CSE 374: Lecture 27

(Week 10!) Concurrency and memory cont.

Previously, on Threading

```
Pthread_t threadID;
```

The threadID keeps track of to which thread we are referring.

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

Note - pthread_create takes two generic (untyped) pointers
interprets the first as a function pointer and the second as an argument pointer.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Puts calling thread ‘on hold’ until *‘thread’* completes - useful for waiting to thread to exit

Memory Consideration

(ex. pthreadex.c)

- If one thread did nothing of interest to any other thread, why bother running?
- Threads must communicate and coordinate
  - Use results from other threads, and coordinate access to shared resources
- Simplest ways to not mess each other up:
  - Don't access same memory (complete isolation)
  - Don't write to shared memory (write isolation)
- Next simplest:
  - One thread doesn't run until/unless another is done

Parallel Processing

Common pattern for expensive computations (such as data processing)

1. split the work up, give each piece to a thread (fork)
2. wait until all are done, then combine answers (join)

To avoid bottlenecks, each thread should have about the same amount of work

Performance will always be less than perfect speedup

What about when all threads need access to the same mutable memory?

Multiple threads with one memory

Often you have a bunch of threads running at once and they might need the same mutable (writable) memory at the same time but probably not

    Want to be correct, but not sacrifice parallelism

Example: bunch of threads processing bank transactions

    withdraw, deposit, transfer, currentBalance, etc…

unlikely two will overlap, but there’s a chance

very important that answer is correct when they overlap

Data races

```
struct Acct {int balance; /*etc…*/ };
int withdraw(struct Acct* a, int amt) {
    if (a->balance < amt) return FAIL;
    a->balance -= amt; return SUCCESS;
}
```

This code is correct in a sequential program

It may have a race condition in a concurrent program, allowing for a negative balance

Discovering this bug with testing is very hard

A Data Race - *two threads withdraw $100 simultaneously*

Thread 1

```
struct Acct {int balance; /*etc…*/ };
int withdraw(struct Acct* a, int amt) {
     if (a->balance < amt) {
          return FAIL; }








     a->balance -= amt; return SUCCESS;
}
```

Thread 2

```



struct Acct {int balance; /*etc…*/ };
int withdraw(struct Acct* a, int amt) {
     if (a->balance < amt) {
          return FAIL; }
     a->balance -= amt; return SUCCESS;
}
```

Atomic Operations

- An operation we want to be done all at once
    - No interruptions
- Note: Must be the right size
    - Too big - program runs sequentially
    - Too small - program has potential races
- ‘Atomic’ requires a hardware primitive

We can wrap the hardware primitive with a lock

In C : ‘mutex’

```
std::mutex BankAccount::m_;
void BankAccount::withdraw(double amount) {
     m_.lock();
     if (getBalance() > b) {
          throw std::invalid_argument();
     }
     setBalance(getBalance() - amount);
     m_.unlock();
}
```

Deadlocking

Every piece of code that refers to a datum calls the lock for that datum

If `foo` locks `D`, and then calls `bar` which also must lock `D`, we get a deadlock - we can't progress because `bar` can not complete

One solution is to write a helper function to replace `bar` - `lockedBar`

```
void BankAccount::withdraw(double amount) {
  m_.lock();
  if (getBalance() < amount) {
    throw std::invalid_argument();
  }
  setBalanceUnderLock(getBalance() - amount);
  m_.unlock();
}

void setBalance(double amount) {
  m_.lock();
  setBalanceUnderLock(amount);
  m_.unlock();
}

void setBalanceUnderLock(double amount) {
  balance_ = amount;
}
```

Deadlock

Problem:

If every method that modifies balance_ is locked with mutex m, that balance can not be updated.

Solution:

Must create helper function that allows for modifying balance_ under the lock.

```
void BankAccount::withdraw(double amount) {
  m_.lock();
  if (getBalance() < amount) {
    throw std::invalid_argument();
  }
  setBalanceUnderLock(getBalance() - amount);
  m_.unlock();
}

void setBalance(double amount) {
  m_.lock();
  setBalanceUnderLock(amount);
  m_.unlock();
}

void setBalanceUnderLock(double amount) {
  balance_ = amount;
}
```

C++ Lock Guards

- A “lock guard” is an object that
  - Locks the mutex in the constructor
  - Unlocks in the destructor
- If the lock guard is added to the stack it is locked upon creation
- Mutex is unlocked when object is removed from the stack; even correctly responding for an exception.

```
void deposit(double amount) {

    std::lock_guard<std::mutex> lock(m_);
   // locks mutex m_ in the lock_guard constructor
   // mutex is now locked

    setBalanceWithLock(getBalance() + amount);
     // When deposit() returns,
     // the stack-allocated lock_guard will be deleted,

    // calling the destructor and releasing the mutex.

  }
```

Another Deadlock

If we have an operation for two accounts

Must lock the value on each account.

But what happens if one transfer is started from account A to account B while a simultaneous transfer is started from account B to account A?

```
void transferTo(double amount, BankAccount& other) {
    m_.lock();
    other.m_.lock();

    setBalanceInternal(getBalance() - amount);
    other.setBalanceInternal(other.getBalance() + amount);

    other.m_.unlock();
    m_.unlock();
  }
```

Thread T1: A.transferTo(50, B);	Thread T2: B.transferTo(20, A);
m_.lock(); // Locks A's mutex	
	m_.lock(); // Lock's B's mutex
other.m_.lock(); // Waits for B's mutex	
	other.m_.lock() // Waits for A's mutex

Another Solution

- **Use smaller critical sections.** Lock A's mutex only around the modification of A's balance, and lock B's mutex when modifying B's balance.
    a. But - we expose an intermediate state in which A's account has been debited but the funds haven't been put in B's account yet - we've temporarily lost money, which isn't great.
- **Use larger critical sections.** Add a single lock for all bank accounts that must be acquired before doing multi-account transactions.
    a. But it means that we can only do one transaction at a time throughout the entire bank, even if the accounts aren't related to each other. This is a performance loss.
- **Always lock mutexes in a specific order.** We can choose to always lock the mutex of the account with the lower account id first, then lock the id of the higher account id. This works because account ids are unique and immutable, thus we can rely on them without synchronization.

C++ Atomic

Even single line integer operations (`++accountCount`) may be subject to race conditions.

Instead of manually locking and unlocking every integer operation, can make the data declaration `std::atomic`

Atomic renders that variable safe for read/write operations.

```
// In h:
static std::atomic<int> accountCount_;

// In cpp:
BankAccount::BankAccount() {
     accountId_ = ++accountCount_;
    balance_ = 0;
}
```

Other types of locks

There are other types of locks and primitives that are useful, besides the regular mutex, lock guard, and std::atomic:

- **Reentrant locks.** We had a problem earlier where one function that locked the mutex tried to call another function that would lock the same mutex, but this didn't work because the first function already had the lock! Use this behavior with a "reentrant lock": the same thread may re-lock the same lock any number of times. The lock will be released to a different thread once all of the lock() calls have been correspondingly unlock()'ed. Re-entrant locks can be difficult to trace.
- **Reader-writer locks.** All of the problems that we've seen so far have resulted by read/write or write/write combinations of calls. It is only the writing that causes problem. To improve efficiency, you might use "reader-writer locks": these allow multiple threads to read the same data at a time, but if any thread tries to write, it will make sure that no other thread is either reading or writing at the same time. This improves the performance of reads (allowing them to happen at once) while still maintaining correctness of the program.
- **Condition variables.** Let's say you are trying to dequeue from a queue, but there's no data in the queue at the moment. You want to wait until some other thread inserts into the queue, then you can wake up and dequeue that element! In this case you can use a "condition variable": a primitive that can be used to block a thread until another thread notifies the condition variable that the waiting condition has been satisfied.

Memory Guidelines

For every memory location, you should obey at least one of the following:

- Make it **thread-local**. Whenever possible, avoid sharing resources between threads - make a copy for each thread. If threads do not need to communicate with each other through the shared resource (for example, a random-number generator), then make it thread-local. In typical concurrent programs, the vast majority of objects should be thread-local.
  - Shared-memory should be rare - minimize it.
- Make it **immutable**. Whenever possible, do not update objects; make new objects instead. If a location is only read (never written), then no synchronization is necessary. Simultaneous reads are not data races, and not a problem.
  - In practice, programmers over-use mutation - minimize it.
- Make access **synchronized**, ie use locks and other primitives to prevent race conditions.

Synchronicity

1. **No data races.** Never allow two threads to read/write or write/write a location at the same time.
2. **Think of what operations need to be atomic.** Consider atomicity first, then figure out how to implement it with locks).
3. **Consistent locking.** For each location that should be synchronized, have a lock that is ALWAYS locked when reading or writing that location. The same lock may (and often should) be used to guard multiple locations/pieces of memory. Clearly document with comments the mutex that guards a particular piece of memory.
4. **Start with coarse-grained locking; move to finer-grained locking only if blocking for locks becomes an issue.** Coarse-grained locking is the practice of having fewer locks: one for the whole data structure, or one for all bank accounts. It is simpler to implement, but performance can be bad (fewer operations can be done at the same time). But if there isn't a lot of concurrent access, then coarse locking is probably fine. Fine-grained locking is the practice of having more locks, each guarding less data: one lock per data element, or one lock per field in the bank account. Fine-grained locking is trickier to get correct, requires more programming, and has more overhead (more locks to lock), but it we can do more things at once.
5. **Don't do expensive computations or I/O in critical sections, but also don't introduce race conditions.** This balances performance with correctness.
6. **Use built-in libraries whenever possible.** Concurrency is extremely tricky and difficult to get right; experts have spent countless hours building tools for you to use to make your code safe.

Very Brief Intro to GUIs

Gui Programming is multi-threaded Event-driven programming

- Initial threads - sets up the program
- Event dispatch thread (EDT) handles all GUI events
    - Mouse events
    - Keyboard events
    - Timer events, etc.
- Program registers callbacks (“listeners”)
- Worker threads
    - Function objects invoked in response to events

Ground Rules for GUIs

1. ALL GUI activity is on an event dispatch thread
    a. Violating this can cause safety errors (want to keep the GUI contained)
2. No other time-consuming activity is on the dispatch thread
    a. Blocking calls (like I/O) are absolutely forbidden
    b. This could cause ‘liveness’ errors

There are many GUI Frameworks

Open GL (old school graphics)

GTK (Gnome tool-kit - very common and portable)

Swing (Ubiquitous Java Framework)

React.js (New hotness)

- Most of these do some thread handling on their own; can obfuscate the concurrency structure of a program.

React.js

```
ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```

Javascript library

Creates a specific document object model with React objects

React renders components in a document

React Component API contains code that responds to events and manipulates data.

Course Topics

★ Linux, Shells & Emacs, Redirection
★ Shell variables and scripting
★ Regular expressions, grep & sed
★ C (parameters, scope, malloc, structs, lists, trees)
★ Stack & Heap structures, memory management
★ Debugging (gdb), Valgrind, Testing
★ Makefiles
★ Git
★ C++ (object oriented programming, classes, namespaces)
★ Threads and concurrency
★ Profiling and process review

Course Take-aways

- **Confidence**
  - You know enough to believe that you can figure out solutions
  - You know some basic commands to get started
  - You know some ideas to start searching
  - You know about ‘man’
  - You have some good resources (Pocket Guide/cplusplus.com)
  - You know about processes and how to link them
  - You’ve practice breaking down problems to come up with a tool chain
  - You’ve seen enough C & C++ to picture alternate computer languages
  - You’ve practice with memory to understand how the computer might work

When to use which language?

Please fill in your course reviews!