Computer Systems

CSE 410 Autumn 2022

14 – Virtual Memory

OS "Review": Why bother with an OS?

- Application benefits
  - programming simplicity
    - see high-level abstractions (files) instead of low-level hardware details (device registers)
    - abstractions are reusable across many programs
  - **portability** (across machine configurations or architectures)
    - (ideally) code may be OS dependent, but isn't hardware dependent

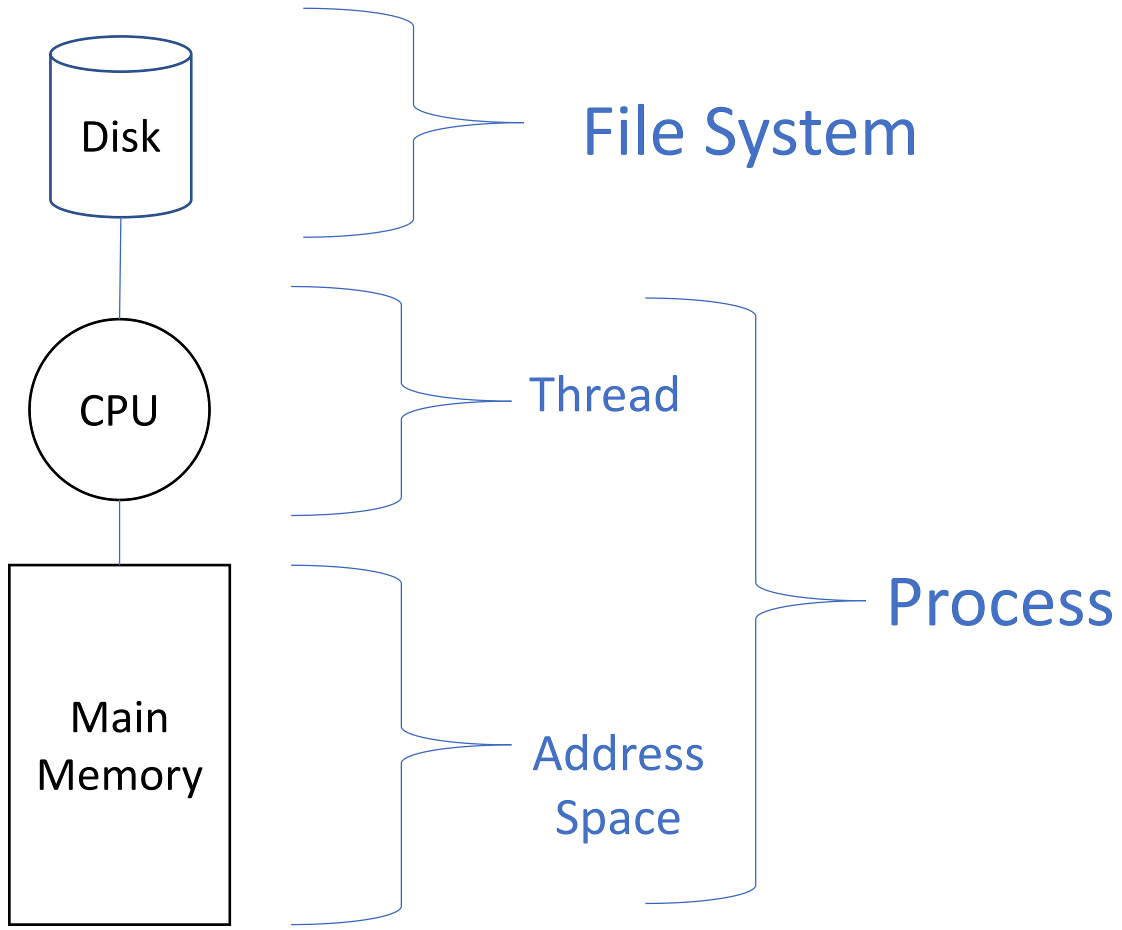
- User benefits
  - safety
    - program "sees" its own virtual machine, thinks it "owns" the computer
    - OS protects programs from each other
    - OS fairly multiplexes resources across programs
  - efficiency (cost and speed)
    - share one computer across many users
    - concurrent execution of multiple programs

The OS and hardware

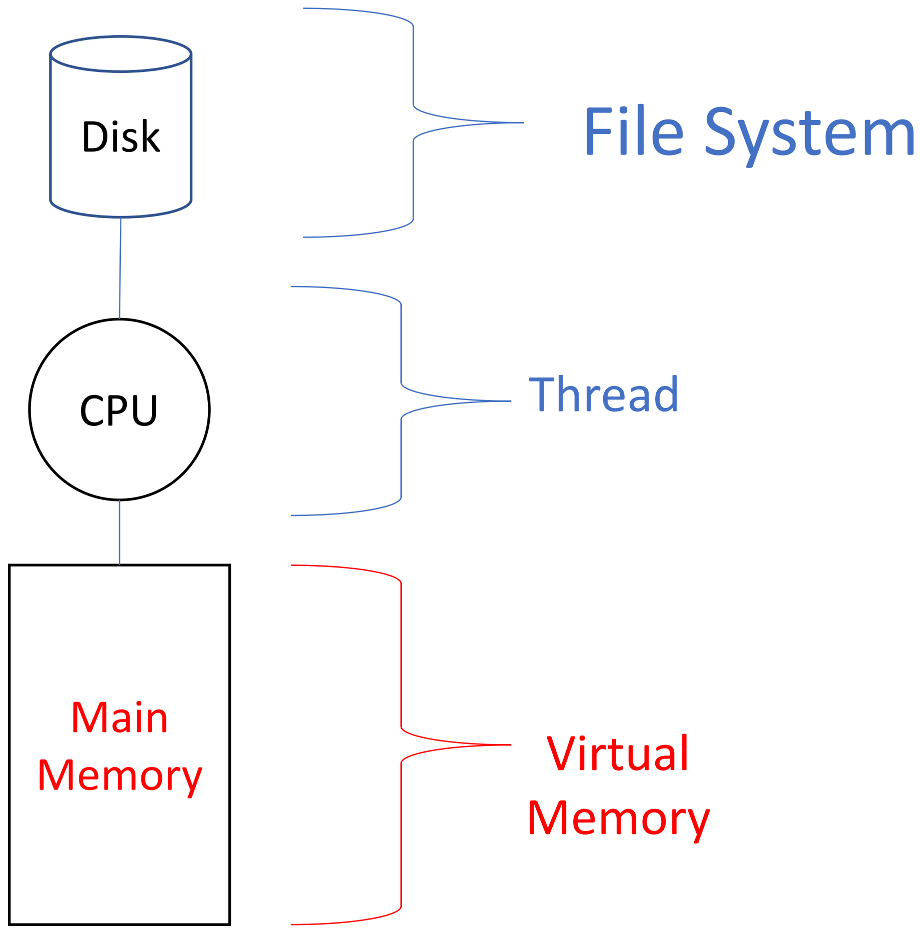
- An OS mediates programs' access to hardware resources (*sharing* and *protection*)
  - computation (CPU)
  - volatile storage (memory) and persistent storage (disk, etc.)
  - network communications (TCP/IP stacks, Ethernet cards, etc.)
  - input/output devices (keyboard, display, sound card, etc.)

- The OS abstracts hardware into logical resources and well-defined interfaces to those resources (*ease of use*)
  - processes (CPU, memory)
  - files (disk)
  - programs (sequences of instructions)
  - sockets (network)

OS Abstractions



OS Abstractions: Virtual Memory



Virtual Memory (VM)

- Overview and motivation
- Indirection
- VM as a tool for caching
- Memory management/protection and address translation
- Virtual memory example

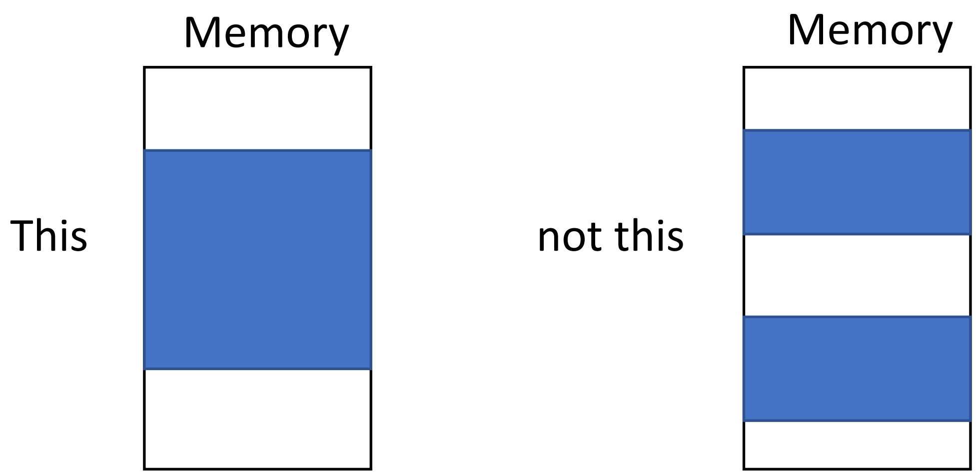
Virtual Memory Motivation

- Virtual memory:

1. Solves problems due to fragmentation
2. Provides memory protection
3. Insulates the program from considerations of the amount of physical memory available on the systems

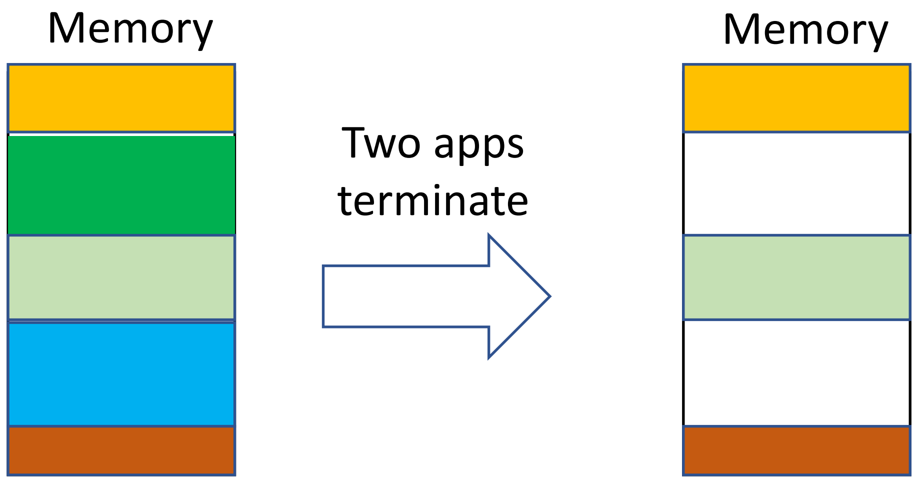
Fragmentation

- Programs need to run in contiguous hunks of memory (because the compiler assumes that – e.g., instructions, arrays)



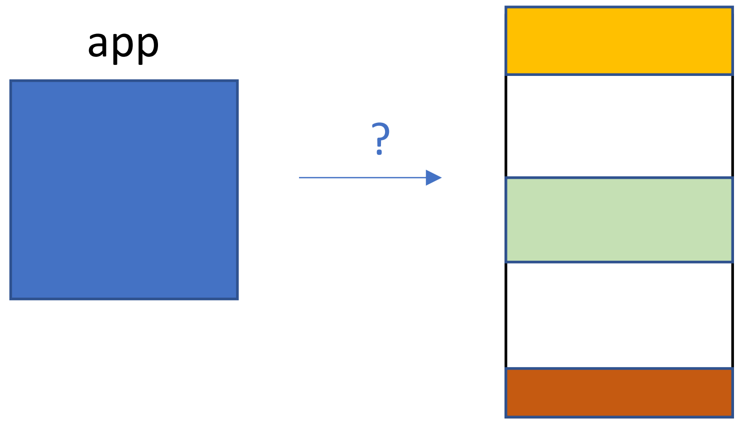
Fragmentation

- The OS supports multiprogramming
- So many programs are loaded into memory at once



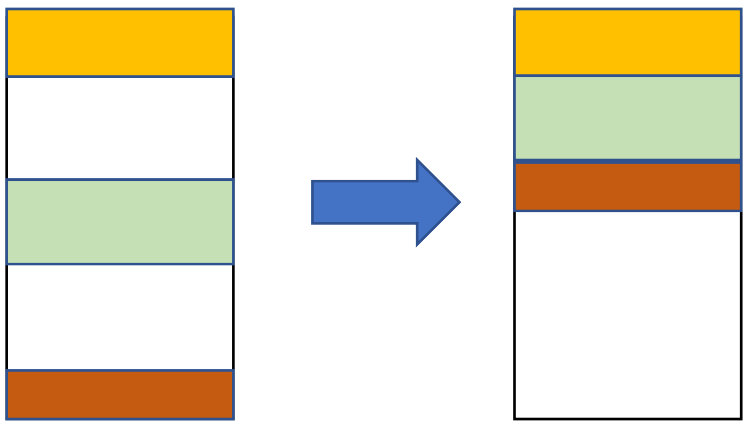
Fragmentation

- Want to start next app
- There is enough free memory for it, but...
- The free memory isn't contiguous
- Free memory is "fragmented"



Fragmentation: Possible Solutions

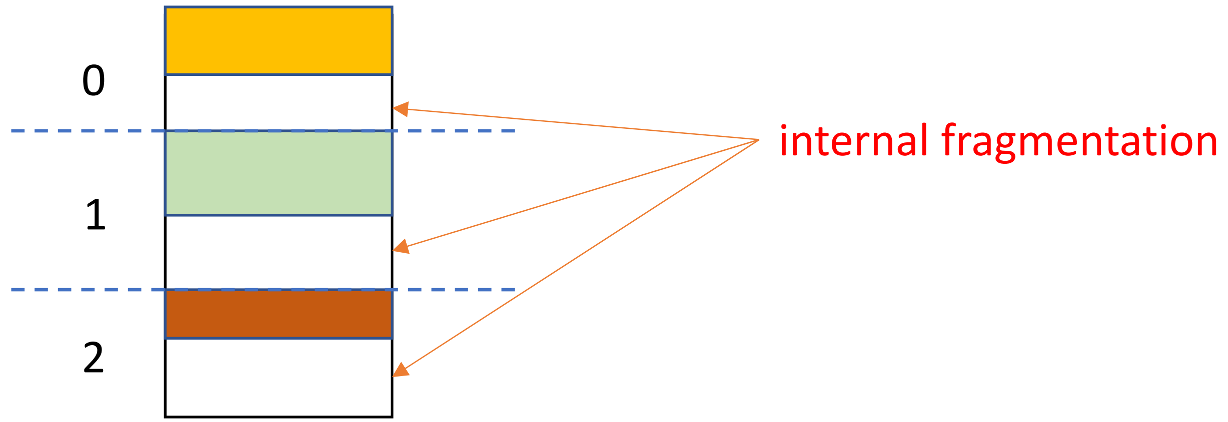
1. Move all programs in memory to coalesce free space



- *super slow*
- *doesn't work anyway (because the application may have addresses stored in registers, causing huge problems if it is suddenly moved in memory)*

Fragmentation: Possible Solutions

2. Divide memory into equal sized chunks and require every program to fit in one chunk
   (like parking spaces in a parking lot)

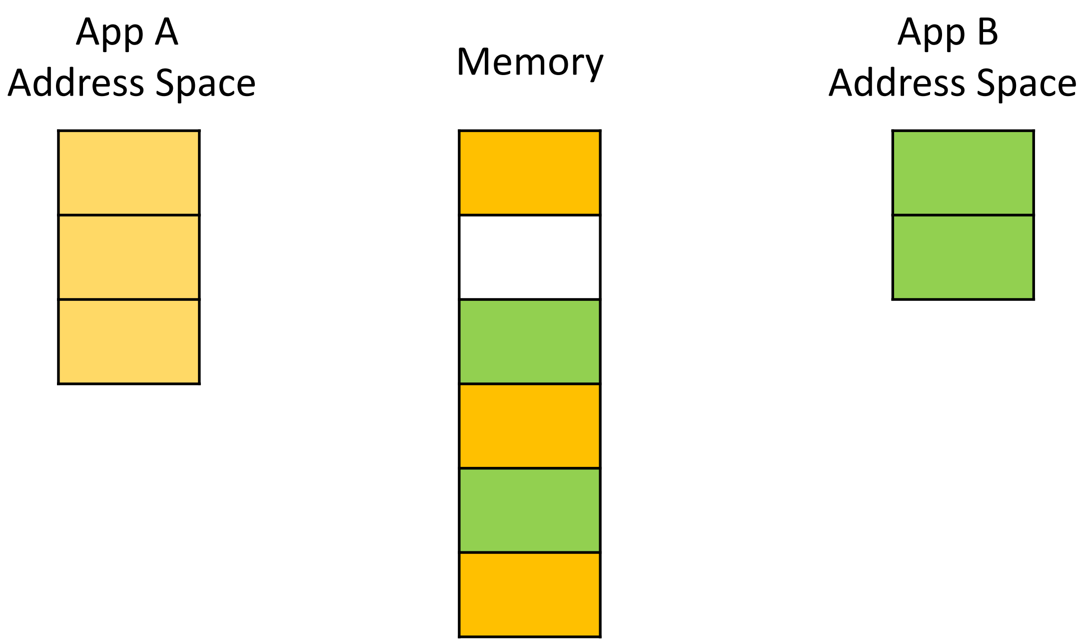


- *Pros*
  - *Any program can be loaded into any unallocated memory*
- *Cons*
  - *Internal fragmentation*
  - *program that needs more than the fixed size chunk can't ever run*

Address Translation

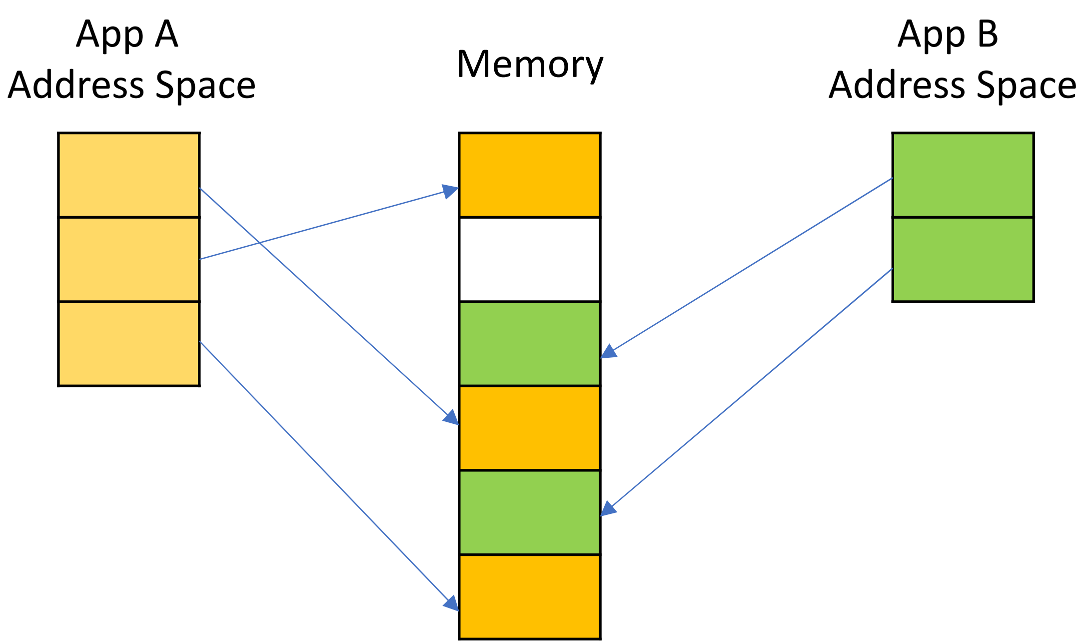
- Main idea:
  - Divide memory into small, fixed size pieces, called *frames*
  - Divide the program's memory into small, (identically) fixed sized pieces called *pages*
  - Now any page can fit in any frame. Yeah!

  - But, didn't we say that the program had to occupy contiguous memory?



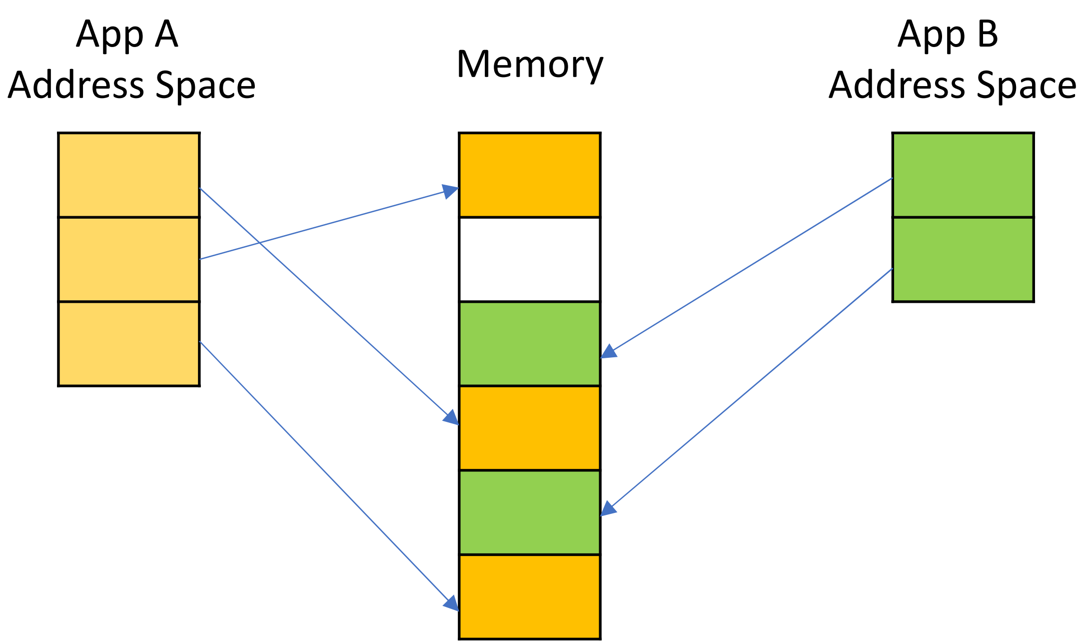
Virtual Address Space

- The program operates inside a “virtual address space”
  - Addresses are contiguous, starting at 0
  - The compiler compiles for execution in a virtual address space
- The pages of the virtual address space are “mapped” to real memory
- The hardware translates the virtual addresses issued by the program to real addresses during execution



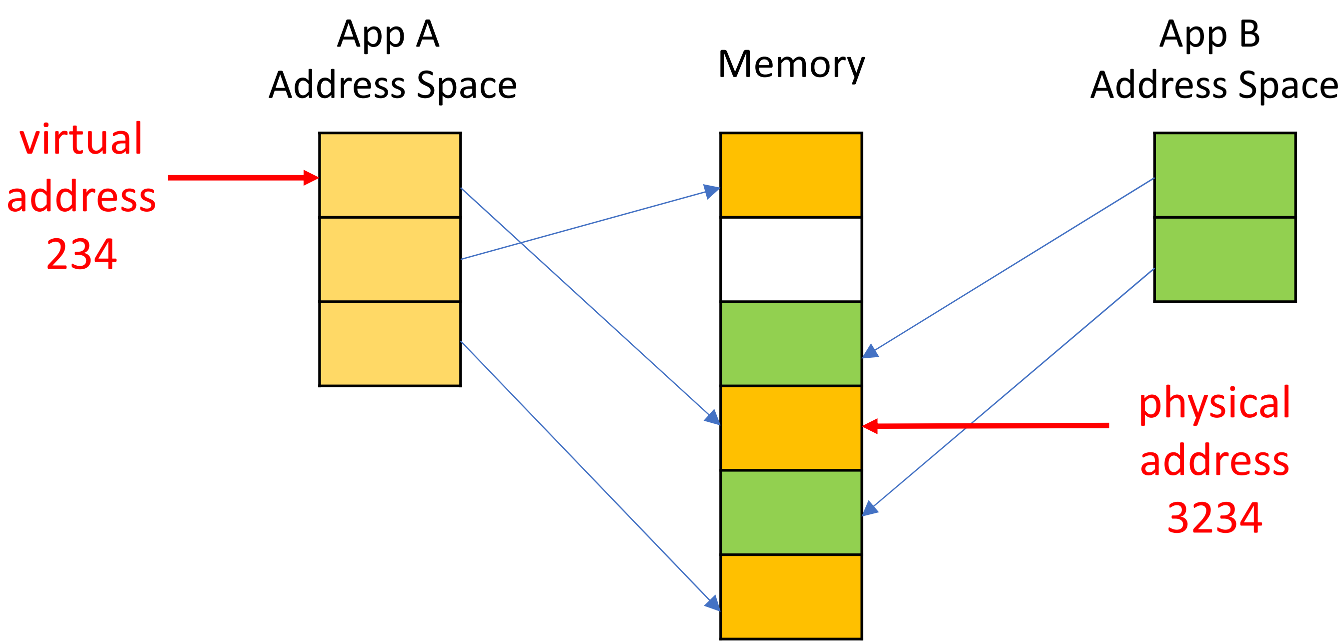
Address Translation

- *(We're going to do this in decimal, but machine would work in binary)*
- Suppose pages are 1000 bytes long
- Then virtual address 00234 in app A's virtual address space is:
  - virtual page 00, which is physical frame 3
  - at offset 234 in that page / frame
- So the physical address of 01234 would be 03234



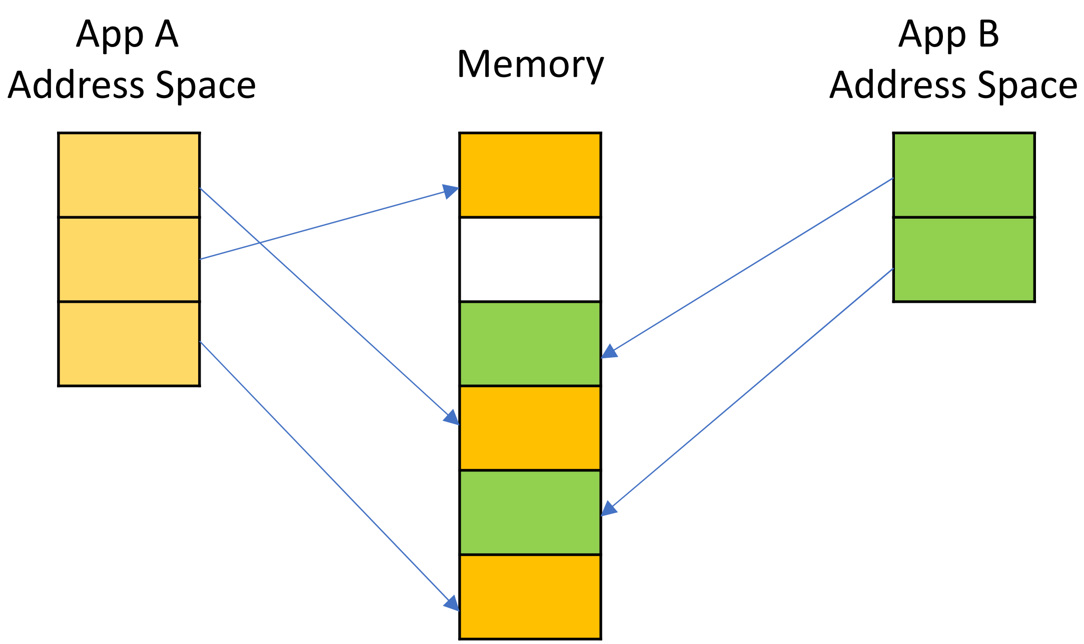
Address Translation

- Programs issue *virtual addresses*
  - Same instruction set as before
  - lw x5, 234(x0) # what does this do?
  - CPU adds 234 to x0 and gets 234 as the virtual address
- Virtual address is translated to physical address 3234 and the word there is loaded from memory



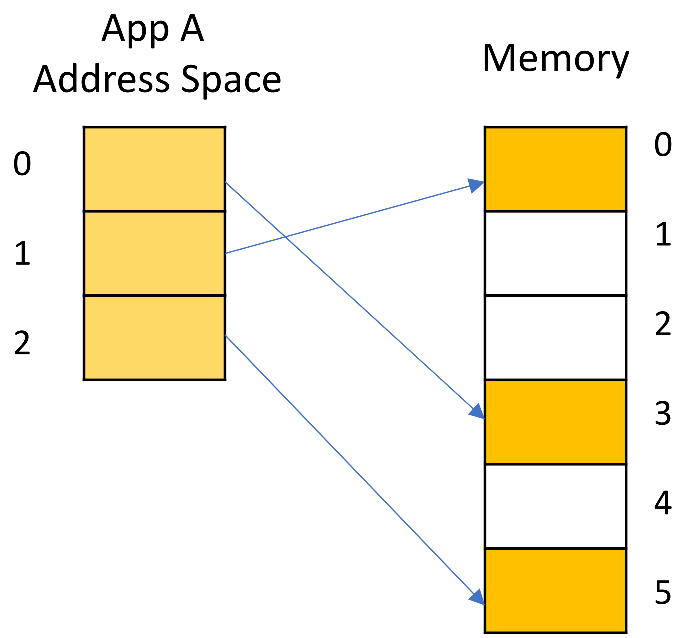
Address Translation

- We need a way to "represent the arrows in the diagram"
  - Page Tables
- We need hardware to translate from virtual addresses to physical addresses
  - The Memory Map Unit (MMU)



Page Tables

- There is a page table associated with each virtual address space
- The page table is an array, indexed by the virtual page number
- Each entry is a physical frame number
  - Each entry also has a “valid bit” indicating whether the virtual page exists

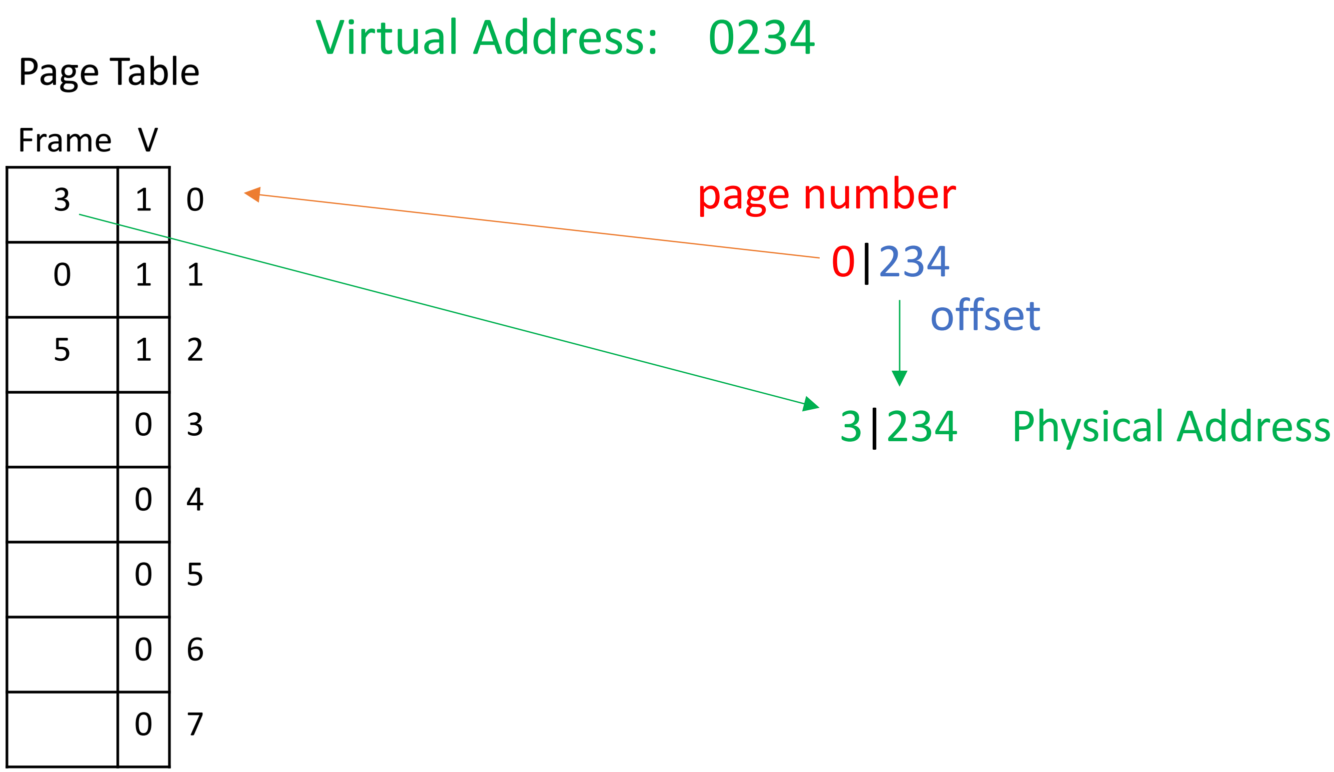


Page Table

Frame	V	
3	1	0
0	1	1
5	1	2
	0	3
	0	4
	0	5
	0	6
	0	7

Page Tables: Address Translation

- Performed by the MMU
- *Pages in our example are artificially 1000 (decimal) bytes long*

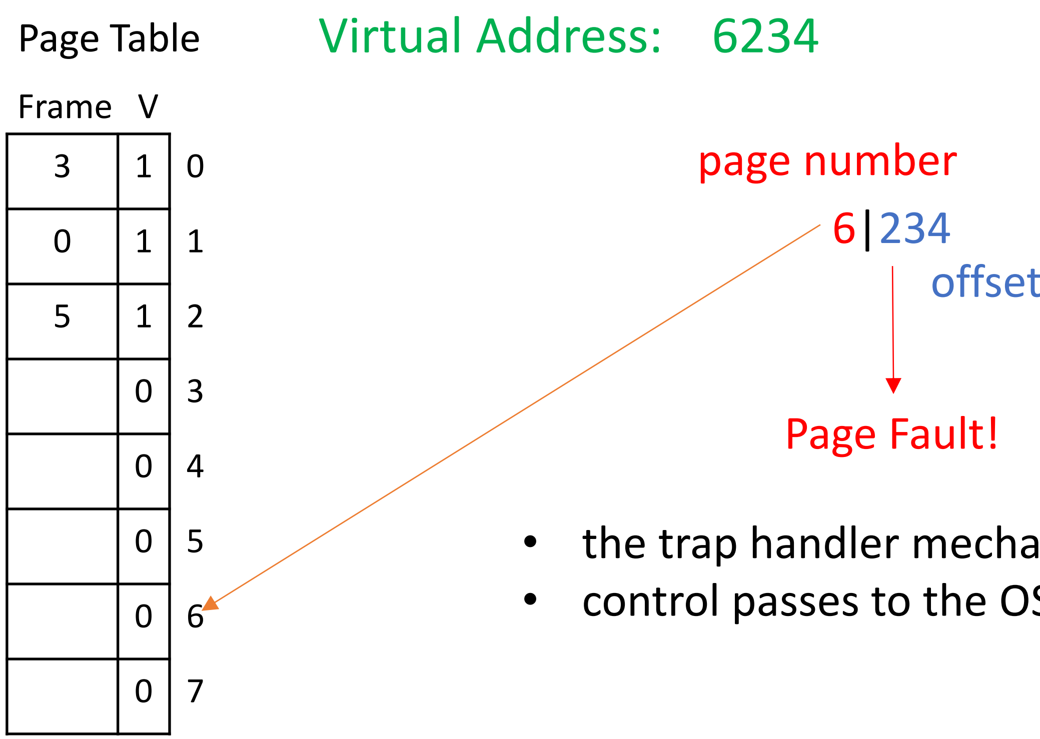


Address Translation: Memory Protection!

- The OS allocates physical memory and sets up the page table for each virtual address space
- So long as the OS ensures that the physical frames in use by application A don't appear in the page table for the virtual address space in use by application B, B cannot possibly read or write A's memory
  - None of B's virtual addresses map to the memory used by A

Page Tables: Page Faults

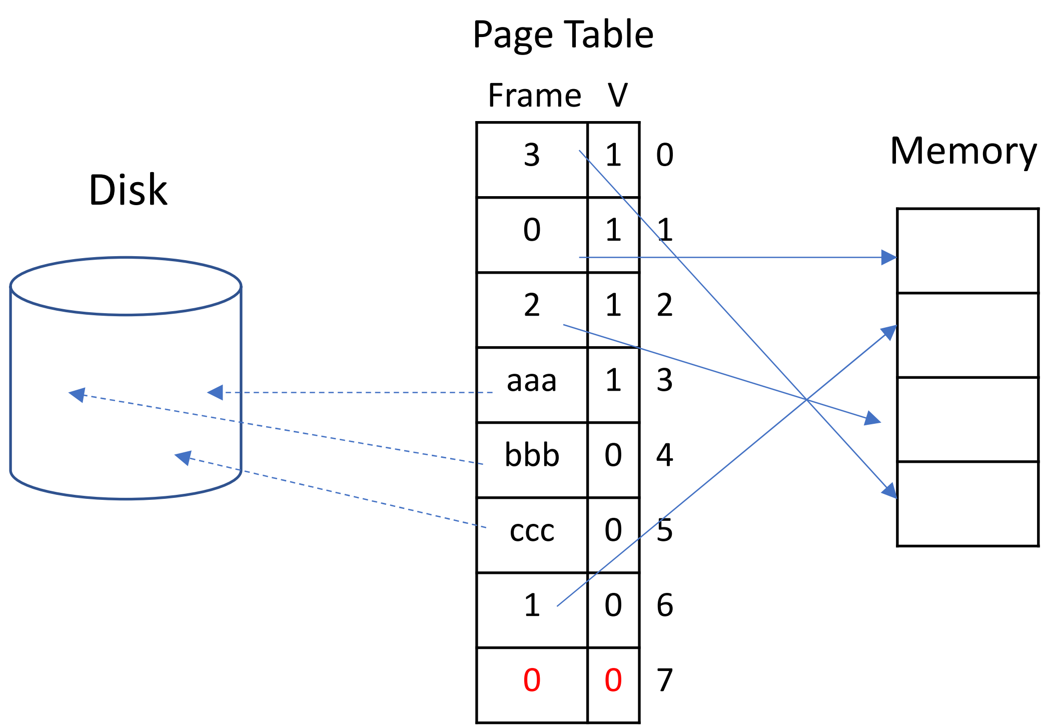
- Suppose a virtual address is issued and the page table doesn't have a valid mapping
- That means the virtual page is not loaded anywhere in memory
  - Why?



- the trap handler mechanisms is invoked
- control passes to the OS

Page Faults: Independence from Physical Memory Size

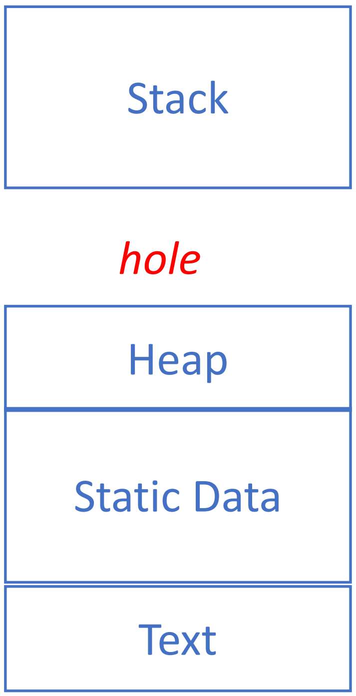
- Virtual memory supports virtual address spaces that are larger than physical memory
- Pages not current in physical memory are stored on disk



- Memory is 4 frames
- Max virtual address space size is 8 pages
- Actual VAS space is 7 pages

Page Faults: Non-Contiguous VAS

- The virtual address space can have “holes”
- References to the holes causes a page fault



Page Protection

- The MMU is fetching a page table entry on every memory reference
- So long as we're doing that, it's useful to add some additional functionality to the page table entries
- In particular, we can provide access right bits
  - read
  - write
  - (execute)
- An operation that tries to violate the page's access rights causes a page fault

- Example Page Table Entry

Frame Number	Valid	Read	Write	Execute
22	1	1	0	1

Page Protection

- Example Page Table Entry

Frame Number	Valid	Read	Write	Execute
22	1	1	0	1

- Stack: writable? readable? executable?
- Heap (space for “new”): writable? readable? executable?
- Static Data: writable? readable? executable?
- Text (instructions): writable? readable? executable?

Page Protection

- Example Page Table Entry

Frame Number	Valid	Read	Write	Execute
22	1	1	0	1

- Stack: **writable**? **readable**? executable?
- Heap (space for “new”): **writable**? **readable**? executable?
- Static Data: **writable**? **readable**? executable?
- Text (instructions): writable? **readable**? **executable**?

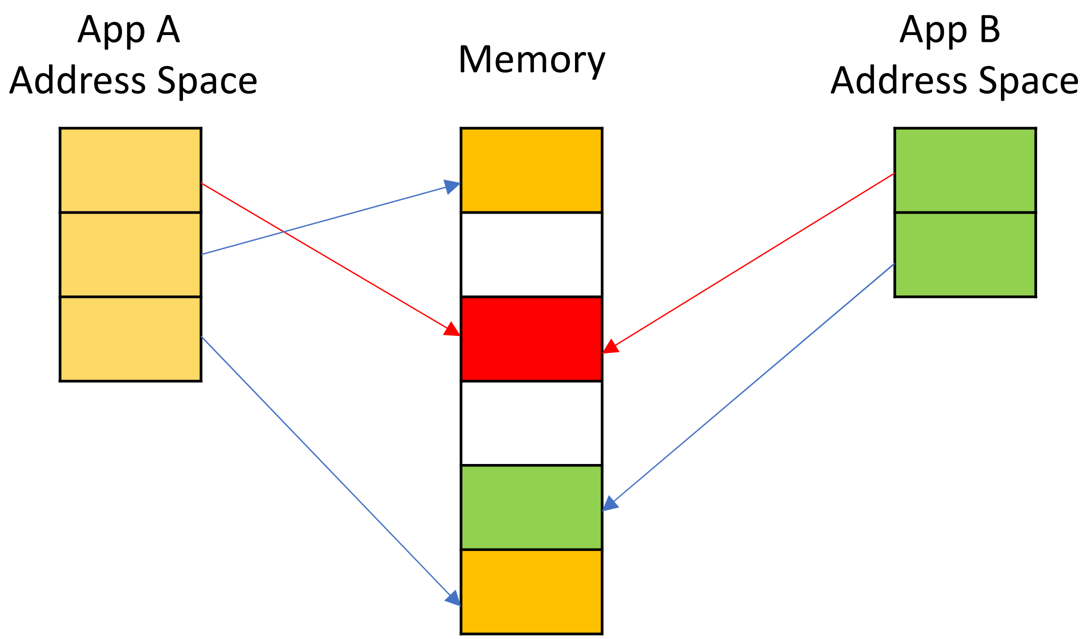
Other Uses of Page Tables

1. Sharing memory between address spaces
    1. Page table entry in two different page tables names same physical frame
    2. Writes to that page in one address space show up in the other address space
    3. Provides very fast communication from one address space to another

2. Memory Mapped I/O
    1. One way the ISA can provide control of I/O devices is to use low memory addresses “to mean” the IO devices
        1. E.g., a store word to address 224 is actually sending 32 bits to the Ethernet controller
    2. No provide protection by page

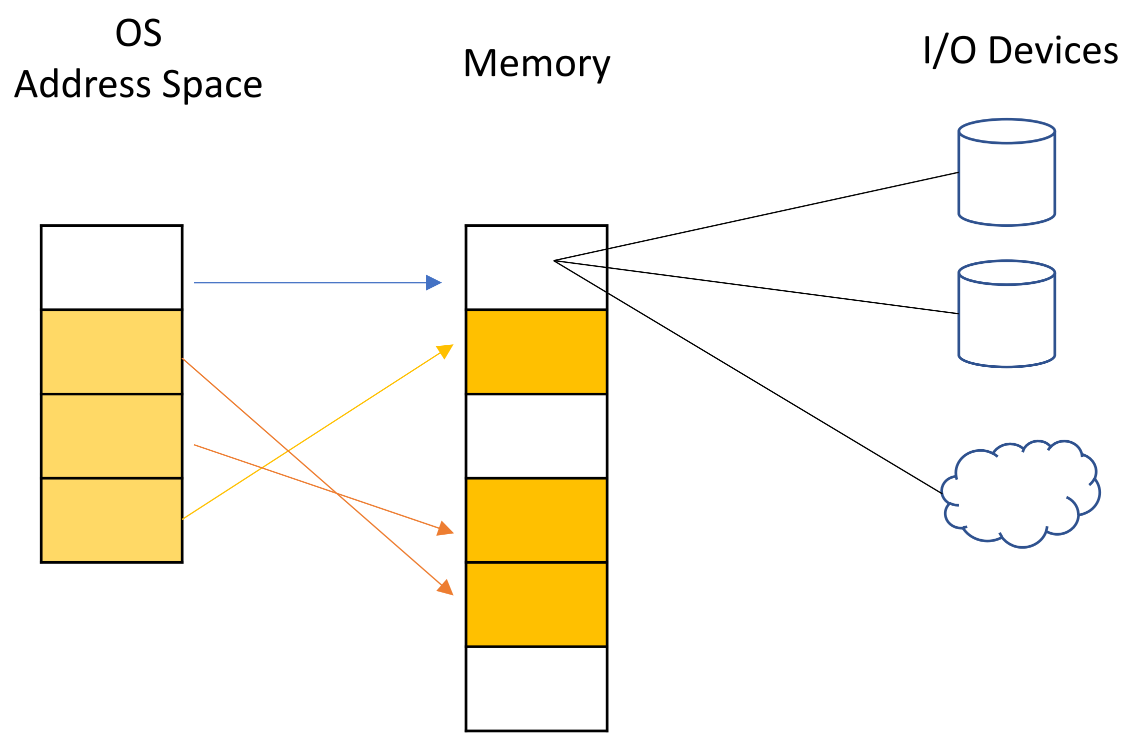
Page Tables: Shared Memory

- Communicating between address spaces is as fast as writing memory
- But usually need some kind of synchronization on top of that



Memory Mapped I/O

- The hardware recognizes that low addresses are to be sent to the IO devices, and are not talking about memory
- Only the page table for the OS maps any virtual addresses to the low physical address



Virtual Memory: Summary 1

- Each running program has its own virtual address space
- Compilers know this and compile code as if all of memory belongs to the program being compiled
  - It does, it's just that it's virtual memory
- Virtual memory relies on address mapping
- The operating system maintains a page table for each virtual address space that maps virtual pages to physical frames
- Page table entries have access permission bits as well as the mapping information
- A page fault occurs when accessing a virtual address:
  - that isn't currently mapped to physical memory, or
  - with an operation that isn't allowed on that page (e.g., writing)

Virtual Memory: Summary 2

- Virtual memory solves the fragmentation issue that the OS has when trying to load applications into variable sized chunks of contiguous memory
  - Any virtual page can be loaded into any physical frame

- Virtual memory solves the memory protection issue
  - Applications issue virtual addresses, not physical addresses
  - The OS ensures that physical memory in use by application A is not pointed to by any page table entry for application B, meaning that there is no virtual address B can issue that will map to A's memory

Virtual Memory: Summary 3

- Virtual memory / address mapping has can be used in other useful ways
  - OS can help applications by making pages that hold instructions read-only
  - OS can insert a "hole" in the virtual address space between the stack and the help so that a page fault occurs if the stack gets too big, rather than just overwriting the heap
  - Programs are more portable, because they don't require the system they're running on to have any specific amount of real memory
    - A virtual address space can be larger than the amount of physical memory on the system
    - OS tries to get "hot pages" in memory, and keeps the rest on disk
    - A page fault occurs if a reference is made to a page that is currently on disk, causing it to be brought into memory
  - Virtual memory can be used to protect I/O devices, using "memory mapped I/O"