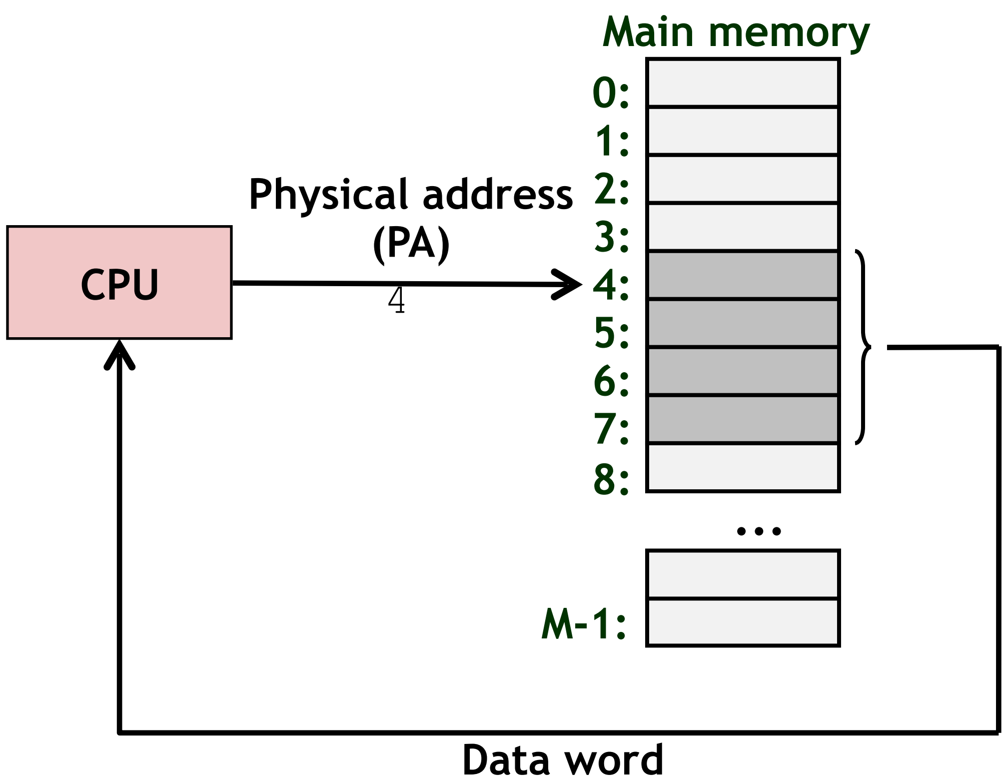
Virtual Memory

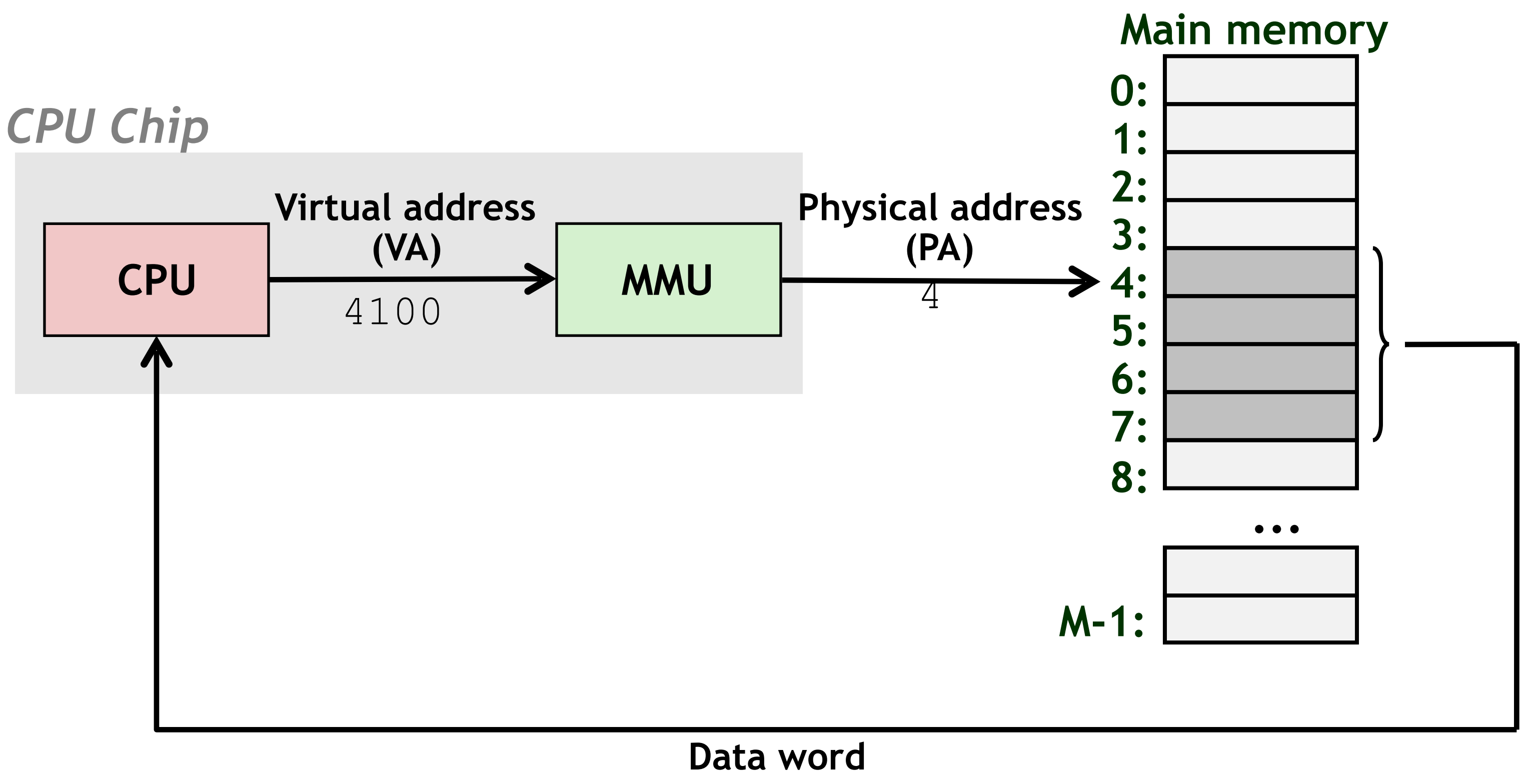
+ A System Using Physical Addressing

- 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**



+ Address Spaces

- **Linear address space: Ordered set of contiguous non-negative integer addresses:**

  `{ 0, 1, 2, 3 … }`

- **Physical address space: Set of $M = 2^m$ physical, linear addresses**

  `{ 0, 1, 2, 3, …, M-1 }`

- **Virtual address space: Set of $N = 2^n$ virtual, linear addresses**

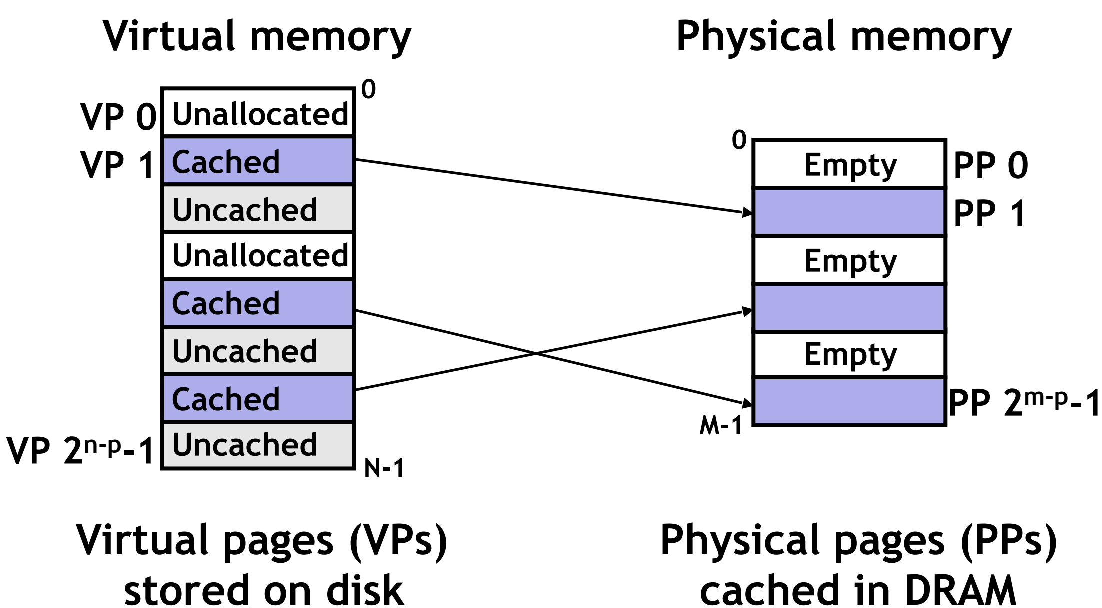
  `{ 0, 1, 2, 3, …, N-1 }`

+ Why Virtual Memory (VM)?

- **Uses main memory efficiently**
  - Use DRAM as a cache for parts of a virtual address space
- **Simplifies memory management**
  - Each process gets the same uniform linear address space
- **Isolates address spaces**
  - One process can’t interfere with another’s memory
  - User program cannot access privileged kernel information and code

+ VM as a Tool for Caching

- **Conceptually, virtual memory is an array of N contiguous bytes stored on disk.**
- **The contents of the array on disk are cached in physical memory (DRAM)**
  - These cache blocks are called *pages* (size is $P = 2^p$ bytes)

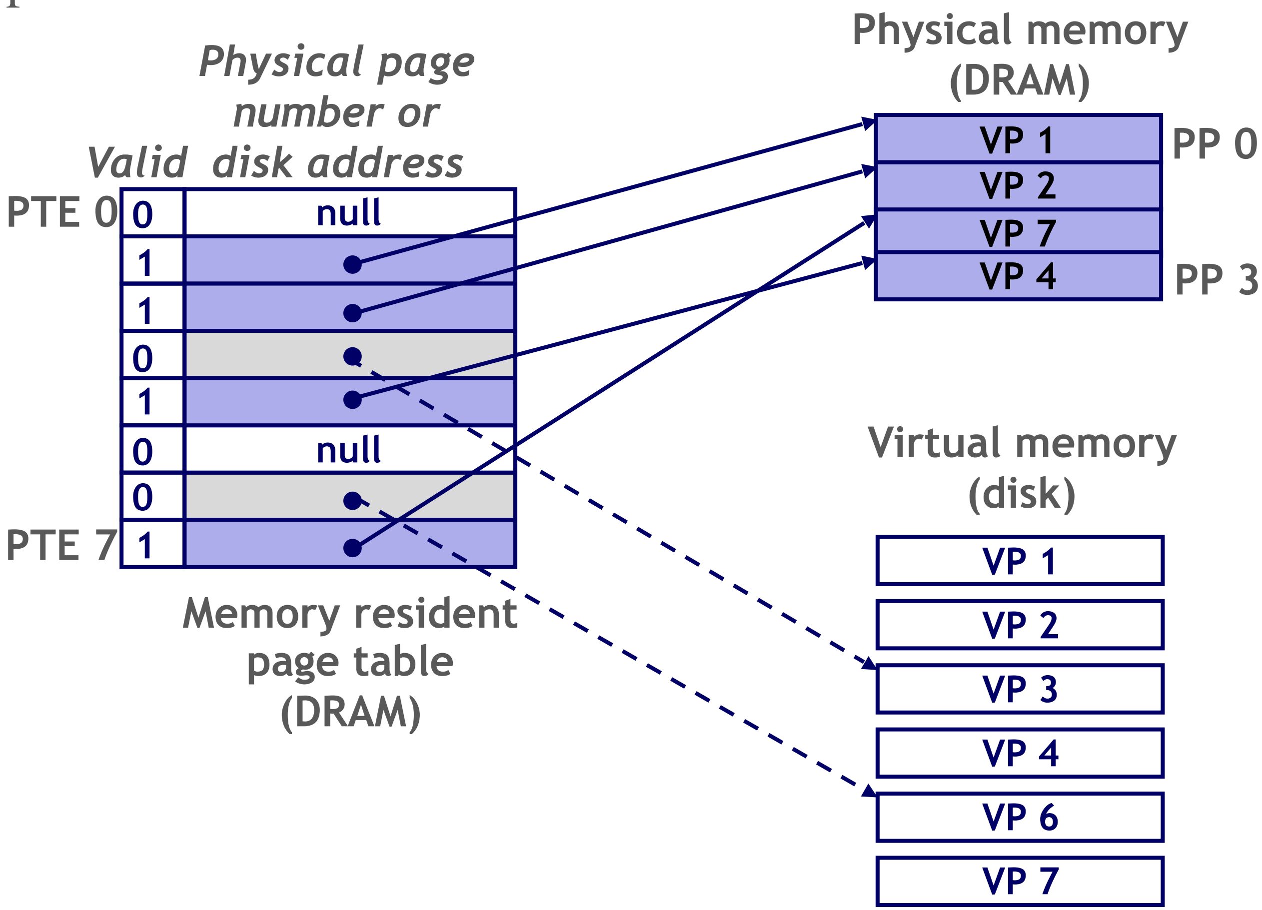


+ DRAM Cache Organization

- **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 complex mapping function
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated to be implemented in hardware
  - *Write-back* rather than *write-through*

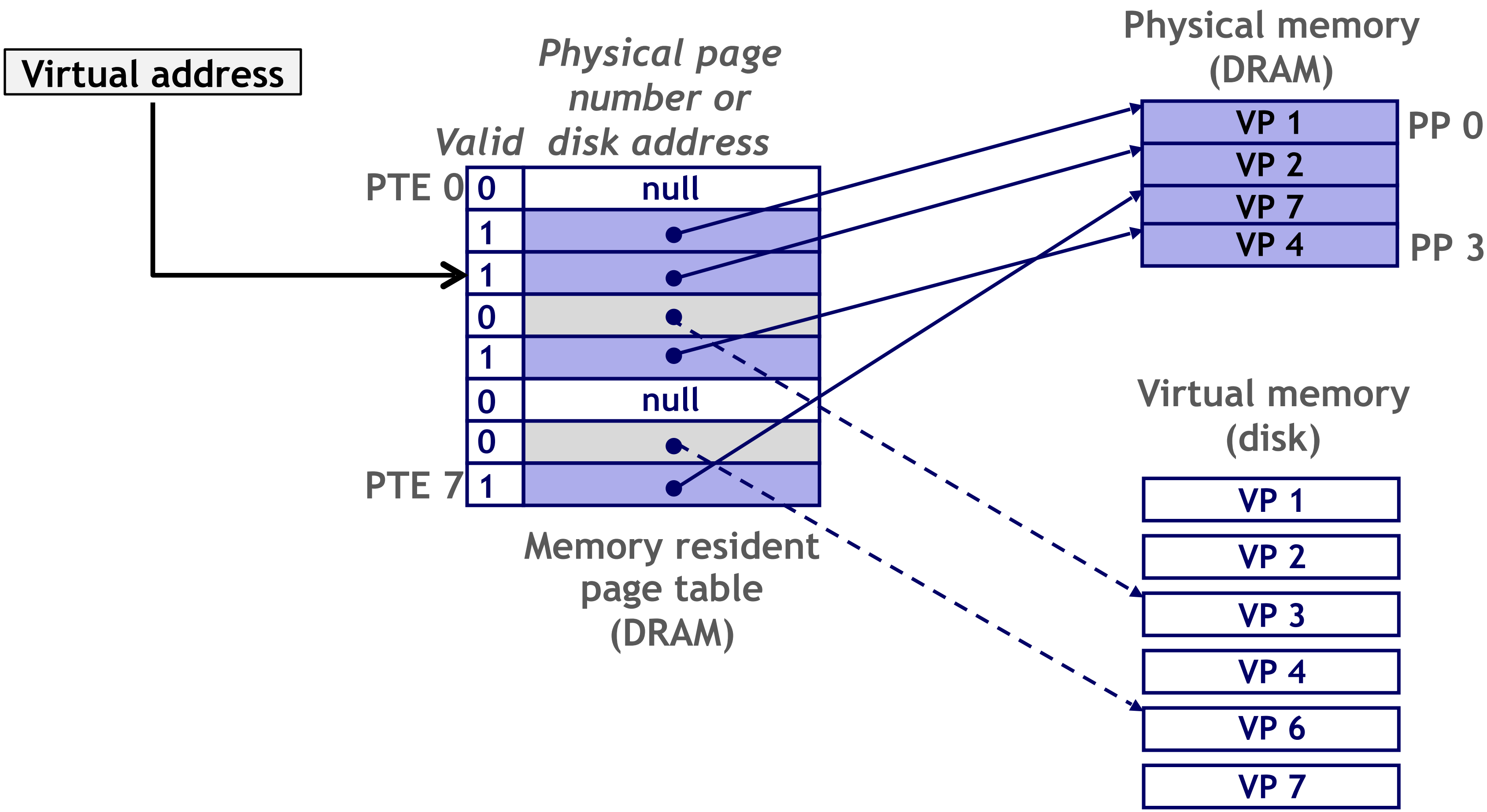
+ Enabling Data Structure: Page Table

- ***Page table:*** **an array of page table entries (PTEs) that maps virtual pages to physical pages.**
  - Per-process kernel data structure in DRAM



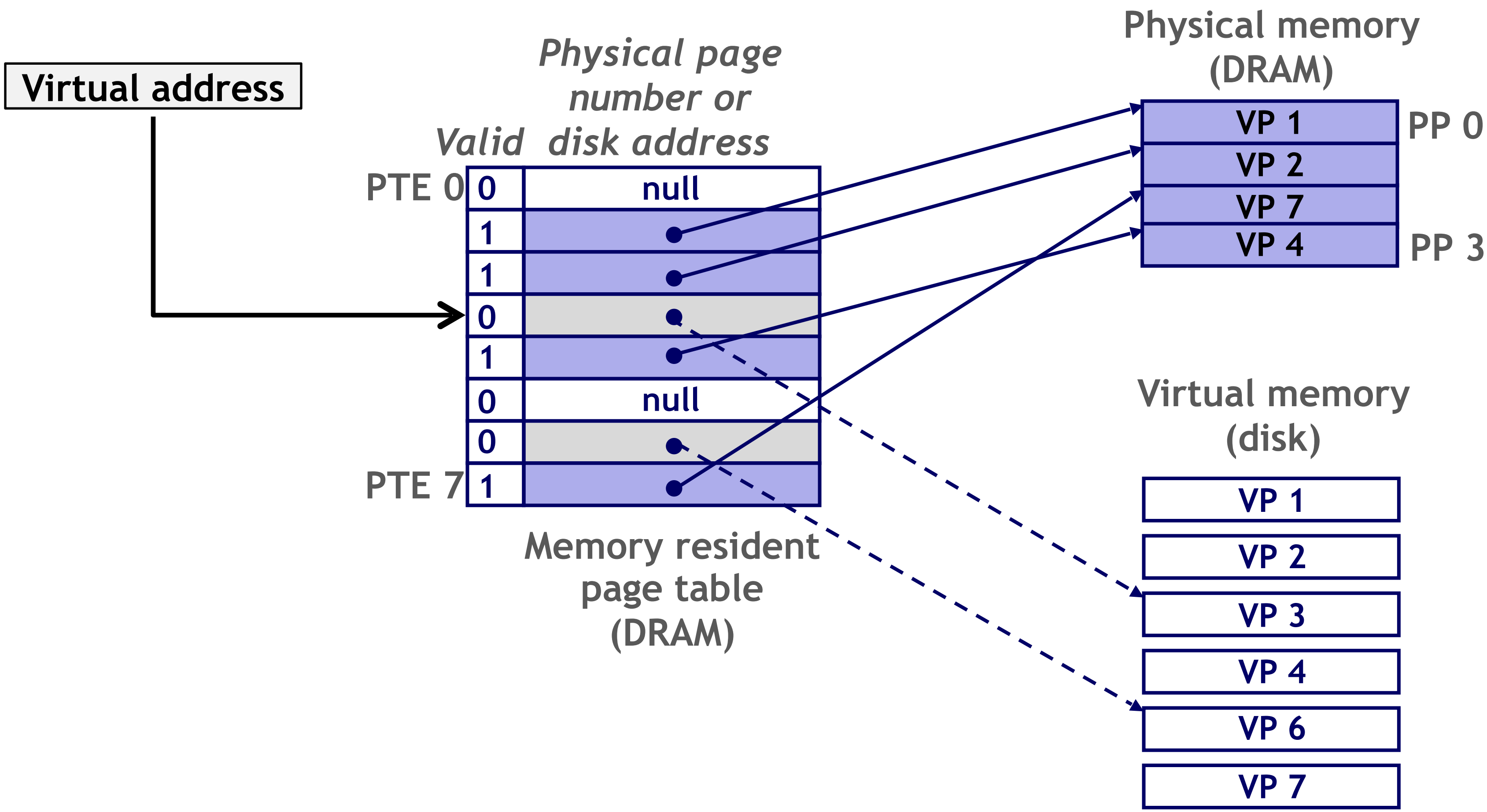
Page Hit

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



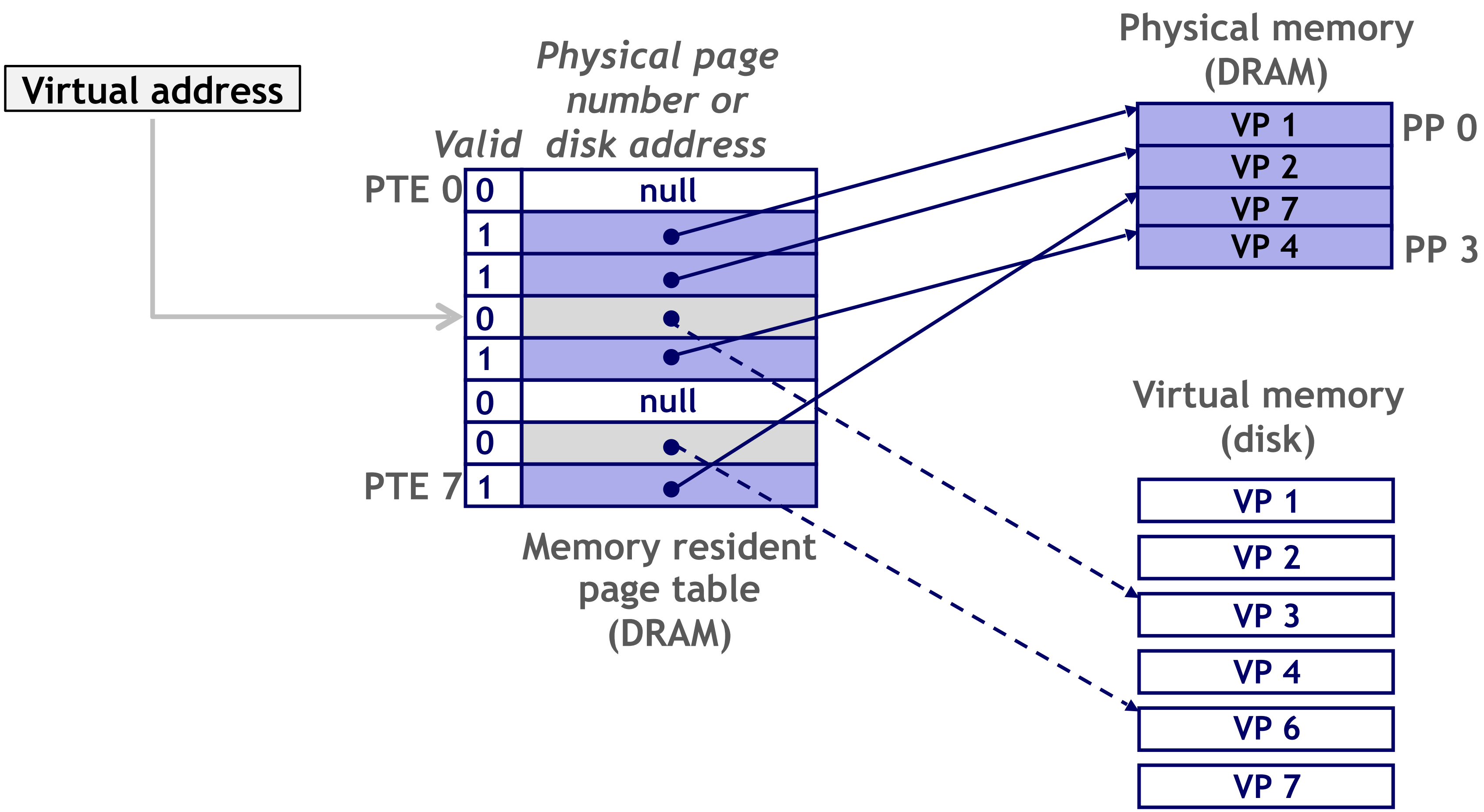
Page Faults

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



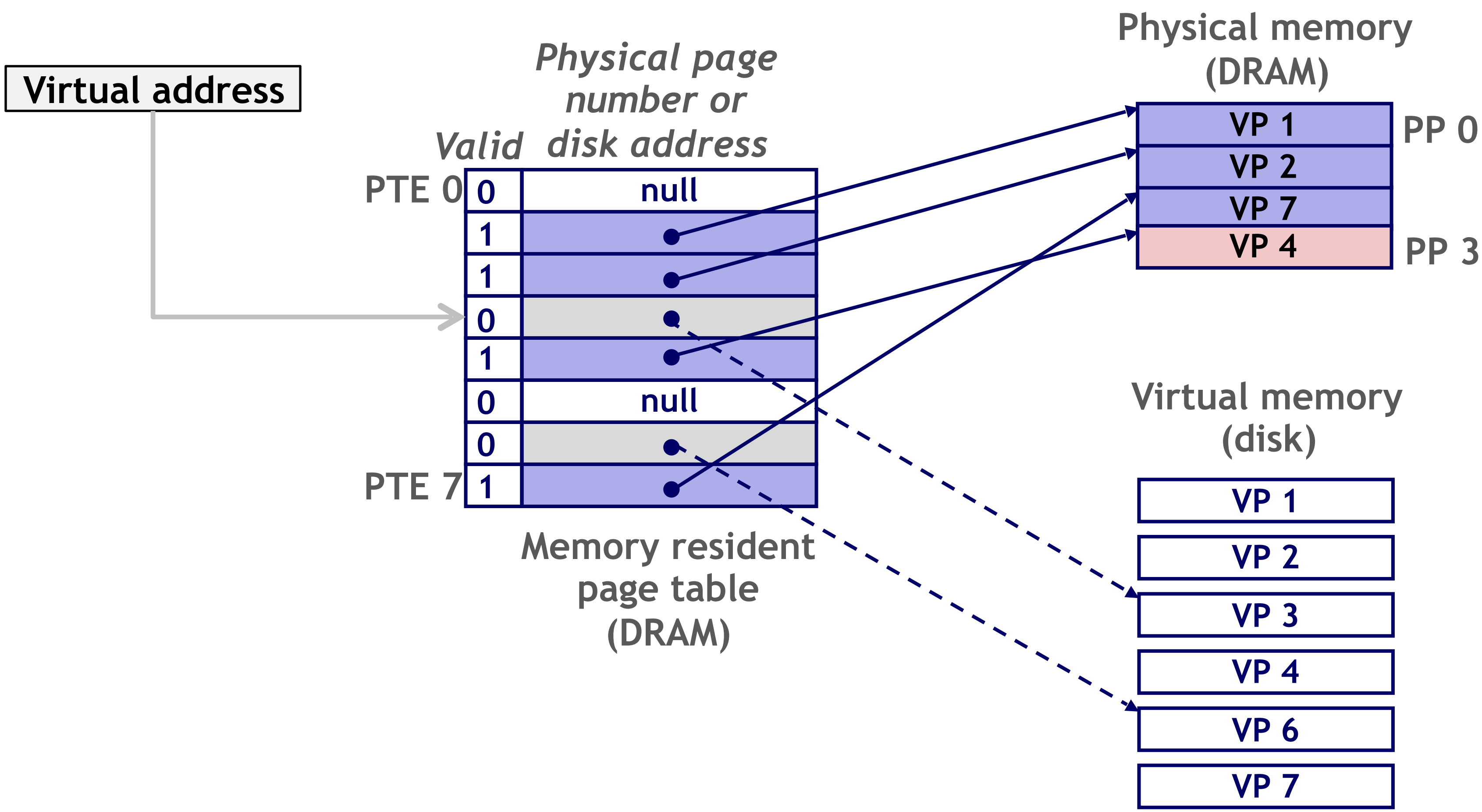
Page Faults *con't*

- **Page miss causes page fault (an exception)**



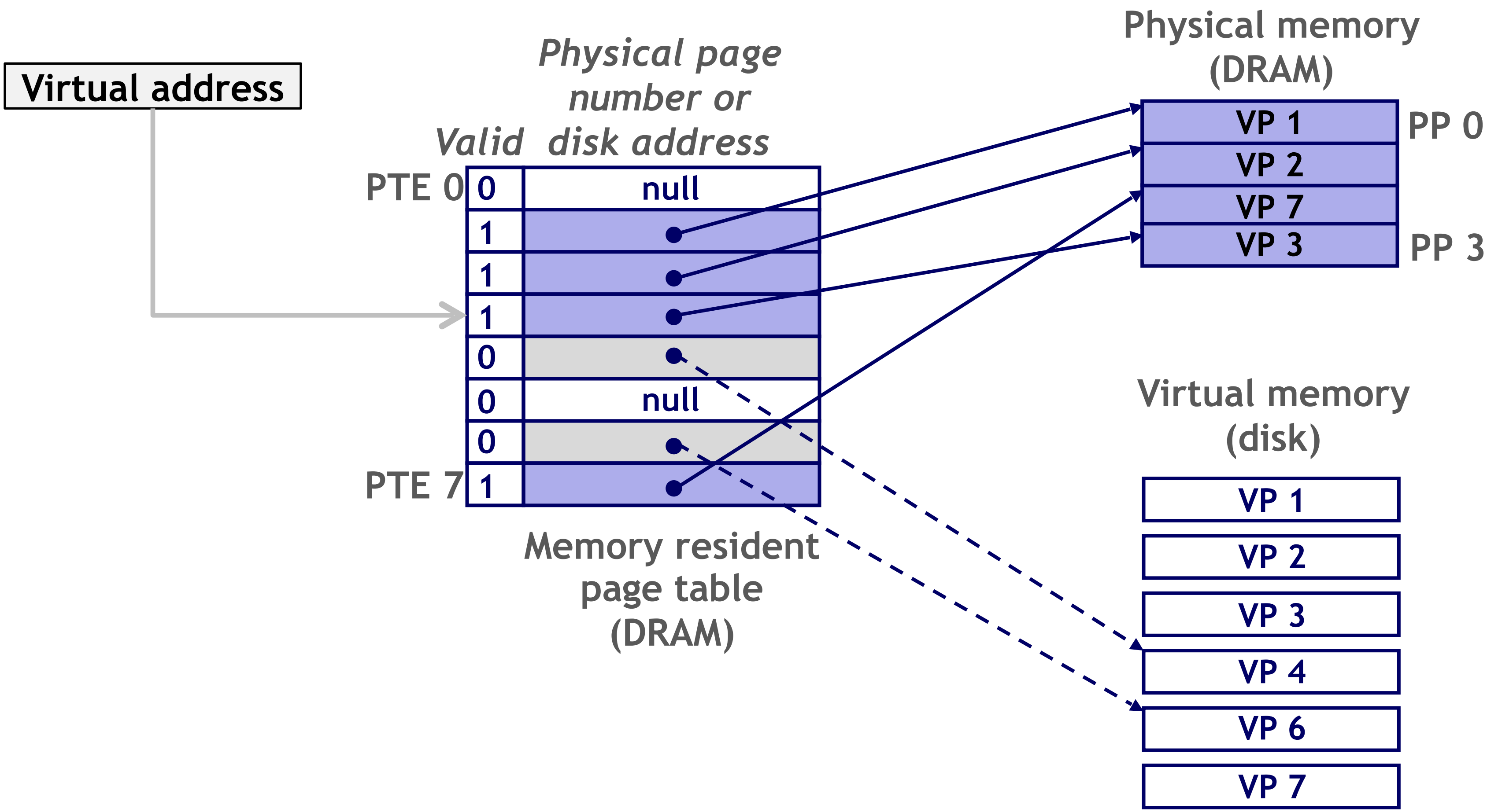
Page Faults *con't*

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



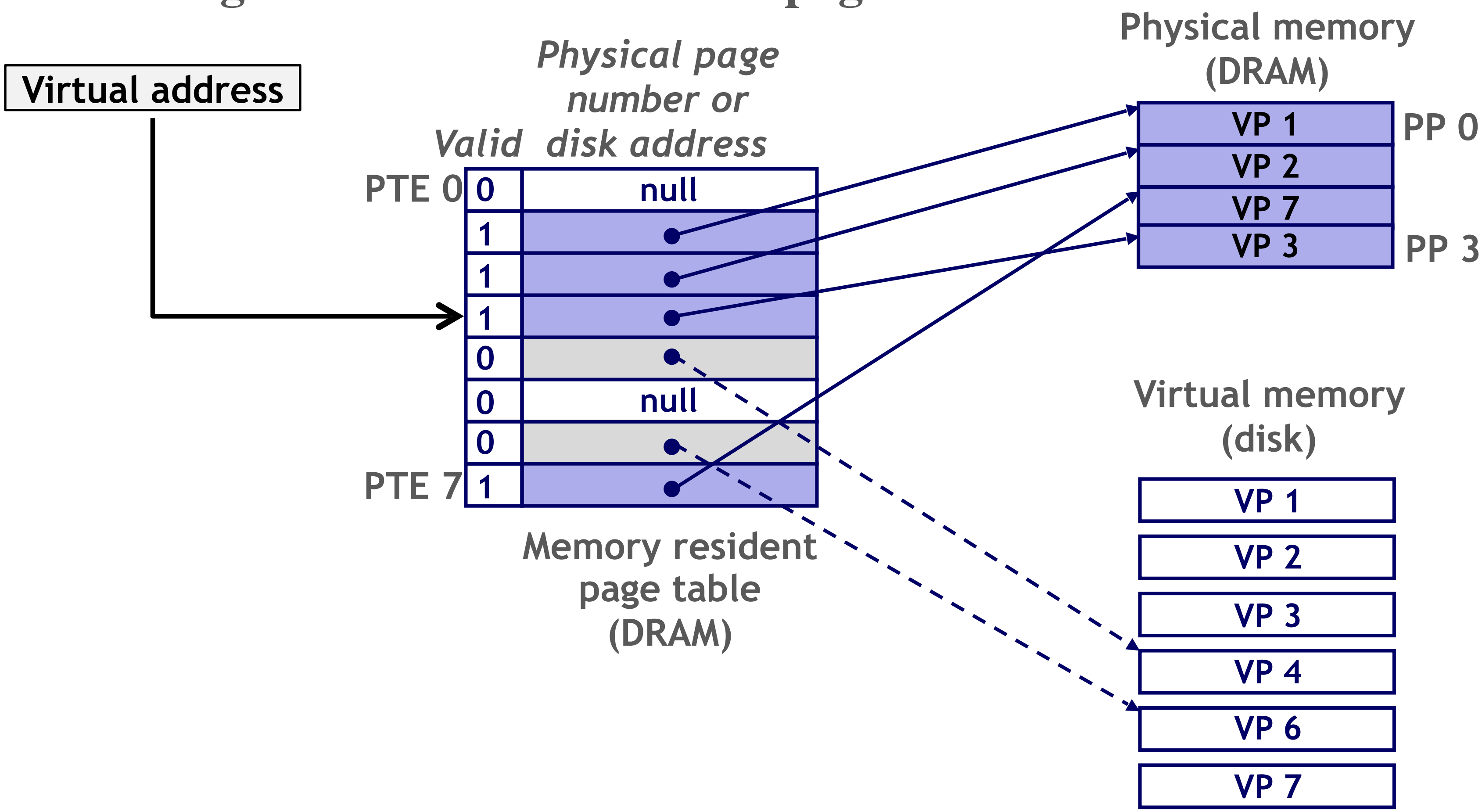
Page Faults *con't*

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



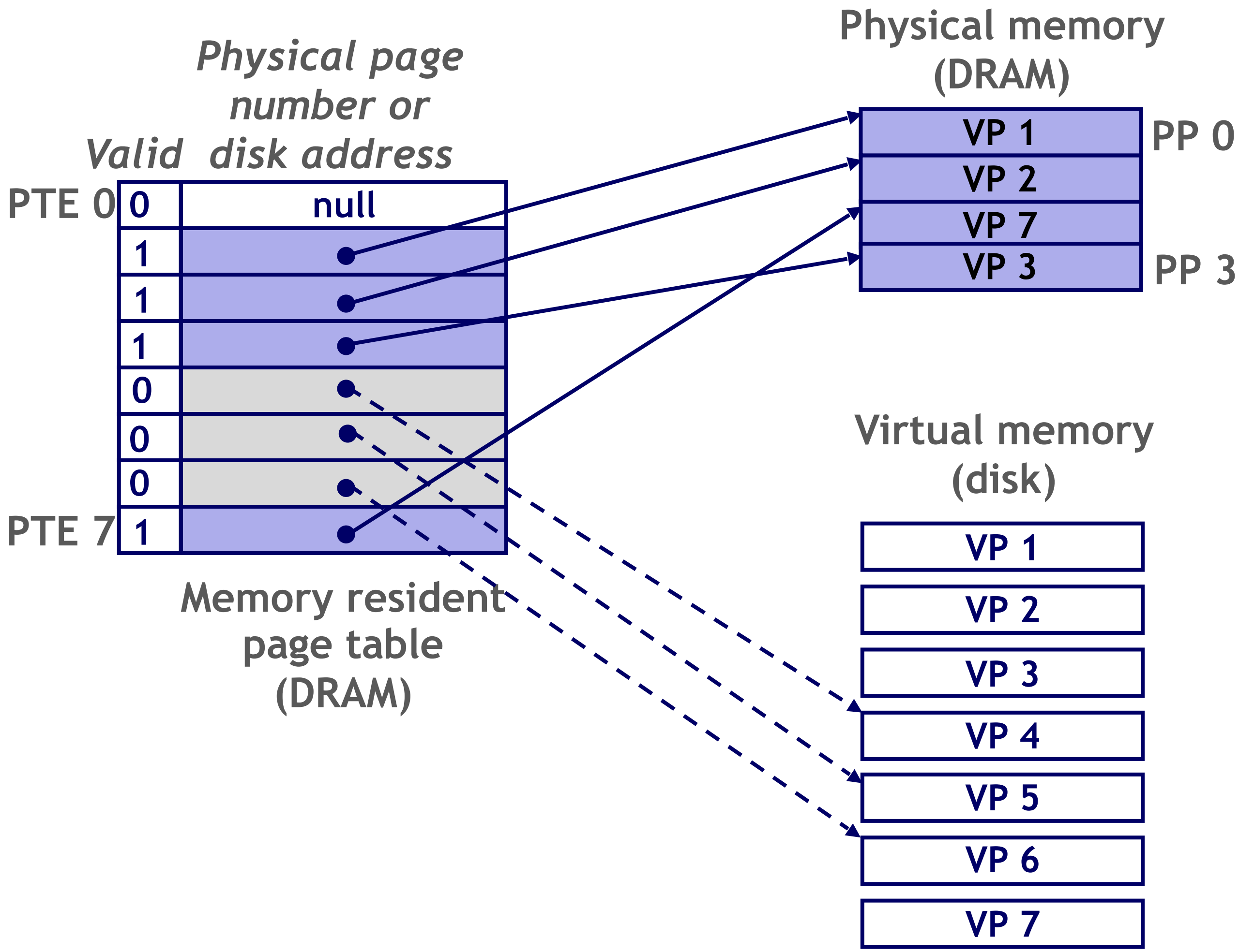
Page Faults *con't*

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



Allocating Pages

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

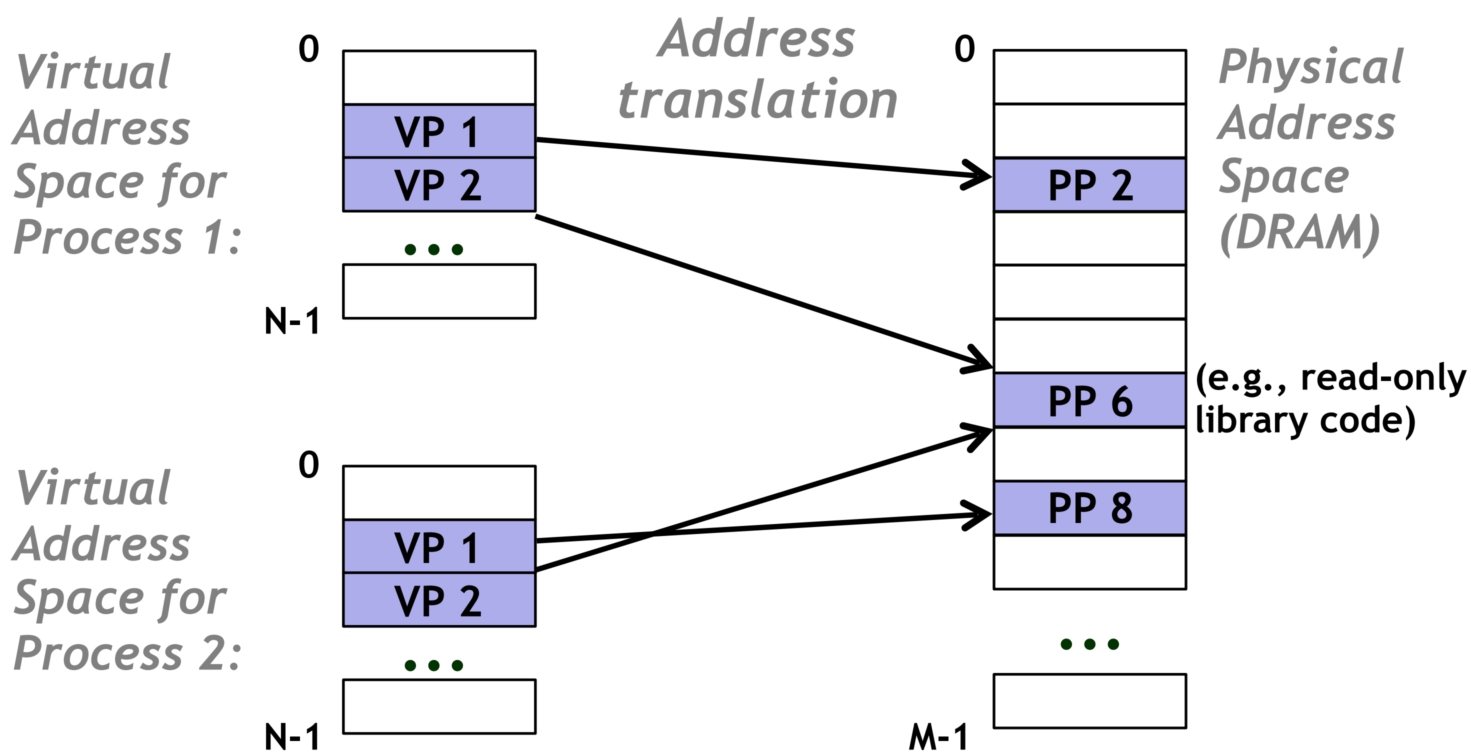


+ Locality to the Rescue Again!

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

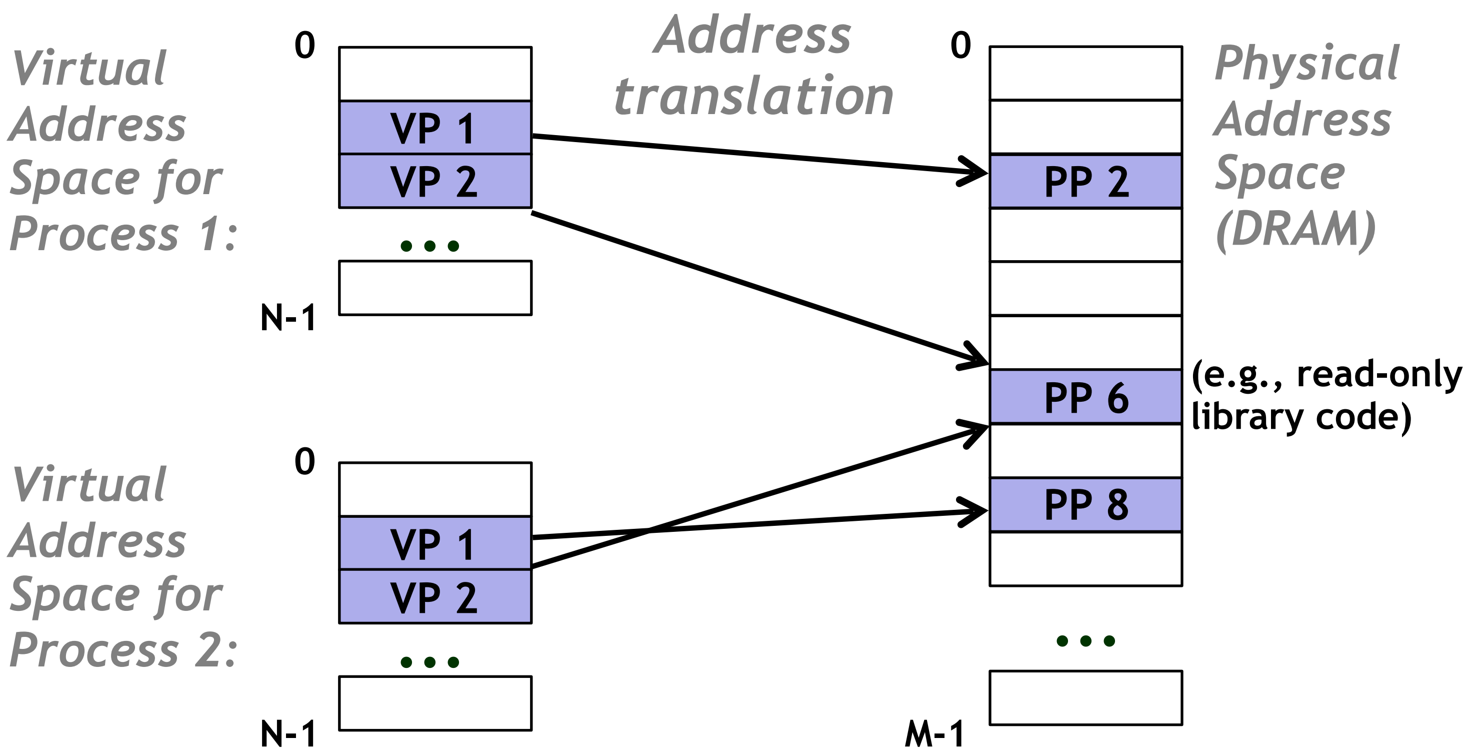
VM as a Tool for Memory Management

- **Key idea: each process has its own virtual address space**
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory



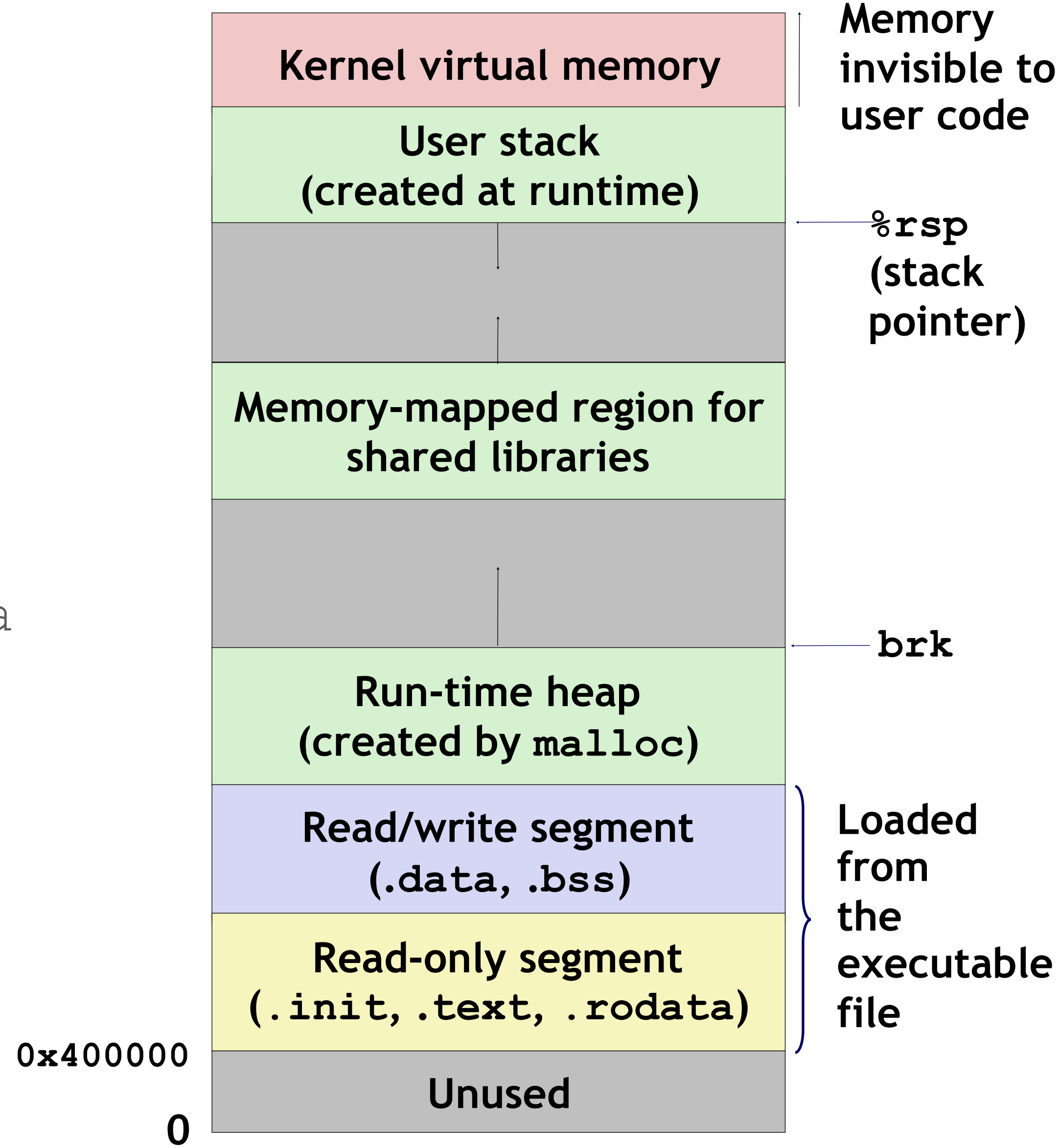
+ VM as a Tool for Memory Management

- **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)



+ Simplifying Linking and Loading

- **Linking**
  - Each program has similar virtual address space
  - Code, data, and heap always start at the same addresses.
- **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



VM as a Tool for Memory Protection

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

Process i:

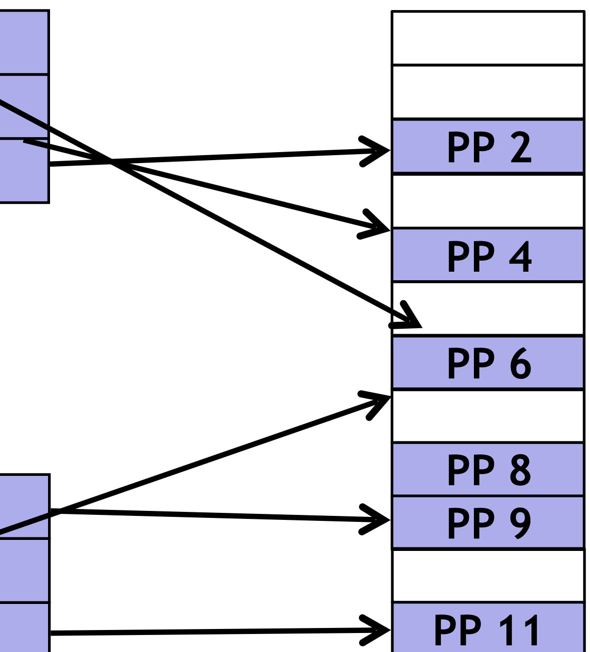
	SUP	READ	WRITE	EXEC	Address
VP 0:	No	Yes	No	Yes	PP 6
VP 1:	No	Yes	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	No	PP 2

⋮

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



+ Summary

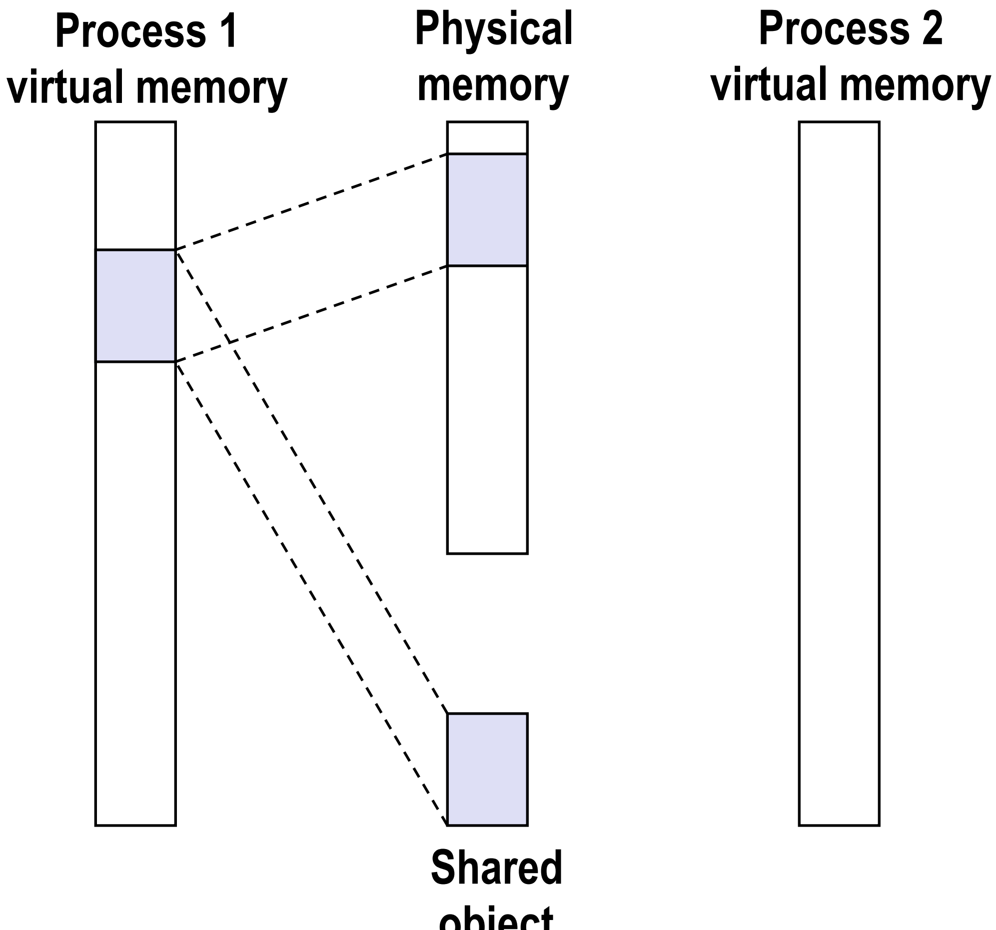
- **Programmer's view of virtual memory**
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes
- **System view of virtual memory**
  - Uses memory efficiently by caching virtual memory pages
    - Efficient because of locality
  - Simplifies memory management and programming
  - Enables memory protection

Memory Mapping

Memory Mapping

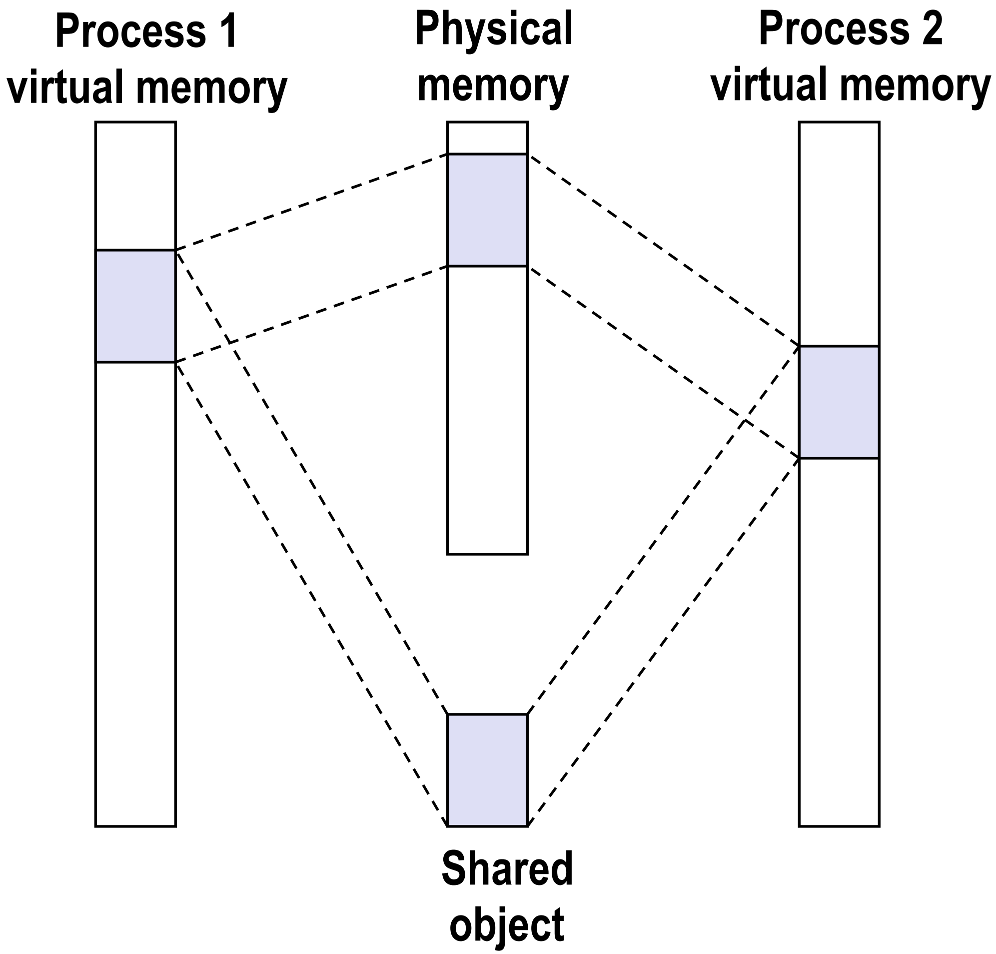
- **VM areas initialized by associating them with disk objects.**
  - Process is known as memory mapping.
- **Area can be backed by (i.e., get its initial values from) :**
  - Regular file on disk
  - Executable
  - "Anonymous" files
- **Dirty pages are copied back and forth between memory and a special swap file.**

Sharing Revisited: Shared Objects



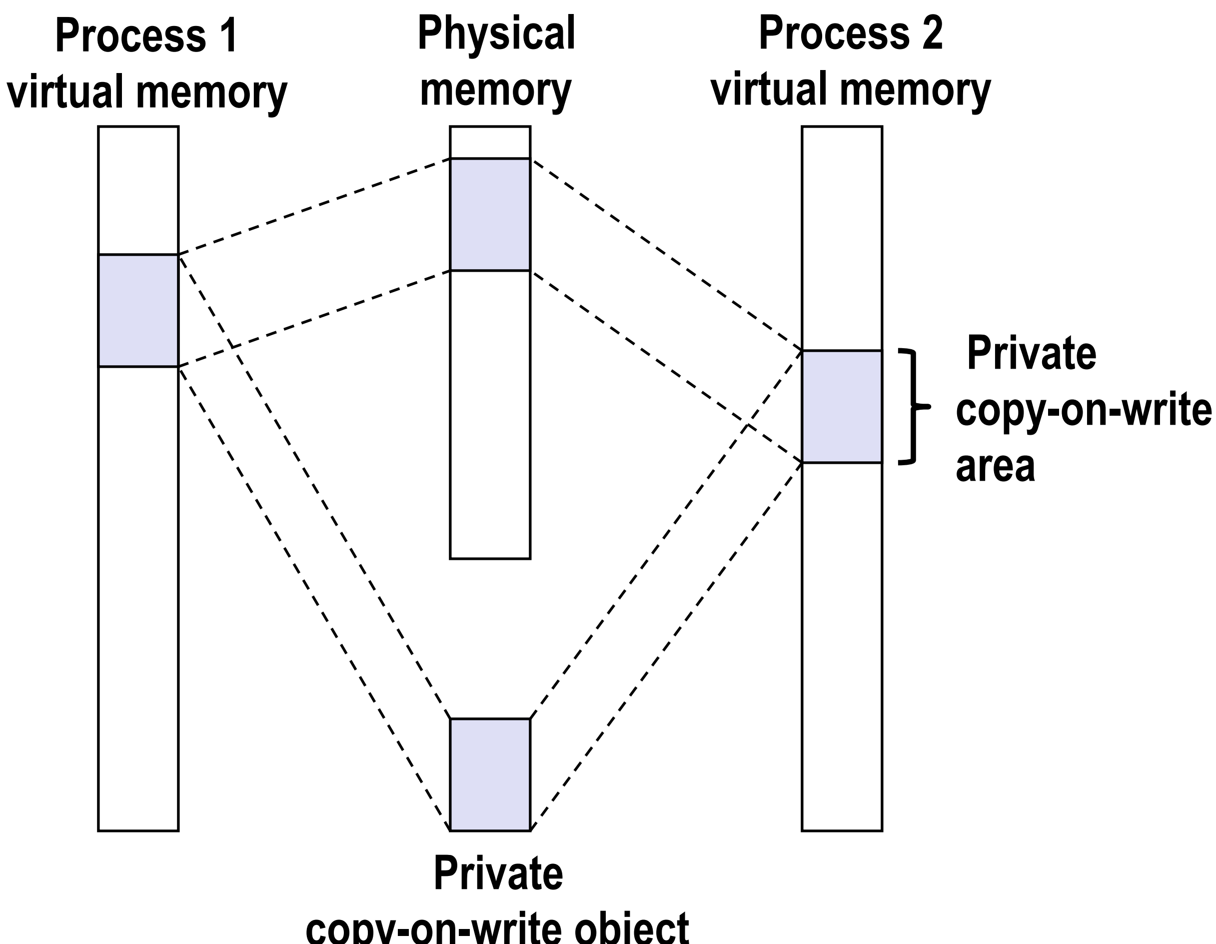
- Process 1 maps the shared object.

+ Sharing Revisited: Shared Objects



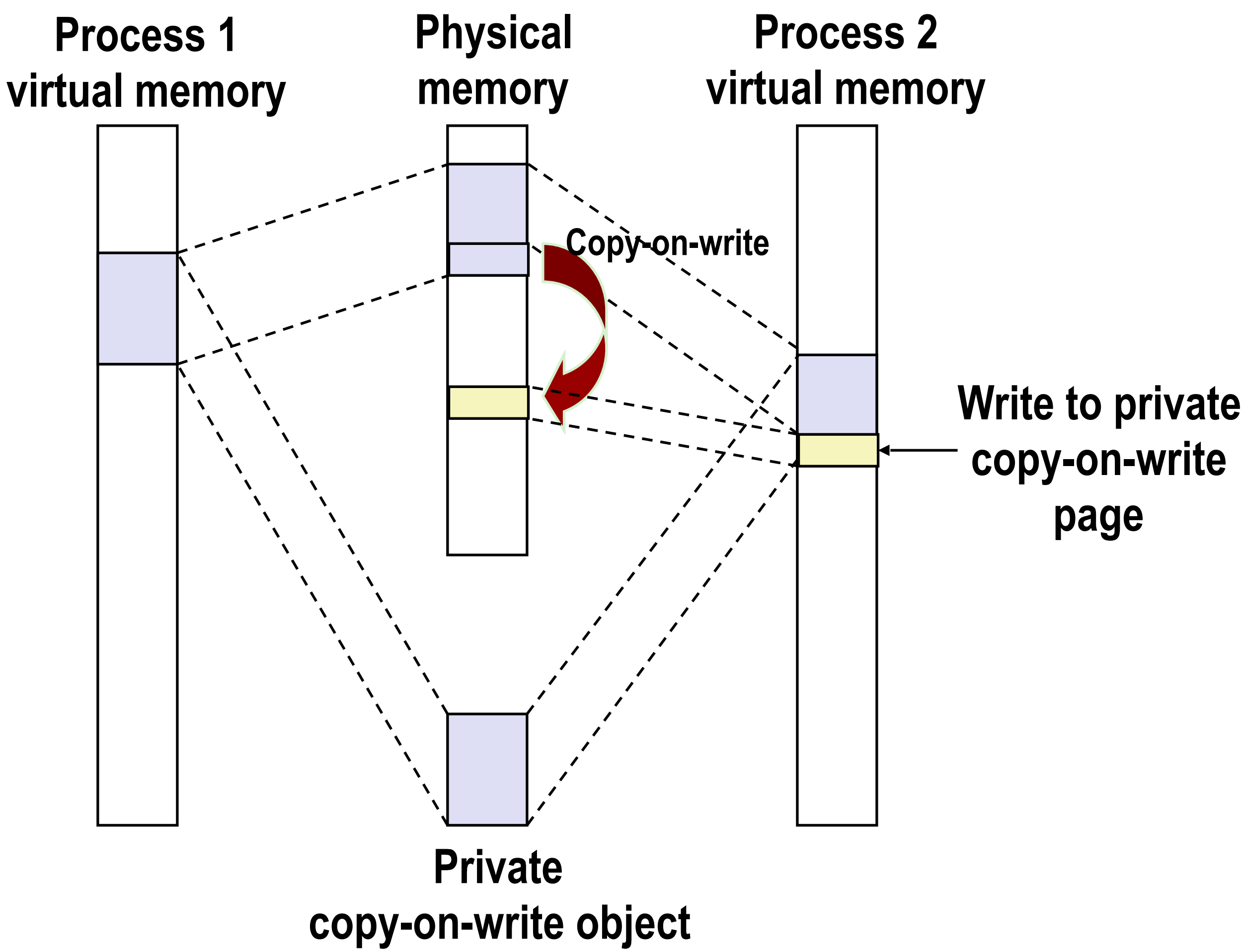
- Process 2 maps the shared object.
- Notice how the virtual addresses can be different

+ Sharing Revisited: Private Copy-on-write (COW) Objects



- **Two processes mapping a *private copy-on-write (COW) object*.**
- **Area flagged as private copy-on-write**
- **PTEs in private areas are flagged as read-only**

Sharing Revisited: Private Copy-on-write (COW) Objects



- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

+ The `fork` Function Revisited

- **VM and memory mapping explain how fork provides private address space for each process.**
- **To create virtual address for new new process**
  - Create exact copy of virtual memory address space
  - Flag virtual memory context in both processes as COW
- **On return, each process has exact copy of virtual memory**
- **Subsequent writes create *new pages* using COW mechanism.**

+ User-Level Memory Mapping

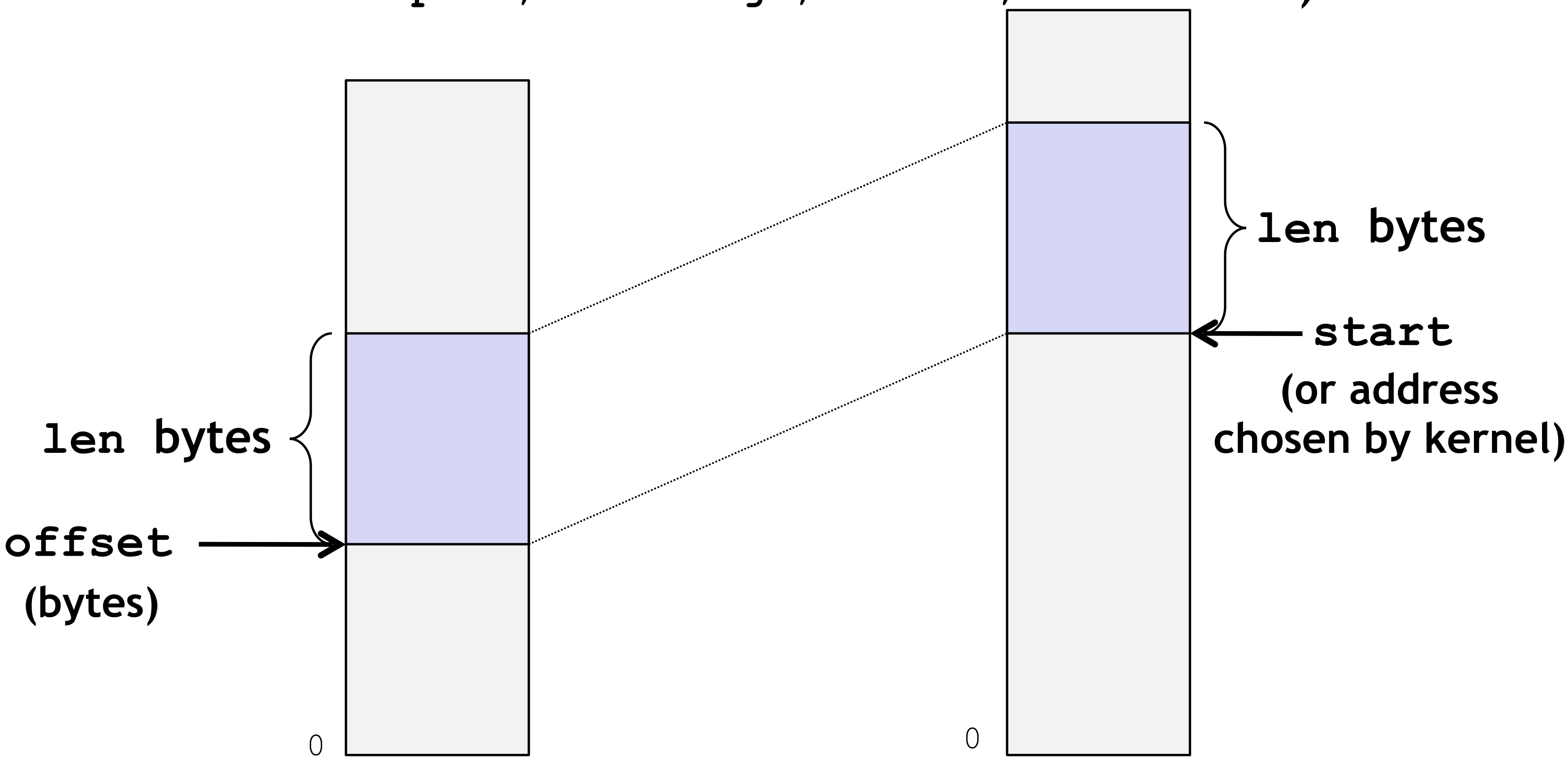
- **`void* mmap(void* start, int len, int prot, int flags, int fd, int offset)`**

- **Map `len` bytes starting at `offset` of the file specified by file description `fd`, preferably at address `start`**
  - `start`: may be 0 for "pick an address"
  - `prot`: PROT_READ, PROT_WRITE, ...
  - `flags`: MAP_PRIVATE, MAP_SHARED, ...

- **Return a pointer to start of mapped area (may not be `start`)**

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



Disk file specified by file descriptor **fd**

Process virtual memory

Example: Using mmap to Copy Files

- **Copying a file to stdout without transferring data to user space**

```
void mmapcopy(int fd, int size)
{

    /* Ptr to memory mapped area */
    char *bufp;

    bufp = mmap(NULL, size,
                PROT_READ,
                MAP_PRIVATE,
                fd, 0);

    write(1, bufp, size);

    return;
}
```

```
/* mmapcopy driver */
int main(int argc, char **argv)
{
    struct stat stat;
    int fd;

    /* Check for required cmd line arg */
    if (argc != 2) {
        printf("usage: %s <filename>\n",
               argv[0]);
        exit(0);
    }

    /* Copy input file to stdout */
    fd = open(argv[1], O_RDONLY, 0);
    fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    exit(0);
}
```