Concordia University
Engineering and Computer Science

LABORATORY ASSIGNMENT #2

Simulating Fair-Share Process Scheduling

COEN 346

Team members:

Ahmed Enani (26721281)
Josh Lafleur (40189389)
Eden Bouskila (40170349)

We certify that this submission is our original work and meets the Faculty's Expectations of Originality

Name: Ahmed Enani ID No: 26721281

Signature: **Date:** 11/11/2022

Name: Josh Lafleur ID No: 40189389

Signature: Joshua Lafleur **Date:** 11/11/2022

Name: Eden Bouskila ID No: 40170349

Signature: **Date:** 11/11/2022

1. HIGH-LEVEL DESCRIPTION

1.1 Structures

A. **thread_data_S**: Holds the current state of the process as well as the time the process will run for.
B. **proc_S**: Holds the information relevant to the OS for the running process. It stores the pthread, start time of the process, the thread data used as the argument to the function, the run time quantum of that specific process, and lastly if it is running or not.
C. **user_S:** Stores the name, processes, and the index into the array of active users for the OS of a given user.
D. **scheduler_S**: Stores the waiting users, the time quantum to be given to all users, and the current user and current process executing.

1.2 Methods

There are 4 application specific methods used in the code, as well as the utilization of 1 pthread library calls.

A. Application specific
    a. **void* scheduler (void*)**: This is the main scheduler of the program. It schedules all user processes a fair amount between all users, and a fair amount between all user process'.
    b. **void* func (void*)**: This is the simple user process. It simply counts every second until it is finished and kills itself.
    c. **void scdl(int)**: This is the timer handler used to handle the scheduler and to un-pause the scheduler so it can continue execution.
    d. **void thread_sig(int)**: This is used to handle signals to each thread to pause and unpause it's execution.
B. pthread library
    a. **pthread_create**: Creates a thread.

1.3 Threads

All user threads are functions of `void* func(void*)`. The `func` function is a rudimentary function just counts the seconds until it no longer executes. The `void* scheduler(void*)` is a thread which will dynamically schedule the executing process, and change process' when they can no longer run.

1.4 Program Flow

The main function reads in the Input.txt file and assigns the different waiting users and their processes from the relevant information. It then creates the scheduler thread. The scheduler then creates it's timer and signal handlers and starts the execution of the algorithm. THe algorithm checks if any processes from the waiting users are ready to be run. If the process is ready to be run, it adds the process to the relevant active user. If there is no active user for the waiting user, it creates one and stores it's index. Next, the scheduler removes the process from the waiting user and adds it to the active user. It does this for all waiting users. Then, the scheduler calculates the fair amount of time to be given to all processes and users, and follows by giving each of them that amount of time. The scheduler then starts/runs each of the processes in the respective order and pauses. At every second, the scheduler is signaled to either re-run the scheduler, or continue running through the

current round robin scheme that was calculated during the last scheduler session. Upon completion of all active and waiting processes, the program exits.

2. CONCLUSION

In conclusion, this assignment was much more difficult due to the complexities of how dynamic allocation works within C++ vectors and its impacts it had on the memory addresses passed to the child threads. A work around that ended up being necessary is the reservation of memory for the vectors so that the vectors were not moved in memory which ended up being a downfall of the initial implementation. For future improvements, it should be handled directly through malloc and free. It ended up working appropriately, however would need to be improved to scale past 32 users with 32 threads each. Emulation was achieved through setting the CPU affinity of each process to core 0. This is a rudimentary single core emulation method, and more detailed emulation can be achieved through the creation of a CPU struct including mutex's to prevent multiple simultaneous use. In our opinion, this is a more realistic implementation to single core execution as no two process' can execute at the same time from the start.

3. CONTRIBUTIONS

	Tasks	
Team Members	Code Implementation	Writing Report
Josh Lafleur	Threading and scheduling	Accuracy of document
Eden Bouskila	Signals and scheduling	Transferred information
Ahmed Enani	documentation and order	Formatting and organization