CS3211 Tutorial 1

Threads and Synchronization in C++

(AY 25/26 Semester 2)

January 29, 2026

(Compiled by Benson, thanks to all past and current TAs!)

Join our telegram group!
https://t.me/+MqCIT5ObVgxiMzRl

Contents

Contents

Introduction

YEUNG Man Tsung (Benson)

- First time teaching CS3211!
  - Took this course two years ago...
- Year 4 Computer Science
- **Email**: mtyeung@u.nus.edu
- **Discord**: @mtyeung
- **Telegram**: @mtyeung
- **Consultation**: By appointment / Right after class.

About CS3211

CS3210

"Making things run fast"

- **Raw speedup** on **real hardware**
  - Multi-Core CPU
  - GPU
  - Distributed

CS3211

"Making it concurrent and correct"

- Programming Paradigms & *Language* Constructs
  - Threads, Atomics, Futures, Channels, Async/Await, ...
- Speed with Correctness
  - Harnessing maximum concurrency

About CS3211

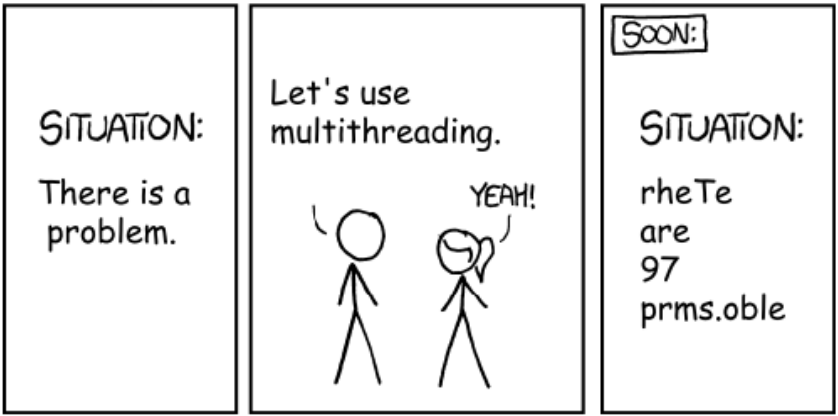
"Concurrency allows you to **structure a problem**
so that it can be **solved in parallel**."
– Concurrency is not Parallelism by Rob Pike

Our Goal:
We learn about different *structures* through different *languages*!

About CS3211

If you don't need concurrency, don't use it!

We teach you the dark arts – use it responsibly.



About CS3211

C++ (50%)

- Threads
- Synchronization
- Atomics
- Memory Ordering
- Debugging
- Lock-Free Programming

Go (25%)

- Lightweight Co-routines (Goroutines)
- Channels and Message Passing
- Etc

Rust (25%)

- Compile-time safety checking Safe futures / async
- Safe data parallelism
- Etc

About CS3211 Tutorials

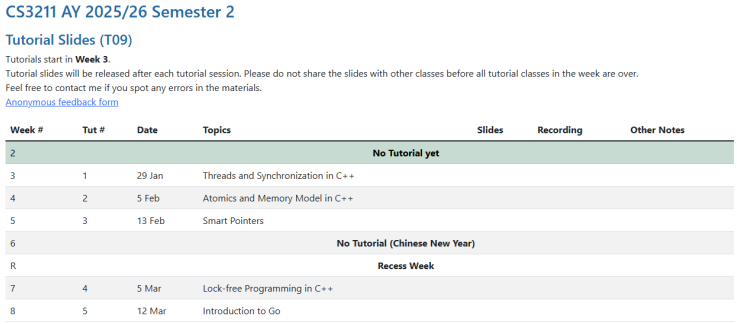
Format:

- We'll go through the tutorial writeup, with substantial amount of live coding.
  - Slides don't contain much, the source of truth is the tutorial writeup.
- You may (1) sit back and watch, or (2) try messing around with the code!
  - Choose one that maximizes your learning.
  - Do try to change the code though (during / after class).
- Ask questions anytime!
  - May not answer immediately.
  - We can discuss advanced stuff after class.
- Attendance is taken (scan QR code at the end of the tutorial).

About CS3211 Tutorials

Resources: https://www.comp.nus.edu.sg/~mtyeung/cs3211_2520.html

- Tutorial slides and recording (no guarantees, best effort).
- Other supplementary materials.



CS3211 AY 2025/26 Semester 2

Tutorial Slides (T09)

Tutorials start in **Week 3**.
Tutorial slides will be released after each tutorial session. Please do not share the slides with other classes before all tutorial classes in the week are over.
Feel free to contact me if you spot any errors in the materials.
Anonymous feedback form

Week #	Tut #	Date	Topics	Slides	Recording	Other Notes
2			**No Tutorial yet**			
3	1	29 Jan	Threads and Synchronization in C++			
4	2	5 Feb	Atomics and Memory Model in C++			
5	3	13 Feb	Smart Pointers			
6			**No Tutorial (Chinese New Year)**			
R			**Recess Week**			
7	4	5 Mar	Lock-free Programming in C++			
8	5	12 Mar	Introduction to Go			

Contents

RAII (Resource Acquisition is Initialization)

What happens if we compile & run this program?

```cpp
#include <chrono>
#include <iostream>
#include <string>
#include <thread>

int main() {
  std::string* raw_world_ptr = nullptr;

  {
    std::string world("12345678901234567890");
    raw_world_ptr = &world;
  }

  std::cout << *raw_world_ptr << std::endl;

  return 0;
}
```

Link to fsmbolt

A. Compile-time error
B. Undefined behaviour
C. Legal behaviour: Segfault
D. Legal behaviour: Program prints "123..."

Classic "use-after-free" bug – undefined behavior!

Programs as Data

"For one person's program is another program's data." — Olivier Danvy

std::thread RAII Behaviour

```cpp
#include <chrono>
#include <iostream>
#include <string>
#include <thread>

void test() {
    std::cout << "hello!\n";
}

int main() {
    std::thread t1(test);
    t1.join();
    return 0;
}
```

Link to fsmbolt

`join()`: Waits for the thread to finish.
`detach()`: Detach the thread.

std::thread RAII Behaviour

What happens if we compile & run this program?

```
1  #include <chrono>
2  #include <iostream>
3  #include <string>
4  #include <thread>
5
6  void test() {
7    std::cout << "hello!\n";
8  }
9
10 int main() {
11   std::thread t1(test);
12   // t1.join();
13   return 0;
14 }
```

Link to fsmbolt

A. Compile-time error
B. Undefined behaviour
C. Legal behaviour: Program returns 0
D. Legal behaviour: Program does not return 0

std::thread C++ Standard
If the thread is joinable, `std::terminate` is called.

std::thread Move Semantics

What happens if we compile & run this program?

```
1  #include <chrono>
2  #include <iostream>
3  #include <string>
4  #include <thread>
5
6  void test() {
7    std::cout << "hello!\n";
8  }
9
10 int main() {
11   std::thread t1(test);
12   std::thread t2 = std::move(t1);
13   t1.join();
14   return 0;
15 }
```

Link to fsmbolt

A. Compile-time error
B. Undefined behaviour
C. Legal behaviour: Program returns 0
D. Legal behaviour: Program does not return 0

Program throws an exception, should call `t2.join()` instead.

Takeaways

- RAII is a useful idea for **safety**: clean up resources at the end of a scope.
  - However, you can easily and unintentionally bypass this safety mechanism.
- RAII for threads can also be useful to clean up.
  - But it will behave in unexpected ways if you don't `join()` or `detach()`.
- `std::move` is a way to efficiently transfer resources and 'ownership'.
  - Actual move semantics depend on the specifications, and can differ based on type!

Contents

Task 1: Preventing a data race

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

int counter;

int main() {
  std::thread t0{[]() { ++counter; }};
  std::thread t1{[]() { ++counter; }};

  t0.join();
  t1.join();

  std::cout << counter << std::endl;

  return 0;
}
```

Link to fsmbolt

What is a data race?

Where is the data race?

Task 1: Preventing a data race

Utilizing RAII:

- `std::unique_lock`
  - Supports manual lock/unlock, deferred locking, try-lock.
  - Uses more memory.
  - Movable.
- `std::lock_guard`
  - The most lightweight option.
- `std::scoped_lock`
  - Locks zero or more mutexes.
  - Implements deadlock avoidance for multiple locks.

Or try atomics!

Task 2: Concurrent Thread-Safe Queue

```
1  class JobQueue {
2    std::queue<Job> jobs;
3
4  public:
5    void enqueue(Job job) {
6      jobs.push(job);
7    }
8
9    std::optional<Job> try_dequeue() {
10     if (jobs.empty()) {
11       return std::nullopt;
12     } else {
13       Job job = jobs.front();
14       jobs.pop();
15
16       return job;
17     }
18   }
19 };
```

Link to fsmbolt

Is this thread safe?

- Producer-producer.
- Producer-consumer.
- Consumer-consumer.

Task 2: Concurrent Thread-Safe Queue

```
1  void enqueue(Job job) {
2    mut.lock();
3
4    jobs.push(job); // exception?
5
6    mut.unlock();
7  }
```

Much simpler with RAII!

```
1  void enqueue(Job job) {
2    mut.lock();
3    try {
4      jobs.push(job); // exception?
5    } catch (...) {
6      mut.unlock();
7      throw;
8    }
9    mut.unlock();
10 }
```

Task 2: Concurrent Thread-Safe Queue

Thread 1:

```
1  std::optional<Job> try_dequeue() {
2    std::scoped_lock lock{mut}; mutex
3    if (jobs.empty()) {
4      return std::nullopt;
5    }
6    count.acquire();            semaphore
7    Job job = jobs.front();
8    jobs.pop();
9    return job;
10 }
```

Thread 2:

```
1  Job dequeue() {
2    count.acquire();            semaphore
3    std::unique_lock lock{mut}; mutex
4    Job job = jobs.front();
5    jobs.pop();
6    return job;
7  }
```

How can a deadlock happen?

Task 2: Concurrent Thread-Safe Queue

Condition Variable: "Sleep/wake coordination"

Waiting Thread

```
1 cond.wait(lock);
2 // lock is released while waiting
```

Signaling Thread

```
1 cond.notify_one();
2 // or
3 cond.notify_all();
```

Issue of **spurious wakes**: we can be released from the condition variable without a notify from another thread (an optimization).

- We have to check each time we wake up (which hopefully doesn't happen too often).

Task 2: Concurrent Thread-Safe Queue

Monitor: A mutex + A condition variable + A condition to wait for.

Waiting Thread

```
1  std::unique_lock lock{mutex};
2
3  while (/*condition not satisfied*/) {
4    cond.wait(lock);
5  }
```

Signaling Thread

```
1  std::scoped_lock lock{mutex};
2
3  // when condition is satisfied
4  cond.notify_one();
5  // or
6  cond.notify_all();
```

Extra: Monitors in Java

Extra Slide

Every object in Java is a monitor.

```
synchronized (object) { // enter monitor if no one is in, else join monitor queue
    ...
    object.wait(); // join wait queue
    ...
    object.notify(); // pick one process from wait queue and unblock
    ...
    object.notifyAll(); // unblock all processes in wait queue
    ...
}
```

Summary

- RAII (Resource Acquisition is Initialization).
- Synchronization mechanisms in C++.
  - `std::mutex`
  - `std::unique_lock`, `std::scoped_lock`, `std::lock_guard`
  - `std::counting_semaphore`
  - `std::condition_variable` and spurious wakeups
- The difficulty of managing even simple thread-safe structures - have to be entirely sure of safety!

Attendance

Session ID: 1938252
CS3211 Tutorial 1
29 January 2026 14:00-16:00
BIZ2-02-02 - SEMINAR ROOM 2-2

That's it!

Anonymous Feedback (throughout the semester):

https://forms.gle/6a9T4t88wNwYxG4G8

The link to the tutorial slides will be posted in telegram.