COEN 346

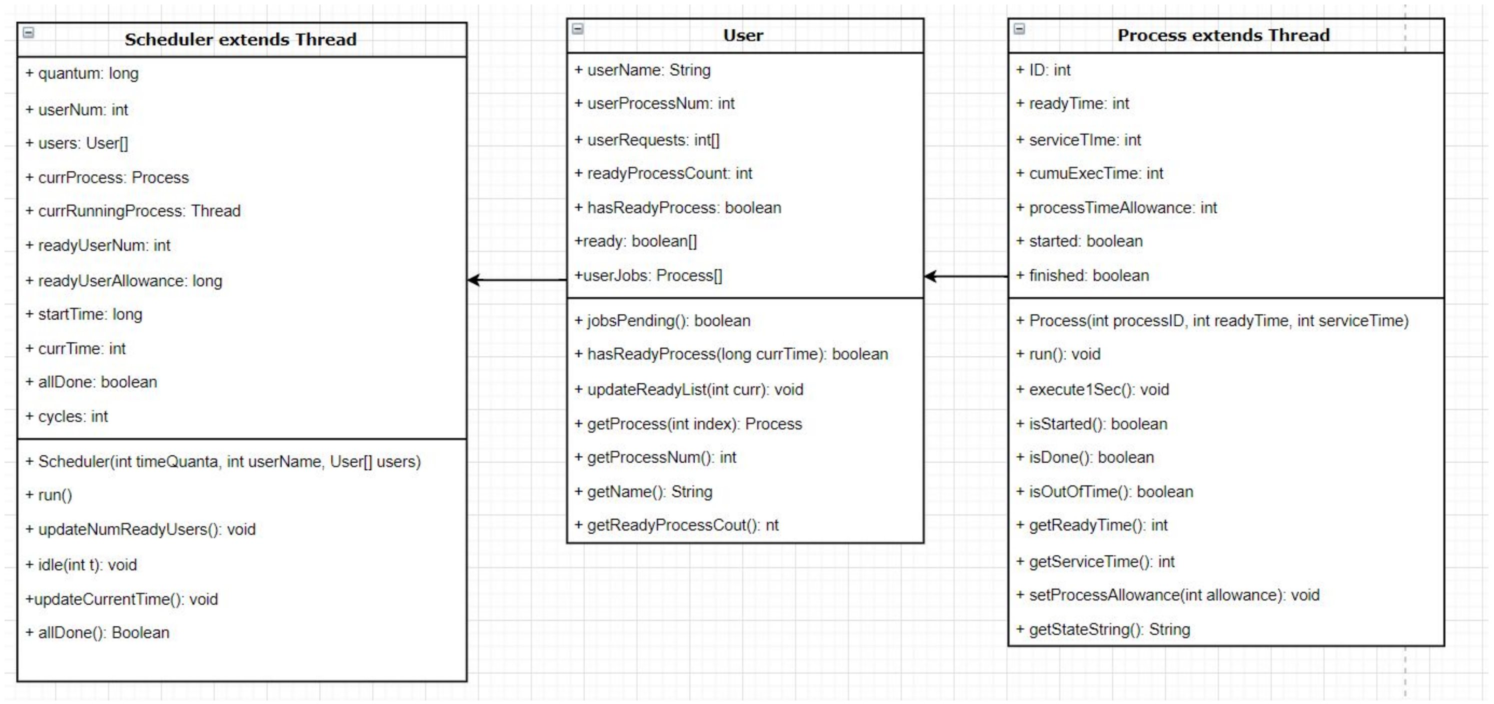
Group member: Marc-Andre Lauzier (alone)

Beginning from the *Scheduler* which initiates the scheduling cycles and performs the initialization of the *users[]* and their processes at startup. *Scheduler extends Thread* since it must run as a Thread. The run method is meant to cycle through the users and their processes until all the processes are finished, then exit.

First the run method updates the current time, *currTime* and it does this only at the beginning of the scheduling cycle. The value *currTime* is then used in *updateNumReadyUsers()* to figure out how many users [were] ready at *currTime*. It does this by feeding *currTime* to *User* class' *hasReadyProcess()* which updates its own *ready[]* boolean array and *readyProcessCount*.

Now, with the number of ready users, the scheduler can divide up the quantum appropriately, but only if there actually are users with ready processes. Next, the *run* method can iterate through the ready users to find out how many ready processes they have, via the *users[i].getReadyProcessCount()* method. The portion of the divided time quantum (*readyUserAllowance)* can then be segmented on a per user basis so that each user gets the same CPU time and it is divided equally amongst it's ready processes.

The synchronized block then starts the *currRunningProcess* Thread and 'waits' for it via a *join()*. Once all the processes of a user are done, the outer for-loop continues on to the next user, without updating the *currTIme* until the next scheduling cycle and re-evaluating the *numReadyUsers* and their respective ready processes at the updated *currTIme*.



Conclusion

The fair-share scheduling experience with Threads seemed unnecessarily complex and difficult. This is partially because I wanted to use actual time input and because of my inexperience programming with threads. Since my code never ran as I had intended (there were some thread exceptions that I could not fix) I did not get to set up an interrupt method where the scheduler would take back control from the processes, but I would have done this with a *sleep* method that terminated the thread. I relied on the process' to 'give up' the CPU when they were done via a *notify* call, which would be a very bad idea in practice. I attempted to make the scheduler as accurate as possible, and the general idea was there, but adding unnecessary complexity to the system made programming and debugging the program difficult.

The current implementation of round robin is more fair to the users than doing a round-robin independant of the users. Suppose one user has many more processes than the others, then he/she would take most of the CPU execution time and would not allow the other users to get their fair-share.