Memory Allocation

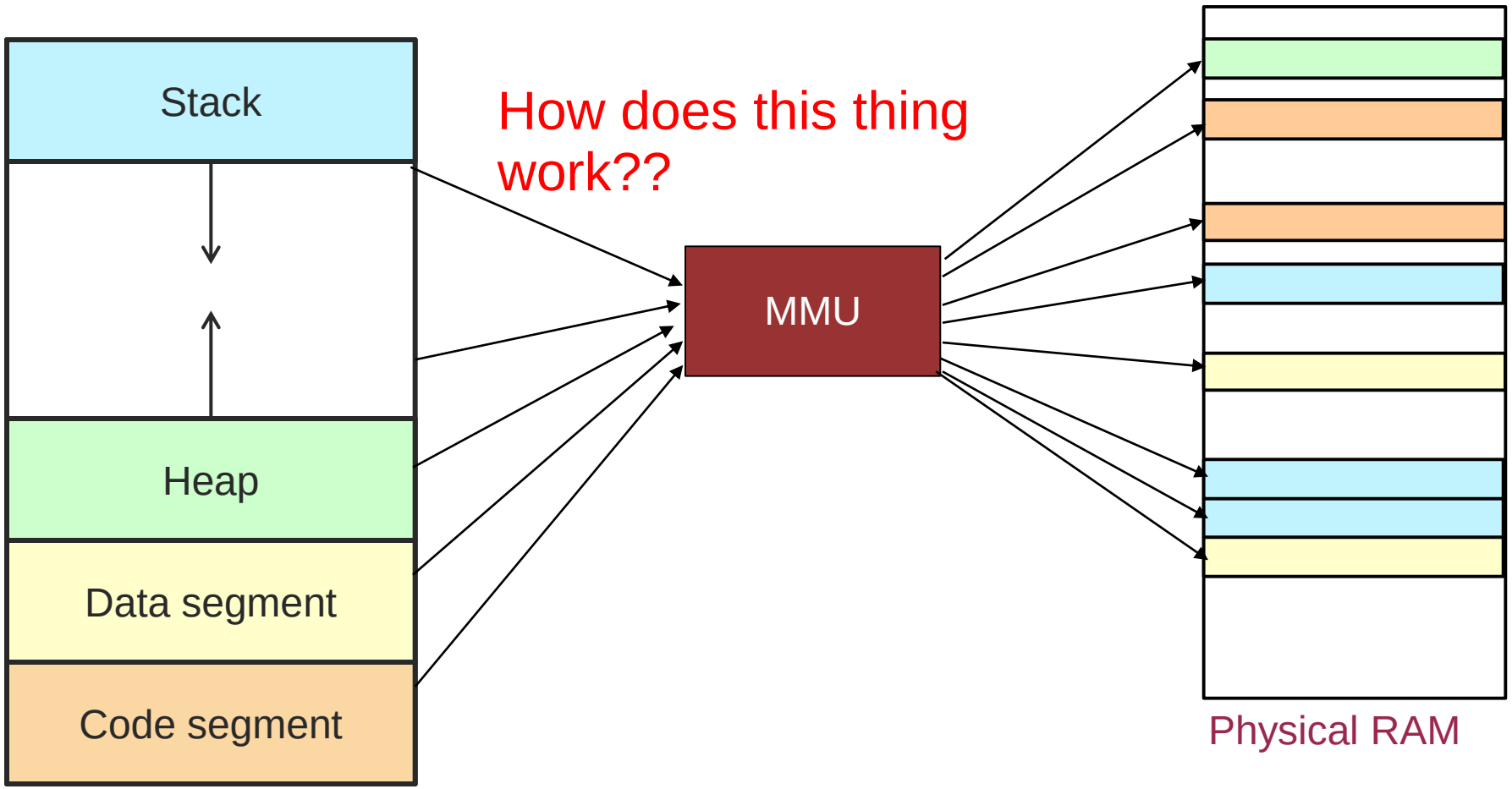
Recap: Virtual Addresses

- 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

Recap: Virtual Addresses

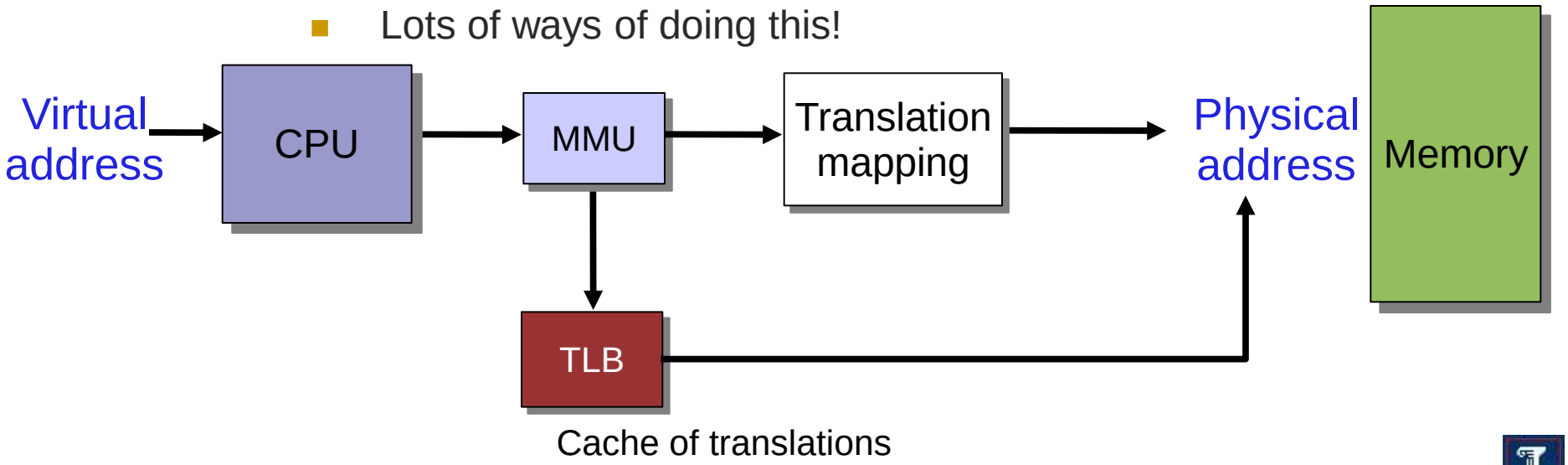
- Benefit: Isolation
  - Virtual addresses in one process refer to different physical memory than virtual addresses in another
  - Exception: shared memory regions between processes (discussed later)
- Benefit: Illusion of larger memory space
  - Can store unused parts of virtual memory on disk temporarily
- 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



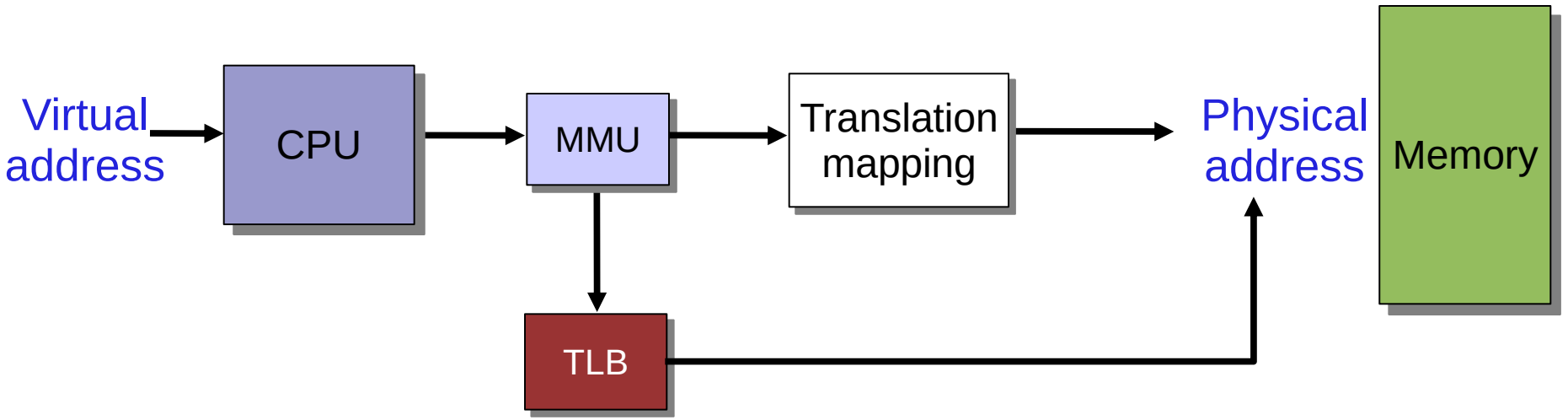
MMU and TLB

- 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!



MMU and TLB

- Translation Lookaside Buffer (TLB)
  - Cache for MMU virtual-to-physical address translations
  - Just an optimization – but an important one!



Translating virtual to physical

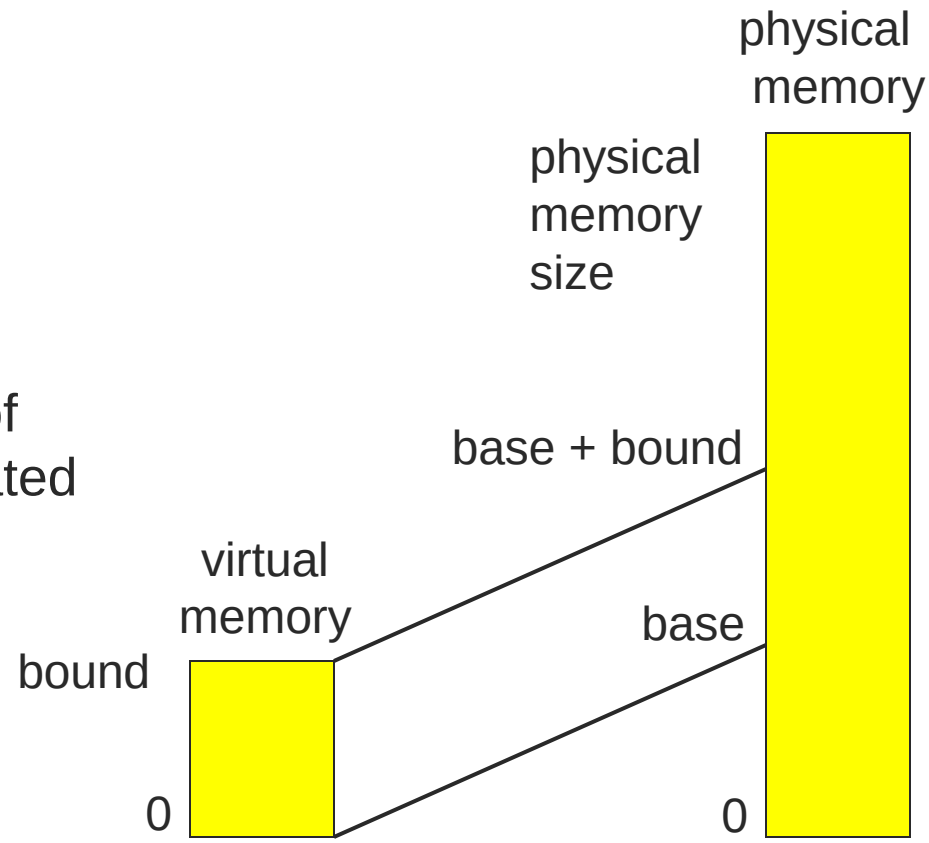
- Can do it almost any way we like
- But, some ways are better than others…

- Strawman solution from last time
  - Base and bound

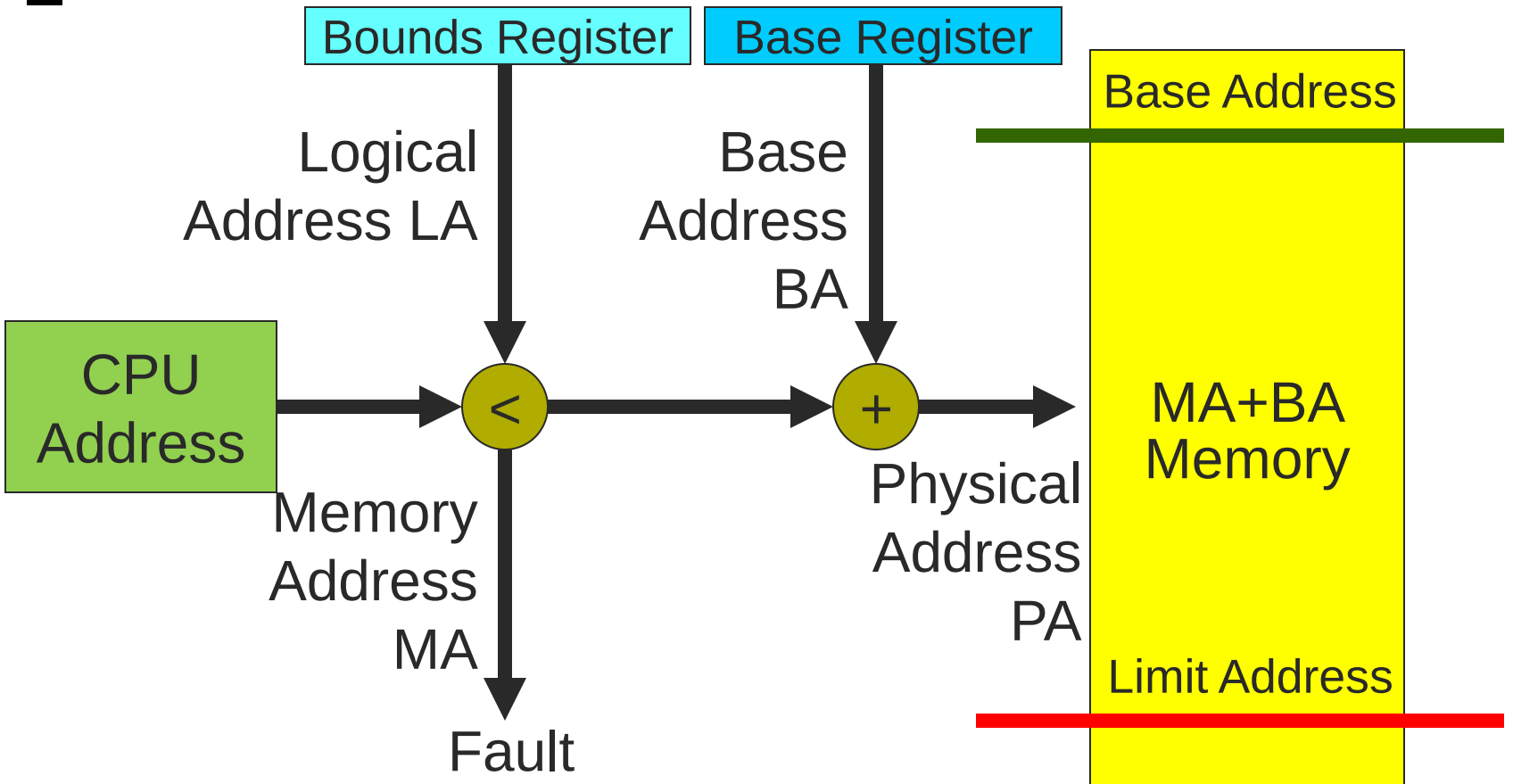
Base and bounds

```
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



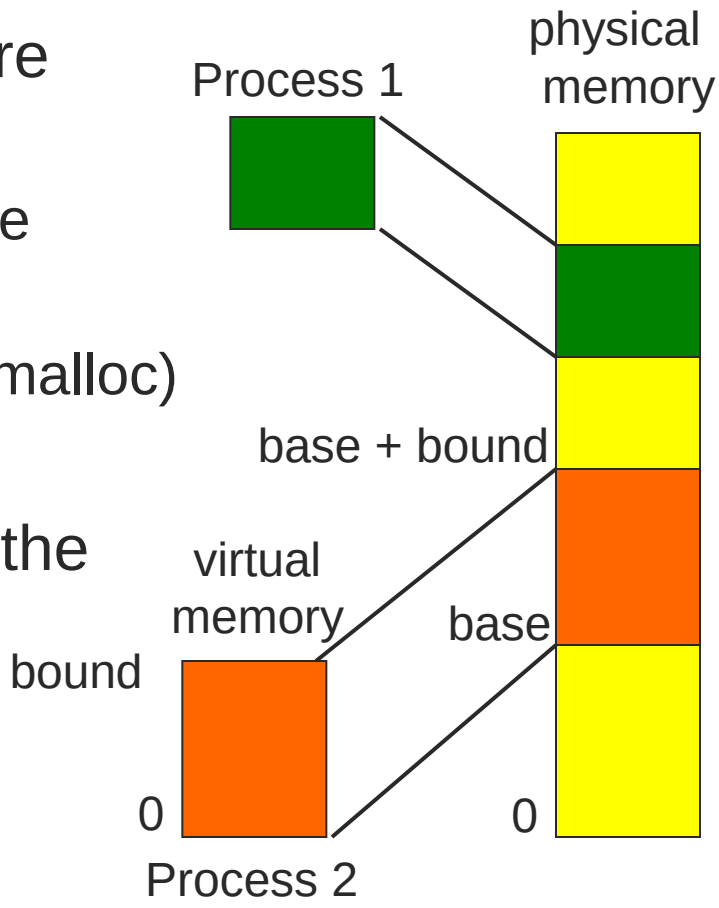
Base and Bounds Registers



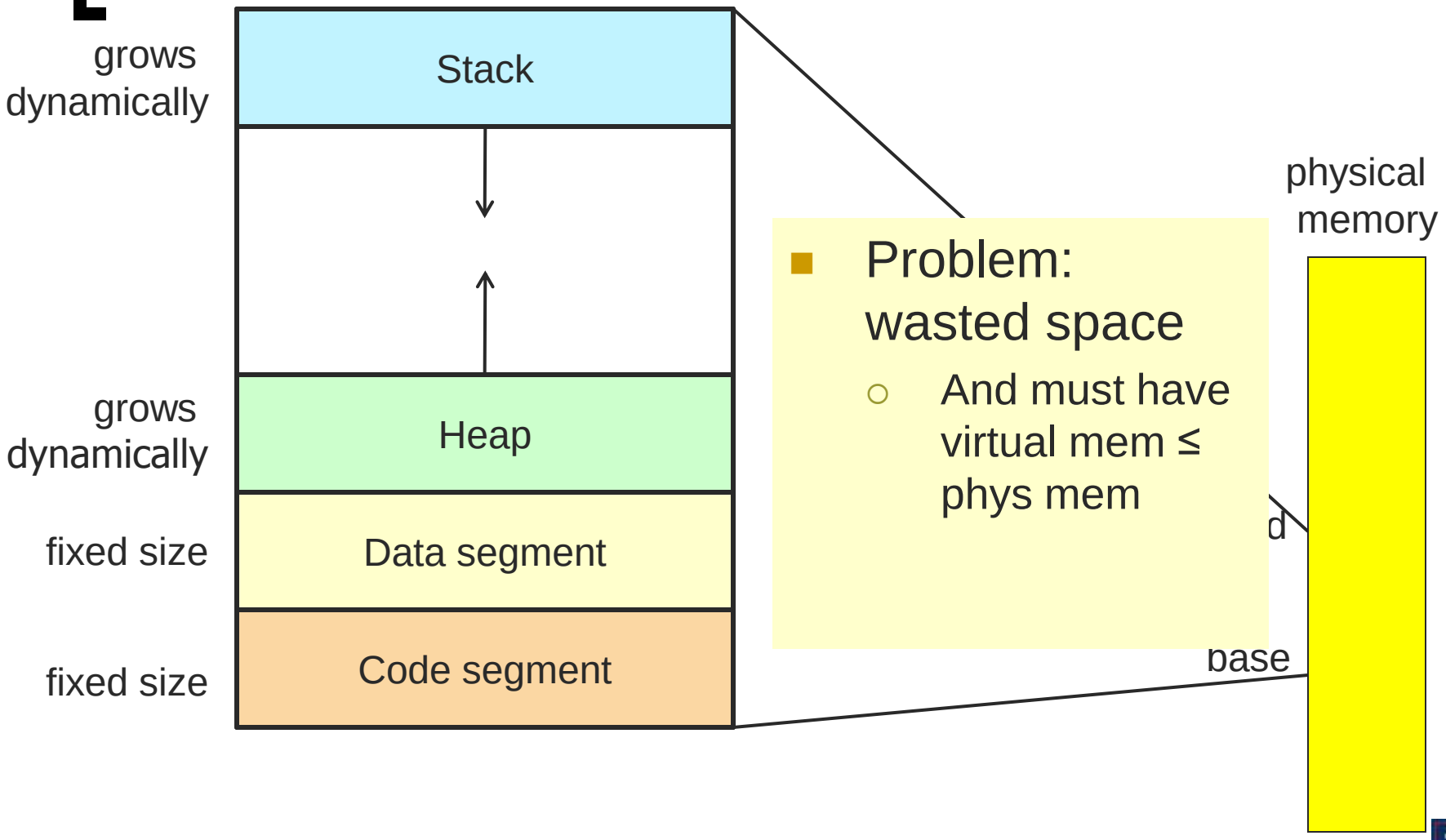
Base: start of the process's memory partition
Limit: max address in the process's memory partition

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 algorithm for allowing processes to grow



But wait, didn't we solve this?

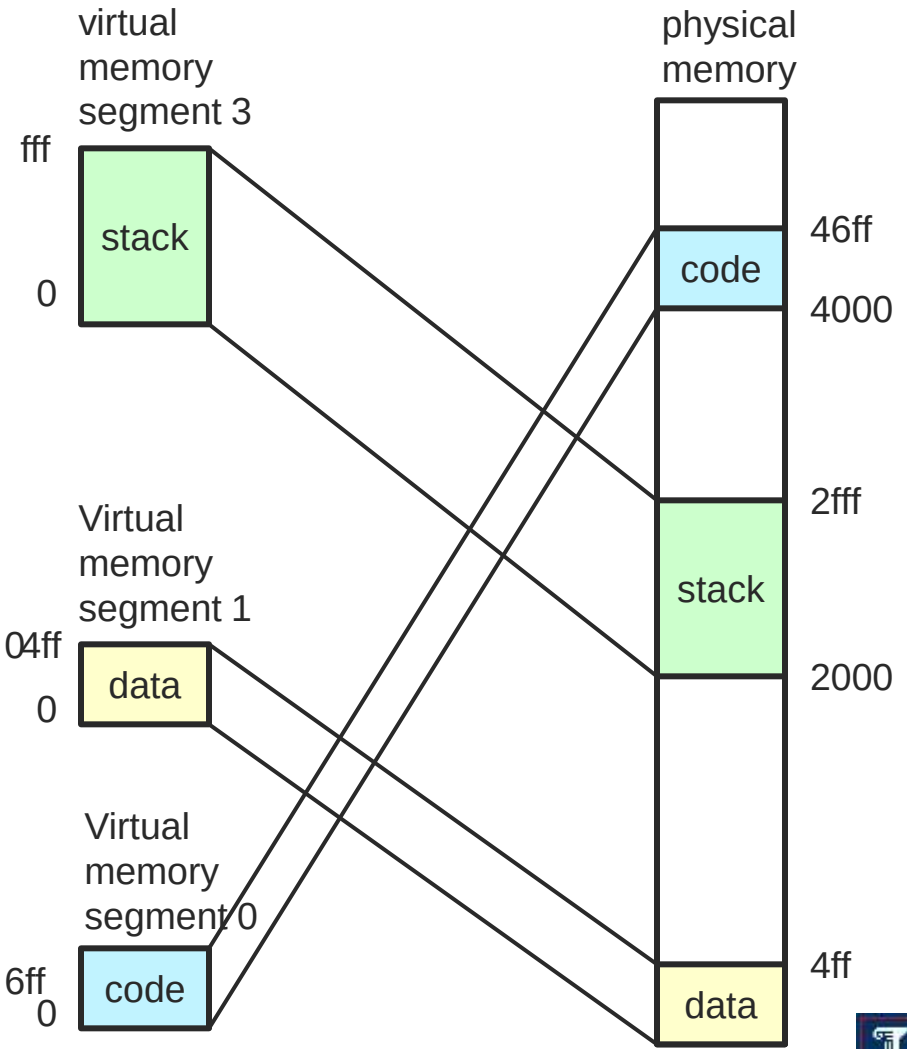


Another attempt: Segmentation

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

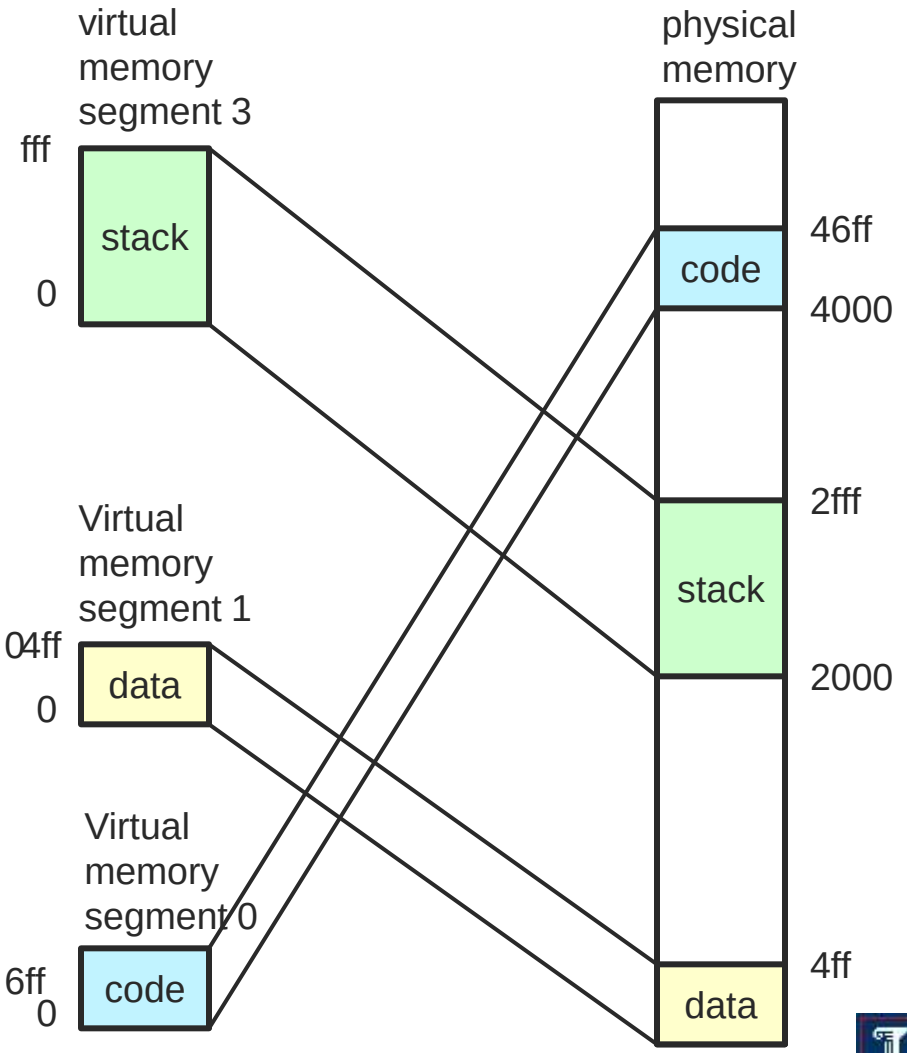
Segmentation

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



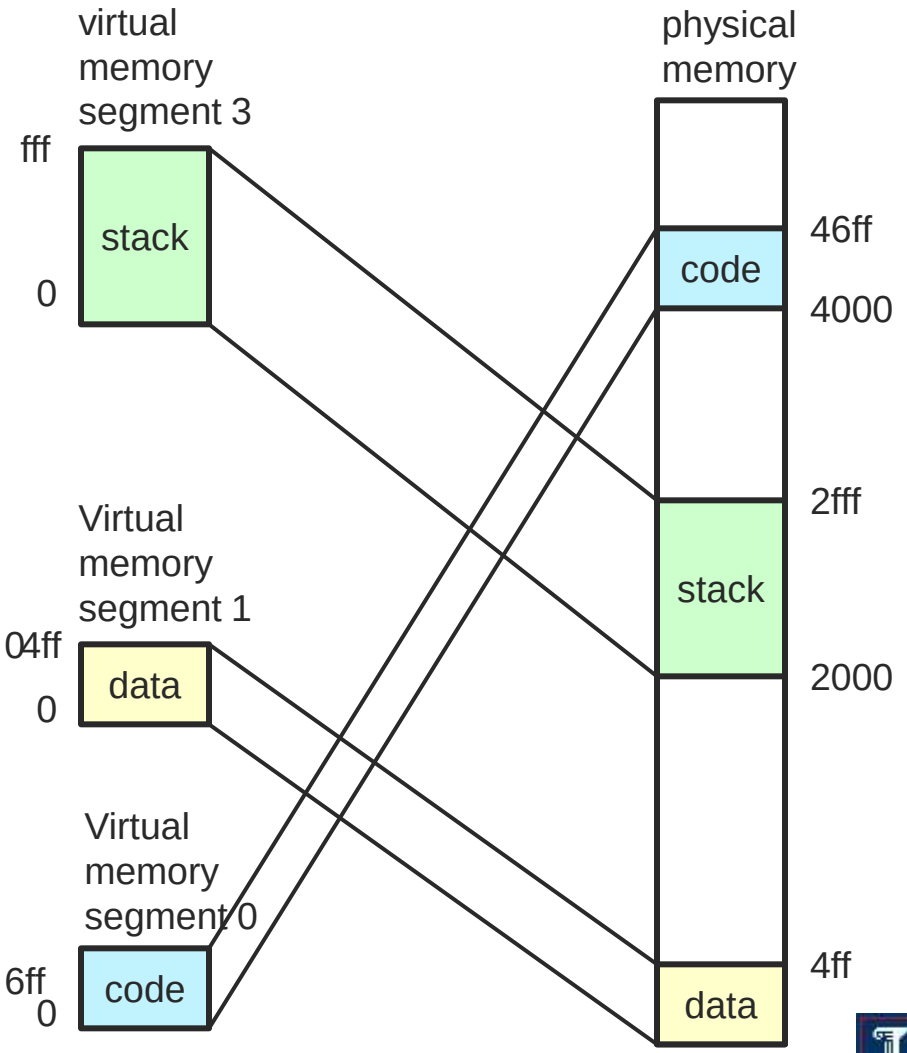
Segmentation

- 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
- 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)

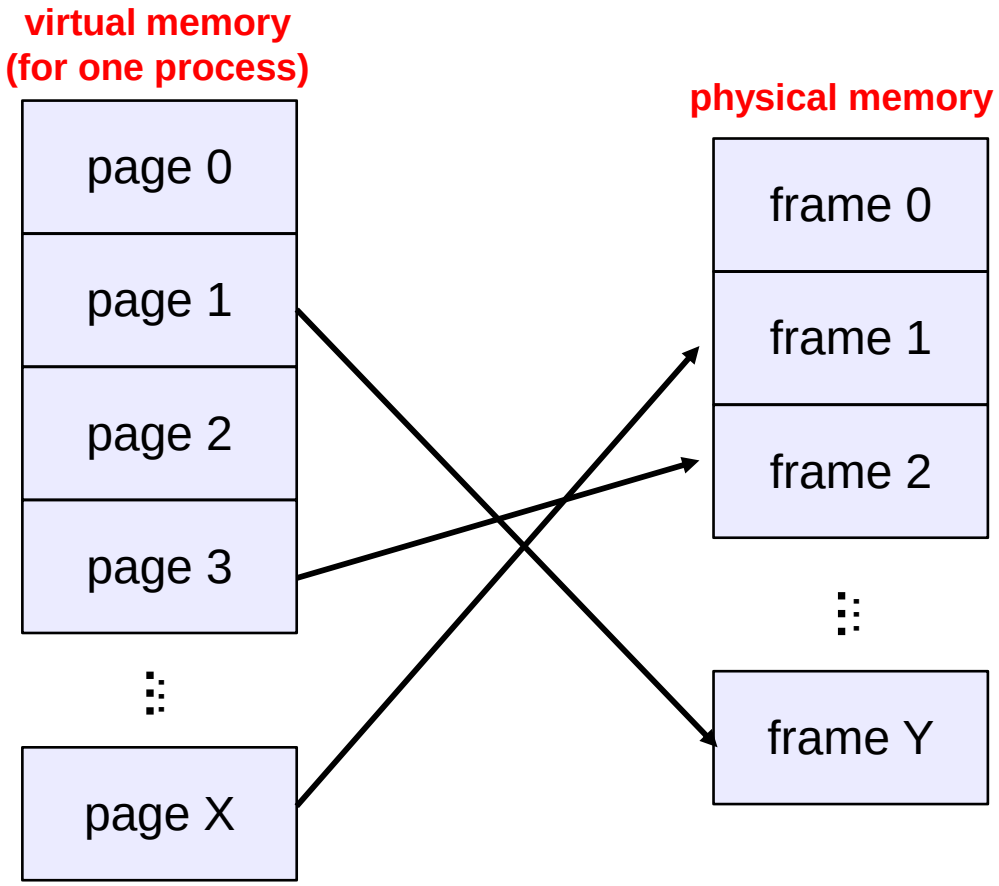


Recap: Mapping Virtual Memory

- 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

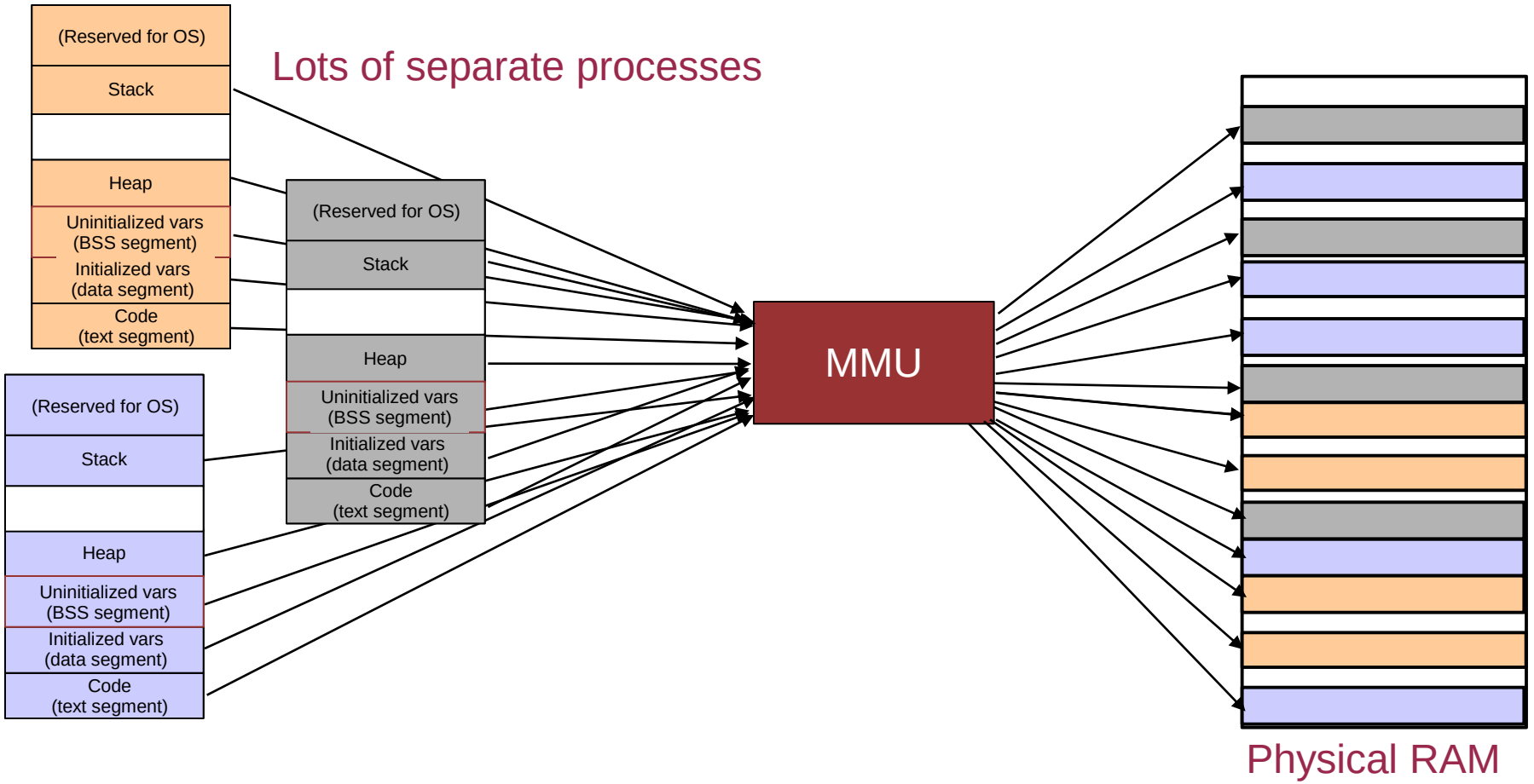
- Solve the external fragmentation problem by using fixed-size chunks of virtual and physical memory
  - Virtual memory unit called a page
  - Physical memory unit called a frame (or sometimes page frame)



Application Perspective

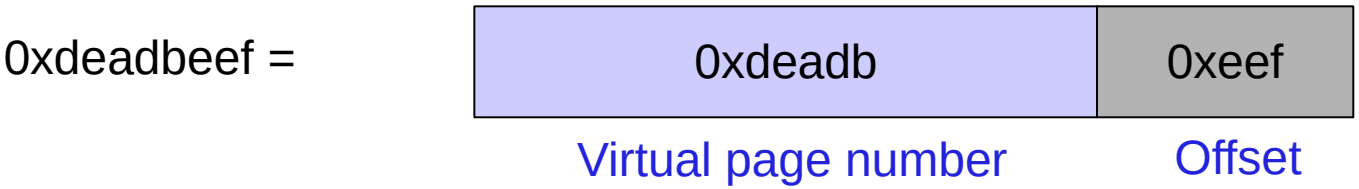
- Application believes it has a single, contiguous address space ranging from 0 to 2P – 1 bytes
  - Where P is the number of bits in a pointer (e.g., 32 bits)
- In reality, virtual pages are scattered across physical memory
  - This mapping is invisible to the program, and not even under it's control!

Application Perspective

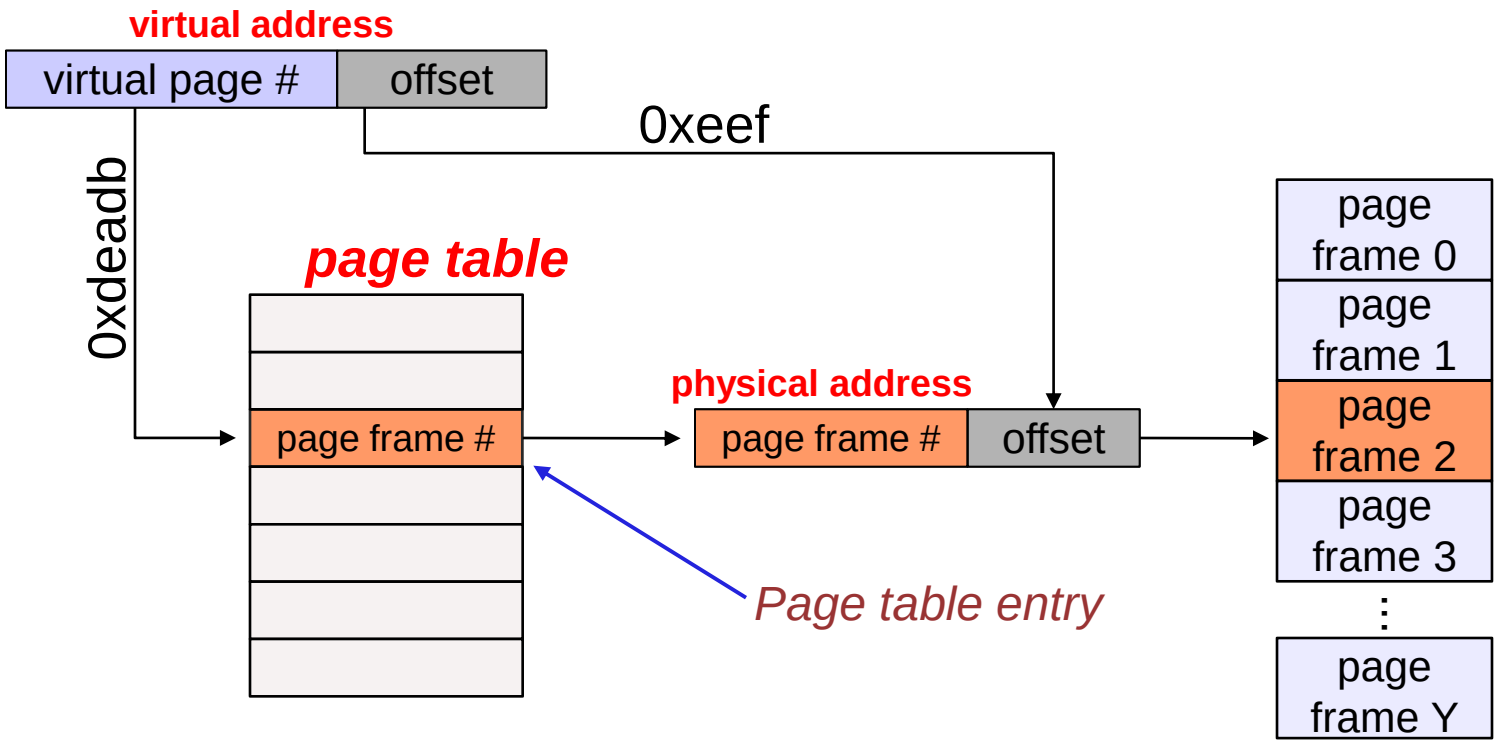


Translation process

- 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



Translation process

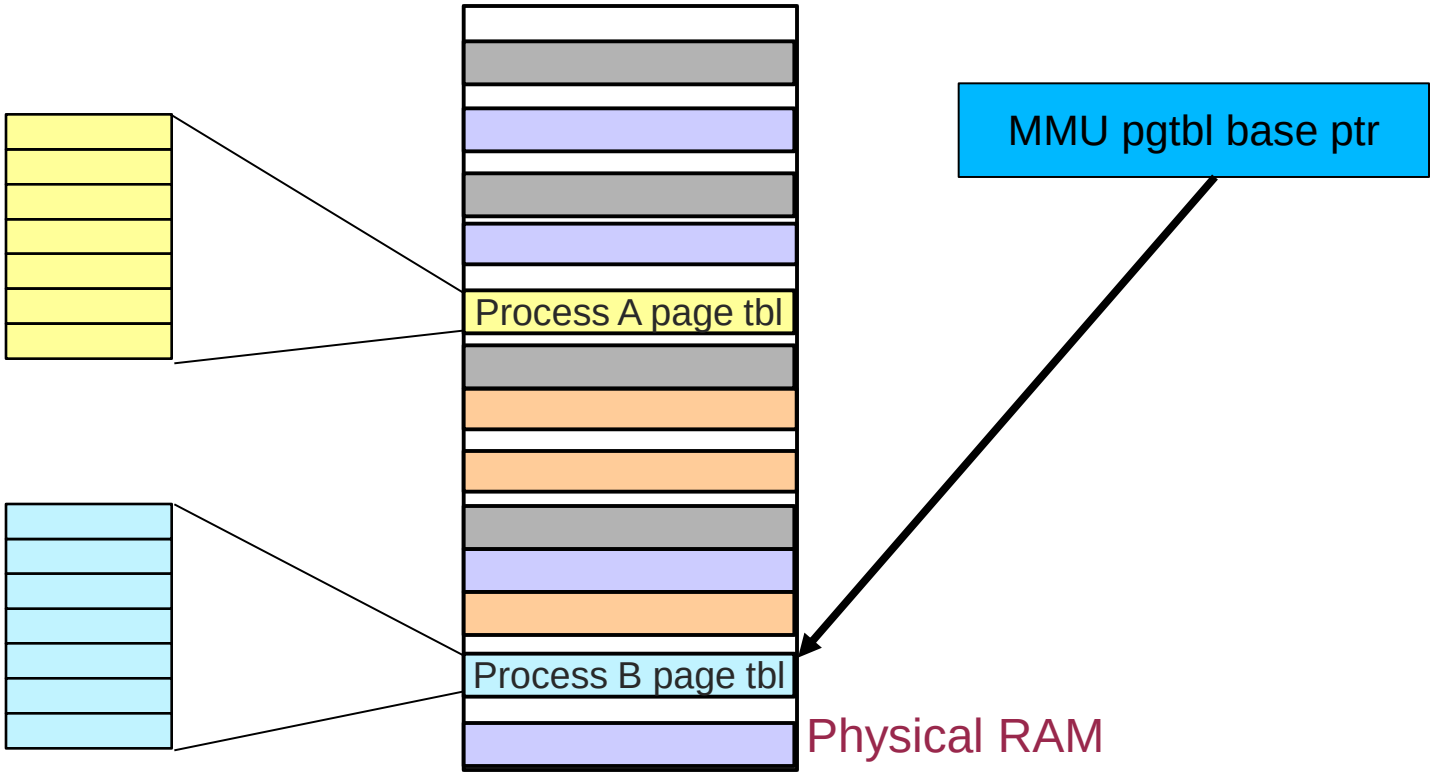
```
if (virtual page is invalid or non-resident or protected)
    trap to OS fault handler
else
    physical frame # = pageTable[virtpage#].physPageNum
```

- 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

Where is the page table?



Paging

- Can add read, write, execute protection bits to page table to protect memory
  - Check is done by hardware during access
  - Can give shared memory location different protections from different processes by having different page table protection access bits
- How does the processor know that a virtual page is not in memory?
  - Resident bit tells the hardware that the virtual address is resident or non-resident

Valid vs. Resident

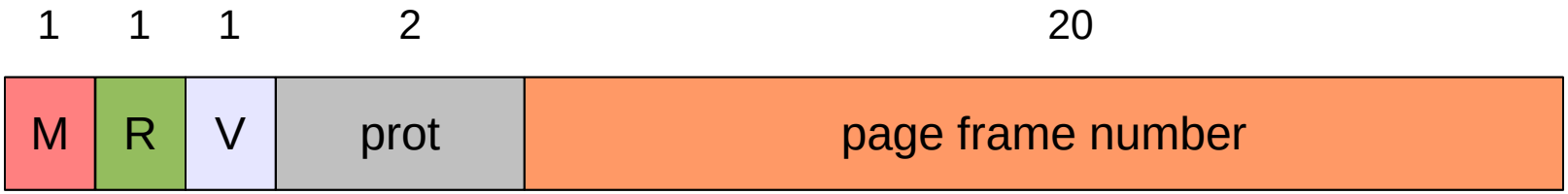
- Resident
  - Virtual page is in memory
  - NOT an error for a program to access non-resident page
- Valid
  - Virtual page is legal for the program to access
  - e.g., part of the address space

Valid vs. Resident

- Who makes a page resident/non-resident?
  - OS memory manager
- Who makes a virtual page valid/invalid?
  - User actions
- Why would a process want one if its virtual pages to be invalidated?
  - Avoid accidental memory references to bad locations

Page Table Entry

- Typical PTE format (depends on CPU architecture!)



- Various bits accessed by MMU on each page access:
  - Modify bit: Indicates whether a page is “dirty” (modified)
  - Reference bit: Indicates whether a page has been accessed (read or written)
  - Valid bit: Whether the PTE represents a real memory mapping
  - Protection bits: Specify if page is readable, writable, or executable
  - Page frame number: Physical location of page in RAM
    - Why is this 20 bits wide in the above example?

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

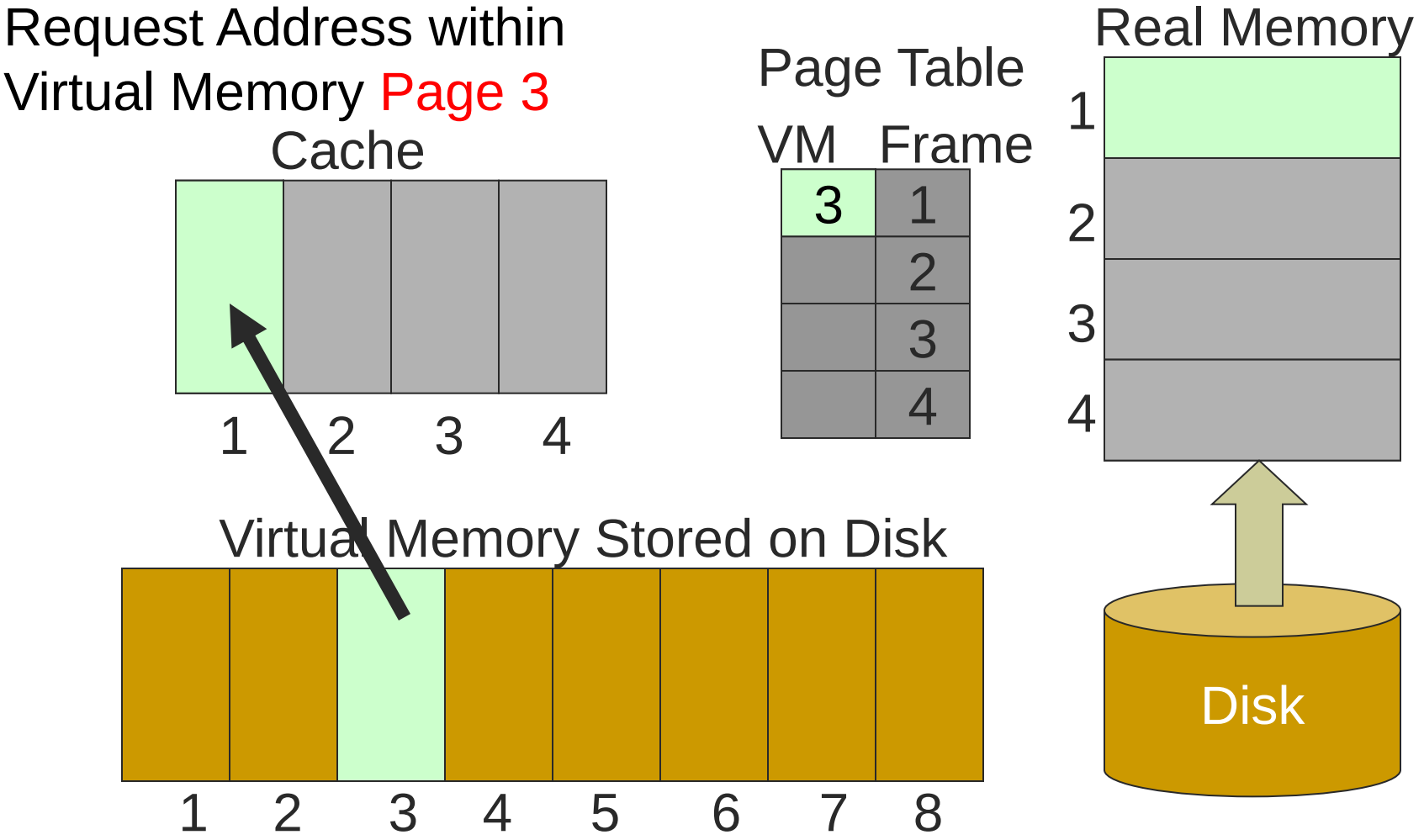
- Simplifies physical memory management
  - OS maintains a free list of physical page frames
  - To allocate a physical page, just remove an entry from this list
- No external fragmentation!
  - Virtual pages from different processes can be interspersed in physical memory
  - No need to allocate pages in a contiguous fashion

Advantages of Paging

- 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!)

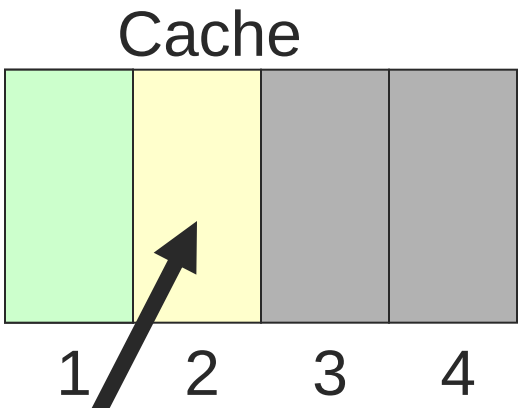
Paging Example

Request Address within Virtual Memory Page 3



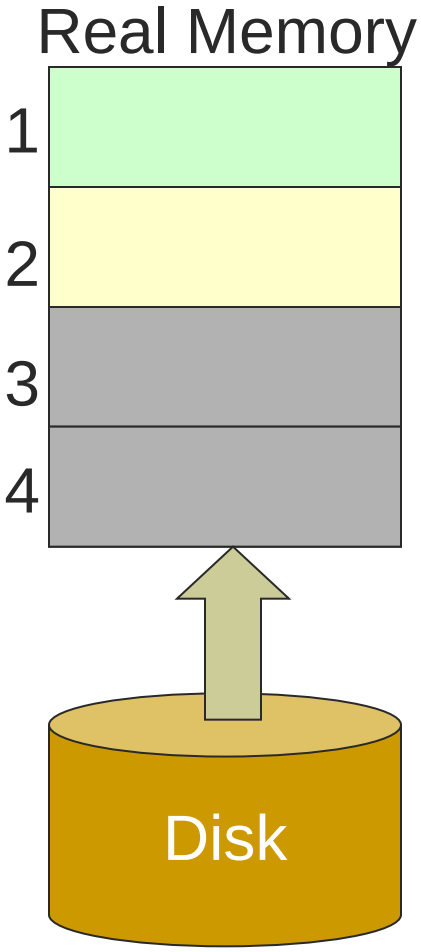
Paging Example

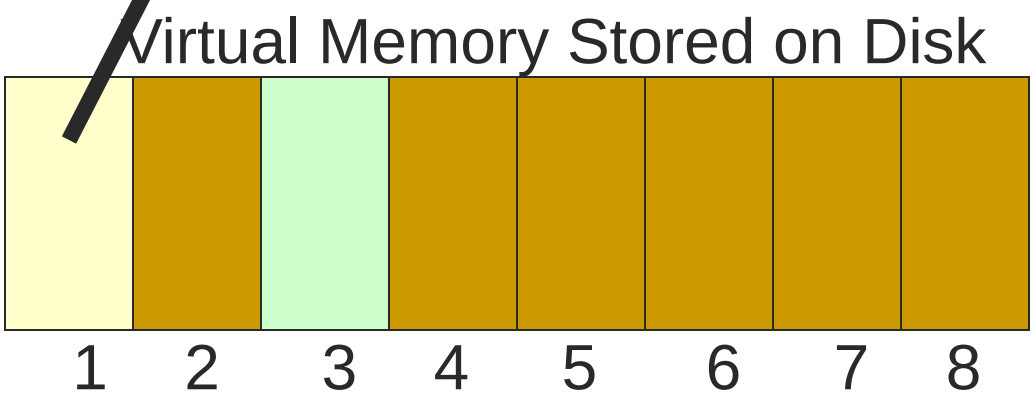
Request Address within Virtual Memory Page 1



Page Table

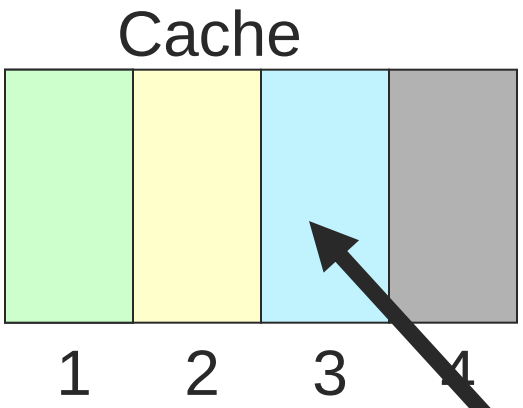
VM	Frame
3	1
1	2
	3
	4





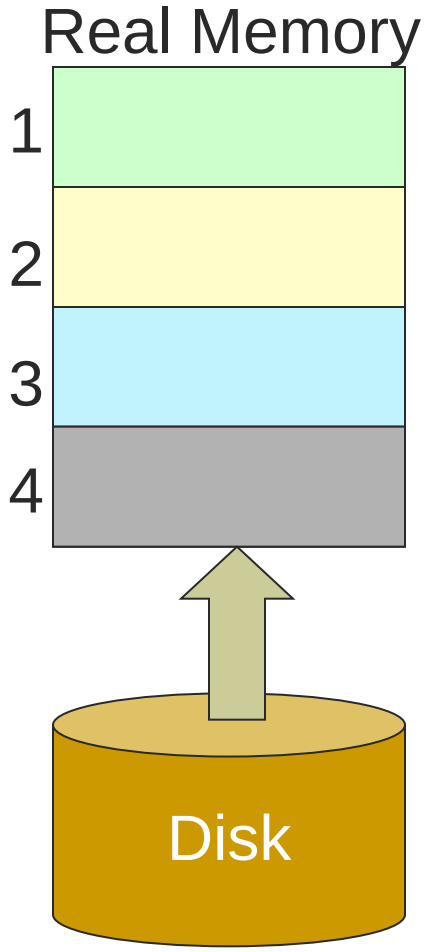
Paging Example

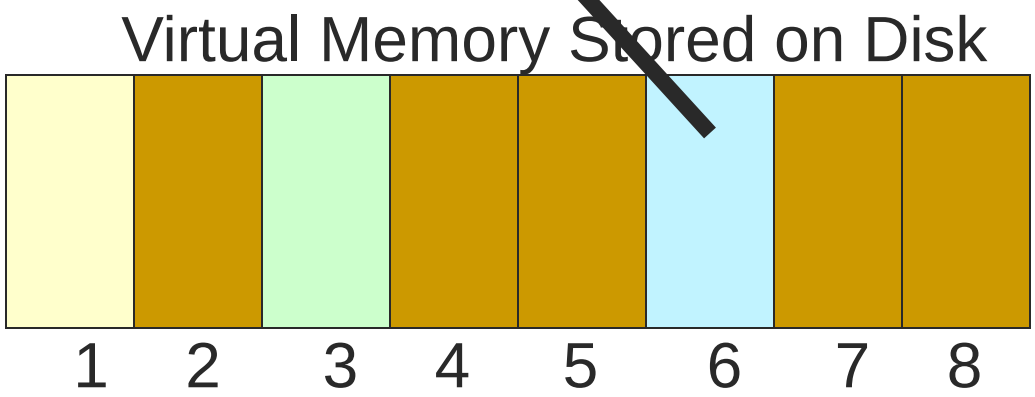
Request Address within Virtual Memory Page 6



Page Table

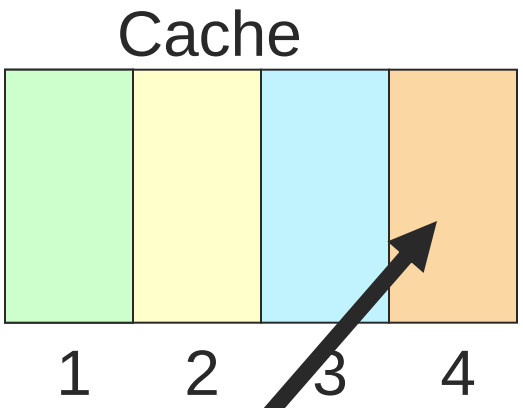
VM	Frame
3	1
1	2
6	3	
	4





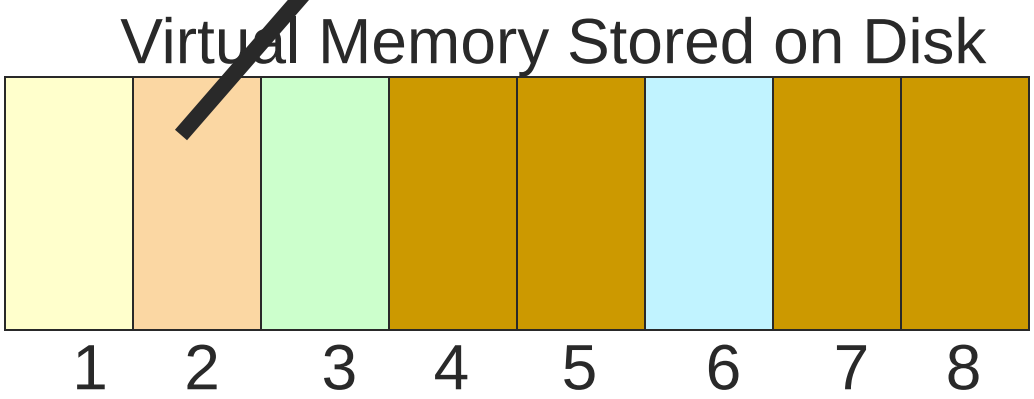
Paging Example

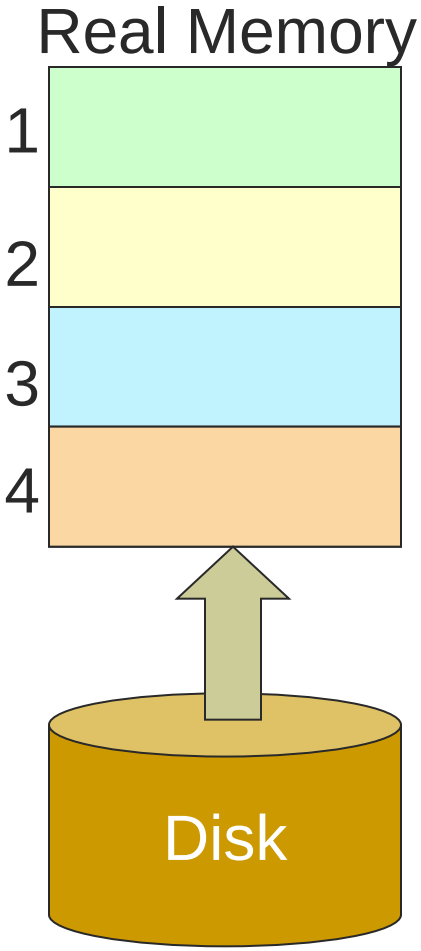
Request Address within Virtual Memory Page 2



Page Table

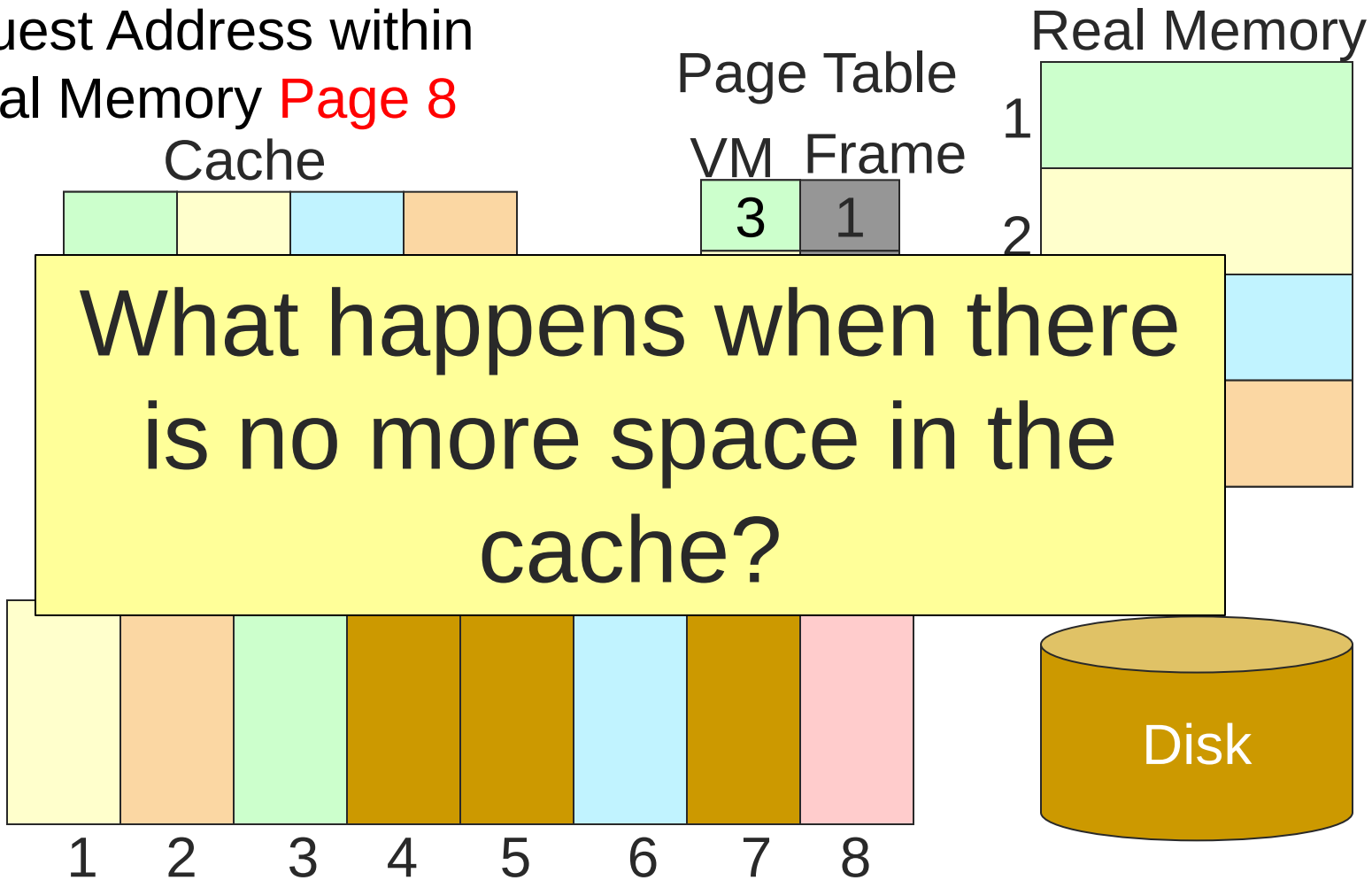
VM	Frame
3	1
1	2
6	3
2	4





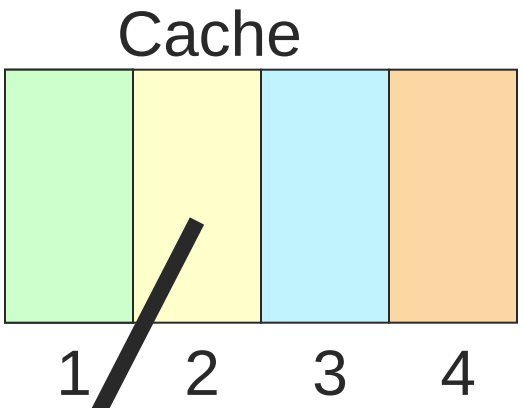
Paging Example

Request Address within Virtual Memory Page 8



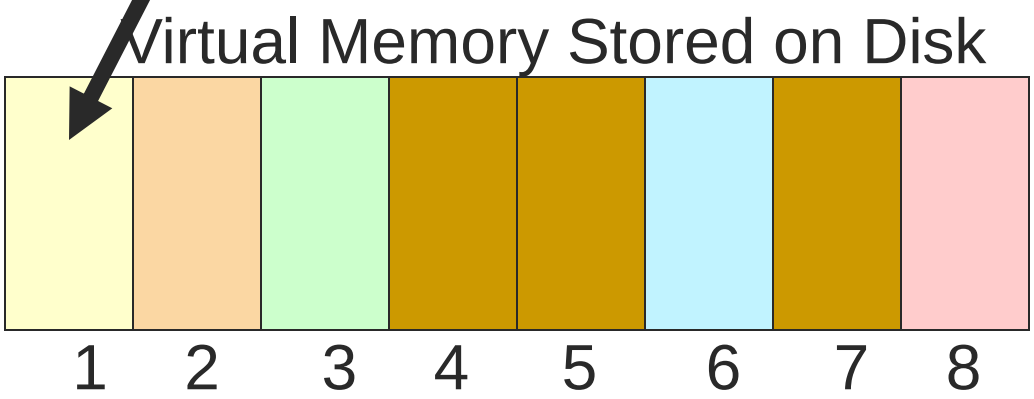
Paging Example

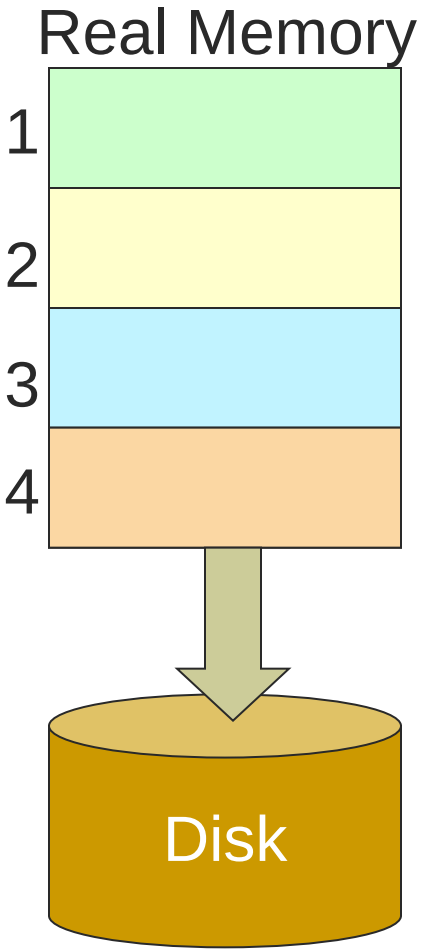
Store Virtual Memory
Page 1 to disk



Page Table

VM	Frame
3	1
1	2
6	3
2	4

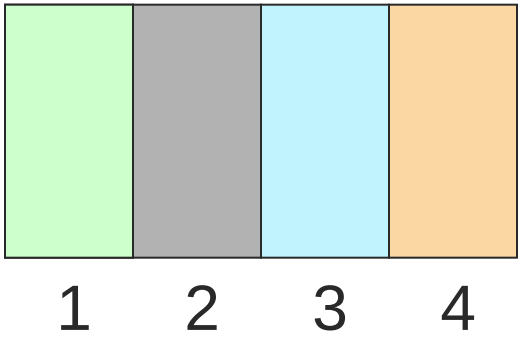




Paging Example

Process request for Address within Virtual Memory Page 8

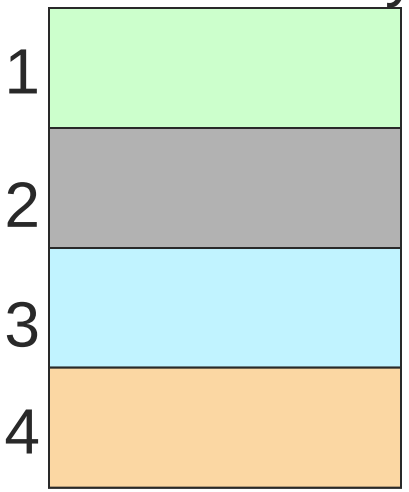
Cache



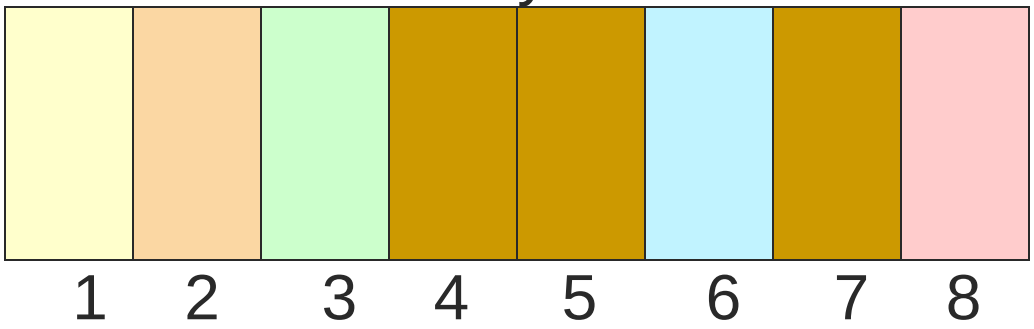
Page Table

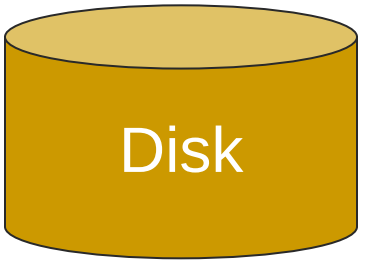
VM	Frame
3	1
	2
6	3
2	4

Real Memory



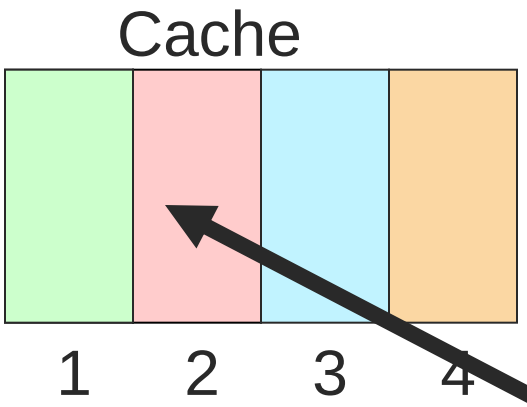
Virtual Memory Stored on Disk





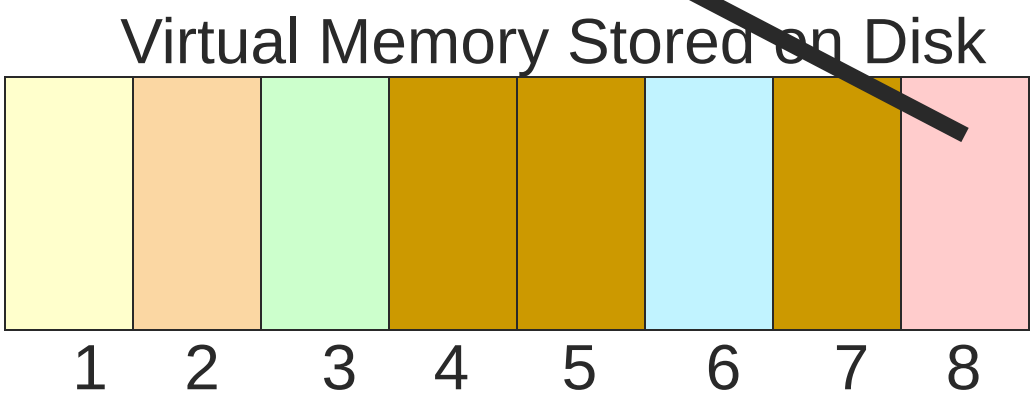
Paging Example

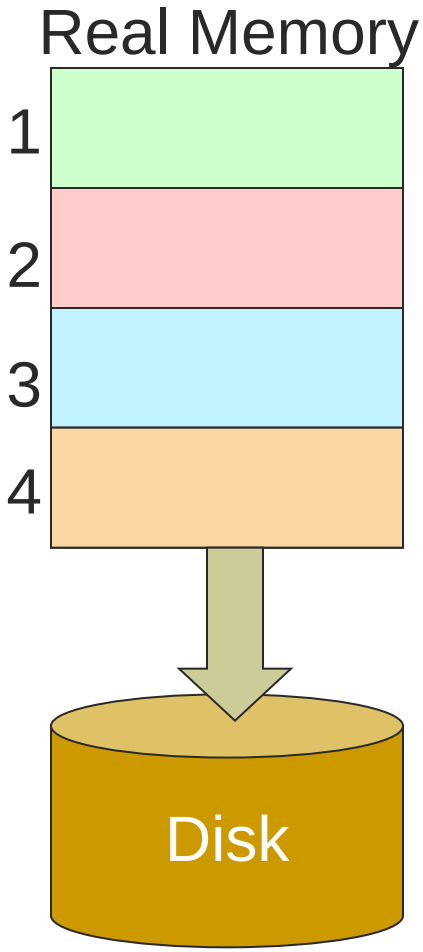
Load Virtual Memory
Page 8 to cache



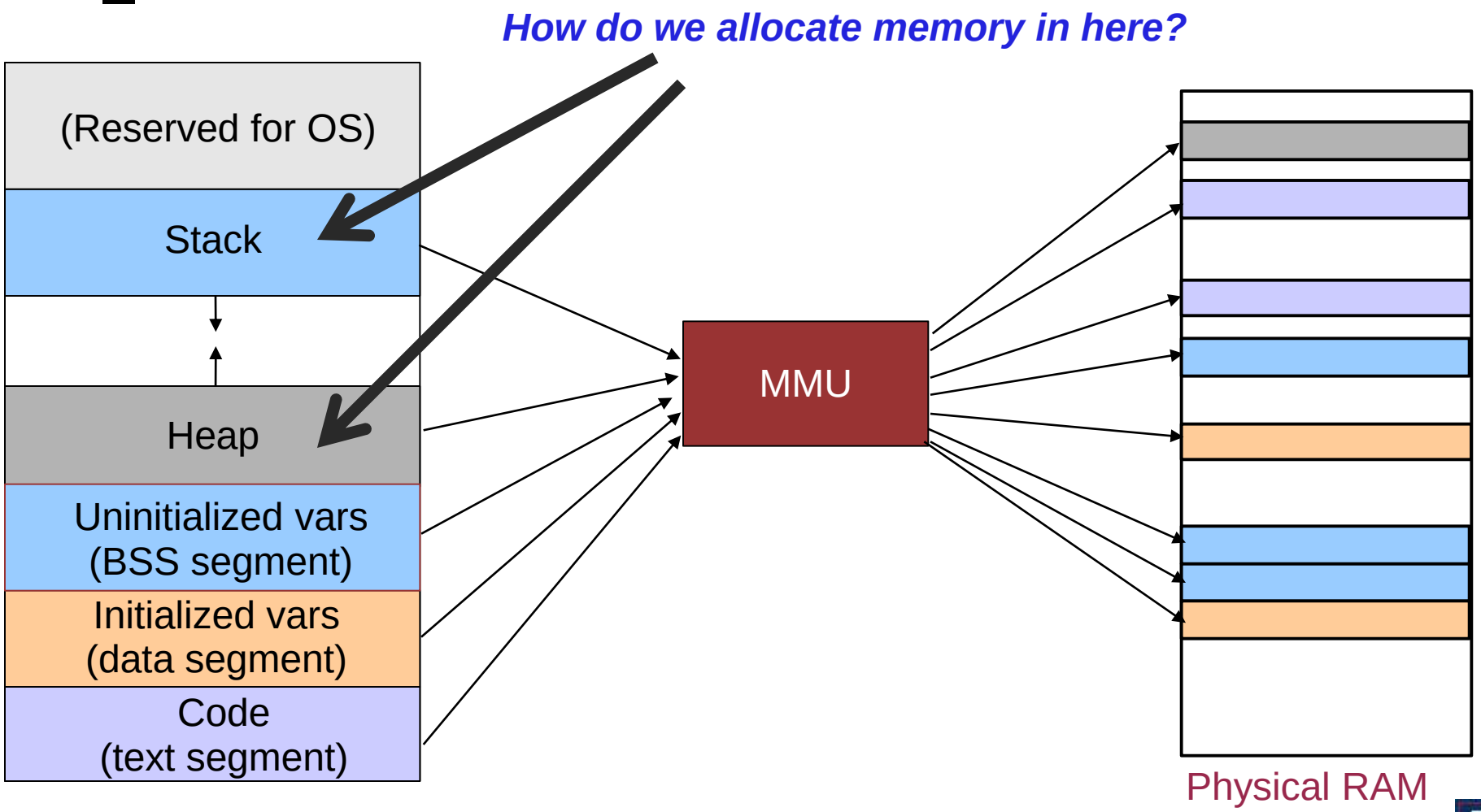
Page Table

VM	Frame
3	1
8	2
6	3
2	4





Is paging enough?



Memory allocation within 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

Memory allocation within a process

- Solution (stack): stack data structure
  - 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

Problems

- What was the key abstraction not supported well by segmentation and by B&B?
  - Supporting an address space larger than the size of physical memory
- How could you support this using B&B and segmentation?
  - Use lots of segments and have the user switch between them (this is kind of how x86 segmentation works)
- Note: x86 used to support segmentation, now effectively deprecated with x86-64

Paging

- On heavily-loaded systems, memory can fill up
- Need to make room for newly-accessed pages
  - Heuristic: try to move “inactive” pages out to disk
    - What constitutes an “inactive” page?
- Paging
  - Refers to moving individual pages out to disk (and back)
  - We often use the terms “paging” and “swapping” interchangeably
  - Different from context switching
    - Background processes often have their pages remain resident in memory

Demand Paging

- Never bring a page into primary memory until its needed
- Fetch Strategies
  - When should a page be brought into primary (main) memory from secondary (disk) storage.
- Placement Strategies
  - When a page is brought into primary storage, where should it be put?
- Replacement Strategies
  - Which page now in primary storage should be removed from primary storage when some other page or segment needs to be brought in and there is not enough room

Page Eviction

- When do we decide to evict a page from memory?
  - Usually, at the same time that we are trying to allocate a new physical page
  - However, the OS keeps a pool of “free pages” around, even when memory is tight, so that allocating a new page can be done quickly
  - The process of evicting pages to disk is then performed in the background

Page Eviction: Which page?

- Hopefully, kick out a less-useful page
  - Dirty pages require writing, clean pages don't
  - Where do you write? To "swap space"
- Goal: kick out the page that's least useful
- Problem: how do you determine utility?
  - Heuristic: temporal locality exists
  - Kick out pages that aren't likely to be used again

Basic Page Replacement

- How do we replace pages?
  - Find the location of the desired page on disk
  - Find a free frame
    - If there is a free frame, use it
    - If there is no free frame, use a page replacement algorithm to select a *victim* frame
  - Read the desired page into the (newly) free frame. Update the page and frame tables.
  - Restart the process

Page Replacement Strategies

- Random page replacement
  - Choose a page randomly
- FIFO - First in First Out
  - Replace the page that has been in primary memory the longest
- LRU - Least Recently Used
  - Replace the page that has not been used for the longest time
- LFU - Least Frequently Used
  - Replace the page that is used least often
- NRU - Not Recently Used
  - An approximation to LRU.
- Working Set
  - Keep in memory those pages that the process is actively using.