Programming Assignment 3: Build your own container using Linux namespaces and cgroups

Back to CS695 home page

Introduction

In this assignment, you will understand how containers work by building one from scratch yourself. You will also experiment with existing container frameworks.

Before you begin, you are expected to have a good understanding of Linux namespaces and cgroups as studied in class. You will also benefit from reading helpful articles online on how to build containers from scratch, like **this tutorial.** Every container needs a good root filesystem. The easiest way is to download one online, e.g., from this link that is provided in the previous tutorial. If you wish to build a root filesystem yourself, you can find some great resources online like this talk. This talk demonstrates several tools to build a root filesystem, including Buildroot.

If you have never used containers before, it is a good idea to install and run one or more popular container frameworks like LXC and Docker, to understand the behavior of containers.

Part A: Build your own container runtime

In this part of the assignment, you will write a C/C++ program which will a spawn a command shell in a container-like isolated environment. The following are the specifications of this container shell:

- The shell must run in separate **network**, **mount**, **PID** and **UTS** namespaces. You can achieve this in one of two ways: you can either create these namespaces beforehand and pass suitable arguments to your program to join the newly created namespaces, or you can create the namespaces directly from within your program. You can use some subset of the `clone`, `setns`, `unshare` system calls or their commandline wrappers. You can use the `ip netns` shell commands to create network namespaces. Supporting **user** and **IPC** namespaces is *optional*.
- You must pass a root filesystem as an argument to your program, and your shell must begin execution in the top-level root directory of this root filesystem. That is, running `ls` in the shell must show the files in the root of your new root filesystem.
- You must pass a new hostname as an argument to your program, and the shell must display this hostname upon running the `hostname` command.
- Running the `ps` command in the shell of your container must display a view of processes in the new PID namespace. You can achieve this by mounting a new proc filesystem in the container.
- You must be able to demonstrate network connectivity between your container's namespace and the default/parent namespace by running a client server application across namespaces. You can choose any client-server

application of your choice. The server must be started from the shell of your container, and a client running in the parent namespace should be able to successfully connect to this server and exchange information. The two namespaces must run with different IP addresses, and connectivity between these namespaces must be suitably configured via a **veth** device pair. You must also ensure that the server code/executable is part of your root filesystem, so that it can be executed from your container's shell.

- You must configure limits on any one resource (e.g., CPU, memory) used by this container via Linux **cgroups**. You must also run a suitable program from the shell to demonstrate that the limits are being enforced. For example, if you set a limit on the maximum memory that can be used by your container, you must show that a memory-hungry application run from the shell cannot consume more memory beyond the configured limit.
- You must be able to run multiple of such container shells, and show that they are isolated from one another. For example, the two separate containers must be able to start servers listening on the same port numbers (something you cannot do with regular Linux shell). Further, you must show that the processes in the two containers are not aware of each other using the output of the `ps` command.
- Your program must not invoke an existing container runtime like LXC, but must accomplish the above by directly invoking the cgroups and namespaces functionality of the Linux kernel.

You must design a suitable demo for your shell, where you will run tests to demonstrate all of the above requirements, using commands of your choice. For example, you should start processes and demonstrate the `ps` command. You must create resource-hungry applications to demonstrate resource limits, and so on. You are responsible for writing code for all of these testcases as well.

Part B: Containers vs. VMs (ungraded, optional)

In this part of the assignment, you will compare the performance of VMs and containers in terms of their virtualization overhead. You can use any container runtime of your choice (e.g., LXC, Docker) and any VM platform or your choice (e.g., QEMU/KVM). You can either compare the performance of an application when it is running inside a VM vs. inside a container, or you can compare the startup latency or boot times of containers vs. VMs. That is, the performance metric that is being compared across containers and VMs can be chosen by you.

If you choose to compare application performance, choose any application of your choice and run it both within a container and the VM. Be careful to allocate similar resources (CPU, memory etc.) to both the VM and the container. Measure the capacity or saturation performance (in terms of throughput or requests processed per second, for example) of your application in both cases. Usually, the saturation performance of the application occurs when some hardware resource (e.g., CPU or network) is saturated and running at full utilization. Compare the saturation capacity of the application when it is running inside a container vs. when run inside a VM, and draw suitable conclusions. Also compare the container/VM performance with baremetal performance, i.e., when the application is directly running on the host without any virtualization. You may want to experiment with different kinds of applications in different scenarios to reach a well-informed conclusion on the relative performance overheads of VMs vs. containers.

Alternately, instead of comparing application performance, you may also compare VMs and containers along other axes such as memory footprint, bootup time, and so on. Start a large number of containers or VMs in sequence, and measure how long it takes to boot them. Analyze the difference in startup latencies, and explain them correctly.

This part of the assignment is open-ended, and there is no one correct answer. For example, the relative performance of the application in containers vs. VMs will depend on the application you choose and the workload you run it with. What we are expecting from you is a careful measurement of performance in both scenarios, and an analysis/explanation of the results you observe.

Submission and grading

You must solve this assignment individually. You are required to submit code (to create the container, as well as the test cases) for only part A. Create a tar-gzipped file of your submission, with the filename being your roll number, and upload it on Moodle. Grading this assignment will involve a demo/viva with the instructor/TA.

Good luck!

Back to CS695 home page