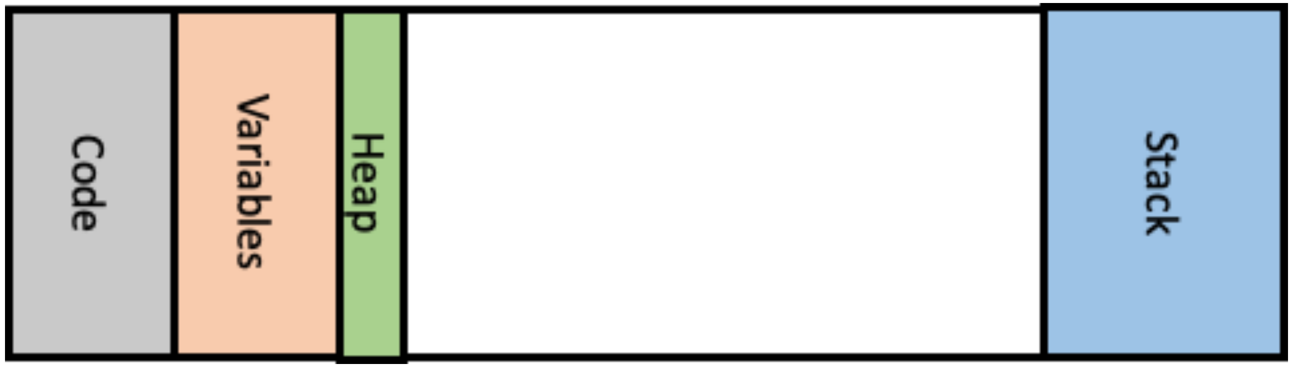
CS 1217

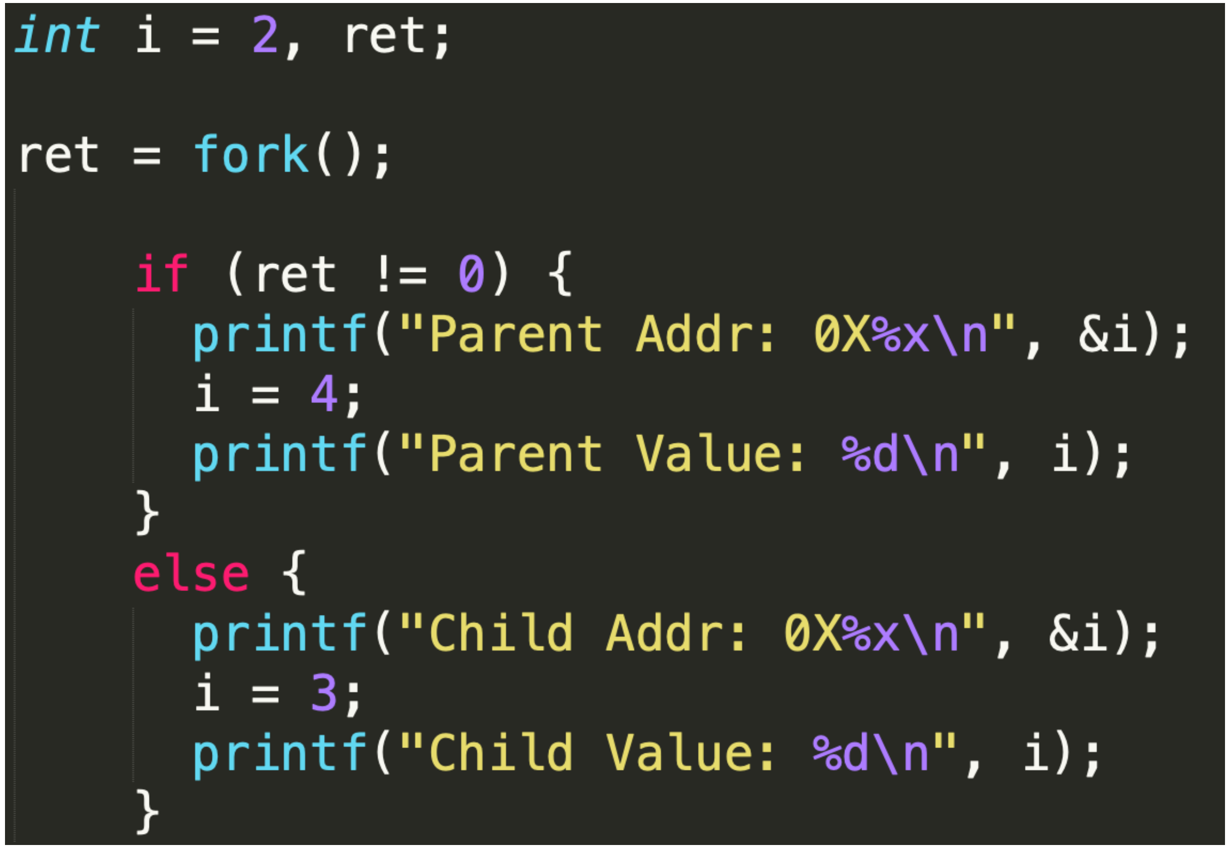
Base and Bounds, Segmentation, Intro to Paging

Recap

- Virtual Addresses present one more level of indirection
- Virtual Address, in addition to pointing to physical memory could point to “other things”
  - Disk, Block, Offset
  - IP Address, Physical Address
  - Device, Port
- Can also associate read/write permissions with blocks of virtual addresses



What is happening here?



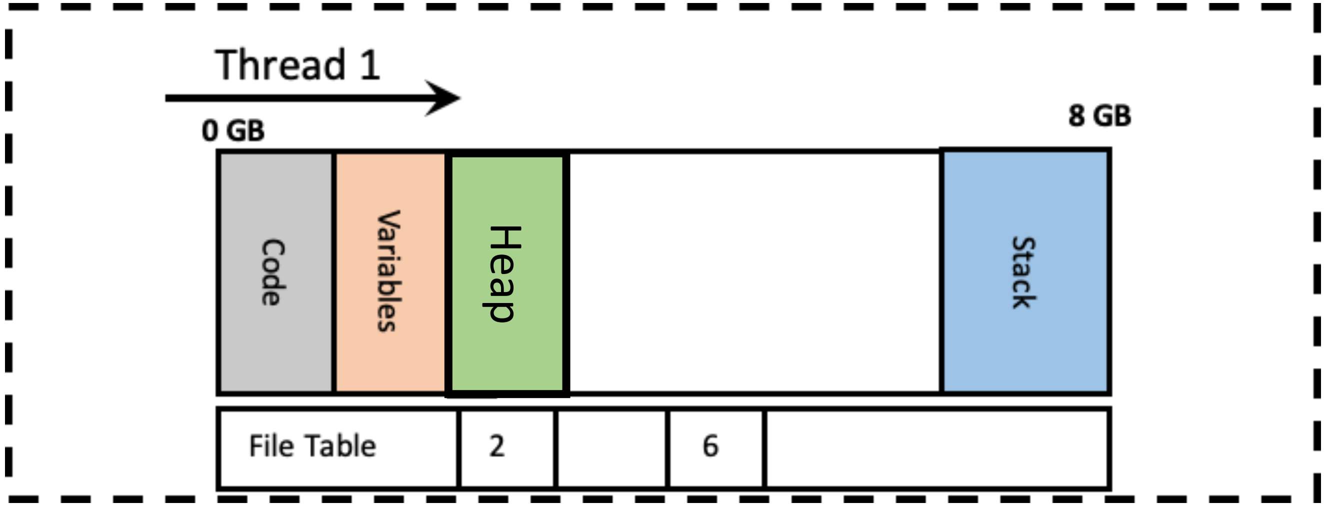
```
int i = 2, ret;

ret = fork();

    if (ret != 0) {
      printf("Parent Addr: 0X%x\n", &i);
      i = 4;
      printf("Parent Value: %d\n", i);
    }
    else {
      printf("Child Addr: 0X%x\n", &i);
      i = 3;
      printf("Child Value: %d\n", i);
    }
```

Creating Virtual Addresses: exec()

- How does exec() know what the address space of the new process is going to look like?
  - The ELF file has a blueprint
- exec() creates and initializes **virtual addresses** that (mainly) point to **memory**
  - **code**, usually marked *read-only* and executable
  - **data**, marked *read-write*, but not executable
  - **heap**, an area used for *dynamic allocations*, marked read-write
  - **stack** space for the *first* thread

Creating Virtual Addresses: sbrk()

- Programmers use malloc() for dynamic memory allocation in C
- malloc() in turn uses sbrk(), a system call
- sbrk() asks the kernel to move the **break point**, or the point at which the process heap ends



Creating Virtual Addresses: mmap()

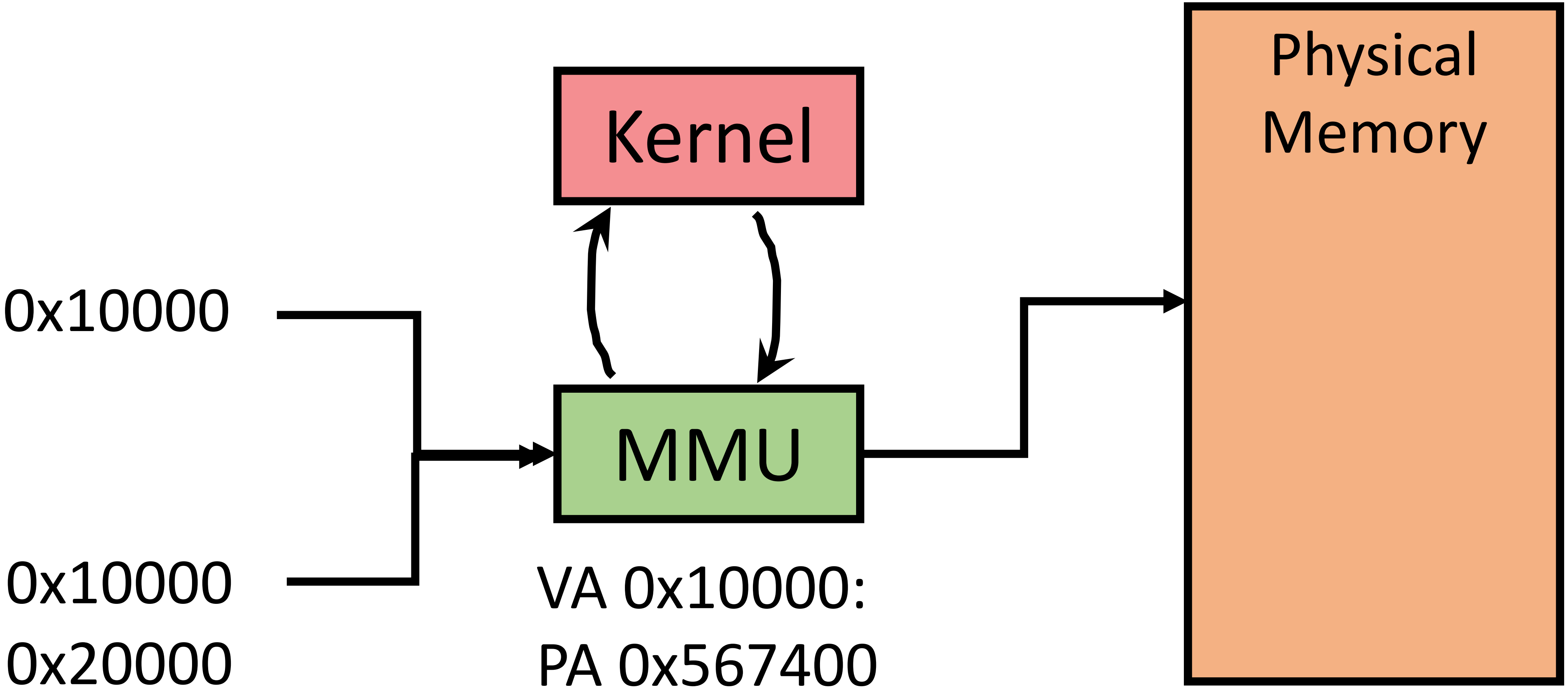
- mmap() is a system call that creates virtual addresses that map to a portion of a **file**

- Advantages?

Policy vs. Mechanism

- Virtual Address Spaces
    - Policy
- Virtual Address Translation
    - Mechanism

Address Translation



VA 0x10000:
PA 0x567400

Address Translation

- What are the possible drawback of virtual address **translation**?
- Terrible for performance!
  - Every memory access has to now be **translated**
  - The kernel now has to be **involved for every** memory access
- Hardware involvement in translation can make it faster : enter the Memory Management Unit (MMU)
- The kernel now can decide mapping and translation **policies**, and the MMU can provide the translation **mechanism**

Efficient Translation

- Constraints:
    - We don’t want to involve the kernel in every address translation
    - We don’t want to provide the process the physical address

- Solution: Involve the kernel to translate **chunks of addresses** at a time, invoke again only if you want translations outside of the chunk

Example

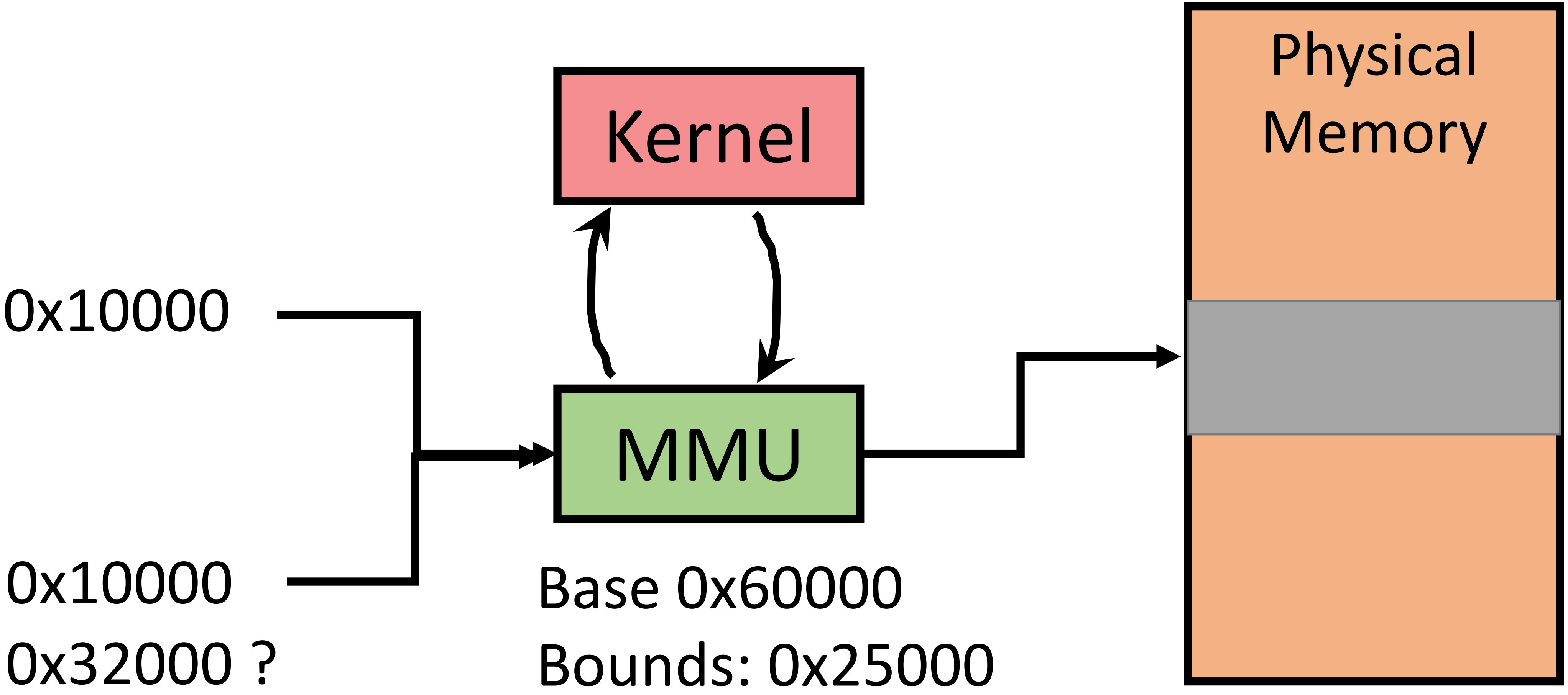
- Process: Store to address 0x10000!
- MMU: I don’t know what 0x10000 maps to
  - Exception!
- MMU -> kernel : Please provide translation
- Kernel: 0x10000 maps to 0x567400
- Process (completes store)
- Process: makes forward progress

Base and Bounds

- One of the simplest translation mechanisms possible

- Assign each process a **base** physical address and **bound**
- **Check:** Virtual Address is OK if Virtual Address < bound.
- **Translate:** Physical Address = Virtual Address + Base

Base and Bounds Translation



Base 0x60000

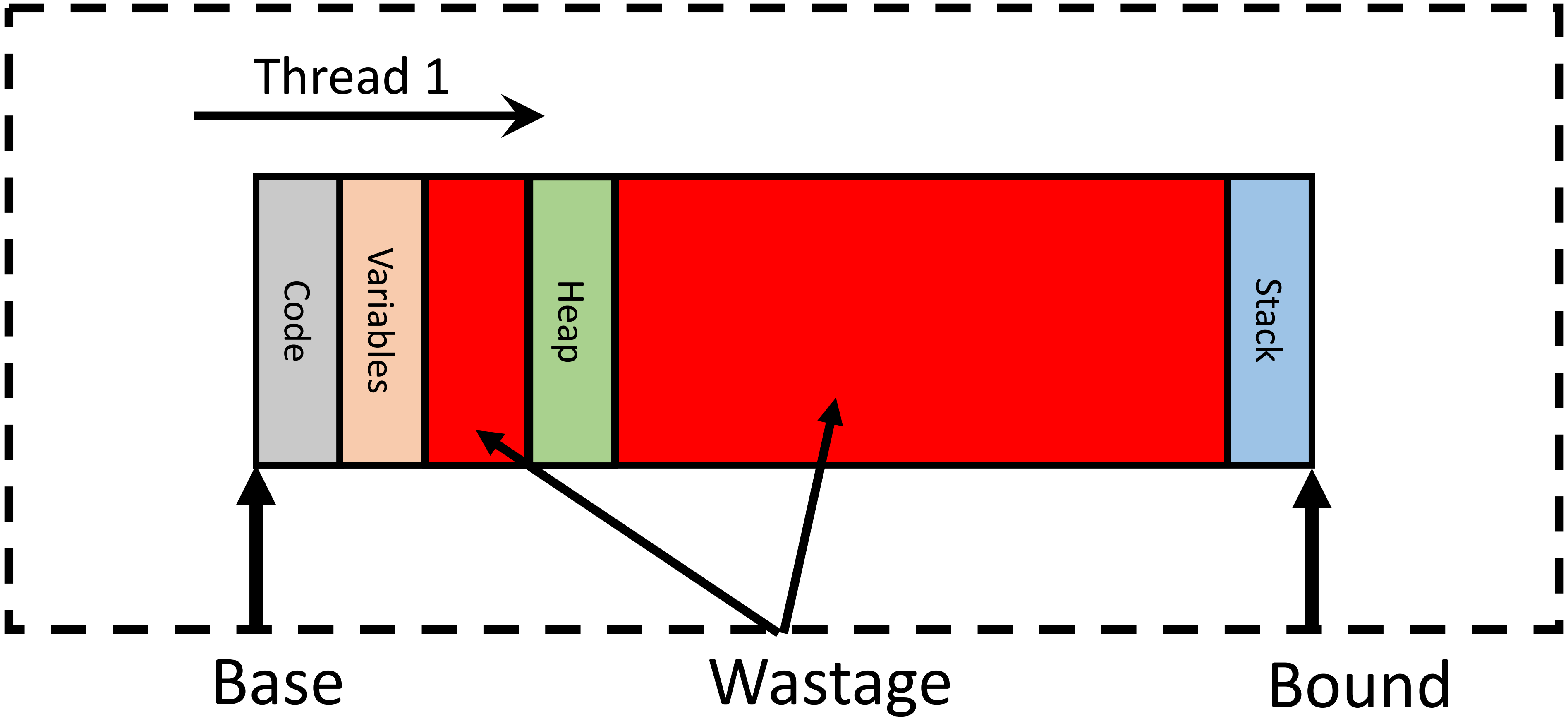
Bounds: 0x25000

Base and Bounds : Pros and Cons

- Pros?

Base and Bounds : Pros and Cons

- Allocations are contiguous : good or bad?



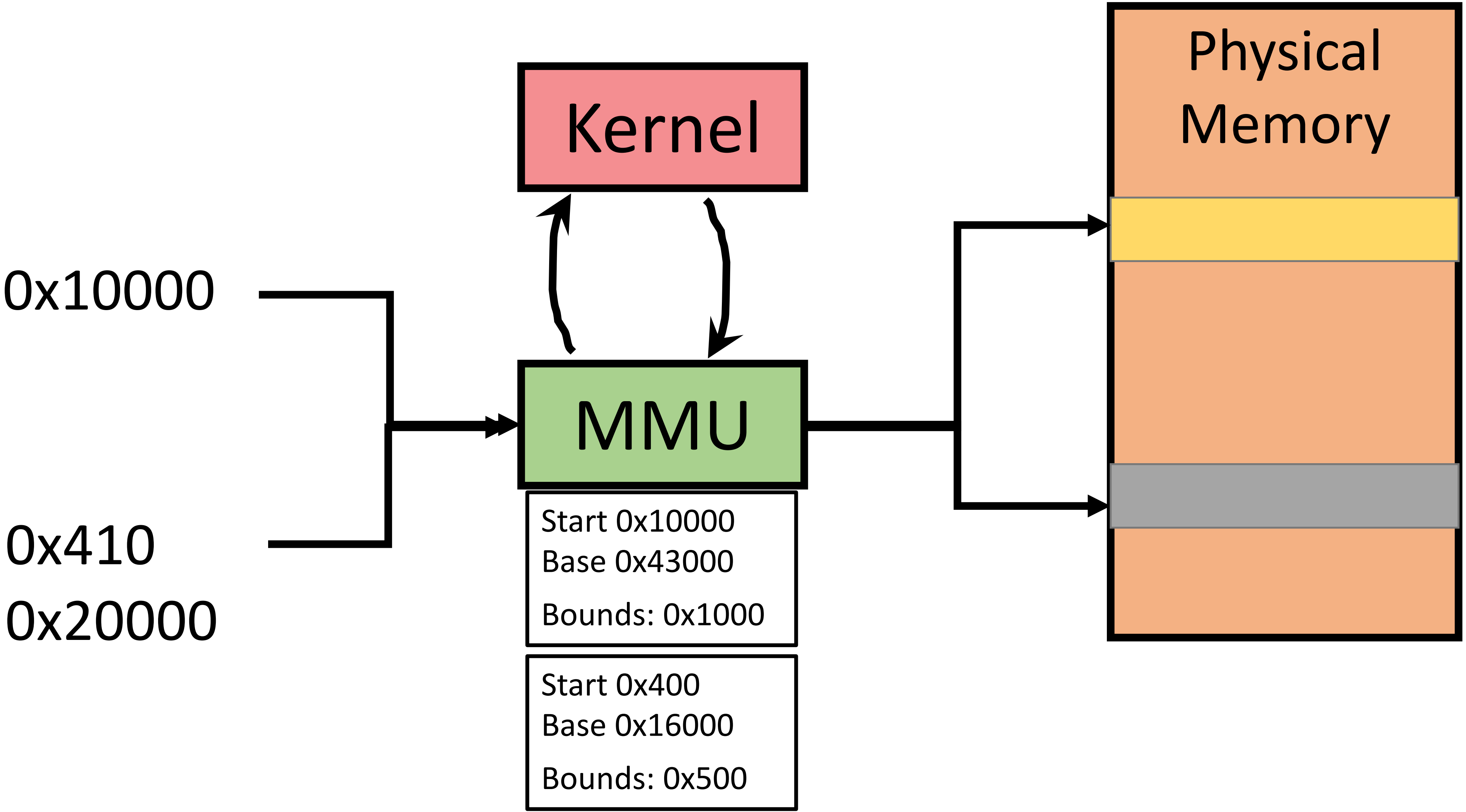
Extension : Segmentation

- **One** base and bounds isn’t a good fit for the address space abstraction : leads to fragmentation
- **Solution : Multiple** bases and bounds per process, each called a **segment**.
- What can be a segment?
  - each logical region of the address space—code, data, heap, stack
  - Each can be a separate **size**.
  - Each can have separate permissions

Segmentation Details

- Each **segment** has a **start** virtual address, **base** physical address, and **bound**
- **Check:** Virtual Address is OK if it inside some segment, or for some segment:
  - Segment Start < V.A. < Segment Start + Segment Bound.
- **Translate:** For the segment that contains this virtual address: Physical Address = (V.A. - Segment Start) + Segment Base

Segment Based Translation



0x10000

0x410

0x20000

Kernel

MMU

Start 0x10000
Base 0x43000
Bounds: 0x1000

Start 0x400
Base 0x16000
Bounds: 0x500

Physical Memory