

## CPSC 3220 Assignment 2 (50 points)

This assignment can be done individually or in teams of two. ONE submission per team with both teammate names on top in comments.

### Problem Statement

You will implement a simulation of two preemptive scheduling policies on a single CPU: Round Robin and Fair Mix, and compare the results to determine which of the two works better on your mix of jobs. Simulation parameters will be passed as pairs on the command line:

-t 2        -(any int) time slice, specified in seconds  
-i 7000    -(any int) starting task ids  
-o 1        -(any int) overhead of context switches, with the first context has to be included when the task runs for the first time.  
  
-f filename - output file that contains all output data

These pairs can be specified in any order, they set the parameters for your simulation. You will use a getopt function to parse the command line arguments. Simulation data will be read from a file via input redirection. (Examples below)

### Grading (rubric will be updated)

- ( 5 points) Correctly process CLAs using the getopt function, and correctly handle input redirection
- ( 8 points) RR implementation works correctly and efficiently and produces correct output: cpu, times and the queue
- ( 15 points) FairMix implementation works correctly and efficiently and produces correct output: cpu, times and the queue
- ( 10 points) Statistics table contains correct data and printed in ascending order of tids
- ( 4 points) Correct use of pointers/linked list
- ( 2 points) Process ids correctly generated
- ( 2 points) Submission in tar.gz format that contains source code files without subdirectories. No binaries should be submitted
- ( 2 points) makefile has all the required functionality
- ( 2 points) Formatting – number of comments, style, indentation, line length, etc.

**Note: any corrupt tarball submission or a submission that does not compile on SoC Linux machines will receive 0 points. You should have NO compile or runtime warnings. Please recompile after making the last moment changes to make sure your program still works.**

## Discussion

You are a programmer at a company that designs scheduling algorithms for the new Linux-like Operating system. You were tasked with implementing two preemptive scheduling algorithms for a system with one CPU.

Round Robin algorithm works as described in the textbook, with time slice specified via CLA. Each task will run for the specified time slice and will then be placed back onto the ready queue to wait for its next turn to run. Every context switch, including loading the first task onto the CPU will incur an overhead, also specified via CLA.

FairMix algorithm works the following way. It works similar to the RR in regards with time slices and overhead, but the order of tasks will be different. You will alternate running the largest task, then the smallest task, then the second largest and second smallest. This means you will always reevaluate the ready queue before running next task to select the largest, then the smallest, etc.

For example, if you have a list of tasks with service times 20, 12, 5, 7, 1, then you will run them in this order: 20, 1, 12, 5, 7. If the new tasks arrive during the execution, as soon as the currently running task finished its time slice, you will reevaluate the ready queue and again select the correct task that matches your criteria. Always select the largest task followed by the smallest task that had not yet run in the current rotation.

Another example, if you have the same task list 20, 12, 5, 7, 1 and you have executed tasks 20 and 1, and suddenly task with service time of 3 arrives, you will execute 12, and now 3, followed by 7 and 5 in their correct order. If you have executed 20 and 1, and task 16 arrives, then you will execute 16, 5, then 12 and 7. Always evaluate the queue before running the next task and select the largest task and then the smallest.

For each of the two scheduling policies you will calculate the average response time and the overhead of context switches. Remember that context switches will occur between tasks and before the first task is loaded onto the cpu, because you have to populate the cpu registers with the data that belongs to that task.

You will read groups of two numbers from an input file via input redirection. The first number on each line is the arrival time of the task, and the second number is its service time.

Input data will be provided via input redirection on the command line. Your program will assign task identifiers in the ascending order, in the order they are read from input file starting with tid passed as an option on the command line (described above and shown below). For example, if "-i 7000" is specified, then the first task is assigned an id of 7000, the next one 7001, etc. Each task will keep their assigned ID for both the RR and FM runs. Algorithms start running at the time of a 0 (zero). If time slice is smaller than the service time, then the task will run the service time.

Output will be written to both onto the screen and into the file specified on the CL.

Below is an example of a workload:

```
1 12
3 1
```

```
3 8
5 5
7 2
```

Your trace should first print a trace of each time unit, labeled as the discrete time value at the beginning of the time unit and showing what task is running on the CPU along with its remaining time.

You should stop the simulation after all tasks have completed. (Note that this means that the CPU is empty, the ready queue is empty, and there is no more input.)

After the simulation ends, your program should print a summary table, ordered by ascending task identifier and containing the tid, service time, completion time, response time, and wait time for each task, most of the values come from the input file, while some will need to be calculated. Finally, your program will print the average response time and total overhead time for both algorithms and make a determination of which algorithm performs better with the current mix of jobs.

### Language and filename Requirements

This program is written in C. You will use linked lists, with each task being represented by a *struct task* shown below. You can only access this struct via a pointer to that struct. You can name that pointer as you wish. Name your source file *sched.c*. Ensure it has your name on top of the file in comments.

```
struct task{
    int
        task_id,      /* specified on CL via -i x (x is any large integer)*/
        service_time,
        remaining_time,
        completion_time,
        response_time,
        wait_time;
    struct task *next;
};
```

Turn in your program via Canvas. Your submitted file is a *.tar.gz* archive with no subdirectories. Your code must run on the School of Computing machines to be graded.

### Guidelines

The entire code should be written by you, not generated, downloaded, etc. You are not allowed to consult anyone else except your teacher.

You should not send code to anyone or receive code from anyone, whether by email, Discord, social media, printed listings, photos, visual display on a computer/laptop/cell-phone/etc. screen, or *any* other method of communication.

Do not post the assignment, or a request for help, or your code on *any* web sites.

The key idea is that you shouldn't short-circuit the learning process for others once you know the answer. (And you shouldn't burden anyone else with inappropriate requests for code or "answers" and thus short-circuit your own learning process.)

### Example Expected Output

Let *in1* be an input file containing:

```
1 12
2 1
3 8
5 5
7 3
```

For the command line `./sched -i 2000 -o 1 -t 2 -f output.data < in1`, the output will be:

#### Round Robin scheduling results

time	cpu	serv time	remaining time	ready queue
0				
1	2000	12	11	
2	2000	11	10	2001-1
3	2001	1	0 (done)	2000-10, 2002-8
4	2002	8	7	2000-10, 2003-8
5	2002	7	6	2000-10, 2002-6, 2003-5
6	2003	5	4	2000-10, 2002-6
7	2003	4	3	2000-10, 2002-6, 2004-3
8	2004	3	2	2000-10, 2002-6, 2003-3
9	2004	2	1	2000-10, 2002-6, 2003-3
10	2000	10	9	2002-6, 2003-3, 2004-1
11	2000	9	8	2002-6, 2003-3, 2004-1
12	2002	6	5	2003-3, 2004-1, 2000-8
13	2002	5	4	2003-3, 2004-1, 2000-8
14	2003	3	2	2004-1, 2000-8, 2002-2
15	2003	2	1	2004-1, 2000-8, 2002-2
16	2004	1	0 (done)	2000-8, 2002-2, 2003-1
17	2000	8	7	2002-2, 2003-1
18	2000	7	6	2002-2, 2003-1
19	2002	2	1	2003-1, 2000-6
20	2002	1	0 (done)	2003-1, 2000-6
21	2003	1	0 (done)	2000-6
22	2000	6	5	
23	2000	5	4	
24	2000	4	3	
25	2000	3	2	
26	2000	2	1	
27	2000	1	0 (done)	

tid	service time	completion time	response time	wait time
2000	12	27	27	15
2001	1	3	2	1
2002	8	20	(your numbers here)	
2003	5	21		
2004	3	16		

average response time:  
overhead time:

1 12  
2 1  
3 8  
5 5  
7 3

#### Fair Mix scheduling results

time	cpu	serv time	remaining time	ready queue
0				
1	2000	12	11	(the only one, so run it)
2	2000	11	10	2001-1 (the only one on the queue, run)
3	2001	1	0 (done)	2000-10, 2002-8 (pick the largest)
4	2000	10	9	2002-8
5	2000	9	8	2002-8, 2003-5 (pick the smallest)
6	2003	5	4	2002-8, 2000-8
7	2003	4	3	2002-8, 2000-8, 2004-3 (pick the largest)
8	2002	8	7	2000-8, 2004-3, 2003-3
9	2002	7	6	2000-8, 2004-3, 2003-3 (pick the smallest)
10	2004	3	2	2000-8, 2003-3, 2002-6
11	2004	2	1	2000-8, 2003-3, 2002-6 (pick the largest)
12	2000	8	7	2003-3, 2002-6, 2004-1,
13	2000	7	6	2003-3, 2002-6, 2004-1 (pick the smallest)
14	2004	1	0 (done)	2003-3, 2002-6, 2000-6 (pick the largest)
15	2002	6	5	2003-3, 2000-6
16	2002	5	4	2003-3, 2000-6 (smallest)
17	2003	3	2	2000-6, 2002-4
18	2003	2	1	2000-6, 2002-4 (largest)
19	2000	6	5	2002-4, 2003-1
20	2000	5	4	2002-4, 2003-1 (smallest)
21	2003	1	0 (done)	2002-4, 2000-4 (largest)
22	2002	4	3	2000-4
23	2002	3	2	2000-4 (the only one, run)
24	2000	4	3	
25	2000	3	2	
26	2000	2	1	
27	2000	1	0 (done)	

tid	service time	completion time	response time	wait time
2000	12	27		
2001	1	3		
2002	8	23	(your numbers here)	
2003	5	21		
2004	3	14		

average response time:  
overhead:

\*\*\*\*\*

Best algorithm for this mix of tasks is: (RR or FM)

Average response time for RR:  
Average response time for FM:

NB: Depending on the set of tasks and how you select between queued tasks with the same remaining time – results may vary.

It is recommended that you use several input files with different number of processes to test your program. Sample input numbers were rather simple, it may be a good idea to test your program with an input similar to the one below.

```
0 10
2 2
2 14
2 16
3 1
3 11
5 8
5 7
6 12
```

Hand trace data from each simulation to verify the numbers in the table. It is your responsibility to determine the correctness of your output. Please refer to the chapter discussing the five times (arrival, service, wait, response and finish times).

Your makefile should contain targets *make* and *make clean*.

If you believe there is an error or a typo in the file, please report it to your teacher.