ECE 3574: Applied Software Design

Thread Synchronization

Today we are going to look at how to manage access to shared memory using a *mutex* and how to build higher-level abstractions, a *semaphore* and a *thread-safe* queue.

- Races and Atomics
- Mutex's and locking
- Condition Variables
- Semaphore
- Building a Semaphore using C++11
- QSemaphore

Recall the QSharedMemory class had a lock/unlock mechanism.

- Why was this needed?
- How is such a thing implemented?

The Why Question: Data races

Consider two threads that share an integer pointer, x with a loop (see race.cpp)

```
while(*x > 0){
  std::this_thread::sleep_for(std::chrono::nanoseconds(10
  *x -=1;
}
```

where the sleep is a stand-in for some compuational work.

- what is the value of `*x` after the threads execute?

We say the threads are *racing* to get to the value of `*x == 0`.

Synchronization is built on the idea of atomics

- An atomic operation is one that is guaranteed to not cause data races.
- Example `int` vs `atomic_int`

```
int x;
x = anothervar;
```

v/s

```
std::atomic_int x;
x.store(anothervar);
```

- The latter is compiled to instructions that lock the memory bus during the assignment.

How this works at the hardware level is complicated due to cache lines etc.

Atomics can be used directly or form the basic of a locking mechanism

- create an atomic boolean initialized to false
- to lock test if the value is false and if so set it to true (exchange), else try again (AKA *test and set*).
- access the locked resource
- to unlock, set the atomic bool back to false

See race_atomic.cpp

- Lucky for us, more high-level locking semantics are defined in the threading library

A *mutex* (MUTual EXclusion) is an object with exclusive ownership semantics.

It providess a synchronization primitive for protecting shared memory from simultaneous writes and/or reads.

- In the case of IPC the memory being protected is shared memory between processes.
- In the case of threads the memory being protected is the heap.

I will use threads to describe the ideas, but this works for IPC shared memory as well.

A mutex has two states: locked and unlocked

Multiple threads may share a mutex variable, but only one can *own* it at a time.

To gain ownership a thread locks the mutex. If already locked by another thread this blocks.

To release ownership a thread unlocks the mutex.

Typically a thread can also try to lock a mutex getting a bool flag indicating success/failure. This does not block.

Basic protection of object using a mutex

0. Associate a mutex with the object.
1. Before accessing the object, lock the mutex.
2. Perform the access (read or write).
3. Unlock the mutex

All access goes through this lock/unlock sequence.
If you forget to unklock you get a *deadlock*, and that object cannot be accessed.

`std::mutex` in C++11

- lock(): locks the mutex, blocks if the mutex is not available
- try_lock(): try to lock the mutex, returns false if the mutex is not available
- unlock(): unlocks the mutex

Failing to unlock causes a *deadlock*.
See `simple_mutex_ex.cpp`.

std::lock_guard in C++11

The mutex is a resource that requires careful handling (e.g. to prevent deadlocks). The C++ RAII mechanism is ideal for this.

- lock in a constructor
- unlock in a destructor
- let stack allocation handle the duration of the lock

This can prevent many deadlocks, particularly those caused by an exception interrupting the lock process.
This is what `std::lock_guard` does. It cannot be locked/unlocked outside its constructor/destructor
See `lock_guard_ex.cpp`.

`std::unique_lock` in C++11

`std::unique_lock` is a more sophisticated wrapper around a `std::mutex`. Adds:

- RAII lock/unlock like lock_guard
- deferred locking (for simultaneous locking of multiple mutex's)
- time-constrained try_lock: try_lock_for and try_lock_until
- recursive locking
- transfer of lock ownership
- ability to use with condition variables

See `unique_lock_ex.cpp`.

A *condition variable*, and its associated mutex, allows multiple threads to communicate by one thread notifying others they can proceed.

Suppose multiple threads are sharing a variable with an associated `std::mutex`. A thread that wants to access the variable:

- locks the mutex
- reads or updates the shared variable
- unlocks the mutex
- calls notify_one or notify_all method of the `std::condition_variable` object

A thread waiting on the notification (via a std::condition_variable):

- instantiates a unique_lock on the shared variable's mutex, but does not try to lock it
- instread it calls wait, wait_for, or wait_until method, suspending the thread (possibly with a timeout)

The condition variable recieves a notification when

- another thread calls notify
- a timeout expires
- a spurious wakeup occurs

Upon notification the thread is awakened, and the mutex acquired. You should check the condition and call wait again in case the notification was suprious.
Spurious wakeups are a bit mysterious. Why would the notification be sent if the condition was not true?

- you might have a bug in the other thread
- there are also performance-related reasons why checking in the thread implementation is not done.

See `condition_variable_ex.cpp`.

Semaphores

A *semaphore* is an abtraction of an integer that can be used to share resources between threads.
A threshold (default of 0) can be used to allow multiple threads access, but limit it.

- use Semaphore::up to release a resource, increments the integer
- use Semaphore::down to acquire a resource, decrements the integer (blocks until above threshold).

`up` is also called `release`, `down` is also called `acquire`.
A semaphore can be used to give threshold number of worker threads access to a resource at a time. A common example is limited file-system IO in distrubuted systems.
See `test_semaphore.cpp`.

Building a semaphore using C++11

See `semaphore.h` and `semaphore.cpp`.

Qt has a built in QSempahore

- The constructor creates a semaphore guarding n resource units (by default, 0).
- `acquire(int n = 1)`, acquires n resource units, blocking until n are available
- `available()` returns the number of resources available
- `release(int n = 1)` releases n resource units

There are try versions of aquire that return immediately on failure or after a timeout.

Next Actions and Reminders

Project 3 officially released today:

- Beta Due 4/25 at 8 am
- Final Due 5/2 at 11:59 pm (last day of class)