UCLA
Account
Dashboard
Courses
Calendar
Inbox
History
Search
Help

23F-COM SCI-111-LEC-1 › Assignments

Search this course
Immersive Reader

2023 Fall Quarter
Home
Announcements
Assignments
UCLA Media Reserves
Discussions
Grades
Pages
Syllabus
Modules
Collaborations
Google Drive
Zoom
Library Resources
Gradescope
UCLA Store Course Materials
Search
UCLA Course Reader Solutions
People
Media Gallery

# Spin me round robin

**Due: Sun Oct 29, 2023 11:59pm**

LATE 0 Possible Points

Attempt 1

Submitted on Oct 31, 2023 12:36am
**NEXT UP: Review Feedback**

Attempt 1 Score:
N/A

Add Comment

**Unlimited Attempts Allowed**
Available until Nov 5, 2023 10:55pm

**Details**

Lab 2. You spin me round robin

## Lab 2. You spin me round robin

[course home]

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 discussion section 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.

To characterize a round robin run you need to know the following things:

- The context-switch time. This is the amount of time it takes for the CPU to switch from one process to a another. In this lab, context switch times are always 1. (The unit of time is arbitrary; you could take the unit to be 1 μs, which is not too far off for context switch times in the Linux kernel.)
- For each process in the system, its arrival time and its burst time. The system starts at time 0, so arrival times are nonnegative integers. Each process consumes at least one unit of CPU time, so burst times are positive integers.
- The quantum. This is a positive integer. It is the amount of time that a process will be given before the kernel takes the CPU away from the currently running process and gives the CPU to the next process in the queue via a context switch. (If a process ends before its quantum is up, the kernel context-switches immediately to the next process.)
- The algorithm used to determine the quantum. It is described by a character string. It is either a string of ASCII decimal digits that denotes a positive integer, giving a fixed quantum size; or it is the string `median`, which means that each time you schedule a process the quantum should be the median of all the CPU times consumed so far by each currently running process. If the median is not a whole number, round to the nearest integer, breaking ties by rounding to even. If the resulting median is 0, treat it as if it were 1.

To keep things simple, this lab assumes just one CPU with one core.

## 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

Download the skeleton code . 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 (or `median`) as the second. For example, you can run: `./rr processes.txt 30` to use a fixed quantum length 30.

## 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 allocations and input are handled for you, so there should be no need to handle allocation or input errors. Use the type `long` to do internal calculations. You need not worry about integer overflows, as we won't give you schedules so large that the total wait time, total response time, or any process's finishing time does not fit into `long`

## Example output

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

```
$ ./rr processes.txt 30
Average waiting time: 7.00
Average response time: 2.75
```

although the exact output numbers may differ from this example.

## Testing

We will give you a set of basic tests. We'll withhold more advanced tests which we'll use for grading. Part of programming is coming up with tests yourself.

## Submission

1. All lab submissions will take place on BruinLearn. You will find submission links for all labs under the Assignments page.
2. Your submission should be a single tarball that includes only the following files: `rr.c` and `README.md`. Now the notes are collected at the end of each discussion. The tarball should be named `YOURUID-lab2-submission.tar`, where `YOURUID` is your 9 digit UID without any separators. Any submission that does not follow the submission guideline will receive `-15` points. (Note: if you make multiple submissions, Bruinlearn will automatically append the filename with `-SOMENUMBER`; you don't need to worry about this.)
3. Your submission will be graded solely based on your last submission to BruinLearn, without any exceptions. Please double check your submission to make sure you submit the correct version of your implementation.

## Grading

1. 90% code implementation
2. 7% documentation in `README.md`
3. 3% lab2 discussion notes (submitted at the end of each discussion)

Written by Can Aygün, Victor Zhang, Yadi Cao, and Paul Eggert. Derived from an earlier assignment by Jonathan Eyolfson. $Id: spin-me.html,v 1.1 2023/10/20 05:18:37 eggert Exp $

Preview Unavailable
305785051.tar

Download