VIRTUALIZATION: CPU TO MEMORY

Shivaram Venkataraman

CS 537, Spring 2019

ADMINISTRIVIA

- Project 1a is due today
- Extra office hours from 7pm to 9pm?

- Project 1b is out, due Feb 8th (1day shorter)
- Discussion section: xv6 code walk through!
- Schedule updates

AGENDA / LEARNING OUTCOMES

CPU virtualization

- Recap of scheduling policies
- Work through problems

Memory virtualization

- What is the need for memory virtualization?
- How to virtualize memory?

RECAP: CPU VIRTUALIZATION

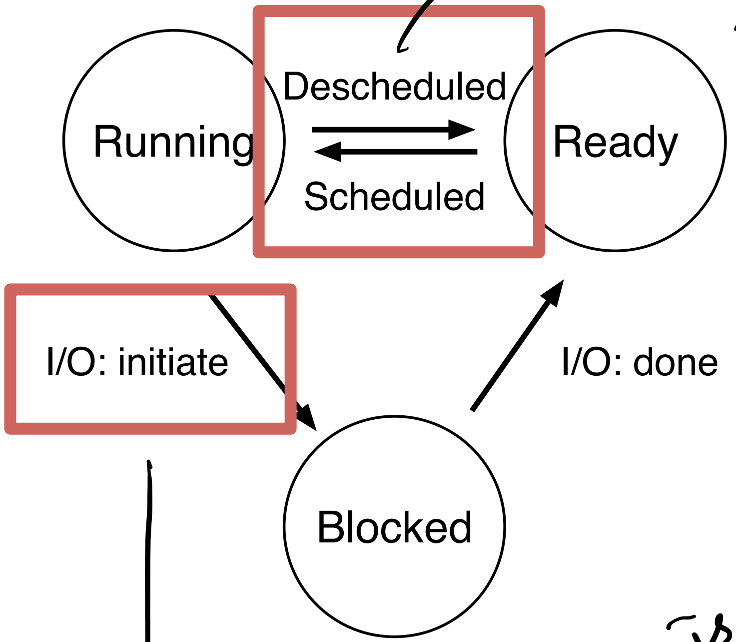
RECAP: SCHEDULING MECHANISM

Process: Abstraction to virtualize CPU

Use time-sharing in OS to switch between processes

Limited Direct Execution

- Use system calls to run access devices etc. from user mode
- Context-switch using interrupts for multi-tasking



context switch = How is this process selected

POLICY

How is I/O handled by scheduler

METRICS → POLICIES

Turnaround time = *completion_time - arrival_time*

FIFO: First come, first served

SJF: Shortest job first

SCTF: Shortest completion time first

METRICS → POLICIES

Response time = *first_run_time* - *arrival_time*

RR: Round robin with time slice

Minimizes response time but could increase turnaround?

QUIZ!

≥ ./scheduler.py -p RR -j 3 -s 121

Here is the job list, with the run time of each job:

Job 0 (length = 1)

Job 1 (length = 6)

Job 2 (length = 4)

Compute response time, turn around time for RR, SJF and FIFO

Policy	Avg TA	Avg Resp
FIFO	6.3	2.86
SJF	17/3	2
RR	21/3 = 7	1

RR

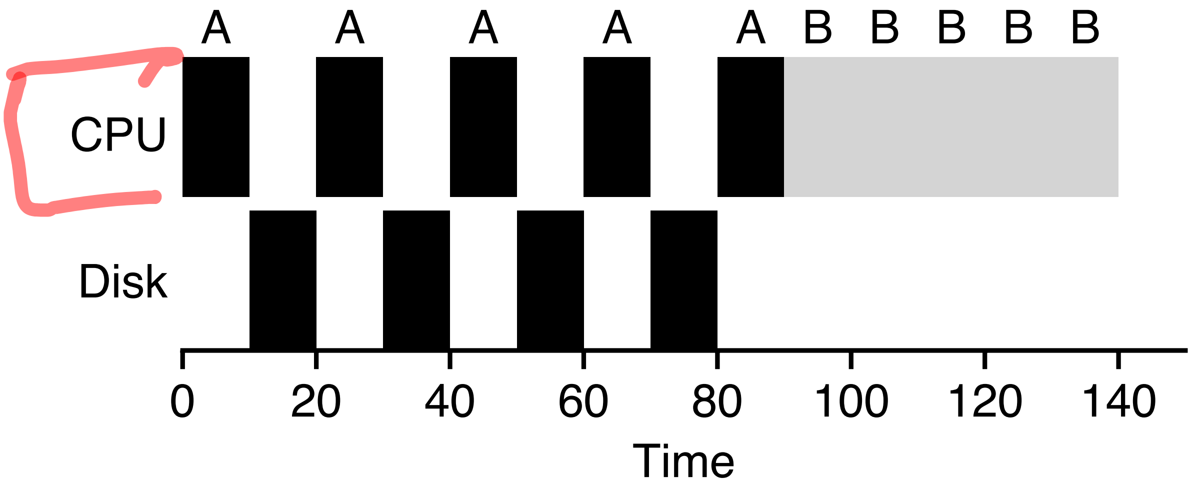
0 1 2 1 2 . . . . 1 1 1

$0 + 1 + 2 = 3/3 = 1$

ASSUMPTIONS

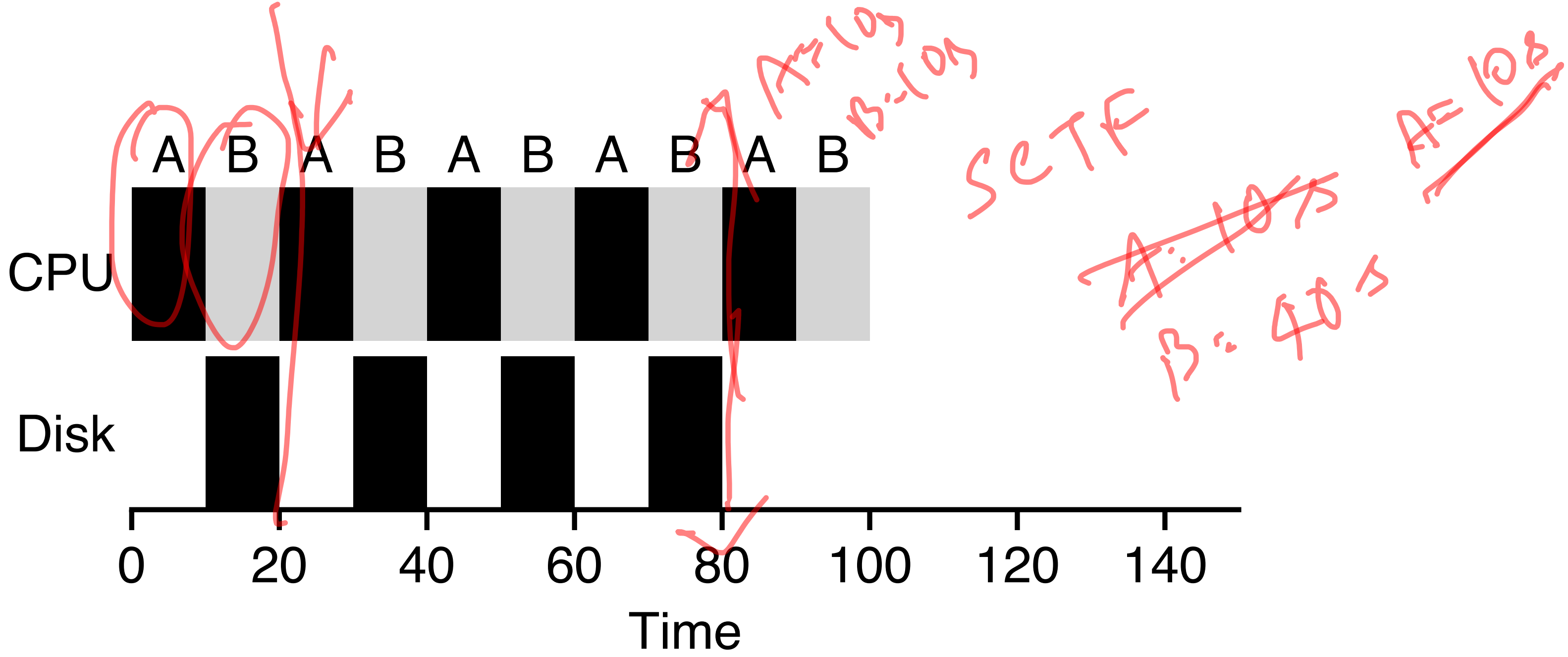
1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

NOT IO AWARE



Job holds on to CPU while blocked on disk!

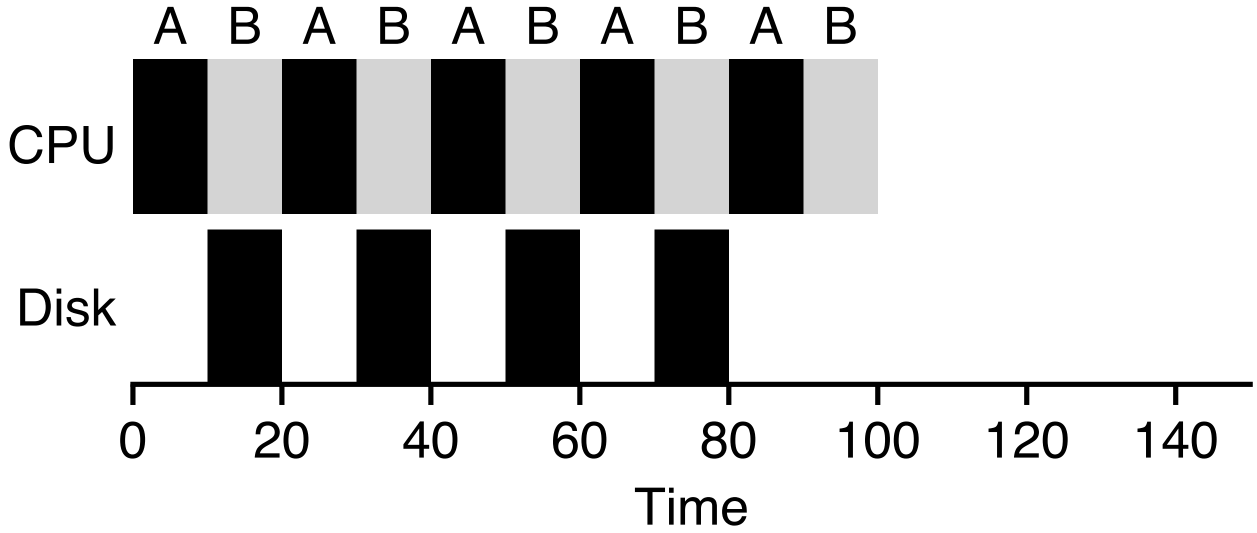
I/O AWARE SCHEDULING



Treat Job A as 3 separate CPU bursts.

When Job A completes I/O, another Job A is ready

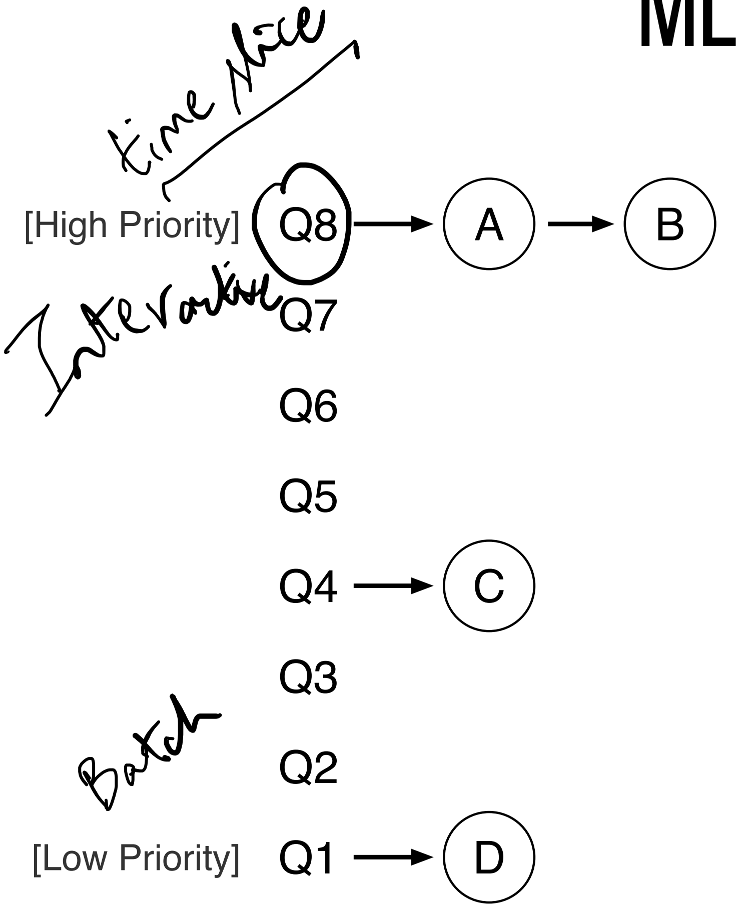
I/O AWARE SCHEDULING



Treat Job A as 3 separate CPU bursts.
When Job A completes I/O, another Job A is ready

MULTI-LEVEL FEEDBACK QUEUE

MLFQ EXAMPLE



Rules for MLFQ

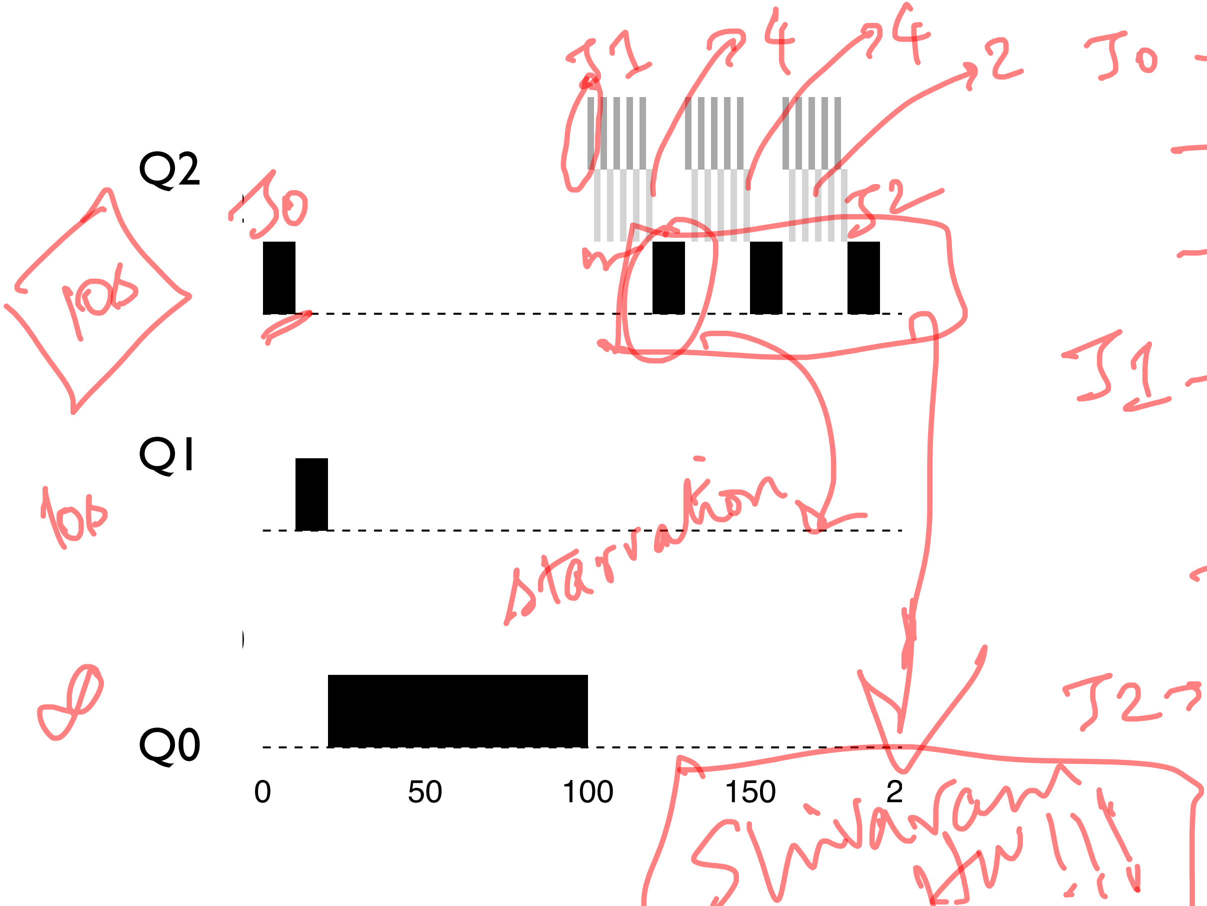
Rule 1: If priority(A) > Priority(B)
A runs

Rule 2: If priority(A) == Priority(B),
A & B run in RR

Rule 3: Processes start at top priority

Rule 4: If job uses whole slice, demote process.
If not stay at level

MLFQ WALKTHROUGH



J0 → high priority Q2
→ demoted Q1
→ demoted Q0

J1 → enter high priority
→ 1s

J2 → enter high priority
RR

HOMEWORK

This program, mlfq.py, allows you to see how the MLFQ scheduler presented in this chapter behaves. As before, you can use this to generate problems for yourself using random seeds, or use it to construct a carefully-designed experiment to see how MLFQ works under different circumstances. To run the program, type:

```
prompt> ./mlfq.py
```

Use the help flag (-h) to see the options:

http://pages.cs.wisc.edu/~remzi/OSTEP/Homework/homework.html

CPU SUMMARY

Mechanism

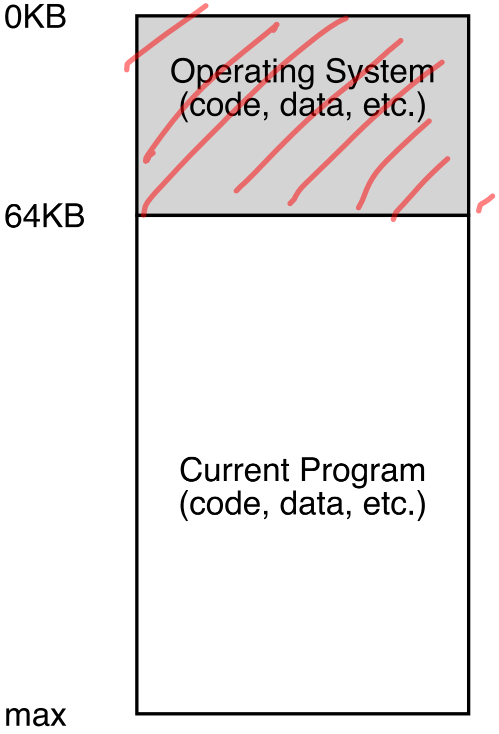
- Process abstraction
- System call for protection
- Context switch to time-share

Policy

- Metrics: turnaround time, response time
- Balance using MLFQ

VIRTUALIZING MEMORY

BACK IN THE DAY...



Uniprogramming: One process runs at a time

Disadvantages?

MULTIPROGRAMMING GOALS

Transparency: Process is unaware of sharing

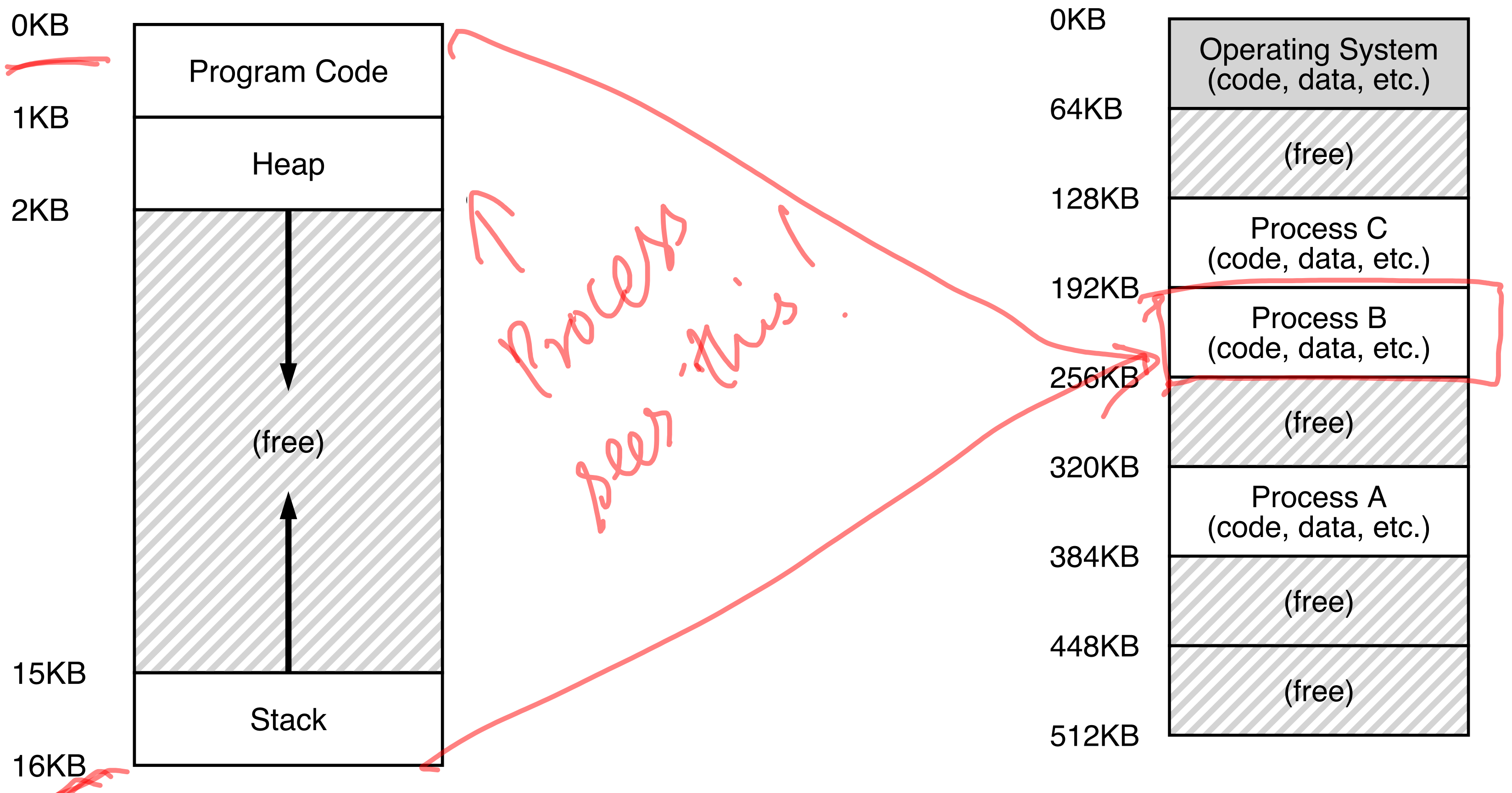
Protection: Cannot corrupt OS or other process memory

Efficiency: Do not waste memory or slow down processes

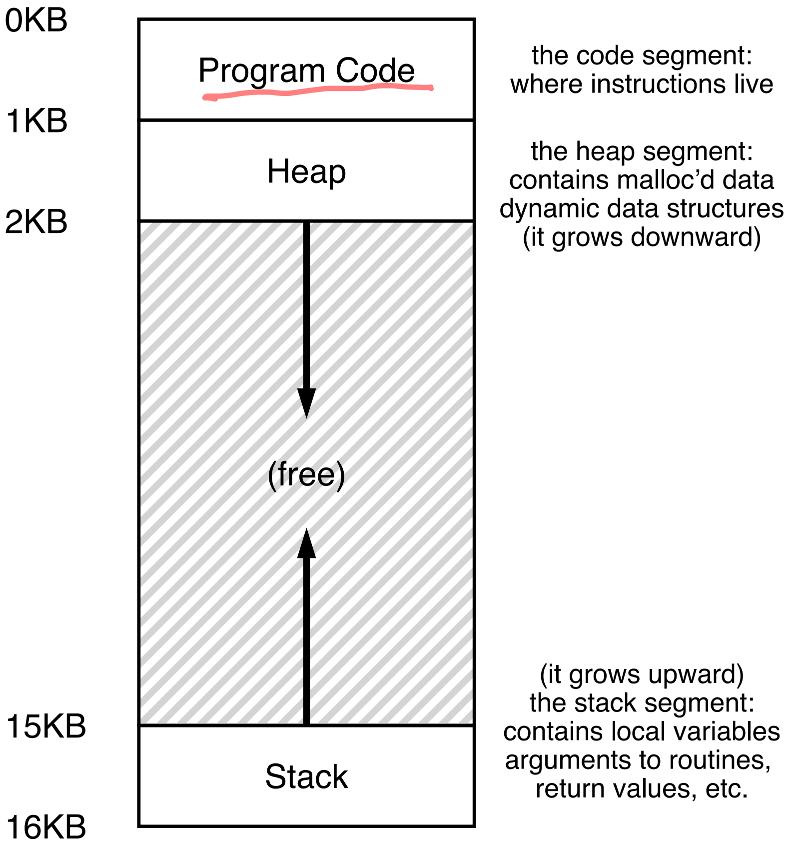
LPE

Sharing: Enable sharing between cooperating processes

ABSTRACTION: ADDRESS SPACE



WHAT IS IN ADDRESS SPACE?



instructions live here!

Static: Code and some global variables

Dynamic: Stack and Heap

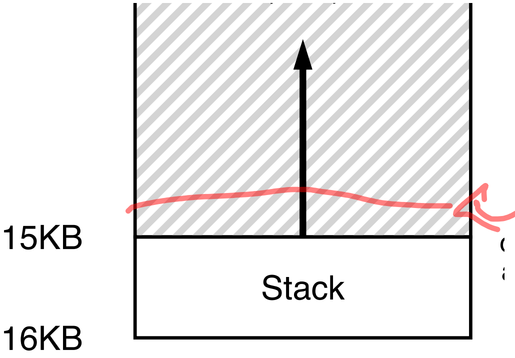
Last In First out

STACK ORGANIZATION

push

pop

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



Pointer between allocated and free space

- Allocate: Increment pointer
- Free: Decrement pointer

Stack pointer

No fragmentation!

WHAT GOES ON STACK?

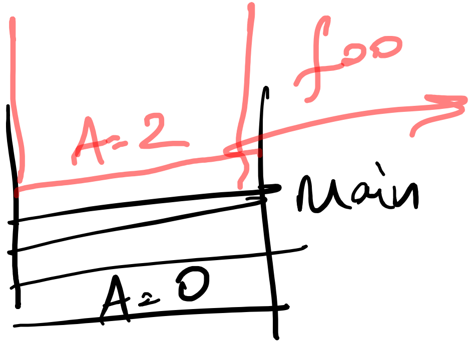
```
main () {
    int A = 0;
    foo(A);
    printf("A: %d\n", A);
}
void foo (int Z) {
    int A = 2;
    Z = 5;
    printf("A: %d Z: %d\n", A, Z);
}
```

local variables are on the stack.

parameters

return value.

stack frame



HEAP ORGANIZATION

C Java

Allocate from any random location: malloc(), new() etc.

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

Memory Allocators

fragmentation

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

QUIZ

global

pointer

int

malloc

code

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

int * z

Possible segments: static data, code, stack, heap

Address	Location
x	code
main	Code
y	stack
z	stack
*z	heap

MEMORY ACCESS

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

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

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

stack pointer

general purpose

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

MEMORY ACCESS

Initial %rip = 0x10 ← Instruction pointer
%rbp = 0x200 ← Stack/Frame pointer

```
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 0x10

Fetch 0x208

→ edi

Fetch 0x13

Fetch 0x19

Store 0x208

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

HOW TO VIRTUALIZE MEMORY

Problem: How to run multiple processes simultaneously?

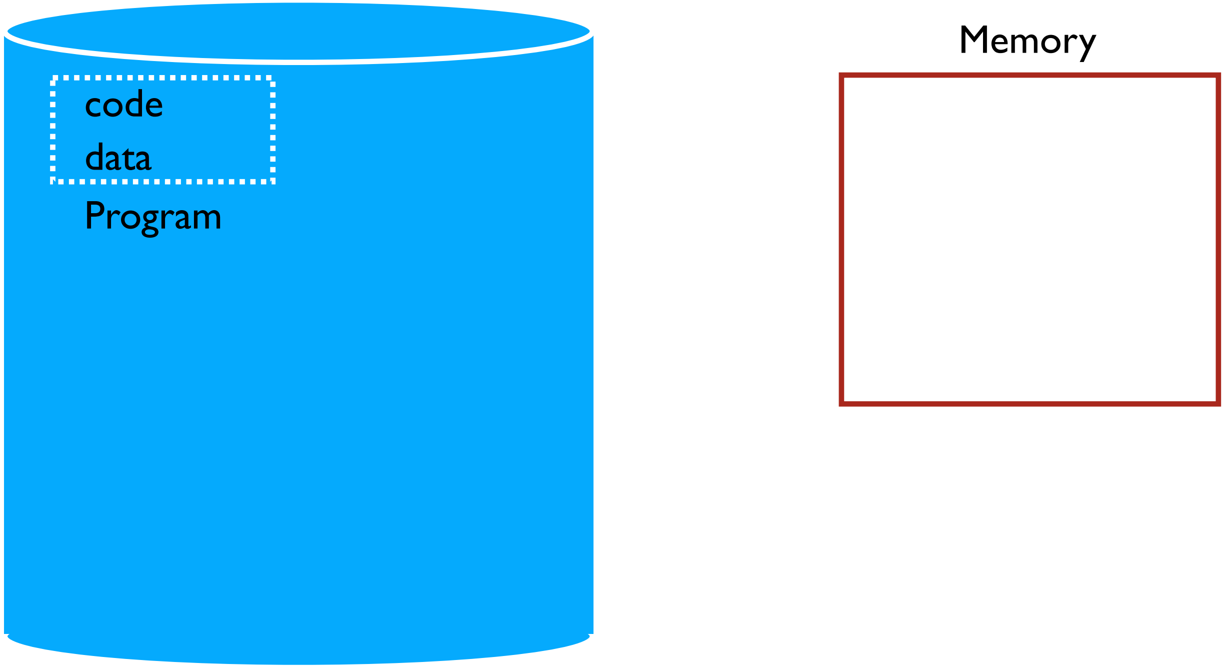
Addresses are "hardcoded" into process binaries

How to avoid collisions?

next class

Possible Solutions for Mechanisms (covered ~~today~~):

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



TIME SHARE MEMORY: EXAMPLE

PROBLEMS WITH TIME SHARING?

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

```
0x1010: movl  0x8(%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)`

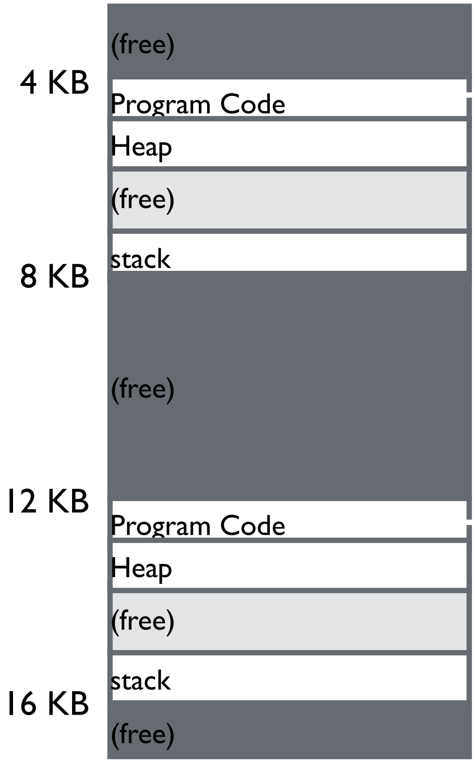
rewrite

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

STATIC: LAYOUT IN MEMORY

process 1

process 2




```
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)
```

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

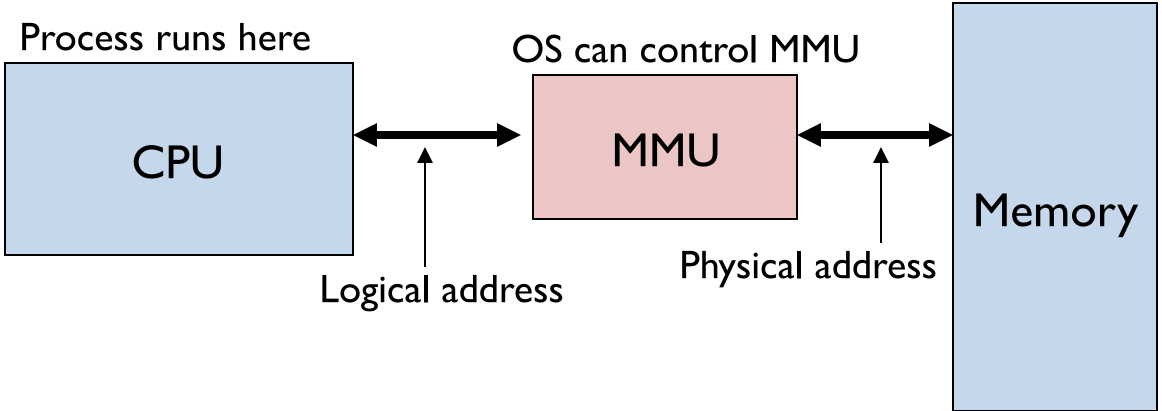
Goal: Protect processes from one another

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**

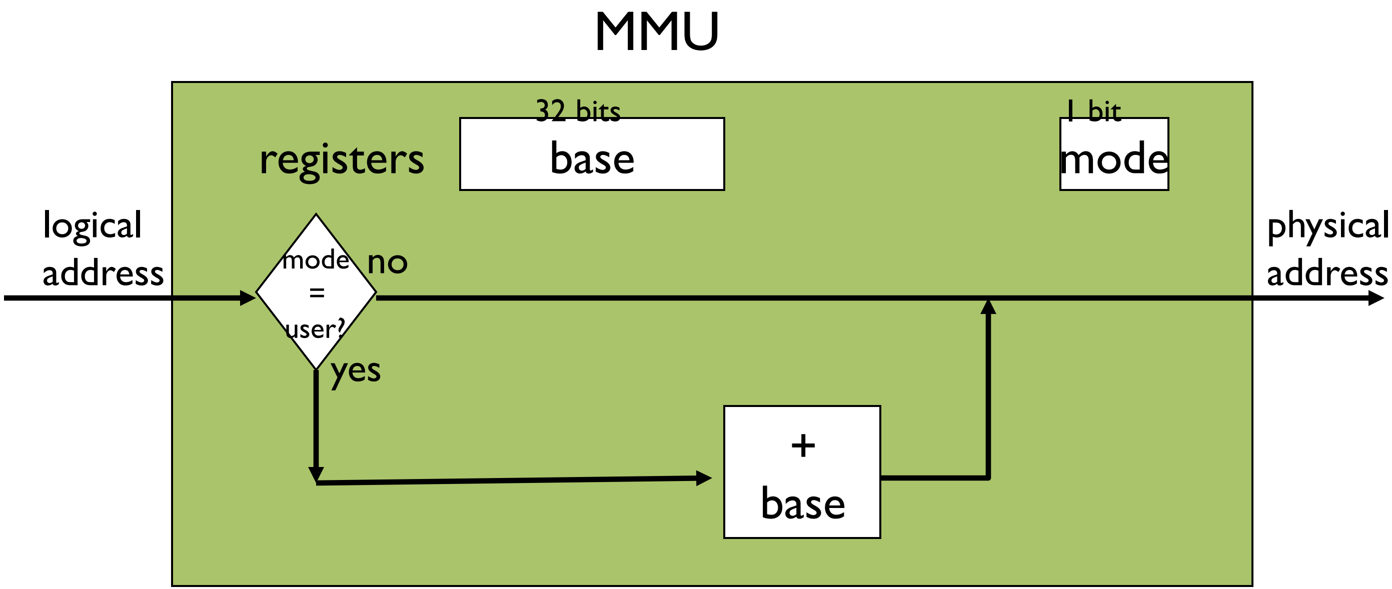
User mode: User processes run

- **Perform translation of logical address to physical address**

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



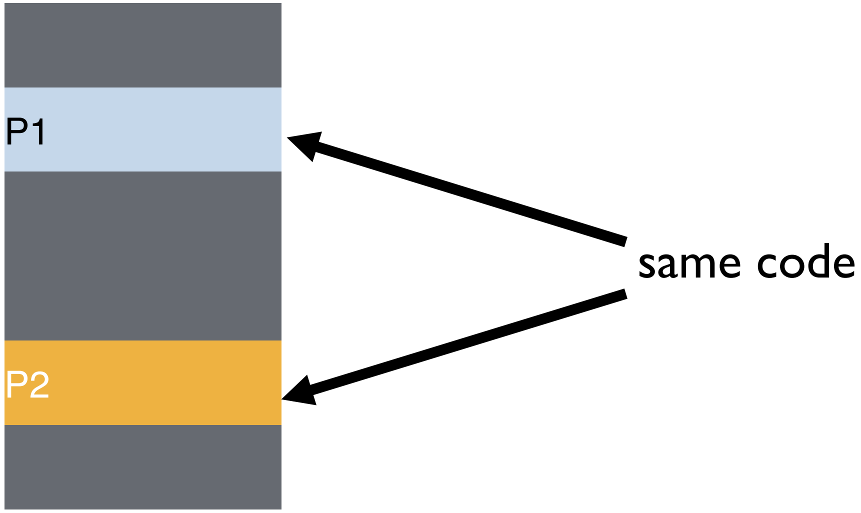
DYNAMIC RELOCATION WITH BASE REGISTER

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!



Virtual

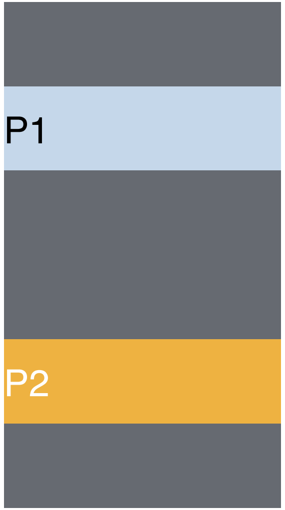
P1: load 100, R1

P2: load 100, R1

P2: load 1000, R1

P1: load 100, R1

VISUAL EXAMPLE OF DYNAMIC RELOCATION: BASE REGISTER



Virtual

P1: load 100, R1

P2: load 100, R1

P2: load 1000, R1

P1: load 100, R1

VISUAL EXAMPLE OF DYNAMIC RELOCATION: BASE REGISTER

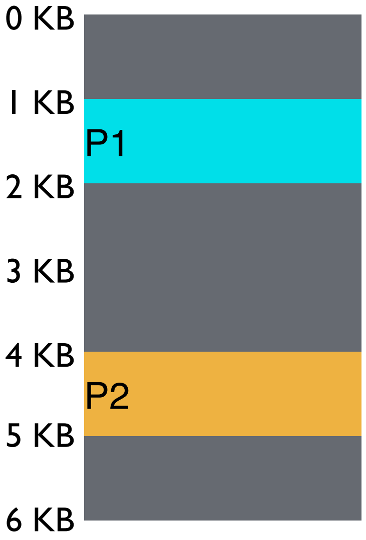
QUIZ: WHO CONTROLS THE BASE REGISTER?

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)

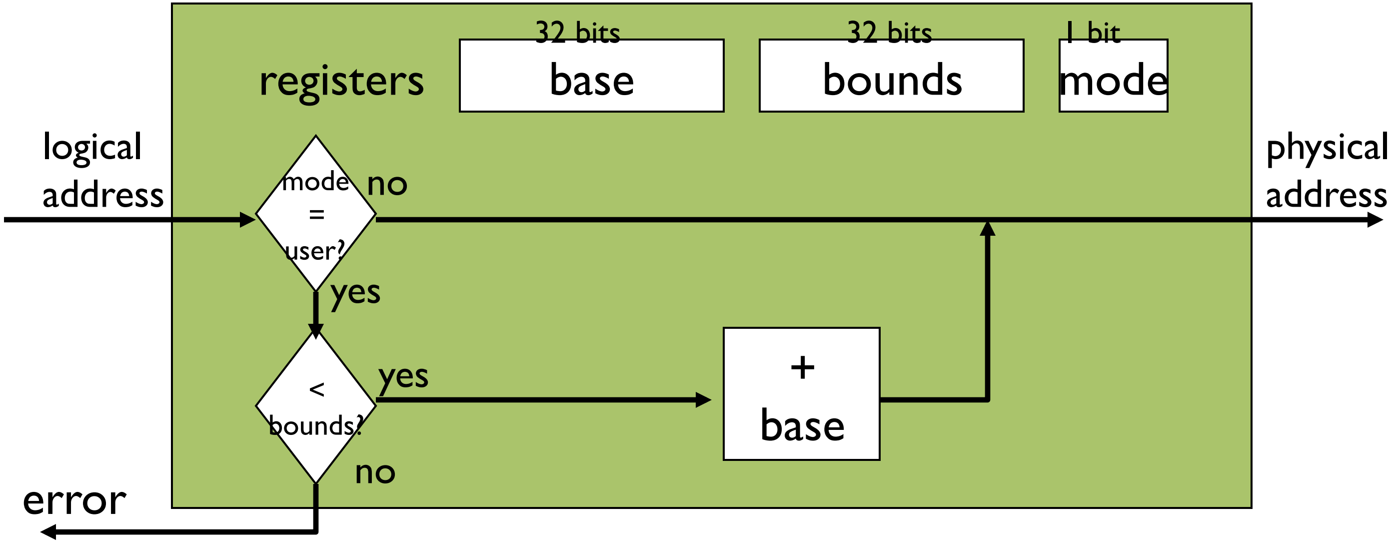
OS kills process if process loads/stores beyond bounds

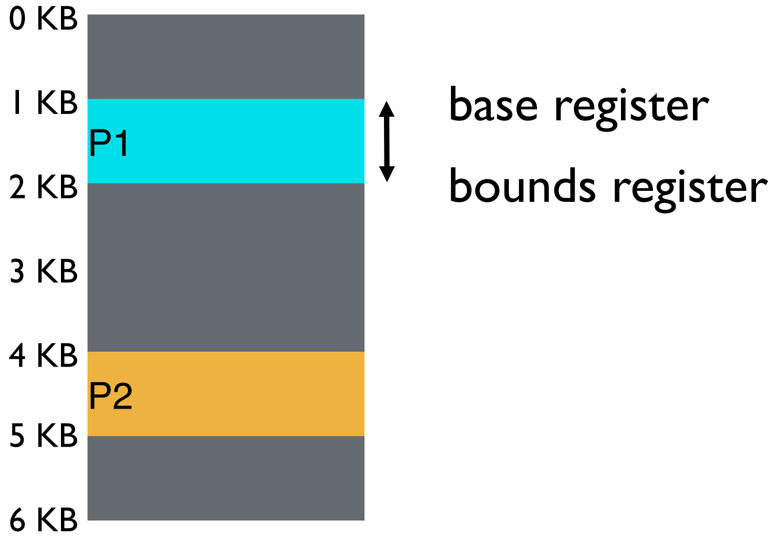
IMPLEMENTATION OF BASE+BOUNDS

Translation on every memory access of user process

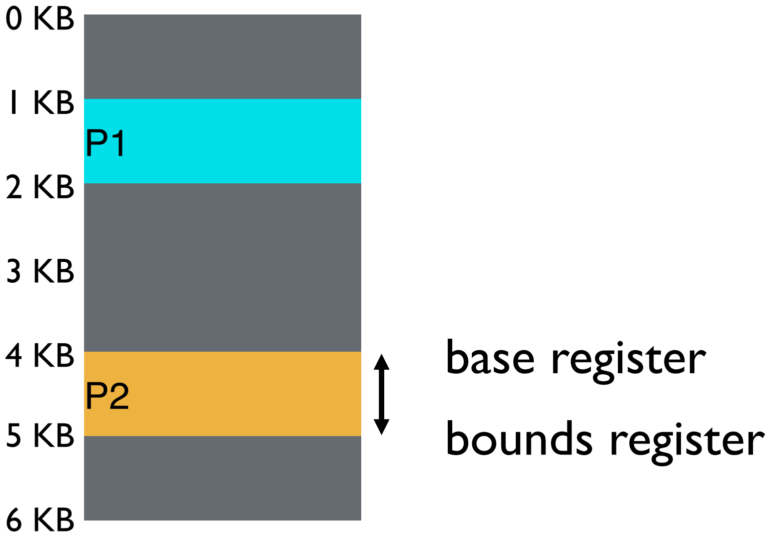
- MMU compares logical address to bounds register

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

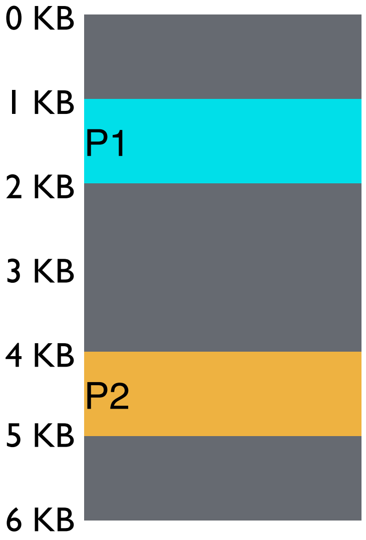




P1 is running



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?

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

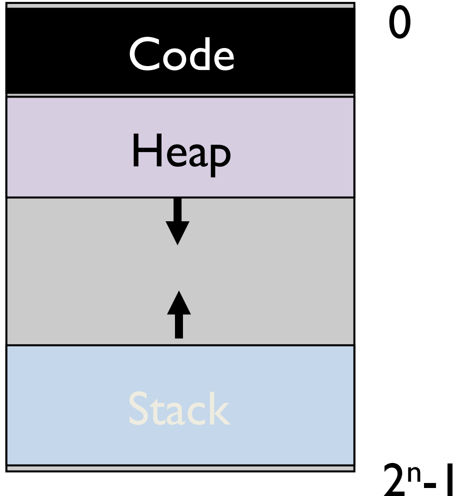
Fast: Add and compare in parallel

BASE AND BOUNDS DISADVANTAGES

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



NEXT STEPS

Project 1a: Due today! at 11.59pm

Project 1b: Out now, due Feb 8th

Thursday discussion

- xv6 introduction, walk through
- Project 1b tips

Next week: Virtual memory segmentation, paging and more!