CSE 1325: Object-Oriented Programming

Lecture

C++ Threads

Mr. George F. Rice

george.rice@uta.edu

Santa's Elves are subordinate clauses

Today's Topics

- Thread Objects
  - C++ lambda
  - Sleeping a thread
- Mutual Exclusion Objects (mutex)
- Kentucky Derby
  Now in C++!

C++ vs Java Concurrency

- Concepts and code are extremely similar
- Use **`std::thread`** instead of **`Thread`**
  - **`std::thread`** accepts any *function*
    (use a lambda to convert a method into a function)
  - No **`start()`** method – threads run when instanced
- No synchronized methods
  - Use **`std::mutex`** with **`lock()`** / **`unlock()`** methods instead of **`synchronized(mutex) { }`**

Creating a C++ Function Thread

Java

```java
public class SimpleThread implements Runnable {
    @Override
    public void run() {
        for(int i=0; i<10; ++i)
            System.out.println("Thread count " + i);
    }
    public static void main(String args[]) {
        SimpleThread st = new SimpleThread();  // runnable object
        Thread t = new Thread(st);             // Thread instance referencing st
        t.start();                             // Start st.run() in a thread!
        for(int i=0; i<10; ++i)                // Main continues while st.run() runs
            System.out.println("Main count " + i);
        try {t.join();} catch(InterruptedException e) {}
    }
}
```

C++

```cpp
// The function we want to execute on the new thread.
void task1() {
    for(int i=0; i<10; ++i) std::cout << "Thread count " << i << std::endl;
}
int main() {
    // Constructs the new thread. It starts automatically.
    std::thread t{ task1 };
    // Main thread continues in parallel.
    for(int i=0; i<10; ++i) std::cout << "Main count " << i << std::endl;
    // Makes the main thread wait for the new thread to finish execution,
    // therefore blocks its own execution.
    t.join();
}
```

Compiling a C++ Function Thread

- The compiler's -pthread flag is required!

```
ricegf@antares:~/dev/202308/22-c++-threads-file-io/code_from_slides/threads$ \
> g++ --std=c++17 simple_thread.cpp
/usr/bin/ld: /tmp/cce2dgQE.o: in function `std::thread::thread<void (&)(), , void>(void (&)())':
simple_thread.cpp:(.text._ZNSt6threadC2IRFvvEJEvEEOT_DpOT0_[_ZNSt6threadC5IRFvvEJEvEEOT_DpOT0_]+0
x33): undefined reference to `pthread_create'
collect2: error: ld returned 1 exit status
ricegf@antares:~/dev/202308/22-c++-threads-file-io/code_from_slides/threads@ \
> g++ --std=c++17 -pthread simple_thread.cpp
ricegf@antares:~/dev/202308/22-c++-threads-file-io/code_from_slides/threads$
```

std::cout is Not Synchronized

Thread Interference!

```
ricegf@antares:~/dev/202308
Main count 0
Thread count Main count 1
Main count 2
Main count 3
Main count 4
Main count 5
0Main count 6
Thread count 1
Thread count 2
Thread count 3
Thread count 4
Thread count 5
Thread count 6
Thread count 7
Thread count 8

Main count 7
Main count 8
Main count 9
Thread count 9
```

Creating a C++ Method Thread

Lambda Approach

```
#include <iostream>
#include <thread>

// The class containing the method we want to execute on the new thread.
class Bobbin {
  public:
    void task1() {
        for(int i=0; i<10; ++i) std::cout << "Thread count " << i << std::endl;
    }
};
int main() {
    Bobbin bobbin;

    // Constructs the new thread and runs it. Does not block execution.
    std::thread t1{[&bobbin] {bobbin.task1();}};

    // Main thread continues in parallel
    for(int i=0; i<10; ++i) std::cout << "Main count " << i << std::endl;

    // Makes the main thread wait for the new thread to finish execution,
    // therefore blocks its own execution.
    t1.join();
}
```

Lambda!

C++ Lambda

- A **lambda** is a an anonymous function object. It is usually defined where it is invoked.
  - The “capture clause” list variables to be copied or referenced within the lambda
    - All other variables are out of scope
  - The lambda body is always in { }


```
std::thread t1{
 [&bobbin]    {bobbin.task1();}};
```

Capture Clause
(or lambda-introducer)

Lambda Body
(may be multi-line)

Parameter List
(optional – if omitted, parentheses may be omitted also, as we did here)

I recommend *always* using a lambda to create threads.

A reference to “bobbin” is provided to the lambda *function*. Then, the lambda *function* calls our *method*. Voilà – we’ve converted a method to a function!

C++ vs. Java Lambda

- C++ lambda has visibility only to captured variables, either by value `[i]` or by reference `[&bobbin]`
  - Java lambda has same visibility as surrounding code
- C++ lambda drops the () only for zero parameters
  - Java lambda drops the () only for one parameter
- C++ lambda always requires {} around the body
  - Java lambda omits {} for an expression or single statement

Java

```
Thread t1 = new Thread(() -> bobbin.task1());
```

C++

```
std::thread t1{[&bobbin] {bobbin.task1();}};
```

Sleeping a Thread

- In Java, Thread has a simple sleep method

```
Thread.sleep(6000); // Sleep for (at least) 6 seconds
```

- C++ has this_thread::sleep_for

```
std::this_thread::sleep_for(std::chrono::milliseconds(6000));
// OR
std::this_thread::sleep_for(std::chrono::seconds(6));
```

std::mutex Replaces synchronized

```
#include <iostream>
#include <thread>
#include <mutex>

static const int num_threads = 50;
static const int num_decrements = 5000;

int counter = num_threads * num_decrements;

std::mutex m; // global scope
              // (or static field)
// This is the code to be run as threads
void decrementer() {
    for (int i=num_decrements; i > 0; --i) {
        m.lock(); --counter; m.unlock();
    }
}
```

The std::mutex in action.

```
ricegf@pluto:~/dev/cpp/201808/23$ make mutex
g++ -std=c++17 -pthread    mutex.cpp   -o mutex
ricegf@pluto:~/dev/cpp/201808/23$ ./mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201808/23$ ./mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201808/23$ ./mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201808/23$ ./mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201808/23$ ./mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201808/23$
```

Rather than Java's `synchronized(mutex) {--counter;}`

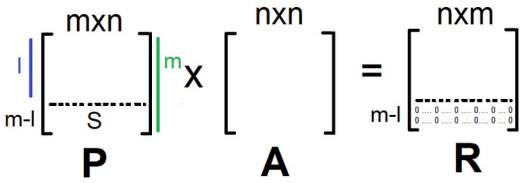
```
int main() {
    //Launch a group of threads
    std::thread t[num_threads];
    for (int i = 0; i < num_threads; ++i) t[i] = std::thread(decrementer);

    //Join the threads with the main thread
    for (int i = 0; i < num_threads; ++i) t[i].join();

    std::cout << "This should be 0: " << counter << std::endl;
    return counter;
}
```

Now in C++!

Kentucky Derby Simulator

- Threads work great for stochastic simulations such as (ahem) games
- We'll let 20 horses (threads) count down their distance from the finish line
  - Competing to be first to grab the mutex that enables them to write THEIR name into the winner's string!

(The matrix multiplier from the Java Concurrency lecture is also available on GitHub for your comparision.)



C++ Kentucky Derby

Horse Interface (including thread)

horse.h

```
#ifndef __HORSE_H
#define __HORSE_H

#include <string>
#include <mutex>

class Horse {
    public:
        Horse(std::string name, int speed, int track_length = 40);
        std::string name(); // What to call this horse
        bool running();     // True if thread is still running
        std::string view(); // String showing its position
        void gallop();      // The thread the moves the horse
        static std::string winner();  // Racing to grab the mutex
                                      // and insert _name here!
    protected:
        std::string _name;  // Name by which horse is known
        bool _running;      // True while thread is running
        int _position;      // Distance from the finish line
        int _speed;         // Rough time between gallops (ms)
        static std::mutex m;  // Controls write access to _winner
        static std::string _winner; // Name of the winning horse
};

#endif
```

Horse Implementation sans Thread

horse.cpp
(1 of 2)

```
#include "horse.h"
#include <chrono>
#include <thread>

Horse::Horse(std::string name, int speed, int track_length) : _name{name},
_speed{speed}, _position{track_length}, _running{true} { }

// The horses race to be first to grab the mutex and insert their name in the
static _winner string!
std::mutex Horse::m;
std::string Horse::_winner = "";
std::string Horse::winner() {return _winner;}

// Getters
std::string Horse::name() {return _name;}
bool Horse::running() {return _running;}

// Representation of the horse on the racetrack
std::string Horse::view() {
    std::string result;
    for (int i = 0; i < _position; ++i) result += (i%5 == 0 ? ':' : '.');
    result += " " + _name;
    if (_winner == _name) result += " (WINNER!!!!!)";
    return result;
}
```

C++ Kentucky Derby

Horse Thread Implementation

```
// The thread
void Horse::gallop() {
    _running = true;
    while (_winner.empty()) {
        std::this_thread::sleep_for(
            std::chrono::milliseconds(_speed + std::rand() % 200));
        if (_winner.empty() && (--_position <= 0)) {
            m.lock();
            if (_winner.empty()) _winner = _name;
            m.unlock();
        }
    }
    _running = false;
}
```

horse.cpp
(2 of 2)

C++ Kentucky Derby
Main Function

horserace.cpp
(1 of 2)

```
#include <iostream>
#include <thread>
#include <chrono>
#include <ctime>
#include <algorithm>
#include <vector>
#include <array>
#include "horse.h"

const int HORSES = 20;
const int TRACK_LENGTH = 40;

int main() {
    // Randomize the pseudorandom number generator
    srand(time(NULL));

    // Pick random names for the horses (based mostly on Kentucky Derby winners)
    std::vector<std::string> names {"Legs of Spaghetti","Ride Like the Calm",
        "Duct-taped Lightning","Flash Light","Speedphobia","Cheat Ah!",
        "Go For Broken","Whining Racer","Spectacle","Cannons a'Boring",
        "Plodding Prince","Lucky Snooze","Wrong Way","Fawlty Powers","Broken Tip",
        "American Zero","Exterminated","Great Regret","Manual","Lockout",
    };
    std::random_shuffle(names.begin(), names.end());
    names.push_back("2 Biggaherd"); // Default name for "too many"
```

C++ Kentucky Derby

Main Function

horserace.cpp

(2 of 2)

```
    // Our competitors, to be assigned random names and speeds (smaller is faster)
    std::vector<Horse> horses;

    for (int i=0; i<HORSES; ++i)
        horses.push_back(Horse{names[std::min(i,
                    static_cast<int>(names.size())-1)], 100 + std::rand() % 100});

    // Instance a thread for each horse
    std::vector<std::thread> threads;
    for (int i=0; i<HORSES; ++i)
        threads.push_back(std::thread{[&horses,i] {horses[i].gallop();}});

    // Display the horse track as the race runs
    bool running = true;
    while (running) {
        std::cout << "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n";
        for (Horse& h : horses) {
            running &= h.running();  // Stop when any horse stops running
            std::cout << h.view() << std::endl;  // Display this horse's position
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    // Join the threads (let all of the horses come to a stop)
    for (std::thread& t : threads) t.join();

    // Announce the winner!
    std::cout << "\n### The winner is " << Horse::winner() << std::endl;
}
```

The Obligatory Makefile

Makefile

```
CXXFLAGS += -std=c++14 -pthread

all: main

debug: CXXFLAGS += -g
debug: main

rebuild: clean main

main: horserace.o horse.o
     g++ -o horserace $(CXXFLAGS) horserace.o horse.o
horserace.o: horserace.cpp horse.h
     g++ $(CXXFLAGS) -c horserace.cpp
horse.o: horse.cpp horse.h
     g++ $(CXXFLAGS) -c horse.cpp
clean:
     -rm -f *.o *~ horserace
```

Running the Kentucky Derby

```
ricegf@pluto:~/dev/cpp/201808/23/horserace$ make
g++ -std=c++14 -pthread -c horserace.cpp
g++ -std=c++14 -pthread -c horse.cpp
g++ -o horserace -std=c++14 -pthread horserace.o horse.o
ricegf@pluto:~/dev/cpp/201808/23/horserace$ ./horserace
```

(It's a lot easier to follow live!)

```
:....:....:....:....:....:.... Fawlty Powers
:....:....:....:....:....:.... Speedphobia
:....:....:....:....:....:.... Broken Tip
:....:....:....:....:....:.... Cannons a'Boring
:....:....:....:....:....:.... Flash Light
:....:....:....:....:....:.... Great Regret
:....:....:....:....:....:.... Whining Racer
:....:....:....:....:....:.... Spectacle
:....:....:....:....:....:.... American Zero
:....:....:....:....:....:.... Plodding Prince
:....:....:....:....:....:.... Wrong Way
:....:....:....:....:....:.... Lockout
:....:....:....:....:....:.... Manual
:....:....:....:....:....:.... Duct-taped Lightning
:....:....:....:....:....:.... Lucky Snooze
:....:....:....:....:....:.... Exterminated
:....:....:....:....:....:.... Legs of Spaghetti
:....:....:....:....:....:.... Cheat Ah!
:....:....:....:....:....:.... Ride Like the Calm
:....:....:....:....:....:.... Go For Broken
```

```
 Fawlty Powers
:....: Speedphobia
:.... Broken Tip
:... Cannons a'Boring
:....: Flash Light
:....:.. Great Regret
:....:. Whining Racer
:....:.. Spectacle
:....:.. American Zero
:.. Plodding Prince
:....:.. Wrong Way
:.... Lockout
:....: Manual
:.. Duct-taped Lightning
:.... Lucky Snooze
 Exterminated
: Legs of Spaghetti
:.. Cheat Ah!
 Ride Like the Calm
:.... Go For Broken

### The winner is Fawlty Powers
ricegf@pluto:~/dev/cpp/201808/23/horserace$
```

What We Learned Today

- Instance std::thread with a function
  - Use a lambda to run a method instead:
    ```
    std::thread t1{[&object] {object.method();}};
    ```
  - The thread runs automatically – NO start() method
- Instance `std::mutex` and use the `lock` / `unlock` methods to manage thread interference