Concordia University
Engineering and Computer Science

PROJECT REPORT
Operating Systems

Course:	COEN 346	**Lab Section:**	FL-X
Assignment No.:	2		

Name:	Ravjotdeep Kaur	**ID No. :**	40108180
Name:	Kevin Phan	**ID No. :**	40097439
Name:	Samuel Lopez-Ferrada	**ID No. :**	40112861

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

Signature: Ravjotdeep Kaur
Kevin Phan
Samuel Lopez-Ferrada

Date: 2022-11-11

Contribution by the group members

Group member	Tasks accomplished
Samuel Lopez-Ferrada	File Io, User Class, Scheduler Class
Kevin Phan	Callable Process, ProcessNode classes
Ravjotdeep Kaur	File IO, User Class, Scheduler Class

Introduction

This assignment focused on the implementation of simulating a process scheduler that is responsible for scheduling a given list of processes which belongs to a given list of users. The scheduler has to run concurrently with a clock and the processes it's managing. The type of scheduling used for this program will be fair-share scheduling. The main approach will be utilizing the Thread class, Callable Interface and the Linked List data structure to implement the process scheduler.

Code description

The following methods/functions/threads/data structures were used in this assignment. A diagram depicting the flow of the code is shown below along with a detailed description.

- ❖ Constructors:

```
➢ Process(int arrival_time, int burst_time)
```
➢ To create the process constructor where the name is given by incrementing a static value.

```
➢ ProcessNode(Process p, Character user_name)
```
➢ To retrieve the user's name with its process.

```
➢ Scheduler(int time_quantum)
```
➢ To take in a time quantum and initialize its corresponding data members.

```
➢ User(Character name)
```
➢ To initialize the user's data members.

- ❖ Functions:

```
➢ run()
```
➢ The run function in the Clock class's scope knows when to stop by waiting and incrementing the time by 1s.

```
➢ run()
```
➢ The run function in the Process class sleeps the thread for 1s and decrements its burst time by 1.

```
➢ run()
```
➢ The run function in the Scheduler class starts a new clock thread in a while loop to start the scheduling of the users and processes.

➢ `distributeTimeForEachProcessForEachUser()`

➢ To distribute the time for each process of each activer user through the user time distribution function from the User class.

➢ `setTimeQuantumForEachUser(int time_quantum)`

➢ To distribute the time to each user currently with a process arrived and needs more CPU time.

➢ `runUserProcesses()`

➢ To iterate through each User's process list according to its allocated CPU time and determine if the process is starting, resuming or finishing its execution depending on arrival and burst time. Also, it ignores any process with a burst time of 0 and ignores starving processes with no allocated time. Sleeps the Scheduler thread for the time the process needs to run

➢ `addUser(User u)`

➢ To add a user to the repository with an empty copy to the active user list.

➢ `checkForNewProcesses()`

➢ To use the transferArrivedProcesses() for each user.

➢ `printRepo()`

➢ To print out the library of processes for debug purposes.

➢ `printActiveUsers()`

➢ To print all currently active users.

➢ `removeEmptyUsers()`

➢ To remove any user in the repository if their list of processes is empty.

➢ `updateSizeOfActiveUsers()`

➢ To update the number of users currently with processes ready to run and needing allocated CPU time.

➢ `distributeTimeForEachProcess(int carry)`

➢ To distribute the allocated time in a fair-share type to each process. It increments each process allocated time by 1 and loops in a circular fashion to allow each process to receive its appropriate time.

➢ `checkIfFull()`

➢ To return true when all processes are saturated with cpu time, otherwise returns false if any process still requires more allocated CPU time.

```
➢ transferArrivedProcesses(User active_users)
```

➢ To iterate through the list and for each ProcessNode it checks if it has arrived by comparing the current time with arrival time.

```
➢ addProcessNode(ProcessNode node)
```

➢ To pass an existing ProcessNode to a user.

```
➢ isEmpty()
```

➢ Returns true or false depending if the list of processes of a user object is empty.

❖ Data Structures & Algorithms:

➢ Linked List

This data structure was used to create a library containing all the users. Each user also had a linked list containing all its processes.

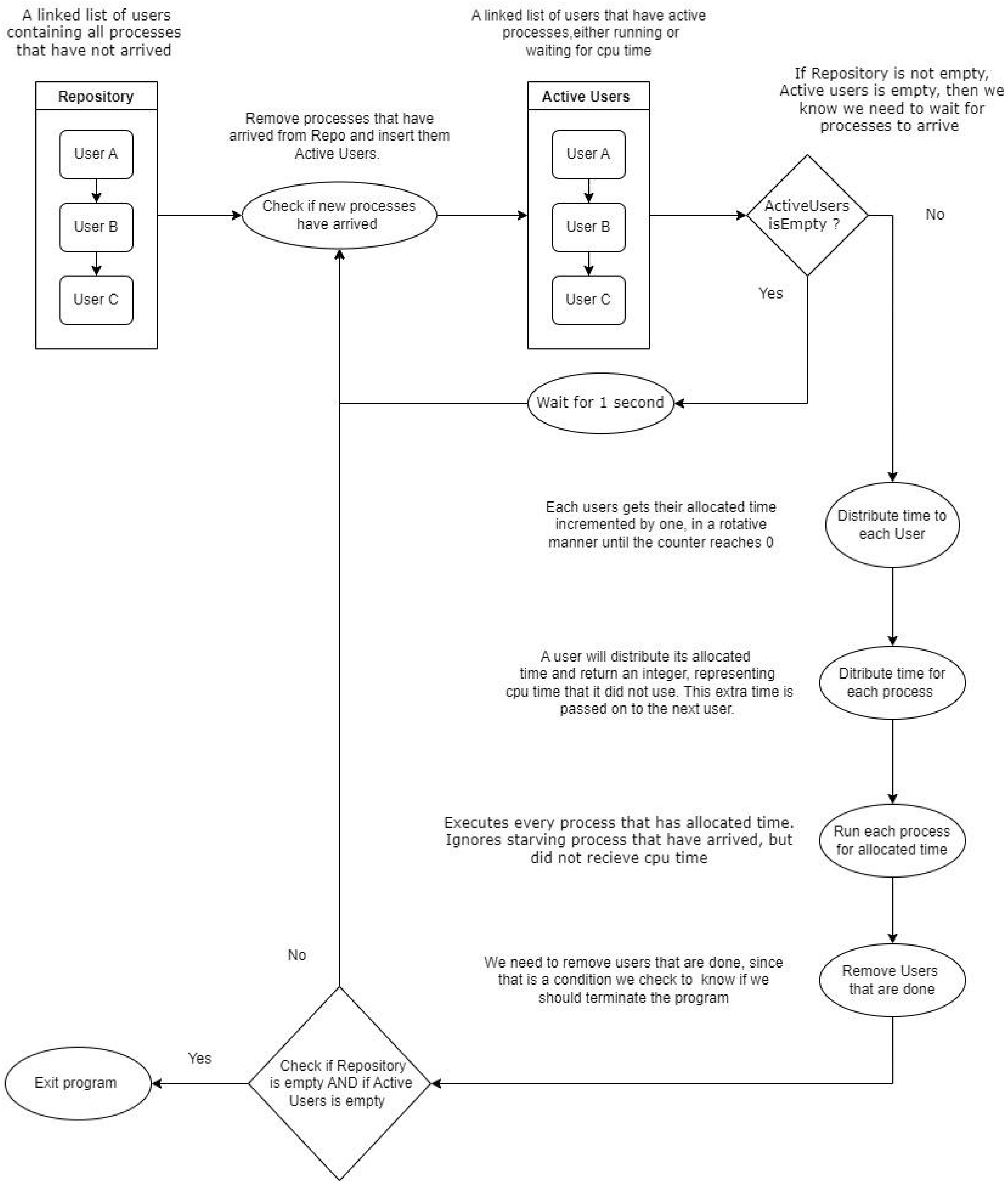
➢ LinkedList Rotation

To distribute time for each process (or users), we incremented each process's time by 1. If we reached the end of the list, we rotated back to the start. This was done with the modulo operator and a position rotating through the array. This covers every case, whether there is not enough time for all users or the case where each user gets more than 1 second.

➢ Library of Users & Active Users

The main concept of our implementation was to use a library of users, and a library of active users. Each scheduling cycle we update the both libraries. Time management is cascaded from the Scheduler to the Active Users. And each Active user takes care of its active processes. This helps reduce overhead by the Scheduler and allows it to delegate time management to each user. This also avoided constructing a Queue, since the queue would have to be built from scratch on each scheduling cycle. This implementation manages Active User library by removing any process with a burst time of 0.

Program flow


A linked list of users containing all processes that have not arrived
Repository
User A
User B
User C
Remove processes that have arrived from Repo and insert them Active Users.
Check if new processes have arrived
A linked list of users that have active processes,either running or waiting for cpu time
Active Users
User A
User B
User C
If Repository is not empty, Active users is empty, then we know we need to wait for processes to arrive
ActiveUsers isEmpty ?
No
Yes
Wait for 1 second
Each users gets their allocated time incremented by one, in a rotative manner until the counter reaches 0
Distribute time to each User
A user will distribute its allocated time and return an integer, representing cpu time that it did not use. This extra time is passed on to the next user.
Ditribute time for each process
Executes every process that has allocated time. Ignores starving process that have arrived, but did not recieve cpu time
Run each process for allocated time
We need to remove users that are done, since that is a condition we check to know if we should terminate the program
Remove Users that are done
No
Yes
Check if Repository is empty AND if Active Users is empty
Exit program

Conclusion and Discussion

Overall the program achieved the required specifications of the assignment. The Fair Share Scheduler (FSS) seemed quite simple theoretically, but it was not too long before it seemed like it required a high amount of data structure management and the complexity of the different operations appeared. It’s clear that minimizing the amount of operation and loops to schedule the next cycle can drastically reduce overhead. Thus, finding a solution with the minimal steps would be ideal, but supporting all the criterias of fairness was complex.

An unknown number of users, each with different numbers of processes, each with different burst times and arrival times creates a problem with many dynamic variables. Therefore, being able to keep the solution as abstract and flexible as possible was essential to cover almost every combination.

One of the main drawbacks of our FSS is that some processes could be starving for a long time since there is no aging or priority framework, and we can only distribute integer amounts of seconds on the cpu. The other drawback is the management of users and the time distribution requires various nested loops. This amount of overhead on a real time system could prove very inefficient. A Round Robin Scheduler could avoid such overhead and give each process a time quantum which would favor short processes, altho there would be no fairness between users.

Lastly, the experience with threads with the Java libraries was mostly positive, although communication and synchronization between the clock thread and the scheduler thread was confusing. This led to researching on monitors and thread pertaining to monitors. Since Java uses the monitor class to manage threads, a thread can communicate to other threads using the notify() and notifyAll() functions, and the wait() functions. This was avoided by having the clock thread inside the monitor for the Scheduler class(). Further understanding of monitors and synchronization could lead to more efficient code and thread management.