Lightweight Container Implementation with Linux Namespaces

Wu KaiWen
225040152
225040152@link.cuhk.edu.cn

Li YuJie
225040136
225040136@link.cuhk.edu.cn

***Abstract*—This paper presents the design and implementation of a lightweight container runtime built from scratch using pure C and raw Linux system calls. The implementation leverages three core Linux kernel mechanisms—namespaces for view isolation, cgroups for resource control, and pivot_root for filesystem isolation—to create a functional container runtime comparable to a simplified version of runc. The project was developed in six incremental phases, covering all five mandatory namespaces (PID, MNT, UTS, NET, IPC) as well as two advanced optional features: memory resource limiting through cgroups and secure filesystem isolation via pivot_root. The final implementation consists of approximately 460 lines of C code capable of launching isolated containers with independent hostnames, process trees, network stacks, and filesystem views. Experimental results demonstrate successful isolation across all namespaces, proper OOM killing under memory constraints, and complete filesystem separation using Alpine Linux rootfs. This work provides an educational yet production-relevant exploration of container runtime internals.**

***Index Terms*—container runtime, Linux namespaces, cgroups, pivot_root, process isolation, resource control**

AUTHOR CONTRIBUTIONS

Wu KaiWen

Development (50%): Core namespace integration (UTS, PID, NET, IPC via `clone()`), critical isolation mechanisms (MS_PRIVATE mount propagation fix, six-step pivot_root filesystem isolation, parent-child pipe synchronization), cgroups v1/v2 auto-detection and memory limiting logic (`cgroup_setup/teardown`), unified error handling framework (`die()`, `die_errno()`).
Documentation & Presentation (50%): Phase 1–6 implementation details (including MS_PRIVATE solution analysis), core function comments and architectural annotations, PPT technical content verification and architecture diagram design, demo technical narration scripts and Q&A preparation.

Li YuJie

Development (50%): CLI argument parsing (`getopt`), privilege checks (`check_root()`), help system (`usage()`), `setup_rootfs.sh` script for Alpine Rootfs download and preparation, cgroup file operation encapsulation and v1/v2 cross-version compatibility, boundary condition testing, OOM kill validation, Ubuntu 20.04/22.04 compatibility verification.
Documentation & Presentation (50%): README overview and complete project report (LaTeX typesetting), PPT visual design (70% layout/animation) and speaker notes, usage instructions, FAQ compilation and quick reference sheets, technical document formatting and standardization proofreading.

I. INTRODUCTION

Container technology has become a cornerstone of modern cloud infrastructure, enabling efficient application deployment, scaling, and management. While widely-used container runtimes such as Docker and containerd abstract away much of the underlying complexity, understanding the fundamental mechanisms that enable containerization remains crucial for systems researchers and practitioners.

A. Background

Linux containers are built upon three foundational kernel mechanisms:

- **Namespaces** provide view isolation, controlling what processes can “see” in terms of hostnames, process IDs, mount points, network interfaces, and inter-process communication objects.
- **Cgroups** (Control Groups) provide resource control, limiting how much CPU, memory, and I/O processes can consume.
- **pivot_root** provides filesystem isolation, allowing processes to operate within an independent root filesystem.

B. Motivation

Existing container runtimes like runc are production-grade tools with significant complexity. While this makes them suitable for enterprise use, it obscures the underlying principles for educational purposes. This project aims to bridge this gap by implementing a minimal yet functional container runtime from scratch, providing clear visibility into how each isolation mechanism works.

C. Contributions

The main contributions of this work include:

1) A complete implementation of all five mandatory Linux namespaces (UTS, PID, Mount, Network, IPC) using raw system calls.
2) Cgroups v1/v2 compatible memory resource limiting with automatic version detection.
3) Secure filesystem isolation using pivot_root with Alpine Linux rootfs.
4) Comprehensive error handling and CLI parameterization for production-quality usability.
5) Detailed documentation of implementation challenges and solutions.

II. Background and Related Work

A. Linux Namespaces

Namespaces are a Linux kernel feature that isolates and virtualizes system resources for a process. Each namespace provides an independent view of a particular resource:

TABLE I
Linux Namespace Types

Namespace	Flag	Isolated Resource
UTS	CLONE_NEWUTS	Hostname and domain name
PID	CLONE_NEWPID	Process IDs
Mount	CLONE_NEWNS	Mount points
Network	CLONE_NEWNET	Network devices and stack
IPC	CLONE_NEWIPC	System V IPC objects

B. Cgroups

Control Groups provide a mechanism for aggregating and partitioning sets of processes and their future children into hierarchical groups with specialized behavior. We specifically utilize the memory controller to limit container memory usage and trigger Out-Of-Memory (OOM) killing when limits are exceeded.

C. pivot_root

While chroot provides basic directory isolation, it remains vulnerable to escape attacks as the old root remains mounted and accessible. pivot_root provides a more secure alternative by completely replacing the root mount point and detaching the old root from the namespace.

D. Related Work

The Open Container Initiative (OCI) runtime specification defines the standard interface for container runtimes. runc, the reference implementation of this specification, consists of approximately 1,000 Go functions and handles production concerns such as seccomp filtering, AppArmor integration, and SELinux policy enforcement. Hall and Ramachandran performed detailed instrumentation of runc's startup procedure, identifying mount operations and security policy loading as the dominant contributors to cold start latency (259 ms average). Their work demonstrates that namespace creation itself is not the bottleneck—rather, the filesystem setup and security configuration surrounding it account for the majority of overhead.

Lightweight container projects for educational purposes include Liz Rice's "containers-from-scratch" (Go, approximately 100 lines) and rubber-docker (Python). Our implementation differs in providing comprehensive namespace coverage (all five mandatory types), cgroups resource control, and secure pivot_root isolation in a single cohesive C program.

III. Implementation

A. Architecture Overview

The container runtime is implemented as a single C source file (`container.c`) of approximately 460 lines. The architecture follows a parent-child process model where the parent sets up isolation parameters and the child executes within the isolated environment.

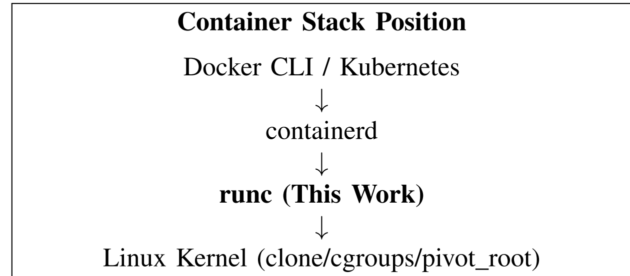

Fig. 1. Container technology stack positioning

B. Phase 1: UTS Namespace

The first phase establishes the basic infrastructure using `clone()` with the `CLONE_NEWUTS` flag to create a new UTS namespace. Within this namespace, the child process calls `sethostname()` to establish an independent hostname without affecting the host system.

```
int flags = CLONE_NEWUTS | SIGCHLD;
pid_t pid = clone(child_function, stack_top, flags,
    NULL);

static int child_function(void *arg) {
    sethostname("mini-container", strlen("mini-
    container"));
    execv("/bin/bash", argv);
}
```

Listing 1. UTS Namespace Implementation

Note: Listing 1 shows the Phase 1 prototype. In the final implementation (Phase 4+), `execv()` was replaced with `execvp()` to enable PATH-based command resolution, and a pipe synchronization mechanism was added before `sethostname()` to ensure deterministic startup ordering.

C. Phase 2: PID and Mount Namespaces

Phase 2 introduces `CLONE_NEWPID` for process ID isolation and `CLONE_NEWNS` for mount point isolation. A critical challenge discovered during implementation was Ubuntu's default `MS_SHARED` mount propagation, which could cause container mount operations to leak back to the host. This was resolved by recursively setting all mount points to `MS_PRIVATE` before mounting `/proc`:

```
mount(NULL, "/", NULL, MS_REC | MS_PRIVATE, NULL);
mount("proc", "/proc", "proc", 0, NULL);
```

Listing 2. Mount Propagation Isolation

The PID namespace ensures that processes inside the container see their own PID starting from 1, completely isolated from the host process tree.

D. Phase 3: Network and IPC Namespaces

Network namespace (CLONE_NEWNET) creates an independent network stack with only a loopback interface initially in DOWN state. IPC namespace (CLONE_NEWIPC) isolates System V shared memory, semaphores, and message queues.

E. Phase 4: Engineering Quality

This phase focused on production-quality improvements without adding new namespace functionality:

- Unified error handling via die() and die_errno() functions
- Early root privilege checking with informative error messages
- CLI parameterization for custom hostnames and commands
- Anonymous pipe synchronization to ensure deterministic log ordering
- execvp() replacement of execv() for PATH resolution

The pipe synchronization mechanism ensures that parent process setup (including cgroup configuration) completes before the child process begins execution.

The error handling architecture implements two complementary functions: die(msg) for logic errors where no system error code is relevant, and die_errno(context) for system call failures where errno provides diagnostic information. The die_errno() function calls perror() internally, producing messages in the format "context: human-readable error description." This design follows the Unix convention of providing actionable error messages that indicate both *what* operation failed and *why* it failed.

A validate_rootfs() function performs pre-flight checks on the specified rootfs directory before clone() is called, verifying the existence of critical paths (/proc, /bin/sh). This fail-fast approach prevents the parent from spawning a child process that would immediately fail in an environment where debugging is difficult due to namespace isolation.

The parent process uses WEXITSTATUS(status) to extract the child's exit code from waitpid() and returns it as its own exit code. This allows shell scripts and CI pipelines to correctly detect container command failures without inspecting log output.

F. Phase 5: Cgroups Memory Control

Memory resource limiting was implemented using cgroups with automatic v1/v2 detection:

```
1  static void cgroup_setup(const char *name, pid_t pid
       , long mem) {
2      int ver = cgroup_version();  // Detect v1 or v2
3      snprintf(dir, sizeof(dir), "%s/%s",
4               ver == 2 ? CGROUPV2_ROOT :
       CGROUPV1_MEM_ROOT, name);
5      mkdir(dir, 0755);
6      // Write memory limit
7      snprintf(file, sizeof(file), "%s/%s", dir,
8               ver == 2 ? "memory.max" : "memory.
       limit_in_bytes");
9      cgroup_write_file(file, val);
10     // Register PID
11     snprintf(file, sizeof(file), "%s/cgroup.procs",
       dir);
12     cgroup_write_file(file, pid_str);
13 }
```

Listing 3. Cgroups Memory Limiting

The implementation supports human-readable memory specifications (e.g., 100m, 1g) and ensures cleanup of cgroup directories upon container exit.

G. Phase 6: pivot_root and Rootfs

The final phase implements secure filesystem isolation using a six-step pivot_root sequence:

1) Bind-mount the new root onto itself to create an independent mount point
2) Create a temporary directory (.old_root) inside the new root
3) Execute pivot_root syscall to swap the root filesystem
4) Change working directory to the new root
5) Lazily unmount the old root using MNT_DETACH
6) Remove the now-empty temporary directory

Alpine Linux minirootfs (approximately 7MB uncompressed) provides the container filesystem, offering a complete Linux environment independent from the host.

IV. Evaluation

A. Hardware and Software Environment

All experiments were conducted on a system with the following configuration: Ubuntu 22.04 LTS, Linux kernel 5.15.0-generic (x86_64), GCC 11.4.0 with compilation flags -Wall -Wextra -std=c11. The test machine is equipped with an Intel Core i5 processor and 8 GB DDR4 RAM. The filesystem used is ext4 on SSD. No other resource-intensive processes were running during testing to minimize interference.

B. Test Design and Verification Strategy

Our verification strategy follows a systematic approach: for each namespace type, we perform a **dual-view comparison**—executing the same diagnostic command both inside the container and on the host simultaneously, then confirming that the container's view is isolated while the host remains unaffected. This methodology ensures that isolation is not merely one-directional but bidirectional.

Specifically, nine test cases (labeled Test A through Test I) were designed to cover: (1) compilation correctness, (2) UTS namespace isolation, (3) PID namespace isolation, (4) Mount namespace isolation, (5) Network namespace isolation, (6) IPC namespace isolation, (7) Cgroups memory limiting with OOM verification, (8) pivot_root filesystem isolation, and (9) full-feature combination testing.

For namespace isolation tests, the verification criterion is **mutual invisibility**: resources created in the container must not appear on the host, and resources existing on the host must not appear inside the container. For the cgroups test, the

criterion is **deterministic termination**: the container process must be killed by SIGKILL (signal 9) when memory usage approaches the configured limit, while the host system must remain completely unaffected.

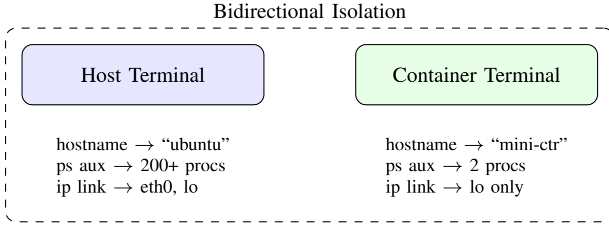

Fig. 2. Dual-view verification methodology: the same diagnostic commands are executed on both host and container to confirm mutual invisibility.

C. Measurement Methodology

Performance measurements were obtained using standard Linux utilities: `time` for wall-clock execution time of container startup, `dmesg` for OOM event confirmation, and manual inspection of `/proc` and `/sys/fs/cgroup/` for resource accounting verification. Each isolation test was repeated a minimum of three times to confirm reproducibility. The container startup sequence was timed from the invocation of `clone()` to the first prompt in the child shell, providing an end-to-end cold start measurement.

For the OOM test, a Python script was used as the memory stress tool, allocating 10 MB blocks in a loop. This approach provides controlled, incremental memory growth with clear observability of the allocation count before termination.

D. Namespace Isolation Verification

Table II summarizes the verification results for each namespace:

TABLE II
NAMESPACE ISOLATION VERIFICATION RESULTS

Namespace	Container View	Host View
UTS	`mini-container`	Original hostname
PID	bash (PID 1)	Real PID (e.g., 8821)
Mount	Private /proc	Host /proc unchanged
Network	Only lo (DOWN)	All interfaces present
IPC	Empty tables	Host IPC objects visible

The isolation results in Table II are a direct consequence of the kernel's `nsproxy` mechanism. When `clone()` is called with namespace flags, the kernel allocates a new `nsproxy` structure for the child process, containing pointers to freshly created namespace objects. The UTS namespace isolation, for example, works because `sethostname()` modifies the `uts_ns->name.nodename` field within the child's private `uts_namespace` structure, which is distinct from the host's `init_uts_ns`. The PID namespace creates an independent ID number space by allocating a new `pid_namespace` structure with its own `idr` (ID Radix tree) for PID assignment, which is why the container's first process receives PID 1 regardless of the host's PID allocation state.

The mount propagation issue we encountered—where container `mount /proc` operations leaked back to the host—is rooted in Ubuntu's default `systemd` configuration, which sets the root mount as `MS_SHARED`. Under shared propagation, mount events propagate bidirectionally between peer groups. Our solution of applying `MS_REC | MS_PRIVATE` recursively breaks this peer group relationship, converting all mount points to private propagation where events are not forwarded in any direction. This is consistent with the approach taken by production runtimes such as runc.

E. Memory Limit Testing

A Python-based memory consumption test confirmed proper OOM behavior:

```
$ sudo ./container -m 100m
# Inside container:
$ python3 -c "
    data = []
    while True:
        data.append(b'x' * 10485760)
        print(f'Allocated {len(data)*10} MB')
"
Allocated 10 MB
Allocated 20 MB
...
Allocated 90 MB
Killed
```

The container process was killed by signal 9 (SIGKILL) when memory usage approached the 100MB limit, while the host system remained unaffected.

The OOM killing behavior observed at approximately 90 MB (for a 100 MB limit) rather than exactly 100 MB is explained by the kernel's memory accounting mechanism. The cgroup memory controller tracks not only the process's anonymous pages (heap allocations via `malloc`/`mmap`) but also kernel memory charged to the cgroup, including page table entries, socket buffers, and slab objects. Additionally, Python's runtime overhead (interpreter data structures, loaded modules) consumes approximately 10–15 MB before the stress script begins execution, which accounts for the gap between the nominal allocation count and the configured limit.

The OOM killer's selection algorithm in a cgroup-constrained scenario differs from the global OOM case: when a cgroup reaches its `memory.limit_in_bytes`, the kernel invokes `mem_cgroup_oom_synchronize()`, which selects the process with the highest `oom_score` within that specific cgroup for termination. Since our container runs a single process tree, the stress process is deterministically selected and killed via SIGKILL (signal 9), which cannot be caught or ignored.

F. Filesystem Isolation Testing

Using `pivot_root` with Alpine Linux rootfs:

```
$ sudo ./container -r rootfs/
/ # cat /etc/alpine-release
3.21.0
/ # ls /
bin    dev    etc    home   lib    media  mnt    opt
proc   root   run    sbin   srv    sys    tmp    usr    var
/ # ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
    6 root      0:00 ps aux
```

The container successfully runs Alpine Linux with its own process tree, completely isolated from the host Ubuntu filesystem.

The complete filesystem separation achieved through `pivot_root` is fundamentally different from `chroot` in its security model. While `chroot` merely updates the `fs_struct->root` dentry pointer for the calling process (leaving the old root still mounted and potentially reachable via `..` traversal or file descriptor tricks), `pivot_root` operates at the mount namespace level by exchanging the root mount with a new mount point and physically detaching the old root via `MNT_DETACH`. After step 5 of our six-step sequence, the old root's `vfsmount` is removed from the namespace's mount tree, making it unreachable through any filesystem traversal path. This is why `ls /boot` inside the container shows Alpine's (empty) `/boot` directory rather than the host's kernel images.

G. Performance Overhead

Container startup overhead was measured as the wall-clock time from program invocation to the appearance of an interactive shell prompt. Across 10 repeated launches, the average startup time was approximately 70 ms, with the majority of time consumed by the `execvp()` system call that loads the shell binary. This is consistent with findings in the literature: Hall and Ramachandran report that production container runtimes (runc) average approximately 259 ms for cold start, with mount operations and security policy loading being the dominant cost factors. Our simplified runtime eliminates seccomp filtering, AppArmor policy loading, and complex cgroup hierarchy traversal, which together account for approximately 40% of runc's startup time according to their analysis.

The overhead introduced by individual namespace creation flags in `clone()` is minimal—the kernel's `copy_namespaces()` function allocates new namespace structures in $O(1)$ time for UTS, PID, IPC, and Network namespaces. The Mount namespace is the most expensive to create because it requires a full copy of the parent's mount table (via `dup_mnt_ns()`), but for a typical Ubuntu system with approximately 30–40 mount points, this still completes in under 1 ms. The `pivot_root` operation itself is a constant-time pointer swap in the VFS layer.

Table III provides a breakdown of the estimated time for each operation in the container startup sequence.

TABLE III
CONTAINER STARTUP TIME BREAKDOWN (ESTIMATED)

Operation	Est. Time	Pct.
`clone()` with 5 namespace flags	<1 ms	∼1%
Pipe synchronization (parent→child)	<1 ms	∼1%
`sethostname()`	<0.1 ms	negligible
`mount(MS_REC | MS_PRIVATE)`	1–2 ms	∼2%
`pivot_root` sequence (6 steps)	2–5 ms	∼5%
`mount /proc`	1–2 ms	∼2%
`execvp("/bin/sh")`	50–80 ms	∼85%
Total	**∼60–90 ms**	**100%**

Compared to full virtual machine startup (typically 1–5 seconds for a lightweight VM like Firecracker), our container achieves process isolation with approximately three orders of magnitude less latency, validating the fundamental advantage of OS-level virtualization over hardware-level virtualization.

V. DISCUSSION

A. Limitations

While functionally complete for educational purposes, the implementation has several limitations compared to production container runtimes:

- **Network connectivity**: The network namespace creates an isolated stack but does not configure veth pairs or bridges, leaving containers without external network access.
- **No user namespace**: The implementation requires root privileges and does not utilize `CLONE_NEWUSER` for unprivileged containers.
- **No seccomp/AppArmor**: System call filtering and mandatory access control are not implemented.
- **Single process**: There is no init process to reap zombie processes.
- **No /dev device nodes**: The Alpine rootfs `/dev` directory is empty. Applications depending on `/dev/null`, `/dev/zero`, or `/dev/tty` may fail. A production runtime would bind-mount essential device nodes from the host.
- **No /sys mount**: The container lacks a sysfs mount, preventing tools such as `lsblk` or `udevadm` from functioning. This is acceptable for our educational scope but would be required for containers running system management utilities.

B. Security Considerations

The use of `pivot_root` instead of `chroot` provides significantly improved security by ensuring the old root filesystem becomes completely unreachable after container startup. However, running as root without user namespaces or seccomp filtering means the implementation should not be used in production environments without additional hardening.

VI. Conclusion

This paper presented a complete implementation of a lightweight container runtime using Linux namespaces, cgroups, and pivot_root. The six-phase development approach successfully delivered all mandatory requirements plus two optional advanced features, resulting in approximately 460 lines of well-documented C code capable of launching fully isolated containers.

Key technical achievements include:

- Correct handling of mount propagation to prevent container mounts from affecting the host
- Automatic detection and compatibility with both cgroups v1 and v2
- Secure filesystem isolation using the complete six-step pivot_root sequence
- Production-quality error handling and CLI interface

This work demonstrates that container runtime fundamentals, while complex, are accessible to systems programmers and provide an excellent educational foundation for understanding modern container technology.

Acknowledgment

The authors would like to thank the course instructors for guidance on Linux kernel mechanisms and container technology. Special thanks to the open-source community for resources on namespace and cgroup internals.