Lab: Control Groups (cgroups)

> ***Difficulty**: Intermediate*
>
> ***Time**: Approximately 15 minutes*

Control Groups (cgroups) are a feature of the Linux kernel that allow you to limit the access processes and containers have to system resources such as CPU, RAM, IOPS and network.

In this lab you will use **cgroups** to limit the resources available to Docker containers. You will see how to pin a container to specific CPU cores, limit the number of CPU shares a container has, as well as how to prevent a *fork bomb* from taking down a Docker Host.

You will complete the following steps as part of this lab.

- Step 1 - cgroups and the Docker CLI
- Step 2 - Max-out two CPUs
- Step 3 - Set CPU affinity
- Step 4 - CPU share constraints
- Step 5 - Docker Compose and cgroups
- Step 6 - Preventing a fork bomb

Prerequisites

You will need all of the following to complete this lab:

- A Linux-based Docker Host with at least two CPU cores
- Docker 1.11 or higher
- The `htop` tool installed on your Docker Host (`apt-get install htop` on Ubuntu hosts)
- Root access on the Docker Host

<a name=“cli”></a>Step 1: cgroups and the Docker CLI

The `docker run` command provides many flags that allow you to apply cgroup limitations to new containers. The following flags are of note for this lab:

```
$ docker run --help
...
--cpu-shares                    CPU shares (relative weig
ht)
...
--cpuset-cpus                   CPUs in which to allow ex
ecution (0-3, 0,1)
...
--pids-limit                    Tune container pids limit
 (set -1 for unlimited)
```

For more information on cgroup related flags available to the `docker run` command, see the docker run reference guide.

<a name=“cpu_max”></a>Step 2: Max-out two CPUs

In this step you'll start a new container that will max out two CPU cores. You will need `htop` installed on your Docker Host, and you will need to be running a Docker Host that has two or more CPU cores. This step will still work if your Docker Host only has a single core. However, some of the `htop` outputs will be slightly different.

1. If you haven't already done so, install `htop` on your Docker Host.

Ubuntu with `apt`.

```
$ sudo apt-get install htop
```

CentOS with `yum`.

```
$ sudo yum install htop
```

2. Clone the lab's GitHub repo locally on your Docker Host and change into the `cgroups/cpu-stress` directory.

   ```
   $ sudo git clone https://github.com/docker/labs.git
   Cloning into 'labs'...
   remote: Counting objects: 1638, done.
   ```

```
Receiving objects: 100% (1638/1638), 8.84 MiB | 2.37 MiB/s, done.
Resolving deltas:  22% (33/148)   d 0 (delta 0), pack-reused 1638
Resolving deltas: 100% (148/148), done.
Checking connectivity... done.
Checking out files: 100% (1389/1389), done.

$ cd labs/security/cgroups/cpu-stress
```

3. Verify that the directory has a single `Dockerfile` and a single `docker-compose.yml` file.

```
$ ls -l
-rw-r--r-- 1 root root 23 Jul 11 09:49 docker-compose.yml
-rw-r--r-- 1 root root 85 Jul 11 09:49 Dockerfile
```

4. Inspect the contents of the `Dockerfile`.

```
$ cat Dockerfile
FROM ubuntu:latest

RUN apt-get update && apt-get install -y stress

CMD stress -c 2
```

As you can see, the Dockerfile describes a simple Ubuntu-based container that runs a single command `stress -c 2`. This command spawns two processes - both spinning on `sqrt()`. The effect of this command is to stress two CPU cores.

5. Build the image specified in the `Dockerfile`.

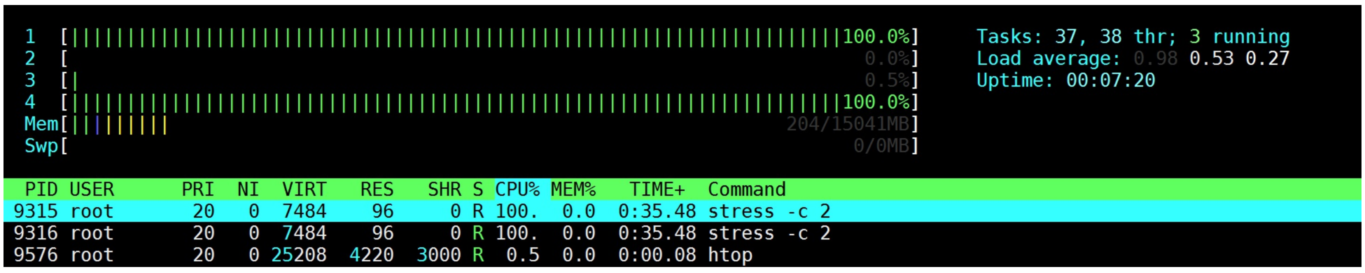
```
$ sudo docker build -t cpu-stress .
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM ubuntu:latest
latest: Pulling from library/ubuntu
90d6565b970a: Pull complete
40553bdb8474: Pull complete
<snip>
Step 3 : CMD stress -c 2
 ---> Running in b90defcccbb8
 ---> 9e6a3f316e91
Removing intermediate container b90defcccbb8
Successfully built 9e6a3f316e91
```

6. Run a new container called **stresser** based on the image built in the previous step.

```
$ sudo docker run -d --name stresser cpu-stress
stress: info: [5] dispatching hogs: 2 cpu, 0 io, 0
vm, 0 hdd
```

Be sure to run the container in the background with the `-d` flag so that you can run the `htop` command in the next step from the same terminal.

7. View the impact of the container using the `htop` command.



```
  1  [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]     Tasks: 37, 38 thr; 3 running
  2  [                                                                      0.0%]     Load average: 0.98 0.53 0.27
  3  [|                                                                     0.5%]     Uptime: 00:07:20
  4  [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]
  Mem[|||||||||                                                      204/15041MB]
  Swp[                                                                    0/0MB]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 9315 root       20   0  7484    96     0 R 100.  0.0  0:35.48 stress -c 2
 9316 root       20   0  7484    96     0 R 100.  0.0  0:35.48 stress -c 2
 9576 root       20   0 25208  4220  3000 R  0.5  0.0  0:00.08 htop
```

The output above shows two stress processes (**stress -c 2**) maxing out two of the CPUs on the system (CPU 1 and CPU 4). Both `stress` processes are in the running state, and both consuming 100% of the CPU they are executing on.

8. Stop and remove the **stresser** container.

```
$ sudo docker stop stresser && sudo docker rm stresser

stresser
stresser
```

You have seen how it is possible for a single container to max out CPU resources on a Docker Host. You would max out your entire Docker Host if you were to start a **stress** worker processes for each CPU core.

<a name=“cpu_affinity”></a>Step 3: Set CPU affinity

Docker makes it possible to restrict containers to a particular CPU core, or set of CPU cores. In this step you'll see how to restrict a container to a single CPU core using `docker run` with the `--cpuset-cpus` flag.

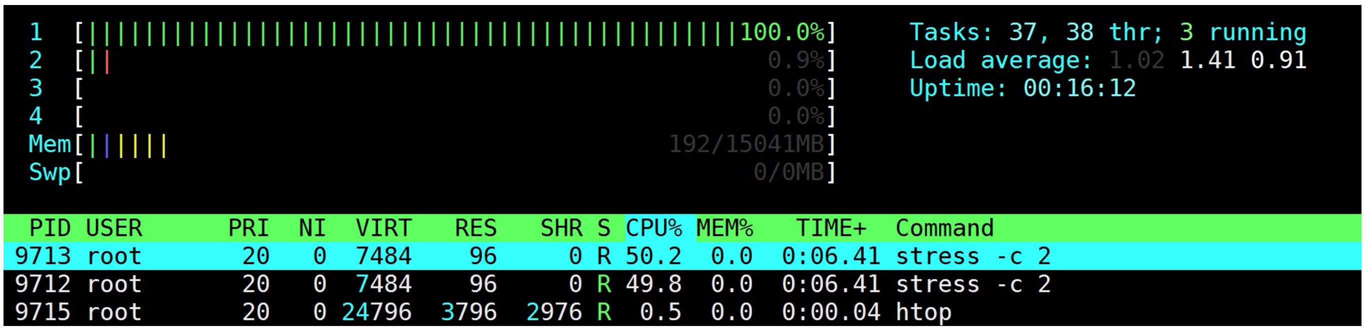
1. Run a new Docker container called **stresser** and restrict it to running on the first CPU on the system.

   ```
   $ sudo docker run -d --name stresser --cpuset-cpus
   0 cpu-stress

   0bfbf2d33516065bbcfa56bd8f9df24749312460141bca729f5
   3d66a9b2dba6b
   ```

The `--cpuset-cpus` flag indexes CPU cores starting at 0. Therefore, CPU 0 is the first CPU on the system. You can specify multiple CPU cores as `0-4`, `0,3` etc.

2. Run the `htop` command to see the impact the container is having on the Docker Host.


```
  1  [|||||||||||||||||||||||||||||||||||||||||||||||||100.0%]     Tasks: 37, 38 thr; 3 running
  2  [||                                                0.9%]     Load average: 1.02 1.41 0.91
  3  [                                                  0.0%]     Uptime: 00:16:12
  4  [                                                  0.0%]
  Mem[||||||||                                  192/15041MB]
  Swp[                                               0/0MB]

  PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 9713 root        20   0  7484    96     0 R 50.2  0.0  0:06.41 stress -c 2
 9712 root        20   0  7484    96     0 R 49.8  0.0  0:06.41 stress -c 2
 9715 root        20   0 24796  3796  2976 R  0.5  0.0  0:00.04 htop
```

There are a few things worth noting about what you have just done:

- The container is based on the same **cpu-stress** image that spawns two stress worker processes.
- The two stress worker processes have been restricted to running against a single CPU core by the `--cpuset-cpus` flag.
- Each of the two stress processes is consuming ~50% of available time on the single CPU core they are executing on.
- `htop` indexes CPU cores starting at 1 whereas `--cpuset-cpus` indexes starting at 0.

You have seen how to lock a container to a executing on a single CPU core on the Docker Host. Feel free to experiment further with the `--cpuset-cpus` flag.

<a name=“cpu_share”></a>Step 4: Set CPU share constraints

By default, all containers get an equal share of time executing on the Docker Host’s CPUs. This allocation of time can be modified by changing the container’s CPU share weighting relative to the weighting of all other running containers.

The `--cpu-shares` flag takes a value from 0-1024. The default value is 1024, and a value of 0 will also default to 1024. If three containers are running and one has 1024 shares, while the other two have 512, the first container will get 50% of processor time while the other two will each get 25%. However, these shares are only enforced when CPU intensive tasks are running. When a container is not busy, other containers can use the free CPU time.

Shares of CPU time are balanced across all cores on a multi-core Docker Host.

In this step you will use the `docker run` command with the `--cpu-shares` flag to influence the amount of CPU time containers get. You will start one container (**container-1**) with 768 shares, and another container (**container-2**) with 256 shares. Each container will be locked to the first CPU on the system, and each will be running the `stress -c 2` process. **Container-1** will receive 75% of the CPU time whereas **container-2** will receive 25%.

1. Start the first container with 768 CPU shares.

   ```
   $ sudo docker run -d --name container-1 --cpuset-cp
   us 0 --cpu-shares 768 cpu-stress
   ```

2. Start the second container with 256 CPU shares.

   ```
    $ sudo docker run -d --name container-2 --cpuset-c
   pus 0 --cpu-shares 256 cpu-stress
   ```

3. Verify that both containers are running with the `docker ps` command.

   ```
   $ sudo docker ps

   CONTAINER ID        IMAGE               COMMAND
                 CREATED             STATUS
      PORTS               NAMES
   ```

725dc16fac5a cpu-stress “/bin/sh -c 'stress -” 2 minutes ago Up 2 minutes container-2
f82f95757d3f cpu-stress “/bin/sh -c 'stress -” 2 minutes ago Up 2 minutes container-1

```


4. View the output of `htop`.

![](http://i.imgur.com/zRqkQHt.png)

Notice two things about the `htop` output. First, only a
single CPU is being maxed out. Second, there are four `st
```

```
ress` processes running. The first two in the list equate
 to ~75% of CPU time, and the second two equate to ~25% o
f CPU time.

In this step you've seen how to use the `--cpu-shares` fl
ag to influence the amount of CPU time containers get rel
ative to other containers on the same Docker Host.

Feel free to experiment further by running more container
s with different relative weights.

# <a name="compose"></a>Step 5: Docker Compose and cgroup
s

In this step you'll see how to set CPU affinities via Doc
ker Compose files.

See [this section](https://docs.docker.com/compose/compos
e-file/#cpu-shares-cpu-quota-cpuset-domainname-hostname-i
pc-mac-address-mem-limit-memswap-limit-privileged-read-on
ly-restart-shm-size-stdin-open-tty-user-working-dir) of t
he Docker Compose documentation for more information on l
everaging other cgroup capabilities using Docker Compose.

Make sure you are in the `dockercon-workshop/cgroups/cpu-
stress` directory of the repo that you pulled in Step 2.

1. Edit the `docker-compose.yml` file and add the `cpuset
```

```
: '3'` line as shown below:

```

cpu-stress:

build: .

cpuset: ‘3’

```

The above `docker-compose.yml` file will ensure that cont
ainers based from it will run on CPU core #3. You will ob
viously need a Docker Host with at least 4 CPU cores for
this to work.

2. Bring the application up in the background.

  ```
  $ sudo docker-compose up -d

  Creating cpustress_cpu-stress_1
  ```

3. Run `htop` to see the effect of the `cpuset` parameter
.

![](http://i.imgur.com/DsCOSSB.png)

The `htop` output above shows the container and it's two
```

```
`stress` processes locked to CPU core 4 (`cpuset` in Dock
er Compose indexes CPU cores starting at 0 whereas `htop`
 indexes CPU cores starting at 1).

In this step you've seen how Docker Compose can set conta
iner CPU affinities. Remember that Docker Compose can als
o set CPU quotas and shares. See the [documentation](http
s://docs.docker.com/compose/compose-file/#cpu-shares-cpu-
quota-cpuset-domainname-hostname-ipc-mac-address-mem-limi
t-memswap-limit-privileged-read-only-restart-shm-size-std
in-open-tty-user-working-dir) for more detail.


# <a name="fork_bomb"></a>Step 6: Preventing a fork bomb

A *fork bomb* is a form of denial of service (DoS) attack
 where a process continually replicates itself with the g
oal of depleting system resources to the point where a sy
stem can no longer function.

In this step you will use the `--pids-limit` flag to limi
t the number of processes a container can fork at runtime
. This will prevent a fork bomb from consuming the Docker
 Host's entire process table.

1. Start a new container and limit the number of processe
s it can create to 200 with the following command.

```

```
$ sudo docker run --rm -it --pids-limit 200 debian:jessie bash

Unable to find image 'debian:jessie' locally
jessie: Pulling from library/debian
5c90d4a2d1a8: Pull complete
Digest:
sha256:8b1fc3a7a55c42e3445155b2f8f40c55de5f8bc8012992b26b57053
Status: Downloaded newer image for debian:jessie

root@c0eb76d2481c:/#
```

```
**Do not proceed to the next step** if you receive the fo
llowing warning about pids limit support - *"WARNING: You
r kernel does not support pids limit capabilities, pids l
imit discarded"*. If you receive this warning, you can [w
atch the demo](https://asciinema.org/a/dewdpjlzom4zasdz0c
0c46jt9) as a video instead.

2.  Run the following command to create a fork bomb in th
e container you just started.

The previous command will have attached your terminal to
the `bash` shell inside of the container created in the p
revious step.
```

```
**Do not complete this step if you received the error tha
t your kernel does not support the pids limit feature...*
*

```

root@c0eb76d2481c:/# :(){ :|: & };:

[1] 6

root@a3eeb655f301:/# bash: fork: retry: No child processes

bash: fork: retry: No child processes

bash: fork: retry: No child processes

bash: fork: retry: No child processes

bash: fork: retry: No child processes

bash: fork: retry: Resource temporarily unavailable

bash: fork: retry: No child processes

<snip>

bash: fork: retry: No child processes

bash: fork: retry: Resource temporarily unavailable

bash: fork: retry: No child processes

^C

[1]+ Done : | :

```

> **Note:** The command above is `:(){ :|: & };:`.

You will need to press `Ctrl-C` to stop the process.

```

```
In this step you have seen how to set a process limits on
 a container that will prevent it from consuming all proc
ess table resources on the underlying Docker Host.


# Summary


Congratulations. You've seen how to use some of the **cgr
oups** features supported by Docker that allow you to lim
it and constrain a container's access to Docker Host syst
em resources.

```