VIRTUALIZATION: CPU TO MEMORY

Andrea Arpaci-Dusseau

CS 537, Fall 2019

ADMINISTRIVIA

- Project 1 Due Last Night
- Project 2 Available: Due Monday
- Discussion section: xv6 code walk through!
- Homeworks:
  - Process (Due Thursday)
  - Scheduling and MLFQ (Due Tuesday)

AGENDA / LEARNING OUTCOMES

- CPU virtualization
  - Process Creation
  - MLFQ scheduler

- Memory virtualization
  - Why do we need memory virtualization?
  - How to virtualize memory? Static, dynamic, base+bounds

CPU VIRTUALIZATION

NEW TOPIC: PROCESS CREATION

Two ways to create a process

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

OPTION 1: NEW PROCESS

- Option 1: Create new process with specified executable
- Steps
  - 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: CLONE AND CHANGE

Option 2: Clone existing process and change as needed

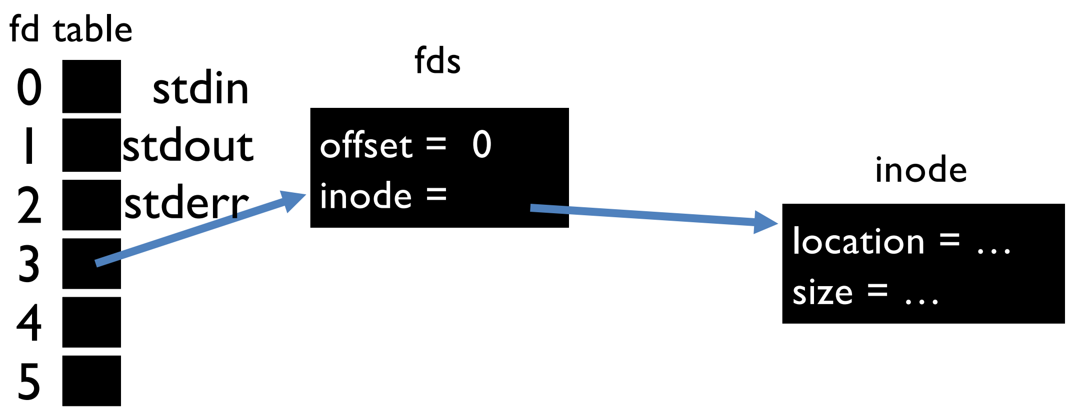
- Example: Unix fork() and exec()
  - Fork(): Clones calling process
  - Exec(char *file): Overlays file image on calling process
- Fork()
  - Stop current process and save its state
  - Make copy of code, data, stack, and PCB
  - Add new PCB to ready list
  - Any changes needed to child process?
- Exec(char *file)
  - Replace current data and code segments with those in specified 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 (retval == 0) {
    // 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);
    // exec does not return if it succeeds
    printf(“ERROR: Could not execute %s\n”, cmd);
    exit(1);
   } else {
    // This is the parent process; Wait for child to finish
    int pid = retval;
    wait(pid);
   }
}
```

STDIN AND STDOUT REDIRECTION WITH FILE DESCRIPTORS

```
int fd1 = open("file.txt"); // returns 3
```



File redirection? `ls > newfile.txt`

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

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

SCHEDULING POLICY: REVIEW

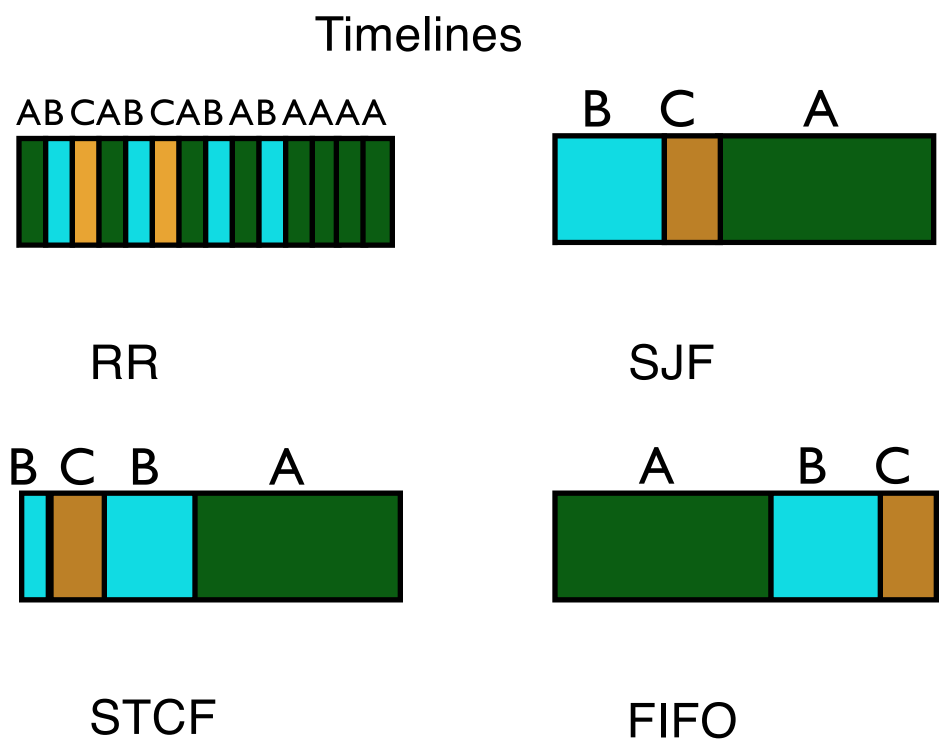
SCHEDULING POLICY: REVIEW

Workload

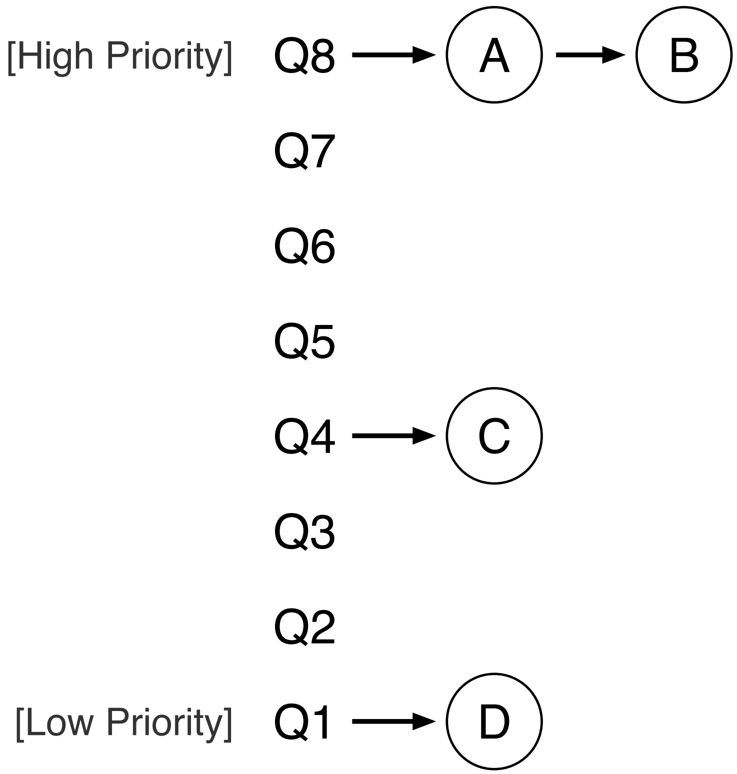
JOB	arrival	run
A	0	40
B	0	20
C	5	10

Schedulers:

FIFO
SJF
STCF
RR



MLFQ EXAMPLE



Rules for MLFQ

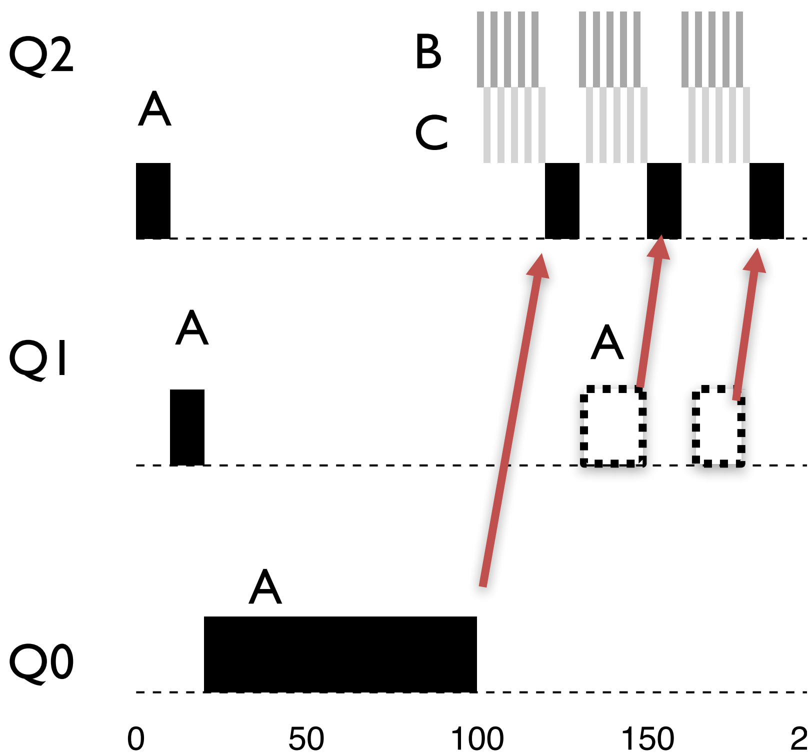
Rule 1: If priority(A) > Priority(B)
A runs

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

MLFQ WITH STARVATION MECHANISM



CANVAS QUIZ: MLFQ

This program, mlfq.py, allows you to see how the MLFQ scheduler presented in this chapter behaves. As before, you can use this to generate problems for yourself using random seeds, or use it to construct a carefully-designed experiment to see how MLFQ works under different circumstances. To run the program, type:

```
prompt> ./mlfq.py
```

Use the help flag (-h) to see the options:

http://pages.cs.wisc.edu/~remzi/OSTEP/Homework/homework.html

VIRTUALIZING MEMORY

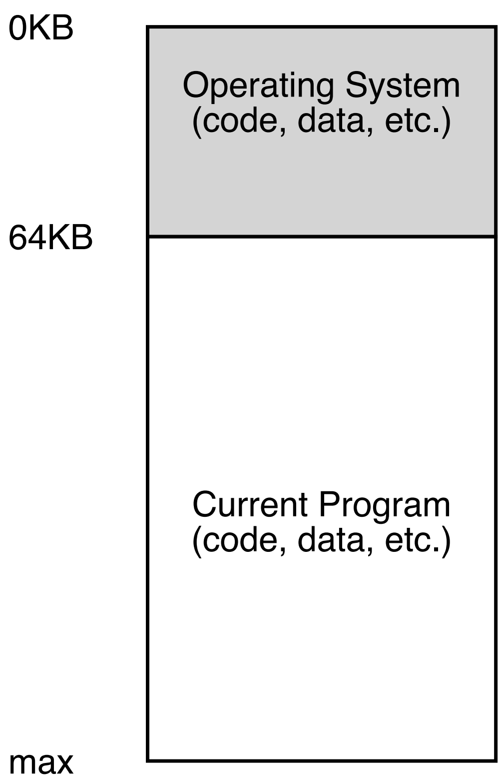
MORE VIRTUALIZATION

1st part of course: Virtualization

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

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

MOTIVATION FOR VIRTUALIZING MEMORY



First systems did not virtualize

Uniprogramming: One process runs at a time

Disadvantages?

- Only one process ready at a time
- Process can destroy OS

MULTIPROGRAMMING GOALS

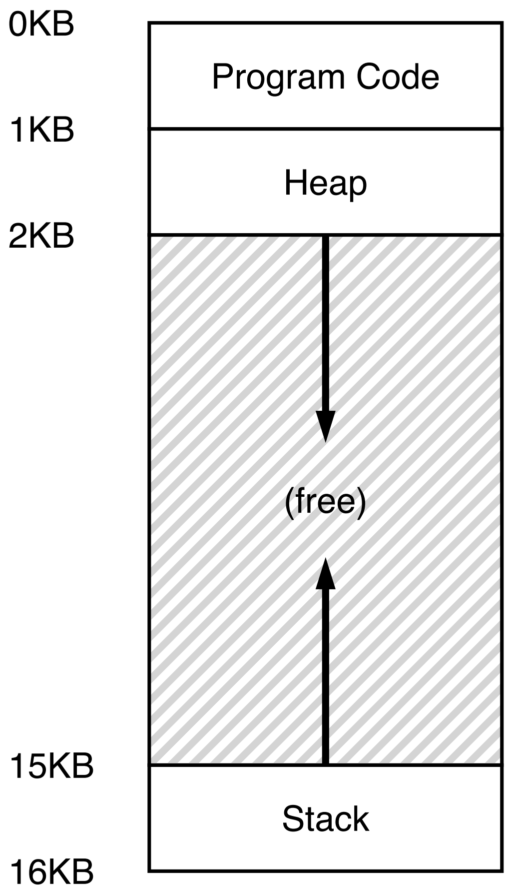
Transparency:
Process is unaware of sharing
Work regardless of number of processes

Protection:
Cannot corrupt or read OS or other process memory

Efficiency:
Do not waste memory (no fragmentation) or slow down processes

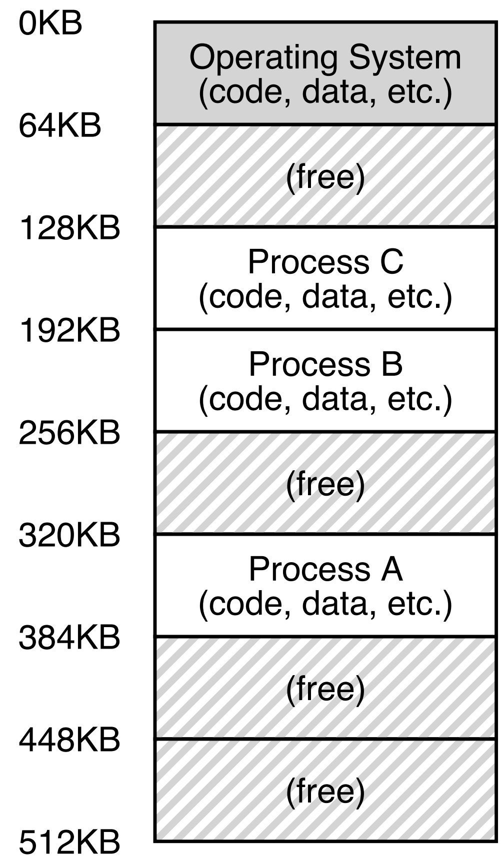
Sharing:
Enable sharing between cooperating processes

ABSTRACTION: ADDRESS SPACE

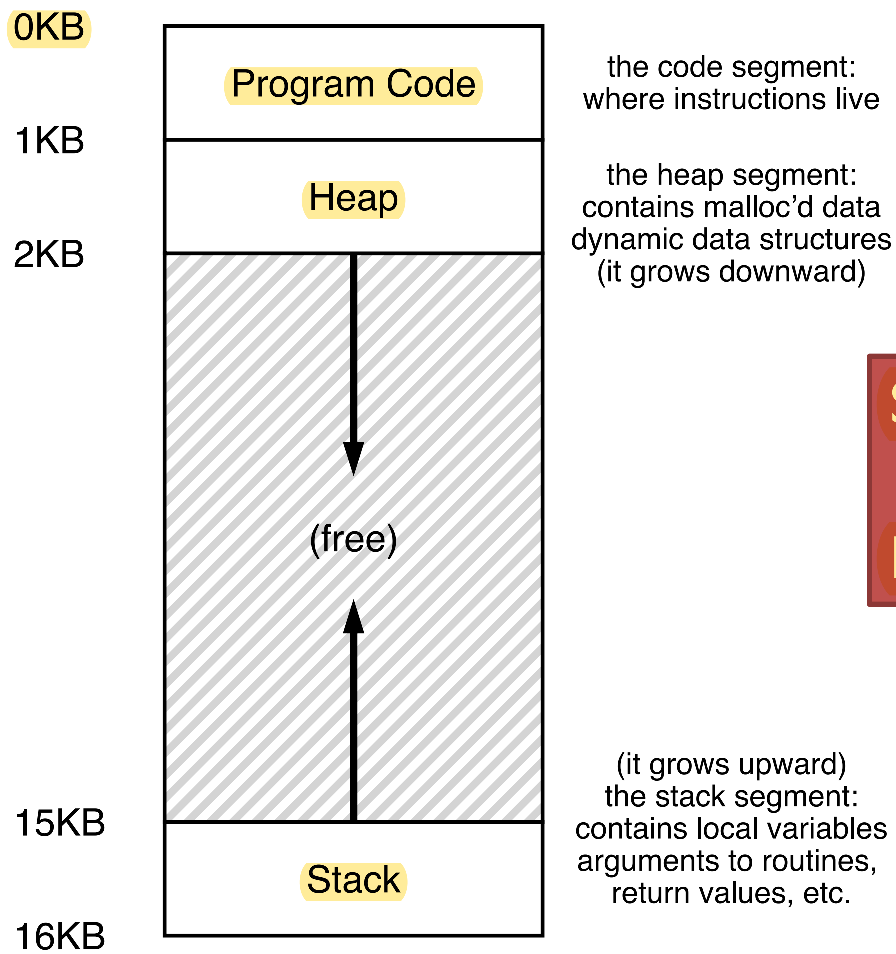


Address space:
Each process has own set of addresses

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



WHAT IS IN ADDRESS SPACE?



Static: Code and some global variables

Dynamic: Stack and Heap

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
  - Allocate enough for worst possible case; Storage is used inefficiently

Recursive procedures

- Do not know how many times procedure will be nested

Complex data structures: lists and trees

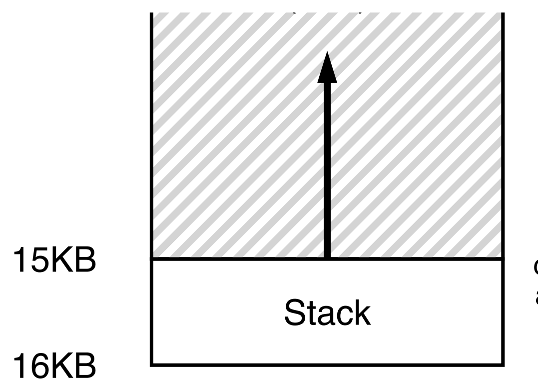
- `struct my_t *p = (struct my_t *)malloc(sizeof(struct my_t));`

Two types of dynamic allocation

- Stack
- 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 GOES ON 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);
}
```

OS uses stack for procedure call frames (local variables and parameters)

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?

– OS gives big chunk of free memory to process; library manages individual allocations

ONE-MINUTE NEIGHBOR CHAT

```
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	

MEMORY ACCESS

```
#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)
```

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

```
otool -tv demo1.o
```

TWO-MINUTE CHAT: 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 Accesses to what addresses?

How many?

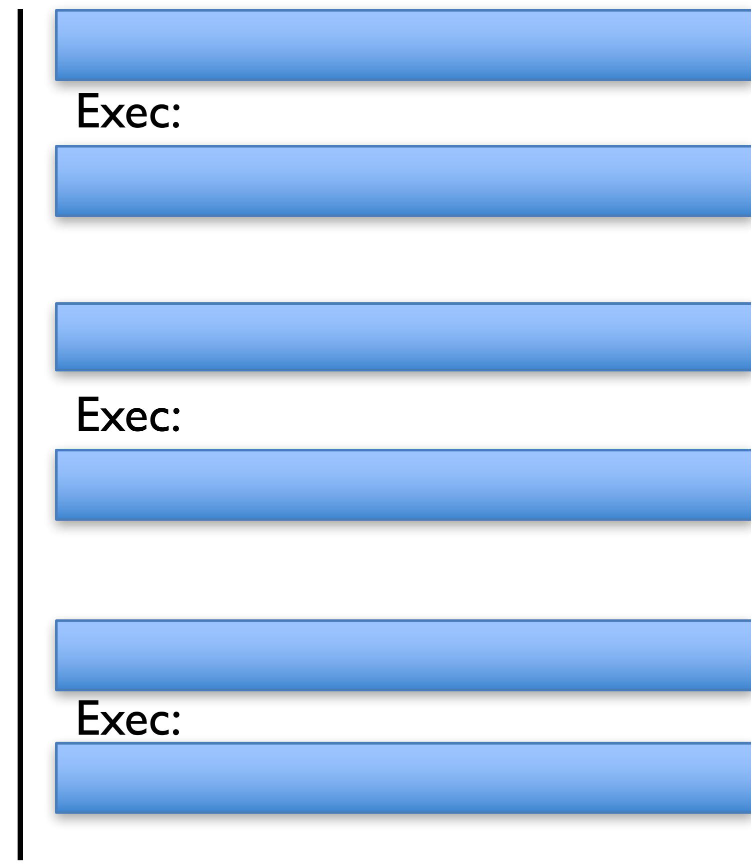
MEMORY ACCESS

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)



HOW TO VIRTUALIZE MEMORY

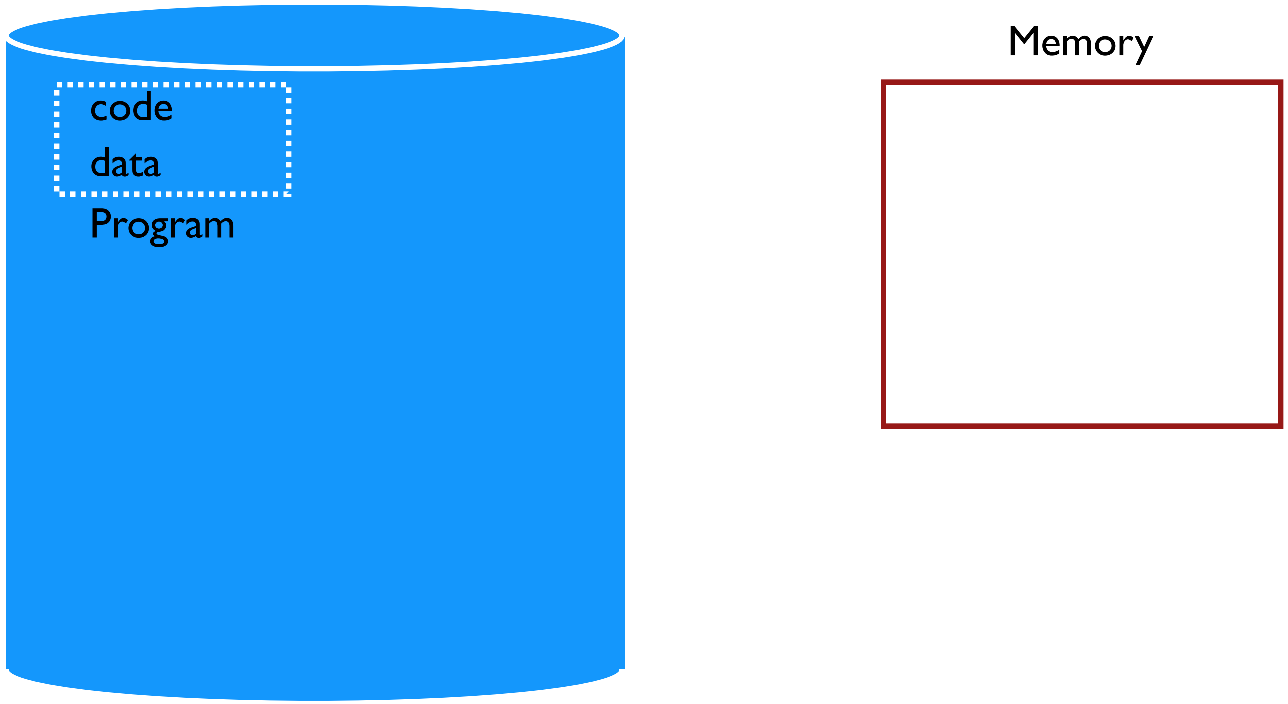
Problem: How to run multiple processes simultaneously?

Addresses are "hardcoded" into process binaries

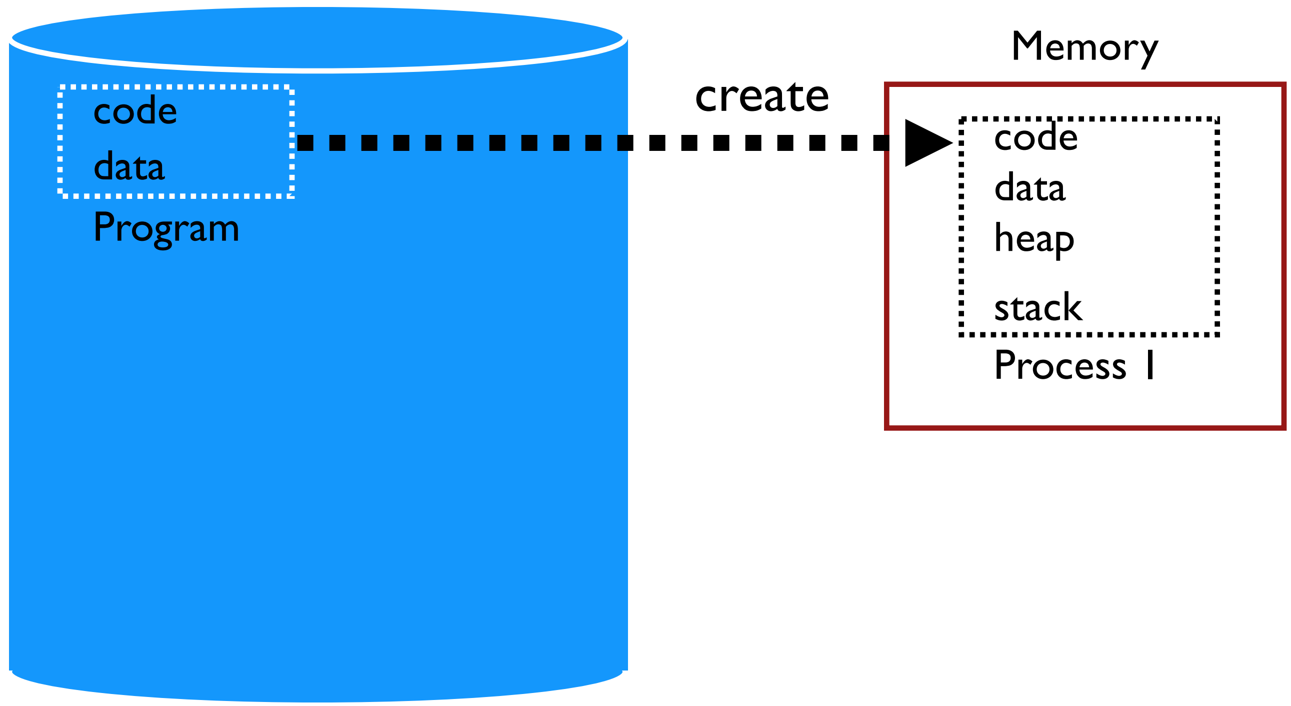
How to avoid collisions?

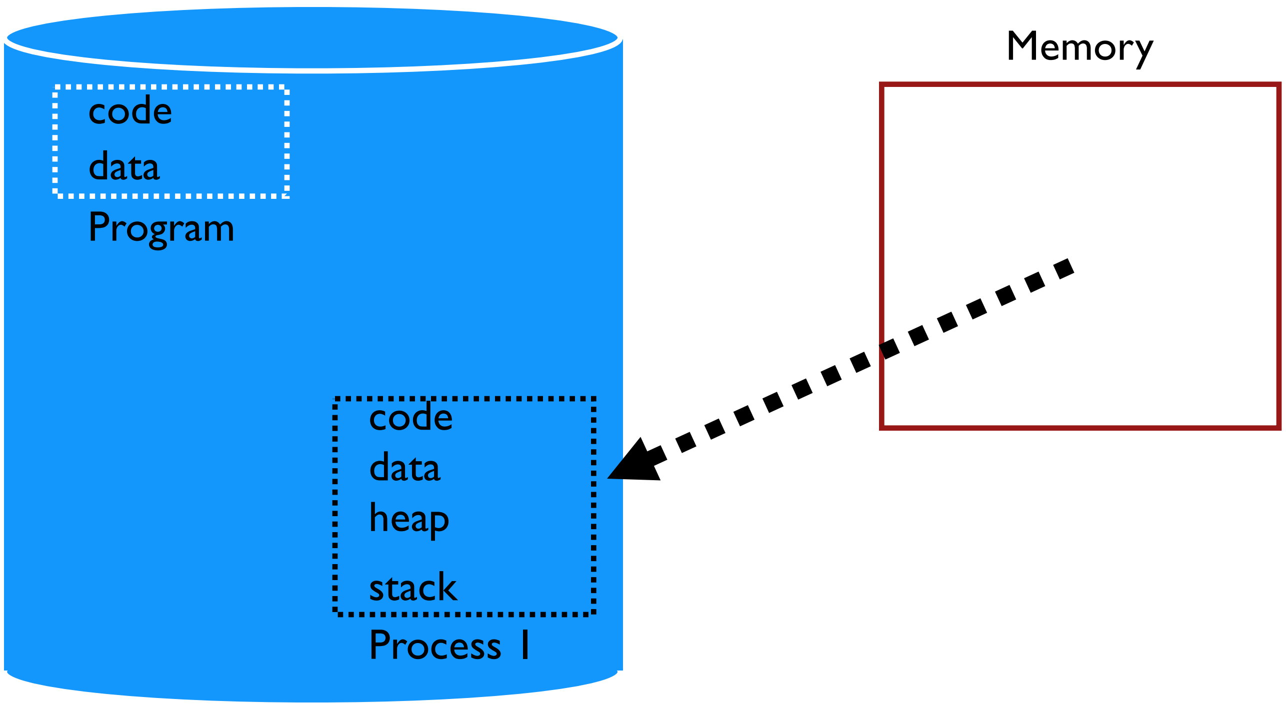
Possible Solutions for Mechanisms (covered today):

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

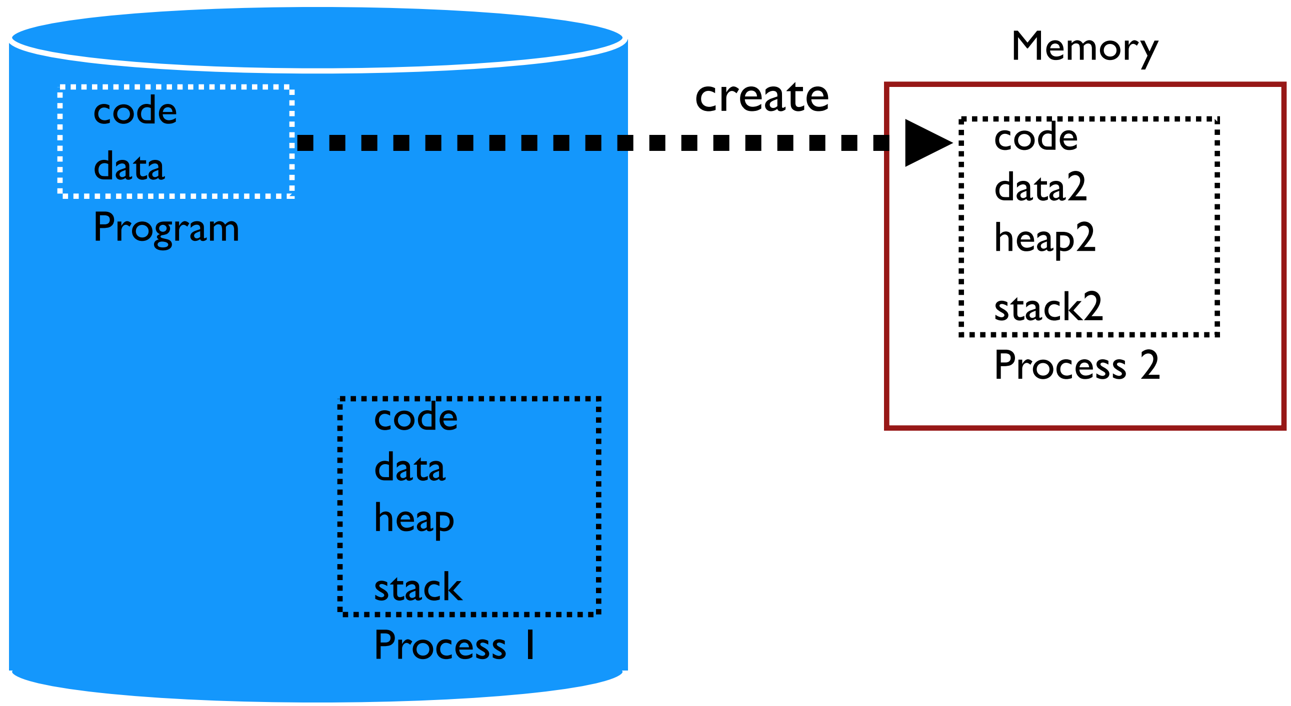


TIME SHARE MEMORY: EXAMPLE

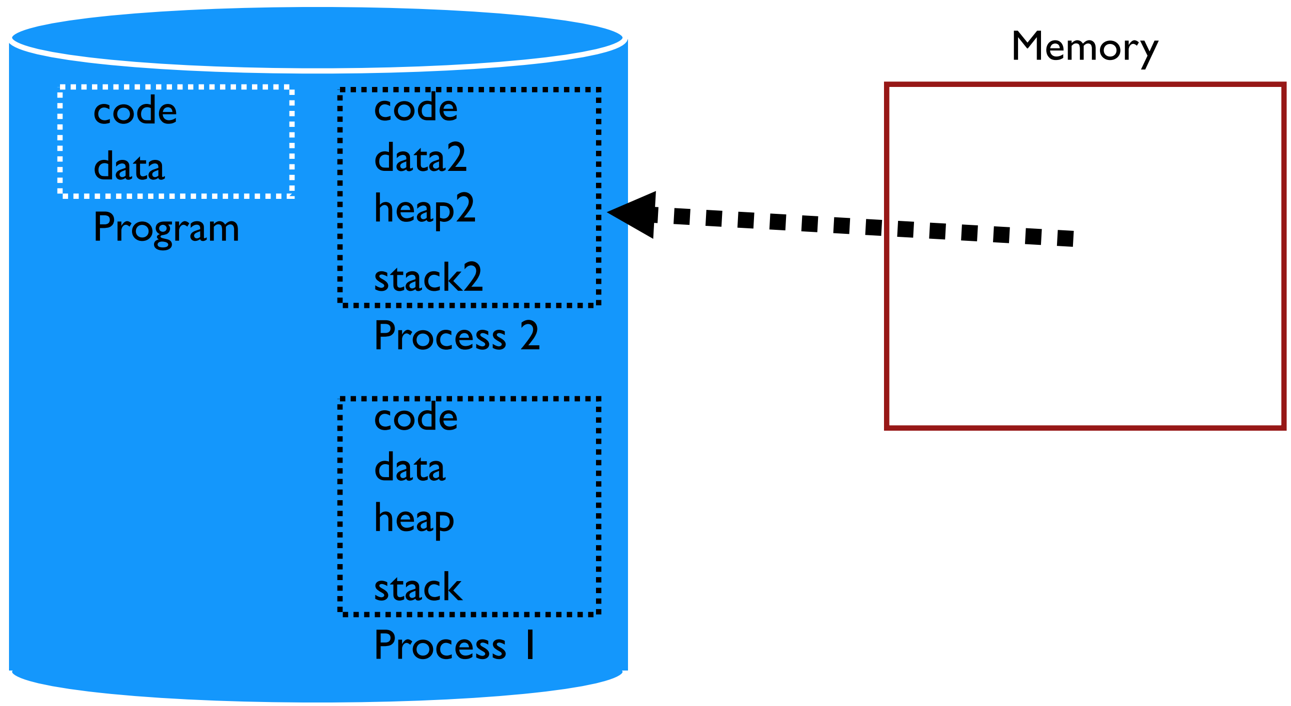

code
data
Program
create
Memory
code
data
heap
stack
Process 1

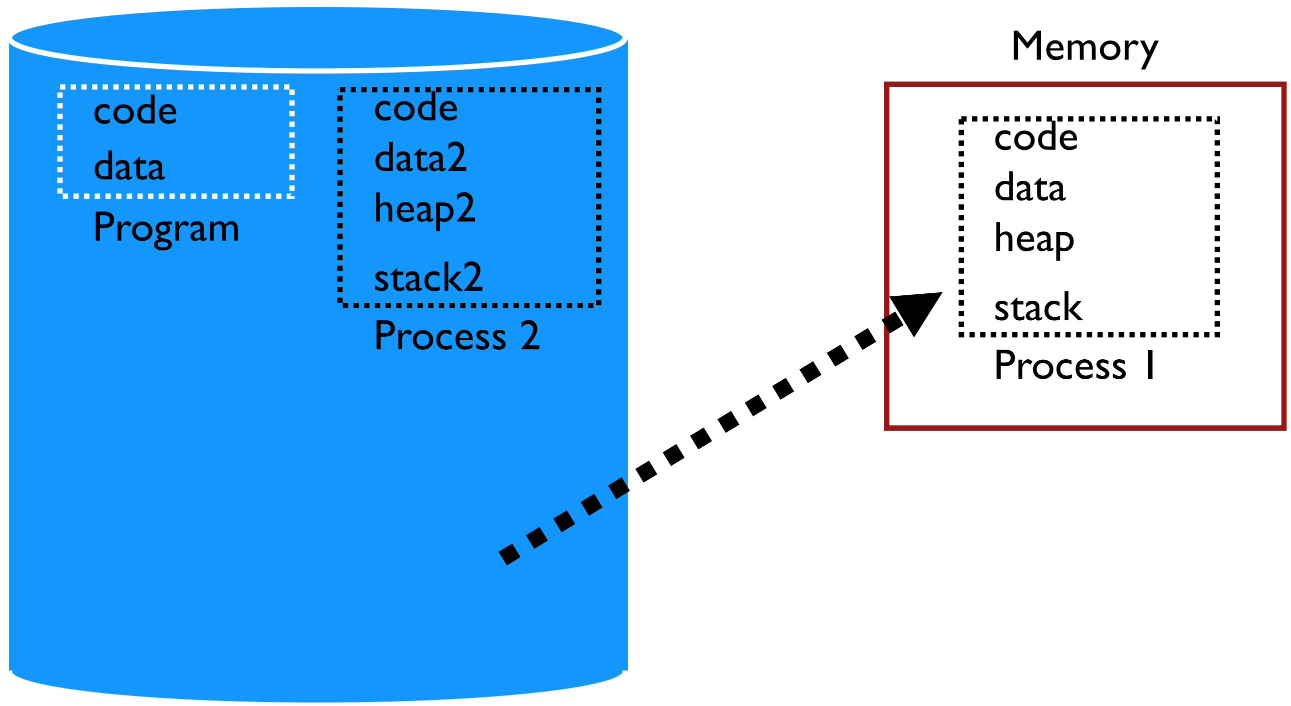


code
data
Program
code
data
heap
stack
Process 1
Memory



code
data
Program
create
Memory
code
data2
heap2
stack2
Process 2
code
data
heap
stack
Process 1


code
data
Program
code
data2
heap2
stack2
Process 2
code
data
heap
stack
Process 1
Memory


code
data
Program
code
data2
heap2
stack2
Process 2
Memory
code
data
heap
stack
Process 1

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

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

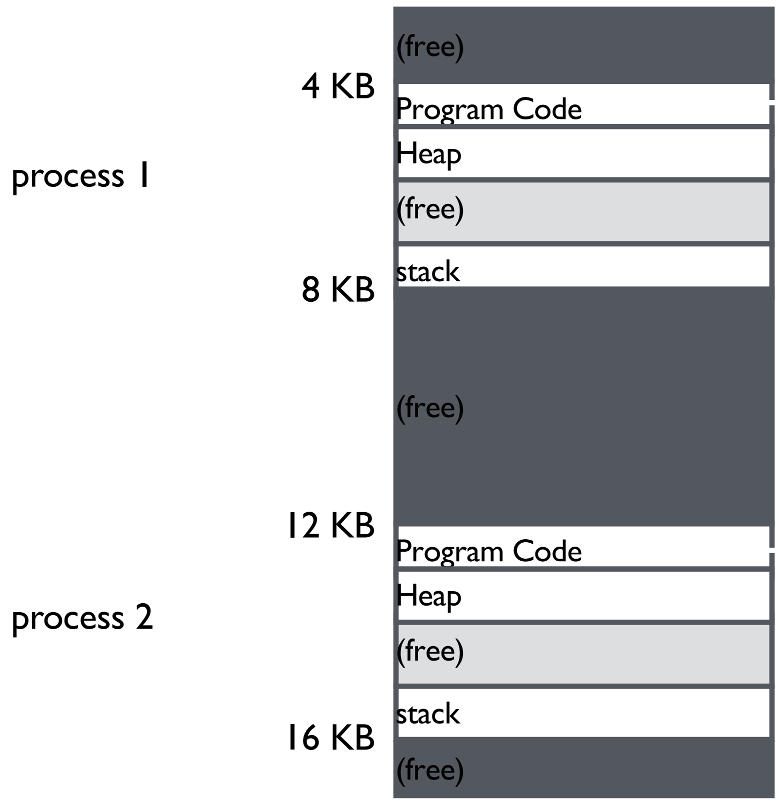
rewrite

- `0x10:movl0x8(%rbp), %edi`
- `0x13:addl$0x3, %edi`
- `0x19: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)
```

why didn’t OS rewrite stack addr?

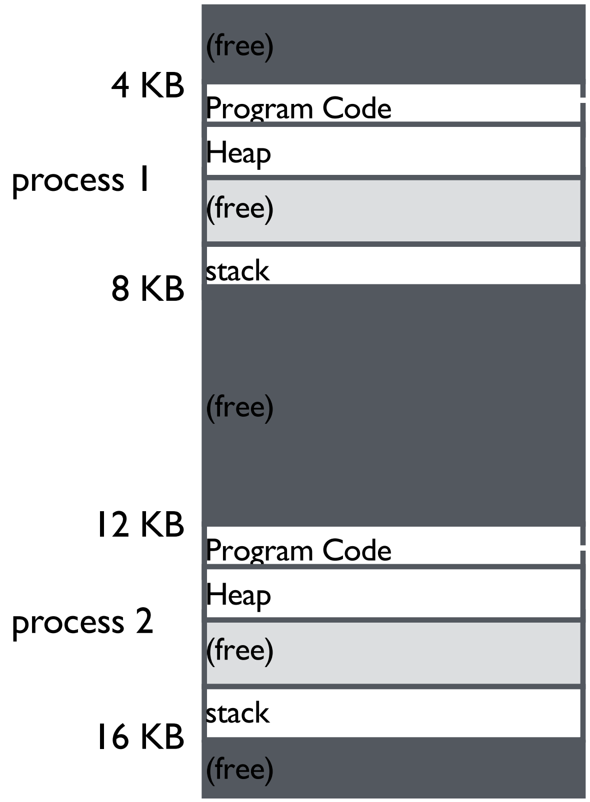
STATIC RELOCATION: DISADVANTAGES

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

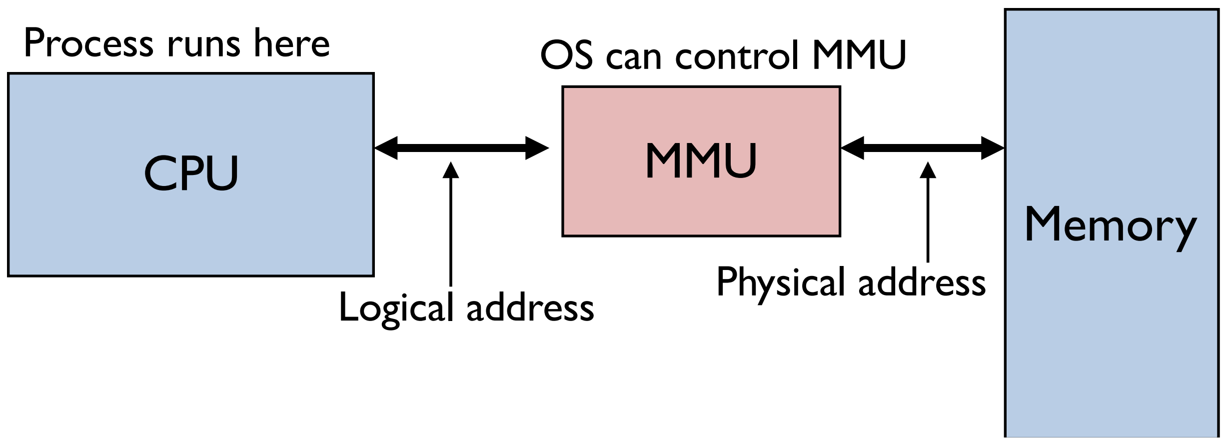
Goal: Protect processes from one another

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

Two operating modes

Privileged (protected, kernel) mode: OS runs

- When enter OS (trap, system calls, interrupts, exceptions)
- Allows privileged instructions to be executed
  - **Can manipulate contents of MMU**
- **Allows OS to access all of physical memory**

User mode: User processes run

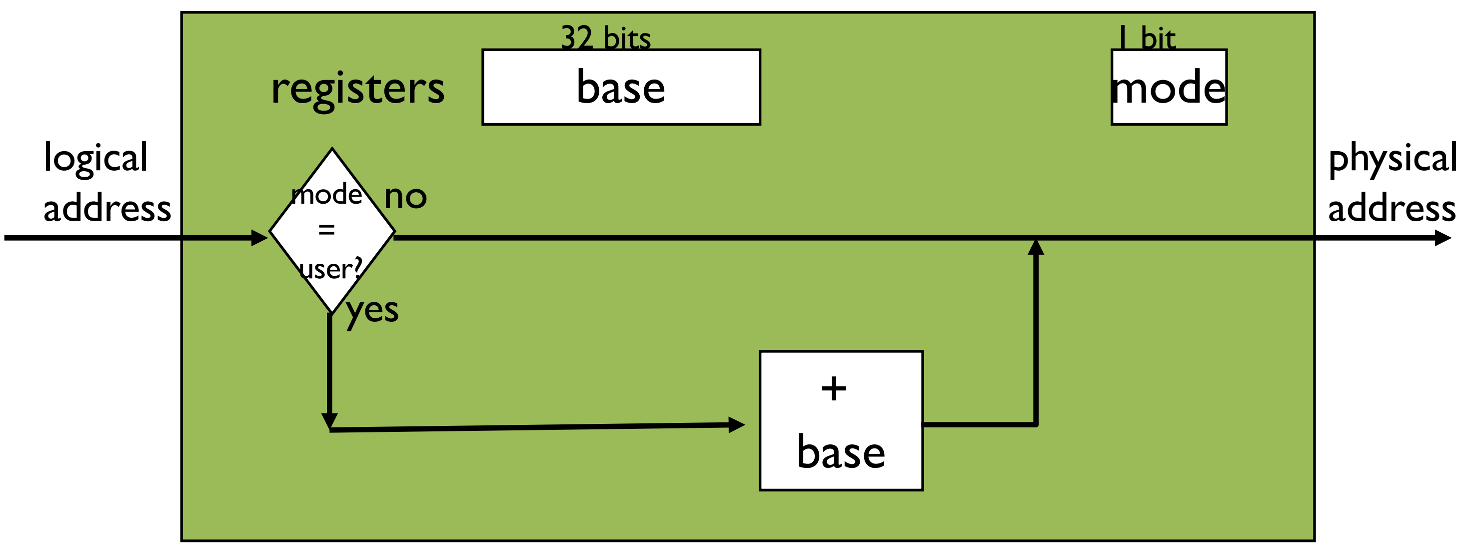
- **Perform translation of logical address to physical address**

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

MMU



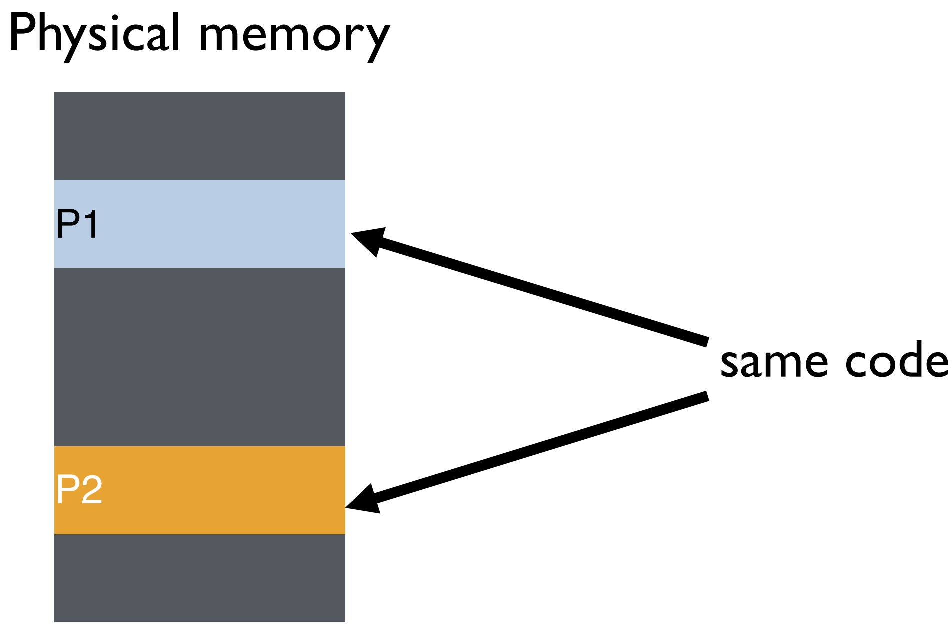
DYNAMIC RELOCATION WITH BASE REGISTER

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

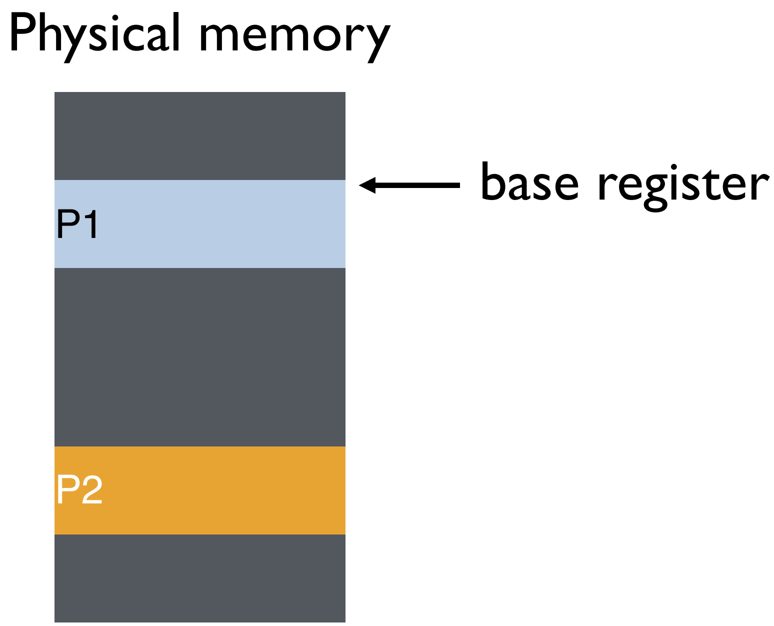
P1: load 100, R1

P2: load 100, R1

P2: load 1000, R1

P1: load 100, R1

VISUAL EXAMPLE OF DYNAMIC RELOCATION: BASE REGISTER



Virtual

P1: load 100, R1

P2: load 100, R1

P2: load 1000, R1

P1: load 100, R1

VISUAL EXAMPLE OF DYNAMIC RELOCATION: BASE REGISTER

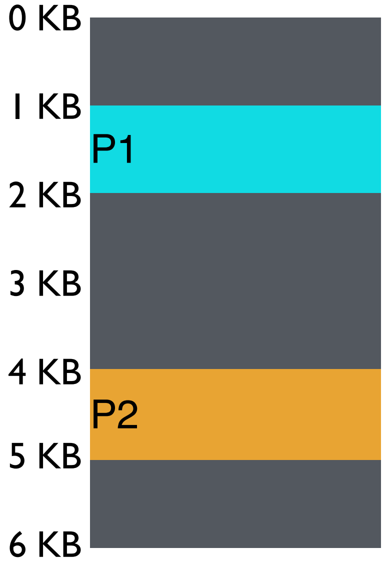
QUIZ: WHO CONTROLS THE BASE REGISTER?

What entity performs translation of addresses with base register?

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

What entity should determine contents and modify the base register?

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



Can P2 hurt P1?
Can P1 hurt P2?

Virtual

P1: load 100, R1

P2: load 100, R1

P2: load 1000, R1

P1: load 100, R1

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 5196, R1	
P1: load 100, R1	load 2024, R1	
P1: store 3072, R1	store 4096, R1	(3072 + 1024)

Can P2 hurt P1?

Can P1 hurt P2?

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

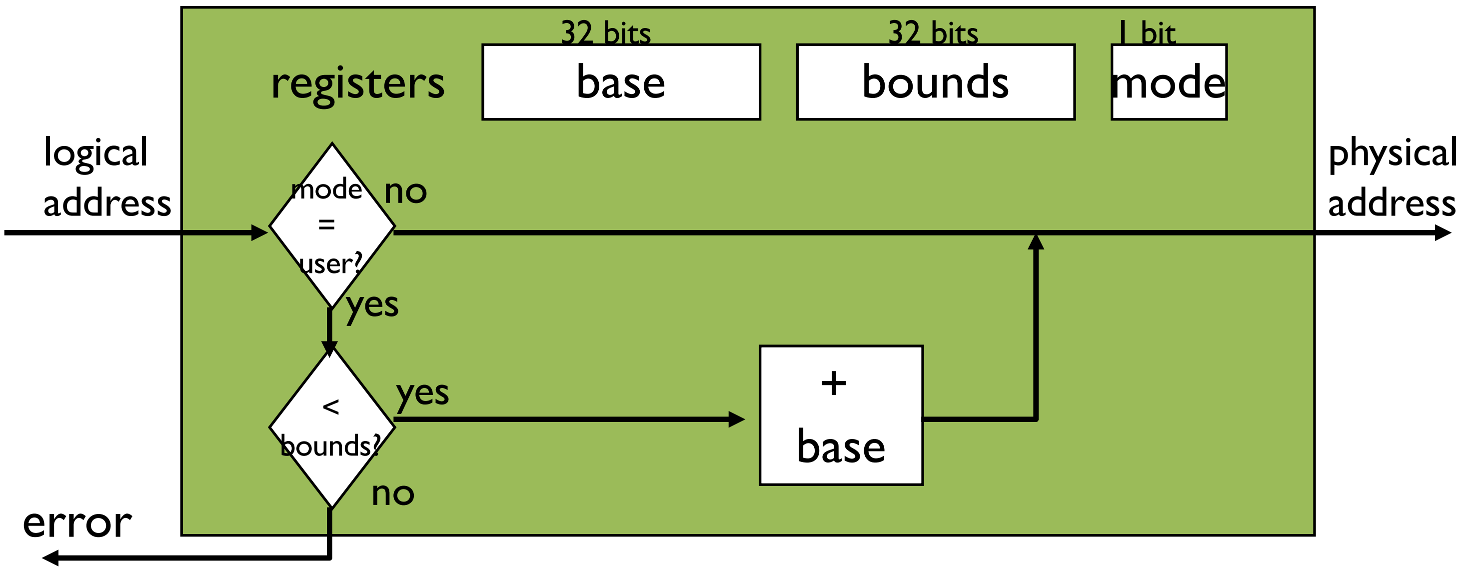
- Sometimes defined as largest physical address (base + size)

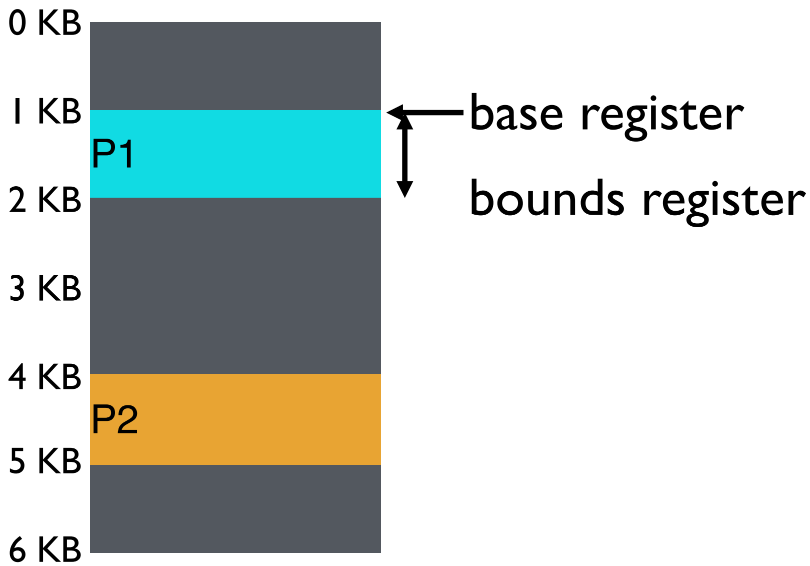
OS kills process if process loads/stores beyond bounds

IMPLEMENTATION OF BASE+BOUNDS

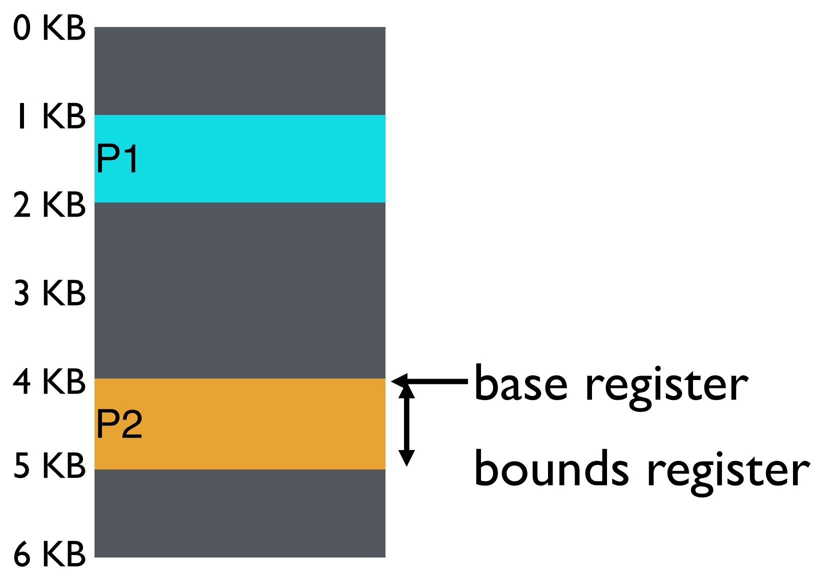
Translation on every memory access of user process

- 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

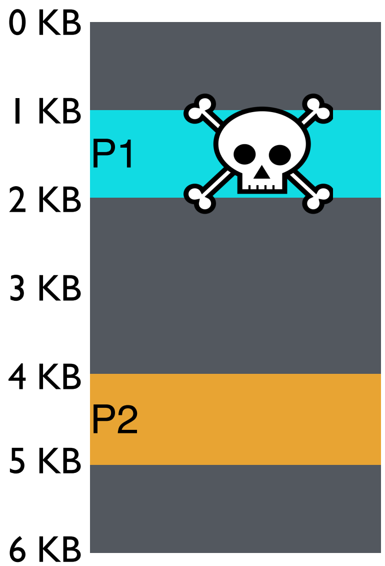




P1 is running



P2 is running



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	Interrupt OS!

Can P1 hurt P2?

MANAGING PROCESSES WITH BASE AND BOUNDS

Context-switch: Add base and bounds registers to PCB (process-control-block)

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

What if don't change base and bounds registers when switch? Threads!

Protection requirement

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

BASE AND BOUNDS ADVANTAGES

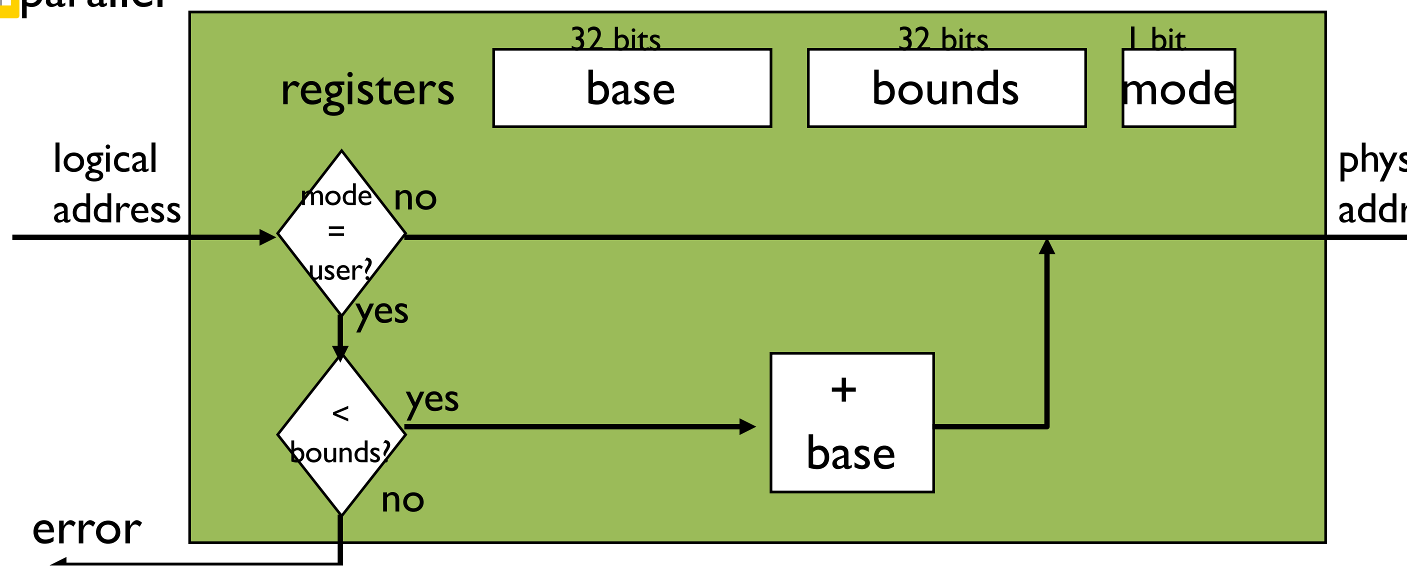
Provides protection (both read and write) across address spaces

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

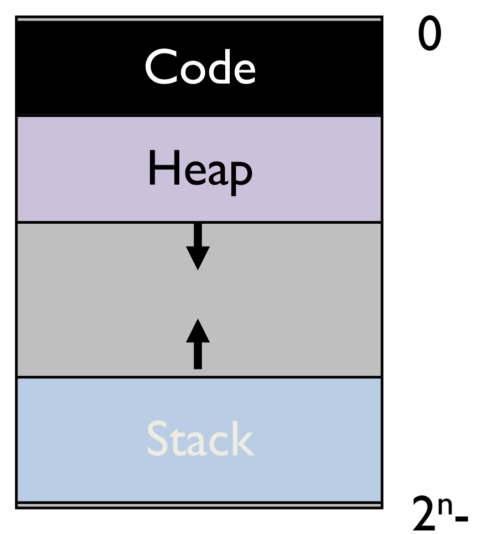


BASE AND BOUNDS DISADVANTAGES

Disadvantages

- Each process must be allocated contiguously in physical memory
- Must 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

TO DO

- Project 1 Due Last Night
- Project 2 Available: Due Monday
- Discussion section: xv6 code walk through!
- Homeworks:
  - Process (Due Thursday)
  - Scheduling and MLFQ (Due Tuesday)