CS 111: Operating System Principles
Lab 2

You Spin Me Round Robin 3.0.0

Can Aygun, Alexandre Tiard, Vishnu Vardhan Bachupally
Derivative document by: Jonathan Eyolfson
April 22, 2022
Due: May 6, 2022 @ 11:59 PM PT

In this lab you'll be writing the implementation for round robin scheduling for a given workload and quantum length. You'll be given a basic skeleton that parses an input file and command line arguments. You're expected to understand how you would implement round robin if you were to implement it yourself in a kernel (which means doing it in C). The lab section in week 5 will give a quick introduction into how to use the C style linked lists (if you're unfamiliar) and the structure of the skeleton code.

Additional APIs. You may need a doubly linked list for your implement. For this lab you should use `TAILQ` from `sys/queue.h`. Use `man 3 tailq` to see all of the macros you can use. There's already a list created for you called `process_list` with a `TAILQ` entry name of `pointers`. You should not have to include any more headers or use any additional APIs, besides adding your code.

Starting the lab. Run the following command to get the skeleton for Lab 2: `git pull upstream main`. You should be able to run `make` in the `lab-02` directory to create a `rr` executable, and then `make clean` to remove all binary files. There is also an example `processes.txt` file in your lab directory. The `rr` executable takes a file path as the first argument, and a quantum length as the second. For example, you can run: `./rr processes.txt 3`.

Files to modify. You should only be modifying `rr.c` and `README.md` in the `lab-02` directory.

Your task. You should only add additional fields to `struct process` and add your code to `main` between the comments in the skeleton. You may add functions to call from main if you wish, but calls should only be between the comments. We assume a single time unit is the smallest atomic unit (we cannot split it even smaller). You should ensure your scheduler calculates the total waiting time and total response time to the variables `total_waiting_time` and `total_response_time`. The program then outputs the average waiting time and response time for you. Finally, fill in your `README.md` so that you could use your program without having to use this document.

Errors. All the allocations and input are handled for you, so there should be no need to handle any errors. You may assume integer overflows will not happen with a valid schedule.

Tips. You should ensure your implementation works with the examples that are attached to this document.

Example output. The `process.txt` file is based on these examples. You should be able run:

```
> ./rr processes.txt 3
Average waiting time: 7.00
Average response time: 2.75
```

Testing. There are a set of basic test cases given to you. We'll withhold more advanced tests which we'll use for grading. Part of programming is coming up with tests yourself. To run the provided test cases please run the following command in your lab directory:

```
    python -m unittest
```

Submission. Simply push your code using `git push origin main` (or simply `git push`) to submit it. *You need to create your own commits to push, you can use as many as you'd like.* You'll need to use the `git add` and `git commit` commands. You may push as many commits as you want, your latest commit that modifies the lab files counts as your submission. For late days we will look at the timestamp on our server. We will never use your commit times (or file access times) as proof of submission, only when you push your code to the course Git server.

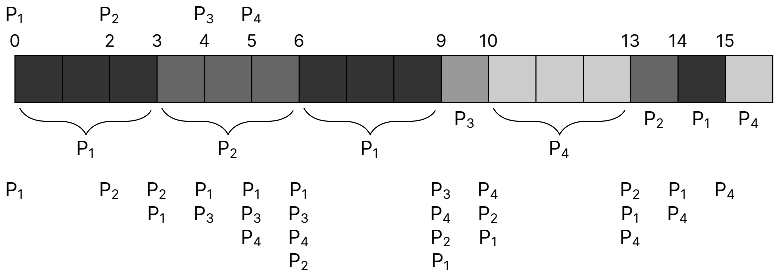
Please check `https://sefer.cs.ucla.edu/cs111/grades/` to see your status. You're expected to properly merge in upstream code without rebasing. Note that the website only updates your lab modification status if you've merged the latest code.

Grading. 100% - code implementation in `rr.c`, there will be no report grading for this lab.

RR with a Quantum Length of 3 Units

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

For RR, our schedule is (arrival on top, queue on bottom):



Metrics for RR (3 Unit Quantum Length)

Number of context switches: 7

Average waiting time: $\frac{8+8+5+7}{4} = 7$

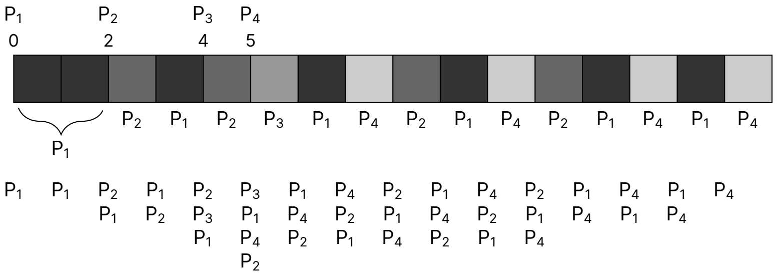
Average response time: $\frac{0+1+5+5}{4} = 2.75$

Note: on ties (a new process arrives while one is preempted), favor the new one

RR with a Quantum Length of 1 Units

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

For RR, our schedule is (arrival on top, queue on bottom):



Metrics for RR (1 Unit Quantum Length)

Number of context switches: 14

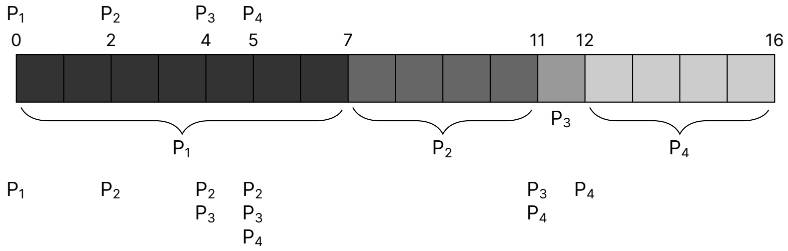
Average waiting time: $\frac{8+6+1+7}{4} = 5.5$

Average response time: $\frac{0+0+1+2}{4} = 0.75$

RR with a Quantum Length of 10 Units

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

For RR, our schedule is (arrival on top, queue on bottom):



Metrics for RR (10 Unit Quantum Length)

Number of context switches: 3

Average waiting time: $\frac{0+5+7+7}{4} = 4.75$

Average response time: $\frac{0+5+7+7}{4} = 4.75$

Note: in this case it’s the same as FCFS without preemptions