Cache & Virtual Memory

System Programming

Woong Sul

Questions You May Have

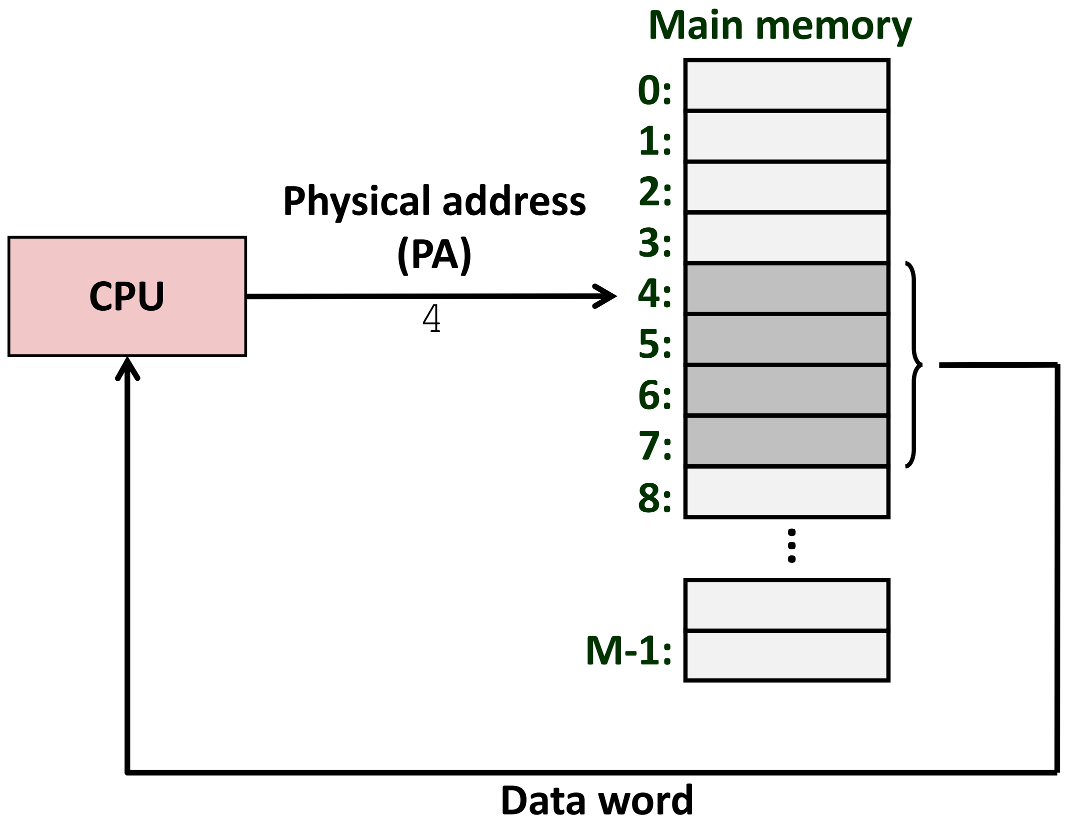
- How each process can have
  - Own memory address space
  - Even more memory than its physical memory size
- How each process can be protected from
  - Invalid memory access from the process itself
  - Any memory access from other processes
- How we can make each process
  - Efficiently load its process image from an executable file
  - Efficiently share libraries among different processes

Today

- Address spaces
- General cache concepts
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Memory mapping

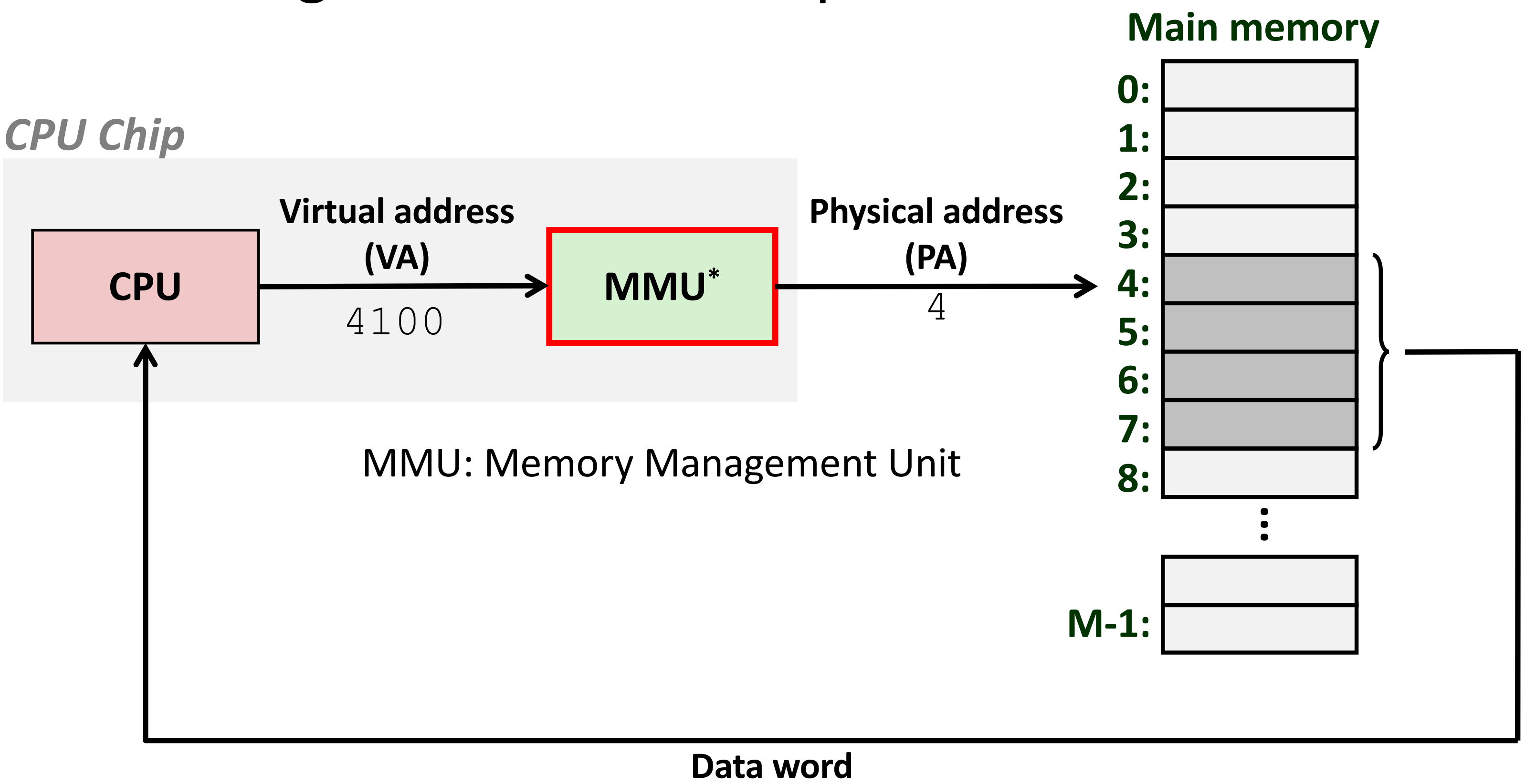
Systems Using Physical Addressing

- A process uses the same address space as its physical address
  - Two processes can use the same address
- Used in ***simple*** systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames



Systems Using Virtual Addressing

- A process uses virtual address space with the help of **MMU**s
  - Processes can use the same address in their own address space
- 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}

What Virtual Memory Enables

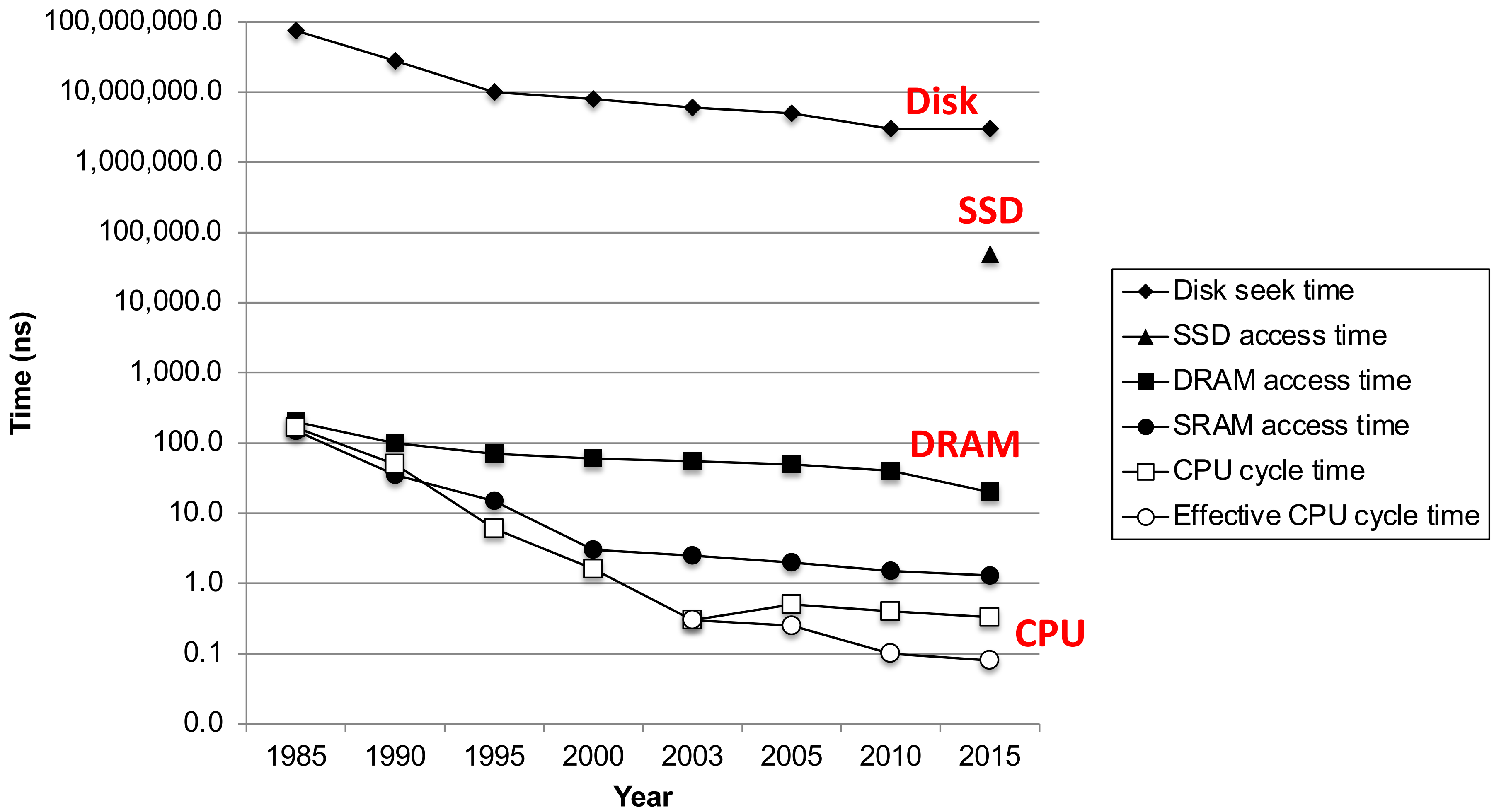
- Using main memory efficiently
  - Use DRAM as a ***cache*** for portions of a virtual address space
- Simplifying memory management
  - Each process gets the same uniform linear address space
- Isolating address spaces
  - One process can't interfere with another's memory
  - User program cannot access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

Today

- Address spaces
- General cache concepts
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Memory mapping

Illusion of Fast and Large Main Memory

- The gap **widens** between DRAM, disk, and CPU speeds
- Processes require memory faster than DRAM

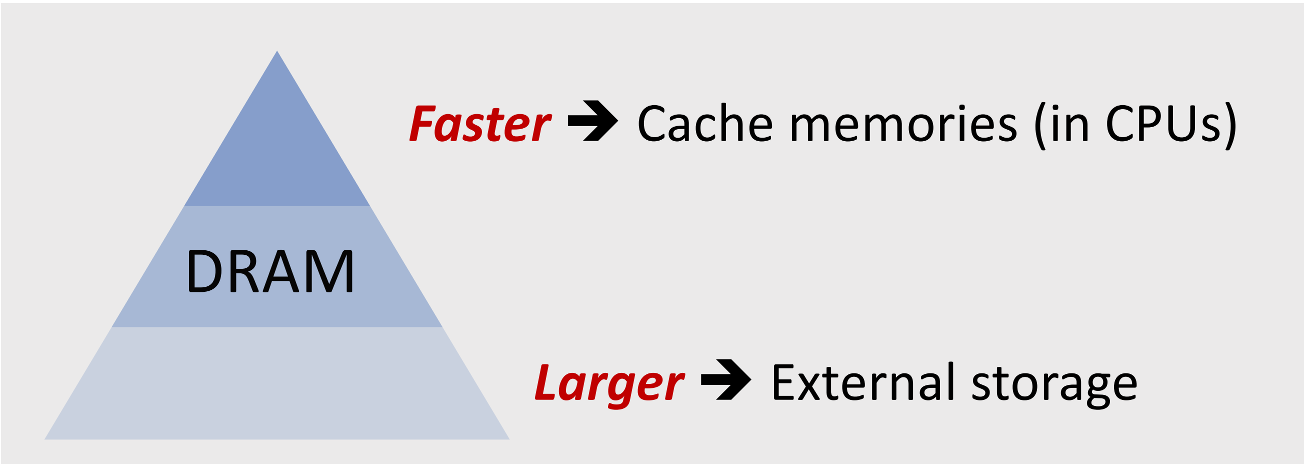


Illusion of Fast and Large Main Memory

- Processes require memory larger than the physical DRAM size
  - Main memory requires large space for different processes

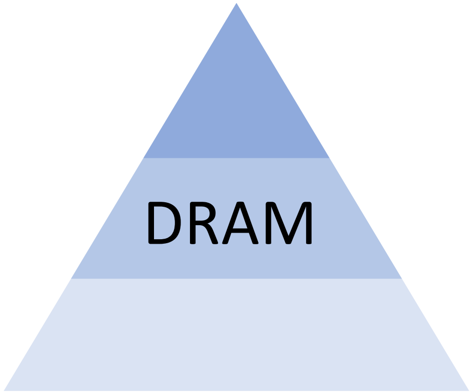
- Process memory may consist of different storage media
  - For main memory faster and larger than DRAM

Process Memory



Memory Hierarchies

- Storage media typically trades off speed with capacity
  - The larger the storage capacity, the slower the speed



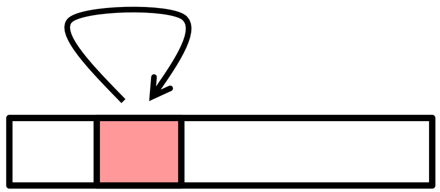
Type	Latency(cycles)	Capacity	IO size
Registers	0	Hundreds B	8B
Cache memory	4-10	Hundreds KB	64B
Main memory	100	Hundreds GB	4KB
SSD	10,000	Tens TB	4-8KB
HDD	10,000,000	Tens TB	0.5-16KB

- Processes can have non-uniform memory access, but this will not be a problem in general
  - Processes typically have ***locality*** in their memory access

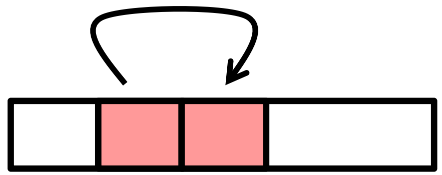
Locality

- Tendency to access data and instructions with addresses ***equal to*** or ***near*** those they have used recently

- ***Temporal locality***
  - Recently referenced items are likely to be referenced again in near future



- ***Spatial locality***
  - Items with nearby addresses tend to be referenced close together in time



Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

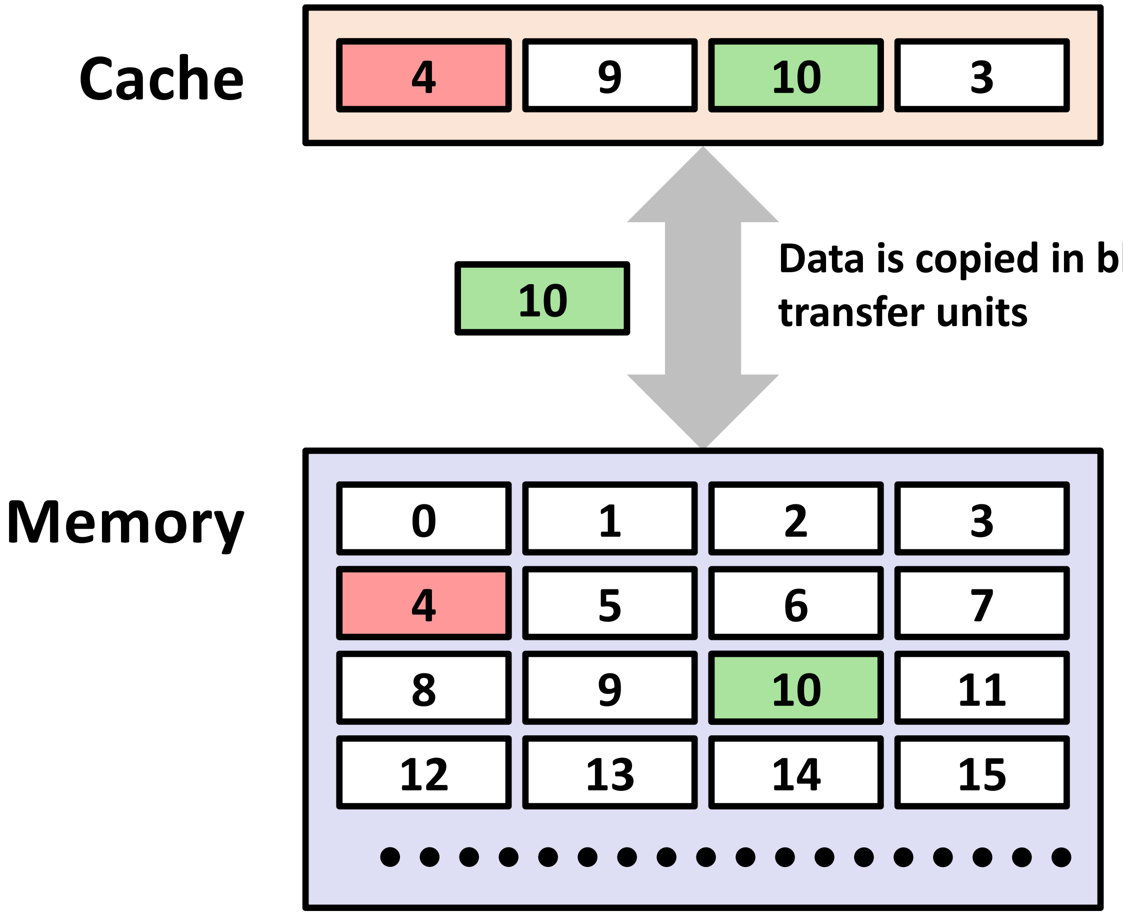
- Data references
  - Reference array elements in succession (stride-1 reference pattern) **Spatial locality**
  - Reference variable sum each iteration **Temporal locality**
- Instruction references
  - Reference instructions in sequence **Spatial locality**
  - Cycle through loop repeatedly **Temporal locality**

General *Cache* Concepts

- Fundamental idea of a memory hierarchy
  - Placing a subset of data in slower storage to faster storage
  - For each k, storage at level k serves as a cache for storage at level k+1
- What makes memory hierarchies work effectively?
  - Move every accessed data to faster storage
  - Evict less frequently accessed data from faster storage
  - Then frequently accessed data will be left in faster storage
- Big idea
  - All data can be stored in a large pool of storage
  - A subset of data can be accessed at speed of the fastest storage

General *Cache* Concepts Example

- Data is copied to faster memory at every access ➔ temporal
- Tranfering more data than requested ➔ spatial

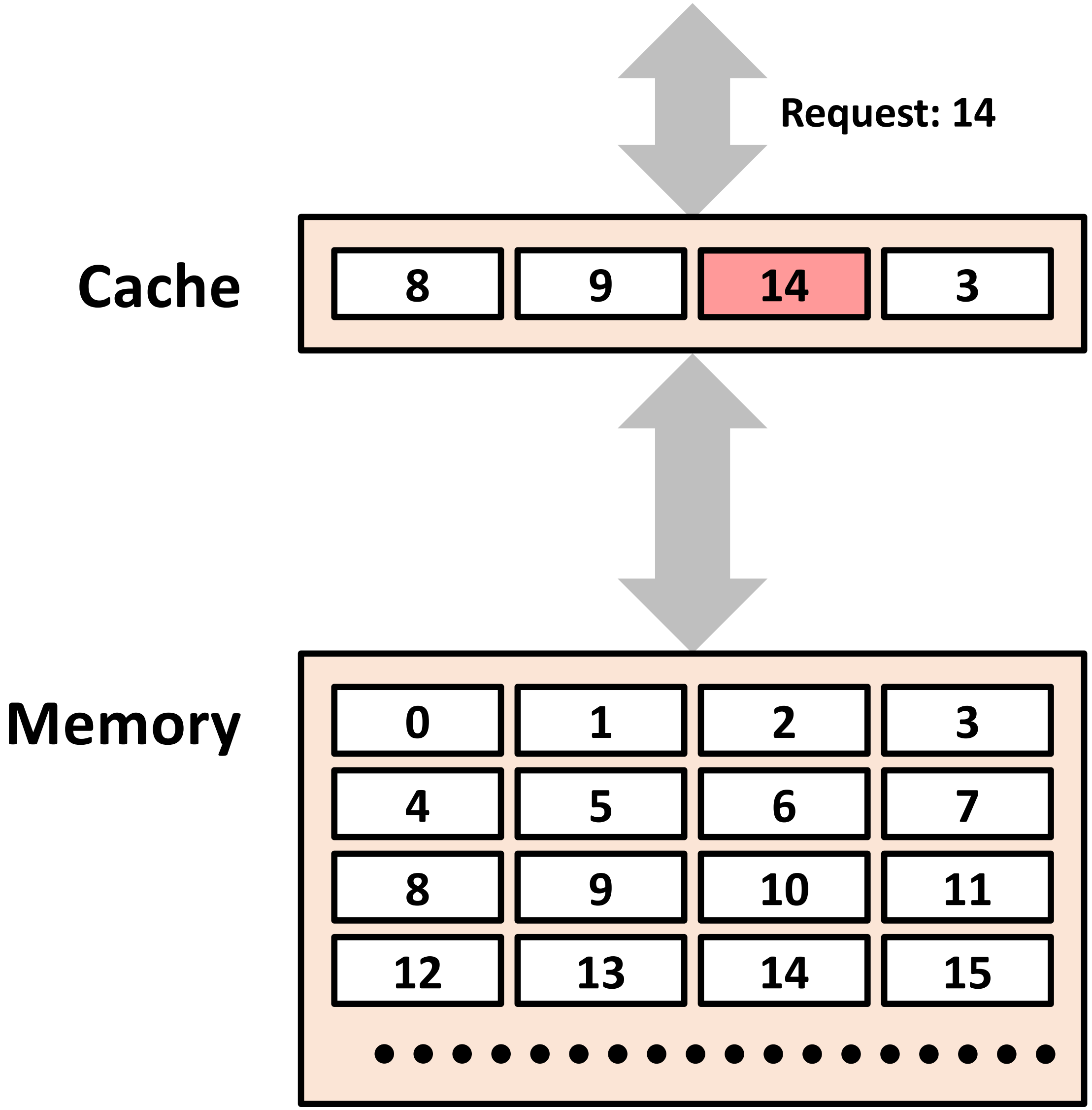


Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into "blocks"

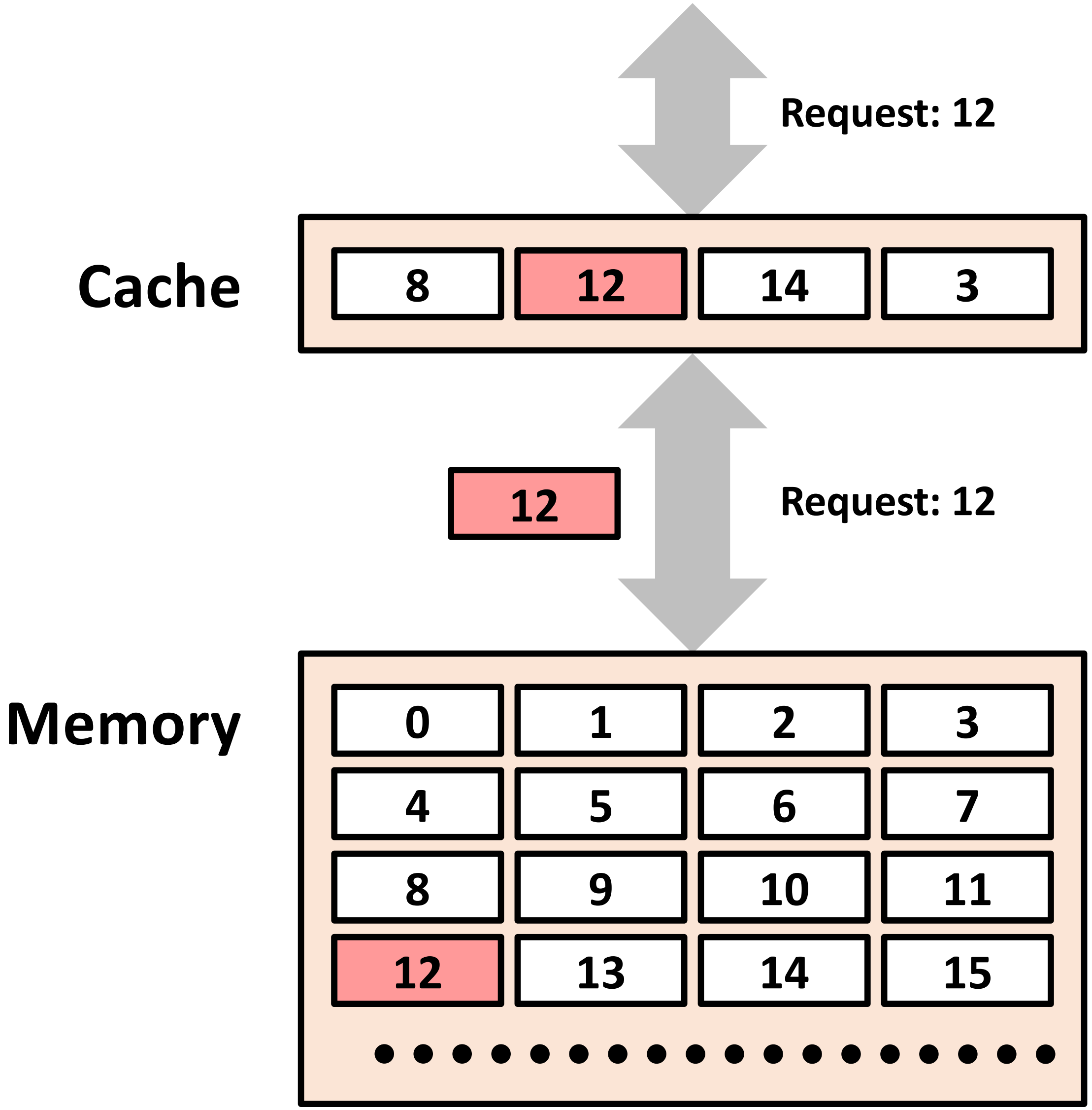
Cache Hits



Data in block b is needed

Block b is in cache:
Hit!

Cache Misses



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

Block b is stored in cache

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

Types of Cache Misses (a.k.a 3C)

- Cold miss (or compulsory miss)
  - Cold misses occur because the cache is empty
- Conflict miss
  - Most caches limit blocks at level k+1 to a small subset of the block positions at level k

    E.g., Block i at level k+1 must be placed in block (i mod 4) at level k
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block

    E.g., Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time
- Capacity miss
  - The set of active cache blocks (working set) is larger than the cache

Cache Misses Greatly Affect Performance

- Calling `test()` after warming up caches

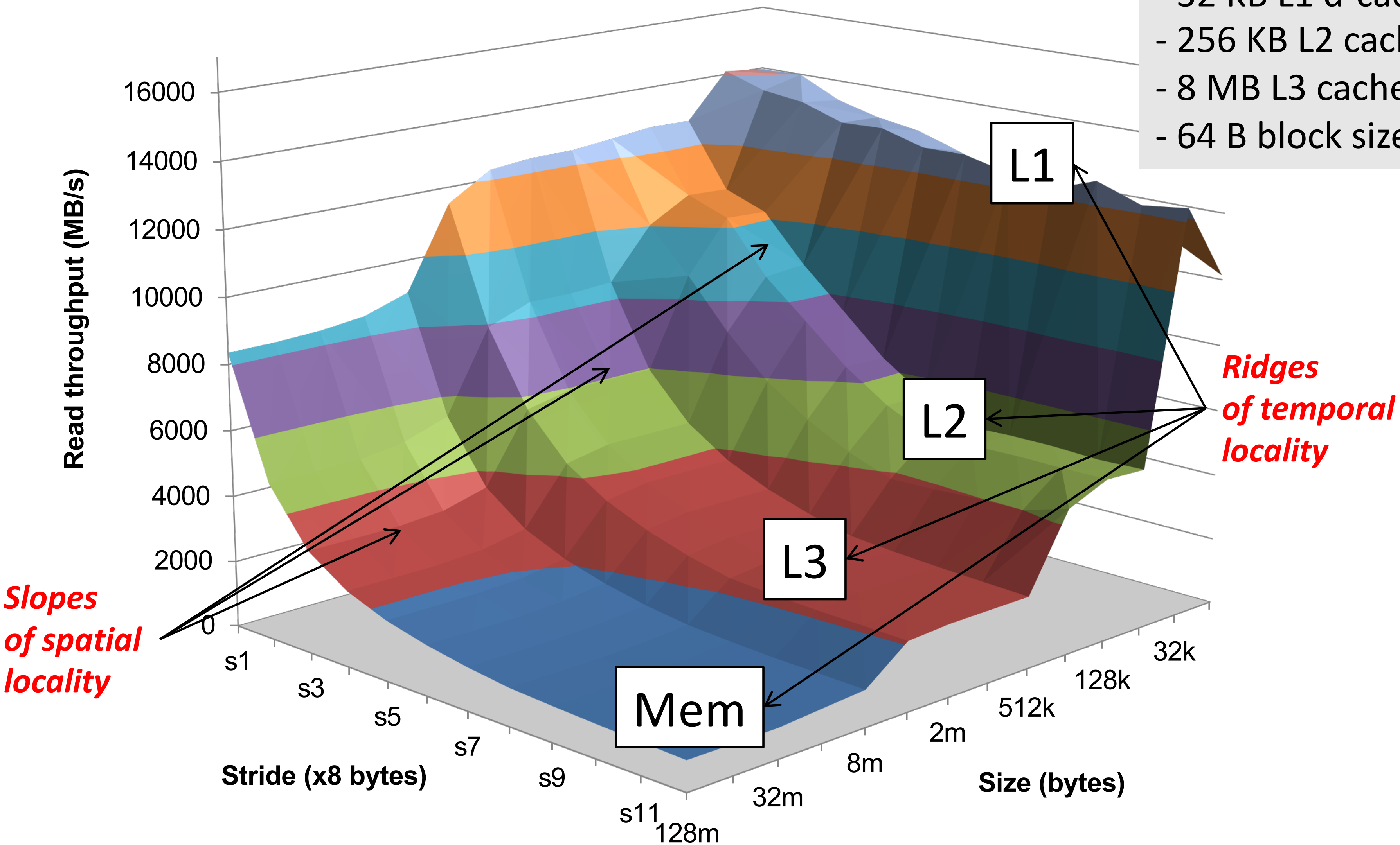
```
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *        array “data” with stride of "stride", using
 *        using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

mountain/mountain.c

Cache Misses Greatly Affect Performance

- Core i7 Haswell Example

Multi-level caches
- 32 KB L1 d-cache
- 256 KB L2 cache
- 8 MB L3 cache
- 64 B block size



System Design Related to Cache Misses

- Cache organization of fixed size cache

  Total size = Block size x # of blocks

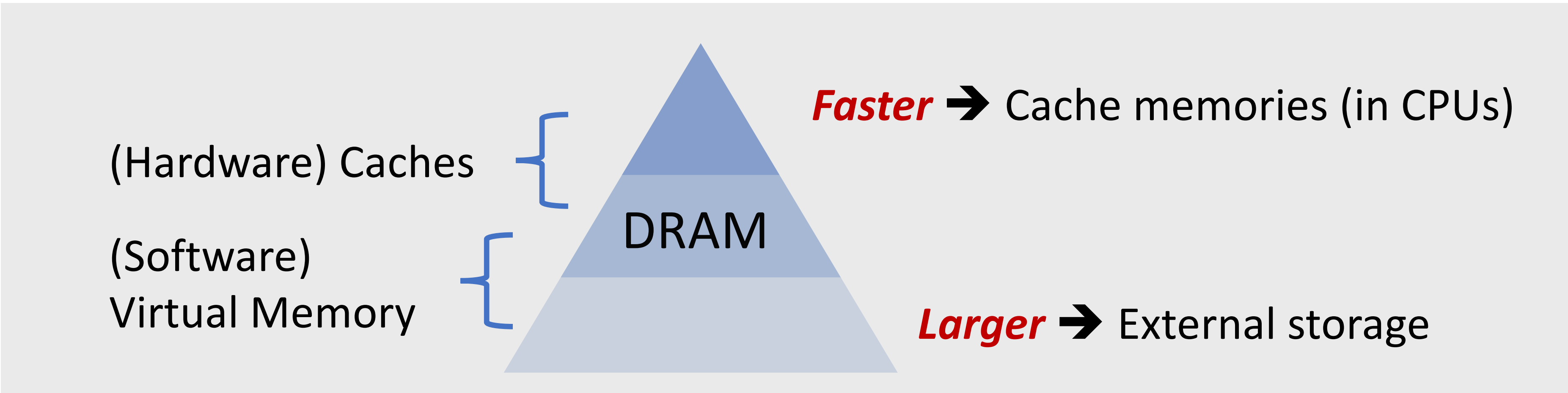
- Increasing block size
  - Exploits better spartial locality
  - Makes cache faster by comparing fewer blocks for replacement
- But decreasing # of blocks
  - Increases **miss rate** due to eviction of more words
  - Increases **miss penalty** due to increased IO amount

System Design Related to Cache Writes

- Using cache memory duplicates data
  - Data in caches and the same copy of the data in memory
- What if we change the data in cache?
  - ***Write-through***

    Updates data in both cache and memory
  - ***Write-back***

    Updates data only in cache, and later data in memory if **dirty bit** is on

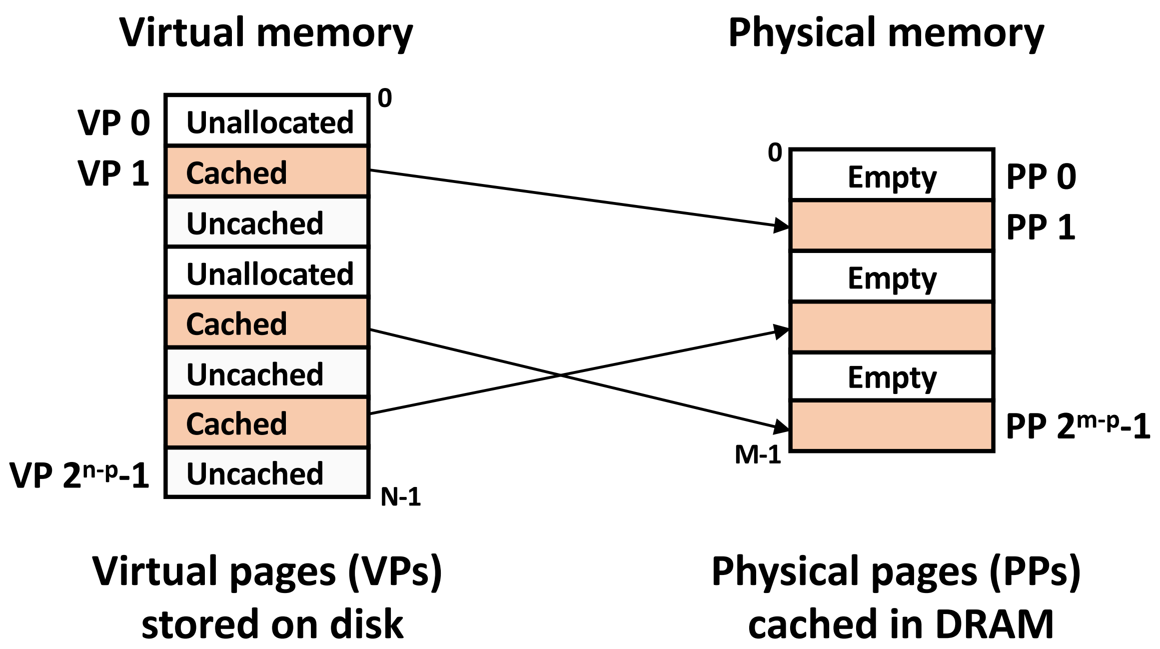


Today

- Address spaces
- General cache concepts
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- 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*** (or ***DRAM cache***)
  - These cache blocks are called ***pages***, whose size is $P = 2^p$ bytes



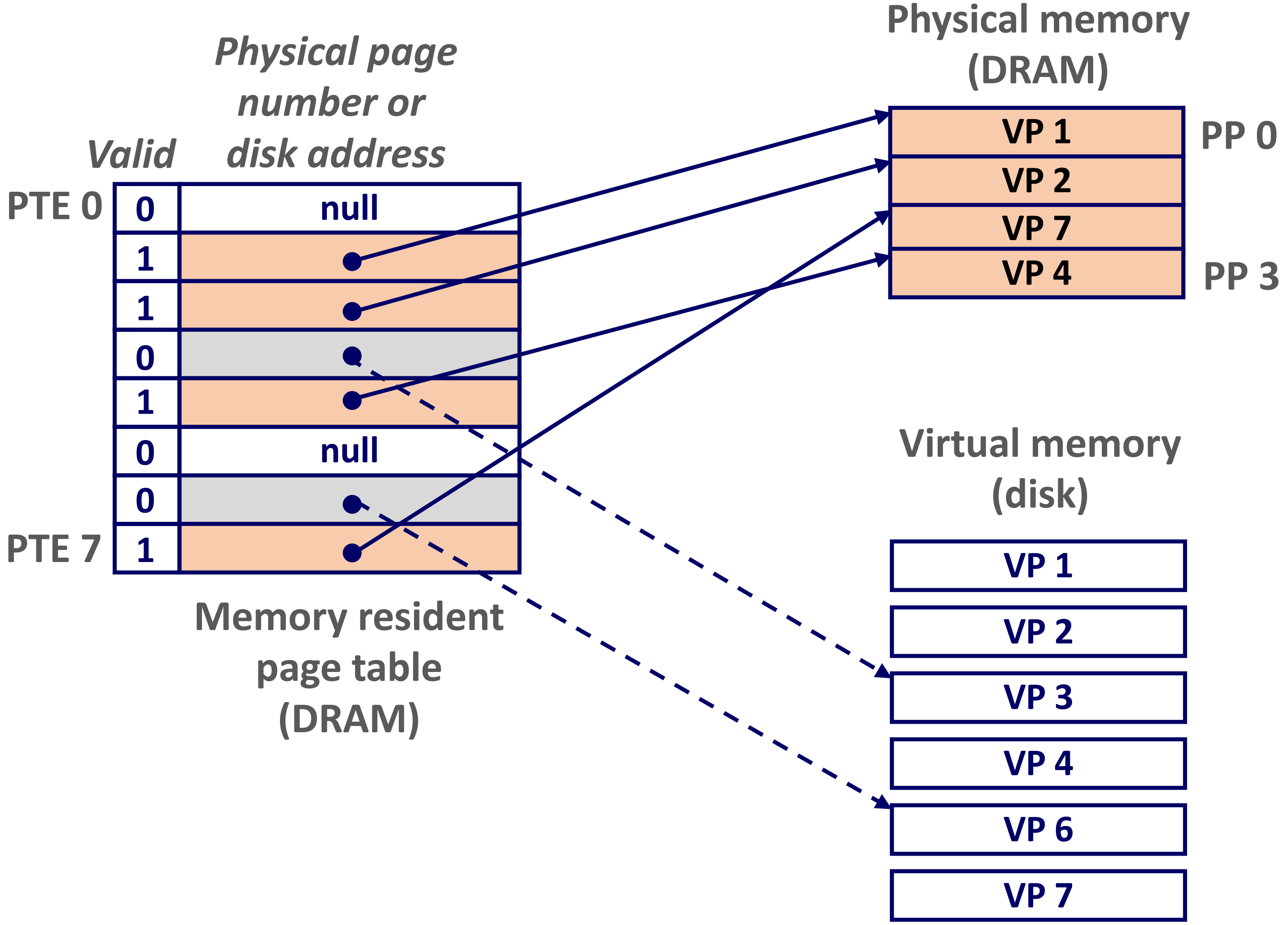
DRAM Cache Organization

- DRAM cache organization driven by the huge **miss penalty**
  - DRAM is about **10x** slower than SRAM (cache memory)
  - Disk is about **10,000x** slower than DRAM (main memory)

- 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

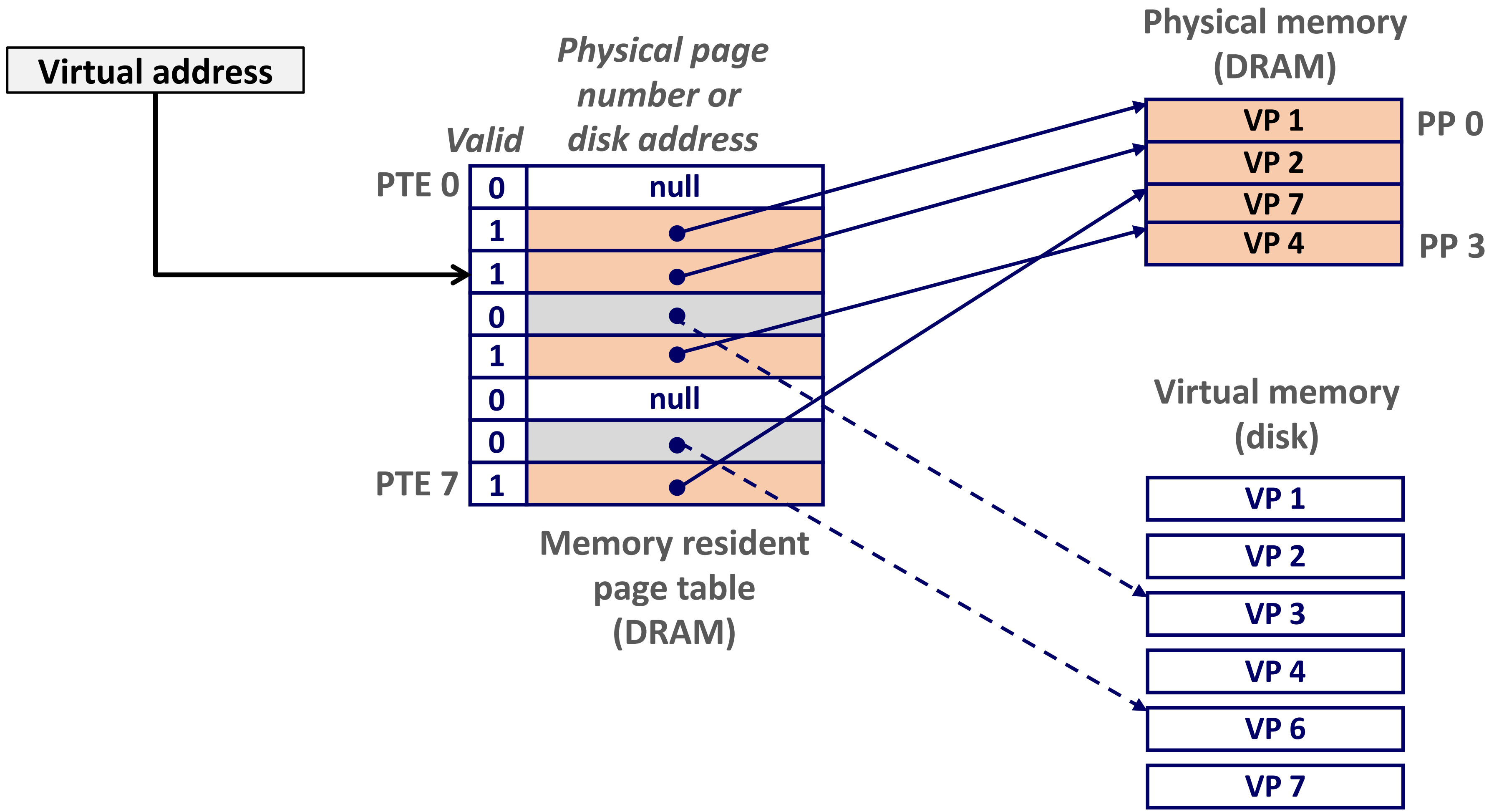
Core Data Structure in Virtual Memory

- 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



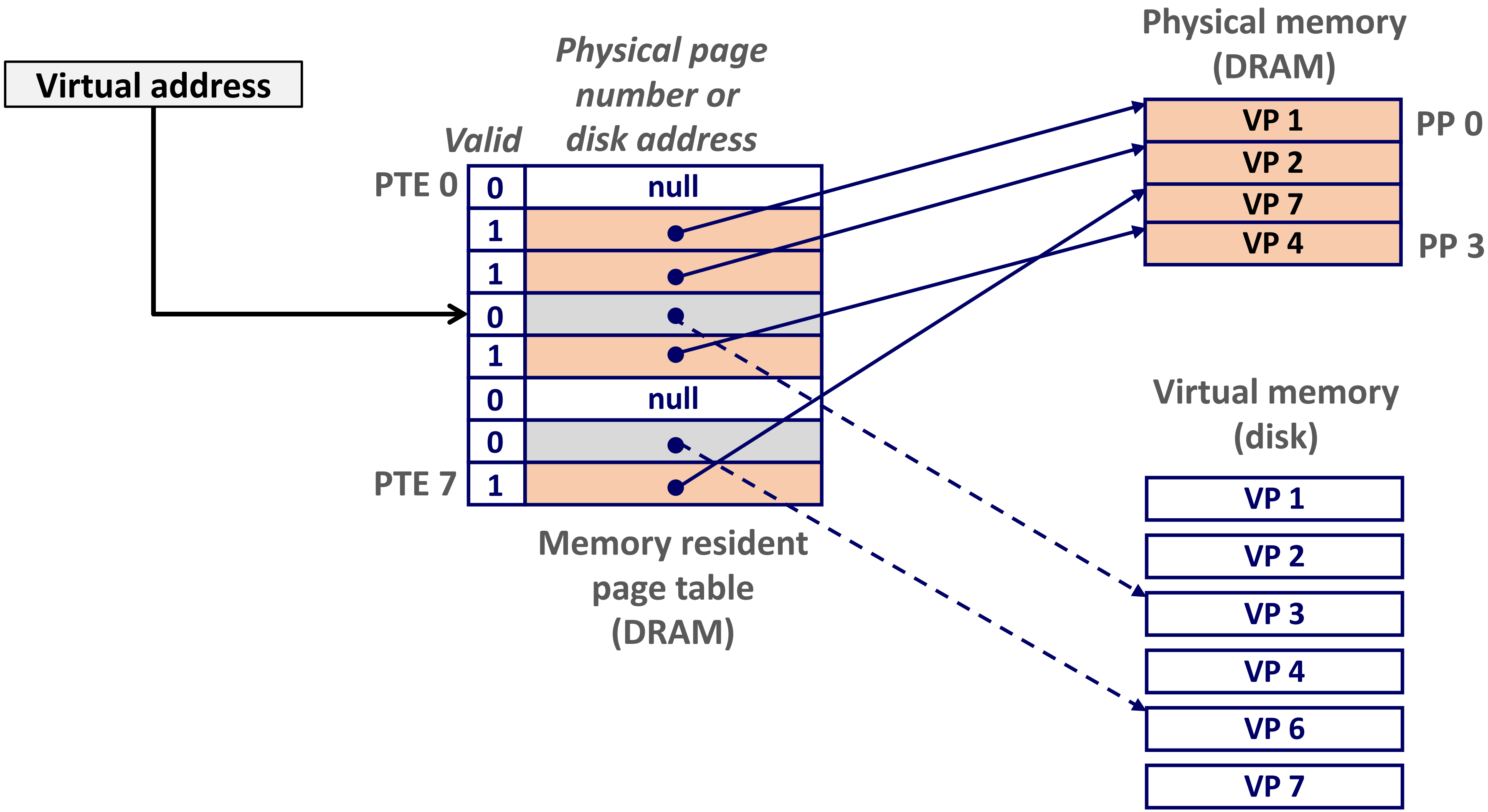
Page Hit

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



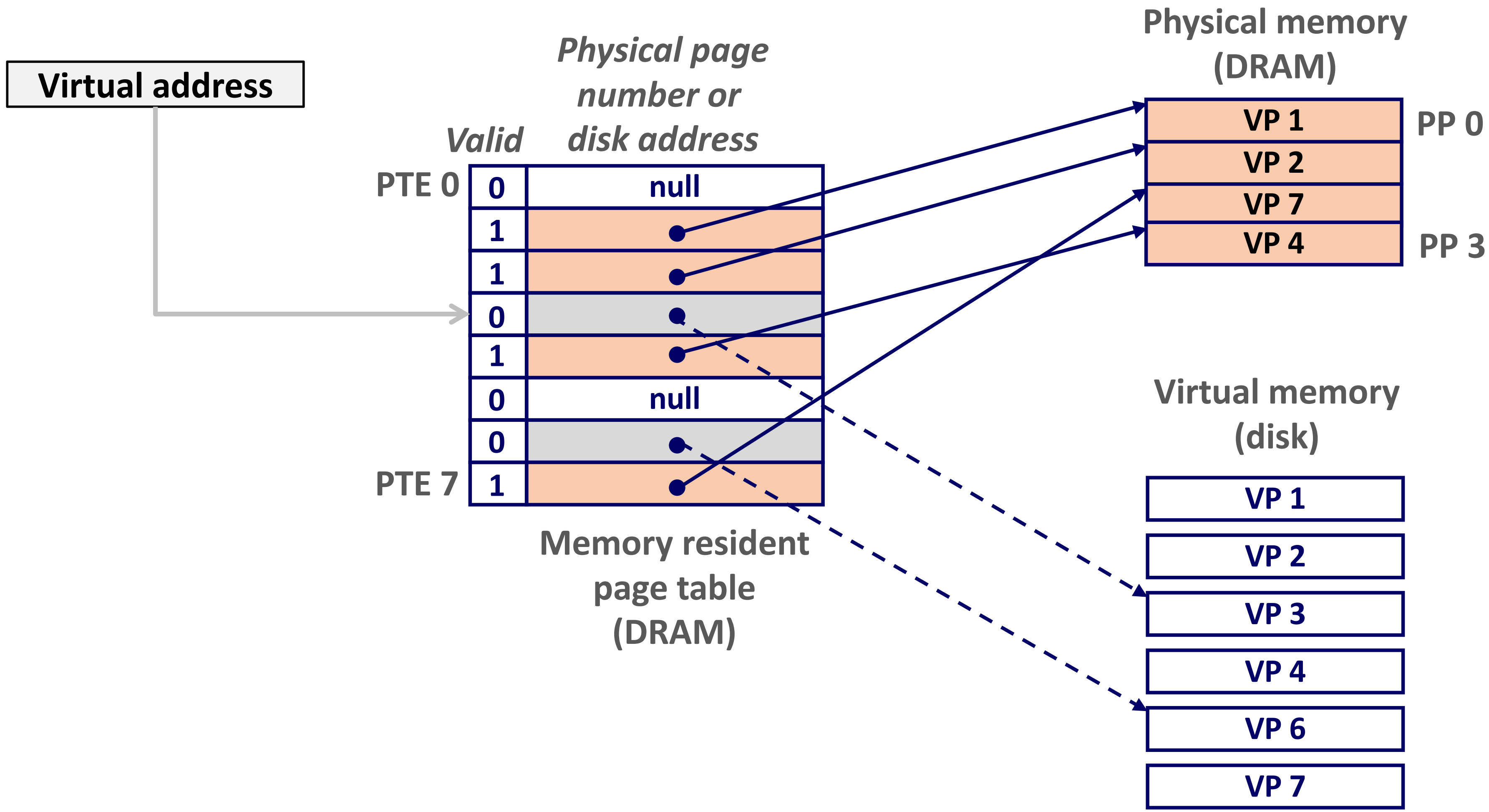
Page Fault

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



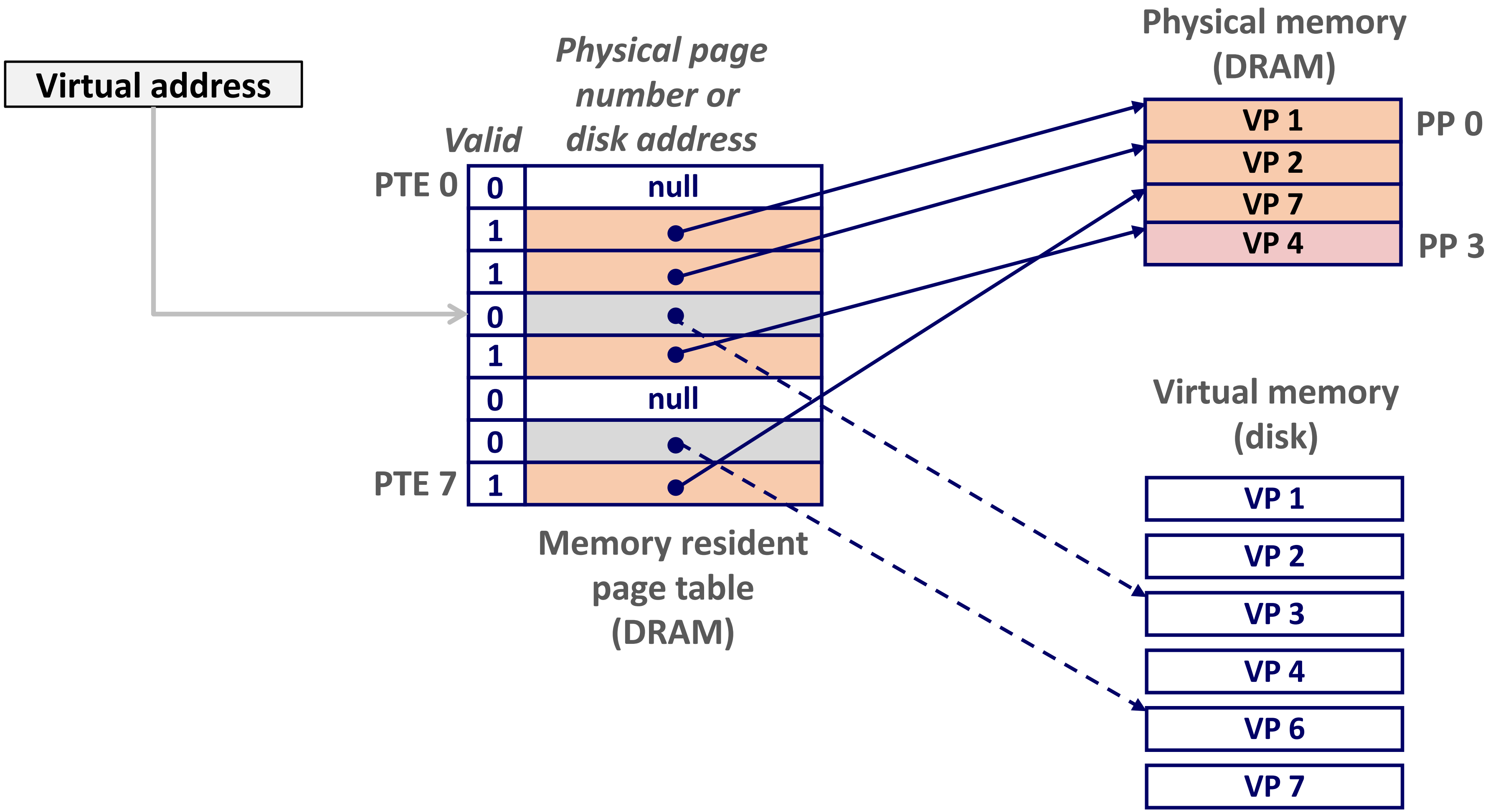
Handling Page Fault

- Page miss causes page fault (an exception)
  - But physical memory has no space for VP 3



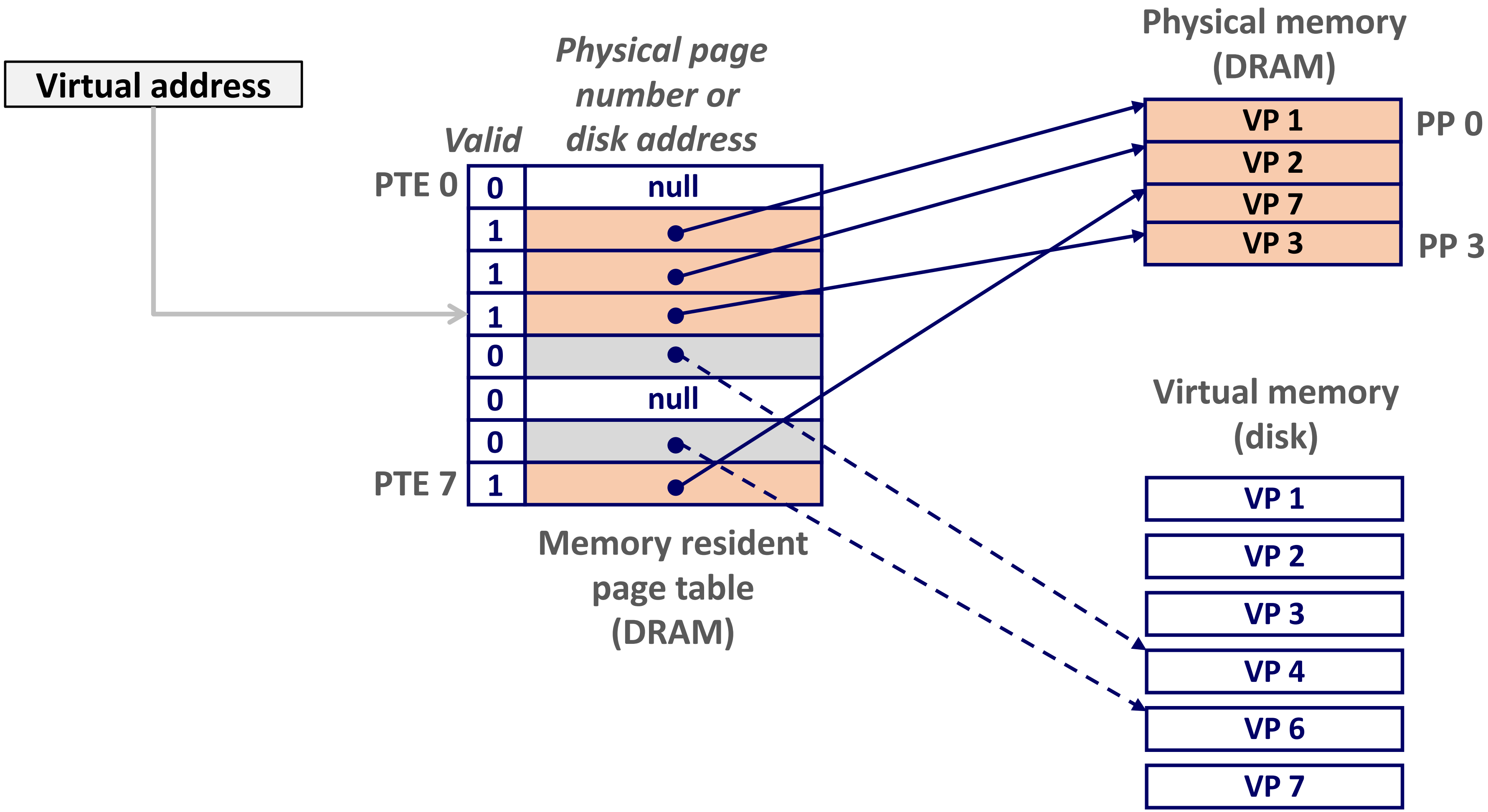
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a ***victim*** to be evicted
  - VP 4 is selected for this example



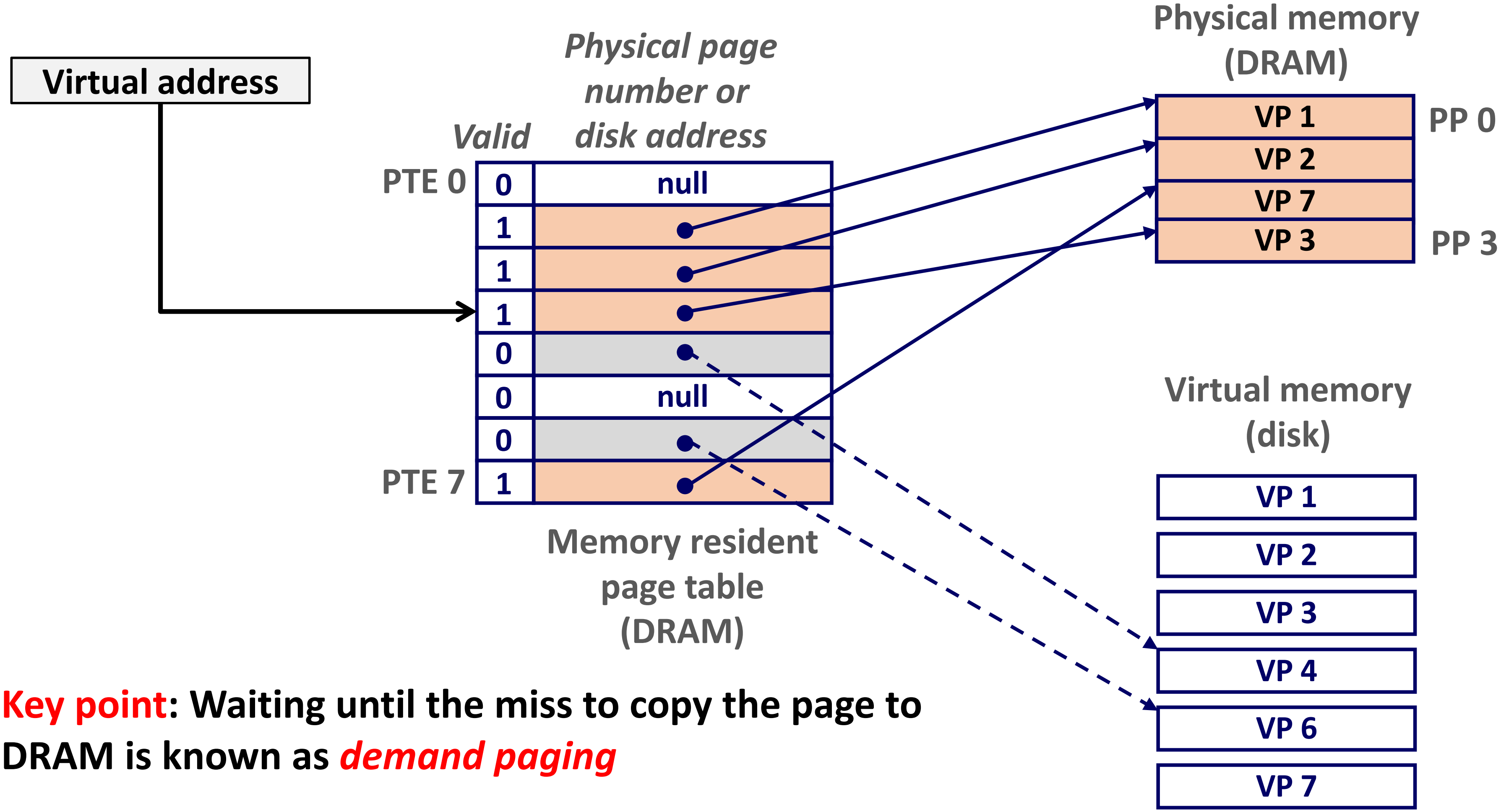
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a ***victim*** to be evicted
  - VP 4 is swapped out to disk, and VP 3 is now swapped into DRAM



Handling Page Fault

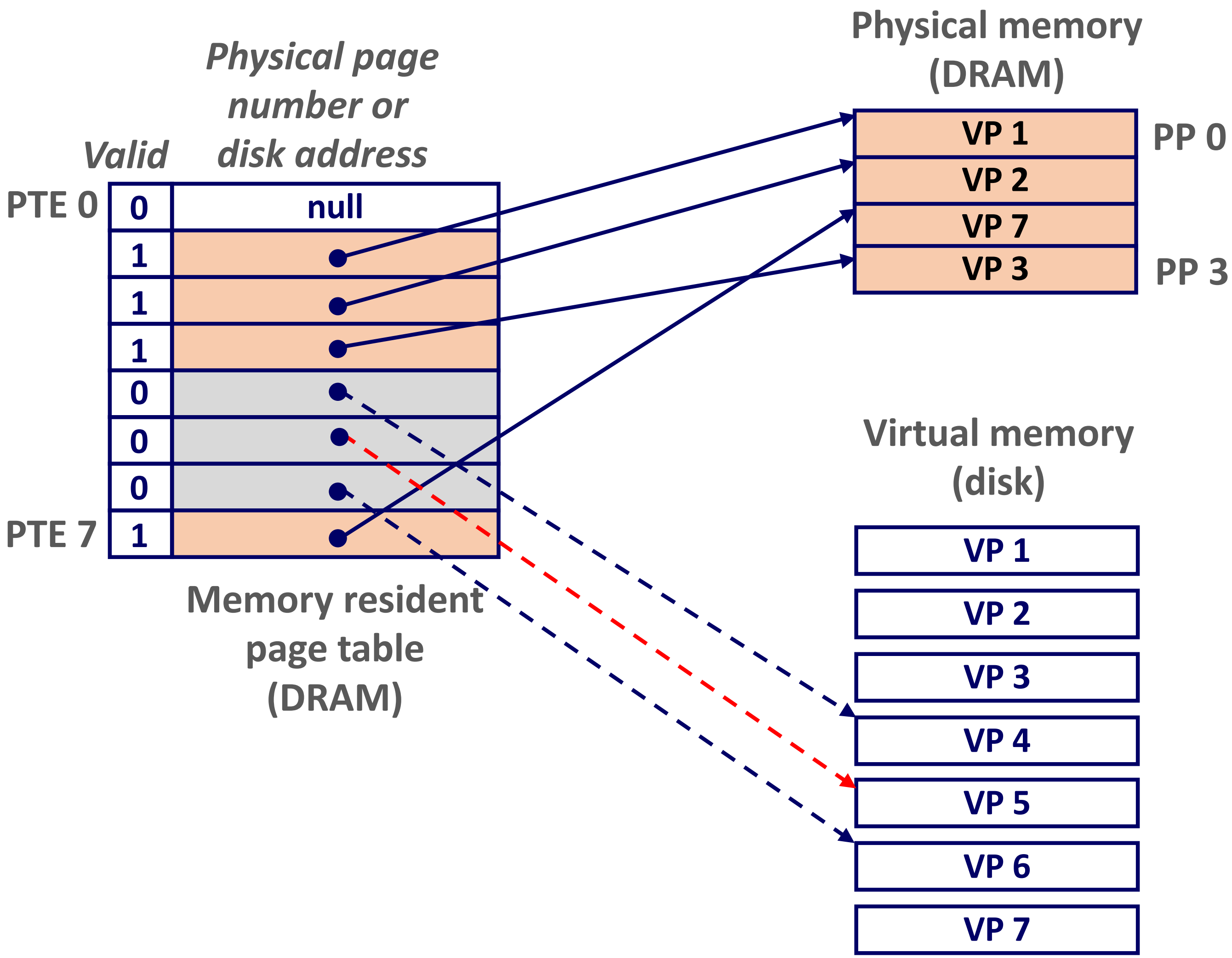
- Page miss causes page fault (an exception)
- Page fault handler selects a ***victim*** to be evicted (here VP 4)
- Offending instruction is restarted ➔ Now page hit!



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



Virtual Memory Has Huge Miss Penalty

- Virtual memory seems terribly inefficient, but it works well
- ***Locality*** greatly helps virtual memory again!!
- 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

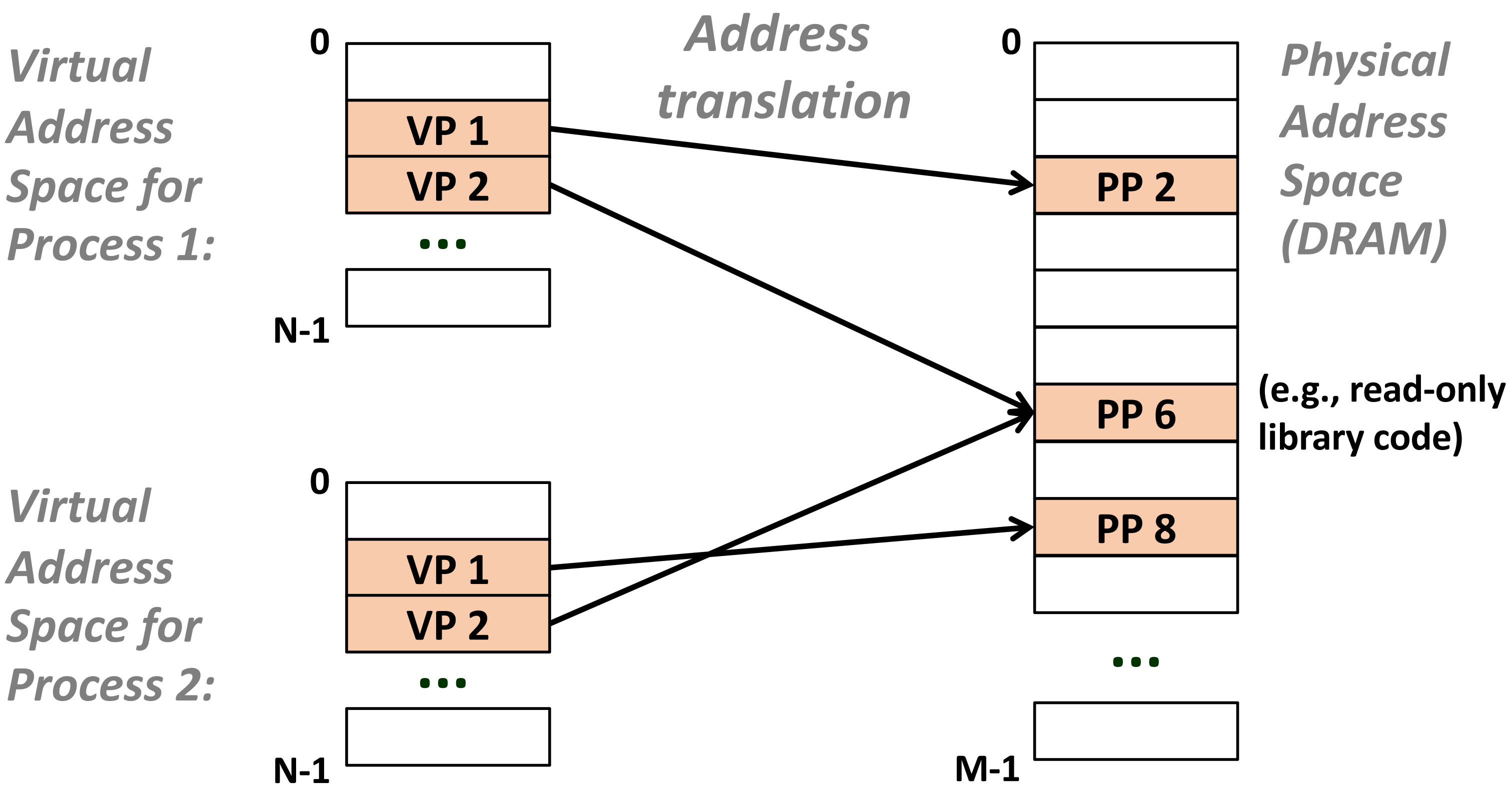
- working set size < main memory size
  - Good performance for one process after compulsory misses
- SUM(working set sizes) > main memory size
  - ***Thrashing***: Performance meltdown where pages are swapped (copied) in and out continuously

Today

- Address spaces
- General cache concepts
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Memory mapping

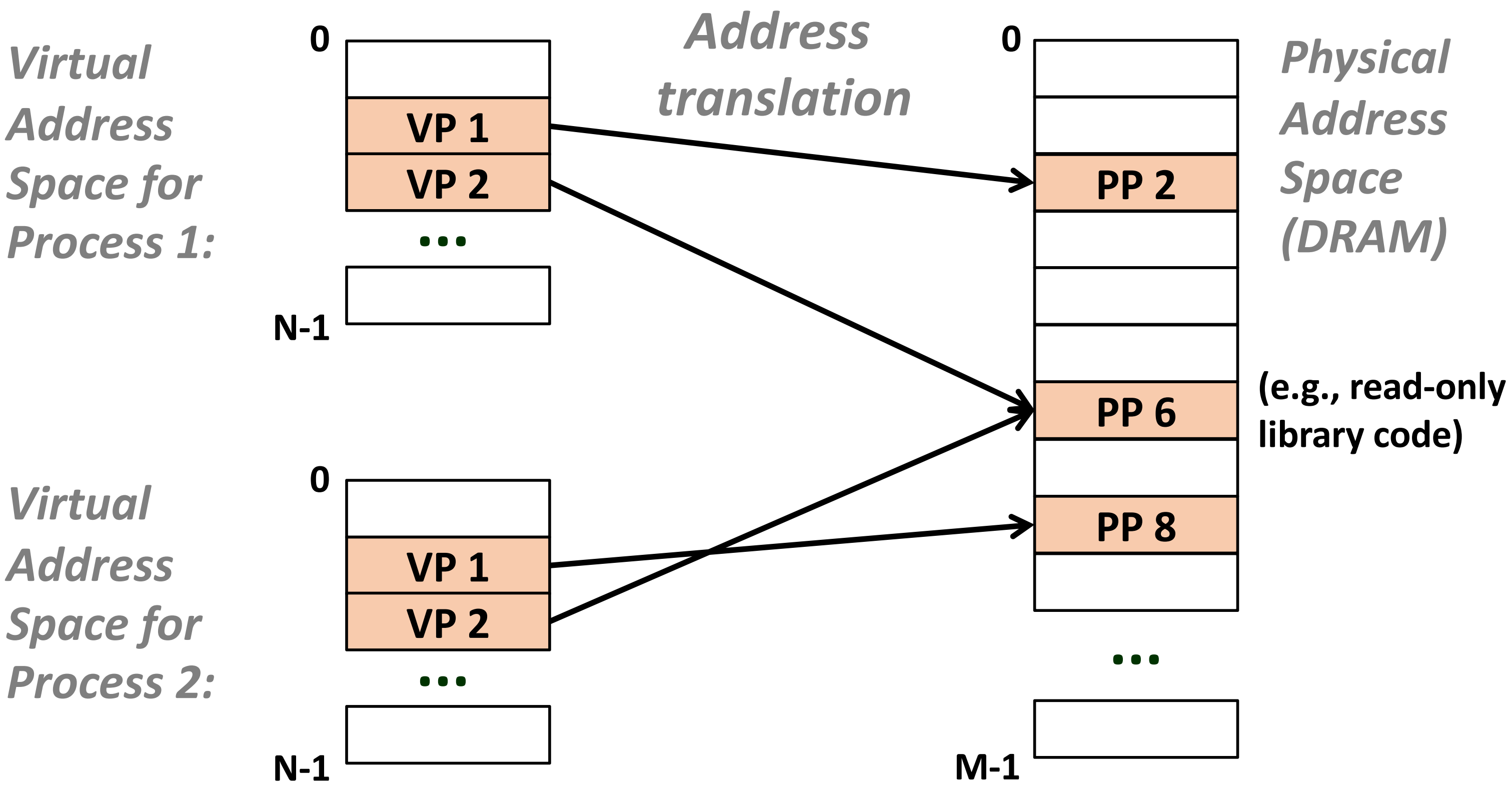
VM as a Tool for Memory Management

- 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***



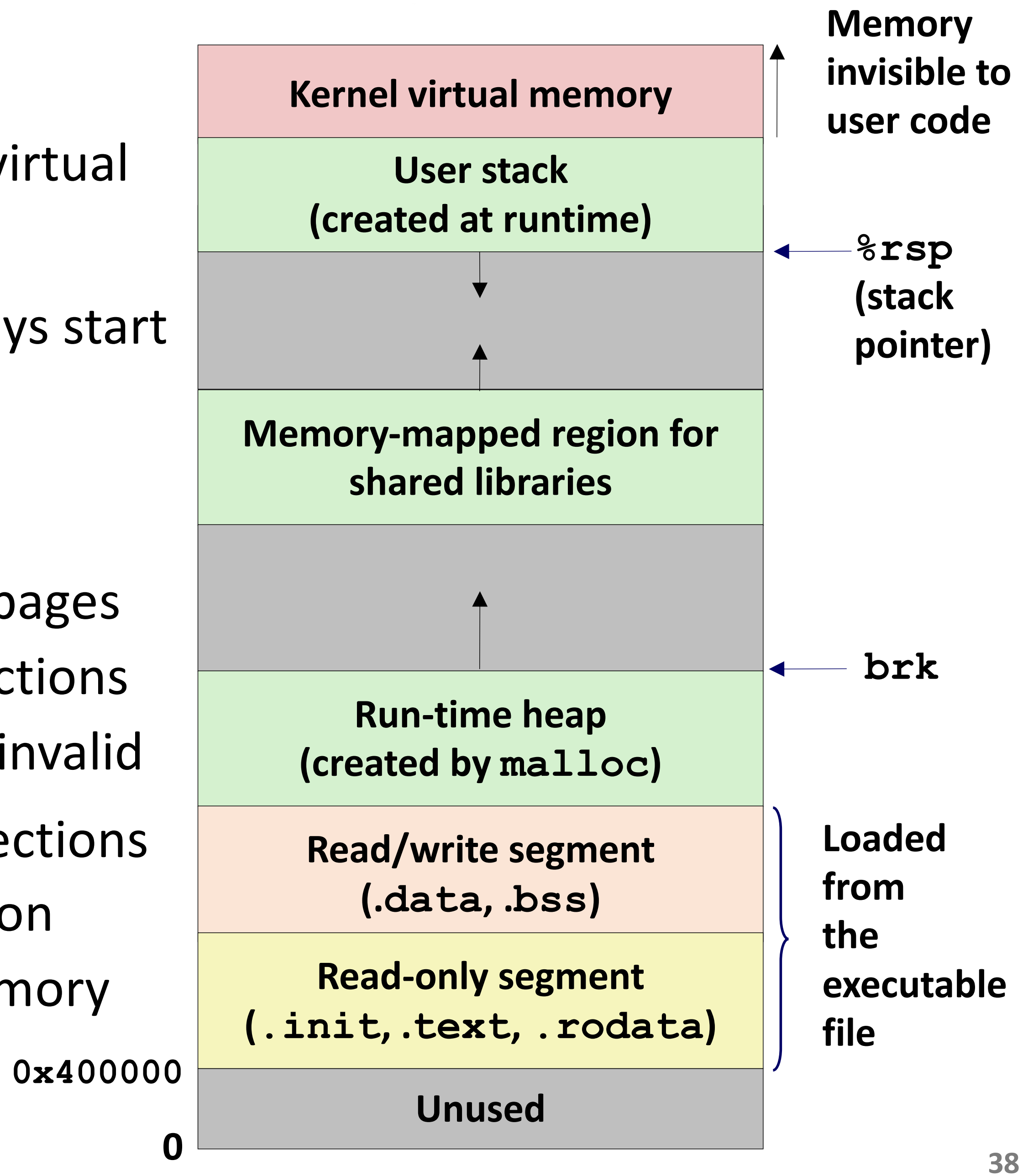
VM as a Tool for Memory Management

- 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
- 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



Today

- Address spaces
- General cache concepts
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Memory mapping

VM as a Tool for Memory Protection

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

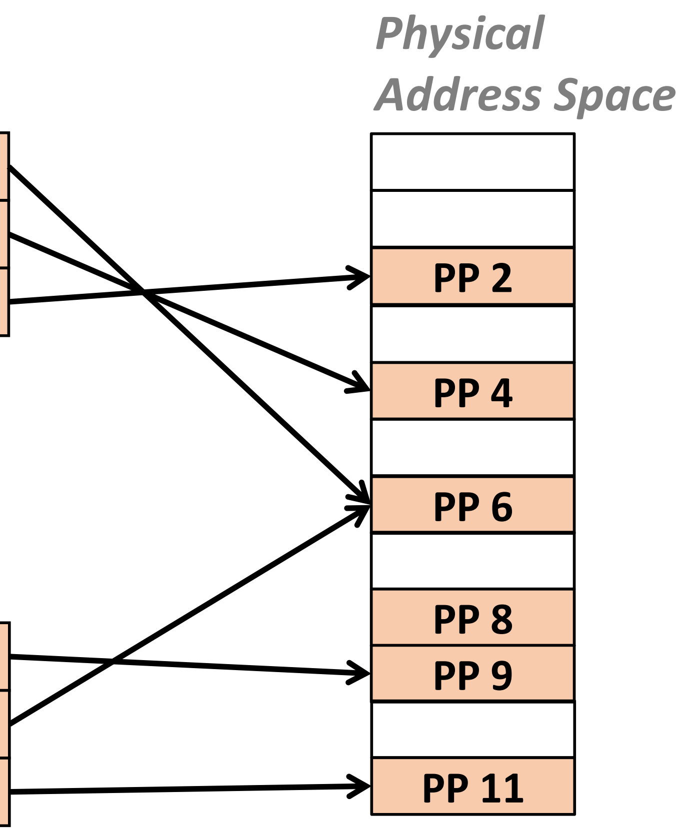
Process i:

	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



Today

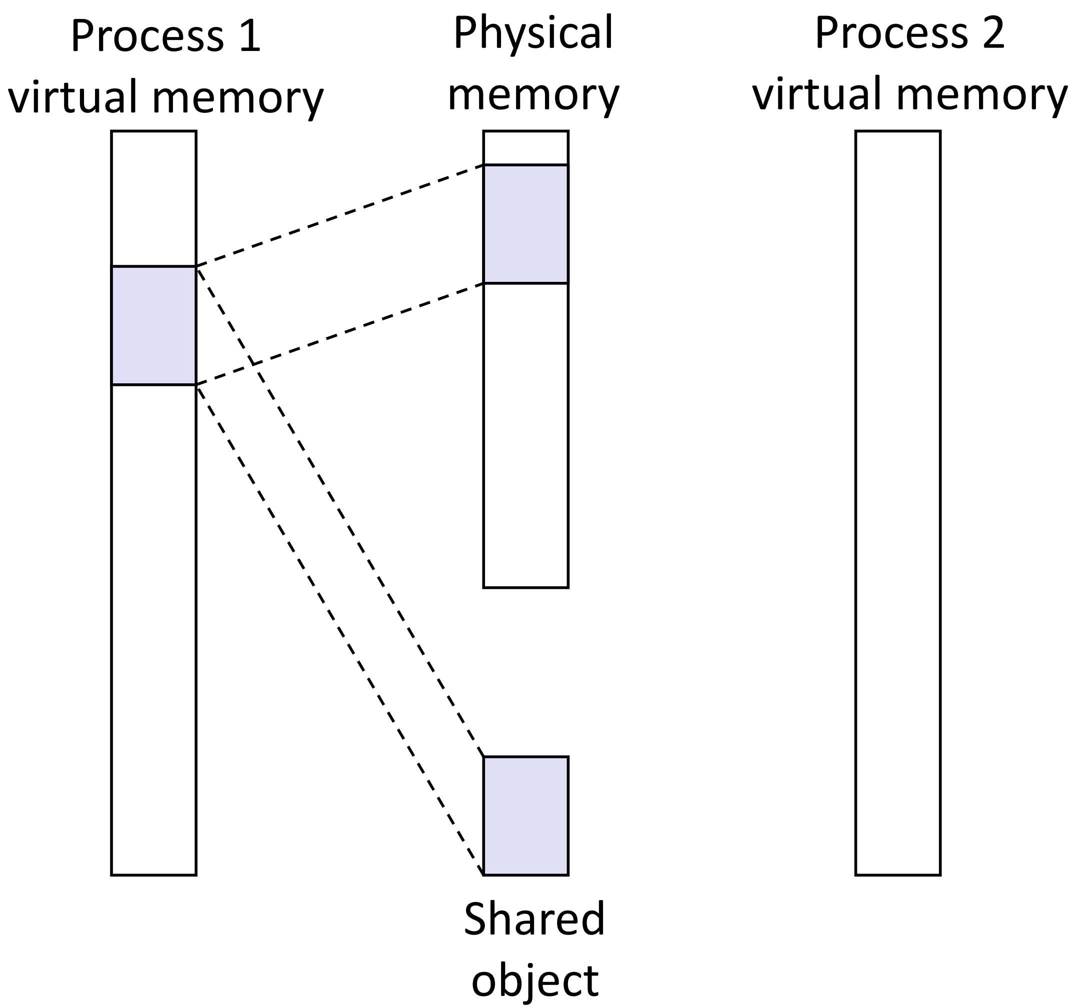
- Address spaces
- General cache concepts
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Memory mapping

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 (e.g., an executable object file)
    - Initial page bytes come from a section of a file
    - Zero-padded if the area is larger than the section of the 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***

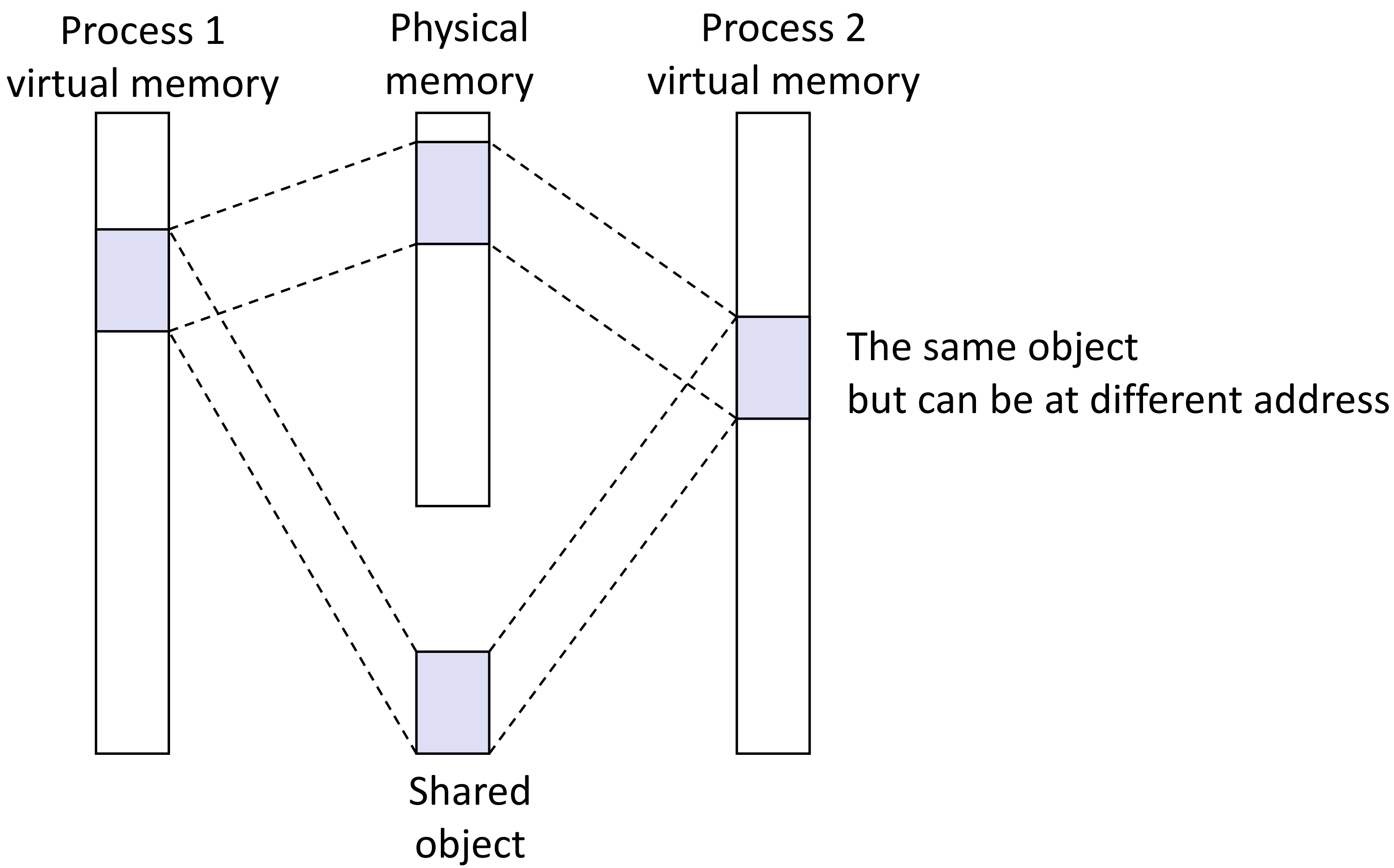
Sharing Revisited: Shared Objects

- Process 1 maps the shared object



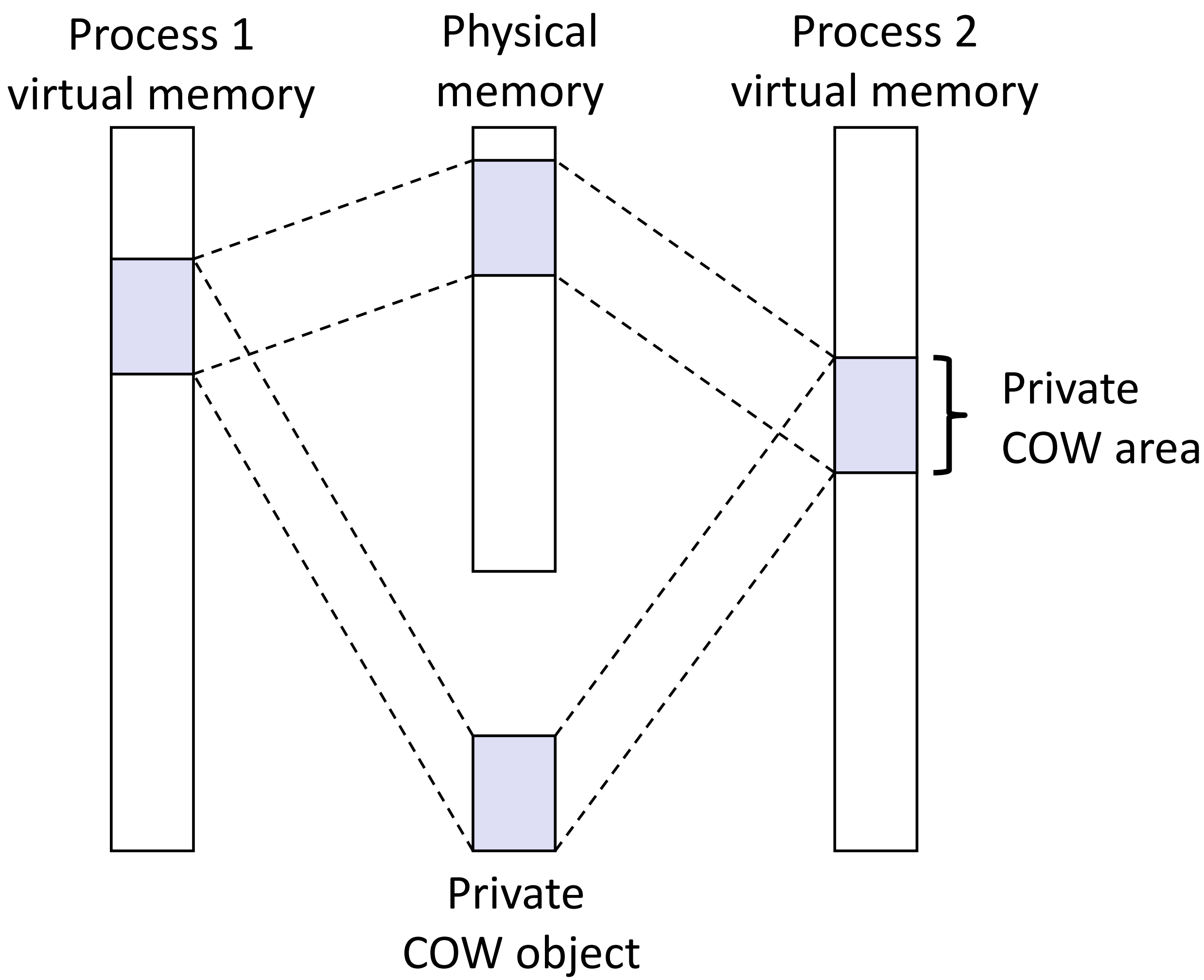
Sharing Revisited: Shared Objects

- Process 2 maps the shared object
  - The kernel identifies the file of the shared object before loading it



Sharing Revisited: Private COW Objects

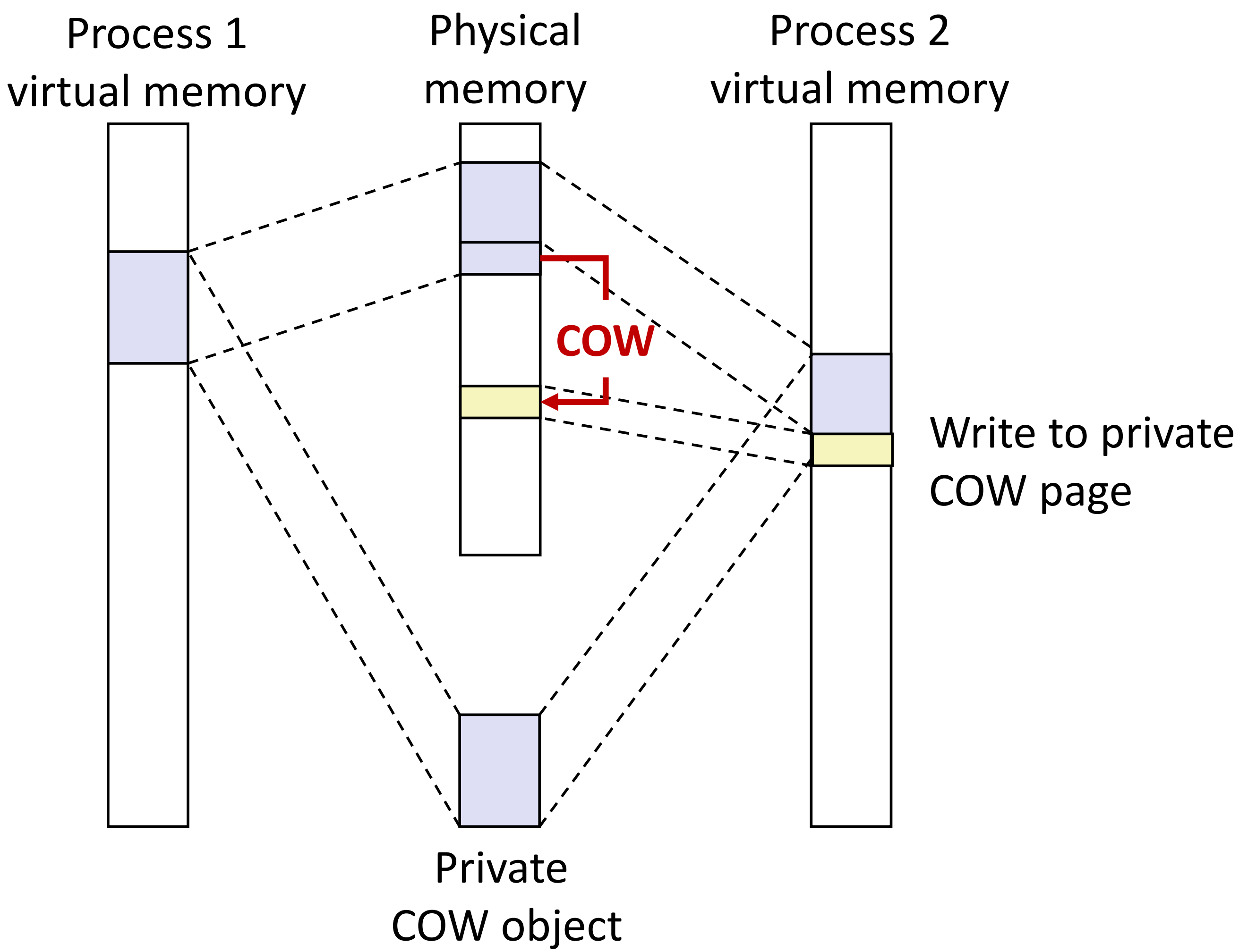
- Processes mapping a ***private copy-on-write(COW)*** object
  - Not logically shared, but physically shared



- ***Area*** flagged as private COW
- ***PTE***s in private areas are flagged as *read-only*

Sharing Revisited: Private COW Objects

- Instruction writing to private page triggers protection fault



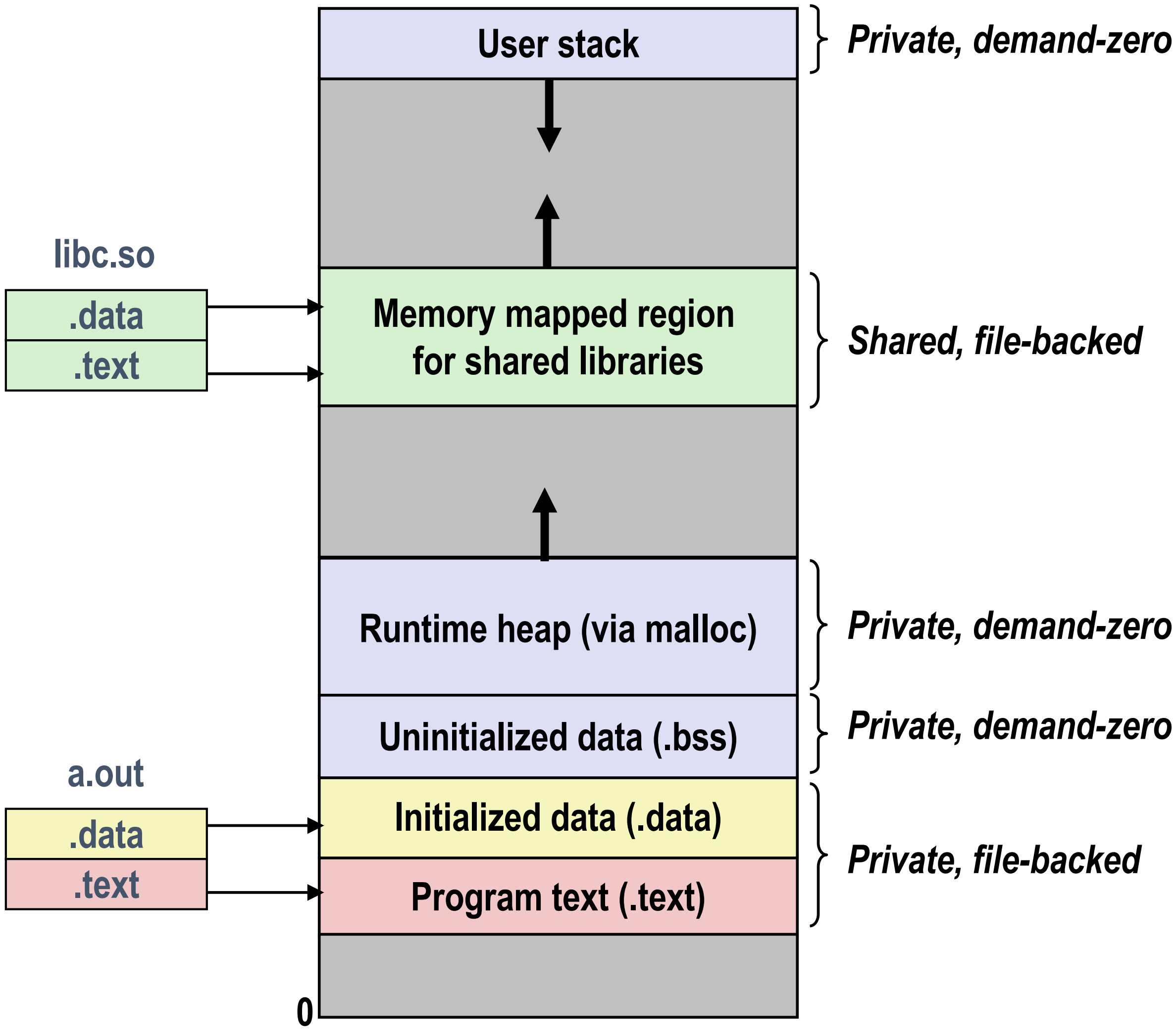
- Handler creates new R/W page
- Instruction restarts upon handler return
- Copying deferred as long as possible!

The `fork()` 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 copies of current `mm_struct` `vm_area_struct` and page tables
  - Flag each page in both processes as read-only
  - Flag each `vm_area_struct` in both processes as private COW
- On return, each process has exact copy of virtual memory
- Subsequent writes create new pages using COW mechanism

The `execve()` Revisited



- To load and run a new program `a.out` in the current process using `execve`:
- Free `vm_area_struct`'s and page tables for old areas
- Create `vm_area_struct`'s and page tables for new areas
  - Programs and initialized data backed by object files.
  - `.bss` and stack backed by anonymous files .
- Set PC to entry point in `.text`
  - Linux will fault in code and data pages as needed.

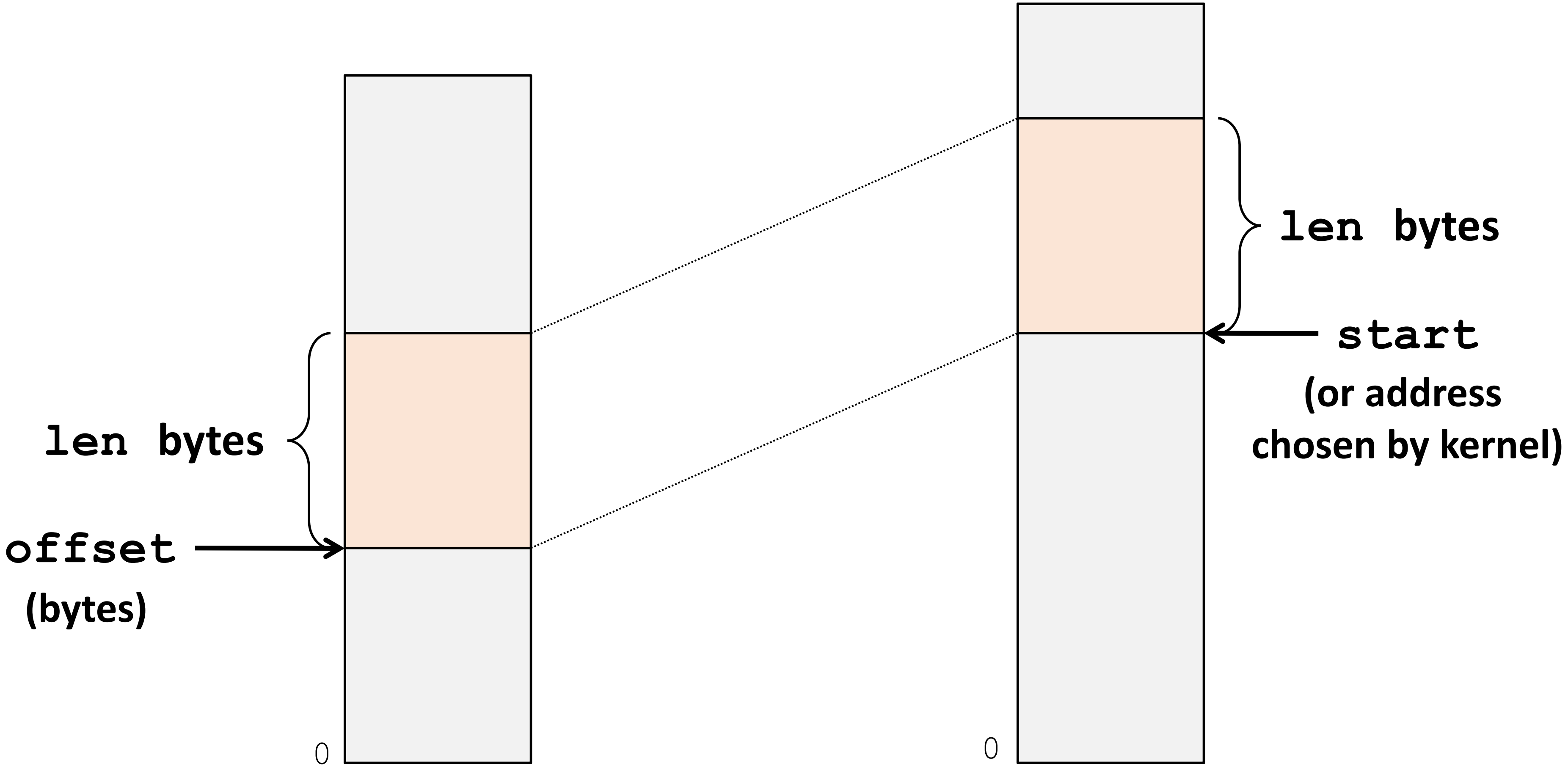
User-Level Memory Mapping

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

- Map `len` bytes starting at offset `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_ANON, 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 tranferring data to user space

```
#include "csapp.h"

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);
    /* Unmap the area from Ptr */
    Munmap(bufp, size);
    return;
}
```

mmapcopy.c

```
/* 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);
}
```

mmapcopy.c

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 only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions