Memory Management and Virtual Memory

Some of the slides are adapted from Matt Welsh's.

Some slides are from Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639

Some slides are from Silberschatz, and Gagne.

Memory Management

- **Today we start a series of lectures on memory management**
- **Goals of memory management**
  - Convenient abstraction for programming
  - Provide isolation between different processes
  - Allocate scarce physical memory resources across processes
    - Especially important when memory is heavily contended for
  - Minimize overheads
- **Mechanisms**
  - Virtual address translation
  - Paging and TLBs
  - Page table management
- **Policies**
  - Page replacement policies

Virtual Memory

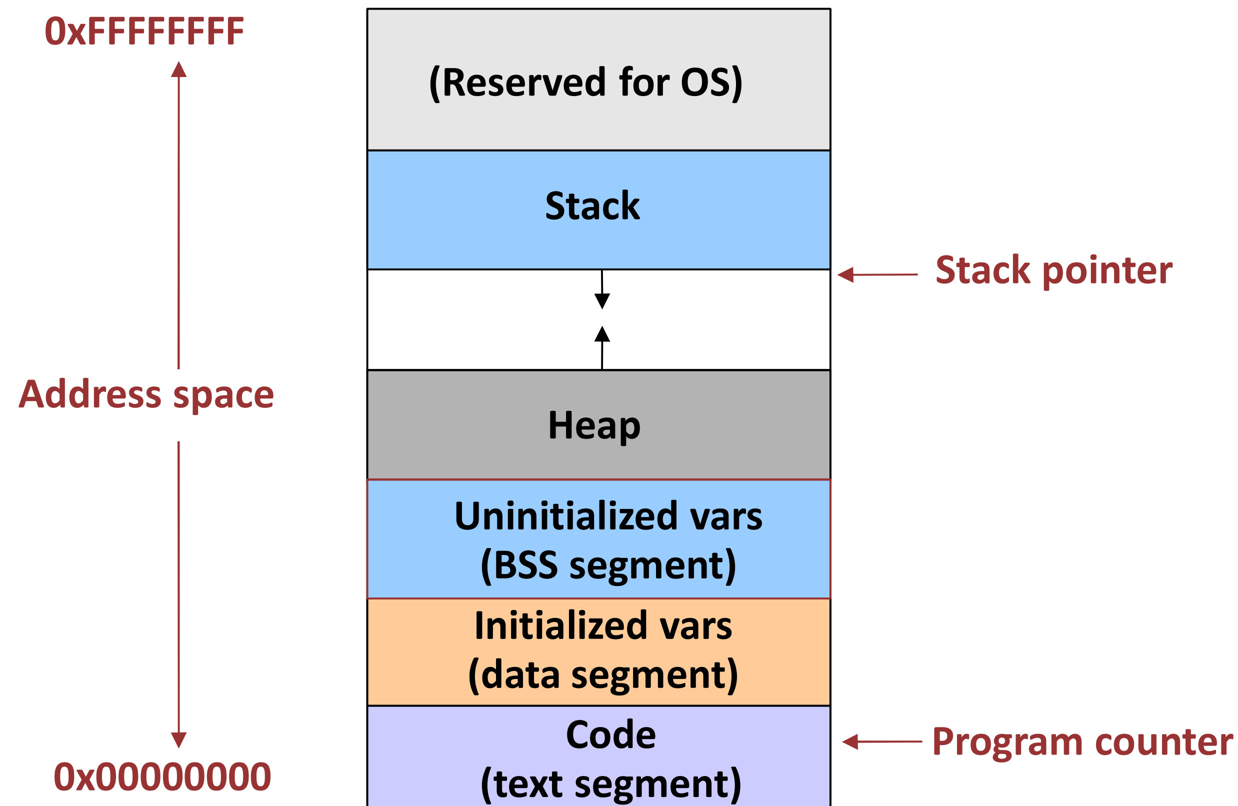
- **The basic abstraction provided by the OS for memory management**
- **VM enables programs to execute without requiring their entire address space to be resident in physical memory**
  - Program can run on machines with less physical RAM than it "needs"
- **Many programs don't use all of their code or data**
  - e.g., branches they never take, or variables never accessed
  - Observation: No need to allocate memory for it until it's used
  - OS should adjust amount allocated based on its run-time behavior
- **Virtual memory also *isolates* processes from each other**
  - One process cannot access memory addresses in others
  - Each process has its own isolated address space
- **VM requires both hardware and OS support**
  - Hardware support: memory management unit (MMU) and translation lookaside buffer (TLB)
  - OS support: virtual memory system to control the MMU and TLB

Memory Management Requirements

- **Protection**
  - Restrict which addresses processes can use, so they can't stomp on each other
- **Fast translation**
  - Accessing memory must be fast, regardless of the protection scheme
  - It would be a bad idea to have to call into the OS for every memory access.
- **Fast context switching**
  - Overhead of updating memory hardware on a context switch must be low
  - For example, it would be a bad idea to copy all of a process's memory out to disk on every context switch.

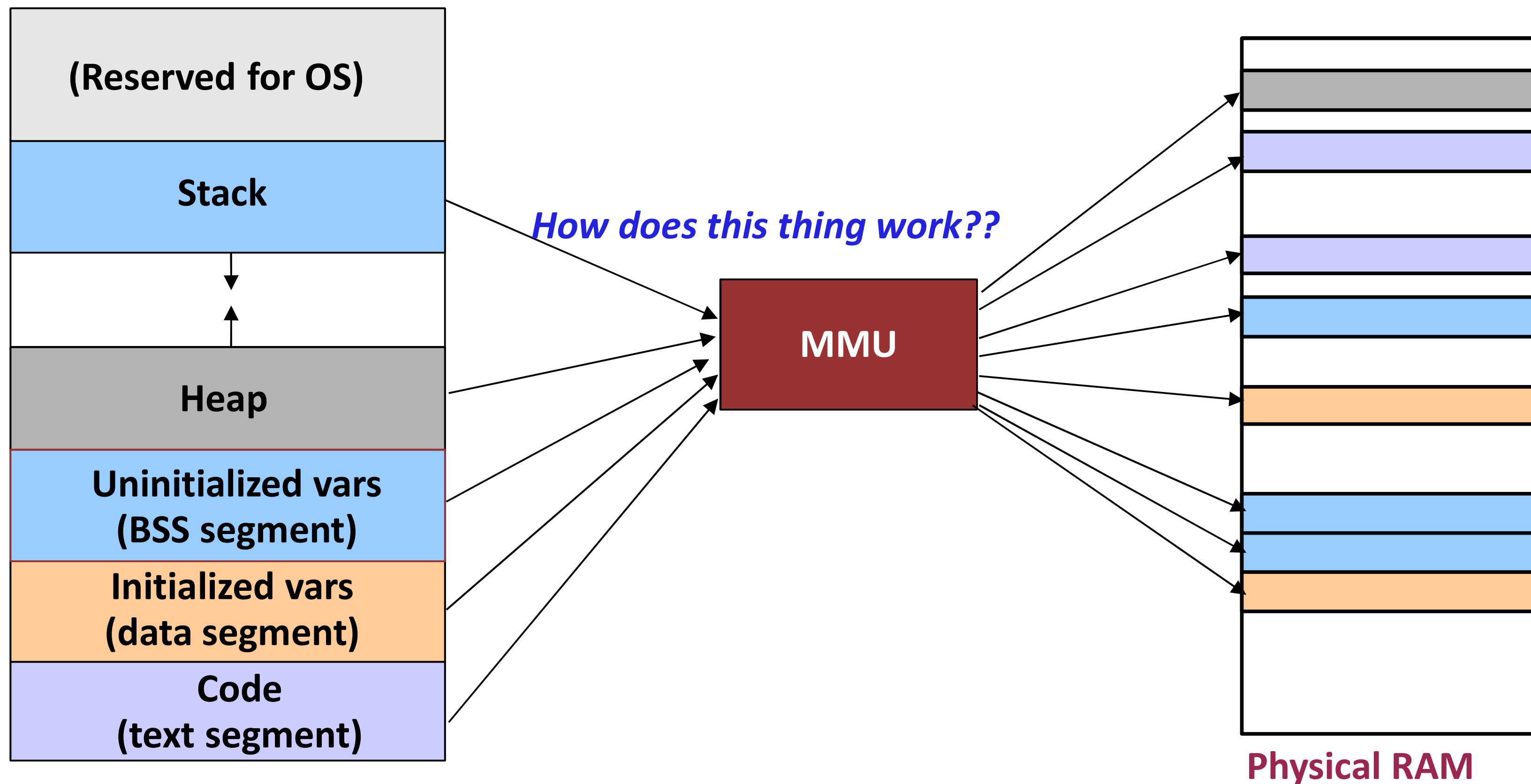
Virtual Addresses

- A ***virtual address*** **is a memory address that a process uses to access its own memory**
  - The virtual address is *not the same* as the actual physical RAM address in which it is stored
  - 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



Virtual Addresses

- A *virtual address* is a memory address that a process uses to access its own memory
  - The virtual address is *not the same* as the actual physical RAM address in which it is stored
  - 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

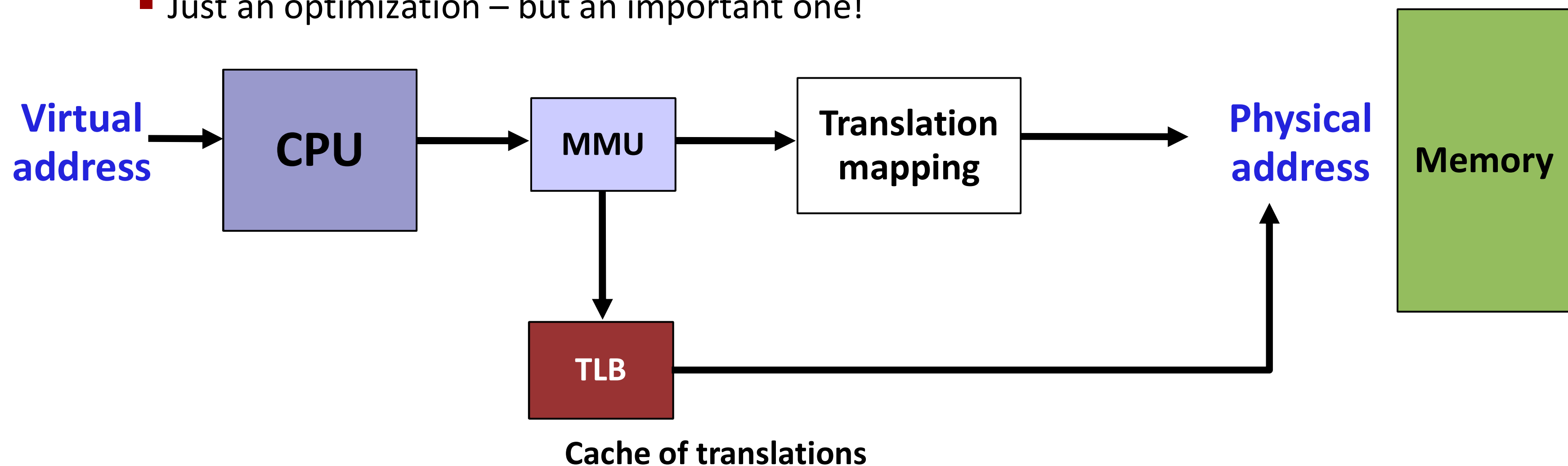


Virtual Addresses

- **A *virtual address* is a memory address that a process uses to access its own memory**
  - The virtual address is *not the same* as the actual physical RAM address in which it is stored
  - 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
- **Virtual addresses allow *isolation***
  - *Virtual addresses in one process refer to **different** physical memory than virtual addresses in another*
    - Exception: shared memory regions between processes (discussed earlier)
- **Virtual addresses allow *relocation***
  - *A program does not need to know which physical addresses it will use when it's run*
  - *Compiler can generate relocatable code – code that is independent of physical location in memory*

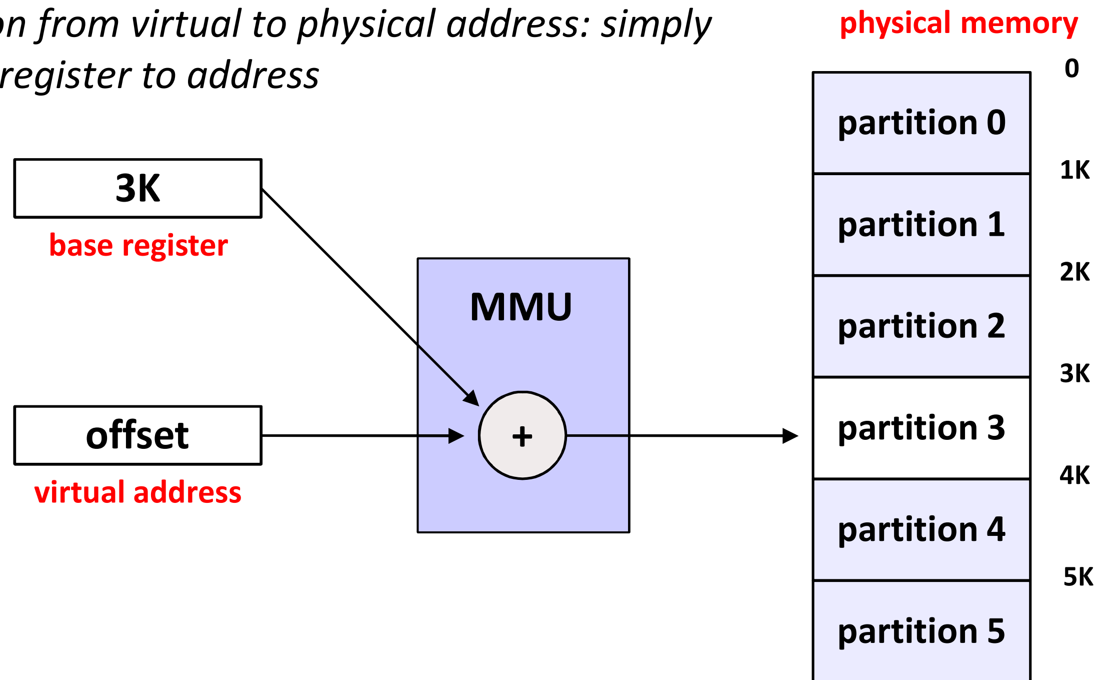
MMU and TLB

- **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!



Fixed Partitions

- **Original memory management technique: Break memory into fixed-size *partitions***
  - *Hardware requirement: base register*
  - *Translation from virtual to physical address: simply add base register to address*



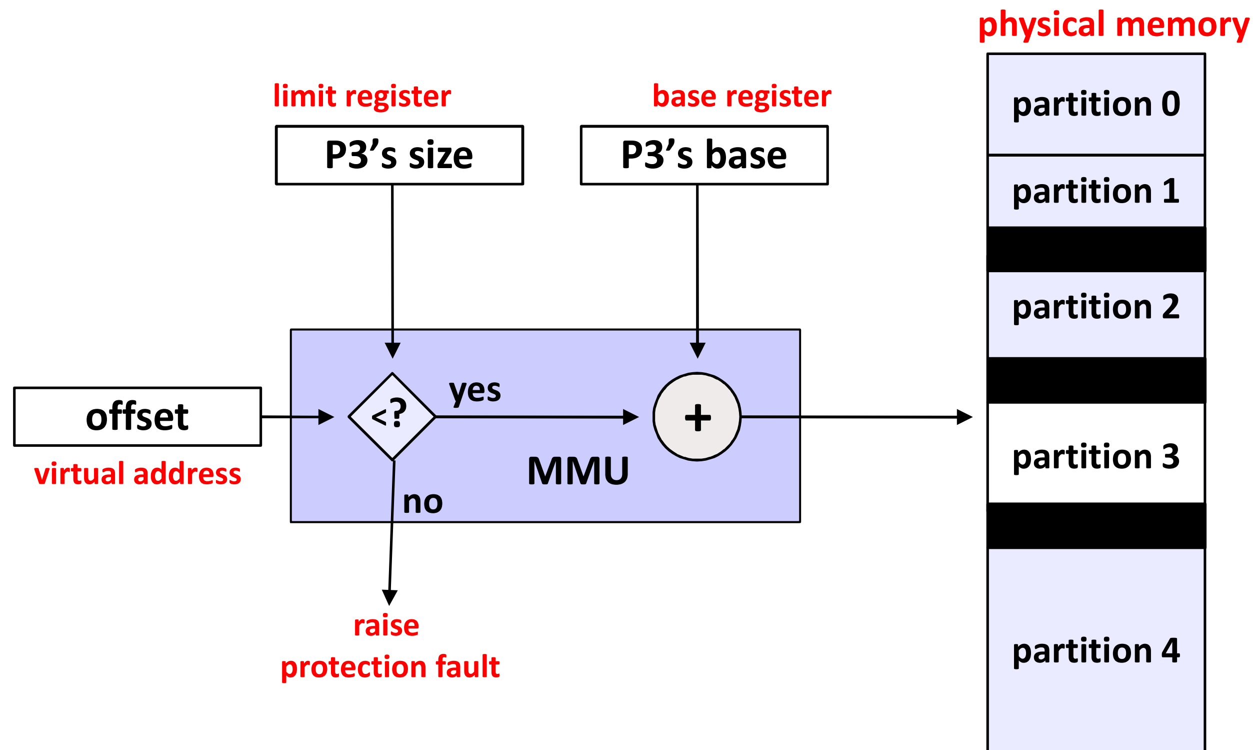
Advantages and disadvantages of this approach??

Fixed Partitions

- **Advantages:**
  - Fast context switch – only need to update base register
  - Simple memory management code: Locate empty partition when running new process
- **Disadvantages:**
  - Internal fragmentation
    - Must consume entire partition, rest of partition is "wasted"
  - Static partition sizes
    - No single size is appropriate for all programs!

Variable Partitions

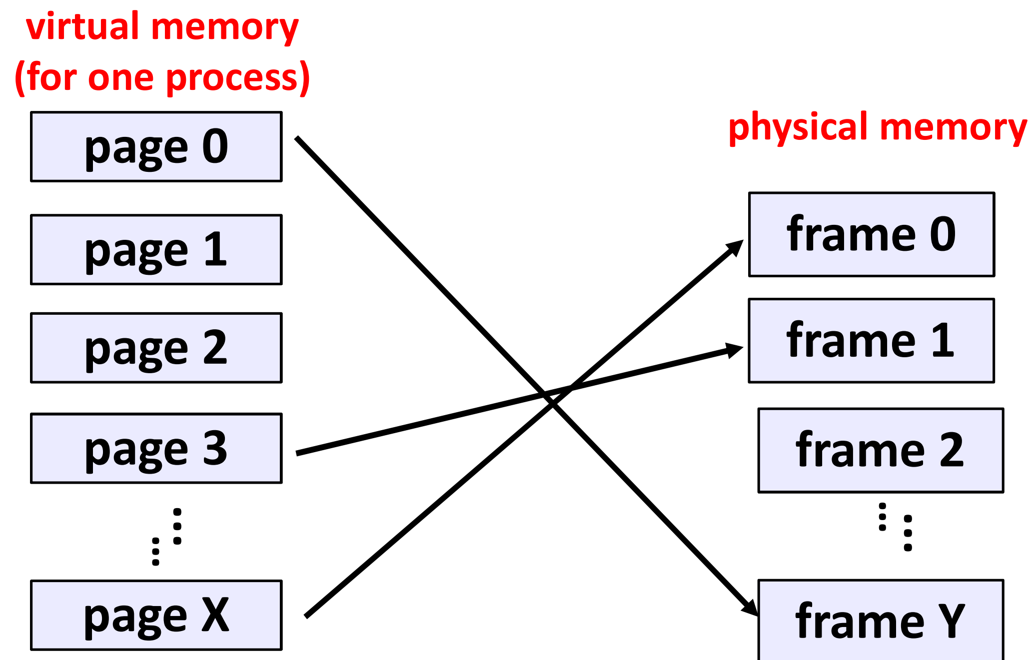
- **Obvious next step: Allow variable-sized partitions**
  - Now requires both a *base register* and a *limit register* for performing memory access
  - Solves the internal fragmentation problem: size partition based on process needs



- **New problem: *external fragmentation***
  - **As jobs run and complete, holes are left in physical memory**

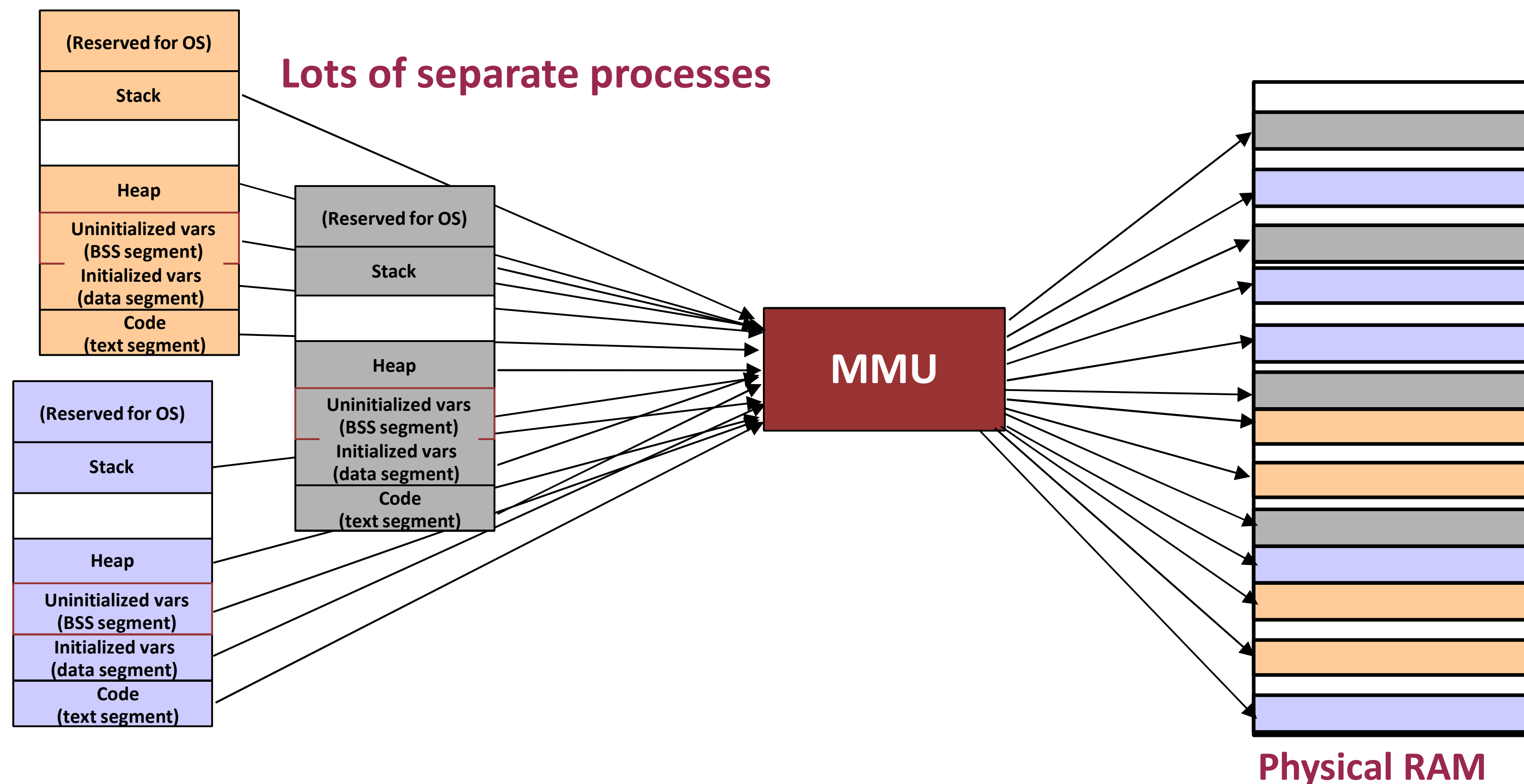
Modern technique: paging

- **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)*

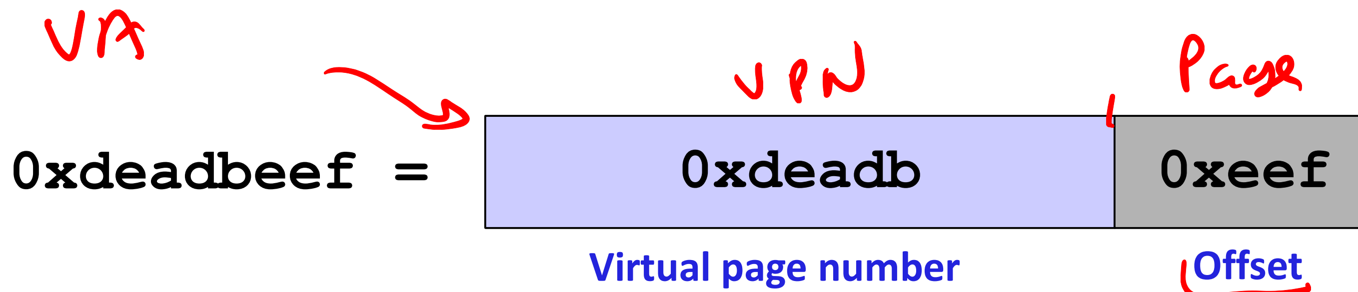


Application Perspective

- **Application believes it has a single, contiguous address space ranging from 0 to $2^P - 1$ 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!



VM with Paging



virtual address

virtual page # | offset

0xeeef

0xdeadb

CR3
PTBR

page table

physical memory

page frame #

VPN

physical address

page frame # | offset

PPN

PA

page frame 0

page frame 1

page frame 2

page frame 3

⋮

page frame Y

Page table entry

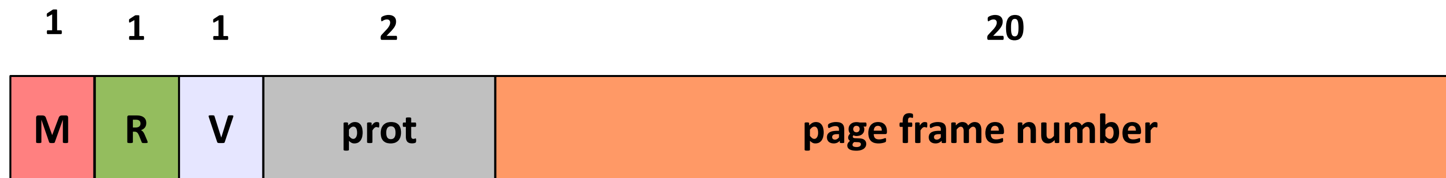
PTE

PTE → V | 1 | R | M | P | PPN

→ 0 | Disk Block #

Page Table Entries (PTEs)

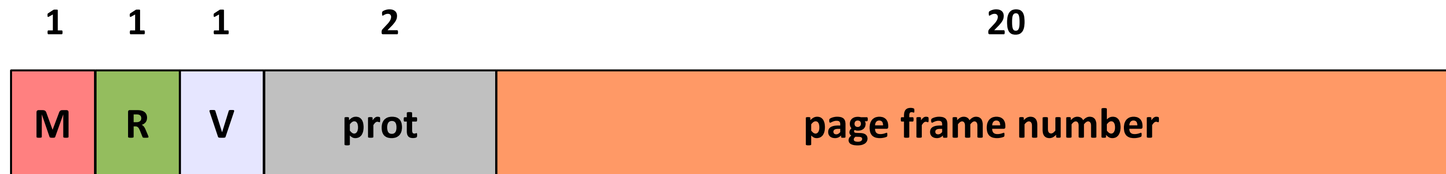
- **Typical PTE format (depends on CPU architecture!)**



- **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 Table Entries (PTEs)

- **What are these bits useful for?**



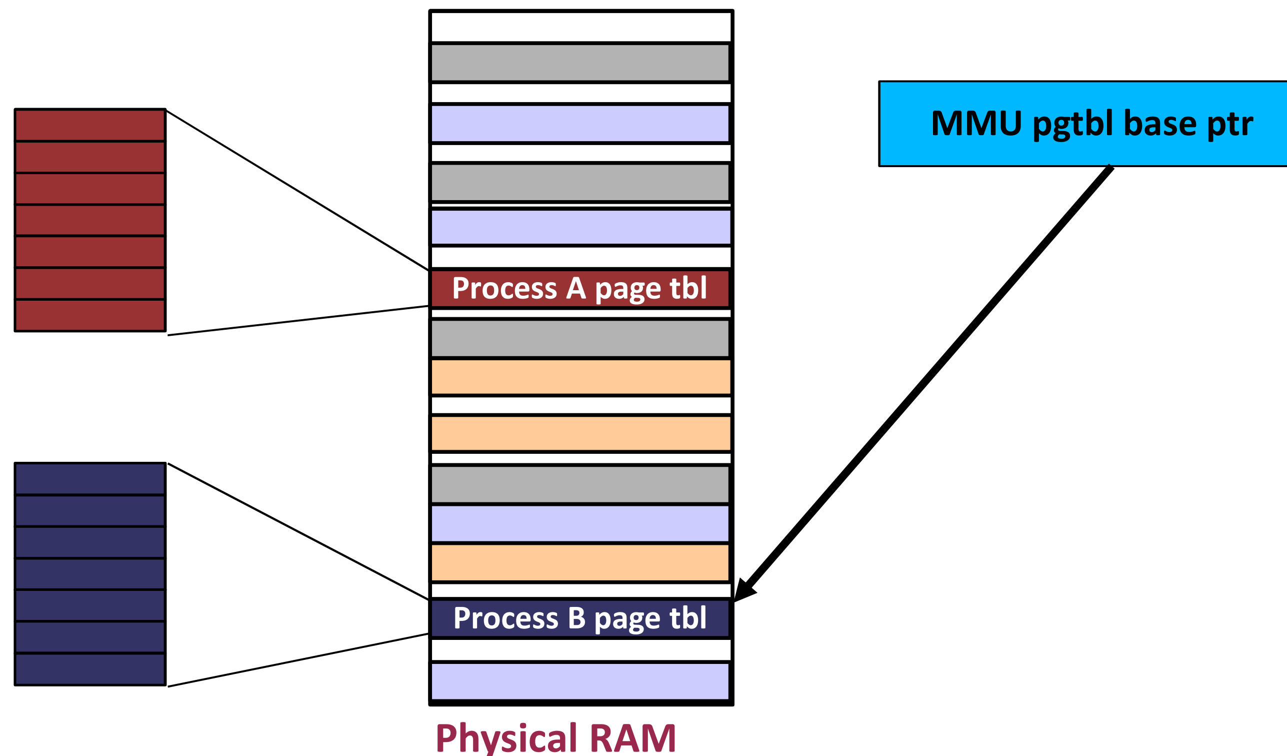
- **The R bit is used to decide which pages have been accessed recently.**
  - Next lecture we will talk about swapping "old" pages out to disk.
  - Need some way to keep track of what counts as an "old" page.
    - "Valid" bit will not be set for a page that is currently swapped out!
- **The M bit is used to tell whether a page has been modified**
  - Why might this be useful?
- **Protection bits used to prevent certain pages from being written.**
  - Why might this be useful?
- **How are these bits updated?**

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 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!)

Page Tables

- **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.

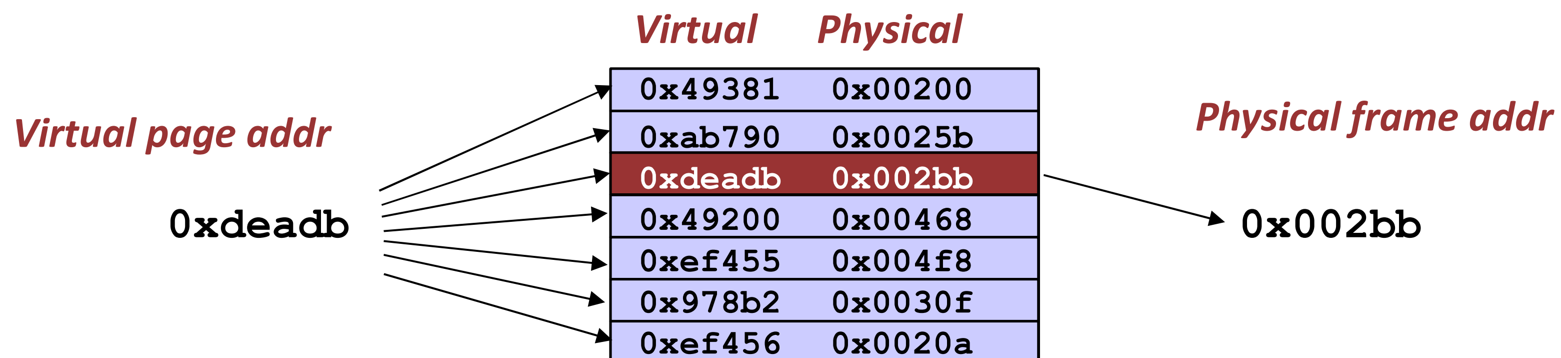


The TLB

- **Now we've introduced a high overhead for address translation**
  - On every memory access, must have a *separate* access to consult the page tables!
- **Solution: *Translation Lookaside Buffer (TLB)***
  - Very fast (but small) cache directly on the CPU
    - Intel Haswell architecture have separate data and instruction TLBs
  - Caches most recent virtual to physical address translations
  - A TLB miss requires that the MMU actually try to do the address translation

The TLB

- **Now we've introduced a high overhead for address translation**
  - On every memory access, must have a *separate* access to consult the page tables!
- **Solution: *Translation Lookaside Buffer (TLB)***
  - Very fast (but small) cache directly on the CPU
    - Intel Haswell architecture have separate data and instruction TLBs.
  - Caches most recent virtual to physical address translations
  - Implemented as highly (used to be fully) associative cache
  - **A TLB miss requires that the MMU actually try to do the address translation**



Loading the TLB

Two ways to load entries into the TLB.

1) MMU does it automatically

- MMU looks in TLB for an entry
- If not there, MMU handles the TLB miss directly
- MMU looks up virtual -> physical page mapping in page tables and loads new entry into TLB

2) Software-managed TLB

- TLB miss causes a trap to the OS
- OS looks up page table entry and loads new TLB entry

- **Why might a software-managed TLB be a good thing?**

Loading the TLB

Two ways to load entries into the TLB.

1) MMU does it automatically

- MMU looks in TLB for an entry
- If not there, MMU handles the TLB miss directly
- MMU looks up virtual -> physical page mapping in page tables and loads new entry into TLB

2) Software-managed TLB

- TLB miss causes a trap to the OS
- OS looks up page table entry and loads new TLB entry

- **Why might a software-managed TLB be a good thing?**
  - OS can dictate the page table format!
    - MMU does not directly consult or modify page tables.
  - Gives a lot of flexibility for OS designers to decide memory management policies
  - But ... requires TLB misses even for updating modified/referenced bits in PTE
    - OS must now handle many more TLB misses, with some performance impact.

Page Table Size

- How big are the page tables for a process?
- Well ... we need one PTE per page.
- Say we have a 32-bit address space, and the page size is 4KB
- How many pages?

Page Table Size

- **How big are the page tables for a process?**
- **Well ... we need one PTE per page.**
- **Say we have a 32-bit address space, and the page size is 4KB**
- **How many pages?**
  - 2^32 == 4GB / 4KB per page == 1,048,576 (1 M pages)
- **How big is each PTE?**
  - Depends on the CPU architecture ... on the x86, it's 4 bytes.
- **So, the total page table size is: 1 M pages * 4 bytes/PTE = 4 Mbytes**
  - And that is *per process*
  - *If we have 100 running processes, that's over 400 Mbytes of memory just for the page tables.*
- ***Solution: Swap the page tables out to disk!***

Application Perspective

Remember our three requirements for memory management:

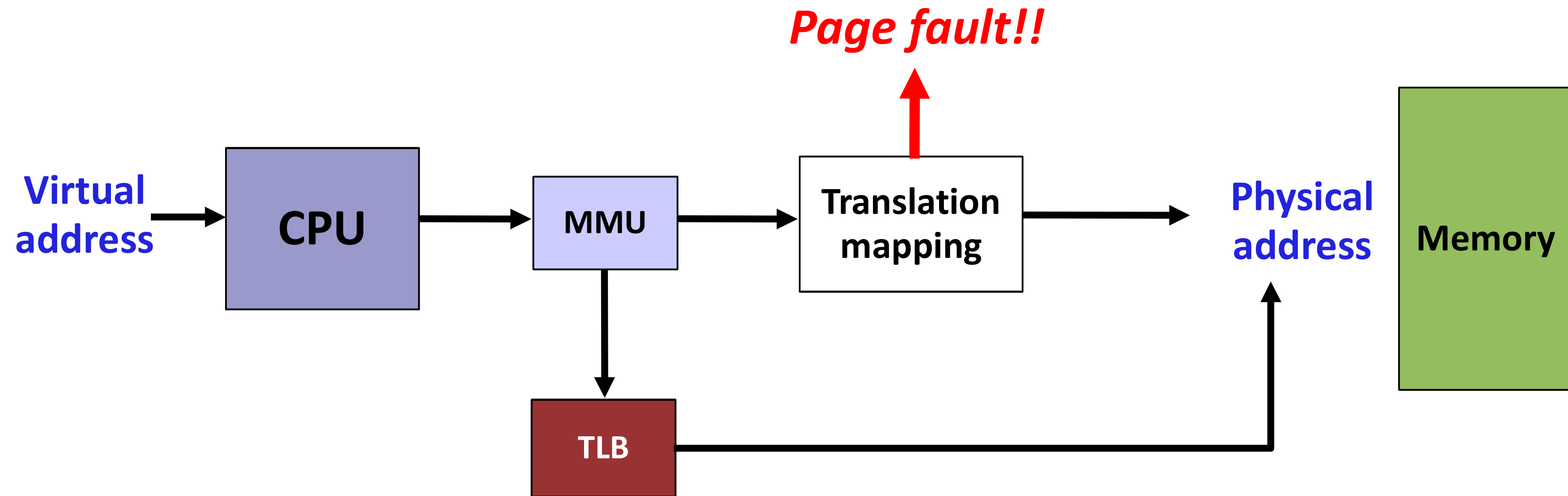
- **Isolation**
  - One process cannot access another's pages. Why?
    - Process can only refer to its own virtual addresses.
    - O/S responsible for ensuring that each process uses disjoint **physical** pages
- **Fast Translation**
  - Translation from virtual to physical is fast. Why?
    - MMU (on the CPU) translates each virtual address to a physical address.
    - TLB caches recent virtual->physical translations
- **Fast Context Switching**
  - Context Switching is fast. Why?
    - Only need to swap pointer to current page tables when context switching!
    - (Though there is one more step ... what is it?)

More Issues

- What happens when a page is not in memory?

- How do we prevent having page tables take up a huge amount of memory themselves?

Page Faults



- When a virtual address translation cannot be performed, it's called a *page fault*

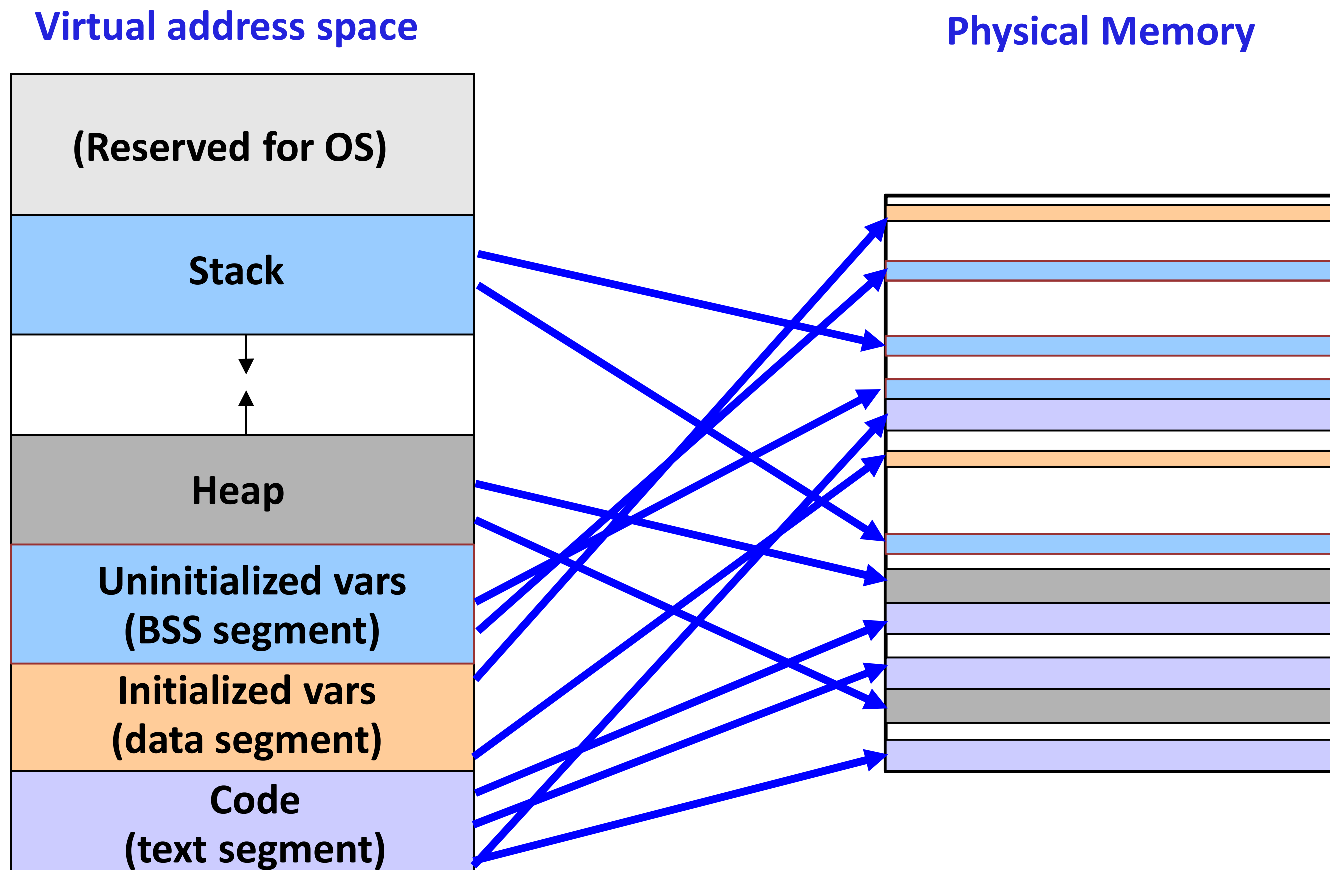
Page Faults

- **Recall the PTE format:**

  - Valid bit indicates whether a page translation is valid
  - If Valid bit is 0, then a page fault will occur
  - Page fault will also occur if attempt to write a read-only page (based on the Protection bits, not the valid bit)
    - This is sometimes called a protection fault

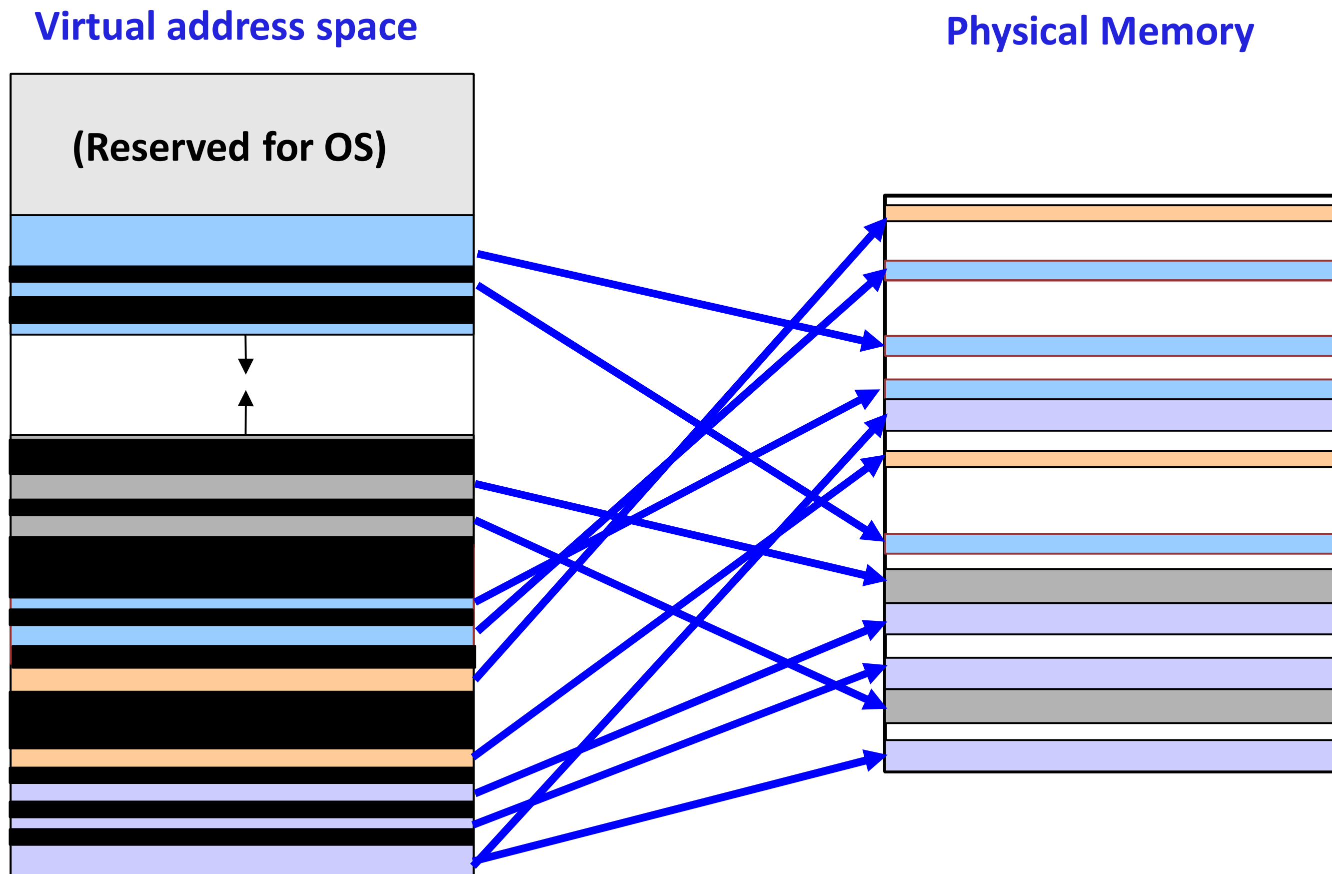
Demand Paging

- **Does it make sense to read an entire program into memory at once?**
  - No! Remember that only a small portion of a program's code may be used!
  - For example, if you never use the “print” capability of Powerpoint on this machine...



Demand Paging

- **Does it make sense to read an entire program into memory at once?**
  - No! Remember that only a small portion of a program's code may be used!
  - For example, if you never use the “print” capability of Powerpoint...



Where are the “holes” ??

Where are the “holes”?

Three kinds of “holes” in a process's page tables:

1. Pages that are on disk

- Pages that were swapped out to disk to save memory
- Also includes code pages in an executable file
  - When a page fault occurs, load the corresponding page from disk

2. Pages that have not been accessed yet

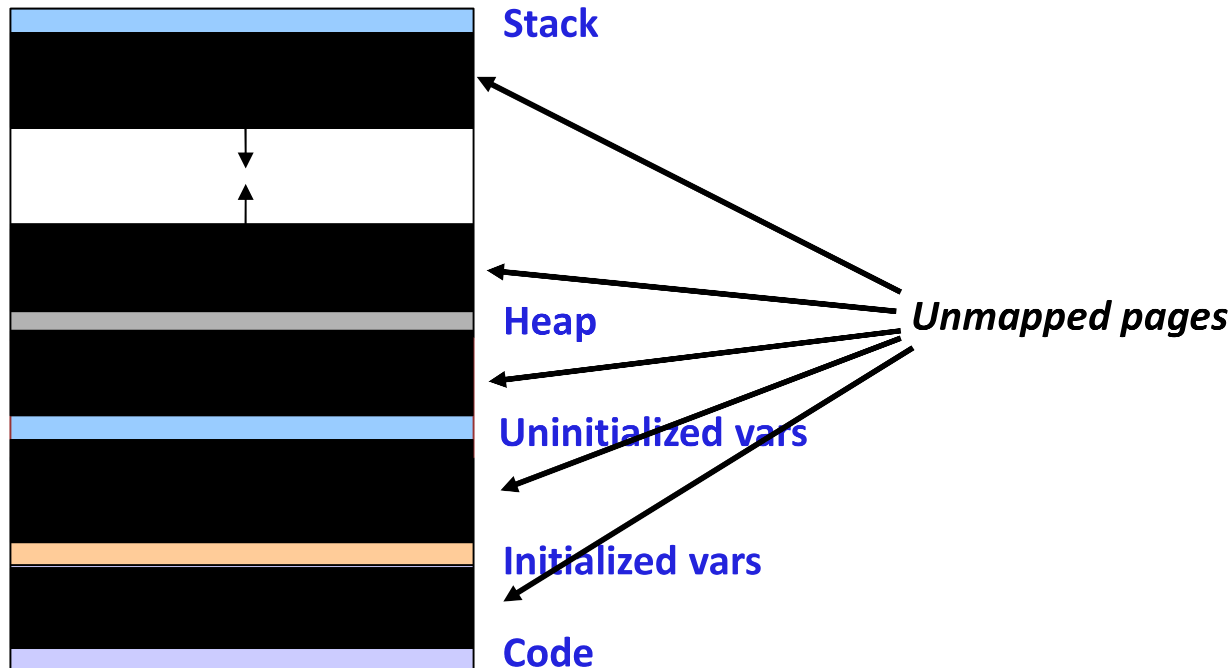
- For example, newly-allocated memory
  - When a page fault occurs, allocate a new physical page
- What are the contents of the newly-allocated page???

3. Pages that are invalid

- For example, the “null page” at address 0x0
  - When a page fault occurs, kill the offending process

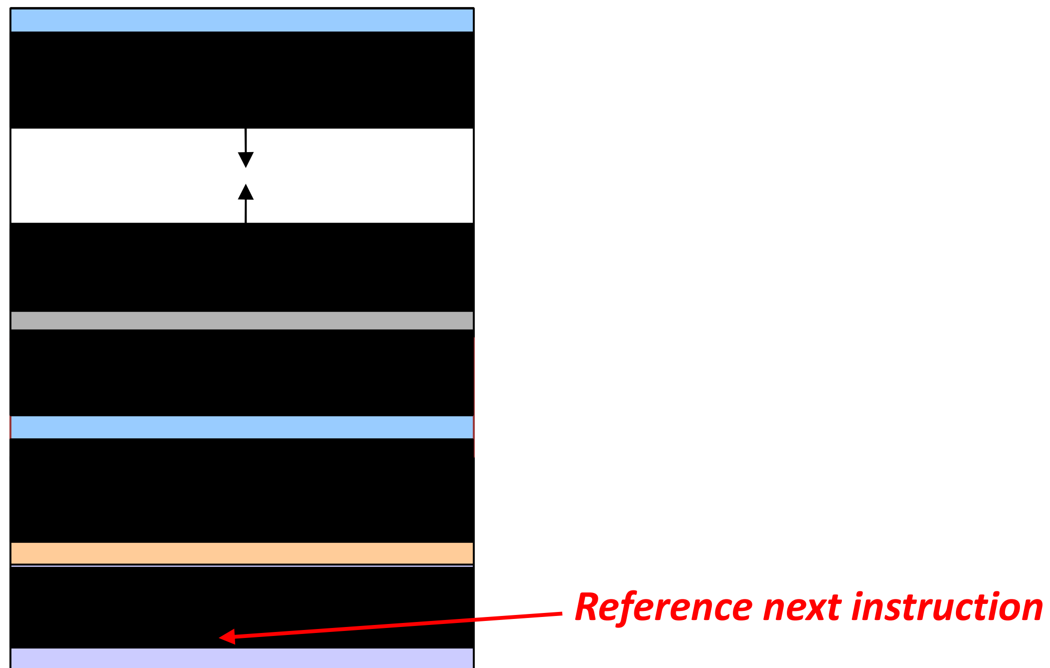
Starting up a process

- What does a process's address space look like when it first starts up?



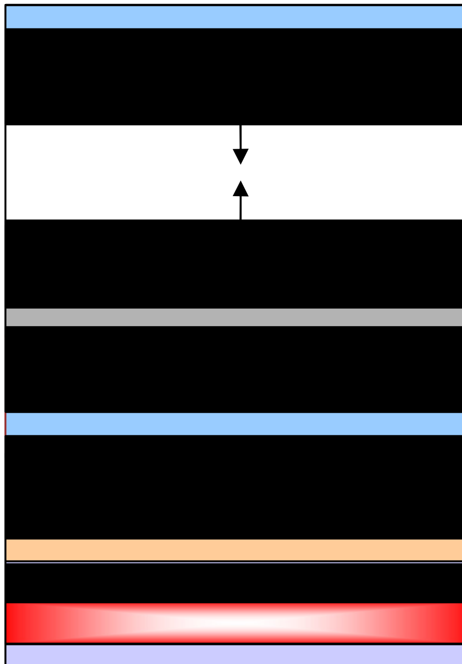
Starting up a process

- What does a process's address space look like when it first starts up?



Starting up a process

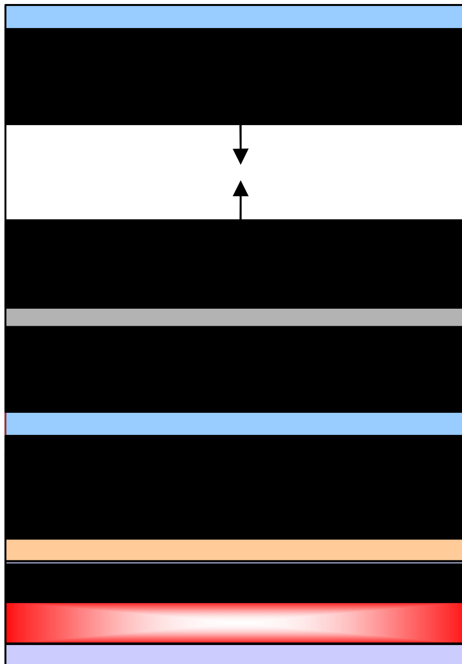
- What does a process's address space look like when it first starts up?

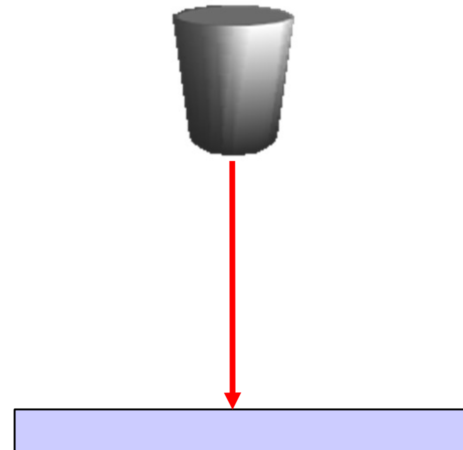


Page fault!!!

Starting up a process

- What does a process's address space look like when it first starts up?





Starting up a process

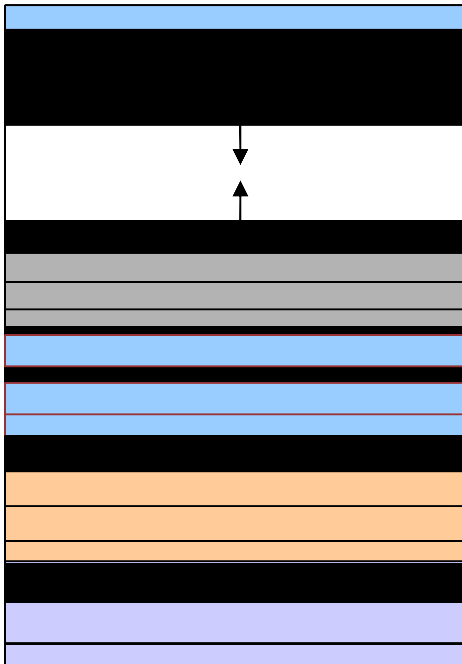
- What does a process's address space look like when it first starts up?

Starting up a process

- What does a process's address space look like when it first starts up?

Starting up a process

- What does a process's address space look like when it first starts up?



Over time, more pages are brought in from the executable as needed

Uninitialized variables and the heap

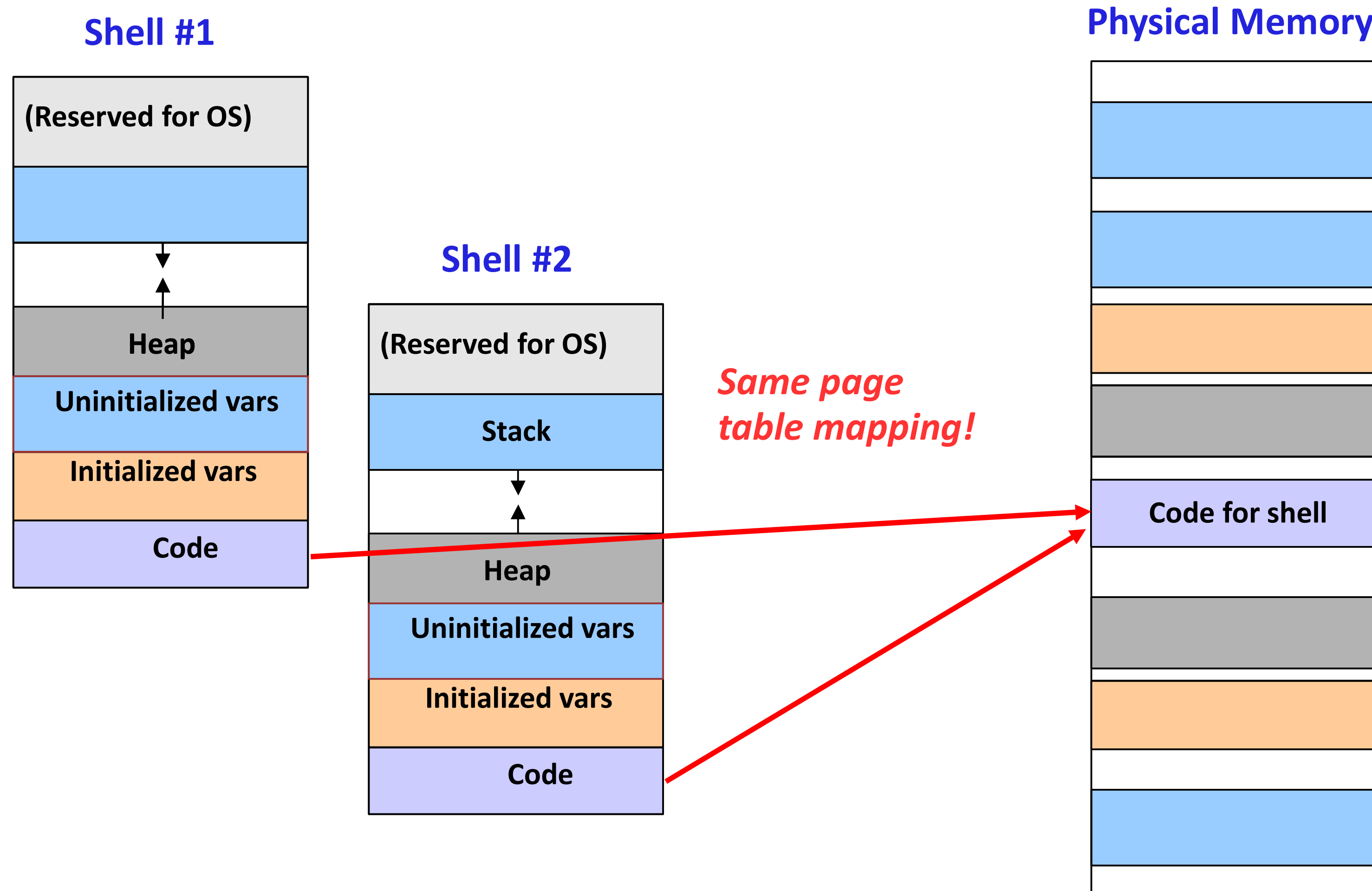
- **Page faults bring in pages from the executable file for:**
  - Code (text segment) pages
  - Initialized variables
- **What about uninitialized variables and the heap?**
- **Say I have a global variable "`int c`" in the program ... what happens when the process first accesses it?**

Uninitialized variables and the heap

- **Page faults bring in pages from the executable file for:**
  - Code (text segment) pages
  - Initialized variables
- **What about uninitialized variables and the heap?**
- **Say I have a global variable "`int c`" in the program ... what happens when the process first accesses it?**
  - Page fault occurs
  - OS looks at the page and realizes it corresponds to a *zero page*
  - Allocates a new physical frame in memory **and sets all bytes to zero**
    - **Why???**
  - Maps the frame into the address space
- **What about the heap?**
  - **`malloc()`** just asks the OS to map new zero pages into the address space
  - Page faults allocate new empty pages as above

More Demand Paging Tricks

- **Paging can be used to allow processes to share memory**
  - A significant portion of many process's address space is identical
  - For example, multiple copies of your shell all have the same exact code!



More Demand Paging Tricks

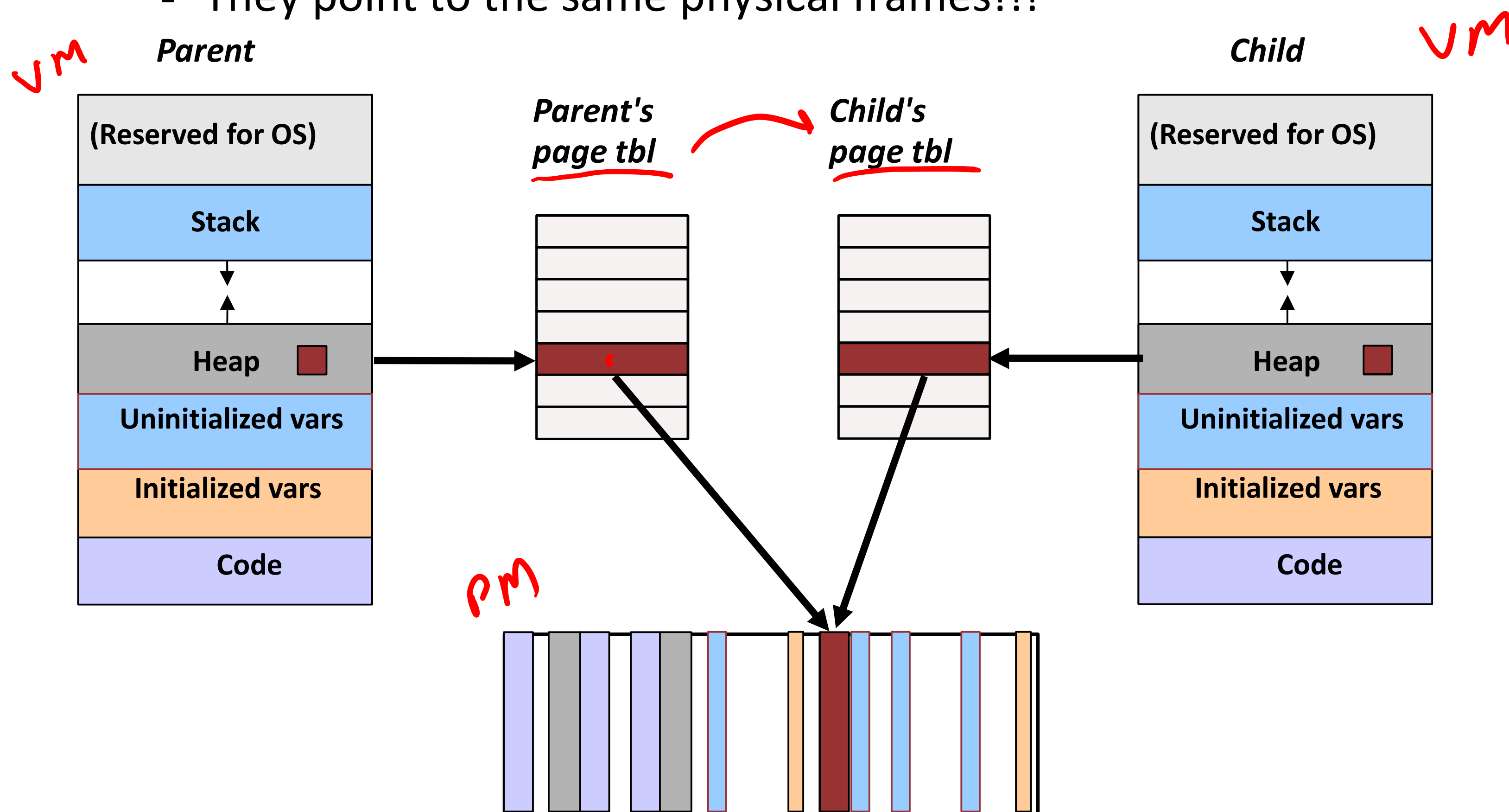
- **This can be used to let different processes share memory**
  - UNIX supports shared memory through the **`shmget/shmat/shmdt`** system calls
  - Allocates a region of memory that is shared across multiple processes
  - Some of the benefits of multiple threads per process, but the rest of the processes address space is protected
    - Why not just use multiple processes with shared memory regions?
- **Memory-mapped files**
  - Idea: Make a file on disk look like a block of memory
  - Works just like faulting in pages from executable files
    - In fact, many OS's use the same code for both
  - One wrinkle: Writes to the memory region must be reflected in the file
  - How does this work?
    - When writing to the page, mark the “modified” bit in the PTE
    - When page is removed from memory, write back to original file

Remember `fork()`?

- **`fork()` creates an exact copy of a process**
  - What does this imply about page tables?
- **When we fork a new process, does it make sense to make a copy of all of its memory?**
  - Why or why not?
- **What if the child process doesn't end up touching most of the memory the parent was using?**
  - Extreme example: What happens if a process does an **`exec()`** immediately after **`fork()`** ?

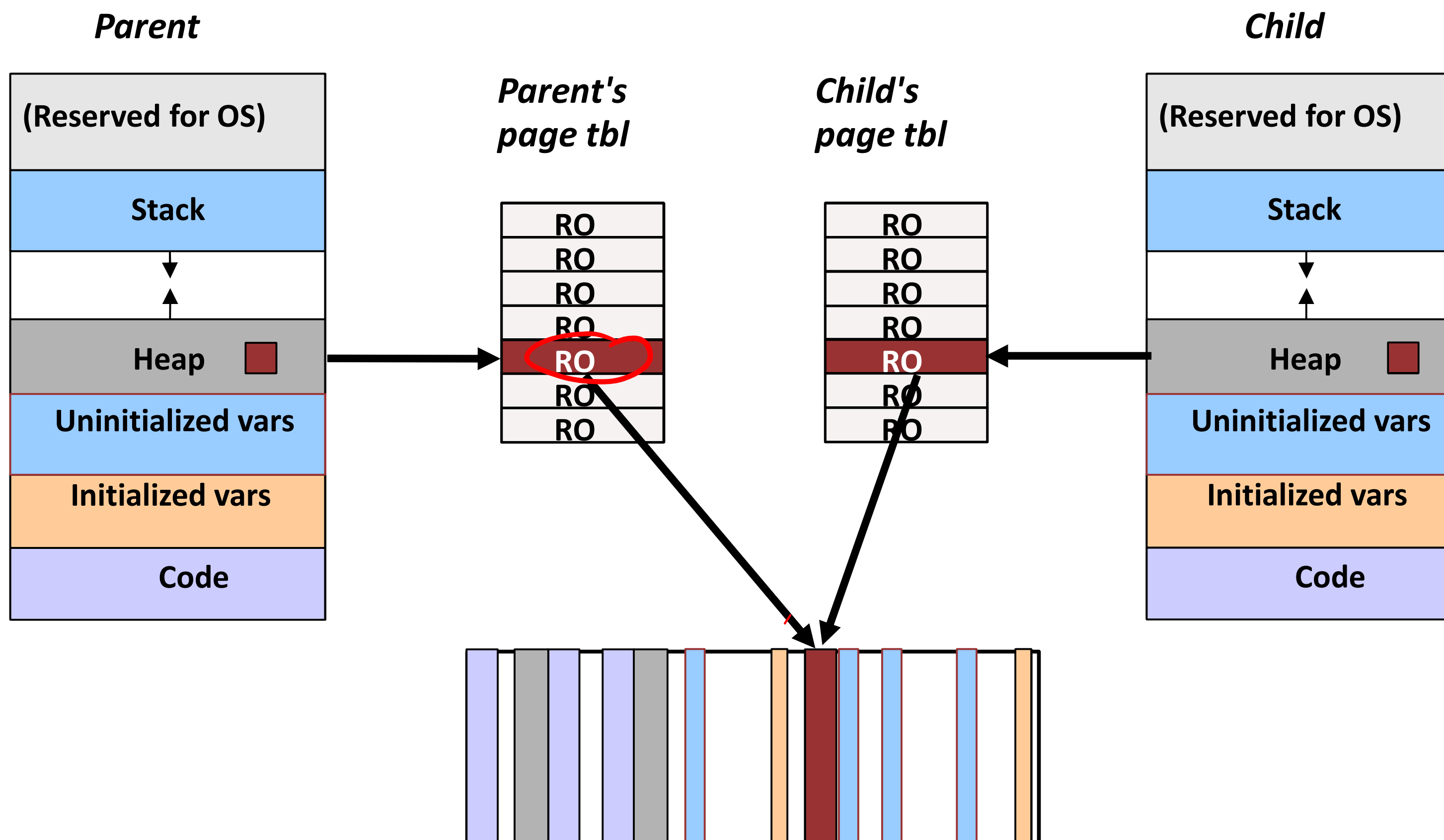
Copy-on-write

- Idea: Give the child process access to the same memory, but don't let it write to any of the pages directly!
  - 1) Parent forks a child process
  - 2) Child gets a copy of the parent's page tables
    - They point to the same physical frames!!!



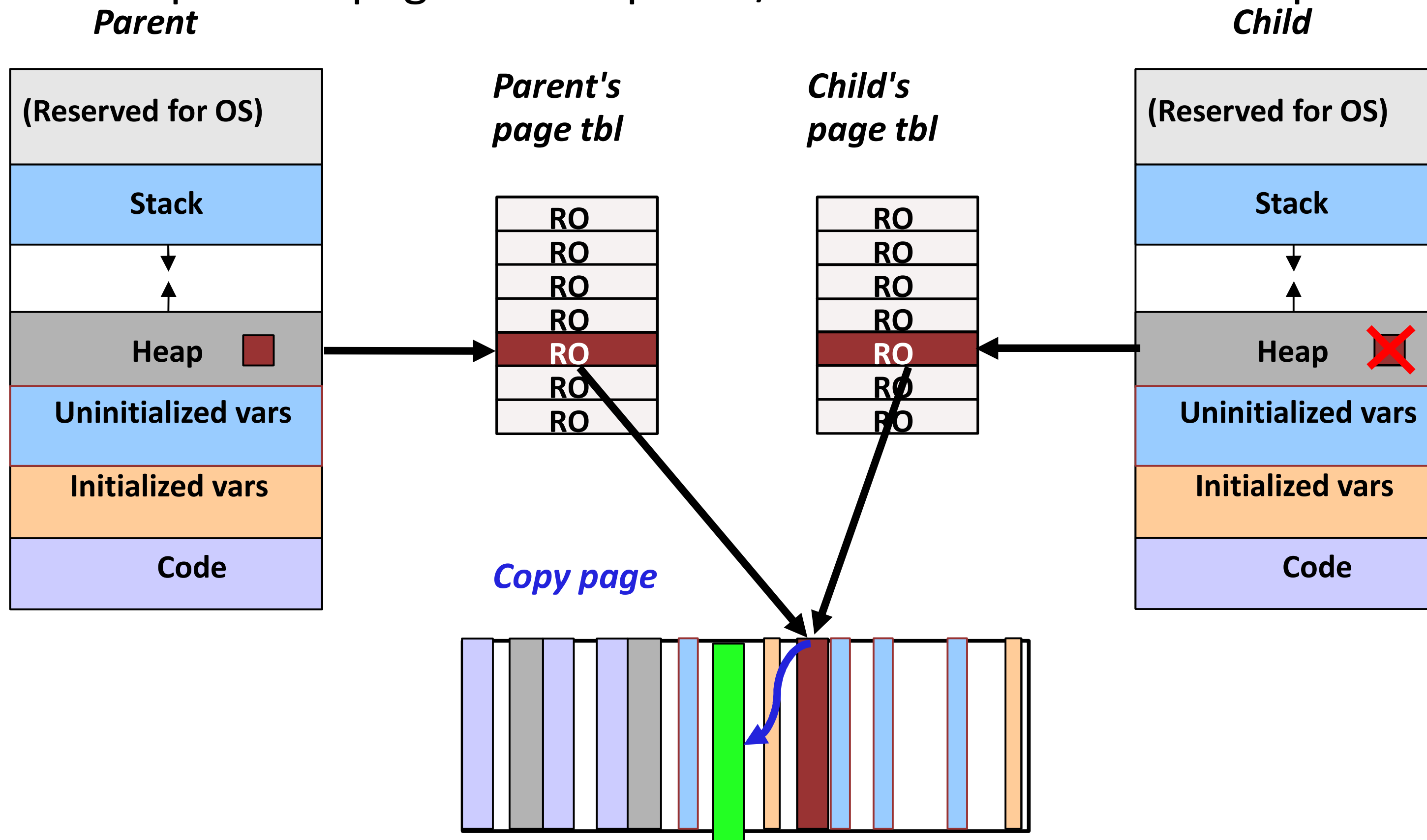
Copy-on-write

- All pages (both parent and child) marked read-only
  - Why???



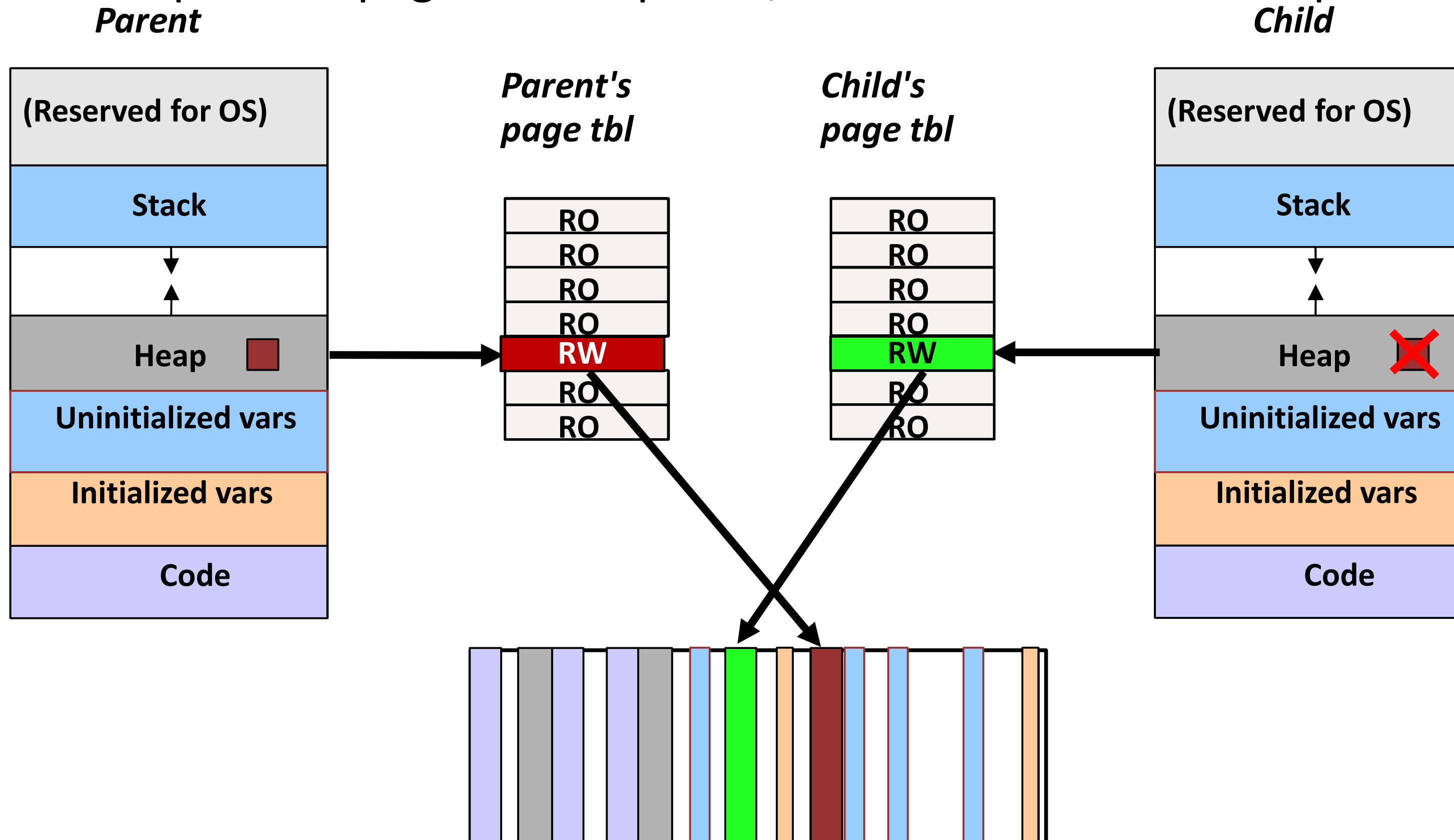
Copy-on-write

- **What happens when the child *reads* the page?**
  - Just accesses same memory as parent .... niiiiiice
- **What happens when the child *writes* the page?**
  - Protection fault occurs (page is read-only!)
  - OS copies the page and maps it R/W into the child's addr space



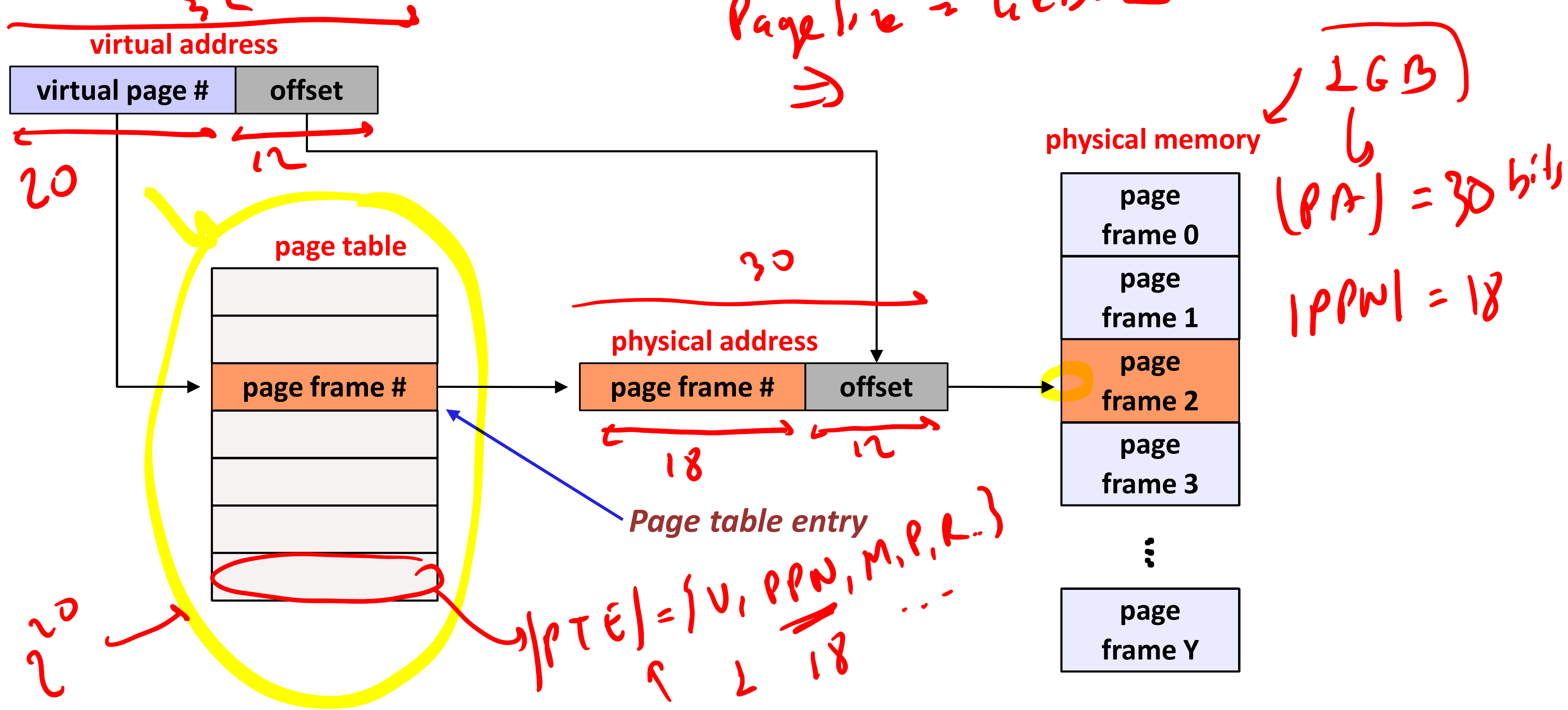
Copy-on-write

- **What happens when the child *reads* the page?**
  - Just accesses same memory as parent .... niiiiiice
- **What happens when the child *writes* the page?**
  - Protection fault occurs (page is read-only!)
  - OS copies the page and maps it R/W into the child's addr space



Page Tables

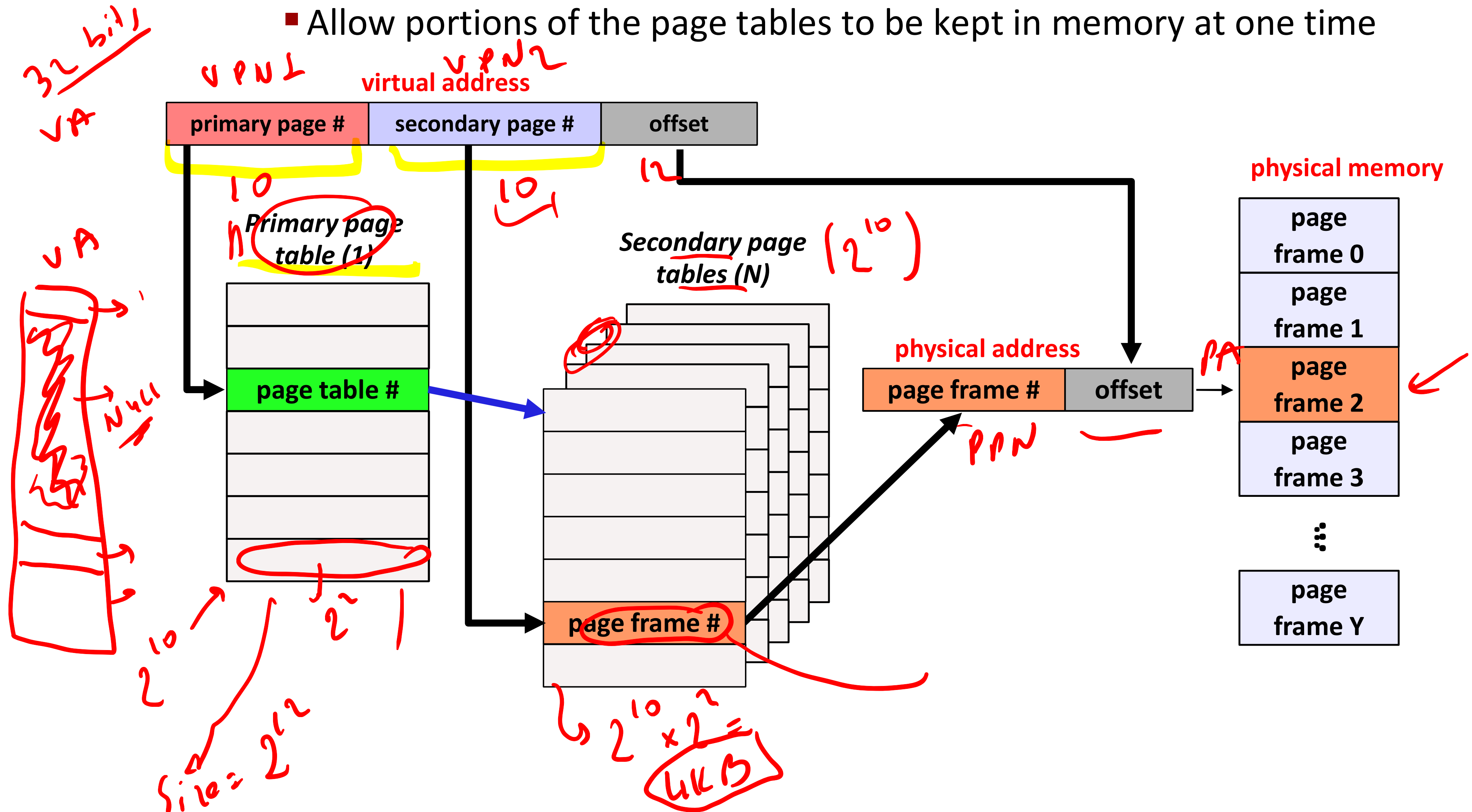
- Remember how paging works:



- Recall that page tables for one process can be very large!
  - 2^20 PTEs * 4 bytes per PTE = 4 Mbytes per process

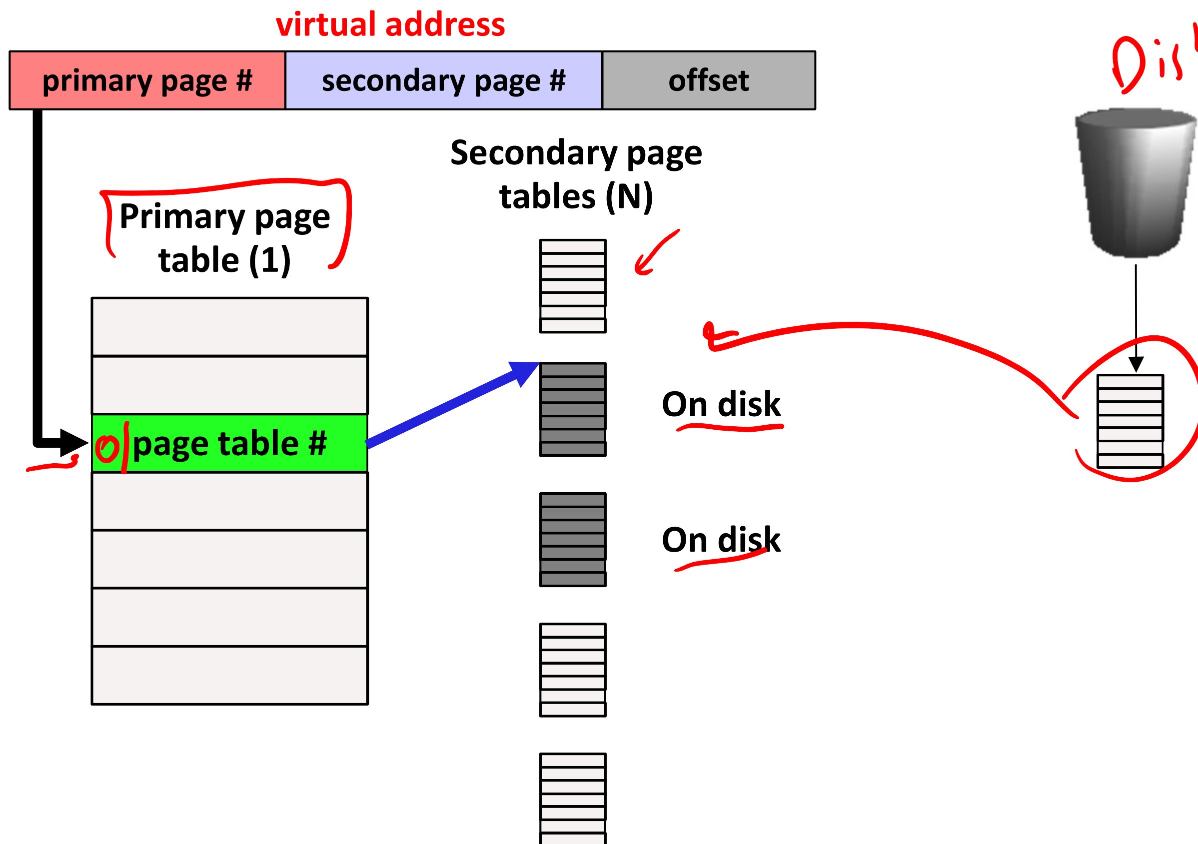
Multilevel Page Tables

- Problem: Can't hold all of the page tables in memory
- Solution: Page the page tables!
  - Allow portions of the page tables to be kept in memory at one time



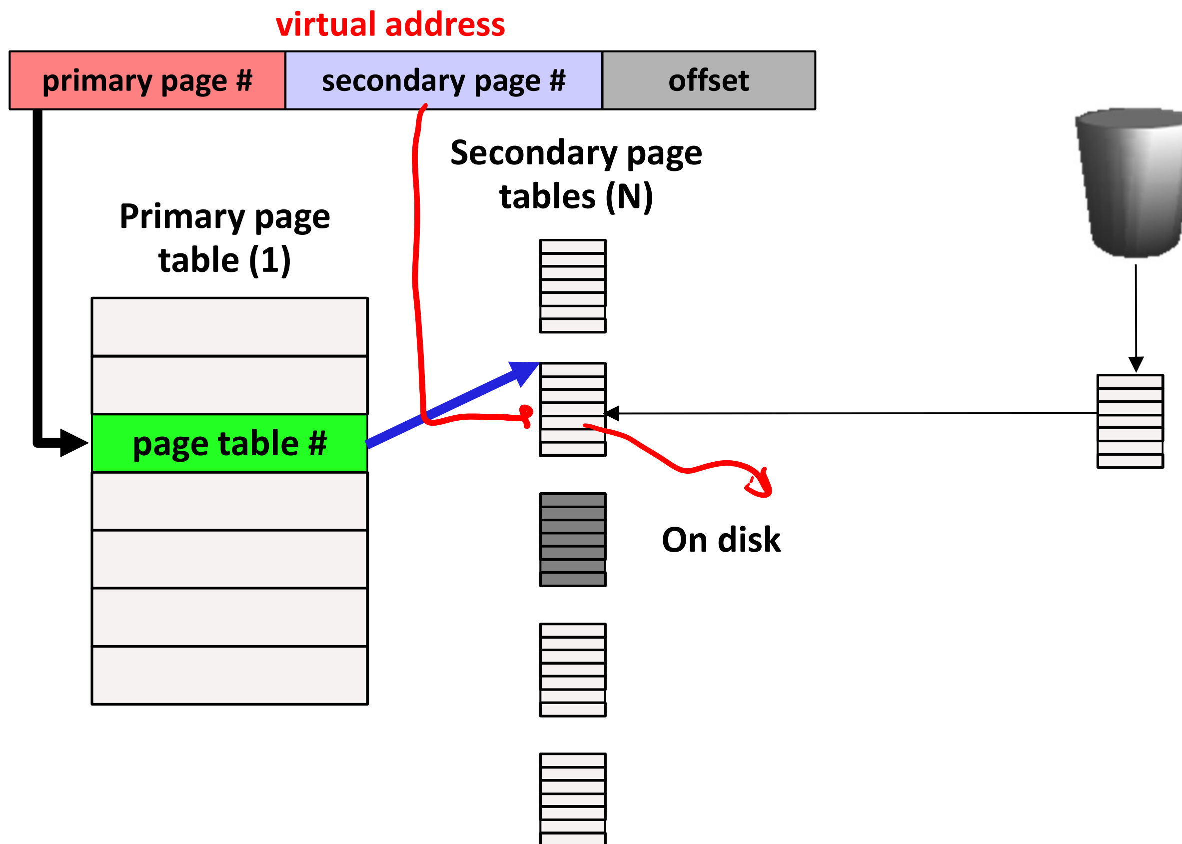
Multilevel Page Tables

- **Problem: Can't hold all of the page tables in memory**
- **Solution: Page the page tables!**
  - Allow portions of the page tables to be kept in memory at one time



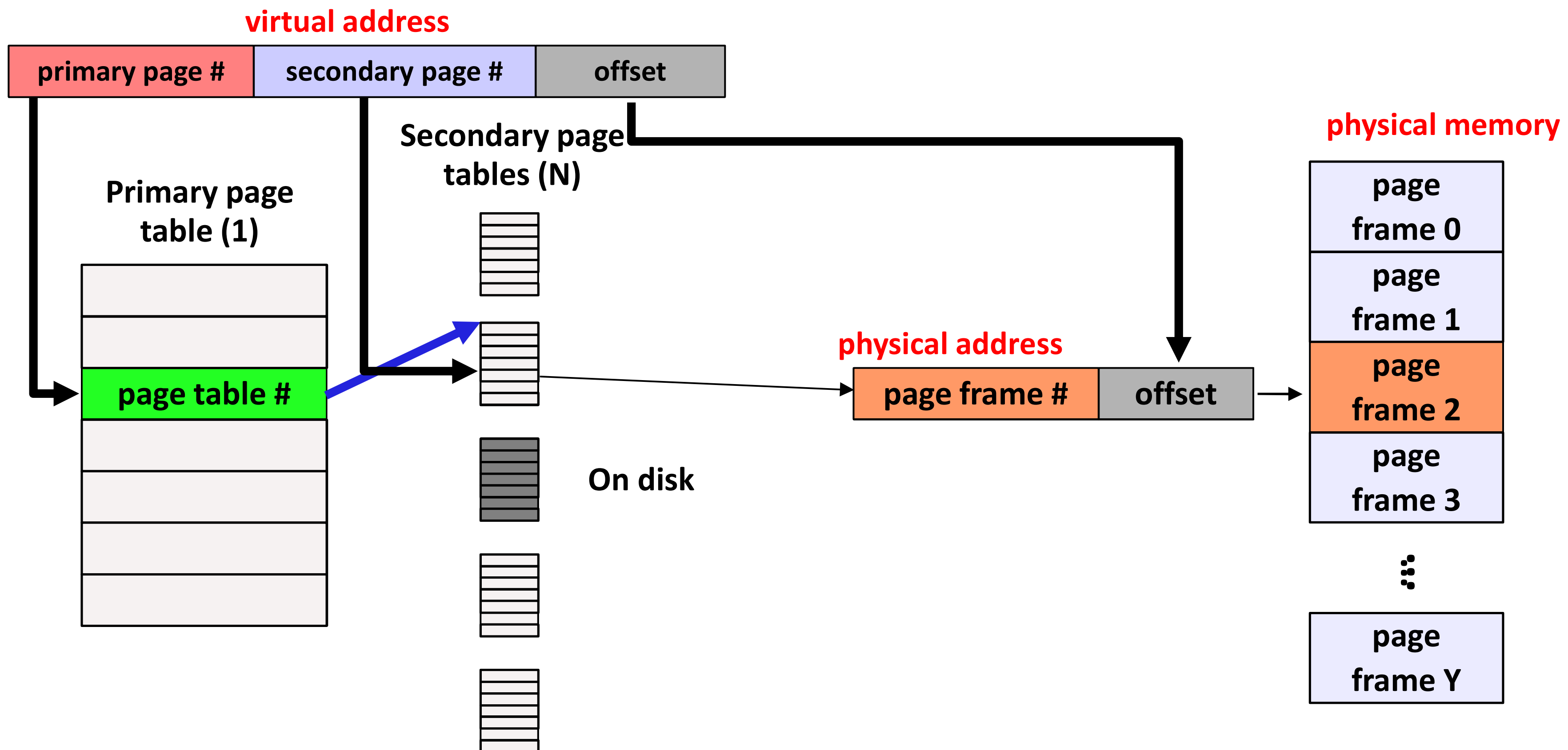
Multilevel Page Tables

- **Problem: Can't hold all of the page tables in memory**
- **Solution: Page the page tables!**
  - Allow portions of the page tables to be kept in memory at one time



Multilevel Page Tables

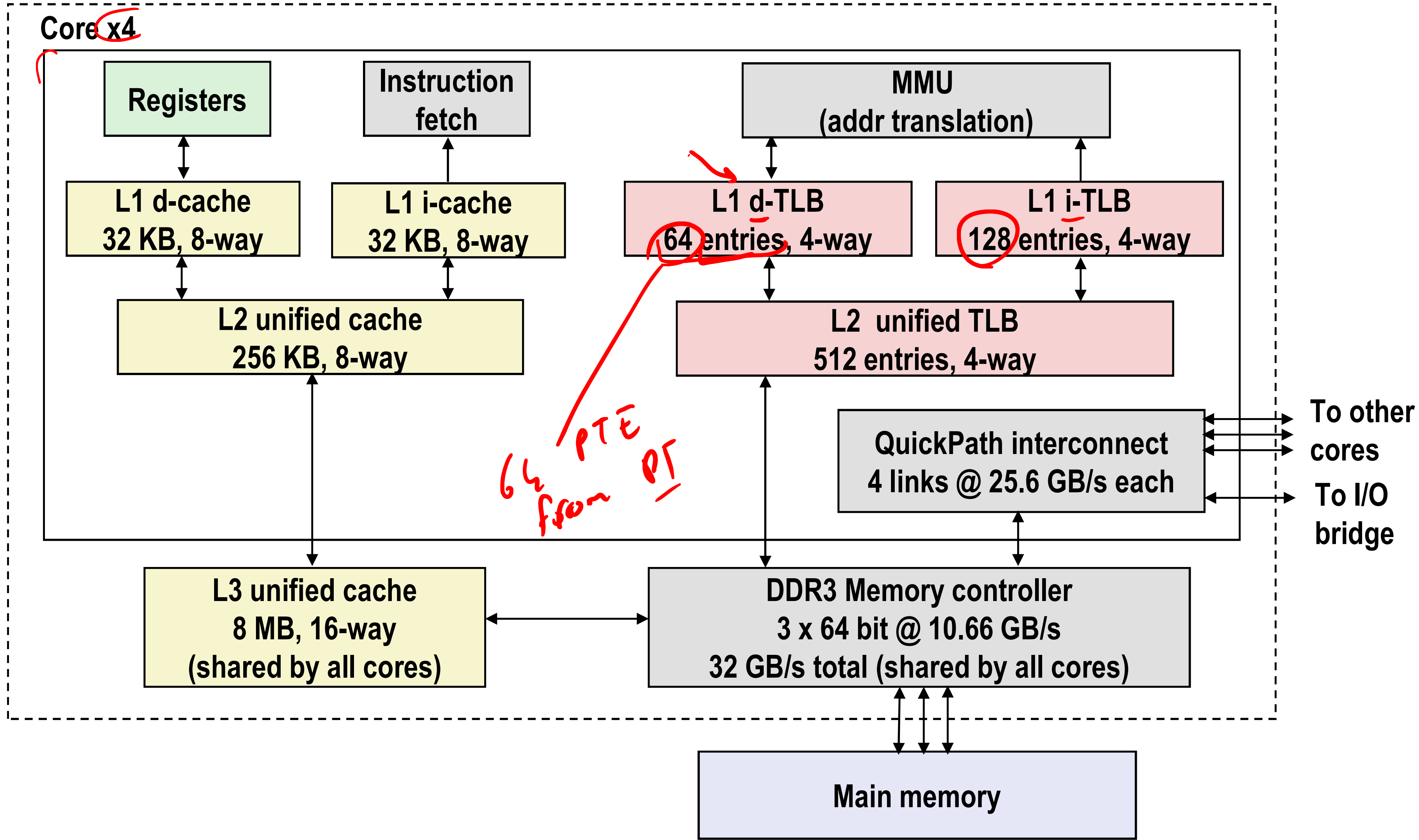
- **Problem: Can't hold all of the page tables in memory**
- **Solution: Page the page tables!**
  - Allow portions of the page tables to be kept in memory at one time



Multilevel page tables

- **With two levels of page tables, how big is each table?**
  - Say we allocate 10 bits to the primary page, 10 bits to the secondary page, 12 bits to the page offset
  - Primary page table is then 2^10 * 4 bytes per PTE == 4 KB
  - Secondary page table is also 4 KB
    - Hey ... that's exactly the size of a page on most systems ... cool
- **What happens on a page fault?**
  - MMU looks up index in primary page table to get secondary page table
    - Assume this is “wired” to physical memory
  - MMU tries to access secondary page table
    - May result in another page fault to load the secondary table!
  - MMU looks up index in secondary page table to get PFN
  - CPU can then access physical memory address
- **Issues**
  - Page translation has very high overhead
    - Up to three memory accesses plus two disk I/Os!!
  - TLB usage is clearly very important.

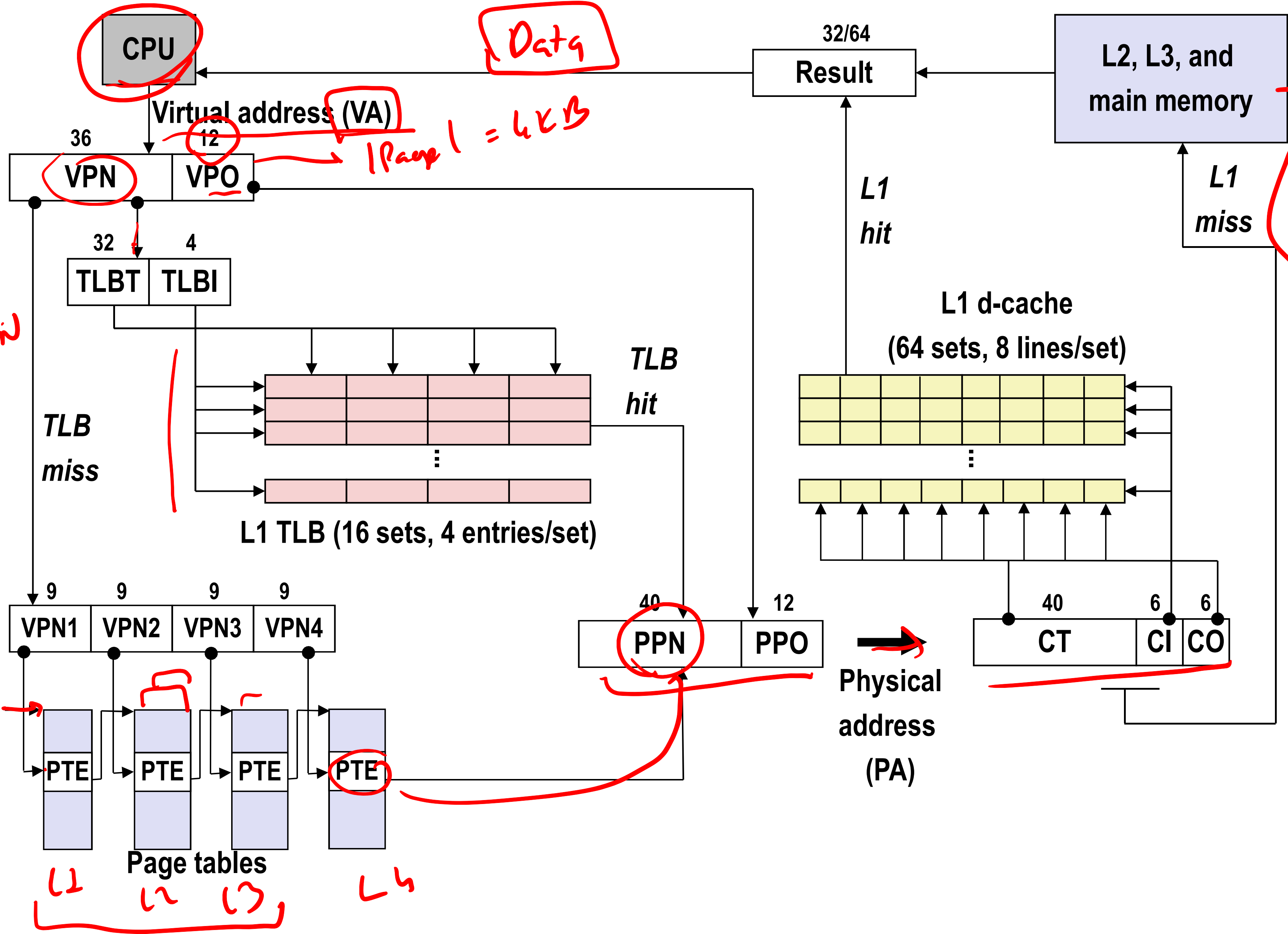
Intel Core i7 Memory System



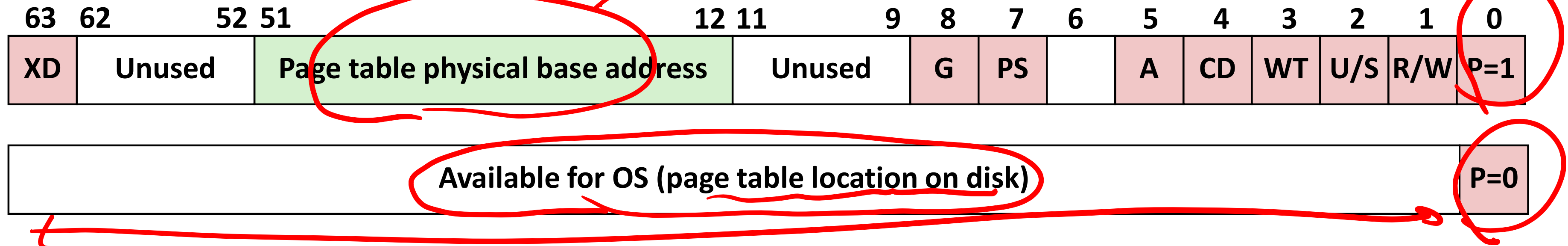
Review of Symbols

- **Basic Parameters**
  - **$N = 2^n$** : Number of addresses in virtual address space
  - **$M = 2^m$** : Number of addresses in physical address space
  - **$P = 2^p$** : Page size (bytes)
- **Components of the virtual address (VA)**
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
- **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number
  - **CO**: Byte offset within cache line
  - **CI:** Cache index
  - **CT**: Cache tag

End-to-end Core i7 Address Translation



Core i7 Level 1-3 Page Table Entries



Each entry references a 4K child page table. Significant fields:

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

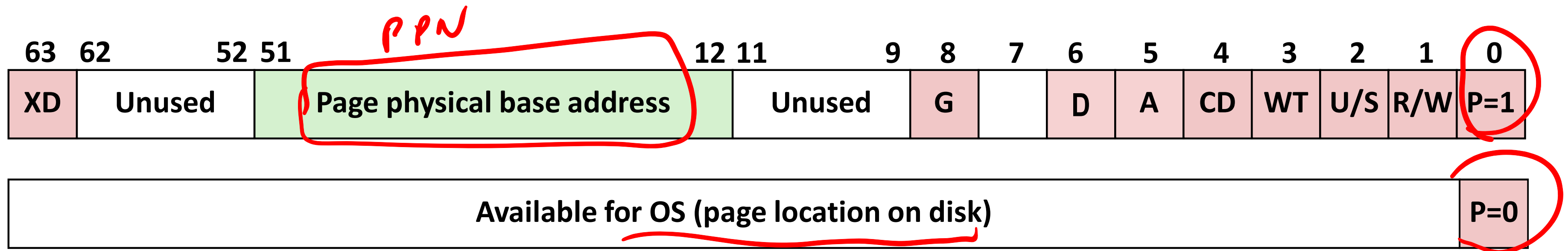
A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

XD: Disable or enable instruction fetches from all pages reachable from this PTE.

Core i7 Level 4 Page Table Entries



Each entry references a 4K child page. Significant fields:

P: Child page is present in memory (1) or not (0)

R/W: Read-only or read-write access permission for child page

U/S: User or supervisor mode access

WT: Write-through or write-back cache policy for this page

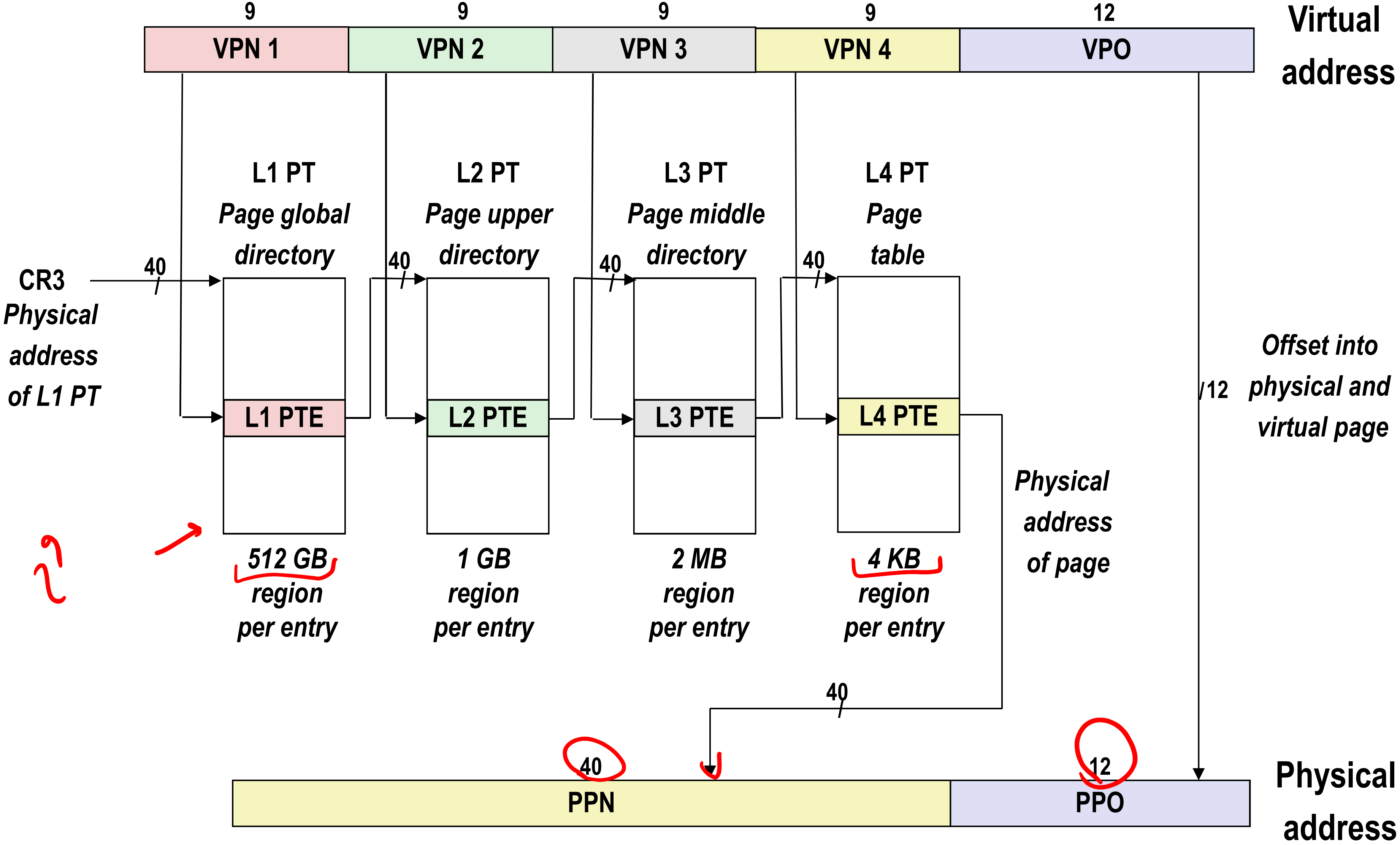
A: Reference bit (set by MMU on reads and writes, cleared by software)

D: Dirty bit (set by MMU on writes, cleared by software)

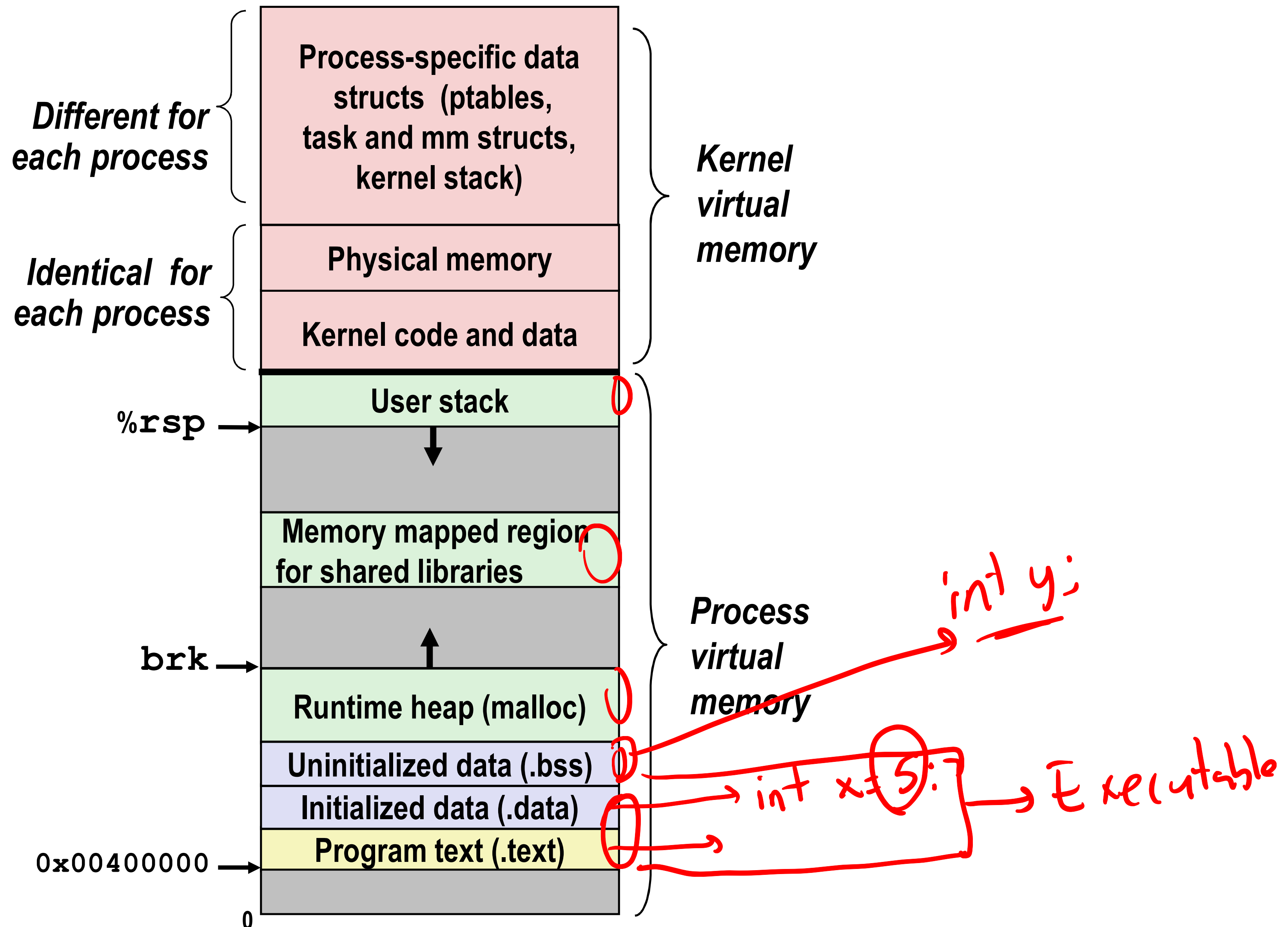
Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.

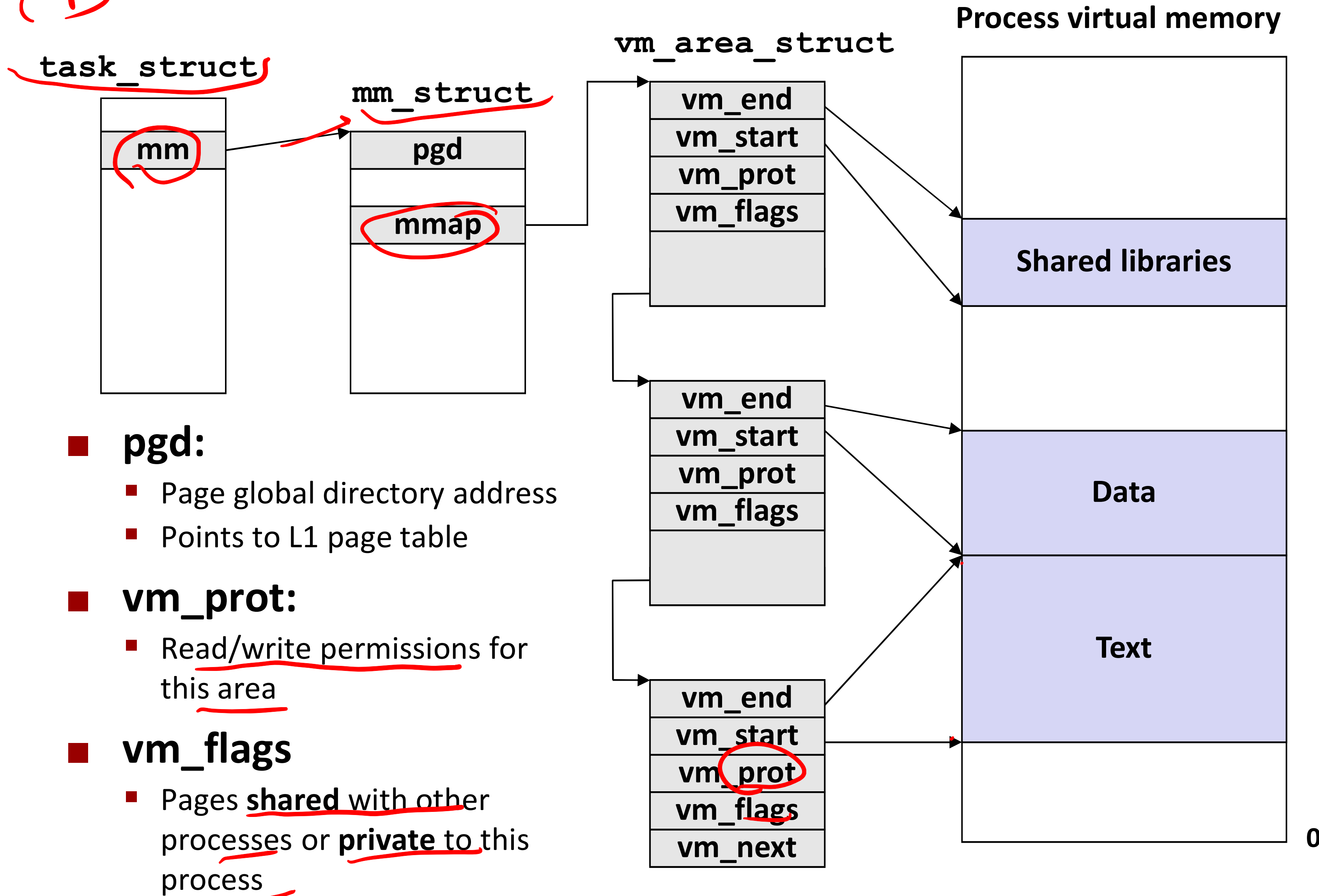
Core i7 Page Table Translation



Virtual Address Space of a Linux Process

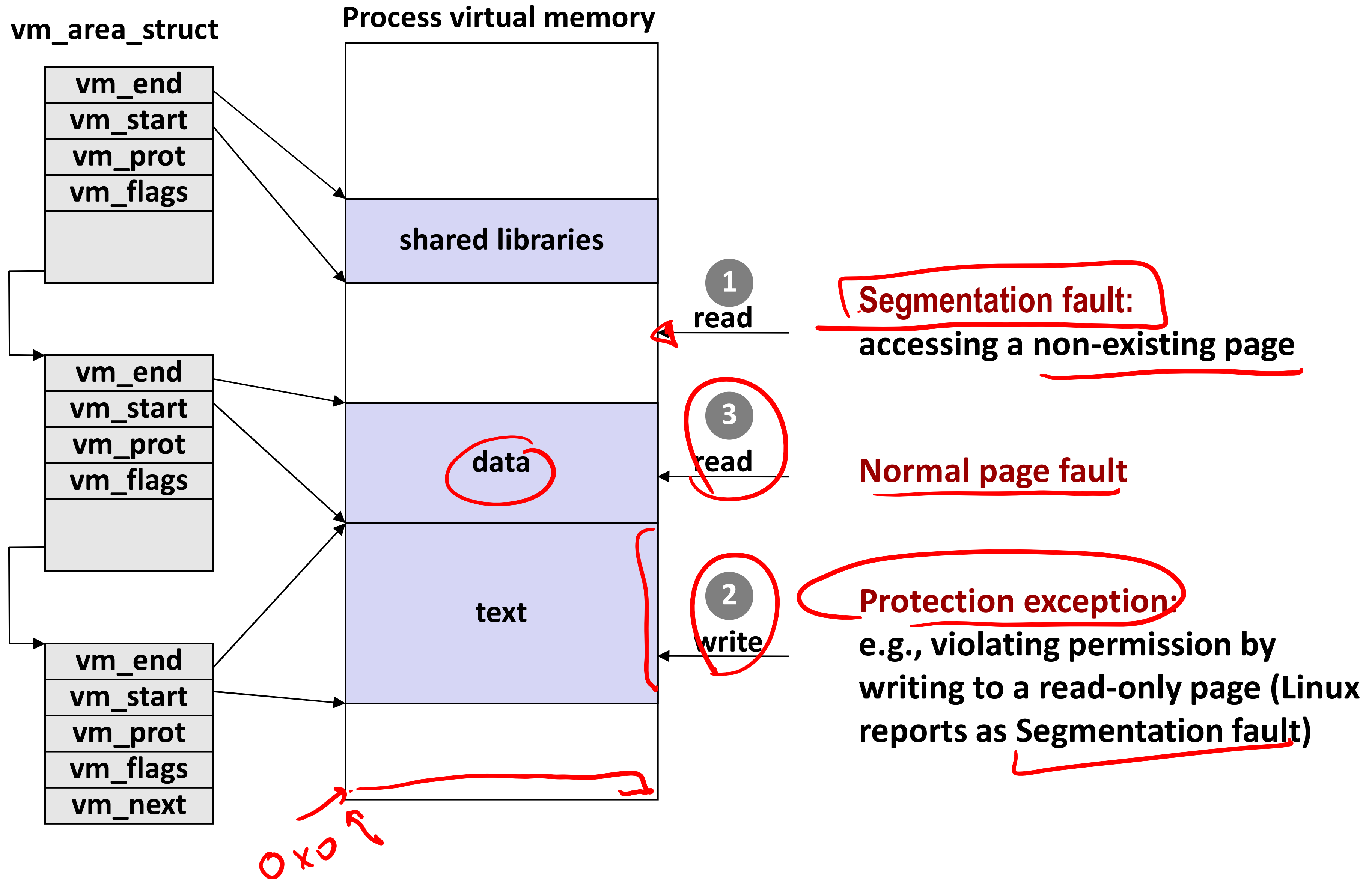


Linux Organizes VM as Collection of “Areas”

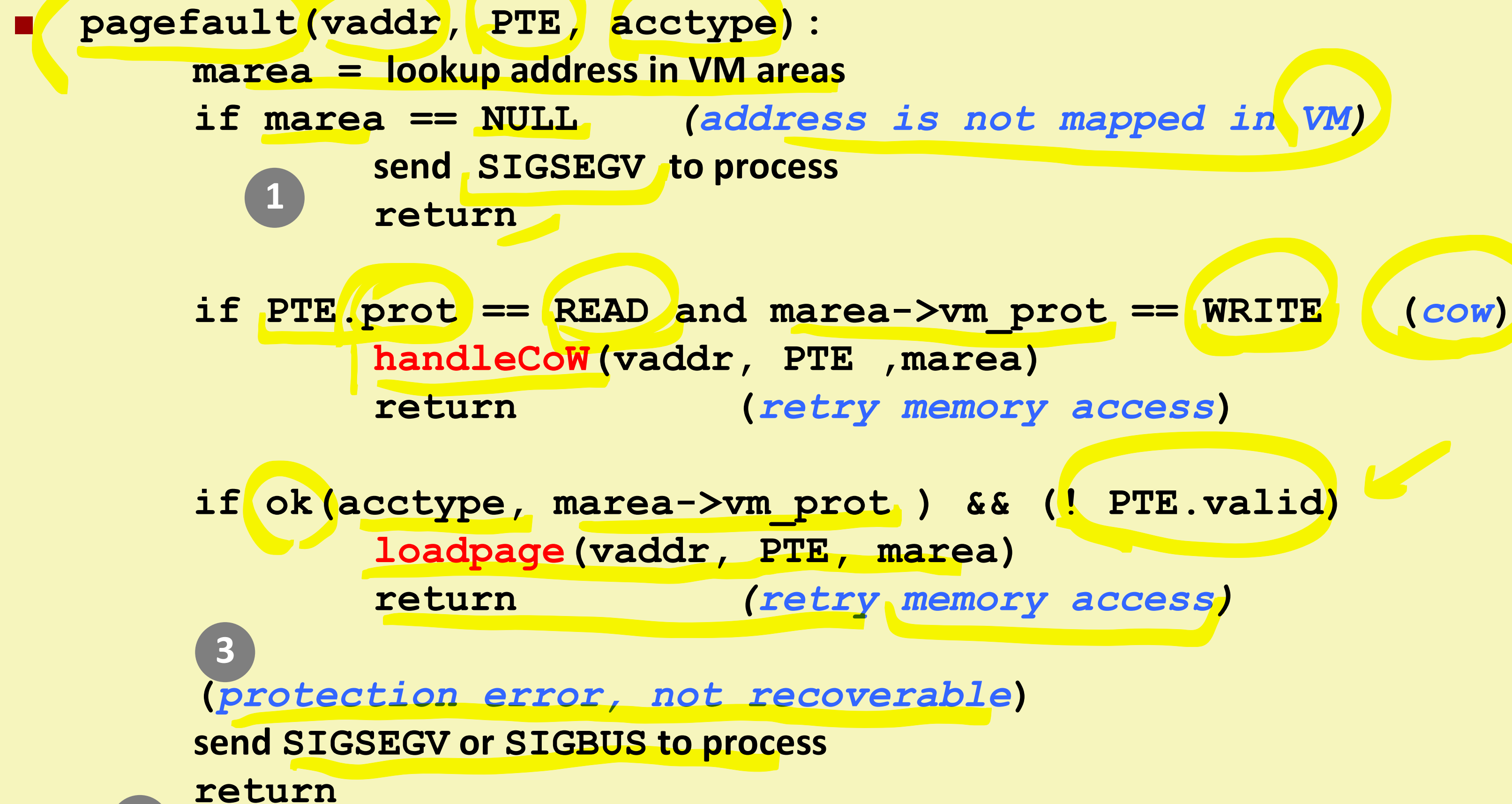


- **pgd:**
  - Page global directory address
  - Points to L1 page table
- **vm_prot:**
  - Read/write permissions for this area
- **vm_flags**
  - Pages **shared** with other processes or **private** to this process

Linux Page Fault Handling

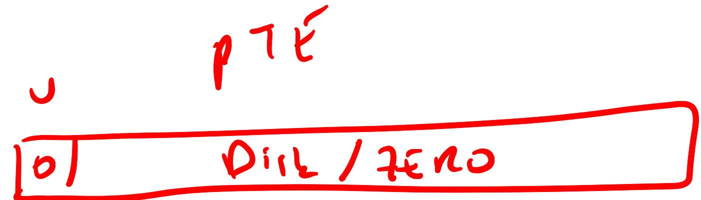


Page Fault Pseudo Code

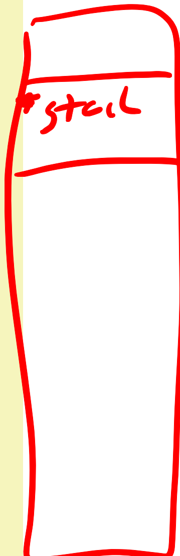


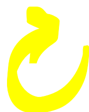
Load an Invalid Page

- **PTE is invalid. Either:**
  - Due to **demand paging**, it is not loaded yet
  - Page is **evicted**, to get free space in system.

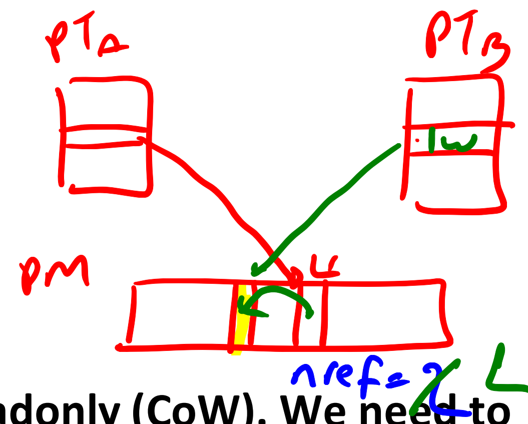


```
loadpage(vaddr, PTE, marea):
        newpage = page_alloc()          (allocate a new page)
        if marea is backed by a file
                (either a mmap'ed file, code or initialized
                 data segments, or swapped out)
                start IO for loading page from file
                make task sleep until I/O completes
        else                (an anonymous page with 0 content)
                memset(newpage, 0)
        PTE.valid = 1
        PTE.prot = marea.vm_prot
        PTE.address = newpage.pageno
        update_address_translation(vaddr, PTE)
```





Break Copy on Write (CoW)



- We tried to write a CoW page, which is marked as readonly (CoW). We need to break CoW and mark it as writable

```
handleCoW(vaddr,PTE, marea):
        page = PTE.address                  (frame information)
        if page.nrefs == 1:       (only one task refers this frame)
                (as last task, just mark it as writable)
                PTE.prot |= WRITE
                update_address_translation(vaddr, PTE)
        else               (there are others referring this frame)
                newpage = page_alloc()   (allocate a new frame)
                memcopy(page,newpage)
                PTE.prot |= WRITE
                PTE.valid = 1
                PTE.address = newpage
                --page.nrefs
                update_address_translation(vaddr, PTE)
```