Matthew Poncini
J.B. Speed School of Engineering
University of Louisville

15 July 2024

—

Project 2: Round Robin CPU Scheduling

Description of the Project and Background

The goal of this project is to simulate Round Robin (RR) CPU scheduling. For a given list of jobs, RR only process each job for a small amount of time. This designated time limit is referred to as a quantum. The scheduler goes through every job on the queue and then repeats itself until all jobs on the queue are completed. RR is a purely preemptive algorithm because it purposely interrupts each running job with the intention of resuming it later. After a process is interrupted, the RR scheduler saves the current state of the process, otherwise known as the context. The context of the process is put back on the queue and is loaded when it is it's turn to run again.

This project uses RR to schedule 1001 jobs stored on text file called (`job.txt`). Each job is represented by a tuple: (job number, requesting time, duration). The requesting time refers to the time that the job is added to the queue. For example if a job has a request time of 20, that means the job will be added to the queue 20ms since the CPU started.

Project Output Example

With 1001 job inputs and the scheduler working on each job for 5 milliseconds at a time, it is easy to lose track of what is going on. Here is an example of the code output with only 3 jobs. `Job.txt` was edited to the following:

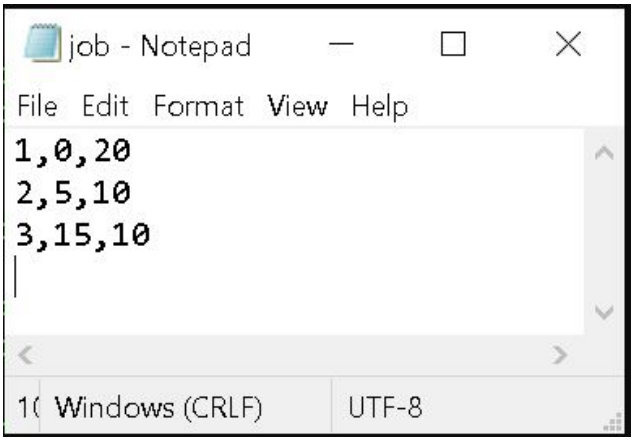

Figure 1: Job.txt with 3 Jobs

Job 1 is scheduled at 0 ms after the CPU begins running and has a duration of 20 ms. Job 2 is scheduled at 5 ms with a duration of 10 ms, and Job 3 is scheduled at 15 ms for a time duration of 10 ms. The next page shows the logic that the program follows. Jobs are only added to the queue when the run time of the program passes the Job's request time. Notice that Job 1 is processed twice before Job 3 is added. At the 5 ms mark and the 15 ms mark, there are options to what jobs should be processed. It could make sense for Job 1 to be processed again for a second consecutive time instead of Job 2 at the 5 ms mark. At the 15 ms mark, Job 2 is ran instead of Job 3 when either could have been selected. In my code, Tie Breakers are determined by a job's position in the input vector and the position of the current job that is being processed.

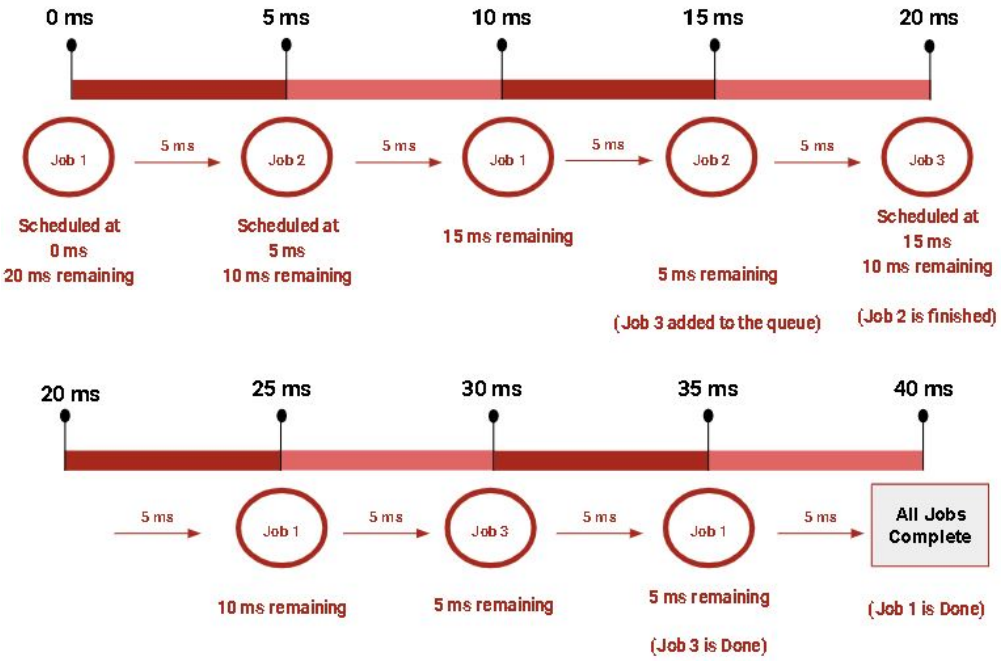

Figure 2: RR Logic

Below is the actual output of the RR Program with the jobs described above. Notice difference between Estimated run time, actual run time, and the process return time.

```
$>Job 1 scheduled for 5 ms
$>Job 2 scheduled for 5 ms
$>Job 1 scheduled for 5 ms
$>Job 2 scheduled for 5 ms, completed
$>Job 3 scheduled for 5 ms
$>Job 1 scheduled for 5 ms
$>Job 3 scheduled for 5 ms, completed
$>Job 1 scheduled for 5 ms, completed


Estimated run time: 0.04 seconds
Jobs completed: 3
Actual Run Time: 0.041 seconds
Average wait time: 0.026 seconds

--------------------------------
Process exited after 0.1151 seconds with return value 0
Press any key to continue . . .
```

Figure 3: Output of 3 Jobs

Round Robin Function Code Logic

The next page shows the definition for the `round_robin()` function used for the program. The `round_robin()` function takes in a vector filled with vectors. The sub-vectors represent each individual job's name, request time, and duration while the outer vector represents the 1001 jobs to be processed. This vector is created by the `get_jobs_from_file()` function defined earlier in the program.

Most of the variables being used are initialized at the beginning of the function. The `float time` variable is used to make sure that the only jobs being processed are ones with request times that the `time` variable has surpassed. The `time` variable is incremented by the `run_time` of each job. This value is usually equal to the `const quantum` variable (set to 5 ms). If a job can be completed in less than 5 ms, then `run_time` will be set equal to that remaining time. In cases where not all jobs are complete but no jobs are present on the queue, `time` will be incremented by 1 ms. The `time` variable is also used to stall the function so that it actually waits the full `run_time` of a job (or 1 ms if there are no jobs on the queue). A `while` loop is utilized to achieve this.

round_robin() Timer

```
auto start_time = steady_clock::now();
// function logic, time variable is incremented
while (duration_cast<milliseconds>(steady_clock::now()
        - start_time).count() < time) {
    // Simulated Job Runs
}
```

The logic of the function is primarily contained in the following `while` loop:

```
while (jobs_complete_count < num_of_jobs ) {
    // function logic
}
```

The `jobs_complete_count` simply keeps track of how many jobs are completed. It starts at 0 and is only incremented when a job is finished. The `num_of_jobs` variable is set equal to the number of jobs in the input vector. Inside of this `while` loop, there is a `for` loop that iterates over the inputted vector. It scans for the jobs who's request time (`input_job_list[i][1]`) is less than or equal to the current value of the `time` variable. The jobs that satisfy this requirement are considered to be "on the queue" and are eligible to be processed when `for` loop reaches them. Each time a job is processed, the duration of the job is updated and the `time` variable is incremented:

```
input_job_list[i][2] = input_job_list[i][2] - run_time;
time = time + run_time;
```

If the duration time of a job is equal to 0, that means the job is completed and needs to be taken off the queue. An `if` function determines if this is the case for a job. When a job finishes, the `jobs_complete_count` goes up by 1 and the `erase()` function is used to remove that job from the inputted vector.

Round Robin Function Code

```
const int quantum = 5;

void round_robin(vector<vector<int>> input_job_list) {
    float time = 0;
    int jobs_complete_count = 0;
    int run_time;
    int num_of_jobs = input_job_list.size();
    float wait_time = 0;
    float estimated_run_time = 0;

    for (int i=0; i < input_job_list.size(); i++) {
        estimated_run_time += input_job_list[i][2];
    }

    auto start_time = steady_clock::now();

    while (jobs_complete_count < num_of_jobs ) {
        for (int i = 0; i < input_job_list.size(); i++) {
            if (time >= input_job_list[i][1] ) {
                run_time = min(input_job_list[i][2], quantum);
                cout << "$>Job " << input_job_list[i][0] << " scheduled for "
                        << run_time << " ms";
                input_job_list[i][2] = input_job_list[i][2] - run_time;
                time = time + run_time;
                if (input_job_list[i][2] == 0 ) {
                    jobs_complete_count++;
                    cout << ", completed" << endl;
                    wait_time = wait_time + time - input_job_list[i][1] - input_job_list[i][2];
                    input_job_list.erase(input_job_list.begin()+i);
                    i--;
                }
                else    {
                    cout << endl;
                }
            }
            else    {
                time += 1;
            }
            while (duration_cast<milliseconds>(steady_clock::now()
                        - start_time).count() < time) {
                // Simulated Job Runs
            }
        }
    }
    float actual_time = duration_cast<milliseconds>(steady_clock::now() - start_time).count();
    cout << "\n\nEstimated run time: " << estimated_run_time / 1000 << " seconds" << endl;
    cout << "Jobs completed: " << jobs_complete_count << endl;
    cout << "Actual Run Time: " << actual_time / 1000 << " seconds" << endl;
    cout << "Average wait time: " << wait_time / jobs_complete_count / 1000
                    << " seconds" << endl;
}
```

The end of the function displays a summary of the program. It displays an estimated run time for the function. This value is simply the summation of each duration in the inputted job list. The function displays the number of jobs completed. The function then uses the timer described earlier to determine the actual run time of the program. This value is usually always slightly longer than the estimated time due to the small amount of time the CPU processes after the `float actual_time` variable is defined. Lastly, the average wait time is displayed. This value is found by using the `wait_time` variable. This variable represents the total wait time for all jobs on the input vector. After a job completes, the it's wait time is

calculated by finding the amount of time it was on the queue and not being processed. That value is then added to the total wait time.

Program Output

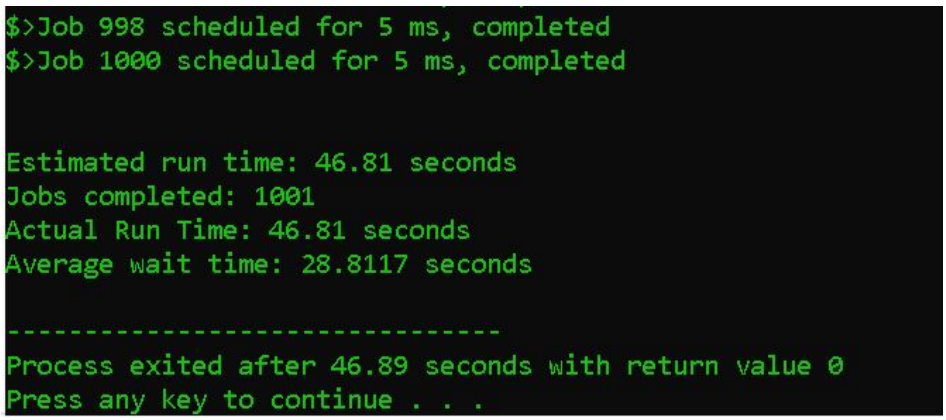

Figure 4: Original Job.txt File Output

Analysis

Time Complexity

The time complexity of the `round_robin()` function is $O(n^2 * m)$ where n is the number of jobs in the input vector and m is the average number of times a job processed before completing. The complexity of the function is nested in the central `while` loop. This loop contains a `for` that is size $O(n)$ and an `erase()` function that is also sized $O(n)$. The `while` loop continues until all of the jobs are completed, which depends on the job duration for each job in the input list. This multiplies the time complexity by the average number of times each job is processed. Therefore, the time complexity of the `round_robin()` function is $O(n^2 * m)$.

The most obvious way to decrease time complexity is to avoid using the `erase` function. This can be done by adding a fourth placeholder in the sub-vector that held job name, request time and duration. This fourth place holder could store state: 0 representing awaiting queue, 1 representing in the queue, and 2 representing completion. This would be a simple way to make the function $O(n * m)$ instead of $O(n^2 * m)$.

Quantum Affecting Wait Time

To calculate the job's wait time, the equation looked the time the job completed and subtracted it's arrival time and it's duration. So, how does the quantum value effect the average wait time? Outside of the simulated RR scheduling in this project, real RR scheduling spends CPU time switching between processes. Shorter quantum values lead to a greater number of context switches, which incurs overhead. This can cause time deficiencies since more time is spent on context switching than actually processing each job. When the quantum value is higher, less time is spent on context switching. However, larger quantum values increase the response time. Response time refers to the time that the CPU initially processes a job. If larger jobs are earlier in the queue, the average wait time can become very large with larger

quantum values.

In this project, the quantum value was set from 1 to 10 and once at 50. 100 jobs were processed. The table below shows the average weight times calculated. The data does show a trend of wait time decreasing as the quantum value increases. There does seem to be an anomaly at the 5 ms and 10 ms quantum values. This can be attributed to the job durations in the `Job.txt` file being multiples of 5.

Quantum	Average Wait Time
1 ms	2.79264 sec
2 ms	2.79189 sec
3 ms	2.79969 sec
4 ms	2.80619 sec
5 ms	2.7188 sec
6 ms	2.80495 sec
7 ms	2.81316 ms
8 ms	2.82311 sec
9 ms	2.83988 sec
10 ms	2.70955 sec
50 ms	2.63175 sec

Submitted by Matthew Poncini on 15 July 2024.