Virtual Memory
Part I

CS 105: Computer Systems
Lecture 22

Prof Melissa O'Neill

April 15, 2026

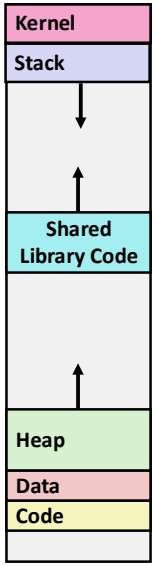
Learning Goals

- **Motivate the idea of virtual memory (VM)**
  - Provide abstraction of main memory (DRAM) for programmers
- **Understand the role of the memory management unit (MMU) and pages tables for translating virtual addresses into physical addresses**
- **Describe what happens when there is a page table *hit* vs. a page table *fault***

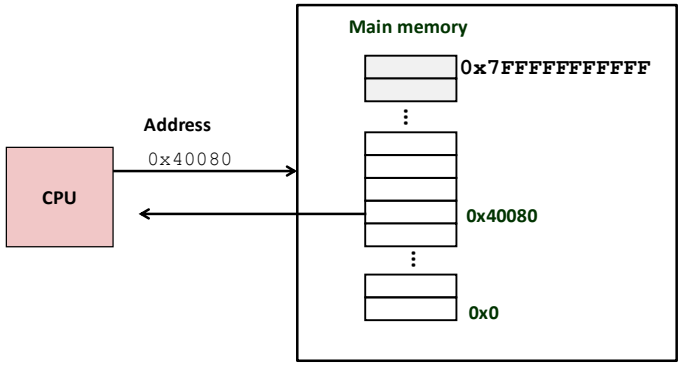
- **Quiz 5**
  - **Caches: how to use a memory address to do a lookup in a cache; array reference patterns and fitting array elements into a cache block; calculating misses**
  - **Virtual memory: determining number of virtual pages; number of bits in parts of virtual address and physical address**

Our view of a process's memory so far

- **Stack**
  - Runtime stack
  - E. g., local variables not in registers
- **Shared library code**
  - Executable machine instructions
  - Read-only
- **Heap**
  - Dynamically allocated as needed
  - When call `malloc()`, `calloc()`, `new()`
- **Data**
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- **Code (your program)**
  - Executable machine instructions
  - Read-only



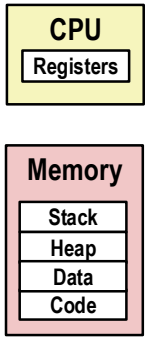
Processor (and process) view of memory



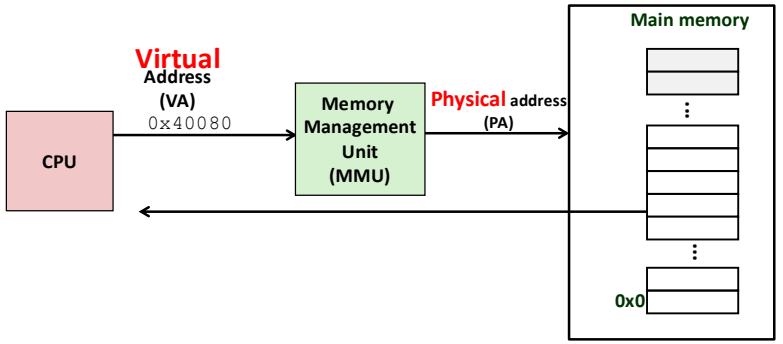
Multiple processes?

Processes

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



Reality: Reading Memory



Any memory address you see as a programmer is a virtual address!

Big Idea: *Virtualize* Memory

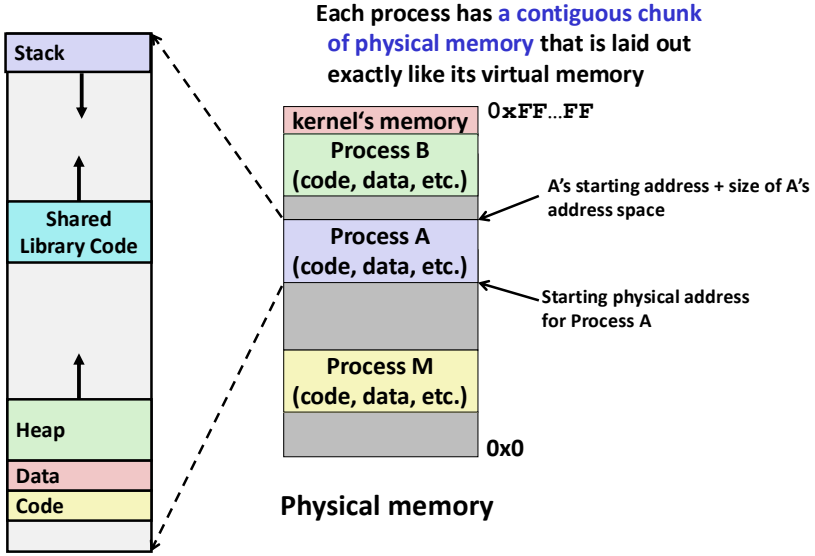
- ***Goal*: the operating system creates an abstraction of physical memory for each process; a virtual address space**
- **Virtual address space** for each process:
  Set of N = 2^n virtual addresses → {0, 1, 2, 3, …, N-1}
- **Physical address space**:
  Set of M = 2^m virtual addresses → {0, 1, 2, 3, …, M-1}
- **How to implement this abstraction?**
  - Want to run multiple processes at same time, sharing one single physical memory
  - Each process should have private, potentially large address space

Design Questions for implementing VM

- **Where in physical memory is Process A's memory contents?**
- **How is the translation done from a virtual address to a physical address?**
- **Next:**
  - **Simple (inefficient) design** to introduce address translation ideas and demonstrate problems that are addressed in modern design
  - Modern design uses **pages** and **page tables** for address translation

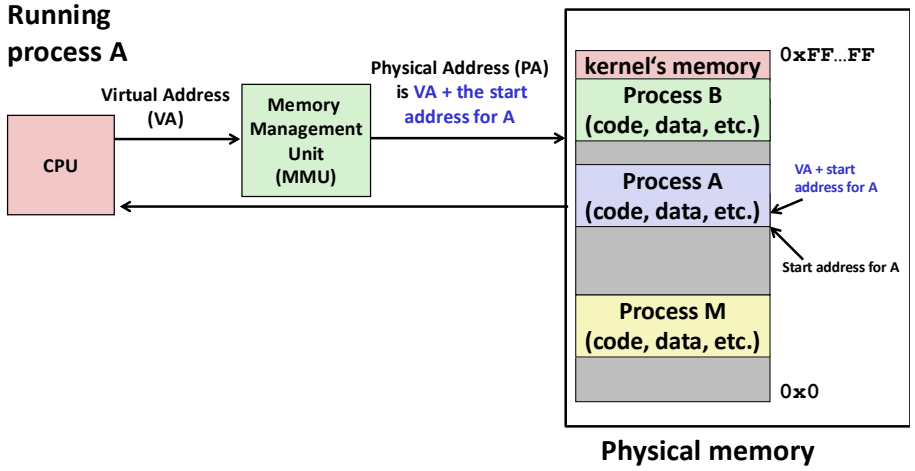
Simple (inefficient) Design

motivation only



motivation only

Simple (inefficient) Design: Address translation



Exercise: Calculating Offsets (Simple Design)

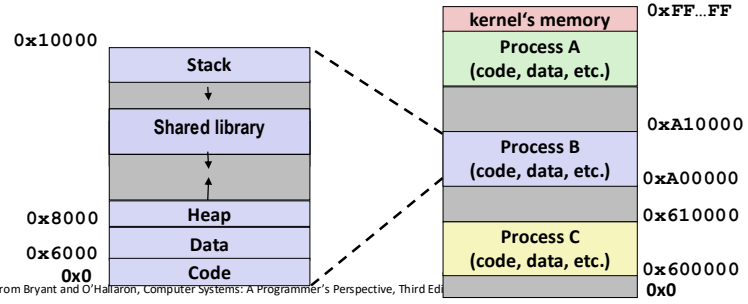
motivation only

- **Suppose we have a 0x10000-byte virtual address space, as shown below; The data region starts at address `0x6000` and the heap starts at `0x8000`.**

  Process B's physical memory starts at address `0xA00000`.

- **Consider this instruction from Process B's code:**
  `movq 0x6108, %rdi`

  Which *physical address* will be accessed?

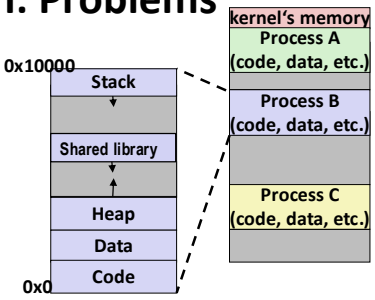


motivation only

Simple (inefficient) Design: Problems

- Consider the following program:

```
int main(argc, argv) {
  char name[512];
  printf("What is your name?" );
  scanf("%s",name);
  printf("Hello %s\n",name);
}
```



- When run, how much physical memory will a process occupy *in the simple design*? How does that compare to how much memory the process actually needs to use for this program?

- What if we ran this program concurrently in 10,000 processes?

Resolving Issues with Simple Design

- Issue: might not be able to fit the virtual address space for all processes in physical memory

  Solution: only keep *some* in physical memory!

  → use hard disk drive to store contents; working set *cached in memory*

- Issue: want to use physical memory space efficiently since it's not infinite in size

  Solution: only store the parts of processes' virtual address space that has used content

  → need a way to slice up virtual address space for finer granularity

modern design

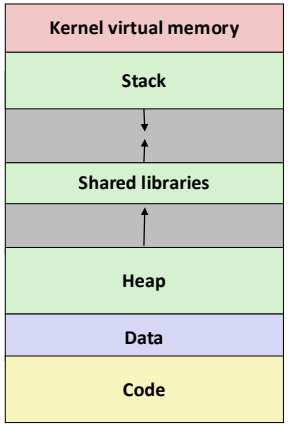
Virtual Memory: *Pages* Design

- Partition both virtual address space and physical address space into fixed-size chunks called pages
  - Only allocate space in physical memory for the virtual pages that have content on them
  - Only keep virtual pages in physical memory if they are currently in use
  - Allow processes with the same content on virtual pages to "share" them in physical memory

- Typical page size is 4-16 KB (2^{12} – 2^{14} bytes)
  - x86 uses 4K
  - ARM64 uses 16K (or 4K(

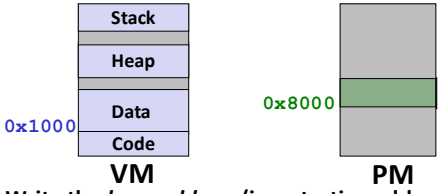
Example: Mapping VM Pages to PM Pages

Virtual memory (VM)



Exercise

- Suppose we have 14-bit virtual addresses and 20-bit physical addresses, and the page size is 4KB (4KB = 2^{12} bytes = 0x1000 bytes).
- The first page in the data segment of VM starts at address `0x1000`. It is mapped to the page of PM starting at `0x8000`.



1. Write the *base address* (i.e., starting address) for these pages in binary for both VM and PM. Show all bits for both addresses. Hint: label bits starting from the *least significant bit*

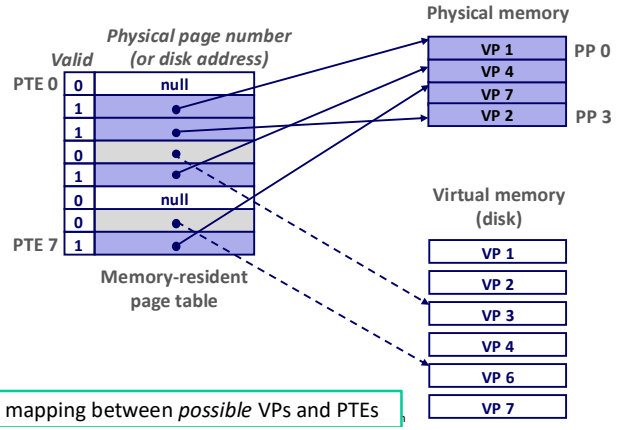
Exercise (cntd)

2. Write the virtual address 0x1400 in binary and its corresponding physical address (in both hex and binary).

3. The lower 12 bits in both addresses identify a byte within a page. Underline these 12 bits. What do you think the upper (non-underlined) bits represent in both addresses?

Page Table

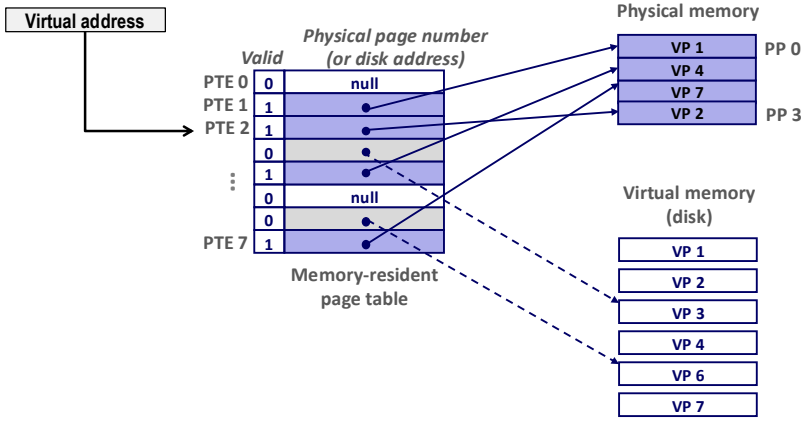
- How does MMU know which physical page goes with which virtual page? → it consults *a page table*!
- A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages; it's a per-process kernel data structure in main memory.



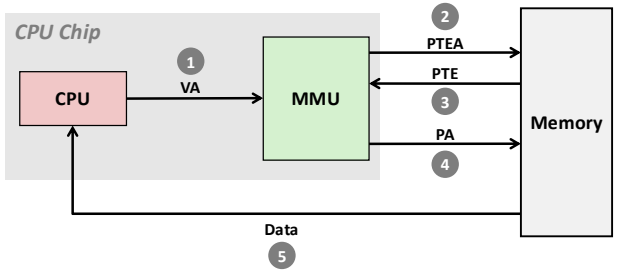
One-to-one mapping between *possible* VPs and PTEs

Page Hit

- *Page hit:* reference to virtual memory address that is currently resident in physical memory



Address Translation: Page Hit



1) Processor sends virtual address to MMU

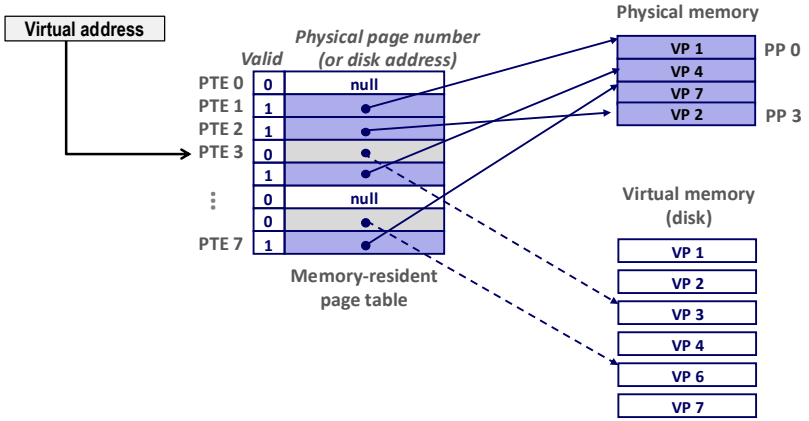
2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to memory

5) Memory sends data word to processor

Page Fault

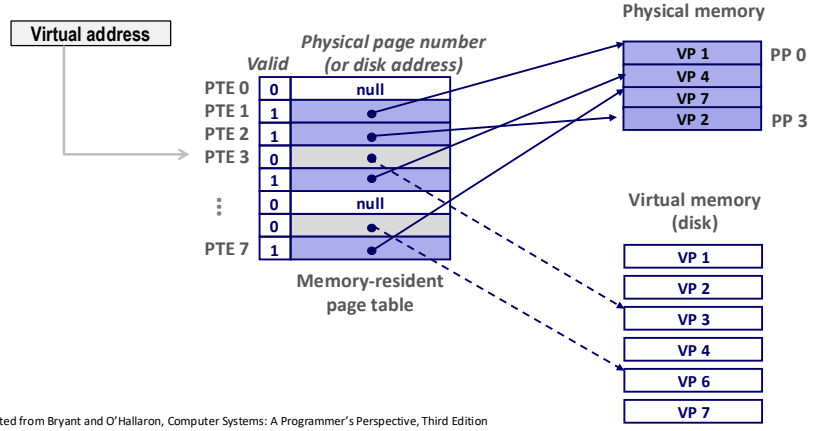
- ***Page fault:*** **reference to virtual memory address that is *not* currently in physical memory**



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

Handling Page Fault

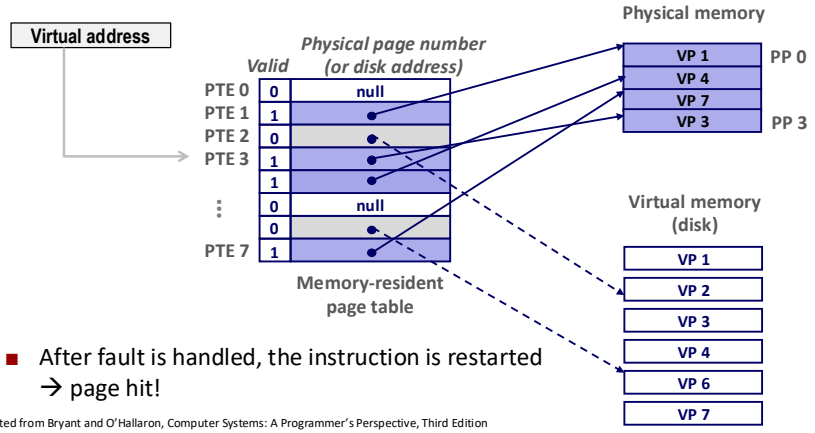
- Page miss causes page fault (an exception)
- Page fault handler selects a page in PM to be evicted (suppose it picks VP 2)



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

Handling Page Fault (continued)

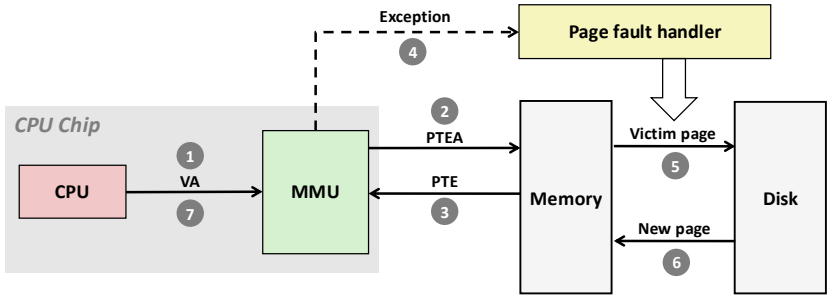
- Page miss causes page fault (an exception)
- Page fault handler selects a page in PM to be evicted (suppose it picks VP 2)
- Copies VP page 3 from disk and updates virtual page pointer



- After fault is handled, the instruction is restarted → page hit!

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

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

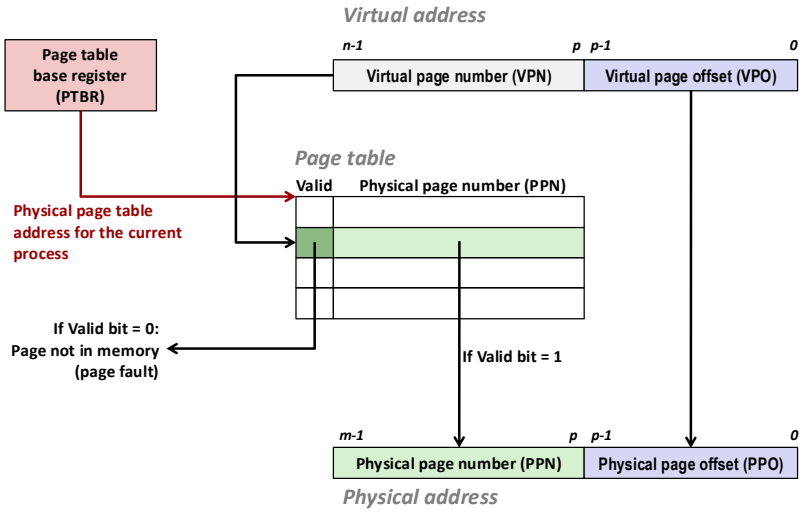
5) Handler identifies victim (and, if dirty, pages it out to disk)

6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

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

Address Translation With a Page Table



Exercise

- **Suppose you have 24-bit virtual addresses and 20-bit physical addresses. The page size is 1KB (2^{10} bytes).**

1. **How many page table entries (PTEs) will there be?**

2. **How many bits of the memory addresses are used for each of: VPO, VPN, PPO, PPN**

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

We'll talk about TLB parts later