MYDOCKER: Implementation of Containerization Using Linux Kernel Features

Ostap Seryvko*, Sofiia Folvarochna*, Olesia Omelchuk*, Anastasiia Beheni*, Hermann Yavorskyi*
* *Faculty of Applied Sciences, Ukrainian Catholic University, Lviv, Ukraine*
{ostap.seryvko, sofiia.folvarochna, olesia.omelchuk, anastasiia.beheni, yavorskyi}@ucu.edu.ua

***Abstract*—This project demonstrates the implementation of the Docker analogue using Linux kernel features.**

For more details, see: https://github.com/redn1nja/mydocker

***Index Terms*—Docker · containers · containerization · cgroups · namespaces · Linux**

I. Introduction

Docker is software that enables the isolated execution of processes within containers. As a paradigm, containerization encapsulates applications and their dependencies within isolated environments. It resolves the challenge of constraining processes in resources and accelerates the deployment of the required environment for a program, enhancing its portability across different machines.

The advantages of this approach are evident. The previously mentioned portability to other machines increases convenience and accelerates development speed. Additionally, departing from portability, containers offer greater control possibilities by allowing resource limitations to be defined. They also ensure isolation from other processes on the machine, preventing conflicts. Another advantage is that containers are lighter than virtual machines.

According to the latest data, Docker has over 15 million [1] users.

This paper delves into implementing this Docker-like containerization concept, leveraging features within the Linux kernel features.

II. Docker Overview

Docker is a platform for developing, shipping and running applications in containers.

- **Docker Engine** is responsible for building and running containers. It consists of a daemon process (dockerd) and a Command Line Interface (CLI) for interacting with the daemon.
- **Images** are the blueprints for containers. They contain everything needed to run an application, including the code, runtime, libraries, and dependencies.
- **Containers** are running instances of Docker images.
- **Dockerfile** is a script that defines the steps to build a Docker image. It specifies the base image, sets up the environment, and includes any necessary configurations.
- **Registry** is a repository for Docker images, allowing users to share and distribute their images.

III. Used Linux Kernel Features

Linux kernel contains various features used for memory management, filesystem handling, process supervision, device driver management, and networking support.

For implementing Mydocker we used the following Linux kernel attributes:

A. Namespaces

Namespaces [2] envelop a global system resource in a way that creates the illusion that processes within the namespace possess their own independent instance of the global resource. Other processes within the namespace can observe changes to the global resource while remaining isolated from processes outside the namespace. There are many different namespaces, which correspond to different crucial resources: IPC, filesystem, network and others.

Most namespaces when they're 'unshared' from global recource, possess an illusion of full resource capacity. For example, an process which is unshared in the PID namespace, considers itself of a process with PID 1, however, it will have other PID in the global process tree. But not all namespaces behave simirally. Opposite to latter, mount namespaces, which are the most important part of containarization process in terms of isolation of resources, don't create a 'clean' copy of mount points, as it is impossible to not have any mount points. Therefore, when a mount namespace is created, a copy of mount history is initialized, which in fact, can be easily changed after, which will not affect global mount history.

In order to create an isolated filesystem, a following algorithm is considered:

- Create an unshared process in the mount namespace using *clone(2)*.
- Create a mountpoint for a root, which will be hidden from the global mount history and will contain the new root filesystem of the container.
- Use *chroot(2)* or *pivot_root(2)* system calls to change the root.
- Run the executable, which must be inside the new root filesystem, as the old one is inaccesible.

B. Cgroups

Cgroups [3] is short for control groups. It's a mechanism that allows processes to be organized into hierarchical groups that can use different types of resources. These resources can be monitored and controlled for each such group. Cgroups

also allow prioritizing groups in resource allocation. Interface to cgroups is provided through a pseudo-filesystem called cgroupfs.

Mydocker can limit its containers in memory, number of processes and CPU share – the most crucial recourses.

C. Volumes/Bind Mounts

Despite of having an isolated filesystem using mount namespaces, it may be helpful to access the data outside of the container. Docker uses two different ways of achieving this, called volumes and bind mounts, respectively. In fact, volumes are improved version of bind mounts and bind mounts can be done using *mount(2)* system call, with the **MS_BIND** flag. Mydocker allows to create bind mounts, which should be listed in the "mount_points" category of mydockerfile.

IV. Implementation Details

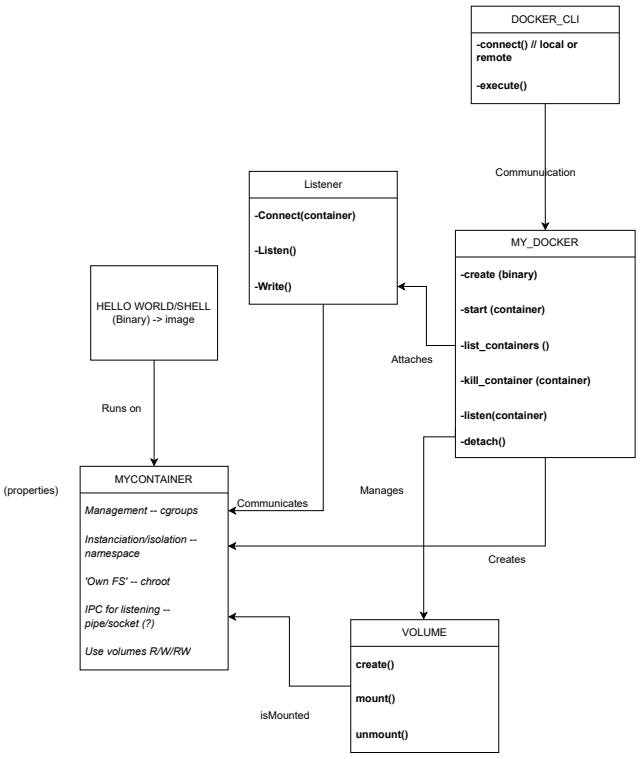

Fig. 1. UML diagram of our classes

A. Mydocker Daemon

Daemon is the main executable of the project, which does all the housekeeping job. It is organized as a network server, which can be connected by a client using sockets. The main data structure of the daemon is the MyDocker class, which contains, most importantly, the list of MyDockerContainers, which will be discussed later. Daemon accepts commands from the network socket and executes them with respect to arguments provided in the command.

B. Mydocker container

Containers are basically isolated applications. To create a container it is required to setup all parameters that provide containerization, e.g. namespaces and control groups and then run application. Then to have control of a container, it may be useful to store the data about the process and utilize using the client CLI.

C. Mydockerfile

Dockerfiles are rulesets which allow to control the flow of container creation. It is a configuration file, which provides the information of a future container such as: application name, mounts, resource limitations.

Listing 1. Mydockerfile example

```
{
  "bin" : "/bin/sh",
  "args" : [],
  "mount_points" : ["/tmp"],
  "mem_limit": 200,
  "pids_limit": 20
}
```

D. Mydocker CLI

CLI is a convenient way of controlling the flow of a daemon and containers. In synergy with MyDocker daemon, it is a tool, which allows to create, destroy, run and manipulate containers, including the possibility to attach the output (and input, if needed) of container to CLI's I/O.

V. Challenges and Obstacles

The main challenge of the whole project was to isolate containers as much as possible and, at the same time, leave the opportunity for convenient communication with the "outer world" (mounts and sockets played the biggest role here).

VI. Conclusions

However, Mydocker's functionality is far from the original Docker; it has accomplished an important didactic role.

References

[1] S. Johnston, "Celebrating our second fiscal year," Feb 2022. [Online]. Available: https://www.docker.com/blog/celebrating-our-second-fiscal-year/

[2] *Namespaces Manual Page*, Linux Documentation Project, Available at https://man7.org/linux/man-pages/man7/namespaces.7.html.

[3] *Cgroups Manual Page*, Linux Documentation Project, Available at https://man7.org/linux/man-pages/man7/cgroups.7.html.