

CS 537

Lecture 5

Memory

Michael Swift

1

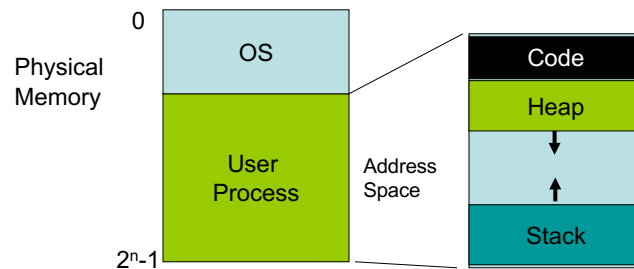
Questions answered in this lecture:

Questions answered in this lecture:

- What is in the address space of a process (review)?
- What are the different ways that that OS can virtualize memory?
 - Time sharing, static relocation, dynamic relocation
 - (base, base + bounds, segmentation)
- What hardware support is needed for dynamic relocation?

Motivation for Virtualization

Uniprogramming: One process runs at a time



Disadvantages:

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

Multiprogramming Goals

Transparency

- Processes are not aware that memory is shared
- Works regardless of number and/or location of processes

Protection

- Cannot corrupt OS or other processes
- Privacy: Cannot read data of other processes

Efficiency

- Do not waste memory resources (minimize fragmentation)

Sharing

- Cooperating processes can share portions of address space

Abstraction: Address Space

Address space: Each process has set of addresses that map to bytes

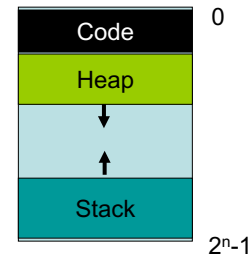
Problem:

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

Review: What is in an address space?

Address space has static and dynamic components

- 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

Definition: Memory is freed in opposite order from allocation

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

Simple and efficient implementation:

Pointer separates allocated and freed space

Allocate: Increment pointer

Free: Decrement pointer

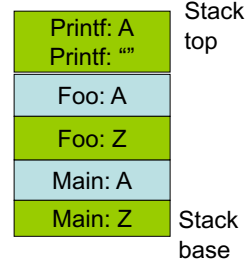
No fragmentation

Where Are stacks Used?

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

```
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);  
}
```

Stack ptr →



Heap Organization

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

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

Advantage

- Works for all data structures

Disadvantages

- Allocation can be slow
- End up with 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



Quiz: Match that Address Location

```
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	Static data
main	Code
y	Stack
z	Stack
*z	Heap

Memory Accesses

```
#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

Quiz: Memory Accesses?

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?

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?

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
5. Segmentation

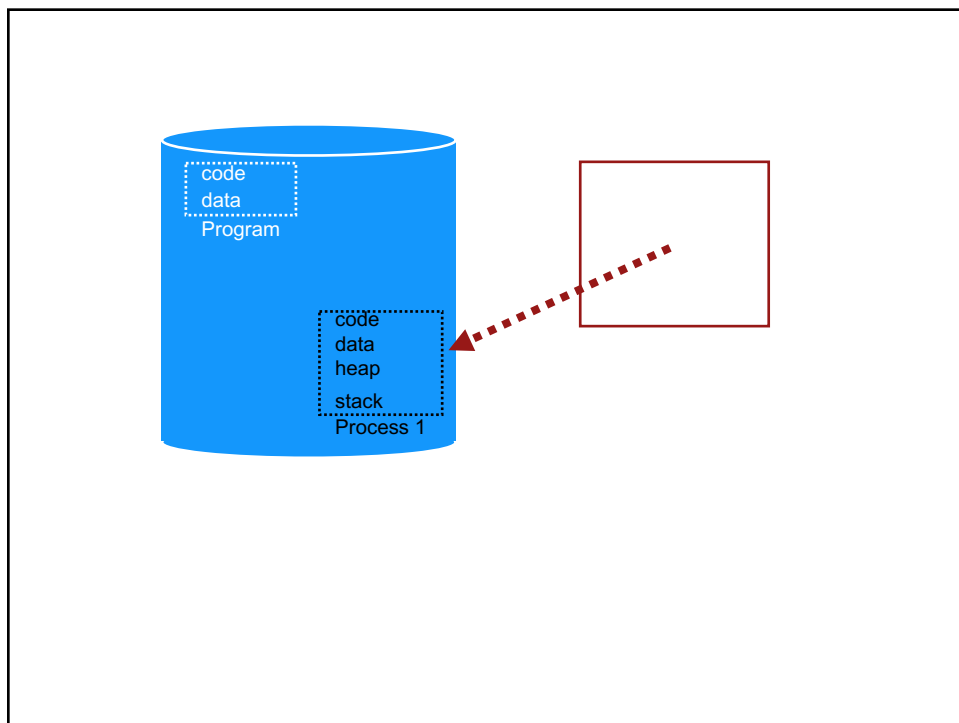
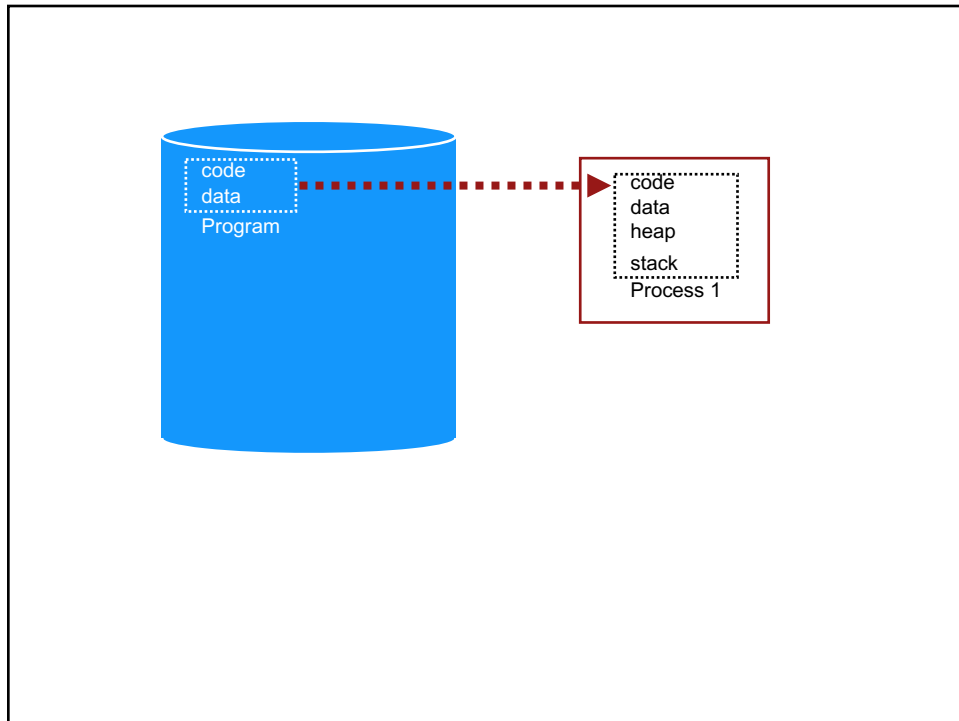
1) Time Sharing of Memory

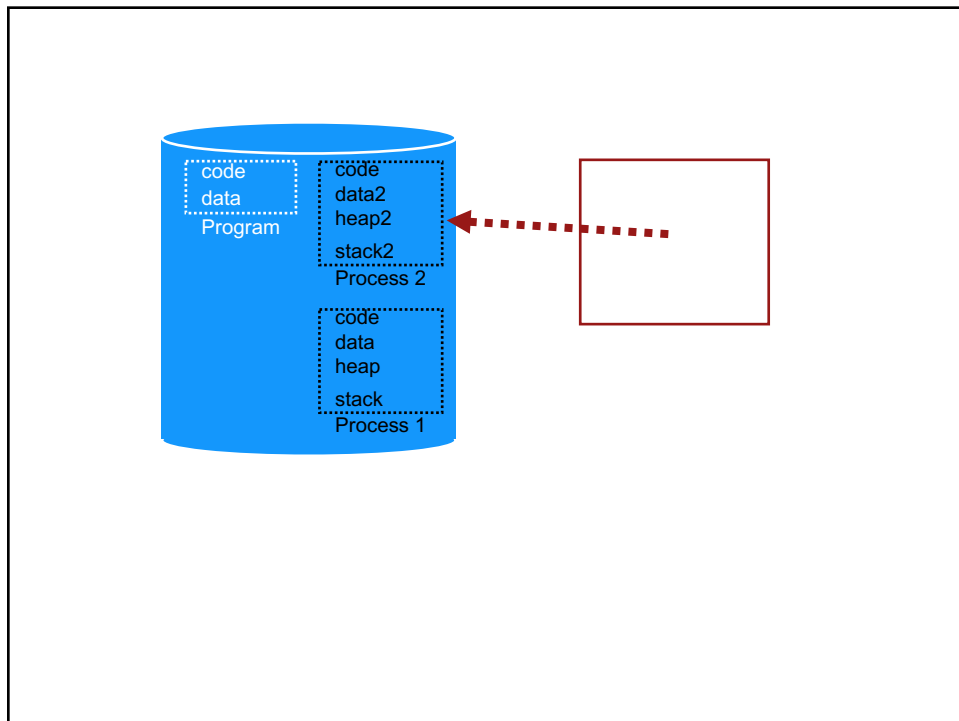
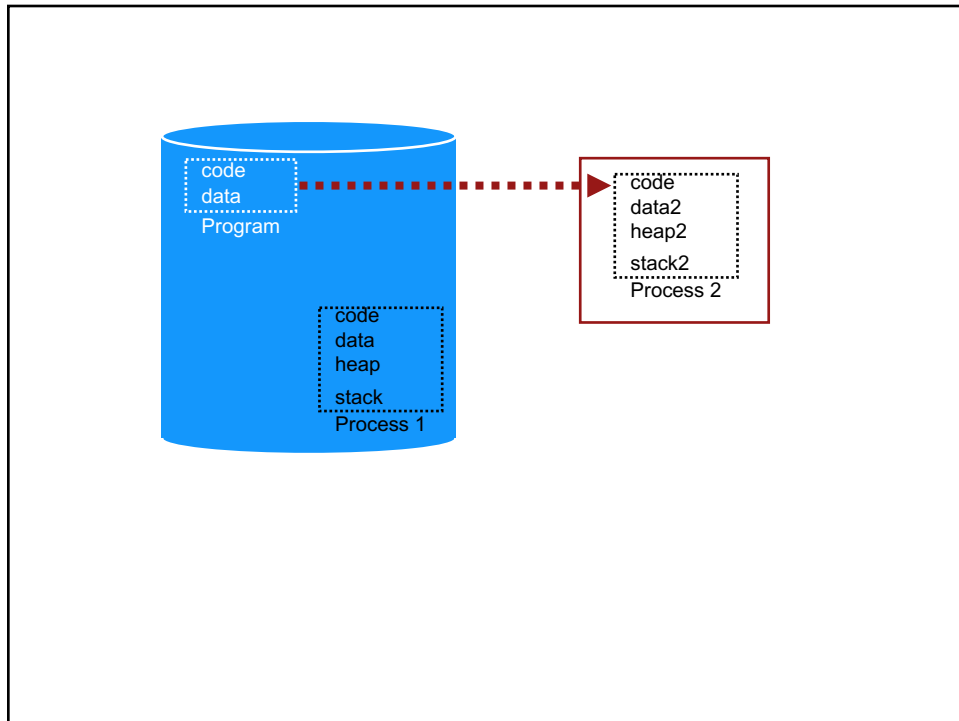
Try similar approach to how OS virtualizes CPU

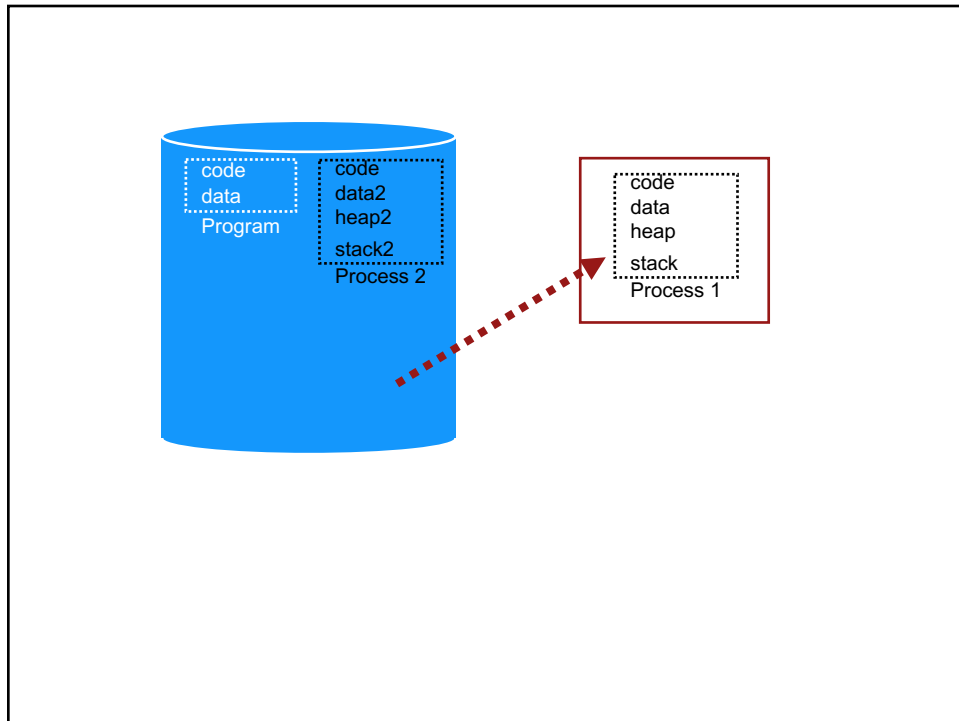
Observation:

OS gives illusion of many virtual CPUs by saving **CPU registers to memory** when a process isn't running

Could give illusion of many virtual memories by saving **memory to disk** when process isn't running







Problems with Time Sharing Memory

Problem: 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
rewrite 0x1019: movl %edi, 0x8(%rbp)

• 0x10:movl0x8(%rbp), %edi
• 0x13:addl$0x3, %edi
• 0x19:movl%edi, 0x8(%rbp)

                                0x3010: movl 0x8(%rbp), %edi
                                0x3013: addl $0x3, %edi
rewrite 0x3019: movl %edi, 0x8(%rbp)
```

Static: Layout in Memory



Static Relocation: Disadvantages

No protection

- Process can destroy OS or other processes
- No privacy

Cannot move address space after it has been placed (need to rewrite again)

- May not be able to allocate new process

3) Dynamic Relocation

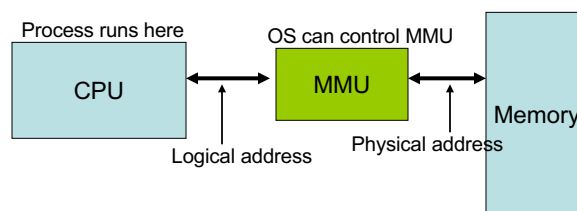
Goal: Protect processes from one another

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



Virtual Addresses

- To make it easier to manage memory of multiple processes, make processes use **virtual addresses**
 - virtual addresses are independent of location in physical memory (RAM) that referenced data lives
 - OS determines location in physical memory
 - instructions issued by CPU reference virtual addresses
 - e.g., pointers, arguments to load/store instruction, PC, ...
 - virtual addresses are translated by hardware into physical addresses (with some help from OS)

25

Hardware Support for Dynamic Relocation

Two operating modes

- Privileged (protected, kernel) mode: OS runs
 - When enter OS (trap, system calls, interrupts, exceptions)
 - Allows certain instructions to be executed
 - **For memory: 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**

Minimal MMU contains **base register** for translation

- base: start location for address space

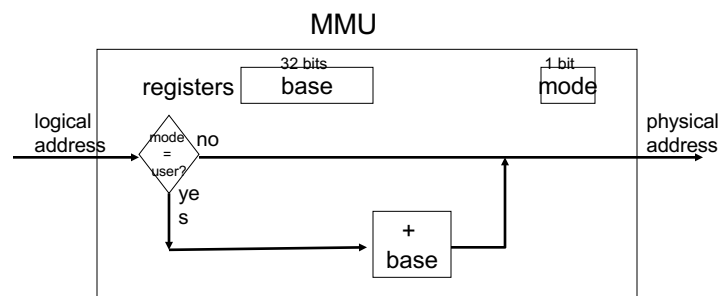
Dynamic Relocation with Base Register

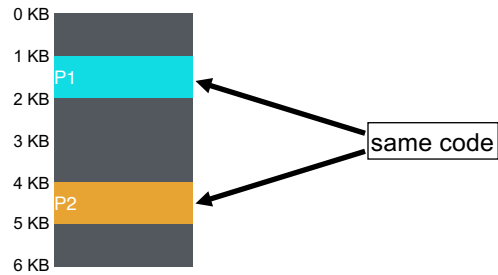
- Idea: translate virtual addresses to physical by adding a fixed offset each time.
- Store offset in base register
- Each process has different value in base register

Implementation of Dynamic Relocation: BASE REG

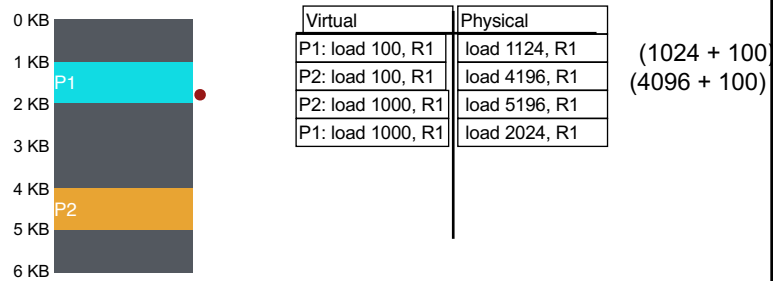
Translation on every memory access of user process

- MMU adds base register to logical address to form physical address





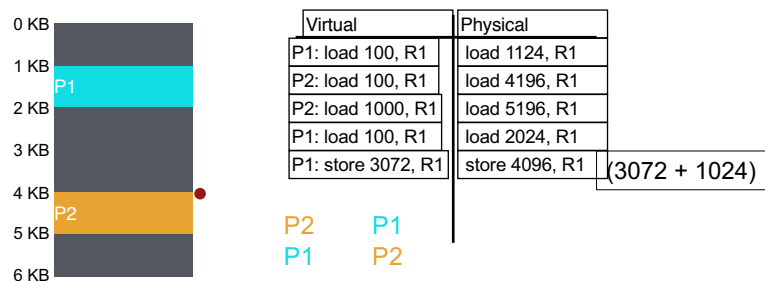
**VISUAL Example of DYNAMIC RELOCATION:
BASE REGISTER**



Quiz: Who Controls the Base Register?

What entity should **do translation of addresses** with base register?
(1) process, (2) OS, or (3) HW

What entity should **modify the base register**?
(1) process, (2) OS, or (3) HW



How well does dynamic relocation do with base register for protection?

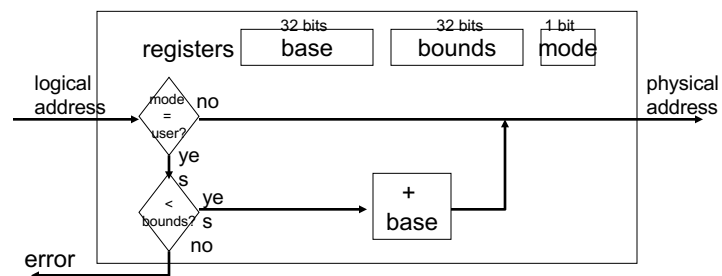
4) Dynamic with Base+Bounds

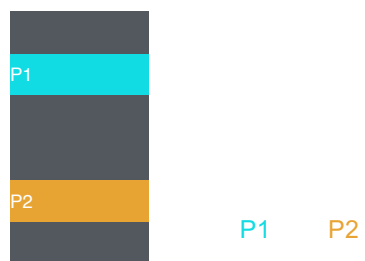
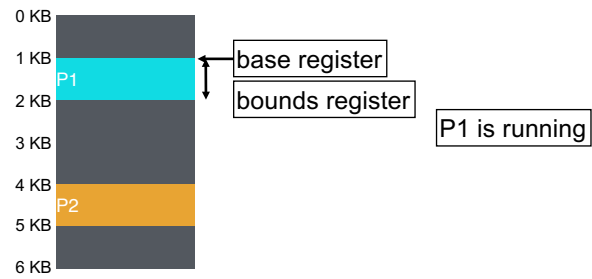
- Idea: limit the address space with a bounds register
- **Base register:** smallest physical addr (or starting location)
- **Bounds register:** size of this process's virtual address space
 - Sometimes defined as largest physical address (base + size)
- OS kills process if process loads/stores beyond bounds

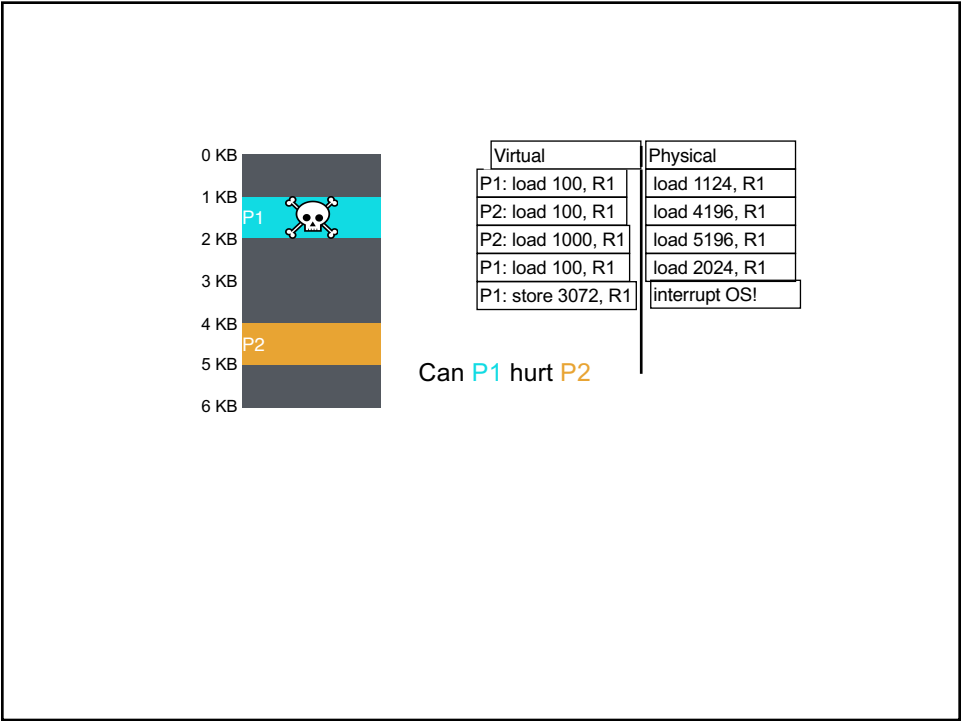
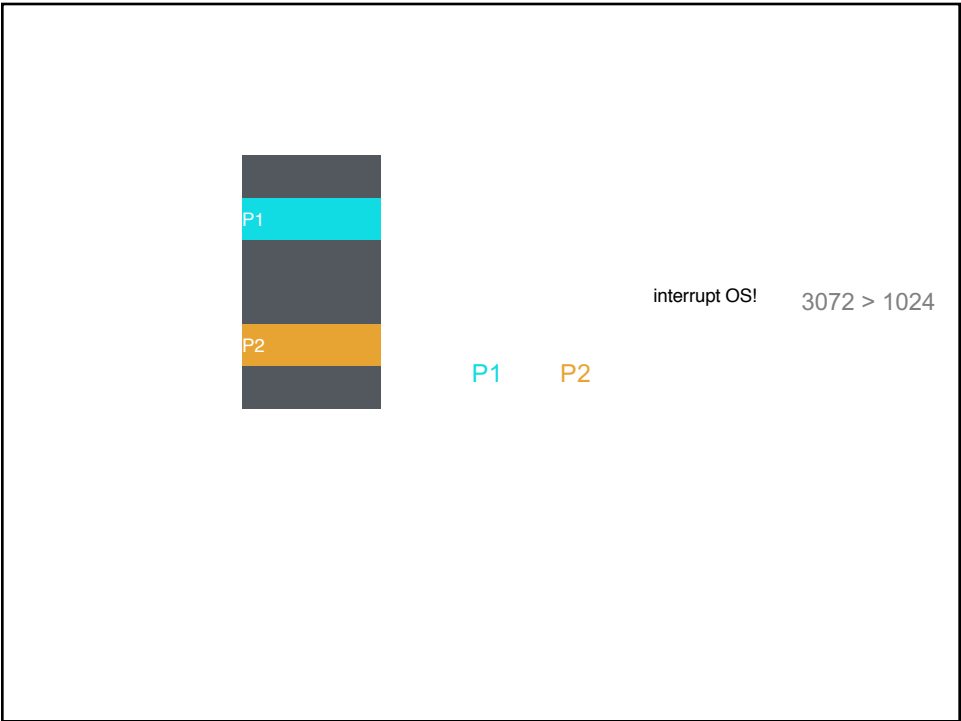
Implementation of BASE+BOUNDS

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







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

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

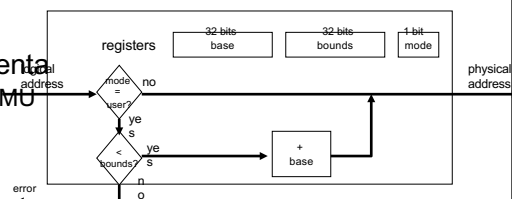
Protection requirement

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

Base and Bounds Advantages

Advantages

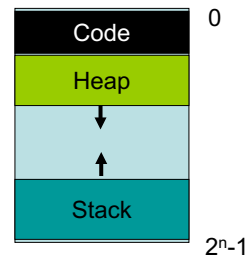
- Provides protection (both read and write) across address spaces
- Supports dynamic relocation
 - Can place process at different locations initially and also move address spaces
- Simple, inexpensive implementation
 - Few registers, little logic in MMU
- Fast
 - Add and compare in parallel



Base and Bounds DISADVANTAGES

Disadvantages

- Each process must be allocated contiguously in physical memory
 - Must allocate memory that may not be used by process
- No partial sharing: Cannot share limited parts of address space
 - Example: code



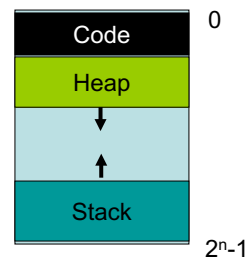
5) Segmentation

Divide address space into logical segments

- Each segment corresponds to logical entity in address space
 - code, stack, heap

Each segment can independently:

- be placed separately in physical memory
- grow and shrink
- be protected (separate read/write/execute protection bits)



Segmented Addressing

Process now specifies segment and offset within segment

How does process designate a particular segment?

- Use part of logical address
 - **Top bits** of logical address select segment
 - **Low bits** of logical address select offset within segment

What if small address space, not enough bits?

- Implicitly by type of memory reference
- Special registers

Address Translations with Segmentation

MMU contains Segment Table (per process)

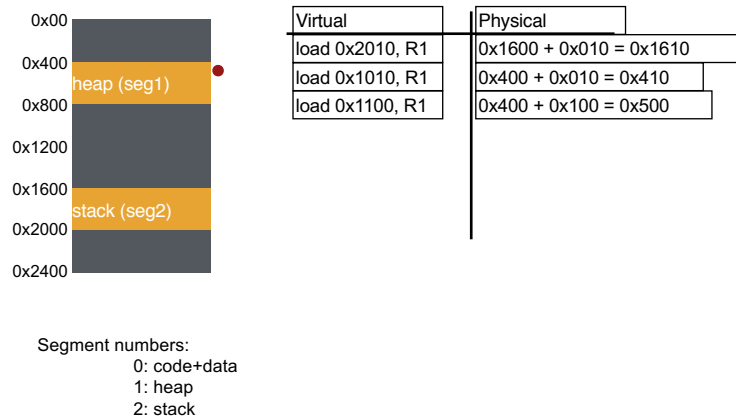
- Each segment has own base and bounds, protection bits
- Example: 14 bit logical address, 4 segments; how many bits for segment? How many bits for offset?

Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xffff	1 1
3	0x0000	0x000	0 0

remember:
1 hex digit->4 bits

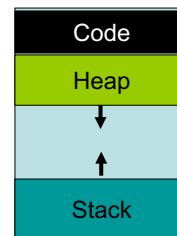
Translate logical addresses (in hex) to physical addresses

0x0240:
0x1108:
0x265c:
0x3002:



Advantages of Segmentation

- Enables sparse allocation of address space
 - Stack and heap can grow independently
 - Heap: If no data on free list, dynamic memory allocator requests more from OS (e.g., UNIX: malloc calls sbrk())
 - Stack: OS recognizes reference outside legal segment, extends stack implicitly
- Different protection for different segments
 - Read-only status for code
- Enables sharing of selected segments
- Supports dynamic relocation of each segment



Disadvantages of Segmentation

Each segment must be allocated contiguously

- May not have sufficient physical memory for large segments

Fix in next lecture with paging...

Conclusion

HW+OS work together to virtualize memory

- Give illusion of private address space to each process

Add MMU registers for base+bounds so translation is fast

- OS not involved with every address translation, only on context switch or errors

Dynamic relocation with segments is good building block

- Next lecture: Solve fragmentation with paging