

COEN 241: Cloud Computing

Homework 1(System vs OS Virtualization)

System Configurations:

Host System:

MacBook Pro 13-inch, 2019
CPU: 2.4 GHz Quad-Core Intel Core i5
Memory: 8 GB 2133 MHz LPDDR3
OS: MacOS Ventura 13.1

Configurations for experiments:

1. 2GB of memory, 1 CPU cores, 1 thread per core, no hardware accel
2. 2GB of memory, 1 CPU cores, 1 thread per core, with hardware accel
3. 4GB of memory, 2 CPU cores, 1 thread per core, with hardware accel

Config 1 and 2 will help us determine the performance gain that is guaranteed by hardware acceleration and config 3 with double the resources can make things clear.

QEMU Setup:

1. Install homebrew using Terminal with below command

```
$ /bin/bash -c "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```
2. Now, using homebrew, install qemu

```
$ brew install qemu
```
3. Get the desired version of Ubuntu ISO (20.04 is used in this experiment):

```
https://releases.ubuntu.com/focal/ubuntu-20.04.5-live-server-amd64.iso
```
4. Create the virtual disk for VM with desired size

```
$ qemu-img create -f qcow2 ubuntu.qcow2 10G
```

5. Now we need to install the Ubuntu VM on the previously created virtual disk

```
$ sudo qemu-system-x86_64 \  
-m 2G \  
-cdrom ./ubuntu-20.04.5-live-server-amd64.iso \  
-drive file=ubuntu.qcow2 \  
-cpu host \  
-machine type=q35,accel=hvf \  

```

Follow on screen instructions to install the OS

6. Now boot into VM using

```
$ sudo qemu-system-x86_64 \  
-m 2G \  
-drive file=ubuntu.qcow2 \  
-cpu host \  
-machine type=q35,accel=hvf \  

```

7. If everything went OK, we will be booted into a virtual Ubuntu environment

- Notes on possible resource flags in command

-m: Changes the amount of memory your VM has allocated

-smp: Changes the cpu configuration

-accel: Changes the form of acceleration your VM will use, eg, kvm, xen, hax, hvf, nvmm, whpx or tcg. Default is tcg

-cpu model: Select CPU model (-cpu help for list and additional feature selection)

-vga type: Select type of VGA card to emulate. Valid values for type are cirrus, std, vmware etc.

-cdrom file: Use file as CD-ROM image (you cannot use -hdc and -cdrom at the same time). You can use the host CD-ROM by using /dev/cdrom as filename.

-serial dev: Redirect the virtual serial port to host character device dev. The default device is vc in graphical mode and stdio in non graphical mode. This option can be

used several times to simulate up to 4 serial ports. Use `-serial none` to disable all serial ports.

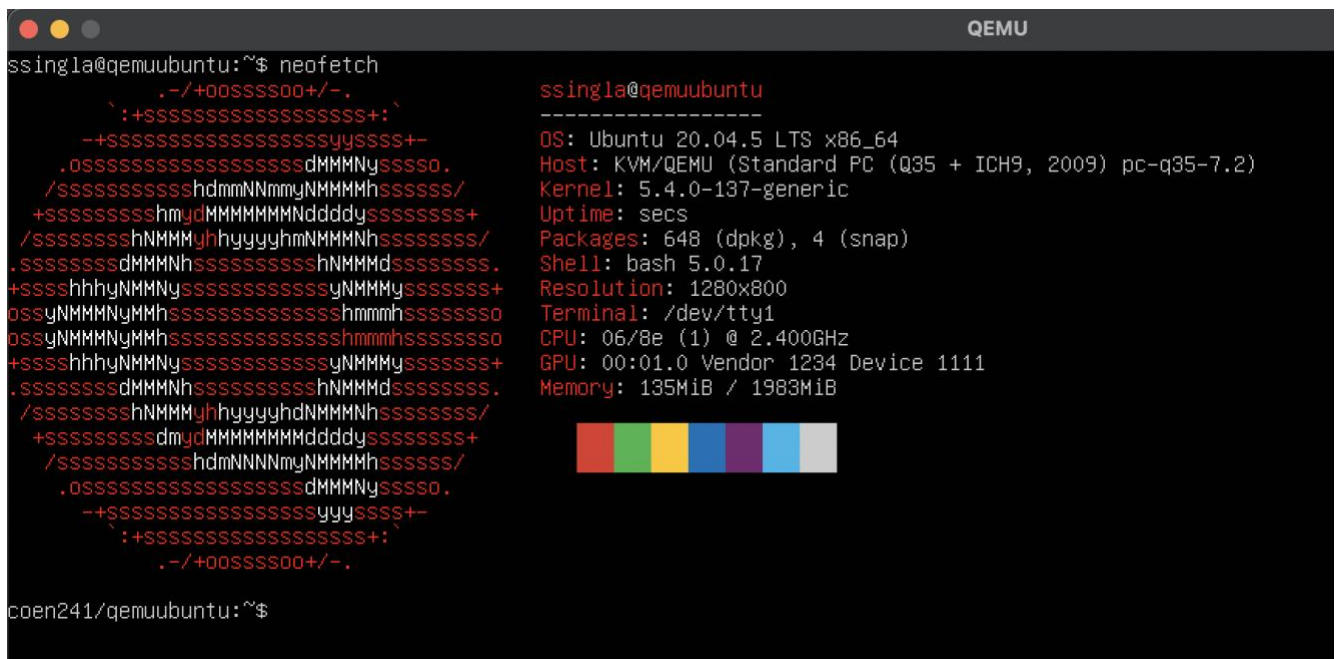
-drive option[,option[,option[,...]]]: Define a new drive. This includes creating a block driver node (the backend) as well as a guest device, and is mostly a shortcut for defining the corresponding `-blockdev` and `-device` options.

-monitor dev: Redirect the monitor to host device `dev` (same devices as the serial port). The default device is `vc` in graphical mode and `stdio` in non graphical mode.

Use `-monitor none` to disable the default monitor.

- My commands for different configurations:
 - `$ sudo qemu-system-x86_64 -m 2G -drive file=ubuntu.qcow2`
 - `$ sudo qemu-system-x86_64 -m 2G -drive file=ubuntu.qcow2 -cpu host -machine type=q35,accel=hvf`
 - `$ sudo qemu-system-x86_64 -m 4G -smp 2 -drive file=ubuntu.qcow2 -cpu host -machine type=q35,accel=hvf`

Screenshots for QEMU:



```
ssingla@qemuubuntu:~$ neofetch
      .-/+00SSSS00+/-.,
      `:+SSSSSSSSSSSSSSSS+:`
      +-SSSSSSSSSSSSSSSSSSSSSS+
      .0SSSSSSSSSSSSSSSSSSSSSSdMMMNySSSS0.
      /SSSSSSSSSSShdmmNNmmyNMMMMhSSSSSS/
      +SSSSSSSSShmydMMMMMMMMNdddySSSSSSSS+
      /SSSSSSSSShNMMMyhhyyyhmNMMMNhSSSSSSSS/
      .SSSSSSSSdMMMNhSSSSSSSSShNMMMdSSSSSSSS.
      +SSShhhyNMMNySSSSSSSSSSSyNMMMySSSSSSSS+
      ossyNMMMNyMMhSSSSSSSSSSShmmhSSSSSSSS0
      ossyNMMMNyMMhSSSSSSSSSSShmmhSSSSSSSS0
      +SSShhhyNMMNySSSSSSSSSSSyNMMMySSSSSSSS+
      .SSSSSSSSdMMMNhSSSSSSSSShNMMMdSSSSSSSS.
      /SSSSSSSSShNMMMyhhyyyhdNMMMNhSSSSSSSS/
      +SSSSSSSSdmydMMMMMMMMNdddySSSSSSSS+
      /SSSSSSSSSSShdmmNNmmyNMMMMhSSSSSS/
      .0SSSSSSSSSSSSSSSSSSSSdMMMNySSSS0.
      +-SSSSSSSSSSSSSSSSSSSSSS+
      `:+SSSSSSSSSSSSSSSS+:`
      .-/+00SSSS00+/-.,

ssingla@qemuubuntu
-----
OS: Ubuntu 20.04.5 LTS x86_64
Host: KVM/QEMU (Standard PC (Q35 + ICH9, 2009) pc-q35-7.2)
Kernel: 5.4.0-137-generic
Uptime: secs
Packages: 648 (dpkg), 4 (snap)
Shell: bash 5.0.17
Resolution: 1280x800
Terminal: /dev/tty1
CPU: 06/8e (1) @ 2.400GHz
GPU: 00:01.0 Vendor 1234 Device 1111
Memory: 135MiB / 1983MiB
```

```
QEMU
ssingla@qemuubuntu:~/coen241/COEN241_CloudComputing/HW1$ sysbench --test=cpu --cpu-max-prime=20000 run
WARNING: the --test option is deprecated. You can pass a script name or path on the command line without any options.
sysbench 1.0.18 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 1
Initializing random number generator from current time

Prime numbers limit: 20000

Initializing worker threads...

Threads started!

CPU speed:
  events per second:   493.04

General statistics:
  total time:          10.0014s
  total number of events: 4932

Latency (ms):
  min:                 1.90
  avg:                 2.03
  max:                 6.19
  95th percentile:    2.39
  sum:                 9997.05

Threads fairness:
  events (avg/stddev): 4932.0000/0.00
  execution time (avg/stddev): 9.9970/0.00

ssingla@qemuubuntu:~/coen241/COEN241_CloudComputing/HW1$
```

Docker Setup:

1. First download docker for mac from the [official website](#), and install it.
2. Docker Hub is the recommended repository to get publicly verified and official Docker images. So using it, we can download the latest image of Ubuntu using the following command:

```
$ sudo docker pull ubuntu
```


Or to get a specific version, we can use:

```
$ sudo docker pull ubuntu:focal
```
3. Now, running the ubuntu image:

```
$ docker run -it --rm ubuntu:focal
```
4. Since it's a very light installation of Ubuntu, its missing some necessary packages for our experimentation, they can be installed using:

```
$ apt update && apt install lsb-core
```
5. If the docker container is stopped in this stage, we'll lose all the changes we have made.

So to save the docker container state:

- a. First, check the container ID, using:

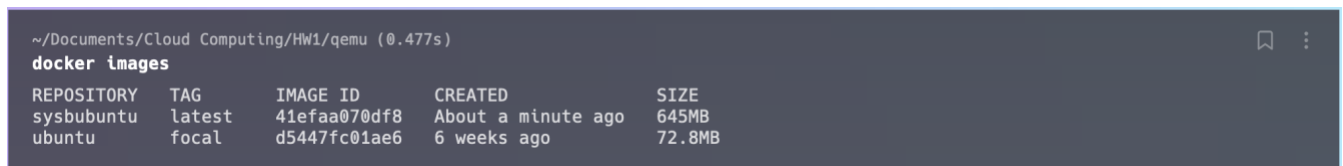
```
$ sudo docker ps
```

- b. Save the state of the container by running the following command:

```
$ sudo docker commit -p container_id sysbubuntu
```

6. To see the current images:

```
$ docker images
```



REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
sysbubuntu	latest	41efaa070df8	About a minute ago	645MB
ubuntu	focal	d5447fc01ae6	6 weeks ago	72.8MB

7. To see the image history:

```
$ docker history sysbubuntu
```

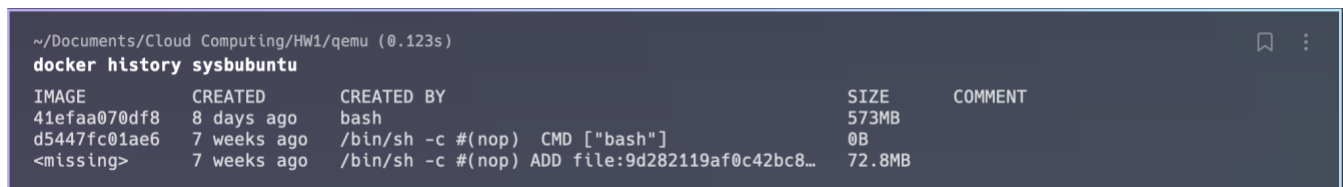


IMAGE	CREATED	CREATED BY	SIZE	COMMENT
41efaa070df8	8 days ago	bash	573MB	
d5447fc01ae6	7 weeks ago	/bin/sh -c #(nop) CMD [\"bash\"]	0B	
<missing>	7 weeks ago	/bin/sh -c #(nop) ADD file:9d282119af0c42bc8...	72.8MB	

Some Docker Operations:

start - Run an existing Docker container

create - Create a Docker container from an image

run - Create a new Docker container and start running it

ls - List all the containers running currently

stop - Stop running containers

rm - Remove a container which is not running

kill - Kill one or more containers

logs - Retrieves logs present at the time of execution

inspect - Get more information about the Containers / Images

build - Build Docker image from a Dockerfile

push - Push an image or a repository to a registry

ls - List all the Images

rm - Delete the image

docker login - Log in to Docker registry to get Images

docker version - Get more information about the Docker client and server versions

docker system prune - Remove all unused data - like containers, networks, images etc

Screenshots of Docker container:

```
~/Documents/Cloud Computing/HW1/qemu
docker run -it sysbubuntu

root@ef867e9cb230:/# neofetch

      .-/+oossssoo+/-.
      `:+ssssssssssssss++`
        -+ssssssssssssssyyssss+-
        .ossssssssssssssdMMMMyssso.
        /ssssssssshdmmNmmymMMMMHssssss/
        +ssssssssshmydMMMMMMNdddyssssss+
        /ssssssshNMMMyhhyyyhmmMMMNhssssss/
        .sssssssdMMMNhssssssssshNMMMdssssss.
        +ssssshhyNMMNyssssssssssyNMMMyssssss+
        ossyNMMMNyMMHssssssssssshmmhssssssso
        ossyNMMMNyMMHssssssssssshmmhssssssso
        +ssssshhyNMMNyssssssssssyNMMMyssssss+
        .sssssssdMMMNhssssssssshNMMMdssssss.
        /ssssssshNMMMyhhyyyhdNMMMNhssssss/
        +ssssssssdmydMMMMMMNdddyssssss+
        /ssssssssshdmmNNNmmymMMMMHssssss/
        .ossssssssssssssdMMMMyssso.
        -+ssssssssssssssyyssss+-
        `:+ssssssssssssss++`
          .-/+oossssoo+/-.

root@ef867e9cb230:/#
```

```

root@ef867e9cb230:/# sysbench --test=cpu --cpu-max-prime=20000 --max-time=20 run
WARNING: the --test option is deprecated. You can pass a script name or path on the command line without any options.
WARNING: --max-time is deprecated, use --time instead
sysbench 1.0.18 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 1
Initializing random number generator from current time


Prime numbers limit: 20000

Initializing worker threads...

Threads started!

CPU speed:
  events per second:   488.63

General statistics:
   total time:          20.0012s
   total number of events: 9774

Latency (ms):
  min:                  1.92
  avg:                  2.04
  max:                  11.15
 95th percentile:      2.39
  sum:                  19986.62

Threads fairness:
   events (avg/stddev): 9774.0000/0.00
   execution time (avg/stddev): 19.9866/0.00

root@ef867e9cb230:/# █

```

Script for running experiments:

This script runs 3 different instance of CPU test with prime number limits of 10000, 20000, 30000 as well as FileIO test with 1GB and 2GB each five times and the result is stored in the output text file.

Example: \$ bash script.sh output.txt

```
outputFile=$1
sync; sh -c "echo 3 > /proc/sys/vm/drop_caches"
touch $outputFile
> $outputFile

fileIoTests(){
    size=$1
    shift
    for i in {1..5};
    do
        sysbench --threads=16 --test=fileio --file-total-
size=$size --file-test-mode=rndrw prepare
        sysbench --threads=16 --test=fileio --file-total-
size=$size --file-test-mode=rndrw run >> $outputFile
        sysbench --threads=16 --test=fileio --file-total-
size=$size --file-test-mode=rndrw cleanup
        sync; sh -c "echo 3 > /proc/sys/vm/drop_caches"
        echo "-----"
    >> $outputFile
    done
}

cpuTests(){
    number=$1
    shift
    for i in {1..5};
```

```

do
    sysbench --test=cpu --cpu-max-prime=$number run >>
$outputFile
    echo "-----"
>> $outputFile
done
}

echo "<===CPU Tests===>" >> $outputFile
echo "Prime numbers limit: 10000" >> $outputFile
cpuTests 10000
echo "Prime numbers limit: 20000" >> $outputFile
cpuTests 20000
echo "Prime numbers limit: 30000" >> $outputFile
cpuTests 30000

echo
"<=====>" >>
$outputFile

echo "<===File IO Tests===>" >> $outputFile
echo "##### 1G 128Files 16Threads" >> $outputFile
fileIoTests 1G
echo "##### 2G 128Files 16Threads" >> $outputFile
fileIoTests 2G

```

Usage of performance tools:

Sysbench is the tools used to collect and show performance data of a virtualized linux environment space.

- The sysbench commands used
 - sysbench -test=cpu [FLAGS] run
 - --cpu-max-prime=N: Prime generator test for cpu

- sysbench --test=fileio [FLAGS] [prepare|run|cleanup]
 - --file-num=N: Number of files, default amount is 128
 - --file-total-size=SIZE: Size of all files combined
 - --file-test-mode=STRING: Type of FileIO test: seqwr, seqrewr, seqrd, rndrd, rndwr, or rndrw

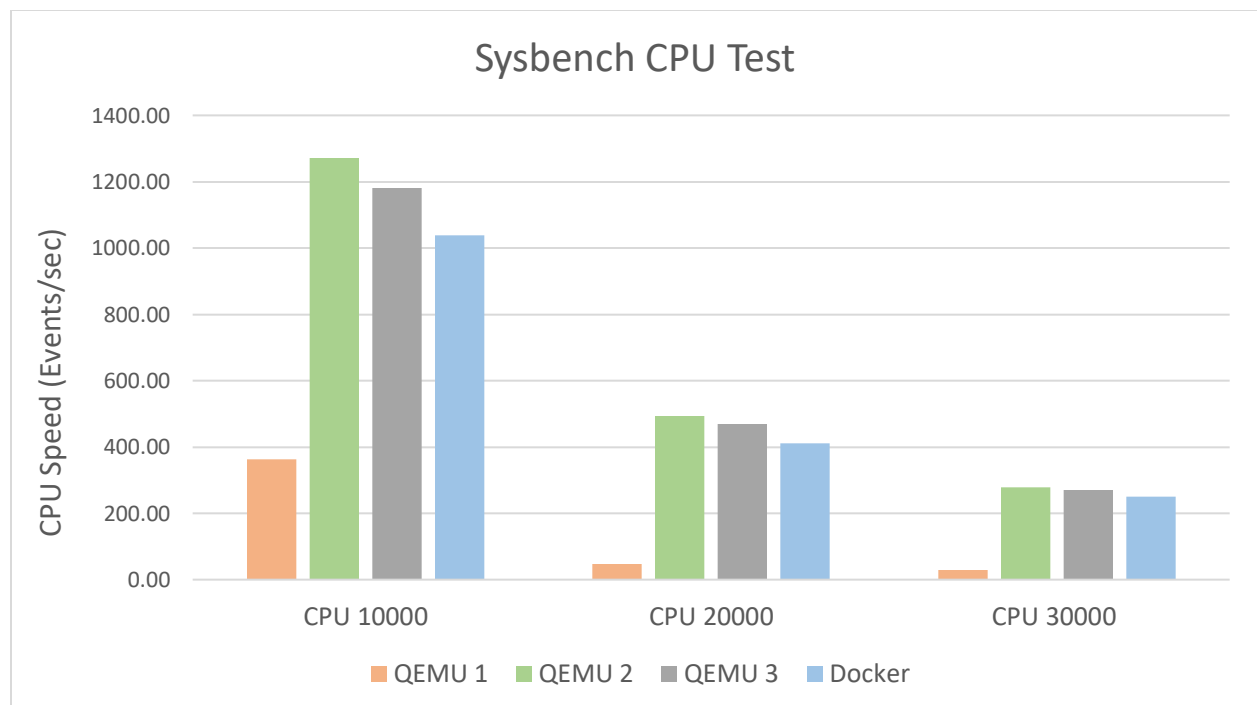
If the resource allocation of both environments were the same then much could be said about how CPU utilization differs in the user-space vs the kernel space however due to the erratic nature of my configurations we cannot make these connections. The fileIO throughput can be seen in the kb/sec measurement, the latency would be seen if you compare the “total time taken by event execution” of 2 different systems, and the disk utilization can be calculated by taking the total operations performed and dividing that by the “total time taken by event execution” resulting in a disk operations per second measurement.

Data Collection and Analysis:

- **CPU Test:**

		QEMU 1		QEMU 2	
		Exec Time	CPU speed	Total Time	CPU speed
CPU 10000	1st run	9.9478	402.40	9.9924	1259.88
	2nd run	9.8892	343.69	9.9887	1275.11
	3rd run	9.9568	368.57	9.9932	1266.59
	4th run	9.9644	382.02	9.9933	1275.33
	5th run	9.9400	320.89	9.9938	1281.11
	Avg	9.9396	363.51	9.9923	1271.60
CPU 20000	1st run	9.9458	52.20	9.9956	495.20
	2nd run	9.9806	42.43	9.9959	494.01
	3rd run	9.9267	44.56	9.9959	494.63
	4th run	9.9281	48.95	9.9962	491.74
	5th run	9.9442	50.96	9.9956	488.65
	Avg	9.9451	47.82	9.9958	492.85
CPU 30000	1st run	9.9473	27.65	9.9986	284.20
	2nd run	10.0104	30.20	9.9963	266.92
	3rd run	9.9416	29.36	9.9986	278.21
	4th run	9.9541	26.54	9.9981	283.31
	5th run	9.9849	30.89	9.9975	282.82
	Avg	9.9677	28.93	9.9978	279.09

		QEMU 3		Docker	
		Total Time	CPU speed	Total Time	CPU speed
CPU 10000	1st run	9.7179	1196.27	9.9478	1018.06
	2nd run	9.7165	1187.70	9.8806	870.69
	3rd run	9.7167	1170.55	9.9841	1111.12
	4th run	9.7217	1172.90	9.9666	1078.50
	5th run	9.7151	1174.00	9.9785	1117.70
	Avg	9.7176	1180.28	9.9515	1039.21
CPU 20000	1st run	9.8844	474.44	9.9819	427.32
	2nd run	9.8855	469.44	9.9906	443.34
	3rd run	9.8858	456.74	9.9910	437.83
	4th run	9.8834	475.11	9.9921	439.72
	5th run	9.8847	471.69	9.6318	309.87
	Avg	9.8848	469.48	9.9175	411.62
CPU 30000	1st run	9.9329	267.23	9.9858	243.10
	2nd run	9.9310	274.24	9.9960	265.99
	3rd run	9.9345	274.68	9.9950	255.09
	4th run	9.8979	262.27	9.9859	236.84
	5th run	9.9319	270.99	9.9923	245.63
	Avg	9.9256	269.88	9.9910	249.33



- Averages:

	QEMU 1	QEMU 2	QEMU 3	Docker
CPU 10000	363.51	1271.60	1180.28	1039.21
CPU 20000	47.82	492.85	469.48	411.62
CPU 30000	28.93	279.09	269.88	249.33

In the above chart (By plotting all the averages) we can clearly see that QEMU 1 is the worst performant of all as there was no hardware acceleration.

Moreover QEMU 2 has the best performance among all as it is using the latest acceleration method, hvf, which provides almost native like speeds.

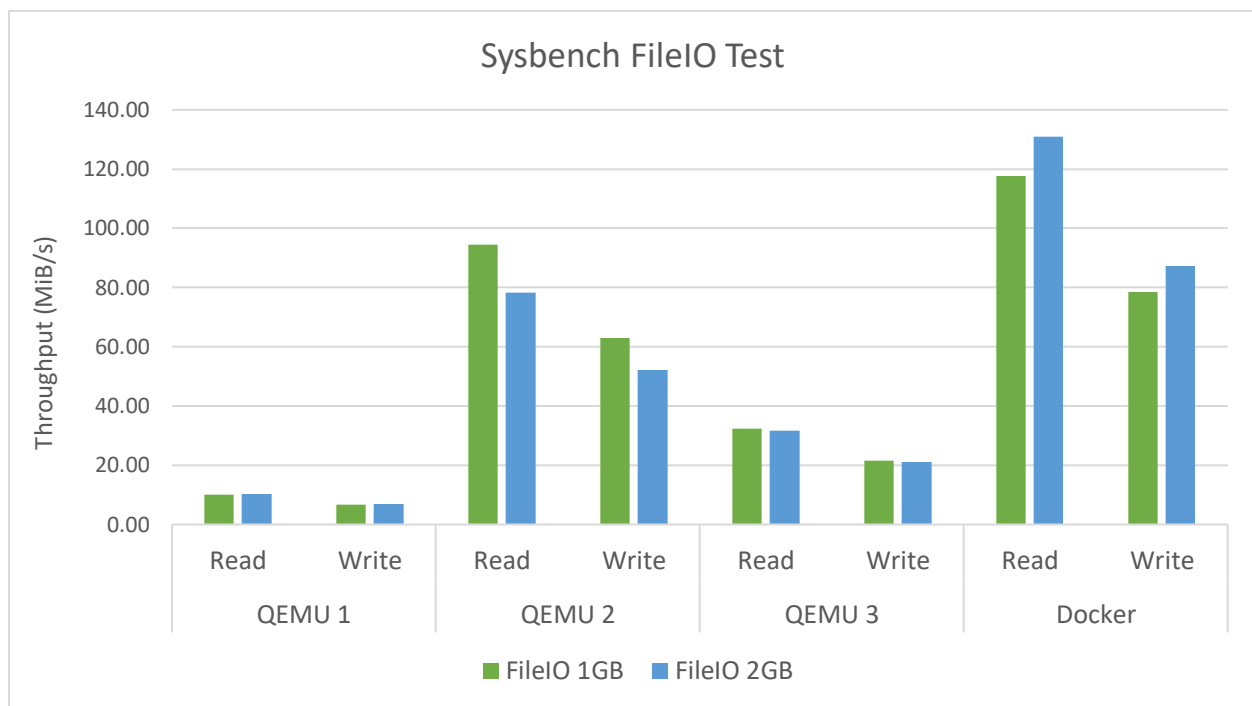
On the other hand, QEMU 3 has seen a slight dip in performance, even though it has almost double the resources of QEMU 2, this might be because, the CPU test mostly measures single core performance and doesn't consider the multi-core nature of the CPU.

For Docker, I was expecting to see the highest performance because it's the lightest of all the systems, but that is not the case. It seems like QEMU hardware acceleration for intel-based mac is top notch.

▪ FileIO Test:

		QEMU 1			QEMU 2		
		Total Time	Throughput		Total Time	Throughput	
			Read(MiB/s)	Write(MiB/s)		Read(MiB/s)	Write(MiB/s)
FileIO 1GB	1st run	21.0598	1.2900	0.8600	10.1968	93.2100	62.1400
	2nd run	10.4566	16.2200	10.8100	10.1787	94.8800	63.2500
	3rd run	12.1544	1.3900	0.9300	10.2099	95.1100	63.4100
	4th run	10.4238	14.9100	9.9500	10.2059	94.6200	63.0800
	5th run	10.4114	16.3800	10.9200	10.1673	94.5000	63.0000
	Avg	12.9012	10.0380	6.6940	10.1917	94.4640	62.9760
FileIO 2GB	1st run	10.3915	12.3600	8.2400	10.1830	76.6800	51.1100
	2nd run	12.7631	9.7700	6.5100	10.1798	80.0200	53.3400
	3rd run	13.4871	1.6000	1.0700	10.2208	78.3300	52.2100
	4th run	10.4007	14.9600	9.9700	10.1780	76.7100	51.1300
	5th run	10.4379	12.8300	8.5600	10.1795	79.3600	52.9000
	Avg	11.4961	10.3040	6.8700	10.1882	78.2200	52.1380

		QEMU 3			Docker		
		Total Time	Throughput		Total Time	Throughput	
			Read(MiB/s)	Write(MiB/s)		Read(MiB/s)	Write(MiB/s)
FileIO 1GB	1st run	10.3124	30.6300	20.4200	10.0631	127.2400	84.8300
	2nd run	10.3014	32.6700	21.7800	10.0621	127.4300	84.9600
	3rd run	10.3319	32.3900	21.5900	10.0676	67.3100	44.8800
	4th run	10.3041	33.0200	22.0200	10.0420	120.6000	80.4000
	5th run	10.3339	32.4900	21.6600	10.0585	145.2700	96.8500
	Avg	10.3167	32.2400	21.4940	10.0587	117.5700	78.3840
FileIO 2GB	1st run	10.3179	31.9800	21.3200	10.0508	136.1600	90.7800
	2nd run	10.2995	32.0400	21.3600	10.2070	106.1500	70.7700
	3rd run	10.2977	31.9500	21.3000	10.0403	151.7000	101.1400
	4th run	10.3280	31.2200	20.8100	10.0654	152.3500	101.5700
	5th run	10.3365	31.2900	20.8600	10.0508	108.4700	72.3100
	Avg	10.3159	31.6960	21.1300	10.0829	130.9660	87.3140



○ Averages:

	QEMU 1		QEMU 2		QEMU 3		Docker	
In(MiB/s)	Read	Write	Read	Write	Read	Write	Read	Write
FileIO 1GB	10.04	6.69	94.46	62.98	32.24	21.49	117.57	78.384
FileIO 2GB	10.30	6.87	78.22	52.14	31.70	21.13	130.97	87.314

For FileIO tests, almost all the configs have similar results as of CPU tests.

QEMU 1, having no hardware acceleration is the worst performant.

Second best performance can be seen with QEMU 2 as it has the latest accel method.

Here also, QEMU 3 has seen a significant dip in performance, might be because the test is designed for single core processors.

Docker as expected, has the best read and write throughput of all.

Final Thoughts:

Usually, docker containers have the best performance of all, but in case of hvf accel QEMU has better optimizations and performs better in some cases. This can be verified with the collected results. So, it can be said containers are good choice for virtualization but security point of view Virtual machines are superior.

GitHub Repository Information:

https://github.com/shubhamsingla27/COEN241_CloudComputing.git

HW1: https://github.com/shubhamsingla27/COEN241_CloudComputing/tree/main/HW1