CSE 437 Real Time Systems Architecture HW1

1-Design

ITimer.h

I did not make any changes to the ITimer interface.

```
using CLOCK = std::chrono::high_resolution_clock;
using TTimerCallback = std::function<void()>;
using Millisecs = std::chrono::milliseconds;
using Timepoint = CLOCK::time_point;
using TPredicate = std::function<bool()>;
class ITimer {
    public:
        // run the callback once at time point tp.
        virtual void registerTimer(const Timepoint& tp, const TTimerCallback& cb) = 0;
        // run the callback periodically forever. The first call will be executed after the first period.
        virtual void registerTimer(const Millisecs& period, const TTimerCallback& cb) = 0;
        // Run the callback periodically until time point tp. The first call will be executed after the first period.
        virtual void registerTimer(const Timepoint& tp, const Millisecs& period, const TTimerCallback& cb) = 0;
        // Run the callback periodically. After calling the callback every time, call the predicate to check if the
        //termination criterion is satisfied. If the predicate returns false, stop calling the callback.
        virtual void registerTimer(const TPredicate& pred, const Millisecs& period, const TTimerCallback& cb) = 0;
};
```

Timer.h

```
class Timer : public ITimer {
    public:
        //constructor
        Timer();
        //destructor
        ~Timer();
        // run the callback once at time point tp.
        void registerTimer(const Timepoint& tp, const TTimerCallback& cb);
        // run the callback periodically forever. The first call will be executed after the first period.
        void registerTimer(const Millisecs& period, const TTimerCallback& cb);
        // Run the callback periodically until time point tp. The first call will be executed after the first period.
        void registerTimer(const Timepoint& tp, const Millisecs& period, const TTimerCallback& cb);
        // Run the callback periodically. After calling the callback every time, call the predicate to check if the
        //termination criterion is satisfied. If the predicate returns false, stop calling the callback.
        void registerTimer(const TPredicate& pred, const Millisecs& period, const TTimerCallback& cb);
        //Thread function
        void singleThreadFunction();
    private:
        // single thread
        std::thread single_thread;
        // priority queue for TimerEntries
        std::priority_queue<TimerEntry, std::vector<TimerEntry>, std::greater<TimerEntry>> queue;
        //mutex to wait
        std::mutex mutex;
        //condition variable to wait and wake up
        std::condition_variable cond_var;
        //flag to end program
        bool halt_program=false;
};
```

TimerEntry.h

```
// timer entry for each register timer is created
class TimerEntry {
    private:
        // when timer should work, updated on every period if any
        Timepoint timepoint;
        //for register type 2, end timepoint
        Timepoint endpoint;
        // what is the period of the timer
        Millisecs period;
        //what is termination predicate for this timer
        TPredicate predicate;
        //what code will work at this time
        TTimerCallback callback;
        //determine the register type
        //default = -1, single timepoint = 0, period forever = 1
        //period until timepoint = 2, period with a predicate = 3
        int registerType=-1;
public:
        // > operator overload for comparing two TimerEntries
        // priority queues priority depends on this operator
        bool operator> (const TimerEntry& other) const;
        // give access to Timer class
        friend class Timer;
        // run once, just callback and timepoint is given.
        TimerEntry(const Timepoint& tp, const TTimerCallback& cb);
        // run forever with a period.
        TimerEntry(const Millisecs& period, const TTimerCallback& cb);
        // run until given timepoint with period.
        TimerEntry(const Timepoint& tp, const Millisecs& period, const TTimerCallback& cb);
        // run periodically, after run check predicate.
        TimerEntry(const Millisecs& period,const TPredicate& pred,  const TTimerCallback& cb);
};
```

On TimerEntry.cpp implementation file, in 4 different constructors I get the necessary arguments and assign them to timepoint, endpoint, period, predicate, callback and registerType members. I overloaded the > operator to hold the TimerEntries in a priority queue. I used priority queue for TimerEntries to keep the TimerEntries sorted depend on their timepoints.

```
// operator overload
bool TimerEntry::operator> (const TimerEntry& other) const {
    return timepoint> other.timepoint;
}
```

In first constructor of TimerEntry that takes a timepoint and a callback, this means that this callback should run only once when timepoint is reached. So I don’t assign an endpoint and a period for that entry.

```
// run once, just callback and timepoint is given.
TimerEntry::TimerEntry(const Timepoint& tp, const TTimerCallback& cb) {
    this->timepoint=tp;
    this->callback=cb;
    this->registerType=0;
}
```

In second constructor of TimerEntry that takes a period and a callback, this means that this callback should run forever with a period, I don't assign any endpoint for that entry. I take the current time, sum with the given period, and assign to the timepoint member.

```
// run forever with a period.
TimerEntry::TimerEntry(const Millisecs& period, const TTimerCallback& cb) {
    auto current_time=CLOCK::now();
    this->timepoint=current_time+period;
    this->period=period;
    this->callback=cb;
    this->registerType=1;       You, 2 days ago • first mode not working …
}
```

In the third constructor of TimerEntry that takes a timepoint, period, and callback, this means that this callback should run until given point with given period. I assigned endpoint for that timer entry. I take the current time, sum with given period, and assign the result to the timepoint again.

```
// run until given timepoint with period.
TimerEntry::TimerEntry(const Timepoint& tp, const Millisecs& period, const TTimerCallback& cb) {
    auto current_time=CLOCK::now();
    this->timepoint=current_time+period;
    this->period=period;
    this->callback=cb;
    this->registerType=2;
    this->endpoint=tp;
}
```

In the last and fourth constructor of TimerEntry that takes period, predicate and callback, this means that this callback should run until given predicate returns false with the given period. I did not assign endpoint for that timer entry.

```
// run periodically, after run check predicate.
TimerEntry::TimerEntry(const Millisecs& period, const TPredicate& pred,  const TTimerCallback& cb) {
    auto current_time=CLOCK::now();
    this->predicate=pred;
    this->timepoint=current_time+period;
    this->callback=cb;
    this->period=period;
    this->registerType=3;
}
```

Timer.cpp implementation.

In the constructor, I just created the thread.

```
//constructor
Timer::Timer() {
    single_thread=std::thread(&Timer::singleThreadFunction,this);
}
```

In destructor, I get the lock to ensure that no deadlock is occurred, program is ready to end. And I set the program ending flag to the true. I notified to thread to wake up and exit from its while loop. And I wait for thread the end.

```
//destructor
Timer::~Timer() {
    {
        std::unique_lock<std::mutex> lock(mutex);
        halt_program= true;
    }
    cond_var.notify_all();
    if (single_thread.joinable()) {
        single_thread.join();
    }
}
```

In my registerTimer functions, I get the lock, I push TimerEntry to the queue and I notify the thread to wake up.

```
// run the callback once at time point tp.
void Timer::registerTimer(const Timepoint& tp, const TTimerCallback& cb) {
    std::unique_lock<std::mutex> lock(mutex);
    queue.push(TimerEntry(tp,cb));
    cond_var.notify_all();
}

// run the callback periodically forever. The first call will be executed after the first period.
void Timer::registerTimer(const Millisecs& period, const TTimerCallback& cb) {
    std::unique_lock<std::mutex> lock(mutex);
    queue.push(TimerEntry(period,cb));
    cond_var.notify_all();
}

// Run the callback periodically until time point tp. The first call will be executed after the first period.
void Timer::registerTimer(const Timepoint& tp, const Millisecs& period, const TTimerCallback& cb) {
    std::unique_lock<std::mutex> lock(mutex);
    queue.push(TimerEntry(tp,period,cb));
    cond_var.notify_all();
}

// Run the callback periodically. After calling the callback every time, call the predicate to check if the
//termination criterion is satisfied. If the predicate returns false, stop calling the callback.
void Timer::registerTimer(const TPredicate& pred, const Millisecs& period, const TTimerCallback& cb) {
    std::unique_lock<std::mutex >lock(mutex);
    queue.push(TimerEntry(period,pred,cb));
    cond_var.notify_all();
}
```

My thread function design:
First I get the unique lock for the mutex. Then thread goes in a while loop until halt_program is true. Then I check first queue is empty or not. If queue is empty, then thread waits on condition variable with a lock. If queue is not empty, then I get the current time and top element of the queue.

```
void Timer::singleThreadFunction() {
    std::cout<<"Timer::thread function starting..."<<endl;
    std::unique_lock<std::mutex> lock(mutex);
    while (!halt_program) {
        if (!queue.empty()) {
            //queue is not empty
            auto now = CLOCK::now();
            auto next = queue.top();
```

Then I check are we in any timepoint for that entry. If timepoint is reached, then I have a switch - case statement that looks for registerType.

```
            if(next.timepoint<=now) {
                //closest timepoint is reached
                switch (next.registerType)
                {
```

on case 0, I pop the element from queue, unlock the lock and call the callback and lock the lock again and exit from the switch case.

```
                case 0:
                    //timepoint reached, do the task
                    queue.pop();
                    lock.unlock();
                    next.callback();
                    lock.lock();
                    break;
```

on case 1, I pop the element from queue, unlock the lock and call the callback and registerTimer again so that this entry can run forever. Then I lock the lock and exit from switch case.

```
                case 1:
                    //run with a period until forever
                    queue.pop();
                    lock.unlock();
                    next.callback();
                    registerTimer(next.period,next.callback);
                    lock.lock();
                    break;
```

on case 2, I pop the element from queue, unlock the lock and call the callback and check if we reached the endpoint for that entry. If we did not reached the endpoint, then I registerTimer again and after if statement I lock the lock and exit from switch case.

```
                case 2:
                    //run with a period until tp
                    queue.pop();
                    lock.unlock();
                    next.callback();
                    if((next.period+now)<=next.endpoint) {
                        registerTimer(next.timepoint,next.period,next.callback);
                    }
                    lock.lock();
                    break;
```

on case 3, I pop the element from queue, unlock the lock and call the callback and check if the predicate returns true or not. If predicate returns true then I registerTimer again, then after if statement I lock the lock and exit from switch case.

```
            case 3:
                //run with a period until predicate is false
                queue.pop();
                lock.unlock();
                next.callback();
                if(next.predicate()) {
                    registerTimer(next.predicate,next.period,next.callback);
                }
                lock.lock();
                break;
```

on default case, I inform that it is unintended operation.

switch case statement ends here and we have else statement for if reached any timepoint or not for that next entry. If we did not reach any timepoint yet, we wait until the timepoint on a condition variable.

```
        else {
            //there is time for the next timepoint
            cond_var.wait_until(lock,next.timepoint);
        }
```

we checked queue is empty or not before, now we are in the else statement that indicates queue is empty. We wait on a lock so that somebody will wake up us and added a timer entry to the queue.

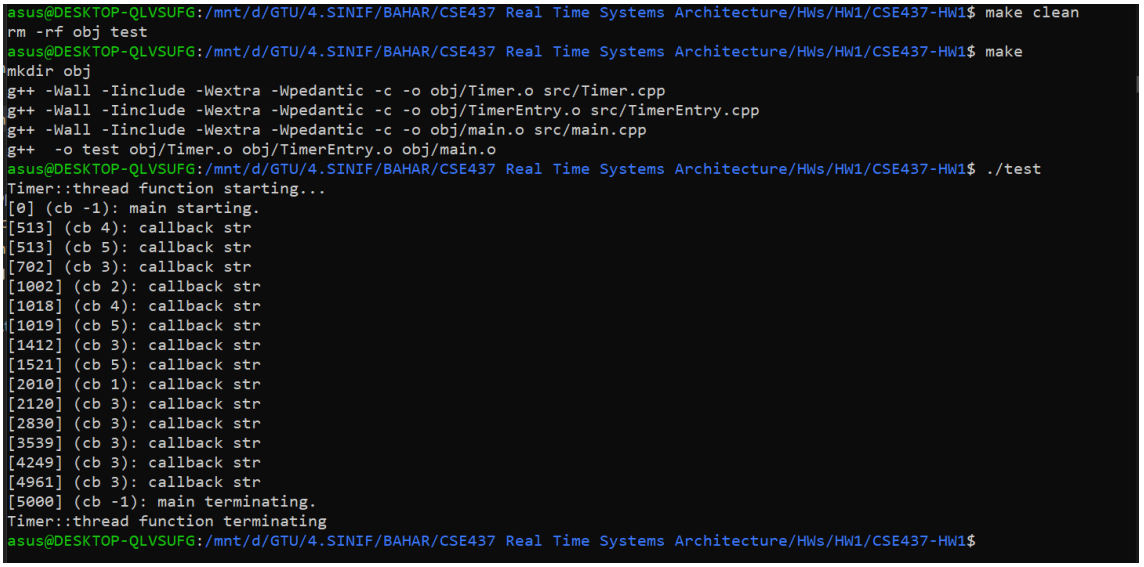
```
    else {
        //queue is empty, wait for the addition
        cond_var.wait(lock);
    }
```

thread function ends here.

Requirements of Timer
a thread, a priority queue with a comparator depends on timepoints of TimerEntries, a mutex, a condition variable, and a program end flag.

Requirements of TimerEntry
Timepoint, endpoint, period, predicate, callback and registerType, operator overload >, friend property with Timer class.

How to build
enter make to the console and run ./test



If you want to change test case, then make changes on main.cpp file