Virtual Memory

Address Space Abstraction

- Program address space
  - All memory data
  - i.e., program code, stack, data segment

- Hardware interface (physical reality)
  - Computer has one small, shared memory
- Application interface (illusion)
  - Each process wants private, large memory

How can we close this gap?

Address Space Illusions

- Address independence
  - Same address can be used in different address spaces yet remain logically distinct
- Protection
  - One address space cannot access data in another address space
- Virtual memory
  - Address space can be larger than the amount of physical memory on the machine

Address Space Illusions: Virtual Memory

Illusion	**Reality**
Giant (virtual) address space	Many processes sharing one (physical) address space
Protected from others	Limited memory
(Unless you want to share it)	
More whenever you want it	

Today:
An introduction to Virtual Memory:
The memory space seen by your programs!

Multi-Programming

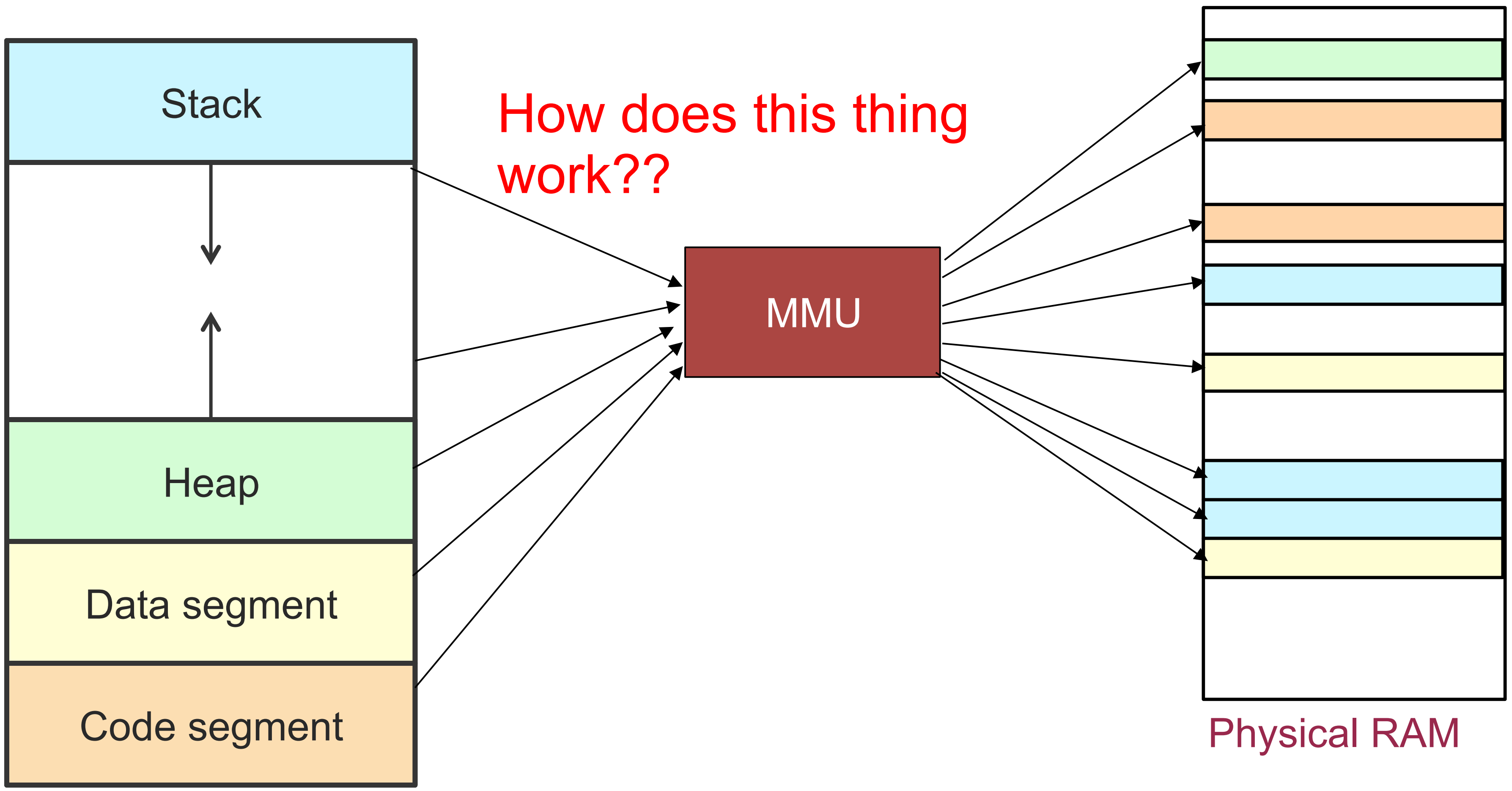
- Multiple processes in memory at the same time
- Goals
  1. Layout processes in memory as needed
  2. Protect each process's memory from accesses by other processes
  3. Minimize performance overheads
  4. Maximize memory utilization

OS & memory management

- Main memory is normally divided into two parts: kernel memory and user memory
- In a multiprogramming system, the "user" part of main memory needs to be divided to accommodate multiple processes.
- The user memory is dynamically managed by the OS (known as memory management) to satisfy following requirements:
  - Protection
  - Relocation
  - Sharing
  - Memory organization (main mem. (RAM) + secondary mem. (Hard-Disk))

➔ since available RAM memory is often not big enough to accommodate all the memory needs of user processes, some processes (or part of it) might be swapped to disk when they are not executing.

Mapping logical to physical addresses



Physical RAM

Virtual Memory today: Paging

- Paging
  - Suppose to partition physical main memory in small equal-size chunks (frames)
  - Suppose each process memory is also divided into small fixed-size chunks (pages) of the same size
  - **Pages of a process can be mapped to available frames in physical memory**

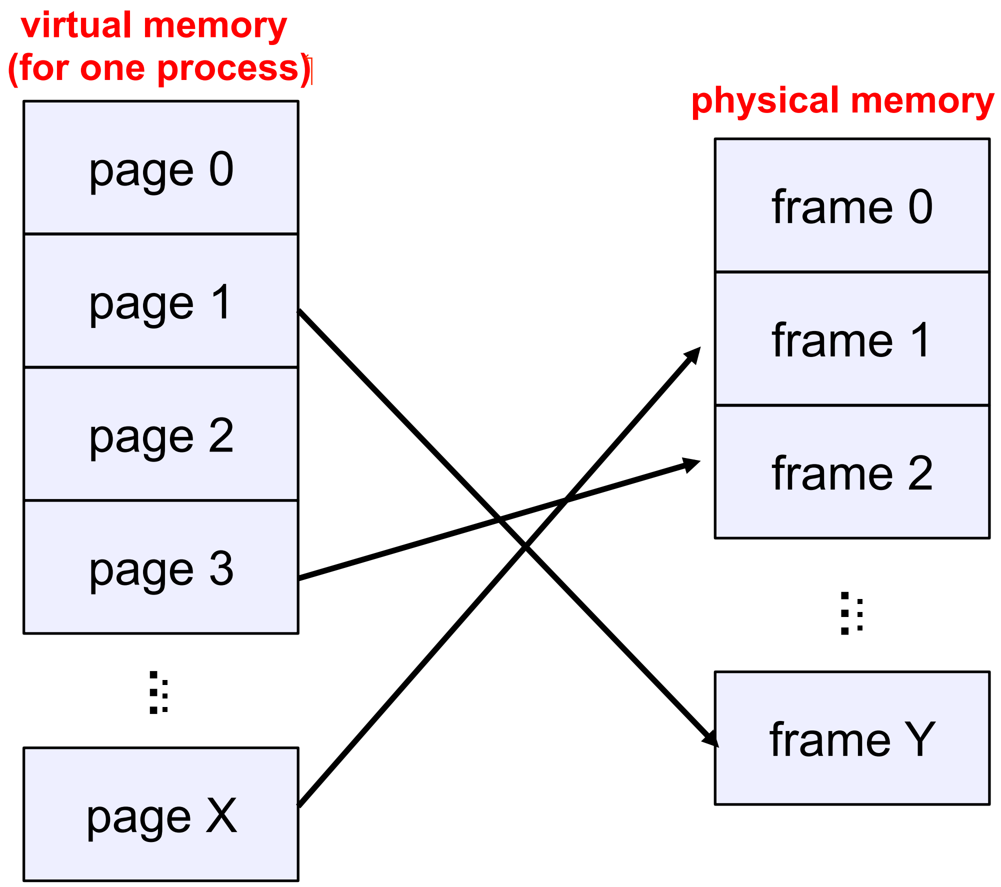
- We would like to allocate non-contiguous physical frames to a process
  - can we do that?
  - if this is the case, is the system suffering internal or external fragmentation?

Paging

- We would like to allocate non-contiguous frames to a process
- We can implement such a scheme as follows:
  - The programmer uses a contiguous logical address space (Virtual Memory divided in pages)
  - System uses a process page table to identify page<->frame mapping for each process
  - Translation from logical to physical address is performed by MMU at run-time
  - A logical address is composed of (page #, offset); the processor uses active process page table to produce a physical address (frame #, offset)

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)

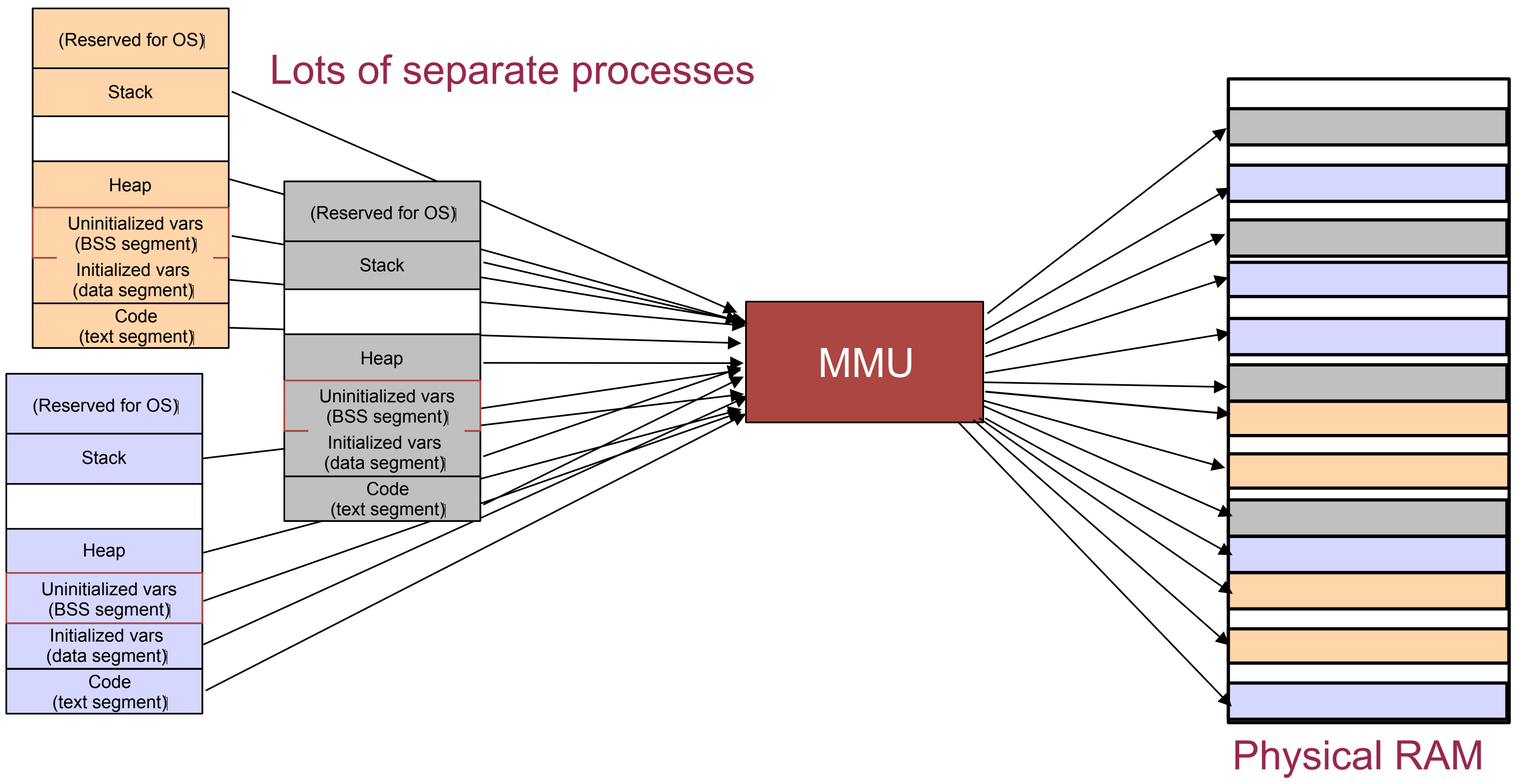


Application Perspective

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

Example of Virtual Address space:
32 bits pointers → 4 Gbyte virtual address space)

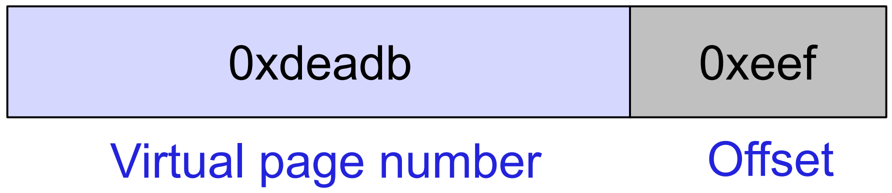
Application Perspective


(Reserved for OS)
Stack
Heap
Uninitialized vars (BSS segment)
Initialized vars (data segment)
Code (text segment)
Lots of separate processes
(Reserved for OS)
Stack
Heap
Uninitialized vars (BSS segment)
Initialized vars (data segment)
Code (text segment)
(Reserved for OS)
Stack
Heap
Uninitialized vars (BSS segment)
Initialized vars (data segment)
Code (text segment)
MMU
Physical RAM

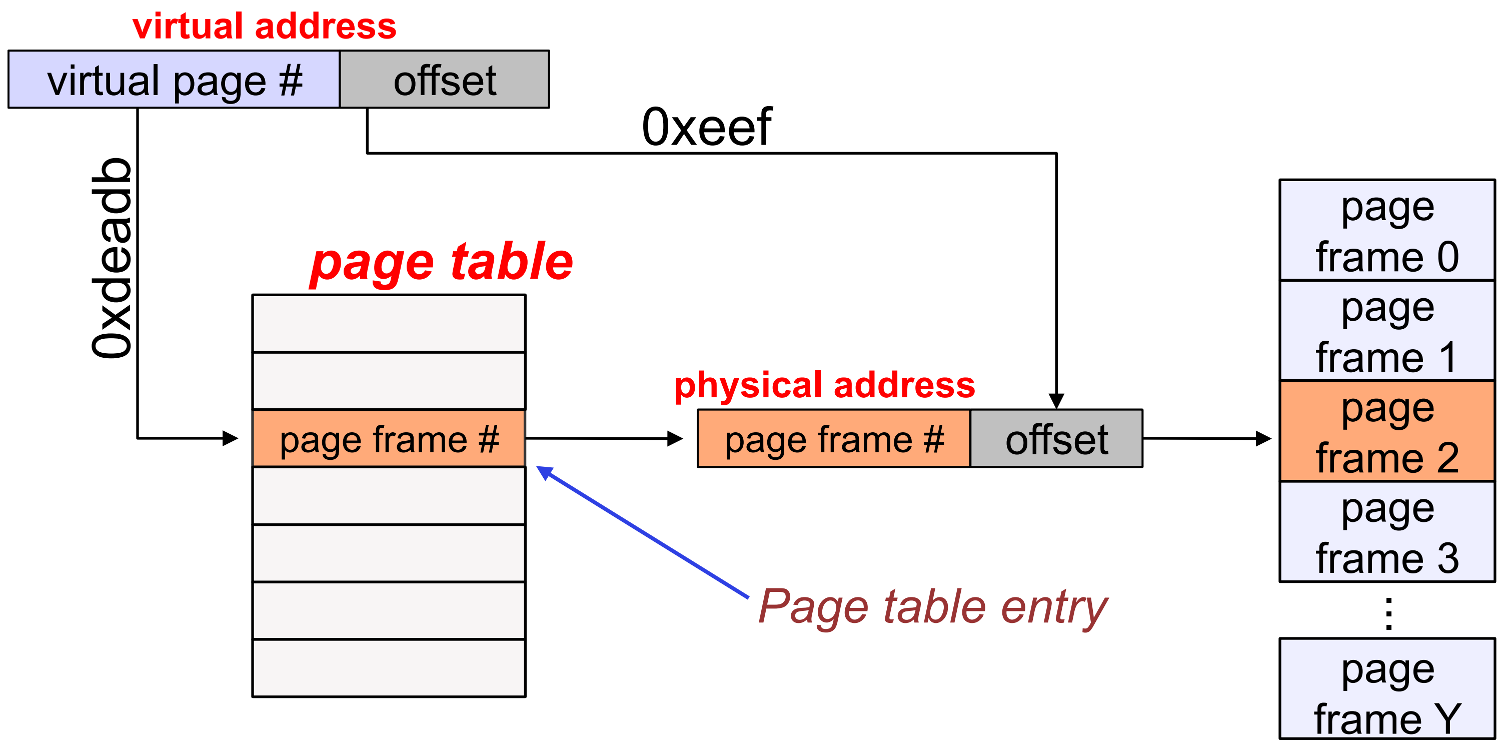
Translation process

- 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)

0xdeadbeef =



Translation process



Translation process

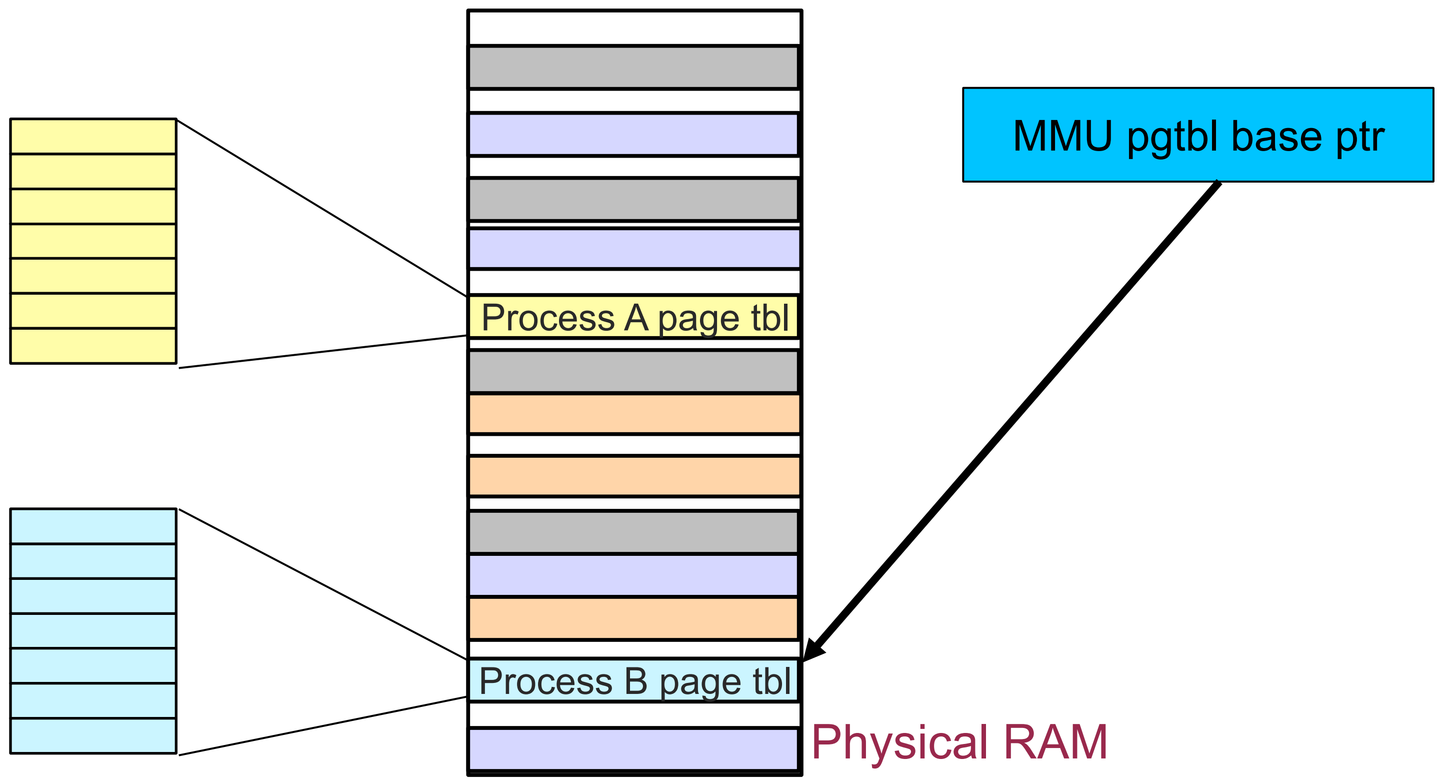
```
if (virtual page is invalid or non-present or protected)
    trap to OS fault handler
else
    physical frame # = pageTable[virtpage#].physPageNum
```

- Each virtual page can be in physical memory or swapped out to disk (called “paged out” or just “paged”), or invalid (VP was not mapped!)
- What must change on a context switch?
  - Could copy entire content of table, but this would 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

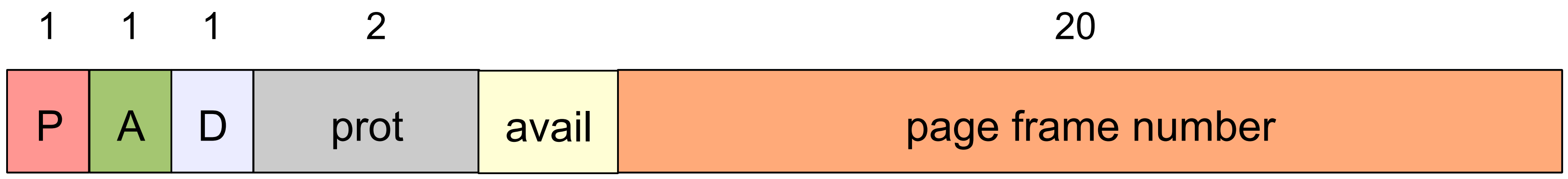
- Page Table Entries have read, write, execute protection bits to protect memory
  - Check is done by hardware during access
  - Can give shared memory location different protections from different processes by having different page table protection access bits
- How does the processor know that a virtual page is in memory?
  - Present (P) bit tells the hardware that the virtual address is present or non-present

Valid (or Mapped) vs. Present

- Present
  - Virtual page is in memory
  - NOT an error for a program trying to access non-present page
- Valid (or Mapped)
  - Virtual page has been mapped in the process virtual address space; hence, it is legal for the program to access it
  - Remember: 32 bits pointers ➔ 4 Gbyte virtual address space (not all of it will be mapped!)

Page Table Entry

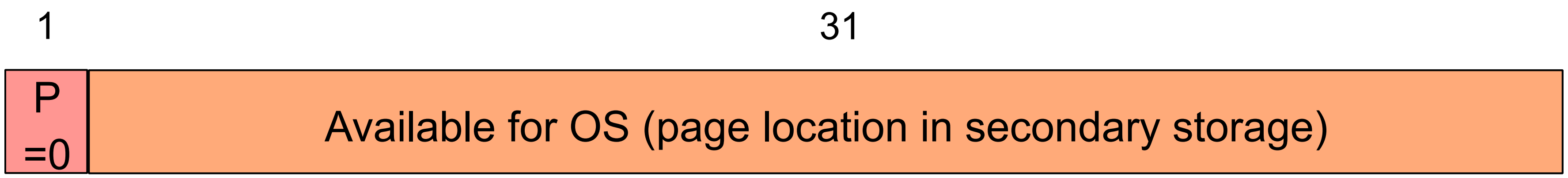
- Typical PTE format (depends on CPU architecture!)



- Various bits accessed by MMU on each page access:
  - Dirty bit: Indicates whether a page is “dirty” (modified)
  - Accessed bit: Indicates whether a page has been accessed (read or written)
  - Present bit: Page is present in physical memory (1) or not (0)
  - Protection bits: Specify if page is readable, writable, or executable
  - Page frame number: Physical location of page in RAM
  - Avail: Available for system programmers

Page Table Entry

- Typical PTE format (depends on CPU architecture!)



- Present bit = 0: Page is not present in physical memory; hence, it was swapped to disk.
- The Page Table Entry contains information about page location in secondary storage (disk).

Page Faults

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

Advantages of Paging

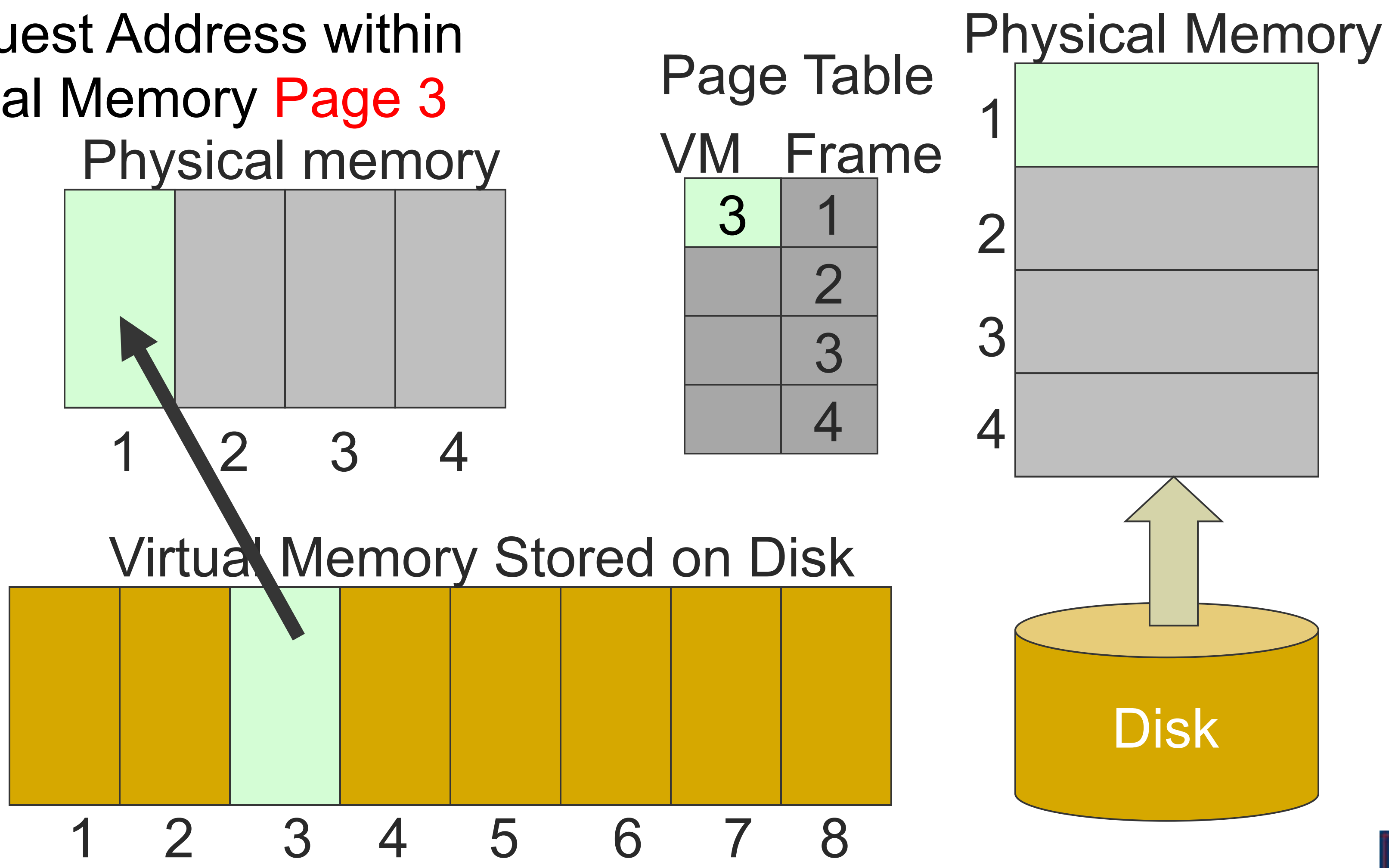
- Simplifies physical memory management
  - OS maintains a list of free physical page frames
  - To allocate a physical frame, just remove an entry from this list
- No external fragmentation!
  - Virtual pages from different processes can be interspersed in physical memory
  - No need to allocate physical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

Advantages of Paging

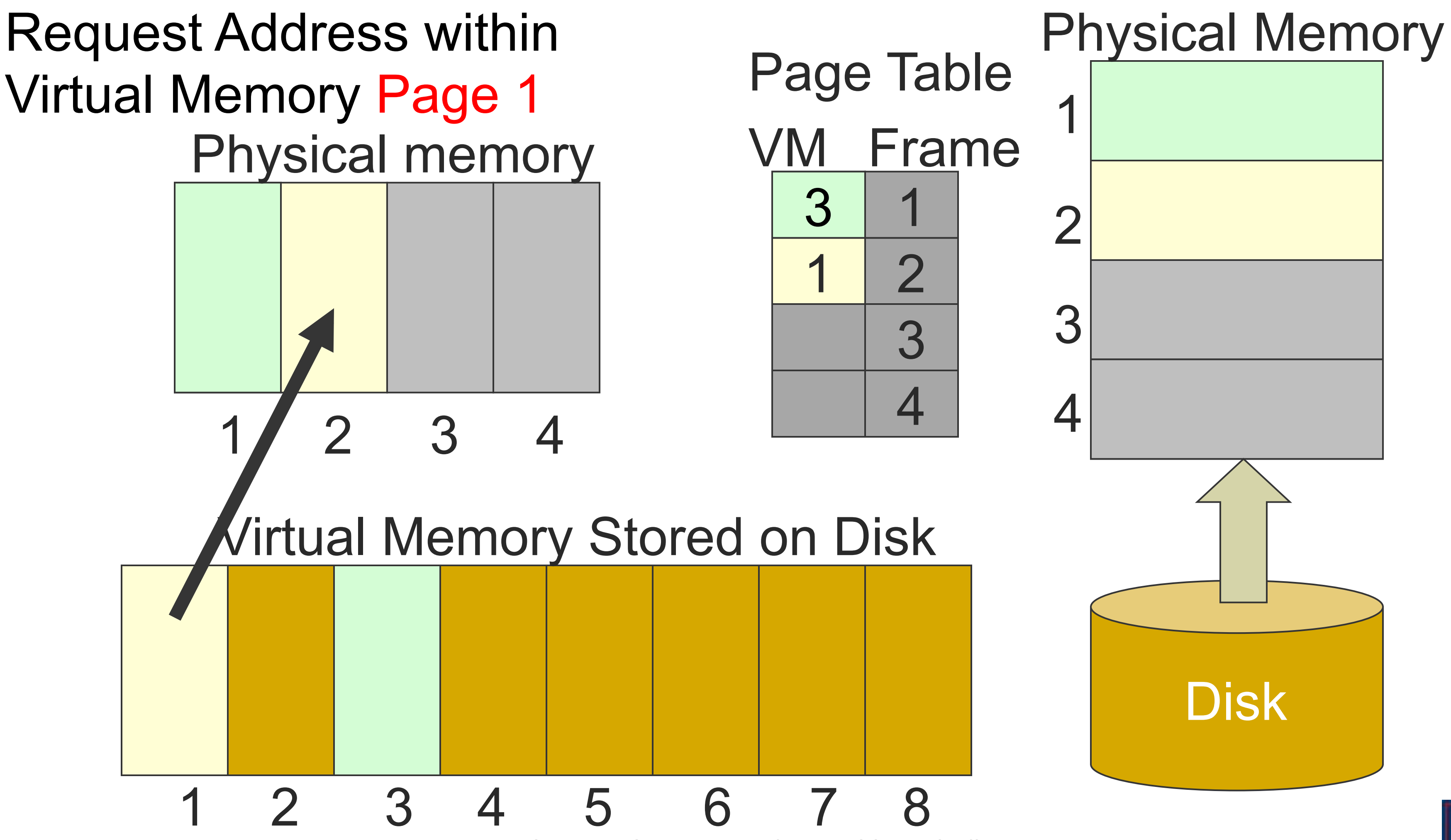
- Allocation of memory can be performed at a (relatively) fine granularity
  - Only allocate physical memory to those parts of the address space that require it
  - Can swap unused pages out to disk when physical memory is running low
  - Idle programs won't use up a lot of memory (even if their address space is huge!)

Paging Example

Request Address within Virtual Memory Page 3

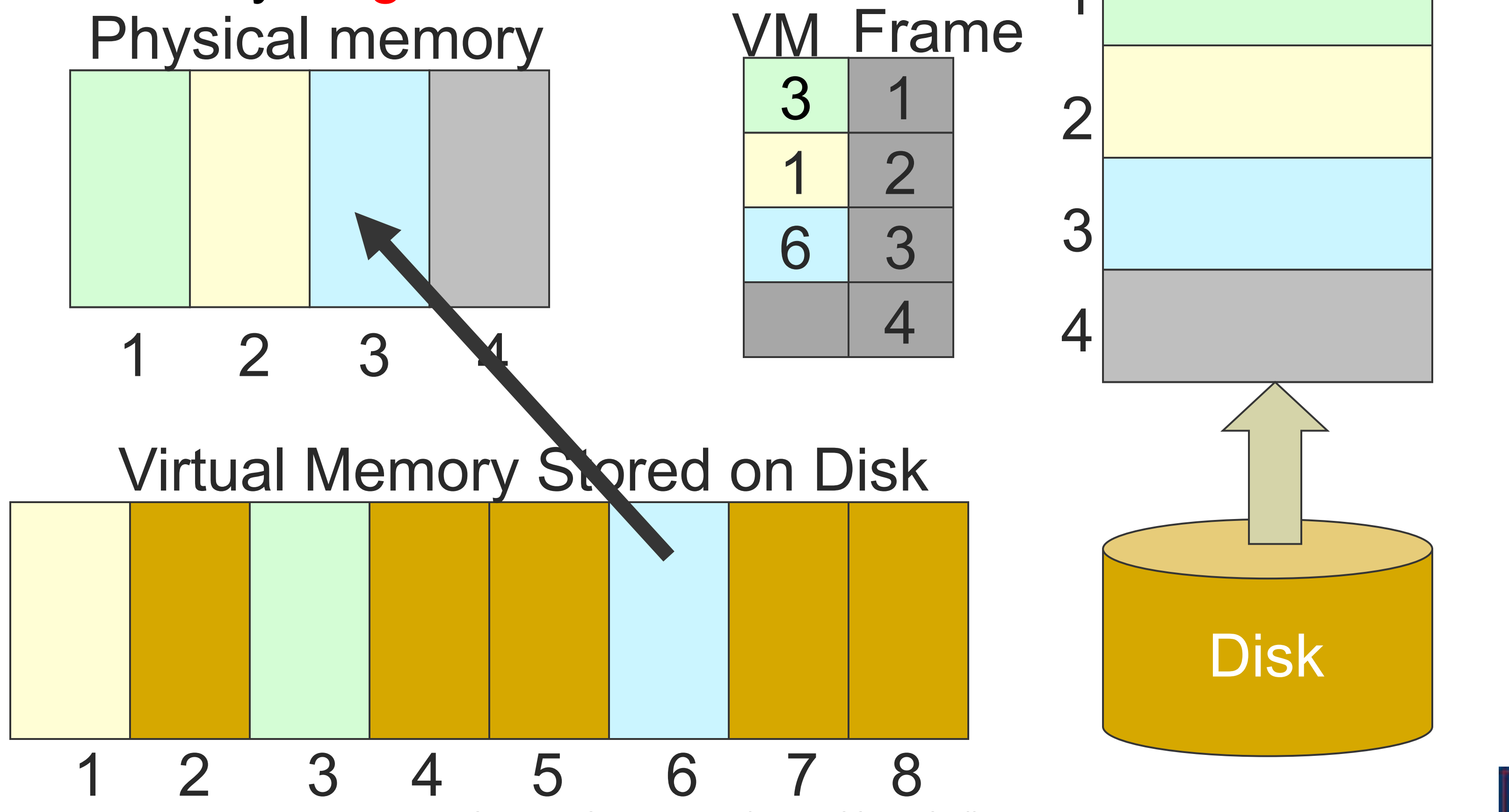


Paging Example



Paging Example

Request Address within Virtual Memory Page 6

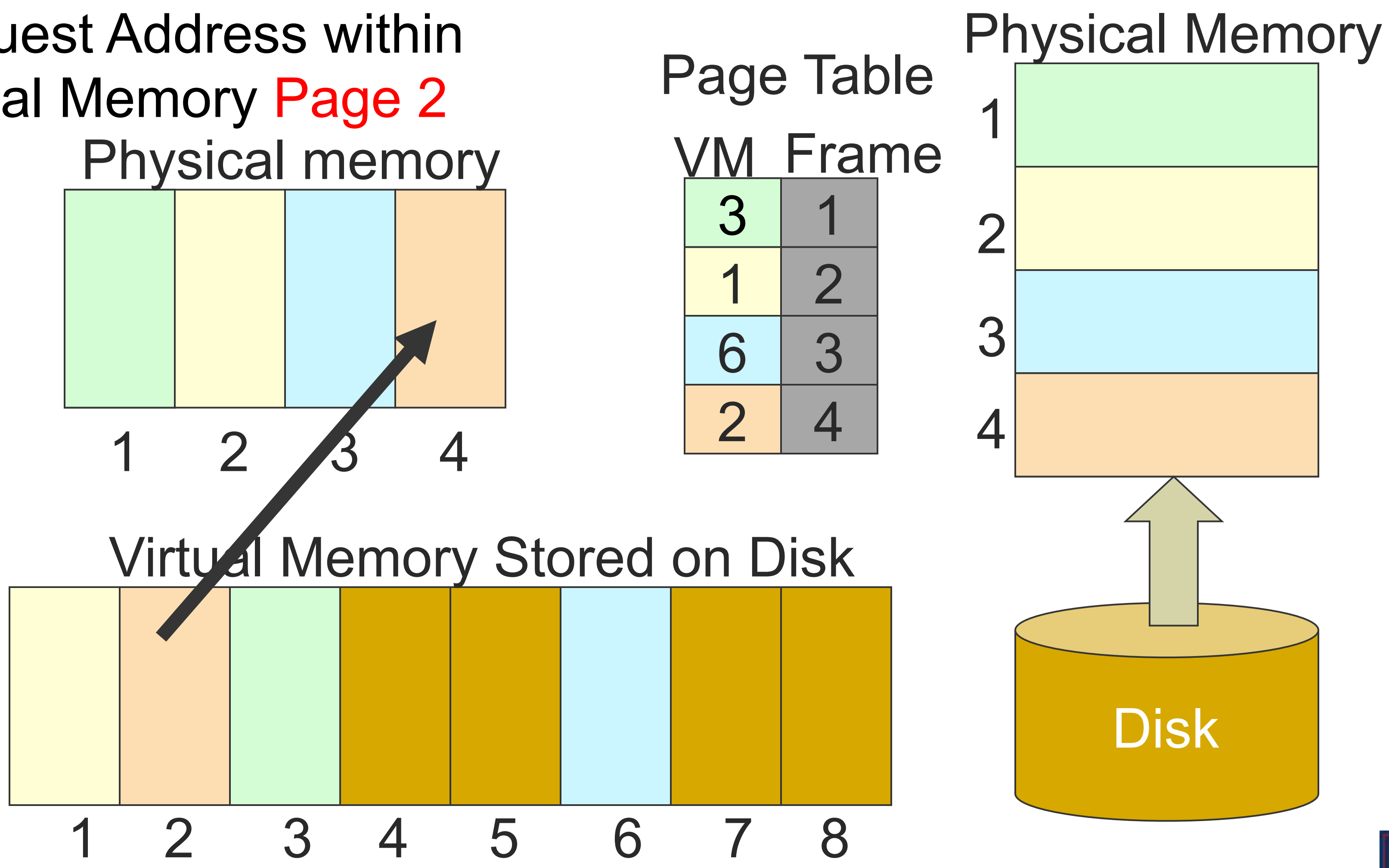


Page Table

VM	Frame
3	1
1	2
6	3
	4

Paging Example

Request Address within Virtual Memory Page 2

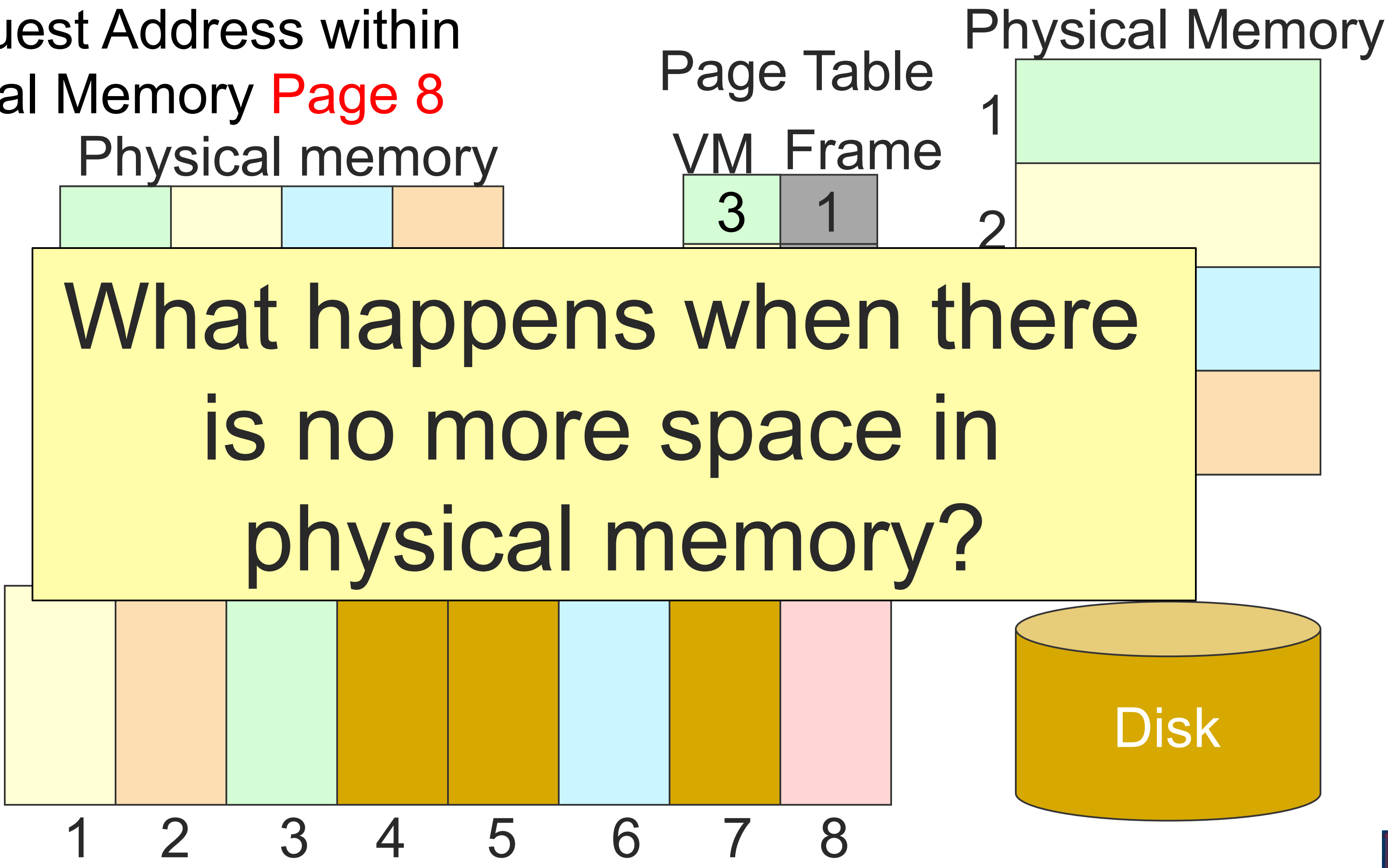


Page Table

VM	Frame
3	1
1	2
6	3
2	4

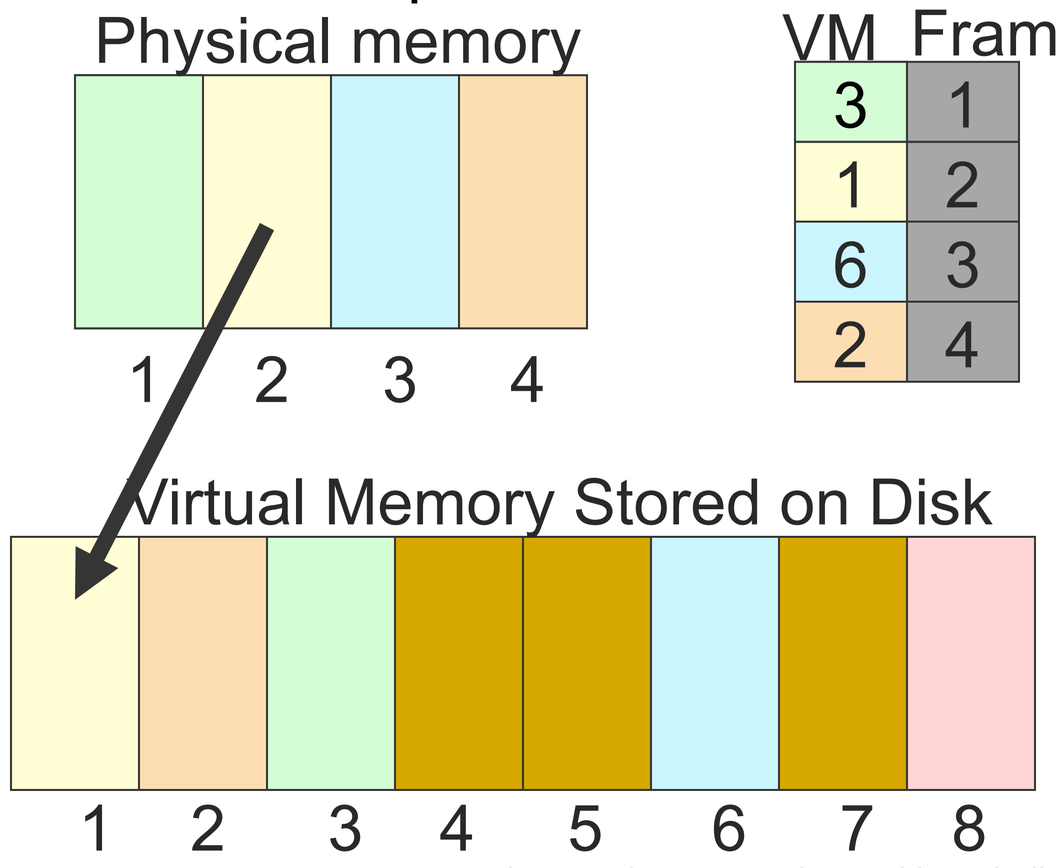
Paging Example

Request Address within Virtual Memory Page 8



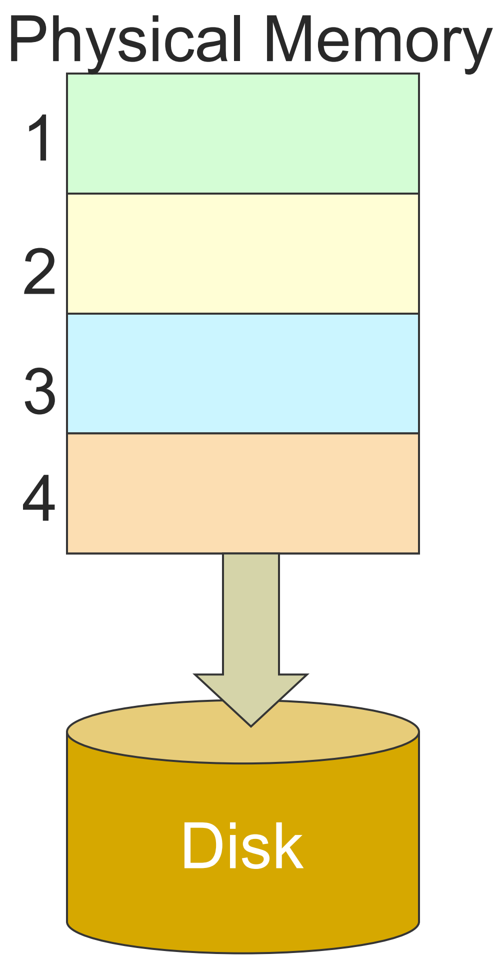
Paging Example

Store Virtual Memory Page 1 to disk swap area



Page Table

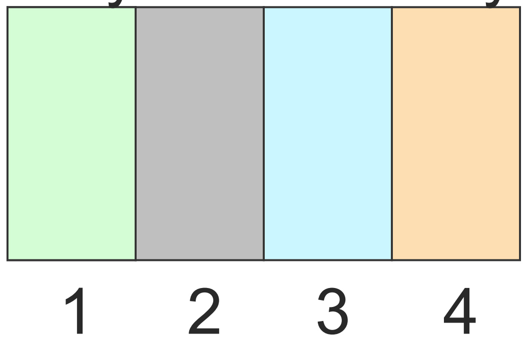
VM	Frame
3	1
1	2
6	3
2	4



Paging Example

Process request for Address within Virtual Memory Page 8

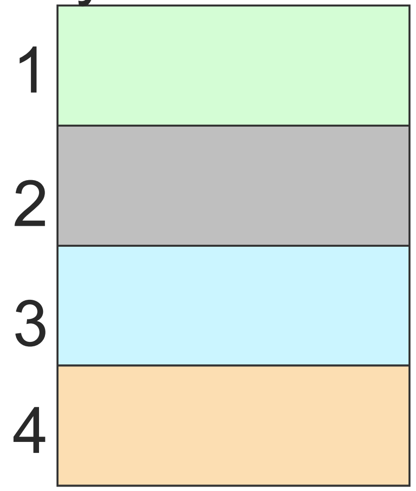
Physical memory



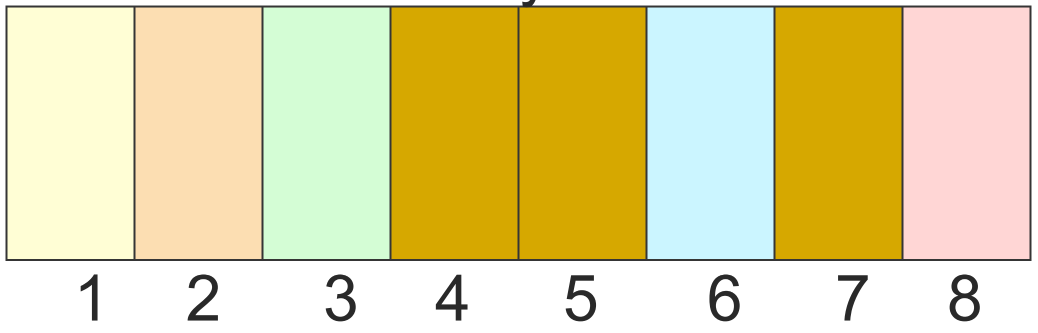
Page Table

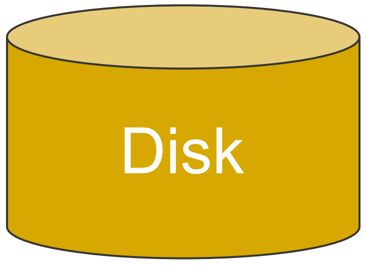
VM	Frame
3	1
	2
6	3
2	4

Physical Memory



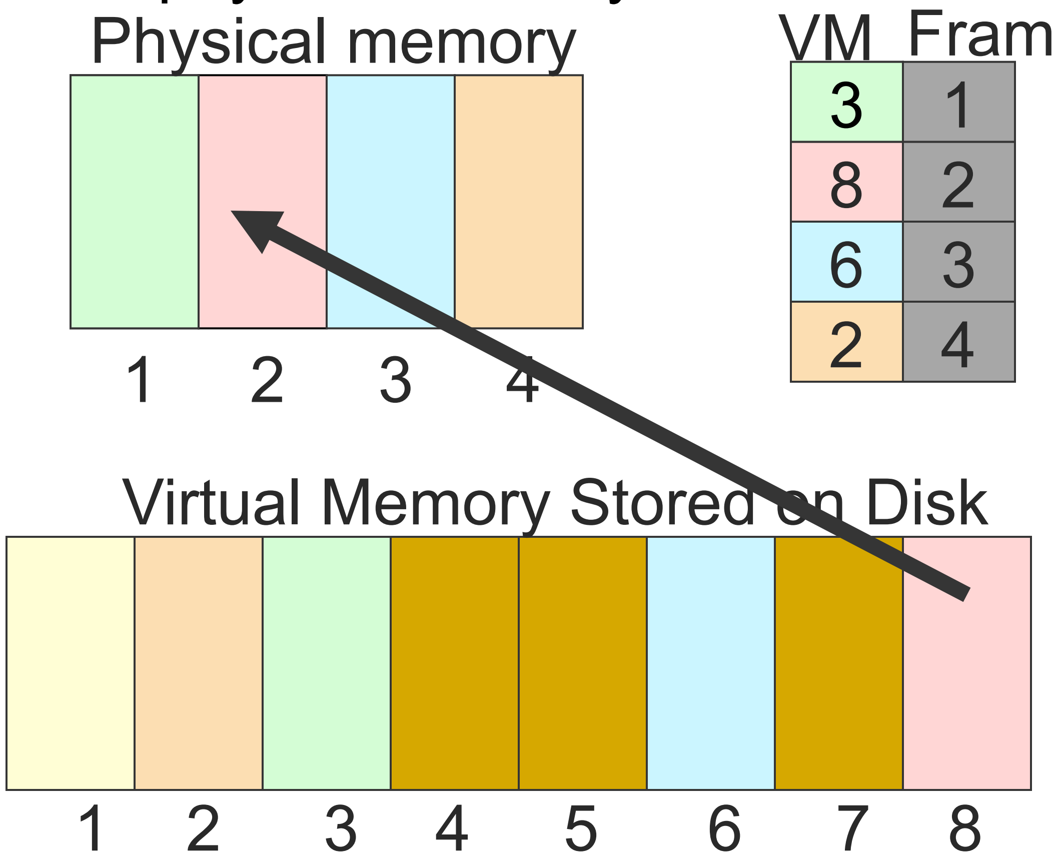
Virtual Memory Stored on Disk





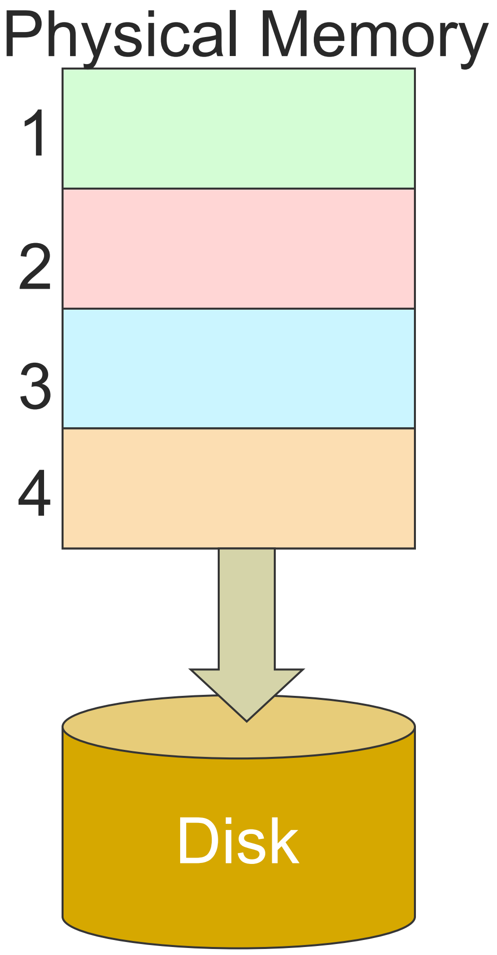
Paging Example

Load Virtual Memory Page 8 to physical memory



Page Table

VM	Frame
3	1
8	2
6	3
2	4



Page Eviction: When?

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

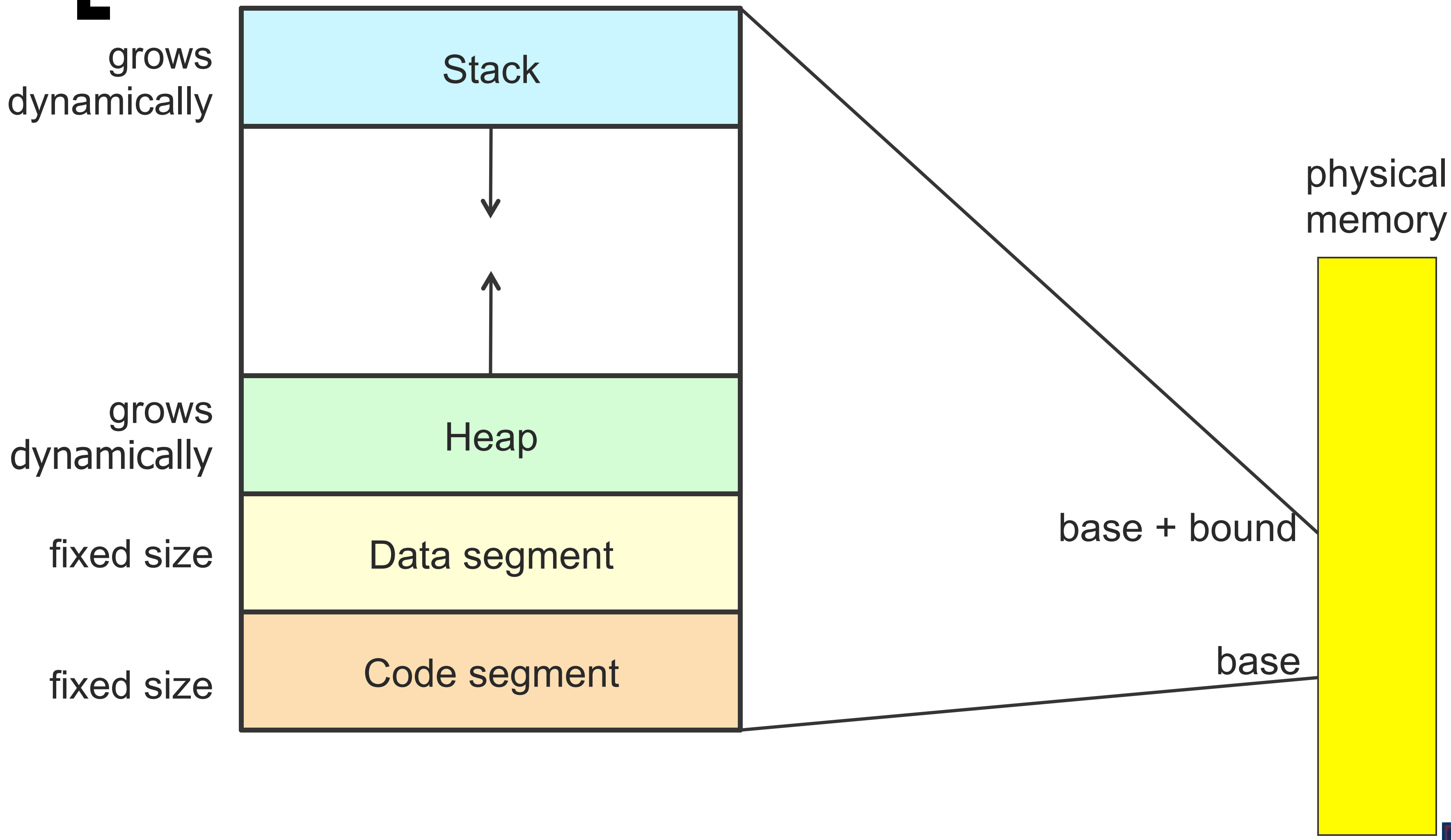
Basic Page Replacement

- 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 policy to select a *victim* page
  - Copy the desired page into the (newly) free frame. Update the page and frame tables.
  - Re-execute the instruction that caused the page fault

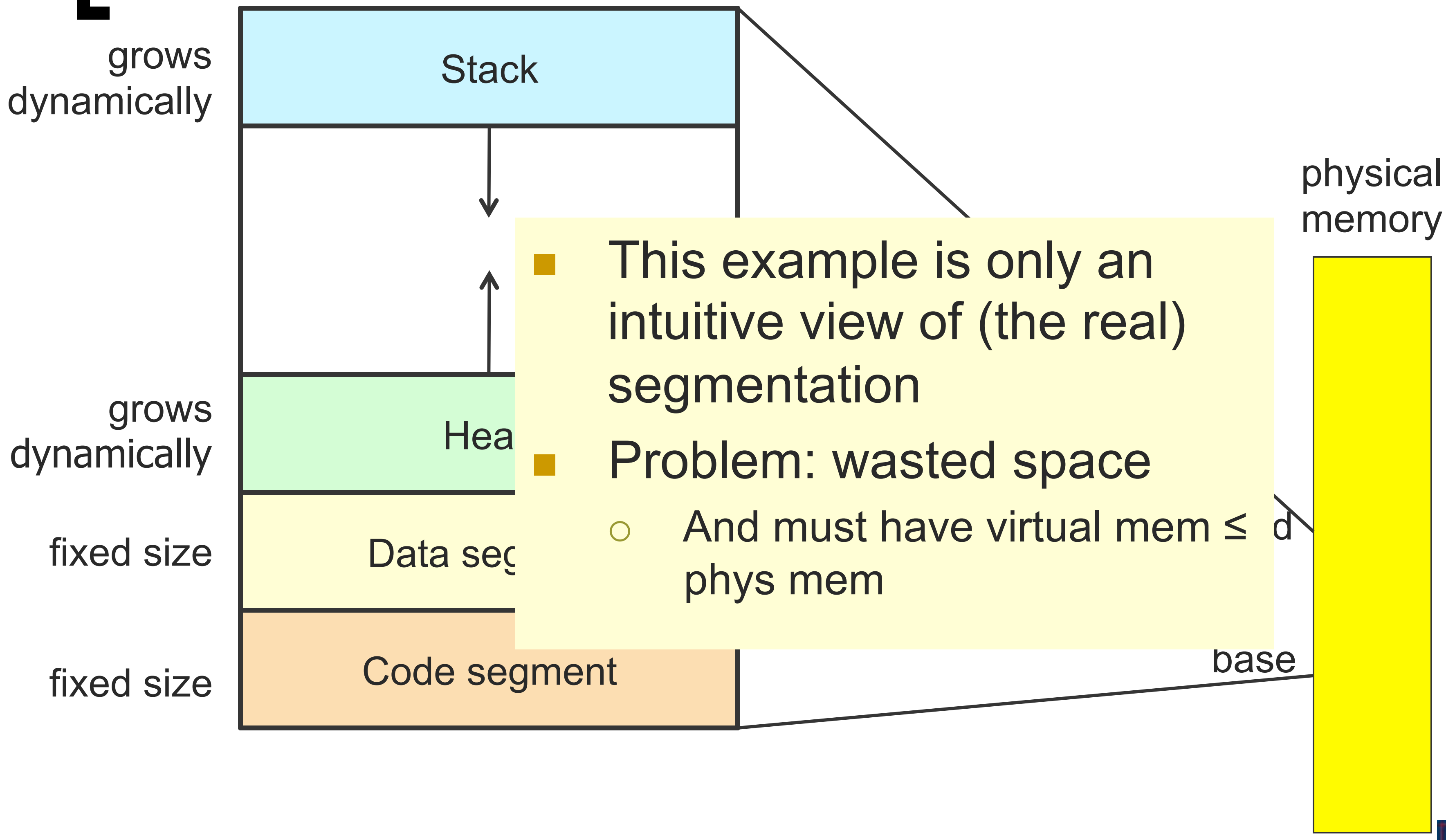
Page Replacement Strategies

- 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.

Segmentation was once an option: deprecated with x86-64



Segmentation was once an option: deprecated with x86-64



- This example is only an intuitive view of (the real) segmentation
- Problem: wasted space
  - And must have virtual mem ≤ phys mem