Threads & Mutex
Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk
Angie Cao
Aniket Ghorpade
Ashwin Alaparthi
Austin Lin
Hassan Rizwan
Lobi Zhao
Pearl Liu
Perrie Quek

Poll Everywhere

pollev.com/tqm

- What is your favourite programming language?

Administrivia

- HW07 – File Readers
  - Posted☺
  - Due Friday 3/28 at midnight, leaving open till Sunday night tho
  - AG posted soon

- Check-in to be posted soon

Lecture Outline

- **Threads**
- Data Sharing & Mutex

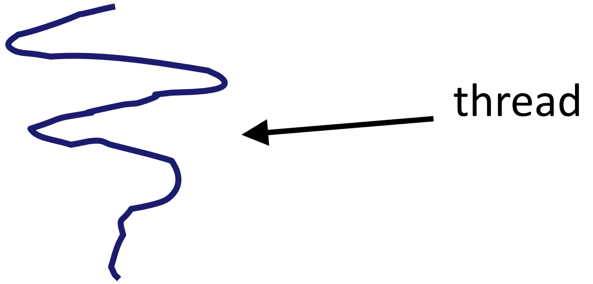
Recall: past poll

- 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;
}
```

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

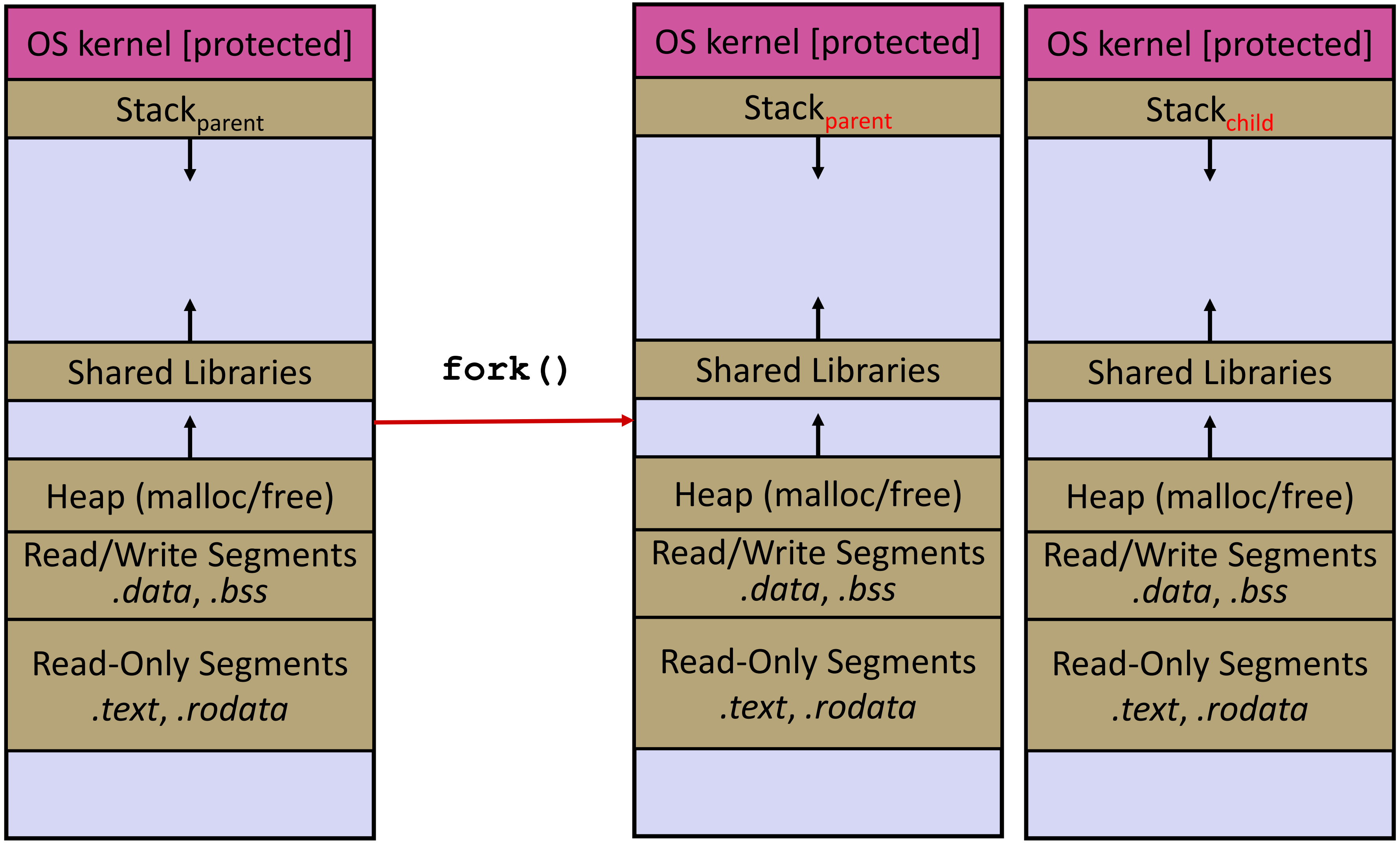


- In most modern OS's:
  - Threads are the *unit of scheduling.*

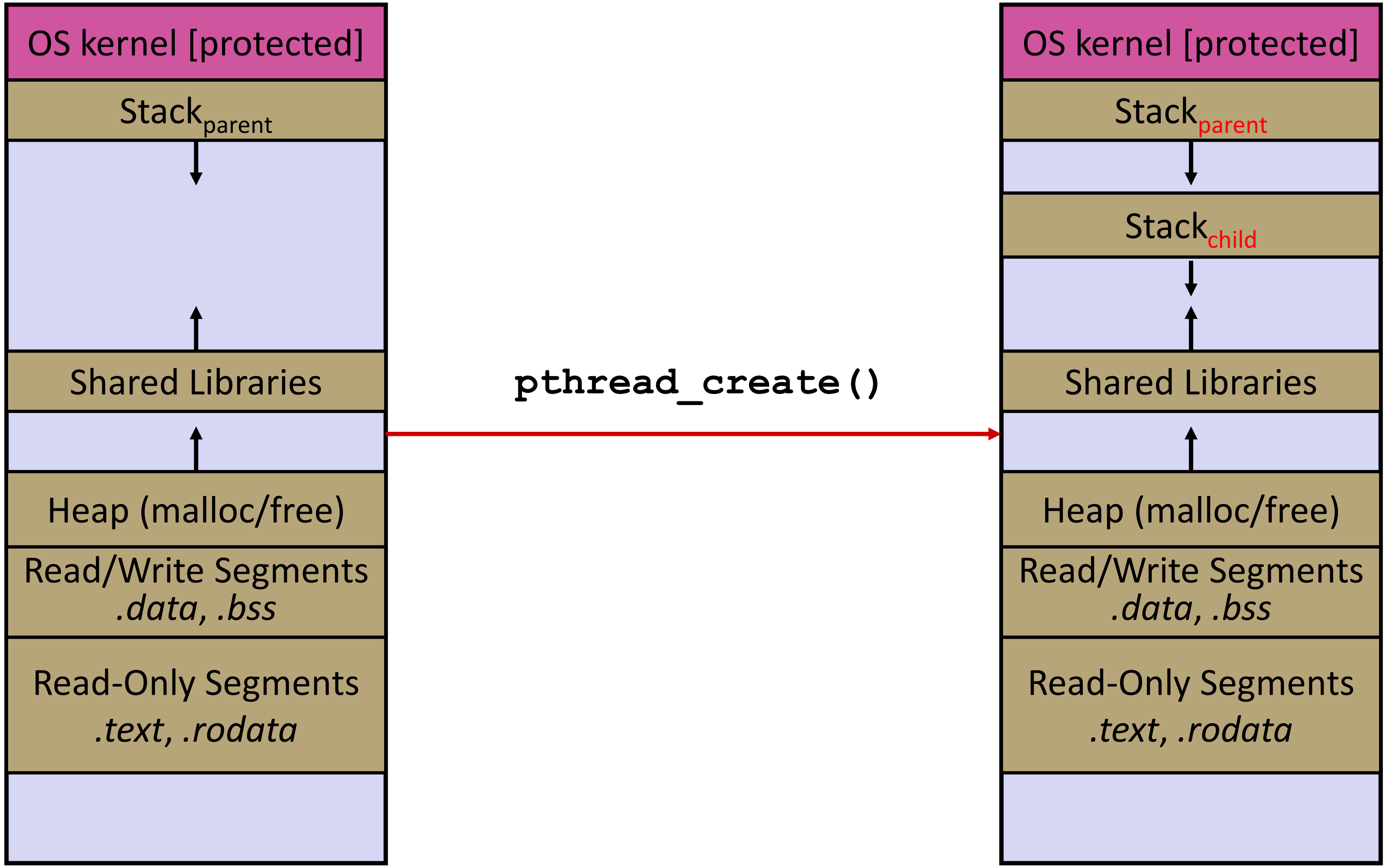
Threads vs. Processes

- In most modern OS's:
  - A <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

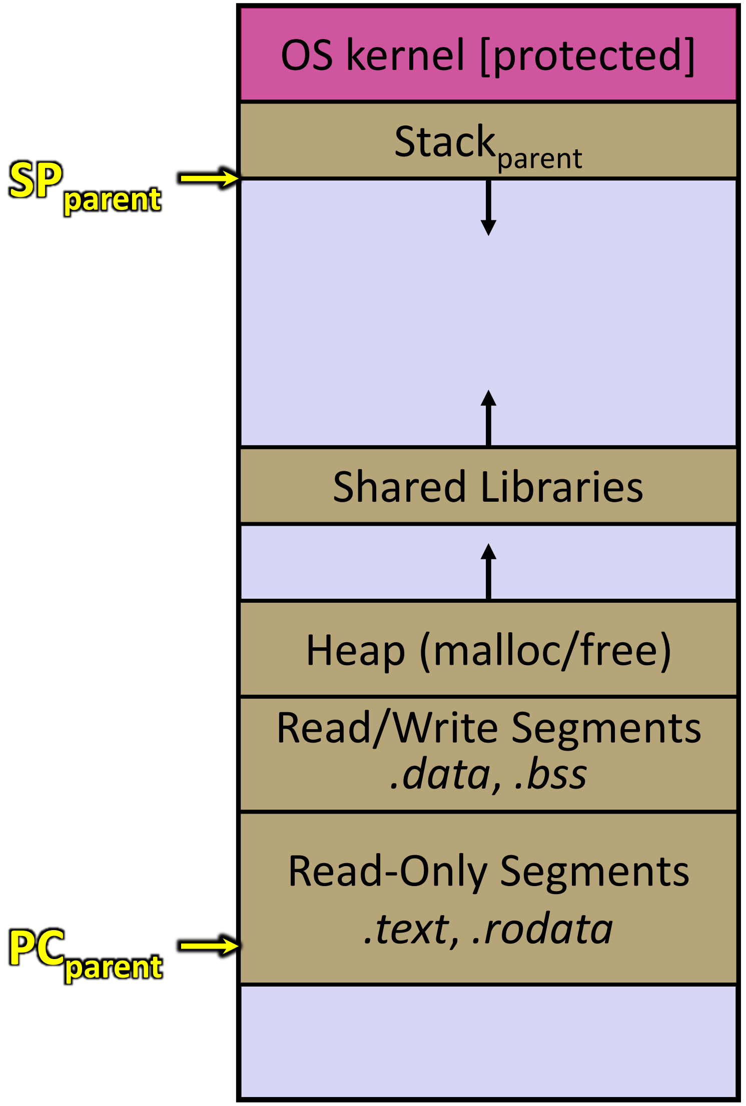


Threads

- 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

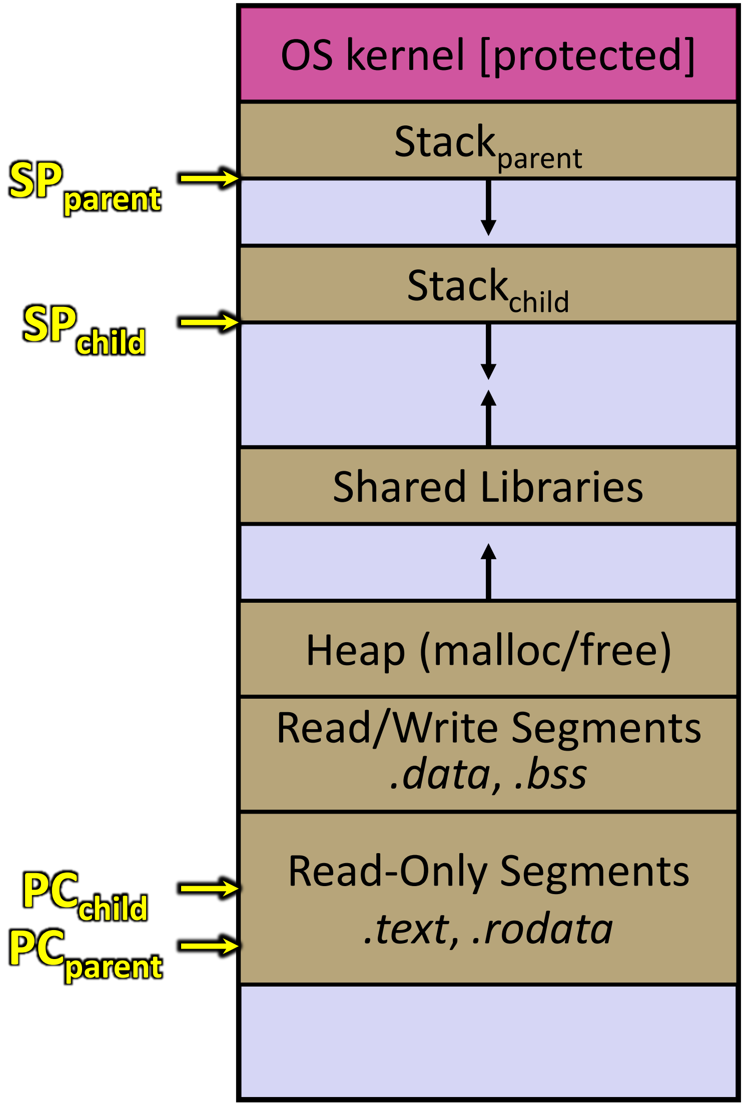
- Analogy: restaurant kitchen
  - Kitchen is process
  - Chefs are threads

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

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);
```

Function pointer!
Takes & returns void*
to allow "generics" in C

Argument for the thread function

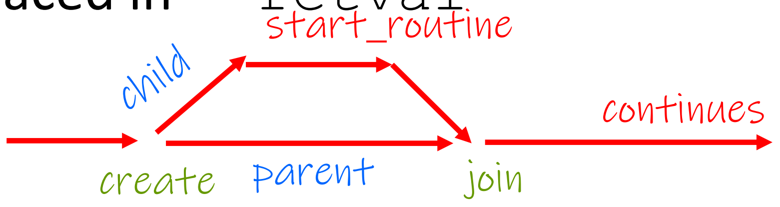
- Creates a new thread into `*thread`, with attributes `*attr` (`NULL` means default attributes)
- Returns **0** on success and an error number on error (can check against error constants)
- The new thread runs **`start_routine`**`(arg)`

pthread_create child start_routine parent continues

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



Poll Everywhere

pollev.com/tqm

- 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;
}
```

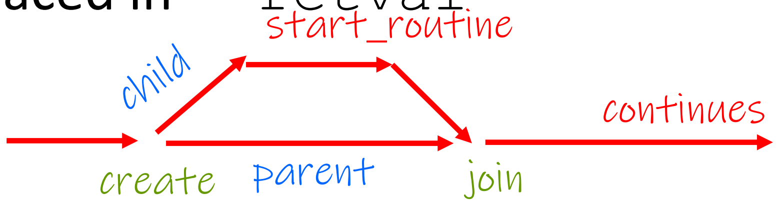
Thread Example

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

What To Do After Forking Threads?

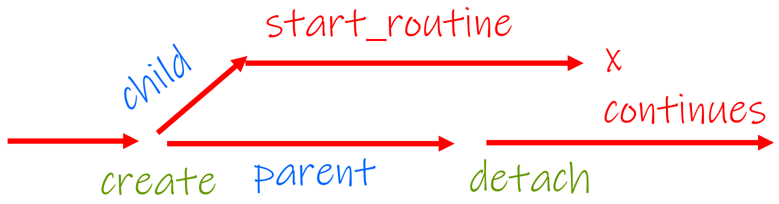
- `int pthread_join(pthread_t thread, void** retval);`
  - Waits for the thread specified by `thread` to terminate
  - The thread equivalent of `waitpid()`
  - The exit status of the terminated thread is placed in `**retval`

Parent thread waits for child thread to exit, gets the child's return value, and child thread is cleaned up



- `int pthread_detach(pthread_t thread);`
  - Mark thread specified by `thread` as detached – it will clean up its resources as soon as it terminates

Detach a thread.
Thread is cleaned up when it is finished



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()

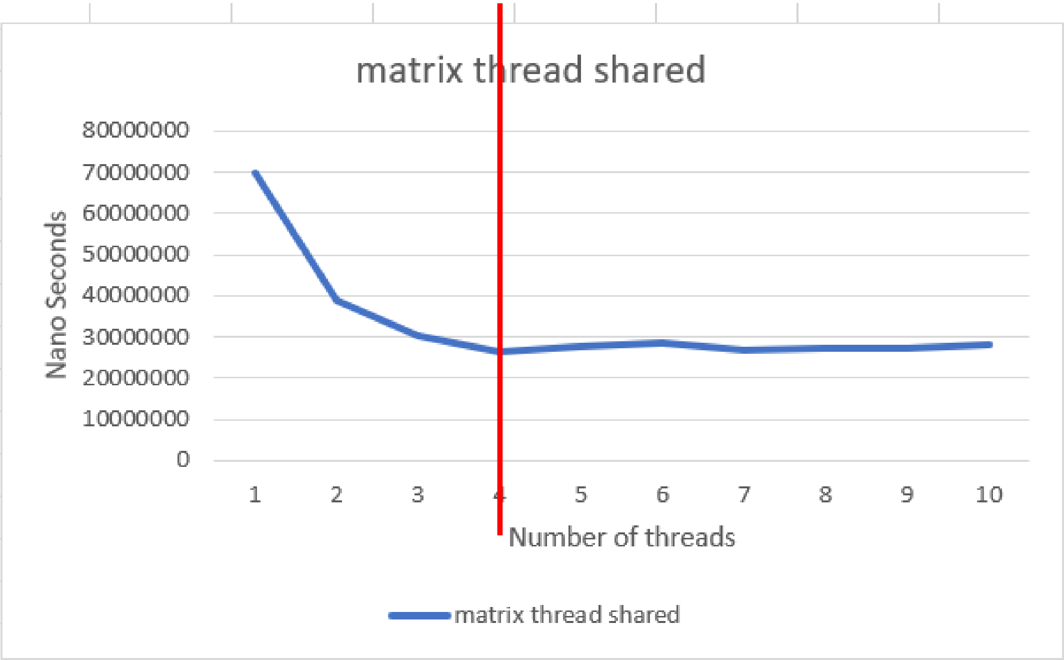
Parallelism

- 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**



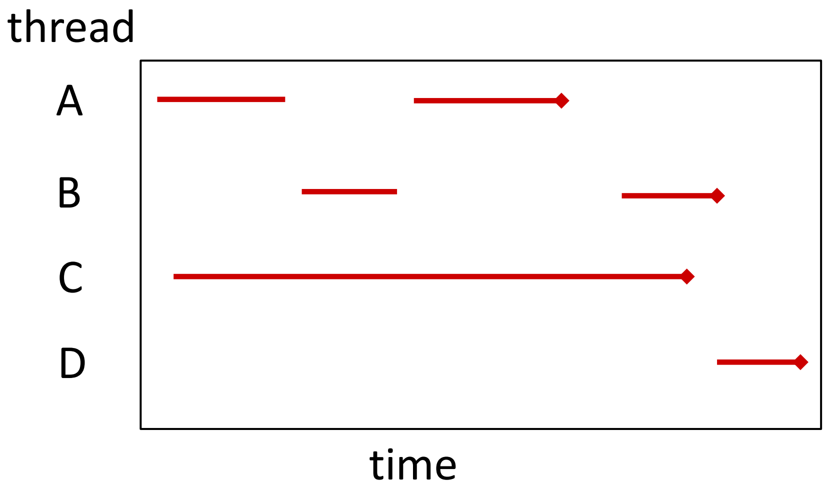
Diminishing returns

After 4 threads, no gain in speed

why? Machine run on only has 4 cores

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

Poll Everywhere

pollev.com/tqm

- 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 output possible?

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

Visualization

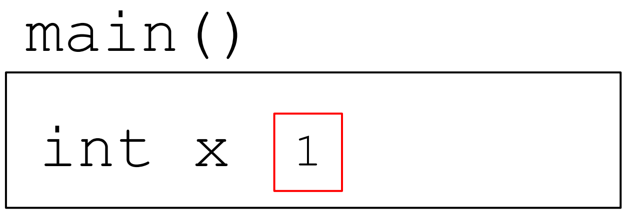
```
int main() {
  int x = 1;
  pthread_create(...);
  x = 2;
  pthread_create(...);

  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.




```
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);
}
```

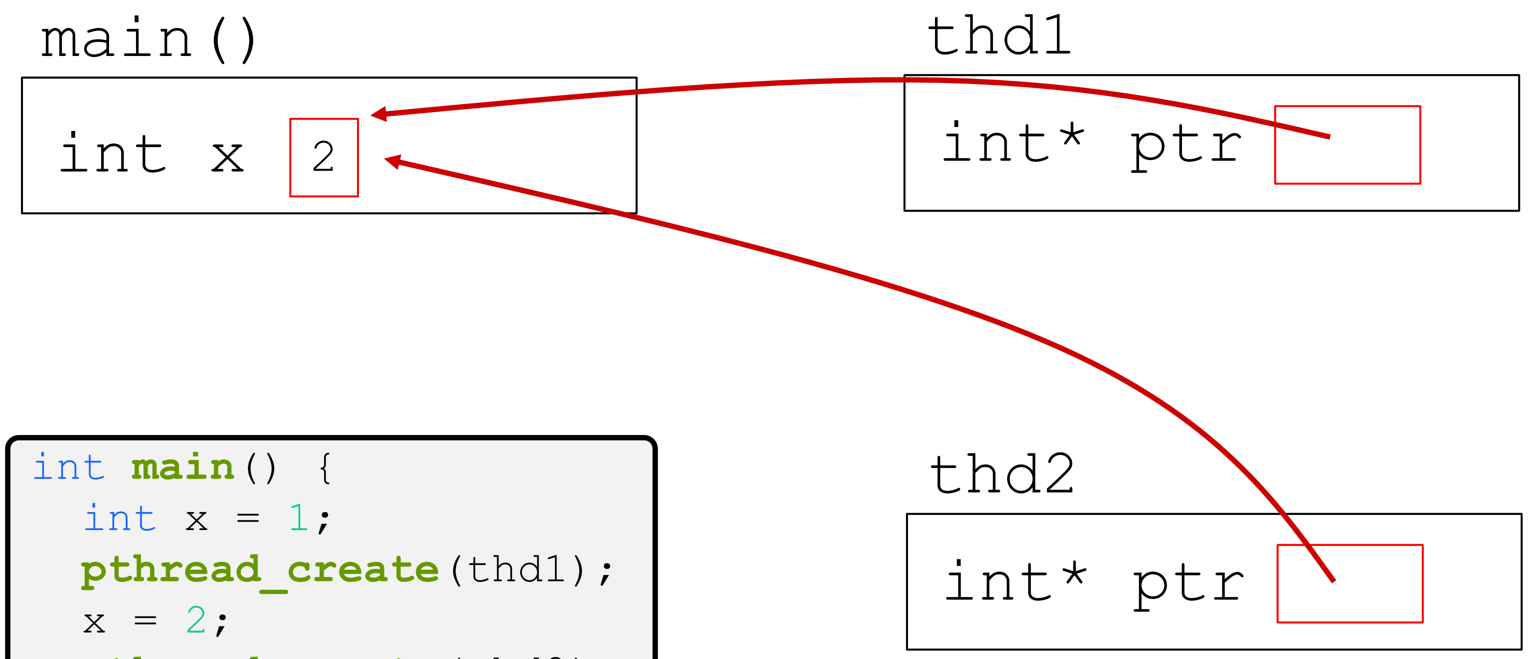
Visualization: Memory

- 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.




```c
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.

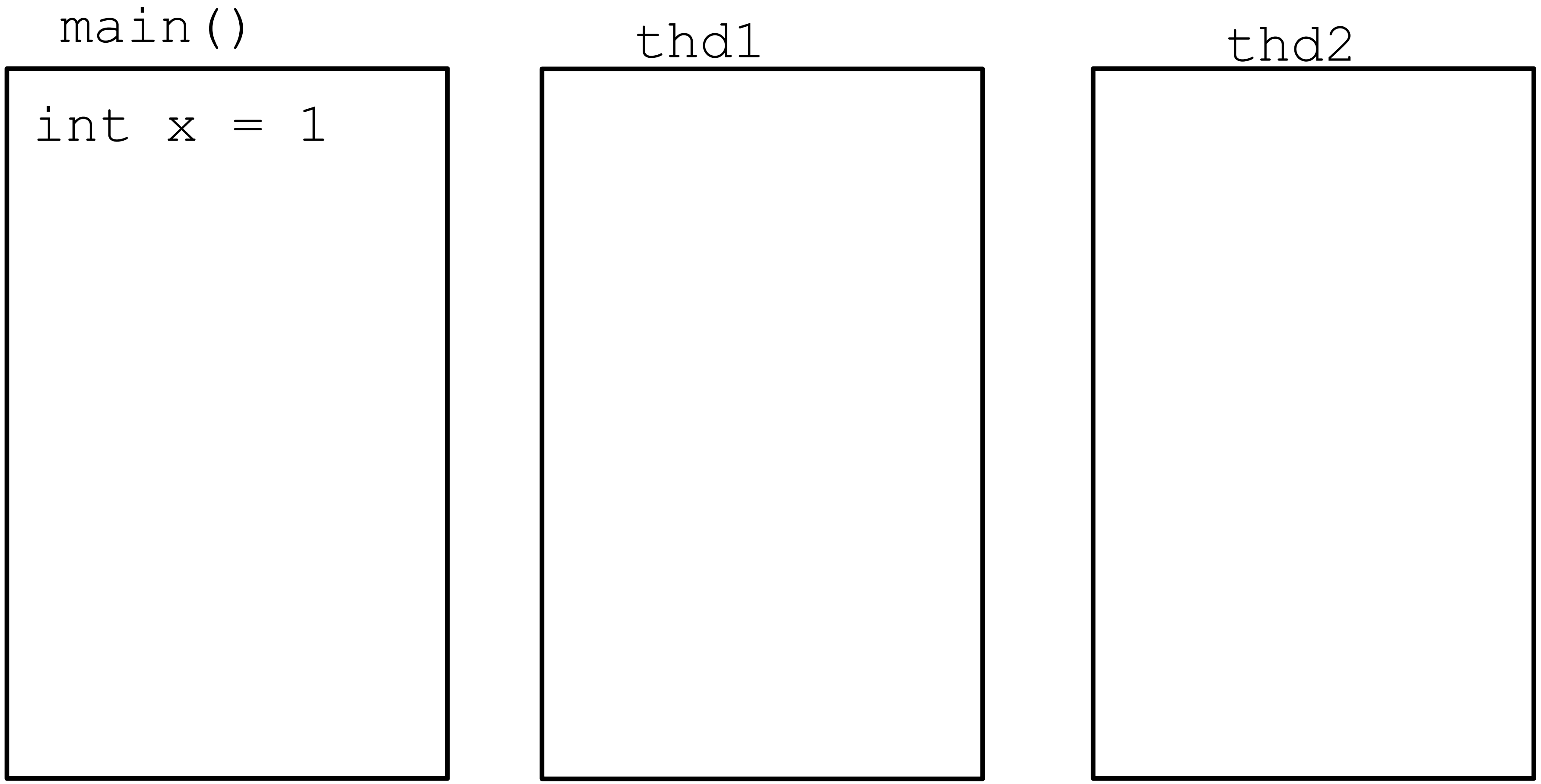
`main()`

```
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

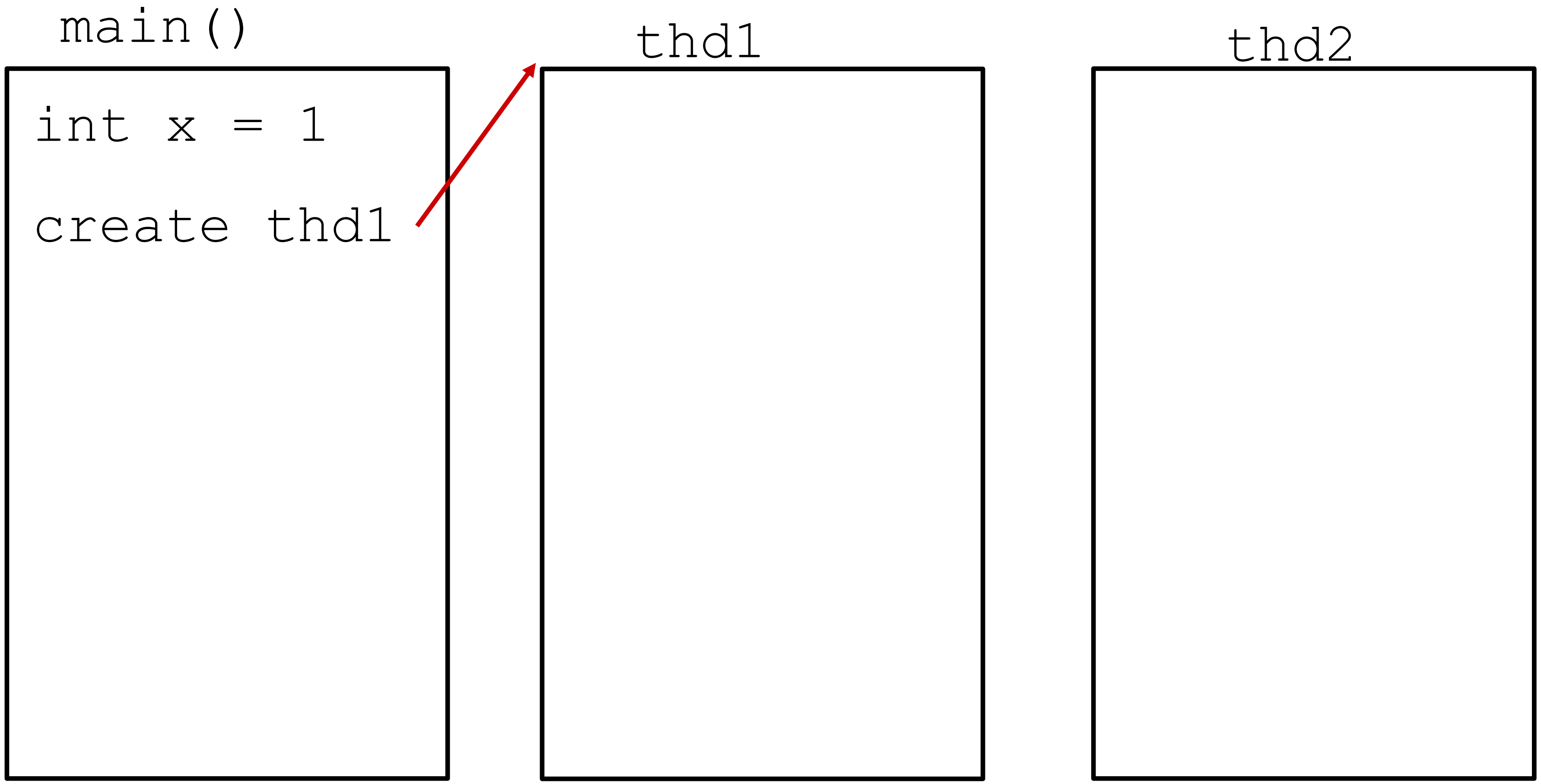
Visualization: Ordering

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



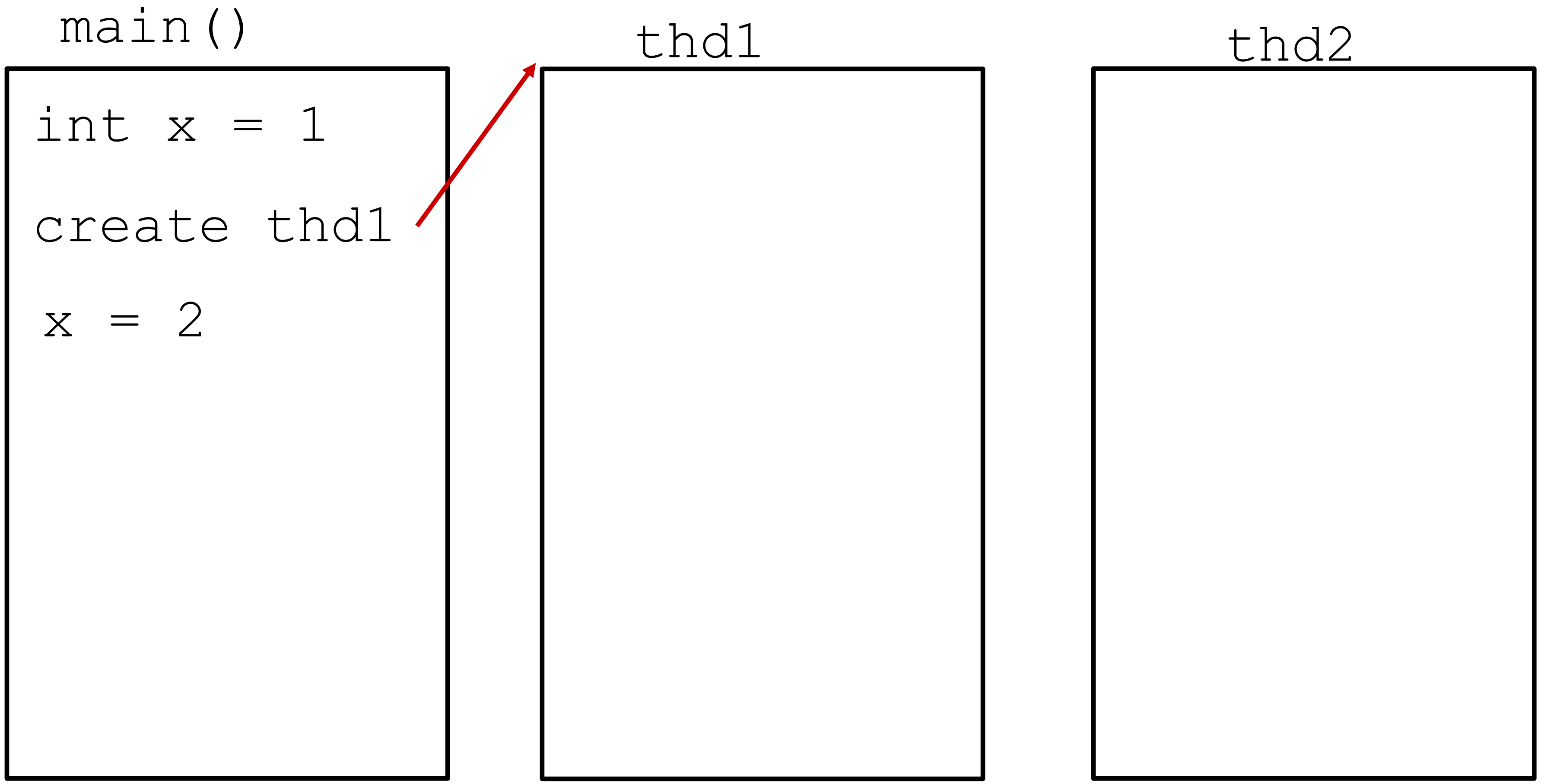
Visualization: Ordering

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



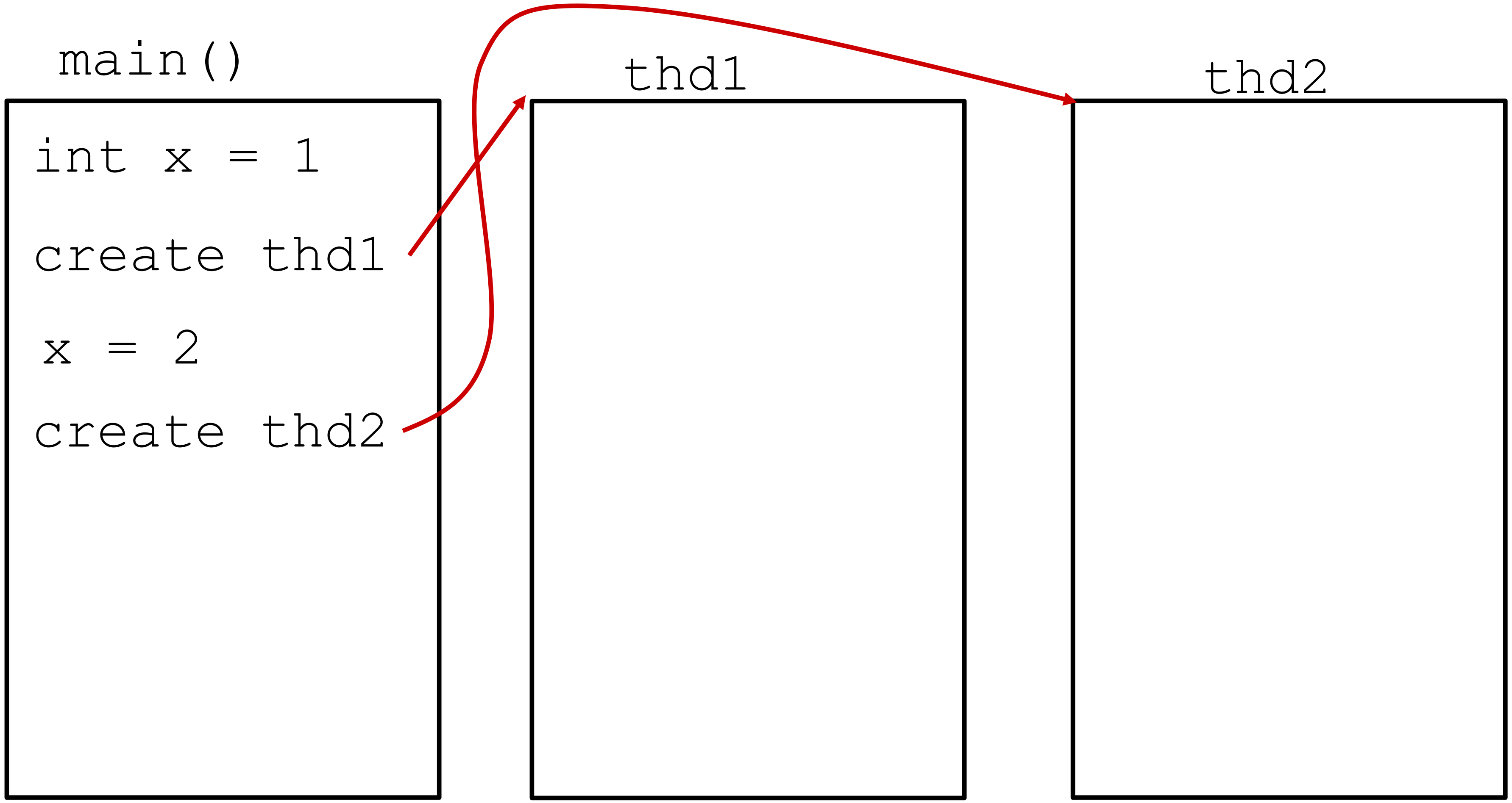
Visualization: Ordering

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



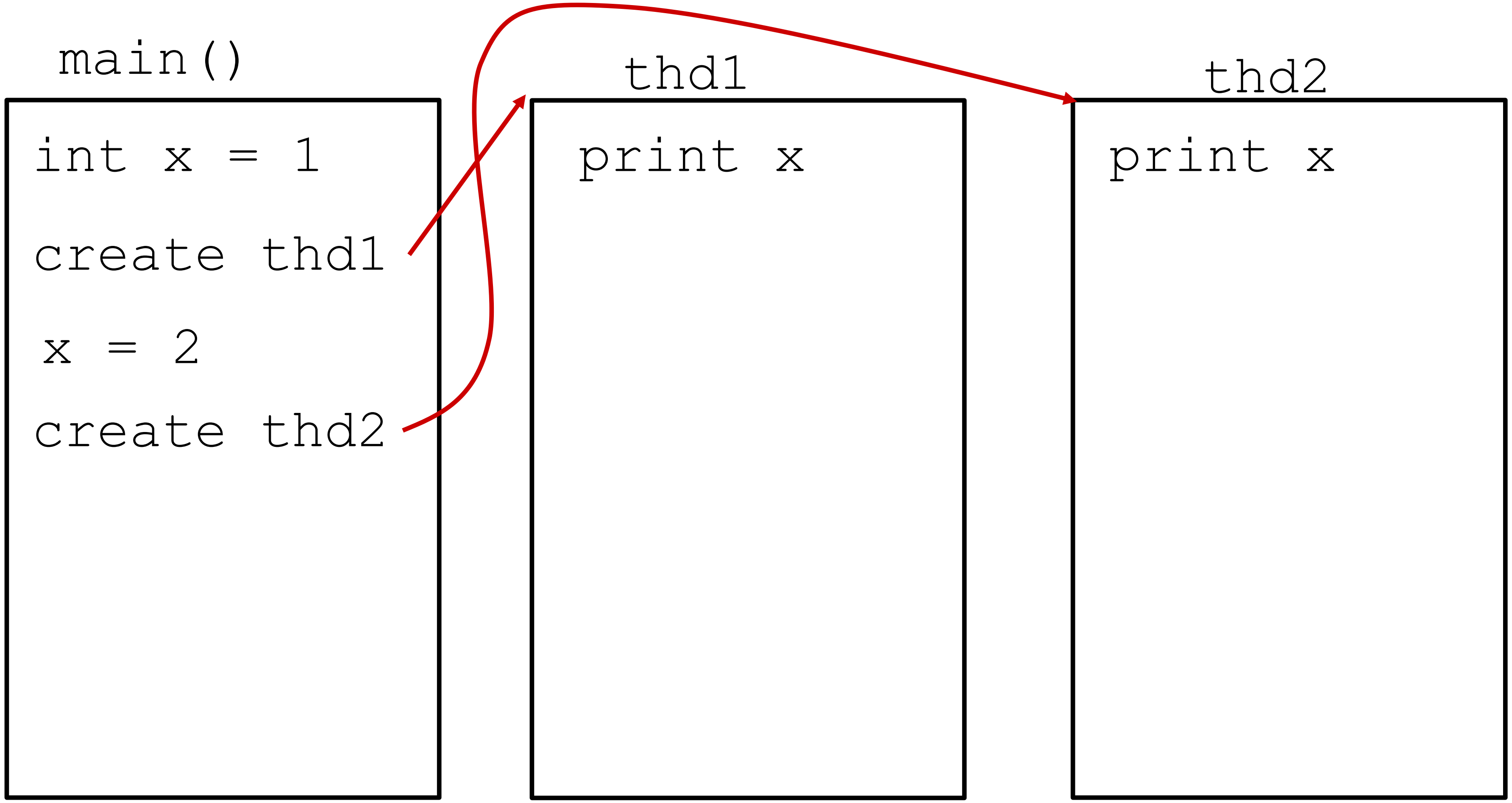
Visualization: Ordering

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



Visualization: Ordering

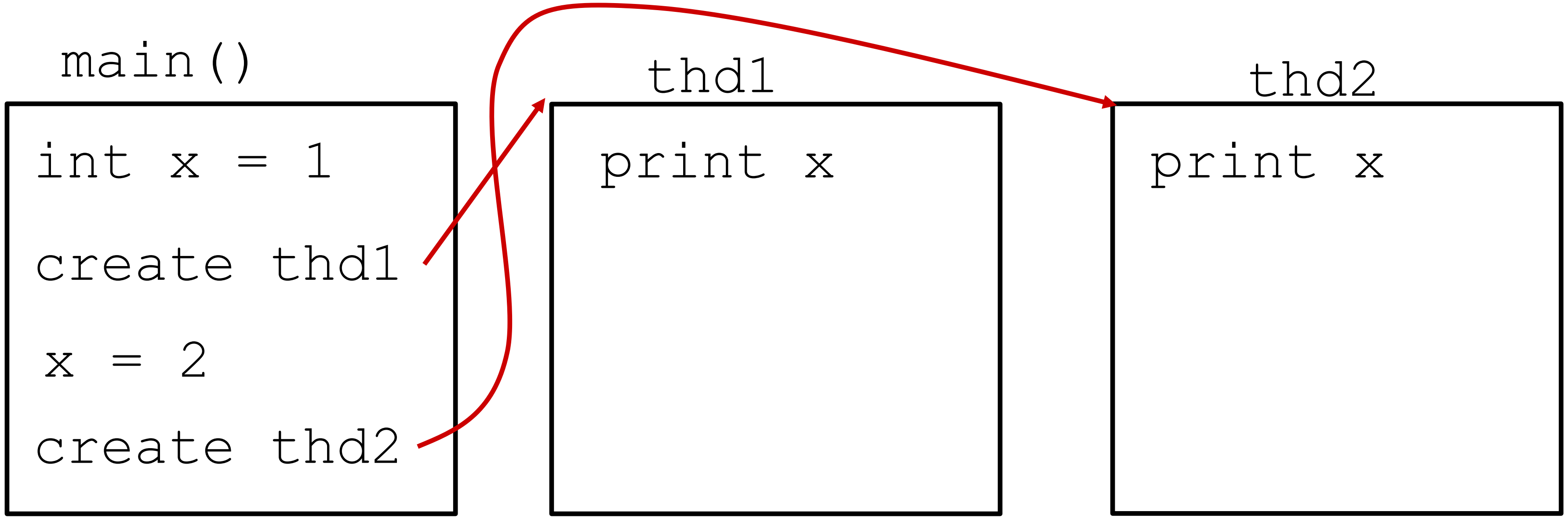
- Threads run concurrently; we can't be sure of the ordering of things across threads.



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.

Lecture Outline

- Threads
- **Data Sharing & Mutex**

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:

Poll Everywhere

pollev.com/tqm

- 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…

Poll Everywhere

pollev.com/tqm

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

We can be interrupted between checking note and leaving note ☹

```
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…

*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 the data structure
- Example: 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"
- Example: 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?

```c
#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`

Milk Example – What is the Critical Section?

- What if we use a lock on the refrigerator?
  - Probably overkill – what if roommate wanted to get eggs?
- For performance reasons, only put what is necessary in the critical section
  - Only lock the milk
  - But lock *all* steps that must run uninterrupted (*i.e.* must run as an atomic unit)

```
fridge.lock()
if (!milk) {
  buy milk
}
fridge.unlock()
```

```
milk_lock.lock()
if (!milk) {
  buy milk
}
milk_lock.unlock()
```

Poll Everywhere

pollev.com/tqm

- There are at least 4 bad practices/mistakes done with locks in the following code. Find them.
  - Assume `g_lock` and `k_lock` have been initialized and will be cleaned up.
  - Assume that these functions will be called by multi-threaded code.

```
pthread_mutex_t g_lock, k_lock;
int g = 0, k = 0;

void fun1() {
  pthread_mutex_lock(&g_lock);
  g += 3;
  pthread_mutex_unlock(&g_lock);
  k++;
}

void fun2(int a, int b) {
  pthread_mutex_lock(&g_lock);
  g += a;
  pthread_mutex_unlock(&g_lock);
  pthread_mutex_lock(&k_lock);
  a += b;
  pthread_mutex_unlock(&k_lock);
}

void fun3() {
  int c;
  pthread_mutex_lock(&g_lock);
  cin >> c;  // have the user enter an int
  k += c;
  pthread_mutex_unlock(&g_lock);
}
```

That's all!

- Next time:
  - Deadlocks
  - Spinning
  - Condition variables!