COEN 346 – Operating Systems
Team: Thomas Le (40096120), Yiwei Wen (27562446)

Programming Assignment #2: Simulating Round-Robin Process Scheduling

Overview

The program is given a list of processes, each on a new line. Each line is scanned for the two values that determine the process' arrival and burst time. The program creates a process object for each line and adds it to the list of processes. A thread is created that serves to determine which process is to be run next. Before beginning, this thread acquires a semaphore. Once that process has been determined, it is run, and the scheduler releases the semaphore. The process is run for a quantum, before releasing its semaphore back to the scheduler. Once all processes reach completion, an output text file is created with the information of the waiting time of each process.

List of Classes and Functions

- public class scheduler implements Runnable
    - run
- public class process implements Runnable
    - run

<u>public class scheduler implements Runnable</u>

The scheduler class contains our main method. Scheduler begins by initializing two arrayList objects, as well as a binary semaphore. In the main method, a scanner object is created and reads from an input text file. The data in the file is interpreted to create process objects, which are added to the first arrayList. Finally, a thread is created of the scheduler class and then .run(). This thread is put inside a synchronized loop and runs until the process arrayList is empty.

<u>scheduler.run()</u>

The run method in scheduler first acquires the binary semaphore. It then goes though the process list to determine which process has priority. Once the process has been determined, it calls .run() on the process, and releases the binary semaphore.

<u>public class process implements Runnable</u>

The process class simulates a process. The class contains several members that store the characteristics of the process (arrival time, process time, remaining time, etc.) The process class also has a static variable, elapsedTime, that keeps track of the total time passed. The constructor for the process class takes two integers and a semaphore. The first integer is the

arrival time; the second is the burst time. The quantum is calculated as 10% of the burst time, rounded down with a minimum value of one. The flags ‘started’ and ‘finished’ are initialized to false. The process’ name is determined by a second global integer that counts the number of processes created, and assigns the current count as the name.

process.run()

The run method in process first acquires the binary semaphore. It then checks to see if the process has been started before; if not, it will print a message saying so. The process is then progressed by a quantum. The values of remaining time and elapsed time are updated. If the process is complete, the flag for completion is set to true. The waiting time is also updated. It is calculated as the current time as of that moment, subtracted by the arrival and process time. A message is also printed stating that the process has finished. If the process is still in progress, the elapsed time is simply incremented by a quantum, and the semaphore is released.

Program Flow

1. Several global members are declared: processList, completeList, and hasCPU, a binary semaphore.
2. A text file containing process data is scanned by the program.
    a. if the input in invalid, it will throw an exception.
3. The data is interpreted and process objects are created, then added to processList.
    a. A process is initialized with an arrival time, a burst time, and a semaphore.
    b. The name of the process is determined by the order in which it arrives.
    c. The quantum of a process is calculated as 10% of the burst time, rounded down with a minimum value of 1.
4. A runnable object of the scheduler class is created.
5. The scheduler is run in a synchronized block while processList contains elements.
6. The scheduler acquires the binary semaphore and begins to assign the next process.
7. It first checks for processes that have started, with the lowest remaining processing time.
    a. The ‘next’ variable is assigned this process should it exist.
8. The scheduler then checks to see if any process has recently arrived. This is determined by the ‘started’ status being false, and the process’ arrival time being less than or equal to the current time.
    a. The ‘next’ variable is assigned this process should it exist.
9. If the ‘next’ variable is not empty, it will call .run() on it. If it is finished, it will be removed from the list of processes and added to completeList.
10. The semaphore is then released.
11. Once the process has .run() called on it, it will wait to acquire the semaphore.
12. The process will check to see if it has been started.
    a. If not, it will output a message stating so.
13. The process will increment the elapsed time, and remaining time by a quantum.
14. If the process has not completed, the current time will be incremented by a quantum.

    a. If the process has been completed, a correction is performed to ensure the correct amount of time is counted (in cases where a quantum is greater than the remaining time).
    b. The elapsed and remaining times are updated to maximum and zero respectively.
    c. The finished flag is set to true
    d. The waiting time is calculated based on the current time, minus the elapsed and arrival times.
15. Messages are printed to record the status of the process.
16. The process releases the semaphore.
17. The scheduler thread reacquires the semaphore and the process repeats until there are no more processes.
18. The waiting times of the processes are printed to an output file and the program terminates.

Conclusion

From this assignment we learned about mutual exclusion, the use of semaphores and monitors, and how to use synchronized methods in java. We were able to simulate a single CPU core though the use of binary semaphores and the synchronized keyword. This allowed us to run processes individually, including the process scheduler. This implementation could be useful for specific processes that need to be run on a single core, or for single core machines.