CS 153
Design of Operating Systems

Fall 19

Lecture 12: Advanced Paging

Instructor: Chengyu Song

Advanced Paging

- So far we have discussed how to make memory access faster under paging

- Next, we will discuss interesting tricks on using paging (how those bits in the PTE are used)

  - Sharing

  - Copy-on-Write

  - Memory mapped file

  - On-demand mapping

  - Virtual memory

Core i7 Level 1-3 Page Table Entries

63	62	52	51	12	11	9	8	7	6	5	4	3	2	1	0
XD	Unused		Page table physical base address		Unused		G	PS		A	CD	WT	U/S	R/W	P=1

Available for OS (page table location on disk)	P=0

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.

CD: Caching disabled or enabled 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 2 MB (defined for Level 1 PTEs only).

G: Global page (don't evict from TLB on task switch)

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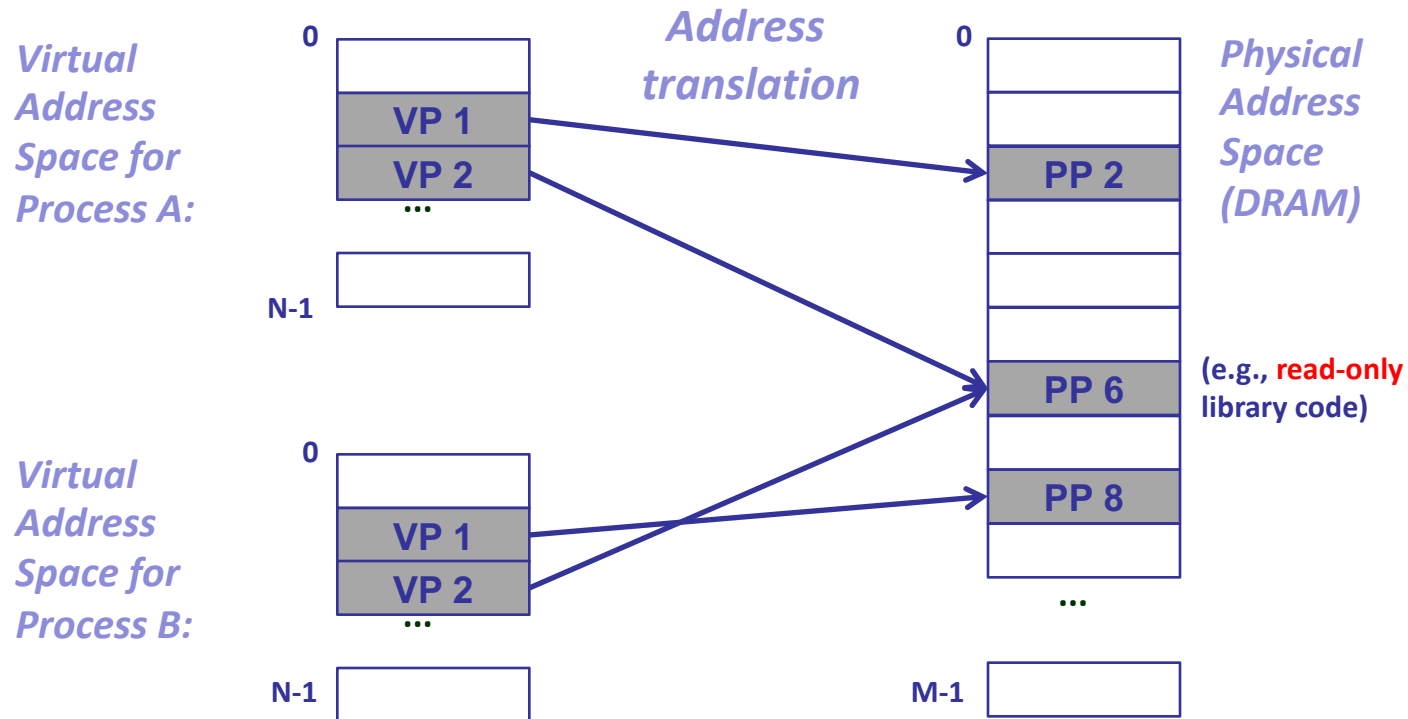
XD: Non-executable pages

Sharing

- Private virtual address spaces protect applications from each other

    - Usually exactly what we want

- But this makes it difficult to share data (have to copy)

    - Parents and children in a forking Web server or proxy will want to share an in-memory cache without copying

- We can use shared memory to allow processes to share data using direct memory references

    - Both processes see updates to the shared memory segment

        - » Process B can immediately read an update by process A

Sharing (2)

- Sharing code and data among processes
  - Map virtual pages to the same physical page (here: PP 6)

Virtual Address Space for Process A:

Address translation

Physical Address Space (DRAM)

0

VP 1
VP 2
...

N-1

0

PP 2

PP 6

(e.g., **read-only** library code)

PP 8

...

Virtual Address Space for Process B:

0

VP 1
VP 2
...

N-1

M-1

Sharing (3)

- Can map shared memory at same or different virtual addresses in each process' address space

  - Different:

    - » 10th virtual page in P1 and 7th virtual page in P2 correspond to the 2nd physical page

    - » Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid

  - Same:

    - » 2nd physical page corresponds to the 10th virtual page in both P1 and P2

   - » Less flexible, but shared pointers are valid

Sharing (4)

- Linux API

  - Map to different address

    - » `shm_open()`: create and open a new object, or open an existing object.

    - » `mmap()`: map the shared memory object into the virtual address space of the calling process.

  - Map to the same address

    - » `mmap()`: with MAP_SHARED

Copy on Write

- Recall what happens during `fork()`

  - Entire address spaces needs to be copied

- Use Copy on Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether

  - Instead of copying pages, create shared mappings of parent pages in child virtual address space

  - Shared pages are protected as read-only in parent and child

    » Reads happen as usual

    » Writes generate a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction
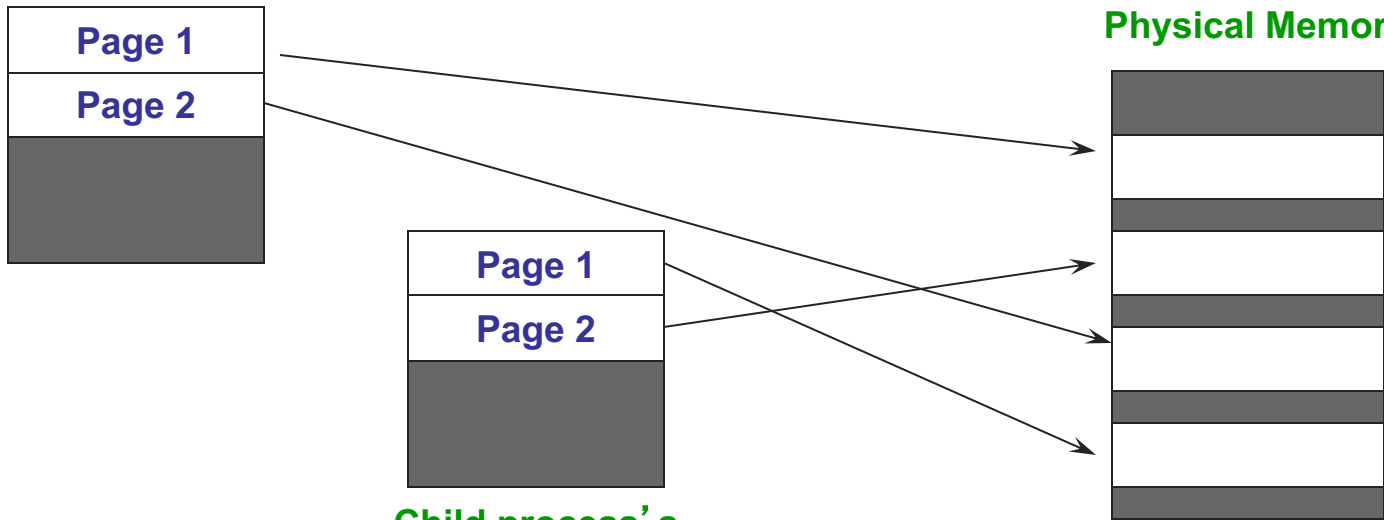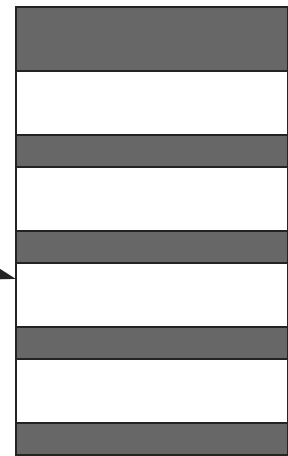
Execution of fork()

Parent process's page table

Page 1
Page 2
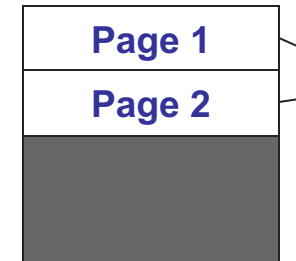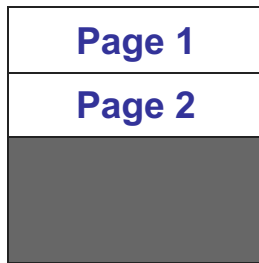
Child process's page table

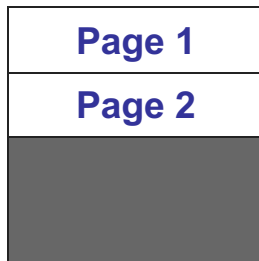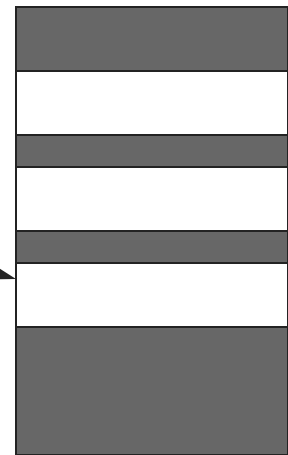| Page 1 |
|--------|
| Page 2 |
| |

Physical Memory

fork() with Copy on Write

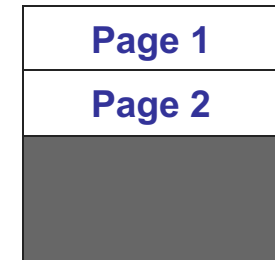When either process modifies Page 1, page fault handler allocates new page and updates PTE in child process

Parent process's page table

| Page 1 |
| Page 2 |

Physical Memory

Protection bits set to prevent either process from writing to any page

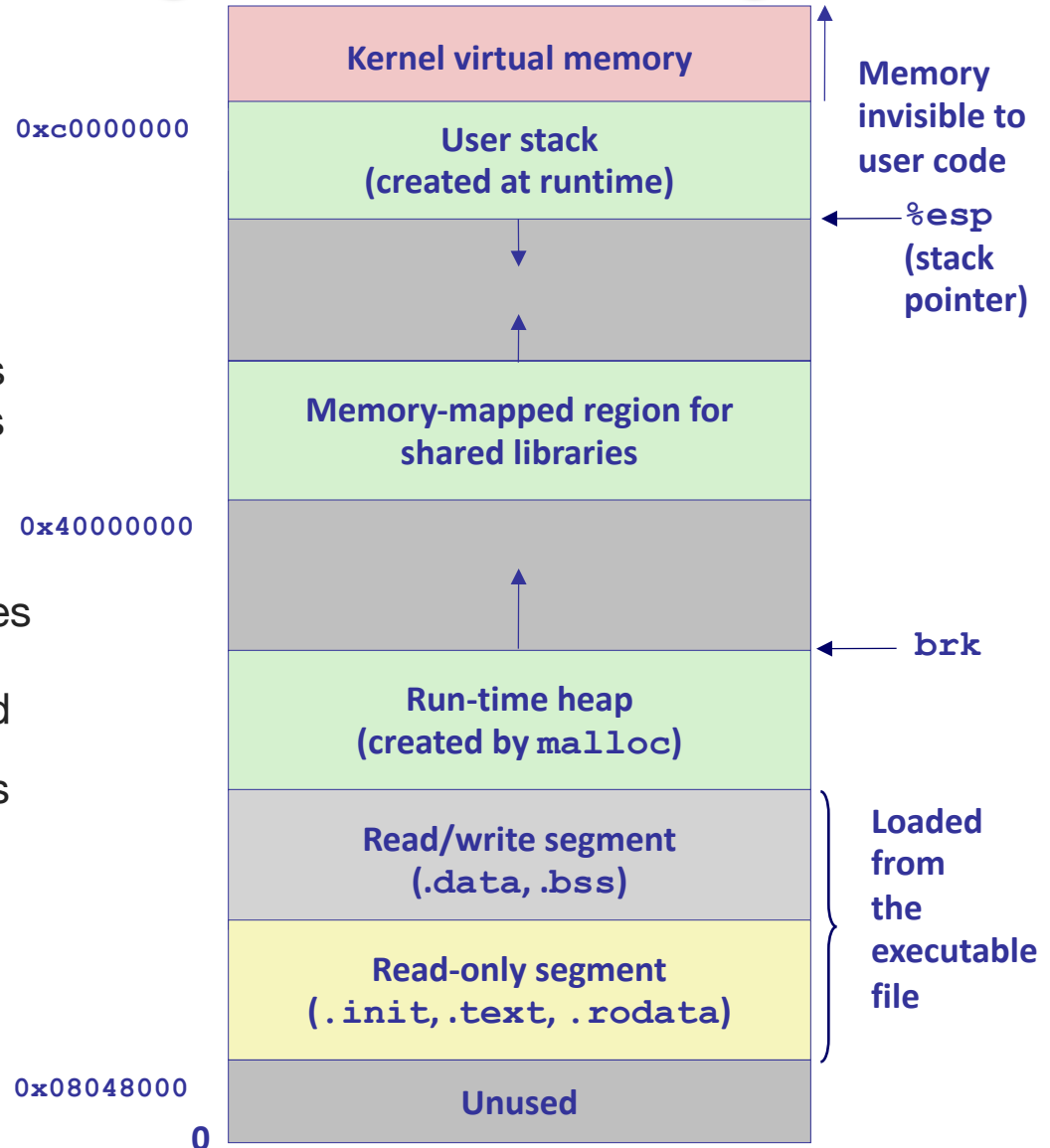**Child process's page table**

| Page 1 |
| Page 2 |

Simplifying Linking and Loading

- Linking

  - Each program has similar virtual address space

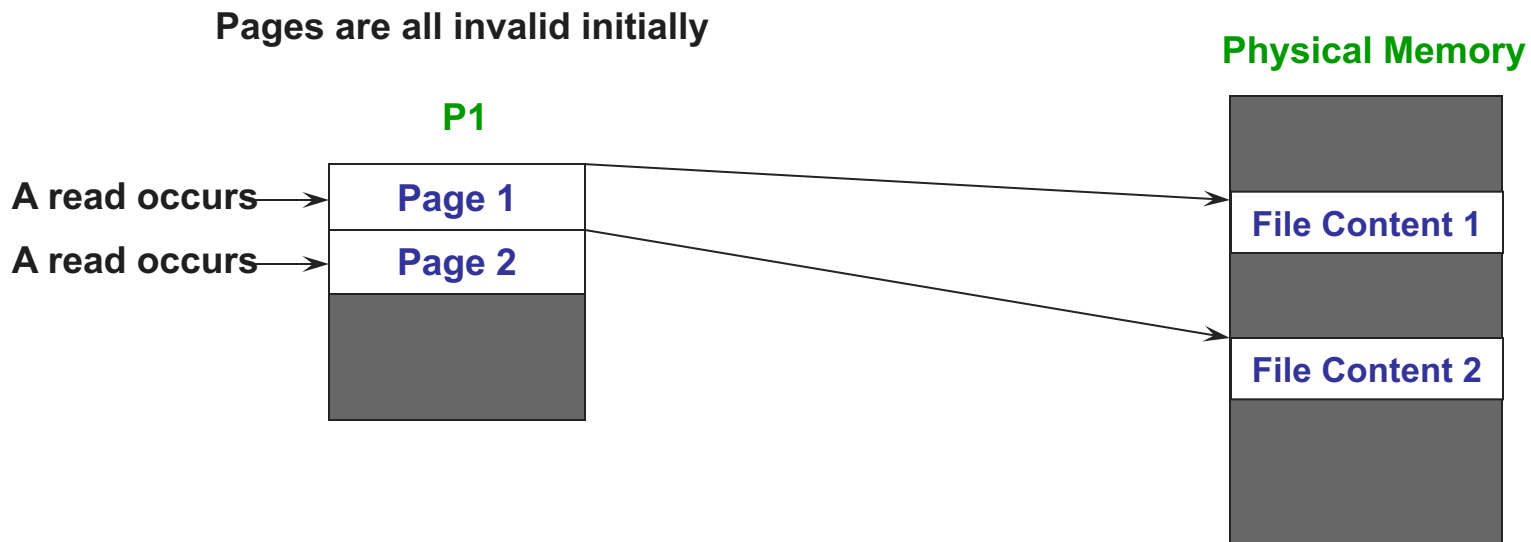 - Code, stack, and shared libraries always start at the same address

- Loading

  - `execve()` allocates 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

Kernel virtual memory	→
User stack **(created at runtime)**	**Memory invisible to user code**
	← `%esp` (stack pointer)
↓	
↑	
Memory-mapped region for shared libraries	
↑	
	← `brk`
Run-time heap **(created by `malloc`)**	
Read/write segment **(.data, .bss)**	**Loaded from the executable file**
Read-only segment **(.init, .text, .rodata)**	
Unused	

`0xc0000000`

`0x40000000`

`0x08048000`

`0`

Mapped Files

- Mapped files enable processes to do file I/O using loads and stores

  - Instead of "open, read into buffer, operate on buffer, …"

- Bind a file to a virtual memory region (mmap() in Unix)

  - PTEs map virtual addresses to physical frames holding file data

  - Virtual address base + N refers to offset N in file

- Initially, all pages mapped to file are invalid

  - OS reads a page from file when invalid page is accessed

    » How?

Memory-Mapped Files

Pages are all invalid initially

Physical Memory

P1

A read occurs → Page 1

A read occurs → Page 2

File Content 1

File Content 2

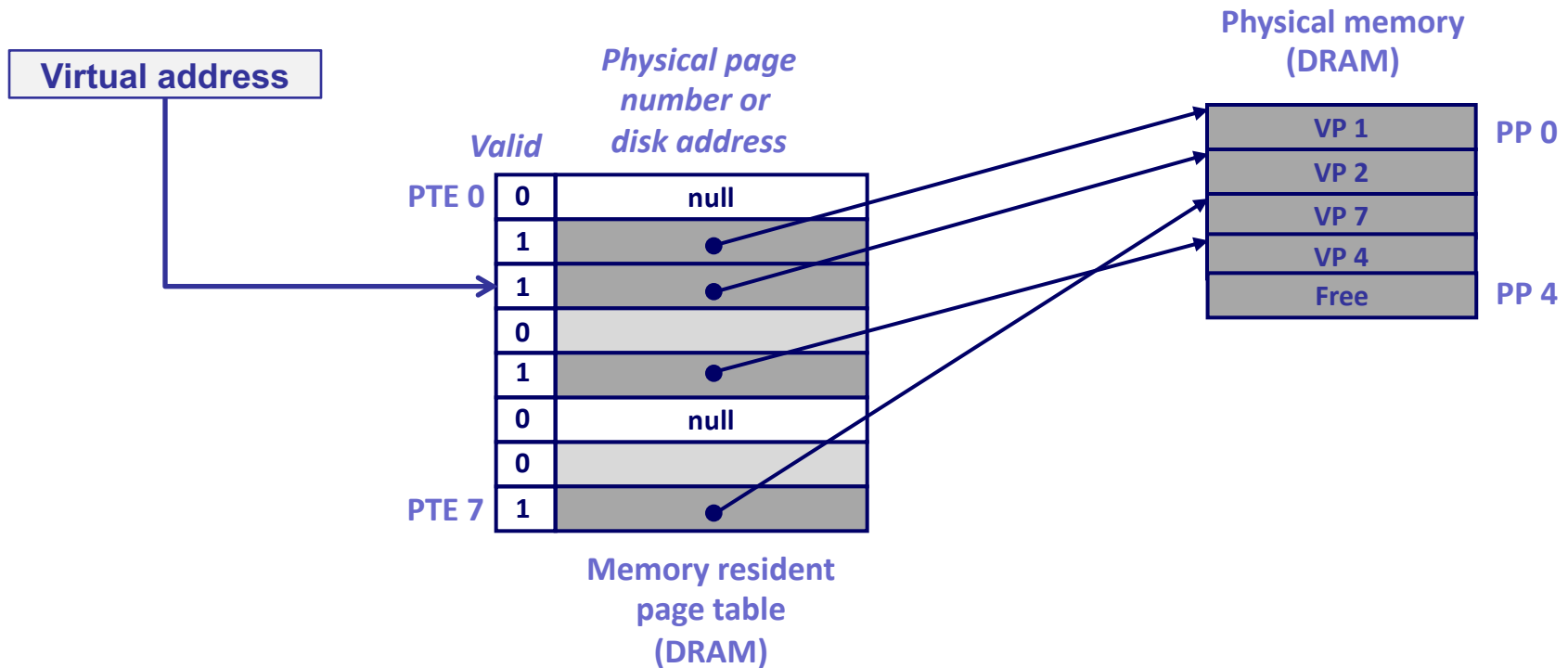What happens if we unmap the memory?
How do we know whether we need to write changes back to file?

Writing Back to File

- OS writes a page to file when evicted, or region unmapped

- Dirty bit trick (not protection bits)

  - If page is not dirty (has not been written to), no write needed

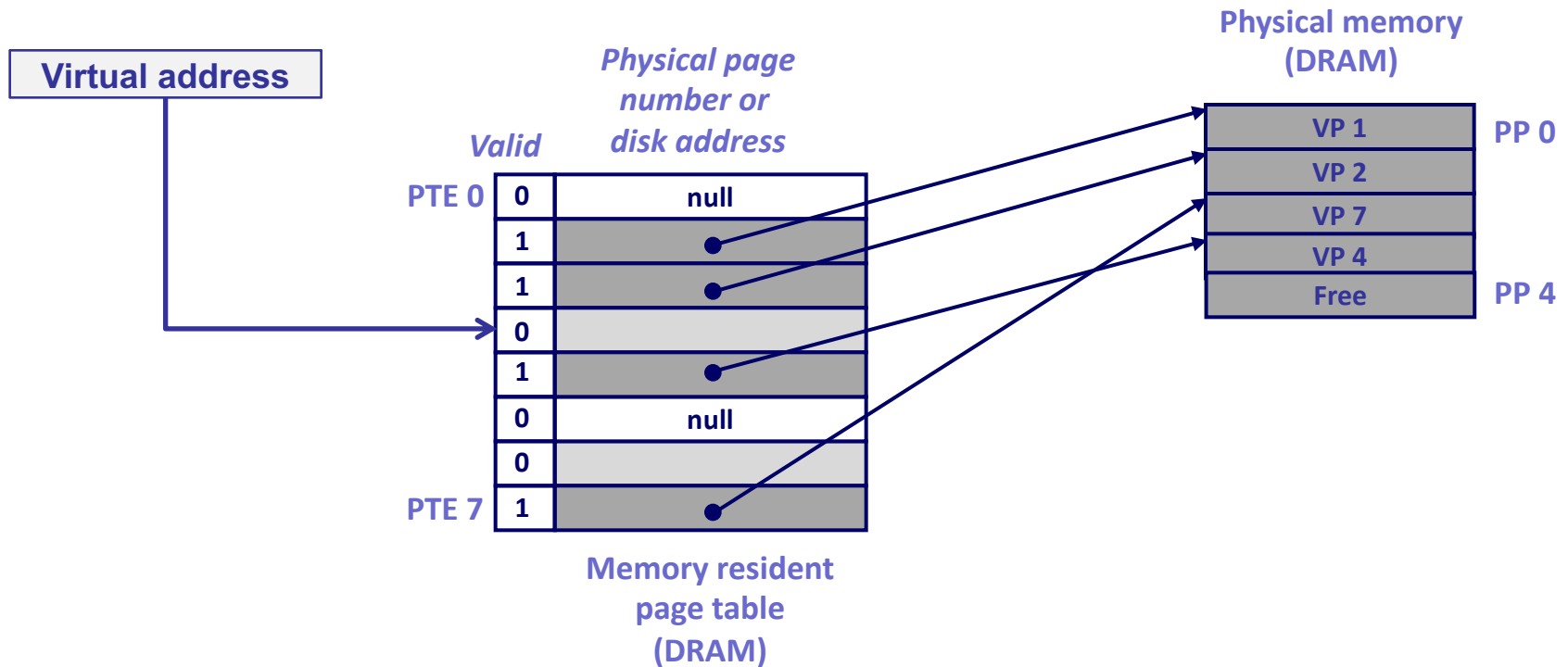# Page Hit

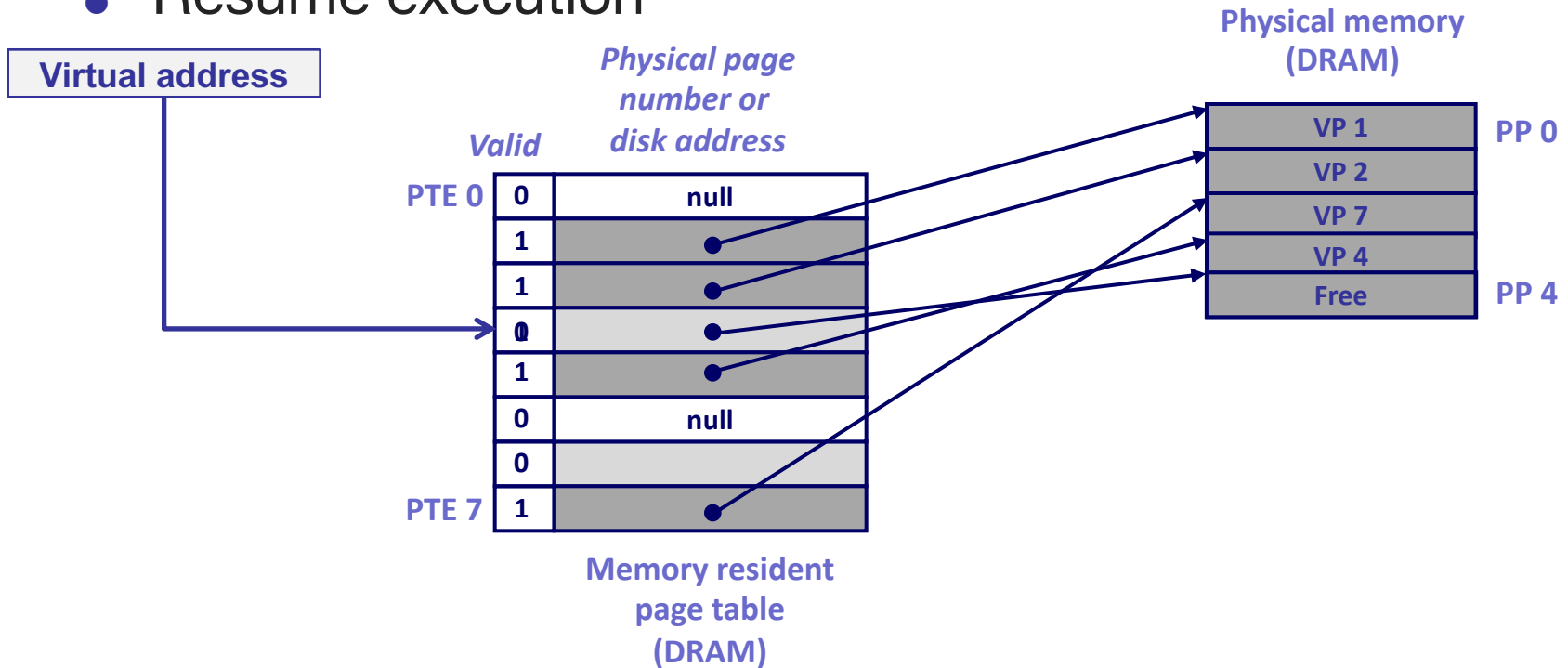- Page hit: reference to VM word that is in physical memory (DRAM cache hit)

Page Fault

- Page fault: reference to VM word that is not in physical memory (DRAM cache miss)

On-demand Mapping

- Allocate physical page

- Fix the page table

- Resume execution

How do we know whether the fault is fixable?

On-demand Mapping

- When the process calls `mmap()`, the kernel remembers

  - The region [addr, addr+length]

    » What virtual addresses are valid/mapped

  - The backing: just memory (ANONYMOUS) or file

- During page fault handling, the kernel checks

  - If the faulty virtual address is valid

  - If so, fix based on the backing

Memory Protection

- R/W (read-only or writable)

  - We've seen how it is used in CoW

  - It is also important in preventing attacks (e.g., mark code as read-only so attackers cannot modify them)

- U/S (user or kernel)

  - How do we protect the kernel? Give it a different address space?

    - » Too expensive for context switch during system calls

    - » May not be a bad idea if security is a concern (recent Meltdown attack)

Memory Protection (2)*

- ## U/S

  - Besides protecting the kernel from directly accessed from user space, this bit is also used to prevent kernel from executing wrong code or access wrong data, why?

    - » Attackers can attack the kernel and try to execute user space code under kernel context (privilege)

- ## XD (executable or not)

  - In the old days there's an attack technique called "code injection" where attacker force the CPU to interpret data as code

  - XD is a response to such attacks by marking data pages as not executable