Introduction to Virtual Memory

Computer Operating Systems, Fall 2025

Instructors: Joel Ramirez

Head TAs: Maya Huizar Akash Kaukuntla
Vedansh Goenka Joy Liu

TAs:

Eric Zou Joseph Dattilo Aniket Ghorpade Shriya Sane
Zihao Zhou Eric Lee Shruti Agarwal Yemisi Jones
Connor Cummings Shreya Mukunthan Alexander Mehta Raymond Feng
Bo Sun Steven Chang Rania Souissi Rashi Agrawal
Sana Manesh

Logistical Stuff

- Recitation Today! Same time and same place. Led by Akash and Vedansh!
  - Focusing on Condition Variables, Caching, and Virtual Memory!

- PennOS Milestone 0!
  - Due tomorrow – Make sure to set up a time with TAs if you haven't.
  - Will be penalized if you reach out late. Do not procrastinate please.

Poll Everywhere

discuss

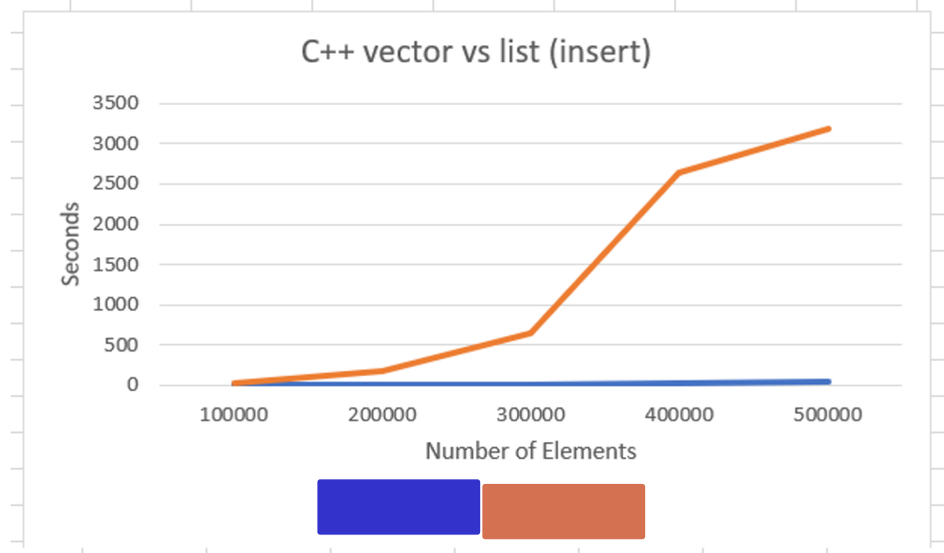
- Data Structures Review: I want to randomly generate a sequence of sorted numbers. To do this, we generate a random number and insert the number so that it remains sorted. Would a LinkedList or an ArrayList work better?
  - Assume we only use a Linear search!

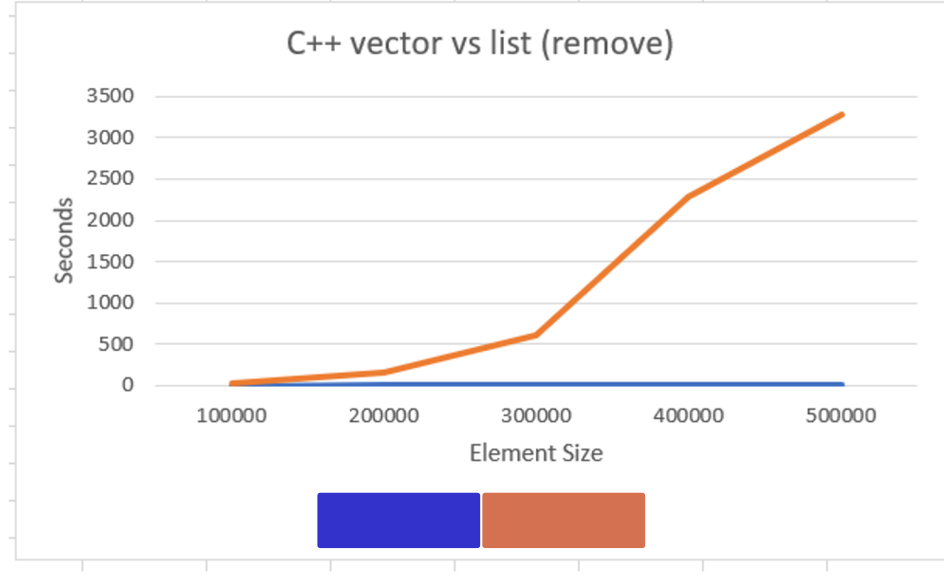
e.g. if I have sequence [5, 9, 23] and I randomly generate 12, I will insert 12 between 9 and 23

- Part 2: Let's say we take the list from part 1, randomly generate an index and remove that index from the sequence until it is empty. Would this be faster on a LinkedList or an ArrayList?
  - Assume we only use a Linear search!

Answer:

- Run using c++:
- Terminology
  - Vector == ArrayList
  - List == LinkedList
- On Element size from 100,000 -> 500,000





Back to the Poll Questions

- Data Structures Review: I want to randomly generate a sequence of sorted numbers. To do this, we generate a random number and insert the number so that it remains sorted. Would a LinkedList or an ArrayList work better?

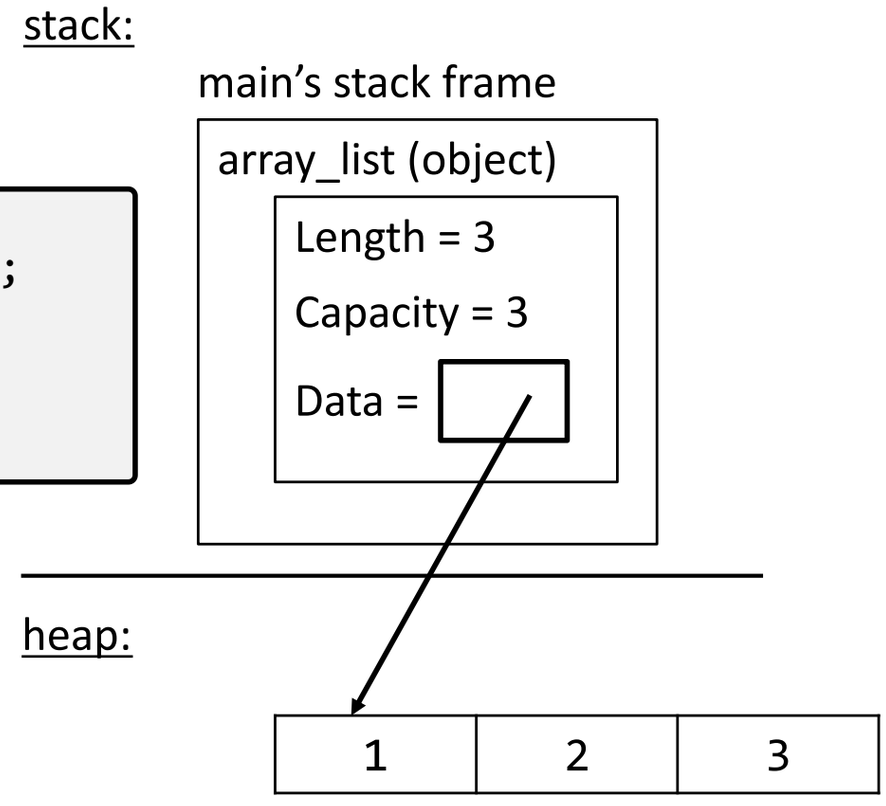
- Part 2: Let's say we take the list from part 1, randomly generate an index and remove that index from the sequence until it is empty. Would this be faster on a LinkedList or an ArrayList?

Data Structure Memory Layout

- Important to understanding the poll questions, we understand the memory layout of these data structures

- ArrayList In C++:

```
int main() {
  vector<int> array_list {1, 2, 3};
  // …
}
```



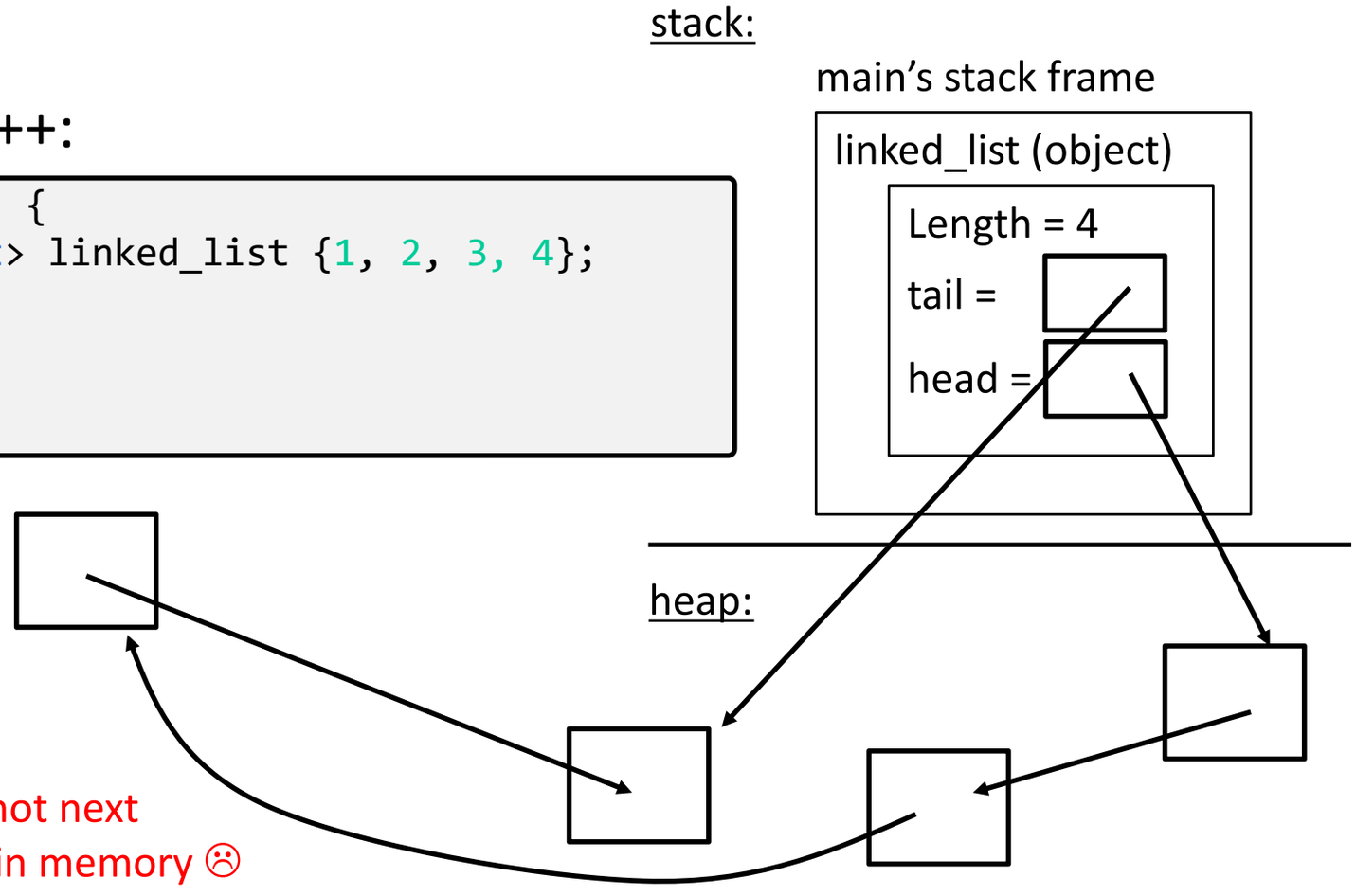
Elements are next to each other in memory ☺

Data Structure Memory Layout

- Important to understanding the poll questions, we understand the memory layout of these data structures

- LinkedList In C++:

```
int main() {
  list<int> linked_list {1, 2, 3, 4};
  // …
}
```



Poll Question: Explanation

- Vector wins in-part for a few reasons:
  - Less memory allocations
  - Integers are next to each other in memory, so they benefit from spatial complexity (and temporal complexity from being iterated through in order)

- Does this mean you should always use vectors?
  - No, there are still cases where you should use lists, but your default in C++, Rust, etc should be a vector
  - If you are doing something where performance matters, your best bet is to experiment try all options and analyze which is better.

What about other languages?

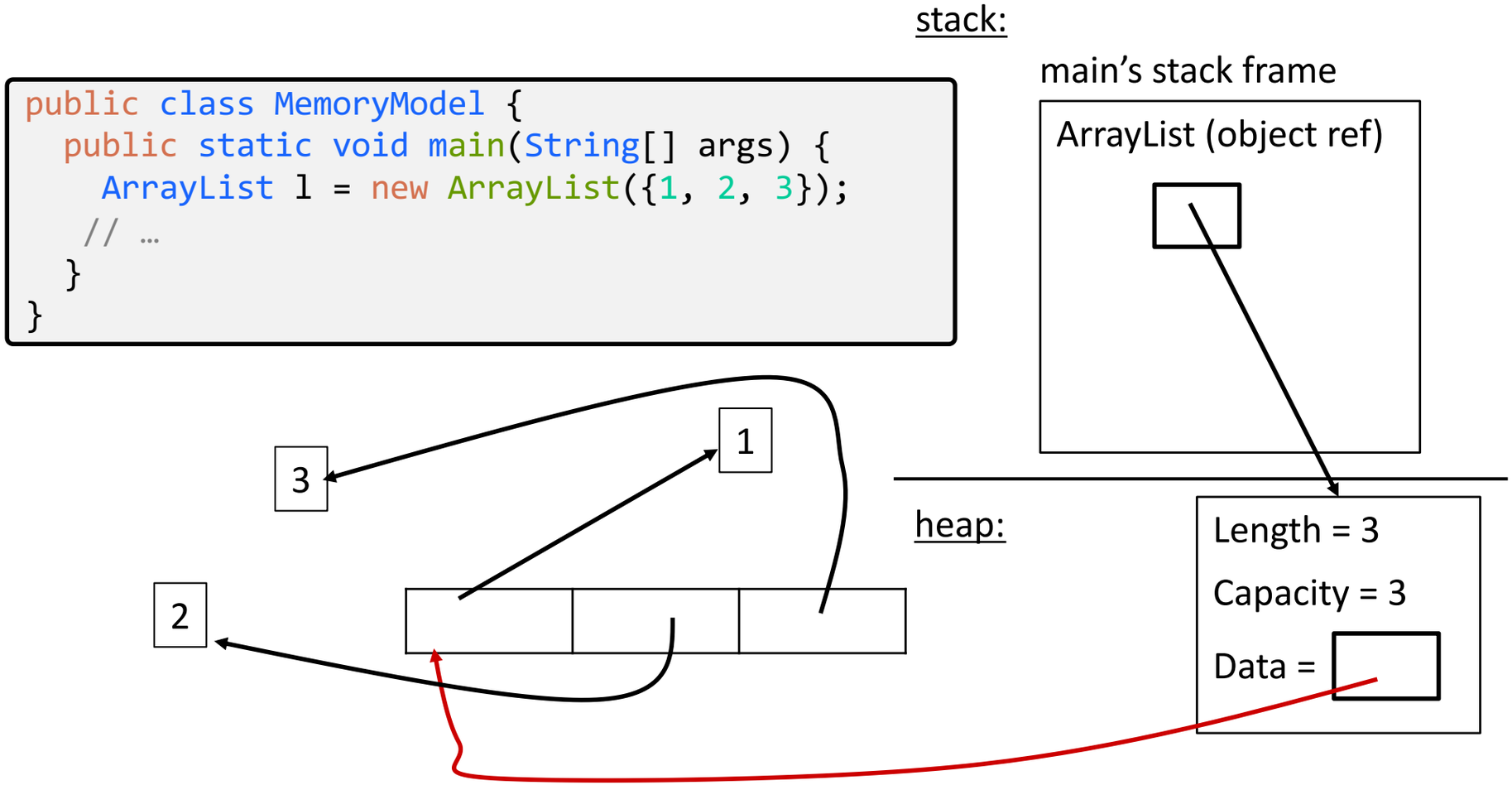
- In C++ (and C, Rust, Zig …) when you declare an object, you have an instance of that object. If you declare it as a local variable, it **exists on the stack**

- In most other languages (including Java, Python, etc.), the memory model is slightly different. Instead, **all object variables are object references, that refer to an object on the heap**

ArrayList in Java Memory Model

- In Java, the memory model is slightly different. all object variables are object references, that refer to an object on the heap

```
public class MemoryModel {
  public static void main(String[] args) {
    ArrayList l = new ArrayList({1, 2, 3});
   // …
  }
}
```



Poll Everywhere

pollev.com/cis5480

- Let's say I had a matrix (rectangular two-dimensional array) of integers, and I want the sum of all integers in it

- Would it be faster to traverse the matrix row-wise or column-wise?
  - row-wise (access all elements of the first row, then second)
  - column:-wise (access all elements of the first column, …)

1	5	8	10
11	2	6	9
14	12	3	7
0	15	13	4

Poll Everywhere

discuss

- Let's say I had a matrix (rectangular two-dimensional array) of integers, and I want the sum of all integers in it

- Would it be faster to traverse the matrix row-wise or column-wise?
  - row-wise (access all elements of the first row, then second)
  - column:-wise (access all elements of the first column, …)

1	5	8	10
11	2	6	9
14	12	3	7
0	15	13	4

Hint: Memory Representation in C & C++

1	5	8	10	11	2	6	9	14	12	3	7	0	15	13	4

Instruction Cache

- The CPU not only has to fetch data, but it also fetches instructions. There is a separate cache for this
  - which is why you may see something like `L1I` cache and `L1D` cache, for Instructions and Data respectively

- Consider the following three fake objects linked in inheritance

```
public class A {
  public void compute() {
    // …
  }
}
```

```
public class B extends A {
  public void compute() {
    // …
  }
}

public class C extends A {
  public void compute() {
    // …
  }
}
```

Instruction Cache

- Consider this code

```
public class ICacheExample {
  public static void main(String[] args) {
    ArrayList<A> l = new ArrayList<A>();
    // …
    for (A item : l) {
       item.compute();
    }
  }
}
```

```
public class A {
  public void compute() {
    // …
  }
}
```

```
public class B extends A {
  public void compute() {
    // …
  }
}

public class C extends A {
  public void compute() {
    // …
  }
}
```

- When we call item.compute that could invoke A's compute, B's compute or C's compute
- Constantly calling different functions, may not utilizes instruction cache well

Instruction Cache

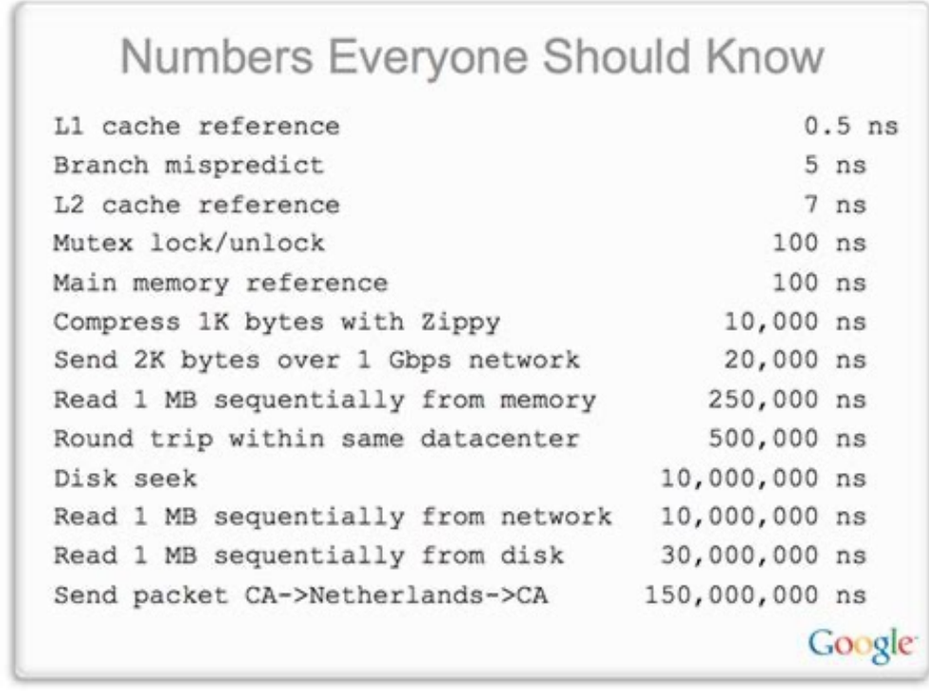
- Consider this code new code: makes it so we always do A.compute() -> B.compute() -> C.compute()
- Instruction Cache is happier with this

```
public class ICacheExample {
  public static void main(String[] args) {
    ArrayList<A> la = new ArrayList<A>();
    ArrayList<B> lb = new ArrayList<B>();
    ArrayList<C> lc = new ArrayList<C>();
    // …
    for (A item : la) {
       item.compute();
    }
    for (B item : lb) {
       item.compute();
    }
    for (C item : lc) {
       item.compute();
    }
  }
}
```

Numbers Everyone Should Know

- There is a set of numbers that called “numbers everyone you should know”

- From Jeff Dean in 2009
- Numbers are out of date but the relative orders of magnitude are about the same



- More up to date numbers: https://colin-scott.github.io/personal_website/research/interactive_latency.html

Lecture Outline

- Wrapping Up: Caching
- **Problems with Old Memory Model**
- Virtual Memory High Level
- Address Translation

Poll Everywhere

pollev.com/cis5480

- What does this print for **`x`** at all three points?
- How does the value of **`ptr`** change?

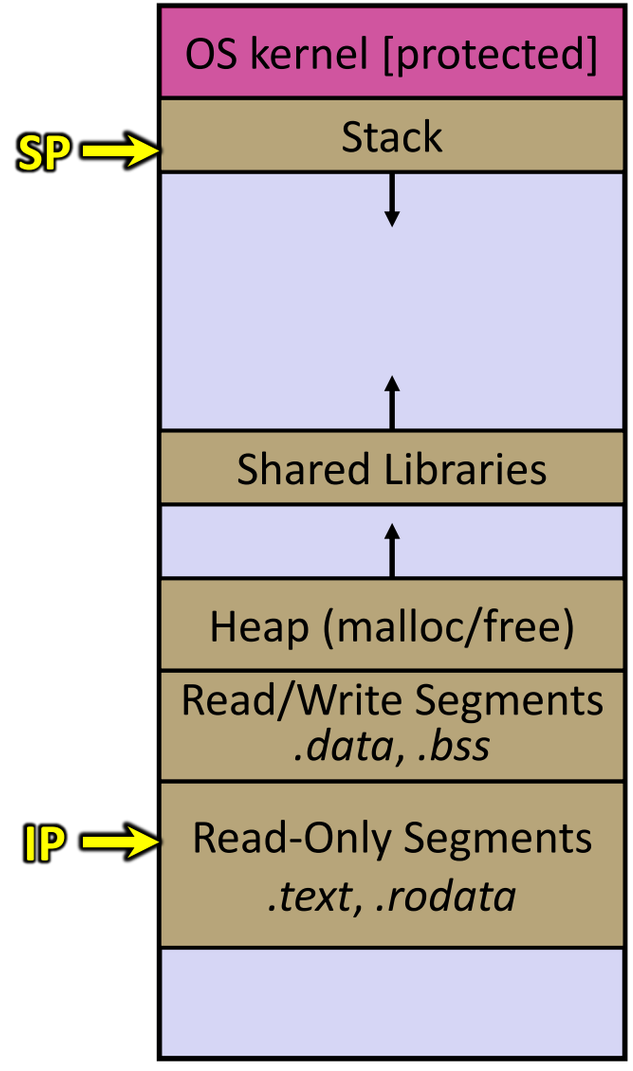
```c
int main() {
  int x = 3;
  int *ptr = &x;

  printf("[Before Fork]\t x = %d\n", x);
  printf("[Before Fork]\t ptr = %p\n", ptr);

  pid_t pid = fork();
  if (pid < 0) {
    perror("fork errored");
    return EXIT_FAILURE;
  }

  if (pid == 0) {
    x += 2;
    printf("[Child]\t\t x = %d\n", x);
    printf("[Child]\t\t ptr = %p\n", ptr);

    return EXIT_SUCCESS;
  }

  // assume no error
  waitpid(pid, NULL, 0);

  x -= 2;
  printf("[Parent]\t x = %d\n", x);
  printf("[Parent]\t ptr = %p\n", ptr);

  return EXIT_SUCCESS;
}
```

fork_addr.c

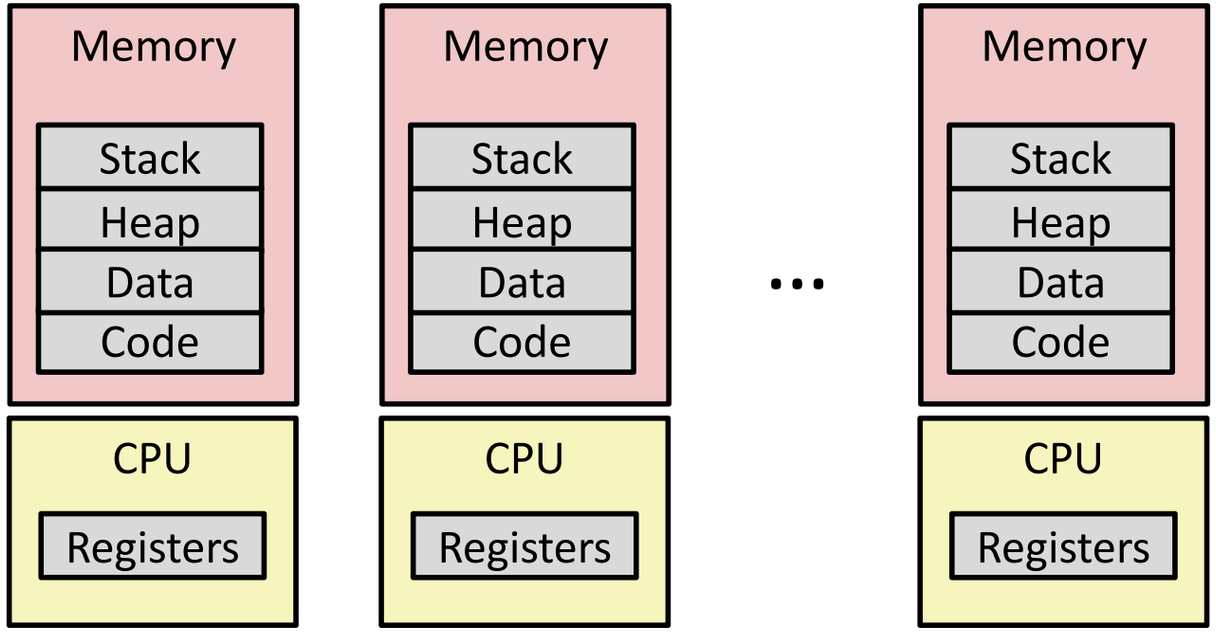
Review: Processes

- Definition: An instance of a program that is being executed (or is ready for execution)

- Consists of:
  - Memory (code, heap, stack, etc)
  - Registers used to manage execution (stack pointer, program counter, ...)
  - Other resources

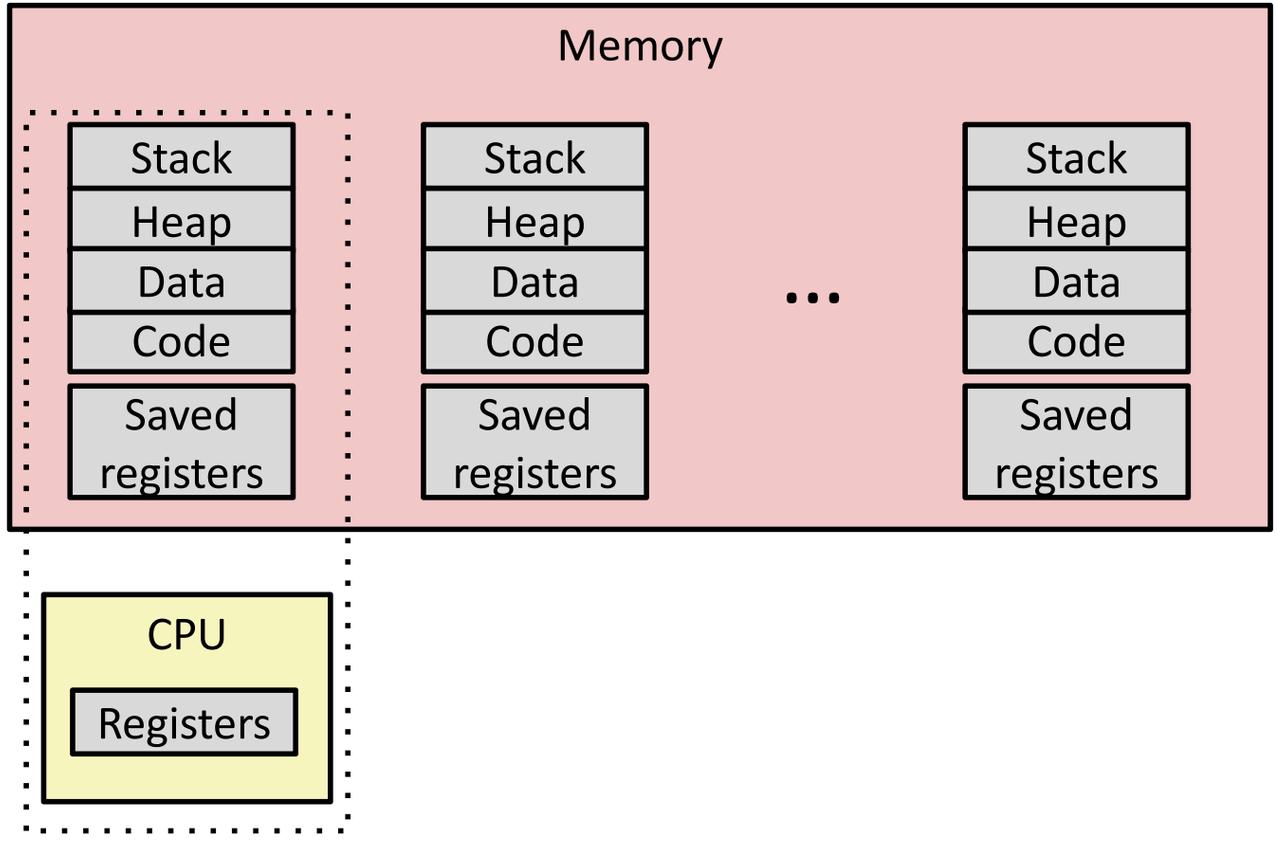


Multiprocessing: The Illusion



- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

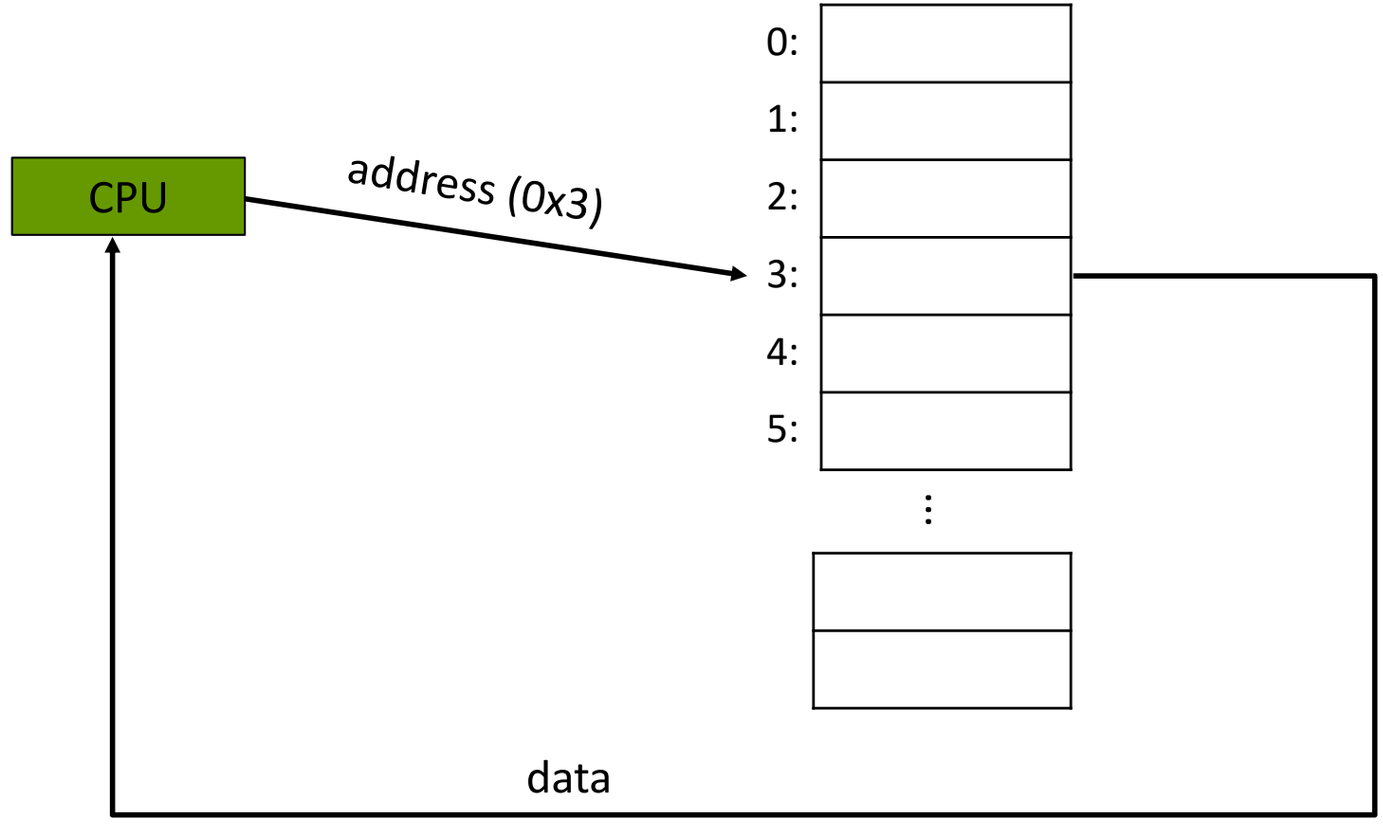
Multiprocessing: The (Traditional) Reality



- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course (now!))
  - Register values for non-executing processes saved in memory

Memory As We Know It

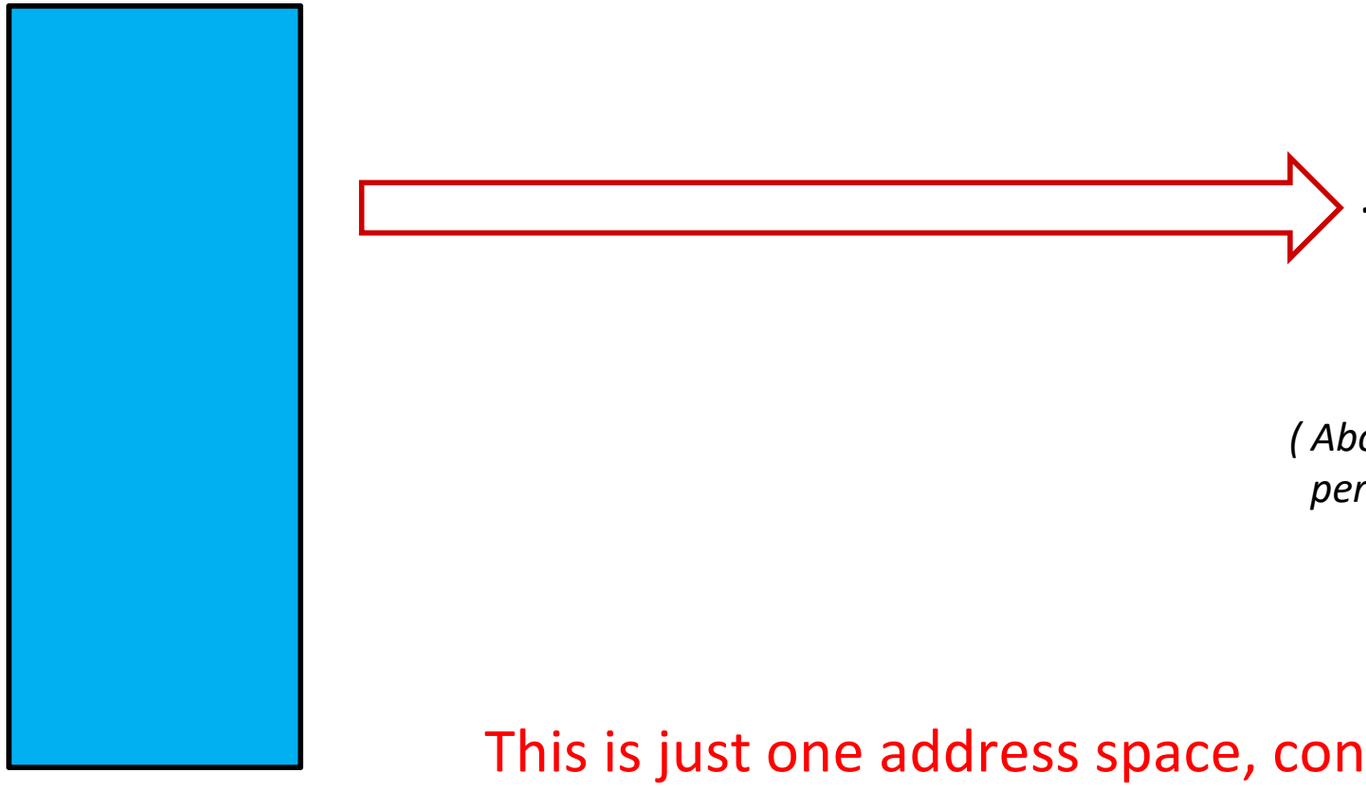
- The CPU directly uses an address to access a location in memory



Problem 1: How does everything fit?

On a 64-bit machine, there are ~2^{64} ***Addressable bytes***, which is: 18,446,744,073,709,551,616 Bytes (1.844×10^{19})

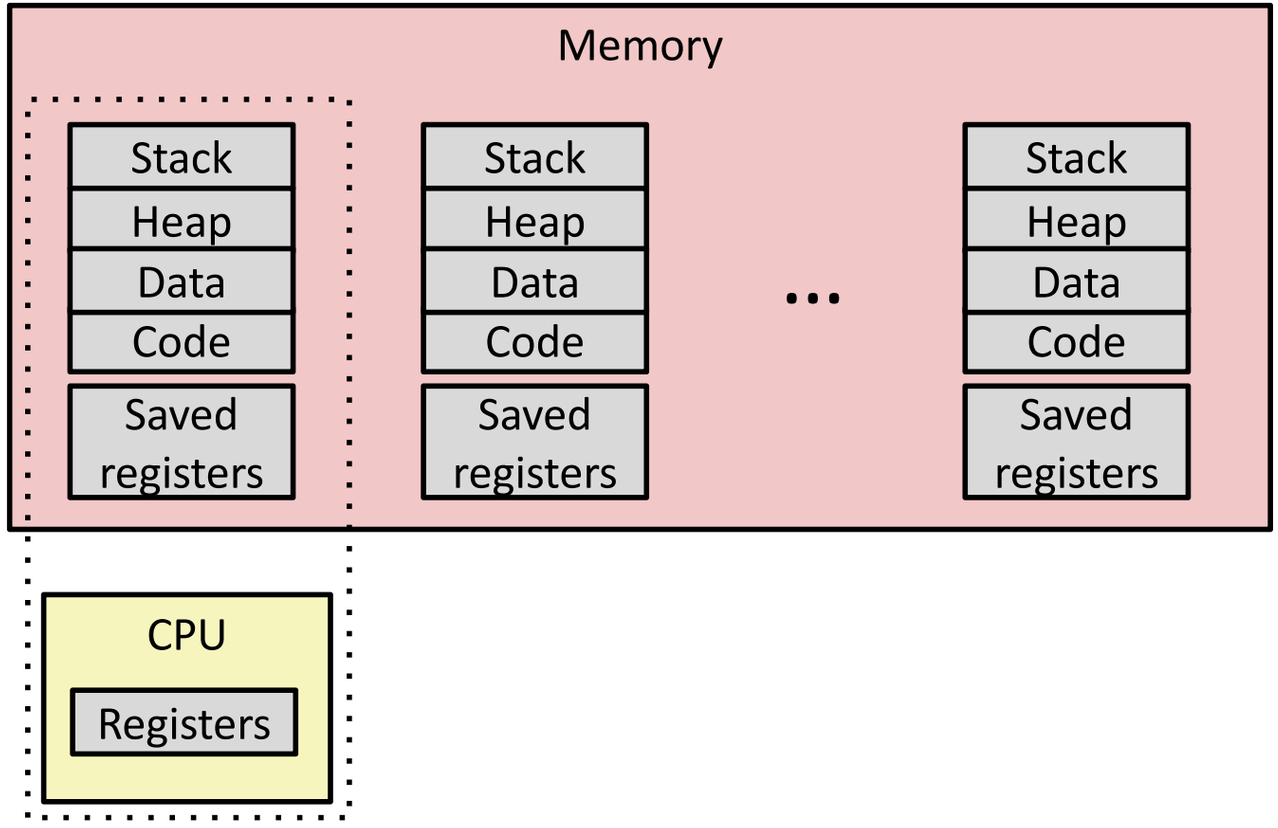
Laptops usually have around 8GB which is 8,589,934,592 Bytes (8.589×10^{9})



(About to scale; physical memory is smaller than the period at the end of the sentence compared to the virtual address space.)

This is just one address space, consider multiple processes…

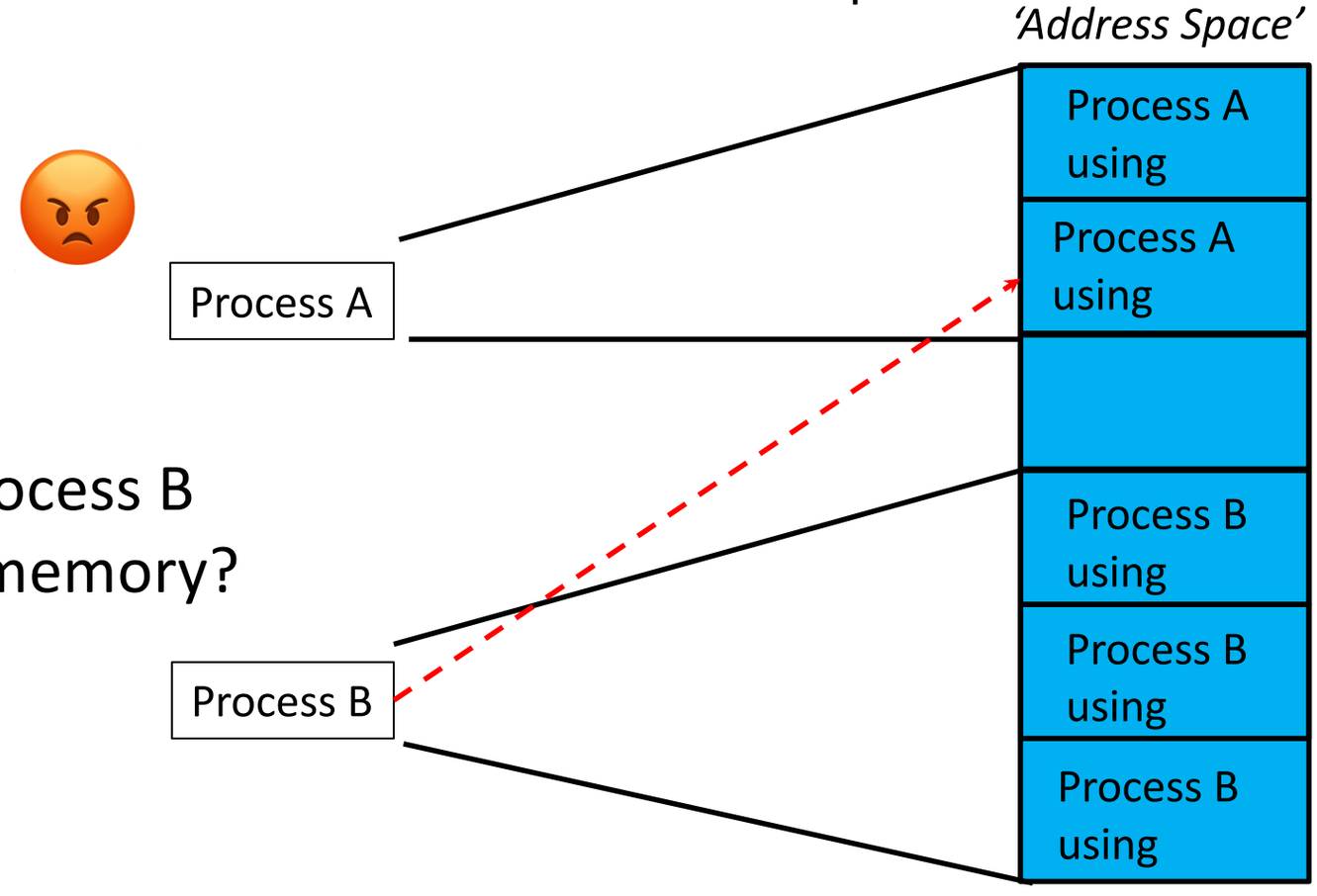
Problem 2: Sharing Memory



- How do we enforce process isolation?
  - Could one process just calculate an address into another process?

Problem 2: Sharing Memory

- How do we enforce process isolation?
  - Could one process just calculate an address into another process?

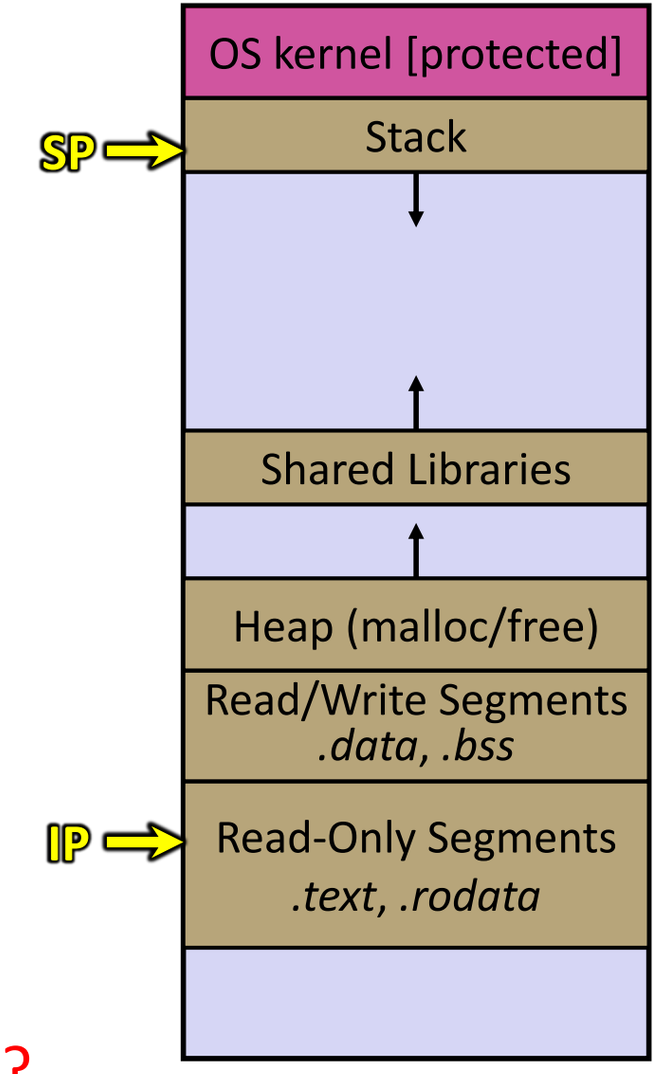


- What is stopping process B from accessing A's memory?

Problem 3: How do we segment things

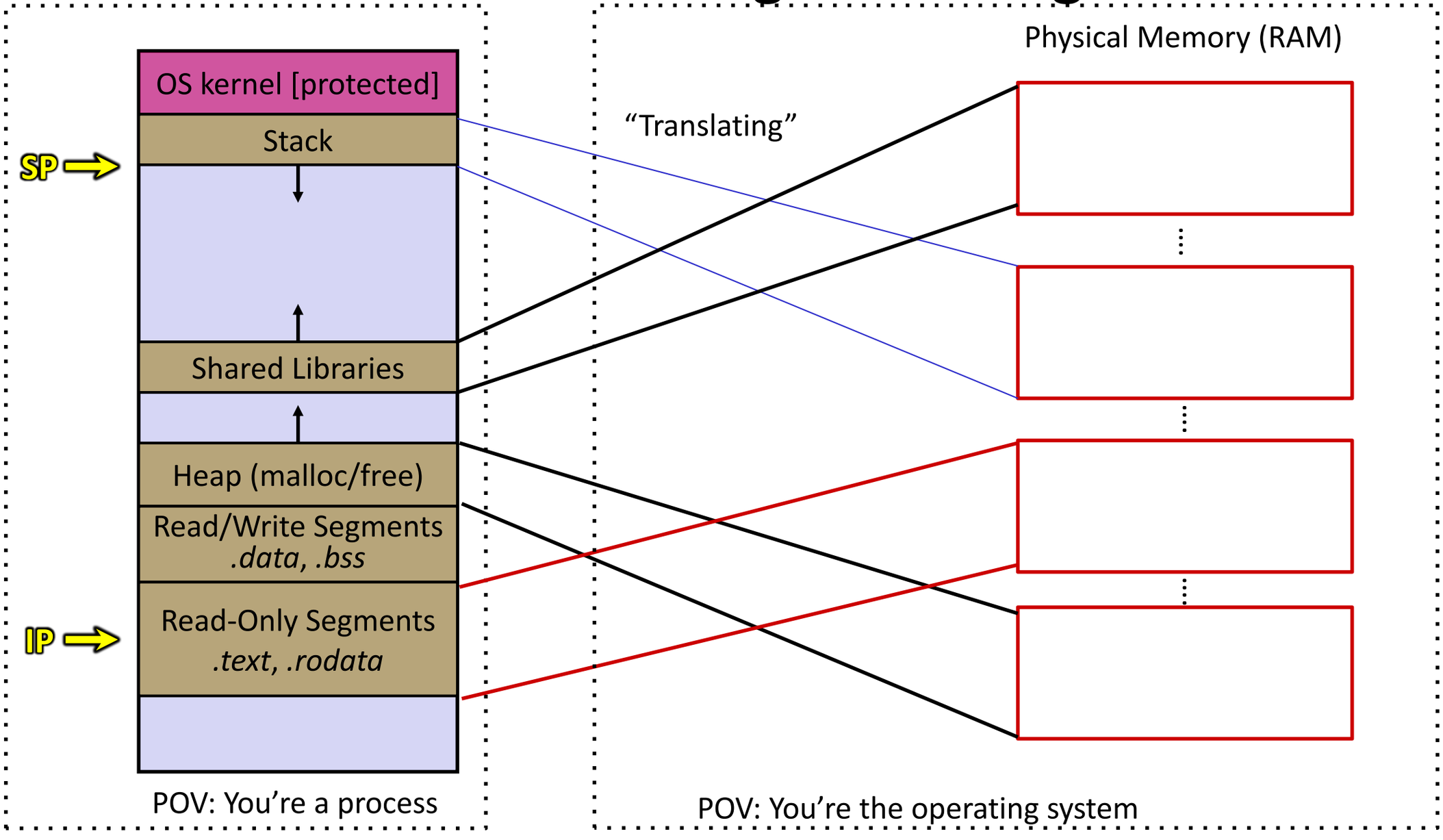
- A process' address space contains many different "segments" that have specific functionality.

- Problem: How do we keep track of the location and permissions (Read/Write) each segment may have?
  - (e.g., that Read-Only data can't be written to)



The real question is *who* is keeping track of this?

Problem 3: How do we segment things?



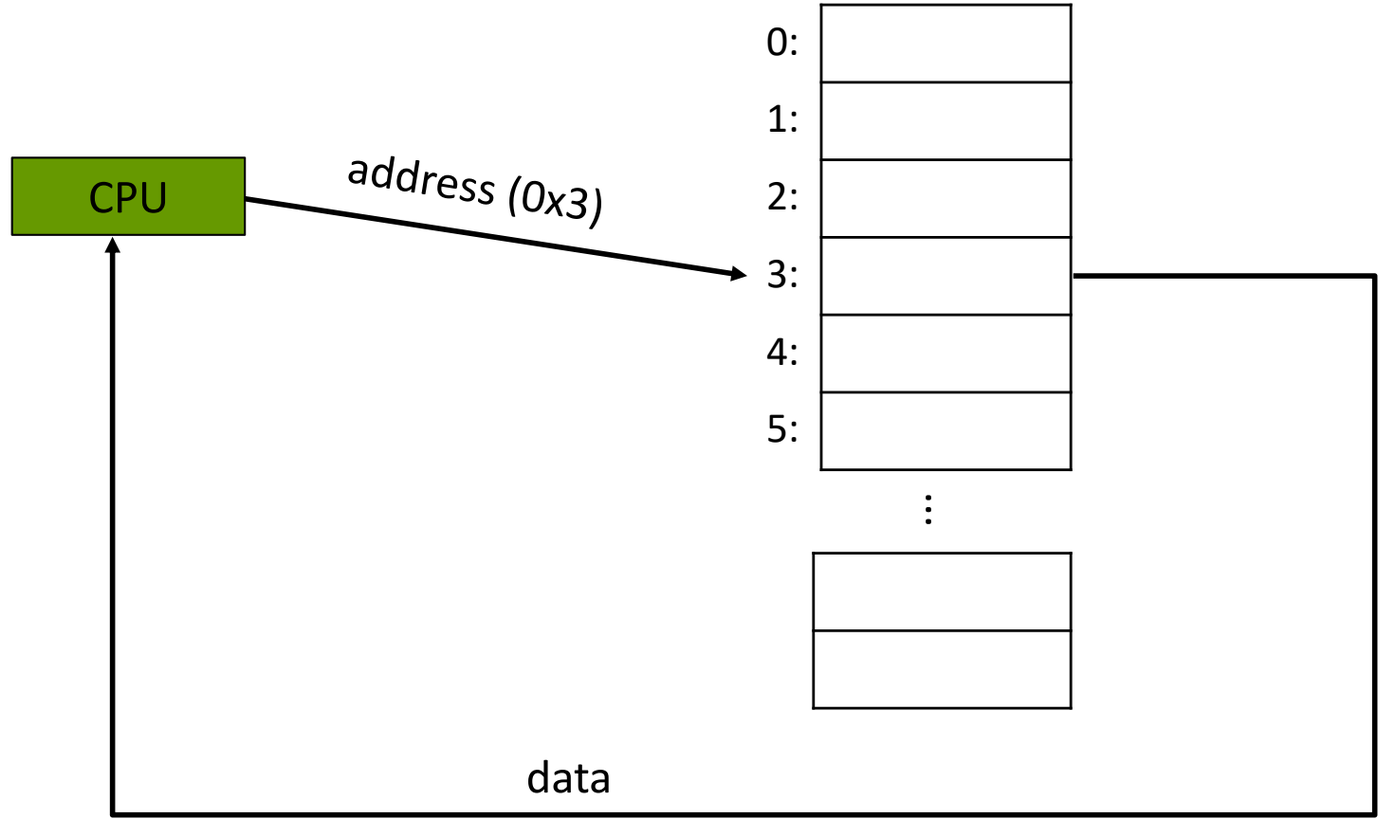
Note: some mappings are missing, not enough space.

Lecture Outline

- Wrapping Up: Caching
- Problems with Old Memory Model
- **Virtual Memory High Level**
- Address Translation

This Is Not What Happens

- The CPU directly uses an address to access a location in memory



Indirection

- "Any problem in computer science can be solved by adding another level of indirection."
  - David wheeler, inventor of the subroutine (e.g. functions)

- The ability to indirectly reference something using a name, reference or container instead of the value itself. A flexible mapping between a name and a thing allows changing the thing without notifying holders of the name.
  - May add some work to use indirection (understatement of the year)
  - Example: If the physical location of a variable changes, you don't need to tell the process that…

- Idea: instead of directly referring to physical memory, add a level of indirection

Idea:

- We don’t need all processes to have their data in physical memory, **just the ones that are currently running**

- For the process’ that are currently running: we don’t need all their data to be in physical memory, **just the parts that are currently being used**

- Data that isn’t currently stored in physical memory, can be stored elsewhere (disk).
  - Disk is "permanent storage" usually used for the file system
  - Disk has a longer access time than physical memory (RAM)

Pages

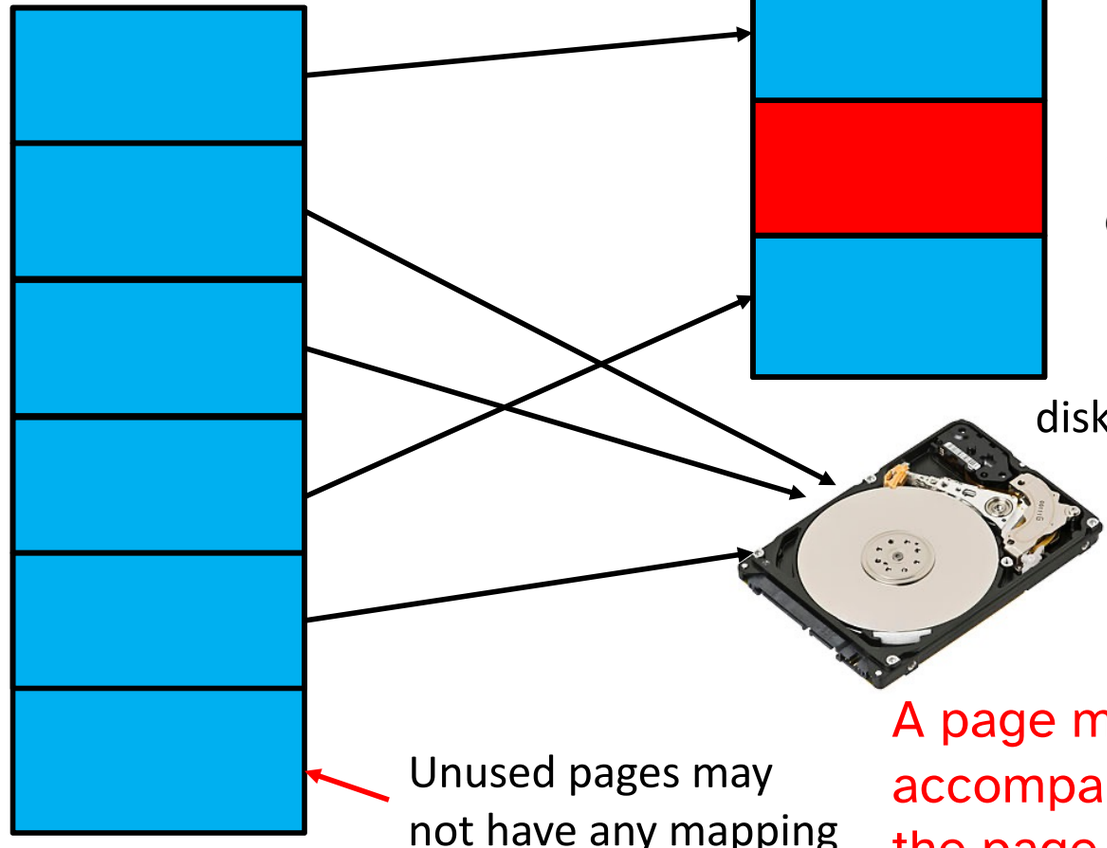
Pages are of fixed size ~4KiB
4KiB -> (4 * 1024 = 4096 bytes.)

- Memory can be split up into units called “pages”

(what the process thinks it has)

Address space

Physical memory



Pages currently in use are stored in physical memory (RAM)

← Ram may contain pages from other active processes

Pages in physical memory are called “Page frames”

disk

Pages not currently in use (but were used in the past) are stored on disk

Unused pages may not have any mapping

A page may not have an accompanying page frame until the page is used

Definitions

Sometimes called “virtual memory”
or the “virtual address space”

- Addressable Memory: the total amount of memory that can be theoretically be accessed based on:
  - number of addresses (“address space”)
  - bytes per address (“addressability”)

IT MAY OR MAY NOT EXIST ON HARDWARE
(like if that memory is never used)
(As in the references might not even be “valid”)

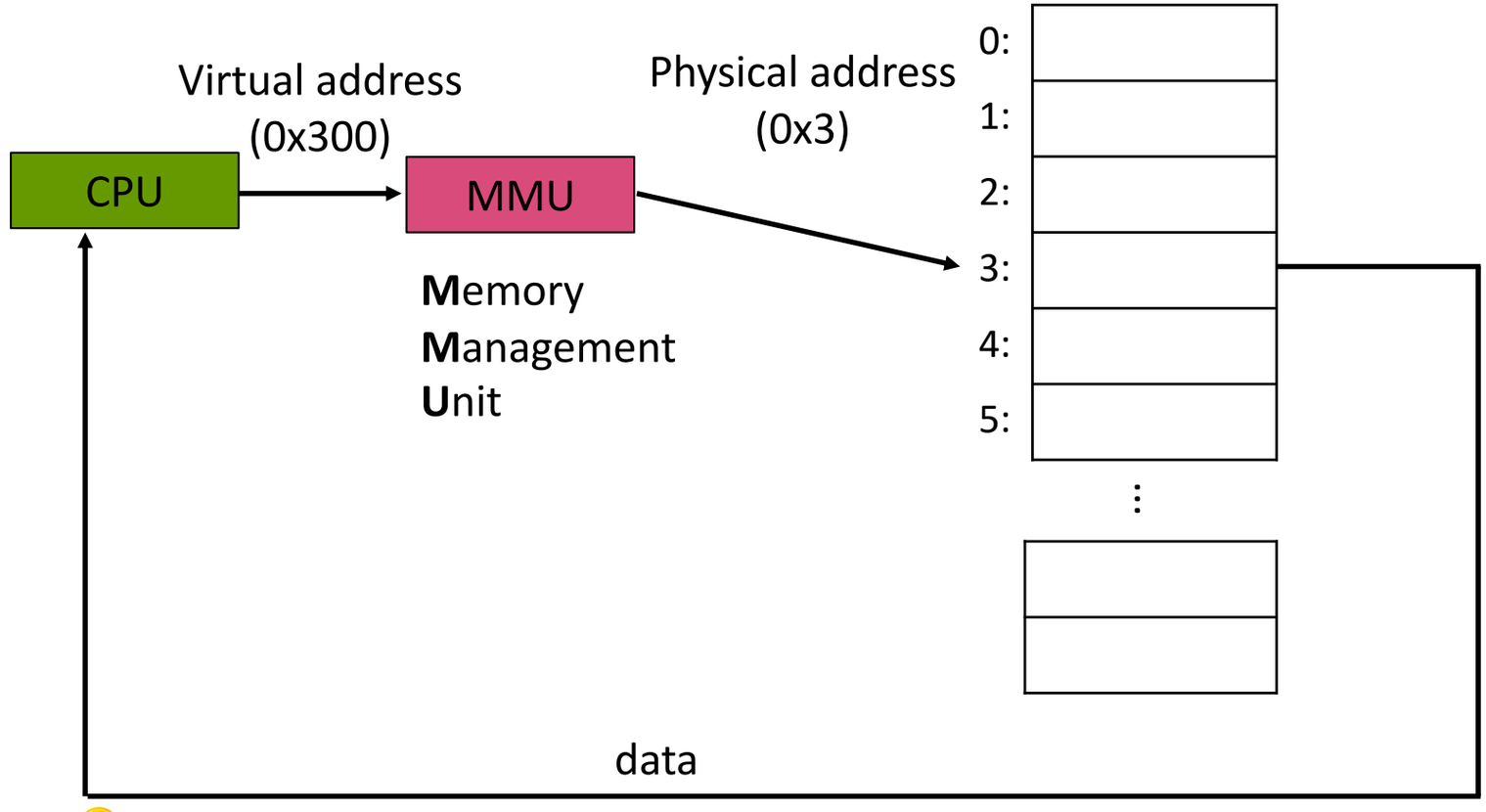
- Physical Memory: the total amount of memory that is physically available on the computer

Physical memory holds a subset of the addressable memory being used

- Virtual Memory: An abstraction technique for making memory look larger than it is and hides many details from the programs.

Virtual Address Translation

- Programs don't know about physical addresses; virtual addresses are translated into them by the MMU



THIS SLIDE IS 🔑 TO THE WHOLE IDEA

Page Tables

More details about translation later

- Virtual addresses can be converted into physical addresses via a page table.

- There is one page table per processes, managed by the MMU

Virtual page #	Valid	Physical Page Number
0	0	null //page hasn't been used yet
1	1	0
2	1	1
3	0	disk

"A page has not been allocated for this page".

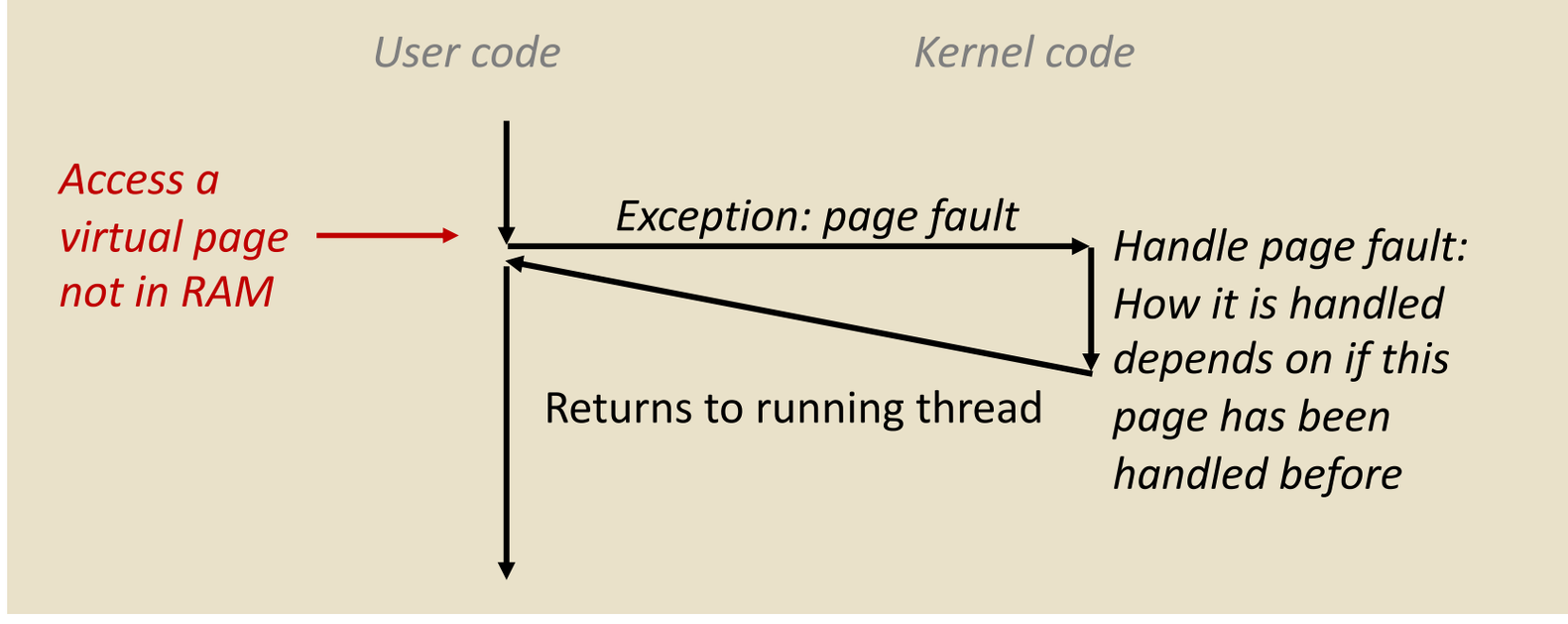
"A page is allocated but not in memory".

Valid determines if the page is in physical memory

If a page is on disk, MMU will fetch it

Page Fault Exception

- An *Exception* is a transfer of control to the OS *kernel* in response to some ***synchronous event*** *(directly caused by what was just executed)*
- In this case, writing to a memory location that is not in physical memory currently



Problem: Paging Replacement

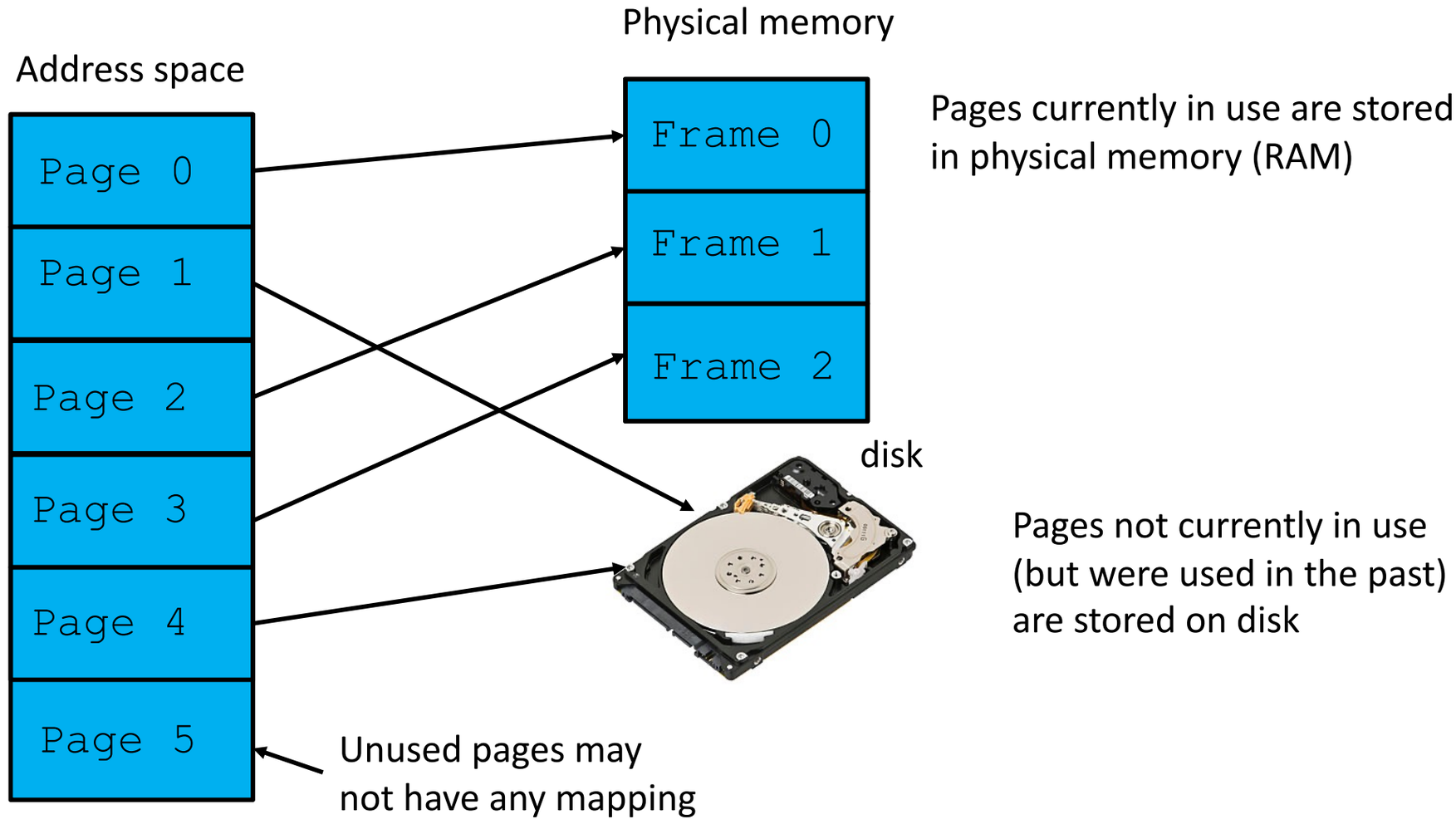
More details about page replacement later

- We don't have space to store all active pages in physical memory.
- If physical memory is full and we need to load in a page, then how do we choose a page in physical memory to store on disk in the **swap file**
- If we need to load in a page from disk, how do we decide which page in physical memory to "evict"
- Goal: Minimize the number of times we have to go to disk. It takes a while to go to disk.

Poll Everywhere

pollev.com/cis5480

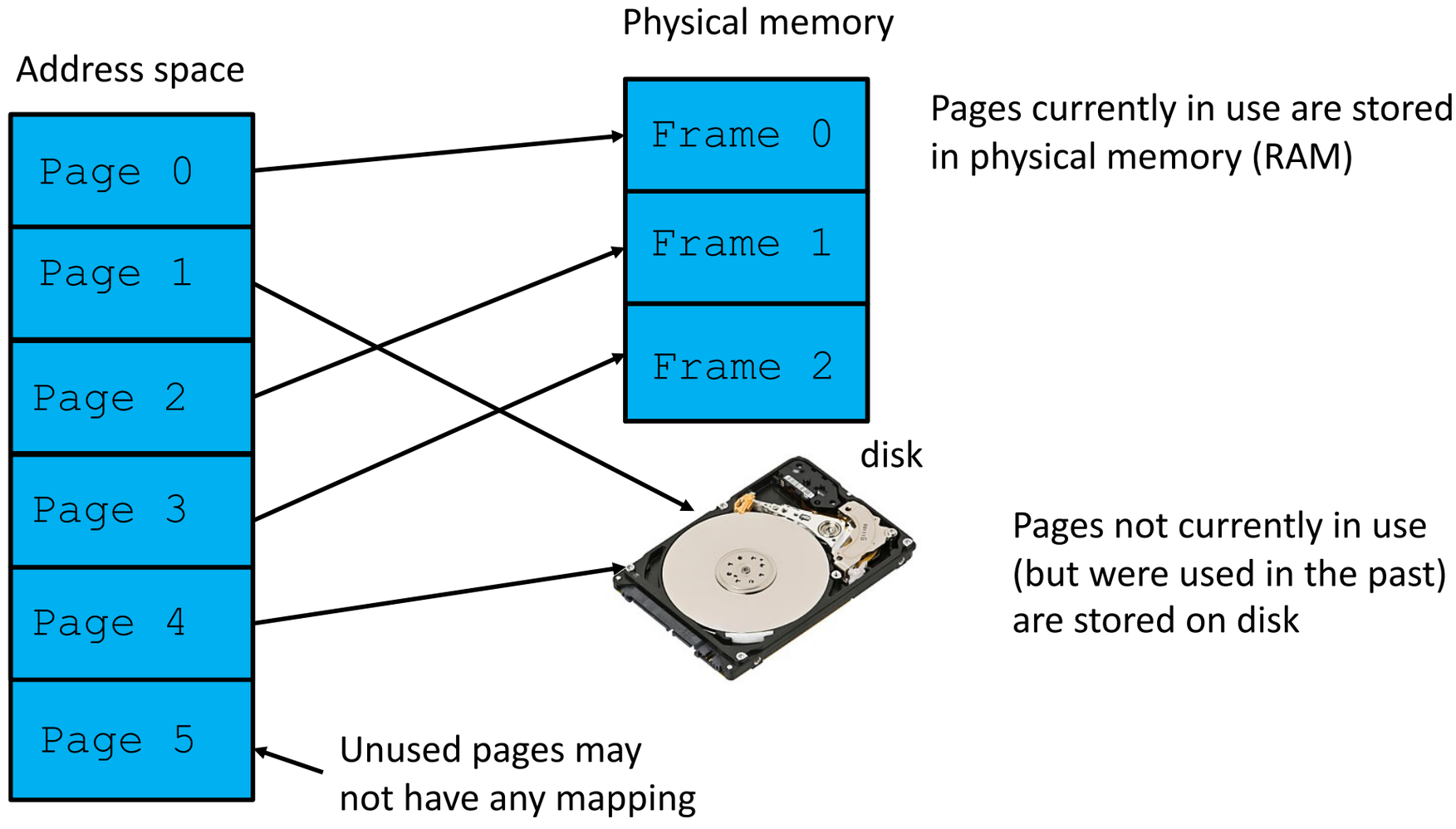
- What happens if this process tries to access an address in page 3?



Poll Everywhere

pollev.com/cis5480

- What happens if we need to load in page 1 and physical memory is full?



Lecture Outline

- Problems with old memory model
- Virtual Memory High Level
- **Address Translation**

Aside: Bits

- We represent data on the computer in binary representation (base 2)
- A bit is a single “digit” in a binary representation.
- A bit is either a 0 or a 1

- In decimal -> 243
- In binary -> 0b11110011

Hexadecimal

- Base 16 representation of numbers
- Allows us to represent binary with fewer characters
  - 0b11110011 == 0xF3
    ^ **b**inary ^ he**x**

Decimal	Binary	Hex
0	0000	0x0
1	0001	0x1
2	0010	0x2
3	0011	0x3
4	0100	0x4
5	0101	0x5
6	0110	0x6
7	0111	0x7
8	1000	0x8
9	1001	0x9
10	1010	0xA
11	1011	0xB
12	1100	0xC
13	1101	0xD
14	1110	0xE
15	1111	0xF

Poll Everywhere

pollev.com/cis5480

- A page is typically 4 KiB -> 2^{12} -> 4096 bytes

- If physical memory is 32 KiB, how many page frames are there?

A. 5 B. 4 C. 32 D. 8 E. We're lost…

- If ***addressable memory*** for a single process consists of 64 KiB bytes, how many pages are there for one process?

A. 64 B. 16 C. 20 D. 6 E. We're lost…

- If there is one page table per process, how many entries should there be in a single page table?

A. 6 B. 8 C. 16 D. 5 E. None of These…

Addresses

- Virtual Address:
  - Used to refer to a location in a virtual address space.
  - Generated by the CPU and used by our programs

- Physical Address
  - Refers to a location on physical memory
  - Virtual addresses are converted to physical addresses

Page Offset

- This idea of Virtual Memory abstracts things on the level of Pages
  - (4096 bytes == 2^{12} bytes)

- On almost every machine, memory is ***byte-addressable*** meaning that each byte in memory has its own address

- How many distinct addresses can correspond to the ***same page***?

4096 addresses to a single page

- At a minimum, how many bits are dedicated to calculating the location (offset) of an address within a page?

12 bits

64 bits: '0000000000000000000000000000000000000000000000000000000000000000'

Poll Everywhere

pollev.com/cis5480

- If there are 16 pages (virtual), how many bits would you need to represent the number of pages?

- If there are 8 pages frames (physical), how many bits would we need to represent the number of page frames?

	Page bits	Frame bits
A.	4	2
B.	4	3
C.	3	3
D.	5	3
E.	We're lost…	

High Level: Steps For Translation

- Derive the virtual page number from a virtual address
- Look up the virtual page number in the page table
  - Handle the case where the virtual page doesn't correspond to a physical page frame
- Construct the physical address

Address Translation: Virtual Page Number

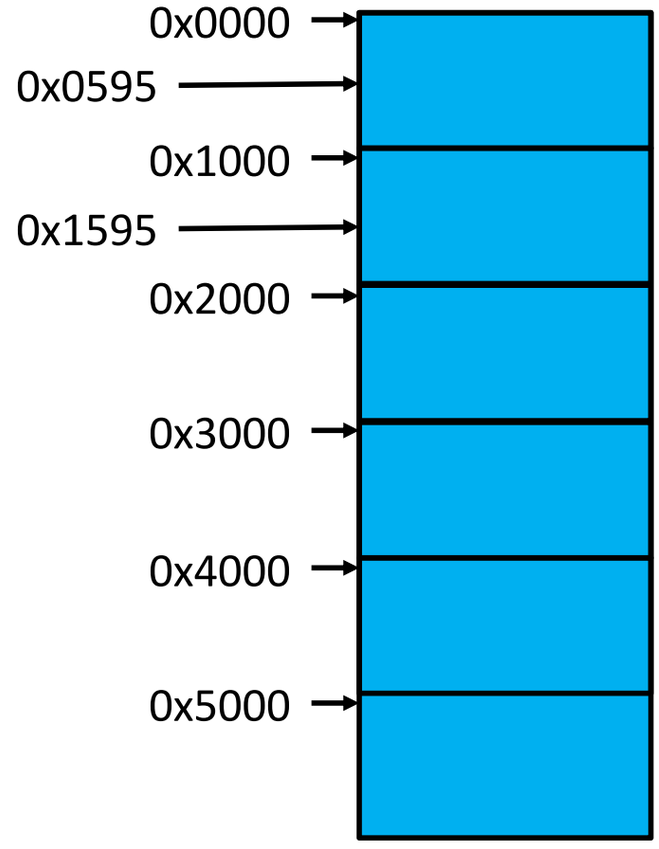
- A virtual address is composed of two parts relevant for translating:
  - Virtual Page Number length = bits to represent number of pages
  - Page offset length = bits to represent number of bytes in a page

- The virtual page number determines which page we want to access

- The page offset determines which location within a page we want to access.
  - Remember that a page is many bytes (~4KiB -> 4096 bytes)

Virtual Address High Level View

- High level view:
  - Each page starts at a multiple of 4096 (0X1000)
  - If we take an address and add 4096 (0x1000) we get the same offset but into the next page



Address Translation: Virtual Page Number

- A virtual address is composed of two parts relevant for translating:
  - Virtual Page Number length = bits to represent number of pages
  - Page offset length = bits to represent number of bytes in a page

Virtual Page Number	Page Offset

pollev.com/cis5480

- Example address: 0x1234
  - What is the page number?
  - What is the offset?
  - Reminder: there are 16 virtual pages, and a page is 4096 bytes

Address Translation: Lookup & Combining

- Once we have the page number, we can look up in our page table to find the corresponding physical page number.
  - For now, we will assume there is an associate page frame

Virtual page #	Valid	Physical Page Number
0x0	0	null
0x1	1	0x5
…	…	…

- With the physical page number, combine it with the page offset to get the physical address

  - In our example, with 0x1234, our physical address is 0x5234

Translation Done!

Page Faults

- What if we accessed a page whose page frame was not in physical memory?

Virtual page #	Valid	Physical Page Number
0x0	0	null
0x1	1	0x0
0x2	1	0x5
0x3	0	Disk
…	…	…

- In this example, Virtual page 0x3

Page Faults

Virtual page #	Valid	Physical Page Number
0x0	0	null
0x1	1	0x0
0x2	1	0x5
0x3	0	Disk
…	…	…

- In this example, Virtual page 0x3, whose frame is on disk (page 0x3 handled before, but was evicted at some point)
  - MMU fetches the page from disk
  - Evicts an old page from physical memory if necessary
    - Uses LRU or some page replacement algorithm
    - Writes the contents of the evicted page back to disk
  - Store the previously fetched page to physical memory

Page Faults

Virtual page #	Valid	Physical Page Number
0x0	0	null
0x1	1	0x0
0x2	1	0x5
0x3	0	Disk
…	…	…

- In this example, Virtual page 0x0, which has never been accessed before
  - Evict an old page if necessary (A page that isn't needed)
  - Claim an empty frame and use it as the frame for our virtual page

That's all for now!

Poll Everywhere Solutions

Poll Everywhere

pollev.com/cis5480

- What does this print for **`x`** at all three points?
- How does the value of **`ptr`** change?

The value of ptr stays constant showing that the virtual address is identical for both the parent and child!

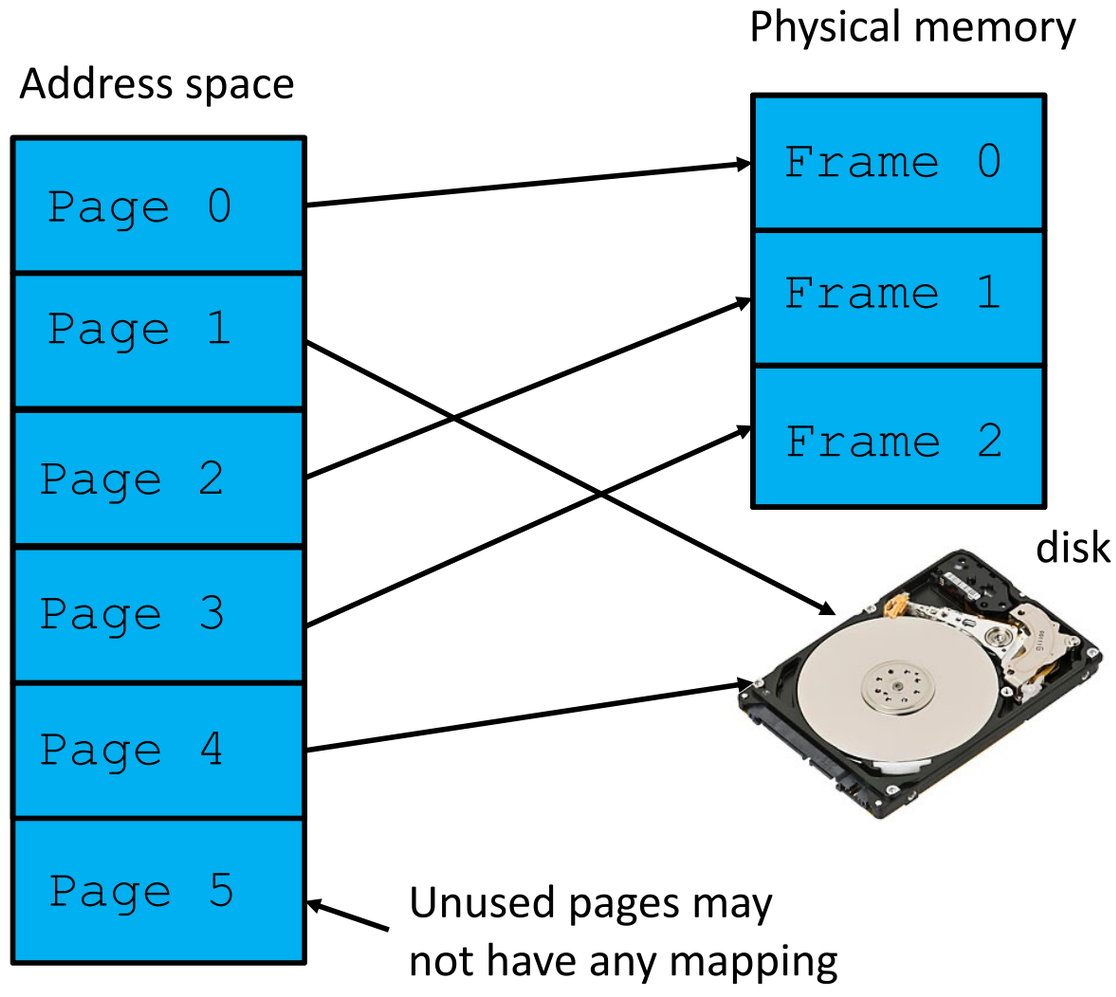
```c
int main() {
  int x = 3;
  int *ptr = &x;

  printf("[Before Fork]\t x = %d\n", x);
  printf("[Before Fork]\t ptr = %p\n", ptr);

  pid_t pid = fork();
  if (pid < 0) {
    perror("fork errored");
    return EXIT_FAILURE;
  }

  if (pid == 0) {
    x += 2;
    printf("[Child]\t\t x = %d\n", x);
    printf("[Child]\t\t ptr = %p\n", ptr);

    return EXIT_SUCCESS;
  }

  // assume no error
  waitpid(pid, NULL, 0);

  x -= 2;
  printf("[Parent]\t x = %d\n", x);
  printf("[Parent]\t ptr = %p\n", ptr);

  return EXIT_SUCCESS;
}
```

fork_addr.c

Poll Everywhere

pollev.com/cis5480

- What happens if this process tries to access an address in page 3?



Pages currently in use are stored in physical memory (RAM)

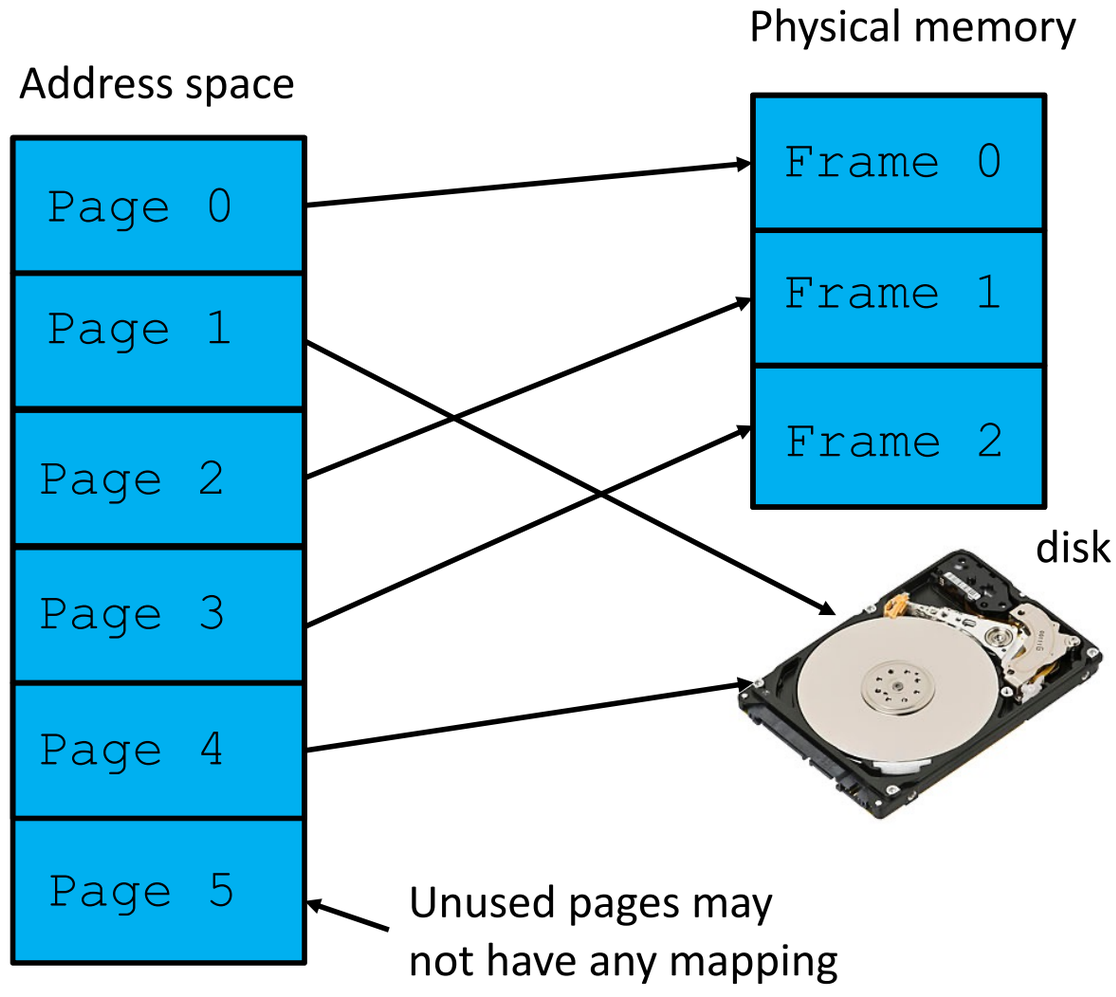
The MMU accesses the corresponding frame (frame 2)

Pages not currently in use (but were used in the past) are stored on disk

Poll Everywhere

pollev.com/cis5480

- What happens if we need to load in page 1 and physical memory is full?



Pages currently in use are stored in physical memory (RAM)

We get a page fault, the OS evicts a page from a frame, loads in new page into that frame

Pages not currently in use (but were used in the past) are stored on disk

Poll Everywhere

pollev.com/cis5480

- A page is typically 4 KiB -> 2^{12} -> 4096 bytes

- If physical memory is 32 KiB, how many page frames are there?

32 KiB / 4 KiB = 8 frames

A. 5 B. 4 C. 32 D. 8 E. We're lost…

- If ***addressable memory*** for a single process consists of 64 KiB bytes, how many pages are there for one process?

64 KiB / 4 KiB = 16 pages

A. 64 B. 16 C. 20 D. 6 E. We're lost…

- If there is one page table per process, how many entries should there be in a single page table?

A. 6 B. 8 C. 16 D. 5 E. None of These

One entry per page

Poll Everywhere

pollev.com/cis5480

- If there are 16 pages (virtual), how many bits would you need to represent the number of pages?

  num_bits = $\log_2(16) = 4$

  or

  $16 = 2^4$, so 4

- If there are 8 pages frames (physical), how many bits would we need to represent the number of page frames?

  num_bits = $\log_2(8) = 3$

  or

  $8 = 2^3$, so 3

	Page bits	Frame bits
A.	4	2
B.	4	3
C.	3	3
D.	5	3
E.	We're lost…	

Address Translation: Virtual Page Number

- A virtual address is composed of two parts relevant for translating:
  - Virtual Page Number length = bits to represent number of pages
  - Page offset length = bits to represent number of bytes in a page

Virtual Page Number	Page Offset

pollev.com/cis5480

- Example address: 0x1234
  - What is the page number?
  - What is the offset?
  - Reminder: there are 16 virtual pages, and a page is 4096 bytes

```
0001    0010    0011    0100

0001    ->      0x1

0010 0011 0100   ->    0x234
```