Lab 9 and 10 - Race Conditions

Race Condition

- A race condition is an unpredictable ordering of events where some orderings may cause undesired behavior.
- Like with processes, threads can execute in unpredictable orderings.
- A thread-safe function is one that will always execute correctly, even when called concurrently from multiple threads.
- printf is thread-safe, but operator<< is not. This means e.g. cout statements could get interleaved!

Threads Share Memory

- Unlike parent/child processes, threads execute in the same virtual address space This means we can e.g. pass parameters by reference and have all threads access/modify them!
- To pass by reference with thread(), we must use the special ref() function around any reference parameters:

```
static void greeting(size_t& i) {
...
}
for (size_t i = 0; i < kNumFriends; i++) {
 friends[i] = thread(greeting, ref(i));
}
```

Mutexes

- We can create a lock-and-key combo by creating a variable of type **mutex**.
- A mutex is technically a type of lock; there are others, but we focus just on mutexes
- When you create a mutex, it is initially unlocked with the key available
- You call *lock()* on the mutex to attempt to lock it and take the key
- You call *unlock()* on a mutex if you have ownership of it and wish to unlock it and return the key. That thread continues normally; one waiting thread (if any) then takes the lock and is scheduled to run.

```
// Assume multiple threads share this same mutex
mutex myLock;
...
myLock.lock();
// only one thread can be executing here at a time
myLock.unlock()
```

When a thread calls lock():

- If the lock is unlocked: the thread takes the lock and continues execution
- If the lock is locked: the thread blocks and waits until the lock is unlocked
- If multiple threads are waiting for a lock: they all wait until it's unlocked, one receives lock (not necessarily one waiting longest)

Mutex Usage

1. Identify a critical section; a section that only one thread should execute at a time.
2. Create a mutex and pass it by reference to all threads executing that critical section.
3. Add a line to lock the mutex at the start of the critical section.
4. Add a line to unlock the mutex at the end of the critical section.

Deadlock

Deadlock is a situation where a thread or threads rely on mutually blocked-on resources that will never become available.

Dining Philosophers Problem

- This is a canonical multithreading program of the potential for deadlock and how to avoid it.
- Five philosophers sit around a circular table, eating spaghetti. There is one fork for each of them. Each philosopher thinks, then eats, and repeats this three times for their three daily meals. To eat, a philosopher must grab the fork on their left and the fork on their right. Then they chow on spaghetti to nourish their big, philosophizing brain. When they're full, they put down the forks in the same order they picked them up and return to thinking for a while. To think, the a philosopher keeps to themselves for some amount of time. Sometimes they think for a long time, and sometimes they barely think at all.

Visit Wikipedia page Link

Condition Variables

A **condition variable** is a variable that can be shared across threads and used for one thread to notify other threads when something happens. Conversely, a thread can also use this to wait until it is notified by another thread.

- We can call wait() to sleep until another thread signals this condition variable.
- We can call notify_all() to send a signal to waiting threads.

```
condition_variable_any myConditionVariable;
...
// thread A waits until another thread signals
myConditionVariable.wait();
...
// thread B signals, waking up thread A
myConditionVariable.notify_all();
```

Semaphore

- A semaphore is a variable type that lets you manage a count of finite resources.
    - You initialize the semaphore with the count of resources to start with
    - You can request permission via semaphore::wait() - aka waitForPermission
    - You can grant permission via semaphore::signal() - aka grantPermission

Note: count can be negative! This allows for some interesting use cases.

```
class semaphore
    {
    public:
        semaphore(int value = 0);
        void wait();
        void signal();

    private:
        int value;
        std::mutex m;
        std::condition_variable_any cv;
    }
```

Programming Exercises

1. simple_muutex.cpp : An example of thread-level parallelism in selling tickets.
2. dining_philosophers.cpp : This program implements the classic dining philosophers simulation. This version works the vast majority of the time, but there's a non-zero chance the simulation deadlocks.
3. dining_philosophers_busy_waiting.cpp : This program implements the classic dining philosophers simulation. This version removes the deadlock concern present in dining_philosophers.cpp, but it allows busy waiting.
4. dining_philosopher_cv.cpp : This program uses condition variables to implement the classic dining philosophers simulation. This is the first version that actually does the right thing.
5. dining_philosophers.cpp : This program implements the classic dining philosophers simulation using the sempahore class.
6. writer_reader.cpp : Implements the classic reader-writer thread example, where one thread writes to a shared buffer and a second thread reads from it. This version uses no thread coordination and is BUGGY!
7. writer_reader_fixed.cpp : Implements the classic reader-writer thread example, where one thread writes to a shared buffer and a second thread reads from it. This version uses thread coordination to ensure that the reader only reads valid data and the writer doesn't overwrite unread data.