**SID: Benjamin Ahamed 45934029**

**Introduction**

Stage 2 of this project focuses on optimising the program based on one or more of the following metrics:

* Minimisation of average turnaround time
* Maximisation of average resource utilisation
* Minimisation of total server rental cost

This to be done via communication with the job server using specific commands and the internal algorithm created by this project. The algorithm created must of course work while also being optimised on the previously stated metrics. It needs to outperform the algorithms of Best Fit, First Fit and Worst Fit in some of the tests that prioritise one or more of the previous metrics.

**Problem Definition**

My main objective with my scheduling algorithm was a multi-faceted approach. I wanted to keep costs down as low as possible while also decreasing turnaround time but mainly focusing on Resource Utilisation. While Resource Utilisation is important the cost of running a server is also a serious aspect to keep in mind, thereby ensuring that the server whose capabilities is closest to that of the job is prioritised.

By also incorporating target on costs, it help keep costs from getting to high then would be the case if the algorithm purely focused on Resource Utilisation. While my algorithm won’t be able to beat turnaround results from best fit algorithm it is possible to provide increased performance in Resource Utilisation. Therefore I decided to utilise aspects of First fit however choose aspects of Best fits as well.

**Algorithm Description**

The client and algorithm I have constructed asks for a job from the server and when it receives the JOB data it parses it into an algorithm contained with the method toLargest();. The algorithm asks the sever to send the data for all servers capable of running the job. The algorithm passes through this server data and the first server it finds not being used it schedules the job for the server. If the algorithm can’t find an unused server it will then parse through the sent servers and find servers with the lowest possible number of waiting jobs. These servers are then put into a array list which is then sorted based on number of cores available. The algorithm then returns the server with the largest number of available cores and schedules the job on the server.

Severs

server type="juju" limit="2" bootupTime="60" hourlyRate="0.2" coreCount="2" memory="4000" disk="16000"

server type="joon" limit="2" bootupTime="60" hourlyRate="0.4" coreCount="4" memory="16000" disk="64000"

server type="super-silk" limit="1" bootupTime="80" hourlyRate="0.8" coreCount="16" memory="64000" disk="512000"

Jobs

job type="short" minRunTime="1" maxRunTime="300" populationRate="60"

job type="medium" minRunTime="301" maxRunTime="1800" populationRate="30"

job type="long" minRunTime="1801" maxRunTime="100000" populationRate="10"

Clients = C

Server = S

Simple Example Scheduling Scenario

The C establishes a connection with S and sends the REDY command. The C asks for a job and is sent JOB “Short”, the C will then send a REPLY asking GETS CAPABLE + Sever Information. The S will then send DATA containing all servers capable of handling the JOB. The S will automatically sort the data with the first server being the smallest and the last being the largest i.e. ascending. The C will then be starting with the first sever sent, look at the Status of the Server and if it is not being used will SCHD the JOB with that server. This process will continue until when all servers are being used the method getsCapable2() is used.

getsCapable2() will then look at the sent servers number of waiting jobs and store the smallest number it could find . Since multiple servers may have 0 jobs waiting, the C will look at the number of cores available of the servers with 0 jobs waiting and will schedule a job on the server with the largest number of available cores. This is done in order to maximise the number of jobs running concurrently and ensure all available resources are used.

**Implementation**

getsCapable(String s) is the first algorithm that accepts a string input (the server information) will then look at the status of the server by breaking the string in to an array and comparing it to strings of known active states. If the server’s status is active, booting or unavailable the method will return a Boolean statement of false. Else it will return true indicating the server is not being used and that it can have a job scheduled on it. It will then break the for loop that parses through the DATA sent by the server and initialise a new input Stream to clear the old one.

GetsCapable2(ArrayList a, ArrayList b) accepts 2 Arraylist one containing the capable servers sent by the Server (array) and the other containing the number of waiting jobs (numJobs) of each server as an Integer. This was done to allow easier sorting of numbers using inbuilt methods and reduces the number of times the arraylist needs to be iterated through. An Arraylist was chosen as the data structure as it is dynamic in size and the Collections library contains many useful methods to manipulate an ArrayList.

**Evaluation**

The new Algorithm succeed in optimising TurnAround Time while causing a minimal increase in Rental Cost. The algo significantly improved in performance relative to WF algorithm in all aspects, particularly turnaround time.

The Results of running the test simulation provided in week 12 focusing on resource utilisation (ru)

Turnaround Time

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | AllToLargest | First Fit | Best Fit | Worst Fit | New Algorithm |
| Average | 254086.33 | 1473.33 | 1462.83 | 6240.72 | 1543.89 |
| Improvement | 49.53% |  |  |  |  |

Resource Utilisation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | AllToLargest | First Fit | Best Fit | Worst Fit | New Algorithm |
| Average | 100.00 | 66.79 | 64.94 | 72.85 | 69.12 |
| Improvement | 1.36% |  |  |  |  |

Total rental cost

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | AllToLargest | First Fit | Best Fit | Worst Fit | New Algorithm |
| Average | 256.05 | 417.90 | 414.42 | 443.03 | 442.15 |
| Improvement | -4.01% |  |  |  |  |

These results demonstrate that my Algorithm, consistently outperforms Worst Fit in all categories. The algorithm beats AlltoLargest in all categories except total rental cost which would be expected.

The difference in performance in TT for First Fit and Best Fit when compared with the New Algorithm is minimal as is the Resource Utilisation and Total Rental Cost. In multiple config run during the test such as those for config20-med-high in Resource Utilisation my Algorithm was able to outperform all other Algorithms.

**Conclusion**

The algorithm was successful in trying to maximise optimisation while not increasing price to much relative to the other algorithms. It out performed all the other Algorithms in certain tests for Resource Allocation all why only having a slightly worse Cost efficiency then the average of the other algorithms. I believe there is still room to improve turnaround times to levels more comparable to that of the Best fit algorithms. The goal of this project has been met by achieving improvements in resource allocation.

References

Github: https://github.com/BenAhamed/COMP3100assignment2