Monte Carlo Simulation of Partitioning Algorithms.

Table of Contents

Problem Statement

The requirements for this programming assignment were to write Monte Carlo simulations to determine which of various partitioning algorithms perform the best when given memory requirements that follow a poisson distribution with a mean of 8 and a uniform time distribution between 1 and 10 inclusively. The data for each experiment needed to be processed in a FIFO order, and data that was too large to fit in the largest partition needed to be counted as a failure. If it is counted as a failure, the data still needs to use the partition for however long it would have normally.

For simulations using the single queue and unequal partitioning configuration, if the head of the queue cannot enter the partition table, then it blocks access to smaller data members behind it in the queue until it can be put into the partition table. For the multiple queues and unequal partitioning configuration, the original queue can be pre-processed into the multiple queues if the relative order is maintained. The simulation is done with a uniprocessor system that has a single time quantum which loops through the “tasks” in a round robin fashion. The code for this program had to follow the structure shown in the “Pgm3 System Specification” document.

The three partitioning methods that we were required to design simulations for were an equally sized partition table with seven partitions that are all 8MB in size and two simulations that use an unequally sized partition table with partitions of sizes of 2MB, 4MB, 6MB, 8MB, 8MB, 12MB, and 16MB. One of the unequal partitioning simulations used a single queue and the other used multiple queues, one for each partition.

The importance of this programming assignment was that I could get hands-on experience working with various methods for memory partitioning. Alongside that, I could gain an intuitive understanding of why some methods are better than others. Essentially, this project can give me experience that I can actually use in writing an OS.

Approach

To solve this problem, I chose to use C++ and compile it using the GNU g++ compiler. To edit text, I used vscode because I am familiar with it. For the operator system, I used Ubuntu 20.04.

My approach to solving this problem involved creating various structs that were used to handle that information used in the simulations. I had structures to hold the results, information for a data member, and information for static and dynamic partitions. The work was done by various functions, the parameters of which were either passed by value if the function didn't need to manipulate it, passed by reference if the function did need to manipulate it, or passed as a pointer mainly for the result structures since there's ever only one instance for each partitioning configuration.

My solution followed the structure of the requirements document, so my program loops through the 1000 experiments, creating 1000 samples for each experiment and storing them in an array of data members. For each experiment, I performed the three static partitioning configuration simulations and the dynamic configuration simulation. The array of data members and the structures that hold the results are passed into each function that performs the experiments. After all 1000 experiments are done, I report the results by calculating the average number of failures, the average turnaround time, the average relative turnaround time, and the average number of data members in the partition table. The last calculation was not in the requirements, but I added it because I felt that it gives good insights into which partition style is better.

Each of the simulations for the static configurations start the same way. They all create a copy of the array of experiment data and then create the partition table. The equal partitioning configuration creates an array of 7 static partition structures, all of which are 8MB. Both of the unequal partitioning configurations create an array of 7 static partition structures with sizes that are 2MB, 4MB, 6MB, 8MB, 8MB, 12MB, and 16MB. The one queue unequal configuration and the equal configuration both use one queue, which is implemented by simply using the array of data members and keeping track of the front of the queue using an index variable which is incremented every time something is "popped" from the queue. The multiple queue unequal configuration uses actual queue data structures to make it easy on myself. There's one queue for each partition and they're filled up at the beginning of the simulation. Every data member with sizes that are less than or equal to 2MB go in the 2MB queue. Data members that are greater than 2MB but less than or equal to 4MB go in the 4MB queue. And so on and so forth. Data members that are greater than 16MB go into the 16MB queue and are counted as failures.

The execution of the static partitioning configurations are all done pretty much in the same way. The only difference is how data is inserted into the partition tables once a data member is finished "processing". They all use an infinite loop for the clock. Each loop performs the work for one quantum, which entails decrementing the time left for each member in the current partition, and then going to the next partition in the next loop in a round robin fashion. After decrementing the time left for a data member, it checks whether that member has no time left. If it doesn't, it then discards the data member and brings in a new data member from either the single queue or the queue that corresponds to the current partition in the case of the multiple queues configuration. For the single queue unequal partitioning configuration, the next item may not be able to fit in any empty partitions, so it will block until the next time that a data member finishes. Naturally, multiple data members can be brought in at once, so the algorithm for filling partitions loops and continues filling until no more partitions can be filled. Once no more data members exist in either the single queue or any of the multiple queues, the experiment is finished, so it breaks out of the infinite loop. During the entire process, various calculations are made, such as the average turnaround time.

The dynamic partitioning configuration starts by copying the data member array like the static configurations. Instead of creating an array of static partitions, a linked list is created that holds dynamic partitions. The reason for this is because the list can theoretically hold as low as 1 partition, or as high as 56 partitions, so a static array wouldn't be reasonable. The first fit algorithm is performed once, initially, by using the list of partitions and the array of data members. This configuration uses a single queue, for input, that is implemented similarly to the static partitioning configurations with a single queue.

The execution of the dynamic partitioning configuration simulation is mostly similar to the static partitioning simulations, except that how items are inserted into memory is completely different. There's an infinite loop where each loop is one quantum and deals with each partition in a round robin fashion. The time left of the member in the current partition is decremented, and if it reaches zero, it's discarded. At this point, a "first fit" partitioning algorithm is executed, which basically means that the data is inserted into the first memory location that it can fit. If there's no place for it to fit despite there being enough free space in memory, a compaction algorithm is performed. That compaction algorithm simply removes all holes and makes all partitions contiguous, and the freespace is pushed to the end of memory, allowing the data member to be inserted as a new dynamic partition at the end of memory.

Solution Description

For my solution, I had to use a few standard C++ libraries. Figure 1 is a table which lists each of the libraries that I used. The table also describes what I used them for.

<iostream>	Used for outputting to stdout.
<random>	Used to generate the random numbers.
<queue>	Used to create the queues in the multiple queues unequal partitioning configuration.
<list>	Used to create the list in the dynamic partitioning configuration.
<ctime>	Used to get the seed for the random number generator.

Figure 1. This is a table that lists the C++ libraries that I used in my program. On the left, it shows the name of the library. On the right, it shows a description of what the library is used for.

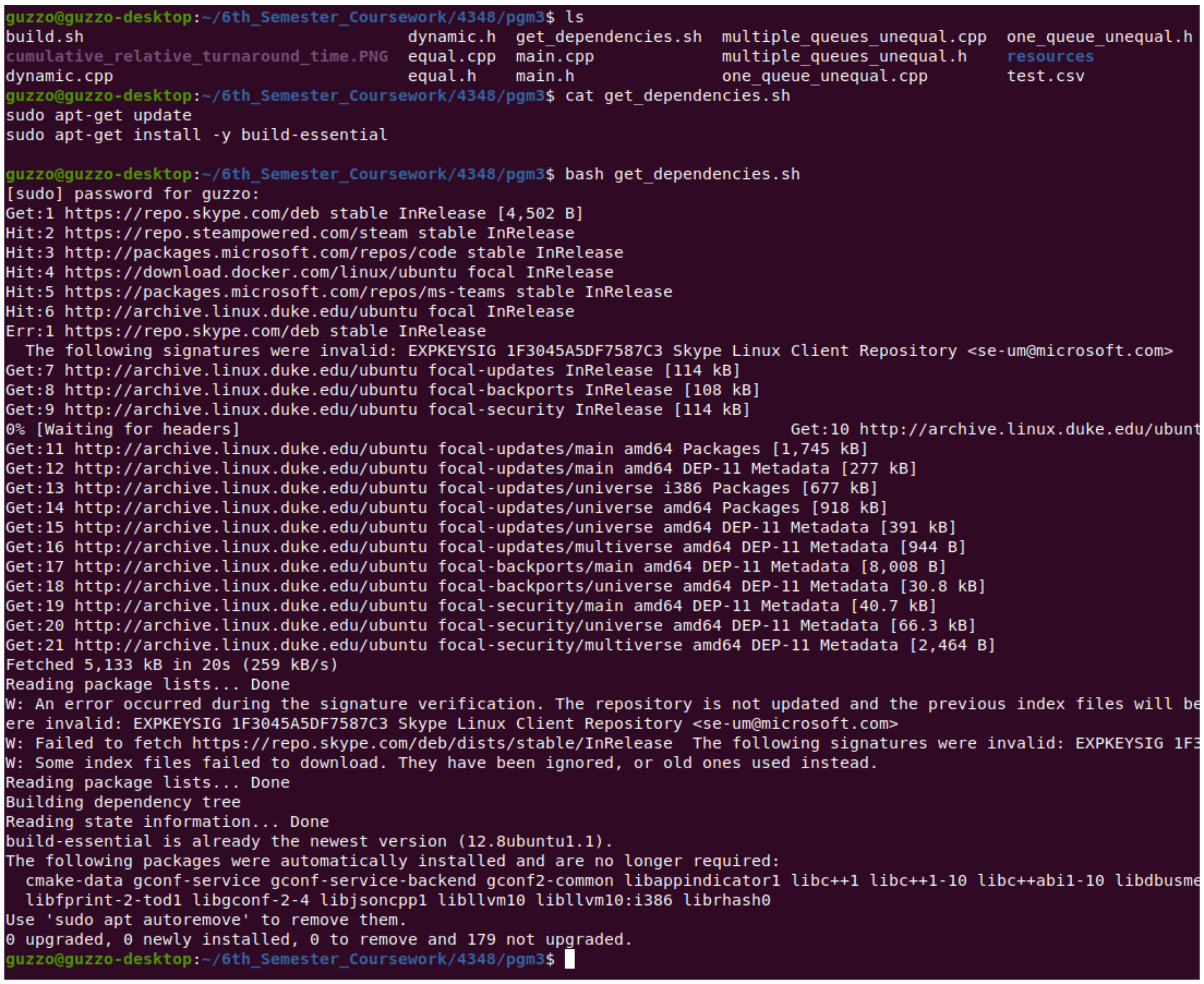
Naturally, I created a few functions and structures for this program. Figure 2 is a table that lists both the functions and structures. The left side lists the function and structure prototypes, and the right side gives a brief description of what they're used for.

struct Results	Holds the results for all experiments.
struct Data	Holds the information for a single data member.
struct StaticPartition	Holds the information for a single static partition.
struct DynamicPartition	Holds the information for a single dynamic partition.
int main(int argc, char* argv[])	Entry point for this program. Initializes the data and initiates the experiments.
void report_results(Results* equal, Results* one_queue_unequal, Results* multiple_queues_unequal, Results* first_fit)	Reports the results after all experiments are complete.

void setup_unequal_static_partitions(StaticPartition (&partitions)[7])	Sets up the sizes for the unequal partitions and initializes each partition.
void equal_partitioning(Data data[NUMBER_OF_SAMPLES], int number_of_samples, Results* equal)	Performs the experiment for the equal partitioning style.
void one_queue_unequal_partitioning(Data data[NUMBER_OF_SAMPLES], int number_of_samples, Results* one_queue_unequal)	Performs the experiment for a single queue using an unequal partitioning style.
void one_queue_fill_unequal_partitions(Data (&data)[NUMBER_OF_SAMPLES], int &next_data, int &number_of_failures, StaticPartition (&partitions)[7], int &num_data_members_in_partition_table, int clock)	Fills the next available partition with the data member at the front of the queue if there is an available partition.
void multiple_queues_unequal_partitioning(Data data[NUMBER_OF_SAMPLES], int number_of_samples, Results* multiple_queues_unequal)	Performs the experiment for multiple queues using an unequal partitioning style.
void preprocess_multiple_queues(queue<Data> (&queues)[7], Data(&data)[NUMBER_OF_SAMPLES], int &number_of_failures)	Place the experiment data into the multiple queues based on their sizes.
void dynamic_partitioning(Data data[NUMBER_OF_SAMPLES], int number_of_samples, Results* first_fit)	Performs the experiment for the dynamic partitioning style that uses the first fit placement algorithm.
void perform_first_fit_algorithm(Data (&data)[NUMBER_OF_SAMPLES], list<DynamicPartition> &partitions, int &next_data, int &num_data_members_in_partition_table, int clock, int &number_of_failures)	Performs the first fit placement algorithm.
int compact(list<DynamicPartition> &partitions)	Performs the compaction algorithm.
void print_dynamic_partitions(list<DynamicPartition> partitions, Data data[NUMBER_OF_SAMPLES], int next_data)	Prints the current set of partitions in memory for the dynamic partitioning style. Used for debugging purposes.

Figure 2. This is a table that contains all of the functions and structures that I created in my program. On the left, it shows the function/structure prototype. On the right, it shows a brief description of what it does.

In order to run my program, you need a few dependencies. Figure 3 is a screenshot which shows how you can get the dependencies. It uses a bash script which uses the APT package manager to get the dependencies.

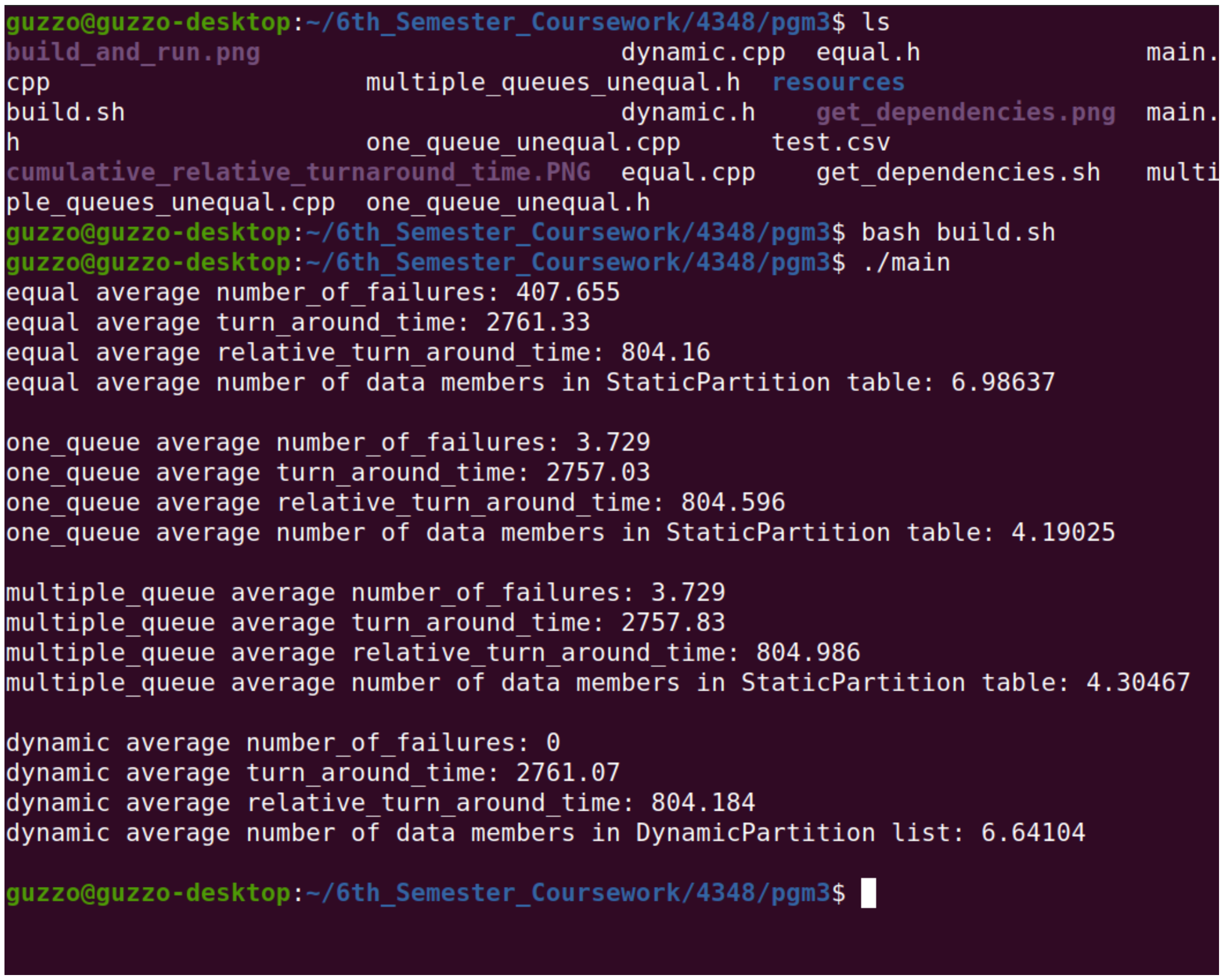


```
guzzo@guzzo-desktop:~/6th_Semester_Coursework/4348/pgm3$ ls
build.sh                                  dynamic.h  get_dependencies.sh  multiple_queues_unequal.cpp  one_queue_unequal.h
cumulative_relative_turnaround_time.PNG  equal.cpp  main.cpp             multiple_queues_unequal.h    resources
dynamic.cpp                               equal.h    main.h               one_queue_unequal.cpp        test.csv
guzzo@guzzo-desktop:~/6th_Semester_Coursework/4348/pgm3$ cat get_dependencies.sh
sudo apt-get update
sudo apt-get install -y build-essential

guzzo@guzzo-desktop:~/6th_Semester_Coursework/4348/pgm3$ bash get_dependencies.sh
[sudo] password for guzzo:
Get:1 https://repo.skype.com/deb stable InRelease [4,502 B]
Hit:2 https://repo.steampowered.com/steam stable InRelease
Hit:3 http://packages.microsoft.com/repos/code stable InRelease
Hit:4 https://download.docker.com/linux/ubuntu focal InRelease
Hit:5 https://packages.microsoft.com/repos/ms-teams stable InRelease
Hit:6 http://archive.linux.duke.edu/ubuntu focal InRelease
Err:1 https://repo.skype.com/deb stable InRelease
  The following signatures were invalid: EXPKEYSIG 1F3045A5DF7587C3 Skype Linux Client Repository <se-um@microsoft.com>
Get:7 http://archive.linux.duke.edu/ubuntu focal-updates InRelease [114 kB]
Get:8 http://archive.linux.duke.edu/ubuntu focal-backports InRelease [108 kB]
Get:9 http://archive.linux.duke.edu/ubuntu focal-security InRelease [114 kB]
0% [Waiting for headers]                                                Get:10 http://archive.linux.duke.edu/ubunt
Get:11 http://archive.linux.duke.edu/ubuntu focal-updates/main amd64 Packages [1,745 kB]
Get:12 http://archive.linux.duke.edu/ubuntu focal-updates/main amd64 DEP-11 Metadata [277 kB]
Get:13 http://archive.linux.duke.edu/ubuntu focal-updates/universe i386 Packages [677 kB]
Get:14 http://archive.linux.duke.edu/ubuntu focal-updates/universe amd64 Packages [918 kB]
Get:15 http://archive.linux.duke.edu/ubuntu focal-updates/universe amd64 DEP-11 Metadata [391 kB]
Get:16 http://archive.linux.duke.edu/ubuntu focal-updates/multiverse amd64 DEP-11 Metadata [944 B]
Get:17 http://archive.linux.duke.edu/ubuntu focal-backports/main amd64 DEP-11 Metadata [8,008 B]
Get:18 http://archive.linux.duke.edu/ubuntu focal-backports/universe amd64 DEP-11 Metadata [30.8 kB]
Get:19 http://archive.linux.duke.edu/ubuntu focal-security/main amd64 DEP-11 Metadata [40.7 kB]
Get:20 http://archive.linux.duke.edu/ubuntu focal-security/universe amd64 DEP-11 Metadata [66.3 kB]
Get:21 http://archive.linux.duke.edu/ubuntu focal-security/multiverse amd64 DEP-11 Metadata [2,464 B]
Fetched 5,133 kB in 20s (259 kB/s)
Reading package lists... Done
W: An error occurred during the signature verification. The repository is not updated and the previous index files will be
ere invalid: EXPKEYSIG 1F3045A5DF7587C3 Skype Linux Client Repository <se-um@microsoft.com>
W: Failed to fetch https://repo.skype.com/deb/dists/stable/InRelease  The following signatures were invalid: EXPKEYSIG 1F3
W: Some index files failed to download. They have been ignored, or old ones used instead.
Reading package lists... Done
Building dependency tree
Reading state information... Done
build-essential is already the newest version (12.8ubuntu1.1).
The following packages were automatically installed and are no longer required:
  cmake-data gconf-service gconf-service-backend gconf2-common libappindicator1 libc++1 libc++1-10 libc++abi1-10 libdbusme
  libfprint-2-tod1 libgconf-2-4 libjsoncpp1 libllvm10 libllvm10:i386 librhash0
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 179 not upgraded.
guzzo@guzzo-desktop:~/6th_Semester_Coursework/4348/pgm3$
```

Figure 3. This screenshot shows how to install the dependencies for my program. The script is located in “get_dependencies.sh”. To run this script, enter in “bash get_dependencies.sh”.

After getting the dependencies, Figure 4 demonstrates how to build and run the program. The script to build is located in “build.sh”. This script uses the g++ compiler.



```
guzzo@guzzo-desktop:~/6th_Semester_Coursework/4348/pgm3$ ls
build_and_run.png                         dynamic.cpp   equal.h              main.
cpp                     multiple_queues_unequal.h  resources
build.sh                                  dynamic.h    get_dependencies.png  main.
h                       one_queue_unequal.cpp      test.csv
cumulative_relative_turnaround_time.PNG  equal.cpp    get_dependencies.sh   multi
ple_queues_unequal.cpp  one_queue_unequal.h
guzzo@guzzo-desktop:~/6th_Semester_Coursework/4348/pgm3$ bash build.sh
guzzo@guzzo-desktop:~/6th_Semester_Coursework/4348/pgm3$ ./main
equal average number_of_failures: 407.655
equal average turn_around_time: 2761.33
equal average relative_turn_around_time: 804.16
equal average number of data members in StaticPartition table: 6.98637

one_queue average number_of_failures: 3.729
one_queue average turn_around_time: 2757.03
one_queue average relative_turn_around_time: 804.596
one_queue average number of data members in StaticPartition table: 4.19025

multiple_queue average number_of_failures: 3.729
multiple_queue average turn_around_time: 2757.83
multiple_queue average relative_turn_around_time: 804.986
multiple_queue average number of data members in StaticPartition table: 4.30467

dynamic average number_of_failures: 0
dynamic average turn_around_time: 2761.07
dynamic average relative_turn_around_time: 804.184
dynamic average number of data members in DynamicPartition list: 6.64104

guzzo@guzzo-desktop:~/6th_Semester_Coursework/4348/pgm3$ 
```

Figure 4. This is a screenshot that shows you how to build and run my program. The script to build is located in “build.sh”. To build the program, enter in “bash build.sh”. You can then run the program by entering in “./main”.

Below are a few charts that show the differences between each partitioning configuration. Figure 5 shows the average number of failures. Figure 6 shows the average turnaround times. Figure 7 shows the average relative turnaround times. Lastly, Figure 8 shows the average number of data members in the partition tables.

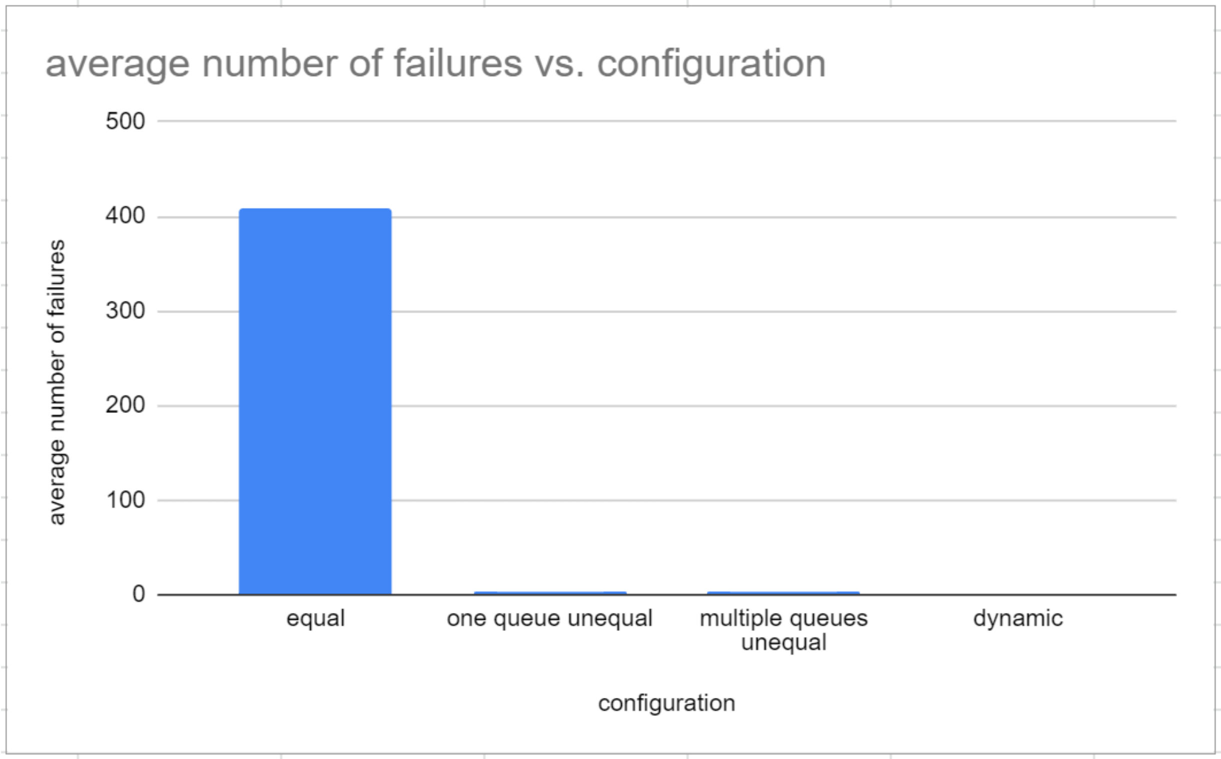


Figure 5. This bar graph shows the average number of failures for each configuration. As can be seen, the equal partitions configuration has significantly more failures than the rest of the configurations. Both unequal configurations have very small numbers of failures, on average. The dynamic partitioning configuration has practically no failures.

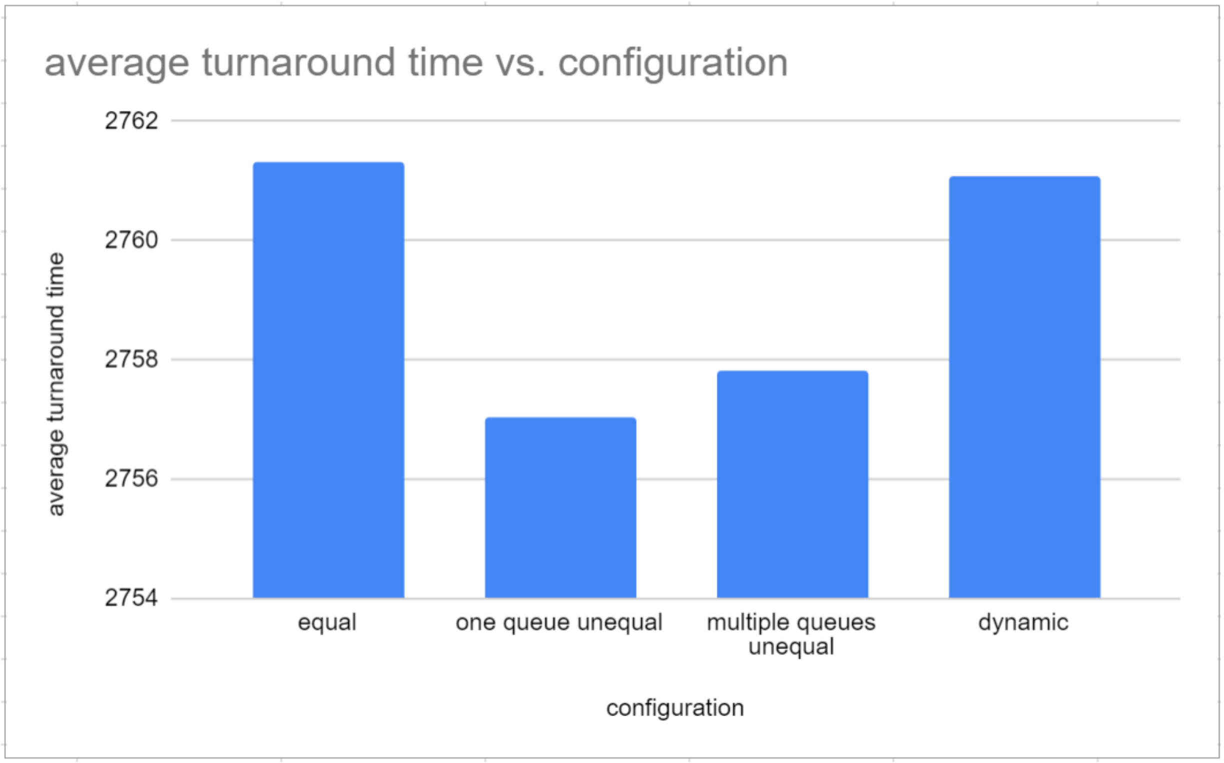


Figure 6. This bar graph shows the average turnaround times for each configuration. As can be seen, since time starts at zero for all data members, the average turnaround times are very close for the different configurations. The equal partitioning and dynamic partitioning configurations seem to have turnaround times that are slightly larger than the unequal partitioning configurations.

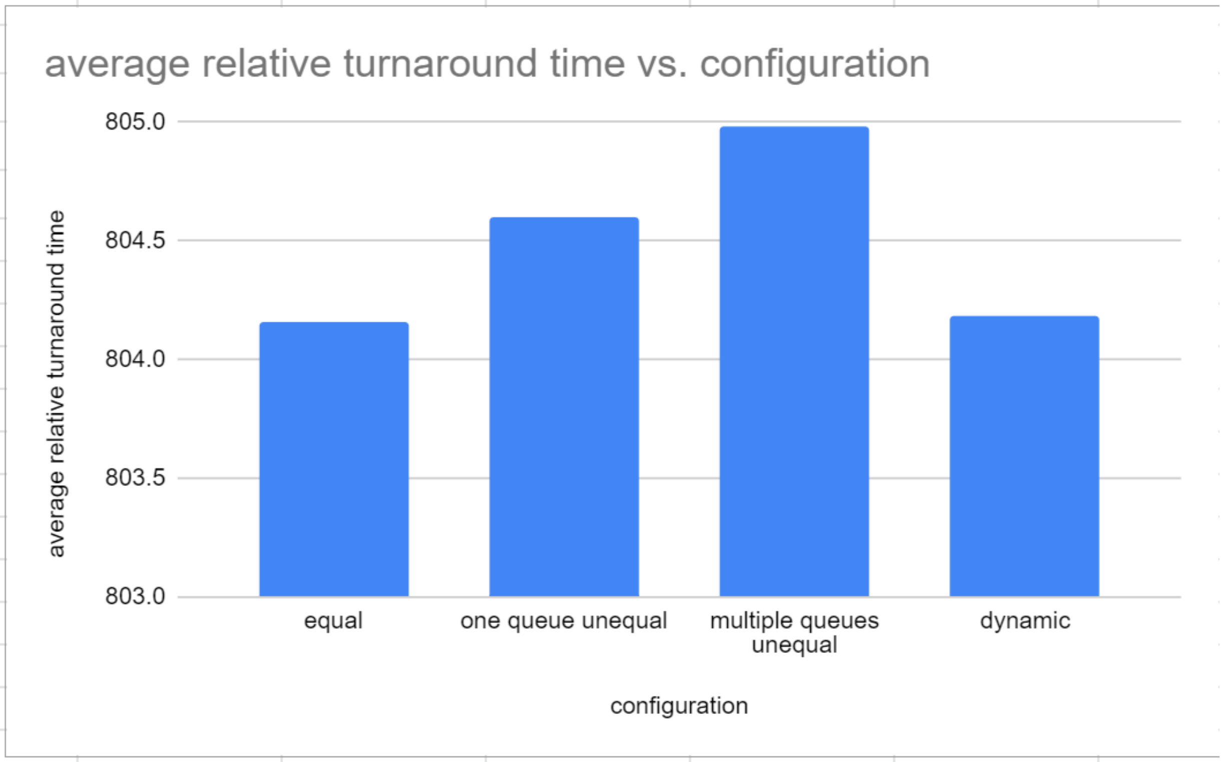


Figure 7. This bar graph shows the average relative turnaround times for each configuration. As can be seen, the values are very close to each other for all configurations, with only slight differences. The equal and dynamic partitioning configurations seem to have slightly lower average relative turnaround times.

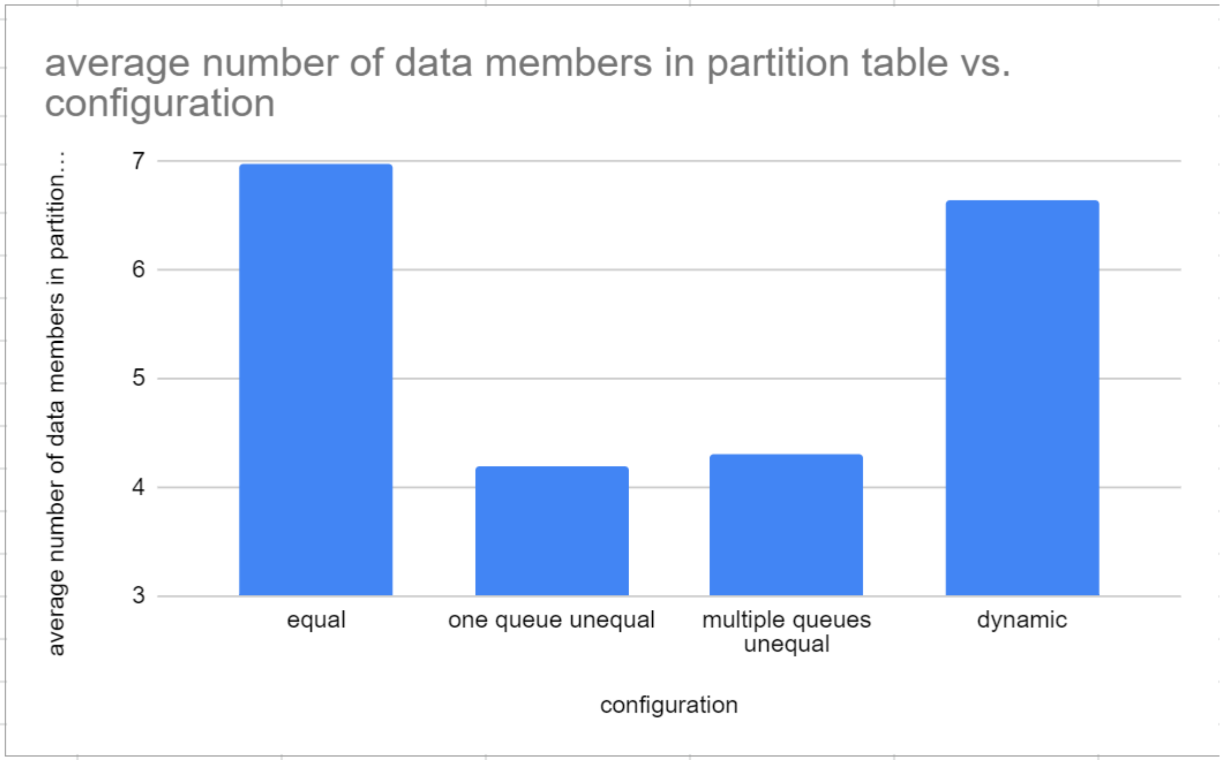


Figure 8. This bar graph shows the average number of data members in the partition tables. For the dynamic partitioning configuration, it's a list, not a table. As can be seen, the equal partitioning configuration almost has an average of 7 data members. The dynamic partitioning configuration has almost 7 data members, but it's a bit lower. Both the one-queue and multiple-queue unequal partitioning configurations are slightly above 4 data members, on average.

As can be seen, I added a data point for consideration: the average data members in the partition table. The reason I added this is because I believe it gives a good idea of how capable the partitioning configuration is for efficiently utilizing main memory. I also felt that I needed it so that we could better understand the benefits of dynamic partitioning. For the comparison of the four partitioning configurations that I tested, I will talk about each data point, and then make a final decision on what I think is the better partitioning configuration.

Starting off with the average number of failures, it is clear that equal partitioning results in way too many failures. About 40% of the data members resulted in a failure. This is because the sizes are based on a poisson random distribution with a mean of 8, which is the size of each equal partition. Both of the unequal partitioning configurations have the same value for the average number of failures, which is much lower than the equal partitioning configuration. Having a 12MB and 16MB partition allows for the OS to deal with most of the larger data members, with only about 0.38% of the data members resulting in failure. The dynamic partitioning configuration practically results in zero failures because the only data members that can fail are larger than 56MB, and that size is highly unlikely. In this case, the dynamic partitioning configuration wins.

For the average turnaround times, it is clear that they are practically the same for all of the configurations. The unequal configurations seem to have slightly lower average turnaround times than the equal and dynamic configurations. This might be attributed to the fact that unequal configurations have a lower average number of data members in the partition tables. In this case, the one queue unequal configuration is the lowest, but not that much lower. This data point should not significantly affect my decision.

For the average relative turnaround times, it appears that equal and dynamic configurations are better, but not by that much. The differences are miniscule. This data point should also not significantly affect my decision.

The last data point considered is the average number of data points in the partition table/list. This point is important because it basically shows how efficiently the configuration utilizes main memory. As can be seen, the equal and dynamic partitioning algorithms seem to be superior with values very close to 7. The dynamic partitioning configuration is slightly lower. The average number of data members in the partition tables for the unequal configurations are much lower, meaning they use memory less efficiently.

Considering all of these data points, I have come to the conclusion that the **dynamic partitioning configuration** is the best one. The average turnaround times and average relative turnaround times are practically the same as the other configurations. However, it is almost tied with the equal partitioning configuration with the lowest average relative turnaround time, and I believe that average relative turnaround time is a bit more important. Also, the dynamic partitioning configuration has the lowest failure rate, which is pretty much non-existent, while still maintaining a higher average number of data members in the partition list. The dynamic configuration uses memory efficiently while eliminating failures and having one of the best relative turnaround times.