Virtual Memory

Computer Systems Programming, Spring 2024

Instructor: Travis McGaha

TAs:

Ash Fujiyama
CV Kunjeti
Felix Sun
Heyi Liu
Kevin Bernat
Lang Qin
Sean Chuang
Serena Chen
Yuna Shao

Poll Everywhere

pollev.com/tqm

- Any questions, comments or concerns so far about **anything?**

Upcoming Due Dates

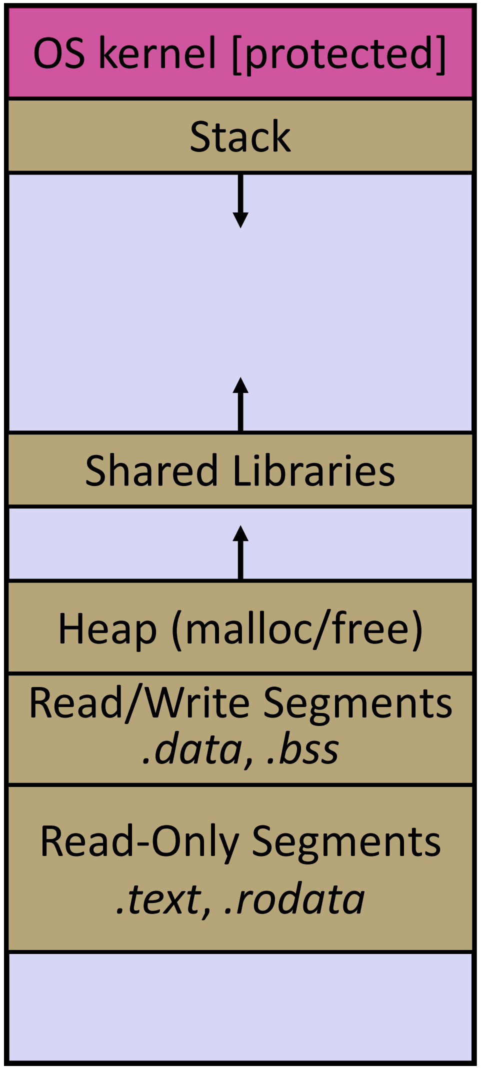
- HW2 (Threads)
  - Due a week from Thursday
- Midterm
  - Exams still being graded
  - A few makeups still happening

Lecture Outline

- **Pointers & Old Memory Model**
- Problems with old memory model
- Virtual Memory High Level
- Page Replacement

Memory

- Where all data, code, etc are stored for a program
- Broken up into several segments:
  - The stack
  - The heap
  - The kernel
  - Etc.
- Each "unit" of memory has an address



Memory as a giant array

- In CIT 5930 we introduced memory as a giant array of bytes, with each byte having its own address:

- Our variables live in memory

```
int main(int argc, char* argv[]) {
  char a = 'a';
  char b = 'b';
  return 0;
}
```

0x0	0x1	0x2

…

0x55	0x56	0x57	0x58	0x59	0x5A	0x5B	0x5C	0x5D	0x
		'a'	'b'						

Pointers

POINTERS ARE EXTREMELY IMPORTANT IN C (and C++ to a lesser extent)

- Variables that store addresses
  - It stores the address to somewhere in memory
  - Must specify a type so the data at that address can be interpreted

- Generic definition: `type* name;` or `type *name;` (equivalent)
  - Example: `int *ptr;`
    - Declares a variable that can contain an address
    - Trying to access that data at that address will treat the data there as an int

Pointer Operators

- *Dereference* a pointer using the unary `*` operator
  - Access the memory referred to by a pointer
  - Can be used to read or write the memory at the address
  - Example:

```
int *ptr = ...; // Assume initialized
int a = *ptr; // read the value
*ptr = a + 2; // write the value
```

- Get the address of a variable with `&`
  - `&foo` gets the address of foo in memory
  - Example:

```
int a = 595;
int *ptr = &a;
*ptr = 2; // 'a' now holds 2
```

Pointers as References

- The exact value stored in a pointer almost never matters, we treat them more like references
- In this class we will never hardcode in an address into a pointer. We will never do something like :

```
int *ptr = 0x7ffff5194;
```

  - Read as: "`ptr` contains the address 0x7ffff5194"
  - *with the exception of NULL
- Instead, we write code that is more often like:

```
int example = 5;
int *ptr = &a;
```

  - Read as: "`ptr` refers to the integer `example`"
  - Or "`ptr` contains the address of the integer `example`"

NULL

- `NULL` is a memory location that is guaranteed to be invalid
  - In C on Linux, `NULL` is `0x0` and an attempt to dereference `NULL` *causes a segmentation fault*
- Useful as an indicator of an uninitialized (or currently unused) pointer or allocation error
  - It's better to cause a segfault than to allow the corruption of memory!

```
int main(int argc, char** argv) {
  int* p = NULL;
  *p = 1;  // causes a segmentation fault
  return EXIT_SUCCESS;
}
```

Pointer Example

```
int main(int argc, char** argv) {
  int a, b, c;
  int* ptr;    // ptr is a pointer to an int

  a = 5;
  b = 3;
  ptr = &a;

  *ptr = 7;
  c = a + b;

  return 0;
}
```

Initial values are garbage

0x2001	**a**	--
0x2002	**b**	--
0x2003	**c**	--
0x2004	**ptr**	--

In real code, you should always initialize variables

Assuming that integers and pointers each fit into a single memory location

Pointer Example

```
int main(int argc, char** argv) {
  int a, b, c;
  int* ptr;    // ptr is a pointer to an int

→ a = 5;
→ b = 3;
  ptr = &a;

  *ptr = 7;
  c = a + b;

  return 0;
}
```

0x2001	a	5
0x2002	b	3
0x2003	c	--
0x2004	ptr	--

Assuming that integers and pointers each fit into a single memory location

Pointer Example

```
int main(int argc, char** argv) {
  int a, b, c;
  int* ptr;    // ptr is a pointer to an int

  a = 5;
  b = 3;
→ ptr = &a;

  *ptr = 7;
  c = a + b;

  return 0;
}
```

Address	Name	Value
0x2001	**a**	5
0x2002	**b**	3
0x2003	**c**	--
0x2004	**ptr**	**0x2001**

Assuming that integers and pointers each fit into a single memory location

Pointer Example

```
int main(int argc, char** argv) {
  int a, b, c;
  int* ptr;    // ptr is a pointer to an int

  a = 5;
  b = 3;
  ptr = &a;

  *ptr = 7;
  c = a + b;

  return 0;
}
```

0x2001	**a**	7
0x2002	**b**	3
0x2003	**c**	--
0x2004	**ptr**	0x2001

Assuming that integers and pointers each fit into a single memory location

Pointer Example

```c
int main(int argc, char** argv) {
  int a, b, c;
  int* ptr;     // ptr is a pointer to an int

  a = 5;
  b = 3;
  ptr = &a;

  *ptr = 7;
  c = a + b;

  return 0;
}
```

0x2001	**a**	7
0x2002	**b**	3
0x2003	**c**	10
0x2004	**ptr**	0x2001

Assuming that integers and pointers each fit into a single memory location

Poll Everywhere

pollev.com/tqm

- What does this code print?

```
int main(int argc, char** argv) {
  int x {5};
  int y {10};
  int* z {&x};

  *z += 1;
   x += 1;

   z  = &y;
  *z += 1;

  cout << "x: " <<  x << endl;
  cout << "y: " <<  y << endl;
  cout << "z: " << *z << endl;

  return EXIT_SUCCESS;
}
```

Lecture Outline

- Pointers & Old Memory Model
- **Problems with old memory model**
- Virtual Memory High Level
- Page Replacement

Poll Everywhere

pollev.com/tqm

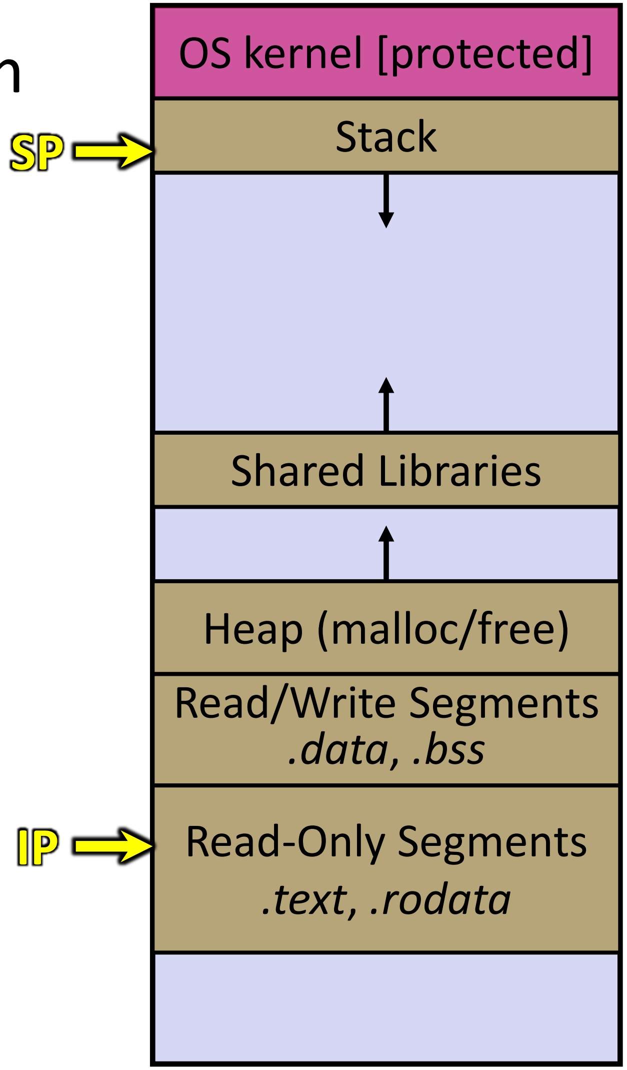
- What does this print for **x** at all three points in the code?
- Is the value of ptr the same for all three spots?
- (yes this is C code not C++ you can assume it compiles)

```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;
}
```

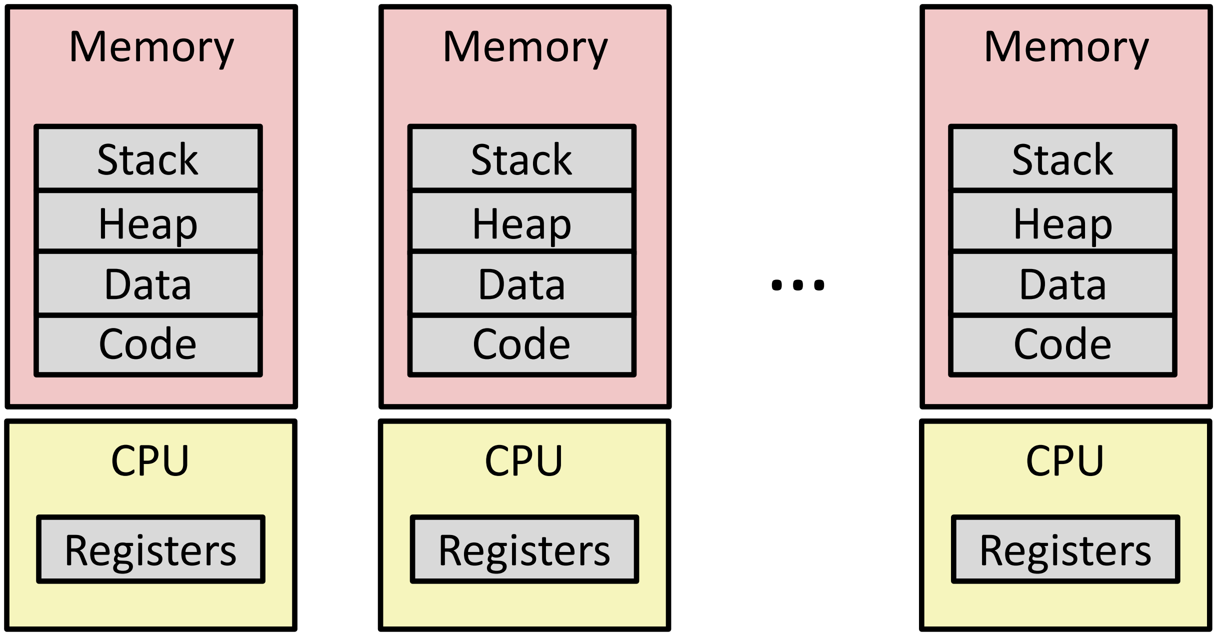
Review: Processes

- 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

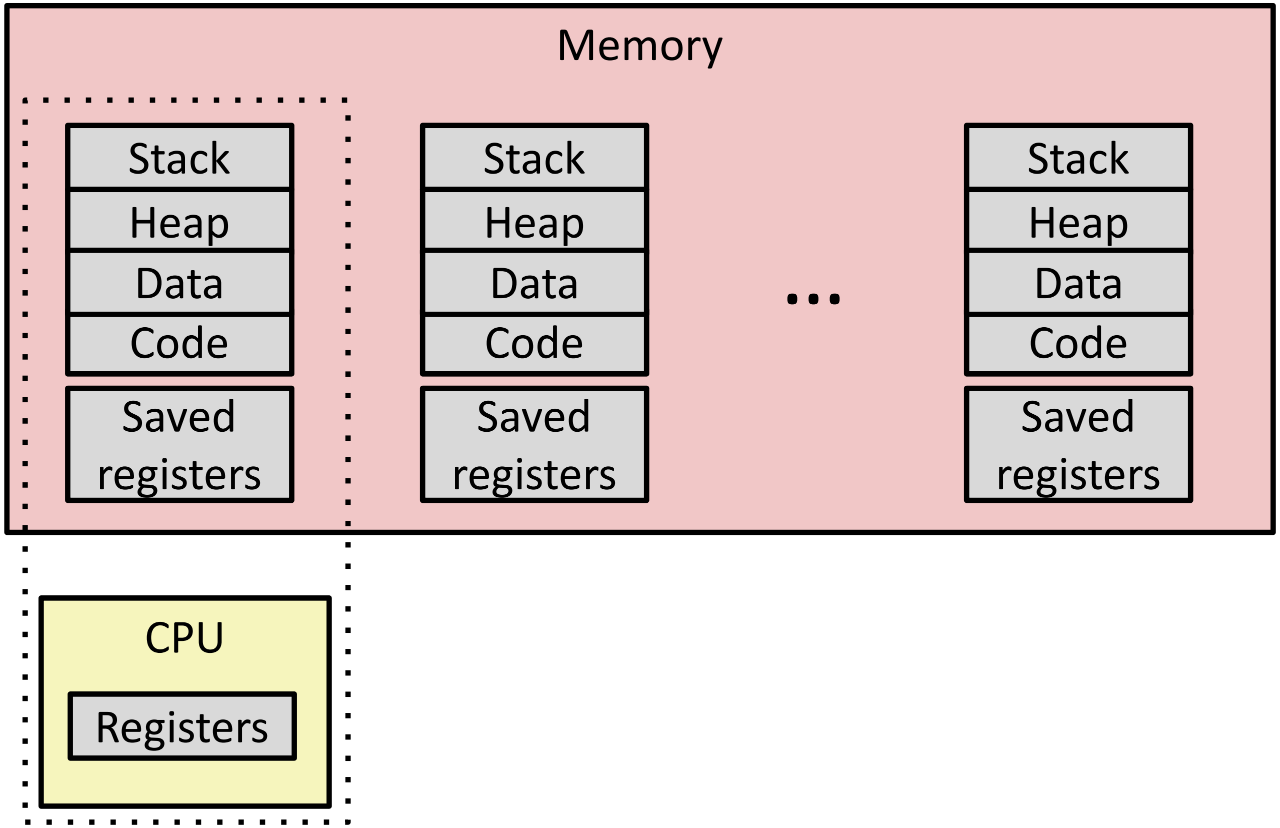


Multiprocessing: The Illusion



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

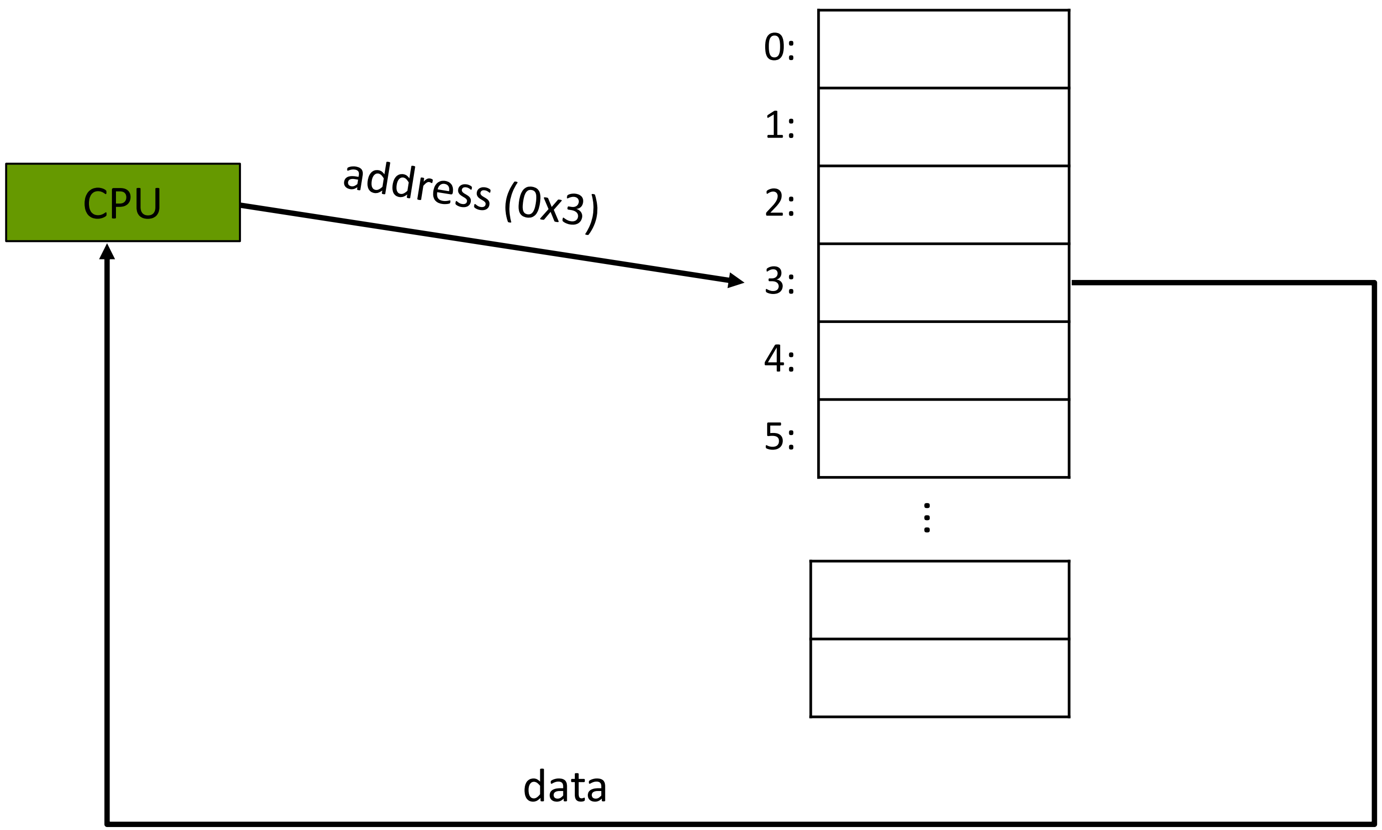
Multiprocessing: The (Traditional) Reality



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

Memory (as we know it now)

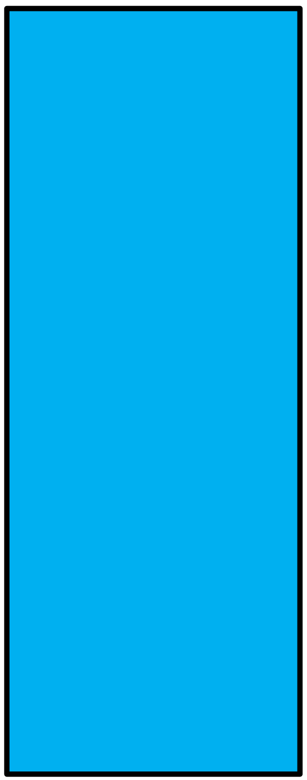
- The CPU directly uses an address to access a location in memory

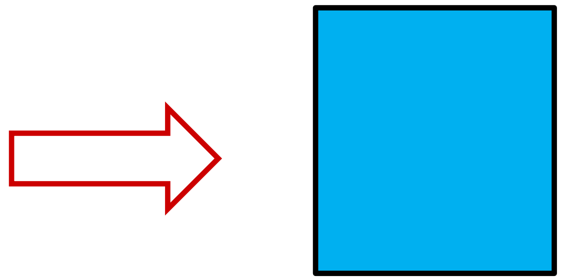


Problem 1: How does everything fit?

On a 64-bit machine, there are 2^{64} 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})

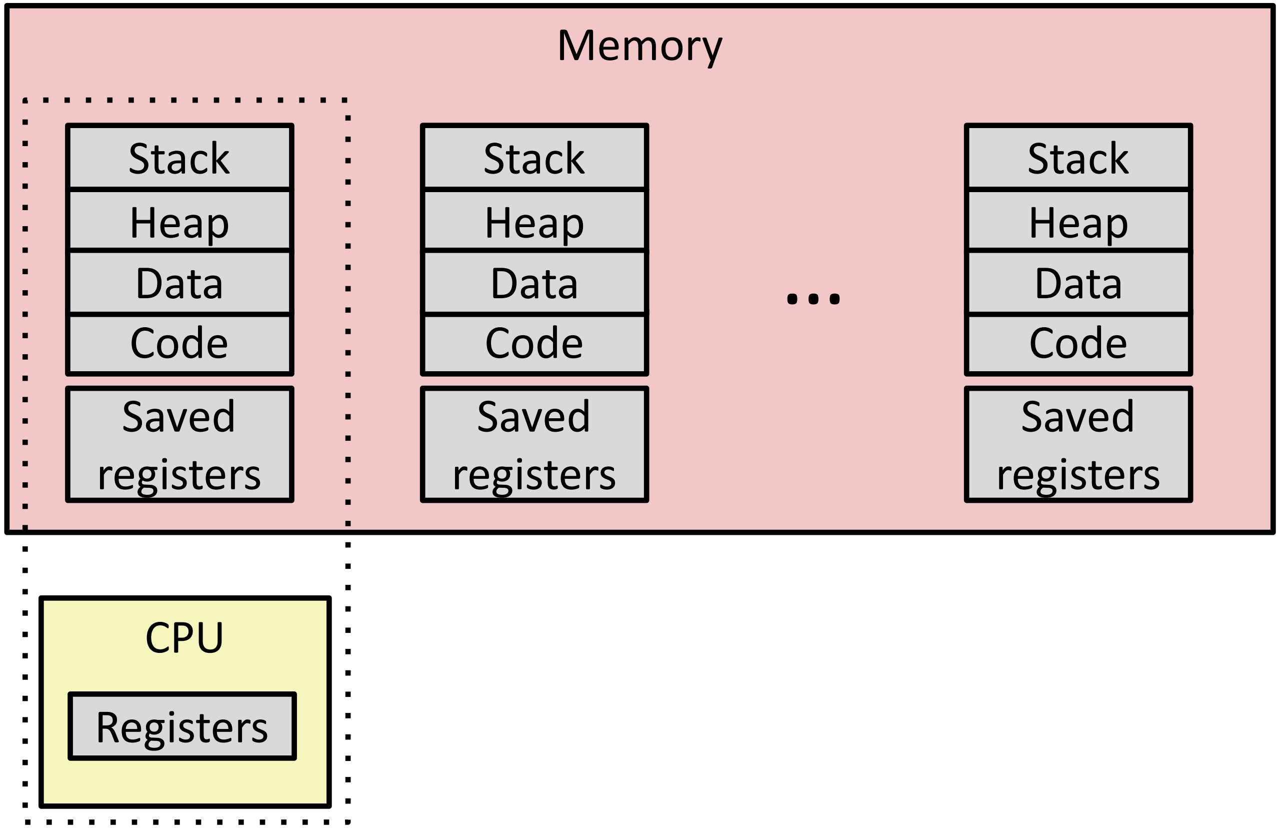




(Not to scale; physical memory is smaller than the period at the end of the sentence compared to the virtual address space.)

This is just one address space, consider multiple processes…

Problem 2: Sharing Memory

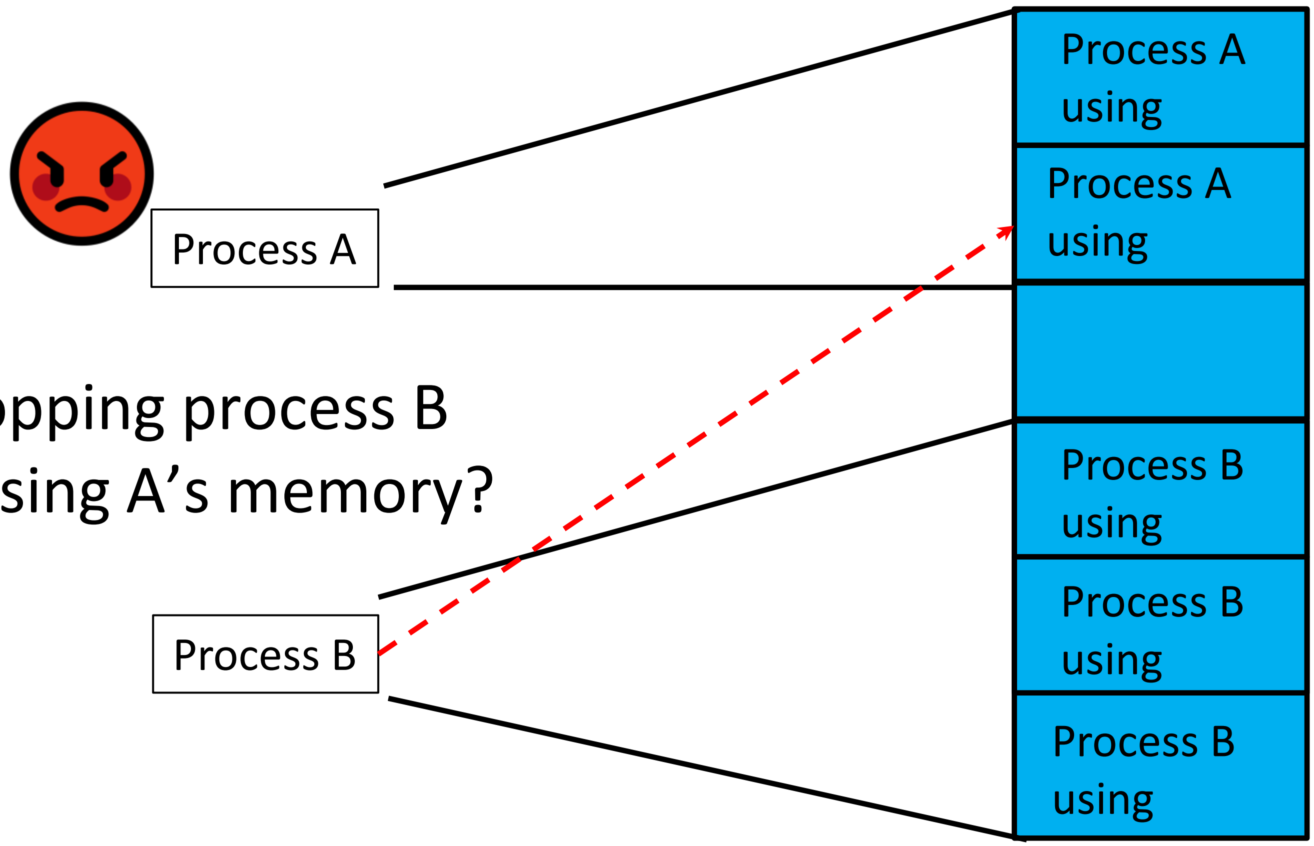


- How do we enforce process isolation?
  - Could one process just calculate an address into another process?

Problem 2: Sharing Memory

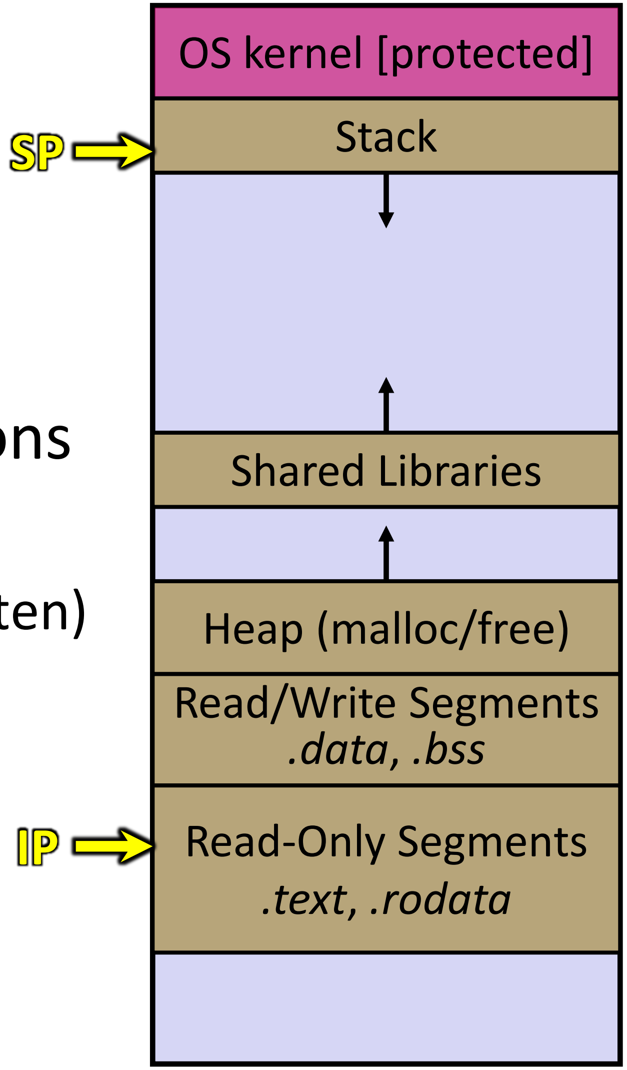
- How do we enforce process isolation?
  - Could one process just calculate an address into another process?

- 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"
- How do we keep track of which segment is which and the permissions each segment may have?
  - (e.g., that Read-Only data can't be written)



Lecture Outline

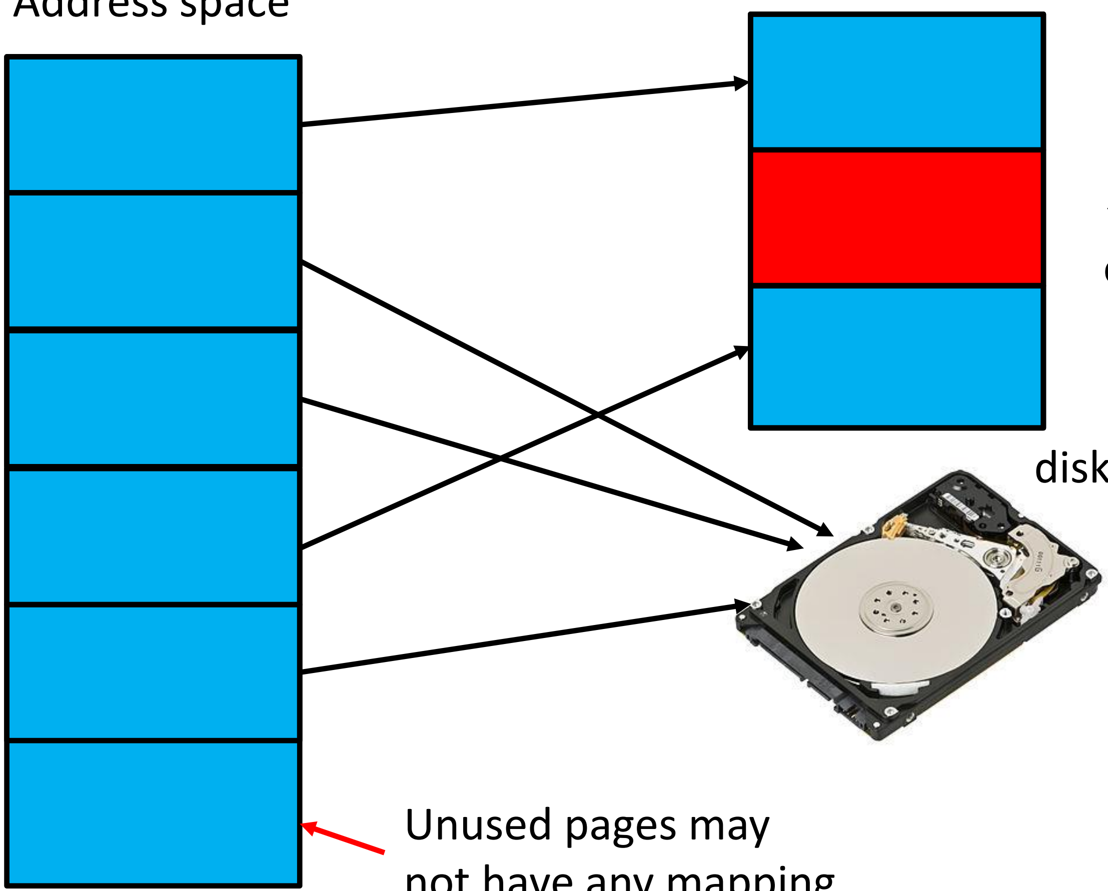
- Pointers & Old Memory Model
- Problems with old memory model
- **Virtual Memory High Level**
- Page Replacement

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 of 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)

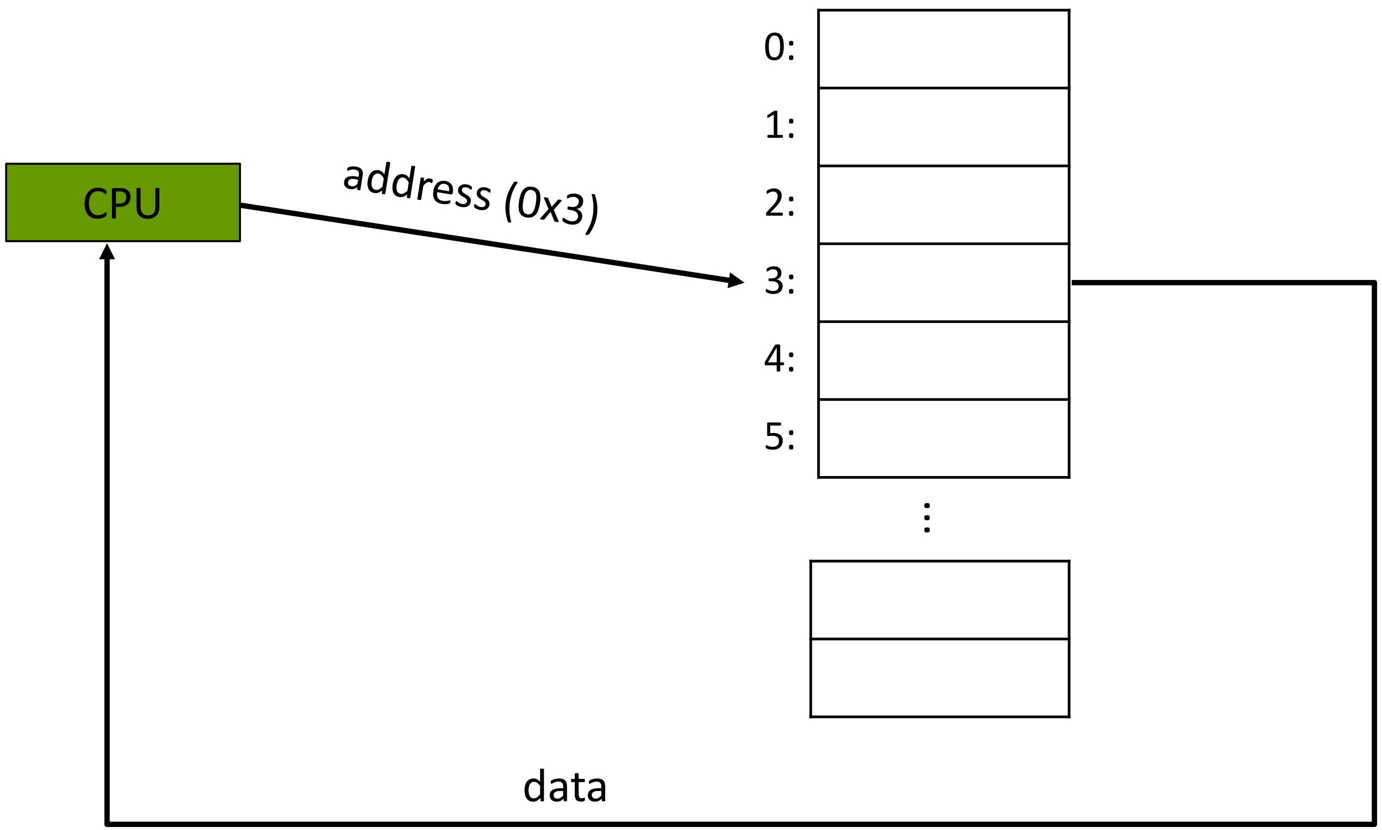
Pages

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



This doesn't work anymore

- 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 chagcing the thing without notifying holders of the name.
  - May add some work to use indirection
  - Example: Phone numbers can be transferred to new phones

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

Definitions

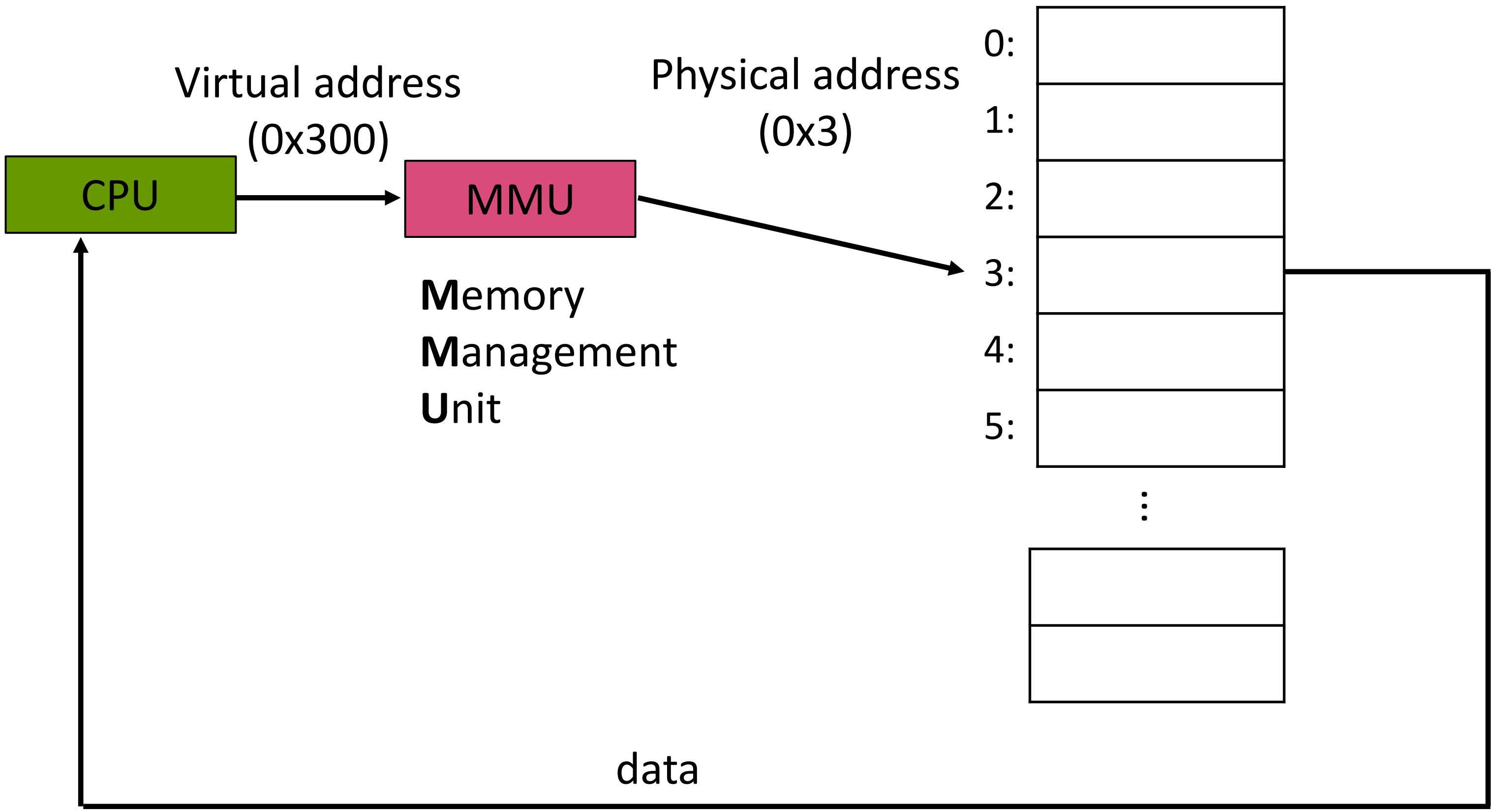
- Addressable Memory: the total amount of memory that can be theoretically be accessed based on:
  - number of addresses (“address space”)
  - bytes per address (“addressability”)

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

- 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



Page Tables

More details about translation on Wednesday

- 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
1	1	0
2	1	1
3	0	disk

Valid determines if the page is in physical memory

If a page is on disk, MMU will fetch it

This doesn't work anymore

- 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 chagcing the thing without notifying holders of the name.
  - May add some work to use indirection
  - Example: Phone numbers can be transferred to new phones

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

Idea:

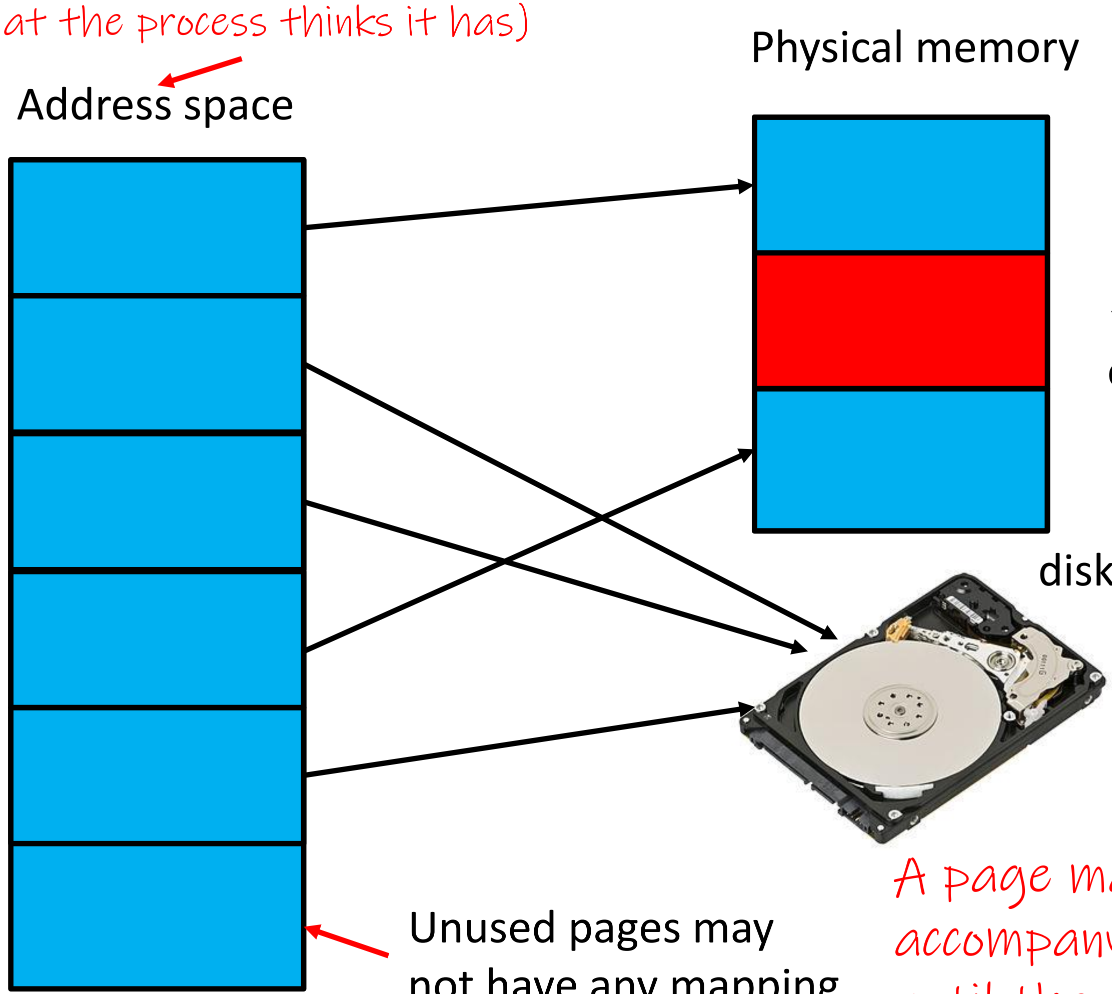
- We don’t need all processes to have their data in physical memory, **just the ones that are currently running**
- For the process’ that are currently running: we don’t need all their data to be in physical memory, **just the parts that are currently being used**
- Data that isn’t currently stored in 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 ~4KB
4KB -> (4 * 1024 = 4096 bytes.)

- Memory can be split up into units called "pages"

(what the process thinks it has)



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"

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

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 ONHARDWARE (like if that memory is never used)

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

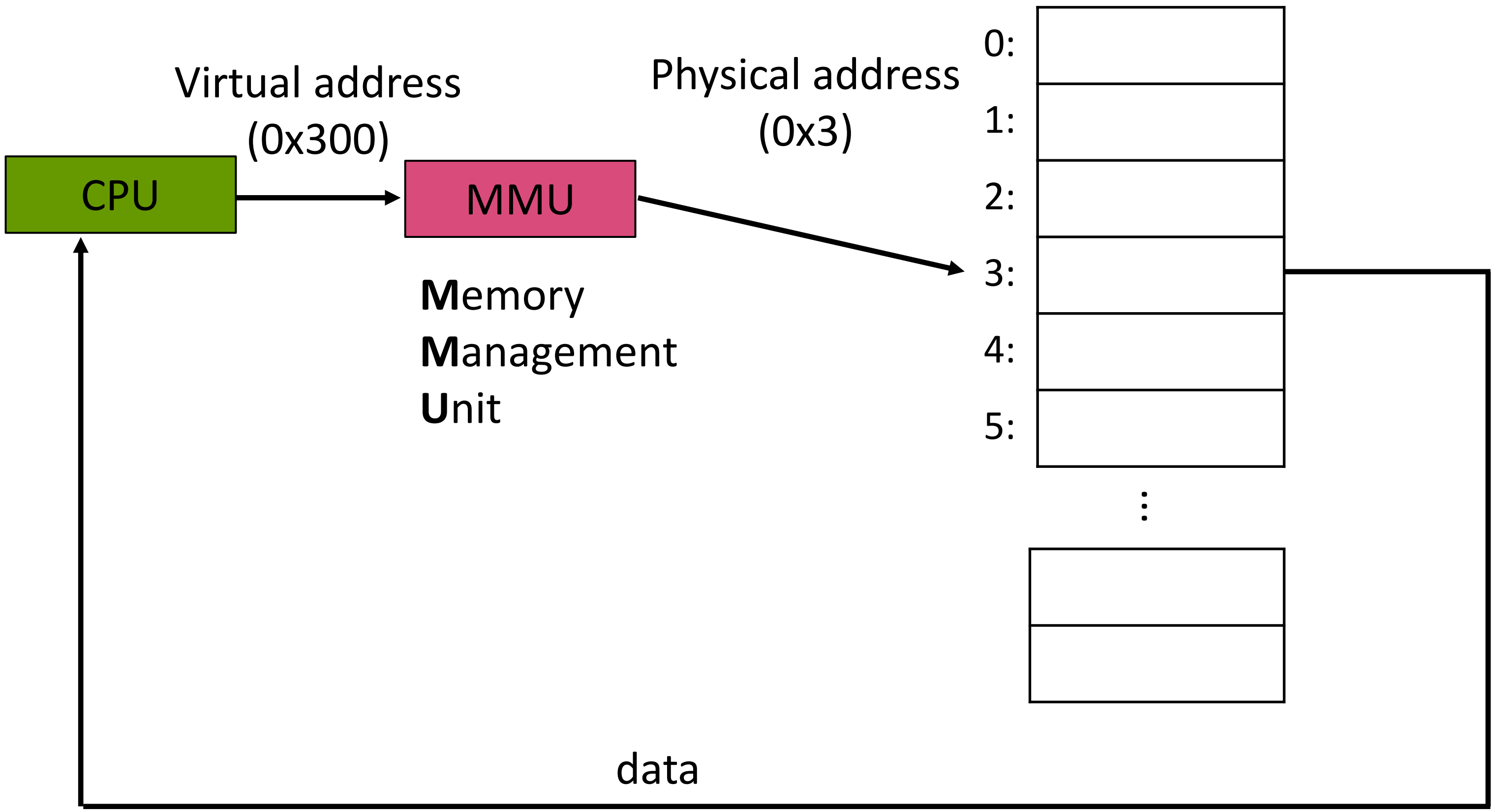
Physical memory holds a subset of the addressable memory being used

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

Virtual Address Translation

THIS SLIDE IS KEY TO THE WHOLE IDEA

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



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

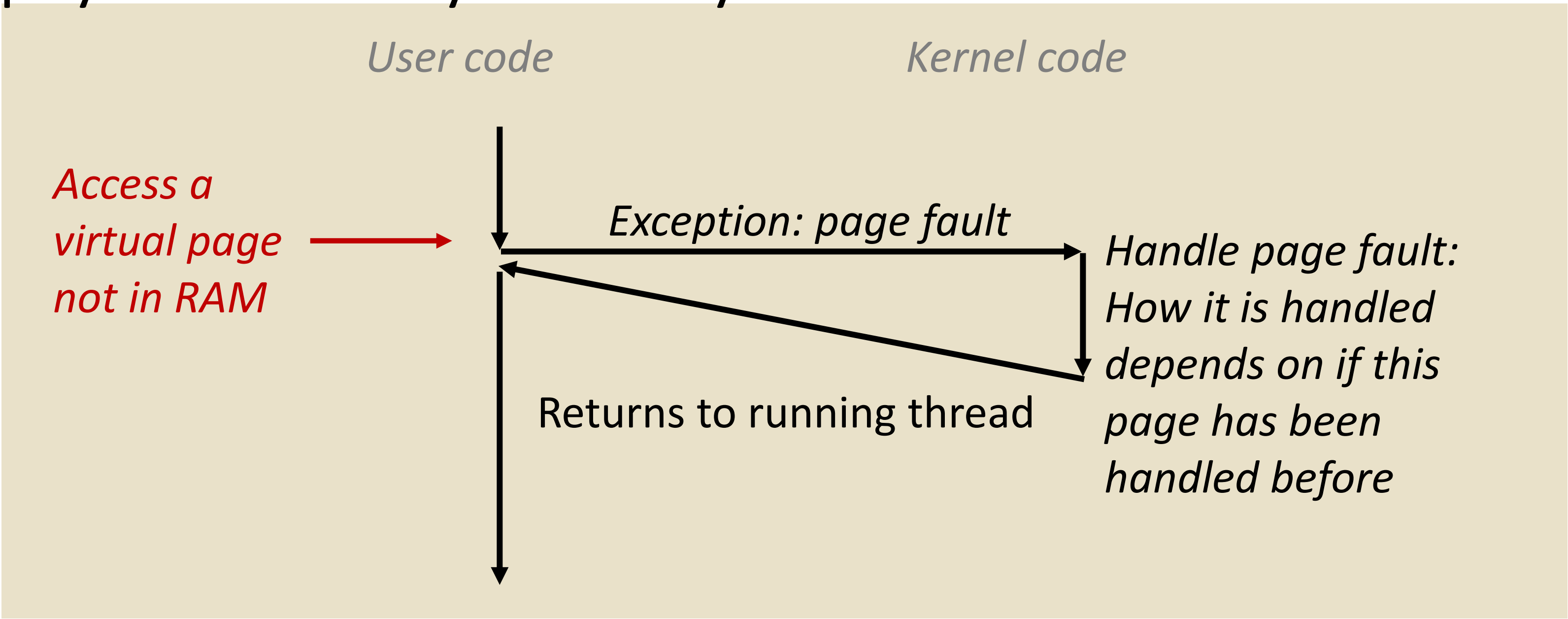
Virtual page #	Valid	Physical Page Number
0	0	null //page hasn't been used yet
1	1	0
2	1	1
3	0	disk

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 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.

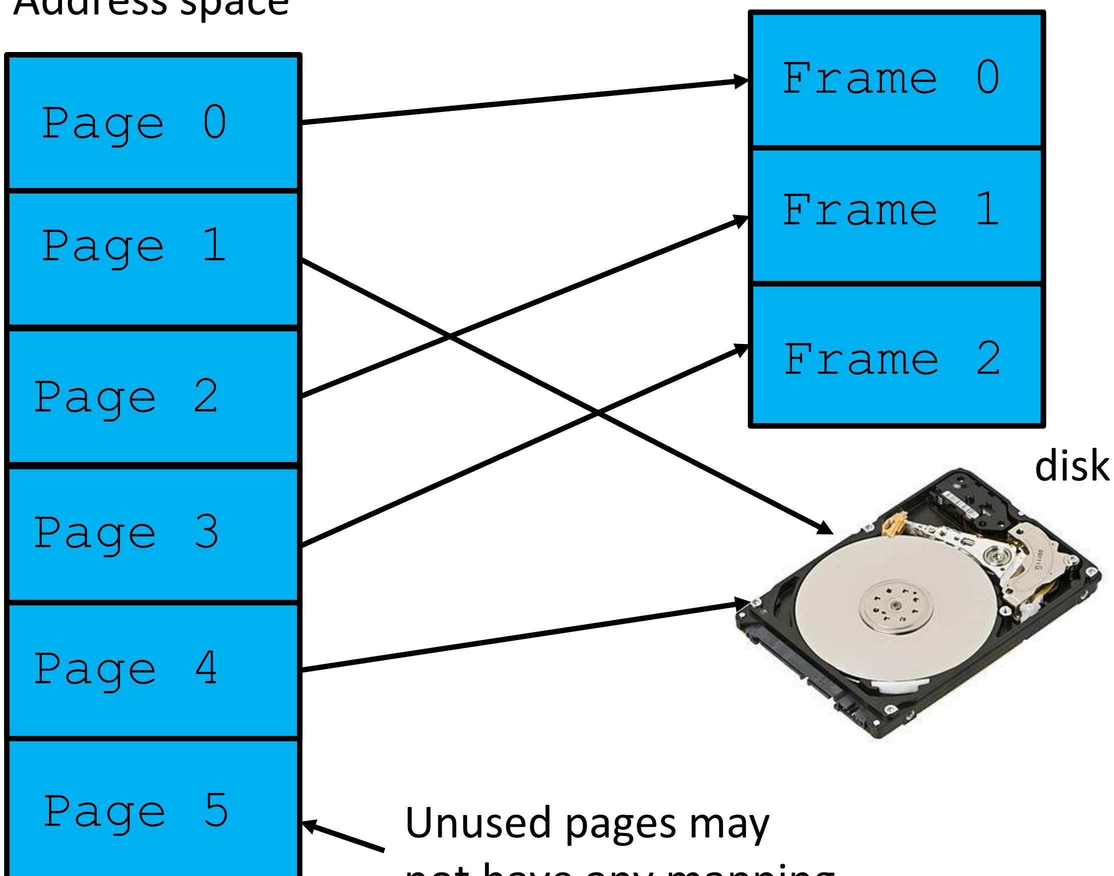
Poll Everywhere

pollev.com/tqm

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

Address space

Physical memory



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

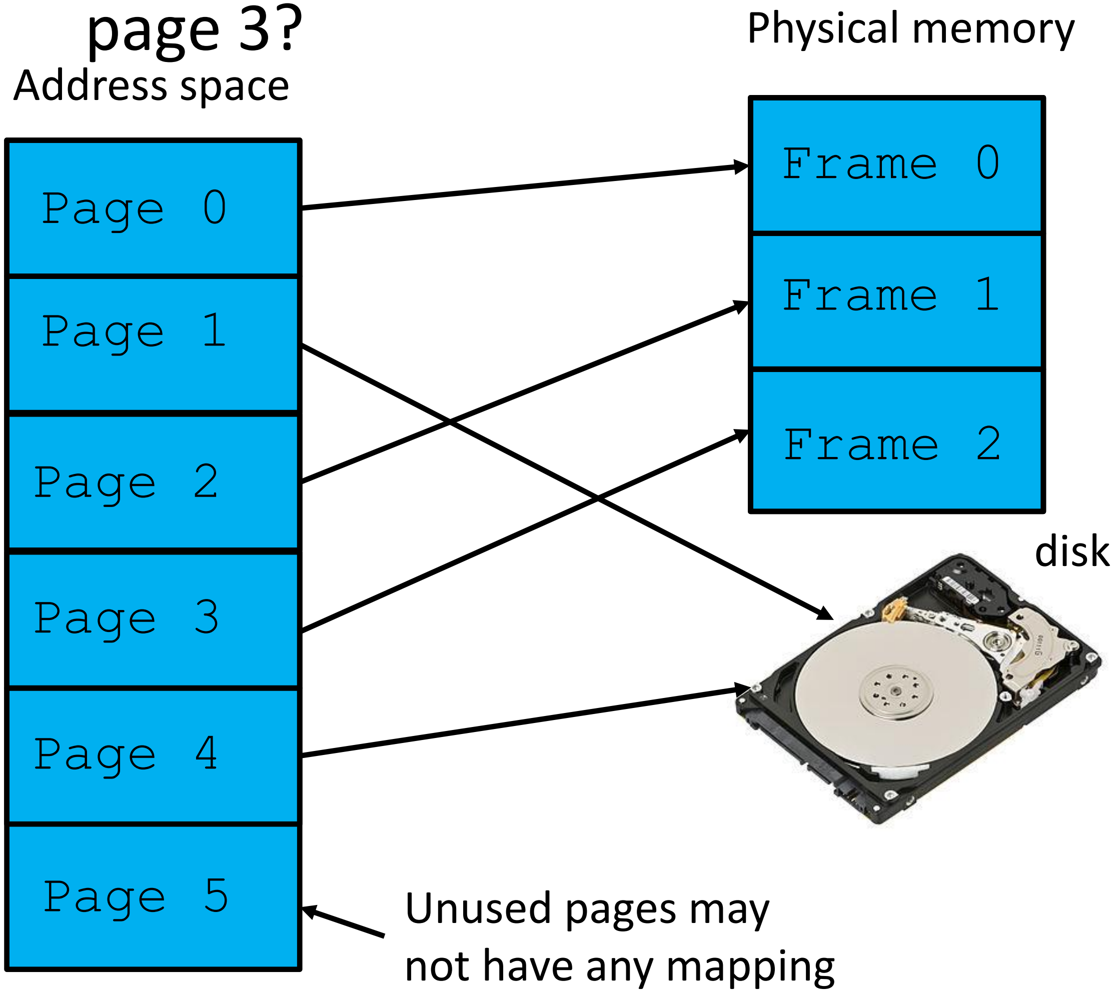
Pages not currently in use (but were used in the past) are stored on disk

Unused pages may not have any mapping

Poll Everywhere

pollev.com/tqm

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



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

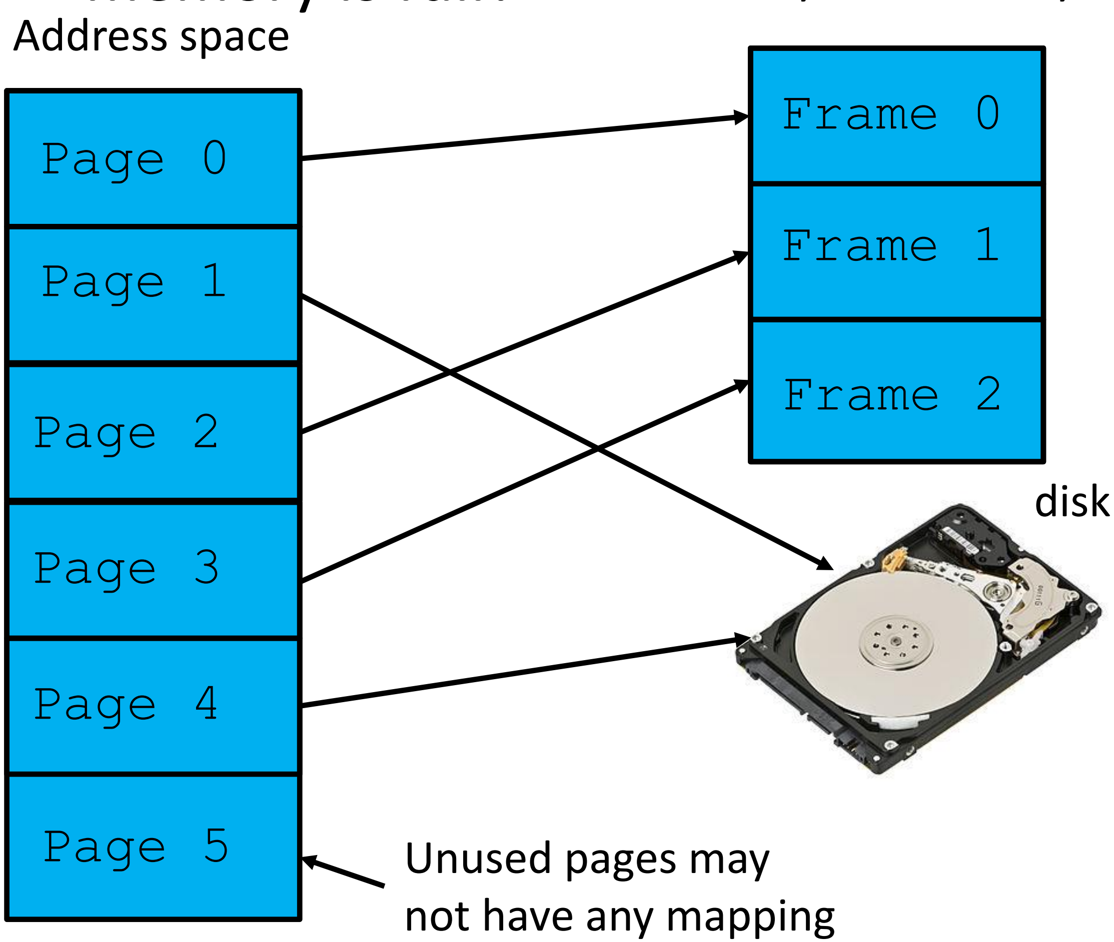
The MMU access the corresponding frame (frame 2)

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

Poll Everywhere

pollev.com/tqm

- 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)

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

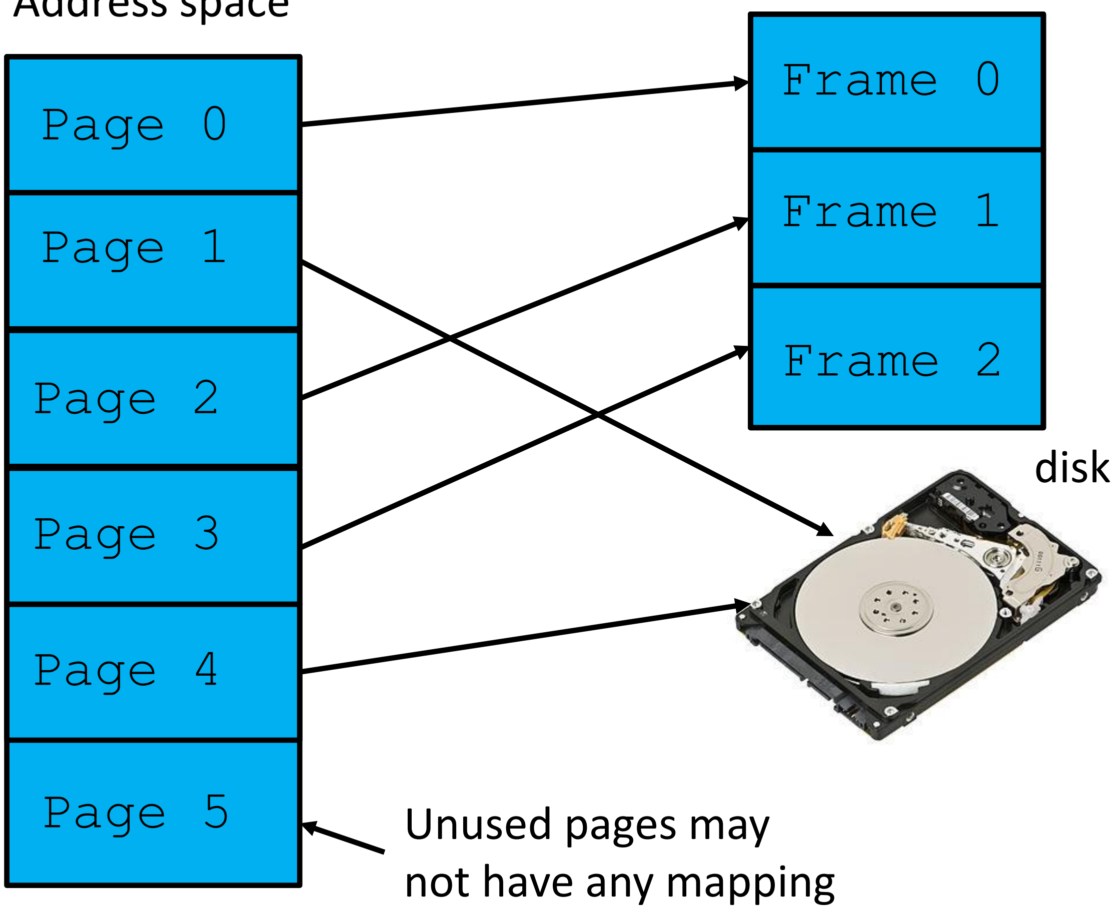
Poll Everywhere

pollev.com/tqm

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

Address space

Physical memory



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

Lecture Outline

- Pointers & Old Memory Model
- Problems with old memory model
- Virtual Memory High Level
- **Page Replacement**

Problem: Paging Replacement

- We don’t have space to store all active pages in physical memory.

- 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.

Paging Replacement Algorithms

- Simple Algorithms:
  - Random choice
    - "dumbest" method, easy to implement
  - FIFO
    - Replace the page that has been in physical memory the longest

- Both could evict a page that is used frequently and would require going to disk to retrieve it again.

(Theoretically) Optimal Algorithm

- If we knew the precise sequence of requests for pages in advance, we could optimize for smallest overall number of faults
  - Always replace the page to be used at the farthest point in future
  - Optimal (but unrealizable since it requires us to know the future)

- Off-line simulations can estimate the performance of a page replacement algorithm and can be used to measure how well the chosen scheme is doing

- Optimal algorithm can be approximated by using the past to predict the future

Least Recently Used (LRU)

- Assume pages used recently will be used again soon
  - Throw out page that has been unused for longest time

- Past is usually a good indicator for the future

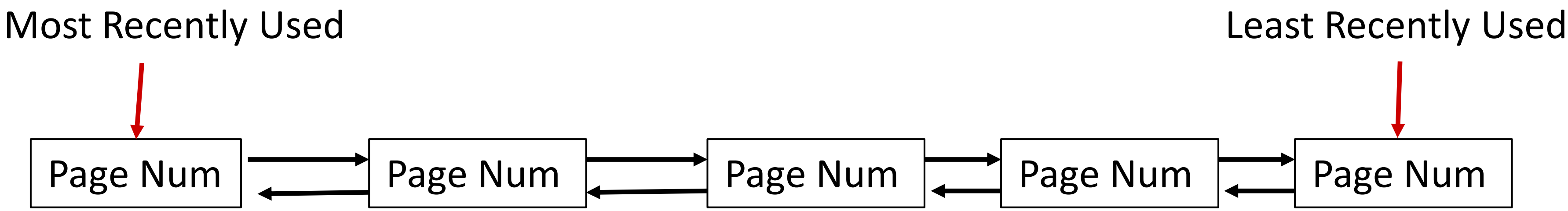
- LRU has significant overhead:
  - A timestamp for *each* memory access that is updated in the page table
  - Sorted list of pages by timestamp

How to Implement LRU?

- Counter-based solution:
  - Maintain a counter that gets incremented with each memory access
  - When we need to evict a page, pick the page with lowest counter
- List based solution
  - Maintain a linked list of pages in memory
  - On every memory access, move the accessed page to end
  - Pick the front page to evict
- HashMap and LinkedList
  - Maintain a hash map and a linked list
  - The list acts the same as the list-based solution
  - The HashMap has keys that are the page number, values that are pointers to the nodes in the linked list to support O(1) lookup

LRU Data Structure

- We can use a linked list to implement LRU

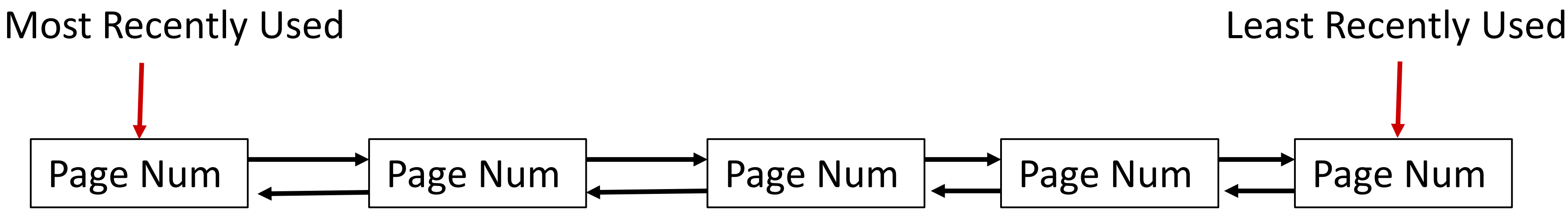


- What is the algorithmic runtime analysis to:
  - lookup a specific block?
  - Removal time?
  - Time to move a block to the front or back?

Discuss

LRU Data Structure

- We can use a linked list to implement LRU



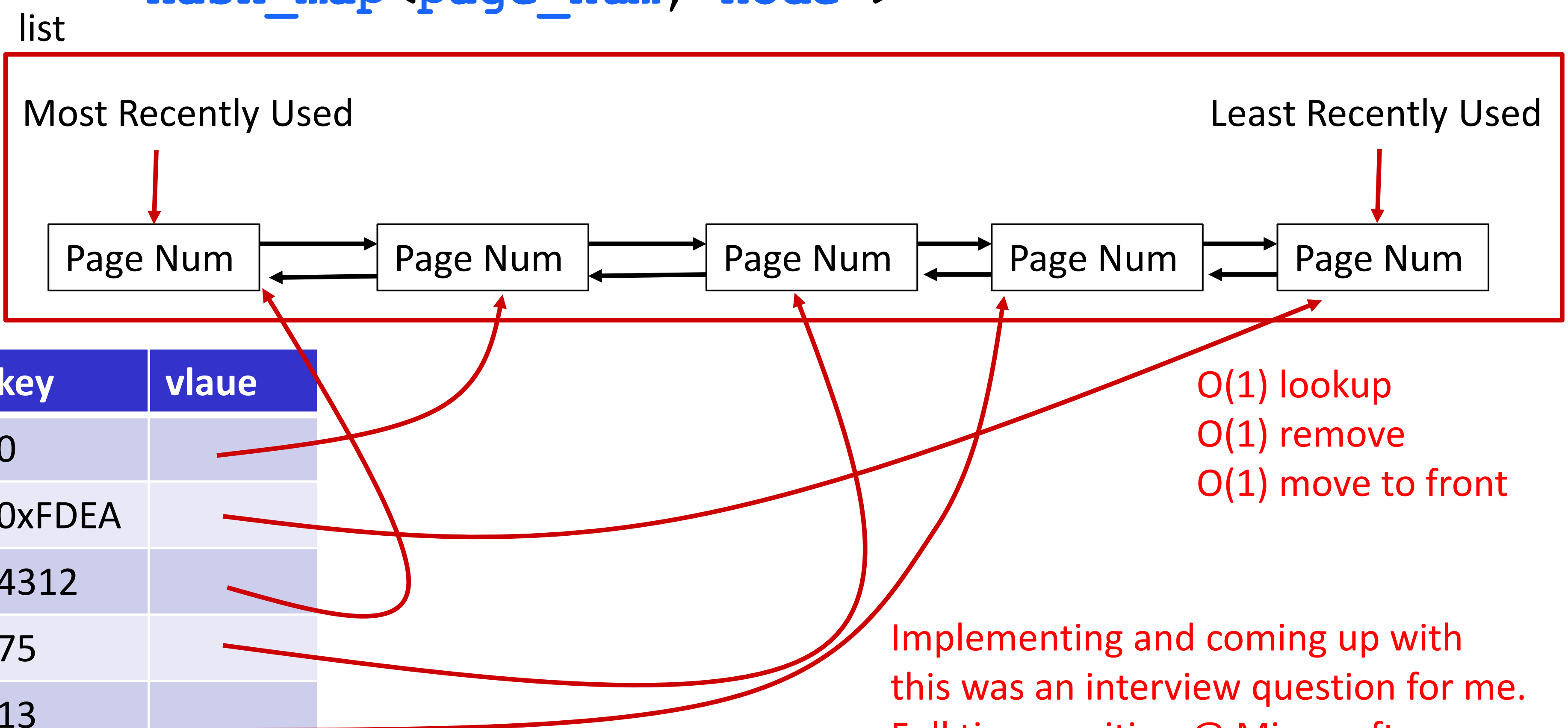
- What is the algorithmic runtime analysis to:
  - lookup a specific block? O(n)
  - Removal time? O(1)
  - Time to move a block to the front or back? O(1)

Discuss

Is there a structure we know of that has O(1) lookup time?

Chaining Hash Cache

- We can use a combination of two data structures:
  - `linked_list<page_info>`
  - `hash_map<page_num, node*>`



key	vlaue
0	
0xFDEA	
4312	
75	
13	

O(1) lookup
O(1) remove
O(1) move to front

Implementing and coming up with this was an interview question for me. Full time position @ Microsoft