VIRTUALIZATION: CPU TO MEMORY

Yuvraj Patel

CS 537

Annoucements

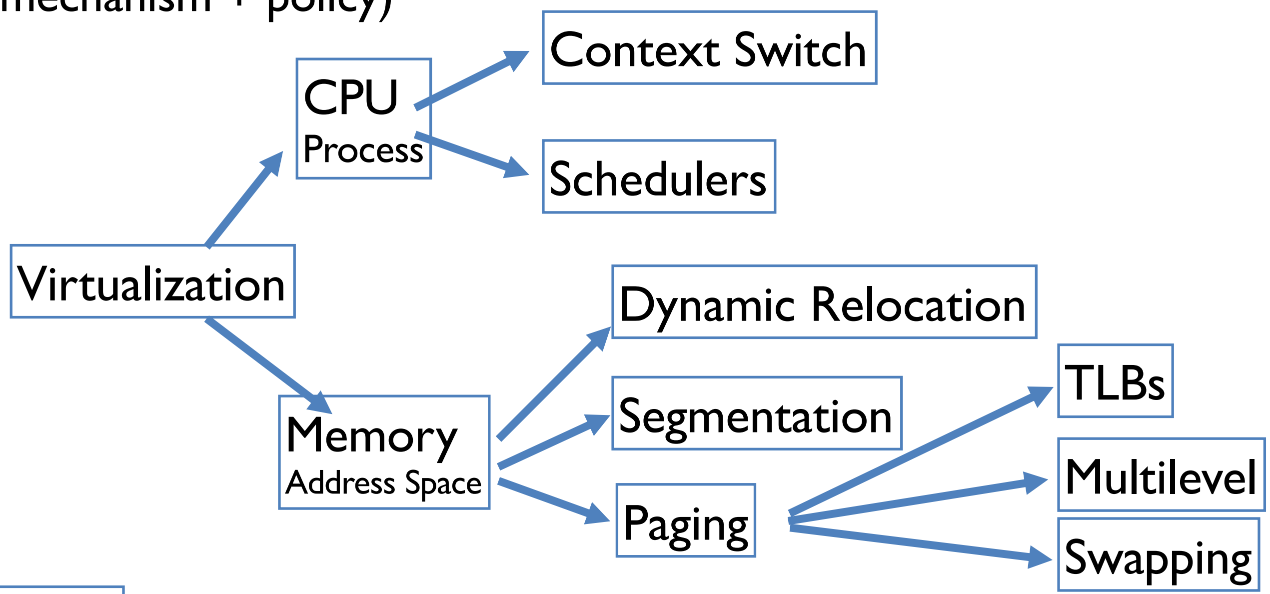
- P1 due today midnight.
- P2 will be released tomorrow
  - Make groups as soon as possible (Group = 2 students)
  - Not compulsory to have groups, but recommended
  - Discussion section introduced xv6
  - Project P2 will involve xv6 + syscalls & Unix shell

Lottery vs. MLFQ

- Multi Level Feedback Queue
  - For general purpose, handle CPU & I/O aware jobs
  - Turn around time, Throughput & Response time critical
  - Cannot guarantee a portion of the CPU to one process
- Lottery Scheduling
  - Used for proportional or equal CPU share/ fair share
  - Cannot guarantee low response time

OverView: Easy Piece 1

Virtual CPU: *illusion* of **private CPU registers**
- 2 lectures (mechanism + policy)



2) Concurrency
3) Persistence

Virtual RAM: *illusion* of **private memory**
- 5 lectures

AGENDA / LEARNING OUTCOMES

CPU virtualization

- Process Creation

Memory virtualization

- What is an address space?
- Why do we need memory virtualization?
- How to virtualize memory? Static, dynamic, base+bounds

CPU VIRTUALIZATION

Scheduling Policy: Review

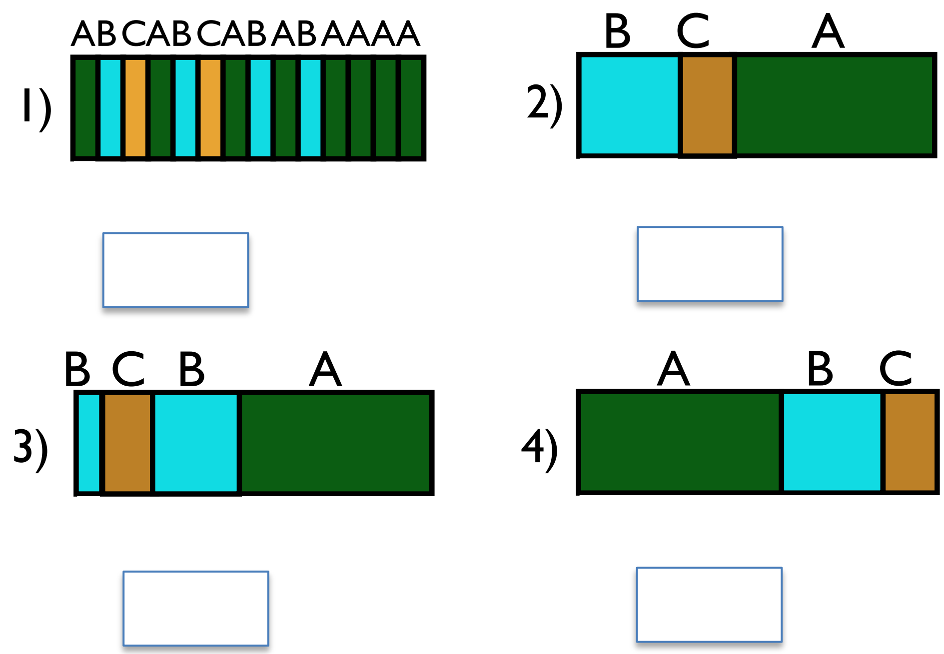
Workload

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

Schedulers:

- FIFO
- SJF
- STCF
- RR

Timelines



New Topic: Process Creation

Two ways to create a process

- Option 1: Build a new process from scratch
- Option 2: Copy an existing process and change it appropriately

Option 1: New Process

Create new process with specified executable and state

- Load specified code and data into memory;
  Create empty call stack
- Create and initialize PCB (make look like context-switch)
- Put process on ready list

Advantages:

- No wasted work

Disadvantages:

- Difficult to setup process and to express all possible options
  - Process permissions, where to write I/O, environment variables
  - Example: WindowsNT has call with 10 arguments

Option 2: Copy and Change

Copy existing process (fork) and change as needed (exec)

Fork()

- Calling process (parent) creates a child process
- Make copy of code, data, stack, and PCB of parent
- Add new PCB to ready list
- Any changes needed to child process?

Exec(char *file)

- Replace current data and code segments with those in specified executable file

Advantages:

Flexible, clean, simple

Disadvantages:

Wasteful to perform copy and then overwrite of memory

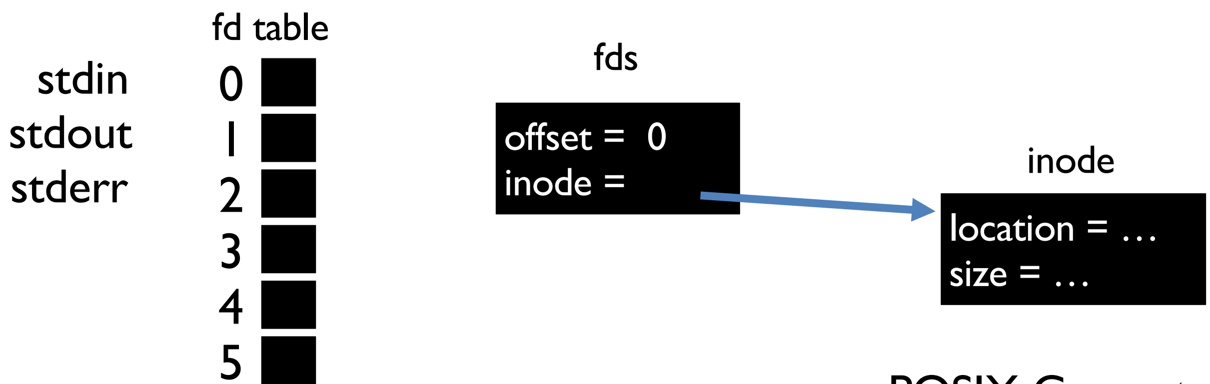
Unix Shells

```
while (1) {
  char *cmd = getcmd();
  int retval = fork();

  if (              ) {
    // This is the child process
    // Setup the child's process environment here
    // E.g., where is standard I/O, how to handle signals?
    exec(cmd);


   } else {
    // This is the parent process; Wait for child to finish
    int pid = 
    wait(pid);
   }
}
```

Stdin and stdout redirection with file descriptors



POSIX Guarantee?

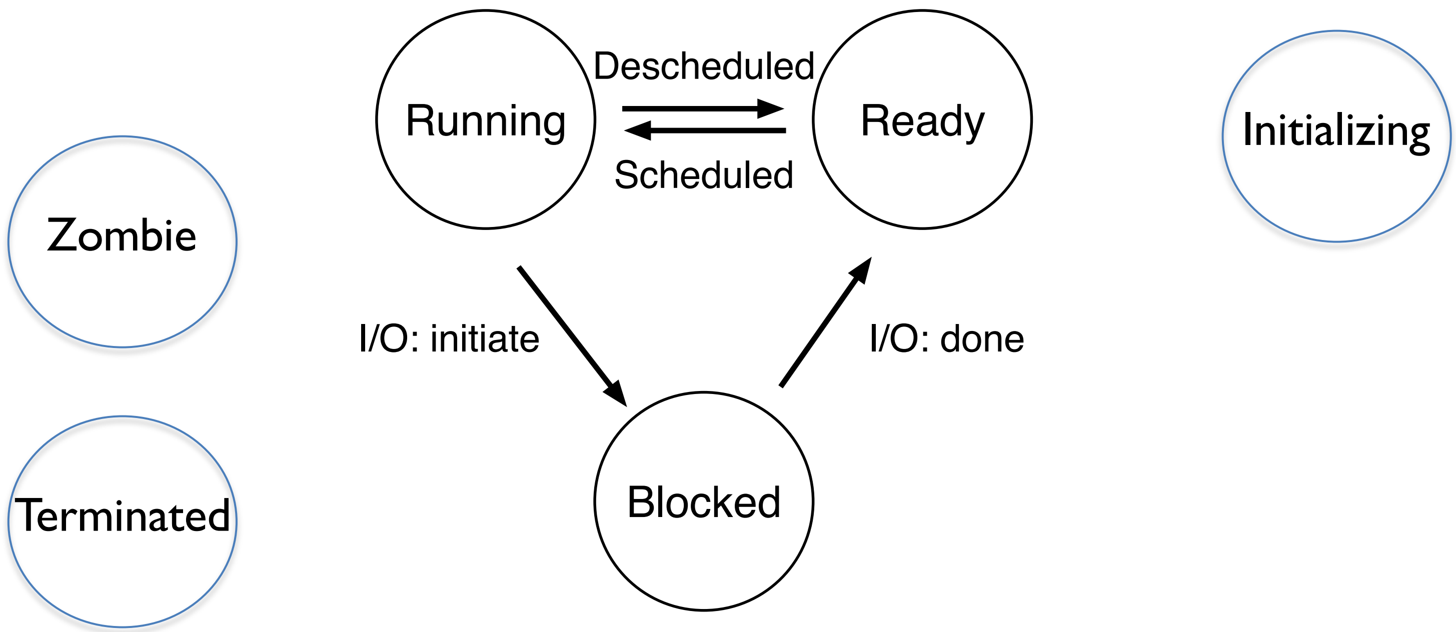
`int fd1 = open("file.txt"); // returns`

File redirection? `ls > newfile.txt`

```
Close(stdout)
Open("newfile.txt")
```

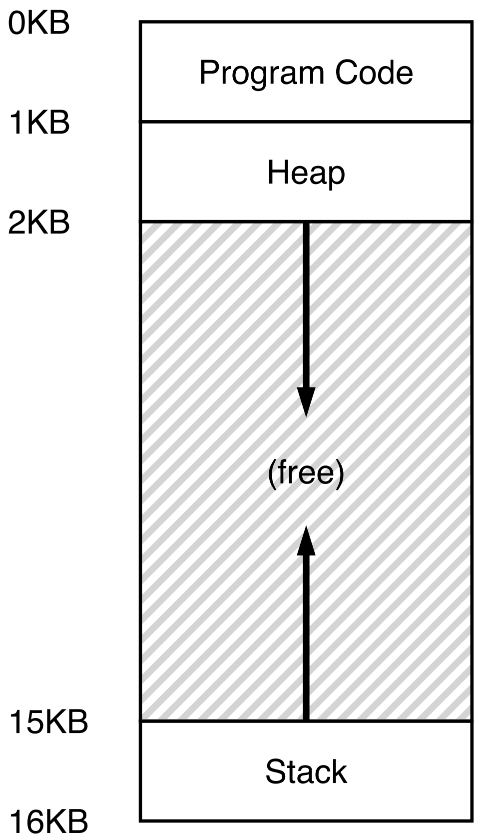
```
fprintf(stdout,
"where do I show up?");
```

STATE TRANSITIONS



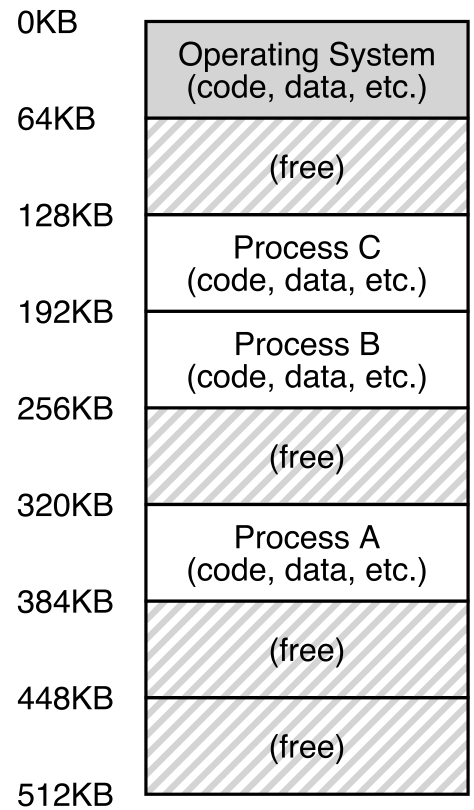
VIRTUALIZING MEMORY

ABSTRACTION: ADDRESS SPACE

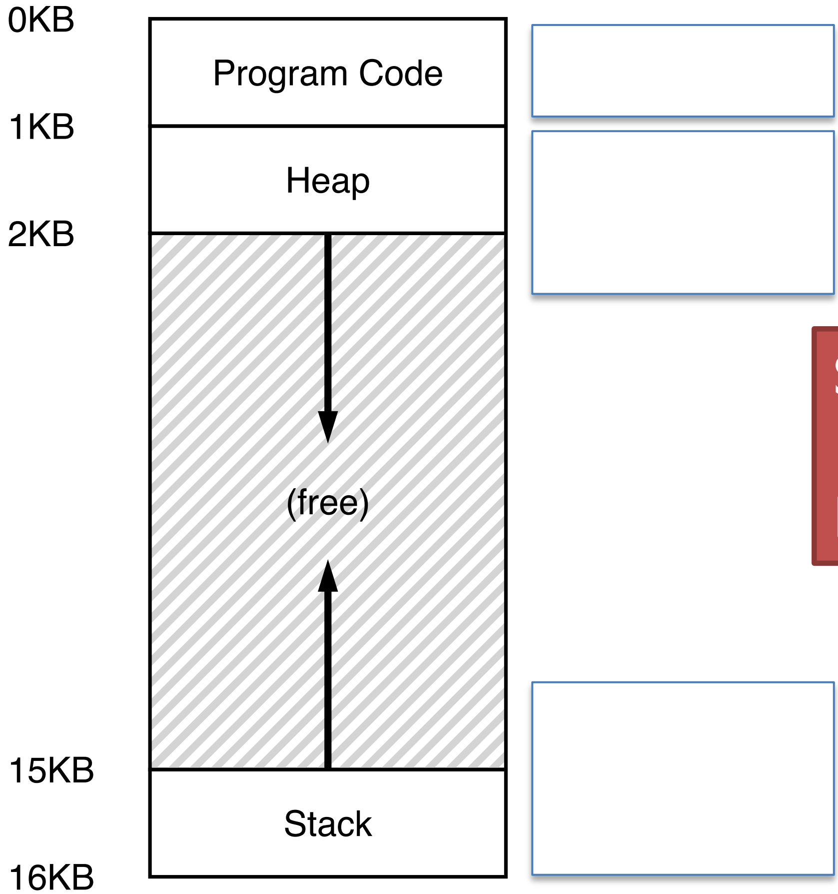


Address space:
Each process has own set of addresses

How can OS provide illusion of private (virtual) address space to each process?



WHAT IS IN ADDRESS SPACE?



Static:

Dynamic:

Motivation for Dynamic Memory

Why do processes need dynamic allocation of memory?

- Do not know amount of memory needed at compile time
- Must be pessimistic when allocate memory statically
  - Don't want to allocate enough for worst possible case

Recursive procedures

- Do not know how many times procedure will be nested

Complex data structures: lists and trees

- `struct my_t *p = (struct my_t *)malloc(sizeof(struct my_t));`

Two types of dynamic allocation

- Heap
- 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

Advantage

- Works for all data structures

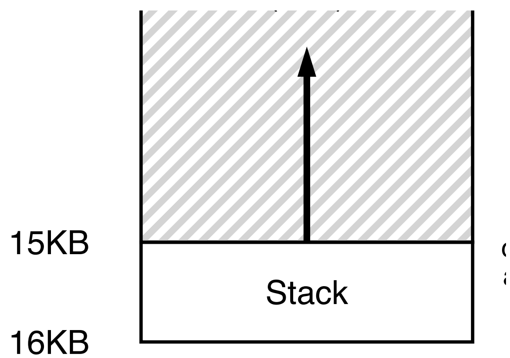
Disadvantages

- Allocation can be slow
- Has small chunks of free space - fragmentation
- Where to allocate 12 bytes? 16 bytes? 24 bytes??

What is OS's role in managing heap?

STACK ORGANIZATION

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



Memory must be freed in opposite order from allocation

Pointer between allocated and free space
- Allocate: Increment pointer
- Free: Decrement pointer

No fragmentation!

What is OS's role in managing stack?

WHAT GOES ON Process 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);
}
```

Use per-process stack for procedure call frames (local variables and parameters)

Stack vs. Heap

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

Possible segments: static data, code, stack, heap

Address	Location
x	
main	
y	
z	
*z	

How many MEMORY ACCESSes?

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

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

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

%rip is instruction pointer
(or program counter)

Initial %rip = 0x10
%rbp = 0x200

```
objdump test.o
```

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

How many memory accesses?

To what addresses?

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

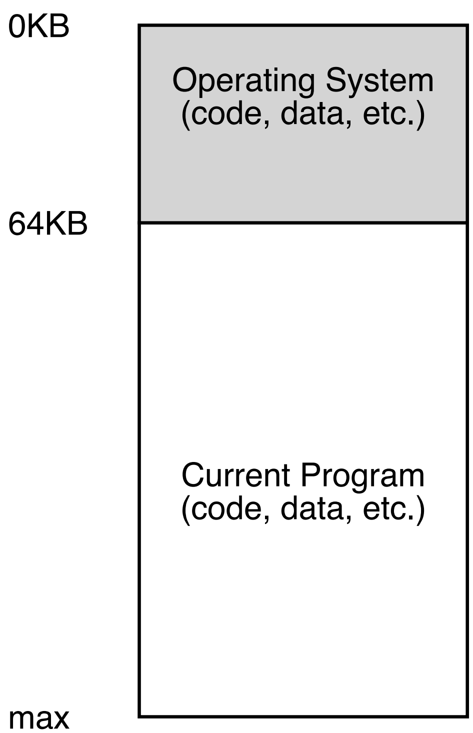
%rip is instruction pointer (or program counter)

Exec:

Exec:

Exec:

Motivation for virtualizing memory



First systems did not virtualize

Uniprogramming: One process runs at a time

Disadvantages?

MULTIPROGRAMMING GOALS

Transparency:

Protection:

Efficiency:

Sharing:

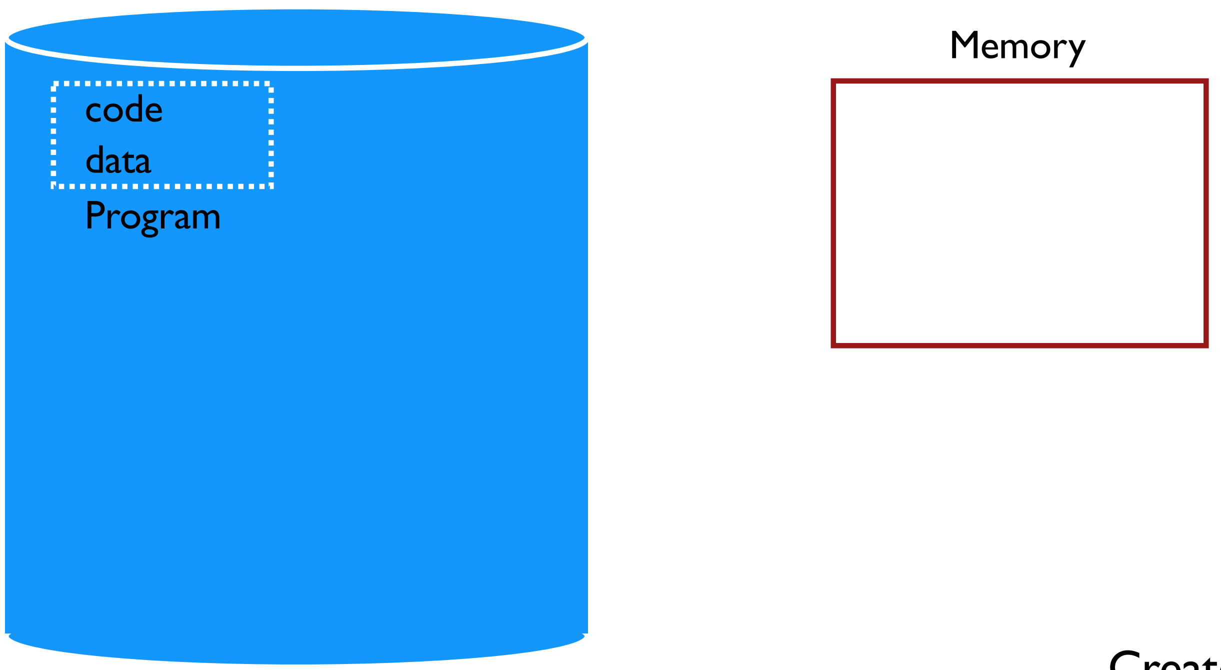
HOW TO VIRTUALIZE MEMORY

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



Create p1, swap out P1, create P2, swap out P2, swap in P1

PROBLEMS WITH TIME SHARING memory?

Ridiculously poor performance

Better Alternative: Space sharing of Physical Memory

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

2) Static Relocation

OS picks static physical location for each process when

each code segment before loading it in memory

Each rewrite for different process uses different addresses and pointers

Change jumps, loads of static data

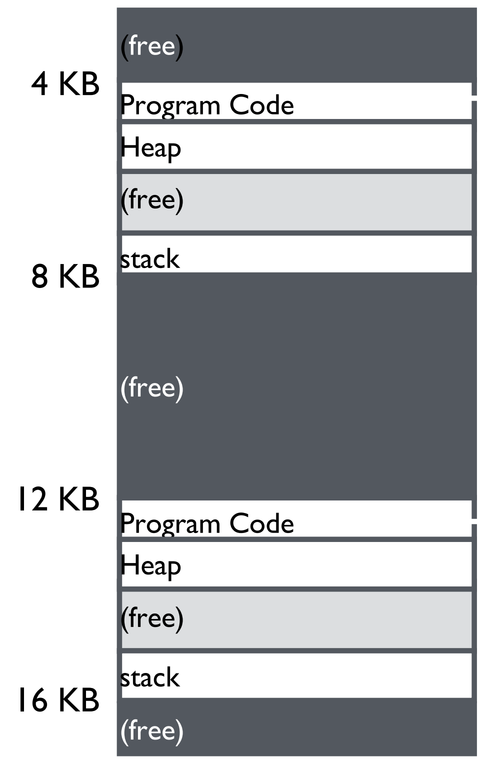
```
0x10: movl  0x8(%rbp), %edi
0x13: addl  $0x3, %edi
0x19: movl  %edi, 0x8(%rbp)
```

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)
```



why didn't OS rewrite stack addr?

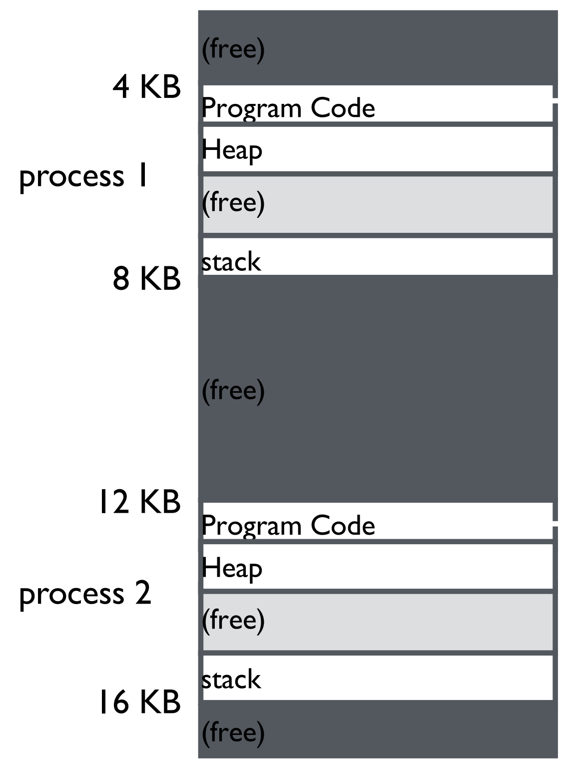
Static Relocation: Disadvantages

No protection

- Process can
- No privacy

Cannot move address space after it has been placed

- May not have free space to allocate new process



3) Dynamic Relocation

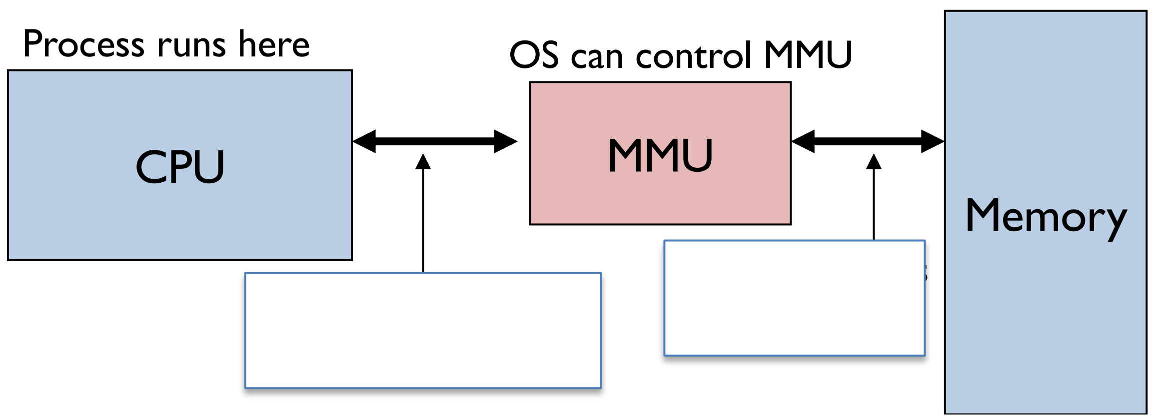
Goal: Protect processes from one another

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 privilege levels

User: User processes run in user mode

Privileged (protected, kernel) mode: OS runs

- When enter OS (trap, system calls, interrupts, exceptions)
- Allows privileged instructions to be executed
-

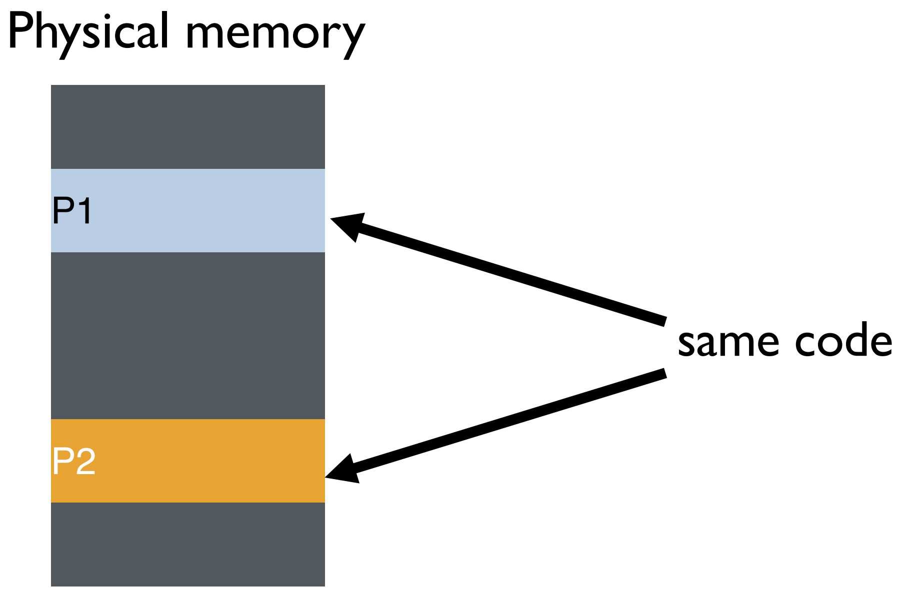
Dynamic Relocation with Base Register

Translate virtual addresses to physical by adding a fixed offset each time.

Store offset in base register in MMU

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

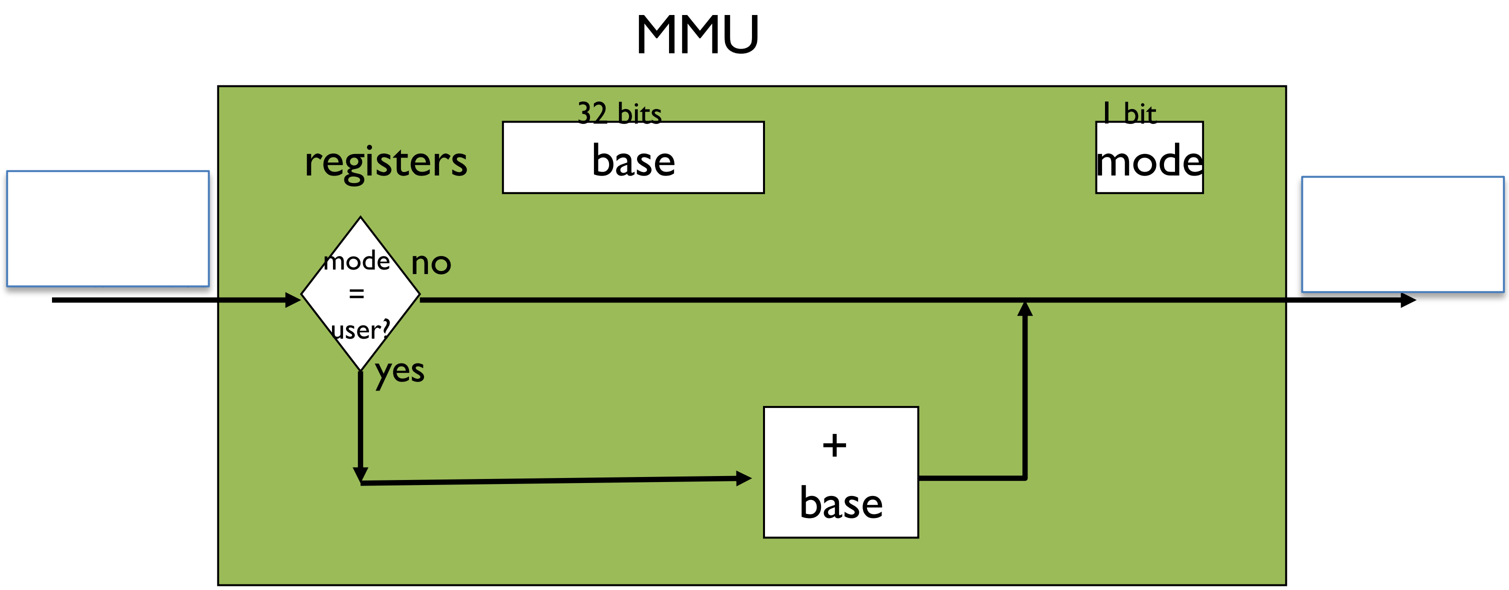
P1: load 100, R1

VIsual Example of DYNAMIC RELOCATION: BASE REGISTER

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



Who Controls the Base Register?

a) What entity translates virtual addresses to physical addresses with base register?

(1) process, (2) OS, or (3) HW

a) What entity should determine contents and modify the base register?

(1) process, (2) OS, or (3) HW

Protection with Base Register?

P1 base: 1024
P2 base: 4096

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	
P1: store 3072, R1	store 4096, R1	(3072 + 1024)

Can P2 hurt P1?
Can P1 hurt P2?

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

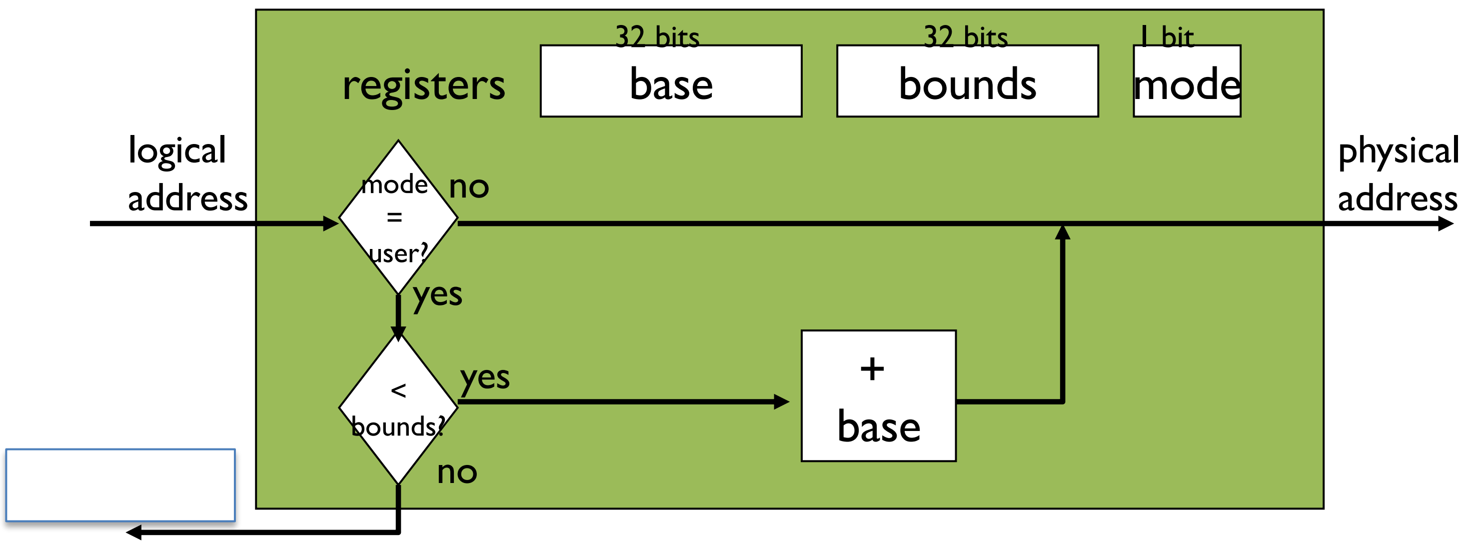
- Sometimes defined as largest physical address (base + size)

If process loads/stores beyond bounds, what does OS do?

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



Protection with Base + Bounds 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
P1: store 3072, R1	

Can P2 hurt P1?
Can P1 hurt P2?

Managing Processes with Base and Bounds

Track base and bounds registers in PCB (process-control-block)

Context-switch Steps

- OS in 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 to privileged mode
- User process cannot change base and bounds registers

Base and Bounds Advantages

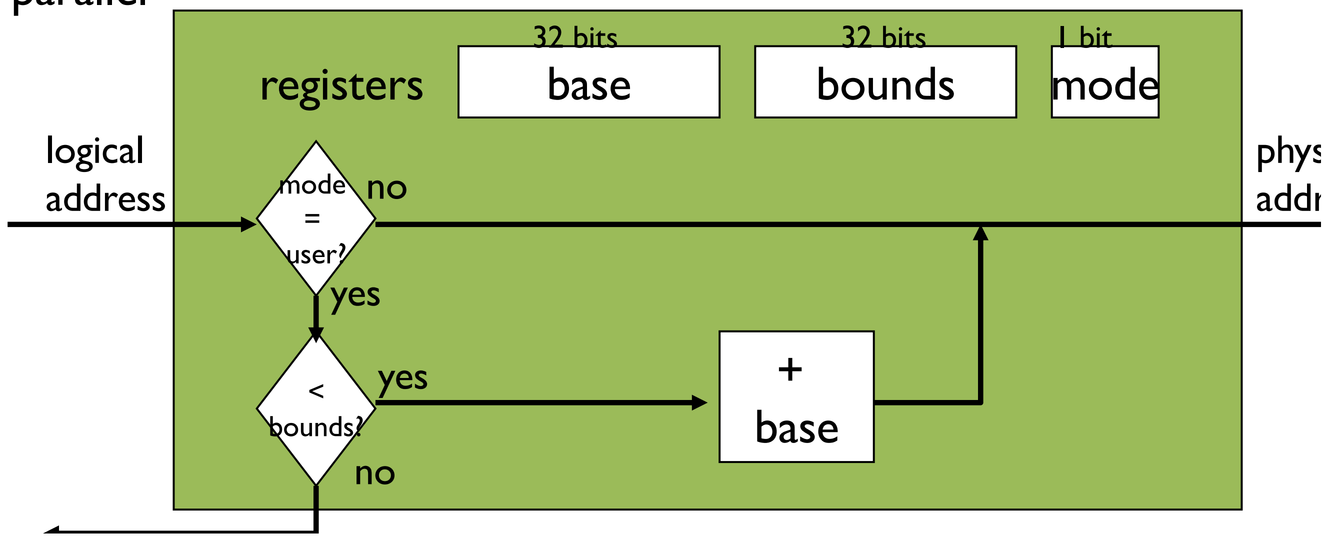
Provides protection (both read and write) across address spaces

Supports dynamic relocation

Can place process at different locations initially and also move address spaces

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

Fast: Add and compare in parallel

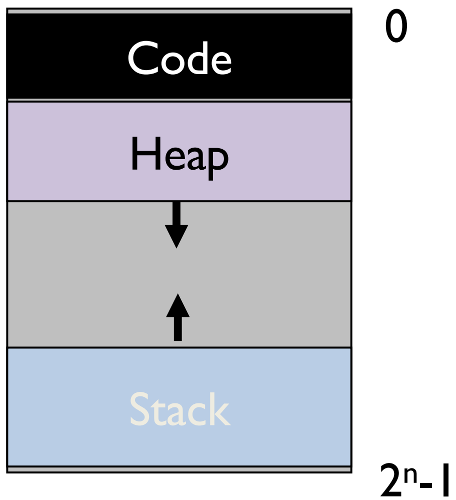


Base and Bounds DISADVANTAGES

Disadvantages

- Each process must be allocated contiguously in physical memory
- Must reserve memory that may not be used by process

- No partial sharing between processes:
  Cannot share limited parts of address space



Next VM Topics

- Remove those disadvantages, add new ones, fix those…
  - Segmentation
  - Paging
  - Segmentation + Paging
  - Multi-level Page Tables
  - TLBs