Department of Physics,
Computer Science & Engineering

CPSC 410 – Operating Systems I

Virtualizing Memory: Memory Virtualization

Keith Perkins

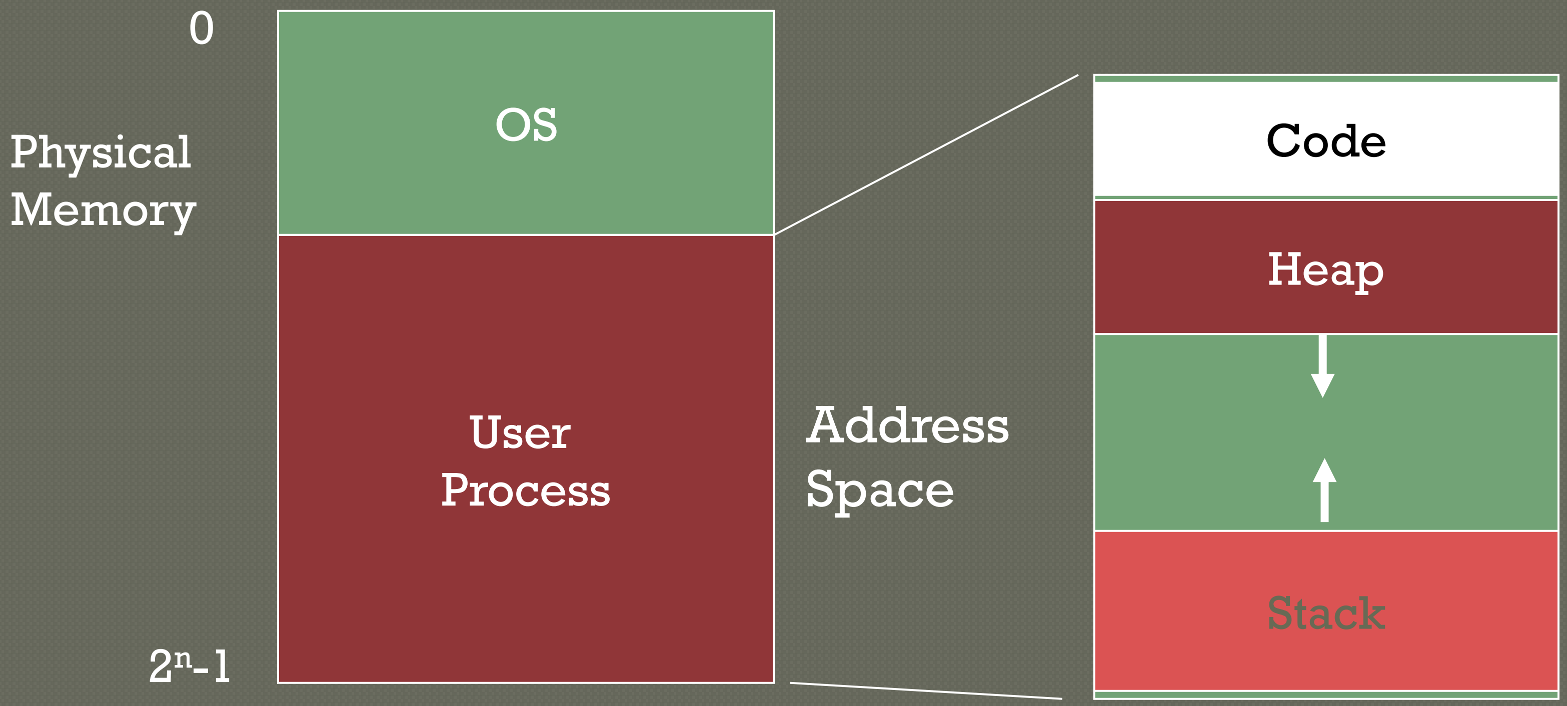
Adapted from “CS 537 Introduction to Operating Systems” Arpaci-Dusseau

Questions answered in this lecture:

- What is in the address space of a process (review)?
- What are the different ways that that OS can virtualize memory?
- Time sharing, static relocation, dynamic relocation
- (base, base + bounds, segmentation)
- What hardware support is needed for dynamic relocation?

Motivation for Virtualization

Uniprogramming: One process runs at a time



Disadvantages:

- Only one process runs at a time
- Process can destroy OS

Multiprogramming Goals

Transparency

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

Protection

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

Efficiency

- Do not waste memory resources (minimize fragmentation)

Sharing

- Cooperating processes can share portions of address space

Abstraction: Address SPace

Address space: Each process has set of addresses that map to bytes

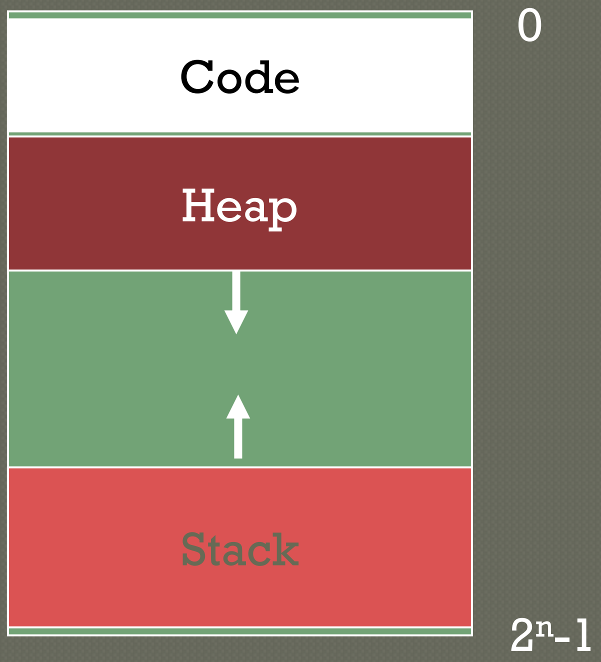
Problem:

How can OS provide illusion of private address space to each process?

Review: What is in an address space?

Address space has static and dynamic components

- Static: Code and some global variables
- Dynamic: Stack and Heap



Motivation for Dynamic Memory

Why do processes need dynamic allocation of memory?

- Do not know amount of memory needed at compile time
- Must be pessimistic when allocate memory statically
  - If you allocate enough for worst possible case then storage is used inefficiently

Recursive procedures

- Do not know how many times procedure will be nested

Complex data structures: lists and trees

- `struct my_t *p = (struct my_t *)malloc(sizeof(struct my_t));`

Two types of dynamic allocation

- Stack
- Heap

Stack Organization

Definition: Memory is freed in opposite order from allocation

```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```

Simple and efficient implementation:

Pointer separates allocated and freed space

- Allocate: Increment pointer
- Free: Decrement pointer

No fragmentation

Stack Organization

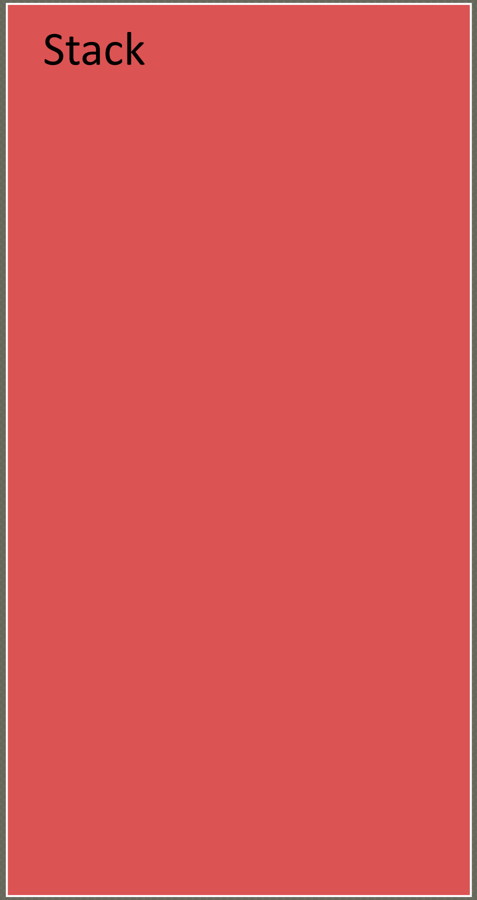
Definition: Memory is freed in opposite order from allocation

```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```

Simple and efficient implementation:
Pointer separates allocated and freed space

- Allocate: Increment pointer
- Free: Decrement pointer

No fragmentation



Stack

Stack Organization

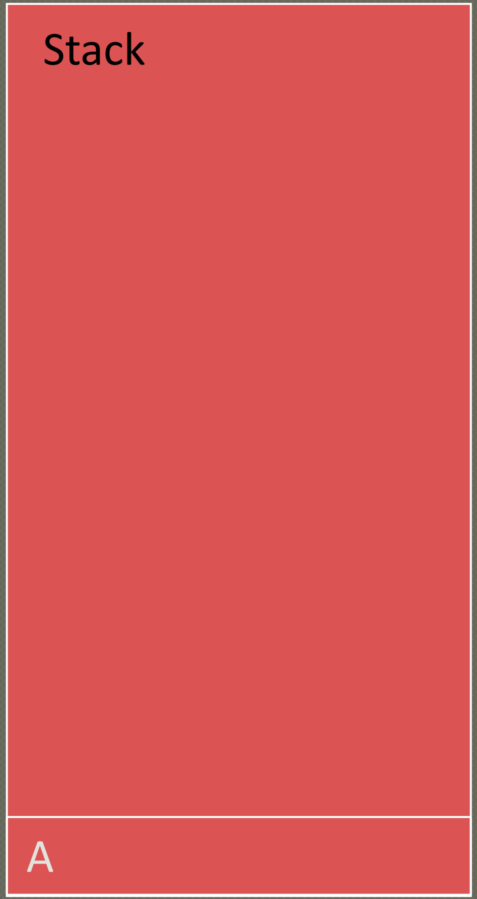
Definition: Memory is freed in opposite order from allocation

```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```

Simple and efficient implementation:
Pointer separates allocated and freed space

- Allocate: Increment pointer
- Free: Decrement pointer

No fragmentation



Stack Organization

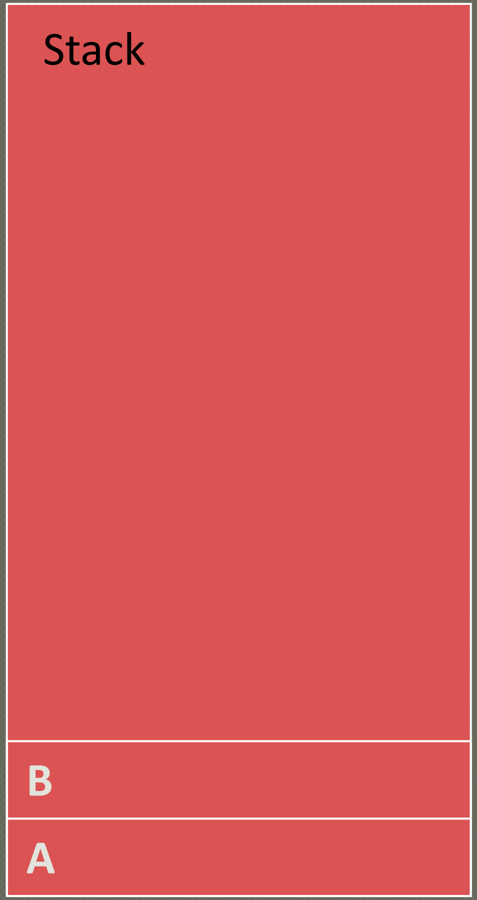
Definition: Memory is freed in opposite order from allocation

```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```

Simple and efficient implementation:
Pointer separates allocated and freed space

- Allocate: Increment pointer
- Free: Decrement pointer

No fragmentation



Stack Organization

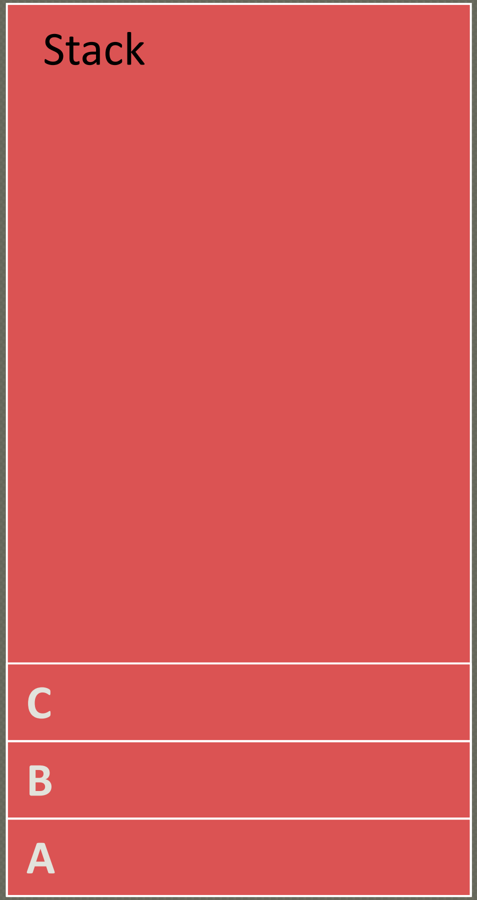
Definition: Memory is freed in opposite order from allocation

```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```

Simple and efficient implementation:
Pointer separates allocated and freed space

- Allocate: Increment pointer
- Free: Decrement pointer

No fragmentation



Stack Organization

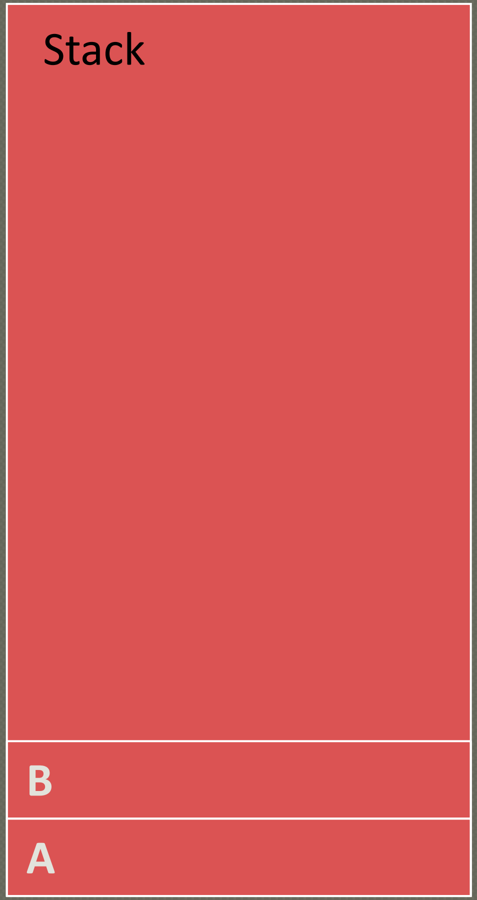
Definition: Memory is freed in opposite order from allocation

```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```

Simple and efficient implementation:
Pointer separates allocated and freed space

- Allocate: Increment pointer
- Free: Decrement pointer

No fragmentation



Stack Organization

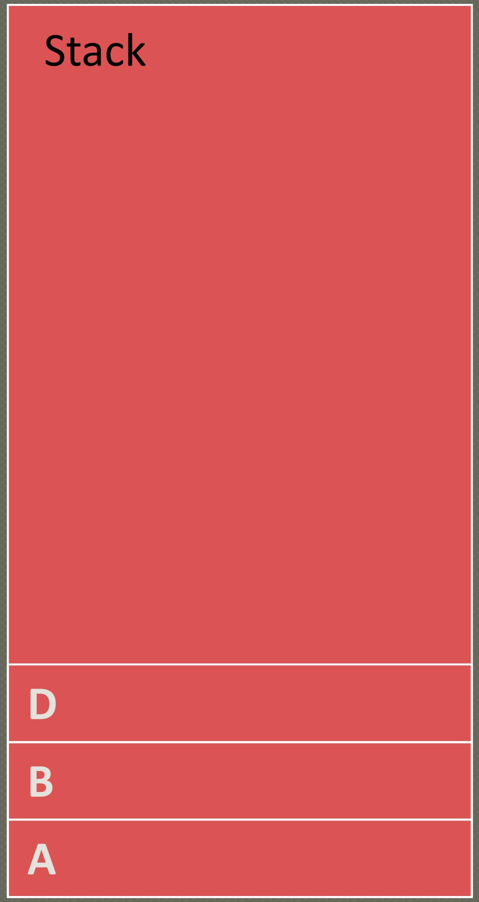
Definition: Memory is freed in opposite order from allocation

```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```

Simple and efficient implementation:
Pointer separates allocated and freed space

- Allocate: Increment pointer
- Free: Decrement pointer

No fragmentation



Stack

D

B

A

Stack Organization

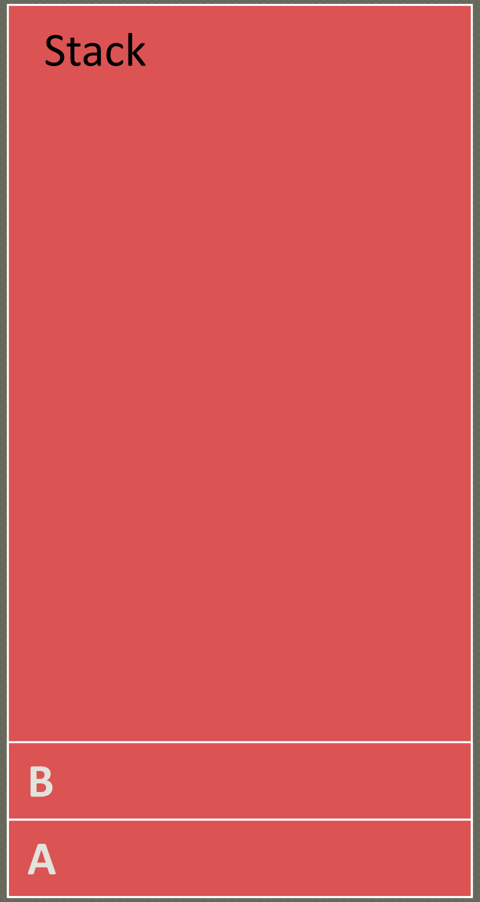
Definition: Memory is freed in opposite order from allocation

```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```

Simple and efficient implementation:
Pointer separates allocated and freed space

- Allocate: Increment pointer
- Free: Decrement pointer

No fragmentation

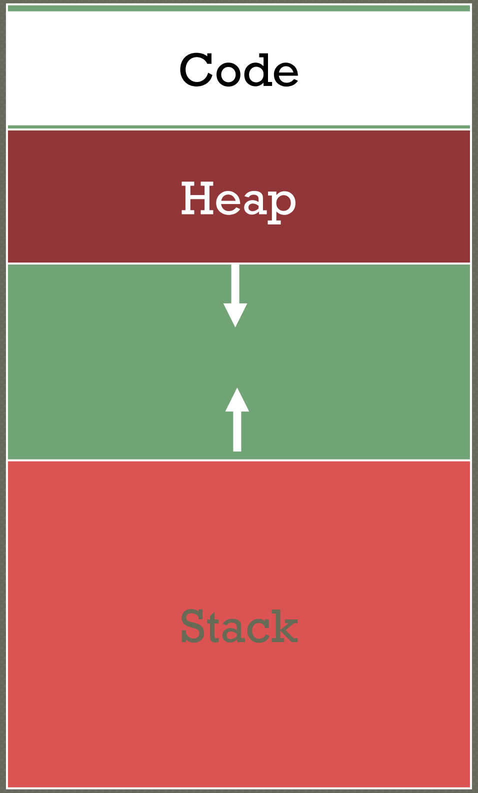


A stack acts as a Last In First Out (LIFO) buffer that grows as needed

Where Are stacks Used?

OS uses stack for procedure call (stack) frames (local variables and parameters)

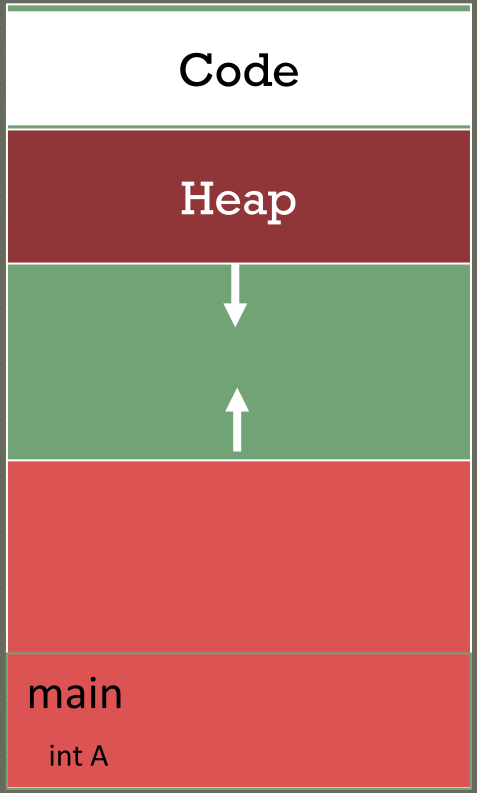
```
main () {
      int A = 0;
      foo (A);
      printf("A: %d\n", A);
}
void foo (int Z) {
      int A = 2;
      Z = 5;
      printf("A: %d Z: %d\n", A, Z);
}
```



Where Are stacks Used?

OS uses stack for procedure call (stack) frames (local variables and parameters)

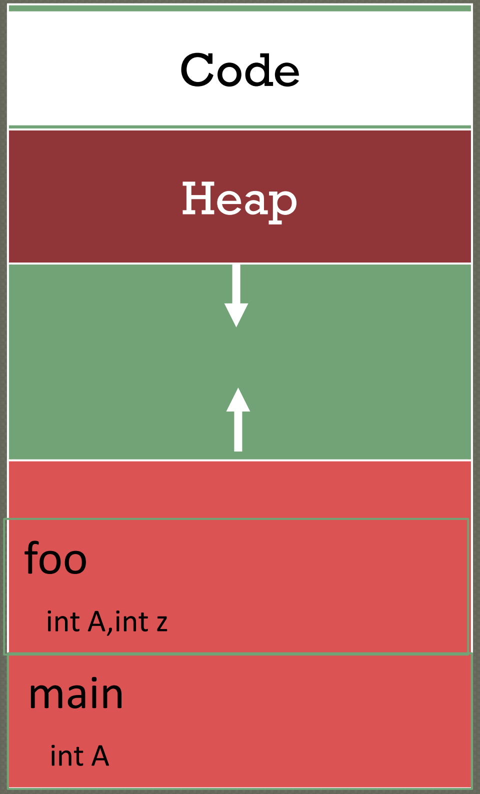
```
main () {
      int A = 0;
      foo (A);
      printf("A: %d\n", A);
}
void foo (int Z) {
      int A = 2;
      Z = 5;
      printf("A: %d Z: %d\n", A, Z);
}
```



Where Are stacks Used?

OS uses stack for procedure call (stack) frames (local variables and parameters)

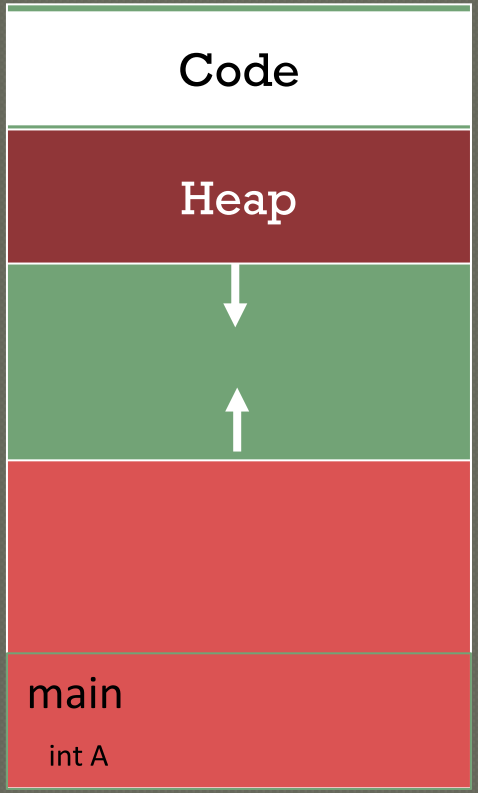
```
main () {
        int A = 0;
        foo (A);
        printf("A: %d\n", A);
}
void foo (int Z) {
        int A = 2;
        Z = 5;
        printf("A: %d Z: %d\n", A, Z);
}
```



Where Are stacks Used?

OS uses stack for procedure call (stack) frames (local variables and parameters)

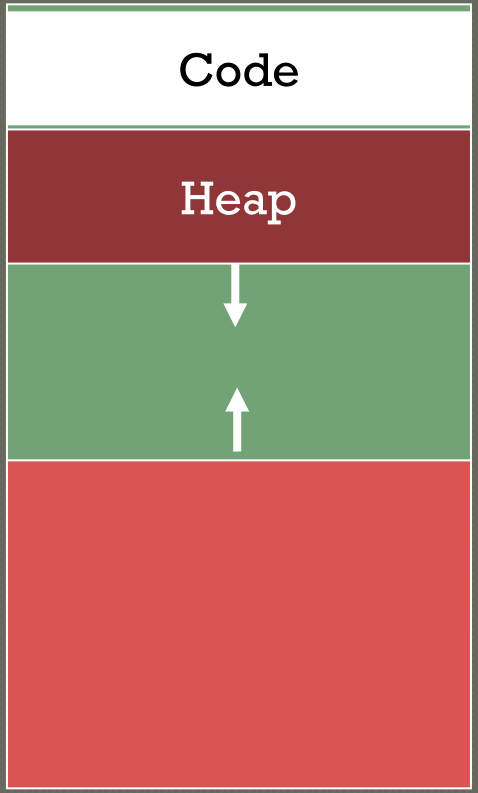
```
main () {
      int A = 0;
      foo (A);
      printf(“A: %d\n”, A);
}
void foo (int Z) {
      int A = 2;
      Z = 5;
      printf(“A: %d Z: %d\n”, A, Z);
}
```



Where Are stacks Used?

OS uses stack for procedure call (stack) frames (local variables and parameters)

```
main () {
        int A = 0;
        foo (A);
        printf(“A: %d\n”, A);
}
void foo (int Z) {
        int A = 2;
        Z = 5;
        printf(“A: %d Z: %d\n”, A, Z);
}
```



Heap Organization

Definition: Allocate from any random location: malloc(), new()

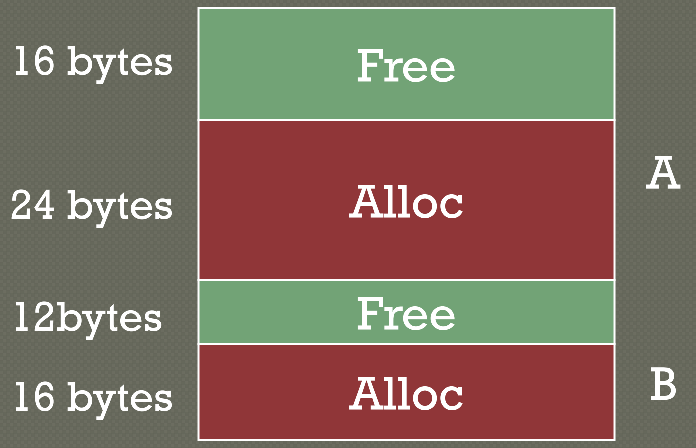
- Heap memory consists of allocated areas and free areas (holes)
- Order of allocation and free is unpredictable

Advantage

- Works for all data structures

Disadvantages

- Allocation can be slow
- End up with small chunks of free space - fragmentation
- Where to allocate 12 bytes? 16 bytes? 24 bytes??



16 bytes	Free	
24 bytes	Alloc	A
12bytes	Free	
16 bytes	Alloc	B

- What is OS’s role in managing heap?
  - OS gives big chunk of free memory to process; library manages individual allocations

Quiz: Match that Address Location

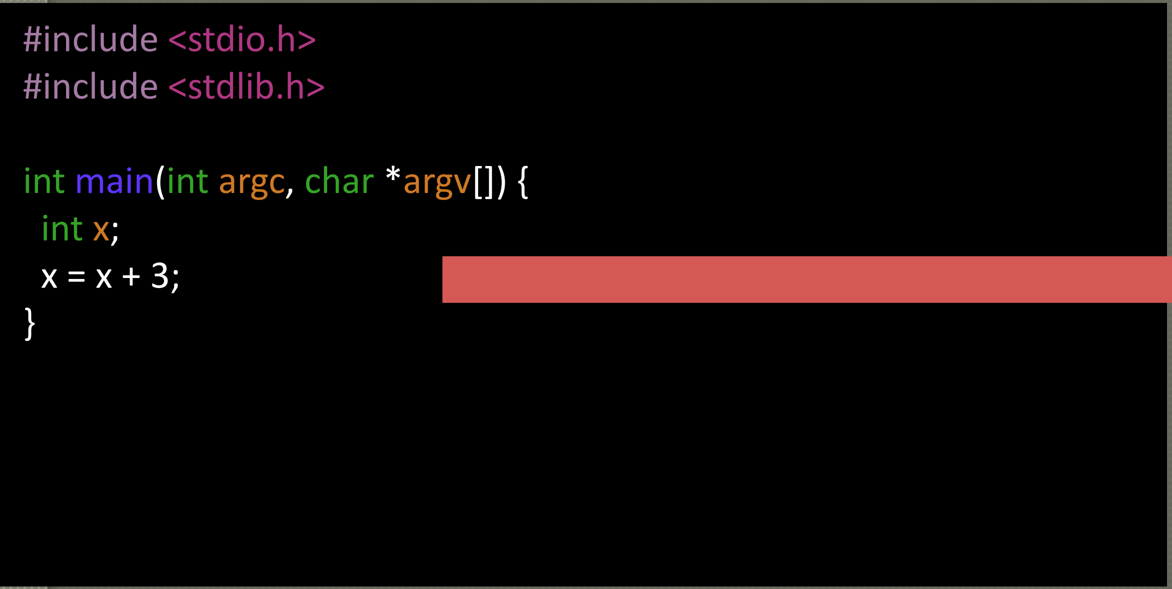
```
int x;
int main(int argc, char *argv[]) {
  int y;
  int *z = malloc(sizeof(int)););
}
```

Possible segments: static data, code, stack, heap

What if no static data segment?

Address	Location
x	Static data → Code
main	Code
y	Stack
z	Stack
*z	Heap

Memory Accesses



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

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

objdump -d demo1.o

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

%**rbp** is the base pointer:
points to base of current stack frame

Quiz: Memory Accesses?

Initial %rip = 0x10
%rbp = 0x200

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

%**rbp** is the base pointer:
points to base of current stack frame

%rip is instruction pointer (or program counter)

Fetch instruction at addr 0x10
Exec:

load from addr 0x208

Fetch instruction at addr 0x13
Exec:

no memory access

Fetch instruction at addr 0x19
Exec:

store to addr 0x208

Memory Accesses to what addresses?
So far they are relative to address in rbp

How to Virtualize Memory?

Problem: How to run multiple processes simultaneously?

Addresses are “hardcoded” into process binaries

How to avoid collisions?

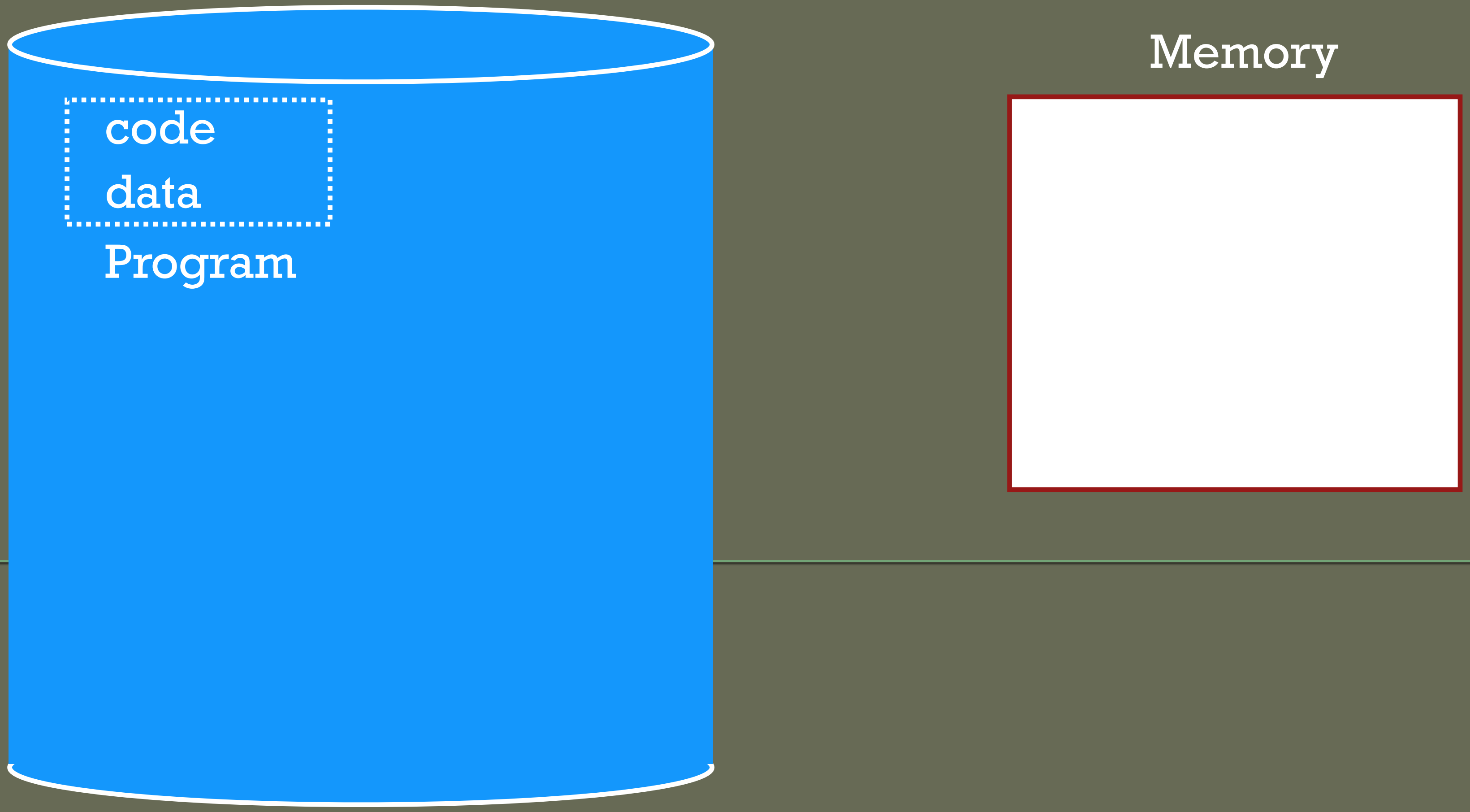
Possible Solutions for Mechanisms (covered today):

1. Time Sharing
2. Static Relocation
3. Base
4. Base+Bounds
5. Segmentation

1) Time Sharing of Memory

Try similar approach to how OS virtualizes CPU
Observation:
OS gives illusion of many virtual CPUs by saving **CPU registers** to **memory** when a process isn't running

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



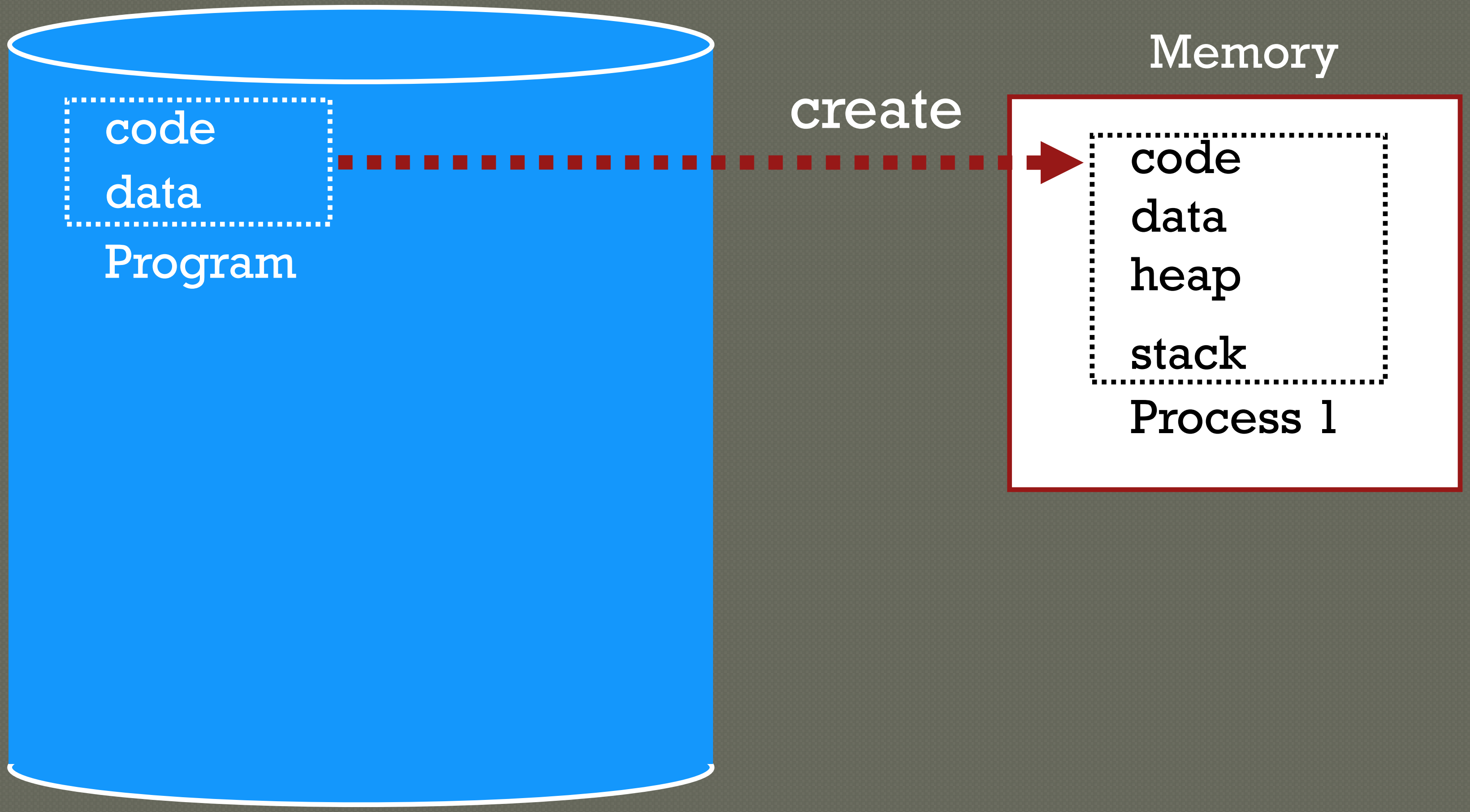
code

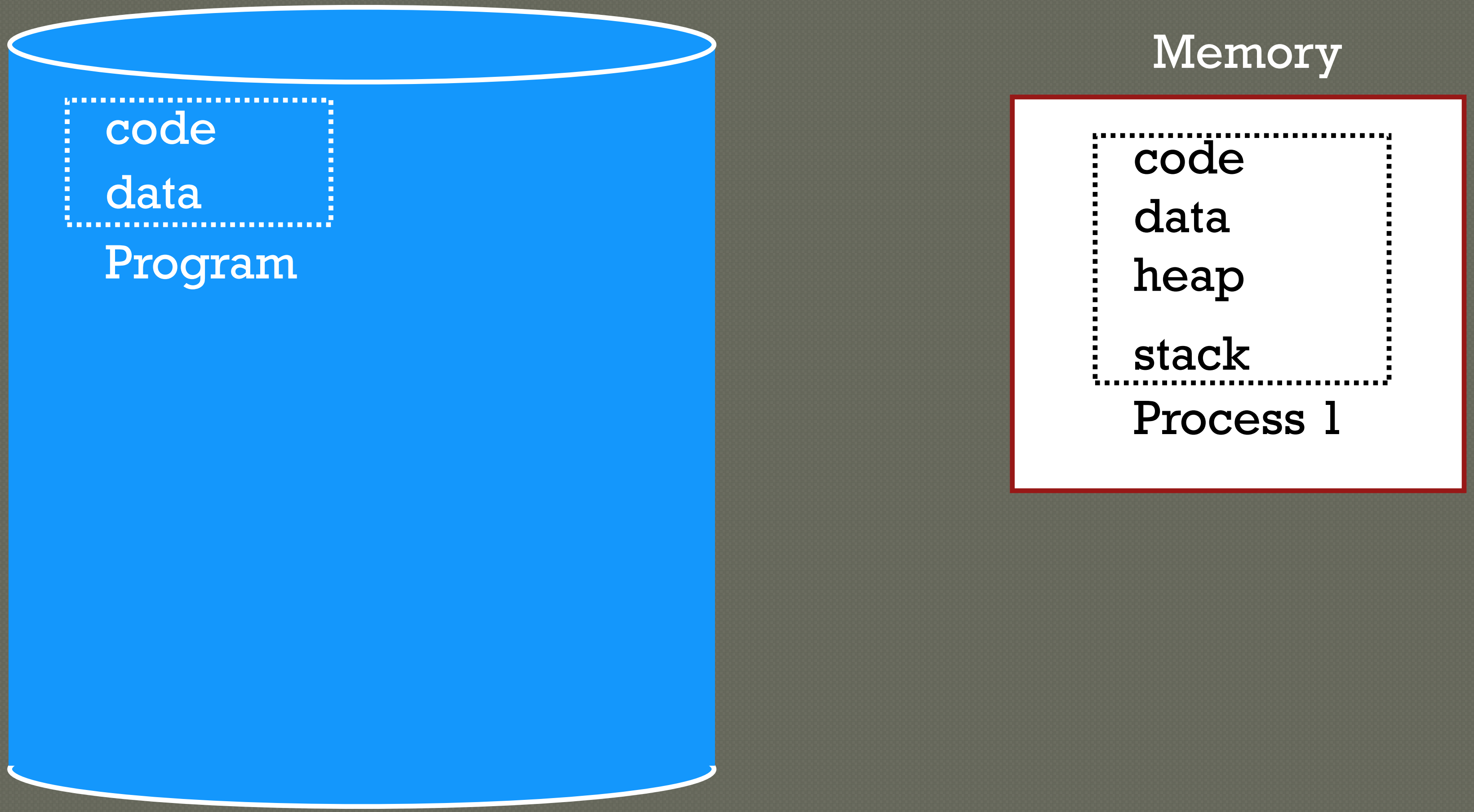
data

Program

Memory

Time Share Memory: Example

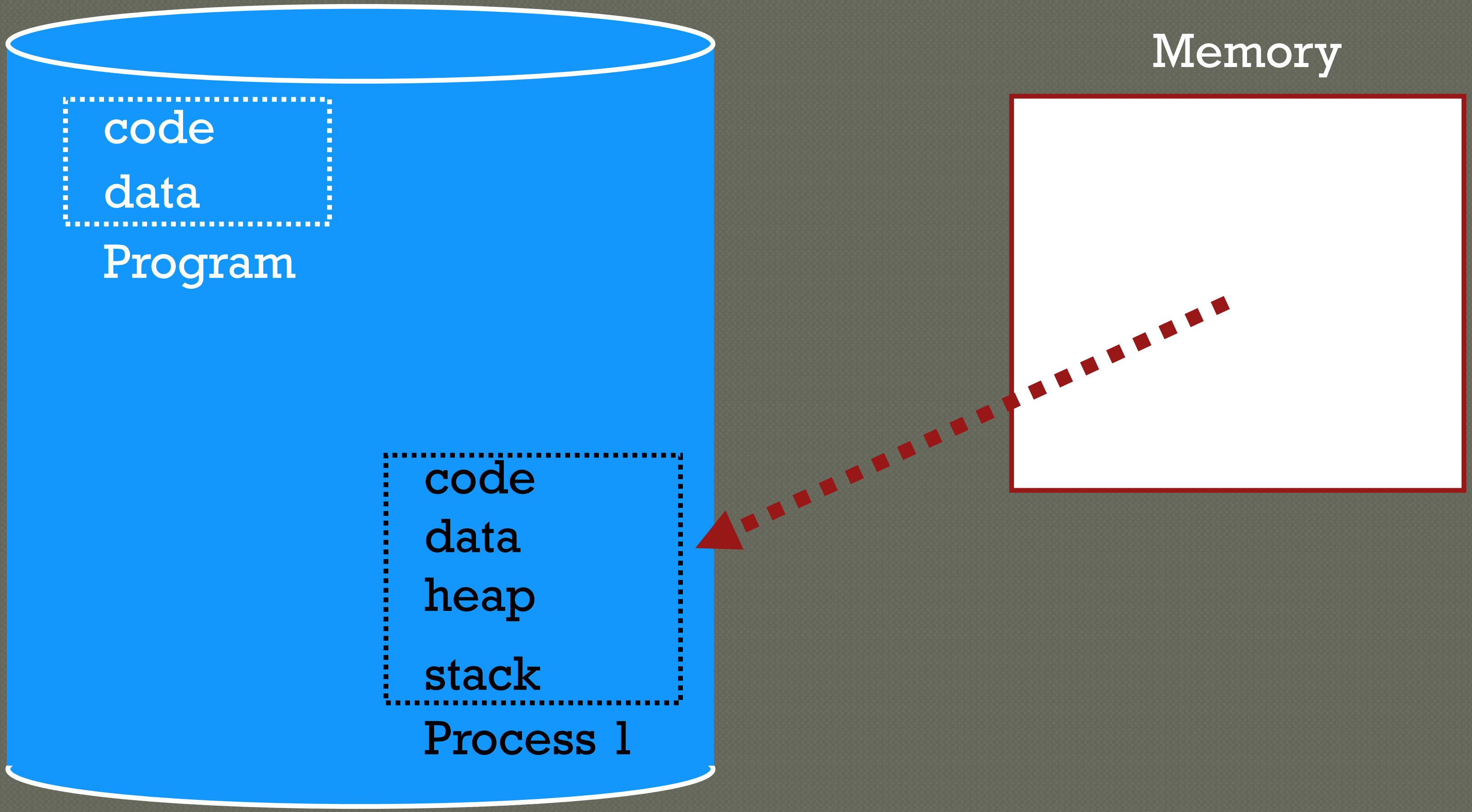

code
data
Program
create
Memory
code
data
heap
stack
Process 1

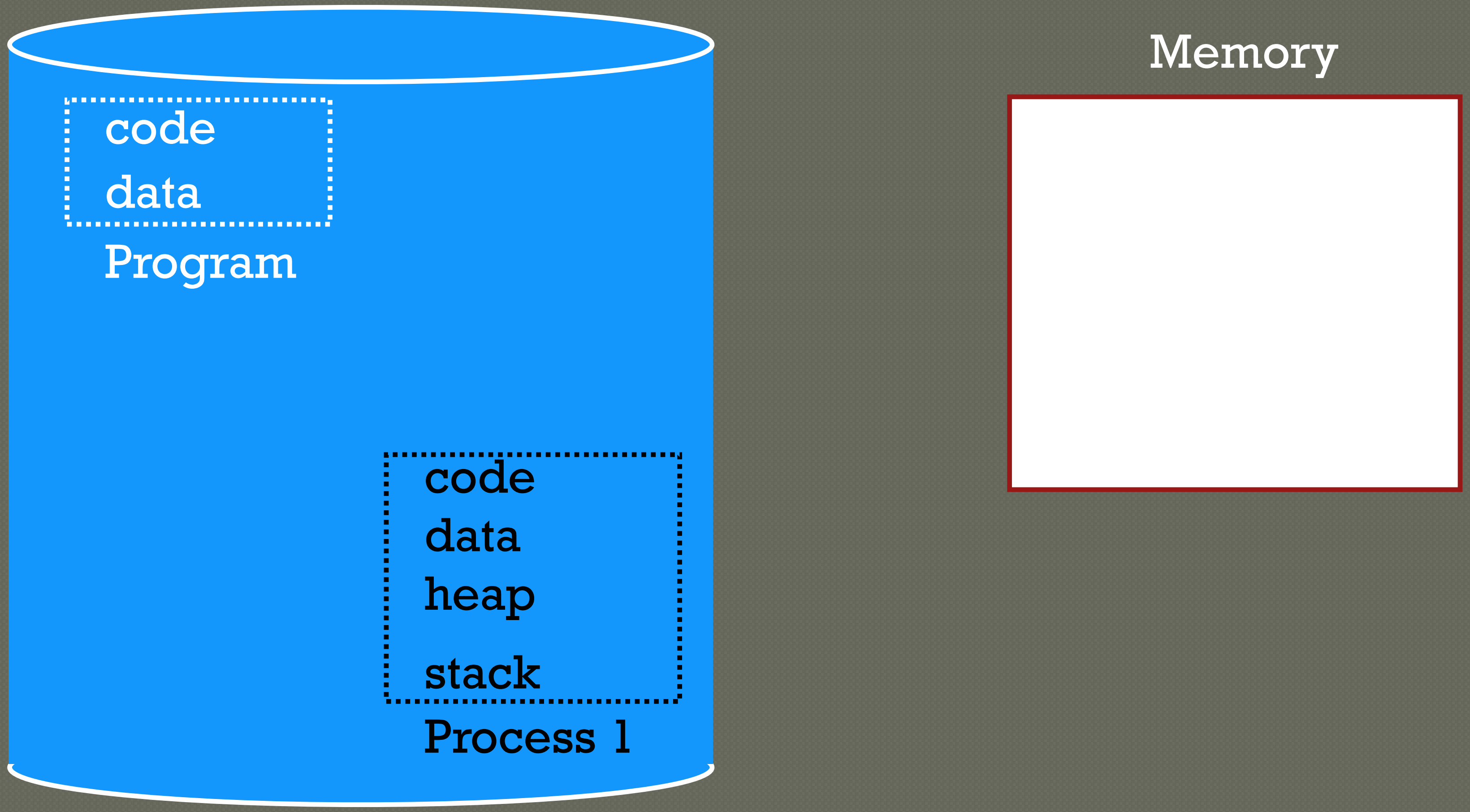


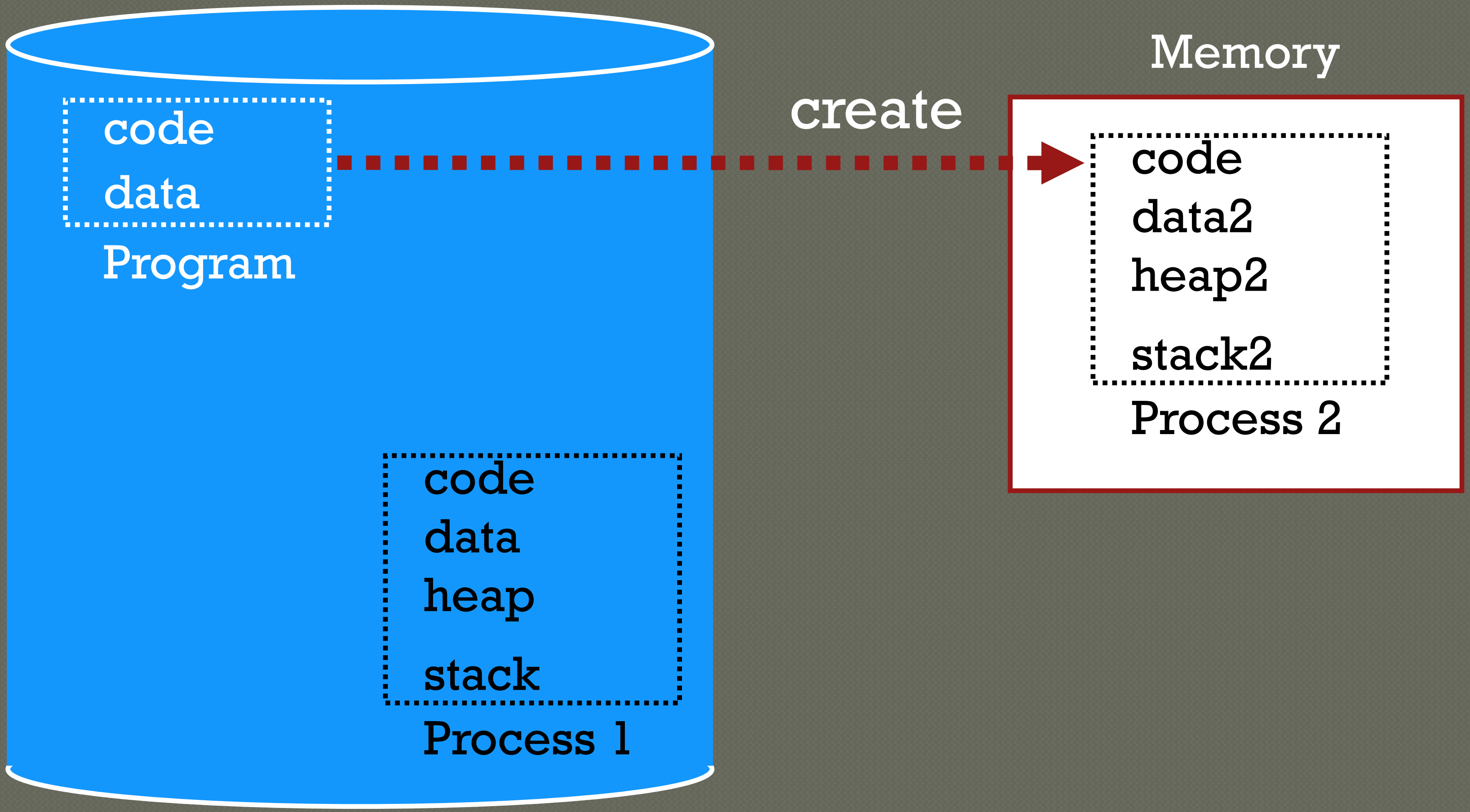
code
data
Program

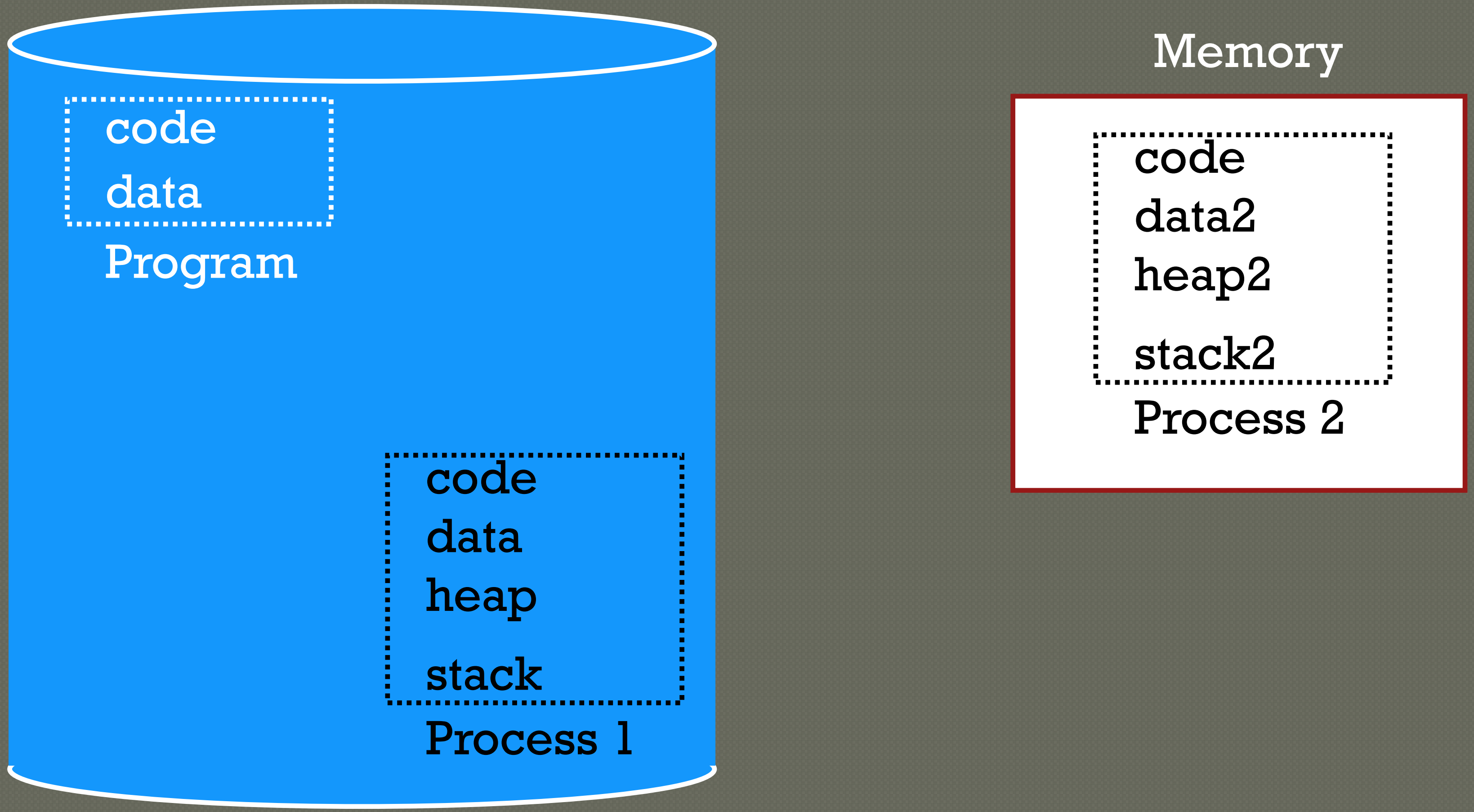
Memory

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
create
Memory
code
data2
heap2
stack2
Process 2

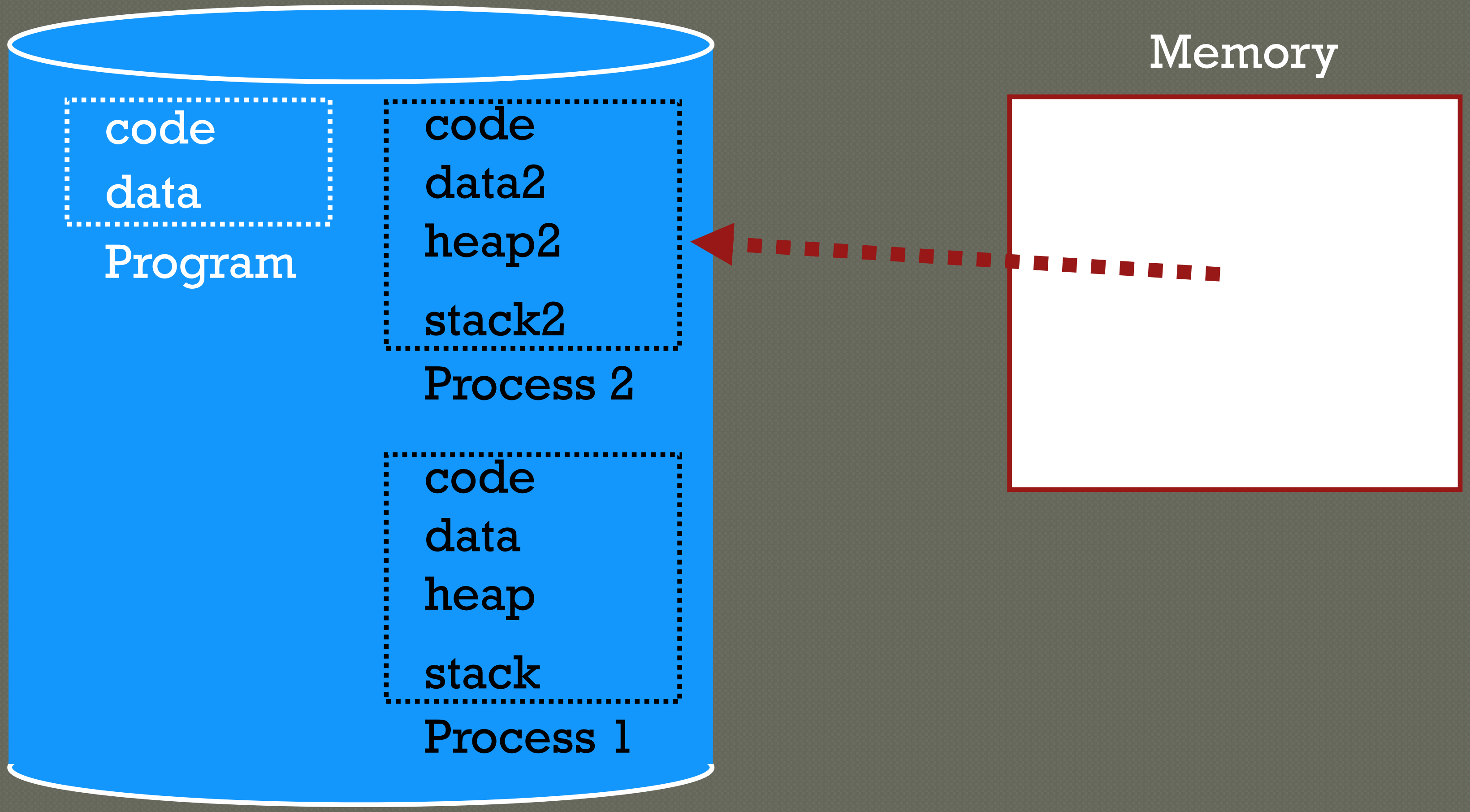


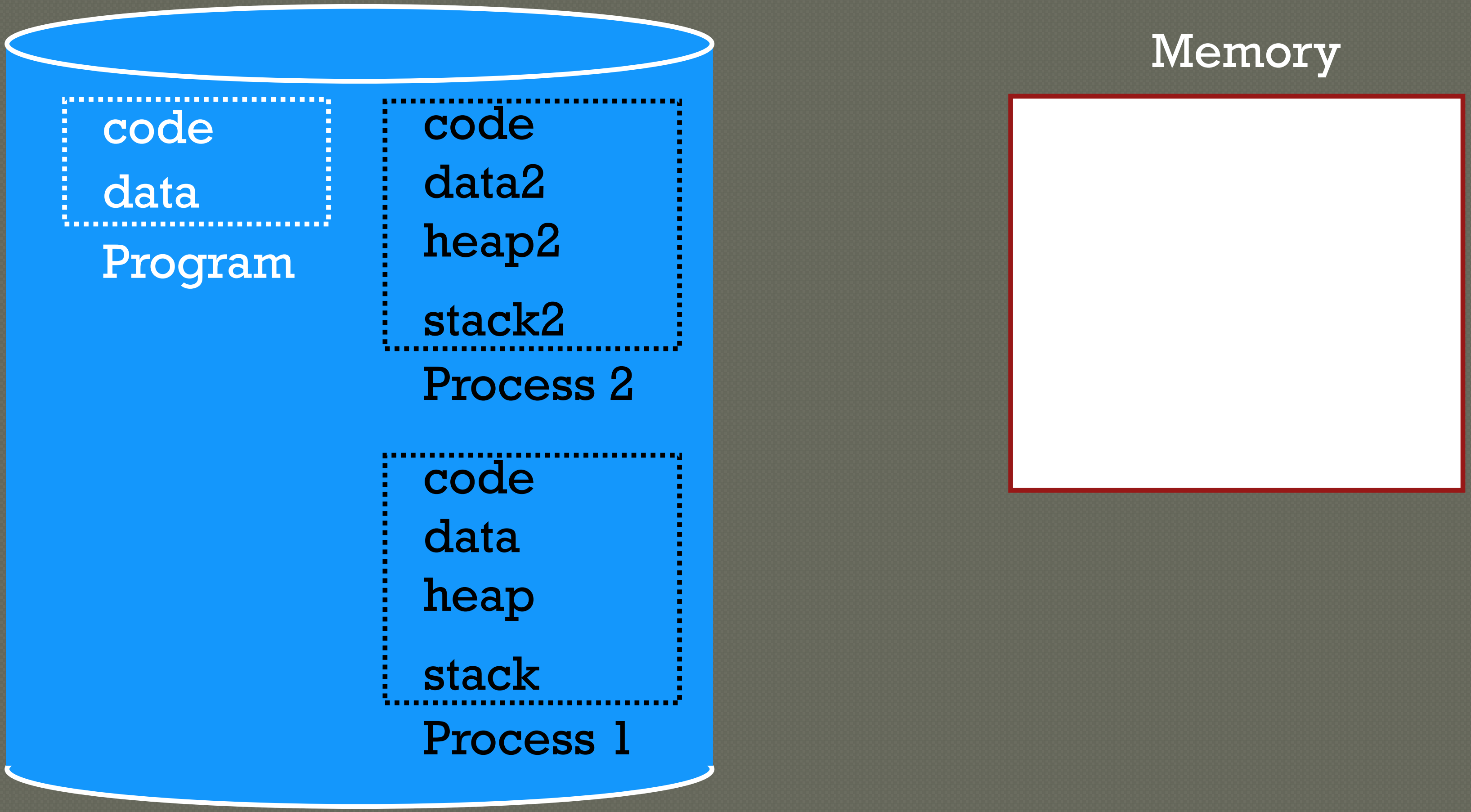
code
data
Program

code
data
heap
stack
Process 1

Memory

code
data2
heap2
stack2
Process 2


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

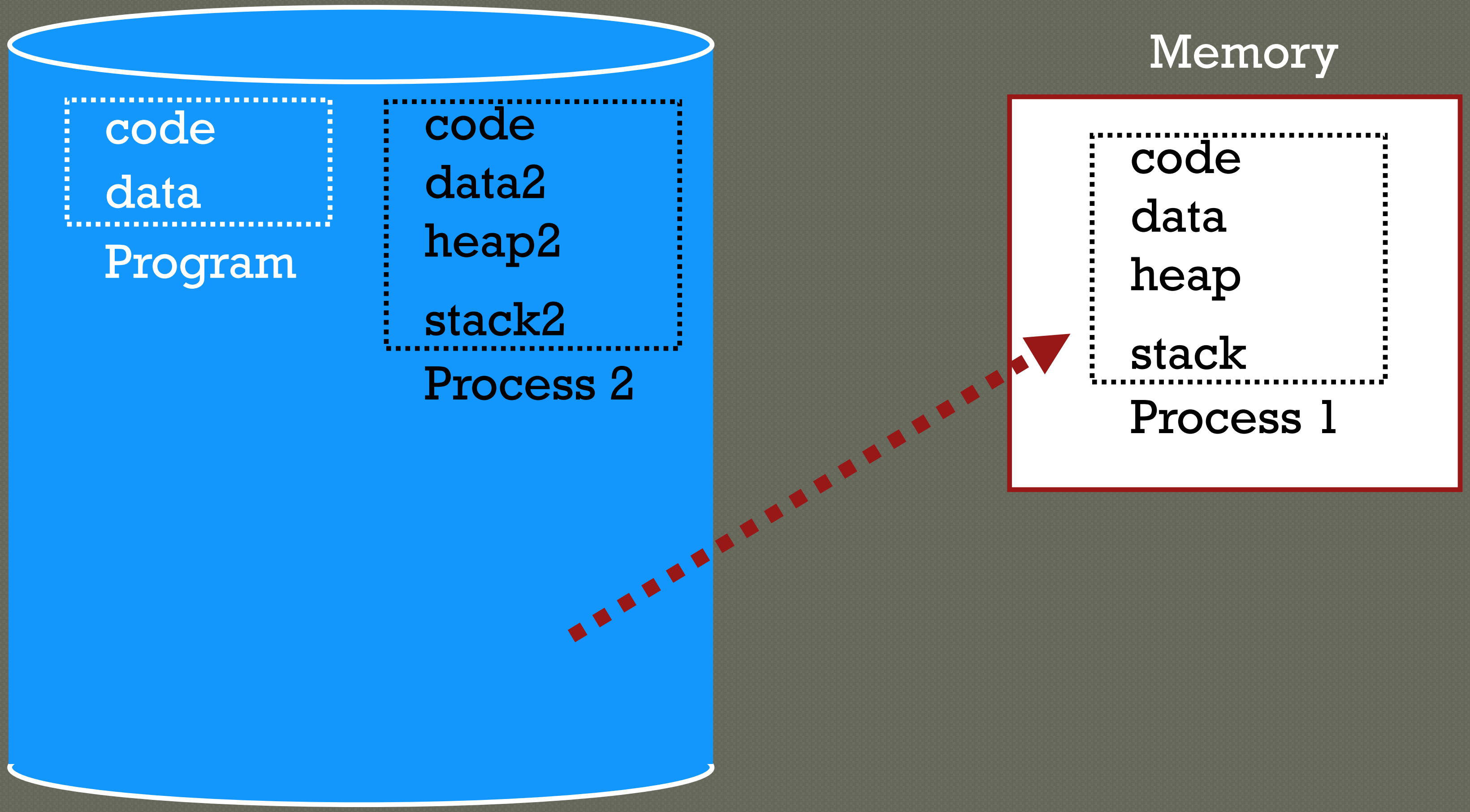


code
data
Program

code
data2
heap2
stack2
Process 2

code
data
heap
stack
Process 1

Memory

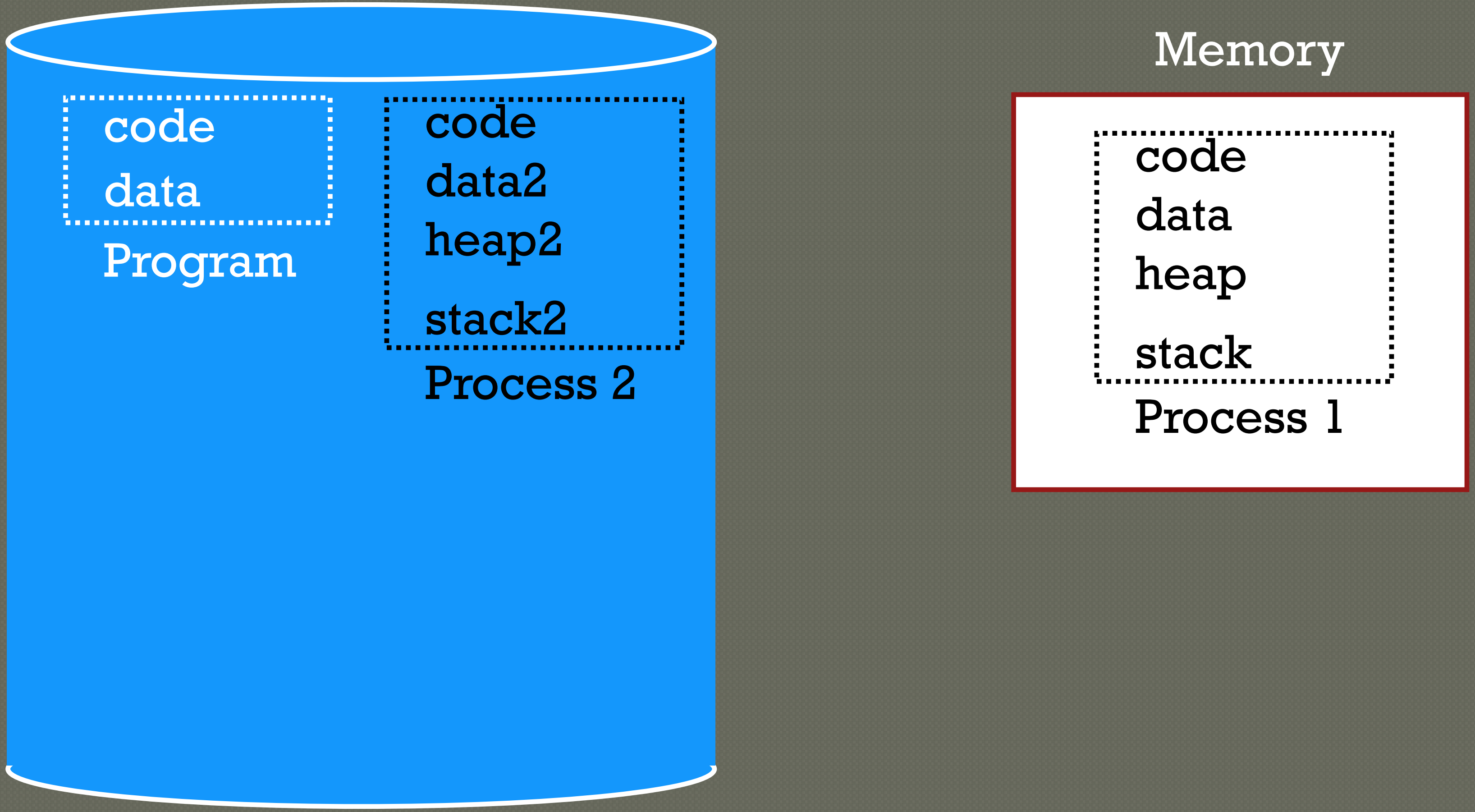


code
data
Program

code
data2
heap2
stack2
Process 2

Memory

code
data
heap
stack
Process 1



But disk is much slower than memory.
Or, it takes a long time to load process from disk

Problems with Time Sharing Memory

Problem: Ridiculously poor performance, so its not used

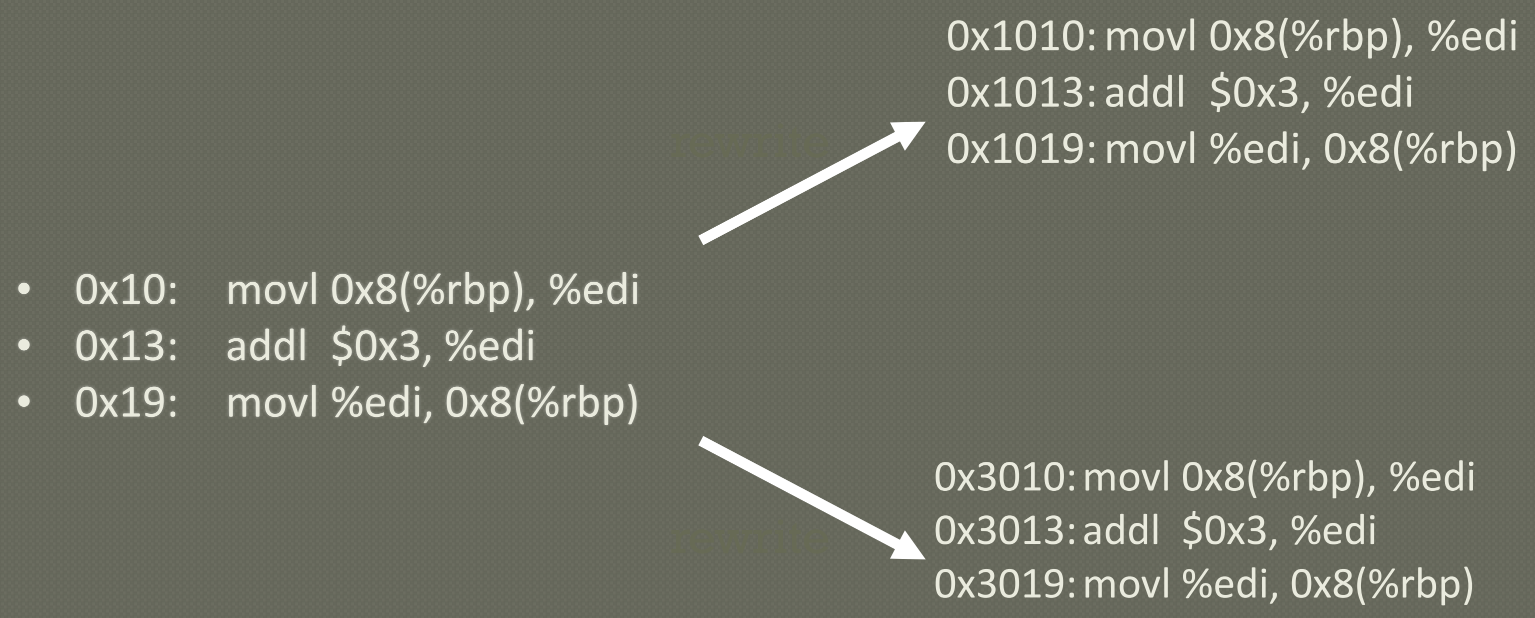
Better Alternative: space sharing

- At same time, space of memory is divided across processes

Remainder of solutions all use space sharing

2) Static Relocation

- Idea: OS rewrites each program before loading it as a process in memory
- Each rewrite for different process uses different addresses and pointers
- Change jumps, loads of static data

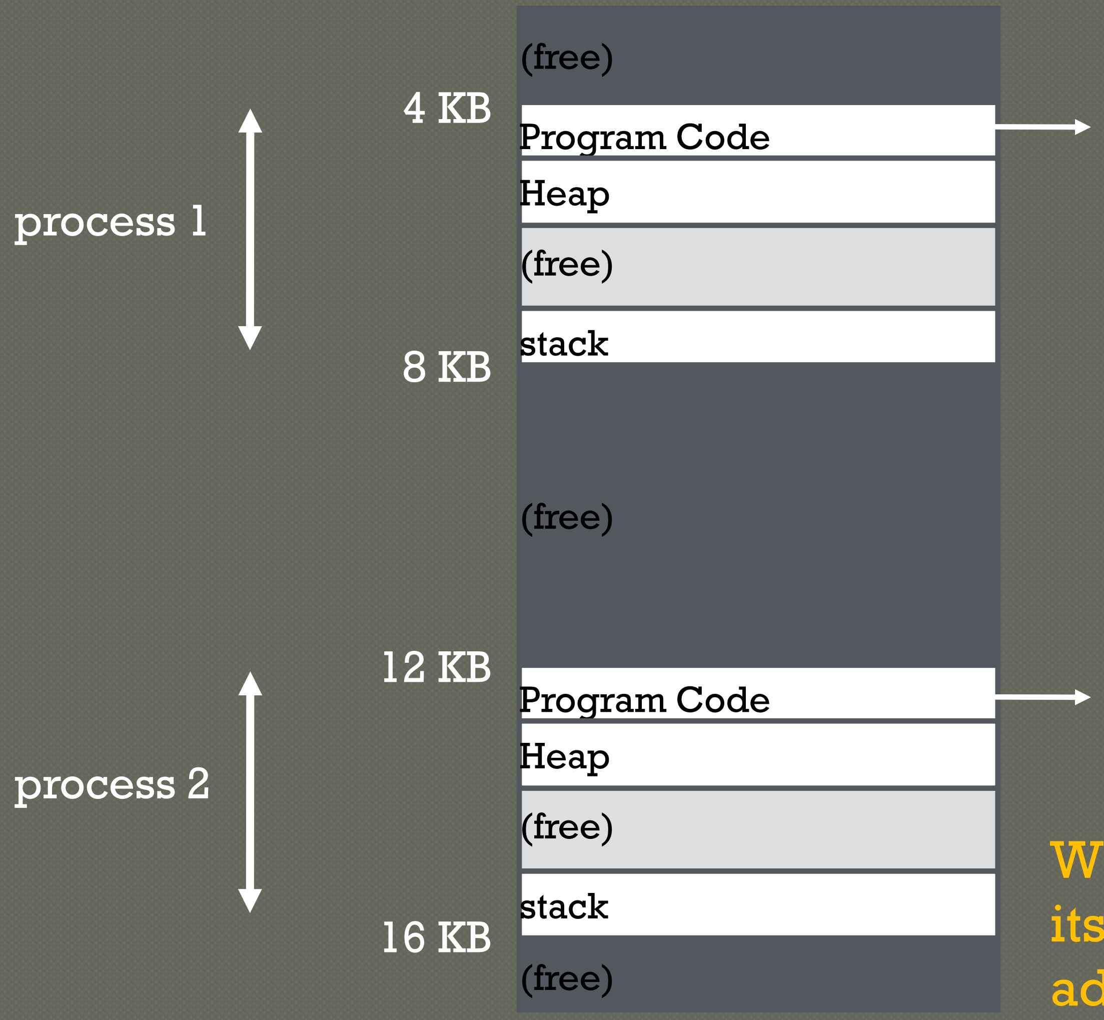


```
0x1010: movl 0x8(%rbp), %edi
0x1013: addl  $0x3, %edi
0x1019: movl %edi, 0x8(%rbp)
```

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

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

Static: Layout in Memory



```
 x1010: 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)
```

Why didn’t OS rewrite stack addr? Because its relative to rbp register, just change address in rbp and it works

Static Relocation: Disadvantages

No protection

- Process can destroy OS or other processes
- No privacy
- See “Aside: Software based relocation” in text

Hard to move address space after it has been placed

- May not be able to allocate new process

3) Dynamic Relocation

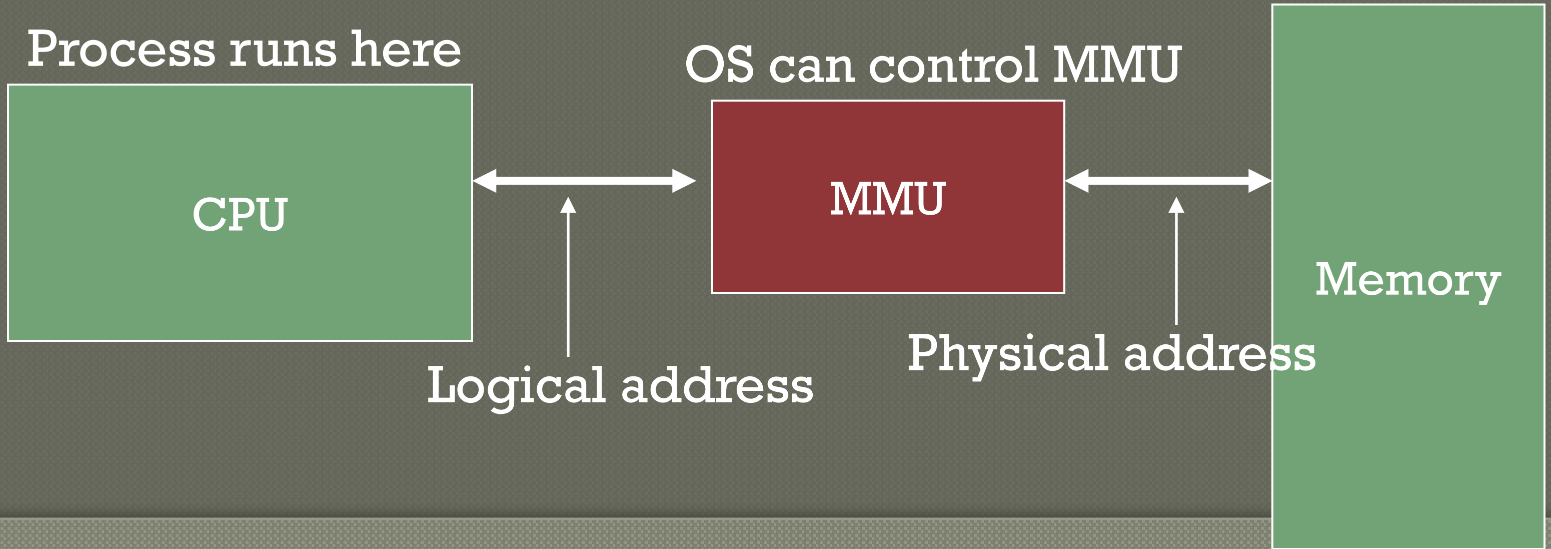
Goal: Protect processes from one another

Requires hardware support

- Memory Management Unit (MMU)

MMU dynamically changes process address at every memory reference

- Process generates logical or virtual addresses (in their address space)
- Memory hardware uses physical or real addresses



Hardware Support for Dynamic Relocation

Two operating modes

- Privileged (protected, kernel) mode: OS runs
    - When enter OS (trap, system calls, interrupts, exceptions)
    - Allows certain instructions to be executed
        - **Can manipulate contents of MMU**
    - **Allows OS to access all of physical memory**
- User mode: User processes run
    - **Perform translation of logical address to physical address**

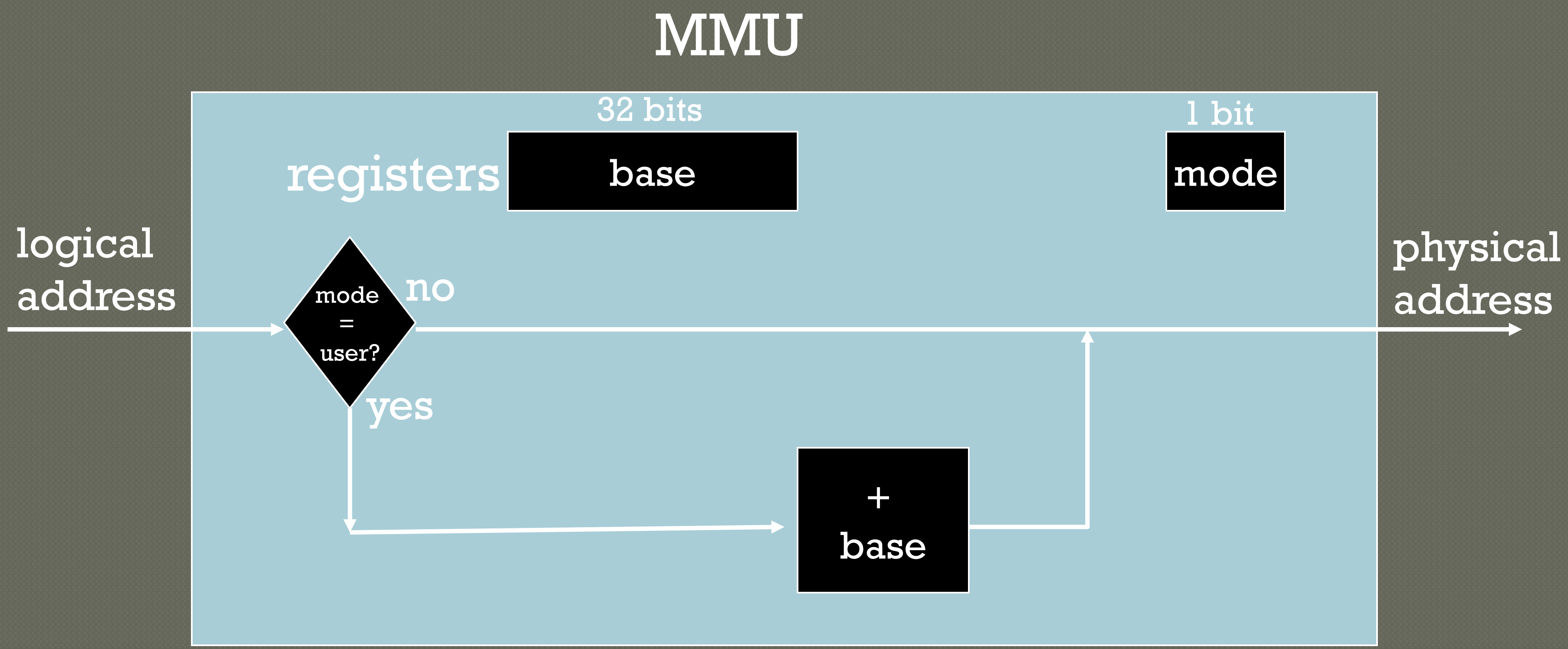
Minimal MMU contains **base register** for translation

- base: start location for address space

Implementation of Dynamic Relocation: BASE REG

Translation on every memory access of user process

- MMU adds base register to logical address to form physical address

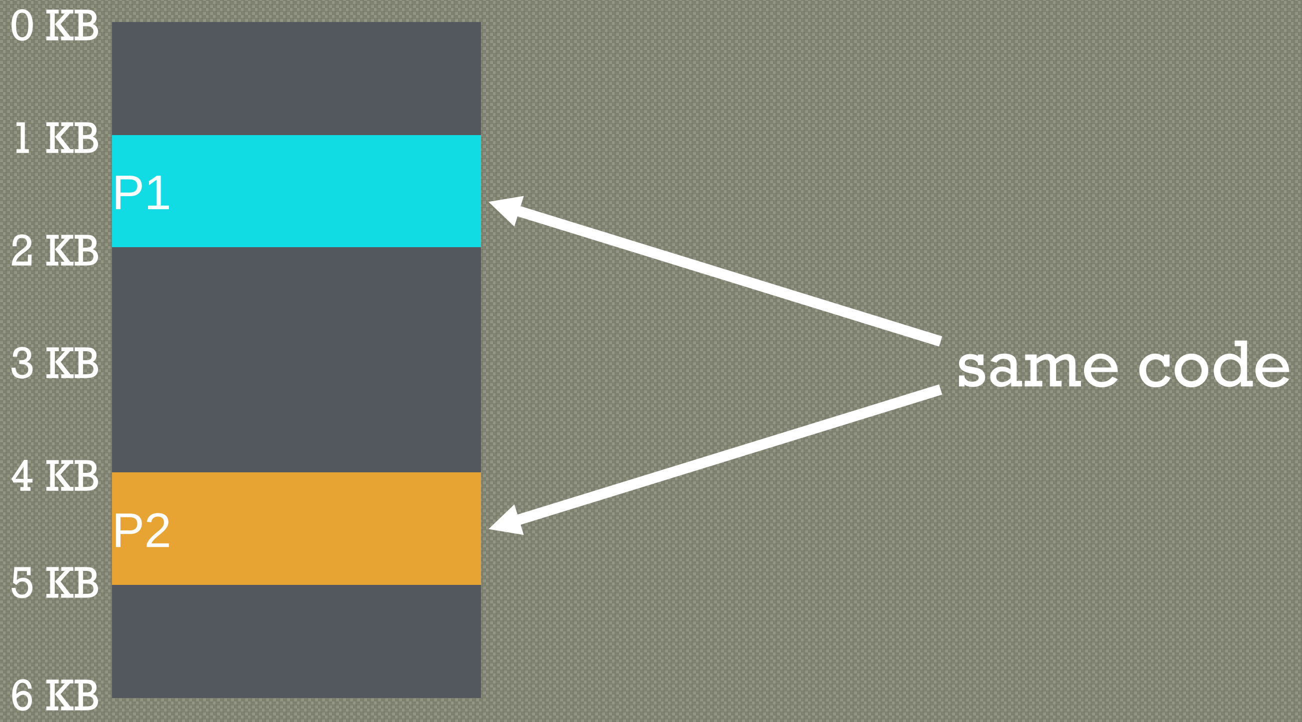


Dynamic Relocation with Base Register

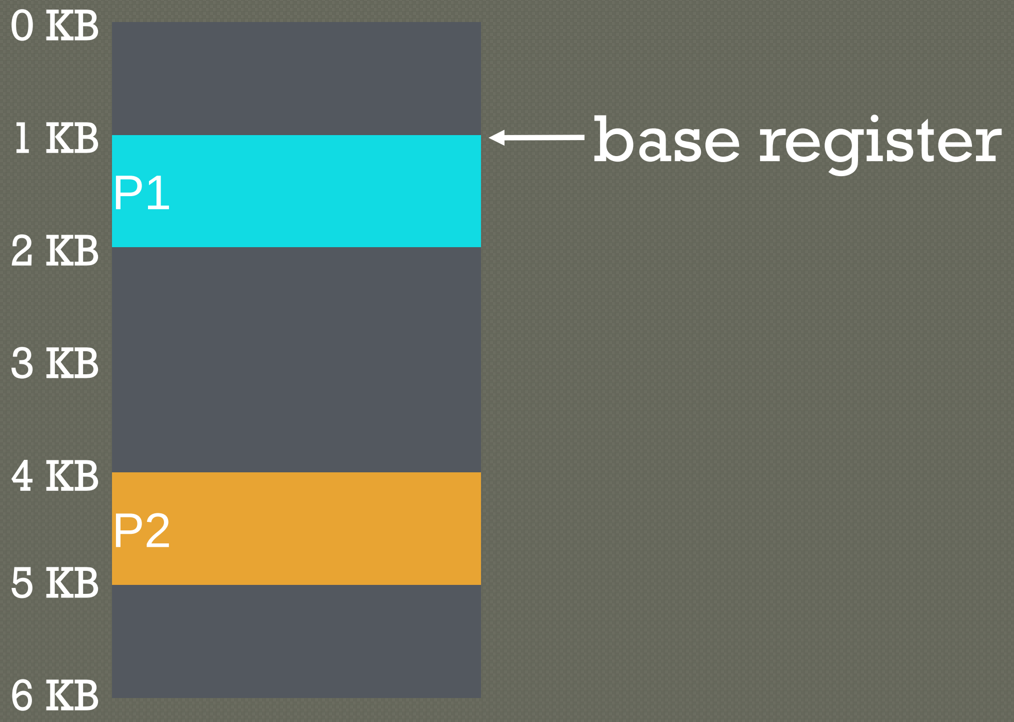
Idea: translate virtual addresses to physical by adding a fixed offset each time.

Store offset in base register

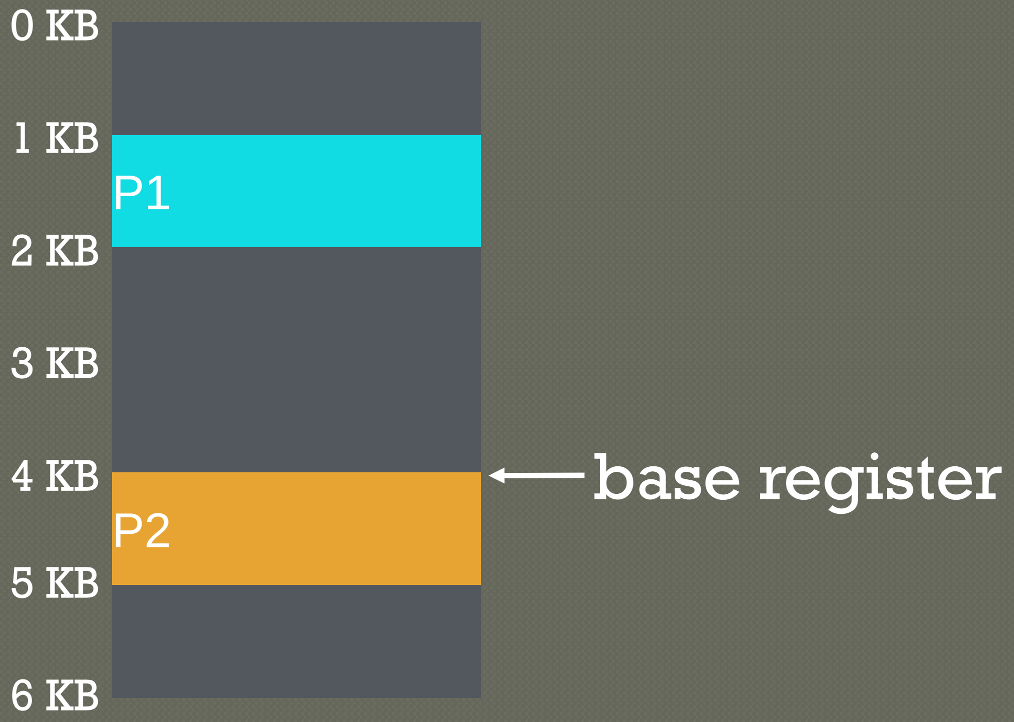
Each process has different value in base register



VISUAL Example of DYNAMIC RELOCATION:
BASE REGISTER

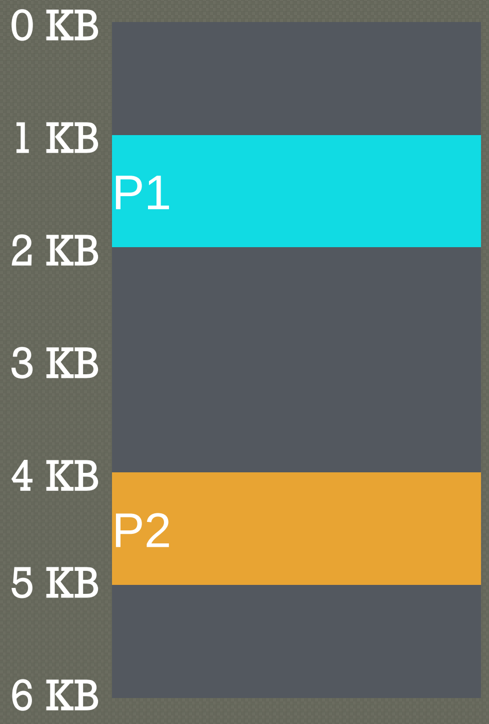


P1 is running

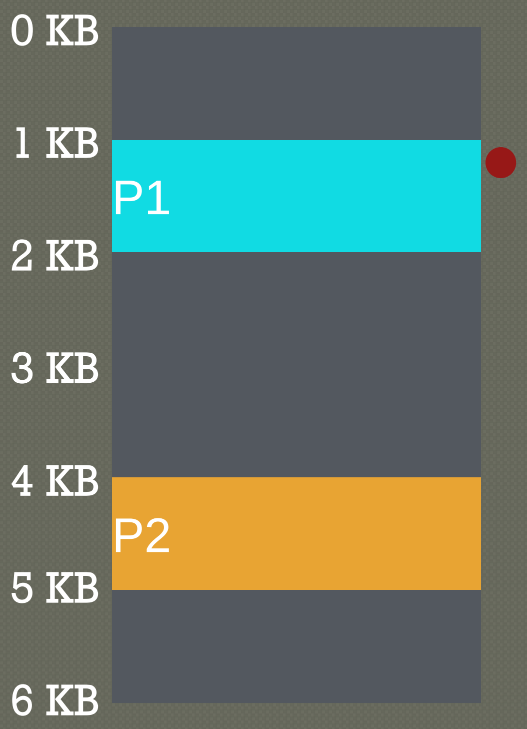


P2 is running

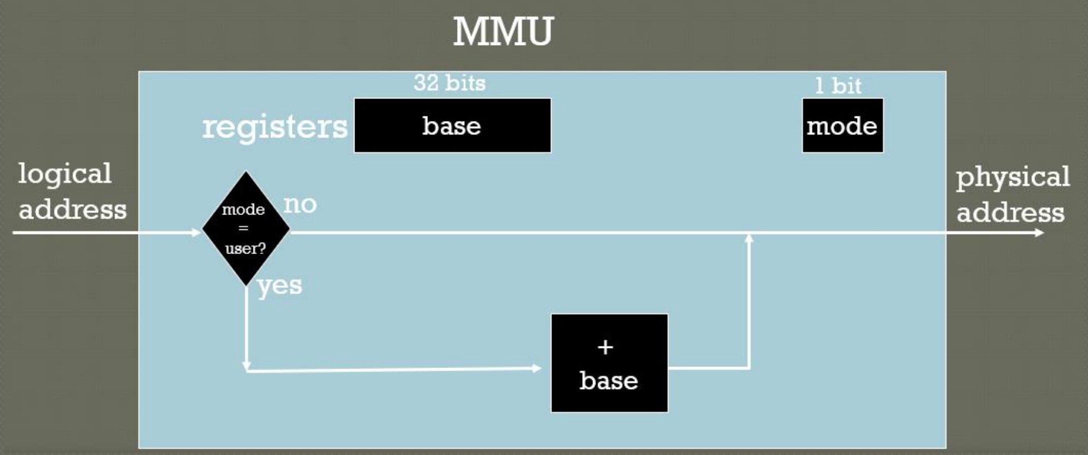
(Decimal notation)

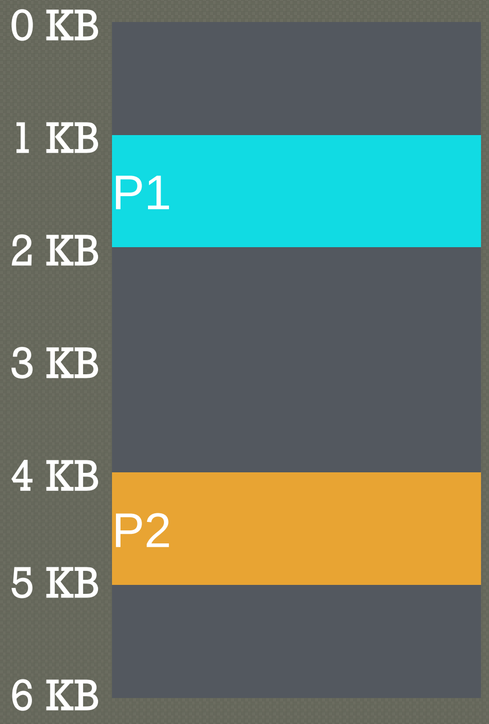


Virtual	Physical
P1: load 100, R1	

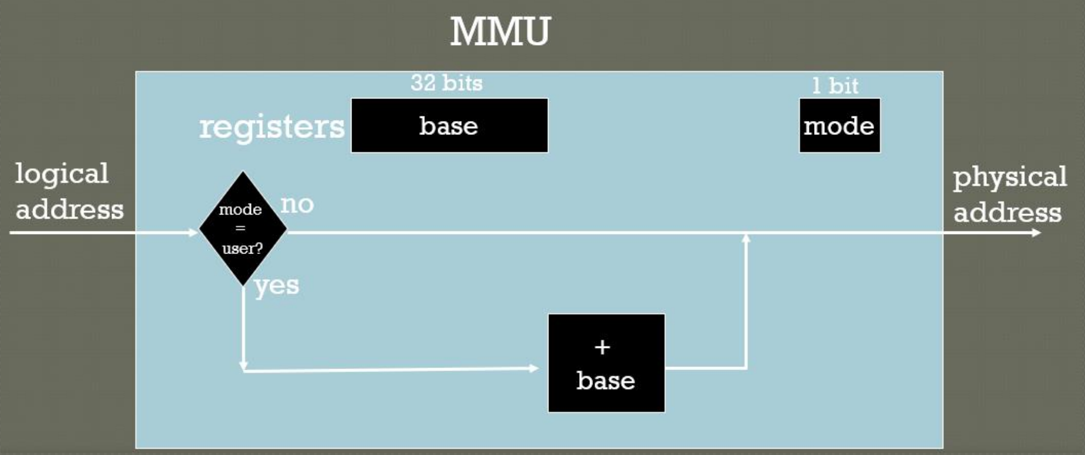


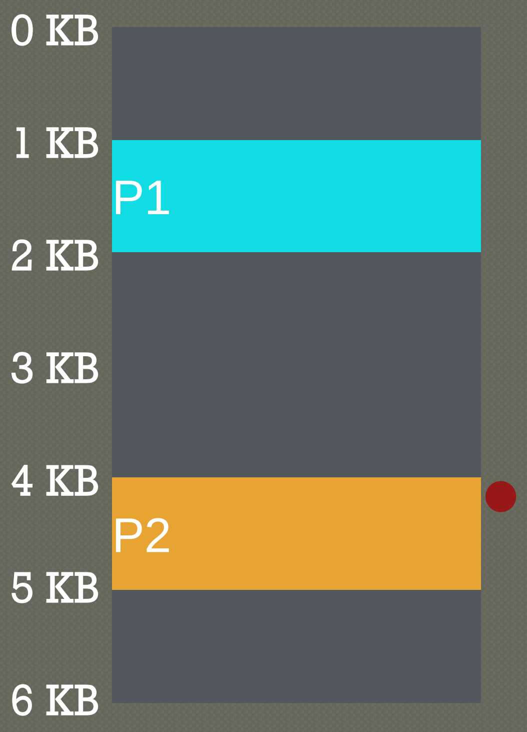
Virtual	Physical	
P1: load 100, R1	load 1124, R1	(1024 + 100)





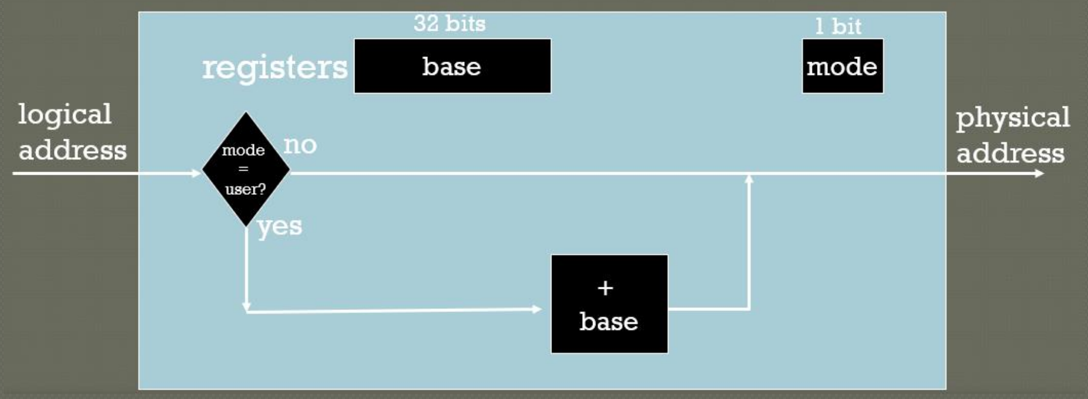
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	

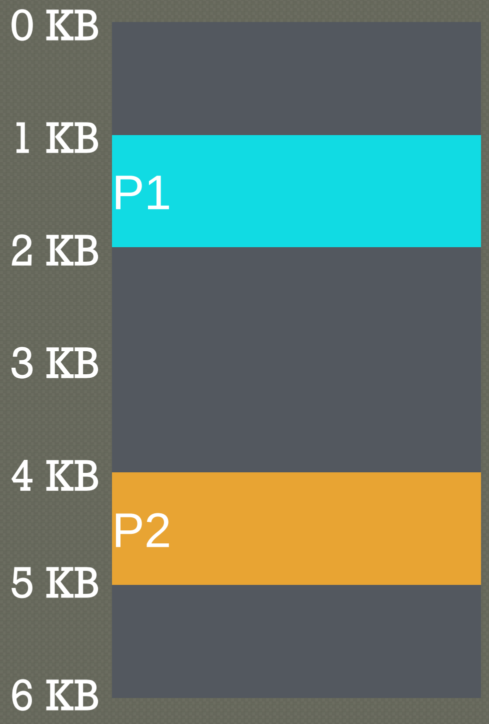




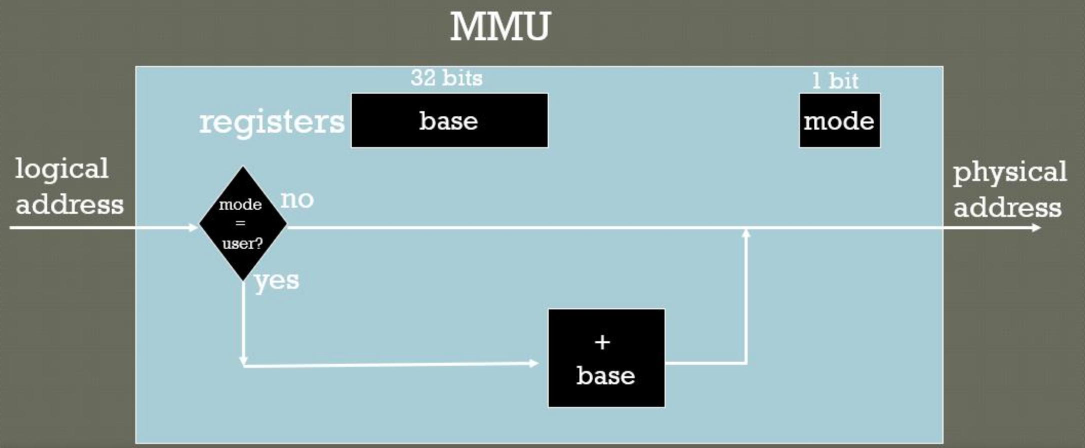
Virtual	Physical	
P1: load 100, R1	load 1124, R1	
P2: load 100, R1	load 4196, R1	(4096 + 100)

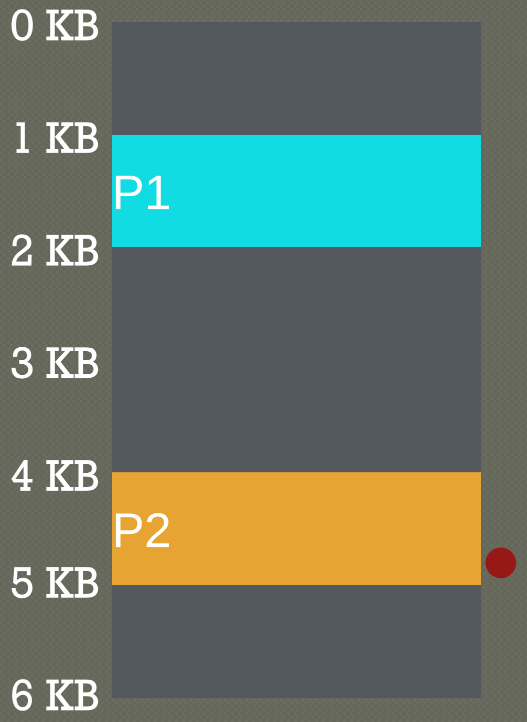
MMU





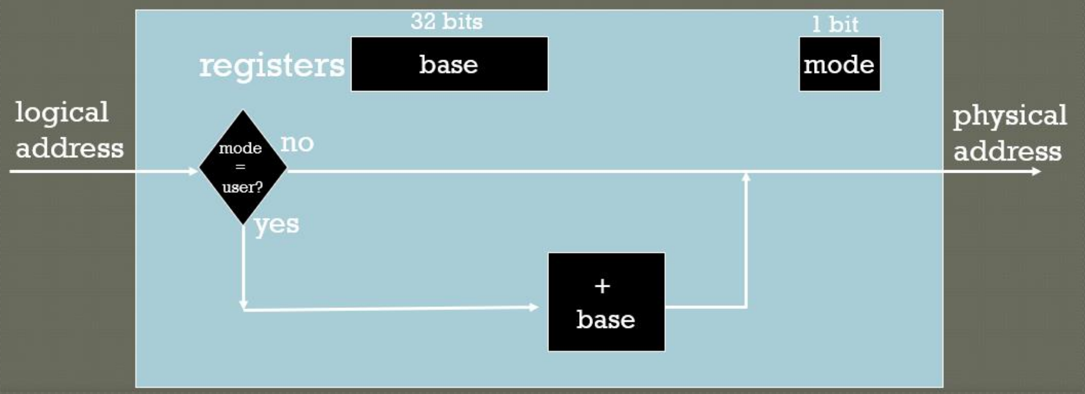
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	

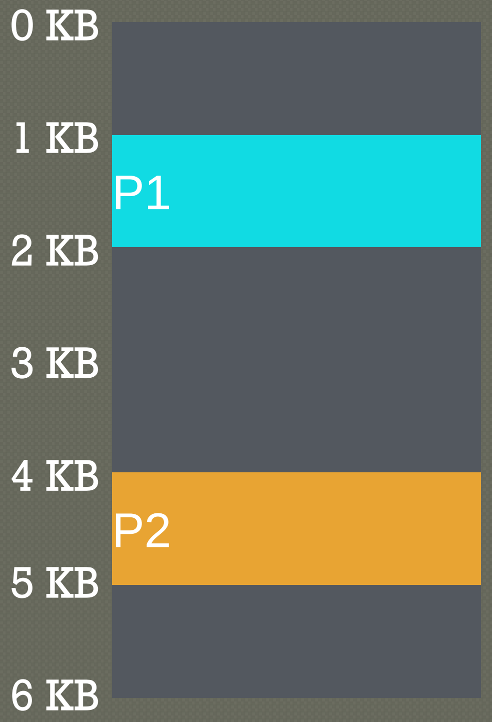




Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1

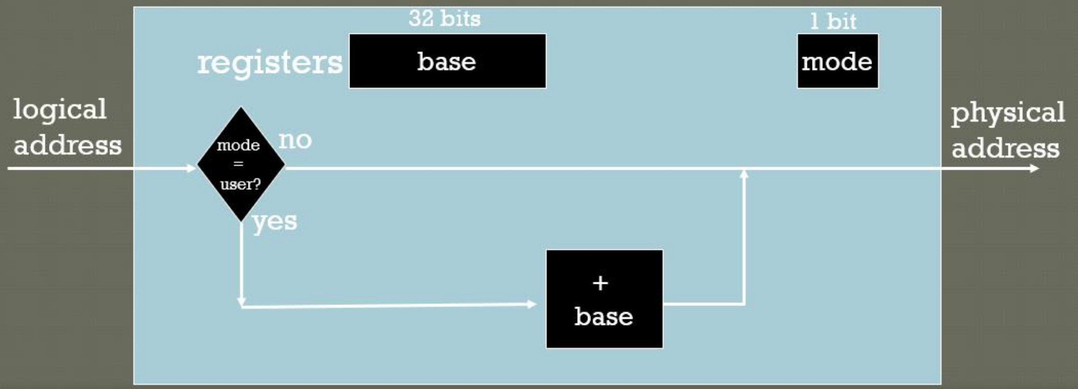
MMU

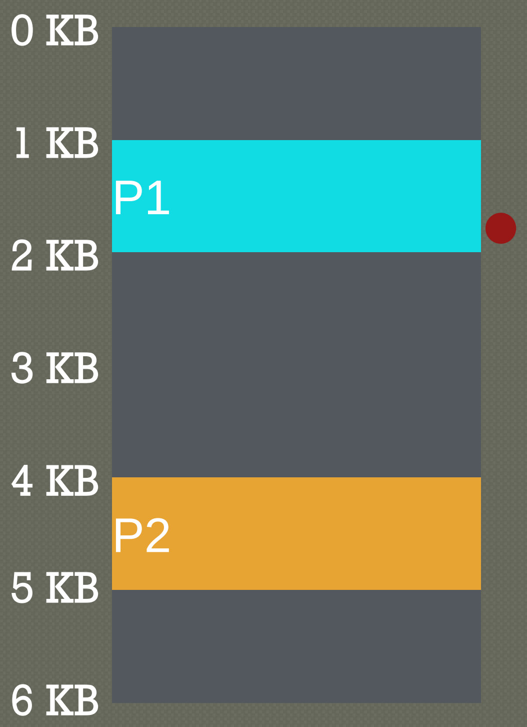




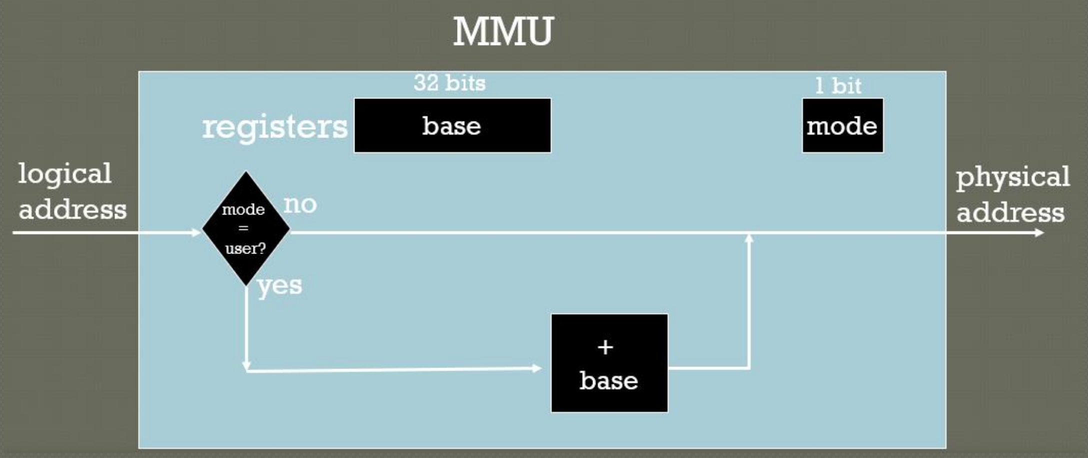
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	

MMU





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 1000, R1	load 2024, R1



Quiz: Who Controls the Base Register?

What entity should do translation of addresses with base register?

(1) process, (2) OS, or (3) HW

What entity should modify the base register?

(1) process, (2) OS, or (3) HW

Quiz: Who Controls the Base Register?

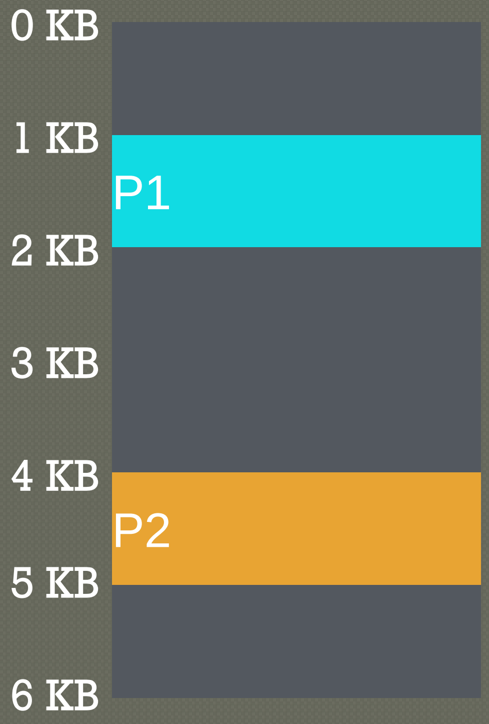
What entity should do translation of addresses with base register?

(1) process, (2) OS, or (3) [✓ HW] Speed!

What entity should modify the base register?

(1) process, (2) [✓ OS], or (3) HW

Changes when new process loaded

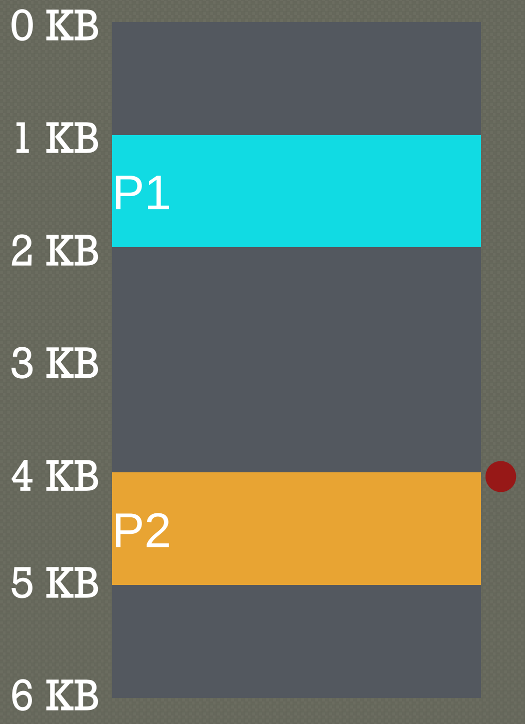


Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1

Can P2 hurt P1?

Can P1 hurt P2?

How well does dynamic relocation do with base register for protection?



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	(3072 + 1024)

Can P2 hurt P1?

Can P1 hurt P2?

How well does dynamic relocation do with base register for protection?

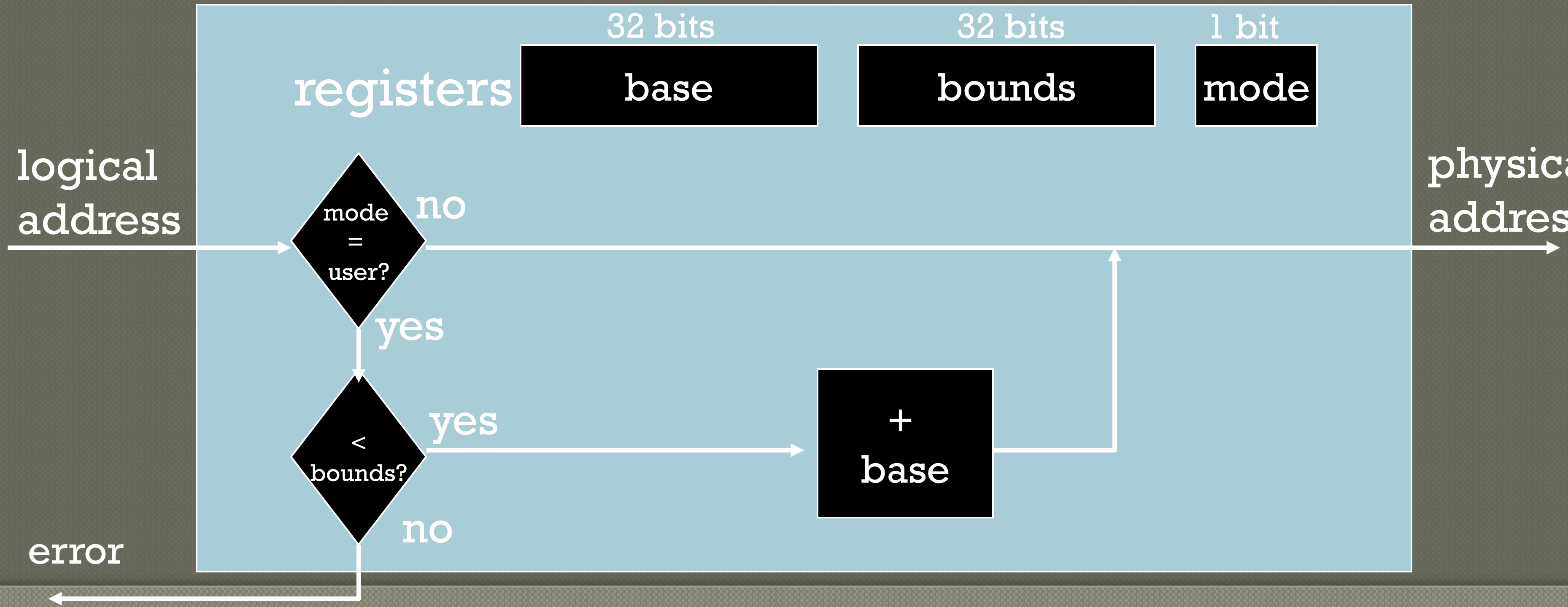
4) Dynamic with Base+Bounds

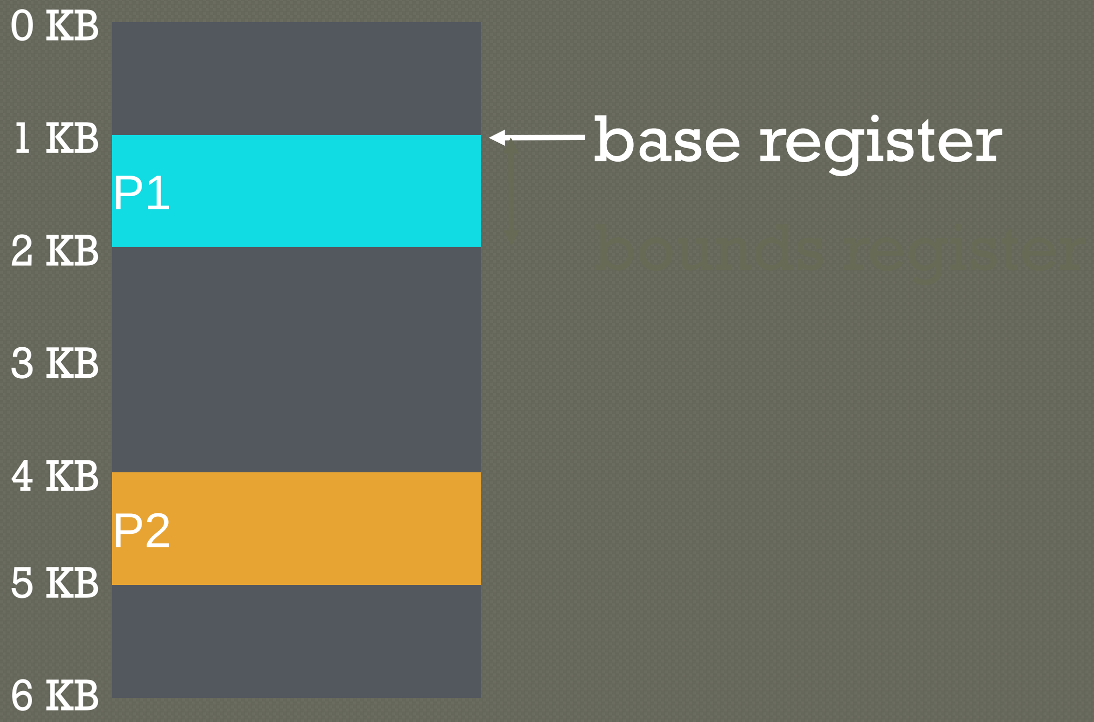
- Idea: limit the address space with a bounds register
- Base register: smallest physical addr (or starting location)
- Bounds register: size of this process’s virtual address space
  - Sometimes defined as largest physical address (base + size)
- OS kills process if process loads/stores beyond bounds

Implementation of BASE+BOUNDS

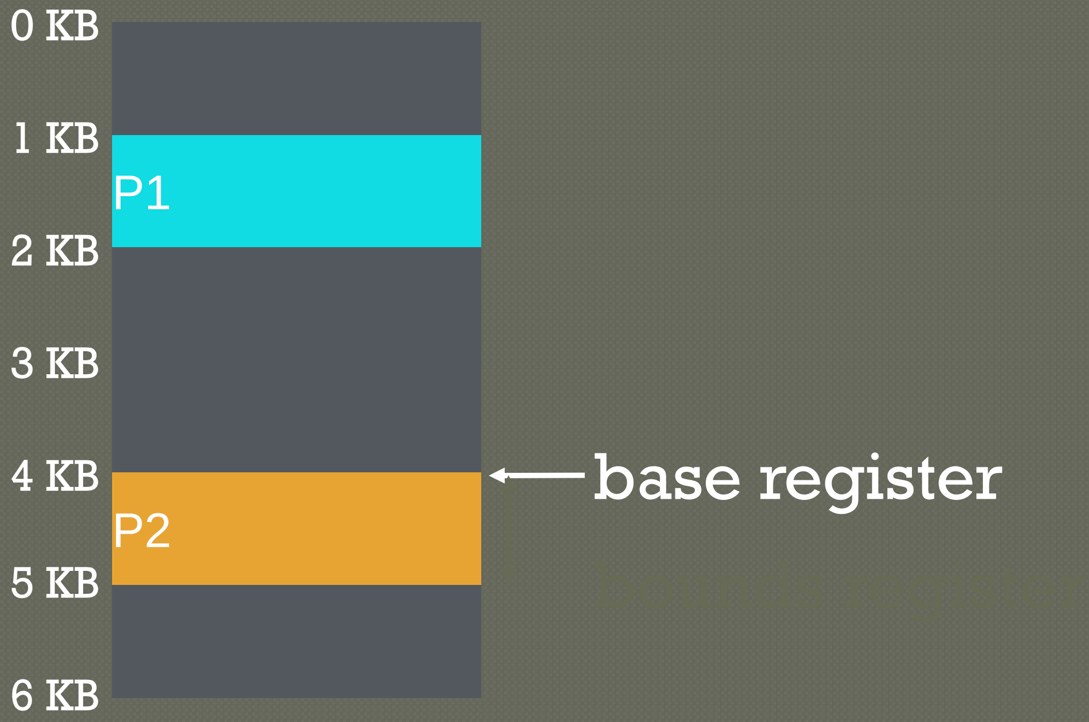
Translation on every memory access of user process

- MMU compares logical address to bounds register
  - if logical address is greater, then generate error
- MMU adds base register to logical address to form physical address

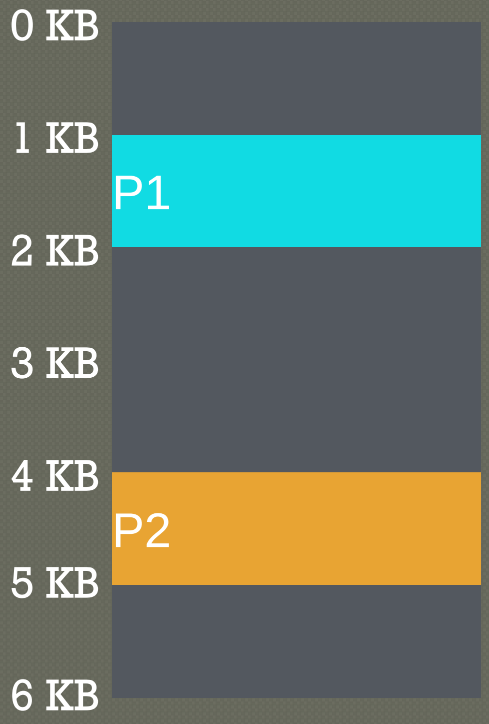




P1 is running

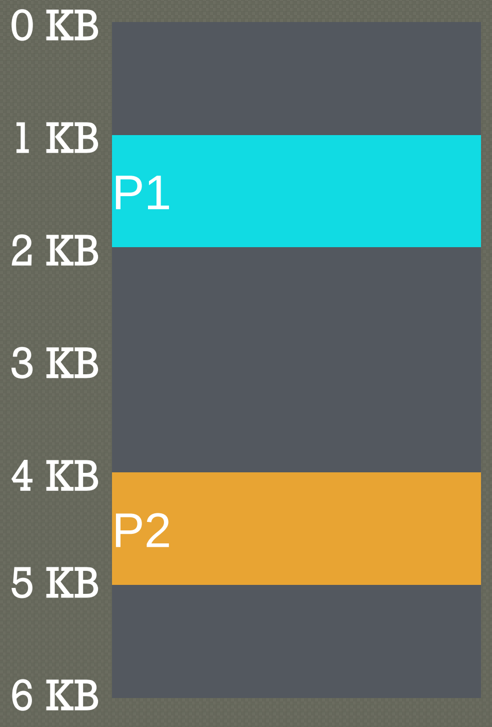


P2 is running



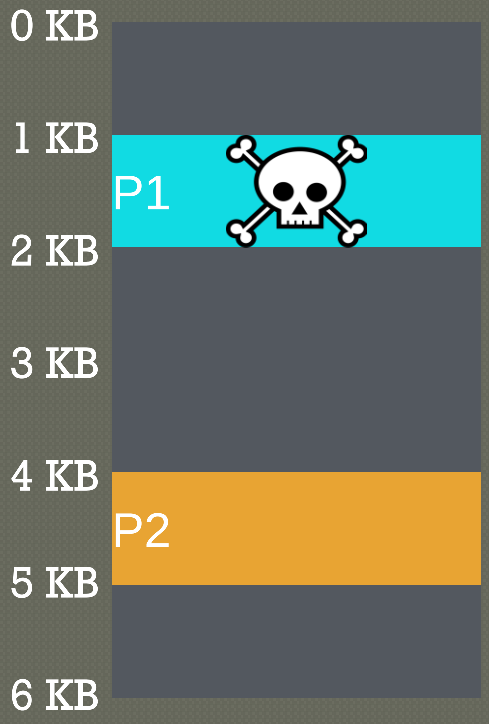
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	

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!	3072 > 1024

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?

Managing Processes with Base and Bounds

Context-switch

- Add base and bounds registers to PCB
- Steps
  - Change to privileged mode
  - Save base and bounds registers of old process
  - Load base and bounds registers of new process
  - Change to user mode and jump to new process

What if don’t change base and bounds registers when switch?

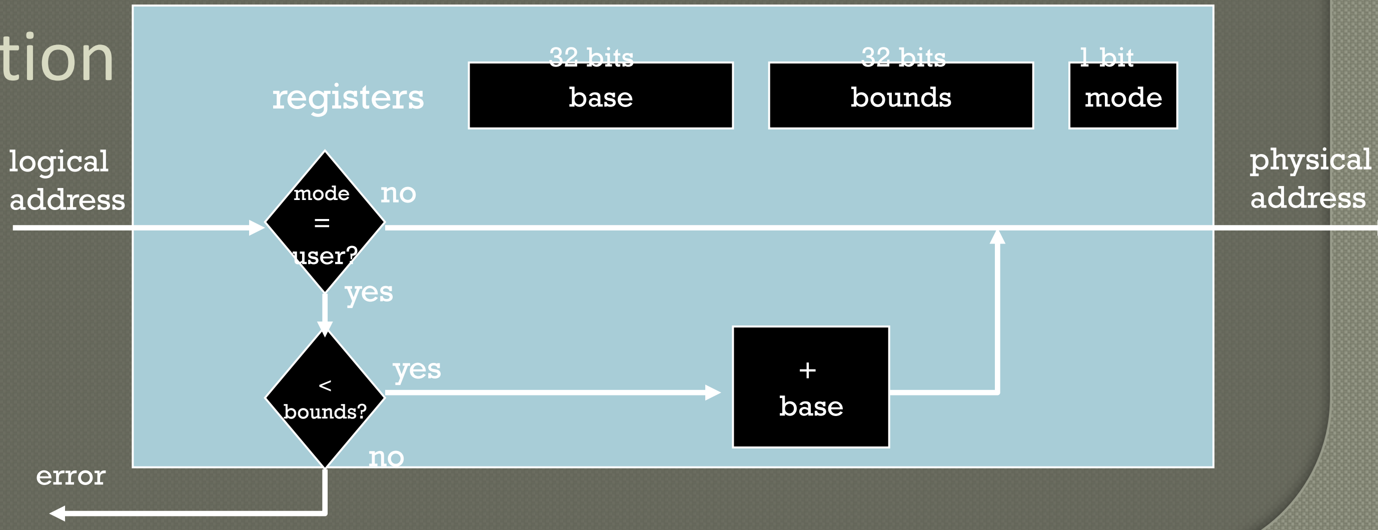
Protection requirement

- User process cannot change base and bounds registers
- User process cannot change to privileged mode

Base and Bounds Advantages

Advantages

- Provides protection (both read and write) across address spaces
- Supports dynamic relocation
  - Can place process at different locations initially and also move address spaces
- Simple, inexpensive implementation
  - Few registers, little logic in MMU
- Fast
  - Add and compare in parallel

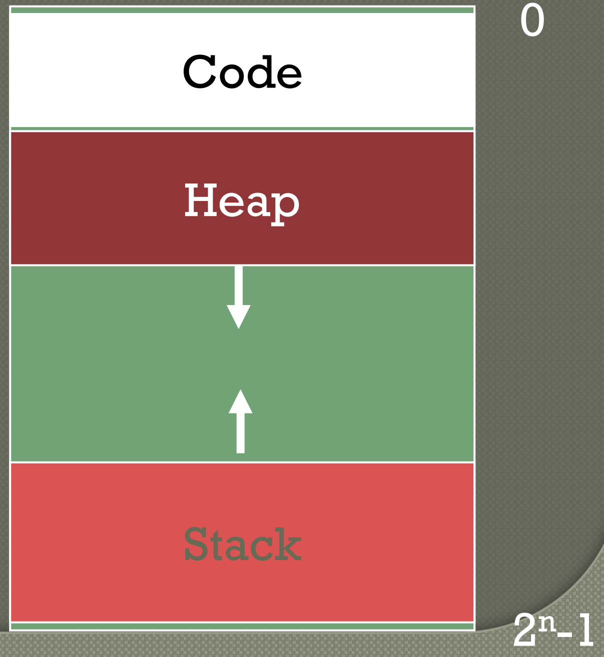


Base and Bounds DISADVANTAGES

Disadvantages

- Each process must be allocated contiguously in physical memory
  - Must allocate memory that may not be used by process

- No partial sharing: Cannot share limited parts of address space



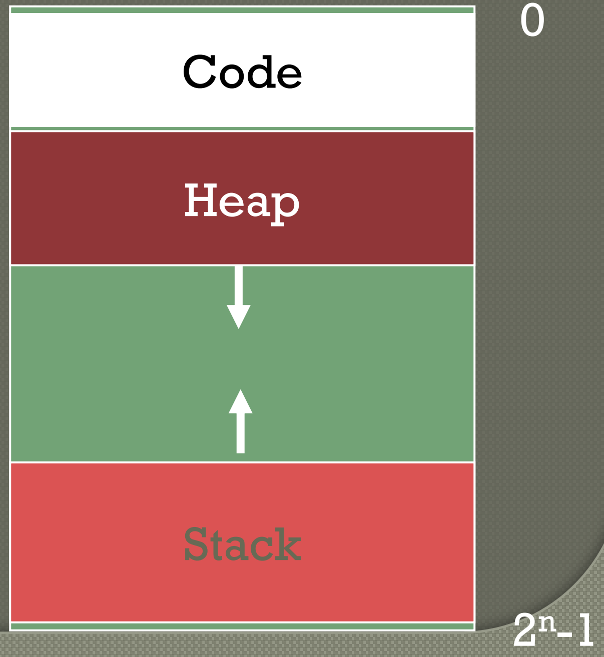
5) Segmentation

Divide address space into logical segments

- Each segment corresponds to logical entity in address space
  - code, stack, heap

Each segment can independently:

- be placed separately in physical memory
- grow and shrink
- be protected (separate read/write/execute
- protection bits)



Segmented Addressing

Process now specifies segment and offset within segment

How does process designate a particular segment?

- Use part of logical address
  - Top bits of logical address select segment
  - Low bits of logical address select offset within segment

Segmentation Implementation

MMU contains Segment Table (per process)

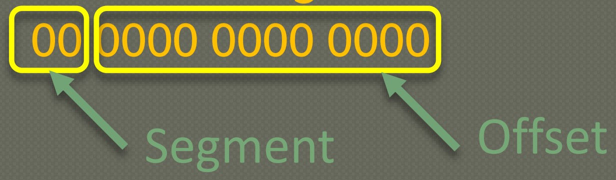
- Each segment has own base and bounds, protection bits
- Example: 14 bit logical address, 4 segments; how many bits for segment? How many bits for offset?

Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x000	0 0

remember:
1 hex digit->4 bits

Have 3 unique segments- need 2 bits to uniquely identify (00,01,10 with 11 unused)

12 remaining bits are offset (can address up to 2**12 = 4096 memory locations



00 0000 0000 0000

Segment

Offset

Quiz: Address Translations with Segmentation

MMU contains Segment Table (per process)

- Each segment has own base and bounds, protection bits
- Example: 14 bit logical address, 4 segments; how many bits for segment? How many bits for offset?

Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x000	0 0

remember:
1 hex digit->4 bits

Translate logical addresses (in hex) to physical addresses

0x0240:

0x1108:

0x265c:

Quiz: Address Translations with Segmentation

MMU contains Segment Table (per process)

- Each segment has own base and bounds, protection bits
- Example: 14 bit logical address, 4 segments; how many bits for segment? How many bits for offset?

Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x000	0 0

remember:
1 hex digit->4 bits

Translate logical addresses (in hex) to physical addresses

```
0x0240:=>00 0010 0100 0000=>00 in segment 0=>Physical Address=0x2000+240=0x2240
0x1108:
0x265c:
```

Quiz: Address Translations with Segmentation

MMU contains Segment Table (per process)

- Each segment has own base and bounds, protection bits
- Example: 14 bit logical address, 4 segments; how many bits for segment? How many bits for offset?

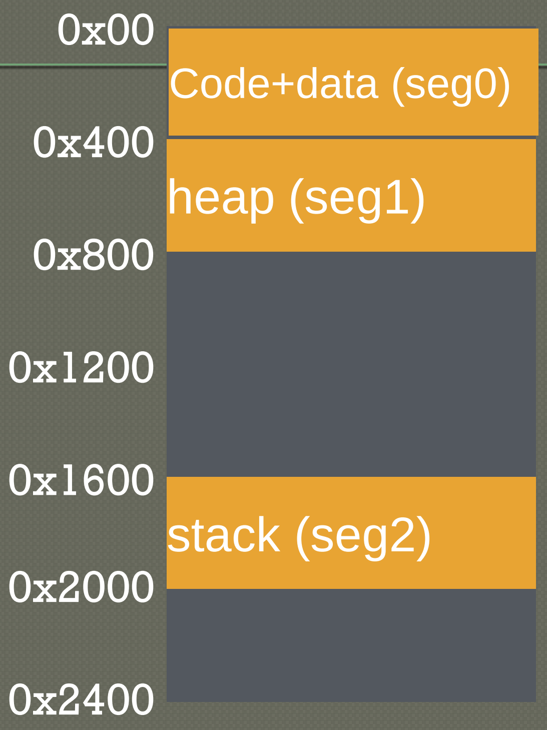
Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x000	0 0

remember:
1 hex digit->4 bits

Translate logical addresses (in hex) to physical addresses

```
0x0240:=>00 0010 0100 0000=>00 in segment 0=>Physical Address=0x2000+240=0x2240
0x1108:=>01 0001 0000 1000=>01 in segment 1=>Physical Address=0x0000+108=0x0108
0x265c:=>10 0110 0101 1100=>01 in segment 2=>Physical Address=0x3000+65c=0x365c
```

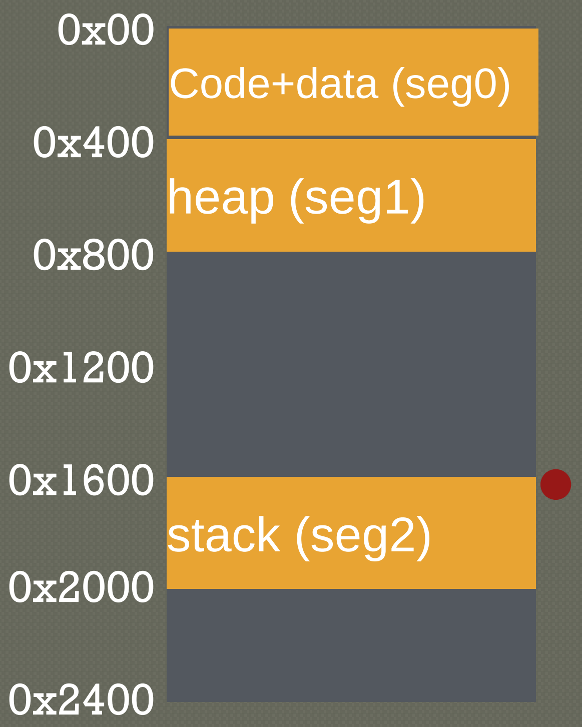
Visual Interpretation



Virtual (hex)	Physical
load 0x2010, R1	

Segment numbers:

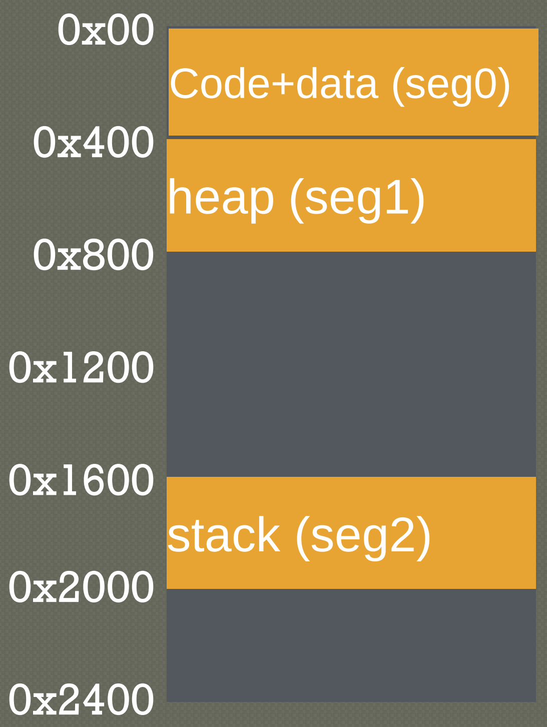
0: code+data
1: heap
2: stack



Virtual (hex)	Physical
load 0x2010, R1	0x1600 + 0x010 = 0x1610

Segment numbers:

0: code+data
1: heap
2: stack



0x00

Code+data (seg0)

0x400

heap (seg1)

0x800

0x1200

0x1600

stack (seg2)

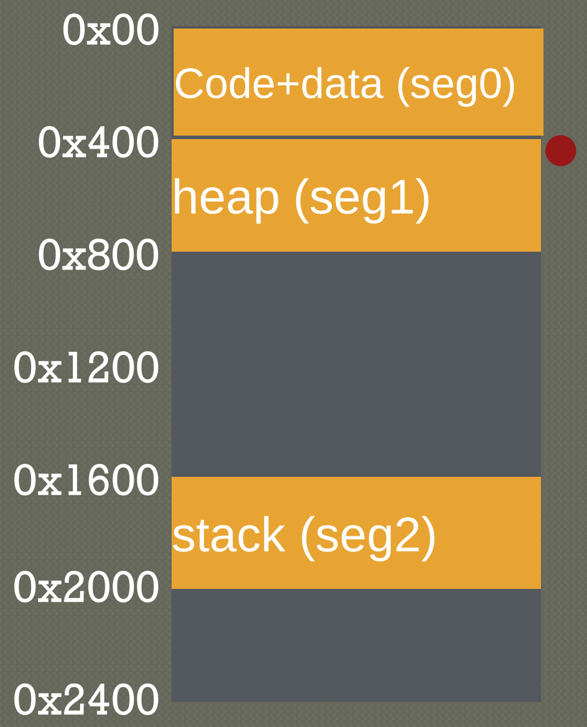
0x2000

0x2400

Virtual (hex)	Physical
load 0x2010, R1	0x1600 + 0x010 = 0x1610
load 0x1010, R1	

Segment numbers:

0: code+data
1: heap
2: stack



0x00

Code+data (seg0)

0x400

heap (seg1)

0x800

0x1200

0x1600

stack (seg2)

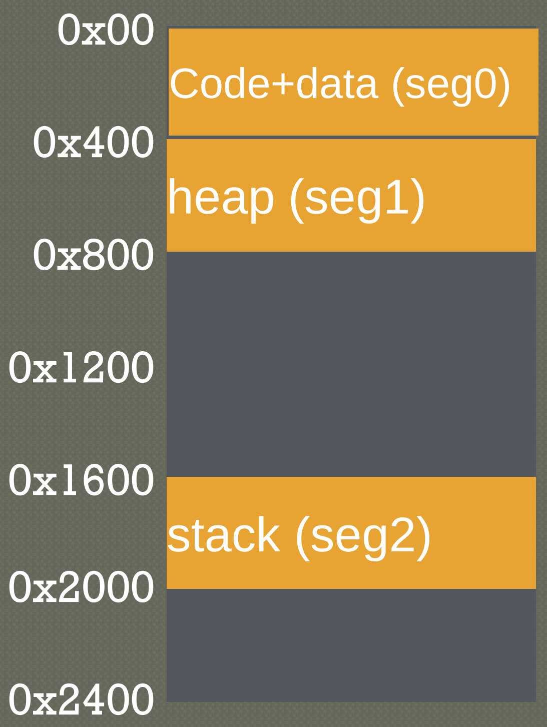
0x2000

0x2400

Virtual (hex)	Physical
load 0x2010, R1	0x1600 + 0x010 = 0x1610
load 0x1010, R1	0x400 + 0x010 = 0x410

Segment numbers:

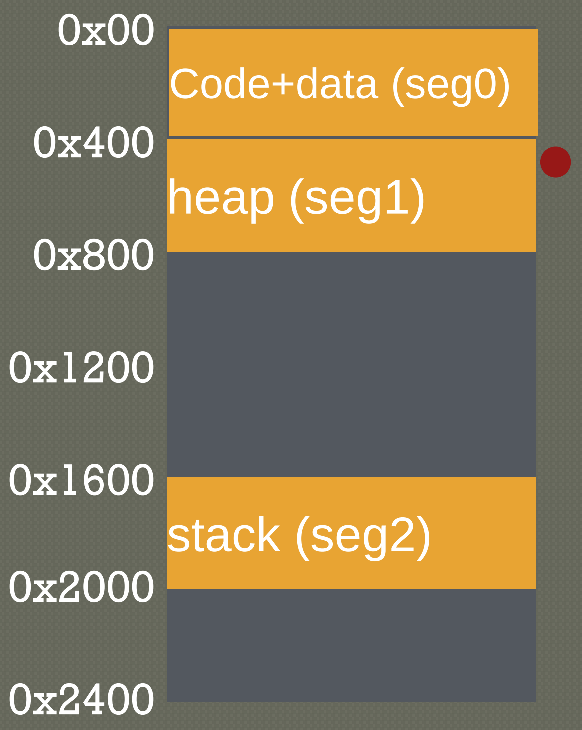
0: code+data
1: heap
2: stack



Virtual	Physical
load 0x2010, R1	0x1600 + 0x010 = 0x1610
load 0x1010, R1	0x400 + 0x010 = 0x410
load 0x1100, R1	

Segment numbers:

0: code+data
1: heap
2: stack

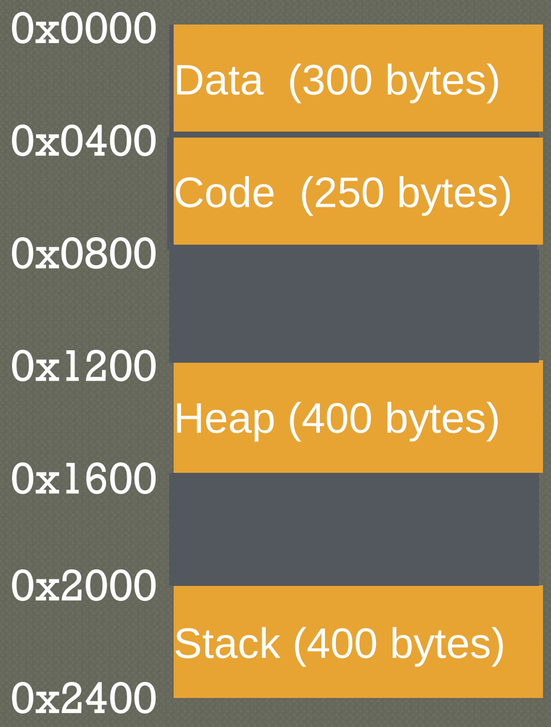


Virtual	Physical
load 0x2010, R1	0x1600 + 0x010 = 0x1610
load 0x1010, R1	0x400 + 0x010 = 0x410
load 0x1100, R1	0x400 + 0x100 = 0x500

Segment numbers:

0: code+data
1: heap
2: stack

Example:



0x0000

Data (300 bytes)

0x0400

Code (250 bytes)

0x0800

0x1200

Heap (400 bytes)

0x1600

0x2000

Stack (400 bytes)

0x2400

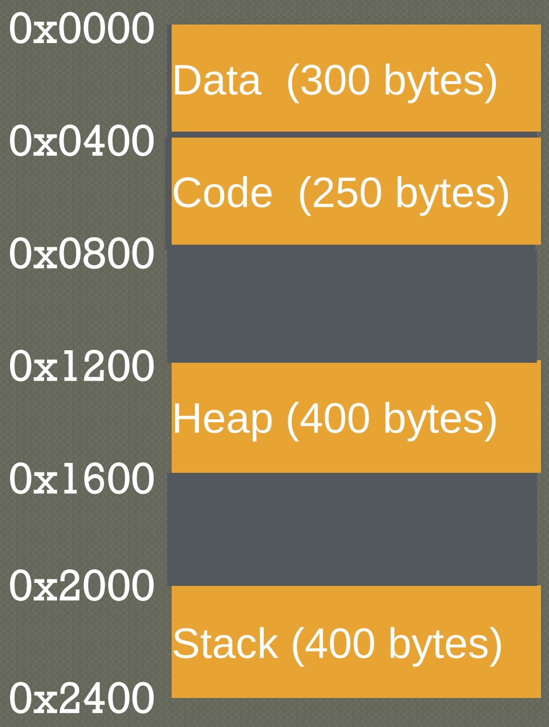
For a 14 bit system. Given the program layout in physical memory to the left. Fill in following segment table

Segment	Base	Bounds	R W
0			
1			
2			
3			

Segment numbers:

0: data
1: heap
2: stack
3:code

Example:



0x0000

Data (300 bytes)

0x0400

Code (250 bytes)

0x0800

0x1200

Heap (400 bytes)

0x1600

0x2000

Stack (400 bytes)

0x2400

For a 14 bit system. Given the program layout in physical memory to the left. Fill in following segment table

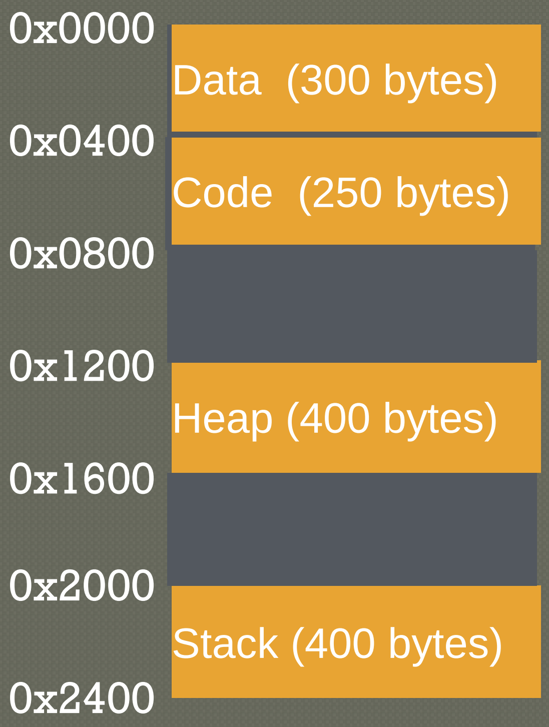
Segment	Base	Bounds	R W
0	0x0000	0x300	1 1
1	0x1200	0x400	1 1
2	0x2000	0x400	1 1
3	0x0400	0x250	0 0

Bounds is a size, so its 0x400 NOT 0x1600

Segment numbers:

0: data
1: heap
2: stack
3:code

Example:



Segment numbers:

0: data
1: heap
2: stack
3:code

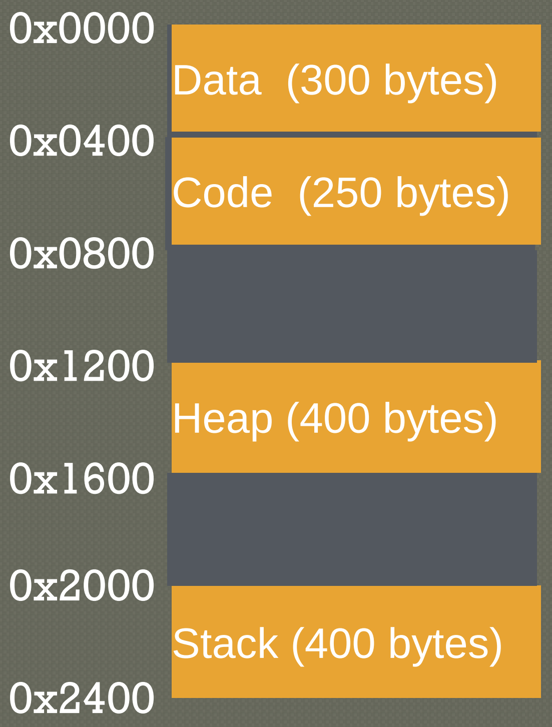
For a 14 bit system. Given the program layout in physical memory to the left. Fill in following segment table

Segment	Base	Bounds	R W
0	0x0000	0x300	1 1
1	0x1200	0x400	1 1
2	0x2000	0x400	1 1
3	0x0400	0x250	0 0

Given the following virtual addresses, what are the physical addresses? Or will the MMU throw an exception?

0x1010
0x3300

Example:

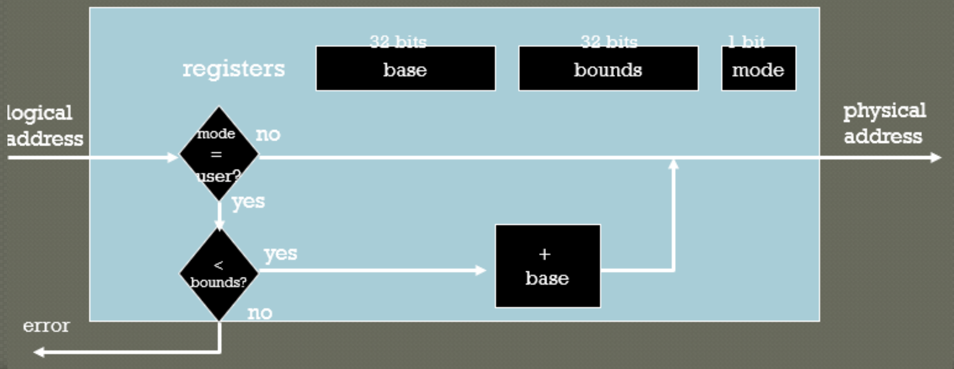


For a 14 bit system. Given the program layout in physical memory to the left. Fill in following segment table

Segment	Base	Bounds	R W
0	0x0000	0x300	1 1
1	0x1200	0x400	1 1
2	0x2000	0x400	1 1
3	0x0400	0x250	0 0

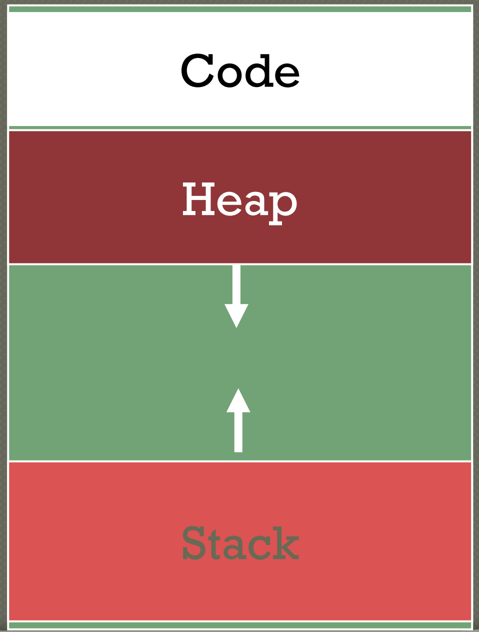
Given the following virtual addresses, what are the physical addresses? Or will the MMU throw an exception?

0x1010 =>seg 1=>0x1200+010 = 0x1210
0x3300 =>seg 3=>0x0400 + 300 (>0x250) =>FAULT!



Advantages of Segmentation

- Enables sparse allocation of address space
  - Stack and heap can grow independently
  - Heap: If no data on free list, dynamic memory allocator requests more from OS (e.g., UNIX: malloc calls sbrk())
  - Stack: OS recognizes reference outside legal segment, extends stack implicitly
- Different protection for different segments
  - Read-only status for code
- Enables sharing of selected segments
- Supports dynamic relocation of each segment



Disadvantages of Segmentation

Each segment must be allocated contiguously

- May not have sufficient physical memory for large segments

Fix in next lecture with paging…

Conclusion

HW+OS work together to virtualize memory

- Give illusion of private address space to each process

Add MMU registers for base+bounds so translation is fast

- OS not involved with every address translation, only on context switch or errors

Dynamic relocation with segments is good building block

- Next lecture: Solve fragmentation with paging