Operating Systems Project: Topic 4

Lightweight Container Implementation with Namespaces

Liangsen Wang

224040364@link.cuhk.edu.cn

2026.1.15

Outline

Outline

The Mission: Basic Requirements (Mandatory)

Objective: Build a "Mini-Docker" using raw Linux System Calls.

Requirement 1: The Core (Implementation)

- **Constraint:** No Docker/Runc. Pure C code interacting with Kernel.
- **Mandatory Namespaces:** PID, MNT, UTS, NET, IPC.
- **Key Syscall:** `clone()` with flags.

Requirement 2: The Proof (Verification)

- **Process View:** `ps aux` inside must ONLY show container processes.
- **Identity:** `hostname` must be independent.

Advanced Options

Option A: Resource Control (Cgroups)

- **Goal:** Limit CPU or Memory usage (e.g., Max 100MB RAM).
- **Mechanism:** Manipulate `/sys/fs/cgroup/` (cgroup v1 or v2).
- **Test:** Launch a memory hog (malloc loop) inside the container and verify it gets killed (OOM) without crashing the host.

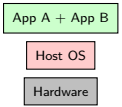
Option B: Image Management (Rootfs)

- **Goal:** Make the container look like a real OS (e.g., Alpine Linux).
- **Mechanism:** Use `pivot_root` (secure) or `chroot` to switch the root directory to an extracted tarball.

Outline

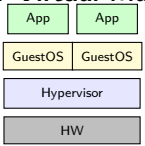
Theory 1: The Evolution of Deployment

Era 1: Bare Metal



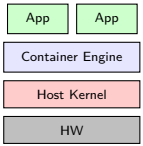
- **Issue:** Dependency Hell. App A needs Lib v1, App B needs Lib v2. Conflict!

Era 2: Virtual Machines



- **Issue:** Heavy. Each App carries a full OS kernel (GBs of RAM).

Era 3: Containers



- **Solution:** Shared Kernel. Lightweight isolation via Namespaces.

Theory 2: The Container Architecture Stack

"Docker" is actually a stack of tools. Where does your project fit?

1. **High Level (Docker CLI / Kubernetes):** Orchestration, Image downloading.
2. **Container Runtime (containerd):** Manages lifecycle.
3. **Low-Level Runtime (runc): <– THIS IS YOUR PROJECT!**
   - Specifically talks to the Kernel.
   - Sets up Namespaces and Cgroups.
   - Execs the user process.
4. **Linux Kernel:** Provides the primitives (`clone`, `cgroups`).

*You are essentially building a simplified version of **runc**.*

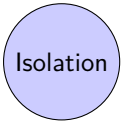
Outline

Mechanics 1: The Three Pillars of Containers

A container is not a single feature. It is the combination of three kernel primitives:

1. Namespaces

"What you can SEE"



- Isolation of View.
- Example: "I am PID 1."

2. Cgroups

"What you can USE"

- Limitation of Resources.
- Example: "Max 1GB RAM."

3. UnionFS / Chroot

"Where you LIVE"

- File System Jail.
- Example: "My root is here."

Mechanics 2: Deep Dive into Namespaces (The "Matrix")

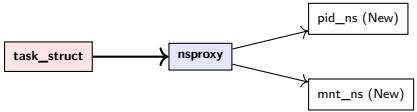
Namespaces partition global system resources.

Namespace	Flag	Effect
PID	`CLONE_NEWPID`	Processes inside get their own PIDs starting from 1. Essential for `init`-like behavior.
Mount	`CLONE_NEWNS`	Private list of mount points. Mounting `/proc` here doesn't affect the host.
UTS	`CLONE_NEWUTS`	Independent Hostname and Domain name.
Network	`CLONE_NEWNET`	Private Loopback, IP, Ports, Routing Table.
IPC	`CLONE_NEWIPC`	Private Shared Memory segments.

Mechanics 3: Kernel Implementation (`nsproxy`)

How does the kernel track this? Through `struct nsproxy`.

- Every process (`task_struct`) points to an `nsproxy`.
- **Normal Process:** Points to the **same** nsproxy as init.
- **Container Process:** Points to a **new** nsproxy with new Namespace pointers.



Project Hint: You rely on the kernel to manage this. You just set the flags in `clone()`.

Mechanics 4: Control Groups (Cgroups)

(Advanced Requirement) Namespaces hide processes, but they don't stop a container from using 100% CPU.

What are Cgroups?

A kernel feature that organizes processes into hierarchical groups and limits/monitors their resource usage.

- **Interface:** The Virtual Filesystem at `/sys/fs/cgroup/`.
- **How to use (Logic):**
  1. `mkdir /sys/fs/cgroup/memory/my_container`
  2. `echo 100M > .../memory.limit_in_bytes`
  3. `echo [PID] > .../tasks`

Mechanics 5: Changing the Root (`pivot_root`)

Problem: Even in a namespace, `ls /` shows the Host's files. This is dangerous. **Solution:** We need to change what "/" means.

Old Way: `chroot`

- Changes root directory for current process.
- Can be escaped easily.

Modern Way: `pivot_root`

- Swaps the mount point of / with a new directory.
- Actually unmounts the old root from the container's view.
- More secure.

Outline

Step 1: The Engine - clone()

We don't use `fork()`. We use `clone()` to finely control sharing.

```c
#define _GNU_SOURCE
#include <sched.h>

// 1. Define the stack for the child (it needs its own memory)
char child_stack[1024 * 1024]; // 1MB Stack

// 2. Define the Flags (The "Magic")
// NEWPID: I am PID 1. NEWUTS: I have my own name. NEWNS: My own mounts.
int flags = CLONE_NEWPID | CLONE_NEWUTS | CLONE_NEWNS | SIGCHLD;

// 3. Launch
pid_t pid = clone(child_function,
                  child_stack + sizeof(child_stack),
                  flags,
                  NULL);

```

Step 2: Inside the Container (The Child)

Once execution enters `child_function`, you are isolated. Now you must setup the environment.

```c
int child_function(void *arg) {
    // A. Identity
    sethostname("my-container", 12);

    // B. Environment (CRITICAL)
    // Unmount old proc (if propagated) and mount new proc
    mount("proc", "/proc", "proc", 0, NULL);

    // C. Life
    // Replace this setup process with an interactive shell
    char *args[] = {"/bin/bash", NULL};
    execv(args[0], args);

    return 0;
}
```

Resources & Next Steps

Action Plan:

1. Write a simple C program calling `clone` with just `CLONE_NEWUTS`.
2. Verify `hostname` change affects only the child.
3. Add `CLONE_NEWPID` and the `mount` logic.
4. Verify with `ps aux`.

Reading:

- `man 2 clone`, `man 7 namespaces`.
- Look at simple C container tutorials online.