About how long did Exercise 11 take you?

A. [0, 2) hours
B. [2, 4) hours
C. [4, 6) hours
D. [6, 8) hours
E. 8+ Hours
F. I didn’t submit / I prefer not to say
Concurrency: Threads
CSE 333 Winter 2023

Instructor: Justin Hsia

Teaching Assistants:
Adina Tung
James Froelich
Noa Ferman
Saket Gollapudi
Timmy Yang
Zhuochun Liu

Danny Agustinus
Lahari Nidadavolu
Patrick Ho
Sara Deutscher
Wei Wu

Edward Zhang
Mitchell Levy
Paul Han
Tim Mandzyuk
Yiqing Wang
Relevant Course Information

❖ Exercise 12 released today, due Wednesday (3/8)
   ▪ Concurrency via pthreads

❖ Homework 4 due next Thursday (3/9)
   ▪ Submissions accepted until Sunday (3/12)

❖ Please fill out the course evaluations for lecture and your section next week!

❖ Final will be open from Monday through Wednesday of Finals Week (3/13-15)
   ▪ Same policies as the midterm
   ▪ ex8-ex12, hw3-hw4, overall course questions
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
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 (cf., Java)
  - To enable support for multithreading, must include `-pthread`
    flag when compiling and linking with `gcc` command
    - `gcc -g -Wall -std=c17 -pthread -o main main.c`
Creating and Terminating Threads

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

  - 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)`

- **void pthread_exit(void* retval);**

  - Equivalent of `exit(retval);` for a thread instead of a process
  - The thread will automatically exit once it returns from `start_routine()`
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

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

- A single *process* handles all of the connections, but a parent *thread* dispatches (creates) a new thread to handle each connection
  - The child thread handles the new connection and then exits when the connection terminates

- See `searchserver_threads/` for code if curious
Multithreaded Server

client

connect

accept()
Multithreaded Server

client

server

pthread_create()

pthread_detach()
Multithreaded Server

client

accept()

client

server
Multithreaded Server

```c
pthread_create()
```
Multithreaded Server

client -> server

shared data structures

client

client

client

client

client
Thread Examples

❖ See `ctthreads.c`
  ▪ How do you properly handle memory management?
    • Who allocates and deallocates memory?
    • How long do you want memory to stick around?

❖ See `pthreads.cc`
  ▪ More instructions per thread = higher likelihood of interleaving

❖ See `searchserver_threads/searchserver.cc`
  ▪ When calling `pthread_create()`, `start_routine` points
to a function that takes only one argument (a `void*`)
    • To pass complex arguments into the thread, create a struct to bundle
      the necessary data
Why Concurrent Threads? (Review)

❖ Advantages:
   ▪ Almost as simple to code as sequential
     - In fact, most of the code is identical! (but a bit more complicated to dispatch a thread)
   ▪ Concurrent execution with good CPU and network utilization
     - Some overhead, but less than processes
   ▪ Shared-memory communication is possible

❖ Disadvantages:
   ▪ Synchronization is complicated
   ▪ Shared fate within a process
     - One “rogue” thread can hurt you badly
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?)
Data Race Example

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

- If you live alone:

- If you live with a roommate:

```java
if (!milk) {
    buy milk
}
```
Does leaving a note on the fridge fix our milk data race problem?

A. Yes, problem fixed
B. No, could end up with no milk
C. No, could still buy multiple milk
D. We’re lost...

```java
if (!note) {
    if (!milk) {
        leave note
        buy milk
        remove note
    }
}
```
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 or intermediate (i.e., invalid) 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! 😱
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 (see CSE 451)

- **Goals of synchronization:**
  - **Liveness** – ability to execute in a timely manner (informally, “something good 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:

```java
// non-critical code
lock.acquire();
// critical section
lock.release();
// non-critical code
```
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 \textit{all} steps that must run uninterrupted (\textit{i.e.}, must run as an atomic unit)

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

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

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

❖ `pthread_mutex_init`()
  - Initializes a mutex with specified attributes

❖ `pthread_mutex_lock`()
  - Acquire the lock – blocks if already locked

❖ `pthread_mutex_unlock`()
  - Releases the lock

❖ `pthread_mutex_destroy`()
  - "Uninitializes" a mutex – clean up when done

```c
int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* attr);
int pthread_mutex_lock(pthread_mutex_t* mutex);
int pthread_mutex_unlock(pthread_mutex_t* mutex);
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```
pthread Mutex Examples

❖ See total.cc
  ▪ Data race between threads

❖ See total_locking.cc
  ▪ Adding a mutex fixes our data race

❖ How does this compare to sequential code?
  ▪ Likely slower – only 1 thread can increment at a time, but have to 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
Your Turn! (pthread mutex)

- Rewrite `thread_main` from `total_locking.cc`:
  - It need to be passed an `int*` with the `address` of `sum_total` and an `int` with the number of times to loop (in that order)
  - Increment a local sum variable `NUM` times, then add it to `sum_total`
  - Handle synchronization properly!
C++11 Threads

- C++11 added threads and concurrency to its libraries
  - `<thread>` – thread objects
  - `<mutex>` – locks to handle critical sections
  - `<condition_variable>` – used to block objects until notified to resume
  - `<atomic>` – indivisible, atomic operations
  - `<future>` – asynchronous access to data
  - These might be built on top of `<pthread.h>`, but also might not be

- Definitely use in C++11 code if local conventions allow, but pthreads will be around for a long, long time
  - Use pthreads in current exercise