Virtual Memory: Concepts

15-213: Introduction to Computer Systems
“17th” Lecture, July 8, 2020

Instructor:

Sol Boucher

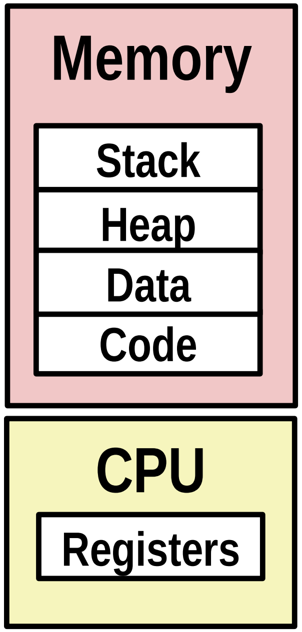
Today

- **Processes: Concepts**
- Address spaces
- VM as a tool for memory management
- VM as a tool for memory protection
- VM as a tool for caching

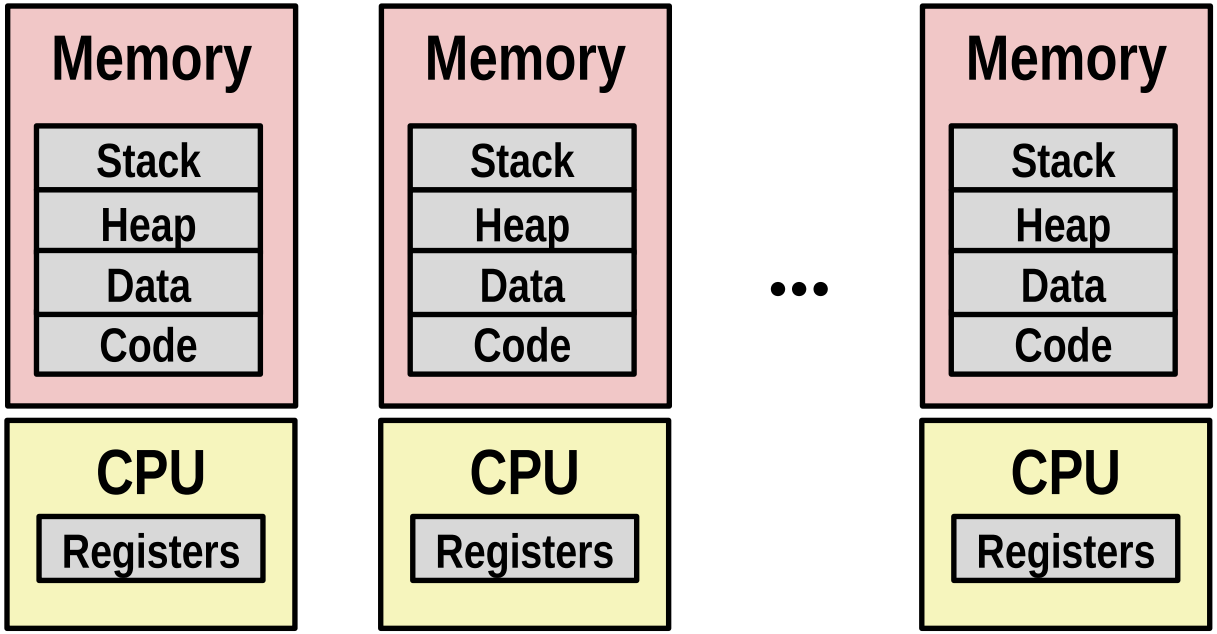
Processes

- **Definition: A *process* is an instance of a running program.**
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

- **Process provides each program with two key abstractions:**
  - *Logical control flow*
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel feature called *context switching*
  - *Private address space*
    - Each program seems to have exclusive use of main memory.
    - Provided by CPU feature called *virtual memory*

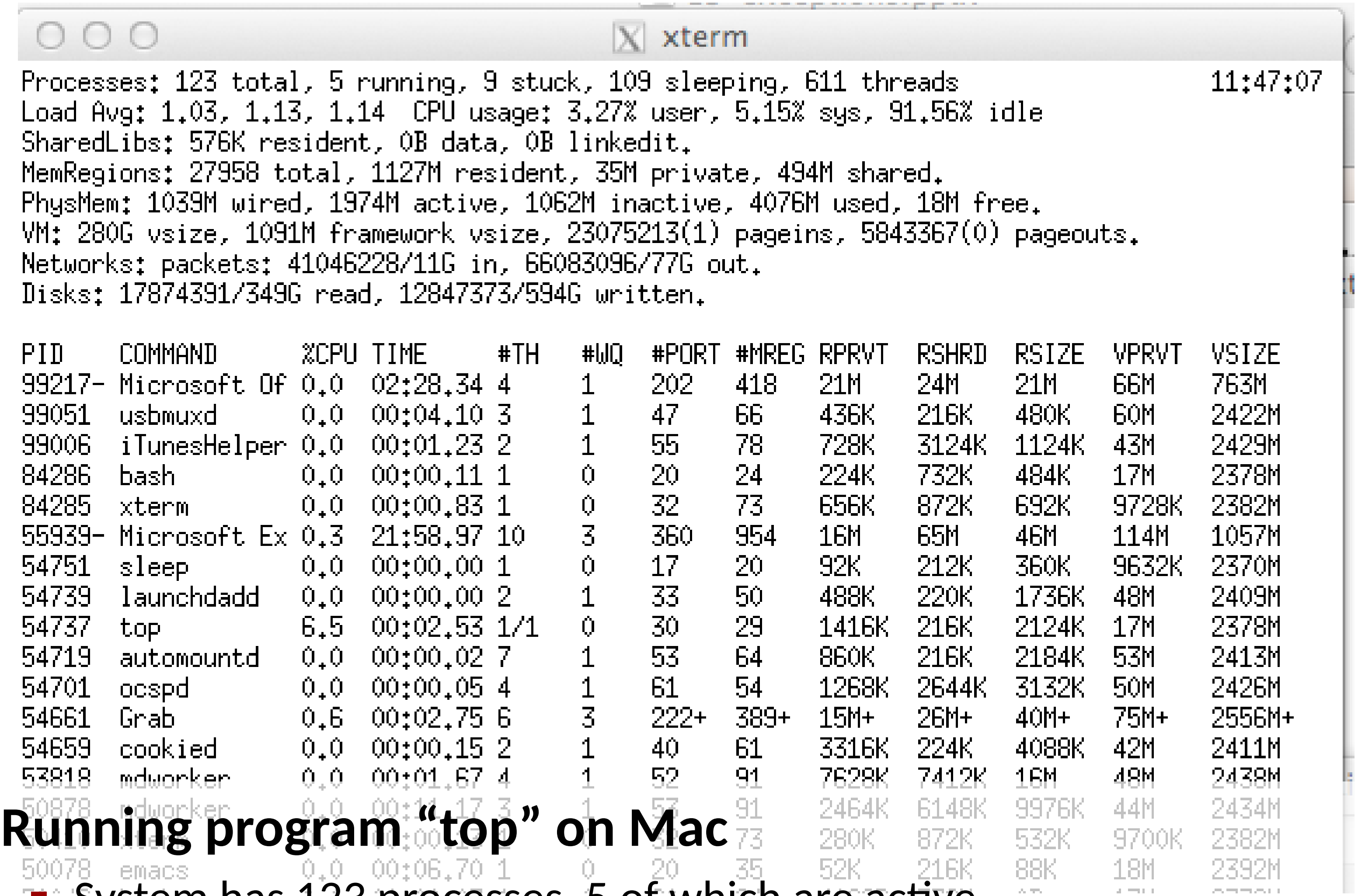


Multiprocessing: The Illusion



- **Computer runs many processes simultaneously**
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing Example



```
Processes: 123 total, 5 running, 9 stuck, 109 sleeping, 611 threads          11:47:07
Load Avg: 1.03, 1.13, 1.14  CPU usage: 3.27% user, 5.15% sys, 91.56% idle
SharedLibs: 576K resident, 0B data, 0B linkedit.
MemRegions: 27958 total, 1127M resident, 35M private, 494M shared.
PhysMem: 1039M wired, 1974M active, 1062M inactive, 4076M used, 18M free.
VM: 280G vsize, 1091M framework vsize, 23075213(1) pageins, 5843367(0) pageouts.
Networks: packets: 41046228/11G in, 66083096/77G out.
Disks: 17874391/349G read, 12847373/594G written.

PID    COMMAND      %CPU TIME     #TH   #WQ  #PORT #MREG RPRVT  RSHRD  RSIZE  VPRVT  VSIZE
99217- Microsoft Of 0.0  02:28.34 4     1    202   418   21M    24M    21M    66M    763M
99051  usbmuxd      0.0  00:04.10 3     1    47    66    436K   216K   480K   60M    2422M
99006  iTunesHelper 0.0  00:01.23 2     1    55    78    728K   3124K  1124K  43M    2429M
84286  bash         0.0  00:00.11 1     0    20    24    224K   732K   484K   17M    2378M
84285  xterm        0.0  00:00.83 1     0    32    73    656K   872K   692K   9728K  2382M
55939- Microsoft Ex 0.3  21:58.97 10    3    360   954   16M    65M    46M    114M   1057M
54751  sleep        0.0  00:00.00 1     0    17    20    92K    212K   360K   9632K  2370M
54739  launchdadd   0.0  00:00.00 2     1    33    50    488K   220K   1736K  48M    2409M
54737  top          6.5  00:02.53 1/1   0    30    29    1416K  216K   2124K  17M    2378M
54719  automountd   0.0  00:00.02 7     1    53    64    860K   216K   2184K  53M    2413M
54701  ocspd        0.0  00:00.05 4     1    61    54    1268K  2644K  3132K  50M    2426M
54661  Grab         0.6  00:02.75 6     3    222+  389+  15M+   26M+   40M+   75M+   2556M+
54659  cookied      0.0  00:00.15 2     1    40    61    3316K  224K   4088K  42M    2411M
53818  mdworker     0.0  00:01.67 4     1    52    91    7628K  7412K  16M    48M    2438M
50878  mdworker     0.0  00:[illegible].17 3     1    53    91    2464K  6148K  9976K  44M    2434M
[illegible]                                                73    280K   872K   532K   9700K  2382M
50078  emacs        0.0  00:06.70 1     0    20    35    52K    216K   88K    18M    2392M
```

- Running program “top” on Mac
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)

Preview: Creating and Terminating Processes

From a programmer's perspective, we can think of a process as being in one of three states

- **Running**
  - Process is executing (or waiting to, as we'll see next week)
- **Stopped**
  - Process execution is *suspended* until further notice (covered later)
- **Terminated**
  - Process is stopped permanently

Terminating Processes

- **Programmer can explicitly terminate process by:**
  - Returning from the **`main`** routine
  - Calling the **`exit`** function

- **`void exit(int status)`**
  - Terminates with an *exit status* of **`status`**
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

- **`exit` is called once but never returns.**

Creating Processes

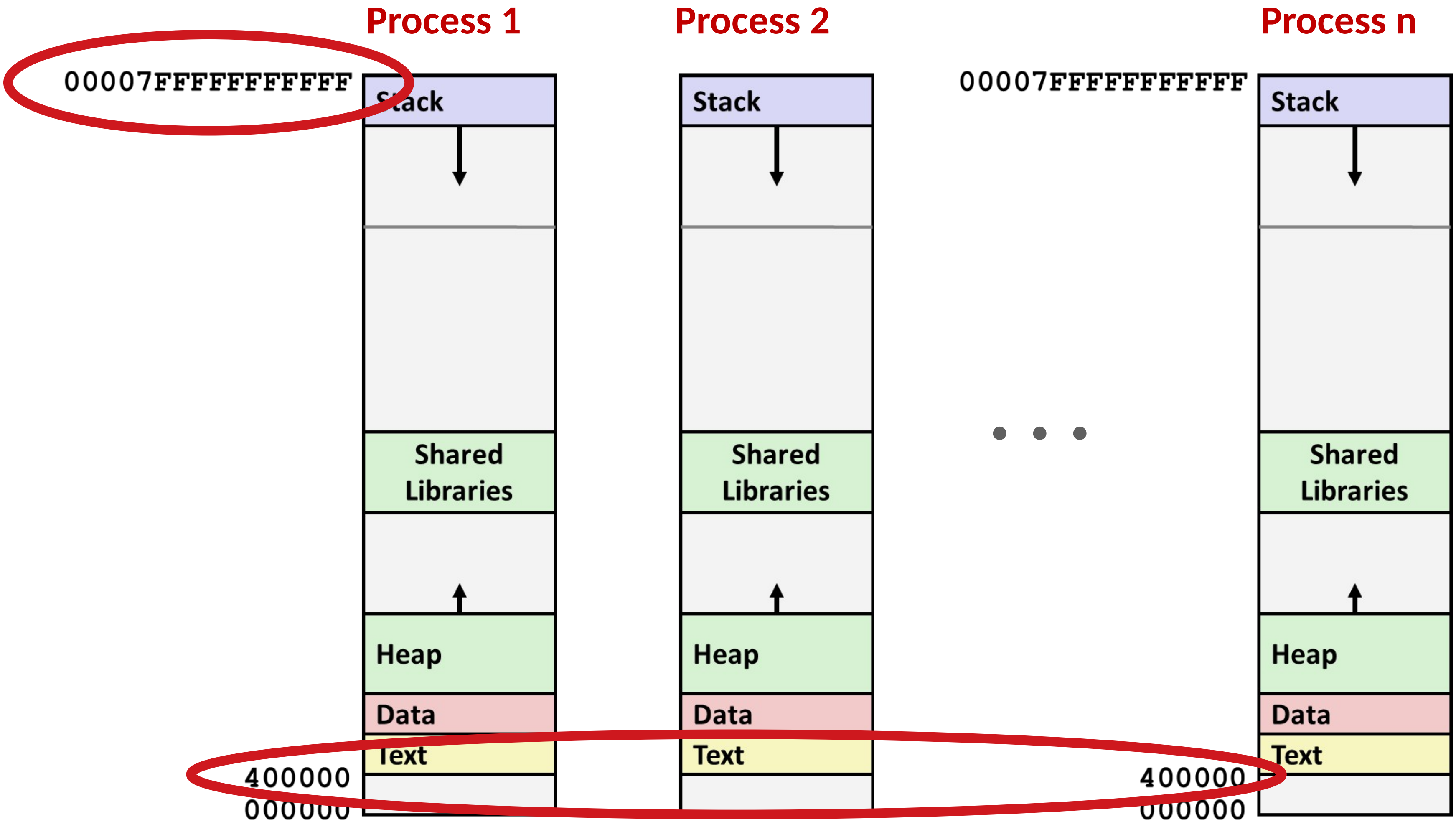
- ***Parent process*** **creates a new running** ***child process*** **by calling `fork`**

- **`int fork(void)`**
  - Returns 0 to the child process, child's PID to parent process
  - Child is *almost* identical to parent…

Different how?

- **`fork` is interesting (and often confusing) because it is called** ***once*** **but returns** ***twice***

Hmmm, How Does This Work?!



Solution: Virtual Memory (today and next lecture)

Creating Processes

- ***Parent process* creates a new running *child process* by calling `fork`**

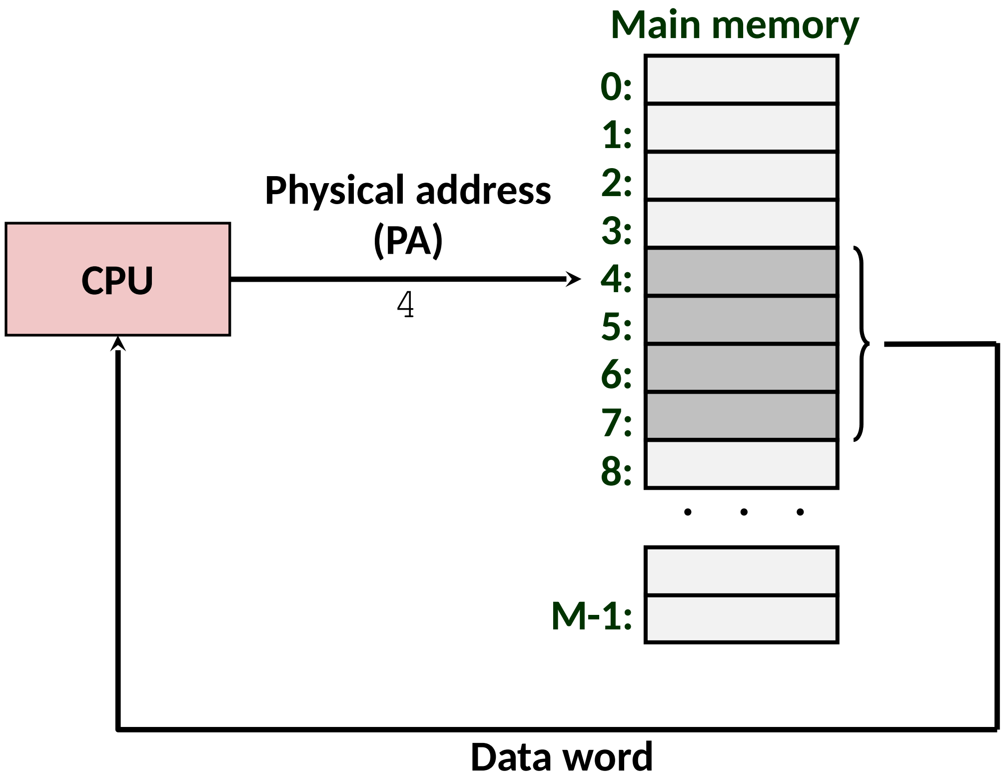
- **`int fork(void)`**
  - Returns 0 to the child process, child's PID to parent process
  - Child is *almost* identical to parent:
    - Child get an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent

- **`fork` is interesting (and often confusing) because it is called *once* but returns *twice***

Today

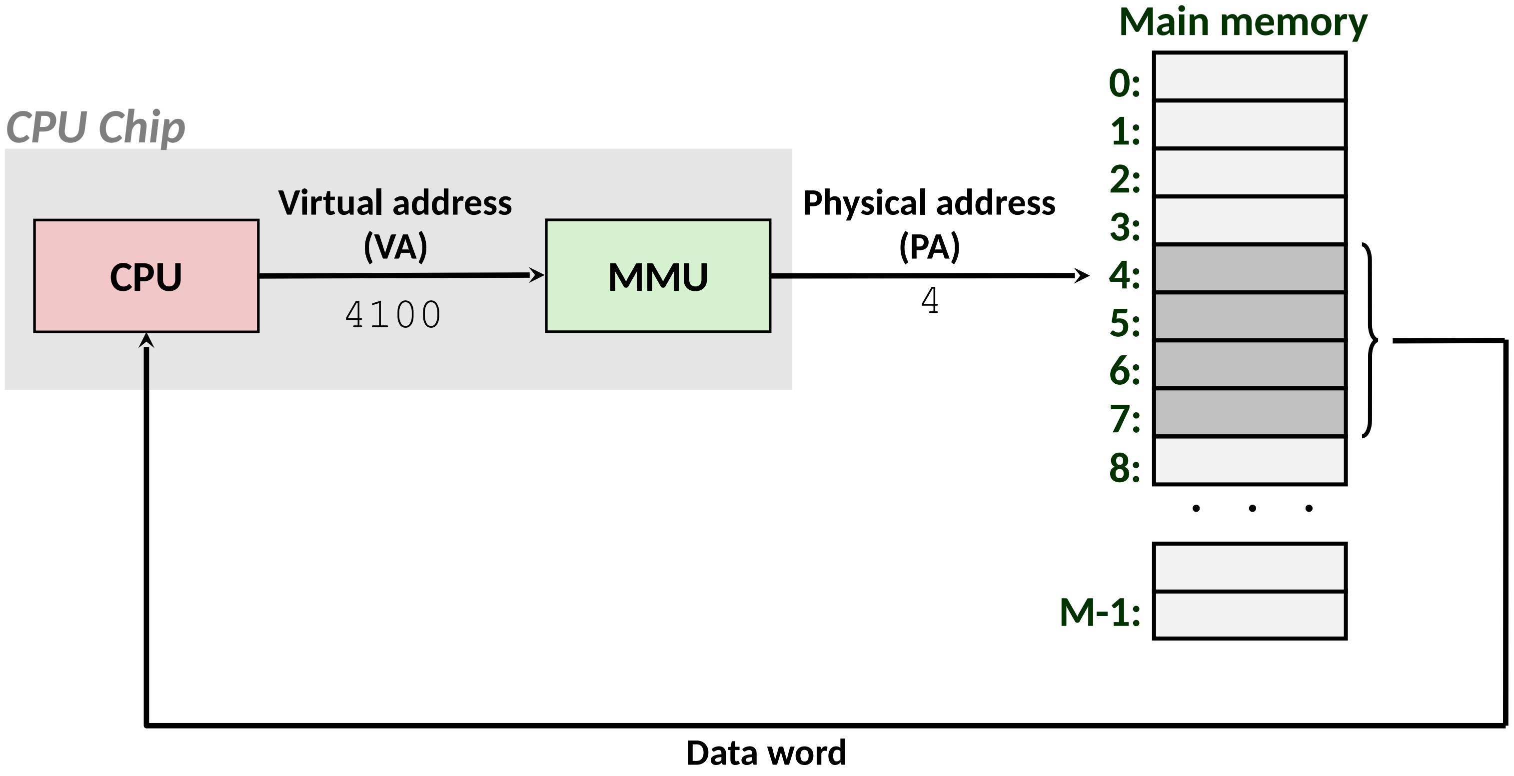
- Processes: Concepts
- **Address spaces**
- VM as a tool for memory management
- VM as a tool for memory protection
- VM as a tool for caching

A System Using Physical Addressing



- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing



- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science

Address Spaces

- **Linear address space:** Ordered set of contiguous non-negative integer addresses:
  {0, 1, 2, 3 … }

- **Virtual address space:** Set of N = 2^n virtual addresses
  {0, 1, 2, 3, …, N-1}

- **Physical address space:** Set of M = 2^m physical addresses
  {0, 1, 2, 3, …, M-1}

Why Virtual Memory (VM)?

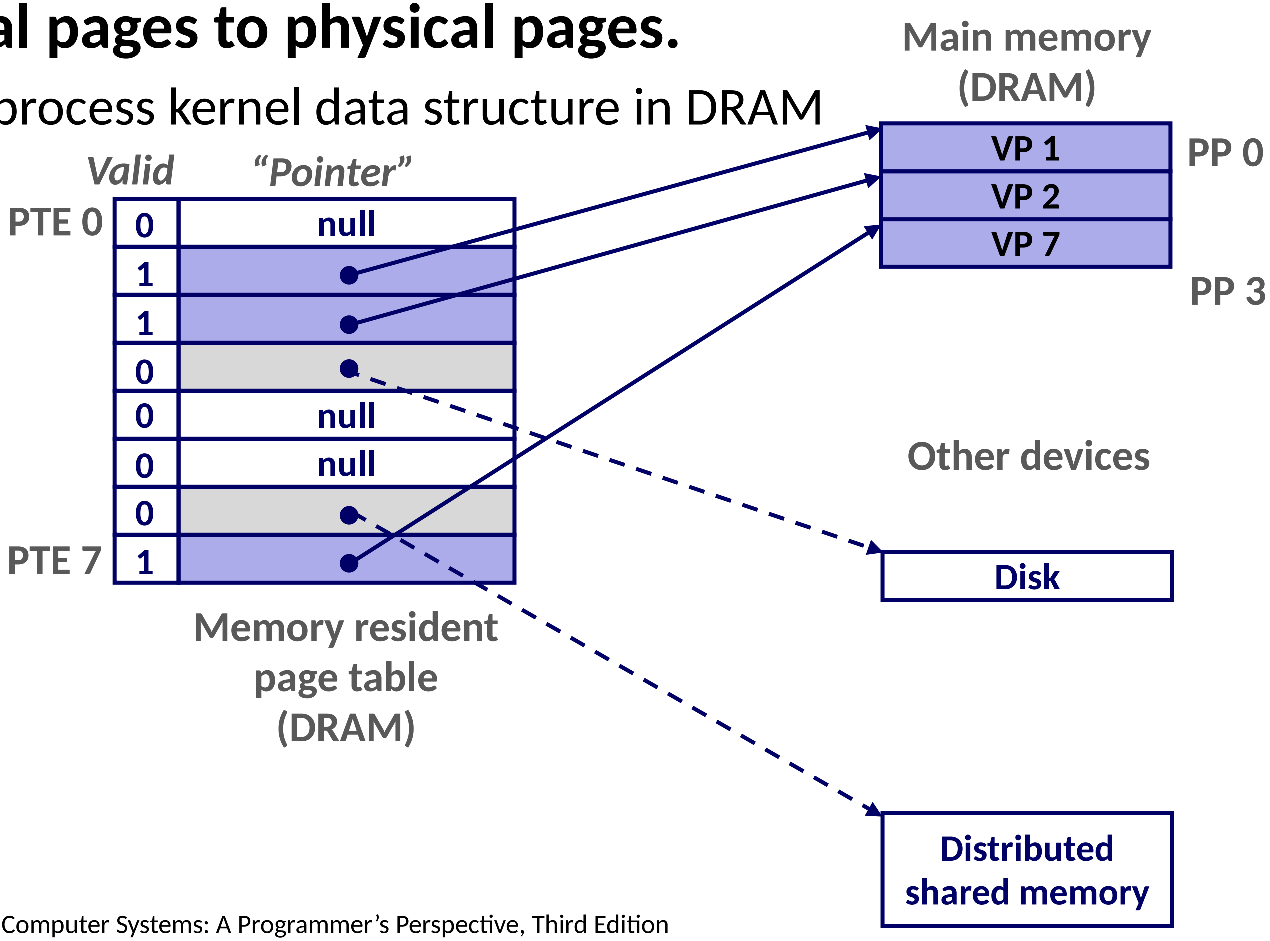
- **Simplifies memory management**
  - Each process gets its own private address space

- **Isolates address spaces**
  - One process can't interfere with another's memory
  - User program cannot access privileged kernel information and code

- **Allows addressing locations outside DRAM**
  - Programs can access "memory" to communicate with other devices
  - The kernel can handle such accesses in software

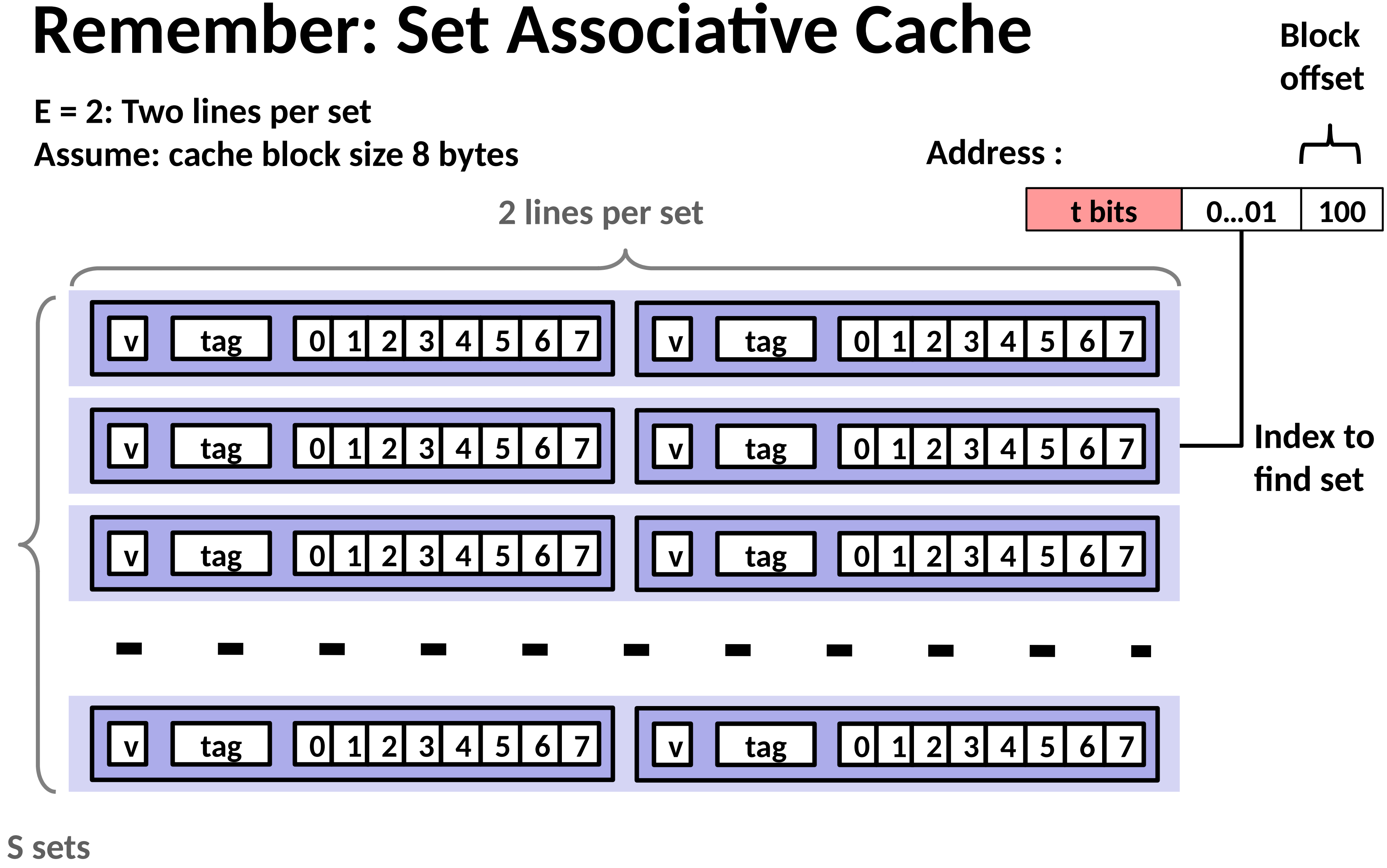
Paging: Pages and Page Tables

- A ***page*** is the *aligned* unit at which mapping is customized
  - Typically 4 KB on modern systems
- A ***page table*** is an array of page table entries (PTEs) that maps virtual pages to physical pages.
  - Per-process kernel data structure in DRAM

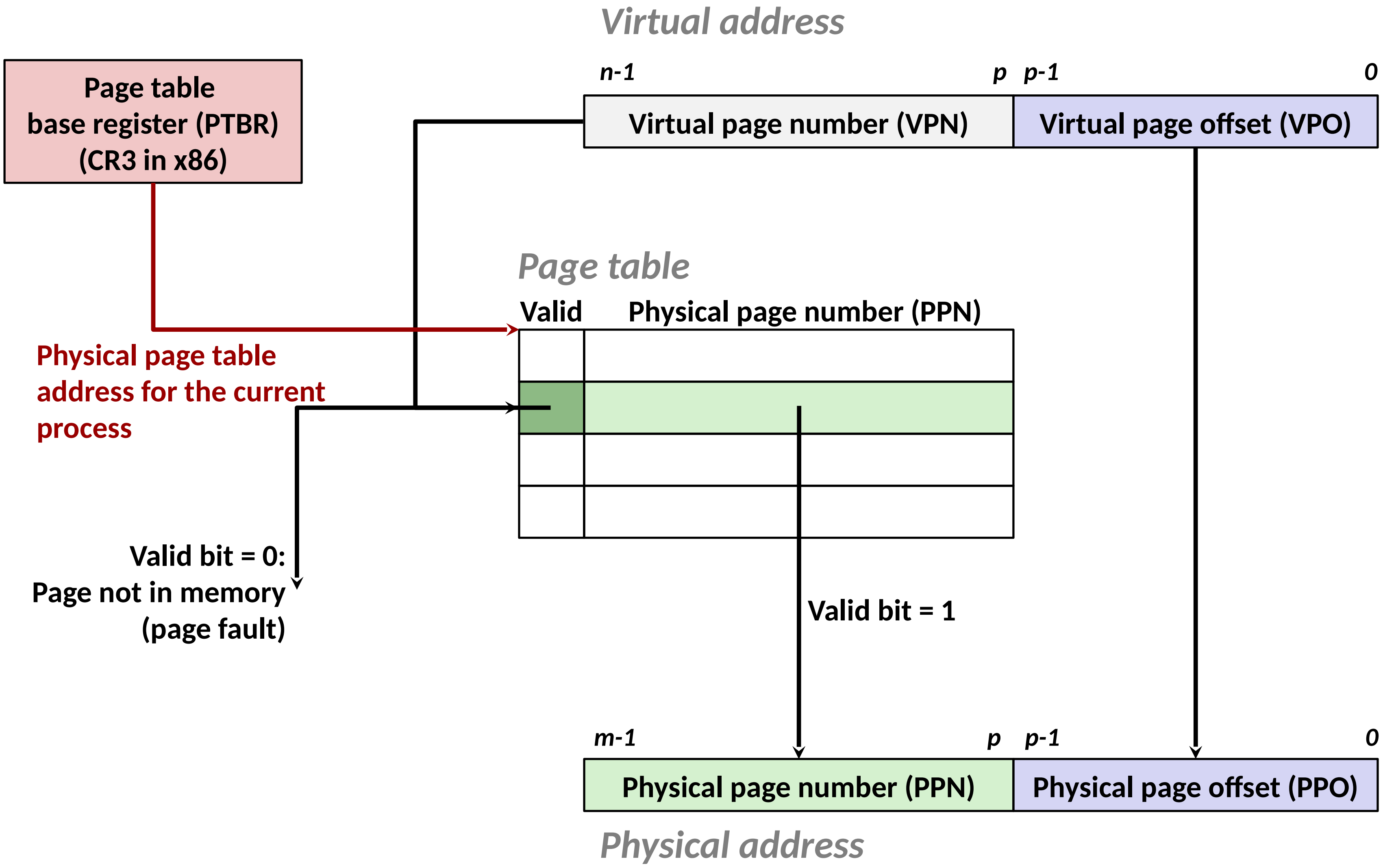


Remember: Set Associative Cache

E = 2: Two lines per set
Assume: cache block size 8 bytes



Preview: Address Translation



Admission of Guilt

- **Lie: “Memory can be viewed as an array of bytes”…**
  - Actually discontinuous, with unmapped regions

- **Lie: “Memory addresses refer to locations in RAM”…**
  - Programmer sees only *virtual* addresses, which CPU’s MMU translates to *physical* addresses before sending them to the memory controller

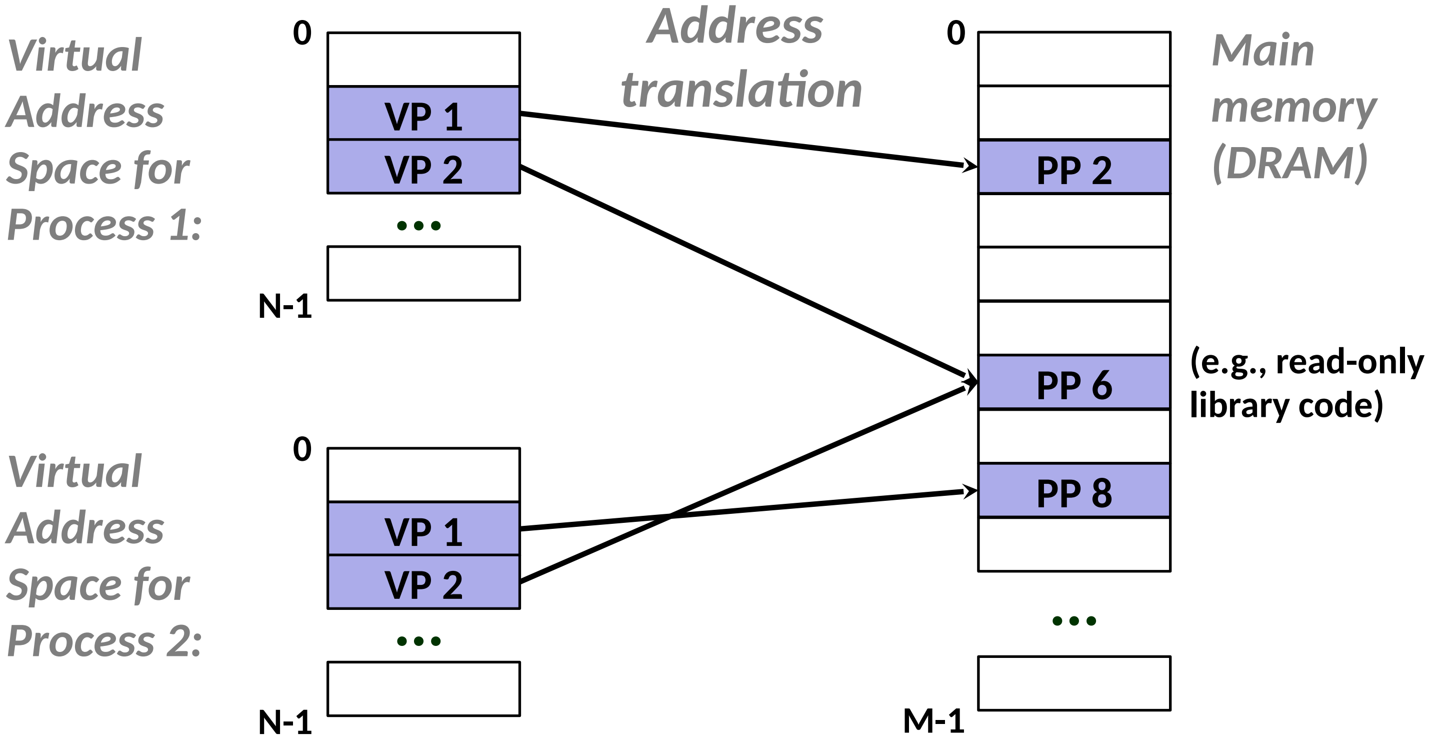
- **Lie: “Memory addresses are 64 bits”…**
  - Current x86-64 CPU MMUs only support 48-bit virtual addresses, which is enough to address 256 TB of RAM
  - Future CPUs may widen this without a change to the ISA

Today

- Processes: Concepts
- Address spaces
- **VM as a tool for memory management**
- VM as a tool for memory protection
- VM as a tool for caching

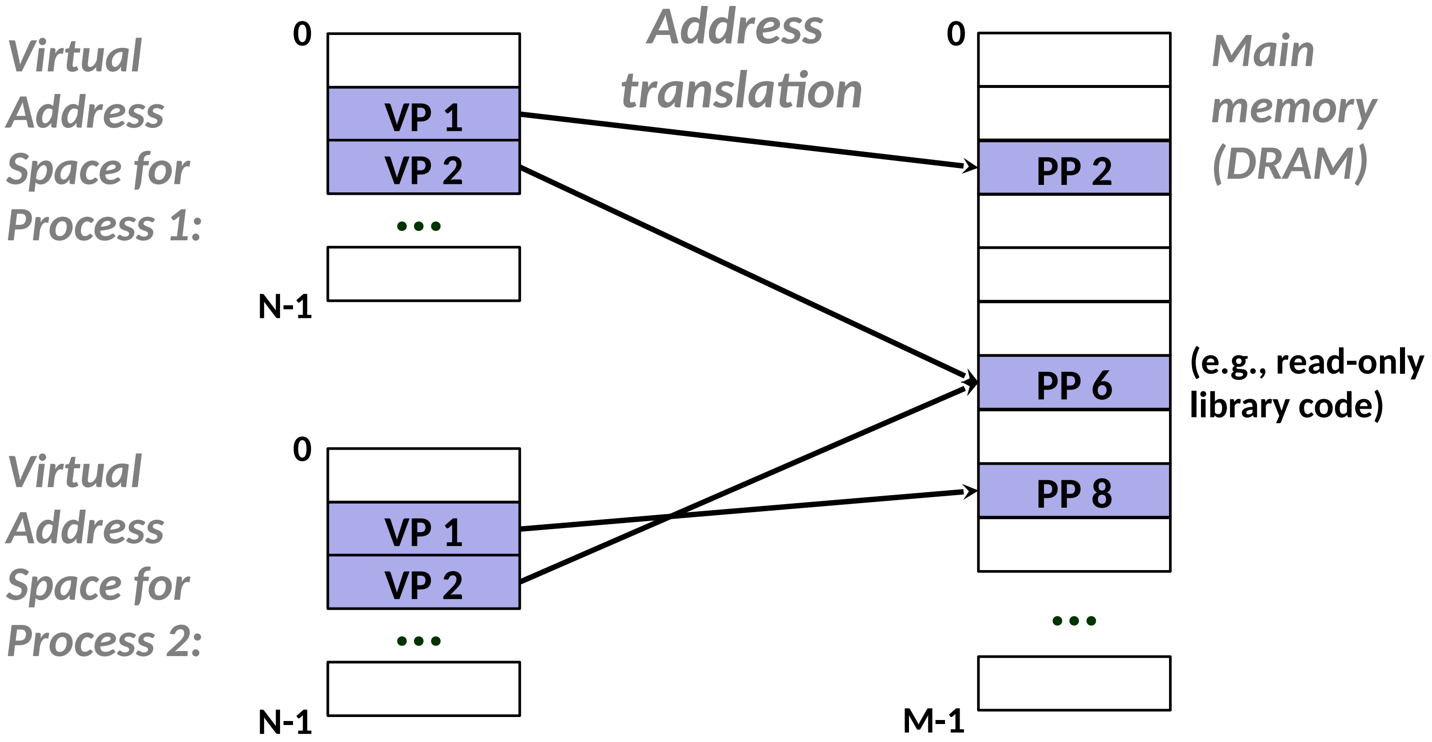
VM as a Tool for Memory Management

- **Key idea: each process has its own virtual address space**
  - Mapping function scatters addresses through physical memory
  - Process only knows about virtual addresses, so mappings can change

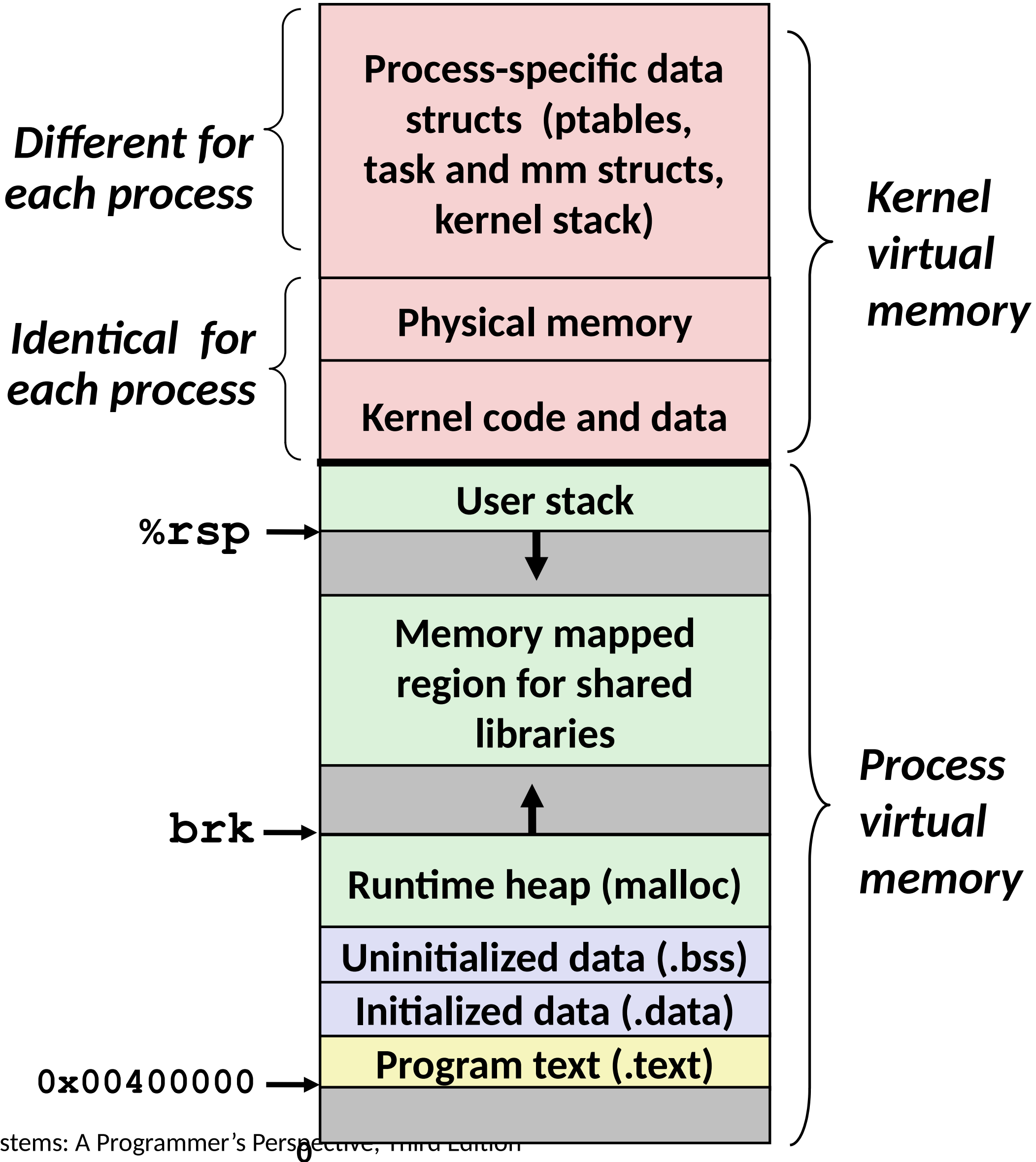


VM as a Tool for Memory Management

- **Simplifying memory allocation**
  - Each virtual page can be mapped to any physical page
  - A virtual page can be stored in different physical pages at different times
- **Sharing code and data among processes**
  - Map virtual pages to the same physical page (here: PP 6)

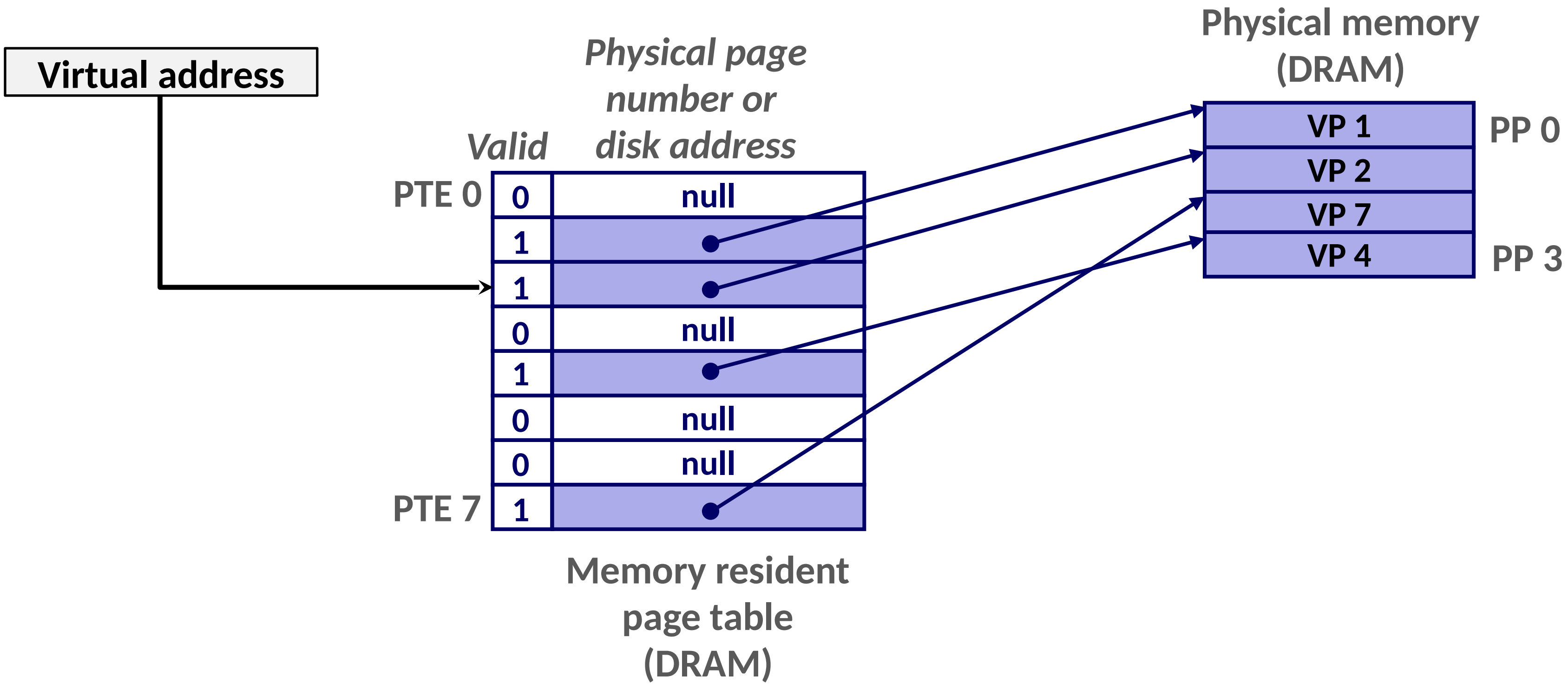


Virtual Address Space of a Linux Process



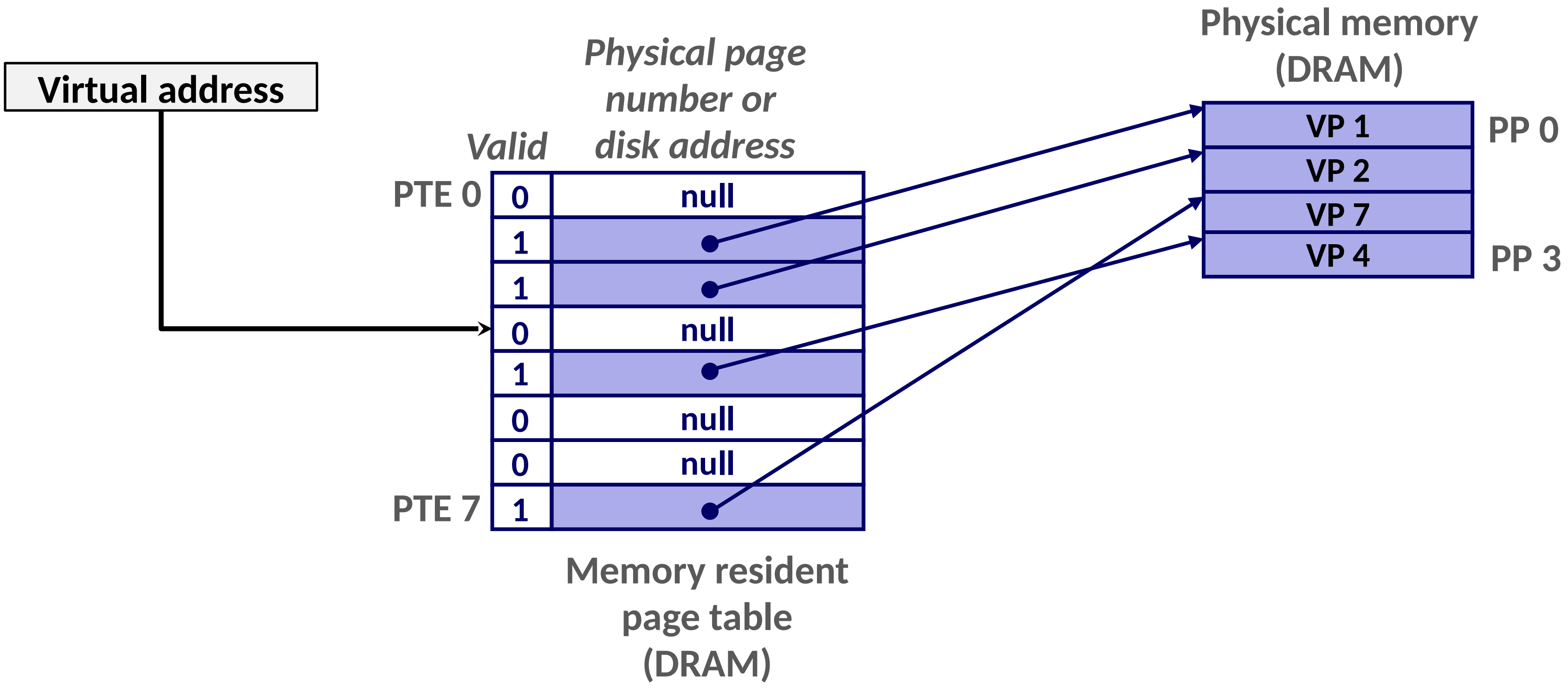
Page Hit

- ***Page hit:*** **reference to page that is in physical memory**



Page Fault

- *Page fault:* **reference to page that is not in physical memory**

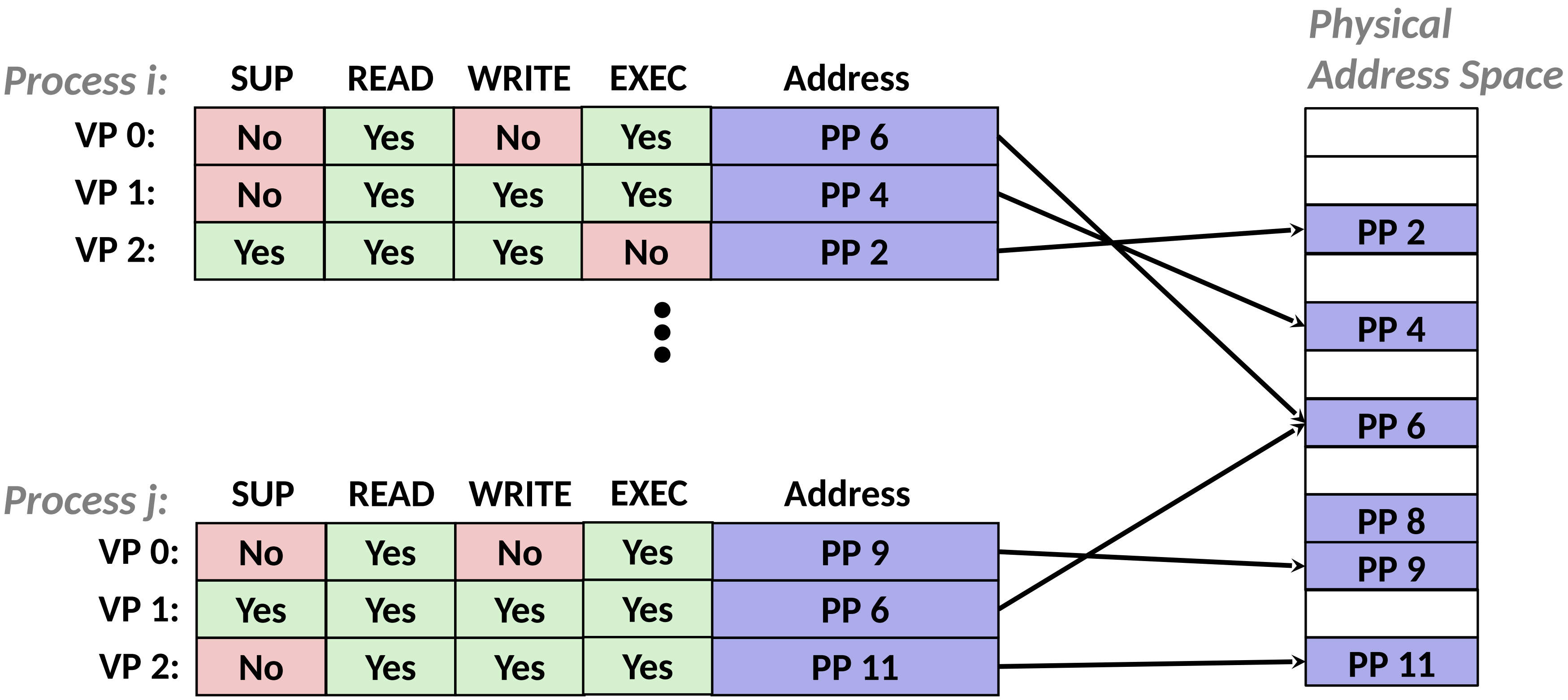


Today

- Processes: Concepts
- Address spaces
- VM as a tool for memory management
- **VM as a tool for memory protection**
- VM as a tool for caching

VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- MMU checks these bits on each access



Process i:

	SUP	READ	WRITE	EXEC	Address
VP 0:	No	Yes	No	Yes	PP 6
VP 1:	No	Yes	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	No	PP 2

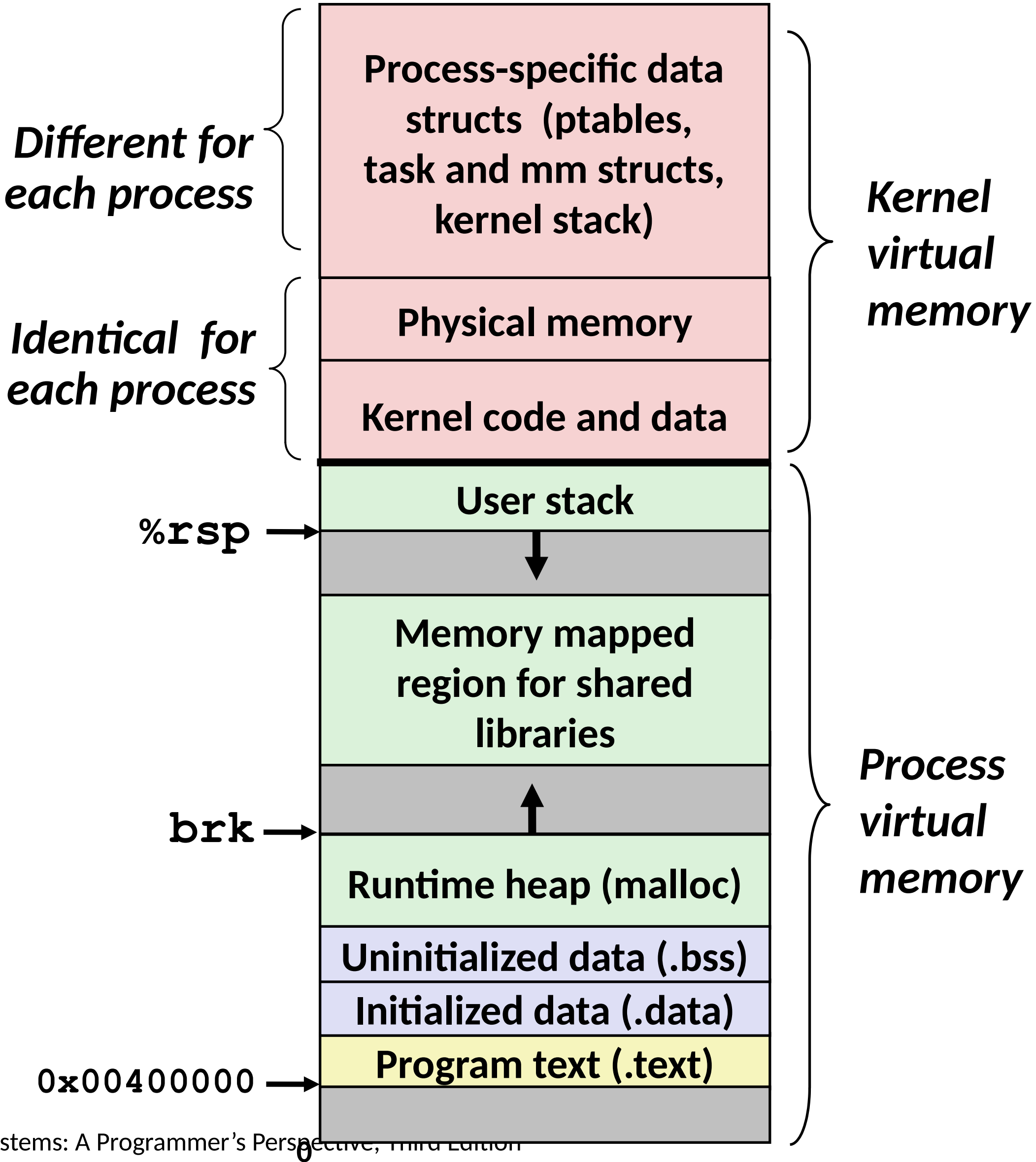
⋮

Process j:

	SUP	READ	WRITE	EXEC	Address
VP 0:	No	Yes	No	Yes	PP 9
VP 1:	Yes	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	Yes	PP 11

Physical Address Space

Virtual Address Space of a Linux Process



Different for each process

Process-specific data structs (ptables, task and mm structs, kernel stack)

Identical for each process

Physical memory

Kernel code and data

Kernel virtual memory

User stack

`%rsp`

Memory mapped region for shared libraries

`brk`

Runtime heap (malloc)

Uninitialized data (.bss)

Initialized data (.data)

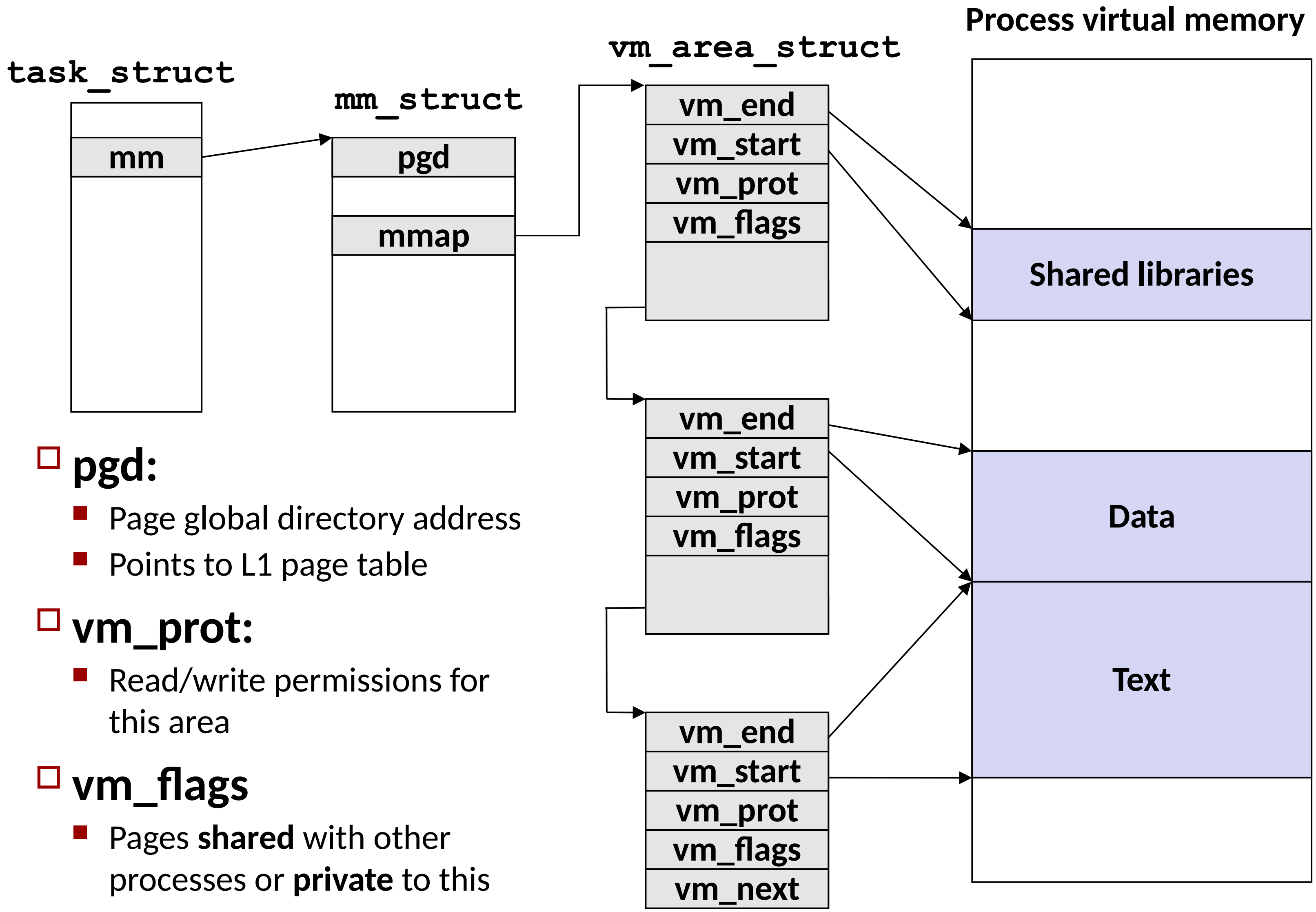
Program text (.text)

`0x00400000`

`0`

Process virtual memory

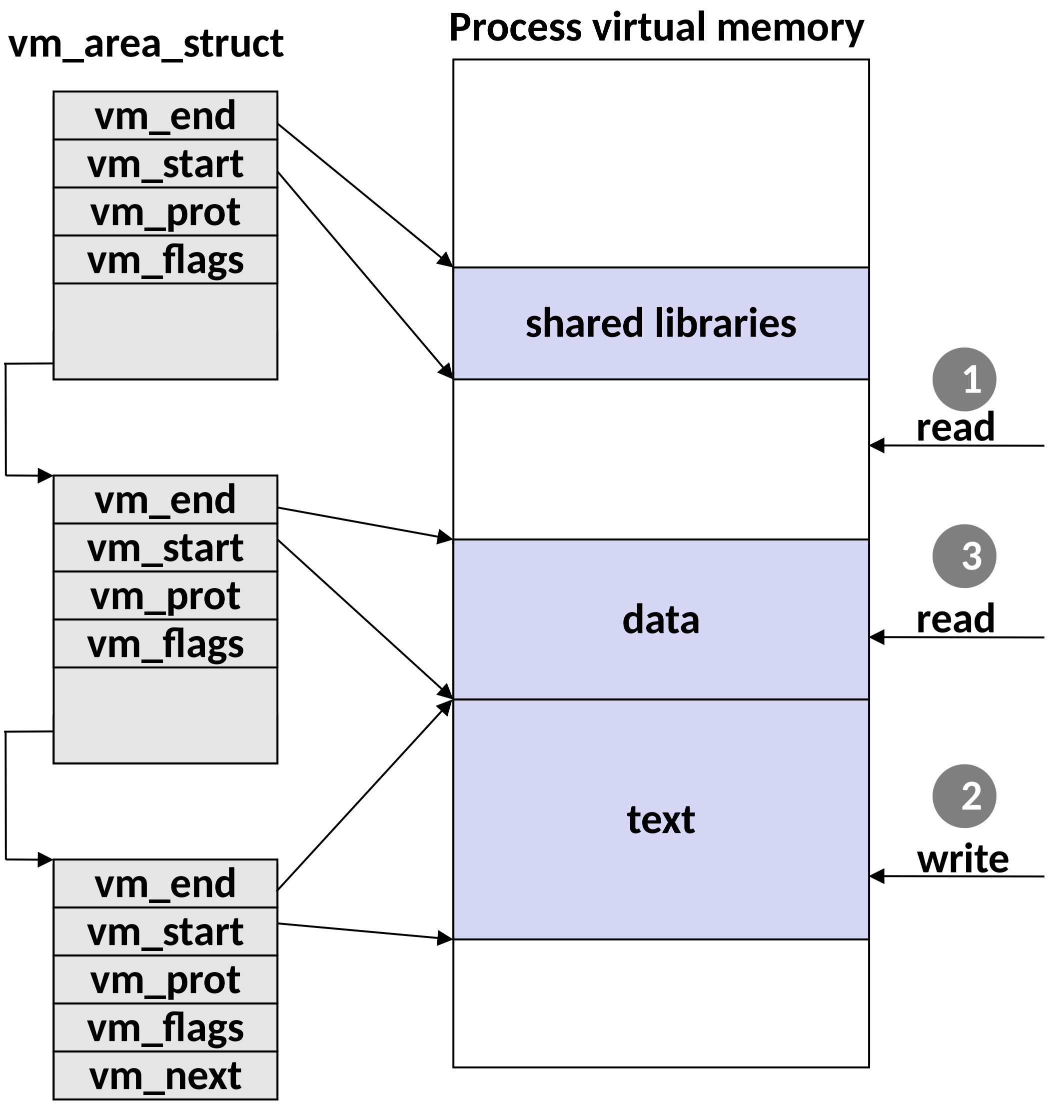
Linux Organizes VM as Collection of "Areas"



- **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

Each process has own `task_struct`, etc

Linux Page Fault Handling



Segmentation fault:
accessing a non-existing page

***Normal* page fault …?!**

Protection exception:
e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

Today

- Processes: Concepts
- Address spaces
- VM as a tool for memory management
- VM as a tool for memory protection
- **VM as a tool for caching**

Caching… as in a cache like this, right?

- No! Doesn’t work like a CPU cache.
- ***Cache:*** A smaller, faster storage… staging area.

2 lines per set

S sets

Address :

Block offset

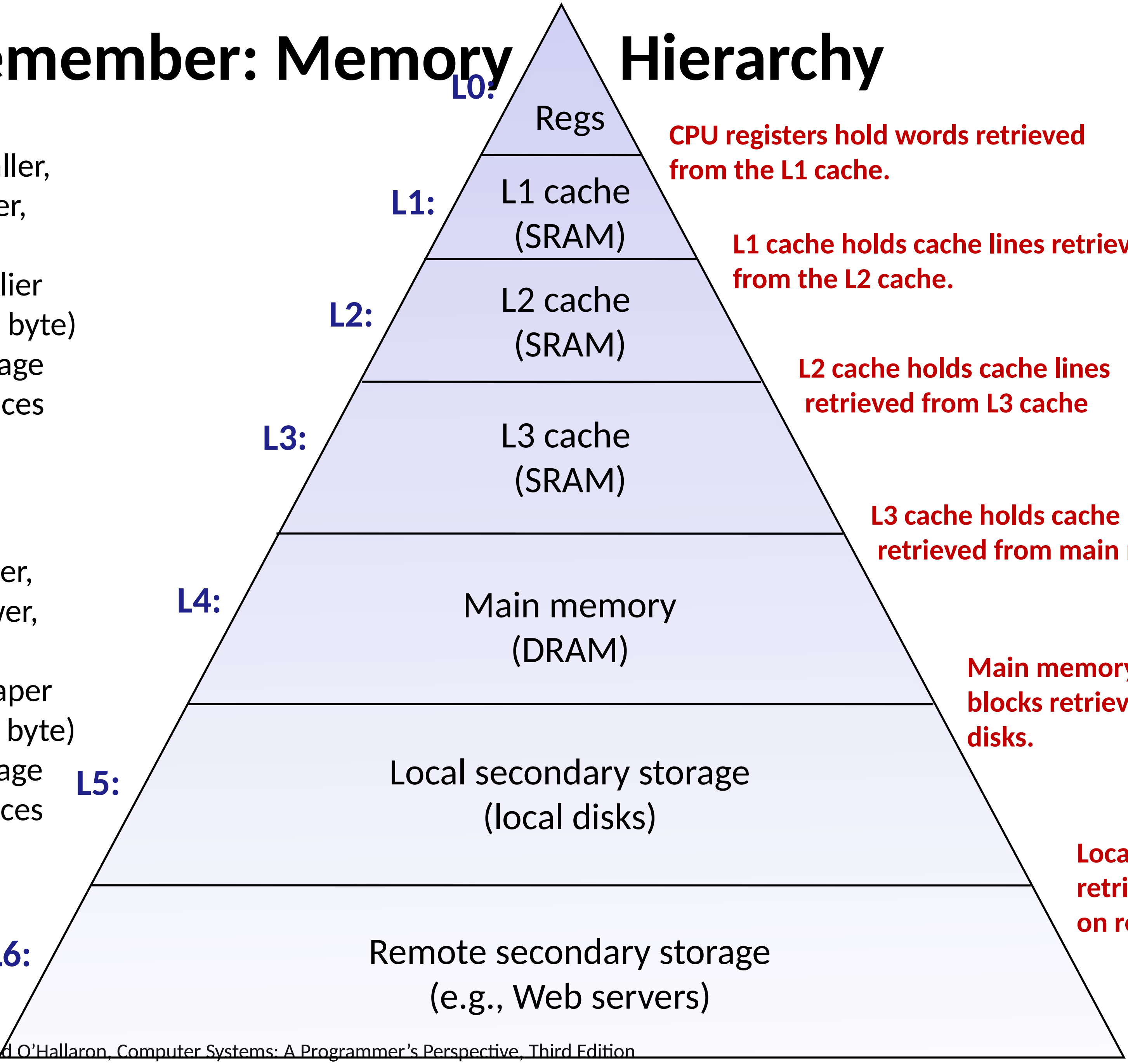
t bits	0…01	100

Index to find set

Remember: Memory Hierarchy

Smaller,
faster,
and
costlier
(per byte)
storage
devices

Larger,
slower,
and
cheaper
(per byte)
storage
devices



CPU registers hold words retrieved from the L1 cache.

L1 cache holds cache lines retrieved from the L2 cache.

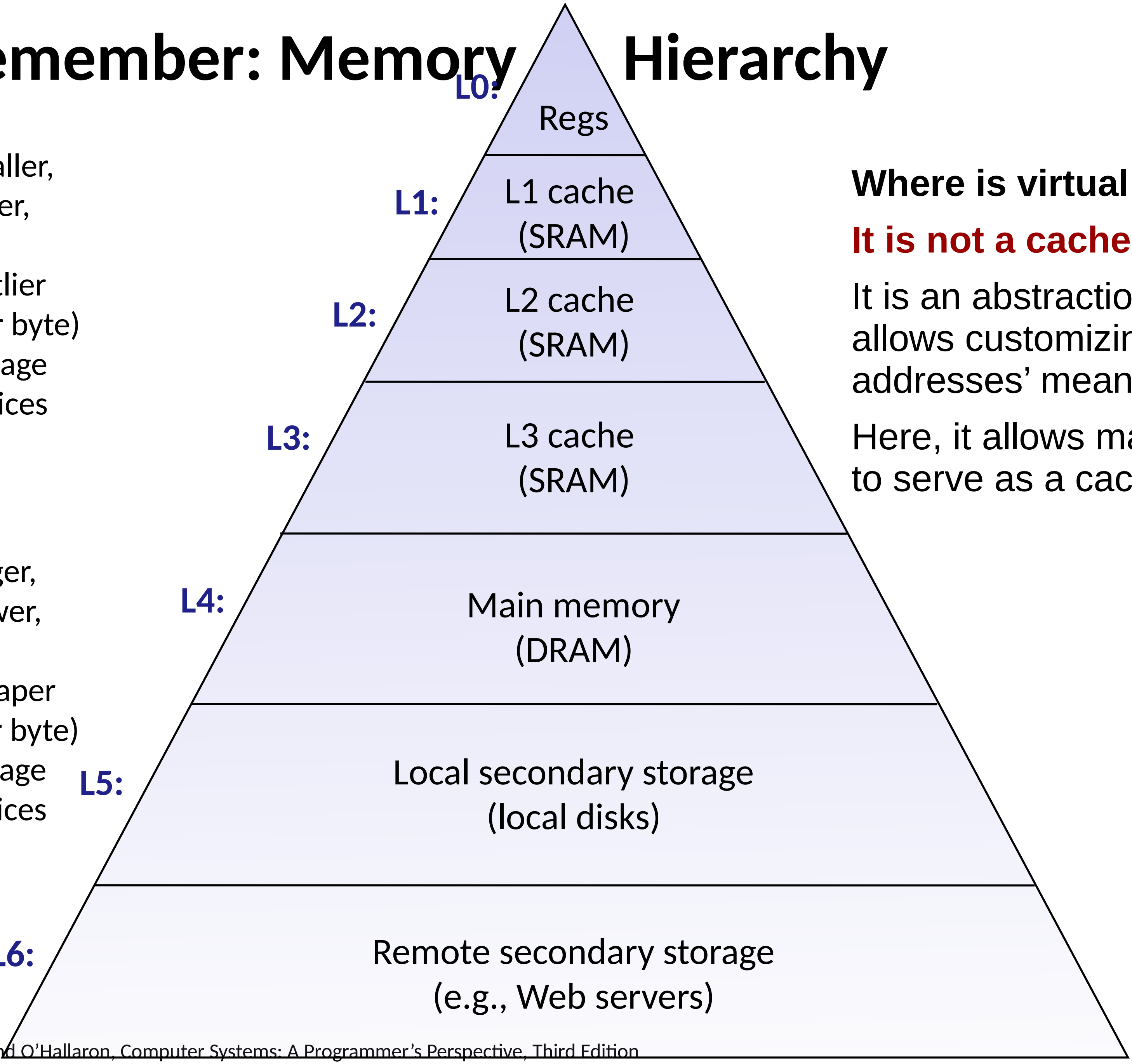
L2 cache holds cache lines retrieved from L3 cache

L3 cache holds cache lines retrieved from main memory.

Main memory holds disk blocks retrieved from local disks.

Local disks hold files retrieved from disks on remote servers

Remember: Memory Hierarchy



Smaller,
faster,
and
costlier
(per byte)
storage
devices

L0: Regs

L1: L1 cache (SRAM)

L2: L2 cache (SRAM)

L3: L3 cache (SRAM)

L4: Main memory (DRAM)

L5: Local secondary storage (local disks)

L6: Remote secondary storage (e.g., Web servers)

Larger,
slower,
and
cheaper
(per byte)
storage
devices

Where is virtual memory?!

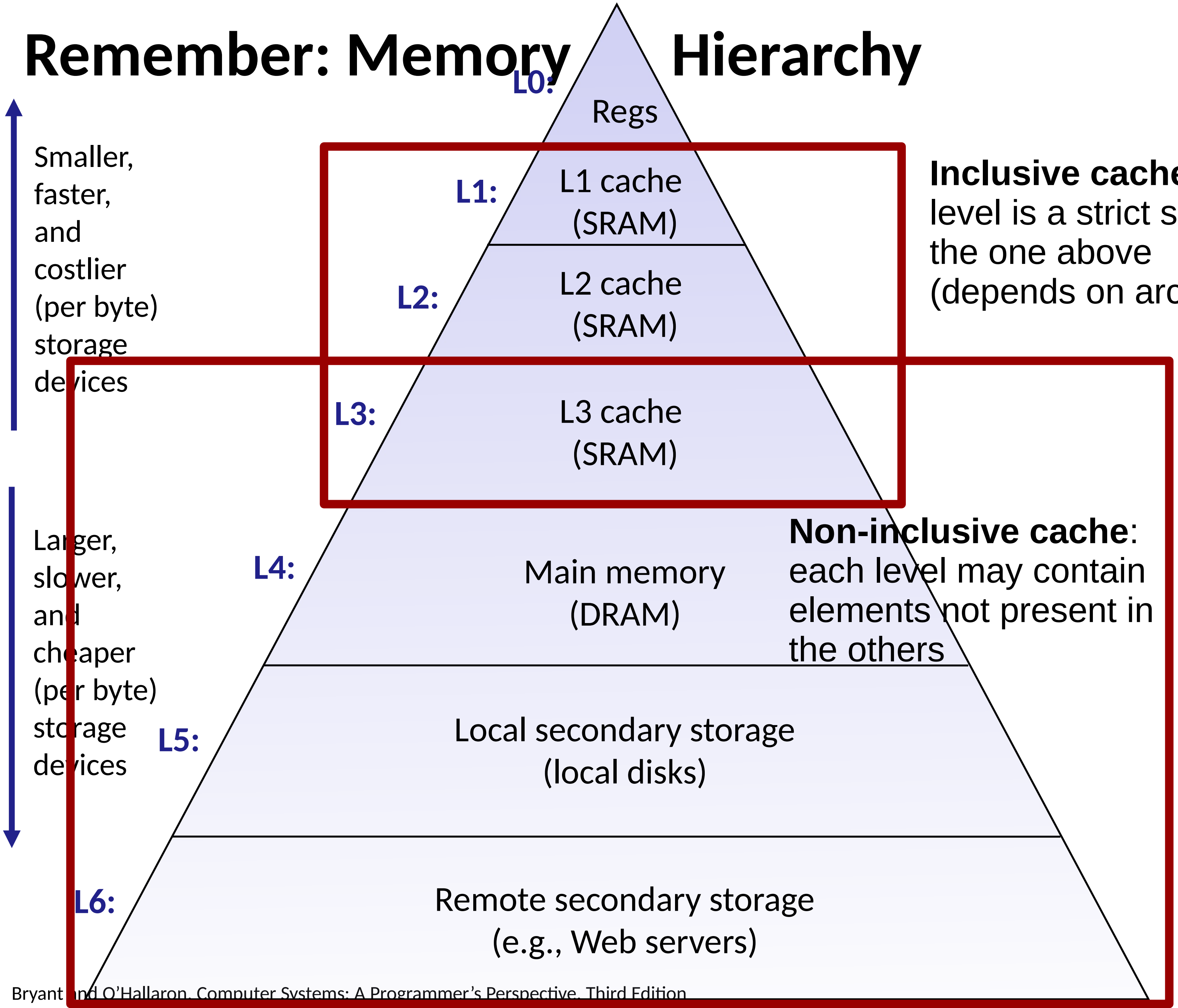
It is not a cache!

It is an abstraction that allows customizing memory addresses' meanings.

Here, it allows main memory to serve as a cache for disk.

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Remember: Memory Hierarchy



Smaller,
faster,
and
costlier
(per byte)
storage
devices

Larger,
slower,
and
cheaper
(per byte)
storage
devices

Inclusive cache: each level is a strict superset of the one above (depends on architecture)

Non-inclusive cache: each level may contain elements not present in the others

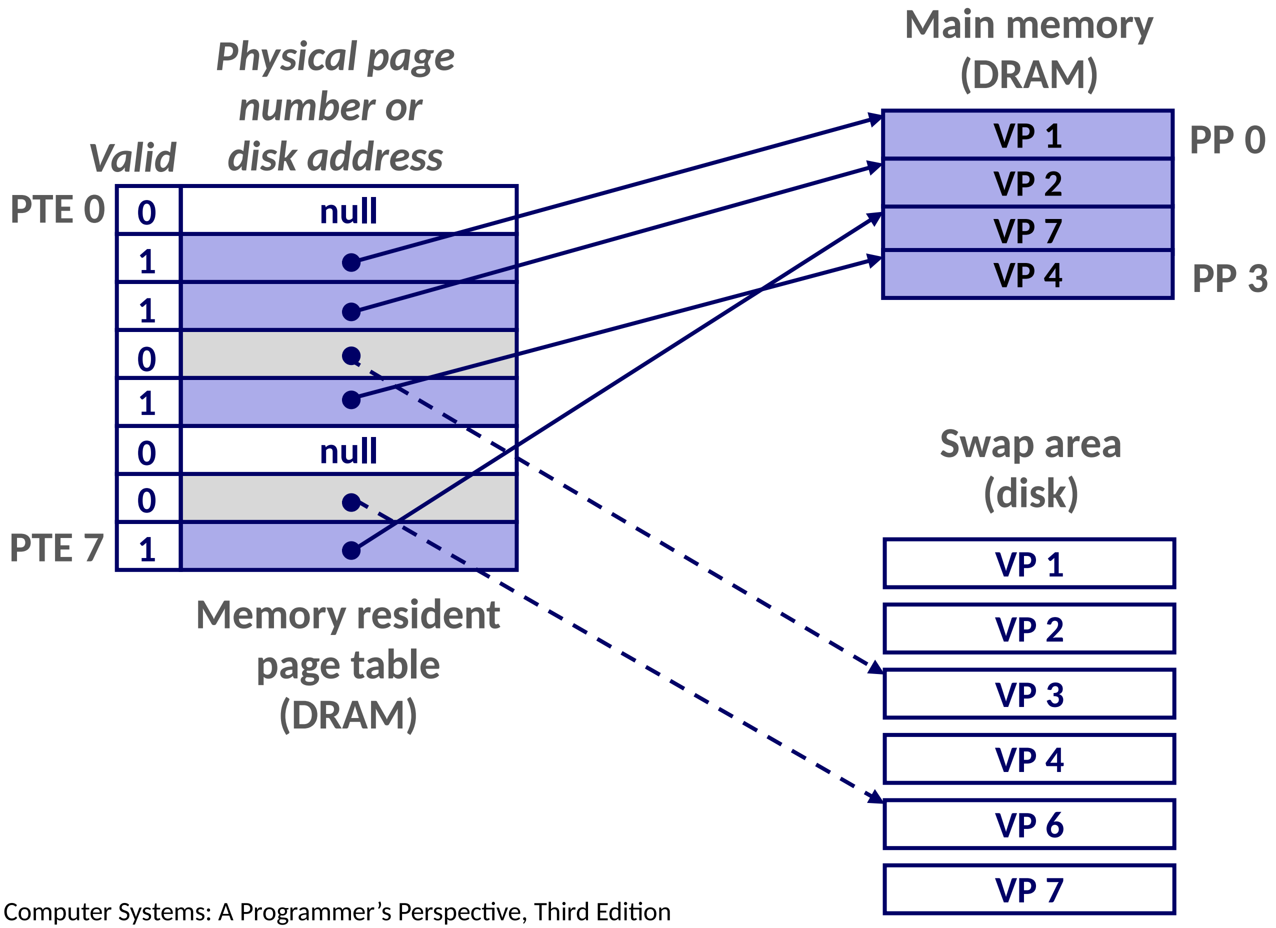
DRAM Cache Organization

- **DRAM cache organization driven by the enormous miss penalty**
  - DRAM is about ***10x*** slower than SRAM
  - Disk is about ***10,000x*** slower than DRAM

- **Consequences**
  - Large page (block) size: typically 4 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a “large” mapping function – different from cache memories
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through

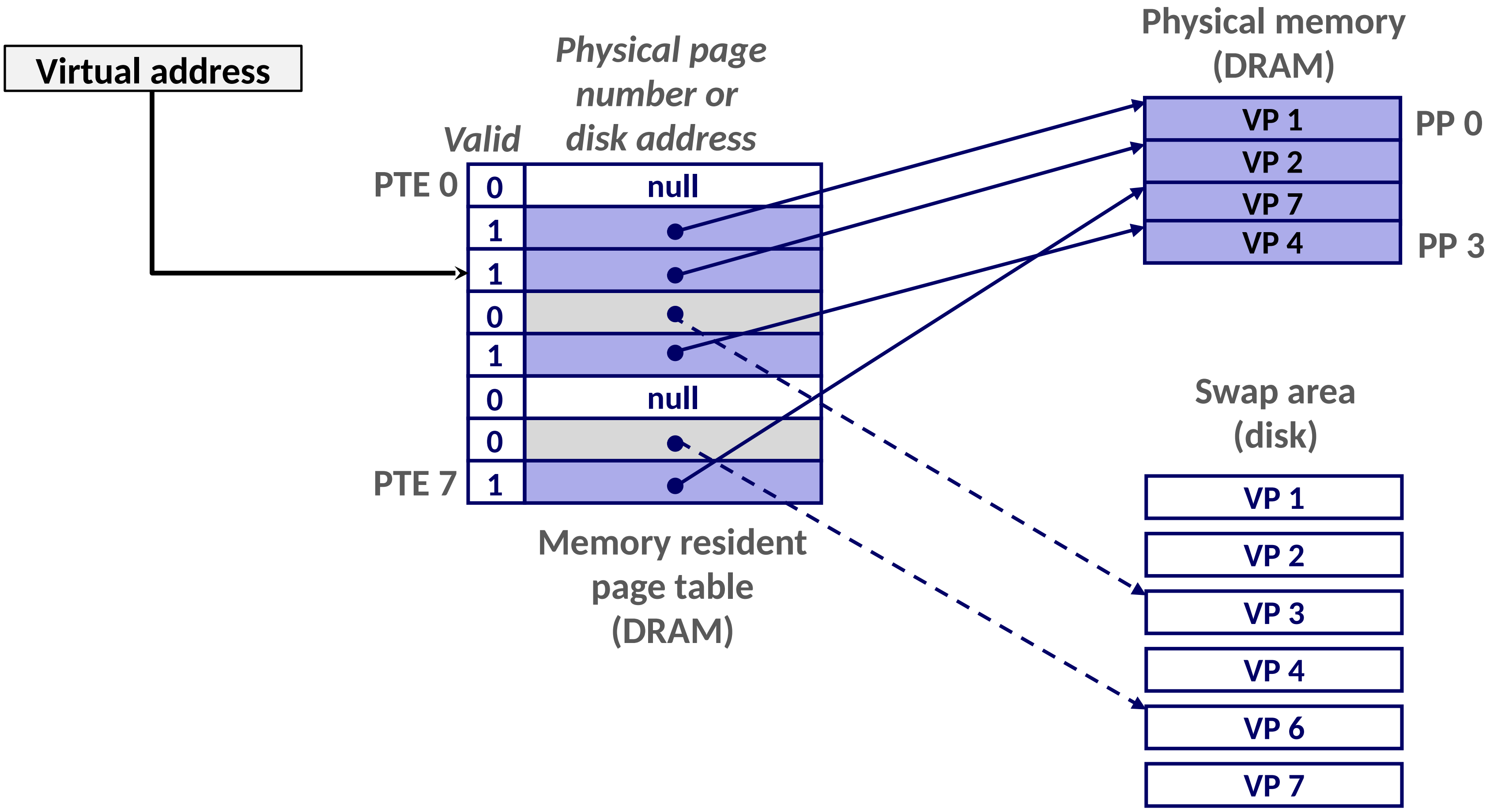
Paging: Once More w/ Feeling—err, swap

- A ***swap area*** **is an on-disk "overflow scratch space"**
  - When running out of DRAM, the operating system can move pages here instead of crashing.



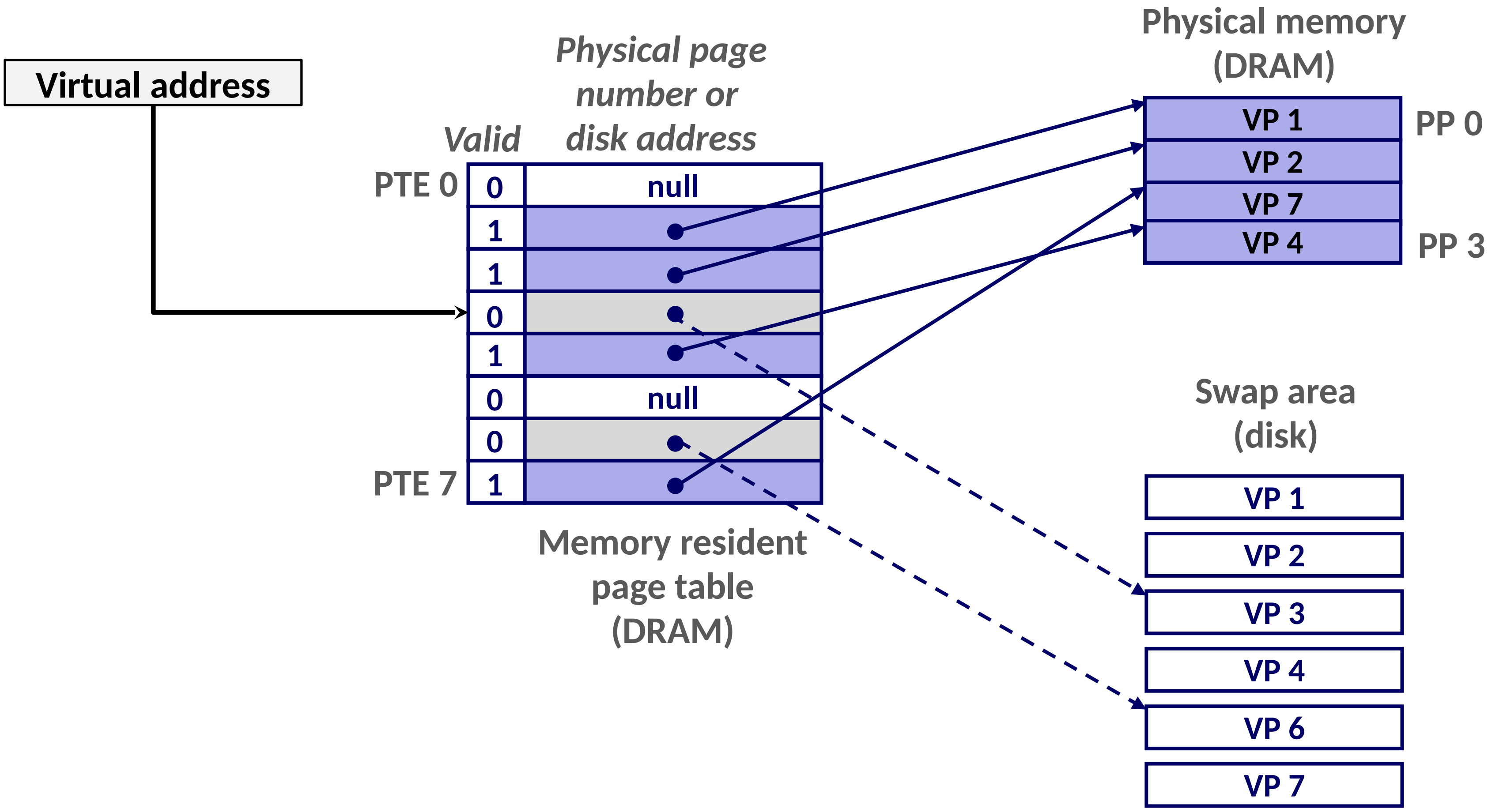
Page Hit

- ***Page hit:*** **in some ways like a DRAM “cache hit”**



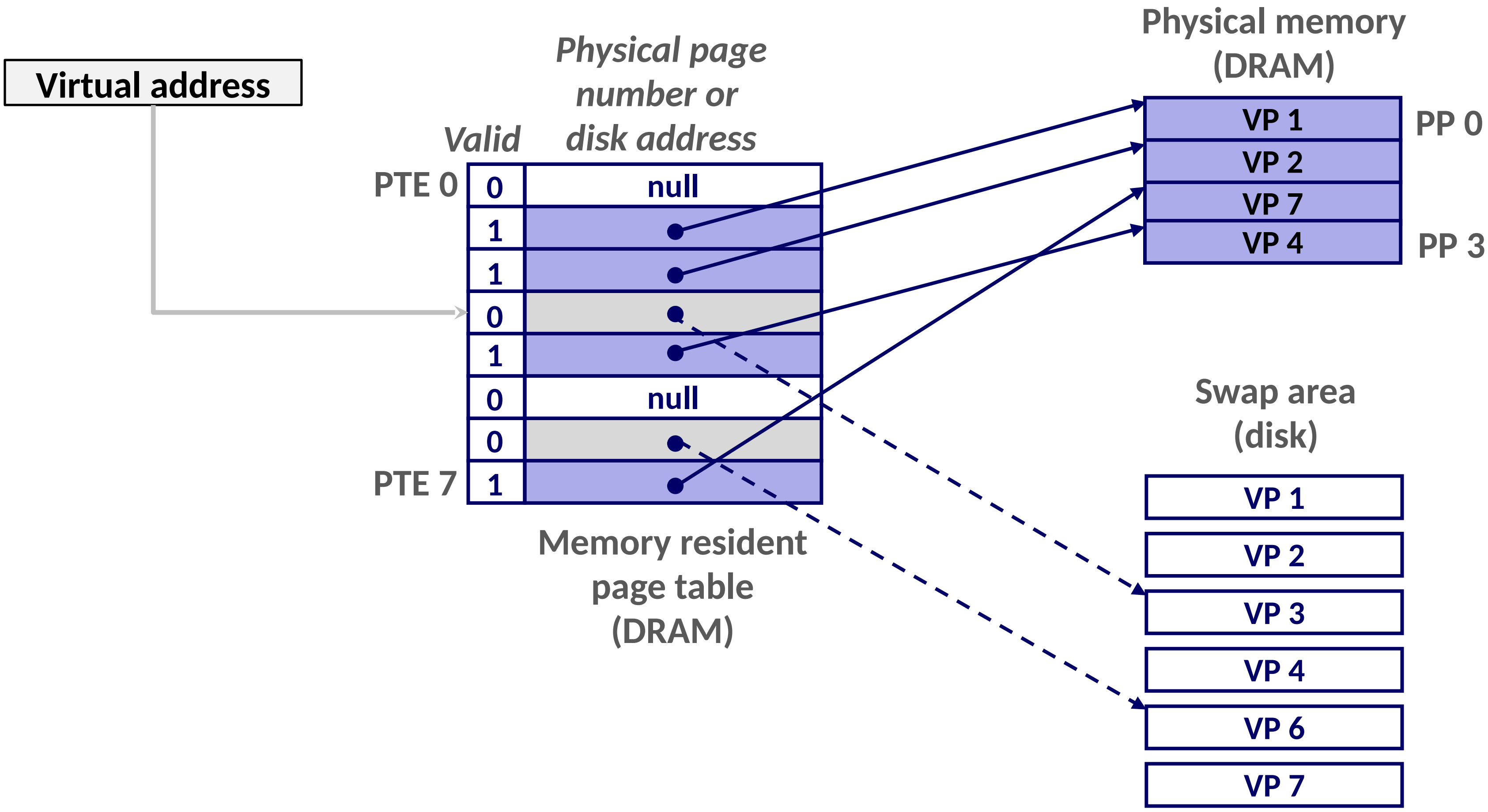
Page Fault

- *Page fault:* in some ways like a DRAM "cache miss"



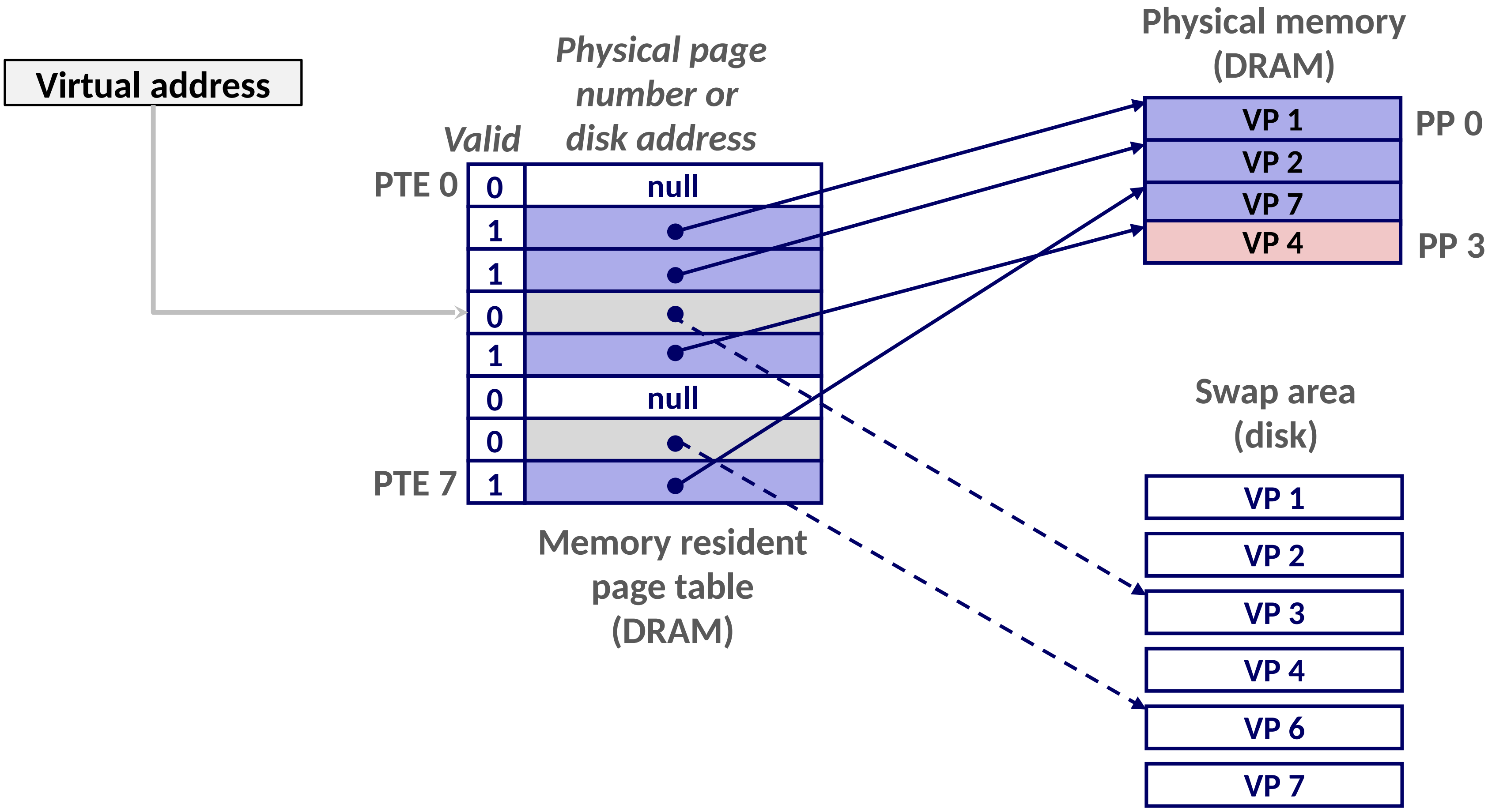
Handling Page Fault

- Page miss causes page fault (an exception)



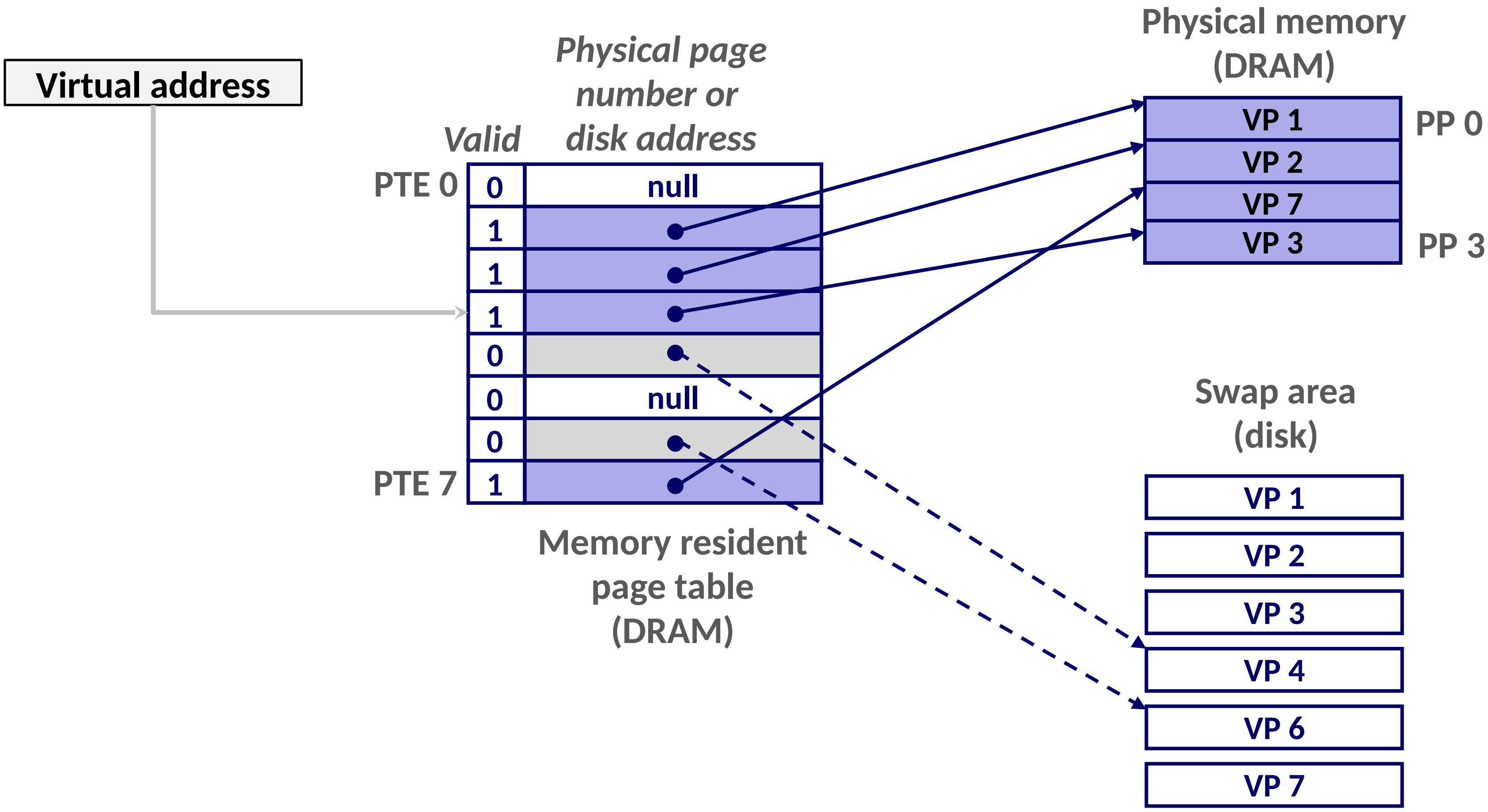
Handling Page Fault

- Page miss causes page fault (an exception)
- Operating system selects a victim to be evicted (here VP 4)



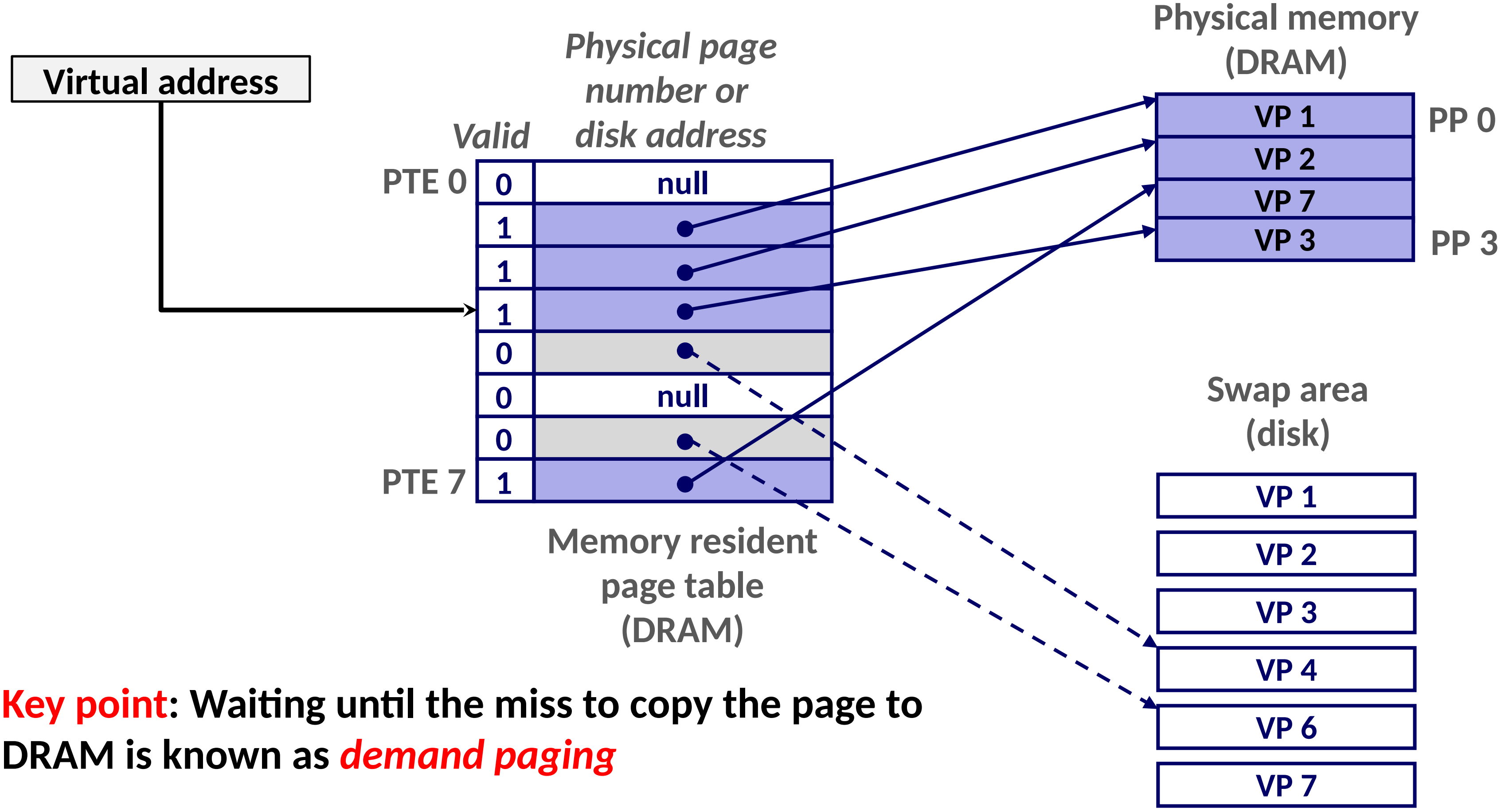
Handling Page Fault

- Page miss causes page fault (an exception)
- Operating system selects a victim to be evicted (here VP 4)



Handling Page Fault

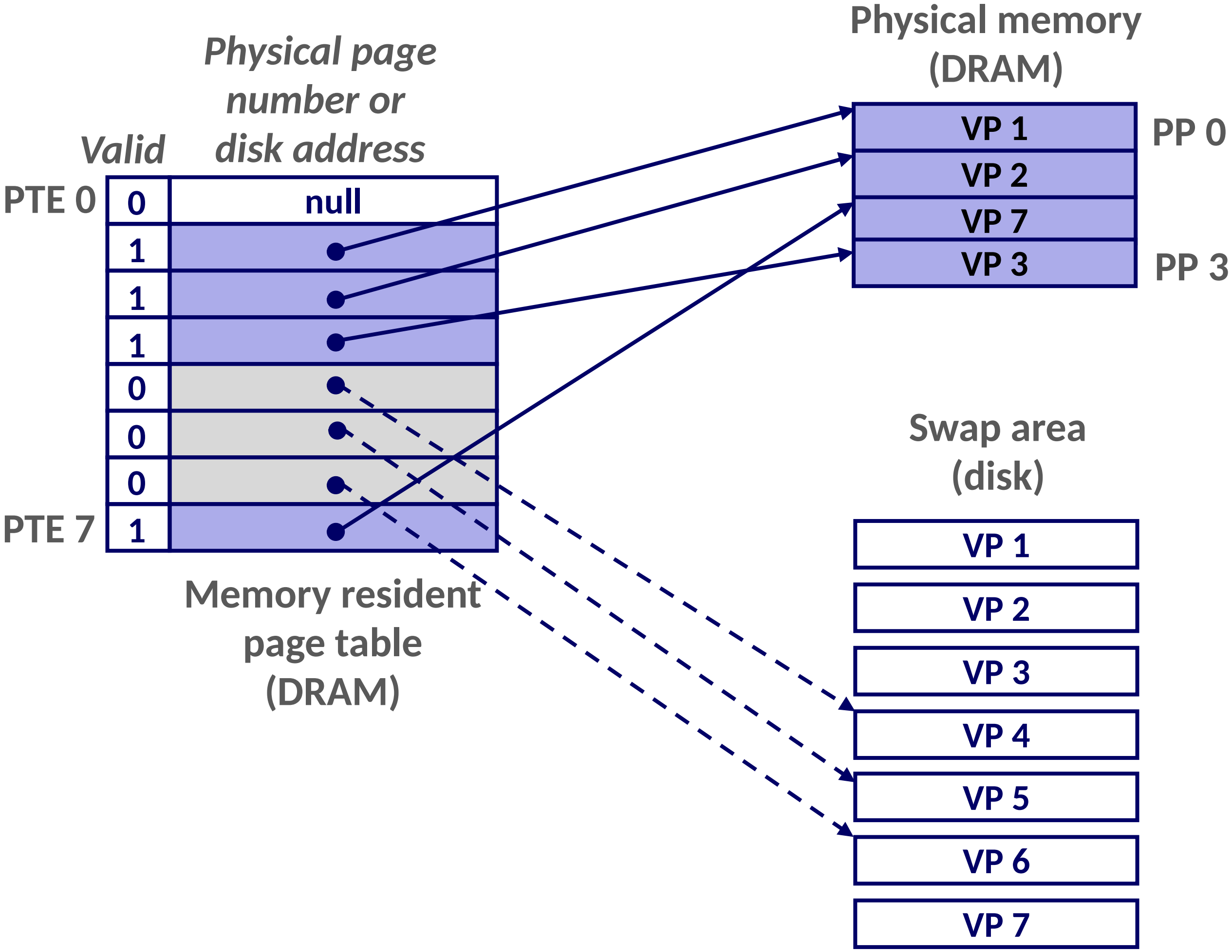
- Page miss causes page fault (an exception)
- Operating system selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!



Key point: Waiting until the miss to copy the page to DRAM is known as *demand paging*

Allocating Pages

- Allocating a new page (VP 5) of virtual memory.



Locality to the Rescue Again!

- **Virtual memory seems terribly inefficient, but it works because of locality.**

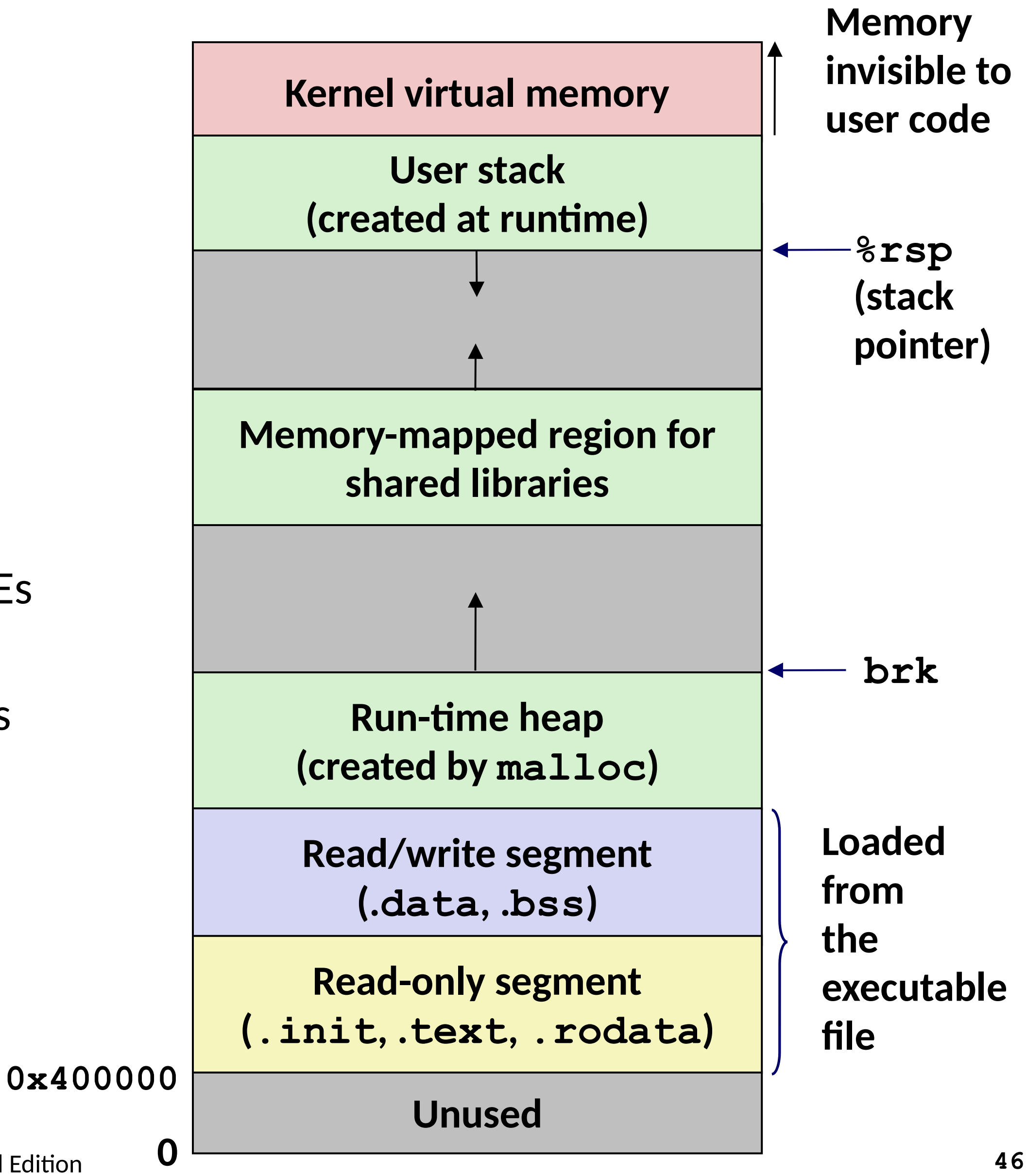
- **At any point in time, programs tend to access a set of active virtual pages called the *working set***
  - Programs with better temporal locality will have smaller working sets

- **If (working set size < main memory size)**
  - Good performance for one process after compulsory misses

- **If (SUM(working set sizes) > main memory size)**
  - *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously

Linking and Loading Revisited

- **Linking**
  - Each program has similar virtual address space
  - Code, data, and heap always start at the same addresses.
- **Loading**
  - Allocate virtual pages for `.text` and `.data` sections & creates PTEs marked as invalid
  - The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system



Summary

- **Programmer's view of virtual memory**
  - Each process has its own private address space
  - Cannot be corrupted by other processes

- **System view of virtual memory**
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions
  - Allows using DRAM as a cache of disk when low on memory
    - Efficient only because of locality