CS 423
Operating System Design: Virtualization: CPU & Memory
Feb 03

Ram Kesavan

Acknowledgement: Prof. Shivaram Venkataraman (Wisconsin)

Logistics

MP0: Due on 02/09 11:59pm CT

4Cr: UNIX paper summary (was) due 2pm CT today

This lecture:

- Finish up CPU virtualization
- Start memory virtualization - memory layout and translation

AGENDA / LEARNING OUTCOMES

CPU virtualization

- Recap scheduling policies
- Proportional share scheduling
- Brief look at Linux scheduler

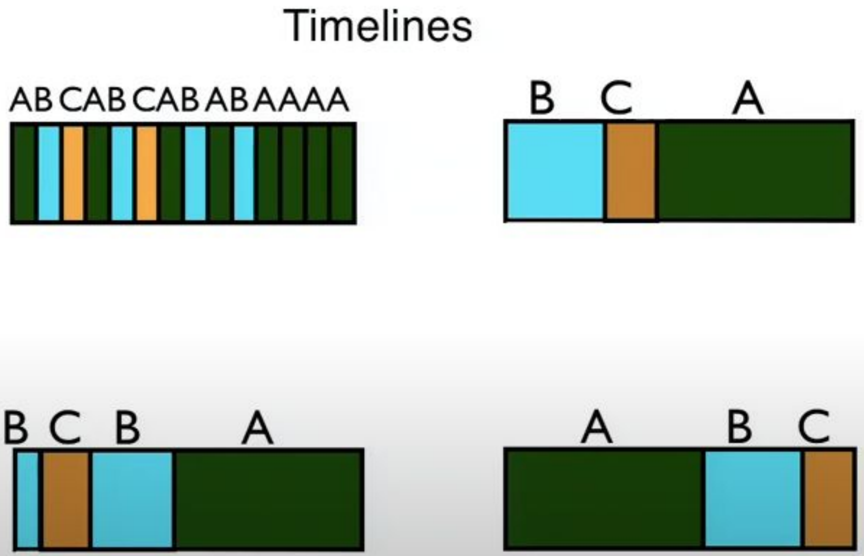
Memory virtualization

- What is the need for memory virtualization?
- Basics of virtualizing memory

RECAP: Scheduling Policies

Workload

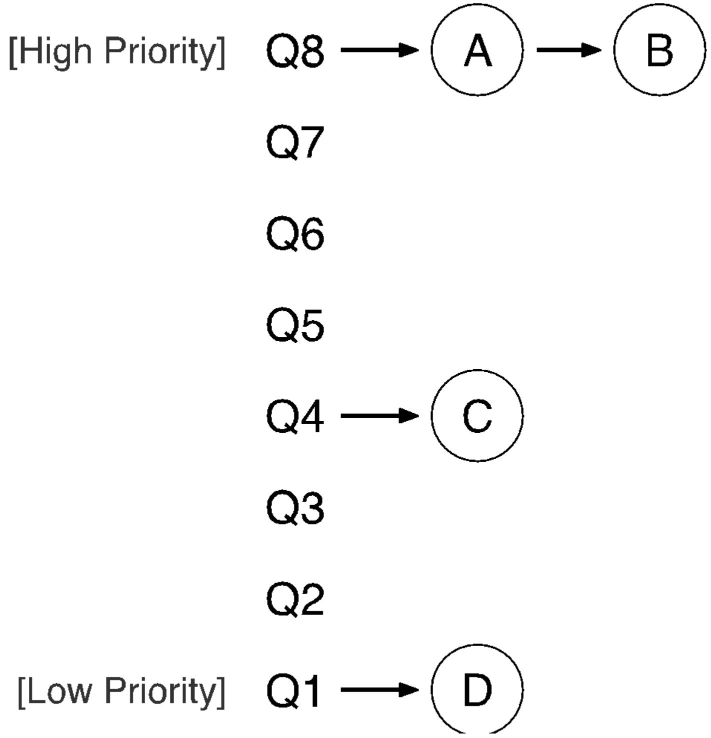
JOB	arrival	run
A	0	40
B	0	20
C	5	10



Schedulers:

- FIFO
- SJF
- STCF
- RR

RECAP: MLFQ



Rules for MLFQ

Rule 1: If pri(A) > pri(C), A runs

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

Rule 3: All jobs start at top pri

Rule 4: If job uses whole slice*, demote job.
Else, stay at level;
4a*time allotment

Rule 5: After time period S, boost all jobs top pri

Proportional Share in Scheduling

Metrics so far: turnaround time, response time

New metric: proportional-share

E.g., if Job A has paid 5x more price than Job B, then A is 5 times more likely to get the CPU than B

If both paid equally, A and B are equally likely to get the CPU

Lottery Scheduling: Probabilistically Proportional

A	10 tickets
B	20 tickets
C	30 tickets

Give each job tickets; #tickets based on priority

Conduct a lottery every period

Winner gets scheduled on CPU

Lottery: Generate a random number 0 <= r < 60; schedule:

A if 0 <= r < 10

B if 10 <= r < 30

C if 30 <= r < 60

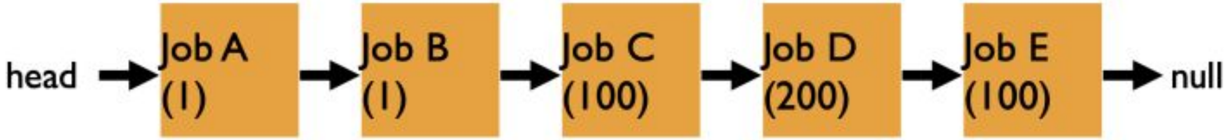
Probabilistic: but the longer they run, closer we get to the 1:2:3 proportioning

Implementing Lottery Scheduling

```
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
```

Who runs if winner is:

50
350
0



Lottery Scheduling (Misc)

A job can transfer its ticket to another

When is this useful?

If all jobs have the same #tickets, isn’t that just RR?

DEADLINE BASED SCHEDULING

Used in real-time systems (RTOS)

Each job gets a deadline

Classic policy: EDF (earliest deadline first)

    But works only when system is not overloaded

Eg: It's Mon, 3 HWs due T, W, and Thu and each takes 1.5 days

Linux CFS (Completely Fair Scheduler)

Linux scheduler has evolved to CFS

- multi-core & variety of applications

Each job accumulates *vruntime* whenever it runs

CFS picks job with lowest *vruntime*

CFS uses *sched_latency* to decide time slice per run

- time_slice = max(*min_granularity*, (*sched_latency*)/(#runnable-jobs))

Params: sched_latency = 48ms, min_granularity = 6ms

Eg. 1: If 3 jobs, what is each job’s time_slice?

Eg. 2: If 10 jobs, what is each job’s time_slice?

Linux CFS More

UNIX pri: -20 < *nice* <= 19 translates to a weight, which biases the math

$$time\_slice_k = max(min\_granularity, \frac{weight_k}{\sum_{i=0}^{n-1} weight_i} * sched\_latency)$$

$$vruntime_k += \frac{weight_0}{weight_k} * runtime_k$$

Runnable jobs kept in a (balanced) red-black tree, ordered by vruntime
All updates are O(log n)
- Picking job with lowest vruntime
- Adjusting vruntime of a job

2 Problems:
(1) Very low vruntime for a job that just became runnable
(2) What about multi core systems?

CPU VIRTUALIZATION SUMMARY

Mechanism

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

Policy

- Metrics: turnaround time, response time, proportional share
- Various policies: eg. MLFQ
- Linux scheduling

DIGRESSION: Common Design Pattern in Systems

A single struct lives in many structures at the same time.

Eg. The per-process *task_struct*

1. RunQ structure for scheduler
2. Lookup by PID
3. Parent<->child relationships

By using pointers

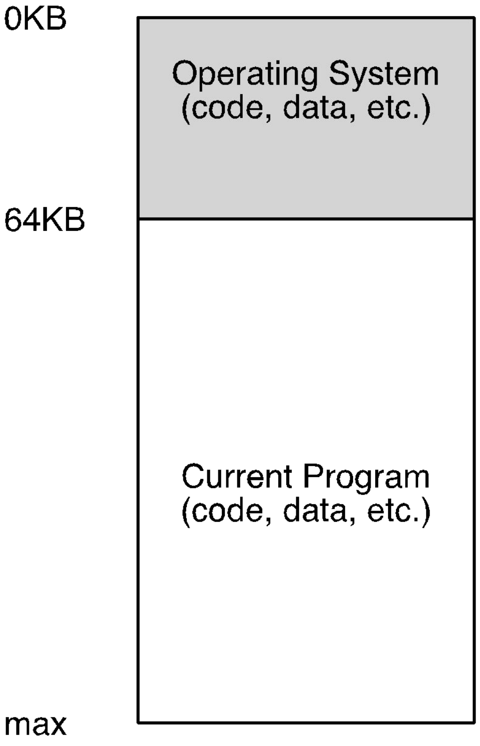
MEMORY VIRTUALIZATION

Create the illusion that each process has its own memory

Goals:

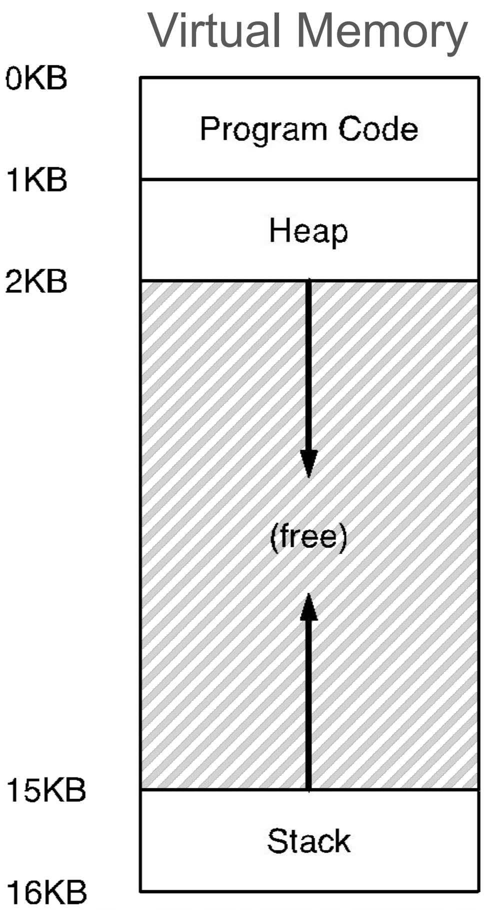
1. Sharing, but with…
2. …Transparency
3. Efficiency: minimize memory and CPU-cycles wastage
4. Protection: No process can read/write another’s memory

BACK IN THE DAY…



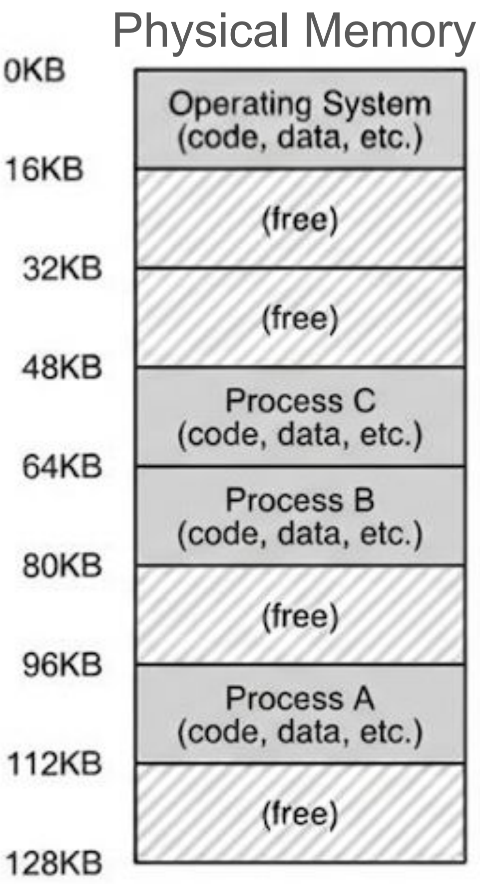
Early systems did not virtualize memory
Uniprogramming: One process at a time, with no protection

ABSTRACTION: ADDRESS SPACE

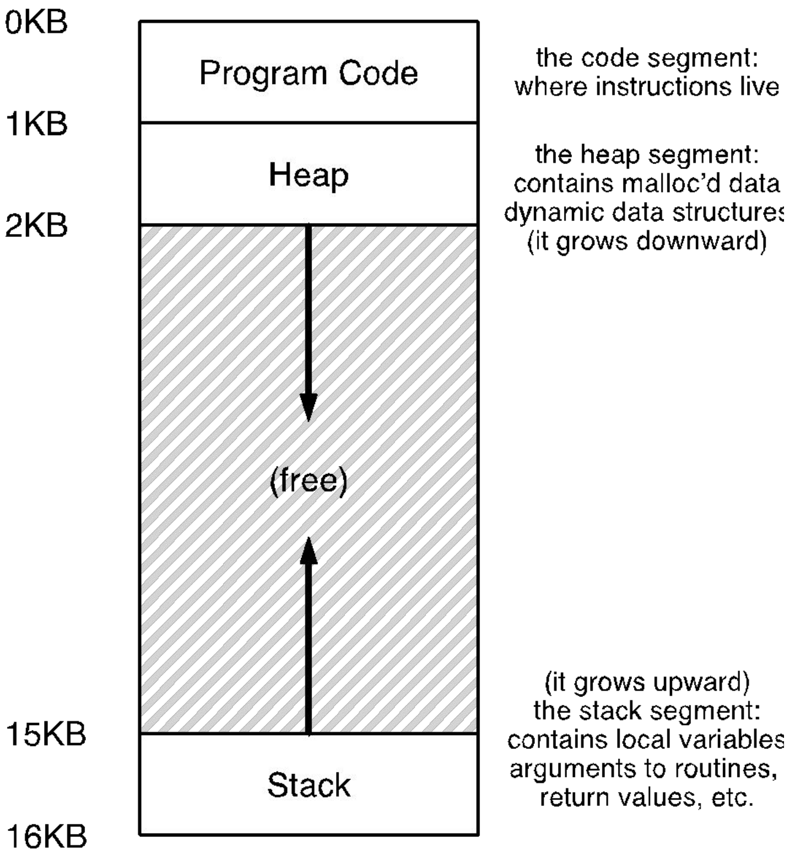


Virtual Address Space:
Each process has its own address range

OS provides that Illusion by mapping to physical memory



WHAT IS IN ADDRESS SPACE?



Static: Code and global variables
Dynamic: Heap & Stack

Q1: Why put stack and heap at the opposite ends?

Q2: What if the process has multiple threads?

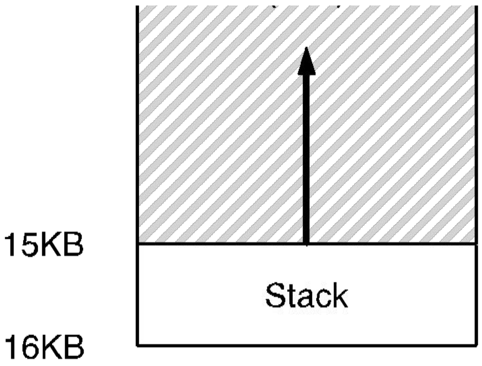
STACK ORGANIZATION

```
main()       : alloc(main)
 call A()    : alloc(A)
  call B()   : alloc(B)
  exit B()   : free(B)
  call C()   : alloc(C)
  exit C()   : free(C)
 exit A()    : free(A)
exit main(): free(main);
```

SP demarcates allocated from free space

- Call func: Increment pointer
- Return from func: Decrement pointer

No fragmentation!



Possible locations:
static data/code, stack, heap

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

Address	Location
x	
main	
y	
z	
*z	

Why do programmers need dynamic memory?

We don’t always know (at compile time) how much memory is needed?

Complex data structures: trees, graphs, lists, maps

- on-demand: Eg. *void *ptr = malloc(size(struct my_struct))*
- malloc returns **contiguous** space
- pass around to other functions
- *free(ptr)* when it’s not needed

Key observation:

1. Each allocation is of custom/specified size
2. But, the *free* function somehow knows what that size it!
3. If “leaked”, all allocations freed when process eventually dies

HEAP ORGANIZATION

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

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

Adv: works for all data structures

Disadv: Can result in fragmentation

How to allocate 20 (contiguous) bytes?

What is OS's role in managing the heap?

OS gives out big chunks of free memory & library manages the individual allocations and frees

Interesting Topic: Slab Allocators

MEMORY ACCESS

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

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

```
0x10:  movl   0x8(%rbp), %edi
0x18:  addl   $0x3, %edi
0x20:  movl   %edi, 0x8(%rbp)
```

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

MEMORY ACCESS

Initial %rip = 0x10
%rbp = 0x200

```
0x10:   movl   0x8(%rbp), %edi
0x18:   addl   $0x3, %edi
0x20:   movl   %edi, 0x8(%rbp)
```

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

%rip is instruction pointer (or program counter)

How many memory accesses?

To what addresses?

Chat with neighbors for 2 mins.

HOW TO VIRTUALIZE MEMORY

Problem: How to run multiple processes simultaneously?

Compiler “hardcodes” the VM addresses into the binaries it generates

How to avoid collisions?

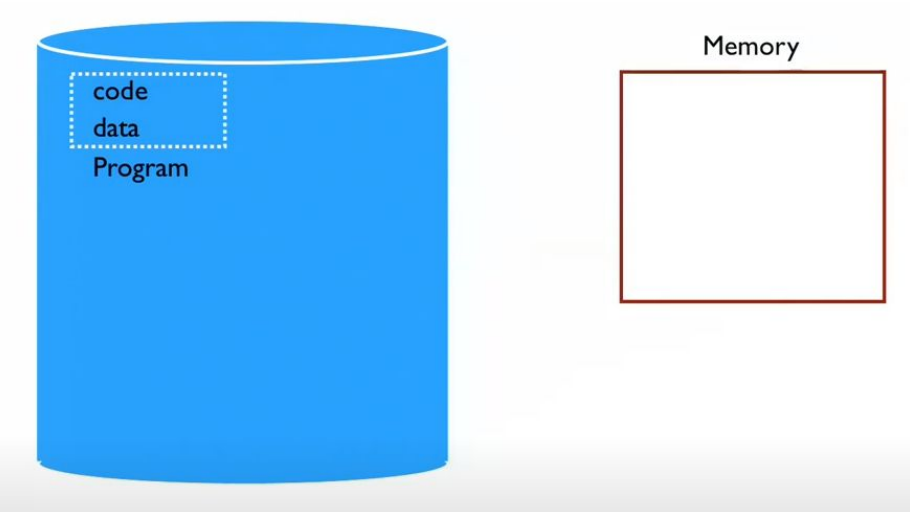
Possible Solutions for Mechanisms (in today’s class):

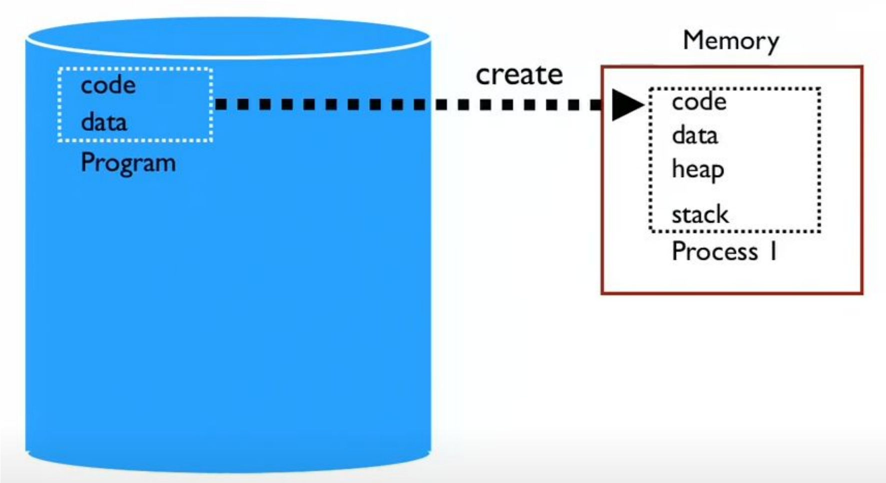
Time Sharing, Static Relocation, Base, Base+Bounds

Unrealistic assumptions:

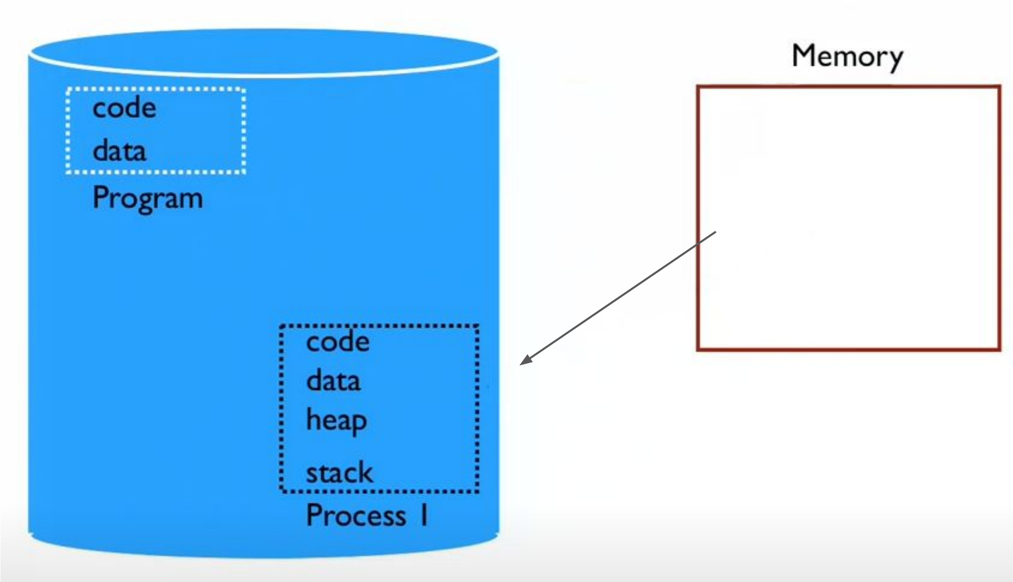
1. Each process gets the same sized VM address space, which is smaller than the physical memory on the machine
2. VM address space can be mapped contiguously in physical memory

1. TIME SHARE MEMORY





Time Sharing Memory



PROBLEMS WITH TIME SHARING?

Ridiculously poor performance

Better Alternative: space sharing!

- Divide physical memory across processes
- Rest of solutions all use space sharing

LOADER: STATIC RELOCATION

OS rewrites each program before loading it into memory (as a process)

Each rewrite for a process uses different addresses and pointers

Need to change jumps, loads of static data

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

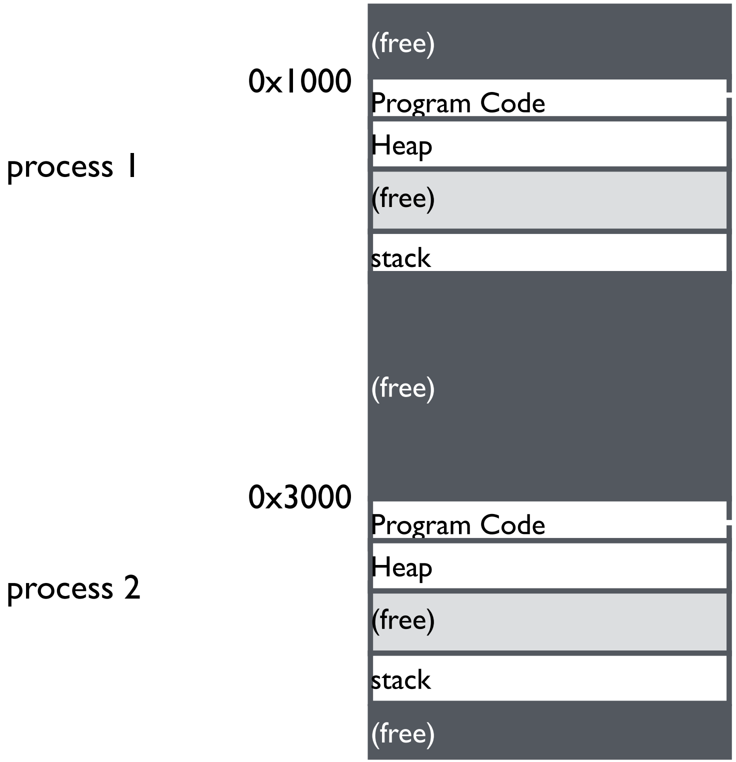
rewrite

```
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)
```

DYNAMIC RELOCATION

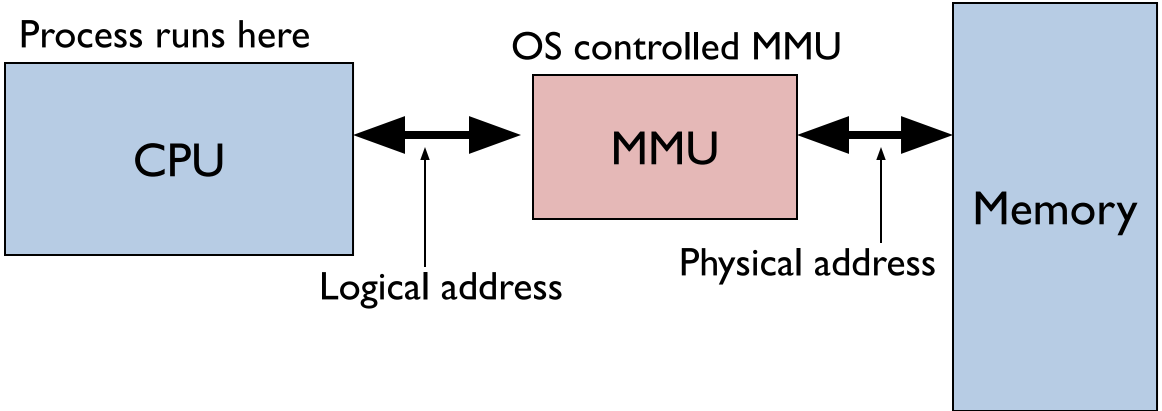
Goal: Protect processes from one another

Requires hardware support

- Memory Management Unit (MMU)

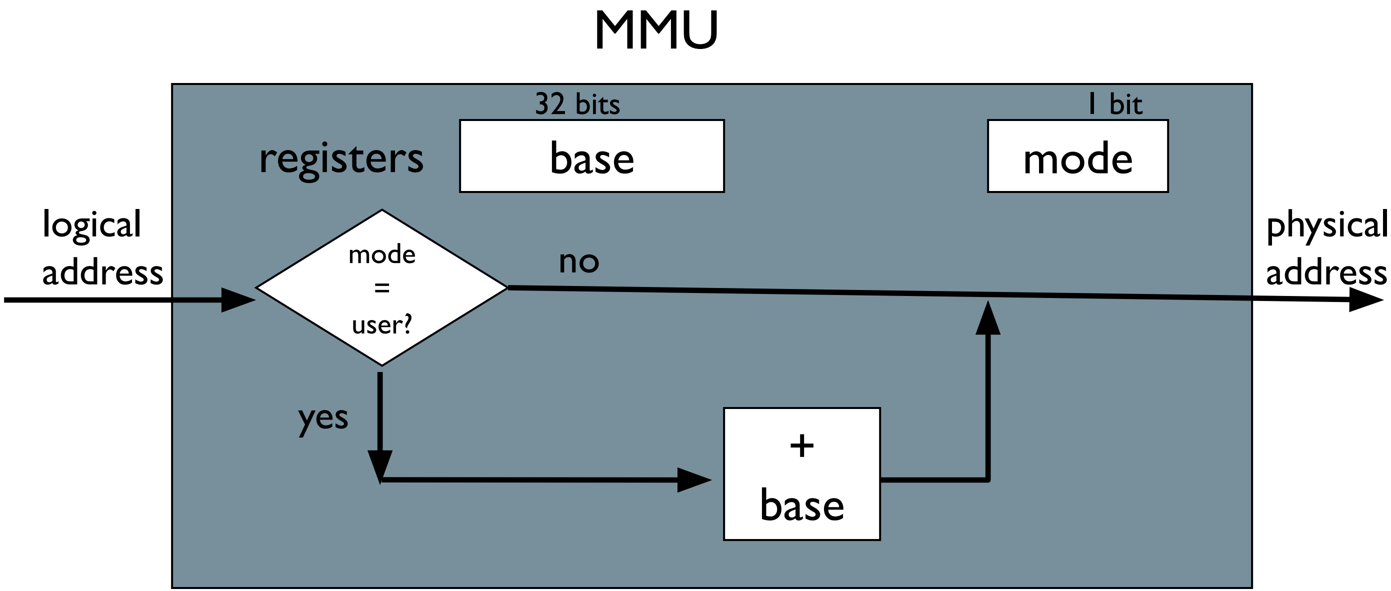
MMU dynamically changes process address at every memory reference

- Process generates logical or virtual address (in its address space)
- MMU converts to physical or real address



Dynamic Relocation: BASE REG

All VM addresses in compiled binary are offsets from zero
HW translates every VM address for a user process
MMU adds base register to VM address => physical address



PER-PROCESS BASE REG

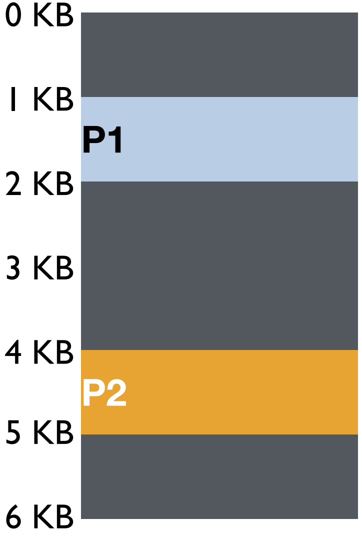
Each process has a unique value in its base register

MMU mapping of VM->PM requires only an addition

- Fast & efficient

Also, OS can relocate a process!

- Find free contiguous space in physical memory
- Don’t let the process run for a bit
- Copy the process entirely to the new location
- Change the process’s base register



Virtual

P1: load 100, R1

P2: load 100, R1

P2: load 1000, R2

P1: load 1000, R2

Assume addresses are in decimal format

Visual Example of DYNAMIC RELOCATION

DYNAMIC with BASE+BOUNDS

Idea: limit the address space with a bounds register

Base Reg: smallest physical addr (or starting location)

Bounds Reg: size of this process's virtual address space

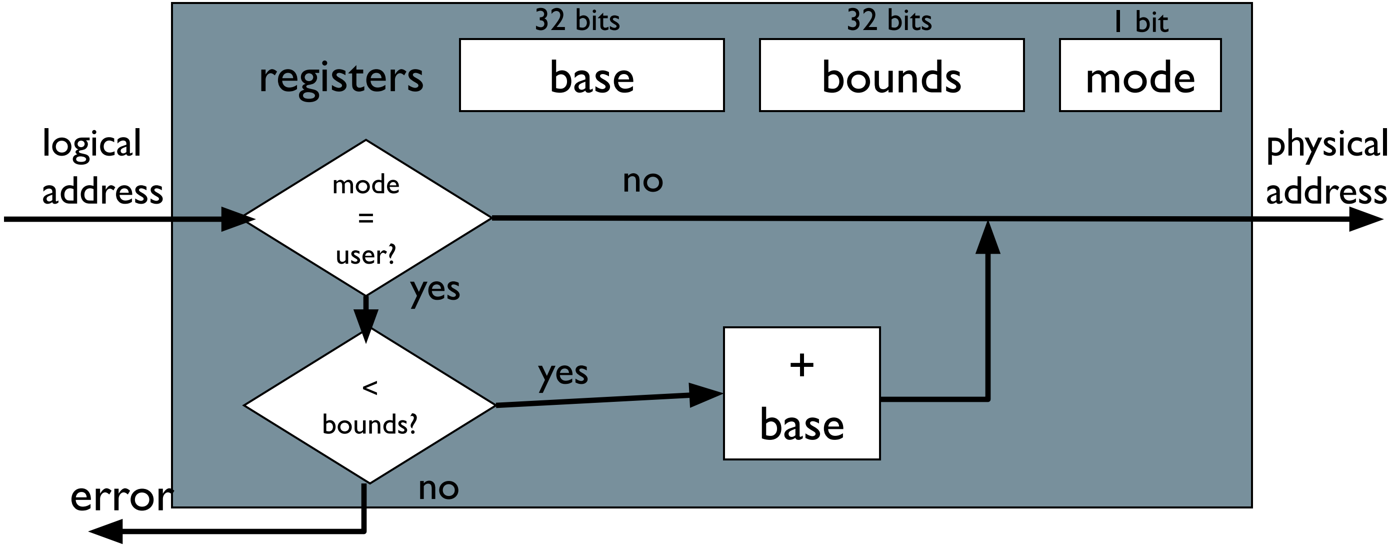
- The largest physical address (base + size)

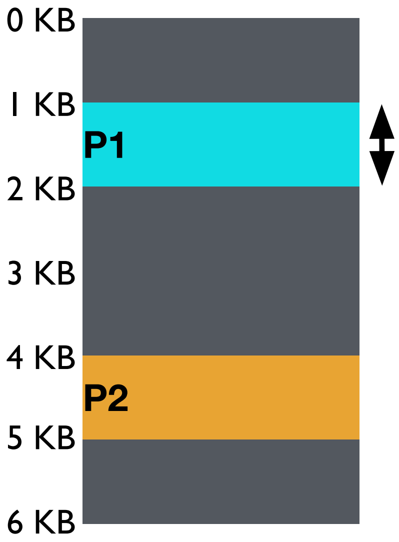
OS can kill process if it loads/stores beyond bounds

IMPLEMENTATION OF BASE+BOUNDS

Translate on every memory access in user mode

- MMU compares VM address to bounds register
  - If VM address is greater, then errors
- Else, MMU adds base reg to VM address => physical address





base register = 1024

bounds register = 1024

Does the bounds reg contain a physical or virtual address?

HW SUPPORT FOR DYNAMIC RELOCATION

Restricted (user) mode: MMU

- performs translation of logical address to physical address

Privileged (kernel) mode: OS can

- To manipulate contents of MMU
- Needed during context switching: load new base+bounds regs
- Allows OS to access all of physical memory

Managing Processes with Base and Bounds

For 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

Base and Bounds Advantages

Provides protection (both read and write) across address spaces

Supports dynamic relocation

- Can place process initially at a physical address…
- …but, can move it to a different physical address

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

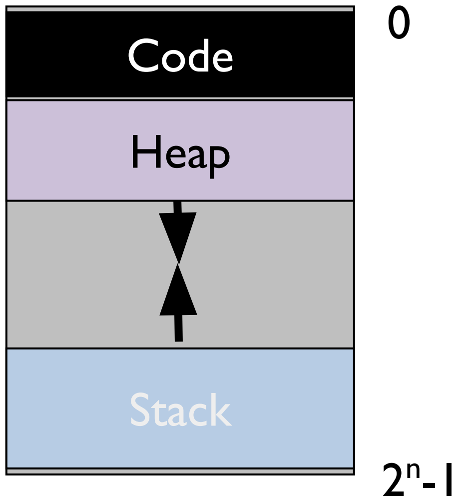
Fast: Add and compare in parallel

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 LECTURES…

Relax assumptions:

1. VM address space < physical memory address space
2. VM address space is mapped contiguously in physical memory

Segmentation

Paging

Segmentation + Paging

Multi-level Page Tables

TLBs