Virtual Memory & Threads

Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk
Angie Cao
Aniket Ghorpade
Ashwin Alaparthi
Austin Lin
Hassan Rizwan
Lobi Zhao
Pearl Liu
Perrie Quek

Poll Everywhere

pollev.com/tqm

- How are you?

Administrivia

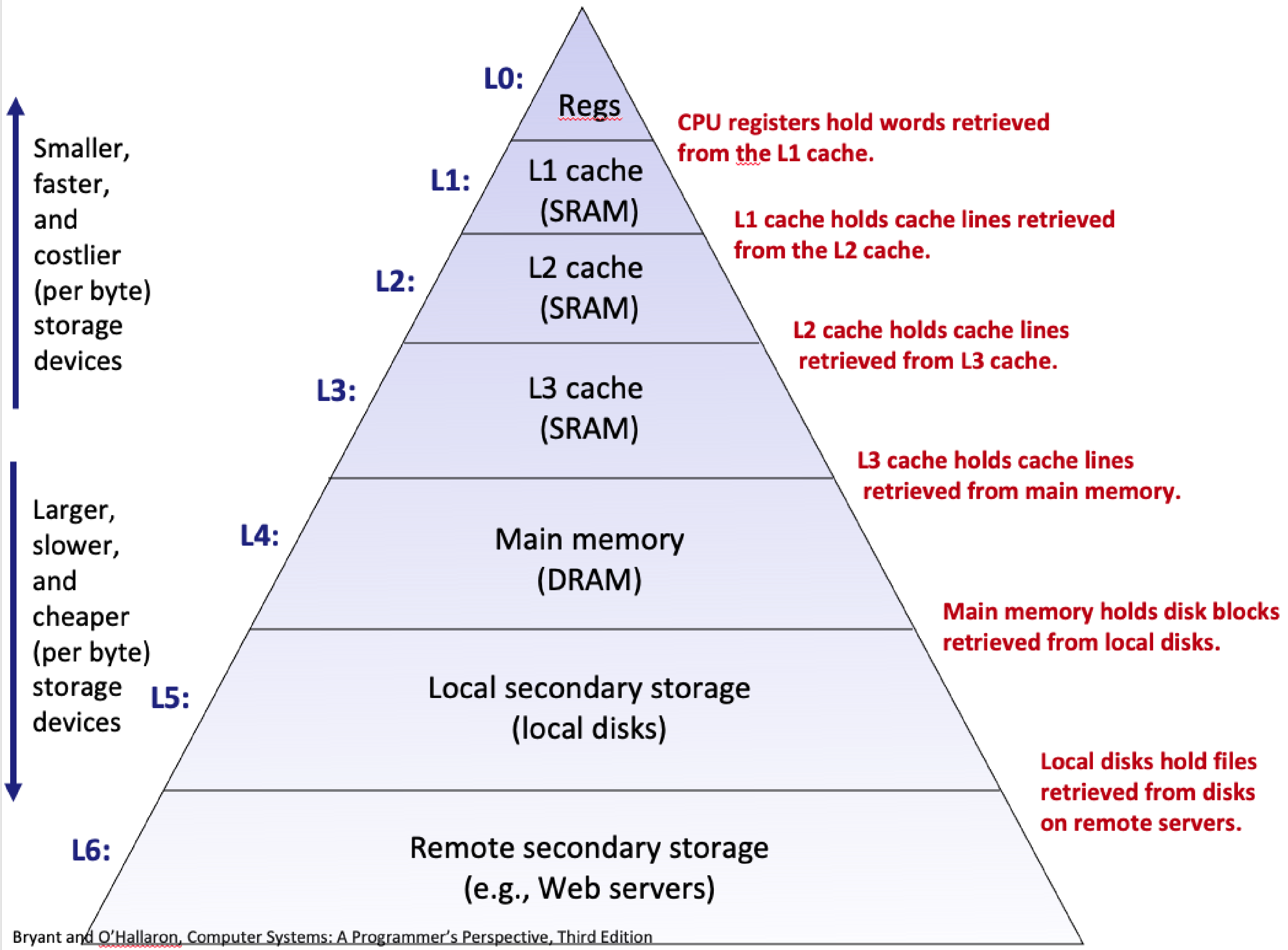
- HW07 – File Readers
  - Posted☺
  - Due Friday 3/28 at midnight, leaving open till Sunday night tho
  - AG posted soon

- Midterm grades to be posted today
  - Wide range in scores
  - Do not panic if it didn’t go well, there is the clobber policy
  - Coding on paper will be on the final too

Lecture Outline

- **Virtual Memory**
- Threads

Memory Hierarchy



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

Memory as an array of bytes

- Everything in memory is made of bits and bytes
  - Bits: a single 1 or 0
  - Byte: 8 bits

- Memory is a giant array of bytes where everything* is stored
  - Each byte has its own address ("index")

- Some types take up one byte, others more

```c
int main() {
   char c = 'A';
   char other = '0';
   int x = 5950;
   int* ptr = &x;
}
```

0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F	0x10	0x11	0x12	
'A'	'0'			5950				0x0000000000000008							…

Poll from last time

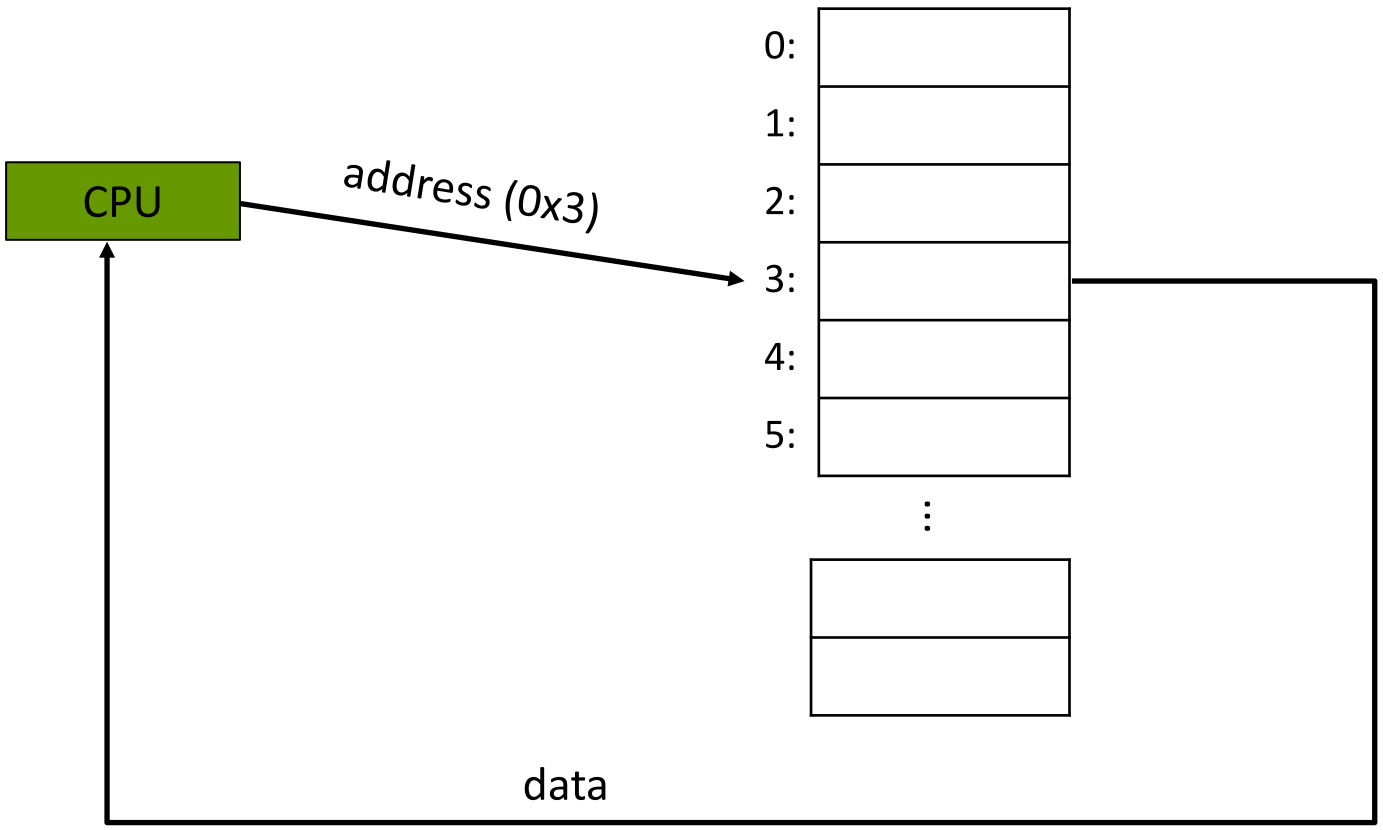
- What does this print for **`x`** and the **`ptr`**?

- ptr is printed as the same address from child and parent. Each has their own value.

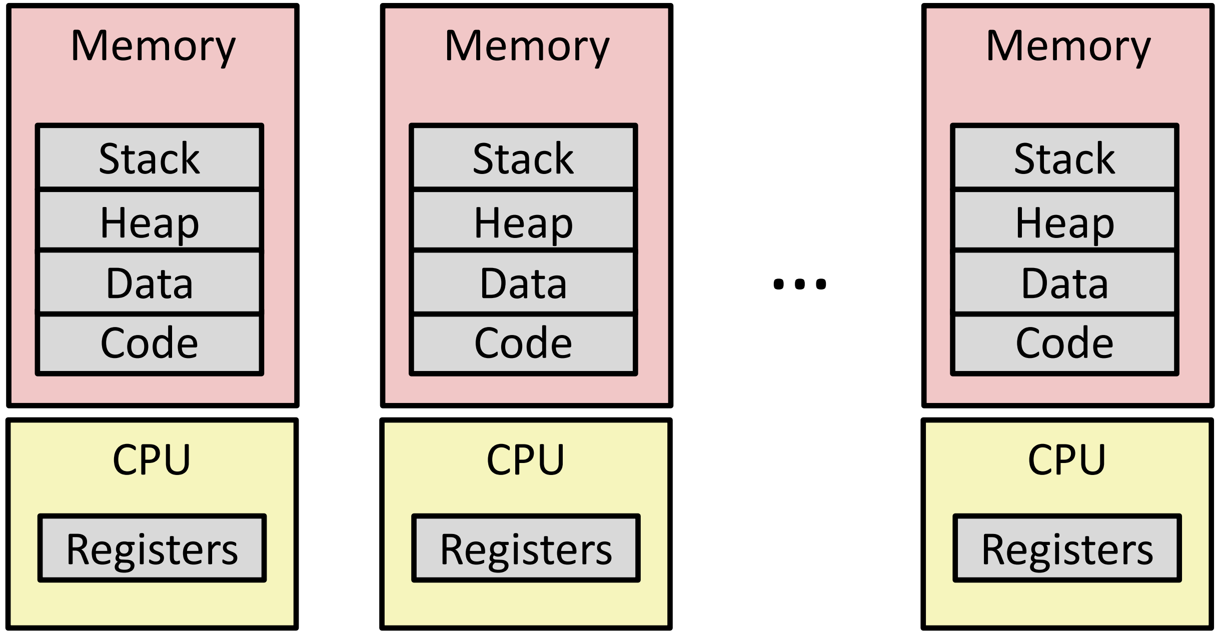
```
int main() {
  int x = 5;
  int* ptr = &x;
  pid_t pid = fork();

  if (pid == 0) {
    *ptr += 1;
    cout << x << endl;
    cout << ptr << endl;
    exit(EXIT_SUCCESS);
  }

  waitpid(pid, NULL, 0);
  *ptr += 1;
  cout << x << endl;
  cout << ptr << endl;
}
```

Memory (as we know it now)

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

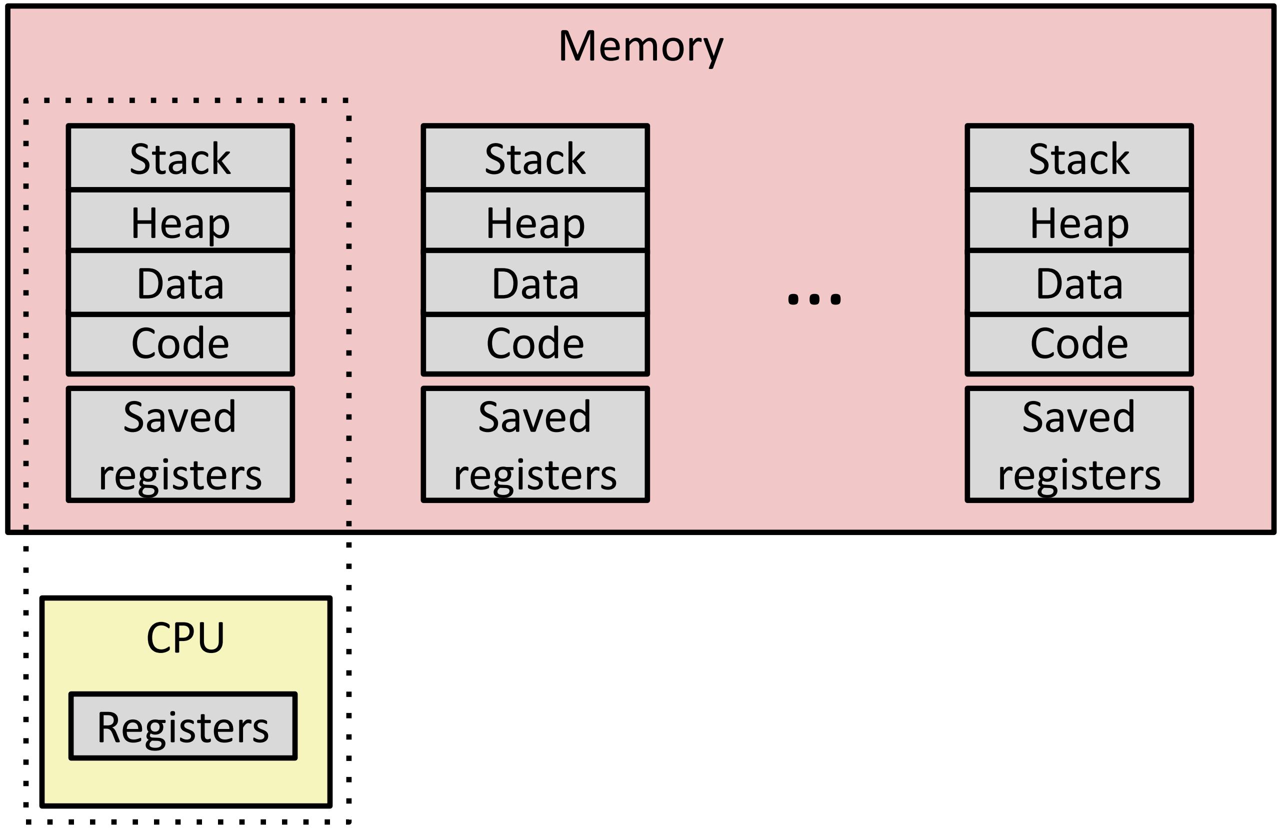


Multiprocessing: The Illusion



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

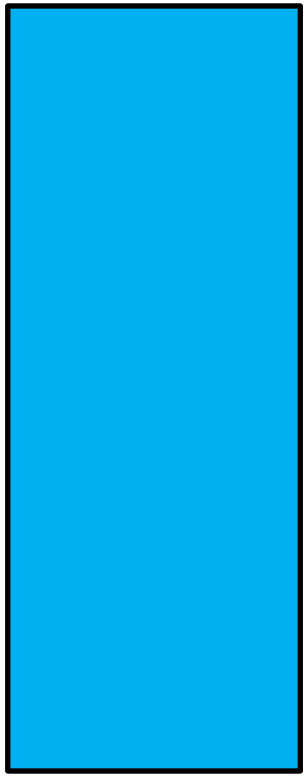
Multiprocessing: The (Traditional) Reality



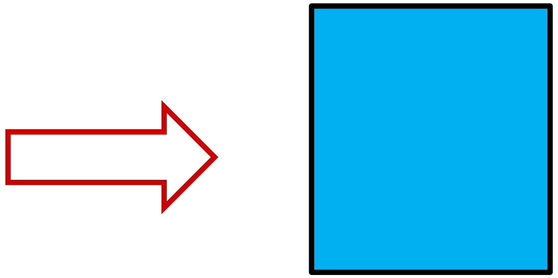
- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

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})



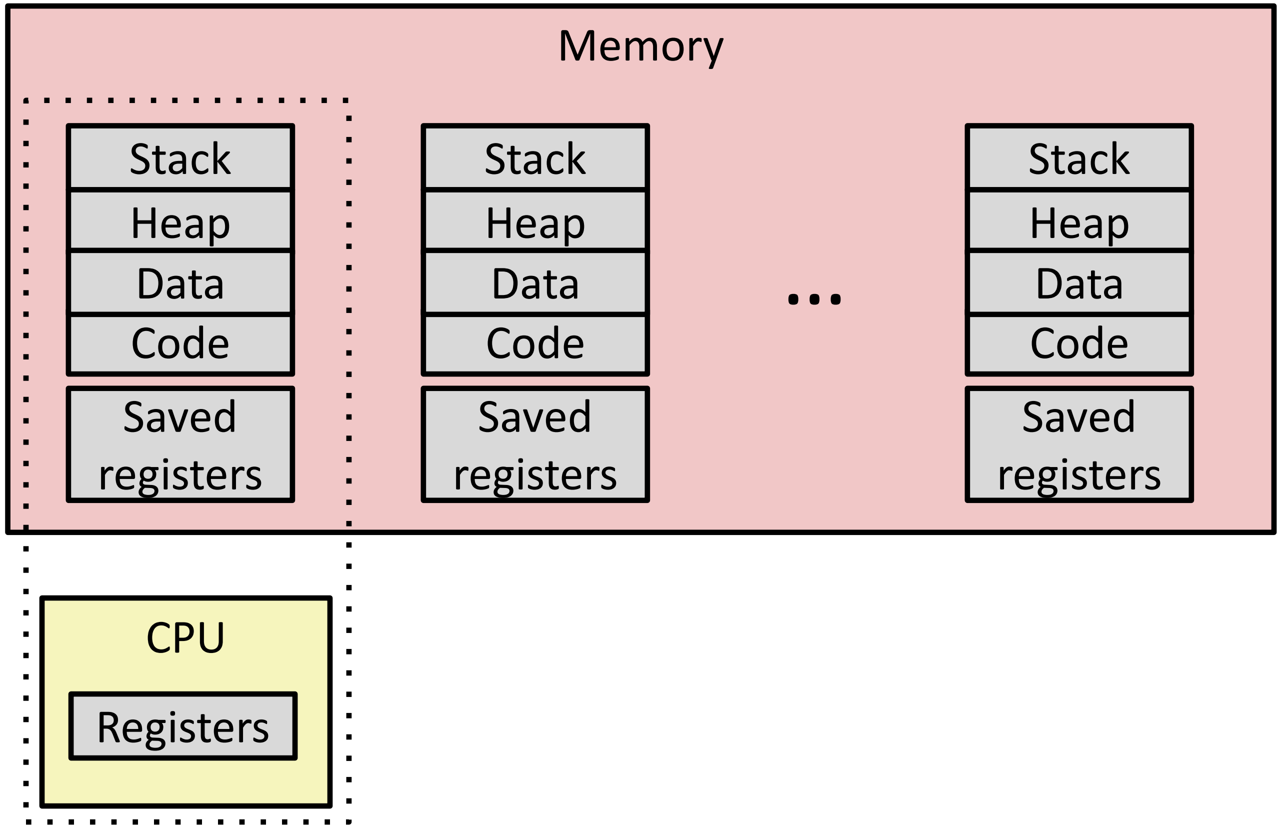
Laptops usually have around 8GB which is 8,589,934,592 Bytes (8.589×10^{9})



(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…

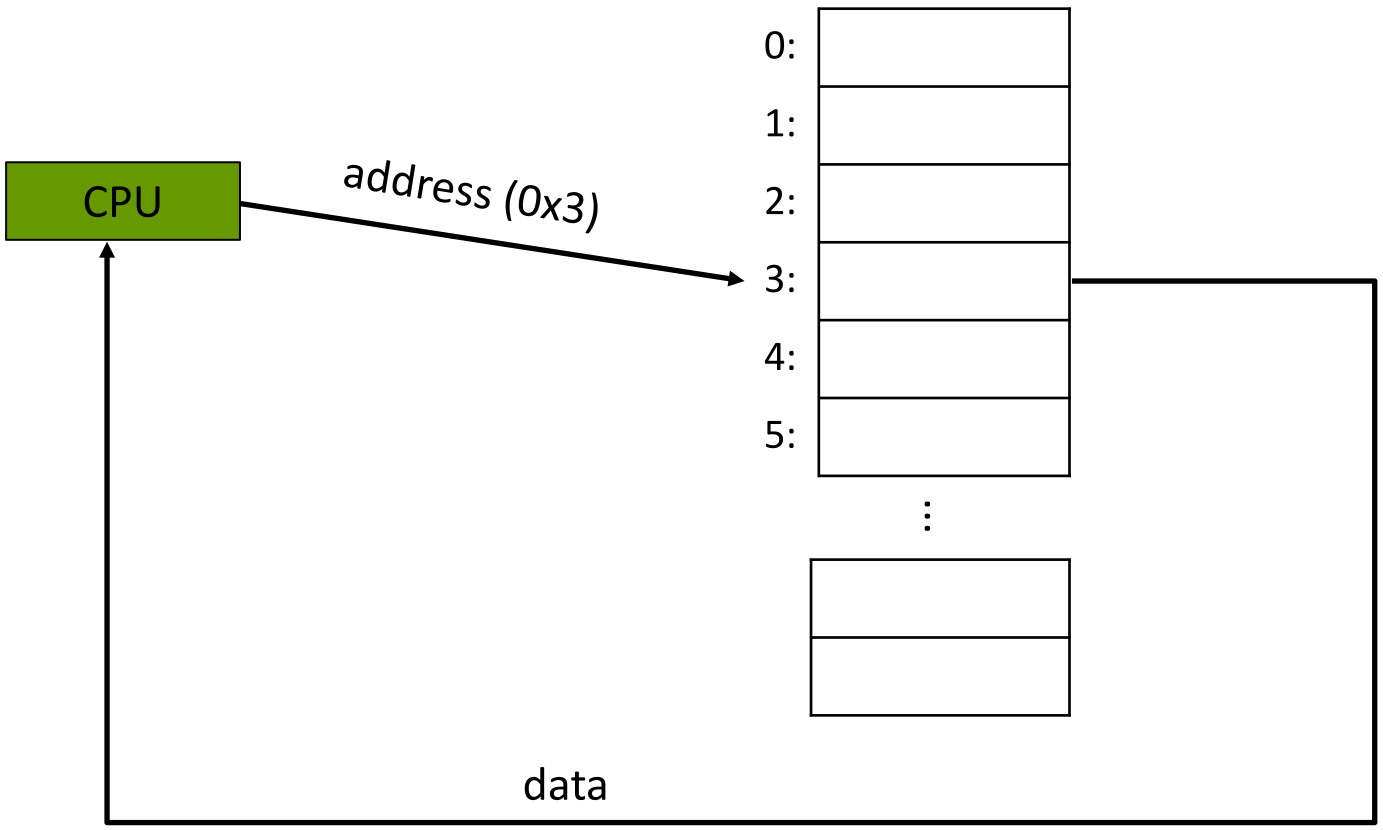
Problem 2: Sharing Memory



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

This doesn't work anymore

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



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)

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

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)

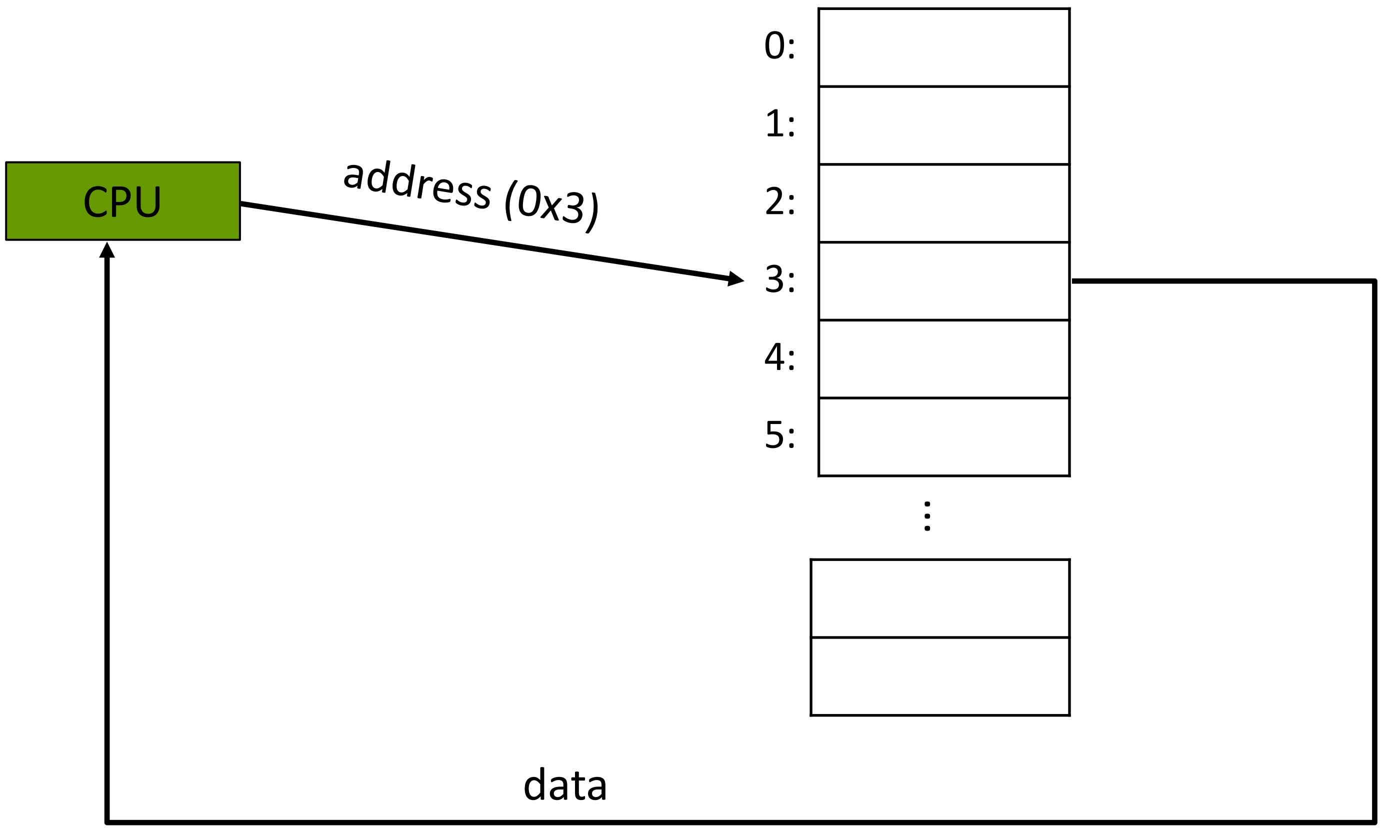
- Physical Memory: the total amount of memory that is physically available on the computer

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.

This doesn't work anymore

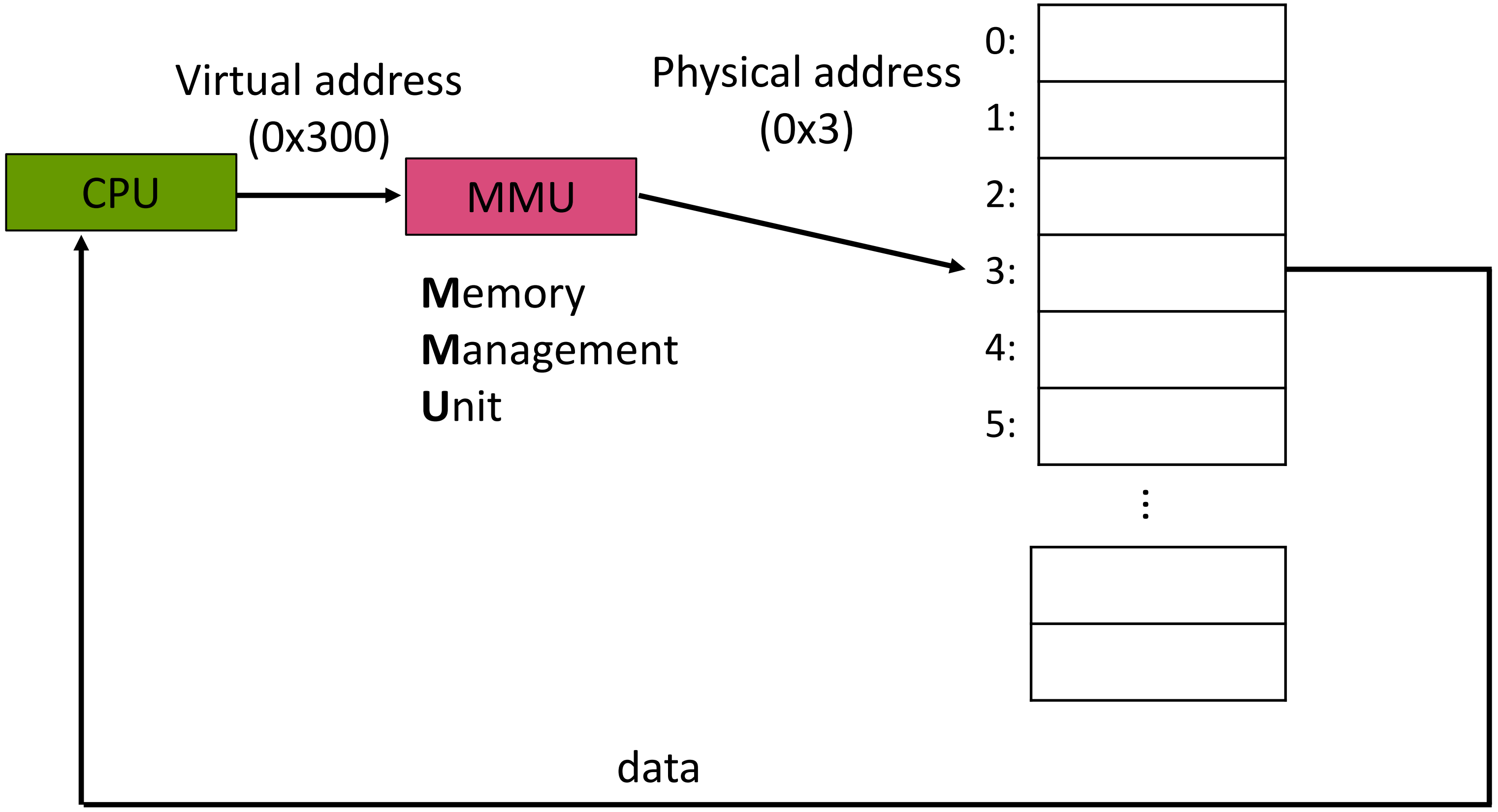
- The CPU directly uses an address to access a location in memory



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

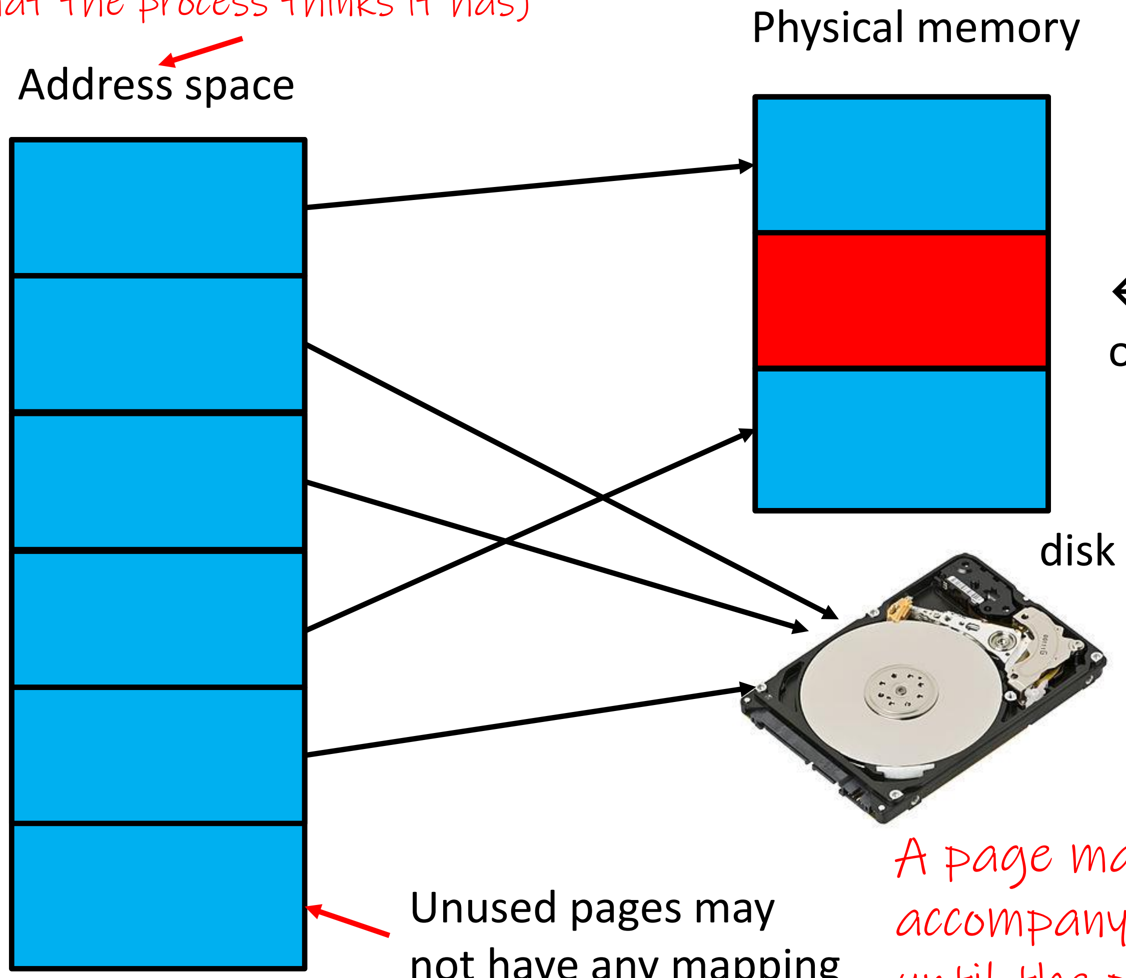


Pages

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

Unused pages may not have any mapping

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

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

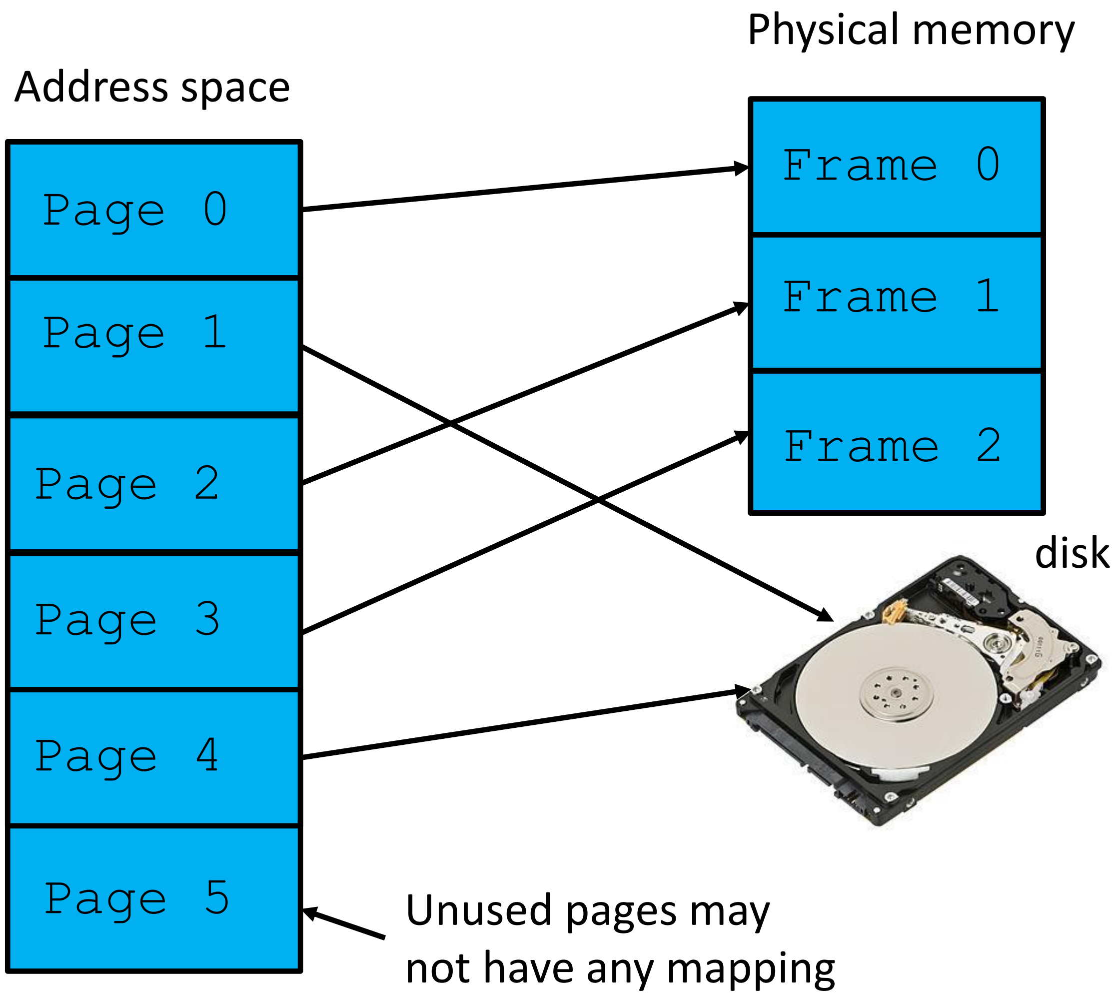
Valid determines if the page is in physical memory

If a page is on disk, MMU will fetch it

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)

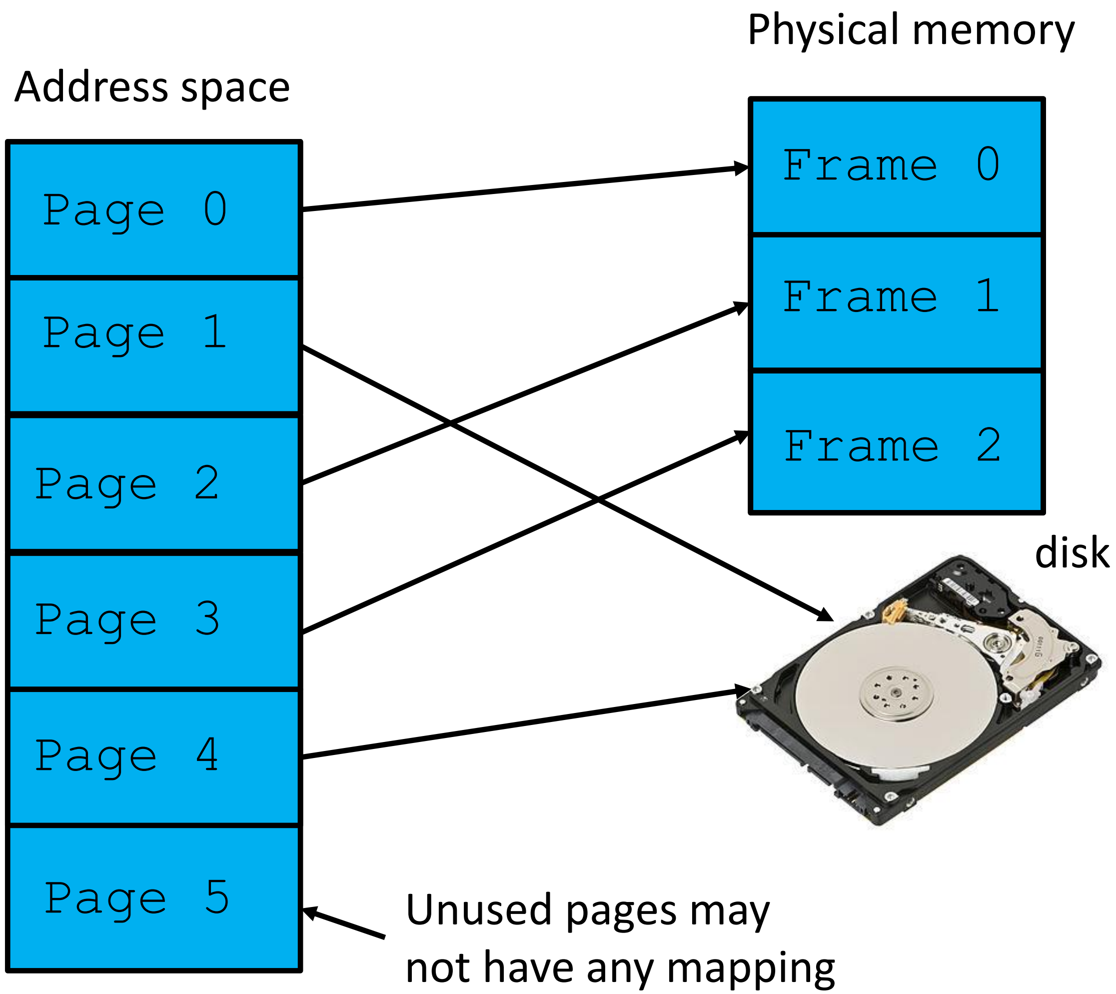
Pages not currently in use (but were used in the past) are stored on disk

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?



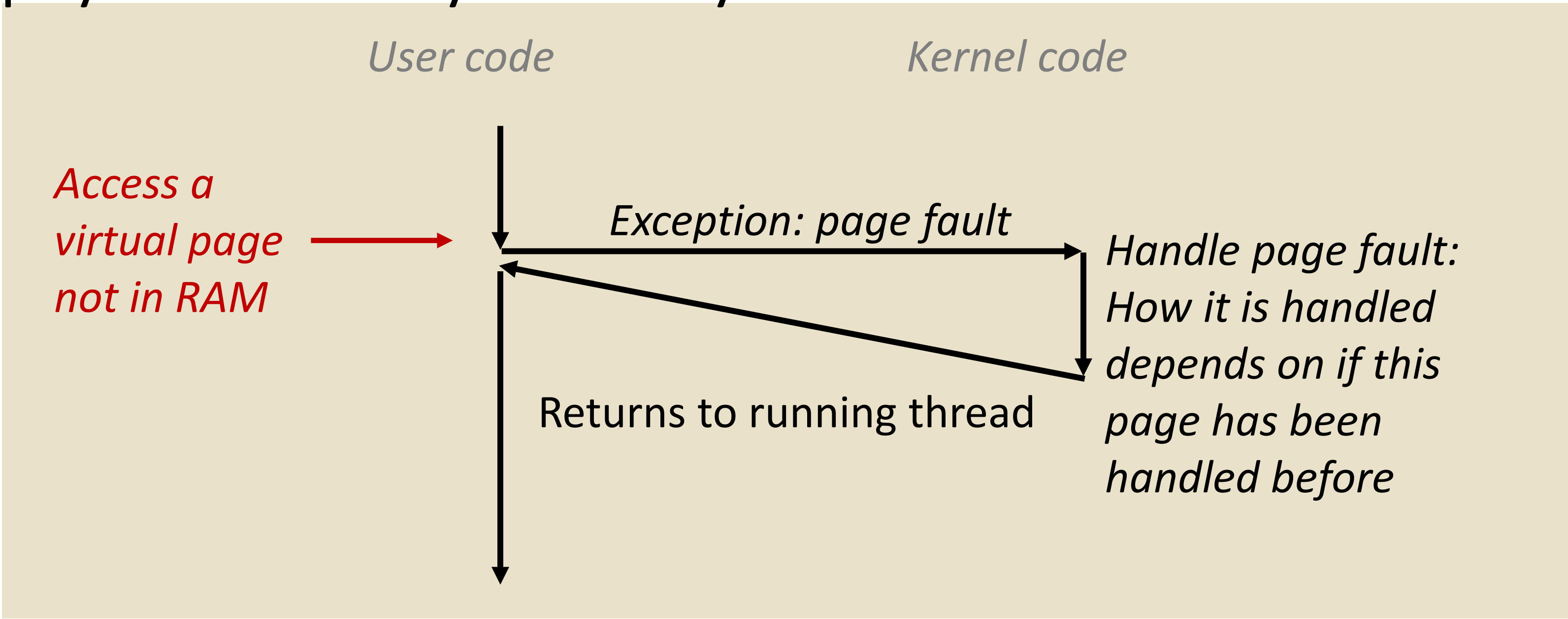
Pages currently in use are stored in physical memory (RAM)

The MMU access the corresponding frame (frame 2)

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

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



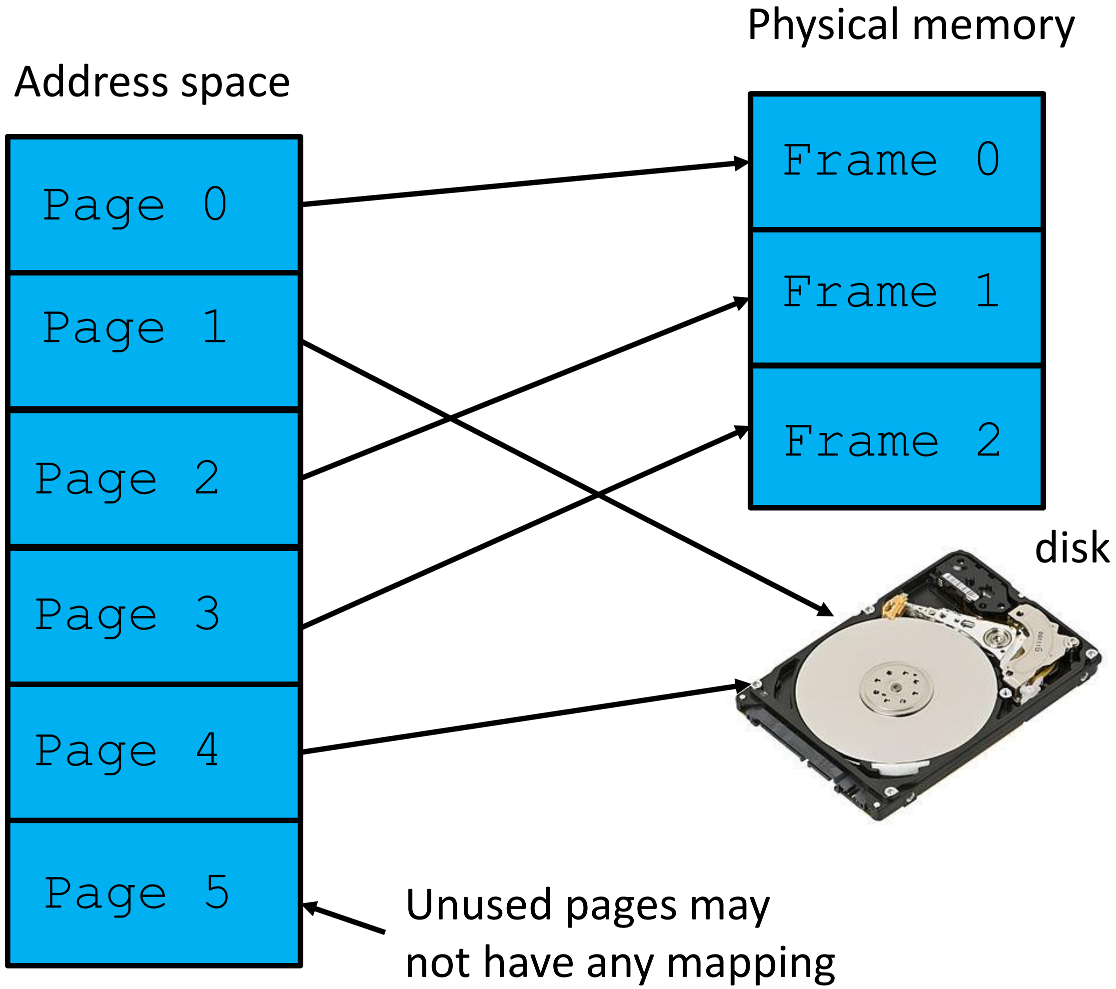
Problem: Paging Replacement

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



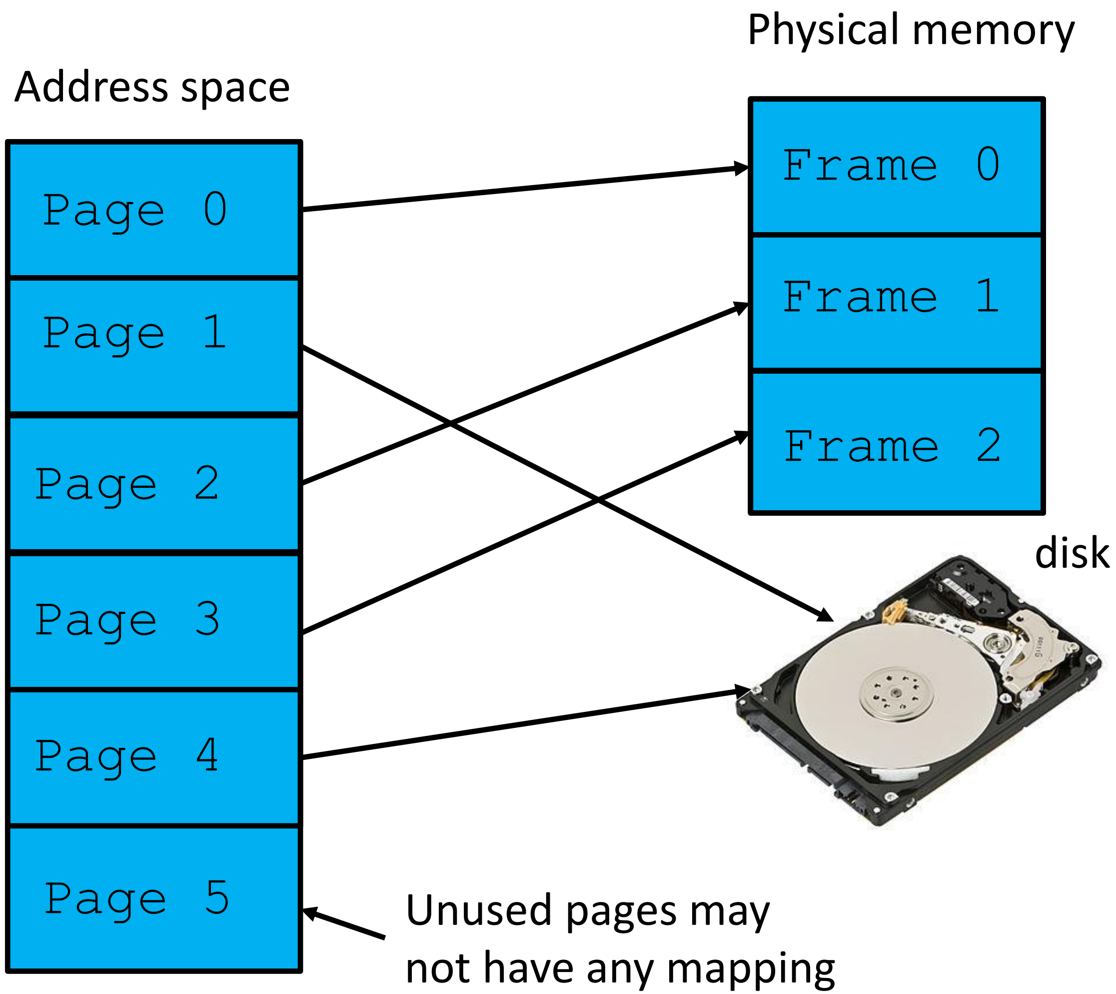
Pages currently in use are stored in physical memory (RAM)

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?



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

Unused pages may not have any mapping

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.

Poll Everywhere

pollev.com/tqm

- FIFO: Replace the page that has been in physical memory the longest

- If Memory can hold 4 physical pages, memory starts empty, and we access the pages numbered 1 2 1 2 3 1 2 4 1 2 5 1 2 6
  - How many page faults occur?

(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

Poll Everywhere

pollev.com/tqm

- "Prove" (or provide a counter example) that LRU is always better than LIFO in every case.

  - LIFO: Replace the page that has been in physical memory the longest
  - LRU: throw out page that has been unused for longest time

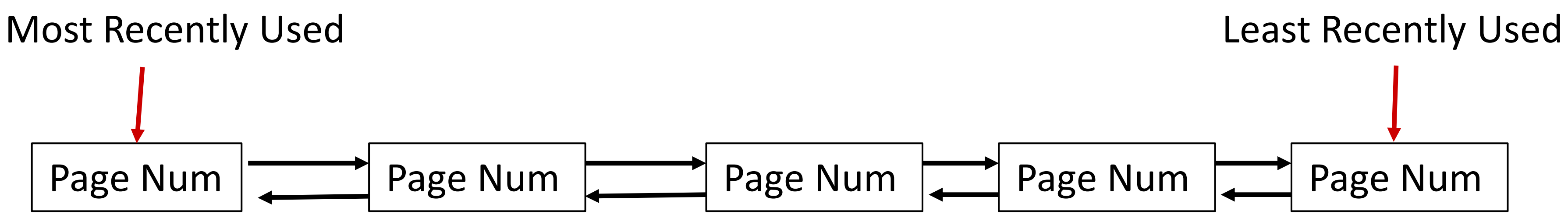
  - Can assume Physical memory can hold 4 pages

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

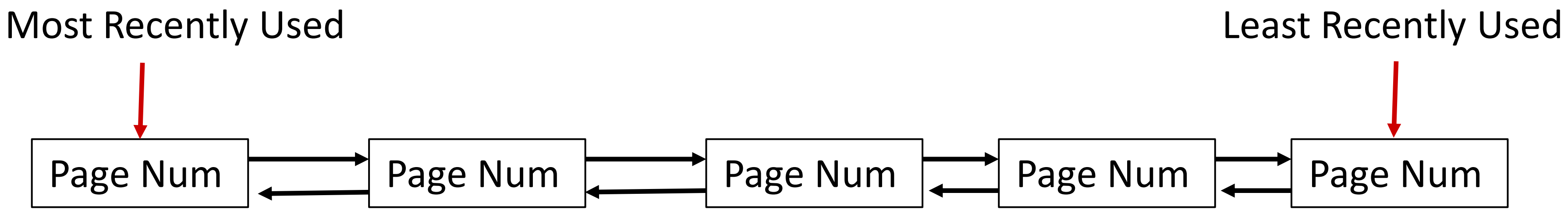


- What is the algorithmic runtime analysis to:
  - lookup a specific block?
  - Removal time?
  - Time to move a block to the front or back?

Discuss

LRU Data Structure

- We can use a linked list to implement LRU



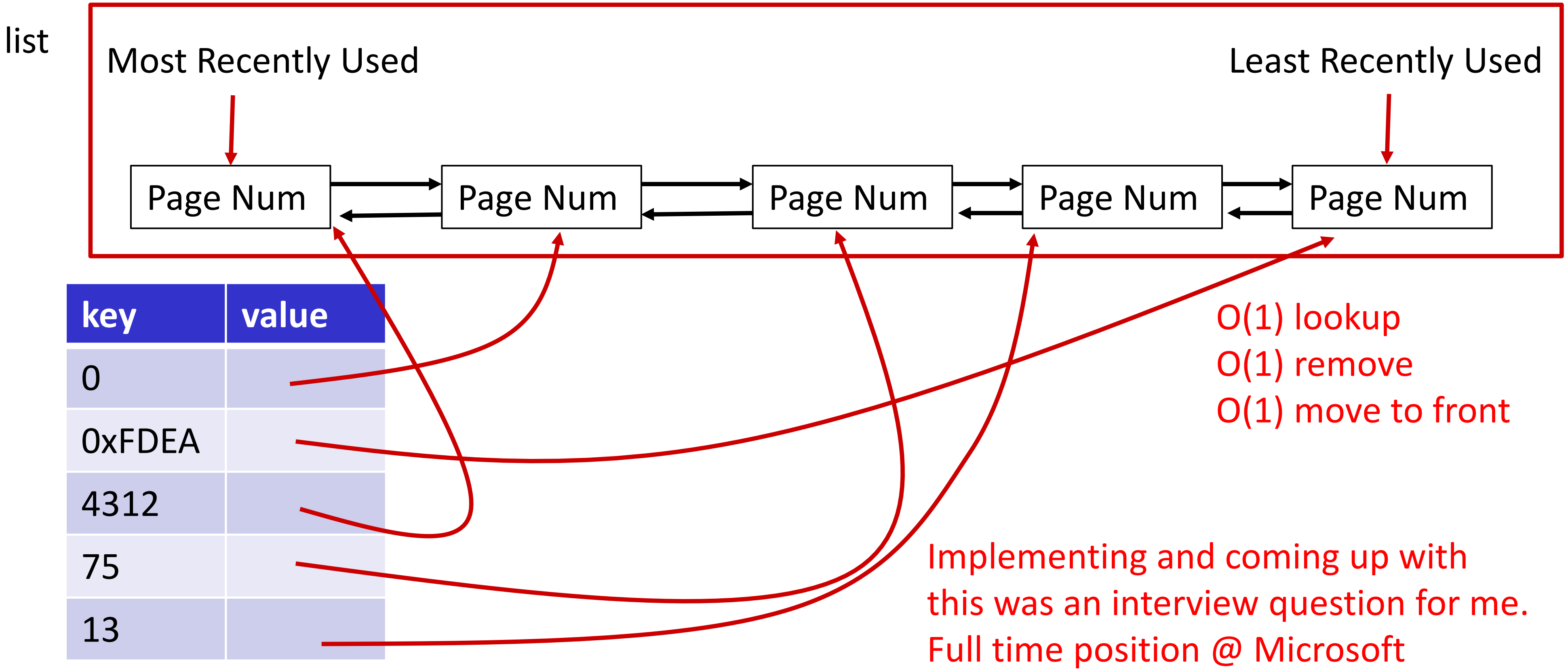
- What is the algorithmic runtime analysis to:
  - lookup a specific block? O(n)
  - Removal time? O(1)
  - Time to move a block to the front or back? O(1)

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*>`



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

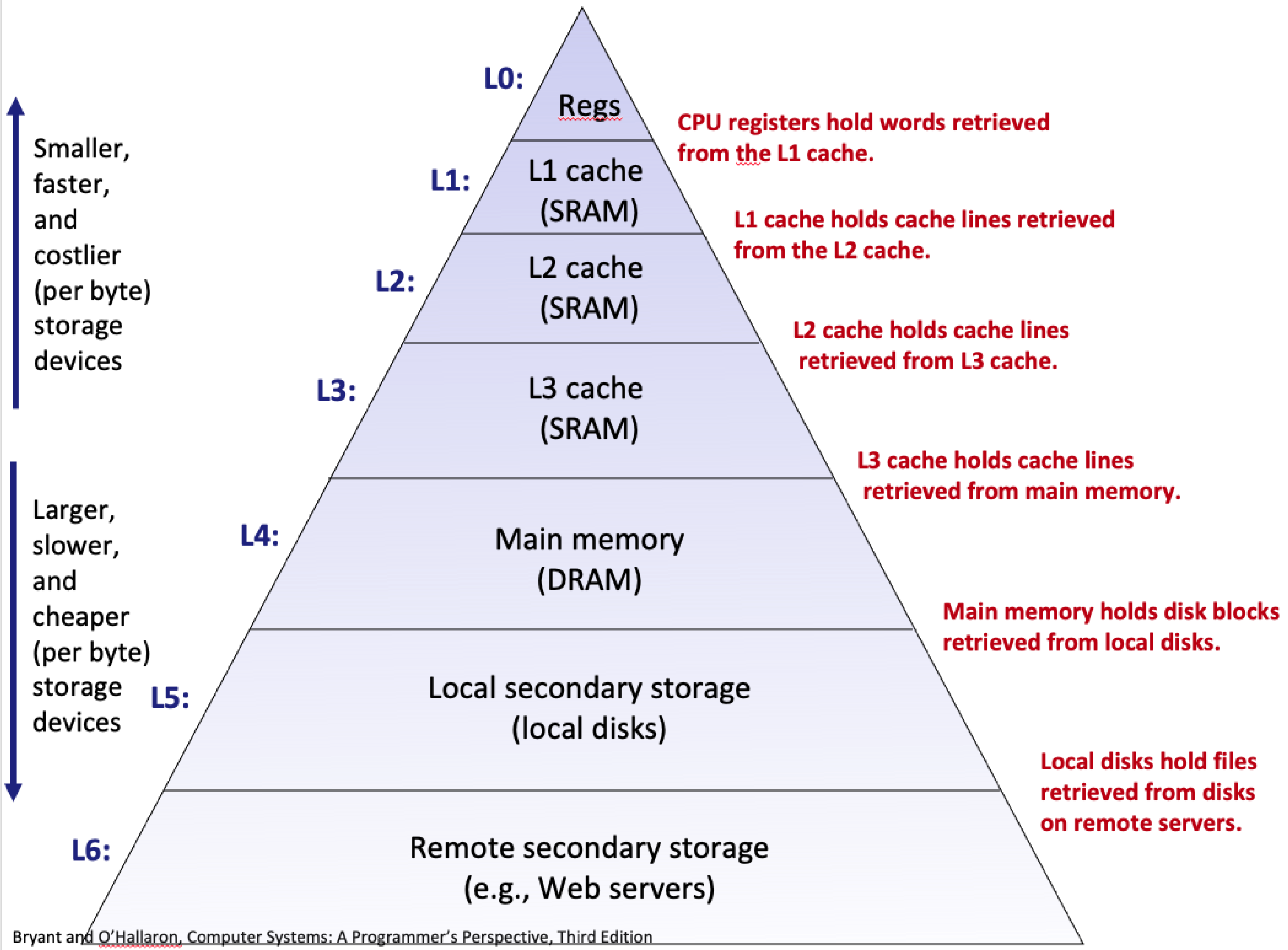
Implementing and coming up with this was an interview question for me. Full time position @ Microsoft

Poll Everywhere

pollev.com/tqm

- What happens when a program dereferences memory to a page that we haven’t accessed before?
  - CONSIDER EVERYTHING
  - What happens when we access that memory again?

Memory Hierarchy



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

Lecture Outline

- Virtual Memory
- **Threads**

Poll Everywhere

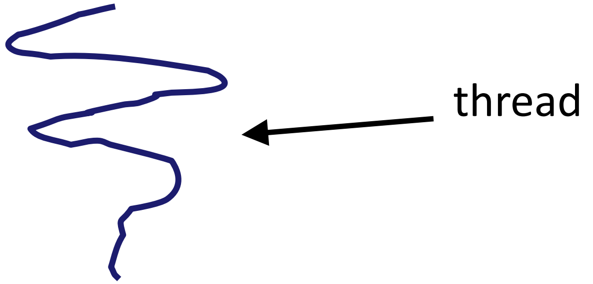
pollev.com/tqm

- What does this print?

```c
#define NUM_PROCESSES 50
#define LOOP_NUM 100

int sum_total = 0;

void loop_incr() {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
}


int main(int argc, char** argv) {
  pid_t pids[NUM_PROCESSES];  // array of process ids

  // create processes to run loop_incr()
  for (int i = 0; i < NUM_PROCESSES; i++) {
    pids[i] = fork();
    if (pids[i] == 0) {
      // child
      loop_incr();
      exit(EXIT_SUCCESS);
    }
    // parent loops and forks more children
  }

  // wait for all child processes to finish
  for (int i = 0; i < NUM_PROCESSES; i++) {
    waitpid(pids[i], NULL, 0);
  }

  printf("%d\n", sum_total);

  return EXIT_SUCCESS;
}
```

Introducing Threads

- Separate the concept of a process from the "*thread of execution*"
  - Threads are contained within a process
  - Usually called a thread, this is a sequential execution stream within a process

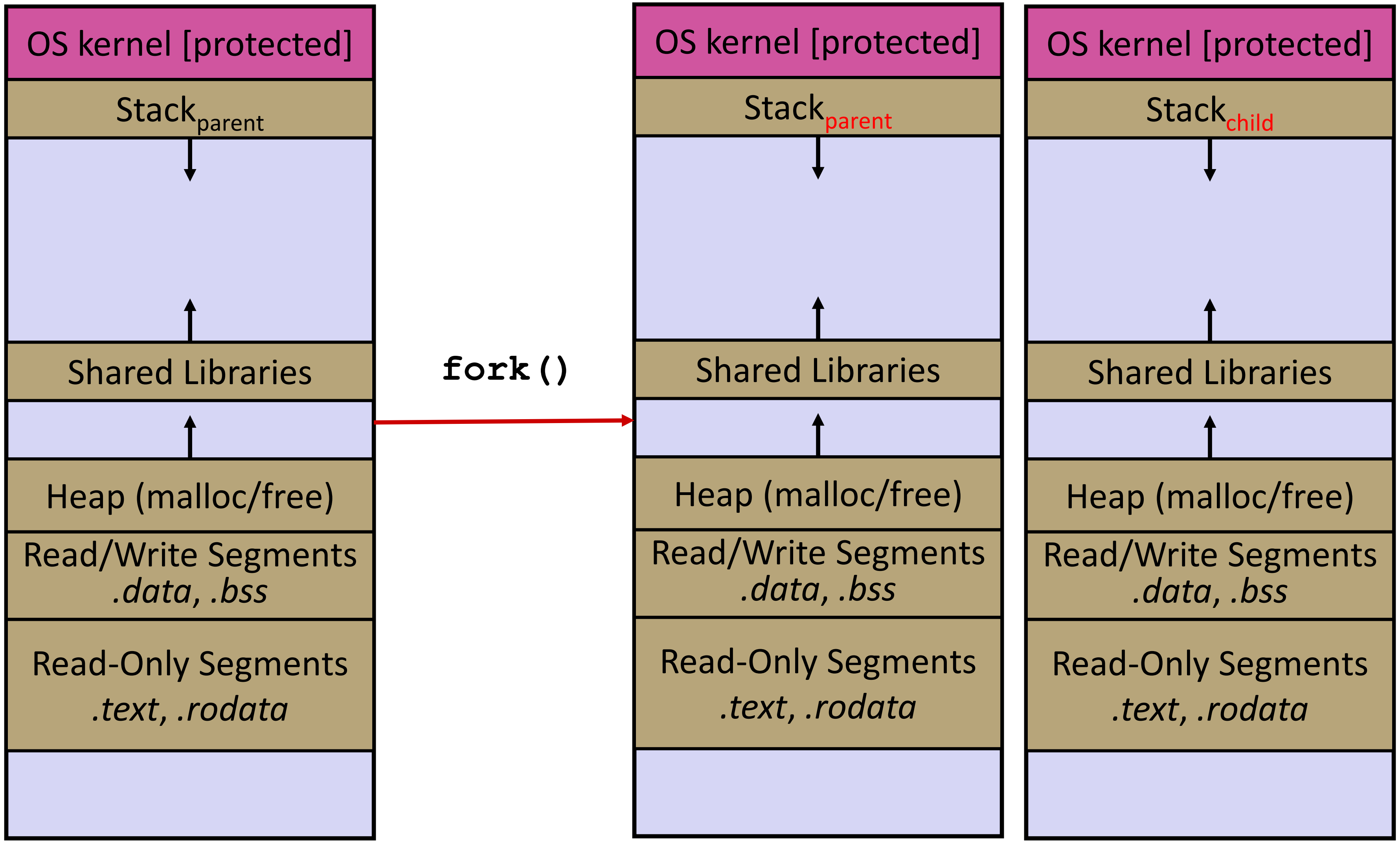


- In most modern OS's:
  - Threads are the *unit of scheduling.*

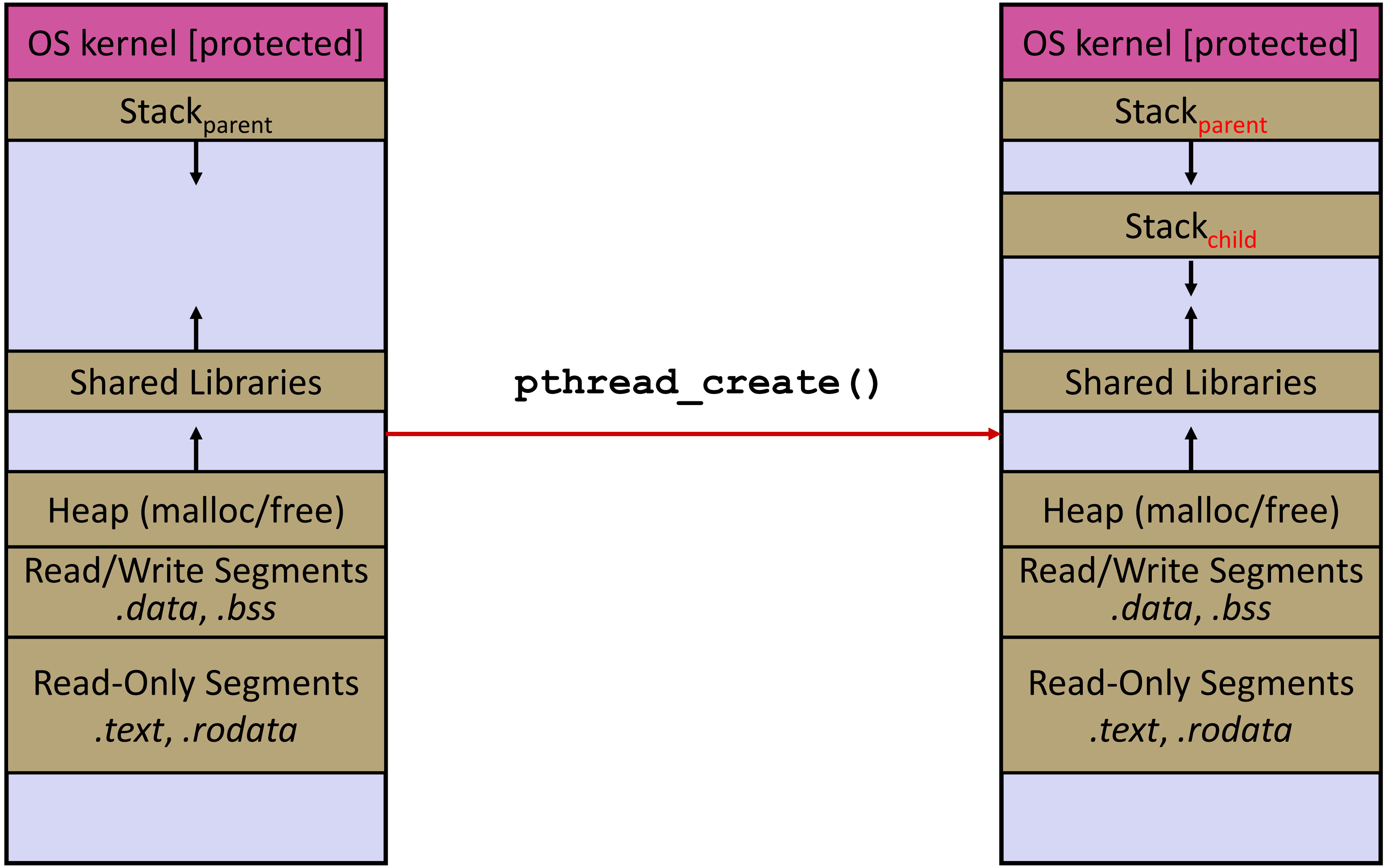
Threads vs. Processes

- In most modern OS's:
  - A <u>Process</u> has a unique: address space, OS resources, & security attributes
  - A <u>Thread</u> has a unique: stack, stack pointer, program counter, & registers
  - Threads are the *unit of scheduling* and processes are their *containers*; every process has at least one thread running in it

Threads vs. Processes



Threads vs. Processes

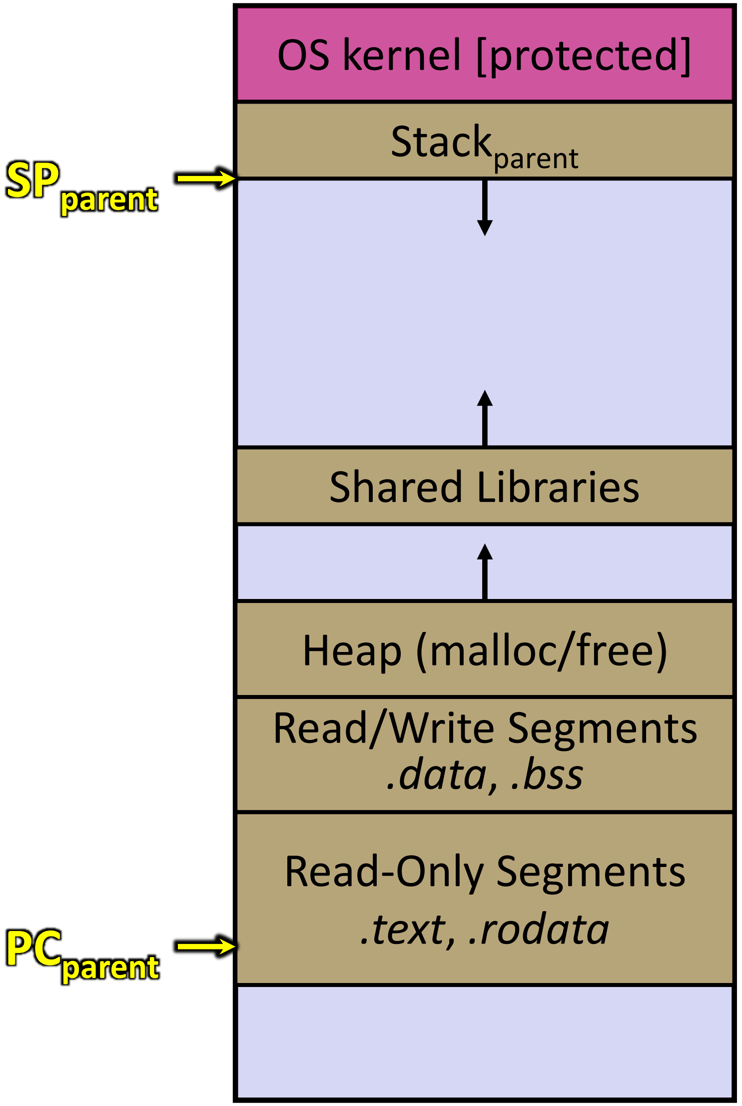


Threads

- Threads are like lightweight processes
  - They execute concurrently like processes
    - Multiple threads can run simultaneously on multiple CPUs/cores
  - Unlike processes, threads cohabitate the same address space
    - Threads within a process see the same heap and globals and can communicate with each other through variables and memory
      - But, they can interfere with each other – need synchronization for shared resources
    - Each thread has its own stack

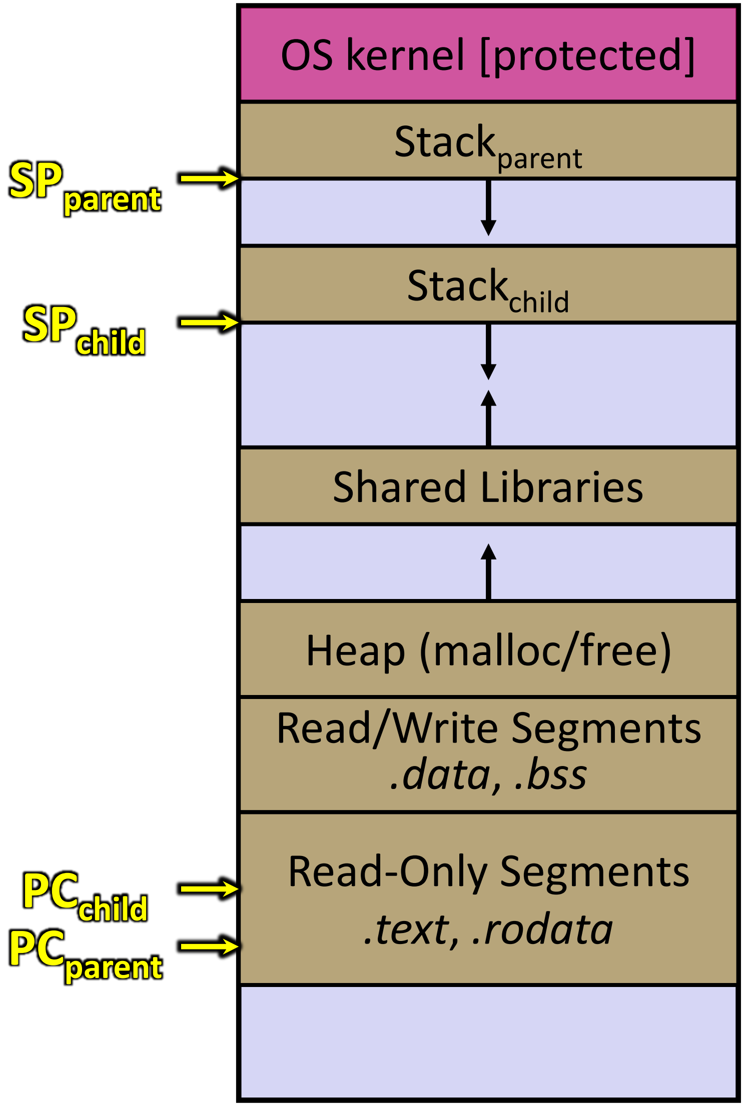
- Analogy: restaurant kitchen
  - Kitchen is process
  - Chefs are threads

Single-Threaded Address Spaces



- Before creating a thread
  - One thread of execution running in the address space
    - One PC, stack, SP
  - That main thread invokes a function to create a new thread
    - Typically `pthread_create()`

Multi-threaded Address Spaces



- After creating a thread
  - *Two* threads of execution running in the address space
    - Original thread (parent) and new thread (child)
    - New stack created for child thread
    - Child thread has its own *values* of the PC and SP
  - Both threads share the other segments (code, heap, globals)
    - They can cooperatively modify shared data

POSIX Threads (pthreads)

- The POSIX APIs for dealing with threads
  - Declared in `pthread.h`
    - Not part of the C/C++ language
  - To enable support for multithreading, must include `-pthread` flag when compiling and linking with `gcc` command
    - `g++ -g -Wall -std=c++23 -pthread -o main main.c`
  - Implemented in C
    - Must deal with C programming practices and style

Creating and Terminating Threads

Output parameter.
Gives us a "thread_descriptor"

```
int pthread_create(
        pthread_t* thread,
        const pthread_attr_t* attr,
        void* (*start_routine)(void*),
        void* arg);
```

Argument for the thread function

Function pointer!
Takes & returns void*
to allow "generics" in C

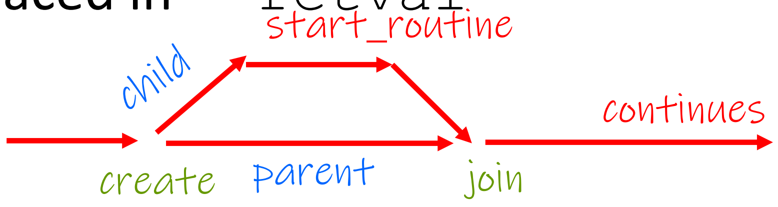
- Creates a new thread into `*thread`, with attributes `*attr` (`NULL` means default attributes)
- Returns **0** on success and an error number on error (can check against error constants)
- The new thread runs **`start_routine`**`(arg)`

child start_routine
continues
pthread_create parent

What To Do After Forking Threads?

- `int pthread_join(pthread_t thread, void** retval);`
  - Waits for the thread specified by `thread` to terminate
  - The thread equivalent of `waitpid()`
  - The exit status of the terminated thread is placed in `**retval`

Parent thread waits for child thread to exit, gets the child's return value, and child thread is cleaned up



Poll Everywhere

pollev.com/tqm

- What does this print?

```
#define NUM_THREADS 50
#define LOOP_NUM 100

int sum_total = 0;

void* thread_main(void* arg) {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  return NULL;  // return type is a pointer
}


int main(int argc, char** argv) {
  pthread_t thds[NUM_THREADS];  // array of thread ids

  // create threads to run thread_main()
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_create(&thds[i], NULL, &thread_main, NULL) != 0) {
      fprintf(stderr, "pthread_create failed\n");
    }
  }

  // wait for all child threads to finish
  // (children may terminate out of order, but cleans up in order)
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_join(thds[i], NULL) != 0) {
      fprintf(stderr, "pthread_join failed\n");
    }
  }

  printf("%d\n", sum_total);

  return EXIT_SUCCESS;
}
```

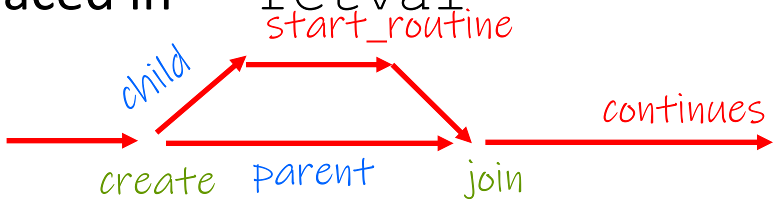
Thread Example

- See `cthreads.cpp`
  - How do you properly handle memory management?
    - Who allocates and deallocates memory?
    - How long do you want memory to stick around?

What To Do After Forking Threads?

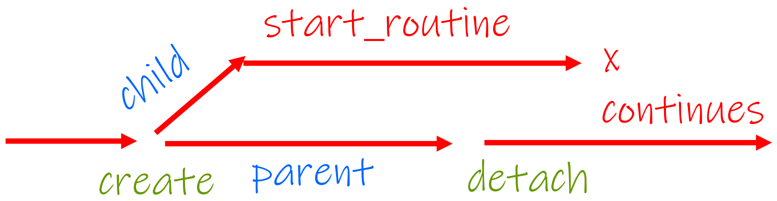
- `int pthread_join(pthread_t thread, void** retval);`
  - Waits for the thread specified by `thread` to terminate
  - The thread equivalent of **`waitpid()`**
  - The exit status of the terminated thread is placed in `**retval`

Parent thread waits for child thread to exit, gets the child's return value, and child thread is cleaned up



- `int pthread_detach(pthread_t thread);`
  - Mark thread specified by `thread` as detached – it will clean up its resources as soon as it terminates

Detach a thread.
Thread is cleaned up when it is finished



Thread Examples

- See `cthreads.cpp`
  - How do you properly handle memory management?
    - Who allocates and deallocates memory?
    - How long do you want memory to stick around?

- See `exit_thread.cpp`
  - Do we need to join every thread we create?

Poll Everywhere

pollev.com/tqm

- What gets printed?

```
void* thrd_fn(void* arg) {
  int* ptr = reinterpret_cast<int*>(arg);
  cout << *ptr << endl;
}

int main() {
  pthread_t thd1{};
  pthread_t thd2{};
  int x = 1;
  pthread_create(&thd1, nullptr, thrd_fn, &x);
  x = 2;
  pthread_create(&thd2, nullptr, thrd_fn, &x);

  pthread_join(thd1, nullptr);
  pthread_join(thd2, nullptr);
}
```