Hello!

VIRTUALIZATION: CPU TO MEMORY

Shivaram Venkataraman

CS 537, Spring 2023

ADMINISTRIVIA

- Project 1: DONE!?
- How to use slip days? (Piazza) / Canvas project spec

- Project 2 is out, due next Wednesday

→ OH from 3pm to 4pm CS 7367

AGENDA / LEARNING OUTCOMES

CPU virtualization

- Recap of scheduling policies (Tue)
- Lottery Scheduling, ~~Multi-CPU~~

Memory virtualization

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

RECAP: CPU VIRTUALIZATION

RECAP: SCHEDULING MECHANISM

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

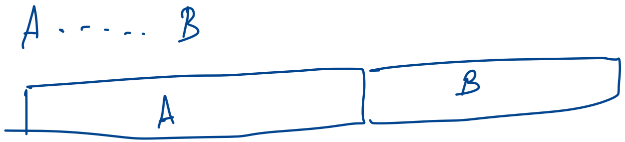
RECAP: METRICS → POLICIES

↳ what metrics

Turnaround time = *completion_time - arrival_time*

FIFO: First come, first served

jobs



schedule jobs in their arrival order

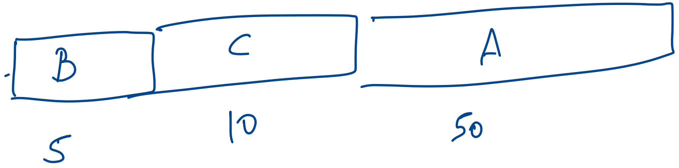
SJF: Shortest job first

→ short jobs can get stuck behind long jobs

A : 50

B : 5

C : 10



RECAP: METRICS → POLICIES

Response time = *first_run_time - arrival_time*

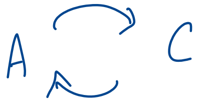
Pre-emptive scheduling

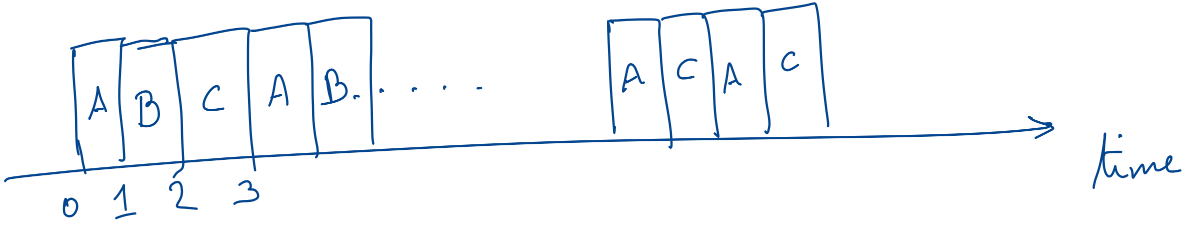
at any point OS can stop running & replace it.

RR: Round robin with time slice = 1

Minimizes response time but could increase turnaround?

A: 50, B: 5, C: 10





RECAP: MULTI-LEVEL FEEDBACK QUEUE

What if we don't know how long a job will run?

Support two job types with distinct goals

- "interactive" programs care about response time → code editor
- "batch" programs care about turnaround time → Compiling code

Running tests

Approach:

Multiple levels of round-robin

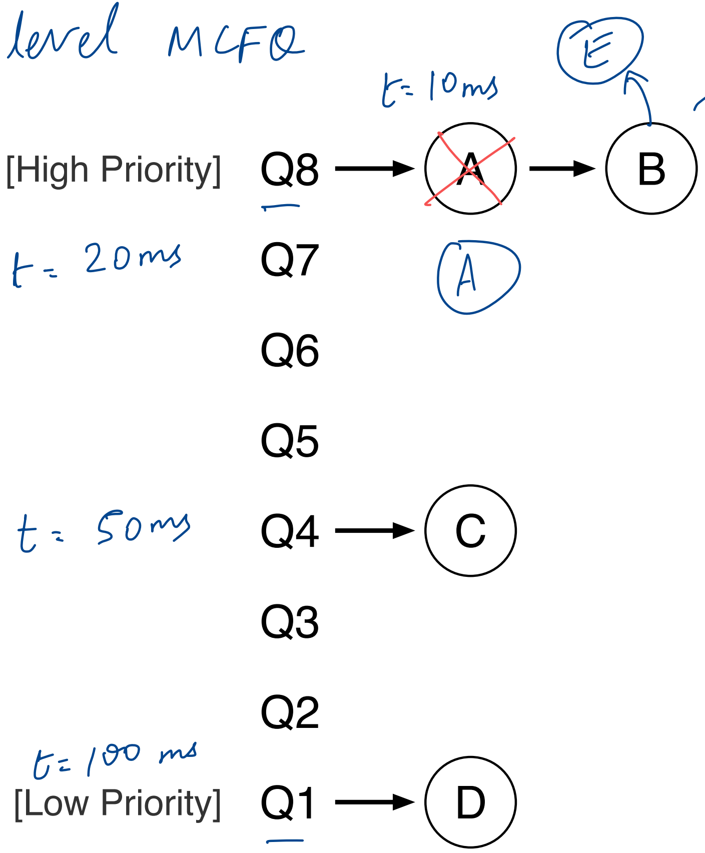
Each level has higher priority than lower level

Can preempt them

RECAP: MULTI-LEVEL FEEDBACK QUEUE

8-level MLFQ

round robin scheduling in Q8 → A, B, E



Rules for MLFQ

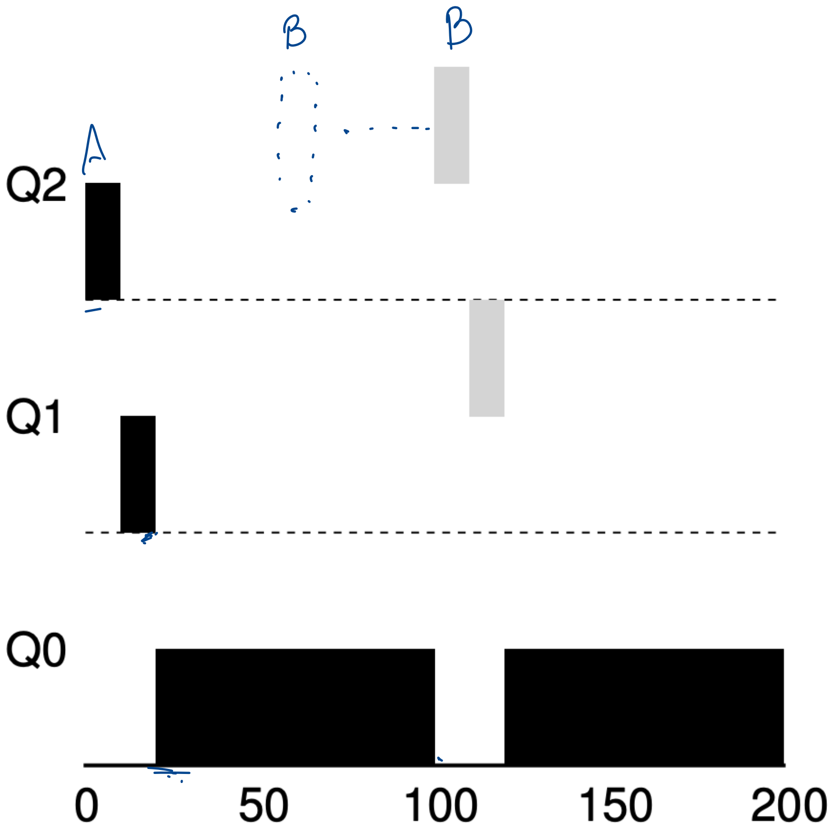
Rule 1: If priority(A) > Priority(B)
A runs

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

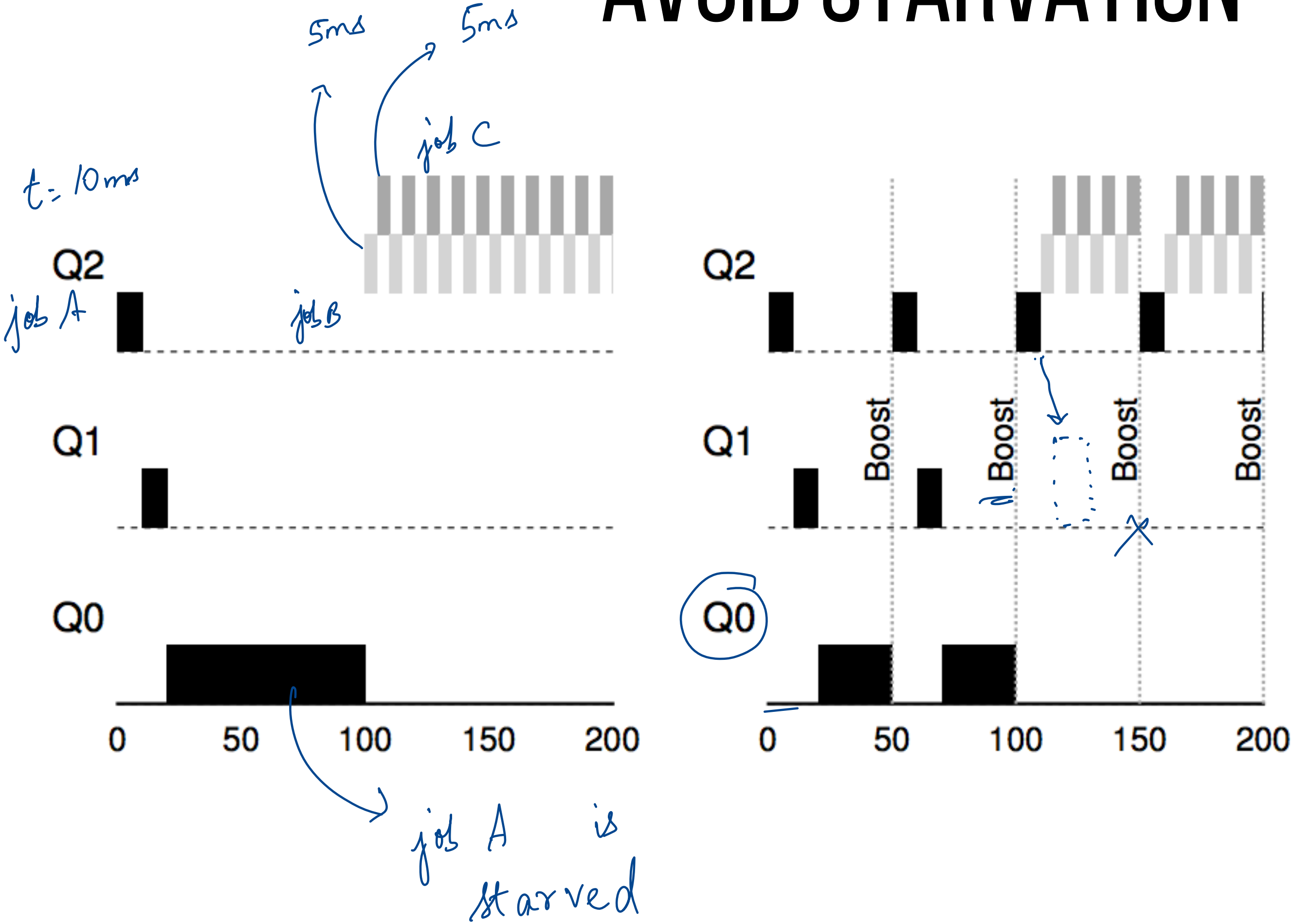
Rule 3: Processes start at top priority

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

INTERACTIVE PROCESS JOINS



AVOID STARVATION

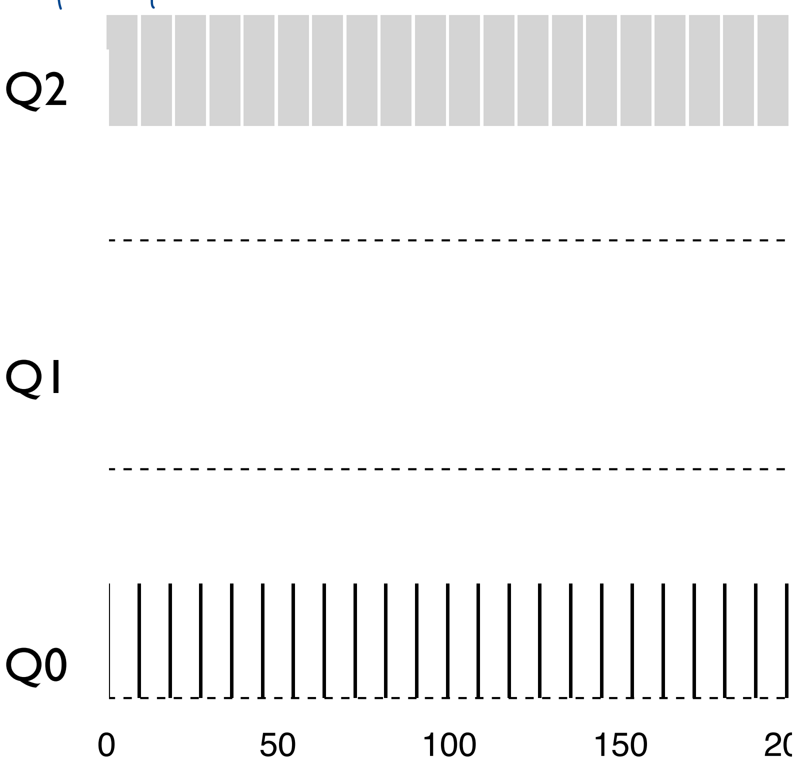


Priority Boost!

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

GAMING THE SCHEDULER ?

9ms 9ms 9+9 = 18ms → demoted to Q 1



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

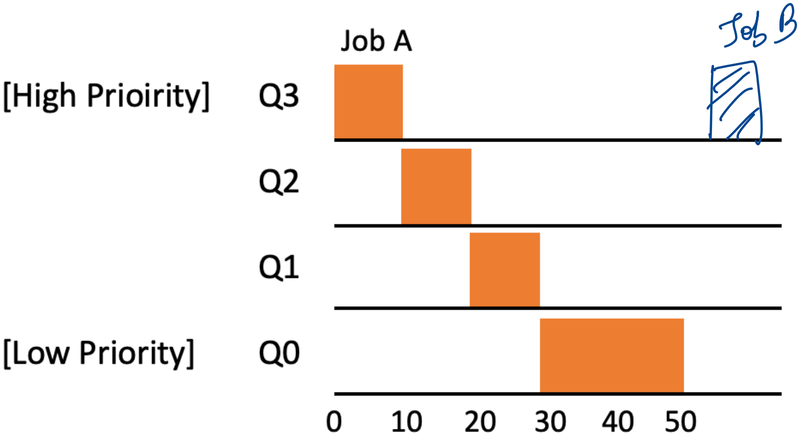
https://tinyurl.com/cs537-sp23-quiz5

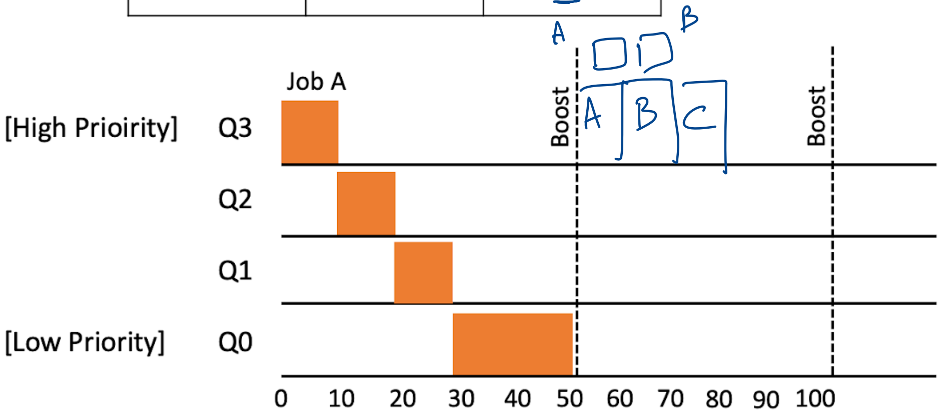
Q0

Q3

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

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





FAIRNESS IN SCHEDULING

Metrics so far: turn around time, response time.

New metric: Fairness!

→ all running jobs

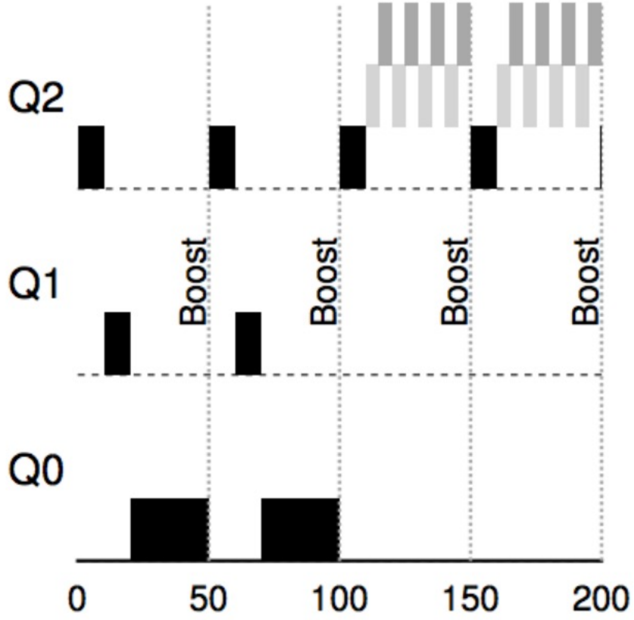
3 users; each get 1/3rd of CPU

no matter how long they run for

Is MLFQ fair?

→ jobs which run at Q0

get more cycles



LOTTERY SCHEDULING ~ mid-90s

Approach:

1. Give processes lottery tickets
2. Whoever wins the lottery runs
3. Higher priority => more tickets

Administrator or Config. →

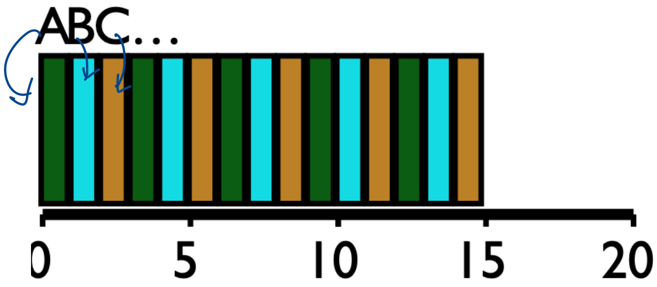
JOB	Tickets
A	10
B	20
C	30

Total 60

random

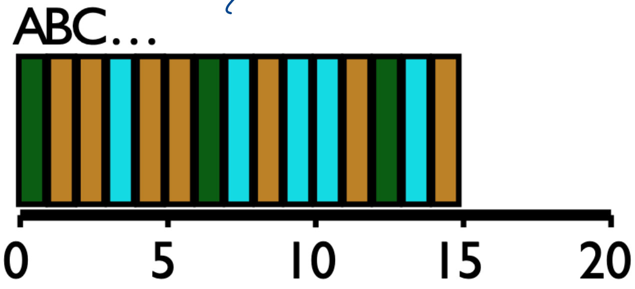
0 – 10	A
10 - 30	B
30 - 60	C

RR



Lottery

C ~ 7/15



IMPLEMENTING LOTTERY SCHEDULING

Scheduler:

```
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t *current = head;
while (current) {
    counter += current->tickets;
    if (counter > winner) break;
    current = current->next;
}
// current gets to run
```

402

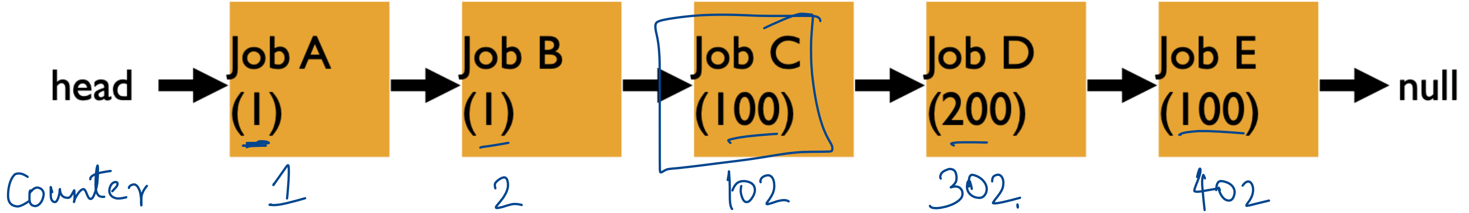
50

0 - 1	A
1 - 2	B
2 - 102	C

⋮

Who runs if winner is:

50	Job C
350	Job E
0	



5 jobs

linked list

CPU SUMMARY

Mechanism

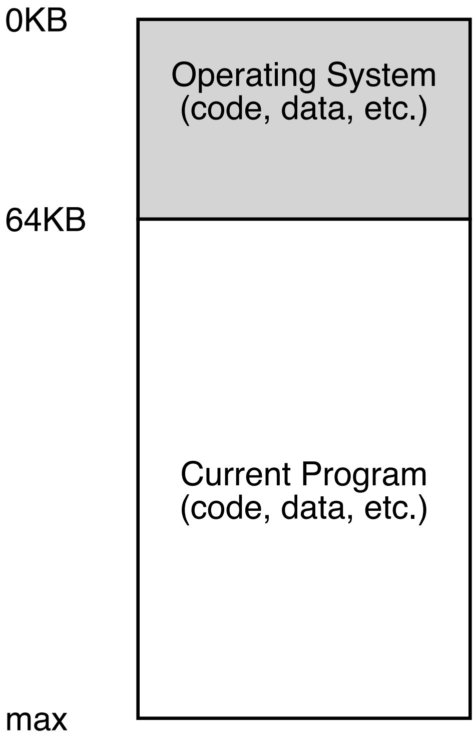
- Process abstraction
- System call for protection
- Context switch to time-share

Policy

- Metrics: turnaround time, response time
- Balance using MLFQ
- Fairness with Lottery Scheduling

VIRTUALIZING MEMORY

BACK IN THE DAY...



0KB

Operating System
(code, data, etc.)

64KB

Current Program
(code, data, etc.)

max

Uniprogramming: One process runs at a time

↳ we want to run more than one process

MULTIPROGRAMMING GOALS

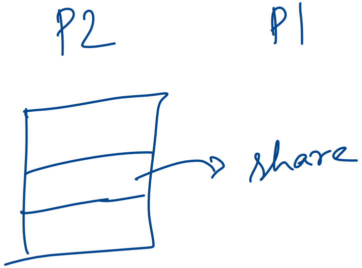
Transparency: Process is unaware of sharing → simplifies programming

Protection: Cannot corrupt OS or other process memory

Efficiency: Do not waste memory or slow down processes

↳ Minimize

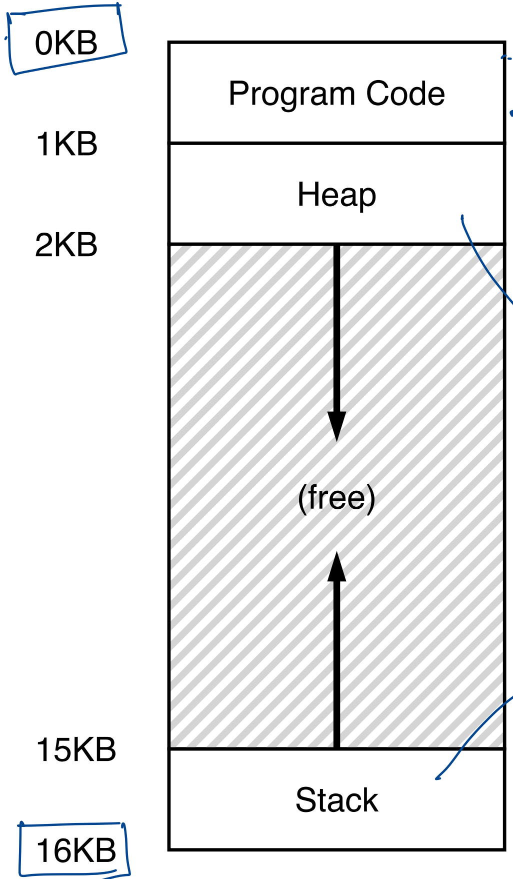
Sharing: Enable sharing between cooperating processes



Virtual address ≡ Process sees

ABSTRACTION: ADDRESS SPACE

Physical



0KB
Program Code
1KB
Heap
2KB
(free)
15KB
Stack
16KB

Every Process has memory regions

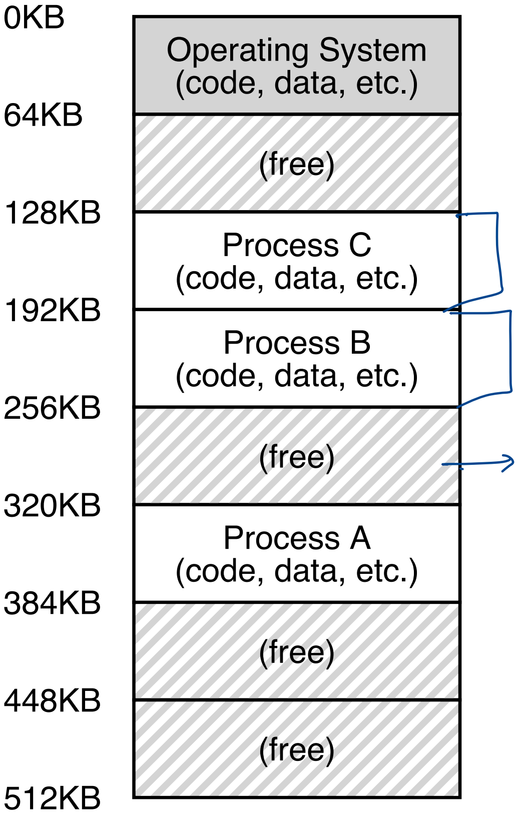
instructions used

static data

dynamic memory allocations

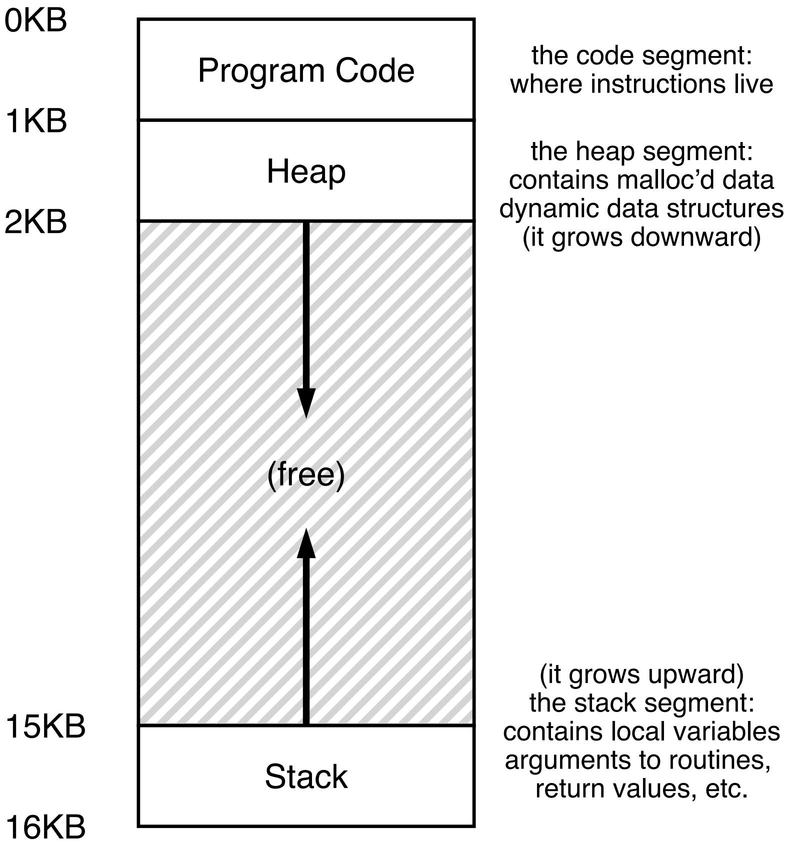
local variables

arguments



0KB
Operating System
(code, data, etc.)
64KB
(free)
128KB
Process C
(code, data, etc.)
192KB
Process B
(code, data, etc.)
256KB
(free)
320KB
Process A
(code, data, etc.)
384KB
(free)
448KB
(free)
512KB

WHAT IS IN ADDRESS SPACE?



0KB
Program Code
the code segment: where instructions live
1KB
Heap
the heap segment: contains malloc'd data dynamic data structures (it grows downward)
2KB
(free)
(it grows upward) the stack segment: contains local variables arguments to routines, return values, etc.
15KB
Stack
16KB

Static: Code and some global variables

Dynamic: Stack and Heap

ASIDE: HOW TO CREATE A PROCESS?

Unix-like OS use *fork()* → system call

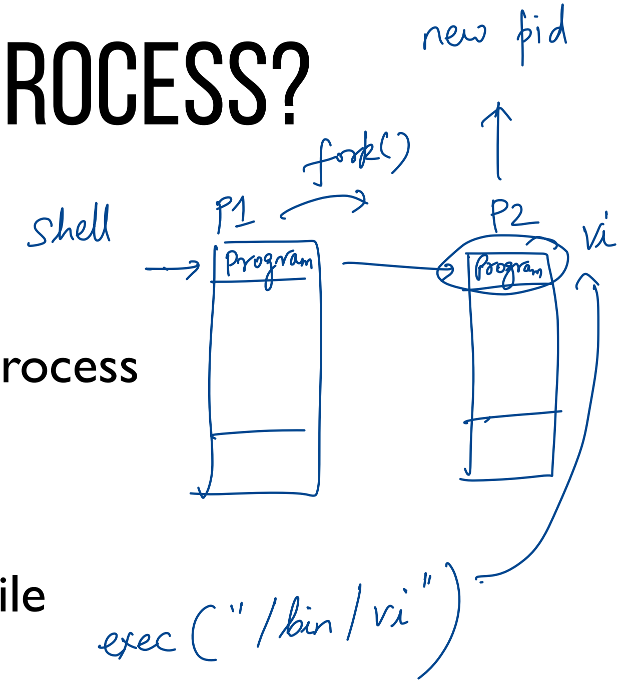
Fork() - Clones the calling process to create a child process

  Make copy of code, data, stack etc.

  Add new process to ready list

Exec(char *file): Replace current data and code with file

→ system call



Advantages: Flexible, clean, simple

Disadvantages: Wasteful to perform copy and overwrite of memory

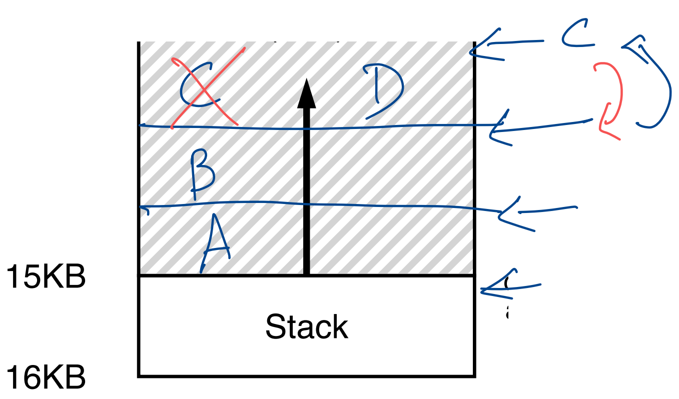
STACK ORGANIZATION

```
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

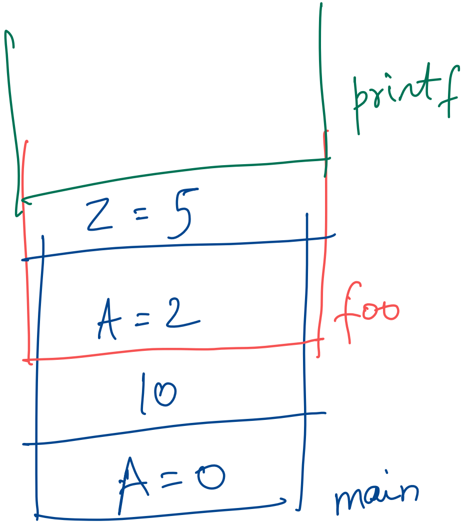
No fragmentation!



Stack Last In First Out

WHAT GOES ON STACK?

```
main () {
    int A = 0;
    foo(10);
    printf("A: %d\n", A);
}
void foo (int Z) {
    int A = 2;
    Z = 5;
    printf("A: %d Z: %d\n", A, Z);
}
```



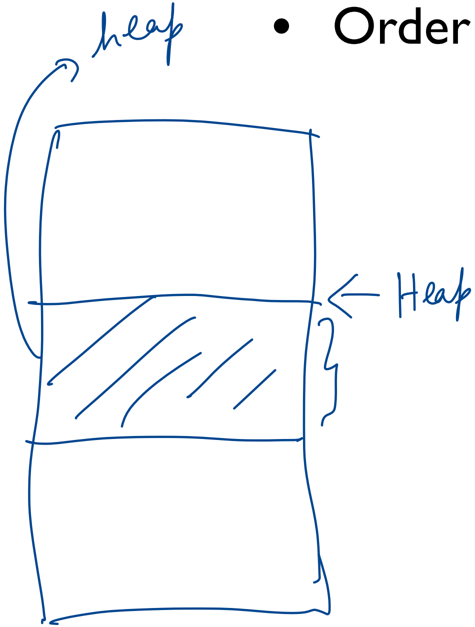
HEAP ORGANIZATION

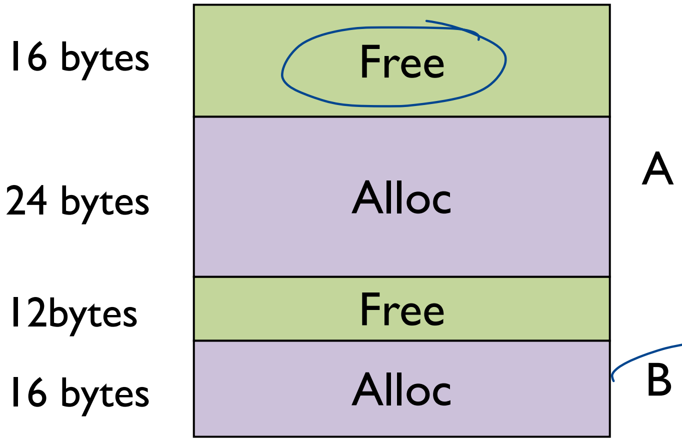
Allocate 24 bytes

→ No contiguous region

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

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





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

memory allocators which handle allocation within range

B = malloc (16 bytes)

MEMORY ACCESS

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

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

Copy data register

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

add

Copy back to 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)

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	
main	
y	
z	
*z	

HOW TO VIRTUALIZE MEMORY

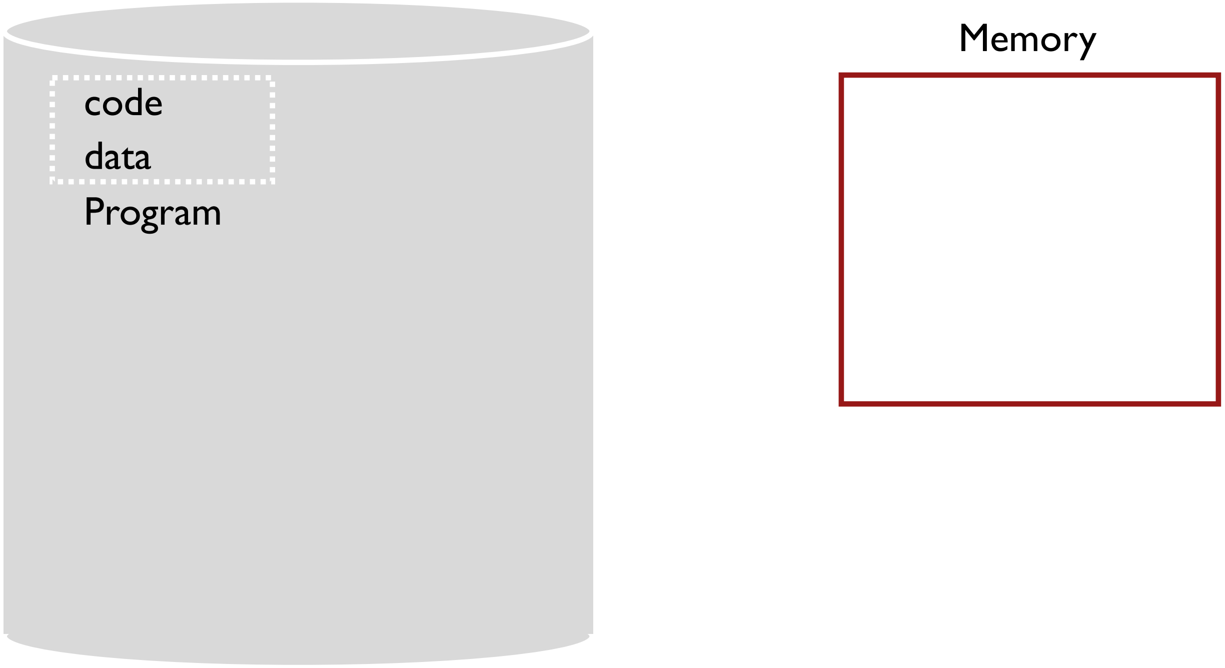
Problem: How to run multiple processes simultaneously?

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!

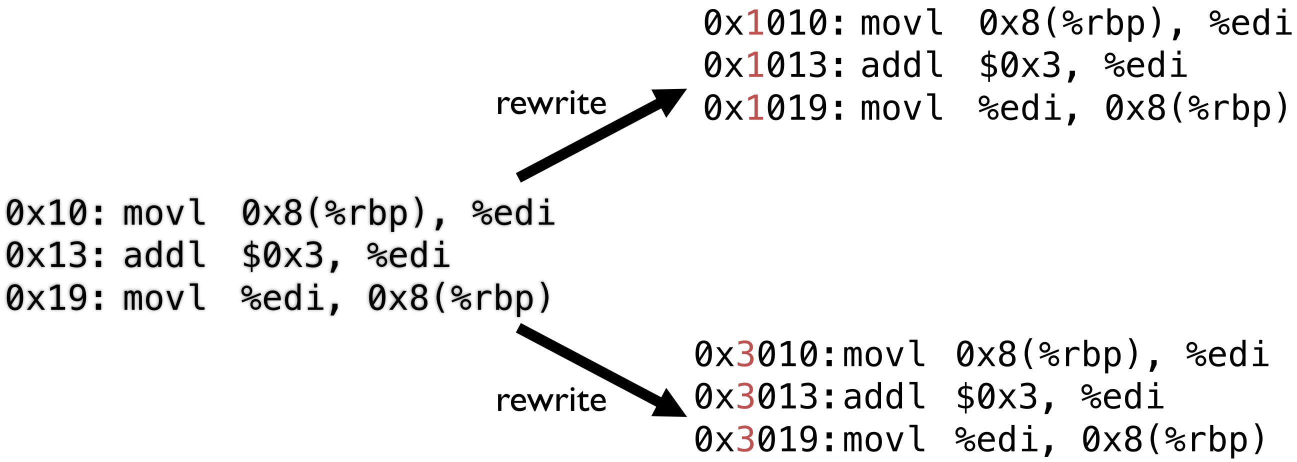
- At same time, space of memory is divided across processes
- Remainder of solutions all use space sharing

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)
```

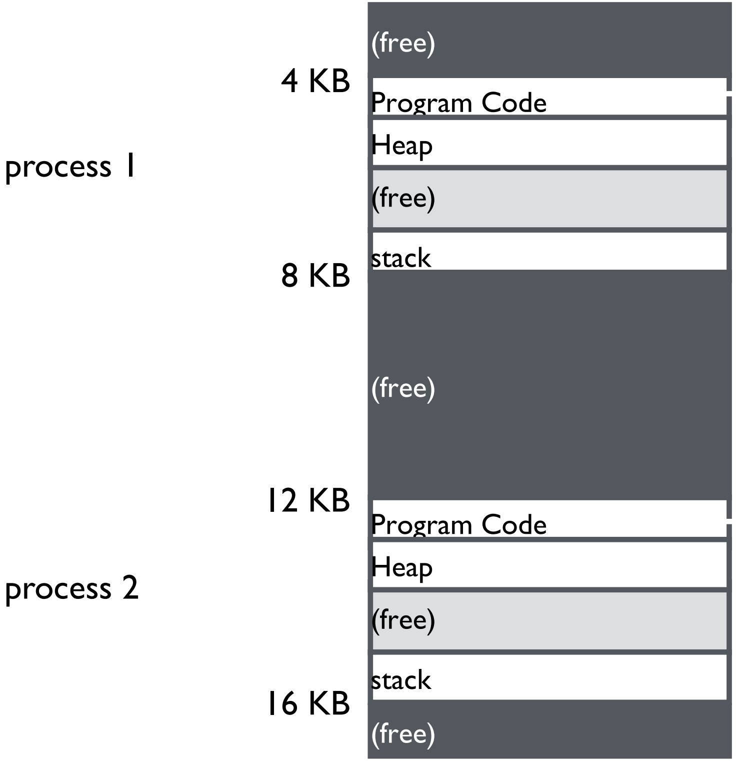
rewrite

```
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

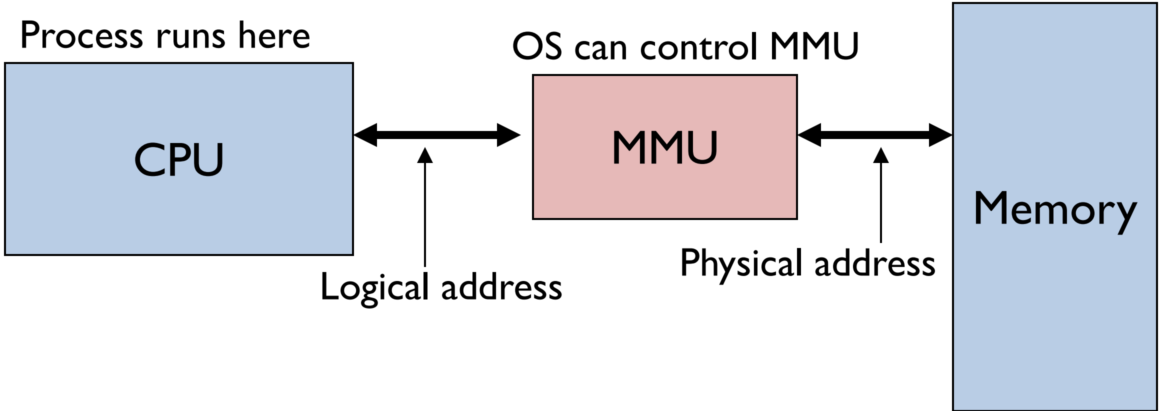
Goal: Protect processes from one another

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

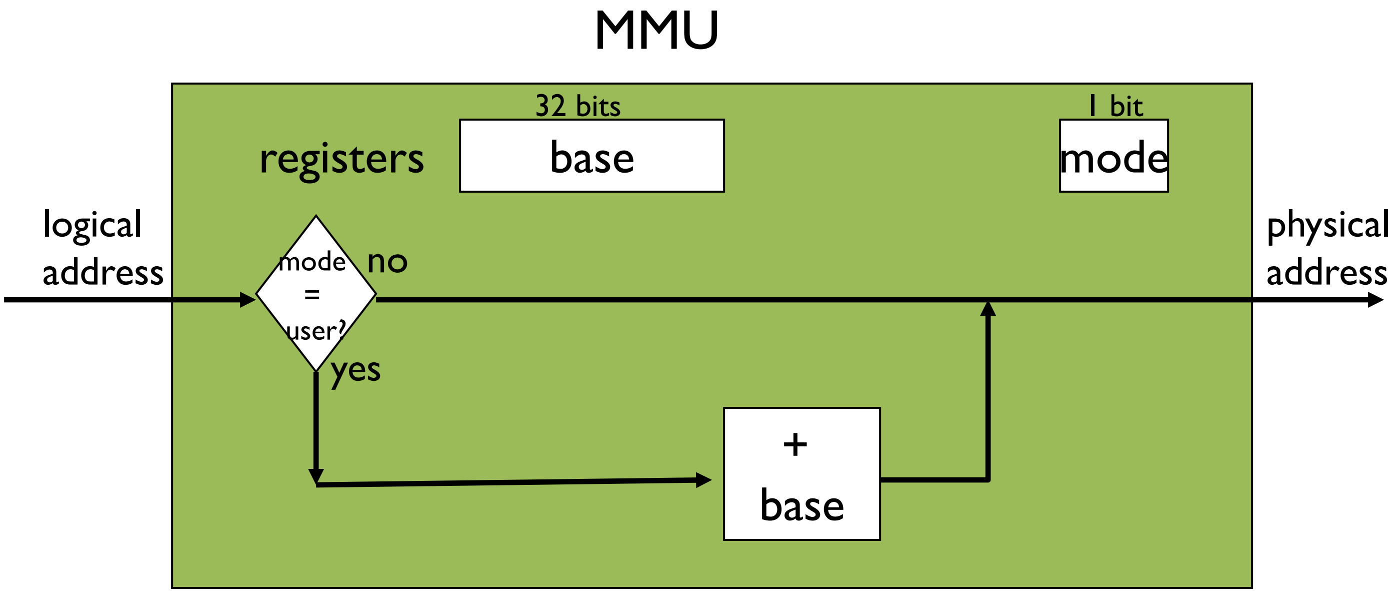
User mode: User processes run

- 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



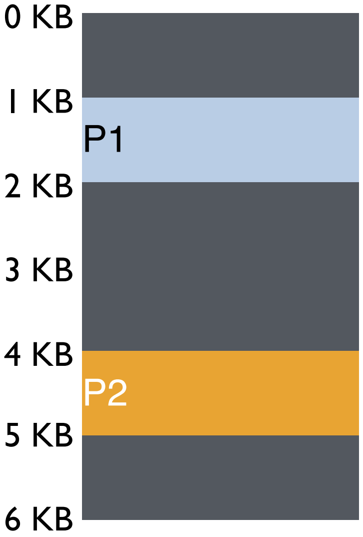
DYNAMIC RELOCATION WITH BASE REGISTER

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

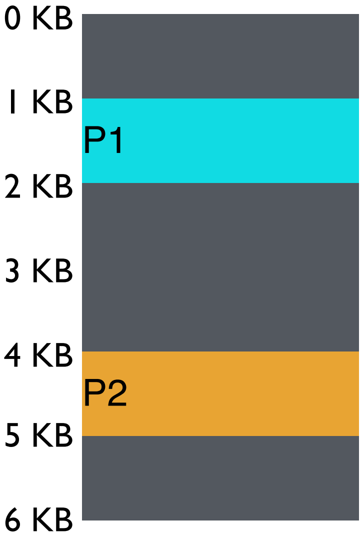
P1: load 100, R1

P2: load 100, R1

P2: load 1000, R1

P1: load 100, R1

VISUAL EXAMPLE OF DYNAMIC RELOCATION: BASE REGISTER

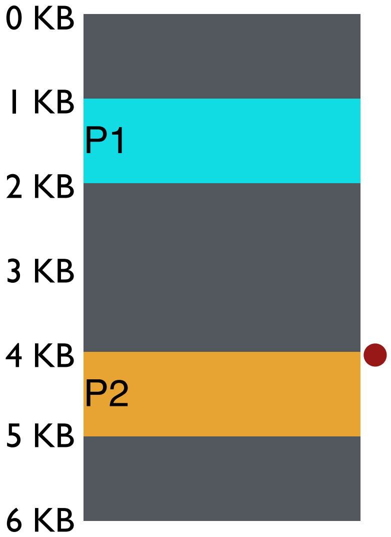


Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5096, R1
P1: load 1000, R1	load 2024, R1

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

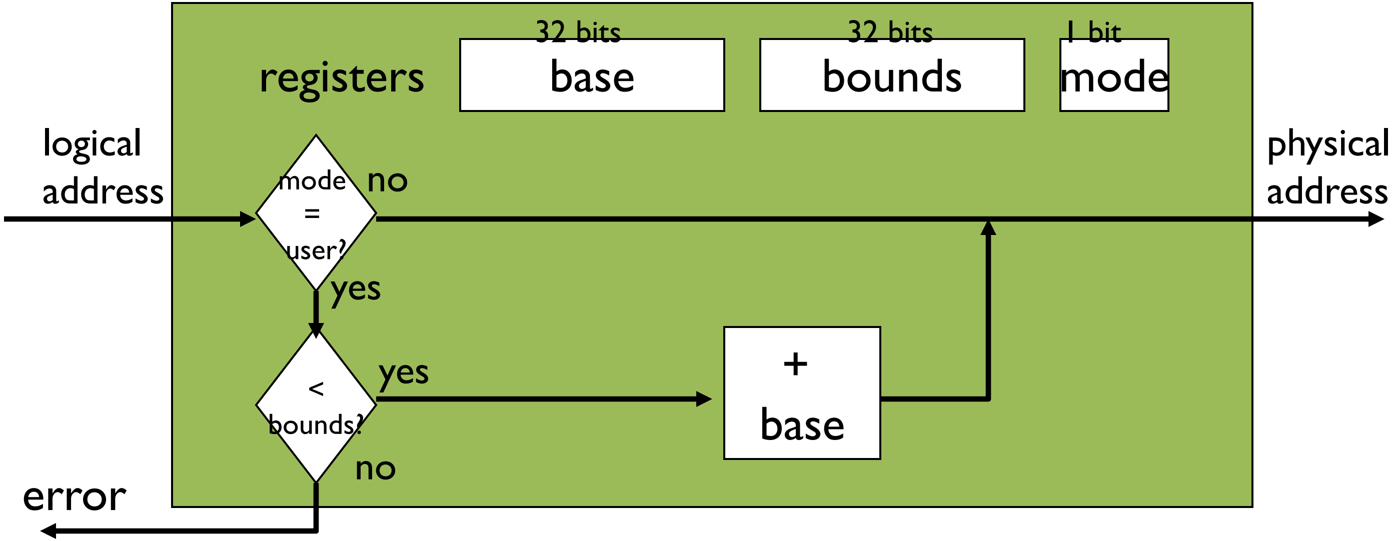
- Sometimes defined as largest physical address (base + size)

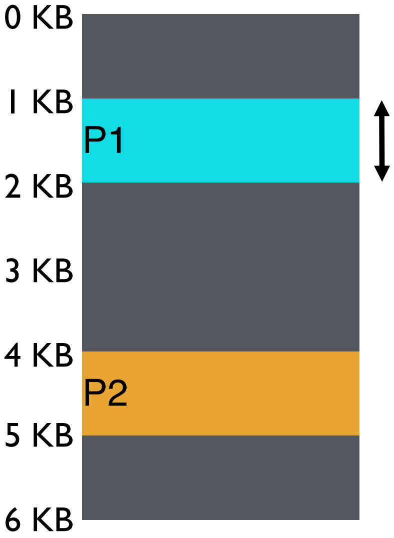
OS kills process if process loads/stores beyond bounds

IMPLEMENTATION OF BASE+BOUNDS

Translation on every memory access of user process

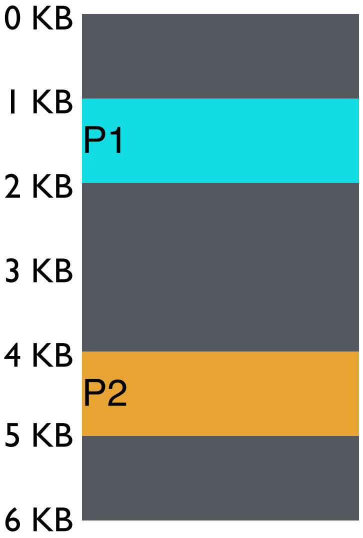
- MMU compares logical address to bounds register
  - if logical address is greater, then generate error
- MMU adds base register to logical address to form physical address





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

Advantages

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

Fast: Add and compare in parallel

Disadvantages

- Each process must be allocated contiguously in physical memory

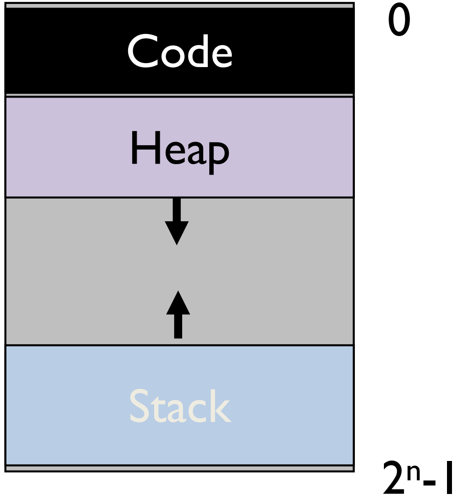
  Must allocate memory that may not be used by process
- No partial sharing: Cannot share parts of address space

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 2: Out now, due Feb 5th

Next week: Virtual memory segmentation, paging and more!