Welcome back!

MEMORY VIRTUALIZATION

Shivaram Venkataraman

CS 537, Spring 2023

ADMINISTRIVIA

- Project 2 is due Wednesday →
- Project 1 grading in progress

- Midterm 1: in-class exam → March 2nd

THE DATA BUDDIES SURVEY

- Longitudinal
- Computer science departments nationwide
- Measures students' sense of belonging, community, pre-college preparation, and satisfaction with program

FEEDBACK LEADS TO CHANGE

- More emphasis on encourage student study groups
- More TA/Peer Mentor support in classes
- Increased community-building efforts

Complete the survey by February 17th

WIN!

One of TEN Amazon gift cards!

AGENDA / LEARNING OUTCOMES

- Memory virtualization
  - What are main techniques to virtualize memory?
  - What are their benefits and shortcomings?

RECAP

MEMORY VIRTUALIZATION

Transparency: Process is unaware of sharing

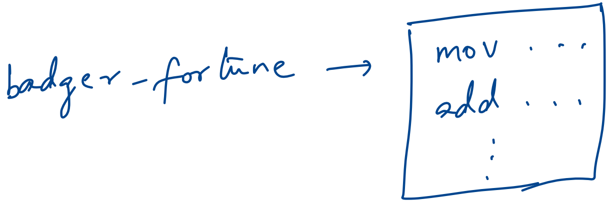
Protection: Cannot corrupt OS or other process memory

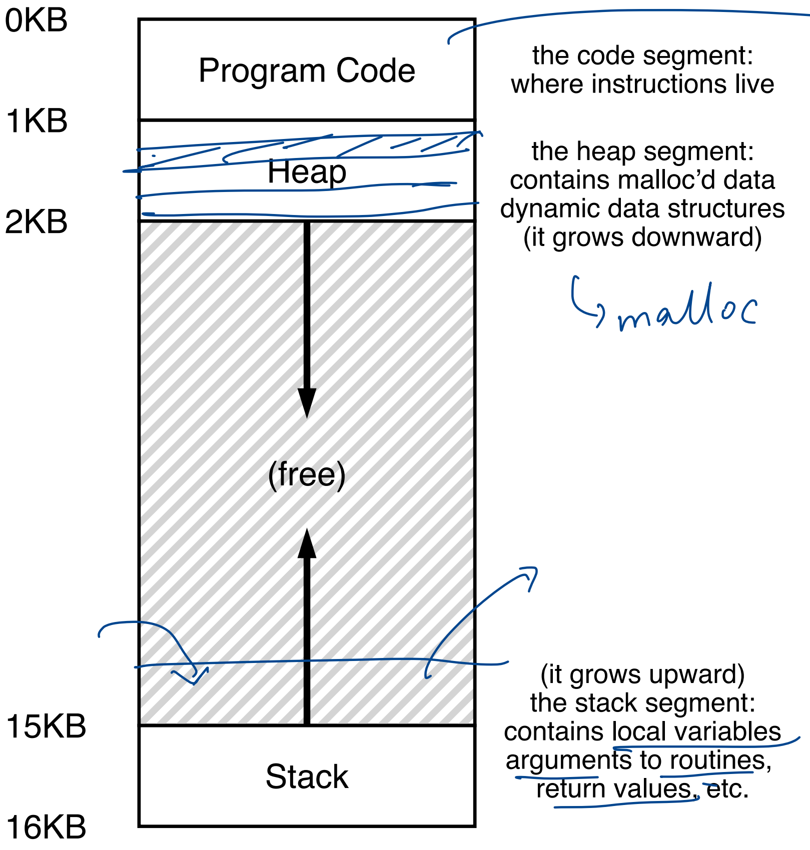
Efficiency: Do not waste memory or slow down processes

→ fragmentation

Sharing: Enable sharing between cooperating processes

RECAP: WHAT IS IN ADDRESS SPACE?





Static: Code and some global variables

Dynamic: Stack and Heap

MEMORY ACCESS

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

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

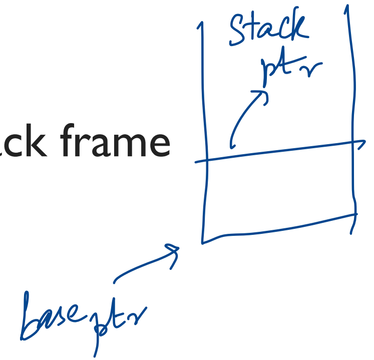
Instructions ≡ Code region

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

Memory accesses

- Get instruction at 0x10 (Code)
- Get data from %rbp + 0x8 (stack)
- Get instruction at 0x13 (code)
- Store data at %rbp + 0x8 (stack)

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



MEMORY ACCESS

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 ACCESS

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

QUIZ 6

https://tinyurl.com/cs537-sp23-quiz6

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

Possible locations:
static data/code, stack, heap

Address	Location
x	static data / code
main	Code
y	stack
z	stack
*z	heap

MEMORY VIRTUALIZATION: MECHANISMS

HOW TO VIRTUALIZE MEMORY

Problem: How to run multiple processes simultaneously?

Addresses are "hardcoded" into process binaries → Transparency

How to avoid collisions? ←

Process 1

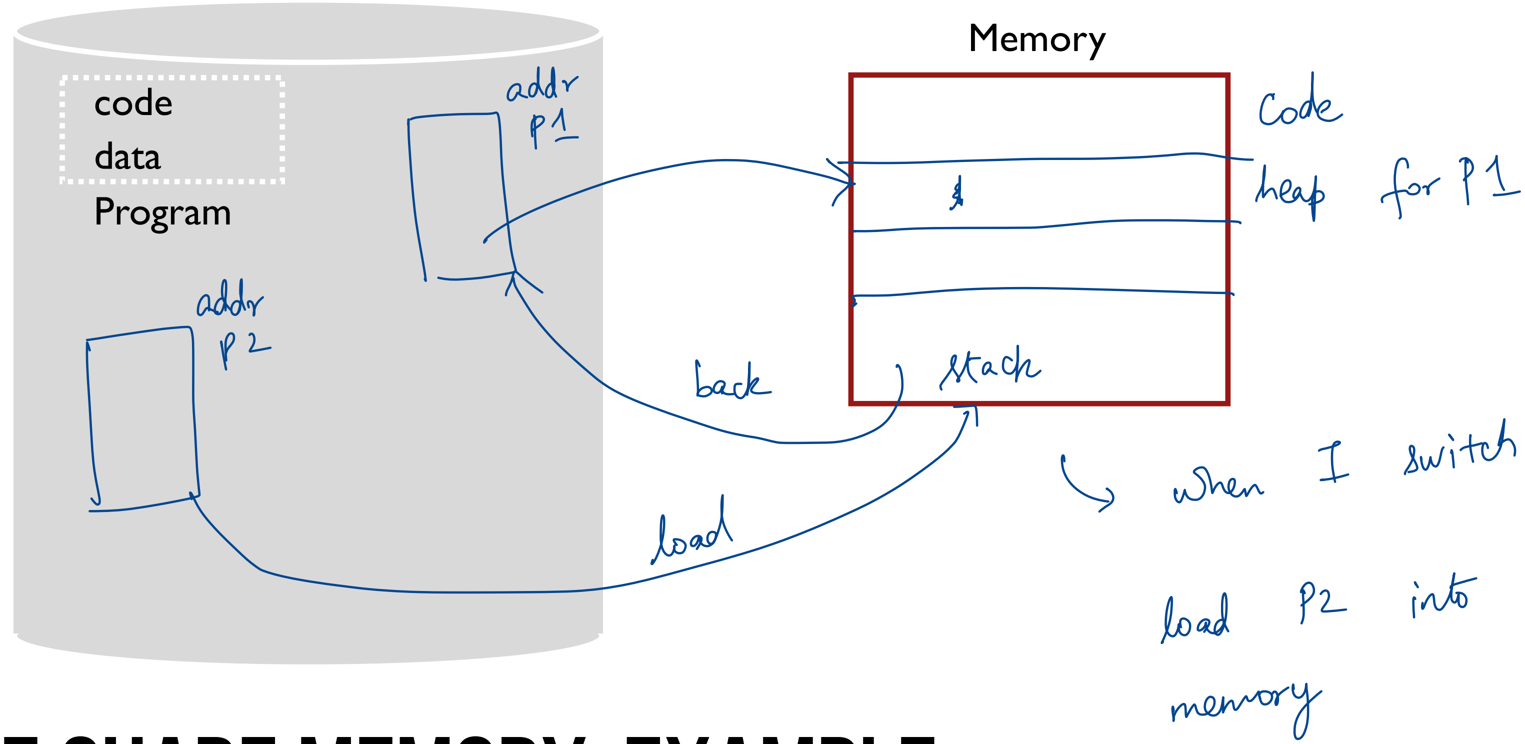
int x; 0x10

Process 2

int x; 0x10

Possible Solutions for Mechanisms (covered in this class):

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



TIME SHARE MEMORY: EXAMPLE

↳ CPU virtualization

PROBLEMS WITH TIME SHARING?

Ridiculously poor performance → slow to switch addr spaces → might waste memory

Better Alternative: space sharing!

At same time, space of memory is divided across processes

Remainder of solutions all use space sharing

2) STATIC RELOCATION

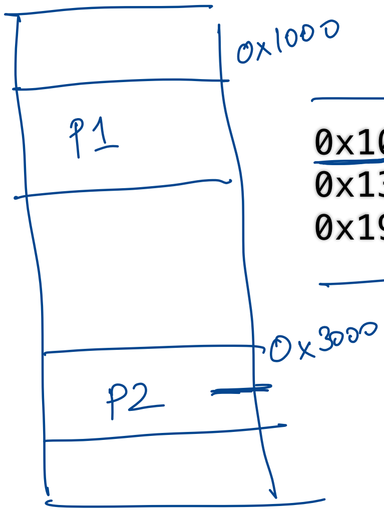
(2) Rewrite not easy to do?

(1) Violates protection

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




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

rewrite (add 0x1000)

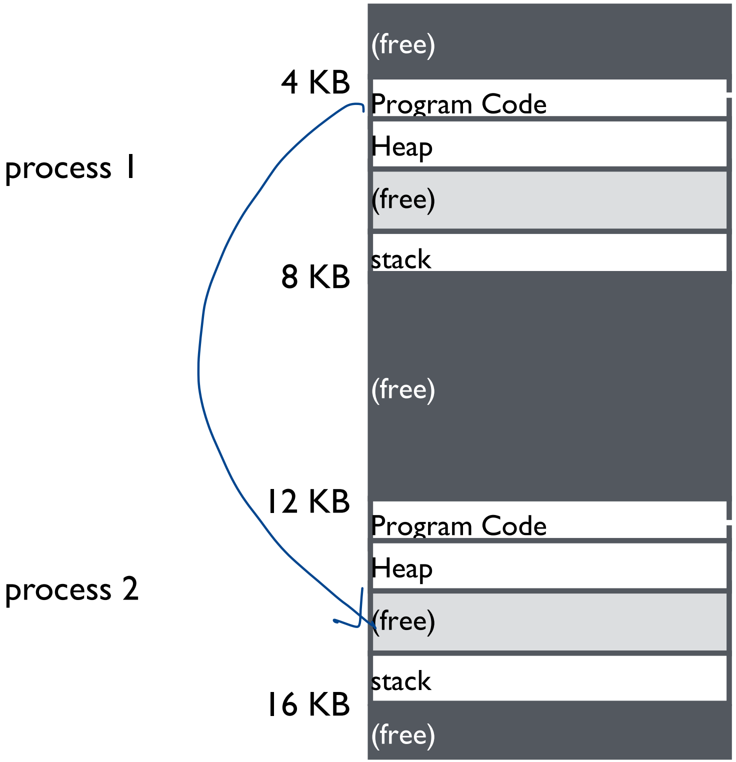
```
0x1010: movl  0x8(%rbp), %edi
0x1013: addl  $0x3, %edi
0x1019: movl  %edi, 0x8(%rbp)
```

movl 0x3042 %edi

rewrite (add 0x3000)

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

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

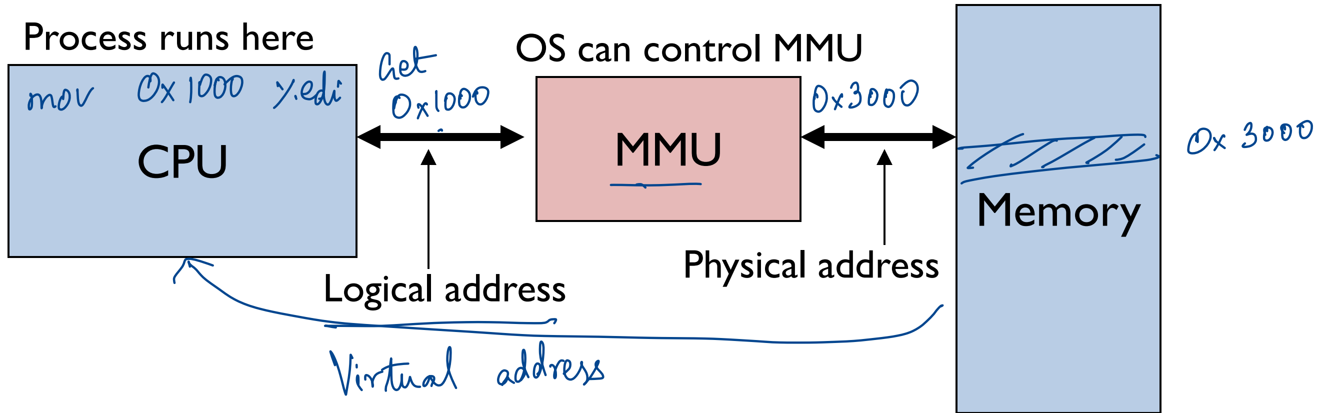
Goal: Protect processes from one another

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

Privileged (protected, kernel) mode: OS runs

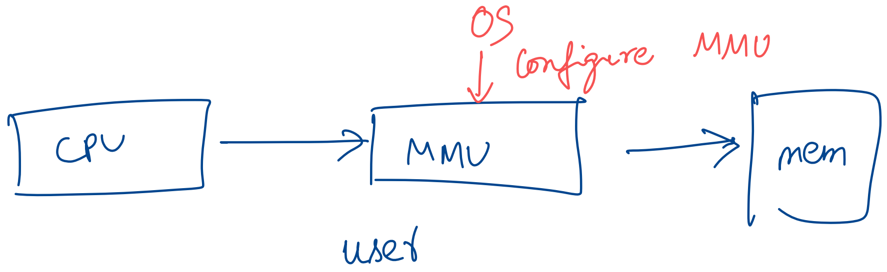
timer interrupt

- 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

new x86 instructions

User mode: User processes run

- Perform translation of logical address to physical address



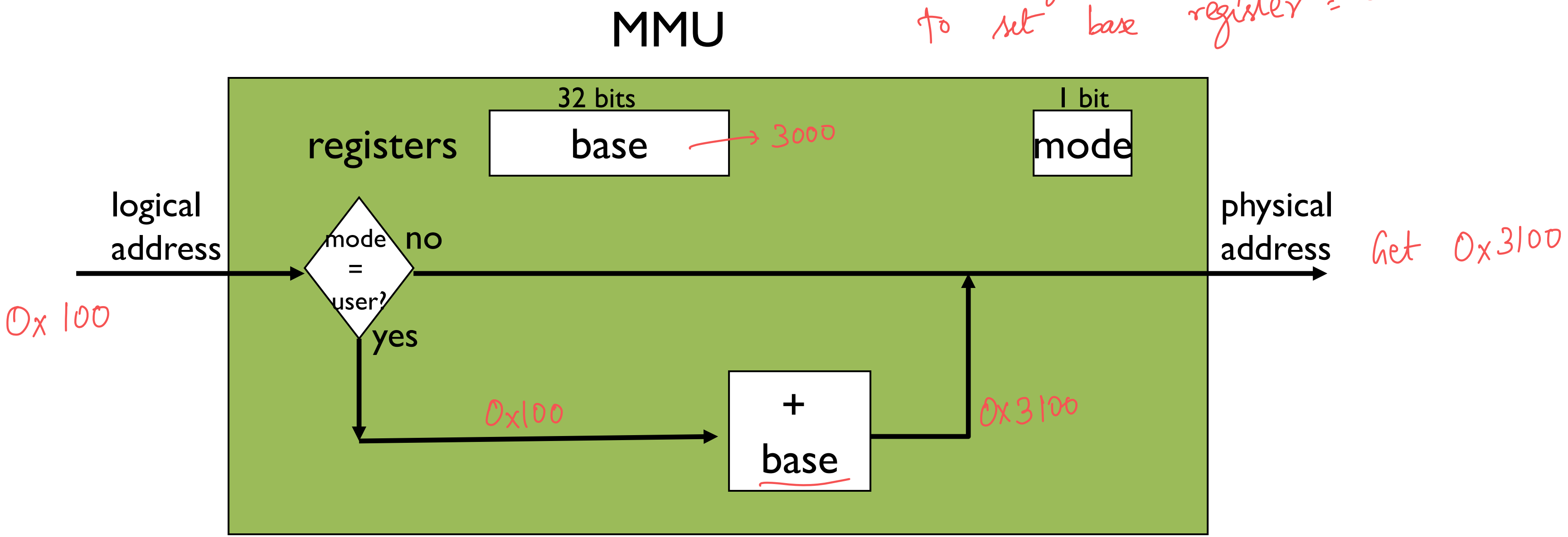
IMPLEMENTATION OF DYNAMIC RELOCATION: BASE REG

Choose P2, OS updates base register

Translation on every memory access of user process

MMU adds base register to logical address to form physical address

OS configures MMU to set base register = 0x 3000



DYNAMIC RELOCATION WITH BASE REGISTER

Translate virtual addresses to physical by adding a fixed offset each time.

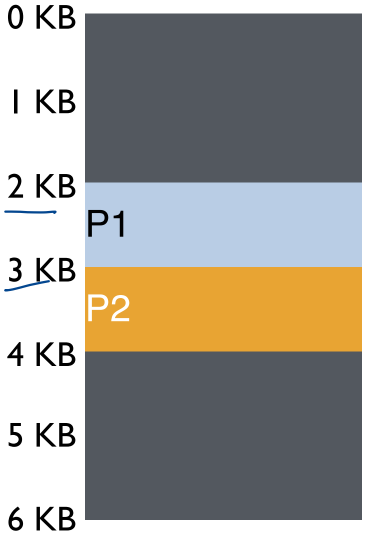
Store offset in base register

Each process has different value in base register

Dynamic relocation by changing value of base register!

- Dynamic relocation makes it possible to move processes at runtime
- Still lack protection: Clever process could still read another process memory

VISUAL EXAMPLE OF DYNAMIC RELOCATION: BASE REGISTER



Base Register for P1 = 2048

Base Register for P2 = 3072

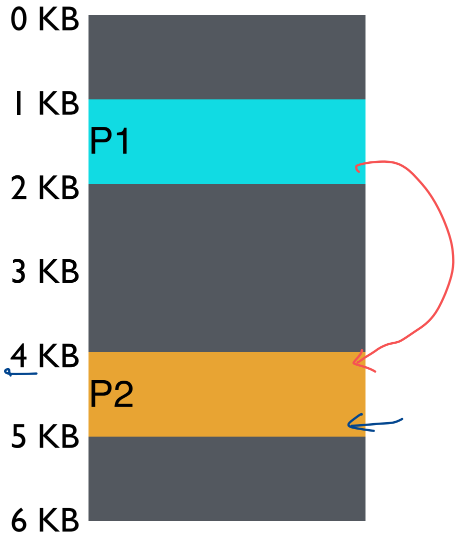
Virtual	Physical
P1: load 10, R1	2048+10 = 2058
P1: load 200, R1	2048+200 = 2248
P2: load 500, R1	3072+500 = 3572

virtual address

Context switch

QUIZ 7

https://tinyurl.com/~~quiz7-sp23~~ cs537-sp23-quiz7



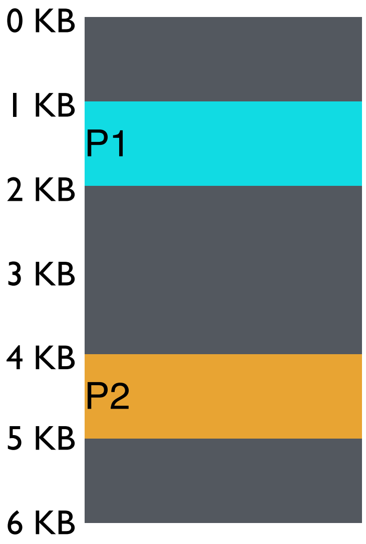
Virtual

P1: load 100, R1 — 1024 + 100 = 1124

P2: load 1000, R1 — 4096 + 1000 = 5096

P1: store 3072, R1 — 1024 + 3072 = 4096

violating protection



Virtual	Physical	
P1: load 100, R1	load 1124, R1	
P2: load 1000, R1	load 5096, R1	
P1: store 3072, R1	store 4096, R1	(3072 + 1024)

4) DYNAMIC WITH BASE+BOUNDS

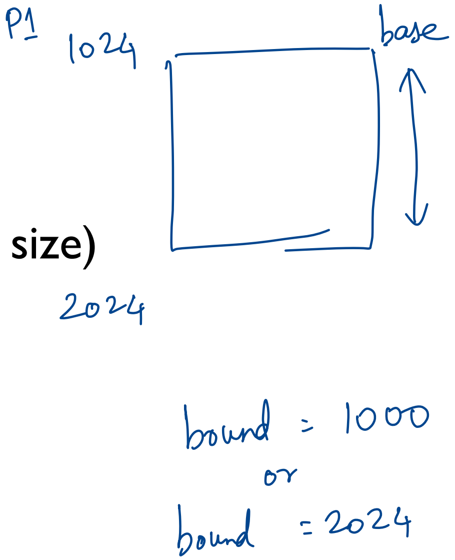
Idea: limit the address space with a bounds register

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)

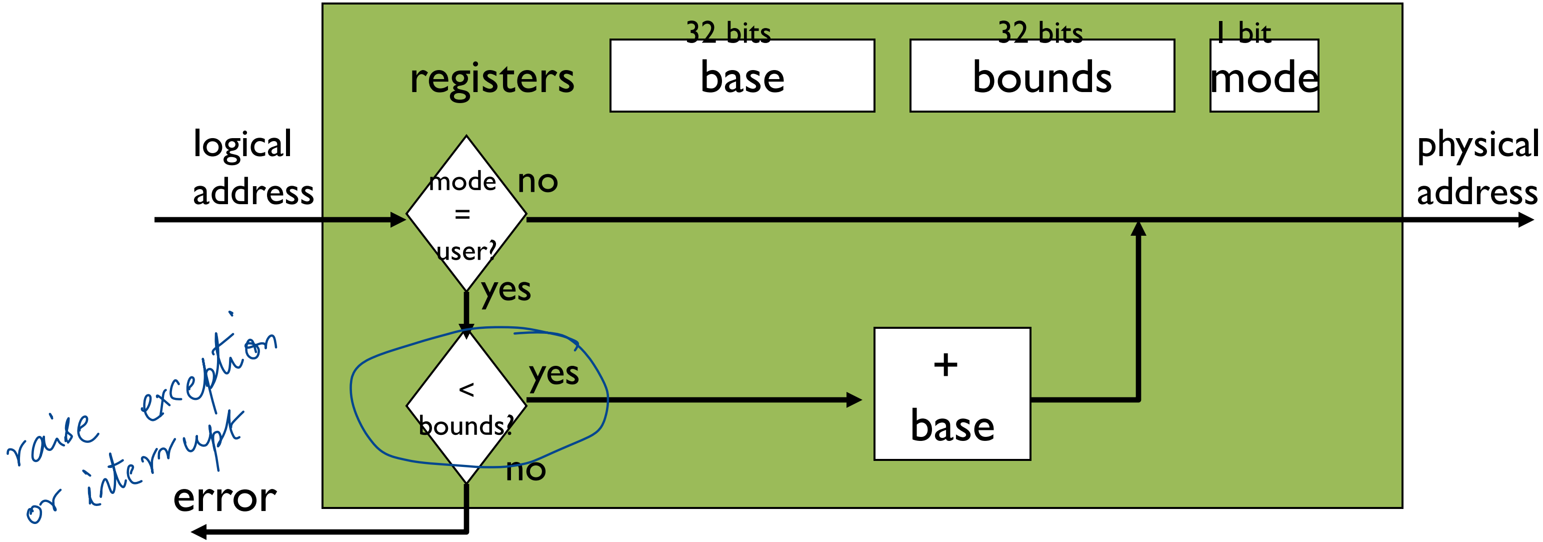
OS kills process if process loads/stores beyond bounds

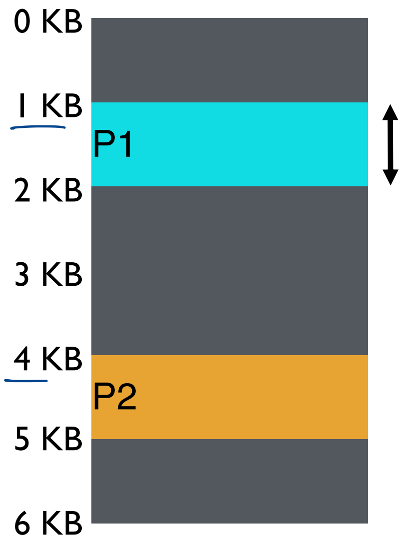


IMPLEMENTATION OF BASE+BOUNDS

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



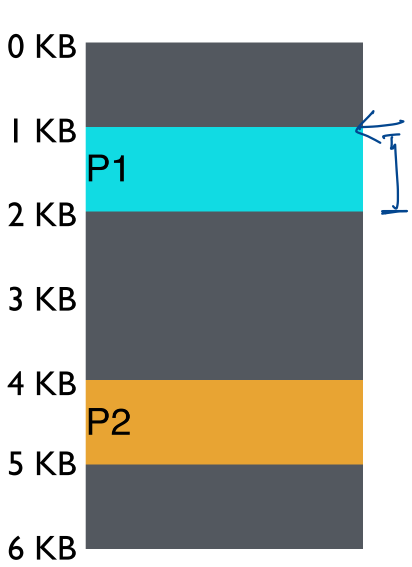


base register

bounds register

each process could have different addr space size

Base + bounds does not enable sharing



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?

MMU will check

if 1024 + 3072 < 2048

Not true

raise an error to the OS

MANAGING PROCESSES WITH BASE AND BOUNDS

Context-switch: Add base and bounds registers to proc struct

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

Protection requirement

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

Assumptions that yield protection

BASE AND BOUNDS

Advantages

- Provides protection (both read and write) across address spaces
- Supports dynamic relocation
  - Can place process at different locations initially and move address spaces

- Simple, inexpensive implementation: Few registers, little logic in MMU

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 parts of address space

5) SEGMENTATION

Divide address space into logical segments

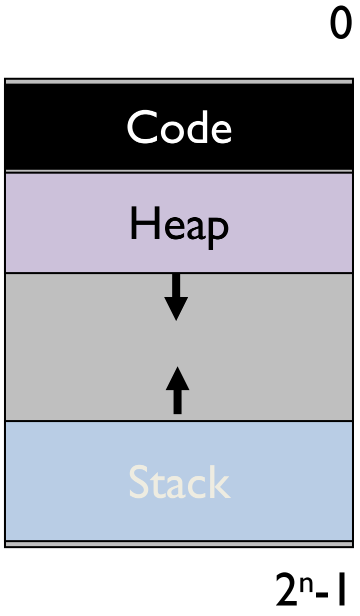
- Each segment corresponds to logical entity in address space (code, stack, heap)

Each segment has separate base + bounds register

Code base, bounds register

Heap base, bounds

Stack base, bounds



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

What if small address space, not enough bits?

- Implicitly by type of memory reference
- Special registers

Process
↳ segment
↳ address within segment

01 10 1000

8-bit address

2-bits select 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?

2 bits segment

How many bits for offset?

12 bits 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

code heap stack

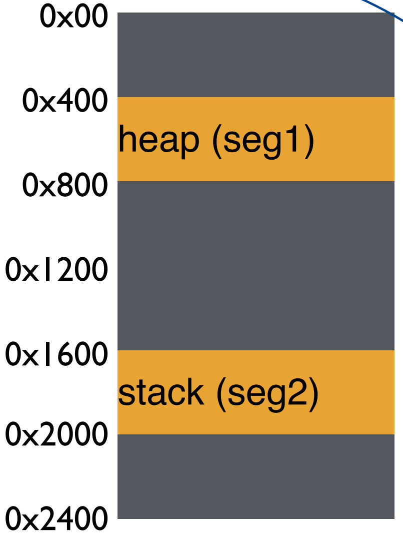
process can read but not write to this segment

remember:
1 hex digit → 4 bits

VISUAL INTERPRETATION

each number is 4 bits

14 bit addr



Virtual (hex)	Physical
load 0x2010, R1	0x1600 + 0x 010 = 0x1610
load 0x1010, R1	
load 0x1100, R1	

segment no: 2
offset : 0x010

01 0000 0001 0000 | 0x400 + 0x010 = 0x410

Segment numbers:

0: code+data

1: heap

2: stack

base register

0x400

0x1600

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

QUIZ 8!

https://tinyurl.com/cs537-sp23-quiz8

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 (in hex) to physical

0x0240:

0x1108:

0x265c:

0x3002:

HOW DOES THIS LOOK IN X86

Stack Segment (SS): Pointer to the stack

Code Segment (CS): Pointer to the code

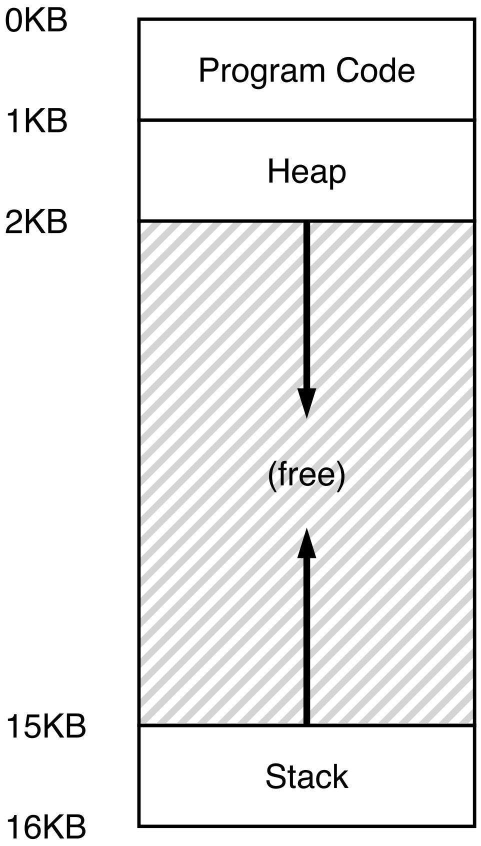
Data Segment (DS): Pointer to the data

Extra Segment (ES): Pointer to extra data

F Segment (FS): Pointer to more extra data

G Segment (GS): Pointer to still more extra data

NOTE: HOW DO STACKS GROW ?



Stack goes 16K → 12K, in physical memory is 28K → 24K

Segment base is at 28K

Virtual address 0x3C00 = 15K

→ top 2 bits (0x3) segment ref, offset is 0xC00 = 3K

How do we make CPU translate that ?

Negative offset = subtract max segment from offset

= 3K – 4K = -1K

Add to base = 28K – 1K = 27K

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

- Enables sharing of selected segments
- Read-only status for code

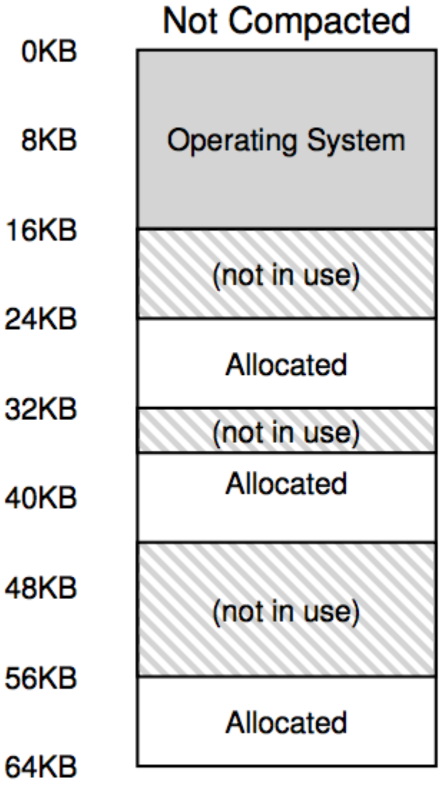
Supports dynamic relocation of each segment

DISADVANTAGES OF SEGMENTATION

Each segment must be allocated contiguously

May not have sufficient physical memory for large segments?

External Fragmentation



NEXT STEPS

Project 2: Due Wednesday!

Next class: Paging, TLBs and more!