[537] Virtual Memory

Tyler Harter
9/15/14

Overview

Review Scheduling

Address Spaces (Chapter 13)

Address Translation (Chapter 15)

Segmentation (Chapter 16)

Review: Schedulers

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

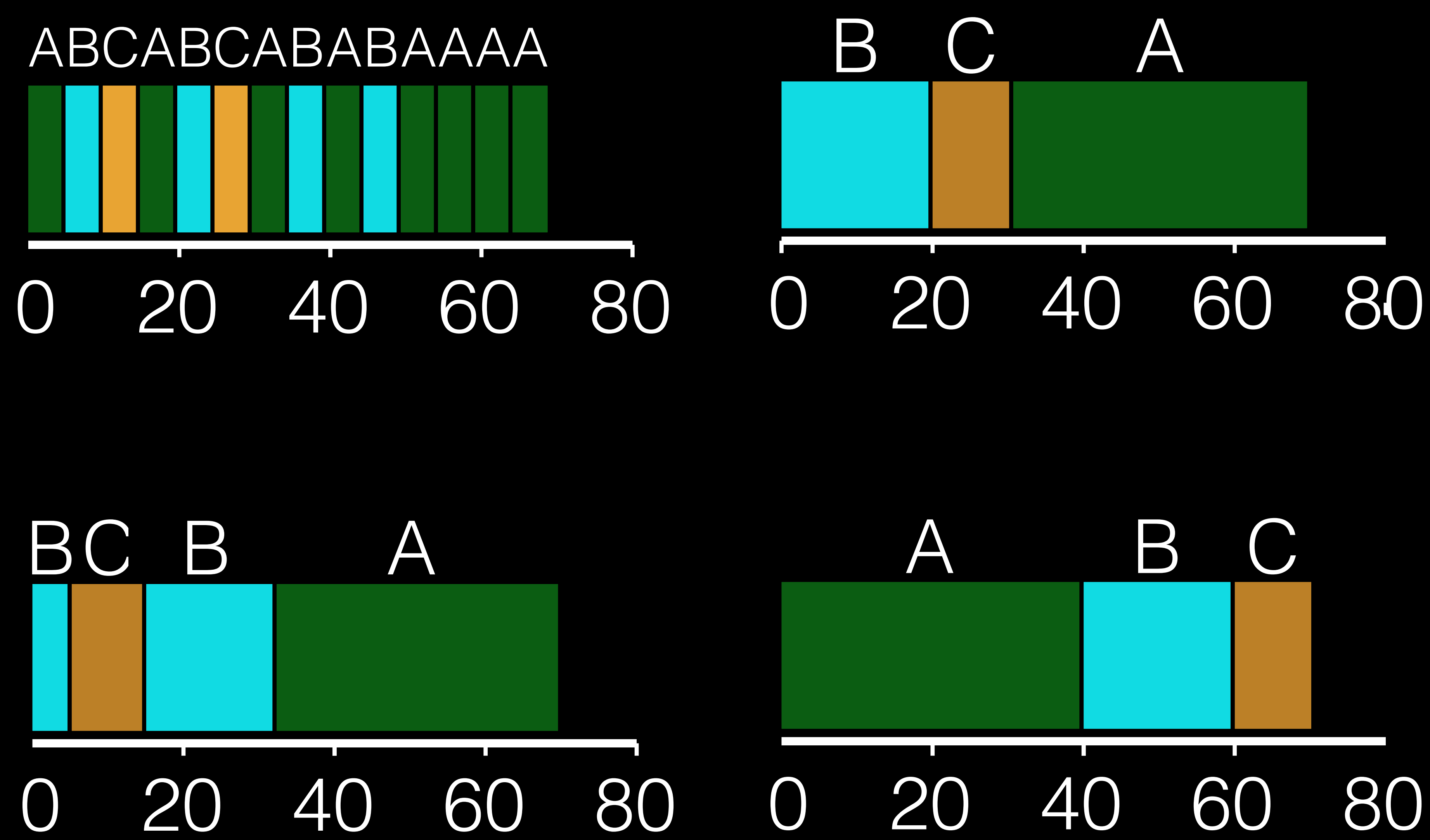
Metrics:
- turnaround_time
- response_time

Project grading will be based on turnaround time!

Workload

JOB	arrival	run
A	0	40
B	0	20
C	5	10

Timelines



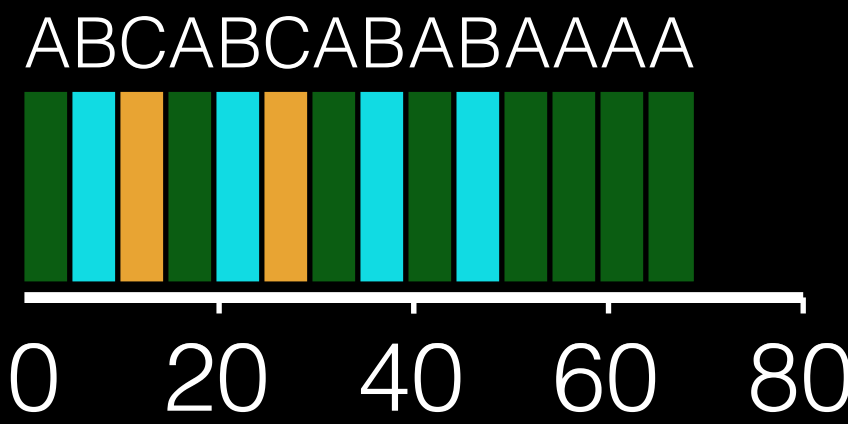
Schedulers:

FIFO
SJF
STCF
RR

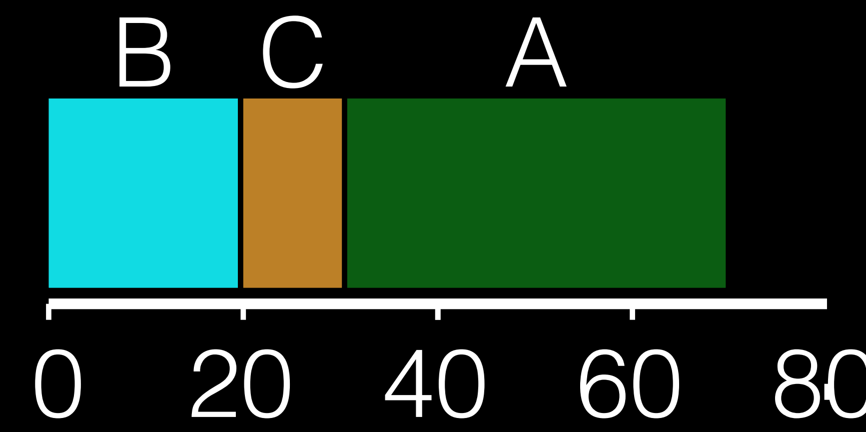
Workload

JOB	arrival	run
A	0	40
B	0	20
C	5	10

Timelines



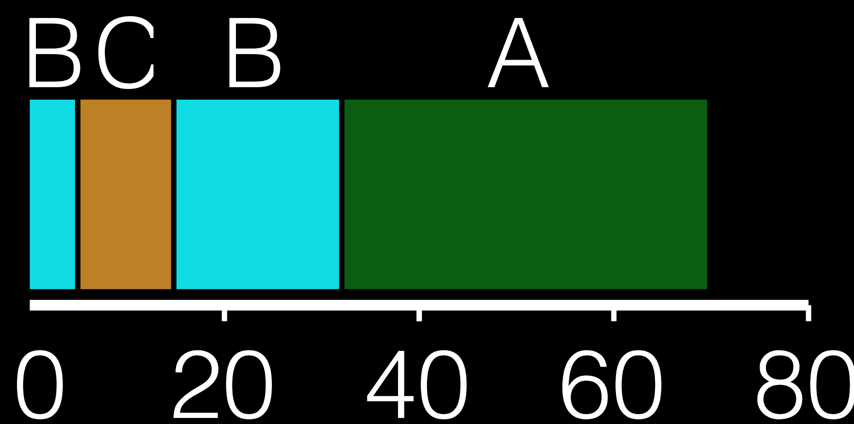
RR



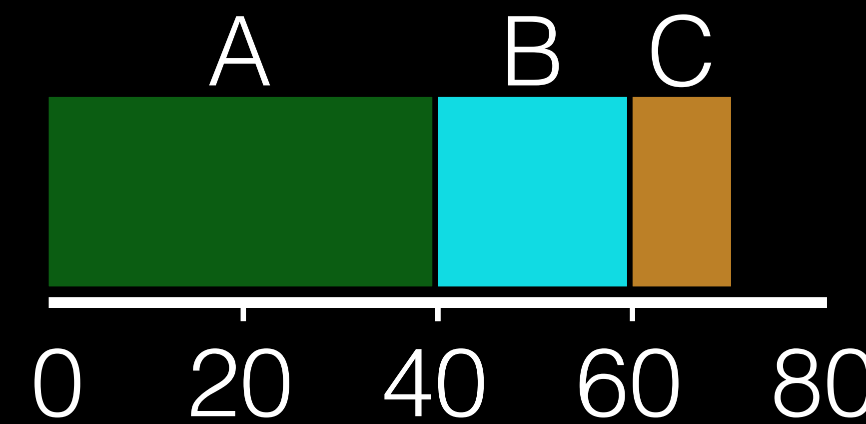
SJF

Schedulers:

- FIFO
- SJF
- STCF
- RR

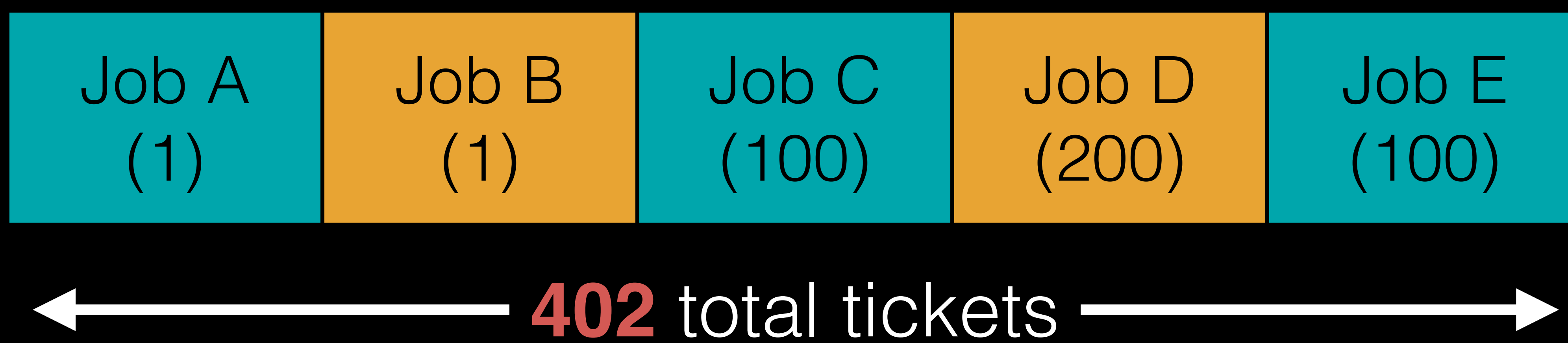


STCF

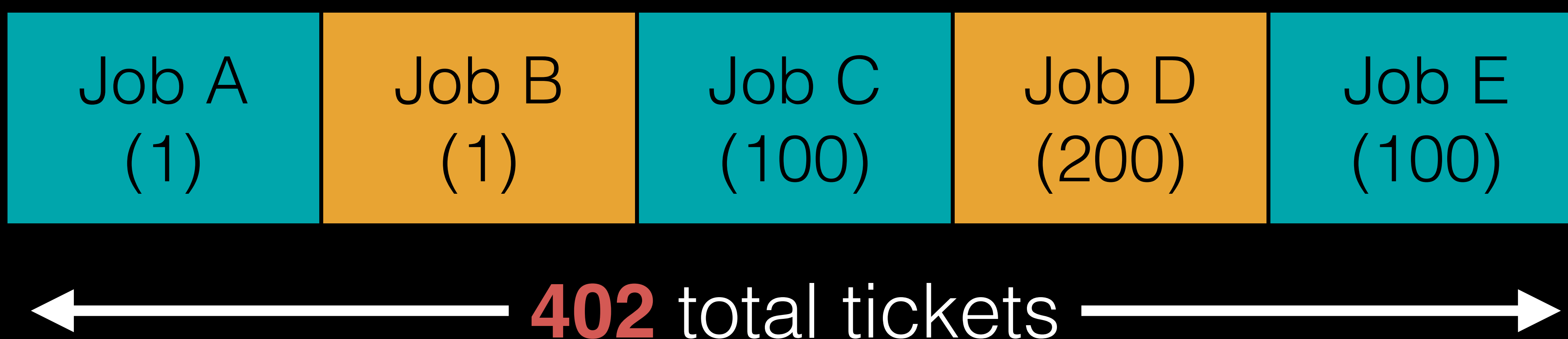


FIFO

Lottery Scheduler



winner = random(402)

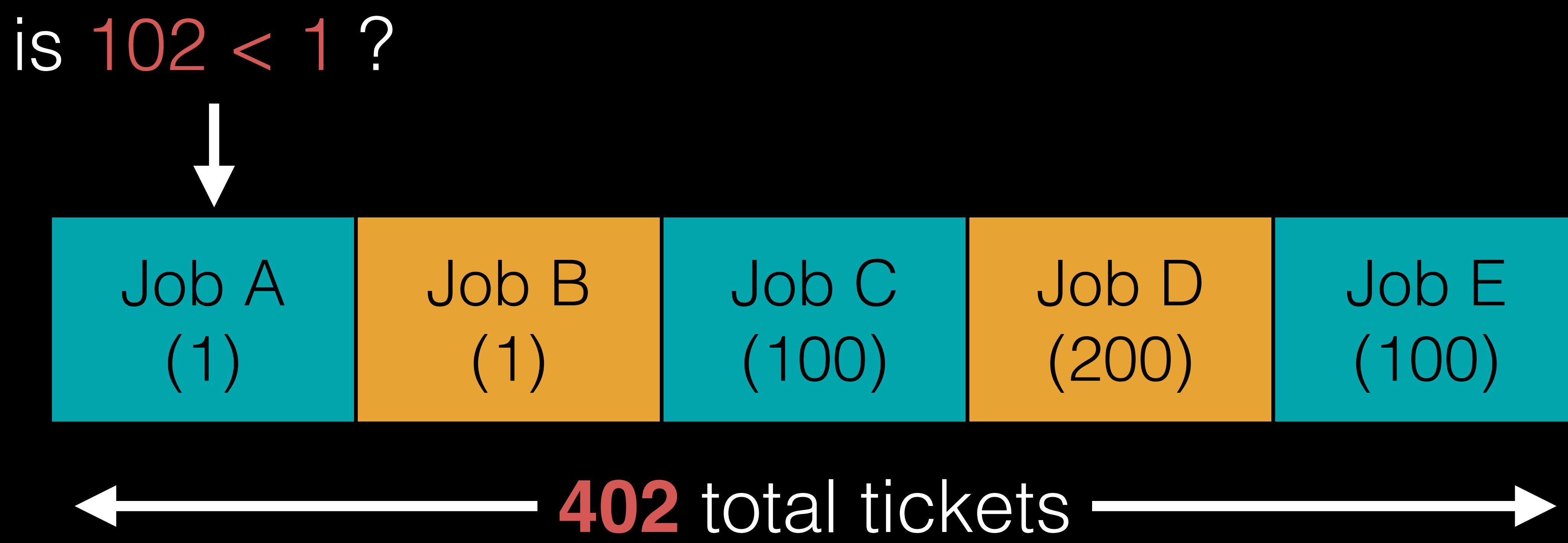


winner = 102

Job A (1)	Job B (1)	Job C (100)	Job D (200)	Job E (100)

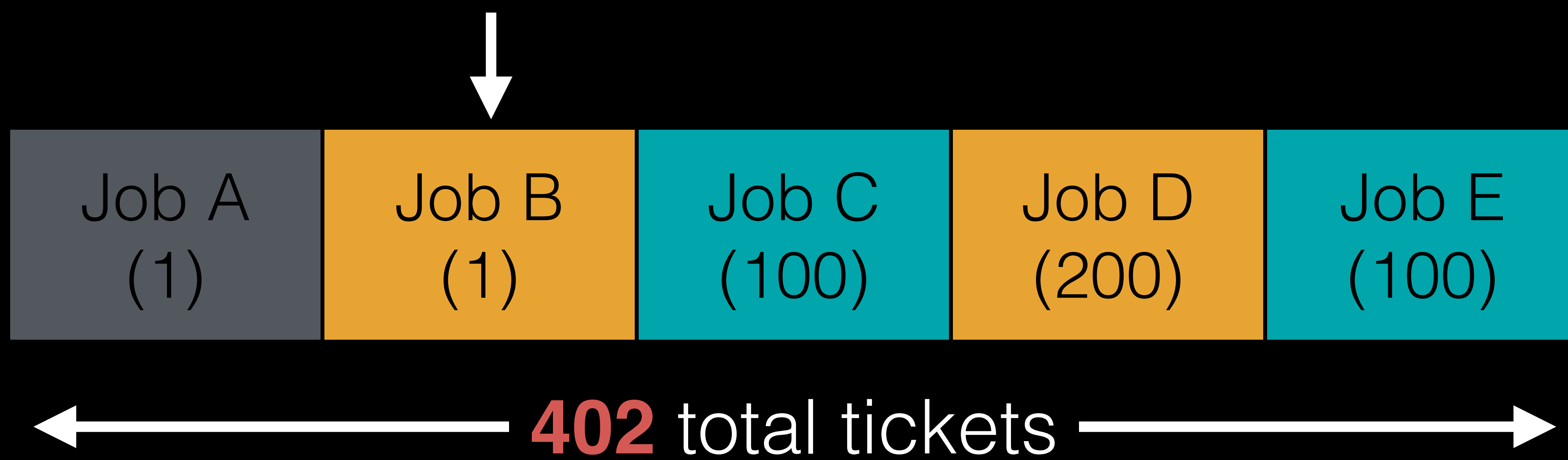
402 total tickets

winner = 102


is 102 < 1 ?
Job A (1)
Job B (1)
Job C (100)
Job D (200)
Job E (100)
402 total tickets

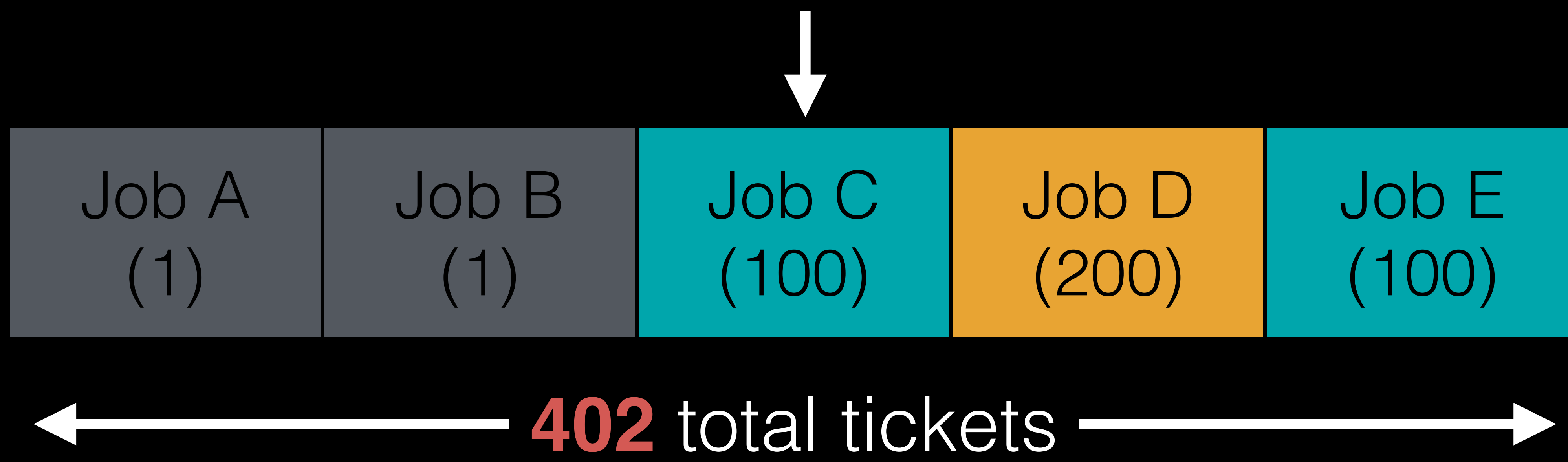
winner = 101

is 101 < 1 ?

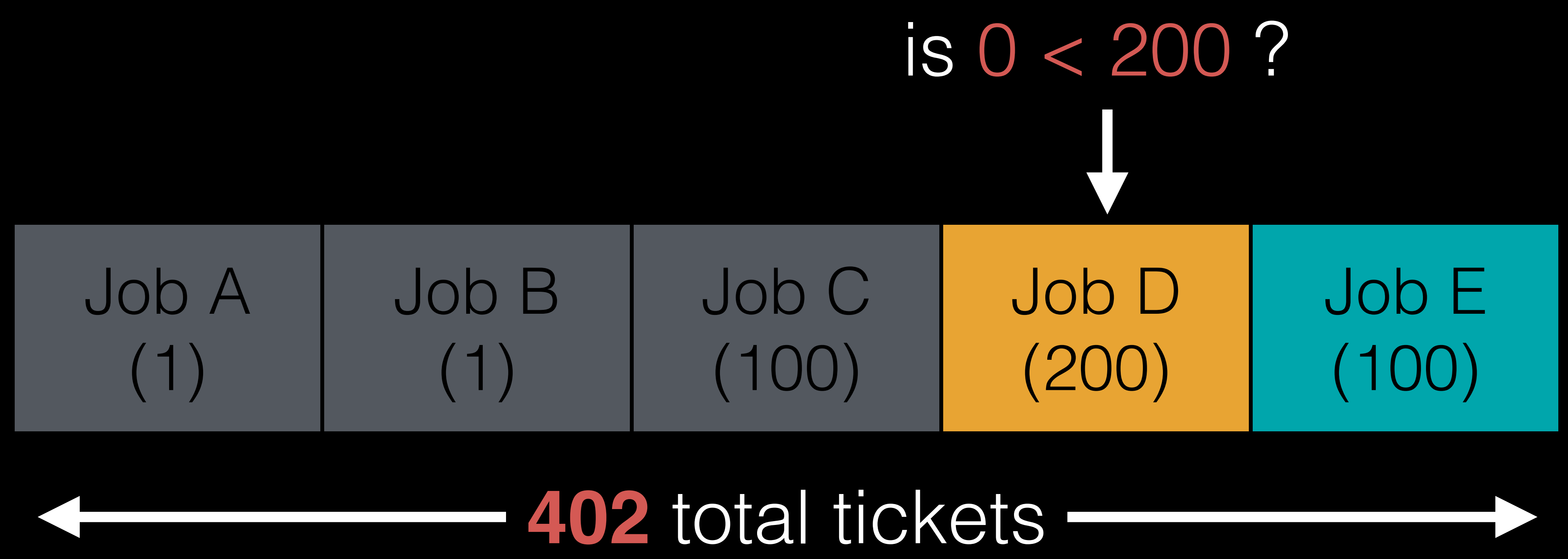


winner = 100

is 100 < 100 ?

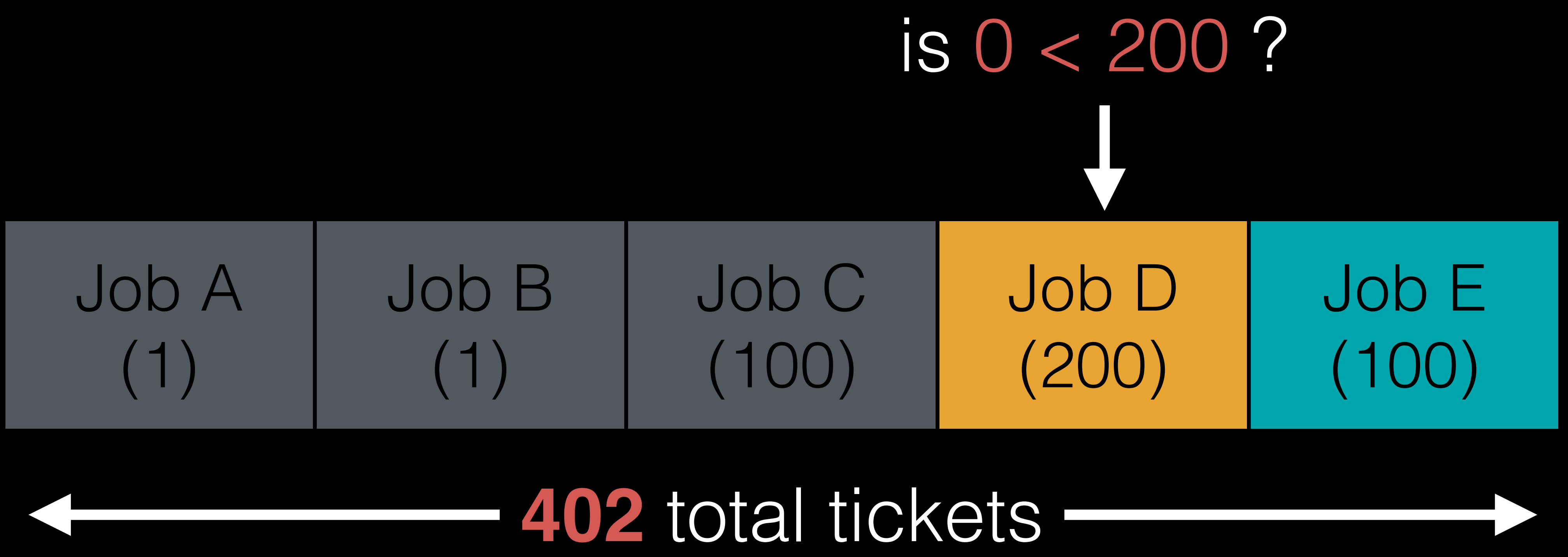


winner = 0



is 0 < 200 ?
Job A (1)
Job B (1)
Job C (100)
Job D (200)
Job E (100)
402 total tickets

Run D!



is 0 < 200 ?
Job A
(1)
Job B
(1)
Job C
(100)
Job D
(200)
Job E
(100)
402 total tickets

Address Spaces

More Virtualization

Virtual CPU: *illusion* of private CPU registers
- 2 lectures

Virtual RAM: *illusion* of private memory
- 5 lectures

The 1st "Easy Piece" in OSTEP is virtual CPU+RAM

The Abstraction

A process has a set of addresses that map to bytes

This set is called on address space

How can we provide a private address space?

Extend LDE (limited direct execution)

Review: what stuff is in an address space?

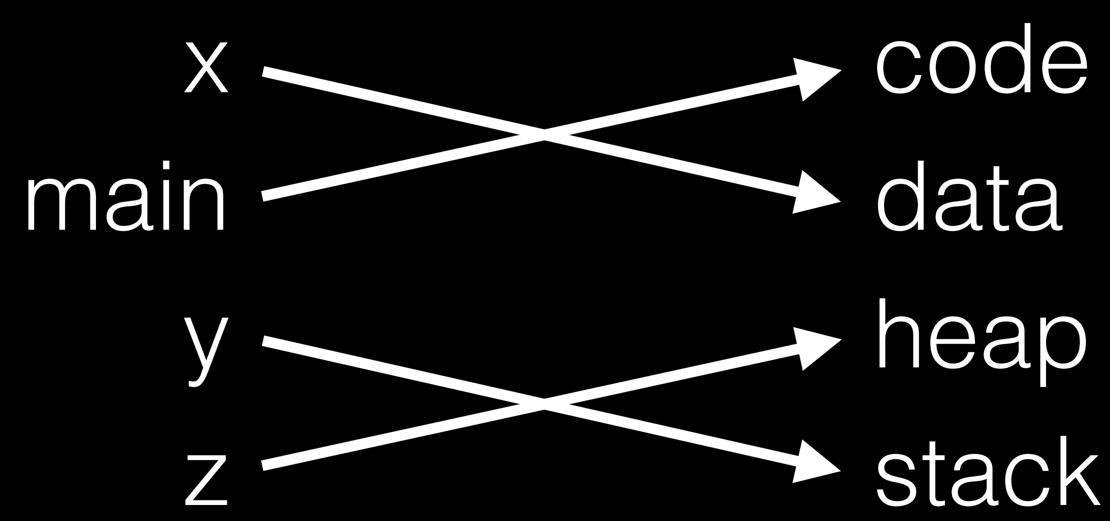
Match that Segment!

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

x	code
main	data
y	heap
z	stack

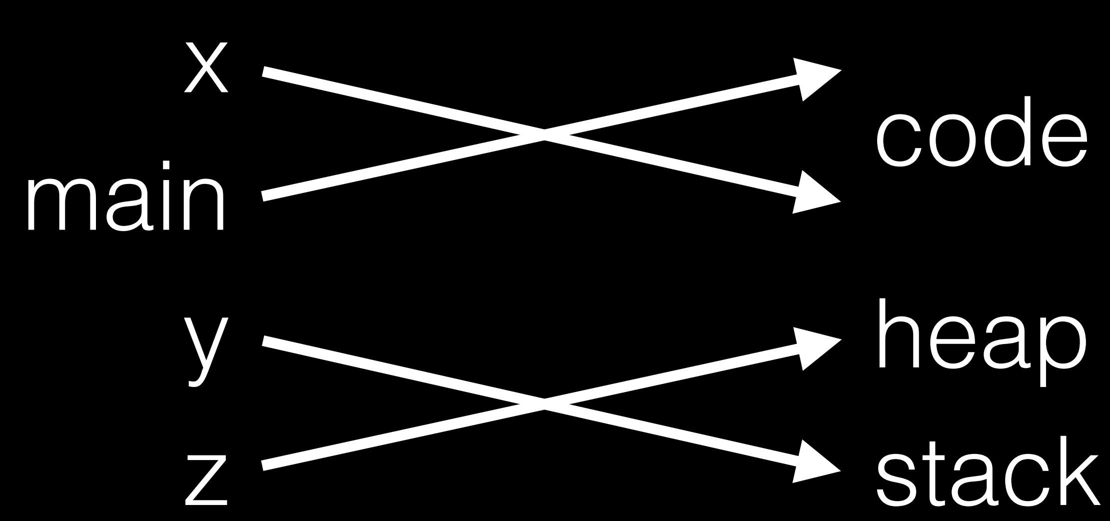
Match that Segment!

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

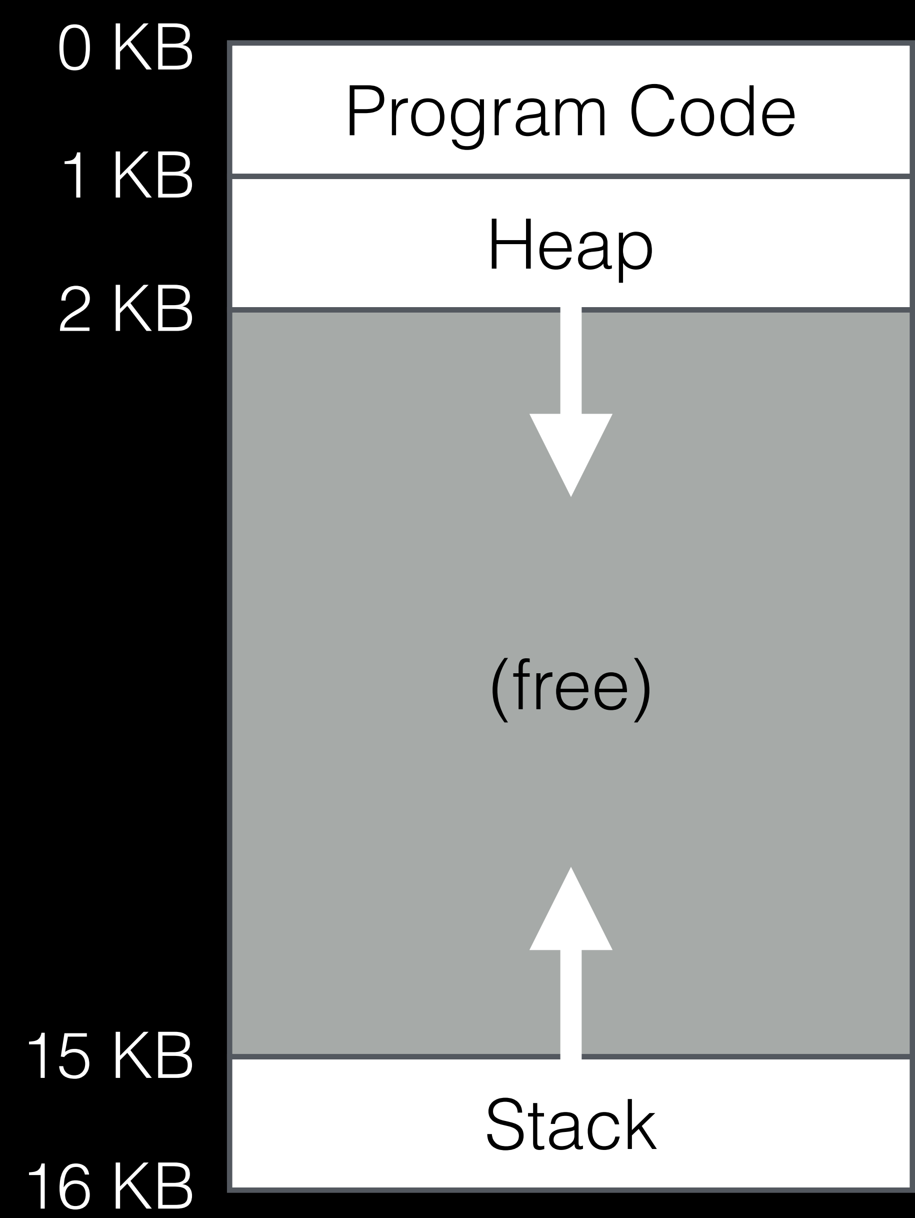


Match that Segment!

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



in OSTEP


0 KB
Program Code
1 KB
Heap
2 KB
(free)
15 KB
Stack
16 KB

demo0.c output

```
where is code?  0x100f2ddd0    (4 GB)
where is data?  0x100f2e020    (4 GB)
where is heap?  0x7ff659403930 (131033 GB)
where is stack? 0x7fff5ecd2a1c (131069 GB)
```

demo1.c disassemble

Memory Accesses

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

int main(int argc, char *argv[])
{
  int x;
  x = x + 3;
}
```

```
_main:
0000000000000000  pushq %rbp
0000000000000001  movq  %rsp, %rbp
0000000000000004  movl  $0x0, %eax
0000000000000009  movl  %edi, 0xfffffc(%rbp)
000000000000000c  movq  %rsi, 0xfffff0(%rbp)
0000000000000010  movl  0xffffec(%rbp), %edi
0000000000000013  addl  $0x3, %edi
0000000000000019  movl  %edi, 0xffffec(%rbp)
000000000000001c  popq  %rbp
000000000000001d  ret
```

otool -tv demo1.o
(or objdump on Linux)

Memory Accesses

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

int main(int argc, char *argv[])
{
  int x;
  x = x + 3;
}
```

```
_main:
0000000000000000 pushq %rbp
0000000000000001 movq  %rsp, %rbp
0000000000000004 movl  $0x0, %eax
0000000000000009 movl  %edi, 0xfffffc(%rbp)
000000000000000c movq  %rsi, 0xfffff0(%rbp)
0000000000000010 movl  0xffffec(%rbp), %edi
0000000000000013 addl  $0x3, %edi
0000000000000019 movl  %edi, 0xffffec(%rbp)
000000000000001c popq  %rbp
000000000000001d ret
```

`otool -tv demo1.o`
(or objdump on Linux)

Memory Accesses

%rip = 0x10
%rbp = 0x200

Memory Accesses:

```
0x10: movl  0x8(%rbp), %edi
0x13: addl  $0x3, %edi
0x19: movl  %edi, 0x8(%rbp)
```

Memory Accesses

%rip = 0x10
%rbp = 0x200

```
0x10: movl  0x8(%rbp), %edi
0x13: addl  $0x3, %edi
0x19: movl  %edi, 0x8(%rbp)
```

Memory Accesses:
Fetch instruction at addr 0x10

Memory Accesses

%rip = 0x10
%rbp = 0x200

```
0x10: movl  0x8(%rbp), %edi
0x13: addl  $0x3, %edi
0x19: movl  %edi, 0x8(%rbp)
```

Memory Accesses:

Fetch instruction at addr 0x10
Exec, load from addr 0x208

Memory Accesses

%rip = 0x13
%rbp = 0x200

```
0x10: movl  0x8(%rbp), %edi
0x13: addl  $0x3, %edi
0x19: movl  %edi, 0x8(%rbp)
```

Memory Accesses:

Fetch instruction at addr 0x10
Exec, load from addr 0x208

Memory Accesses

%rip = 0x13
%rbp = 0x200

```
0x10: movl  0x8(%rbp), %edi
0x13: addl  $0x3, %edi
0x19: movl  %edi, 0x8(%rbp)
```

Memory Accesses:

Fetch instruction at addr 0x10
Exec, load from addr 0x208

Fetch instruction at addr 0x13

Memory Accesses

%rip = 0x13
%rbp = 0x200

```
0x10: movl  0x8(%rbp), %edi
0x13: addl  $0x3, %edi
0x19: movl  %edi, 0x8(%rbp)
```

Memory Accesses:

Fetch instruction at addr 0x10
Exec, load from addr 0x208

Fetch instruction at addr 0x13
Exec, no load

Memory Accesses

%rip = 0x19
%rbp = 0x200

```
0x10: movl  0x8(%rbp), %edi
0x13: addl  $0x3, %edi
0x19: movl  %edi, 0x8(%rbp)
```

Memory Accesses:

Fetch instruction at addr 0x10
Exec, load from addr 0x208

Fetch instruction at addr 0x13
Exec, no load

Memory Accesses

%rip = 0x19
%rbp = 0x200

```
0x10: movl  0x8(%rbp), %edi
0x13: addl  $0x3, %edi
0x19: movl  %edi, 0x8(%rbp)
```

Memory Accesses:

Fetch instruction at addr 0x10
Exec, load from addr 0x208

Fetch instruction at addr 0x13
Exec, no load

Fetch instruction at addr 0x19

Memory Accesses

%rip = 0x19
%rbp = 0x200

```
0x10: movl  0x8(%rbp), %edi
0x13: addl  $0x3, %edi
0x19: movl  %edi, 0x8(%rbp)
```

Memory Accesses:

Fetch instruction at addr 0x10
Exec, load from addr 0x208

Fetch instruction at addr 0x13
Exec, no load

Fetch instruction at addr 0x19
Exec, store to addr 0x208

Problem: How to Run Multiple Processes?

Addresses are “hardcoded” into process binaries.
How to avoid collisions?

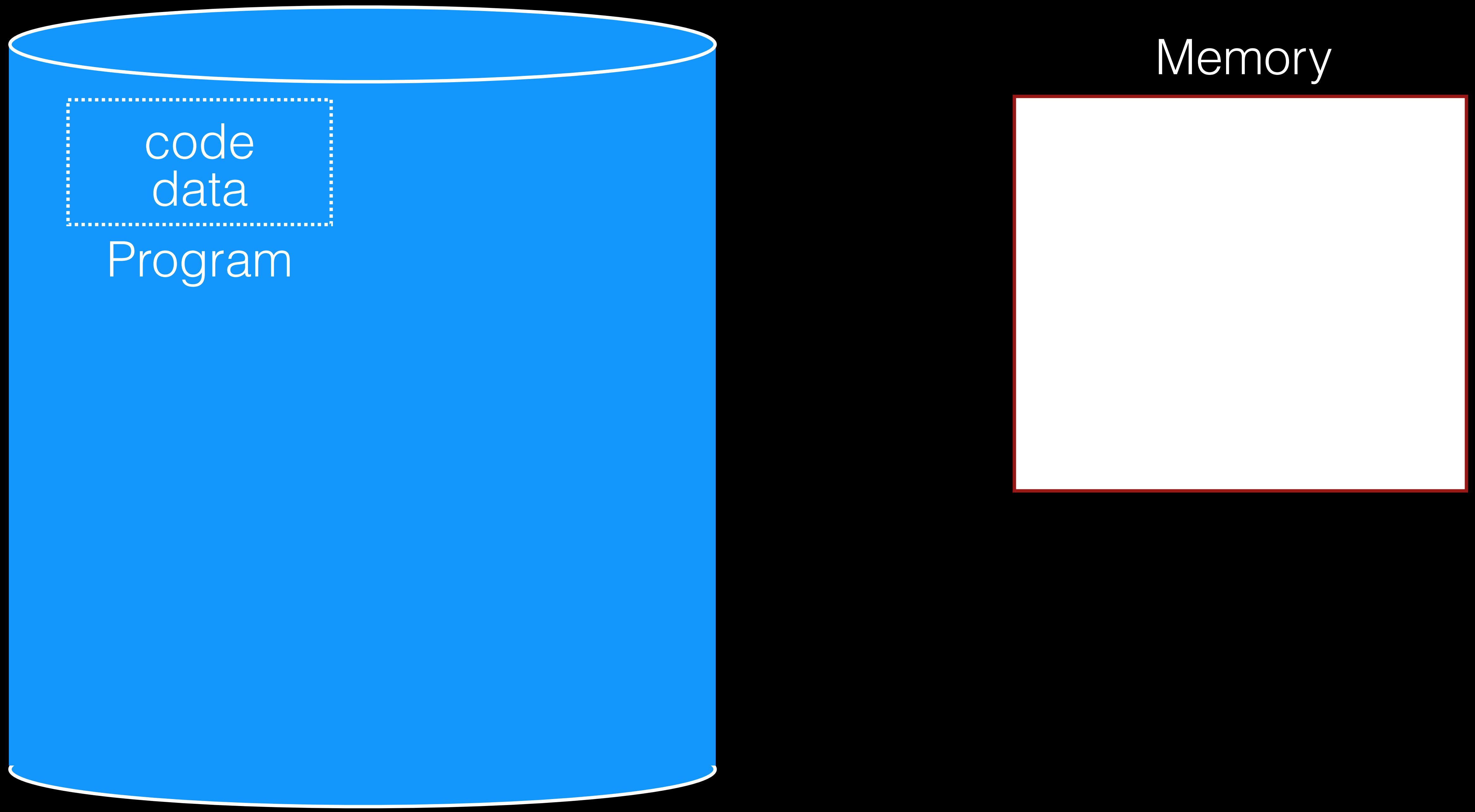
Approaches (covered today):

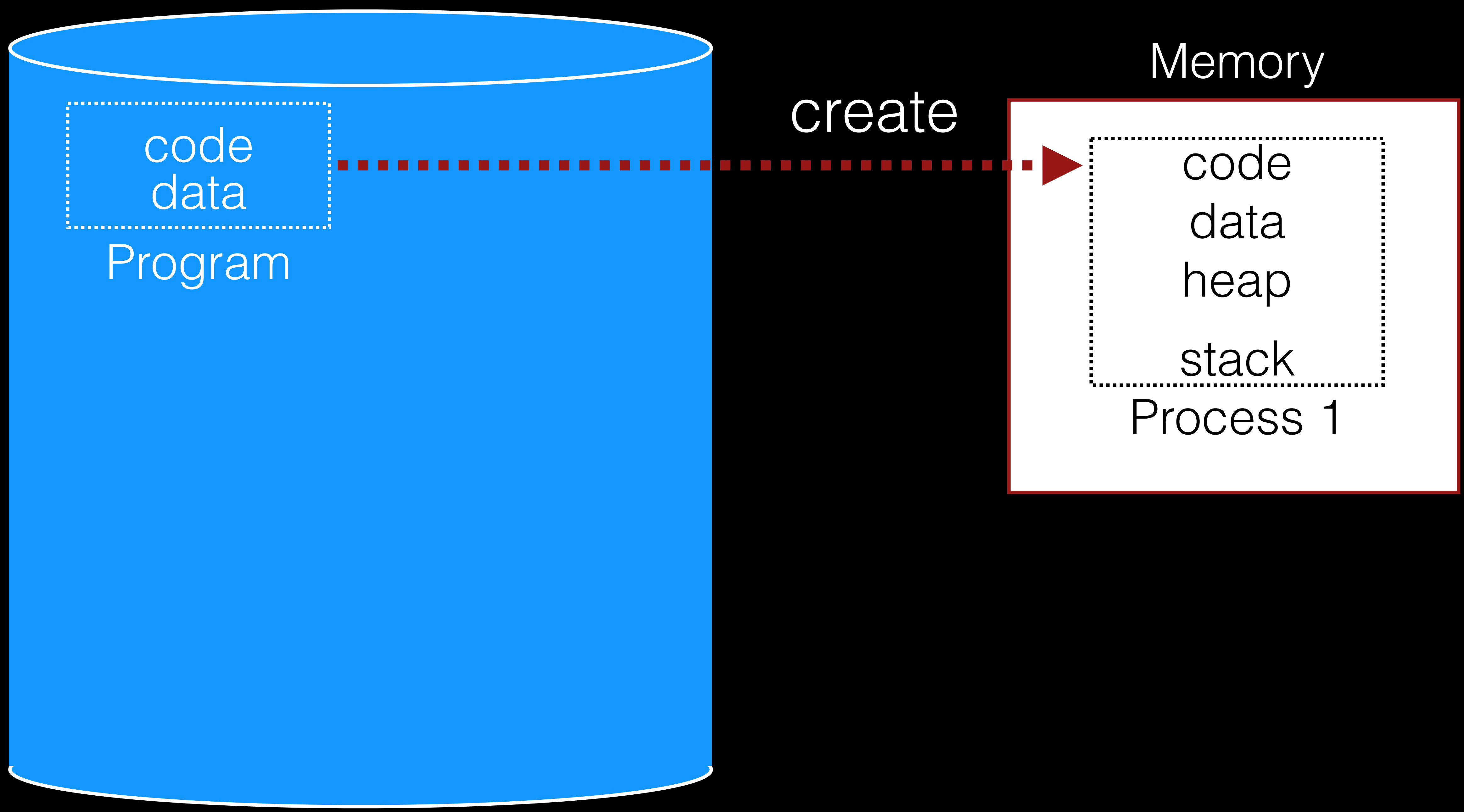
- Time Sharing
- Static Relocation
- Base
- Base+Bounds
- Segmentation

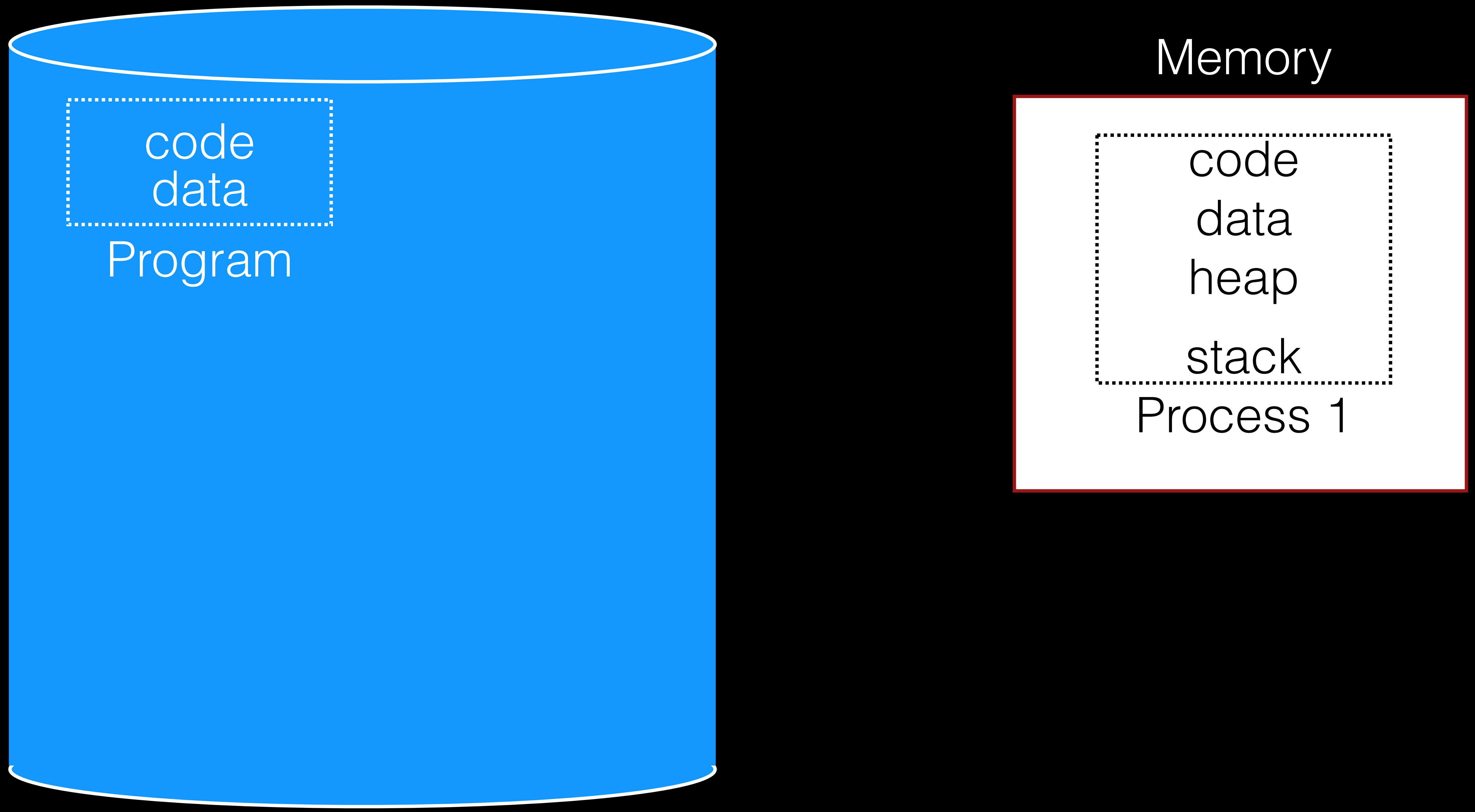
Time Sharing

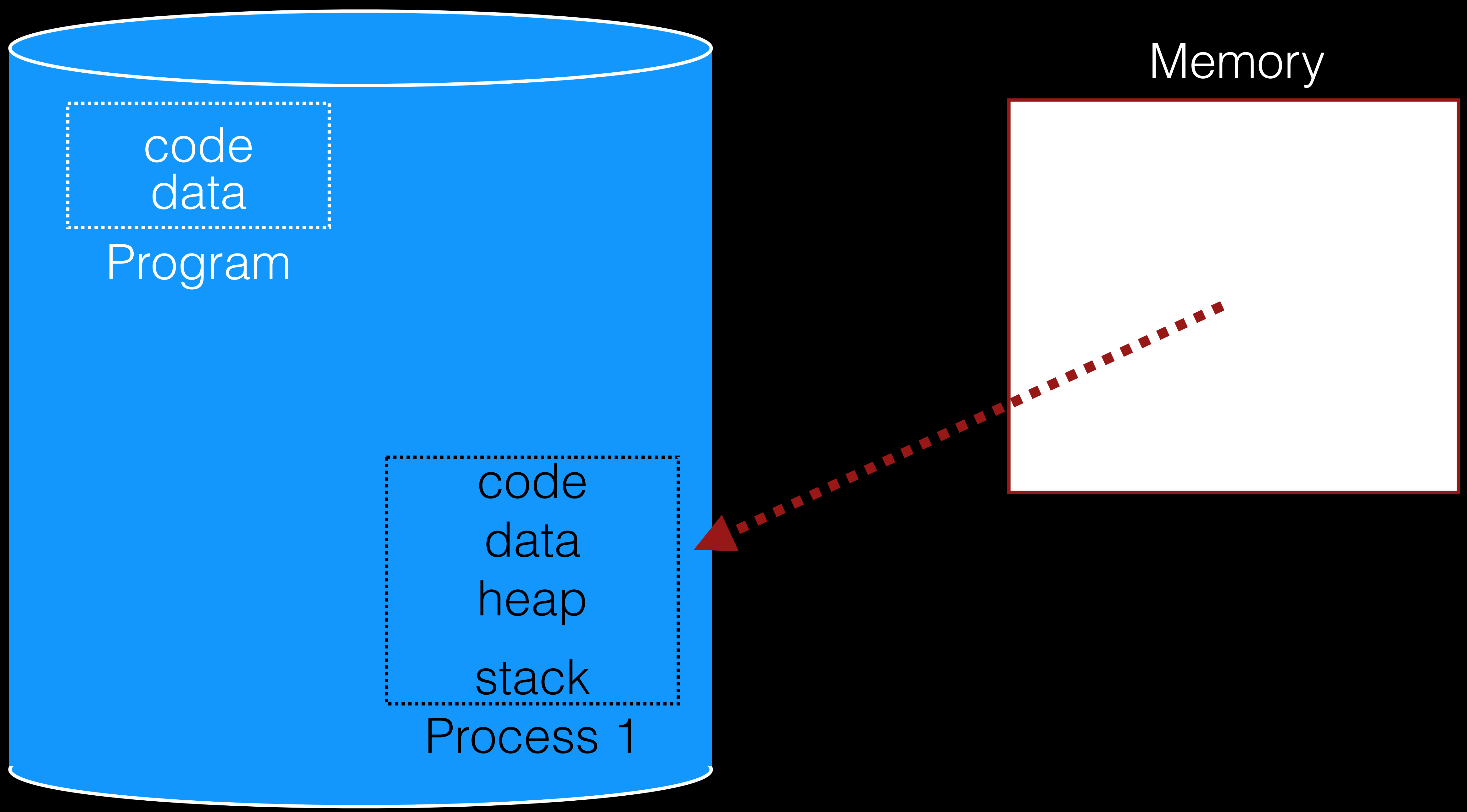
We give the illusion of many virtual CPUs by saving **CPU registers** to **memory** when a process isn't running

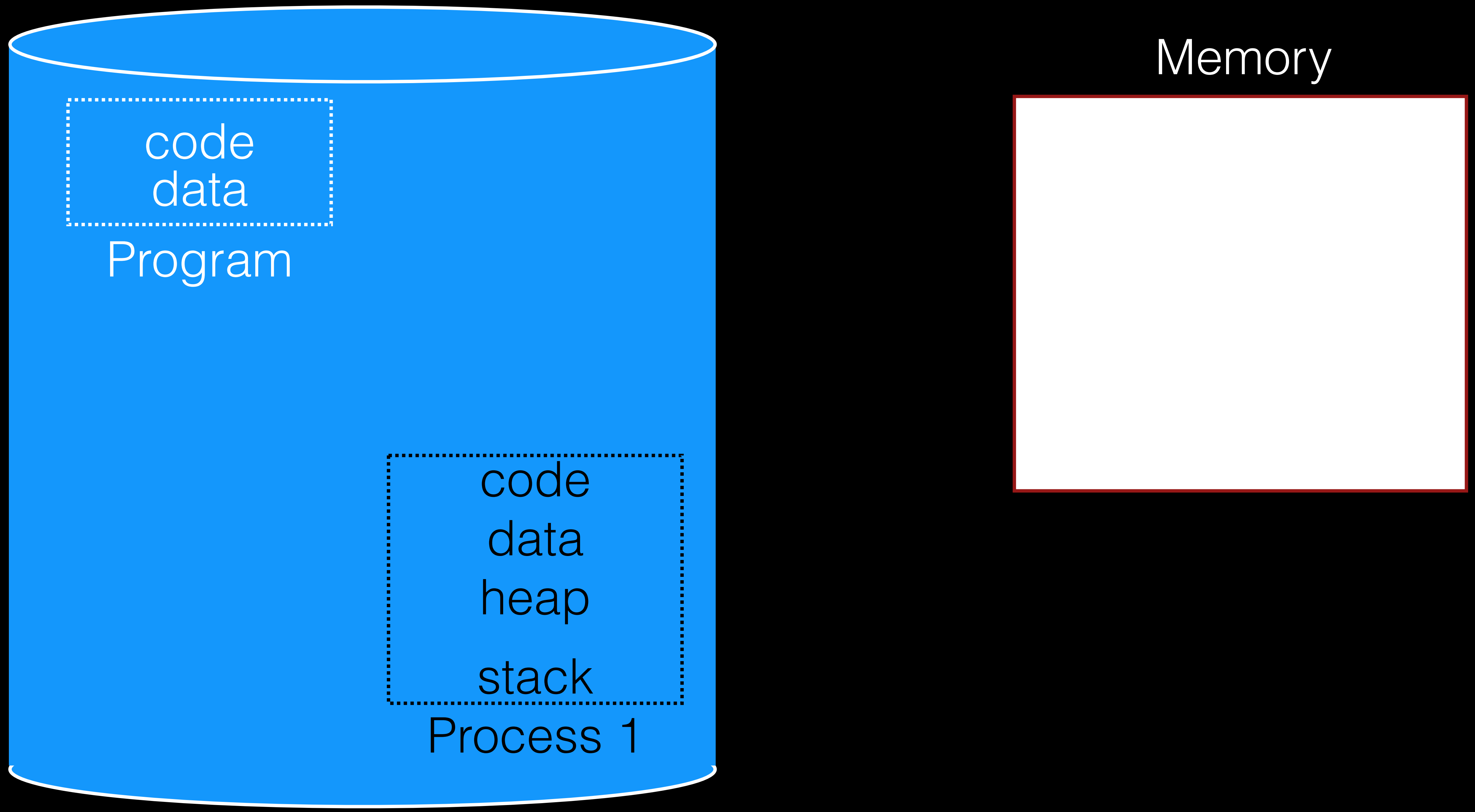
We give the illusion of many virtual memories by saving **memory** to **disk** when a process isn't running

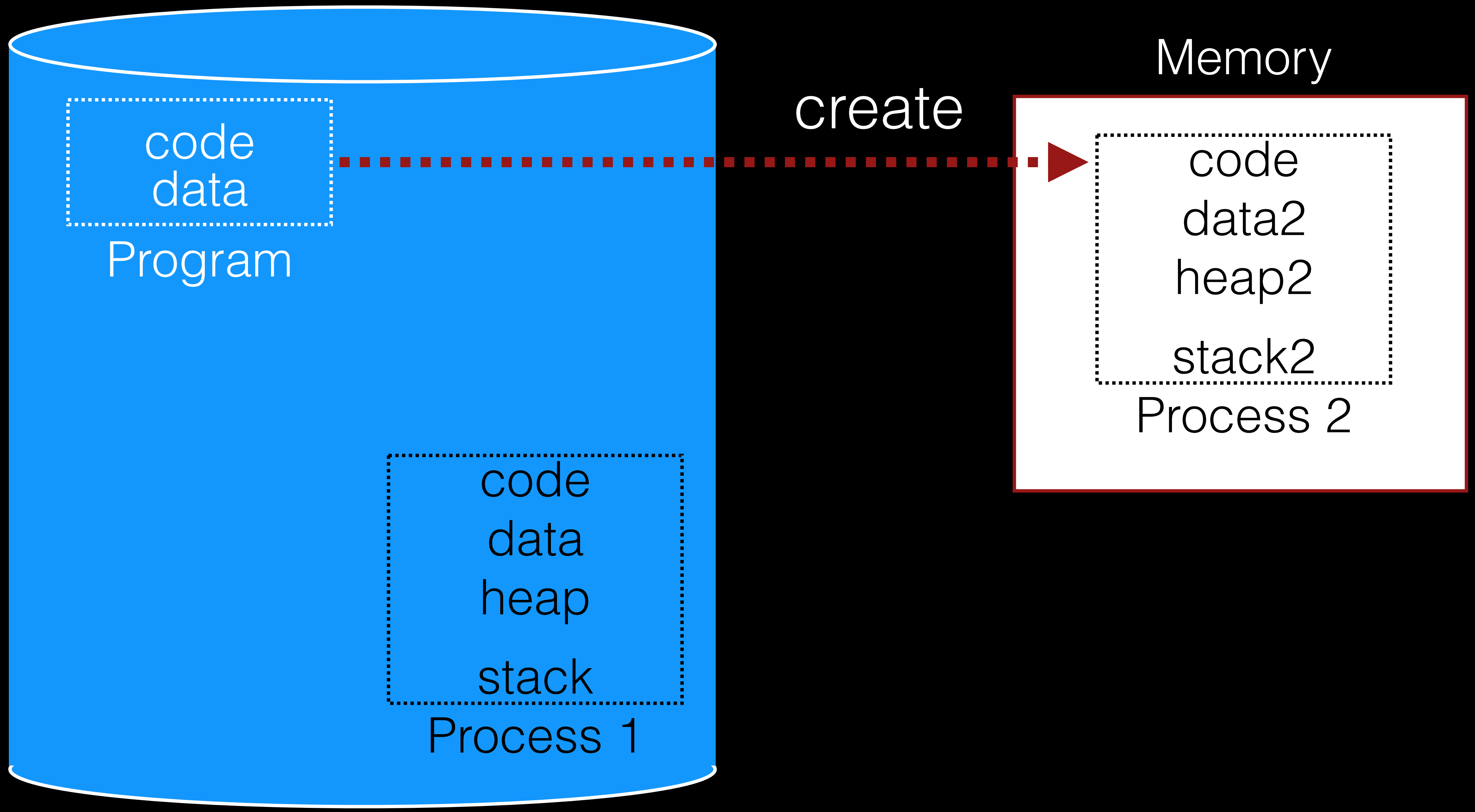

code
data
Program
Memory

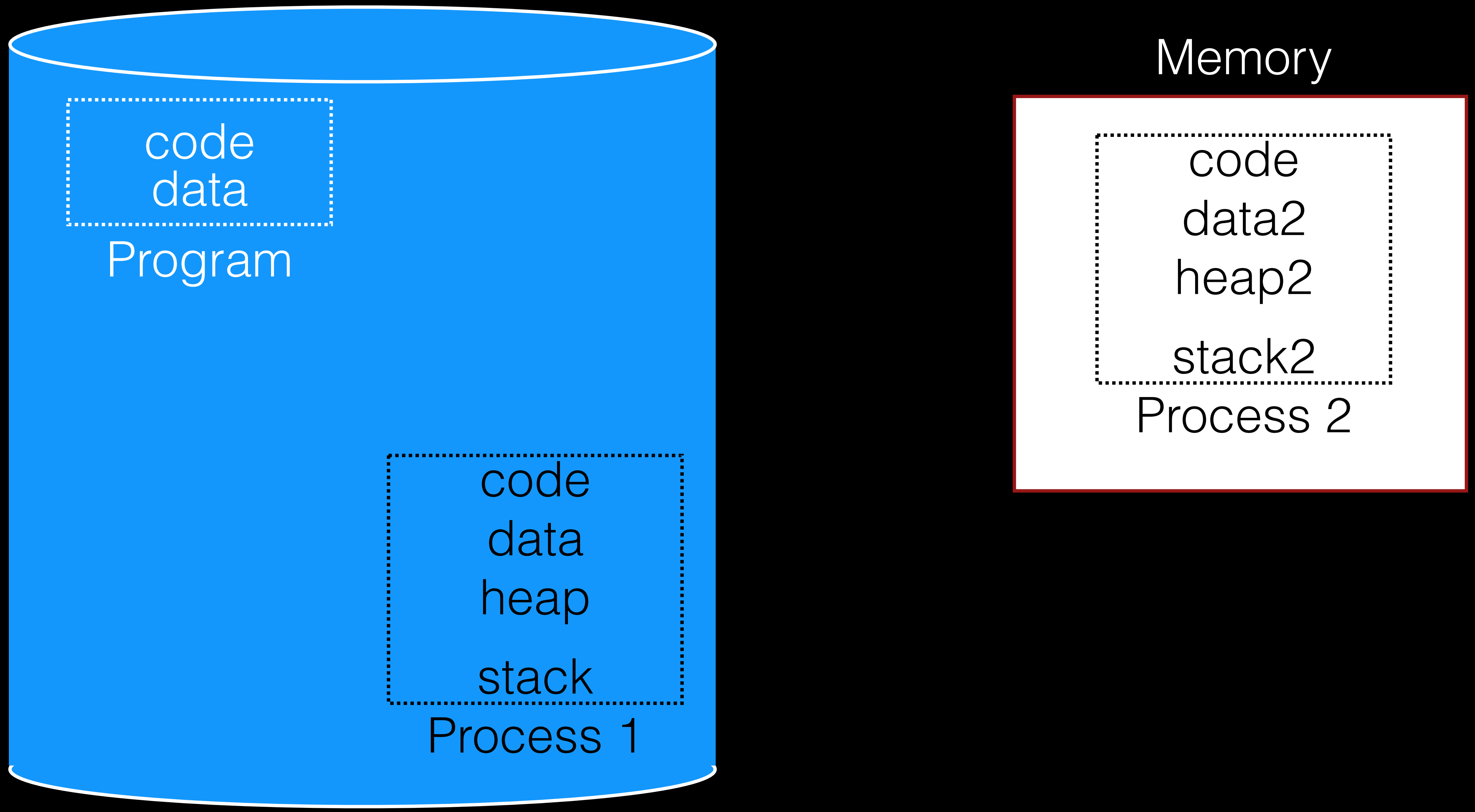

code
data
Program
create
Memory
code
data
heap
stack
Process 1

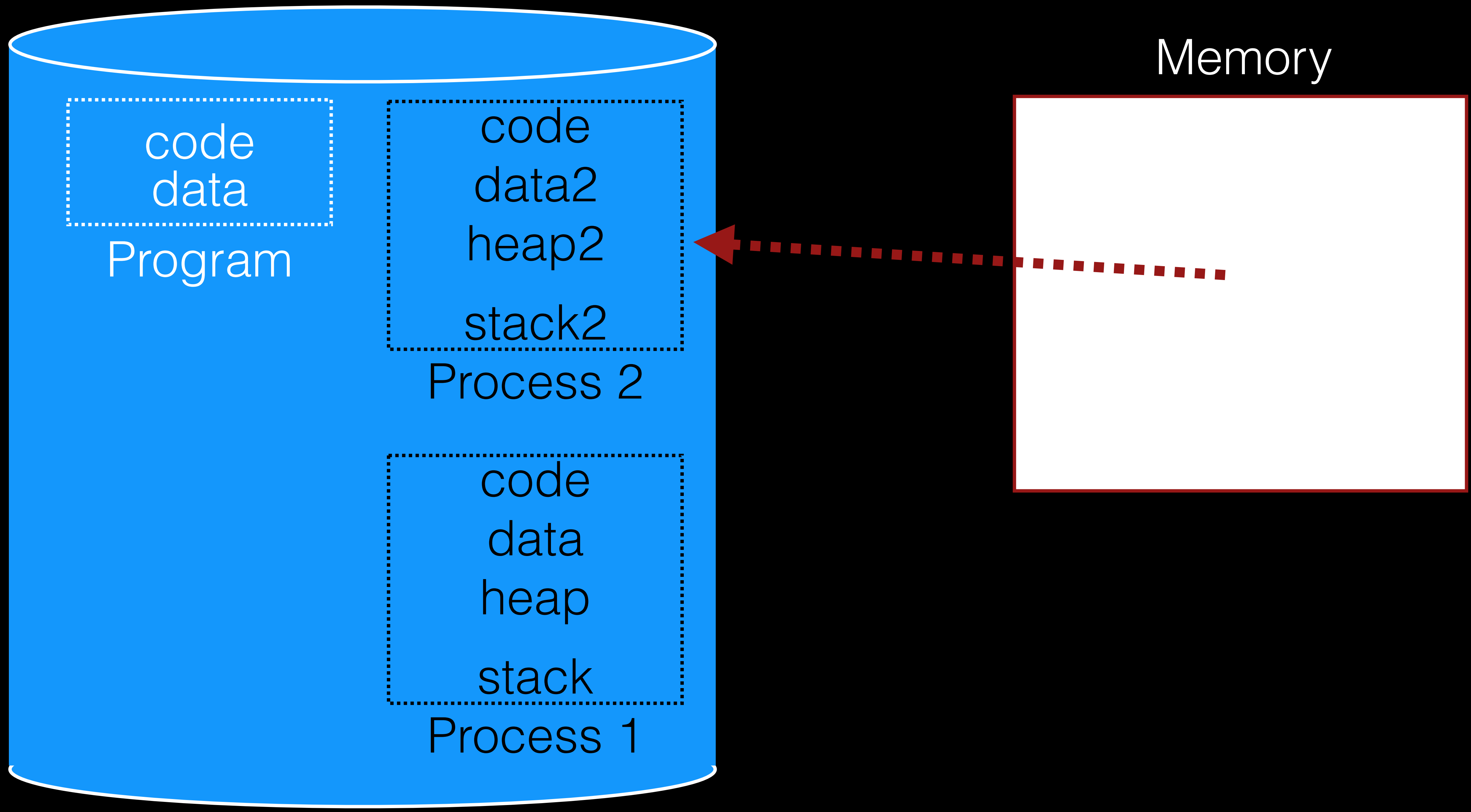

code
data
Program
Memory
code
data
heap
stack
Process 1

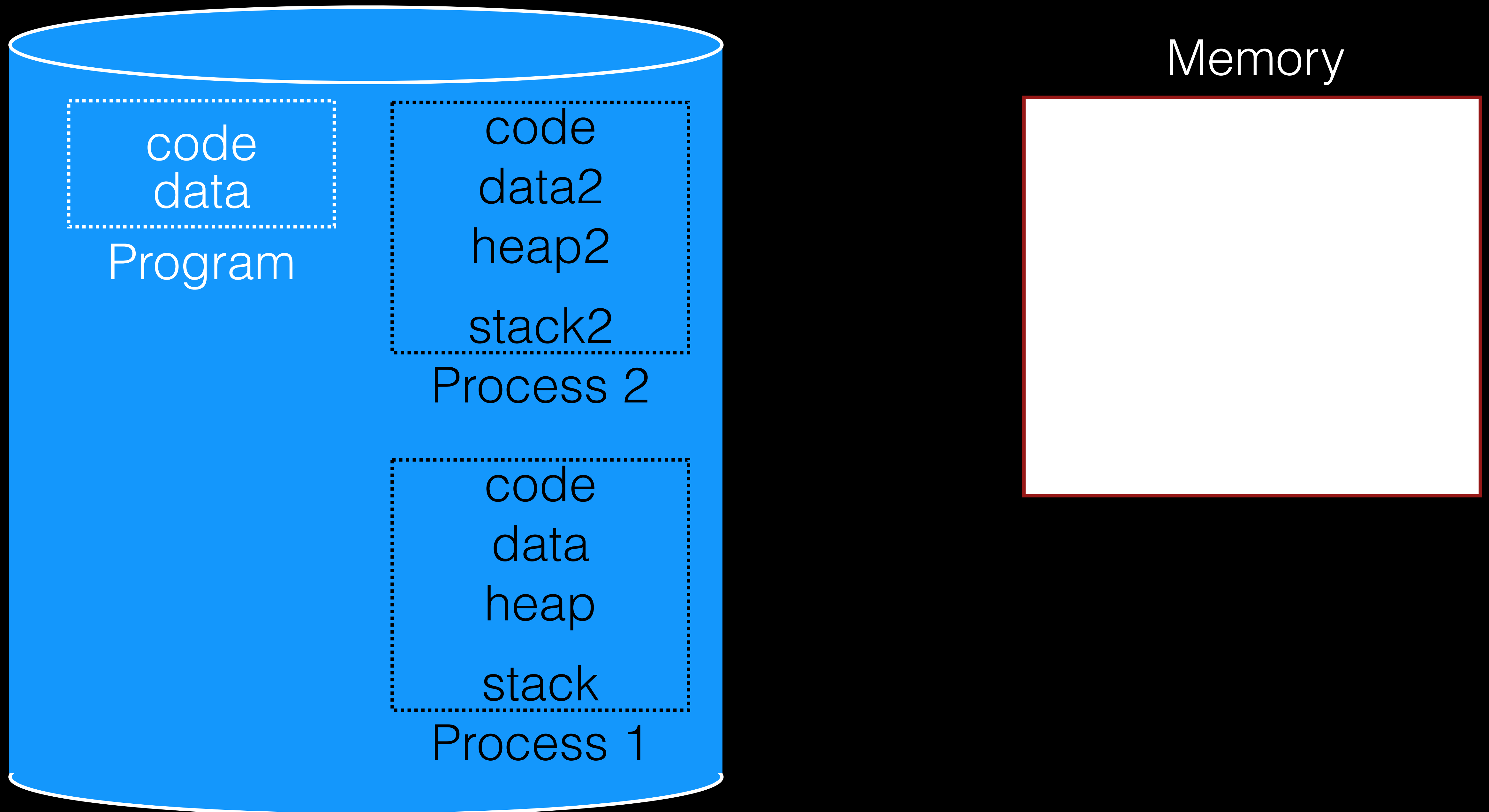

code
data
Program
code
data
heap
stack
Process 1
Memory

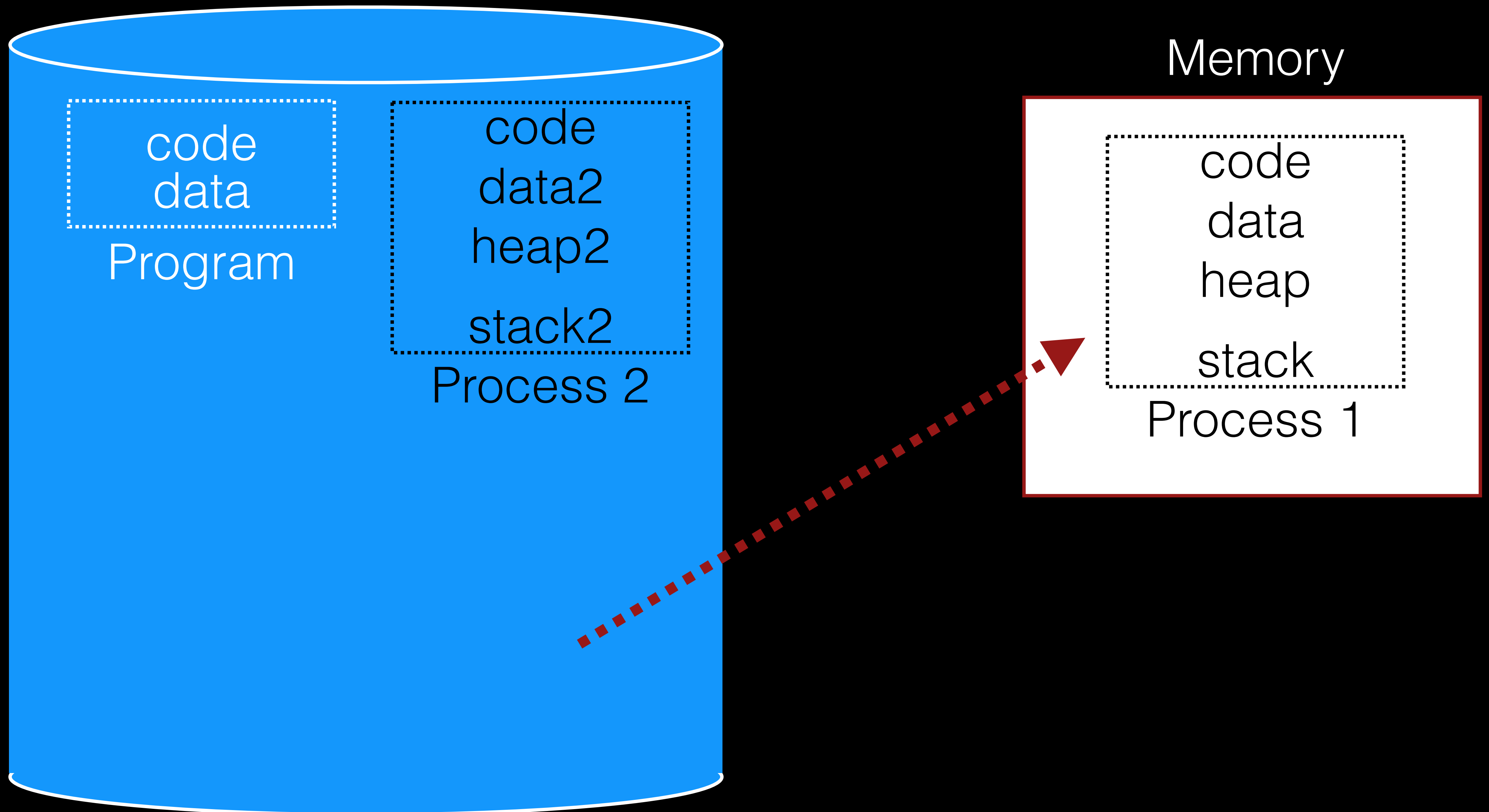

code
data
Program
code
data
heap
stack
Process 1
Memory


code
data
Program
create
Memory
code
data2
heap2
stack2
Process 2
code
data
heap
stack
Process 1

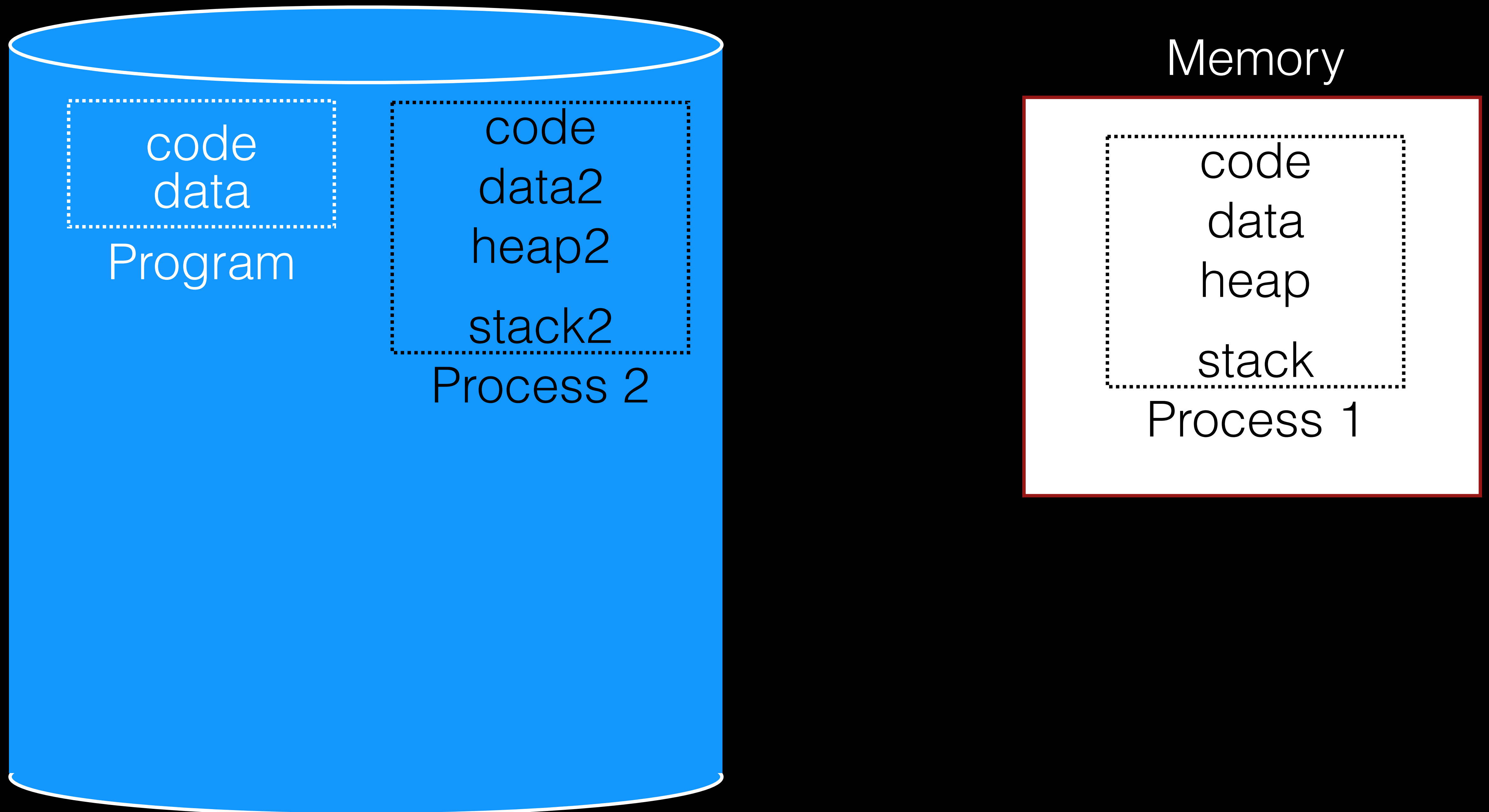

code
data
Program
code
data
heap
stack
Process 1
Memory
code
data2
heap2
stack2
Process 2


code
data
Program
code
data2
heap2
stack2
Process 2
code
data
heap
stack
Process 1
Memory


code
data
Program
code
data2
heap2
stack2
Process 2
code
data
heap
stack
Process 1
Memory



code
data
Program
code
data2
heap2
stack2
Process 2
Memory
code
data
heap
stack
Process 1


code
data
Program
code
data2
heap2
stack2
Process 2
Memory
code
data
heap
stack
Process 1

Time Sharing

Problems?

What schedulers would time sharing work well with?

Alternative: space sharing

Problem: How to Run Multiple Processes?

Approaches (covered today):

- Time Sharing
- Static Relocation
- Base
- Base+Bounds
- Segmentation

Static Relocation

Idea: rewrite each program before loading it as a process

Each rewrite uses different addresses and pointers

Change jumps, loads, etc.
Can any addresses be unchanged?

Rewrite for Each New Process

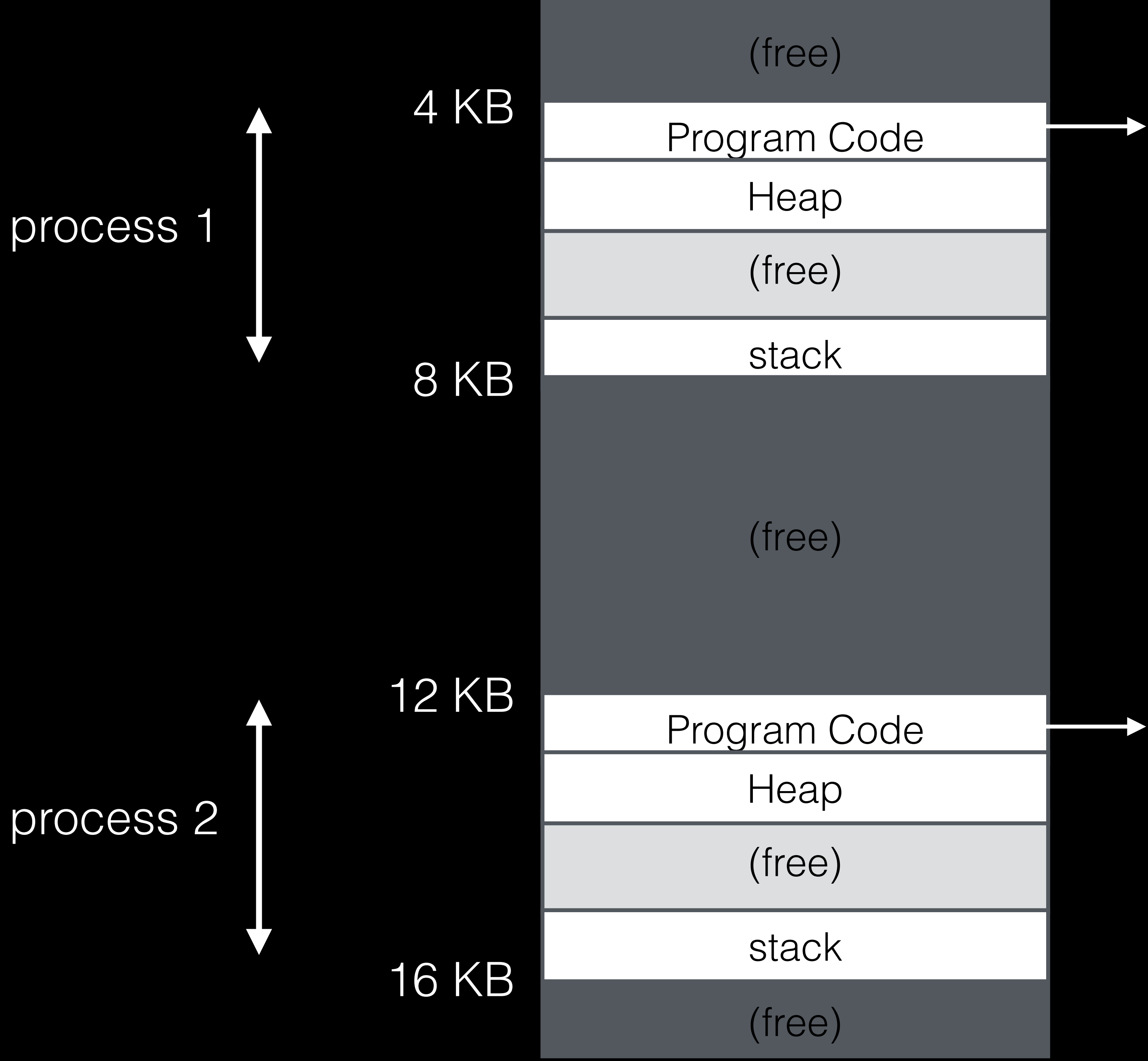
```
0x1010: movl  0x8(%rbp), %edi
0x1013: addl  $0x3, %edi
0x1019: movl  %edi, 0x8(%rbp)
```

rewrite

```
0x10: movl  0x8(%rbp), %edi
0x13: addl  $0x3, %edi
0x19: movl  %edi, 0x8(%rbp)
```

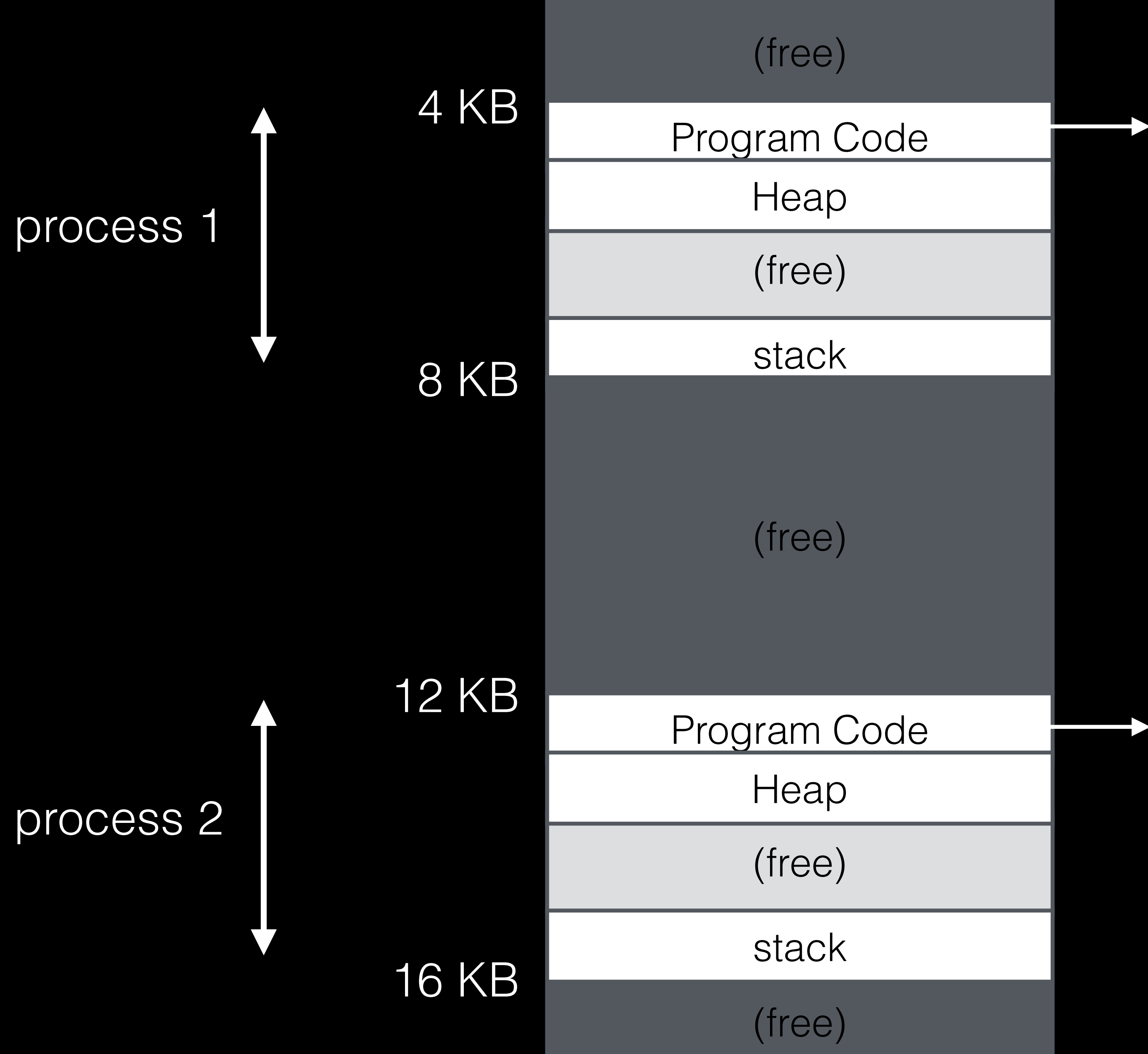
rewrite

```
0x3010: movl  0x8(%rbp), %edi
0x3013: addl  $0x3, %edi
0x3019: movl  %edi, 0x8(%rbp)
```




```
0x1010: movl  0x8(%rbp), %edi
0x1013: addl  $0x3, %edi
0x1019: movl  %edi, 0x8(%rbp)
```

```
0x3010: movl  0x8(%rbp), %edi
0x3013: addl  $0x3, %edi
0x3019: movl  %edi, 0x8(%rbp)
```




```
0x1010: movl  0x8(%rbp), %edi
0x1013: addl  $0x3, %edi
0x1019: movl  %edi, 0x8(%rbp)
```

why didn’t we have to rewrite the stack addr?

```
0x3010: movl  0x8(%rbp), %edi
0x3013: addl  $0x3, %edi
0x3019: movl  %edi, 0x8(%rbp)
```

Problem: How to Run Multiple Processes?

Approaches (covered today):

- Time Sharing
- Static Relocation
- Base
- Base+Bounds
- Segmentation

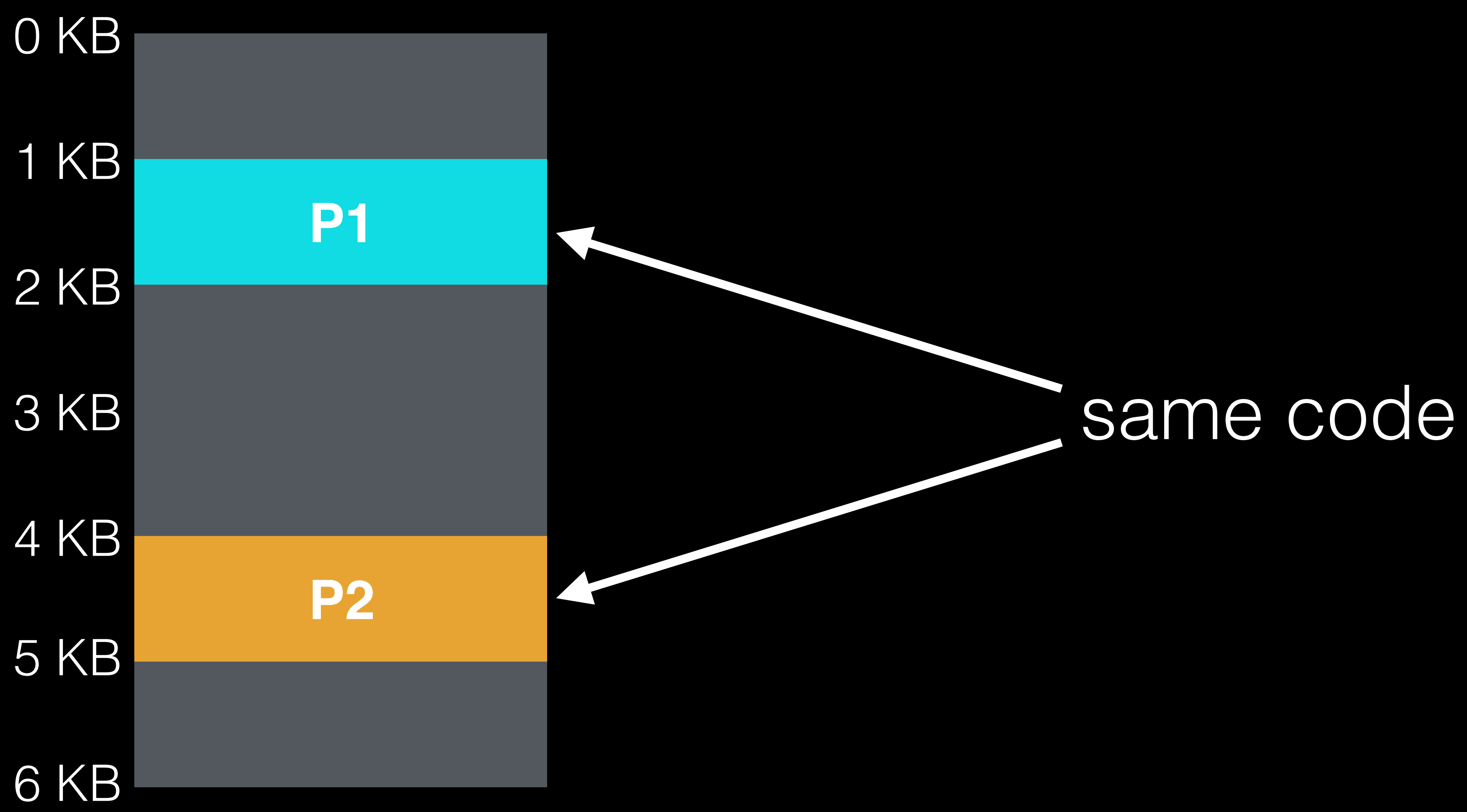
Base

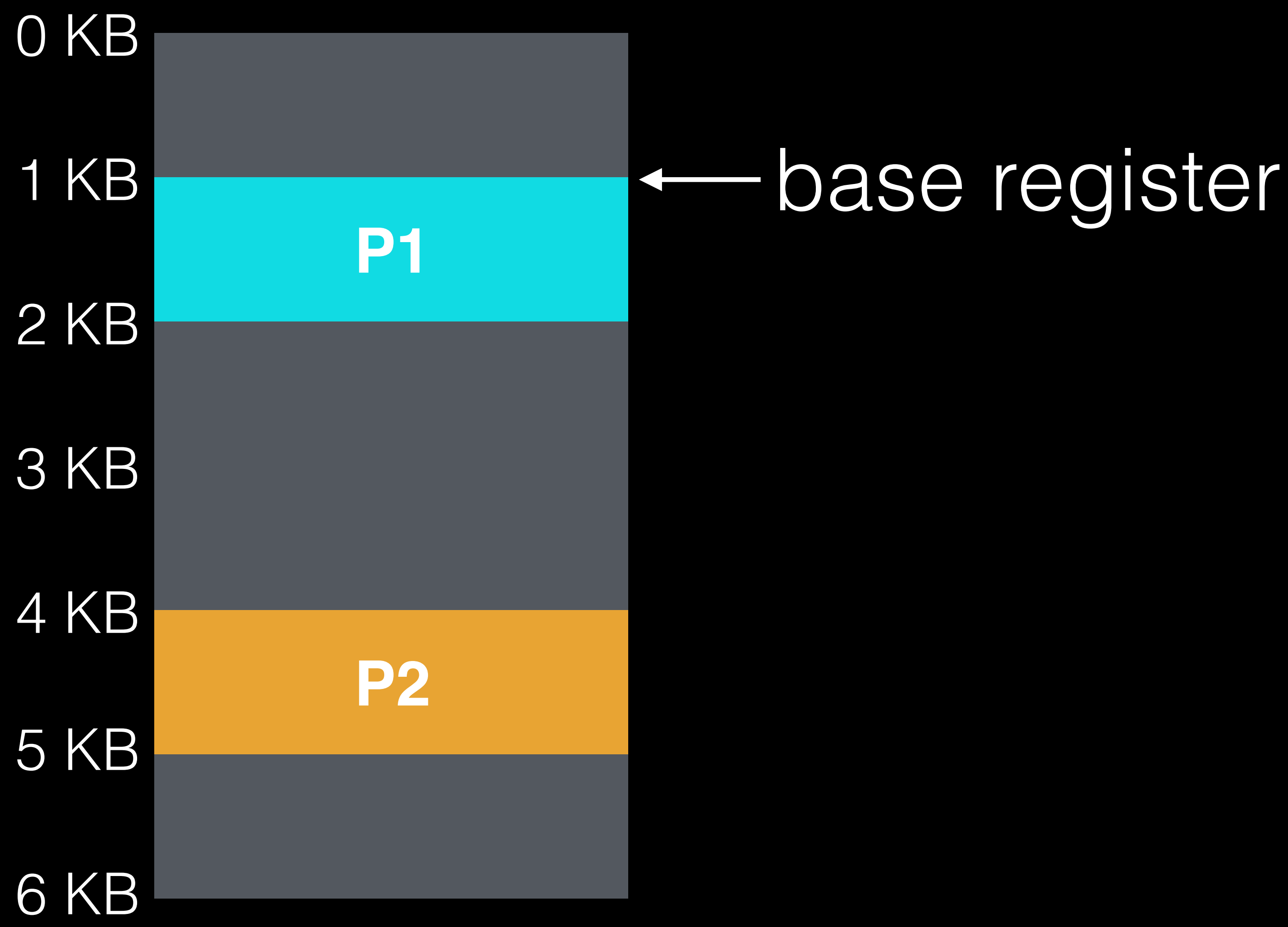
Idea: translate virtual addresses to physical by adding a fixed offset each time.

Store offset in a base register.

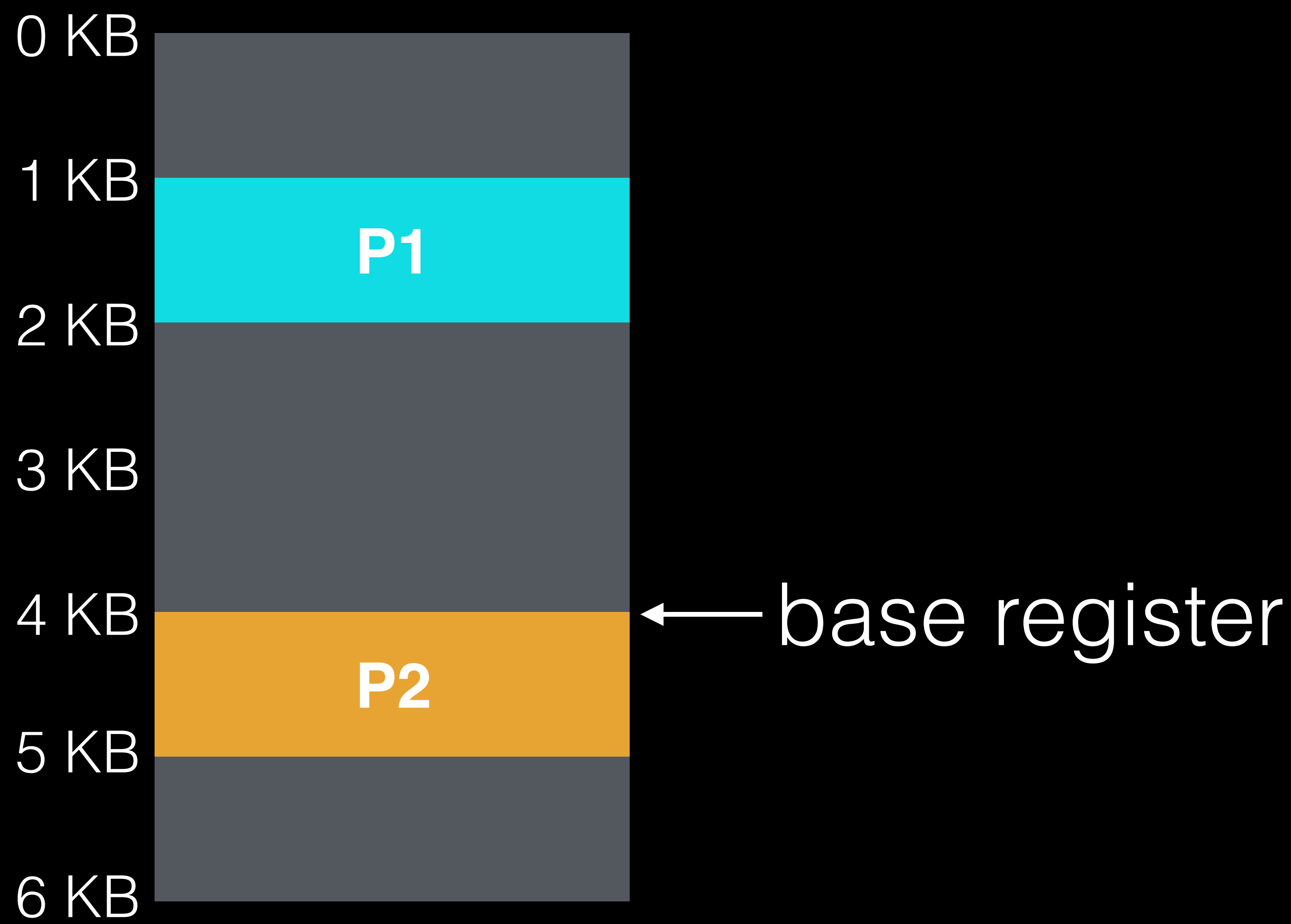
Each process has a different value in the base register when running.

This is a “dynamic relocation” technique

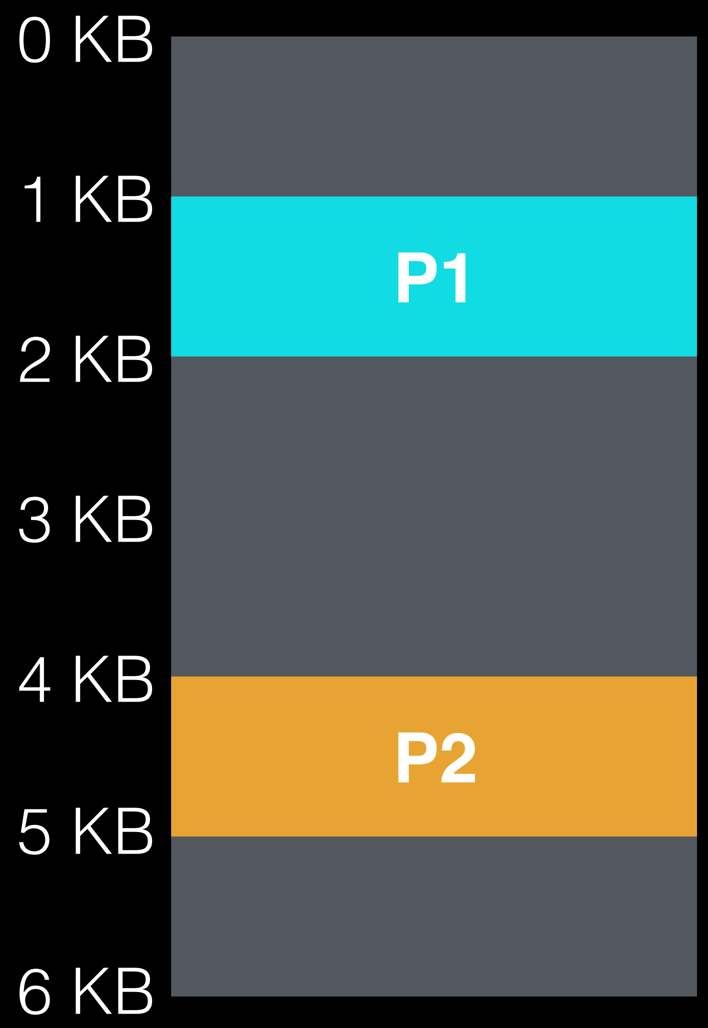

0 KB
1 KB
P1
2 KB
3 KB
4 KB
P2
5 KB
6 KB
same code



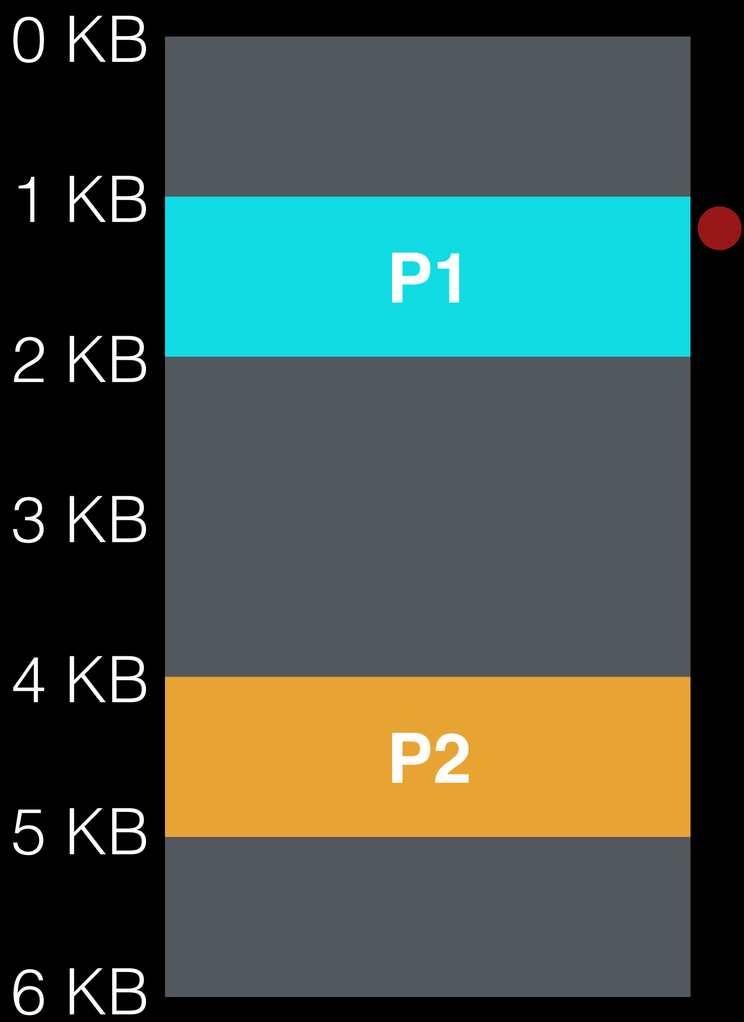
P1 is running



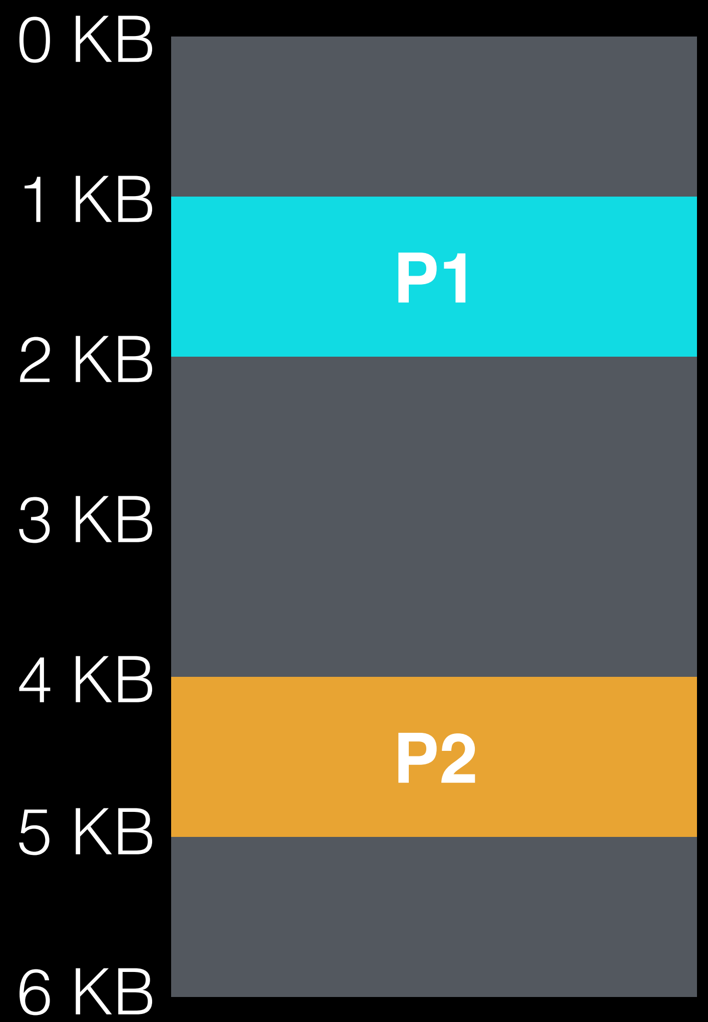
P2 is running



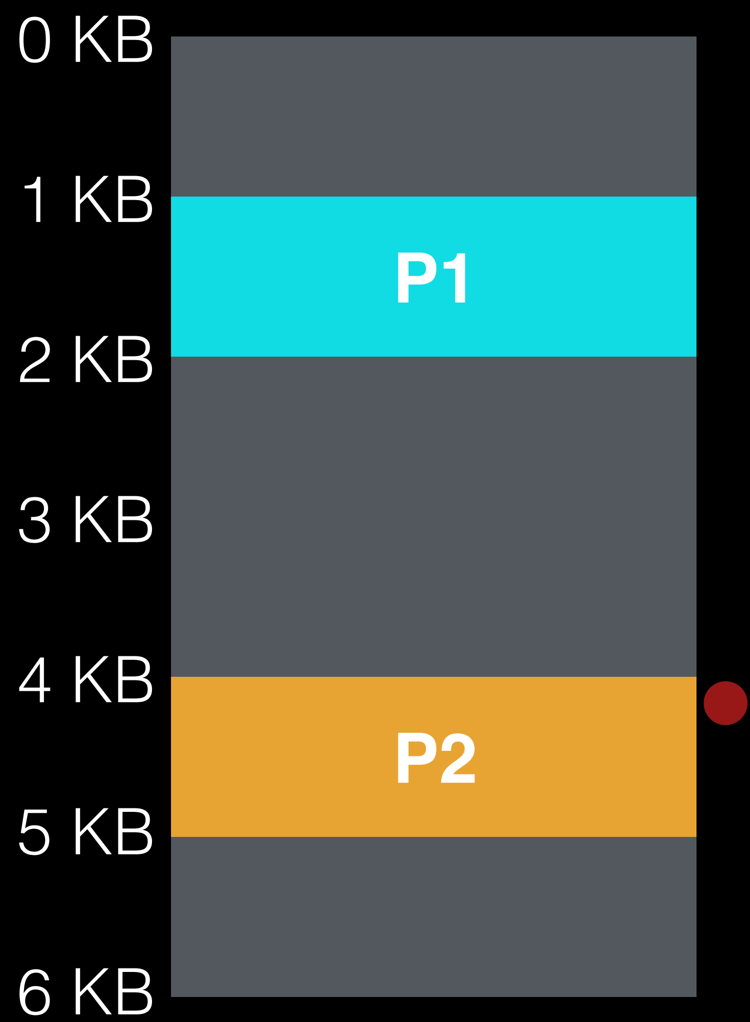
Virtual	Physical
P1: load 100, R1	



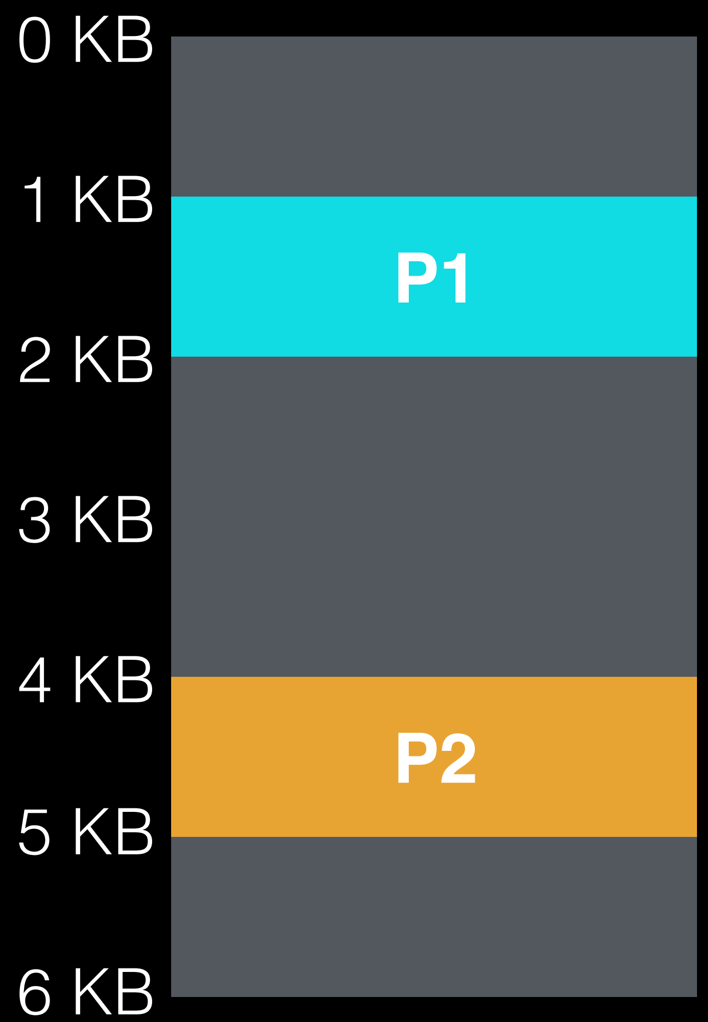
Virtual	Physical
P1: load 100, R1	load 1124, R1



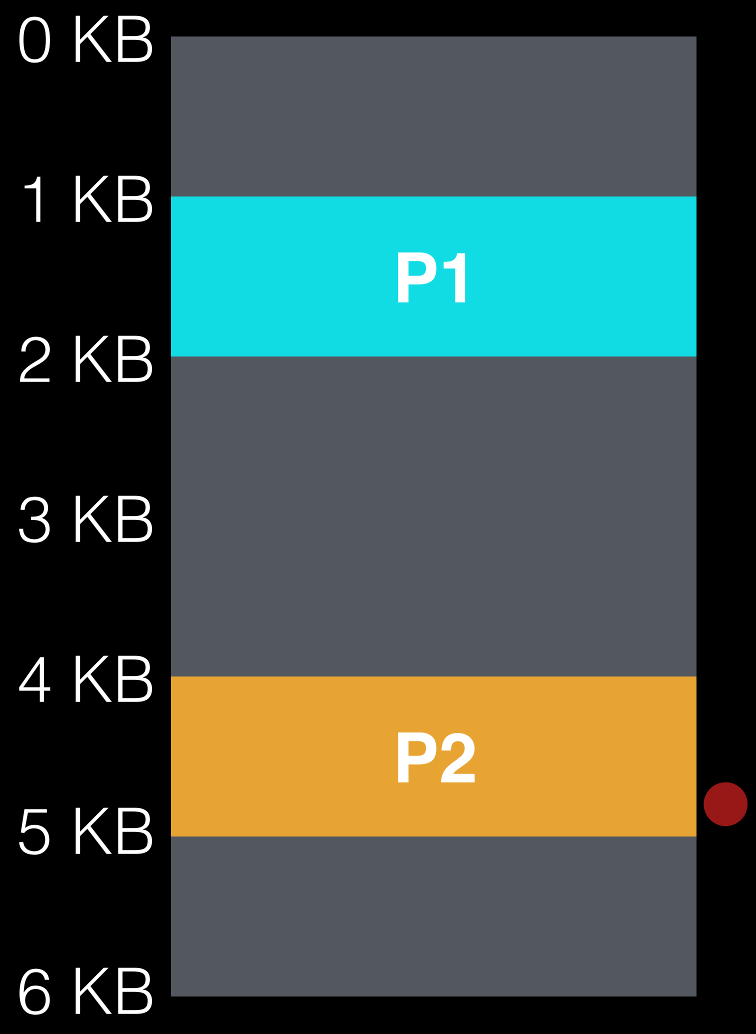
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	



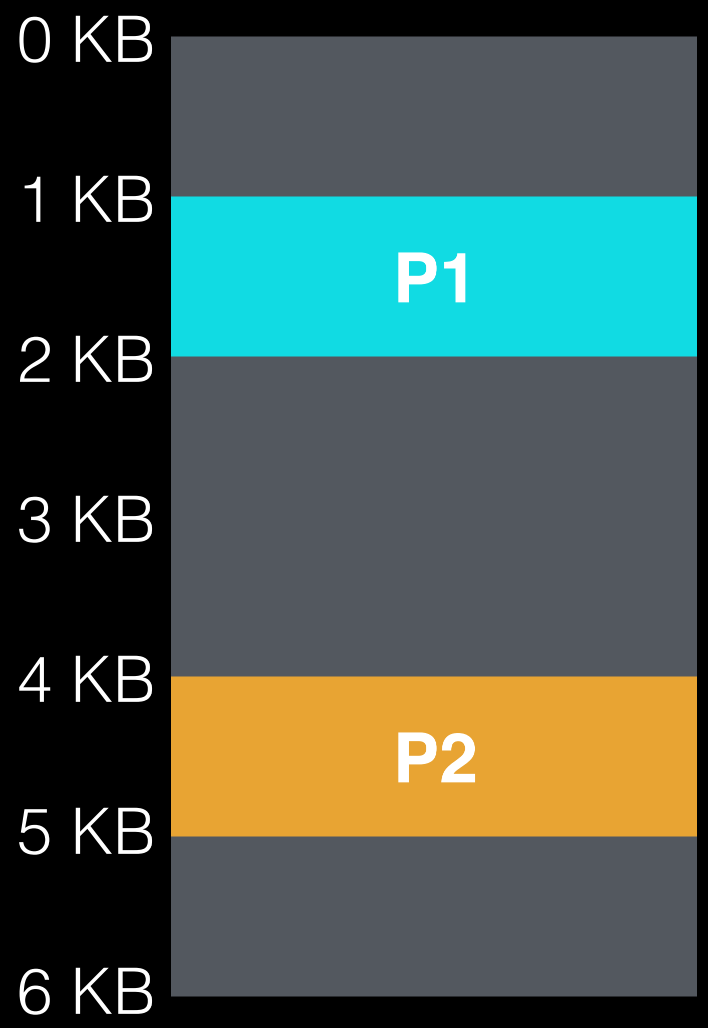
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1



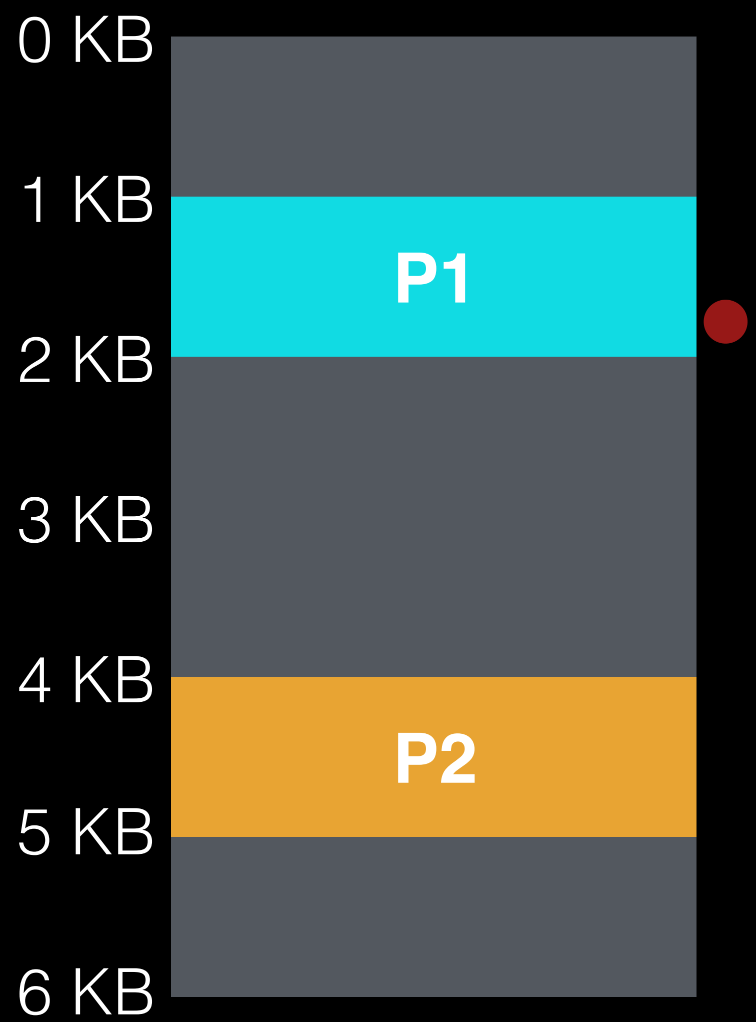
Virtual	**Physical**
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	

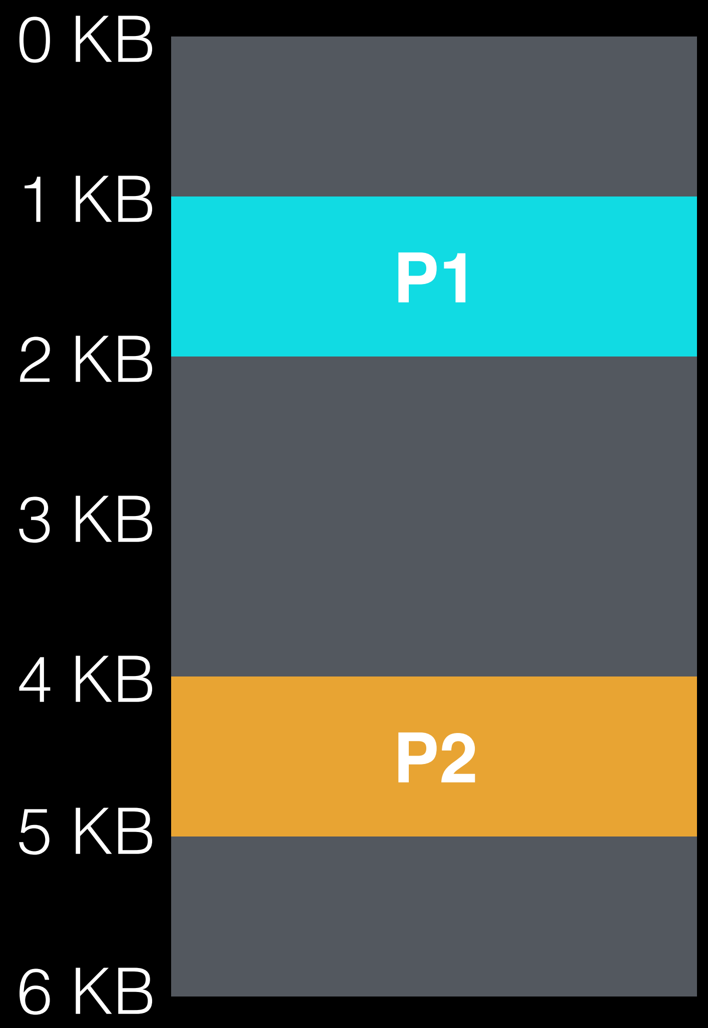


Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1

Who Controls the Base Register?

Who should do translation with base register?
(1) process, (2) OS, or (3) HW

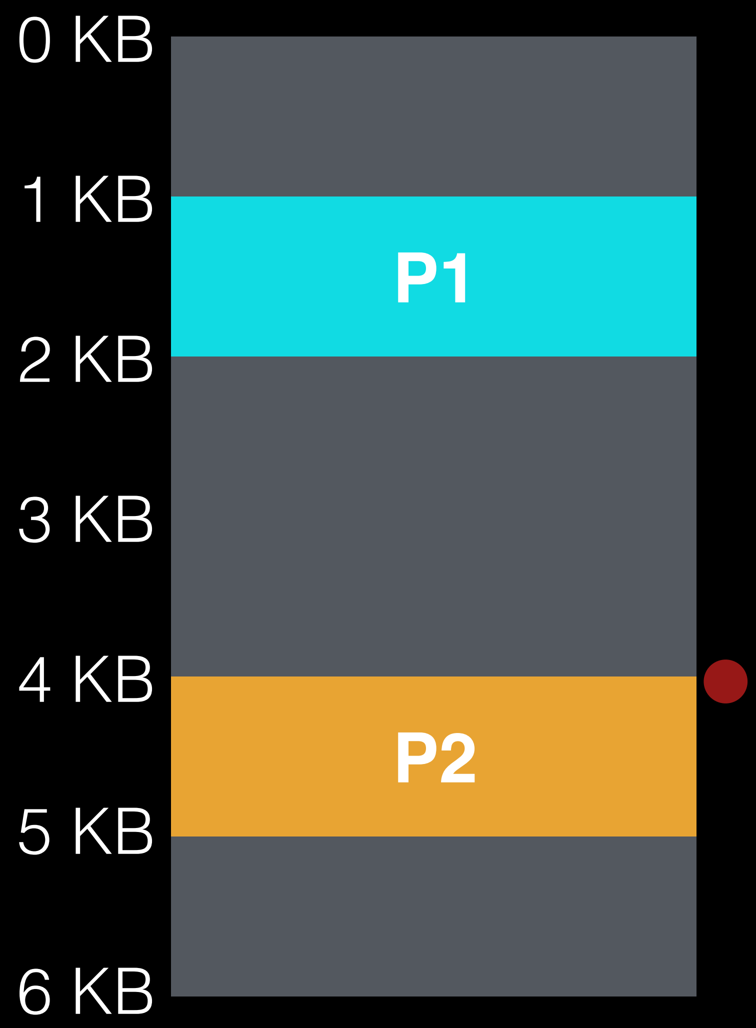
Who should modify the base register?
(1) process, (2) OS, or (3) HW



Virtual	**Physical**
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1

Can P2 hurt P1?

Can P1 hurt P2?



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	store 4096, R1

Can P2 hurt P1?

Can P1 hurt P2?

Problem: How to Run Multiple Processes?

Approaches (covered today):

- Time Sharing
- Static Relocation
- Base
- Base+Bounds
- Segmentation

Base+Bounds

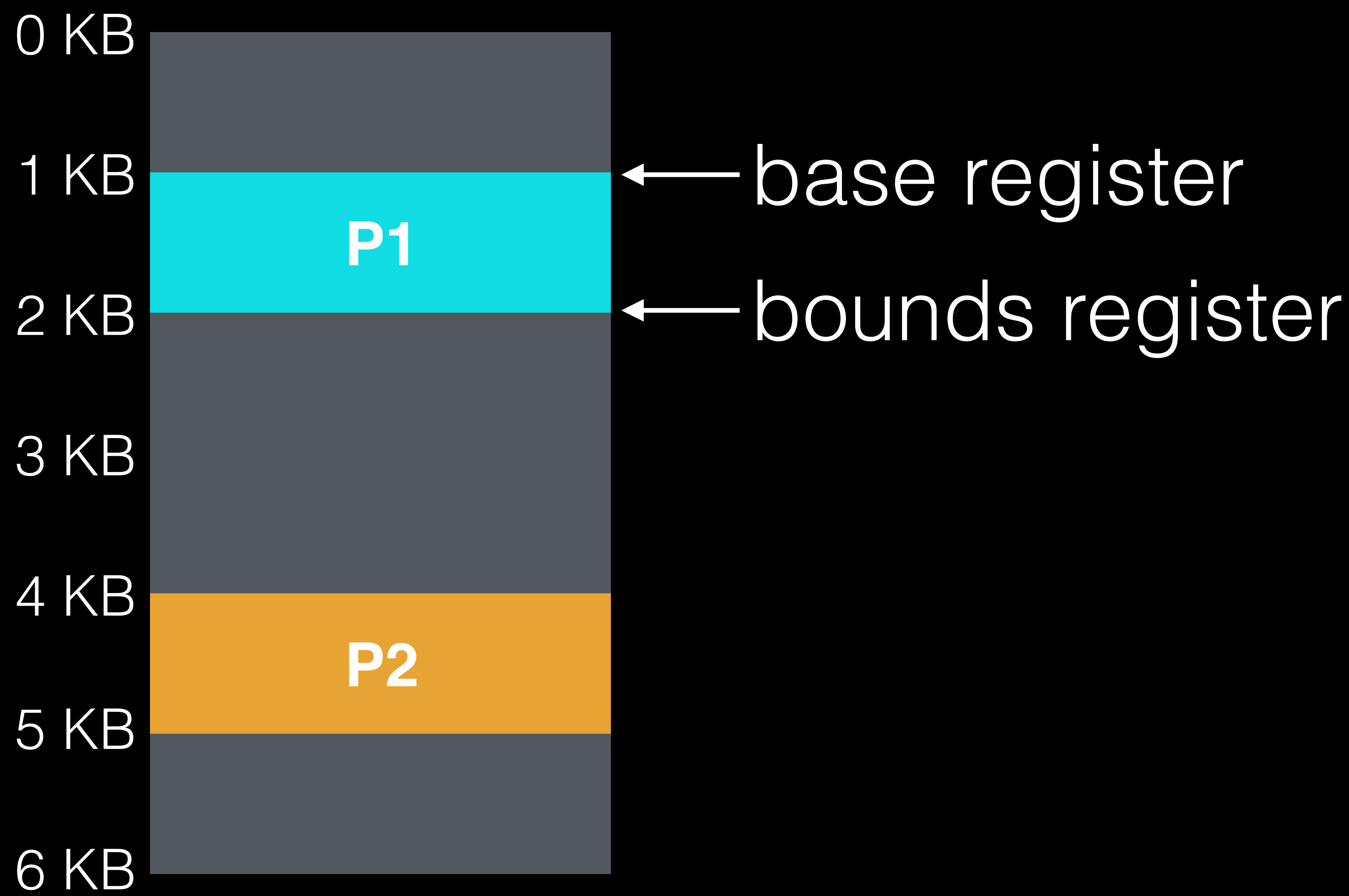
Idea: contain the address space with a bounds register marking the largest physical address

Base register: smallest physical addr

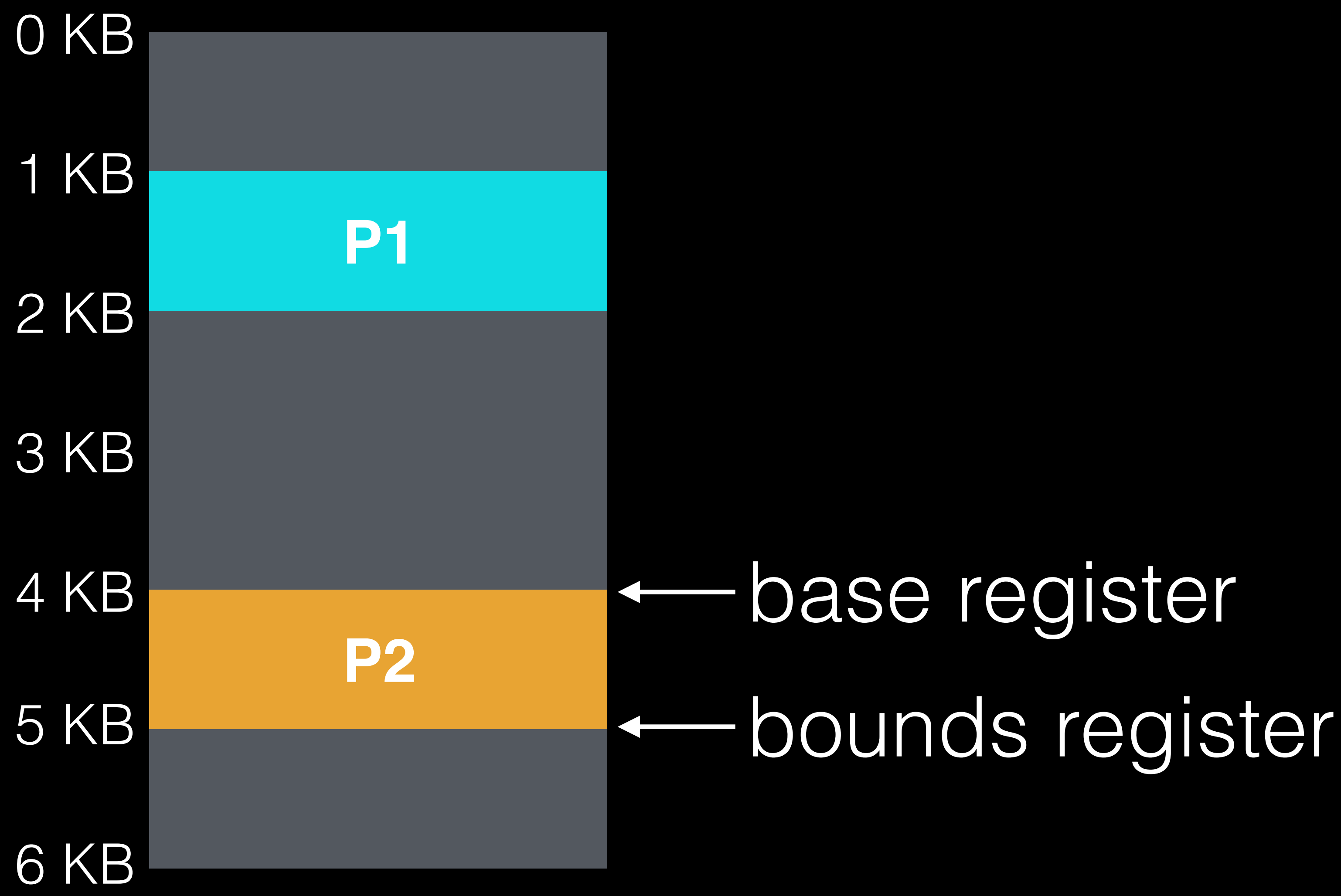
Bounds register: largest physical addr

OSTEP!

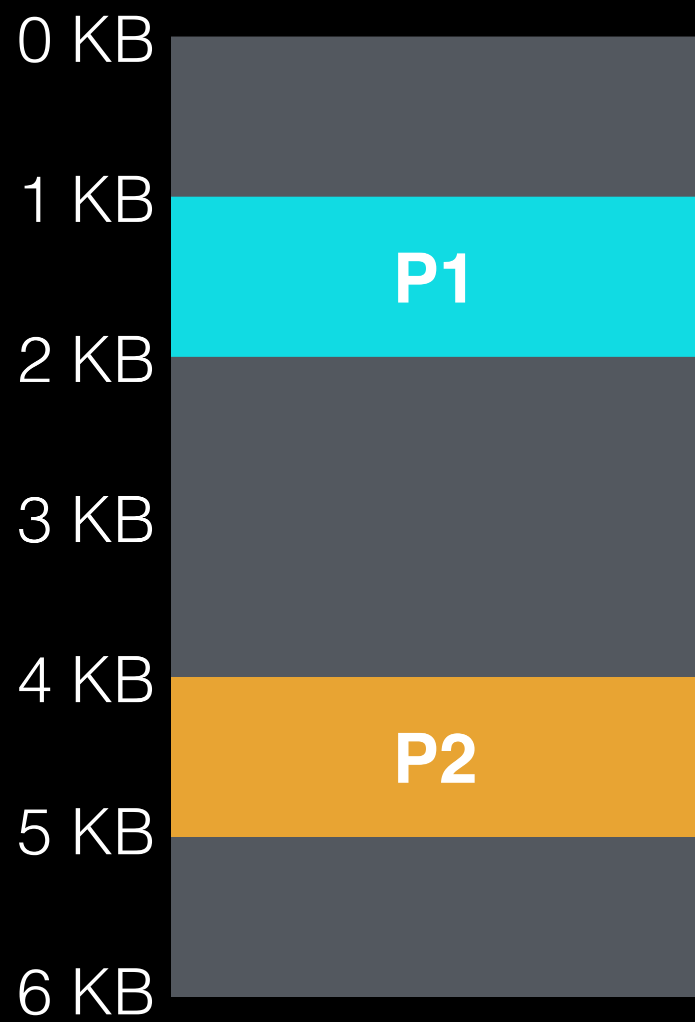
What happens if you load/store after bounds?



P1 is running

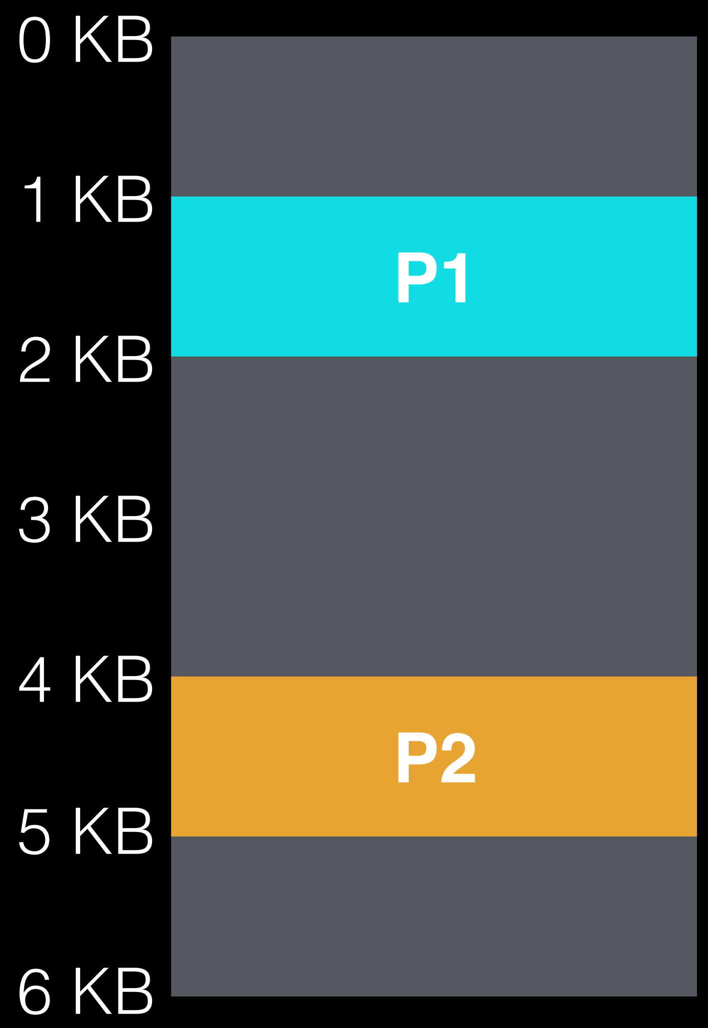


P2 is running



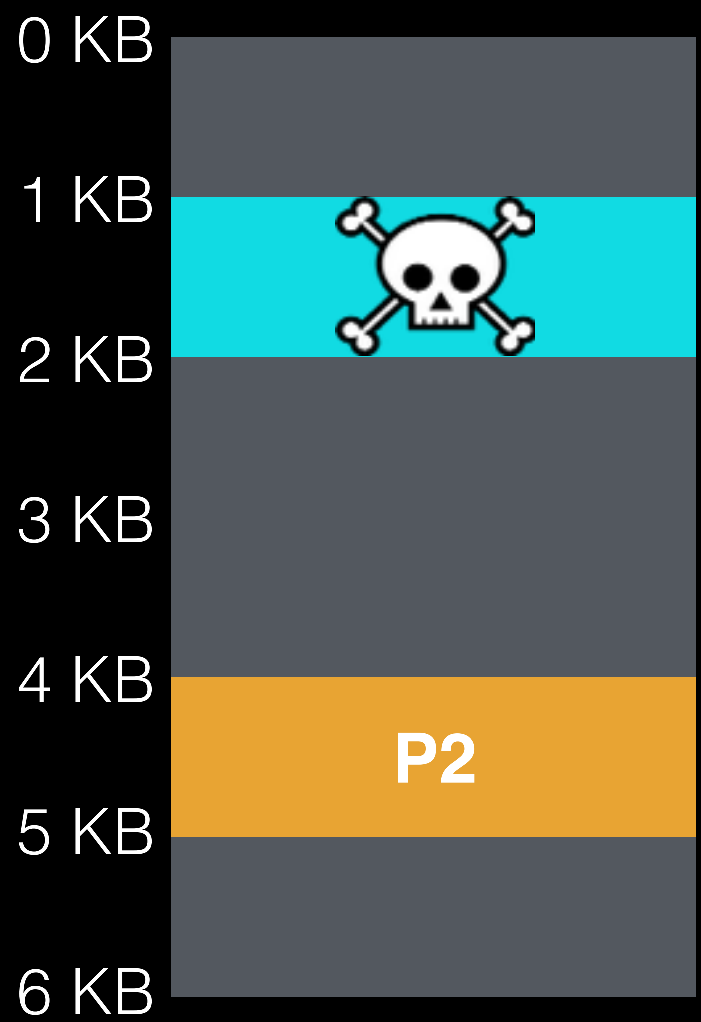
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	

Can P1 hurt P2?



Virtual	**Physical**
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	**interrupt OS!**

Can P1 hurt P2?



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	**interrupt OS!**

Can P1 hurt P2?

Base+Bounds Pros/Cons

Pros?

Cons?

Base+Bounds Pros/Cons

Pros?
- fast + simple
- little bookkeeping overhead (2 registers / proc)

Cons?
- not flexible
- wastes memory for large address spaces

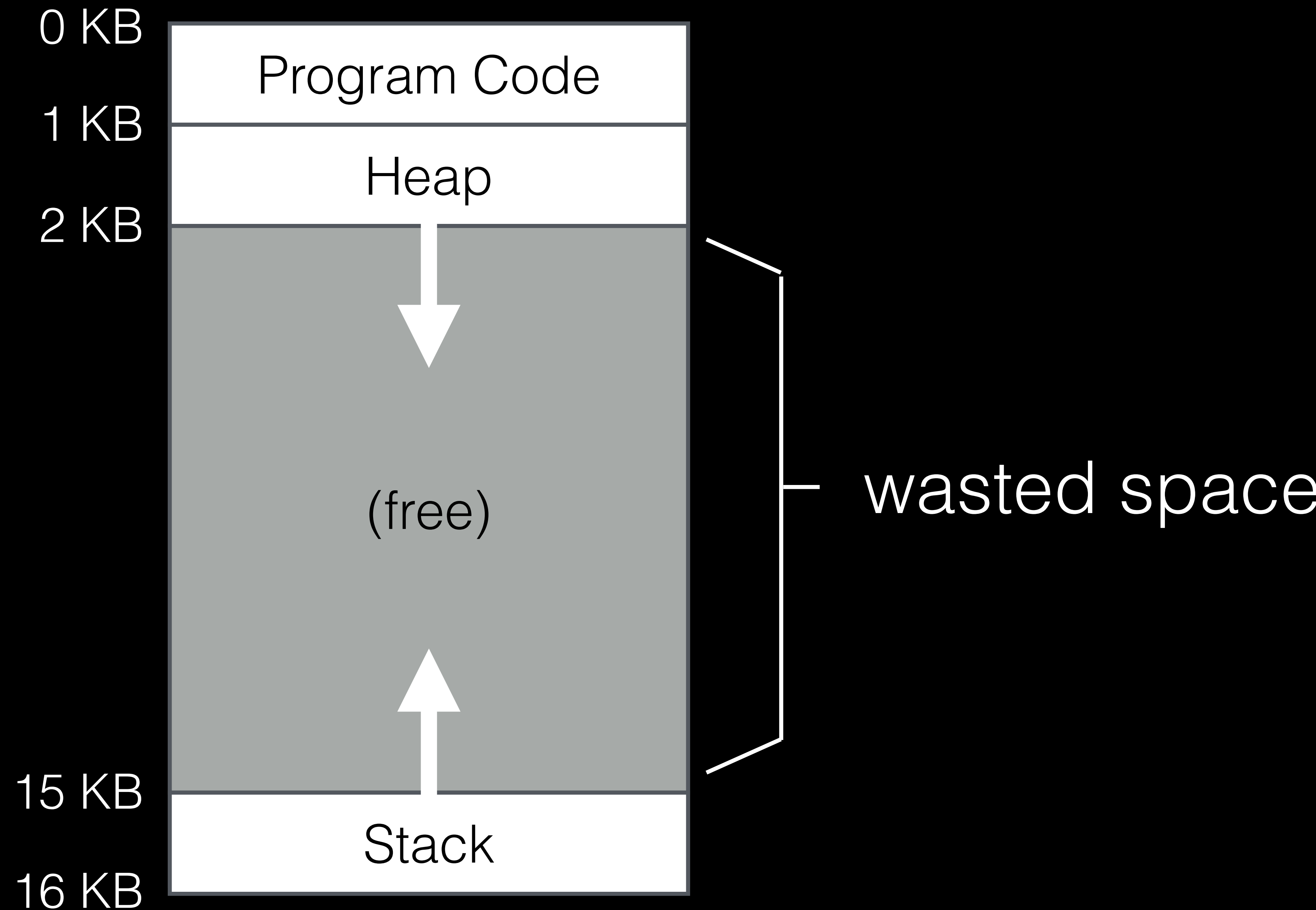
Base+Bounds Pros/Cons

Pros?

- fast + simple
- little bookkeeping overhead (2 registers / proc)

Cons?

- not flexible
- wastes memory for large address spaces


0 KB
Program Code
1 KB
Heap
2 KB
(free)
wasted space
15 KB
Stack
16 KB

Problem: How to Run Multiple Processes?

Approaches (covered today):

- Time Sharing
- Static Relocation
- Base
- Base+Bounds
- Segmentation

Segmentation

Idea: generalize base+bounds

Each base+bound pair is a *segment*

Use different segments for heap and memory
- how does this help?
- requires more registers!

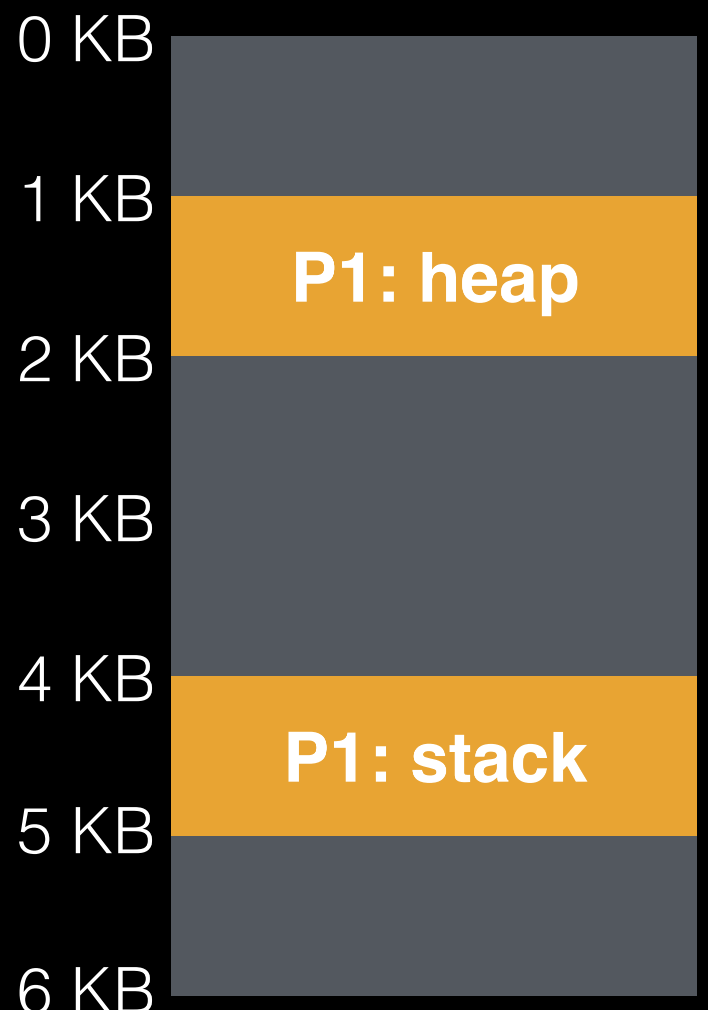
Resize segments as needed
- how does this help?

Multi-segment translation

One (broken) approach:

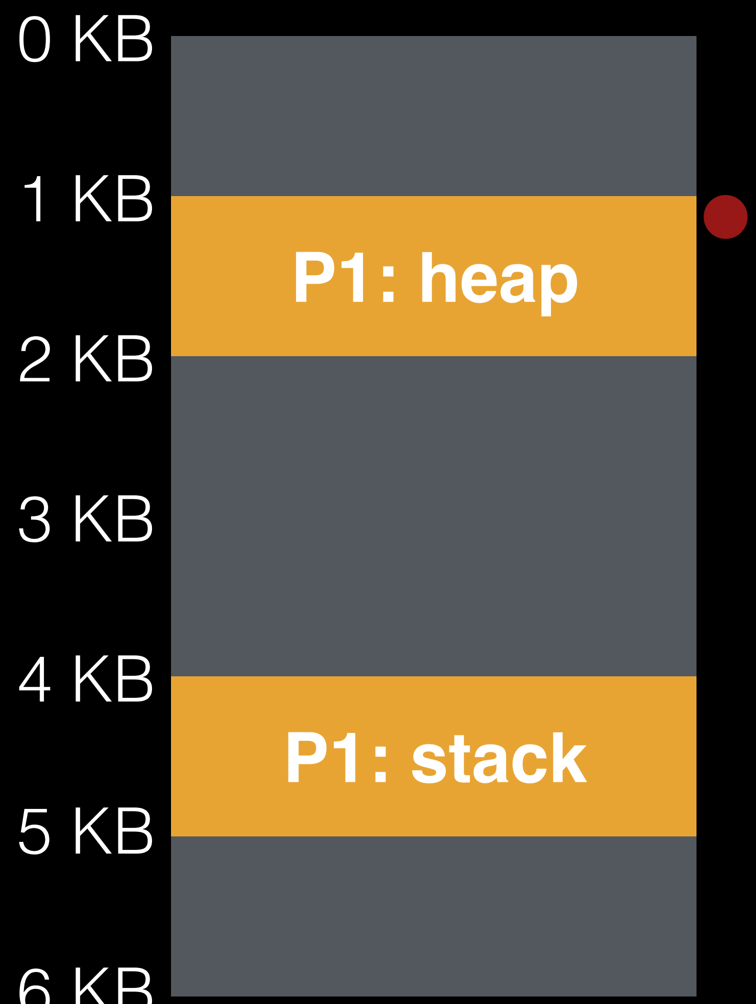
- have no gaps in virtual addresses
- map as many low addresses to the first segment as possible, then as many as possible to the second (on so on)

A tricky example



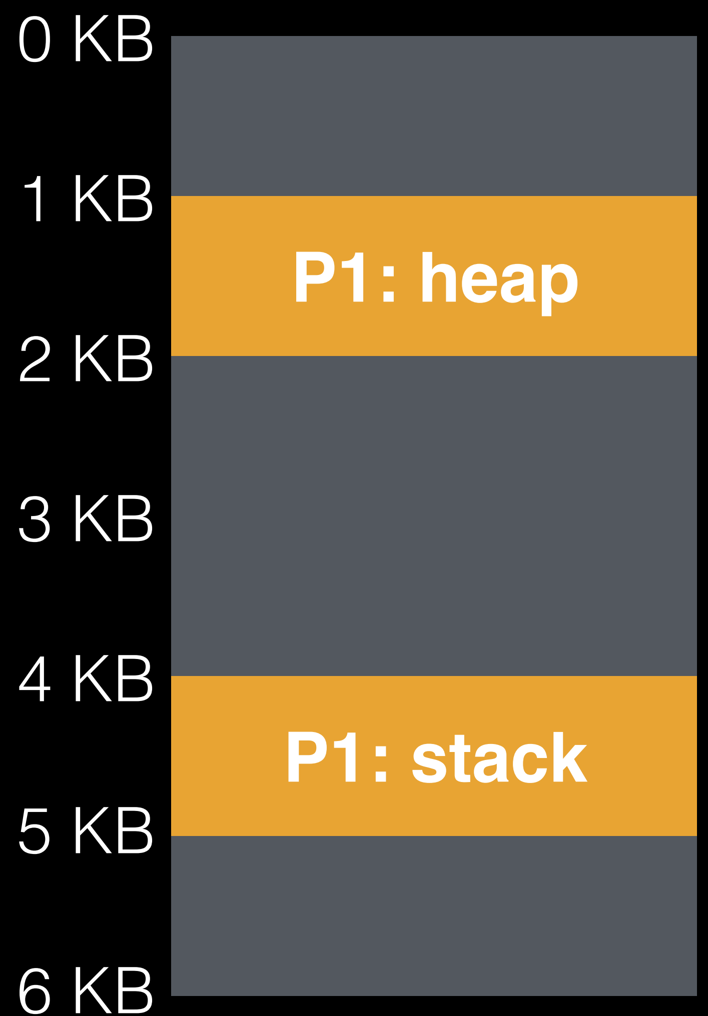
Virtual	Physical
P1: load 100, R1	

A tricky example



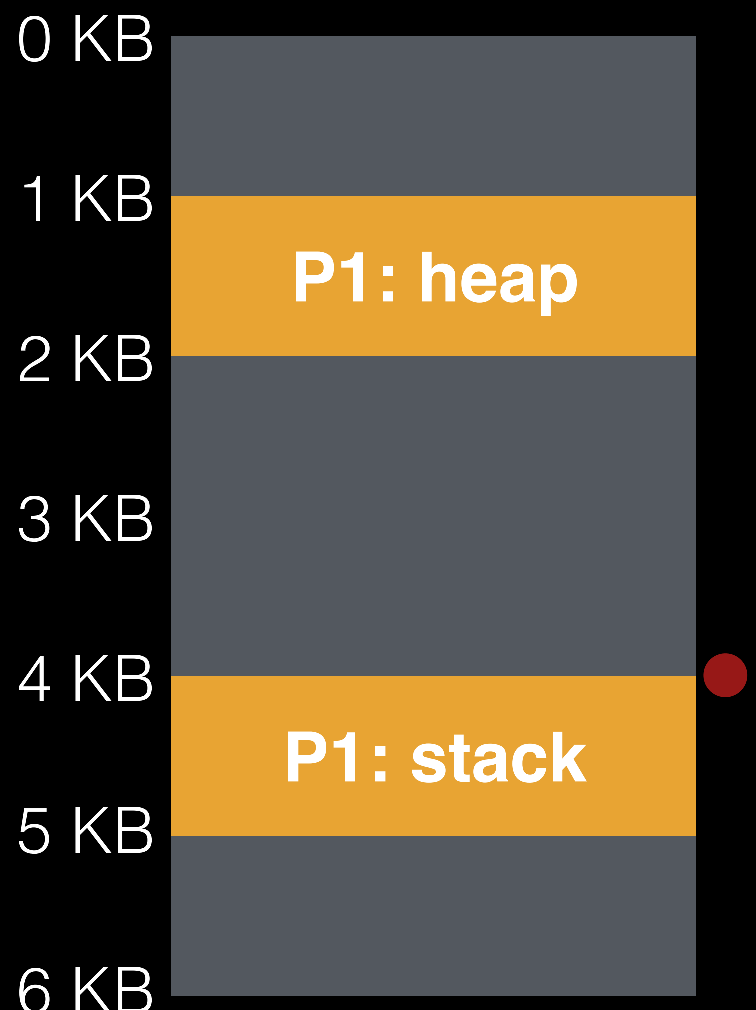
Virtual	Physical
P1: load 100, R1	load 1124, R1

A tricky example



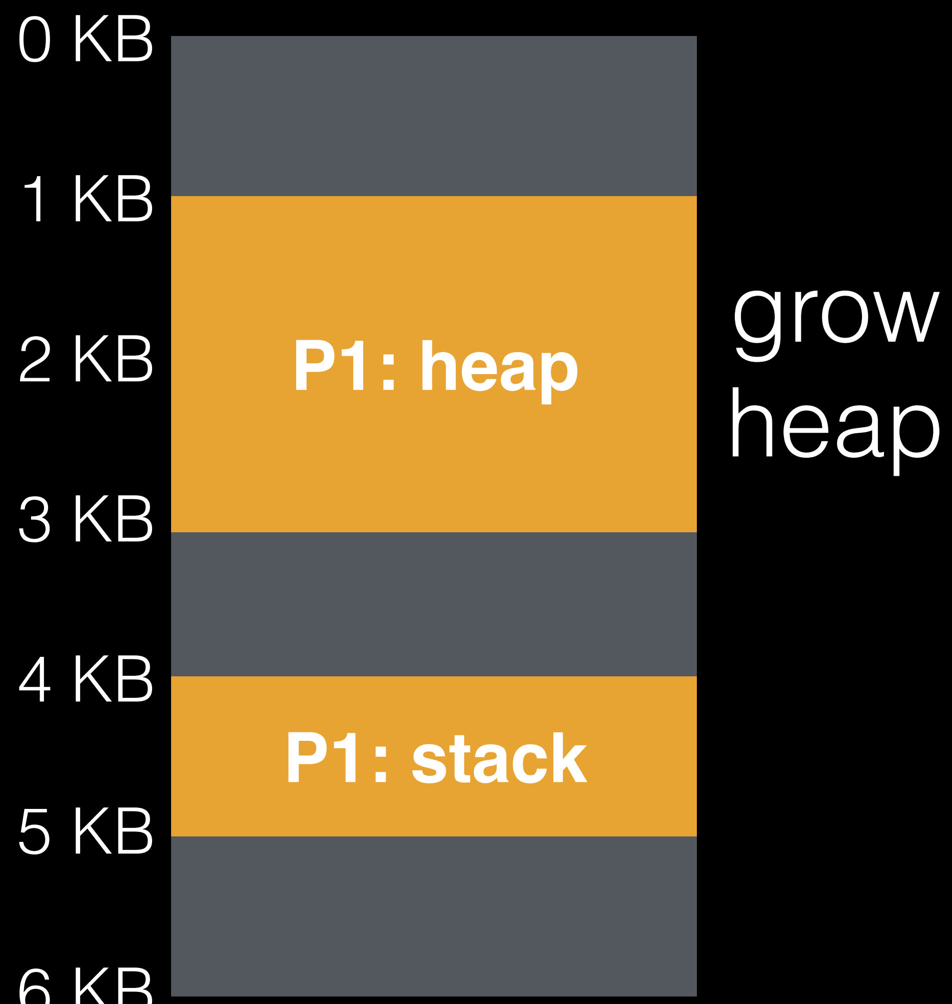
Virtual	Physical
P1: load 100, R1	load 1124, R1
P1: load 1024, R1	

A tricky example



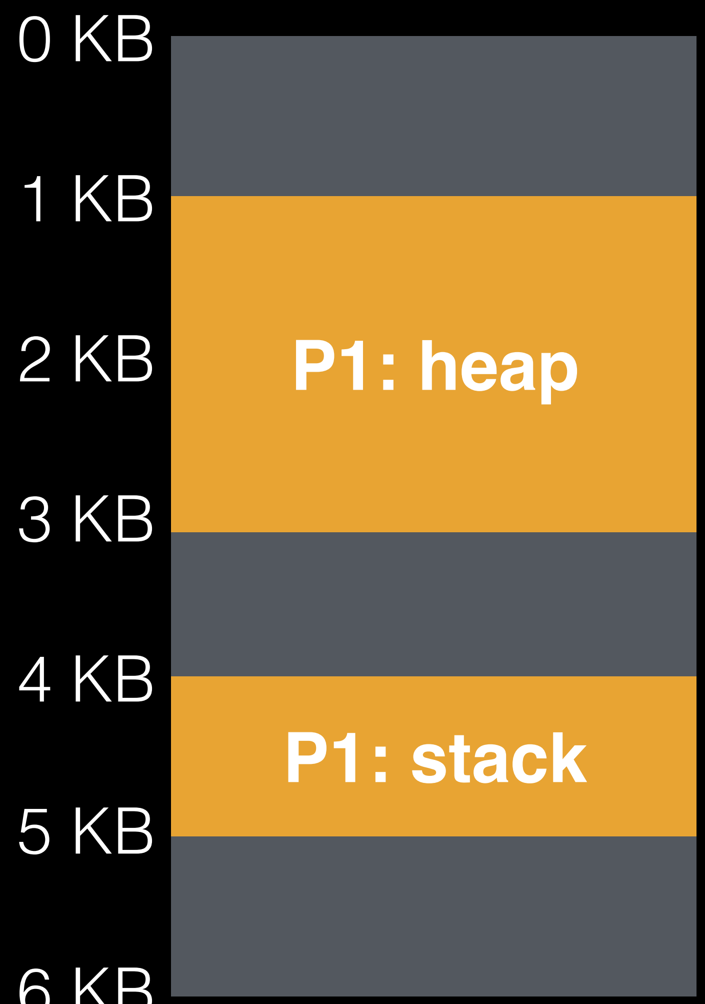
Virtual	Physical
P1: load 100, R1	load 1124, R1
P1: load 1024, R1	load 4096, R1

A tricky example



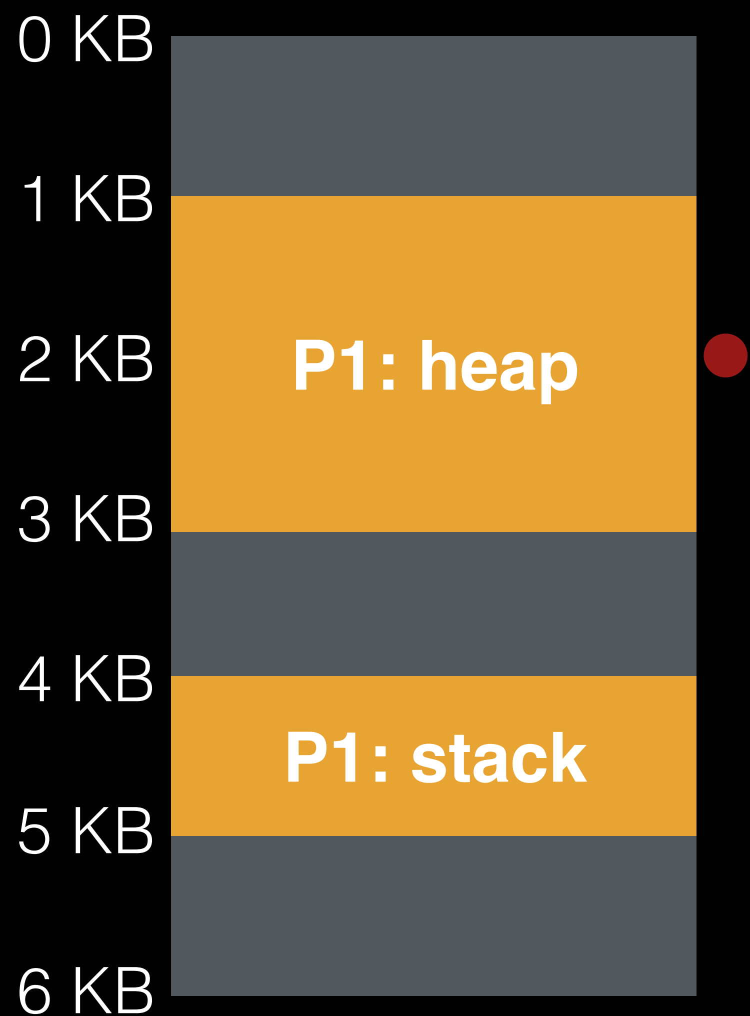
Virtual	**Physical**
P1: load 100, R1	load 1124, R1
P1: load 1024, R1	load 4096, R1

A tricky example



Virtual	**Physical**
P1: load 100, R1	load 1124, R1
P1: load 1024, R1	load 4096, R1
P1: load 1024, R1	

A tricky example



Virtual	Physical
P1: load 100, R1	load 1124, R1
P1: load 1024, R1	load 4096, R1
P1: load 1024, R1	load 2048, R1

Multi-segment translation

One (correct) approach:
- break virtual addresses into two parts
- one part indicates segment
- one part indicates offset within segment

Virtual Address

For example, say addresses are 14 bits.
Use 2 bits for segment, 12 bits for offset

An address might look like 201E

Virtual Address

For example, say addresses are 14 bits.
Use 2 bits for segment, 12 bits for offset

An address might look like 2 01E

segment 2 offset 30

Virtual Address

For example, say addresses are 14 bits.
Use 2 bits for segment, 12 bits for offset

An address might look like 2 01E

Choose some segment numbering, such as

- 0: code+data
- 1: heap
- 2: stack

What is the segment/offset?

Segment numbers:

0: code+data

1: heap

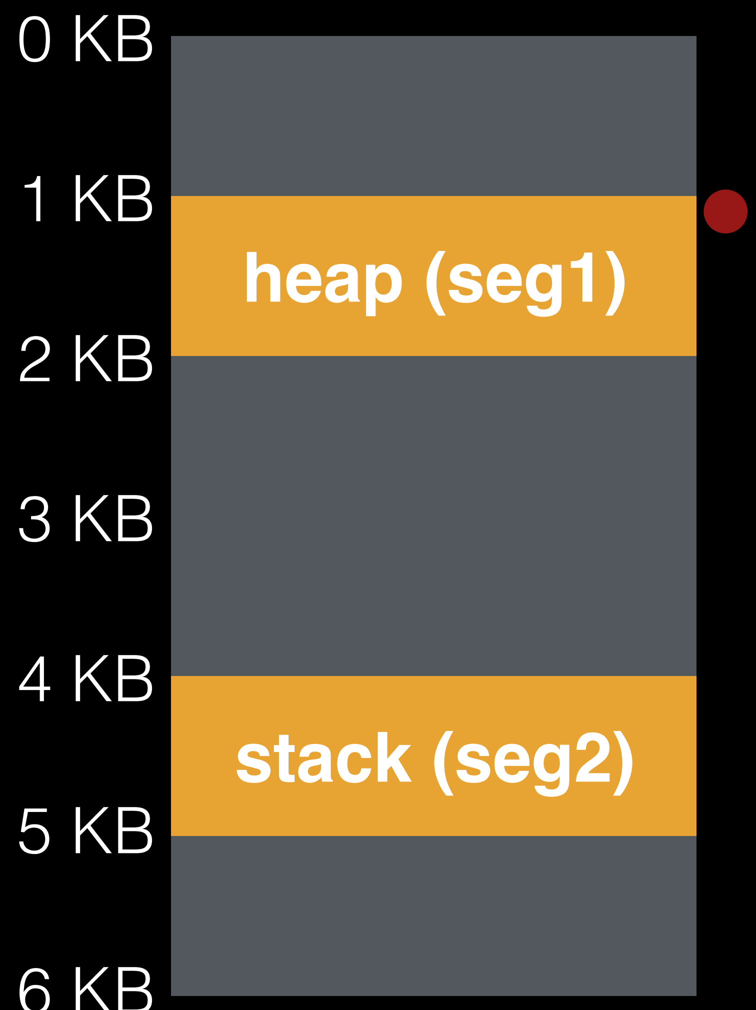
2: stack

10 0000 0001 0001	(binary)
110A	(hex)
4096	(decimal)

Virtual	**Physical**
load 0x2010, R1	

Virtual	Physical
load 0x2010, R1	4KB + 16

Virtual	Physical
load 0x2010, R1	4KB + 16
load 0x1010, R1	



Virtual	**Physical**
load 0x2010, R1	4KB + 16
load 0x1010, R1	1KB + 16

Virtual	Physical
load 0x2010, R1	4KB + 16
load 0x1010, R1	1KB + 16
load 0x1100, R1	

Virtual	Physical
load 0x2010, R1	4KB + 16
load 0x1010, R1	1KB + 16
load 0x1100, R1	1KB + 256

Virtual	Physical
load 0x2010, R1	4KB + 16
load 0x1010, R1	1KB + 16
load 0x1100, R1	1KB + 256
load 0x1400, R1	

Virtual	Physical
load 0x2010, R1	4KB + 16
load 0x1010, R1	1KB + 16
load 0x1100, R1	1KB + 256
load 0x1400, R1	interrupt OS!

Stack Growth Problem

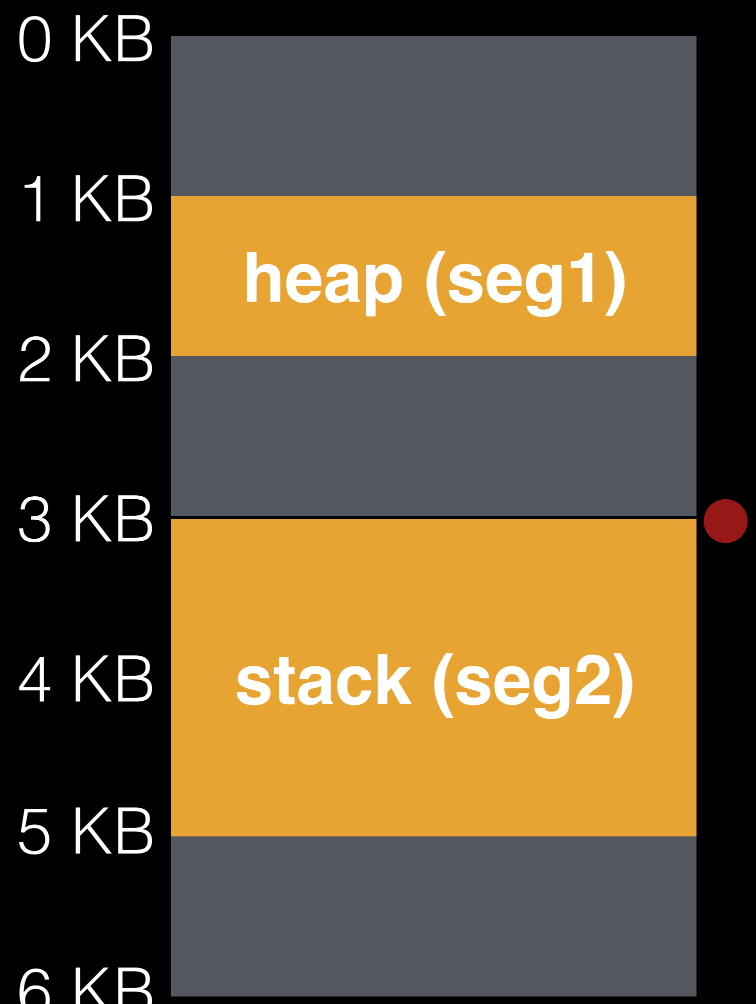
Example…

Virtual	Physical
load 0x2000, R1	

Virtual	Physical
load 0x2000, R1	4KB

Virtual	Physical
load 0x2000, R1	4KB

Virtual	Physical
load 0x2000, R1	4KB
load 0x2000, R1	



Virtual	Physical
load 0x2000, R1	4KB
load 0x2000, R1	3KB

Stack Growth Problem

Example…

Problem: phys = virt_offset + base_reg
phys is anchored to base_reg, which moves

Stack Growth Problem

Example…

Problem: phys = virt_offset + base_reg
phys is anchored to base_reg, which moves

Solution: anchor heap segment to bounds_reg:
phys = bounds_reg - (max_offset - virt_offset)

Stack Growth Problem

Example…

Problem: phys = virt_offset + base_reg
phys is anchored to base_reg, which moves

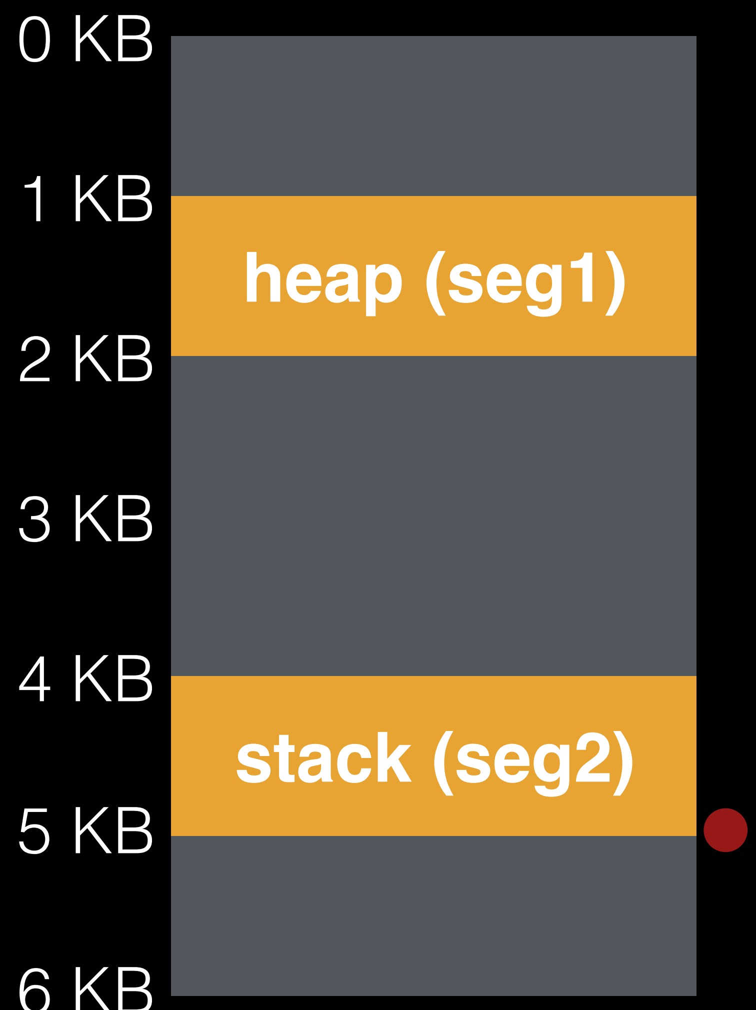
Solution: anchor heap segment to bounds_reg:
phys = bounds_reg - (max_offset - virt_offset)

Example (with max_offset = FFF)…

stack's max_offset = FFF

Virtual	Physical
load 0x2FFE, R1	

stack's max_offset = FFF

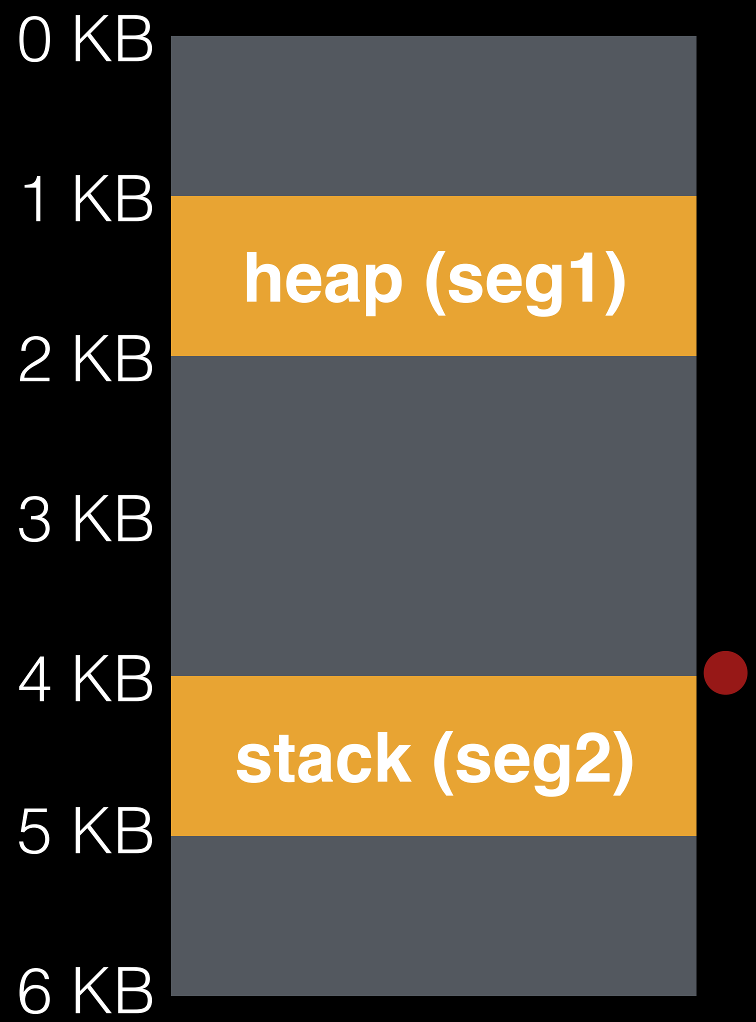


Virtual	Physical
load 0x2FFE, R1	5KB - 1

stack's max_offset = FFF

Virtual	Physical
load 0x2FFE, R1	5KB - 1
load 0x2BFF, R1	

stack's max_offset = FFF



Virtual	Physical
load 0x2FFE, R1	5KB - 1
load 0x2BFF, R1	4KB

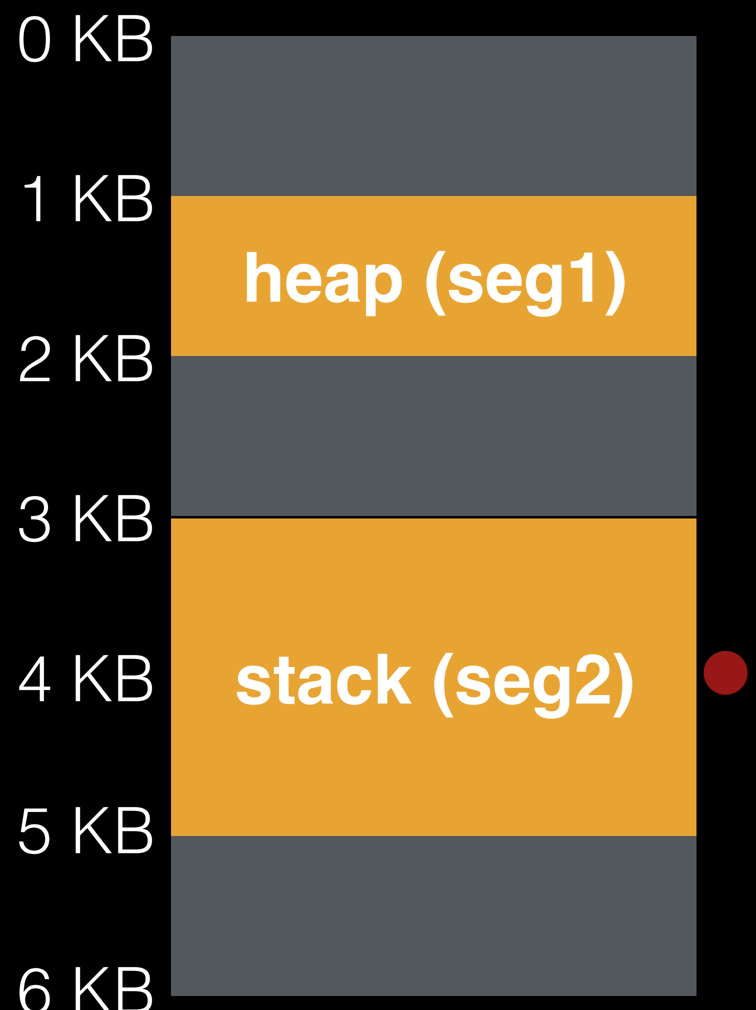
stack's max_offset = FFF

Virtual	Physical
load 0x2FFE, R1	5KB - 1
load 0x2BFF, R1	4KB

stack's max_offset = FFF

Virtual	Physical
load 0x2FFE, R1	5KB - 1
load 0x2BFF, R1	4KB
load 0x2BFF, R1	

stack's max_offset = FFF



Virtual	Physical
load 0x2FFE, R1	5KB - 1
load 0x2BFF, R1	4KB
load 0x2BFF, R1	4KB

Translation Summary

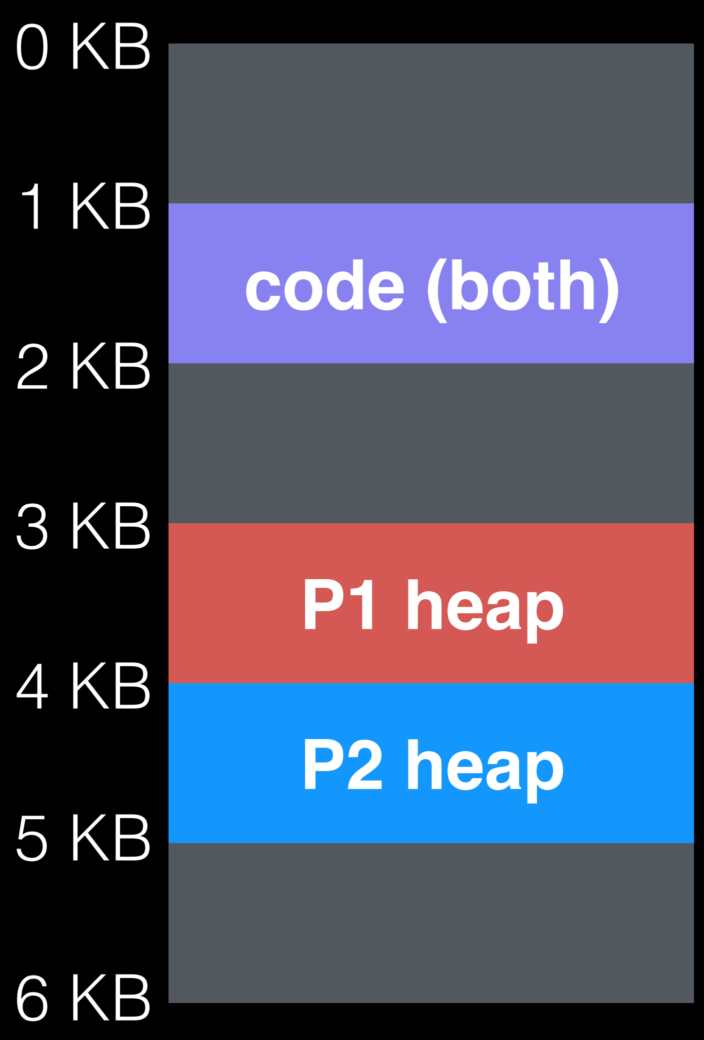
Heap: phys = base_reg + virt_offset
Stack: phys = bounds_reg - (max_offset - virt_offset)

Anchors:
- for heap, anchor smallest address to base register
- for stack, anchor biggest address to bounds register

Code Sharing

Idea: make base/bounds for the code of several processes point to the same physical mem

Careful: need extra protection!



Segmentation Pros/Cons

Pros?

- supports sparse address space
- code sharing
- fine grained protection

Cons?

- external fragmentation

Conclusion

HW+OS work together to trick processes, giving the illusion of private memory

Adding CPU registers for base+bounds extends LDE, so translation is fast (does not always need OS)

Next time: solve fragmentation with paging