Locality, Buffering, Caches

Computer Operating Systems, Spring 2024

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Adam Gorka	Haoyun Qin	Kyrie Dowling	Ryoma Harris
Andy Jiang	Jeff Yang	Oliver Hendrych	Shyam Mehta
Charis Gao	Jerry Wang	Maxi Liu	Tom Holland
Daniel Da	Jinghao Zhang	Rohan Verma	Tina Kokoshvili
Emily Shen	Julius Snipes	Ryan Boyle	Zhiyan Lu

Administrivia

- Penn-shell is out!
  - **Full thing is due at the end of the week** (2/23 @ 11:59 pm)
  - Done in partners
  - Should have everything you need to complete the assignment
  - Please add your partner to the gradescope submission if you can.

Administrivia

- Midterm booked:
  - 5:15 - 7:15 pm in Meyerson B1
  - Thursday 2/29 (the Thursday before break)
  - Let me know if you conflicts

- Final Tentatively Booked
  - Tuesday May 7th, Noon – 2pm in Towne 100
  - Not confirmed yet, but this is likely it

- Travis is still a little sick, but probably be in-person for next lecture

Penn-Shell Compatibility

- From the signal(2) man page

```
Portability
    The only portable use of signal() is to set a signal's disposition to SIG_DFL or SIG_IGN.  The semantics when using signal()
    to establish a signal handler vary across systems (and POSIX.1 explicitly permits this variation); do not use  it  for  this
    purpose.
```

- If you want to have better help from TA's put this at the top of your file before you #include anything
  - This *should* get signals to behave as we expect, so TAs can better help
  - If you got it working another way, that is OK. Auto-grader *should* still accept it

```
#ifndef _POSIX_C_SOURCE
#define _POSIX_C_SOURCE 200809L
#endif

#ifndef _DEFAULT_SOURCE
#define _DEFAULT_SOURCE 1
#endif
```

Poll Everywhere

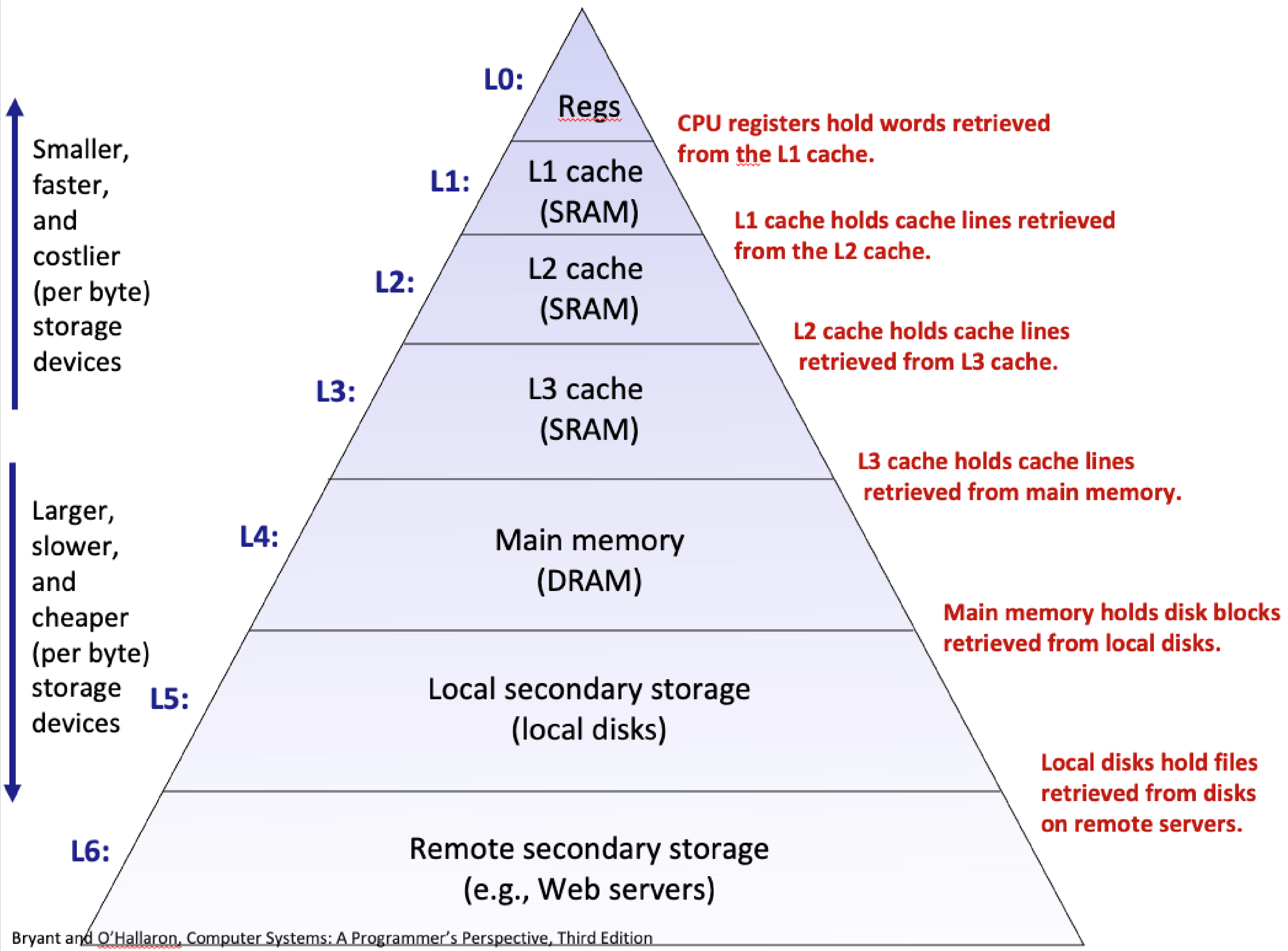
pollev.com/tqm

- How are you doing?

Lecture Outline

- **Locality**
- I/O Buffering
- Caches

Memory Hierarchy



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

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.

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

- Locality
- **I/O Buffering**
- Caches

Poll Everywhere

pollev.com/tqm

- If we compile this and run it, how many times is hello printed?

```
int main() {
  if (fork() == 0) {
    write(STDOUT_FILENO, "hello", 5);
  }
  if (fork() == 0) {
    write(STDOUT_FILENO, "hello", 5);
  }
  return EXIT_SUCCESS;
}
```

Raise Your Hands

- If we compile this and run it, how many times is hello printed?

```
int main() {
  if (fork() == 0) {
    printf("hello");
  }
  if (fork() == 0) {
    printf("hello");
  }
  return EXIT_SUCCESS;
}
```

Raise Your Hands

- If we compile this and run it, how many times is hello printed?

```
int main() {
  if (fork() == 0) {
    printf("hello\n");
  }
  if (fork() == 0) {
    printf("hello\n");
  }
  return EXIT_SUCCESS;
}
```

C stdio vs POSIX

- Why are we getting these different outputs?

- Let's start with the first two. Both use different ways of writing to standard out.

  - C stdio : user level portable library for **st**andar**d** **i**nput/**o**utput. Should work on any environment that has the C standard library
    - E.g. printf, fprintf, fputs, getline, etc.

  - POSIX C API: **P**ortable **O**perating **S**ystem **I**nterface. Functions that are supported by many operating systems to support many OS-level concepts (Input/Output, networking, processes, threads…)

Buffered writing

- By default, C `stdio` uses buffering on top of POSIX:
  - When one writes with **`fwrite`**`()`, the data being written is copied into a buffer allocated by `stdio` inside your process' address space
  - As some point, once enough data has been written, the buffer will be "flushed" to the operating system.
    - When the buffer fills (often 1024 or 4096 bytes)
  - This prevents invoking the write system call and going to the filesystem too often

Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  FILE* fout = fopen("hi.txt", "wb");

  // read "hi" one char at a time
  fwrite(&buf, sizeof(char), 1, fout);

  fwrite(&buf+1, sizeof(char), 1, fout);

  fclose(fout);
  return EXIT_SUCCESS;
}
```

buf

h	i

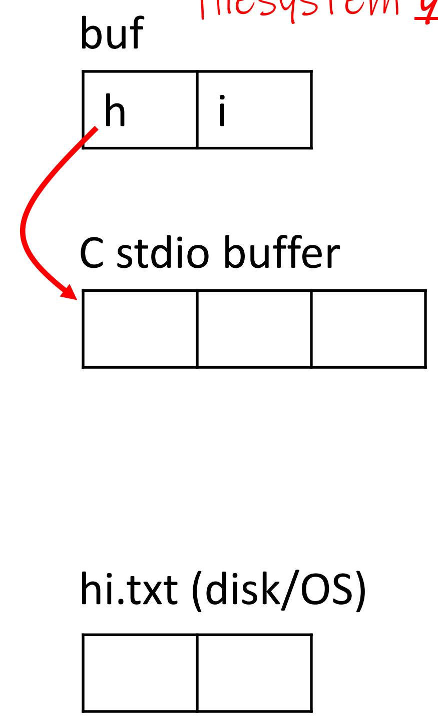
hi.txt (disk/OS)

Buffered Writing Example

Arrow signifies what will be executed next

Store 'h' into buffer, so that we do not go to filesystem *yet*

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  FILE* fout = fopen("hi.txt", "wb");

  // read "hi" one char at a time
  fwrite(&buf, sizeof(char), 1, fout);   <-- next

  fwrite(&buf+1, sizeof(char), 1, fout);

  fclose(fout);
  return EXIT_SUCCESS;
}
```

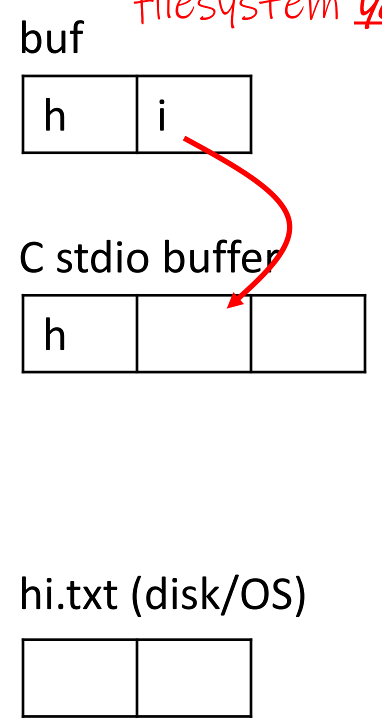


Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  FILE* fout = fopen("hi.txt", "wb");

  // read "hi" one char at a time
  fwrite(&buf, sizeof(char), 1, fout);

→ fwrite(&buf+1, sizeof(char), 1, fout);

  fclose(fout);
  return EXIT_SUCCESS;
}
```

Store 'i' into buffer, so that we do not go to filesystem *yet*



Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  FILE* fout = fopen("hi.txt", "wb");

  // read "hi" one char at a time
  fwrite(&buf, sizeof(char), 1, fout);

  fwrite(&buf+1, sizeof(char), 1, fout);

→ fclose(fout);
  return EXIT_SUCCESS;
}
```

buf

h	i

C stdio buffer

h	i	

When we call fclose, we deallocate and flush the buffer to disk

hi.txt (disk/OS)

Buffered Writing Example

Arrow signifies what will be executed next

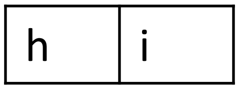
```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  FILE* fout = fopen("hi.txt", "wb");

  // read "hi" one char at a time
  fwrite(&buf, sizeof(char), 1, fout);

  fwrite(&buf+1, sizeof(char), 1, fout);

  fclose(fout);
  return EXIT_SUCCESS;
}
```

Unbuffered Writing Example

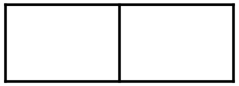
Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &buf, sizeof(char));

  write(fd, &buf+1, sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```

buf



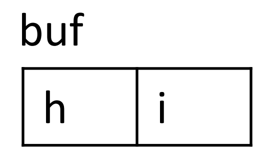
hi.txt (disk/OS)

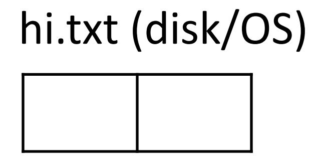


Unbuffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &buf, sizeof(char));

  write(fd, &buf+1, sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```

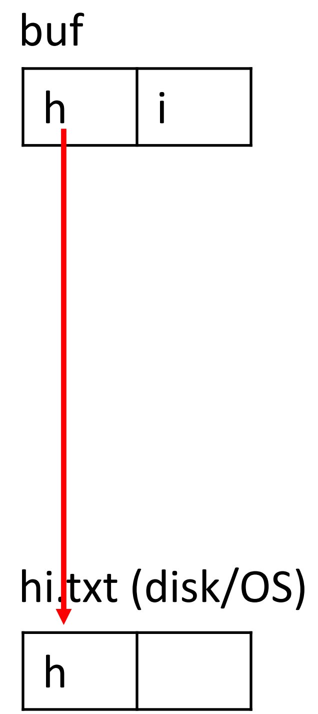




Unbuffered Writing Example

Arrow signifies what will be executed next

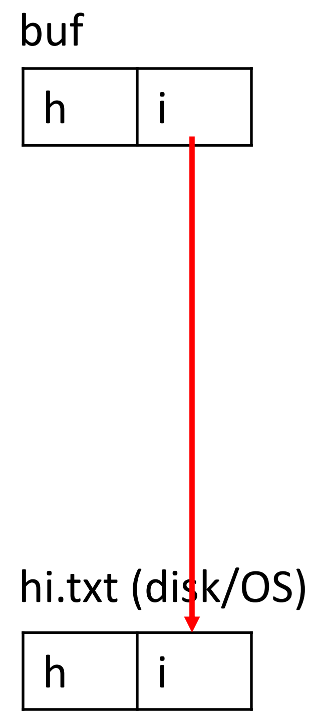
```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &buf, sizeof(char));

  write(fd, &buf+1, sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```



Unbuffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &buf, sizeof(char));

  write(fd, &buf+1, sizeof(char));

→ close(fd);
  return EXIT_SUCCESS;
}
```



Unbuffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &buf, sizeof(char));

  write(fd, &buf+1, sizeof(char));

  close(fd);
→ return EXIT_SUCCESS;
}
```

buf

h	i

Two OS/File system accesses instead of one ☹

hi.txt (disk/OS)

h	i

Buffered Reading

- By default, C `stdio` uses buffering on top of POSIX:
  - When one reads with `fread()`, a lot of data is copied into a buffer allocated by `stdio` inside your process' address space
  - Next time you read data, it is retrieved from the buffer
    - This avoids having to invoke a system call again
  - As some point, the buffer will be "refreshed":
    - When you process everything in the buffer (often 1024 or 4096 bytes)
  - Similar thing happens when you write to a file

Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2];
→ FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

  fclose(fin);
  return EXIT_SUCCESS;
}
```

buf

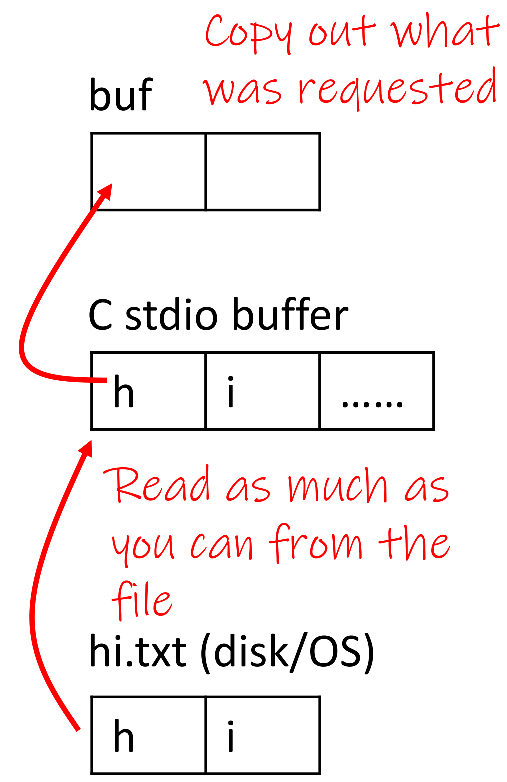
hi.txt (disk/OS)

h	i

Buffered Reading Example

Arrow signifies what will be executed next

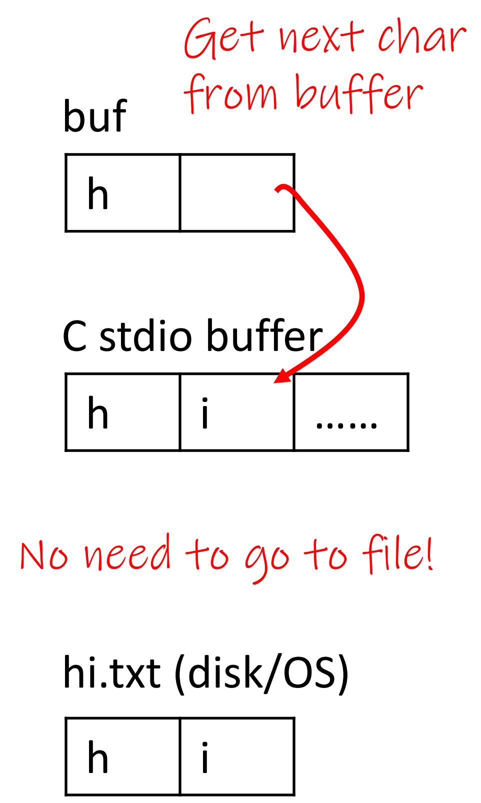
```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);    <--

  fread(&buf+1, sizeof(char), 1, fin);

  fclose(fin);
  return EXIT_SUCCESS;
}
```



Buffered Reading Example

Arrow signifies what will be executed next

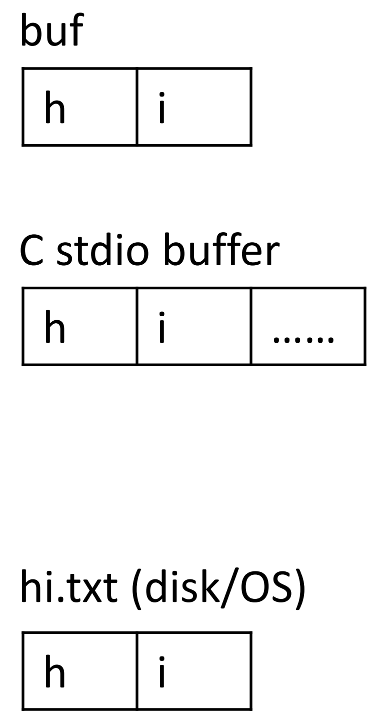
```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

→ fread(&buf+1, sizeof(char), 1, fin);

  fclose(fin);
  return EXIT_SUCCESS;
}
```



Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

→ fclose(fin);
  return EXIT_SUCCESS;
}
```



Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

  fclose(fin);
→ return EXIT_SUCCESS;
}
```

buf

h	i

hi.txt (disk/OS)

h	i

Why NOT Buffer?

- Reliability – the buffer needs to be flushed
  - Loss of computer power = loss of data
  - "Completion" of a write (*i.e.* return from `fwrite()`) does not mean the data has actually been written
- Performance – buffering takes time
  - Copying data into the `stdio` buffer consumes CPU cycles and memory bandwidth
  - Can potentially slow down high-performance applications, like a web server or database (*"zero-copy"*)
- When is buffering faster? | Slower?

Many small writes
Or only writing a little

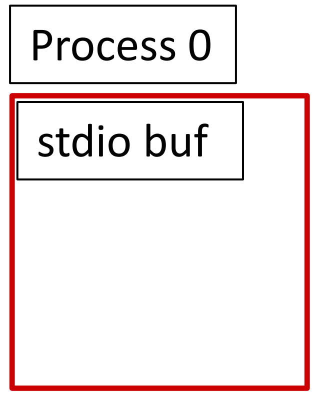
Large writes

Fork Problem Explained

Arrow signifies what will be executed next.
I execute processes in parallel and "in sync" for demonstration purposes

- Remember: printf (and stdio) buffers input in the programs address space

```
int main() {
→ if (fork() == 0) {
    printf("hello");
  }
  if (fork() == 0) {
    printf("hello");
  }
  return EXIT_SUCCESS;
}
```

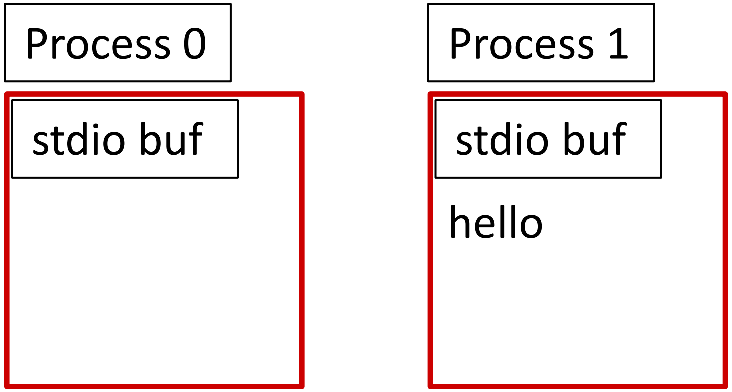


Fork Problem Explained

Arrow signifies what will be executed next.
I execute processes in parallel and "in sync" for demonstration purposes

- Remember: printf (and stdio) buffers input in the programs address space

```
int main() {
  if (fork() == 0) {
    printf("hello");
  }
  if (fork() == 0) {
    printf("hello");
  }
  return EXIT_SUCCESS;
}
```

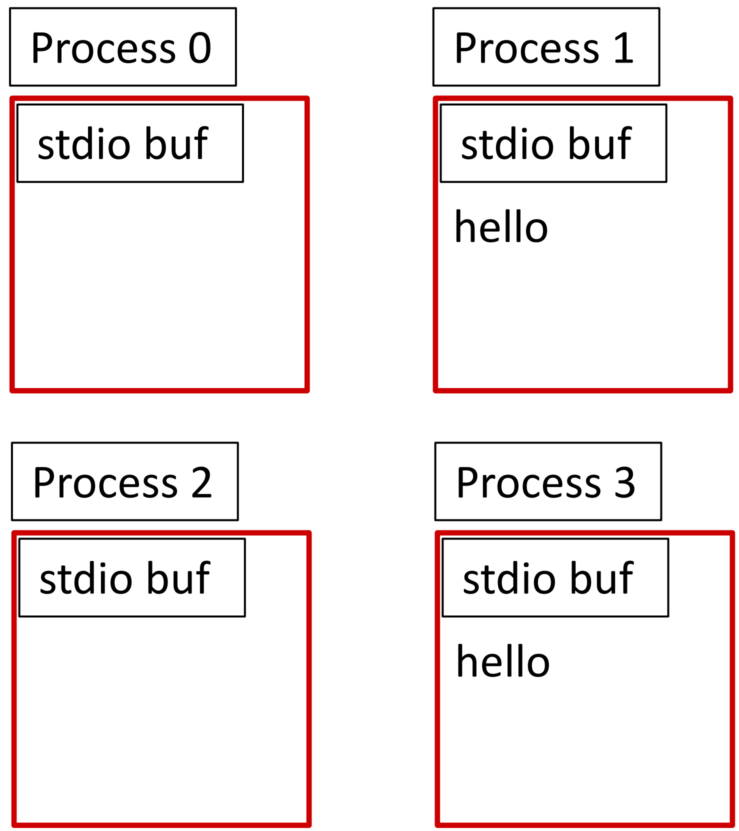


Fork Problem Explained

Arrow signifies what will be executed next.
I execute processes in parallel and "in sync" for demonstration purposes

- Remember: printf (and stdio) buffers input in the programs address space

```
int main() {
  if (fork() == 0) {
    printf("hello");
  }
  if (fork() == 0) {
    printf("hello");
  }
  return EXIT_SUCCESS;
}
```

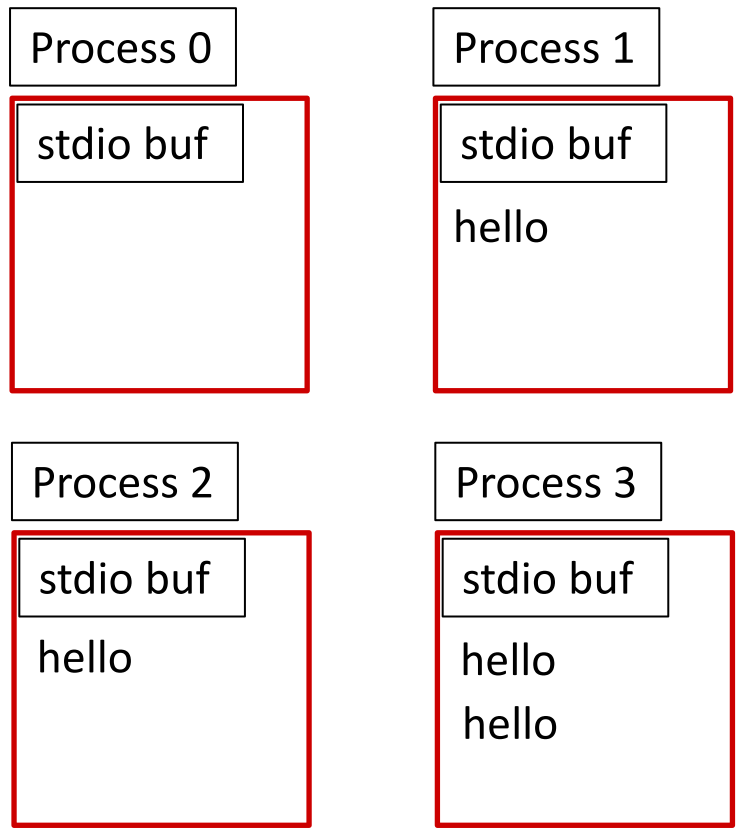


Fork Problem Explained

Arrow signifies what will be executed next.
I execute processes in parallel and “in sync” for demonstration purposes

- Remember: printf (and stdio) buffers input in the programs address space

```
int main() {
  if (fork() == 0) {
    printf("hello");
  }
  if (fork() == 0) {
    printf("hello");
  }
  return EXIT_SUCCESS;
}
```



Hello is printed 4 times!

Fork Problem Explained (pt.2)

- Why did we get different outputs when printf printed a newline character after hello?
  - Only difference was:

```
printf("hello");
```

vs

```
printf("hello\n");
```

- All we needed to do to get the expected output was add a \n. why?

- `printf` prints to stdout and by default stdout is line buffered. Meaning it flushes the buffer on a newline character
  - If we ran ./prog > out.txt (redirect the output), we would get different output since buffering policy changes.

How to flush/modify the cstdio buffer

- For C stdio:
  - ```
    int fflush(FILE* stream);
    ```
  - Flushes the stream to the OS/filesystem
  - ```
    int setvbuf(FILE* stream, char* buf,
                int mode, size_t size);
    ```
  - Has a family of related functions like setbuf(), setbuffer(), setlinebuf();
  - Can set the stream to be unbuffered or a specified buffer

How to flush POSIX?

- When we write to a file with POSIX it is sent to the filesystem, is it immediately sent to disc? No

  - Well, we do have the block cache… so it may not be written to disc

  - Since all File I/O requests go to the file system, if another process accesses the same file, then it should see the data even if it is the block cache and not in disc.

  - If we lose power though…

How to flush POSIX to disk

- Two functions
  - `int fsync(int fd);`
  - Flushes all in-core data and metadata to the storage medium
  - `int fdatasync(int fd);`
  - Sends the file data to disk
  - Does not flush modified metadata unless necessary for data.
- C stdio is usually implemented using POSIX on posix compliant systems
  - **`fflush`** may not necessarily call **`fsync`**

Blank slide

- Blank slide

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?

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

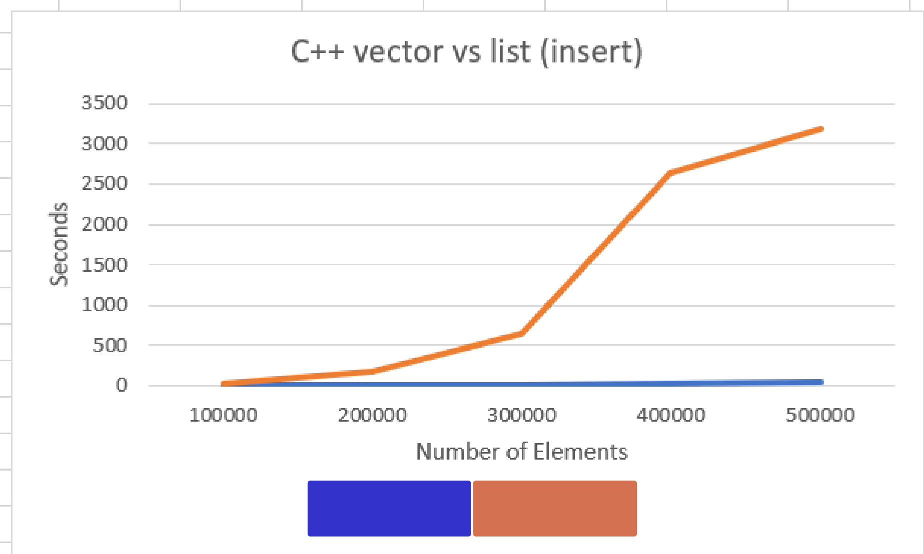
- Part 2: Let's say we take the list from part 1, randomly generate an index and remove that index from the sequence until it is empty. Would this be faster on a LinkedList or an ArrayList?

Lecture Outline

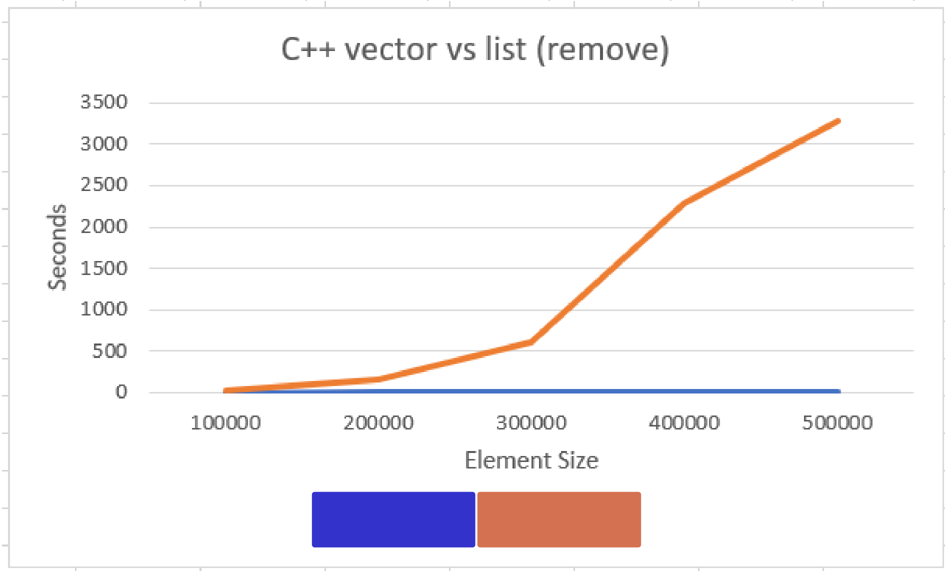
- Locality
- I/O Buffering
- **Caches**

Answer:

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



- Terminology
  - Vector == ArrayList
  - List == LinkedList



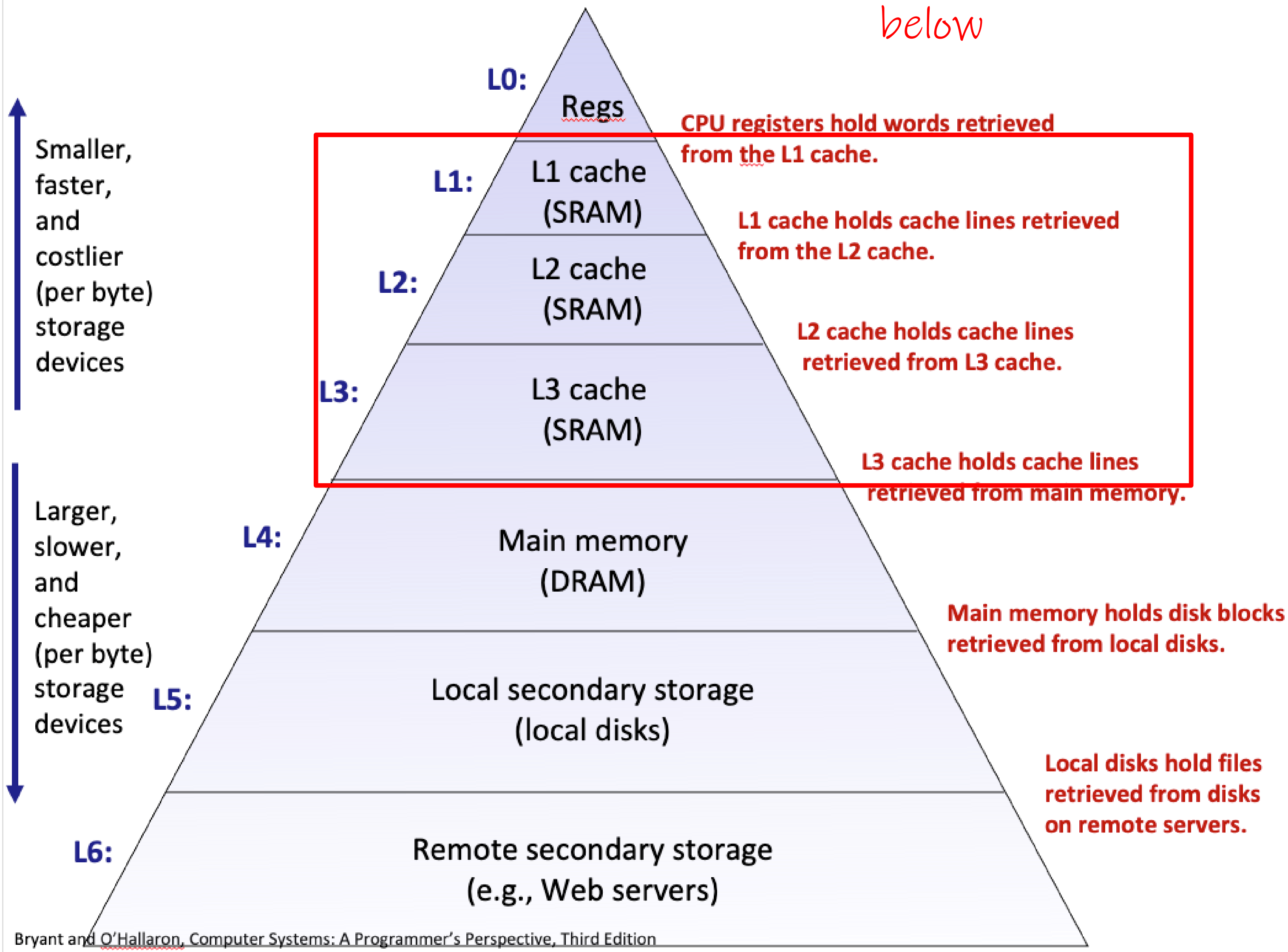
- On Element size from 100,000 -> 500,000

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



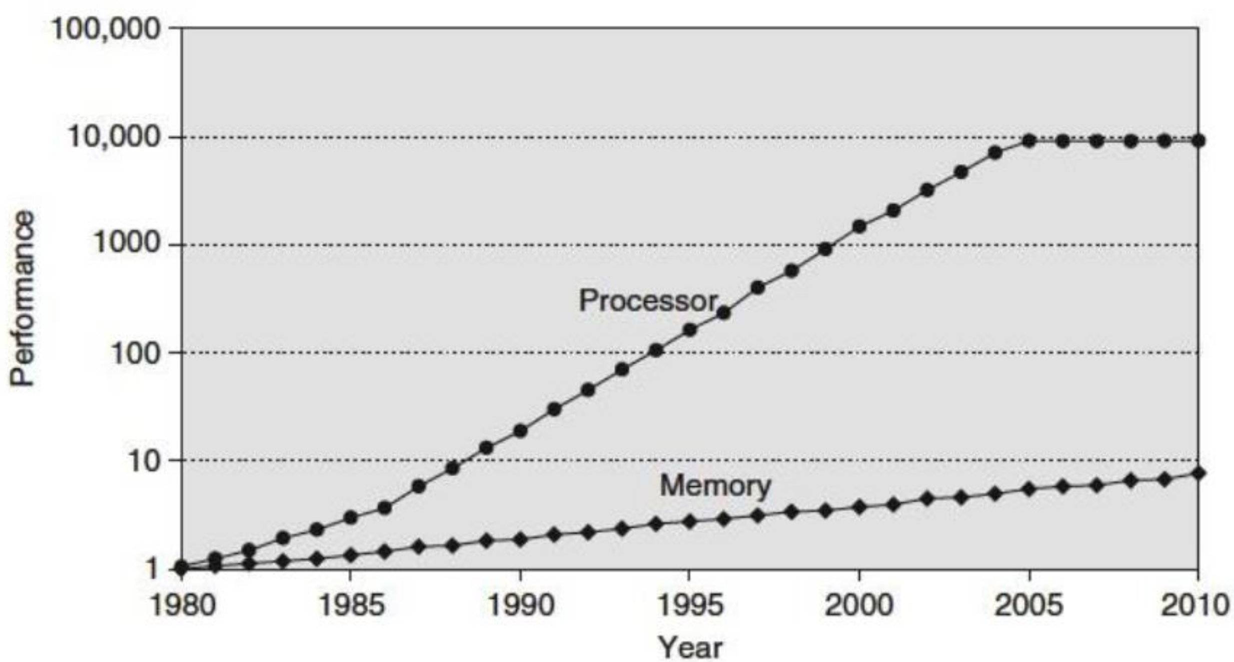
Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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.

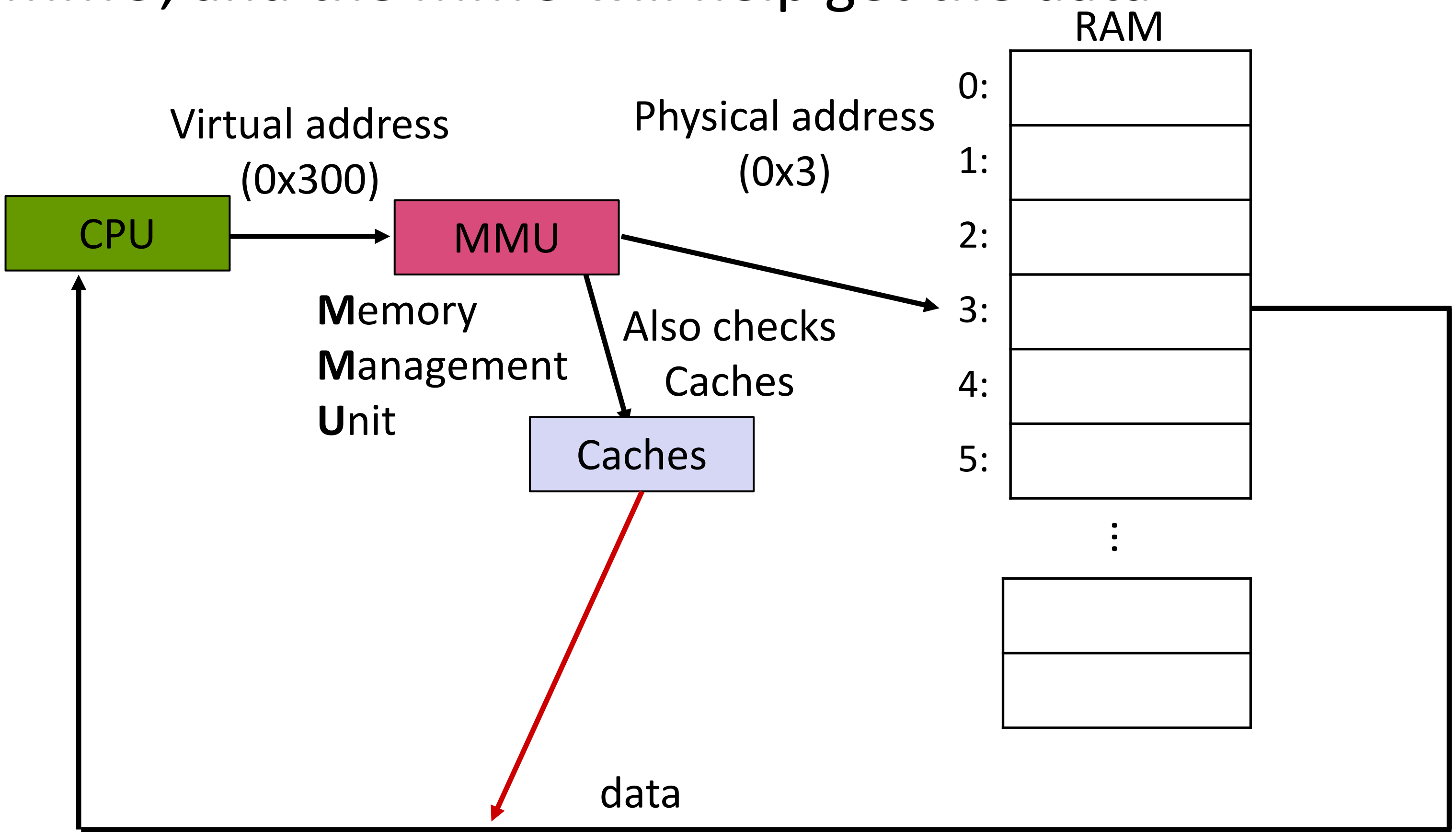
- Computer: Memory with shorter access time used for the storage of data for increased performance. Data is usually either something frequently and/or recently used.
  - Physical memory is a “Cache” of page frames which may be stored on disk. (Instead of going to disk, we can go to physical memory which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

Memory (as we know it now)

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

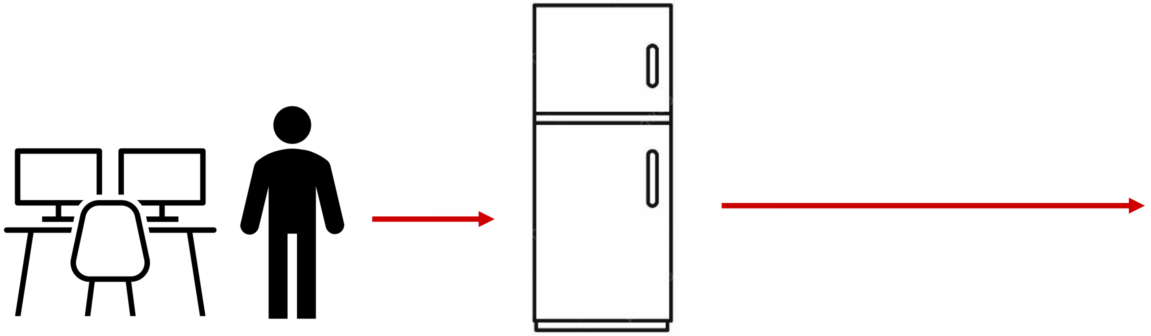
Virtual Address Translation

- Programs don't know about many of things going on under the hood with memory.they send an address to the MMU, and the MMU will help get the data



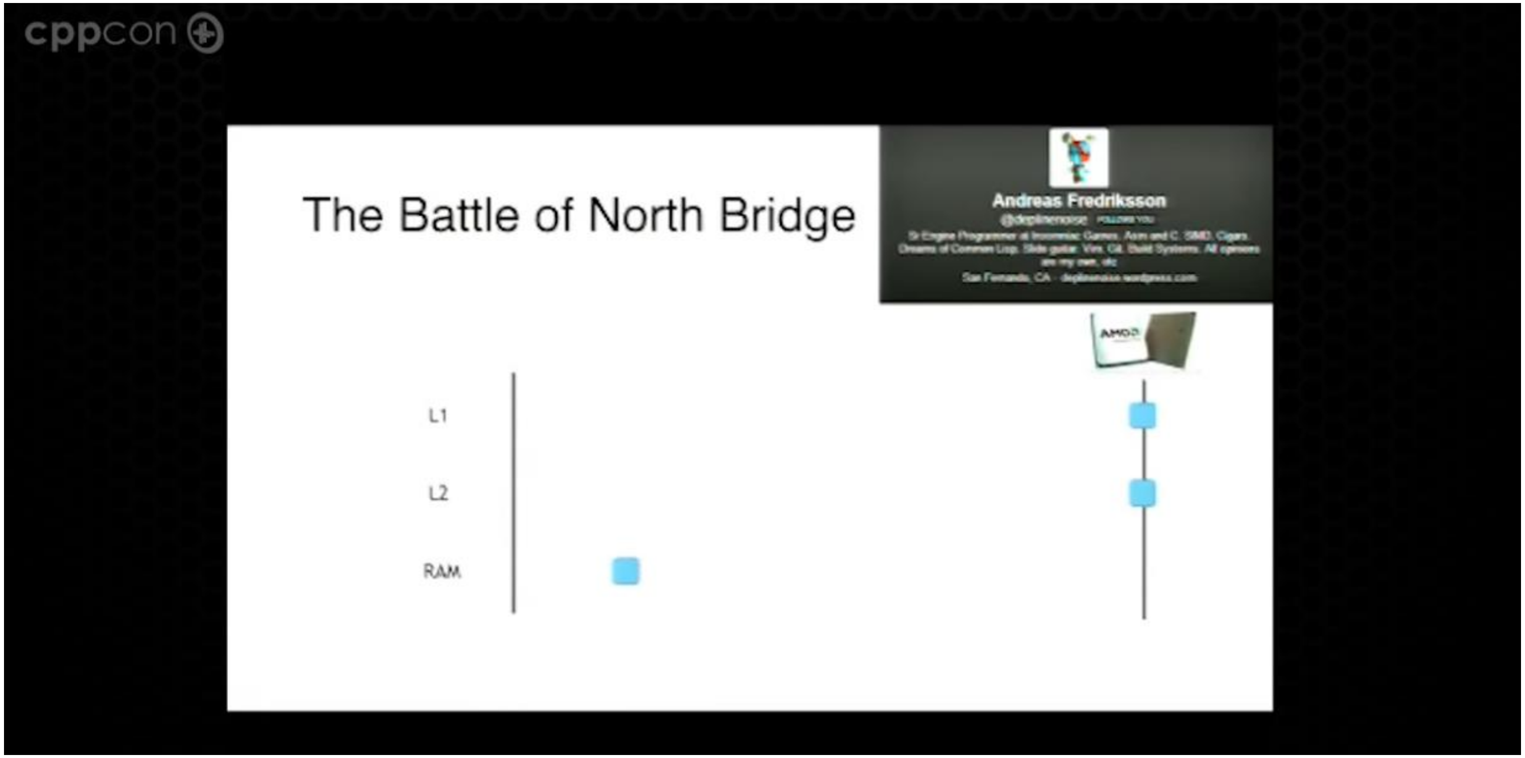
Cache Analogy

- If we are at home and we are hungry, were 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 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



Cache Performance

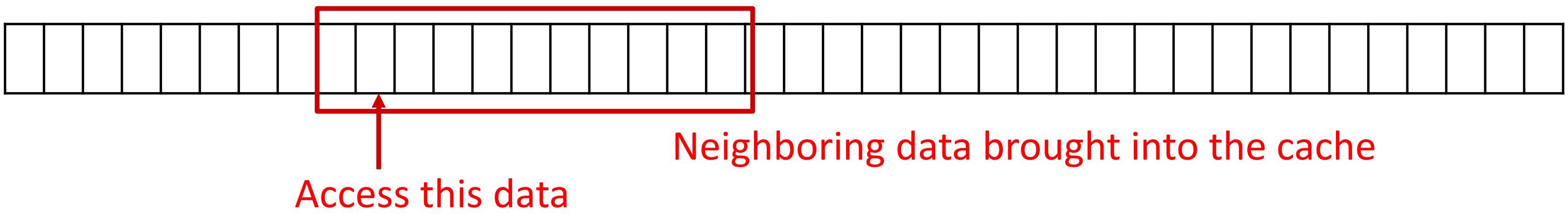
- Accessing data in the cache allows for much better utilization of the CPU

- 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)

- 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!

Principle of Locality

- The tendency for the CPU 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.

- Caches take advantage of these tendencies to help with cache management

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

Back to the Poll Questions

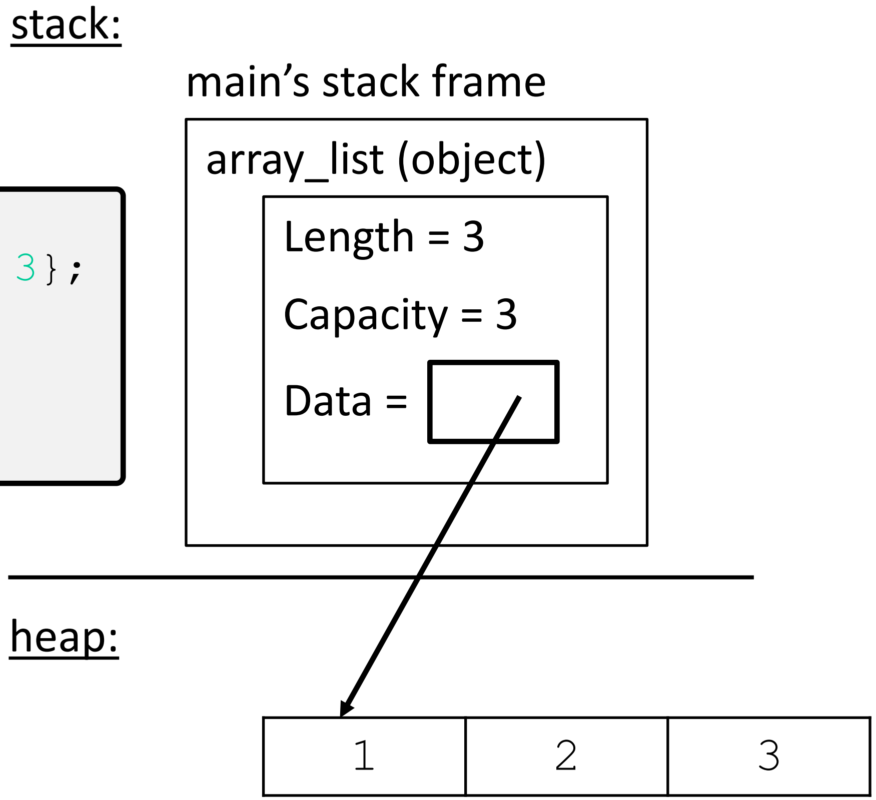
- Data Structures Review: I want to randomly generate a sequence of sorted numbers. To do this, we generate a random number and insert the number so that it remains sorted. Would a LinkedList or an ArrayList work better?

- 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};
  // ...
}
```



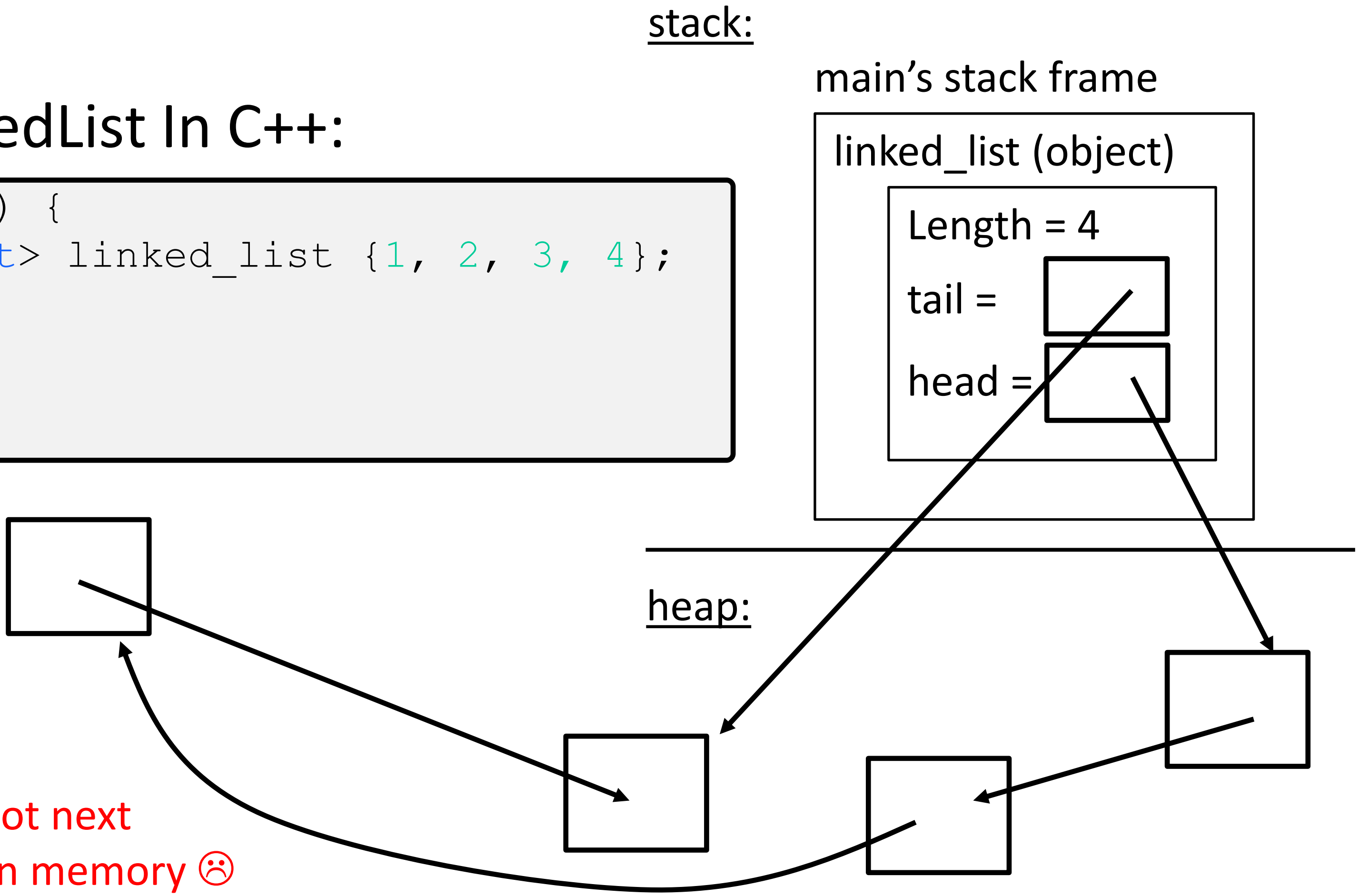
Elements are next to each other in memory ☺

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};
  // …
}
```



Elements are not next to each other in memory ☹

Poll Question: Explanation

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

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

What about other languages?

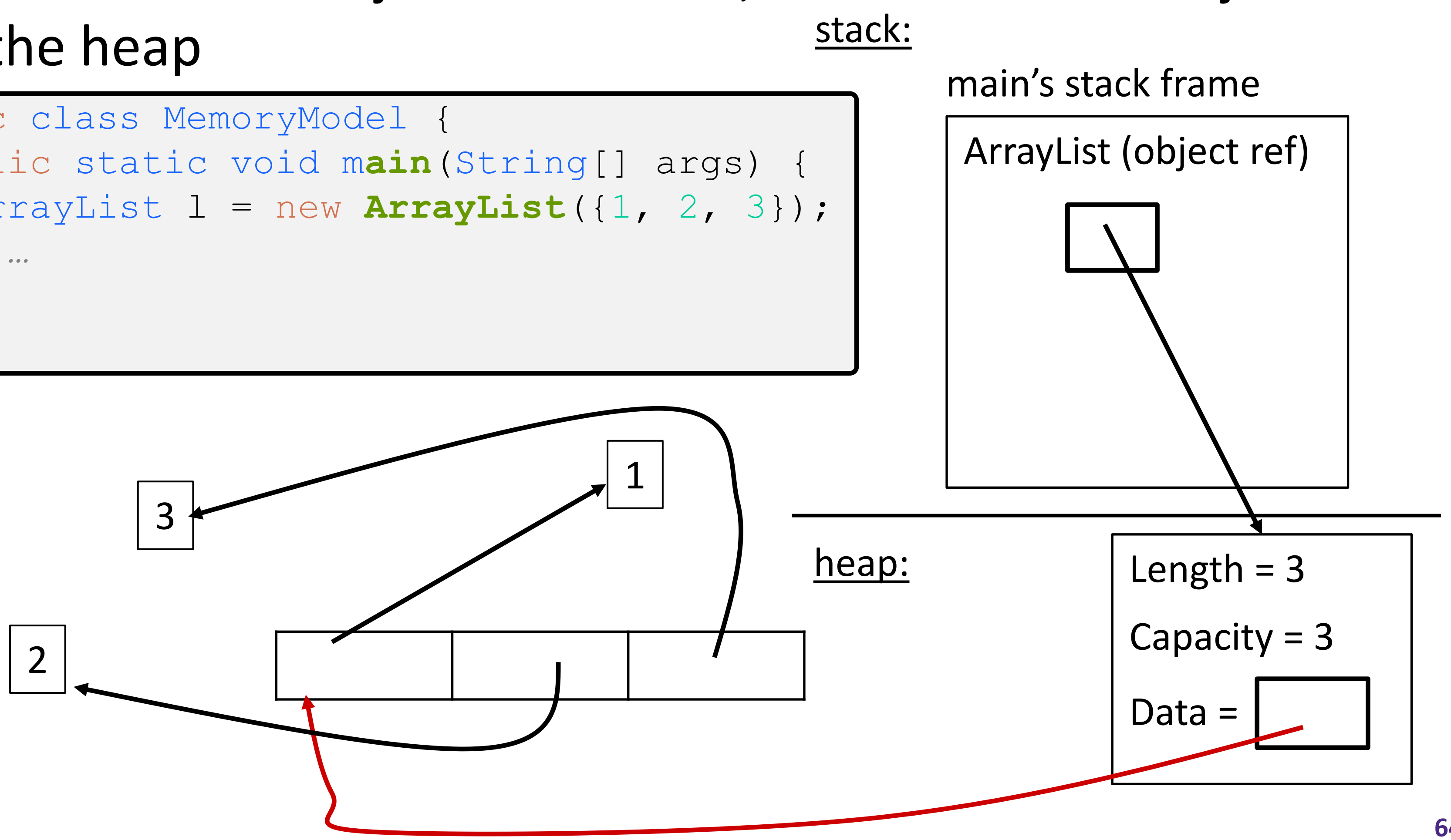
- In C++ (and C, Rust, Zig …) when you declare an object, you have an instance of that object. If you declare it as a local variable, it **exists on the stack**

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

ArrayList in Java Memory Model

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

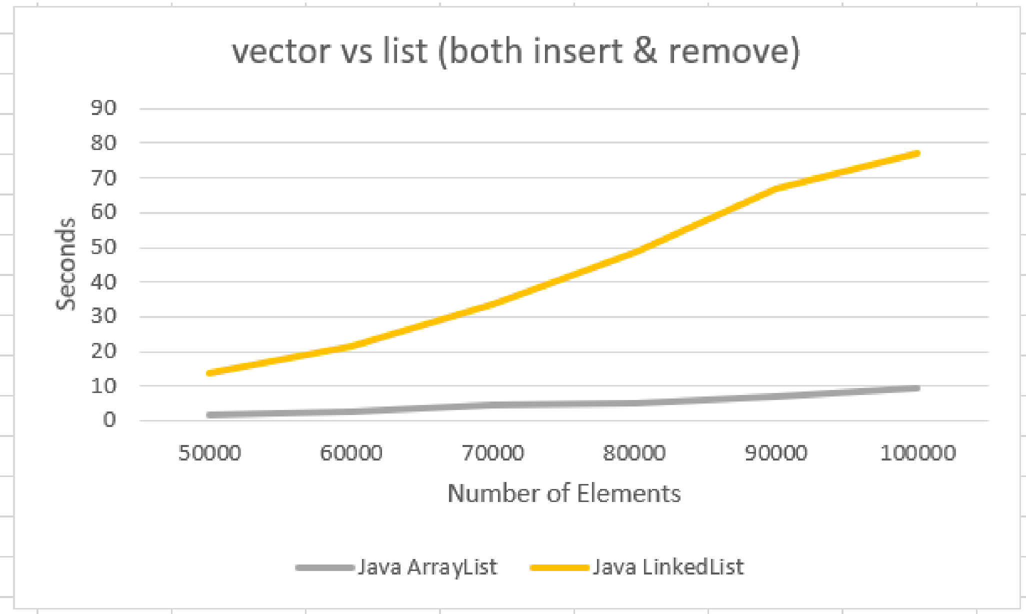
```
public class MemoryModel {
  public static void main(String[] args) {
    ArrayList l = new ArrayList({1, 2, 3});
    // ...
  }
}
```



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



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, …)

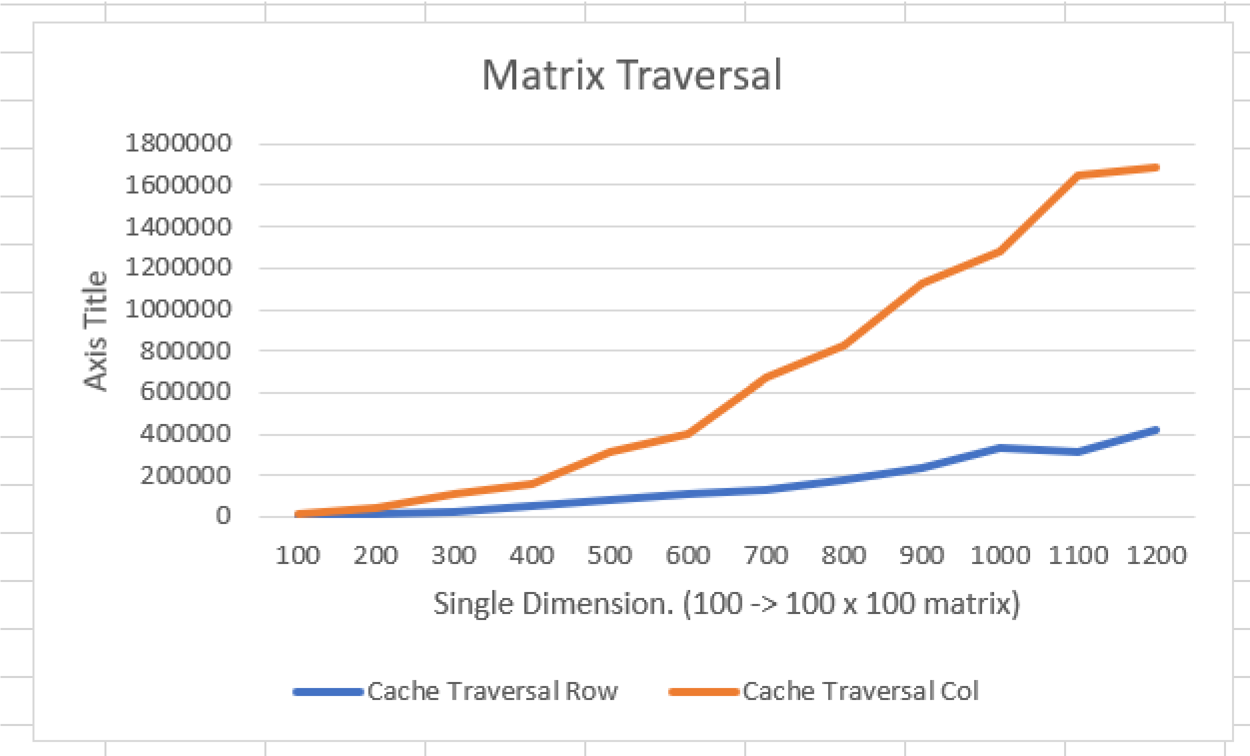
1	5	8	10
11	2	6	9
14	12	3	7
0	15	13	4

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();
    }
  }
}
```