SPECTRAL:
TECHNOLOGIES

WEEK 8

Multithreading in C++

Basic building blocks

Threads

- Threads allow to run parallel computations. On Linux thread is a single unit of scheduling
- C++ provides only basic api to manipulate threads and synchronize them. Anything more complex is left for a programmer to implemement
- C++ also doesn't provide any api to manage the scheduling properties, like affinity or priority. However, Linux api can be used for that
- Creating a new thread

```
std::thread t([]() {
    // code to run
});
```

Threads

- Before any thread can be destructed, it should be **joined** or **detached**
  - otherwise `std::terminate` is called
- `t.join()` to wait until `t` finished it's execution. After that, the destructor of `t` can be called safely
- `t.detach()` to detach thread. Detached thread runs independently of `t` and cannot be joined after that. They are killed when the process stops

SPECTRAL:
TECHNOLOGIES

Threads

- Note, that threads cannot be copied, but can be moved
- The most basic pipeline:

```
std::vector<std::thread> workers;
workers.reserve(workers_count);
for (int i = 0; i < workers_count; ++i) {
     workers.emplace_back([i]() {
          work(i); // we usually want to know the worker's num
     });
}

// some work in the main thread
// ...

for (auto& t : workers) {
     t.join();
}
```

Threads

- C++ doesn't allow to stop a thread, so it should be implemented manually if needed
- `detach` is rarely needed, usually for some background threads, like logging, monitoring and etc
- `t.native_handle()` can be used to access raw os threads api
- You can set affinity, priority and many other properties pthreads provide

```
std::thread t([]() {
    work();
});
pthread_setaffinity_np(t.native_handle(), sizeof(cpu_set_t), &affinity);
t.join();
```

Threads

- By default only *main* thread is running. It doesn't have any special properties
  - However, the process exits when the main thread stops
- Each thread has it's own stack, but all threads have access to each other's memory:

```
Object o;
std::vector<std::thread> workers;
for (int i = 0; i < workers_count; ++i) {
    workers.emplace_back([&o, i]() {
        // o sits in the main's thread stack
        // i is copied to the current's thread stack
        // this call goes to the current's thread stack
        work(o, i);
    });
}
```

- Due to memory sharing, data synchronization is important

SPECTRAL:
TECHNOLOGIES

Data synchronization

- **Multiple** threads can access the same memory cell in **read-only** manner
- There can be **only one** thread **writing** to some memory cell
- Otherwise its **data race** (which is UB)
  - Due to the complexity of optimizations and cpu execution, its hard to predict the behaviour of a program with data race

```
void worker(const std::vector<int>& shared_data, int worker_num) {
    // safe, since each worker has read-only access to shared_data
}
void worker(std::vector<int>& shared_data, int worker_num) {
    shared_data[worker_num] = ...;
    // safe, since each worker modifies it's own cell
    // may be bad due to false sharing
}
void worker(std::vector<int>& shared_data, int worker_num) {
    shared_data.push_back(...);
    // data race, since push_back modifies vector's state
```

SPECTRAL:
TECHNOLOGIES

Data synchronization

- Most of std isn't thread-safe
- `std::mutex` can be used to provide mutual exclusion:

```
std::mutex m;
for (int i = 0; i < workers_count; ++i) {
    workers.emplace_back([&m, i]() {
        // do something
        m.lock();
        // critical section
        m.unlock();
        // do something else
    });
}
```

- `mutex` cannot be copied or moved

Data synchronization

- Its better to use a RAII wrapper:

```
std::mutex m;
{
    std::lock_guard lock(m);
    // critical section
}
```

- Every not thread-save container can be turned into one by securing everything with `mutex`, but it may be inefficient

Data synchronization

- `mutex` can be used for simple synchronization (but there are obviously better ways):

```
bool ready = false;
std::mutex m;
void thread1() {
    // some work
    std::lock_guard lock(m);
    ready = true;
}

void thread2() {
    // some work
    for (;;) {
        std::lock_guard lock(m);
        if (ready) {
            break;
        }
    }
    // more work
}
```

Condition Variable

- Synchronization primitive used with a `std::mutex` to block one or more threads until another thread both modifies a shared variable (the condition) and notifies the condition_variable

Condition Variable

```
bool ready = false;
std::mutex m;
std::condition_variable cv;
void thread1() {
     // some work
     {
         std::unique_lock lock(m);
         ready = true;
     }
     cv.notify_one();
}

void thread2() {
     // some work
     {
         std::unique_lock lock(m);
         cv.wait(lock, []{return ready;});
     }
     // more work
}
```

Condition Variable

- There are also `wait_for` and `wait_until`
- There is also `notify_all` to notify all waiting threads
- Any `notify` before `wait` call is lost. This greatly limits the possible usages of cv
- The order in which threads are notified isn't specified
- Sometimes *spurious wakeup* can happen: waiting thread is notified even if the condition is false

Shared mutex

- Another synchronization primitive, also called *rw-lock* (read-write lock)
- Provides either an exclusive access to critical section (for a writer) or shared access for multiple threads (for readers)

SPECTRAL:
TECHNOLOGIES

Shared mutex

```
class ThreadSafeCounter {
 public:
  // Multiple threads/readers can read the counter's value at the same time.
  unsigned int get() const {
    std::shared_lock lock(mutex_);
    return value_;
  }

  // Only one thread/writer can increment/write the counter's value.
  void increment() {
    std::unique_lock lock(mutex_);
    ++value_;
  }

  // Only one thread/writer can reset/write the counter's value.
  void reset() {
    std::unique_lock lock(mutex_);
    value_ = 0;
  }

 private:
  mutable std::shared_mutex mutex_;
  unsigned int value_ = 0;
};
```

Atomics and memory model

- Both compiler and cpu reorder instructions and memory accesses for faster execution
- *Memory model* is a specification of the allowed behavior of multithreaded programs executing with shared data. It specifies what can and cannot be reordered, gives a formal definition of data race and much more
- Understanding C++ memory model is essential for writing efficient low-level multithreaded code with atomics
- However, its not a part of this course. Instead we focus here on a small subset of `std::atomic`, which is quite simple to understand

SPECTRAL:
TECHNOLOGIES

Atomics

- Data race, assert may fail

```
int x = 0, y = 0;
-Thread 1-          -Thread 2-
y = 1               if (x == 2)
x = 2;                assert (y == 1)
```

- No data race, assert cannot fail

```
std::atomic<int> x = 0, y = 0;
-Thread 1-          -Thread 2-
y.store(1)          if (x.load() == 2)
x.store(2);           assert (y.load() == 1)
```

- No data race, but assert can fail

```
std::atomic<int> x = 0, y = 0;
-Thread 1-                                  -Thread 2-
y.store(1, std::memory_order_relaxed)         if (x.load(std::memory_order_relaxed) == 2)
x.store(2, std::memory_order_relaxed);          assert (y.load(std::memory_order_relaxed) == 1)
```

Atomics

- `std::atomic` itself doesn't introduce any memory overhead for primitive types
  - so `sizeof(std::atomic<int>) == sizeof(int)`
  - on x86/64 its true for types with size <= 8
  - otherwise `std::atomic<T>` is usually `T + std::mutex`
- Operations with `std::atomic` introduce *memory barrier*, which serves as a constraint of how operations around this barrier can be reordered. Its a constraint for both compilers and cpu. Some operations may translate into atomic instructions.
- Memory order in `std::atomic` specifies how strict are these reordering rules.
- We only consider the default memory order, which is `std::memory_order_seq_cst` or sequentially-consistent ordering

Sequential-consistent model

- The most intuitive model. It guarantees, that all threads must see the same order of memory operations
- In another words, it guarantees there is a single timeline, where all operations are ordered, and this timeline is common for all threads
  - The ordering is basically the same as written in the code
- Its not true for other orderings, like `relaxed`
  - e.g. the same variable may have different values in different threads
  - Its closely tied with cache coherence as well
- Sequential-consistent atomics are the simplest to use for data synchronization between threads. It may be much faster, than mutexes or cond. vars do to the absense of syscalls

Atomics

- Usually used via `std::atomic<bool>` for some flags (ready flags) or `std::atomic<int>` for some counters
- Apart from `load/store` and arithmetic operations, you may need to use these:
  - `exchange` to atomically return the current value and exchange it with some new value
  - `compare_exchange_weak` and `compare_exchange_strong` to perform `exchange` on some condition atomically

SPECTRAL:
TECHNOLOGIES

Atomics and shared_ptr

- `std::shared_ptr` is partially thread-safe
- You can't do that (data race):

```
std::shared_ptr<Object> p;
-Thread 1-                            -Thread 2-
p = std::make_shared<Object>(...)    p->method();
```

- But you can do that (ref. counters are atomics):

```
std::shared_ptr<Object> p;
-Thread 1-                            -Thread 2-
auto a = p;                           auto b = p;
```

SPECTRAL:
TECHNOLOGIES

Atomics and shared_ptr

- In some cases (mostly, lock-free programming) its nice to have the actual `atomic<shared_ptr<T>>`
- Before c++20, there were functions like `std::atomic_load(const std::shared_ptr<T>* p)`. They use `mutex` inside, so they are not "atomics" as you might expect by the name
- `C++20` introduced `std::atomic<std::shared_ptr<T>>`, however its only implemented since libstdc++12

SPECTRAL:
TECHNOLOGIES

Additional materials

- Anthony Williams "C++ Concurrency in Action"