Intro to Threads

Operating Systems, Fall 2026

Instructors: Travis McGaha

Head TAs: Shriya Sane

Arnuv Batra

Khoi Dinh

Sydney Fitzgerald

Rania Souissi

Jeffrey Oduman

Abdullah Goher

Monisha Krothapalli

Veer Kakar

Kevin Xue

Shreya Mukunthan

Daniel Price

Wesley Liu

Leo Zhang

pollev.com/tqm

- How are you doing? Anything you want to talk about?

Logistics

- Penn-FAT is out!
  - It is due October 29th in pairs!
  - It is a HUGE assignment. And make sure to put in a lot of elbow grease.
  - Some questions on the midterm will ask about concepts enforced by this assignment.
  - Recitation last week was designed to be helpful for it!

- Penn-fat milestone due by end of the week!
  - Please try to meet with your TA by end of tomorrow if possible tho

Logistics

- Final exam schedule posted:
  - Tuesday, December 15, 2026: 3:00pm to 5:00pm in JMHH G06

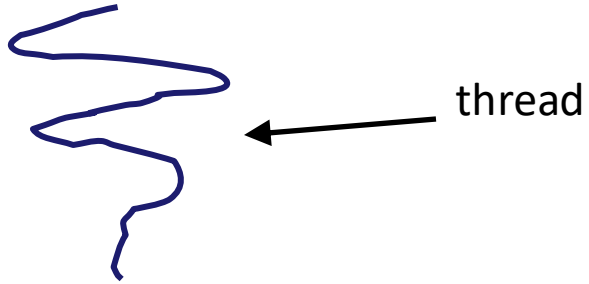
- Midterm coming up in class on October 22nd – details posted today
  - One double sided sheet of notes
  - 90 minutes
  - In lecture
  - Covers everything up to AND including lecture this week on Thursday
  - **RECITATION THIS WEEK IS MIDTERM REVIEW**
  - Midterm review in class Tuesday next week
  - Clobber policy still exists ☺

Lecture Outline

- **What is a thread?**
- pthreads
- Processes vs threads
- Thread Interleaving & Sequential Consistency
- Benefits of Concurrency
- Processes vs Threads

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

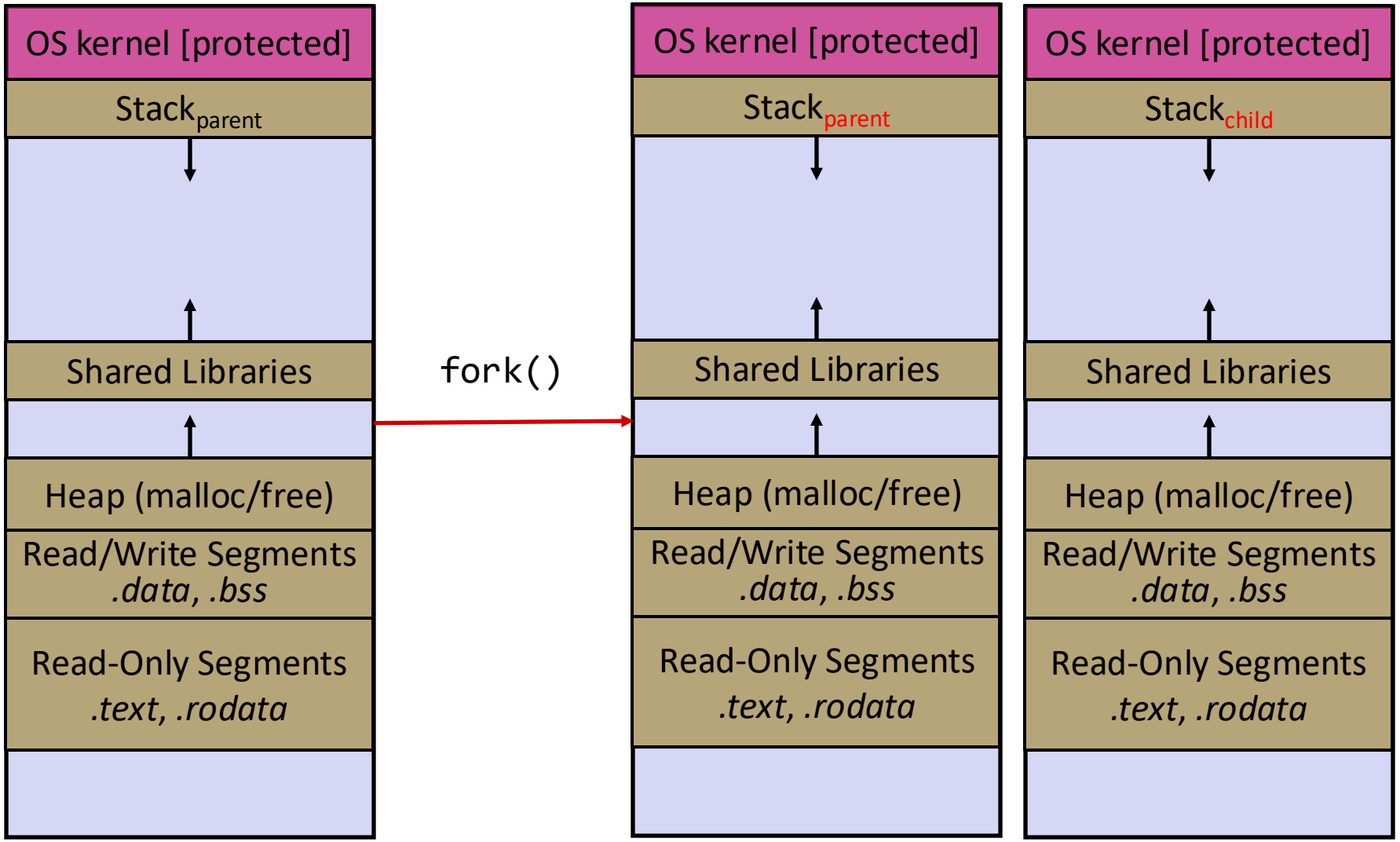


- In most modern OS's:
  - Threads are the *unit of scheduling.*

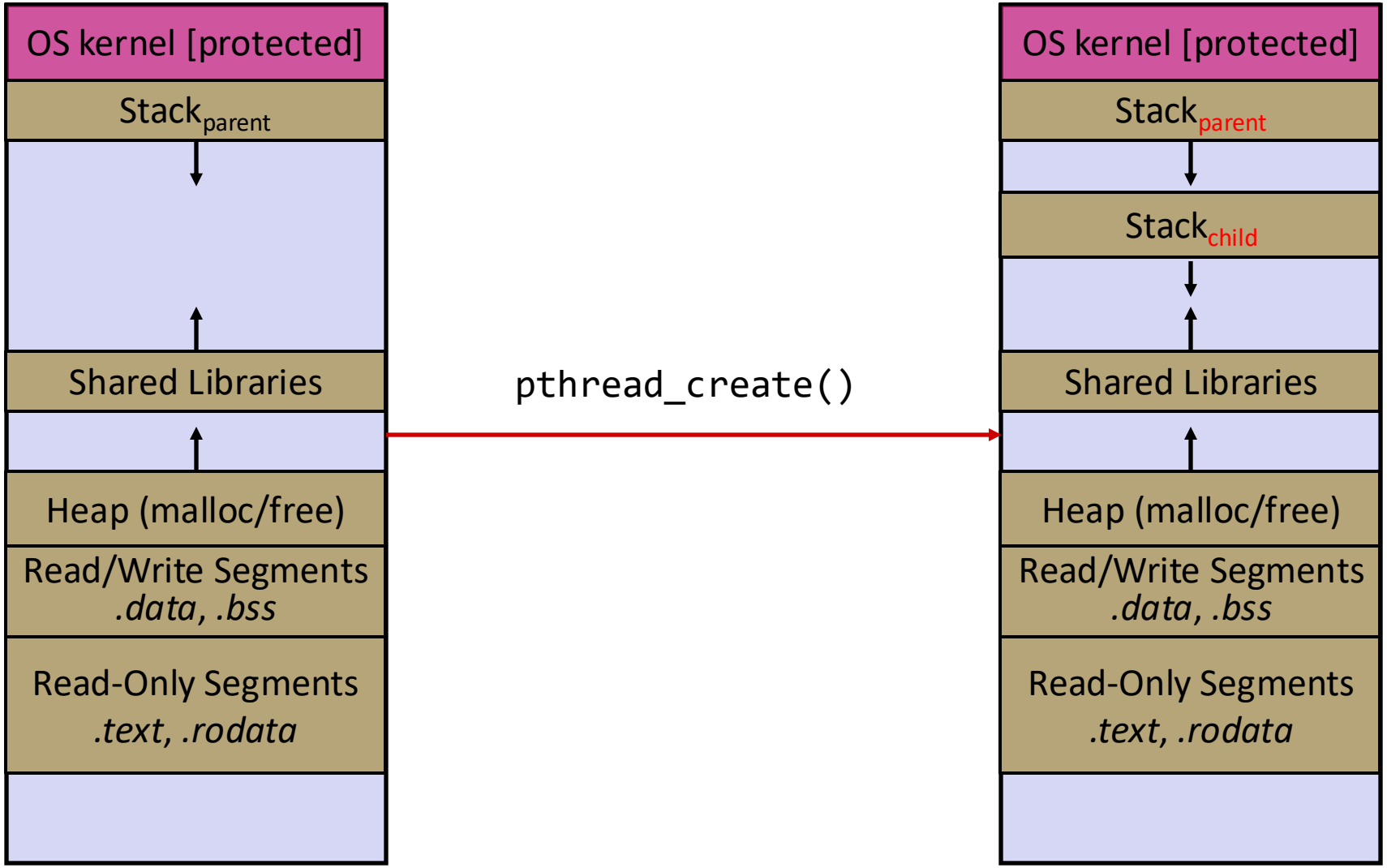
Threads vs. Processes

- 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

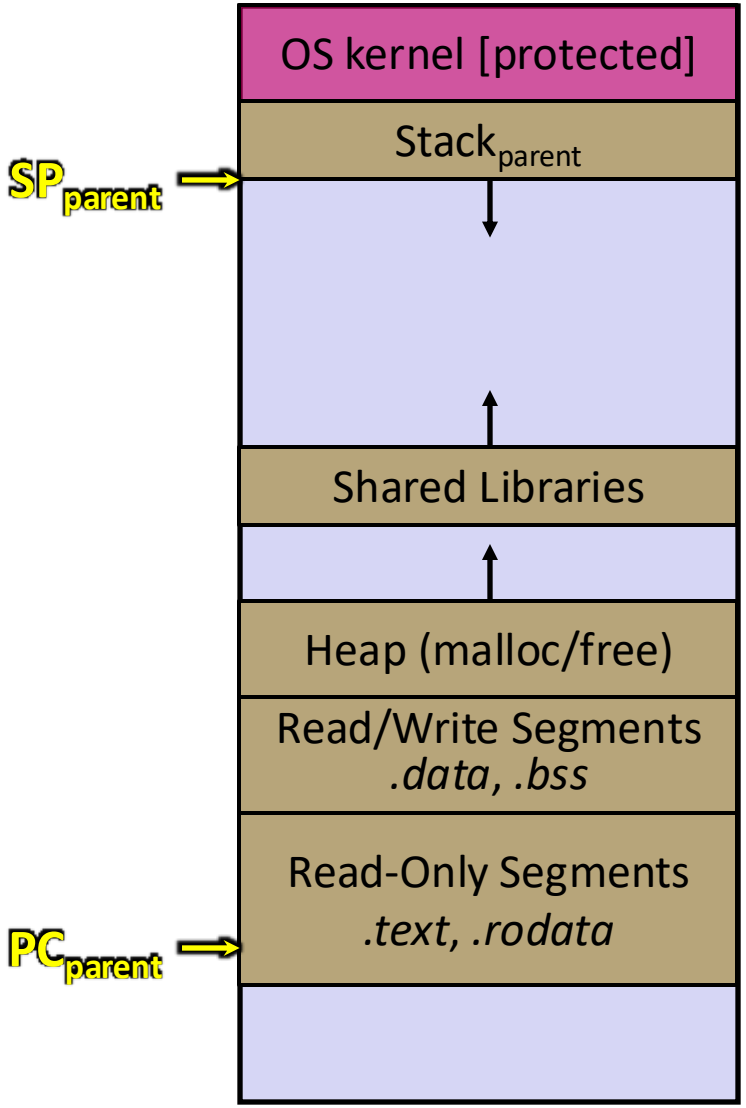


Threads

- 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

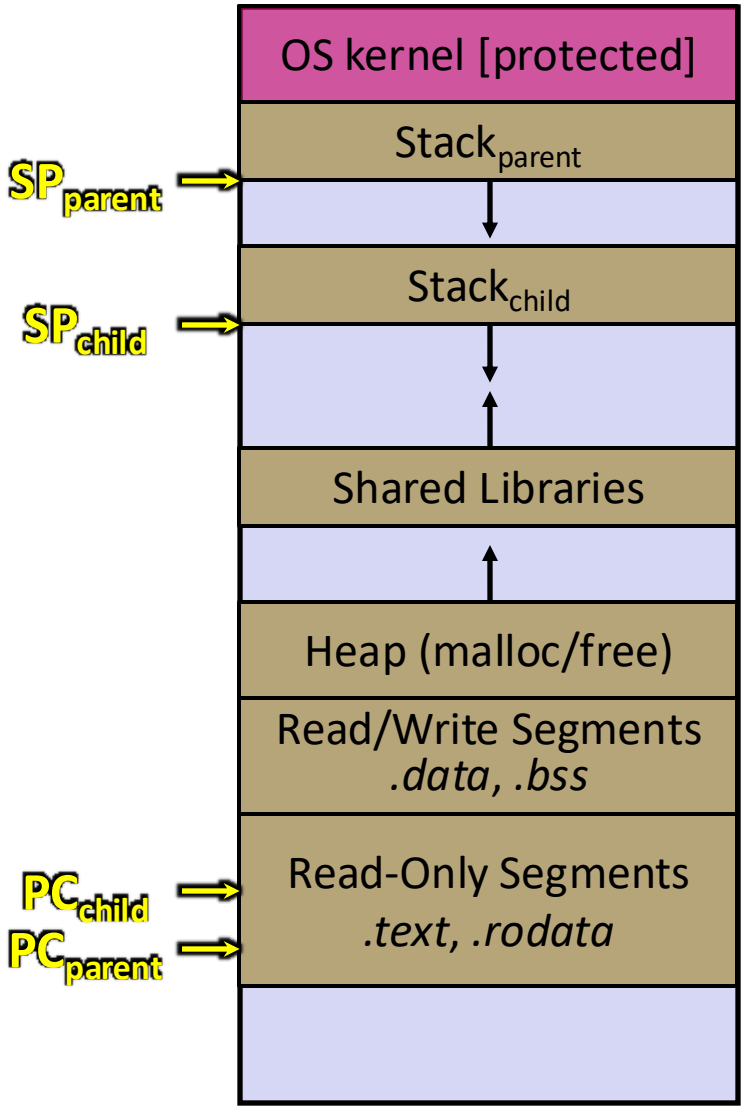
- Analogy: restaurant kitchen
  - Kitchen is process
  - Chefs are threads

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

Process Control Blocks in Linux

- As we said, threads are the unit of scheduling. How does an OS track threads?

- In modern Linux, a process control block (PCB) is called a “task” struct
  - Instead of having one struct per process, it instead has one struct per thread
  - Each task struct has some information about which memory space it belongs to, which process ID it has, etc.
  - Threads in the same process have the same memory information and PID, but things like register value and thread ID will be different.

Lecture Outline

- What is a thread?
- **pthreads**
- Processes vs threads
- Thread Interleaving & Sequential Consistency
- Benefits of Concurrency
- Processes vs Threads

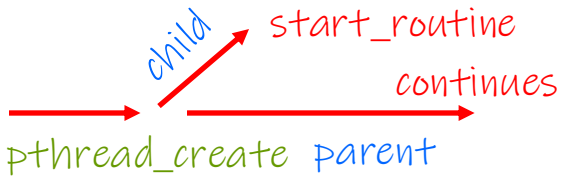
POSIX Threads (pthreads)

- 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 `clang-15` command
    - `clang-15 -g -Wall -pthread -o main main.c`
  - Implemented in C
    - Must deal with C programming practices and style

Creating Threads

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

- Creates a new thread with attributes `*attr`
- Returns `0` on success and an error number on error (can check against error constants)
- The new thread runs `start(arg)`



This uses our previous conception of child vs parent metaphor, but really, they are not treated this way. There is no hierarchy of threads. They are more like siblings.

pthread_create in Reality

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

- `pthread_t* thread`
  - Output parameter: gives us a thread identifier
    - Varies from OS to OS, in Linux it is an unsigned long in others, a struct
- `const pthread_attr_t* attr`
  - An struct detailing the attributes that the thread will take on. NULL for default attributes
- `void* (*start)(void*)`
  - Function that the newly created thread will commence executing from
    - e.g., `void *thread_fn(void *arg)`
- `void* arg`: the *single* argument that will be passed into start

pthread_create in Reality

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

- `void* arg`: the *single* argument that will be passed into start

- How can we pass in multiple arguments to the start routine?
  - Create a struct with all the args and pass a pointer to it!

- No variable length arguments in C
  - C++ has them but those aren't POSIX threads

pthread_create is *not* fork

- Unlike fork, the ***newly*** created execution context starts executing the ***given function***. **It does not continue from pthread_create!**

```
void* print_hello(void* arg) {
    printf("Hi I am a thread!\n");
    return NULL;
}


int main() {
    pthread_t new_thread;
    pthread_create(&new_thread, NULL, print_hello, NULL);
    printf("Yeah but I'm the main thread.\n");
   //more down here...
}
```

new_thread will start its execution from here

When it returns, it will ***not return to main***

The lifetime of new_thread is contained to this function (and the functions it calls).

pthread_create is *not* fork

```c
int x = 10;

void* print_hello(void* arg) {
    printf("X is the number %d.\n", x);
    return NULL;
}


int main() {
    pthread_t new_thread;
    pthread_create(&new_thread, NULL, print_hello, NULL);
    printf("Yeah but I'm the main thread.\n");
   //more down here...
}
```

The lifetime of new_thread is contained to this function

But it still has access to the same variables as the main thread

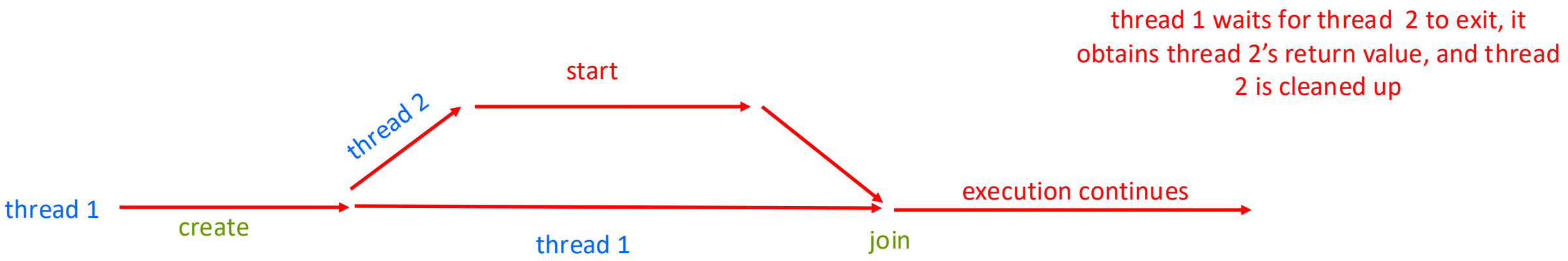
THEY ARE NOT COPIES!

All threads share the same address space.

We Created a Thread, Now What?

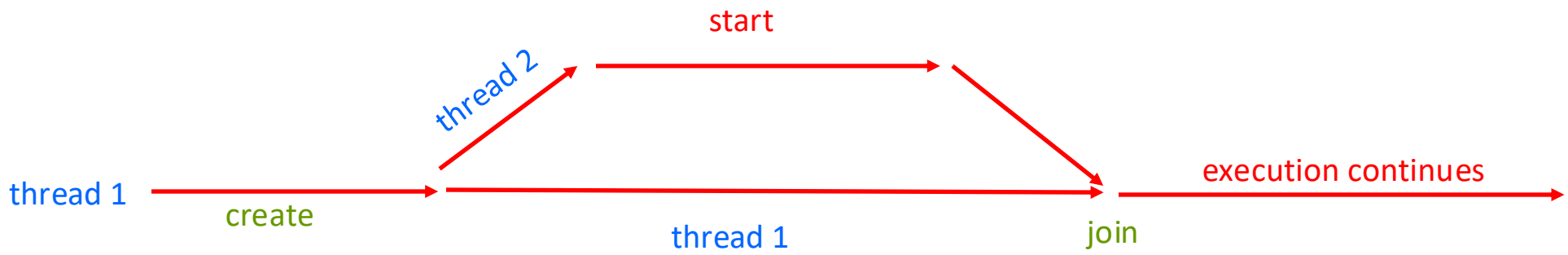
```
int pthread_join(pthread_t thread, void** retval);
```

- The calling thread waits for the thread specified by `thread` to terminate, and as the name says, joins the two executions stream into one (hence, "join")
- You can think of it as the thread equivalent of `waitpid()`, although it really isn't
- The exit status of the terminated thread is placed in `**retval`



No more, parent and child. JUST THREADS!

We Created a Thread, Now What?



Once a new thread is created, we are at the mercy of the scheduler.
We can not assume when it will run.

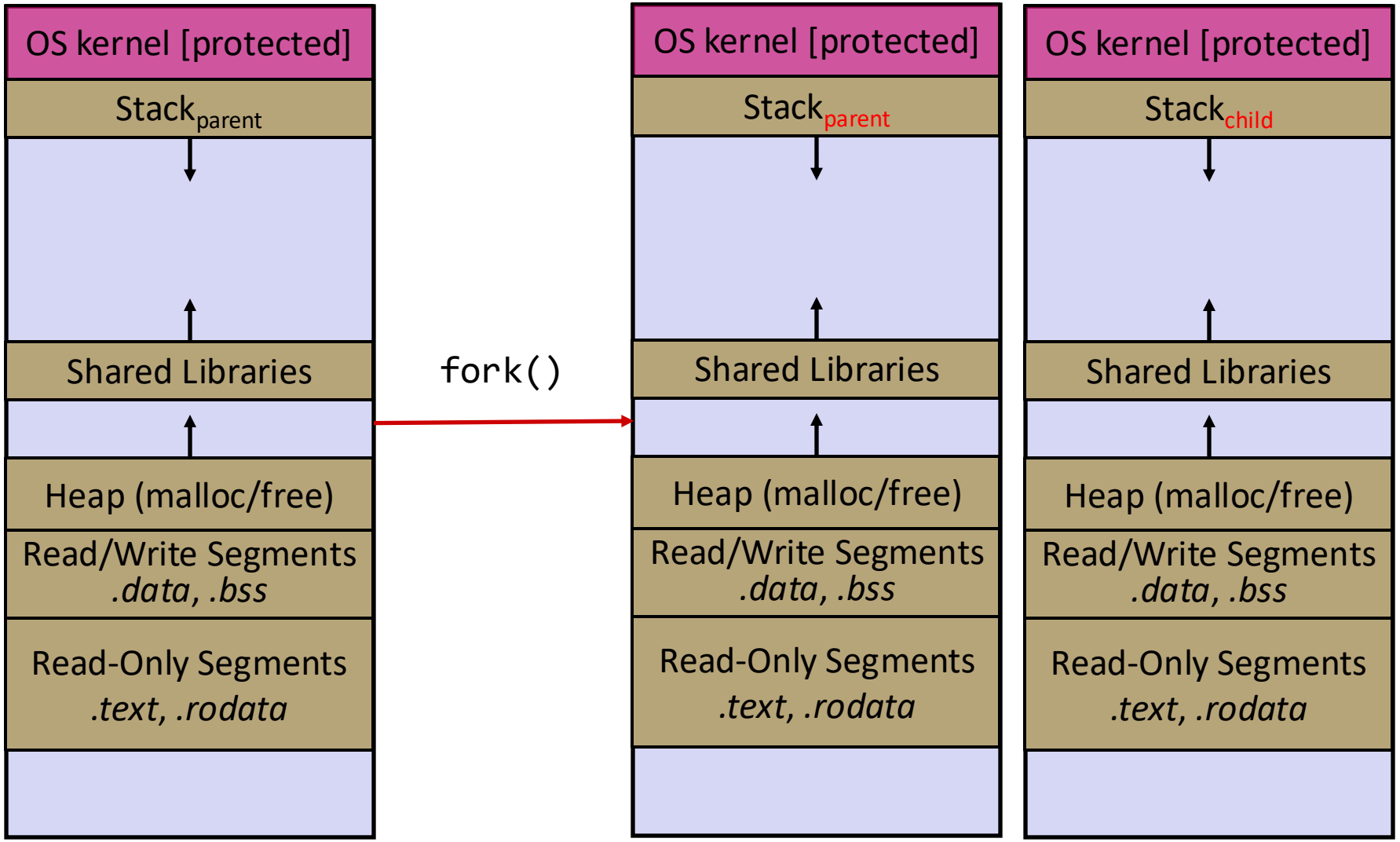
Thread Example

- 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

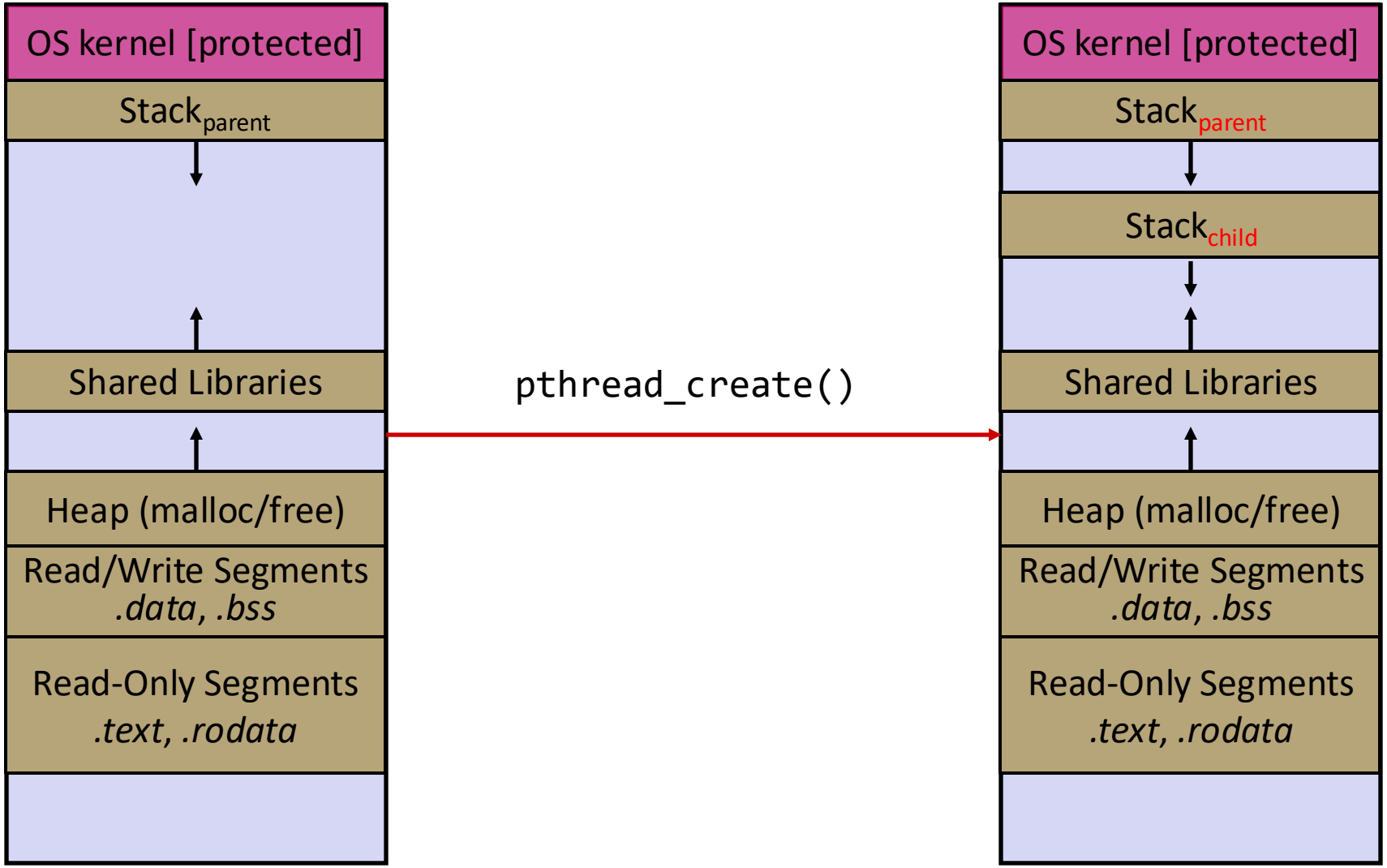
Lecture Outline

- Threads High Level
- pthreads
- **Processes vs threads**
- Thread Interleaving & Sequential Consistency
- Benefits of Concurrency
- Processes vs Threads

Threads vs. Processes



Threads vs. Processes



Note: want to change where the stacks are for different threads? One of the many attributes you can set. ☺

Poll Everywhere

pollev.com/tqm

- What does each process print?
- What is the ultimate sum_total?

```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++;
  }
  printf("Process ID: %d with sum total of %d.\n", getpid(), 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);
    }
  }
  // wait for all child processes to finish
  for (int i = 0; i < NUM_PROCESSES; i++) {
    waitpid(pids[i], NULL, 0);
  }

  printf("The ultimate sum total is %d\n", sum_total);
  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- What does each thread print?
- What is the ultimate sum total?

- Note: in Linux, we can grab a thread's id using `gettid()`

```c
#define NUM_THREADS 50
#define LOOP_NUM 100

int sum_total = 0;

void *loop_incr(void *arg) {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  printf("Thread ID: %d with sum total of %d.\n", gettid(), sum_total);
  return NULL;
}


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

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

  // wait for all non-main threads to finish
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_join(thds[i], NULL) != 0) {
      fprintf(stderr, "pthread_join failed\n");
    }
  }

  printf("The ultimate sum total is %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 unsafe if done wrong (we are doing it wrong in this example…). **More on this later!**

Revisiting: 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 (More on this later)
  - Processes have privilege levels to restrict access to resources (Not covered in this class)
  - If one process crashes, others will keep running (The scheduler is agnostic, just one less thing to schedule. ☺)

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

- Threads on the other hand, can see *everything* within the process.

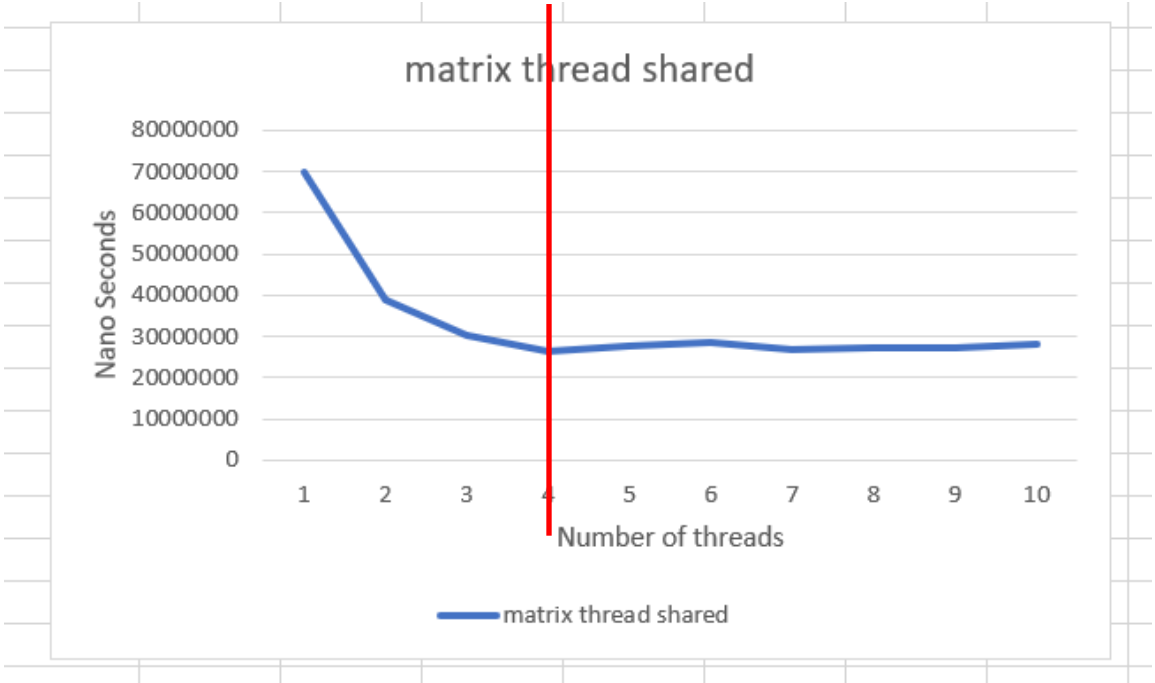
Parallelism

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

- Let’s say 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

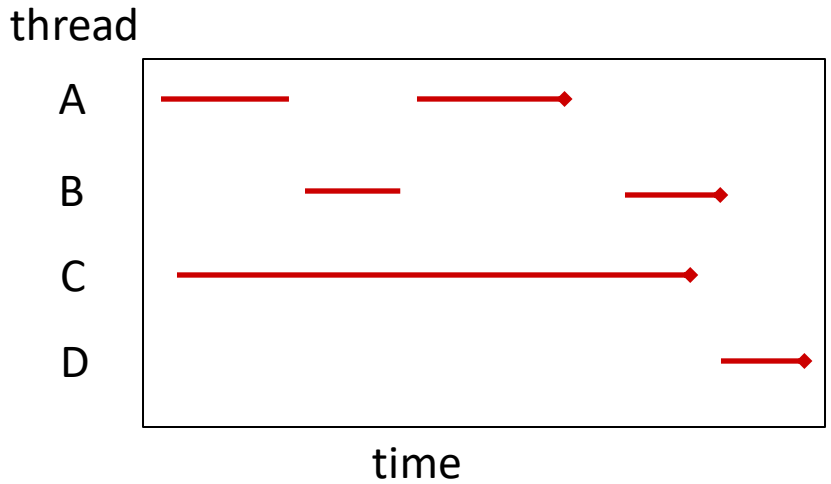
Parallelism vs Concurrency

- Two commonly used terms (often mistakenly used interchangeably)

- **Concurrency**: When there are one or more "tasks" that have overlapping lifetimes (between starting, running and terminating)
  - That these tasks are both running within the same **period**

- **Parallelism**: when one or more "tasks" run at the same **instant** in time

- Consider the lifetime of these threads. Which are concurrent with A? Which are parallel with A?



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

Lecture Outline

- Threads High Level
- pthreads
- Processes vs threads
- **Thread Interleaving & Sequential Consistency**
- Benefits of Concurrency
- Processes vs Threads

Poll Everywhere

pollev.com/tqm

- Which output is **not** possible for this program?

```
void* thrd_fn(void* arg) {
  int* ptr = (int*) arg;
  printf("%d\n", *ptr);
  return NULL;
}

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

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

Outputs to consider:

1
2

2
2

1
1

2
1

Visualization

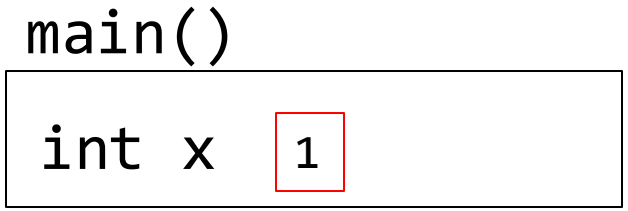
```c
int main() {
  int x = 1;
  pthread_create(...);
  x = 2;
  pthread_create(...);

  pthread_join(...);
  pthread_join(...);
}
```

```c
thrd_fn() {
  printf(*ptr);
  return NULL;
}
```

```c
thrd_fn() {
  printf(*ptr);
  return NULL;
}
```

Visualization: Memory

- The variable x is shared across all threads.




```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

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

Visualization: Memory

- The variable x is shared across all threads.

```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

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

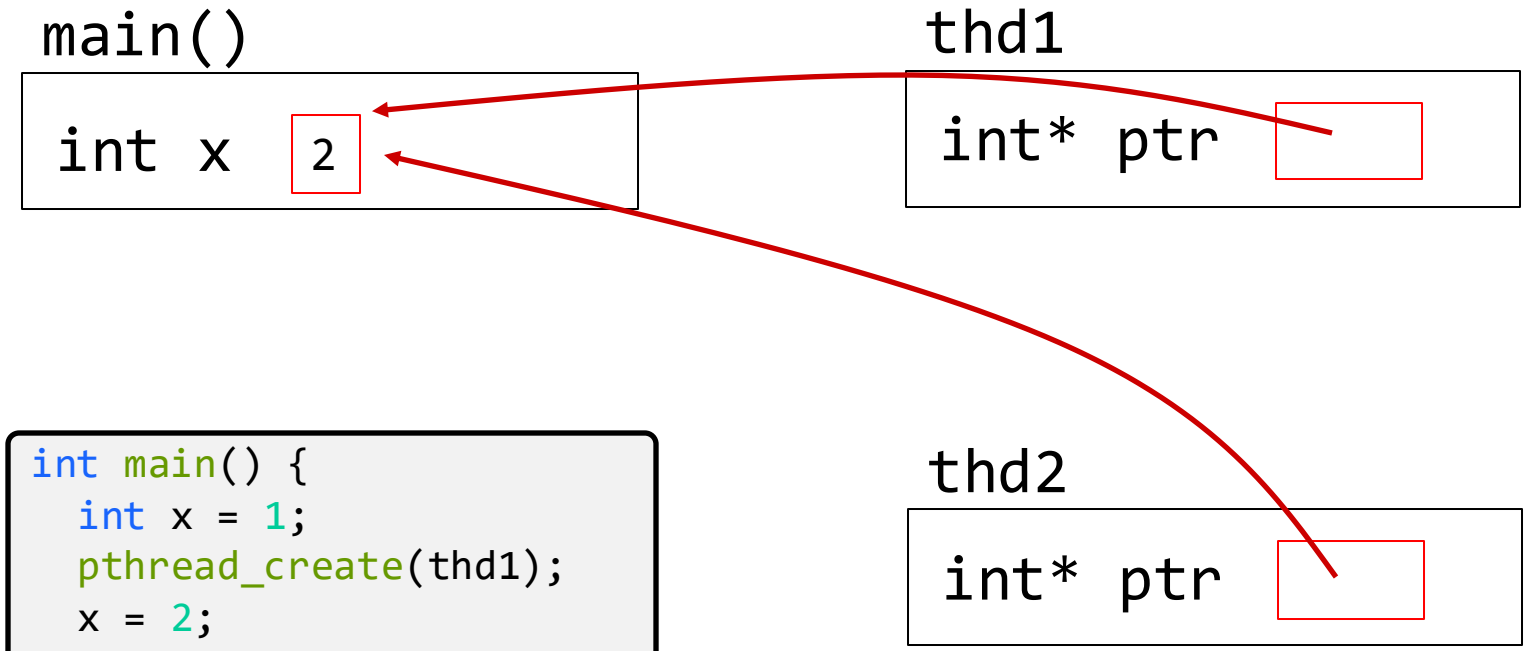
Visualization: Memory

- The variable x is shared across all threads.

```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

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

Visualization: Memory

- The variable x is shared across all threads.




```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

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

We only know that these threads have been created. We know nothing about their execution.

Sequential Consistency

- Within a single thread, we assume* that there is sequential consistency. That the order of operations within a single thread are the same as the program order.

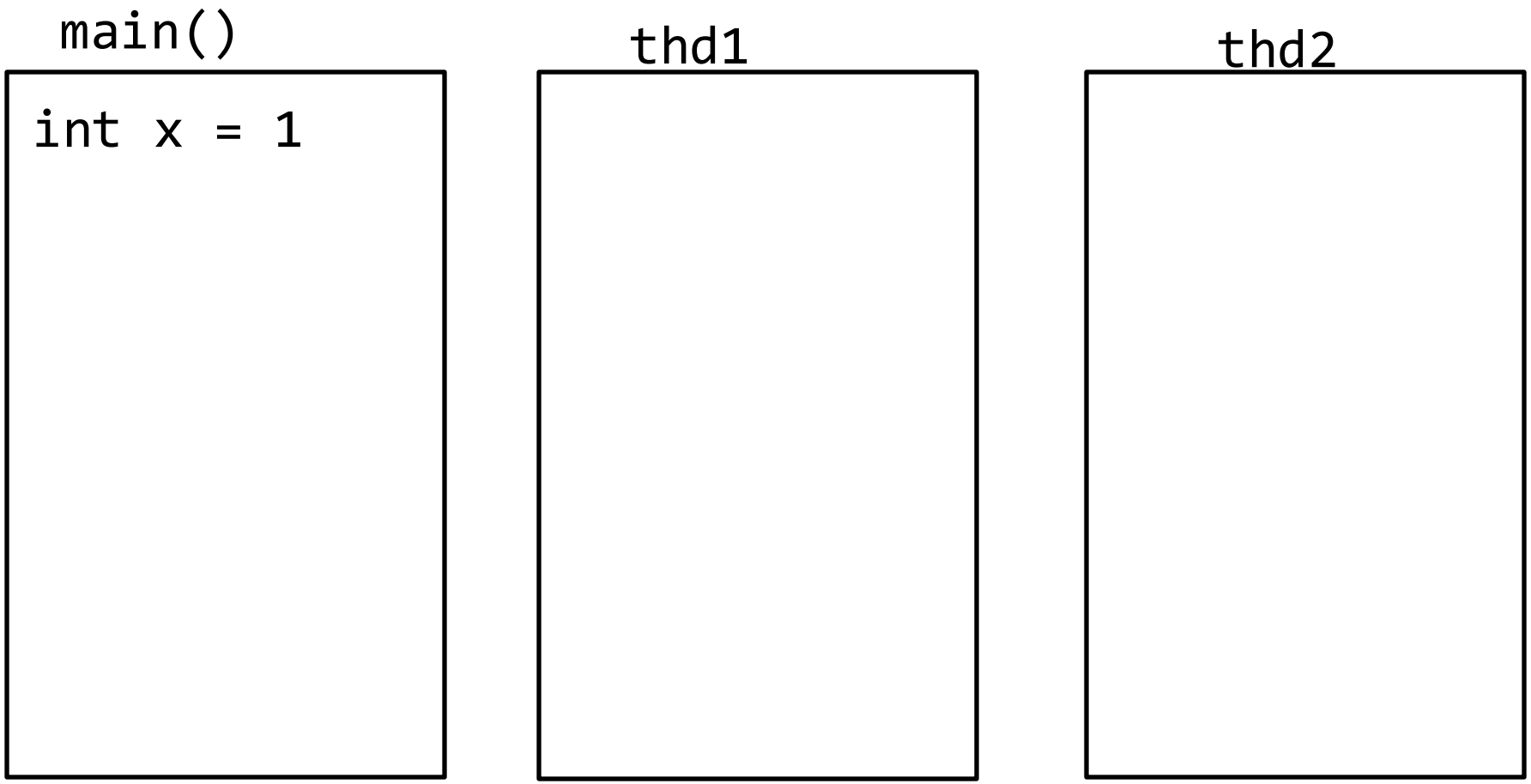
main()

```
int x = 1

create thd1

x = 2

create thd2
```

In the main thread,
- x is set to 1
- then thread 1 is created
- then x is set to 2
- then thread 2 is created

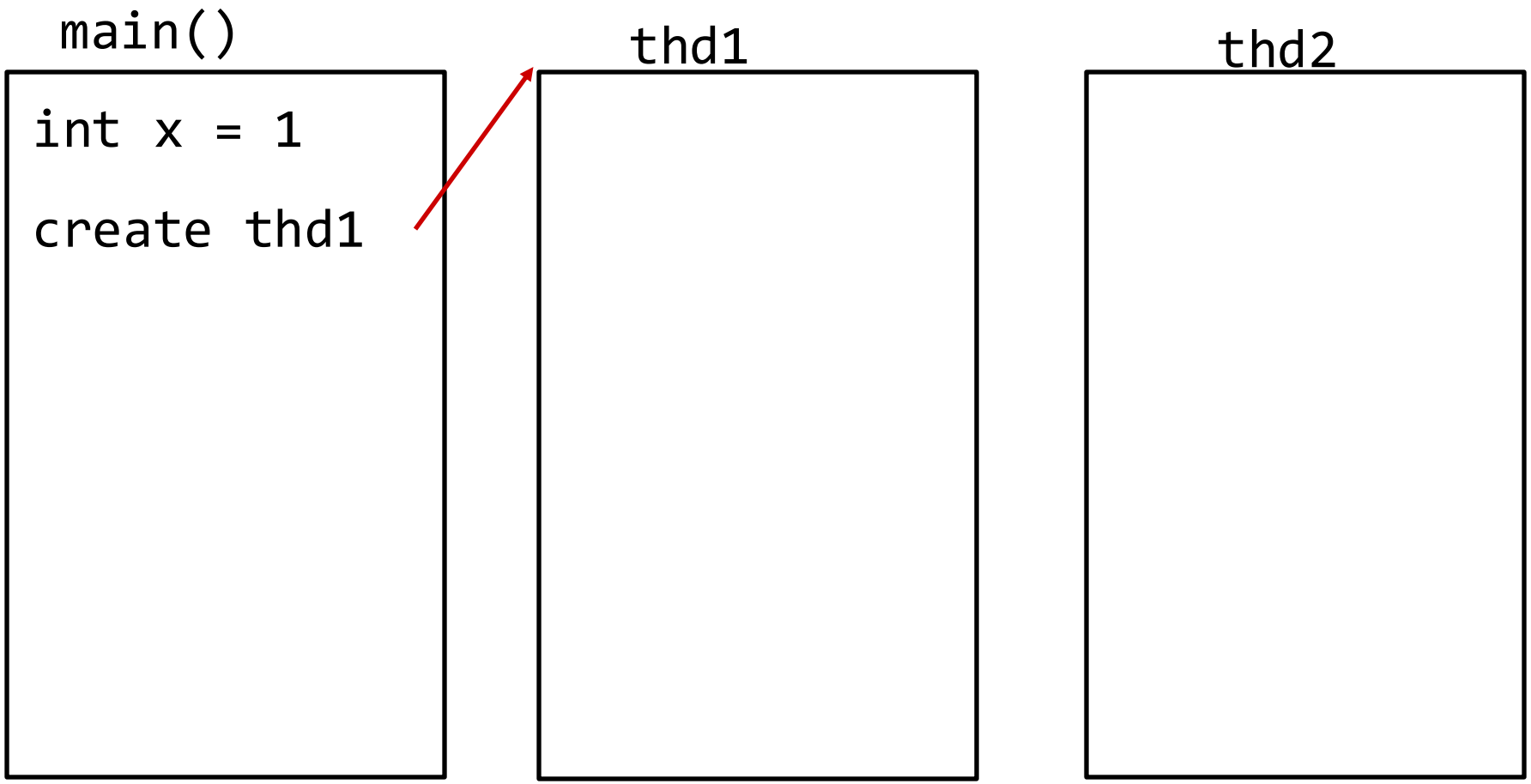
Visualization: Ordering

- Threads run concurrently; we can't be sure of the ordering of things across threads.



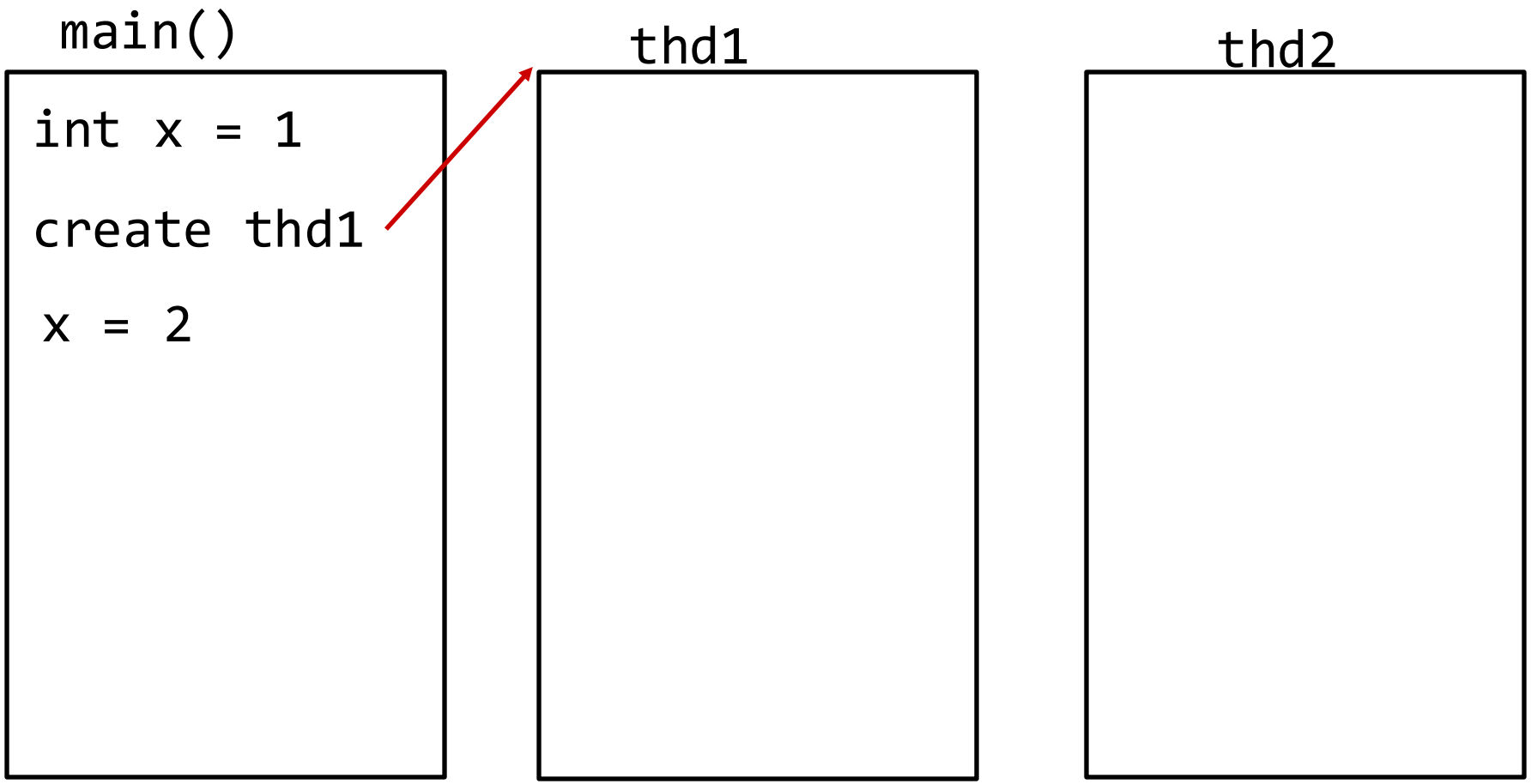
Visualization: Ordering

- Threads run concurrently; we can't be sure of the ordering of things across threads.



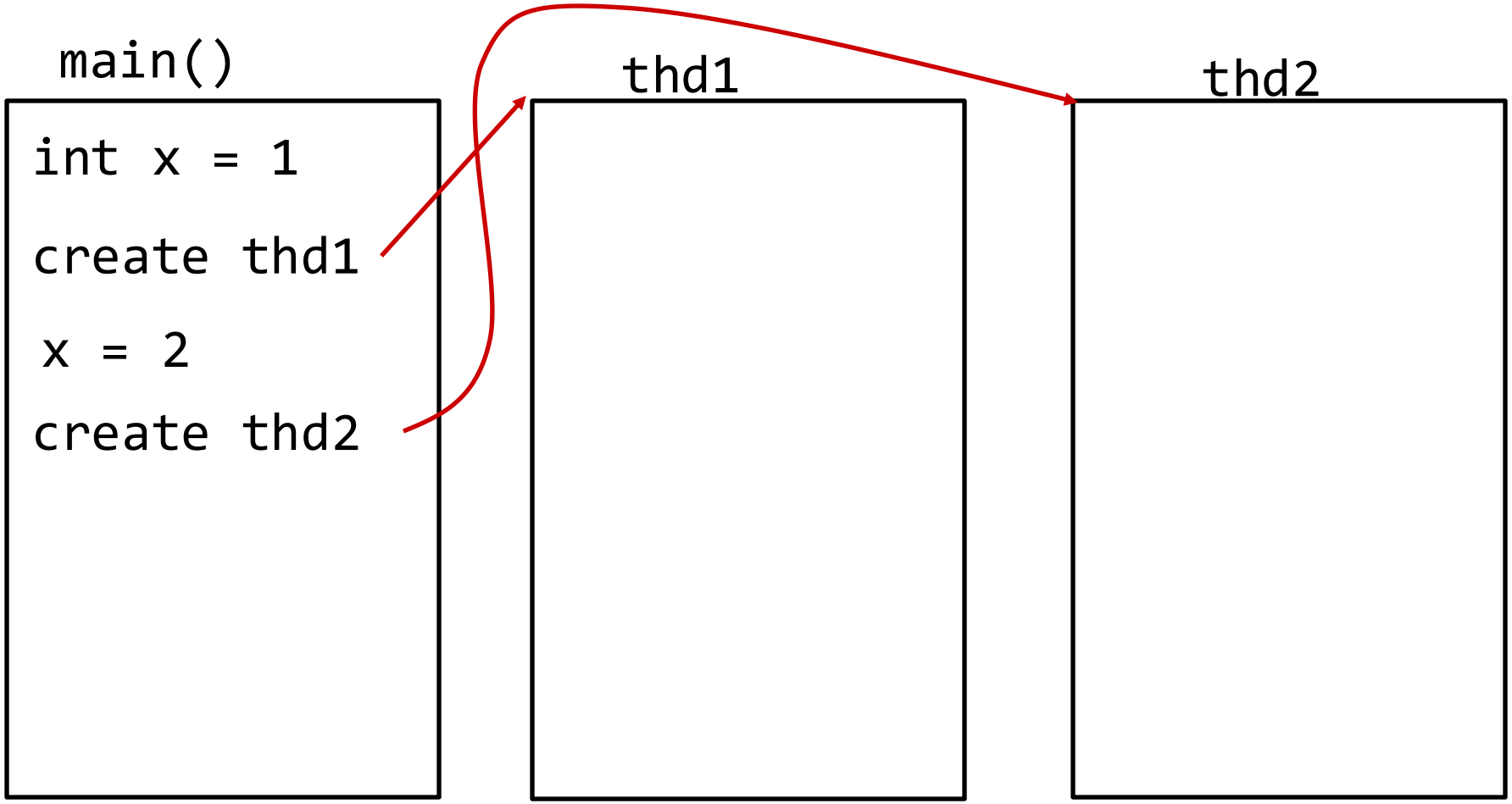
Visualization: Ordering

- Threads run concurrently; we can't be sure of the ordering of things across threads.



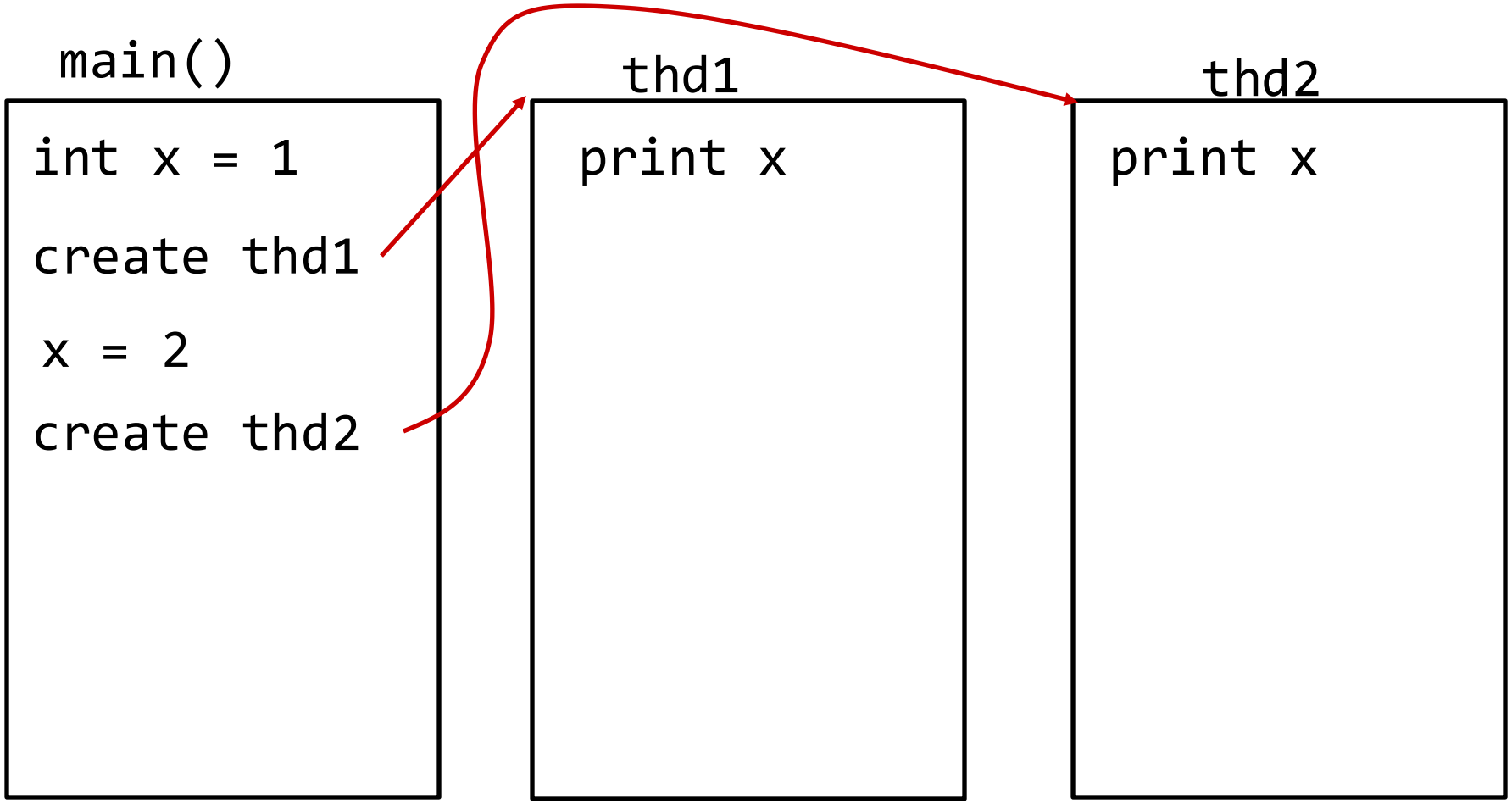
Visualization: Ordering

- Threads run concurrently; we can't be sure of the ordering of things across threads.



Visualization: Ordering

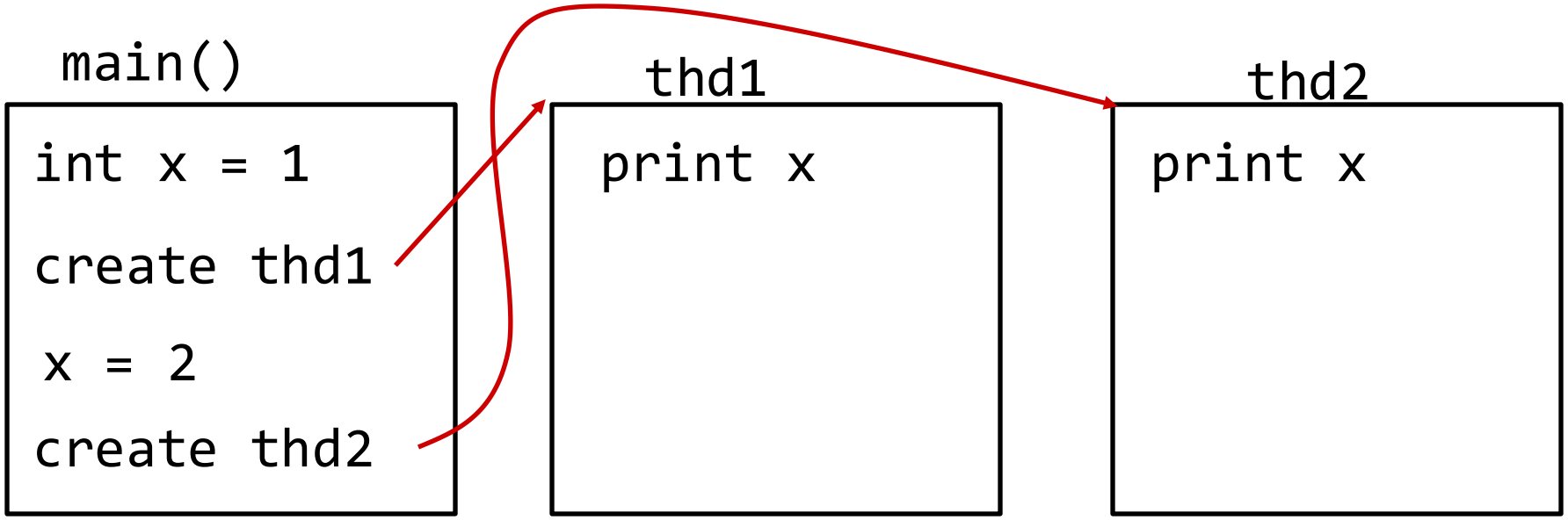
- Threads run concurrently; we can't be sure of the ordering of things across threads.



Visualization: Ordering

This is also why total.c malloc'd individual integers for each thread.
Though it could have also just made an array on the stack

- Threads run concurrently; we can't be sure of the ordering of things across threads.



We know that x is initialized to 1 before thd1 is created
We know that x is set to 2 and thd1 is created before thd2 is created

Anything else that we know? **No**. Beyond those statements, we do not know the ordering of main and the threads running.

Closer to the Truth: Assembly

```c
void* thrd_fn(void* arg) {
  int* ptr = (int*) arg;
  printf("%d\n", *ptr);
  return NULL;
}
```

```
thrd_fn:
    addi sp, sp, -16
    sw ra, 12(sp)
    lw a1, 0(a0)
    lui a0, %hi(.L.str)
    addi a0, a0, %lo(.L.str)
    call printf
    li a0, 0
    lw ra, 12(sp)
    addi sp, sp, 16
    ret


.L.str:
        .asciz "%d\n"
```

This is where we set up registers for the call to printf.

"Load a word from the address @ a0 into a1"

a0 contains the address in arg

a1 becomes the second arg to printf

Scenario 1:

main

```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

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

Exact syntax abbreviated

x

Address	Value
0x10000	1

thread 1

```
thrd_fn:
    addi sp, sp, -16
    sw ra, 12(sp)
    lw a1, 0(a0)
    lui a0, %hi(.L.str)
    addi a0, a0, %lo(.L.str)
    call printf
    li a0, 0
    lw ra, 12(sp)
    addi sp, sp, 16
    ret


.L.str:
        .asciz "%d\n"
```

Thread 1:
a1 = 1

Scenario 1:

main

```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

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

Exact syntax abbreviated

x

Address	Value
0x10000	2

thread 1
thread 2

```
thrd_fn:
    addi sp, sp, -16
    sw ra, 12(sp)
    lw a1, 0(a0)
    lui a0, %hi(.L.str)
    addi a0, a0, %lo(.L.str)
    call printf
    li a0, 0
    lw ra, 12(sp)
    addi sp, sp, 16
    ret


.L.str:
        .asciz "%d\n"
```

Thread 1:
a1 = 1

Thread 2:
a1 = 2

Now, it can go either way.

Either thread 2 can finish first or thread 1.

2 1 or 1 2

REMEMBER: EACH THREAD HAS ITS OWN REGISTERS! AHHH!

Scenario 2:

main

```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

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

Exact syntax abbreviated

x

Address	Value
0x10000	1

thread 1

thread 1 is pre-empted by main before load.

```
thrd_fn:
    addi sp, sp, -16
    sw ra, 12(sp)
    lw a1, 0(a0)
    lui a0, %hi(.L.str)
    addi a0, a0, %lo(.L.str)
    call printf
    li a0, 0
    lw ra, 12(sp)
    addi sp, sp, 16
    ret


.L.str:
        .asciz "%d\n"
```

Thread 1:
a1 = ?

Pre-empted: taken off a CPU to run another thread

Scenario 2:

```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

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

main

Exact syntax abbreviated

	Address	Value
x	0x10000	~~1~~ 2

thread 1

```
thrd_fn:
    addi sp, sp, -16
    sw ra, 12(sp)
    lw a1, 0(a0)
    lui a0, %hi(.L.str)
    addi a0, a0, %lo(.L.str)
    call printf
    li a0, 0
    lw ra, 12(sp)
    addi sp, sp, 16
    ret


.L.str:
        .asciz "%d\n"
```

Thread 1:
a1 = ?

Pre-empted: taken off a CPU to run another thread

Scenario 2:

```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

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

main

Exact syntax abbreviated

	Address	Value
x	0x10000	2

thread 1

thread 2

```
thrd_fn:
    addi sp, sp, -16
    sw ra, 12(sp)
    lw a1, 0(a0)
    lui a0, %hi(.L.str)
    addi a0, a0, %lo(.L.str)
    call printf
    li a0, 0
    lw ra, 12(sp)
    addi sp, sp, 16
    ret


.L.str:
        .asciz "%d\n"
```

Thread 1:
a1 = 2

Now when thread 1 continues it will load in the value 2.

Thread 2:
a1 = 2

Now, it can go either way.

Either thread 2 can finish first or thread 1.

2 2 or 2 2

But they're identical.

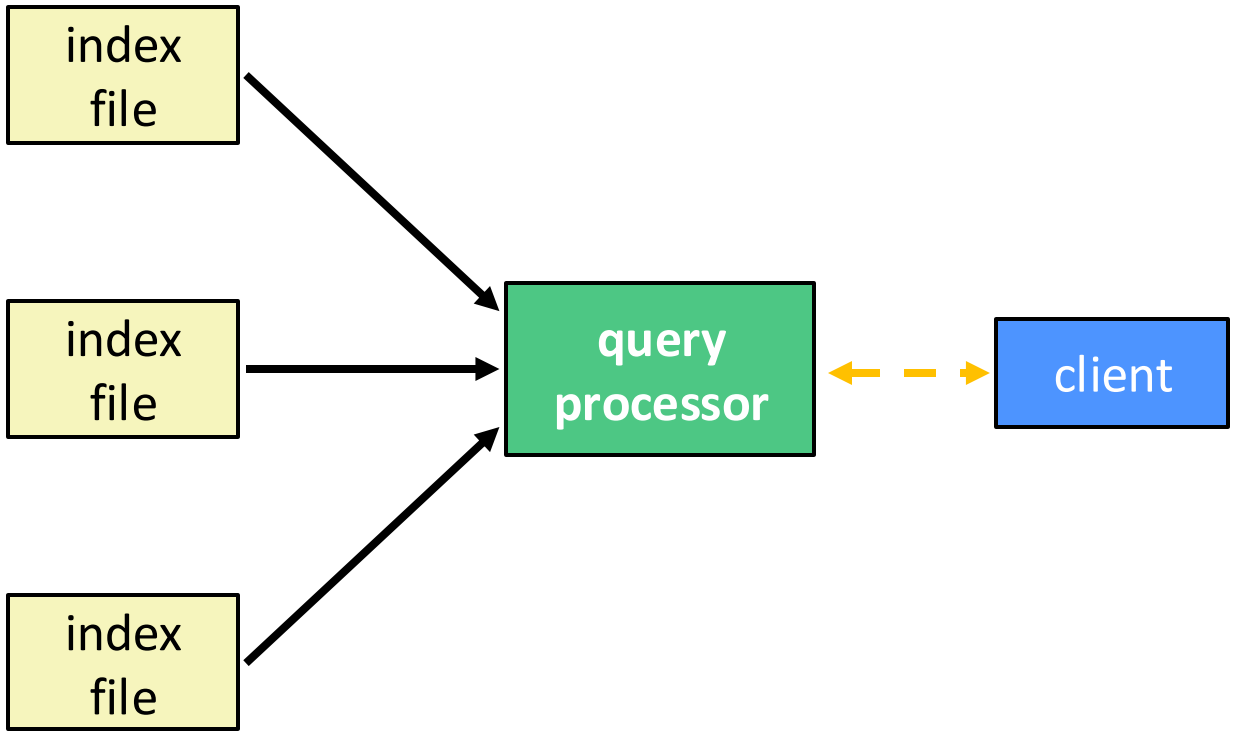
Lecture Outline

- Threads High Level
- pthreads
- Processes vs threads
- Thread Interleaving & Sequential Consistency
- **Benefits of Concurrency**
- Processes vs Threads

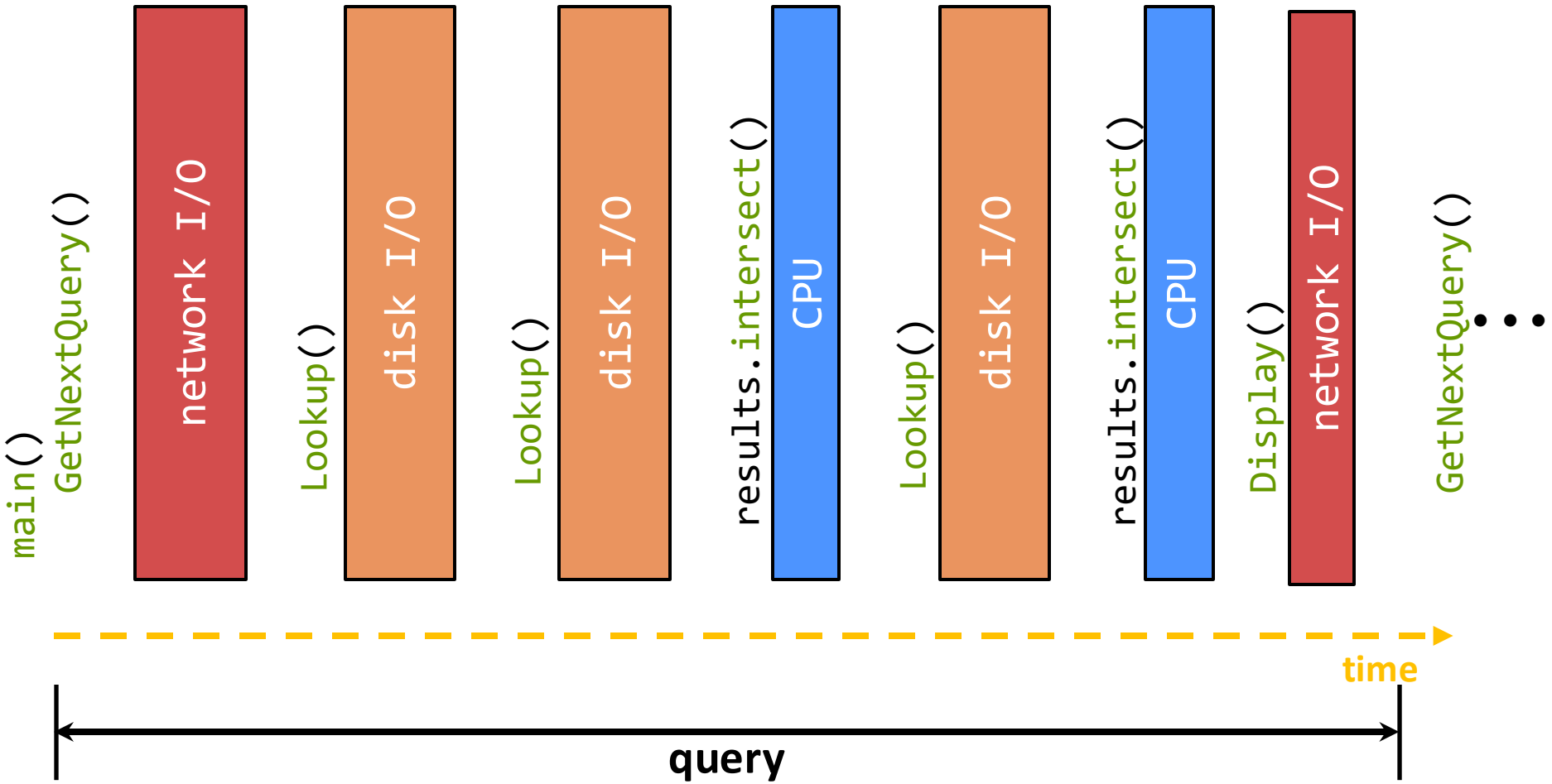
Building a Web Search Engine

- We have:
  - A web index
    - A map from <*word*> to <*list of documents containing the word*>
    - This is probably *shared* over multiple files
  - A query processor
    - Accepts a query composed of multiple words
    - Looks up each word in the index
    - Merges the result from each word into an overall result set

Search Engine Architecture

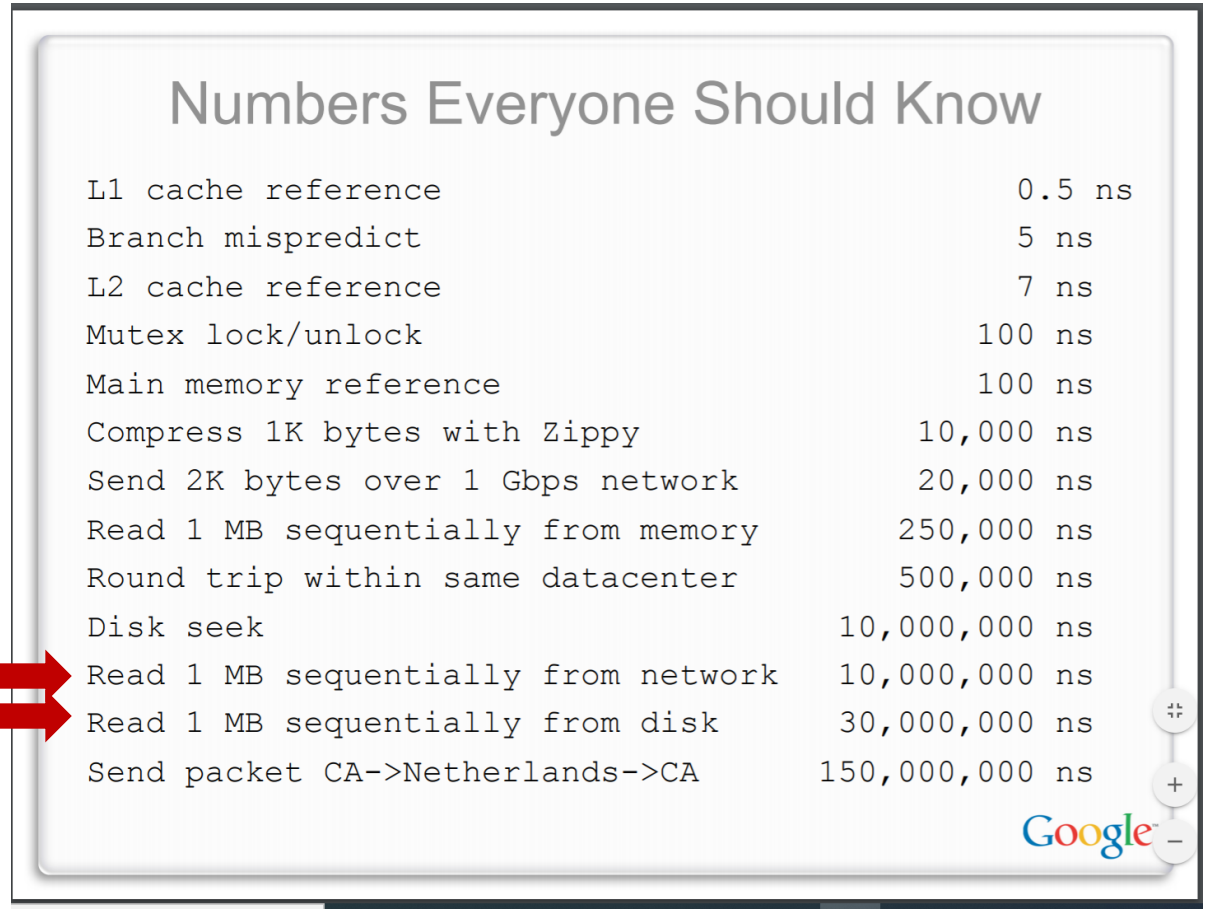


Execution Timeline: a Multi-Word Query



What About I/O-caused Latency?

- Jeff Dean's "Numbers Everyone Should Know" (LADIS '09)

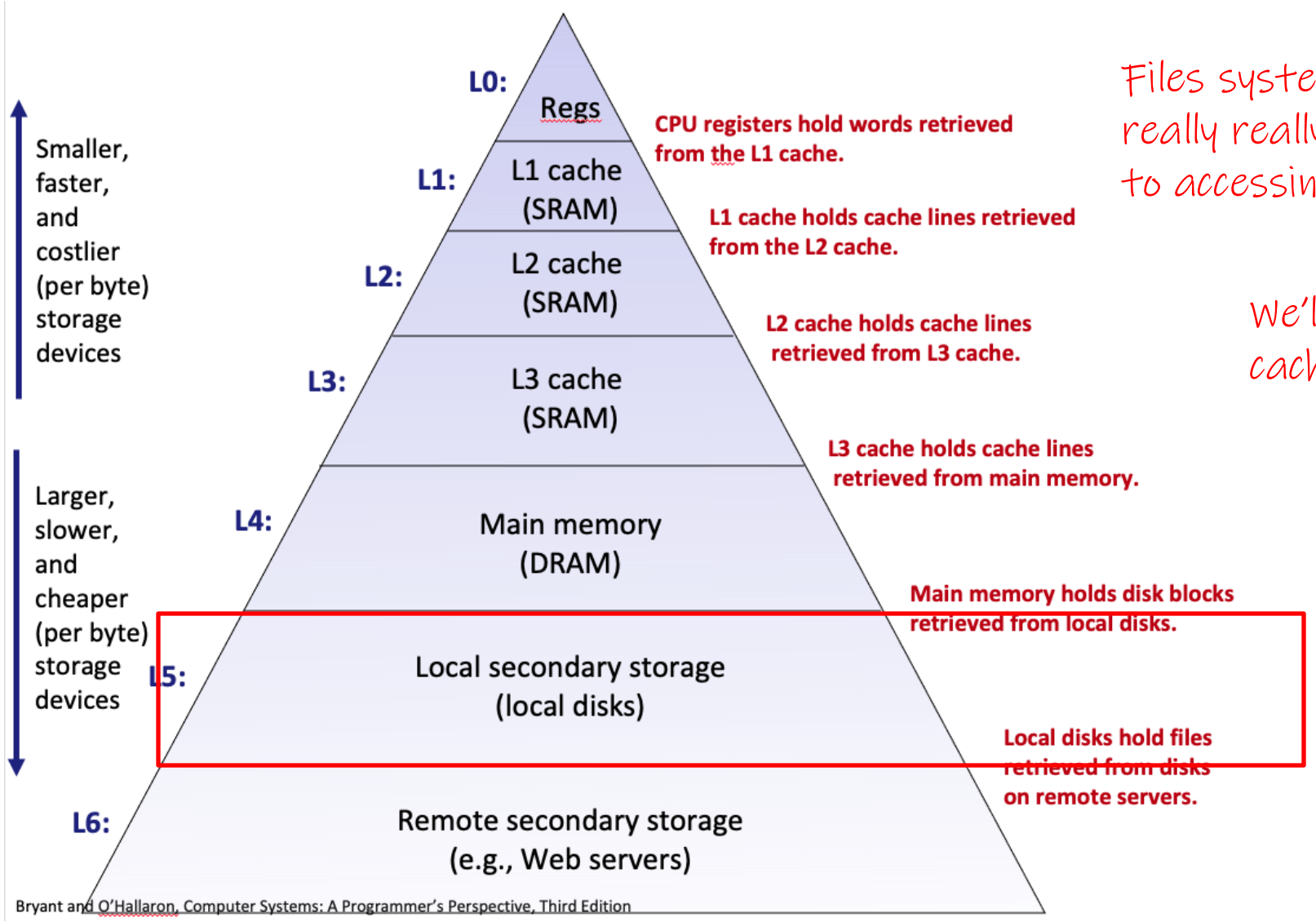


Numbers Everyone Should Know

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

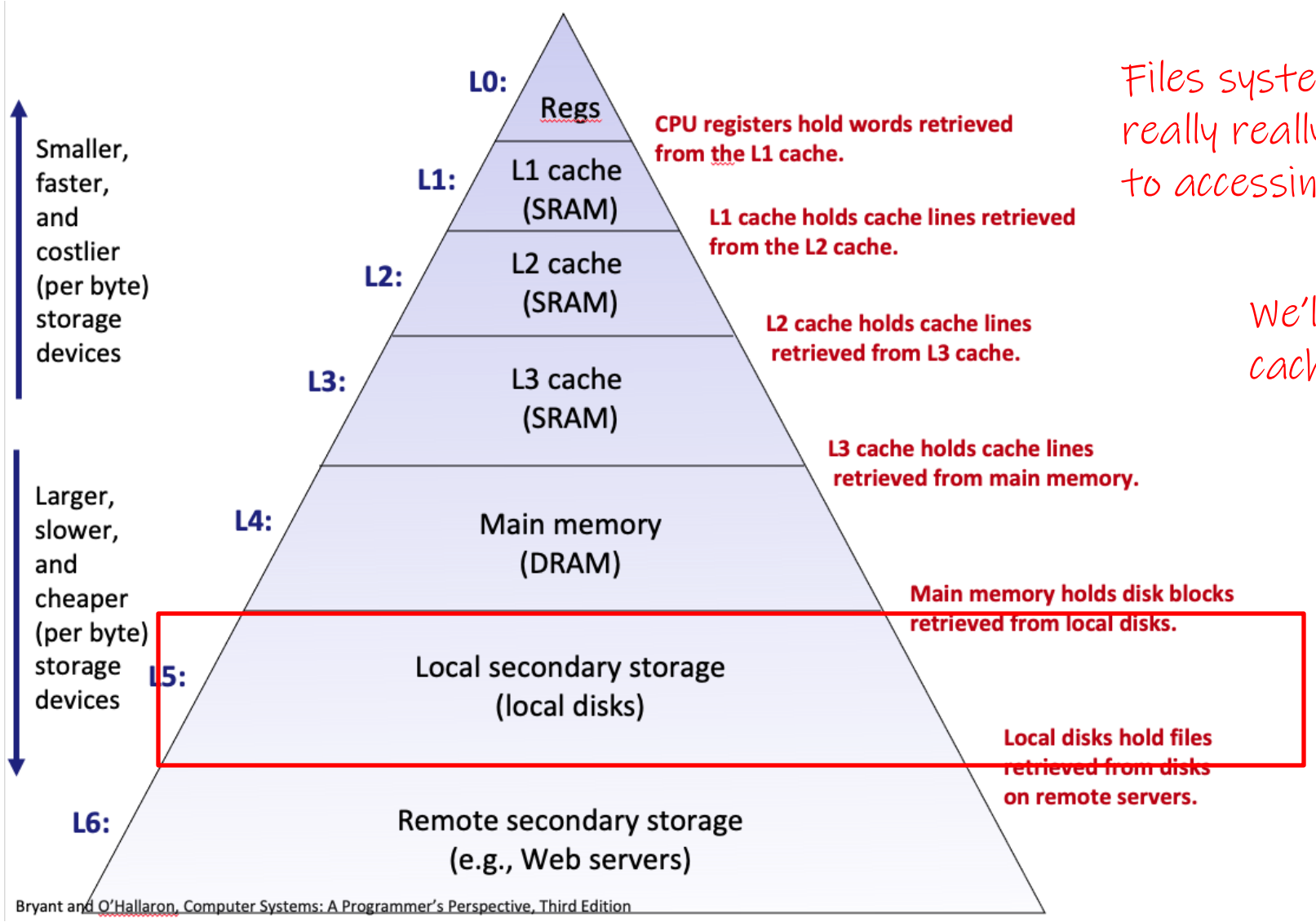
Memory Hierarchy



Files systems are really really really slow compared to accessing memory

We'll talk about caches later

THIS IS REALLY IMPORTANT (discussed later in the sem)

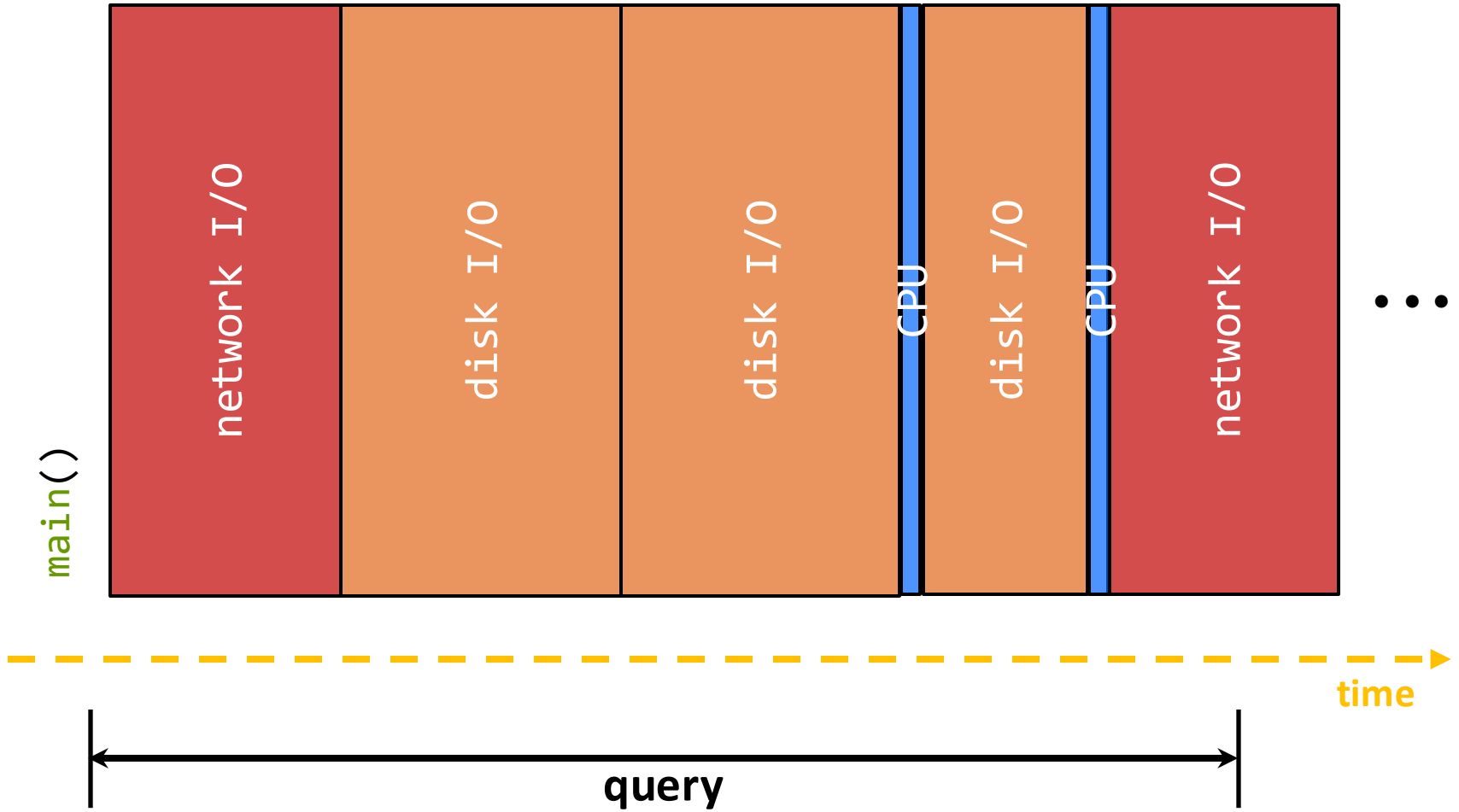


Files systems are really really really slow compared to accessing memory

We'll talk about caches later

Execution Timeline: To Scale

Model isn't perfect:
Technically also some cpu usage to setup I/O.
Network output also (probably) won't block program .....



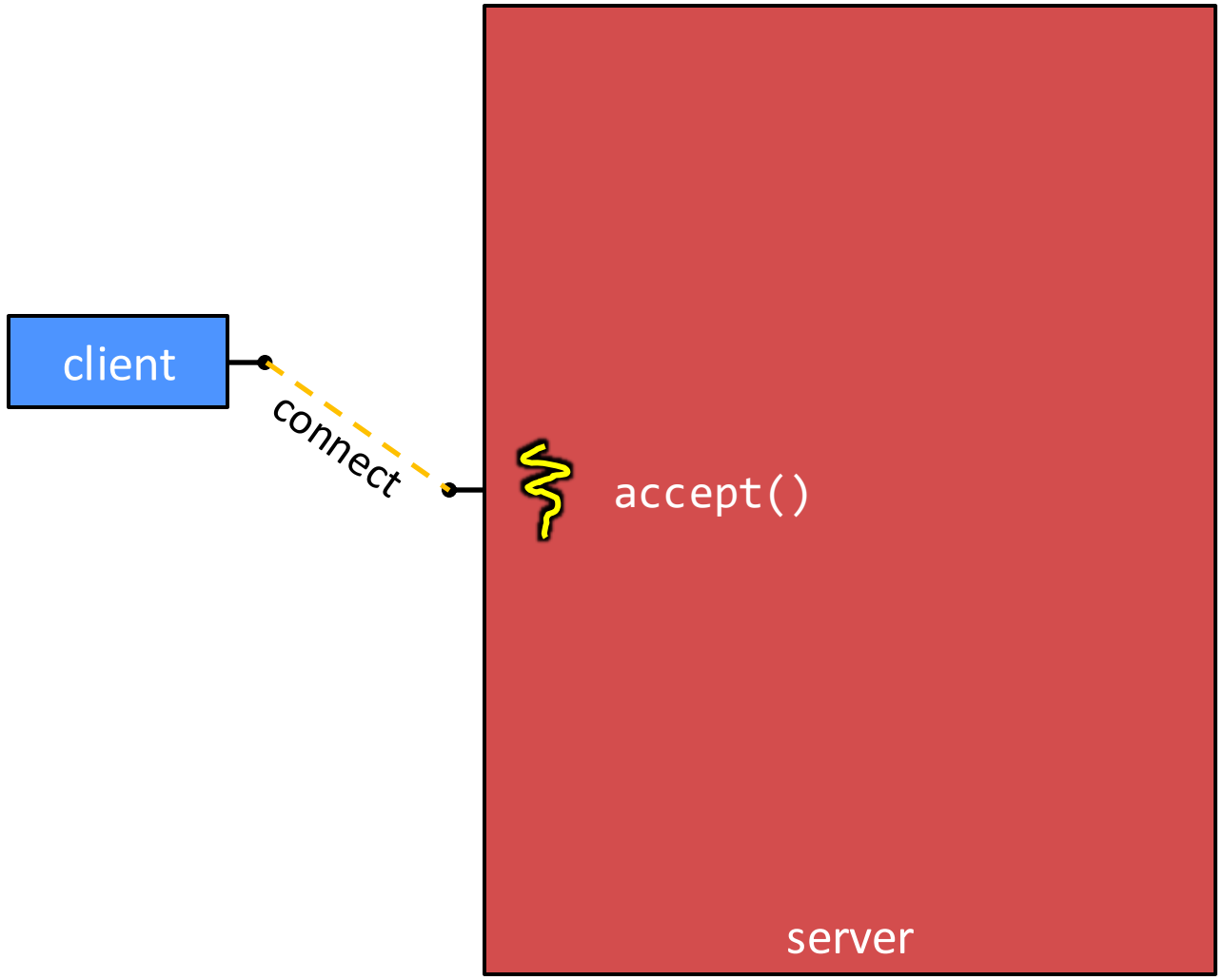
Sequential Can Be Inefficient

- Only one query is being processed at a time
  - All other queries queue up behind the first one
  - And clients queue up behind the queries …
- Even while processing one query, the CPU is idle the vast majority of the time
  - It is *blocked* waiting for I/O to complete
    - Disk I/O can be very, very slow (10 million times slower …)
- At most one I/O operation is in flight at a time
  - Missed opportunities to speed I/O up
    - Separate devices in parallel, better scheduling of a single device, etc.

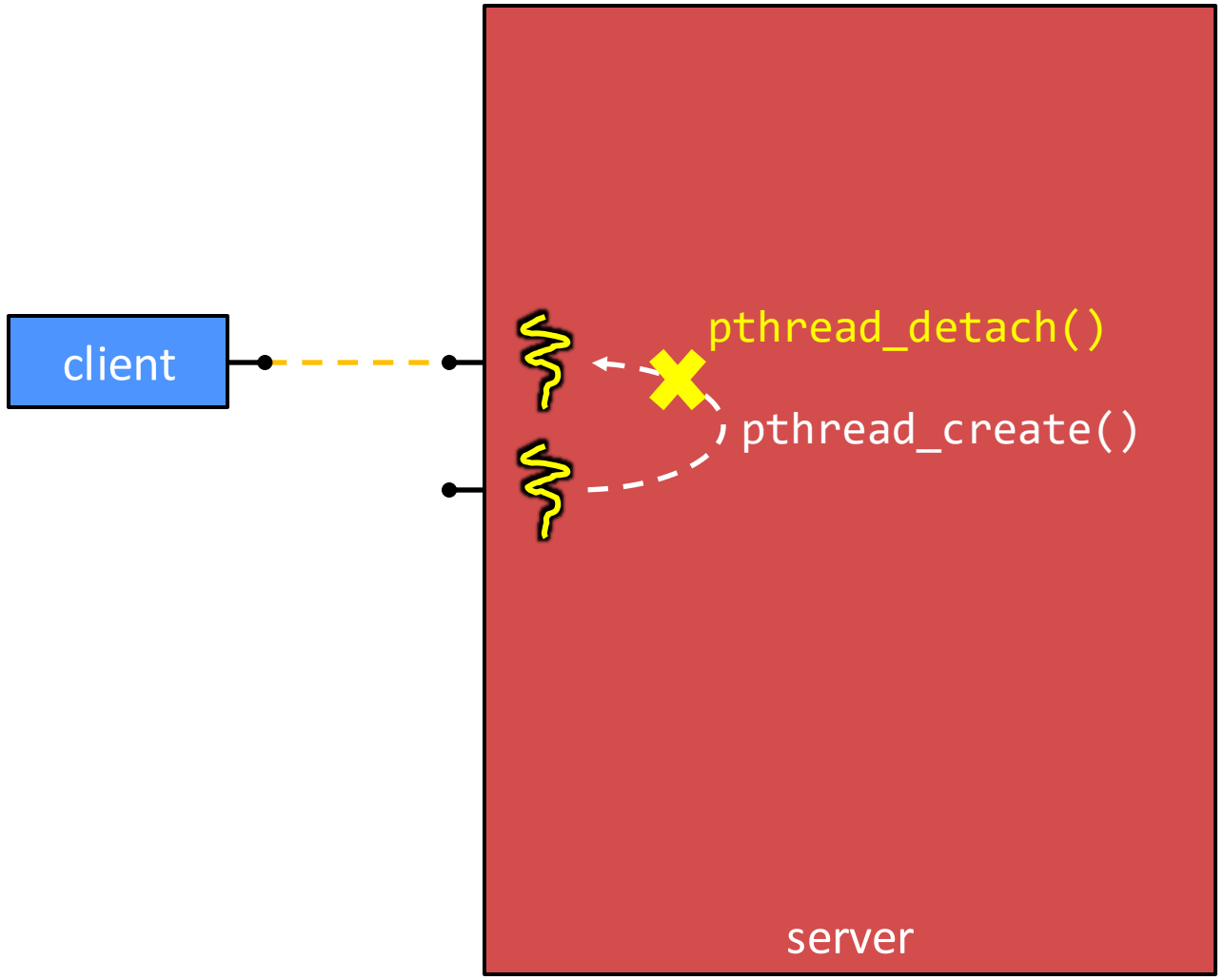
A Concurrent Implementation

- Use multiple “workers” (Threads or processes, but usually threads)
  - As a query arrives, create a new “worker” to handle it
    - The “worker” reads the query from the network, issues read requests against files, assembles results and writes to the network
    - The “worker” uses blocking I/O; the “worker” alternates between consuming CPU cycles and blocking on I/O
  - The OS context switches between “workers”
    - While one is blocked on I/O, another can use the CPU
    - Multiple “workers’” I/O requests can be issued at once

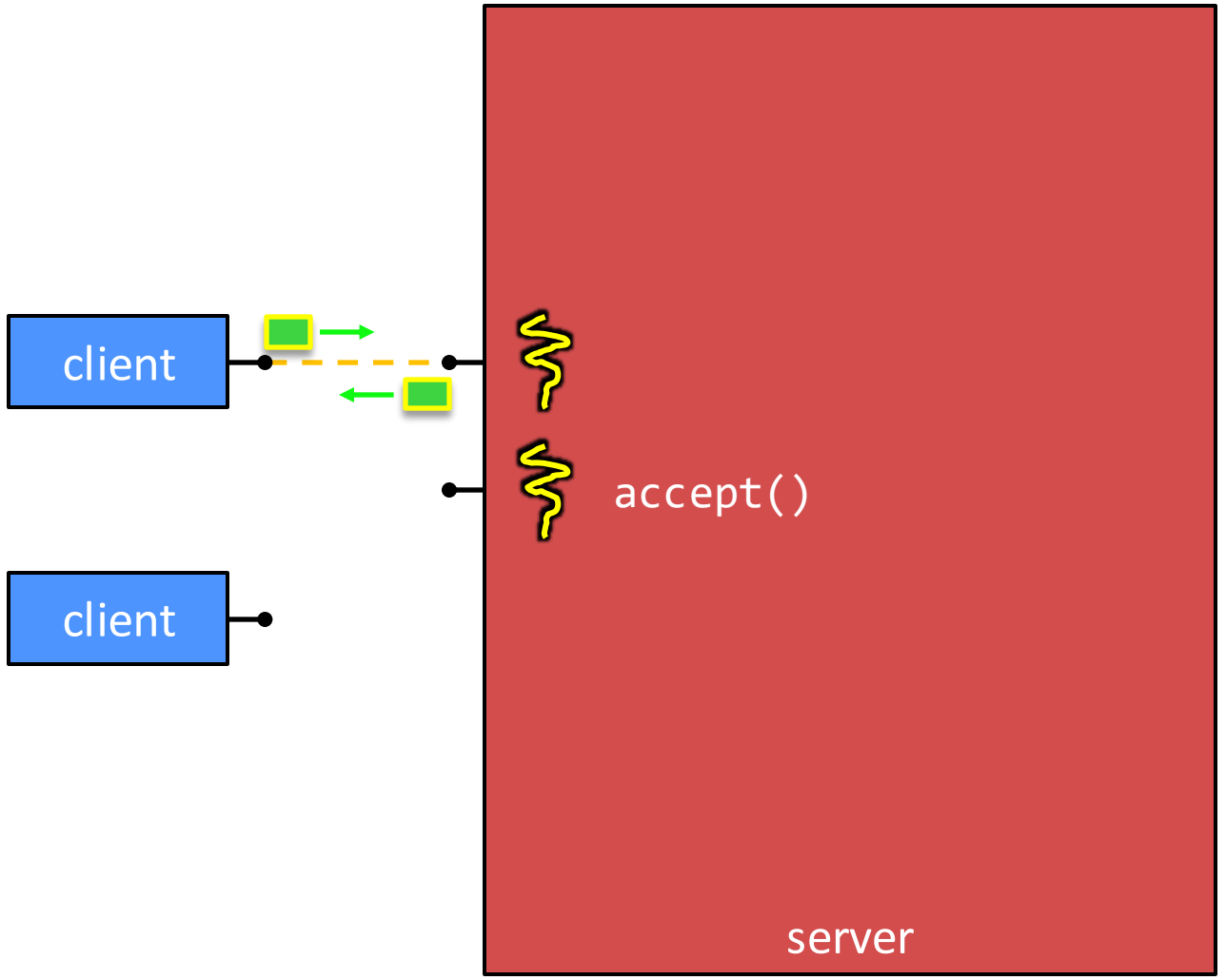
Multithreaded Server



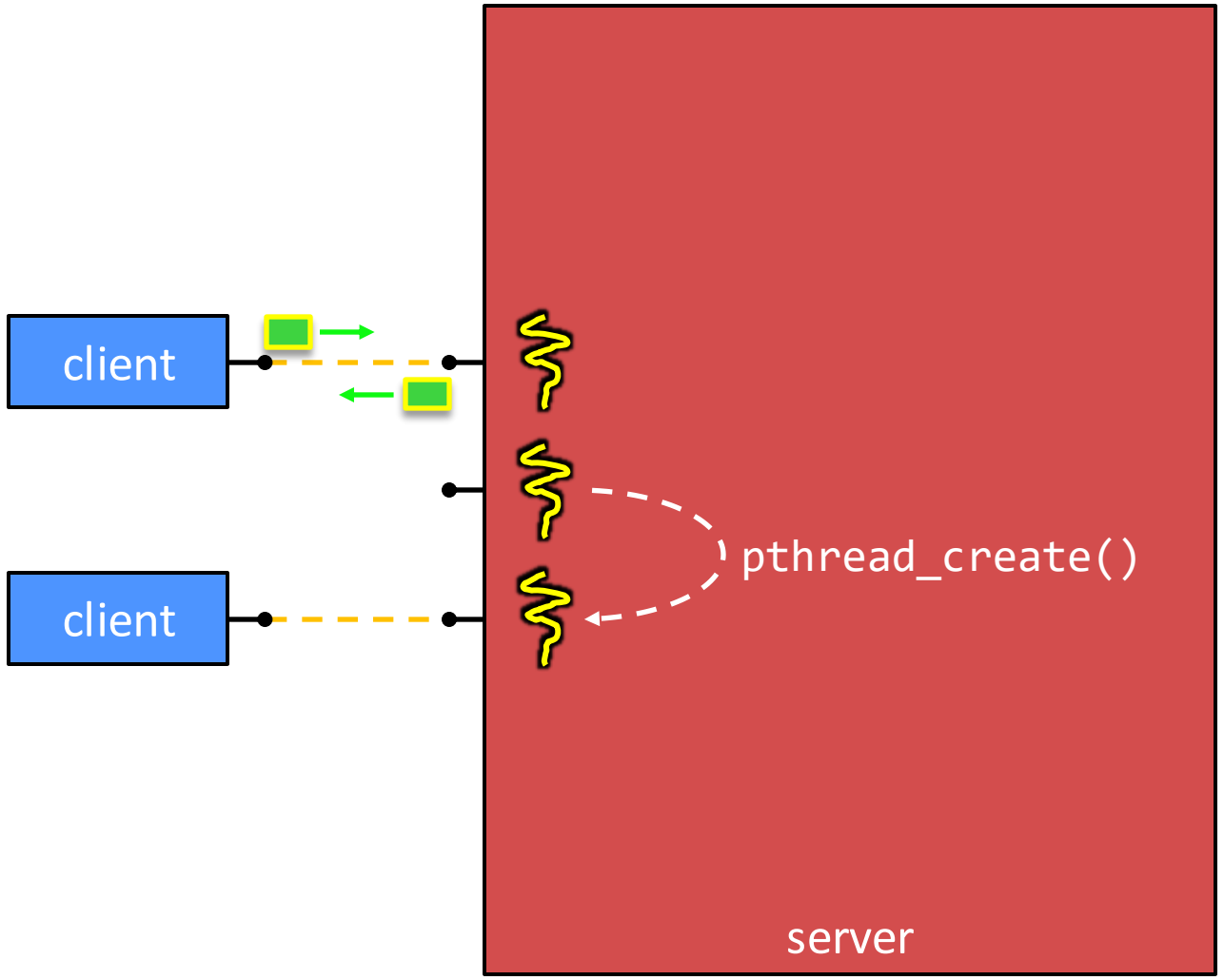
Multithreaded Server



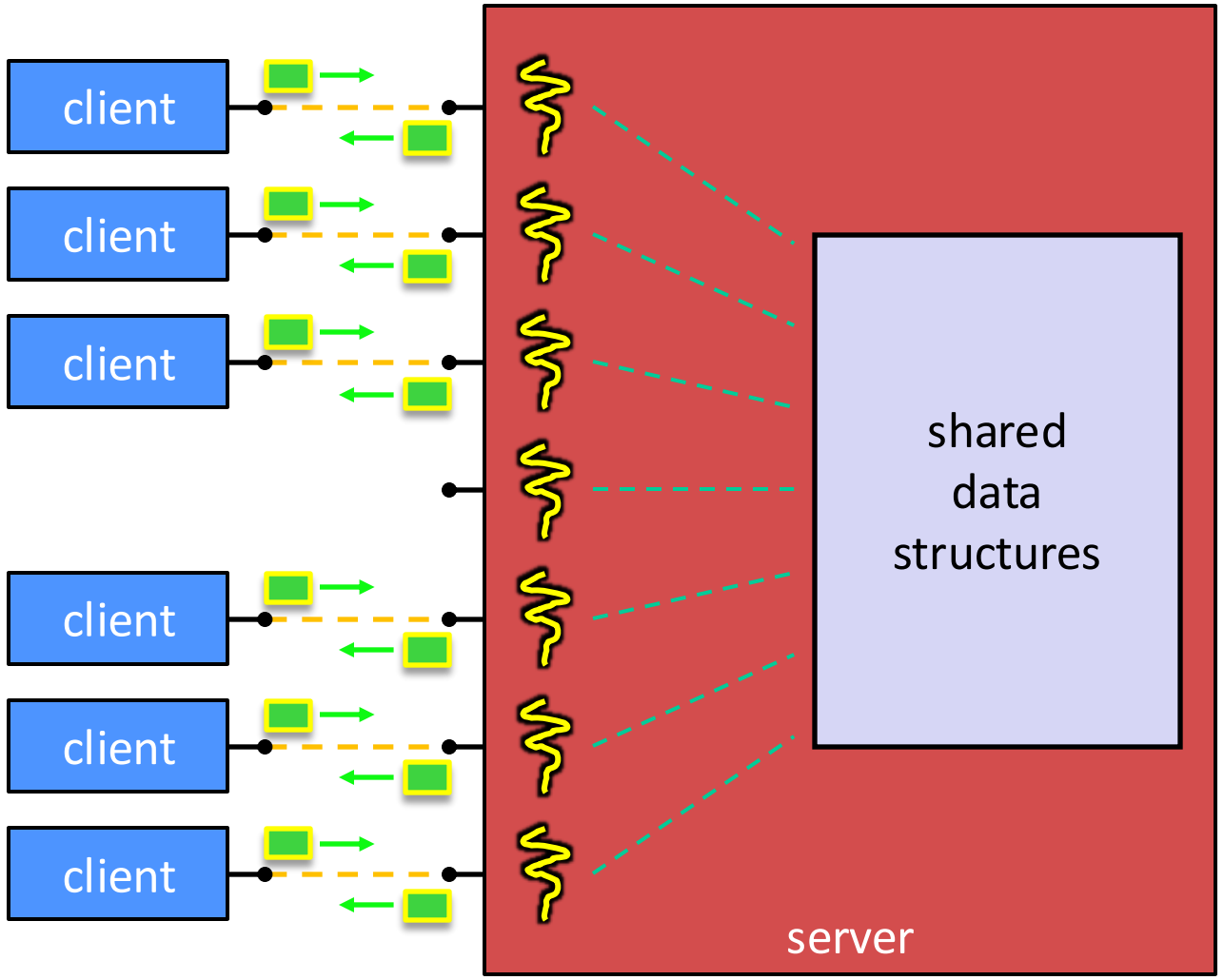
Multithreaded Server



Multithreaded Server

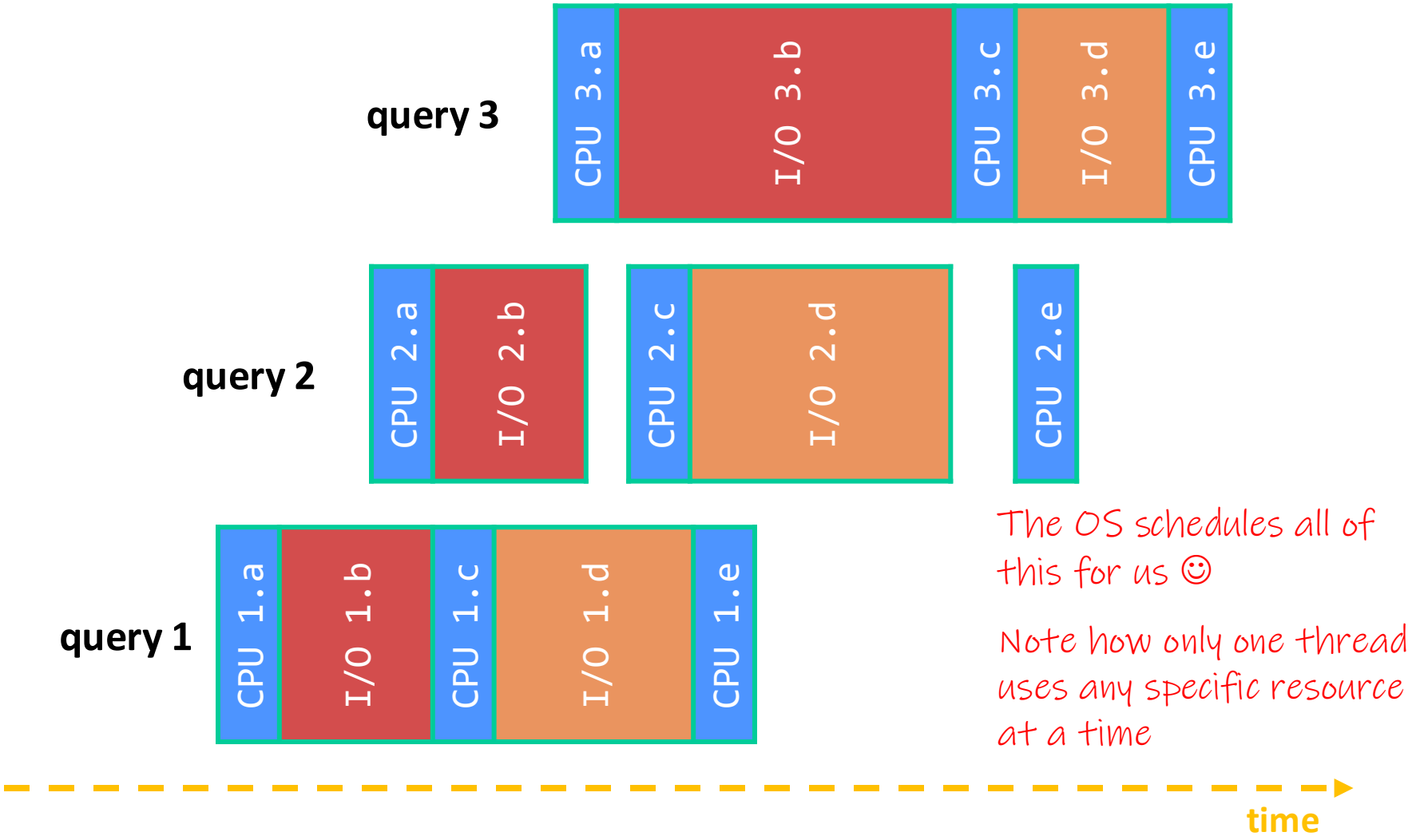


Multithreaded Server



Multi-threaded Search Engine (Execution)

*Running with 1 CPU



The OS schedules all of this for us ☺

Note how only one thread uses any specific resource at a time

Lecture Outline

- Threads High Level
- pthreads
- Processes vs threads
- Thread Interleaving & Sequential Consistency
- **Benefits of Concurrency**
- Processes vs Threads

Why Threads?

- Advantages:
  - You it looks more like sequential code than alternatives (processes, async, etc.)
  - Threads can run in parallel if you have multiple CPUs/cores

- Disadvantages:
  - If threads share data, you need locks or other synchronization
    - Very bug-prone and difficult to debug
  - Threads can introduce overhead
    - Lock contention, context switch overhead, and other issues
  - Need language support for threads

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

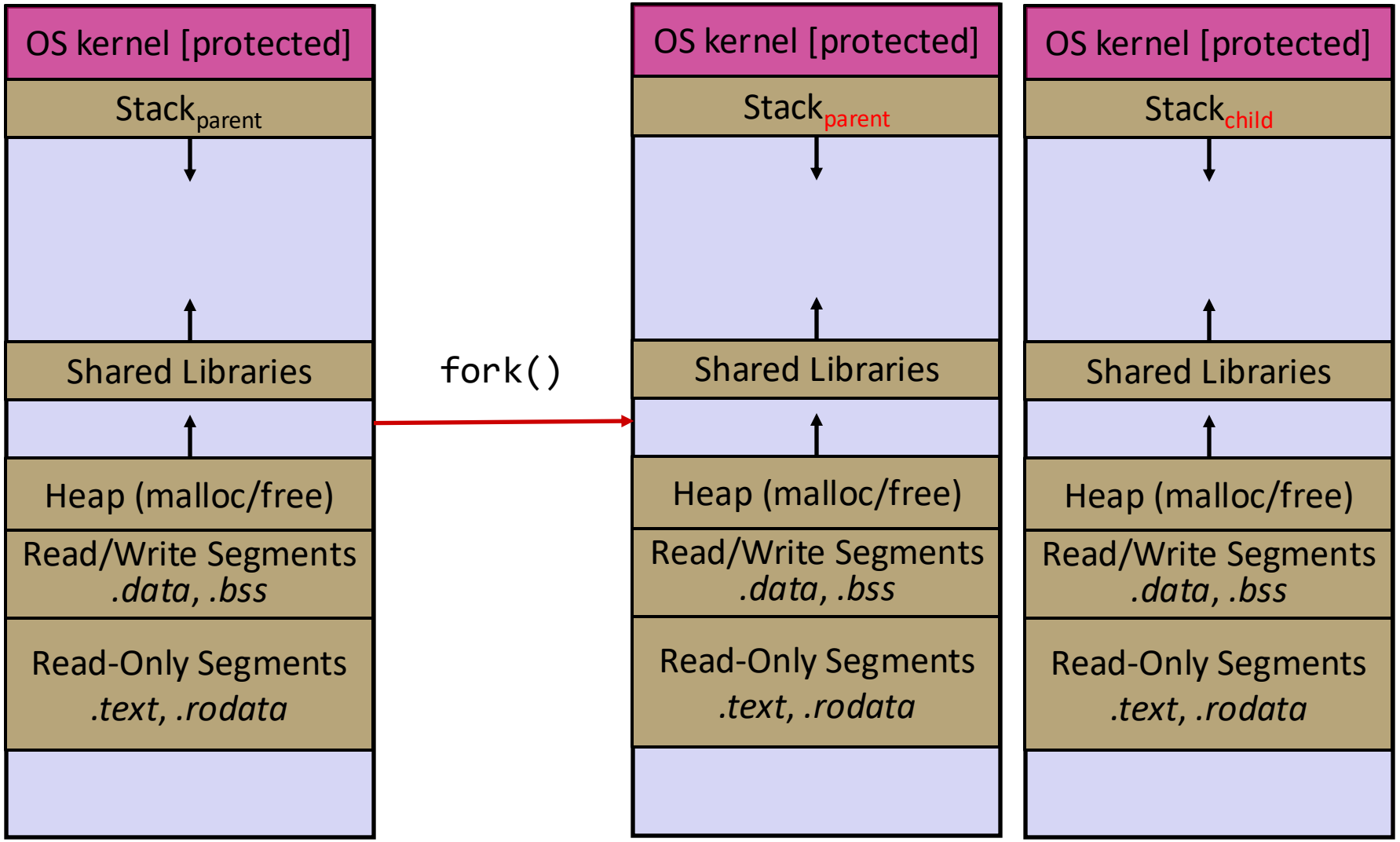
Threads vs. Processes

- Signals are a process level communication mechanism – who gets the signal?
  - Could go to any thread running in the process!
  - `pthread_kill(pthread_t thread, int sig)` can be used to target a specific thread

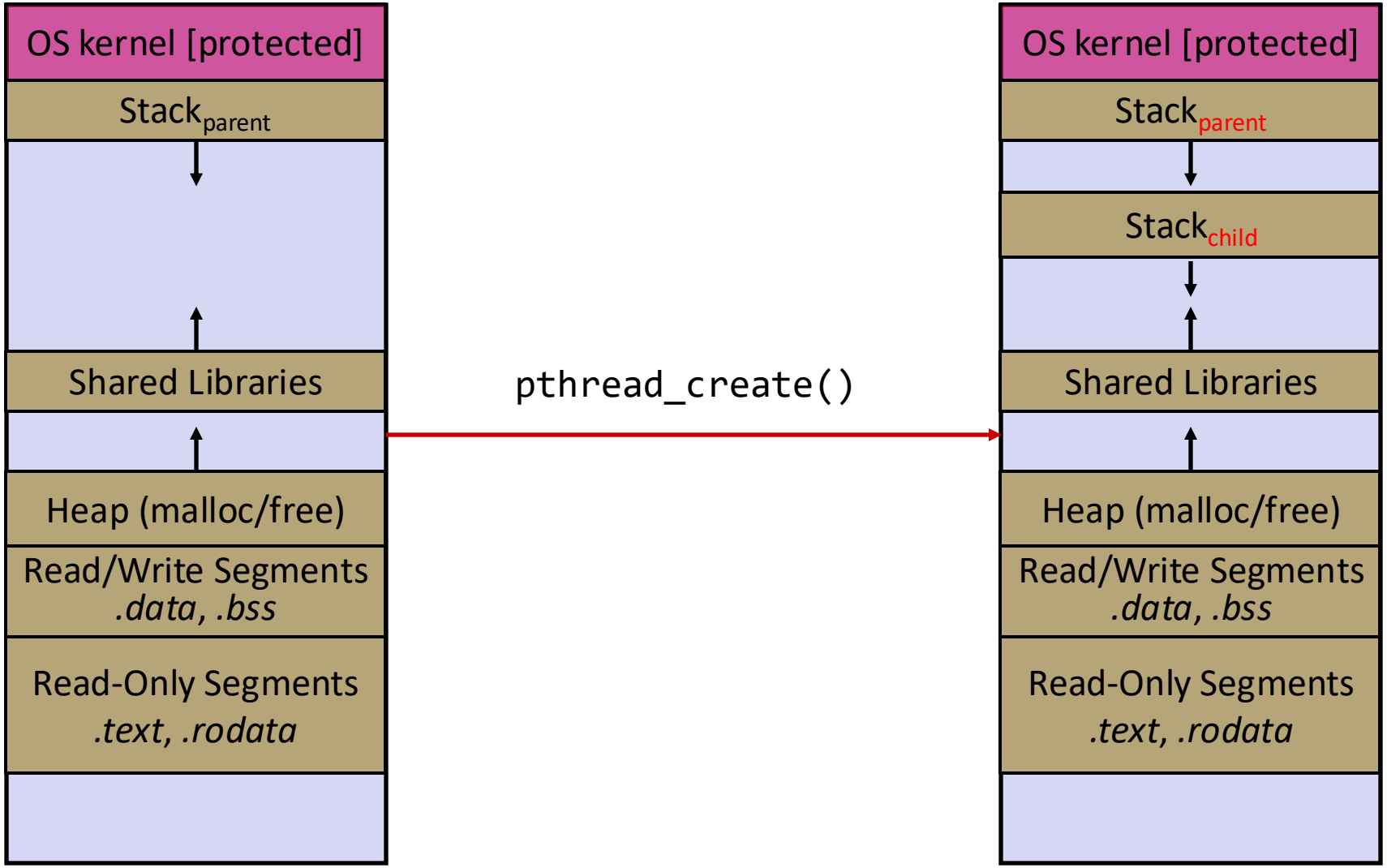
- `fork()` is still a valid system call any thread can use – what happens?
  - New process does **not** copy all threads – only the thread which calls fork!
  - What happens to the program state of the non-copied threads in the child process?

- A process is the **program environment**, and threads are a **specific execution context** in that environment – what does the OS track to maintain these?
  - In Linux, idea of a "task"

Threads vs. Processes



Threads vs. Processes



Alternative: Processes

- What if we forked processes instead of threads?

- Advantages:
  - No shared memory between processes
  - No need for language support; OS provides “fork”
  - Processes are isolated. If one crashes, other processes keep going

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

That's all!

- See you next time!