Siena Hanna

29 January 2023

COEN 241

Dr. Sean Choi

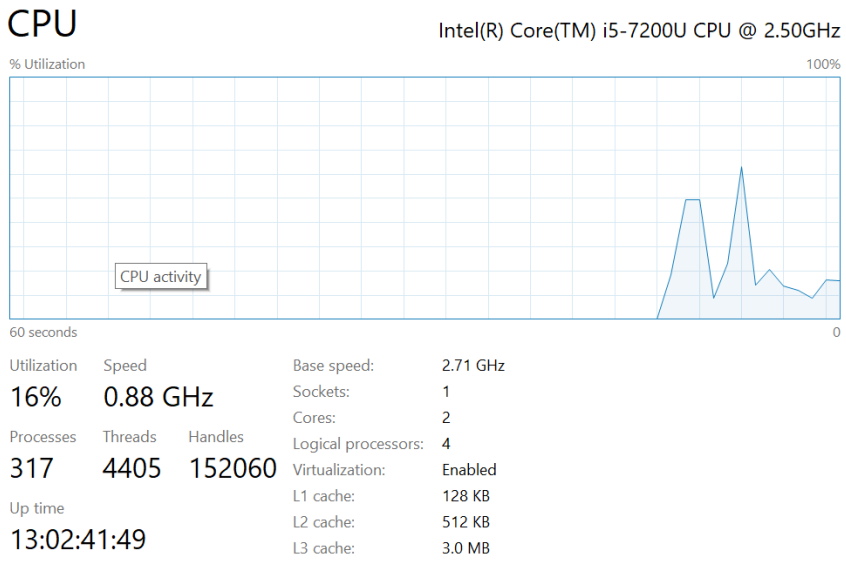
Docker Containers vs QEMU VM Experiments

Environment/Experimental Setup

My host computer has the following specifications:

- 8.00 GB RAM
- OS: Windows 10 Pro
- Cores: 2

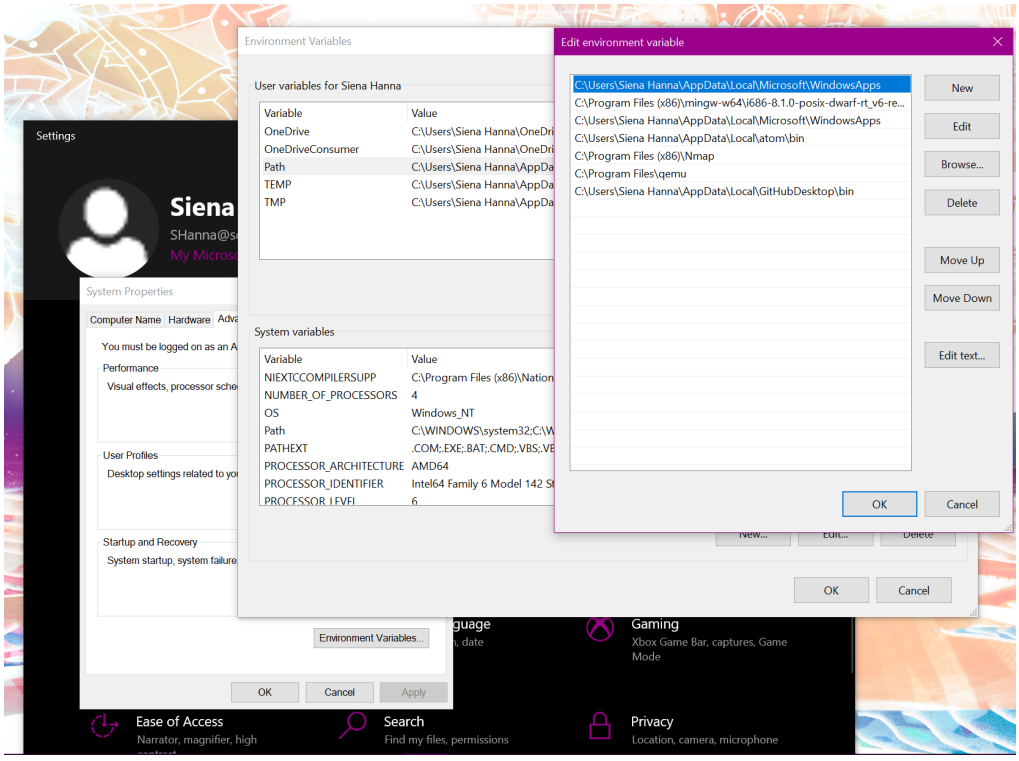
Screenshots for RAM/Cores:



For the container and QEMU VM, I ran them with the -m option to create my 3 scenarios and assigned a maximum of 1024MB (1GB), 2048MB (2GB), and 3072MB (3GB) of RAM (less than half of my computer's RAM). There will be further details for this in the "Experimental Scenarios" section on page 8.

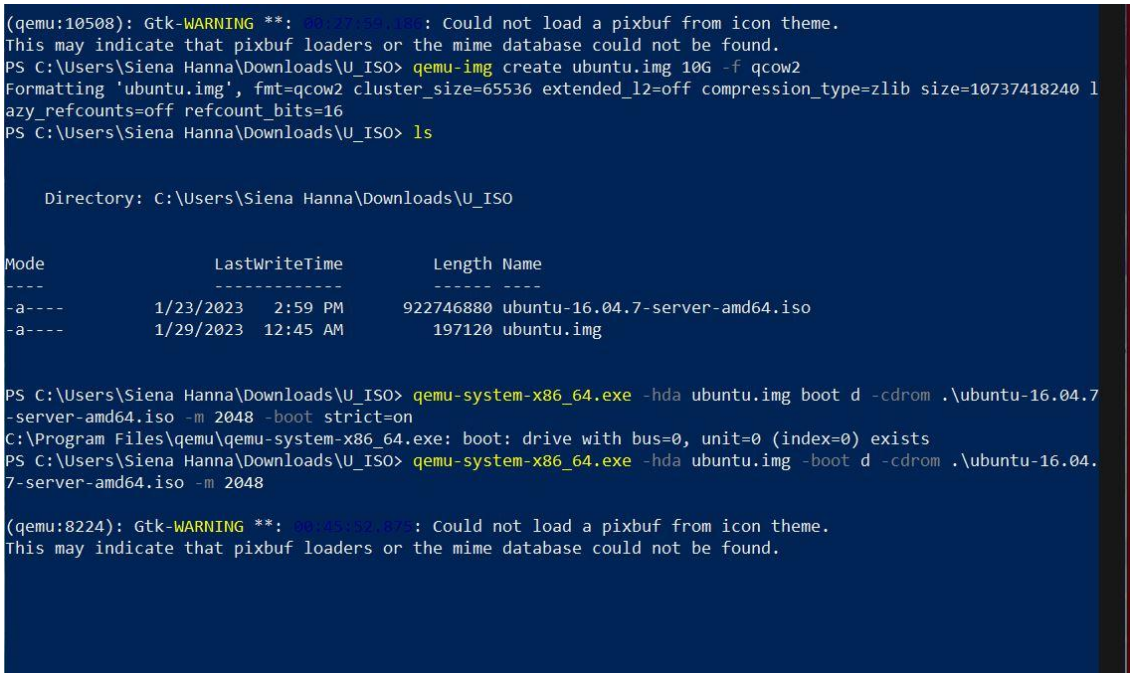
QEMU Setup

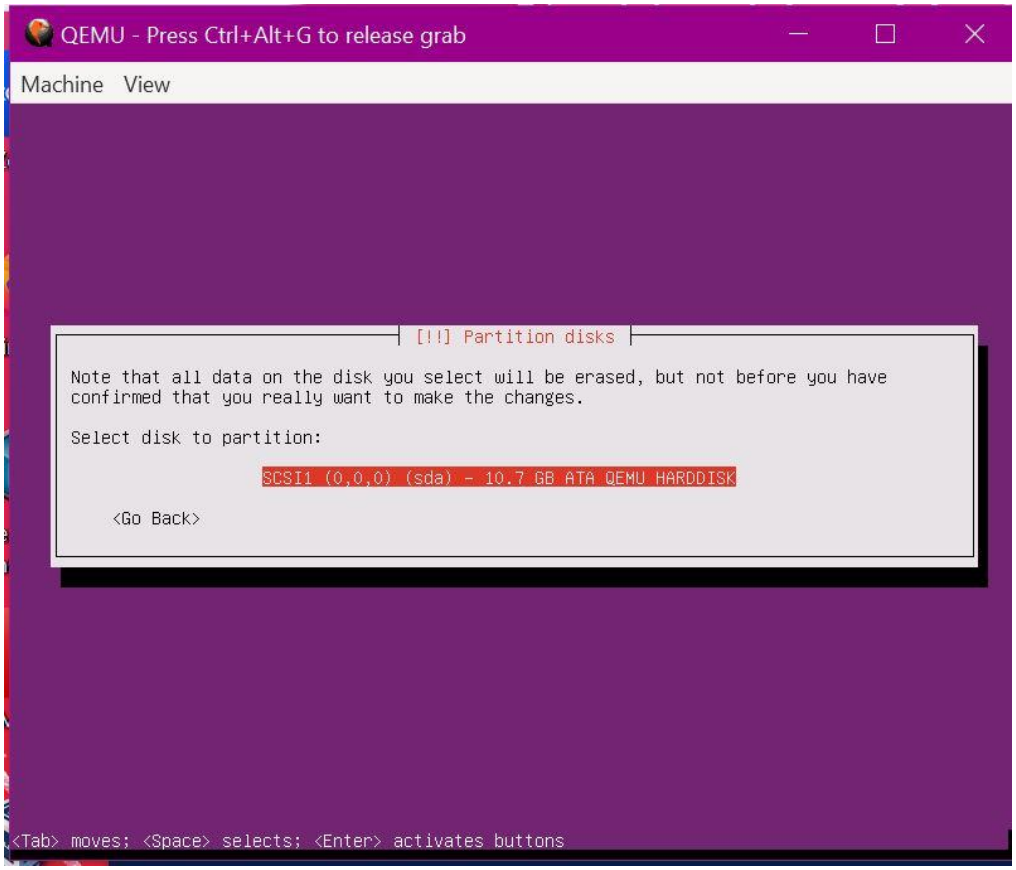
To enable a QEMU VM on Windows, there were a number of specific steps that I needed to take, as recommended by the homework assignment and detailed in https://linuxhint.com/qemu-windows/. First, I downloaded and ran the QEMU installer that was appropriate for Windows. Then once installed, I went to Settings->Advanced system settings-> Environment Variables. I double-clicked on path and added the filepath for qemu to my environment variables (so that I can easily run qemu in any directory). The below image shows that qemu is now added to the path in the second-last position in "Edit environment variable."



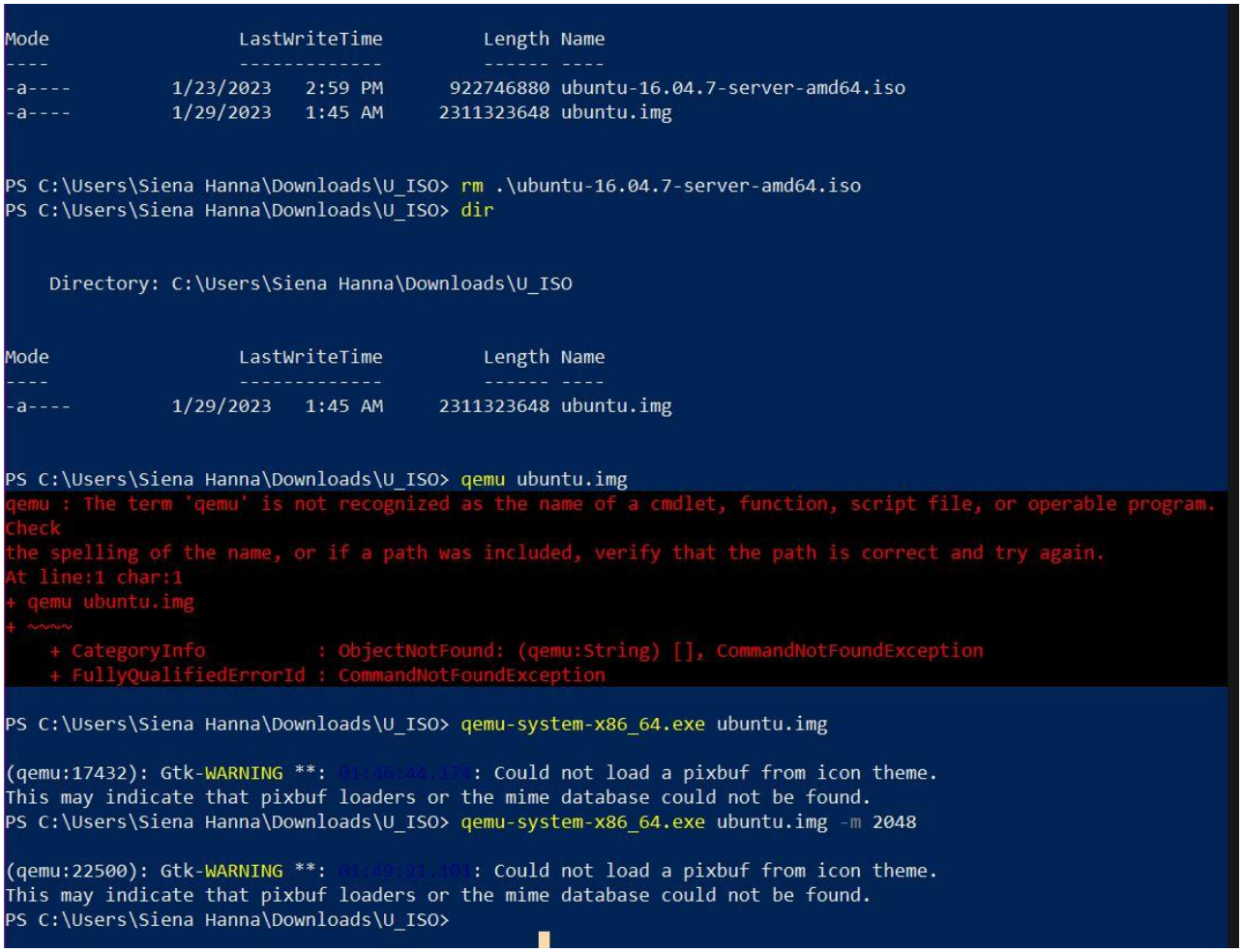
I also downloaded the appropriate .iso file, which was for Ubuntu 16.04 Server (recommended by the assignment for Windows), created a directory called U_ISO for it, and navigated to the directory in PowerShell. I originally tried to boot it using "qemu-system-x86_64.exe -boot d -cdrom .\ubuntu-server-amd64.iso -m 2048." qemu-system-x86_62.exe is the correct command to launch QEMU as a 64-bit VM. The -boot d option indicates that I was booting the first virtual CD-ROM drive, the .iso file. I used -m 2048 to assign RAM. The installer came up, but when I tried to go through the installation, I would not be able to complete installation because the VM could not find a disk to partition.

I was eventually able to fix this using "qemu-img create ubuntu.img 10G -f qcow2" to create a "disk image" of 10GB in qcow2 format for the VM to be installed on, and then I added "-hda ubuntu.img" to my launch command, which sets a virtual hard drive, and finally the installer was able to scan for it. The following screenshots show the creation of the virtual disk, re-launch of the VM, and successful disk partition as part of the Ubuntu Server 16.04 installation.





After I finished installation of Ubuntu 16.04 Server as a QEMU VM, I deleted the .iso file that I originally used to create the VM. Thereafter, I used "**qemu-system-x86_64.exe ubuntu.img**" to launch the VM, with various RAM using the "**-m**" option. Please note that using simply "qemu" does not work. The image on the next page shows the removal of the .iso and launching the VM. with qemu (failed) and properly with **qemu-system-x86_64.exe**.



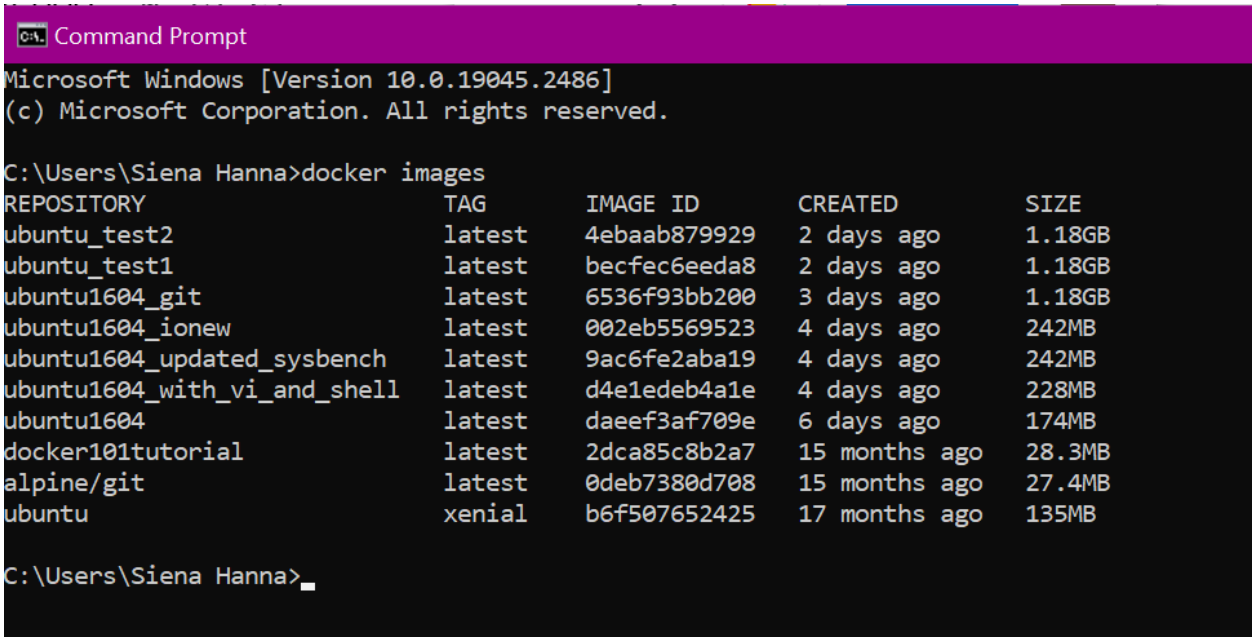
Docker Container

For this experiment, I specifically wanted to use the same version of Ubuntu as my QEMU VM so that the experimental results would be more comparable. Therefore, when I pulled the base image, I used “docker pull ubuntu:xenial,” since including the “xenial” tag would ensure that the image would be of Ubuntu 16.04 rather than the default, latest version.

These images show the final record of my Docker images and the history of my final image, ubuntu_test2. From ubuntu:xenial, I did various updates and installation of commands, such as vi, sh, sysbench (and updated versions), and finally git. Other images are mostly committed due to file changes since that was before I set up my GitHub repo.

After creating the ubuntu_test1 image, I started git, so for ubuntu_test2, I simply used "git pull/push" to keep the files updated instead of using "docker commit" to make a new image. This is because I had already installed all the tools I needed and did not see the need to update the image because the files that I changed were already committed on git, and I could always pull them down to the next container I ran based on the same image. Only one time did I accidentally close the container (after running my whole battery of tests on that experimental scenario) without using git push, and I never did it again because I had to re-run over half an hour of tests.

My Docker images and ubuntu_test2 history:



These are the main commands I found myself running when working with Docker.

- "**docker images**": see what images there are
- "**docker image rm <IMAGE>/docker rmi <IMAGE>"**: remove a specific image. Images that are base layers of other images cannot be removed
- "**docker commit -m "<message>" <CONTAINER ID> <new image name**>": commits the current container state to a new image
- **"docker history <IMAGE>**": show the history of an image (shows the image layers the image is based off of and any commit messages). You can use the name under "REPOSITORY" or the "IMAGE ID" (in output of "**docker images"** to indicate what image you want to look at the history for.
- "**docker pull <IMAGE>"**: get an image from Docker, automatic tag is latest — if you want anything but latest you have to use format image:tag
- "**docker run <IMAGE>"**: makes and runs a container based on the given image. If the image is not local, it will automatically pull it down before running.
    - **-m:** set upper limit of memory (has a minimum)
    - "**-it**": create interactive bash shell for the container
- "**docker ps**": see what containers are currently running
- "**docker kill <CONTAINER ID>"**: if a container is running, this will kill it (use if something is running in docker ps that you want to stop)

To run the containers directly in Command Prompt, I usually used "**docker run --rm -it --entrypoint /bin/bash -m <number>M ubuntu_test2.**" -m indicates the with the option for -m being 1024M/2048M/3072M based on which environmental setup it was, and "ubuntu_test2" being my final image name.

Interestingly, I also found out that “docker run” appears to require any options to the command be used *before* the image name, otherwise it will not work. Also, at one point, I accidentally ran it with /bin/sh vs /bin/bash due to a typo and the latter is widely better because I can use autocompletion and the up arrow to access previous commands like I would in a normal Linux terminal (with the former I have to type everything out).

Experimental Scenarios

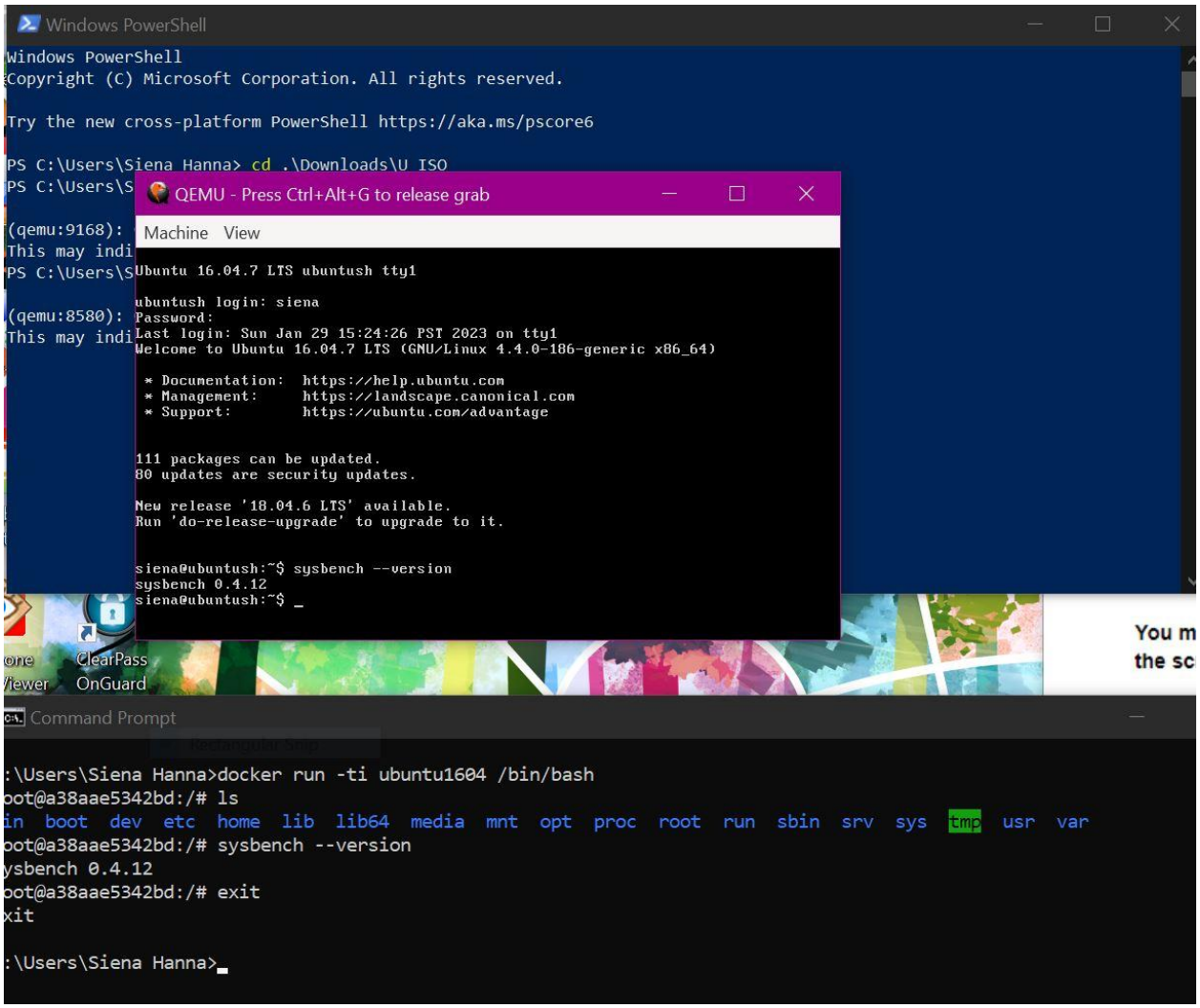
One of my major goals was to conduct similar experiments for the VM vs. the container. However, this was a slightly complicated prospect, as running QEMU virtual machines and running Docker containers have varying options that do not necessarily correspond. I was curious about using the -accel option for QEMU, but as I have a Windows host, I could not use “-accel kvm” since that is not the correct hardware accelerator for Windows. If I were to use -accel, I would have to use “-accel hax” and download additional software and take other steps such as disabling Hyper-V, so I decided not to do so.

Ultimately, it became clear that both virtualization technologies allowed for a certain amount of memory (or memory cap) to be specifically allowed to be used by the VM or container with the -m option for both command lines to run the container/launch the VM. I chose 1024, 2048, and 3072MB as the three major options for both. I selected these options because I wanted to use less than half of the RAM on my host computer (so <4GB), and these options corresponded to 1, 2, and 3 GB.

Performance Tools for Data Collection

I installed sysbench on both the QEMU VM and Docker container (and committed the container, then at image ubuntu:xenial, to a new image, ubuntu1604) and VM as part of my

initial setup. I checked that I had the same version on both, shown in the screenshot below.



I also ensured that top was installed on both to check user vs. system CPU utilization for the CPU tests. These are the major two tools I used to collect my information.

However, when I ran a sysbench fileio test for the first time, I realized that version 0.4.12 of sysbench was not what I wanted, since it did not show throughput, and my desired measurements were **throughput and latency** (as well as recording the **total file size (disk utilization)** for each experiment). Therefore, I used apt-get install sysbench=1.0.20-1 on both my setups to get a more updated version that would output the desired measurements.

For the CPU test, I used top to record the percentage of **CPU usage by the user (us) vs. system (sy)**. I also initially tried to see how long the sysbench cpu test with max-primes=20000 took for different environments, but it always took approximately 10 seconds. I realized this was because, for the updated version of sysbench, the CPU test by default goes for a maximum of 10 seconds. When I had run the earlier version which does not have a time limit by default, it ran for ~48 seconds on Docker with 3GB RAM, so I expected that limiting the test to **30 seconds** would be appropriate for all the scenarios, since they would use less or the same amount of RAM. Since it was no longer appropriate to record the amount of time for the CPU test since each one would have the same limits, I instead recorded the **CPU speed (events per second) and latency**, both of which are shown as part of the systench cpu test output.

Shell Scripts

Once I knew what data I wanted to collect and what tools to use, I was ready to set up my automated experiments using shell scripts.

Please note that my shell scripts (in the same directory in GitHub) are all organized in a hierarchical manner. First, I experimented with customizable configurations of single runs, then made a script to automate the 5-time repetition required for those single runs, then made a (very short) script to run the entire experiment with selected parameters for the experiments.

I/O experiments: **mode.sh, io.sh, and io_test.sh**. I experimented with doing different individual experiments by passing command-line arguments to mode.sh, which runs sysbench prepare, run (30s time), and cleanup for a given fileio mode and total file size. The mid-level file, io.sh, takes two arguments (delay between experiments and file size) and then runs 5 repetitions of experiments with mode.sh for fileio modes seqwr/rd and rndwr/rd and writes the results of the

experiments to differently-named files. The top-level file, io_test.sh, simply runs io.sh for a between-test delay of 10 seconds (to allow me to run sync in between) and file sizes of 64MB and 1GB.

CPU experiments: **top.sh, cpu.sh, and cpu_test.sh.** Firstly, I wanted a way to automatically run top (in the background) halfway through each experiment to show user and system CPU utilization, so I created top.sh to automate this and output the results to different files (with arguments indicating half the test time, the whole test time, and the number of threads). Secondly, cpu.sh runs top.sh at the beginning and then does 5 iterations of sysbench in cpu mode, with command-line arguments indicating number of threads, max primes, and max test time. Finally, cpu_test.sh runs cpu.sh with 30 seconds test time (15 seconds half-time) and max-primes=20000 for 1 and for 8 threads.

All I have to do to run the experiments is run **sh cpu_test.sh** and **sh io_test.sh** (and, for the latter, run sync.exe to clear the cache at appropriate times on my host computer).

Proof of Experiments:

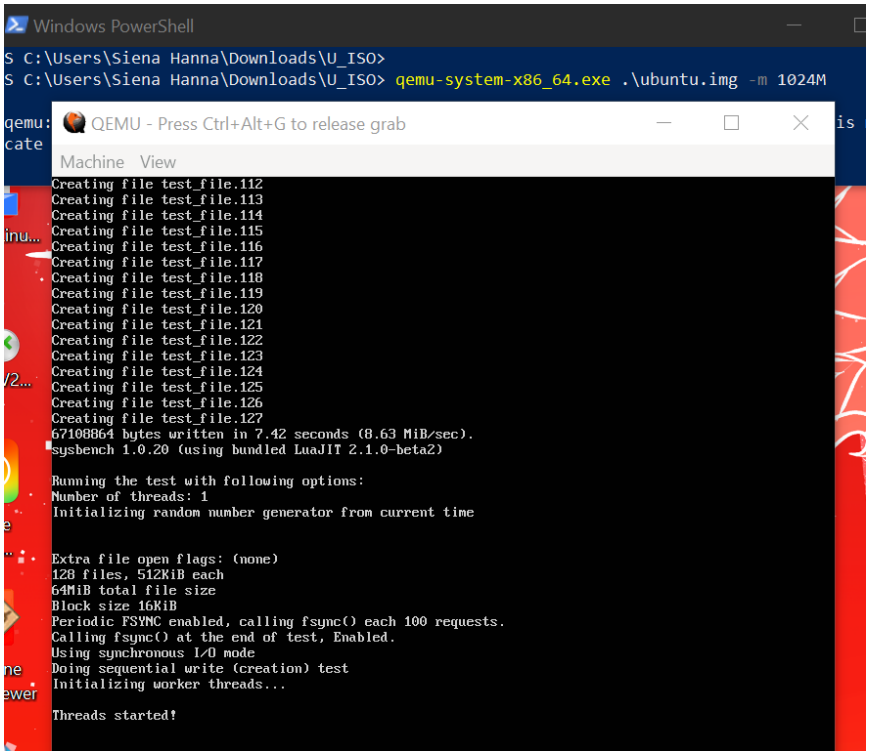
Please note that the output of each experiment or experimental run is saved to a file in the HW1 folder stored in the appropriate test folder (named in format <q/d><memory><io?>). The q indicates QEMU while d indicates a Docker container, the number indicates the MB of memory allocated to that experimental scenario, and the io tests are specified with “io” at the end since they were done after the CPU tests (CPU tests are in files labeled <q/d><memory>). **I have included screenshots of some individual experiments I did before the final runs as well as running the actual shell scripts in the 6 scenarios, but please note that the results of the final experiments (test output) for the shell scripts is saved to**

files in the appropriate directories rather than shown in the terminal and screenshotted.

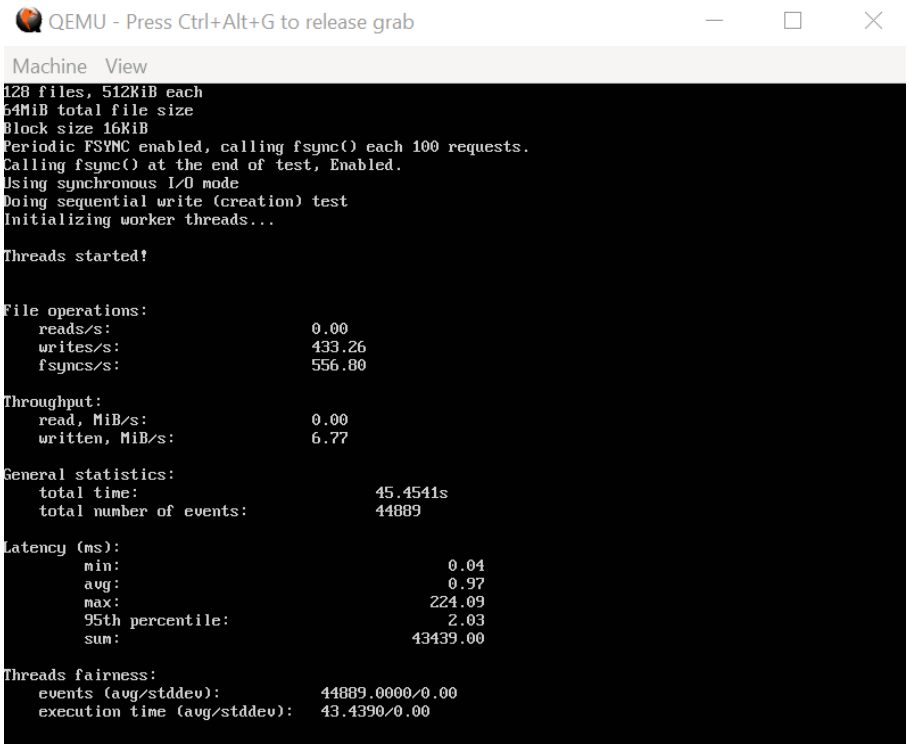
The data for the final results is from these files.

QEMU 1024MB RAM:

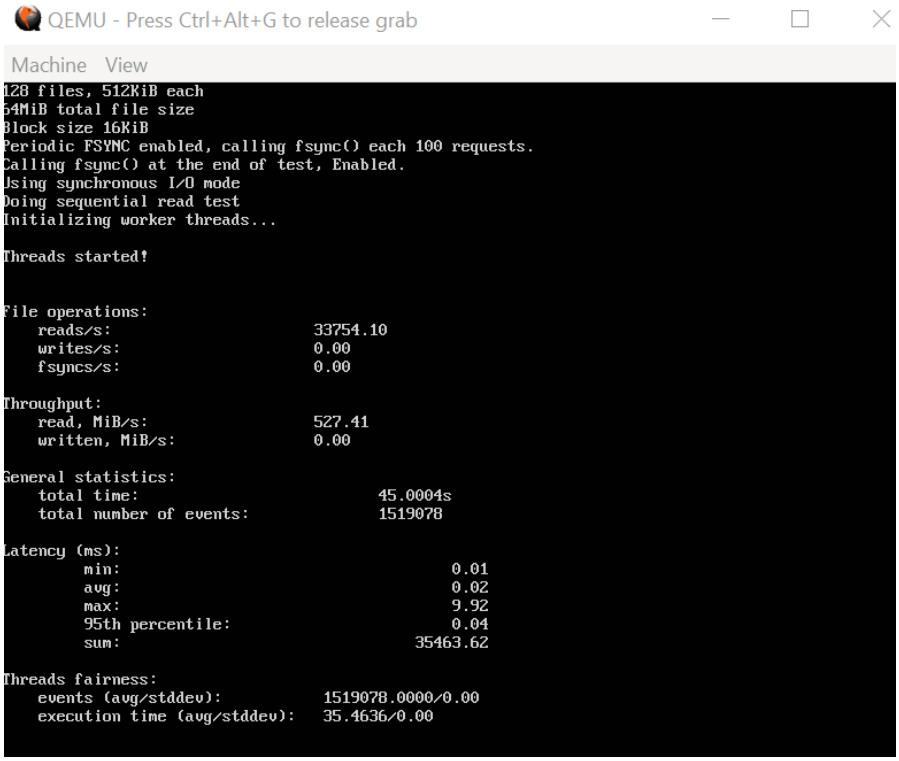
Sample of all modes without output file:



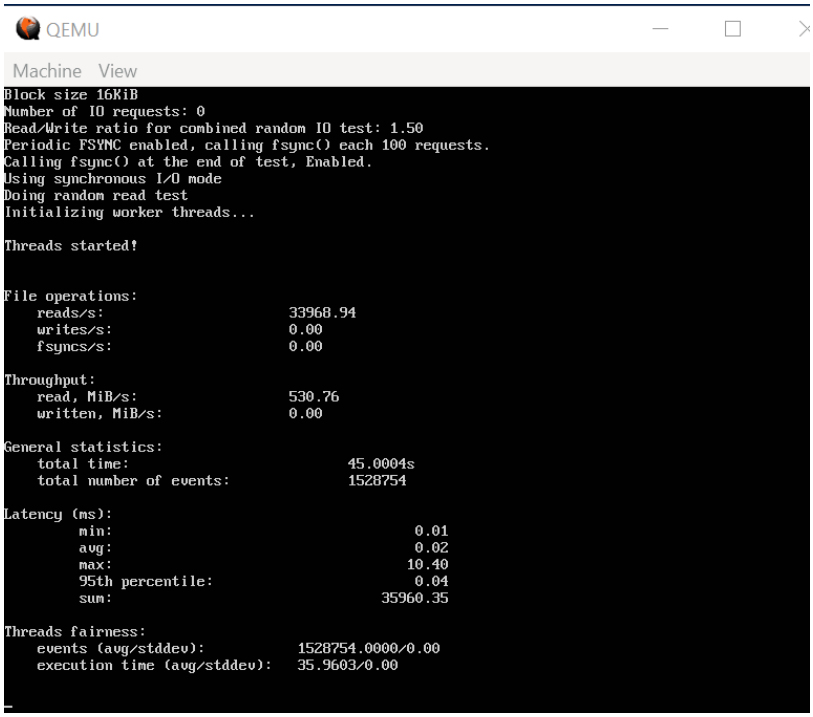
Sequential write (seqwr):



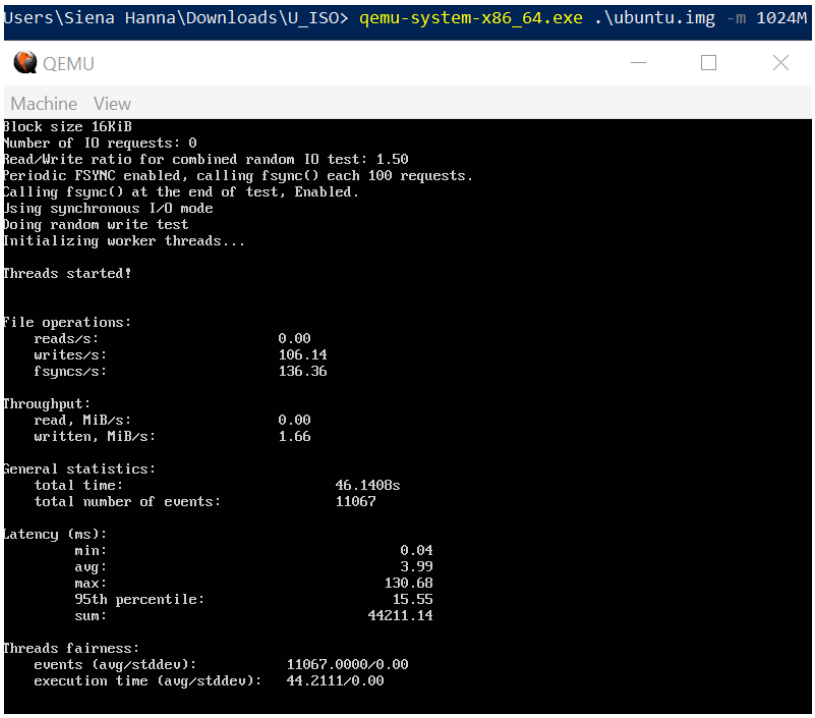
Sequential read (seqrd):



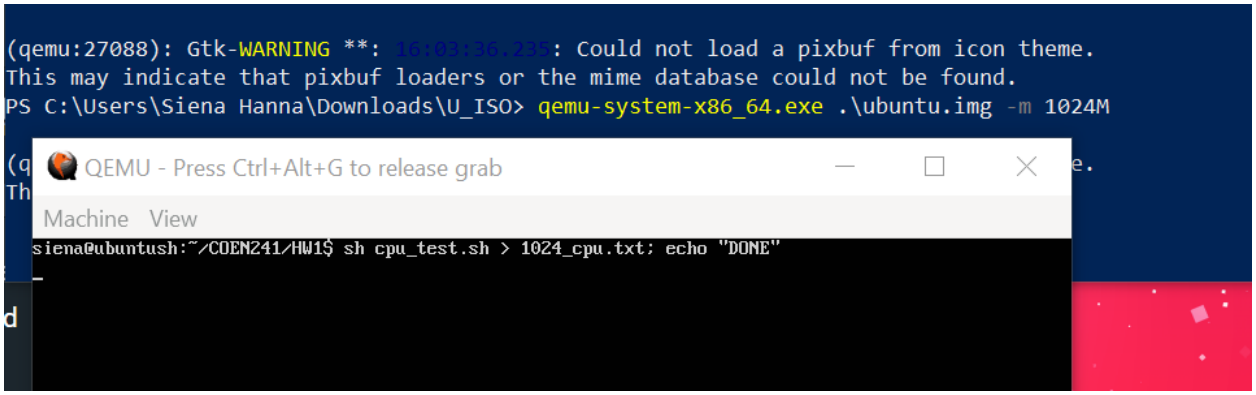
Random read (rndrd):

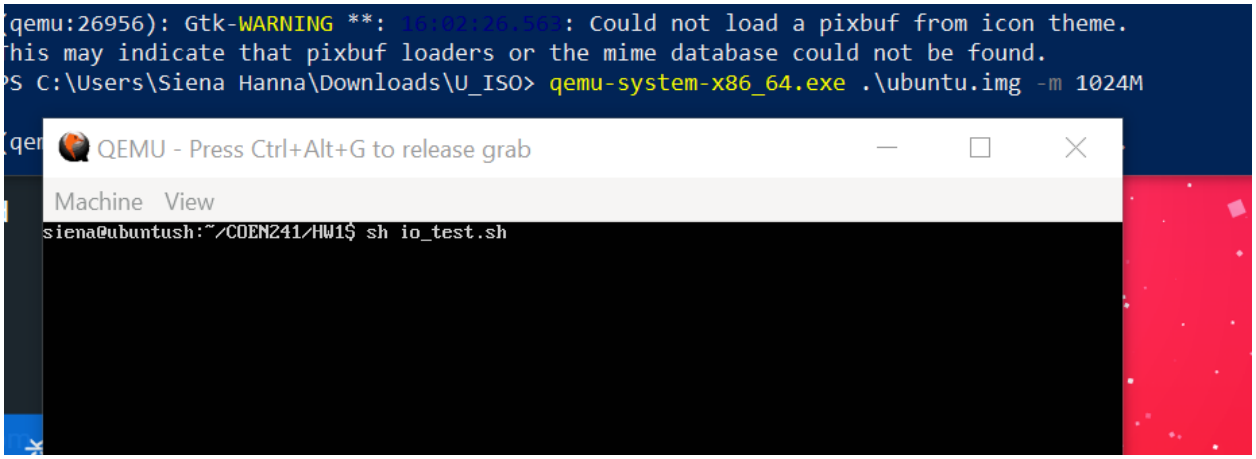


Random write (rndrw)



With output files:

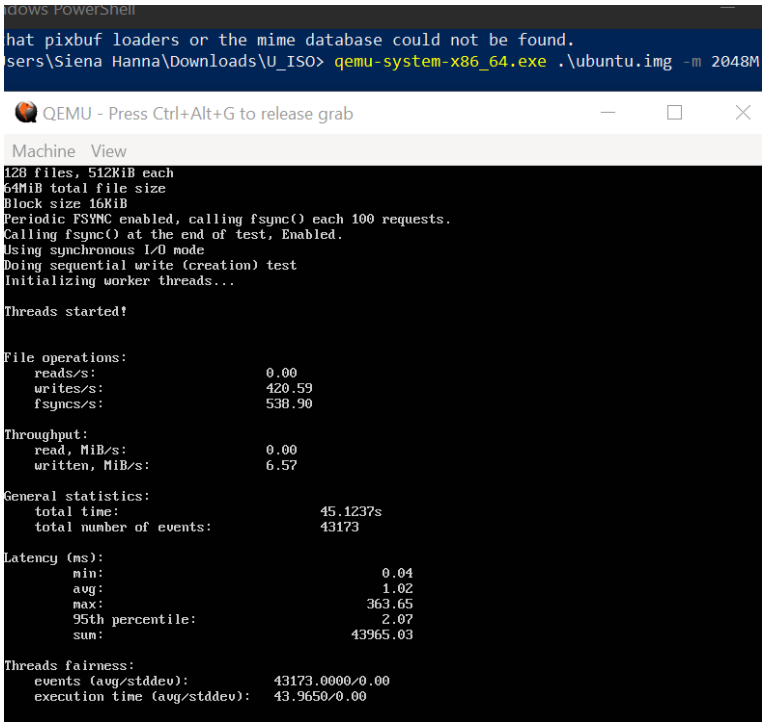




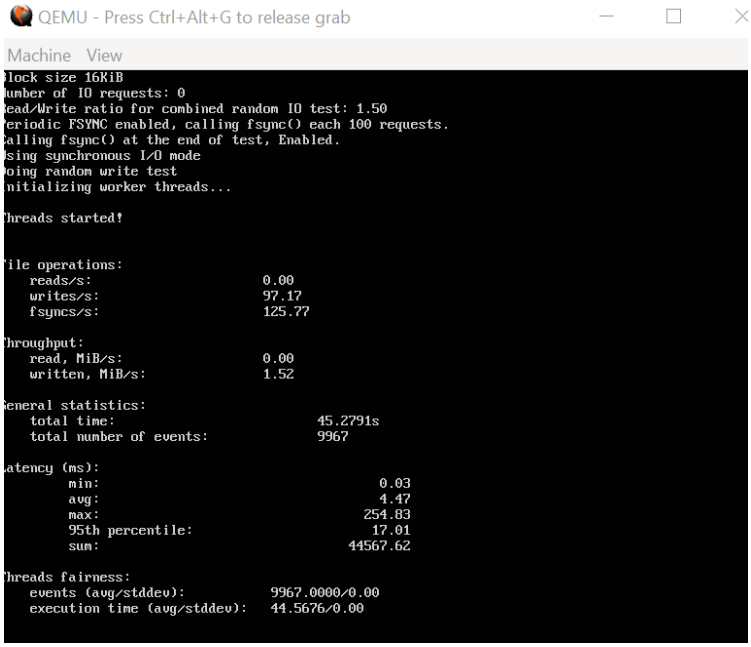
QEMU -m 2048MB:

Sample test without output files:

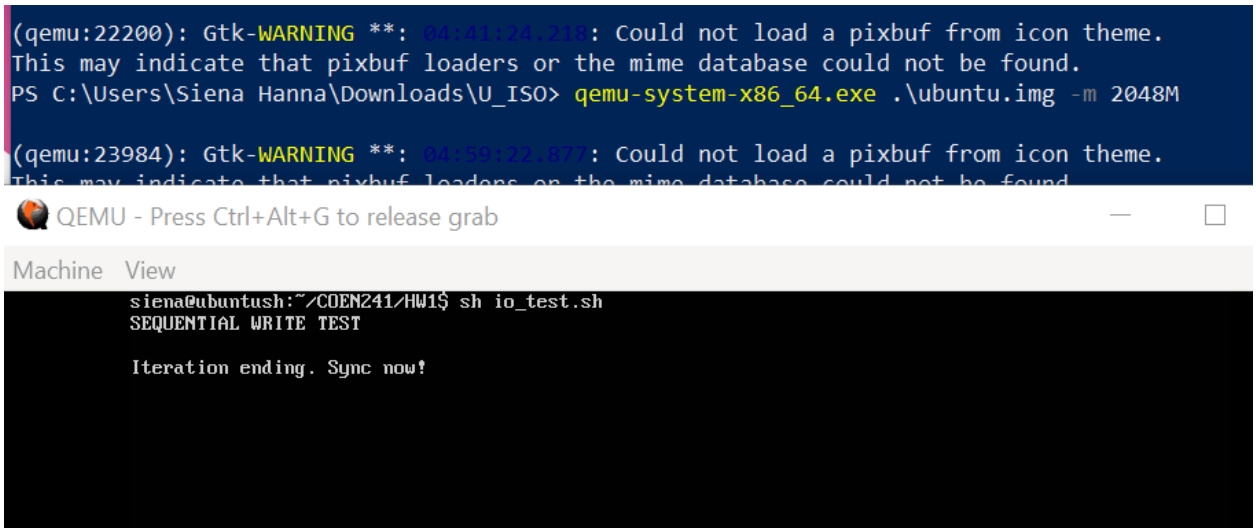
Sequential write (seqwr):

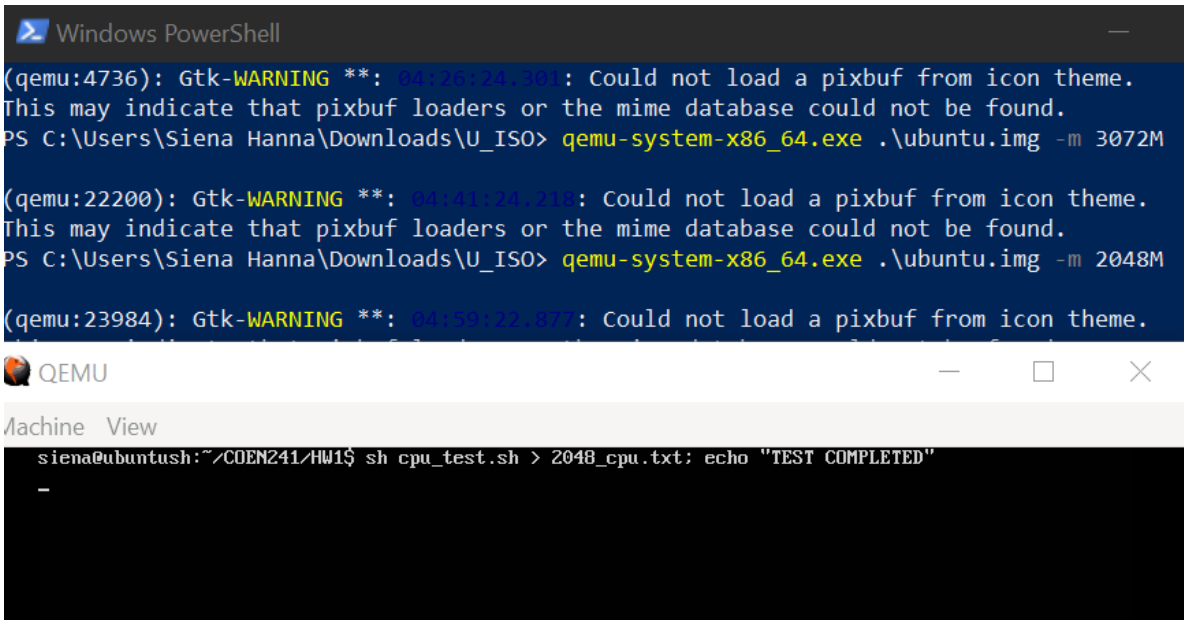


Random write (rndwr):



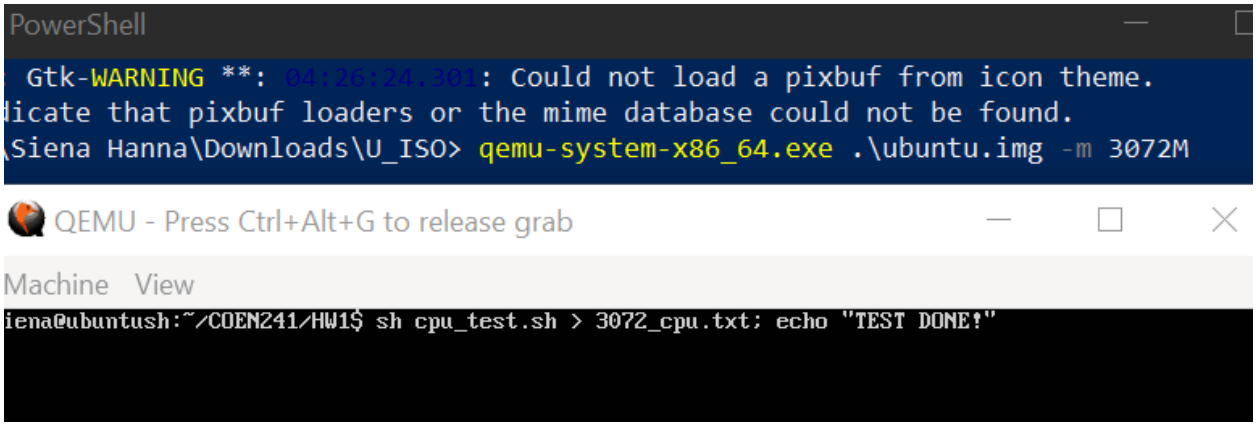
With output files:



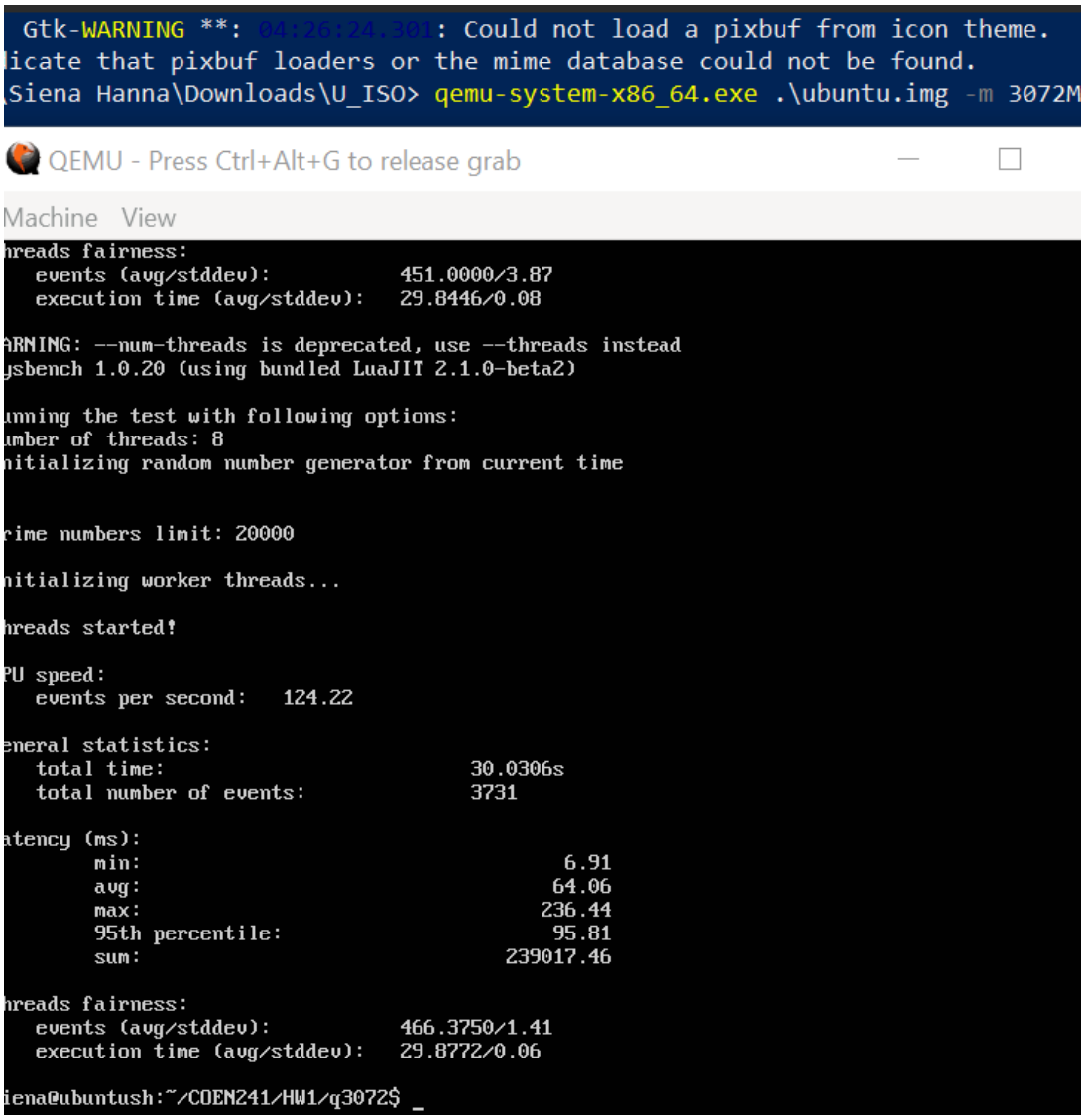


QEMU -m 3072M:

With file:

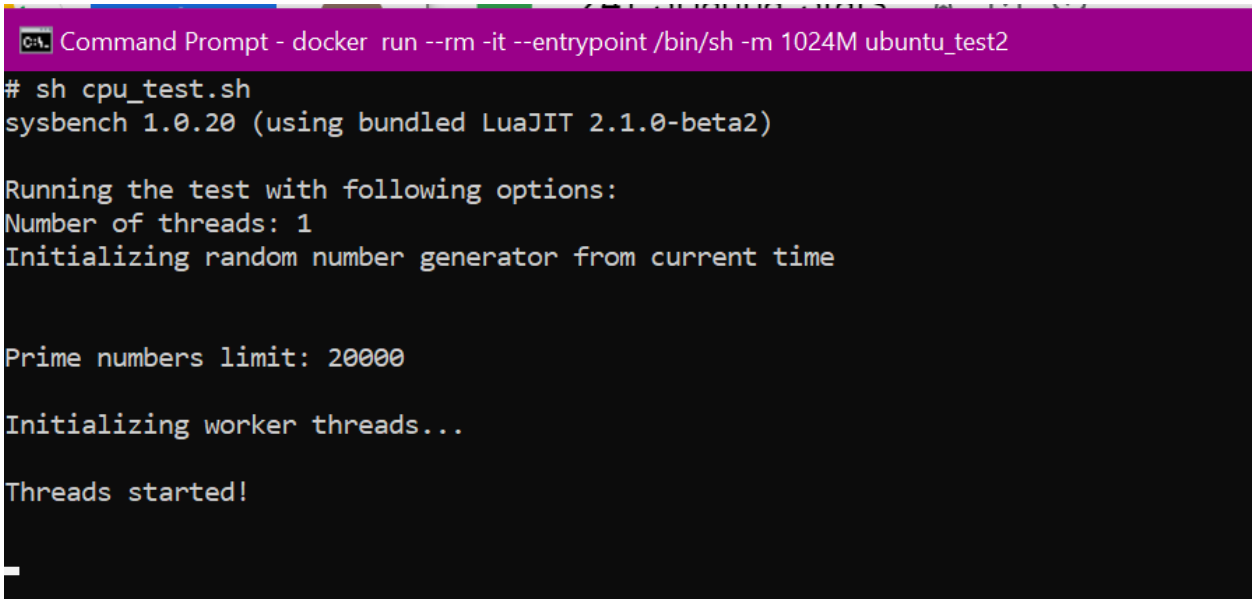


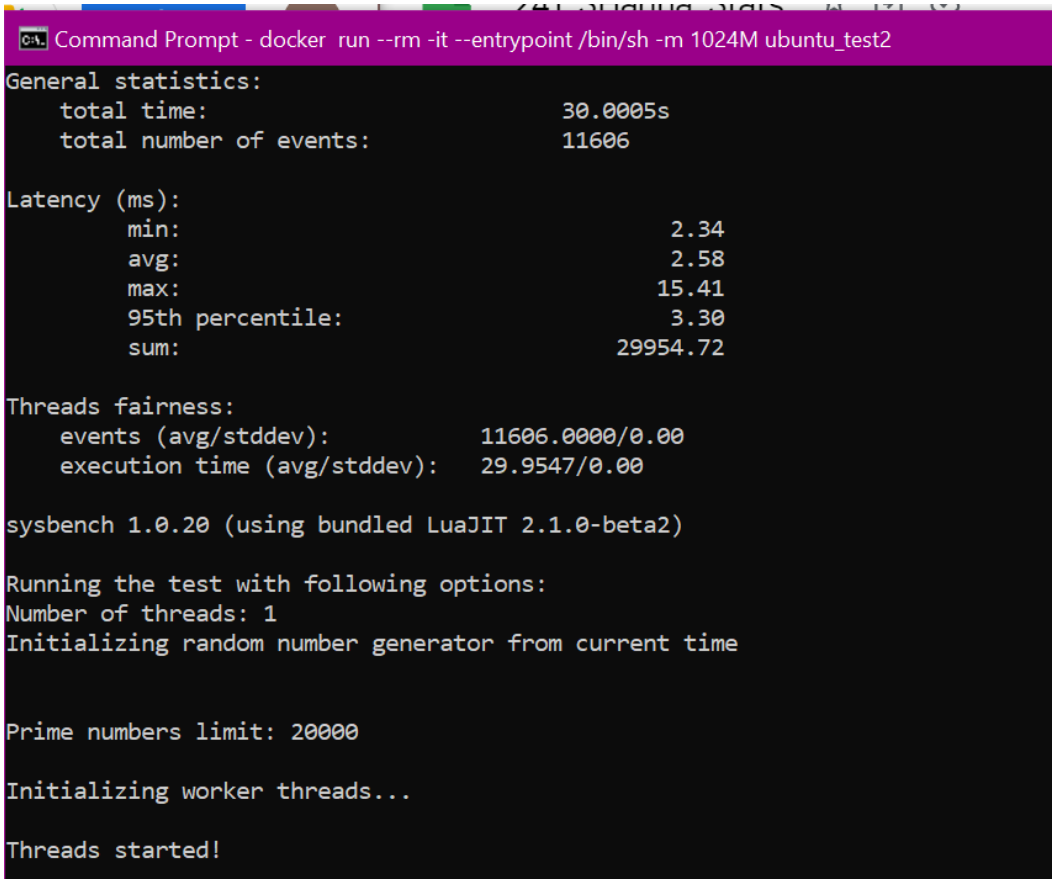
cat 3072_cpu.txt:



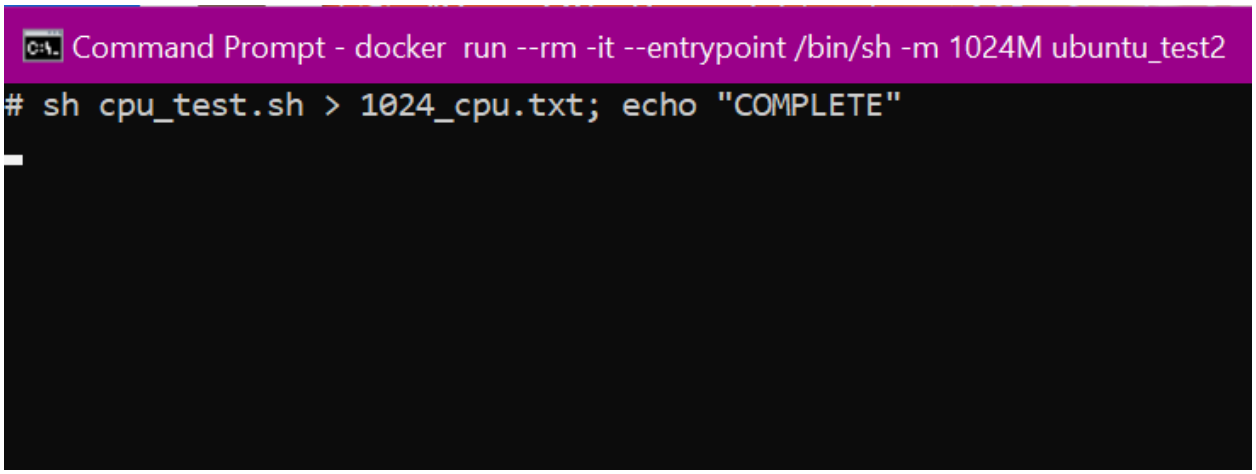
Docker Container -m 1024M:

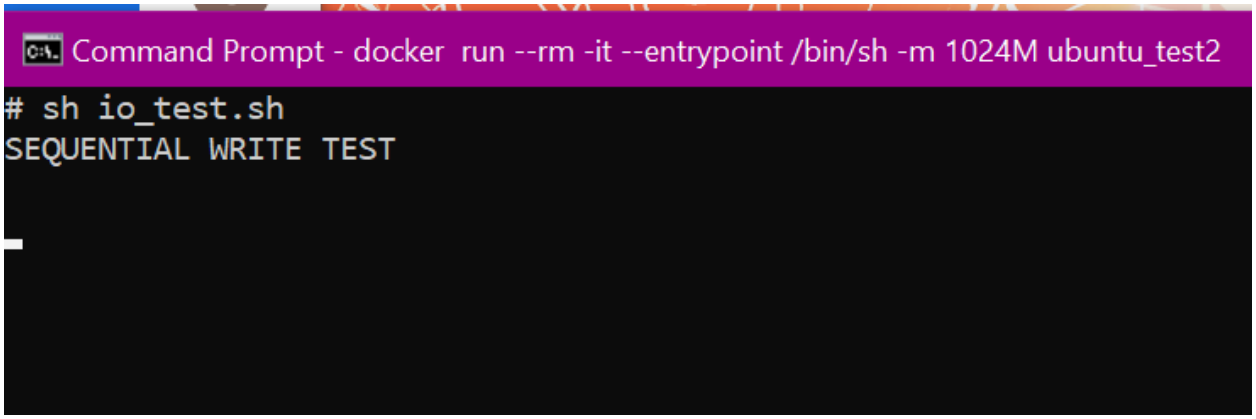
Without file **(just sh cpu_test.sh):**



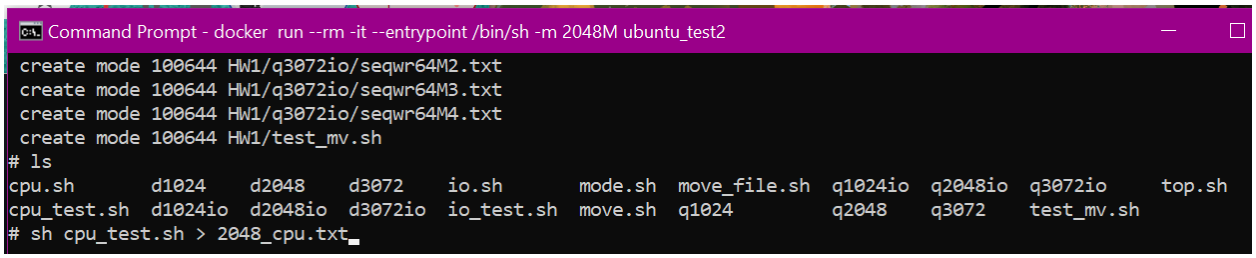


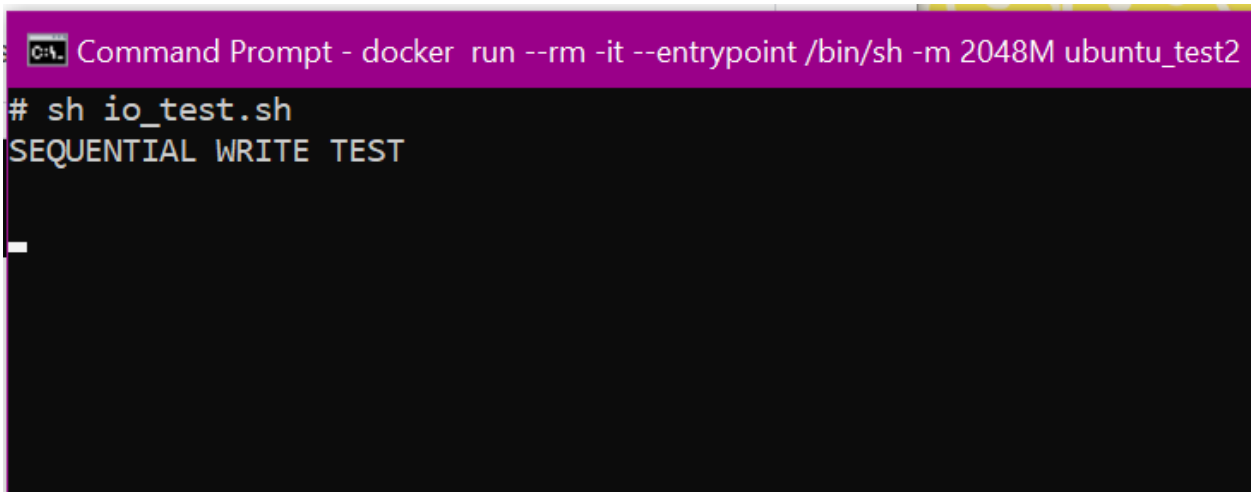
With file:



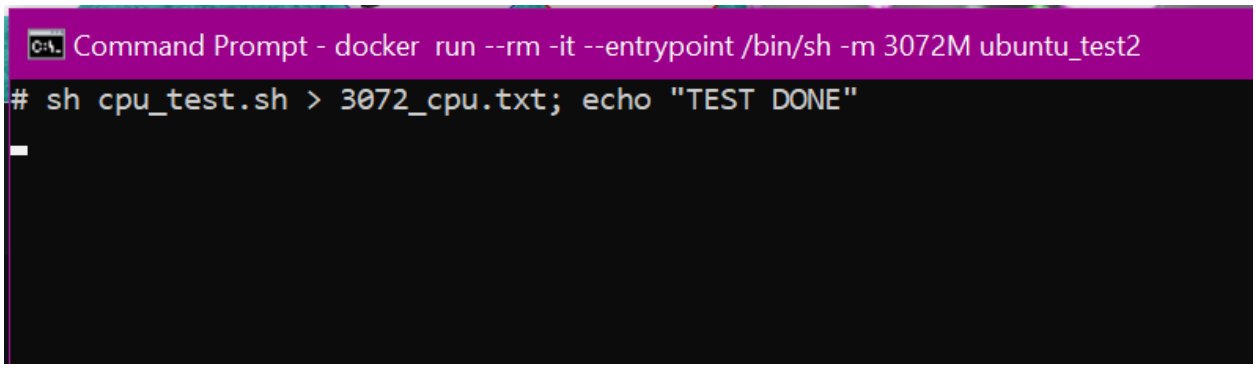


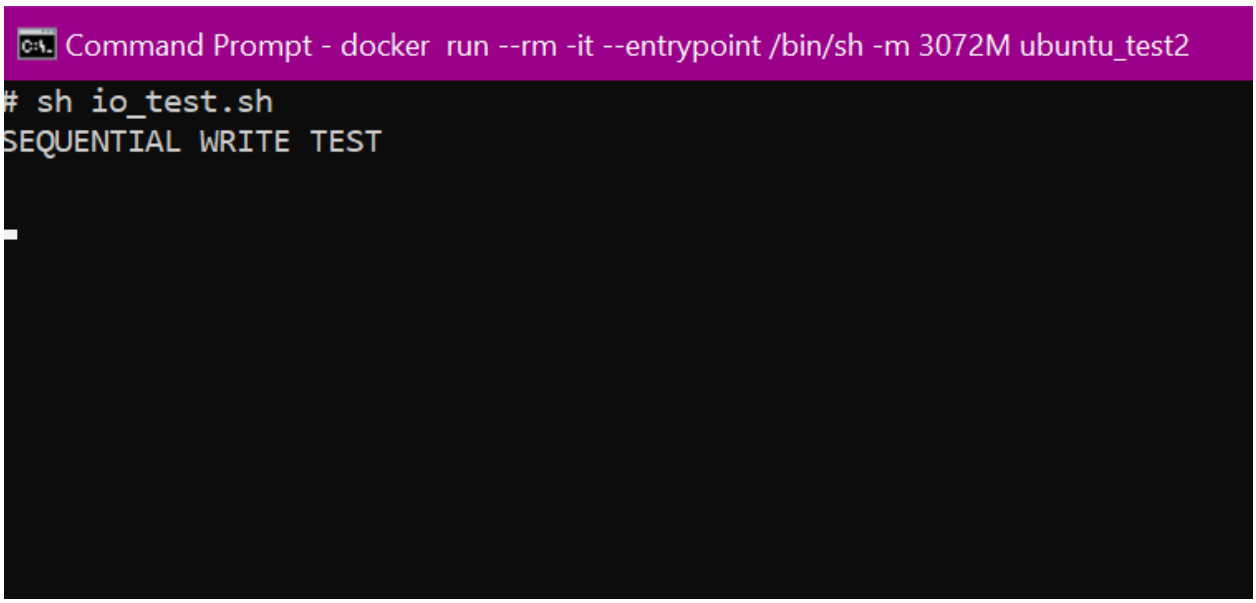
Docker Container -m 2048M with file:





Docker Container -m 3072M with file:





Experimental Results

Ultimately, I ran 10 different experiments (each repeated 5 times) on each setup.

- CPU: sysbench cpu with 1 and 8 threads, time=30, max-primes=20000—intending to report **CPU speed** and **user and system CPU utilization**.
- I/O: sysbench fileio with 1Gb and 64Mb file sizes, each done with 4 modes (Random Read (rndrd), Random Write (rndrw), Sequential Read (seqrd), Sequential Write (seqwr))—reporting **throughput, latency, and file size**.

The 6 experimental setups will be referred to in testing as follows:

- d1024: Docker container with -m 1024M
- d2048: Docker container with -m 2048M
- d3072: Docker container with -m 3072M
- q1024: QEMU VM with -m 1024M
- q2048: QEMU VM with -m 2048M
- q3072: QEMU VM with -m 3072M

I/O Test Results

For each I/O experiment with sysbench fileio, results are separated by the fileio mode used (rndrd, rndwr, seqrd, seqwr) and the file size (64M, 1G) and reported for each experimental setup. Data reported per setup includes the **average, max, and min throughput (MiB/s)** and **standard deviation of throughput (MiB/s)** overall for the 5 repetitions of each test, and the overall **min, max, and average latency[1] (ms)** for the 5 repetitions.

[1] Standard deviation is not reported for latency because each individual test by sysbench already provides an average of latency, and there is not sufficient information provided to determine overall standard deviation.

Please note that sync was run on the host machine between each repetition of the test, so they should be relatively independent of each other. The timing may not have been 100% perfect, but it was reasonably close.

Random Read

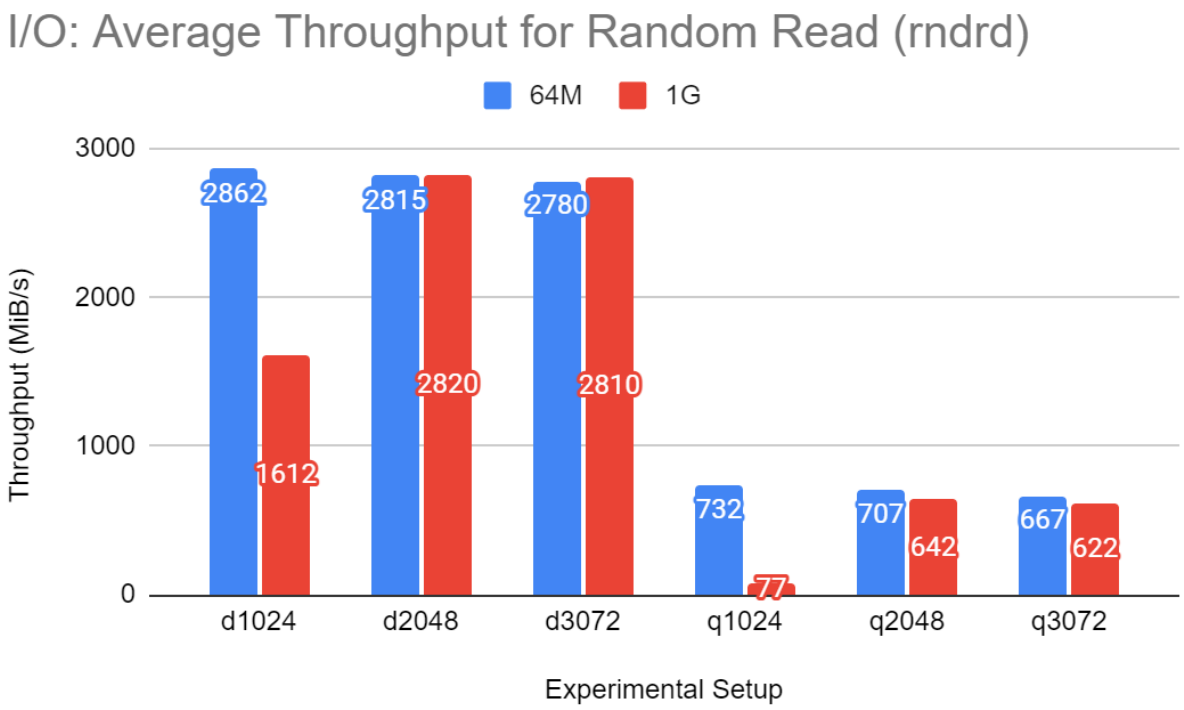
rndrd, file size=64M					
Setup	**Avg. Throughput**	**Std. Dev Throughput**	**Min Latency**	**Avg. Latency**	**Max Latency**
d1024	2862.046	57.27876596	0	0	10.66
d2048	2815.05	41.0269771	0	0	10.97
d3072	2779.766	63.56812511	0	0	17.39
q1024	732.258	15.65350344	0.01	0.02	13.32
q2048	707.162	28.39617351	0.01	0.02	9.8
q3072	667.282	36.38426542	0.01	0.02	22.99

In the above table, results are shown for file size of 64MB. Particularly notable is the very low latency for the Docker containers, with both minimum and average latencies that are too low to be measured in ms and therefore reported as 0s. The highest overall throughput was d1024, and the lowest was q3076. All the containers performed better than the VMs.

Interestingly, the amounts of RAM allocated do not necessarily appear to be positively correlated with increased throughput; in general, the averages follow the opposite pattern. However, using an interval of 1 standard deviation, most of the results for each experimental setup would overlap with each other, so it does not necessarily mean for sure that more RAM means worse performance.

rndrd, file size=1G					
Setup	**Avg. Throughput**	**Std. Dev Throughput**	**Min Latency**	**Avg. Latency**	**Max Latency**
d1024	1611.972	45.75915559	0	0.01	30.19
d2048	2820.436	43.38796469	0	0	16.85
d3072	2810.424	63.56812511	0	0	17.86
q1024	76.7	1.730621276	0.01	0.194	32.66
q2048	642.082	4.609172377	0.01	0.02	15.36
q3072	622.384	43.20277572	0.01	0.02	24.12

For random read with 1GB file size, there are similar results in terms of containers vs VMs —the containers are faster. However, this file size shows a large discrepancy between the container/VM assigned the least amount of RAM and their counterparts with more. Unlike the test with 64MB, the difference is large enough to be far out of standard deviation range of its fellows. Furthermore, the average latency and max latency of the 1024-MB setups are also increased. This is likely because the file size provided is the same as the amount of RAM.



This graph shows the average throughputs for both tables in this section. Interestingly, the throughput is barely affected or only slightly affected (within 1 std deviation) between 64Mb and 1G, with a notable exception for the 1024MB RAM container and VM. Overall, it seems that file size does not greatly affect throughput for random reads unless the file size is at or greater than the amount of RAM allocated. Furthermore, the throughput for containers/VMs tends to be slightly smaller for more RAM, but it may not be a statistically significant trend.

Random Write

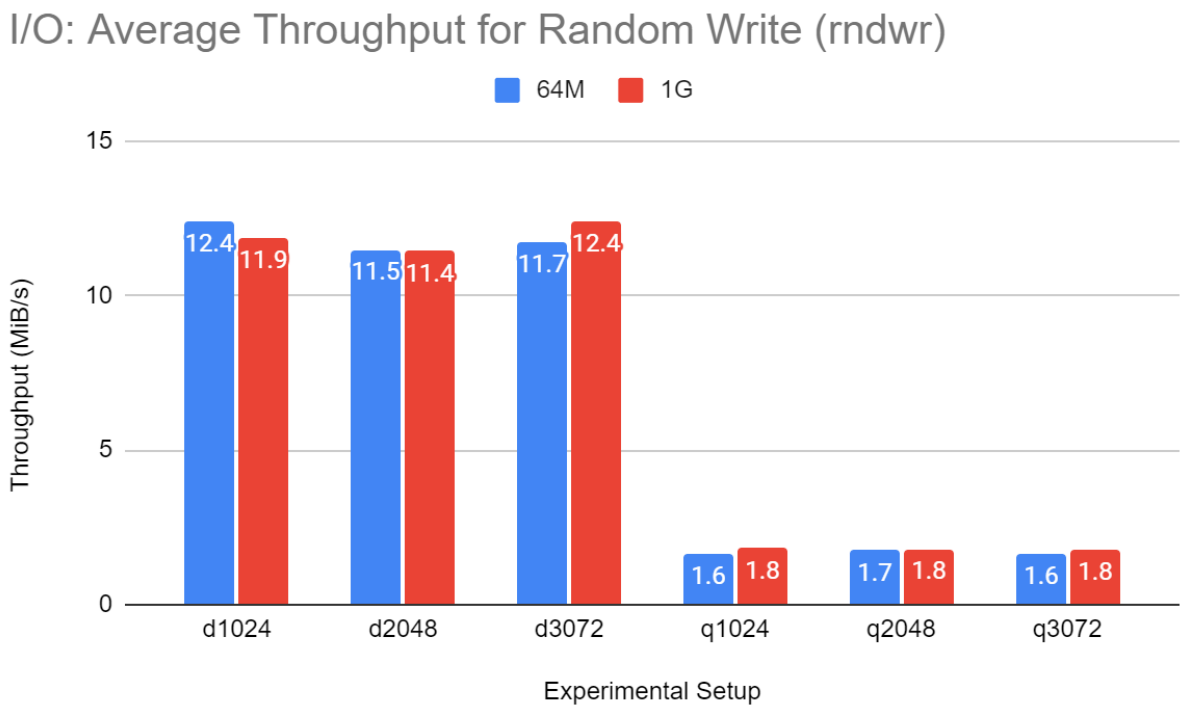
rndwr, file size=64M					
Setup	**Avg. Throughput**	**Std. Dev Throughput**	**Min Latency**	**Avg. Latency**	**Max Latency**
d1024	12.38	0.7888599369	0	0.546	233.2
d2048	11.488	0.4759411728	0	0.59	351.25
d3072	11.718	0.7984798056	0	0.578	292.99
q1024	1.622	0.06300793601	0	4.136	492.97
q2048	1.746	0.1366747965	0.03	3.862	177.36
q3072	1.64	0.1181101181	0.03	4.092	339.92

The table above shows the statistics for random write with file size of 64MB. Overall, the first and most notable thing is that writing has much lower throughput than reading. One reason for this was that I did not alter the default options for sysbench fileio with fsync, and there were also many fsync operations alongside writes. Furthermore, the containers again did much better than the VMs. Furthermore, the average latency was increased compared to random reads. For containers, it was over .55 ms, and for VMs, it was around 4 ms, as opposed to 0 or .02 from random read. Max latency was also dramatically increased from random read.

Furthermore, there is not necessarily a notable trend with increasing RAM for random writes, while increasing random reads seems to slightly decrease throughput (but not in a statistically significant way). The best throughput was at d1024 for containers and q2048 for VMs while the worst were at d2048 and q1024.

rndwr, file size=1G					
Setup	**Avg. Throughput**	**Std. Dev Throughput**	**Min Latency**	**Avg. Latency**	**Max Latency**
d1024	11.87	0.845842775	0	0.572	80.89
d2048	11.428	0.9604269884	0	0.594	73.52
d3072	12.376	0.4714657994	0	0.546	90.45
q1024	1.814	0.0937016542	0.04	3.686	185.17
q2048	1.8	0.1756416807	0.04	3.732	374.55
q3072	1.786	**0.1165332571**	0.04	3.716	940.78

The table above shows the results for random write with file size of 1GB. For the file size of 1GB, there was no specific with the 1024MB RAM, unlike random reads. The min latency slightly increased for the VMs compared to the previous file size, but otherwise the throughput and latency results were usually around the same area. However, the overall max latency for container setups decreased noticeably for containers in particular, while the max latency for VMs was particularly high for q3072.



The chart above shows the differences in average throughput for each setup at 64MB and 1GB. Overall, for VMs, the larger file size increased write throughput very slightly but potentially significantly given the low standard deviations for those tests. For containers, there is not as notable a pattern. The throughput for random writes is much decreased compared to random reads. For example, the minimum of the average container throughputs for random reads

was 1611 MiB/s (for the file size=RAM issue), and the minimum average container throughput for writes is 11.3 MiB/s, a change of magnitude over 10^2.

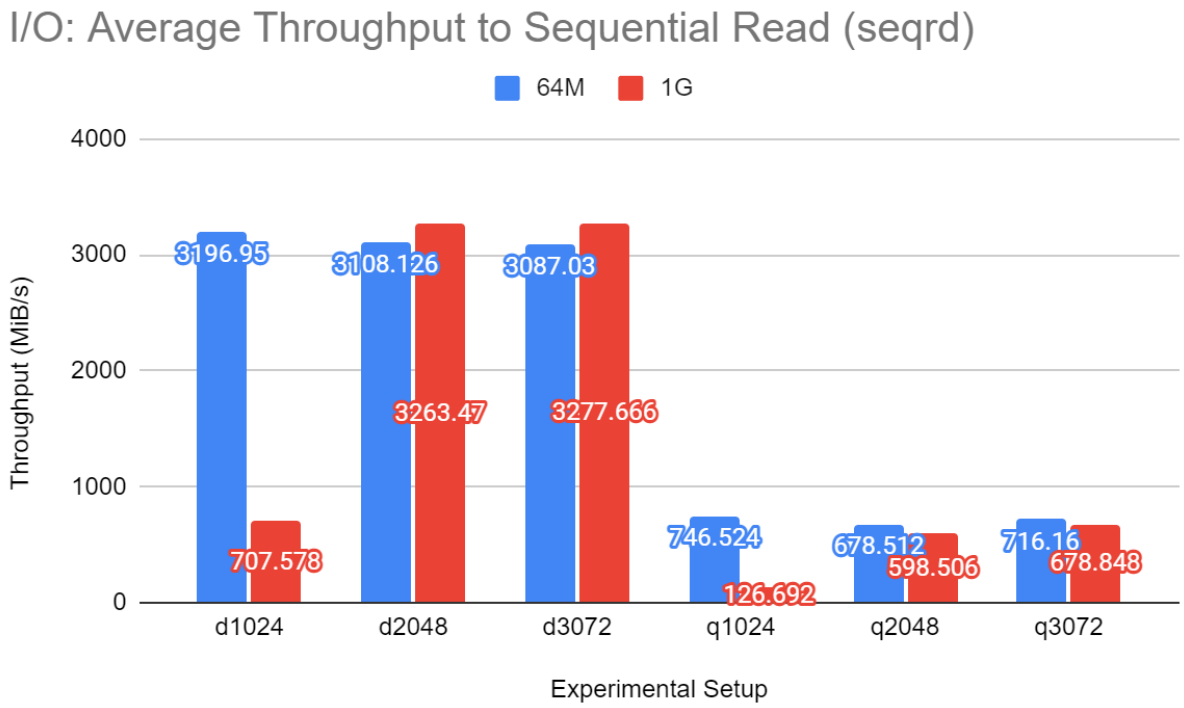
Sequential Read

seqrd, file size=64M					
Experiment	**Avg. Throughput**	**Std. Dev Throughput**	**Min Latency**	**Avg. Latency**	**Max Latency**
d1024	3196.95	46.88460088	0	1.8	9.92
d2048	3108.126	43.5983432	0	0	11.34
d3072	3087.03	41.92820352	0	0	13.19
q1024	746.524	29.92400675	0.01	0.02	14.19
q2048	678.512	133.7938015	0.01	0.022	33.94
q3072	716.16	37.77017011	0.01	0.02	16.87

The table above shows the results for sequential read with a file size of 64MB. Again, compared to containers, the VMs are much slower. Interestingly, q2048 has a very high standard deviation compared to the other experimental setups. I do not believe there could have been a particular caching issue or something similar that would only have affected the QEMU VM for RAM=2048MB *specifically*, because sync should have been run between every single test, but it is quite odd. Something that is also notable is that, compared to random read, the average throughput for containers tends to be better by around 300 MiB/s, but for the VMs, the results are more similar to random read. I would generally expect sequential read to have better throughput than random read all across the board, but that does not necessarily appear to be the case.

seqrd, file size=1G					
Experiment	**Avg. Throughput**	**Std. Dev Throughput**	**Min Latency**	**Avg. Latency**	**Max Latency**
d1024	707.578	41.97659133	0	0.02	29.16
d2048	3263.47	47.29580795	0	0	17.31
d3072	3277.666	42.61386488	0	0	19.33
q1024	126.692	7.428207725	0.01	0.114	32.72
q2048	598.506	155.1132144	0.01	0.026	18.96
q3072	678.848	55.90902315	0.01	0.02	25.59

For the 1GB file size, again there is the issue with d1024 and q1024 having much lower throughputs and higher max/average latencies than the other containers/VMs (respectively). Overall, making the file size the same as RAM for reading causes performance to slow down since the disk needs to be accessed more often. Similarly to the size of 64MB, sequential read does not necessarily perform better than random read for all cases.



In the chart above, the average throughputs for different setups and file sizes for sequential read are shown. Similar to random read, file size of 1GB causes a very notable performance decrease compared to the throughput for the smaller file size. For VMs, the 64MB file size throughputs were all faster than the 1GB throughputs, but that was only true of containers in the RAM=1024 case, and containers were otherwise faster with the 1GB file size than 64MB.

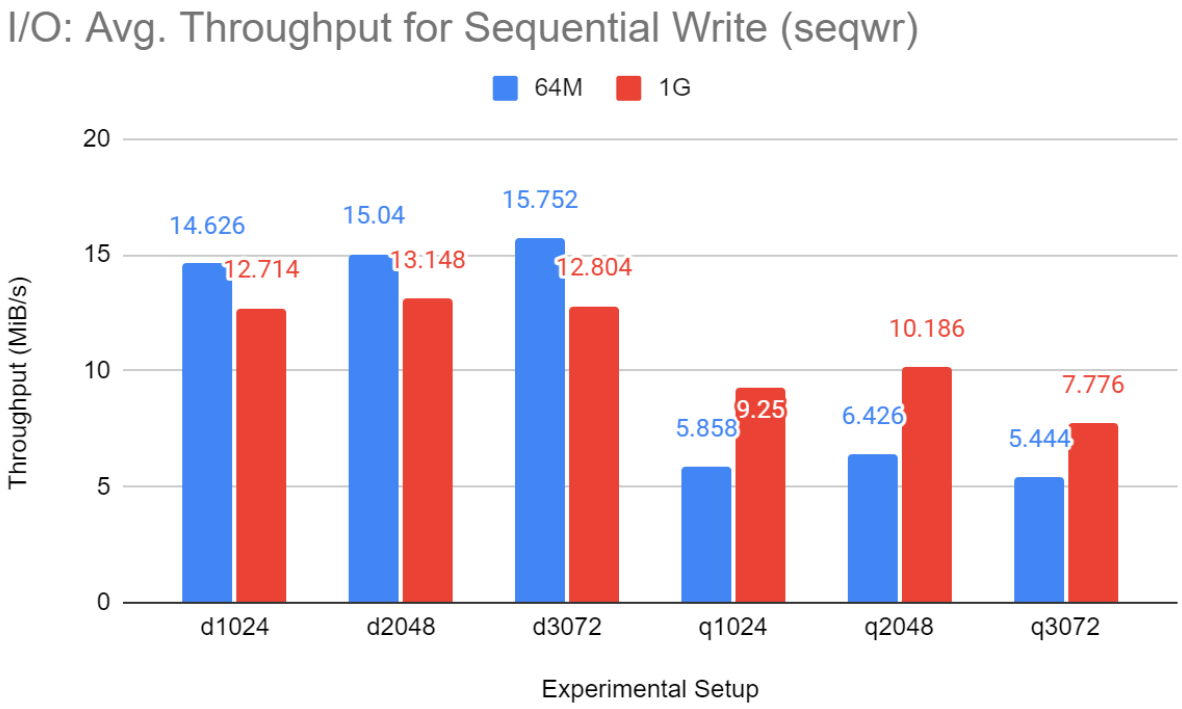
Sequential Write

seqwr, file size=64M					
Experiment	**Avg. Throughput**	**Std. Dev Throughput**	**Min Latency**	**Avg. Latency**	**Max Latency**
d1024	14.626	2.448107432	0	0.472	137.22
d2048	15.04	2.540364147	0	0.458	264.15
d3072	15.752	1.267308171	0	0.432	540.95
q1024	5.858	0.5869156669	0.03	1.148	424.19
q2048	6.426	0.1771440092	0.03	1.042	156.01
q3072	5.444	0.5267637041	0.03	1.234	174.84

The table above shows throughput and latency for sequential write with file size of 64MB. As expected, sequential write is faster for all experimental setups compared to random write (which was not always true for reads). As usual, container min latency is very low, but the average and max latencies have increased for all setups compared to the sequential read mode.

seqwr, file size=1G					
Experiment	**Avg. Throughput**	**Std. Dev Throughput**	**Min Latency**	**Avg. Latency**	**Max Latency**
d1024	12.714	2.16900438	0.01	0.546	463.06
d2048	13.148	1.28336277	0.01	0.516	256.98
d3072	12.804	1.629472307	0.01	0.532	342.66
q1024	9.25	0.9330862768	0.11	0.724	317.49
q2048	10.186	0.9409463322	0.1	0.658	487.31
q3072	7.776	1.767930994	0.11	0.912	2870.74

For a file size of 1GB, throughput for the containers decreased compared to the file size of 64MB, but throughput for the VMs increased. Similarly, average latency for the containers increased slightly while the average latency for the VMs decreased.The minimum latencies for containers was not 0 but 0.01, which is very small but it was uncommon for containers to have minimum latencies that large in the rest of the experiments.



Overall for the sequential write throughput, all containers performed better for the lower file size, and all VMs performed better for the larger file size.

Overall Conclusions

For every test, the Docker throughput was *always* better than the results for QEMU. Furthermore, for Docker in particular, the min and average values for latency are often 0, as they are reported by sysbench in ms to 2 decimal points and the container latency was often too small. This is as expected since Docker containers are actually running with the host's kernel, and the VM is slower because it has its own OS kernel and communicates with the host via a hypervisor. In hindsight, it may have been more interesting to install the hardware accelerator for Windows and test that instead of using different RAM, because that may have enabled the QEMU VM to improve its performance to be closer to containers.

CPU Test Results

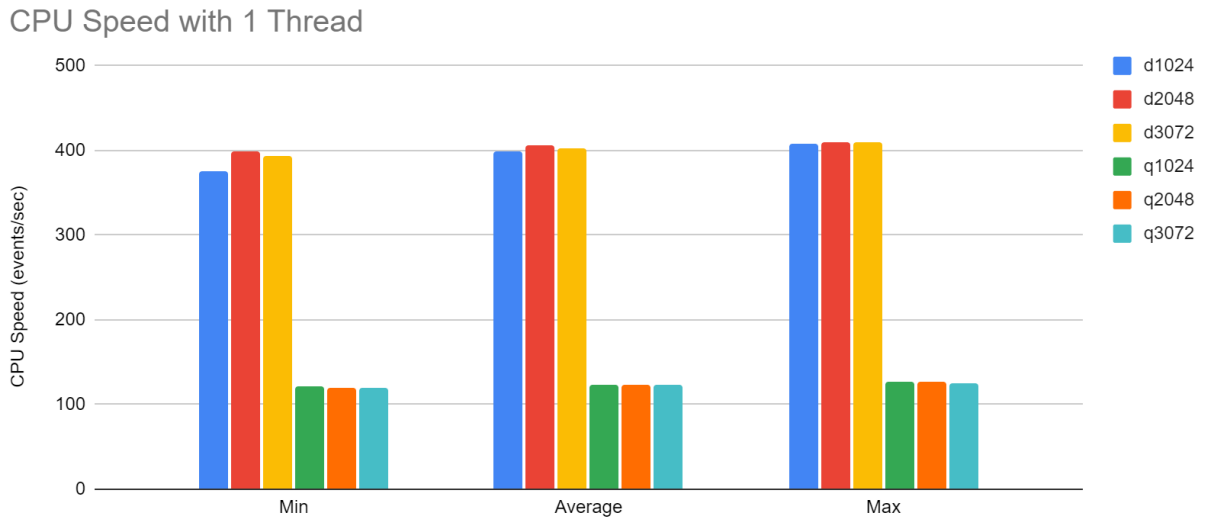
For the CPU tests, results are separated by the number of threads used (1 or 8), since the amount of primes and test time is constant for each experiment.

For each experimental setup, **CPU Speed (events/sec) min, average, max, and std. dev** and **User/System CPU utilization** are reported. Please note that, for the latter, results from Docker with top are not shown because they are almost uniformly user: 0.5, system: 0.5. This may be an oddity in to how top works inside of a container (as the container does not actually have its own OS but merely a lightweight shim that runs off the actual OS).

1 Thread

CPU Speed, 1 thread				
Setup	**Min**	**Average**	**Max**	**Std. Dev**
d1024	375.53	399.14	408.33	13.5048843
d2048	399.6	405.564	410.32	5.310586597
d3072	392.81	402.314	409.29	6.340617478
q1024	120.66	124.064	126.62	2.722467631
q2048	118.97	123.984	126.6	2.937044433
q3072	119.82	122.924	125.84	2.396670607

As with what happened for the I/O tests, containers are much faster (about 3x in this case) than the VMs. Again, this is precisely what is expected since containers are lightweight and run directly on the host OS. Overall, the standard deviation for the CPU Speed (events/sec) was small compared to the average speed. Notably, however, d1024 shows over twice the standard deviation as all the other setups, but there is not necessarily a clear reason for this.



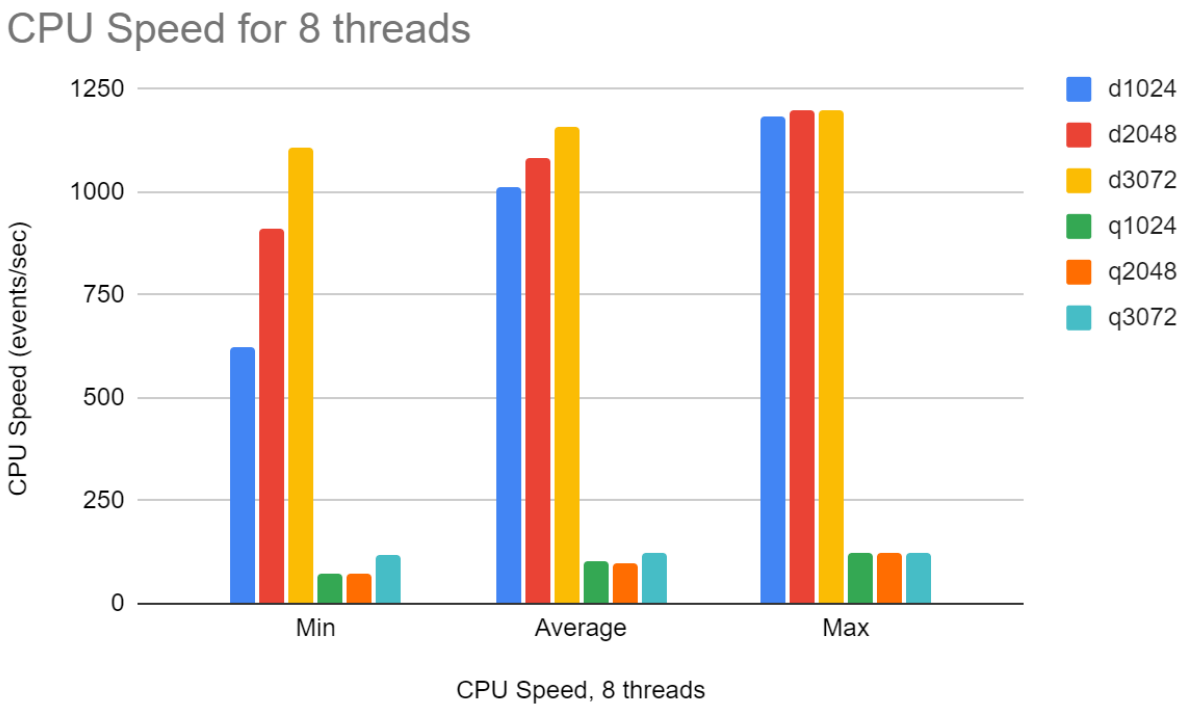
This chart shows the graphical representation of CPU Speed for each thread (min, average, and max). When scaled out, it is clear that the variations between containers and other containers and the variations between VMs and other VMs are very small compared to the large difference in CPU Speed between the containers and VMs.

8 Threads

CPU Speed, 8 threads				
Setup	**Min**	**Average**	**Max**	**Std. Dev**
d1024	623.09	1012.826	1186.08	235.5506488
d2048	910.75	1082.644	1197.15	110.7668031
d3072	1109.73	1158.468	1198.72	38.2935774
q1024	74.21	103.182	125.84	25.6375266
q2048	72.46	99.958	122.48	21.71318424
q3072	117.18	121.834	124.47	3.12698417

For 8 threads, the results are quite interesting. For containers, the speed increases even more than for 1 thread, over 2x as fast, though the magnitude of the increase in speed is not the same as the magnitude of the increase in threads (8x). However, for the VMs, the CPU Speed actually *decreases* from the results from using a single thread. Most likely, the overhead cost of

context switching between threads actually outweighs the usual advantages of using parallelism for the VM, but the container is using the host OS/hardware and is able to take advantage of parallelism.



As seen in this chart compared to the chart for 1 thread, the difference in CPU speed between VMs and containers is much more pronounced for 8 threads, since the CPU speed of containers increased and the CPU speed of VMs actually decreased.

Overall Comments on CPU Speed

Similar to the results from the I/O tests, Docker was vastly faster for all of the tests, for the same reasons. However, the difference between 1 thread and 8 threads is also notable. The average CPU speed for the container setups increases from around 400 events/sec for 1 thread to over 1000 for 8 threads. On the other hand, the average CPU Speed for the VM setups actually *decreases* when the number of threads is increased from 1 to 8. Again, this is most likely due to

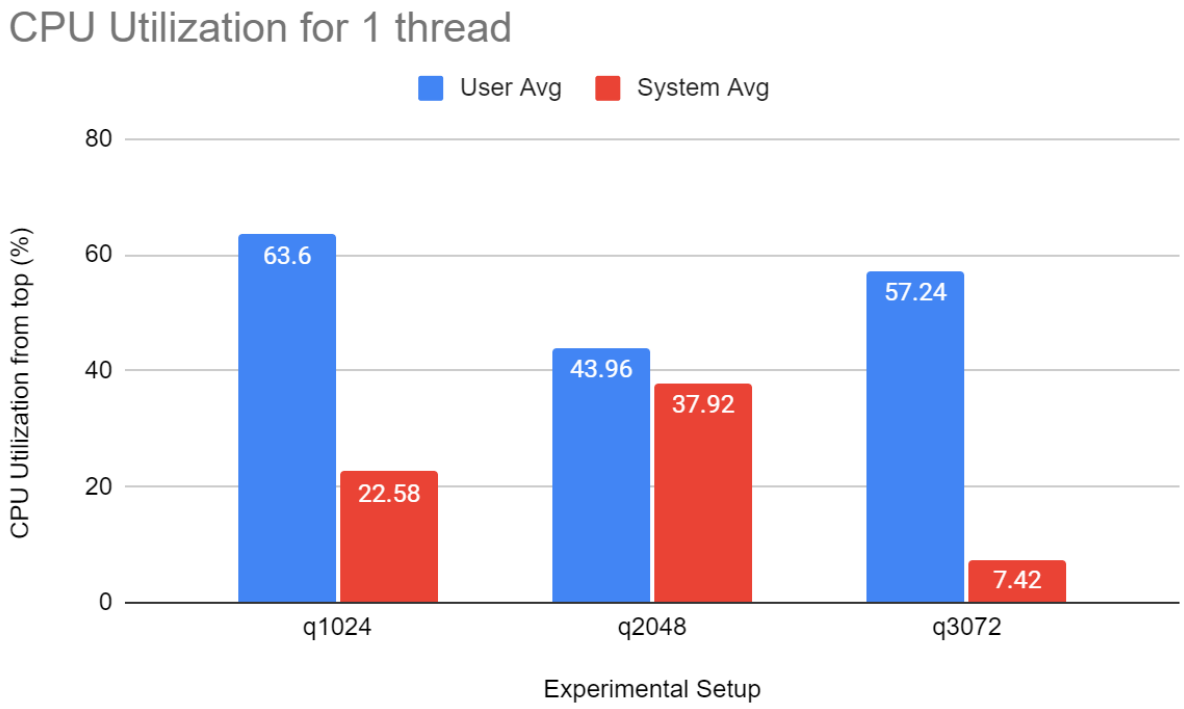
the differences in how containers vs VMs interact with the host OS/hardware. Perhaps multithreading for the VM and using virtualized resources is too complicated for these setup configurations and the cost of context switching outweighs the potential benefits of multithreading.

QEMU only: User vs. System CPU Utilization

As noted in the introduction to the results, I was only able to report results for the VMs, since containers usually had 0.5 for both user and system. Please note that the corresponding values for user and system in the same experimental setup are not necessarily in correspondence with each other, since they are the min, average, or max of *all* the user/system measurements from top (and, for example, max system may not correlate to max user).

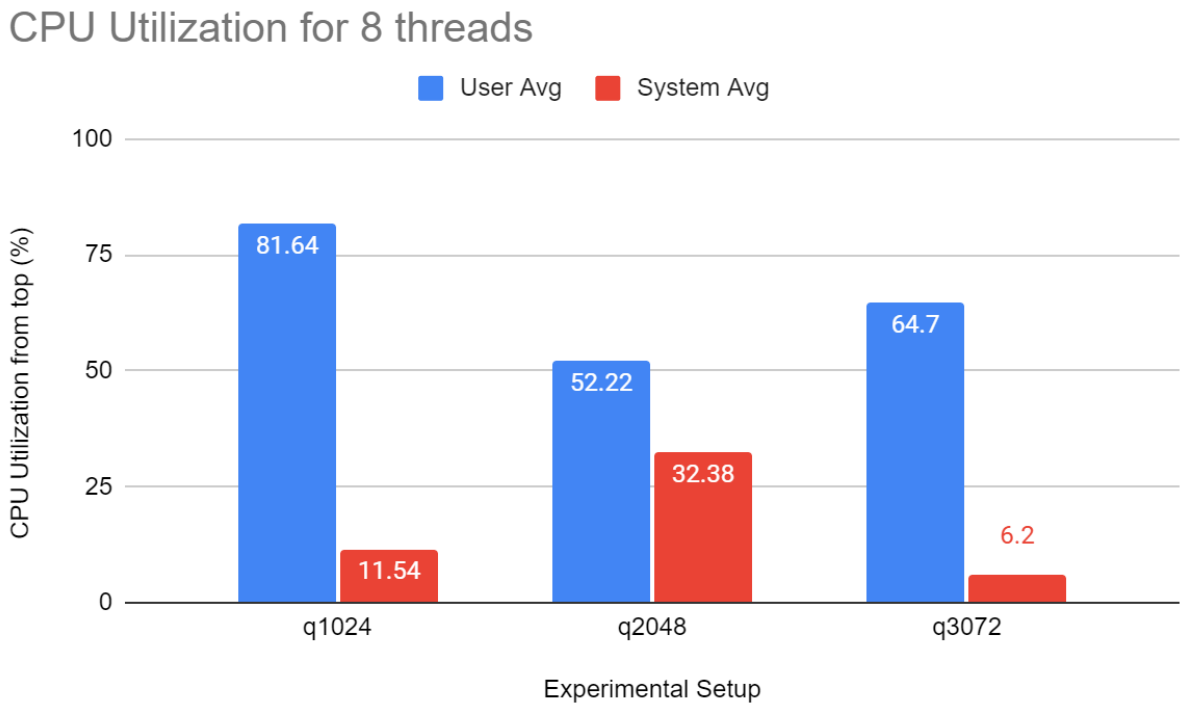
	User Min	User Avg	User Max	User Std. Dev
q1024	47	63.6	75	11.10788009
q2048	40	43.96	47.7	3.037762334
q3072	53.5	57.24	60.7	2.849210417
	System Min	**System Avg**	**System Max**	**System Std. Dev**
q1024	15.6	22.58	32.7	6.788740679
q2048	35.4	37.92	40.6	2.057182539
q3072	6.9	7.42	8	0.4438468204

Overall, user CPU utilization was always larger than the system's for every test. One slight issue is that top itself is also part of that measurement, so it is unfortunately not just based on what sysbench is doing, but sysbench was always the process using the most of the CPU overall.



This chart displays the averages for user and system CPU utilization for each VM setup. System CPU utilization is overall smaller than user CPU utilization, but q2048 having the closest to a 1:1 ratio between the two.

	User Min	User Avg	User Max	User Std. Dev
q1024	77.8	81.64	84.8	2.77001805
q2048	49.3	52.22	55	2.26207869
q3072	62.1	64.7	67.1	1.978635894
	System Min	**System Avg**	**System Max**	**System Std. Dev**
q1024	9.6	11.54	13.9	1.703819239
q2048	30.5	32.38	34.4	1.535252422
q3072	5.8	6.2	6.6	0.316227766



Comparing the average CPU utilization for 8 threads to the chart for 1 thread reveals that, comparing the corresponding VMs, the user-system ratio is increased overall. I would expect there to potentially be more system CPU utilization for multithreading, but that does not appear to be the case—instead, the ratio is even more skewed in the favor of user-space. All of the user CPU utilization percentages have also increased as compared to using a single thread.

Final Discussion

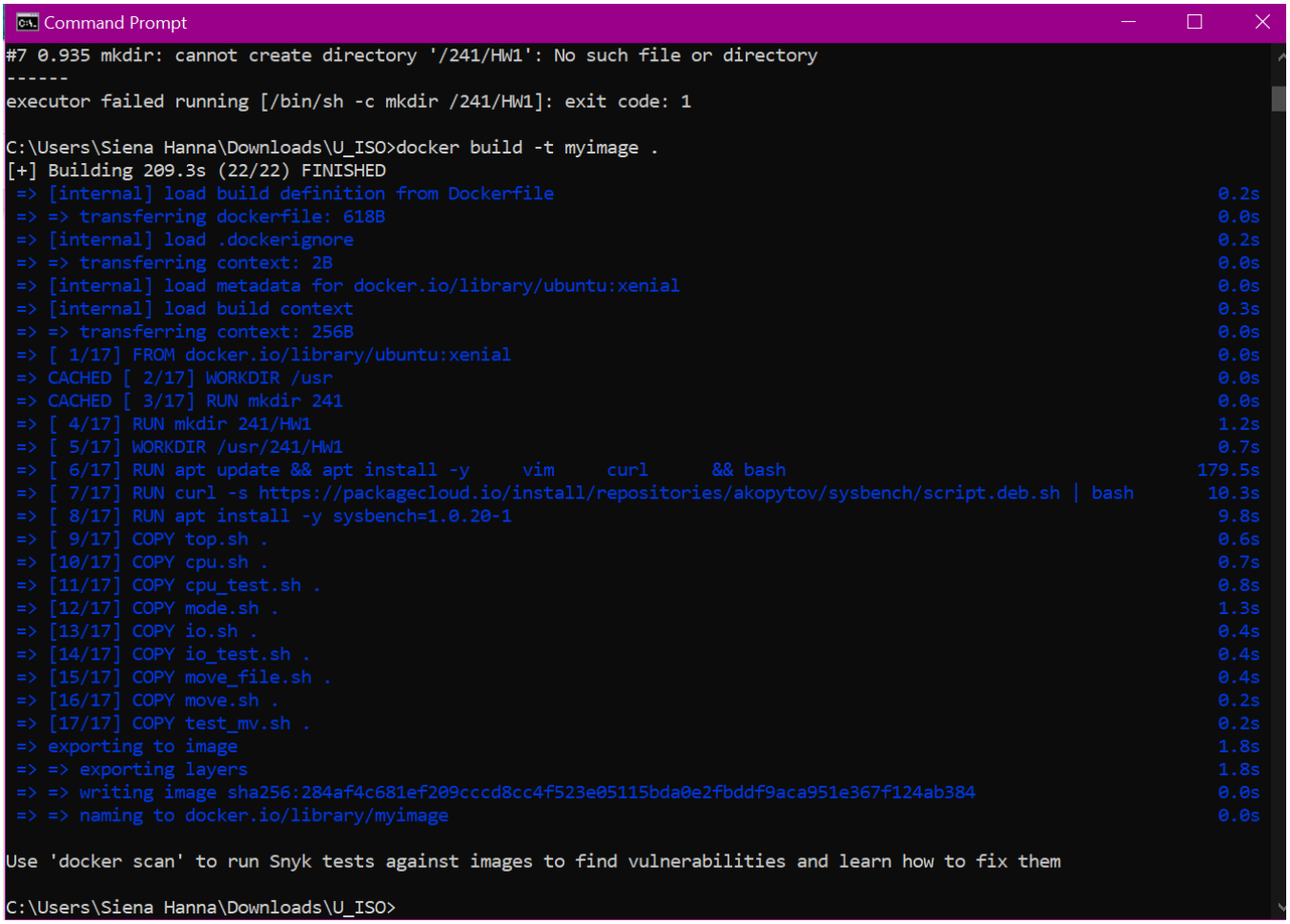
In conclusion, these experiments have shown some of the relative advantages and disadvantages of using containers vs VMs. The Docker containers were always much faster compared to the QEMU VMs, which may also be an argument for using hardware acceleration when using a QEMU VM. Though I set up my experimental scenarios to vary the amount of RAM the VMs and containers were allowed to use, this factor did not necessarily have a particular effect on the performance. I also learned how to write a Windows Batch file to

automate sync.exe, because otherwise I would have to sit in front of the computer for the entire test period waiting for 40ish second intervals to clear the cache.

Extra: Dockerfile

I also made a dockerfile to install all the required package and sysbench version and copy the required shell scripts for testing. Since it copies them from the local directory, docker build should be done in the proper directory in a local version of the GitHub repository for HW1. I used the command "**docker build -t myimage .**" in my local HW1 directory (which contains the shell scripts and Dockerfile) and was able to successfully build it after some debugging. I ran it using the same run command I used for running the containers for testing.

This screenshot shows the successful build of myimage.



This shows that the output of “docker images” now includes myimage and newimage (both created using docker build with slightly different versions of my dockerfile).

```
C:\Users\Siena Hanna\Downloads\U_ISO>docker images
REPOSITORY                      TAG       IMAGE ID       CREATED              SIZE
myimage                         latest    284af4c681ef   About a minute ago   242MB
newimage                        latest    05e42d6a1e4e   17 minutes ago       242MB
ubuntu_test2                    latest    4ebaab879929   2 days ago           1.18GB
ubuntu_test1                    latest    becfec6eeda8   2 days ago           1.18GB
ubuntu1604_git                  latest    6536f93bb200   4 days ago           1.18GB
ubuntu1604_ionew                latest    002eb5569523   4 days ago           242MB
ubuntu1604_updated_sysbench     latest    9ac6fe2aba19   4 days ago           242MB
ubuntu1604_with_vi_and_shell    latest    d4e1edeb4a1e   4 days ago           228MB
ubuntu1604                      latest    daeef3af709e   7 days ago           174MB
alpine/git                      latest    0deb7380d708   15 months ago        27.4MB
ubuntu                          xenial    b6f507652425   17 months ago        135MB
```

The below image shows the image history for myimage (aka the Dockerfile lines in reverse order).

```
C:\Users\Siena Hanna\Downloads\U_ISO>docker history myimage
IMAGE          CREATED              CREATED BY                                      SIZE      COMMENT
284af4c681ef   About a minute ago   COPY test_mv.sh . # buildkit                    488B      buildkit.dockerfile.v0
<missing>      About a minute ago   COPY move.sh . # buildkit                       488B      buildkit.dockerfile.v0
<missing>      About a minute ago   COPY move_file.sh . # buildkit                  488B      buildkit.dockerfile.v0
<missing>      About a minute ago   COPY io_test.sh . # buildkit                    488B      buildkit.dockerfile.v0
<missing>      About a minute ago   COPY io.sh . # buildkit                         488B      buildkit.dockerfile.v0
<missing>      About a minute ago   COPY mode.sh . # buildkit                       488B      buildkit.dockerfile.v0
<missing>      About a minute ago   COPY cpu_test.sh . # buildkit                   488B      buildkit.dockerfile.v0
<missing>      About a minute ago   COPY cpu.sh . # buildkit                        488B      buildkit.dockerfile.v0
<missing>      About a minute ago   COPY top.sh . # buildkit                        488B      buildkit.dockerfile.v0
<missing>      About a minute ago   RUN /bin/sh -c apt install -y sysbench=1.0.2…   6.23MB    buildkit.dockerfile.v0
<missing>      About a minute ago   RUN /bin/sh -c curl -s https://packagecloud.…   1.65MB    buildkit.dockerfile.v0
<missing>      About a minute ago   RUN /bin/sh -c apt update && apt install -y …   99.2MB    buildkit.dockerfile.v0
<missing>      4 minutes ago        WORKDIR /usr/241/HW1                            0B        buildkit.dockerfile.v0
<missing>      4 minutes ago        ENV workdirectory=/usr/241/HW1                  0B        buildkit.dockerfile.v0
<missing>      4 minutes ago        RUN /bin/sh -c mkdir 241/HW1 # buildkit         0B        buildkit.dockerfile.v0
<missing>      29 minutes ago       RUN /bin/sh -c mkdir 241 # buildkit             0B        buildkit.dockerfile.v0
<missing>      32 minutes ago       WORKDIR /usr                                    0B        buildkit.dockerfile.v0
<missing>      32 minutes ago       LABEL readme=This is the dockerfile to set u…   0B        buildkit.dockerfile.v0
<missing>      17 months ago        /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>      17 months ago        /bin/sh -c mkdir -p /run/systemd && echo 'do…   7B
<missing>      17 months ago        /bin/sh -c rm -rf /var/lib/apt/lists/*          0B
<missing>      17 months ago        /bin/sh -c set -xe   && echo '#!/bin/sh' > /…   745B
<missing>      17 months ago        /bin/sh -c #(nop) ADD file:11b425d4c08e81a3e…   135MB
```

Finally, this shows that I can successfully create a container based on myimage and it automatically goes to the /usr/241/HW1 directory (now containing the shell files).

```
C:\Users\Siena Hanna\Downloads\U_ISO>docker run -it --entrypoint /bin/bash myimage
root@30652e1eddcb:/usr/241/HW1# ls
cpu.sh  cpu_test.sh  io.sh  io_test.sh  mode.sh  move.sh  move_file.sh  test_mv.sh  top.sh
root@30652e1eddcb:/usr/241/HW1#
```