Advanced Parallel Programming
Exercise 4

Fabian Czappa

TECHNISCHE UNIVERSITÄT DARMSTADT

Summer term 2025
12.06.2025

Please solve the following tasks by 12.06.2025. The results are not graded, but a solution is discussed on 12.06.2025.

Task 1: Synchronization

Suppose you need to synchronize two tasks:

- A task notifies a second, asynchronously running task that a particular event has occurred, because the second task cannot proceed until the event has taken place;
- The event occurs only once;
- There is no data to be transferred between the two tasks;
- Whether the raw event has occurred is only available to the first task.

1a) Benefits

What are the possible approaches to implement such communication? Give your solutions in code, and discuss their advantages and disadvantages.

1b) Multiple tasks

If there are multiple tasks needed to be notified, what changes are necessary for each of your proposed approach?

Solution:

```
/* Condition variable approach */
std::mutex m;
std::condition_variable cv;

/* checking thread */
{
  ...
  if (condition is true)
    cv.notify_one();
  ...
}

/* reacting thread */
{
  ...
  std::unique_lock<std::mutex> lk(m);
  cv.wait(lk);    // spurious wakeup?
  ...
}

/*****************************************************/

```

```
/* Flag approach  */
std::atomic<bool> flag(false);

/* checking thread */
{
  ...
  if (condition is true)
    flag = true;
  ...
}

/* reacting thread */
{
  ...
  while (!flag);    // busy waiting
  ...
}

/*******************************************************/

/* Combined approach */
std::mutex m;
std::condition_variable cv;
bool flag(false);

/* checking thread */
{
  {
    std::lock_guard<std::mutex> g<m>;
    if (condition is true)
      flag = true;
  }
  cv.notify_one();
  ...
}

/* reacting thread */
{
  ...
  std::unique_lock<std::mutex> lk(m);
  cv.wait(lk, []{ return flag; })
  ...
}

/*******************************************************/

/* Promise/Future approach */
std::promise<void> p;

/* checking thread */
{
  ...
  if (condition is true)
    p.set_value();
  ...
}

/* reacting thread */
{
  ...
  p.get_future().wait();
  ...
}

/*******************************************************/

/* Promis/Future approache
 * for multiple reacting tasks
 */
 std::promise<void> p;
```

```
/* checking thread */
{
  auto sf = p.get_future().share();  // sf: std::shared_future<void>
  std::vector<std::thread> vt;

  for (int i = 0; i < numThreadsToRun; ++i) {
    vt.emplace_back( [sf]{ sf.wait(); react(); } );
  }

  p.set_value();
  for (auto& t : vt) {
    t.join();
  }
}
```

Listing 1: Approaches for one-shot communication

Task 2: Creation of a custom mutex type

In this task, you should create a custom mutex type. Firstly, make yourself familiar with the methods a `std::mutex` provides[1]. You do not need to implement the `native_handle`, but the other functionality should be present.

As an internal locking/unlocking mechanism, you can use an `std::atomic_flag` with the provided functionality – even though it will technically be covered later in the course.[2] You can choose between "busy-waiting", i.e., the thread will test the flag repeatedly, or a defered waiting mechanism by calling `wait`.

Solution:

In this solution, note that we are using an atomic counter to simulate spurious failures and exceptions during acquisition of the lock. Officially, they will be introduced in slide deck 8.

```
#include <atomic>
#include <exception>
#include <iostream>
#include <mutex>

// Example from https://cppreference.com/w/cpp/atomic/atomic_flag.html
class mutex {
    std::atomic_flag m_{};

public:
    void lock() noexcept {
        while (m_.test_and_set())
            m_.wait(true);
    }

    bool try_lock() noexcept {
        return !m_.test_and_set();
    }

    void unlock() noexcept {
        m_.clear();
        m_.notify_one();
    }
};

class might_fail_mutex {
    std::atomic_flag m_{};
    std::atomic<int> counter{ 0 };

public:
    void lock() noexcept(false) {
```

[1] https://en.cppreference.com/w/cpp/thread/mutex.html
[2] https://cplusplus.com/reference/atomic/atomic_flag/

```
        // Technically, you do not know this functionality yet
        auto val = counter++;
        if (val % 7 == 0) {
            throw std::exception("Simulated failure on counter value");
    }

        while (m_.test_and_set())
            m_.wait(true);
    }

    bool try_lock() noexcept(false) {
        // Technically, you do not know this functionality yet
        auto val = counter++;
        if (val % 7 == 0) {
            throw std::exception("Simulated failure on counter value");
        }

        return !m_.test_and_set();
    }

    void unlock() noexcept {
        m_.clear();
        m_.notify_one();
    }
};

int main() {
    auto mut_1 = mutex{};
    auto mut_2 = might_fail_mutex{};

    for (auto i = 0; i < 30; i++) {
        auto thrown = false;
        try {
      auto lg = std::lock_guard<mutex>(mut_1);
        }
        catch (...) {
      thrown = true;
        }

        if (thrown) {
      std::cout << "Iteration " << i << ": mutex lock failed.\n";
        }
        else {
      std::cout << "Iteration " << i << ": mutex lock succeeded.\n";
        }
    }

    for (auto i = 0; i < 30; i++) {
        auto thrown = false;
        try {
            auto lg = std::lock_guard<might_fail_mutex>(mut_2);
        }
        catch (...) {
            thrown = true;
        }

        if (thrown) {
            std::cout << "Iteration " << i << ": might_fail_mutex lock failed.\n";
        }
        else {
            std::cout << "Iteration " << i << ": might_fail_mutex lock succeeded.\n";
        }
    }

    return 0;
}
```

Listing 2: Two different self-implemented mutexes, one with ‘spurious’ failure