Readers-Writers Problem

Parallel Computing

Goals

- ✯ Study an instance of the readers-writers problem.
- ✯ Practice the manipulation of semaphores and mutexes.
- ✯ Think in groups of 2 or 3 students.

1 Readers-Writers Problem

To explain the problem, here an extract of Wikipedia:

> In computer science, the readers–writers problems are examples of a common computing problem in concurrency. There are at least three variations of the problems, which deal with situations in which many concurrent threads of execution try to access the same shared resource at one time.
>
> Some threads may read and some may write, with the constraint that no thread may access the shared resource for either reading or writing while another thread is in the act of writing to it. (In particular, we want to prevent more than one thread modifying the shared resource simultaneously and allow for two or more readers to access the shared resource at the same time). A readers–writer lock is a data structure that solves one or more of the readers–writers problems.

For instance, you can think about a web-page. Many people want to read the page, and from times to times, we want to update the page while avoiding someone to read a page being partially updated.

Your goal is to implement the missing functions `read` and `write` of the following class:

```
template <class T>
class RWResource {
  T data;
  // ... more attributes

public:
  template <class F>
  void read(F f) const {
    // ... some code
    f(data);    // note that here "data" is const
    // ... some code
  }

  template <class F>
  void write(F f) {
    // ... some code
    f(data); // note that here "data" is not const
    // ... some code
  }
};
```

- First, you should try to implement a version that is correct, that is, it preserves the invariants that no write can occur concurrently to a read, while multiple reads are allowed.
- Can your solution lead to starvation of the readers or the writers?
- If yes, propose an alternative.

2 Extract of Semaphore API

```
namespace std {
  template<ptrdiff_t LeastMaxValue = /* implementation-defined */>
  class counting_semaphore {
  public:
    constexpr explicit counting_semaphore(ptrdiff_t desired);
    void release(ptrdiff_t update = 1);
    void acquire();
    bool try_acquire() noexcept;
  };
}
```

3 Producer-consumer code (from the video)

```
constexpr int maxsem = std::numeric_limits<int>::max();

template <class T>
class ConcurrentQueue {
  std::queue<T> queue;
  std::mutex mutex_queue;
  std::counting_semaphore<maxsem> is_empty;

public:
  ConcurrentQueue(): is_empty(0) {}

  void push(T&& x) {
    mutex_queue.lock();
    queue.push(std::move(x));
    is_empty.release();
    mutex_queue.unlock();
  }

  T pop() {
    is_empty.acquire();
    mutex_queue.lock();
    T val(std::move(queue.back()));
    queue.pop();
    mutex_queue.unlock();
    return std::move(val);
  }
};
```

4 Extract of Mutex API

```
namespace std {
  class mutex {
  public:
    constexpr mutex() noexcept;

    void lock();
    bool try_lock();
    void unlock();
  };
}
```

5 Extract of condition variable API

```
namespace std {
  class condition_variable {
  public:
    condition_variable();
    void notify_one() noexcept;
    void notify_all() noexcept;
    void wait(unique_lock<mutex>& lock);

    template<class Pred>
    void wait(unique_lock<mutex>& lock, Pred pred);
  };
}
```

Note that `wait` can wake up even if no notify occured. When waking up, you still need to check a Boolean flag to make sure it is a “real wake-up”. You can use the overload of `wait` to achieve that, as shown in the following examples:

6 Example of condition variable

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


// Note: don't use global variables in your code, thanks.
std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
    // wait until main() sends data
    std::unique_lock lk(m);
    cv.wait(lk, []{ return ready; });

    // after the wait, we own the lock
    std::cout << "Worker thread is processing data\n";
    data += " after processing";
    // send data back to main()
    processed = true;
    std::cout << "Worker thread signals data processing completed\n";

    // manual unlocking is done before notifying, to avoid waking up
    // the waiting thread only to block again (see notify_one for details)
    lk.unlock();
    cv.notify_one();
}

int main()
{
    std::thread worker(worker_thread);
    data = "Example data";
    // send data to the worker thread
    {
        std::lock_guard lk(m);
        ready = true;
        std::cout << "main() signals data ready for processing\n";
    }
    cv.notify_one();
    // wait for the worker
    {
        std::unique_lock lk(m);
        cv.wait(lk, []{ return processed; });
    }
    std::cout << "Back in main(), data = " << data << '\n';
    worker.join();
}
```