Concurrency: Threads
CSE 333 Spring 2019

Instructor: Justin Hsia

Teaching Assistants:
Aaron Johnston  Andrew Hu  Daniel Snitkovskiy
Forrest Timour  Kevin Bi  Kory Watson
Pat Kosakanchit  Renshu Gu  Tarkan Al-Kazily
Travis McGaha
Administrivia

- Exercise 17 released today, due Wednesday (6/5)
  - Concurrency via pthreads

- hw4 due Thursday (6/6)
  - Submissions accepted until Friday (6/7)

- Final is Wednesday (6/12), 12:30-2:20 pm, ARC 147
  - Review Session: Sunday (6/9), 4-6:30 pm, ECE 125
  - Two double-sided, handwritten sheets of notes allowed
  - Topic list and past finals on Exams page on website

- Please fill out the course evaluations for lecture and your section!
Creating and Terminating Threads

- Creates a new thread, whose identifier is placed in *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)

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

```c
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?

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

- \textbf{int pthread_detach}(pthread_t thread);
  - Mark thread specified by \texttt{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
Multithreaded Server

client

server

pthread_create()

pthread_detach()
Multithreaded Server
Multithreaded Server

client

pthread_create()

server
Multithreaded Server
Thread Examples

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

- See pthread.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?

**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:
Data Race Example

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

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

```c
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 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”)

"Let me go first, then you can go"
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 all steps that must run uninterrupted (i.e. must run as an atomic unit)

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

```java
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`

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

  ```c
  int pthread_mutex_lock(pthread_mutex_t* mutex);
  ```
  - Acquire the lock – blocks if already locked

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

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