CSEN 241 Cloud Computing

HW1

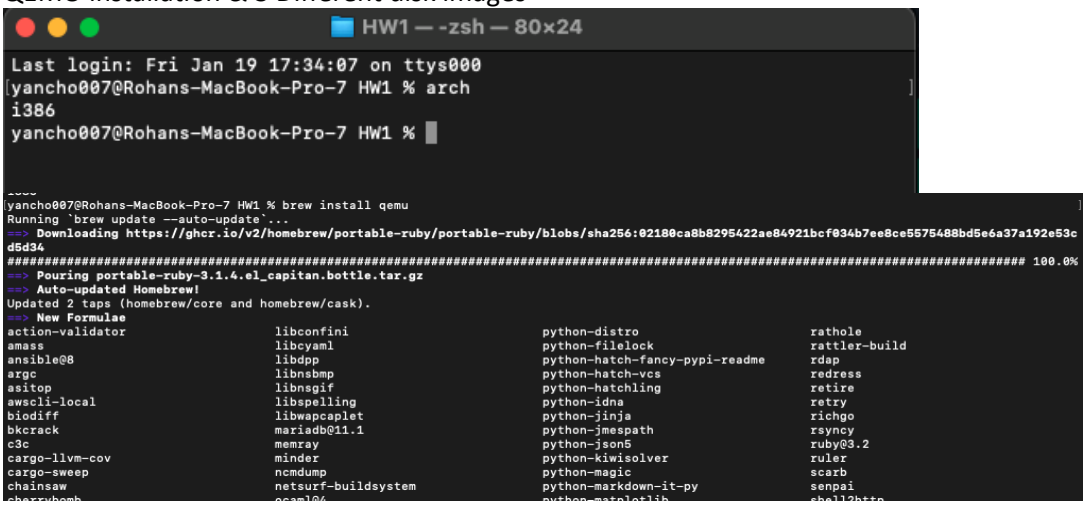
Rohan Pandey W1644990 rpandey4@scu.edu

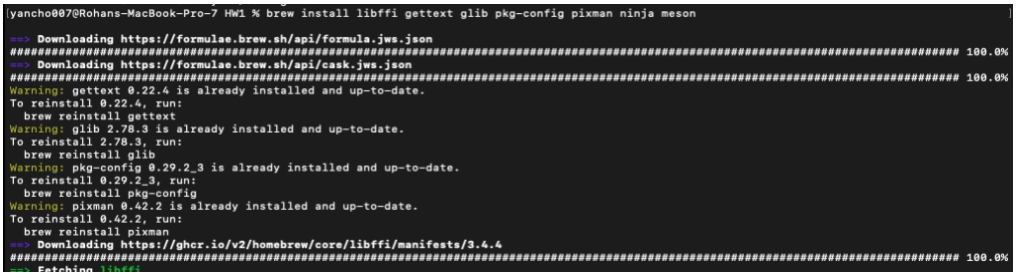
Sno	Rubric point	Data
1	Creation of three different disk images: 5 points	Qcow2, Raw, Docker
2	Present main steps to enable a QEMU VM. In addition, please present the detailed QEMU commands, and VM configurations: 10 points	Steps to follow
3	Present main steps to create the Docker container. This must show your steps in creating your own image and your image history! Do not copy any manage Docker containers (and some other operations which you think are also important): 10 points	Docker setup steps
4	Proof of experiment. Include screen snapshots of your Docker and QEMU running environments for each experiment: 10 points	Bash Scripts & Screen recording of QEMU test
5	Present your measurements in given different scenarios for each virtualization technology: 20 points	Testing Measurements & detailed analysis (+ pt6)
6	Shell scripts for running the experiment: 10 points	Test Scripts & configuration launch scripts
7	Presentation and analysis of the performance data: 20 points	Refer point 5
8	Understandability/Neatness of your report and Git: 5 points	Git
9	Extra Credit I: Run the all the QEMU experiment on disk image with encryption Use encrypt.format=luks and provide all experiment details. 5 points	Only image done
10	Extra Credit II: Provide the correct Vagrant and Docker files for your VM and Docker Container. 5 points (3 for vagrant 2 for Docker file)	NA

Steps to Run Bash scripts-

1. ***Launch the VM using launch scripts (eg. config_1_scripts_raw_launch) or direct command***
2. ***Run test cases using script "sysbench_scripts"***
3. ***Refer results in form of recording (For QEMU) , for docker results scripts are submitted & performance analysis is captured in report***

QEMO installation & 3 Different disk images-





*Steps- **Step 1**:-Making Ubuntu image (raw,qcow2, encrypted,Docker)*

1. qemu-img create ubuntu.img 20G -f qcow2
2. qemu-img create ubuntu_raw.img 20G -f raw
3. qemu-img create -f qcow2 -o encrypt.format=luks -o key-secret=mysecret -o key-size=256 ubuntu_encrypted.img 20G-

    1. This command creates an encrypted QCOW2 disk image named ubuntu_encrypted.img with a size of 10 gigabytes.
    2. The encryption is done using LUKS (Linux Unified Key Setup).
    3. key-secret=mysecret specifies the passphrase to be used for encryption.
    4. key-size=256 specifies the key size (adjust as needed).

Step 2: - Launching VM

1. Launching VM using base command & Bash scripts
2. Launching VM using 4 configurations of CPU & ram & bash scripts

Sno	CPU/smp	Memory
1	4	4096
2	8	8192
3	4	8192
4	8	4096

Step 3: -Testing

1. Testing each image on 6 test cases (2 CPU,2Memory, 2Fileio) using bash scripts

Step 4: -Result Comparison- Qcow2 Vs raw Vs encrypted Vs Docker

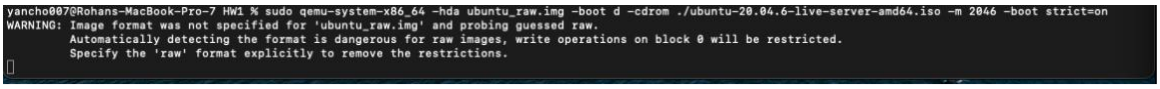
Following is the explanation for all the arguments we have used in our-

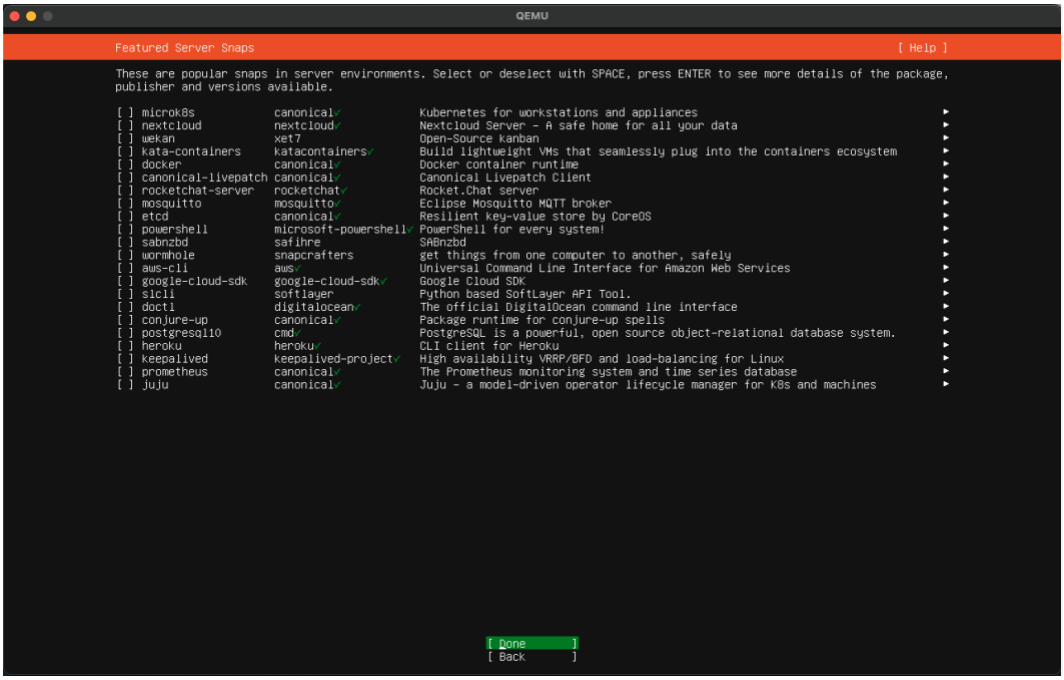
- -accel Used to enable an accelerator. Depending on the target architecture, kvm, xen, hax, hvf, nvmm, whpx or tcg can be available
- -m Stands for memory. We have currently supplied 2G as the argument value which means we are allocating 2GB RAM memory for our VM
- -smp Stands for number of cores. We have currently given the argument value as 2, which means that we have allocated 2 cores for our VM

Note: We would be updating -m and -smp values when running for different configurations.

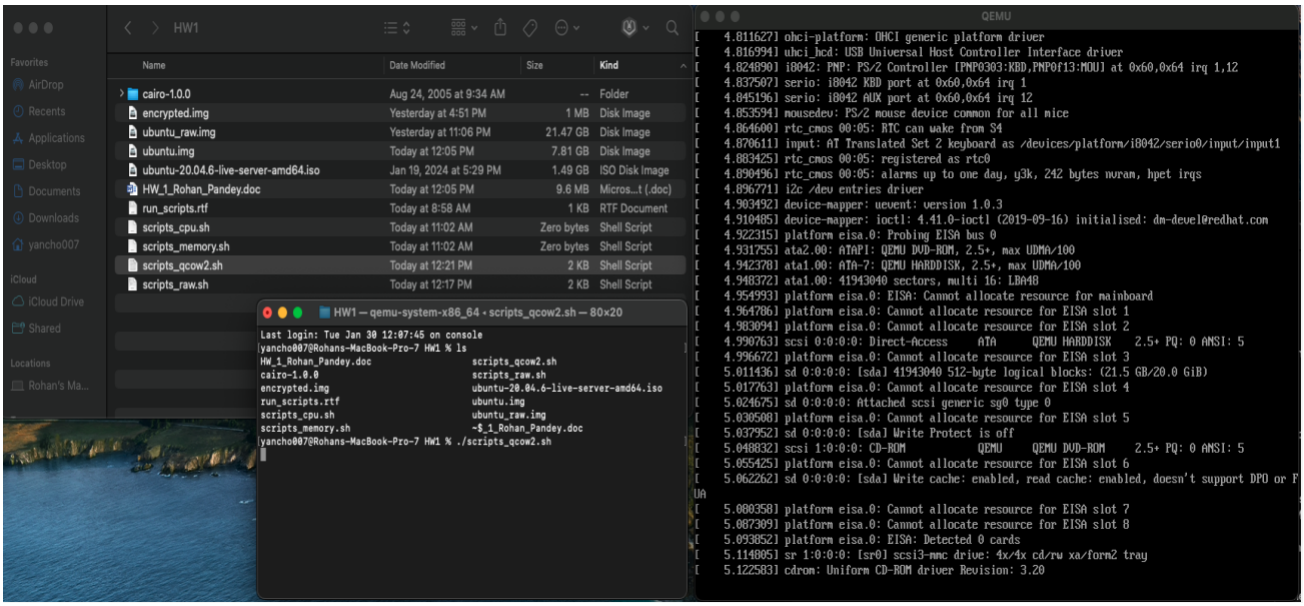
Qcow2 Image:-

Launch VM- sudo qemu-system-x86_64 -hda ubuntu.img -boot d -cdrom ./ubuntu-20.04.6-live-server-amd64.iso -m 4096 -boot strict=on





Launching VM on QEMU using bash script (included in submission)-



4 Configurations for QEMU -Ubuntu VM-

Base command-qemu-system-x86_64 -hda ubuntu.img -boot d -m 2046 -boot strict=on

We will test our image on following 4 test cases using sysbench-

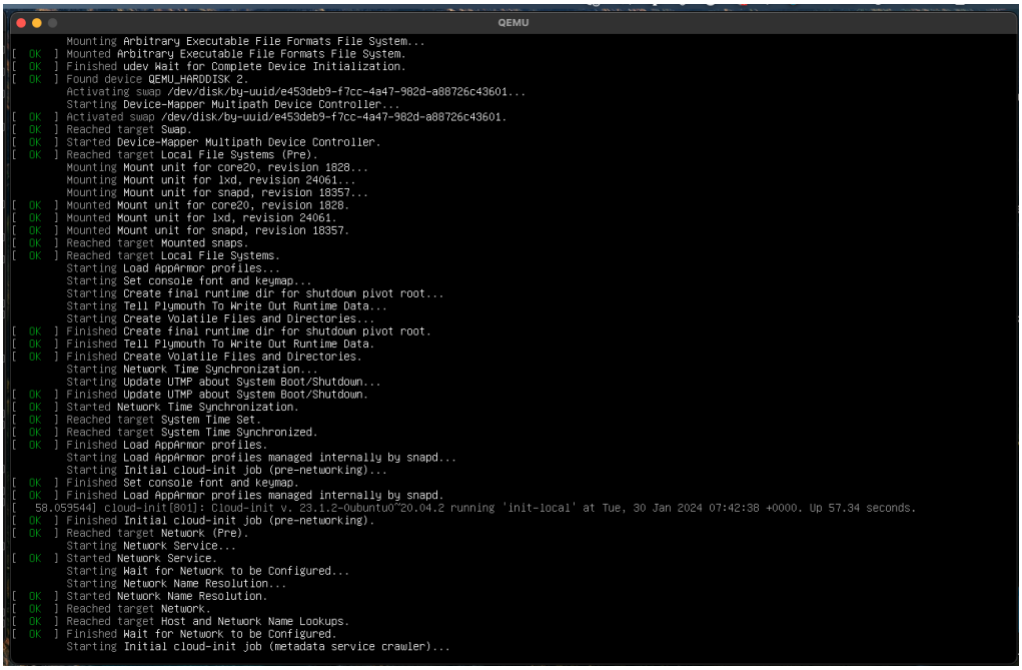
Sno	Description	CPU/smp	Memory
1	Config 1	4	4096
2	Config 2	8	8192
3	Config 3	4	8192
4	Config 4	8	4096

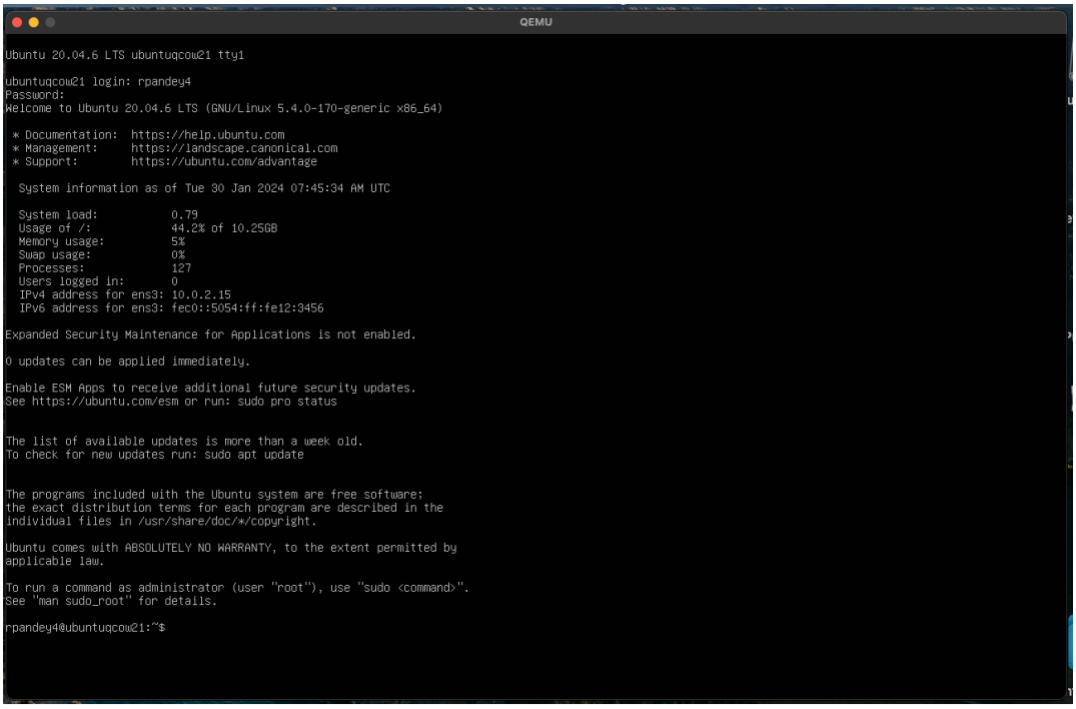
Running

Configuration-1- Launch using bash script file

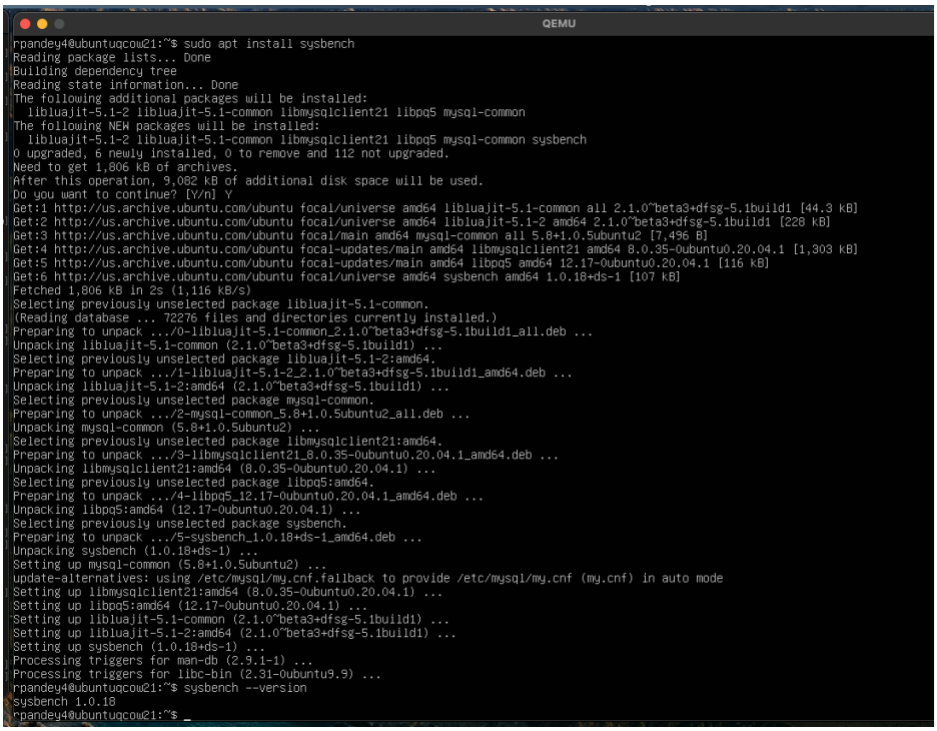
(config_1_scripts_qcow2_launch.sh) or

Command- qemu-system-x86_64 -hda ubuntu.img -boot d -m 4096 -smp 4,maxcpus=4 -boot strict=on

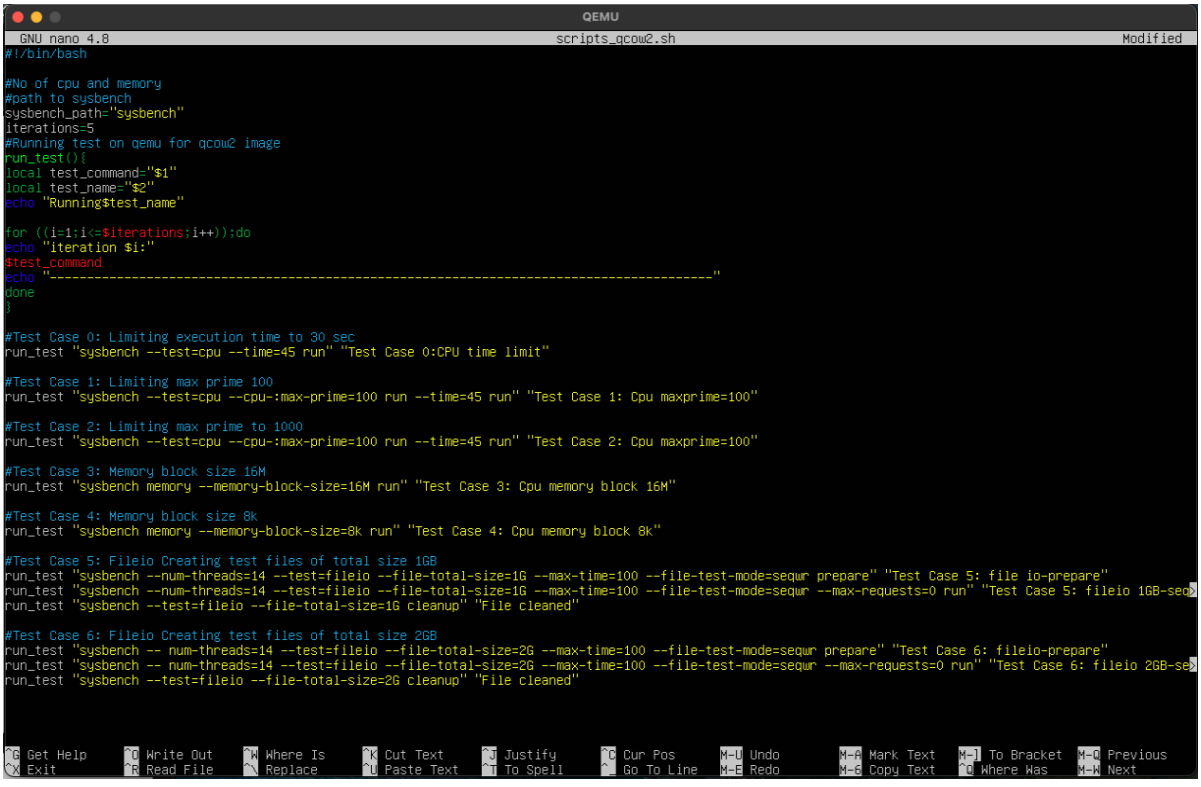




Sysbench installation & version check- sysbench -1.0.18



Generating Bash scripts with 6 test cases & 5 iterations of each



3 CPU cases-

- sysbench --test=cpu --time=45 run"
- sysbench --test=cpu –cpu-max-prime=100 --time=45 run
- sysbench --test=cpu –cpu-max-prime=1000 --tIme=45 run

2 memory cases-

- sysbench memory --memory-block-size=16M run
- sysbench memory --memory-block-size=8k run

The --memory-block-size option in Sysbench is used to define the block size for memory allocation and deallocation operations. This option is relevant when running memory-related tests with Sysbench. Below are a few test cases for --memory-block-size

2 fileio cases-

Test Case 1 (Sequential R/W) : Creating test files of total size 1GB.

sysbench --test=fileio --file-total-size=1G prepare

- sysbench --test=fileio --file-total-size=1G --file-test-mode=seqwr --max-time=250 --max-requests=0 run
- Removing test files:- sysbench --test=fileio --file-total-size=1G cleanup

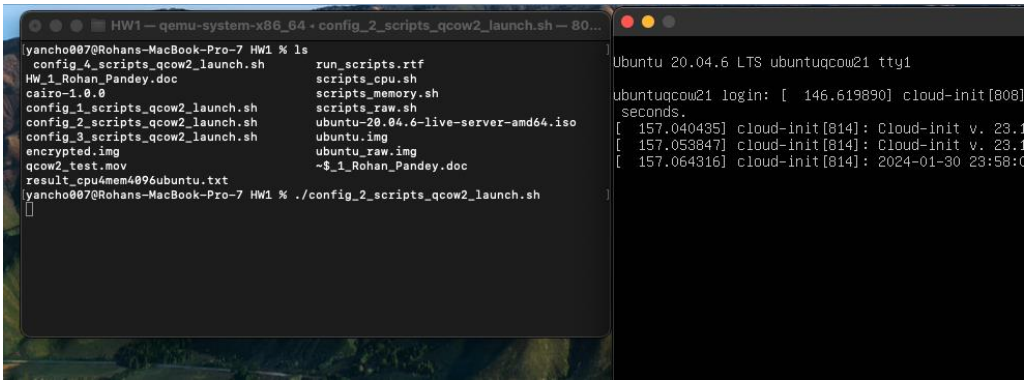
Test Case 2 (Random R/W): Creating test files of total size 2GB.

- sysbench --test=fileio --file-total-size=2G prepare
- sysbench --test=fileio --file-total-size=2G --file-test-mode=rndrw --max-time=250 --max-requests=0 run
- Removing test files:- sysbench --test=fileio --file-total-size=1G cleanup

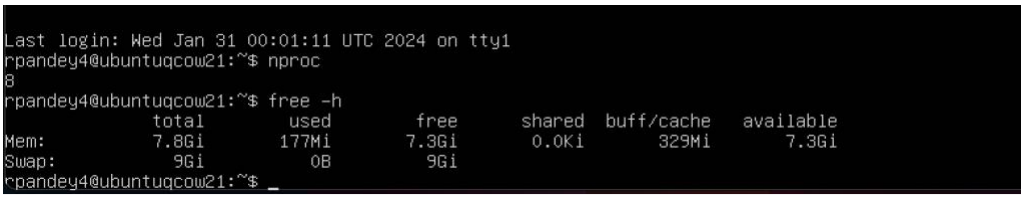
Test Results & screen recording.

Configuration-2- Launch using bash script file

Command- qemu-system-x86_64 -hda ubuntu.img -boot d -m 8192 -smp 8,maxcpus=8 -boot strict=on



Configuration verification from ubuntu VM-

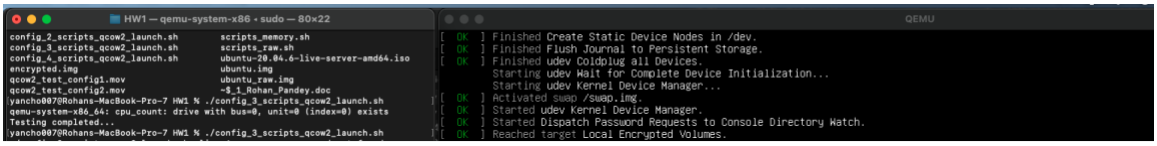


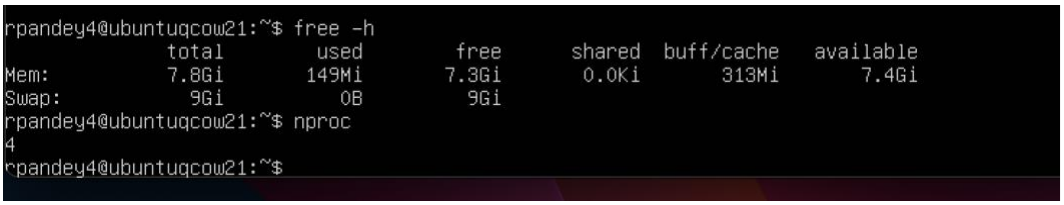
Using Sysbench bash script & launch it- run_scripts.sh

Test Results & screen recording.

Configuration-3- Launch using bash script file

Command- qemu-system-x86_64 -hda ubuntu.img -boot d -m 8192 -smp 4,maxcpus=4 -boot strict=on



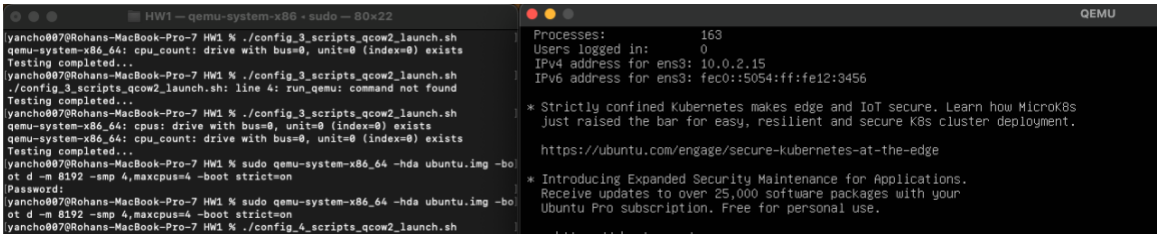


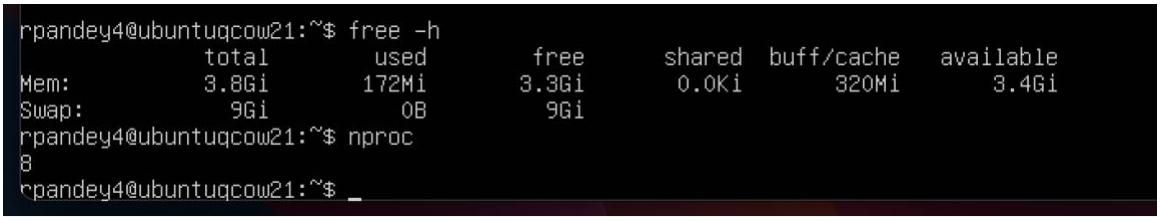
Test Results & screen recording.

Configuration-4- Launch using bash script file

qemu-system-x86_64 -hda ubuntu.img -boot d -m 4096 -smp 8,maxcpus=8 -boot strict=on

Verifying system configurations





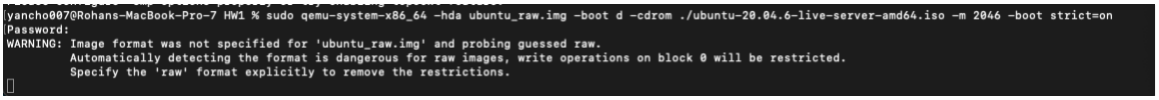
Using Sysbench bash script & launch it- run_scripts.sh

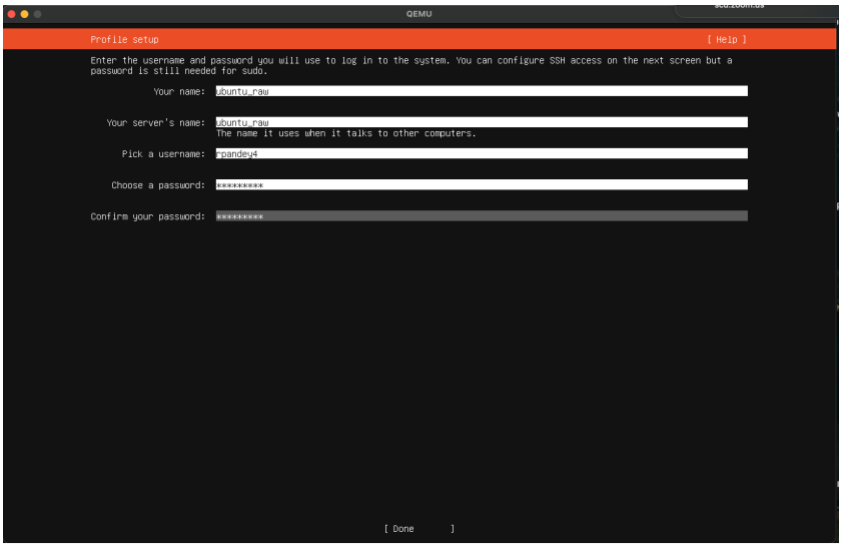
Test Results & screen recording.

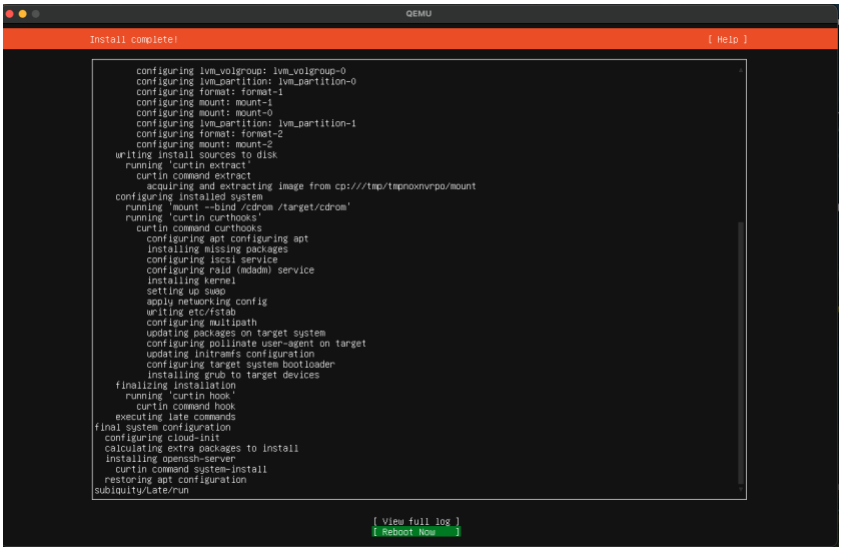
Raw Image->

Install QEMU-

Command-sudo qemu-system-x86_64 -hda ubuntu_raw.img -boot d -cdrom ./ubuntu-20.04.6-live-server-amd64.iso -m 2046 -boot strict=on







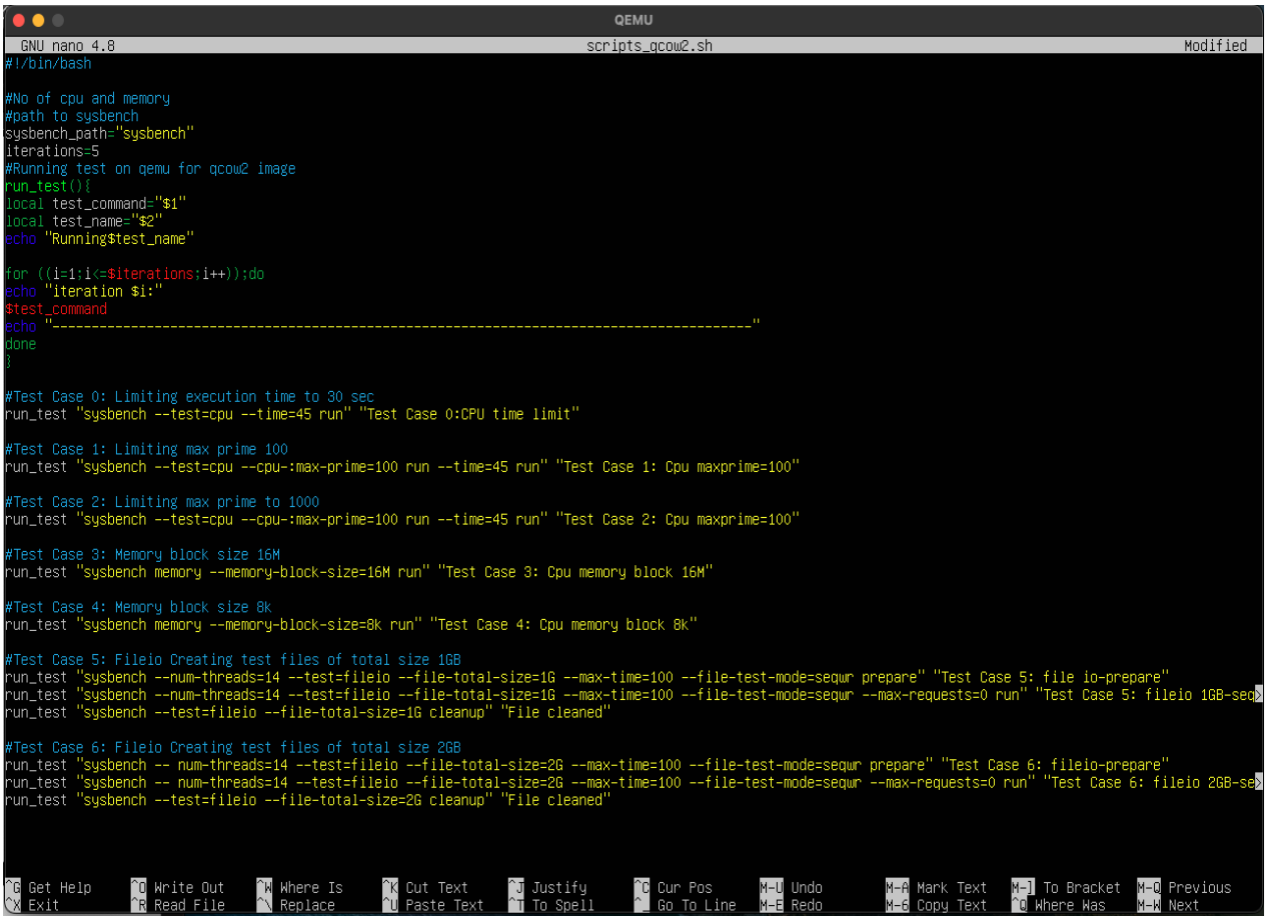
Launching VM on QEMU using bash script (included in submission)-

4 Configurations for QEMU VM-

Sno	CPU/smp	Memory
1	4	4096
2	8	8192
3	4	8192
4	8	4096

- Base Command- >sudo qemu-system-x86_64 -hda ubuntu_raw.img -boot d -m 2046 -boot strict=on

We will test our image on following 4 test cases using sysbench-



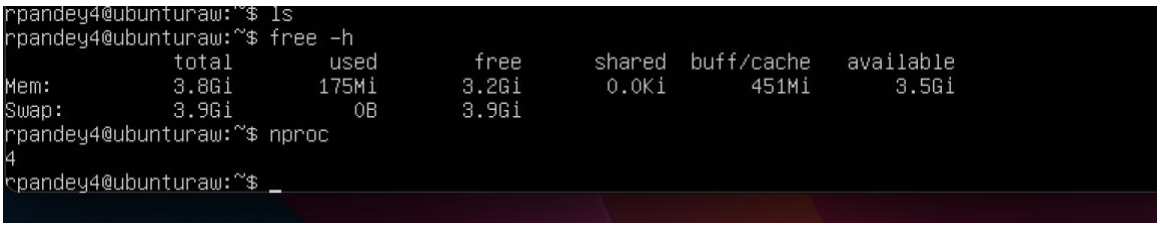
Running

Configuration 1: -

Bash script-config_1_scripts_raw_launch.sh

Command- qemu-system-x86_64 -hda ubuntu_raw.img -boot d -m 4096 -smp 4,maxcpus=4 -boot strict=on

Verify configurations of image-



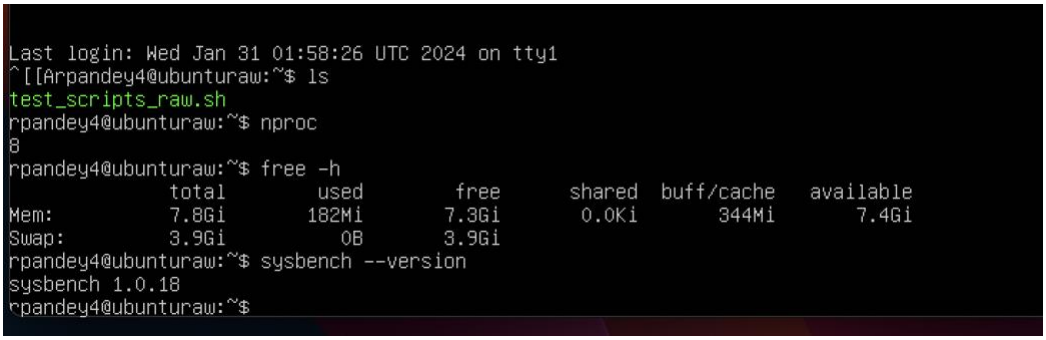
Test Results and screen recording + Bash scripts

Configuration 2: -

Bash script- config_2_scripts_raw_launch.sh

Command- sudo qemu-system-x86_64 -hda ubuntu_raw.img -boot d -m 8192 -smp 8,maxcpus=8 -boot strict=on

System verification-



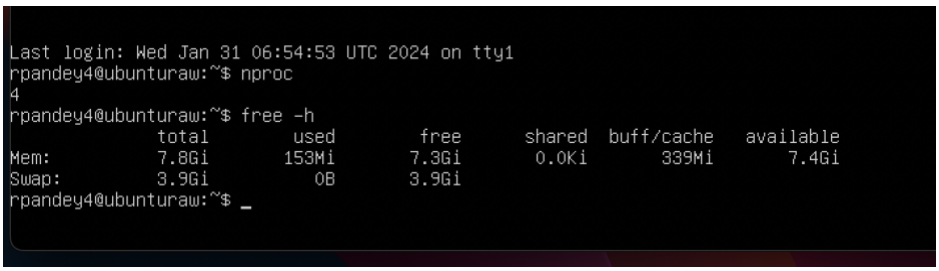
Test Results and screen recording + Bash scripts

Configuration 3: -

Bash script- config_3_scripts_raw_launch.sh

Command- qemu-system-x86_64 -hda ubuntu_raw.img -boot d -m 8192 -smp 4,maxcpus=4 -boot strict=on

System verification



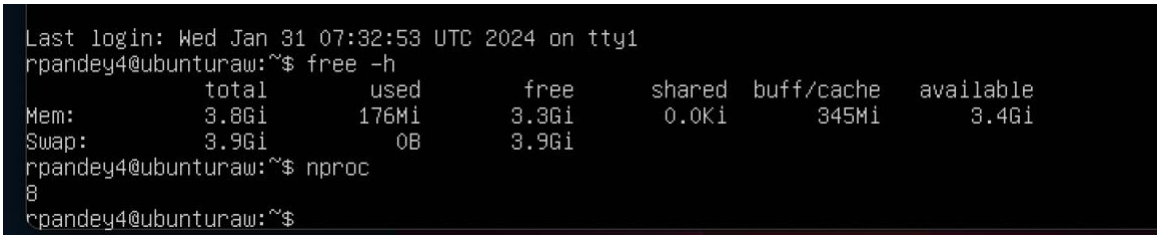
Test Results and screen recording + Bash scripts

Configuration 4: -

Bash script- config_4_scripts_raw_launch.sh

Command- qemu-system-x86_64 -hda ubuntu_raw.img -boot d -m 4096 -smp 8,maxcpus=8 -boot strict=on

System verification

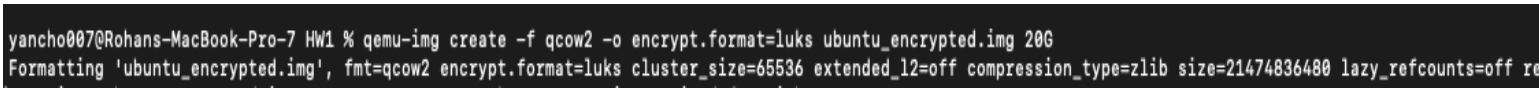


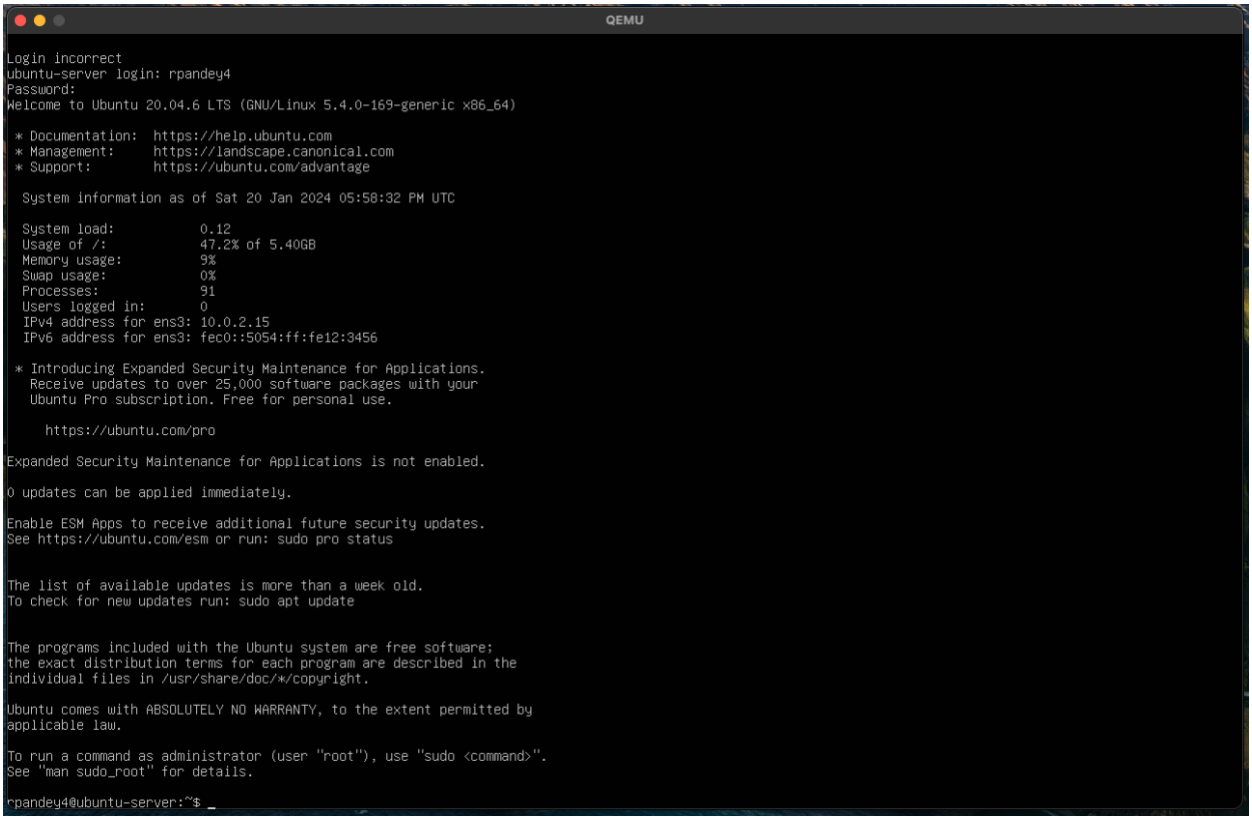
Test Results and screen recording + Bash scripts

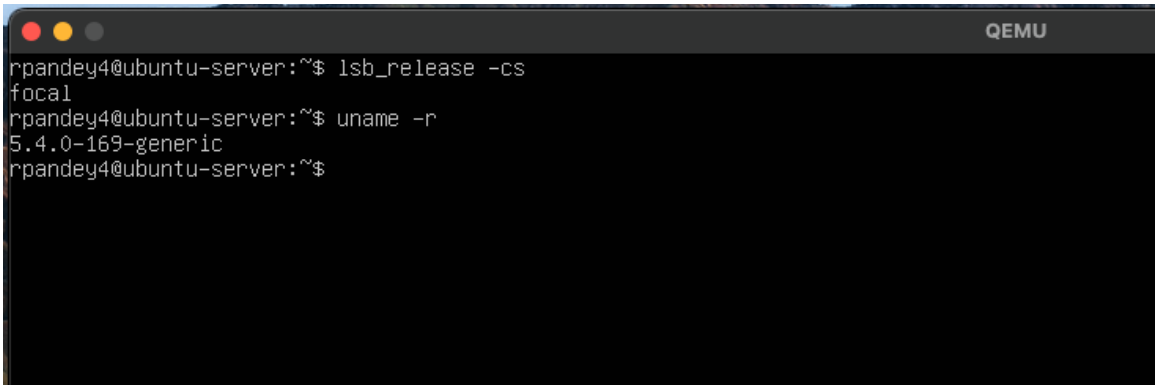
Encrypted Image->

sudo qemu-system-x86_64 -hda ubuntu_encrypted.img -boot d -cdrom ./ubuntu-20.04.6-live-server-amd64.iso -m 2046 -boot strict=on

qemu-img create -f qcow2 -0 encrypt.format=luks ubuntu_encrypted.img 20G





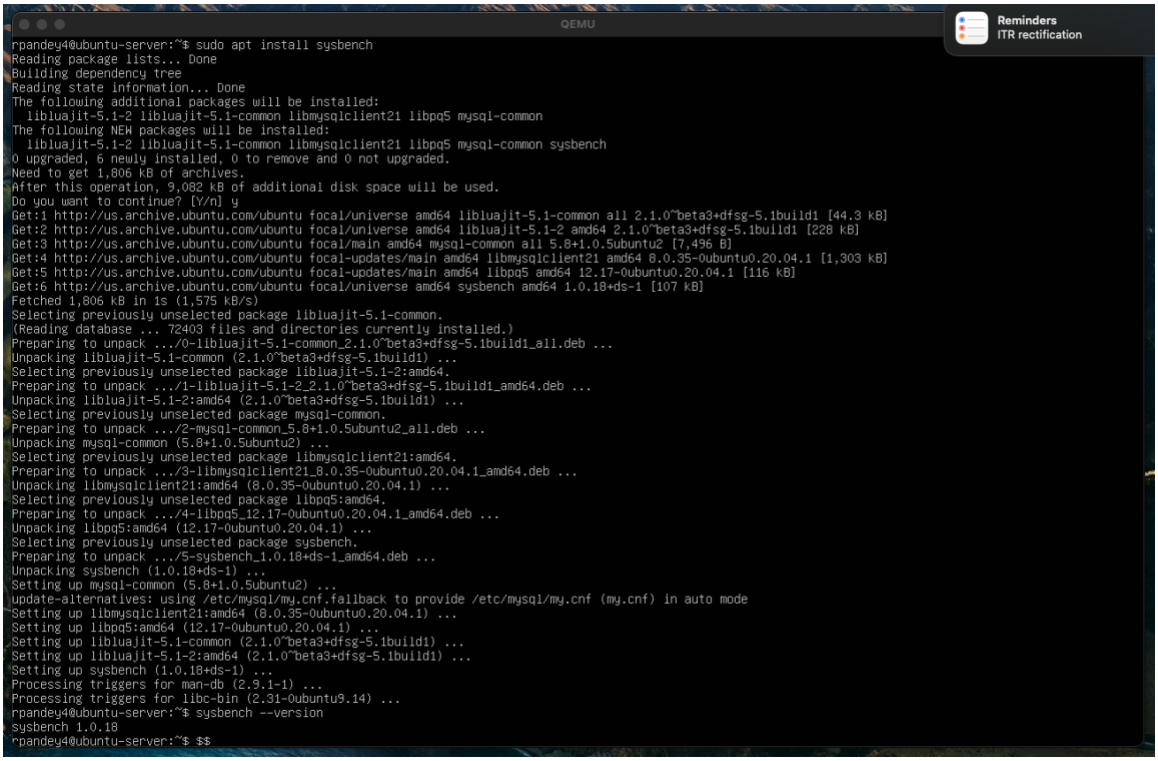


Login to ubuntu

Sysbench installed in VM Ubuntu

Docker Image->

docker pull ubuntu:22.04



Steps for Docker Setup->

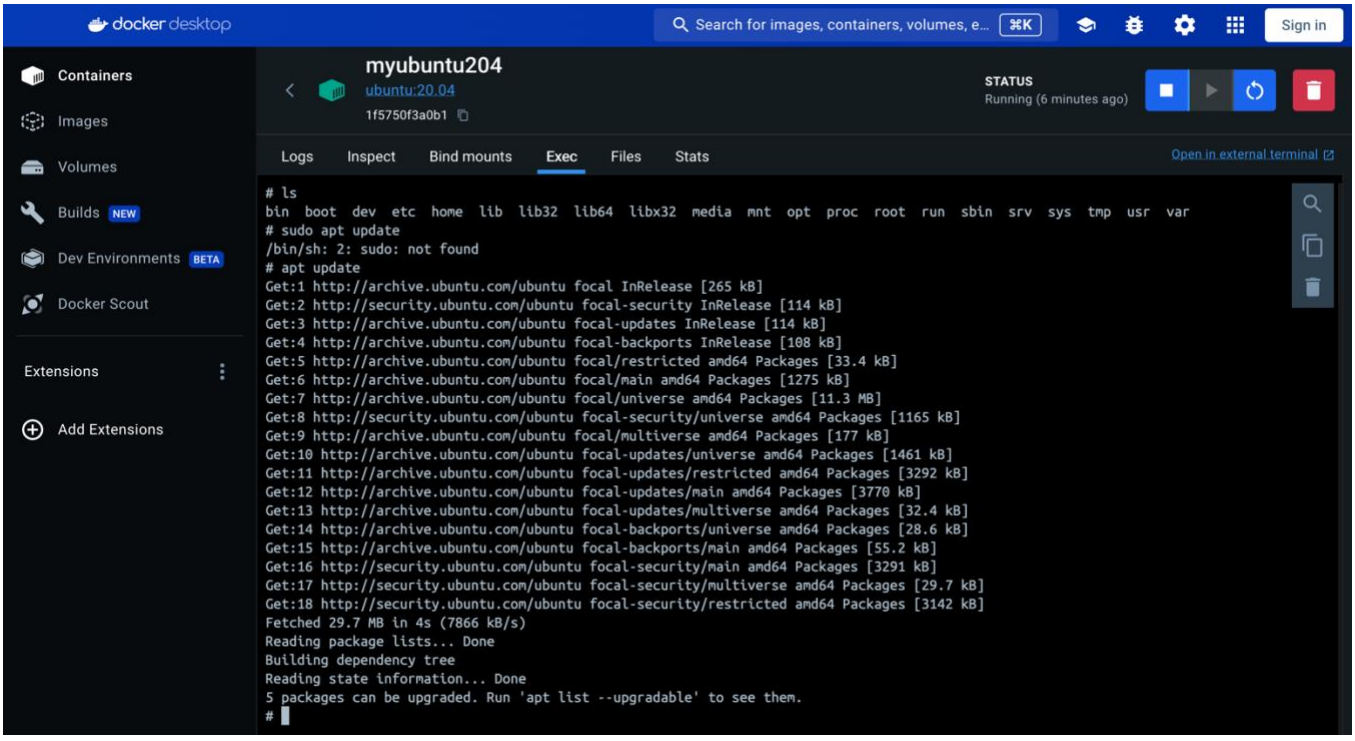
Step 1- Downloaded docker image from-https://hub.docker.com/_/ubuntu

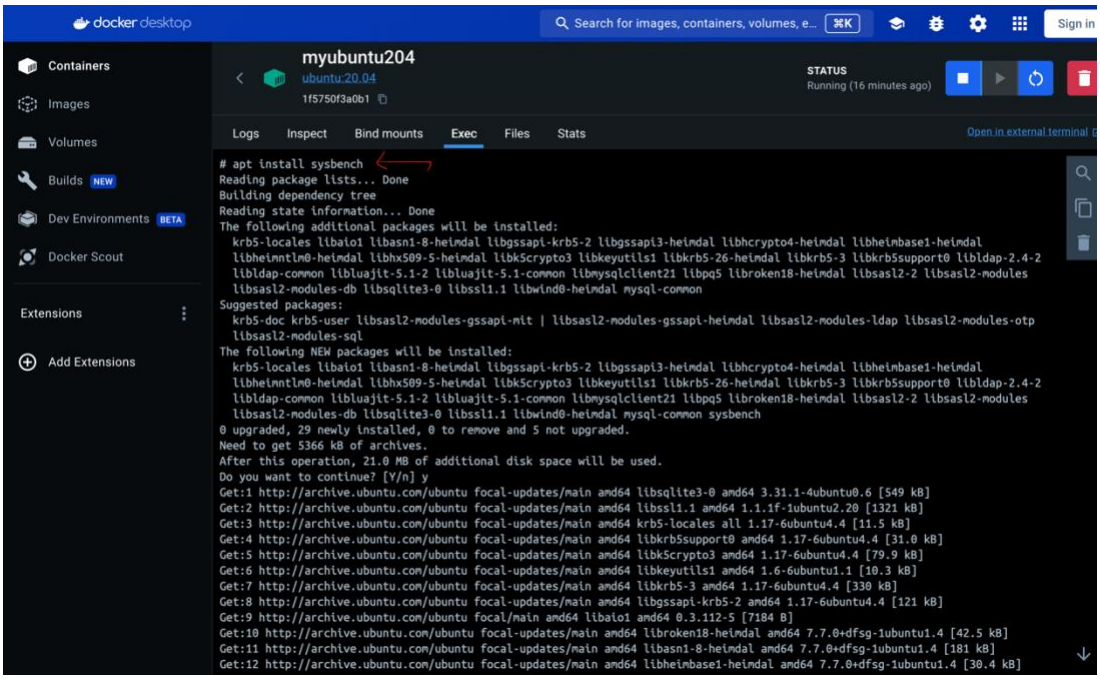
Step 2- started the docker desktop.

Step 3- Sysbench version check 1.0.18 to be installed on both.

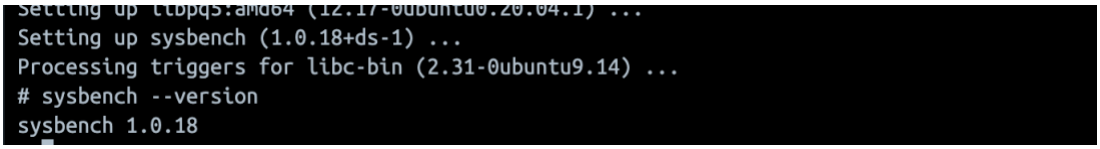
```
yancho007 — root@1f5750f3a0b1: / — com.docker.cli ◂ docker run -it --name=myubuntu204 ubuntu:20.04 /bin/bash — 125...
Last login: Fri Jan 19 17:36:07 on ttys000
yancho007@Rohans-MacBook-Pro-7 ~ %  docker pull ubuntu:20.04

20.04: Pulling from library/ubuntu
527f5363b98e: Pull complete
Digest: sha256:f2034e7195f61334e6caff6ecf2e965f92d11e888309065da85ff50c617732b8
Status: Downloaded newer image for ubuntu:20.04
docker.io/library/ubuntu:20.04

What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout quickview ubuntu:20.04
yancho007@Rohans-MacBook-Pro-7 ~ % docker run -it --name=myubuntu ubuntu /bin/bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
29202e855b20: Pull complete
Digest: sha256:e6173d4dc55e76b87c4af8db8821b1feae4146dd47341e4d431118c7dd060a74
Status: Downloaded newer image for ubuntu:latest
^C%
yancho007@Rohans-MacBook-Pro-7 ~ % docker run -it --name=myubuntu ubuntu:20.04 /bin/bash
docker: Error response from daemon: Conflict. The container name "/myubuntu" is already in use by container "7aba87855eba116d
fac058730e61fc6c8f58bc9497252891a2a1d90dc061ba9a". You have to remove (or rename) that container to be able to reuse that nam
e.
See 'docker run --help'.
yancho007@Rohans-MacBook-Pro-7 ~ % docker run -it --name=myubuntu204 ubuntu:20.04 /bin/bash
root@1f5750f3a0b1:/# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@1f5750f3a0b1:/#
```

Step 4- Exec commands in ubuntu-





Sysbench version match-



Docker Exec Syntax-

The "docker exec" syntax for accessing a container's shell is: -

```
docker exec -it <container-name-or-id> <shell-executable>
```

Here's an explanation of the fields: -

docker exec: This tells Docker to execute a command inside a container.

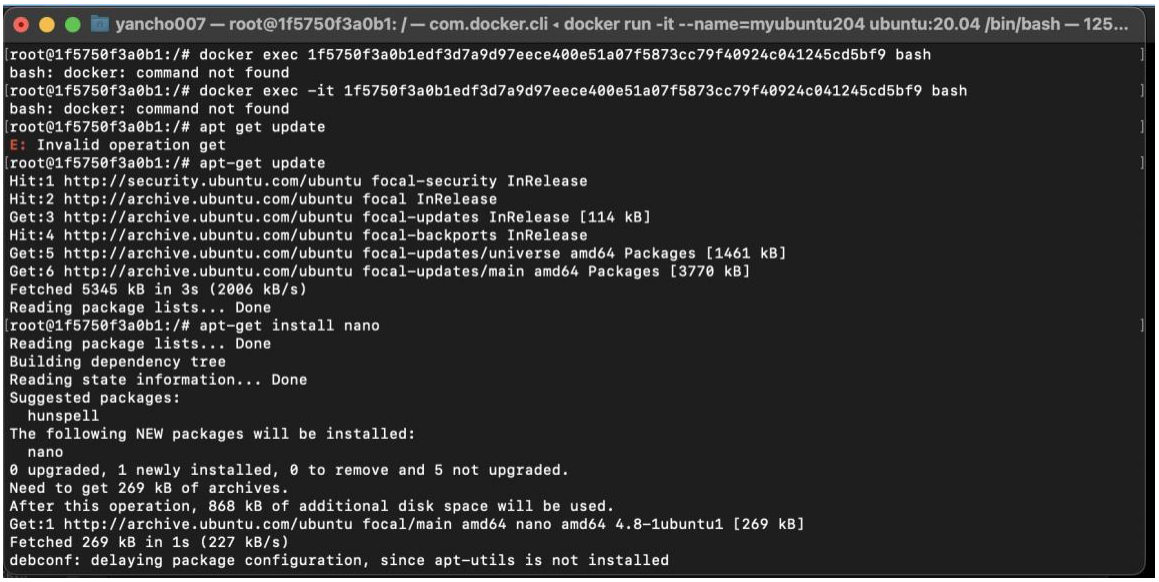
-it: These flags (-i and -t) specify that the command should be run in an interactive (-i) session with a pseudo-TTY (-t terminal) attached to the container. This allows you to enter commands and receive output from the container.

<container-name-or-id>: This is the name or ID of the container to execute the command in.

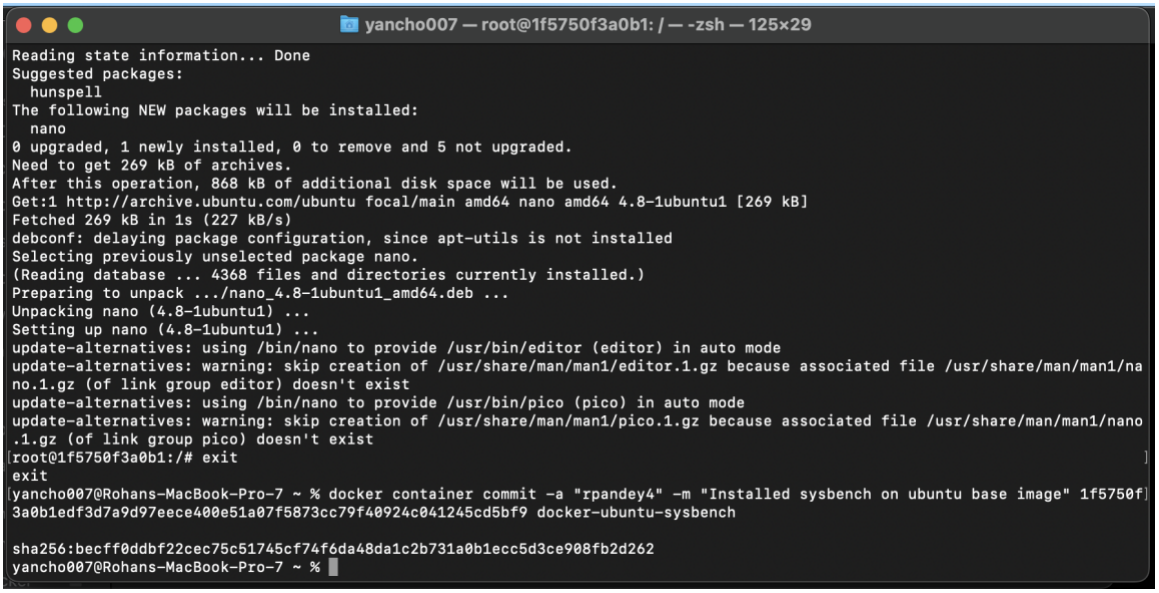
<shell-executable>: This is the name of the shell executable (a file that contains a program) that you want to use to start an interactive session inside the container. For example, you could specify **bash** to start an interactive session with the Bash shell.

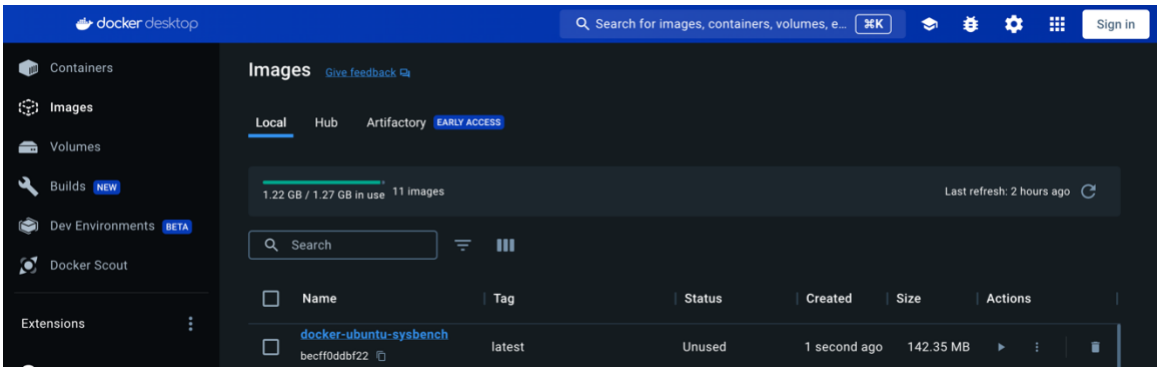
As noted above, "-it" is a combination of two separate flags, "-i", and "-t". Here's a more in-depth explanation of what they do.

Step 5- Installed nano-



Step 6-Create new image (ubuntu + sysbench)





Step 7- Created container out of image

```
yancho007@Rohans-MacBook-Pro-7 ~ % docker images
REPOSITORY                TAG       IMAGE ID       CREATED          SIZE
docker-ubuntu-sysbench    latest    becff0ddbf22   12 minutes ago   142MB
ubuntu                    latest    e34e831650c1   9 days ago       77.9MB
ubuntu                    20.04     f78909c2b360   5 weeks ago      72.8MB
epoupon/lms               latest    4bc301fb85bd   3 months ago     39.1MB
newsnotification-image    v1        7e81b09000af   8 months ago     46.2MB
<none>                    <none>    111021e5e341   8 months ago     46.2MB
redis                     latest    33e3db53b328   9 months ago     117MB
mongo                     latest    9a5e0d0cf6de   10 months ago    646MB
postgres                  9.6       027ccf656dc1   23 months ago    200MB
mongo-express             latest    2d2fb2cabc8f   2 years ago      136MB
redis                     4.0       191c4017dcdd   3 years ago      89.3MB
yancho007@Rohans-MacBook-Pro-7 ~ % docker container run -d --name=my-docker-ubuntu-sysbench docker-ubuntu-sysbench
f57c1ad885bce30763bdab5b8aed77b91b6e34aedc5ba10997b44c8f546ba8d1
yancho007@Rohans-MacBook-Pro-7 ~ %
```

Step 8- Docker Base image running (to be used to launch all configurations)

```
yancho007@Rohans-MacBook-Pro-7 ~ % docker run -it --name=my-docker-sysbench docker-ubuntu-sysbench /bin/bash
root@c5efbf58efbf:/#
```

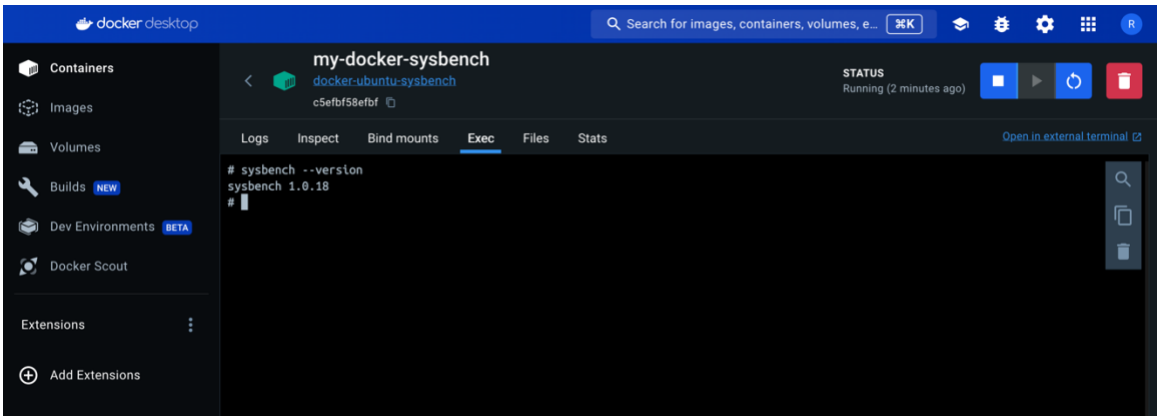


Image ID : c5efbf58efbf2218d8d7e7f0c63498768c0239b3a036251d80b039e901e76c4a

Starting docker in docker Desktop/CLI

We will run below test cases (Same as previous on 4 Docker configuration) – (6*4) ->24 Tests

2 CPU cases-

- Test Case 0: Limiting the total execution time to 30 seconds: sysbench --test=cpu --time=45 run
- Test Case 1: sysbench --test=cpu --cpu-max-prime=100 --time=45 run
- Test Case 2: sysbench --test=cpu --cpu-max-prime=1000 --time=45 run

2 memory cases-

The --memory-block-size option in Sysbench is used to define the block size for memory allocation and deallocation operations. This option is relevant when running memory-related tests with Sysbench. Below are a few test cases for --memory-block-size:-

Test Case:

- sysbench memory --memory-block-size=16M run
- sysbench memory --memory-block-size=8k run

2 fileio cases-

Test Case 1 (Sequential R/W) : Creating test files of total size 1GB.

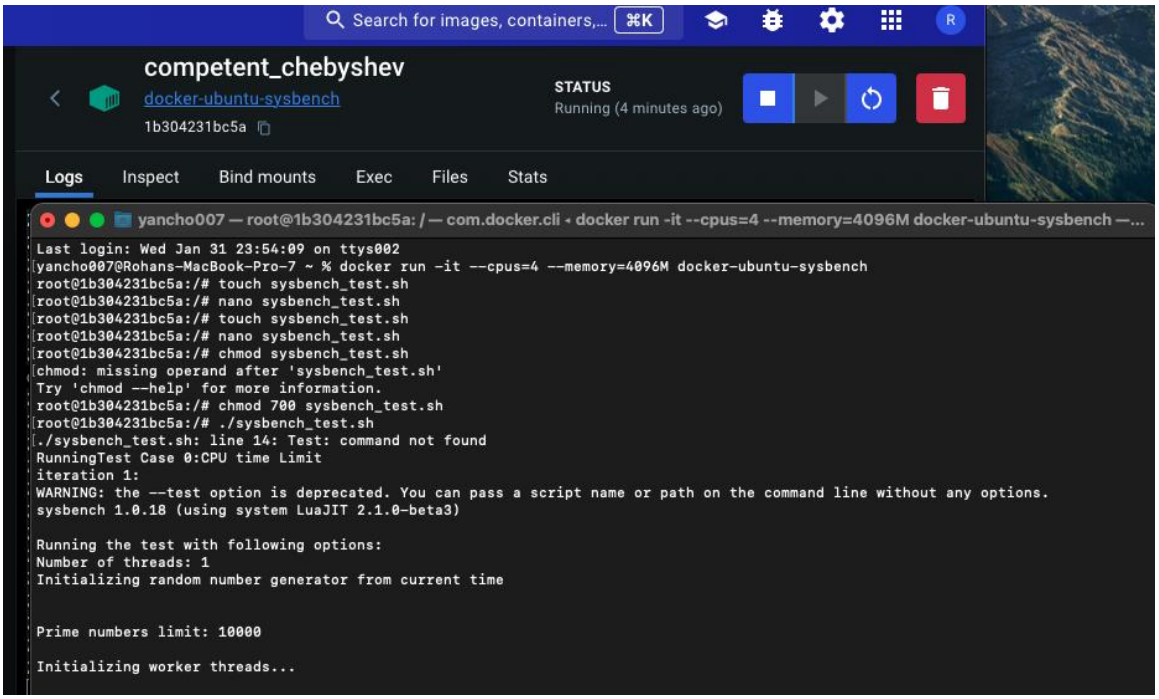
- sysbench --num-threads=14 --test=fileio --file-total-size=1G --max-time=100 --file-test-mode=seqwr prepare
- Running sequential read/write workload for 300 s
- sysbench --num-threads=14 --test=fileio --file-total-size=1G --file-test-mode=seqwr --max-time=100 --max-requests=0 run

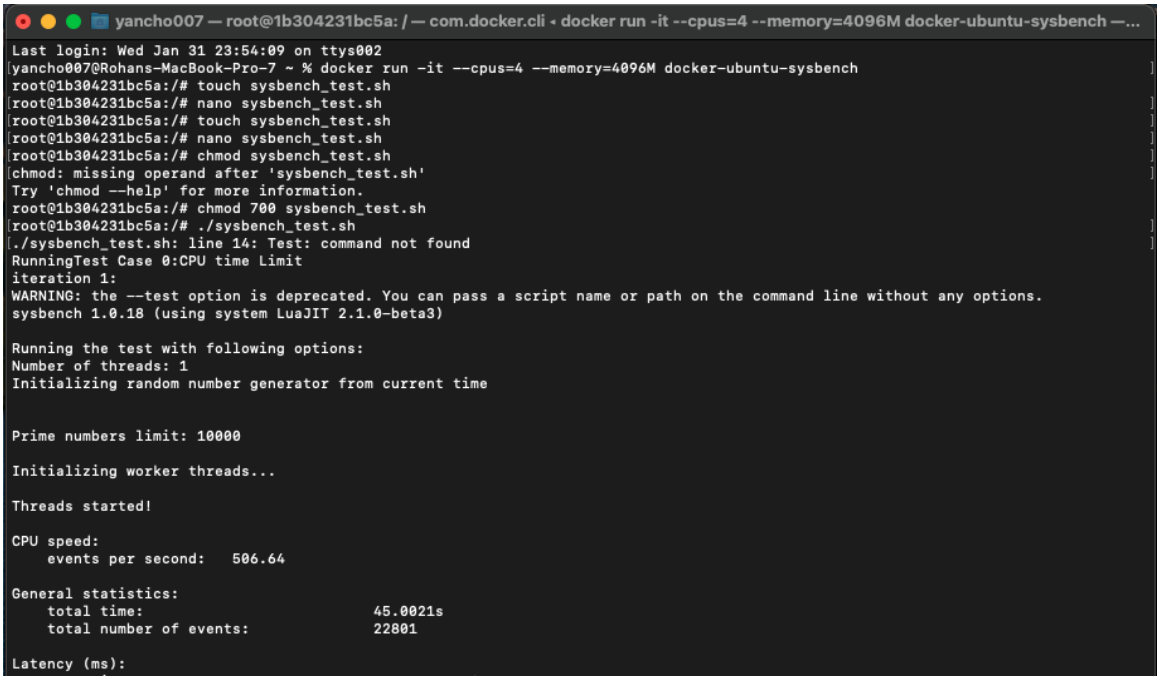
Test Case 2 (Random R/W): Creating test files of total size 2GB.

- sysbench --num-threads=14 --test=fileio --file-total-size=2G --max-time=100 --file-test-mode= rndrw prepare
- sysbench --num-threads=14 --test=fileio --file-total-size=2G --file-test-mode= rndrw --max-time=100 --max-requests=0 run
- sysbench --test=fileio --file-total-size=2G cleanup

Docker Configuration 1->(1b304231bc5a)

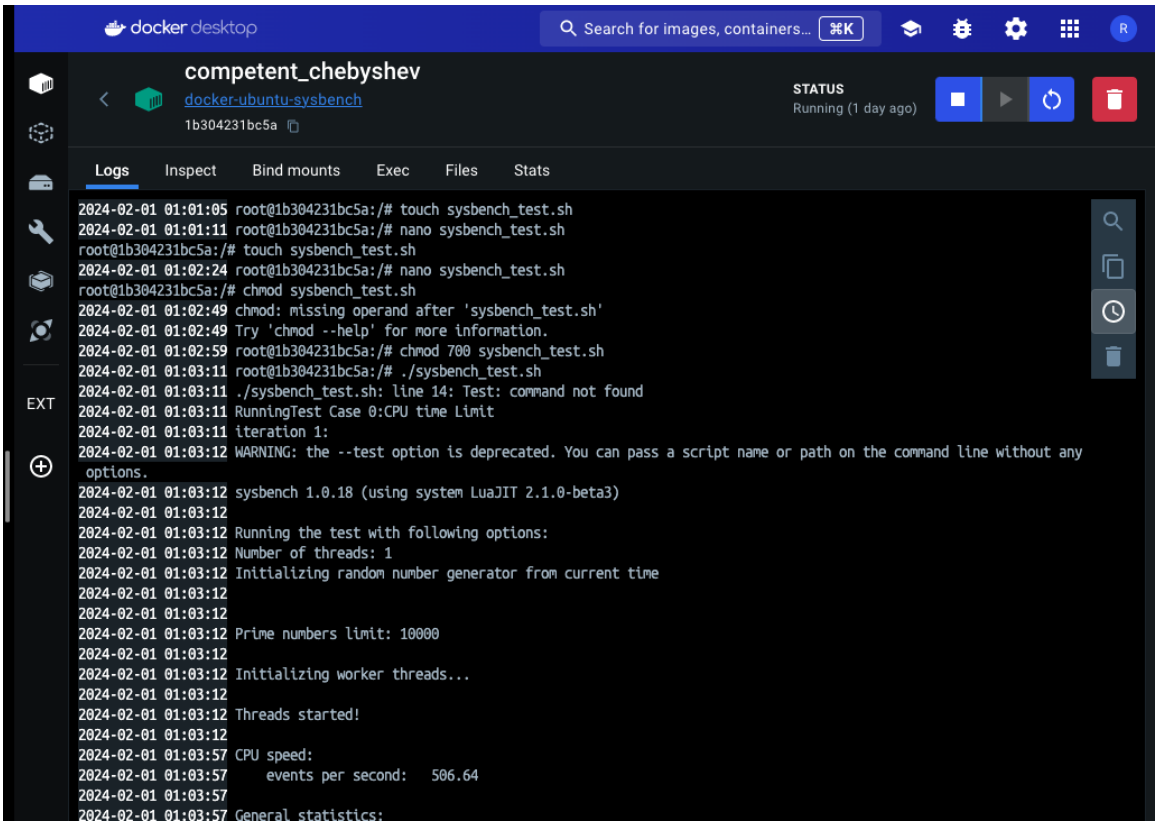
docker run -it --cpus=4 --memory=4096M docker-ubuntu-sysbench





[Test results and config bash scripts]

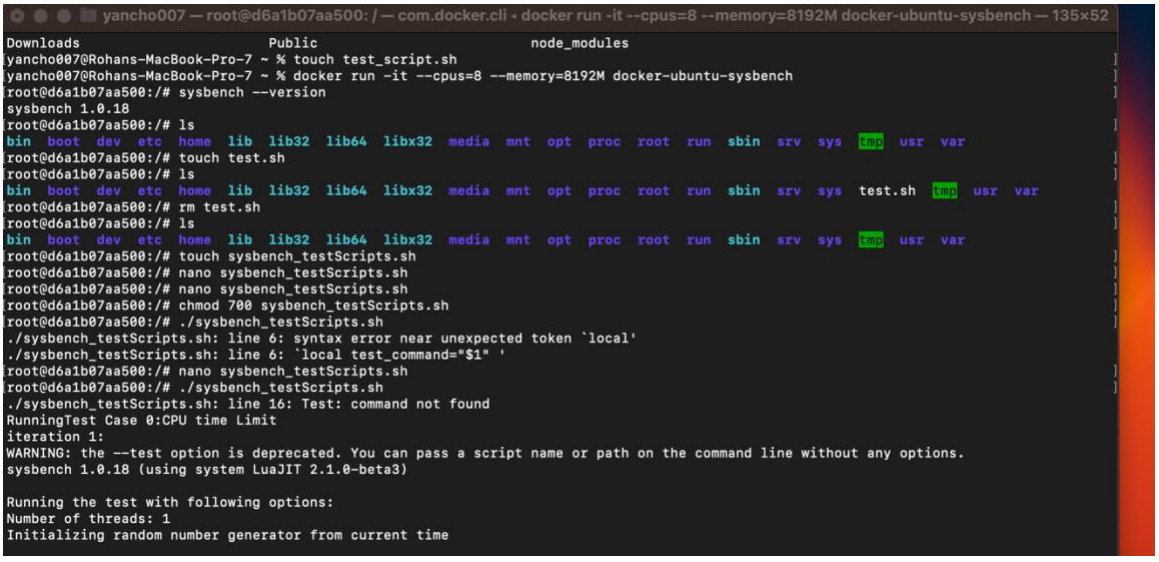
DOCKER HISTORY-

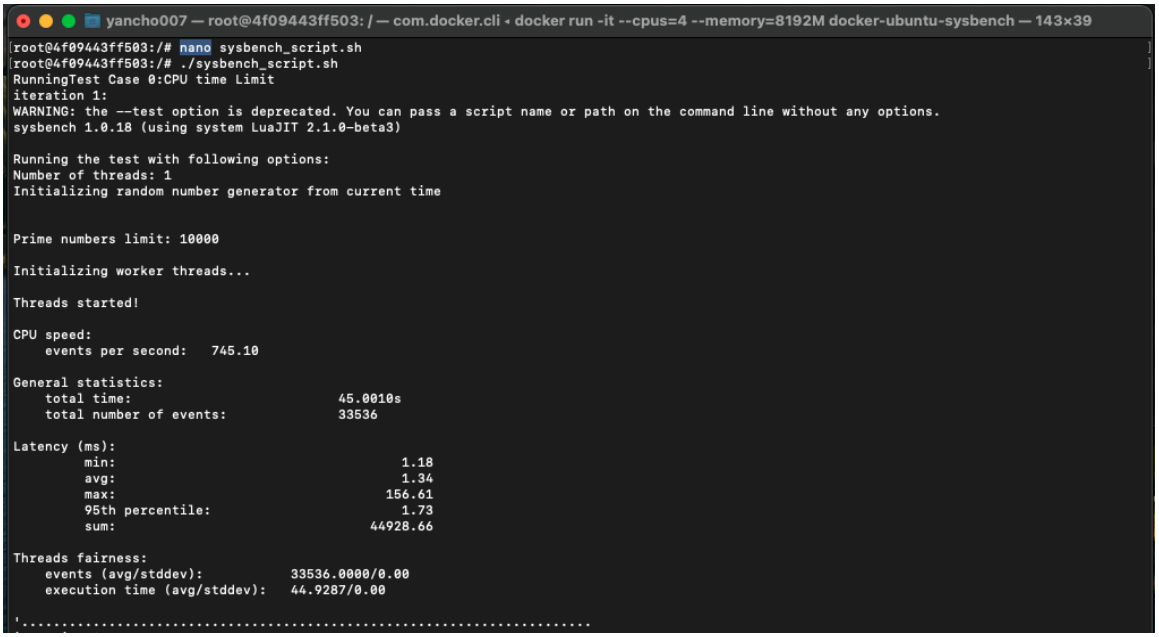


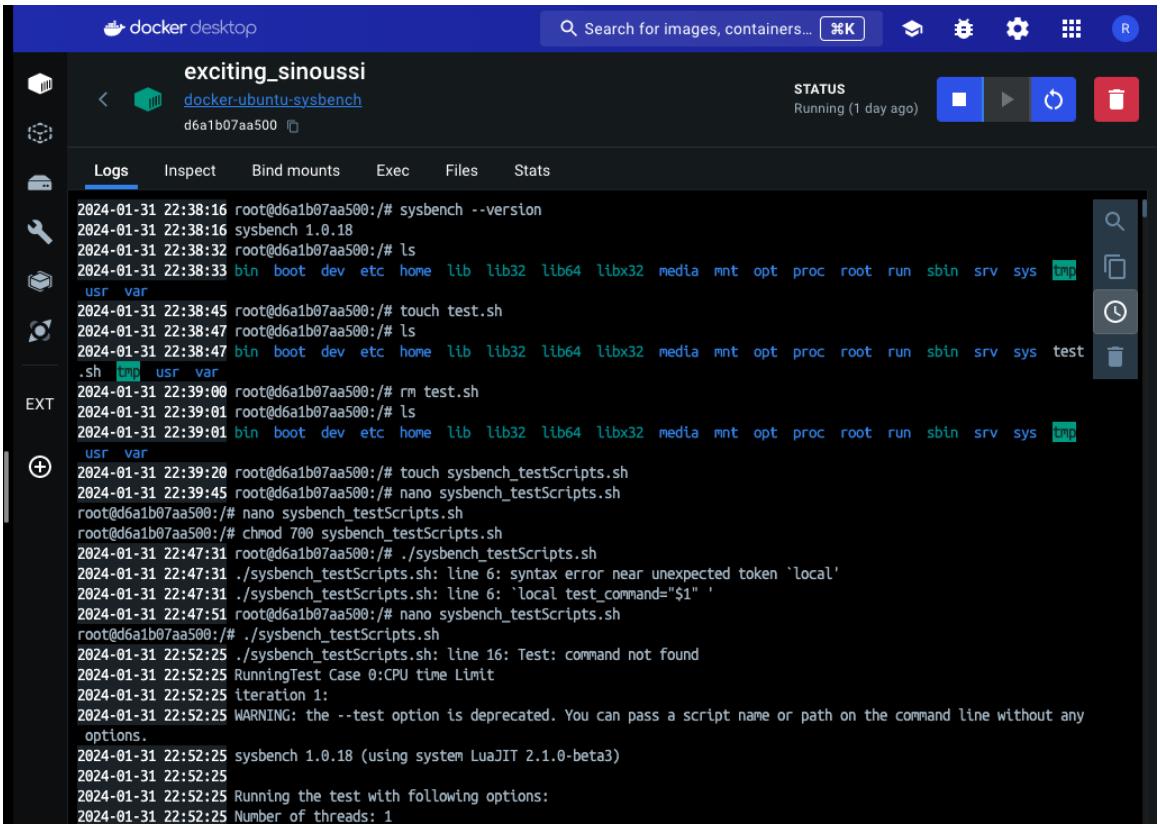
[Test results and config bash scripts]

Docker Configuration 2-> (d6a1b07aa500)

docker run -it --cpus=8 --memory=8192M docker-ubuntu-sysbench



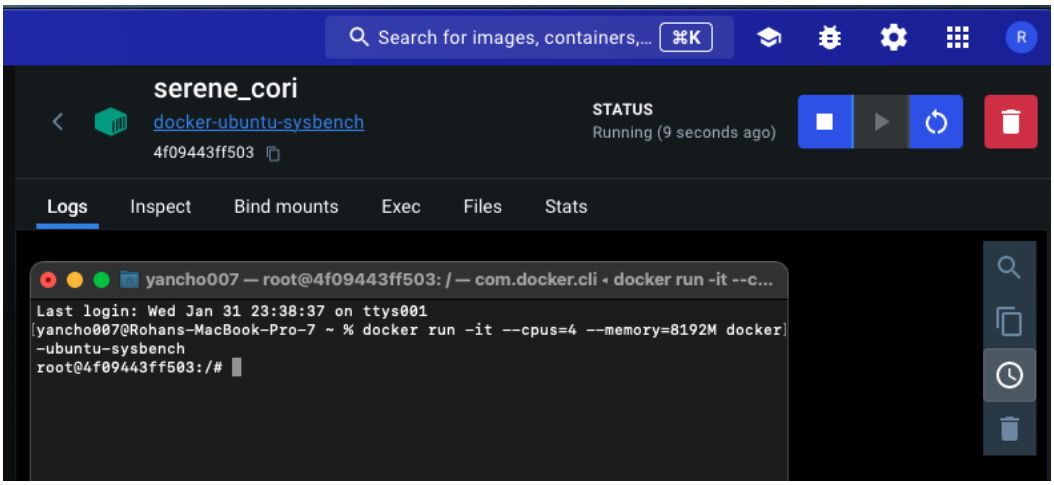


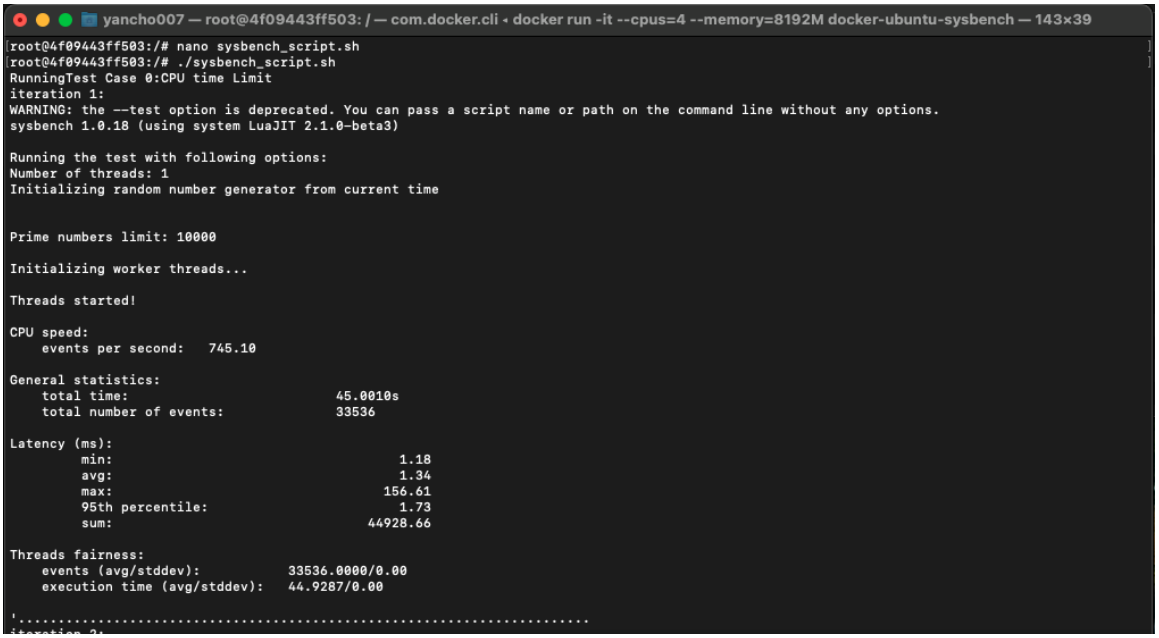


[Test results and config bash scripts]

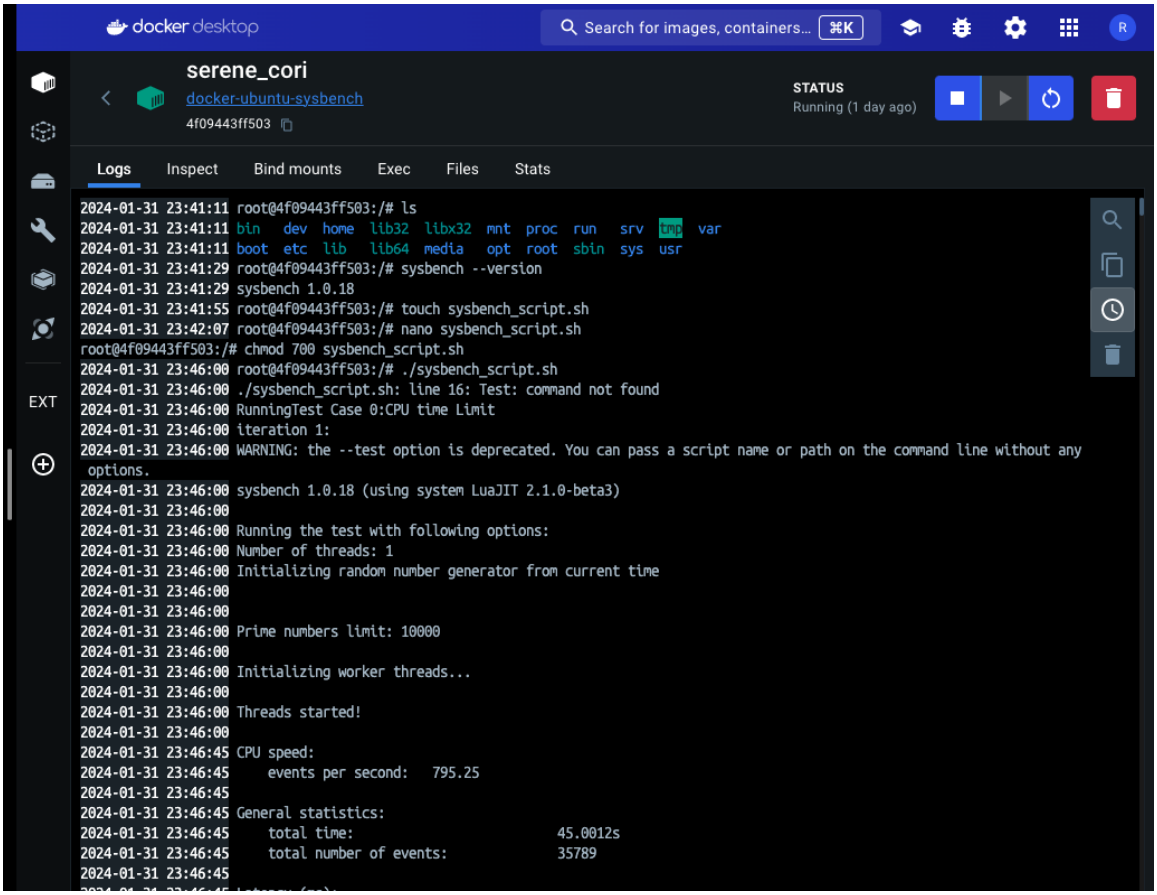
Docker Configuration 3->(4f09443ff503)

docker run -it --cpus=4 --memory=8192M docker-ubuntu-sysbench





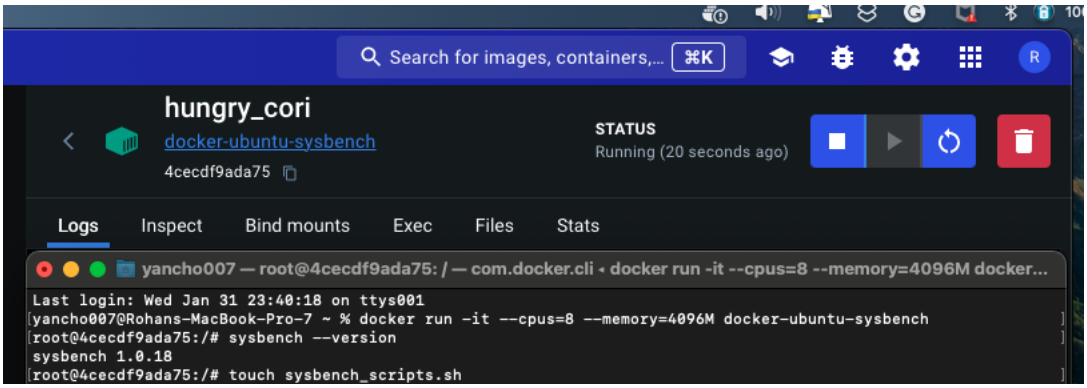
DOCKER HISTORY

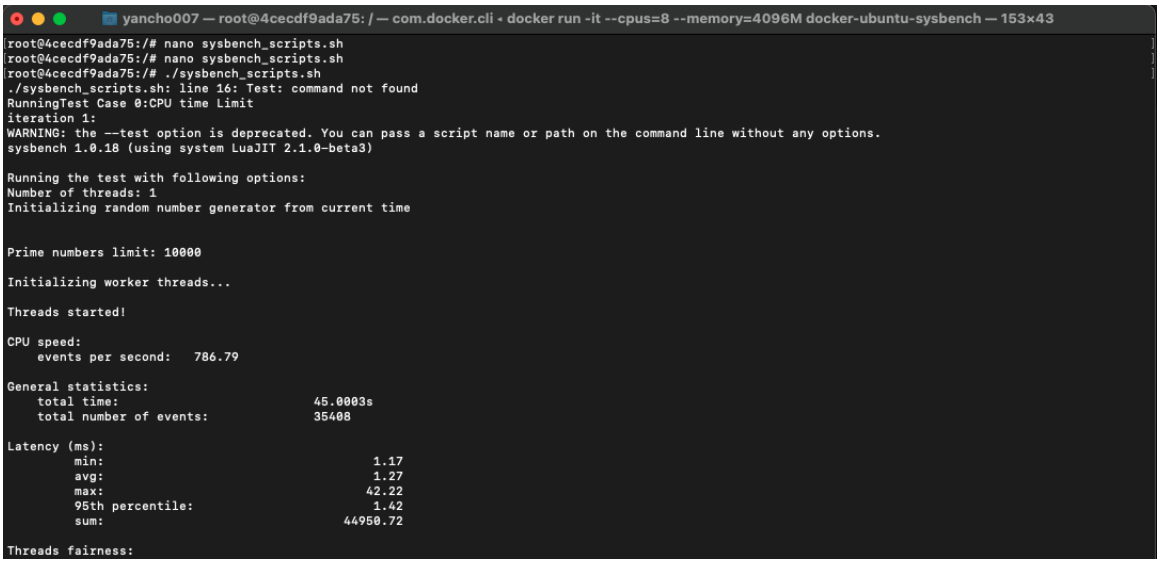


[Test results and config bash scripts]

Docker Configuration 4->(4cecdf9ada7599a60b9c74f3459e7cebf43fcceaeabc1bbd6581c253d0f40bb5)

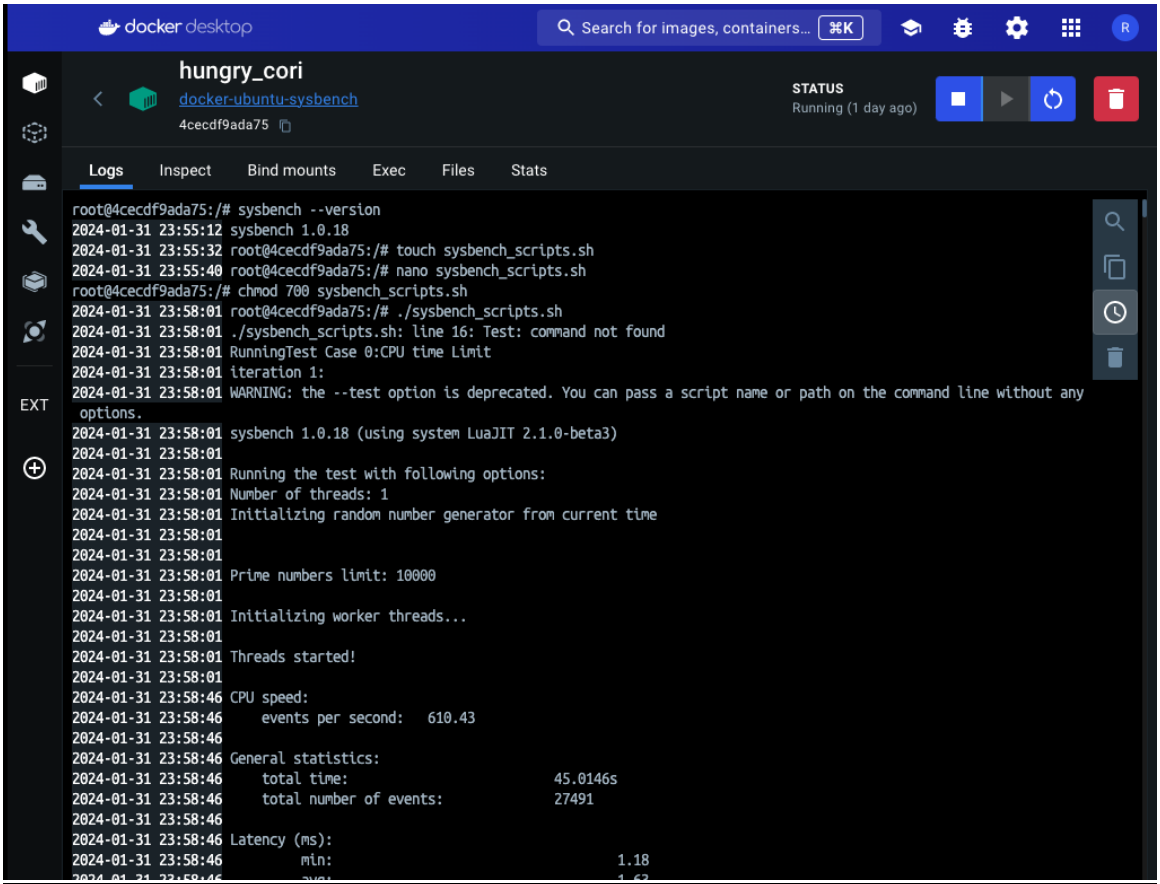
docker run -it --cpus=8 --memory=4096M docker-ubuntu-sysbench





Test results and config bash scripts

Docker history



Tests Comparison and summary

Test Case 0- CPU

Limiting the total execution time to 30 seconds: sysbench --test=cpu --time=45 run

	Prime No limit-10000 & Events per second-							
	Qcow2				raw			
Iteration No	Config 1	Config2	Config 3	Config4	Config 1	Config2	Config 3	Config4
1	236.24	238.12	247.83	247.17	247.49	251.54	251.23	239.93
2	241.76	245.24	228.48	231.98	240.26	240.25	210.32	250.32
3	253.96	240.95	220.61	243.82	253.46	239.42	208.32	251.68
4	254.82	248.75	256.31	256.84	254.58	254.54	225.65	240.65
5	252.45	250.30	235.32	238.45	238.15	248.25	234.80	244.15
Average	247.846	244.672	237.71	243.652	246.788	246.8	226.064	245.346

	Events per second			
	Docker			
Iteration No	Config 1	Config2	Config 3	Config4
1	377.79	736.12	745.56	610.43
2	506.64	768.19	770.65	738.86
3	504.51	727.76	780.54	671.10
4	474.33	643.97	754.80	666.18
5	493.73	562.59	761.03	606.26
Average	471.4	687.726	762.516	658.566

Conclusion: According to our experiment, we observe that QEMU VM is faster than docker probable reasons-

1. Docker introduces some overhead due to containerization. The additional layer of abstraction and isolation might lead to slightly higher execution times compared to running directly on QEMU without containerization.
2. Docker allocates resources to containers, and sometimes these allocations might not perfectly match the requirements of the sysbench CPU test. QEMU might have more direct control over resources in this specific scenario.
3. Docker provides process-level isolation, while QEMU provides virtual machine-level isolation. Depending on the workload, VM-level isolation might have performance advantages over process-level isolation.

Test Case 1- CPU

```
sysbench --test=cpu --cpu-max-prime=100 --time=45 run
```

	CPU Max prime = 100- Events per second							
	Qcow2				raw			
Iteration No	Config 1	Config2	Config 3	Config4	Config 1	Config2	Config 3	Config4
1	36451.01	46246.64	43302.12	49312.21	57879.95	52095.85	44575.32	53320.75
2	47052.31	51119.63	37204.32	37204.19	54439.45	44122.45	39461.45	44122.61
3	52111.31	48093.61	40317.65	46246.64	58163.85	54439.25	52095.28	49880.81
4	49312.21	51270.25	46746.45	48093.61	54122.35	46241.15	52870.91	46241.37
5	45312.03	50409.32	34112.85	51270.25	44218.75	39461.75	47716.54	46682.19
Average	46047.77	49427.89	40336.67	46425.38	53764.87	47272.09	47343.95	48049.54

	Events per second			
	Docker			
Iteration No	Config 1	Config2	Config 3	Config4
1	500735.87	359566.52	484078.55	414841.26
2	476663.42	429460.40	402740.40	409737.01
3	379819.76	371058.35	416013.09	397638.15
4	423306.76	434396.59	384661.09	396469.40
5	370637.05	455412.29	413151.87	446677.05
Average	430232.57	409978.83	420129.85	413072.57

Conclusion: According to our experiment, we observe that QEMU qcow2 VM is faster than docker & docker is faster then QEMU raw image probable reasons-

1. QEMU with qcow2 might benefit from caching mechanisms specific to the qcow2 format, providing advantages in scenarios where caching plays a significant role. Docker, depending on its storage driver and caching strategy, might introduce different caching behavior.
2. The qcow2 format typically introduces some overhead due to compression and snapshot features. If the workload involves significant storage operations, the qcow2 format might perform better than raw, which is a straightforward format
3. As Docker relies on the host OS's kernel, interactions between the workload, Docker, and the kernel could be influencing the results. QEMU with a raw image operates at a more isolated level.

Test Case 2- CPU

```
sysbench --test=cpu --cpu-max-prime=1000 --time=45 run
```

	CPU Max prime = 1000 -Events per second							
	Qcow2				raw			
Iteration No	Config 1	Config2	Config 3	Config4	Config 1	Config2	Config 3	Config4
1	4512.75	4702.98	4525.15	4704.53	4960.84	4106.75	4241.05	4402.15
2	4721.78	4745.36	4646.23	4412.31	5008.15	5036.75	4106.75	4553.73
3	4572.24	4704.53	4403.52	4677.80	5025.25	4505.67	3677.25	4547.27
4	4704.53	4572.24	4721.78	4821.31	4960.84	3677.25	4335.73	4505.67
5	4646.23	4685.35	4590.95	4671.95	5036.75	4568.67	4568.67	4213.29
Average	4631.506	4682.092	4577.526	4657.58	4998.366	4379.018	4185.89	4444.422

	Events per second			
	Docker			
Iteration No	Config 1	Config2	Config 3	Config4
1	16945.80	17689.12	17505.76	17451.94
2	17817.71	18133.35	17473.96	18175.85
3	17338.43	18086.14	18154.96	17875.99
4	18153.66	19359.64	17924.09	18605.95
5	17994.89	18482.04	18656.93	19367.92
Average	17650.098	18350.058	17943.14	18295.53

Conclusion 1: As we see, when we increase the cpu-max-prime argument value the number of events per second decreases for our current experimental configuration. **&** we observe that QEMU VM is faster than docker for number of events probable reasons-

1. A higher value for cpu-max-prime indicates a more intense CPU workload. In such scenarios, the virtualization overhead introduced by Docker may become more noticeable compared to QEMU. QEMU might have optimizations or configurations that handle highly CPU-intensive workloads more efficiently.
2. he increased CPU intensity might put more strain on the resource allocation and isolation mechanisms of Docker. QEMU, operating at the virtual machine level, may handle such workloads with less overhead compared to Docker's process-level isolation.
3. As the CPU workload intensifies, the impact of storage-related operations, including caching, might become more pronounced. QEMU with its qcow2 format or raw image might exhibit different caching behaviors compared to Docker with its storage drivers.

CPU Test Case 3- Memory

sysbench memory --memory-block-size=16M run

	CPU memory- 16 M block-Operations per second							
	Qcow2				raw			
Iteration No	Config 1	Config2	Config 3	Config4	Config 1	Config2	Config 3	Config4
1	105.12	98.79	131.41	110.68	106.20	122.11	83.46	99.41
2	118.65	128.62	120.45	108.32	107.41	113.25	88.61	102.34
3	108.5	108.34	131.30	109.39	108.45	107.87	87.54	108.32
4	125.36	106.85	135.12	111.45	108.15	120.02	90.25	104.78
5	121.52	120.08	111.62	109.63	106.20	109.57	94.31	101.65
Average	115.83	112.536	125.98	109.894	107.282	114.564	88.834	103.3

	Events per second			
	Docker			
Iteration No	Config 1	Config2	Config 3	Config4
1	404.14	414.54	692.12	488.25
2	474.04	400.20	647.32	602.73
3	518.63	428.86	708.62	732.28
4	520.08	535.69	625.23	719.74
5	495.86	536.35	681.60	728.44
Average	482.55	463.12	670.97	654.28

Conclusion 1: As we see, comparing the performance of QEMU and Docker in a sysbench memory test with a memory block size of 16M, and finding that QEMU is faster than Docker, probable reasons-

1. Docker relies on the host OS's memory management for containerized applications, while QEMU manages memory for virtual machines. The way each system allocates and manages memory for the specified block size can impact performance
2. Docker introduces some level of overhead due to containerization. The additional layer of abstraction and isolation might have a more noticeable impact on memory-related tests, especially when compared to the more direct memory access available in QEMU's virtual machine environment.
3. As the CPU workload intensifies, the impact of storage-related operations, including caching, might become more pronounced. QEMU with its qcow2 format or raw image might exhibit different caching behaviors compared to Docker with its storage drivers.

Test Case 4- Memory

```
sysbench memory --memory-block-size=8k run
```

	CPU memory- 8k block-Events per second							
	Qcow2				raw			
Iteration No	Config 1	Config2	Config 3	Config4	Config 1	Config2	Config 3	Config4
1	84696.53	80792.26	86452.62	78558.83	93301.81	69579.47	859935.18	72180.54
2	80986.86	82300.12	84696.53	80455.46	93002.95	85361.76	8231547.2	79213.65
3	80455.46	80986.86	86514.16	88501.52	93540.10	85361.76	842484.57	74764.65
4	90662.61	81300.12	84512.52	90662.61	94523.45	67544.96	813625.45	81245.95
5	86514.16	86601.12	87533.20	83195.63	92854.56	75213.45	863256.85	80861.72
	84663.124	82396.096	85941.806	84274.81	93444.574	76612.28	2322169.85	77653.302

	Total operations per second			
	Docker			
it No	Config 1	Config2	Config 3	Config4
1	695550.76	808425.76	1019472.74	1188424.24
2	571048.28	900319.49	1013454.47	1183309.75
3	529328.04	844760.79	1005276.71	1164973.45
4	184493.17	772844.52	1049588.58	1152173.37
5	377682.02	796758.88	10012354.65	1171149.74
	471620.454	824621.888	2820029.43	1172006.11

Conclusion: According to our experiment, we observe that with decrease in memory block size the number of events per second increased in all 3 cases. Mainly related to the impact of memory block size on the efficiency of memory operations and caching. When dealing with smaller memory block sizes. however config 3 & 4 of docker performs better then QEMU, however at lower config QEMU performs well. Probable reasons-

- Smaller memory block sizes are more likely to fit within CPU caches, such as L1 and L2 caches. This allows for faster access to memory, as data can be retrieved from these caches more quickly compared to fetching from the main memory. As a result, the overall performance of memory operations improves.
- Smaller memory block sizes can lead to lower memory access latencies. Accessing smaller chunks of memory requires less time to fetch data, reducing the overall time it takes to perform memory operations.

File Io Test Case 5-

(Sequential R/W) : Creating test files of total size 1GB.

- sysbench --num-threads=14 --test=fileio --file-total-size=1G --max-time=100 --file-test-mode=seqwr prepare
- Running sequential read/write workload for 300 s sysbench --num-threads=14 --test=fileio --file-total-size=1G --file-test-mode=seqwr --max-time=100 --max-requests=0 run
- sysbench --test=fileio --file-total-size=1G cleanup

	File io 1GB Sequential W/R							
	Qcow2				raw			
It. No	Config 1	Config2	Config 3	Config4	Config 1	Config2	Config 3	Config4
1	reads/s: 0.00 writes/s:1589.43 fsyncs/s:2051.38	reads/s: 0.00 writes/s:1218.43 fsyncs/s:1576.38	reads/s: 0.00 writes/s:1515.94 fsyncs/s:1957.38	reads/s: 0.00 writes/s:1378.94 fsyncs/s:1781.38	reads/s: 0.00 writes/s:1426.20 fsyncs/s:1825.76	reads/s: 0.00 writes/s:912.23 fsyncs/s:1168.66	reads/s: 0.00 writes/s:1142.40 fsyncs/s:1462.77	reads/s: 0.00 writes/s:1153.87 fsyncs/s:1477.93
2	reads/s: 0.00 writes/s:1027.73 fsyncs/s:1332.21	reads/s: 0.00 writes/s:1154.98 fsyncs/s:1495.60	reads/s: 0.00 writes/s:1589.43 fsyncs/s:2051.38	reads/s: 0.00 writes/s:1422.94 fsyncs/s:1837.38	reads/s: 0.00 writes/s:1413.29 fsyncs/s:1810.19	reads/s: 0.00 writes/s:995.25 fsyncs/s:1275.44	reads/s: 0.00 writes/s:1076.82.3 fsyncs/s:1378.66	reads/s: 0.00 writes/s:976.40 fsyncs/s:1250.77
3	reads/s: 0.00 writes/s:1515.94 fsyncs/s:1957.38	reads/s: 0.00 writes/s:1156.50 fsyncs/s:1497.98	reads/s: 0.00 writes/s:1576.43 fsyncs/s:2035.38	reads/s: 0.00 writes/s:1420.01 fsyncs/s:1834.23	reads/s: 0.00 writes/s:1473.33 fsyncs/s:1886.80	reads/s: 0.00 writes/s:1017.60 fsyncs/s:1303.81	reads/s: 0.00 writes/s:1009.23 fsyncs/s:1292.25	reads/s: 0.00 writes/s:1219.40 fsyncs/s:1561.77
4	reads/s: 0.00 writes/s:1378.94 fsyncs/s:1781.38	reads/s: 0.00 writes/s:1330.74 fsyncs/s:1720.00	reads/s: 0.00 writes/s:1378.94 fsyncs/s:1781.38	reads/s: 0.00 writes/s:1301.47 fsyncs/s:1683.46	reads/s: 0.00 writes/s:828.05 fsyncs/s:1061.06	reads/s: 0.00 writes/s:1205.40 fsyncs/s:1544.00	reads/s: 0.00 writes/s:1352.23 fsyncs/s:1731.66	reads/s: 0.00 writes/s:1264.40 fsyncs/s:1618.77
5	reads/s: 0.00 writes/s:1154.98 fsyncs/s:1495.60	reads/s: 0.00 writes/s:1027.73 fsyncs/s:1332.21	reads/s: 0.00 writes/s:1154.98 fsyncs/s:1495.60	reads/s: 0.00 writes/s:1218.43 fsyncs/s:1576.38	reads/s: 0.00 writes/s:995.25 fsyncs/s:1275.44	reads/s: 0.00 writes/s:1076.82.3 fsyncs/s:1378.66	reads/s: 0.00 writes/s:1153.87 fsyncs/s:1477.93	reads/s: 0.00 writes/s:912.23 fsyncs/s:1168.66

	File io 1GB Sequential W/R			
	Docker			
Iteration No	Config 1	Config2	Config 3	Config4
1	reads/s: 0.00 writes/s:2557.43 fsyncs/s:3290.38	reads/s: 0.00 writes/s:6834.43 fsyncs/s:8765.38	reads/s: 0.00 writes/s:9020.43 fsyncs/s:111563.38	reads/s: 0.00 writes/s:11797.43 fsyncs/s:15118.38
2	reads/s: 0.00 writes/s:2827.98 fsyncs/s:3637.60	reads/s: 0.00 writes/s:5460.43 fsyncs/s:7006.38	reads/s: 0.00 writes/s:8532.43 fsyncs/s:10939.38	reads/s: 0.00 writes/s:11357.43 fsyncs/s:14555.38
3	reads/s: 0.00 writes/s:3108.50 fsyncs/s:3997.98	reads/s: 0.00 writes/s:6059.43 fsyncs/s:7774.38	reads/s: 0.00 writes/s:6124.43 fsyncs/s:7857.38	reads/s: 0.00 writes/s:10029.43 fsyncs/s:12854.38
4	reads/s: 0.00 writes/s:3220.74 fsyncs/s:4140.00	reads/s: 0.00 writes/s:4680.43 fsyncs/s:6008.38	reads/s: 0.00 writes/s:5595.43 fsyncs/s:7179.38	reads/s: 0.00 writes/s:115..43 fsyncs/s:150.38
5	reads/s: 0.00 writes/s:4093.73 fsyncs/s:5256.21	reads/s: 0.00 writes/s:2548.43 fsyncs/s:3279.38	reads/s: 0.00 writes/s:6949.43 fsyncs/s:8912.38	reads/s: 0.00 writes/s:10285.43 fsyncs/s:13183.38

Conclusion 1: As we see, comparing the performance of QEMU and Docker in a sysbench fileio test docker performs much better then QEMU. At higher configurations the write speed of docker increases in multifold. Few reasons for this behaviour-

1. QEMU with qcow2 and raw images operates in a virtualized environment, while Docker operates at the container level. Depending on the underlying storage and access mechanisms, Docker may have more direct access to the host's storage, leading to faster sequential read/write operations.
2. Docker's resource allocation and isolation mechanisms, specifically designed for containerized applications, might be more efficient in handling sequential read/write workloads. QEMU's virtualization layer may introduce additional overhead.
3. he choice of filesystem on the host system where Docker is running can impact file I/O performance. Docker containers typically share the host's filesystem, and if it uses a filesystem optimized for sequential operations, it could outperform QEMU with qcow2 and raw images.

File Io Test Case 6-

Test Case 2 (Random R/W): Creating test files of total size 2GB.

- sysbench --num-threads=14 --test=fileio --file-total-size=2G --max-time=100 --file-test-mode= rndrw prepare
- sysbench --num-threads=14 --test=fileio --file-total-size=2G --file-test-mode= rndrw --max-time=100 --max-requests=0 run
- sysbench --test=fileio --file-total-size=2G cleanup

	File io 2GB Random W/R							
	Qcow2				raw			
It. No	Config 1	Config2	Config 3	Config4	Config 1	Config2	Config 3	Config4
1	reads/s: 650.53 writes/s:433.693 fsyncs/s:1403.29	reads/s: 714.53 writes/s:476.693 fsyncs/s:1541.22	reads/s: 708.53 writes/s:472.93 fsyncs/s:1529.29	reads/s: 670.53 writes/s:409.693 fsyncs/s:1477.29	reads/s: 614.53 writes/s:446.693 fsyncs/s:1311.29	reads/s: 510.86 writes/s:340.57 fsyncs/s:1090.69	reads/s: 659.66 writes/s:439.23 fsyncs/s:1408.66	reads/s: 661.66 writes/s:441.23 fsyncs/s:1412.66
2	reads/s: 735.53 writes/s:490.693 fsyncs/s:1587.29	reads/s: 721.53 writes/s:480.693 fsyncs/s:1556.29	reads/s: 738.53 writes/s:492.63 fsyncs/s:1591.29	reads/s: 689.53 writes/s:459.69 fsyncs/s:1487.29	reads/s: 631.96 writes/s:421.31 fsyncs/s:1348.72	reads/s: 567.09 writes/s:378.06 fsyncs/s:1209.87	reads/s: 678.70 writes/s:452.47 fsyncs/s:1448.14	reads/s: 595.66 writes/s:396.23 fsyncs/s:1270.66
3	reads/s: 714.53 writes/s:476.693 fsyncs/s:1542.29	reads/s: 705.53 writes/s:470.693 fsyncs/s:1522.29	reads/s: 756.53 writes/s:504.693 fsyncs/s:1630.29	reads/s: 673.53 writes/s:449.13 fsyncs/s:1454.29	reads/s: 592.45 writes/s:394.97 fsyncs/s:1265.18	reads/s: 309.15 writes/s:206.10 fsyncs/s:660.10	reads/s: 568.22 writes/s:378.23 fsyncs/s:1212.66	reads/s: 615.66 writes/s:410.23 fsyncs/s:1312.66
4	reads/s: 664.53 writes/s:442.63 fsyncs/s:1433.29	reads/s: 571.53 writes/s:380.693 fsyncs/s:1234.29	reads/s: 674.53 writes/s:449.63 fsyncs/s:1457.29	reads/s: 665.53 writes/s:443.49 fsyncs/s:1436.29	reads/s: 649.25 writes/s:432.83 fsyncs/s:1385.33	reads/s: 402.29 writes/s:268.19 fsyncs/s:858.35	reads/s: 590.67 writes/s:393.78 fsyncs/s:1261.66	reads/s: 616.66 writes/s:410.23 fsyncs/s:1315.66
5	reads/s: 658.53 writes/s:439.693 fsyncs/s:1423.29	reads/s: 647.53 writes/s:431.693 fsyncs/s:1399.29	reads/s: 684.53 writes/s:457.693 fsyncs/s:1481.29	reads/s: 667.53 writes/s:445.693 fsyncs/s:1440.29	reads/s: 0.00 writes/s:1027.73 fsyncs/s:1332.21	reads/s: 430.15 writes/s:286.77 fsyncs/s:918.56	reads/s: 491.66 writes/s:327.23 fsyncs/s:1049.66	reads/s: 491.66 writes/s:327.23 fsyncs/s:1049.66

	File io 2GB Random W/R			
	Docker			
Iteration No	**Config 1**	**Config2**	**Config 3**	**Config4**
1	reads/s: 1749.6 writes/s:1166.43 fsyncs/s:3746.38	reads/s: 1667 writes/s:1111.43 fsyncs/s:3575.38	reads/s: 1621 writes/s:1080.43 fsyncs/s:3476.38	reads/s: 3713 writes/s:2475.43 fsyncs/s:7938.38
2	reads/s1.3500 writes/s:0.900 fsyncs/s:3.030	reads/s: 1775.6 writes/s:1183.43 fsyncs/s:3805.38	reads/s: 1707.6 writes/s:1137.43 fsyncs/s:3658.38	reads/s: 4344.6 writes/s:2896.43 fsyncs/s:9285.38
3	reads/s: 2774.86 writes/s:1849.50 fsyncs/s:5937.98	reads/s: 1848.38 writes/s:1232.43 fsyncs/s:3960.38	reads/s: 1689.38 writes/s:1126.43 fsyncs/s:3621.38	reads/s: 4855.6 writes/s:3237.43 fsyncs/s:10376.38
4	reads/s: 2793 writes/s:1862.74 fsyncs/s:5977.00	reads/s: 2295.5 writes/s:1530.43 fsyncs/s:4915.38	reads/s: 1830.5 writes/s:1220.43 fsyncs/s:3922.38	reads/s: 527.6 writes/s:351.43 fsyncs/s:1129.38
5	reads/s: 3149.96 writes/s:2099.73 fsyncs/s:6736.21	reads/s: 2225.49 writes/s:1483.43 fsyncs/s:4764.79	reads/s: 1748.49 writes/s:1165.43 fsyncs/s:3747.79	reads/s: 4168.6 writes/s:2778.43 fsyncs/s:8910.38

Conclusion 1: As we see with 2gb random w/r performance is QEMU & docker is more &, comparing the performance of QEMU and Docker in a sysbench fileio test docker performs much better then QEMU. At higher configurations the write speed of docker increases in multifold. Few reasons for this behaviour-

1. QEMU with qcow2 and raw images operates in a virtualized environment, while Docker operates at the container level. Depending on the underlying storage and access mechanisms, Docker may have more direct access to the host's storage, leading to faster sequential read/write operations.
2. QEMU with qcow2 and raw images has different image formats with distinct characteristics. Qcow2 introduces additional features like compression and snapshots, but it may have overhead. Raw images are more straightforward. Depending on the workload and the nature of data, these formats can impact read and write performance differently.
3. We observe if our application primarily performs large, sequential data transfers, then optimizing for sequential I/O may be beneficial. On the other hand, if your application involves frequent, non-linear access to data, optimizing for random I/O is more appropriate.

Git Repository Information:

Account name	rohanpandeymech
Repository name	COEN-241-Cloud-Computing
Folder which contains HW1	HW1
Link to repository	https://github.com/rohanpandeymech/CloudComputing/tree/main/HW1
Commit ID	8a145d814fccca352317d047ffa65c1016ff55e8