Concurrency and Threads
CSE 333 Spring 2018

Instructor: Justin Hsia

Teaching Assistants:

Danny Allen    Dennis Shao    Eddie Huang
Kevin Bi       Jack Xu       Matthew Neldam
Michael Poulain Renshu Gu    Robby Marver
Waylon Huang    Wei Lin          
Administrivia

- Exercise 17 released yesterday, due Wednesday (5/30)
  - Concurrency via pthreads

- hw4 due next Thursday (5/31)
  - Submissions accepted until Sunday (6/3)

- Final is Tuesday (6/5), 12:30-2:20 pm, KNE 120
  - Review Session: Sunday (6/3), 4-6:30 pm, EEB 125
  - Two double-sided, handwritten sheets of notes allowed
  - Topic list and past finals on Exams page on website
Some Common hw4 Bugs

- Your server works, but is really, really slow
  - Check the 2\textsuperscript{nd} argument to the \texttt{QueryProcessor} constructor

- Funny things happen after the first request
  - Make sure you’re not destroying the \texttt{HTTPConnection} object too early (\textit{e.g.} falling out of scope in a while loop)

- Server crashes on a blank request
  - Make sure that you handle the case that \texttt{read()} (or \texttt{WrappedRead()} ) returns 0
Review

Servers should be concurrent

- Sequential query processing has terrible performance, as client interactions block for arbitrarily long periods of time
- Different ways to process multiple queries simultaneously:
  - Issue multiple I/O requests simultaneously
  - Overlap the I/O of one request with computation of another
  - Utilize multiple CPUs or cores
Outline

- searchserver
  - Sequential
  - Concurrent via dispatching threads – `pthread_create()`
  - Concurrent via forking processes – `fork()`
  - Concurrent via non-blocking, event-driven I/O – `select()`
    - We won’t get to this 😞

- Reference: *CSPP*, Chapter 12
Sequential

- Pseudocode:

```plaintext
listen_fd = Listen(port);

while (1) {
    client_fd = accept(listen_fd);
    buf = read(client_fd);
    resp = ProcessQuery(buf);
    write(client_fd, resp);
    close(client_fd);
}
```

- See `searchserver_sequential/`
Why Sequential?

- Advantages:
  - Super simple to build/write

- Disadvantages:
  - Incredibly poor performance
    - One slow client will cause all others to block
    - Poor utilization of resources (CPU, network, disk)
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
    - Each thread has its own stack
Threads and Address Spaces

- Before creating a thread
  - One thread of execution running in the address space
  - That main thread invokes a function to create a new thread
    - Typically `pthread_create()`
Threads and Address Spaces

- After creating a thread
  - Two threads of execution running in the address space
    - Extra stack created
    - Child thread maintains separate values for its SP and PC
  - Both threads share the other segments
    - They can cooperatively modify shared data
**pthreads Threads**

- **int pthread_create(***thread, attr, start_routine, arg**);
  - Creates a new thread, whose identifier is placed in `thread`, with attributes `attr`. The new thread runs `start_routine(arg)`.

- **int pthread_detach(thread);**
  - Detach child from parent.

- **int pthread_join(thread, retval**);
  - Parent waits for child to finish and then receives its return value and cleans up.

Fork-join model:
- **create**
- **join**
- **child**
- **parent**
Thread Example

- See thread_example.cc
  - Remember *process graphs*? They work for threads, too!
Concurrency with Threads

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

client

connect

accept()

server
Multithreaded Server

client

server

pthread_create()
Multithreaded Server
Multithreaded Server

client

pthread_create()

server

client
Multithreaded Server

- Client
- Server
- Shared data structures
Concurrent Via Threads

- See searchserver_threads/

Notes:

- When calling `pthread_create()`, `start_routine` points to a function that takes only one argument (a `void*`)
  - To pass into the thread, create a struct to bundle the necessary data
- How do you properly handle memory management?
  - Who allocates and deallocates memory?
  - How long do you want memory to stick around?
Why Concurrent Threads?

- **Advantages:**
  - Code is still straightforward
    - Can write threaded code like sequential, but be careful with dispatch
  - 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
Threads and Data Races

- What happens if two threads try to mutate the same data structure?
  - They might interfere in painful, non-obvious ways, depending on the specifics of the data structure

- 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! ☠️
Data Race Example

- If your fridge has no milk, then go out and buy some more

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

- If you live alone:
- If you live with a roommate:
Data Race Example

- Idea: leave a note!
  - Does this fix the problem?
  - Vote at http://PollEv.com/justinh

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
    }
}
```

One possible scenario:

- You
  - check note
  - check milk
  - leave note
  - buy milk

- Roommate
  - check note
  - check milk
  - leave note
  - buy milk

Time
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 (CSE451)

- **Goals of synchronization:**
  - **Liveness** – ability to execute in a timely manner
  - **Safety** – avoid unintended interactions with shared data structures
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
```
Data Race Example With Locks

- 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

```java
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”)
  - pthreads (#include `<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

- `int pthread_mutex_unlock(pthread_mutex_t* mutex);`
  - Releases the lock
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, but pthreads will be around for a long, long time
  - Use pthreads in Exercise 17