CSE 437 HW1 – Yeşim Yalçın -200104004094

The Requirements

1-) The timer is build using C++11 and C++14 utilities. Therefore it needs a compiler that supports both.In my makefile I use g++ with std=c++14 flag.
2-) The timer is designed to allow more than one thread to register events at the same time. The event insertion order is ignored. The events are run according to their calculated next run times.
3-) The timer is designed to run the events as soons as possible when their run time have arrived. There might be a few milliseconds delays depending on CPU performance and the amount of registered events with the same run time.
According to the done example tests:

- **Running on Ubuntu with i7-9700 CPU:** 0 ms delay is observed for each event
- **Running on Ubuntu with worse CPU:** max 14 ms delay is observed

4-) The timer is designed in a way that if the object needs to die because it is the end of its scope etc, the main thread is signalled and the thread exists as well. **Meaning if there were still registered waiting events in the timer queue when the Timer object dies, those events will not run.**

The Design of the Timer

Before starting to implement the custom timer, **ITimer interface** is created. The interface can be found in **ITimer.h** file. Afterwards the **custom class Timer** is created, implementing the ITimer interface. Timer class declarations can be found in **Timer.h** file and the implementations can be found in **Timer.cpp**.

Timer class works this way:

- When a Timer object is created, inside the constructor the main thread that will be run to manage timing operations are created.
- This thread ownership is given to a custom created **ThreadGuard** object. This object is stored inside the Timer class with a **std::unique_ptr**. It is created using **std::make_unique**.
- ThreadGuard ensures that a thread can be owned by only one ThreadGuard and when this object dies, it automatically joins the thread if possible. Class implementation can be found in **ThreadGuard.h** file.
- When the thread starts running, it waits until there is a TimerEvent inside the eventList priority queue. **eventList priority queue** is a queue stored in the Timer class. It can store TimerEvents. **It orders them according to their nextRunTimes**. The TimerEvent with the closest nextRunTime is put at the front. The waiting is done using a **std::mutex queueMutex** and a **std::condition_variable cv.**
- registerTimer functions are used to add a TimerEvent into the queue. **Before adding, it locks the queueMutex and at the and notifies the cv.**
- If a TimerEvent is registered for the first time, the wait is released for the main thread and needed operations are done. From now on, the thread goes back to waiting on the cv. However, this time **cv.wait_for** function is used to wait. **The wait is done until the next run time of the TimeEvent which is at the front of the queue or if there is any new insertion into the queue.** If there is any new insertion and the queue order is changed, the wait time is recalculated.

- Both cv.wait and cv.wait_for functions also **check for the endThread flag. If it is true, the thread returns.**
- endThread flag is an **atomic flag** set by the destructor of the Timer so that if the object dies, the thread is acknowledged about it and it does not keep running forever.
- **To sum up the destruction is done this way:** Timer destructor called, flag is set, ThreadGuard destructor joins on the thread, the thread wakes up after being notifed and exits, ThreadGuard destruction completed, timer destruction completed.
- If the thread wakes up because it is time to run an event, **it pops and runs the event and if the event will need to be run later again, the event is reinserted into the queue with the new nextRunTime.** If it was the last time for the event to run, then it is not inserted again.

TimerEvent class is a custom class that stores information related to events that are run by the Timer. The declarations can be found in **TimerEvent.h** file and the implementation can be found in **TimerEvent.cpp**. TimerEvent objects are created and stored inside the Timer class.

TimerEvent class works this way:

- When an event is created, it's **nextRunTime is calculated** and stored according to the given information to the constructor.
- When it is time to run the event**, runEvent function of the class is called by Timer** class and this function calls the callback of the event. Afterwards calculates and the stores nextRunTime again. **Returns false to the Timer if the event should not be called again.**

Example Output from Example Test:

```
jade@DESKTOP-423HS14:/mnt/d/Yeni klasör/GTÜ/Year 4/RS/hw1$ make run
./hw1
[0] (cb -1): main starting.
[500] (cb 4): callback str
[500] (cb 5): callback str
[700] (cb 3): callback str
[1000] (cb 2): callback str
[1000] (cb 4): callback str
[1000] (cb 5): callback str
[1400] (cb 3): callback str
[1500] (cb 5): callback str
[2000] (cb 1): callback str
[2100] (cb 3): callback str
[2800] (cb 3): callback str
[3500] (cb 3): callback str
[4200] (cb 3): callback str
[4900] (cb 3): callback str
[5000] (cb -1): main terminating.
jade@DESKTOP-423HS14:/mnt/d/Yeni klasör/GTÜ/Year 4/RS/hw1$
```

How to Build

From command prompt, run **"make"** to compile. Afterwards, run **"make run"** to run hw1. Note that -pthread flag is given in the makefile because it was needed to run it in Ubuntu. However, no pthread is used. Only std::threads are used. In Windows, this flag might not be needed.