Laboratory for Parallel Programming
Prof. Dr. Felix Wolf

TECHNISCHE UNIVERSITÄT DARMSTADT

Marcus Ritter, marcus.ritter@tu-darmstadt.de

Advanced Multithreading in C++ (WS 21/22)

Exercise 2

Please solve the following tasks by November 30th, 2021. The results are not graded, but a solution is discussed on November 30th, 2021.

Task 1

A programmer wants to use asynchronous tasks to make her code run more concurrently. Therefore, she wrote a main thread, which reads the content of a file, and a worker thread to process the read data. In order to work, the main thread needs to notify the worker threads to start processing a part of the data once it has been read. Listing 1 shows her code skeleton. To do the necessary synchronization she wants to use conditional variables.

Listing 1: Sample program

```
void processData(){
    // do the processing
}

void readFile(){
    // read the input file
}
```

1. Write a code that uses conditional variables to synchronize the main with the worker thread.
2. What are the disadvantages of using conditional variables for synchronization?
3. What other approaches exist to solve this problem? Write a code example for each of them.
4. The programmer wants to extend the code so that multiple tasks can respond to the main thread. Write an example code for this scenario using the promis/future approach and explain the changes necessary to notify multiple reacting tasks.
5. C++11 introduced a template for atomic types `std::atomic<T>`. Solve the problem of task 4 by the following: All threads are busy-waiting on an atomic int, and the master sets the value to the number of threads that should wake up.

Solution

Listing 2 shows the solution for task 1:

Listing 2: Using conditional variables for synchronization

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

std::mutex mutex_;
std::condition_variable condVar;

void processData(){
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck);
    // do the processing
    std::cout << "Work done." << std::endl;
}

void readFile(){
    // read the input file
    std::cout << "Data is ready."  << std::endl;
    condVar.notify_all();
}
```

Solution task 2: Conditional variables are prone to spurious and lost wakeups. A spurious wakeups can happen, when the receiver finished its task before the sender has sent its notification. The receiver is susceptible for spurious wakeups. Therefore, the receiver wakes up, although no notification has been sent. A lost wakeups happens when the sender sends its notification before the receiver gets to a wait state. In this case the notification gets lost.

Solution task 3: Possible other approaches are the use of flags (atomic booleans), a combined approach of conditional variables and flags, the use of promis/future. See the following listings for code examples.

Listing 3: Code example using flags

```
#include <atomic>

std::atomic<bool> flag(false);

void processData(){
    while (!flag);     // busy waiting
    // do the processing
}

void readFile(){
    // read the input file
    if (condition is true)
      flag = true;
}
```

Listing 4: Code example using flags and conditional variables

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

std::mutex mutex_;
std::condition_variable condVar;

std::atomic<bool> dataReady;

void processData(){
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck, [&]{return dataReady;});
    // do the processing
    std::cout << "Work done." << std::endl;
}

void readFile(){
    // read the input file
    std::cout << "Data is ready."  << std::endl;
    dataReady=true;
    condVar.notify_all();
}
```

Listing 5: Code example using promis/future approach

```
std::promise<void> p;

void processData(){
    p.get_future().wait();
    // do the processing
}

void readFile(){
    // read the input file
    if (condition is true)
      p.set_value();
}
```

Listing 6 shows the solution for task 4:

Listing 6: Code example using promis/future approach

```
 std::promise<void> p;

void processData(){
    p.get_future().wait();
    // do the processing
}

void readFile(){
    // read the input file
    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 7 shows the solution for task 5:

Listing 7: Code example using promis/future approach

```
#include <atomic>
#include <thread>

std::atomic<int> foo (0);

void processData(){
    foo.wait(0);
    // do the processing
}

```

```
void readFile(){
    // read the input file
    if (condition is true){
      foo.store(5, std::memory_order_relaxed);

      std::vector<std::thread> vt;

      for (int i = 0; i < foo.load(std::memory_order_relaxed); ++i) {
        vt.at(i).join();
      }
}
```