Caches & Threads

Computer Operating Systems, Spring 2025

Instructors: Joel Ramirez Travis McGaha

Head TAs: Ash Fujiyama Emily Shen Maya Huizar

TAs:

Ahmed Abdellah	Bo Sun	Joy Liu	Susan Zhang	Zihao Zhou
Akash Kaukuntla	Connor Cummings	Khush Gupta	Vedansh Goenka	
Alexander Cho	Eric Zou	Kyrie Dowling	Vivi Li	
Alicia Sun	Haoyun Qin	Rafael Sakamoto	Yousef AlRabiah	
August Fu	Jonathan Hong	Sarah Zhang	Yu Cao	

Poll Everywhere

pollev.com/tqm

- What is your favourite programming language? (And why?)

Administrivia

- PennOS
  - Groups have been assigned
  - TA's have been assigned to groups
  - You have the first milestone, which needs to be done sometime next week
  - Your group (or at least most of your group) needs to meet with your assigned TA and display the expectations laid out in the PennOS Specification
  - We will send emails to every group that had to be filled by course staff soon (let us know if you don't get this by the end of the week)

- Mid Semester Survey is Posted!
  - Due soon (tonight or tomorrow)
  - Anonymous!

Administrivia

- PennOS Advice:
  - Don’t push .o files to your repository. Add a `.gitignore` file to help reduce this issue
  - After this lecture, PennOS isn’t really talked about.
    - Recitation is focused on PennOS for the rest of the semester
    - Still may have some useful stuff, but not as directly
  - Feel free to change the makefile, you likely will have to do so

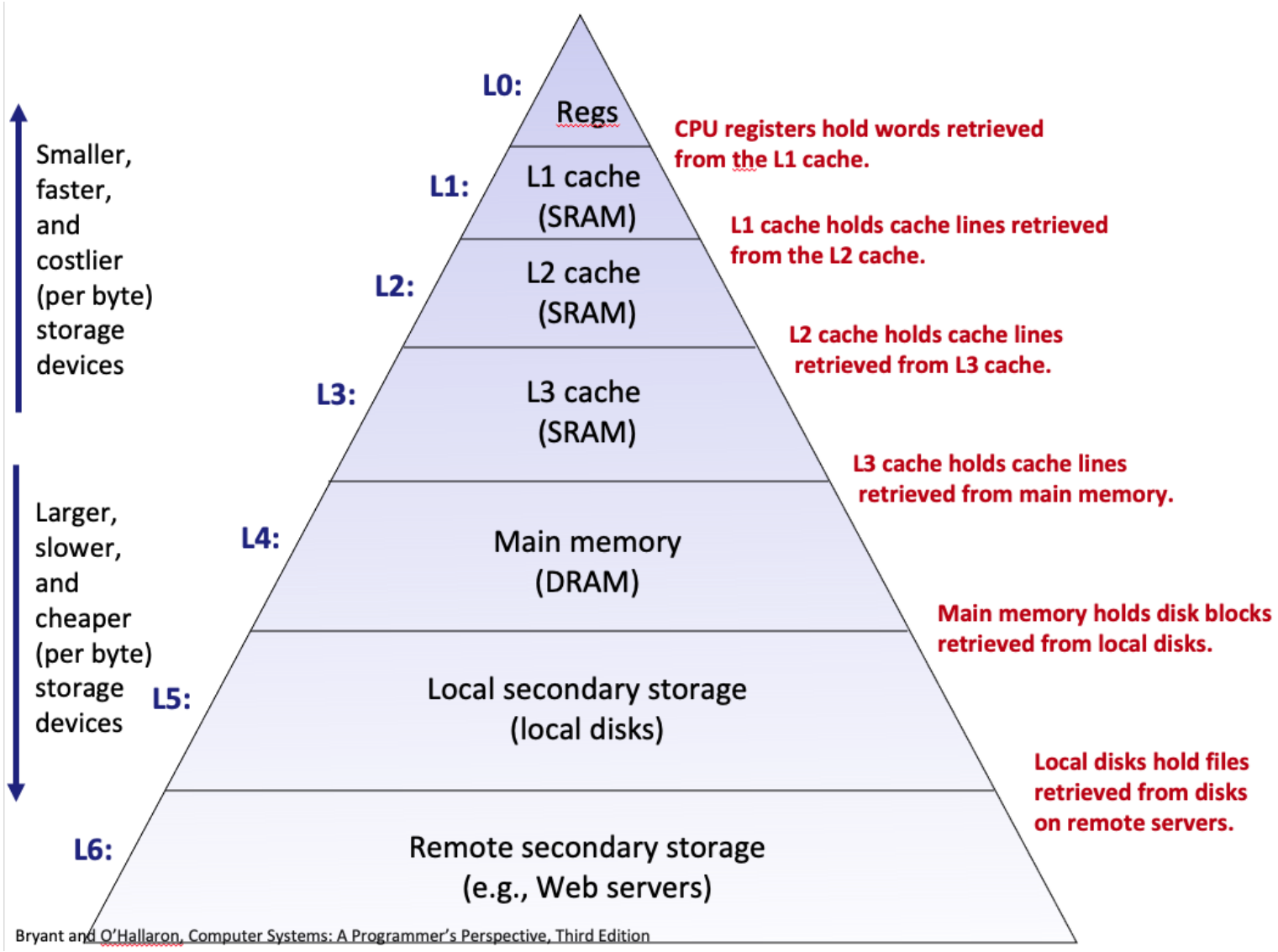
Administrivia

- TA Advice:
  - Shout out to the vs code live share extension
  - Run the yourselves on the same machine you tested the demos on
  - Writing test cases would be VERY helpful for checking your correctness
  - Organize your code into appropriate folders. This is also helpful for doxygen
    - If you have a folder that is empty for now, git won't let you push the folder unless something is in it. Create an empty file or a `.gitkeep`
  - Maybe add a command to your makefile to run doxygen?
    - At least don't put off doxygen to the very last minute
  - Don't assume that integrating the kernel and FS will be short, give your selves enough time to do it
  - Don't start too late ☺

Lecture Outline

- Caches
- Threads
  - Threads refresher
  - spthreads
  - Mutex refresher
  - tsl
  - Disable interrupts
  - Petersons

Memory Hierarchy



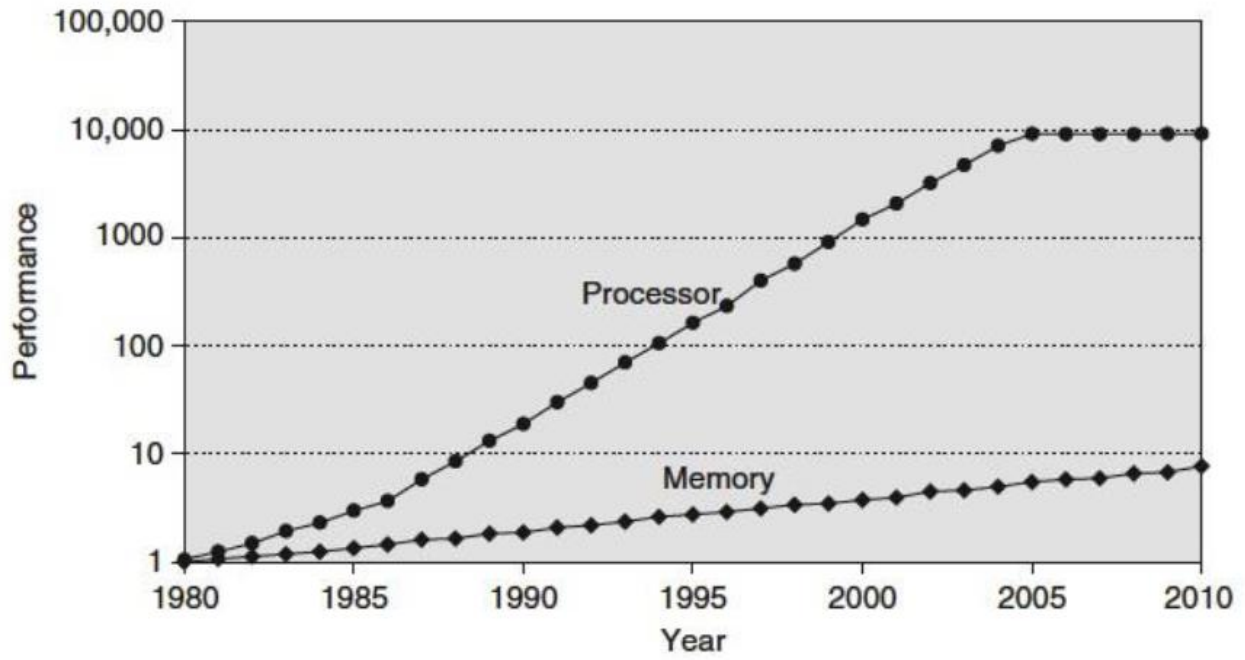
Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Memory Hierarchy so far

- So far, we know of three places where we store data
  - CPU Registers
    - Small storage size
    - Quick access time
  - Physical Memory
    - In-between registers and disk
  - Disk
    - Massive storage size
    - Long access time

- (Generally) as we go further from the CPU, storage space goes up, but access times increase

Processor Memory Gap



- Processor speed kept growing ~55% per year
- Time to access memory didn't grow as fast ~7% per year
- **Memory access would create a bottleneck on performance**
  - **It is important that data is quick to access to get better CPU utilization**

Principle of Locality

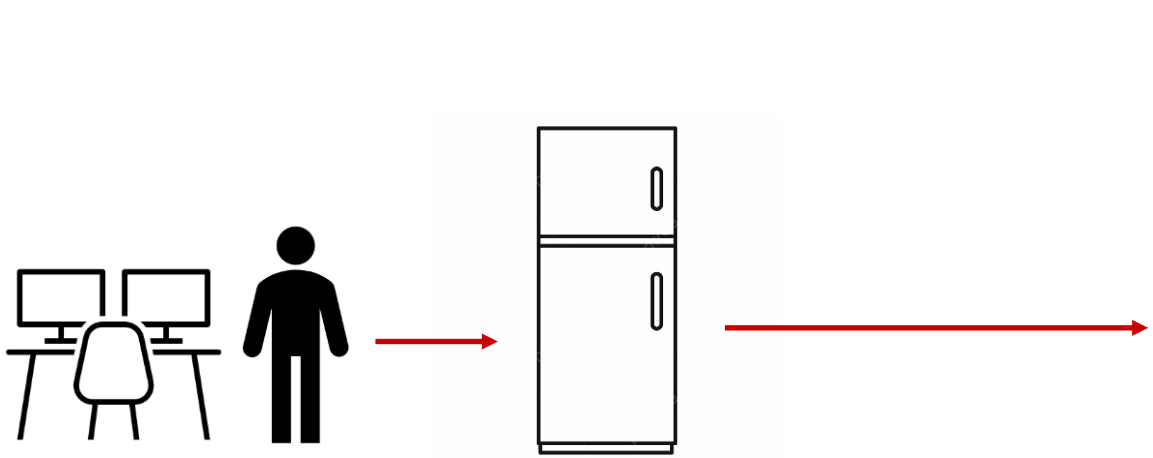
- The tendency for the Programs to access the same set of memory locations over a short period of time

- Two main types:
  - **Temporal Locality**: If we access a portion of memory, we will likely reference it again soon
  - **Spatial Locality**: If we access a portion of memory, we will likely reference memory close to it in the near future.

- Data that is accessed frequently can be stored in hardware that is quicker to access.

Locality Analogy

- If we are at home and we are hungry, where do we get food from?
  - We get it from our refrigerator!
  - If the refrigerator is empty, we go to the grocery store
  - When at the grocery store, we don't just get what we want right now, but also get other things we think we want in the near future (so that it will be in our fridge when we want it)



Cache

- Pronounced “cash”
- English: A hidden storage space for equipment, weapons, valuables, supplies, etc.
- Computer: Memory with shorter access time used for the storage of data for increased performance. Data is usually either something frequently and/or recently used.
  - Physical memory is a “Cache” of page frames which may be stored on disk. (Instead of going to disk, we can go to physical memory which is quicker to access)

Memory & Locality

Poll Everywhere

pollev.com/tqm

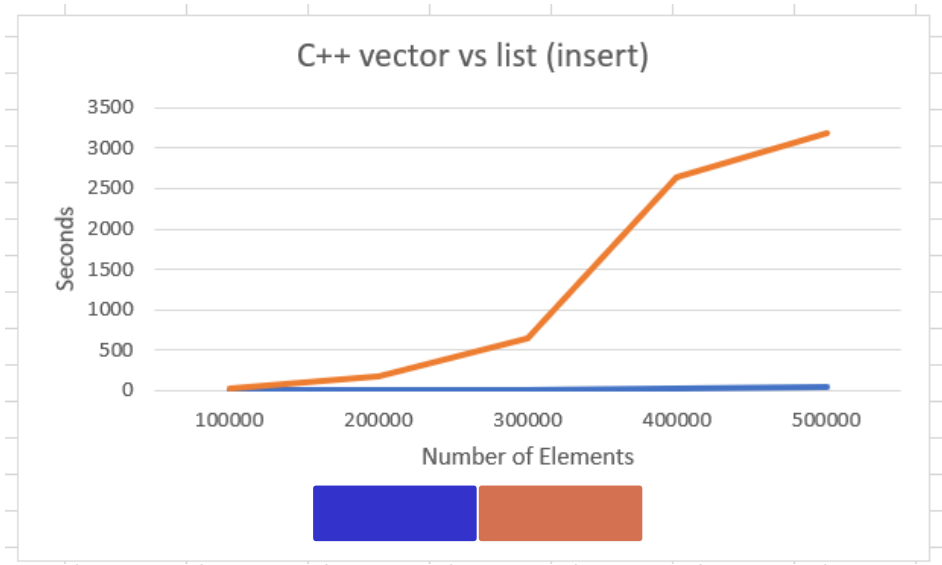
- Data Structures Review: I want to randomly generate a sequence of sorted numbers. To do this, we generate a random number and insert the number so that it remains sorted. Would a LinkedList or an ArrayList work better?
  - What if we need to use Linear search?

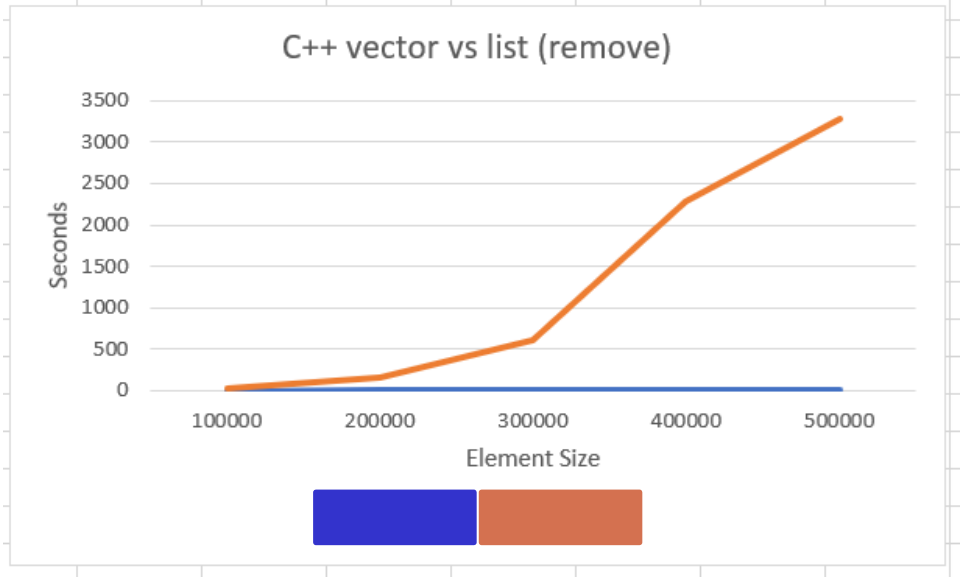
e.g. if I have sequence [5, 9, 23] and I randomly generate 12, I will insert 12 between 9 and 23

- 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?
  - What if we need to use Linear search?

Answer:

- I ran this in C++ on this laptop:
- 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?

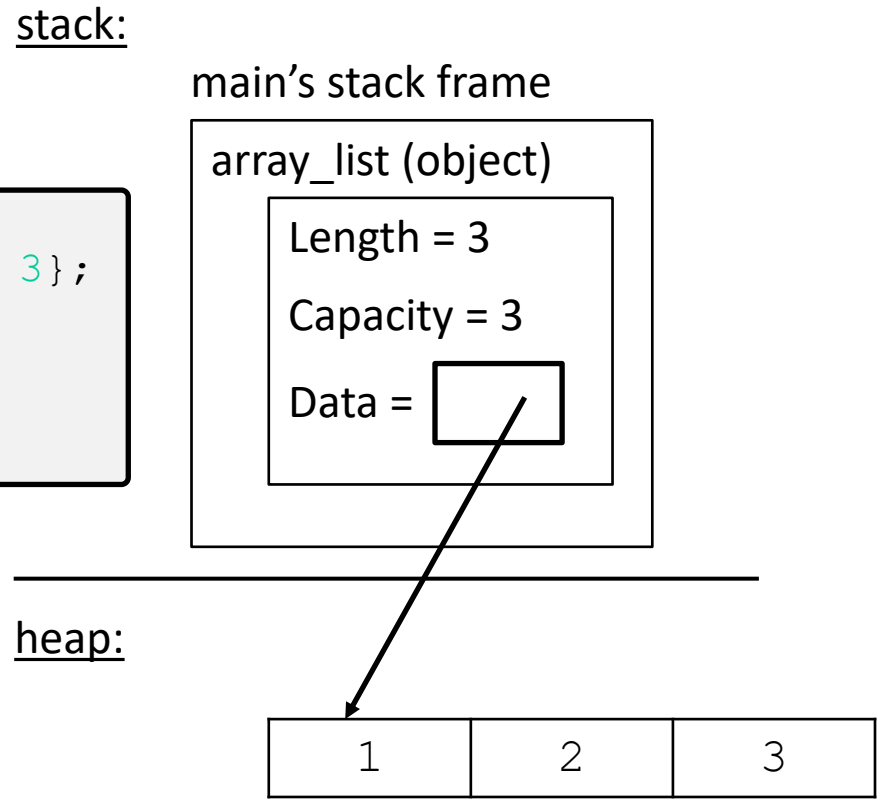
- Part 2: Let's say we take the list from part 1, randomly generate an index and remove that index from the sequence until it is empty. Would this be faster on a LinkedList or an ArrayList?

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};
  // …
}
```



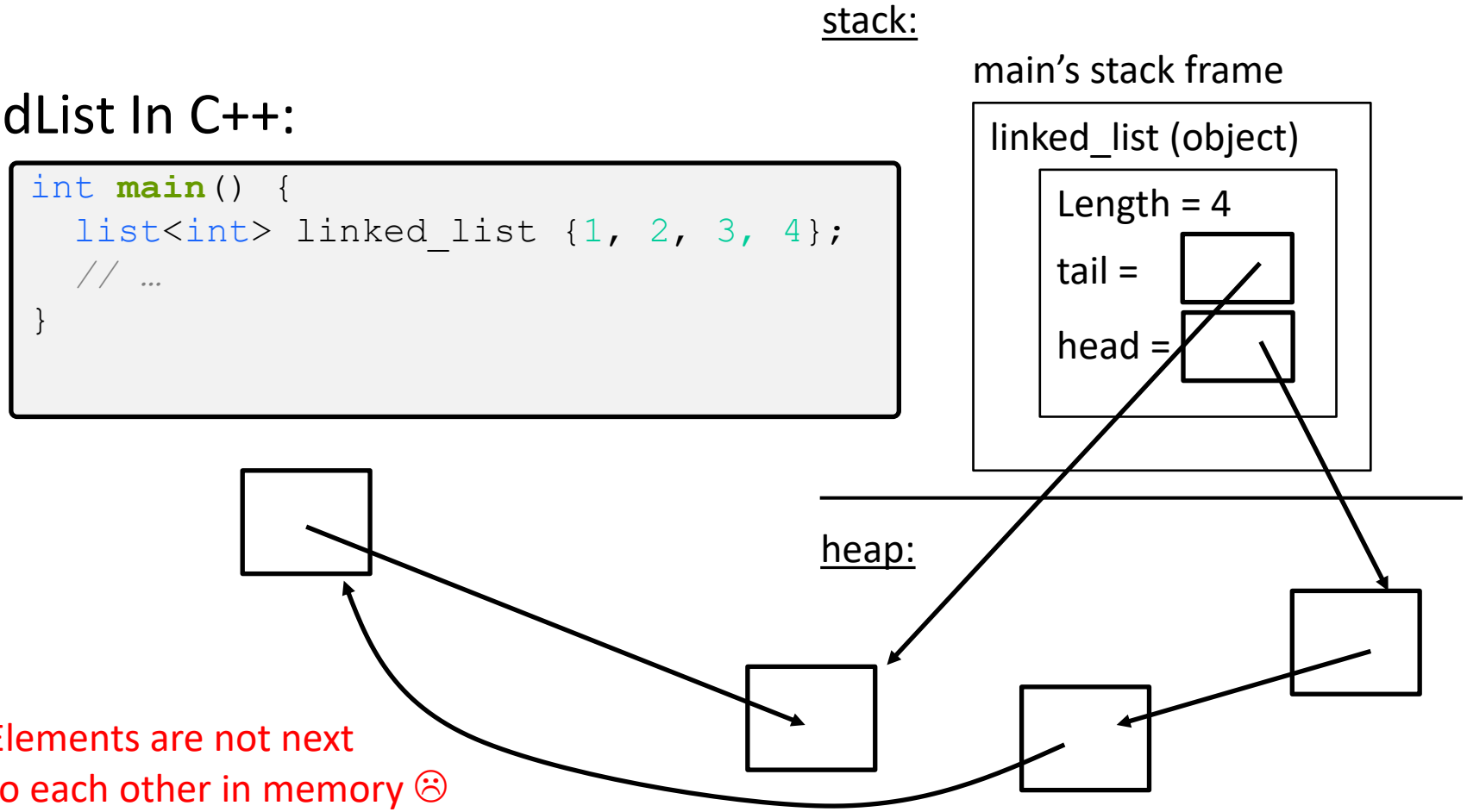
Elements are next to each other in memory ☺

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.

Cache Replacement Policy

- Caches are small and can only hold so many cache lines inside it.
- When we access data not in the cache, and the cache is full, we must evict an existing entry.
- When we access a line, we can do a quick calculation on the address to determine which entry in the cache we can store it in. (Depending on architecture, 1 to 12 possible slots in the cache)
  - Cache's typically follow an LRU (Least Recently Used) on the entries a line can be stored in

LRU (Least Recently Used)

- If a cache line is used recently, it is likely to be used again in the near future
- Use past knowledge to predict the future
- Replace the cache line that has had the longest time since it was last used

What about other languages?

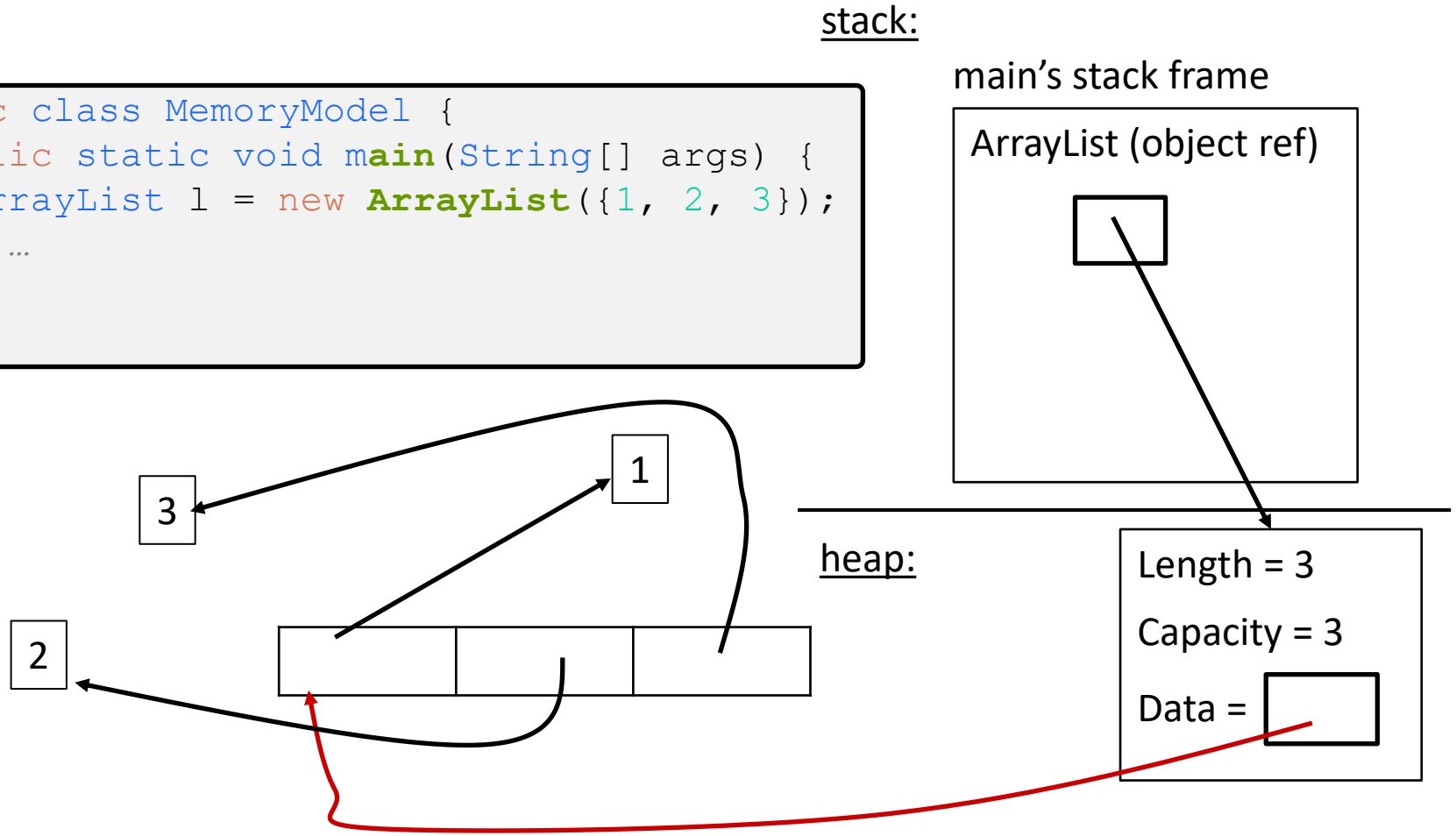
- In C++ (and C, Rust, Zig …) when you declare an object, you have an instance of that object. If you declare it as a local variable, it **exists on the stack**

- 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/tqm

- 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

pollev.com/tqm

- 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, …)

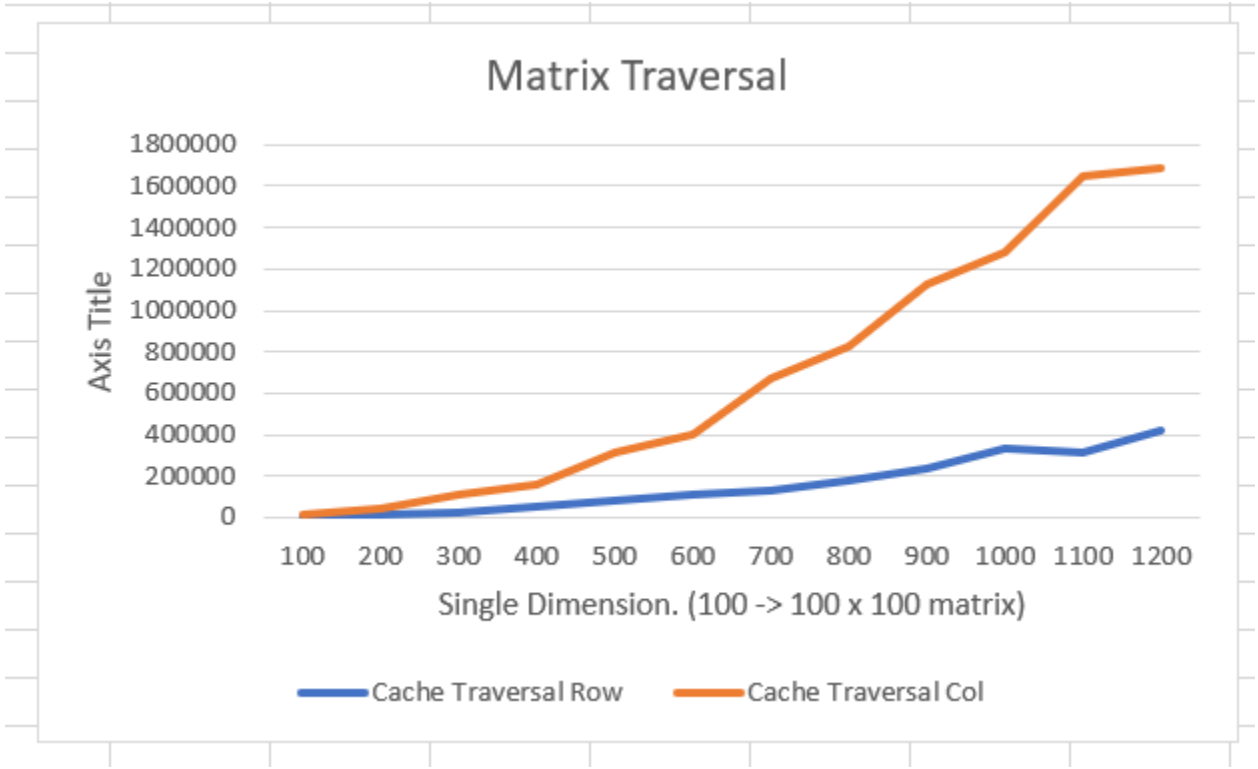
1	5	8	10
11	2	6	9
14	12	3	7
0	15	13	4

Hint: Memory Representation in C & C++

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

Experiment Results

- I ran this in C:



- Row traversal is better since it means you can take advantage of the cache

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

Numbers Everyone Should Know	
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Google

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

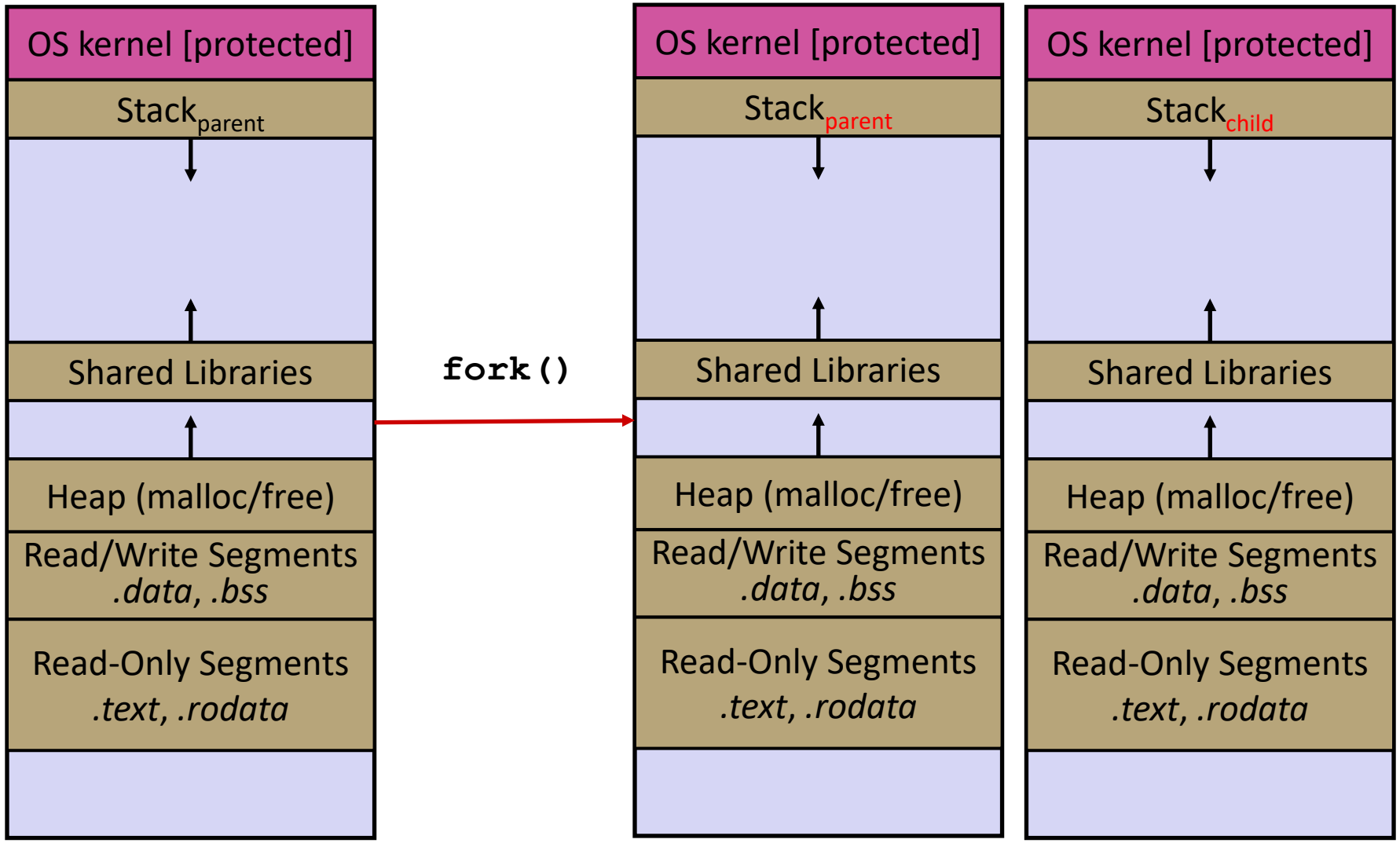
Lecture Outline

- Caches
- **Threads**
  - **Threads refresher**
  - spthreads
  - Mutex refresher
  - tsl
  - Disable interrupts
  - Petersons

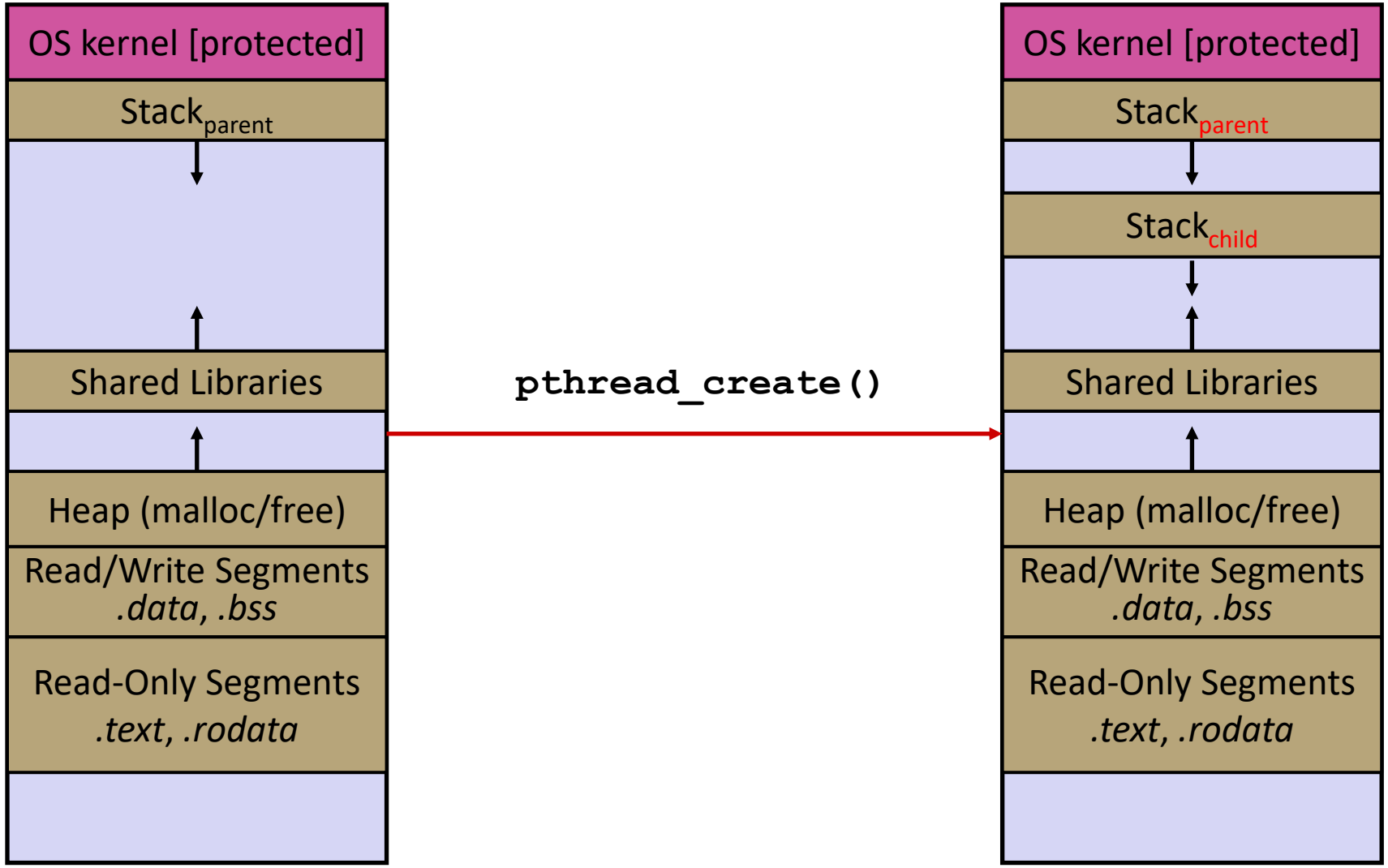
Threads vs. Processes

- In most modern OS's:
  - A Process has a unique: address space, OS resources, & security attributes
  - A Thread 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

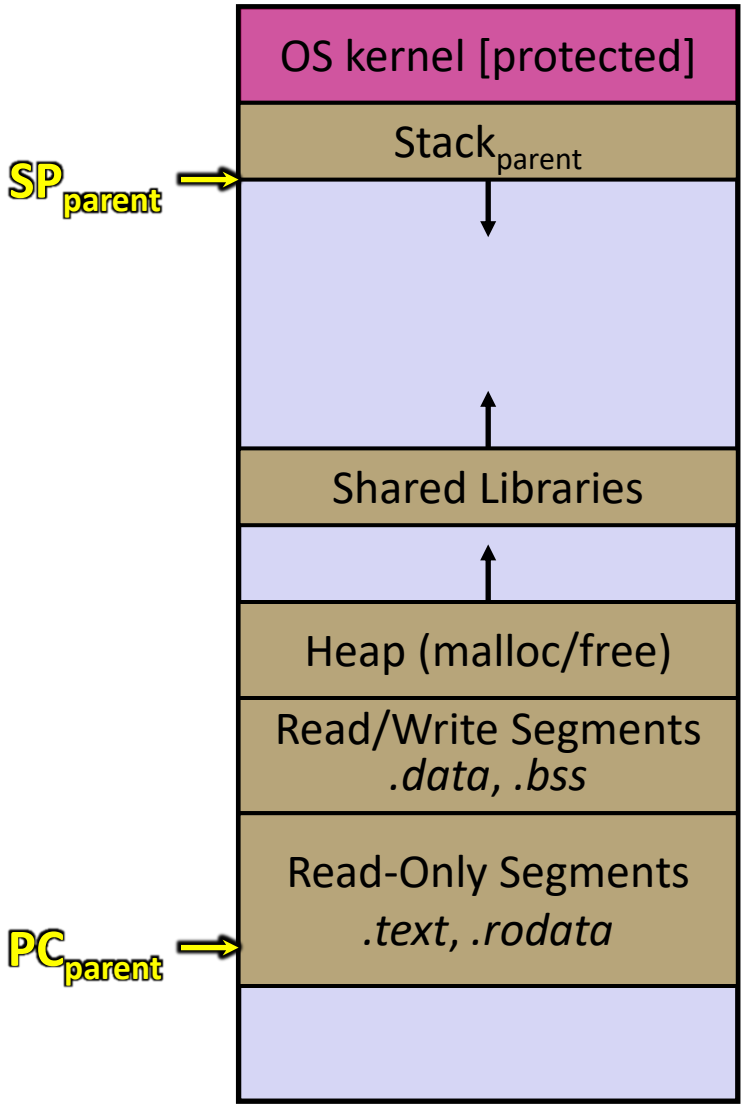
Threads vs. Processes



Threads vs. Processes

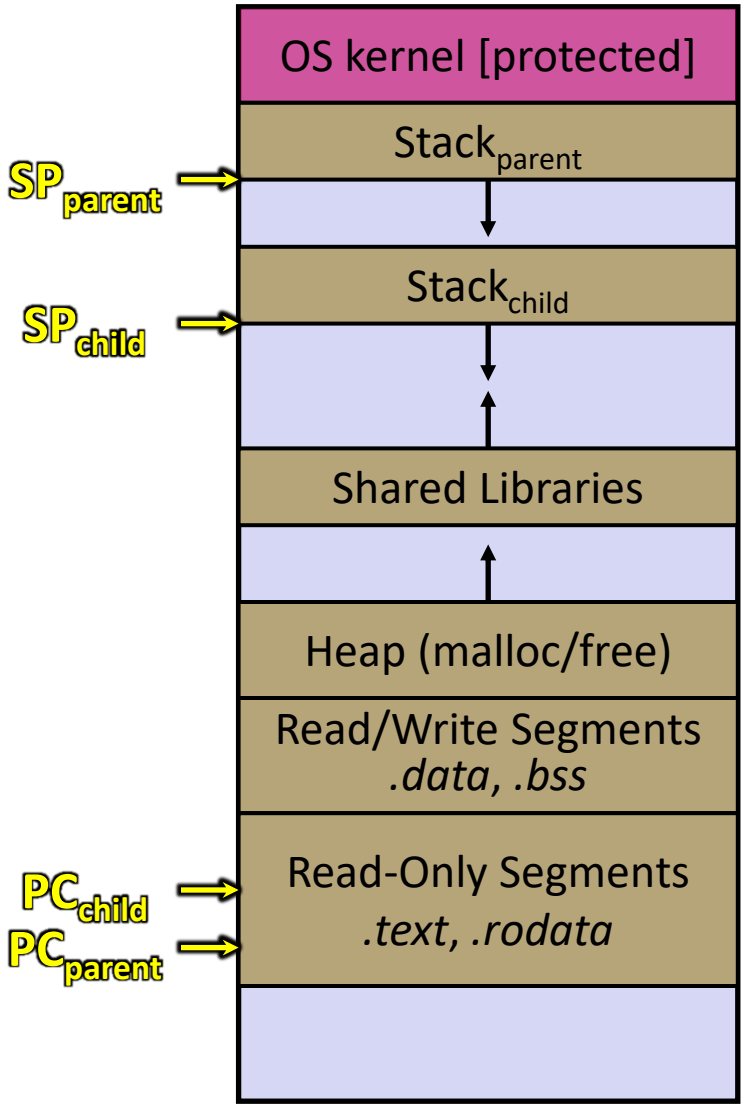


Single-Threaded Address Spaces



- 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

Poll Everywhere

pollev.com/tqm

- What are the possible outputs of this code?

```c
int global_counter = 5;

void* t_fn(void* arg) {
  int num = * (int*) arg;

  global_counter += num;

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

  free(num);
  return NULL;
}
```

```c
int main() {
  pthread_t thds[2];

  for (int i = 0; i < 2; i++) {
    pthread_t temp;
    int* arg = malloc(sizeof(int));
    *arg = i;
    pthread_create(&temp, NULL, t_fn, arg);
    thds[i] = temp;
  }

  for (int i = 0; i < 2; i++) {
    pthread_join(thds[i], NULL);
  }

  return EXIT_SUCCESS;
}
```

Lock Synchronization

- Use a "Lock" to grant access to a *critical section* so that only one thread can operate there at a time
  - Executed in an uninterruptible (*i.e.* atomic) manner

- Lock Acquire
  - Wait until the lock is free, then take it

- Lock Release
  - Release the lock
  - If other threads are waiting, wake exactly one up to pass lock to

- Pseudocode:

```
// non-critical code

lock.acquire();   <- block if locked
// critical section
lock.release();

// non-critical code
```

Lock API

- Locks are constructs that are provided by the operating system to help ensure synchronization
  - Often called a mutex or a semaphore

- Only one thread can acquire a lock at a time,
  No thread can acquire that lock until it has been released

- Has memory barriers built into it and usually uses TSL to ensure that acquiring the lock is atomic (more on TSL and memory barriers in a little bit)

Milk Example – What is the Critical Section?

- What if we use a lock on the refrigerator?
  - Probably overkill – what if roommate wanted to get eggs?
- For performance reasons, only put what is necessary in the critical section
  - Only lock the milk
  - But lock *all* steps that must run uninterrupted (*i.e.* must run as an atomic unit)

```
fridge.lock()
if (!milk) {
  buy milk
}
fridge.unlock()
```

```
milk_lock.lock()
if (!milk) {
  buy milk
}
milk_lock.unlock()
```

pthreads and Locks

- Another term for a lock is a mutex ("mutual exclusion")
  - `pthread.h` defines datatype `pthread_mutex_t`

- ```
  int pthread_mutex_init(pthread_mutex_t* mutex,
                         const pthread_mutexattr_t* attr);
  ```
  - Initializes a mutex with specified attributes
- ```
  int pthread_mutex_lock(pthread_mutex_t* mutex);
  ```
  - Acquire the lock – blocks if already locked Un-blocks when lock is acquired
- ```
  int pthread_mutex_unlock(pthread_mutex_t* mutex);
  ```
  - Releases the lock
- ```
  int pthread_mutex_destroy(pthread_mutex_t* mutex);
  ```
  - "Uninitializes" a mutex – clean up when done

pthread Mutex Examples

- See `total.c`
  - Data race between threads
- See `total_locking.c`
  - Adding a mutex fixes our data race
- How does `total_locking` compare to sequential code and to `total`?
  - Likely *slower* than both– only 1 thread can increment at a time, and must deal with checking the lock and switching between threads
  - One possible fix: each thread increments a local variable and then adds its value (once!) to the shared variable at the end
    - See `total_locking_better.c`

Lecture Outline

- Caches
- **Threads**
  - Threads refresher
  - **spthreads**
  - Mutex refresher
  - tsl
  - Disable interrupts
  - Petersons

Key Differences of spthread vs pthread

- spthread is something Travis wrote about a year ago.
  - It does not exist anywhere else
  - You likely won't find any documentation on it outside of this course

- Main difference:
  - When you create a thread, it starts "suspended"
  - Threads can be explicitly continued and suspended
  - When there is a corresponding spthread function, call that instead of the pthread function

Poll Everywhere

pollev.com/tqm

- There are issues here. What are they?

```
vector(int) vec;

void* s_fn(void* arg) {
  while(true) {
    int num = rand();
    // generate a random number
    vector_push(&vec, num);
  }
  return NULL;
}
```

```
int main() {
  vec = vector_new(int, 10, NULL);
  // initialize a length 10 vector of ints

  spthread_t thds[2];
  spthread_create(&(thds[0]), NULL, s_fn, NULL);
  spthread_create(&(thds[1]), NULL, s_fn, NULL);

  int curr_thread = 0;
  while(vector_len(&vec) < 200) {
    spthread_continue(thds[curr_thread]);
    sleep(1);  // sleep for 1 seconds
    spthread_suspend(thds[curr_thread]);

    curr_thread = 1 - curr_thread;
  }
  printf("%d\n", vector_len(&vec));
}
```

Poll Everywhere

pollev.com/tqm

- Adding a lock causes another issue, what issue is it?

```
vector(int) vec;
pthread_mutex_t lock;

void* s_fn(void* arg) {
  while(true) {
    int num = rand();
    pthread_mutex_lock(&lock);
    vector_push(&vec, num);
    pthread_mutex_unlock(&lock);
  }
  return NULL;
}
```

```
int main() {
  ...

  pthread_mutex_init(&lock, NULL);

  int curr_thread = 0;
  while(vector_len(&vec) < 200) {
    spthread_continue(thds[curr_thread]);
    sleep(1);  // sleep for 1 seconds
    spthread_suspend(thds[curr_thread]);

    curr_thread = 1 - curr_thread;
  }
  printf("%d\n", vector_len(&vec));


}
```

Shared Data & spthread

- The calls to `spthread_suspend` and `spthread_continue` will not return until that thread actually continues/suspends

- This can cause an issue when we use locks to maintain shared memory

- What do we do instead?
  - spthread_disable_interrupts_self
  - spthread_enable_interrupts_self

Lecture Outline

- Caches
- **Threads**
  - Threads refresher
  - spthreads
  - **tsl**
  - Disable interrupts
  - Petersons

TSL

- TSL stands for **T**est and **S**et **L**ock, sometimes just called **test-and-set**.

- TSL is an atomic instruction that is guaranteed to be atomic at the hardware level

- `TSL R, M`
  - Pass in a register and a memory location
  - R gets the value of M
  - M is set to 1 AFTER setting R

TSL to implement Mutex

- A mutex is pretty much this:

```
pthread_mutex_lock(lock) {
    prev_value = TSL(lock);

    // if prev_value = 1, then it was already locked
    while (prev_value == 1) {
        block();
        prev_value = TSL(lock);
    }
}

pthread_mutex_unlock(lock) {
   lock = 0;
   wakeup_blocked_threads(lock);
}
```

Lecture Outline

- Caches
- **Threads**
  - Threads refresher
  - spthreads
  - tsl
  - **Disable interrupts**
  - Petersons

Disabling Interrupts

- If data races occur when one thread is interrupted while it is accessing some shared code….

  What is we don’t switch to other threads while executing that code?

- This can be done by disabling interrupts: no interrupts means that the clock interrupt won’t go off and interrupt the currently running thread

Disabling Interrupts

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 1

Thread 0

```
disable_interrupts();
++sum_total;
enable_interrupts();
```

Thread 1

```
disable_interrupts();
++sum_total;
enable_interrupts();
```

Disabling Interrupts

- Advantages:
  - This is one way to fix this issue
- Disadvantages
  - This is usually overkill
  - This can stop threads that aren't trying to access the shared resources in the critical section. May stop threads that are executing other processes entirely
  - If interrupts disabled for a long time, then other threads will starve
  - In a multi-core environment, this gets complicated

Lecture Outline

- Caches
- **Threads**
  - Threads refresher
  - spthreads
  - tsl
  - Disable interrupts
  - **Petersons**

Poll Everywhere

pollev.com/tqm

- Lets try a more complicated software approach..
- We create two threads running thread_code, one with **`arg = 0`**, other thread has **`arg = 1`**
- Each thread tries to increment sum_total. Does this work?

```
int sum_total = 0;
bool flag[2] = {false, false};
int turn = 0

void thread_code(int arg) {
  int me = arg;

  flag[me] = true;
  turn = 1 - me;
  while((flag[1-me] == true) && (turn != me)) { }
  ++sum_total;
  flag[me] = false;
}
```

Check the index of the other thread

Peterson's Algorithm

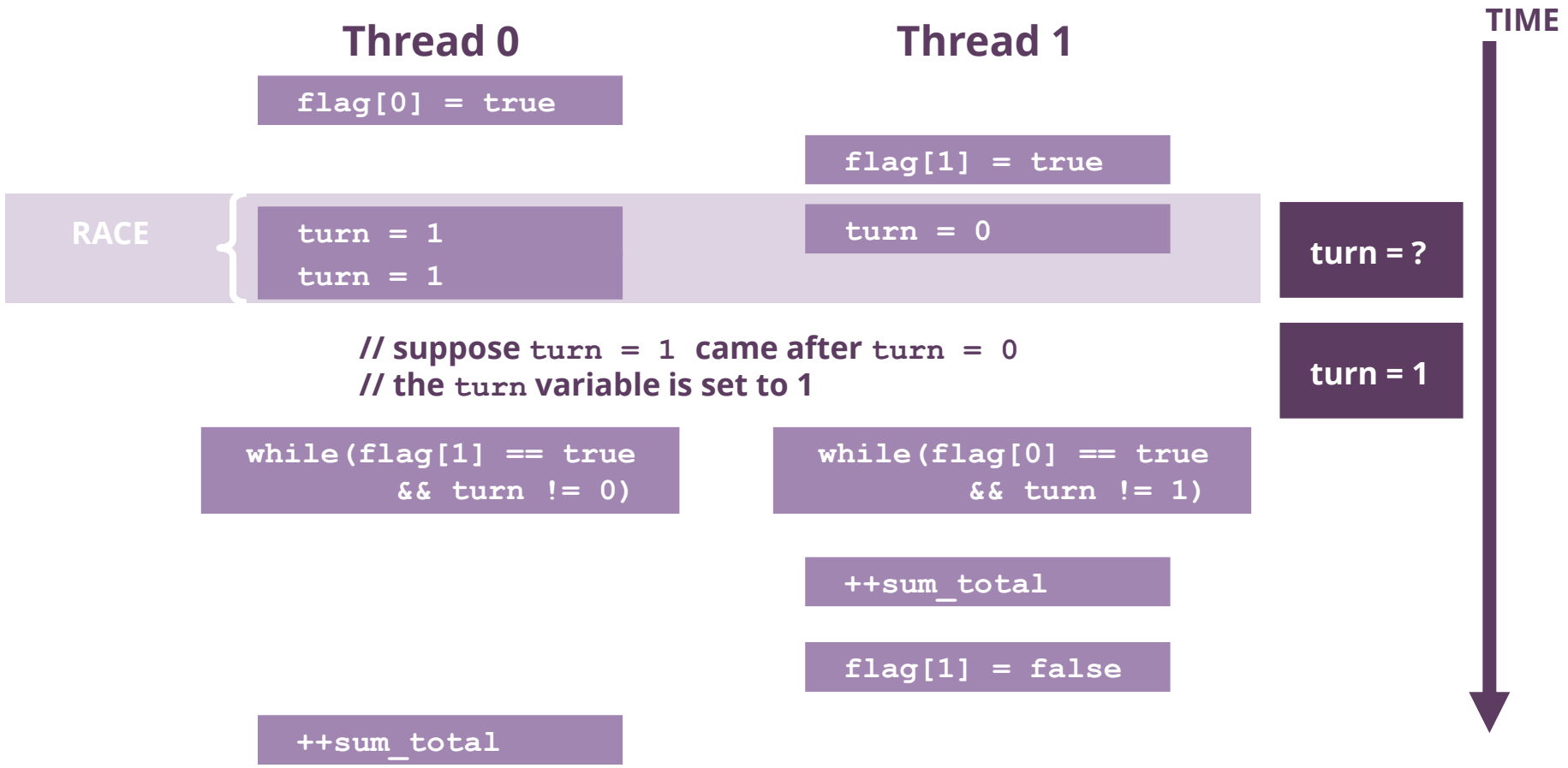
- What we just did was Peterson's algorithm
- Why does it work? (using an analogy)
  - Each thread first declares that they want to enter the critical section by setting their flag
  - Each thread then states (once) that the other should "go first".
    - This is done by setting the turn variable to 1 – me
    - One of these assignments to the turn variable will happen last, that is the one that decides who goes first
  - One of the thread goes first (decided by the value of turn) and accesses the critical section, before saying it is done (by changing their flag to false)

Peterson's Algorithm

- What we just did was Peterson's algorithm
- Why does it work?
  - Case1:
    If P0 enters critical section, flag[0] = true, turn = 0. It enters the critical section successfully.
  - Case2:
    If P0 and P1 enter critical section, flag[0] and flag[1] = true

    Race condition on turn. Suppose P0 sets turn = 0 first. Final value is turn = 1. P0 will get to run first.

Explanation



Peterson's Assumptions

- Some operations are atomic:
  - Reading from the flag and turn variables cannot be interrupted
  - Writing to the flag and turn variables cannot be interrupted
  - E.g setting turn = 1 or 0 will set turn to 0 or 1, you can be interrupted before or after, but not "during" when turn may have some intermediate value that is not 0 or 1

- That the instructions are executed in the specific order laid out in the code

Atomicity

- Atomicity: An operation or set of operations on some data are *atomic* if the operation(s) are indivisible, that no other operation(s) on that same data can interrupt/interfere.

- Aside on terminology:
  - Often interchangeable with the term “Linearizability”
  - Atomic has a different (but similar-ish) meaning in the context of data bases and ACID.

Aside: Instruction & Memory Ordering

- Do we know that **`t`** is set before **`g`** is set?

```
bool g = false;
int t = 0

void some_func(int arg) {
  t = arg;
  g = true;
}
```

Aside: Instruction & Memory Ordering

- Do we know that **t** is set before **g** is set?

NO

```
bool g = false;
int t = 0

void some_func(int arg) {
  t = arg;
  g = true;
}
```

The compiler may generate instructions that sets g first and then t
The Processor may execute these out of order or at the same time

Why? Optimizations on program performance

You can be guaranteed that `t` and `g` are set before `some_func` returns

Aside: Instruction & Memory Ordering

- The compiler may generate instructions with different ordering if it does not appear that it will affect the semantics of the function
  - Since `g = true;` is not affected by `t = arg;` then either one could execute first.

- The Processor may also execute these in a different order than what the compiler says

- Why? Optimizations on program performance
  - If you want to know more, look into “Out-of-Order Execution” and “Memory Order”

Aside: Memory Barriers

- How do we fix this?

- We can emit special instructions to the CPU and/or compiler to create a "memory barrier"
  - "all memory accesses before the barrier are guaranteed to happen before the memory accesses that come after the barrier"

  - A way to enforce an order in which memory accesses are ordered by the compiler and the CPU

  - This is done for us when we mark a variable as atomic or use a lock.