CS1007 A03: Containers

Matriculation Number: 230022876

Introduction

In this coursework, I was required to setup a method to record the usage of resource for three separate programs provided using containers. Using the data collected, the task was to aggregate this, and record the findings of this data. This would then be evaluated and discussed in this report. An explanation of all the points of this experiment have been included and explained below, in the relevant subsections.

Testbed Setup

For the environment of testing, the lab machine computer I used was pc7-026. The data was collected at Mon 13 Nov 17:10:00 GMT 2023, with each program being run individually.

To avoid complications that can arise from the sharing of resources between each task that is running simultaneously (due to a certain time slice being given to each running process etc.), the programs will be run on individual built containers, for each program. This will also allow us to be able to run the programs we are testing without needing to worry that the data collected may have side effects from other processes that may be running simultaneously within the computer (and those that are out of our control to manage). As explained by Google Cloud's explanation on containers, "Containers allow your developers to move much more quickly by avoiding concerns about dependencies and environments", thus why it is recommended to use containers for data collection and analysis of the programs in this assessment.[1]

First, I created a Container Volume called 'A03-Volume', that can be accessed by the container when it is run. Within this volume, I created three test files being Testfile-hn__4.3.03__va_nFXnp12, Testfile-prog1 and Testfile-test that can be used by their designated programs, hn__4.3.03__va_nFXnp12, prog1 and test respectively.

To allow the programs to run, I created a separate directory for each of the programs, copied the programs into their designated directory, and included individual Containerfiles for each of these programs within the designated directory. Each of these Containerfiles would indicate that the image would be built from the latest AlmaLinux image. After this, I would create a new user called 'script-user' and assign the new user to be the 'script-user' that was just created. After this, the program to be used would be copied from the directory into the container, and the owner of this program would be changed to the new 'script-user' that was created and made the user. After the copying of the program, the access modifiers of this program would be changed to 766, giving the 'script-user' the ability to execute this program (in case it was not provided before). Finally the command would be executed, which would be ./<program-name> <test-file>, which would run the program on the designated test file within the created container, which is the intent for creating this container. All three programs have identical container files with only the name of the program and test file being different, allowing for consistency and to reduce any instance of systematic error that could occur.

To be able to run these containers, we first must build the container. All the container build commands are exactly the same, with the directory from the where the Containerfile being built, and the name of the image being built being the only difference between the build command for the three containers. For building, the command used would be "podman build -t <image-name> ./<directory>". I have used the -t flag to specify a specific image name, to allow ease of access when building the image.

After this, it is required to run the container, and so run the image. To do this, I used a 'podman run' command which would have the same syntax for all of the programs, apart from the name of the image, as well as the name of the container running the image. The command specifically would be
"podman run -v A03-Volume:/A03-Volume --name=<container-name> -d --rm <image-name>:latest".
To explain this command, we use the -v flag to mount the A03-Volume to a volume we name 'A03-Volume' inside the container (this allows the ability to access the test files for our programs). The --name flag is used to provide the container with a specific name, allowing the 'Podman Stats' command to specifically gather data for this container

(more on this in the next section). The -d flag is done to detach the container from the terminal, so it can be run in the background, and allow the 'Podman Stats' command to able to gather data for this container while it is still running. The --rm flag is used to remove the container from the processes list, allowing us to run the container again later on with the same name, and therefore not creating an error. It is also used to stop podman stats from running infinitely, and therefore only doing so while the container is being run (and so the program is being run).

Method

For running all my experiments, I created a script called 'CreateRunContainers.sh' which builds the container image, runs the image within a container, and then collects the data for the running program and manipulates this data into a more preferable format (for a script to aggregate the results). This process is done for all three programs to gather the data as required.

The build and run commands are the same as those that were explained in the final paragraph of the section above and are consecutive. After this, I run the sleep command 'sleep 1' to pause the program for one second, before running the podman stats command
'podman stats -i 1 <container-name> --no-reset --format "table {{.CPUPerc}} {{.MemPerc}}"'
and then appending this data to a relevant text file for further parsing through the data later on in the script.

Firstly, to address the issue as to why the 'sleep 1' command has been used, this is done so to allow the container to be properly built before running the podman stats command. Without this, there were instances where the container would not have been built yet, and therefore would skip over the podman stats command, meaning that data for that round of testing would not have been collected. This would not happen too frequently, but in order to avoid this issue, and so decrease the effect of random error to the data caused by this issue, the aforementioned sleep command must be included, ensuring that all the data is collected as it should be.

To record the data, I used the –format flag to format a table that only displays the CPU usage, and the total Memory usage both recorded as a percentage value. The reason for why I used this table, was that it would be easier to extract the value and place it into a data file, rather than other methods that are possible. I also decided to just record the percentage values of these measurements, as they would be easier to parse and analyse this data in comparison to getting the absolute value of these measurements.

In order to record every second (rather than the default of every 5 seconds) I added the -i flag, followed by a 1. This indicates the time interval of each data value being recorded, and in this case, this is being set to 1, so data values for the memory usage and the CPU usage are recorded every second.

I specifically added the container name in the podman stats command, so that if there are any other containers running at the same time (there should not be), only the data for the container running the program we require will have its data recorded.

The data that was collected would be to 2 decimal places, which is a sufficient degree of accuracy for the reasoning behind the collection and processing of this data. This will still allow us to accurately analyse and evaluate the resource usage of the programs being tested. This precision would result in an error uncertainty of 0.005% (0.01/2), resulting in very accurate data for the purpose.

All of the data is recorded into a text file. However, this text file would have the table headers being placed on all the odd lines, with the data we want to parse only on every even line. To combat this, the script loops through the text file, and for every even line, separates the values using a comma, and appends this to a csv file for the designated program being run. Within this, the formatting for the line is also slightly fixed, such as removing the spaces between the percentage sign and the number. This is so that for the parsing and processing of the data, there is a less likelihood of errors, and to be easier to perform the processing. After this the text file is then deleted (there is no need for it, and the data is already saved onto a separate csv file).

One issue I did come across for when the data is being saved, as the program has to stop running before the container is closed, there would very rarely be an instance where the values of the data recorded would be

‘0.00%,0.00%’, where the data is taken in this moment. This is a section of random error, and can heavily skew our final results, especially for the average CPU% usage across the entire program. To prevent this, in the section where the data formatting is fixed, I also check whether the line of data is the same as ‘0.00%,0.00%’. If so, the data will not be saved onto the csv file, as this is erroneous data, and therefore should not be used for the final aggregation of the data results.

This script performs this data collection process for all the programs being investigated individually, so as to prevent any effect that running these containers simultaneously may have on the data (though the usage of containers should reduce to a minimum regardless). The final data values are appended to the csv files, that are named hnRunDataStore.csv, prog1RunDataStore.csv and test1RunDataStore.csv that correspond to the hn__4.3.03__va_nFXnp12, prog1 and test programs respectively. These data files are saved in a data directory, outside of the working environment directory that all the Containerfiles and data collection scripts are placed in.

As it is always important to get a large dataset for data analysis, I had decided to run the ‘CreateRunContainers.sh’ script multiple times. To do this, I created another script within the same directory called ‘RunContainerTests.sh’. The purpose of this script is to run the ‘CreateRunContainers.sh’ a certain number of times. Therefore, for the data collection process, I would run the ‘RunContainerTests.sh’ script, which in turn would run the ‘CreateRunContainers.sh’ in a definite loop.

For the number of times have decided to run the tests, I had selected as a 10 a suitable value. The reason for choosing 10 was that it would provide a large amount of data, that would therefore reduce the effect of any possibility of unaccounted random error occurring within the dataset collected. However, there was less of a need to run it a much larger amount of time, such as 100, as this would not only take a very large significant amount of time, but due there being very little variation in the data results of this tests. This was confirmed by some tests runs and analysis of the data resulting from these tests runs. The data from the test runs was deleted after this confirmation, so none of the current collected data are from any test runs.

Results

In order to collect results that would be relevant to my analysis of the data, I decided to aggregate the data collected for each program being analysed by finding the mean average value of the CPU percentage, and find the highest value recorded for the Memory Usage percentage.

As every program’s data is saved onto a csv file and separate csv files for each program, I decided to use a bash script called ‘FindAggregate.sh’ to parse this data, and output the results as required on a separate text file. The method of aggregating the data is the same for all three programs, with the final data values all being saved onto the save text file ‘FinalDataVals.txt’.

To represent the CPU utilization, I decided to calculate the average CPU utilization, as a percentage. The reason for this is that the value of CPU utilization would be consistent across all data that was collected, therefore a mean average value would be enough to represent the entire data collected. This is calculated by adding up all the data corresponding to the CPU Usage (the first column of data) and dividing this value with the total number of lines in the data file.

To represent the memory usage, I decided to find the largest value of memory recorded, as a percentage. The reason for this is because the value of memory recorded is very varied, with some instances of the program’s runtime using more memory (especially near the end), other times using a much smaller value of memory. This would mean that a mean average value for memory would not be sufficient, or an accurate measurement to draw a conclusion from, and therefore the largest recorded value would provide a more accurate explanation for the resource requirements of each program.

The table below indicates the single value received for each set of the data, and the program that the value corresponds to:

Program	Avg CPU Utilization/%	Max Memory Required/%
hn__4.3.03__va_nFXnp12	**344.797%**	**0.22%**
prog1	**37.2435%**	**48.62%**
test	**99.7415%**	**1.44%**

As mentioned in the previous section, the data values were written to 2 decimal places, and therefore the final values presented in this graph has an error interval of 0.005%.

Using these results, we can make assumptions about each individual program, and evaluate the resources used for each program:

For the program 'hn__4.3.03__va_nFXnp12', the mean average CPU utilization is 344.797%, which indicates that of all the 6 cores, a total of the 344.797% was being used om average, which can correspond to roughly three and a half cores being completely used at maximum. When running this program, all the cores have the CPU utilization shared, as found out by running this, and then checking system monitor, to monitor the core usage. This is a very large value for a single program, considering that the program would roughly only take around 32 seconds per run. This therefore shows that the amount of processing power required would mean that a CPU with a larger number of cores would benefit the execution of this program, allowing a greater number of instructions to be completed at a time frame, and so reduce the amount of stress placed onto the current CPU (from having to execute any background operations, such as allowing the operating system to function). This large number of CPU requirement would also mean that it is imperative that only a computer with 4 cores or greater should be permitted to run this program, as otherwise, there may be errors leading to the execution of the program, and general maintenance of the operating system or other applications running in the background.
In terms of memory usage, the amount of memory used is significantly different. Out of the total amount of memory available, the program 'hn__4.3.03__va_nFXnp12' only used a maximum of 0.22%. The lab machines roughly have 32 Gigabytes of memory available, and for pc7-026, the indicated amount of memory is 32458240 kilobytes(Attained by the bash command 'grep MemTotal /proc/meminfo | awk '{print $2}''[2]). This therefore means that the maximum amount of memory used by this program is 71408 kilobytes (to the nearest byte). Due to the vastly smaller number of memory required, when in comparison to the total number of kilobytes available in memory, we can assume that for running this program we can use a machine with significantly less amounts of memory, and we do not need to worry about the memory usage of this program being too much for the computer.

Moving onto the program 'prog1', the mean average value calculated for CPU utilisation was 37.2435%. Considering that only 6 cores was used, this is a significantly low amount, as only 37.2435% was used of a total 600%. However, when running this container and looking at system monitor it can be seen that at some points during this operation of the program, the entire operation is done on a single core, which changes many times. This can be beneficial in terms of what type of computer can be used, allowing those with only 1 core to still run this program perfectly fine without running into errors.
However, when looking at the maximum memory usage of this program, we can see that the value provided is 48.62%, and therefore almost 50%. Considering that the total amount of memory available is 32458240 kilobytes, the amount of memory used by this program is 15781196 kilobytes (to the nearest byte), and so 15.05 gigabytes in more human readable format. This is a large amount of memory to be used for a single program. This means that in order to be able to run this program, we would need at lead 16 gigabytes, and to be able to run the entire system along the program, we would need at least a total of 19 gigabytes (according to the system monitor, the Operating System on standby takes around 2.7 gigabytes).

Finally, assessing the program 'test', the value calculated for the mean average CPU utilisation was 99.7415%. Despite the entire computer system running with 6 cores, we see through system monitor, that this program appears to be

only using 1 core to the maximum throughout its entire execution. This can be seen as inefficient, as it means that the benefit of parallel processing is not utilised. This indicates that to run this program, we only need a CPU with 1 core, though it is always recommended to have another to run any background processes not related to the program execution. However, considering that only one core is being used, the clock speed of this core is also relevant to the execution of this program, and having a clock speed of a greater value than available would be able to decrease the time spent executing this program.
Regarding the memory usage of the 'test' program, only 1.44% of the memory was used at maximum. Using the total value of bytes in memory from previously, this would correspond to roughly 467399 kilobytes, or 456 megabytes. This is a significantly low amount of memory to be used within this program, and therefore to run this program, we can have a system memory that is fairly small, only needing to accommodate for the memory usage of the operating system. Therefore, the total amount of memory can be easily decreased, without worrying about any memory errors resulting in the execution of this program.

Conclusion

To summarize each program:

It was noticed that the program 'hn__4.3.03__va_nFXnp12' used a very large amount of CPU utilization on average, and therefore would suffice from having a larger number of cores to run the program within the CPU and should not be decreased at all (or execution can become too slow). In terms of the memory for this program, we can easily decrease the amount of memory available by a large amount, as the memory used is insignificant in comparison to memory used by background operations.

For the program 'prog1' the total amount of CPU usage was very small, and shared among the cores at most times, only using 1 core at some points, and therefore can easily still be run on a machine with a smaller number of available cores. However, this program used a large amount of memory, so this cannot be reduced significantly, and memory concerns is valid on most machines available, if running this program, is this will most likely cause a memory error, due to lack of memory.

Finally, the program 'test' also uses a very small amount of CPU but appears to only use the entirety of one core for its entire runtime. This does mean that there is less need of a CPU with many cores and so does not need to be improved. The analysis is similar with the maximum memory available, and therefore we can still run the program with a much lower number of available memory.

To conclude, the data recording and the final analysis within this project appears to have gone accurately, however, had I more time to perform this project, there are a few things I would have done differently. One such example would be the amount of data collected. If given more time, I would have increased the amount of testing to around 100 runs of each container. This would have taken a much longer time, but would have also yielded in more precise results, in the case of random error. Another example would have been to perform the tests on multiple computer clients and present the results in a different table for each computer tested. This would then be used to evaluate the total conclusion of each program's set of data, providing a comparison between them, and would reduce the issues that could result due to systematic error.

[1] *What are containers? | Google Cloud* (no date) *Google*. Available at: https://cloud.google.com/learn/what-are-containers (Accessed: 15 November 2023).

[2] Provided by Neuquino from https://stackoverflow.com/questions/2441046/how-to-get-the-total-physical-memory-in-bash-to-assign-it-to-a-variable