FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

FACULTY OF ENGINEERING

Lecture 13

C++ Threads

Parallel Programming

- Multithreaded programming allows you to perform multiple calculations in parallel
- Typical Problems:
  - Race conditions: can occur when multiple threads want to read/write to a shared memory location
  - Deadlocks: threads that are blocking indefinitely because they are waiting to acquire access to resources currently locked by other blocked threads

C++ Memory Model I

- Problem: Hard for programmers to reason about correctness
- Without precise semantics, hard to reason if compiler will violate semantics
- Compiler transformations could introduce data races without violating language specification and the resulting execution could yield unexpected behaviors.

C++ Memory Model II

- Two aspects to the memory model:
  - I. the basic structural aspects – memory layout
    - Every variable is an object, including those that are members of other objects.
    - Every object occupies at least one memory location.
    - Variables of fundamental type such as int or char are exactly one memory location, whatever their size, even if they're adjacent or part of an array.
    - Adjacent bit fields are part of the same memory location.
  - II. The concurrency aspects
    - If there is no enforced ordering between two accesses to a single memory location from separate threads, these accesses are not atomic,
    - if one or both accesses is a write, this is a data race, and causes undefined behaviour.

Atomics

- Allow atomic accesses, which means that concurrent reading and writing without additional synchronization is possible
- In this way race conditions can be solved
- Example
  - atomic<int> counter(0); // global variable
  - ++counter; // executed in multiple threads

C++ Threads

```
#include <iostream>
#include <thread>
using namespace std;

void counter(int id, int numIterations) {
    for (int i = 0; i < numIterations; ++i) {
      cout << "Counter " << id << " has value "; cout << i << endl;
} }

int main() {
  cout.sync_with_stdio(true); // Make sure cout is thread-safe

  thread t1(counter, 1, 6);
  thread t2(counter, 2, 4);

  t1.join();
  t2.join();

  return 0;
}
```

Mutual Exclusion (mutex)

- Step 1: A thread wants to read/write to memory shared with another thread and tries to lock a mutex object. If another thread is currently holding this lock, the thread blocks until the lock is released
- Step 2: Once the thread has obtained the lock, it is free to read/write to shared memory
- Step 3: After the thread is finished with reading/writing it releases the lock. If two or more threads are waiting on the lock, there are no guarantees as to which thread will be granted the lock

Locks

- lock_guard
  - binds mutex in constructor and frees it in destructor (RAII)
- unique_lock
  - Does not have strict 1:1 relation to mutex

Locks

```
#include <mutex>
using namespace std;

mutex mut1;
mutex mut2;

void process() {
  unique_lock<mutex> lock1(mut1, defer_lock_t());
  unique_lock<mutex> lock2(mut2, defer_lock_t());
  lock(lock1, lock2); // Locks acquired
}

int main() {
  process();
  return 0;
}
```

Inter-thread communication

```
#include <iostream>
#include <future>
using namespace std;

int calculate() {
  return 123;
}

int main() {
  auto fut = async(calculate);
  //auto fut = async(launch::async, calculate);
  //auto fut = async(launch::deferred, calculate);

  // Do some more work...

  // Get result
  int res = fut.get();
  cout << res << endl;
  return 0;
}
```

Future and Promise

- promise
  - The class template std::promise provides a facility to store a value or an exception that is later acquired asynchronously via a std::future object created by the std::promise object
- future
  - The class template std::future provides a mechanism to access the result of asynchronous operations

Volatile Keyword

- Java/C#
  - tells the compiler that the value of a variable must never be cached as its value may change outside of the scope of the program itself. The compiler will then avoid any optimisations that may result in problems if the variable changes "outside of its control“.
  - provides both ordering and visibility between threads
- C++
  - is needed when developing embedded systems or device drivers, where you need to read or write a memory-mapped hardware device. The contents of a particular device register could change at any time, so you need the keyword to ensure that such accesses aren't optimised away by the compiler.

Volatile Keyword

```
int a = 0; int b = 0;
volatile int count = 0;
a = 1;
count = 1;
b = 2;

count = 2;
```

- Java
  - if count == 1, then the assertion a == 1 must be true. Similarly, if count == 2 then assertion that a == 1 && b == 2 must be true. This is what is means by the strict memory guarantee that Java offers that C/C++ does not.
- C++
  - volatile only guarantees that the count variable cannot be reordered against each other, ie. if count == 2, then count = 1 must necessarily precede it. However, there is neither a guarantee that a == 1, nor that b == 2.