Threads: Shared Data

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevin Bernat	Jialin Cai
Mati Davis	Donglun He
Chandravaran Kunjeti	Heyi Liu
Shufan Liu	Eddy Yang

Upcoming Due Dates

- HW2 (Threads)
  - To be released tonight!
  - Should have everything you need now
  - Recitation will help ☺
  - Wednesday lecture may help with thinking about threads, but not strictly required

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")

These are VERY related

First concern we will be looking at with locks

Atomicity

- Atomicity: An operation or set of operations on some data are *atomic* if the operation(s) are indivisible, that no other operation(s) on that same data can interrupt/interfere.

- Aside on terminology:
  - Often interchangeable with the term “Linearizability”
  - Atomic has a different (but similar-ish) meaning in the context of data bases and ACID.

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();    // loop/idle if locked
// critical section
lock.release();

// non-critical code
```

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.cc`
  - Data race between threads
- See `total_locking.cc`
  - 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.cc`

Lecture Outline

- Locks & mutexes
- **Liveness & deadlocks**
- Condition Variables

Liveness

- Liveness: A set of properties that ensure that threads execute in a timely manner, despite any contention on shared resources.

- When `pthread_mutex_lock();` is called, the calling thread blocks (stops executing) until it can acquire the lock.
  - What happens if the thread can never acquire the lock?

Liveness Failure: Releasing locks

- If locks are not released by a thread, then other threads cannot acquire that lock
- See `release_locks.cc`
  - Example where locks are not released once critical section is completed.

Liveness Failure: Deadlocks

- Consider the case where there are two threads and two locks
  - Thread 1 acquires lock1
  - Thread 2 acquires lock2
  - Thread 1 attempts to acquire lock2 and blocks
  - Thread 2 attempts to acquire lock1 and blocks

Neither thread can make progress ☹

- See `milk_deadlock.cc`

- Note: there are many algorithms for detecting/preventing deadlocks

Liveness Failure: Mutex Recursion

- What happens if a thread tries to re-acquire a lock that it has already acquired?

- See `recursive_deadlock.cc`

- By default, a mutex is not re-entrant.
  - The thread won't recognize it already has the lock, and block until the lock is released

Aside: Recursive Locks

- Mutex's can be configured so that you it can be re-locked if the thread already has locked it. These locks are called *recursive locks* (sometimes called *re-entrant locks*).

- Acquiring a lock that is already held will succeed
- To release a lock, it must be released the same number of times it was acquired

- Has its uses, but generally discouraged.

Lecture Outline

- Locks & mutexes
- Liveness & deadlocks
- **Condition Variables**

Aside: sleep()

- `unistd.h` defines the function:

```
unsigned int sleep(unsigned int seconds);
```

  - Makes the calling thread sleep for the specified number of seconds, resuming execution afterwards

- Useful for manipulating scheduling for testing and demonstration purposes
  - Also for asynchronous/non-blocking I/O, but not covered in this course.

- Necessary for HW2 so that auto-graders work ☹

Thread Communication

- Sometimes threads may need to communicate with each other to know when they can perform operations

- Example: Producer and consumer threads
  - One thread creates tasks/data
  - One thread consumes the produced tasks/data to perform some operation
  - The consumer thread can only produce things once the producer has produced them

Naïve Solution

- Consider the example where a thread must wait to be notified before it can print something out and terminate

- Possible solution: “Spinning”
  - Infinitely loop until the producer thread notifies that the consumer thread can print

- See `spinning.cc`

- Alternative: Condition variables

Condition Variables

- Variables that allow for a thread to wait until they are notified to resume

- Avoids waiting clock cycles "spinning"

- Done in the context of mutual exclusion
  - a thread must already have a lock, which it will temporarily release while waiting
  - Once notified, the thread will re-acquire a lock and resume execution

pthreads and condition variables

- `pthread.h` defines datatype `pthread_cond_t`

- ```
  int pthread_cond_init(pthread_cond_t* cond,
                        const pthread_condattr_t* attr);
  ```
  - Initializes a condition variable with specified attributes

- ```
  int pthread_cond_destroy(pthread_cond_t* cond);
  ```
  - "Uninitializes" a condition variable – clean up when done

pthreads and condition variables

- `pthread.h` defines datatype `pthread_cond_t`

- ```
  int pthread_cond_wait(pthread_cond_t* cond,
                        pthread_mutex_t* mutex);
  ```
  - Atomically releases the mutex and blocks on the condition variable. Once unblocked (by one of the functions below), function will return and calling thread will have the mutex locked

- ```
  int pthread_cond_signal(pthread_cond_t* cond);
  ```
  - Unblock at least one of the threads on the specified condition

- ```
  int pthread_cond_broadcast(pthread_cond_t* cond);
  ```
  - Unblock all threads blocked on the specified condition

- See `cond.cc`
  ```

Aside: Things left out

- MANY things left out of this lecture
- Synchronization methods:
  - Semaphores
  - Monitors
- Concurrency properties
  - ACID (databases)
  - CAP theorem
- A lot more concurrency stuff covered in CIS 5050 ☺