CS5460: Operating Systems

Lecture 7: Address Spaces & Address Translation

(Chapters 13, 14, 15, 16)

Slide Credit: Andrea Arpaci-Dusseau

Assignment 2

- Due Tue Feb 16

More Virtualization

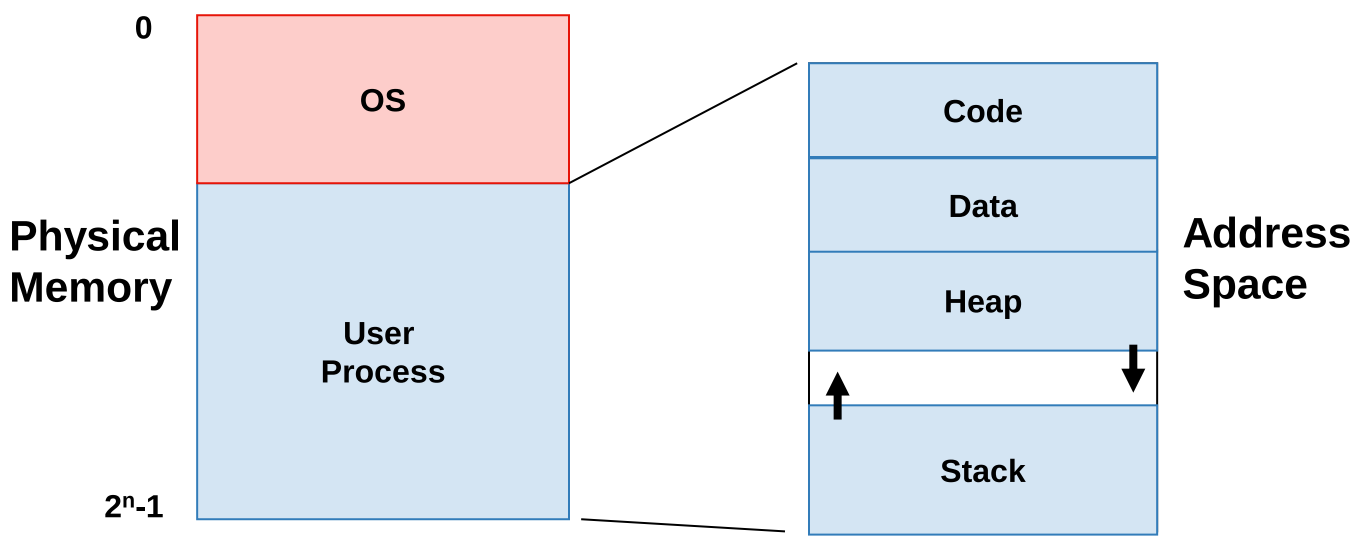
- 1st part of course: Virtualization

- Virtual CPU: illusion of private CPU registers

- Virtual RAM:
  illusion of private addresses and memory

Motivation for Virtualization

- Uniprogramming: One process runs at a time



Disadvantages:

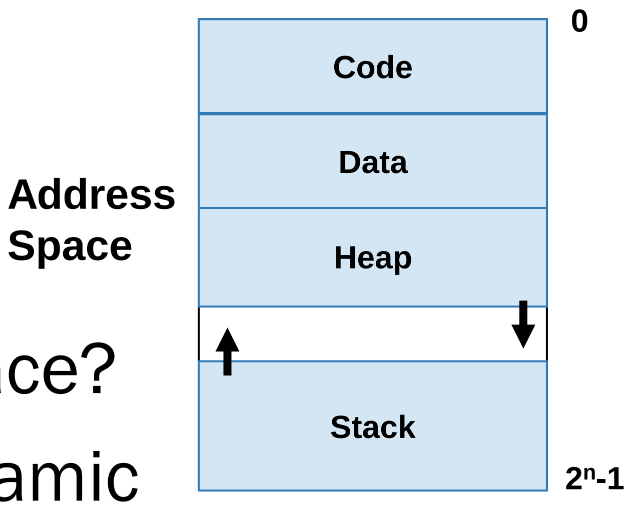
- Only one process runs at a time
- Process can destroy OS

Multiprogramming Goals

- Transparency
  - Processes are not aware that memory is shared
  - Works regardless of number and/or location of processes
- Protection
  - Secrecy: Cannot read data of OS or other processes
  - Integrity: Cannot change data of OS or other processes
- Efficiency
  - Do not waste memory resources (minimize fragmentation)
- Sharing
  - Cooperating processes can share portions of address space

Abstraction: Address Space

- Address space: process' set of addresses (that map to bytes)
  - How does 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 memory allocation?
  - Do not know amount of memory needed at compile time
  - Must be pessimistic when allocate memory statically
  - Allocate for worst case; storage 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

- Stack: memory freed in opposite order from alloc
  - 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
- “Automatic”: compiler adjusts stack pointer on entry/exit to calls to alloc/free space

Where Are Stacks Used?

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

```
void 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);
}
```

Heap Organization

- Heap: memory region where alloc/free are explicit
  - Heap consists of allocated areas and free areas (holes)
- Advantage
  - Alloc lifetime is independent of call/ret
  - Works for all data structures
- Disadvantages
  - Allocation can be slow
  - End up with small chunks of free space – fragmentation
  - Leaks (forgotten free), double free
- What is OS's role in managing heap?
  - OS gives big chunks free memory to process (sbrk/mmap); library manages individual allocations (malloc/free)

Quiz: Match that Address Location

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

Possible segments: data, code, stack, heap

Address	Location
x	
main	
y	
z	
*z	

Memory Accesses

```
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

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)

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

Virtualizing Memory

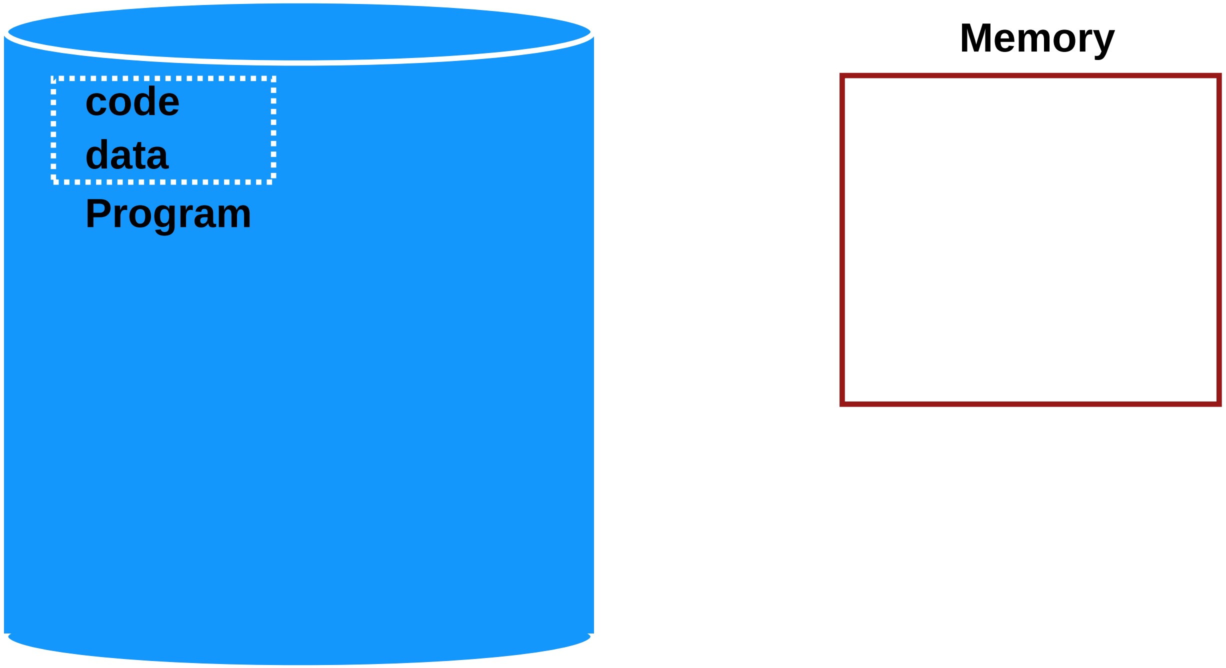
- How do we run multiple processes when addresses are “hardcoded” into process binaries?
- Possible solutions
  - Time Sharing
  - Static Relocation
  - Base & Bound
  - Segmentation
  - Paging (Coming Soon)

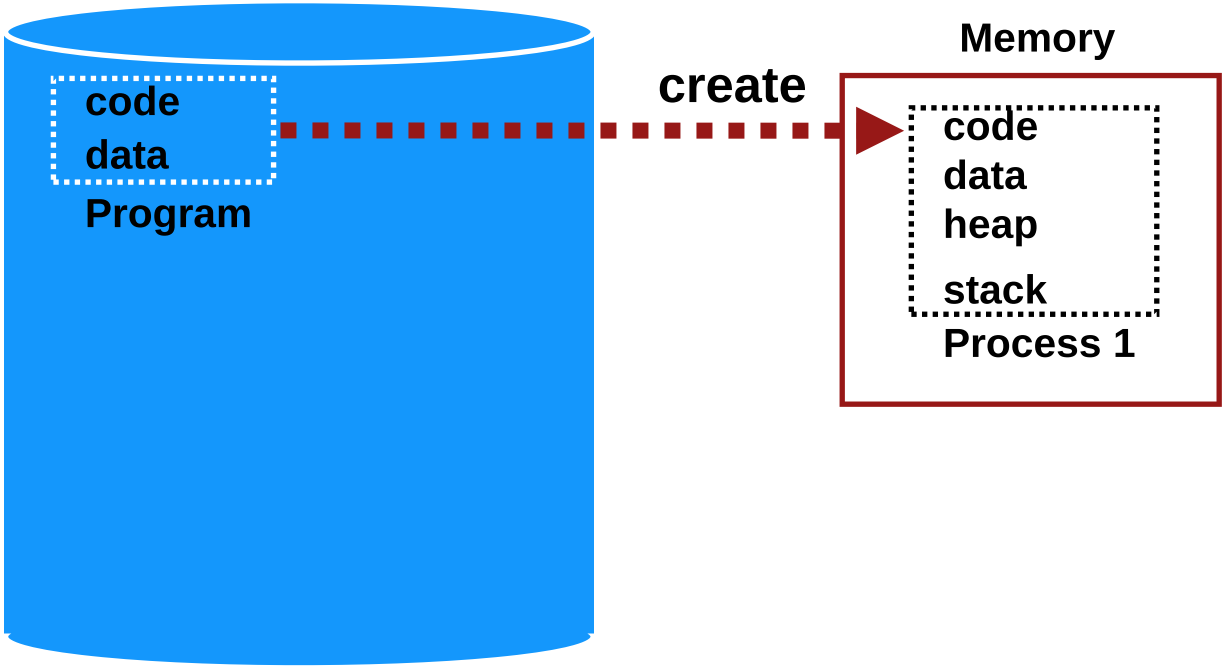
Time Sharing Memory

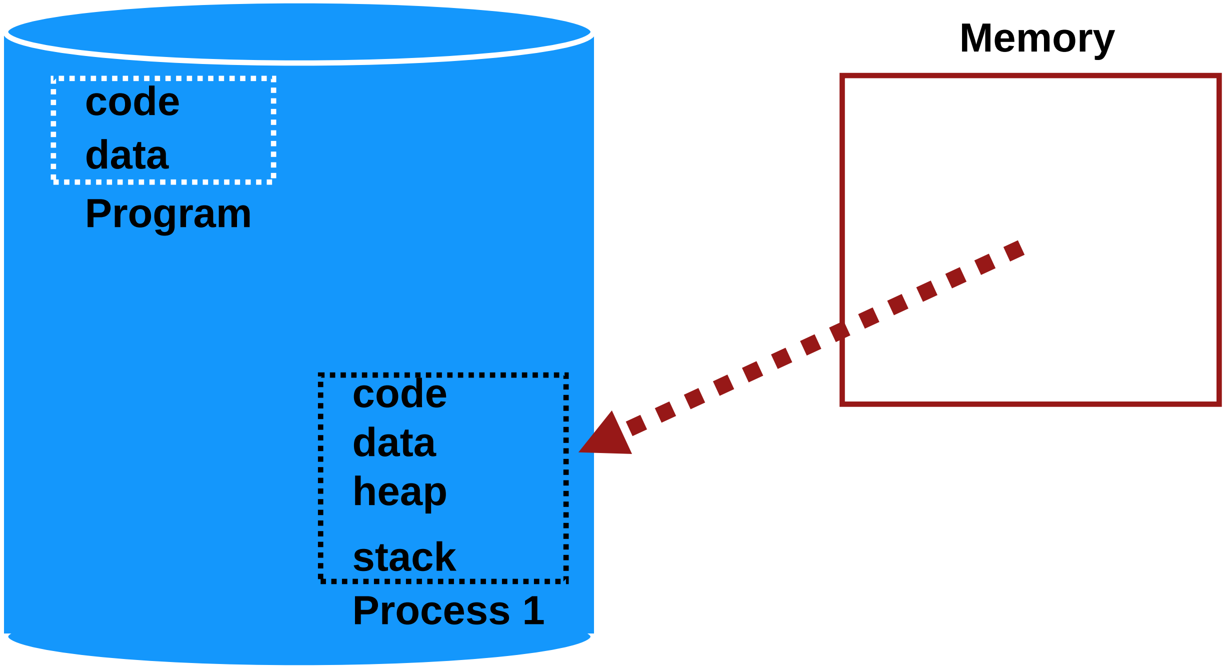
- Try similar approach to how OS virtualizes CPU

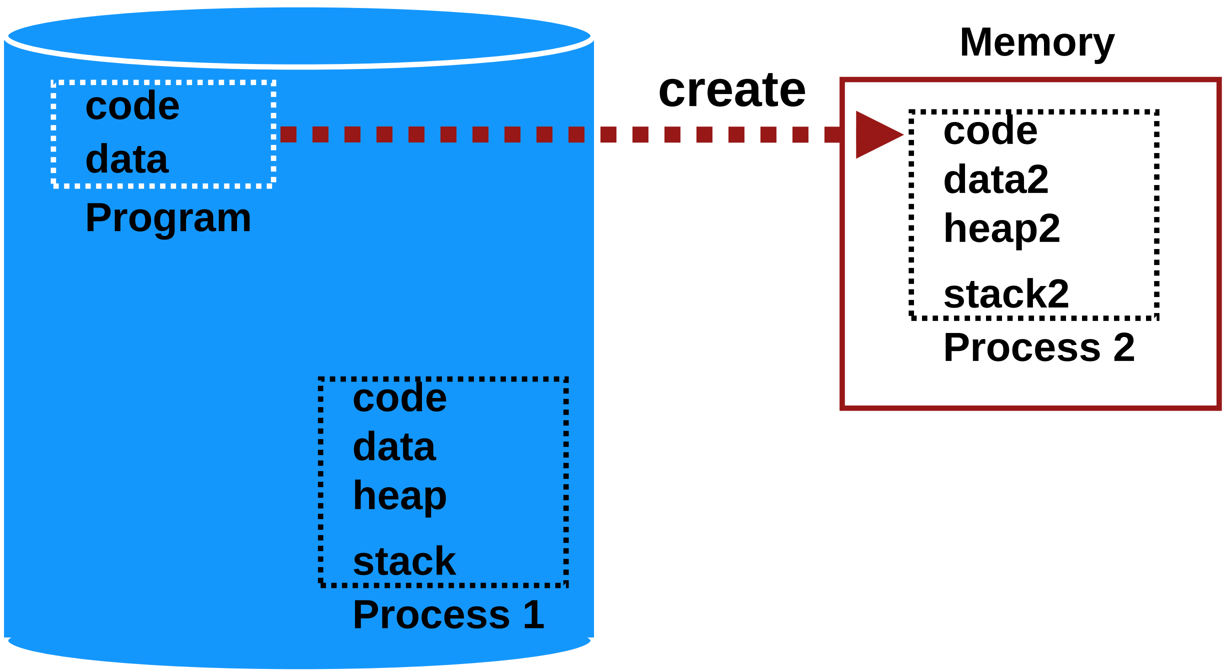
- Observation:
  OS gives illusion of many virtual CPUs by saving CPU registers to memory when a process isn’t running

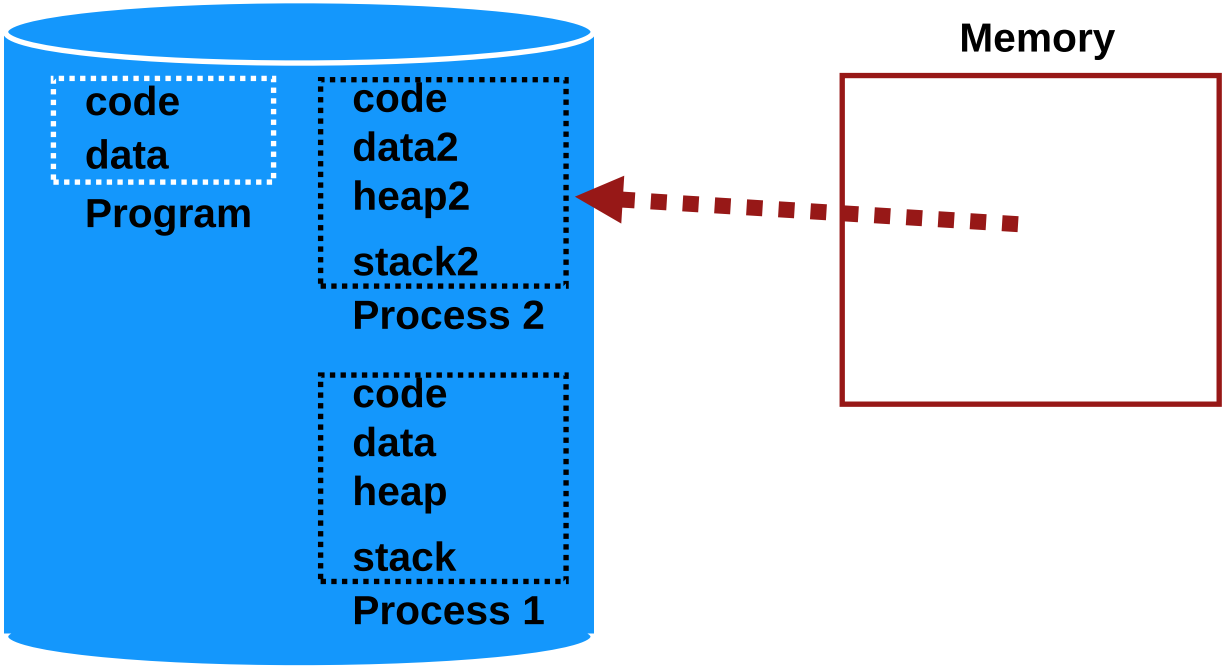
- Could give illusion of many virtual memories by saving memory to disk when process isn’t running

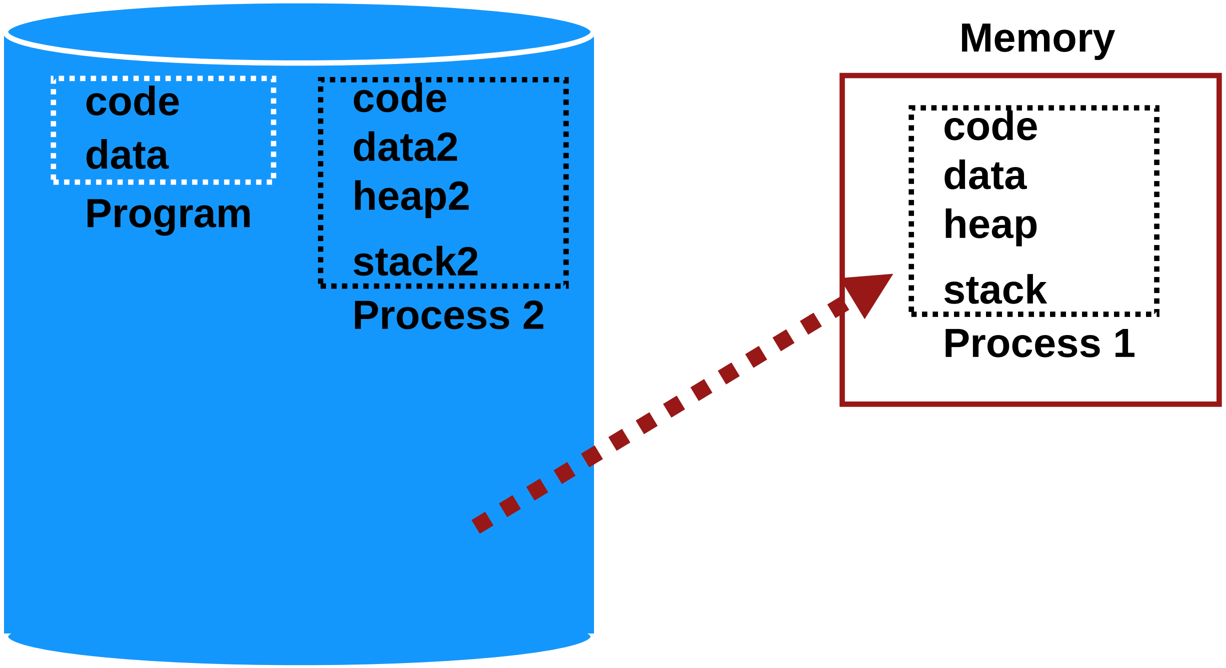

code
data
Program
Memory

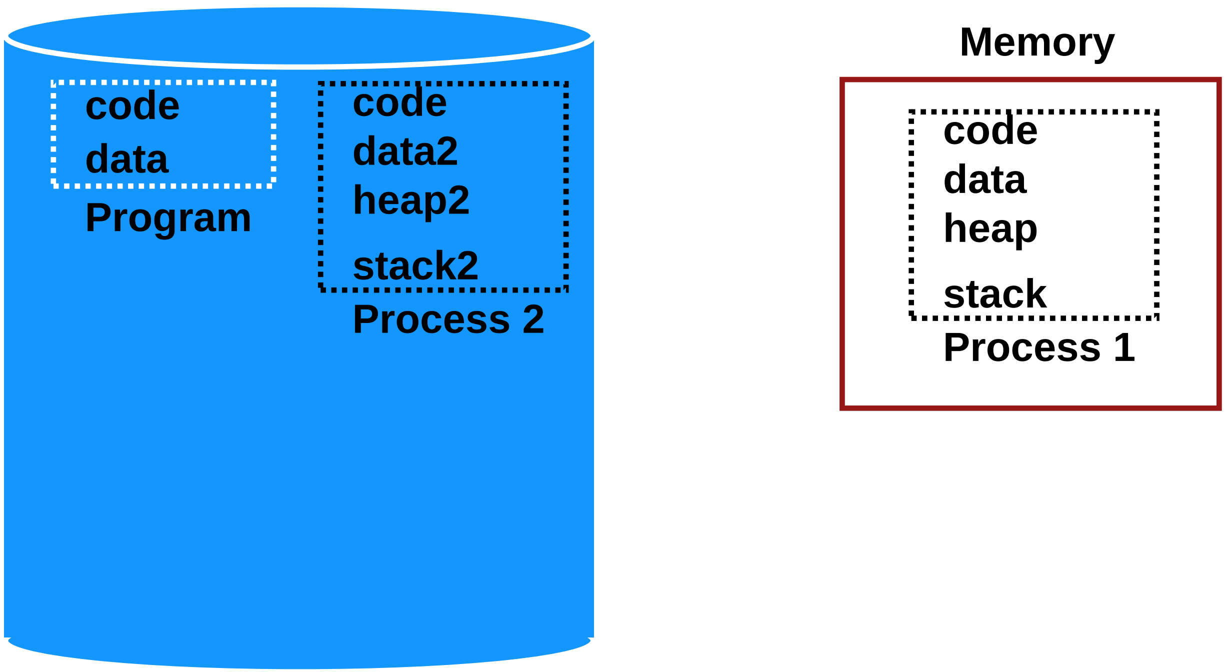

code
data
Program
create
Memory
code
data
heap
stack
Process 1


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


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

Time Sharing is No Good

- 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

Static Relocation

- Idea: OS rewrites programs before loading in memory
  - Make different process use different addresses/pointers
- Change jumps, loads of static data

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

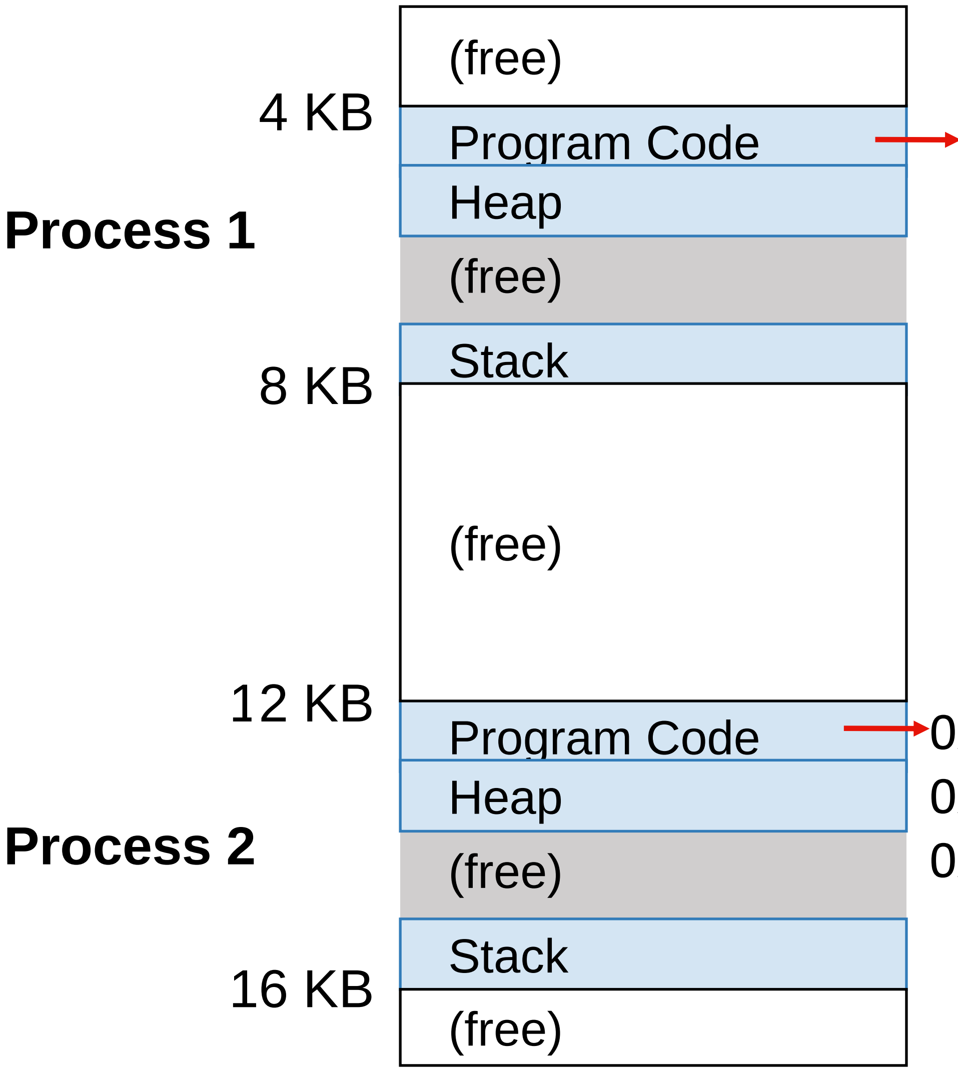
rewrite

```
0x10:   movl 0x8(%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 Physical 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)

Notice, OS didn't have to rewrite to deal with local stack locations

Static Relocation Problems

- No protection
  - No integrity: process can modify other processes, OS
  - No secrecy: process can access other processes, OS
- Cannot move address space after it has been placed
  - e.g. pointers in registers or on stack may refer to specific addresses, so can't easily rewrite while program running

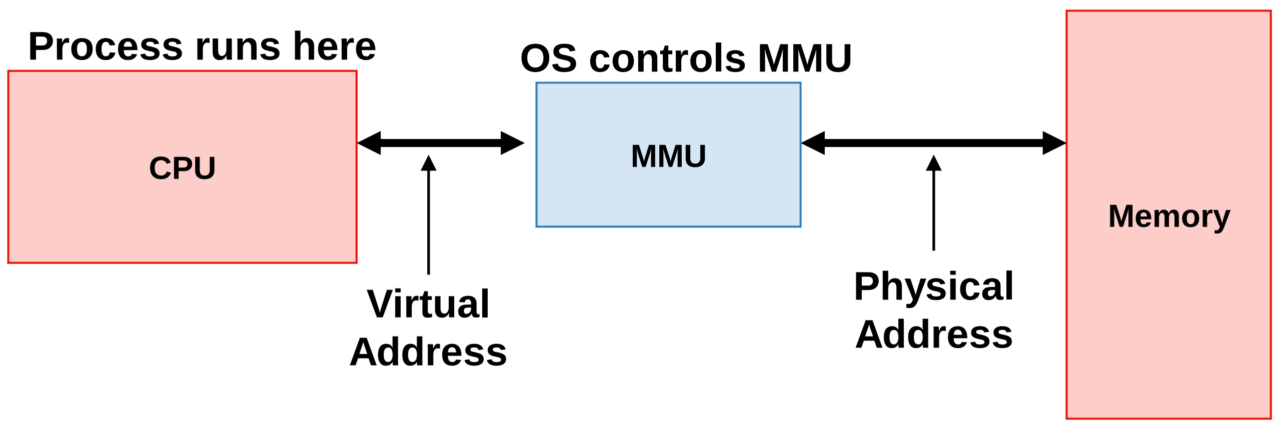
Dynamic Relocation

Goal: Relocation at runtime and protect processes from one another

Requires hardware support: Memory Management Unit (MMU)

MMU dynamically changes every process address at every memory reference

- Process generates logical or virtual addresses (in their address space)
- Memory hardware uses physical or real addresses



HW for Dynamic Relocation

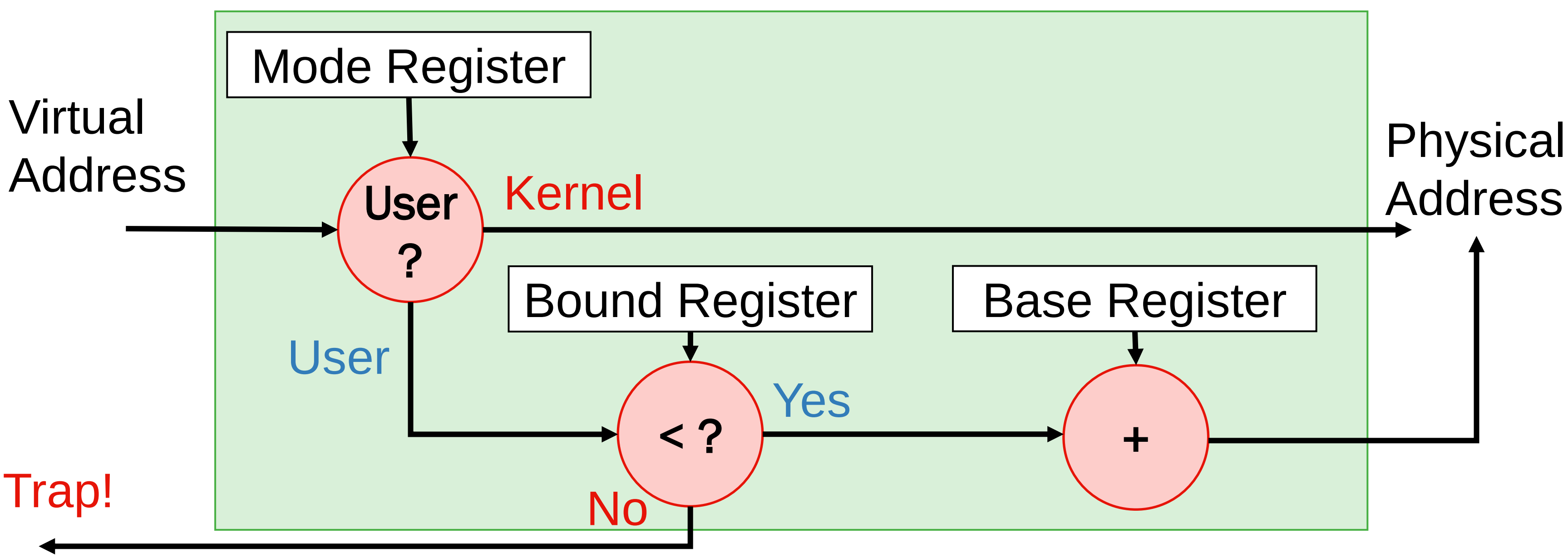
- All addresses automatically translated by MMU
  - Hardware does the translation

- OS sets up translation with privileged registers
  - Indicates where in physical memory the address space of the process starts (**base**)
  - And to what physical memory it extends (**bound**)

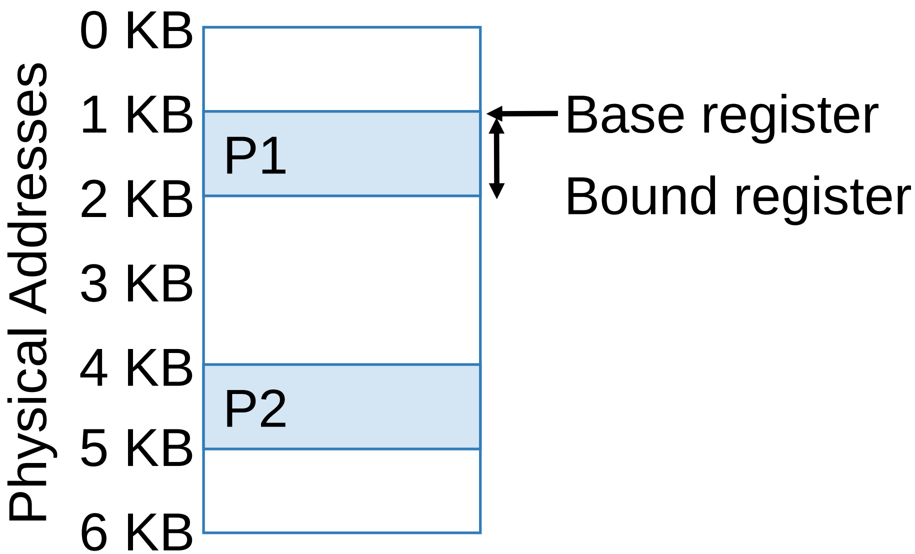
- User space process cannot change the registers
  - OS switches base/bound register at context switch

Base & Bound Implementation

- 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

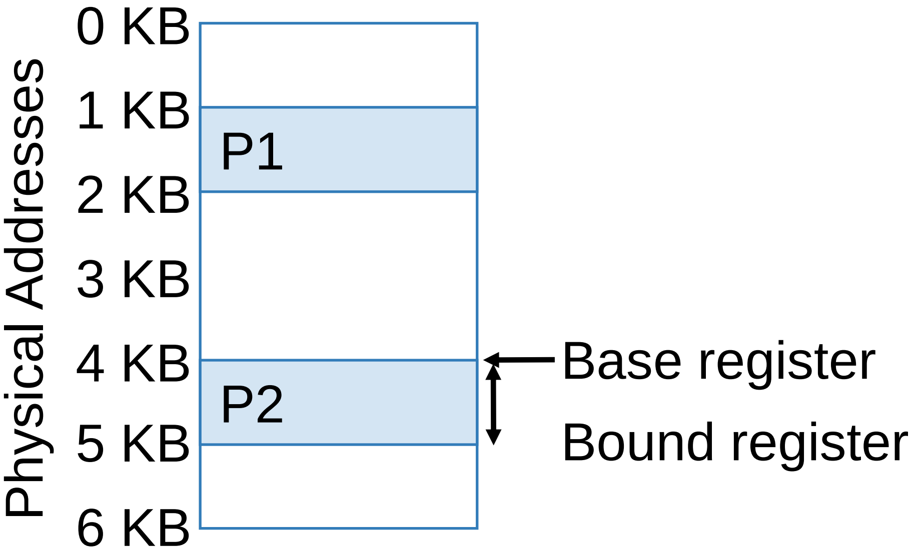


Base & Bound Example



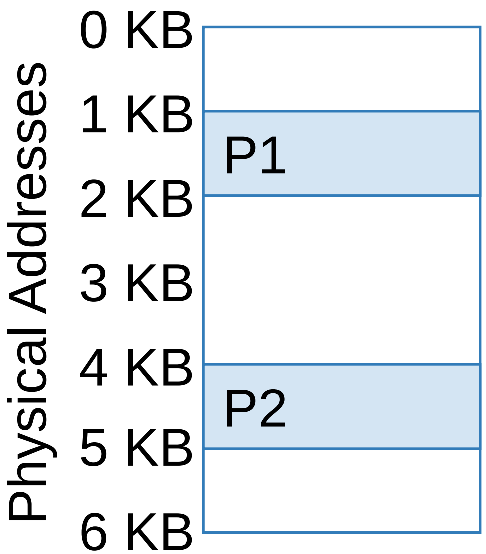
P1 is running

Base & Bound Example



P2 is running

Base & Bound Example



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 P1 hurt P2?

Base and Bounds Advantages

- Advantages
  - Protection (access/no access) across address spaces

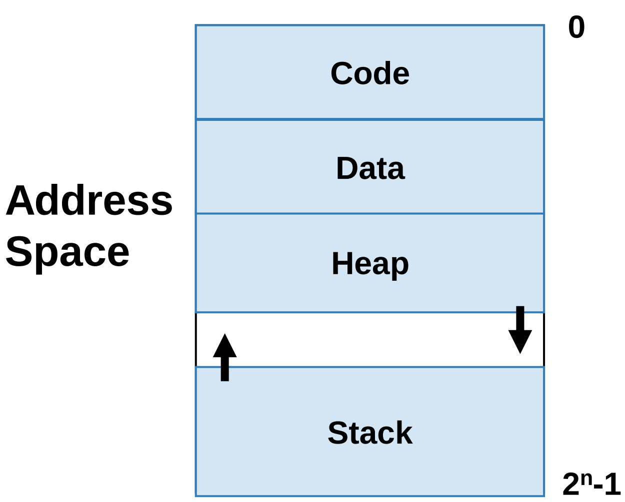
  - Supports dynamic relocation
    - Can place process at different locations initially
    - Also can move address spaces

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

  - Fast
    - Add and compare in parallel

Base and Bounds Problems

- 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
  - Can't control
    e.g. read/write/execute
    permissions



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