Minimal Container Runtime (mdocker)

Project Report

1. Introduction

Modern container platforms like Docker rely on Linux kernel features such as namespaces, cgroups, and filesystem isolation to run applications in lightweight, isolated environments. This project, **mdocker**, is a minimal container runtime implemented in C to demonstrate how containers work internally. The project provides process isolation, resource limiting, filesystem virtualization, and networking using low-level Linux system calls.

2. Objectives

The main objectives of this project are:

- To understand and implement Linux namespaces for isolation
- To enforce resource limits using Linux cgroups (v2)
- To provide a container-specific filesystem using OverlayFS
- To enable basic networking for containers using network namespaces and veth pairs
- To execute arbitrary commands inside an isolated container environment

3. Software Requirements

- Linux kernel with support for:
  - Namespaces
  - cgroups v2
  - OverlayFS
- Root privileges (sudo)
- GCC compiler
- iproute2 and iptables utilities

4. High-Level Architecture

The project follows a **parent–child execution model**:

1. The **parent process**:
   - Sets up OverlayFS
   - Creates cgroups and applies resource limits

   - Configures networking on the host side
   - Creates a child process using clone() with new namespaces
2. The **child process**:
   - Sets up mount points
   - Switches root filesystem using pivot_root
   - Configures container-side networking
   - Executes the user-specified command

5. Key Linux Features Used

5.1 Namespaces

The following namespaces are used for isolation:

- **PID Namespace**: Isolates process IDs
- **UTS Namespace**: Allows a custom hostname inside the container
- **Mount Namespace**: Isolates filesystem mount points
- **Network Namespace**: Provides a private network stack

These namespaces are created using the clone() system call.

5.2 Control Groups (cgroups v2)

Cgroups are used to limit resource usage of the container:

- **Memory limit**: Restricts maximum memory usage
- **CPU limit**: Restricts CPU usage percentage
- **PID limit**: Restricts number of processes

The container process is added to a custom cgroup under /sys/fs/cgroup/.

5.3 OverlayFS

OverlayFS is used to provide a writable container filesystem without modifying the base root filesystem.

- **Lower directory**: Read-only base filesystem (./rootfs)
- **Upper directory**: Writable layer per container
- **Work directory**: Required by OverlayFS
- **Merged directory**: Final view presented to the container

This allows multiple containers to share a common base filesystem.

5.4 Networking

Each container gets isolated networking using:

- Network namespaces
- Virtual Ethernet (veth) pairs
- IP forwarding and NAT using iptables

Network Setup:

- Host IP: 10.200.1.1
- Container IP: 10.200.1.2
- Subnet: 10.200.1.0/24

This enables containers to access the internet through the host.

6. Detailed Module Description

6.1 Parent Process (parent_main)

Responsibilities:

- Validates root permissions
- Sets up OverlayFS
- Creates cgroups and applies limits
- Creates synchronization pipe
- Creates container using clone()
- Configures host-side networking
- Waits for container termination
- Performs cleanup

6.2 Clone Callback (clone_callback)

- Waits until parent finishes setup
- Re-executes the binary in child mode
- Ensures proper namespace initialization

6.3 Child Process (child_main)

Responsibilities:

- Makes mounts private
- Performs pivot_root to switch filesystem
- Mounts /proc, /sys, and /dev
- Sets hostname
- Configures container-side networking
- Executes the target command

- Handles signal forwarding and process reaping

6.4 Signal Handling

Signals such as SIGINT and SIGTERM are forwarded from the container init process to the actual application process to ensure graceful termination.

7. Execution Flow

1. User runs:
2. sudo ./mdocker run /bin/bash
3. Parent process initializes container resources
4. Child process enters new namespaces
5. Filesystem is isolated using OverlayFS
6. Networking is configured
7. Command executes inside the container
8. On exit, all resources are cleaned up

8. Future Enhancements

- Add user namespaces for rootless containers
- Support container images (OCI format)
- Improve networking with bridges

9. Conclusion

This project successfully demonstrates the internal working of containers using core Linux primitives. By building a minimal container runtime from scratch, it provides deep insight into how Docker-like systems achieve isolation, resource control, and portability. The project serves as a strong foundation for further exploration into container security, orchestration, and runtime development.