CS 3733: Operating Systems

Assignment 2: CPU Scheduling and Classical Schedulers (100 Points)

1 Objectives:

- Enhance the understanding of CPU scheduling;
- Learn to implement basic steps of classical schedulers;
- Practice and Improve programming and coding skills;

2 Description:

Overview: This assignment is related to process and CPU scheduling. Basically, you will implement the four classical scheduler functions, which take the parameters of two (2) processes for their CPU and IO bursts. Each scheduler function will generate the process scheduling strings of `R`, `r` and `w`, and calculate the corresponding waiting time and CPU utilization.

Details: Your main C program should take seven (7) integer values as command line parameters, and run as follows:

```
./assign2 q x1 y1 z1 x2 y2 z2
```

where `q` stands for the quantum value needed for Round-Robin scheduler; `(x1, y1, z1)` represents the first process that has `x1` units of CPU burst, followed by `y1` units of I/O burst, and then followed by `z1` units of CPU burst. Similarly, the second process is represented by `(x2, y2, z2)`. **Suppose that both processes arrive at time 0 where process 1 arrives in the ready queue just before Process 2**. No other processes are in the system.

The main program should start by checking that there are exactly 7 parameters. It will exit with an appropriate error message, otherwise. If the number of command line parameters is correct, the program will print your name on a line in the following form:
`CS 3424 Assignment 2 program was written by` *`YOUR NAME`*
followed by printing out the 7 input values on a single line.
Then, the main program should call the four (4) scheduler functions one-by-one. For each scheduler function, it will display 5 lines: a blank line serves as a separator, followed by a heading line to indicate the scheduler function, the two output strings (each string on its own line, with the second string directly under the first string), followed by four numeric values in one line. Here, the first two values are integers that indicate the waiting time for the first and second processes (i.e., the number of times `r` occurs in the first and second string), respectively. The third value is the average waiting time (i.e., average of the two integers) and the last one indicates the CPU utilization (i.e., the total number of `R`'s in the two strings divided by the length of the longer string).

The four scheduler functions (and any other needed functions) should be put in the file `pslibrary.c`. The prototypes of these functions should be in `pslibrary.h` and your main program `assign2.c` must include `pslibrary.h`.

Part 1: FCFS Scheduler: The first scheduler function is for First-Come First-Served (FCFS). It will ignore the first input parameter specifying a quantum and has the following prototype:

```
fcfs(char *s1, char *s2, int x1, int y1, int z1, int x2, int y2, int z2);
```

Here, the result process schedule strings will be returned through the first two parameters. The first process is represented by `(x1, y1, z1)` and the second by `(x2, y2, z2)`.

The scheduler function `fcfs()` needs to be implemented using a state machine design. A state machine has a state (the values of the variables) which changes over time. At each time step the state changes to a new state. The new state depends only on the old one and the input values. Here, the state of the system is the state of the two processes, including information about all of their remaining CPU and I/O bursts. The prototype code for the function can be obtained in the file `sm-proto.txt`. Insert the appropriate code so that it behaves like the `fcfs` algorithm. Note that, the protoype code handles some cases that are not related to FCFS, such as a quantum. You may delete the unncessary code.

Example: For the input parameters: `0 4 2 7 3 6 5`, the output from `fcfs()` should be:

```
scheduler fcfs:
RRRRwwrRRRRRRR
rrrrRRRwwwwwwrRRRRR
1 5 3.0 1.00000
```

Test the scheduler function `fcfs()` with the following inputs and manually check that it generates the correct output:

```
Test 1: 3 8 7 3 6 3 2
Test 2: 3 8 7 3 6 7 2
Test 3: 4 8 7 3 6 1 2
Test 4: 3 3 3 4 2 1 2
Test 5: 3 3 2 3 2 1 2
Test 6: 3 5 2 5 4 1 1
Test 7: 3 2 3 4 1 1 5
Test 8: 3 2 3 4 1 4 1
Test 9: 3 2 3 4 5 6 2
```

Include the output for the above test cases in your Report.txt.

Part 2: SJF Scheduler: Implement the **non-preemptive** SJF algorithm (by ignoring the first input parameter) with the following prototype:

```
sjf(char *s1, char *s2, int x1, int y1, int z1, int x2, int y2, int z2);
```

and add the implementation in `pslibrary.c`. Implement this by modifying your fcfs code or the prototype. Add the code to test this scheduler in `assign2.c` by calling this function. Test the scheduler with the test cases listed above.

Part 3: PSJF Scheduler: Implement the **preemptive-SJF** algorithm (by ignoring the first input parameter) with the following prototype:

```
psjf(char *s1, char *s2, int x1, int y1, int z1, int x2, int y2, int z2);
```

and add the implementation in `pslibrary.c`. Implement this by modifying your fcfs code or the prototype. Add the code to test this scheduler in `assign2.c` by calling this function. Test the scheduler with the test cases listed above.

Part 4: RR Scheduler: Implement the Round Robin algorithm with the following prototype:

```
rr(char *s1, char *s2, int q, int x1, int y1, int z1, int x2, int y2, int z2);
```

and add the implementation in `pslibrary.c`. Implement this by modifying your fcfs code or the prototype. Add the code to test this scheduler in `assign2.c` by calling this function. Test the scheduler with the test cases listed above. Note on breaking ties: Processes can enter the ready queue when they first arrive, when their quantum expires, or when their I/O completes. When the RR algorithm must choose the oldest process from the ready queue it needs a way to break ties. For our algorithms, we break a tie by having process 1 run before process 2. Make sure you understand this and make sure your solution works with the following input: 3 1 1 4 1 3 1 .

Part 5: Final Comprehensive Test: As a final test, get a copy of `runall.c` from Blackboard. Look at the source code and understand it. Compile it with your pslibrary and run the program. It should take less than a minute to complete and generate 4 files, each of about 40 megabytes and having the extension .all. Make sure you have no other files with this extension in your directory and execute:

```
ls -l *.all
wc *.all
md5sum *.all
```

Include the output from the above commands in your Report.txt. Each of the generated output files (*.all) should have $3,543,123$ lines and $3,543,125$ words. You should delete these large files after you write the output of `md5sum` on the program output part of `REPORT.txt`.

3 Submission Requirements

The assignment needs to be submitted on Blackboard. The submission includes two parts:

1. **Source code:** The source files: `assign2.c`, `pslibrary.c` and `pslibrary.h` (and any other source files you have). Your program needs to have proper comments (e.g., explanations for functions and variables etc.). It will be graded based on both functional correctness and clearity of the necessary comments.

2. **REPORT.txt**: In this **text** file, you need to report: a). List all of the people that you have collaborated with on this assignment. For each person indicate the level of collaboration (small, medium, large). Also write a few sentences describing what was discussed. Indicate whether you were mainly giving help or receiving help. b) the status of your program (completed or not; partial credit will be given even the program is not completed); c) Program output (as indicated in the description).

3. **Submission:** Please have all the above files zipped into a single file **a2-*abc123*.zip** and upload the file on Blackboard, where *abc123* should be your myUTSA ID.