Docker-lite Project: From Basics to Complete Container Management

---

Table of Contents

Overview

This Project explores container fundamentals through hands-on implementation of container primitives, tools, and management systems. You'll build understanding from low-level system calls to complete container orchestration tools similar to Docker.

Docker-lite Project Structure

- **Task 1**: Namespace manipulation with system calls
- **Task 2**: CLI-based container creation
- **Task 3**: Full-featured container management tool
- **Task 4**: Multi-container service deployment

Environment Setup

System Requirements

This Project requires specific Virtual Machine environments using Debian:

x86_64 Systems (Linux/Windows/Intel Mac)

- **Virtualization Tool**: VirtualBox

- **VM Credentials**:
  - Username: `cs695`
  - Password: `1234`
  - Root Password: `1234`

Apple Silicon Systems

- **Virtualization Tool**: UTM

Project Structure

```
<student_id>_assignment3/
├── task1/
│   └── namespace_prog.c
├── task2/
│   └── simple_container.sh
├── task3/
│   ├── config.sh
│   └── conductor.sh
└── task4/
    └── deployment_script.sh
```

Task 1: Namespace System Calls

Objective

Explore Linux namespace system calls to create isolated process environments.

Requirements

Implement namespace manipulation using system calls:

1. **Create Child Process 1**: New UTS and PID namespace
2. **Create Child Process 2**: Attach to Child 1's namespaces

Key System Calls

- `clone()` - Create new process with namespace flags
- `setns()` - Join existing namespace
- `unshare()` - Create new namespace for current process
- `pidfd_open()` - Get file descriptor for process

Implementation File

```c
// task1/namespace_prog.c
// Modify only marked sections
```

Expected Output

```
----------------------------------------
Parent Process PID: 43156
Parent Hostname: cs695
----------------------------------------
Child1 Process PID: 1
Child1 Hostname: Child1Hostname
----------------------------------------
Parent Process PID: 43156
Parent Hostname: cs695
----------------------------------------
Child2 Process PID: 2
Child2 Hostname: Child1Hostname
----------------------------------------
Parent Process PID: 43156
Parent Hostname: cs695
----------------------------------------
```

Reference Documentation

- `man 7 pid_namespaces`
- `man 2 setns`
- `man 2 clone`
- `man 2 pidfd_open`
- `man 2 unshare`

Task 2: CLI Container Implementation

Objective

Create simplified containers using command-line tools with progressive feature addition.

Implementation File

```bash
```

```
# task2/simple_container.sh
# Complete marked sections only
```

Subtask 2a: Filesystem Isolation

Goal: Implement `chroot` for filesystem isolation

Requirements:

- Use `container_root` directory as container root
- Copy all required dependencies for `container_prog`
- Handle dynamic library dependencies correctly

Key Concepts:

- `chroot` command usage
- Dynamic library dependency resolution
- Filesystem isolation principles

Subtask 2b: Process and Network Isolation

Goal: Add PID and UTS namespace isolation

Requirements:

- Container processes start with PID 1
- Separate UTS namespace for hostname isolation
- Use `unshare` command with appropriate flags

Expected Changes:

- Process PID changes from host PID to container PID 1
- Hostname isolation enabled

Subtask 2c: Resource Control

Goal: Implement CPU usage limits using cgroups v2

Requirements:

- Limit CPU usage to 50% of single core
- Use cgroup v2 quota/period mechanism
- Apply limits to all container processes

Key Concepts:

- Cgroup v2 hierarchy
- CPU quota and period configuration
- Process group management

Sample Outputs

Subtask 2a Output

```
Process PID: 1693
Child Process PID: 1694
- - - - - - - - - - - - - - - -
Files/Directories in root directory:
lib64
container_prog
.
..
lib
- - - - - - - - - - - - - - - -
________________________________________________
```

Subtask 2b Output

```
Process PID: 1
Child Process PID: 2
- - - - - - - - - - - - - - - -
Files/Directories in root directory:
lib64
container_prog
.
..
lib
- - - - - - - - - - - - - - - -
Hostname within container: new_hostname
Computation Benchmark:
Time Taken: 1440778 ms
Value (Ignore): 107375219085276240
Hostname in the host: cs695
________________________________________________
```

Subtask 2c Output

```
Process PID: 1
Child Process PID: 2
---------------
Files/Directories in root directory:
lib64
container_prog
.
..
lib
---------------
Computation Benchmark:
Time Taken: 2866732 ms
Value (Ignore): 107375219085276240
```

Task 3: Advanced Container Management Tool

Objective

Develop a comprehensive container management system with Docker-like capabilities.

Prerequisites

```bash
sudo apt install debootstrap iptables
```

Configuration

Edit `config.sh` to set `DEFAULT_IFC` to your VM's network interface:

```bash
# Use 'ip a' to find your interface name
DEFAULT_IFC="enp1s0"  # Example interface name
```

Tool Features

Core Commands

Build Image

```bash
./conductor.sh build <image-name>
```

- Creates Debian system using debootstrap
- Stores in configured images directory

List Images

```bash
./conductor.sh images
```

- Shows all available container images

Remove Image

```bash
./conductor.sh rmi <image-name>
```

- Deletes specified image

Run Container

```bash
./conductor.sh run <image-name> <container-name> -- [command <args>]
```

- Starts new container from image
- Isolated UTS, PID, NET, MOUNT, IPC namespaces
- Mounts procfs, sysfs, /dev
- Default command: `/bin/bash`

List Containers

```bash
./conductor.sh ps
```

- Shows all running containers

Stop Container

```bash
./conductor.sh stop <container-name>
```

- Terminates container and cleans up resources

Execute in Container

```bash
./conductor.sh exec <container-name> -- [command <args>]
```

- Runs command in existing container namespace

Network Configuration

```bash
./conductor.sh addnetwork <container-name> [options]
```

Network Options:

- Default: Basic host-container communication
- `-i, --internet`: Enable internet access via NAT
- `-e, --expose <host-port>:<container-port>`: Port forwarding

Container Peering

```bash
./conductor.sh peer <container1-name> <container2-name>
```

- Enables inter-container communication

Implementation Subtasks

Subtask 3a: Implement `run` Command

- Use `unshare` and `chroot` for container isolation
- Mount required filesystems (proc, sys, dev)
- Ensure tools like `ps`, `top` work correctly

Subtask 3b: Implement `exec` Command

- Join all container namespaces (UTS, PID, NET, MOUNT, IPC)
- Execute commands in container context
- Maintain proper filesystem visibility

Subtask 3c: Implement Networking

- Create veth pair for host-container communication

- Configure interfaces and enable them
- Set up proper network connectivity

Sample Usage

Basic Container Operations

```bash
# List images
sudo ./conductor.sh images

# Run container
sudo ./conductor.sh run mydebian eg

# In container
root@cs695:/# ps
PID TTY     TIME CMD
  1 ?     00:00:00 bash
  3 ?     00:00:00 ps

# Exit and stop
root@cs695:/# exit
sudo ./conductor.sh stop eg
```

Network Testing

```bash
# Run container
sudo ./conductor.sh run mydebian eg

# In another terminal, add networking
sudo ./conductor.sh addnetwork eg

# In container, test connectivity
root@cs695:/# ip a
root@cs695:/# ping 192.168.1.1
```

Task 4: Multi-Container Service Deployment

Objective

Deploy interconnected services across multiple containers to demonstrate practical container orchestration.

Architecture

- **Container 1 (c1)**: External service (accessible from outside)
- **Container 2 (c2)**: Counter service (internal only)
- **Networking**: c1 ↔ Internet, c1 ↔ c2, c2 ↔ Internet (no external ports)

Implementation Steps

1. **Build Container Image**

```bash
sudo ./conductor.sh build service-image
```

2. **Launch Background Containers**

```bash
sudo ./conductor.sh run service-image c1 -- sleep infinity
sudo ./conductor.sh run service-image c2 -- sleep infinity
```

3. **Copy Service Files**

```bash
# Copy external-service to c1
cp -r external-service .containers/c1/rootfs/

# Copy counter-service to c2
cp -r counter-service .containers/c2/rootfs/
```

4. **Configure Networking**

```bash
# c1: Internet access + port forwarding (3000 host → 8080 container)
sudo ./conductor.sh addnetwork c1 --internet --expose 3000:8080

# c2: Internet access only
sudo ./conductor.sh addnetwork c2 --internet

# Enable c1 ↔ c2 communication
sudo ./conductor.sh peer c1 c2
```

5. **Get c2 IP Address**

```bash
C2_IP=$(sudo ./conductor.sh exec c2 -- ip route get 1 | grep -oP 'src \K\S+')
```

6. **Launch Services**

```bash
# Start counter service in c2
sudo ./conductor.sh exec c2 -- bash /counter-service/run.sh

# Start external service in c1 (connects to c2)
sudo ./conductor.sh exec c1 -- bash /external-service/run.sh "http://$C2_IP:8080/"
```

7. **Test Deployment**

```bash
# From host system
curl http://<host-ip>:3000

# From external system
curl http://<host-ip>:3000
```

Network Architecture Diagram

```
┌─────────────────────────────────────────────────────────────────┐
┌──────────────────────────────────┐
│Host                │  │Container c1  │
│                  │  │(external-svc) │
├────────────────────┐  ┌──────────────────────────────────┐  │  ├──────────────────────────────┐  │
│default  ││c1-outside  ├───┼───────┤ c1-inside  │ │
│iface  ││(NAT)       │ │  │:8080     │ │
├────────────────────┘  └──────────────────────────────────┘  │  └──────────────────────────────┘  │
│       Port 3000 → 8080  │  │          │
└─────────────────────────────────────────────────────────────────┘
└──────────────────────────────────┘
                        │
                        │Peer Link
                        │
┌─────────────────────────────────────────────────────────────────┐
┌──────────────────────────────────┐
│Host                │  │Container c2  │
│                  │  │(counter-svc)  │
├────────────────────┐  ┌──────────────────────────────────┐  │  ├──────────────────────────────┐  │
│default  ││c2-outside  ├───┼───────┤ c2-inside  │ │
│iface  ││(NAT only)    │ │  │:8080     │ │
├────────────────────┘  └──────────────────────────────────┘  │  └──────────────────────────────┘  │
│       No port forwarding │  │          │
└─────────────────────────────────────────────────────────────────┘
└──────────────────────────────────┘
```

Troubleshooting

Common Issues

Internet Connectivity in VM

- Ensure VM network adapter is properly configured
- For campus networks, use appropriate login scripts
- Check firewall settings

Container Networking Problems

- Verify `DEFAULT_IFC` setting in config.sh
- Check iptables rules for conflicts
- Ensure kernel supports required namespaces

Port Access Issues

- Disable host firewall or configure exceptions
- For NAT networks, configure port forwarding in VM settings
- Use VM IP address, not localhost, for external access

Clean Reset

```bash
# Stop all containers
sudo ./conductor.sh ps | grep -v "No containers" | while read name date; do
  sudo ./conductor.sh stop "$name"
done

# Remove all images
sudo ./conductor.sh images | tail -n +3 | while read name size date; do
  sudo ./conductor.sh rmi "$name"
done
```

Reference Resources

Linux Namespaces

- [LWN Namespaces Articles]
- [Linux Namespaces Manual]

Container Technologies

- chroot Command Guide
- unshare Manual
- cgroup v2 Documentation

Networking

- Linux Network Namespaces
- veth Pair Configuration
- iptables NAT Configuration

Submission Guidelines

File Structure Verification

Ensure your submission follows the exact directory structure:

```
<student_id>_assignment3/
├── task1/namespace_prog.c
├── task2/simple_container.sh
├── task3/config.sh
├── task3/conductor.sh
└── task4/deployment_script.sh
```

Pre-Submission Checklist

- [ ] All tasks compile and run successfully
- [ ] Only marked sections modified in provided files
- [ ] No hardcoded outputs or system-specific paths
- [ ] All required dependencies documented
- [ ] Network configuration properly abstracted
- [ ] Clean code with appropriate comments

Testing Your Implementation

1. Test each task independently
2. Verify all expected outputs match specifications
3. Test edge cases and error conditions
4. Ensure cleanup procedures work correctly
5. Validate on fresh VM environment

Learning Objectives

By completing this Project, you will understand:

- **System-level container implementation** using Linux primitives
- **Namespace isolation** for processes, filesystems, and networks
- **Resource management** through cgroups
- **Container networking** including NAT, port forwarding, and inter-container communication
- **Service orchestration** across multiple containers
- **Real-world container architecture** similar to Docker/Kubernetes

This foundation prepares you for understanding and working with production container platforms and cloud-native architectures.