Bern University
of Applied Sciences

414078-HS2018-0 - C++ Programming II

EXERCISE-05

1 Introduction

This exercise introduces the consumer/producer idioms with multiple threads. In particular, we will implement a clean and efficient solution using `condition_variable`. You will:

- repeat the consumer/producer idiom from the lecture.
- implement a thread safe data queue class with extended functionality.
- use `condition_variable`'s to signal and wait for specific events, *i.e.* thread synchronisation.

2 Submission

Submit your source code (as a zip-file) to Ilias **before the deadline** specified in Ilias.

3 Consumer/Producer Pattern

Re-implement the producer/consumer pattern discussed in the lecture with a small extension. In addition to let the consumer sleep while the queue is empty, provide a second `condition_variable` to let the producer sleep, when the queue is full!

- Provide a header-only implementation of a thread safe class `DataQueue` which takes the maximal buffer size (default = 1000 items), `i.e.` maximal number of buffer elements, as constructor parameter. Use template notation in order to support arbitrary data types. The goal is that you can use the `DataQueue` class in later projects.
- The users of `DataQueue` should not have to worry about race conditions and synchronisation at all! Therefore, provide a simple interface with the following functions:
  1. `add` - Add an element to the queue. The underlying implementation of `add` takes an item as parameter and should notify the consumer that items have been produced (*i.e.* queue is not empty) and produce only if the queue is not yet full (*i.e.* wait until items have been consumed).
  2. `get` - Get an element from the queue. The underlying implementation of `get` returns an item and should notify the producer that items have been consumed (*i.e.* queue is not full) and consume only if the queue is not empty (*i.e.* wait for items to be produced). Of course the consumer thread should finish when the item production is stopped.
  3. `isFull` - Checks whether the queue is full. Used by `add`
  4. `isEmpty` - Checks whether the queue is empty. Used by `get`
- Use `mutex`, `lock` and `condition_variable`'s to synchronise access to the elements.

In the main function we start a producer and consumer thread, let the production stop and wait for the consumers to finish as shown below:

```
int main()
{
    // Producer
    thread pThread{produce, 25};

    // Consumer
    thread cThread{consume};

    pThread.join();
    finished = true;
    cThread.join();

    cout << "Finished!"<< endl;
    return 0;
}
```

Find the complete test script in main.cpp. If `DataQueue` is correctly implemented the output is:

```
Producing -> item 0
item 0 -> comsumed
Producing -> item 1
item 1 -> comsumed
Producing -> item 2
...
...
-- Producer done --
item 20 -> comsumed
item 21 -> comsumed
item 22 -> comsumed
item 23 -> comsumed
item 24 -> comsumed
```

Play with different producer/consumer times, buffer size and number of elements.