Caches & threads

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang
Audrey Yang
August Fu
Daniel Da
Ernest Ng
Haoyun Qin
Jason hom
Jeff Yang
Jerry Wang
Jinghao Zhang
Kevin Bernat
Leon Hertzberg
Maxi Liu
Ria Sharma
Rohan Verma
Ryoma Harris
Shyam Mehta
Tina Kokoshvili
Zhiyan Lu

Administrivia

- Project 1 is out now
  - Project is due 11:59 pm on Wed, Oct 11 **(1 week from yesterday)**
    late deadline 11:59 pm on Sun, Oct 15

- For project 1 full submission, please do a group submission on gradescope (one of you submits but you add your partner to the submission)

- Midterm is coming soon (two weeks from now)
  - Meyerson B1 7:00 pm to 9:00pm Thursday 10/19
  - If you can't make the time, please send me an email

Poll Everywhere

pollev.com/tqm

- Any questions, comments or concerns from last lecture?

Discuss

- 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

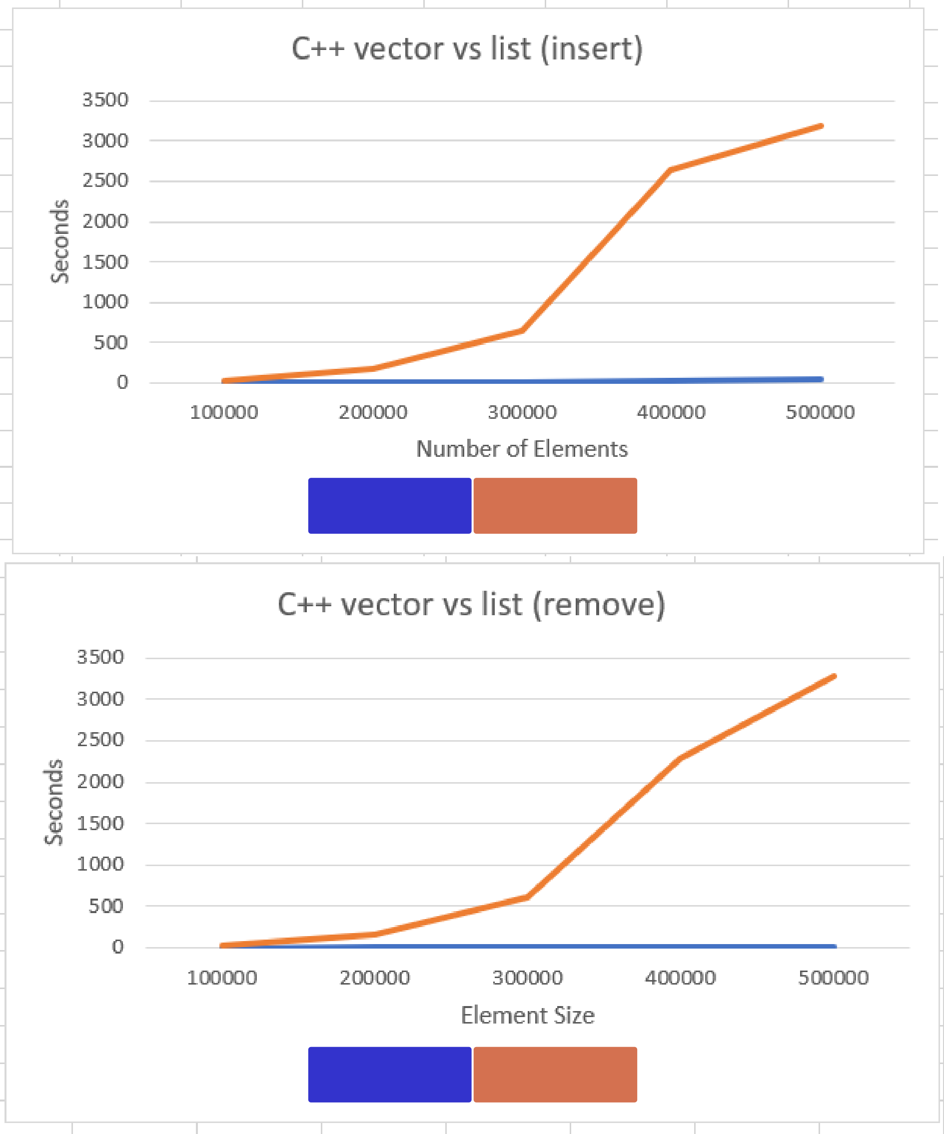
- **Intro to Caches**
- Threads High Level
- Pthreads
- Threads vs processes

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

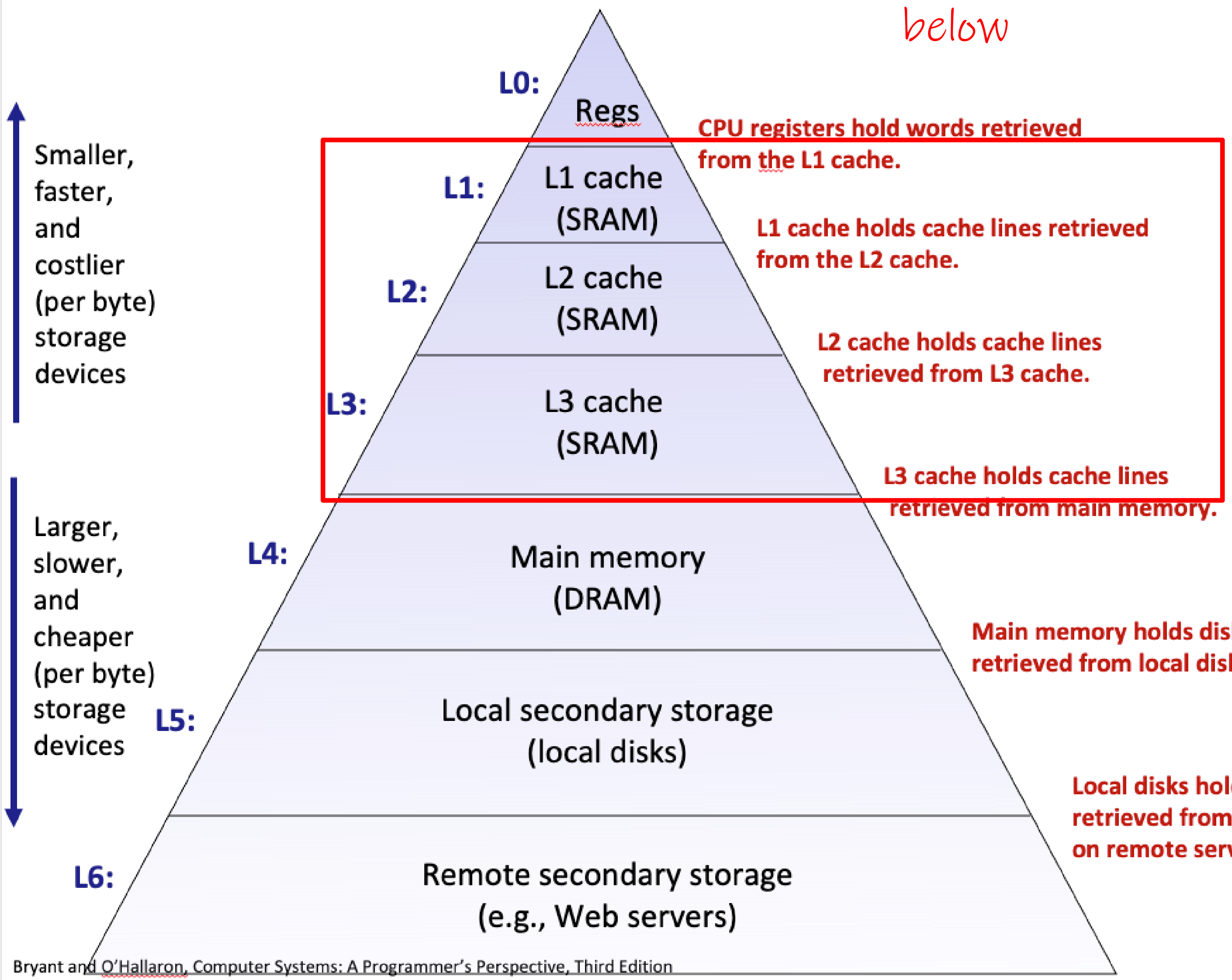
- Data is stored on a physical piece of hardware

- 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



Smaller, faster, and costlier (per byte) storage devices

Larger, slower, and cheaper (per byte) storage devices

L0: Regs

L1: L1 cache (SRAM)

L2: L2 cache (SRAM)

L3: L3 cache (SRAM)

L4: Main memory (DRAM)

L5: Local secondary storage (local disks)

L6: Remote secondary storage (e.g., Web servers)

CPU registers hold words retrieved from the L1 cache.

L1 cache holds cache lines retrieved from the L2 cache.

L2 cache holds cache lines retrieved from L3 cache.

L3 cache holds cache lines retrieved from main memory.

Main memory holds disk blocks retrieved from local disks.

Local disks hold files retrieved from disks on remote servers.

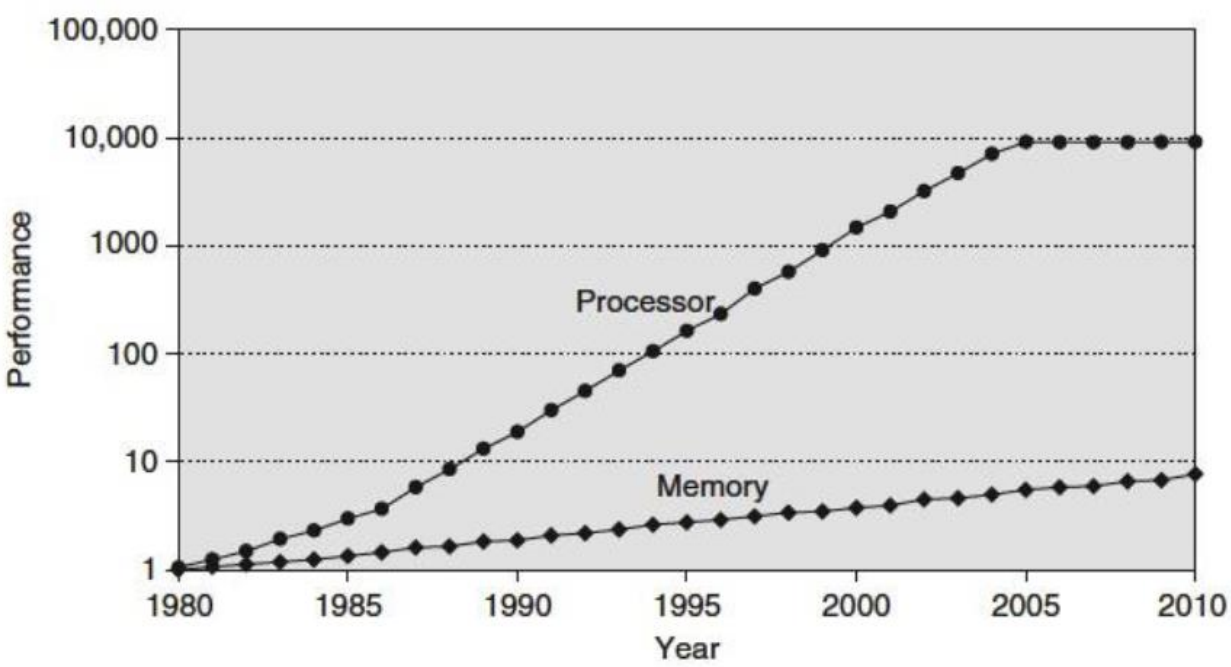
Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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
- **<u>Memory access would create a bottleneck on performance</u>**
  - **It is important that data is quick to access to get better CPU utilization**

Cache

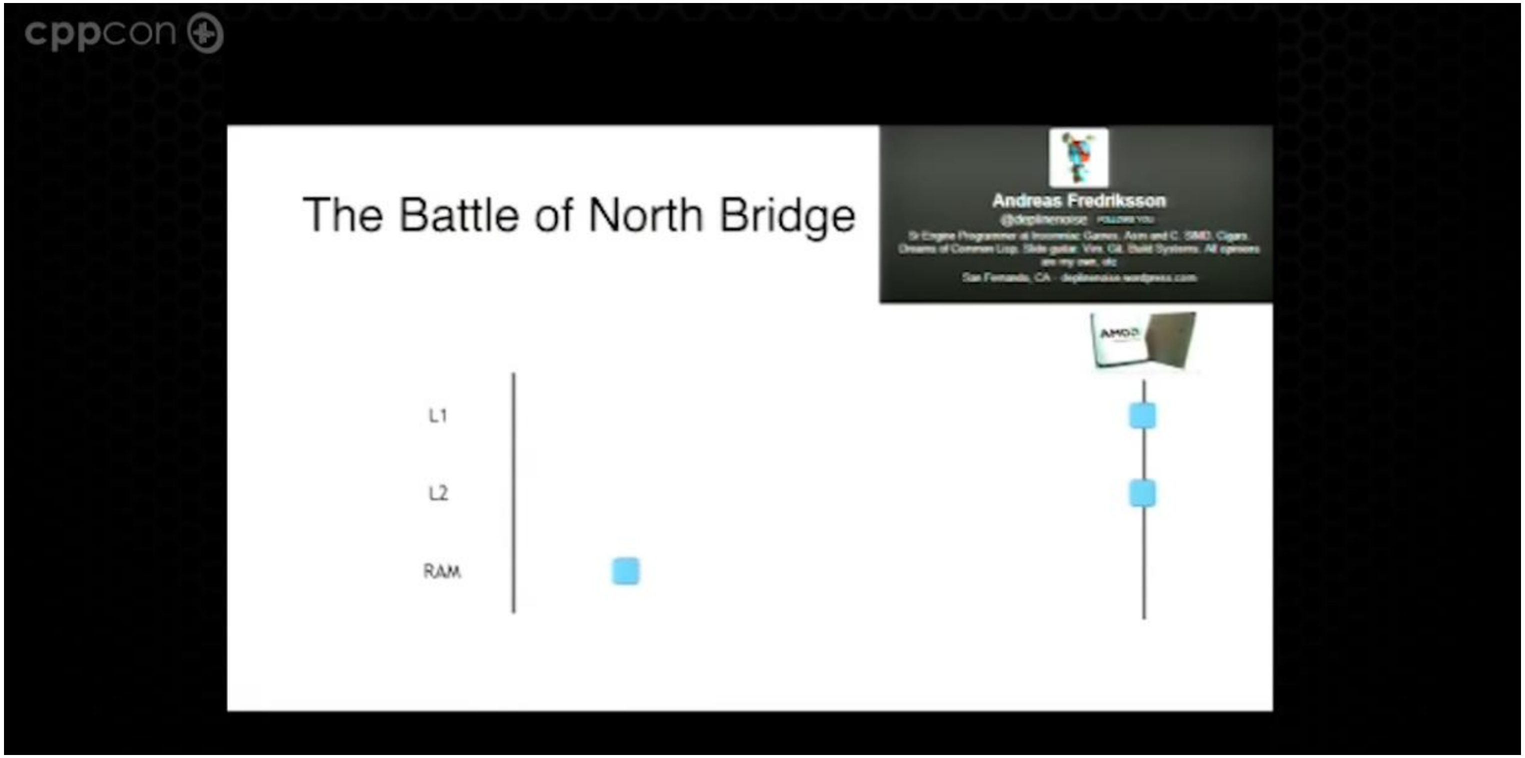
- Pronounced “cash”

- 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



The Battle of North Bridge

Cache Performance

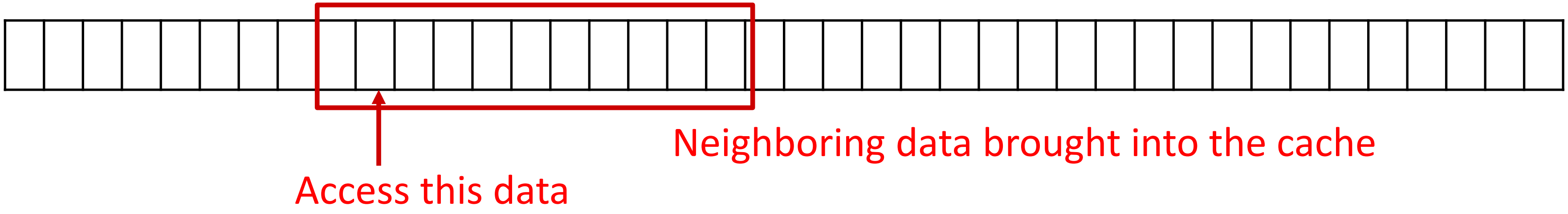
- Accessing data in the cache allows for much better utilization of the CPU

- 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:



- Just like we did with pages, we can split these 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?

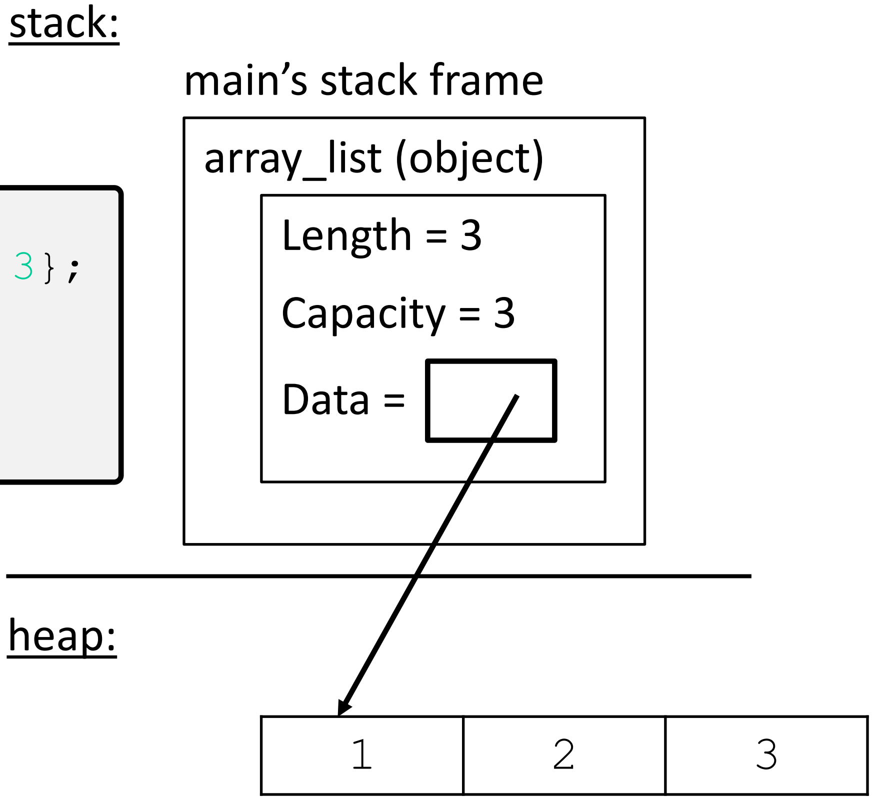
- Part 2: Let’s say we take the list from part 1, randomly generate an index and remove that index from the sequence until it is empty. Would this be faster on a LinkedList or an ArrayList?

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

array_list (object)

Length = 3

Capacity = 3

Data =

heap:

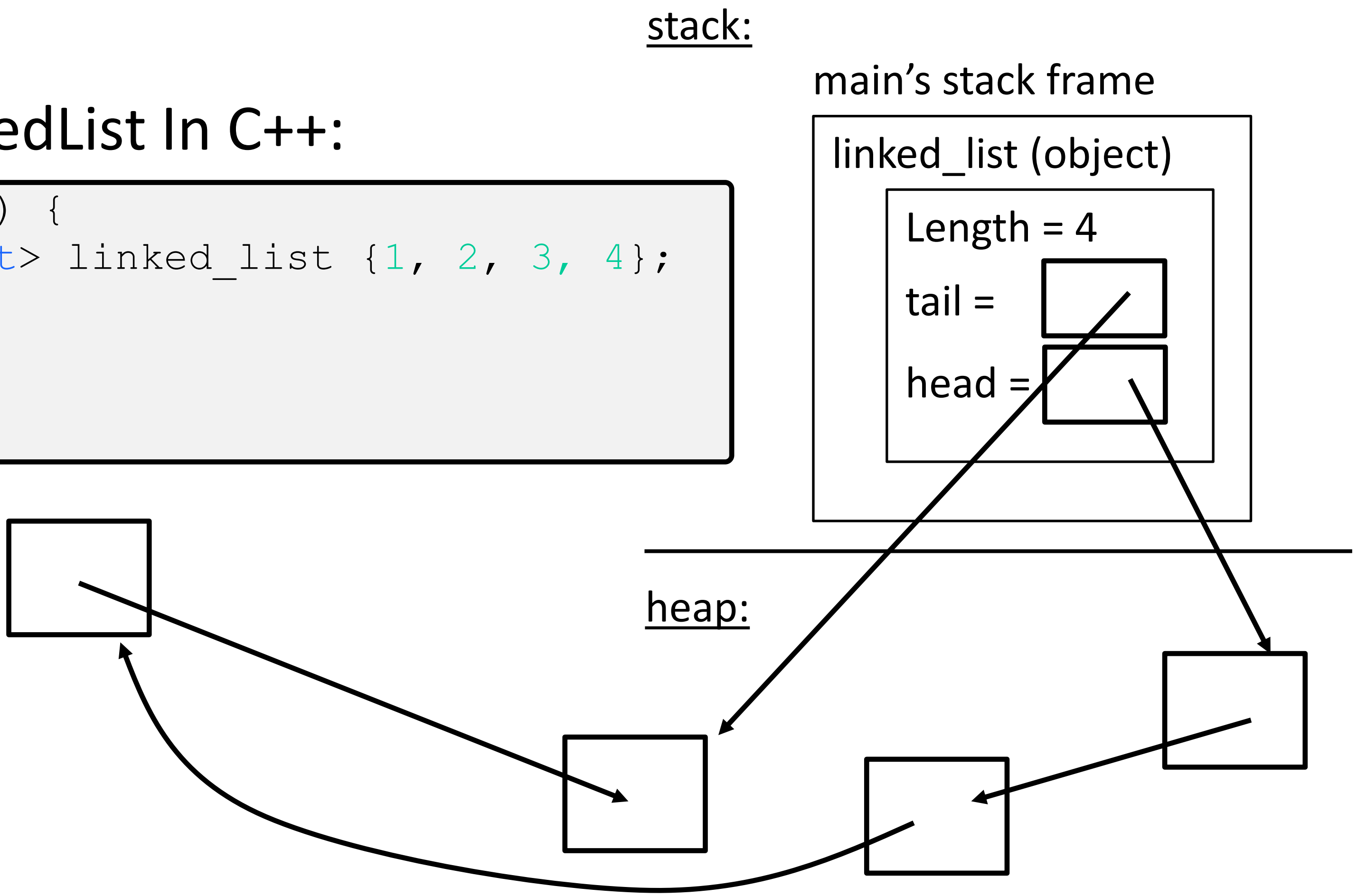
1 2 3

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.

What about other languages?

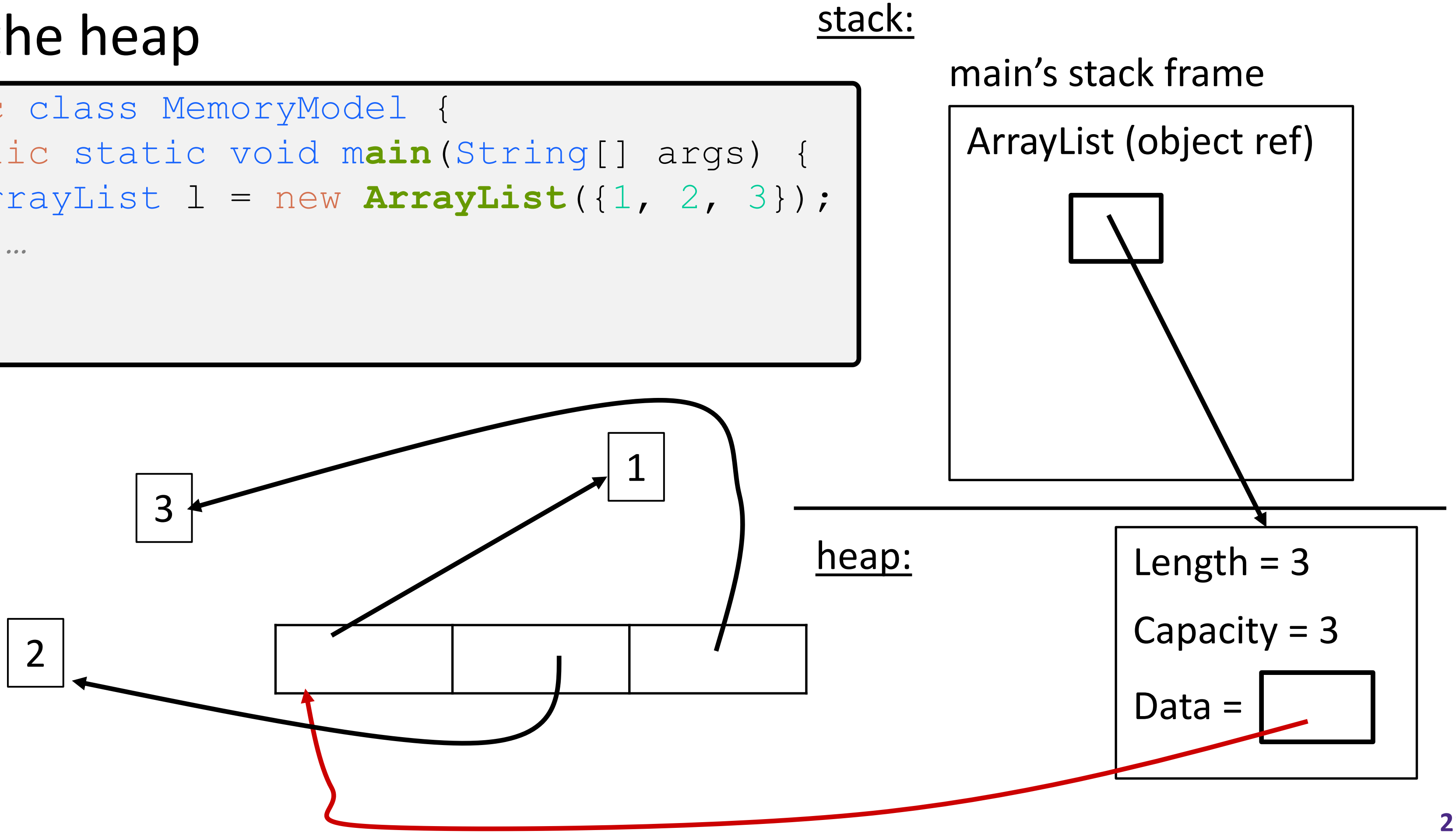
- In C++ (and C, Rust, Zig …) when you declare an object, you have an instance of that object. If you declare it as a local variable, it **exists on the stack**

- 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

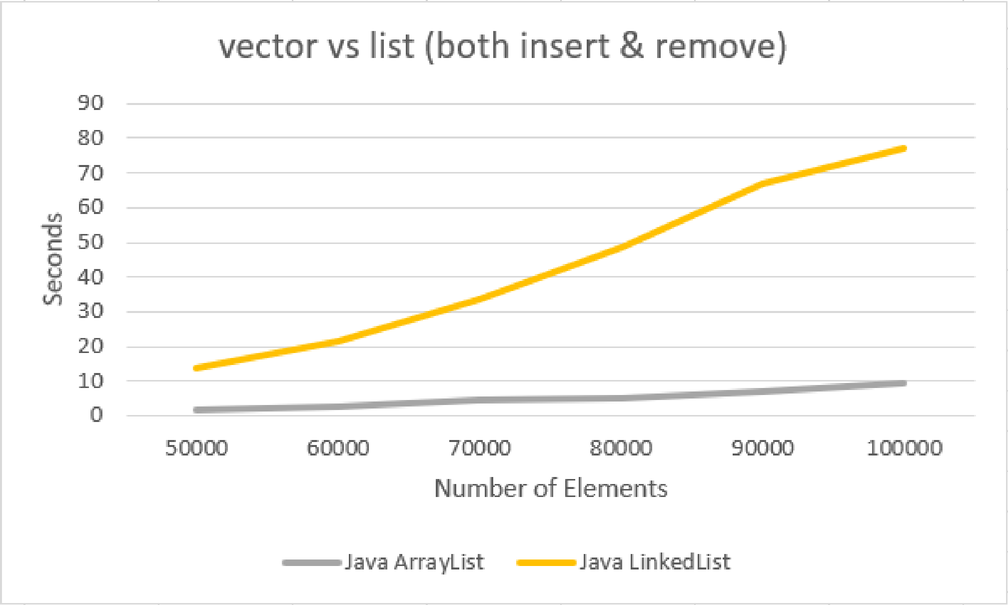
```
public class MemoryModel {
  public static void main(String[] args) {
    ArrayList l = new ArrayList({1, 2, 3});
   // …
  }
}
```



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



Discuss

- 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

Discuss

- 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, …)

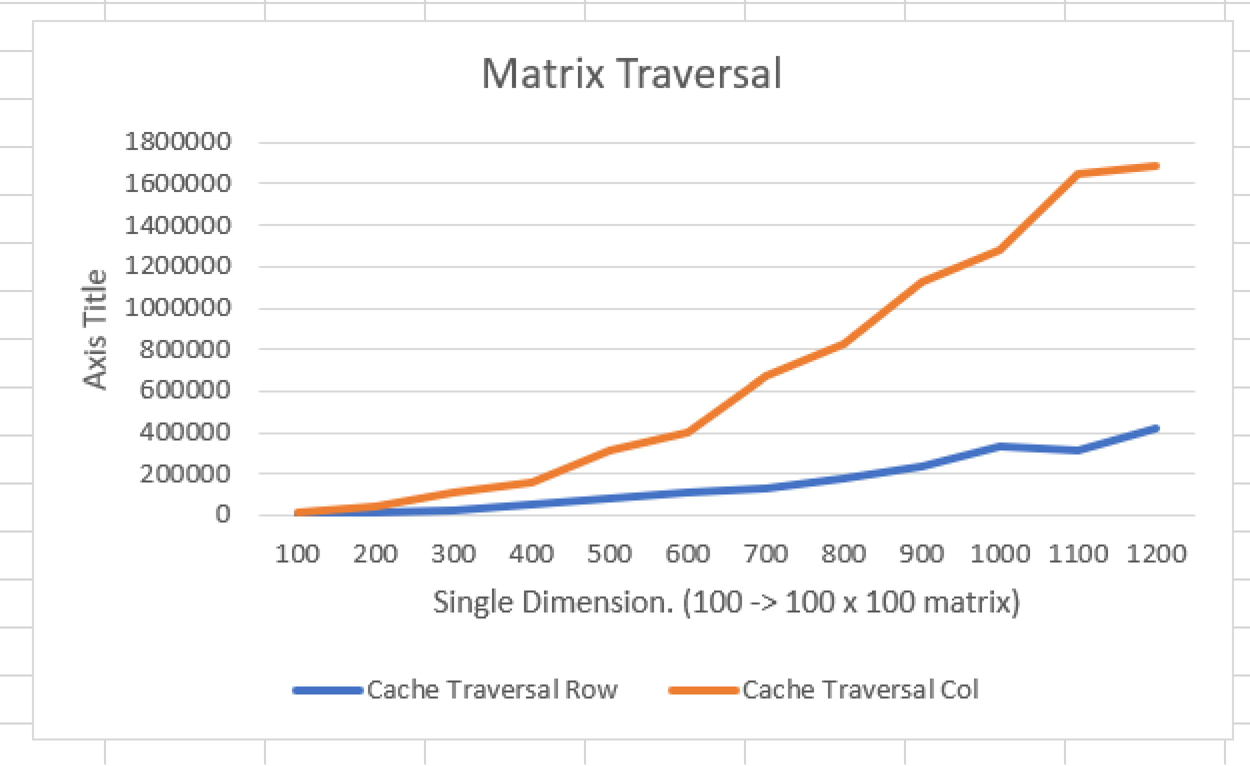
1	5	8	10
11	2	6	9
14	12	3	7
0	15	13	4

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

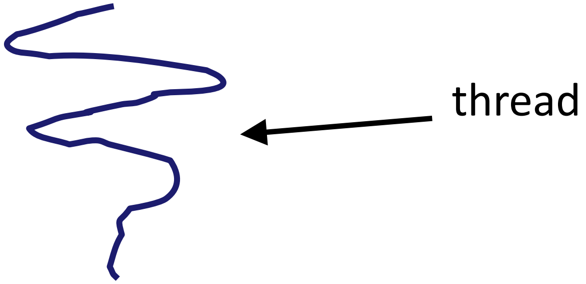
```
public class ICacheExample {
  public static void main(String[] args) {
    ArrayList<A> la = new ArrayList<A>();
    ArrayList<B> lb = new ArrayList<B>();
    ArrayList<C> lc = new ArrayList<C>();
    // …
    for (A item : la) {
       item.compute();
    }
    for (B item : lb) {
       item.compute();
    }
    for (C item : lc) {
       item.compute();
    }
  }
}
```

Lecture Outline

- Intro to Caches
- **Threads High Level**
- Pthreads
- Threads vs processes

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

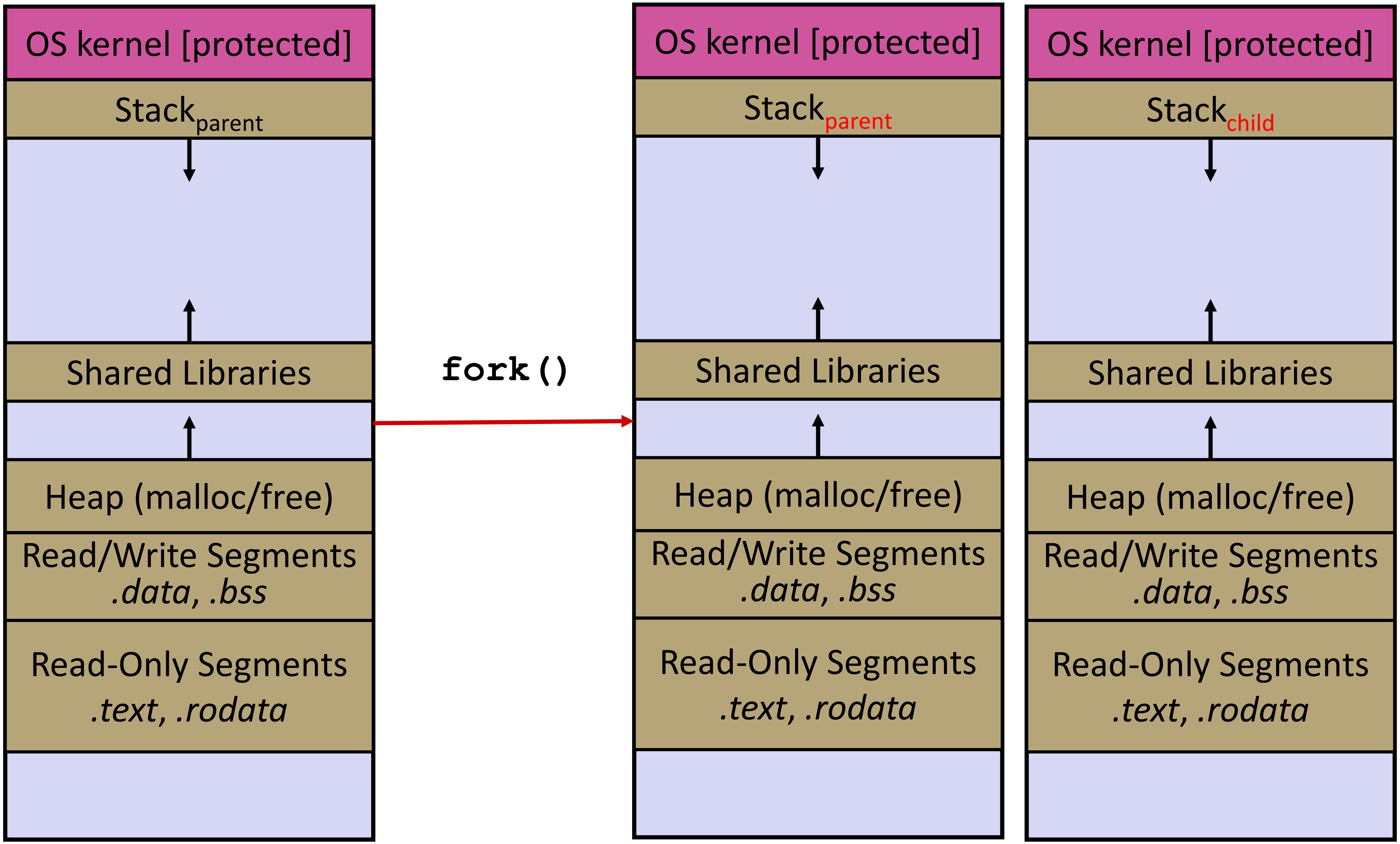


- In most modern OS's:
  - Threads are the *unit of scheduling.*

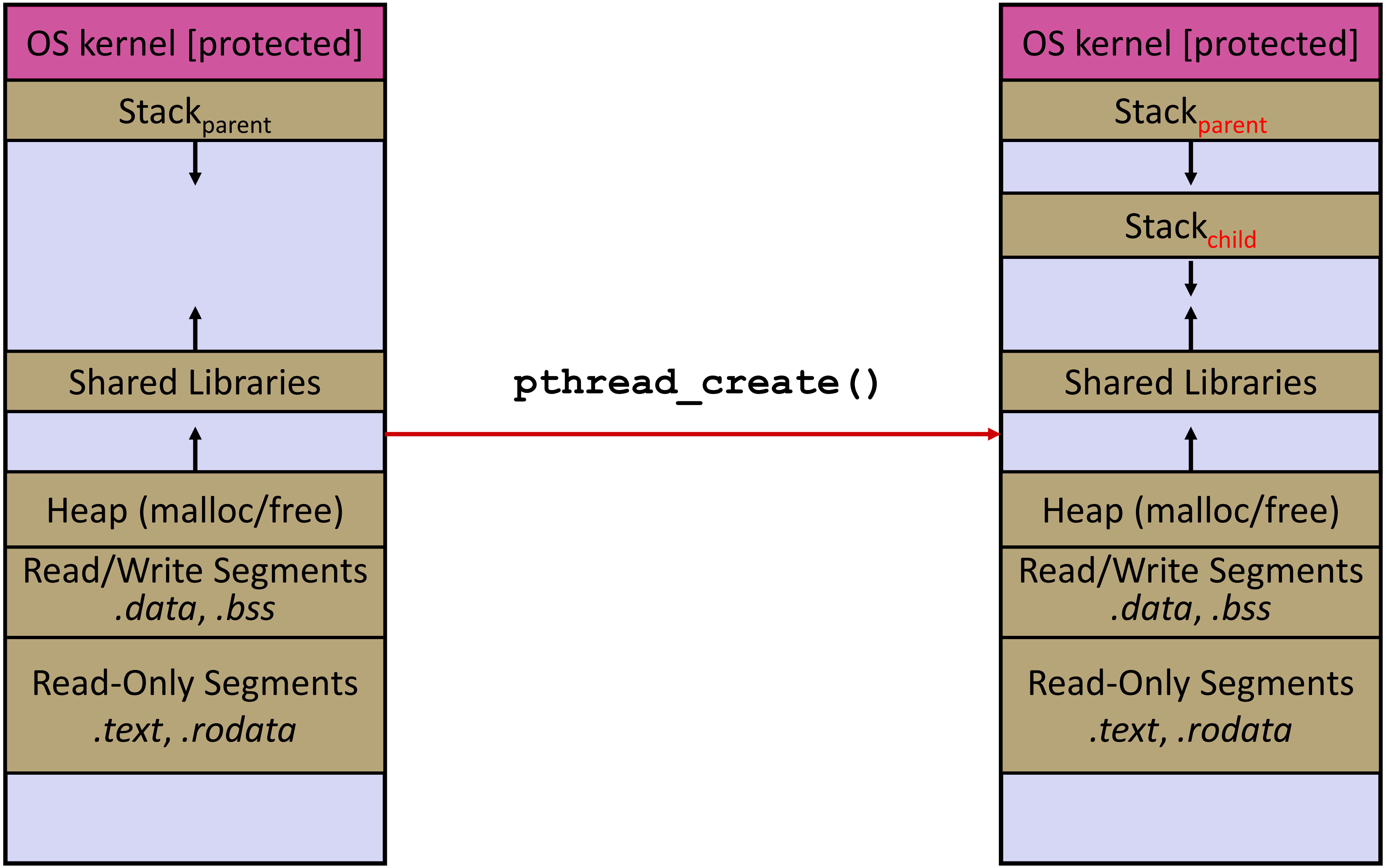
Threads vs. Processes

- In most modern OS's:
  - A 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



OS kernel [protected]

$\text{Stack}_{\text{parent}}$

Shared Libraries

Heap (malloc/free)

Read/Write Segments
.data, .bss

Read-Only Segments
.text, .rodata

`pthread_create()`

OS kernel [protected]

$\text{Stack}_{\text{parent}}$

$\text{Stack}_{\text{child}}$

Shared Libraries

Heap (malloc/free)

Read/Write Segments
.data, .bss

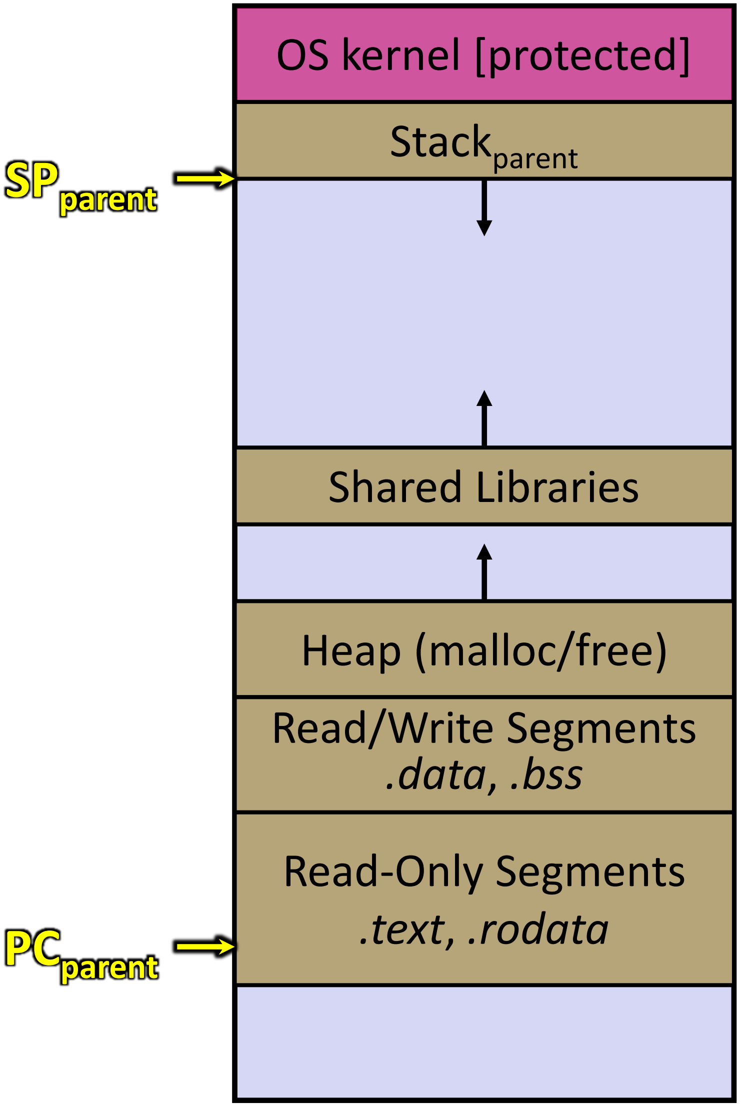
Read-Only Segments
.text, .rodata

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

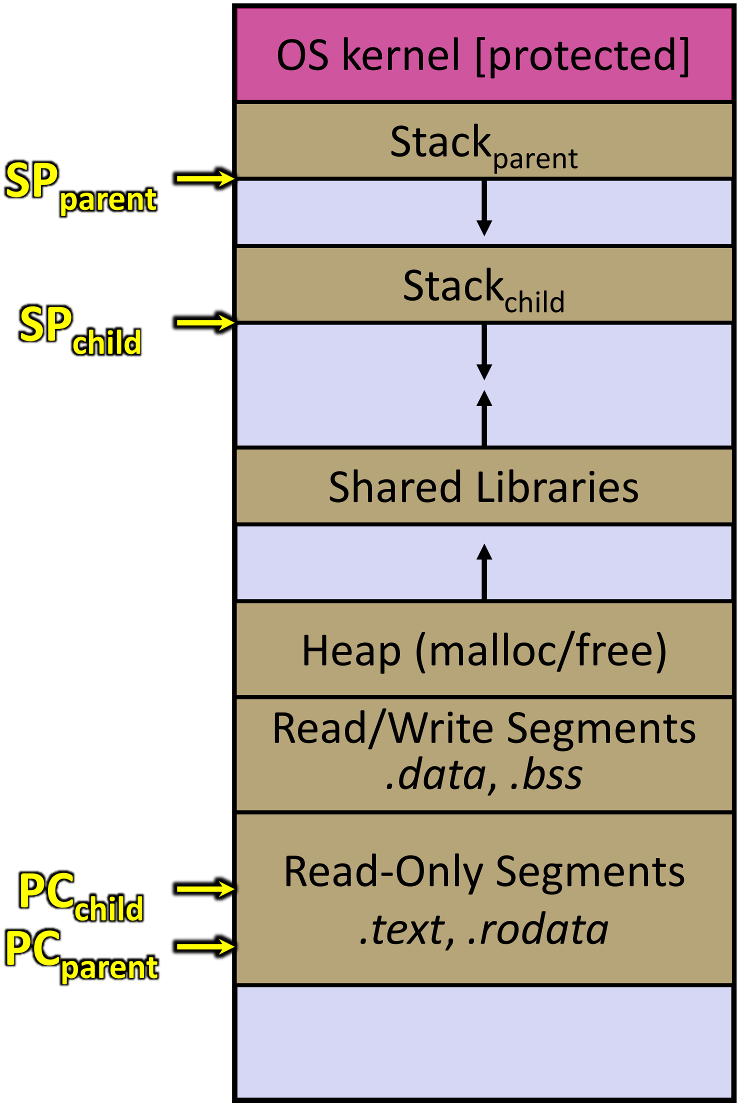
- Analogy: restaurant kitchen
  - Kitchen is process
  - Chefs are threads

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

Lecture Outline

- Intro to Caches
- Threads High Level
- **Pthreads**
- Threads vs processes

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
    - `gcc -g -Wall -pthread -o main main.c`
  - Implemented in C
    - Must deal with C programming practices and style

Creating and Terminating Threads

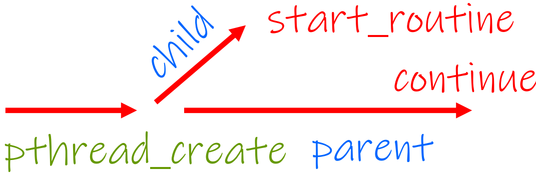
Output parameter.
Gives us a "thread_descriptor"

```
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)`

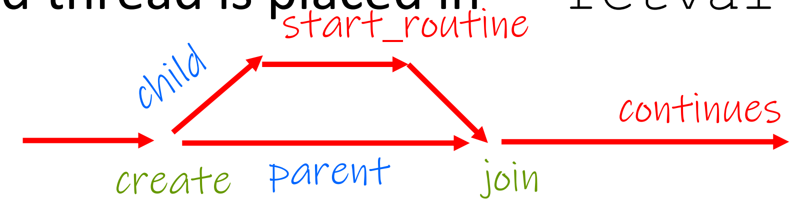


What To Do After Forking Threads?

```
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



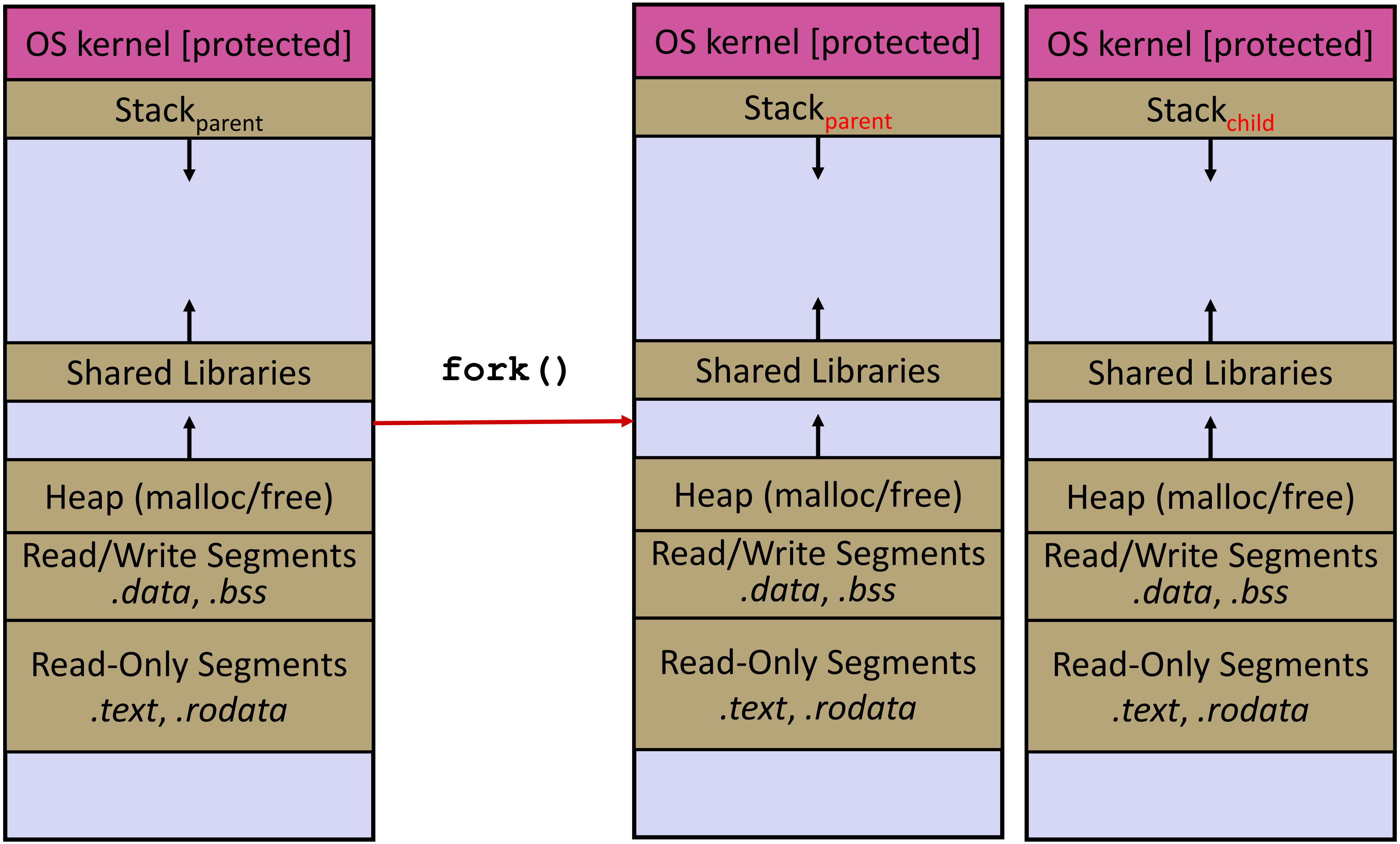
Thread Example

- See `cthreads.c`
    - How do you properly handle memory management?
        - Who allocates and deallocates memory?
        - How long do you want memory to stick around?
    - Threads execute in parallel

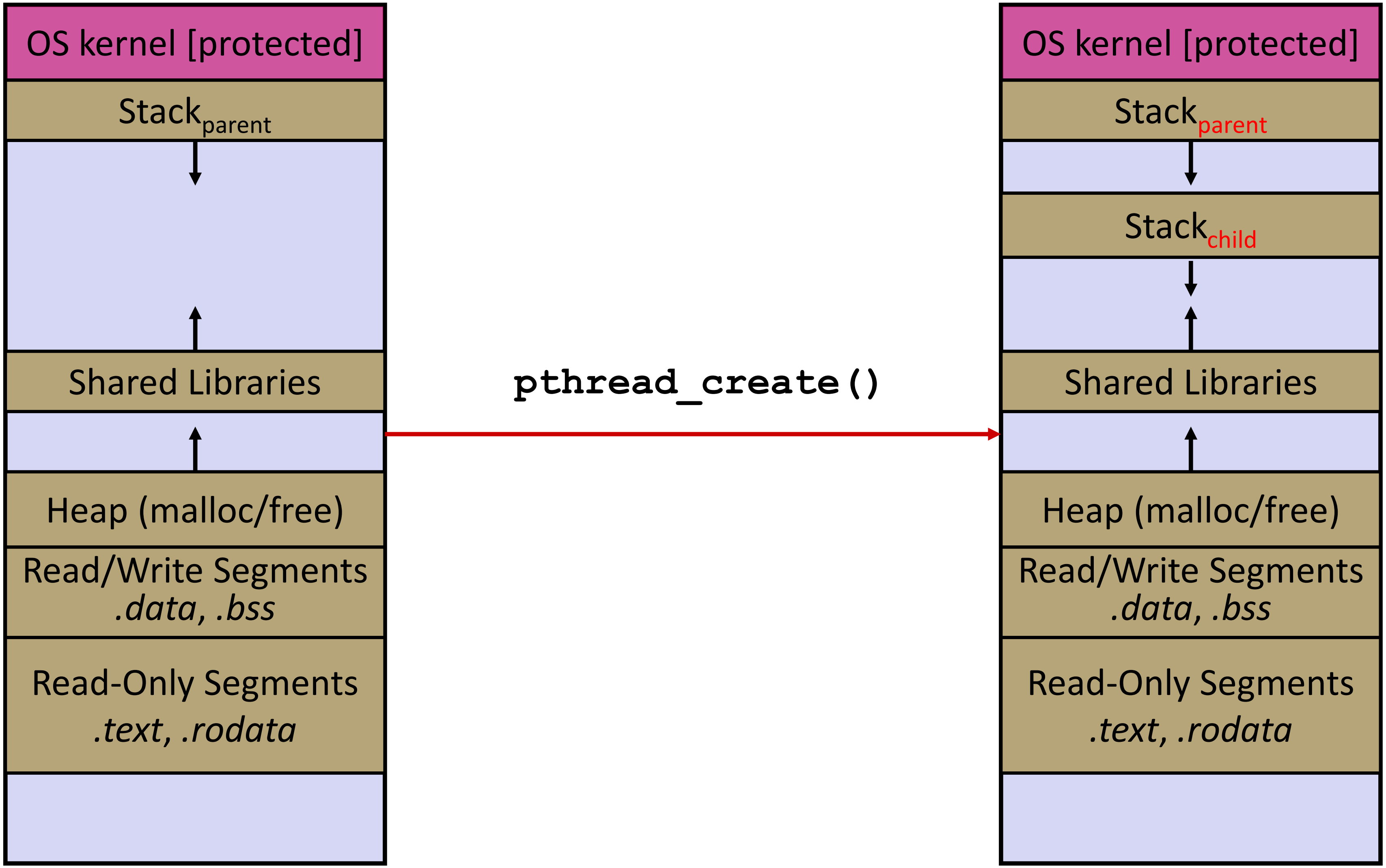
Lecture Outline

- Intro to Caches
- Threads High Level
- Pthreads
- **Threads vs processes**

Threads vs. Processes

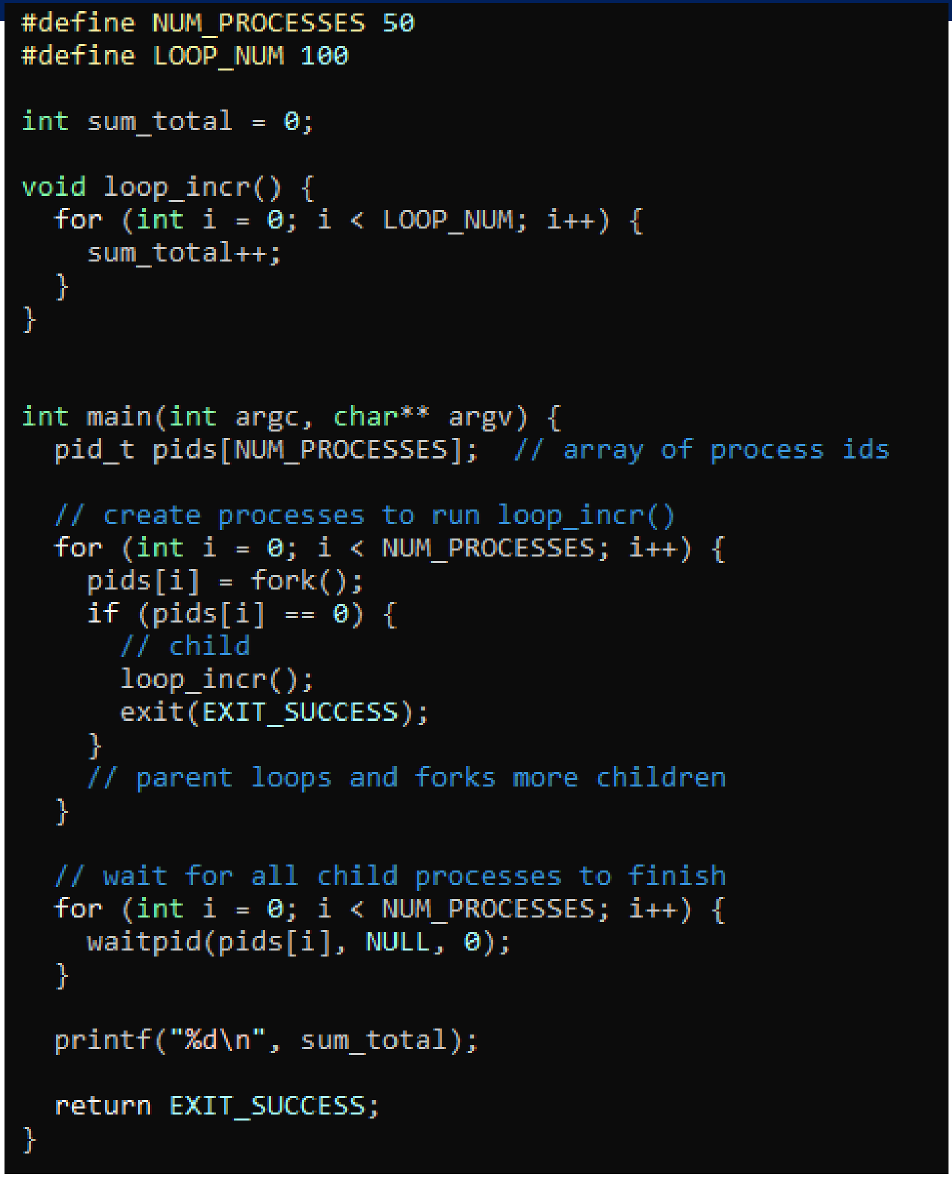


Threads vs. Processes



Discuss

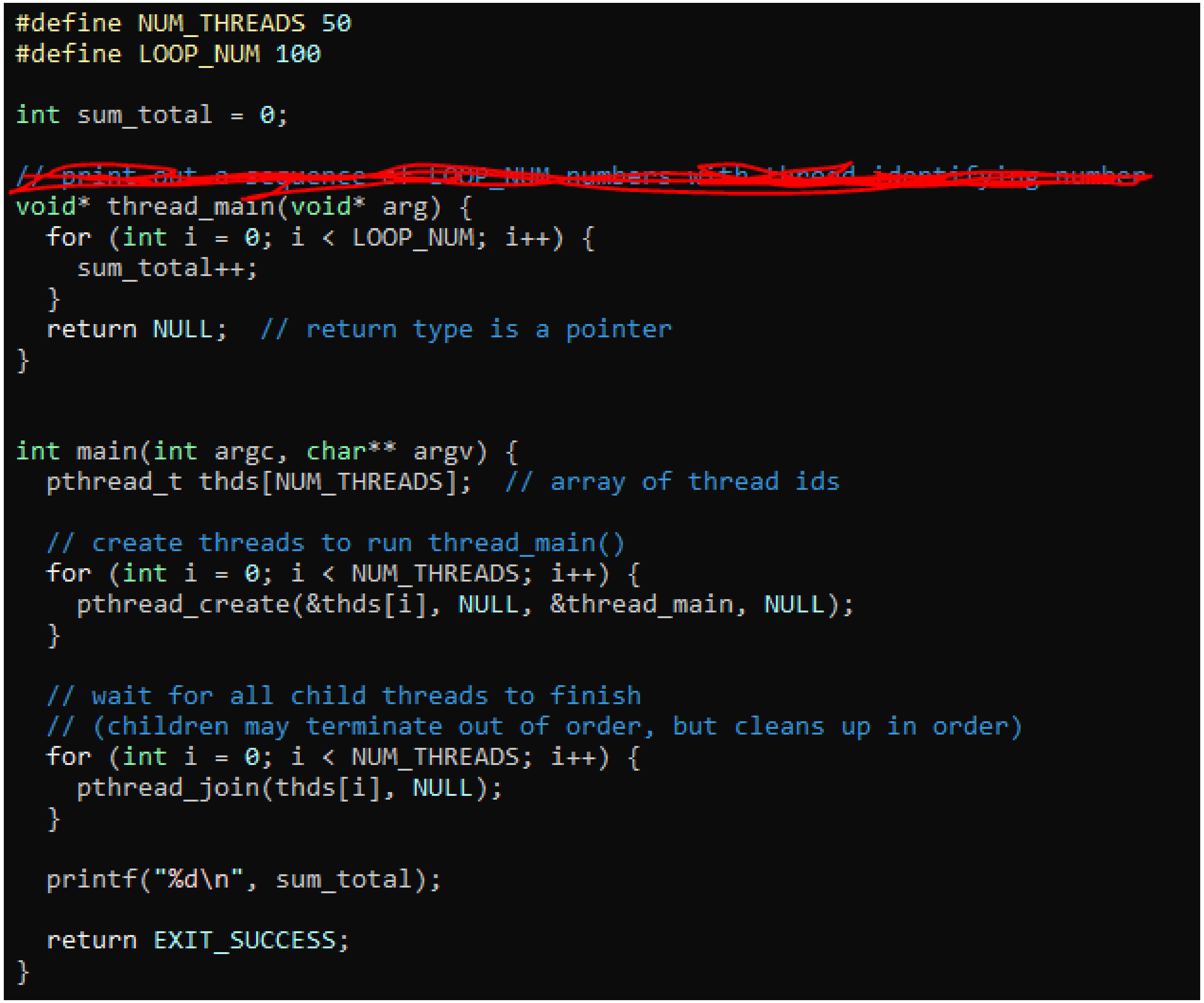
- What does this print?



```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;
}
```

Discuss

- What does this print?



```
#define NUM_THREADS 50
#define LOOP_NUM 100

int sum_total = 0;

~~// print out a sequence of LOOP_NUM numbers with thread identifying number~~
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++) {
    pthread_create(&thds[i], NULL, &thread_main, NULL);
  }

  // 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++) {
    pthread_join(thds[i], NULL);
  }

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

  return EXIT_SUCCESS;
}
```

Demos:

- See `total.c and total_processes.c`
  - Threads share an address space, if one thread increments a global, it is seen by other threads
  - Processes have separate address spaces, incrementing a global in one process does not increment it for other processes

- NOTE: sharing data between threads is actually kinda unsafe if done wrong (we are doing it wrong in this example), **more on this in the second half of the semester**

Process Isolation

- Process Isolation is a set of mechanisms implemented to protect processes from each other and protect the kernel from user processes.
  - Processes have separate address spaces
  - Processes have privilege levels to restrict access to resources
  - If one process crashes, others will keep running

- Inter-Process Communication (IPC) is limited, but possible
  - Pipes via pipe()
  - Sockets via socketpair()
  - Shared Memory via shm_open()

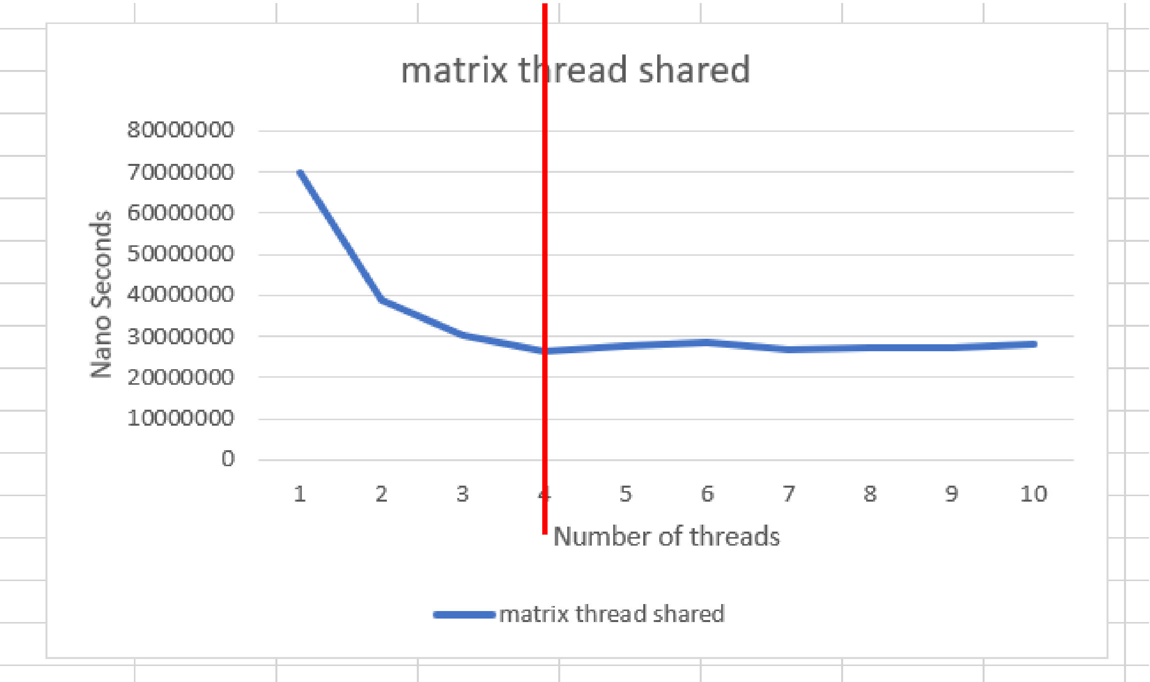
Parallelism

- You can gain performance by running things in parallel
  - Each thread can use another core

- I have a 3800 x 3800 integer matrix, and I want to count the number of odd integers in the matrix

Parallelism

- I have a 3800 x 3800 integer matrix, and I want to count the number of odd integers in the matrix
- I can speed this up by giving each thread a part of the matrix to check!
    - **Works with threads since they share memory**



Diminishing returns

After 4 threads, no gain in speed

why? Machine run on only has 4 cores

How fast is fork()?

- ~ 0.5 milliseconds per fork*
- ~ 0.05 milliseconds per thread creation*
  - 10x faster than fork()

- *Past measurements are not indicative of future performance – depends on hardware, OS, software versions, …
  - Processes are known to be even slower on Windows

Context Switching

- Processes are considered “more expensive” than threads. There is more overhead to enforce isolation

- Advantages:
  - No shared memory between processes
  - Processes are isolated. If one crashes, other processes keep going

- Disadvantages:
  - More overhead than threads during creation and context switching
  - Cannot easily share memory between processes – typically communicate through the file system