CSCM98 Coursework
Simulating the Scheduling Processes

Deadline: 25th of November, 11am

Coursework Submission:

To work on this assignment, you will have to download the visual studio solution available on the course website. Submission will be done on canvas by uploading the *scheduler.h* file only. C++ comments can be added to the file if needed. All code produced should therefore be located into the single file discussed above so that testing can be done uniformly. Please follow the submission guidelines carefully as any deviation from them will result in a loss of marks. A marking scheme will be available on canvas. Note that the work submitted must be your own work and you should not help or receive help from others. If you use extra material from somewhere else, please reference it through comments in the code.

Description Of Assignment

NB_TASKS tasks performing random operations need to be scheduled on *NB_PROCESSORS* processors. Work for these tasks is simulated mainly as a sleep operation internally, with the scheduler unaware of how long a task will take. Many functionalities related to processors and tasks are already implemented for you, but they should be handled as a black box, and only be accessed through an API described later. There is however nothing preventing you from reading the internal code. Please note *scheduler.h* is the file to modify, but pre-implemented functionalities are also available in *scheduler.cpp*. Usually, *.h* files are used for declarations while cpp files contain the definitions (e.g., implementation) of the member functions but we will put here all the new code in the *.h* file. Your role is therefore to ensure **that these tasks are scheduled correctly, and as well and as efficiently as possible.** See the marking scheme for more details about this.

Code Structure

The code contains three important classes:

Scheduler: The *scheduler* class will try to schedule tasks as best as possible. These tasks are already created and just need to be scheduled until completion. There are two functions you will need to complete:

- `ScheduleTasksUntilEnd`: This function will host the code that decides on both which tasks should be started and which processors should be used. Eventually it should make a call to the `LaunchTask` function of the *Processor* class.
- `NotifyEndScheduling`: This function is called by the processor thread when a task has been processed and processor use is being released. It is not necessary to implement code there to have a functional scheduler, but some features of your scheduler will probably need the presence of some code there.

While your code should use these two functions and not change their interface, there is also nothing preventing you to add additional code and data structure in the same file.

Processor: This class will simulate a working processor. This processor is represented by a thread that keeps looping to process tasks that have been passed by the scheduler to the processor. A call to `LaunchTask` will attach a task to a processor. Once a round of scheduling is completed, the thread will notify the scheduler of

completion by calling `NotifyEndScheduling`. The code given to you should be fully functional and you only have to call the scheduling function `LaunchTask` available in this class from the scheduler. Calls to `IsBusy` may also be required in your code.

Task: The basic class simulating a process behaviour. This simulation of work is simulated by sleep operations and some internal parameters. One round of scheduling will not usually be enough and therefore tasks will often forego several rounds of scheduling before being completed. It is possible to change the quantum/time slice duration when scheduling if needed. One can distinguish between three types of classes implemented that will behave differently:

- Simulation tasks, where work is carried out until completion. These tasks will rarely be carried out in just a single round of scheduling and will usually require multiple rounds to complete.
- Service tasks that will require many scheduling steps but will perform little each time.
- IOBound tasks that will frequently wait for an event. These tasks will require little CPU time and will return control quickly to the processor, but before doing this they will be put on a waiting state that will not return to a ready state before some time. Therefore, they should not be scheduled again before reaching a ready state.

You can also use other public function members of this class like `HasTerminated` or `IsReady` but see comments here after.

Finally, there is a *parameter.h* file which can be used to change some parameters of the simulation. Among other things, it allows the simulations to be repeated multiple times to estimate the performance of your scheduling algorithm, as well as estimating reliability. A different file may be used when marking, but you should make sure your code is compatible with this and other files.

The figure hereafter may help understanding the internal mechanisms of the code:

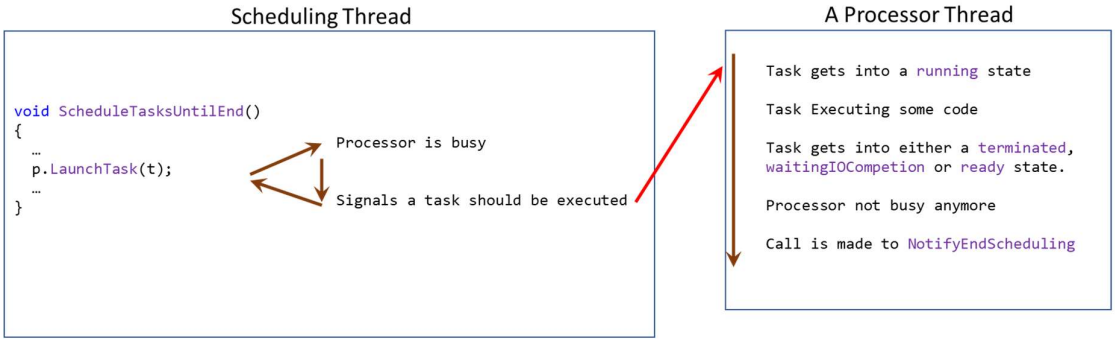

Figure: Overview of the internal sequence of events for the program

Some points and advice

- Try to understand the problem and then the code architecture before starting coding.
- Start simple and reduce the parameter values if needed.
- A task should only be scheduled if a process is not busy and a task is ready.
- The scheduling thread should loop until all tasks are completed.
- Functions like `HasTerminated` or `IsReady` are not thread safe (the value returned by the function may quickly become not representative of the state) and can only be used in some specific scenarios and/or given order.
- Use the (VS) debugger to find out about the state of your program when a bug occurs.

- Please complete the functions related to your name and Id. They will help with marking.
- Use the file(s) given. Most code should be written in the dedicated areas.
- Any partial solution may return only partial marks. The exact marks will depend on the quality of the answers and can only be evaluated upon reception of the coursework.
    - Specifications given need to be met as closely as possible.
- This is a strictly personal work. Any external help or code must be acknowledged and referenced as comments in the code.
- Code is tested and should compile and run under visual studio 2019 or above. You can either download it from the university (Windows) or use it in the labs. Code should also compile on other platforms but a makefile is not provided. Do not assume we will try to rectify issues with compilation, even though we may in some circumstances. We will only mark code provided but cannot accept extra solutions implemented in code.
- Assignment code and text may be updated if significant issues arise. If this is the case, you will be notified of the changes and issue.
- On a side note, unrelated to the exercise, these three tasks should have been implemented in inherited classes but are given in a single class to help with clarity.

Marking Scheme

A detailed marking breakdown will also be available under canvas. However, the following aspects will be marked. Notre that not meeting one criterion will not usually prevent you from getting other criteria rights.

Code compiling and all tasks completing accurately (30 marks)

Code should compile out of the box. Not compilable code may also affect other aspects of marking. All tasks should be scheduled correctly until they complete. There should not be any concurrency issues in the code.

Tasks are scheduled on all processors (20 marks)

More than one of the processors should be processing tasks.

Fairness and efficiency of scheduling (20 marks)

The scheduler schedules all tasks with fairness, distributing CPU resources to all active tasks, as opposed to giving preference to some. The whole approach should be relatively efficient.

Impact of the scheduling thread (10 marks)

The scheduler can be implemented with a while loop. However, this busy waiting may be inefficient as it will dedicate a full CPU resource to scheduling. Marks will be given there exists a good scheme saving some processor resources while scheduling. Comments to code can be added to explain the approach if needed.

Optimization of IO bounded tasks (10 marks)

Marks will be given if the scheduler implement optimizations for IO bound tasks that result is a faster scheduling.

Respecting guidelines (10 marks)

Marks will be given if aspects related to code submission and testing are fully respected (e.g., submitted file name and type, use of the API conventions, etc..). No extra effort needed to test code.