Hello!

VIRTUALIZATION: CPU TO MEMORY

Shivaram Venkataraman

CS 537, Fall 2024

ADMINISTRIVIA

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

- Project 2 is out, due September 24th

- Midterm Oct 15th evening after 5:45 pm
  Nov 7th
  ↳ Alternate exam request

AGENDA / LEARNING OUTCOMES

CPU virtualization

- Recap of scheduling policies

Memory virtualization → ~ 3-4 lectures

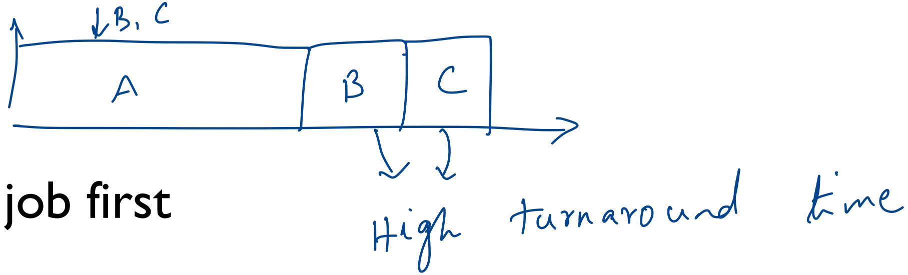
- What is the need for memory virtualization?
- How to virtualize memory?

RECAP: CPU VIRTUALIZATION

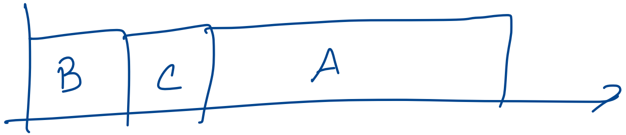
RECAP: METRICS → POLICIES

Turnaround time = *completion_time - arrival_time*

FIFO: First come, first served



SJF: Shortest job first

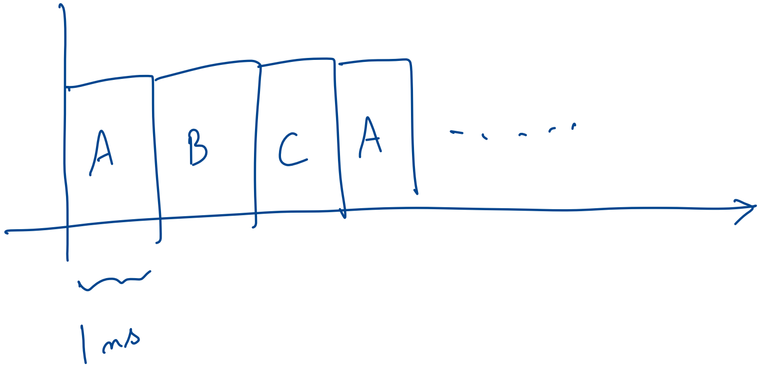


SCTF

RECAP: METRICS → POLICIES

Response time = *first_run_time - arrival_time* → Interactive

- Pre-emptive scheduling
- RR: Round robin with time slice
- Minimizes response time but could increase turnaround? Trade-offs



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
- "batch" programs care about turnaround time

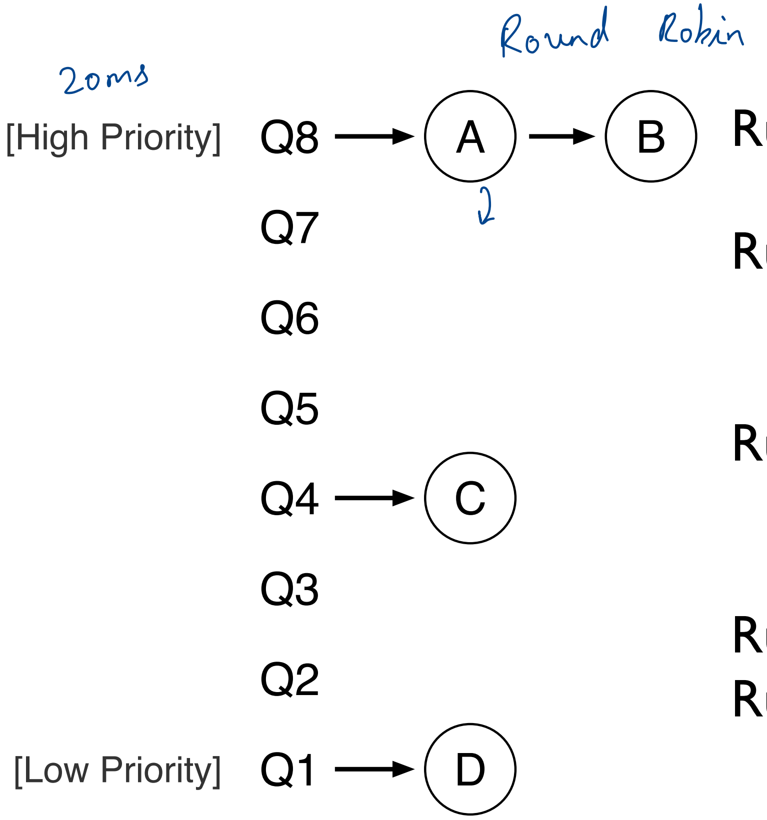
Approach:

Multiple levels of round-robin

Each level has higher priority than lower level

Can preempt them

RECAP: MULTI-LEVEL FEEDBACK QUEUE



Rules for MLFQ

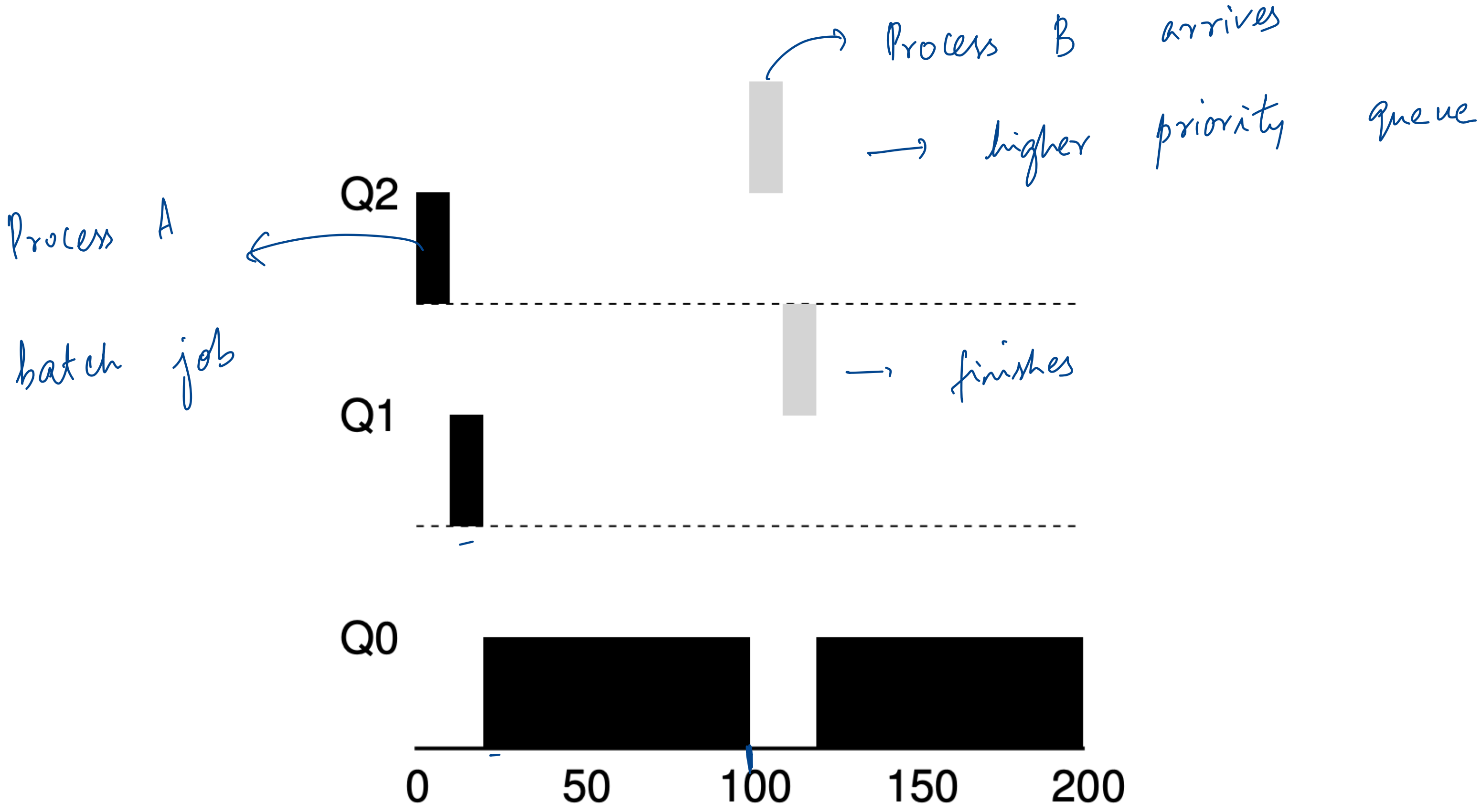
Rule 1: If priority(A) > Priority(B)
A runs

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

Rule 3: Processes start at top priority

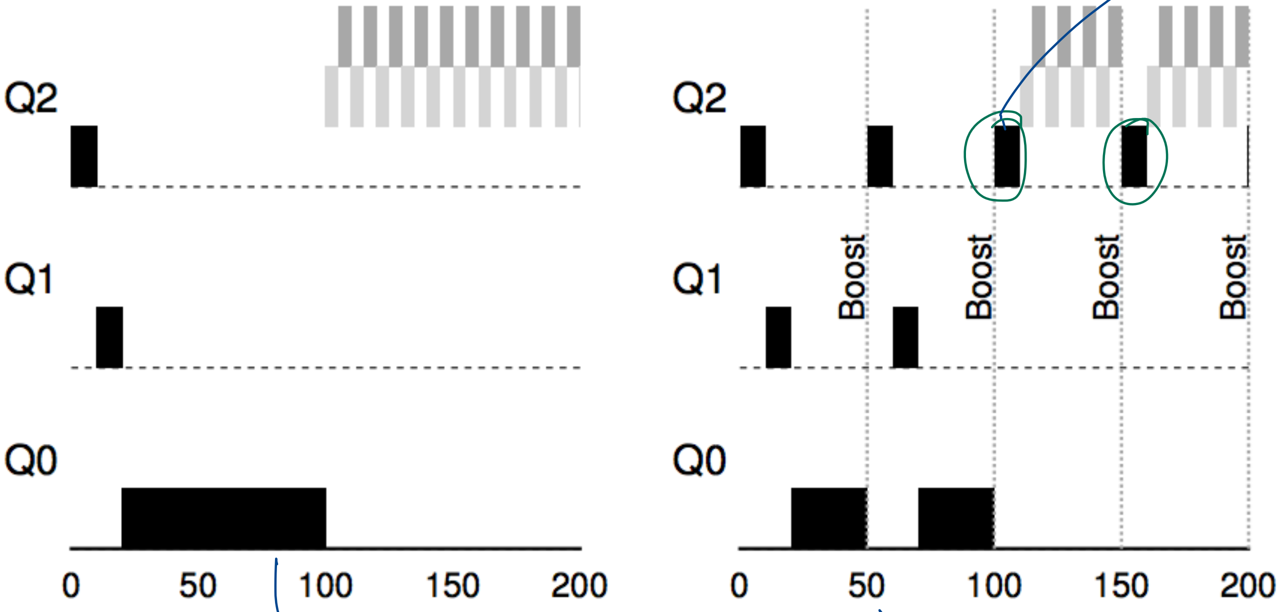
Rule 4: If job uses whole slice, demote process.
If not stay at level

INTERACTIVE PROCESS JOINS



AVOID STARVATION

avoided starvation for process A



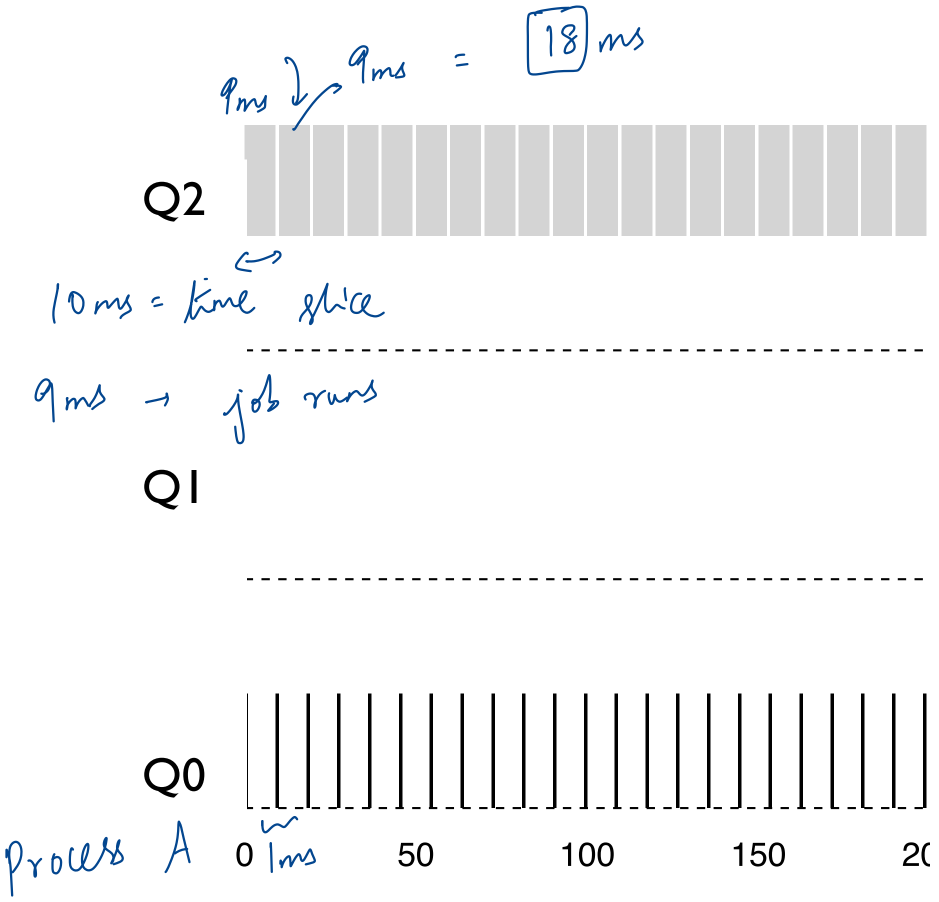
Priority Boost!

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

does not get resources for a long time

50s

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

OTHER SCHEDULERS: FAIRNESS?

New metric: Fairness!

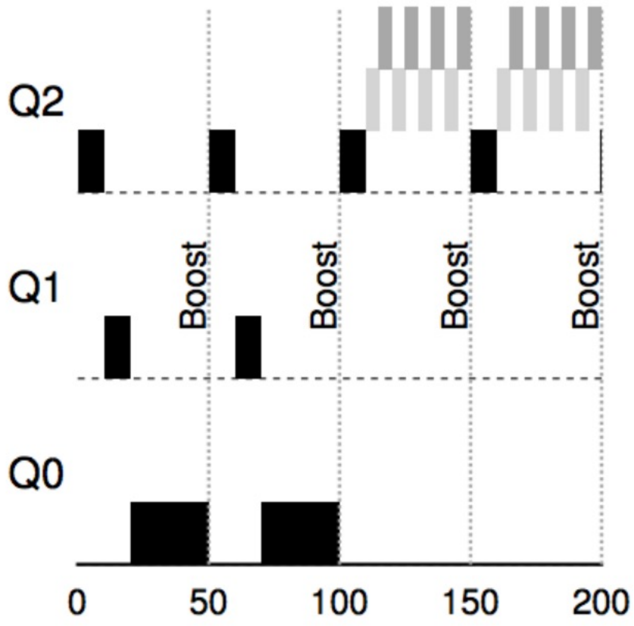
Metrics so far: turn around time, response time.

3 users; each get 1/3rd of CPU

no matter how long they run for

= 33 s out of 100s

Is MLFQ fair?



CPU SUMMARY

Mechanism

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

Policy

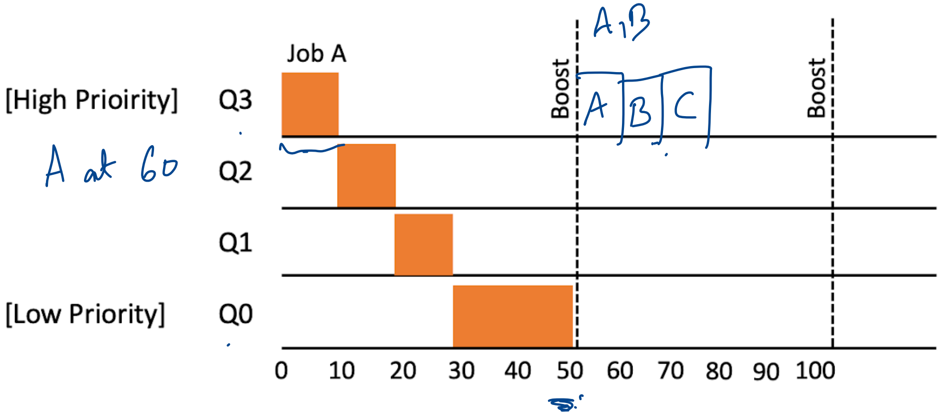
- Metrics: turnaround time, response time
- Balance using MLFQ
- (More scheduling in Multi-CPU)

QUIZ 3

https://tinyurl.com/cs537-fa24-q3

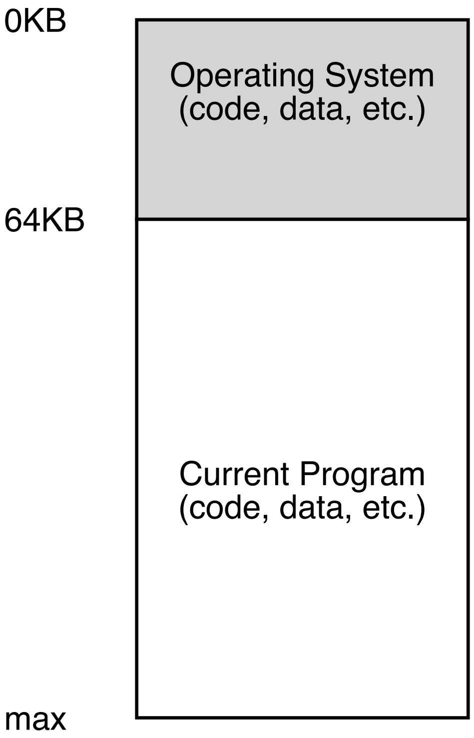
Jobs	Runtime	Arrival Time
Job A	100	0
Job B	10	50

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



VIRTUALIZING MEMORY

BACK IN THE DAY...



Uniprogramming: One process runs at a time

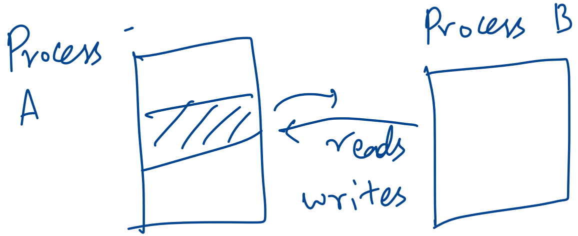
MULTIPROGRAMMING GOALS

Transparency: Process is unaware of sharing → Simplify program

Protection: Cannot corrupt OS or other process memory

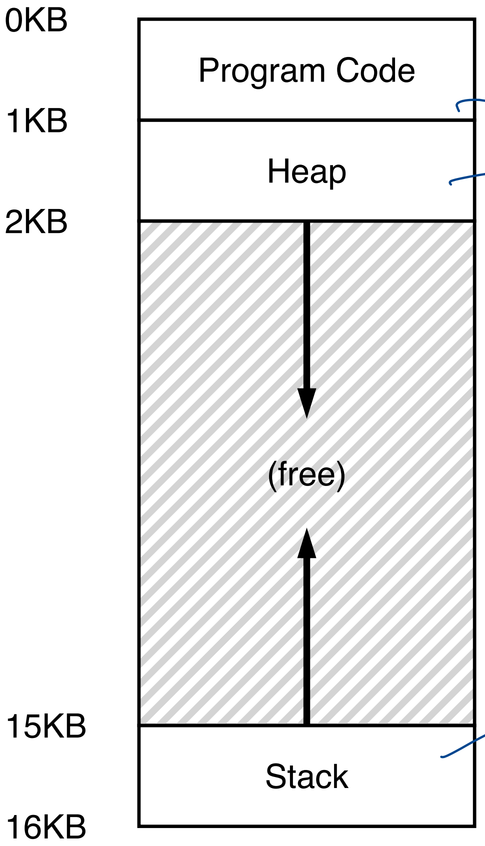
Efficiency: Do not waste memory or slow down processes

Sharing: Enable sharing between cooperating processes



ABSTRACTION: ADDRESS SPACE

Virtual addresses



Programs see memory

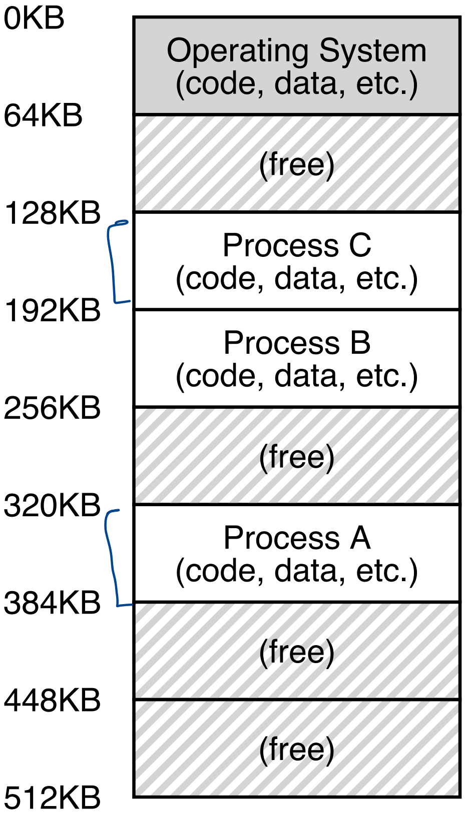
dynamic memory allocations

instructions that constitute program.

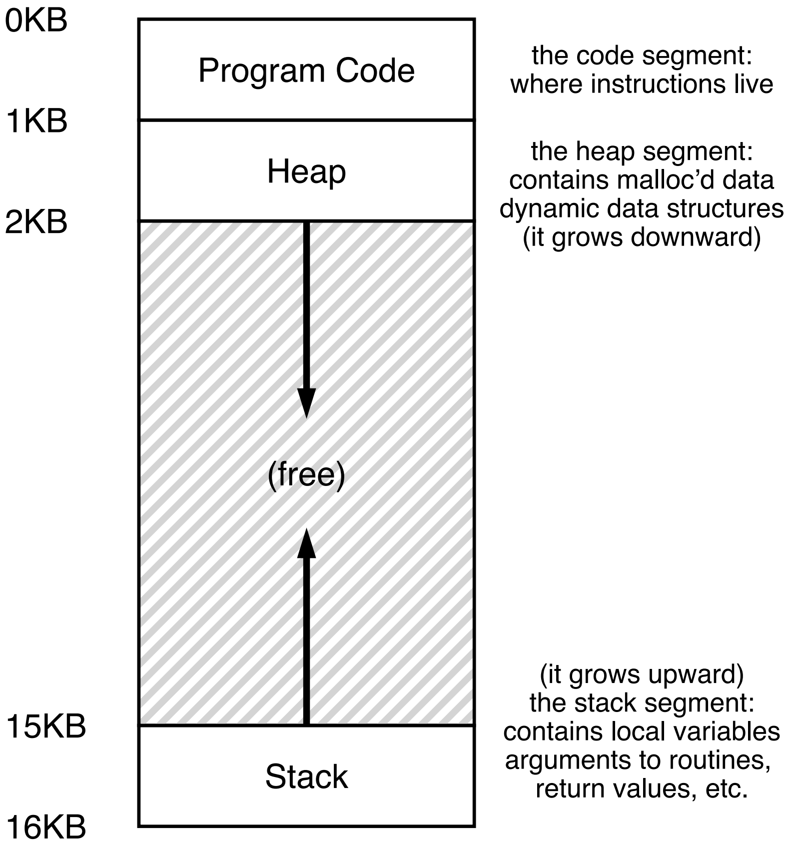
static data

local variables arguments,

Physical addresses



WHAT IS IN ADDRESS SPACE?



Static: Code and some global variables

Dynamic: Stack and Heap

ASIDE: HOW TO CREATE A PROCESS?

Unix-like OS use *fork()*

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

 Make copy of code, data, stack etc.

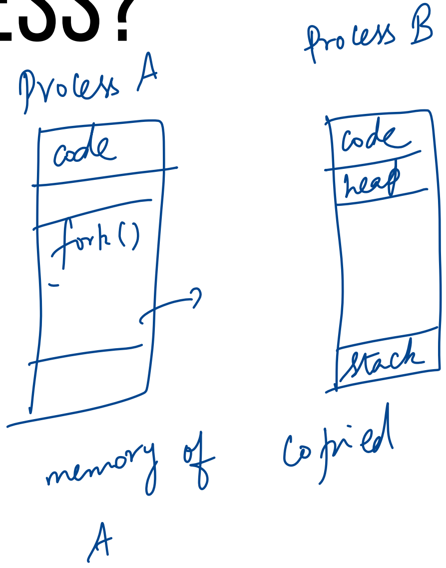
 Add new process to ready list → scheduled

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

→ used to launch a new file as a process

Advantages: Flexible, clean, simple

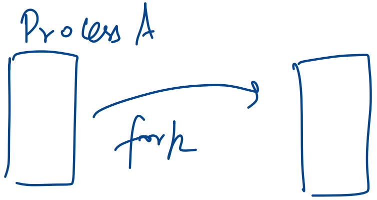
Disadvantages: Wasteful to perform copy and overwrite of memory



FORK EXAMPLE

```
int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
            rc, (int) getpid());
    }
    return 0;
}
```

5



different in parent & child

Process B

Process A

ostep - code

STACK ORGANIZATION

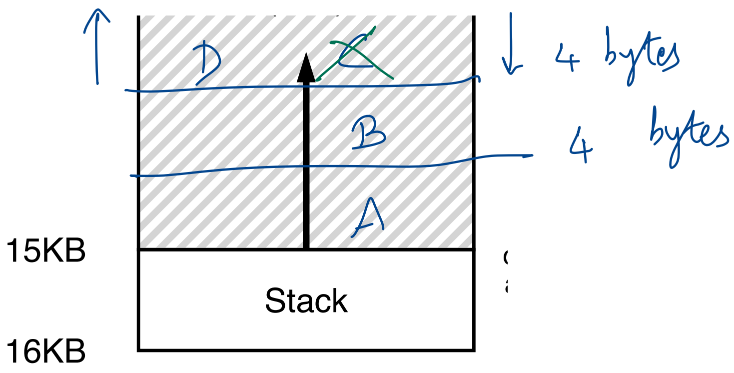
```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```

Pointer between allocated and free space

- Allocate: Increment pointer
- Free: Decrement pointer

No fragmentation!

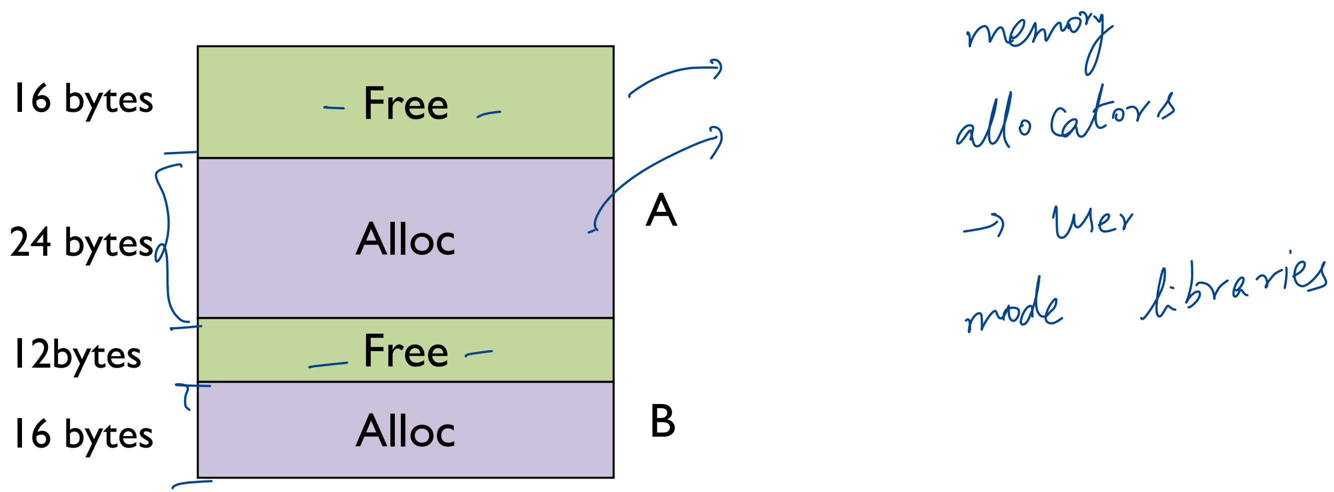
Last In First Out



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



STACK OR HEAP?

```
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	code
main	code
y	stack
z	stack
*z	heap

MEMORY ACCESS

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

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

Instructions = code region

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

↳ Fetch

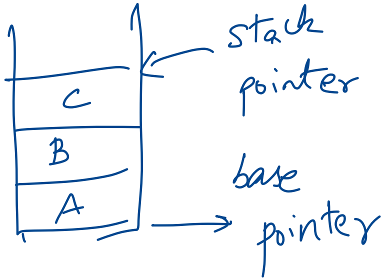
%**rbp** is the base pointer:
points to base of current stack frame

→ Fetch the instruction 0x10

→ Fetch %rbp + 0x8 (stack)

→ Fetch 0x13, 0x19

→ Store %rbp + 0x8



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

HOW TO VIRTUALIZE MEMORY

0x431 printf

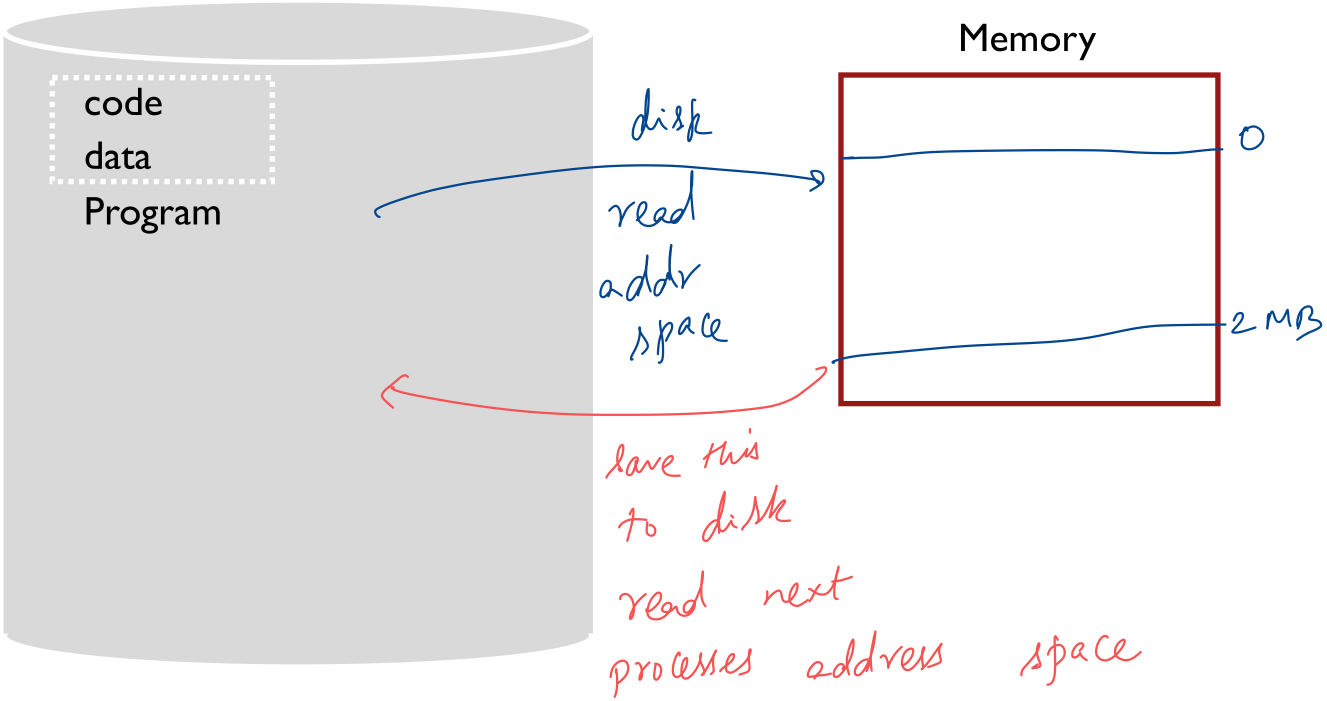
Problem: How to run multiple processes simultaneously?

Addresses are "hardcoded" into process binaries → call 0x431

How to avoid collisions?

Possible Solutions for Mechanisms (covered today):

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



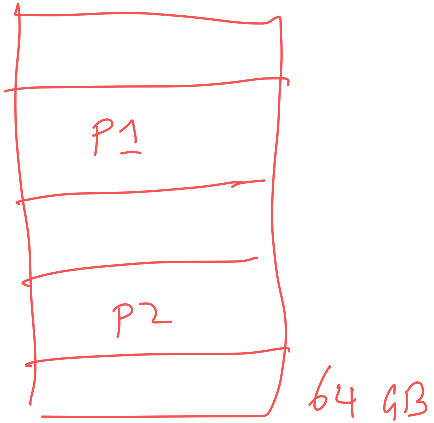
TIME SHARE MEMORY: EXAMPLE

PROBLEMS WITH TIME SHARING?

Ridiculously poor performance

Better Alternative: space sharing!

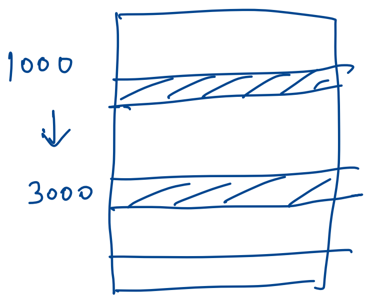
- At same time, space of memory is divided across processes
- Remainder of solutions all use space sharing



2) STATIC RELOCATION

No Protection

↳ Process can address another process memory



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

Call 0x431

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

rewrite →

Add 1000 to every address

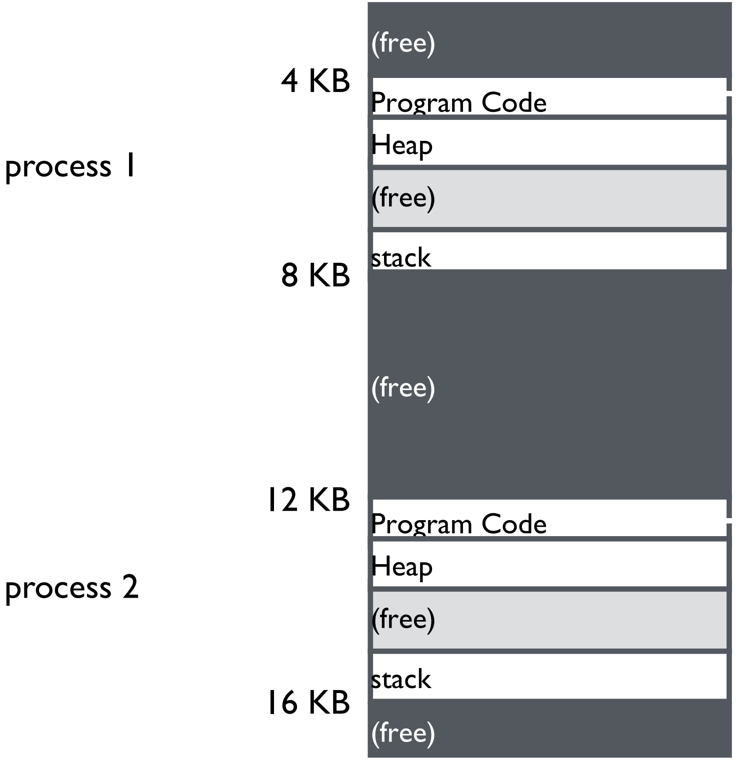
```
0x1010: movl  0x8(%rbp), %edi
0x1013: addl  $0x3, %edi
0x1019: movl  %edi, 0x8(%rbp)
```

rewrite →

Adding 3000 to every address

```
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

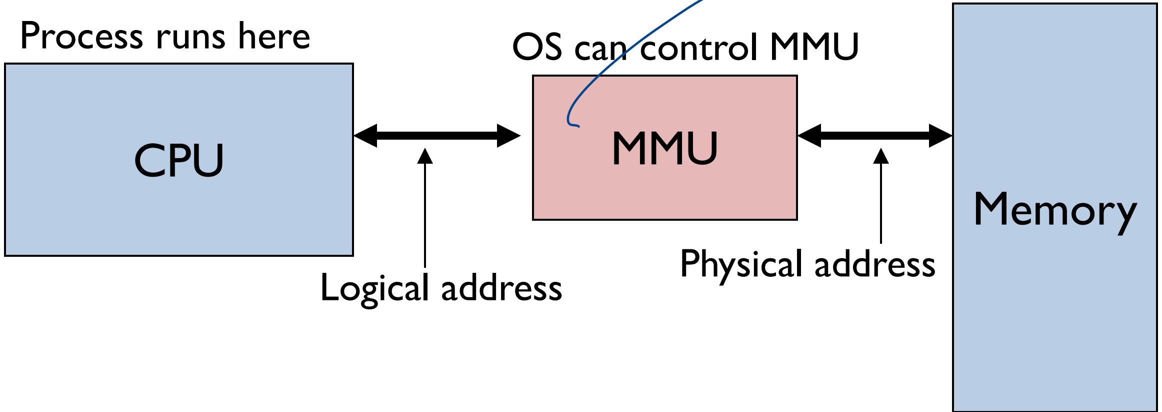
- Memory Management Unit (MMU)

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

address translation

check



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

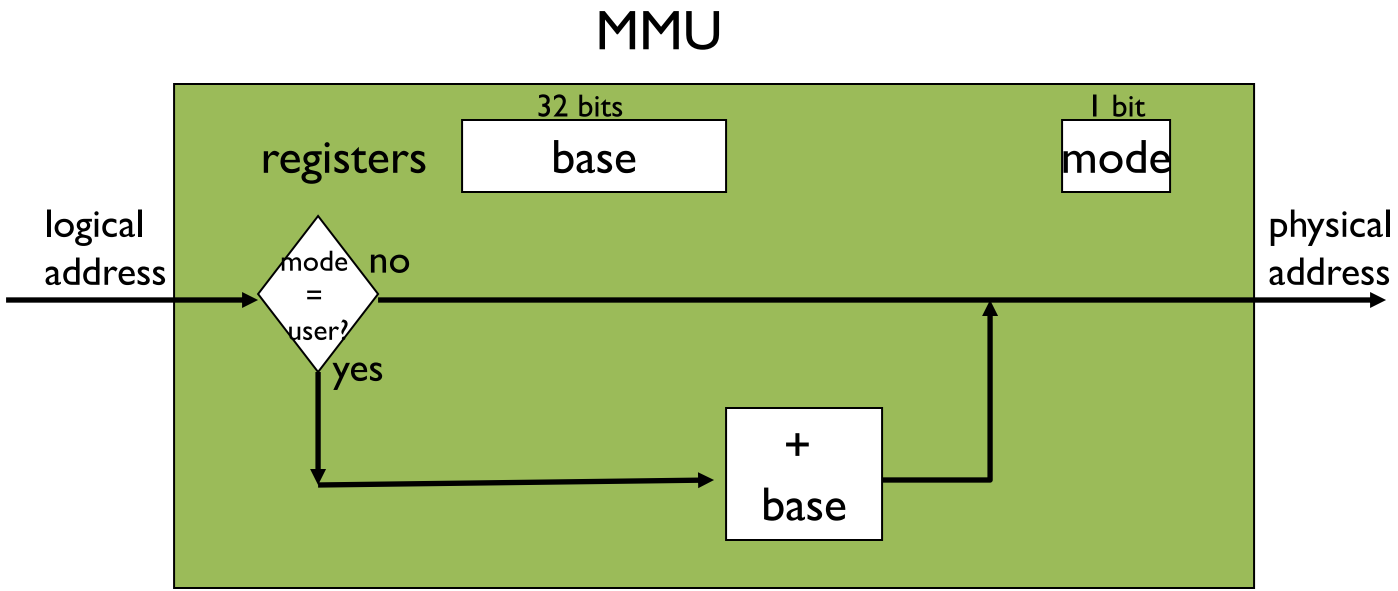
User mode: User processes run

- 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



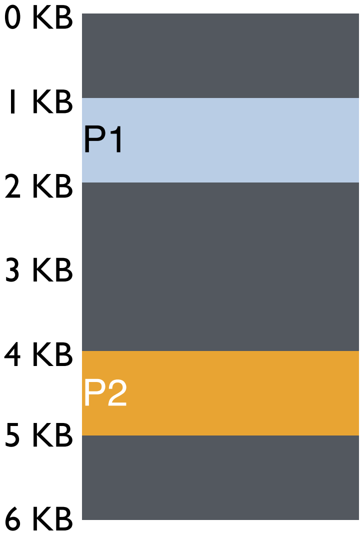
DYNAMIC RELOCATION WITH BASE REGISTER

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

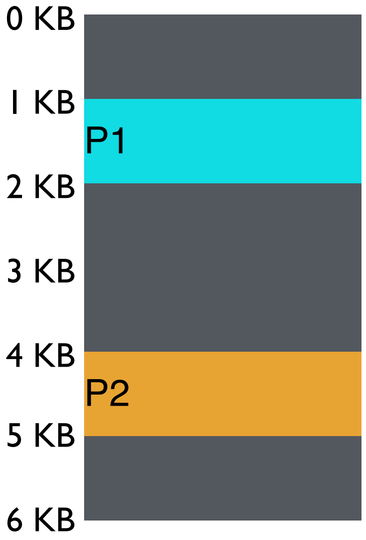
P1: load 100, R1

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

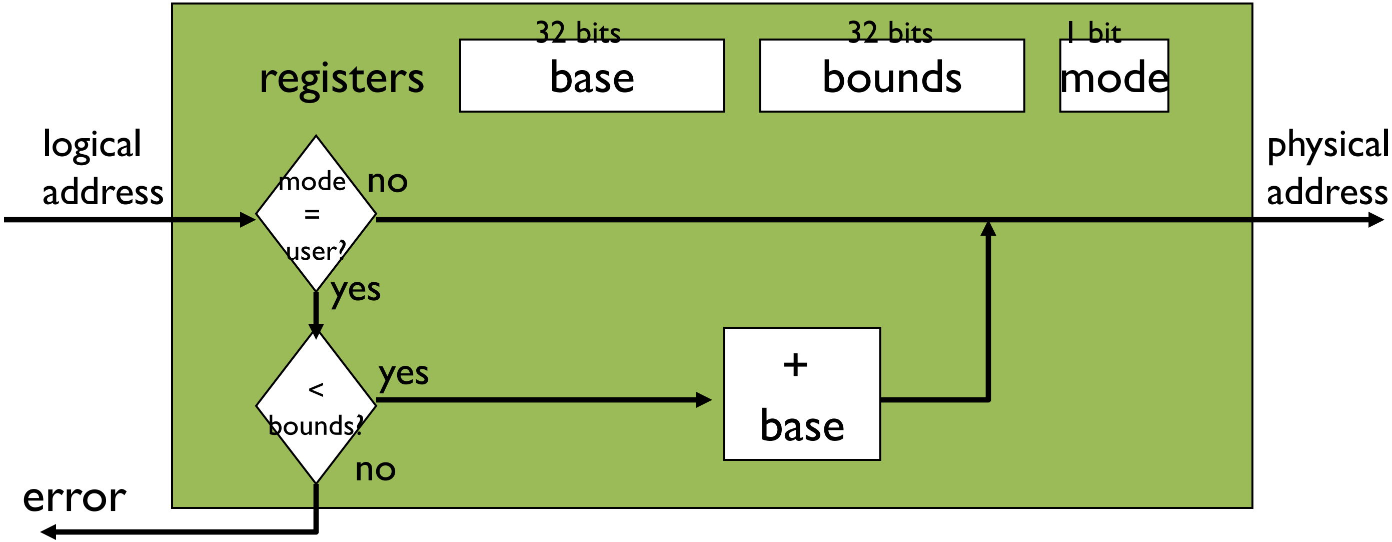
– Sometimes defined as largest physical address (base + size)

OS kills process if process loads/stores beyond bounds

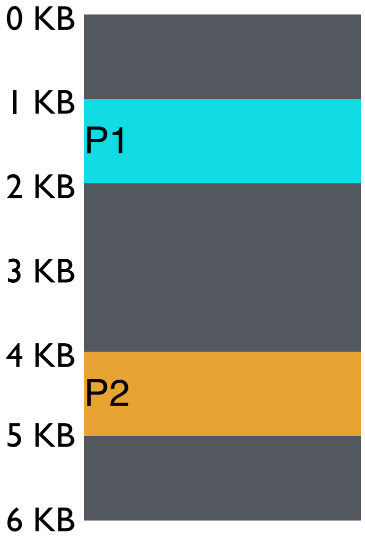
IMPLEMENTATION OF BASE+BOUNDS

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

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

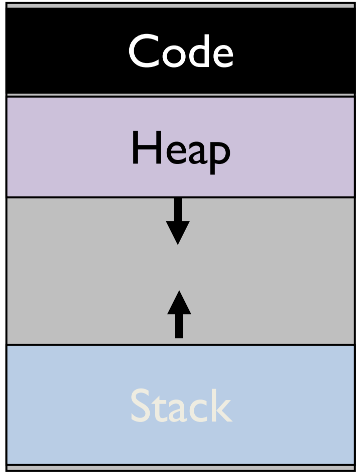
BASE AND BOUNDS DISADVANTAGES

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

0



2^n-1

NEXT STEPS

Project 2: Out now, due Sept 24th

Next class: Virtual memory segmentation, paging and more!