Programming Assignment #2: Simulating Fair-Share Process Scheduling

Class: COEN 346 Section Y
Submitted on 4 March 2025

Names & IDs:
- Clarence Zhen (40166293)
- David Makary (40189198)
- Justin Ma (40175944)

High-Level Description of the Assignment’s Objectives and Code

In this assignment, we are tasked to create a python script that emulates a scheduler when given a series of user processes and the time quantum. The scheduler must divide the time quantum depending on the number of users and the number of running processes for that user. Moreover, to simulate processes running synchronously or rather, according to the scheduler’s demands, this assignment will require multithreading as well. The series of processes and users is given through a text file `input.txt,` and the outcome of the scheduling is written in another text file called `output.txt`.

`main.py`

It contains miscellaneous functions such as the file parser function called `readInputFile` which decodes the input file and creates the parameters necessary to create processes. Moreover, it has a `Process` class that imitates a process but in reality has a `Thread` to mimic the behavior. This class has several properties that will help with the manipulation of the process, such as the name, status, service time, and the arrival time. These properties are especially helpful for the scheduler that is running in the main loop of the file.

The file also contains code for a scheduler that operates using a ready queue. Processes are sorted based on their arrival time and are popped from the list of processes when they are created. The algorithm works in cycles, which have a period equal to the time quantum specified at the top of the input file. At the start of every cycle, the first process in the ready queue is run, and the time allocation for the process is dynamic, based on the remaining number of processes and users left in the queue for that cycle. The scheduling algorithm divides the time in a given cycle by the number of users, and for each user, the time per process is further divided by the amount of processes for that given user. The algorithm employs a simulated CPU constant, which runs processes. The scheduler checks if the ready queue is not empty and if the CPU currently has no running processes, and pops the next process from the ready queue to run if those conditions are met. Periodically, the CPU will also check if the deadline for a process has arrived, in which case the CPU will stop the process and run the next one.

`threads.py`

This file contains an inherited thread class with additional properties and functions. It houses functions to “start” and “stop” a thread as well as internal properties to control its runtime. One thing to note is that when calling the start and stop functions, it only controls the decrement of the thread’s time counter. Therefore, in technical terms, the thread never really stops running until the time counter is 0.

Conclusion

The thread approach for simulating fair-share scheduling results in an accurate simulation to represent this method of process scheduling. The scheduling algorithm and threading classes provide an important platform to implement the scheduling algorithm addressed in the problem statement, which allows the comparison to be made with the round-robin scheduling algorithm. The main advantage of the given algorithm is that fairness is guaranteed amongst users. This prevents starvation from potentially happening for users who have one or few processes. The time quantum being distributed amongst users further ensures that processes in the ready queue will be run equally at the user level. In contrast, a major downside is that the given algorithm does not deal well in situations where there is a large discrepancy between users in the number of processes they own. For example, if user A were to own 50 processes with user B only owning 2. This leads to massive inefficiency in user A's processes, as each process has to fight for 25 times less CPU time than user B's processes. The algorithm further doesn't account for priority amongst processes. Critical processes are scheduled and run at the same priority level as normal processes, leading to potential bottlenecks. Lastly, the scheduling algorithm in the program produces a higher overhead, as the CPU time first needs to be allocated to the users before being allocated to the processes.

Contributions

Clarence Zhen (40166293)	`Thread` class
David Makary (40189198)	`readinputfile` functions and `Process` Class
Justin Ma (40175944)	Main execution loop