SPECTRAL:
TECHNOLOGIES

WEEK 8

Practical multithreading in C++

Operation costs

- `mutex.lock()` takes ~25 ns
- Creation of a new thread may take 10-100 us
- Badly designed multithreaded code may execute slower, than singlethreaded
- It doesn't make sense to create more threads, than the number of cores. However, sometimes you need a lot of logical "threads"
    - In that case you probably need coroutines or fibers or another async primitives

Accurate locks

- Consider we want a multithreaded hashmap. The simplest solution is `hashmap + std::mutex`, or `hashmap + std::shared_mutex`
- Given the structure of hashmap, its not wise to lock the entire table
- Instead lets make `T` single-threaded hashmaps and another hash function, which maps a key to one of these hashmap
  - Every hashmap uses `shared_mutex`, but now queries to different hashmaps may run simultaneously
  - `T` may be selected as `std::thread::hardware_concurrency()`

Thread pools

- Creating a new thread everytime you need to perform some work is highly ineffective
- On the other hand manually managing multiple threads leads to bugs and unreadable code
- Thats why it makes sense to have a separate entity, that manages worker-threads and provides some api to run tasks inside. Such entity is called *thread pool executor*
- There is no such entity in std, but there are in `boost` or `folly`
- **Don't use** `std::async` , since it doesn't provide any control over how exactly your task is gonna be executed

SPECTRAL:
TECHNOLOGIES

Thread pool basic scheme

```
TaskQueue ThreadPool::q(max_workers); // multithreaded queue
void ThreadPool::ThreadPoolWorker() { // executed by every worker
    while (!stopped) { // some atomic flag e.g.
        auto task = q.pop(); // blocks if q is empty
        run(task); // in the current thread
    }
}

void ThreadPool::Run(Task task) {
    q.push(std::move(task)); // blocks if q is full
}

int main() {
    ThreadPool pool(max_workers);
    for (;;) {
         // ...
         Task task = ...; // got a new task
         pool.Run(task);
    }
}
```

Mutex and spinlock

— Consider 2 different ways of entering critical section:

```
void f1(std::mutex& m) {
    m.lock();
    // ...
    m.unlock();
}

void f2(std::atomic<bool>& lock) {
    while (lock.exchange(true)) {
        // loop
    }
    // ...
    lock.store(false);
}
```

— In the first case if the mutex is locked, thread sleeps until it can enter the critical section. CPU isn't loaded with work

SPECTRAL:
TECHNOLOGIES

Mutex and spinlock

- In the second case thread runs actively until it acquires the lock. CPU core is 100% loaded
- This type of lock is called *spinlock*
  - It may be worse if you have big critical section and high cpu cores' usage
  - For small locks it's better, since the latency is much lower
  - For low-latency applications, this is the only type of lock you want to use (ideally, none)
- In Linux `std::mutex` initially works as spinlock for some time
  - if it can't acquire the lock, `futex` is used
  - `futex` works in user-space in a noncontended case
  - kernel is involved in contended cases

SPECTRAL:
TECHNOLOGIES

Debugging tools

- Debugging data races, deadlocks and softlocks is obviously hard
- `-fsanitize=thread` is essential for testing
  - Note, that tsan doesn't support more, than 64 mutexes
- gdb supports threads, but it isn't very convinient
  - `info threads` to see all threads and the current position in each thread
  - `thread <x>` to switch to thread number `x` (by default, main is active)
  - `thread apply` to apply some command to multiple threads, like `threads apply all bt`

SPECTRAL:
TECHNOLOGIES

Summary

- Try to avoid locks as much as possible
- For low latency prefer spinlocks to mutexes
- Its better to build multithreaded code around executors and queues for synchronization, than deal with raw mutexes and cond. vars
- Don't forget about *false sharing*