

Tutorial 3 - Docker Tutorial

Disclaimer: Subject to updates as corrections are found

Version 0.10

The purpose of tutorial #3 is to provide an introduction to Docker, cgroups, and resource isolation with containers. This tutorial should be completed using a Ubuntu system. Any of the following should be sufficient: a Ubuntu system (e.g. laptop), a Ubuntu Virtual Box, or an EC2 instance with Ubuntu. The system must have at least 2 CPU cores, and access to at least 4 GB of memory.

For the tutorial, answer the questions as best as possible based on the observations of performing the tests/activities as described. Submit answers as a PDF file in Canvas. Use Google Docs, or Microsoft Word to create a PDF file.

Task 1 - Working with Docker, creating a Dockerfile

To start, log into your Ubuntu machine. If this is an EC2 instance, a multi-core VM such as a c4.large/m4.large or better is recommended. EC2 instances should be created as spot instances, unless wanting to “pause” the instance, the an on-demand instance is required. (minimum of 10 cents/hour)

Install Docker on Ubuntu

Highlight the commands, and copy-and-paste to the VM:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
$(lsb_release -cs) stable"

# refresh sources
sudo apt-get update

# install packages
apt-cache policy docker-ce

sudo apt-get install -y docker-ce

#verify that docker is running
sudo systemctl status docker
```

The “Docker Application Container Engine” should show as running.

When working with Docker directly on your local VM, we will preface docker commands with “sudo”, so the commands run as the superuser.

Create a docker image for testing

The “Docker Hub” is a public repository of docker images. Many public images are provided which include installations of many different software packages. The “sudo docker search” command enables searching the repository to look for images.

Let’s start by downloading the “ubuntu” docker container image:
Note that docker commands are prefaced as “sudo”.
They must be run as superuser.

```
sudo docker pull ubuntu
```

Verify that the image was downloaded by viewing local images:

```
sudo docker images -a
```

Next, make a local directory to store files which describe a new docker image.

```
mkdir docker_test  
cd docker_test
```

Using a text editor such as vi, vim, pico, or nano, edit the file “Dockerfile” to describe a new Docker image based on ubuntu:

```
nano Dockerfile
```

```
# Test Dockerfile contents:  
FROM ubuntu  
RUN apt-get update  
RUN apt-get install -y stress-ng  
RUN apt-get install -y sysbench  
COPY entrypoint_test.sh /  
ENTRYPOINT ["/entrypoint_test.sh"]  
CMD ["6000"]
```

Next, create a script called “entrypoint_test.sh” under your docker_test directory as follows:

```
#!/bin/bash  
# test daemon - runs container continually as a task...  
# Exits task and container when sleep time expires.  
sleep=$1  
echo "daemon up... sleep_for=$1"  
sleep $sleep  
exit
```

You’ll need to change permissions on this file.
Give the owner execute permission:

```
chmod u+x entrypoint_test.sh
```

Next, build the docker container:
`sudo docker build -t stressng .`

Check that the docker image was build locally:
`sudo docker images`

Next launch the container as follows:
`sudo docker run -d --rm stressng`

Check that the container is up
`sudo docker ps -a`

Next, run the bash shell interactively as a second process inside this container:
Find the container-id from the docker ps command.
`sudo docker exec -it <container-ID> bash`

Next, open a second ssh terminal to your Ubuntu machine.
Navigate to the directory as follows:

```
cd /sys/fs/cgroup/cpuacct/docker
```

Under the docker directory, find the unique identifier for your container.
This matches the first several characters of the container ID as seen using docker ps -a.
Navigate to this directory:

```
cd <container-ID-long>
```

Next, watch the "cpuacct.usage" file:

```
watch -n .5 cat cpuacct.usage
```

The cpu utilization is shown in nano seconds.
Move the decimal 9 places to the left to convert to CPU seconds.

QUESTION 1. Without running any test, how much CPU time has been spent in seconds, since this container was created?

Task 2 - Using Cgroups to monitor resource utilization

Print out the initial CPU utilization value:
`cat cpuacct.usage`

Next, run the stress-ng command:
`stress-ng --cpu 2 --cpu-method fft --cpu-ops 5000`

Next, print out the updated current CPU utilization value:
`cat cpuacct.usage`

QUESTION 2. After running the test, what is the present CPU utilization value in seconds?

QUESTION 3. What is the difference in CPU time in seconds that transpired for running the test? (*subtract the two values*)

The output of stress-ng reports the runtime in seconds. This is considered "wall clock time".

What is the difference between the reported runtime and the CPU time as measured by the linux cgroup `cpuacct` ?

Before proceeding, try repeating the test, and explore various system metrics that are available under the `/sys/fs/cgroup/` directory. You may also explore running different stress-ng tests.

For help in stress-ng, see:

<http://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html>

<https://wiki.ubuntu.com/Kernel/Reference/stress-ng>

<https://www.cyberciti.biz/faq/stress-test-linux-unix-server-with-stress-ng/>

Task 3 - Using Docker to constrain resource allocation

Next, exit the ssh session.

Now, assign the `cpu-shares` of the docker container:

```
sudo docker update --cpu-shares="128" <container-id>
```

Repeat the stress test:

```
stress-ng --cpu 2 --cpu-method fft --cpu-ops 5000
```

QUESTION 4. What happens to the runtime of the test?

For question 4, based on the documentation, describe what we are seeing with respect to the runtime of stressng after assigning `cpu-shares`:

https://docs.docker.com/config/containers/resource_constraints/#cpu

Next, reset the CPU shares to the default

```
sudo docker update --cpu-shares="1024" <container-id>
```

And then assign the containers "cpus"

```
sudo docker update --cpus=".5" <container-id>
```

Now, print out the `cpuacct.usage` before the test:

```
cat cpuacct.usage
```

Now, in the second window, repeat the stress test and observe the run time:

```
stress-ng --cpu 2 --cpu-method fft --cpu-ops 5000
```

Obtain the end cpu usage, and calculate the differences:

```
cat cpuacct.usage
```

QUESTION 5. What was the CPU utilization for the test? How did it vary from our previous measurement? What could explain the behavior we are seeing ?

Next, reset the CPU allocation for the container:

```
sudo docker update --cpus="2" <container-id>
```

Task 3 - Test CPU Isolation with Docker

Now, in a second terminal window, create a second instance of the same container.

Launch the container as follows:

```
sudo docker run -d --rm stressng
```

Check that the new container is up, and check for the new ID:

```
sudo docker ps -a
```

Now, let's test CPU isolation of containization.

Assuming you're on a two-core system, first limit the CPU allocation to 1 core for each of the two containers.

Find the container IDs using the docker ps -a command.

And assign the CPU allocation for both containers:

```
sudo docker update --cpus="1" <container-id-A>
```

```
sudo docker update --cpus="1" <container-id-B>
```

Next, run a bash shell interactively on the second container:

Use the container-id from the docker ps command above.

```
sudo docker exec -it <container-ID> bash
```

In two separate terminals, for each of the containers, type the command, but DO NOT hit enter yet:

```
stress-ng --cpu 2 --cpu-method fft --cpu-ops 5000
```

First, run one container alone to measure the stand-alone performance of the command.

Next, prepare to run the command in both commands in parallel.

This requires submitting commands to both containers as close as possible in time so their execution overlaps as much as possible.

QUESTION 6. What is the performance difference when running the command standalone vs. running two instances at the same time with CPU allocation has been set to 1?

If container isolation is “perfect” for sharing the CPU, then performance should essentially be the same.

Task 4 - Test memory Isolation with Docker

Next, let’s try a memory stress test to test for how well the Docker containers provide isolation from concurrent memory operations on the host.

In one of the terminals, run the sysbench command to stress memory.

```
sysbench --test=memory --memory-block-size=1M --memory-total-size=100G --num-threads=1 run
```

At the conclusion, look for the memory throughput value. This is right below the “Total operations”, and the throughput is shown in “MiB/sec”. This represents the amount of memory that was transferred per second.

Now, stage this command to performance the memory stress test on two containers at the same time. Recall these two containers should have had their CPU’s limited using the setting: --cpus=“1”

Run the command at the same time in two containers:

```
sysbench --test=memory --memory-block-size=1M --memory-total-size=100G --num-threads=1 run
```

If memory isolation is “perfect” for sharing the memory subsystem of the host, then performance should essentially be the same.

QUESTION 7. What is the memory throughput values (MiB/sec) for both containers A and B?

QUESTION 8. What is the average memory latency (in ms) for both containers A and B?

QUESTION 9. How did the memory throughput and memory latency change when comparing the standalone (1 container) test values with the concurrent container test?

QUESTION 10. Do docker containers provide?

- (a) Better memory isolation
- (b) Better CPU isolation

QUESTION 11. Explain why this could be...

Task 5 - Cleanup

At the end of the tutorial, if using EC2, you may want to create an image of your virtual machine with Docker. If you haven't already, reimaging your server VM will allow it to be restored with minimal effort and setup in the future.

After reimaging, be sure to **TERMINATE** all EC2 instances. Failing to do so, could result in loss of AWS credits or AWS charges to a credit card.

You may also want to purge old duplicate snapshots, when you've created more than one image of an EBS-backed instance. It may not be worthwhile to keep old copies around when new images supersede them.