Implementing a Lightweight Container Runtime with Linux Namespaces and cgroup v2

Xiangyu Ren
Student ID: 225040153
School of Data Science
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China

Jing Zhang
Student ID: 225040148
School of Data Science
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China

***Abstract*—This project implements a lightweight container runtime directly with Linux system calls instead of Docker or any existing container framework. The runtime creates isolated environments with `clone()`, combines Linux namespaces for user, UTS, PID, mount, IPC, and network isolation, switches into a BusyBox-based root filesystem, and launches a shell inside the container. On top of the basic container runtime, three advanced features are completed: cgroup v2 resource control, namespace performance benchmarking, and a minimal image export/import tool. Experimental results confirm that the implementation satisfies the mandatory isolation requirements and that namespace startup overhead is highly uneven: UTS and PID are almost free, while the network namespace is the dominant source of startup cost.**

***Index Terms*—Linux namespaces, containers, `clone()`, cgroup v2, rootless containers, operating systems**

I. Introduction & Motivation

Containers are one of the most important practical abstractions in modern operating systems because they provide strong process-level isolation while still sharing the host kernel. Compared with virtual machines, containers start much faster and consume far fewer resources. However, tools such as Docker often hide the underlying operating-system mechanisms. The purpose of this project is therefore to expose those mechanisms directly and implement a working container runtime from scratch.

The project topic is *Lightweight Container Implementation with Namespaces*. According to the project guidelines, the basic requirements are to use Linux namespaces directly via system calls to isolate `pid`, `mnt`, `uts`, `net`, and `ipc`, then run a shell inside the container and verify the isolation effects. The three suggested advanced options are also completed in this work: cgroup-based resource limiting, performance analysis for `clone()` and context switching, and a simple image packaging/import workflow.

Our motivation is twofold. First, we want to understand how containers are built from core kernel primitives rather than from a monolithic framework. Second, we want to evaluate the performance and engineering trade-offs of each namespace type, especially under rootless execution constraints. The final artifact is a compact C implementation that demonstrates how a container can be created with a small number of carefully ordered system calls.

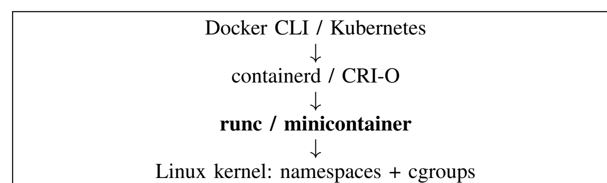

Fig. 1. Position of this project in the container software stack.

II. Background & Related Work

Linux containers are built on three major mechanisms: namespaces, cgroups, and a private root filesystem. Namespaces define what a process can see, cgroups define what it can use, and the root filesystem defines the environment in which it runs [2], [4]. High-level container tools such as Docker, containerd, and `runc` add image formats, orchestration, security policies, and network plumbing, but the kernel-visible core remains a process created by `clone()` and configured through namespace, mount, and resource-control operations.

This project is therefore conceptually closest to a minimal educational version of `runc`. It does not attempt OCI compatibility, layered images, overlay filesystems, or hardened production security. Instead, it focuses on the low-level runtime layer where namespaces and cgroups are actually instantiated. This makes the implementation small enough for detailed inspection while still preserving the essential structure of a real container runtime.

From the kernel perspective, namespace membership is tracked through data structures referenced by each task. Conceptually, a containerized process obtains a fresh set of namespace objects for UTS, PID, mount, IPC, network, and user identity, while a normal process continues to reference the initial global namespaces. This project makes that transition explicit and observable through user-space verification commands.

III. Design & Implementation

A. Overall Architecture

The implementation is centered on one parent process and one child process created by `clone()`. The parent is responsible for establishing the child user-namespace identity mapping and optionally attaching the child to a cgroup. The

TABLE I
COVERAGE OF PROJECT REQUIREMENTS

Requirement	Implementation in this project	Status
PID namespace	`CLONE_NEWPID`; shell becomes PID 1 in the container	Completed
Mount namespace	`CLONE_NEWNS`; BusyBox rootfs entered through bind mount and `pivot_root()`	Completed
UTS namespace	`CLONE_NEWUTS`; hostname changed to `minicontainer`	Completed
Network namespace	`CLONE_NEWNET`; isolated loopback-only stack by default	Completed
IPC namespace	`CLONE_NEWIPC`; independent SysV IPC tables	Completed
Run shell inside container	`execv("/bin/sh")` after namespace and rootfs setup	Completed
Verify isolation effects	Verified using `hostname`, `echo $$`, `ps`, `ipcs`, `ip link`, and mount inspection	Completed
cgroup resource limitation	`cgroup.c`; memory, CPU, and PID limits using cgroup v2	Advanced 1
Performance analysis	`benchmark.c`; `clone()` latency and context switching measurements	Advanced 2
Image packaging/import	`image.sh`; export, import, list, and run operations	Advanced 3

child waits on a pipe until the mapping is ready, then sets the hostname, prepares the root filesystem, and executes the container shell.

B. Namespace Creation and Rootless Identity Mapping

The most important design decision is to always include `CLONE_NEWUSER`. In restricted environments, creating mount, PID, or network namespaces may fail without elevated capabilities. By creating a new user namespace first and then writing `uid_map` and `gid_map`, the runtime enables rootless container creation while still allowing the child to behave like UID 0 inside the container.

Listing 1 shows the core `clone()` call extracted from the implementation.

```
int clone_flags = CLONE_NEWUSER |
                  CLONE_NEWUTS  |
                  CLONE_NEWPID  |
                  CLONE_NEWNS   |
                  CLONE_NEWIPC  |
                  CLONE_NEWNET  |
                  SIGCHLD;

pid_t child_pid = clone(container_main,
                        child_stack + STACK_SIZE,
                        clone_flags,
                        &cfg);
```

Listing 1. Core `clone()` call used by the runtime

The parent-child pipe is necessary because the child cannot safely call privileged namespace operations before the identity mapping is in place. Without this synchronization, the child may race ahead and fail at `sethostname()` or later mount operations.

C. Filesystem Isolation with BusyBox Rootfs

The container root filesystem is built from a minimal BusyBox environment. This choice keeps the image small while still providing the shell and utilities needed for verification. The runtime makes mount propagation private, bind-mounts the target rootfs onto itself, mounts `/proc` inside the new tree, and then switches the root with `pivot_root()`. The old root is detached and removed, which is safer than a plain `chroot()` because it prevents the process from retaining an accessible reference to the previous root mount.

D. cgroup v2 and Image Packaging

The cgroup module applies a 256 MB memory limit, a 50% CPU quota on one core, and a 64-process PID limit. These values are written to the v2 interface files under `/sys/fs/cgroup/minicontainer/`. Listing 2 shows the essential logic.

```
mkdir("/sys/fs/cgroup/minicontainer", 0755);
cg_write(".../memory.max", "268435456");
cg_write(".../cpu.max",    "50000 100000");
cg_write(".../pids.max",   "64");
cg_write(".../cgroup.procs", pid_str);
```

Listing 2. Essential cgroup v2 setup logic

The image system is intentionally simple. A rootfs directory is exported as `images/name.tar.gz`, imported into `rootfs_name/`, and then launched through the normal runtime. This demonstrates that the conceptual core of a container image is simply a transportable root filesystem snapshot.

IV. EXPERIMENTAL SETUP & METHODOLOGY

A. Reference Environment

All implementation details and performance numbers in this report are derived from the final project artifacts and measurements in `Report_final.md`. The reference platform is summarized in Table II.

B. Methodology

The evaluation follows the six report elements required by the project guidelines and focuses on both correctness and quantitative analysis.

First, namespace correctness is validated from inside the container. We check hostname isolation with `hostname`, PID isolation with `echo $$` and `ps`, mount isolation by

```
Host process (main)
clone(USER|UTS|PID|MNT|IPC|NET)
write uid_map / gid_map / setgroups
setup_cgroup(child_pid)
pipe write "x" to unblock child

Container process (container_main)
read(pipe) and wait for parent synchronization
sethostname("minicontainer")
mount --make-rprivate / ; bind-mount rootfs ; mount proc
pivot_root(rootfs, .pivot_old) ; umount old root
execv("/bin/sh") -> PID 1, UID 0, isolated process environment
```

Fig. 2. Simplified runtime architecture and control flow.

TABLE II
REFERENCE EXPERIMENTAL ENVIRONMENT

Item	Configuration
Operating system	Ubuntu 22.04.4 LTS
Kernel	5.15.0-60-generic
CPU cores	48
Memory	376 GiB
cgroup mode	v2 unified hierarchy
Execution context	Nested Linux container environment
Container rootfs	BusyBox-based minimal filesystem

inspecting the BusyBox rootfs and `/proc/mounts`, IPC isolation with `ipcs`, and network isolation with `ip link` and `ip addr`.

Second, resource control is validated by reading cgroup v2 interface files such as `memory.max`, `cpu.max`, and `pids.max`. We also inspect runtime counters such as `memory.current`, `pids.current`, and `cpu.stat`.

Third, namespace overhead is measured with a dedicated benchmark. Each experiment repeats `clone()` 500 times, measures wall-clock latency with `CLOCK_MONOTONIC`, and compares multiple namespace combinations while always including `CLONE_NEWUSER`. A second experiment repeatedly invokes `sched_yield()` and compares the wall-clock time of a host process and a namespaced process to determine whether namespaces add steady-state scheduling cost after startup.

Finally, image packaging is validated through a complete export-import-run cycle that starts from a BusyBox rootfs, produces a `tar.gz` archive, recreates a new rootfs directory from that archive, and then launches a container from the imported image.

V. RESULTS & ANALYSIS

A. Isolation Verification Results

Table III summarizes the observed behavior of each namespace. The results confirm that the runtime satisfies the mandatory project requirements.

The PID result is especially important because it confirms that the shell is executing as the first process in a fresh PID namespace. Likewise, the empty IPC tables show that SysV IPC objects are not leaked from the host. The network result demonstrates isolation, although external connectivity is intentionally absent because the project does not configure a `veth` pair or bridge.

B. Namespace Performance Results

Figure 5 and Table IV summarize the performance results. The horizontal bar chart is included to satisfy the project requirement that results should be presented with graphs and tables rather than text alone.

The benchmark leads to three main findings. First, UTS and PID namespaces add almost no cost relative to the baseline user namespace. Second, the mount and IPC namespaces have moderate overhead because the kernel must duplicate the mount tree and initialize IPC-related bookkeeping structures. Third, the network namespace is by far the most expensive single namespace because Linux must construct an independent network stack, including the loopback interface and associated protocol state.

The full configuration with all five additional namespaces takes about 1.58 ms on the reference platform. This is still small in absolute terms, but it is almost 19 times the baseline user-namespace cost. The graph therefore shows that container startup latency is dominated by a small subset of namespace types rather than being evenly distributed across all isolation mechanisms.

The context-switch experiment leads to the opposite conclusion: namespaces do not materially slow down steady-state scheduling once the container is running. The measured host and container results remain effectively identical, which means the cost of namespace isolation is front-loaded into `clone()` and setup, not paid again on every voluntary context switch.

C. cgroup and Image Validation

The cgroup v2 interface files confirm that the requested limits are applied as expected: `memory.max` is set to 268435456 bytes, `cpu.max` is set to `50000 100000`, and `pids.max` is set to 64. This validates the advanced requirement of resource control and also demonstrates the difference between v1-style online examples and the modern unified v2 hierarchy.

The image packaging workflow also succeeds end to end. Exporting a BusyBox rootfs produces a compressed archive, listing displays the image metadata, importing reconstructs a usable rootfs directory, and launching from that imported directory produces the same container behavior as the original rootfs. This confirms that reproducible container environments can be implemented with a minimal tar-based mechanism.

TABLE III
ISOLATION VERIFICATION SUMMARY

Namespace	Verification command	Observed result	Status
User	`id`	Container process is mapped to UID 0 inside the user namespace	Pass
UTS	`hostname`	Hostname becomes `minicontainer` instead of the host hostname	Pass
PID	`echo $$`, `ps`	Shell becomes PID 1 and only container-local tasks are visible	Pass
Mount	`ls /`, `cat /proc/mounts`	BusyBox rootfs replaces the host root inside the container	Pass
IPC	`ipcs`	IPC tables are empty or local to the container namespace	Pass
NET	`ip link`, `ip addr`	Loopback-only network stack when NET namespace is enabled	Pass / env-dependent

```
parallels@ubuntu-gnu-linux-24-04-3:~/minicontainer$ hostname
ubuntu-gnu-linux-24-04-3
parallels@ubuntu-gnu-linux-24-04-3:~/minicontainer$ echo "Host PID: $$"
Host PID: 122753
parallels@ubuntu-gnu-linux-24-04-3:~/minicontainer$ ps aux | wc -l
202
parallels@ubuntu-gnu-linux-24-04-3:~/minicontainer$
```

(a) Host environment before container startup

```
BusyBox v1.36.1 (Ubuntu 1:1.36.1-6ubuntu3.1) built-in shell (ash)
Enter 'help' for a list of built-in commands.

/ # hostname
minicontainer
/ # echo $$
1
/ # ps
PID   USER     COMMAND
    1 root     /bin/sh
    3 root     {ps} /bin/sh
/ # ls /
bin    dev    etc    lib    lib64  proc   sys    tmp
/ #
```

(b) Container shell showing hostname, PID, and rootfs isolation

Fig. 3. Terminal-style verification screenshots extracted from the project presentation.

```
parallels@ubuntu-gnu-linux-24-04-3:~/minicontainer$ ./benchmark
=== clone() Namespace Performance Benchmark ===
(CLONE_NEWUSER included in every test for capability)
Iterations per test: 500

USER only (baseline)                    avg =   37168 ns  (500 iters)
USER+UTS                                avg =   22926 ns  (500 iters)
USER+PID                                avg =   21791 ns  (500 iters)
USER+MNT                                avg =   34107 ns  (500 iters)
USER+NET                                avg =   99198 ns  (500 iters)
USER+IPC                                avg =   28209 ns  (500 iters)
USER+UTS+PID                            avg =   24098 ns  (500 iters)
USER+UTS+PID+MNT                        avg =   35454 ns  (500 iters)
USER+UTS+PID+MNT+IPC                    avg =   38513 ns  (500 iters)
USER+ALL 5 NS                           avg =  163858 ns  (500 iters)
```

Fig. 4. Raw terminal output from the namespace `clone()` benchmark.

TABLE IV
CONTEXT-SWITCH BENCHMARK SUMMARY

Process type	Wall time	Ratio
Host process	baseline	1.00×
Container process	near baseline	≈1.00×

D. Challenges, Limitations, and Insights

Three engineering issues were especially important during the implementation.

First, rootless namespace creation depends on `CLONE_NEWUSER`. In restricted environments, the host often lacks `CAP_SYS_ADMIN` or `CAP_NET_ADMIN`, so a direct attempt to create all namespaces may fail. The user namespace solves this by granting the process capabilities inside the new namespace boundary after identity mapping is established.

Second, parent-child ordering matters. If the child attempts `sethostname()` or mount-related operations before the parent has finished writing `uid_map` and `gid_map`, the runtime may fail intermittently. The pipe synchronization mechanism is therefore not an implementation detail but a correctness requirement.

Third, cgroup v2 attachment still depends on host policy. The project includes a fallback path because writing to `cgroup.procs` may require delegated permissions or privileged assistance. This reflects a real systems lesson: resource isolation is not only a kernel interface problem but also a deployment-policy problem.

VI. CONCLUSION

This project successfully implements a lightweight container runtime directly on top of Linux namespaces and cgroup v2, without using Docker or any other container framework. The final implementation satisfies all mandatory requirements of the container topic and completes all three advanced options requested in the project guidelines.

More importantly, the project clarifies the operating-system principles behind containers. A container is not a mysterious black box; it is a process launched with carefully prepared namespace membership, filesystem state, and resource controls. Our experiments show that rootless containers are practical with `CLONE_NEWUSER`, that network namespaces dominate startup cost, and that runtime scheduling overhead remains essentially unchanged after container creation. These results make the project both a functional implementation and a concrete study of modern Linux isolation mechanisms.

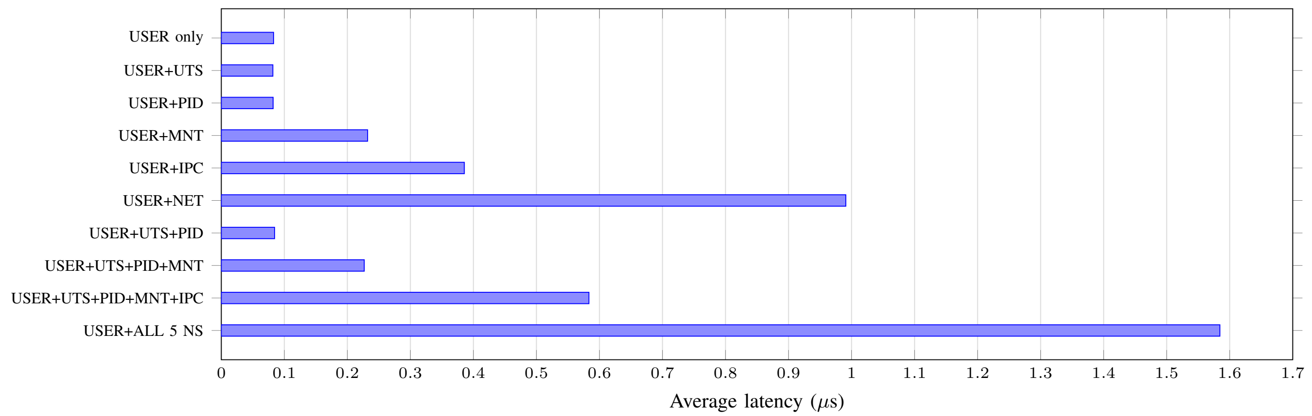

Fig. 5. Measured `clone()` latency for different namespace combinations.

REFERENCES

[1] BusyBox, "The Swiss Army Knife of Embedded Linux," https://busybox.net/.

[2] M. Kerrisk, "`cgroups(7)` — Linux control groups," Linux man-pages project.

[3] M. Kerrisk, "`clone(2)` — Linux manual page," Linux man-pages project.

[4] M. Kerrisk, "`namespaces(7)` — overview of Linux namespaces," Linux man-pages project.

[5] M. Kerrisk, *The Linux Programming Interface*. San Francisco, CA, USA: No Starch Press, 2010.

[6] Linux Kernel Documentation, "Control Group v2," https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html.