

System vs. OS Virtualization Report

COEN 241: Homework 1

Prajwal Bellignur

Table of Contents

Introduction	3
Configuration Details.....	4
Configuration	5
Main Steps to Enable QEMU VM and Docker Container	5
QEMU	5
Docker	5
Proof of Experiment.....	7
QEMU running environment screenshots:.....	7
Docker running environment screenshot:	8
Measurements for Both Virtualization Technologies.....	9
Shell Script for the Experiments	10
Performance Data and Analysis.....	11
Data:.....	11
Analysis:	12

Introduction

In this assignment I use and understand the real-world use cases of two different virtualization technologies. For the first virtualization technology, I learned system virtualization through QEMU which is an open-source hypervisor. For this, I started off by installing Ubuntu 20.04 and QEMU. Next, I installed sysbench to run different tests which demonstrate the performance of the technology using benchmarks. To do this, I wrote a bash script that automated different experiments. Similarly, I also learned OS virtualization through Docker which is a container management platform. For this, I started off by installing Docker on my system. From there, I installed the Ubuntu 20.04 base image and created my own image by adding sysbench to the base image. I used the same bash script that I wrote for QEMU to run the sysbench experiments on this technology. Lastly, I compared and analyzed the results that I got from all the experiments.

Configuration Details

For the QEMU VM, the configurations are as follows:

```
pbellignur@ubuntu:~/COEN-241/hw1$ sudo lshw -short
```

H/W path	Device	Class	Description
		system	Standard PC (i440FX + PIIX, 1996)
/0		bus	Motherboard
/0/0		memory	96KiB BIOS
/0/400		processor	QEMU Virtual CPU version 2.5+
/0/1000		memory	2046MiB System Memory
/0/1000/0		memory	2046MiB DIMM RAM
/0/100		bridge	440FX - 82441FX PMC [Natoma]
/0/100/1		bridge	82371SB PIIX3 ISA [Natoma/Triton II]
/0/100/1.1		storage	82371SB PIIX3 IDE [Natoma/Triton II]
/0/100/1.3		bridge	82371AB/EB/MB PIIX4 ACPI
/0/100/2		display	VGA compatible controller
/0/100/3	ens3	network	82540EM Gigabit Ethernet Controller
/0/1		input	PnP device PNP0303
/0/2		input	PnP device PNP0f13
/0/3		storage	PnP device PNP0700
/0/4		printer	PnP device PNP0400
/0/5		communication	PnP device PNP0501
/0/6		system	PnP device PNP0b00
/0/7	scsi0	storage	
/0/7/0.0.0	/dev/sda	disk	10GB QEMU HARDDISK
/0/7/0.0.0/1	/dev/sda1	volume	1023KiB BIOS Boot partition
/0/7/0.0.0/2	/dev/sda2	volume	1792MiB EXT4 volume
/0/7/0.0.0/3	/dev/sda3	volume	8445MiB EFI partition
/0/8	scsi1	storage	
/0/8/0.0.0	/dev/cdrom	disk	QEMU DVD-ROM

For Docker, the configurations are as follows:

- Kernel Version: 5.15.49-linuxkit
- Operating System: Docker Desktop
- OS Type: Linux
- Architecture: x86_64
- CPUs: 4
- Total Memory: 3.841GiB
- Name: docker-desktop

Configuration

Main Steps to Enable QEMU VM and Docker Container

The main steps to enable QEMU VM are explained below (with the assumption that QEMU has never been installed on the system which is a native MacOS system):

QEMU

- brew install qemu
- sudo qemu-img create ubuntu.img 10G -f qcow2
- sudo qemu-system-x86_64 -hda ubuntu.img -boot d -cdrom ./ubuntu-20.04.5-live-server-amd64.iso -m 2046 -boot strict=on
- **if already opened and ran QEMU VM once** → sudo qemu-system-x86_64 -hda ubuntu.img -boot d -m 2046 -boot strict=on
- follow prompts on QEMU graphical user interface (GUI) to finish setting up the VM

For Docker, the main steps are explained below (with the assumption that Docker Desktop is not installed on the native MacOS system). The highlighted commands are operations which are important and are used in addition to the basic commands in creating our own Docker image:

Docker

- download Docker.dmg file from docker website
- sudo hdiutil attach Docker.dmg
- sudo /Volumes/Docker/Docker.app/Contents/MacOS/install
- sudo hdiutil detach /Volumes/Docker
- (make sure Docker is running or open Docker Daemon)
- docker images (to check if there are any existing images)
- **docker pull** ubuntu:20.04
- vi Dockerfile (creating a dockerfile to write instructions)
 - o FROM ubuntu:20.04
 - o RUN apt-get update
 - o RUN apt-get install -y sysbench
 - o COPY h1_script.sh /
 - o ENTRYPOINT ["/h1_script.sh"]
 - o EXPOSE 3000
 - o CMD /h1_script.sh
 - o :wq (save and exit from vim)
- **docker build** -t new_image (new_image is the name of the image to create)
- docker images (check that the image has been created)
- **docker image history** (to view image history which is shown as Figure 1.0:)
- **docker run** -d b39fb298451a (last parameter is the image id. This command is used to run the container)
- **docker ps -a** (list all running and exited containers)

- **docker exec** -it <container_id> bash (container_id is shown when previous command is executed. This is used to access the running container)
- now it will be in bash mode to run the script on the container

```
Prajwals-MacBook-Pro:hw1 prajwalbellignur$ docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
new_image latest b39fb298451a 11 hours ago 134MB
Prajwals-MacBook-Pro:hw1 prajwalbellignur$ docker image history b39fb298451a
IMAGE CREATED BY SIZE COMMENT
b39fb298451a 11 hours ago CMD ["/bin/sh" "-c" "/h1_script.sh"] 0B buildkit.dockerfile.v0
<missing> 11 hours ago EXPOSE map[3000/tcp:{}] 0B buildkit.dockerfile.v0
<missing> 11 hours ago ENTRYPOINT ["/h1_script.sh"] 0B buildkit.dockerfile.v0
<missing> 11 hours ago COPY h1_script.sh / # buildkit 1.99kB buildkit.dockerfile.v0
<missing> 12 hours ago RUN /bin/sh -c apt-get install -y sysbench #... 21MB buildkit.dockerfile.v0
<missing> 12 hours ago RUN /bin/sh -c apt-get update # buildkit 40.5MB buildkit.dockerfile.v0
<missing> 6 days ago /bin/sh -c #(nop) CMD ["/bin/bash"] 0B
<missing> 6 days ago /bin/sh -c #(nop) ADD file:8b180a9b4497de0c6... 72.8MB
<missing> 6 days ago /bin/sh -c #(nop) LABEL org.opencontainers... 0B
<missing> 6 days ago /bin/sh -c #(nop) LABEL org.opencontainers... 0B
<missing> 6 days ago /bin/sh -c #(nop) ARG LAUNCHPAD_BUILD_ARCH 0B
<missing> 6 days ago /bin/sh -c #(nop) ARG RELEASE 0B
Prajwals-MacBook-Pro:hw1 prajwalbellignur$
```

Figure 1: Docker Image History

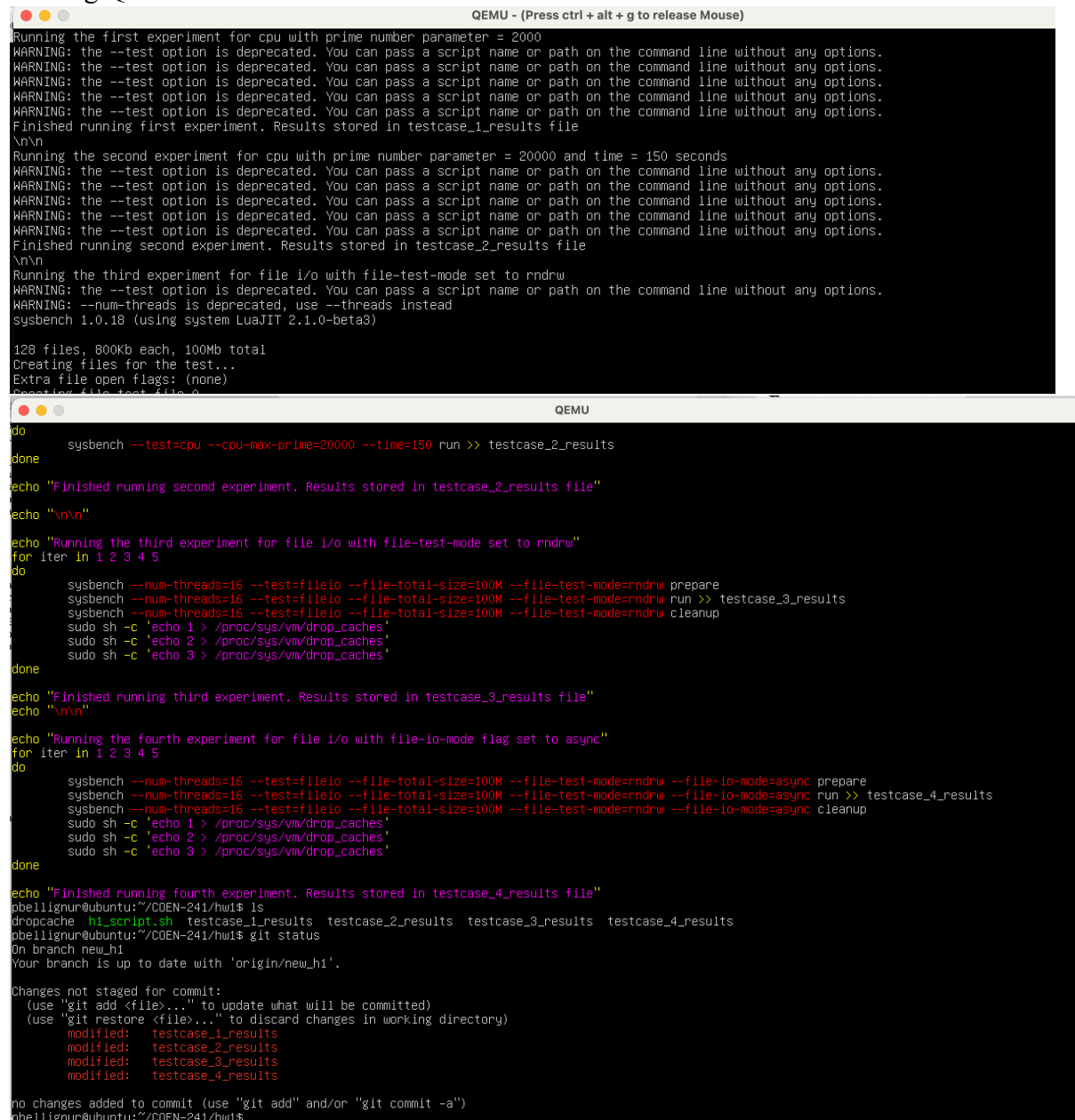
Some additional Docker operations that are used to manage Docker containers but not used above are:

- **docker stop** <container_id> (used to stop the running container)
- **docker commit** <container_id> <username/image_name> (used to create a new image of an edited container on the local system)
- **docker push** <username/image_name> (used to push an image to the docker hub repository)
- **docker rm** <container_id> (used to delete a *stopped* container)
- **docker container cp** (copy files between container and local system)
- **docker container create** (create a new Docker container)
- **docker container top** (display running process of a container)
- **docker container inspect** <container_id> (get information about a specific container)

Proof of Experiment

QEMU running environment screenshots:

For the experiments, I used a bash script which runs and outputs the results to a text file which is saved in the directory. Because of this, I do not have a screenshot of the experiments running on QEMU but have the proof of the output files in my GitHub repository. This image shows the output of my “echo” commands within the script. The second image shows the QEMU VM running on my system which shows part of my script as well as my username which is used for running QEMU.



```
QEMU - (Press ctrl + alt + g to release Mouse)
Running the first experiment for cpu with prime number parameter = 2000
WARNING: the --test option is deprecated. You can pass a script name or path on the command line without any options.
WARNING: the --test option is deprecated. You can pass a script name or path on the command line without any options.
WARNING: the --test option is deprecated. You can pass a script name or path on the command line without any options.
WARNING: the --test option is deprecated. You can pass a script name or path on the command line without any options.
WARNING: the --test option is deprecated. You can pass a script name or path on the command line without any options.
Finished running first experiment. Results stored in testcase_1_results file
\n\n
Running the second experiment for cpu with prime number parameter = 20000 and time = 150 seconds
WARNING: the --test option is deprecated. You can pass a script name or path on the command line without any options.
WARNING: the --test option is deprecated. You can pass a script name or path on the command line without any options.
WARNING: the --test option is deprecated. You can pass a script name or path on the command line without any options.
WARNING: the --test option is deprecated. You can pass a script name or path on the command line without any options.
WARNING: the --test option is deprecated. You can pass a script name or path on the command line without any options.
Finished running second experiment. Results stored in testcase_2_results file
\n\n
Running the third experiment for file i/o with file-test-mode set to rndrw
WARNING: the --test option is deprecated. You can pass a script name or path on the command line without any options.
WARNING: --num-threads is deprecated, use --threads instead
sysbench 1.0.18 (using system LuaJIT 2.1.0-beta3)

128 files, 800Kb each, 100Mb total
Creating files for the test...
Extra file open flags: (none)
Created file testfile_0

do
    sysbench --test=cpu --cpu-max-prime=20000 --time=150 run >> testcase_2_results
done
echo "Finished running second experiment. Results stored in testcase_2_results file"
echo "\n\n"
echo "Running the third experiment for file i/o with file-test-mode set to rndrw"
for iter in 1 2 3 4 5
do
    sysbench --num-threads=16 --test=fileio --file-total-size=100M --file-test-mode=rndrw prepare
    sysbench --num-threads=16 --test=fileio --file-total-size=100M --file-test-mode=rndrw run >> testcase_3_results
    sysbench --num-threads=16 --test=fileio --file-total-size=100M --file-test-mode=rndrw cleanup
    sudo sh -c 'echo 1 > /proc/sys/vm/drop_caches'
    sudo sh -c 'echo 2 > /proc/sys/vm/drop_caches'
    sudo sh -c 'echo 3 > /proc/sys/vm/drop_caches'
done
echo "Finished running third experiment. Results stored in testcase_3_results file"
echo "\n\n"
echo "Running the fourth experiment for file i/o with file-io-mode flag set to async"
for iter in 1 2 3 4 5
do
    sysbench --num-threads=16 --test=fileio --file-total-size=100M --file-test-mode=rndrw --file-io-mode=async prepare
    sysbench --num-threads=16 --test=fileio --file-total-size=100M --file-test-mode=rndrw --file-io-mode=async run >> testcase_4_results
    sysbench --num-threads=16 --test=fileio --file-total-size=100M --file-test-mode=rndrw --file-io-mode=async cleanup
    sudo sh -c 'echo 1 > /proc/sys/vm/drop_caches'
    sudo sh -c 'echo 2 > /proc/sys/vm/drop_caches'
    sudo sh -c 'echo 3 > /proc/sys/vm/drop_caches'
done
echo "Finished running fourth experiment. Results stored in testcase_4_results file"
pbellignur@ubuntu:~/COEN-241/hw1$ ls
dropcache hi_script.sh testcase_1_results testcase_2_results testcase_3_results testcase_4_results
pbellignur@ubuntu:~/COEN-241/hw1$ git status
On branch new_hi
Your branch is up to date with 'origin/new_hi'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   testcase_1_results
        modified:   testcase_2_results
        modified:   testcase_3_results
        modified:   testcase_4_results

no changes added to commit (use "git add" and/or "git commit -a")
pbellignur@ubuntu:~/COEN-241/hw1$
```

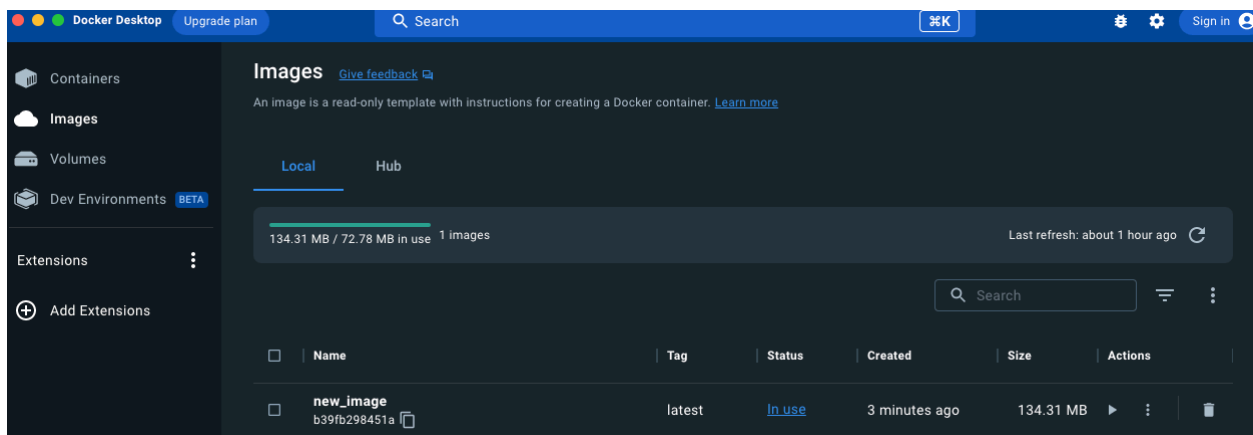
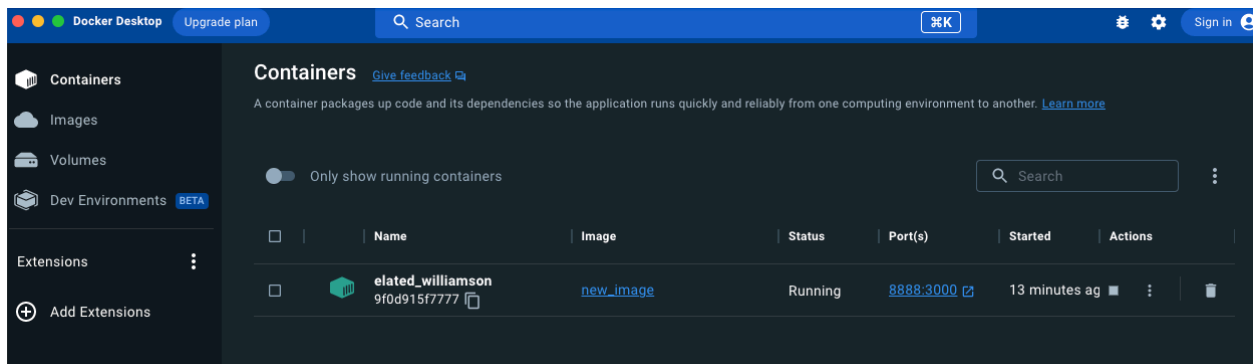
The Docker proof of experiment is shown on the next page:

Docker running environment screenshot:

Similar to QEMU, I used a bash script which runs and outputs the results to a text file which is saved in the container directory. Because of this, I am not able to get the screenshot of the experiment running, but the output files are uploaded on the GitHub for proof. To get this I used the docker cp <container_id>:/ . command and copied the contents of the container to the local directory. The following images show the docker cli and the docker desktop running with my image. In addition, it also shows the Docker container.

```
Prajwals-MacBook-Pro:hw1 prajwalbellignur$ docker build -t new_image .
[+] Building 1.1s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 198B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/ubuntu:20.04
=> [internal] load build context
=> => transferring context: 34B
=> [1/4] FROM docker.io/library/ubuntu:20.04@sha256:b33325a00c7c27b23ae48cf17d2c654e2c30812b35e7846c00
=> => resolve docker.io/library/ubuntu:20.04@sha256:b33325a00c7c27b23ae48cf17d2c654e2c30812b35e7846c00
=> CACHED [2/4] RUN apt-get update
=> CACHED [3/4] RUN apt-get install -y sysbench
=> CACHED [4/4] COPY h1_script.sh /
=> exporting to image
=> => exporting layers
=> => writing image sha256:b39fb298451a103656a67343a56cb339de986726e526d2449b86b39821e2a6ae
=> => naming to docker.io/library/new_image

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
Prajwals-MacBook-Pro:hw1 prajwalbellignur$ docker run -p 8888:3000 new_image
```



Measurements for Both Virtualization Technologies

For the experiments, I split the measurements into four different scenarios. These four scenarios were kept the same for both QEMU and Docker environments to be able to compare and analyze the results at the end. The four experiments are described below:

- **First experiment:** This experiment focused on the cpu test mode where it tests the system when the `--cpu-max-prime` was set to 2000. I specifically chose this number because when I ran this `sysbench` command when the flag was set 10, 50, and 100 the results for the statistics were too small and were not useful to get a good understanding of how the system is performing. To ensure that there were usable results, I set the flag to 2000.
- **Second experiment:** In this experiment, I used the same `--cpu-max-prime` parameter but set it to 20000. In addition, I also added a time parameter and set it to 150 seconds. I chose both these parameters to see how it would affect the system performance if both flags were set and if the time was set to something much higher than the default 10 seconds from the previous experiment.
- **Third experiment:** This experiment focused on the file-io test mode where it tests the I/O performance of the system. To run this experiment, I ran the `sysbench --num-threads=16 --test=fileio --file-total-size=100M --file-test-mode=rndrw prepare` command followed by the same command ending in `run`. In this, the `num-threads` flag was set to 16, `file-total-size` flag was set to 100M, and the `file-test-mode` flag was set to `rndrw`. The `file-test-mode` flag is for the type of workload to produce, which in this case is `rndrw` which is combined random read/write. For the `file-total-size` I went with 100M because I initially tried it with 3G, 2G, and 1G but all those file sizes were too big for my environment as it ran out of space when the 128 sample files were created. So, I concluded that 100M was the best size to run this experiment.
- **Fourth experiment:** For this experiment, it was similar to the third experiment, but with an additional flag. The `sysbench` command I ran for this experiment was: `sysbench --num-threads=16 --test=fileio --file-total-size=100M --file-test-mode=rndrw --file-io-mode=async prepare` followed by the same command but with `run` at the end. The additional flag was the `file-io-mode` which is the I/O mode. I set this flag to `async` whereas in the previous experiment, the default was set to `sync`. The difference is that the default has the threads run synchronously whereas in this experiment they run asynchronously. I left the other parameters the exact same so that I can compare the difference in I/O performance when the `file-io-mode` is synchronous versus asynchronous.

With these four experiments, I combined them into one bash script and ran it all at once. Each experiment outputs the results of all the performances into an output file. I ran the bash script on QEMU first and then ran the bash script on the Docker container. Each output file for both virtualization technologies can be found in the `hw1` folder in the GitHub repository.

Shell Script for the Experiments

All experiments are combined into one bash script which is named h1_script.sh. This bash script can be found in the GitHub repository. This is the image of the script:

```
1  #!/bin/bash
2
3  clear
4  echo "Running the first experiment for cpu with prime number parameter = 2000"
5  for iter in 1 2 3 4 5
6  do
7      sysbench --test=cpu --cpu-max-prime=2000 run >> testcase_1_results &
8      top -n 1 -u pbellignur > top_tc1
9  done
10
11 echo "Finished running first experiment. Results stored in testcase_1_results file"
12
13 echo "\n\n"
14
15 echo "Running the second experiment for cpu with prime number parameter = 20000 and time = 150 seconds"
16 for iter in 1 2 3 4 5
17 do
18     sysbench --test=cpu --cpu-max-prime=20000 --time=150 run >> testcase_2_results &
19     top -n 1 -u root > top_tc2
20 done
21
22 echo "Finished running second experiment. Results stored in testcase_2_results file"
23
24 echo "\n\n"
25
26 echo "Running the third experiment for file i/o with file-test-mode set to rndrw"
27 for iter in 1 2 3 4 5
28 do
29     sysbench --num-threads=16 --test=fileio --file-total-size=100M --file-test-mode=rndrw prepare
30     sysbench --num-threads=16 --test=fileio --file-total-size=100M --file-test-mode=rndrw run >> testcase_3_results
31     sysbench --num-threads=16 --test=fileio --file-total-size=100M --file-test-mode=rndrw cleanup
32     sudo sh -c 'echo 1 > /proc/sys/vm/drop_caches'
33     sudo sh -c 'echo 2 > /proc/sys/vm/drop_caches'
34     sudo sh -c 'echo 3 > /proc/sys/vm/drop_caches'
35 done
36
37 echo "Finished running third experiment. Results stored in testcase_3_results file"
38 echo "\n\n"
39
40 echo "Running the fourth experiment for file i/o with file-io-mode flag set to async"
41 for iter in 1 2 3 4 5
42 do
43     sysbench --num-threads=16 --test=fileio --file-total-size=100M --file-test-mode=rndrw --file-io-mode=async prepare
44     sysbench --num-threads=16 --test=fileio --file-total-size=100M --file-test-mode=rndrw --file-io-mode=async run >> testcase_4_results
45     sysbench --num-threads=16 --test=fileio --file-total-size=100M --file-test-mode=rndrw --file-io-mode=async cleanup
46     sudo sh -c 'echo 1 > /proc/sys/vm/drop_caches'
47     sudo sh -c 'echo 2 > /proc/sys/vm/drop_caches'
48     sudo sh -c 'echo 3 > /proc/sys/vm/drop_caches'
49 done
50
51 echo "Finished running fourth experiment. Results stored in testcase_4_results file"
52 echo "\n\n"
```

Performance Data and Analysis

Data:

The following image is for the performance data for QEMU for all four experiments:

A	B	C	D	E	F	G
QEMU						
	Experiment 1	1st Iteration	2nd Iteration	3rd Iteration	4th Iteration	5th Iteration
	total time (s)	10.0265	10.0471	10.0111	10.0125	10.026
latency (ms)	min	0.28	0.28	0.27	0.28	0.28
	avg	2.17	2.64	2.82	2.98	3.05
	max	80.08	71.45	63.8	78.28	94.18
	95th percentile	16.71	23.52	28.16	27.66	29.19
	Experiment 2	1st Iteration	2nd Iteration	3rd Iteration	4th Iteration	5th Iteration
	total time (s)	150.04	150.0076	150.0016	150.0118	153.6487
latency (ms)	min	6.34	6.42	6.32	6.34	6.33
	avg	102.68	100.55	112.42	95.81	90.39
	max	38301.07	15902.33	40206.33	24005.38	30432.8
	95th percentile	161.51	227.4	125.52	179.94	94.1
	Experiment 3	1st Iteration	2nd Iteration	3rd Iteration	4th Iteration	5th Iteration
	reads/s	711.18	78.34	689.66	1001.64	64.29
	writes/s	474.12	52.23	459.93	667.73	42.95
throughput	read (MiB/s)	11.11	1.22	10.78	15.65	1
	written (MiB/s)	7.41	0.82	7.19	10.43	0.67
	total time (s)	10.6281	51.3103	10.6978	10.4909	18.608
latency (ms)	min	0.01	0.02	0.01	0.01	0.02
	avg	5.54	19.46	5.68	3.97	52.1
	max	86.08	13100.4	273.5	81.26	7351.89
	95th percentile	18.95	20	18.95	13.22	97.55
	Experiment 4	1st Iteration	2nd Iteration	3rd Iteration	4th Iteration	5th Iteration
	reads/s	8.26	745.13	1194.39	1003.1	1021.74
	writes/s	11.09	489.02	804.11	721.51	692.87
throughput	read (MiB/s)	0.13	11.64	18.66	15.67	15.96
	written (MiB/s)	0.17	7.64	12.56	11.27	10.83
	total time (s)	15.4968	10.7693	10.6039	10.606	10.9578
latency (ms)	min	0.03	0.02	0.02	0.02	0.02
	avg	369.69	7.46	4.6	5.31	5.08
	max	10207.62	224.23	105.3	140.06	102.6
	95th percentile	189.93	33.72	20.37	23.1	22.69

The following image is for the performance data for Docker for all four experiments:

Docker						
	Experiment 1	1st Iteration	2nd Iteration	3rd Iteration	4th Iteration	5th Iteration
	total time (s)	10.0005	10.0006	10.0001	10.0007	10.0002
latency (ms)	min	0.09	0.09	0.09	0.09	0.09
	avg	0.42	0.48	0.5	0.45	0.33
	max	60.49	100.98	75.25	64.1	48.92
	95th percentile	1.12	1.47	1.58	1.5	0.77
	Experiment 2	1st Iteration	2nd Iteration	3rd Iteration	4th Iteration	5th Iteration
	total time (s)	150.0406	150.0076	150.0016	150.0118	153.6487
latency (ms)	min	0.09	0.09	0.09	0.09	0.09
	avg	0.46	0.8	0.988	0.49	0.98
	max	38301.07	15902.33	40206.33	24005.38	30432.8
	95th percentile	161.51	227.4	125.52	179.94	94.1
	Experiment 3	1st Iteration	2nd Iteration	3rd Iteration	4th Iteration	5th Iteration
	reads/s	7359.04	6727.47	6388.14	5622.98	3028.36
	writes/s	4905.86	4484.98	4258.93	3748.65	2018.74
throughput	read (MiB/s)	114.98	105.12	99.81	87.86	47.32
	written (MiB/s)	76.65	70.08	66.55	58.57	31.54
	total time (s)	10.0517	10.0411	10.0574	10.237	10.1231
latency (ms)	min	0	0	0	0	0
	avg	0.57	0.62	0.65	0.73	1.37
	max	94.06	101.74	52.2	217.03	186.25
	95th percentile	1.79	1.93	2.07	2.07	5.18
	Experiment 4	1st Iteration	2nd Iteration	3rd Iteration	4th Iteration	5th Iteration
	reads/s	6941.95	5423.74	3751.13	5915.7	6587.27
	writes/s	4669.39	3652.49	2532.87	3979.16	4438.82
throughput	read (MiB/s)	108.47	84.75	58.61	92.43	102.93
	written (MiB/s)	72.96	57.07	39.58	62.17	69.36
	total time (s)	10.0574	10.1345	10.1193	10.0842	10.0746
latency (ms)	min	0	0	0	0	0
	avg	0.82	1.03	1.5	0.94	0.85
	max	91.97	114.88	277.12	130.12	122.05
	95th percentile	3.19	3.96	5.28	3.49	3.13

Analysis:

Looking at all the data for both QEMU and Docker, the main parameters that I reported were the total time, latency, and throughput. I also observed the results of the top command which shows the CPU usage and disk utilization for both kernel level and user level.

Looking at the overall results together, the main thing that sticks out in all the experiments is the latency and how it differs between QEMU and Docker. The latency for all the experiments went down when run on Docker as opposed to QEMU. This shows that Docker is a better system than QEMU purely based on the latency. Taking a closer look at experiments three and four for both

QEMU and Docker, we can see that the main thing that differs is the reads and writes per second and the throughput. When the experiments are run on Docker, the reads and writes per second as well as the throughput all increased compared to QEMU. This is a general trend that can be seen amongst all the iterations of the experiments. These trends can also be observed when looking at the top command result files.

As per the instructions, I divided the CPU utilization into two parts: one for user-level and one for kernel-level. To achieve this, I used the top command and added the -u flag to get the user level performance data where it was set to the username I used (pbellignur). Similarly, for the second experiment I used the top command set the -u flag to root to get the kernel level performance. Similar to the output files, I outputted the results from the top command into a top_tc1 file which contains all the results of the top command. For I/O, I looked at the different performance parameters such as latency and disk utilization which are all present in the data images above.

Looking at the parameters in the top results file which shows the outputs of disk utilization, we can conclude that the results when run on Docker are better for both the user level and kernel level results. This also adds to the hypothesis that Docker is a faster and better system than QEMU.

In conclusion, the system performance when it comes to CPU and when it comes to I/O performance, Docker performed much better than QEMU.

Extra Credit: Dockerfile provided in the GitHub Repository under COEN421/hw1/