OSTEP
Memory Virtualization
Virtual Memory

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?

More Virtualization

1st part of course: Virtualization

Virtual CPU: *illusion* of **private CPU registers**
- 2 lectures (mechanism + policy)

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

Memory Virtualization

What is **memory virtualization**?

- OS virtualizes its physical memory
- OS provides an illusion memory space per each process
- It seems to be seen like each process uses the whole memory

Benefit of Memory Virtualization

Ease of use in programming

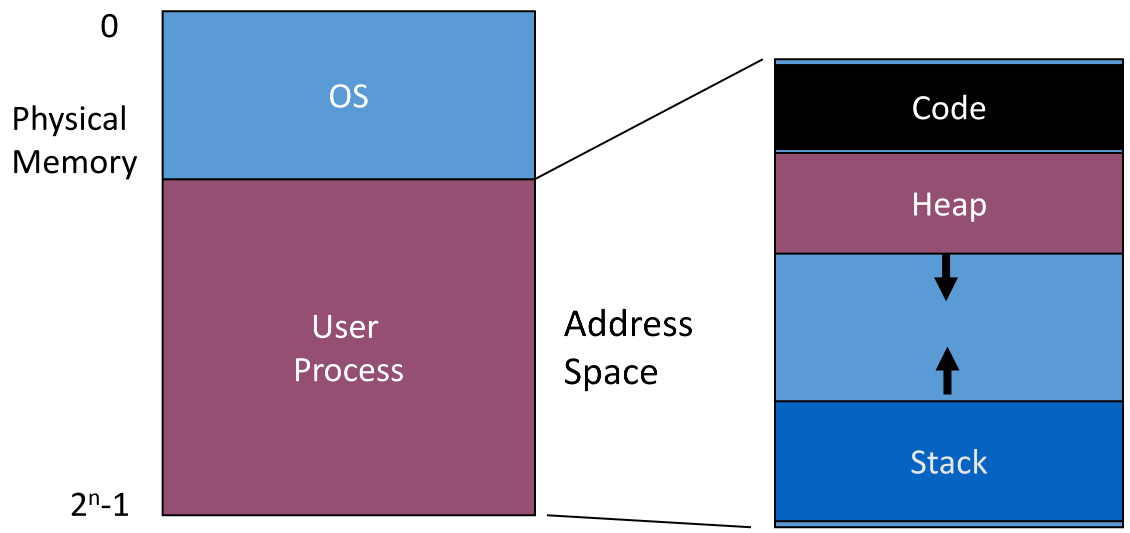
Memory efficiency in terms of times and space

The guarantee of isolation for processes as well as OS

- Protection from errant accesses of other processes

Motivation for Virtualization

Uniprogramming: One process runs at a time



Disadvantages:

- Only one process runs at a time
- Process can destroy OS
- Poor utilization and efficiency

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

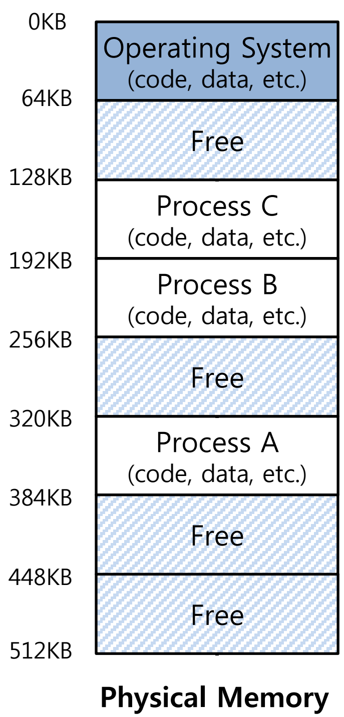
Multiprogramming and Time Sharing

Load multiple processes in memory.

- Execute one for a short while.
- Switch processes between them in memory.
- Increase utilization and efficiency.

Cause an important **protection issue**.

- Errant memory accesses from other processes



Abstraction: Address Space

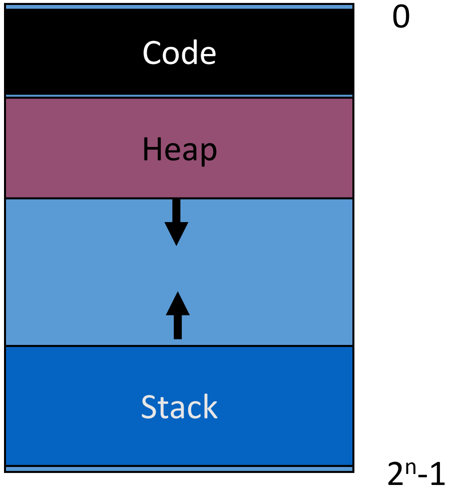
Address space: Each process has set of addresses that map to bytes

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

Review: What is in an address space?

Extend LDE (limited direct execution)

Address space has static and dynamic components
Static: Code and some global variables
Dynamic: Stack and Heap



Address Space(CONT.)

Code

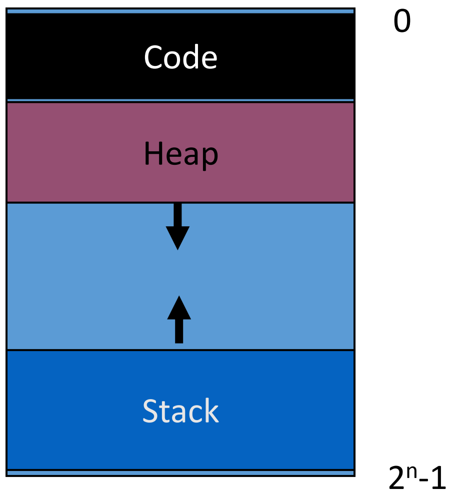
- Where instructions live

Heap

- Dynamically allocate memory.
  - `malloc` in C language
  - `new` in object-oriented language

Stack

- Store return addresses or values
- Contain local variables arguments to routines



Virtual Address

Every address in a running program is virtual.

- OS translates the virtual address to physical address

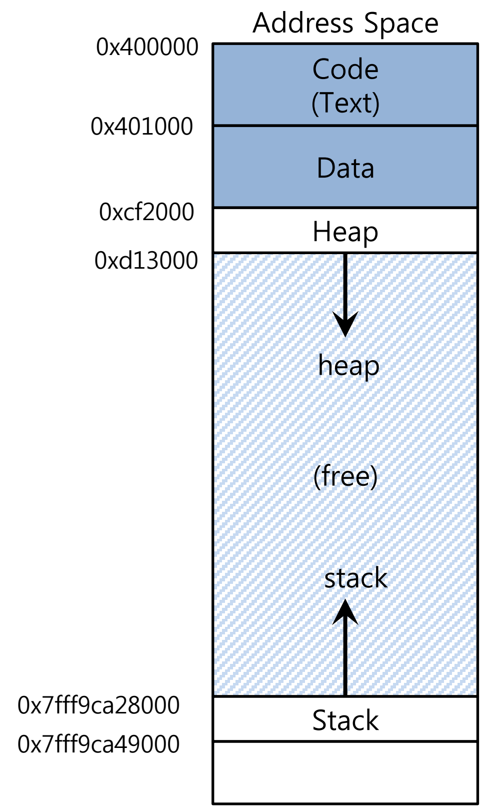
```
#include <stdio.h>
#include <stdlib.h>

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;
}
```

A simple program that prints out addresses

Virtual Address(Cont.)

The output in 64-bit Linux machine

```
location of code  : 0x40057d
location of heap  : 0xcf2010
location of stack : 0x7fff9ca45fcc
```



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
  - Allocate enough for worst possible case; 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

Where Are stacks Used?

OS uses stack for procedure call 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??

- 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;
}
```

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

```
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)

Memory Accesses to what addresses?

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

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):

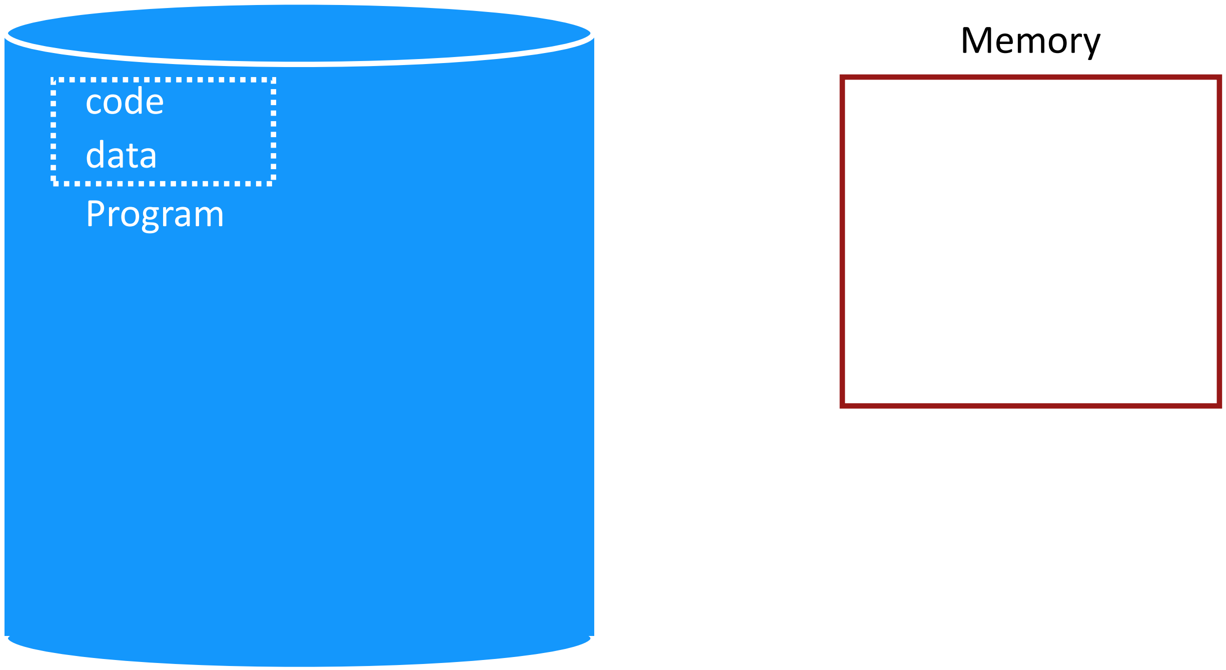
1. Time Sharing
2. Static Relocation
3. Base
4. Base+Bounds
5. Segmentation

1) Time Sharing of Memory

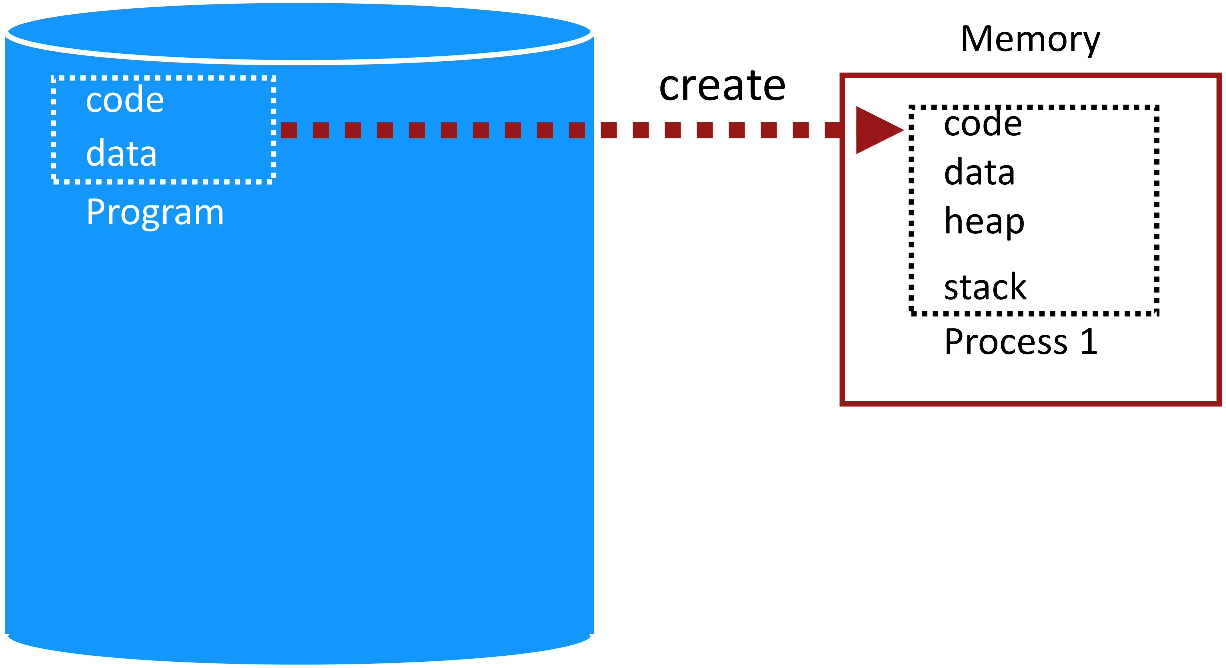
Try similar approach to how OS virtualizes CPU

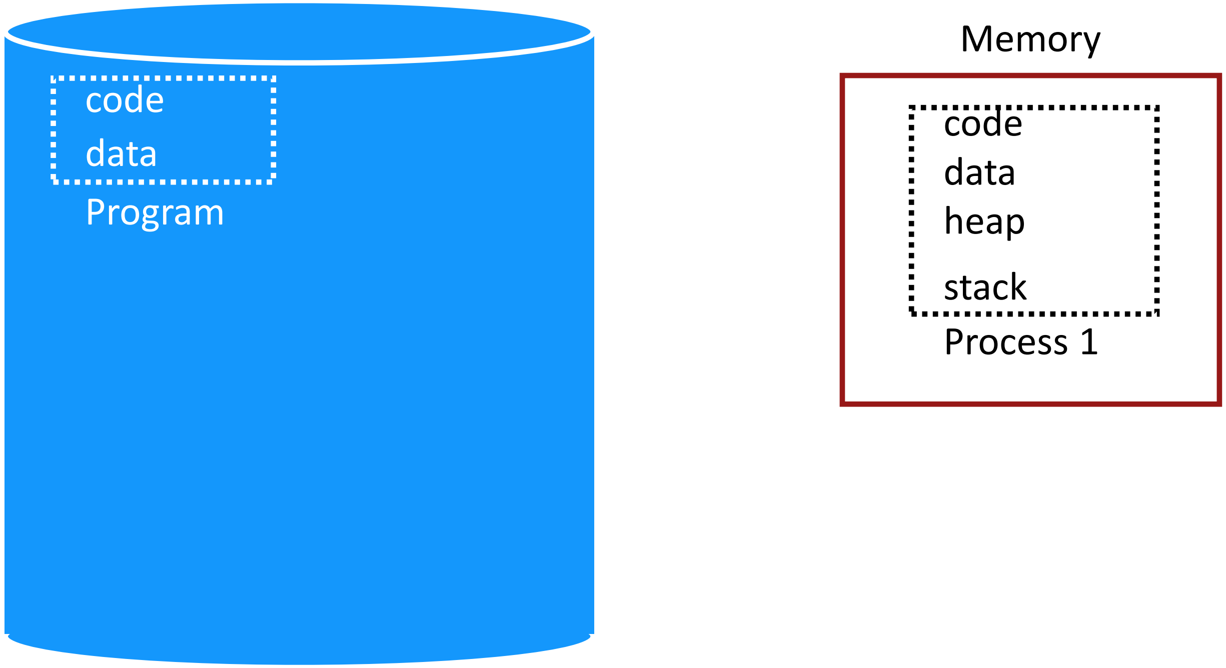
Observation:
OS gives illusion of many virtual CPUs by saving **CPU registers** to **memory** when a process isn't running

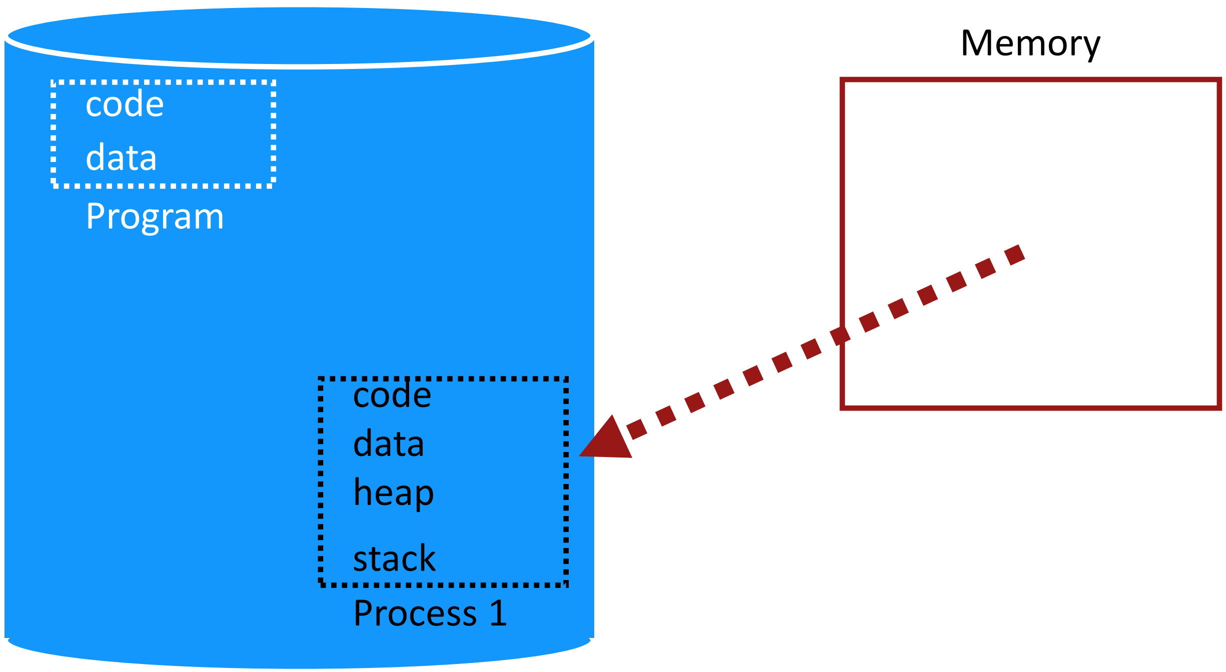
Could give illusion of many virtual memories by saving **memory** to **disk** when process isn't running

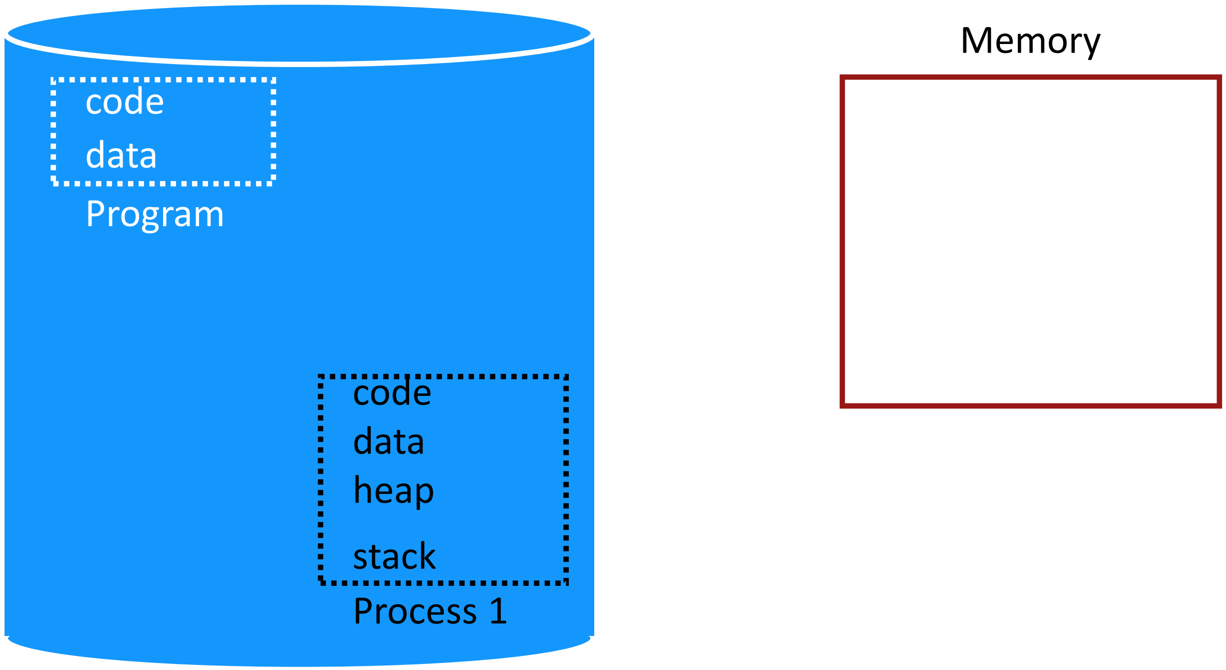


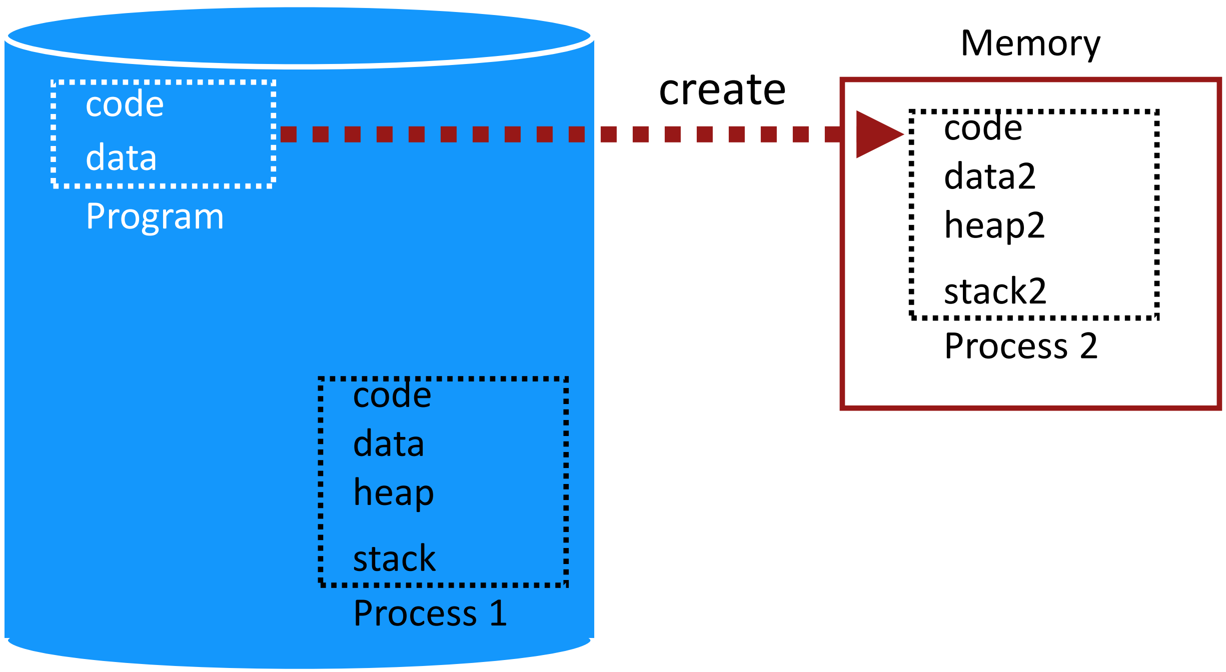
Time Share Memory: Example


code
data
Program
create
Memory
code
data
heap
stack
Process 1

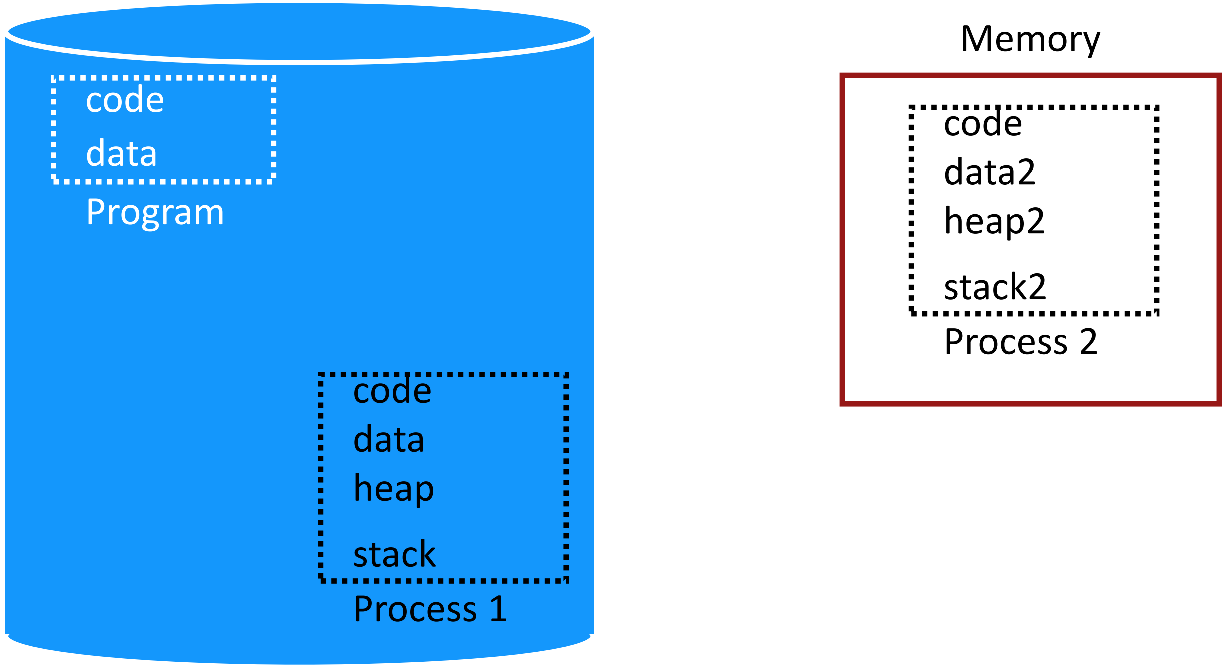

code
data
Program
Memory
code
data
heap
stack
Process 1

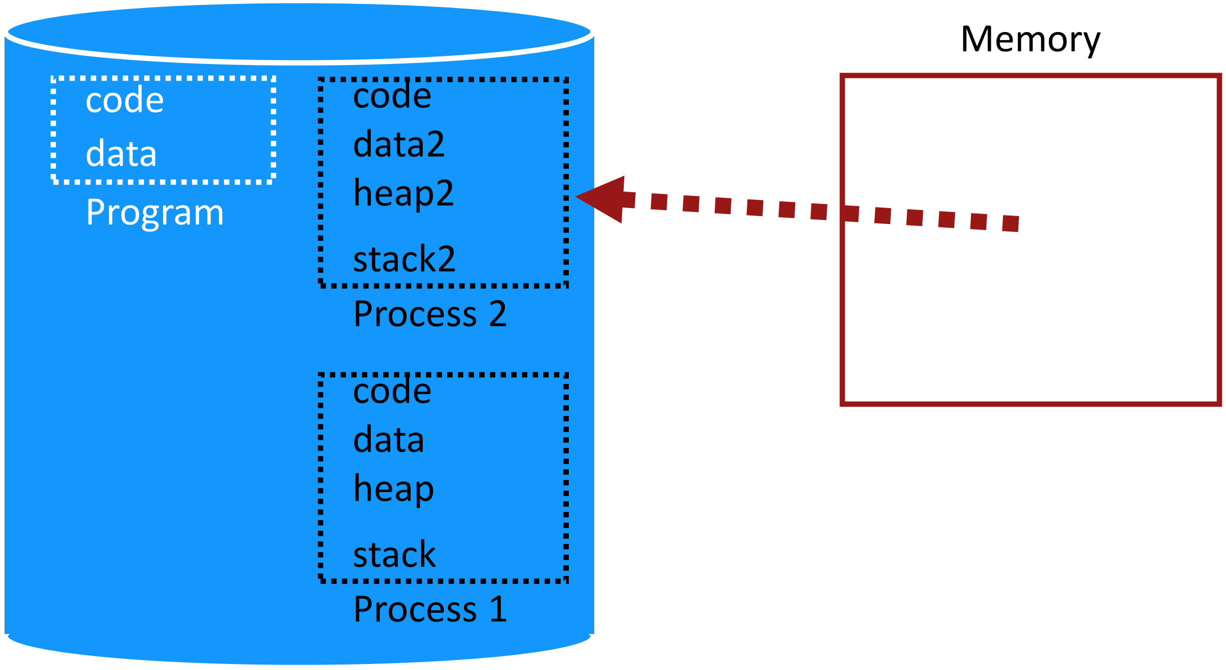

code
data
Program
code
data
heap
stack
Process 1
Memory

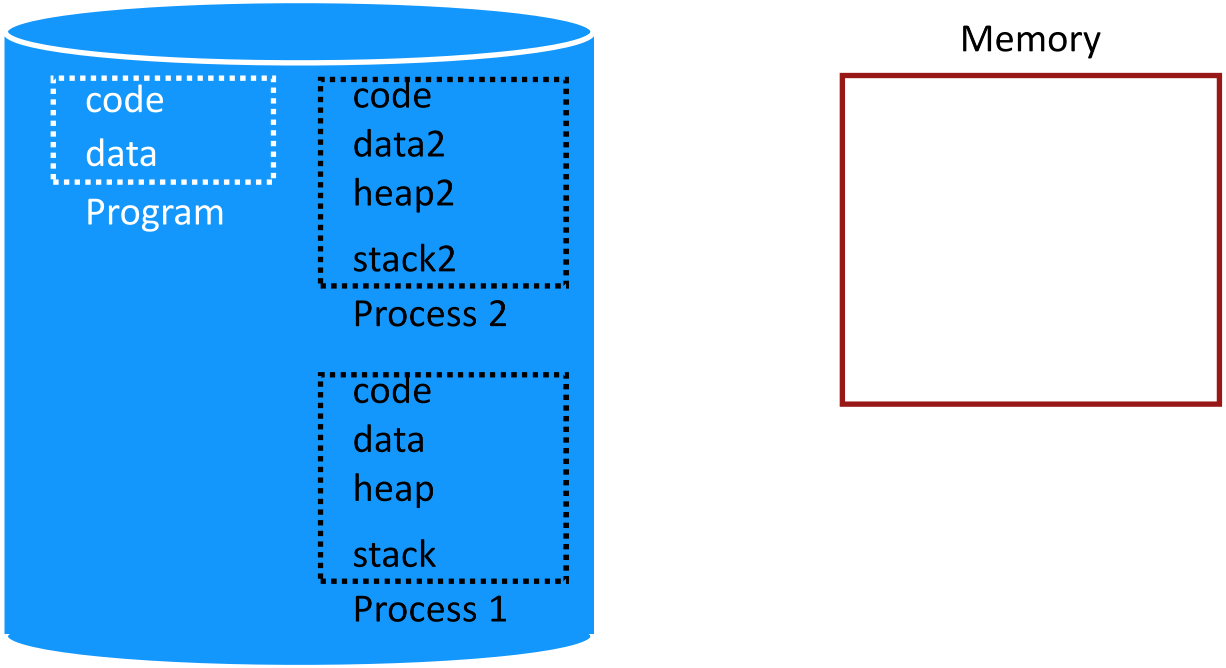

code
data
Program
code
data
heap
stack
Process 1
Memory

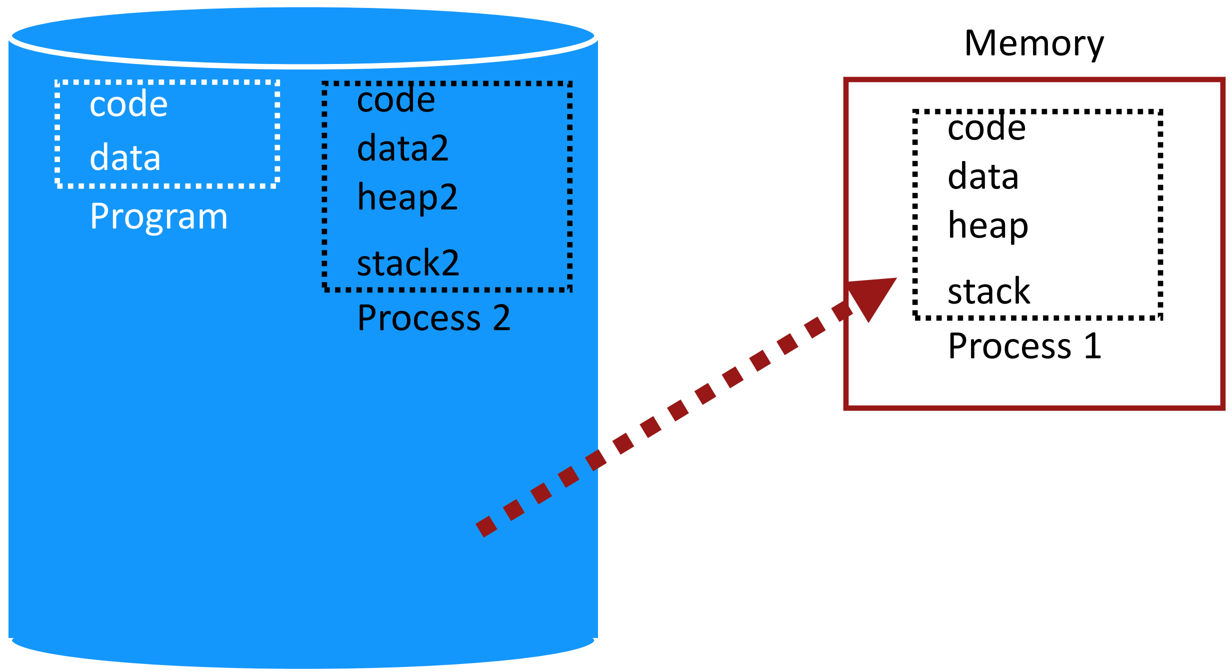


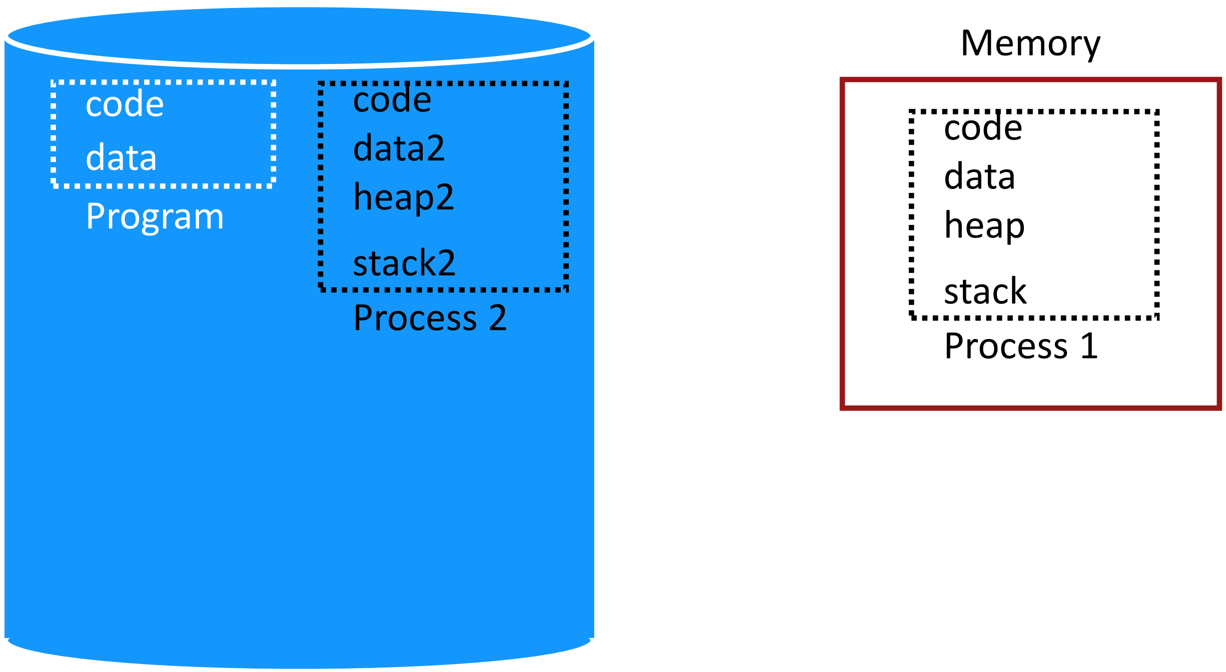
code
data
Program
create
Memory
code
data2
heap2
stack2
Process 2
code
data
heap
stack
Process 1


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

Problems with Time Sharing Memory

Problem: Ridiculously poor performance

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
  Can any addresses be unchanged?

0x1010: movl0x8(%rbp), %edi
0x1013: addl $0x3, %edi
0x1019: movl%edi, 0x8(%rbp)

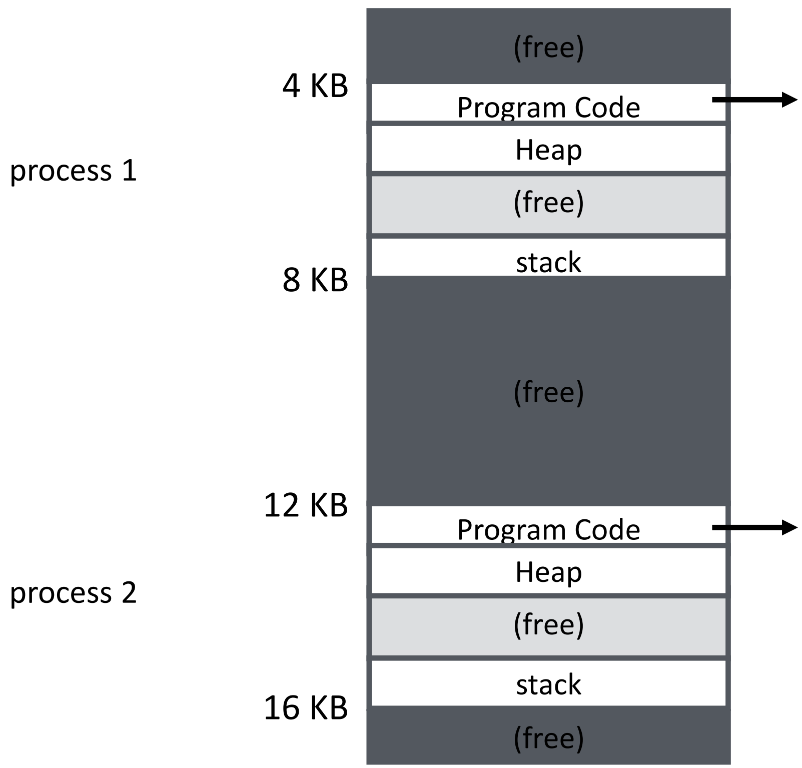
rewrite

- 0x10: movl0x8(%rbp), %edi
- 0x13: addl $0x3, %edi
- 0x19: movl%edi, 0x8(%rbp)

rewrite

0x3010: movl0x8(%rbp), %edi
0x3013: addl $0x3, %edi
0x3019: movl%edi, 0x8(%rbp)

Static: Layout in Memory




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

```
0x3010:  movl0x8(%rbp), %edi
0x3013:  addl $0x3, %edi
0x3019:  movl%edi, 0x8(%rbp)
```

why didn’t OS rewrite stack addr?

Static Relocation: Disadvantages

No protection

- Process can destroy OS or other processes
- No privacy

Cannot move address space after it has been placed

- May not be able to allocate new process

3) Dynamic Relocation

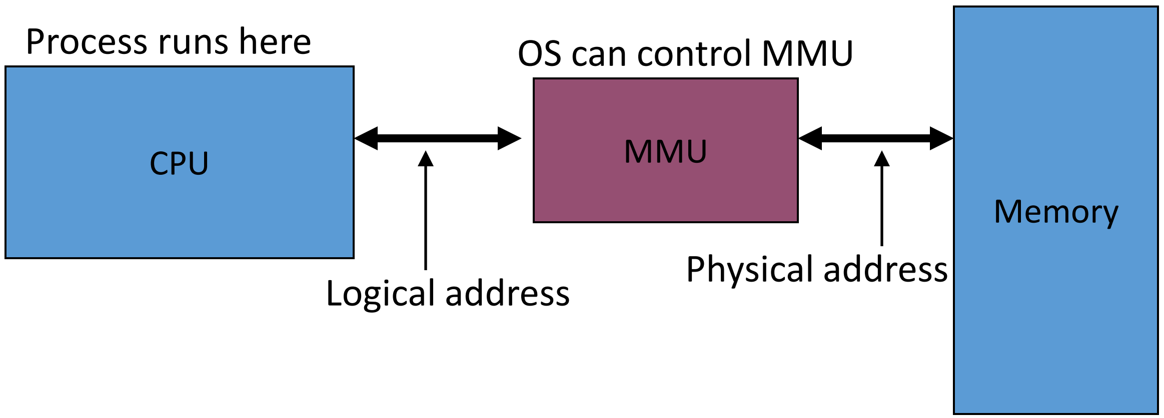
Goal: Protect processes from one another

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**

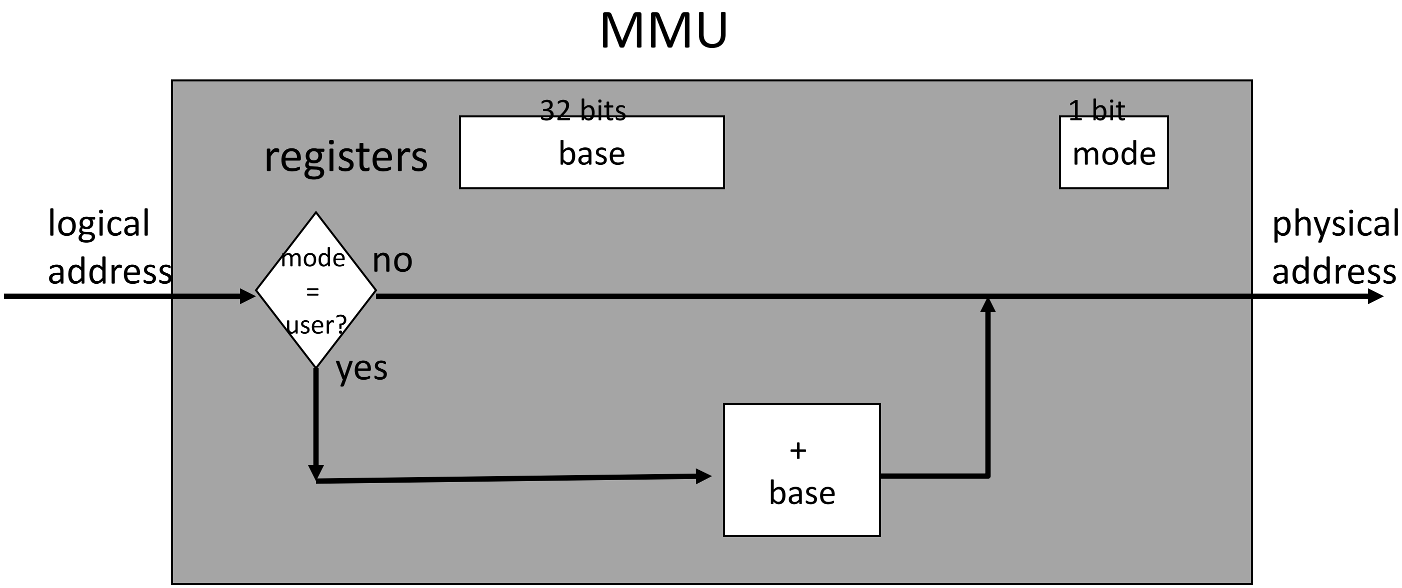
Minimal MMU contains **base register** for translation

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

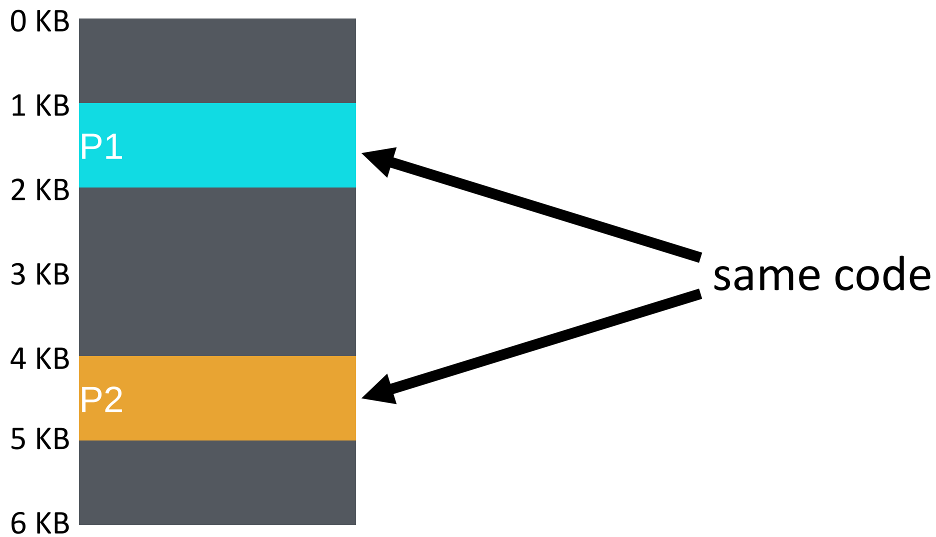


Dynamic Relocation with Base Register

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

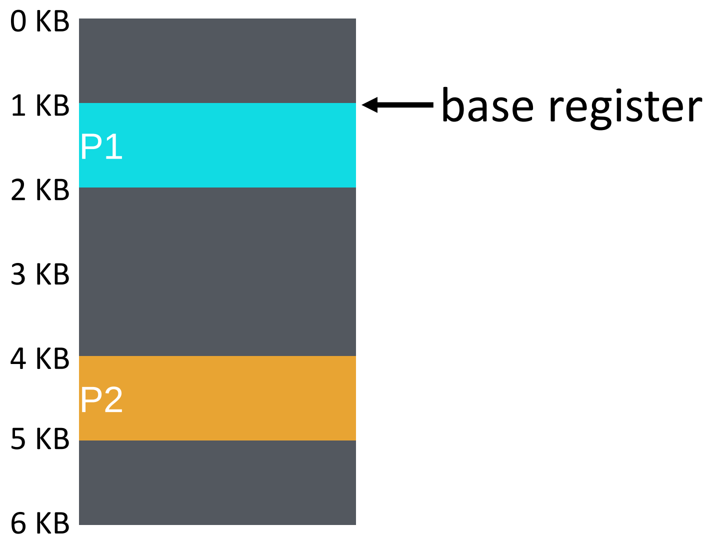
Store offset in base register

Each process has different value in base register

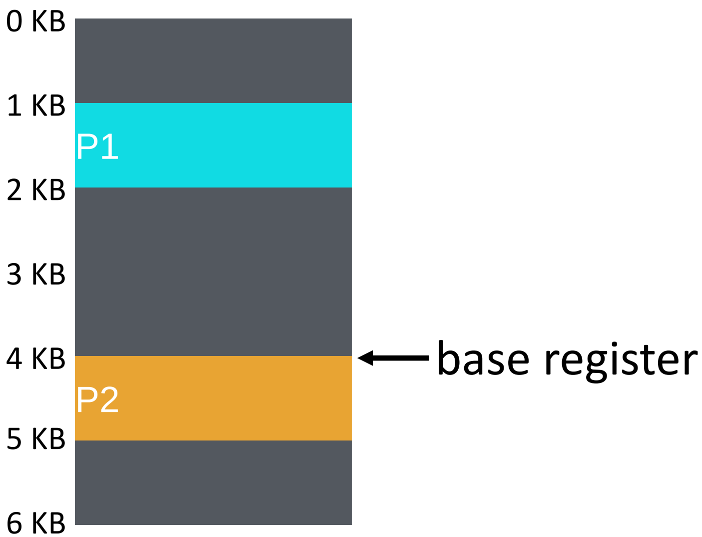


VISUAL Example of DYNAMIC RELOCATION:

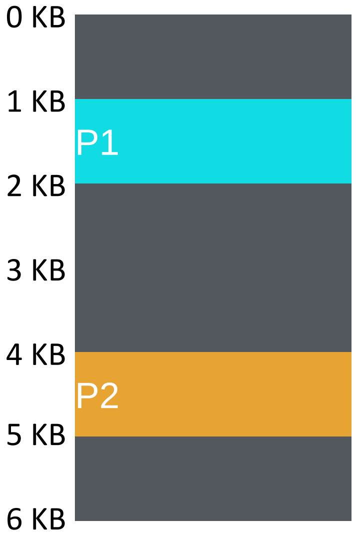
BASE REGISTER



P1 is running



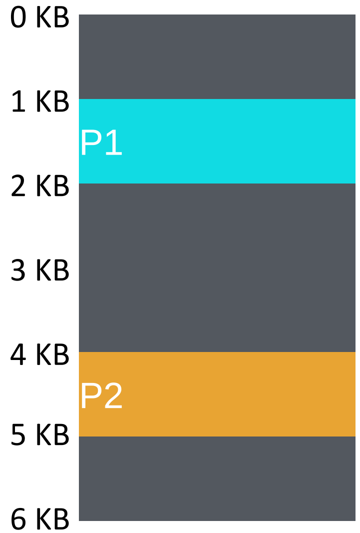
P2 is running



(Decimal notation)

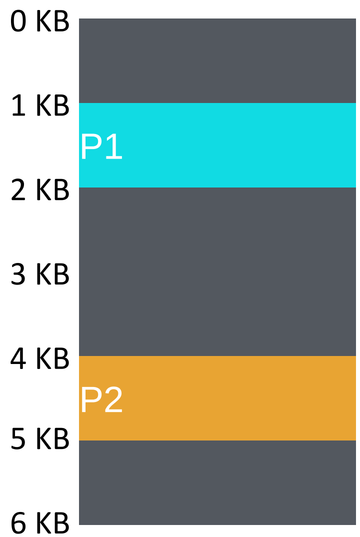
Virtual	Physical
P1: load 100, R1	

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



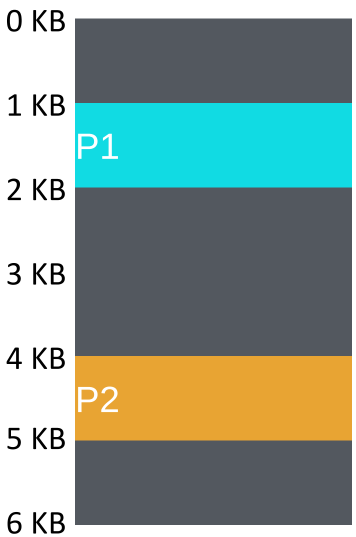
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	

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

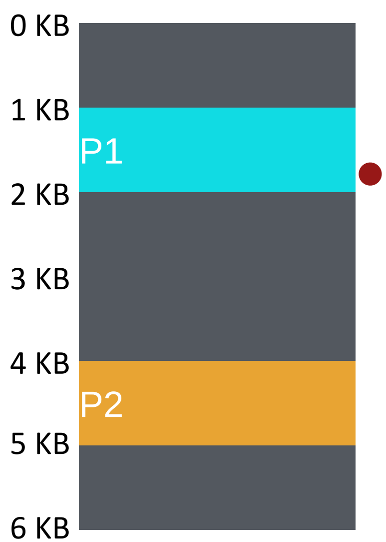


Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	

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	

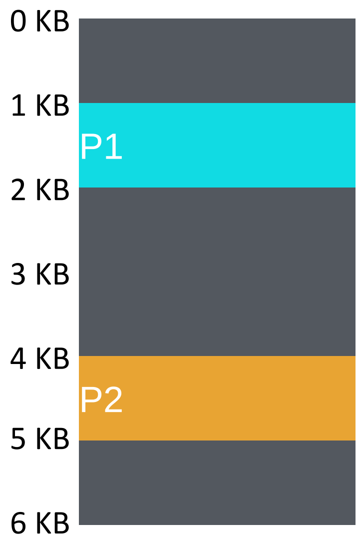


Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 1000, R1	load 2024, R1

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

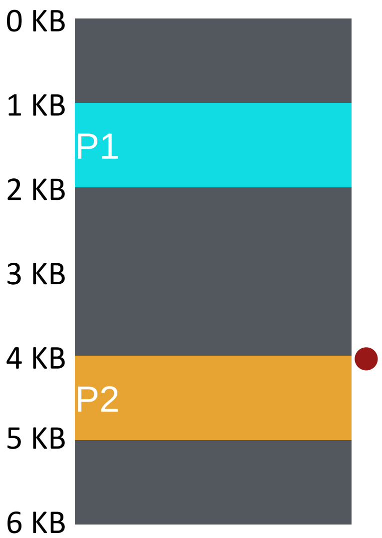


Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1

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?

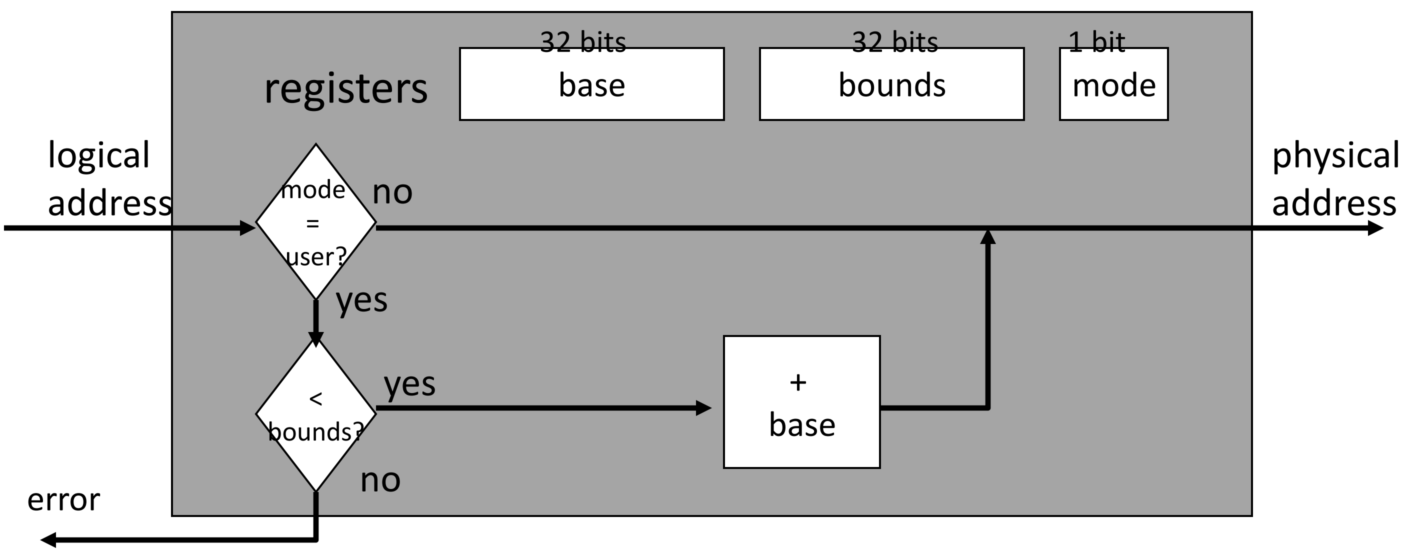
4) Dynamic with Base+Bounds

- 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)
- What happens if you load/store after bounds?
  OS kills process if process loads/stores beyond bounds

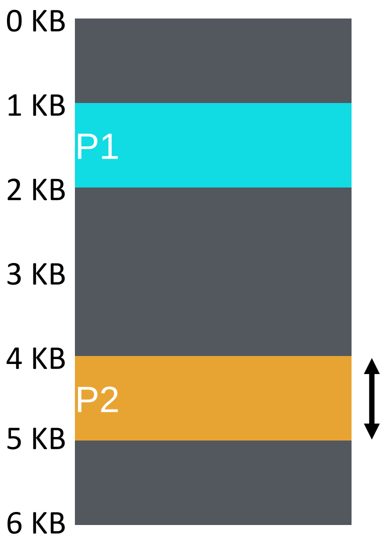
Implementation of BASE+BOUNDS

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



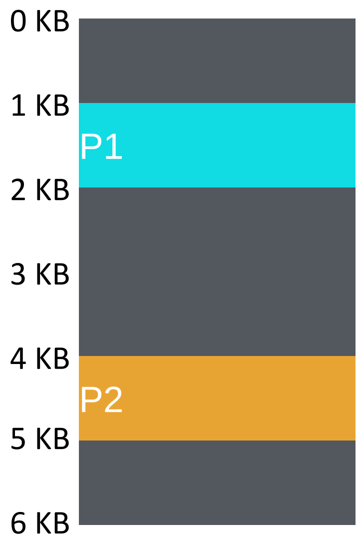
P1 is running



base register

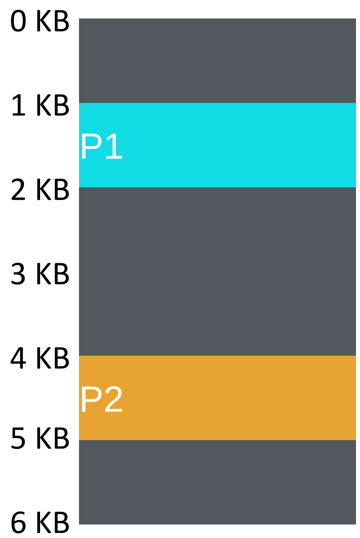
bounds register

P2 is running



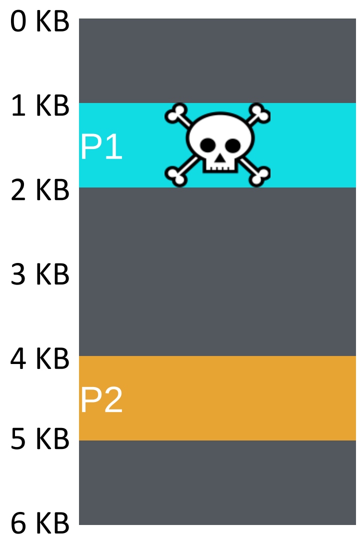
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	

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

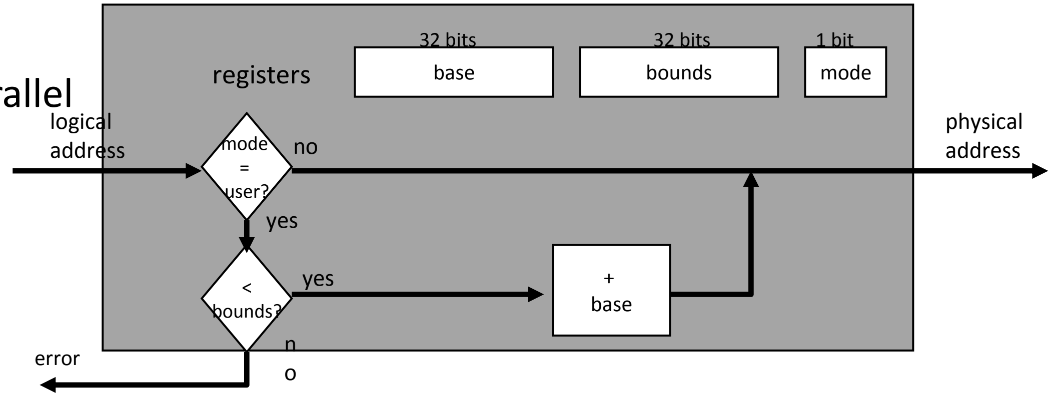
- User process cannot change base and bounds registers
- User process cannot change to privileged mode

Base and Bounds

Advantages

- 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

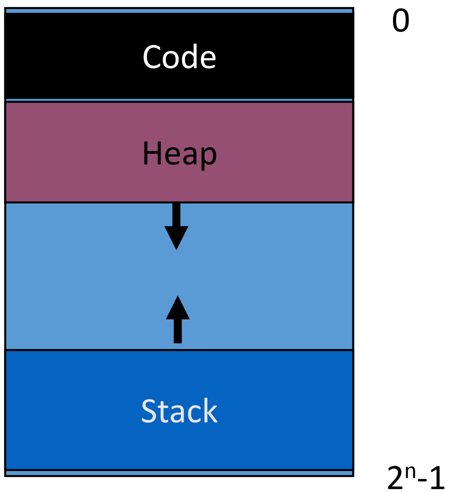


Base and Bounds

Disadvantages

- 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



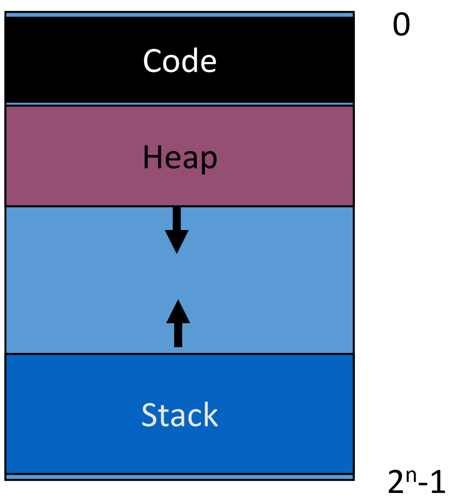
5) Segmentation

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)

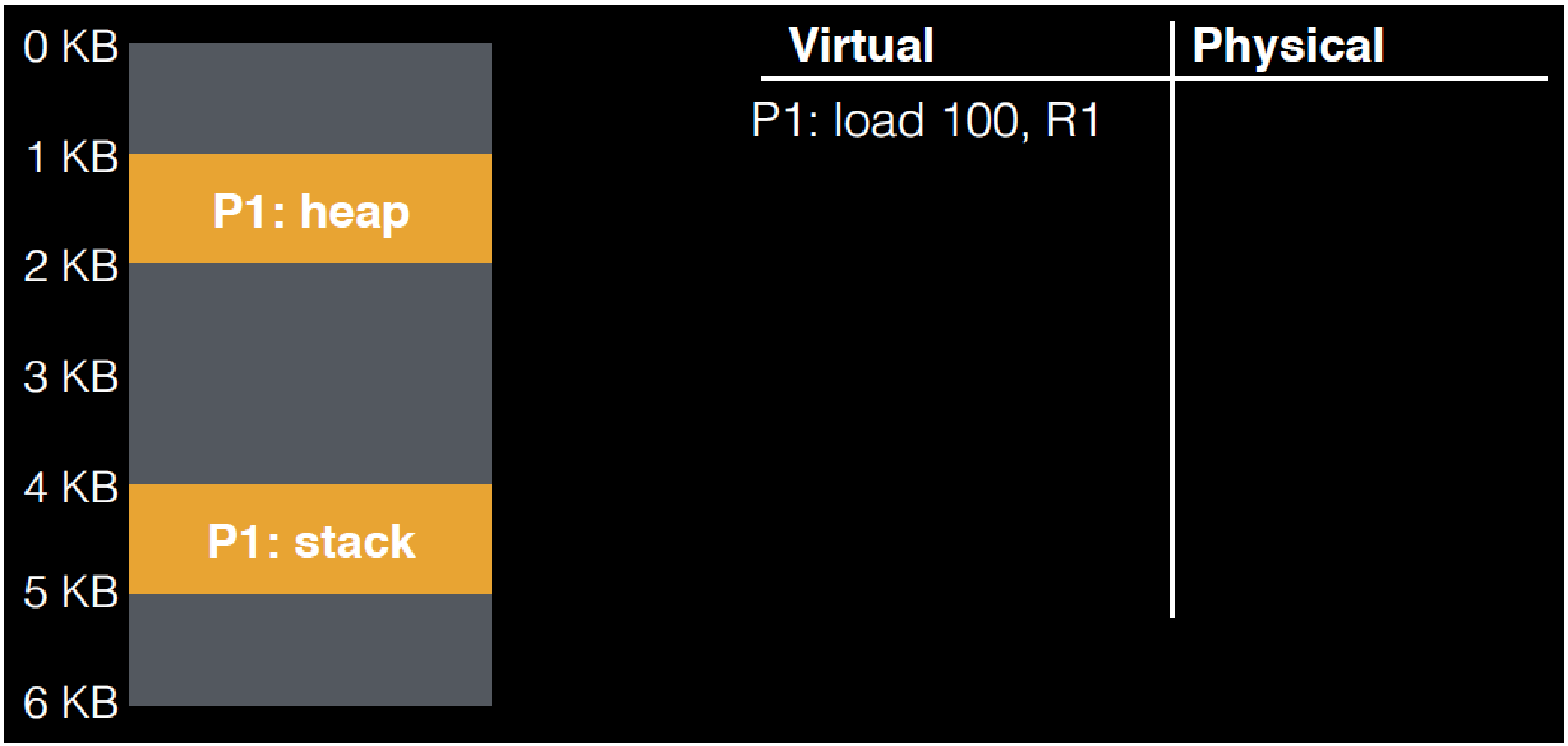


Multi-segment translation

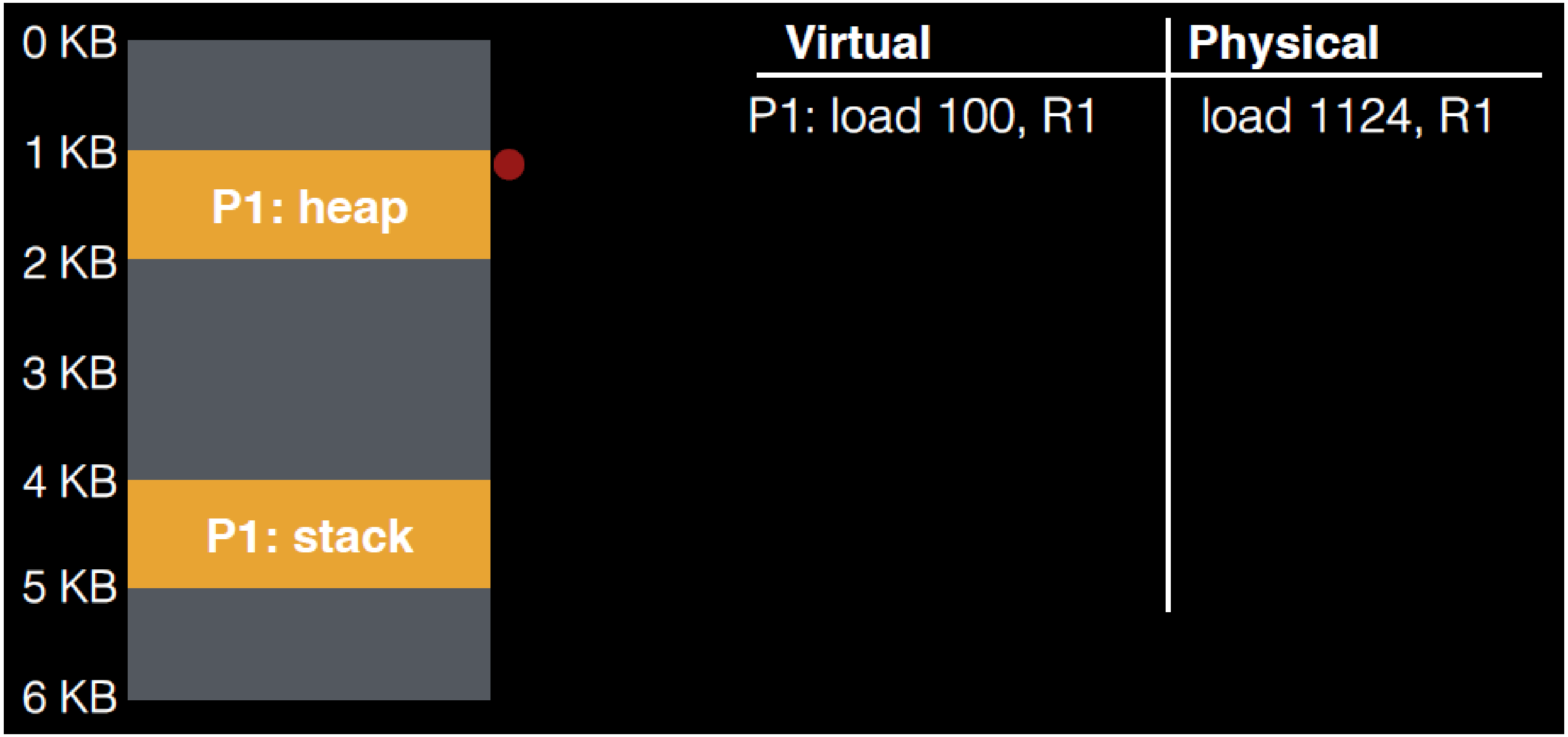
One (broken) approach:

- 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)

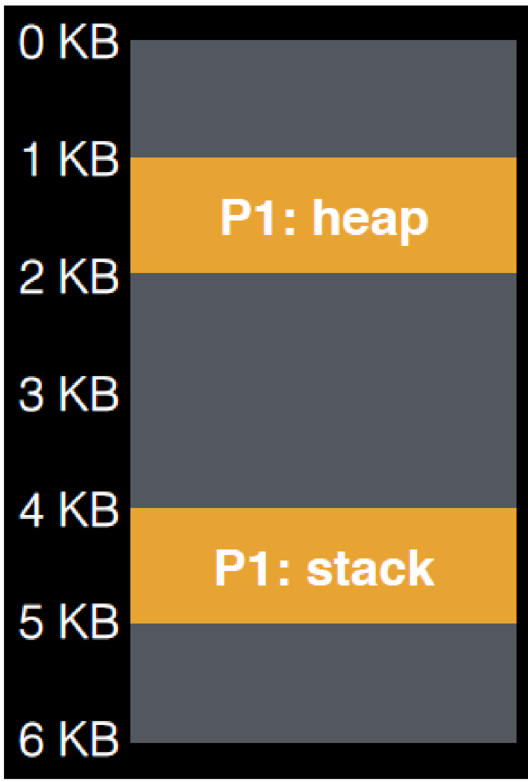
A tricky example



A tricky example

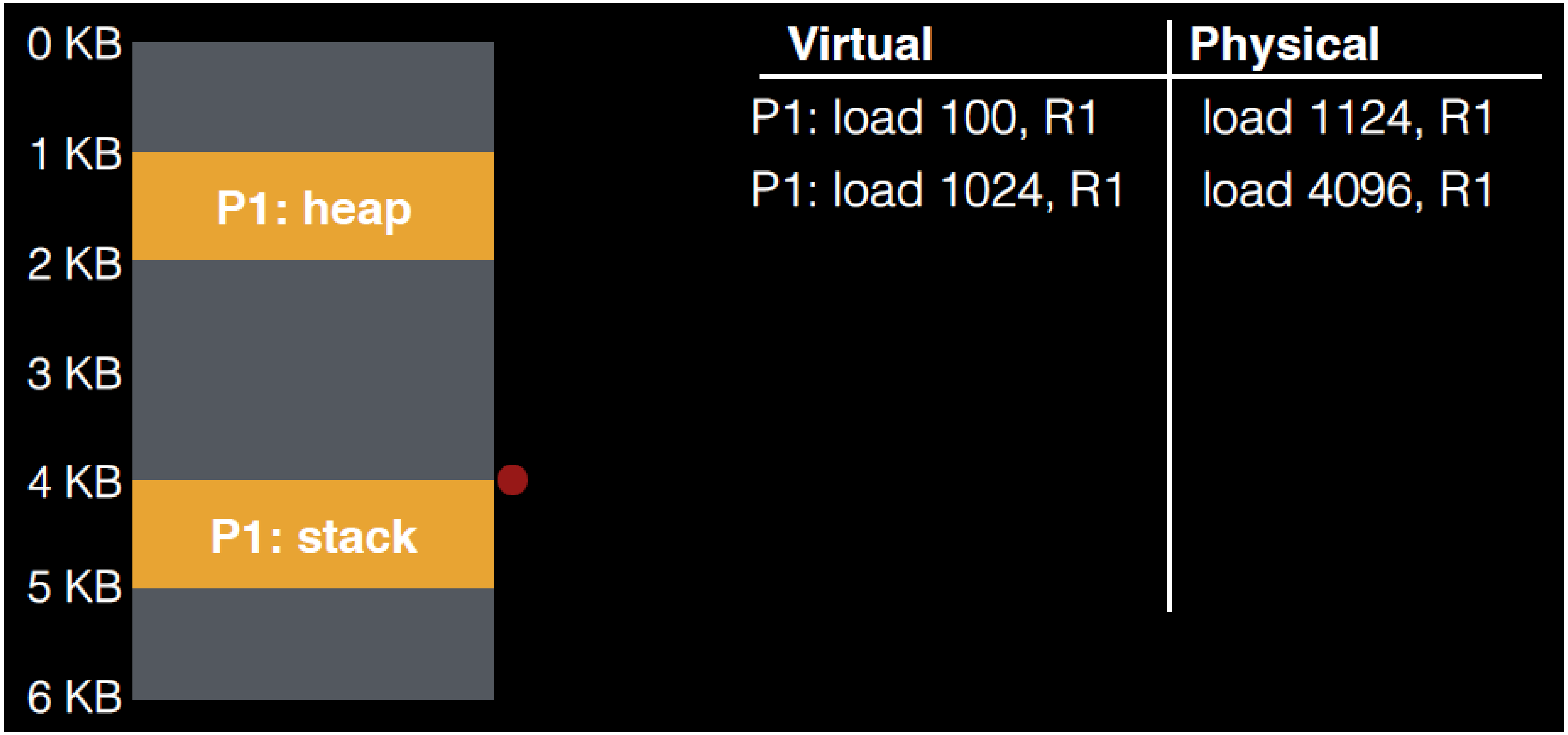


A tricky example

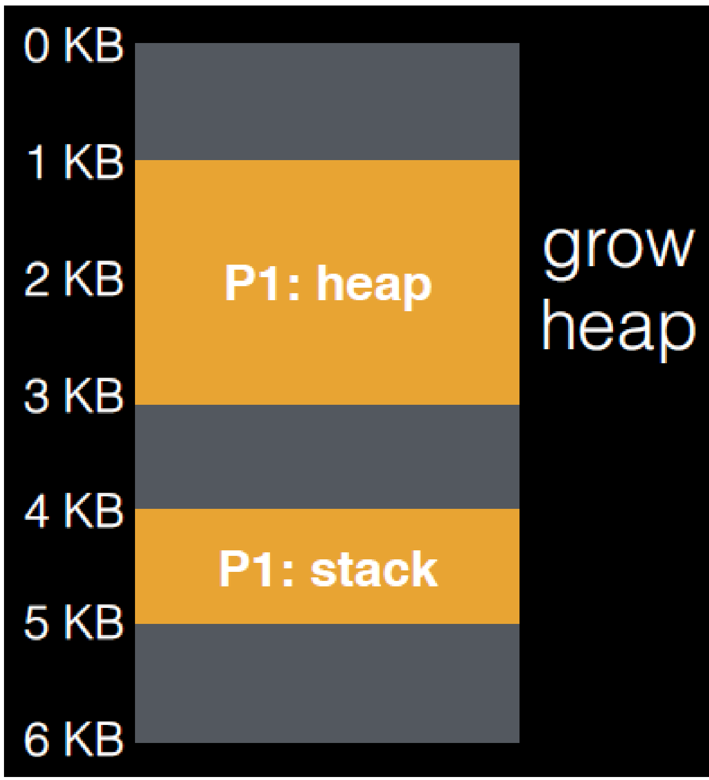


Virtual	Physical
P1: load 100, R1	load 1124, R1
P1: load 1024, R1	

A tricky example

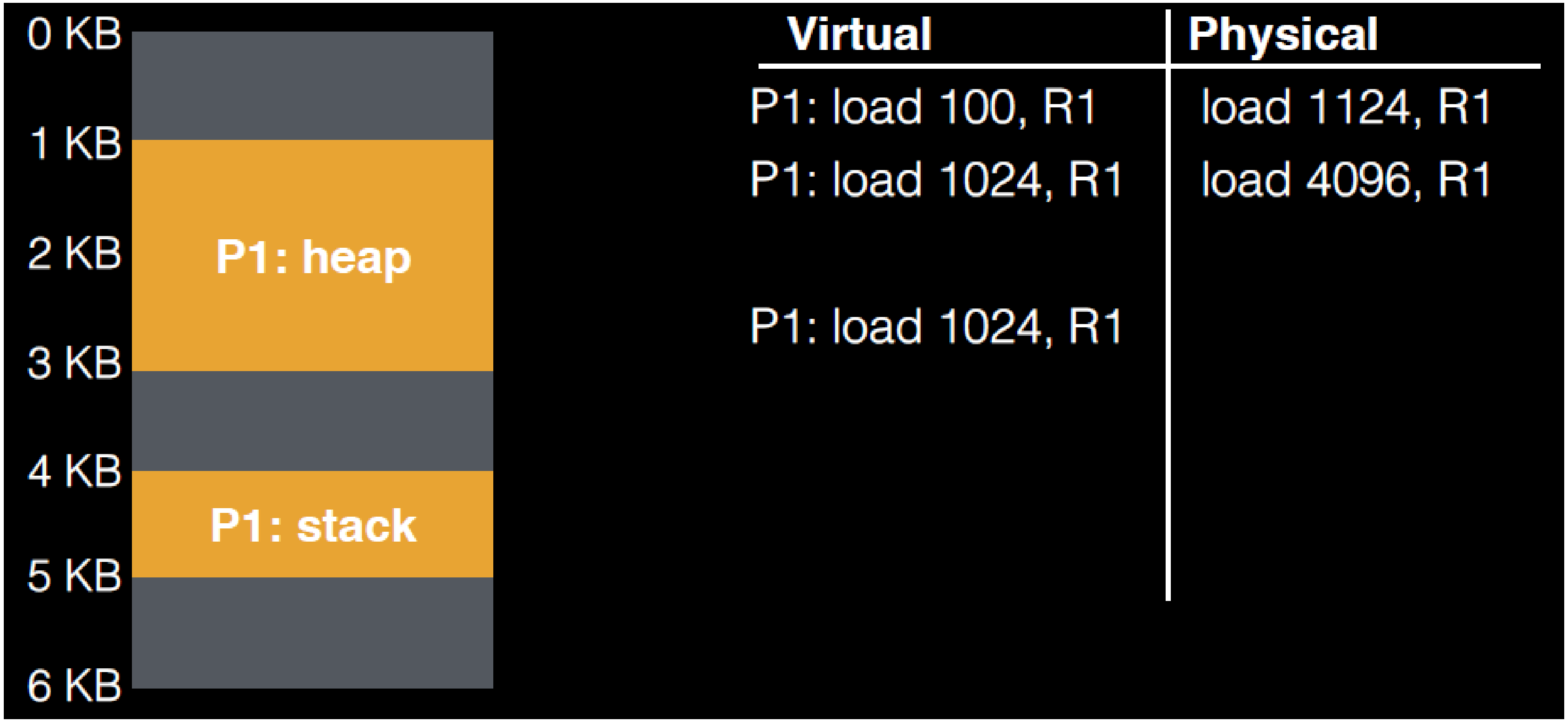


A tricky example

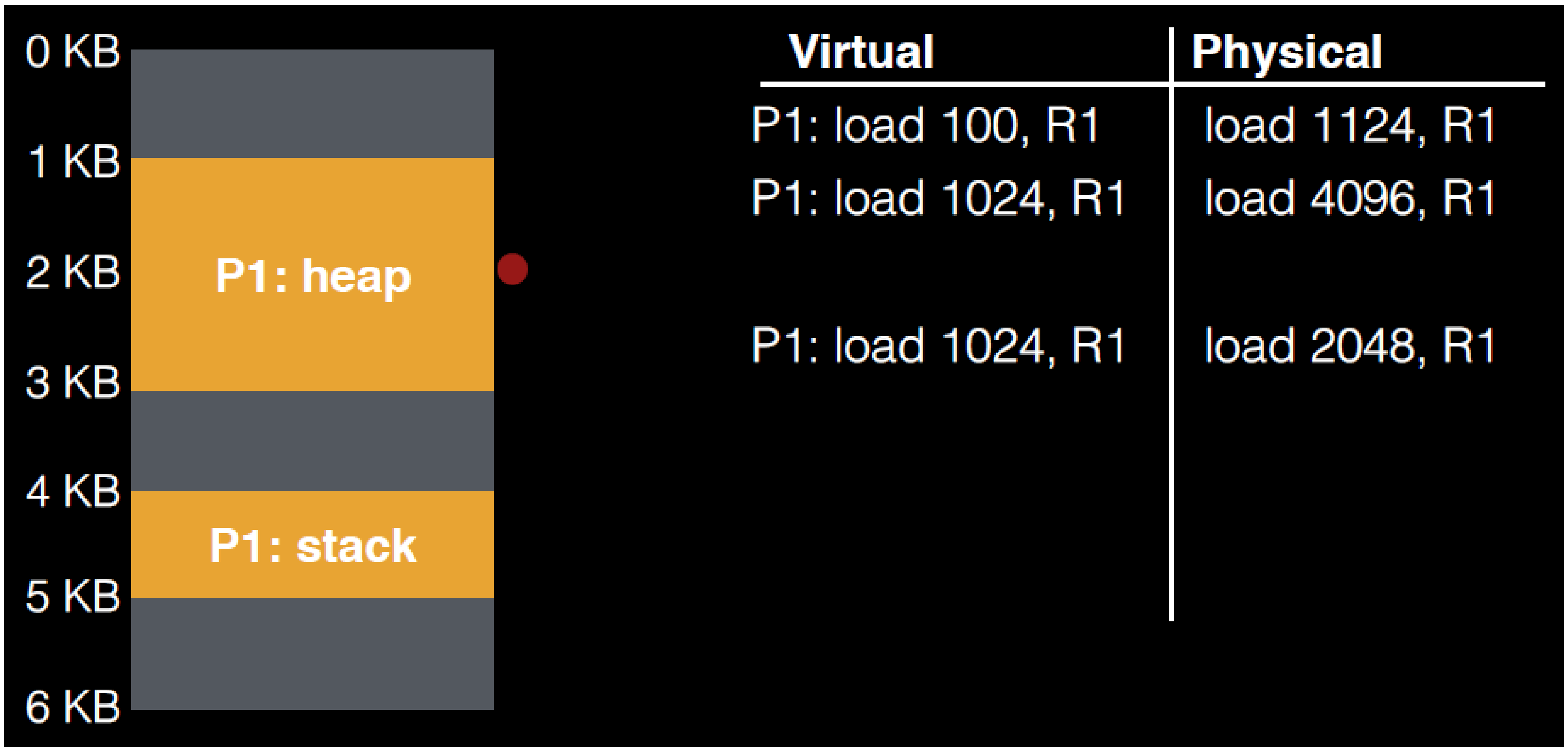


Virtual	Physical
P1: load 100, R1	load 1124, R1
P1: load 1024, R1	load 4096, R1

A tricky example



A tricky example



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

segment 2

offset 30

Choose some segment numbering, such as

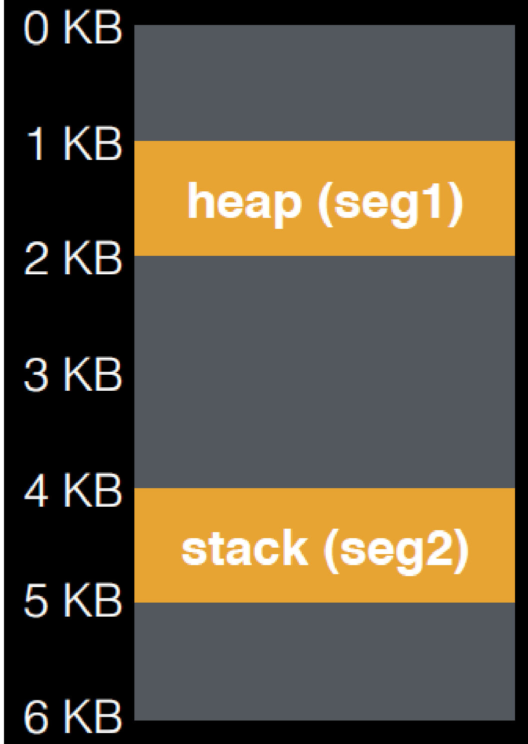
0: code+data
1: heap
2: stack

What is the segment/offset?

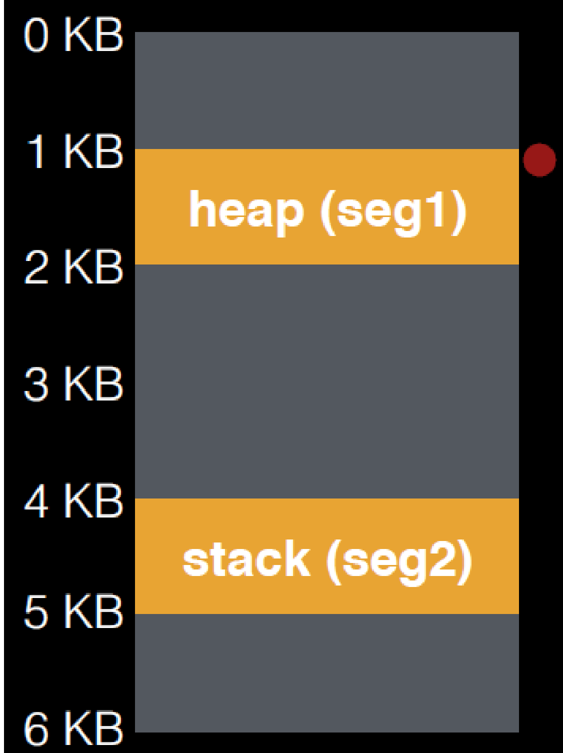
Segment numbers:
- 0: code+data
- 1: heap
- 2: stack

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

Virtual	Physical
load 0x2010, R1	4KB + 16

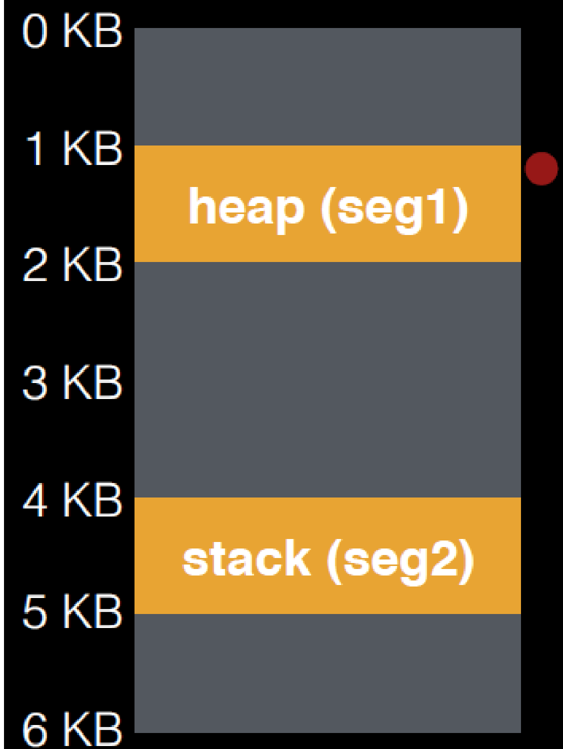


Virtual	Physical
load 0x2010, R1	4KB + 16
load 0x1010, R1	



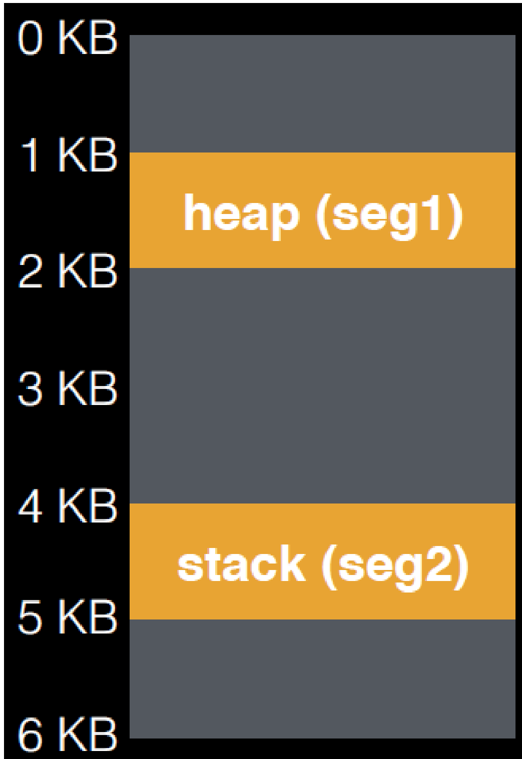
Virtual	Physical
load 0x2010, R1	4KB + 16
load 0x1010, R1	1KB + 16

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	

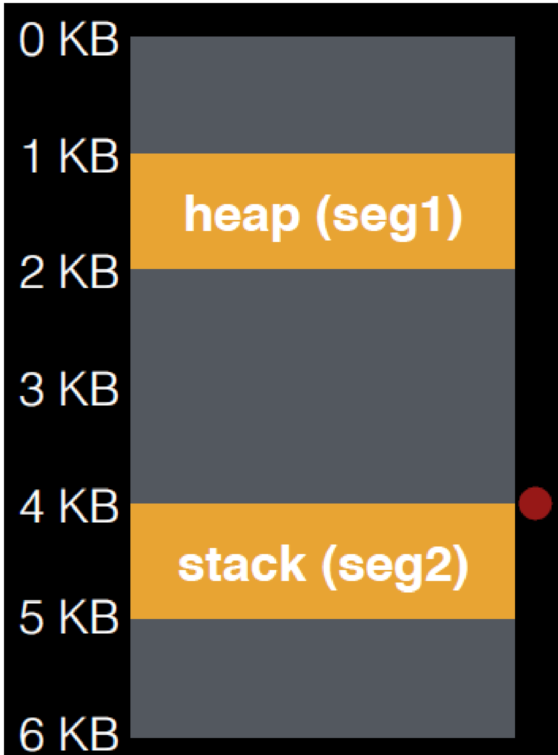


Virtual	Physical
load 0x2010, R1	4KB + 16
load 0x1010, R1	1KB + 16
load 0x1100, R1	1KB + 256
load 0x1400, R1	**interrupt OS!**

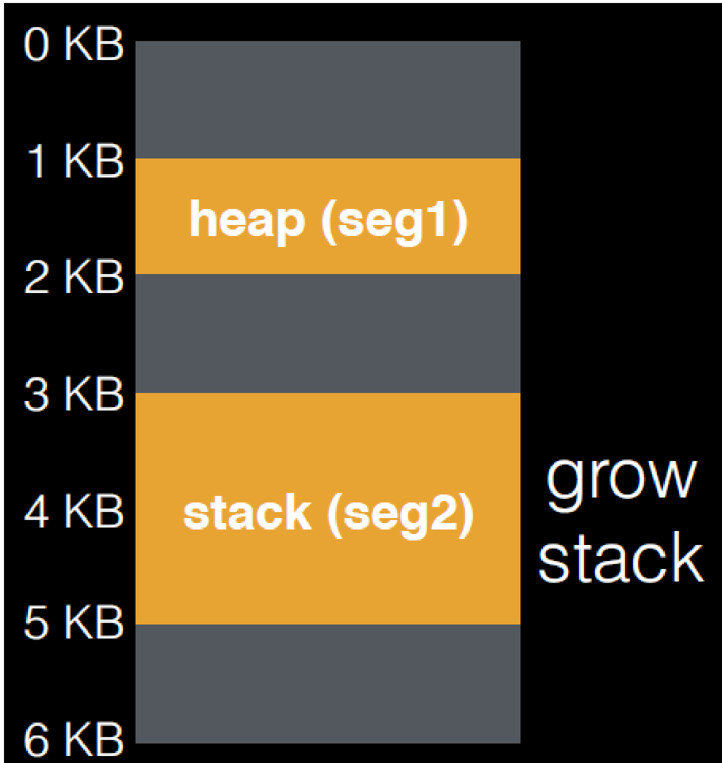
Stack Growth Problem

Example…

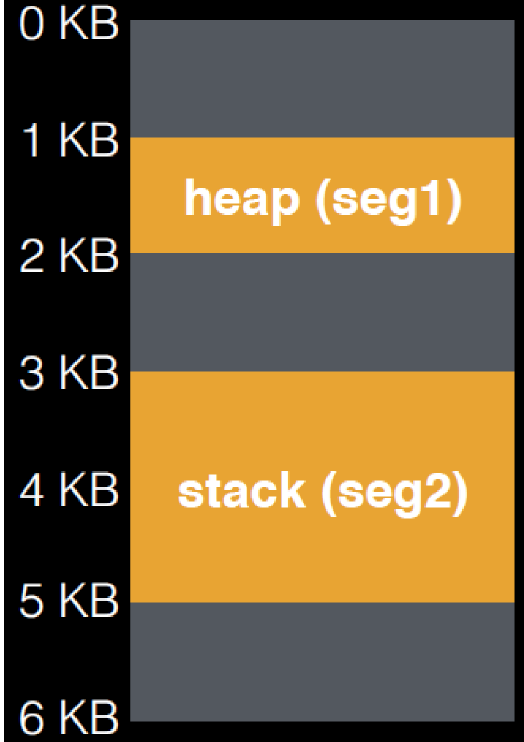
Virtual	Physical
load 0x2000, R1	



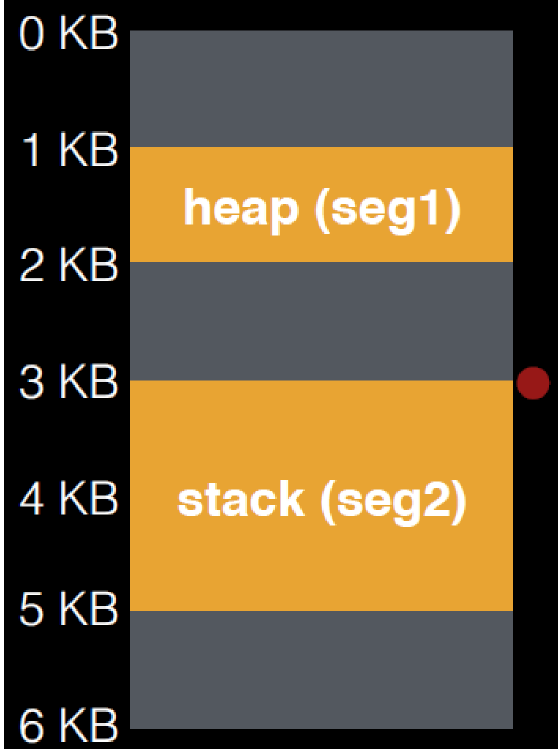
Virtual	Physical
load 0x2000, R1	4KB



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

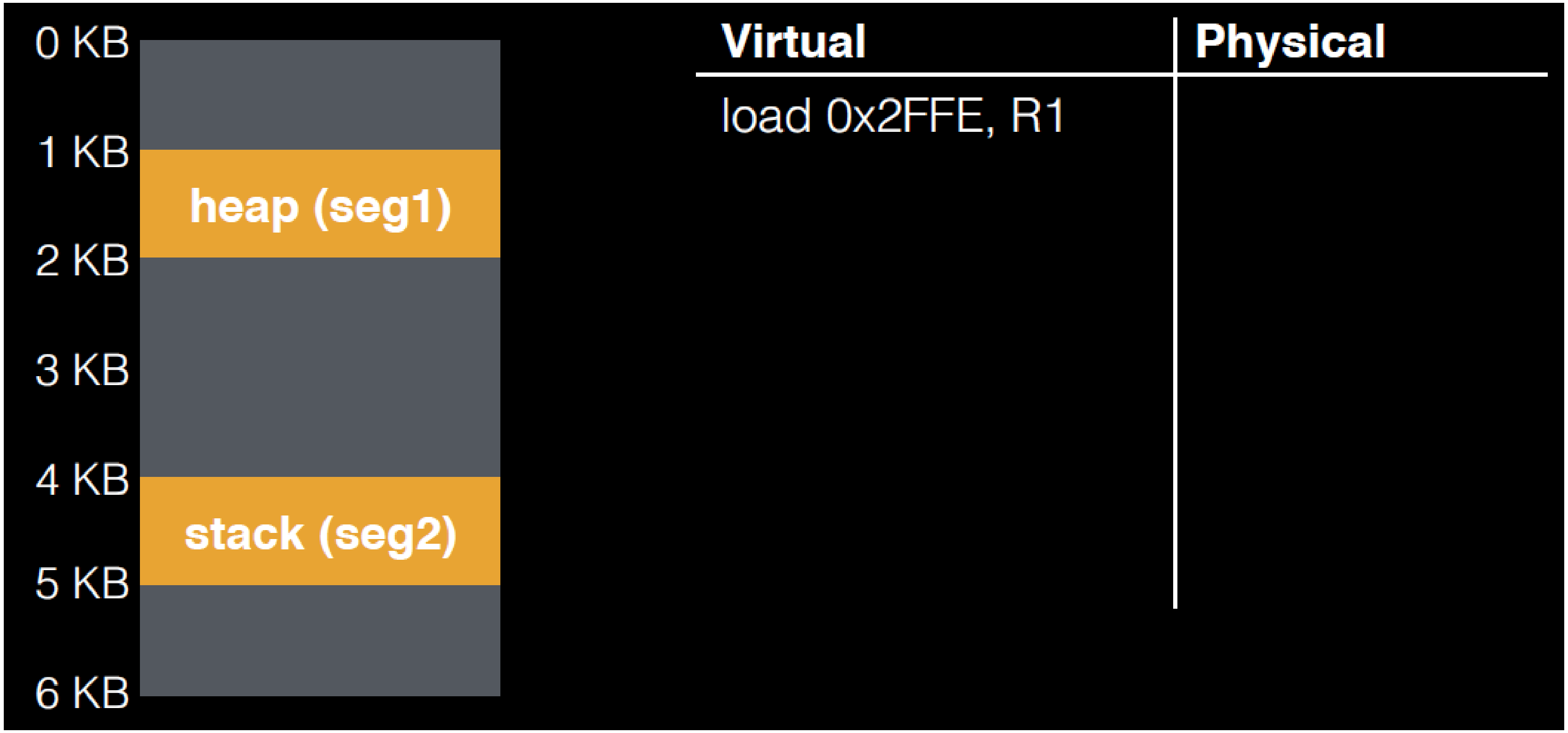
phys is anchored to base_reg, which moves

Solution: anchor heap segment to bounds_reg:

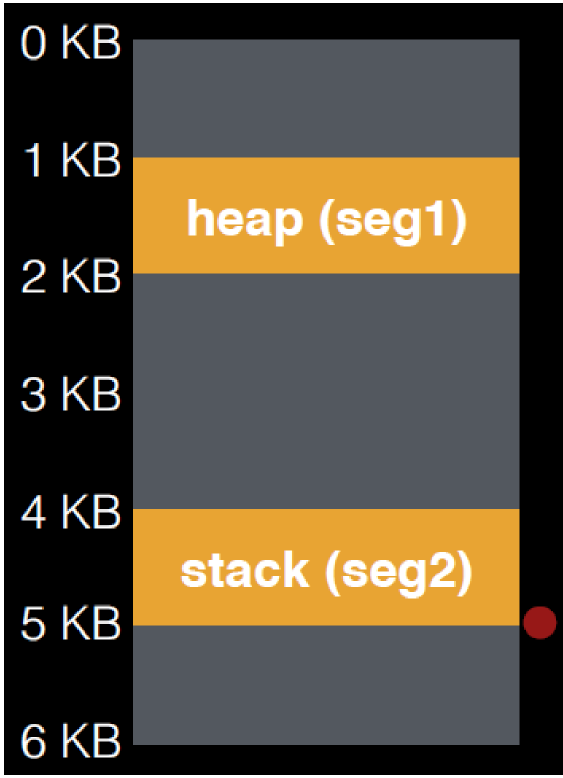
phys = bounds_reg - (max_offset - virt_offset)

Example (with max_offset = FFF)…

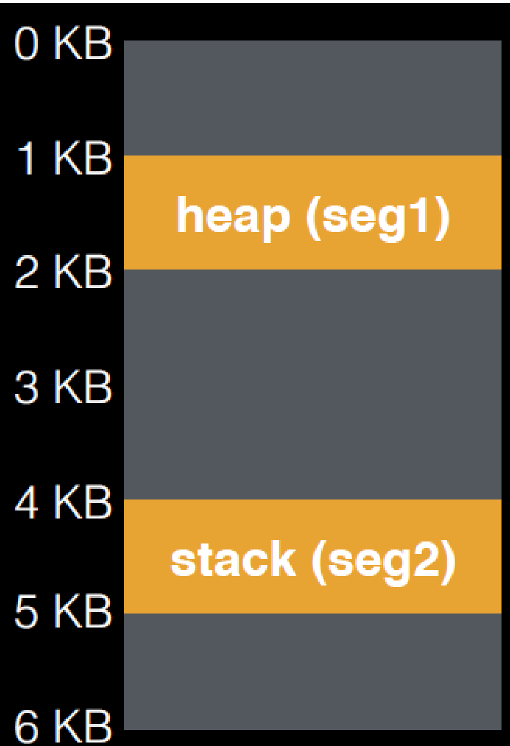
stack's max_offset = FFF



stack's max_offset = FFF



Virtual	Physical
load 0x2FFE, R1	5KB - 1

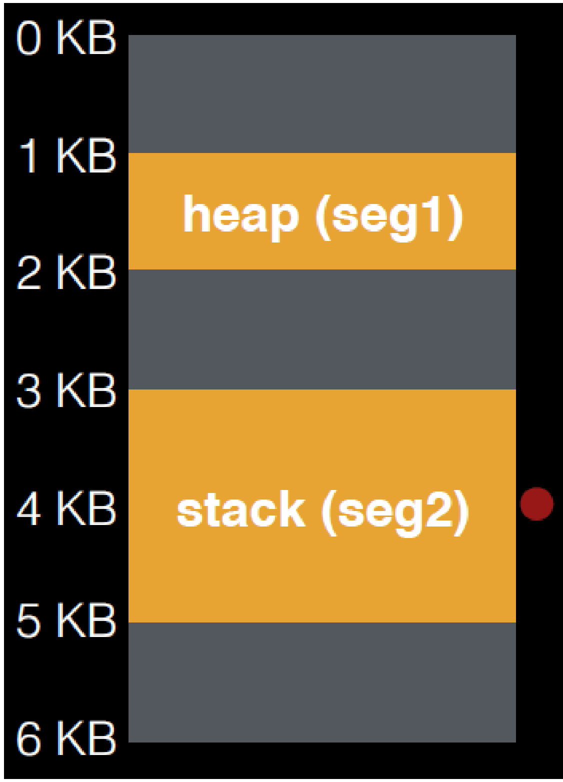


Virtual	Physical
load 0x2FFE, R1	5KB - 1
load 0x2BFF, R1	

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

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

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



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)

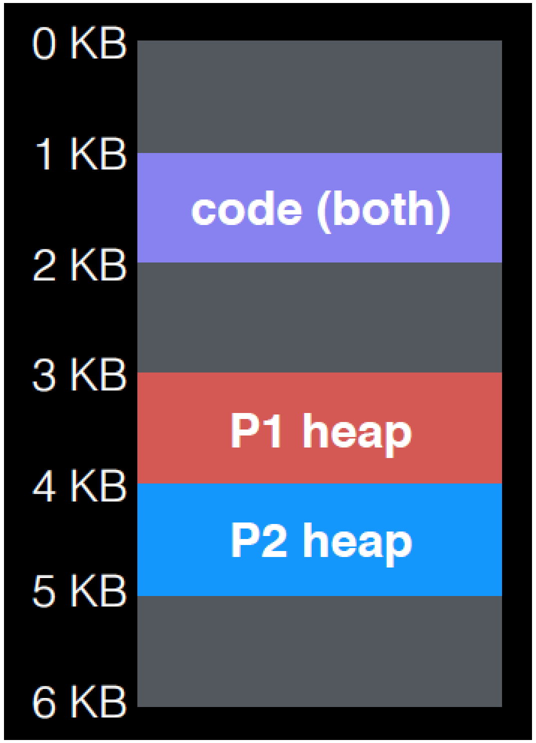
Anchors:

- 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 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