TCSS 462/562: (Software Engineering for) Cloud Computing
Fall 2024

School of Engineering and Technology
University of Washington – Tacoma

Tutorial 7 – Docker Tutorial

Disclaimer: Subject to updates as corrections are found
Version 0.11

Tutorial should be completed using Ubuntu 24.04. Ubuntu 20.04 is not supported.

The purpose of tutorial #7 is to provide an introduction to Docker, cgroups, and resource isolation with containers. This tutorial should be completed using a **c6i.large** Ubuntu 24.04 spot instance on Amazon EC2. This virtual machine has 2 vCPU cores and 4 GB of memory. Refer to tutorial 3 for more information regarding creating an Amazon EC2 **c6i.large**. Please terminate the instance once the tutorial is completed.

TO SUBMIT THIS 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 24.04 virtual machine. EC2 instances should be created as spot instances. If wanting to "pause" the instance, a persistent spot request can be used.

Install Docker on Ubuntu 24.04 c6i.large ec2 instance
The Instructions below are useful for installing Docker beyond the scope of this tutorial.

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

USE THE MS WORD DOCUMENT NOT THE PDF FILE TO COPY AND PASTE THE COMMANDS BELOW

When copying from the PDF file, occasionally certain characters are not interpreted correctly resulting in errors.

If you are a Mac user and experience errors below, switch to using a c6i.large spot instance for tutorial 7. Use of c6i.large is required for consistency with tutorial 7. Virtual Box VMs and MACs should not be used.

```
sudo apt update

sudo apt install apt-transport-https ca-certificates curl software-properties-common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

```
# refresh sources
sudo apt update

apt-cache policy docker-ce

# install packages
sudo apt install docker-ce

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

The "Docker Application Container Engine" should show as running.

The Docker daemon, by default, uses an IPC socket to support interprocess-communication between processes on the same Docker host. The Docker daemon, by default, always runs as the **root** user. Consequently, the Docker IPC socket is owned by the root user, and other users on the Linux system can only access this IPC socket using sudo. This means you will be required to preface all Docker commands with "sudo" on your system.

If you don't like this **default** behavior, please refer to this article to create a Docker group, and then run the Docker daemon using this group. Then add users to the Docker group to avoid requiring the use of "sudo":

https://docs.docker.com/engine/install/linux-postinstall/

This tutorial assumes the superuser root account will always be used.
All docker commands are prefaced with "sudo".

If not wanting to configure the "docker" group, you can save from typing "sudo" for each command by assuming the role of superuser in your bash shell by typing: "sudo bash".

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.

For example, you can search for old Ubuntu 20.04 images (2020), and then newer current images using:

```
sudo docker search ubuntu20
sudo docker search ubuntu
```

Trusted images will be marked as OFFICIAL. Be careful with other images. Searches can target official images:

```
sudo docker search -f=is-official=true rock
```

Let's start by downloading the official "ubuntu" docker container image:

```
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 seconds"
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. Make a note of the ‘CONTAINER ID’, the left-most column.

```
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
```

QUESTION 0. What computer are you using as a Docker Host for tutorial #7? Is this the c6i.large EC2 instance? (YES/NO).

Use of a c6i.large is required for tutorial 7. If not using a c6i.large, restart the tutorial, or contact the instructor.
Docker can be installed on any Linux VM, but for consistency use of **c6i.large** is required for the tutorial.

Next, open a second ssh terminal to your c6i.large ec2 instance with Docker.
CONTROL-SHIFT-T in the terminal, opens a new terminal tab.

In the second terminal, navigate to the directory as follows:

```
cd /sys/fs/cgroup
```

Please note, the next section of Tutorial 7 has been updated for Ubuntu 24.04. As of the release of Docker Engine 20.10 in December 2020 which is included in Ubuntu 22.04/24.04 LTS but not Ubuntu 20.04 LTS, the location for all of the files below has changed.

Using the '**sudo docker ps -a**' command, copy the short version of the 'CONTAINER ID' for the running stressng container (right-click copy).

You can try to find the cgroup data for your container by searching from **`/sys/fs/cgroup`**:

```
find . | grep <replace with your CONTAINER ID>
```

But it is probably easier, to just navigate to the directory with the '**`cd`**' command as follows:

```
cd system.slice/docker-<FULL CONTAINER ID>.scope/
```

This directory will contain CPU metrics only for this container. Watch the "cpu.stat" file:

```
watch -n .5 cat cpu.stat
```

The cpu utilization is shown in microseconds (10^-6).
Move the decimal 6 places to the **LEFT** to convert to CPU seconds.

The cpu.stat file reports three CPU metrics:

usage_usec	The total cgroup CPU utilization in microseconds (10^-6)
user_usec	The cgroup user mode CPU utilization in microseconds (10^-6)
system_usec	The cgroup system (kernel) mode CPU utilization in microseconds (10^-6)

QUESTION 1. Without running any test, how much CPU time has transpired since this container was created? Report total CPU time which is 'usage_usec' in microseconds.

Task 2 – Using Cgroups to monitor resource utilization

Press CONTROL-C to break out of the **`watch`** command.

Print out the initial CPU utilization value (*or refer to the value in the "watch" terminal*):
[>>In the *second terminal window with an ssh session to the ec2 instance*]
```
cat cpu.stat
```

Next, run the stress-ng command:

[>>In the first terminal window *where you ran "sudo docker exec -it <container-id> bash"*]
```
stress-ng --cpu 2 --cpu-method fft --cpu-ops 5000
```

When the command finished, next, print out the updated current CPU utilization value:
[>>In the second terminal window]
```
cat cpu.stat
```

QUESTION 2. After running the test, what is the value of the CPU counter now?
Report 'usage_usec'. This is microseconds.

QUESTION 3. What is the difference in CPU time in microseconds that transpired for running the test?
(subtract cputime2 – cputime1)

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

QUESTION 3B. Observe the difference between the runtime that stress-ng reported (wall clock time in seconds), and the CPU time in microseconds as reported in the linux cgroup cpu.stat file.
Convert CPU time in microseconds to seconds.
After conversion, which time is GREATER in seconds: "**stress-ng-*wall-clock-time*"** or the "**cgroup-*CPU-time*"** ?

Contemplate for a moment why either stress-ng wall-clock-time or container cgroup CPU-time is greater.

Before proceeding, try repeating the test, and explore various system metrics that are available under the **/sys/fs/cgroup/system.slice/docker-<FULL CONTAINER ID>.scope/** 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 – Persisting Docker Images to "Docker Hub" image repository

Docker images are stored in "Docker Hub". Docker Hub can be compared to "GitHub". Where "GitHub" provides a repository for tracking changes to source code for one project, "DockerHub" provides a repository for tracking

changes to a Docker container image. Just like GitHub, with DockerHub there are public and private repositories. DockerHub repositories are used to collect versions of a single image. These version can be tagged with names for quick retrieval. Free DockerHub accounts are limited to only one private repository of images, but they can have unlimited public repositories. So if wanting to maintain more the one private Docker image, it is necessary to upgrade beyond the basic DockerHub account.

To get started, you'll need to create an account on DockerHub.
Using a web browser, navigate to:

https://hub.docker.com/

Next, create an account by completing the form:

Create your account

Continue with Google

Continue with GitHub

OR

Email
wlloyd_@washington.edu

Username
wlloydwa2

Password
•••••••••••••

Send me occasional product updates and announcements.

This site is protected by reCAPTCHA and the Google Privacy Policy and Terms of Service apply.

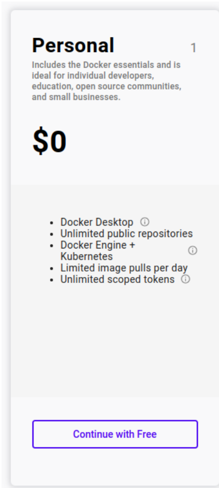
Sign up

By creating an account I agree to the Subscription Service Agreement, Privacy Policy, Data Processing Terms.

Already have an account? Sign in

Please note your account information (Docker ID, email, password) for future use.

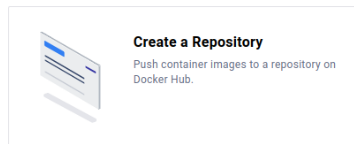
Once creating an account, log in, and select the "Personal" Docker Hub plan:



Next follow the instructions to verify your email address.
Once verifying, you will see the Docker Hub web management console.

Next using the GUI, create a new repository:

Click on the “Create Repository” button:



Give the repository a name.
Enter Name: **`tcss462-562_f24`**

Choose to make the repository **public** or **private**.
Note: the free DockerHub account is limited to 1 private repository.
Then press the **[CREATE]** button.

Now, log into your DockerHub account from the command line:

```
sudo docker login -u <USERNAME>
```

Inspect your IMAGE ID for your stressng Docker image

```
sudo docker images -a
```

Using the IMAGE ID, tag this image for adding into your DockerHub repository

```
sudo docker tag <IMAGE ID> <Docker Hub USERNAME>/tcss462-562_f24:latest
```

Now commit the image to your repository:

```
sudo docker push <Docker Hub USERNAME>/tcss462-562_f24
```

Now manually delete both the stressng image and the tagged image that you just committed to the DockerHub repository.

To remove the images, you'll need to make sure to stop/kill the container.
To kill the container, find it's ID using: sudo docker ps -a
Then kill the container using: sudo docker kill <container-id>

Now remove all traces of the stressng image from your system

```
sudo docker rmi stressng
sudo docker rmi <Docker Hub USERNAME>/tcss462-562_f24
```

Remove cached images:

```
sudo docker image prune -a -f
```

Remove the docker build cache:

```
sudo docker buildx prune -f
```

Now using the DockerHub search command, look for the tcss462-562_f24 repository

```
sudo docker search tcss462-562_f24
```

You may see other students repositories here if they create public repositories.

Go ahead and PULL your pushed docker image, put preface the command with the Linux "time" command to record how long it takes.

```
time sudo docker pull <Docker Hub USERNAME>/tcss462-562_f24
```

Now, purge this image:

```
sudo docker rmi <Docker Hub USERNAME>/tcss462-562_f24
```

Remove cached images:

```
sudo docker image prune -a -f
```

Remove the docker build cache:

```
sudo docker buildx prune -f
```

Next, rebuild your stressng container, but time how long it takes:

```
time sudo docker build -t stressng .
```

QUESTION 4. After clearing image and build caches, is it faster to pull the docker image from DockerHub or rebuild the image from scratch locally? Please list the times for pulling vs. building.

Task 4 – Using Docker to constrain resource allocation

Next, restart the Docker container:

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

Find the CONTAINER ID:

sudo docker ps -a

Run a BASH shell in the container:

sudo docker exec -it <CONTAINER ID> bash

Now, using a second SSH session to the ec2 instance, assign the cpu-shares of the docker container. Recall the container-id from above.

[>>In the *second terminal window with an ssh session to the ec2 instance*]
sudo docker update --cpu-shares="128" <container-id>

[>>In the first terminal *window where you ran "sudo docker exec -it <container-id> bash"*]
Repeat the stress test in the active BASH shell running in the Docker container (first SSH session):
stress-ng --cpu 2 --cpu-method fft --cpu-ops 5000

QUESTION 5. What happens to the runtime of the test?

For question 5, 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
[>>In the *second terminal window with an ssh session to the ec2 instance*]
sudo docker update --cpu-shares="1024" <container-id>

And then assign the containers "cpus"
[>>In the *second terminal window with an ssh session to the ec2 instance*]
sudo docker update --cpus=".5" <container-id>

Now, print out the **/sys/fs/cgroup/system.slice/docker-<FULL CONTAINER ID>.scope/cpu.stat** file before the test:

[>>In the *second terminal window with an ssh session to the ec2 instance*]
cat cpu.stat

[>>In the first terminal *window where you ran "sudo docker exec -it <container-id> bash"*]
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, use **usage_usec** from the **cpu.stat** file:
[>>In the Host window: *the second terminal window with an ssh session to the ec2 instance*]
cat cpu.stat

QUESTION 6. **a. What happened to the runtime of the test with cpus=0.5?**

b. What was the CPU utilization for the test (report the number)? (subtract cputime2 – cputime1)
c. How did it vary from our previous measurement from question 3 (higher vs. lower)?
d. In your own words, provide a *possible* explanation for this behavior. If you read the docker documentation that discusses the difference between "cpu-shares" and "cpus" it should be possible to answer the question. *(qualitative grading)*

Next, reset the CPU allocation for the container:

[>>In the *second terminal window with an ssh session to the ec2 instance*]
```
sudo docker update --cpus="2" <container-id>
```

At anytime, the container's resource configuration can be inspected using the following command:

[>>In the *second terminal window with an ssh session to the ec2 instance*]
```
sudo docker inspect <CONTAINER ID> | more
```

As a challenge, can you find which CPU related parameters change when adjusting "cpus" ?

Task 5 – Test CPU Isolation with Docker

Now, create a **NEW SEPARATE** terminal window to the ec2 instance, and create a second instance of the same container.
CONTROL-SHIFT-N in the terminal, opens a new terminal window.

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 containizeration.

Because the c6i.large is a 2 vCPU virtual machine, 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:
[>>In the *first terminal window, second tab with an ssh session to the ec2 instance*]
```
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.

[>>In the second terminal window *with an ssh session to the ec2 instance*]
```
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:
[>>Type in the 1st and 2nd Docker container terminals: *the windows where you ran "sudo docker exec -it <container-id> bash"*]
```
stress-ng --cpu 2 --cpu-method fft --cpu-ops 5000
```

First, run stress-ng in one container alone to measure the stand-alone performance of the command with cpus=1.

Next, prepare to run the command in both containers in parallel. This requires submitting commands to both containers ***as close as possible in time*** so their execution overlaps as much as possible. **YOU WILL NEED TO PRESS ENTER FOR THE COMMAND IN EACH CONTAINER AT ALMOST THE SAME TIME. PREPARE FOR THIS BEFORE RUNNING. DO NOT RUN TWICE. YOU WILL NEED TO RAPIDLY SW**

QUESTION 7 CPU Isolation: What is the performance difference when running the command standalone vs. running two instances at the same time in separate containers when CPUs have been set to 1? For simplicity, report:
1. the runtime of container A standalone
2. the runtime of container A and container B in parallel
3. the difference of container A standalone vs container A parallel.
4. The % difference: difference of container A parallel minus standalone divided by container A standalone. Then multiply by 100 to get percentage.

If container isolation is "perfect" for sharing the CPU, then performance of stress-ng should essentially be the same regardless of whether one or two containers were run.

Task 6 – 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". The throughput is reported in "MiB/sec".
This represents the memory transfer throughput per second.

Now, stage this command to perform the memory stress test on two containers at the same time (in parallel). PREPARE AHEAD AND PRESS ENTER IN EACH CONTAINER AS CLOSE TO THE SAME TIME AS POSSIBLE. Recall these two containers should have had their CPU's limited using the setting: --cpus="1"

Run the command at the same time in both containers:
```
sysbench --test=memory --memory-block-size=1M --memory-total-size=100G --num-threads=1 run
```

>> Write down the runtime, memory throughput (**total operations per second, and data transferred in MiB/sec**), and average latency (**latency (ms) avg**) of the standalone test.

If memory isolation is "perfect" for sharing the memory subsystem of the host, then memory transfer throughput (MiB/sec) should essentially be the same.

QUESTION 8 Memory Isolation: What is the memory throughput values (MiB/sec) for both containers A and B run in parallel?

QUESTION 9 Memory Isolation: What is the average memory latency (in ms) for both containers A and B run in parallel?

QUESTION 10 Memory Isolation: How did the memory throughput and memory latency change when comparing the standalone (1 container) test values with the concurrent container test?

QUESTION 11 Comparison:
(a) Using the values from container A, what was the % throughput/latency change for the sysbench memory test with 2 containers vs 1?
(b) Using the values from container A, what was the % runtime change for the sysbench memory test with 2 containers vs 1?

To answer this question, use the formulas to PROVIDE the difference between runtime (CPU isolation), and latency/throughput (Memory isolation).

$$\%diff_{throughput} = (throughput_2 - throughput_1) / throughput_1$$
$$\%diff_{latency} = (latency_2 - latency) / latency_1$$
$$\%diff_{runtime} = (runtime_2 - runtime_1) / runtime_1$$

Task 7 – Cleanup

At the end of the tutorial, with using EC2, you can optionally create an Amazon Machine Image (AMI) of your virtual machine with Docker installed. Reimaging your VM will allow it to be restored with Docker installed and ready-to-use with no setup effort required in the future. **This step is optional.**

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. If you create a new AMI, you may want to delete it at the end of the fall quarter to avoid EBS snapshot charges (5c/GB/month).

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.