Memory

CS 241

February 1, 2012

Slides adapted in part from material by
Matt Welsh, Harvard U.

Announcements

- MP2 released
- Brighten's office hours this week
  - Wednesday 3-4
  - Thursday 3-4
- Talk today: Nick Feamster, Georgia Tech

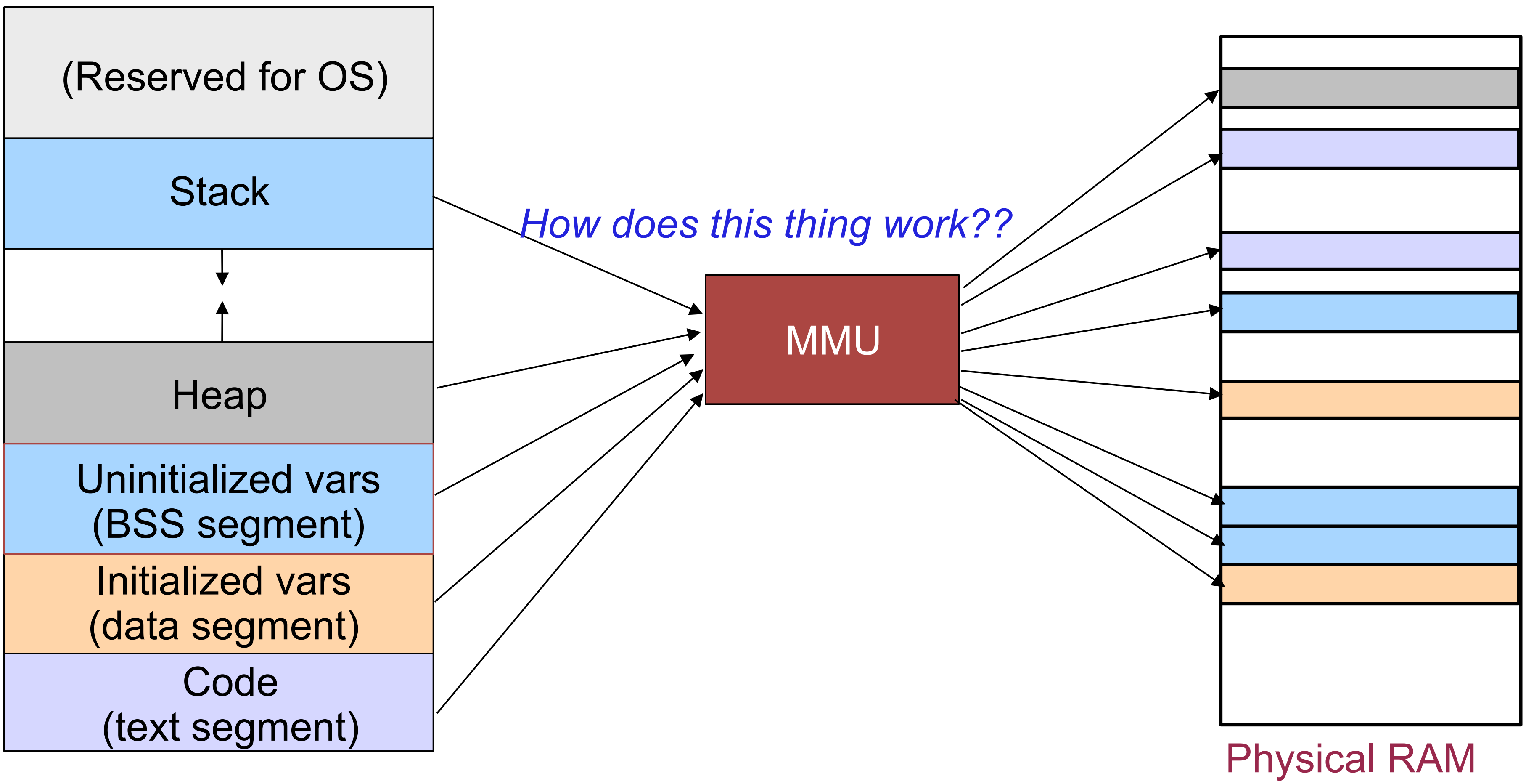
"The Battle for Control of Online Communications"

4:00 p.m.
2405 Siebel Center

Recap: Virtual Addresses

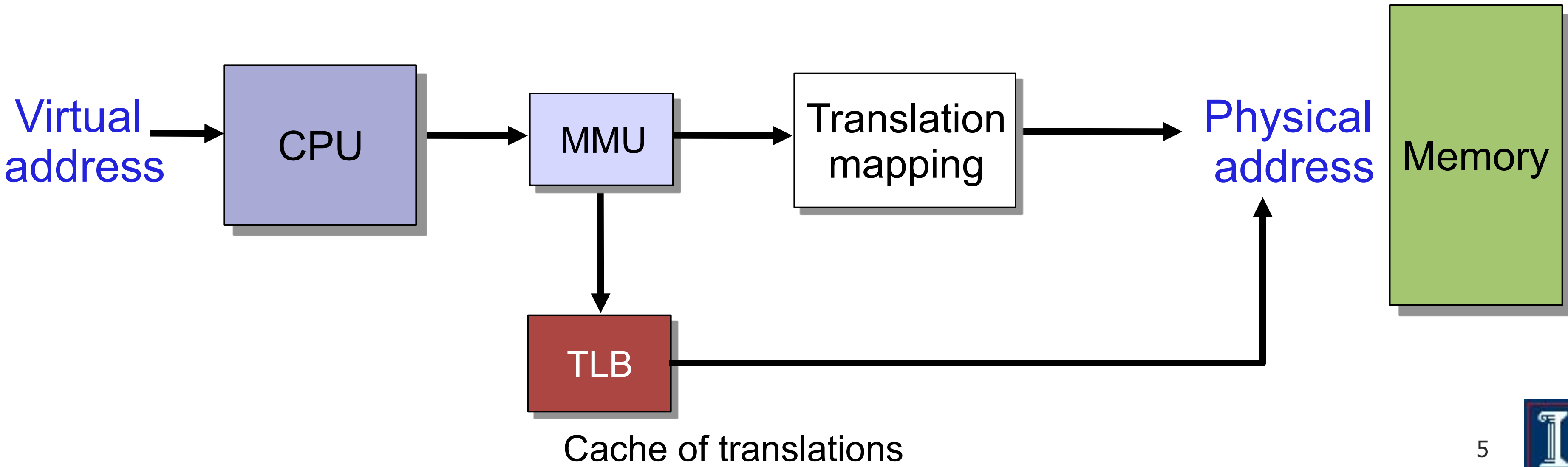
- A **virtual address** is a memory address that a process uses to access its own memory
  - Virtual address ≠ actual physical RAM address
  - When a process accesses a virtual address, the MMU hardware **translates** the virtual address into a physical address
  - The OS determines the mapping from virtual address to physical address
- Benefit: Isolation
  - Virtual addresses in one process refer to ***different*** physical memory than virtual addresses in another
  - Exception: shared memory regions between processes (discussed later)
- Benefit: Illusion of larger memory space
  - Can store unused parts of virtual memory on disk temporarily
- Benefit: Relocation
  - A program does not need to know which physical addresses it will use when it's run
  - Can even change physical location while program is running

Mapping virtual to physical addresses



MMU and TLB

- Memory Management Unit (MMU)
  - Hardware that translates a virtual address to a physical address
  - Each memory reference is passed through the MMU
  - Translate a virtual address to a physical address
    - Lots of ways of doing this!
- Translation Lookaside Buffer (TLB)
  - Cache for MMU virtual-to-physical address translations
  - Just an optimization – but an important one!



Translating virtual to physical

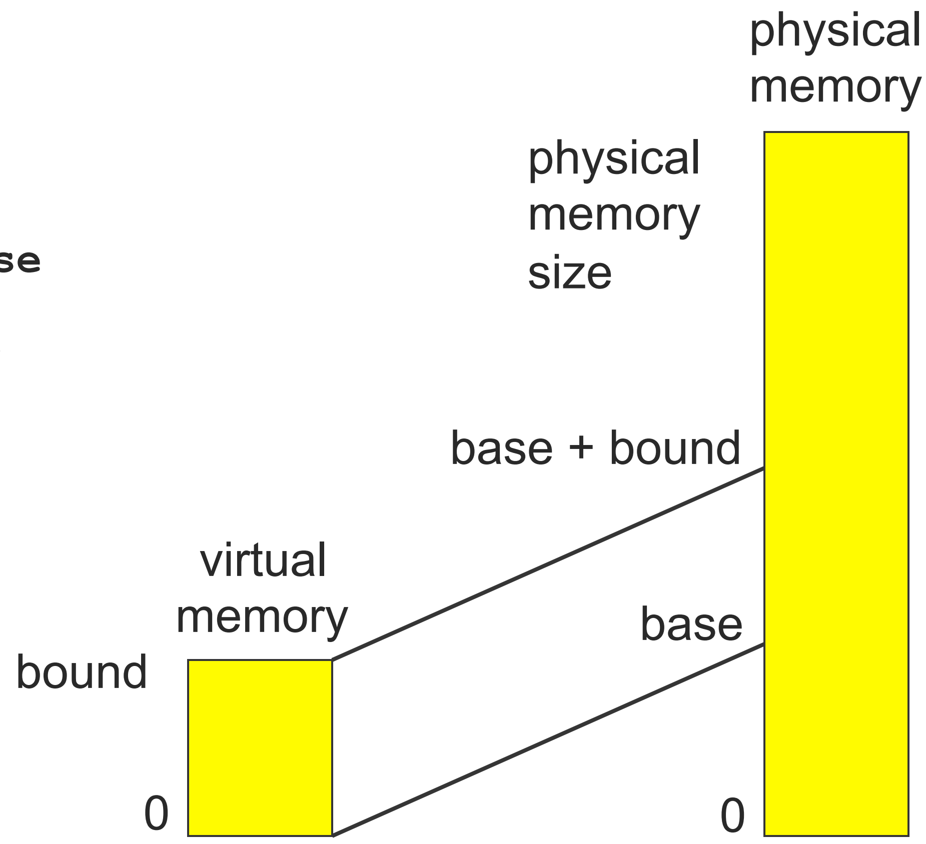
- Can do it almost any way we like
- But, some ways are better than others…

- Strawman solution from last time:
  base and bound

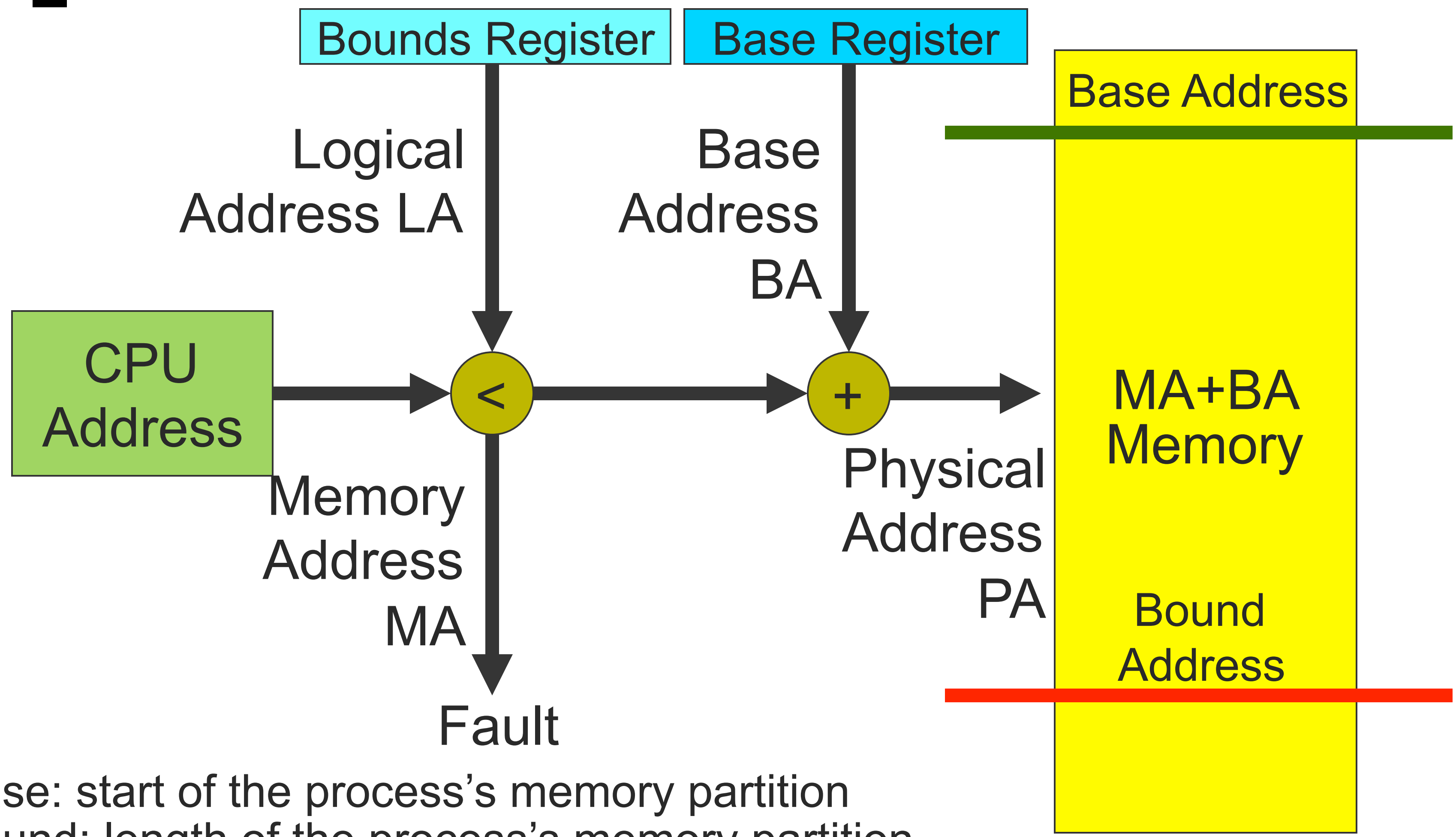
Base and bound

```
if (virt addr > bound)
    trap to kernel
else
    phys addr = virt addr + base
```

- Process has the illusion of running on its own dedicated machine with memory [0,bound)
- Provides protection from other processes also currently in memory



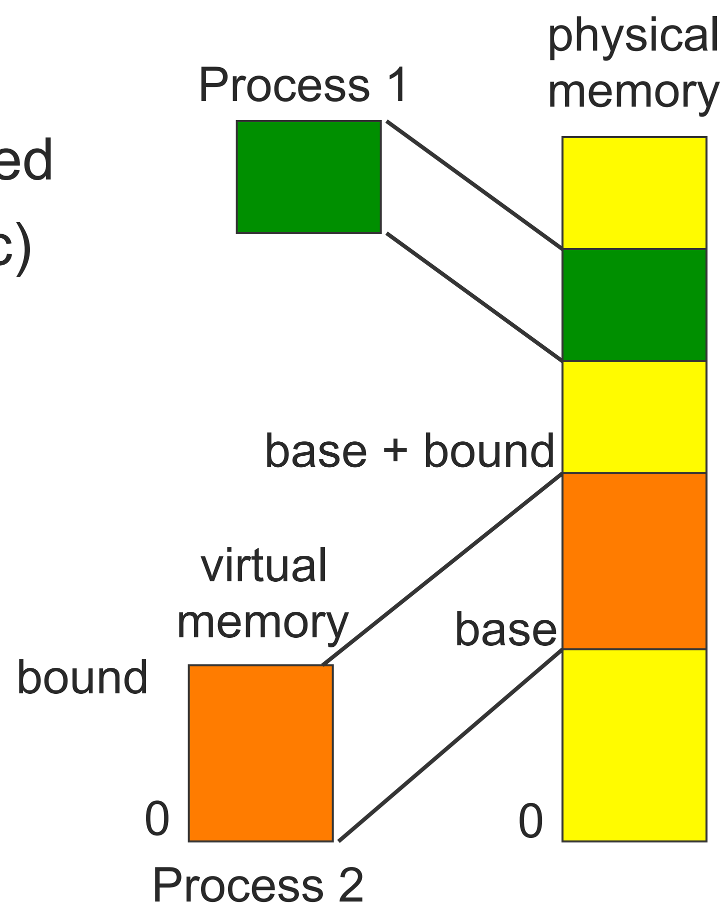
Base and bound



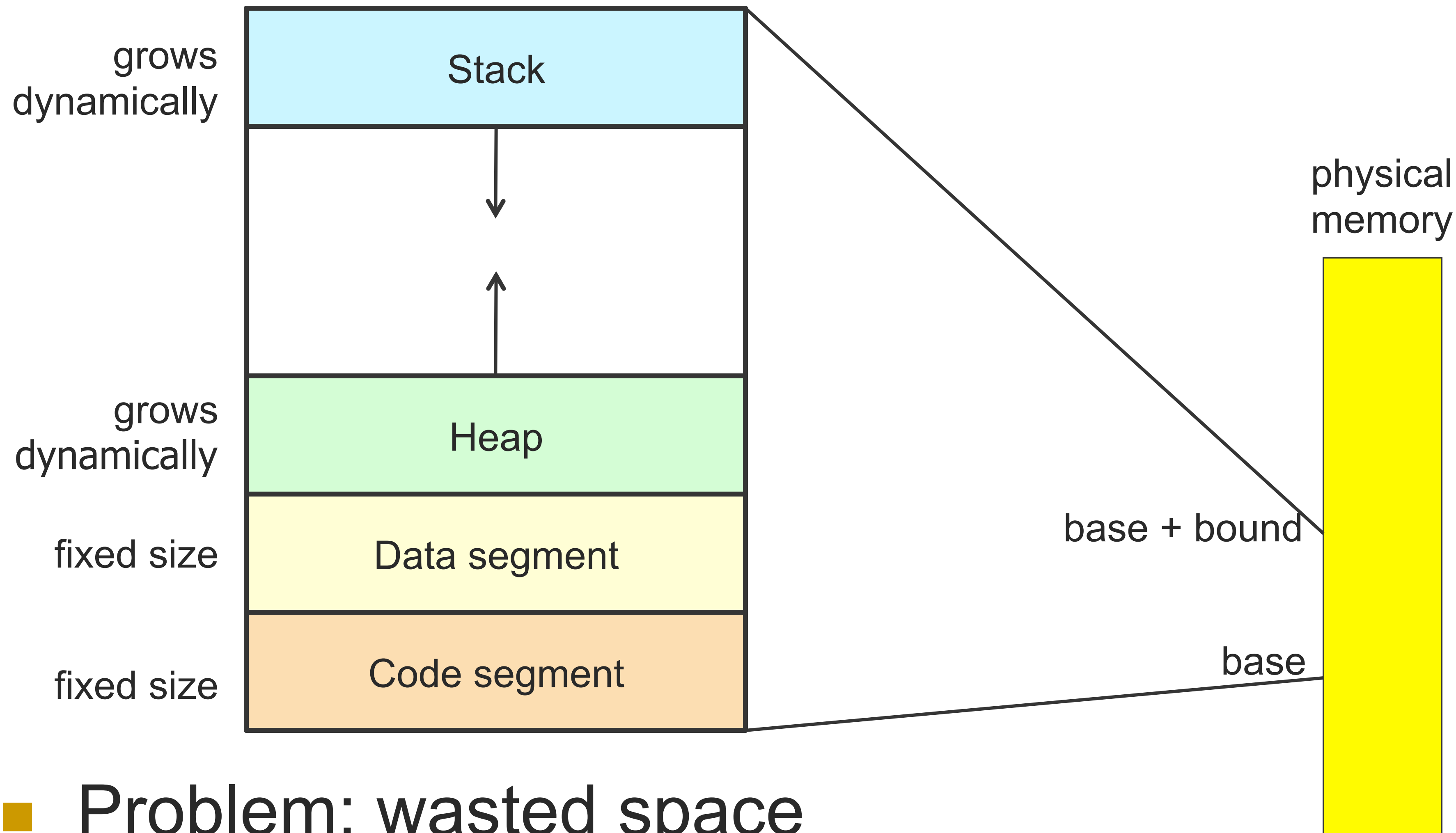
Base: start of the process's memory partition
Bound: length of the process's memory partition

Base and bounds

- Problem: Process needs more memory over time
  - Stack grows as functions are called
  - Heap grows upon request (malloc)
  - Processes start and end
- How does the kernel handle the address space growing?
  - **You are the OS designer**
  - **Design strategy for allowing processes to grow**



But wait, didn't we solve this?



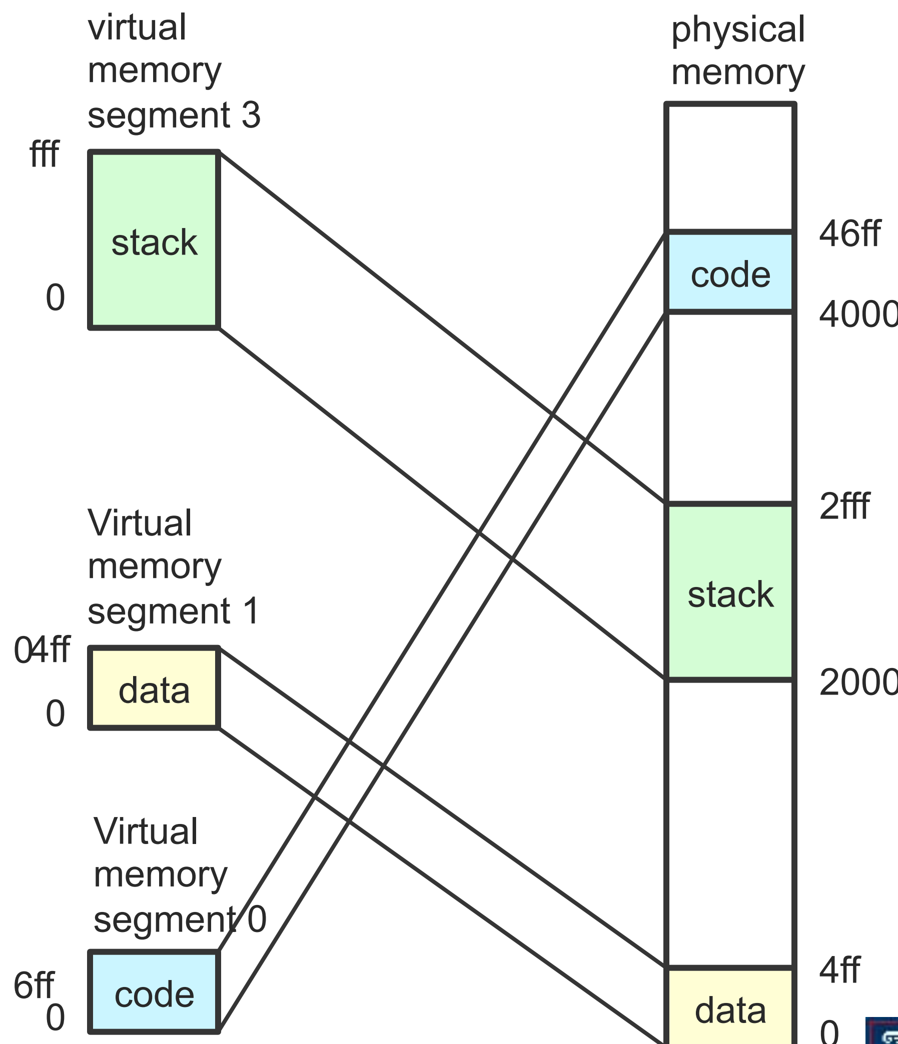
- Problem: wasted space
  - And must have virtual mem ≤ phys mem

Another attempt: segmentation

- Segment
  - Region of contiguous memory
- Segmentation
  - Generalized base and bounds with support for multiple segments at once

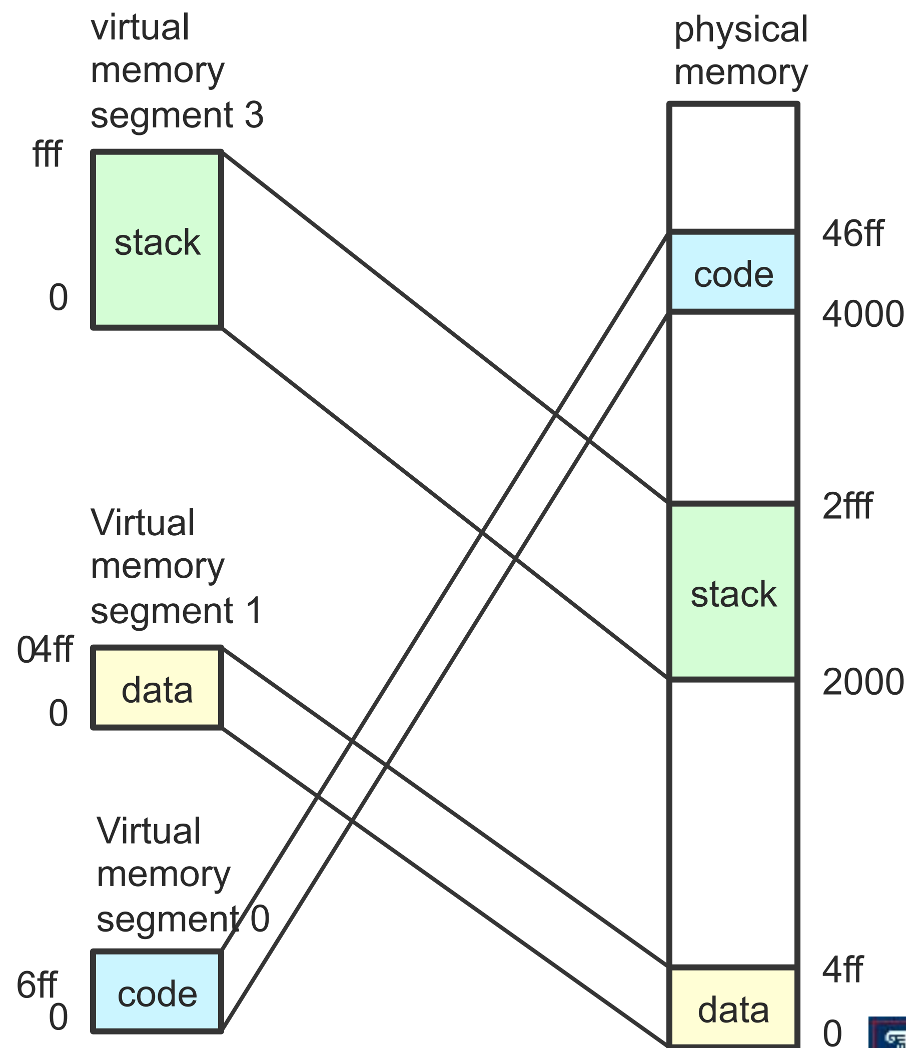
Segmentation

Seg #	Base	Bound	Description
0	4000	700	Code segment
1	0	500	Data segment
2	Unused		
3	2000	1000	Stack segment



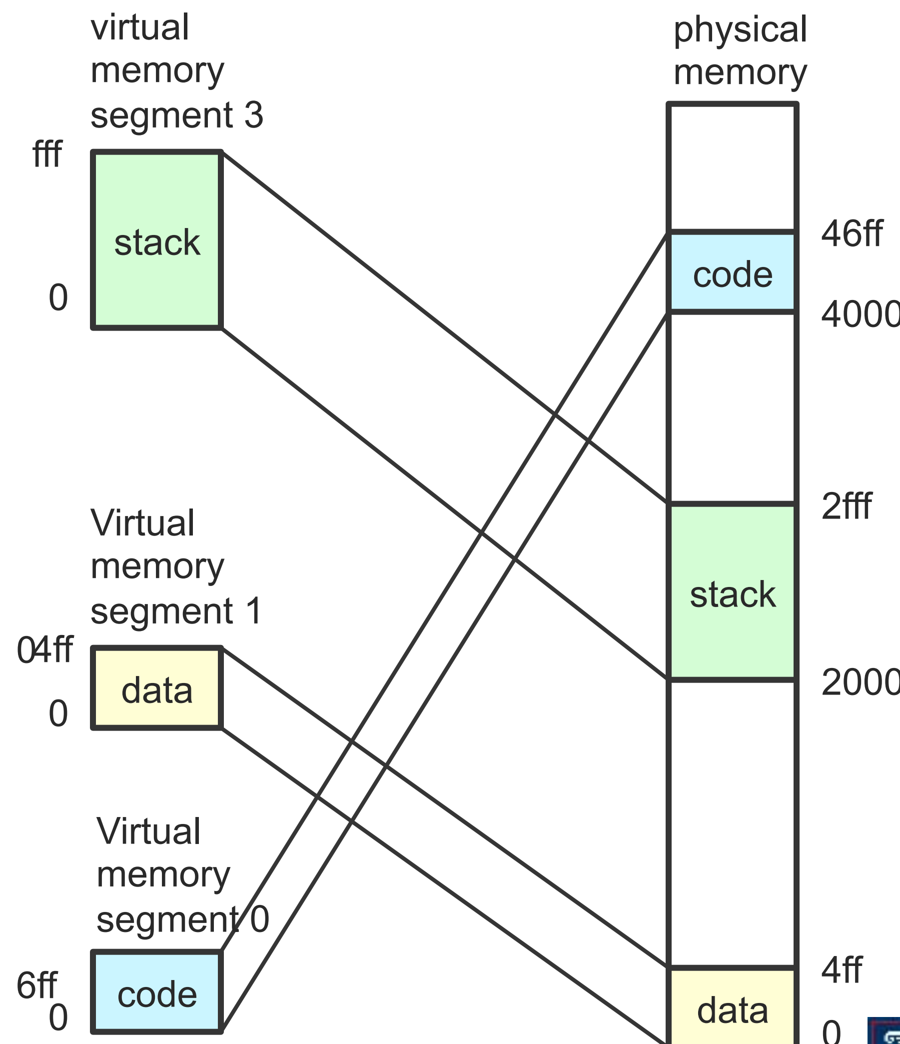
Segmentation

- Segments are specified many different ways
- Advantages over base and bounds?
- Protection
  - Different segments can have different protections
- Flexibility
  - Can separately grow both a stack and heap
  - Enables sharing of code and other segments if needed



Segmentation

- Segments are specified many different ways
- What are the advantages over base and bounds?
- What must be changed on context switch?
  - Contents of your segmentation table
  - A pointer to the table, expose caching semantics to the software (what x86 does)



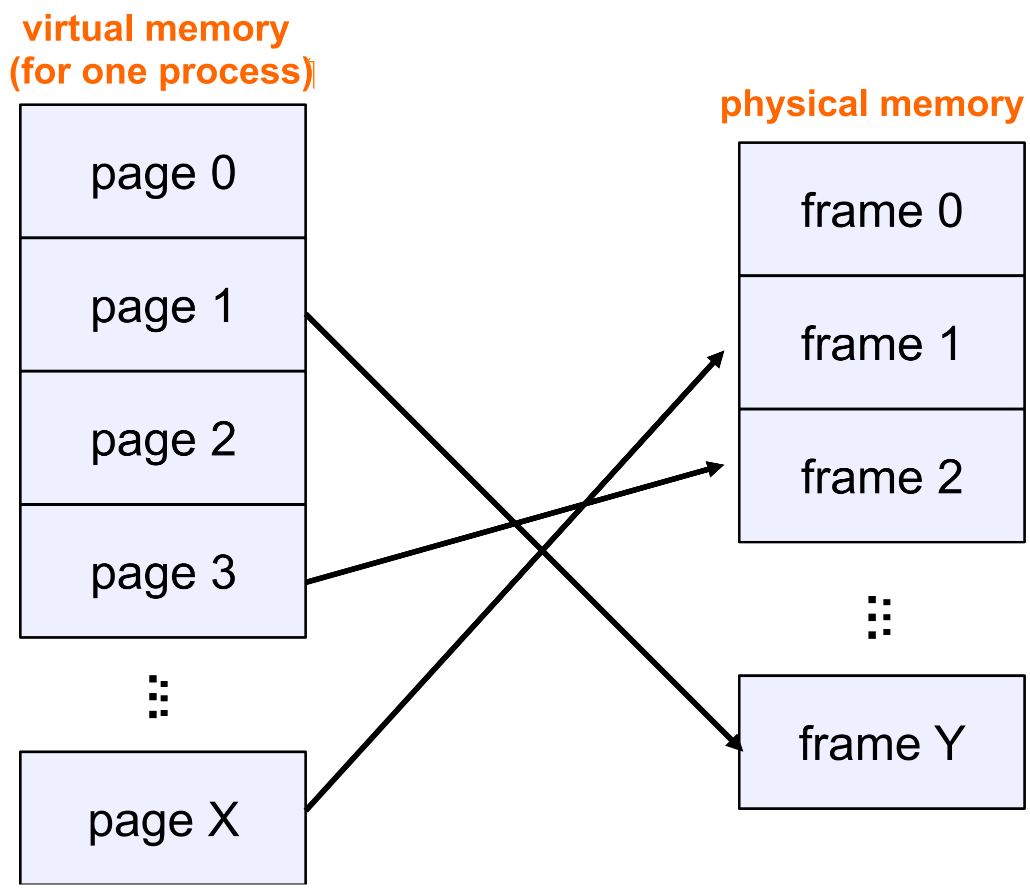
Recap: mapping virtual memory

- **Base & bounds**
  - Problem: growth is inflexible
  - Problem: external fragmentation
    - As jobs run and complete, holes left in physical memory
- **Segments**
  - Resize pieces based on process needs
  - Problem: external fragmentation
  - Note: x86 used to support segmentation, now effectively deprecated with x86-64
- **Modern approach: Paging**

Paging

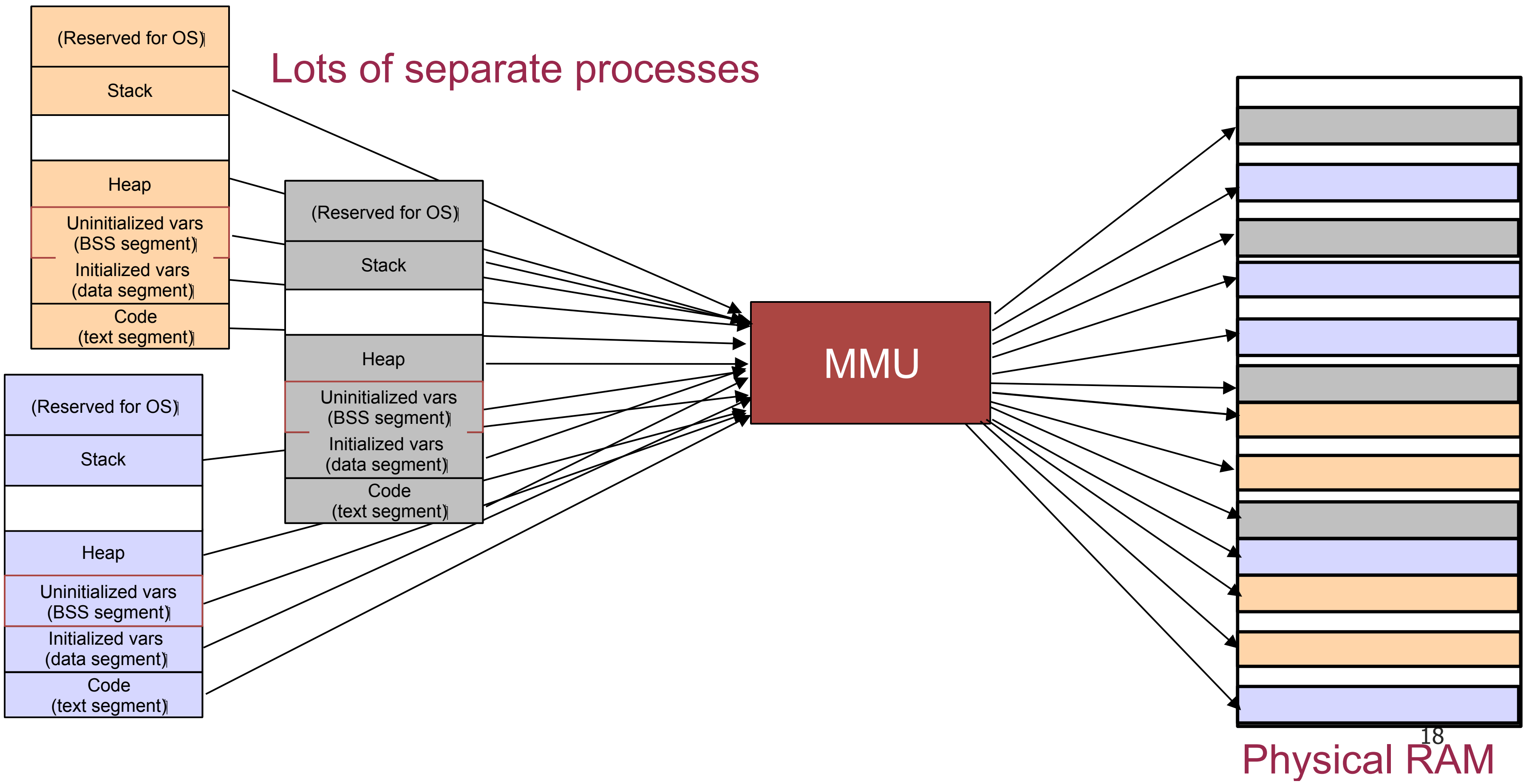
Paging

- Solve the external fragmentation problem by using **fixed-size chunks** of virtual and physical memory
  - Virtual memory unit called a **page**
  - Physical memory unit called a **frame** (or sometimes **page frame**)



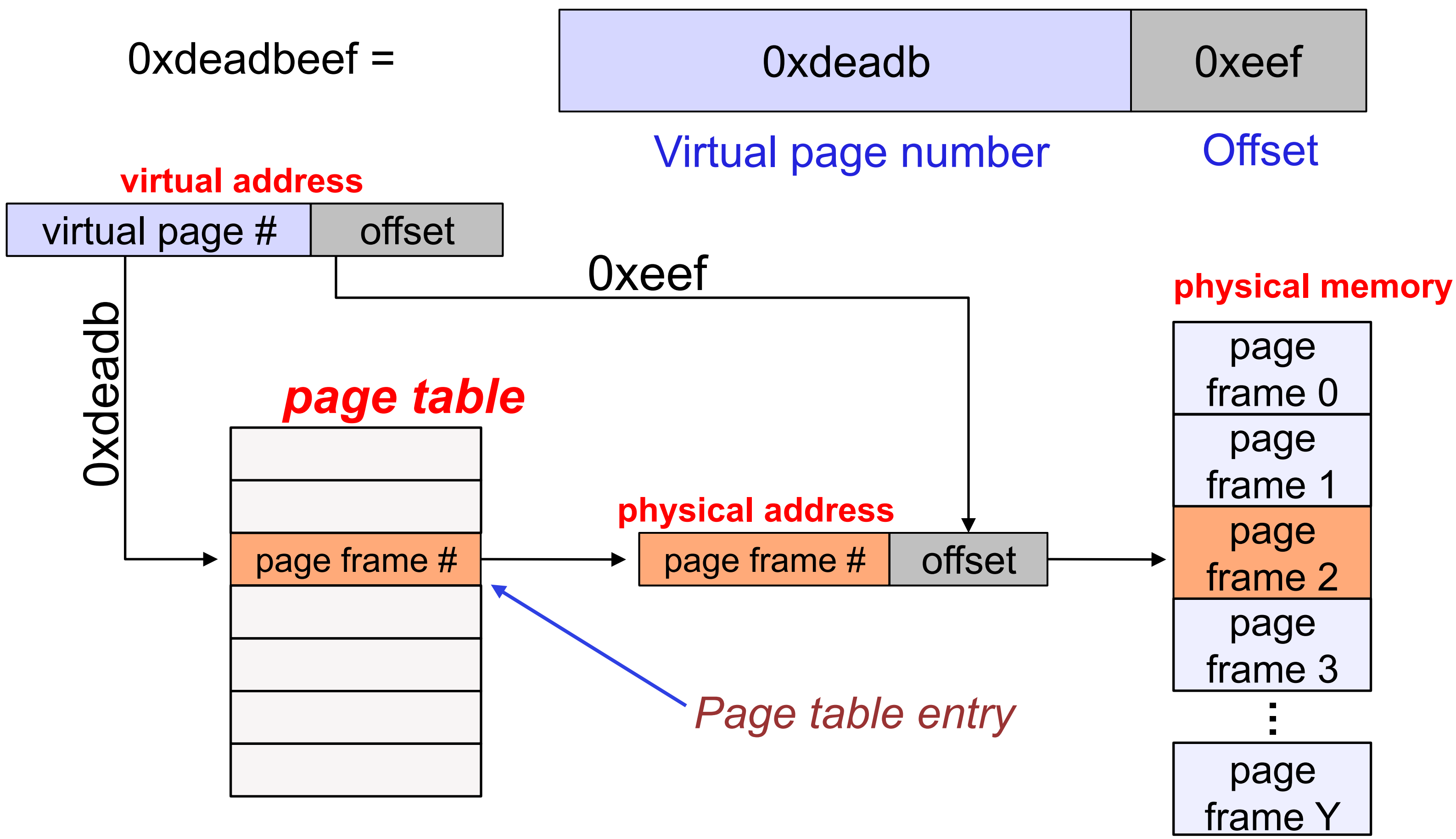
Application Perspective

- Application believes it has a single, contiguous address space ranging from 0 to 2P – 1 bytes
  - Where P is the number of bits in a pointer (e.g., 32 bits)
- In reality, virtual pages are scattered across physical memory
  - This mapping is invisible to the program, and not even under it's control!



Translation process

- Virtual-to-physical address translation performed by MMU
  - Virtual address is broken into a *virtual page number* and an *offset*
  - Mapping from virtual page to physical frame provided by a *page table* (which is stored in memory)



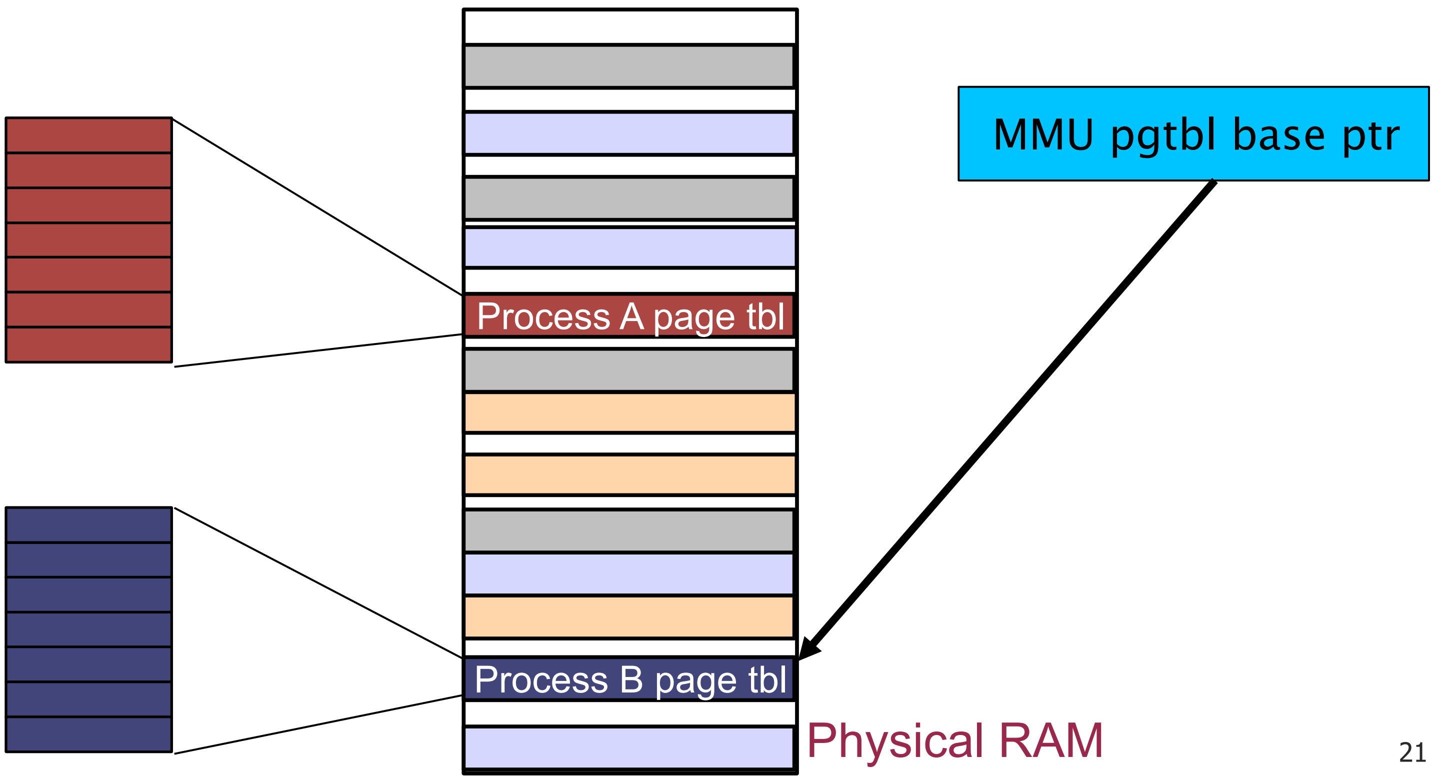
Translation process

```
if (virtual page is invalid or non-resident or protected)
    trap to OS fault handler
else
    physical frame # = pageTable[virtpage#].physPageNum
```

- Each virtual page can be in physical memory or swapped out to disk (called “paged out” or just “paged”)
- What must change on a context switch?
  - Could copy entire contents of table, but this will be slow
  - Instead use an extra layer of indirection: Keep pointer to current page table and just change pointer

Where is the page table?

- Page Tables store the virtual-to-physical address mappings.
- Where are they located? *In memory!*
- OK, then. How does the MMU access them?
  - The MMU has a special register called the *page table base pointer*.
  - This points to the *physical memory address* of the top of the page table for the currently-running process.



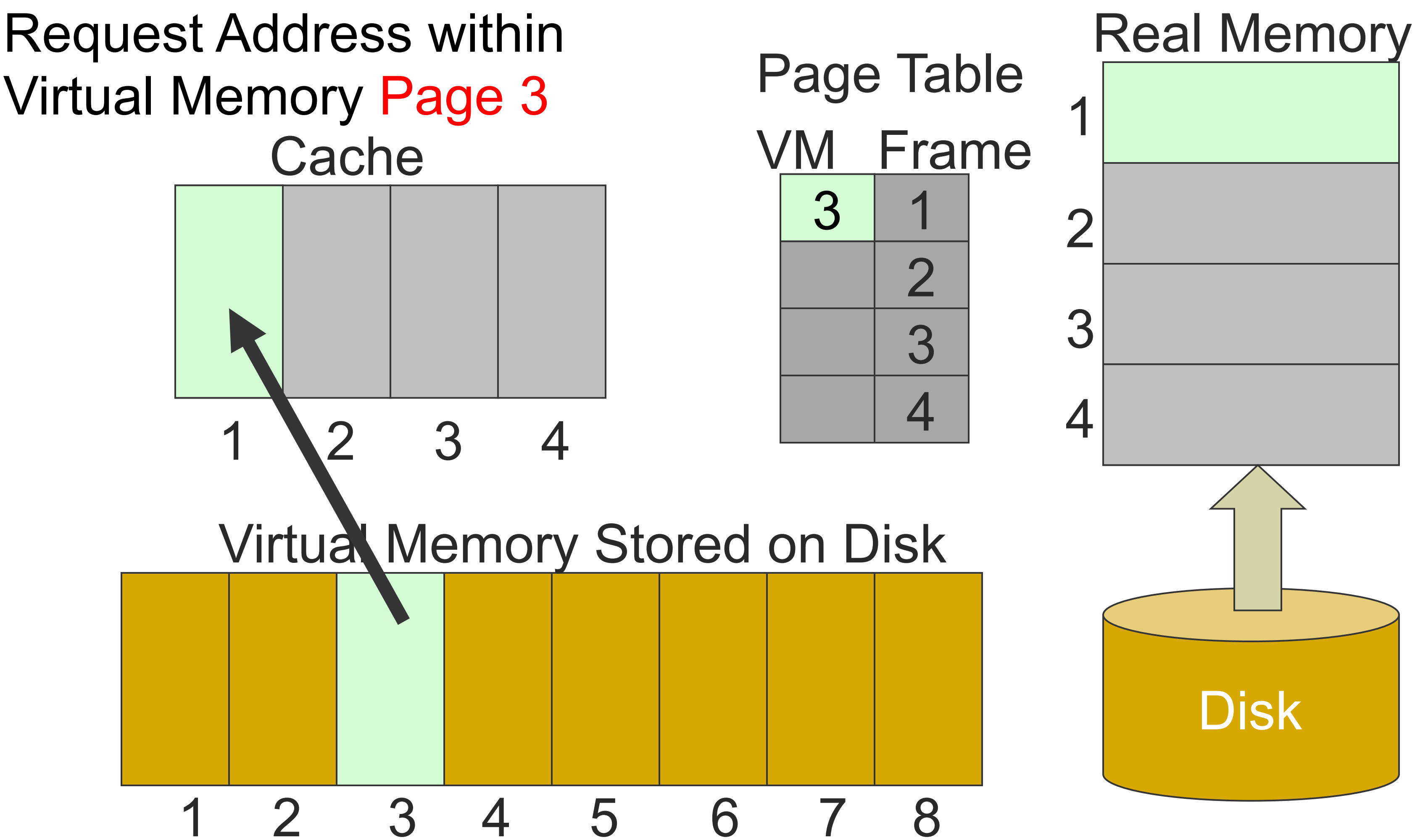
Page Faults

- What happens when a program accesses a virtual page that is not mapped into any physical page?
  - Hardware triggers a page fault
- Page fault handler
  - Find any available free physical page
  - If none, evict some resident page to disk
  - Allocate a free physical page
  - Load the faulted virtual page to the prepared physical page
  - Modify the page table

Advantages of Paging

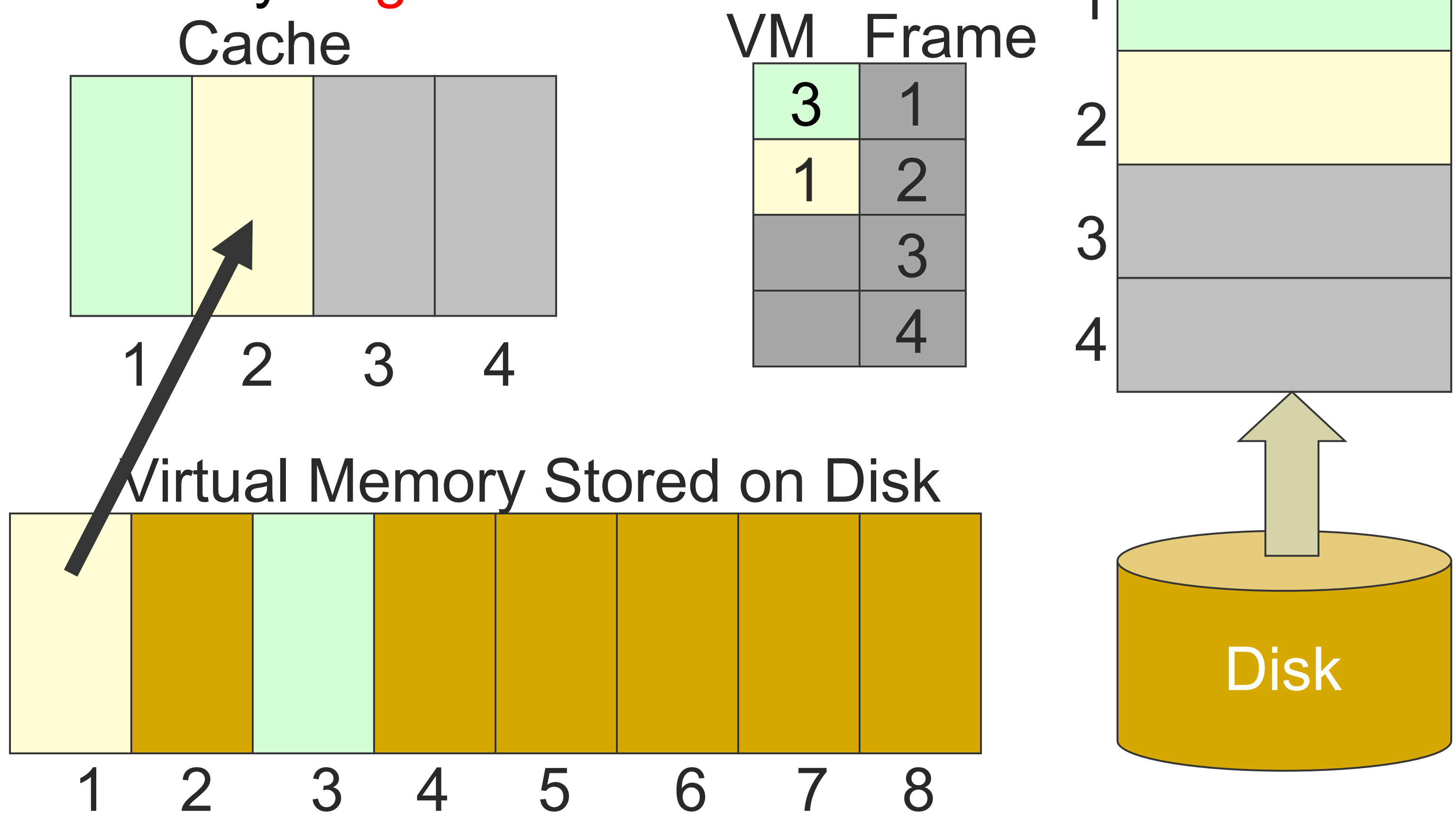
- Simplifies physical memory management
  - OS maintains a free list of physical page frames
  - To allocate a physical page, just remove an entry from this list
- No external fragmentation!
  - Virtual pages from different processes can be interspersed in physical memory
  - No need to allocate pages in a contiguous fashion
- Allocation of memory can be performed at a (relatively) fine granularity
  - Only allocate physical memory to those parts of the address space that require it
  - Can swap unused pages out to disk when physical memory is running low
  - Idle programs won't use up a lot of memory (even if their address space is huge!)

Paging Example



Paging Example

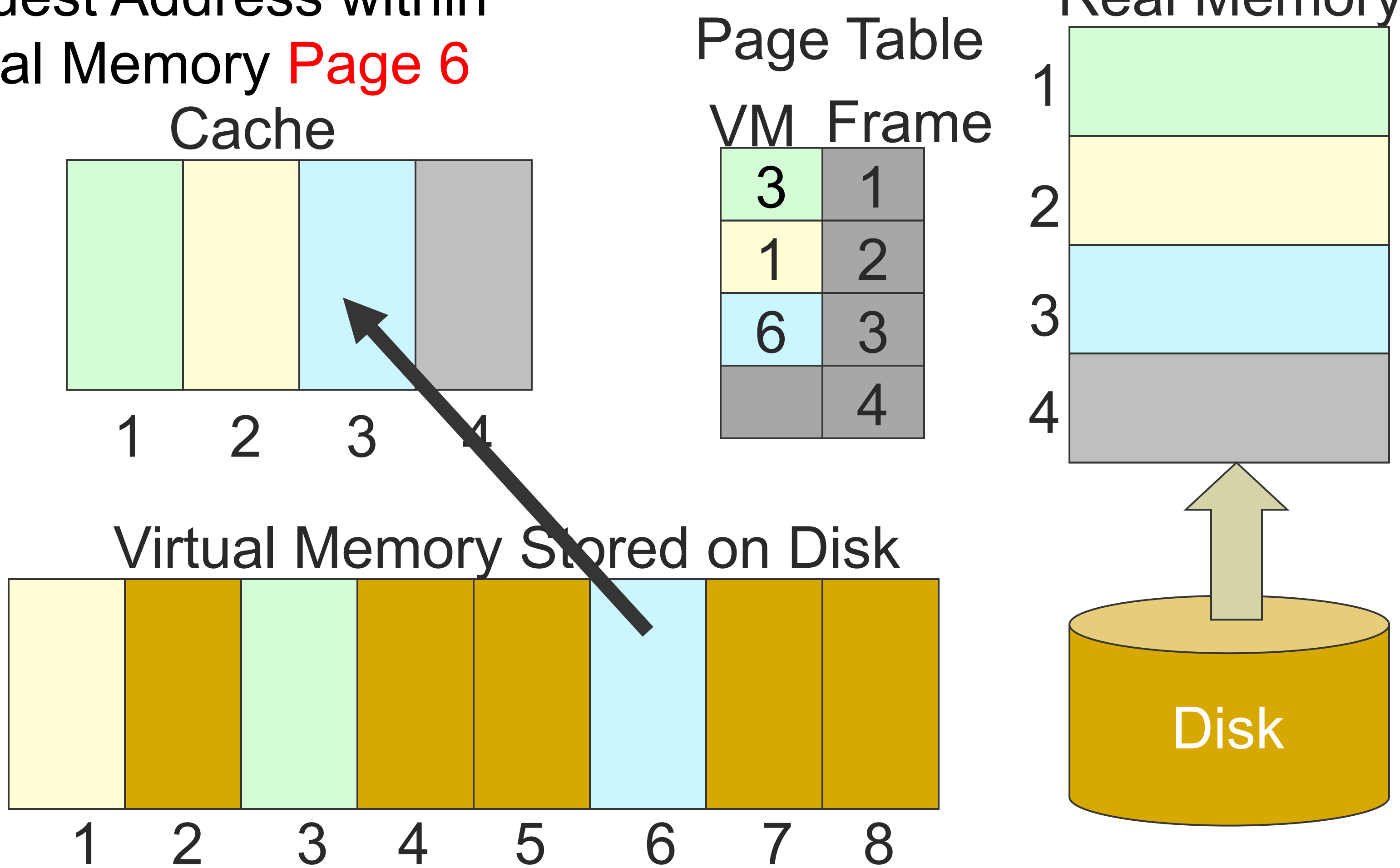
Request Address within Virtual Memory Page 1



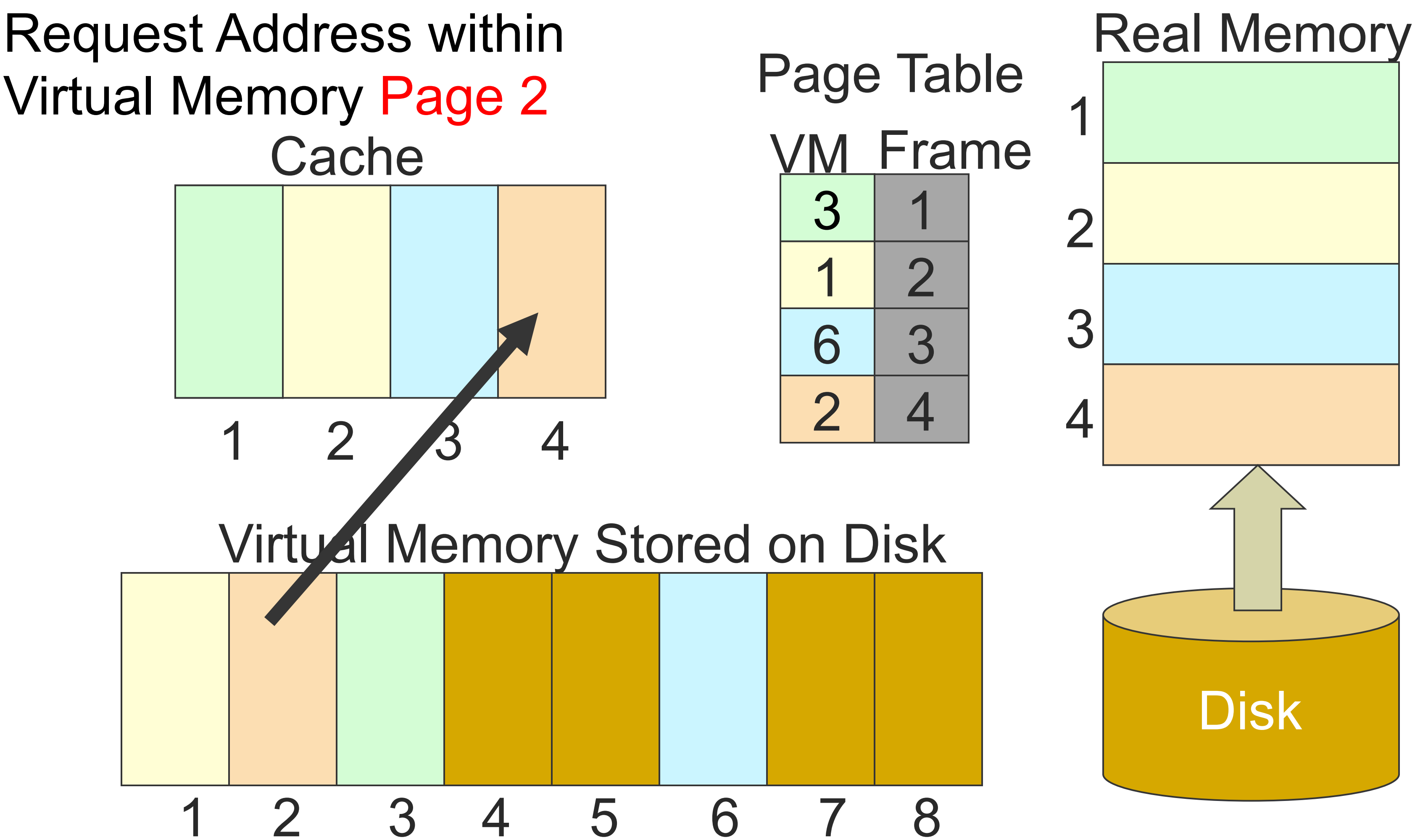
VM	Frame
3	1
1	2
	3
	4

Paging Example

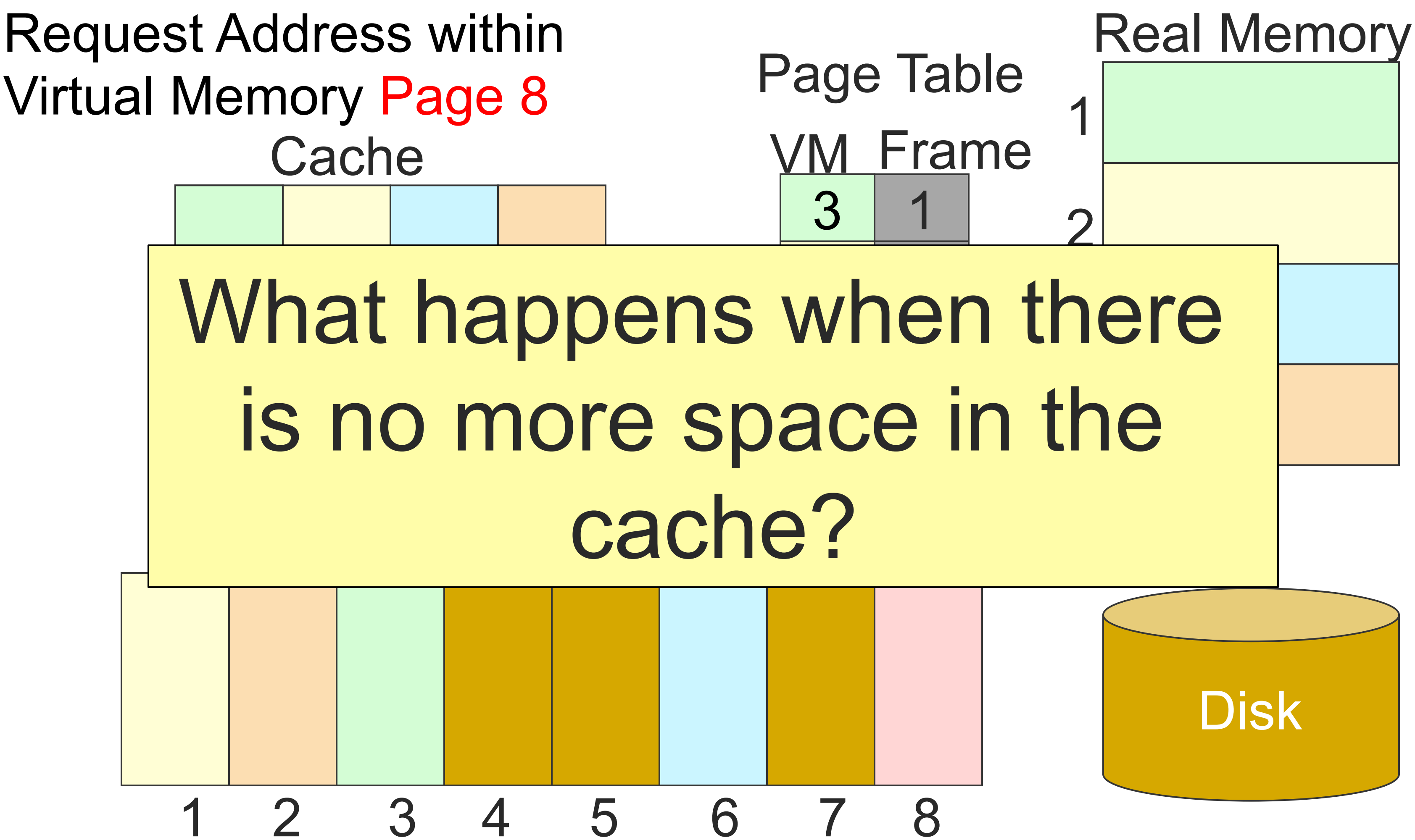
Request Address within Virtual Memory Page 6



Paging Example

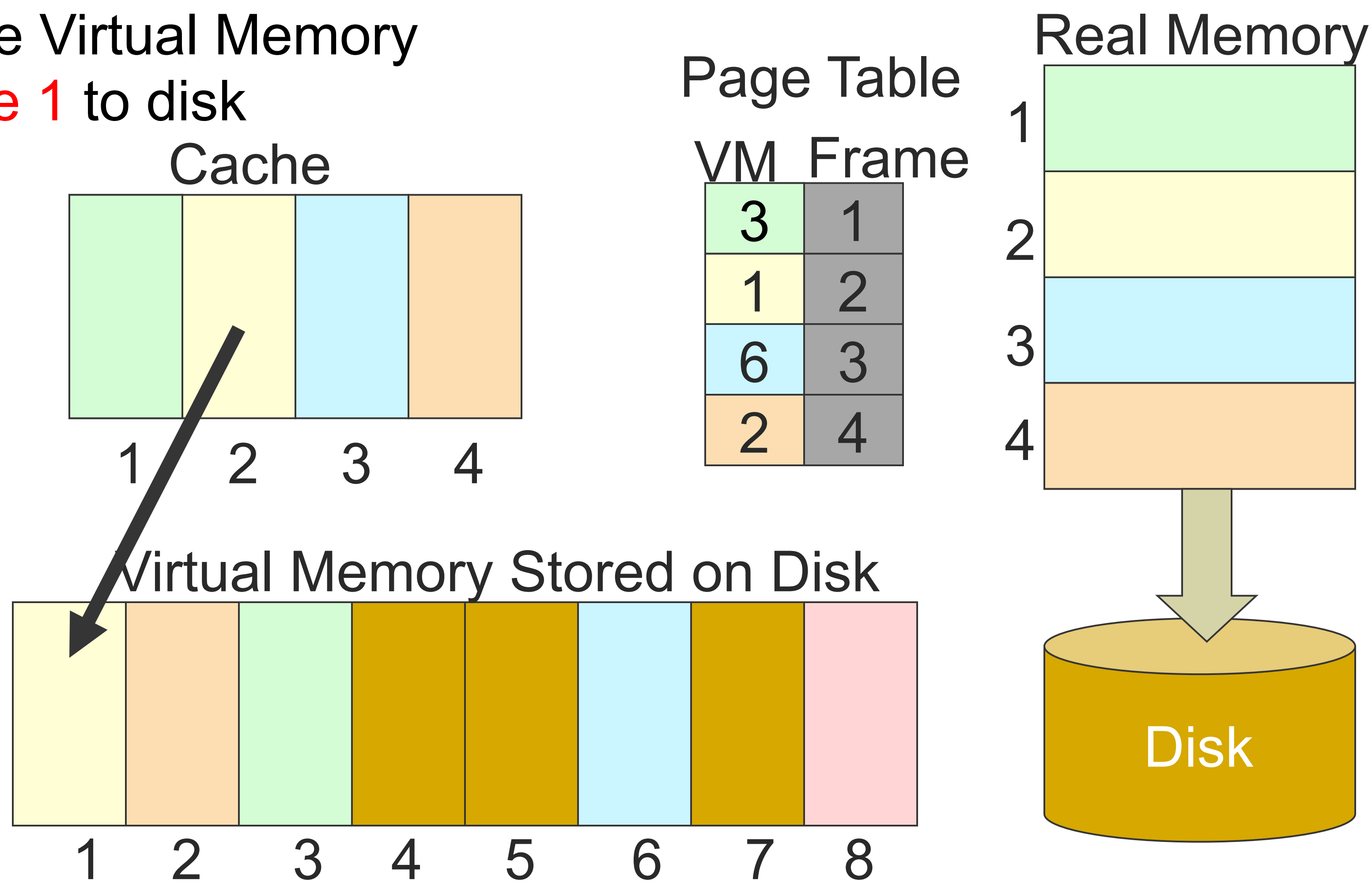


Paging Example



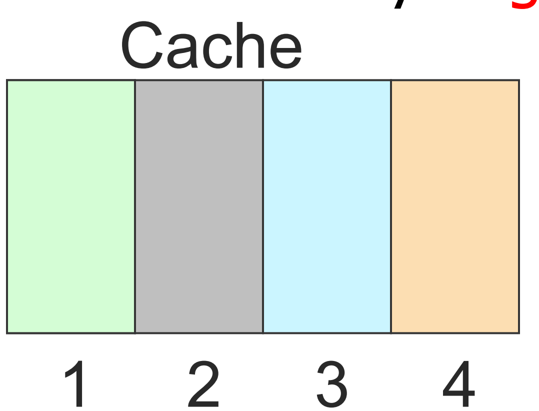
Paging Example

Store Virtual Memory Page 1 to disk



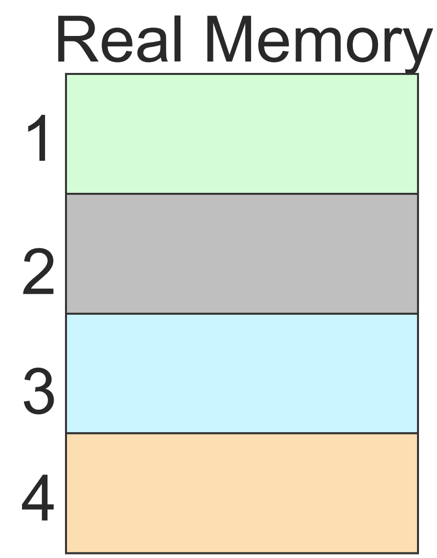
Paging Example

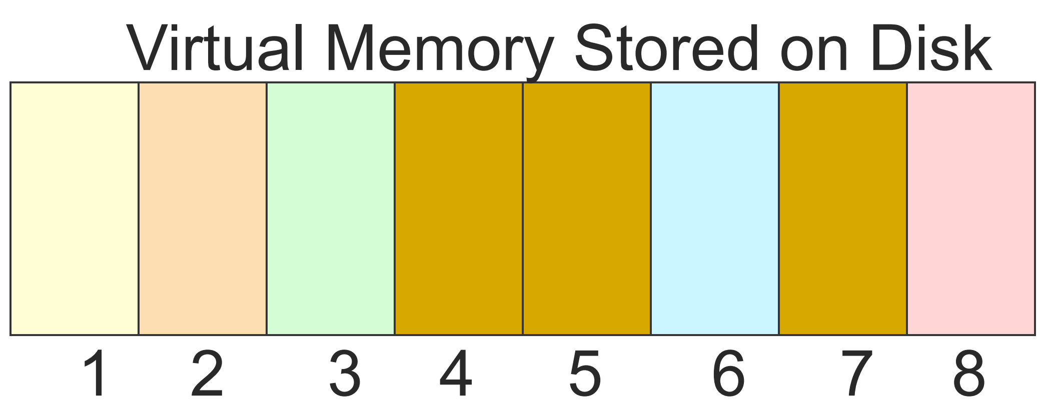
Process request for Address within Virtual Memory Page 8

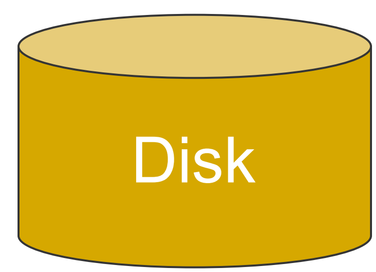


Page Table

VM	Frame
3	1
	2
6	3
2	4

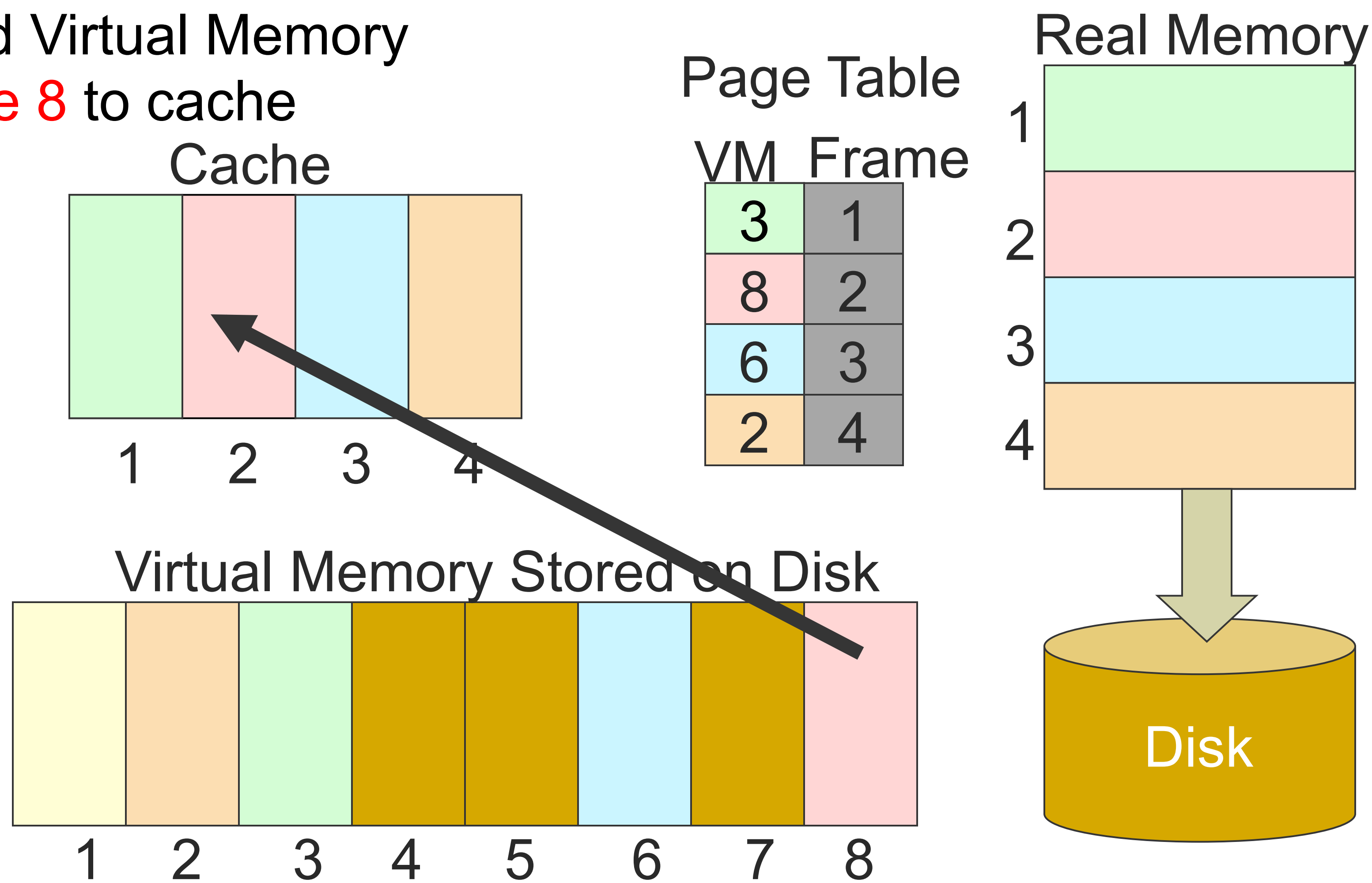




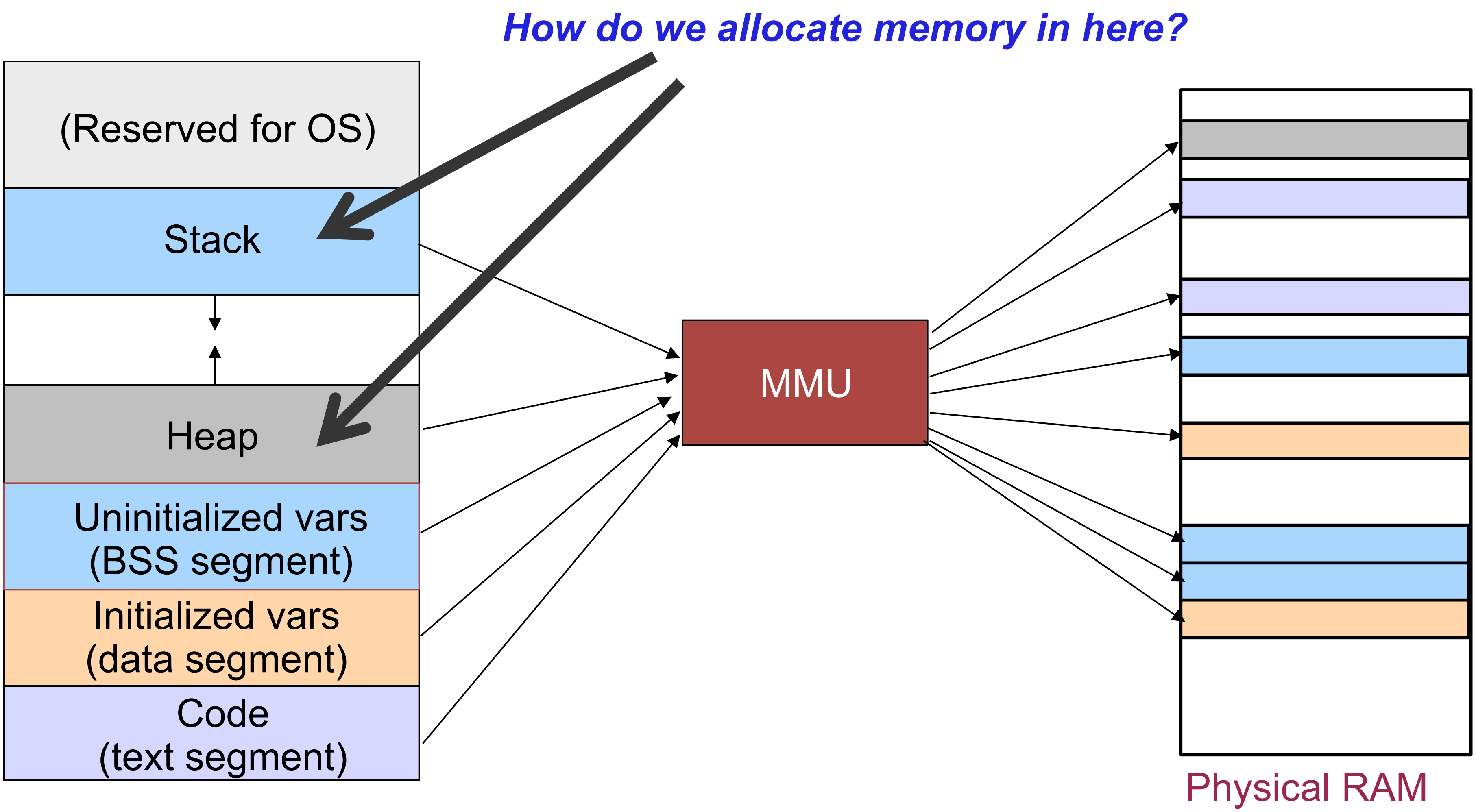


Paging Example

Load Virtual Memory
Page 8 to cache



Is paging enough?



Memory allocation w/in a process

- What happens when you declare a variable?
  - Allocating a page for every variable wouldn't be efficient
  - Allocations within a process are much smaller
  - Need to allocate on a finer granularity

- Solution (stack): stack data structure (duh)
  - Function calls follow LIFO semantics
  - So we can use a stack data structure to represent the process's stack – no fragmentation!

- Solution (heap): **malloc**
  - This is a much harder problem
  - Need to deal with fragmentation

MP2: malloc

- Introduction by Wade