McGill

Programming Assignment #3: Simple Resource Container

Due date: Check My Courses`

This assignment could be completed in groups of 2 (max). You are expected to split the workload evenly in the group.

This assignment needs root privileges. We have created an OS playground in cs310.cs.mcgill.ca. You can use that or your own machine. You are strongly advised to use another container like Docker or a Virtual Machine to develop this assignment. This way you can prevent any harmful modifications to your Linux installation. We have provided a default docker image ***(in cs310.cs.mcgill.ca)*** *which you can use to spawn a container. Then you will enter this container and tryout everything that is listed below. See APPENDIX to how to get into this playground environment.* ***READ APPENDIX before you proceed****.*

This assignment will be done in two stages. In this first stage of the assignment, you are exploring all the concepts that go into a Simple Resource Container (SRContainer). The way you would do this is by building and experimenting with SRContainer without writing any code. We need to make few simplifying assumptions to make this work. In the second stage, you will implement the real SRContainer using the template we provide to you. The knowledge you gain with this part is necessary to complete that part. The template will allow you to complete the assignment with minimal amount of coding.

1. Overview

In this assignment you are expected to develop a SRContainer that would run in Linux. The SRContainer is modelled after the highly popular Docker container format. The SRContainer is implemented using advanced features supported by Linux such as: ***namespaces***, ***control groups***, ***system call filtering***, and fine-grained ***capability*** management.

A container is a virtual machine look alike. It does what we expect from a machine from an applications point-of-view. It is not a machine by itself. Rather, it is part of another machine – the host. The basis for container creation is the clone() system call that was already discussed in Assignment 1. A container is simply a process spawned within the host with proper isolation (using clone() flags). The container has its own resources or partitions of the resources. It has its own file system, network configuration, memory and CPU slices. For example, the file system could be specific to the container. A container running on Ubuntu could have an Alpine Linux file system. Because the containers are sharing the kernel instance running in the host, you cannot have a file system from an incompatible operating system (non Linux OS). For instance, a FreeBSD file system would not work for a container inside a host running some Linux kernel.

In a Unix-based operating system, we already have a strong isolation scheme between processes in terms of fault tolerance. This allows one process to keep running while another process may crash. However, from a security point-of-view the isolation between the processes is quite weak. This has motivated research into sandboxing where a group of processes could be isolated from the host or from another group of processes. The web browser application is a good example that uses process sandboxing to shield the host and the user data from untrusted code downloaded from remote sites and run within the browser.

For the purposes of this assignment we consider the container as follows:

Container = Processes + Isolation + File System Image + Resource Principal + **Host-Kernel**

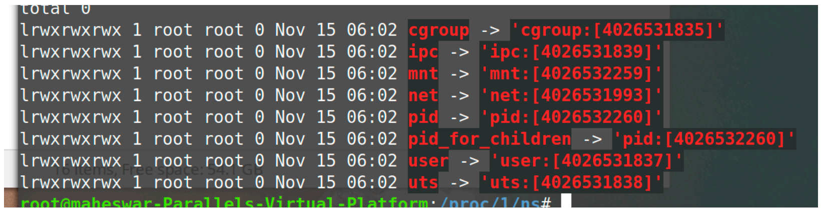
The major purpose of this assignment is to learn the different ways isolation can be added to processes to create containers. Also, we see how the newly created container can be made to run a file system that is different from the host filesystem. You are expected to use this assignment to explore the design space for containers in a Linux like operating system. The knowledge from this first stage will allow you to complete the "from-scratch-containers" skeleton code in the second stage.

Another major feature of containers is process independent resource monitoring, accounting, and control. Using this feature, we can package an application inside a container and use it to monitor and control the resource usage on a per-application basis. This is very helpful with micro-services (a popular software engineering paradigm) when they are hosted in clouds. With an application focused resource usage control framework, we can precisely manage the resource consumption levels.

2. Exploring Namespaces

Processes are built on the idea of address spaces. Each process has its own address space that prevents a process from interfering with another process. The container idea is based on a similar concept called **namespaces**. When you start the operating system all the processes go into the same namespace. In Linux, we have 7 namespaces: PID, IPC, UTS, NET, USER, CGROUP, and MNT. So, without containers all processes would go into the same instance of the PID namespace, same instance of the IPC namespace and so on. First step is to verify this yourself.

Go to the **`/proc`** interface. This interface allows you to see the current kernel state and even control it. Change into this directory and explore it. You will see lot of information. Each process will have its own folder with its process ID (PID). Change into the folder **`self`**. This is pointing to the process of the shell that you are using to explore the **`/proc`** interface. You can confirm that by looking into the **`cmdline`** information available in the folder. Go to the **`ns`** folder to look at the namespace information. You will see something like the following there. It shows the information for each type of namespace for the shell.



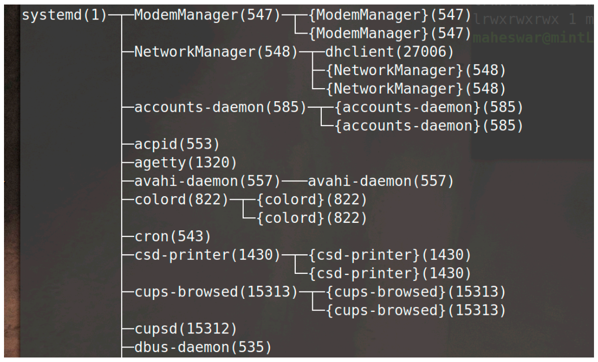
```
total 0
lrwxrwxrwx 1 root root 0 Nov 15 06:02 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 Nov 15 06:02 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 root root 0 Nov 15 06:02 mnt -> 'mnt:[4026532259]'
lrwxrwxrwx 1 root root 0 Nov 15 06:02 net -> 'net:[4026531993]'
lrwxrwxrwx 1 root root 0 Nov 15 06:02 pid -> 'pid:[4026532260]'
lrwxrwxrwx 1 root root 0 Nov 15 06:02 pid_for_children -> 'pid:[4026532260]'
lrwxrwxrwx 1 root root 0 Nov 15 06:02 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 Nov 15 06:02 uts -> 'uts:[4026531838]'
root@maheswar-Parallels-Virtual-Platform:/proc/1/ns#
```

You can explore different processes running in the system including the **`init (or systemd)`** process (process ID 1) that bootstrapped the system. Check the namespace information for the different processes and verify that most of them belong to the same namespace.

Print out the process tree using the following command.

```
pstree -g -s
```

A partial output of the process tree printed by the previous command is shown below.



```
systemd(1)─┬─ModemManager(547)─┬─{ModemManager}(547)
           │                   └─{ModemManager}(547)
           ├─NetworkManager(548)─┬─dhclient(27006)
           │                     ├─{NetworkManager}(548)
           │                     └─{NetworkManager}(548)
           ├─accounts-daemon(585)─┬─{accounts-daemon}(585)
           │                      └─{accounts-daemon}(585)
           ├─acpid(553)
           ├─agetty(1320)
           ├─avahi-daemon(557)───avahi-daemon(557)
           ├─colord(822)─┬─{colord}(822)
           │             └─{colord}(822)
           ├─cron(543)
           ├─csd-printer(1430)─┬─{csd-printer}(1430)
           │                   └─{csd-printer}(1430)
           ├─cups-browsed(15313)─┬─{cups-browsed}(15313)
           │                     └─{cups-browsed}(15313)
           ├─cupsd(15312)
           ├─dbus-daemon(535)
```

You will notice that the host is running a whole lot of processes. The process tree shows all the processes running in the host. This would include even the processes that are inside containers *because they are also running the host.*

Let's start experimenting with namespaces that would allow us to put processes in different namespaces. We begin the experimentation using the **`unshare`** command. The "unshare" command *(a system call and has the shell command wrapper)* allows you to run processes in namespaces disassociated to the parent namespace. You can *unshare* from any/all of the parent's namespaces. Let's use it to run a shell *(let's use the **`/bin/sh`** to start with)* in a different namespace. That is, we want to create a namespace and put the shell in that one. The unshare command does that and you can select the namespace type using one of its options. We will put the shell in its own **PID namespace** (the **`-p`** option).

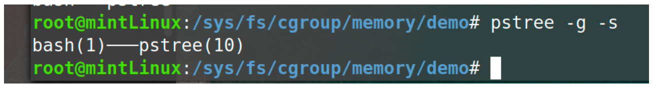
```
sudo unshare -fp /bin/sh
```

Note: The **-f** option tells unshare to run the program **"/bin/sh"** as a sub-process via fork(). If not specified the *unshare* process itself will be replaced by **/bin/sh** which can cause issues. Read here for more clarity.

Run the **`pstree`** command previously shown in the shell. Observe the processes that are shown in the output. They are not much different from the previous list because the pstree command is getting the process listing information from the /proc file system interface. So, we are seeing all the processes instead of confining the output to the newly created namespace that is holding the shell process. To confine the process listing to the ones in the newly created namespace, we need to mount the proc file system again. This is done in the command below.

```
sudo unshare -fp --mount-proc=/proc /bin/bash
```

Run the **`pstree`** command in the shell. Note the processes that are listed over there. You will see an output like the following. It is interesting to note that instead of **`systemd`** (**`init`**) process, we have the **`bash`** program as the originating process.



```
root@mintLinux:/sys/fs/cgroup/memory/demo# pstree -g -s
bash(1)───pstree(10)
root@mintLinux:/sys/fs/cgroup/memory/demo#
```

Using the **`unshare`** command you can start isolated process groups. For instance, in the above command you launched the bash shell. Now, using the bash shell you can any program, which can potentially spawn many numbers of child processes. All of those processes will inherit the new namespace that we just created using the **`unshare`** command. Processes that were put into their own namespaces with "unshare" are what are called ***containers***. You can run multiple processes inside this new namespace, which means multiple processes running inside *the container. Namespaces providing the building blocks for containers.*

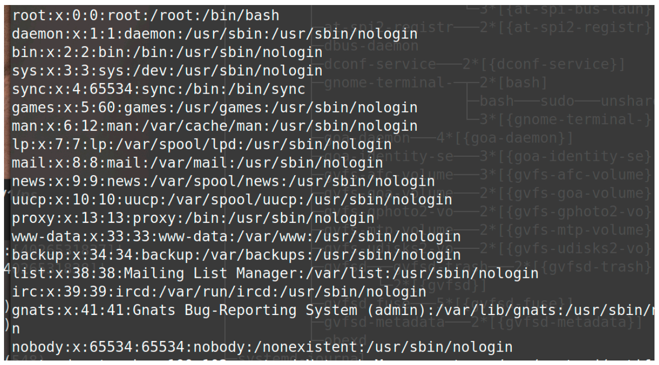
Start two different bash shells using the previous command. These two bash shells will be in two different PID namespaces. Run some arbitrary programs in the two different shells. Once you are comfortable with the shells, do the following experiments.

1. Run arbitrary programs (eg: “ping 8.8.8.8”, “tr ABC 123”) in the two different shells. Do you see the processes that you run in one shell from the other shell?
2. Do you see the programs you run in the shell from the host?
3. Can you kill the programs that you run the shells from outside (i.e., outside the shell but in the same host)? To do this you need to have another terminal. If you are doing this experiment in a Docker container, you need to run a docker exec to get into the container because the docker container is your host.
4. Launch some programs in the host. Do you see those processes inside the shells?

(use: **htop -u <SOCS_USERNAME>** to list processes run by you)

In the above experiment, you used the PID namespace. The general idea of PID namespace is to reuse the same PID values in the different namespaces. For instance, the host would have PID values starting from 1, where 1 is the **init** (**systemd**) process. A child namespace would also have its own PID 1. The container system designer is free to select the program that would actually be the PID 1 inside each namespace. Once the process with PID-1 inside a namespace (container) dies the container ceases to exist. Just like how the system is shutdown when ***init*** is down. There are many advantages of PID reuse in the different namespaces. One of them is the ability to move a set of processes from one machine to another. For more information about PID namespaces type the following command: **`man pid_namespaces`**

Now, let’s turn the attention to another namespace: **USER**. The objective of USER namespace is again the same as the objective of the PID namespace. We want to have the same user ID and group ID values in different namespaces. You can get more information regarding from it man page. For example, UID 0 and group ID 0 are associated with the root user and root group, respectively. The screenshot below shows the entries from a password file (*cat /etc/passwd*). For more information on USER namespaces, consult the man page: **`man user_namespaces`**



```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/
n
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
```

Let’s simply launch a shell with a detached USER namespace like the following. This command is going to put the shell in a different namespace from the parent for both PID and USER namespaces.

```
sudo unshare -fpU --mount-proc=/proc /bin/bash
```

Note: *its uppercase “U”. Lowercase “u” is for UTS namespace.*

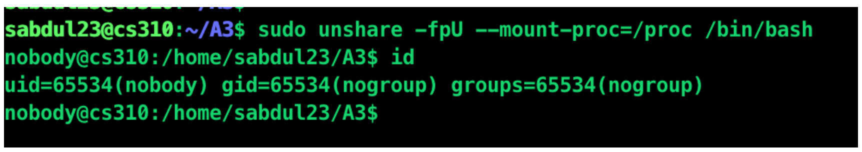
Immediately, you will notice a problem. The user running the bash has changed now from the previous run that had the same user mapping across the host (parent namespace) and the "container" (child namespace). When you run without detaching USER namespace like the following:

```
sudo unshare -fp --mount-proc=/proc /bin/bash
```

***Note:** type "id" to check who the user is inside the container/unshared-namespace.*

The user in the shell in the new namespace is actually **root**! That is, there is a seaming escalation of privileges (i.e., a normal user in the host or the parent namespace went to administrator level in the new namespace). In closer inspection, it turns out that there is no escalation of privilege issue because the **unshare** was run as the administrator. So, although the user was a non-administrator, the **sudo** command was used to escalate the privileges.

With the detached USER namespace, you will notice that the shell is running with a different prompt that it was when you just had PID namespace. Type **id** in the shell to find out user ID (UID) and group ID (gid) of the shell process. You will notice that the shell is running with *nobody* as the user. The *nobody* is a standard user with the least privileges – just the opposite of root! See the screenshot below – it is the last entry there.



```
sabdul23@cs310:~/A3$ sudo unshare -fpU --mount-proc=/proc /bin/bash
nobody@cs310:/home/sabdul23/A3$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
nobody@cs310:/home/sabdul23/A3$
```

Previously, when we ran the shell in the following way, we were running the shell as the root (most privileged user).

```
sudo unshare -fp --mount-proc=/proc /bin/bash
```

So, when we detach the USER namespace, we just created a USER namespace and put the process (shell) in that namespace. This is not sufficient. We need to provide a UID and GID mapping that would relate the IDs in that space to the ones in the host USER namespace.

Run a command such as **adduser** that needs higher privileges in the shell with least privileges. You will notice that the command would not work due to lack of privileges. The same command would have progressed much further with its execution when the shell was running without a separate USER namespace (i.e: when run as **root**).

Let's fix this problem. We need to specify a UID and GID mapping.

Open another terminal in the parent namespace. Run the following command.

```
ps axu | grep bash
```

You will see the **unshare../bin/bash** command there. You need to note down the PID of the process.

```
sudo newuidmap PID 0 0 1
```

This will upload a mapping between the root (0) in the parent namespace and child namespace. Similarly, for the group

```
sudo newgidmap PID 0 0 1
```

Now run the id again in the shell terminal. You will see that we have changed the user identity. It is not nobody anymore. It is back to root. The privilege level you get changes according to the user ID mapping.

Lastly, let's examine the UTS namespace. UTS namespaces provide isolation of two system identifiers: the **hostname** and the **NIS domain** name. Let's consider the hostname here.

Run the following command to create a child namespace that is isolated in UTS from the parent namespace.

```
sudo unshare -fpu --mount-proc=/proc /bin/bash
```

You can change the hostname in the parent namespace using

```
hostname xyxy
```

Observe whether the hostname in the shell (child namespace) is changing or not. You can type **`hostname`** to printout the hostname. Do the same experiment without isolated UTS namespaces and see what happens.

3. Exploring Chroot

So far, we have managed to put the shell process into different process, user and UTS namespaces. Despite those new namespaces we could not provide the virtual machine illusion. The main reason was the fact that the shell was reusing the same file system as the underlying host. So, we could see the files of the host from within the container/namespace. We need to isolate the file system and provide the shell its own file system.

To provide the shell its own file system, we need a **root file system**. We can get a root file system in two different ways: create one using **`debootstrap`** or download a root file system from a repository. Using **`debootstrap`** you can do the following. Create a **`rootfs`** folder.

```
debootstrap jessie jrootfs        (Gets a minimalistic file-system from Debian-jessie)
debootstrap stretch srootfs       (Gets a minimalistic file-system from Debian-stretch)
```

This should create a root file system at **`rootfs/jroofs`** that is based on Debian Jessie. You can use this in your shell as the file system. Change into the directory after having done the other isolations as discussed previously.

```
cd jrootfs
sudo unshare -fpu --mount-proc=/proc /bin/bash
chroot .
```

This should create a file system that is isolated from the host. Although your root file system is a folder in the host, it gives you isolated feel because you are using different files: binaries, libraries, configs, etc. Right away, you will run into some problems. For instance, process reated commands like **`ps`** and **`top`** would not work. You need to mount the proc file system using a command like

```
mount -t proc proc /proc
```

Instead of using debootstrap which is good for Debian root file systems, you could download Linux root file systems for use here. One of them in Alpine Linux (a very lightweight Linux based on busybox). Below is the URL for downloading the root file system. Unpack it and use it just like the previous one.

http://dl-cdn.alpinelinux.org/alpine/v3.8/releases/x86_64/alpine-minirootfs-3.8.1-x86_64.tar.gz

Once you do the change root, you get a VM like feel. However, we also created another problem. We lost association with an important folder. For the next step (control resource allocations), we are going to use control groups (**cgroups**). This is accessed through the **/sys/fs/cgroup** folder. This folder is not accessible after the chroot. In theory, we should be able to inject this folder from the host into the shell after the shell is launched. However, that is not working in our tests. So, here is what is known to work.

You recursively bind mount the **/sys/fs/cgroup** folder on the root file **system before you launch the shell**. Then, you start the shell as described above. You should see the cgroup folder from your "vm" at **/sys/fs/cgroup** and it should be functional!

To recursively bind mount, use the following command.

```
mount --rbind /sys/fs/cgroup $ROOTFS/sys/fs/cgroup
```

$ROOTFS: Is the path to where your root-file-system is (eg: **jrootfs** created above)

You must create the folders **"fs"** and **"cgroup"** at **"/sys"** inside your container

Also, you should run the above command in the host/playground container

4. Exploring Control Groups

We have looked at isolation and file system for the SRContainer in the previous sections. In this section, we are going to see how we can do resource monitoring and control. Let's say we want to restrict the SRContainer instance to kill an application if the application is consuming too much memory. Before we see how that can be done, we show you an example memory-hog. This just keeps eating memory and if we have a out-of-memory (OOM) killer, the process should be killed after it has run for a while.

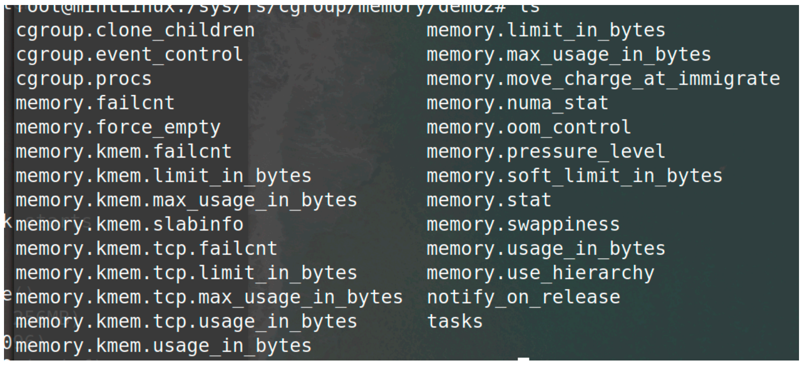
```
int main()
{
   int i;
   long sz = 100000;
   mlockall( MCL_FUTURE | MCL_ONFAULT );    // Lock virtual memory from being swapped to disk

   for (i = 0; i < 10000; i++)
   {
          printf("Allocating block %d \n", i);
          fflush(stdout);
          char *p = malloc(sz);
          if (p == NULL) {
                 printf("Allocation error.. \n");
                 fflush(stdout);
          } else {
                 printf("Success.. \n");
                 fflush(stdout);
          }
          memset(p, 0, sz);                      // malloc'ed memory must be used
          usleep(10000);
   }
    return EXIT_SUCCESS;
}
```

First, we create the executable from the above code. Second, we go into /sys/fs/cgroup. You will see the memory controller. Do the following commands:

```
cd memory
mkdir ExMemLimiter
cd ExMemLimiter
```

At this point, you will see the internal directory (newly created one is already populated!).



```
root@mintLinux:/sys/fs/cgroup/memory/demo2# ls
cgroup.clone_children              memory.limit_in_bytes
cgroup.event_control               memory.max_usage_in_bytes
cgroup.procs                       memory.move_charge_at_immigrate
memory.failcnt                     memory.numa_stat
memory.force_empty                 memory.oom_control
memory.kmem.failcnt                memory.pressure_level
memory.kmem.limit_in_bytes         memory.soft_limit_in_bytes
memory.kmem.max_usage_in_bytes     memory.stat
memory.kmem.slabinfo               memory.swappiness
memory.kmem.tcp.failcnt            memory.usage_in_bytes
memory.kmem.tcp.limit_in_bytes     memory.use_hierarchy
memory.kmem.tcp.max_usage_in_bytes notify_on_release
memory.kmem.tcp.usage_in_bytes     tasks
memory.kmem.usage_in_bytes
```

Set the limit by using the following command.

```
echo <integer> > memory.limit_in_bytes
```

Next, we must add our container to this cgroup (controller). We will do this by writing the **PID** of our container the file called **“tasks”** in the list above. You need to be in the **`/sys/fs/cgroup/memory/ExMemLimiter`** folder.

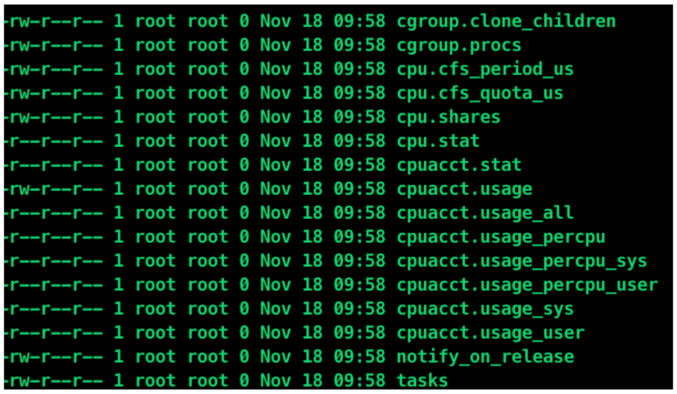
```
echo 1 > tasks_name
```

Now when you run the memory hogging program given above inside your container, it must be killed upon reaching the limit you set above. Any program that runs inside the container is governed by the memory controller. It is important to note that we did not associate the memory hogging program directly. The memory hogging program was restricted by the controller attached to the container.

Another, resource you can restrict per container is the **CPU usage**. To control this, you need to create 2 **cgroup controllers**: ***cpu*** and ***cpuset***. You have to traverse into the “/sys/fs/cgroup” folder and do the following:

```
cd cpu
mkdir CPULimiter
cd CPULimiter
```

You will see the following cgroup metrics enumerated:



```
-rw-r--r-- 1 root root 0 Nov 18 09:58 cgroup.clone_children
-rw-r--r-- 1 root root 0 Nov 18 09:58 cgroup.procs
-rw-r--r-- 1 root root 0 Nov 18 09:58 cpu.cfs_period_us
-rw-r--r-- 1 root root 0 Nov 18 09:58 cpu.cfs_quota_us
-rw-r--r-- 1 root root 0 Nov 18 09:58 cpu.shares
-r--r--r-- 1 root root 0 Nov 18 09:58 cpu.stat
-r--r--r-- 1 root root 0 Nov 18 09:58 cpuacct.stat
-rw-r--r-- 1 root root 0 Nov 18 09:58 cpuacct.usage
-r--r--r-- 1 root root 0 Nov 18 09:58 cpuacct.usage_all
-r--r--r-- 1 root root 0 Nov 18 09:58 cpuacct.usage_percpu
-r--r--r-- 1 root root 0 Nov 18 09:58 cpuacct.usage_percpu_sys
-r--r--r-- 1 root root 0 Nov 18 09:58 cpuacct.usage_percpu_user
-r--r--r-- 1 root root 0 Nov 18 09:58 cpuacct.usage_sys
-r--r--r-- 1 root root 0 Nov 18 09:58 cpuacct.usage_user
-rw-r--r-- 1 root root 0 Nov 18 09:58 notify_on_release
-rw-r--r-- 1 root root 0 Nov 18 09:58 tasks
```

You have make changes to the "**`cpu.shares`**" file.

```
echo <integer> > cpu.shares
```

Now, this control on the CPU is a relative control. That is, it is a ratio of CPU-percentage to be allocated to processes of this cgroup with respect to others. So, to validate this we must have ***at-least two containers*** running; each of them attached to a different cpu-controllers. So, you must run a second unshared-container instance and connect it to new CPU-controller. Let's call it **CPULimiter2** cgroup. You must enter an integer into the new **`cpu.shares`** file.

```
echo 10*<integer> > cpu.shares
```

Notice that now we have "10" times CPU-shares set. Now when you run similar loads on these two instances of unshared-containers (each associated to a different controller), you must see CPU allocation split 1:10 between them. We have to take care of one more thing here. We have to ensure that the containers are restricted to one CPU-core. If not, the loads may run on separate cores to 100%. We can restrict them to one core by changing the **cpuset cgroup controller**. Now traverse to the **cpuset** folder under "/sys/fs/cgroup" and create a new cgroups:

```
mkdir CPURestrictor
```

Change the **`cpuset.cpus`** file under this directory as follows:

```
echo 0 > cpuset.cpus
```

Change the **`cpuset.mems`** file under this directory as follows:

```
echo 0-1 > cpuset.mems
```

This will restrict both containers to CPU-core-0 with 2 memory nodes.

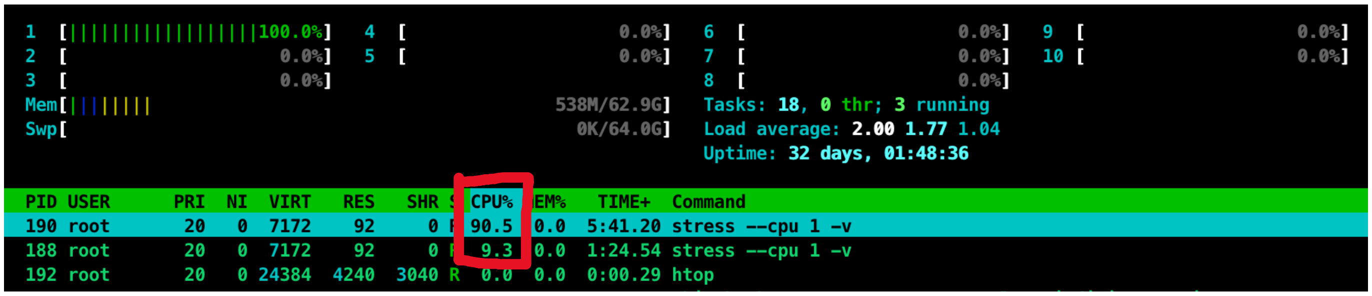
Now add the container PID to the **`tasks`** file under **`CPULimiter`** and the other under the **`tasks`** file within **`CPULimiter2`**. Also add both (container) PIDs inside the **`tasks`** file under **`CPURestrictor.`** The **`CPULimiter`** would set the CPU-limit on the first container instance and **`CPULimiter2`** would set the CPU-limit on the second container instance. The **`CPURestrictor`** would restrict both containers to run on a single core thus having to share it based on their weight (1:10).

Finally, run the following program in **both** unshared containers:

```
stress --cpu 1 -v
```

(you have to install stress in your rootfs)

Now when you run **`htop`** or **`top`** in the host container to observe, the CPU should be split 1:10 between the two containers as shown below:



```
  1  [||||||||||||||||||100.0%]   4  [                    0.0%]   6  [                    0.0%]   9  [                    0.0%]
  2  [                    0.0%]   5  [                    0.0%]   7  [                    0.0%]   10 [                    0.0%]
  3  [                    0.0%]                                   8  [                    0.0%]
  Mem[||||||||                                   538M/62.9G]   Tasks: 18, 0 thr; 3 running
  Swp[                                             0K/64.0G]   Load average: 2.00 1.77 1.04
                                                               Uptime: 32 days, 01:48:36

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
  190 root       20   0  7172    92     0 R 90.5  0.0  5:41.20 stress --cpu 1 -v
  188 root       20   0  7172    92     0 R  9.3  0.0  1:24.54 stress --cpu 1 -v
  192 root       20   0 24384  4240  3040 R  0.0  0.0  0:00.29 htop
```

We will let controlling other resources such as **block-io** and **network** as a practice for you to try. We will ask about this on the viva session for grading this assignment.

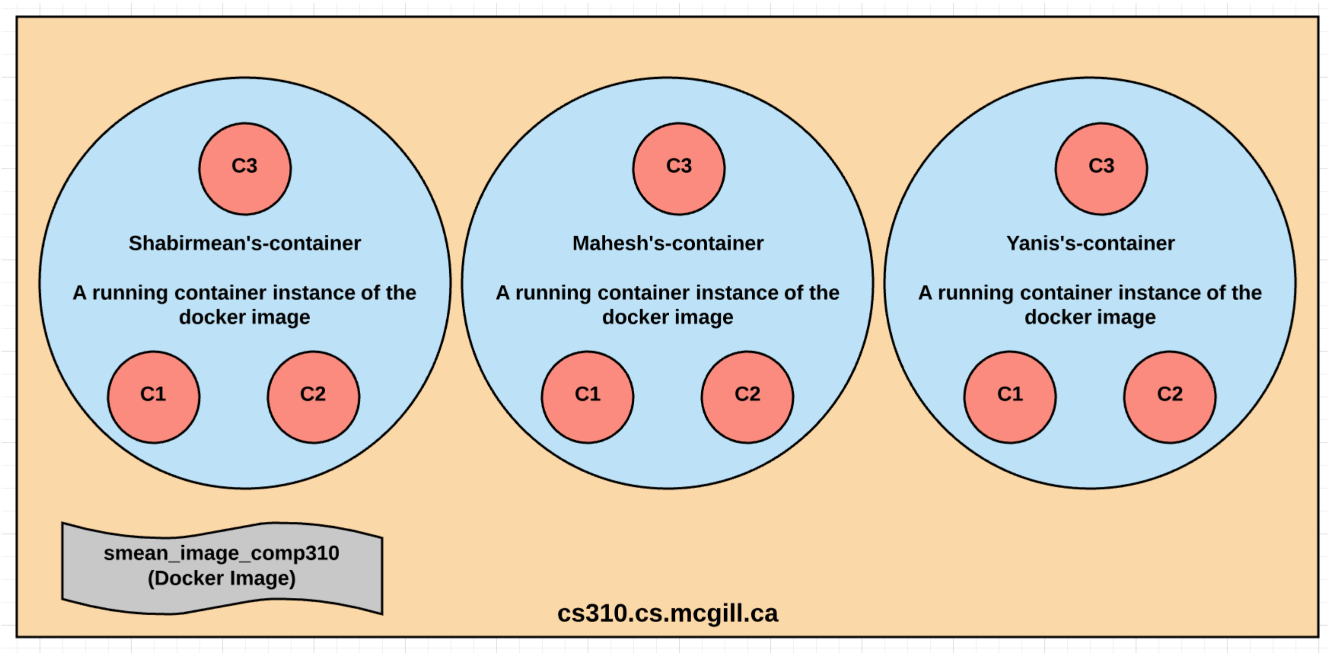
5. Testing and Evaluation

- Change **hostname** inside the container and show that this does not affect the host's hostname
- Create two containers with different UID/GID mappings and explain how these mappings are understood with respect to each namespace
- Show cgroup control over memory
- Show CPU-sharing between multiple containers
- Show Block-IO read restriction on a container
- Show Block-IO write restriction on a container

APPENDIX:

We have set up a dedicated server ***(cs310.cs.mcgill.ca)*** to do this entire assignment. Since, we will be playing with some of the core kernel features, you will need **"root"** access to run most of these commands. And, since it is not possible to provide all students with **"root"** access to this server we have created a Docker-image that can be used by everyone.

Docker is simply a tool-set that allows you to quickly get ***containers*** up and running. A docker image is basically a template for the ***container*** that will be spawned. So, we have created an image that will give you a complete (container) environment within which you can do whatever you want. You will have root *(privileged)* access inside this container. You can do whatever you want inside this container and you will not affect the ***(cs310.cs.mcgill.ca)*** server. So, we request all students to spawn their own container-instance using this image. Then, you can get into this running instance and try-out all that is explained above. You can see how it would look like from the figure below. Here, C1, C2 and C3 would be the containers you will be creating inside the isolated **(root'ed)** container environment. Please note that, whenever the description above uses the term **"host**" or "**host-container**" it means the container instance from within which you will be working *(eg: Shabirmean's-container*). When it says just **container** or **unshared-container**, it denotes one of the containers *"you created"* (i.e. C1, C2 or C3).


C3
Shabirmean's-container
A running container instance of the docker image
C1
C2
C3
Mahesh's-container
A running container instance of the docker image
C1
C2
C3
Yanis's-container
A running container instance of the docker image
C1
C2
smean_image_comp310 (Docker Image)
cs310.cs.mcgill.ca

The name of the image made available for you is: ***smean_image_comp310***

You can create your own running instance of a container using this image by issueing the following command:

```
docker run --privileged -d --rm --name=<MEANINGFUL_NAME> smean_image_comp310
```

Replace **<MEANINGFUL_NAME>** with a proper name so you know which your own container is.

Once you spawn a container as above, it will be ever running. Now you can get into this container by issuing the following command:

```
docker exec -it <MEANINGFUL_NAME> /bin/bash
```

When you issue the following command, you will be inside one of the blue circles shown above. That is, you will inside your own container instance. Now you can freely start experimenting with all the commands listed above. You will be root and can do anything inside here. You can open another terminal, ***ssh*** into the ***cs310*** server and issue the above command again to get another terminal instance inside your container. Likewise, you may enter into this container from as many terminals as you want by issuing the above command.

Useful commands:

kill -9 <PID>	**-**	Kill a process
htop -u <SOCS_USERNAME>	**-**	To list processes run by you