COEN 346

Lab Assignment #2 - Simulating Fair-Share Process Scheduling

Jose Ricardo Monegro Quezada - 40087821

Karl Noory - 40059592

Yevhen Haydar - 40024141

Introduction:

In this programming assignment, process scheduling was explored by implementing a Fair-Share process scheduling algorithm which utilizes threads to simulate the scheduler and the user processes. In order to work, the scheduling algorithm needs key inputs that are provided in an input file. Before starting the scheduling algorithm, we need to parse the input file to extract all of the key variables such as time quantum, users, number of processes for each user, arrival and service times of each process. After parsing the input file, the scheduler algorithm can start equally dividing the time quantum amongst the users. Once every user receives a portion of the time quantum, the scheduler will then split the user's time quantum equally for every process it has in the ready queue. Note the use of the word “equally”, each user is allocated an equal amount of time, and each process in the ready queue that belongs to the same user has an equal amount of CPU time to execute, hence the name “Fair-Share”.

Discussions:

1. Parsers

To run the program, the executable must be called with the path to the input.txt file as a CLI parameter, these parameters are stored in the argv[] vector along with the count of parameters in the argc integer. The program will then create a Parser object with the path to the input.txt as parameter, this step only prepares the parser but does not actually open and read the file yet. In order to extract the info from the file, the program calls the parse() function on the Parser object created in the previous step. Once the parse function completes execution all the necessary data will be stored in a Data struct which can be retrieved by calling the getData()

function on the Parser object. The Data struct contains the following: std::vector<User> *users*, uint32_t *timeQuantum*. User in this case is another struct that contains: std::string *name*, std::vector<Process> *processes*, uint32_t *processCount*. Process in this case is another struct that contains: uint32_t *arrivalTime*, uint32_t *serviceTime*. After parsing, the data from the file will be accessible to the scheduling algorithm.

2. Writer

The writer class is needed to facilitate the output of the program to the output.txt file. First, the program will set the output.txt file path to the same path as the input.txt using std::string manipulation tools. The logic behind determining the output.txt directory is designed to work on both Windows and Linux systems by looking for both forward slashes and backslashes. After determining the output file path, the Writer object is created and the openFile(outputFilePath) function is called on the Writer object. Once the output file is open, the scheduler can start using the fileOutput(msg) function call to tell the writer object to output a new line to the output.txt file with the specified message. In most cases, the output message will contain the following: std::string *userName*, int *pID*, output_action *action*. Example sample output line: Time 3, User B, Process 2, Paused. The *action* in the example is "Paused". To obtain the current time of execution to output, the writer object refers to unix time with a set of custom functions.

3. Scheduler

After parsing the file, an instance of the scheduler is created and given the *timeQuantum* obtained earlier as a parameter. The scheduler is implemented in a way where the timings are handled by both the user subclass and the scheduler itself. Each user is responsible for calculating its burst time for each process at the scheduler's request. At the same time, the scheduler is in charge of handling the appropriate distribution of quantums across all users. An abstract class was implemented as well, thread_controller, to handle thread creation, control and destruction. This is used as a parent class for the process and scheduler classes which use it to manage each of their own spawned threads.

Conclusion:

In conclusion, using threads to represent a Fair-Share scheduling technique is a good way to understand how such scheduling works. As programmers, we have a similar control over a thread, like an operating system has over a process. For this reason we can allocate an execution time to a thread and then pause it when needed.

Overall the fair-share scheduling technique is like applying Round-Robin first to users, where each user obtains an equal time quantum. Then applying Round-Robin to processes within each user's scope. Each user's own processes will have the same time quantum. The difference comes from the fact that if we compare processes' time quantums between different users, they won't be the same unless both users have the same number of processes.

A Fair-Share scheduling approach has both advantages and disadvantages. On one side, a user that has few processes will not be penalized by the fact that another user is trying to execute a lot of processes. Since we will separate the time quantum equally between users, regardless of how many processes they have (as long as they have at least one process in the ready queue). By contrast, the Round-Robin technique divides the time quantum equally between all processes.

If we look at this scheduling technique from a different perspective, a user that is barely active will have a lot of execution time given to its processes. While a user that needs a lot more process to be executed will have smaller execution time for each of its processes, this can be seen as a disadvantage.