CPU Scheduler : Report

Abishek A
22115005

June 19, 2024

Contents

1 Introduction

As modern computer systems continue to evolve, optimizing CPU scheduling algorithms becomes increasingly important to ensure effective utilization and fair use of resources. This project aims to study and benchmark various CPU scheduling algorithms ideas in order to gain a better understanding of them and the tradeoffs they present.
It presents a web-based interface for the user to define their workload and compare and visualise the results of different algorithms through the use of interactive charts and metrics.

2 Architecture

A conscious effort has been made to keep the components of the project as composable as possible. Future improvements, such as more scheduling algorithms, visualization based on other metrics, etc can be facilitated easily due to this quality of the project.

2.1 Tech Stack

Here is an abstracted view of the technologies used to build the project.

- **C++** has been used to write the library of scheduling algorithms, used to perform simulations and emit calculated metrics. It was chosen for its unparalleled speed and robust standard library.
- **Express**, a web framework for rolling up lightweight web servers quick. It was used to create the small web server that responds to frontend API calls and runs the compiled binaries to fetch results.
- **React** was used to make the simple frontend. It was chosen in order to use **Material UI**, lending the frontend a clean, professional look.
- **Recharts** was used to visualize the simulation data dynamically.

2.2 User Flow

The complete flow that the application goes through for a simulation is as follows:

- The user inputs the data for the jobs on the web interface. They can choose whether to simulate all algorithms for the workload and get a performance comparison, or a single algorithm and view its Gantt chart.
- The request is sent from the frontend to the web server. The server parses the request and figures out which binary (there are separate binaries for bulk and individual simulation) to run.
- The server populates the input file with the process data and executes the correct binary with proper arguments based on the request.
- The library simulates the algorithm for the process data and emits the results to the appropriate output file.
- The server reads this output file and sends back the performance data to the frontend, where it is parsed and visualized.

2.3 Components

The project can chiefly be split into three main components - the frontend, backend and library.

2.3.1 Frontend

React, Material UI and Recharts was used to build the frontend.

- **User Interface** : It provides a clean user interface for the user to input their job details, choose their scheduling algorithm and time quantum and choose the mode of simulation.
- **Data Visualization** : It visualizes the simulation results dynamically and responsively using Recharts components.

2.3.2 Backend

Node and Express were used to make the minimal backend. The server simply listens for API calls, and parses the request body to get the process info, bulk/individual scheduling request, and executes the compiled binaries. It further retrieves the results and sends it back.

2.3.3 Library

The actual meat and bones of the project written in C++. The code itself is healthily commented, so I will refrain from including code snippet screenshots so as to limit the size and increase the readability of this document. Here is a flow for simulation for any single algorithm -

- A `proc` struct is used to encapsulate the info related to any particular process/job.
- The `simulate` function first looks for the process data in an input file, in our case `process_info.txt`. It processes this using `get_processes()` and returns a list of `proc`s.
- The specific code for each algorithm is ran (the code and implementation of each is implemented in the next section).
- `calculate_metrics()` is called to return a `metrics` struct containing all the main performance metrics. The completed process list is passed as an argument to this function. The metrics are written to output files using `write_metrics_to_output_file()` and `write_gantt_data()`. These functions are defined in the `utils.h` header file.

The library allows two types of simulatons i.e src files -

- `scheduler_interface.cpp`, which simulates for individual algorithms and writes data to two output files - `gantt_data.txt` and `outfile.txt`. The data is self explanatory - one provides broad metrics while the other provides specific start and completion times for each process - which is used to generate the Gantt chart on the frontend. Its binary, say `scheduler` expects to be called like `./scheduler -pol <policy_name>`. An additional time slice argument is expected for fair share scheduling algorithms.
- `bulk_metrics.cpp` calls `simulate()` for all(except lottery) algorithms and writes the data to `bulk_metrics.txt`. Its binary, say `bulk_metrics` is expected to be called like `./bulk_metrics -sq <scheduling_quantum>`.

The idea behind splitting the scheduling types into these two categories for ease of hooking it up with the server - if I had made separate binaries for each algorithm, the number of API routes on the server would have unnecessarily increased.

3 Definitions, Metrics and Algorithms Implemented

This portion is a more theoretical examination of the algorithms along with the key snippets related to their implementation.

3.1 Definitions

Here, we define some of the parameters and terms that appear constantly throughout the codebase.

- **Arrival Time**: The point in time at which a process becomes available to be scheduled.
- **Burst Time** : The amount of time taken by a process to run to completion.
- **Scheduling Quantum** : Alternately referred to as **time slice** within the codebase - it is the time period for which a process is run before context switching in fair-share algorithms.
- **Firstrun Time** : The point in time at which a process is first scheduled to run.
- **Completion Time** : The point in time at which a process completes running.
- **Preemptive** : An algorithm is preemptive if it can context switch from one running process to another. It "preempts" the running process and runs the other one.
- **Tickets** : A form of priority allotment, used in the lottery scheduling algorithm. In simple terms, the more tickets a process possesses, the more CPU time it should ideally get with respect to other processes.

3.2 Main Performance Metrics

Two main performance metrics are used to evaluate a scheduling algorithm - efficiency and fairness. In fact, most scheduling algorithms optimize one of these metrics by trading off the other. These metrics are evaluated using the following quantities:

- **Response Time** : Used to measure fairness. Fairness, also known as interactivity, is a measure of how interactive the system is with respect to the different processes being scheduled on it. A user would not be pleased if their process ran very late in a bid to maximise efficiency. It is defined as the different between time of first run and time of arrival. Minimizing response time increases fairness.

$$T_{response} = T_{firstrun} - T_{arrival}$$

- **Turnaround Time** : Used to measure efficiency. A straightforward metric, it aims to minimize overall completion time of all the processes.

$$T_{turnaround} = T_{completion} - T_{arrival}$$

We analyze and are concerned with the average values of these metrics.

3.3 Algorithms Implemented

Here, we discuss the details of implementation of each algorithm and their tradeoffs and performance with regards to the above metrics. For the sake of readability, the important differentiating points of each algorithm are what are included, the actual algorithms are again, quite well commented, here I aim to show the key points of each one.

3.3.1 First Come First Serve (FCFS)

The simplest scheduling algorithm of them all. Simply schedules processes in the order that they arrive, no preemption. As expected, it performs poorly on both metrics for most workloads, due to the *convoy effect*, where smaller more efficient processes may be blocked waiting for a large one. Implemented using a simple sort based on arrival time and sequential scheduling.

```
// sort processes by arrival time
std::sort(processes.begin(), processes.end(), comp);
int current_time = processes[0].arrival_time;
for (auto &process : processes)
{
    //parameters are set for each process
}
```

3.3.2 Shortest Job First (SJF)

A step ahead from FCFS, this algorithm schedules jobs based on their time of arrival, but in case two processes arrive at the same time, it schedules the process with less burst time first. It improves turnaround time slightly, but is not enough and is still susceptible to workloads where large processes arrive earlier.

```
// sort processes by arrival time
//if arrivalTime same, sort by burstTime
std::sort(processes.begin(), processes.end(), comp);
int current_time = processes[0].arrival_time;
for (auto &process : processes)
{
    //parameters are set for each process
}
```

3.3.3 Shortest Time to Completion First (STCF)

This is the preemptive version of SJF, it always schedules the shortest job in the queue at any time and preempts any longer running process if a shorter one arrives. In my testing, it was observed that STCF consistently yielded the best turnaround time, and hence is the best algorithm for efficiency, consqueuently, it is quite poor in fairness, as a relatively larger algorithm could be blocked for a very long time if shorter processes exist.

```
while(incomplete_processes)
{
// iterate through all schedulable processes to find the shortest one
int sji = -1;
int shortest_ttc = INT_MAX;
for (auto &process : processes)
{
    // if schedulable
    if (process.time_to_completion > 0 && process.arrival_time <= current_time)
    {
        if (process.time_to_completion < shortest_ttc)
        {
            // a little pointer magic to get the index
```

```
            sji = &process - &processes[0];
            shortest_ttc = process.time_to_completion;
        }
    }
}
//calculate the parameters for the process
}
```

3.3.4 Round Robin

The algorithm that consistently performs the best in fairness. It takes a time slice argument, and it only schedules any process for a time slice. That is, it performs a context switch and schedules another available process every time slice. This leads to it having a very low response time, but relatively high turnaround time.

```
//code is almost identical to STCF, but this is the schedulable check
    // the two edge cases need to ba handled differently as the scheduler will
    if (processes[ptr].time_to_completion == 0)
    {
        // simply advance pointer and check
        ptr = (ptr + 1) % proc_count;
        continue;
    }
    if (processes[ptr].arrival_time > current_time)
    {
        // all subsequent processes are unreachable as well
        // the scheduler will idle
        current_time++;
        continue;
    }
```

3.3.5 Lottery Scheduling

It is a probabilistic fair share algorithm. It expects the user to allot each process a certain number of tickets proportional to its CPU share. It holds a "lottery" every time slice, chooses a number in the range

$$[0, ticketpool - 1]$$

and schedules the "winning" process, i.e the process that holds the winning ticket number. It is relatively longer to implement, so the code snippets will not be displayed here. It nevertheless produces good results for response time on smaller values of time slice.

4 Summary and Points of Improvement

In summary, the project provides valuable insights into the performance of different algorithms and in general reinforces the good ideas in operations scheduling and management.
The homework needed to be done for the project lead to a lot of learning about virtualisation, process management and context switching. It was also, for lack of a better word, fun!
Some things could have been improved/added -

- The code could have been cleaner and more modular on the backend, but I did not devote much time to it as that was not the main crux of the project.
- More complex and complete solutions for scheduling like the Multilevel Feedback Queue or the Linux Completely Fair Scheduler could have been implemented. Will try work on these in the future.