HI!

VIRTUALIZATION: CPU TO MEMORY

Shivaram Venkataraman

CS 537, Spring 2020

ADMINISTRIVIA

- Project 1a: DONE!?
- How to use slip days? (Piazza)

- Project 1b is out, due Feb 5th (Next Wednesday) at 10 pm!
- Discussion section → 5:30pm Thursday
  - xv6 code walk through
  - How to use gdb

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

Timer

RECAP: METRICS → POLICIES

Turnaround time = *completion_time - arrival_time*

FIFO: First come, first served

SJF: Shortest job first

SCTF: Shortest completion time first

Assumptions

Metrics

Policy

RECAP: METRICS → POLICIES

Response time = *first_run_time - arrival_time*

RR: Round robin with time slice

Minimizes response time but could increase turnaround?

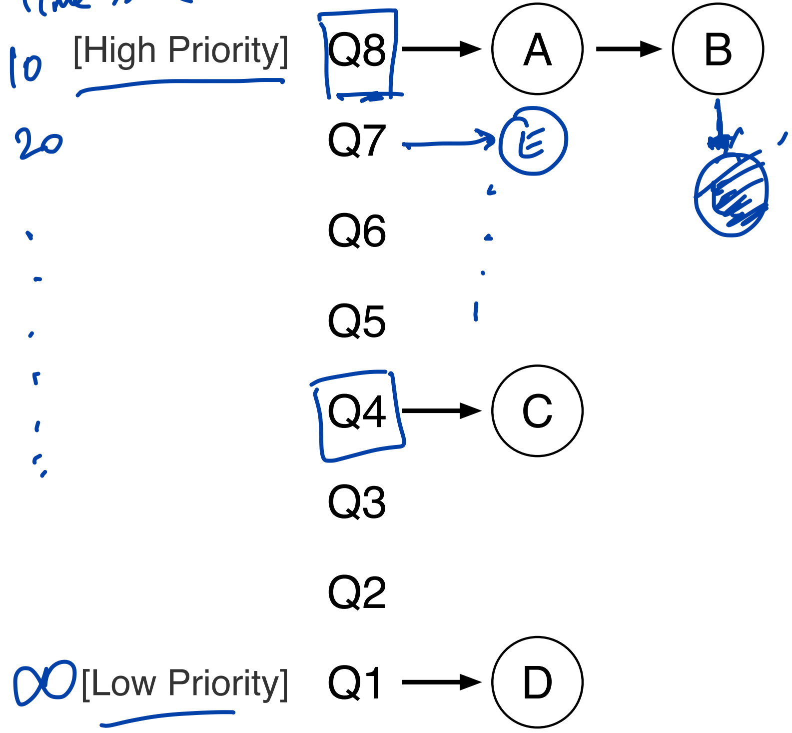
MULTI-LEVEL FEEDBACK QUEUE

MLFQ EXAMPLE

Multiple Queues at Priority levels

4) if E runs for 10ms the demote E to Q7

Time slice



Rules for MLFQ

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

Choose priority 8

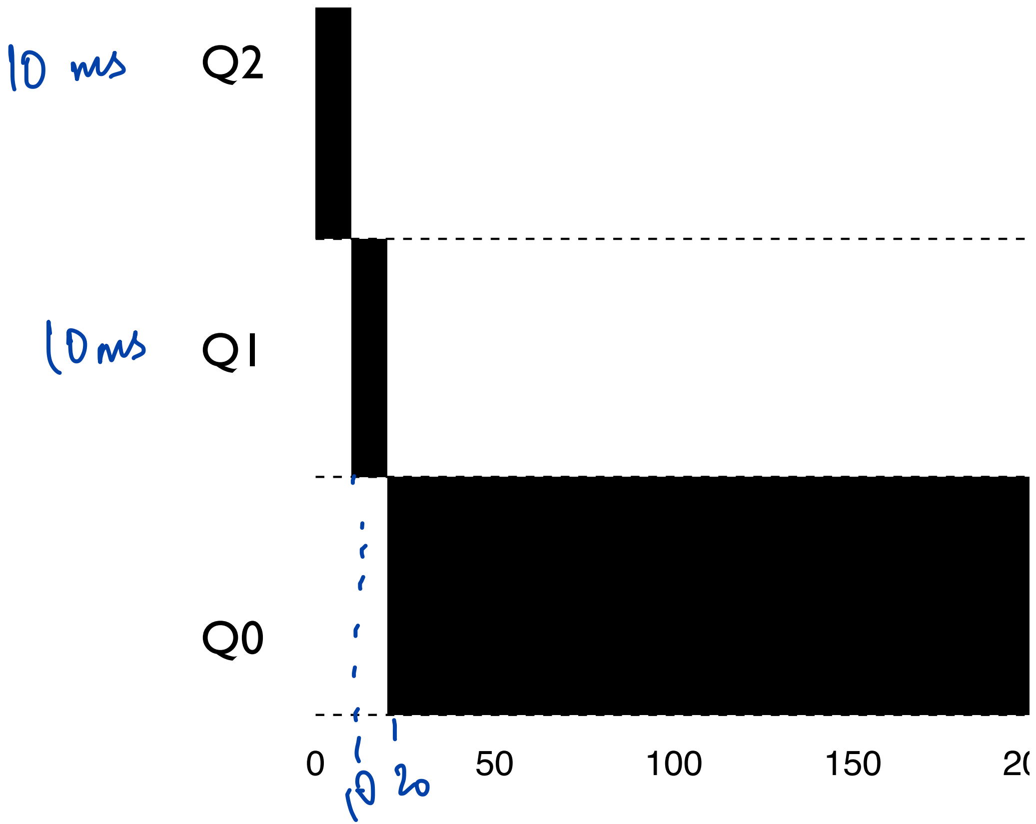
Rule 2: If priority(A) == Priority(B),
A & B run in RR

time slice

Rule 3: Processes start at top priority

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

ONE LONG JOB



INTERACTIVE PROCESS JOINS

At t: 90ms

Q2 → B

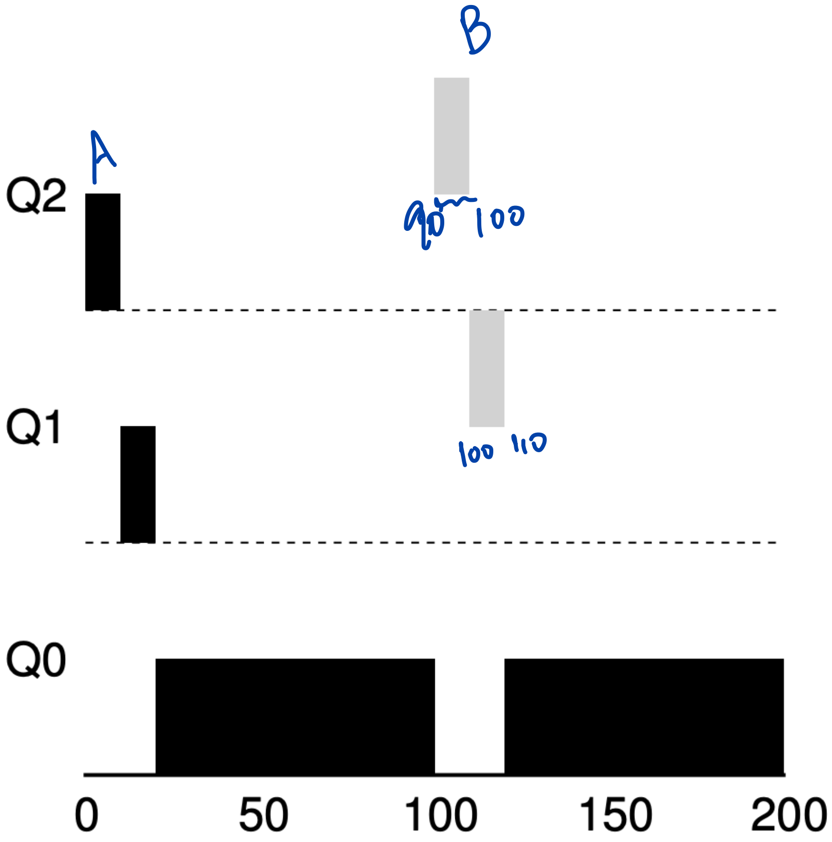
Q1

Q0 → A

At Q0

queue TS = ∞

RR TS ! : ∞



At t: 100

Q2 →

Q1 → B

Q0 → A

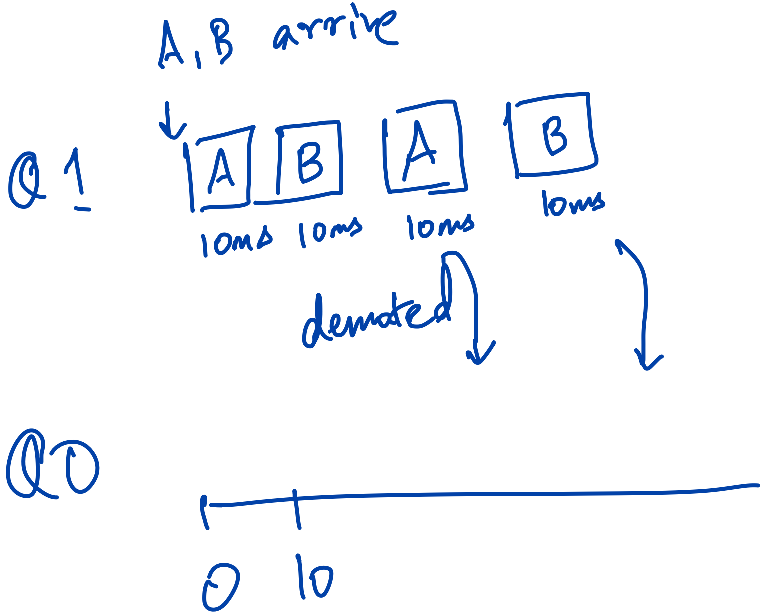
Queue TS : demote

RR TS : within queue

$Q_{TS} : RR_{TS}$

Q_{TS} : 20ms

RR_{TS} : 10ms



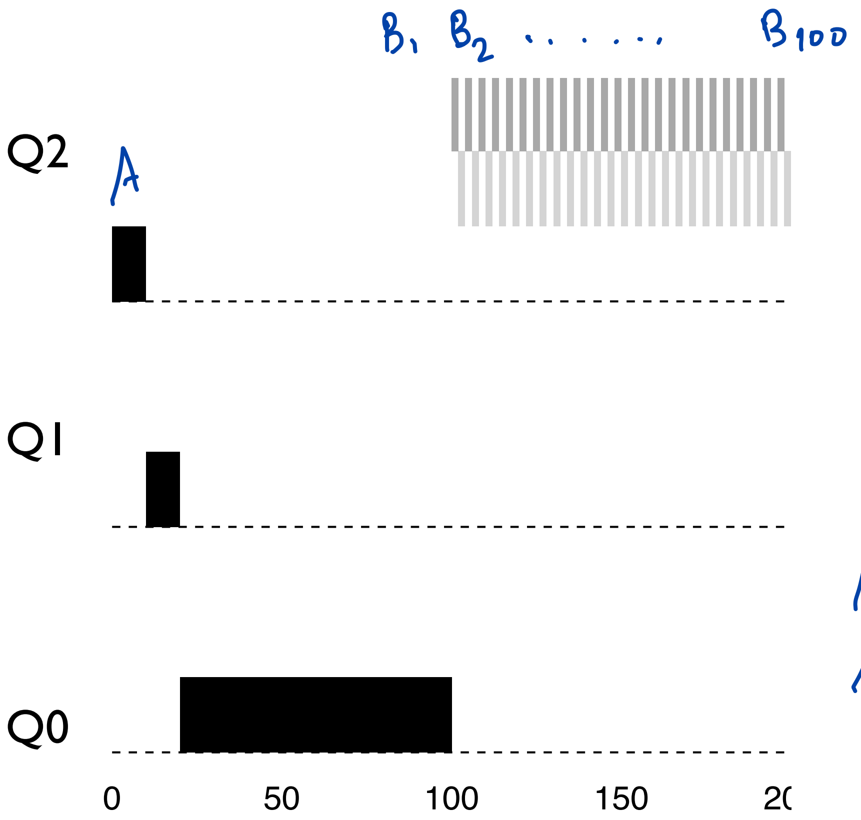
At t: 10 ms

A has used 10 ms

$< Q_{TS}$

Keep A at Q1

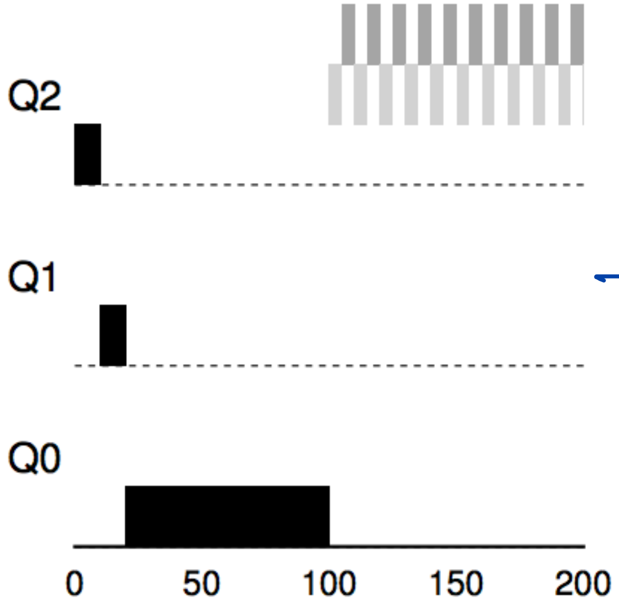
MLFQ PROBLEMS?

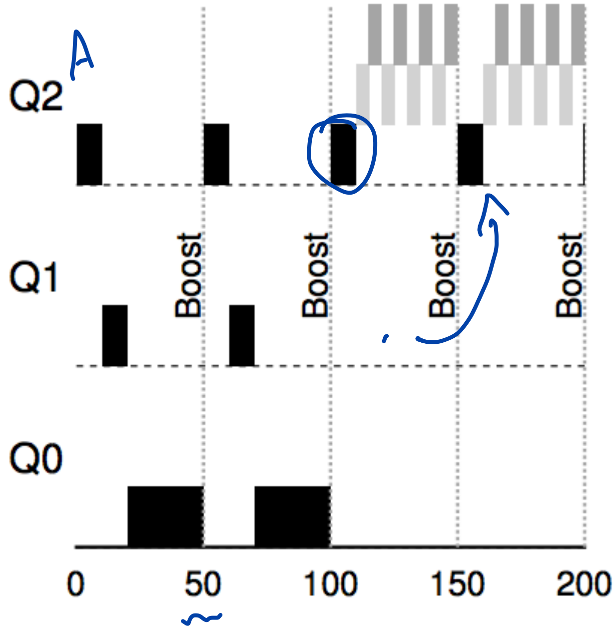


What is the problem with this schedule ?

A suffers from starvation

AVOIDING STARVATION

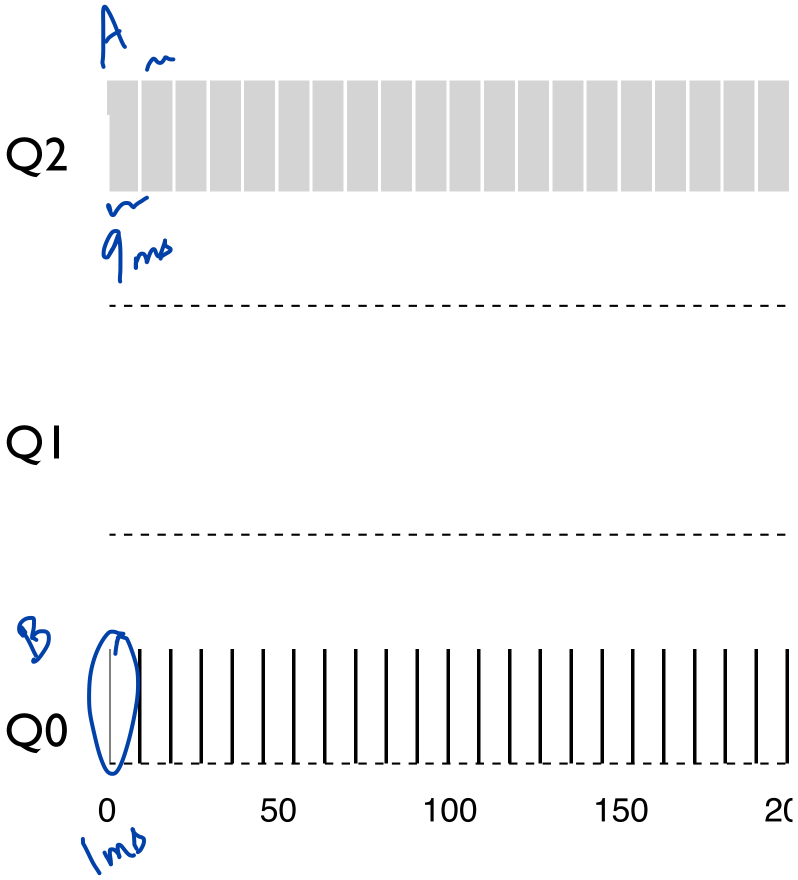




Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

S: 50 ms

GAMING THE SCHEDULER ?

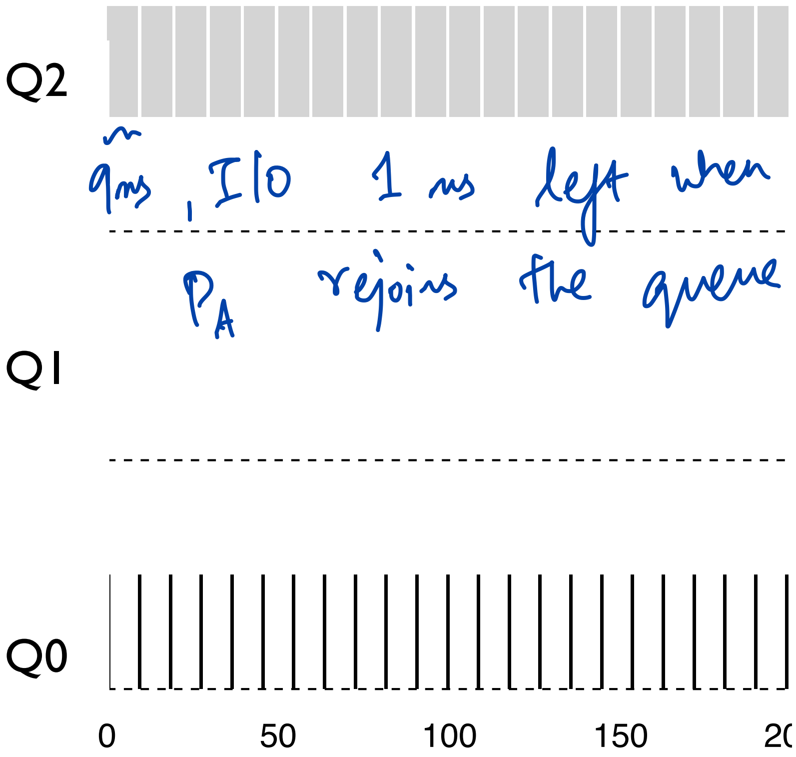


Job could trick scheduler by doing I/O just before time-slice end

P_A finishes I/O

rejoins at Q_2

GAMING THE SCHEDULER ?



Job could trick scheduler by doing I/O just before time-slice end

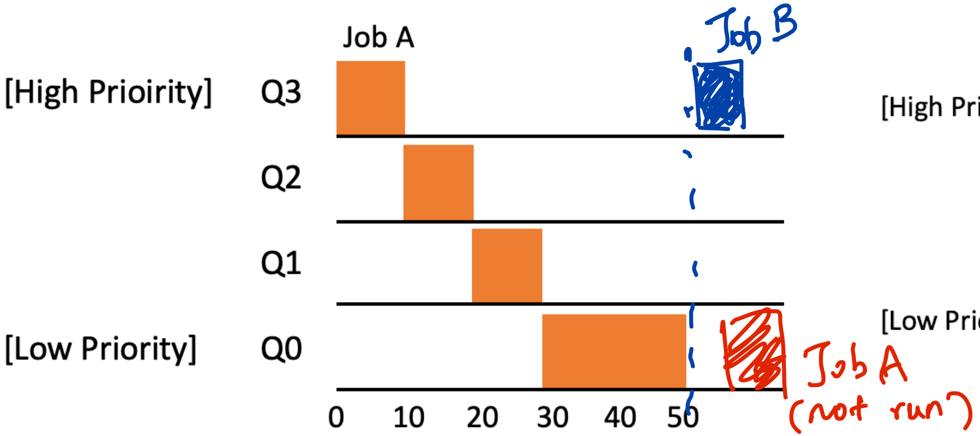
Rule 4*: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced

QUIZ 5

https://tinyurl.com/cs537-sp20-quiz5

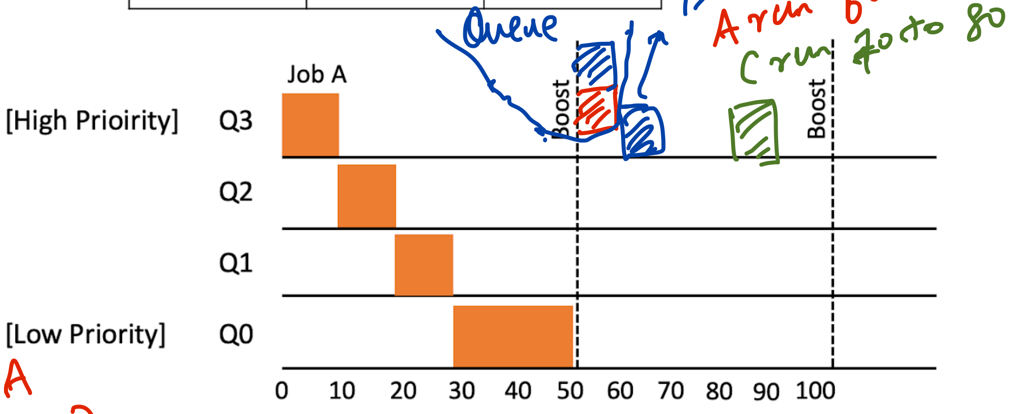
Without priority boosting

Jobs	Runtime	Arrival Time
Job A	100	0
Job B	10	50



Options A & B were both correct!

Jobs	Runtime	Arrival Time
Job A	100	0
Job B	10	50
Job C	20	70



CPU SUMMARY

Mechanism

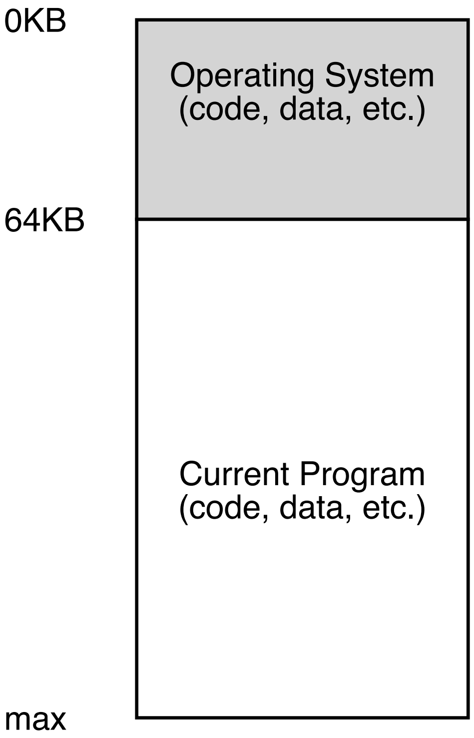
- Process abstraction
- System call for protection
- Context switch to time-share

Policy

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

VIRTUALIZING MEMORY

BACK IN THE DAY…



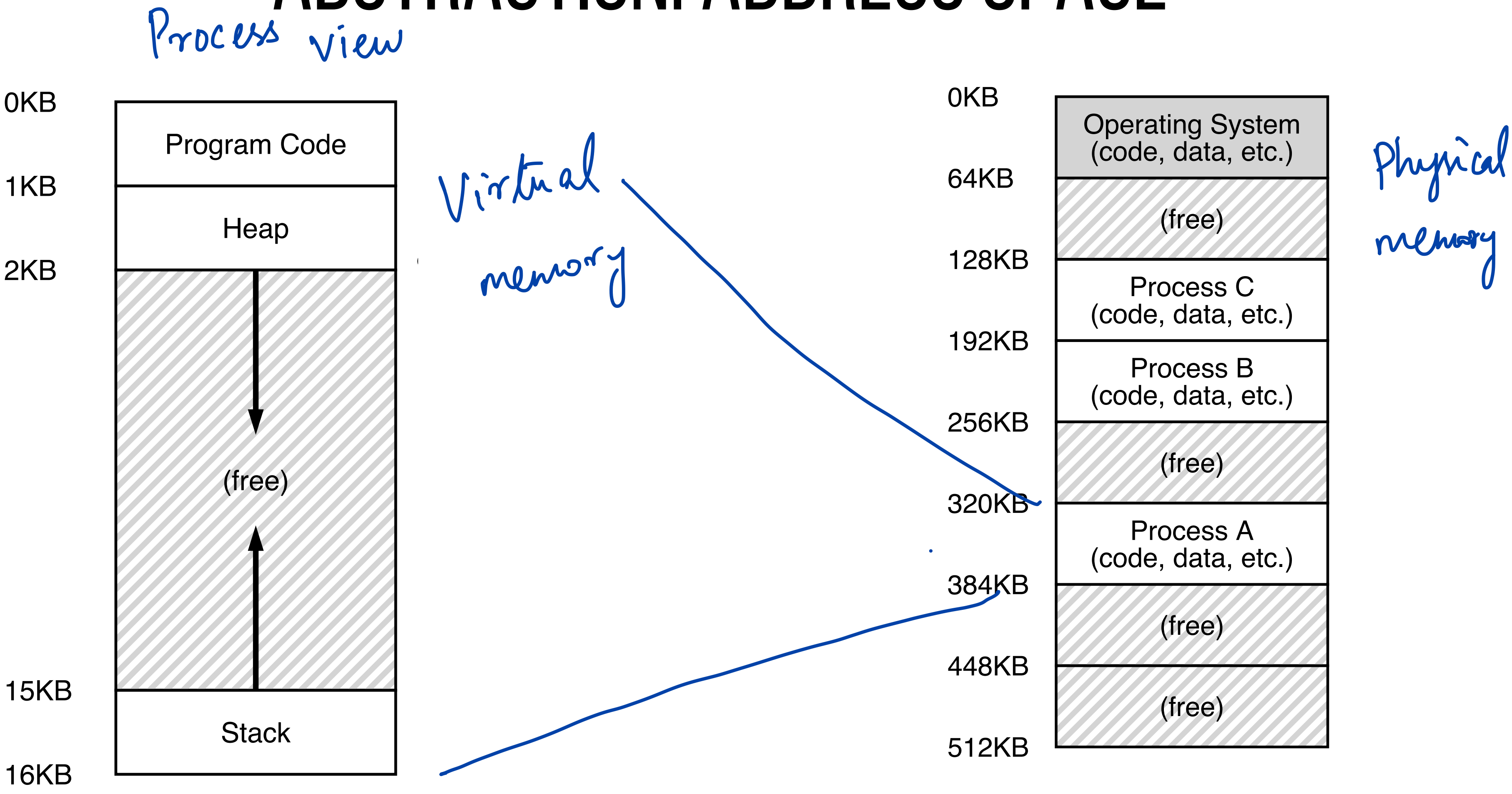
Uniprogramming: One process runs at a time

One program that exists in the system

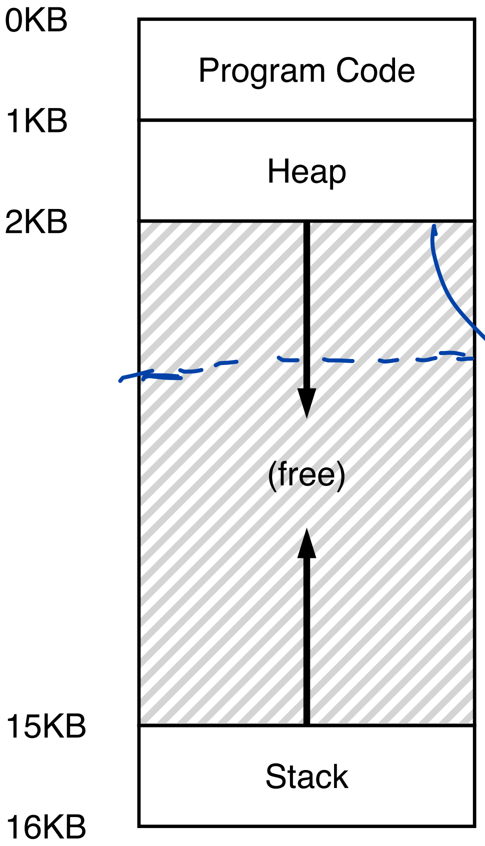
MULTIPROGRAMMING GOALS

- Transparency: Process is unaware of sharing
- Protection: Cannot corrupt OS or other process memory
- Efficiency: Do not waste memory or slow down processes
- Sharing: Enable sharing between cooperating processes

ABSTRACTION: ADDRESS SPACE



WHAT IS IN ADDRESS SPACE?



the code segment:
where instructions live

```
mov $3 %eax
callq add
:
```

the heap segment:
contains malloc'd data
dynamic data structures
(it grows downward)

malloc() , new

as you allocate more memory

Static: Code and some global variables

Dynamic: Stack and Heap

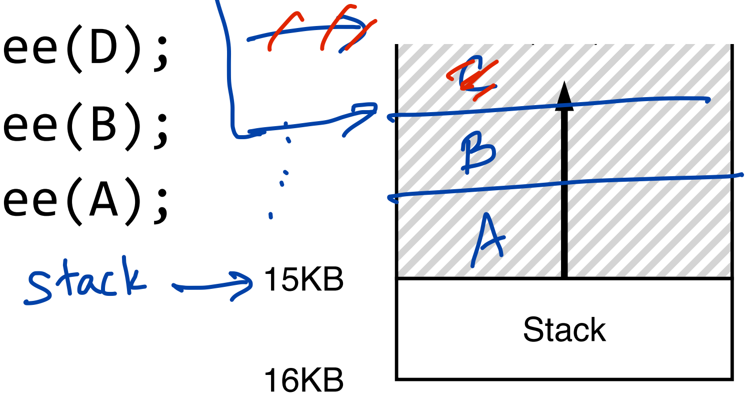
(it grows upward)
the stack segment:
contains local variables
arguments to routines,
return values, etc.

local variables
arguments, return

STACK ORGANIZATION

LIFO behavior

```
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

→ No fragmentation!

No empty regions which are used

WHAT GOES ON STACK?

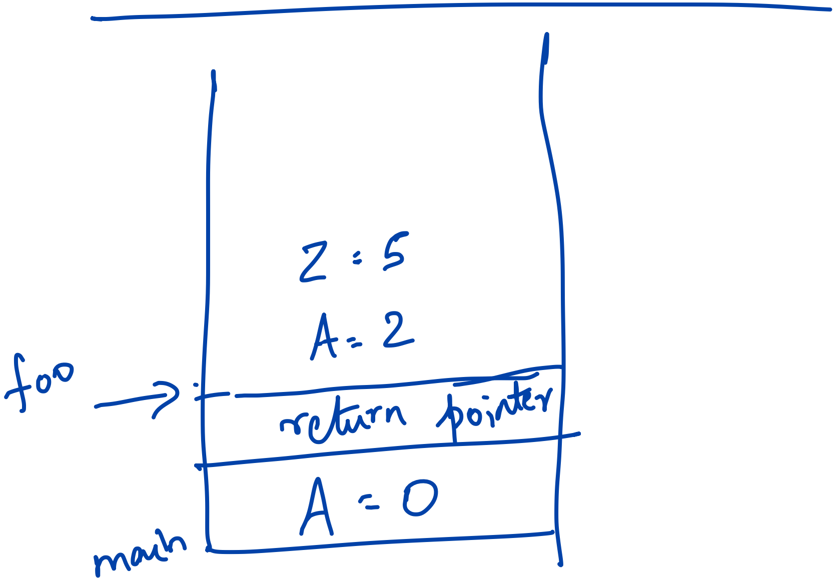
Calling convention
x86

```
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 variable

arguments

return value
also goes on stack



HEAP ORGANIZATION

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

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

fragmentation

c: malloc(12)

a = malloc(16)

b: malloc(24)

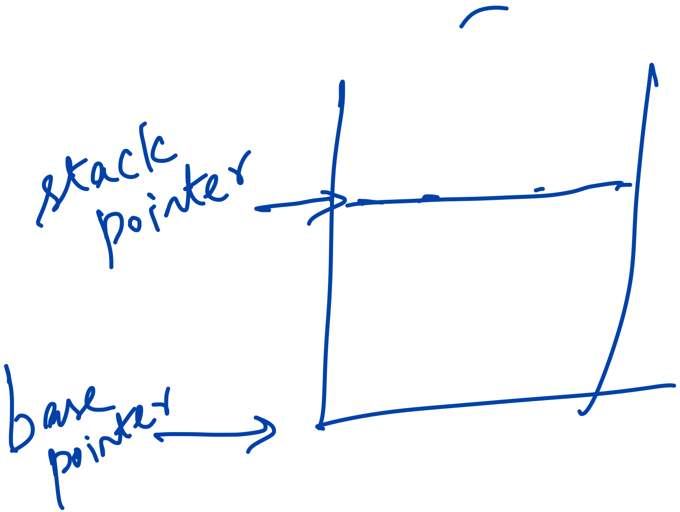
free(a)

free(c)

MEMORY ACCESS

```c
#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)
```

register

memory

%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)

Fetch Inst at addr
0x10

Exec: fetch data at
0x208

Fetch Inst at 0x13

Exec:

Fetch inst at 0x19

Exec: Store data at
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

QUIZ 6

https://tinyurl.com/cs537-sp20-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

HOW TO VIRTUALIZE MEMORY

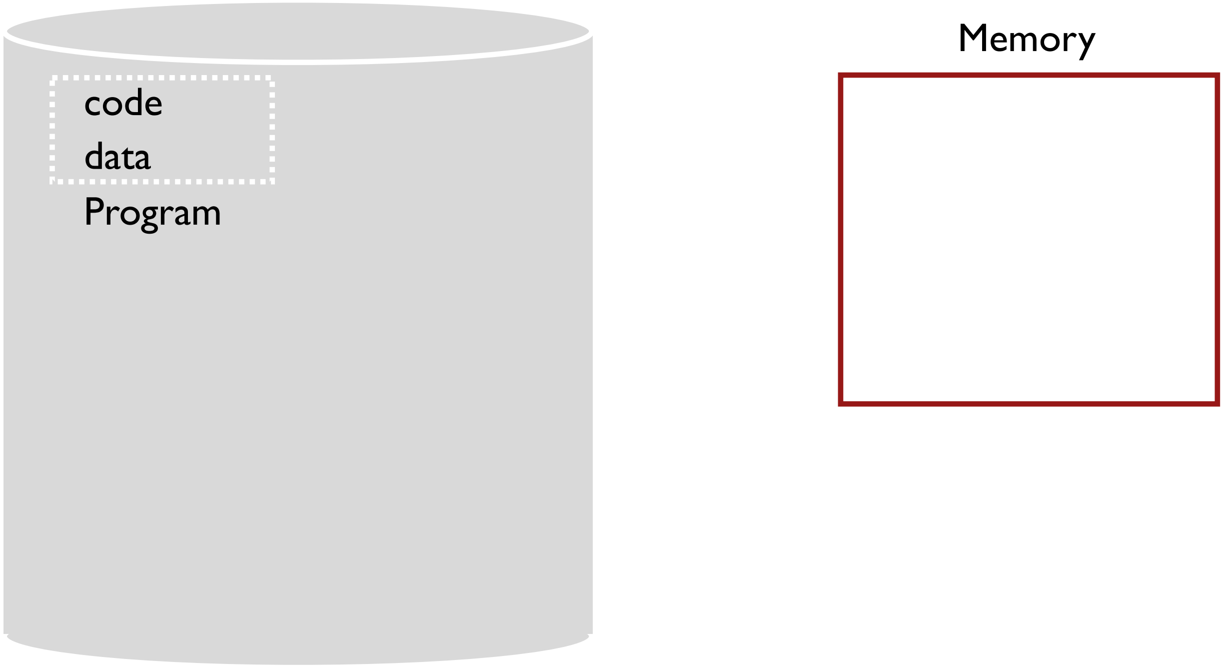
Problem: How to run multiple processes simultaneously?

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



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

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

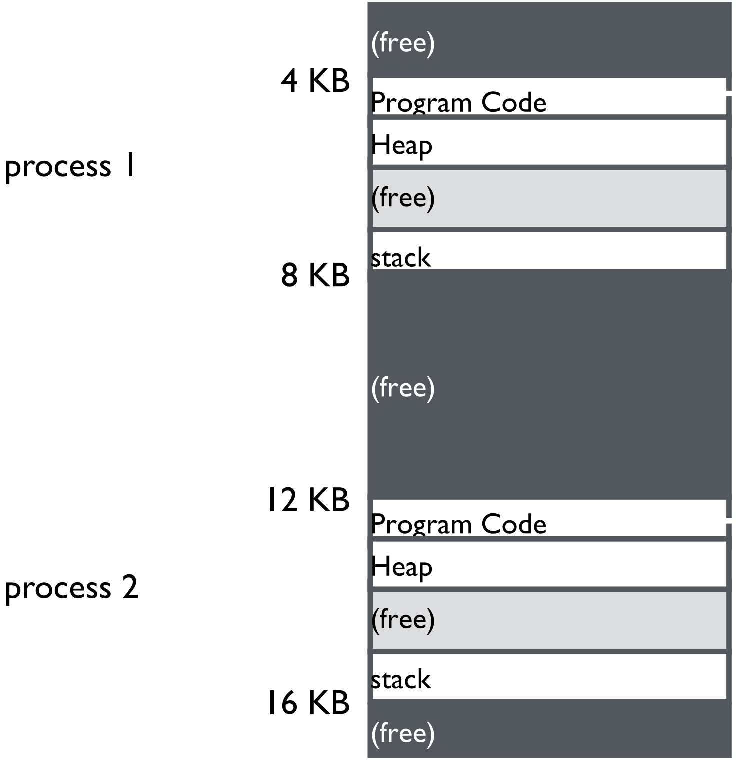
rewrite

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

rewrite

```
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

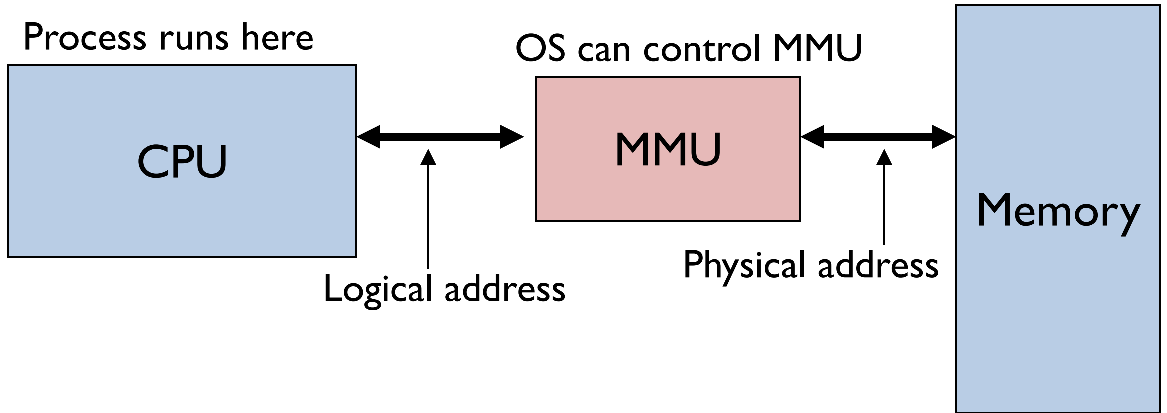
Goal: Protect processes from one another

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

- 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

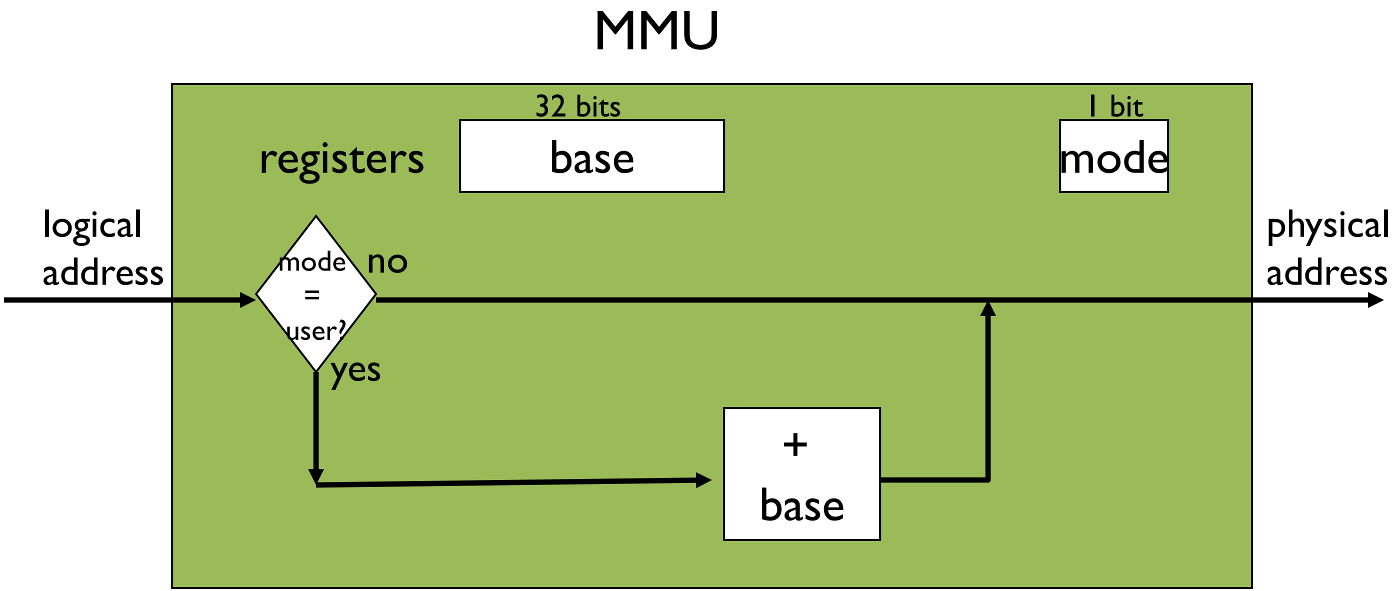
User mode: User processes run

- 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



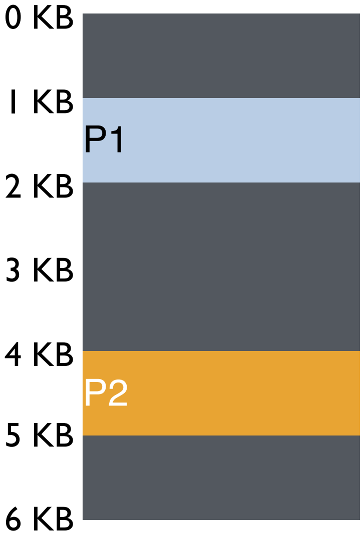
DYNAMIC RELOCATION WITH BASE REGISTER

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

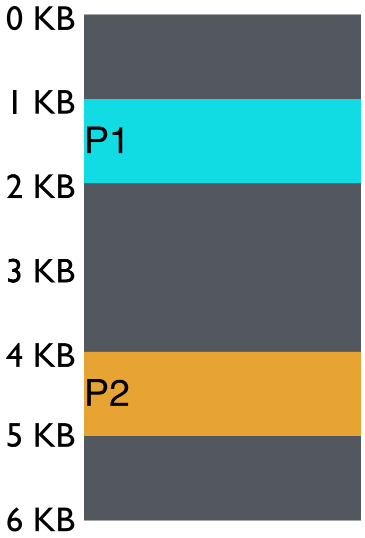
P1: load 100, R1

P2: load 100, R1

P2: load 1000, R1

P1: load 100, R1

VISUAL EXAMPLE OF DYNAMIC RELOCATION: BASE REGISTER



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5096, R1
P1: load 1000, 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 5096, R1	
P1: load 100, R1	load 2024, R1	
P1: store 3072, R1	store 4096, R1	(3072 + 1024)

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

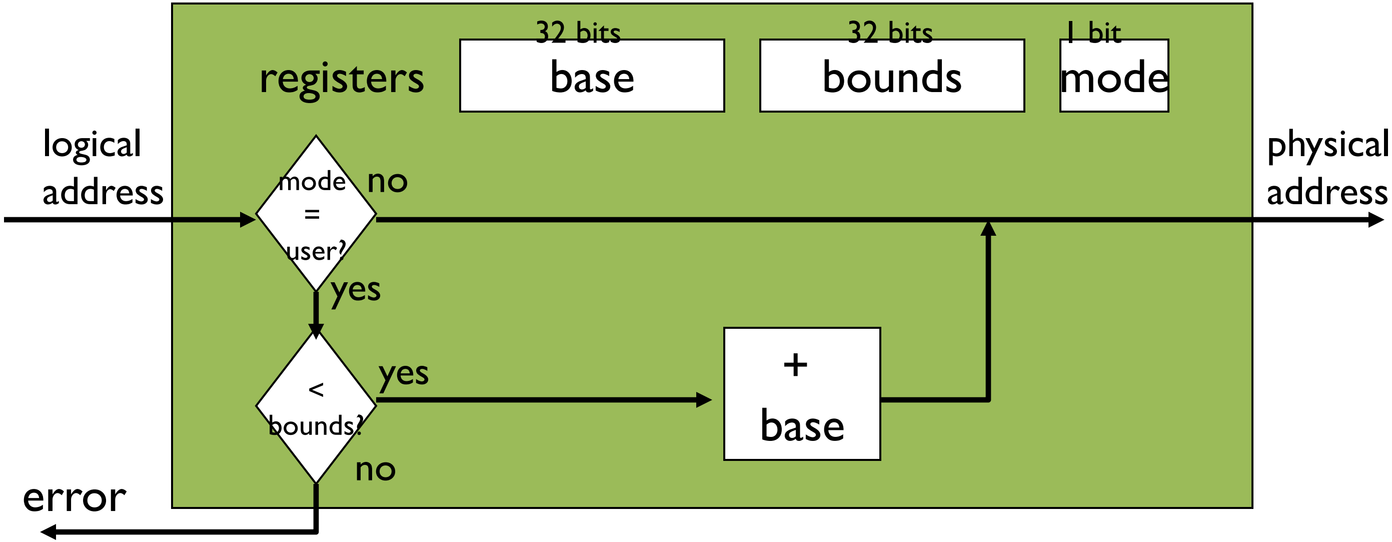
- Sometimes defined as largest physical address (base + size)

OS kills process if process loads/stores beyond bounds

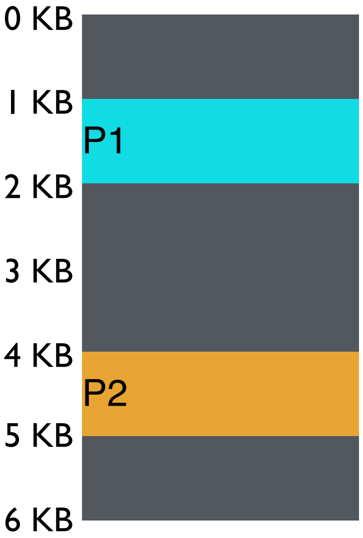
IMPLEMENTATION OF BASE+BOUNDS

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




0 KB
1 KB
2 KB
3 KB
4 KB
5 KB
6 KB
P1
P2
base register
bounds register



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

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

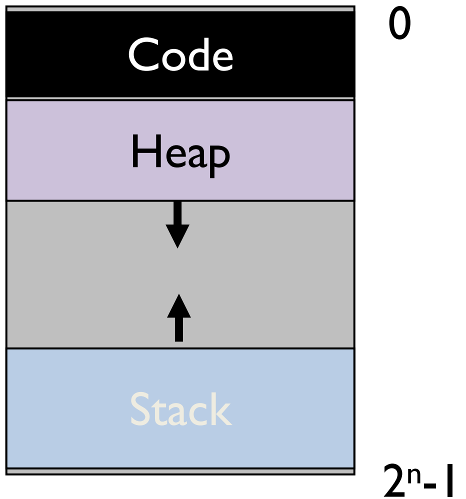
Fast: Add and compare in parallel

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 parts of address space



NEXT STEPS

Project 1b: Out now, due Feb 5th

Thursday discussion

- xv6 introduction, walk through
- Project 1b tips

Next week: Virtual memory segmentation, paging and more!