Lab Exercise: Containers from First Principles

Preamble

This lab exercise involves establishing namespaces using the `clone()` system call within a C programme. The programme is augmented over several iterations in order to demonstrate the characteristics of basic containers via namespaces.

If you are using Windows or OS X instead of Linux, it will be necessary to use a virtual machine in order to conduct this lab exercise. First, you will need to install Oracle VirtualBox, and then refer to the instructions in Appendix 1, before returning to the lab instructions on this page. The lab exercise requires more than one shell, so you will need to create two SSH sessions to the virtual machine (e.g. `ssh alpine@box`).

Step 1 – retrieve source code from GitHub repository

Check whether `git` is installed by typing the command `which git`, and if not, use the Linux package installer to install it. Change into a suitable working directory, and clone the git repository:

```
$ git clone https://github.com/nbrownuk/Namespaces.git
```

Change into the `Namespaces` directory, where you'll find a number of source files.

Step 2 – obtaining information on namespaces

A Linux process belongs in at least one namespace of each type supported by the running kernel. A root, global or default namespace for each namespace type exists for all processes. As with other information related to a process, the namespaces a process belongs in can be obtained via the proc filesystem. Specifically, namespace information can be found in the `/proc/<pid>/ns` directory, where `<pid>` is substituted for the actual PID in question. The directory contains a symbolic link for each namespace (e.g ipc, mnt, net, pid, user, uts), to a special, unique file that represents the namespace. The file has a name with the format `ipc:[4026531839]`, where the number is the inode of the file.

Open a terminal, and execute the command:

```
$ ls -l /proc/$$/ns
```

to identify which namespaces your command shell belongs to. Would you expect pid 1 (which is the init process) to belong to the same or different namespaces as your command shell

process? Check by executing the command:

```
$ sudo ls -l /proc/1/ns
```

Step 3 – PID namespace

The file `invoke_ns1.c` is a C programme which makes use of the `clone()` system call, which creates a child process from an existing process, and allows the programmer to specify how much of the parent's execution context the child inherits. This includes specifying that the process be cloned into a new namespace or namespaces. Take some time reading through the source code to familiarise yourself with what it does.

The programme can be compiled into an executable called `invoke_ns`, by using the command:

```
$ gcc -o invoke_ns invoke_ns1.c
```

Help relating to the command line options can be found by executing:

```
$ ./invoke_ns -h
```

The programme has two other command line options; `-v` specifies that the programme print information on `stdout` as it executes, and `-p` specifies that the cloned process be created in a new PID namespace. Optionally, a command and arguments can be provided to `invoke_ns` for execution in the child process, with the `execvp()` system call replacing the child process image with the command. The format of the `invoke_ns` command is:

```
invoke_ns [options] [cmd [arg...]]
```

After compiling `invoke_ns1.c`, execute the programme with the following:

```
$ sudo ./invoke_ns -vp env PS1="ns # " bash --norc
```

The command clones the process associated with `invoke_ns`, places the child process in a new PID namespace, and then replaces the child process image with the command shell `bash`. In verbose mode, `invoke_ns` prints to `stdout` its own (parent) pid, the pid of the child process (in the parent's namespace), and the pid of the child in its new namespace (which is pid 1). Open another terminal, and list the namespaces that the child process belongs to, with the command:

```
$ sudo ls -l /proc/<pid>/ns
```

replacing `<pid>` with the actual pid of the child process in the parent's namespace. You should see that the filename pointed to by the symbolic link `pid` is different to what you saw in step 1 – in other words, the child process is operating in a different PID namespace.

Now do the same at the bash command shell prompt of the child process in the new PID namespace:

```
$ ls -l /proc/$$/ns
```

What would you expect? What do you see instead? Why does it appear that the child process is in the root PID namespace, when we've just determined that it belongs in a new namespace?

Type `exit` to terminate `invoke_ns`.

Step 4 – MNT namespace

The reason why the child process in step 2 appears to be in the root PID namespace, is because it shares the same `/proc` filesystem as all the other processes on the system, and when you list the contents of the child process' namespace directory with `ls -l /proc/$$/ns` or `ls -l /proc/1/ns`, the kernel is returning the contents of the directory for PID 1 in the root namespace (which is the host's init process). All of the processes belong in the same MNT namespace, and therefore share the same filesystems.

In order to achieve the desired outcome, we need to put the child process in its own MNT namespace, and mount a new procfs on to `/proc` in the new namespace. The programme `invoke_ns2.c` implements some additions to the previous incarnation, which allows for specifying the `-m` command line option. This option specifies the creation of a MNT namespace for the child process, and provided the `-p` command line option has also been specified, a new procfs filesystem is mounted at `/proc` in the new MNT namespace. Review the source code in order to understand the changes introduced with `invoke_ns2.c`.

Compile the new version of `invoke_ns` using `invoke_ns2.c` as the source file, and execute the following in the command shell:

```
$ sudo ./invoke_ns -vpm env PS1="ns # " bash --norc
```

The bash command shell with a '`ns #`' prompt is invoked as the child process, in new PID and MNT namespaces.

In a different terminal, determine which namespaces the child process belongs in by issuing the command:

```
$ sudo ls -l /proc/<pid>/ns
```

replacing pid with the actual pid of the child process in the parent's namespace. Check that the namespaces for pid 1 in the `bash` command shell cloned by `invoke_ns`, are the same as

those listed for the child process in the root namespaces. Both, the `mnt` soft link and the `pid` soft link, should point to different files to those pointed to for any other process running in the root namespaces (e.g. the parent process).

Type `exit` to terminate `invoke_ns`.

Step 5 – UTS namespace

Now we will place our cloned process into a new UTS namespace, by making some changes to our programme, which can be found in `invoke_ns3.c`. The revised code introduces a new command line option, `-u`, which requires an accompanying argument – a hostname for the process in its new UTS namespace.

After you have compiled the new executable using invoke_ns3.c as the source file, execute the following command:

```
$ sudo ./invoke_ns -vpmu calculus bash
```

The `bash` command shell will be invoked in new PID, MNT and UTS namespaces, and have its prompt set to the system default format, which includes the hostname, which should be calculus as specified. Further verify the change in hostname by issuing the command:

```
$ hostname
```

and, as with the previous examples, verify the cloned process exists in new PID, MNT and UTS namespaces by examining the contents of the namespace directory under `/proc`.

Type `exit` to terminate `invoke_ns`.

Step 6 – NET namespace

Now that we've experimented with PID, MNT and UTS namespaces, it's time to introduce the NET namespace. The source file `invoke_ns4.c` contains the relevant changes to implement a NET namespace for our cloned process. The only change we require is to add a `-n` command line option, which specifies that the `CLONE_NEWNET` flag is passed to the `clone()` system call. Familiarise yourself with the changes made in the source code.

After you have compiled the new executable using `invoke_ns4.c` as the source file, execute the following command in order to clone our process into a new NET namespace:

```
$ sudo ./invoke_ns -vpmnu calculus bash
```

First of all, satisfy yourself that the cloned `bash` command shell has been invoked in a new NET namespace.

The new NET namespace will have been created with a completely new, isolated network stack. List the network interfaces available in the new NET namespace by executing the command:

```
$ ip link show
```

at the cloned bash command prompt. The creation of the new NET namespace has resulted in a single network interface, the loopback interface (`lo`), whose status is 'down'. Change the status by issuing the command:

```
$ ip link set lo up
```

In order to allow the contained process to communicate beyond the confines of its NET namespace, we need to arrange some things in the root NET namespace.

We need to create a tunnel between the the two namespaces, using a 'veth pair', where veth is shorthand for virtual ethernet. In a command shell running in the root NET namespace, execute the following command:

```
$ sudo ip link add host type veth peer name guest
```

This will create two, connnected veth interfaces called `host` and `guest`. You can see these with the command:

```
$ ip link show
```

The `guest` interface needs to be moved from the root NET namespace to our newly created NET namespace. This can be achieved with the command:

```
$ sudo ip link set guest netns <pid>
```

where <pid> should be replaced with the pid of the cloned bash command shell in the root NET namespace. Verify that the `guest` end of the veth pair now resides in our newly created NET namespace, by executing `ip link show` in both NET namespaces.

Finally, we can assign an IP address to each end of the veth pair to demonstrate communication between the NET namespaces. We need to do this from the bash command shell for each respective NET namespace. From the `bash` command shell in the root NET namespace, assign an IP address to the `host` end:

```
$ sudo ip addr add 10.0.0.1/24 dev host
```

Verify the interface is configured correctly:

```
$ ip addr show host
```

Now provide an IP address for the guest end of the veth pair inside our created NET namespace:

```
# ip addr add 10.0.0.2/24 dev guest
```

Verify the interface is configured correctly:

```
# ip addr show guest
```

Check that both interfaces are up (bring them up if necessary, using `ip link set <guest|host> up`), and then ping both ends of the veth pair from the opposite end to verify the link between the NET namespaces.

Type `exit` to terminate `invoke_ns`.

Step 7 – IPC namespace

The Linux kernel IPC namespace supports the isolation of System V IPC objects and POSIX message queues. In order to demonstrate the function of the IPC namespace, we'll develop our programme to implement a POSIX message queue.

The updated source code introduces a new command line option, `-i`, which must take as an argument, either `no` or `yes`. In either case, when the `-i` option is specified, a message queue is created. If `no` is specified in conjunction with the `-i` command line option, the parent clones a new process, but in the same IPC namespace as itself, which will enable both processes to access the message queue. If `yes` is specified in conjunction with `-i`, however, then the cloned child is created in a new IPC namespace, and therefore does NOT have access to the message queue.

The message queue is created by the parent process before it clones the child process, and then waits for 60 seconds to receive a message from the child process via the message queue, before giving up. The child process opens the same message queue, and prompts the user to enter a message to send to the parent process. Review the source code changes made in the file `invoke_ns5.c`, before compiling the new version of `invoke_ns` with the `librt` library:

```
$ gcc -o invoke_ns invoke_ns5.c -lrt
```

Execute the following command, taking note of the messages each part of the programme outputs:

```
$ sudo ./invoke_ns -vi no
```

Where the programme requests a message to be supplied, type a short (up to 60 characters)

message at the prompt, followed by RETURN. Notice that the parent process receives the message before closing and shutting down the message queue.

What do you expect to happen if we now execute `invoke_ns`, and clone the child process in a new IPC namespace? Execute the following command to test your thinking:

```
$ sudo ./invoke_ns -vi yes
```

When the child process attempts to open the message queue, it fails with a `No such file or directory` error message, and the parent process eventually times out waiting for a message. This is because the message queue exists in the root namespace, but not the new namespace associated with the cloned child process.

Step 8 – jail the process

The final step in creating our very basic container, is to jail the process inside a segment of the host filesystem, with a set of its own files. To do this, we will use the `chroot()` system call. We will make a minimal environment just to demonstrate the capability, although minimal container environments are the norm for Docker anyhow. A `-c` command line option has been added to our programme, `invoke_ns6.c`, which takes a mandatory, single argument – the path to the chroot jail. Review the changes to the source code to familiarise yourself with the process of creating the chroot jail, and then compile it:

```
$ gcc -o invoke_ns invoke_ns6.c -lrt
```

First, we need to prepare an area of the host filesystem to contain the process, and we're going to provide the following executables; `env`, `bash`, `ps`, `ls` and `top` within that environment. As well as copying the executables into our jail, we also need to provide the shared libraries that these executables rely on. We can use the `ldd` executable to determine what shared libraries each executable uses. A script, `binlibdepcp.sh`, has been prepared to take care of copying the relevant binary and shared libraries to the appropriate location.

Make a directory to act as the rootfs for the container:

```
$ sudo mkdir -p /var/local/jail
```

Then, for each of the executables listed above, execute the script:

```
$ sudo ./binlibdepcp.sh /usr/bin/env /var/local/jail
```

Do the same for `/bin/bash`, `/bin/ps`, `/bin/ls` and `/usr/bin/top`. The paths to some of these executables may not be in the locations indicated, but you can check their locations using the `which` command, e.g.:

```
$ which top
```

In order that the terminal that is used within the container functions correctly, we also need to include a component of the terminfo database. If you're terminal is an `xterm`, then we need to provide that part of the database for the chroot jail. Depending on the Linux distribution you use, the terminfo database may be located in `/usr/lib` or `/usr/share` instead of `/lib`, and you should adjust the source and destination accordingly:

```
$ sudo mkdir -p /var/local/jail/lib/terminfo/x
$ sudo cp -p /lib/terminfo/x/* /var/local/jail/lib/terminfo/x
```

You're now ready to test the revised version of `invoke_ns`, by executing the following:

```
$ sudo ./invoke_ns -vpmu calculus -c /var/local/jail \
env PS1="\[\e[34m\]\h\[\e[m\] [\[\e[31m\]\W\[\e[m\]] " bash --norc
```

Use the `ls`, `ps` and `top` commands to verify that the cloned child process is contained within its own PID, MNT and UTS namespaces, and that it has a very limited filesystem in which to operate.

Type `exit` to terminate `invoke_ns`.