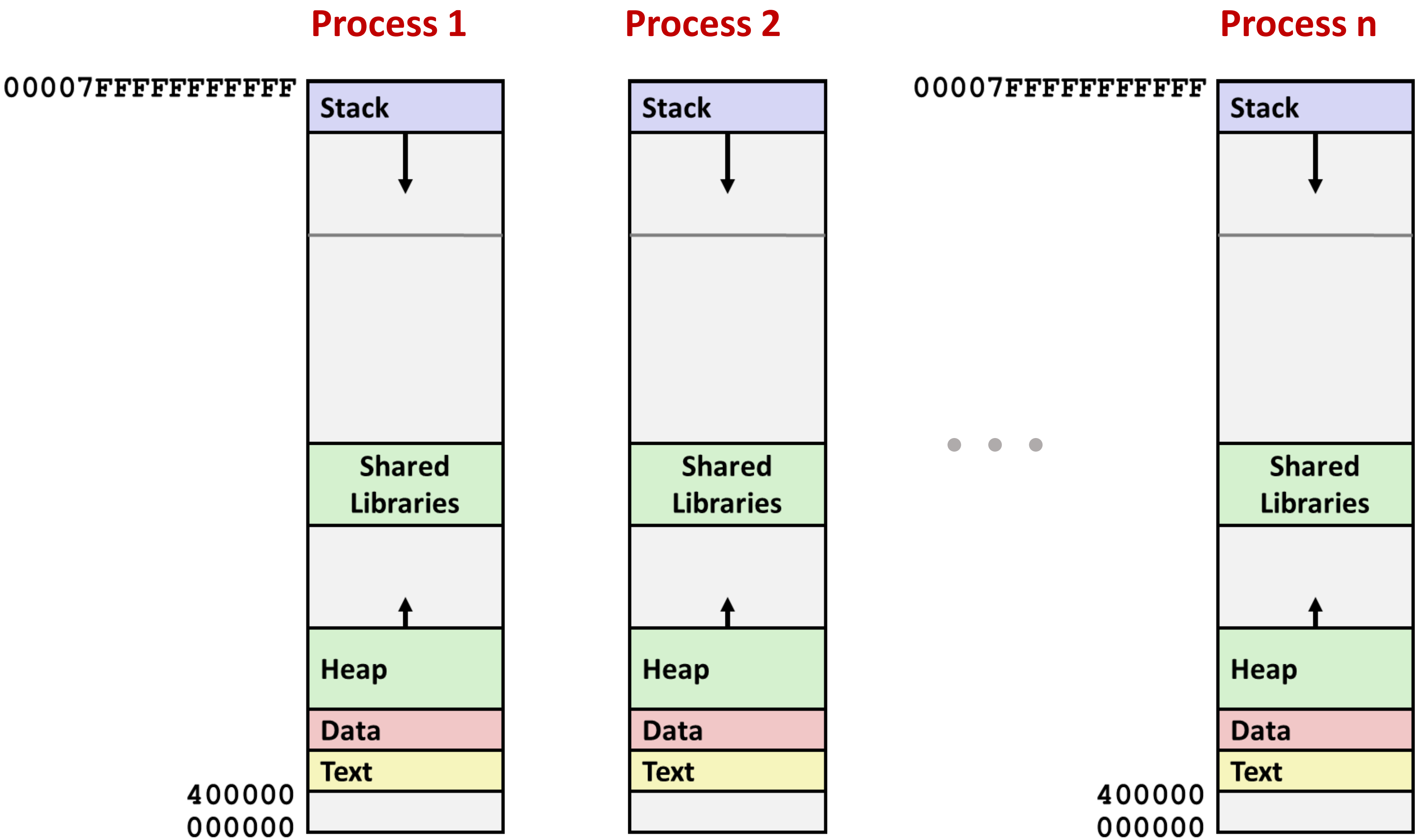
Virtual Memory
(Part 1)

HGU

Contents

- Address Spaces
- VM as a Tool for Caching
- VM as a Tool for Memory Management and Protection
- Address Translation
- Memory Mapping

How Does This Work?!



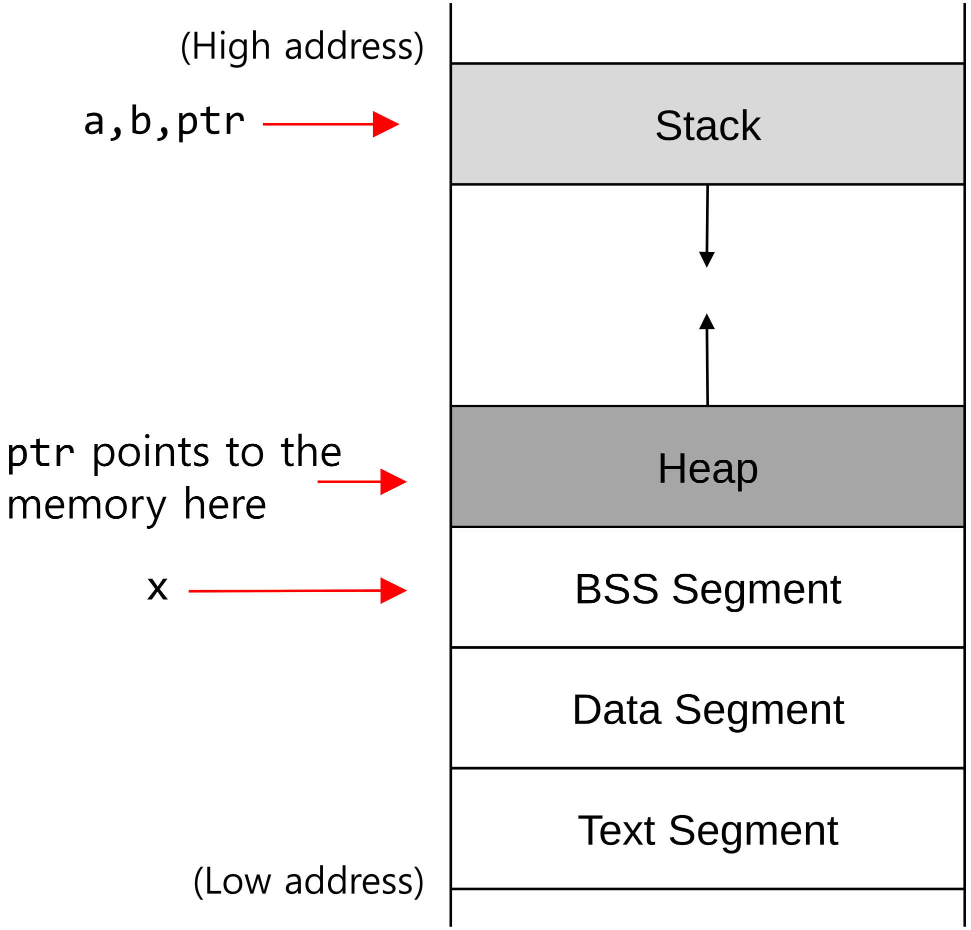
Solution: Virtual Memory !

Process Memory

memprog.c

```
int func(int a, int b)
{
    return (a + b) + (a - b);
}

int main()
{
    int a = 2, b = 4;
    static int x;
    int *ptr = (int *) malloc(2 * sizeof(int));

    ptr[0] = 5;
    ptr[1] = 6;

    x = ptr[0] + ptr[1];
    b = func(a, b);

    printf("b=%d, x=%d\n", b, x);

    free(ptr);

    return 1;
}
```

Set break point at Line 15



Looking at Process Memory using GDB

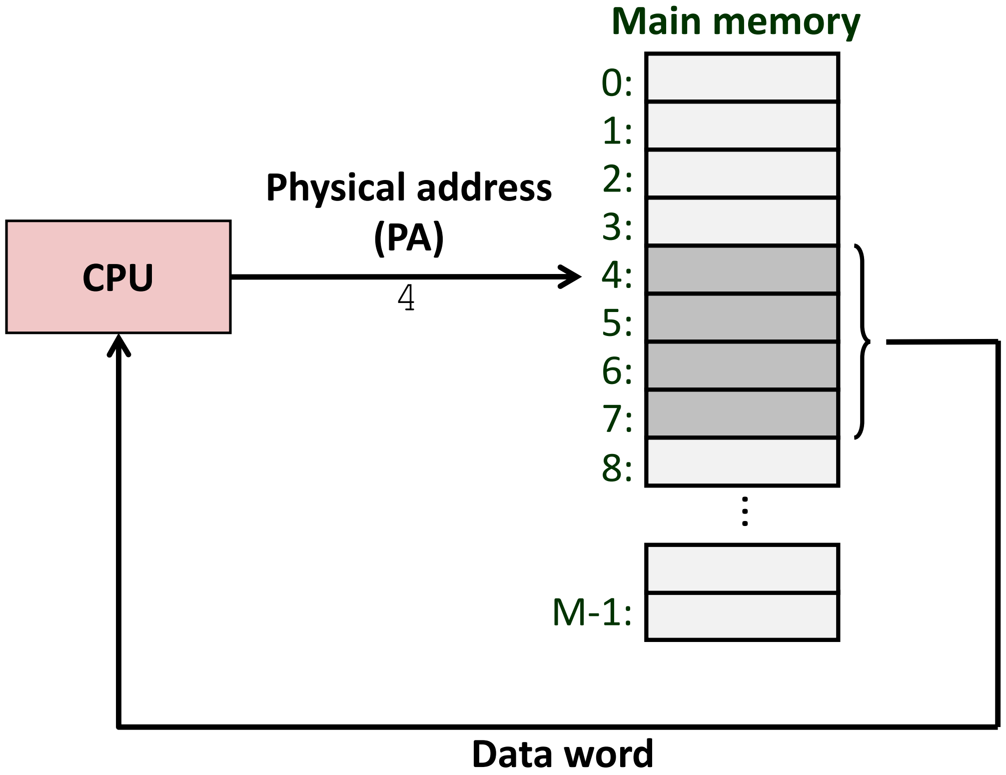
```
yunmin@peace:~/ch11$ gcc memprog.c -o memprog -g
yunmin@peace:~/ch11$ gdb ./memprog
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./memprog...done.
(gdb) b 15
Breakpoint 1 at 0x4005f6: file memprog.c, line 15.
(gdb) r
Starting program: /home/yunmin/ch11/memprog

Breakpoint 1, main () at memprog.c:15
15          ptr[0] = 5;
(gdb) p a
$1 = 2
(gdb) p b
$2 = 4
(gdb) p &a
$3 = (int *) 0x7fffffffe0e0
(gdb) info reg
rax            0x602010 6299664
rbx            0x0      0
rcx            0x7ffff7dd1b20   140737351850784
rdx            0x602010 6299664
rsi            0x602020 6299680
rdi            0x7ffff7dd1b20   140737351850784
rbp            0x7fffffffe0f0   0x7fffffffe0f0
rsp            0x7fffffffe0e0   0x7fffffffe0e0
r8             0x602000 6299648
```

```
(gdb) disassemble
Dump of assembler code for function main:
   0x00000000004005d2 <+0>:     push   %rbp
   0x00000000004005d3 <+1>:     mov    %rsp,%rbp
   0x00000000004005d6 <+4>:     sub    $0x10,%rsp
   0x00000000004005da <+8>:     movl   $0x2,-0x10(%rbp)
   0x00000000004005e1 <+15>:    movl   $0x4,-0xc(%rbp)
   0x00000000004005e8 <+22>:    mov    $0x8,%edi
   0x00000000004005ed <+27>:    callq  0x4004a0 <malloc@plt>
   0x00000000004005f2 <+32>:    mov    %rax,-0x8(%rbp)
=> 0x00000000004005f6 <+36>:    mov    -0x8(%rbp),%rax
   0x00000000004005fa <+40>:    movl   $0x5,(%rax)
   0x0000000000400600 <+46>:    mov    -0x8(%rbp),%rax
   0x0000000000400604 <+50>:    add    $0x4,%rax
```

```
(gdb) n
16          ptr[1] = 6;
(gdb) n
18          x = ptr[0] + ptr[1];
(gdb) n
19          b = func(a, b);
(gdb) s
func (a=2, b=4) at memprog.c:6
6           return (a + b) + (a - b);
(gdb) n
7       }
(gdb) info proc mappings
process 18473
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/yunmin/ch11/memprog
            0x600000           0x601000     0x1000        0x0 /home/yunmin/ch11/memprog
            0x601000           0x602000     0x1000     0x1000 /home/yunmin/ch11/memprog
            0x602000           0x623000    0x21000        0x0 [heap]
      0x7ffff7a0d000     0x7ffff7bcd000   0x1c0000        0x0 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7bcd000     0x7ffff7dcd000   0x200000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dcd000     0x7ffff7dd1000     0x4000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dd1000     0x7ffff7dd3000     0x2000   0x1c4000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dd3000     0x7ffff7dd7000     0x4000        0x0
      0x7ffff7dd7000     0x7ffff7dfd000    0x26000        0x0 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7fd3000     0x7ffff7fd6000     0x3000        0x0
      0x7ffff7ff7000     0x7ffff7ffa000     0x3000        0x0 [vvar]
      0x7ffff7ffa000     0x7ffff7ffc000     0x2000        0x0 [vdso]
```

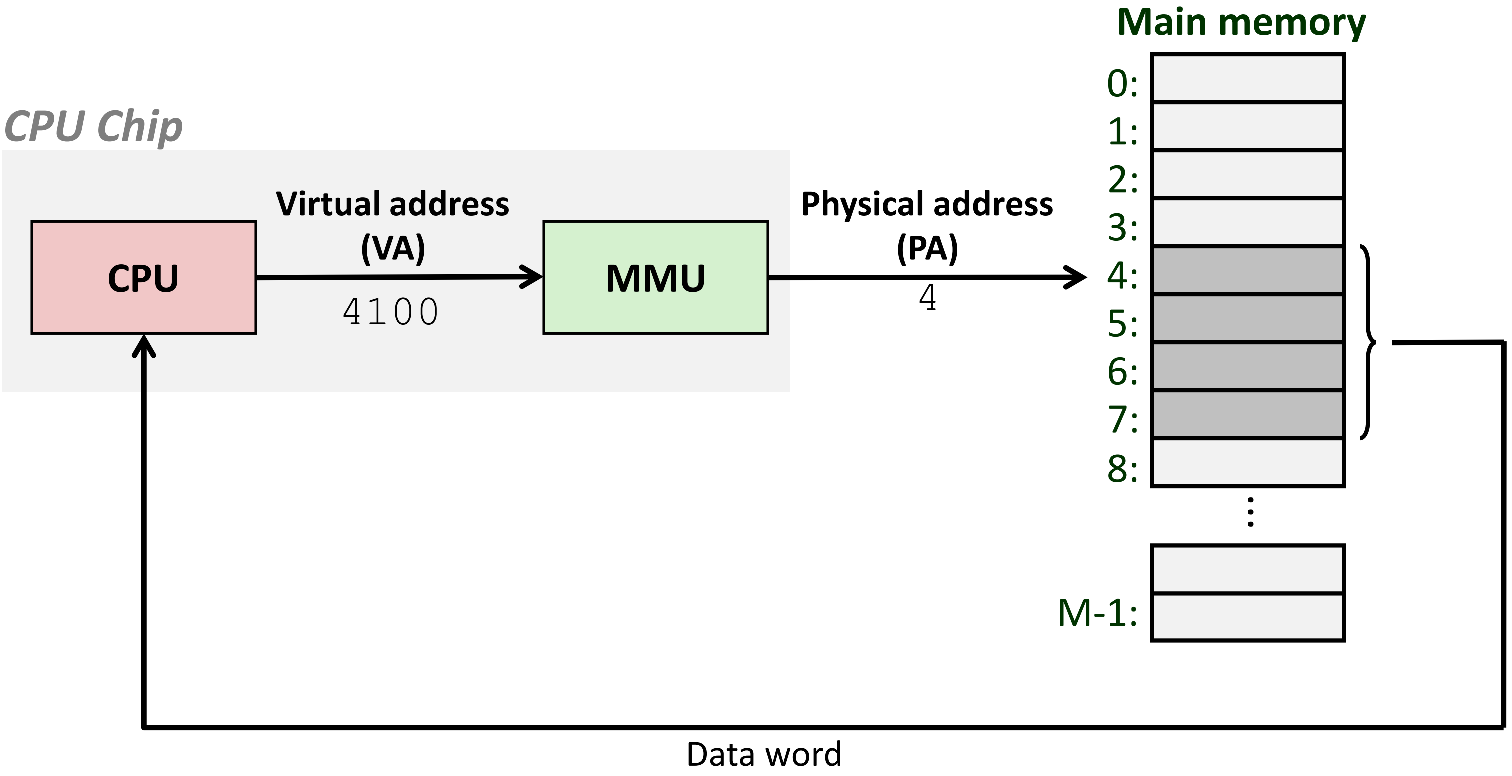
A System Using Physical Addressing

- Used in “simple” systems like embedded microcontrollers in devices like 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:
  $\{0, 1, 2, 3 \ldots \}$

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

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

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

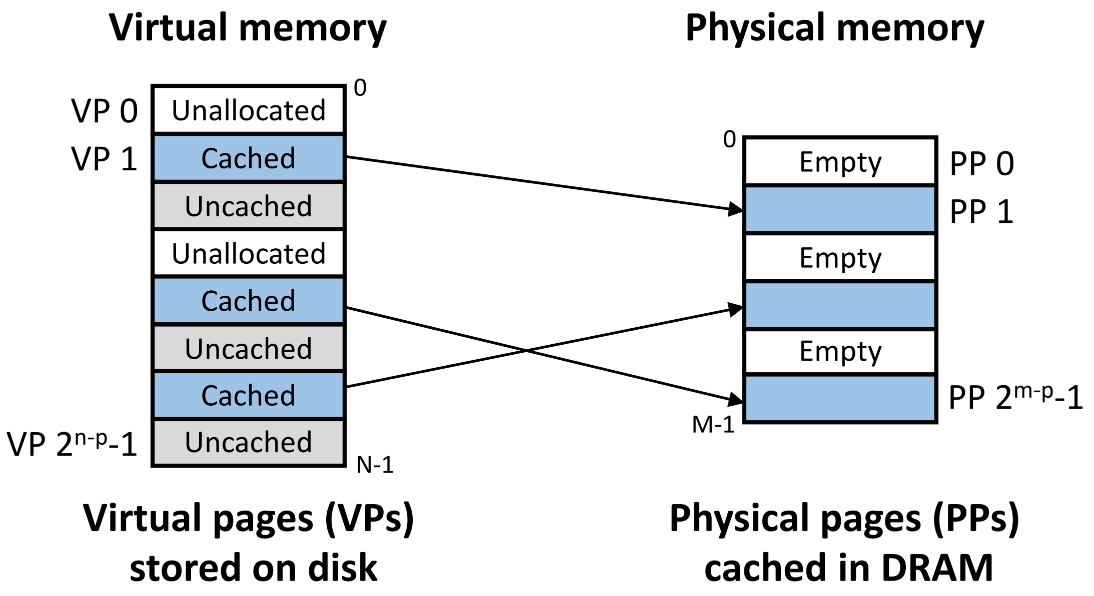
- Isolates address spaces
  - One process can’t interfere with another’s memory
  - User program cannot access privileged kernel information and code

Contents

- Address Spaces
- VM as a Tool for Caching
- VM as a Tool for Memory Management and Protection
- Address Translation
- Memory Mapping

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 cache)
  - These cache blocks are called pages (size is $P = 2^p$ bytes)

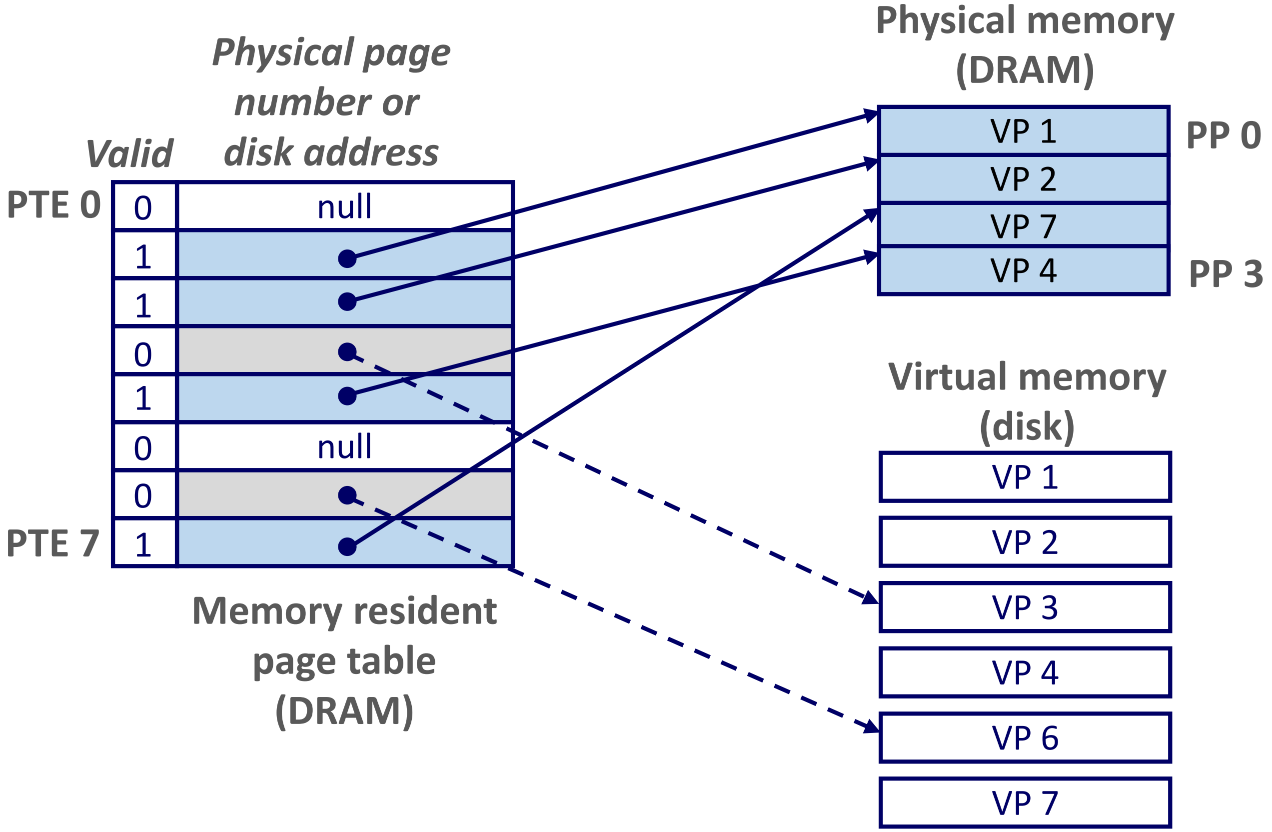


DRAM Cache Organization

- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about 10x slower than SRAM
  - Disk is about 10,000x slower than DRAM
  - Time to load block from disk > 1ms (> 1 million clock cycles)
    - CPU can do a lot of computation during that time
- Consequences
  - Large page (block) size: typically 4 KB
    - Linux "huge pages" are 2 MB (default) to 1 GB
  - Fully associative. Why?
    - Any VP can be placed in any PP
    - Requires a "large" mapping function – different from cache memories
  - Highly sophisticated, expensive replacement algorithms. Why?
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through. Why?

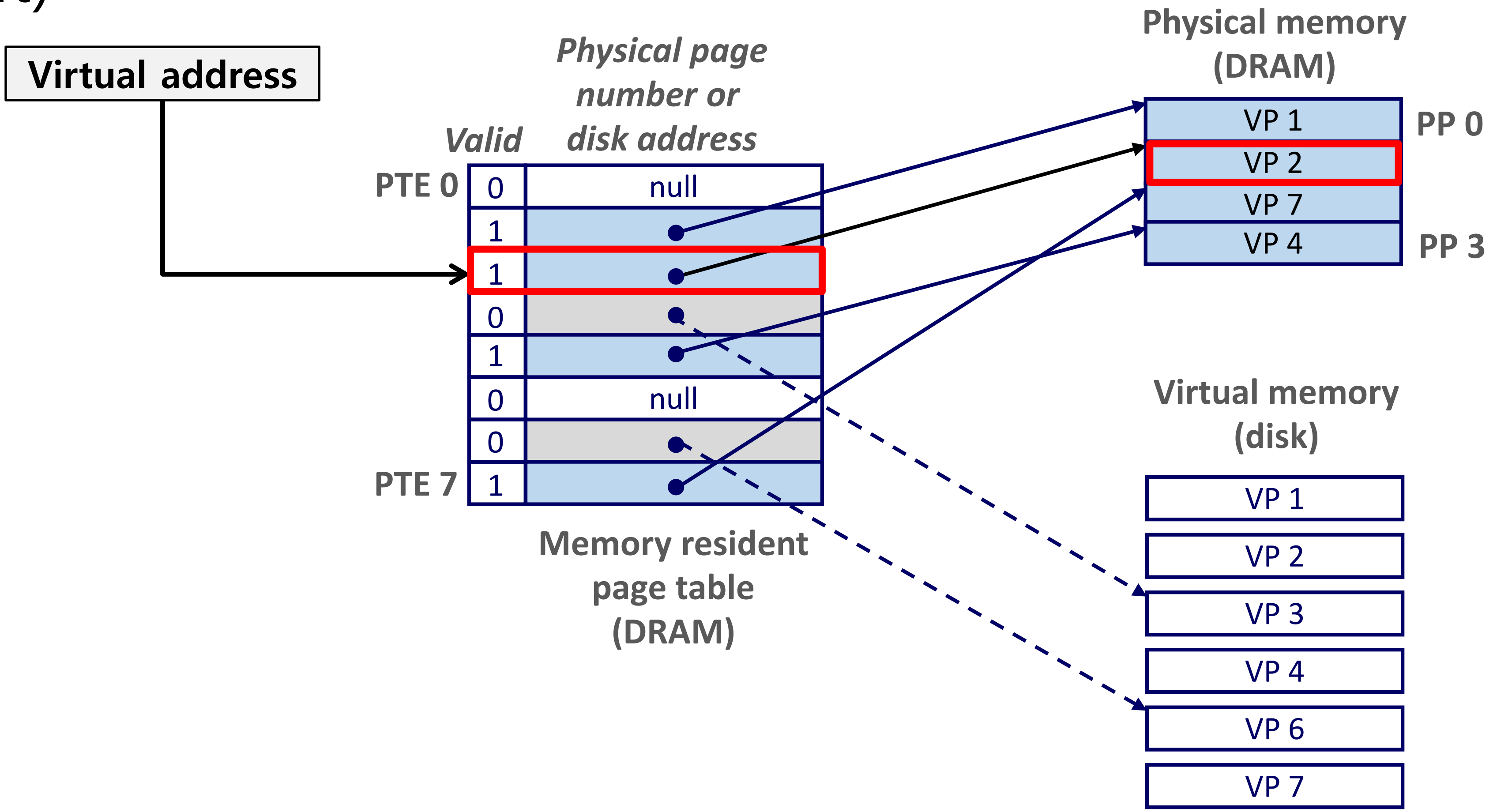
Enabling Data Structure: Page Table

- 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



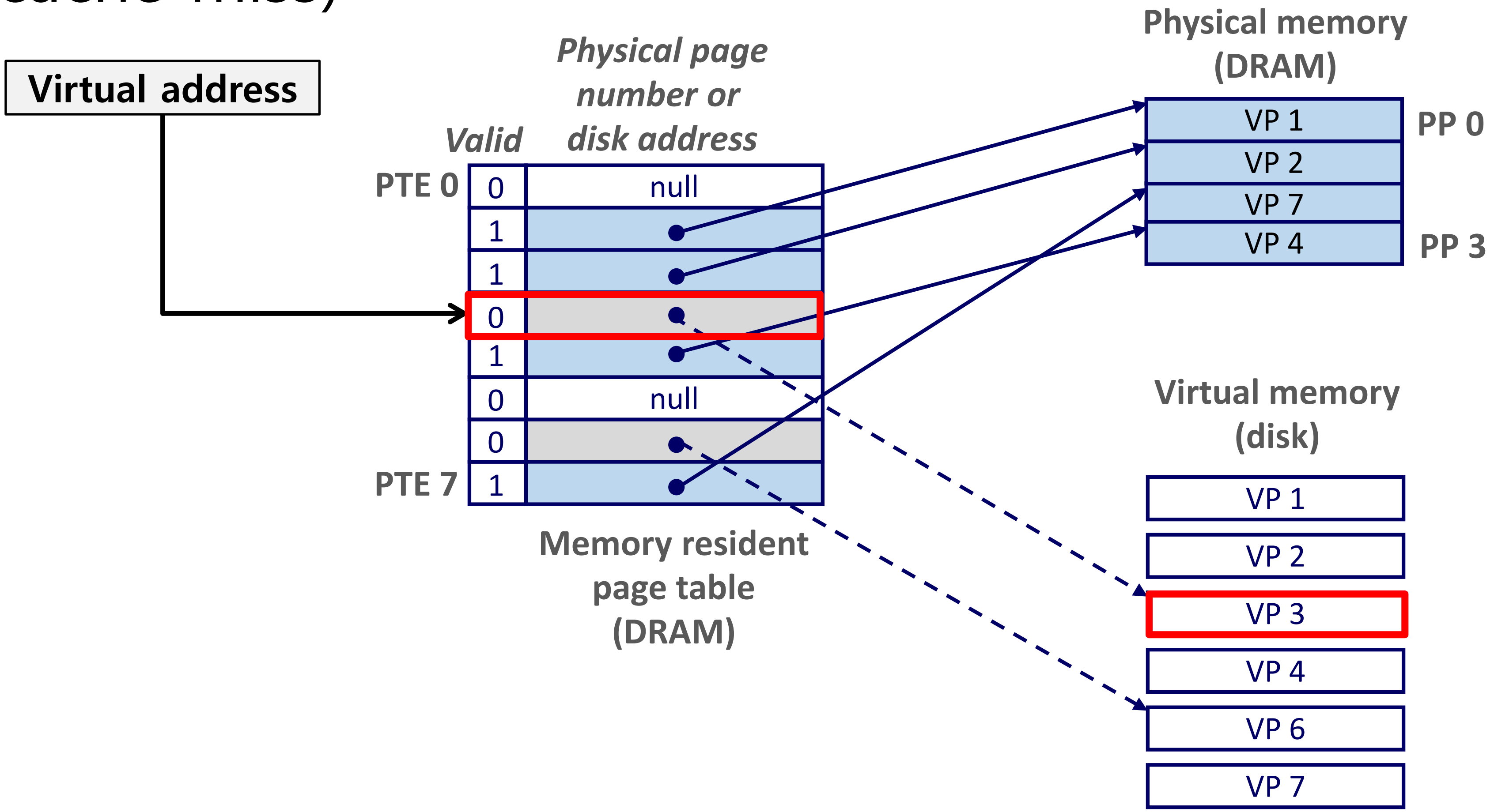
Page Hit

- 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)



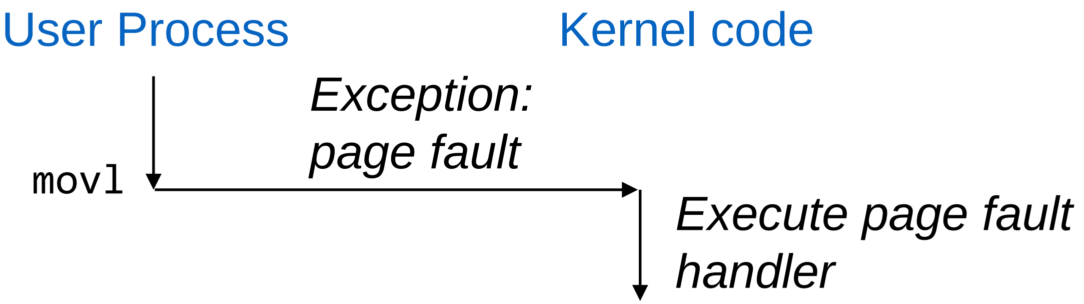
Triggering a Page Fault

- User writes to memory location

```
80483b7:       c7 05 10 9d 04 08 0d   movl    $0xd,0x8049d10
```

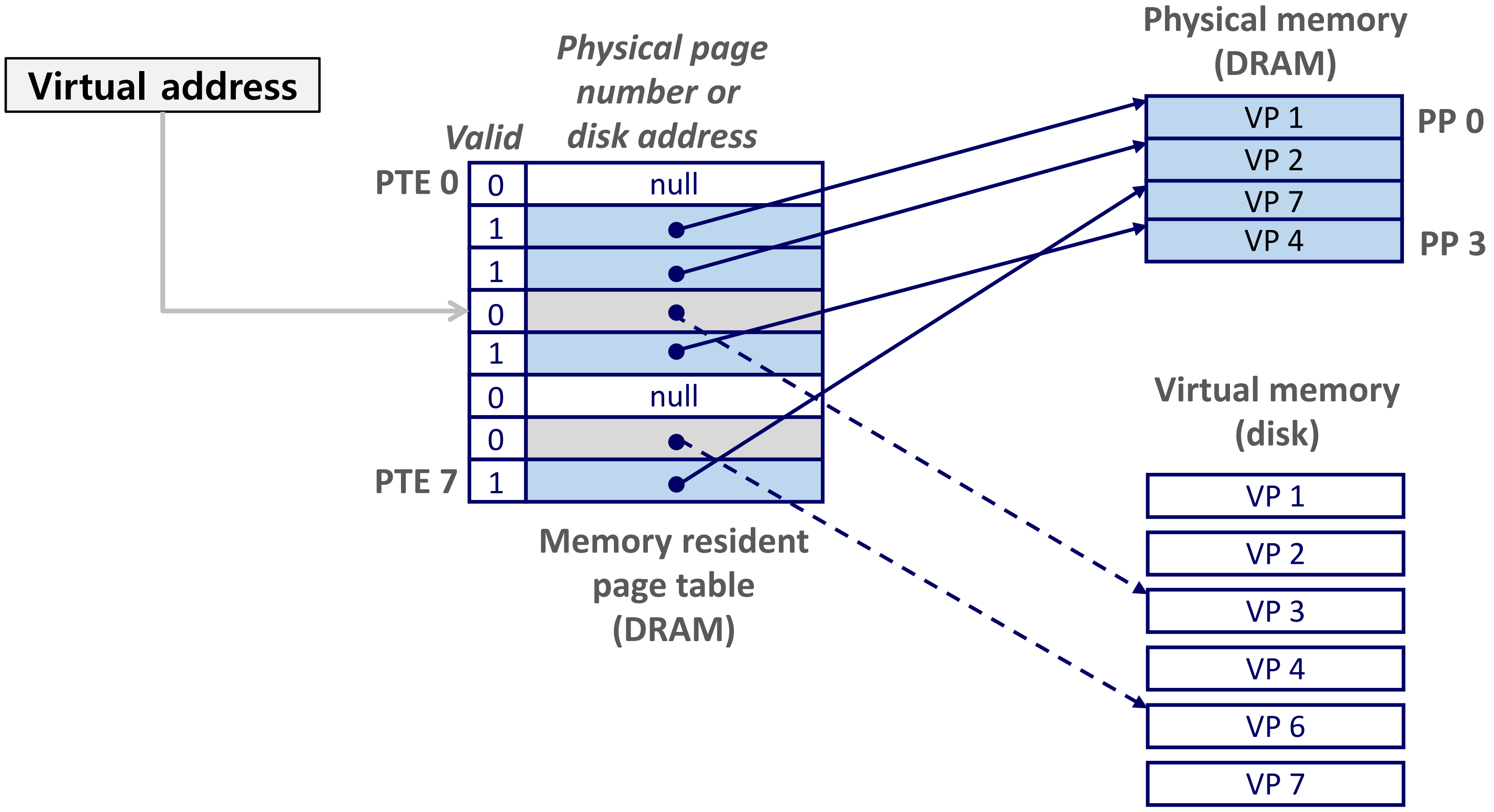
- That portion (page) of user's memory is currently on disk
- MMU triggers page fault exception
  - (More details in later lecture)
  - Raise privilege level to supervisor mode
  - Causes procedure call to software page fault handler

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```



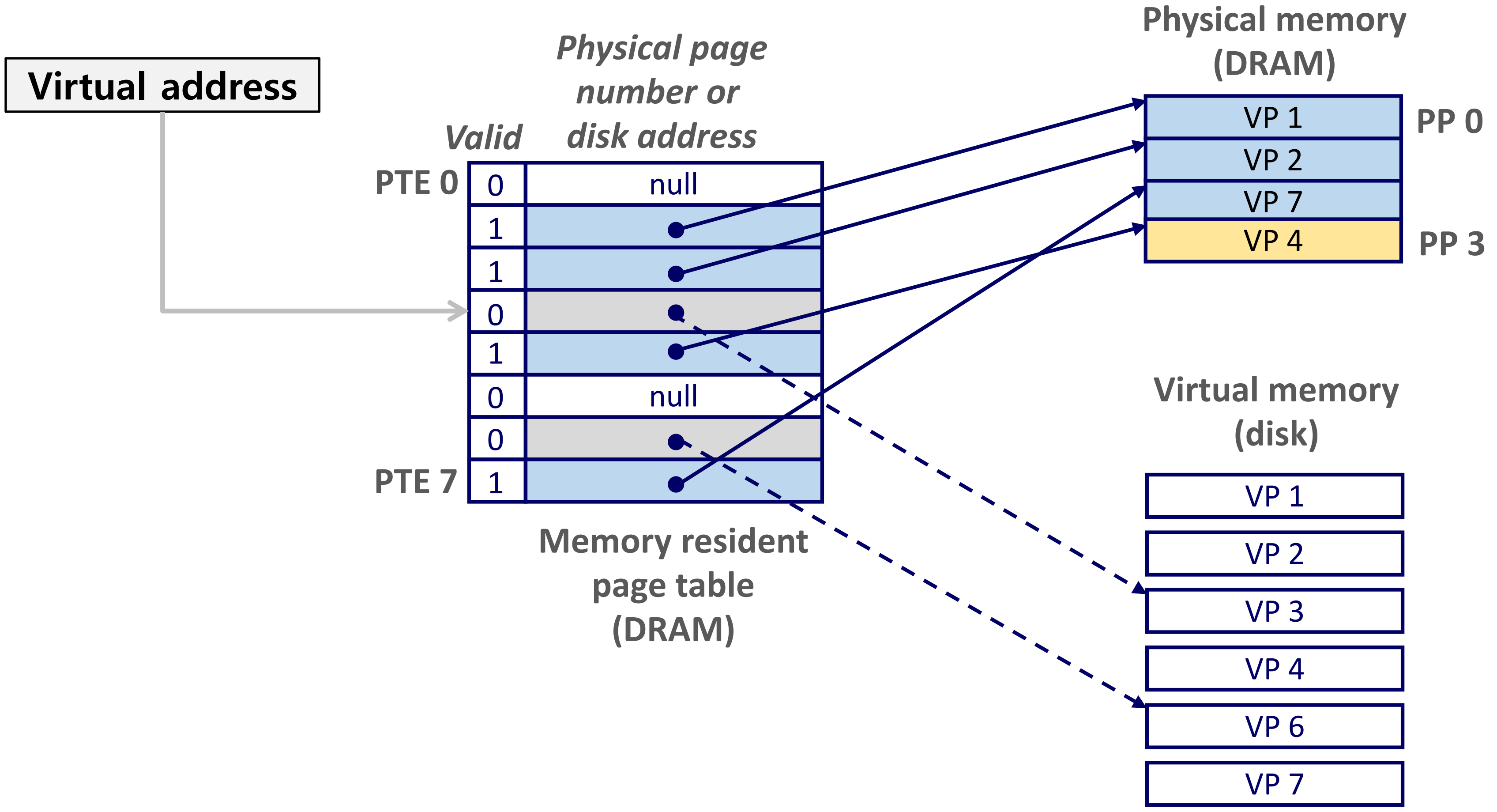
Handling Page Fault

- Page miss causes page fault (an exception)



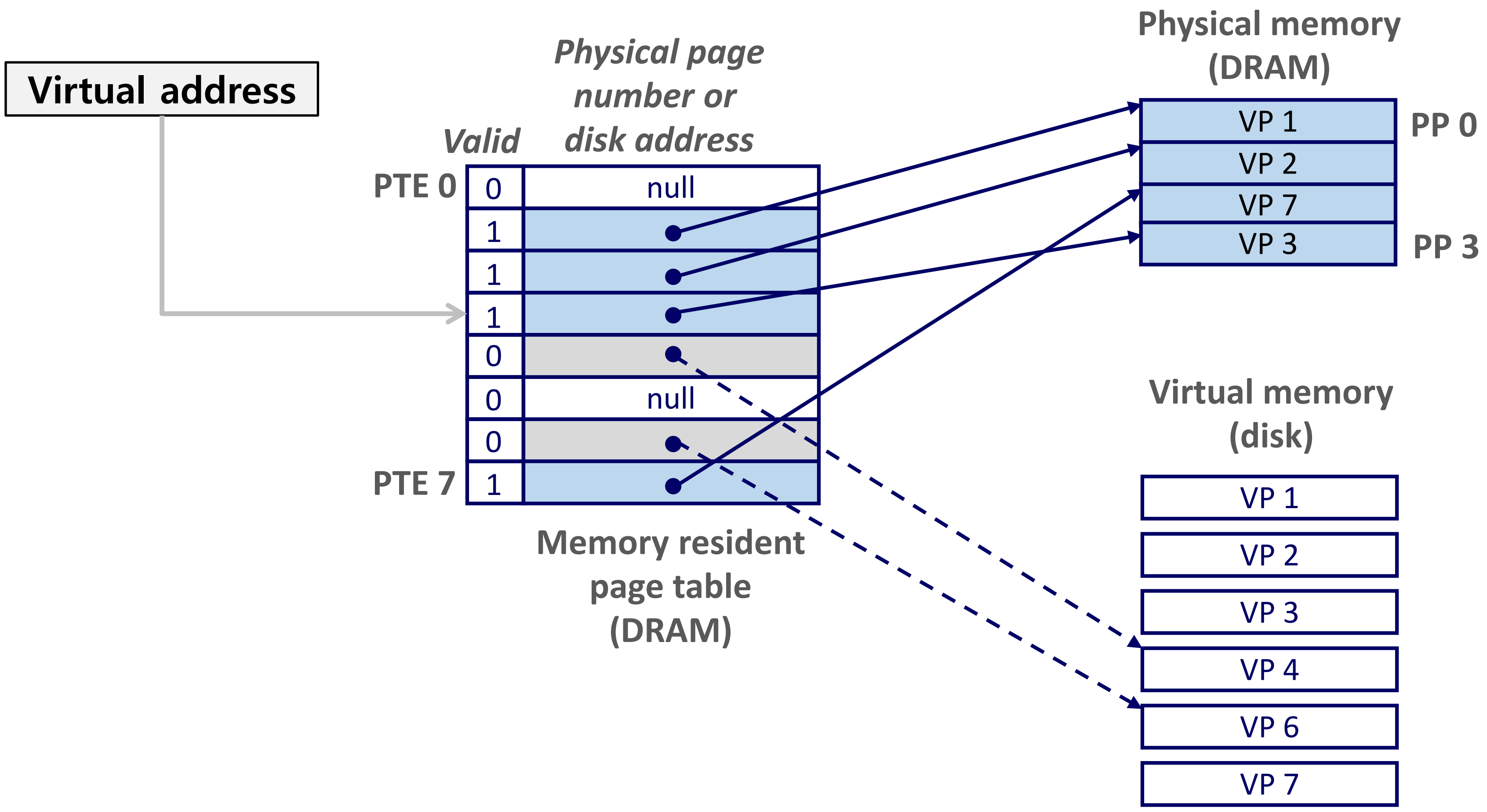
Handling Page Fault

- Page fault handler selects a victim to be evicted (here VP 4)



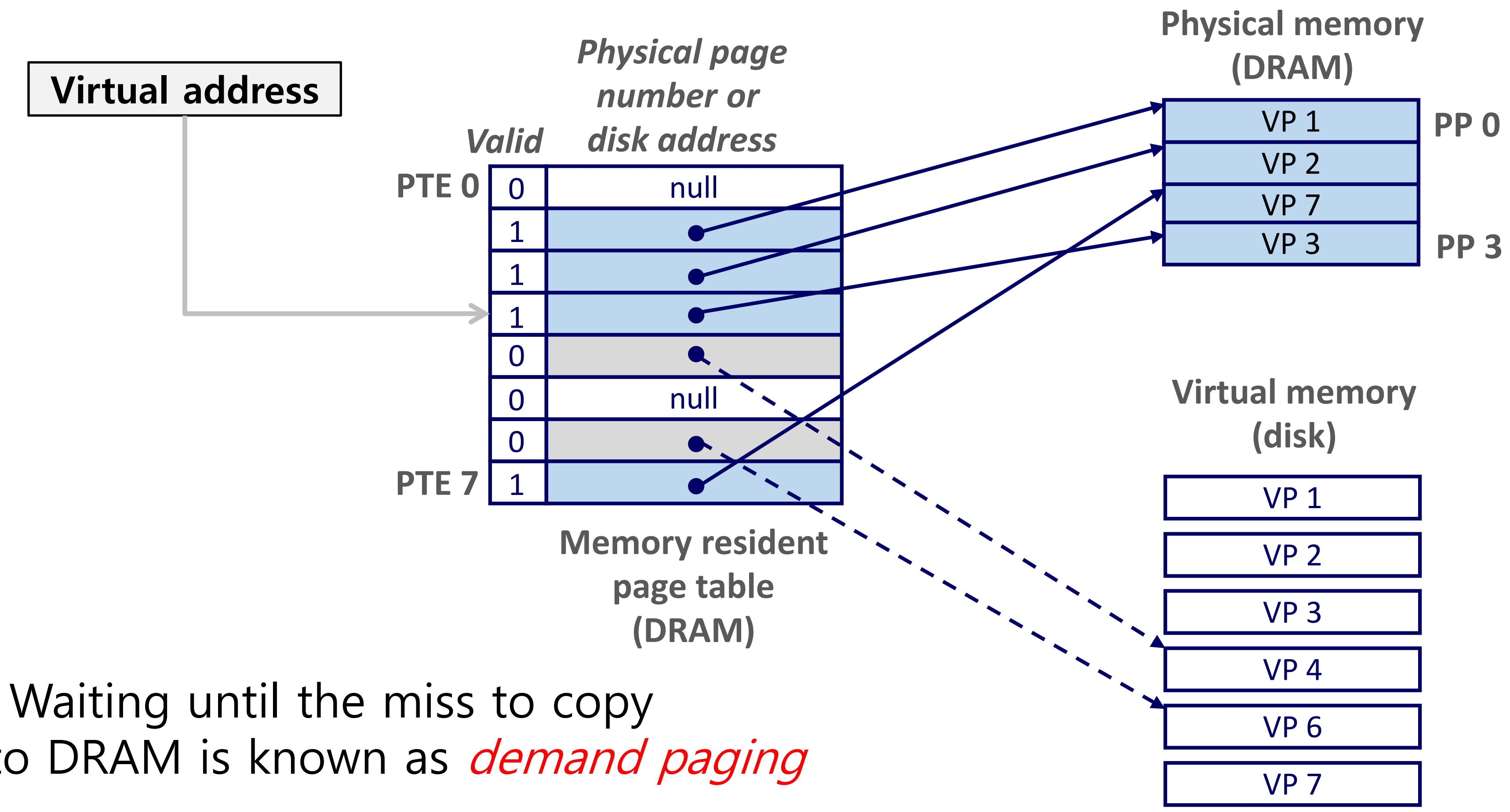
Handling Page Fault

- Page fault handler selects a victim to be evicted (here VP 4)



Handling Page Fault

- Offending instruction is restarted: page hit!



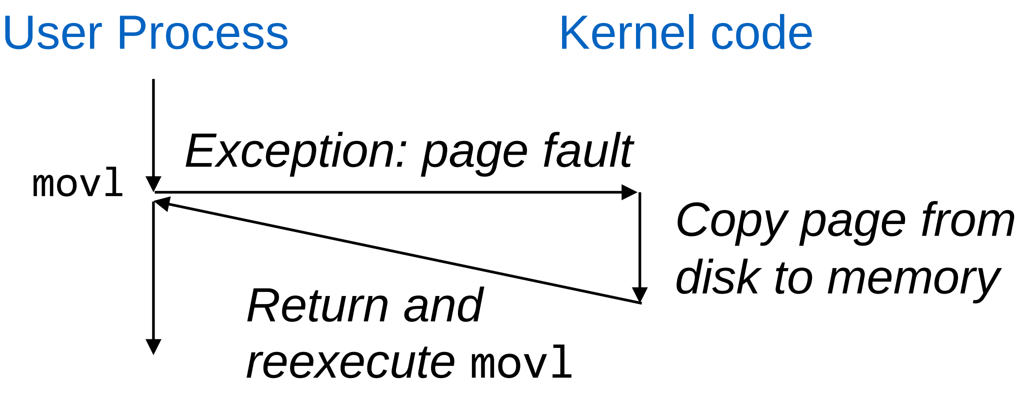
Key point: Waiting until the miss to copy the page to DRAM is known as *demand paging*

Completing Page Fault

- Page fault handler executes return from interrupt (`iret`) instruction
  - Like `ret` instruction, but also restores privilege level
  - Return to instruction that caused fault
  - But, this time there is no page fault

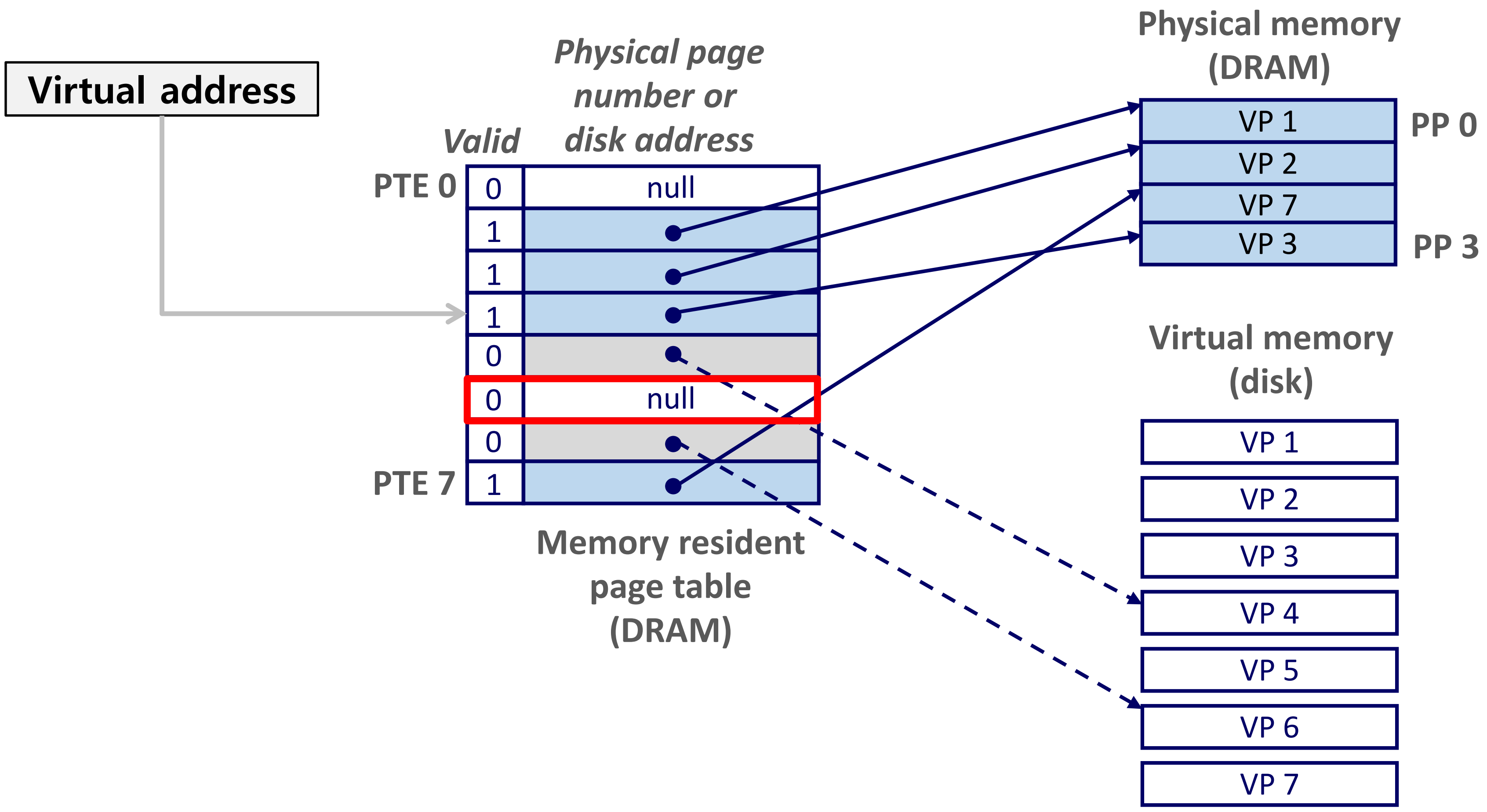
```
80483b7:      c7 05 10 9d 04 08 0d   movl    $0xd,0x8049d10
```

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```



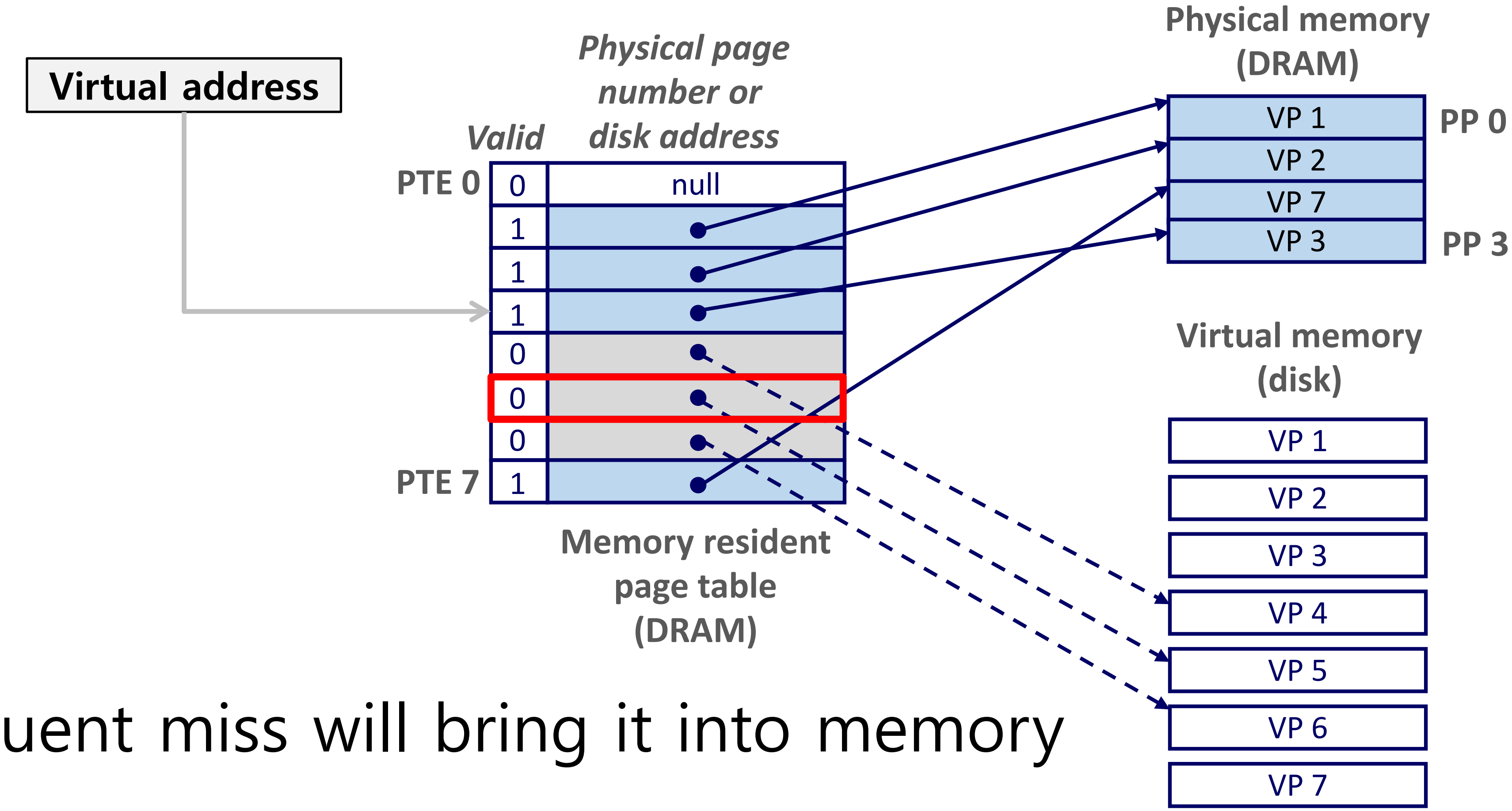
Allocating Pages

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



Allocating Pages

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



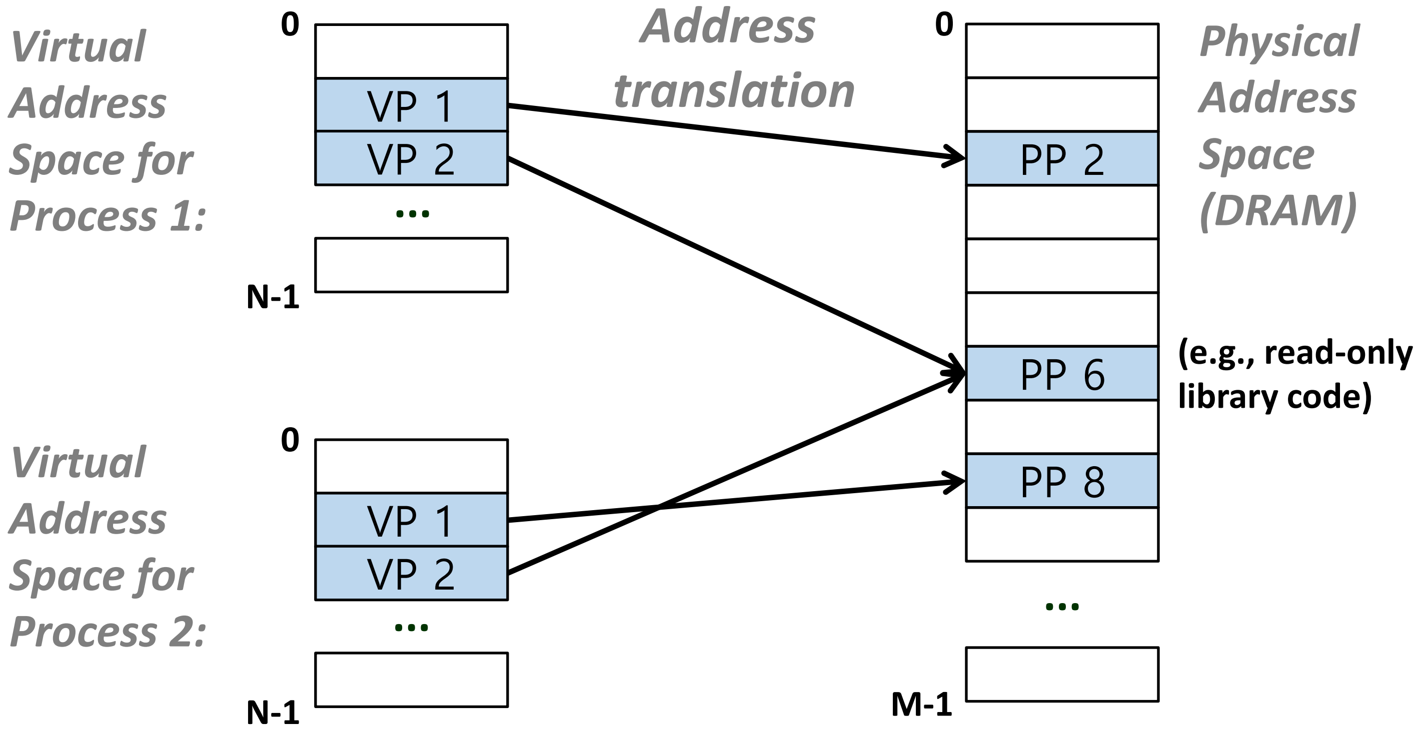
- Subsequent miss will bring it into memory

Contents

- Address Spaces
- VM as a Tool for Caching
- VM as a Tool for Memory Management and Protection
- Address Translation
- Memory Mapping

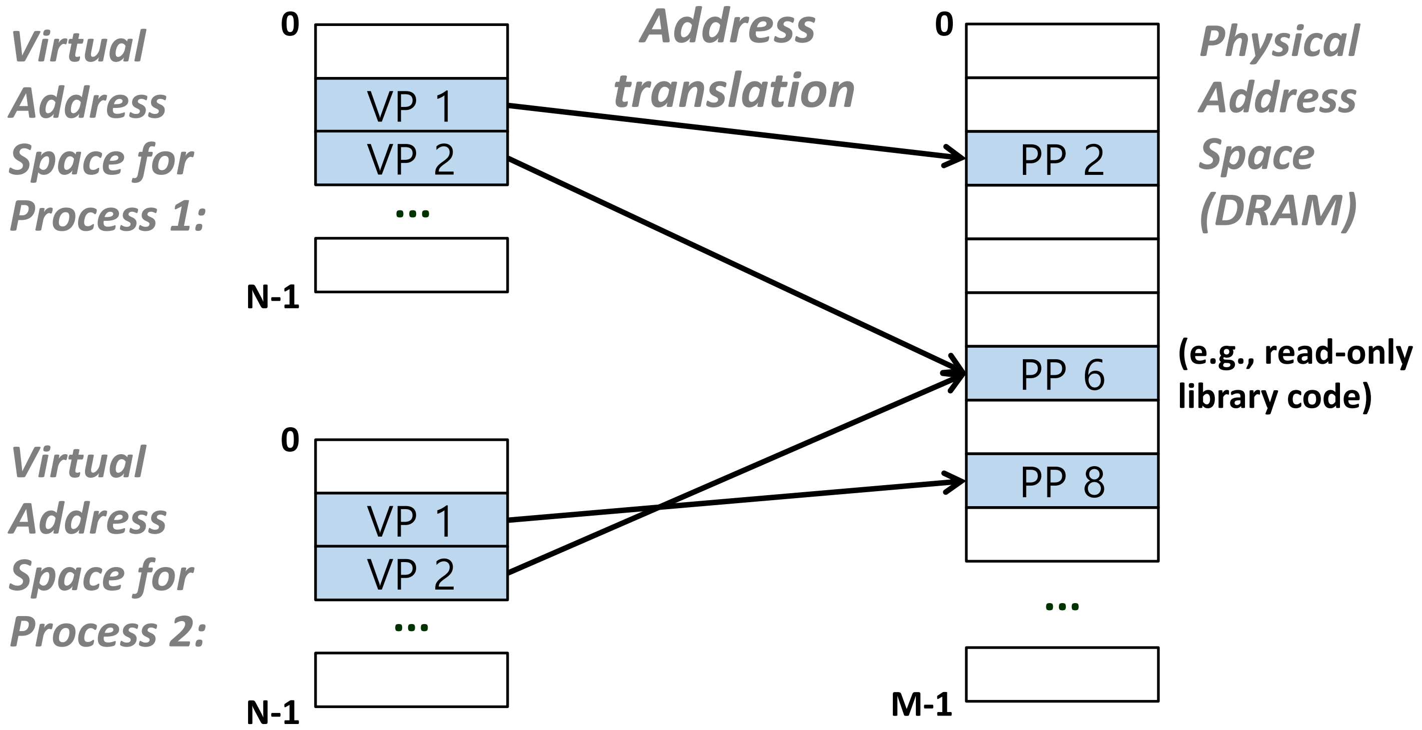
VM as a Tool for Memory Management

- 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
    - Well-chosen mappings can improve locality



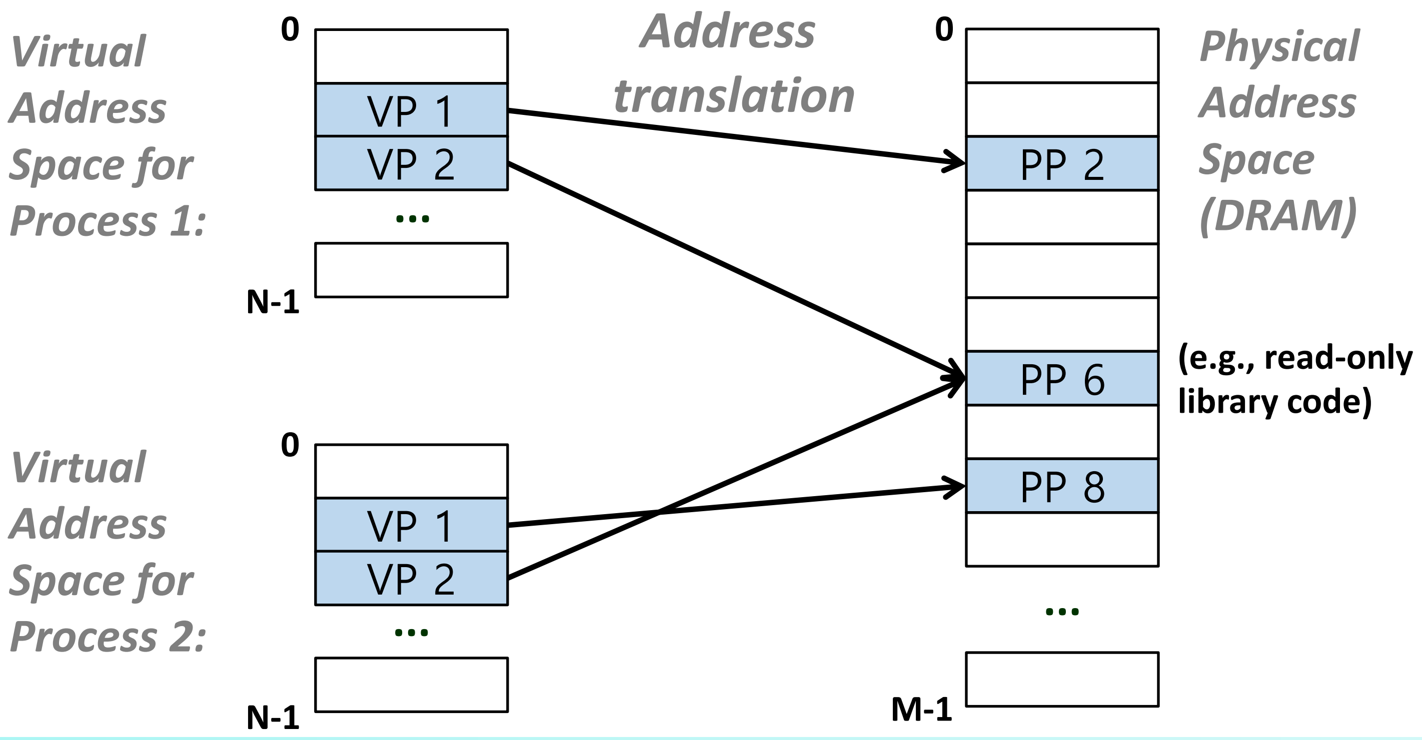
VM as a Tool for Memory Management

- 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
  - Can allocate the same virtual addresses on the heap for multiple processes



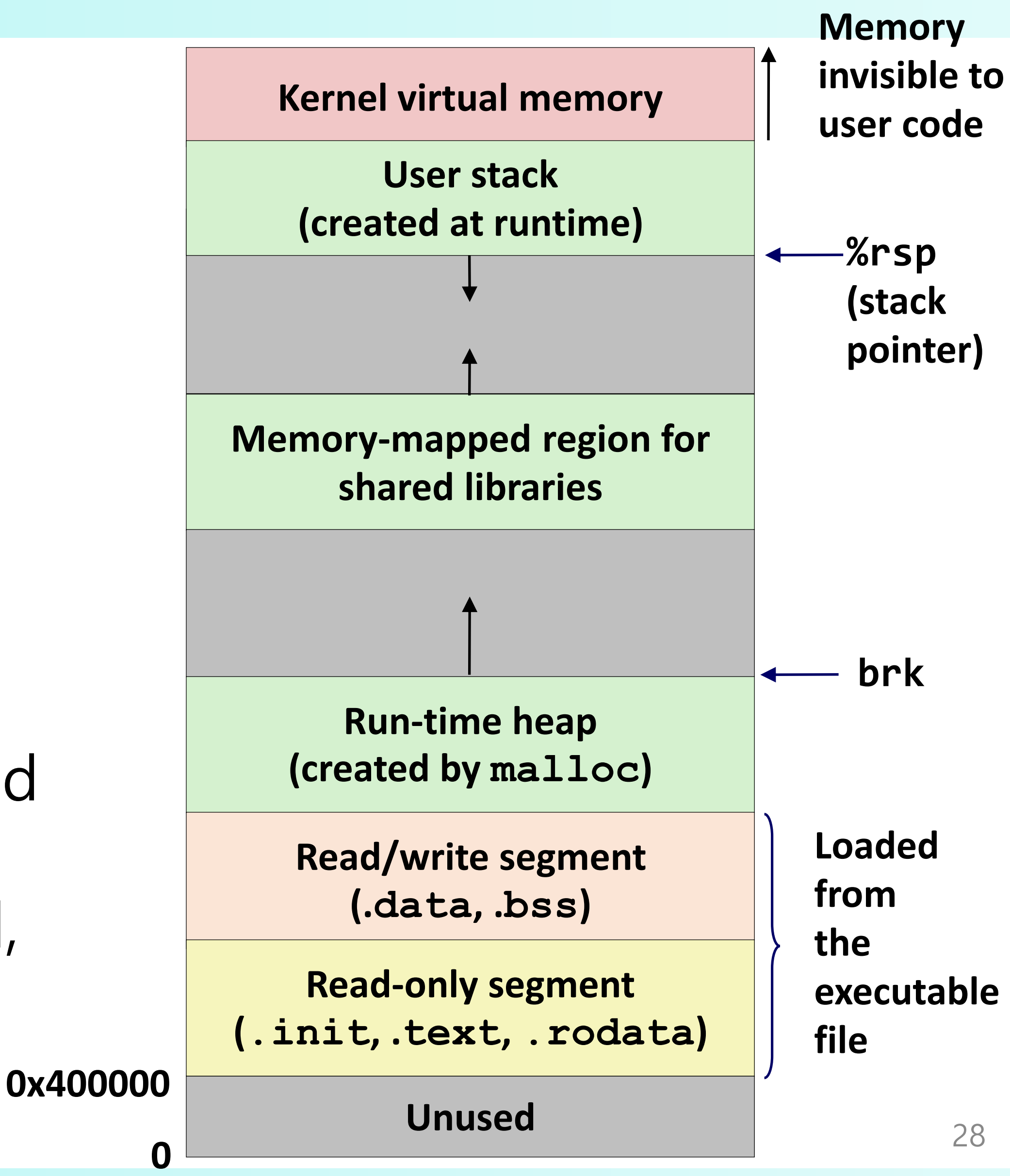
VM as a Tool for Memory Management

- 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

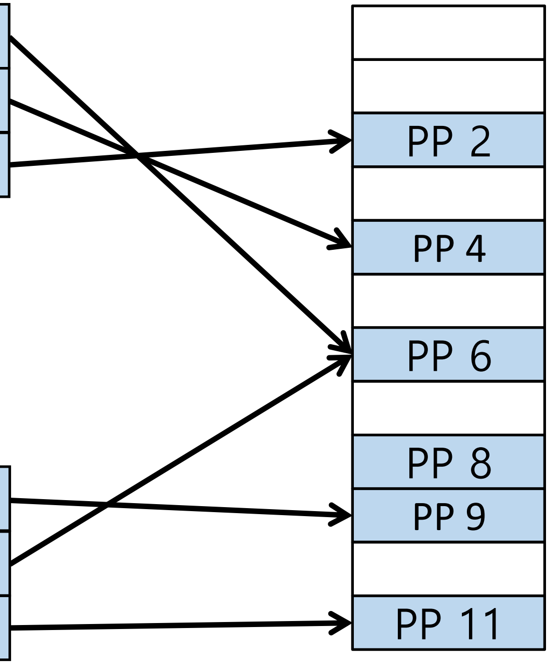
- Extend page table entries (PTEs) with permission bits
- MMU checks these bits on each access

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	Yes	Yes	PP 9
VP 1:	No	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	Yes	PP 11

Physical Address Space



SUP: requires kernel mode

Contents

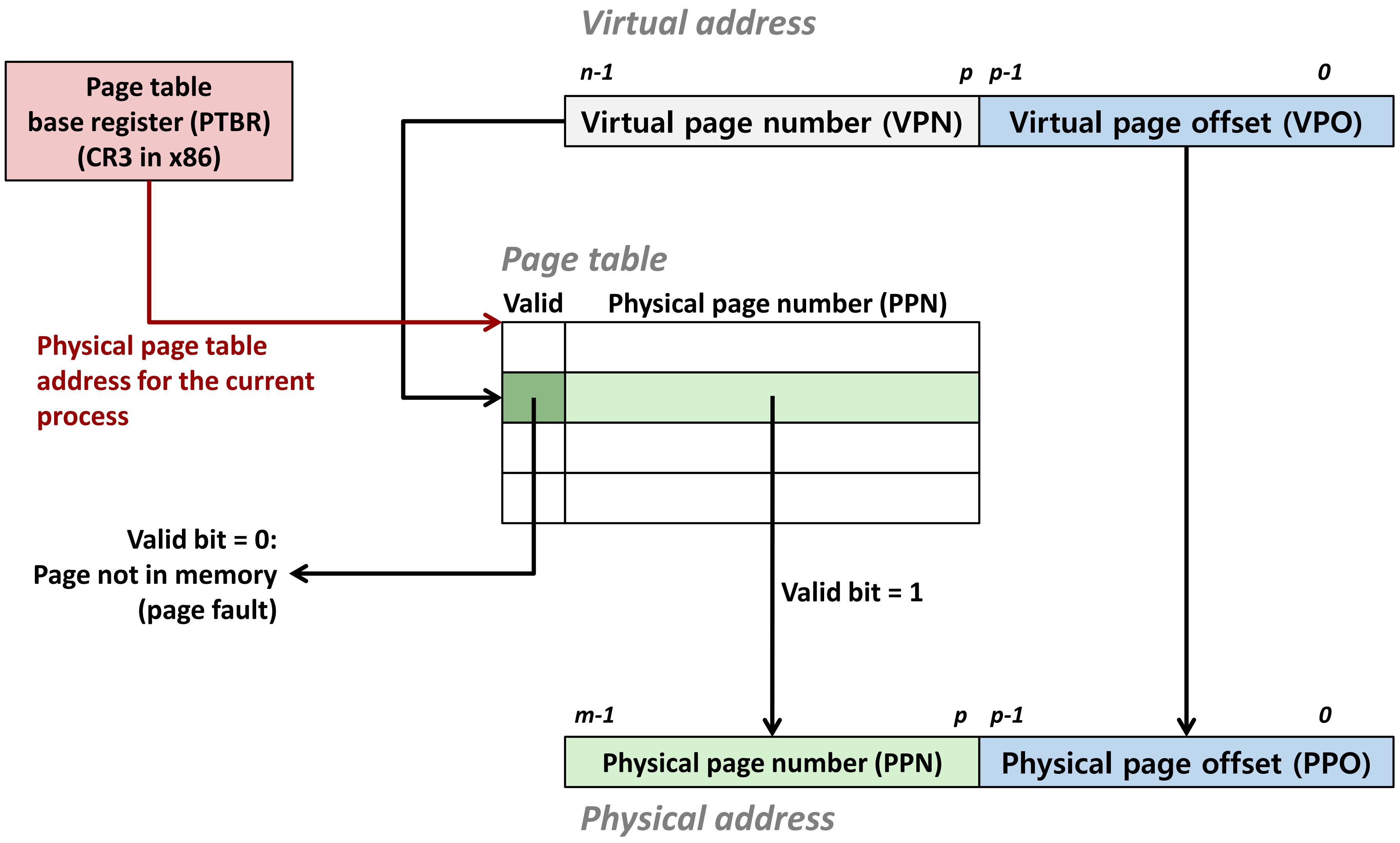
VM Address Translation

- Virtual Address Space
  - $V = \{0, 1, \ldots, N-1\}$
- Physical Address Space
  - $P = \{0, 1, \ldots, M-1\}$
- Address Translation
  - $\boldsymbol{MAP: \ V \rightarrow \ P \ \ U \ \ \{\varnothing\}}$
  - For virtual address ***a***:
    - $\boldsymbol{MAP(a) \ = \ a'}$ if data at virtual address ***a*** is at physical address ***a′*** in ***P***
    - $\boldsymbol{MAP(a) \ = \varnothing}$ if data at virtual address ***a*** is not in physical memory
      - Either invalid or stored on disk

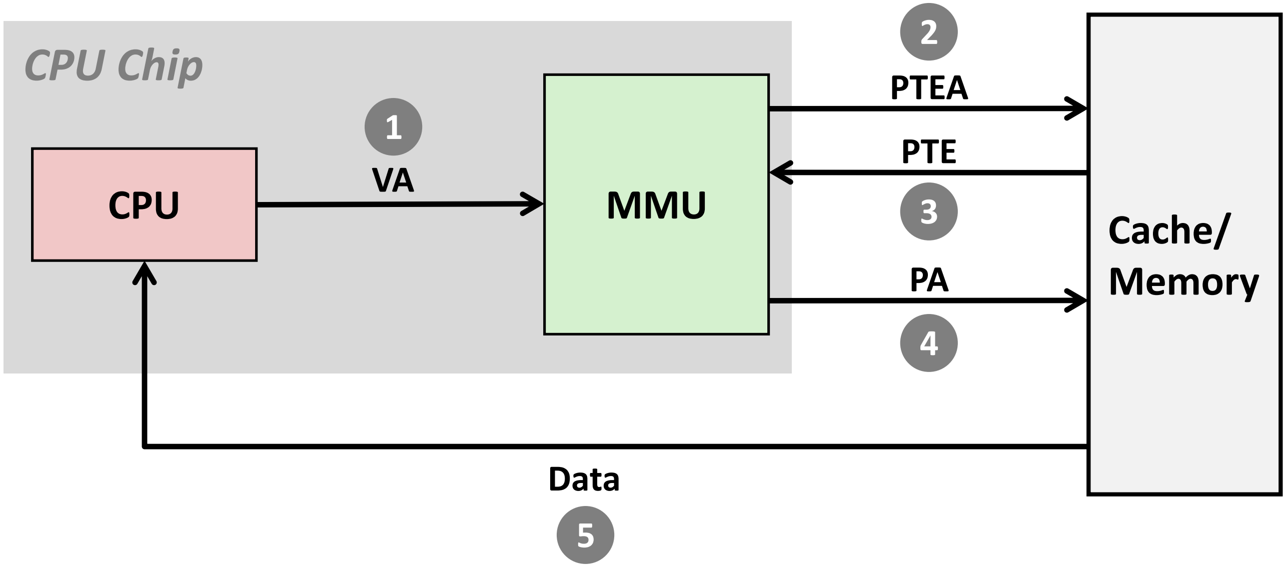
VM Address Translation

- 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)
  - **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

Address Translation With a Page Table



Address Translation: Page Hit



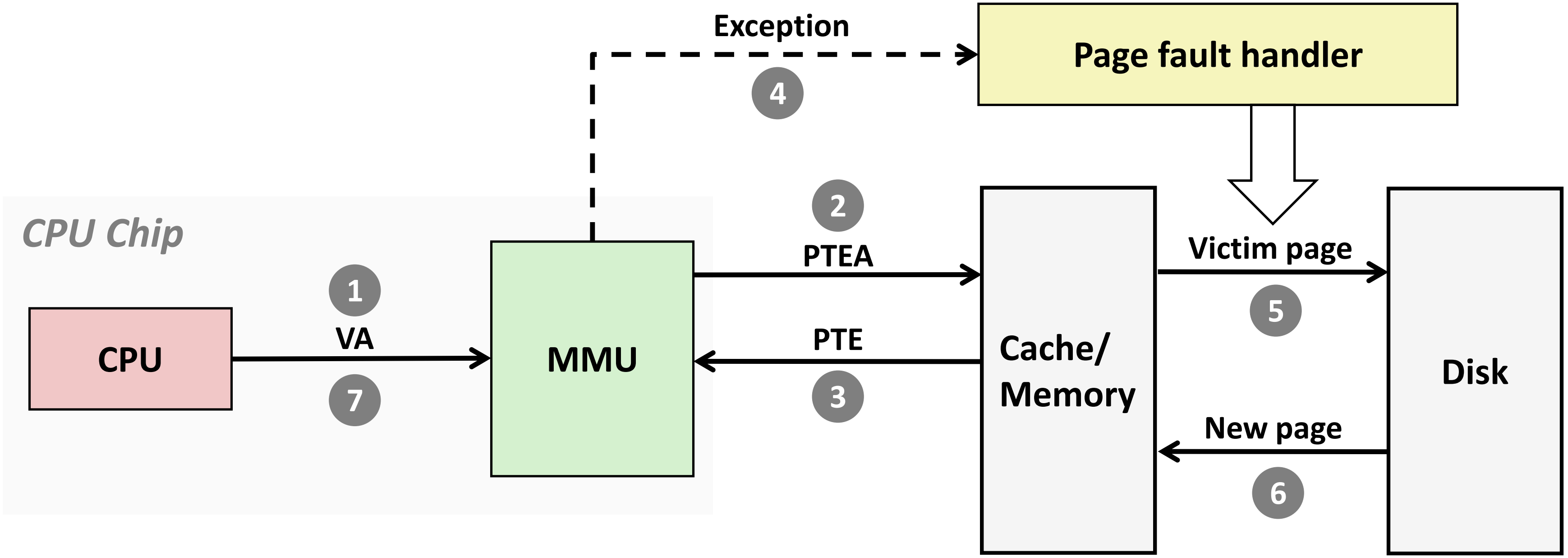
1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

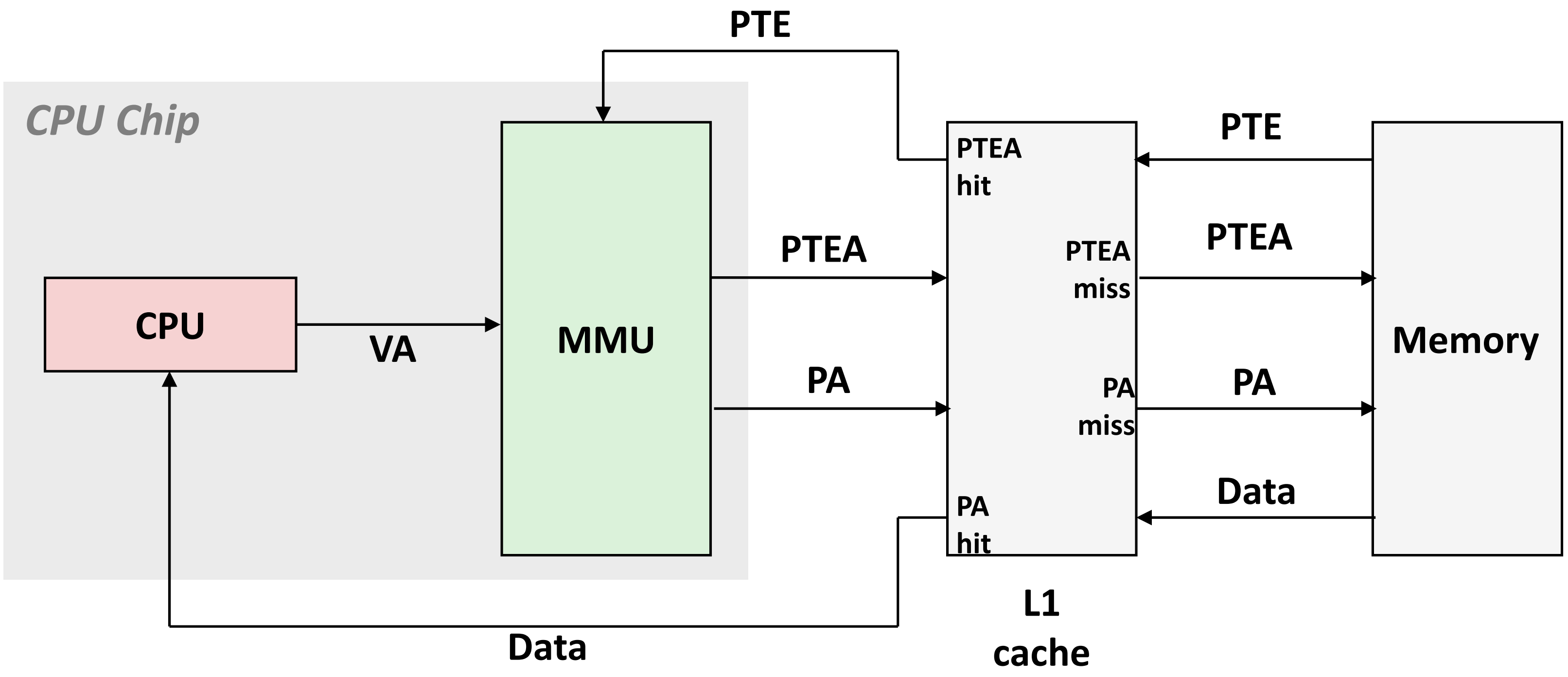
5) Cache/memory sends data word to processor

Address Translation: Page Fault



1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim to page out (if dirty, writes pages to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction

Integrating VM and Cache



VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Speeding up Translation with a TLB

- Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay

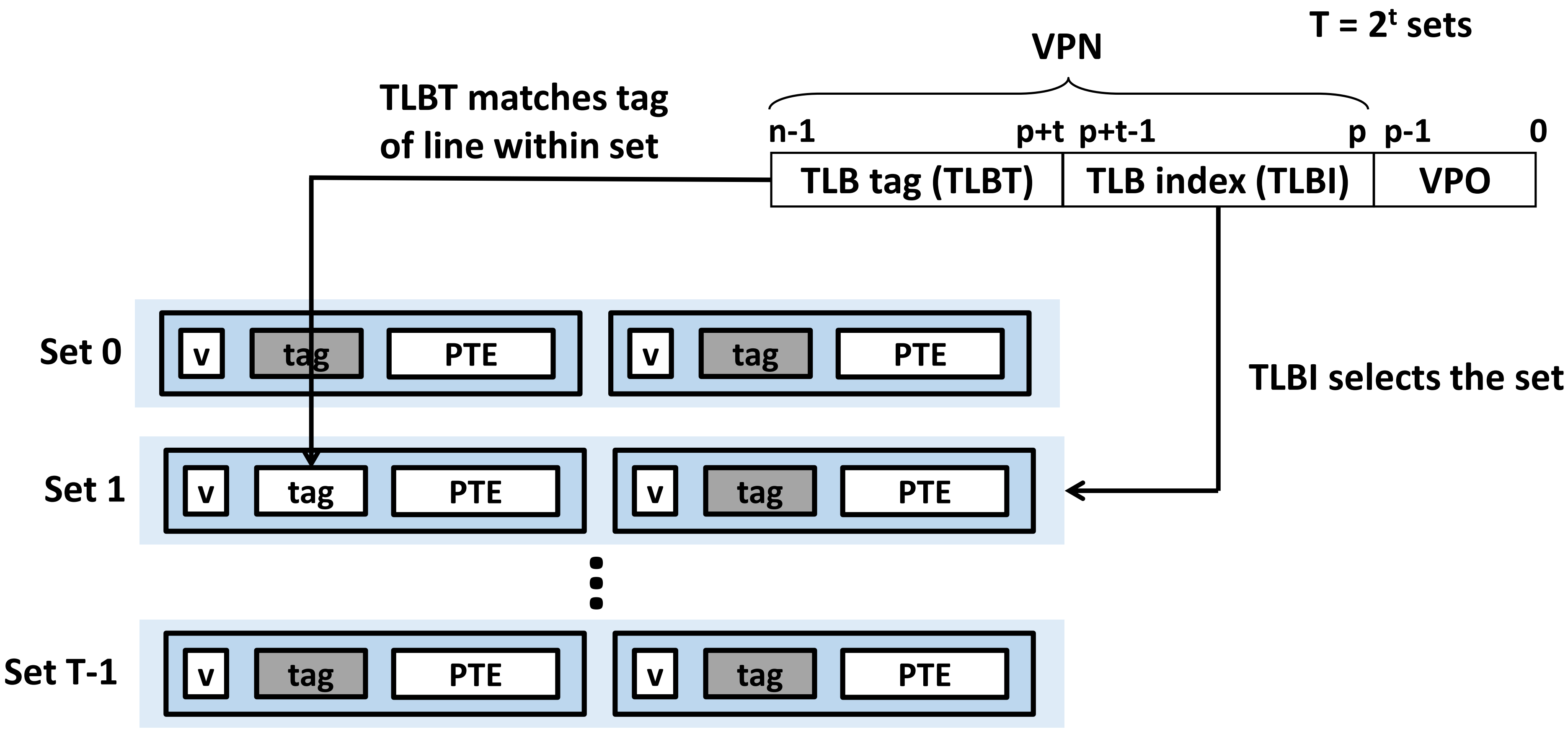
- Solution: **Translation Lookaside Buffer** (TLB)
  - Small set-associative hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

Summary of Address Translation 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

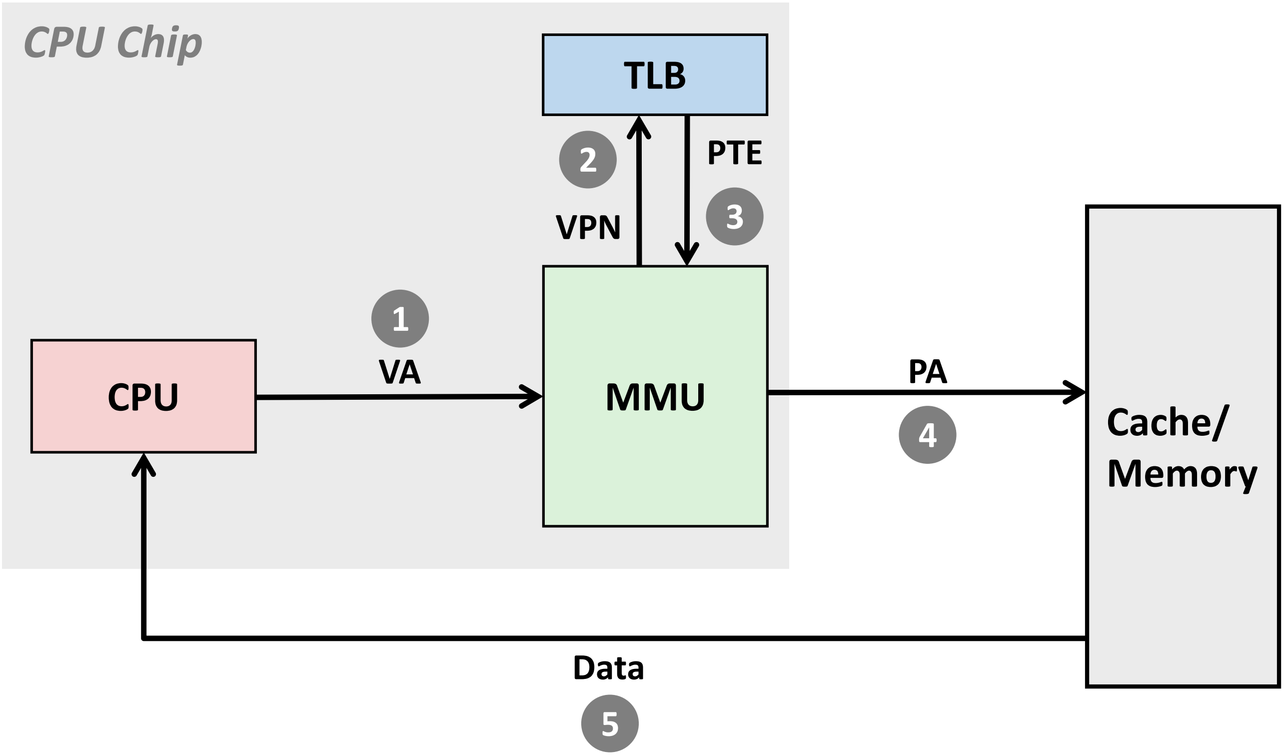
Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:



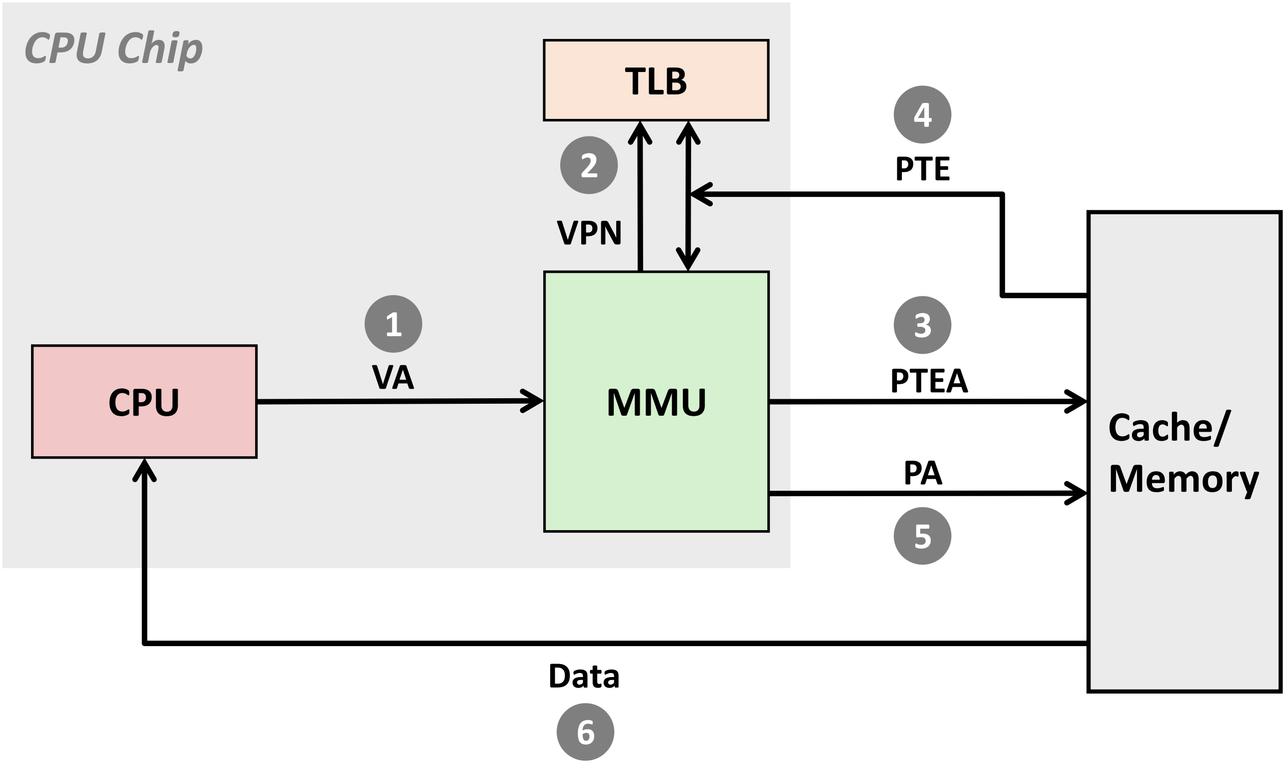
TLB Hit

- A TLB hit eliminates a cache/memory access



TLB Miss

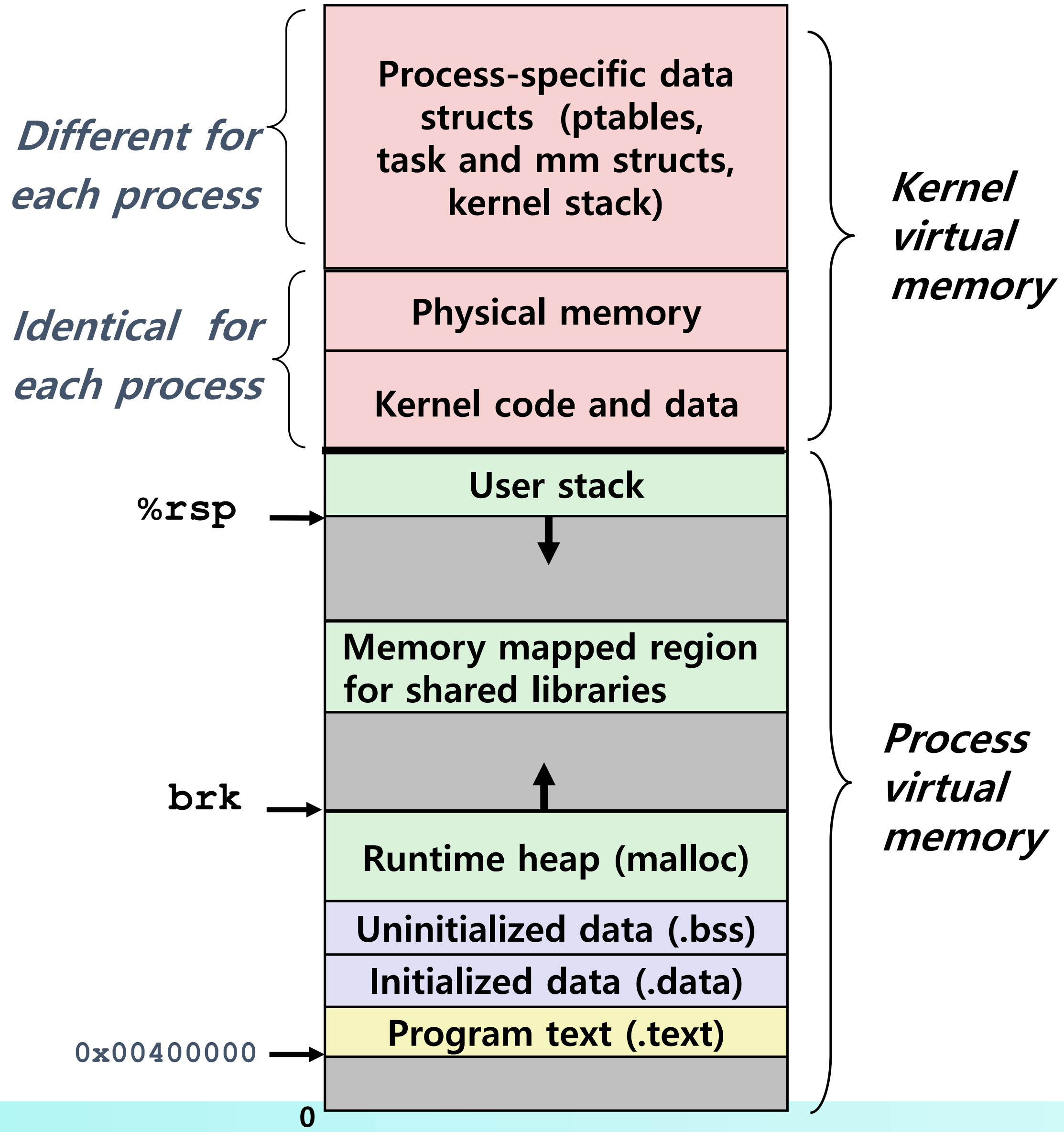
- A TLB miss incurs an additional cache/memory access (the PTE)
  - Fortunately, TLB misses are rare. Why?



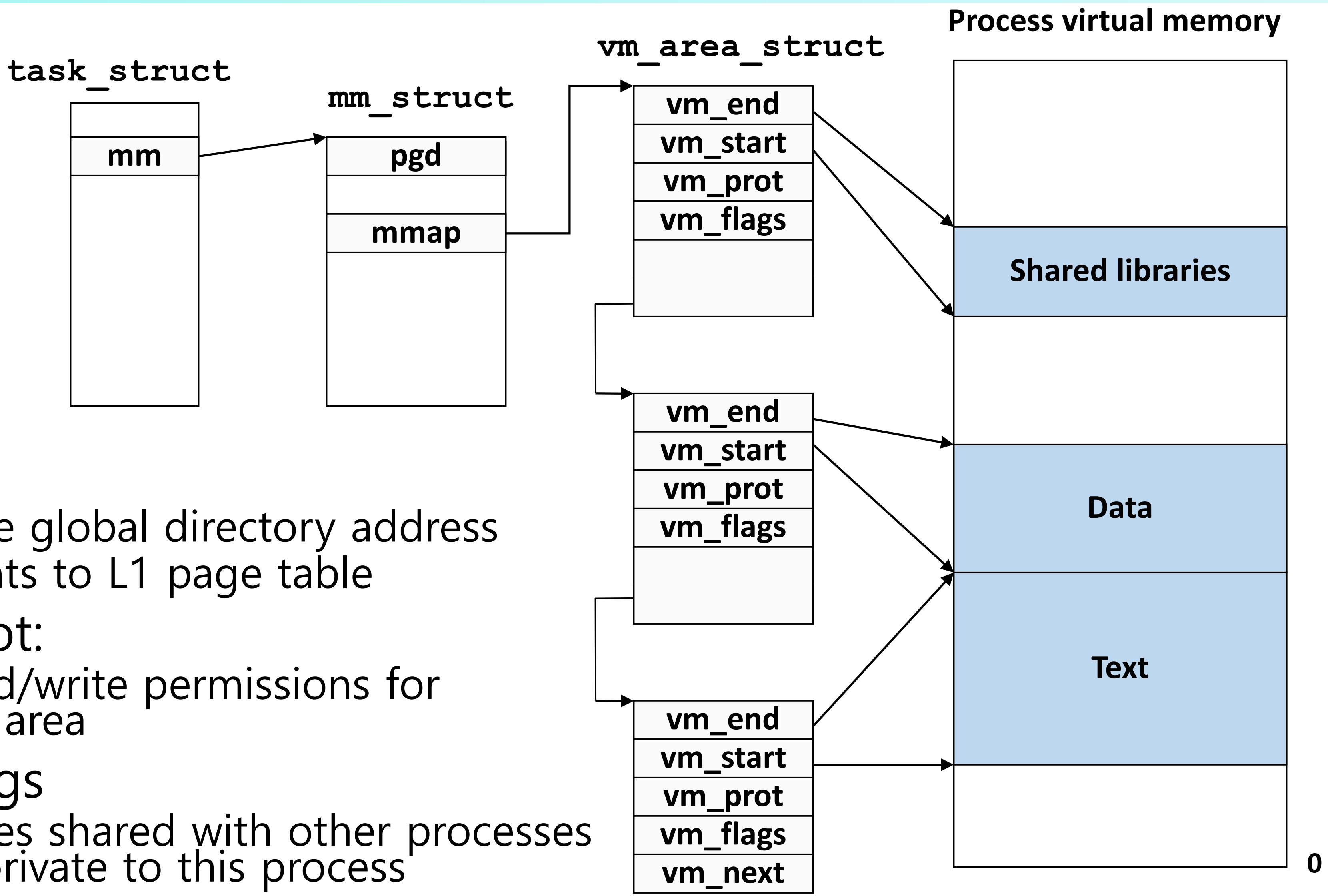
Contents

- Address Spaces
- VM as a Tool for Caching
- VM as a Tool for Memory Management and Protection
- Address Translation
- Memory Mapping

Virtual Address Space of Linux Process



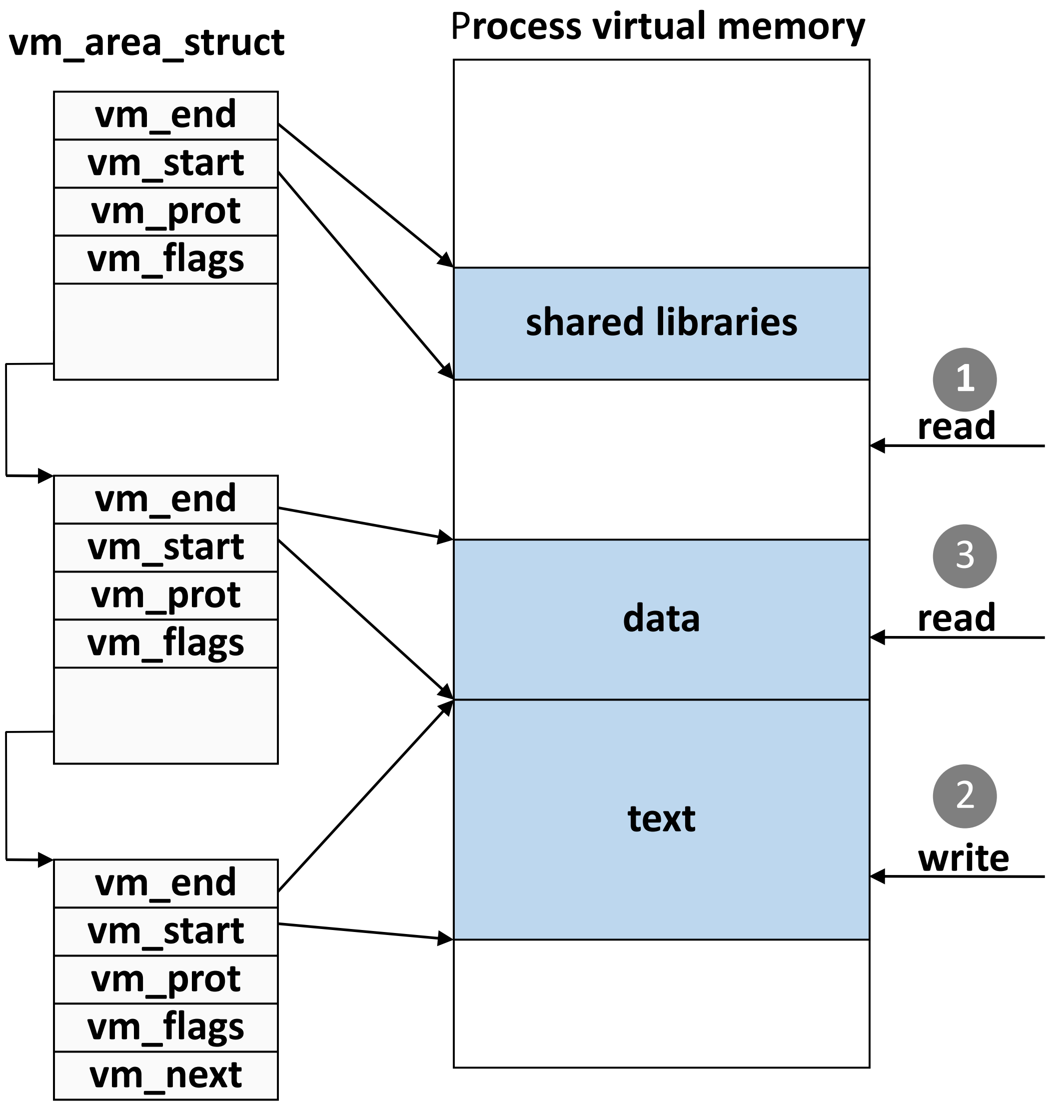
Linux Organizes VM as Collection of "Areas"



- 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

Each process has own `task_struct`, etc

Linux Page Fault Handling



Segmentation fault:
accessing a non-existing page

Normal page fault

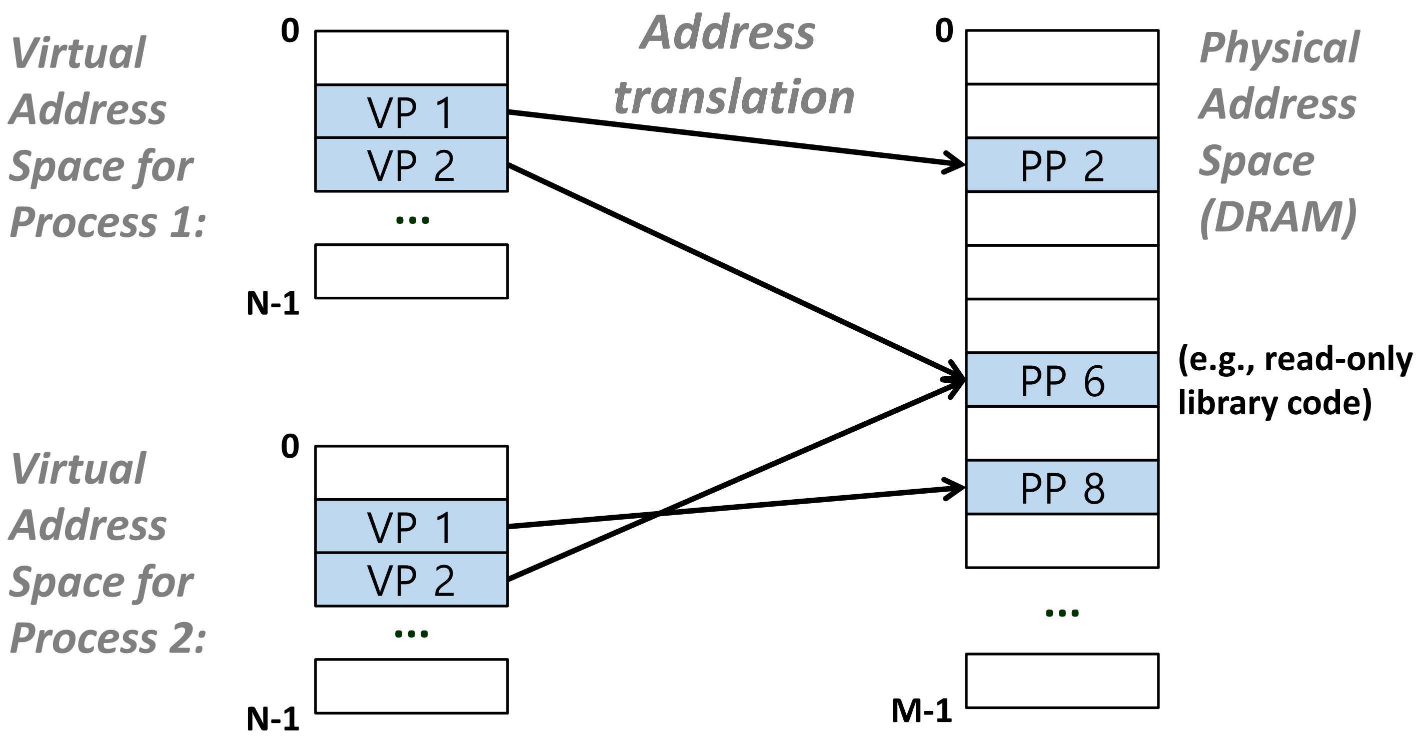
Protection exception:
e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

Memory Mapping

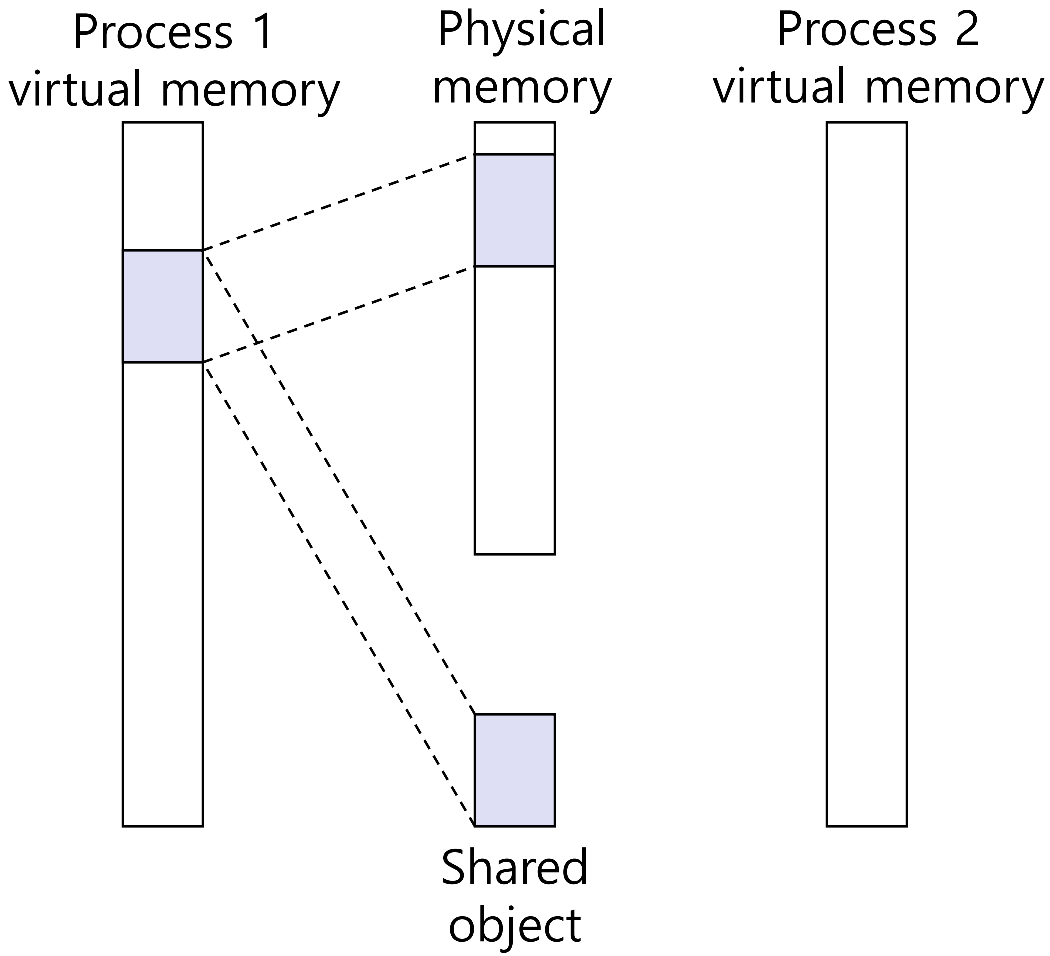
- VM areas initialized by associating them with disk objects.
    - Called memory mapping
- Area can be backed by (i.e., get its initial values from) :
    - Regular file on disk (e.g., an executable object file)
        - Initial page bytes come from a section of a file
    - Anonymous file (e.g., nothing)
        - First fault will allocate a physical page full of 0's (demand-zero page)
        - Once the page is written to (dirtied), it is like any other page
- Dirty pages are copied back and forth between memory and a special swap file.

Review: Memory Management & Protection

- Code and data can be isolated or shared among processes

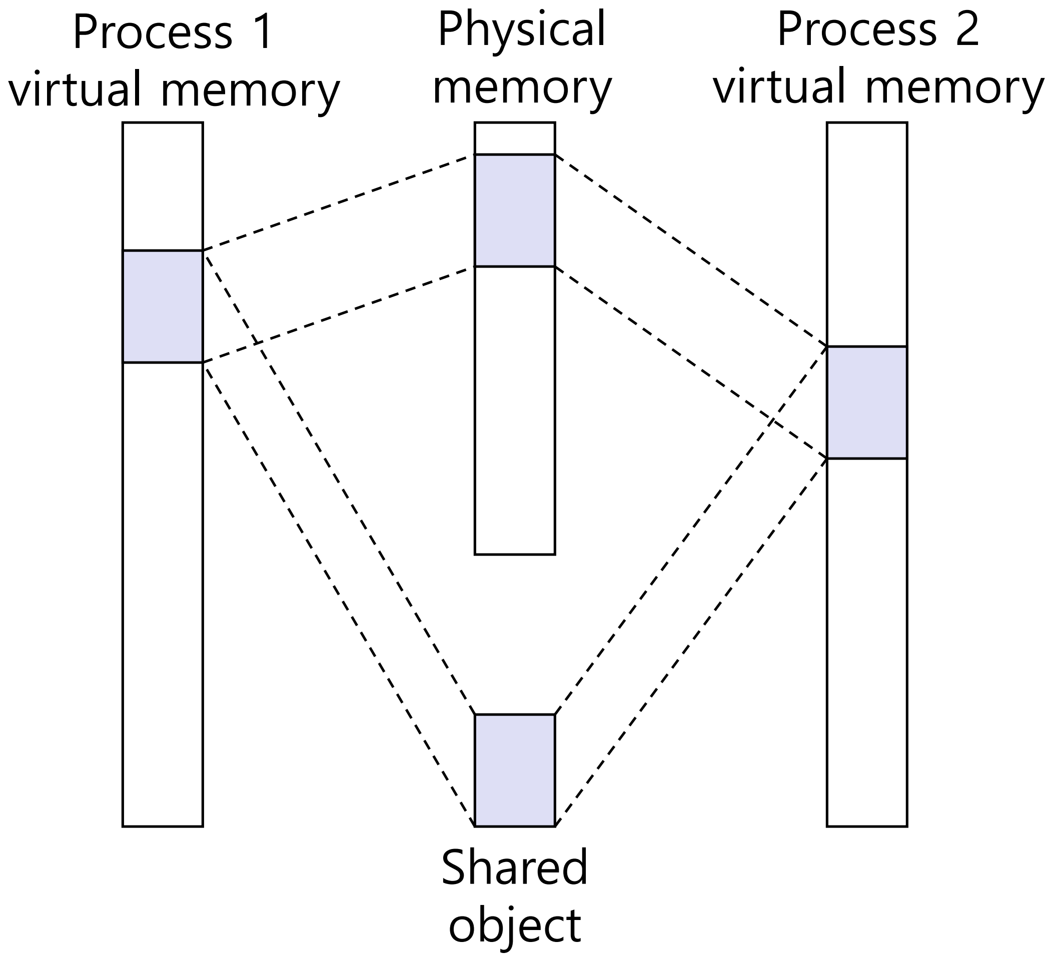


Shared Objects



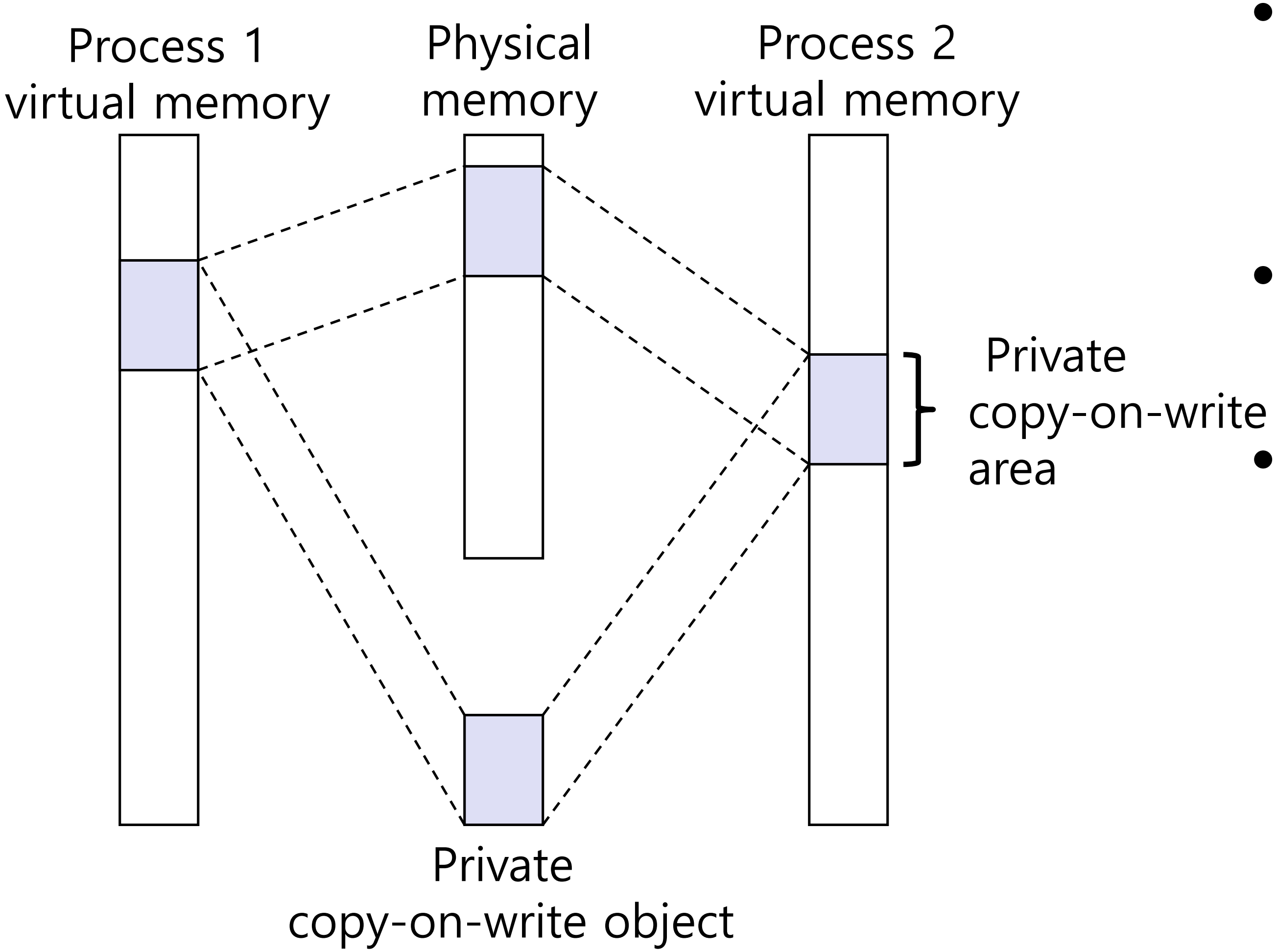
- Process 1 maps the shared object (on disk).

Shared Objects



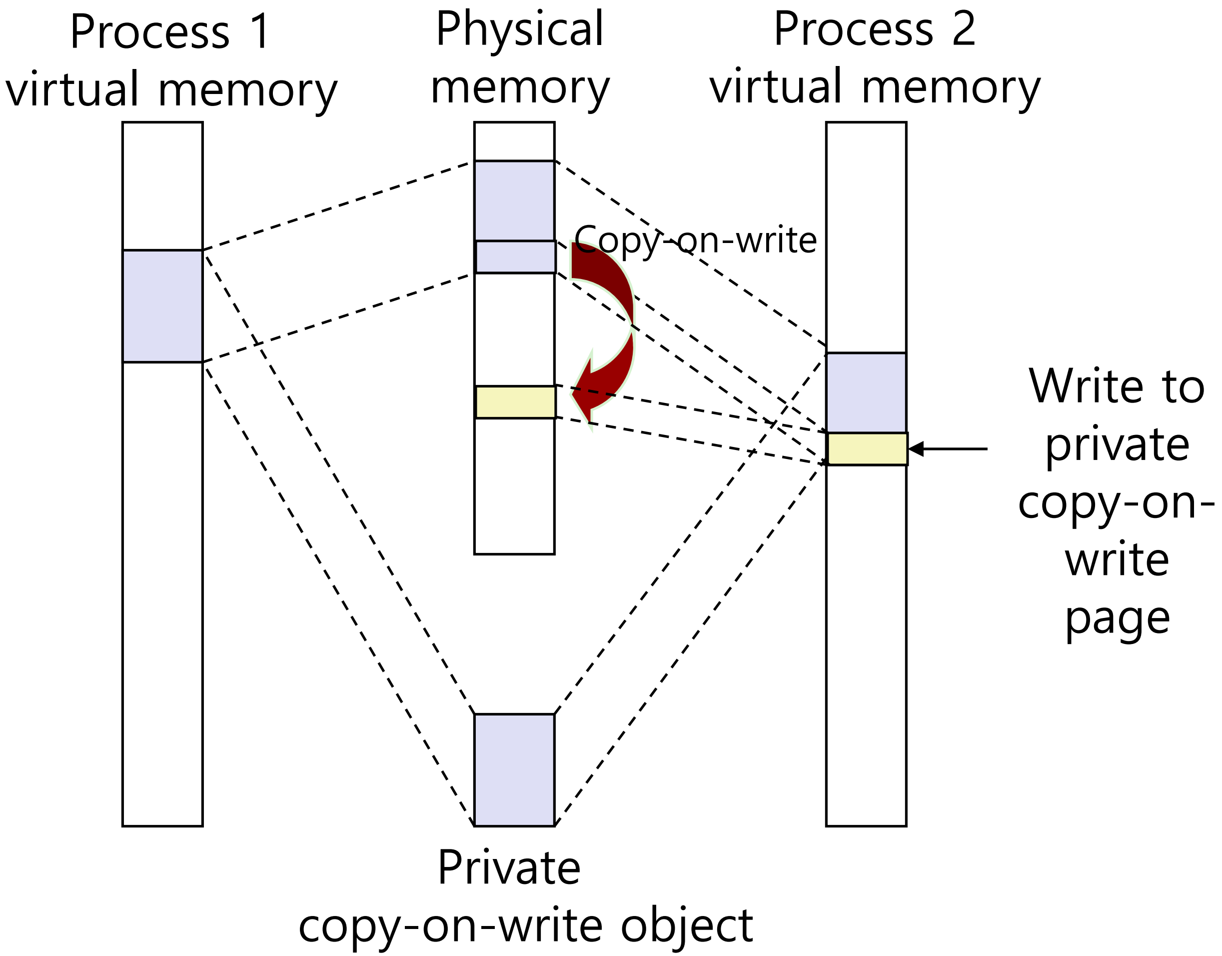
- Process 2 maps the same shared object.
- Notice how the virtual addresses can be different.
- But, difference must be multiple of page size.

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

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!

`mmap()`: User-Level Memory Mapping

- `mmap()`: Map files or devices into memory

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

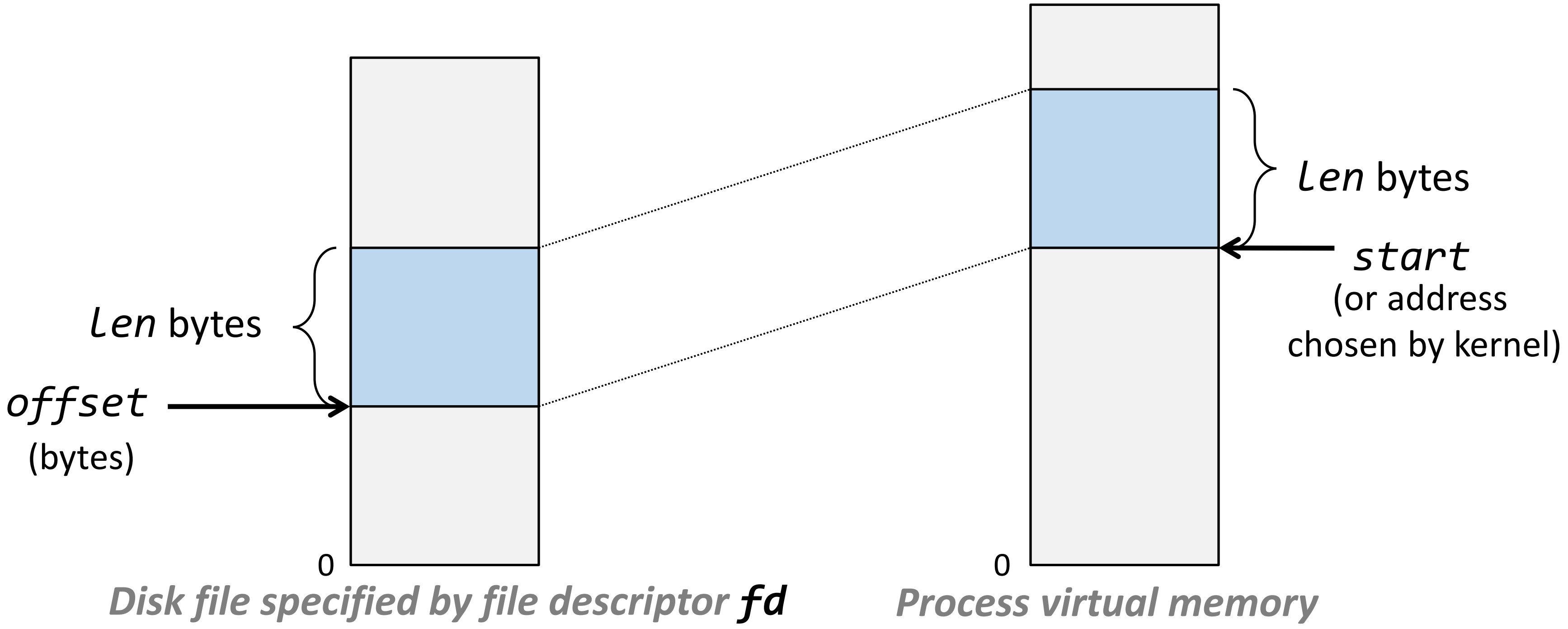
  - Creates a new mapping in the virtual address space of the calling process.
  - *addr*: Starting address for the mapped memory
  - *length*: Size of the mapped memory
  - *prot*: Desired memory protection such as execute, read, and write.
  - *flags*: It determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file.
  - *fd*: File that needs to be mapped
  - *offset*: Offset indicating from where inside the file the mapping should start

https://www.man7.org/linux/man-pages/man2/mmap.2.html

mmap(): User-Level Memory Mapping

- mmap(): Map files or devices into memory

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```



Uses of `mmap`

- Reading big files
  - Uses paging mechanism to bring files into memory
- Shared data structures
  - When call with `MAP_SHARED` flag
    - Multiple processes have access to same region of memory
    - Risky!
- File-based data structures
  - E.g., database
  - Give `prot` argument `PROT_READ | PROT_WRITE`
  - When unmap region, file will be updated via write-back
  - Can implement load from file / update / write back to file

munmap()

- munmap(): Unmap files or devices

```
#include <sys/mman.h>
int munmap(void *addr, size_t length);
```

  - Deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references
  - *addr:* address must be a multiple of the page size (but *length* need not be)

- Return Value
  - On Success: 0
  - On failure: -1 and *errno* is set

Memory Mapping using mmap()

```c
int main()
{
    struct stat st;
    char content[20];
    char *new_content = "New Content";
    void *map;
    int f = open("./zzz", O_RDWR);
    fstat(f, &st);

    // Map the entire file to memory
    map = mmap(NULL, st.st_size, PROT_READ | PROT_WRITE,
               MAP_SHARED, f, 0);

    // Read 10 bytes from the file via the mapped memory
    memcpy((void*)content, map, 10);
    printf("read: %s\n", content);

    // Write to the file via the mapped memory
    memcpy(map+5, new_content, strlen(new_content));

    // Clean up
    munmap(map, st.st_size);
    close(f);
    return 0;
}
```

memmap.c

```
yunmin@peace:~/ch11$ bash -c 'yes 1111111111 | head -c 6647 > ./zzz'
yunmin@peace:~/ch11$ head zzz
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
yunmin@peace:~/ch11$ gcc memmap.c -o memmap
yunmin@peace:~/ch11$ ./memmap
read: 1111111111@
yunmin@peace:~/ch11$ head zzz
11111New Content11111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
```