Memory Virtualization:
Time Sharing, Base+Bounds, Segmentation

OSTEP Chapters 13, (14), 15, 16:
http://pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf
http://pages.cs.wisc.edu/~remzi/OSTEP/vm-api.pdf
http://pages.cs.wisc.edu/~remzi/OSTEP/vm-mechanism.pdf
http://pages.cs.wisc.edu/~remzi/OSTEP/vm-segmentation.pdf

Jan Reineke
Universität des Saarlandes

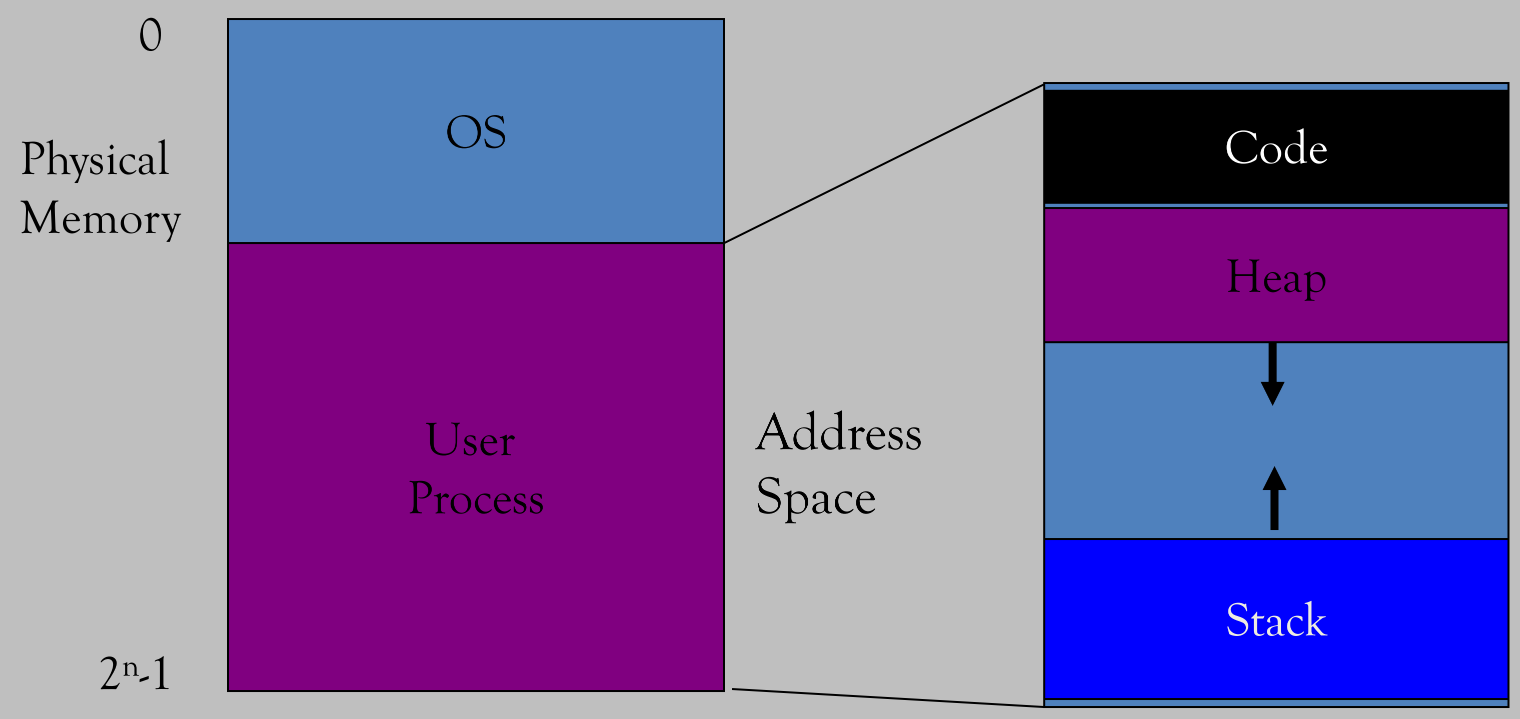
Virtualization

Virtual CPU:
illusion of **private CPU registers**

Virtual RAM:
illusion of **private memory**

Motivation for Virtualization

Uniprogramming: One process runs at a time



Disadvantages:

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

Memory Virtualization Goals

Transparency

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

Protection

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

Efficiency

- Do not waste memory resources (minimize fragmentation)

Sharing

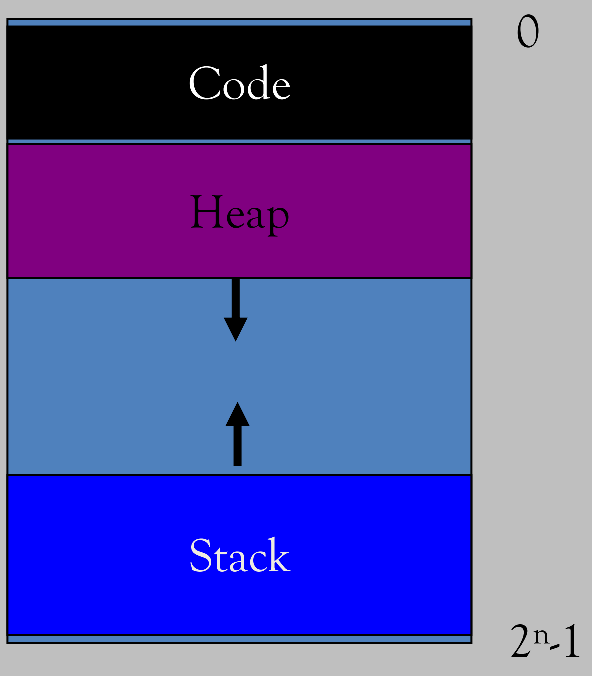
- Cooperating processes can share portions of address space

Abstraction: Address space

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

Address space has *static* and *dynamic* components:

- Static: Code and 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:
  often depends on program inputs
- Would have to statically allocate memory for the "worst case" → inefficient

Examples of dynamic memory allocation

Examples:

- Recursive procedures
- Complex data structures:
  lists, trees, hash maps, etc.

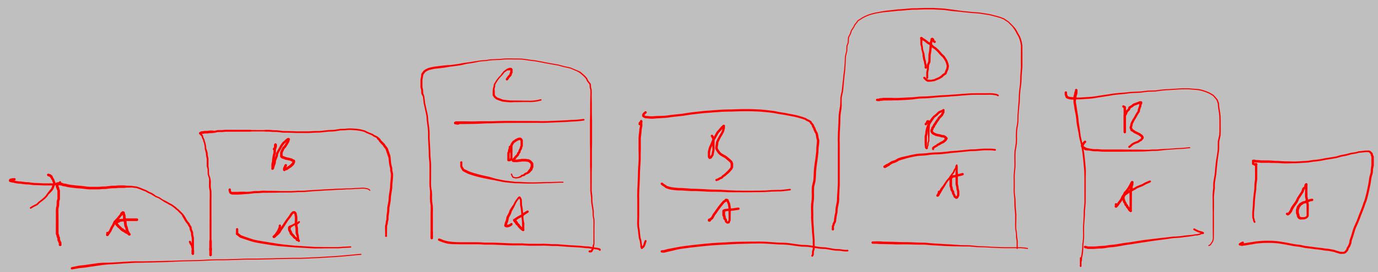
Two types of dynamic allocation:

- 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

- *Allocation*: Decrement pointer
- *Deallocation*: Increment pointer

Where are stacks used?

OS uses stack for procedure call frames:
local variables + parameters on the 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);
}
```

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 more complex, can be slow
- Fragmentation

Divison of work: OS + library

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

Quiz: Match that address allocation

```
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;
}
```

```
movl   -0x14(%rbp), %edi
addl   $0x3, %edi
movl   %edi, -0x14(%rbp)
```

(x86 Assembler)
%rbp is pointing to the base of the current stack frame

```
gcc -o exp exp.c
otool -tv exp
(or objdump under Linux)
```

Quiz: Memory accesses?

```
Initial:
%rip = 0x10 (PC)
%rbp = 0x200 (Basis des Stacks)

0x10: movl   -0x14(%rbp), %edi
0x13: addl   $0x3, %edi
0x19: movl   %edi, -0x14(%rbp)
```

1a) Instruction fetch at address 0x10
1b) Load from address 0x200 + -0x14 = 0x1EC

2a) Instruction fetch from address 0x13
2b) No memory access

3a) Instruction fetch at address 0x19
3b) Store to address 0x1EC

How to virtualize memory?

Problem: How to run multiple processes simultaneously?

Challenge: Addresses are "hardcoded" into process binaries

Possible solutions for mechanisms:

1. Time sharing
2. Static relocation
3. Dynamic relocation
4. Segmentation

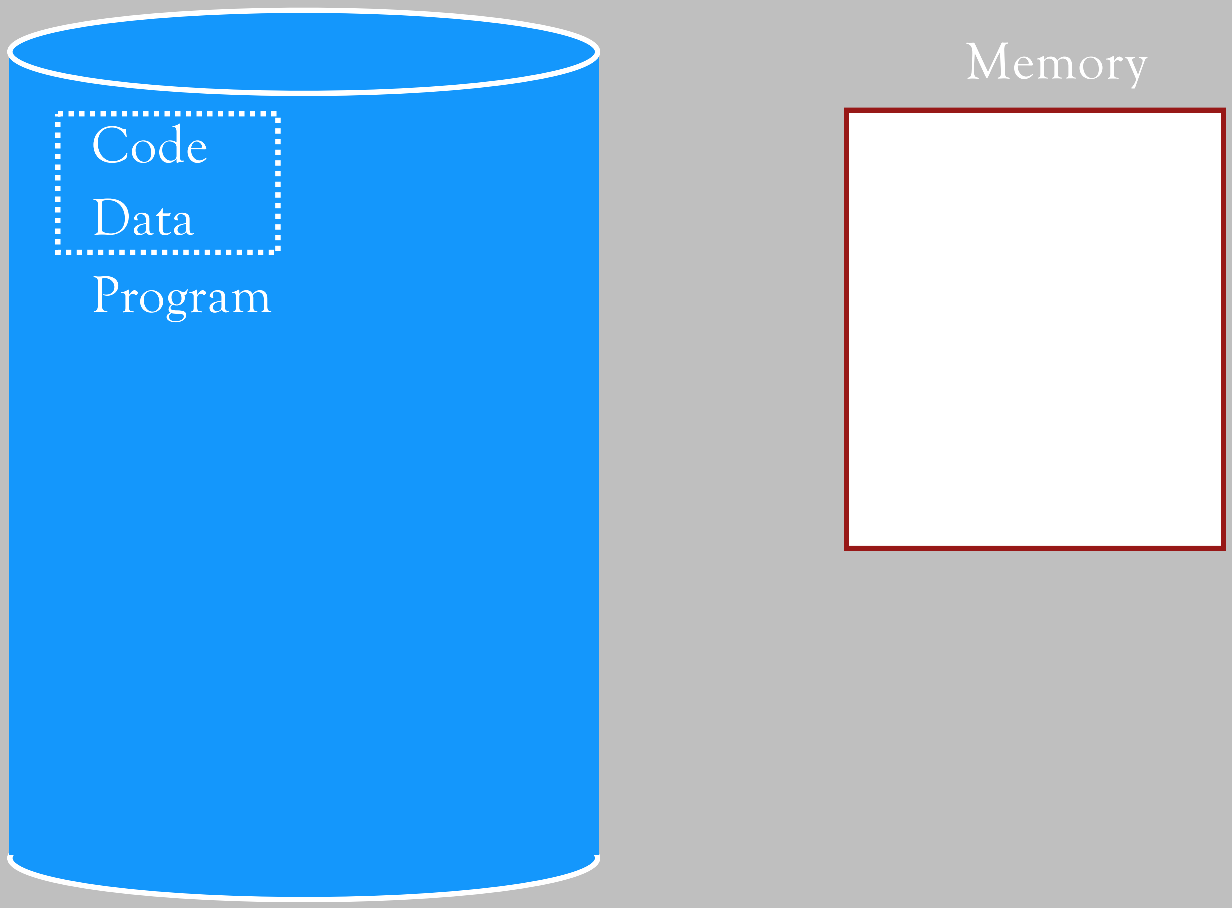
1. Time sharing of memory

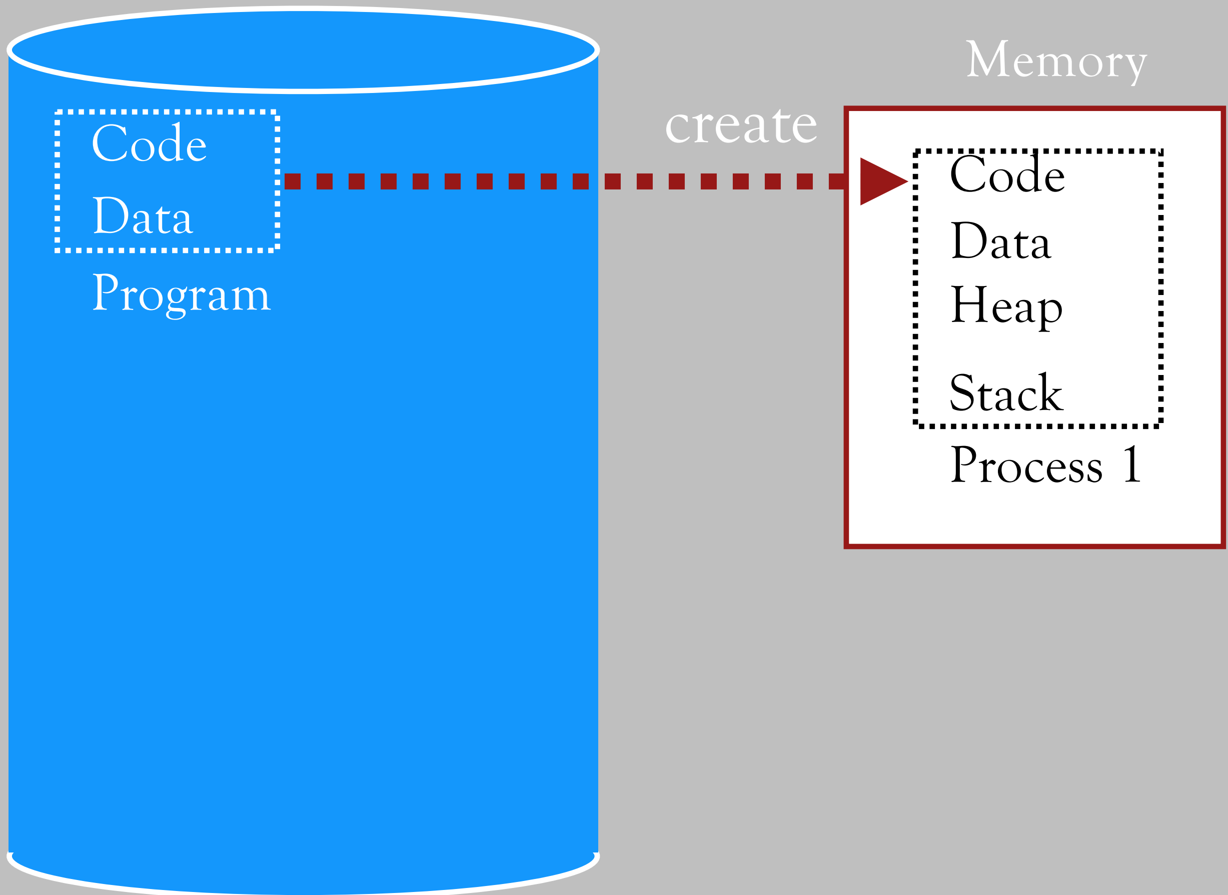
Try similar approach to how OS virtualizes CPU

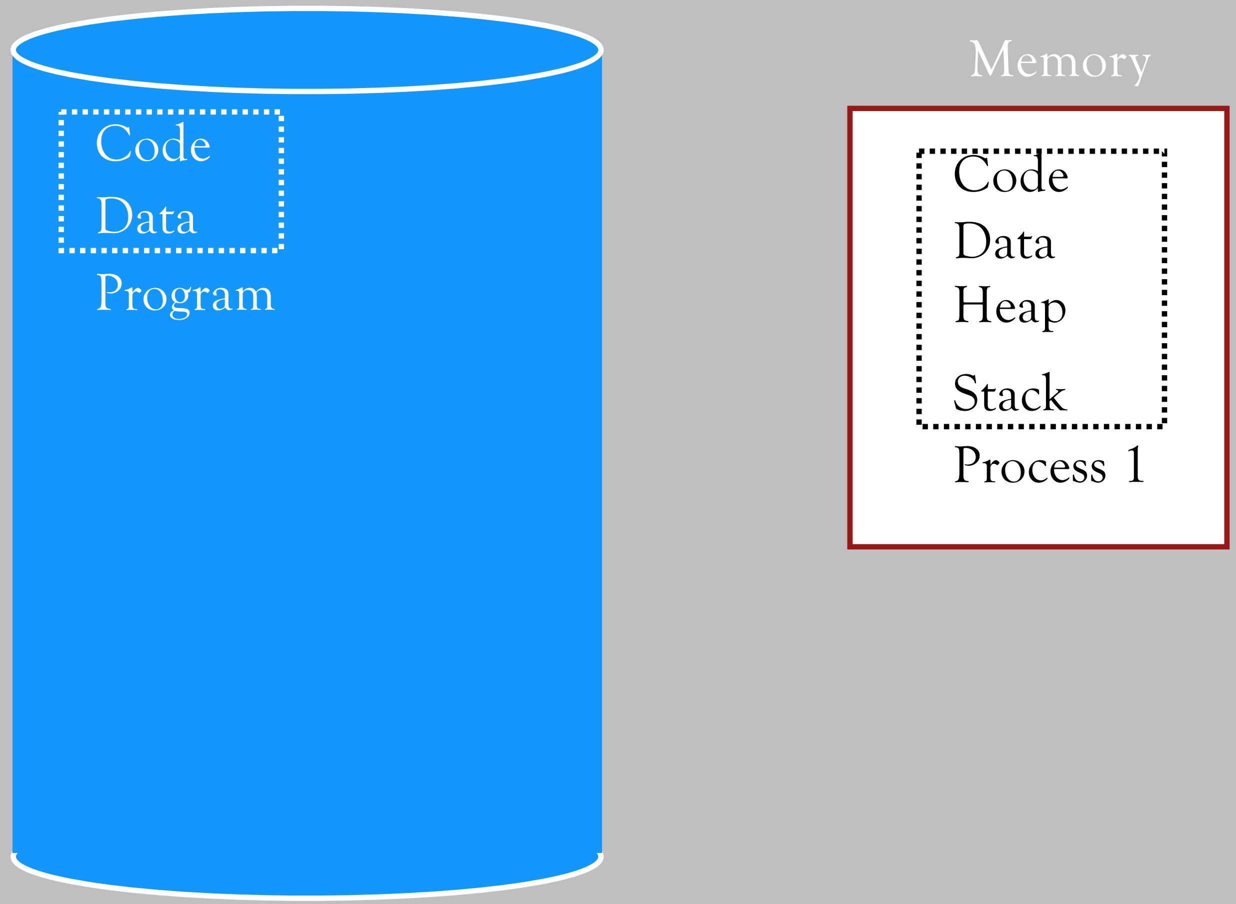
Observation: OS gives illusion of private CPUs by saving **CPU registers** to **memory** when a process isn't running

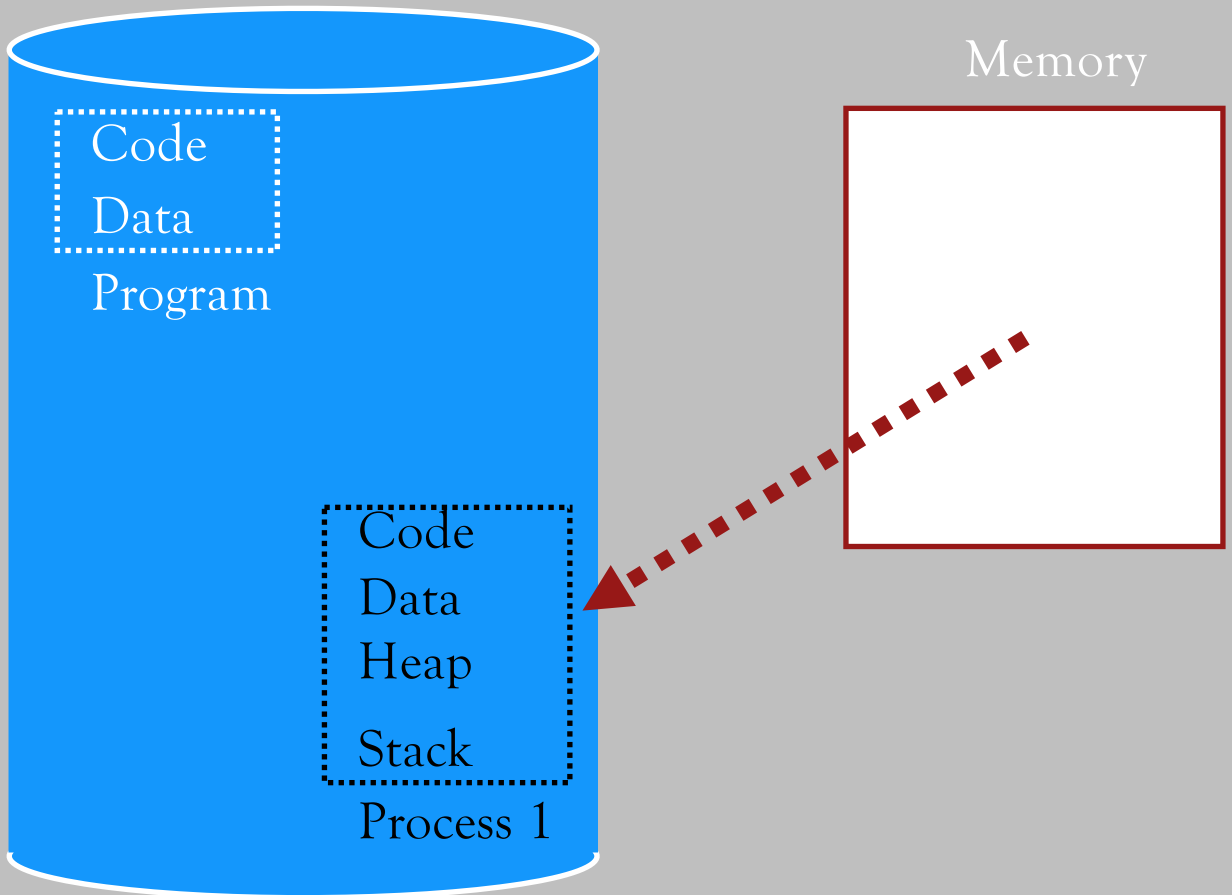
Approach:
Save memory contents to disk when process isn't running

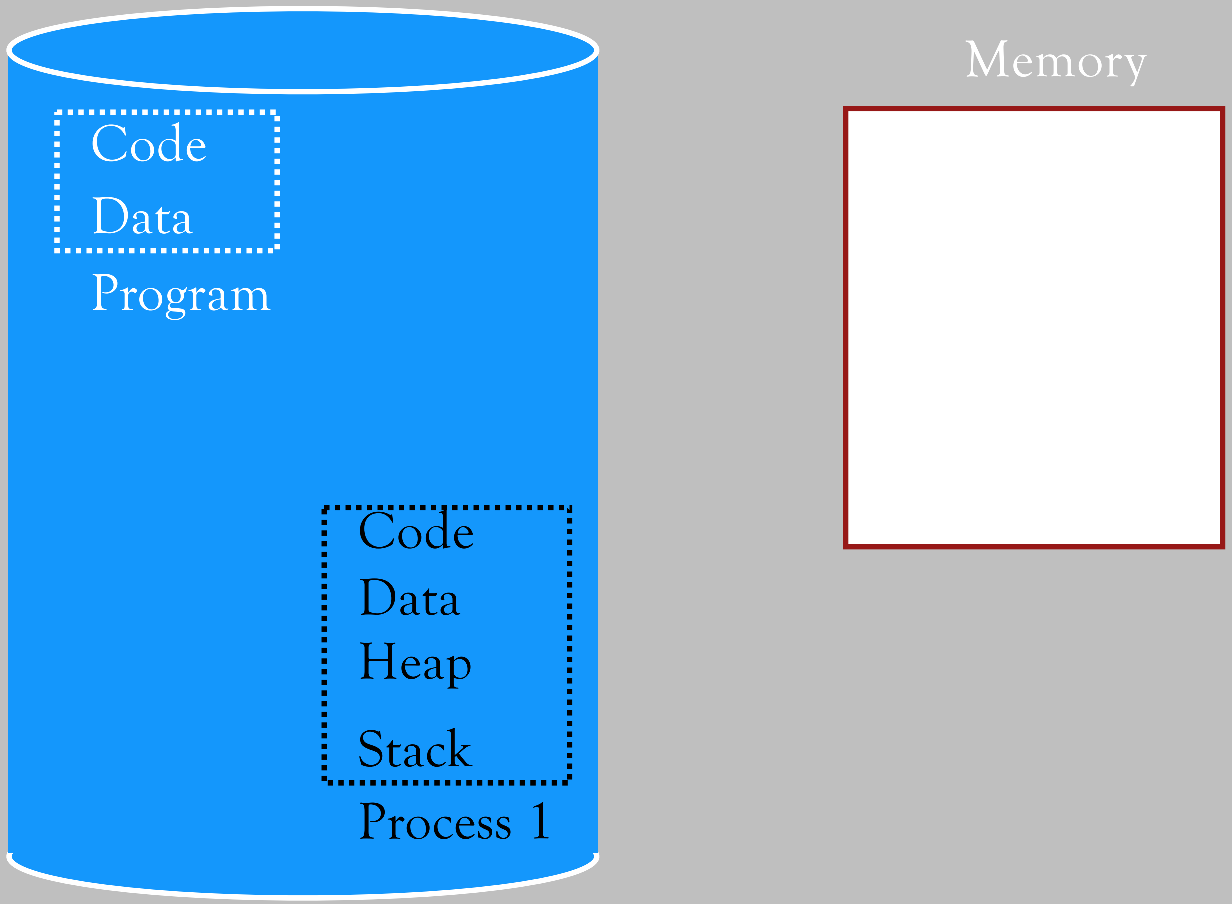
Example: Time sharing

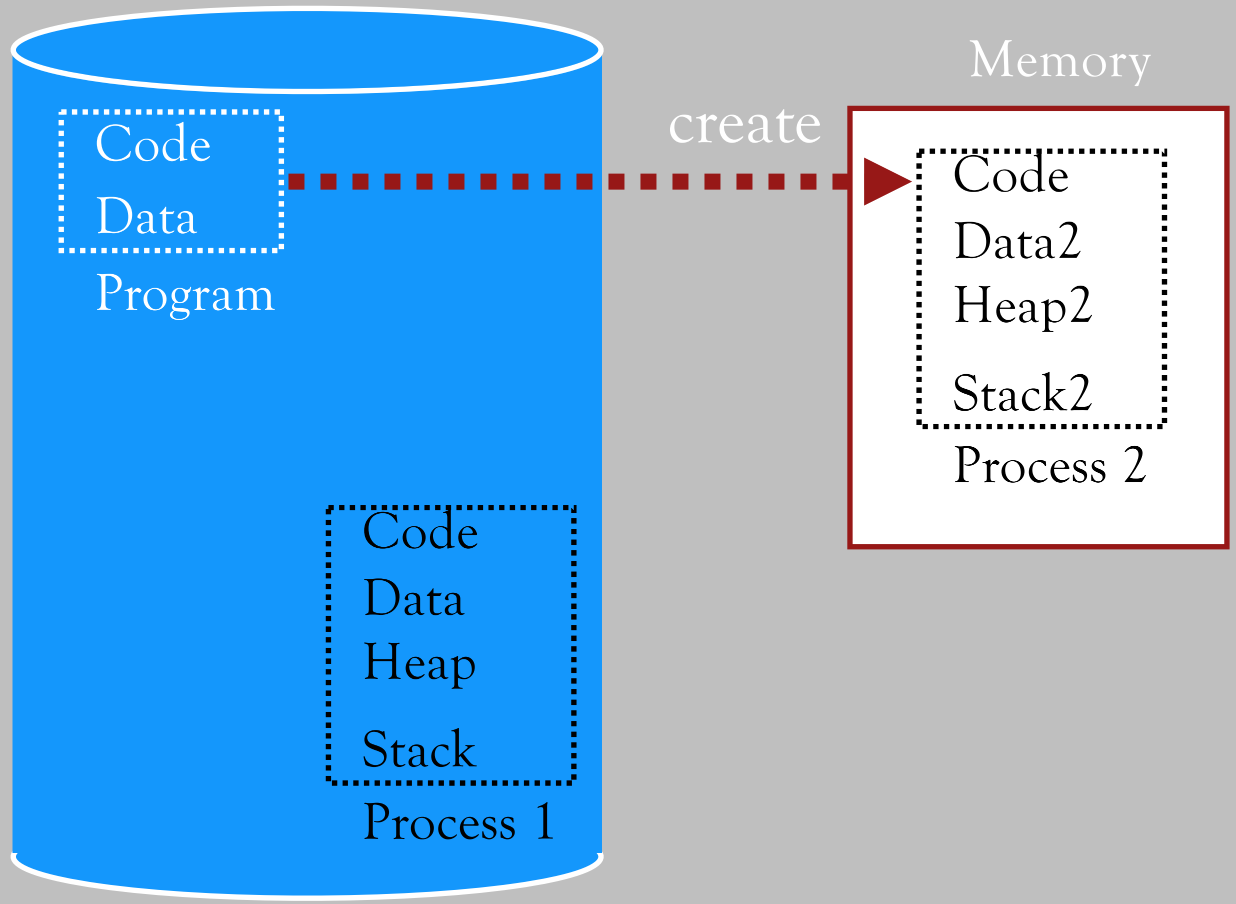


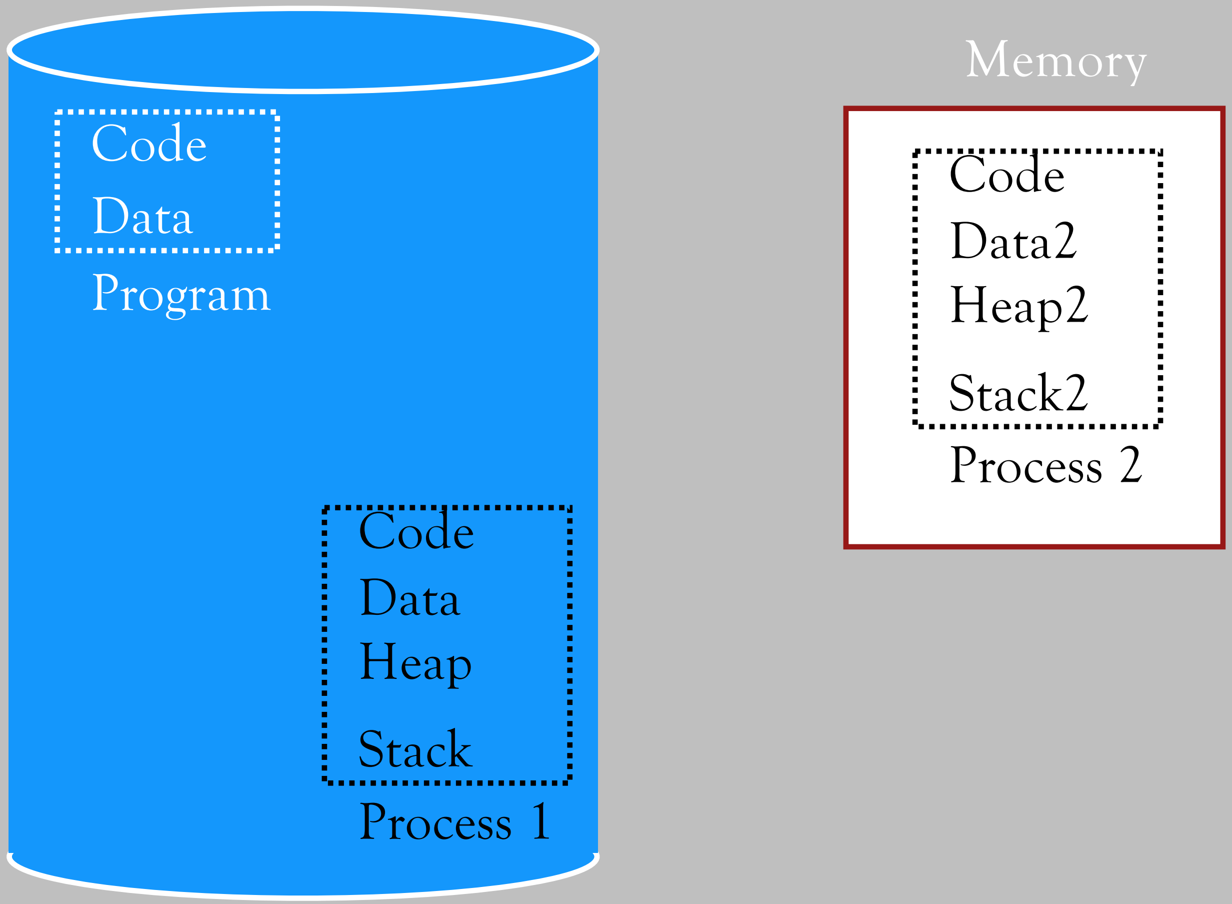

Code
Data
Program
create
Memory
Code
Data
Heap
Stack
Process 1

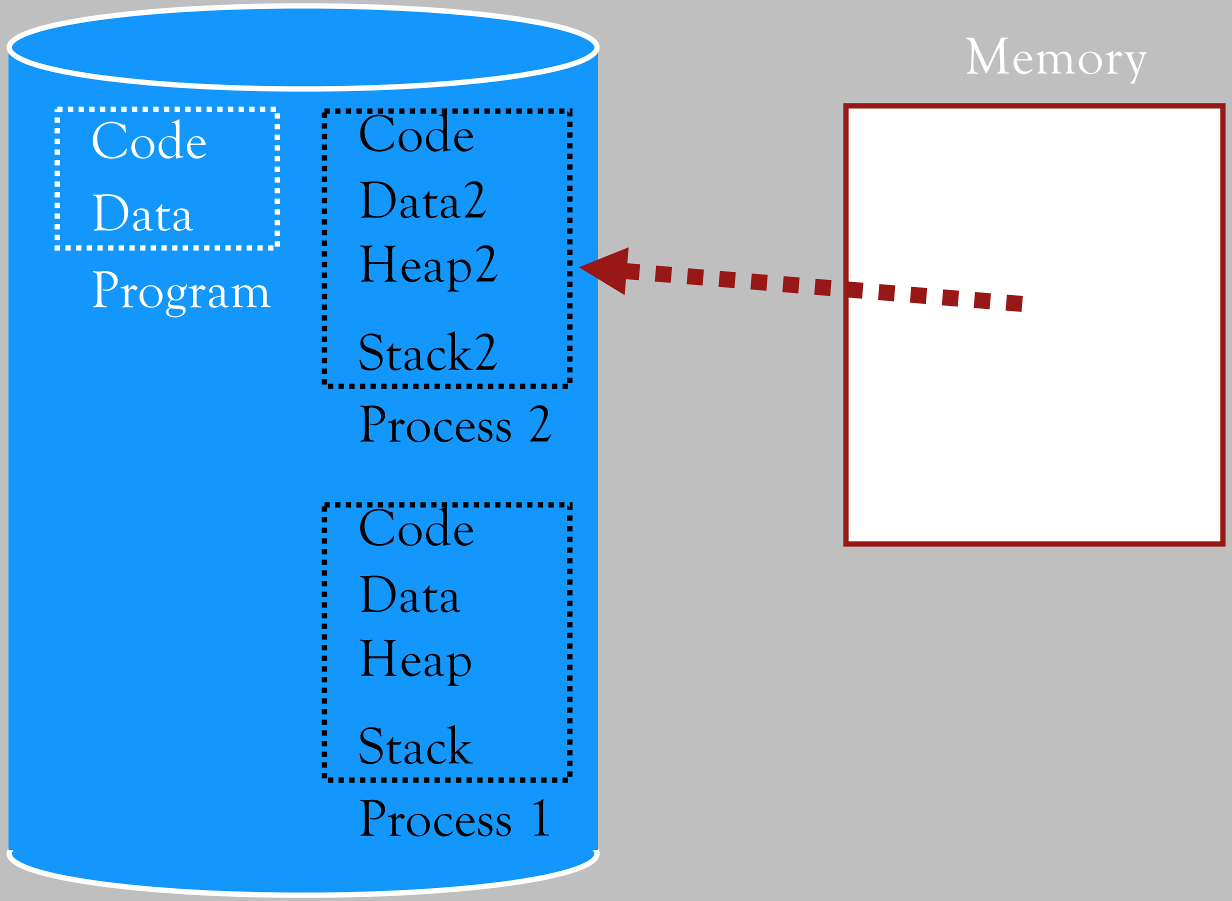

Code
Data
Program
Memory
Code
Data
Heap
Stack
Process 1

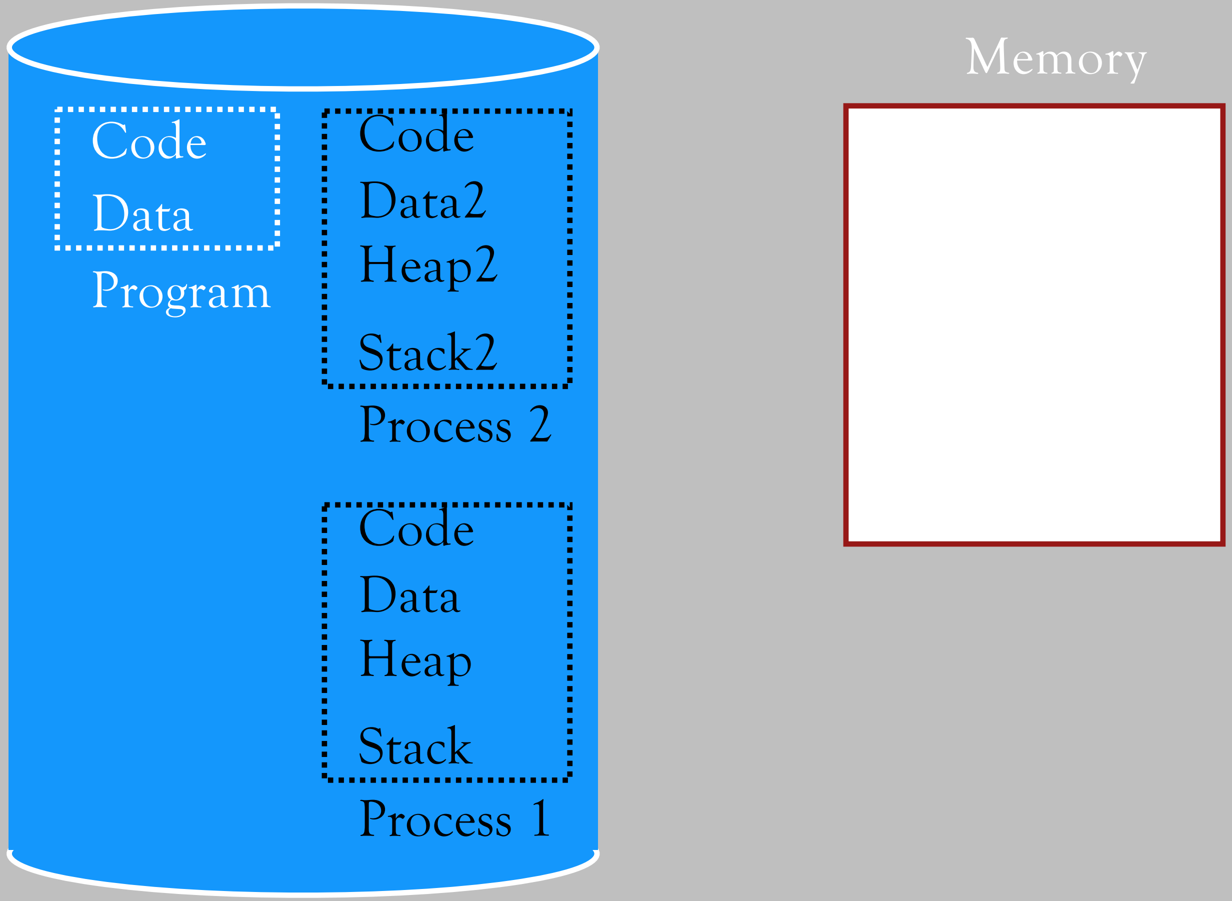

Code
Data
Program
Code
Data
Heap
Stack
Process 1
Memory

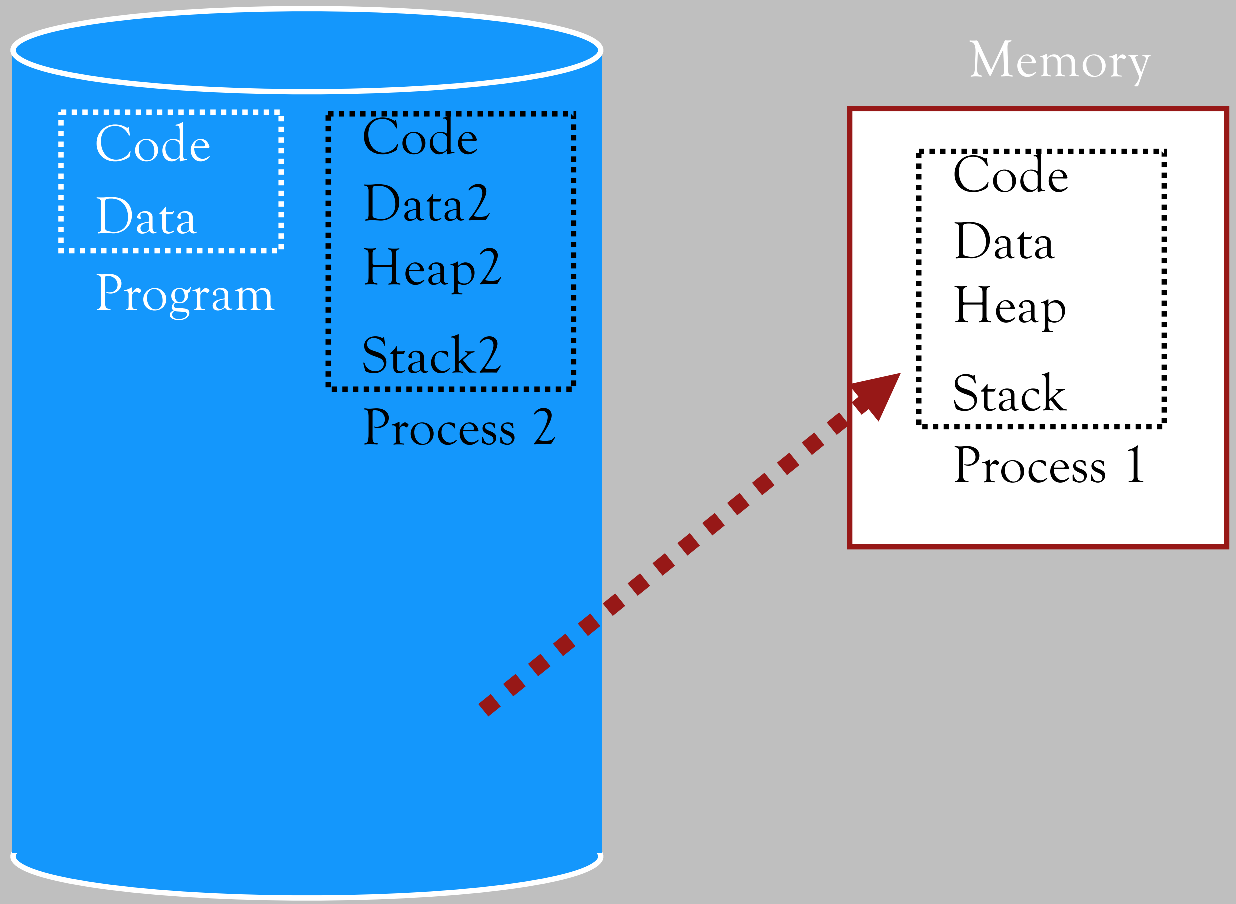

Code
Data
Program
Code
Data
Heap
Stack
Process 1
Memory

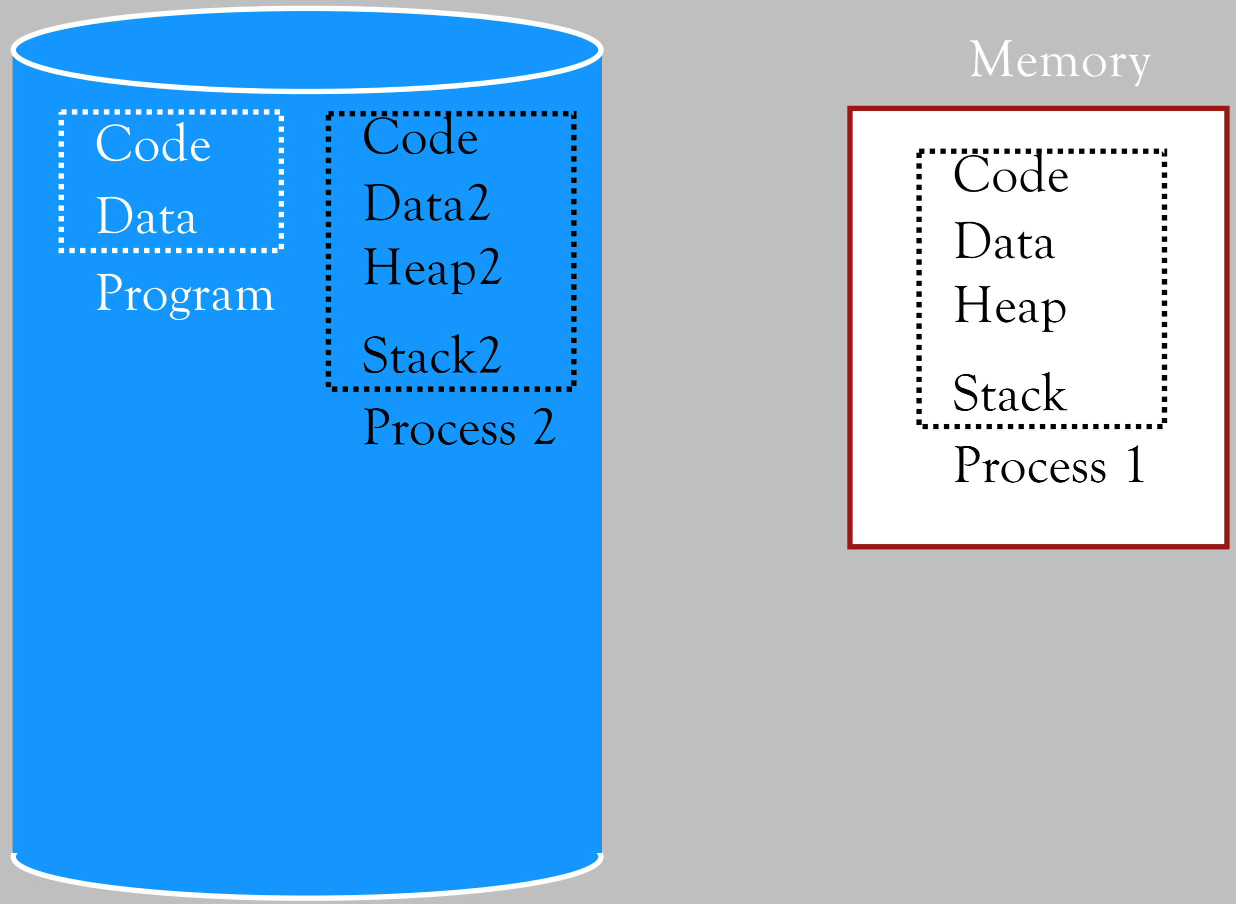

Code
Data
Program
create
Memory
Code
Data2
Heap2
Stack2
Process 2
Code
Data
Heap
Stack
Process 1


Code
Data
Program
Code
Data
Heap
Stack
Process 1
Memory
Code
Data2
Heap2
Stack2
Process 2


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
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

Problem?

Problems with time sharing memory

Problem:
Extremely poor performance, as copying expensive

Better alternative: **space sharing**

- Physical memory is divided across several processes

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

Example: Static relocation

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

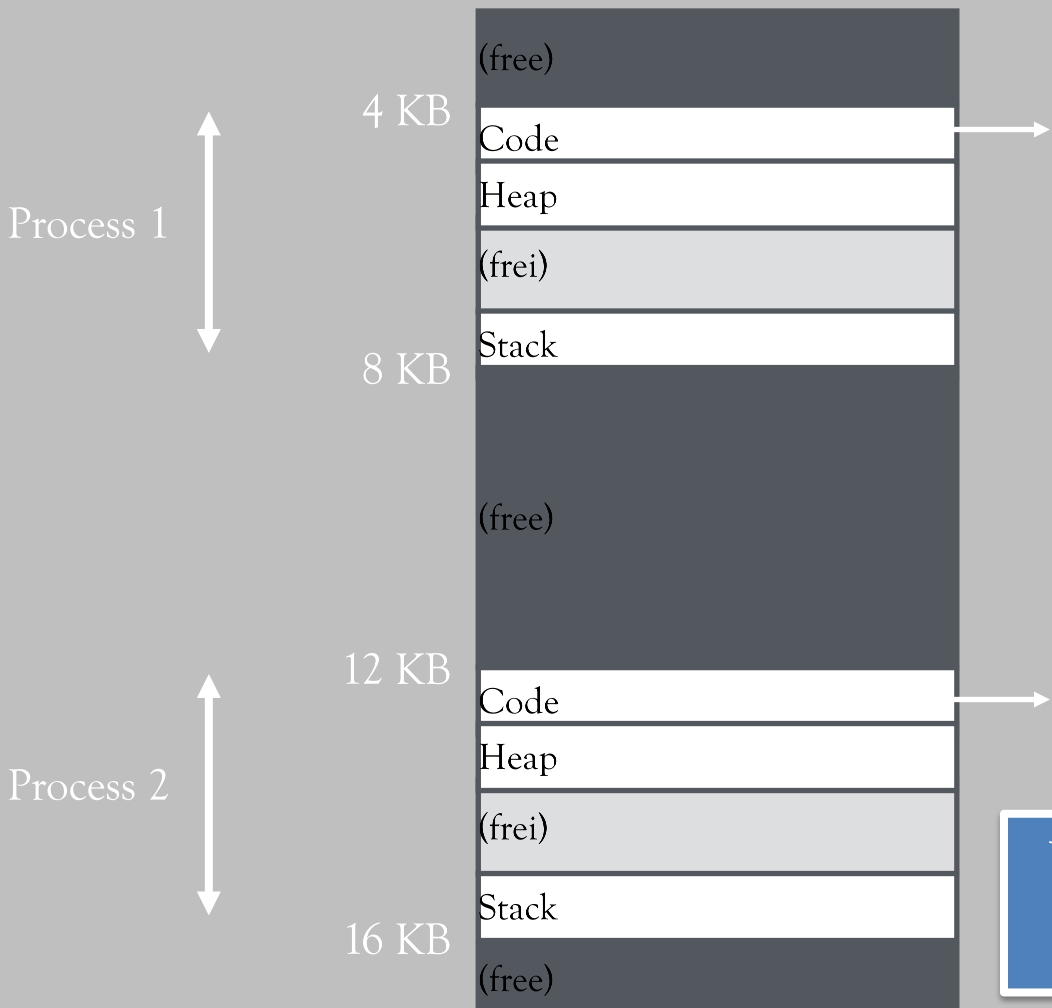
rewrite

- `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)
```

Example: Static relocation




```
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 addresses?

Static relocation: Disadvantages

No protection:

Process can destroy (and spy on) OS
or other processes

Cannot move address space after it has been placed
→ possible fragmentation

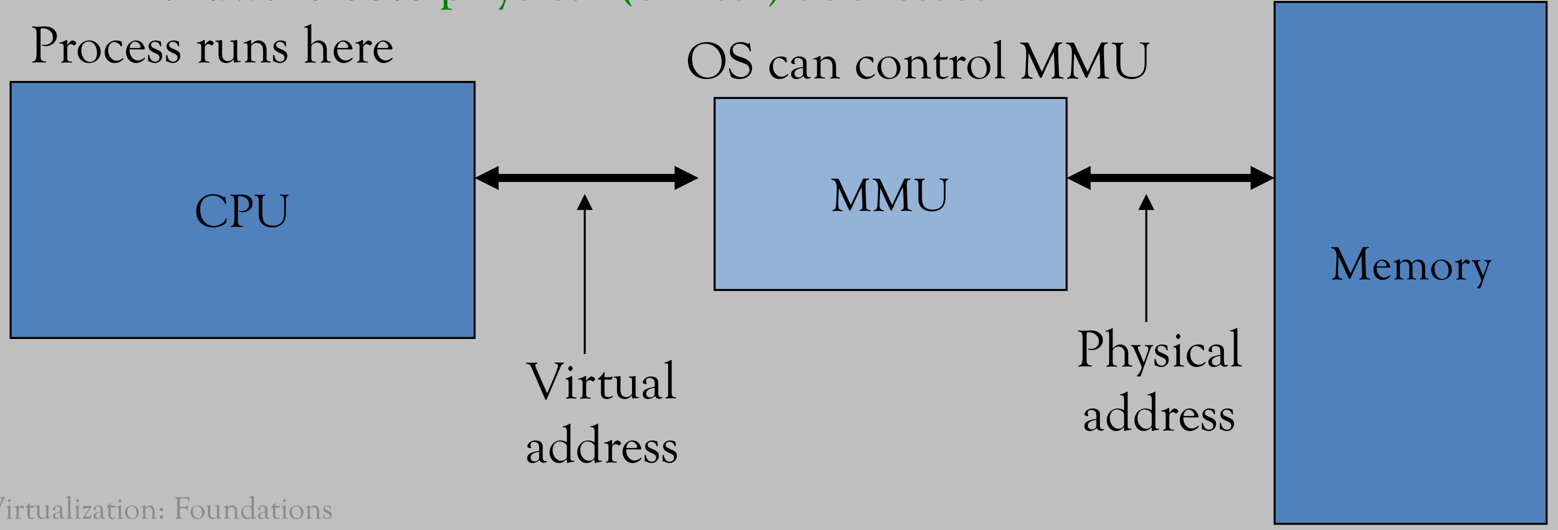
3. Dynamic relocation

Goals:

1. Allow **relocation** of processes even **after** they have been started
2. **Protection** or processes from one another

3. Dynamic relocation

- Requires hardware support:
  Memory Management Unit (MMU)
- MMU dynamically changes process address at every memory access
  - Process generates virtual (or logical) address (in their address space)
  - Hardware uses physical (or real) addresses



Hardware support for dynamic relocation

Two operating modes:

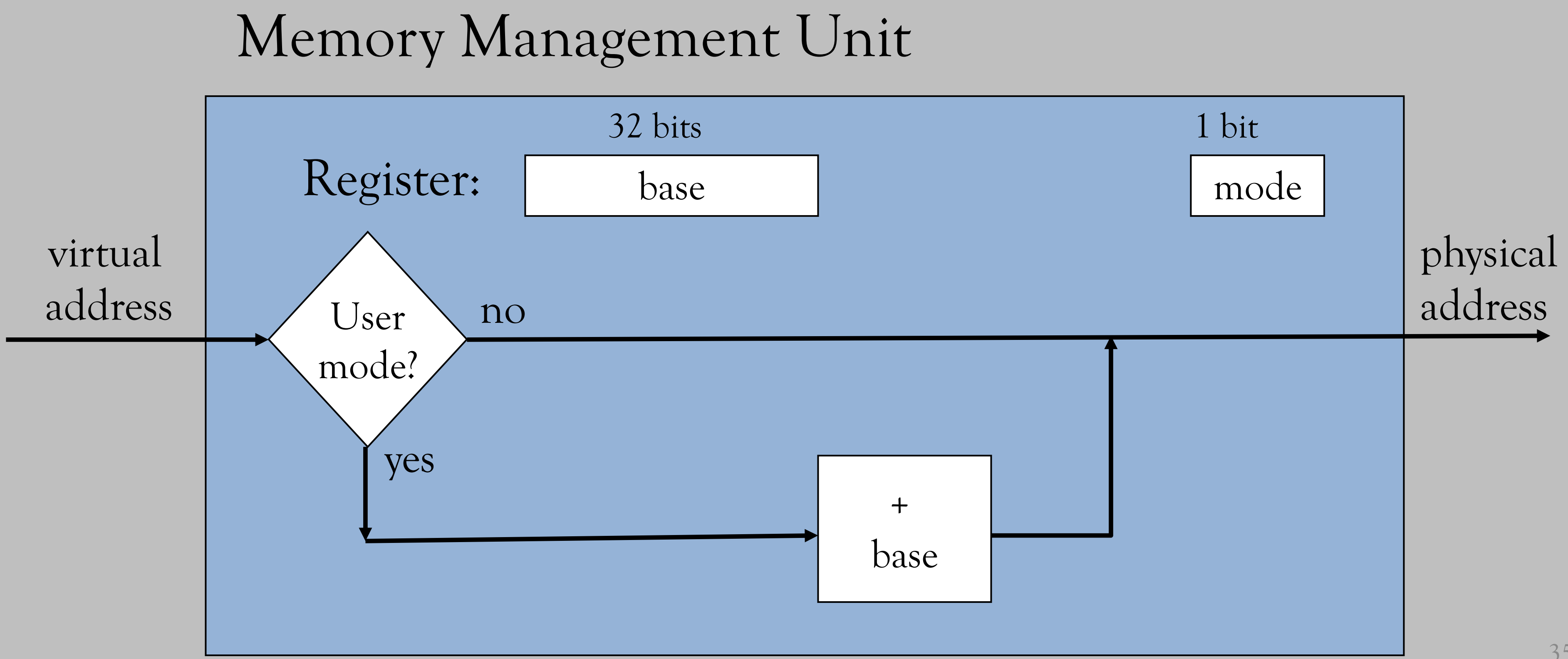
- **Kernel (protected, privileged) mode**: reserved for OS
  - Can manipulate contents of MMU
  - Allows OS to access all of physical memory
- **User mode**: for user processes
  - MMU translates virtual addresses to physical addresses

Minimal MMU contains **base register** for translation

- base: start location for address space

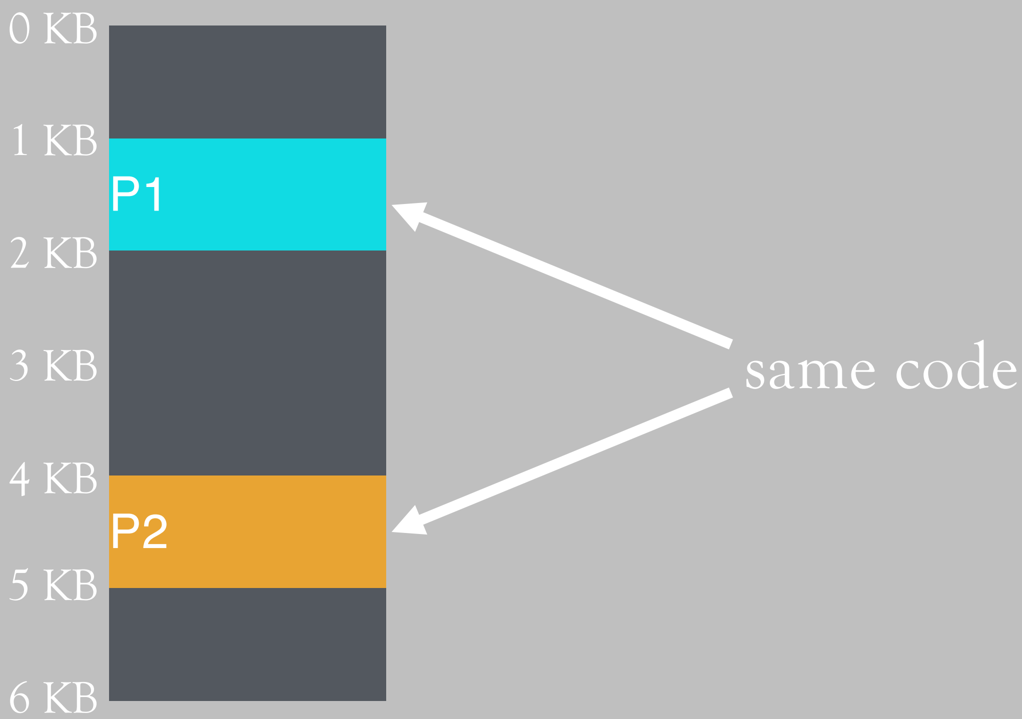
Implementation of dynamic relocation: Base register

MMU sums value of base register onto virtual addresses to obtain physical addresses



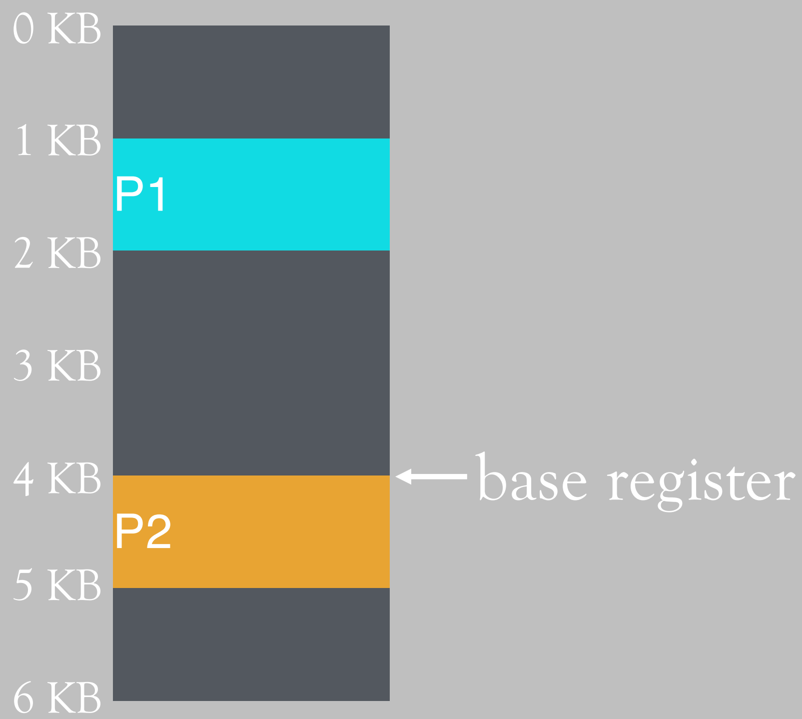
Dynamic relocation with base register

OS writes **correct value** to **base register** upon each context switch



Example: **Dynamic relocation**

P1 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 5096, R1
P1: load 1000, R1	load 2024, R1

Quiz: Who controls the base register?

What entity **does translation** of addresses?

(a) user process, (b) OS, or (c) HW

Which entity **modifies** the base register?

(a) user process, (b) OS, or (c) HW

Quiz: Who controls the base register?

What entity **does translation** of addresses?

(a) user process, (b) OS, or **(c) HW**

Which entity **modifies** the base register?

(a) user process, **(b) OS**, or (c) HW

Virtuell	Physisch
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

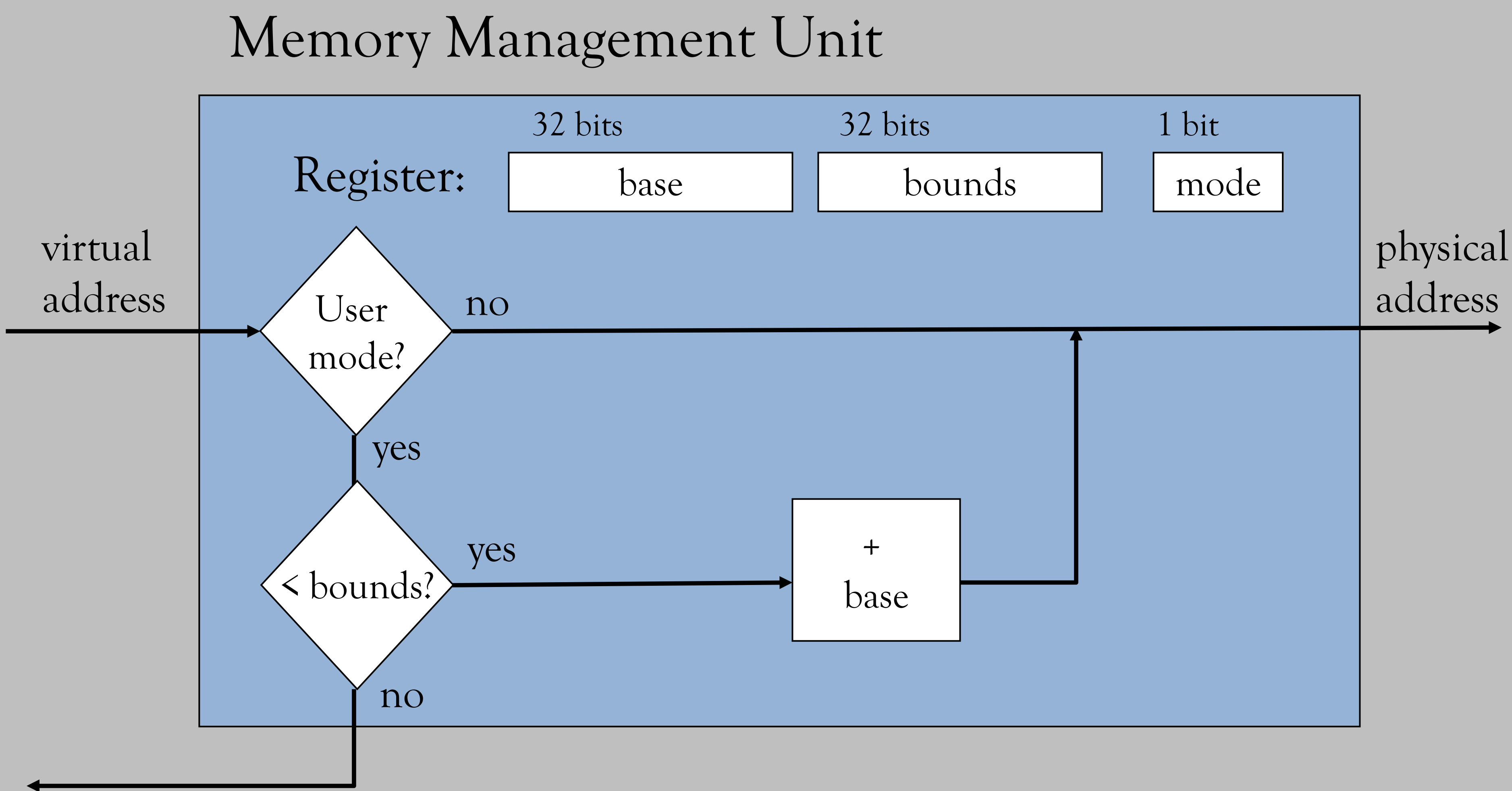
Can P1 modify data of P2?

Dynamic relocation with base + bounds

Idea: Limit address space with a **"bounds register"**

- **Base register**: smallest physical address
- **Bounds register**:
  Size of the process's virtual address space
- OS **kills** process if process loads/stores beyond bounds

Implementation of base + bounds



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 100, R1	load 2024, R1
P1: store 3072, R1	interrupt OS!

Extension of OS for dynamic relocation

Add fields for base and bounds of address space in process control blocks (PCB)

Context switch (in kernel mode):

1. Load base and bounds values of new process from PCB into MMU registers
2. Switch to user mode and jump to new process

Precondition for security:

User processes are unable to modify base and bounds registers
→ Ensured by execution in user mode

Advantages of dynamic relocation

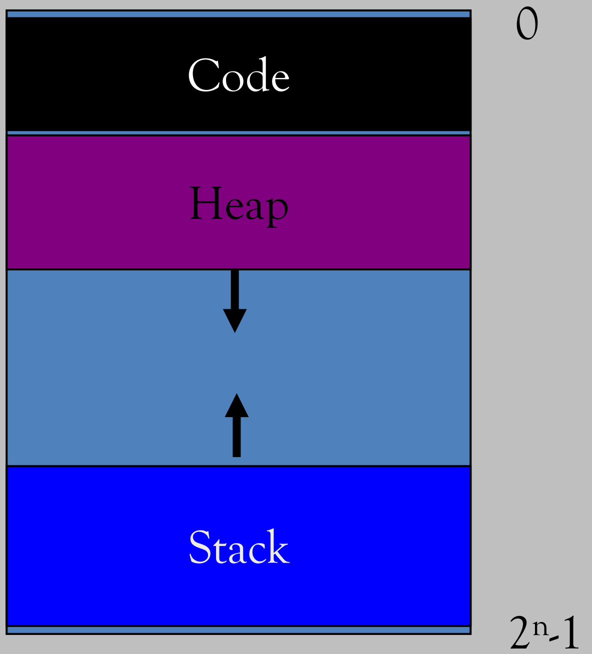
1. Supports dynamic relocation of processes at **runtime**
2. Provides **protection** across address spaces

- Simple and cheap to implement:
  few registers, little logic in MMU
- *Fast*: Add and compare can be performed in parallel

Disadvantages of dynamic relocation

- Each process must be allocated **contiguously** in physical memory
  - *Internal* fragmentation
  - *External* fragmentation

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



4. Segmentation

Divide address space into **logical segments**:

Code, Stack, Heap

Each segment can **independently:**

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

Segmented addressing

How does process designate a particular segment?

- Use part of virtual address:
  - most-significant bits select segment
  - other bits encode offset within segment

Segmentation: Implementation

MMU contains **segment table** (per process):

- Each segment has own base and bounds, protection bits
- *Example*: 14-bit virtual address, 4 segments
  - How many segment bits? 2
  - How many for offset? 12

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

Quiz: Address translations with segmentation

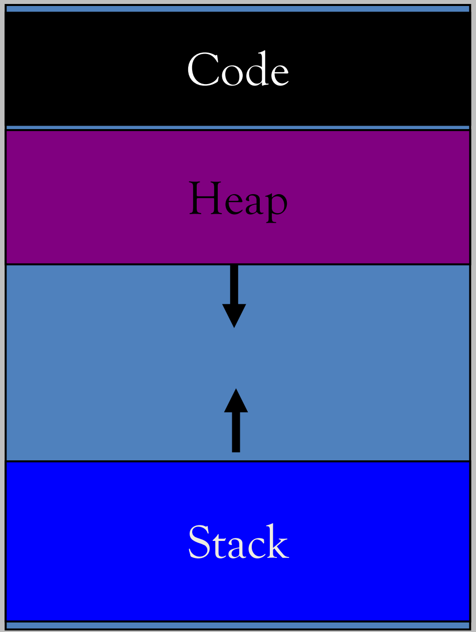
Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x000	0 0

Translate logical addresses (in hex) to physical addresses?

```
0x0240:   0x2240
0x1108:   0x0108
0x265c:   0x365c
0x3002:
```

Advantages of segmentation

- Enables “sparse” allocation of address space:
  - Stack and heap grow independently of each other
    → no **internal** fragmentation
  - *Heap*: If no data on free list, dynamic memory allocator (in library) requests more from OS (e.g., UNIX malloc calls sbrk())
  - *Stack*: OS recognizes references outside legal segment, extends stack implicitly
- Different protection for different segments:
  - E.g. 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** in memory

→ **External** fragmentation of the physical memory

→ Paging as solution next lecture

Conclusion

- HW + OS work together to virtualize memory
  - Memory Management Unit supports fast address translation in HW
  - OS only involved upon context switches or errors