Condition Variables & Caches

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

Administrivia

- HW1 was due this Friday
  - Already out
  - Everything you need has been covered

- HW2 to be released soon
  - Due after break
  - We expect you to look at it and try some of it (maybe implement one of the threading components) before the exam
  - We do not expect you to work on it over break

- Travis still recovering from a stomach virus :(

Administrivia

- Midterm Exam: Wednesday February 28th 7-9 pm in Towne 100
  - Please contact Travis if you cannot make it at that time

pollev.com/tqm

- Any questions?

Lecture Outline

- **Condition Variables**
- Intro to Caches

```
 8 pthread_mutex_t lock;
 9 bool print_ok = false;
10
11 void* producer_thread(void* arg) {
12   pthread_mutex_lock(&lock);
13   print_ok = true;
14   pthread_mutex_unlock(&lock);
15   pthread_exit(nullptr);
16 }
17
18 void* consumer_thread(void* arg) {
19   pthread_mutex_lock(&lock);
20   cout << "print_ok is ";
21   if (print_ok) {
22     cout << "true";
23   } else {
24     cout << "false";
25   }
26   cout << endl;
27   pthread_mutex_unlock(&lock);
28   pthread_exit(nullptr);
29 }
30
31 int main(int argc, char** argv) {
32   pthread_t thd1, thd2;
33   pthread_mutex_init(&lock, nullptr);
34
35   pthread_create(&thd1, nullptr, producer_thread, nullptr);
36   pthread_create(&thd2, nullptr, consumer_thread, nullptr);
37
38   pthread_join(thd1, nullptr);
39   pthread_join(thd2, nullptr);
40
41   pthread_mutex_destroy(&lock);
42   return EXIT_SUCCESS;
43 }
```

- Poll from last lecture
  - race.cpp
- This code doesn't have a data-race, but it still has a synchronization issue (via a race condition)

Race Condition vs Data Race

- Data-Race: when there are concurrent accesses to a shared resource, with at least one write, that can cause the shared resource to enter an invalid or “unexpected” state.

- Race-Condition: Where the program has different behaviour depending on the ordering of concurrent threads. This can happen even if all accesses to shared resources are “atomic” or “locked”

- The previous example has no data-race, but it does have a race condition

Aside: sleep()

- `unistd.h` defines the function:

```
unsigned int sleep(unsigned int seconds);
```

  - Makes the calling thread sleep for the specified number of seconds, resuming execution afterwards

- Useful for manipulating scheduling for testing and demonstration purposes
  - Also for asynchronous/non-blocking I/O, but not covered in this course.

- Necessary for HW2 so that auto-graders work ☹

Thread Communication

- Sometimes threads may need to communicate with each other to know when they can perform operations

- Example: Producer and consumer threads
  - One thread creates tasks/data
  - One thread consumes the produced tasks/data to perform some operation
  - The consumer thread can only produce things once the producer has produced them

Naïve Solution

- Consider the example where a thread must wait to be notified before it can print something out and terminate

- Possible solution: "Spinning"
  - Infinitely loop until the producer thread notifies that the consumer thread can print

- See `spinning.cpp`

- Alternative: Condition variables

Condition Variables

- Variables that allow for a thread to wait until they are notified to resume
- Avoids waiting clock cycles “spinning”
- Done in the context of mutual exclusion
  - a thread must already have a lock, which it will temporarily release while waiting
  - Once notified, the thread will re-acquire a lock and resume execution

pthreads and condition variables

- `pthread.h` defines datatype `pthread_cond_t`

- ```
  int pthread_cond_init(pthread_cond_t* cond,
                        const pthread_condattr_t* attr);
  ```
  - Initializes a condition variable with specified attributes
- ```
  int pthread_cond_destroy(pthread_cond_t* cond);
  ```
  - “Uninitializes” a condition variable – clean up when done

pthreads and condition variables

- `pthread.h` defines datatype `pthread_cond_t`

- ```
  int pthread_cond_wait(pthread_cond_t* cond,
                        pthread_mutex_t* mutex);
  ```
  - Atomically releases the mutex and blocks on the condition variable. Once unblocked (by one of the functions below), function will return and calling thread will have the mutex locked

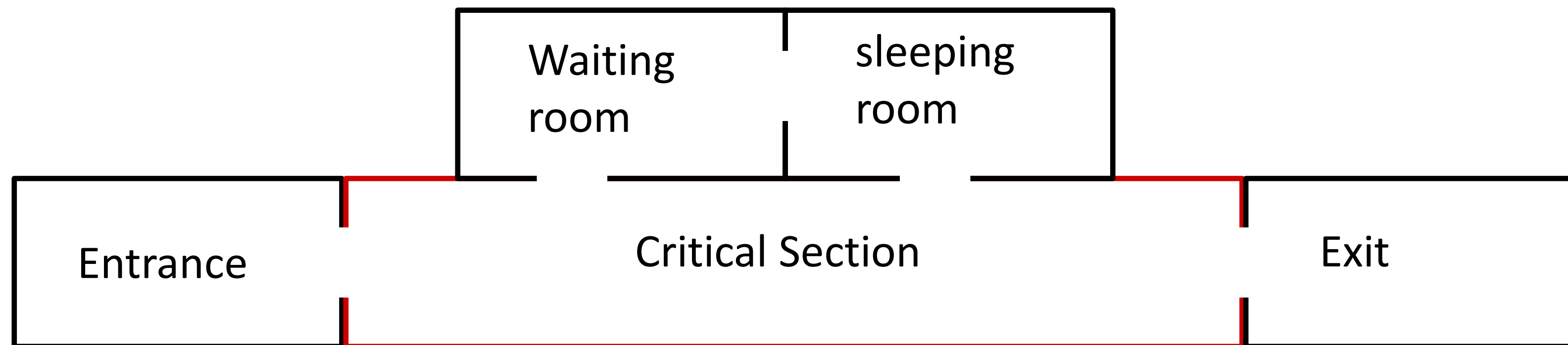
- ```
  int pthread_cond_signal(pthread_cond_t* cond);
  ```
  - Unblock at least one of the threads on the specified condition

- ```
  int pthread_cond_broadcast(pthread_cond_t* cond);
  ```
  - Unblock all threads blocked on the specified condition

- See `cond.cpp`
  ```

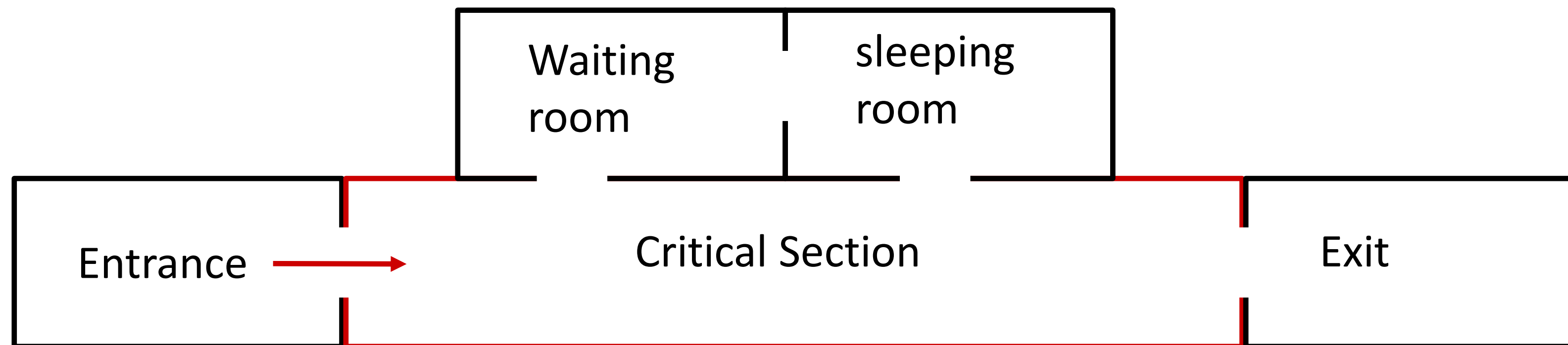
Condition Variable & Mutex Visualization

- This is to visualize how we are using condition variables in this example



Condition Variable & Mutex Visualization

- This is to visualize how we are using condition variables in this example

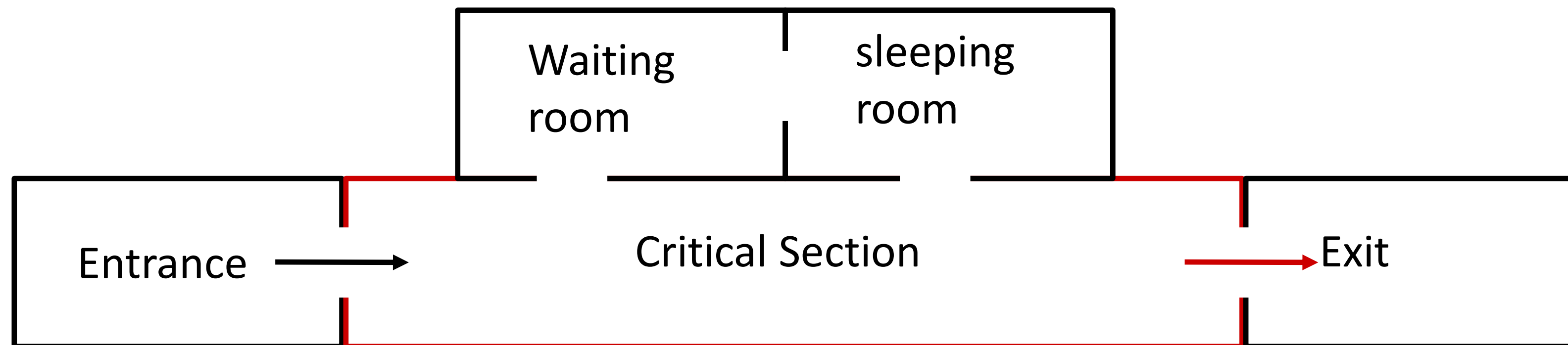


pthread_mutex_lock

A thread enters the critical section by acquiring a lock

Condition Variable & Mutex Visualization

- This is to visualize how we are using condition variables in this example



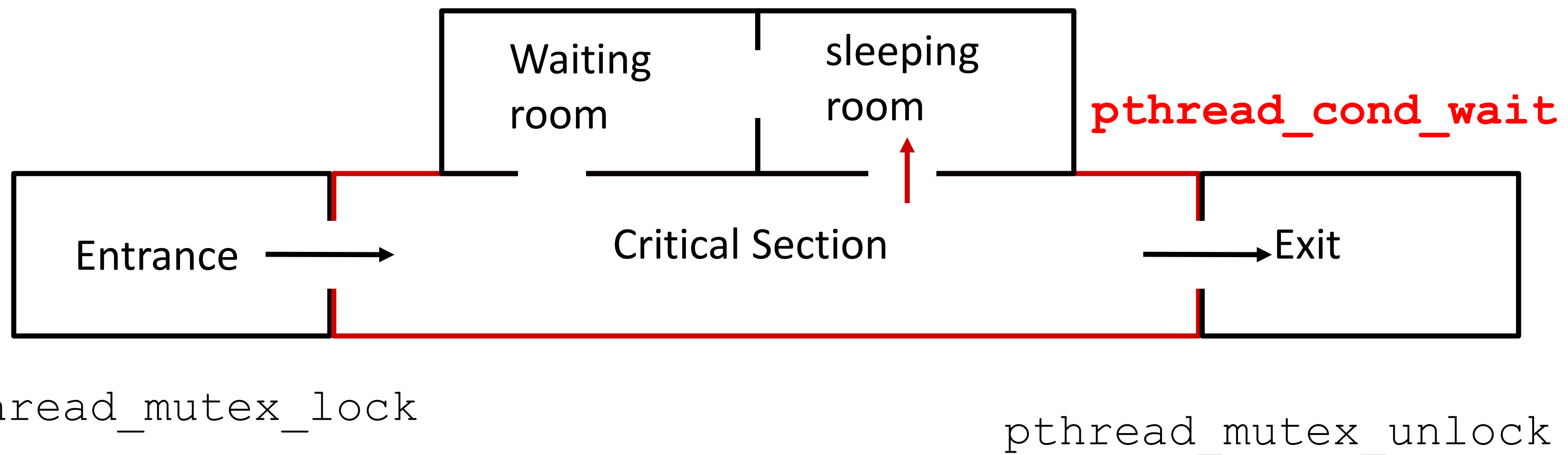
`pthread_mutex_lock`

`pthread_mutex_unlock`

A thread can exit the critical section by acquiring a lock

Condition Variable & Mutex Visualization

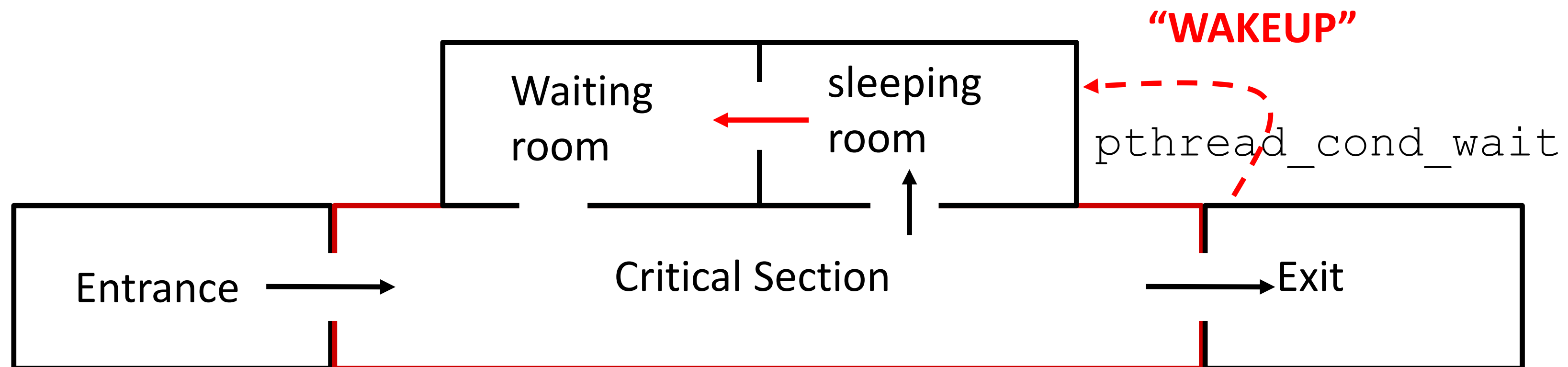
- This is to visualize how we are using condition variables in this example



If a thread can't complete its action, or must wait for some change in state, it can "go to sleep" until someone wakes it up later.
It will release the lock implicitly when it goes to sleep

Condition Variable & Mutex Visualization

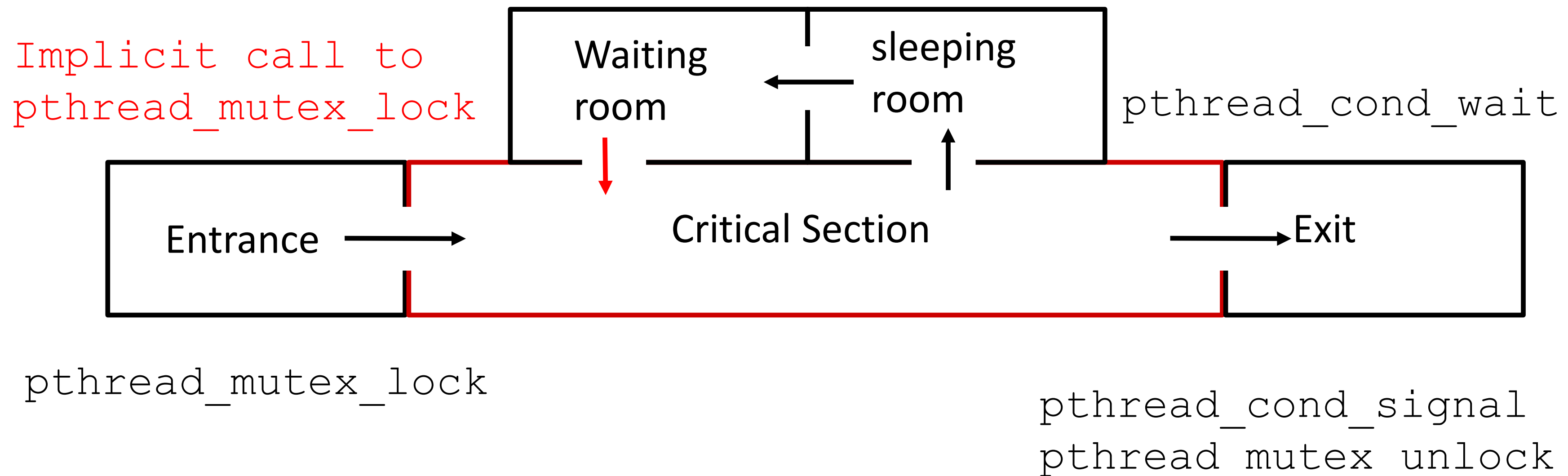
- This is to visualize how we are using condition variables in this example



When a thread modifies state and then leaves the critical section, it can also call pthread_cond_signal to wake up threads sleeping on that condition variable

Condition Variable & Mutex Visualization

- This is to visualize how we are using condition variables in this example



One or more sleeping threads wake up and attempt to acquire the lock.
Like a normal call to pthread_mutex_lock the thread will block until it can acquire the lock

Aside: Things left out

- MANY things left out of this lecture

- Synchronization methods:
  - Semaphores
  - Monitors

- Concurrency properties
  - ACID (databases)
  - CAP theorem

- A lot more concurrency stuff covered in CIS 5050 ☺

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?

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

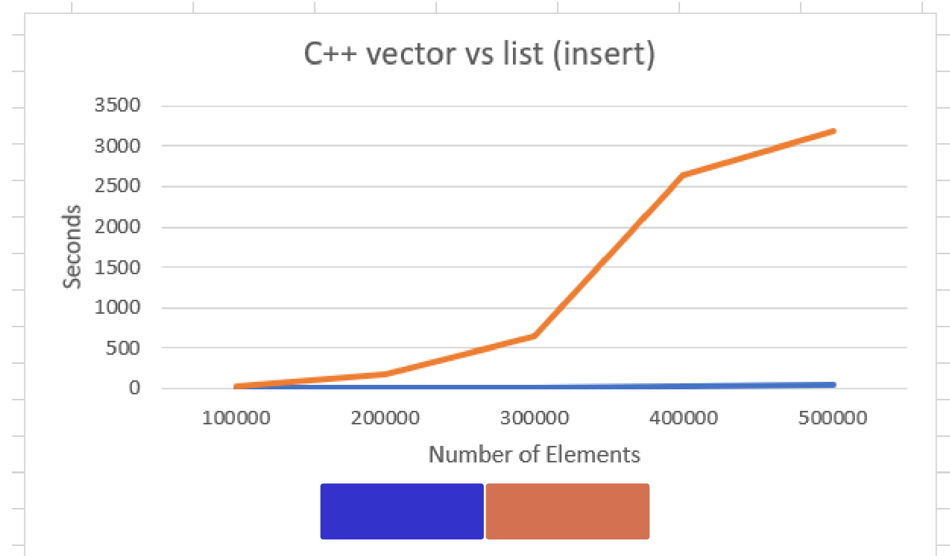
- Part 2: Let's say we take the list from part 1, randomly generate an index and remove that index from the sequence until it is empty. Would this be faster on a LinkedList or an ArrayList?

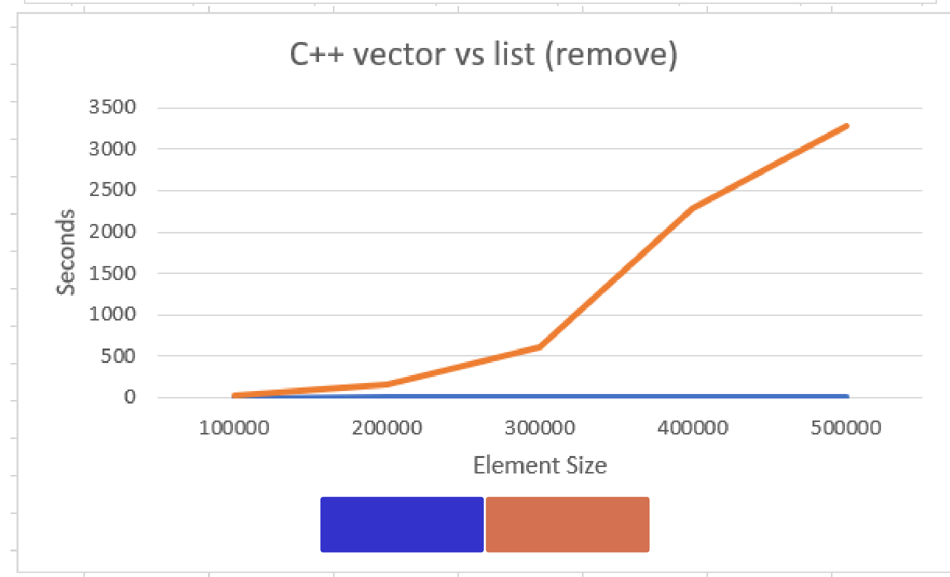
Lecture Outline

- Condition Variables
- **Intro to Caches**

Answer:

- I ran this in C++ on this laptop:
- Terminology
  - Vector == ArrayList
  - List == LinkedList
- On Element size from 100,000 -> 500,000



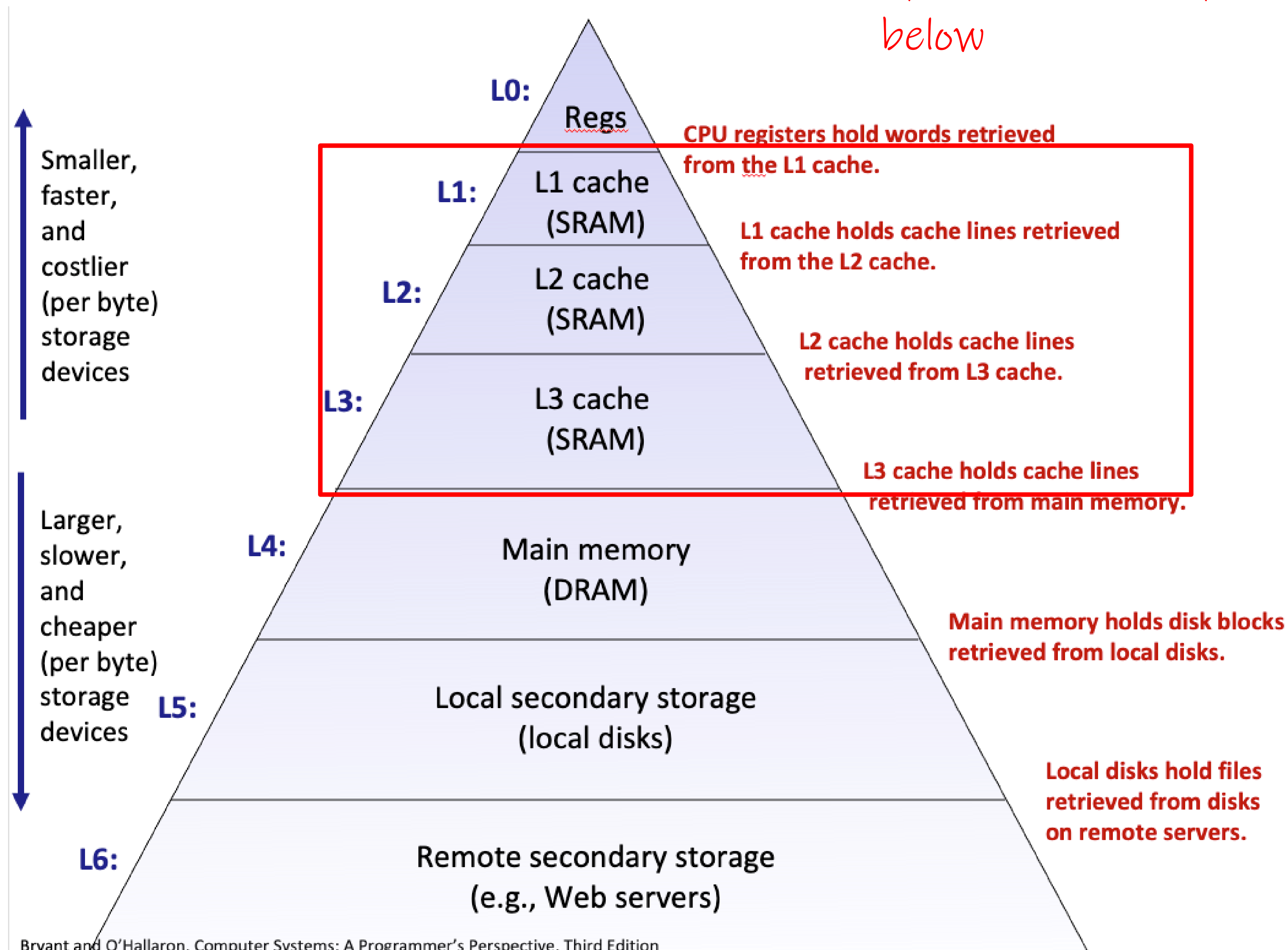


Data Access Time

- Data is stored on a physical piece of hardware
- The distance data must travel on hardware affects how long it takes for that data to be processed
- Example: data stored closer to the CPU is quicker to access
  - We see this already with registers. Data in registers is stored on the chip and is faster to access than registers

Memory Hierarchy

Each layer can be thought of as a "cache" of the layer below



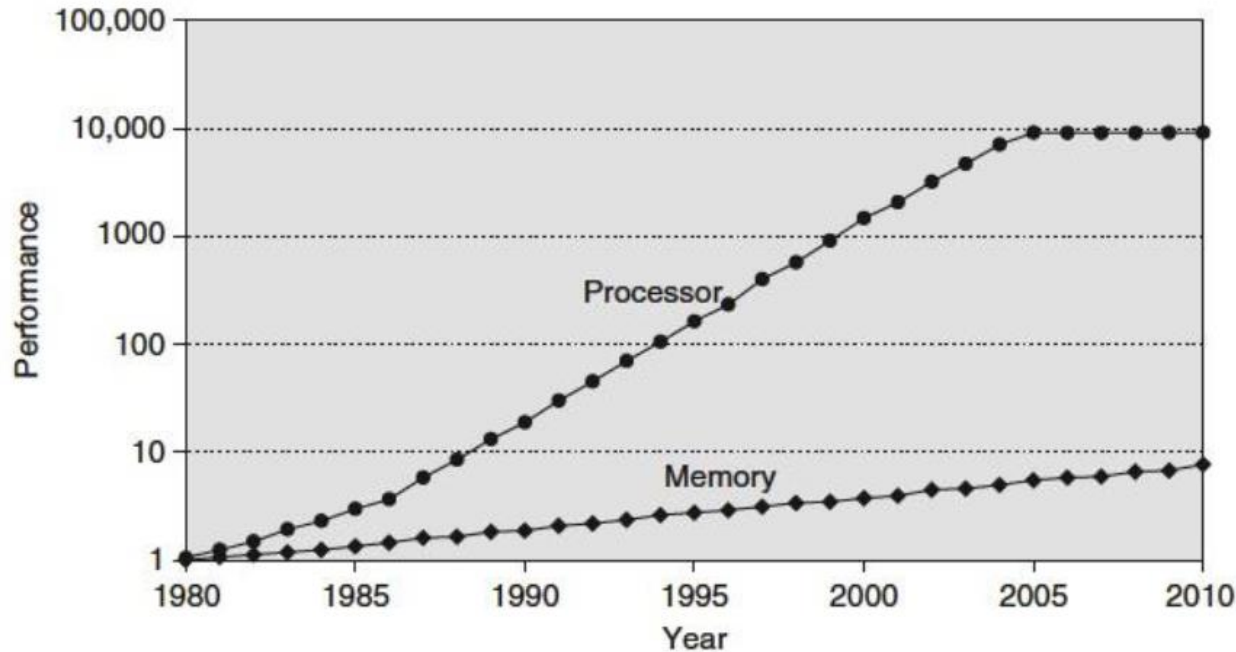
Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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**

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)

Cache vs Memory Relative Speed

- Animation from Mike Acton's Cppcon 2014 talk on "data oriented design".
  - https://youtu.be/rX0ItVEVjHc?si=MRTeW3taRmRU1fpB&t=1830
  - Animation starts at 30:30, ends 31:07 ish

cppcon
The Battle of North Bridge
Andreas Fredriksson
L1
L2
RAM

Cache Performance

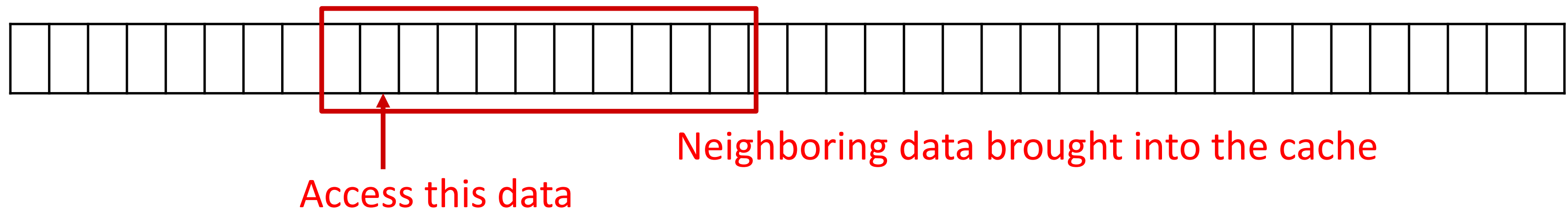
- Accessing data in the cache allows for much better utilization of the CPU

- Accessing data **not** in the cache can cause a bottleneck: CPU would have to wait for data to come from memory.

- How is data loaded into a Cache?

Cache Lines

- Imagine memory as a big array of data:



- we can split memory into 64-byte “lines” or “blocks”(64 bytes on most architectures)
  - This means bottom 6 bits of an address are the offset into a line
  - The top 58 bits of the address specify the “line” number

- When we access data at an address, we bring the whole cache line (cache block) into the L1 Cache
  - Data next to address access is thus also brought into the cache!

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

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

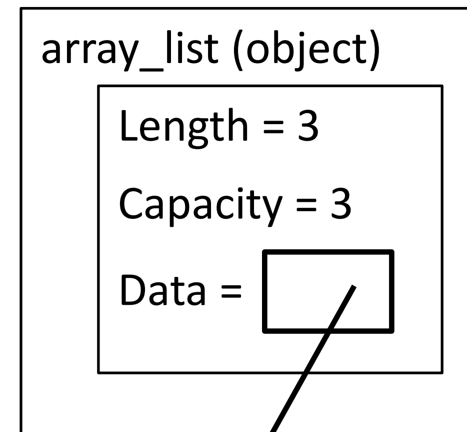
- Important to understanding the poll questions, we understand the memory layout of these data structures

- ArrayList In C++:

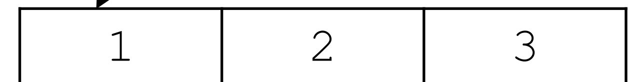
```
int main() {
  vector<int> array_list {1, 2, 3};
  // ...
}
```

stack:

main's stack frame



heap:



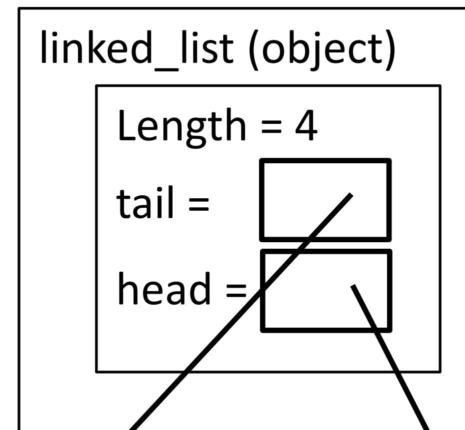
Data Structure Memory Layout

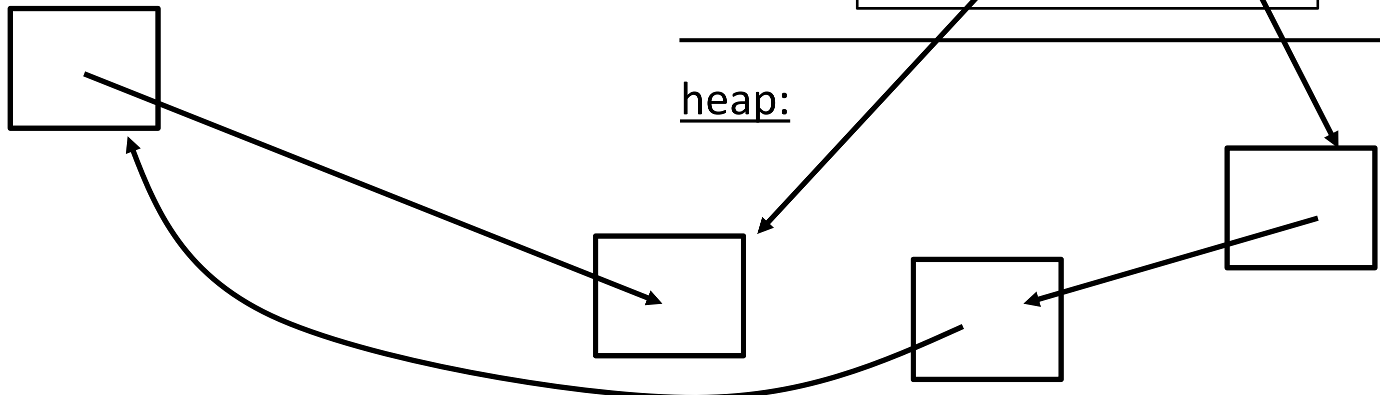
- Important to understanding the poll questions, we understand the memory layout of these data structures

stack:

- 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 locality (and temporal locality 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**

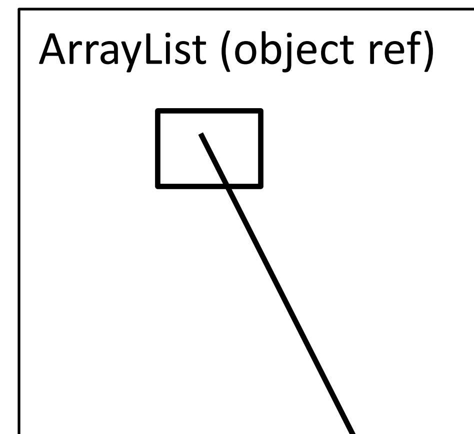
- In most other languages (including Java, Python, etc.), the memory model is slightly different. Instead, **all object variables are object references, that refer to an object on the heap**

ArrayList in Java Memory Model

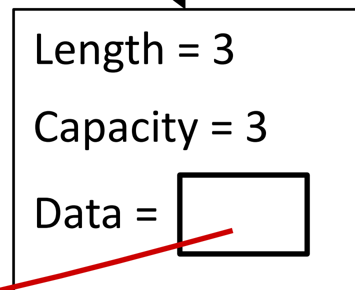
- In Java, the memory model is slightly different. all object variables are object references, that refer to an object on the heap

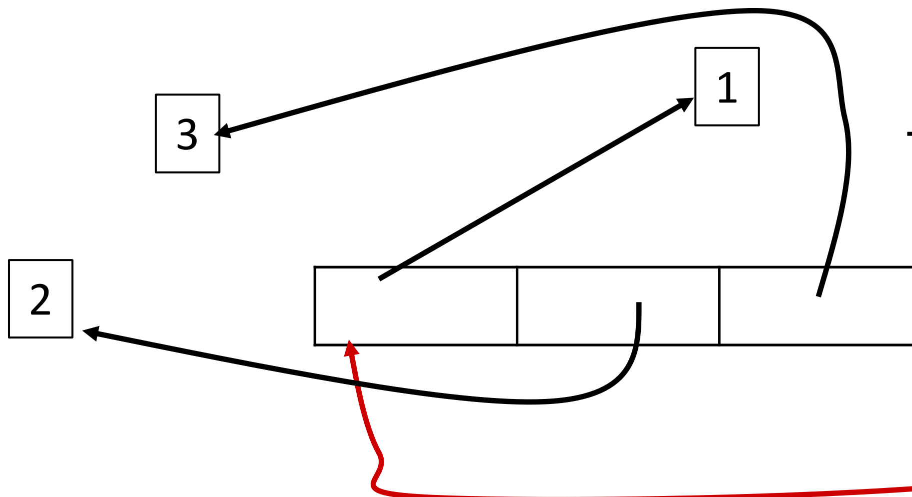
```
public class MemoryModel {
  public static void main(String[] args) {
    ArrayList l = new ArrayList({1, 2, 3});
    // …
  }
}
```

stack:



heap:

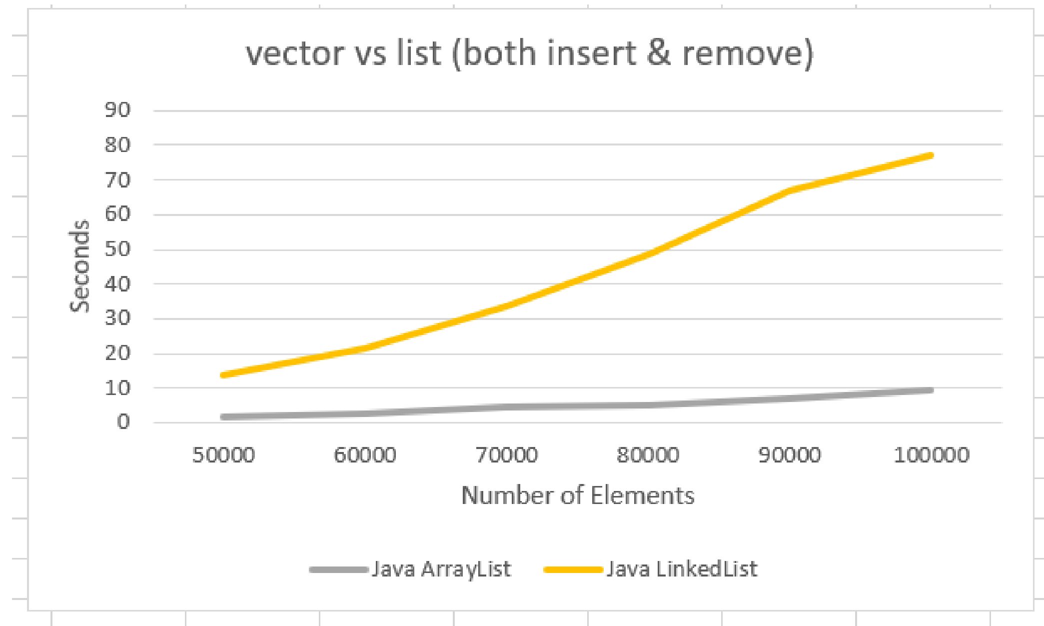




Does Caching apply to Java?

- I believe so, yes. Doing the same experiment in java got:

- Note: did this on smaller number of elements. 50,000 -> 100,000



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

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, …)

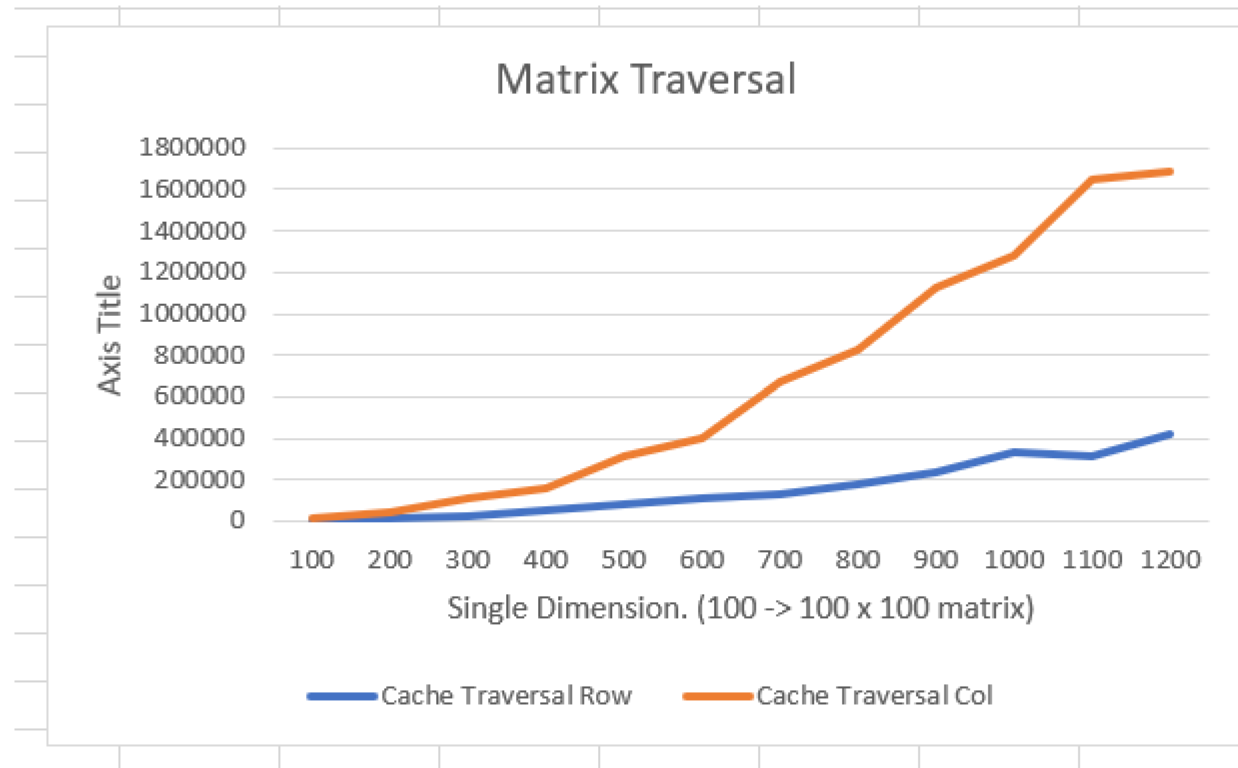
1	5	8	10
11	2	6	9
14	12	3	7
0	15	13	4

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();
    }
  }
}
```