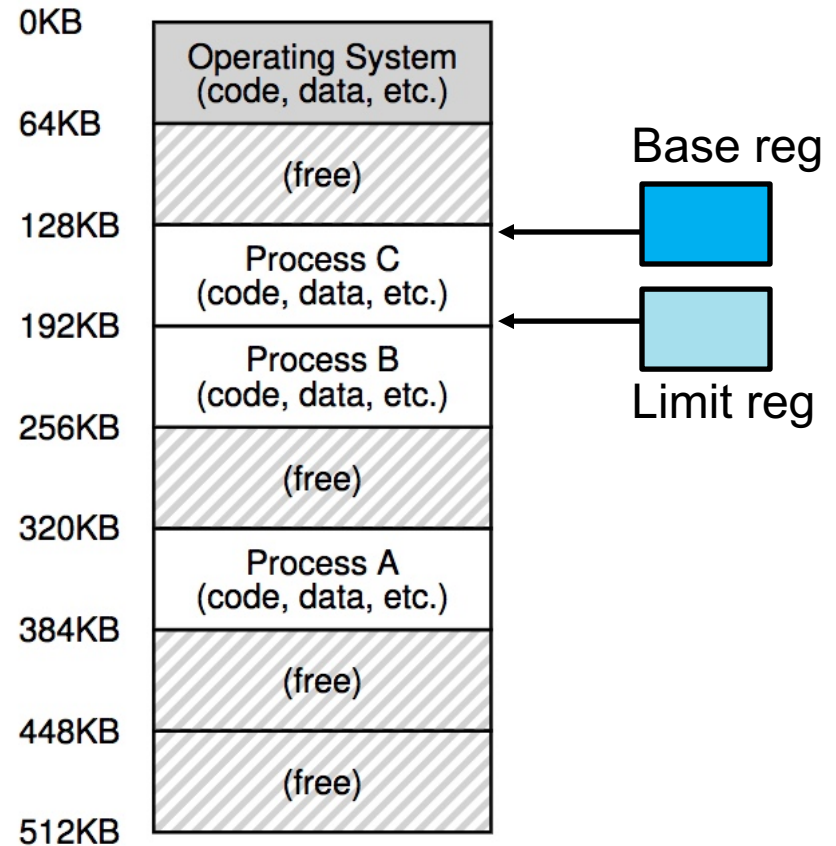
# Early Systems

- OS was a set of libraries
- OS sat in memory starting at physical address 0
- The rest was used by running program



# Multiprogramming & Time Sharing

- OS makes sure each process is confined to its own **address space** in memory
- One naïve implementation:
  - **<base register & limit register>** pair



# The Abstraction

- A process has a set of addresses that map to a collection of bytes
- This set is called an **address space**
- Review: what stuff is in an address space?

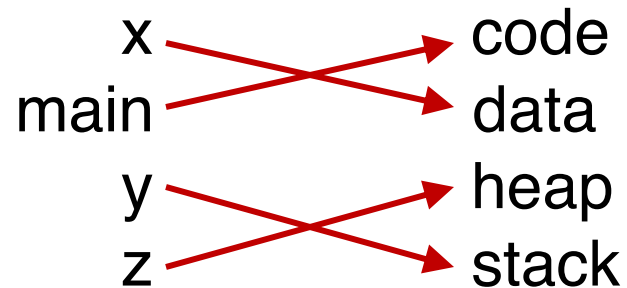
# Match that Segment!

```
int x;  
int main(int argc, char *argv[]) {  
    int y;  
    int *z = malloc(sizeof(int));  
}
```

x	code
main	data
y	heap
z	stack

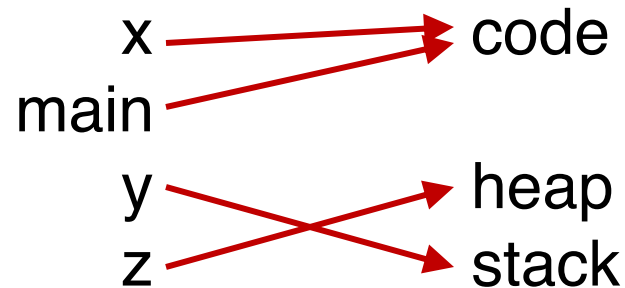
# Match that Segment!

```
int x;  
int main(int argc, char *argv[]) {  
    int y;  
    int *z = malloc(sizeof(int));  
}
```



# Match that Segment!

```
int x;  
int main(int argc, char *argv[]) {  
    int y;  
    int *z = malloc(sizeof(int));  
}
```

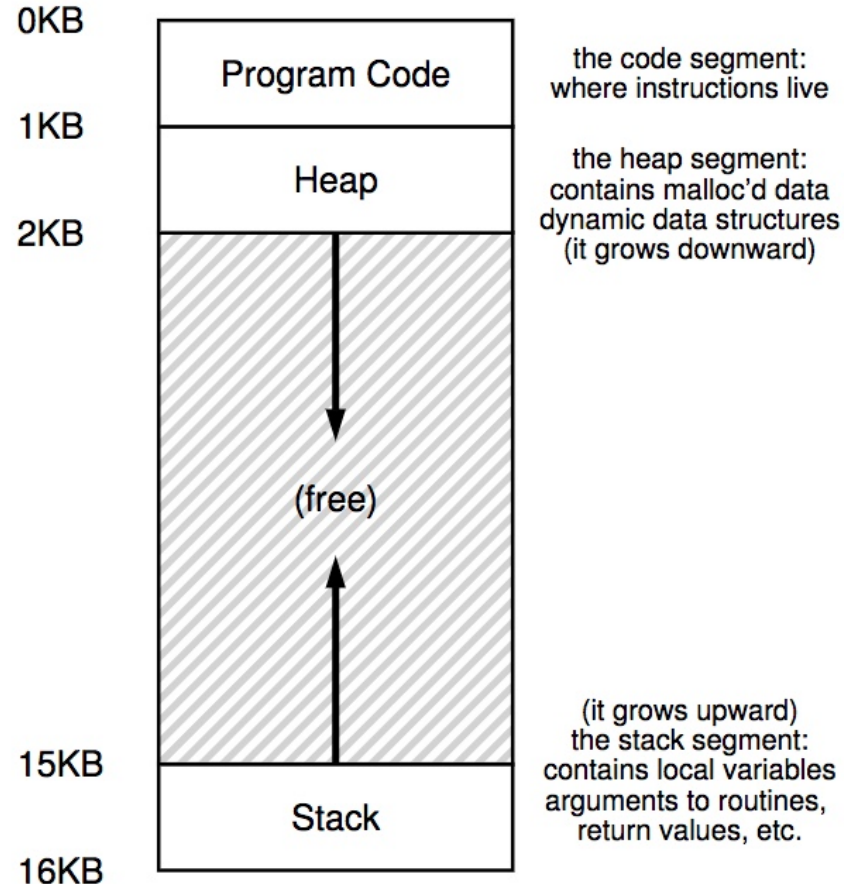


In OSTEP



# The Address Space

- Address space
  - An easy-to-use **abstraction** of physical memory
- The address space is the running program's view of memory in the system
  - **Virtual address** or **logical address**
  - Physical address refers to those seen by the memory unit hardware
- The user program generates *logical* addresses; it never sees the **real** physical addresses



# High-level Goals

- Transparency
  - User program behaves as if it has its own private physical memory
- Efficiency
  - Space and time efficient memory virtualization
  - Performance relies on hardware support (e.g., TLBs)
- Protection
  - Isolation property
  - User process shouldn't access or affect anything outside its own address space

# All Memory Addresses You See are Virtual

- Any address that a programmer can see is a virtual address

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    printf("location of code   : %p\n", (void *) main);
    printf("location of heap   : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack  : %p\n", (void *) &x);
    return x;
}
```

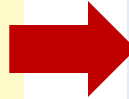
## Result:

```
location of code   : 0x1095afe50
location of heap   : 0x1096008c0
location of stack  : 0x7fff691aea64
```

# Virtual Memory Accesses

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int x;
    x = x + 2;
}
```



```
_main:
00000000100000fa0 pushq   %rbp
00000000100000fa1 movq    %rsp, %rbp
00000000100000fa4 xorl    %eax, %eax
00000000100000fa6 movl    %edi, -0x4(%rbp)
00000000100000fa9 movq    %rsi, -0x10(%rbp)
00000000100000fad movl    0x8(%rbp), %edi
00000000100000fb0 addl    $0x2, %edi
00000000100000fb3 movl    %edi, 0x8(%rbp)
00000000100000fb6 popq   %rbp
00000000100000fb7 retq
```

% otool -tv demo  
(or objdump in Linux)

# Virtual Memory Accesses

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int x;
    x = x + 2;
}
```


```
_main:
00000000100000fa0 pushq    %rbp
00000000100000fa1 movq    %rsp, %rbp
00000000100000fa4 xorl    %eax, %eax
00000000100000fa6 movl    %edi, -0x4(%rbp)
00000000100000fa9 movq    %rsi, -0x10(%rbp)
00000000100000fad movl    0x8(%rbp), %edi
00000000100000fb0 addl    $0x2, %edi
00000000100000fb3 movl    %edi, 0x8(%rbp)
00000000100000fb6 popq    %rbp
00000000100000fb7 retq
```

% otool -tv demo  
(or objdump in Linux)

# Virtual Memory Accesses

```
%rip = 0x100000fad  
%rbp = 0x200
```

**Memory accesses:**




```
0x100000fad movl    0x8(%rbp), %edi  
0x100000fb0 addl    $0x2, %edi  
0x100000fb3 movl    %edi, 0x8(%rbp)
```

# Virtual Memory Accesses

```
%rip = 0x100000fad  
%rbp = 0x200
```

**Memory accesses:**  
Fetch instr. at addr **0x100000fad**




```
0x100000fad movl    0x8(%rbp), %edi  
0x100000fb0 addl    $0x2, %edi  
0x100000fb3 movl    %edi, 0x8(%rbp)
```

# Virtual Memory Accesses

```
%rip = 0x100000fad  
%rbp = 0x200
```

## Memory accesses:

Fetch instr. at addr **0x100000fad**  
Exec, load from addr **0x208**



```
0x100000fad movl    0x8(%rbp), %edi  
0x100000fb0 addl    $0x2, %edi  
0x100000fb3 movl    %edi, 0x8(%rbp)
```




# Virtual Memory Accesses

```
%rip = 0x100000fb0  
%rbp = 0x200
```

## Memory accesses:


Fetch instr. at addr **0x100000fad**  
Exec, load from addr **0x208**



```
0x100000fad movl    0x8(%rbp), %edi  
0x100000fb0 addl    $0x2, %edi  
0x100000fb3 movl    %edi, 0x8(%rbp)
```

# Virtual Memory Accesses

```
%rip = 0x100000fb0  
%rbp = 0x200
```



```
0x100000fad movl    0x8(%rbp), %edi  
0x100000fb0 addl    $0x2, %edi  
0x100000fb3 movl    %edi, 0x8(%rbp)
```


## Memory accesses:

Fetch instr. at addr **0x100000fad**  
Exec, load from addr **0x208**

Fetch instr. at addr **0x100000fb0**

# Virtual Memory Accesses

```
%rip = 0x100000fb0  
%rbp = 0x200
```



```
0x100000fad movl    0x8(%rbp), %edi  
0x100000fb0 addl    $0x2, %edi  
0x100000fb3 movl    %edi, 0x8(%rbp)
```


## Memory accesses:

Fetch instr. at addr **0x100000fad**  
Exec, load from addr **0x208**

Fetch instr. at addr **0x100000fb0**  
Exec, no load

# Virtual Memory Accesses

```
%rip = 0x100000fb3  
%rbp = 0x200
```



```
0x100000fad movl    0x8(%rbp), %edi  
0x100000fb0 addl    $0x2, %edi  
0x100000fb3 movl    %edi, 0x8(%rbp)
```


## Memory accesses:

Fetch instr. at addr **0x100000fad**  
Exec, load from addr **0x208**

Fetch instr. at addr **0x100000fb0**  
Exec, no load

# Virtual Memory Accesses

```
%rip = 0x100000fb3  
%rbp = 0x200
```



```
0x100000fad movl    0x8(%rbp), %edi  
0x100000fb0 addl    $0x2, %edi  
0x100000fb3 movl    %edi, 0x8(%rbp)
```

## Memory accesses:


Fetch instr. at addr **0x100000fad**  
Exec, load from addr **0x208**

Fetch instr. at addr **0x100000fb0**  
Exec, no load

Fetch instr. at addr **0x100000fb3**

# Virtual Memory Accesses

```
%rip = 0x100000fb3  
%rbp = 0x200
```



```
0x100000fad movl    0x8(%rbp), %edi  
0x100000fb0 addl    $0x2, %edi  
0x100000fb3 movl    %edi, 0x8(%rbp)
```

## Memory accesses:


Fetch instr. at addr **0x100000fad**  
Exec, **load** from addr **0x208**

Fetch instr. at addr **0x100000fb0**  
Exec, no load

Fetch instr. at addr **0x100000fb3**  
Exec, **store** to addr **0x208**

# Virtual Memory Accesses

```
%rip = 0x100000fb3  
%rbp = 0x200
```



```
0x100000fad movl    0x8(%rbp), %edi  
0x100000fb0 addl    $0x2, %edi  
0x100000fb3 movl    %edi, 0x8(%rbp)
```

## Memory accesses:

Fetch instr. at addr **0x100000fad**  
Exec, **load** from addr **0x208**

Fetch instr. at addr **0x100000fb0**  
Exec, no load

Fetch instr. at addr **0x100000fb3**  
Exec, **store** to addr **0x208**

**How to relocate the memory access in a way that is transparent to the process?**

# How to Run Multiple Programs?

- Approaches:
  - **Static relocation**
  - Dynamic relocation
  - Segmentation



# Static Relocation

- Idea: **rewrite** each program before loading it into memory as a process
- Each rewrite uses **different** addresses and pointers
- Change jumps, loads, etc.
- Q: Can any addresses be unchanged?

# Rewrite for Each New Process

```
0x100000fad movl 0x8(%rbp), %edi  
0x100000fb0 addl $0x2, %edi  
0x100000fb3 movl %edi, 0x8(%rbp)
```

rewrite

```
0x100010fad movl 0x8(%rbp), %edi  
0x100010fb0 addl $0x2, %edi  
0x100010fb3 movl %edi, 0x8(%rbp)
```

rewrite

```
0x100020fad movl 0x8(%rbp), %edi  
0x100020fb0 addl $0x2, %edi  
0x100020fb3 movl %edi, 0x8(%rbp)
```

# Rewrite for Each New Process

```
0x100000fad movl 0x8(%rbp), %edi  
0x100000fb0 addl $0x2, %edi  
0x100000fb3 movl %edi, 0x8(%rbp)
```

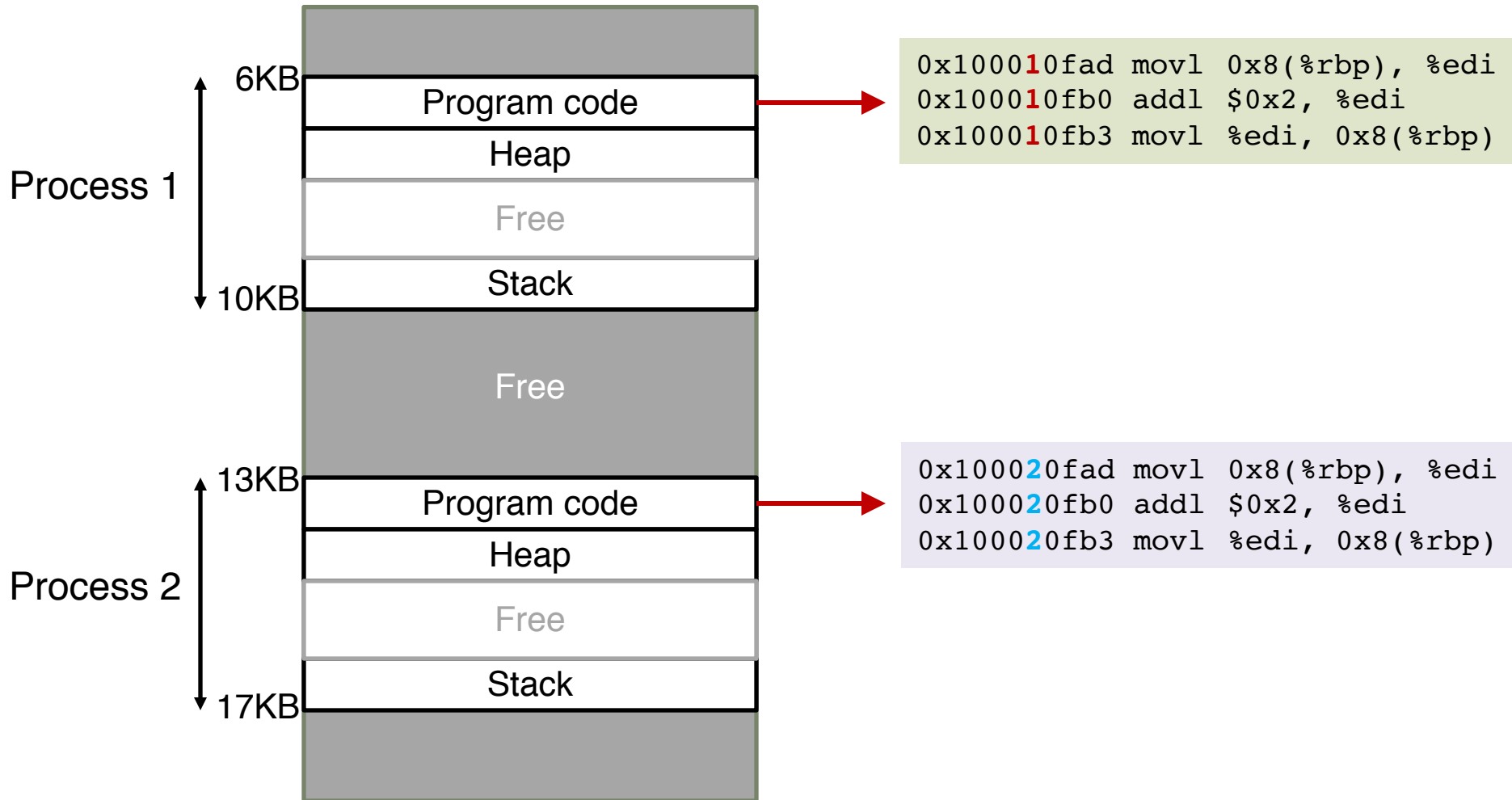
rewrite

```
0x100010fad movl 0x8(%rbp), %edi  
0x100010fb0 addl $0x2, %edi  
0x100010fb3 movl %edi, 0x8(%rbp)
```

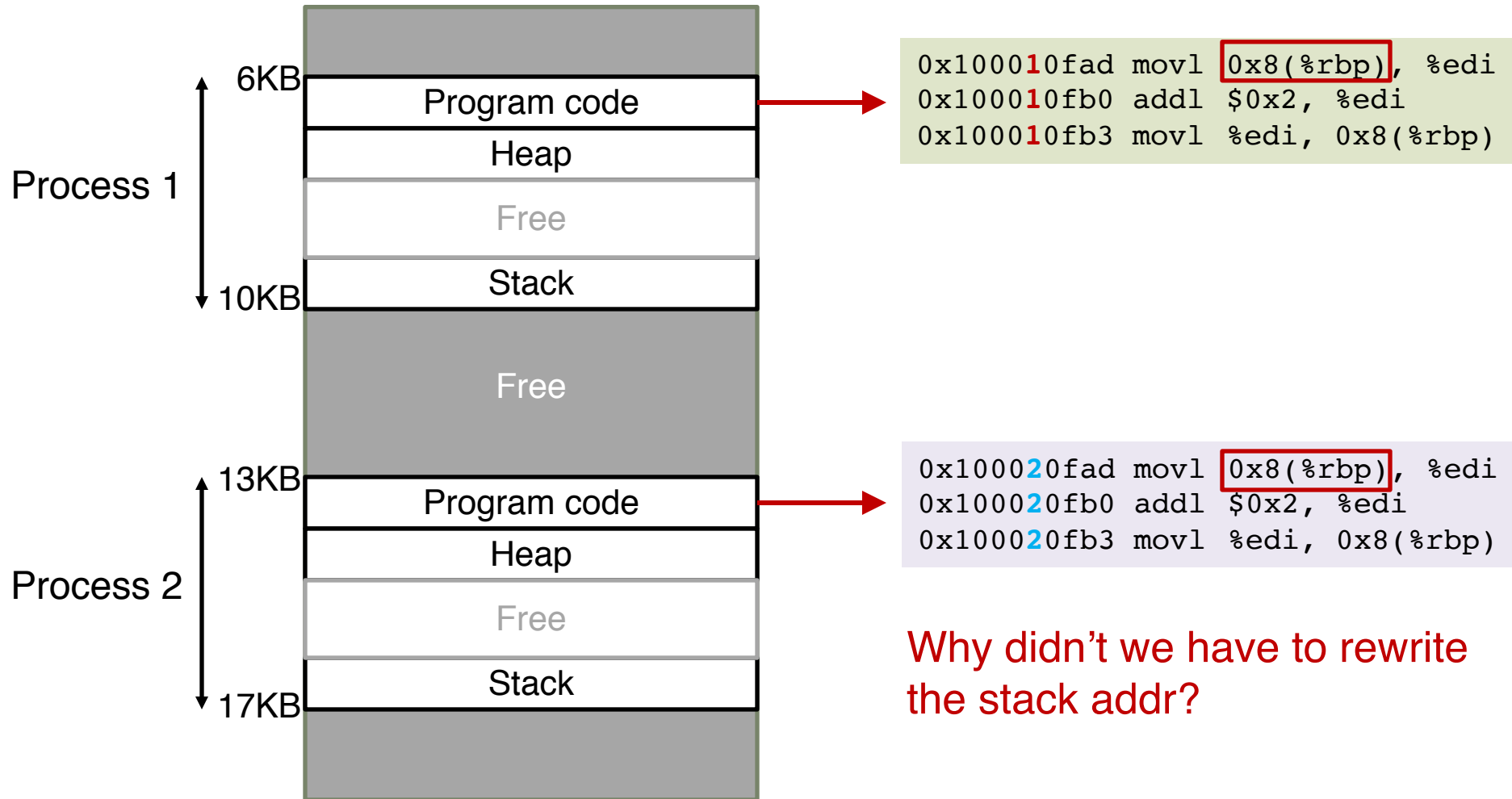
rewrite

```
0x100020fad movl 0x8(%rbp), %edi  
0x100020fb0 addl $0x2, %edi  
0x100020fb3 movl %edi, 0x8(%rbp)
```

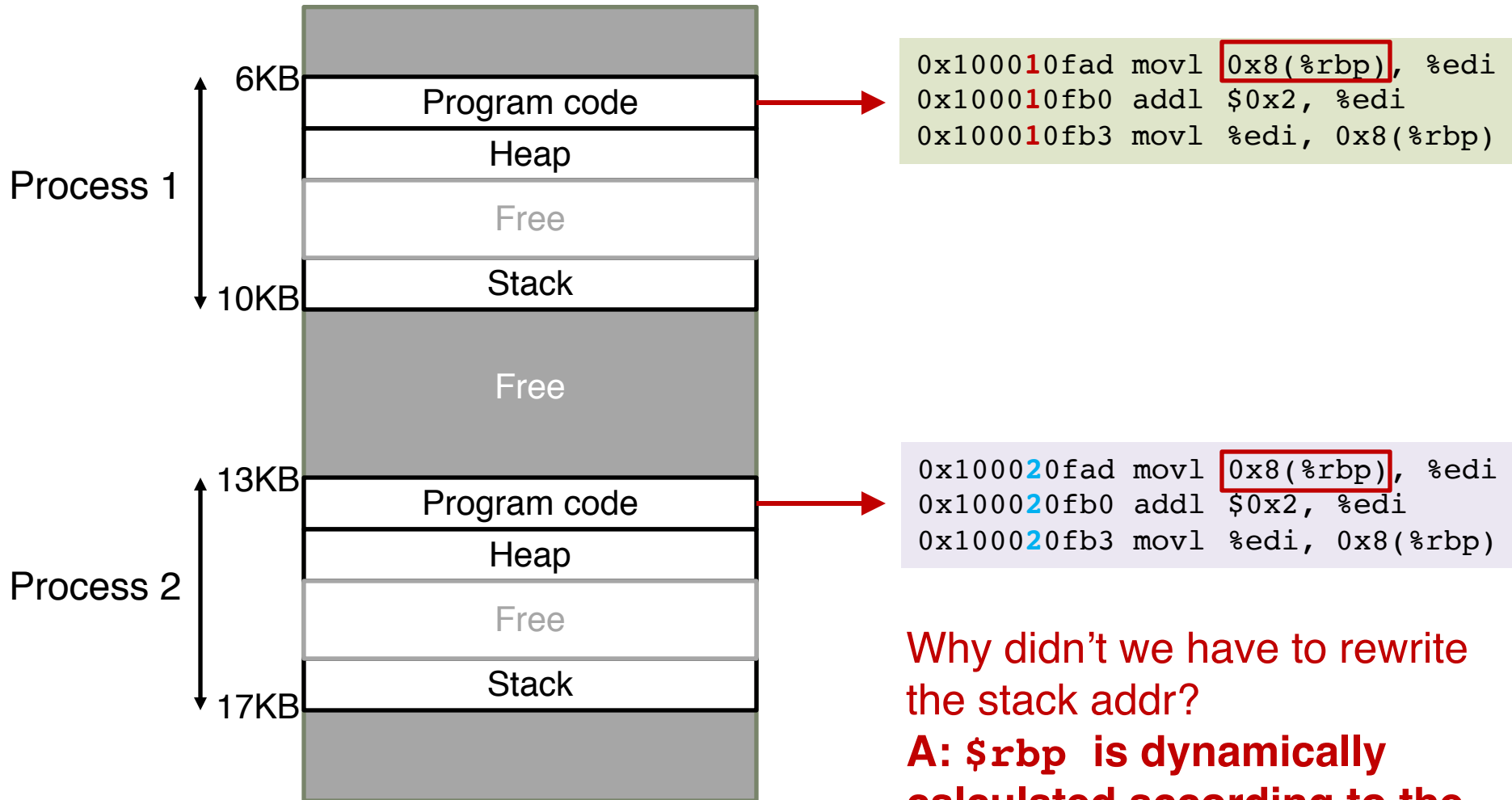
# Rewrite for Each New Process



# Rewrite for Each New Process



# Rewrite for Each New Process



Why didn't we have to rewrite the stack addr?

**A:  $\$rbp$  is dynamically calculated according to the base addr**

# How to Run Multiple Programs?

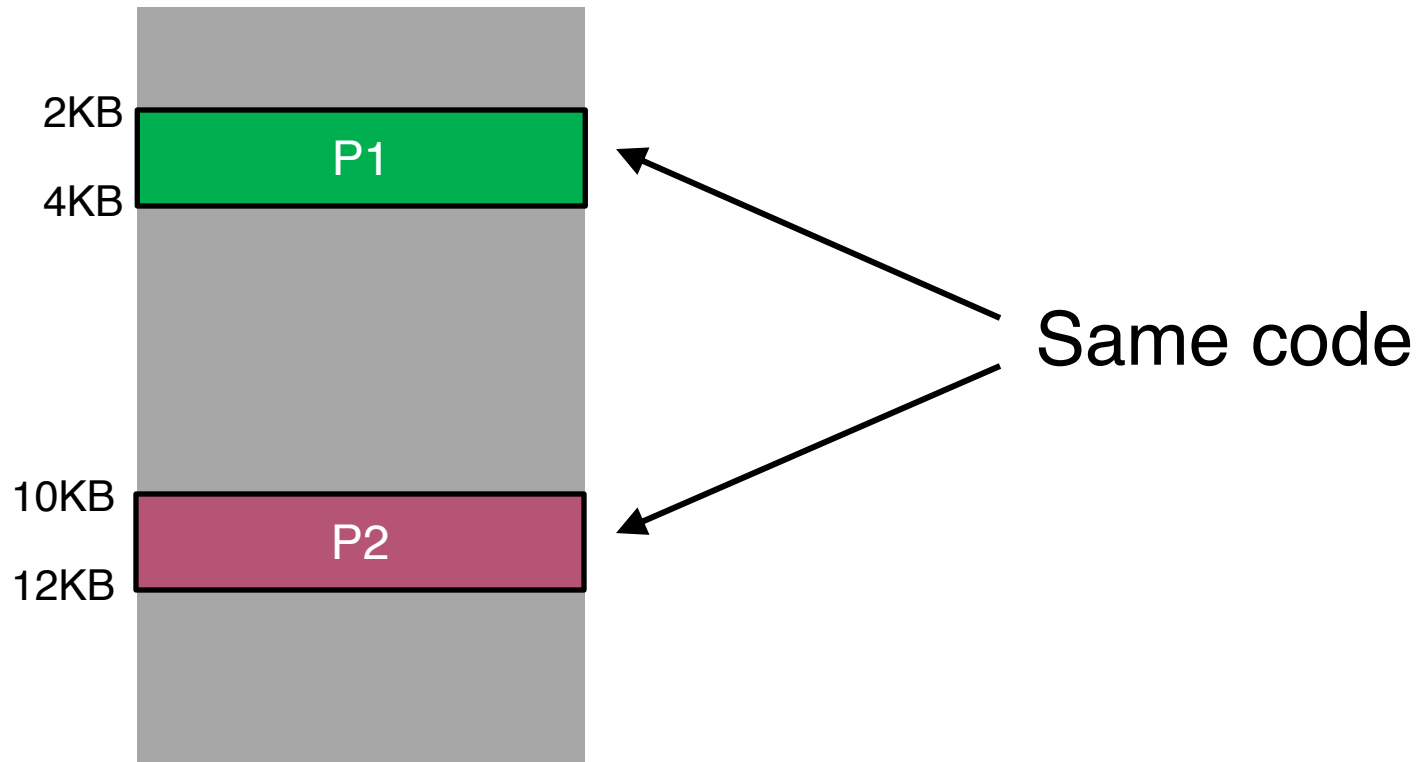
- Approaches:
  - Static relocation
  - **Dynamic relocation**
    - Base
    - Base-and-Bounds
  - Segmentation

# Base

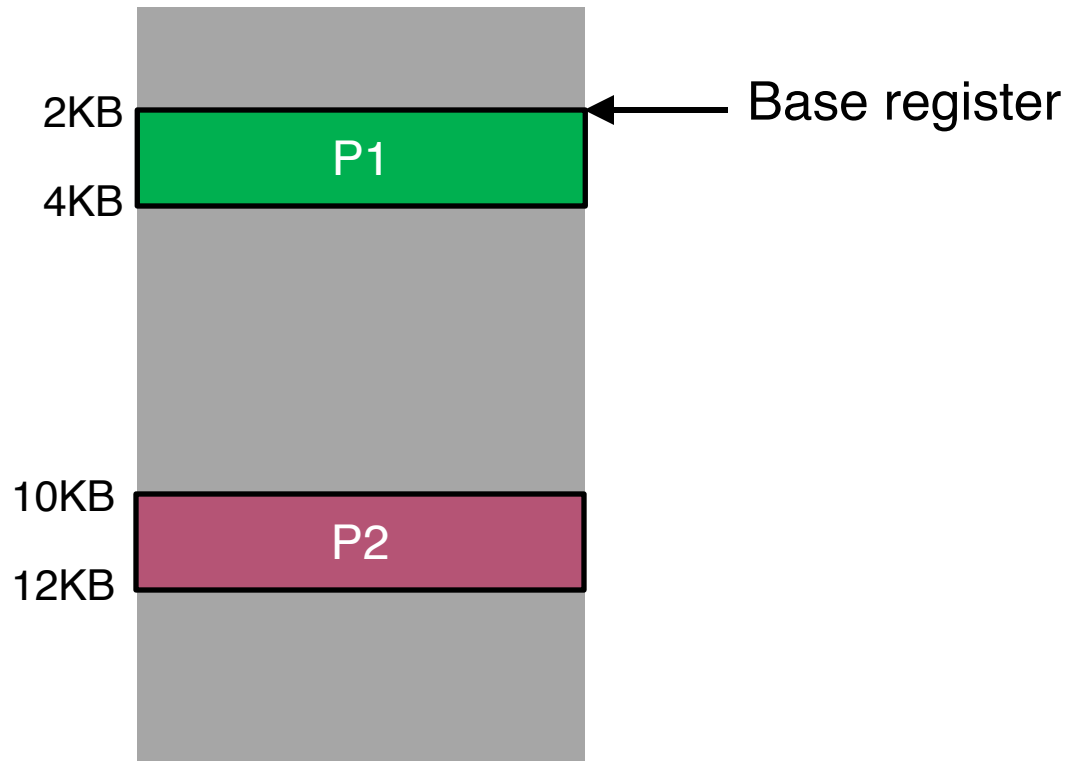
- Idea: **translate** virtual address to physical by adding an offset each time
- Store base addr in a **base** register
- Each process has a **different** value in the base register when running



# Base Relocation

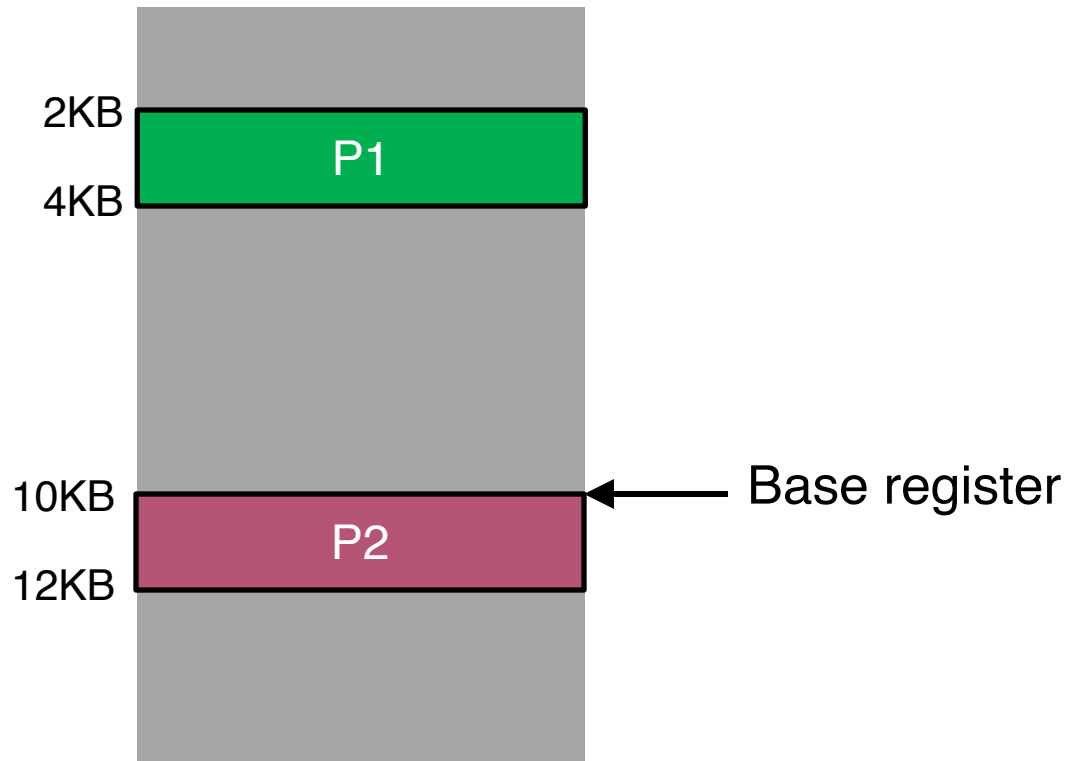


# Base Relocation



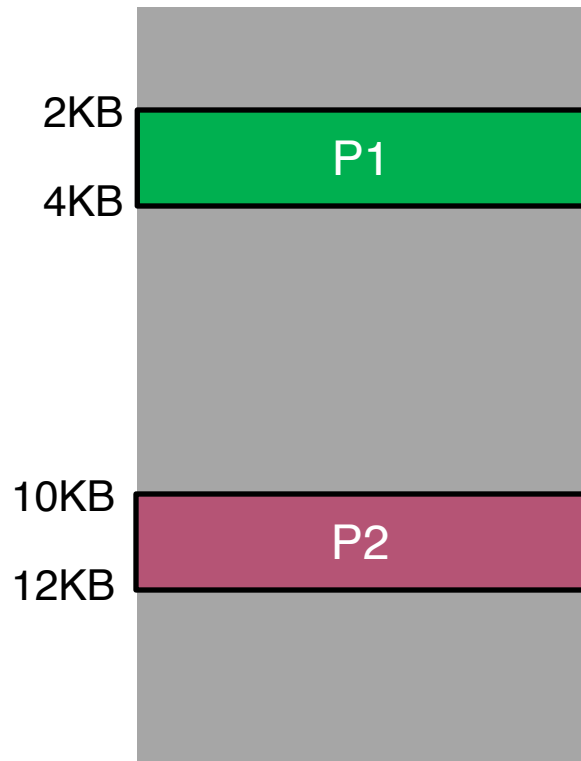
P1 is running ...

# Base Relocation



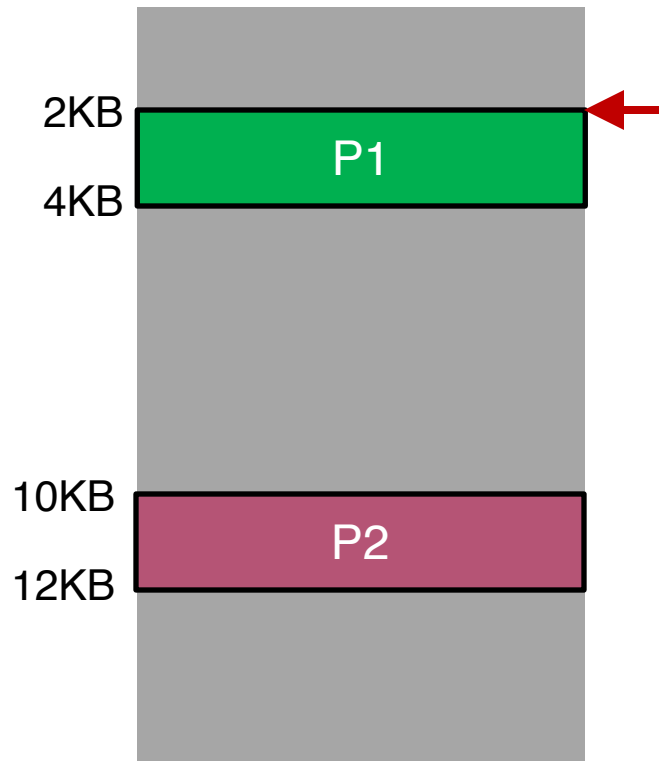
P2 is running ...

# Base Relocation



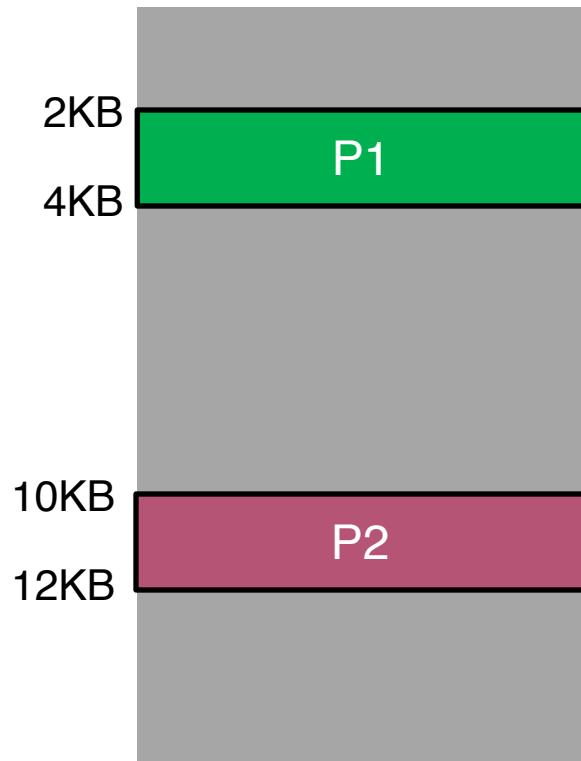
Virtual	Physical
P1: load 100, R1	

# Base Relocation



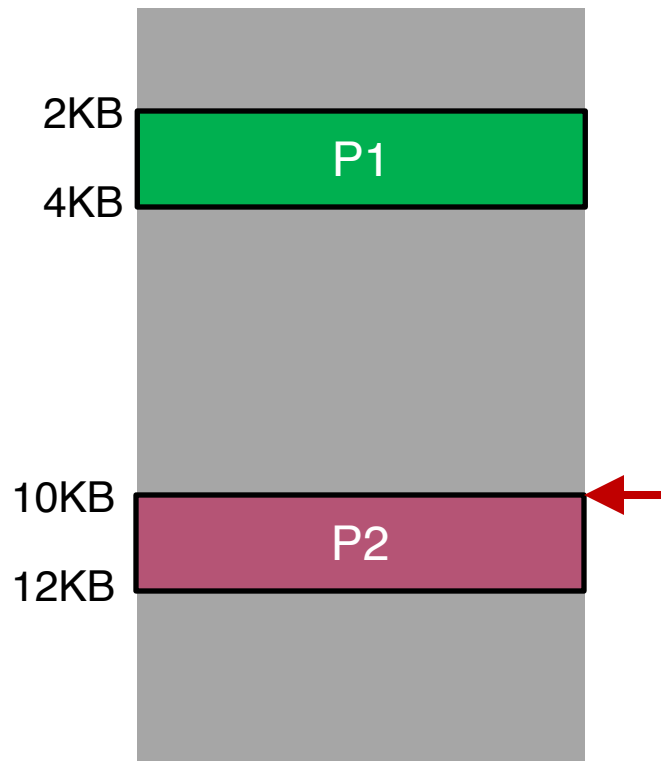
Virtual	Physical
P1: load 100, R1	load 2148, R1

# Base Relocation



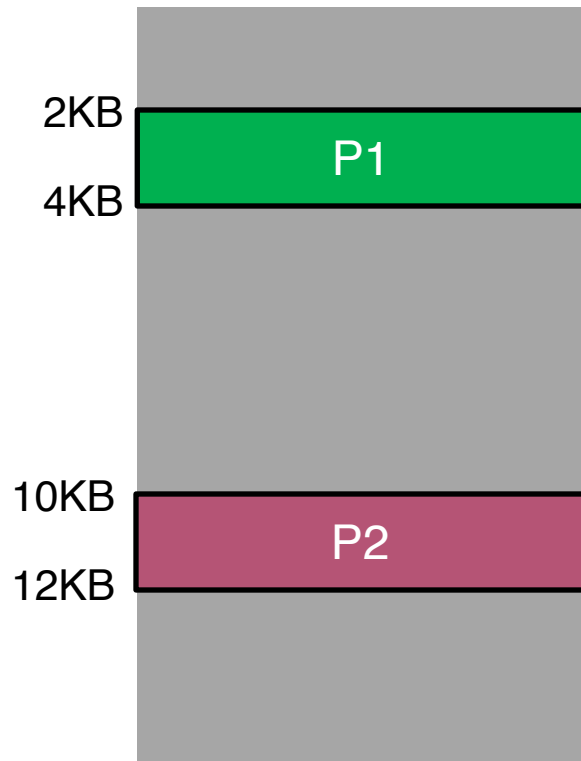
Virtual	Physical
P1: load 100, R1	load 2148, R1
P2: load 100, R1	

# Base Relocation



Virtual	Physical
P1: load 100, R1	load 2148, R1
P2: load 100, R1	load 10340, R1

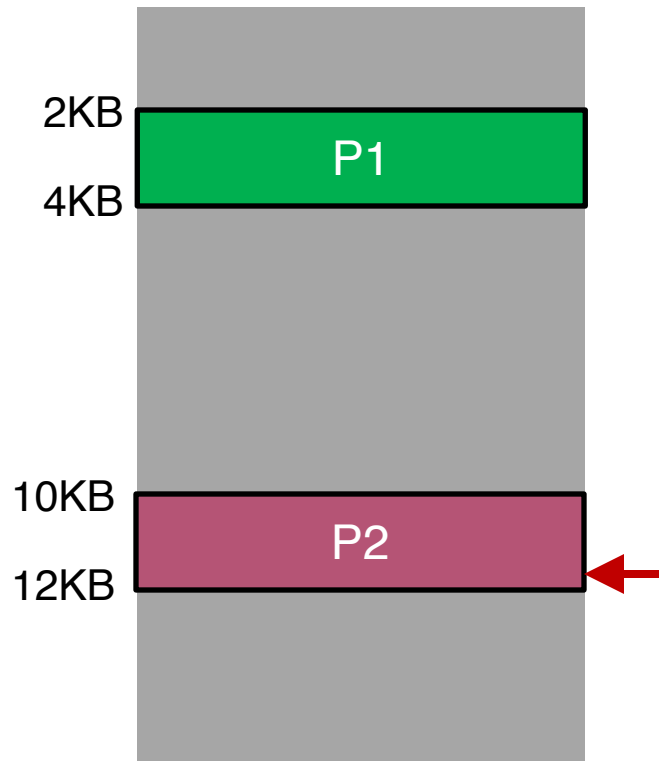
# Base Relocation



Virtual	Physical
P1: load 100, R1	load 2148, R1
P2: load 100, R1	load 10340, R1
P2: load 2000, R1	

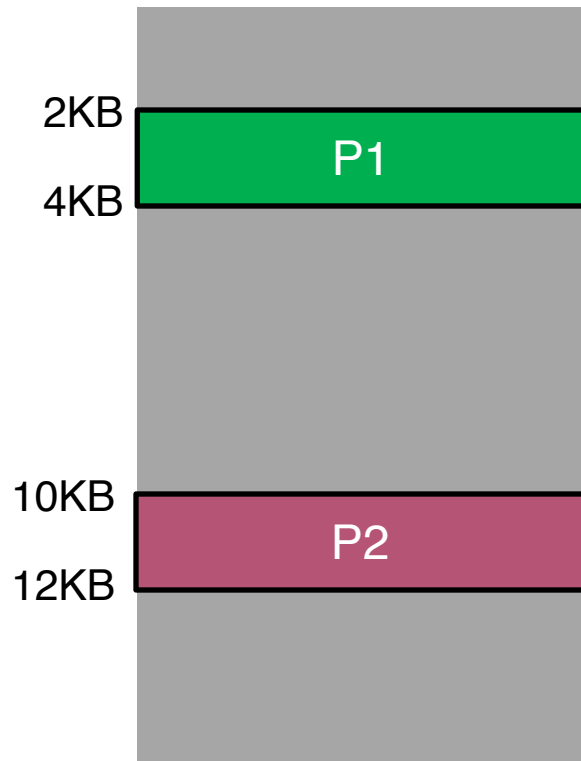


# Base Relocation



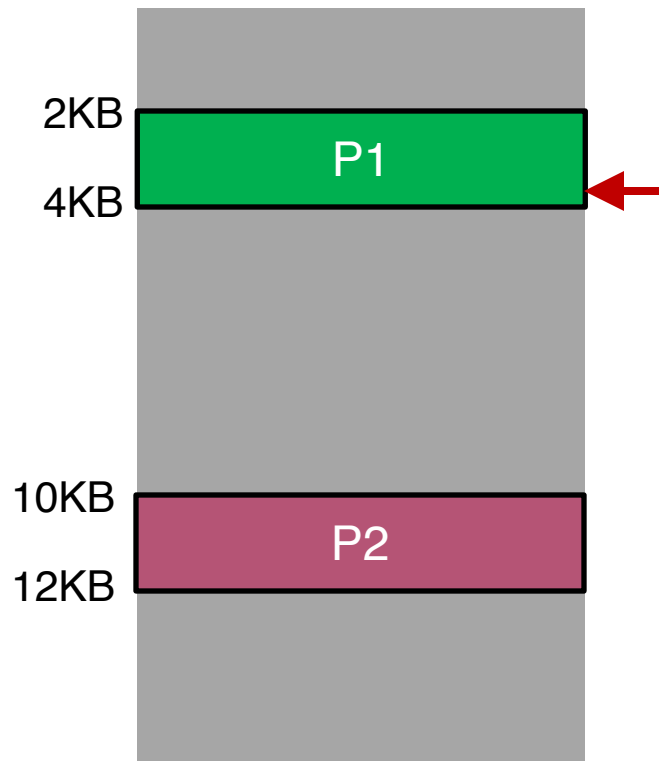
Virtual	Physical
P1: load 100, R1	load 2148, R1
P2: load 100, R1	load 10340, R1
P2: load 2000, R1	load 12240, R1

# Base Relocation



Virtual	Physical
P1: load 100, R1	load 2148, R1
P2: load 100, R1	load 10340, R1
P2: load 2000, R1	load 12240, R1
P1: load 2000, R1	

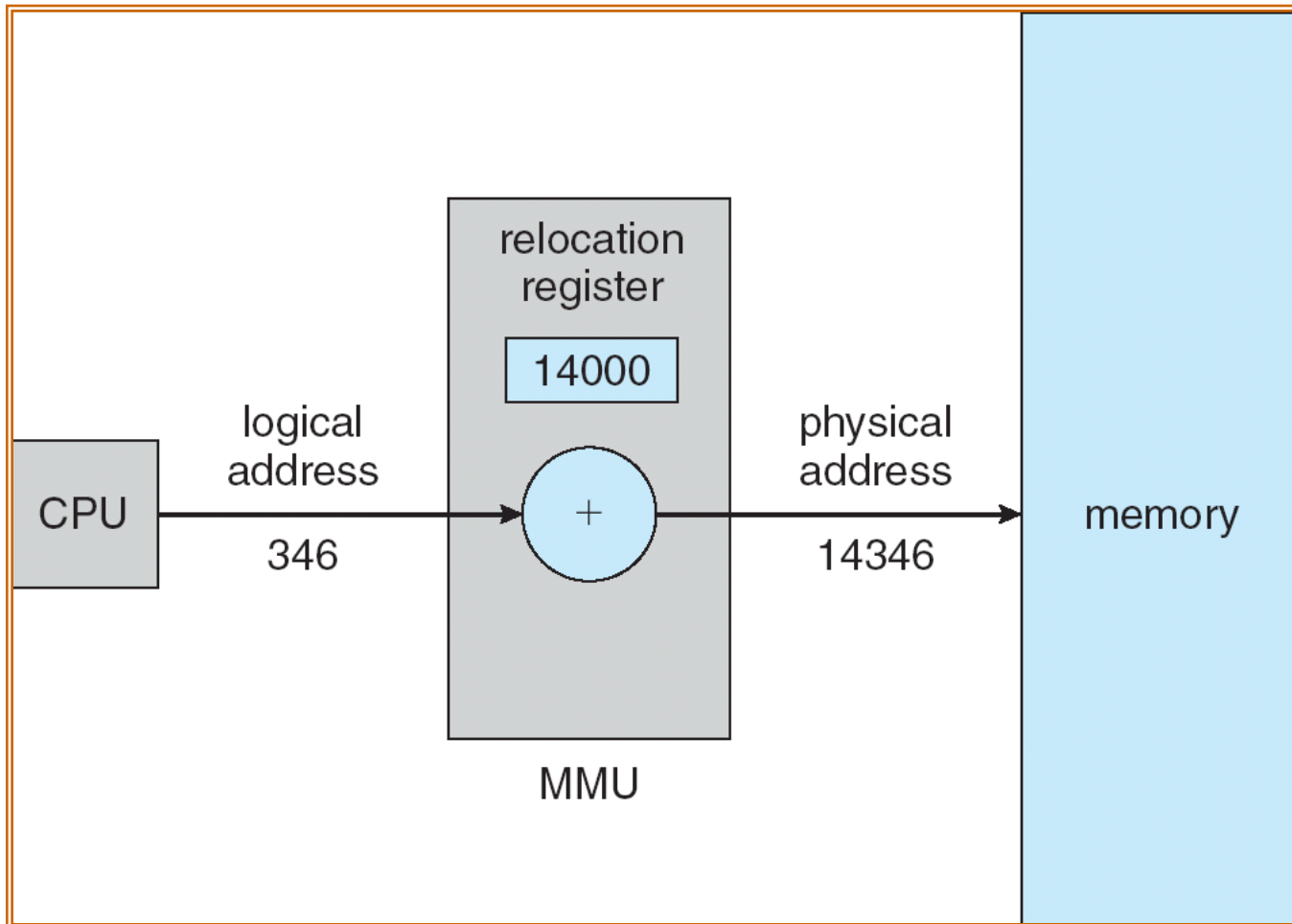
# Base Relocation



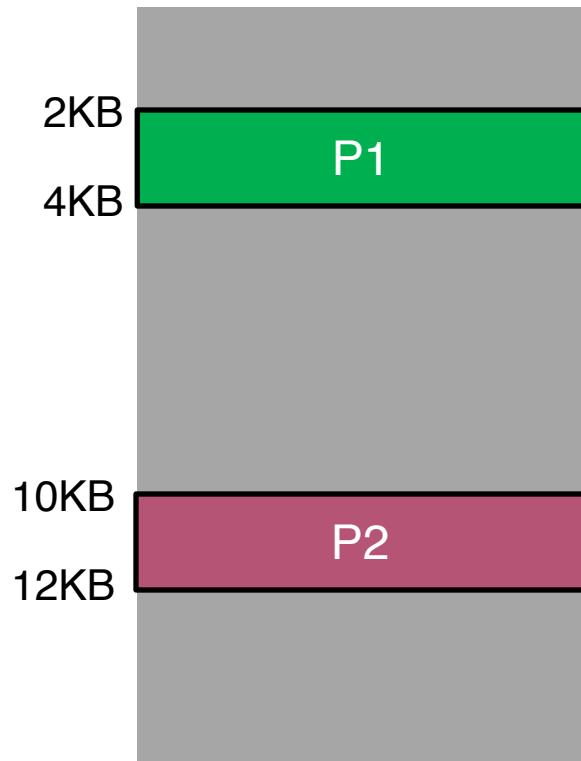
Virtual	Physical
P1: load 100, R1	load 2148, R1
P2: load 100, R1	load 10340, R1
P2: load 2000, R1	load 12240, R1
P1: load 2000, R1	load 4048, R1

# Worksheet

# Base Relocation Hardware



# Base Relocation

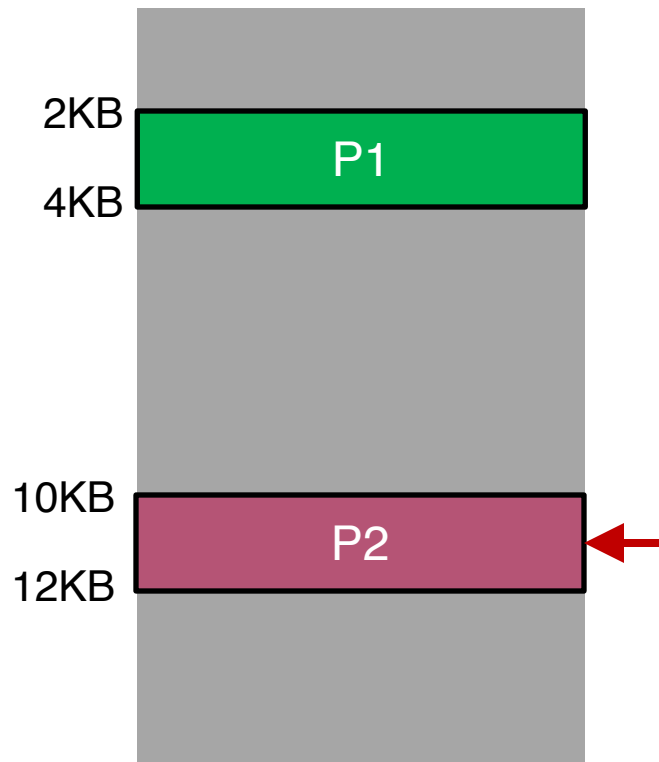


Virtual	Physical
P1: load 100, R1	load 2148, R1
P2: load 100, R1	load 10340, R1
P2: load 2000, R1	load 12240, R1
P1: load 2000, R1	load 4048, R1

**Can P1 hurt P2?**

**Can P2 hurt P1?**

# Base Relocation



Virtual	Physical
P1: load 100, R1	load 2148, R1
P2: load 100, R1	load 10340, R1
P2: load 2000, R1	load 12240, R1
P1: load 2000, R1	load 4048, R1
<b>P1: store 9241, R1</b>	<b>store 11289, R1</b>

**Can P1 hurt P2?**

**Can P2 hurt P1?**

**Overflow!**

# How to Run Multiple Programs?

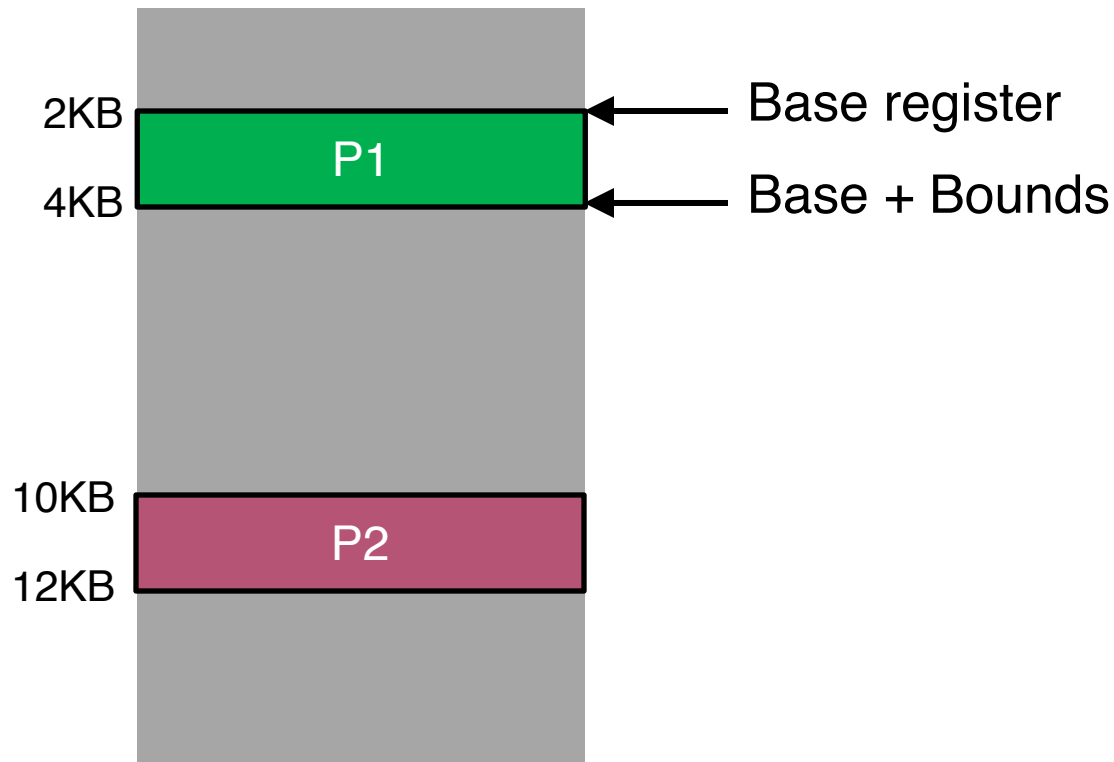
- Approaches:
  - Static relocation
  - **Dynamic relocation**
    - Base
    - **Base-and-Bounds**
  - Segmentation



# Base-and-Bounds

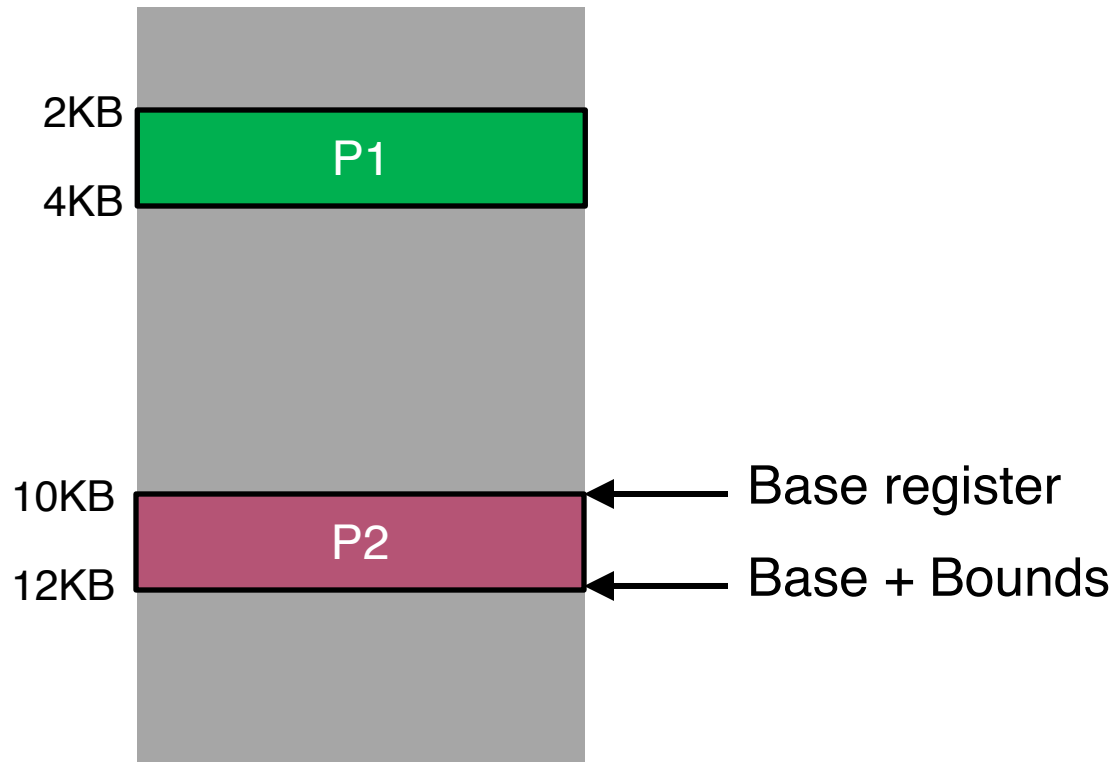
- Idea: add bound register to avoid “overflow”
- Two CPU registers
  - Base register
  - Bounds register (or limit register)  
$$\text{physical addr} = \text{virtual addr} + \text{base}$$
- The base-and-bounds hardware referred to as **Memory Management Unit** (MMU)
- Protection: The hardware provides special instructions to modify the base and bounds register
  - Allowing OS to change them when different processes run
  - **Privileged** (only in kernel mode)

# Base-and-Bounds



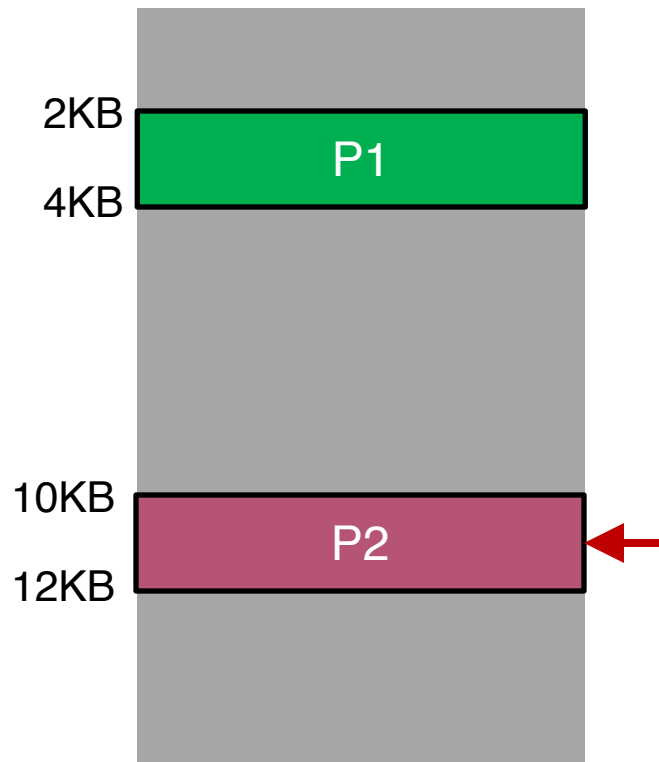
P1 is running ...

# Base-and-Bounds



P2 is running ...

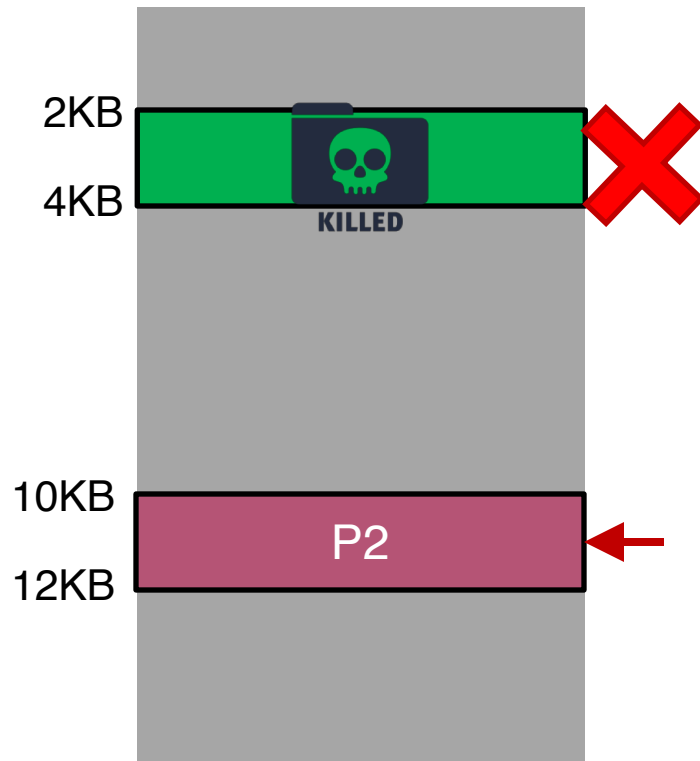
# Base-and-Bounds



Virtual	Physical
P1: load 100, R1	load 2148, R1
P2: load 100, R1	load 10340, R1
P2: load 2000, R1	load 12240, R1
P1: load 2000, R1	load 4048, R1
<b>P1: store 9241, R1</b>	

**Can P1 hurt P2?**

# Base-and-Bounds

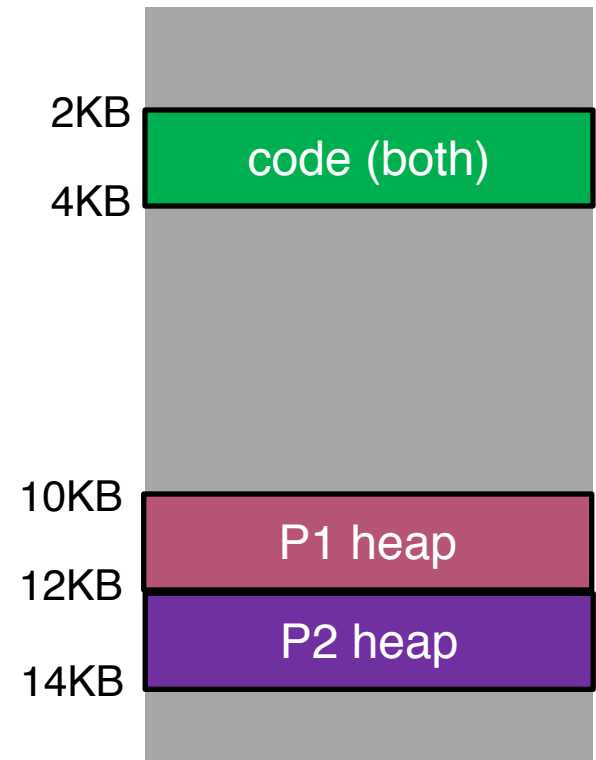


Virtual	Physical
P1: load 100, R1	load 2148, R1
P2: load 100, R1	load 10340, R1
P2: load 2000, R1	load 12240, R1
P1: load 2000, R1	load 4048, R1
<b>P1: store 9241, R1</b>	<b>Interrupt!</b>

**Can P1 hurt P2?**

# Code Sharing

- Idea: make base/bounds for the code of several processes point to the same physical mem
- Note: need careful protection!



# Base-and-Bounds Pros/Cons

- Pros?

- Fast + simple
- Little bookkeeping overhead (2 registers)

- Cons?

- Not flexible
- Wastes memory for large memory addresses

# Base-and-Bounds Pros/Cons

- Pros?

- Fast + simple
- Little bookkeeping overhead (2 registers)

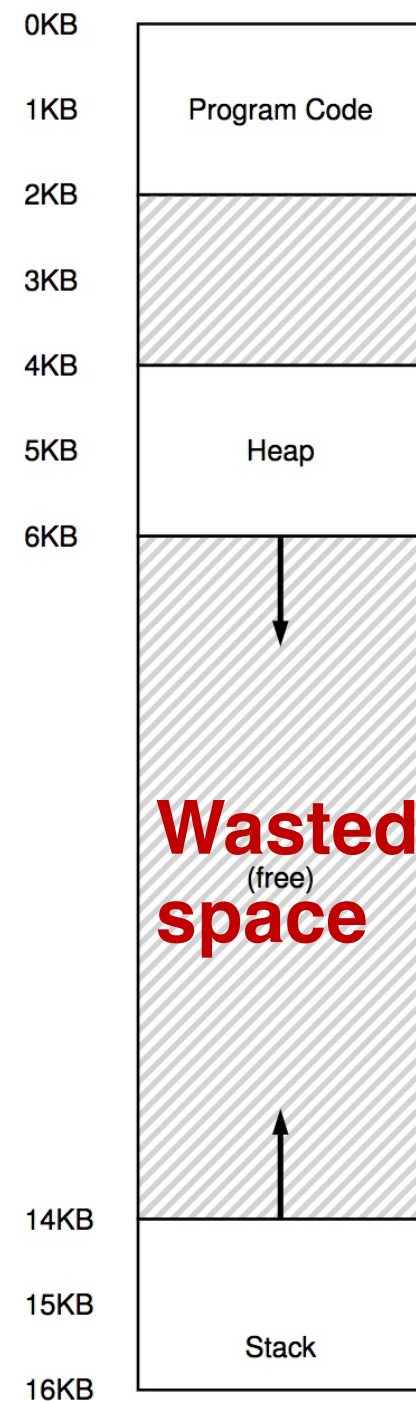
- Cons?

- Not flexible
- **Wastes memory** for large memory addresses



# Problems with Base-and-Bounds

- Simple base-and-bounds approach *wastes* a chunk of “free” space between stack and heap
- Impossible to run a program when its entire address space is greater than the memory capacity



# How to Run Multiple Programs?

- Approaches:
  - Static relocation
  - Dynamic relocation
    - Base
    - Base-and-Bounds
  - **Segmentation**

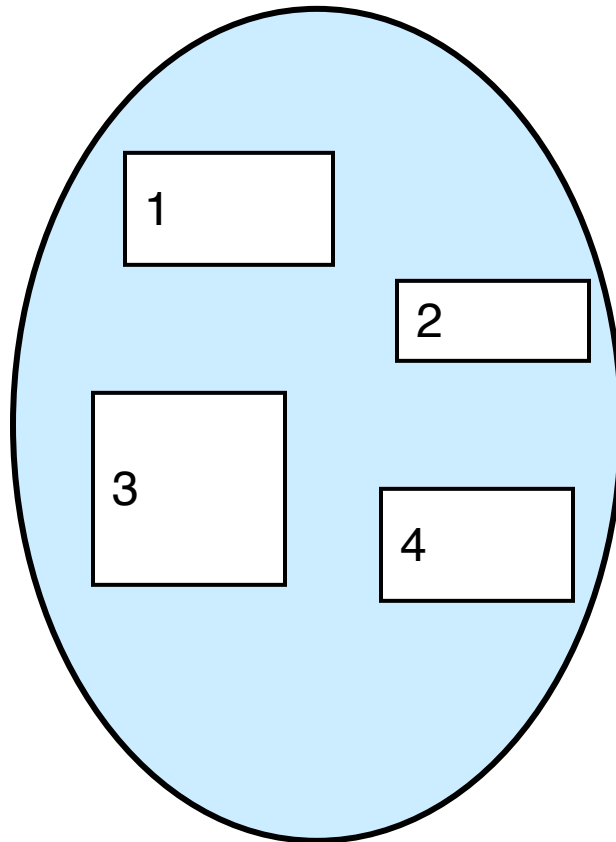
# Segmentation

- Idea: generalize base-and-bounds
- Each base+bounds pair is a **segment**
- Use **different segments** for heap and memory
  - Requires more registers
- **Resize segments** as needed

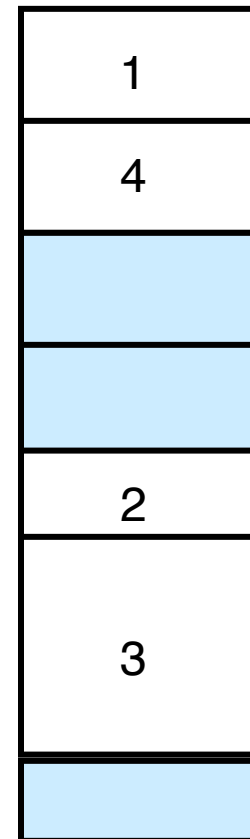
# Segmentation (cont.)

- A segment is a contiguous portion of the address space
- A program is a collection of segments
- A segment can be a **logical** unit:
  - E.g., *main program, procedure, function, object, local variables, global variables, common block, stack, heap, symbol table, or arrays, etc.*

# Logical View of Segmentation

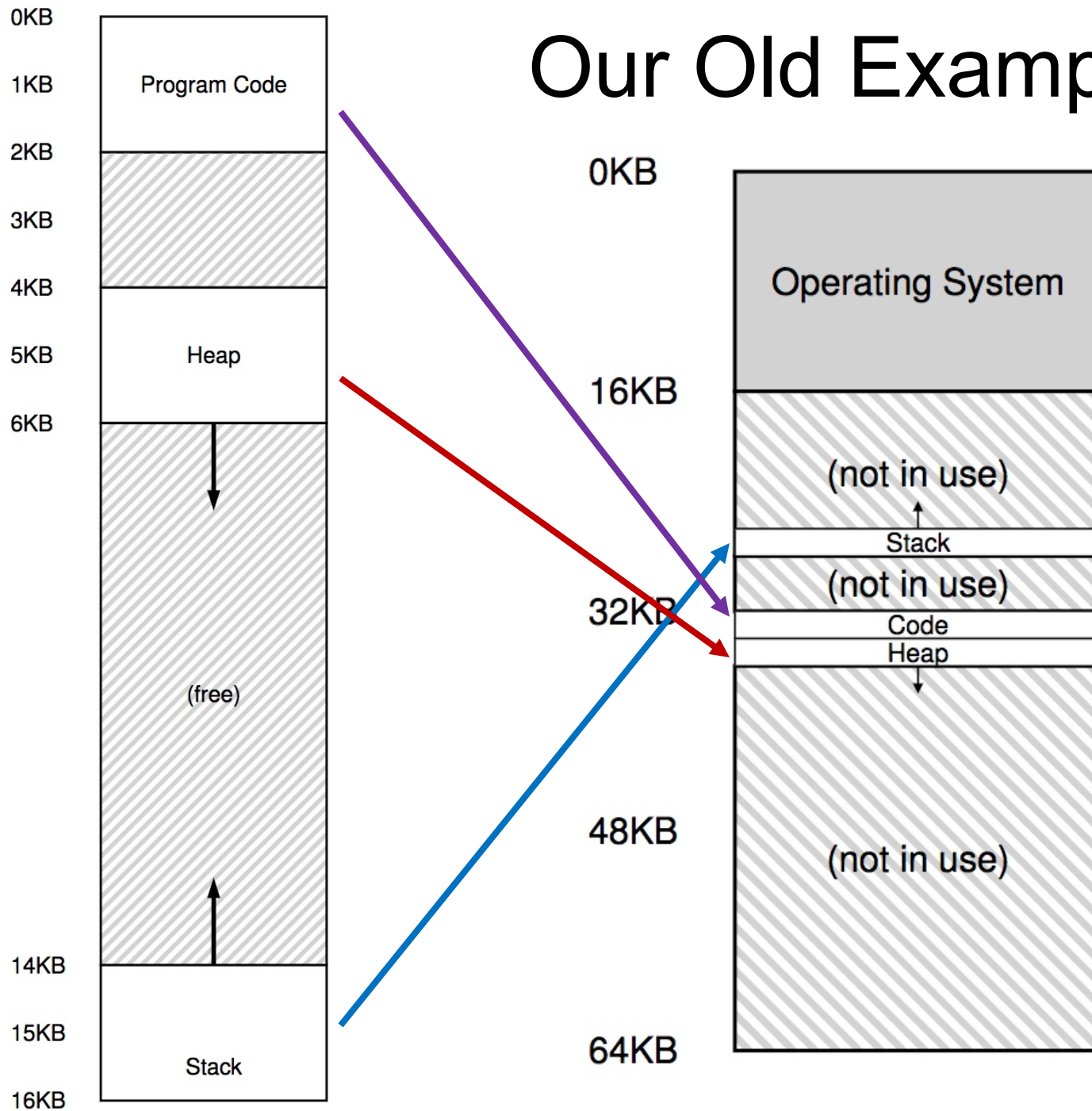


Virtual address space of a process

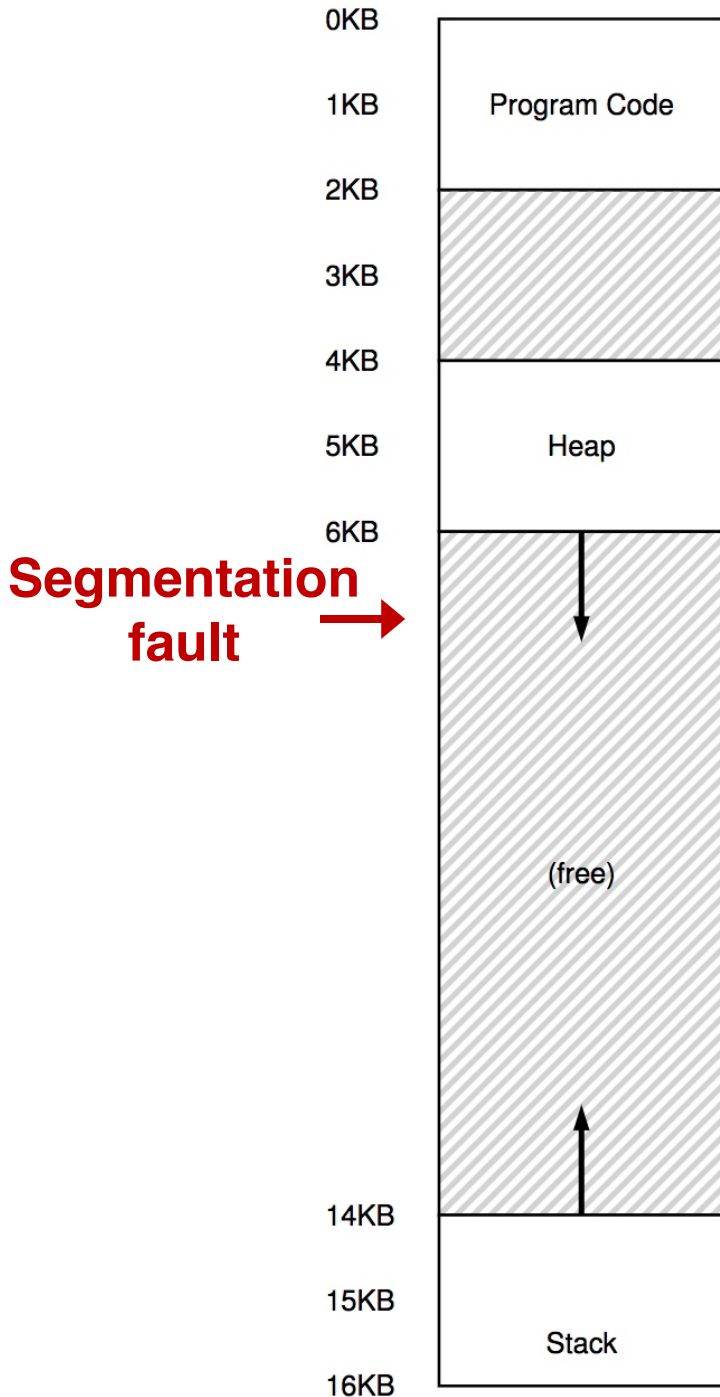


physical memory

# Our Old Example



# Segfault!



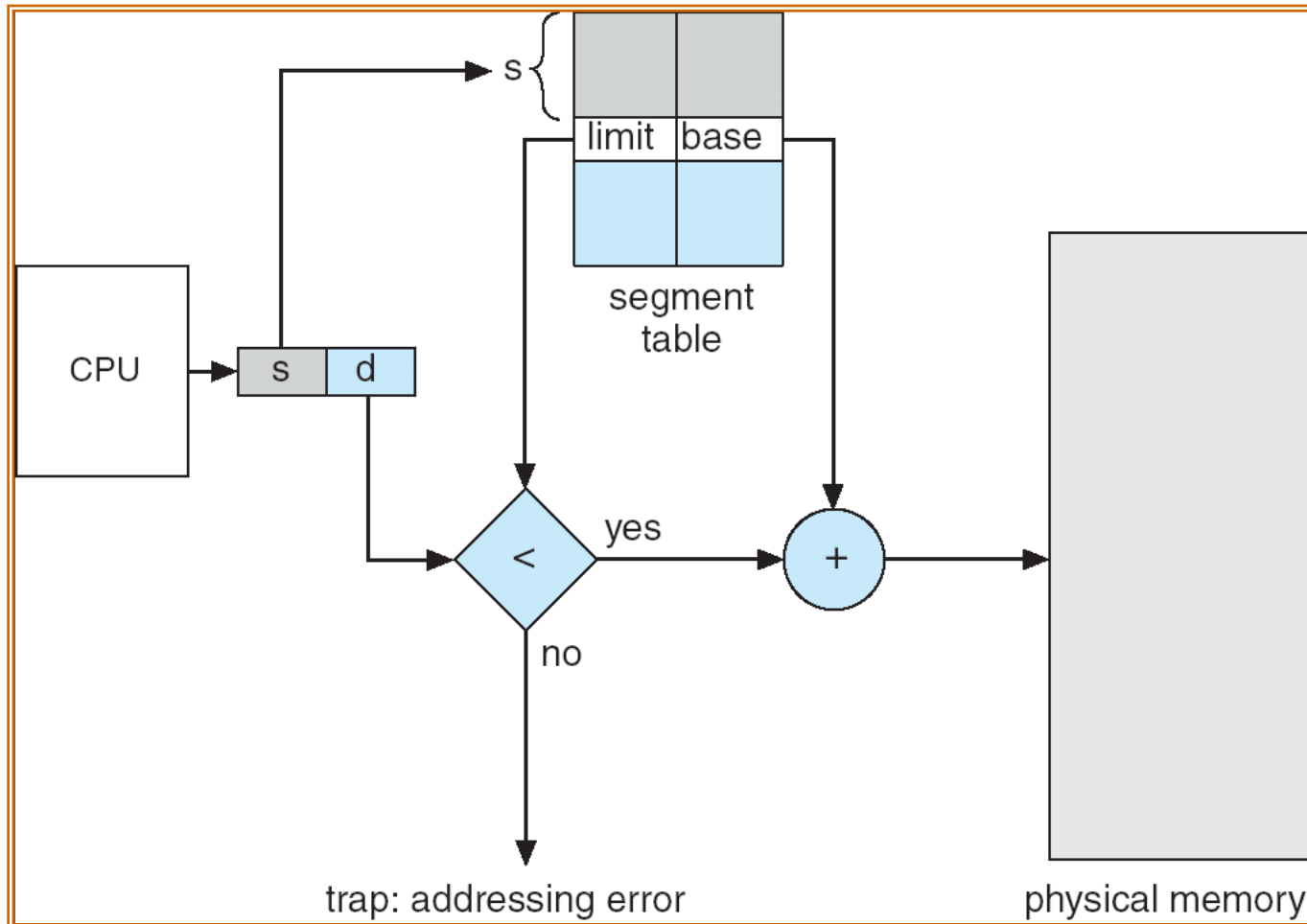
**Access to the address  
7KB ...**

# Segmentation Architecture

- Logical address consists of a pair:  
    <segment-number, offset>
- *Segment table* – maps two-dimensional physical addresses. Each table entry has:
  - *base* – contains the starting physical address where the segments reside in memory
  - *limit* – specifies the length of the segment (or bound)
- *Segment-table base register (STBR)* points to the segment table's location in memory
- *Segment-table length register (STLR)* indicates number of segments used by a process
  - segment number  $s$  is legal if  $s < \text{STLR}$



# Segmentation Hardware



# Example of Segmentation

