IPC & Threads (Intro)

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

TAs: Theodor Bulacovschi Ash Fujiyama

Poll Everywhere

pollev.com/tqm

- How are you? Any feedback?

Administrivia

- HW06 posted before break
  - Due by end of day Friday due to fall break

- Check-in 06 posted after class
  - Due bye end of day today instead of before lecture (Due to fall break)
  - If you forgot it, do it now!

- Midterm Details
  - In-class on Wed Oct 22nd
  - Posted soon

- HW07: posted by end of week. We don't expect you to work on it till after midterm, but you can start whenever. Shouldn't be too long (hopefully).

Lecture Outline

- **IPC**
- Threads
- Data Races

IPC

- Inter Process Communication
  - Sharing data between processes on a computer

- Communication is important to coordinate processes for some overall goal

- Processes have independent memory, so how do they share data?
  - Answer: need to do something outside of memory
  - Files, pipes, network connection, and other things.

Pipes

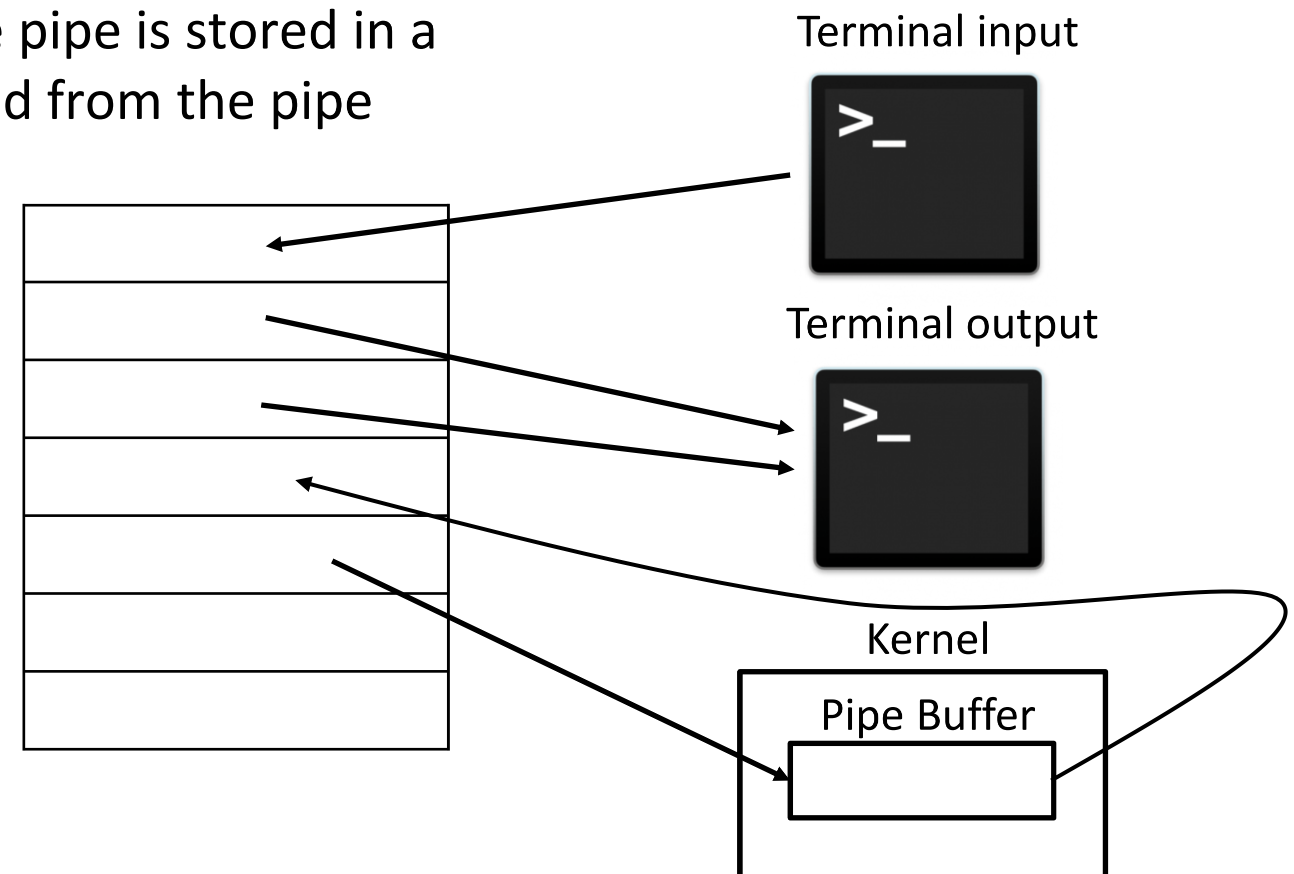
```
int pipe(int pipefd[2]);
```

- Creates a unidirectional data channel for IPC
- Communication through file descriptors! // POSIX ☺
- Takes in an array of two integers, and sets each integer to be a file descriptor corresponding to an "end" of the pipe
- `pipefd[0]` is the reading end of the pipe
- `pipefd[1]` is the writing end of the pipe

- **In addition to copying memory, fork copies the file descriptor table of parent**
- Exec does NOT reset file descriptor table

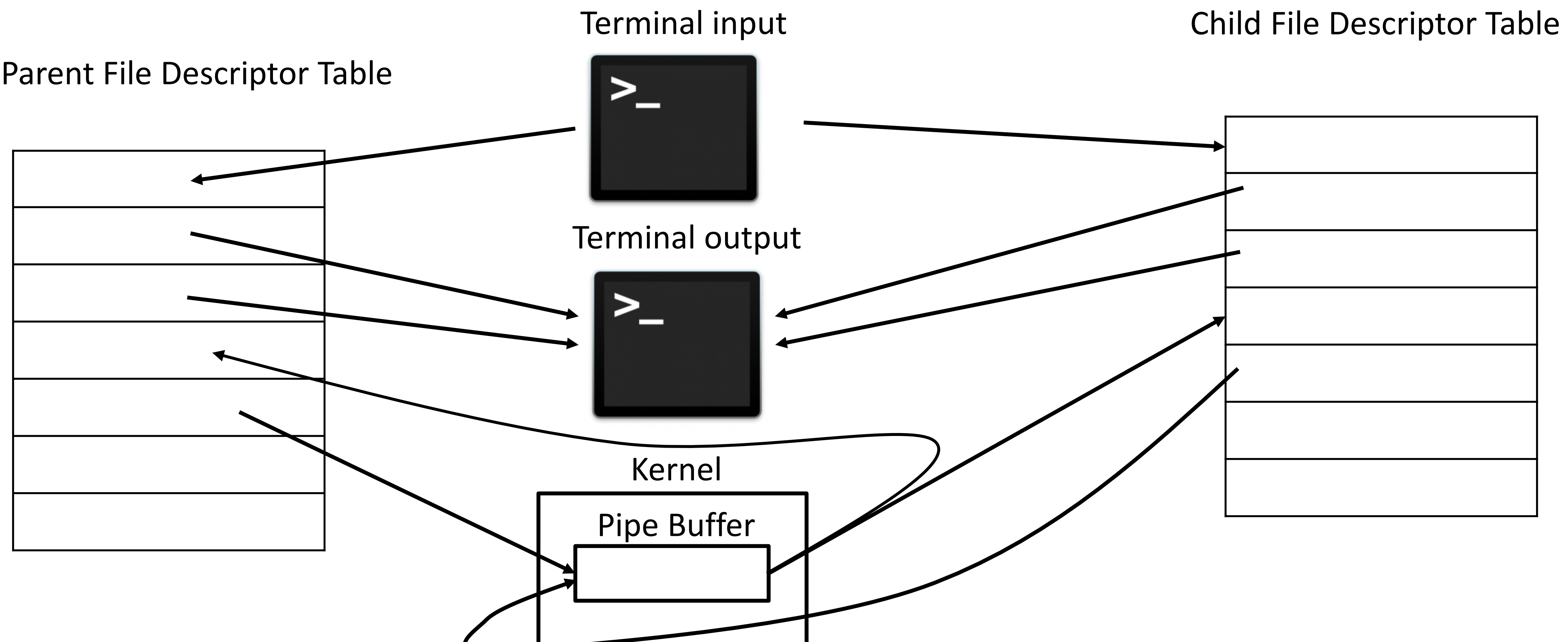
Pipe Visualization

- A pipe can be thought of as a "file" that has distinct file descriptors for reading and writing. This "file" only exists as long as the pipe exists and is maintained by the OS.
  - Data written to the pipe is stored in a buffer until it is read from the pipe



Pipe Visualization

- After fork:

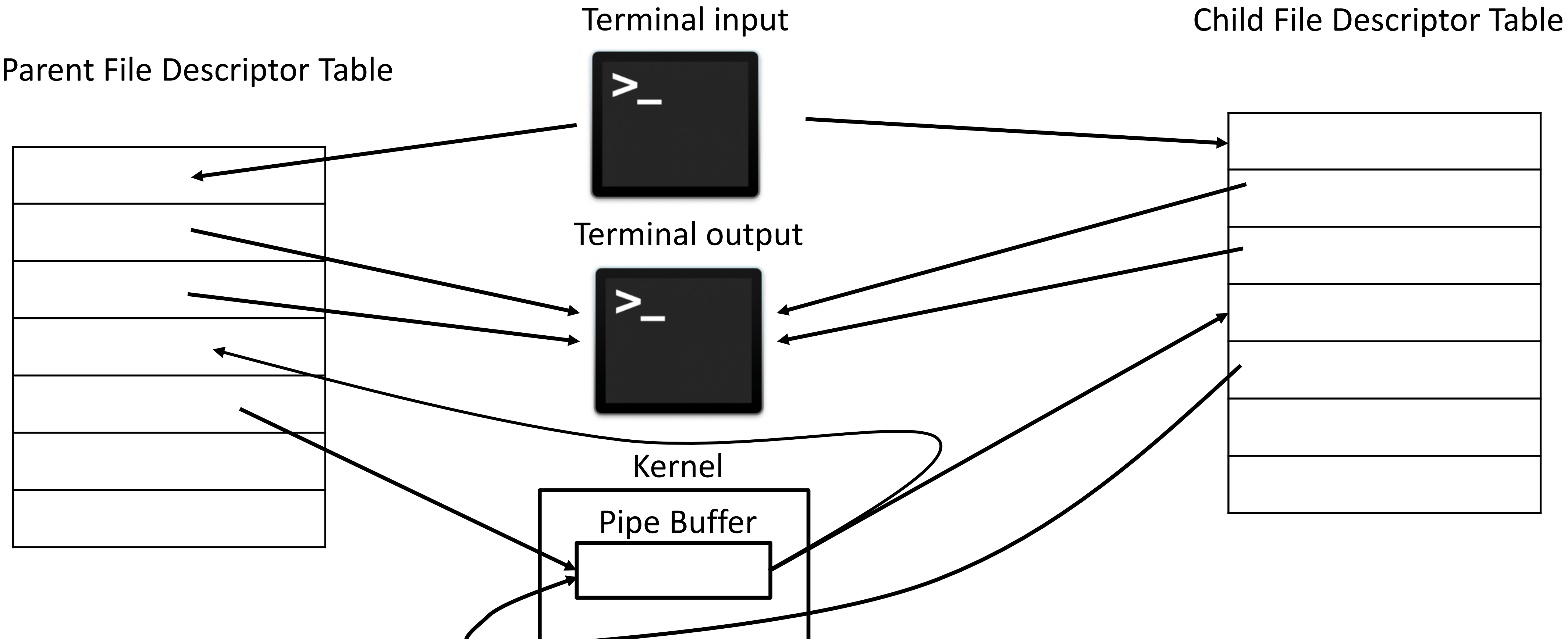


Pipes & EOF

- Many programs will read from a file until they hit EOF and will not terminate until then
- Like reading from the terminal, just because there is nothing in the pipe, does not mean nothing else will ever come through the pipe.
  - EOF is not read in this case
- EOF is only read from a pipe when:
  - There is nothing in the pipe
  - All write ends of the pipe are closed
- **Good practice: CLOSE ALL PIPE FDS YOU ARE DONE WITH**

Pipe Visualization

- If the parent wanted to read the child's stdout from a pipe, what are the proper steps to set that up? (before child calls exec and parent calls wait)



Ed Discussion

- What is the bug in this code (see ed to read it better)

```
int main(int argc, char* argv[]) {
  array<int,2> pipe_fds{};
  pipe(pipe_fds.data());

  pid_t child = fork();
  if (child == 0) {
    // child
    dup2(pipe_fds.at(1), STDOUT_FILENO);
    close(pipe_fds.at(0));

    // should print "hello" to the pipe
    array<const char*, 3> args {"echo", "hello", nullptr};
    execvp(args.at(0), const_cast<char**>(args.data()));
  }

  auto opt = wrapped_read(pipe_fds.at(0));
  if (opt) {
    cout << opt.value() << endl;
  }

  int wstatus;
  waitpid(child, &wstatus, 0);
  return EXIT_SUCCESS;
}
```

Pipes & EOF

- Many programs will read from a file until they hit EOF and will not terminate until then
- Like reading from the terminal, just because there is nothing in the pipe, does not mean nothing else will ever come through the pipe.
  - EOF is not read in this case

- EOF is only read from a pipe when:
  - There is nothing in the pipe
  - All write ends of the pipe are closed

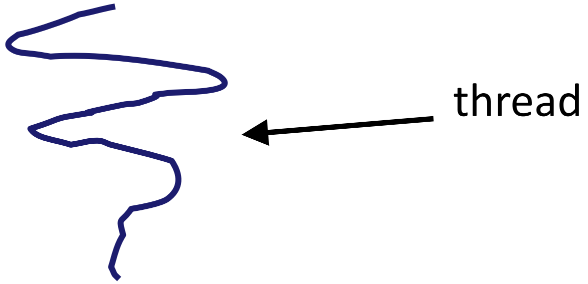
- This is true for other streams like network communication
- **Good practice: CLOSE ALL PIPE FDS YOU ARE DONE WITH**

Lecture Outline

- IPC
- **Threads**
- Data Races

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

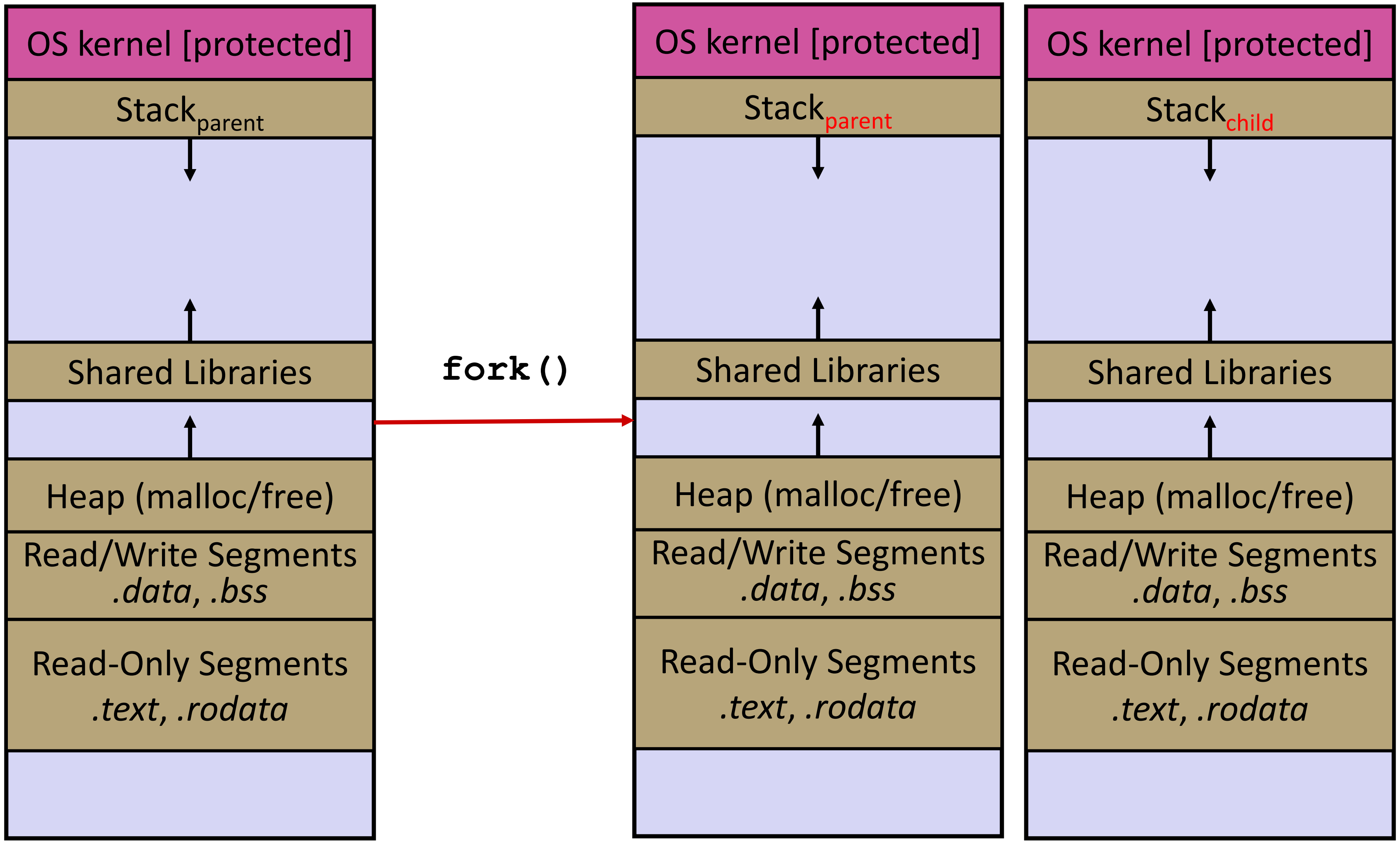


- In most modern OS's:
  - Threads are the *unit of scheduling.*

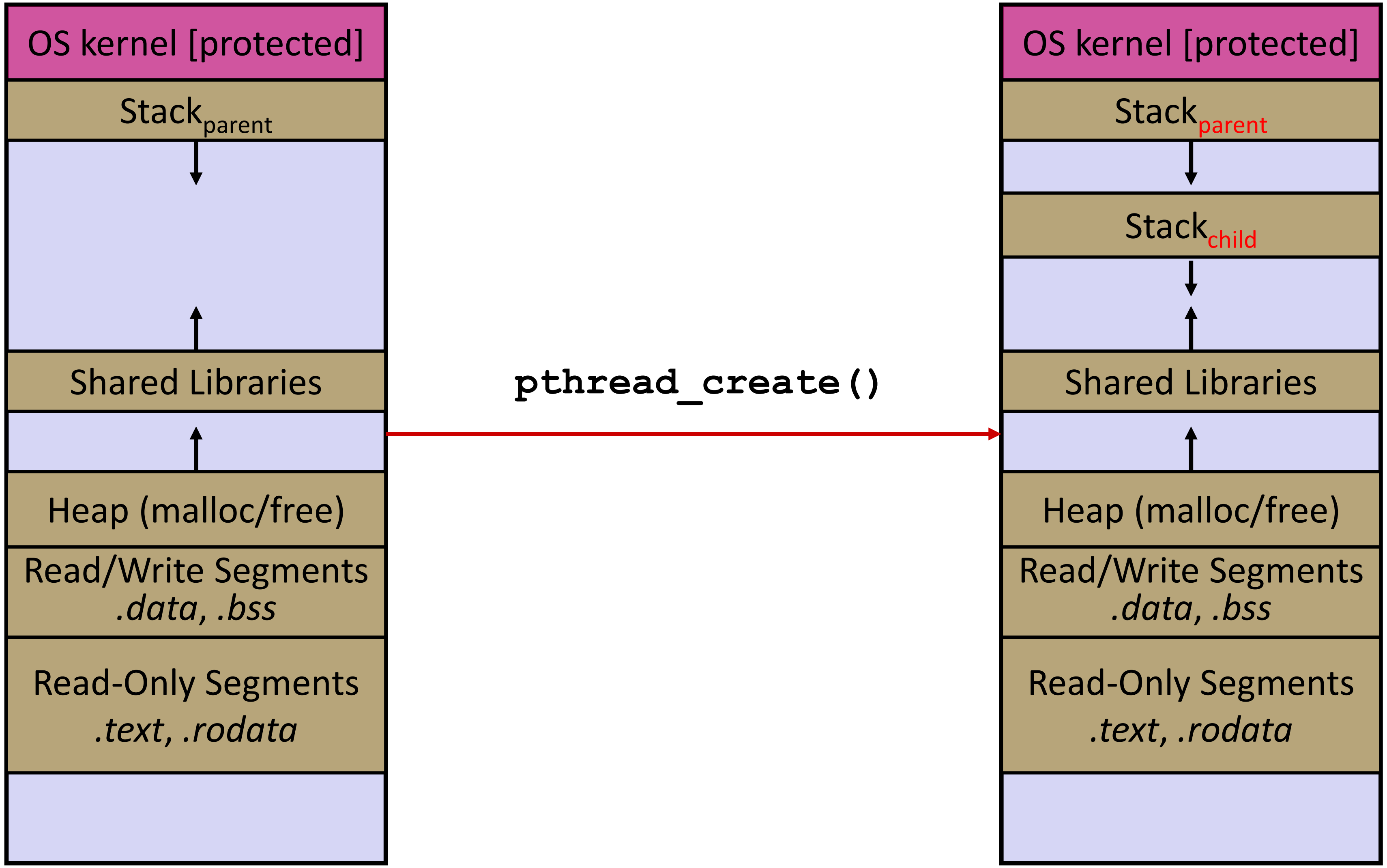
Threads vs. Processes

- 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

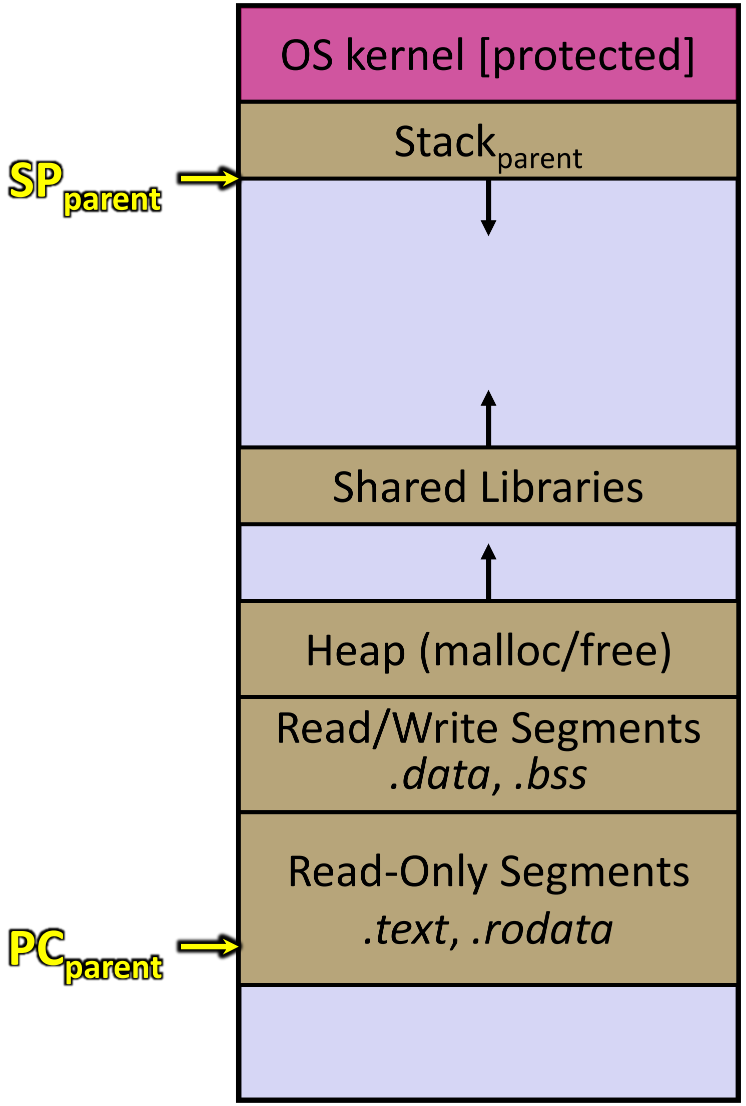


We've been dealing with threads this whole time!

- When we fork() to create a new process, we are cloning the process and the calling thread.
  - After the fork there are two processes, each with their own thread inside of them!

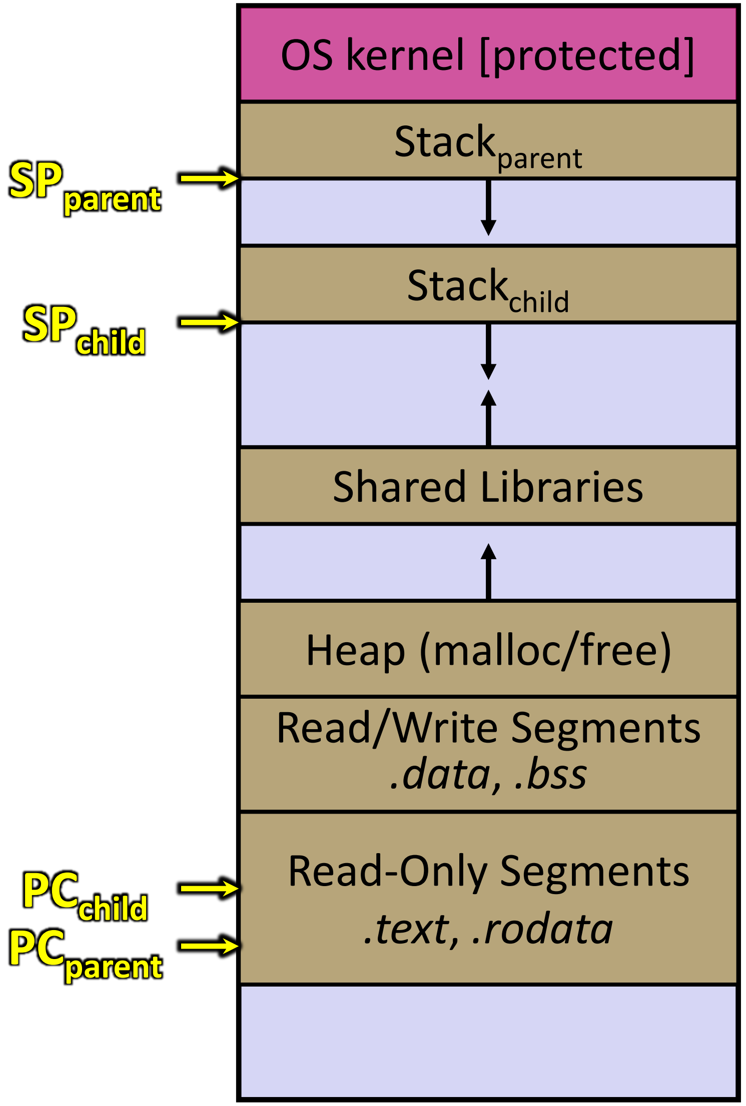
- Similar ideas about ordering and scheduling we learned with processes still apply here!

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

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
    - `g++ -g -Wall -std=c++23 -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

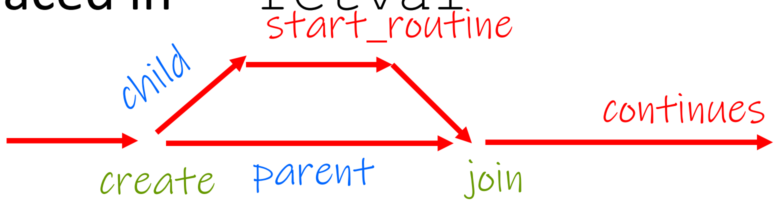
- Creates a new thread into `*thread`, with attributes `*attr` (`NULL` means default attributes)
- Returns **0** on success and an error number on error (can check against error constants)
- The new thread runs **`start_routine`**`(arg)`

child start_routine
continues
pthread_create parent

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.cpp`
  - How do you properly handle memory management?
    - Who allocates and deallocates memory?
    - How long do you want memory to stick around?

Ed Discussion

- What does this print?

```
constexpr int NUM_PROCESSES = 50;
constexpr int LOOP_NUM = 10000;

static int sum_total = 0;

void loop_incr() {
  for (int i = 0; i < LOOP_NUM; ++i) {
    sum_total++;
  }
}

int main(int argc, char** argv) {
  array<pid_t, NUM_PROCESSES> pids{};  // array of process ids

  // create processes to run loop_incr()
  for (int i = 0; i < NUM_PROCESSES; ++i) {
    pids.at(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.at(i), NULL, 0);
  }

  // print out the final sum (expecting NUM_PROCESSES * LOOP_NUM)
  cout << "Total: " << sum_total << endl;

  return EXIT_SUCCESS;
}
```

Ed Discussion

- What does this print?

```
constexpr int NUM_THREADS = 50;
constexpr int LOOP_NUM = 10000;

static int sum_total = 0;

// increment sum_total LOOP_NUM times
void* thread_main(void* arg) {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  return nullptr;  // return type is a pointer
}

int main(int argc, char** argv) {
  array<pthread_t, NUM_THREADS> thds{};  // array of thread ids

  // create threads to run thread_main()
  for (int i = 0; i < NUM_THREADS; i++) {
    pthread_create(&thds.at(i), nullptr, &thread_main, nullptr);
  }

  // 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.at(i), nullptr);
  }

  // print out the final sum (expecting NUM_THREADS * LOOP_NUM)
  cout << "Total: " << sum_total << endl;

  return EXIT_SUCCESS;
}
```

Ed Discussion

- How many times is ":)" printed?

```
void* thread_fn(void* arg) {
  cout << ":)\n";
}

int main() {
  vector<pthread_t> pthds;

  for (int i = 0; i < 4; ++i) {
    pthread_t thd;
    pthread_create(&thd, nullptr, thread_fn, nullptr);
    pthds.push_back(thd);
  }

  for (auto& pthd : pthds) {
    pthread_join(pthd, nullptr);
  }
}
```

Ed Discussion

- What are all possible outputs of this program?

```
void* thrd_fn(void* arg) {
  int* ptr = reinterpret_cast<int*>(arg);
  cout << *ptr << endl;
}

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

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

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() {
  cout << *ptr ...;
  return nullptr;
}
```

```
thrd_fn() {
  cout << *ptr ...;
  return nullptr;
}
```

Visualization: Memory

- The variable x is shared across all threads.

main()

int x $\boxed{1}$

```
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

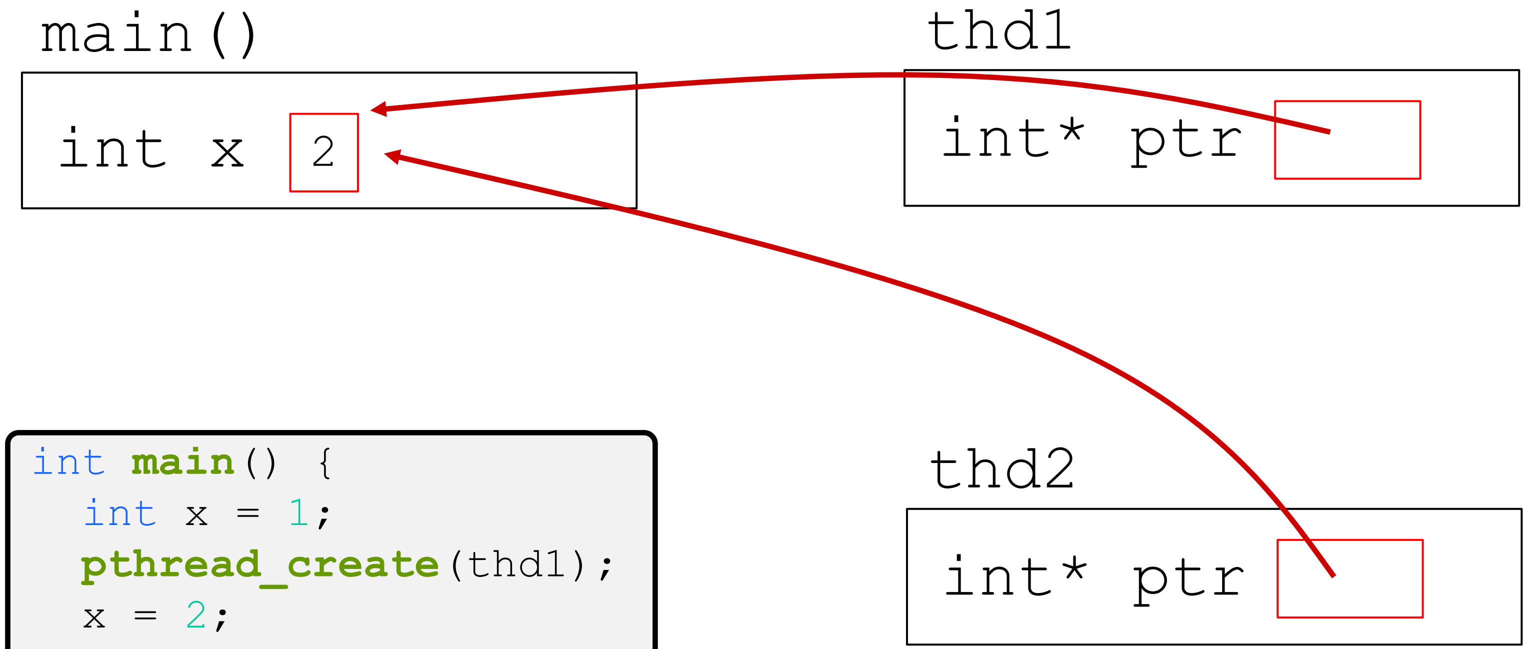
- The variable x is shared across all threads.


```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);
}
```

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
```

Within main(), x is set to 1 before thread 1 is created
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.

`main()`

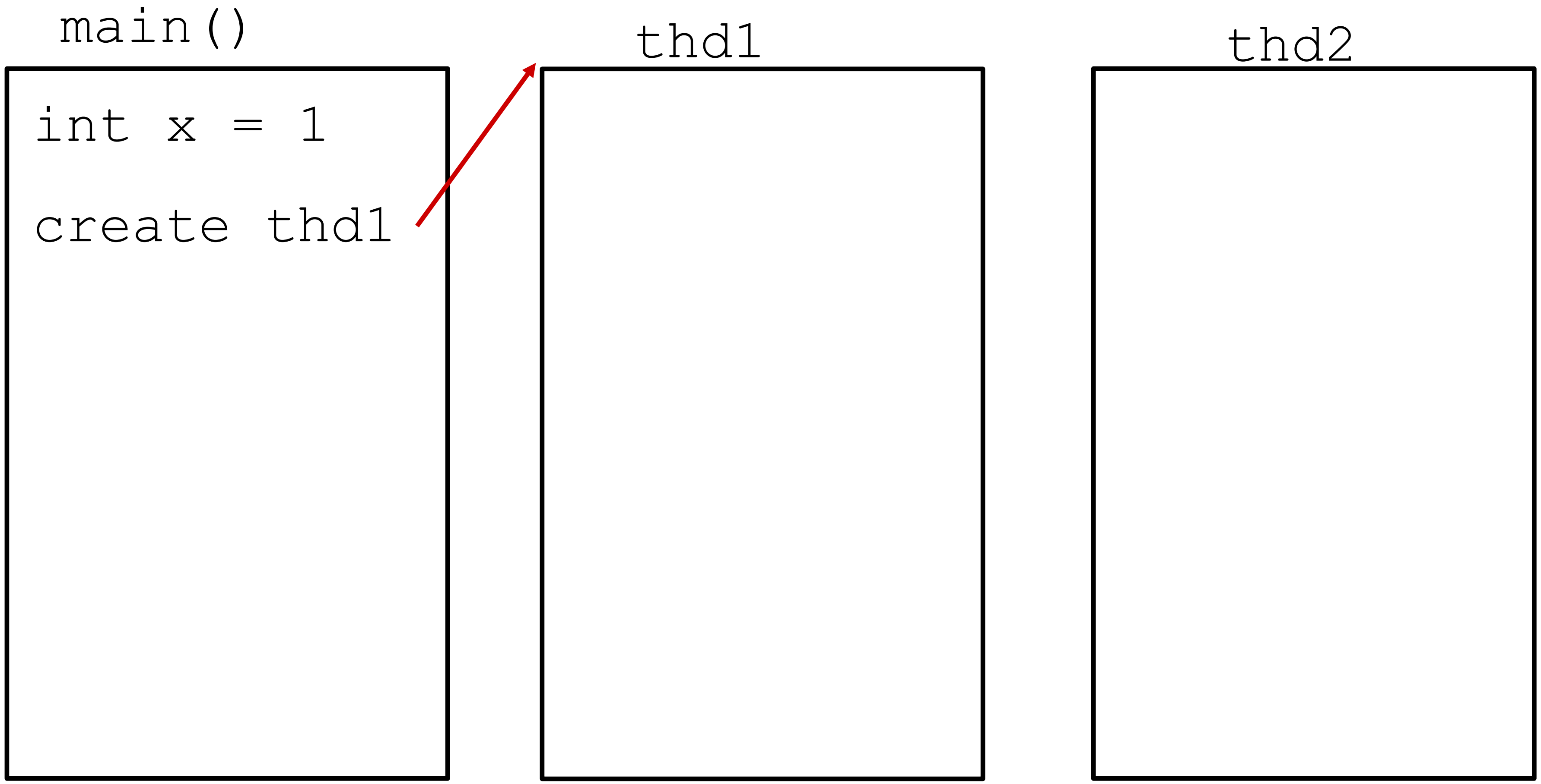
```
int x = 1
```

`thd1`

`thd2`

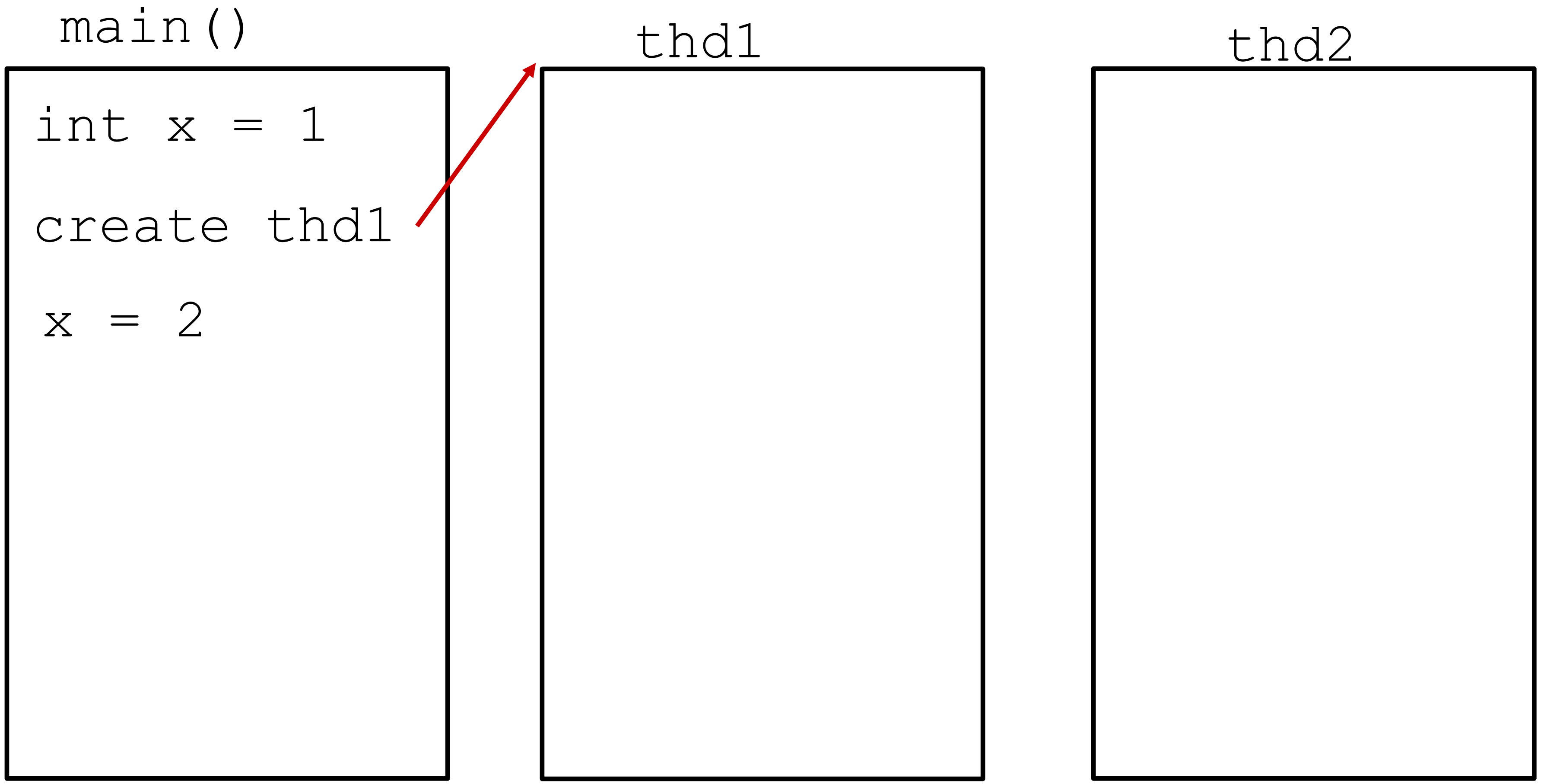
Visualization: Ordering

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



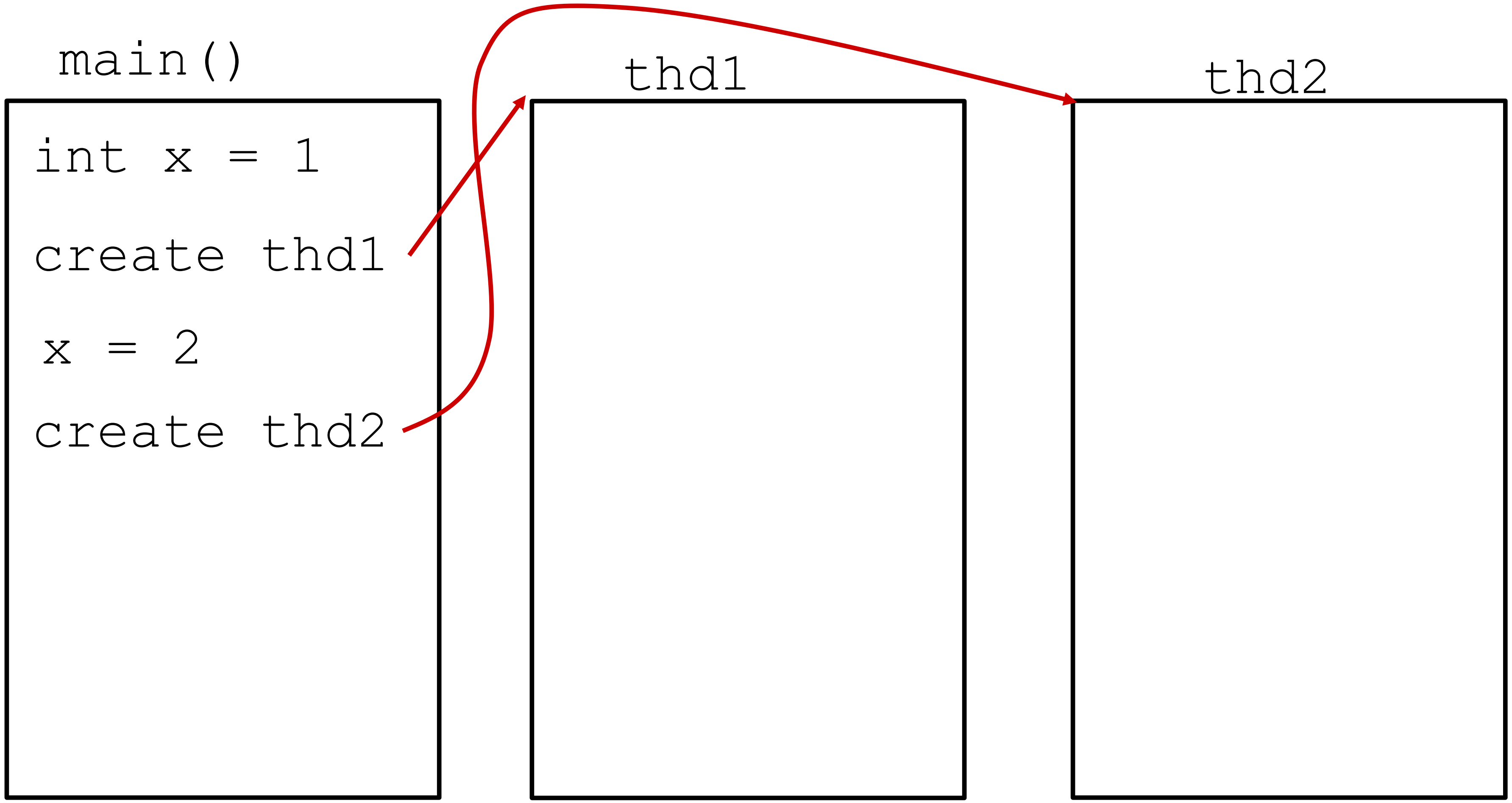
Visualization: Ordering

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



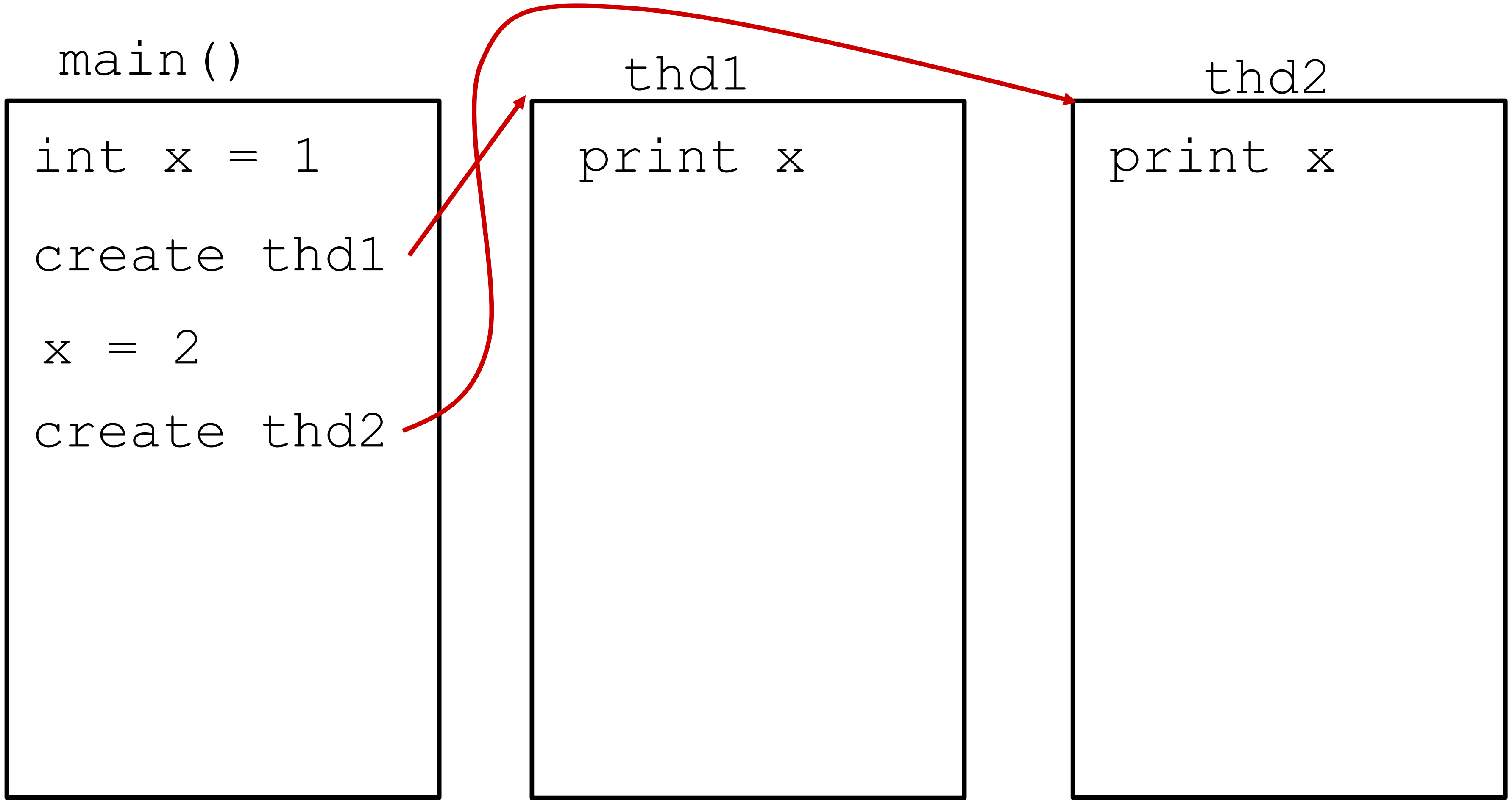
Visualization: Ordering

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



Visualization: Ordering

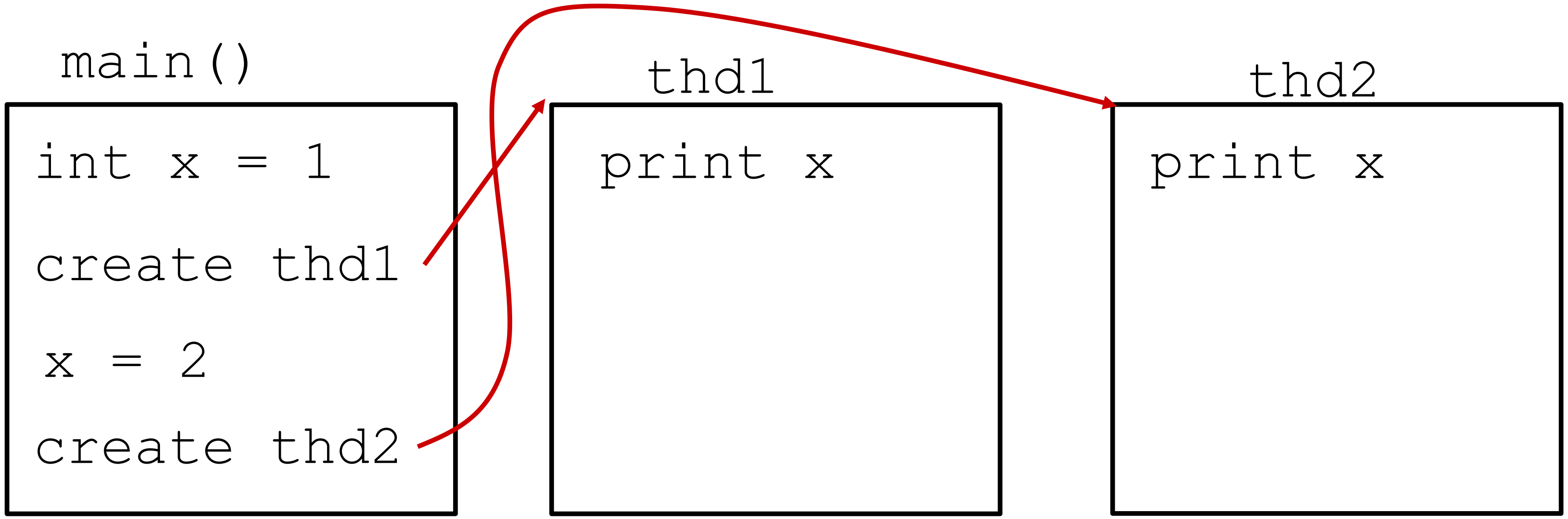
- Threads run concurrently; we can't be sure of the ordering of things across threads.



Visualization: Ordering

This is also why total.cpp allocated 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.



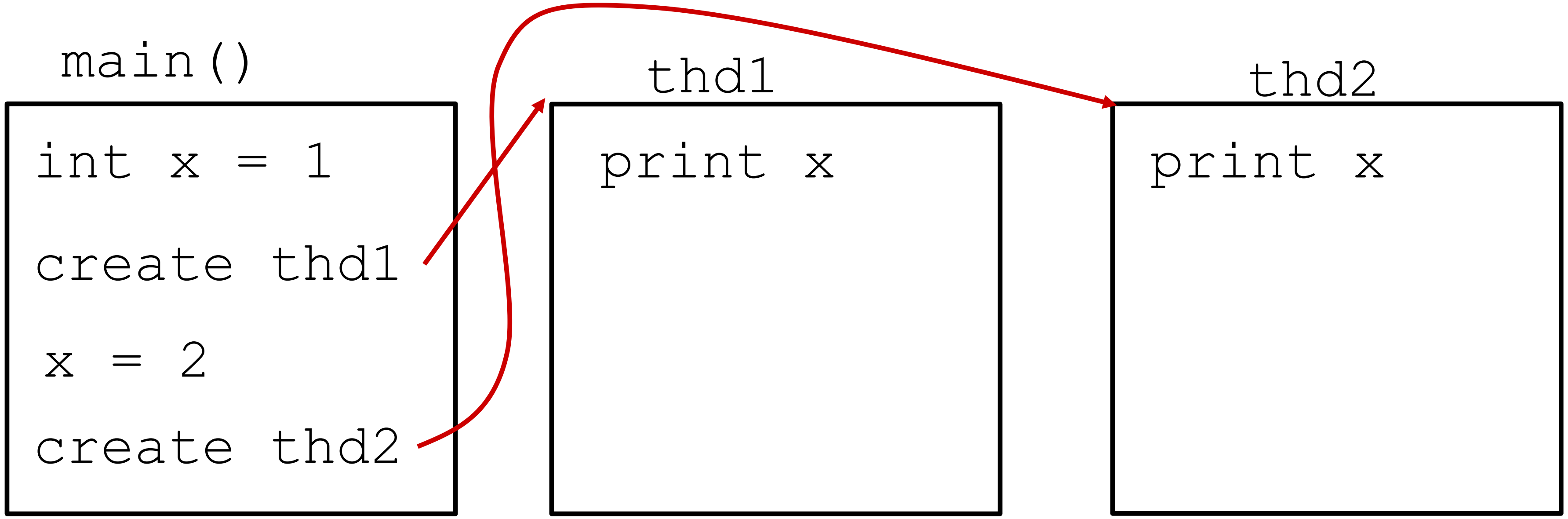
We know that x is initialized to 1 before thd1 is created
We know that x is set to 2 and thd1 is created before thd2 is created

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

Visualization: Ordering

This is also why total.cpp allocated 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.



Technically this diagram is missing something that makes 2 1 possible.

Hint: `basic_ostream& operator<<( int value );`

(cout is an ostream)

Anyone see how?

Lecture Outline

- IPC
- Threads
- **Data Races**

Shared Resources

- Some resources are shared between threads and processes

- Thread Level:
  - Memory
  - Things shared by processes

- Process level
  - I/O devices
    - Files
    - terminal input/output
    - The network

Issues arise when we try to shared things

Data Races

- Two memory accesses form a data race if different threads access the same location, and at least one is a write, and they occur one after another
  - Means that the result of a program can vary depending on chance **(which thread ran first? When did a thread get interrupted?)**

Data Race Example

- If your fridge has no milk,
  then go out and buy some more
  - What could go wrong?

```
if (!milk) {
  buy milk
}
```

- If you live alone:

- If you live with a roommate:

Talk amongst yourselves

- Idea: leave a note!
  - Does this fix the problem?

```
if (!note) {
  if (!milk) {
    leave note
    buy milk
    remove note
  }
}
```

A. Yes, problem fixed

B. No, could end up with no milk

C. No, could still buy multiple milk

D. We’re lost…

Talk amongst yourselves

- Idea: leave a note!
  - Does this fix the problem?

```
if (!note) {
  if (!milk) {
    leave note
    buy milk
    remove note
  }
}
```

We can be interrupted between checking note and leaving note ☹

A. Yes, problem fixed

B. No, could end up with no milk

C. No, could still buy multiple milk

D. We're lost…

*There are other possible scenarios that result in multiple milks

you	roommate
Check note	
	Check note
Check milk	
Leave note	
	Check milk
	Leave note
	Buy milk
Buy milk	

time

Threads and Data Races

- Data races might interfere in painful, non-obvious ways, depending on the specifics of where the data race occurs.
- <u>Example</u>: two threads try to read from and write to the same shared memory location
  - Could get “correct” answer
  - Could accidentally read old value
  - One thread’s work could get “lost” (Like the first note left in the milk example)
- <u>Example</u>: two threads try to push an item onto the head of the linked list at the same time
  - Could get “correct” answer
  - Could get different ordering of items
  - Could break the data structure! ☠

Remember this?

- What does this print?

```
#define NUM_THREADS 50
#define LOOP_NUM 100

int sum_total = 0;

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++) {
    if (pthread_create(&thds[i], NULL, &thread_main, NULL) != 0) {
      fprintf(stderr, "pthread_create failed\n");
    }
  }

  // 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++) {
    if (pthread_join(thds[i], NULL) != 0) {
      fprintf(stderr, "pthread_join failed\n");
    }
  }

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

  return EXIT_SUCCESS;
}
```

Increment Data Race

- What seems like a single operation is actually multiple operations in one. The increment looks something like this in assembly:

```
++sum_total
```

```
LOAD  sum_total into R0
ADD   R0 R0 #1
STORE R0 into sum_total
```

- What happens if we context switch to a different thread while executing these three instructions?

- **Reminder: Each thread has its own registers to work with. Each thread would have its own R0**

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 0

Thread 0 **R0 = 0**

```
LOAD  sum_total into R0
```

Thread 1

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 0

Thread 0 **R0 = 0**

```
LOAD  sum_total into R0
```

Thread 1 **R0 = 0**

```
LOAD  sum_total into R0
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 0

Thread 0 **R0 = 0**

```
LOAD   sum_total into R0
```

Thread 1 **R0 = 1**

```
LOAD   sum_total into R0
ADD    R0 R0 #1
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 1

Thread 0 **R0 = 0**

```
LOAD  sum_total into R0
```

Thread 1 **R0 = 1**

```
LOAD  sum_total into R0
ADD   R0 R0 #1
STORE R0 into sum_total
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 1

Thread 0 **R0 = 1**

```
LOAD   sum_total into R0



ADD    R0 R0 #1
```

Thread 1 **R0 = 1**

```
LOAD   sum_total into R0
ADD    R0 R0 #1
STORE  R0 into sum_total
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 1

Thread 0 **R0 = 1**

```
LOAD  sum_total into R0



ADD   R0 R0 #1
STORE R0 into sum_total
```

Thread 1 **R0 = 1**

```
LOAD  sum_total into R0
ADD   R0 R0 #1
STORE R0 into sum_total
```

- With this example, we could get 1 as an output instead of 2, even though we executed ++sum_total twice

Synchronization

- Synchronization is the act of preventing two (or more) concurrently running threads from interfering with each other when operating on shared data
  - Need some mechanism to coordinate the threads
    - “Let me go first, then you can go”
  - Many different coordination mechanisms have been invented

- Goals of synchronization:
  - Liveness – ability to execute in a timely manner (informally, “something good eventually happens”)
  - Safety – avoid unintended interactions with shared data structures (informally, “nothing bad happens”)

Lock Synchronization

- Use a "Lock" to grant access to a *critical section* so that only one thread can operate there at a time
  - Executed in an uninterruptible (*i.e.* atomic) manner

- Lock Acquire
  - Wait until the lock is free, then take it

- Lock Release
  - Release the lock
  - If other threads are waiting, wake exactly one up to pass lock to

- Pseudocode:

```
// non-critical code

lock.acquire();    block
                   if locked
// critical section
lock.release();

// non-critical code
```

Lock API

- Locks are constructs that are provided by the operating system to help ensure synchronization
  - Often called a mutex or a semaphore
- Only one thread can acquire a lock at a time,
  No thread can acquire that lock until it has been released
- Has memory barriers built into it and usually uses TSL to ensure that acquiring the lock is atomic (more on TSL and memory barriers in a little bit)

pthreads and Locks

- Another term for a lock is a mutex ("mutual exclusion")
  - `pthread.h` defines datatype `pthread_mutex_t`

- ```
  int pthread_mutex_init(pthread_mutex_t* mutex,
                         const pthread_mutexattr_t* attr);
  ```
  - Initializes a mutex with specified attributes

- ```
  int pthread_mutex_lock(pthread_mutex_t* mutex);
  ```
  - Acquire the lock – blocks if already locked Un-blocks when lock is acquired

- ```
  int pthread_mutex_unlock(pthread_mutex_t* mutex);
  ```
  - Releases the lock

- ```
  int pthread_mutex_destroy(pthread_mutex_t* mutex);
  ```
  - "Uninitializes" a mutex – clean up when done

pthread Mutex Examples

- See `total.cpp`
  - Data race between threads
- See `total_locking.cpp`
  - Adding a mutex fixes our data race
- How does `total_locking` compare to sequential code and to `total`?
  - Likely *slower* than both– only 1 thread can increment at a time, and must deal with checking the lock and switching between threads
  - One possible fix: each thread increments a local variable and then adds its value (once!) to the shared variable at the end
    - See `total_locking_better.cpp`

Ed Discussion

- The code below has three functions that could be executed in separate threads. Note that these are not thread entry points, just functions used by threads:
  - Assume that "lock" has been initialized

```
1  // global variables
2  pthread_mutex_t lock;
3  int g =    0;
4  int k = 0;
5
6  void fun1() {
7    pthread_mutex_lock(&lock);
8    g += 3;
9    pthread_mutex_unlock(&lock);
10   k++;
11 }
12
13 void fun2(int a, int b) {
14   g += a;
15   a += b;
16   k = a;
17 }
18
19 void fun3() {
20   pthread_mutex_lock(&lock);
21   g = k + 2;
22   pthread_mutex_unlock(&lock);
23 }
```

- Thread-1 executes line 8 while Thread-2 executes line 21. Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

- Thread-1 executes line 15 while Thread-2 executes line 15. Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

Ed Discussion

- The code below has three functions that could be executed in separate threads. Note that these are not thread entry points, just functions used by threads:
  - Assume that "lock" has been initialized

- Thread-1 executes line 8 while Thread-2 executes line 14
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

- Thread-1 executes line 14 while Thread-2 executes line 16.
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

```
1   // global variables
2   pthread_mutex_t lock;
3   int g =    0;
4   int k = 0;
5
6   void fun1() {
7     pthread_mutex_lock(&lock);
8     g += 3;
9     pthread_mutex_unlock(&lock);
10    k++;
11  }
12
13  void fun2(int a, int b) {
14    g += a;
15    a += b;
16    k = a;
17  }
18
19  void fun3() {
20    pthread_mutex_lock(&lock);
21    g = k + 2;
22    pthread_mutex_unlock(&lock);
23  }
```

Ed Discussion

- Consider the code below, which outputs are possible?

```
int main() {
  pthread_t thd1, thd2;

  int* arg1 = new int(2);
  int* arg2 = new int(3);

  pthread_create(&thd1, nullptr, thread_fn, arg1);
  pthread_create(&thd2, nullptr, thread_fn, arg2);

  pthread_join(thd1, nullptr);
  pthread_join(thd2, nullptr);

  cout << counter << endl;

  return EXIT_SUCCESS;
}
```

```
static int counter = 0; // global var

void* thread_fn(void* arg) {
  int amount = *static_cast<int*>(arg);
  for (int i = 0; i < amount; ++i) {
    counter += 1;
  }
  delete arg;
  return nullptr;
}
```

That’s all for now!

- Next time:
  - Parallelism vs concurrency
  - More processes vs threads
  - Parallel Algorithms

- Hopefully you are doing well ☺