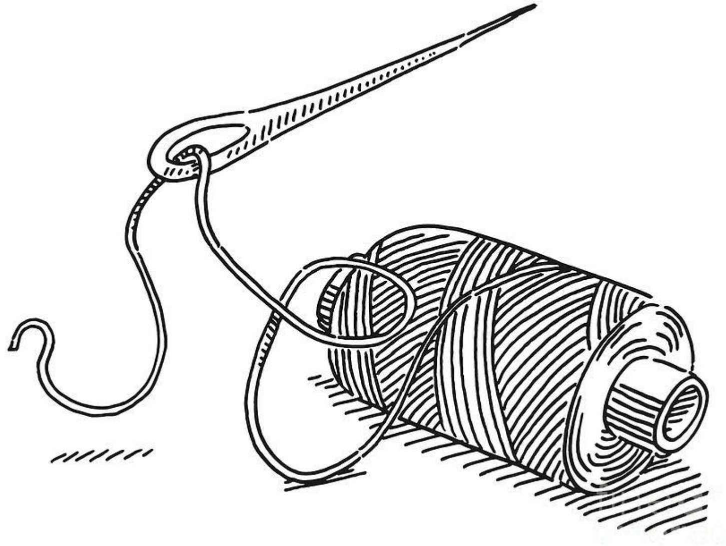
# Intro to Threads

## Computer Operating Systems, Fall 2025




Instructors: Joel Ramirez

Head TAs: Maya Huizar, Akash Kaukuntla, Vedansh Goenka, Joy Liu

TAs:

| | | | |
|---|---|---|---|
| Eric Zou | Joseph Dattilo | Aniket Ghorpade | Shriya Sane |
| Zihao Zhou | Eric Lee | Shruti Agarwal | Yemisi Jones |
| Connor Cummings | Shreya Mukunthan | Alexander Mehta | Raymond Feng |
| Bo Sun | Steven Chang | Rania Souissi | Rashi Agrawal |
| Sana Manesh | | | |

# Poll Everywhere

pollev.com/cis5480

- How are you doing?

# Administrivia

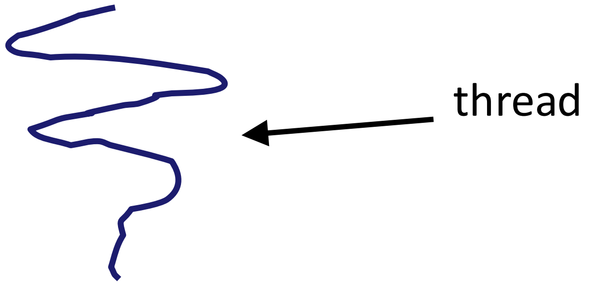
- Penn-shell is due this Friday, October 3rd
  - If you have any questions, please stop by office hours! We are here to help.
- Midterm will be on Thursday, October 16 from 5:15 – 6:45
  - Locations: Towne 100 & Wu and Chen in Levine
  - Assigned locations are TBD; prolly based on last name.
  - Practice exams and all that out this weekend
- Penn OS will come out Friday, October 17th after the midterm!
  - THINK ABOUT WHO YOUR GROUP WILL BE!!!!

# Lecture Outline

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

# 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

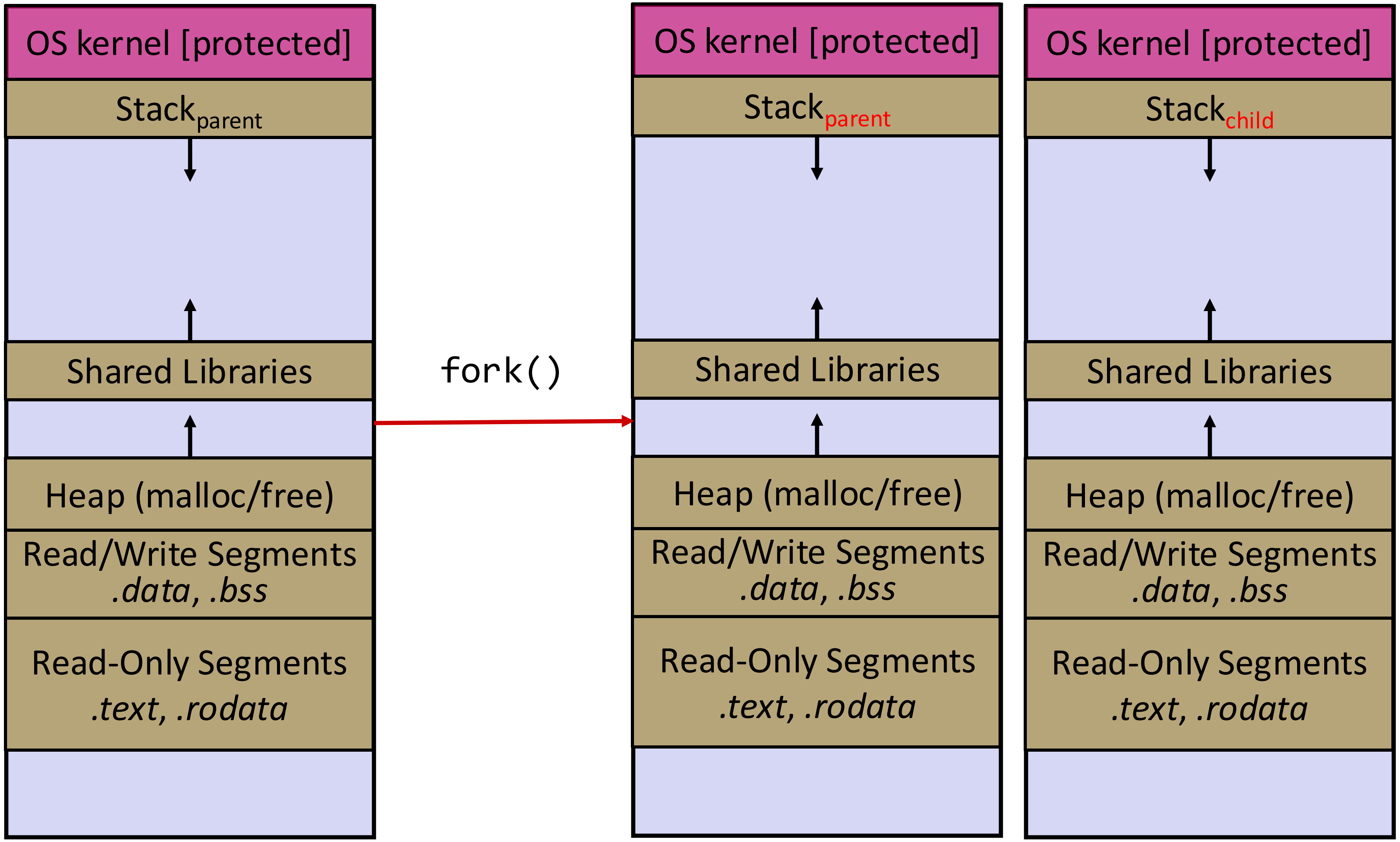



- In most modern OS's:
  - Threads are the *unit of scheduling.*

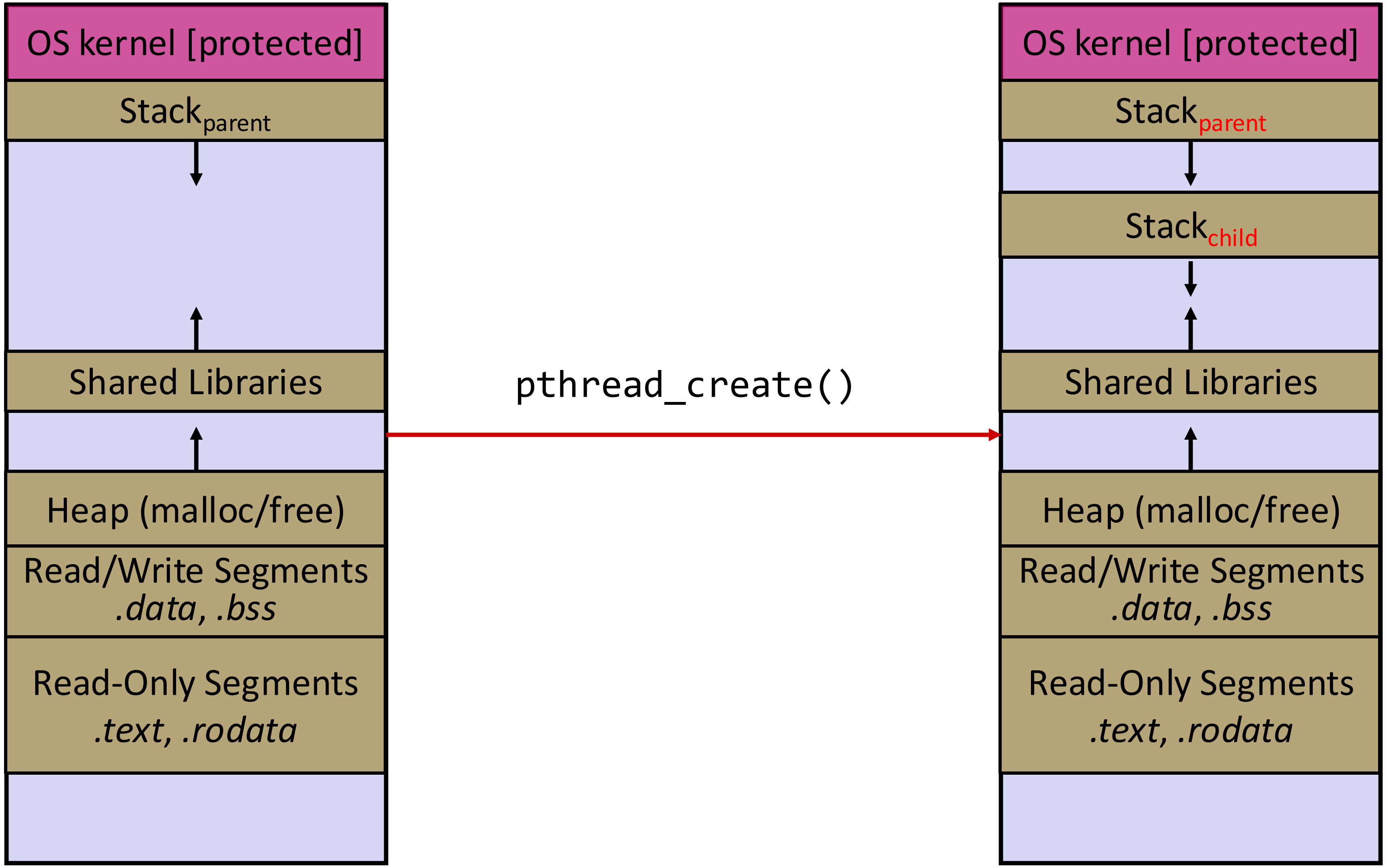
# Threads vs. Processes

- 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

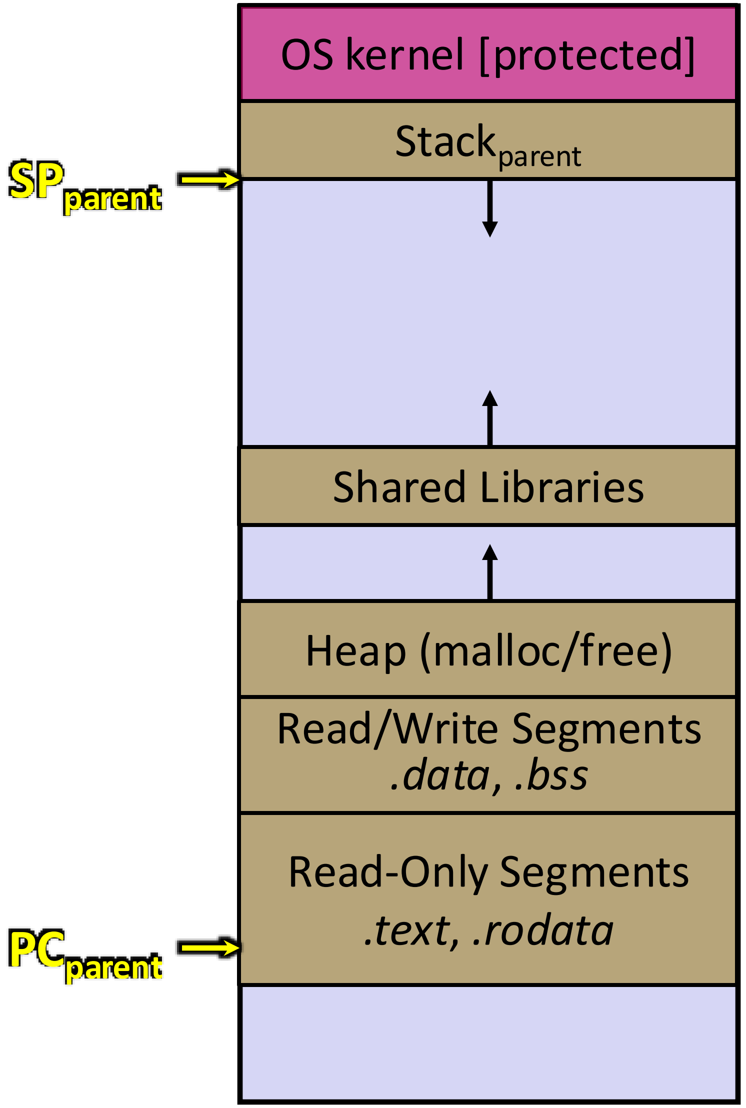


# Threads

- 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

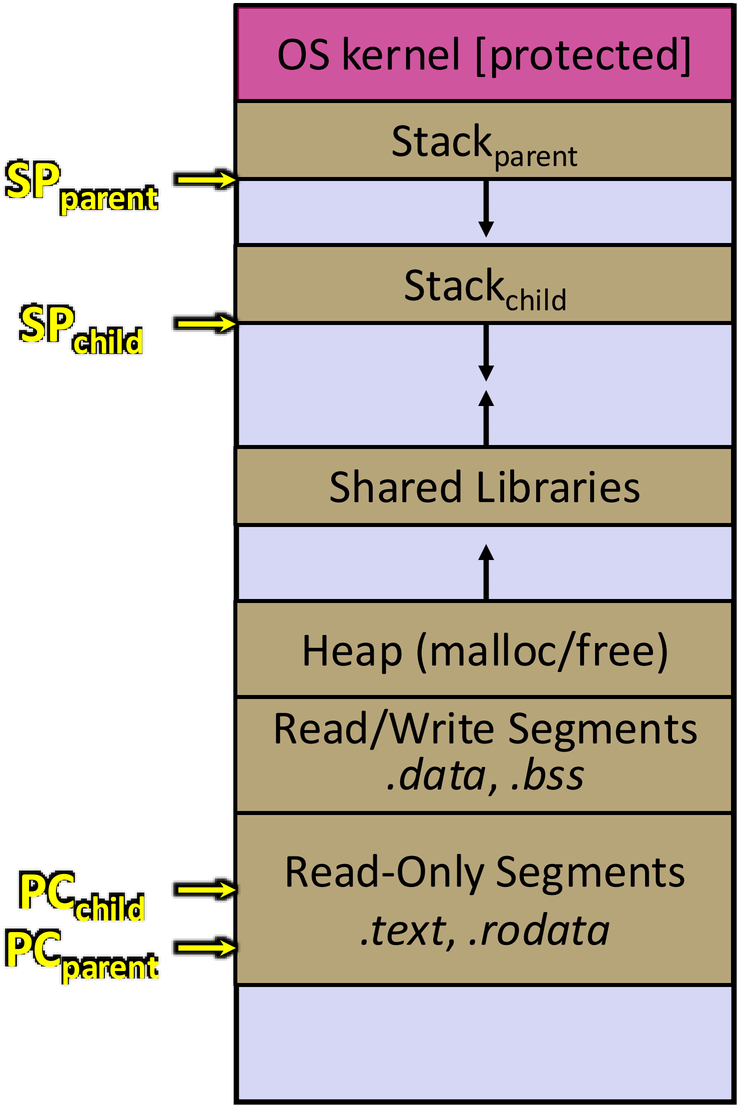
- Analogy: restaurant kitchen
  - Kitchen is process
  - Chefs are threads

# 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

# Lecture Outline

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

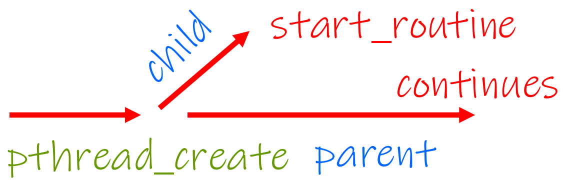
# POSIX Threads (pthreads)

- 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 and Terminating Threads

```c
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.
    - (i.e. `void *func(void *arg)`)
- `void*  arg`: the argument that will be passed into start (you can do a lot with a ptr)

# pthread_create

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

- `void*` `arg`: the argument that will be passed into start (you can do a lot with a ptr)

If you need a function that takes in various arguments, make arg a pointer to a struct that contains all of them.

There are no variable length arguments here. (C++ has them tho, 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 life time of new_thread is contained to this function (and the functions it calls).***

# pthread_create is *not* fork()

```
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 life time of new_thread is contained to this function***

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

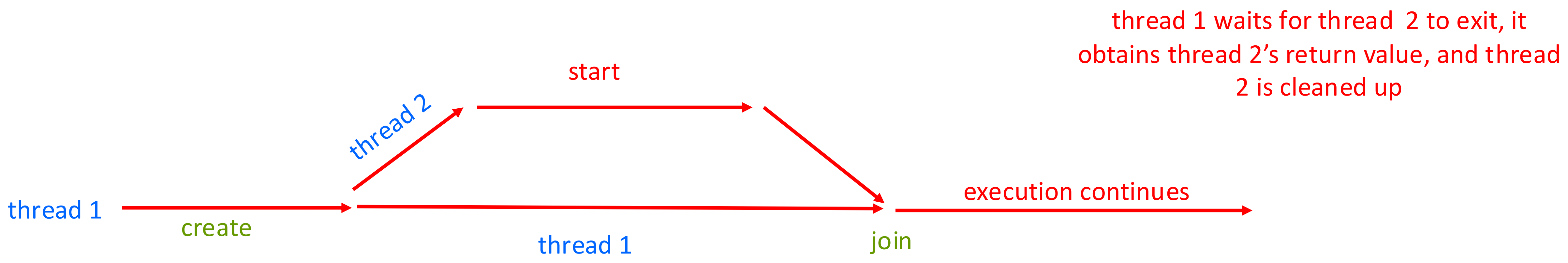
***THEY ARE NOT COPIES!***

***All threads share the same address space.***

# We Created a Thread, Now What?

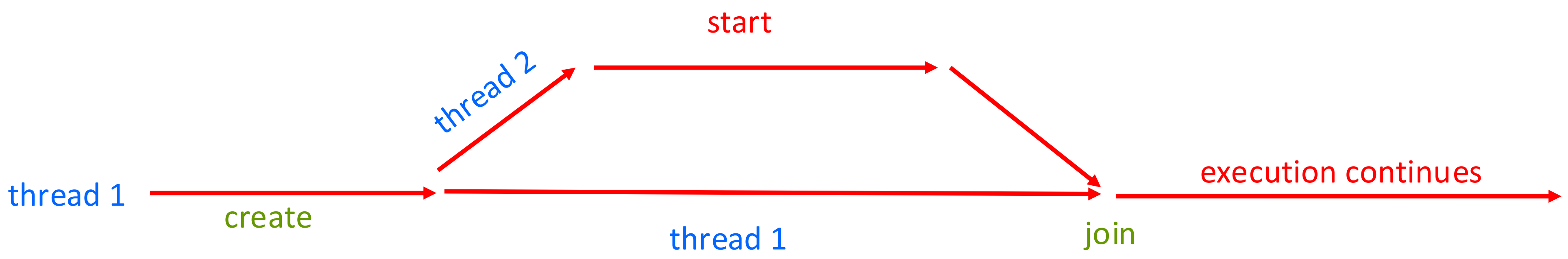
```
int pthread_join(pthread_t thread, void** retval);
```

- 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 schedular.**
**We can not assume when it will run.**

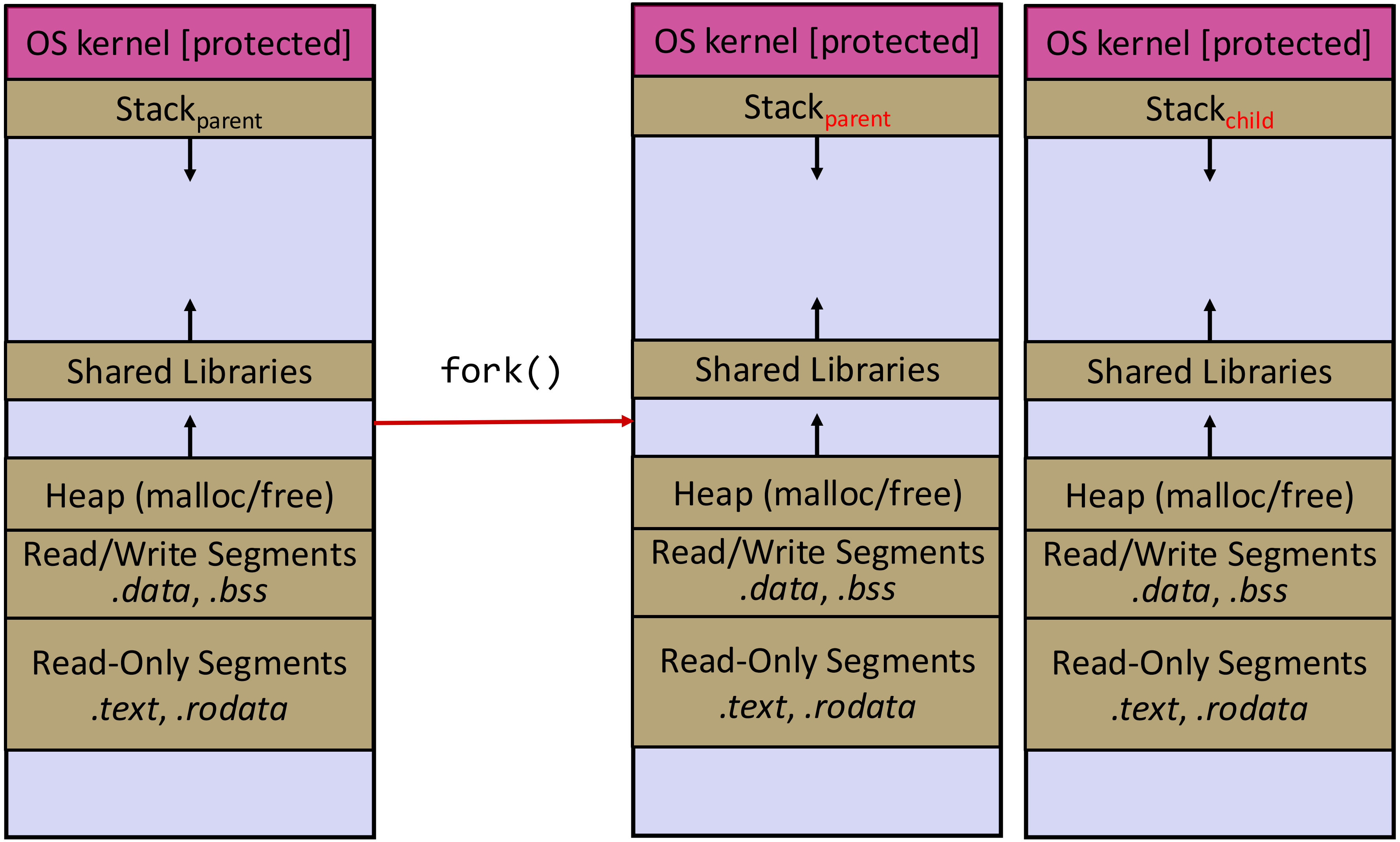
# Thread Example

- Let's take a look at `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

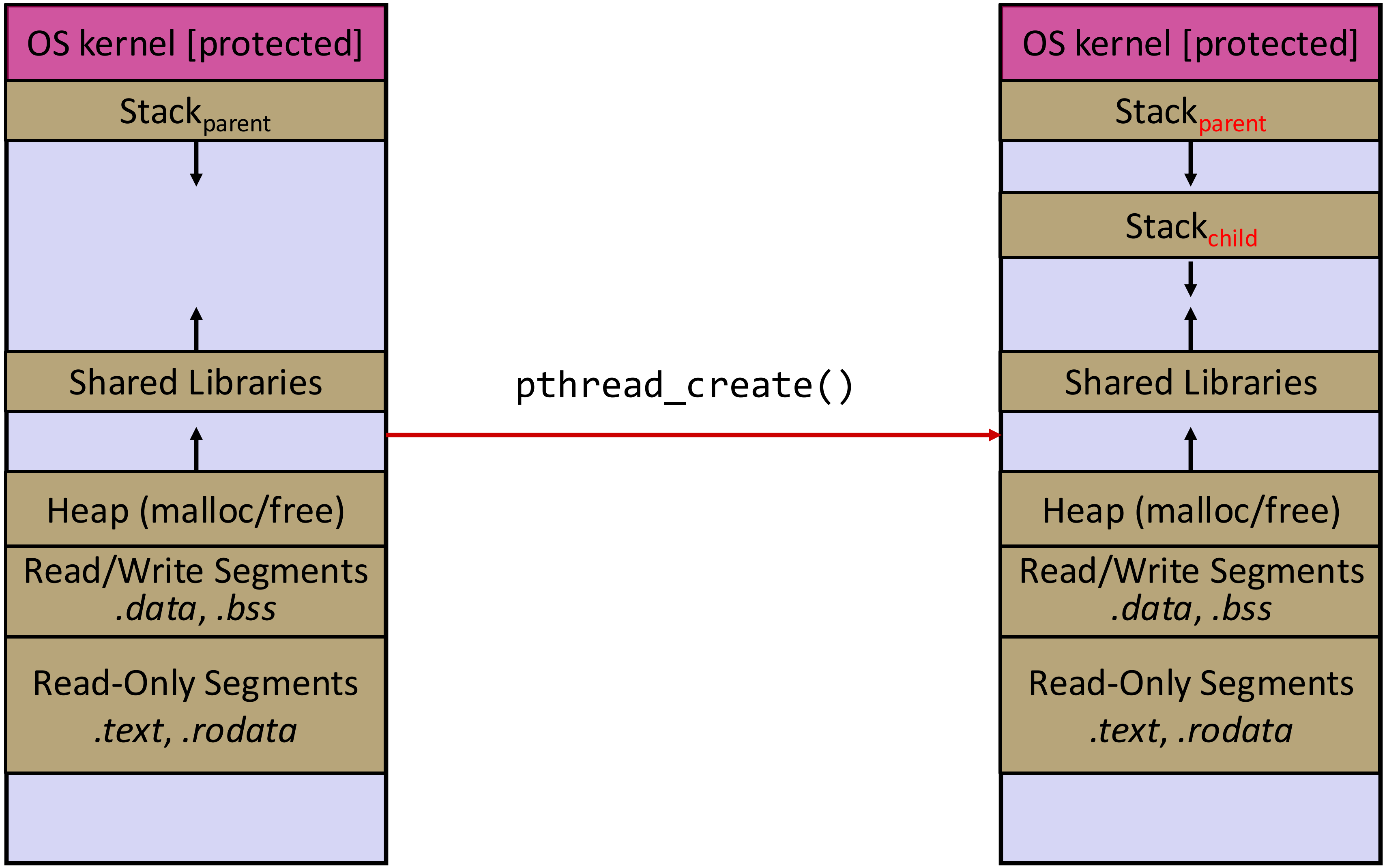
# Lecture Outline

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

# 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/cis5480

- 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/cis4480

- 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 actually kinda unsafe if done wrong (we are doing it wrong in this example), **more on this next laterrrr**

# 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 schedular is agnostic, just another 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.

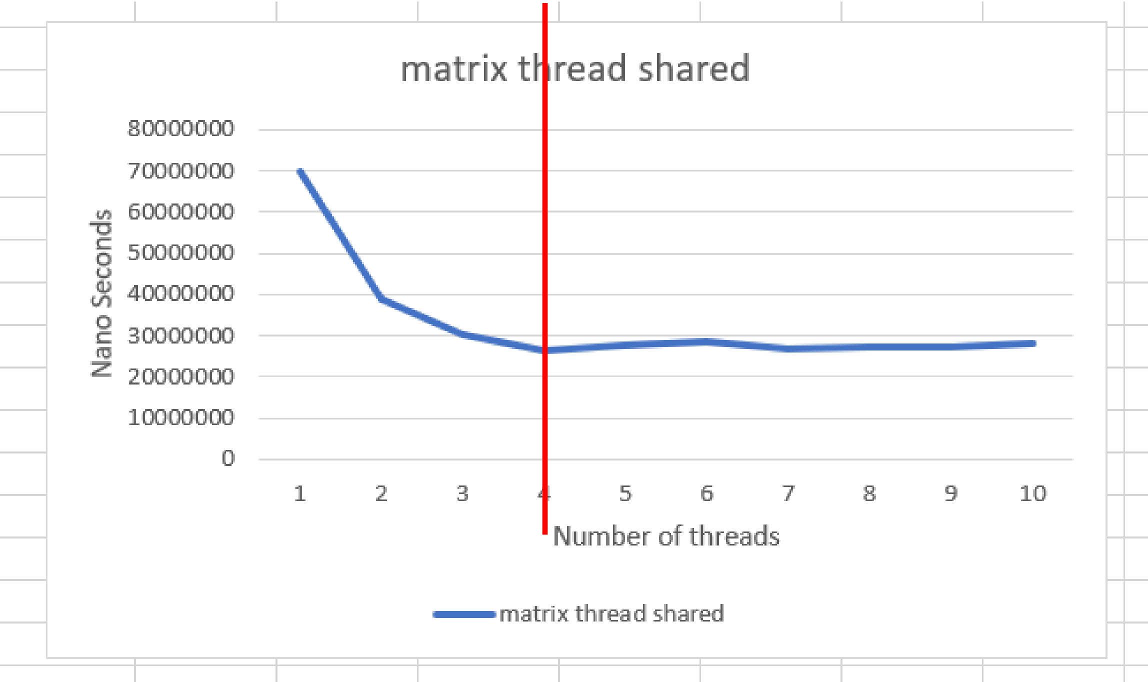
# Parallelism

- 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

Personally got grilled on a question like this when interviewing for the Apple Kernel Team.

# 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

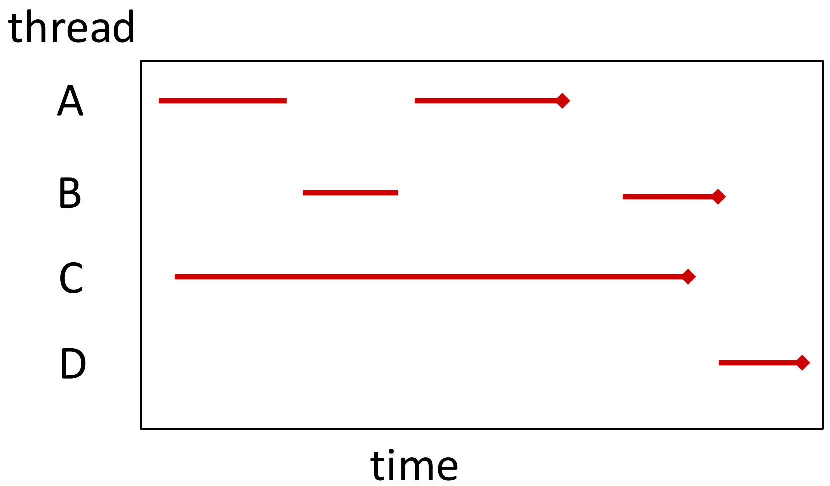
# Parallelism vs Concurrency

- 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

# Poll Everywhere

pollev.com/cis4480

- What are all possible outputs of 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);
}
```

Are these outputs possible?

```
--------------------------
1
2
--------------------------
2
2
--------------------------
1
1
--------------------------
2
1
```

# Visualization

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

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

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

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

# Visualization: Memory

- The variable x is shared across all threads.

main()

int x 1

```
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

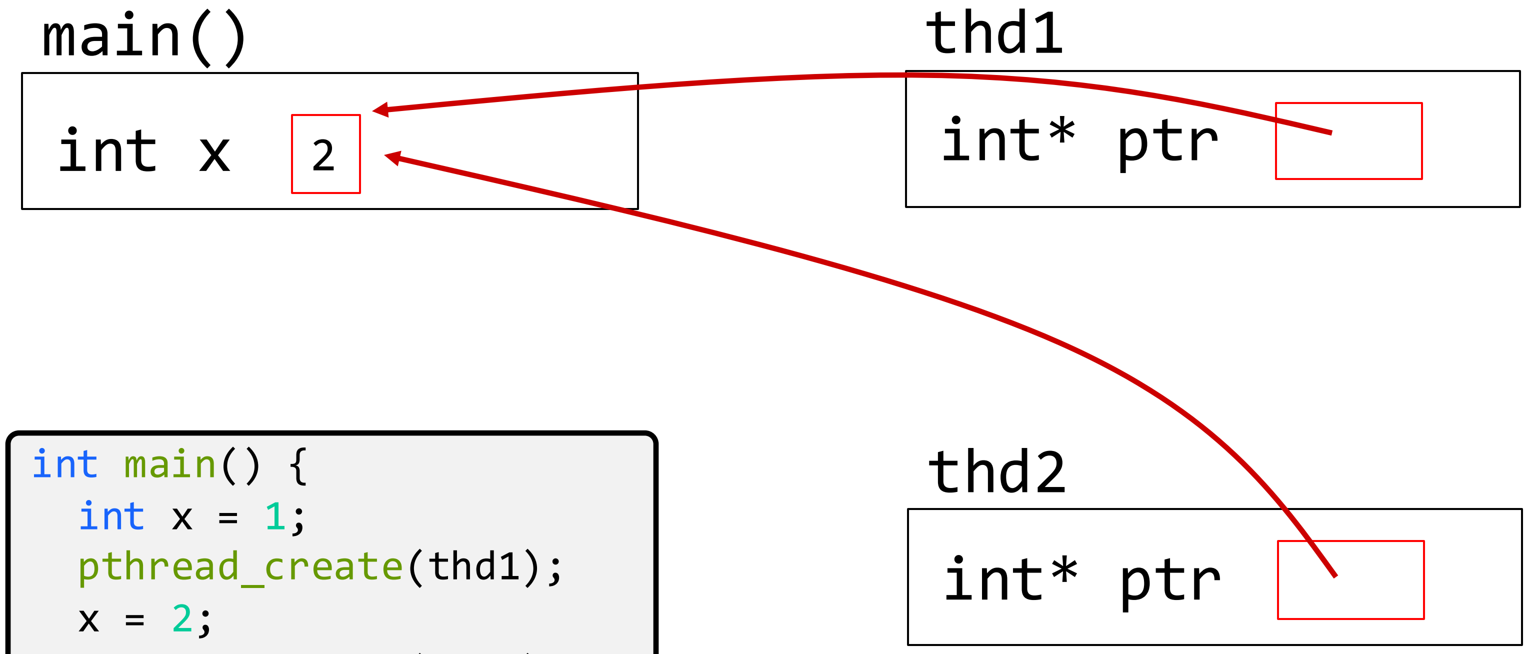
- The variable x is shared across all threads.


```c
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

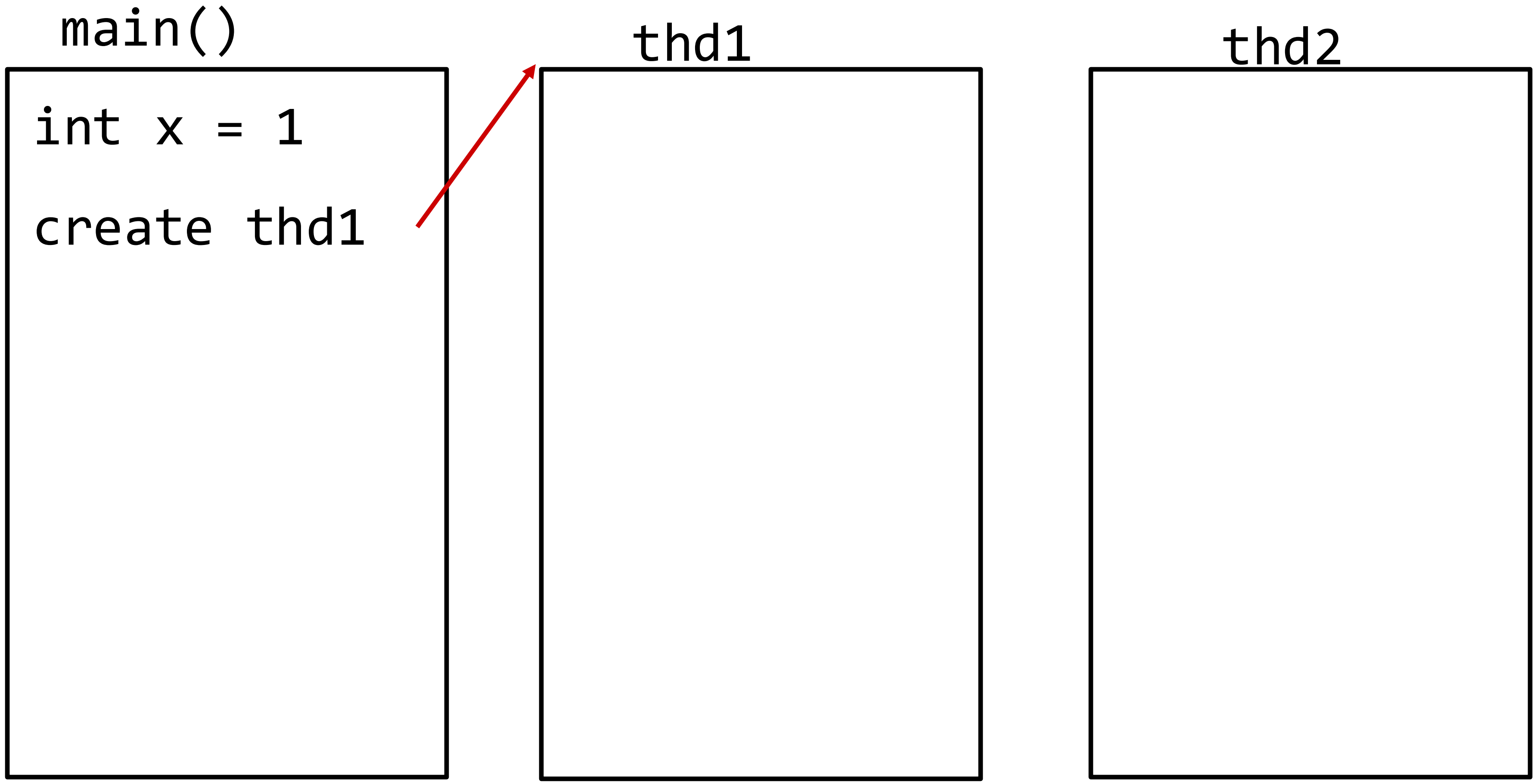
then thread 2 is created

# Visualization: Ordering

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

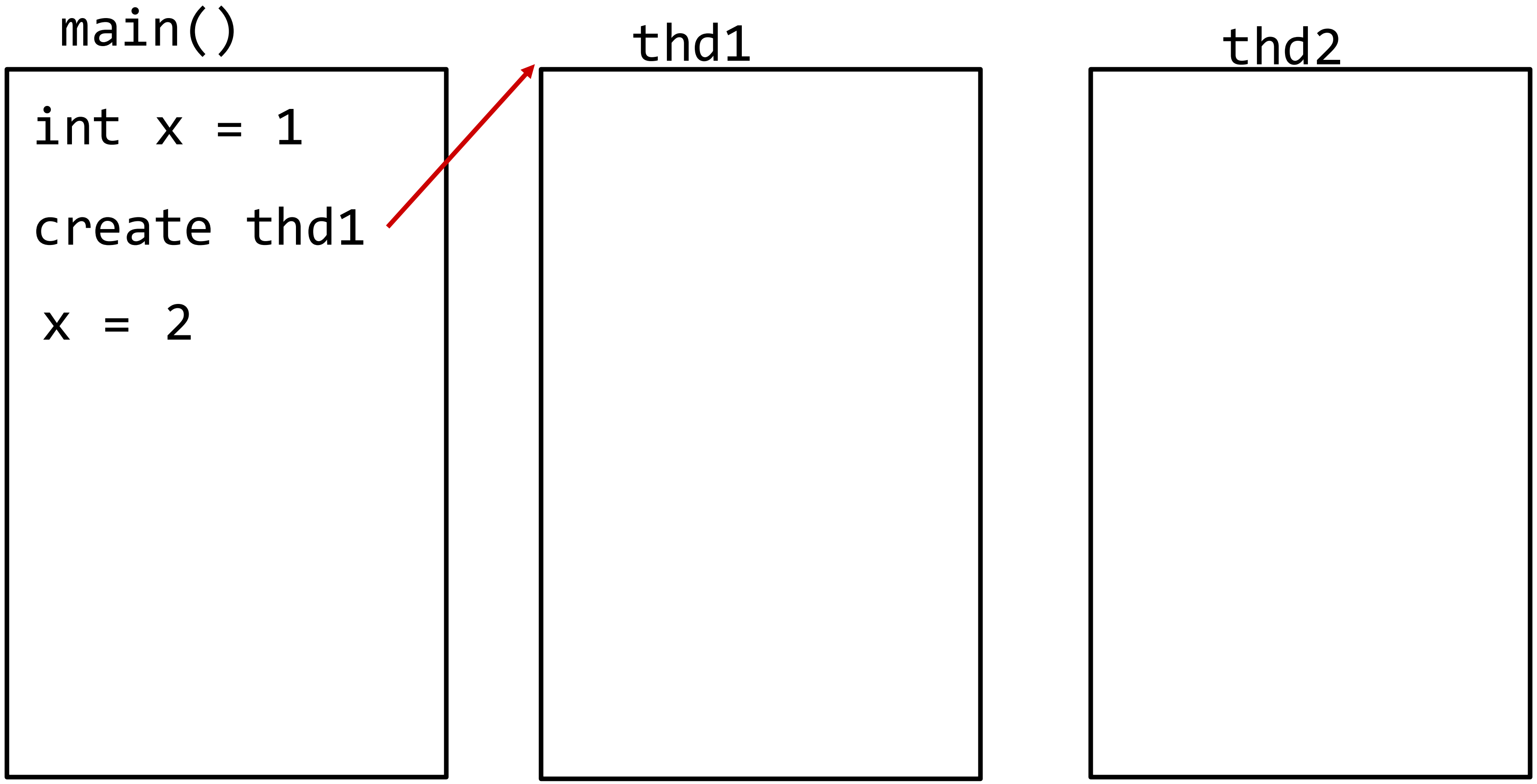
# Visualization: Ordering

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



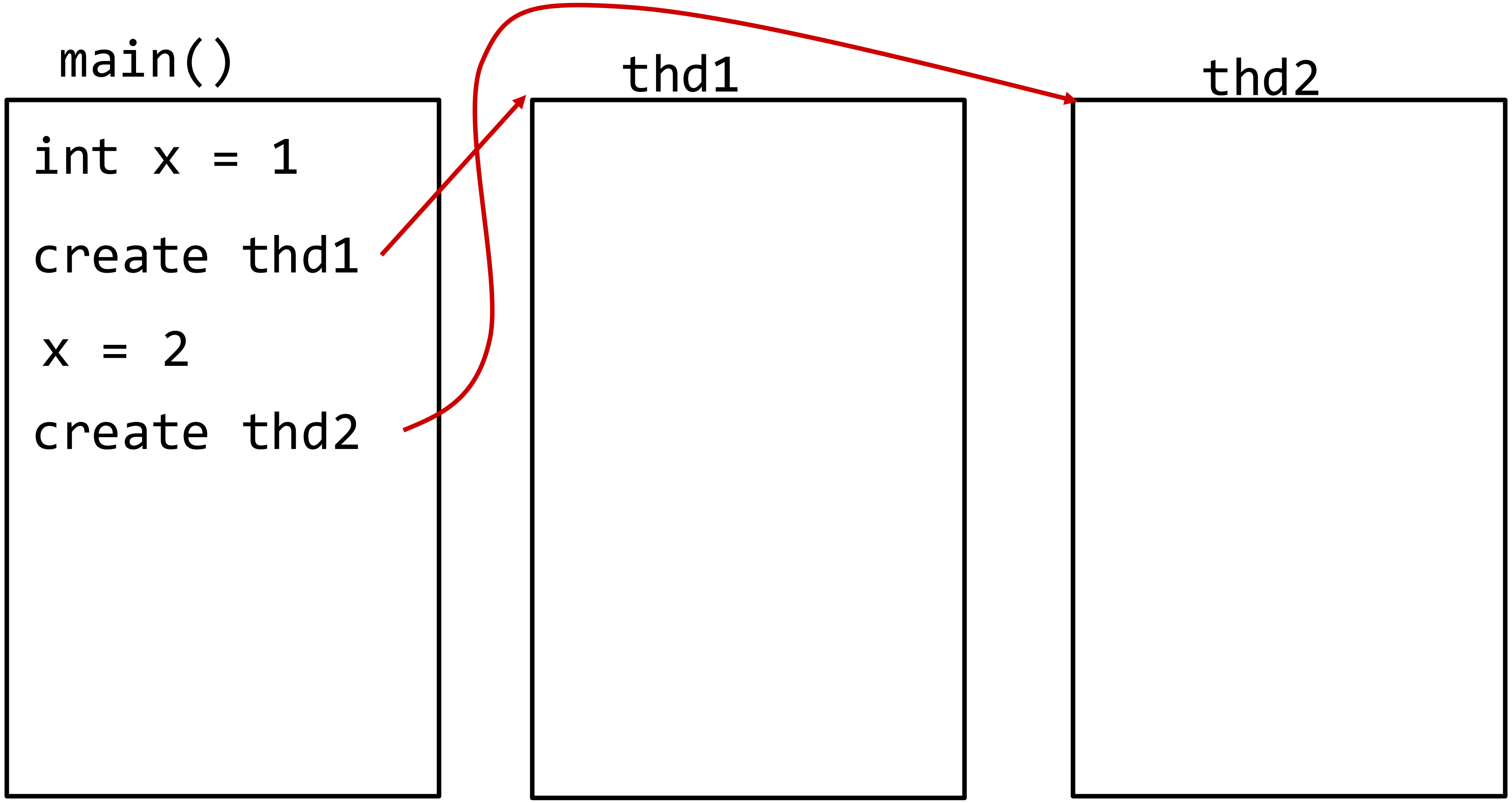
# Visualization: Ordering

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



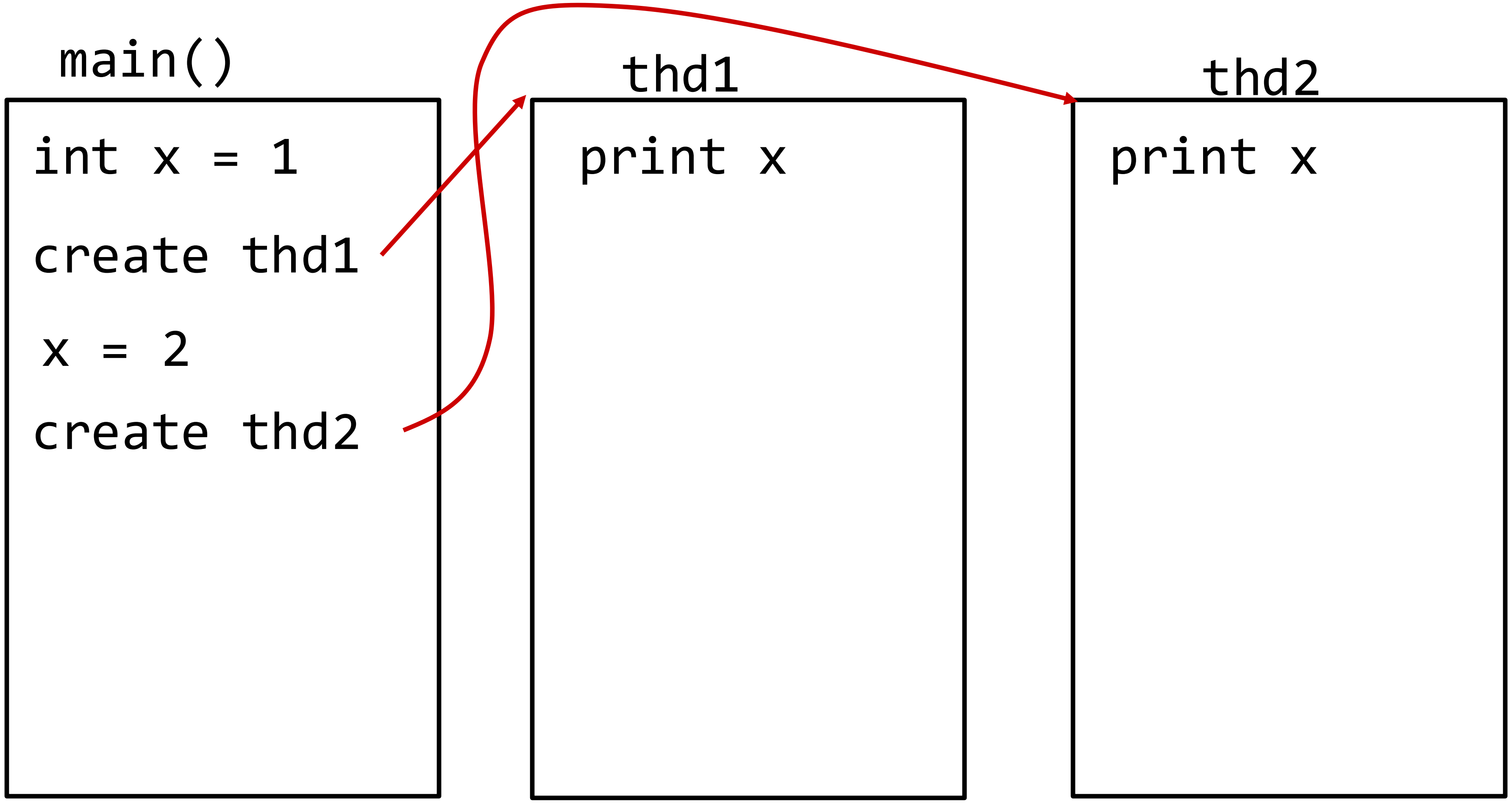
# Visualization: Ordering

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



# Visualization: Ordering

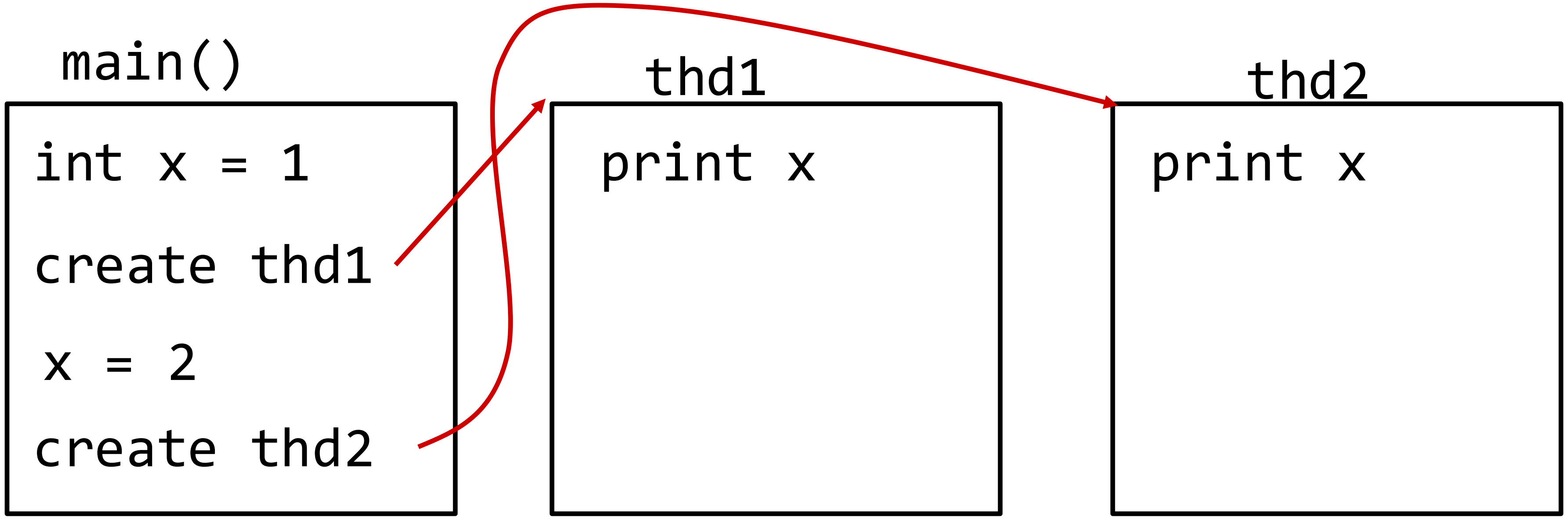
- Threads run concurrently; we can't be sure of the ordering of things across threads.



# 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? <u>**No**</u>. 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);
}
```

| | Address | Value |
|---|---|---|
| x | 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

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

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

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"
```

| | Address | Value |
|---|---|---|
| x | 0x10000 | 2 |

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

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

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

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 = ?

| | Address | Value |
|---|---|---|
| x | 0x10000 | 1 |

pre-empted: taken off a cpu to run another thread.

# Scenario 2:

main

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

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

| | 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:

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

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

main

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"
```

| | Address | Value |
|---|---|---|
| x | 0x10000 | 2 |

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
- **Remember is all assembly, even c is higher level…**

# Why Threads?

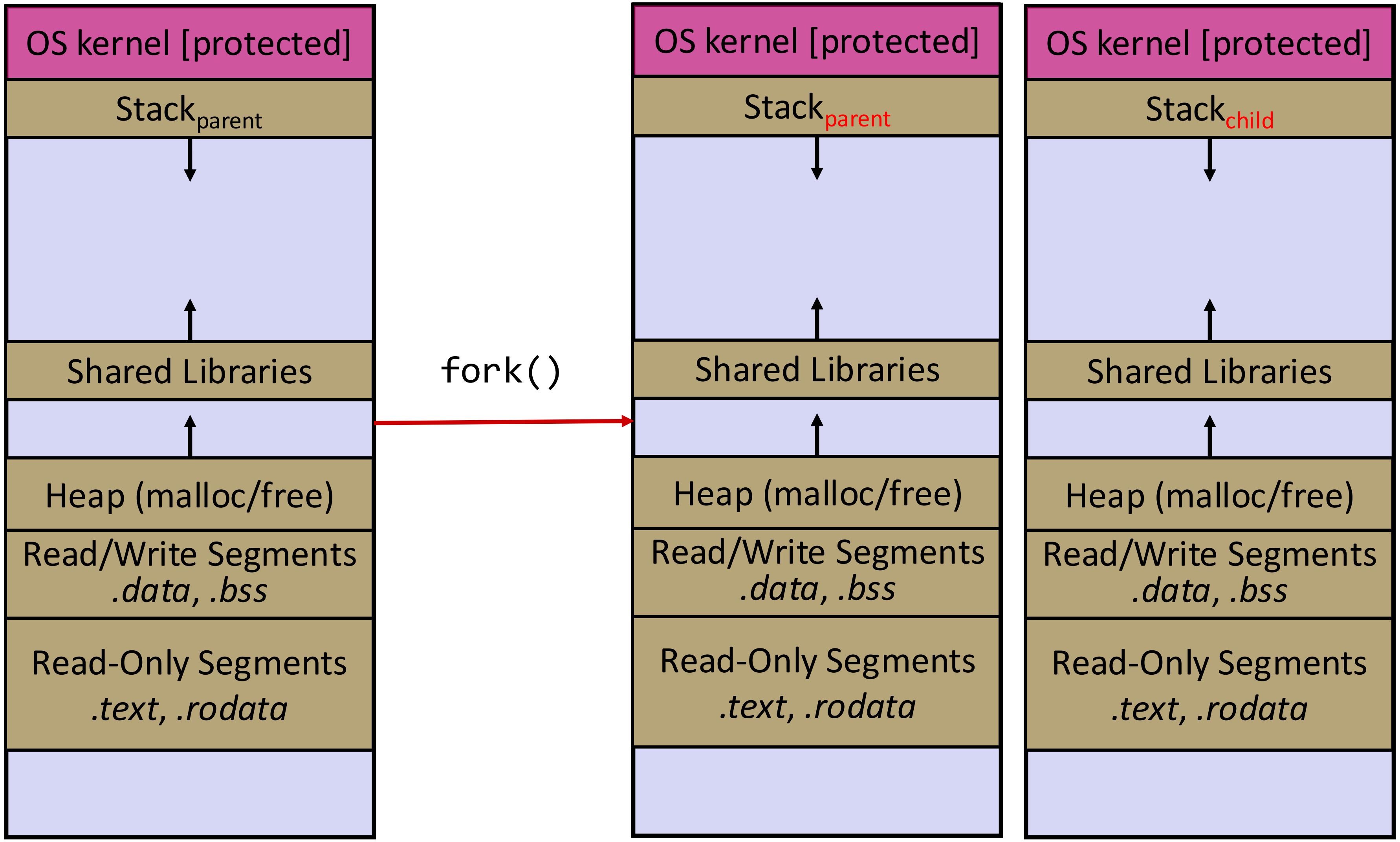
- Advantages:
  - You (mostly) write sequential-looking code
  - Threads can run in parallel if you have multiple CPUs/cores

- 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

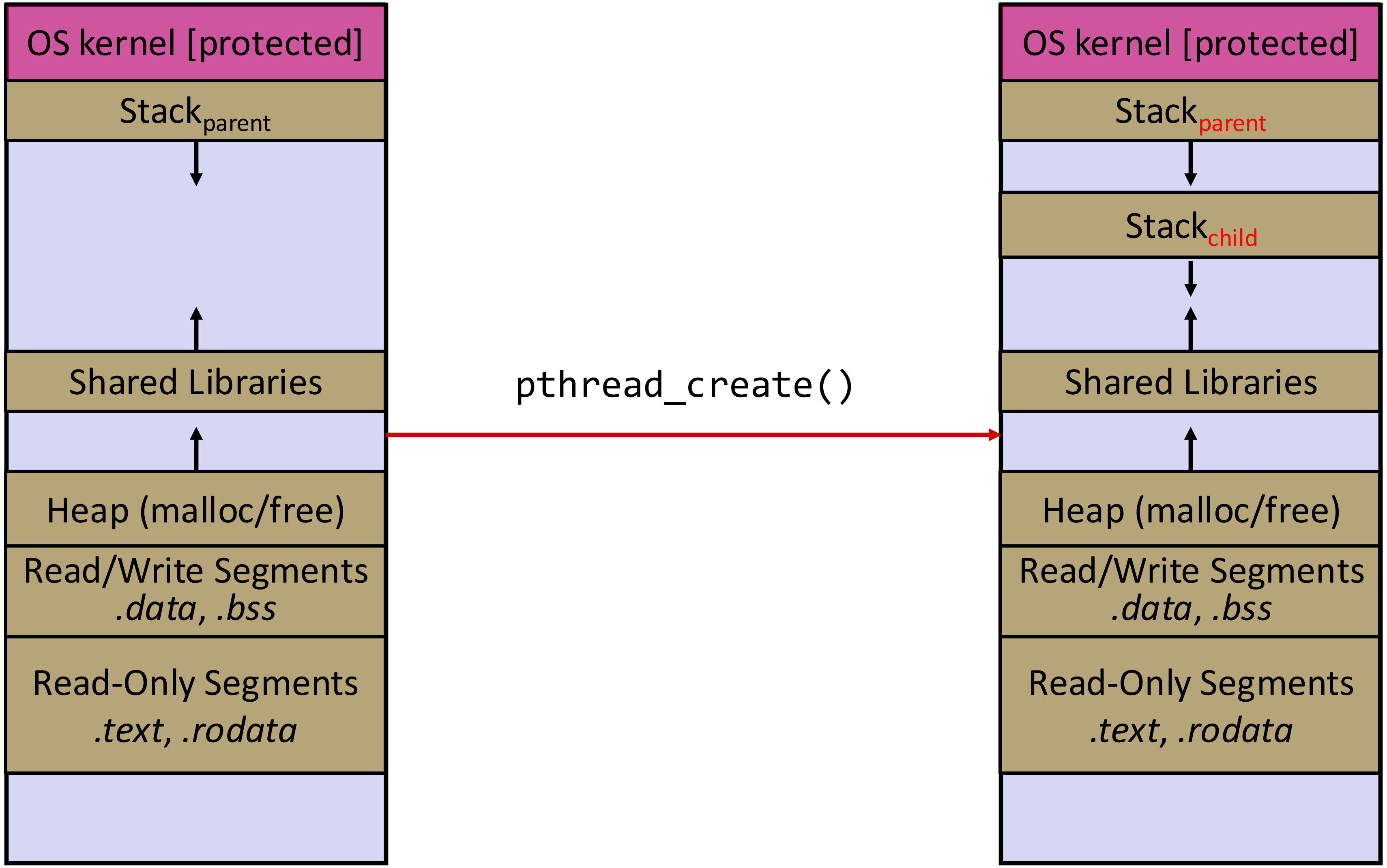
# Threads vs. Processes

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



# 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!