Container (Basic): Design Document

Overview

This system implements a basic container runtime using Linux namespaces and cgroups to achieve process isolation. The key architectural challenge is coordinating multiple kernel isolation mechanisms (PID, mount, network namespaces) with resource controls while handling the subtle timing and lifecycle management issues that arise from namespace interactions.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** This section provides foundational context for all milestones (1-4), establishing why process isolation is necessary before implementing PID, mount, network namespaces, and cgroups.

Modern computing environments face a fundamental challenge: how can multiple processes, applications, or users safely share the same physical machine without interfering with each other? This problem becomes particularly acute when running untrusted code, deploying applications with conflicting dependencies, or managing multi-tenant systems where isolation and resource fairness are critical.

The core issue stems from the traditional Unix process model, where all processes share the same global namespace for process IDs, filesystem mounts, network interfaces, and system resources. While Unix permissions provide some level of access control, they don't address the deeper problems of resource contention, dependency conflicts, and the blast radius of misbehaving applications. A runaway process can consume all available CPU or memory, affecting every other process on the system. Applications with conflicting library versions cannot coexist peacefully. Network services can interfere with each other by binding to the same ports.

Process isolation addresses these challenges by creating controlled boundaries around groups of processes, giving each group its own view of system resources while maintaining efficient resource sharing at the kernel level. Unlike full machine virtualization, process isolation operates at a much finer granularity, allowing for lightweight separation that scales to hundreds or thousands of isolated environments on a single host.

This design document explores the implementation of a basic container runtime that leverages Linux kernel features—specifically namespaces and cgroups—to achieve robust process isolation. The system we'll build demonstrates how these kernel primitives can be orchestrated to create secure, resource-controlled execution environments that are both lighter weight than virtual machines and more secure than traditional process separation.

Mental Model: Apartment Building

Think of a computer system as a large apartment building, and processes as the various tenants living in it. In the traditional Unix model, it's as if all tenants share one massive communal space with no private rooms, no individual mailboxes, and no separate utility meters. Everyone can see everyone else's belongings (process tree), use anyone's phone line (network interfaces), rearrange the shared furniture (filesystem mounts), and there's no way to prevent one tenant from using all the electricity (CPU and memory resources).

This communal arrangement creates obvious problems. If one tenant decides to play loud music at 3 AM (a CPU-intensive process), everyone suffers. If someone rearranges all the furniture (modifies global filesystem mounts), it affects everyone's daily routine. When the mail arrives, anyone can read anyone else's letters (no process isolation). And if someone leaves the water running (memory leak), the entire building's utilities suffer.

Containers provide each tenant with their own private apartment within the same building. Each apartment has:

- **Private room numbering** (PID namespace): Inside apartment 3B, the rooms might be numbered 1, 2, 3, but these numbers are completely independent from apartment 4A's room numbering. A tenant in 3B who thinks they're in "room 1" might actually be in the building's room 847 from the building manager's perspective.
- **Individual interior layouts** (mount namespace): Each apartment can arrange their furniture, hang pictures, and organize their space completely independently. One tenant might prefer a minimalist setup while another fills every corner. These choices don't affect neighboring apartments.
- **Separate phone/internet lines** (network namespace): Each apartment gets its own phone number and internet connection. Tenants can't accidentally answer each other's phones or interfere with each other's network traffic.
- **Individual utility budgets** (cgroups): The building manager allocates specific amounts of electricity, water, and heating to each apartment. If one tenant tries to run too many appliances, their circuit breaker trips, but other apartments continue running normally.

The building manager (kernel) maintains the overall infrastructure—the structural walls, main electrical system, water mains, and building-wide policies. But within each apartment's boundaries, tenants operate independently. They can't see into other apartments, can't consume more than their allocated resources, and can't break things in ways that affect their neighbors.

This apartment model captures the essential insight of containerization: **strong isolation boundaries within shared infrastructure**. The kernel provides the "building" (hardware resources, core services), while containers provide the "apartments" (isolated process environments with controlled resource access).

Just as apartment buildings are more efficient than giving each tenant their own house (virtual machines), containers are more efficient than full virtualization while still providing the isolation benefits tenants need.

Existing Isolation Approaches

Before diving into container implementation, it's important to understand the landscape of existing isolation approaches and why containers occupy a unique position in the design space. Each approach represents different

trade-offs between isolation strength, resource efficiency, and operational complexity.

Virtual Machines: Maximum Isolation, Maximum Overhead

Virtual machines provide the strongest isolation by running complete, separate operating system instances on virtualized hardware. Each VM gets its own kernel, its own device drivers, and its own complete view of virtualized hardware resources.

Aspect	Description	Trade-offs
Isolation Boundary	Hardware virtualization layer	Complete OS-level separation
Resource Overhead	Full OS + kernel per instance	512MB-2GB+ memory per VM
Startup Time	Full OS boot sequence	30+ seconds typical
Resource Efficiency	Poor - duplicate kernels, drivers	10-50 VMs per host typical
Security Model	Hypervisor-enforced boundaries	Very strong - different kernels
Failure Blast Radius	Contained to single VM	Hypervisor bugs affect all VMs

Virtual machines excel when you need to run different operating systems, when you require maximum security isolation (different kernels can't interfere), or when you need to emulate specific hardware configurations. However, their resource overhead makes them impractical for microservice architectures where you might want to run hundreds of small, specialized services.

Chroot Jails: Filesystem Isolation Only

The `chroot` system call, introduced in Unix Version 7, provides a much lighter-weight isolation mechanism by changing a process's view of the filesystem root. A process running in a chroot jail cannot access files outside its designated directory tree.

Aspect	Description	Trade-offs
Isolation Boundary	Filesystem namespace only	No process, network, or resource isolation
Resource Overhead	Minimal - shared kernel	Nearly zero overhead
Startup Time	Near-instantaneous	Milliseconds
Resource Efficiency	Excellent for filesystem isolation	Hundreds per host easily
Security Model	Filesystem access control only	Weak - shared PID space, network
Failure Blast Radius	Processes can still interfere	No resource or process boundaries

Chroot jails are useful for specific scenarios like FTP servers or build environments where you primarily need filesystem isolation. However, they provide no protection against resource exhaustion attacks, processes can still see and potentially interfere with each other, and network resources remain completely shared. A process in a chroot jail can still consume all system memory or CPU, affecting every other process on the host.

Containers: Balanced Isolation and Efficiency

Containers represent a middle ground that combines multiple Linux kernel isolation primitives to create process-level isolation boundaries that are much stronger than chroot but lighter weight than full virtualization.

Aspect	Description	Trade-offs
Isolation Boundary	Multiple namespace types + cgroups	Process, filesystem, network, resource isolation
Resource Overhead	Shared kernel, isolated userspace	10-50MB typical per container
Startup Time	Process startup time	Milliseconds to seconds
Resource Efficiency	Very good - shared kernel	100s-1000s per host
Security Model	Kernel-enforced namespace boundaries	Strong within same kernel
Failure Blast Radius	Isolated by namespace and cgroups	Kernel vulnerabilities affect all

The key insight that makes containers practical is that most application isolation requirements don't actually need separate kernels. Applications typically need:

- Their own process tree (PID namespace)
- Their own filesystem view (mount namespace)
- Their own network stack (network namespace)
- Controlled resource allocation (cgroups)

But they can safely share the same kernel, device drivers, and core system services. This sharing dramatically reduces memory overhead—instead of 500MB+ per virtual machine, containers might use 10-50MB each.

> **Design Insight**: Containers optimize for the common case where applications need isolation from each other but not from the underlying operating system. This allows much higher density while maintaining practical isolation boundaries.

Comparison Summary

Criteria	Virtual Machines	Chroot Jails	Containers	Best Use Case
Isolation Strength	Maximum	Minimal	Strong	VMs for different OSes
Resource Efficiency	Poor	Excellent	Very Good	Containers for microservices
Operational Complexity	High	Low	Medium	Chroot for simple filesystem isolation
Security Boundaries	Hypervisor	Filesystem only	Multiple namespaces	VMs for untrusted multi-tenant
Density	10-50/host	1000s/host	100s-1000s/host	Containers for application packaging
Startup Performance	Slow	Instant	Fast	Containers for elastic scaling

The fundamental architectural decision in our container implementation is to leverage this balanced approach: **strong enough isolation for most application security and interference requirements, while maintaining the resource efficiency needed for modern microservice architectures**.

Our basic container runtime will demonstrate how to coordinate Linux namespaces (PID, mount, network) with cgroups (resource limits) to achieve this balance. The system provides process-level isolation that prevents applications from interfering with each other while allowing efficient resource sharing through a common kernel.

This approach makes containers particularly well-suited for:

- **Microservice architectures** where you need many small, isolated services
- **CI/CD pipelines** where you need fast, clean build environments
- **Development environments** where you need consistent, reproducible setups
- **Multi-tenant applications** where you need fair resource sharing

The trade-off is that all containers share the same kernel, so kernel-level vulnerabilities or crashes can affect all containers on a host. For scenarios requiring maximum security isolation, virtual machines remain the better choice. But for the vast majority of application deployment scenarios, containers provide an excellent balance of isolation, efficiency, and operational simplicity.

Implementation Guidance

The theoretical understanding of isolation approaches provides the foundation for practical container implementation. This section bridges the gap between understanding why containers are useful and actually building one using Linux kernel primitives.

Technology Recommendations

Component	Simple Option	Advanced Option	Recommendation for Learning
Language	C with basic syscalls	Go with advanced libraries	C - direct kernel interaction
Namespace Creation	`clone()` system call	`unshare()` + `fork()`	Start with `clone()` - single step
Process Management	Basic `wait()` loops	Signalfd + epoll	Basic `wait()` - easier debugging
Network Setup	Shell commands via `system()`	Netlink sockets	Shell commands - fewer dependencies
Cgroups Interface	Direct filesystem writes	libcgroup or systemd	Direct filesystem - clearer understanding
Error Handling	Simple errno checking	Structured error types	errno checking - matches kernel interface

Recommended File Structure

Understanding the isolation concepts leads naturally to a modular code organization that mirrors the conceptual separation:

```
container-basic/
├── src/
│   ├── main.c                    ← Entry point and command parsing
│   ├── container.h               ← Main container API definitions
│   ├── container.c               ← Container lifecycle orchestration
│   ├── namespace/
│   │   ├── namespace.h           ← Common namespace utilities
│   │   ├── pid_namespace.c       ← PID isolation (Milestone 1)
│   │   ├── mount_namespace.c     ← Filesystem isolation (Milestone 2)
│   │   └── network_namespace.c   ← Network isolation (Milestone 3)
│   ├── cgroups/
│   │   ├── cgroups.h             ← Resource limit interface
│   │   ├── cgroups.c             ← Cgroups management (Milestone 4)
│   │   └── limits.c              ← Specific limit controllers
│   └── utils/
│       ├── error.h               ← Error handling utilities
│       ├── error.c               ← Error reporting and cleanup
│       └── syscall_wrappers.c    ← Safe syscall wrapper functions
├── tests/
│   ├── test_pid_namespace.c      ← PID isolation verification
│   ├── test_mount_namespace.c    ← Filesystem isolation tests
│   ├── test_network_namespace.c  ← Network isolation tests
│   └── test_cgroups.c           ← Resource limit tests
├── examples/
│   ├── simple_container.c        ← Basic usage example
│   └── isolated_shell.c         ← Interactive container demo
└── Makefile                      ← Build configuration
```

This structure reflects the apartment building model: the main `container.c` file acts as the building manager, coordinating between different isolation mechanisms (the namespace/ modules) and resource management (the cgroups/ module).

Infrastructure Starter Code

Since the focus is on learning namespace and cgroups coordination, here's complete starter code for error handling and utility functions:

src/utils/error.h:

```c
#ifndef ERROR_H
#define ERROR_H

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

// Error reporting macros that preserve errno and provide context
#define LOG_ERROR(msg) do { \
    fprintf(stderr, "ERROR [%s:%d]: %s: %s\n", \
            __FILE__, __LINE__, (msg), strerror(errno)); \
} while(0)

#define LOG_INFO(msg) do { \
    printf("INFO: %s\n", (msg)); \
} while(0)

#define HANDLE_ERROR(condition, msg) do { \
    if (condition) { \
        LOG_ERROR(msg); \
        return -1; \
    } \
} while(0)

// Cleanup helper for partial failures
typedef struct cleanup_list {
    void (*cleanup_func)(void *data);
    void *data;
    struct cleanup_list *next;
```

```
} cleanup_list_t;

// Register cleanup function to call on failure
void register_cleanup(cleanup_list_t **list, void (*func)(void*), void *data);

// Execute all registered cleanup functions
void execute_cleanup(cleanup_list_t **list);

#endif /* ERROR_H */
```

src/utils/syscall_wrappers.c:

```c
#define _GNU_SOURCE

#include <sys/syscall.h>

#include <unistd.h>

#include <sched.h>

#include "error.h"


// Safe wrapper for clone() syscall with error checking

pid_t safe_clone(int (*fn)(void *), void *stack, int flags, void *arg) {

    pid_t pid;



    // Validate stack pointer (common mistake)

    if (!stack) {

        errno = EINVAL;

        return -1;

    }



    pid = clone(fn, stack, flags, arg);

    if (pid == -1) {

        LOG_ERROR("clone() failed");

        return -1;

    }



    return pid;

}


// Safe wrapper for mount() with detailed error reporting

int safe_mount(const char *source, const char *target,

               const char *filesystemtype, unsigned long mountflags,

               const void *data) {
```

```
    int result = mount(source, target, filesystemtype, mountflags, data);
    if (result == -1) {
        char error_msg[256];
        snprintf(error_msg, sizeof(error_msg),
                "mount(%s, %s, %s) failed",
                source ? source : "NULL",
                target ? target : "NULL",
                filesystemtype ? filesystemtype : "NULL");
        LOG_ERROR(error_msg);
        return -1;
    }
    return 0;
}

// Create directory with parent directories, ignore if exists
int ensure_directory(const char *path) {
    char cmd[512];
    snprintf(cmd, sizeof(cmd), "mkdir -p %s", path);
    int result = system(cmd);
    if (result != 0) {
        LOG_ERROR("Failed to create directory");
        return -1;
    }
    return 0;
}
```

Core Logic Skeleton

The learner implements the main coordination logic that orchestrates namespace creation and cgroup setup. This skeleton maps directly to the conceptual understanding from the apartment building model:

src/container.h:

```c
#ifndef CONTAINER_H

#define CONTAINER_H


#include <sys/types.h>

#include <stdint.h>


// Container configuration - the "apartment lease agreement"

typedef struct container_config {

    char *rootfs_path;           // Path to container filesystem root

    char *hostname;              // Container hostname (UTS namespace)



    // Resource limits (cgroups configuration)

    size_t memory_limit_bytes;   // Memory limit (0 = unlimited)

    int cpu_percent;             // CPU percentage (0-100, 0 = unlimited)

    int max_processes;           // Maximum number of processes



    // Network configuration

    char *bridge_name;           // Host bridge to connect to

    char *container_ip;          // IP address for container



    // Command to run inside container

    char **argv;                 // Command and arguments

    char **envp;                 // Environment variables

} container_config_t;


// Container instance - tracks the "apartment tenant"

typedef struct container_instance {

    pid_t child_pid;             // Main process PID in container

    int namespace_fds[6];        // File descriptors for namespaces
```

```
    char *cgroup_path;           // Path to container's cgroup
    cleanup_list_t *cleanup;     // Cleanup functions for failure recovery
} container_instance_t;

// Main container lifecycle functions
int container_create(const container_config_t *config,
                     container_instance_t *container);
int container_wait(container_instance_t *container);
int container_destroy(container_instance_t *container);

#endif /* CONTAINER_H */
```

src/container.c (skeleton for learner implementation):

C

```c
#include "container.h"
#include "namespace/namespace.h"
#include "cgroups/cgroups.h"
#include "utils/error.h"

// Child process entry point - runs inside all namespaces
static int container_child_main(void *arg) {
    container_config_t *config = (container_config_t *)arg;

    // TODO 1: Set up mount namespace and pivot to new root
    // Hint: Call mount_namespace_setup(config->rootfs_path)
    // This gives the container its own filesystem view

    // TODO 2: Set up network namespace networking
    // Hint: Call network_namespace_setup(config->container_ip)
    // This configures the container's network interface

    // TODO 3: Set hostname in UTS namespace
    // Hint: Use sethostname(config->hostname, strlen(config->hostname))

    // TODO 4: Set up minimal /proc and /sys filesystems
    // Hint: mount("proc", "/proc", "proc", 0, NULL)
    //       mount("sysfs", "/sys", "sysfs", 0, NULL)

    // TODO 5: Drop privileges if running as root
    // Hint: This is a good place to switch to non-root user

    // TODO 6: Execute the target command
```

```
    // Hint: execve(config->argv[0], config->argv, config->envp)
    // This should never return on success

    return -1; // Should never reach here
}

int container_create(const container_config_t *config,
                     container_instance_t *container) {
    char *stack;
    pid_t child_pid;

    // TODO 1: Create cgroups and set resource limits
    // Hint: Call cgroups_create_and_configure(config, &container->cgroup_path)
    // This must happen before clone() so child can be added to cgroup

    // TODO 2: Allocate stack for clone() call
    // Hint: Use malloc(STACK_SIZE) and adjust for stack growth direction
    // Remember: stacks grow downward on most architectures

    // TODO 3: Call clone() with appropriate namespace flags
    // Hint: flags = CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWNET | CLONE_NEWUTS
    // Use safe_clone() wrapper to create child in new namespaces

    // TODO 4: Add child process to cgroup
    // Hint: Call cgroups_add_process(container->cgroup_path, child_pid)
    // This enforces resource limits on the container process tree

    // TODO 5: Set up network connectivity from host side
```

```
    // Hint: Call network_setup_host_side(child_pid, config->bridge_name)
    // This creates veth pair and configures bridge connection

    // TODO 6: Store namespace file descriptors for cleanup
    // Hint: Open /proc/child_pid/ns/* files and store in container->namespace_fds
    // These allow cleanup even if child process exits

    container->child_pid = child_pid;
    return 0;
}

int container_wait(container_instance_t *container) {
    int status;

    // TODO 1: Wait for container process to exit
    // Hint: Use waitpid(container->child_pid, &status, 0)
    // This blocks until the container's main process terminates

    // TODO 2: Handle zombie process reaping
    // Hint: Container may have spawned child processes that need reaping
    // Check for additional processes in the PID namespace

    // TODO 3: Return exit status from container process
    // Hint: Use WEXITSTATUS(status) if WIFEXITED(status)

    return 0;
}

int container_destroy(container_instance_t *container) {
```

```
    // TODO 1: Terminate container process if still running
    // Hint: Send SIGTERM, wait briefly, then SIGKILL if needed

    // TODO 2: Clean up network interfaces
    // Hint: Remove veth pair and bridge configuration

    // TODO 3: Remove cgroup and clean up hierarchy
    // Hint: Call cgroups_destroy(container->cgroup_path)

    // TODO 4: Close namespace file descriptors
    // Hint: Close all fds in container->namespace_fds array

    // TODO 5: Execute any registered cleanup functions
    // Hint: execute_cleanup(&container->cleanup)

    return 0;
}
```

Language-Specific Implementation Notes

Working with Linux namespaces and cgroups in C requires attention to several system-specific details:

Stack Management for clone():

```c
#define STACK_SIZE (1024 * 1024)  // 1MB stack

// Allocate stack growing downward (most architectures)
char *stack = malloc(STACK_SIZE);
char *stack_top = stack + STACK_SIZE;  // Point to high address
```

Namespace Flags Reference:

```c
// Common namespace combinations
#define CONTAINER_NS_FLAGS (CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWNET | \
                            CLONE_NEWUTS | CLONE_NEWIPC)

// For testing individual namespaces
#define PID_ONLY_FLAGS CLONE_NEWPID
#define MOUNT_ONLY_FLAGS CLONE_NEWNS
```

Cgroups Filesystem Paths:

```c
// Check which cgroups version is available
#define CGROUPS_V1_PATH "/sys/fs/cgroup"
#define CGROUPS_V2_PATH "/sys/fs/cgroup/unified"

// Container-specific cgroup naming
#define CONTAINER_CGROUP_PREFIX "container-basic"
```

Milestone Checkpoints

After implementing each milestone, verify the isolation is working correctly:

Milestone 1 - PID Namespace Checkpoint:

```bash
# Compile and run
make && ./container-basic /bin/bash

# Inside container, check process isolation
ps aux                    # Should only see processes in container
echo $$                   # Should show PID 1
cat /proc/sys/kernel/pid_max  # Should work (need /proc mounted)

# Outside container (different terminal)
ps aux | grep container-basic  # Should see real host PID
```

Expected behavior: Container process sees itself as PID 1 and can only see processes within its PID namespace. Host can see the container process with its real PID.

Milestone 2 - Mount Namespace Checkpoint:

```bash
# Run container with custom rootfs
./container-basic -r ./test-rootfs /bin/sh

# Inside container
ls /                    # Should see only container filesystem
mount | grep proc       # Should show container's /proc mount
touch /test-file        # Should not appear on host filesystem

# Outside container
ls /                    # Should see normal host filesystem
ls ./test-rootfs/       # Should see test-file if bind-mounted
```

Expected behavior: Container has completely different filesystem view. Changes inside container don't affect host filesystem unless explicitly bind-mounted.

Signs something is wrong:

- Container sees host filesystem: Mount namespace not created properly
- Container can't access /proc: Forgot to mount proc filesystem inside container
- pivot_root fails: Check that new root has old_root directory and is on different filesystem

Goals and Non-Goals

> **Milestone(s):** This section defines the scope for all milestones (1-4), establishing clear boundaries for what our basic container runtime will implement versus advanced features left for future development.

Mental Model: Building a Safe Playground

Think of our container system like designing a safe playground for children. We need to provide essential safety features - fences to keep kids in the right area (process isolation), separate sandboxes so they don't interfere with each other (namespaces), and rules about how many swings each child can use (resource limits). However, we're not building a full amusement park with roller coasters, water slides, and gift shops (advanced orchestration features). Our goal is a functional, safe playground that demonstrates the core safety mechanisms, not a commercial-grade entertainment complex.

Just as a playground designer must decide which safety features are essential versus which amenities can be added later, we must clearly define what isolation primitives are fundamental to container functionality versus what

advanced features would distract from learning the core concepts. This section establishes those boundaries to keep our implementation focused and educational.

Functional Goals

Our basic container runtime must provide the four fundamental pillars of container technology: process isolation, filesystem isolation, network isolation, and resource control. These represent the minimum viable container implementation that demonstrates how modern container runtimes like Docker and containerd work under the hood.

Core Isolation Capabilities

The primary functional goal is implementing **process isolation** through Linux namespaces. Our container must create isolated views of system resources so that processes inside the container cannot interfere with or observe processes on the host system or in other containers. This isolation forms the security boundary that makes containers useful for running untrusted code safely.

Isolation Type	Namespace	Goal Description
Process Tree	PID	Container processes see isolated PID numbering starting from PID 1
Filesystem	Mount	Container has private filesystem root with essential system directories
Network Stack	Network	Container has isolated network interfaces and routing tables
System Identity	UTS	Container can have different hostname without affecting host
Inter-Process Communication	IPC	Container has private message queues and shared memory segments

The **PID namespace** isolation must ensure that the container's init process appears as PID 1 within the container while the host sees the real process ID. This creates the fundamental process boundary that prevents container processes from sending signals to host processes or observing the full system process tree. The container's init process must properly handle zombie reaping responsibilities that normally belong to the system's init process.

Filesystem isolation through mount namespaces must provide a private filesystem view where the container can mount and unmount filesystems without affecting the host. Our implementation must use `pivot_root` to switch the container to a new filesystem root, mount essential filesystems like `/proc` and `/sys` inside the container, and ensure that mount operations inside the container don't propagate to the host system.

Network isolation through network namespaces must give the container its own network stack with private interfaces, routing tables, and firewall rules. The container should be able to bind to any port without conflicting with host services, and we must implement connectivity between the container and host using veth pairs connected to a bridge network.

Resource Control Requirements

The second pillar is **resource limitation** through Linux cgroups. Our container must prevent runaway processes from consuming all system resources and affecting other containers or host processes. This implements the "noisy

neighbor" protection essential for multi-tenant systems.

Resource Type	Control Mechanism	Goal Description
Memory	Memory Controller	Hard limit on RAM usage with OOM killing when exceeded
CPU	CPU Controller	Proportional CPU time allocation using quota and period
Processes	PIDs Controller	Maximum number of processes to prevent fork bombs
File Descriptors	Files Controller	Limit open file handles to prevent descriptor exhaustion

The memory controller must enforce hard limits where processes are killed by the Out-of-Memory (OOM) killer when they exceed their allocation. This prevents memory leaks or memory-intensive applications from consuming all system RAM. Our implementation must demonstrate both setting the limit and observing the enforcement behavior when limits are exceeded.

CPU control must implement proportional sharing using the CPU quota and period mechanism. Rather than hard CPU limits (which can cause performance issues), our container should receive a guaranteed minimum CPU allocation while being able to burst above that allocation when CPU is available. This provides predictable performance isolation without wasting resources.

Configuration and Lifecycle Management

Our container runtime must provide a clean configuration interface and proper lifecycle management. The configuration should be declarative, specifying desired resource limits and networking parameters without requiring users to understand the underlying namespace and cgroup implementation details.

Lifecycle Phase	Required Capabilities
Creation	Parse configuration, validate parameters, prepare filesystem
Startup	Create namespaces, setup cgroups, fork container process
Running	Monitor container process, handle signals, maintain isolation
Cleanup	Destroy namespaces, remove cgroups, cleanup filesystem mounts

The container creation process must validate that all specified parameters are achievable (sufficient memory available, valid network configuration, accessible root filesystem) before beginning namespace creation. This prevents partial container creation that leaves system resources in an inconsistent state.

Container cleanup must be robust and handle partial failure scenarios. If namespace creation fails partway through, the cleanup process must remove any successfully created namespaces and cgroups. This prevents resource leaks that could accumulate over time and affect system stability.

Error Handling and Recovery

Our container runtime must gracefully handle common failure scenarios and provide meaningful error messages that help users diagnose configuration problems. The most common failures involve insufficient privileges, missing

kernel features, and resource exhaustion.

Failure Category	Detection Method	Recovery Action
Insufficient Privileges	Check for CAP_SYS_ADMIN before namespace creation	Provide clear error about required capabilities
Missing Kernel Features	Test namespace creation with CLONE_* flags	Report which namespaces are unsupported
Resource Exhaustion	Monitor cgroup creation and process limits	Fail fast with resource availability information
Configuration Errors	Validate filesystem paths and network parameters	Report specific configuration problems

The error handling must distinguish between temporary failures (resource exhaustion) and permanent failures (missing kernel features). Temporary failures should be retryable, while permanent failures should provide guidance on system requirements or configuration changes needed.

Non-Goals

Clearly defining what we will NOT implement is equally important as defining our goals. These non-goals keep our implementation focused on learning core container isolation primitives rather than building a production container runtime with enterprise features.

Container Image Management

We will NOT implement container image formats, image layers, or image registries. Our container will work with a simple directory containing a root filesystem, not with Docker images, OCI images, or layered filesystems. This means no support for:

- Image pulling from registries like Docker Hub
- Image layer caching and deduplication
- Dockerfile parsing and image building
- Image vulnerability scanning
- Multi-architecture image support

The rationale is that image management is a separate concern from process isolation. Understanding how to create namespaces and cgroups is independent of understanding how to manage layered filesystems. Implementing image support would require overlay filesystems, cryptographic hash validation, and network protocols that would distract from the core learning objectives.

Users will need to prepare their own root filesystem directory containing the files and directory structure needed for their containerized application. This could be created by extracting an existing Docker image or by manually assembling the required files.

Container Orchestration and Clustering

We will NOT implement multi-container coordination, service discovery, or container orchestration features. Our runtime will manage single containers in isolation without any knowledge of other containers or cluster-wide policies. This excludes:

- Container scheduling across multiple hosts
- Service mesh and load balancing between containers
- Container health checks and automatic restart policies
- Volume management and persistent storage
- Secrets management and configuration distribution
- Rolling updates and blue-green deployment strategies

These features belong to container orchestration platforms like Kubernetes, Docker Swarm, or Nomad. Our basic container runtime serves as the foundation that orchestrators build upon, but implementing orchestration features would require distributed systems concepts (consensus, leader election, cluster membership) that are beyond our scope.

Advanced Security Features

We will NOT implement advanced security isolation beyond basic namespaces. Our security model relies on namespace isolation and assumes containers run with appropriate user privileges. We will not provide:

- User namespace mapping for rootless containers
- SELinux or AppArmor integration for mandatory access control
- Seccomp filtering to restrict system call access
- Container image signing and signature verification
- Runtime security monitoring and anomaly detection
- Network policy enforcement and micro-segmentation

While these features are important for production container security, they require deep knowledge of Linux security modules and cryptographic systems. Our basic implementation demonstrates the fundamental isolation mechanisms that these advanced features build upon.

Performance Optimization and Enterprise Features

We will NOT implement performance optimizations or enterprise-grade reliability features. Our focus is on correctness and educational value, not on production performance or scalability. This means no:

- Container startup time optimization through pre-warming or caching
- Resource usage monitoring and metrics collection
- Log aggregation and centralized logging infrastructure
- Backup and disaster recovery for container data
- High availability and failover mechanisms
- Performance profiling and resource usage analytics

These features would require sophisticated monitoring infrastructure, storage systems, and distributed computing concepts that would obscure the core container isolation mechanisms we're trying to understand.

> **Key Design Principle**: We prioritize understanding over features. Every component we implement must clearly demonstrate a core container isolation concept. Any feature that doesn't directly contribute to understanding namespaces, cgroups, or basic container lifecycle is explicitly out of scope.

Network Policy and Advanced Networking

While we implement basic container networking through veth pairs and bridge networking, we will NOT implement advanced networking features that production container platforms provide:

- Network policies and firewall rule management
- Service mesh integration with encrypted inter-container communication
- Load balancing and traffic routing between containers
- Network address translation (NAT) with port forwarding rules
- VLAN tagging and advanced network topologies
- Container network interface (CNI) plugin architecture

Our networking implementation demonstrates how containers achieve network isolation and basic connectivity. Advanced networking features require understanding of network protocols, traffic engineering, and distributed networking concepts that are separate from the core isolation mechanisms.

Storage and Filesystem Features

Beyond basic mount namespace isolation, we will NOT implement advanced storage features:

- Volume mounting with different filesystem types
- Container data persistence across container restarts
- Filesystem encryption and secure data handling
- Storage quotas and disk usage monitoring
- Snapshot and backup functionality for container filesystems
- Distributed storage integration

Our filesystem isolation demonstrates how containers achieve private filesystem views using mount namespaces and `pivot_root`. Storage management involves understanding filesystem drivers, storage protocols, and data durability concepts that would distract from namespace learning objectives.

Implementation Guidance

This subsection provides concrete technical recommendations for implementing the goals defined above while avoiding the complexity of features marked as non-goals.

Technology Recommendations

Component	Simple Option	Advanced Option
Namespace Creation	`clone()` system call with `CLONE_NEW*` flags	`unshare()` with separate process creation
Mount Management	Direct `mount()` and `pivot_root()` calls	`libmount` library for complex mount operations
Network Setup	Manual veth and bridge creation via netlink	CNI (Container Network Interface) plugins
Cgroups Interface	Direct filesystem writes to `/sys/fs/cgroup`	`libcgroup` library for abstraction
Configuration Format	Simple key-value text file or command line args	YAML/JSON with schema validation
Error Reporting	`errno` values with `perror()` messages	Structured error codes with detailed context

For learning purposes, choose the simple options to understand the underlying mechanisms directly. Advanced options abstract away the details we want to understand.

Recommended File Structure

Organize the codebase to separate concerns and make testing easier:

```
container-basic/
├── src/
│   ├── main.c                    ← CLI entry point and argument parsing
│   ├── container.c               ← Core container lifecycle (create/wait/destroy)
│   ├── container.h               ← Public API and data structure definitions
│   ├── namespaces.c              ← Namespace creation and management
│   ├── mounts.c                  ← Filesystem mounting and pivot_root
│   ├── network.c                 ← Network namespace and veth setup
│   ├── cgroups.c                 ← Resource limit configuration
│   ├── cleanup.c                 ← Error recovery and resource cleanup
│   └── utils.c                   ← Helper functions and system call wrappers
├── tests/
│   ├── test_isolation.c          ← Verify namespace isolation works
│   ├── test_resources.c          ← Verify cgroup limits are enforced
│   └── test_networking.c         ← Verify container networking
├── rootfs/                       ← Sample root filesystem for testing
│   ├── bin/
│   ├── etc/
│   └── proc/                     ← Will be mounted inside container
└── Makefile                      ← Build system with test targets
```

Core Data Structure Implementation

Implement the configuration and instance structures to match our naming conventions:

```c
// container.h - Core data structures


#include <sys/types.h>

#include <unistd.h>


#define STACK_SIZE (1024 * 1024)

#define CONTAINER_NS_FLAGS (CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWNET | CLONE_NEWUTS |
CLONE_NEWIPC)

#define CGROUPS_V1_PATH "/sys/fs/cgroup"

#define CGROUPS_V2_PATH "/sys/fs/cgroup/unified"

#define CONTAINER_CGROUP_PREFIX "container-basic"


// Container configuration - what the user specifies

typedef struct {

    char *rootfs_path;           // Path to container root filesystem

    char *hostname;              // Container hostname (UTS namespace)

    size_t memory_limit_bytes;   // Memory limit in bytes (0 = no limit)

    int cpu_percent;             // CPU percentage (0-100, 0 = no limit)

    int max_processes;           // Maximum process count (0 = no limit)

    char *bridge_name;           // Host bridge name for networking

    char *container_ip;          // IP address for container

    char **argv;                 // Command and arguments to run

    char **envp;                 // Environment variables

} container_config_t;


// Container instance - runtime state and cleanup handles

typedef struct {

    pid_t child_pid;             // PID of container init process

    int namespace_fds[6];        // File descriptors for namespaces (PID, mount, net, UTS,
IPC, user)

    char *cgroup_path;           // Path to container's cgroup directory
```

```
    struct cleanup_list *cleanup; // Linked list of cleanup functions
} container_instance_t;

// Cleanup function registration for error recovery
typedef struct cleanup_list {
    void (*cleanup_func)(void *); // Function to call for cleanup
    void *data;                   // Data to pass to cleanup function
    struct cleanup_list *next;    // Next cleanup function in list
} cleanup_list_t;
```

Core API Skeleton

Implement the main container API functions with detailed TODOs mapping to the milestone requirements:

```c
// container.c - Main container lifecycle implementation


/**

 * container_create - Create and start a new container

 * @config: Container configuration specifying resources and command

 * @container: Output parameter filled with container instance data

 *

 * Returns: 0 on success, -1 on failure (check errno for details)

 */

int container_create(const container_config_t *config, container_instance_t *container) {

    // TODO 1: Validate configuration parameters (rootfs_path exists, memory_limit reasonable, etc.)

    // TODO 2: Initialize cleanup list for error recovery

    // TODO 3: Create cgroup hierarchy and set resource limits (Milestone 4)

    // TODO 4: Prepare container root filesystem and create necessary directories

    // TODO 5: Allocate stack for clone() call (use STACK_SIZE constant)

    // TODO 6: Call safe_clone() with CONTAINER_NS_FLAGS and container init function

    // TODO 7: In parent: store child PID and namespace file descriptors

    // TODO 8: In parent: setup networking (create veth pair, configure bridge) (Milestone 3)

    // TODO 9: In parent: move container process into cgroup

    // TODO 10: Return success, container instance now contains cleanup handles

    // Error handling: call execute_cleanup() if any step fails

}


/**

 * container_wait - Wait for container process to exit

 * @container: Container instance to wait for

 *

 * Returns: Exit status of container process, -1 on error
```

```
 */
int container_wait(container_instance_t *container) {
    // TODO 1: Use waitpid() to wait for child_pid
    // TODO 2: Handle SIGCHLD signals appropriately
    // TODO 3: Return WEXITSTATUS() for normal exit, WTERMSIG() for signals
    // Hint: Use WIFEXITED() and WIFSIGNALED() to distinguish exit types
}

/**
 * container_destroy - Clean up all container resources
 * @container: Container instance to destroy
 *
 * Returns: 0 on success, -1 if some cleanup failed
 */
int container_destroy(container_instance_t *container) {
    // TODO 1: Send SIGTERM to container process if still running
    // TODO 2: Wait for graceful shutdown, then SIGKILL if necessary
    // TODO 3: Execute all registered cleanup functions
    // TODO 4: Close namespace file descriptors
    // TODO 5: Remove cgroup directory and all files
    // TODO 6: Free allocated memory in container instance
    // Note: Continue cleanup even if some steps fail, return overall status
}
```

System Call Wrapper Functions

Implement safe wrappers for system calls that provide better error reporting:

```c
// utils.c - System call wrappers with error handling

/**
 * safe_clone - Wrapper for clone() with error checking and logging
 */
pid_t safe_clone(int (*fn)(void *), void *stack, int flags, void *arg) {
    // TODO 1: Calculate stack top address (stack grows down on most architectures)
    // TODO 2: Call clone() with provided parameters
    // TODO 3: Check for -1 return value and log specific error
    // TODO 4: For EPERM errors, suggest checking capabilities
    // TODO 5: For EINVAL errors, check which CLONE_ flags are unsupported
    // Hint: Use strerror(errno) for human-readable error messages
}

/**
 * safe_mount - Wrapper for mount() with detailed error reporting
 */
int safe_mount(const char *source, const char *target, const char *filesystemtype,
               unsigned long mountflags, const void *data) {
    // TODO 1: Check that target directory exists before mounting
    // TODO 2: Call mount() with provided parameters
    // TODO 3: On failure, log source, target, and filesystem type
    // TODO 4: For EACCES, suggest checking permissions or capabilities
    // TODO 5: For ENODEV, suggest checking if filesystem type is supported
    // Hint: Print mount command equivalent for debugging
}

/**
 * register_cleanup - Add cleanup function to be called on error or exit
```

```
 */
void register_cleanup(cleanup_list_t **list, void (*func)(void *), void *data) {
    // TODO 1: Allocate new cleanup_list_t node
    // TODO 2: Set function pointer and data
    // TODO 3: Insert at head of linked list
    // TODO 4: Handle malloc() failure gracefully
}

/**
 * execute_cleanup - Call all registered cleanup functions in reverse order
 */
void execute_cleanup(cleanup_list_t *list) {
    // TODO 1: Traverse linked list and call each cleanup function
    // TODO 2: Pass stored data pointer to each function
    // TODO 3: Continue even if cleanup functions fail
    // TODO 4: Free cleanup_list_t nodes after calling functions
    // TODO 5: Log any errors but don't propagate them
}
```

Milestone Checkpoints

After implementing each milestone, verify the functionality with these specific tests:

Milestone 1 (PID Namespace):

```bash
# Compile and run basic PID namespace test
make test_pid_namespace
sudo ./test_pid_namespace

# Expected behavior:
# - Container process reports PID 1 when running `echo $$`
# - Host can see real PID (e.g., 12345) in process table
# - Container cannot see host processes in `ps` output
# - Container init properly reaps zombie child processes
```

Milestone 2 (Mount Namespace):

```bash
# Test filesystem isolation
make test_mount_namespace
sudo ./test_mount_namespace /tmp/test_rootfs

# Expected behavior:
# - Container sees different root filesystem than host
# - Container can mount /proc and see only container processes
# - Mount operations inside container don't affect host
# - Container cannot access host filesystem outside mounted volumes
```

Milestone 3 (Network Namespace):

```bash
# Test network isolation and connectivity
make test_network_namespace
sudo ./test_network_namespace

# Expected behavior:
# - Container has different network interfaces than host (ip addr show)
# - Container can ping its own IP address
# - Container can reach external networks through host bridge
# - Host can communicate with container through veth pair
```

Milestone 4 (Cgroups):

```bash
# Test resource limits enforcement
make test_cgroups
sudo ./test_cgroups --memory-limit=100M --cpu-percent=50

# Expected behavior:
# - Memory-intensive process gets OOM killed at 100MB
# - CPU usage stays around 50% under sustained load
# - Process count limited to configured maximum
# - Cgroup files show current resource usage
```

Debugging Tips

When container creation fails, use these debugging techniques:

Symptom	Likely Cause	How to Diagnose	Fix
clone() returns EPERM	Missing CAP_SYS_ADMIN	Run `getcap /path/to/binary` or check `id`	Run as root or add capabilities
mount() fails with EACCES	SELinux or permission issue	Check `dmesg` and `/var/log/audit/audit.log`	Disable SELinux or fix context
Container sees host processes	PID namespace not created	Check `/proc/self/ns/pid` in container	Verify CLONE_NEWPID flag
Network unreachable	veth pair misconfigured	Run `ip addr` and `ip route` in container	Check bridge and routing setup
Process killed immediately	Memory limit too low	Check `dmesg` for OOM killer messages	Increase memory limit or check usage
Cgroup creation fails	cgroups v2 vs v1 mismatch	Check `/proc/filesystems` for cgroup2	Use correct cgroup path and API

Use these inspection commands during debugging:

- `lsns` - List all namespaces and their processes
- `nsenter -t <pid> -p -n -m <command>` - Enter container namespaces
- `cat /proc/<pid>/cgroup` - Show process cgroup membership
- `cat /sys/fs/cgroup/memory/<cgroup>/memory.usage_in_bytes` - Check memory usage

High-Level Architecture

> **Milestone(s):** This section provides architectural foundation for all milestones (1-4), establishing the component structure and relationships needed to implement PID namespaces, mount namespaces, network namespaces, and cgroups resource management.

Component Overview

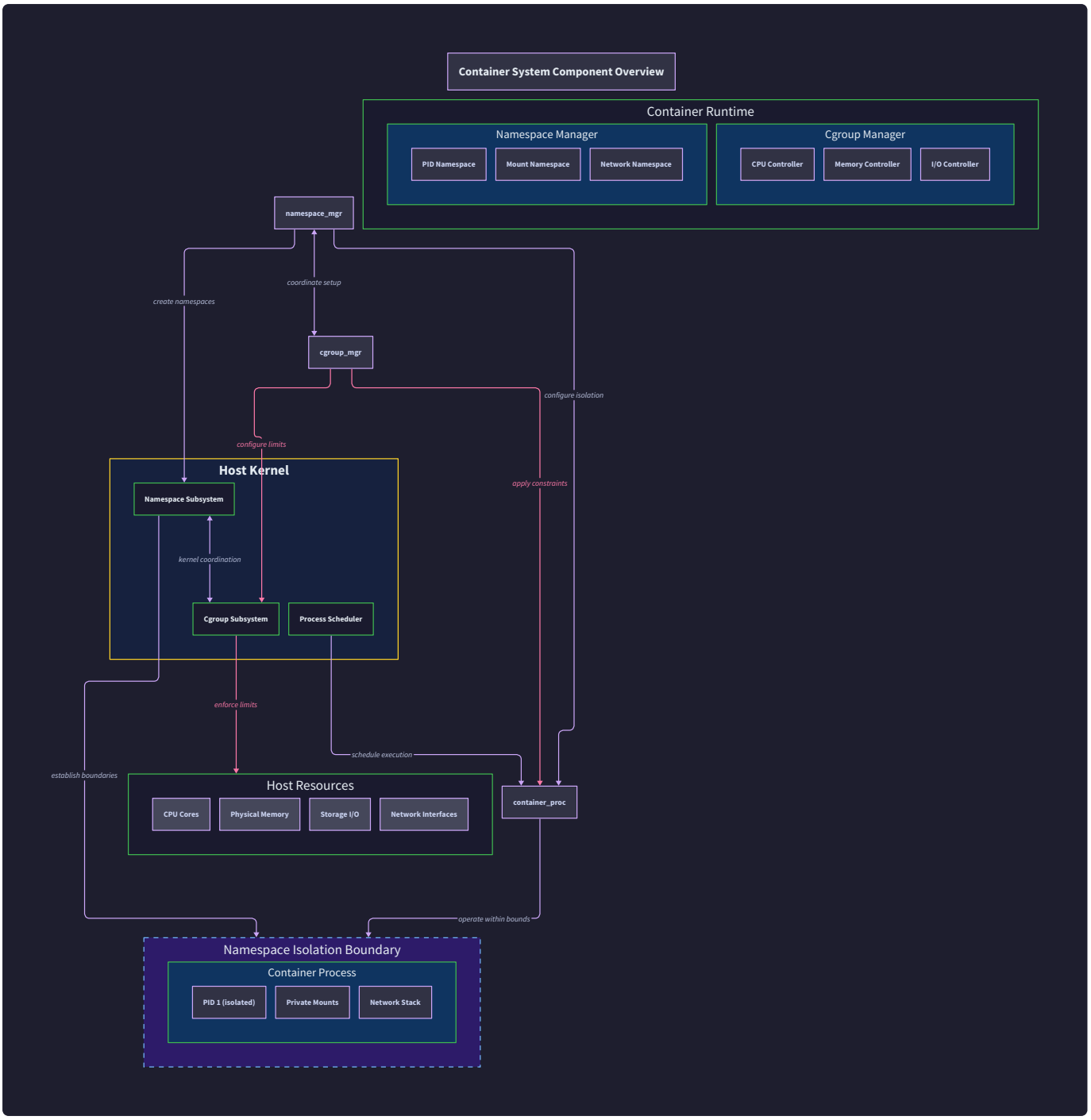
Think of our container runtime as a **theater production company** that needs to set up completely isolated stages for different performances. Just as a theater company has specialized departments—set design (mount namespace), lighting and sound (network), casting and direction (process management), and budget control (resource limits)—our container runtime has specialized managers that coordinate to create perfect isolation for each "performance" (container).

Our container runtime architecture centers around four core managers that work together to create process isolation. Each manager has a specific responsibility and operates semi-independently, but they must coordinate carefully during container startup and cleanup to avoid timing issues and resource leaks.

The **namespace managers** handle different aspects of process isolation. The PID namespace manager creates isolated process trees where the container's init process appears as PID 1. The mount namespace manager creates isolated filesystem views and handles the complex pivot_root operation to switch the container's root directory. The network namespace manager creates isolated network stacks and sets up connectivity through virtual ethernet pairs.

The **cgroup manager** operates orthogonally to the namespace managers, focusing purely on resource limits rather than isolation. It creates cgroup hierarchies and assigns the container process to appropriate cgroups with memory, CPU, and process limits configured.

These managers are coordinated by a **container orchestrator** that handles the precise sequencing required during container creation. This orchestration is critical because namespace creation must happen in a specific order, and cgroup assignment must occur at exactly the right moment in the process lifecycle.



The following table describes each major component and its primary responsibilities:

Component	Primary Responsibility	Key Operations	Dependencies
Container Orchestrator	Coordinates startup/cleanup sequence	`container_create()`, `container_wait()`, `container_destroy()`	All managers
PID Namespace Manager	Process isolation and PID 1 duties	`clone()` with `CLONE_NEWPID`, zombie reaping	None
Mount Namespace Manager	Filesystem isolation	`unshare(CLONE_NEWNS)`, `pivot_root()`, essential mounts	PID namespace
Network Namespace Manager	Network stack isolation	`unshare(CLONE_NEWNET)`, veth pair setup, bridge config	Mount namespace
Cgroup Manager	Resource limit enforcement	Cgroup creation, process assignment, limit configuration	PID namespace
Cleanup Manager	Resource cleanup and error recovery	Cleanup function registration and execution	All managers

> **Design Insight**: The key architectural challenge is that Linux kernel namespaces and cgroups were designed as independent mechanisms, but containers require them to work together seamlessly. Our architecture acknowledges this by keeping the managers loosely coupled but providing strong coordination through the orchestrator and cleanup system.

Container Orchestrator Responsibilities

The container orchestrator serves as the main entry point and coordination layer. It maintains the `container_instance_t` structure that tracks all active resources associated with a container. During startup, it sequences the manager operations to avoid race conditions—for example, ensuring the PID namespace exists before attempting cgroup assignment, or ensuring mount namespace isolation before setting up network connectivity.

The orchestrator also implements the **cleanup registration pattern**. Each manager registers cleanup functions as it creates resources, building a cleanup chain that can be executed in reverse order during normal shutdown or error recovery. This approach ensures that even if container creation fails halfway through, all successfully created resources get properly cleaned up.

Namespace Manager Coordination

The three namespace managers must coordinate carefully because kernel namespace operations have subtle ordering dependencies. The PID namespace manager always runs first because it fundamentally changes how process creation works—subsequent operations need to account for the new PID space. The mount namespace manager runs second because filesystem isolation affects where network configuration tools can find their

resources. The network namespace manager runs last because it often needs to access files and execute commands that depend on the filesystem isolation being in place.

Each namespace manager maintains file descriptors to the created namespaces in the `namespace_fds` array. These descriptors serve dual purposes: they keep the namespaces alive even if the initial process exits, and they provide handles for external tools like debuggers to enter the namespaces for inspection.

Cgroup Manager Integration

The cgroup manager operates differently from the namespace managers because cgroups work through filesystem operations rather than system calls. It creates directories in the cgroup filesystem hierarchy, writes configuration files to set limits, and assigns process IDs to the appropriate cgroup. Unlike namespaces, cgroups can be modified after container startup, allowing for dynamic resource limit adjustments.

The cgroup manager must handle both cgroups v1 and v2 hierarchies because different Linux distributions use different defaults. It detects the available cgroup version at startup and adapts its filesystem paths and configuration syntax accordingly.

> **Architecture Decision Record: Component Separation Strategy**
>
> - **Context**: We need to organize namespace and resource management code that involves complex kernel interactions and precise timing
> - **Options Considered**:
>   1. Monolithic container creation function with all operations inline
>   2. Separate managers with loose coupling and shared state
>   3. Separate managers with strict interfaces and no shared state
> - **Decision**: Separate managers with loose coupling through shared `container_instance_t` structure
> - **Rationale**: Each type of isolation (PID, mount, network, cgroups) has distinct kernel APIs and failure modes that benefit from specialized handling, but they need to coordinate timing and share cleanup responsibilities
> - **Consequences**: More complex orchestration logic but much better testability, debugging, and maintainability of individual isolation mechanisms

The following table compares the architectural options:

Option	Pros	Cons	Chosen?
Monolithic Function	Simple control flow, no coordination complexity	Impossible to test individual mechanisms, difficult to debug failures	No
Loose Coupling	Easy coordination, shared cleanup, simple interfaces	Some shared state complexity	**Yes**
Strict Separation	Perfect isolation, no shared state	Complex coordination, duplicate cleanup logic, rigid interfaces	No

Recommended File Structure

Organizing the codebase properly from the beginning prevents the common mistake of putting all container logic in a single massive file. Our file structure reflects the component architecture and makes it easy to test individual managers in isolation while maintaining clear dependencies.

```
container-basic/
├── src/
│   ├── main.c                    ← Entry point, argument parsing, basic container operations
│   ├── container.h               ← Main public API and shared data structures
│   ├── container.c               ← Container orchestrator implementation
│   ├── namespaces/
│   │   ├── pid_namespace.h       ← PID namespace manager interface
│   │   ├── pid_namespace.c       ← PID namespace implementation, zombie reaping
│   │   ├── mount_namespace.h     ← Mount namespace manager interface
│   │   ├── mount_namespace.c     ← Mount namespace, pivot_root, essential mounts
│   │   ├── network_namespace.h   ← Network namespace manager interface
│   │   └── network_namespace.c   ← Network namespace, veth pairs, bridge setup
│   ├── cgroups/
│   │   ├── cgroup_manager.h      ← Cgroup manager interface
│   │   ├── cgroup_manager.c      ← Cgroup creation, limits, process assignment
│   │   ├── cgroup_v1.c          ← Cgroups v1 specific implementation
│   │   └── cgroup_v2.c          ← Cgroups v2 specific implementation
│   ├── utils/
│   │   ├── cleanup.h            ← Cleanup registration and execution
│   │   ├── cleanup.c            ← Cleanup manager implementation
│   │   ├── syscall_wrappers.h   ← Safe syscall wrappers with error handling
│   │   ├── syscall_wrappers.c   ← Implementation of safe_clone, safe_mount, etc.
│   │   └── logging.h            ← Logging utilities for debugging
│   └── logging.c                ← Logging implementation
├── tests/
│   ├── test_container.c         ← Integration tests for full container lifecycle
│   ├── test_pid_namespace.c     ← Unit tests for PID isolation
│   ├── test_mount_namespace.c   ← Unit tests for filesystem isolation
│   ├── test_network_namespace.c ← Unit tests for network isolation
│   ├── test_cgroups.c           ← Unit tests for resource limits
│   └── test_utils.c             ← Unit tests for utility functions
├── examples/
│   ├── simple_container.c       ← Basic usage example
│   ├── resource_limited.c       ← Example with cgroup limits
│   └── networked_container.c    ← Example with network connectivity
├── rootfs/                      ← Example root filesystem for testing
│   ├── bin/                     ← Basic utilities (busybox, etc.)
│   ├── etc/                     ← Configuration files
│   └── proc/                    ← Empty directory for /proc mount
├── Makefile                     ← Build configuration
└── README.md                    ← Build and usage instructions
```

Header File Organization

The header file structure follows a clear dependency hierarchy. The main `container.h` defines the core data structures (`container_config_t`, `container_instance_t`) and orchestrator functions. Individual manager headers define only their specific interfaces and can be included independently for unit testing.

Each manager header includes comprehensive documentation comments describing the expected call sequences and error conditions. For example, `pid_namespace.h` documents that PID namespace creation must happen before any other namespace operations because it affects subsequent `clone()` calls.

Source File Responsibilities

Each source file has a single, well-defined responsibility. The `container.c` file implements only the orchestration logic—sequencing manager operations and coordinating cleanup. Individual manager files implement only their specific kernel interactions without knowledge of other managers.

The `utils/` directory contains all the infrastructure code that handles error-prone system calls. The `syscall_wrappers.c` file implements safe versions of `clone()`, `mount()`, `unshare()`, and other system calls with comprehensive error checking and logging. This centralization makes it easier to handle the subtle differences in error codes and failure modes across different Linux kernel versions.

> **Architecture Decision Record: File Organization Strategy**
>
> - **Context**: Container implementation involves multiple complex kernel APIs that need to work together but should be testable in isolation
> - **Options Considered**:
>   1. Single file with all container logic
>   2. Separate files by namespace type with shared utilities
>   3. Layered architecture with clear abstraction boundaries
> - **Decision**: Separate files by namespace type with centralized utilities and clear interfaces
> - **Rationale**: Each namespace type has distinct kernel APIs and testing requirements, but they share common patterns for error handling and cleanup that benefit from centralization
> - **Consequences**: More files to manage but much better testability, easier debugging of specific namespace issues, and cleaner separation of concerns

The following table compares the file organization approaches:

Approach	Maintainability	Testability	Learning Curve	Chosen?
Single File	Low - everything mixed together	Very low - can't test parts	Easy - everything in one place	No
By Namespace	High - clear separation	High - isolated testing	Moderate - need to understand structure	**Yes**
Layered	Highest - strict abstractions	Moderate - complex mocking	Hard - many abstraction layers	No

Build and Test Organization

The `Makefile` supports building individual components for testing, which is crucial during development. For example, `make test-pid` builds and runs only the PID namespace tests, allowing rapid iteration on that specific

component without waiting for the full test suite.

The test directory mirrors the source structure, with unit tests for each manager and integration tests for the full container lifecycle. Each test file includes both positive tests (verifying correct behavior) and negative tests (verifying proper error handling for various failure scenarios).

⚠️ **Pitfall: Circular Dependencies** A common mistake is creating circular dependencies between managers. For example, having the mount namespace manager call cgroup functions, which then call mount functions. Our architecture prevents this by having all coordination flow through the container orchestrator, with managers only calling utility functions and never calling each other directly.

⚠️ **Pitfall: Inadequate Error Propagation** Another common issue is not properly propagating error information up through the component layers. Each manager must translate low-level system call errors into meaningful error messages that include context about what operation was being attempted. The utility functions help with this by providing detailed error logging, but each manager must add its own contextual information.

Implementation Guidance

The container implementation requires careful coordination of several Linux kernel features, each with their own complexity and failure modes. This guidance provides concrete starting points and file organization to help you build a maintainable implementation.

Technology Recommendations

Component	Simple Option	Advanced Option
Build System	Simple Makefile with basic targets	Autotools or CMake with feature detection
Error Handling	errno checking with perror()	Custom error types with detailed context
Logging	printf to stderr	Structured logging with levels and timestamps
Testing	Basic assert() macros	Full unit test framework like Check or Unity
Process Management	Direct clone() syscalls	Higher-level process spawning library
Filesystem Operations	Direct mount() syscalls	libmount for mount table management
Network Configuration	Direct netlink socket operations	libnl for network namespace management

Essential Data Structures

Here are the complete type definitions that all components will use:

```c
// container.h - Core data structures and API


#include <sys/types.h>

#include <unistd.h>

#include <stddef.h>


#define STACK_SIZE (1024 * 1024)

#define CONTAINER_NS_FLAGS (CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWNET | CLONE_NEWUTS |
CLONE_NEWIPC)

#define CGROUPS_V1_PATH "/sys/fs/cgroup"

#define CGROUPS_V2_PATH "/sys/fs/cgroup/unified"

#define CONTAINER_CGROUP_PREFIX "container-basic"


// Container configuration - all parameters needed to create a container

typedef struct {

    char *rootfs_path;           // Path to root filesystem directory

    char *hostname;              // Hostname to set inside container

    size_t memory_limit_bytes;   // Memory limit in bytes (0 for no limit)

    int cpu_percent;             // CPU percentage limit (0-100, 0 for no limit)

    int max_processes;           // Maximum number of processes (0 for no limit)

    char *bridge_name;           // Host bridge name for networking (NULL for no networking)

    char *container_ip;          // IP address to assign to container (NULL for auto)

    char **argv;                 // Command and arguments to run in container

    char **envp;                 // Environment variables for container process

} container_config_t;


// Active container instance - tracks all resources that need cleanup

typedef struct {

    pid_t child_pid;             // PID of container init process

    int namespace_fds[6];        // File descriptors for each namespace type
```

```
    char *cgroup_path;           // Full path to container's cgroup directory
    struct cleanup_list *cleanup; // Registered cleanup functions
} container_instance_t;

// Cleanup function registration for error recovery
typedef struct cleanup_list {
    void (*cleanup_func)(void *data);
    void *data;
    struct cleanup_list *next;
} cleanup_list_t;

// Main container API
int container_create(const container_config_t *config, container_instance_t *container);
int container_wait(container_instance_t *container);
int container_destroy(container_instance_t *container);
```

Utility Functions Implementation

```c
// utils/syscall_wrappers.c - Safe syscall wrappers with comprehensive error handling

#define _GNU_SOURCE
#include <sys/wait.h>
#include <sys/mount.h>
#include <sched.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

// Safe wrapper for clone() syscall with proper error checking
pid_t safe_clone(int (*fn)(void *), void *stack, int flags, void *arg) {
    pid_t pid = clone(fn, stack, flags, arg);
    if (pid == -1) {
        fprintf(stderr, "clone() failed: %s (flags=0x%x)\n", strerror(errno), flags);
        // Common clone failures and their meanings:
        // EINVAL: Invalid flags combination or stack pointer
        // EPERM: Insufficient privileges for namespace creation
        // ENOSPC: Kernel ran out of PIDs or namespace resources
        // EUSERS: Too many nested namespaces
        return -1;
    }
    return pid;
}

// Safe wrapper for mount() syscall with detailed error reporting
int safe_mount(const char *source, const char *target, const char *type,
```

```
                unsigned long flags, const void *data) {
    if (mount(source, target, type, flags, data) != 0) {
        fprintf(stderr, "mount('%s', '%s', '%s', 0x%lx) failed: %s\n",
                source ? source : "NULL",
                target ? target : "NULL",
                type ? type : "NULL",
                flags, strerror(errno));
        // Common mount failures:
        // ENOENT: Source or target path doesn't exist
        // EPERM: Insufficient privileges
        // EBUSY: Target is busy (already mounted or in use)
        // EINVAL: Invalid filesystem type or flags
        return -1;
    }
    return 0;
}

// Initialize cleanup list
void cleanup_list_init(cleanup_list_t **list) {
    *list = NULL;
}

// Register cleanup function (called in reverse order during cleanup)
int register_cleanup(cleanup_list_t **list, void (*func)(void *), void *data) {
    cleanup_list_t *entry = malloc(sizeof(cleanup_list_t));
    if (!entry) {
        return -1;
    }
    entry->cleanup_func = func;
```

```
    entry->data = data;
    entry->next = *list;
    *list = entry;
    return 0;
}

// Execute all registered cleanup functions in reverse registration order
void execute_cleanup(cleanup_list_t **list) {
    cleanup_list_t *current = *list;
    while (current) {
        cleanup_list_t *next = current->next;
        if (current->cleanup_func) {
            current->cleanup_func(current->data);
        }
        free(current);
        current = next;
    }
    *list = NULL;
}
```

Container Orchestrator Skeleton

```c
// container.c - Main orchestrator implementation skeleton

#include "container.h"
#include "namespaces/pid_namespace.h"
#include "namespaces/mount_namespace.h"
#include "namespaces/network_namespace.h"
#include "cgroups/cgroup_manager.h"
#include "utils/cleanup.h"
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>

int container_create(const container_config_t *config, container_instance_t *container) {
    // TODO 1: Initialize container instance structure
    //   - Set all fields to safe defaults (child_pid = 0, namespace_fds = -1, etc.)
    //   - Initialize cleanup list for error recovery
    //   - Validate config parameters (rootfs_path exists, limits are reasonable)

    // TODO 2: Create PID namespace and fork container init process
    //   - Call pid_namespace_create() to set up PID isolation
    //   - Store child PID in container->child_pid
    //   - Register PID cleanup function for process termination
    //   - Handle clone() failures and provide meaningful error messages

    // TODO 3: Set up cgroups for resource limits (parent process context)
    //   - Call cgroup_create() with memory, CPU, and process limits from config
    //   - Assign child process to cgroup using container->child_pid
    //   - Store cgroup path in container->cgroup_path for later cleanup
```

```
    //   - Register cgroup cleanup function

    // TODO 4: Wait for child process to signal setup completion
    //   - Use pipe or other IPC mechanism to synchronize with child
    //   - Child will set up mount and network namespaces, then signal ready
    //   - Handle case where child fails during setup (read error status)

    // TODO 5: Store namespace file descriptors for external access
    //   - Open /proc/[child_pid]/ns/* files to keep namespaces alive
    //   - Store FDs in container->namespace_fds array
    //   - Register FD cleanup functions to close on container destruction

    return 0; // Success - container is running
}

int container_wait(container_instance_t *container) {
    // TODO 1: Validate container instance
    //   - Check that container->child_pid is valid (> 0)
    //   - Return error if container is not in running state

    // TODO 2: Wait for container process to exit
    //   - Use waitpid() to wait for container->child_pid
    //   - Handle EINTR (interrupted system call) by retrying
    //   - Extract exit status and return to caller

    // TODO 3: Update container state to reflect process exit
    //   - Set container->child_pid = 0 to indicate process is gone
    //   - Keep other resources (namespaces, cgroups) for potential inspection
```

```
    return 0; // Return exit status of container process
}

int container_destroy(container_instance_t *container) {
    // TODO 1: Terminate container process if still running
    //   - Send SIGTERM to container->child_pid if > 0
    //   - Wait briefly for graceful shutdown
    //   - Send SIGKILL if process doesn't exit cleanly

    // TODO 2: Execute all registered cleanup functions
    //   - Call execute_cleanup() to run cleanup chain in reverse order
    //   - This handles cgroup removal, namespace FD closing, etc.
    //   - Continue cleanup even if individual steps fail

    // TODO 3: Reset container instance to safe state
    //   - Set all fields back to defaults
    //   - Free any dynamically allocated strings (cgroup_path)
    //   - Clear namespace_fds array

    return 0; // Success - all resources cleaned up
}
```

File Organization Implementation

Create the directory structure shown above, starting with these essential files:

Makefile:

```makefile
CC = gcc
CFLAGS = -Wall -Wextra -std=c99 -D_GNU_SOURCE
SRCDIR = src
TESTDIR = tests
OBJDIR = obj

# Core source files
SOURCES = $(SRCDIR)/container.c \
          $(SRCDIR)/namespaces/pid_namespace.c \
          $(SRCDIR)/namespaces/mount_namespace.c \
          $(SRCDIR)/namespaces/network_namespace.c \
          $(SRCDIR)/cgroups/cgroup_manager.c \
          $(SRCDIR)/utils/cleanup.c \
          $(SRCDIR)/utils/syscall_wrappers.c

OBJECTS = $(SOURCES:$(SRCDIR)/%.c=$(OBJDIR)/%.o)
TARGET = container-basic

all: $(TARGET)

$(TARGET): $(OBJECTS) $(SRCDIR)/main.c
        $(CC) $(CFLAGS) -o $@ $^

# Individual component tests
test-pid: $(TESTDIR)/test_pid_namespace.c $(SRCDIR)/namespaces/pid_namespace.c
        $(CC) $(CFLAGS) -o test_pid $^ && ./test_pid

test-mount: $(TESTDIR)/test_mount_namespace.c $(SRCDIR)/namespaces/mount_namespace.c
        $(CC) $(CFLAGS) -o test_mount $^ && ./test_mount

clean:
        rm -rf $(OBJDIR) $(TARGET) test_*

.PHONY: all clean test-pid test-mount
```

Milestone Checkpoints

After implementing the basic architecture structure, verify these behaviors:

1. **File Structure Checkpoint**: All header files compile without errors when included individually. Run `gcc -c src/container.h` and similar commands for each header.
2. **Data Structure Checkpoint**: Create a simple test program that initializes `container_config_t` and `container_instance_t` structures. Verify all fields are accessible and have expected types.
3. **Utility Functions Checkpoint**: Test the syscall wrappers with simple operations like creating a directory and mounting tmpfs. Verify error messages are helpful and cleanup functions execute properly.
4. **Build System Checkpoint**: The Makefile should build the project without warnings. Individual component tests should compile (but may not pass until components are implemented).

Common Implementation Pitfalls

⚠️ **Pitfall: Header Include Guards** Each header file must have proper include guards to prevent multiple inclusion problems. Use the pattern:

```c
#ifndef CONTAINER_COMPONENT_H

#define CONTAINER_COMPONENT_H

// ... header content ...

#endif
```

⚠️ **Pitfall: Memory Management in Error Paths** The cleanup system only handles resources that were successfully registered. If allocation fails during container creation, you may leak partially created resources. Always check allocation results and register cleanup functions immediately after successful resource creation.

⚠️ **Pitfall: File Descriptor Leaks** Namespace file descriptors and other FDs must be properly closed during cleanup. Initialize the `namespace_fds` array to -1 values and check for valid FDs before closing in cleanup functions.

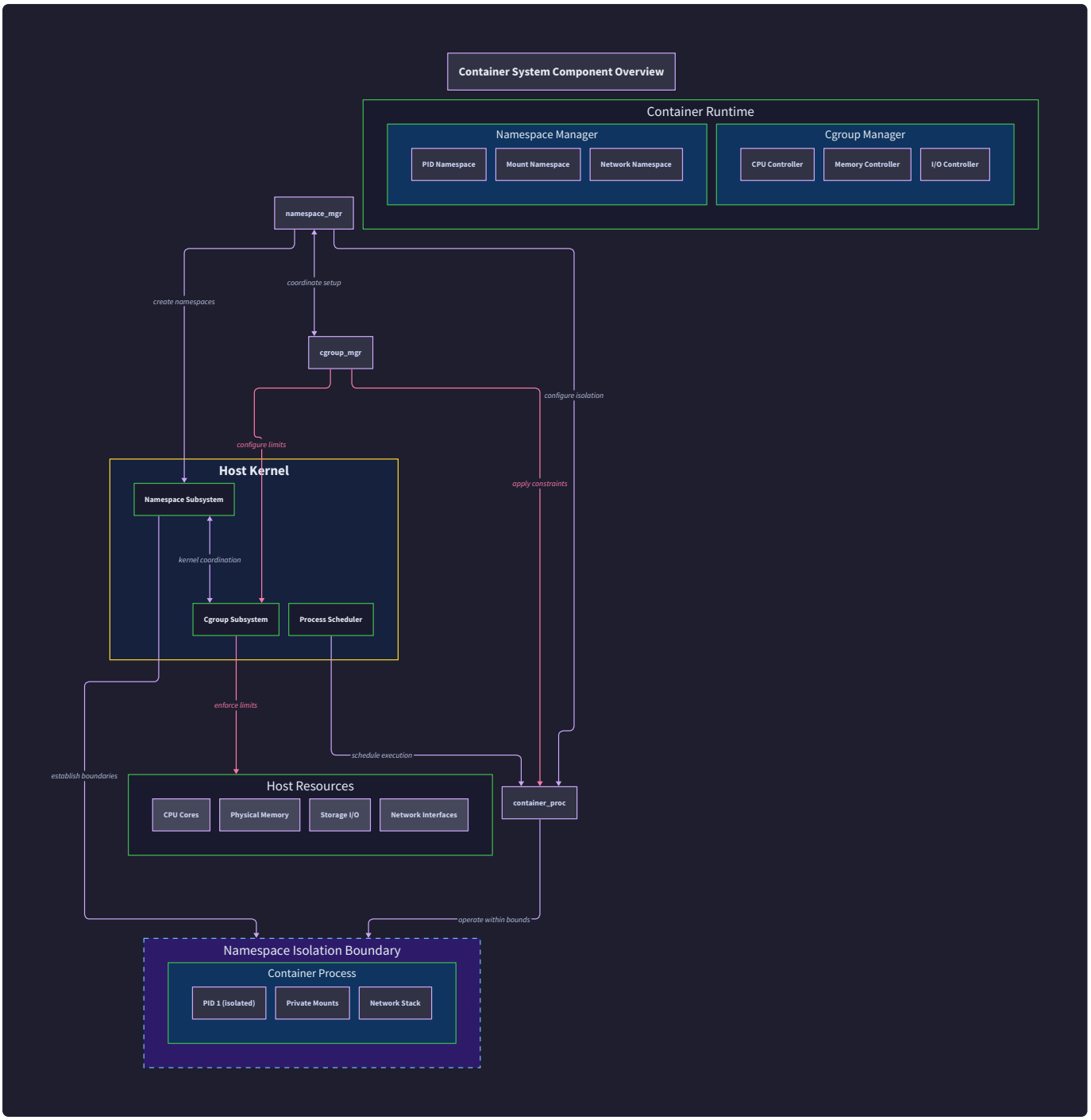
⚠️ **Pitfall: Inconsistent Error Codes** Different components should use consistent error code conventions. Consider defining an enum of container-specific error codes rather than returning raw errno values, which can be ambiguous.

Data Model

> **Milestone(s):** This section provides the foundational data structures for all milestones (1-4), establishing how container configuration, namespace handles, and resource limits are represented and managed throughout the container lifecycle.

The data model forms the backbone of our container runtime, defining how we represent container configurations, track active namespaces, and manage resource limits. Think of the data model as the **blueprint and inventory system** for a construction project - just as a builder needs detailed blueprints specifying what to build and an inventory system tracking materials and tools, our container runtime needs structured data representing what isolation to create and tracking mechanisms for the resources we allocate.

The data model addresses several critical challenges in container management. First, we need a comprehensive **configuration structure** that captures all the parameters required to create a container - from filesystem paths and resource limits to network configuration and process arguments. Second, we need **runtime tracking structures** that maintain handles to active namespaces, cgroup hierarchies, and cleanup responsibilities. Finally, we need **error recovery mechanisms** that ensure proper cleanup even when container creation partially fails.



The key insight driving our data model design is the **separation between intent and state**. The configuration structures represent what we want to create (intent), while the instance structures track what we have actually created (state). This separation enables robust error handling - if namespace creation fails halfway through, we know exactly what resources need cleanup by examining the instance state, regardless of what the original configuration requested.

Container Configuration

The `container_config_t` structure serves as the **master specification** for container creation, analogous to an architect's detailed building plans that specify every aspect of construction before work begins. This structure captures all user intentions about how the container should be configured, from filesystem isolation to resource limits to network connectivity.

Think of container configuration as a **restaurant order** - it specifies exactly what the customer wants (filesystem location, memory limits, network setup) without containing any information about kitchen state (which pots are being used, which burners are occupied). The configuration is pure intent, completely separate from the runtime state needed to fulfill that intent.

The configuration structure addresses the **complexity of container parameterization** by providing a single, comprehensive data structure that components can reference throughout the container lifecycle. Rather than passing dozens of individual parameters between functions, we pass a single configuration reference that contains everything needed for namespace creation, resource limit setup, and process execution.

> **Design Principle: Configuration Immutability** Once created, configuration structures should never be modified. This immutability enables safe sharing between threads and processes during container creation, and ensures that all components operate on consistent parameters even if creation takes significant time.

Field	Type	Purpose
`rootfs_path`	`char*`	Absolute path to container's root filesystem directory; used as new root after pivot_root operation
`hostname`	`char*`	Container hostname visible inside UTS namespace; appears in /proc/sys/kernel/hostname within container
`memory_limit_bytes`	`size_t`	Maximum memory consumption in bytes; enforced through memory cgroup controller
`cpu_percent`	`int`	CPU usage limit as percentage (0-100); translated to cgroup cpu.cfs_quota_us and cpu.cfs_period_us
`max_processes`	`int`	Maximum number of processes/threads; enforced through pids cgroup controller
`bridge_name`	`char*`	Host bridge name for container networking; veth pair connects container to this bridge
`container_ip`	`char*`	IP address assigned to container's network interface; must be within bridge subnet range
`argv`	`char**`	Null-terminated array of command-line arguments for container's main process; argv[0] is executable path
`envp`	`char**`	Null-terminated array of environment variables in "KEY=value" format; defines container process environment

The **filesystem configuration** through `rootfs_path` represents one of the most critical container parameters. This path must point to a complete filesystem hierarchy containing all binaries, libraries, and configuration files needed by the container process. The path serves as the target for the pivot_root operation that switches the container's view of the filesystem root.

> **Critical Requirement: Rootfs Completeness** The `rootfs_path` directory must contain a complete, bootable filesystem including essential directories (/bin, /lib, /etc, /proc, /sys) and all dependencies for the target executable. Missing libraries or incorrect permissions will cause container startup failures that are difficult to debug.

The **resource limit fields** (`memory_limit_bytes`, `cpu_percent`, `max_processes`) translate directly into cgroup controller configurations. The memory limit becomes a hard boundary enforced by the kernel - processes exceeding this limit trigger OOM (Out of Memory) killer behavior. The CPU percentage translates into a quota system where the container receives the specified percentage of CPU time over each scheduling period.

The **networking configuration** (`bridge_name`, `container_ip`) defines how the container connects to the host network. The bridge name must reference an existing host bridge interface, and the container IP must fall within the bridge's subnet range. These parameters drive the creation of veth pair interfaces that connect the container's network namespace to the host bridge.

> **Architecture Decision: Static vs Dynamic IP Assignment**
>
> - **Context**: Containers need IP addresses for network communication, but we must choose between static assignment (user specifies IP) and dynamic assignment (runtime allocates from pool)
> - **Options Considered**:
>   - Static IP specification in configuration
>   - Dynamic allocation from subnet pool
>   - DHCP-based assignment within namespace
> - **Decision**: Static IP specification in configuration
> - **Rationale**: Simplifies implementation by avoiding IP allocation logic and subnet management. Users have full control over network topology. Reduces startup complexity and potential IP conflicts.
> - **Consequences**: Users must manage IP assignment manually. Risk of IP conflicts if multiple containers specify same address. No automatic network planning capabilities.

Configuration Approach	Pros	Cons
Static IP Assignment	Simple implementation, predictable addresses, user control	Manual IP management, potential conflicts
Dynamic Pool Allocation	Automatic conflict avoidance, easier multi-container setup	Complex allocation logic, subnet management
DHCP Integration	Industry standard, flexible configuration	Requires DHCP server, network complexity

The **process execution parameters** (`argv`, `envp`) define what actually runs inside the container once all namespaces are created. The argv array follows standard Unix conventions where argv[0] contains the executable

path and subsequent elements contain command-line arguments. The envp array provides the process environment, allowing injection of configuration through environment variables.

Namespace Handles

The `container_instance_t` structure represents the **runtime state** of an active container, analogous to a construction site's status board that tracks which workers are active, which equipment is deployed, and what cleanup tasks remain. While configuration represents intent, the instance structure represents reality - the actual kernel objects, file descriptors, and process IDs that implement container isolation.

Think of namespace handles as **safety deposit box keys** - each handle provides access to a specific kernel resource (namespace, cgroup, process) that must be carefully tracked and eventually returned. Losing a handle means losing the ability to properly clean up that resource, leading to kernel object leaks that accumulate over time.

The instance structure addresses the **complexity of partial failure recovery** by maintaining explicit references to every kernel object created during container setup. If container creation fails after successfully creating PID and mount namespaces but before network setup completes, the instance structure contains exactly the information needed to clean up the successfully created resources.

Field	Type	Purpose
`child_pid`	`pid_t`	Process ID of container's main process; used for signaling and wait operations
`namespace_fds`	`int[6]`	File descriptors for namespace handles; indexed by namespace type for cleanup access
`cgroup_path`	`char*`	Absolute path to container's cgroup directory; used for resource limit cleanup
`cleanup`	`cleanup_list_t*`	Linked list of cleanup functions; executed in reverse order during container destruction

The **child process tracking** through `child_pid` provides the primary handle for container lifecycle management. This PID allows the parent process to send signals to the container (for graceful shutdown), wait for container exit, and retrieve exit status information. The PID also serves as the primary identifier for associating kernel resources with the specific container instance.

> **Critical Timing Issue: PID Reuse** Process IDs can be reused by the kernel after process exit. Always verify that the PID still refers to the expected process before performing operations. Use additional validation like checking /proc/[PID]/comm or process start time to ensure PID validity.

The **namespace file descriptor array** (`namespace_fds`) maintains handles to each namespace created for the container. The array indexes correspond to namespace types: PID (0), mount (1), network (2), UTS (3), IPC (4),

and user (5). These file descriptors enable namespace operations like entering the namespace from other processes or querying namespace properties through /proc filesystem interfaces.

Namespace Index	Type	File Descriptor Source	Cleanup Requirement
0	PID	`/proc/[PID]/ns/pid`	Close FD; namespace dies with process
1	Mount	`/proc/[PID]/ns/mnt`	Close FD; unmount private mounts
2	Network	`/proc/[PID]/ns/net`	Close FD; delete veth interfaces
3	UTS	`/proc/[PID]/ns/uts`	Close FD; no additional cleanup
4	IPC	`/proc/[PID]/ns/ipc`	Close FD; IPC objects cleaned by kernel
5	User	`/proc/[PID]/ns/user`	Close FD; no additional cleanup

The **cgroup path tracking** (`cgroup_path`) maintains the filesystem location of the container's cgroup hierarchy. This path serves multiple purposes: writing resource limits to controller files, adding processes to the cgroup, and removing the cgroup directory during cleanup. The path typically follows the pattern `/sys/fs/cgroup/[controller]/container-basic-[container-id]` .

> **Architecture Decision: Cleanup List vs Individual Cleanup Functions**
>
> - **Context**: Container creation involves many kernel objects that require cleanup in specific order. We need a mechanism to ensure proper cleanup even during partial failures.
> - **Options Considered**:
>   - Individual cleanup functions called manually
>   - Cleanup list with automatic execution
>   - Resource Acquisition Is Initialization (RAII) pattern
> - **Decision**: Cleanup list with automatic execution
> - **Rationale**: Provides guaranteed cleanup even during error conditions. Enables registration of cleanup functions as resources are created. Supports proper cleanup ordering through list reversal.
> - **Consequences**: Additional complexity in data structures. Memory overhead for cleanup list nodes. Clear separation between success and failure paths.

The **cleanup list mechanism** (`cleanup`) implements a sophisticated error recovery system that ensures proper resource cleanup regardless of where container creation fails. Each successful resource allocation registers a cleanup function that knows how to properly release that specific resource. During container destruction or error recovery, the cleanup list executes in reverse order, ensuring that dependencies are respected during teardown.

```
Registration Order: Create PID namespace -> Create mount namespace -> Create network -> Setup cgroups
Cleanup Order: Remove cgroups -> Cleanup network -> Cleanup mount -> Cleanup PID (automatic)
```

Cleanup Function Type	Registration Trigger	Cleanup Action	Failure Impact
`cleanup_cgroup_fn`	Cgroup directory creation	Remove cgroup hierarchy	Cgroup directory leak
`cleanup_network_fn`	Veth pair creation	Delete veth interfaces	Interface leak
`cleanup_mount_fn`	Mount namespace setup	Unmount private mounts	Mount table pollution
`cleanup_namespace_fd_fn`	Namespace FD opening	Close namespace file descriptors	File descriptor leak

The cleanup list structure enables **composable error handling** where each component registers its own cleanup requirements without needing knowledge of other components' cleanup needs. This separation of concerns ensures that adding new isolation mechanisms (like additional namespaces or cgroup controllers) doesn't require modifying existing cleanup logic.

Cleanup List Management

The `cleanup_list_t` structure implements a **reverse-order cleanup system** that ensures proper resource deallocation during both successful container shutdown and error recovery. Think of the cleanup list as a **stack of undo operations** - each time we successfully create a resource, we push an "undo" operation onto the stack. When cleanup time comes, we pop and execute each undo operation in reverse order.

This design pattern addresses the **dependency ordering problem** inherent in container resource management. Network interfaces must be cleaned up before network namespaces are destroyed. Cgroups must be emptied before directories can be removed. Mount points must be unmounted before mount namespaces exit. The cleanup list automatically handles these ordering requirements by executing cleanup functions in the reverse of their registration order.

Field	Type	Purpose
`cleanup_func`	`void(*)(void*)`	Function pointer to cleanup implementation; receives `data` parameter for resource-specific cleanup
`data`	`void*`	Opaque pointer to cleanup context; typically points to resource handles or configuration needed for cleanup
`next`	`struct cleanup_list*`	Pointer to next cleanup node; forms singly-linked list for traversal during execution

The **function pointer design** (`cleanup_func`) enables type-safe cleanup while maintaining flexibility for different resource types. Each cleanup function receives a generic data pointer that it casts to the appropriate type for its specific resource. This design avoids the need for complex union types or switch statements in the cleanup execution logic.

> **Memory Management Principle: Cleanup Data Ownership** The cleanup list takes ownership of the `data` pointers passed during registration. Cleanup functions are responsible for freeing both their resource-specific data and any dynamically allocated structures referenced by that data. This ownership transfer ensures memory cleanup even during error conditions.

Common cleanup function patterns follow a consistent structure: validate the data pointer, cast to the appropriate type, perform the resource-specific cleanup operation, and free any associated memory. The function should be idempotent - calling it multiple times should be safe, as error conditions may cause partial cleanup list execution.

The **linked list traversal** during cleanup execution walks the list from head to tail, executing each cleanup function and freeing the list node itself. This traversal pattern ensures that cleanup functions execute in reverse registration order while also cleaning up the list structure memory.

⚠️ **Pitfall: Cleanup Function Failures** Cleanup functions should never fail in ways that prevent subsequent cleanup operations. If a cleanup function encounters an error (like failing to delete a network interface), it should log the error but continue execution. Throwing exceptions or calling exit() from cleanup functions prevents other resources from being properly cleaned up, leading to resource leaks.

⚠️ **Pitfall: Circular Cleanup Dependencies** Avoid registering cleanup functions that depend on resources cleaned up by functions registered later. For example, don't register a mount namespace cleanup that requires network access if network cleanup is registered after mount cleanup. The reverse-order execution means network cleanup runs first, potentially breaking mount cleanup.

⚠️ **Pitfall: Double-Cleanup Registration** Never register cleanup functions for the same resource multiple times. This leads to double-free errors or resource cleanup conflicts. Use the instance structure's tracking fields to determine whether a resource has already been created and cleanup registered.

Implementation Guidance

This data model implementation provides the foundation for all container runtime operations, from initial configuration parsing through namespace creation to final resource cleanup. The structures bridge the gap between user intent and kernel resource management, providing type safety and error recovery throughout the container lifecycle.

Technology Recommendations

Component	Simple Option	Advanced Option
Configuration Parsing	`getopt()` + manual struct filling	`libconfig` or YAML parser with validation
Memory Management	Manual `malloc()/free()` with cleanup lists	`talloc` hierarchical memory management
String Handling	Standard C string functions with length checking	`bstring` library for safe string operations
Error Reporting	`errno` + `perror()` with custom error codes	Structured error codes with detailed context

Recommended File Structure

```
container-basic/
├── src/
│   ├── container.h              ← Public API and main data structures
│   ├── container.c              ← Main container lifecycle functions
│   ├── config/
│   │   ├── config.h             ← Configuration parsing and validation
│   │   ├── config.c             ← Container configuration management
│   │   └── config_test.c        ← Configuration parsing tests
│   ├── cleanup/
│   │   ├── cleanup.h            ← Cleanup list management
│   │   ├── cleanup.c            ← Cleanup execution and registration
│   │   └── cleanup_test.c       ← Cleanup mechanism tests
│   └── utils/
│       ├── utils.h              ← Common utilities and error handling
│       └── utils.c              ← Helper functions and error reporting
├── include/
│   └── container_types.h        ← Type definitions and constants
└── tests/
    ├── test_data_model.c        ← Data structure validation tests
    └── integration/
        └── test_container_lifecycle.c  ← End-to-end container tests
```

Core Data Structure Definitions

```c
#include <sys/types.h>
#include <unistd.h>

// Container configuration structure - immutable after creation
typedef struct {
    char* rootfs_path;           // Path to container root filesystem
    char* hostname;              // Container hostname for UTS namespace
    size_t memory_limit_bytes;   // Memory limit for memory cgroup
    int cpu_percent;             // CPU limit percentage (0-100)
    int max_processes;           // Process limit for pids cgroup
    char* bridge_name;           // Host bridge for container networking
    char* container_ip;          // IP address for container interface
    char** argv;                 // Command arguments for container process
    char** envp;                 // Environment variables for container
} container_config_t;

// Cleanup list node for resource management
typedef struct cleanup_list {
    void (*cleanup_func)(void*);  // Cleanup function pointer
    void* data;                   // Data for cleanup function
    struct cleanup_list* next;    // Next cleanup item
} cleanup_list_t;

// Container runtime instance - tracks active resources
typedef struct {
    pid_t child_pid;             // Container process PID
    int namespace_fds[6];        // File descriptors for namespaces
    char* cgroup_path;           // Path to container cgroup
```

```
    cleanup_list_t* cleanup;     // Cleanup function list
} container_instance_t;

// Container creation and management functions
int container_create(const container_config_t* config, container_instance_t* container);
int container_wait(container_instance_t* container);
int container_destroy(container_instance_t* container);

// Cleanup list management functions
void register_cleanup(cleanup_list_t** list, void (*func)(void*), void* data);
void execute_cleanup(cleanup_list_t** list);
```

Configuration Management Implementation

```c
// config/config.h
#ifndef CONFIG_H
#define CONFIG_H

#include "container_types.h"

// Configuration validation and creation
int config_create_default(container_config_t* config);
int config_validate(const container_config_t* config);
int config_parse_args(int argc, char** argv, container_config_t* config);
void config_destroy(container_config_t* config);

// Configuration validation helpers
int validate_rootfs_path(const char* path);
int validate_ip_address(const char* ip);
int validate_bridge_exists(const char* bridge);

#endif
```

Cleanup System Implementation

```c
// cleanup/cleanup.c


#include <stdlib.h>

#include <stdio.h>

#include "cleanup.h"


// Register cleanup function in reverse order for proper teardown

void register_cleanup(cleanup_list_t** list, void (*func)(void*), void* data) {

    // TODO: Allocate new cleanup_list_t node

    // TODO: Set cleanup_func to provided function pointer

    // TODO: Set data to provided data pointer

    // TODO: Set next to current list head (*list)

    // TODO: Update list head to point to new node

    // HINT: This creates a stack (LIFO) for reverse-order execution

}


// Execute all cleanup functions in reverse registration order

void execute_cleanup(cleanup_list_t** list) {

    // TODO: Walk the linked list from head to tail

    // TODO: For each node, call cleanup_func(data)

    // TODO: Free the current node before moving to next

    // TODO: Set list head to NULL after cleanup complete

    // HINT: Handle NULL function pointers gracefully

    // HINT: Continue cleanup even if individual functions fail

}


// Cleanup function for closing namespace file descriptors

void cleanup_namespace_fds(void* data) {

    // TODO: Cast data to int* (namespace_fds array)
```

```
    // TODO: Iterate through all 6 namespace types
    // TODO: Close any valid file descriptors (>= 0)
    // TODO: Set closed descriptors to -1 to prevent double-close
}

// Cleanup function for removing cgroup directory
void cleanup_cgroup(void* data) {
    // TODO: Cast data to char* (cgroup_path)
    // TODO: Remove cgroup directory using rmdir()
    // TODO: Free the cgroup_path string
    // TODO: Log errors but don't fail - other cleanup must continue
}
```

Instance Management Skeleton

```c
// container.c - Core lifecycle management

// Initialize container instance structure to safe defaults
int container_instance_init(container_instance_t* instance) {
    // TODO: Set child_pid to -1 (invalid PID)
    // TODO: Initialize all namespace_fds entries to -1
    // TODO: Set cgroup_path to NULL
    // TODO: Set cleanup list to NULL
    // TODO: Return 0 for success
}

// Create container with all namespaces and resource limits
int container_create(const container_config_t* config, container_instance_t* container) {
    // TODO: Initialize container instance to safe defaults
    // TODO: Validate configuration before starting creation
    // TODO: Create PID namespace with clone() or unshare()
    // TODO: Register cleanup for PID namespace resources
    // TODO: Create mount namespace and setup filesystem
    // TODO: Register cleanup for mount namespace resources
    // TODO: Create network namespace and veth pair
    // TODO: Register cleanup for network resources
    // TODO: Setup cgroups and apply resource limits
    // TODO: Register cleanup for cgroup resources
    // TODO: Execute container process in new namespaces
    // HINT: Use cleanup list for error recovery if any step fails
    // HINT: Store all resource handles in container instance
}
```

```
// Clean up all container resources using registered cleanup functions
int container_destroy(container_instance_t* container) {
    // TODO: Send SIGTERM to child process if still running
    // TODO: Wait brief period for graceful shutdown
    // TODO: Send SIGKILL if process still exists
    // TODO: Execute complete cleanup list
    // TODO: Reset container instance to safe defaults
    // HINT: Always execute cleanup even if process operations fail
}
```

Milestone Checkpoints

After Configuration Implementation:

- Create a simple test program that parses command-line arguments into `container_config_t`
- Verify that configuration validation catches invalid rootfs paths and IP addresses
- Test that configuration cleanup properly frees all allocated strings
- Expected output: Configuration parsing succeeds with valid inputs, fails gracefully with invalid inputs

After Cleanup System Implementation:

- Write unit tests that register multiple cleanup functions and verify reverse-order execution
- Test cleanup execution with some functions that intentionally fail
- Verify that cleanup list properly frees all allocated nodes
- Expected behavior: Cleanup functions execute in reverse registration order, failures don't prevent subsequent cleanup

After Instance Management:

- Create container instances and verify proper initialization
- Test partial container creation with cleanup verification
- Use `lsof` to verify no file descriptor leaks after container destruction
- Expected result: Clean creation and destruction with no resource leaks visible in `/proc` filesystem

Debugging Data Structure Issues

Symptom	Likely Cause	Diagnosis	Fix
Segfault during config access	Uninitialized config pointer	Check if config_create was called	Always initialize structures before use
Memory leak after container destroy	Missing cleanup registration	Use valgrind to identify leaked allocations	Register cleanup for every resource allocation
Container creation hangs	Blocking operation without error handling	Check for infinite waits in clone/mount	Add timeouts and error checking to all syscalls
Cleanup functions crash	Invalid data pointers in cleanup list	Verify data pointers before casting	Validate data != NULL in cleanup functions
Double-free errors	Multiple cleanup registrations	Check cleanup list for duplicate entries	Track registration state in instance structure

PID Namespace Component

> **Milestone(s):** This section corresponds to Milestone 1 (Process Namespace), which implements process isolation using PID namespaces and establishes the foundation for container process management.

Mental Model: Hotel Room Numbers

Think of PID namespaces like the room numbering systems in different buildings of a large hotel chain. Each hotel building has its own room numbering scheme - Building A has rooms 101, 102, 103, while Building B also has rooms 101, 102, 103. From inside Building A, you only see the room numbers for that building. Room 101 in Building A is completely different from Room 101 in Building B, even though they share the same number within their respective buildings.

Similarly, PID namespaces create isolated process numbering systems. The host system (like the hotel management office) can see all processes across all namespaces with their "real" PIDs, but processes inside a PID namespace only see the processes within their own namespace, numbered starting from PID 1. When a container process thinks it's PID 1, the host might see it as PID 15432 - just like how the guest in room 101 of Building A doesn't know that the hotel management system tracks their room as "A-101".

The critical insight is that PID 1 in a namespace has special responsibilities, just like how the manager of Building A is responsible for that building's operations. PID 1 must handle "orphaned" processes (zombie reaping) within its namespace, similar to how a building manager must handle abandoned rooms and maintenance issues within their building.

> **Key Design Insight**: The PID namespace component must handle the dual nature of process identity - maintaining the illusion of isolation within the namespace while coordinating with the host system for actual process management. This duality drives many of the implementation complexities around process creation, signal handling, and cleanup.

PID Namespace Interface

The PID namespace interface provides functions for creating isolated process environments and managing the complex lifecycle of processes within those environments. The design centers around the `safe_clone` function and supporting infrastructure for handling the unique requirements of becoming PID 1 within a namespace.

Function Name	Parameters	Returns	Description
`create_pid_namespace`	`container_config_t* config, container_instance_t* container`	`int`	Creates new PID namespace using clone() with CLONE_NEWPID flag
`safe_clone`	`int (*fn)(void*), void* stack, int flags, void* arg`	`pid_t`	Wrapper for clone() syscall with comprehensive error checking and stack validation
`container_init_process`	`void* arg`	`int`	Entry point function for container process, handles PID 1 initialization and exec
`setup_init_signals`	`void`	`int`	Configures signal handlers for zombie reaping and graceful shutdown
`reap_zombies`	`void`	`void`	Signal handler that reaps zombie child processes using waitpid()
`exec_container_command`	`char** argv, char** envp`	`int`	Replaces init process with user-specified command via execve()

The PID namespace creation process involves careful coordination between the parent (host) process and the child (container) process. The parent process calls `safe_clone` with the `CLONE_NEWPID` flag, which causes the child process to be created in a new PID namespace where it becomes PID 1. This transition is fundamental to container isolation but introduces significant complexity around process initialization and signal handling.

The `safe_clone` function serves as a critical abstraction over the raw `clone()` system call, handling the numerous error conditions and edge cases that arise from namespace creation. It validates the stack pointer direction (growing up vs. down on different architectures), checks for sufficient privileges to create namespaces,

and provides detailed error reporting for common failure modes like ENOSPC (no space in PID namespace) or EPERM (insufficient privileges).

> **Critical Design Principle**: The container init process must be prepared to handle all the responsibilities of PID 1, including zombie reaping and signal forwarding, before executing the user's intended command. Failing to properly initialize can lead to zombie process accumulation and signal delivery problems within the container.

The `container_init_process` function represents the entry point for the container process after namespace creation. This function runs as PID 1 within the new namespace and must establish the proper runtime environment before transitioning to the user's intended command. It configures signal handlers, sets up minimal process state, and then uses `execve()` to replace itself with the actual container workload.

Process State	PID (Host View)	PID (Container View)	Responsibilities
Host Process	Variable	N/A	Creates namespaces, monitors container
Container Init	Variable (e.g. 15432)	1	Zombie reaping, signal handling, exec user command
Container Children	Variable	2, 3, 4...	User workloads, inherit namespace

The signal handling configuration is particularly critical because PID 1 in a namespace has special signal semantics. Unlike regular processes, PID 1 can only be killed by signals it has explicitly configured handlers for. This means the container init process must explicitly handle signals like SIGTERM and SIGINT to enable graceful shutdown, and must handle SIGCHLD to perform zombie reaping.

Init Process Responsibilities

The init process within a PID namespace carries the fundamental responsibility of being the root of the process tree and the ultimate parent for all orphaned processes. When any process within the namespace exits, if its parent has already terminated, that process becomes a child of PID 1 (the init process). This creates an obligation for the init process to continuously reap zombie processes to prevent resource exhaustion.

The zombie reaping mechanism operates through the SIGCHLD signal handler, which is delivered whenever a child process changes state (typically when it exits). The signal handler must call `waitpid()` in a loop with the WNOHANG flag to reap all available zombie processes without blocking. This is critical because multiple children might exit simultaneously, but only one SIGCHLD signal might be delivered due to signal coalescing.

Signal	Purpose	Handler Action	Criticality
SIGCHLD	Child process exit	Call waitpid() in loop to reap zombies	Critical - prevents resource leaks
SIGTERM	Graceful shutdown request	Forward to child processes, then exit	High - enables clean container stop
SIGINT	Interrupt (Ctrl+C)	Forward to child processes, then exit	High - enables interactive termination
SIGUSR1	Custom container signal	Forward to main container process	Medium - application-specific
SIGKILL	Force termination	Cannot be caught - handled by kernel	N/A - immediate termination

The signal forwarding mechanism ensures that signals sent to the container (via docker kill or similar commands) reach the actual application process rather than being handled solely by the init wrapper. This requires maintaining a reference to the main container process PID and forwarding appropriate signals while still handling init-specific signals locally.

Here's the step-by-step process for handling init responsibilities:

1. **Signal Handler Installation**: The init process configures signal handlers for SIGCHLD, SIGTERM, and SIGINT before doing anything else. This ensures zombie reaping capability is established immediately.
2. **Initial Process Setup**: The init process may spawn the main container application as a child process, or it may exec directly into the application after setting up signal forwarding.
3. **Zombie Reaping Loop**: When SIGCHLD is received, the handler enters a loop calling `waitpid(-1, &status, WNOHANG)` until no more zombie children are available. The WNOHANG flag prevents blocking if no zombies are present.
4. **Signal Forwarding**: When SIGTERM or SIGINT is received, the init process forwards these signals to its child processes (if any) and then waits a reasonable timeout before exiting itself.
5. **Graceful Shutdown**: The init process coordinates an orderly shutdown by signaling children, waiting for them to exit, reaping their zombie processes, and finally exiting itself.
6. **Emergency Cleanup**: If children don't respond to graceful signals within the timeout period, the init process may send SIGKILL to force termination before exiting.

> **Critical Implementation Detail**: The zombie reaping must handle the case where the main application process exits but has spawned daemon processes that become orphaned. These daemons become children of the container init process and must be properly reaped when they eventually exit.

The exec transition from init process to application process is an important optimization for containers that run a single primary application. Instead of maintaining the init process as a separate parent, the init process can

configure signal handlers and then exec directly into the application. However, this approach sacrifices the ability to reap orphaned processes that might be created by the application.

Approach	Pros	Cons	Best For
Persistent Init	Handles zombies, signal forwarding	Extra memory overhead, complexity	Multi-process applications
Exec Replacement	Minimal overhead, direct execution	No zombie reaping, limited signal handling	Single-process applications
Hybrid (Setup then Exec)	Good signal setup, minimal runtime overhead	Limited zombie handling for orphans	Simple applications with known behavior

Architecture Decision Records

Decision: Clone() vs Unshare() for PID Namespace Creation

- **Context**: We need to create a new PID namespace for container process isolation. Linux provides two system calls: `clone()` which creates a new process in a new namespace, and `unshare()` which moves the current process into a new namespace.
- **Options Considered**:
  1. Use `clone()` with CLONE_NEWPID to create child in new namespace
  2. Use `unshare()` to move current process into new namespace
  3. Hybrid approach using both calls
- **Decision**: Use `clone()` with CLONE_NEWPID for PID namespace creation
- **Rationale**: PID namespaces have a special restriction - when a process calls `unshare(CLONE_NEWPID)`, the process itself remains in the original namespace, and only its children enter the new namespace. This creates confusion about which process is actually isolated. Using `clone()` ensures the child process is immediately in the new namespace as PID 1, providing clear semantics and avoiding the need for additional process spawning.
- **Consequences**: We must handle the complexity of inter-process communication between parent and child, manage stack allocation for clone(), and handle the timing of namespace setup. However, we get predictable PID namespace semantics and avoid the confusion of mixed namespace membership.

Option	Pros	Cons
clone() with CLONE_NEWPID	Child immediately becomes PID 1, clear namespace boundaries	Complex stack management, IPC needed
unshare() then fork	Simpler control flow, no stack allocation	Process calling unshare stays in old namespace
Hybrid approach	Flexibility in namespace transition	Increased complexity, harder to debug

Decision: Stack Direction Handling in safe_clone()

- **Context**: The `clone()` system call requires a stack pointer, but different CPU architectures have different stack growth directions (up vs down). ARM typically grows up, x86 grows down.
- **Options Considered**:
  1. Assume downward growing stack (x86-style) and use (stack + STACK_SIZE)
  2. Use architecture-specific preprocessor directives to handle both directions
  3. Probe stack direction at runtime and adjust accordingly
- **Decision**: Use architecture-specific preprocessor directives with fallback to downward growth
- **Rationale**: Stack direction is determined at compile time based on the target architecture. Runtime probing adds unnecessary complexity and performance overhead. Preprocessor directives provide compile-time optimization while supporting both common stack directions.
- **Consequences**: We need architecture-specific code paths, but this provides optimal performance and correctly handles the most common architectures. The fallback ensures compatibility with less common architectures.

Decision: Init Process Design Pattern

- **Context**: The container process becomes PID 1 in its namespace and must handle init responsibilities while also running the user's intended application.
- **Options Considered**:
  1. Minimal init that immediately exec()s into user application
  2. Persistent init that spawns user application as child and manages it
  3. Configurable init that can operate in either mode based on container config
- **Decision**: Implement persistent init that spawns user application as child process
- **Rationale**: While exec-style init has lower memory overhead, the persistent init approach provides essential container functionality including zombie reaping, signal forwarding, and graceful shutdown coordination. These capabilities are critical for production container usage and handling multi-process applications correctly.
- **Consequences**: We use additional memory for the init process and increase complexity in signal handling and process management. However, we gain robust process tree management and compatibility with applications that spawn child processes.

Common Pitfalls

⚠️ Pitfall: Incorrect Stack Pointer Calculation

Many developers incorrectly assume that all architectures use downward-growing stacks and pass the raw stack buffer pointer to `clone()`. On architectures with upward-growing stacks (like some ARM configurations), this causes the child process to start with an invalid stack pointer, leading to immediate segmentation faults.

The mistake typically looks like:

- Allocating a stack buffer: `char stack[STACK_SIZE];`
- Passing the base address directly: `clone(fn, stack, flags, arg)`
- The child process crashes immediately with SIGSEGV

The correct approach is to check the architecture and adjust the stack pointer accordingly. For downward-growing stacks, pass `stack + STACK_SIZE`. For upward-growing stacks, pass the base address `stack`. Use preprocessor directives like `#ifdef __hppa__` to detect architectures with upward-growing stacks.

⚠️ Pitfall: Ignoring Zombie Process Accumulation

A common mistake is implementing a container init process that doesn't properly handle SIGCHLD signals or doesn't loop when reaping zombies. This leads to zombie process accumulation, eventually exhausting the process table and preventing new process creation.

The mistake occurs when developers either:

1. Don't install a SIGCHLD handler at all
2. Install a handler but only call `waitpid()` once instead of looping

3. Use blocking `wait()` instead of non-blocking `waitpid()` with WNOHANG

The consequence is that zombie processes accumulate until the system reaches its process limit, causing new process creation to fail with "Resource temporarily unavailable" errors. This is particularly problematic in long-running containers or containers that spawn many short-lived processes.

The fix involves installing a proper SIGCHLD handler that loops until no more zombies are available:

```c
// Handler must loop because multiple children might exit simultaneously
while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
    // Process reaped successfully, continue looping
}
```

⚠️ Pitfall: Signal Handling Race Conditions

PID 1 has special signal semantics in Linux - it can only be terminated by signals for which it has explicitly installed handlers. Many developers forget this and wonder why their containers don't respond to SIGTERM or SIGINT signals. Without proper signal handlers, the container init process becomes unkillable except via SIGKILL.

The race condition occurs during the window between process creation and signal handler installation. If a signal arrives before handlers are installed, it may be ignored or cause unexpected behavior. This is particularly problematic during container shutdown when orchestration systems send SIGTERM expecting graceful termination.

The solution is to install signal handlers as early as possible in the init process, before any other initialization:

1. Install SIGCHLD handler for zombie reaping
2. Install SIGTERM handler for graceful shutdown
3. Install SIGINT handler for interactive termination
4. Block signals during critical sections to prevent races

⚠️ Pitfall: Parent Process Waiting Strategy

A subtle error occurs when the parent process (host side) doesn't properly wait for the container child process, or waits incorrectly. This can lead to the container process becoming a zombie itself, or the parent process hanging indefinitely.

Common mistakes include:

- Using `wait()` instead of `waitpid()` with specific PID
- Not handling EINTR when wait is interrupted by signals
- Forgetting to check the exit status to distinguish normal exit from signals

The parent process must use `waitpid()` with the specific child PID and handle interruption correctly:

```c
int status;

pid_t result;

do {

    result = waitpid(container->child_pid, &status, 0);

} while (result == -1 && errno == EINTR);
```

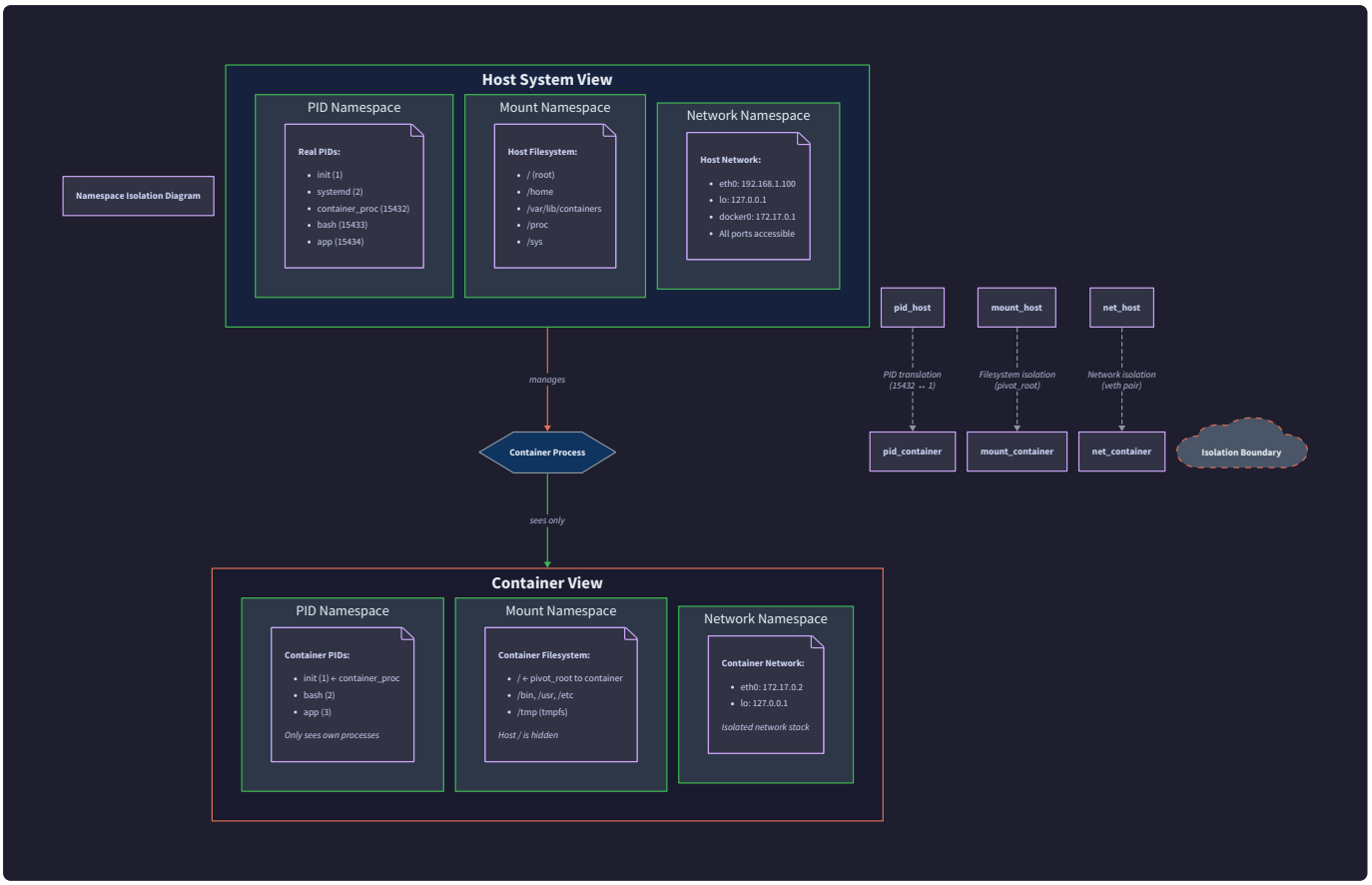
⚠️ Pitfall: Namespace Cleanup Timing

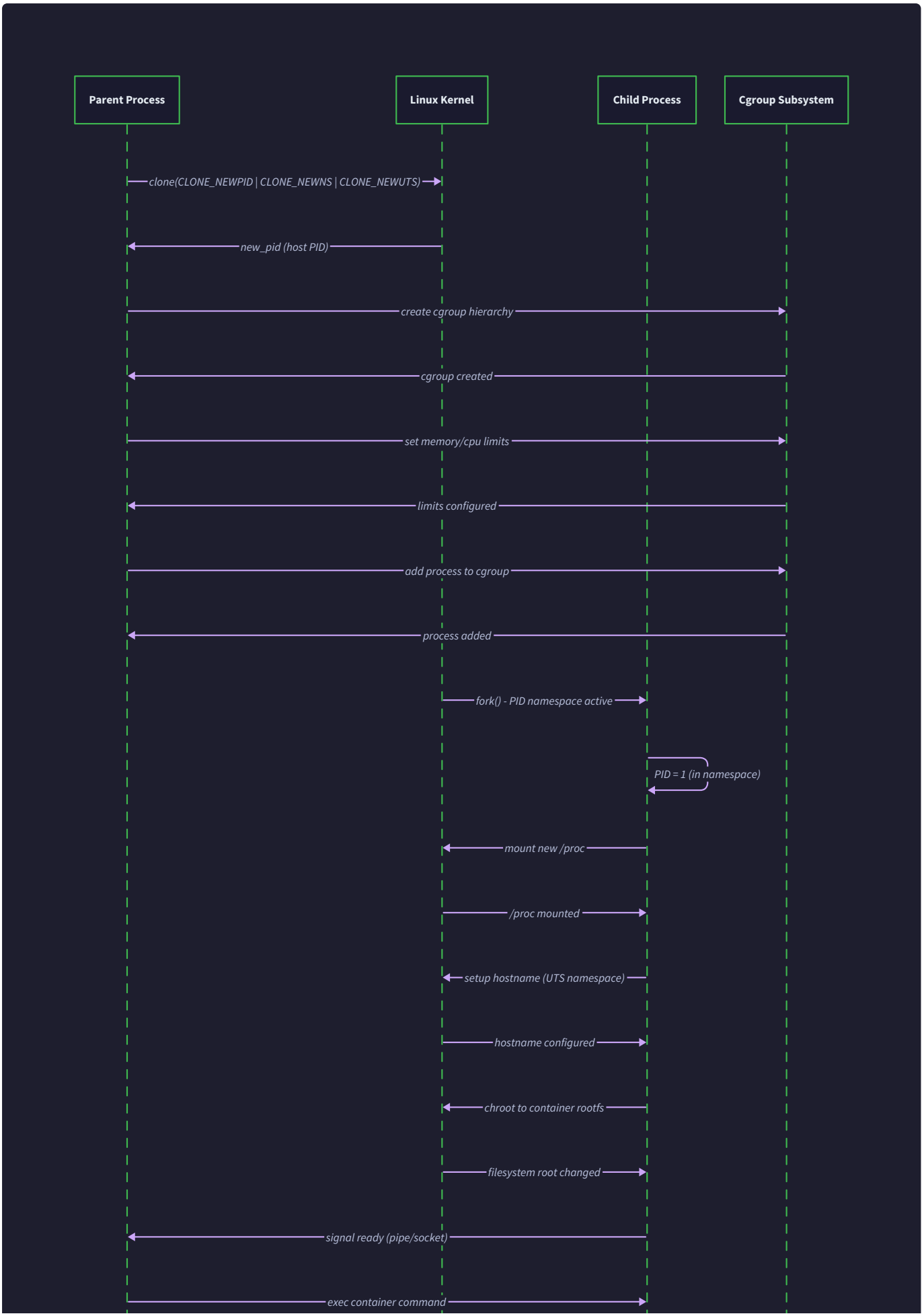
When a container exits, the PID namespace persists until all processes within it have exited and all file descriptors referring to the namespace are closed. Developers often forget that holding open namespace file descriptors (from /proc/PID/ns/) prevents namespace cleanup, leading to resource leaks.

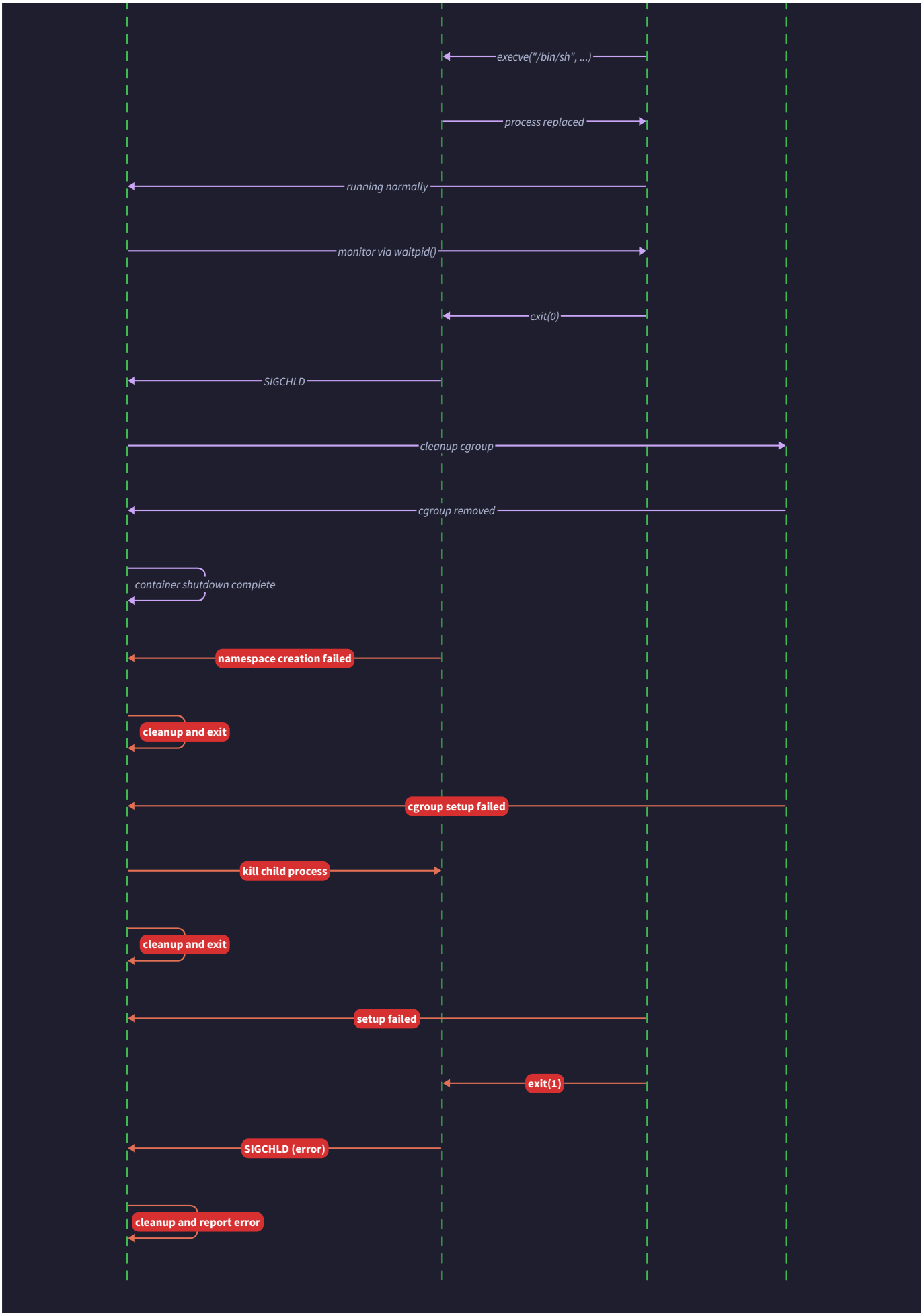
The mistake manifests as:

- Opening namespace file descriptors for monitoring or debugging
- Forgetting to close them when the container exits
- Namespace directories remaining in /proc after container termination
- Eventually exhausting namespace resources

The fix requires careful tracking of all namespace file descriptors and ensuring they're closed during container cleanup. The cleanup process should close namespace descriptors before waiting for the container process to exit.




Parent Process
Linux Kernel
Child Process
Cgroup Subsystem
clone(CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWUTS)
new_pid (host PID)
create cgroup hierarchy
cgroup created
set memory/cpu limits
limits configured
add process to cgroup
process added
fork() - PID namespace active
PID = 1 (in namespace)
mount new /proc
/proc mounted
setup hostname (UTS namespace)
hostname configured
chroot to container rootfs
filesystem root changed
signal ready (pipe/socket)
exec container command


execve("/bin/sh", ...)
process replaced
running normally
monitor via waitpid()
exit(0)
SIGCHLD
cleanup cgroup
cgroup removed
container shutdown complete
namespace creation failed
cleanup and exit
cgroup setup failed
kill child process
cleanup and exit
setup failed
exit(1)
SIGCHLD (error)
cleanup and report error

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Process Creation	`fork()` + `execve()` + manual namespace setup	`clone()` with namespace flags for atomic creation
Signal Handling	Basic `signal()` calls with simple handlers	`sigaction()` with advanced signal mask control
Error Reporting	`perror()` + exit codes	Structured error codes with detailed context
Stack Management	Fixed-size stack arrays	Dynamic stack allocation with guard pages
Namespace Monitoring	Manual /proc filesystem inspection	File descriptor-based namespace tracking

Recommended File Structure

```
container-basic/
  src/
    pid_namespace.c          ← PID namespace creation and management
    pid_namespace.h          ← Function declarations and structures
    container_init.c         ← Container init process implementation
    signal_handling.c        ← Signal handler implementations
    safe_clone.c            ← Clone wrapper with error handling
    common.h                ← Shared constants and structures
  tests/
    test_pid_namespace.c    ← Unit tests for PID namespace functionality
    test_container_init.c   ← Tests for init process behavior
    test_zombie_reaping.c   ← Tests for zombie process handling
  examples/
    simple_container.c      ← Basic container demonstration
```

Infrastructure Starter Code

safe_clone.c - Complete clone() wrapper with error handling:

```c
#include <sys/wait.h>
#include <sys/syscall.h>
#include <sched.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include "common.h"

// Architecture-specific stack pointer calculation
static void* calculate_stack_pointer(void* stack_base) {
#ifdef __hppa__
    // HPPA has upward-growing stack
    return stack_base;
#else
    // Most architectures have downward-growing stack
    return (char*)stack_base + STACK_SIZE;
#endif
}

// Validate stack alignment and bounds
static int validate_stack(void* stack_base) {
    if (!stack_base) {
        return -1;
    }

    // Check for reasonable alignment (at least 8-byte aligned)
    if ((uintptr_t)stack_base & 7) {
        return -1;
```

```
    }

    return 0;
}

pid_t safe_clone(int (*fn)(void*), void* stack, int flags, void* arg) {
    if (!fn || validate_stack(stack) != 0) {
        errno = EINVAL;
        return -1;
    }

    void* stack_ptr = calculate_stack_pointer(stack);
    pid_t pid = clone(fn, stack_ptr, flags, arg);

    if (pid == -1) {
        // Provide detailed error information
        switch (errno) {
        case EPERM:
            fprintf(stderr, "clone failed: insufficient privileges for namespace creation\n");
            break;
        case ENOSPC:
            fprintf(stderr, "clone failed: no space left in PID namespace\n");
            break;
        case ENOMEM:
            fprintf(stderr, "clone failed: insufficient memory\n");
            break;
        default:
            fprintf(stderr, "clone failed: %s\n", strerror(errno));
```

```
            break;
        }
    }

    return pid;
}

// Stack allocation helper with proper alignment
void* allocate_container_stack(void) {
    void* stack = aligned_alloc(4096, STACK_SIZE);
    if (!stack) {
        perror("Failed to allocate container stack");
        return NULL;
    }
    return stack;
}

void free_container_stack(void* stack) {
    free(stack);
}
```

signal_handling.c - Complete signal handler infrastructure:

```c
#include <signal.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include "common.h"

// Global state for signal handling
static volatile sig_atomic_t shutdown_requested = 0;
static pid_t main_child_pid = 0;

// SIGCHLD handler - reaps zombie processes
static void sigchld_handler(int sig) {
    int saved_errno = errno;
    int status;
    pid_t pid;

    // Loop to reap all available zombie children
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        // Log child exit if needed (avoid non-async-safe functions in real handlers)
        // Child process reaped successfully
    }

    errno = saved_errno;
}

// SIGTERM/SIGINT handler - initiates graceful shutdown
static void sigterm_handler(int sig) {
    shutdown_requested = 1;
```

```c
    // Forward signal to main child process if it exists
    if (main_child_pid > 0) {
        kill(main_child_pid, sig);
    }
}

int setup_init_signals(void) {
    struct sigaction sa;

    // Configure SIGCHLD handler for zombie reaping
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART | SA_NOCLDSTOP;
    sa.sa_handler = sigchld_handler;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        perror("Failed to install SIGCHLD handler");
        return -1;
    }

    // Configure SIGTERM handler for graceful shutdown
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = sigterm_handler;
    if (sigaction(SIGTERM, &sa, NULL) == -1) {
        perror("Failed to install SIGTERM handler");
        return -1;
    }
```

```c
    // Configure SIGINT handler (same as SIGTERM)
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("Failed to install SIGINT handler");
        return -1;
    }

    return 0;
}

void set_main_child_pid(pid_t pid) {
    main_child_pid = pid;
}

int is_shutdown_requested(void) {
    return shutdown_requested;
}

// Wait for graceful shutdown with timeout
int wait_for_graceful_shutdown(int timeout_seconds) {
    int elapsed = 0;
    while (elapsed < timeout_seconds && main_child_pid > 0) {
        sleep(1);
        elapsed++;

        // Check if main child has exited
        if (waitpid(main_child_pid, NULL, WNOHANG) > 0) {
            main_child_pid = 0;
            return 0;  // Graceful exit
        }
```

```
    }


    return -1;  // Timeout
}
```

Core Logic Skeleton Code

pid_namespace.c - Core PID namespace functionality to implement:

```c
#include <sched.h>

#include <unistd.h>

#include <sys/wait.h>

#include <errno.h>

#include "common.h"


// Creates a new PID namespace and container process

int create_pid_namespace(container_config_t* config, container_instance_t* container) {

    // TODO 1: Allocate stack for container process using allocate_container_stack()



    // TODO 2: Register stack cleanup in case of failure

    // Hint: Use register_cleanup() to ensure stack is freed on failure



    // TODO 3: Call safe_clone() with CLONE_NEWPID flag and container_init_process as entry
point

    // Hint: Flags should include CLONE_NEWPID | SIGCHLD for basic PID namespace



    // TODO 4: Check clone() return value - negative means error, positive is child PID



    // TODO 5: Store child PID in container->child_pid for later management



    // TODO 6: Store namespace file descriptor in container->namespace_fds[0] if needed

    // Hint: Open /proc/<child_pid>/ns/pid for namespace monitoring



    // TODO 7: Return 0 on success, -1 on failure with errno set

    return -1;  // Replace with actual implementation

}


// Container init process entry point (runs as PID 1 in new namespace)
```

```c
int container_init_process(void* arg) {
    container_config_t* config = (container_config_t*)arg;

    // TODO 1: Verify we're actually PID 1 in the namespace
    // Hint: getpid() should return 1 if we're in the new PID namespace

    // TODO 2: Install signal handlers using setup_init_signals()
    // This must be done before any other operations to handle zombies

    // TODO 3: Set up minimal process environment (working directory, umask, etc.)
    // Hint: chdir() to appropriate directory, umask(0022) for reasonable permissions

    // TODO 4: If config specifies running as persistent init, fork() the main application
    // Store the child PID using set_main_child_pid() for signal forwarding

    // TODO 5: If persistent init mode, enter main loop watching for signals and reaping zombies
    // Use is_shutdown_requested() to check for graceful shutdown

    // TODO 6: If exec mode, directly exec into the user application
    // Use execve() with config->argv and config->envp

    // TODO 7: Handle cleanup and error cases - this should not return normally
    return 255;  // Should not reach here
}

// Wait for container process to exit and collect status
int container_wait(container_instance_t* container) {
    // TODO 1: Validate container parameter and child_pid
```

```
    // TODO 2: Use waitpid() to wait for specific child process
    // Handle EINTR interruption by retrying the wait

    // TODO 3: Store exit status and return code for caller
    // Distinguish between normal exit and signal termination

    // TODO 4: Mark container as exited (set child_pid to 0 or -1)

    return -1;  // Replace with actual implementation
}
```

Language-Specific Hints

- Use `clone()` instead of `fork()` for namespace creation - it's the only way to create PID namespaces atomically
- Always check `errno` after failed system calls and use `strerror()` or `perror()` for debugging
- Use `sigaction()` instead of `signal()` for reliable signal handling behavior across platforms
- The `CLONE_NEWPID` flag creates the new PID namespace; combine with `SIGCHLD` for proper child handling
- Use `waitpid()` with `WNOHANG` in signal handlers to avoid blocking
- File descriptor `/proc/PID/ns/pid` can be used to hold references to namespaces for monitoring
- Use `prctl(PR_SET_PDEATHSIG, SIGKILL)` to ensure container dies if parent dies unexpectedly

Milestone Checkpoint

After implementing the PID namespace component, verify the following behavior:

Test Command:

```bash
gcc -o test_pid_namespace test_pid_namespace.c pid_namespace.c safe_clone.c signal_handling.c

sudo ./test_pid_namespace
```

Expected Output:

```
Container PID from host perspective: 1234
Container PID from inside namespace: 1
Container successfully created and isolated
Container exited with status: 0
```

Manual Verification Steps:

1. **PID Isolation Check**: Run `ps aux` in another terminal while the container is running. You should see the container process with a normal PID (e.g., 1234), but from inside the container, `echo $$` should return 1.
2. **Zombie Reaping Test**: Create a test that spawns child processes inside the container and verifies they're properly reaped. No zombie processes should accumulate in `ps aux | grep defunct`.
3. **Signal Handling Test**: Send `SIGTERM` to the container and verify it shuts down gracefully within a reasonable timeout.

Signs Something is Wrong:

Symptom	Likely Cause	Fix
Container process crashes immediately	Stack pointer calculation error	Check architecture-specific stack direction
`clone()` fails with EPERM	Insufficient privileges	Run with sudo or check namespace availability
Zombie processes accumulate	Missing or broken SIGCHLD handler	Verify signal handler installation and waitpid loop
Container doesn't respond to signals	Missing signal handlers for PID 1	Install handlers for SIGTERM, SIGINT early in init
Container hangs on exit	Parent not properly waiting	Use waitpid() with correct PID, handle EINTR

Mount Namespace Component

> **Milestone(s):** This section corresponds to Milestone 2 (Mount Namespace), which implements filesystem isolation to create a contained filesystem view separate from the host system.

Mental Model: Stage Sets

Think of mount namespaces like **theater stage sets**. In a theater production, each act might require a completely different scene - a living room for Act 1, a garden for Act 2, and a castle for Act 3. The audience sees only one set at a time, and when the curtain falls, the stage crew can completely transform the visible environment without affecting the real world outside the theater.

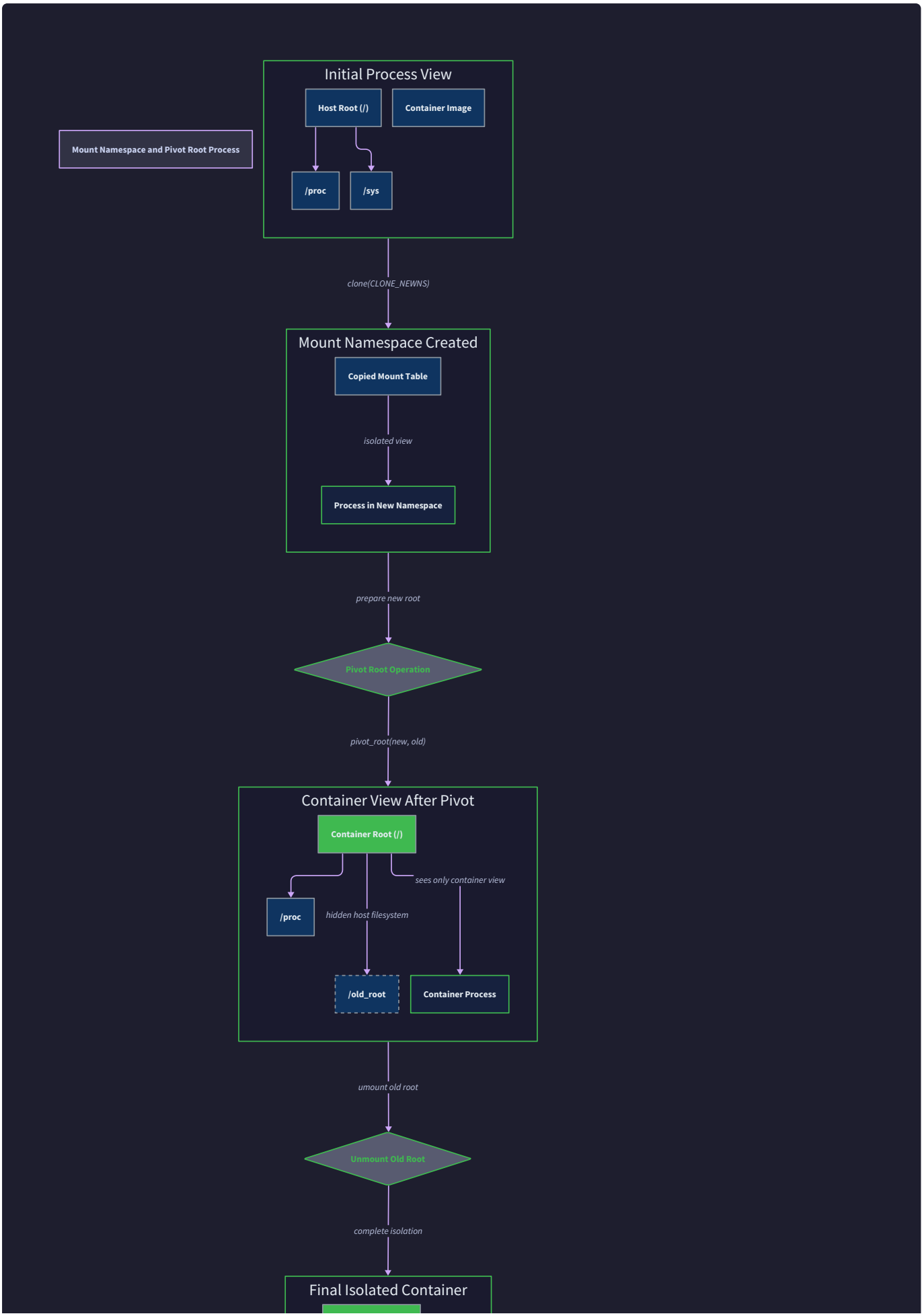
A mount namespace works similarly for processes. Just as the theater audience sees only the current stage set and remains unaware of the other sets stored backstage, a process in a mount namespace sees only its own filesystem view and cannot access or even detect the filesystem mounts that exist on the host or in other namespaces. When we create a new mount namespace, we're essentially giving the container process its own "stage" where it can mount and unmount filesystems without affecting the host's filesystem layout.

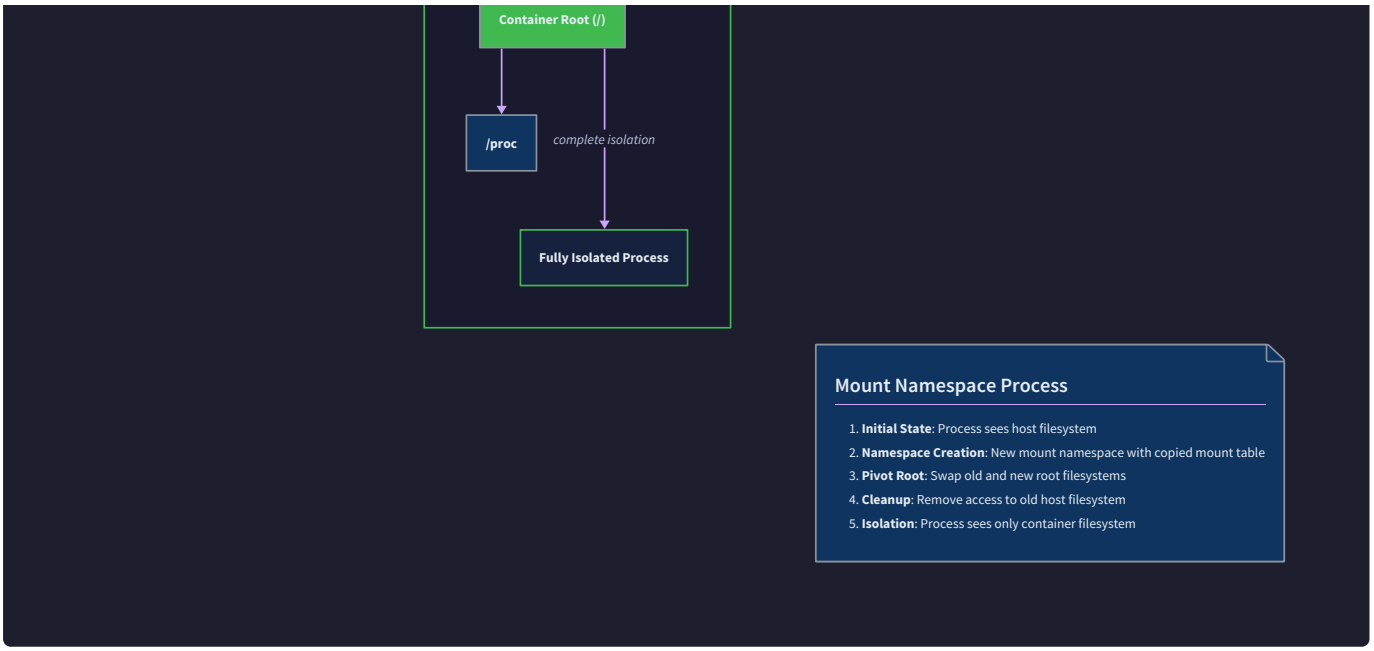
The **pivot_root operation** is like the moment when the stage crew wheels out the old set and wheels in the new one. The audience (our container process) suddenly sees a completely different environment as their reality. What was once the host's `/` root directory becomes hidden away (like storing the old stage set), and the container's new root directory takes center stage as the process's new `/`.

This analogy helps explain why mount namespace isolation is so powerful: just as theater sets allow multiple completely different scenes to coexist in the same building without interfering with each other, mount namespaces allow multiple completely different filesystem views to coexist on the same machine without processes being able to escape their designated view.

Mount Namespace Interface

The mount namespace component provides the core functions necessary to create filesystem isolation and establish a contained root filesystem for the container process. These functions handle the complex sequence of operations required to safely transition from the host filesystem view to an isolated container filesystem view.


Mount Namespace and Pivot Root Process
Initial Process View
Host Root (/)
Container Image
/proc
/sys
clone(CLONE_NEWNS)
Mount Namespace Created
Copied Mount Table
isolated view
Process in New Namespace
prepare new root
Pivot Root Operation
pivot_root(new, old)
Container View After Pivot
Container Root (/)
sees only container view
/proc
hidden host filesystem
/old_root
Container Process
umount old root
Unmount Old Root
complete isolation
Final Isolated Container



The primary interface functions coordinate multiple low-level mount operations to achieve filesystem isolation while maintaining the essential filesystems that the container process needs to function properly.

Method Name	Parameters	Returns	Description
`create_mount_namespace`	`config container_config_t*`, `container container_instance_t*`	`int`	Creates new mount namespace and sets up isolated filesystem view
`setup_container_rootfs`	`rootfs_path char*`, `old_root char*`	`int`	Prepares container root filesystem directory structure
`pivot_to_container_root`	`new_root char*`, `old_root char*`	`int`	Switches filesystem root using pivot_root syscall
`mount_essential_filesystems`	`void`	`int`	Mounts /proc, /sys, and /dev inside container namespace
`setup_mount_propagation`	`void`	`int`	Configures mount propagation to prevent host leakage
`bind_mount_host_directories`	`config container_config_t*`	`int`	Creates bind mounts for shared host directories

The `create_mount_namespace` function serves as the main orchestrator for filesystem isolation. It begins by calling the `unshare` system call with the `CLONE_NEWNS` flag to create a new mount namespace, separating the

container's filesystem view from the host. This operation creates a copy of the host's mount table that the container can then modify without affecting the host system.

After creating the namespace, the function calls `setup_mount_propagation` to configure how mount events propagate between the new namespace and the host. This step is critical because the default mount propagation can cause container mounts to leak back to the host filesystem, breaking isolation. The function sets the propagation to `MS_PRIVATE` for the entire mount tree, ensuring that subsequent mount operations remain contained within the namespace.

The `setup_container_rootfs` function prepares the directory structure needed for the pivot_root operation. This involves creating the new root directory if it doesn't exist and setting up a temporary directory that will hold the old root filesystem after the pivot. The function also ensures that the new root directory is on a different filesystem from the current root, as required by the pivot_root syscall.

> **Critical Design Insight**: The mount namespace isolation must occur before any filesystem modifications because once we start changing mounts, we need to ensure those changes remain contained within the container's namespace and don't affect the host system.

The `pivot_to_container_root` function performs the actual filesystem root transition using the pivot_root syscall. This operation atomically swaps the current root filesystem with the new container root, moving the old root to a subdirectory where it can be safely unmounted. This is more secure than using chroot because it completely removes access to the old root filesystem rather than simply changing the apparent root directory.

Pivot Root Process

The pivot root process represents one of the most complex aspects of container filesystem isolation, requiring careful coordination of multiple mount operations to safely transition the container process from the host filesystem view to its own isolated root filesystem.

The process begins with **namespace preparation**, where we ensure that the container has its own mount namespace and that mount propagation is configured to prevent operations from affecting the host. This preparation phase is critical because any mistakes here can cause container filesystem changes to leak back to the host system, breaking the isolation boundary.

Step-by-step pivot root algorithm:

1. **Validate filesystem requirements**: Check that the new root directory exists and is accessible, and verify that it's mounted on a different filesystem from the current root (pivot_root requirement).
2. **Create old root directory**: Inside the new root filesystem, create a temporary directory (typically `/old-root`) that will hold the current root filesystem after the pivot operation.
3. **Bind mount new root**: Perform a bind mount of the new root directory onto itself to ensure it appears as a proper mount point, as required by pivot_root semantics.
4. **Execute pivot_root syscall**: Call `pivot_root(new_root, old_root_path)` to atomically swap the filesystem roots, making the new root become `/` and moving the old root to `/old-root`.

5. **Change working directory**: Update the current working directory to the new root filesystem to ensure subsequent operations occur in the container context.
6. **Mount essential filesystems**: Mount `/proc`, `/sys`, and `/dev` filesystems inside the new root to provide essential kernel interfaces that the container process needs.
7. **Setup bind mounts**: Create any required bind mounts to share specific host directories with the container (like shared volumes or socket directories).
8. **Unmount old root**: Safely unmount the old root filesystem from `/old-root` and remove the temporary directory, completing the isolation.

The **filesystem validation step** requires checking several conditions that are often overlooked. The new root directory must exist and be accessible, but it also must be a mount point or be bind-mounted to itself. This requirement stems from the way pivot_root verifies that it's operating on actual filesystems rather than simple directories.

Mount point verification involves checking `/proc/mounts` to confirm that the new root appears as a separate mount entry. If the new root is just a directory on the existing filesystem, we must perform a bind mount operation: `mount --bind /path/to/new/root /path/to/new/root` to make it appear as a mount point.

> **Essential Insight**: The pivot_root operation is atomic from the kernel's perspective, but the surrounding setup operations are not. This means we must carefully handle partial failures during the setup phase to avoid leaving the container in an inconsistent filesystem state.

The **old root management** phase requires creating the old root directory inside the new root filesystem before calling pivot_root. This directory serves as the mount point where the current root filesystem will be relocated. The directory name should be unique and temporary since it will be unmounted and removed after the pivot completes.

Essential filesystem mounting happens after the pivot_root completes because these virtual filesystems need to appear in the container's filesystem namespace. The `/proc` filesystem provides process information and kernel interfaces, `/sys` exposes kernel and device information, and `/dev` provides device nodes. Each of these requires specific mount options and must be mounted in the correct order.

Architecture Decision Records

The mount namespace implementation requires several critical design decisions that affect both security and functionality. These decisions represent trade-offs between isolation strength, implementation complexity, and compatibility with existing systems.

Decision: pivot_root vs chroot for Root Filesystem Isolation

- **Context**: Container processes need to see a different root filesystem than the host, and we must choose between pivot_root and chroot for implementing this isolation
- **Options Considered**: pivot_root syscall, chroot syscall, bind mounts with chroot
- **Decision**: Use pivot_root for root filesystem switching
- **Rationale**: pivot_root provides true filesystem isolation by completely removing access to the old root, while chroot only changes the apparent root and allows escape via relative paths like `../../../` or file descriptors opened before the chroot
- **Consequences**: Requires more complex setup with mount namespace creation and old root management, but provides stronger security isolation

Option	Pros	Cons	Security Level
pivot_root	Complete isolation, no escape paths, proper filesystem semantics	Complex setup, requires mount namespace, needs old root handling	High
chroot	Simple implementation, widely understood, minimal setup	Escape vulnerabilities, relative path issues, file descriptor leaks	Low
bind + chroot	Moderate complexity, some isolation improvements	Still vulnerable to chroot escapes, inconsistent behavior	Medium

Decision: Private Mount Propagation

- **Context**: Mount operations inside the container namespace can propagate back to the host filesystem depending on mount propagation settings
- **Options Considered**: shared propagation, slave propagation, private propagation
- **Decision**: Use private mount propagation (MS_PRIVATE) for container mount namespace
- **Rationale**: Private propagation ensures that mount and unmount operations inside the container remain completely isolated and cannot affect the host filesystem, preventing accidental host filesystem modification
- **Consequences**: Container mounts are fully isolated but cannot share filesystems mounted on the host after container creation

The **mount propagation decision** significantly impacts how container filesystems interact with host filesystem changes. Private propagation creates the strongest isolation boundary but requires careful planning for any shared filesystem access needs.

Propagation Type	Host→Container	Container→Host	Use Case
Private (chosen)	No	No	Full isolation
Shared	Yes	Yes	Shared filesystems
Slave	Yes	No	Read-only sharing

Decision: Essential Filesystem Mount Strategy

- **Context**: Container processes need access to kernel interfaces like /proc and /sys, but mounting them incorrectly can create security vulnerabilities
- **Options Considered**: bind mount from host, fresh mount in namespace, selective bind mounting
- **Decision**: Fresh mount of essential filesystems in container namespace
- **Rationale**: Fresh mounting provides proper isolation and ensures container sees only its own processes and kernel state, while bind mounting would expose host process information
- **Consequences**: Requires knowledge of correct mount options for each filesystem type, but provides better security isolation

Essential filesystem mounting requires understanding the specific purpose and security implications of each virtual filesystem. The `/proc` filesystem must be mounted fresh to show only container processes, while bind mounting would expose all host processes to the container.

Filesystem	Mount Type	Mount Options	Purpose
/proc	Fresh	`proc /proc proc defaults`	Container process view
/sys	Fresh	`sysfs /sys sysfs defaults`	Kernel interface access
/dev	Bind subset	Selected devices only	Essential device nodes
/dev/pts	Fresh	`devpts /dev/pts devpts defaults`	Pseudo-terminal support

Common Pitfalls

Mount namespace implementation contains several subtle pitfalls that can compromise container isolation or cause runtime failures. These issues often manifest as seemingly unrelated problems, making them particularly challenging for developers new to container implementation.

⚠️ Pitfall: Mount Propagation Leakage

Many developers forget to configure mount propagation before performing container mount operations, causing container mounts to appear on the host filesystem. This happens because the default mount propagation setting is often `shared`, meaning mount operations propagate between namespaces.

Symptom: Container directories appear mounted on the host system after container startup, and host system shows container-specific mount points in `/proc/mounts`.

Why it's wrong: This breaks the fundamental isolation promise of containers. Host administrators can accidentally access container filesystems, and container mount operations can interfere with host system management.

Fix: Call `mount(NULL, "/", NULL, MS_PRIVATE | MS_REC, NULL)` immediately after creating the mount namespace to set private propagation for the entire mount tree. The `MS_REC` flag applies the setting recursively to all existing mount points.

⚠️ Pitfall: pivot_root Directory Requirements

The pivot_root syscall has strict requirements about the relationship between the old and new root directories that are not well documented. Failing to meet these requirements causes pivot_root to fail with unclear error messages.

Symptom: pivot_root syscall fails with `EINVAL` error, even though both directories exist and are accessible.

Why it's wrong: pivot_root requires that the new root directory be a mount point (not just a directory) and that the old root directory be located within the new root filesystem hierarchy.

Fix: Before calling pivot_root, ensure the new root is a mount point by bind mounting it to itself: `mount(new_root, new_root, NULL, MS_BIND, NULL)`. Create the old root directory inside the new root filesystem, not alongside it.

⚠️ Pitfall: Device Node Accessibility

Containers often fail to access essential device nodes like `/dev/null`, `/dev/zero`, or `/dev/random` because these devices are not properly created or have incorrect permissions in the container namespace.

Symptom: Container processes fail with "Permission denied" or "No such file or directory" errors when trying to open standard device files, causing applications to crash unexpectedly.

Why it's wrong: Many applications and system libraries expect these device nodes to exist and be accessible. Without them, basic operations like discarding output (`> /dev/null`) or generating random numbers fail.

Fix: Create essential device nodes in the container's `/dev` directory using `mknod` or bind mount selected devices from the host. Ensure proper permissions are set: `chmod 666 /dev/null /dev/zero /dev/random`.

⚠️ Pitfall: Filesystem Type Detection

Different filesystem types require different mount options and may behave unexpectedly when used as container root filesystems. Developers often assume all filesystems behave identically during pivot_root operations.

Symptom: Container startup fails with mount errors or pivot_root failures that only occur on certain storage configurations or filesystem types.

Why it's wrong: Some filesystems (like NFS or FUSE) have restrictions on operations like pivot_root, and some require specific mount options to work correctly in container scenarios.

Fix: Detect the filesystem type using `statfs` or by parsing `/proc/mounts` before attempting pivot_root. Implement filesystem-specific handling for known problematic types, and provide clear error messages when unsupported filesystems are detected.

⚠️ Pitfall: Cleanup Order Dependencies

Mount namespace cleanup must occur in the correct order to avoid leaving orphaned mounts or causing unmount failures. Incorrect cleanup order can leave host system resources tied up or cause subsequent container operations to fail.

Symptom: Container cleanup fails with "Device or resource busy" errors, and `lsof` or `fuser` shows processes holding references to container mount points.

Why it's wrong: Mount points have dependency relationships - unmounting a parent before its children fails, and some mount points cannot be unmounted while processes have open files on them.

Fix: Implement cleanup in reverse order of creation: unmount children before parents, terminate all container processes before unmounting filesystems, and use lazy unmounting (`umount2` with `MNT_DETACH`) as a last resort for stuck mounts.

Implementation Guidance

The mount namespace component requires careful coordination of multiple system calls and precise error handling to achieve reliable filesystem isolation. The implementation must handle the complex interactions between mount namespaces, pivot_root requirements, and essential filesystem mounting.

Technology Recommendations

Component	Simple Option	Advanced Option
Filesystem Detection	Parse `/proc/mounts` manually	Use `libmount` library for robust parsing
Device Node Creation	Manual `mknod` calls with fixed device numbers	Dynamic device discovery from `/proc/devices`
Mount Option Handling	Hard-coded options for common filesystems	Flexible configuration with validation
Error Reporting	Simple errno checking with `perror`	Structured error context with operation details

Recommended File Structure

```
container-basic/
├── src/
│   ├── container.c              ← main container orchestration
│   ├── mount_namespace.c        ← mount namespace implementation
│   ├── mount_namespace.h        ← mount namespace interface
│   ├── filesystem_utils.c       ← helper functions for mount operations
│   └── filesystem_utils.h       ← filesystem utility interface
├── tests/
│   ├── test_mount_namespace.c   ← unit tests for mount isolation
│   └── test_filesystem_setup.c  ← integration tests for root setup
└── examples/
    └── minimal_container.c      ← demonstration of mount namespace usage
```

Infrastructure Starter Code

filesystem_utils.h:

```c
#ifndef FILESYSTEM_UTILS_H
#define FILESYSTEM_UTILS_H

#include <sys/types.h>
#include <sys/stat.h>

// Essential device nodes that containers typically need
struct device_node {
    const char* path;
    mode_t mode;
    dev_t device;
};

// Common filesystem types and their characteristics
struct filesystem_info {
    const char* name;
    int supports_pivot_root;
    const char* default_options;
};

// Utility functions for mount operations
int safe_mount(const char* source, const char* target, const char* type,
               unsigned long flags, const void* data);
int ensure_directory_exists(const char* path, mode_t mode);
int create_device_node(const char* path, mode_t mode, dev_t device);
int detect_filesystem_type(const char* path, char* type_buf, size_t buf_size);
int is_mount_point(const char* path);
int recursive_umount(const char* path);

// Mount propagation utilities
```

```
int set_mount_propagation(const char* path, int propagation_type);

int make_private_recursive(const char* path);


#endif // FILESYSTEM_UTILS_H
```

filesystem_utils.c:

```c
#include "filesystem_utils.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <sys/mount.h>
#include <sys/stat.h>
#include <sys/statfs.h>

// Common device nodes needed in containers
static const struct device_node essential_devices[] = {
    {"/dev/null", S_IFCHR | 0666, makedev(1, 3)},
    {"/dev/zero", S_IFCHR | 0666, makedev(1, 5)},
    {"/dev/random", S_IFCHR | 0666, makedev(1, 8)},
    {"/dev/urandom", S_IFCHR | 0666, makedev(1, 9)},
    {NULL, 0, 0} // sentinel
};

int safe_mount(const char* source, const char* target, const char* type,
               unsigned long flags, const void* data) {
    if (mount(source, target, type, flags, data) != 0) {
        fprintf(stderr, "Mount failed: source=%s target=%s type=%s: %s\n",
                source ? source : "none",
                target ? target : "none",
                type ? type : "none",
                strerror(errno));
        return -1;
    }
```

```c
    return 0;
}

int ensure_directory_exists(const char* path, mode_t mode) {
    struct stat st;
    if (stat(path, &st) == 0) {
        if (S_ISDIR(st.st_mode)) {
            return 0; // Directory already exists
        } else {
            errno = ENOTDIR;
            return -1; // Path exists but is not a directory
        }
    }

    // Create directory and all parent directories
    char* path_copy = strdup(path);
    if (!path_copy) return -1;

    char* p = path_copy;
    while ((p = strchr(p + 1, '/')) != NULL) {
        *p = '\0';
        if (mkdir(path_copy, mode) != 0 && errno != EEXIST) {
            free(path_copy);
            return -1;
        }
        *p = '/';
    }
```

```
    int result = mkdir(path_copy, mode);
    free(path_copy);

    return (result == 0 || errno == EEXIST) ? 0 : -1;
}

int create_device_node(const char* path, mode_t mode, dev_t device) {
    // Remove existing file if it exists
    unlink(path);

    if (mknod(path, mode, device) != 0) {
        fprintf(stderr, "Failed to create device node %s: %s\n",
                path, strerror(errno));
        return -1;
    }

    if (chmod(path, mode & 0777) != 0) {
        fprintf(stderr, "Failed to set permissions on %s: %s\n",
                path, strerror(errno));
        return -1;
    }

    return 0;
}

int is_mount_point(const char* path) {
    FILE* mounts = fopen("/proc/mounts", "r");
    if (!mounts) return -1;
```

```
    char line[1024];
    char mount_point[512];
    int found = 0;

    while (fgets(line, sizeof(line), mounts)) {
        if (sscanf(line, "%*s %511s %*s", mount_point) == 1) {
            if (strcmp(mount_point, path) == 0) {
                found = 1;
                break;
            }
        }
    }

    fclose(mounts);
    return found;
}

int set_mount_propagation(const char* path, int propagation_type) {
    return mount(NULL, path, NULL, propagation_type | MS_REC, NULL);
}

int make_private_recursive(const char* path) {
    return set_mount_propagation(path, MS_PRIVATE);
}
```

Core Logic Skeleton Code

mount_namespace.h:

```c
#ifndef MOUNT_NAMESPACE_H

#define MOUNT_NAMESPACE_H


#include "container.h"


// Mount namespace creation and management

int create_mount_namespace(container_config_t* config, container_instance_t* container);

int setup_container_rootfs(const char* rootfs_path, const char* old_root);

int pivot_to_container_root(const char* new_root, const char* old_root);

int mount_essential_filesystems(void);

int setup_mount_propagation(void);

int bind_mount_host_directories(container_config_t* config);


// Cleanup functions

int cleanup_mount_namespace(container_instance_t* container);


#endif // MOUNT_NAMESPACE_H
```

mount_namespace.c skeleton:

C

```c
#include "mount_namespace.h"
#include "filesystem_utils.h"
#include <sched.h>
#include <sys/mount.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int create_mount_namespace(container_config_t* config, container_instance_t* container) {
    // TODO 1: Call unshare(CLONE_NEWNS) to create new mount namespace
    // TODO 2: Set up private mount propagation to prevent leakage to host
    // TODO 3: Register cleanup function for mount namespace teardown
    // TODO 4: Store namespace file descriptor in container->namespace_fds[1]
    // TODO 5: Return 0 on success, -1 on failure with errno set
}

int setup_container_rootfs(const char* rootfs_path, const char* old_root) {
    // TODO 1: Check if rootfs_path exists and is accessible
    // TODO 2: Ensure rootfs_path is a mount point (bind mount if necessary)
    // TODO 3: Create old_root directory inside the new root filesystem
    // TODO 4: Verify that new root and old root are on different filesystems
    // TODO 5: Set appropriate permissions on directories
    // Hint: Use is_mount_point() and ensure_directory_exists() helper functions
}

int pivot_to_container_root(const char* new_root, const char* old_root) {
    // TODO 1: Change to the new root directory
```

```
    // TODO 2: Call pivot_root(new_root, old_root) to swap filesystems
    // TODO 3: Change working directory to new root ("/")
    // TODO 4: Verify that pivot operation completed successfully
    // TODO 5: Handle cleanup if pivot_root fails partway through
    // Hint: Check that old_root path exists inside new_root before pivot
}

int mount_essential_filesystems(void) {
    // TODO 1: Mount /proc filesystem: mount("proc", "/proc", "proc", 0, NULL)
    // TODO 2: Mount /sys filesystem: mount("sysfs", "/sys", "sysfs", 0, NULL)
    // TODO 3: Create /dev directory and mount tmpfs on it
    // TODO 4: Create essential device nodes in /dev using create_device_node()
    // TODO 5: Mount /dev/pts for pseudo-terminal support
    // Hint: Use safe_mount() wrapper for better error reporting
}

int setup_mount_propagation(void) {
    // TODO 1: Make entire mount tree private: mount(NULL, "/", NULL, MS_PRIVATE | MS_REC,
NULL)
    // TODO 2: Verify that propagation was set correctly
    // TODO 3: Handle case where MS_PRIVATE is not supported on this kernel
    // Hint: This must be done immediately after unshare(CLONE_NEWNS)
}

int bind_mount_host_directories(container_config_t* config) {
    // TODO 1: Parse any host directory bind mount specifications from config
    // TODO 2: For each bind mount, create target directory in container
    // TODO 3: Perform bind mount: mount(host_path, container_path, NULL, MS_BIND, NULL)
    // TODO 4: Set appropriate permissions on mounted directories
    // TODO 5: Register cleanup functions for each bind mount
```

```
    // Hint: Validate that host paths exist before attempting bind mount
}

int cleanup_mount_namespace(container_instance_t* container) {
    // TODO 1: Unmount all container filesystems in reverse order
    // TODO 2: Remove temporary directories created during setup
    // TODO 3: Close namespace file descriptor stored in container->namespace_fds[1]
    // TODO 4: Use lazy unmounting (MNT_DETACH) for stuck mounts
    // TODO 5: Return 0 if all cleanup succeeded, -1 if any failures occurred
    // Hint: Use recursive_umount() for complex mount hierarchies
}
```

Language-Specific Hints

System Call Error Handling:

- Always check return values from `mount()`, `umount()`, and `pivot_root()` syscalls
- Use `strerror(errno)` to get human-readable error descriptions
- Log both the operation being attempted and the errno value for debugging

Mount Flags and Options:

- Use `MS_BIND` for bind mounts, `MS_PRIVATE` for mount propagation
- Combine flags with bitwise OR: `MS_PRIVATE | MS_REC`
- Pass `NULL` for unused parameters rather than empty strings

Directory Path Handling:

- Use `realpath()` to resolve symbolic links in filesystem paths
- Check for trailing slashes in directory paths and normalize them
- Validate that paths don't contain `..` components to prevent directory traversal

File Descriptor Management:

- Open namespace file descriptors from `/proc/self/ns/mnt` for cleanup reference
- Use `O_CLOEXEC` flag when opening namespace descriptors
- Close all file descriptors before calling `pivot_root()` to avoid "busy" errors

Milestone Checkpoint

After implementing the mount namespace component, verify the following behaviors:

Test Command:

```bash
# Compile and run with a simple test rootfs

sudo ./container-basic --rootfs /tmp/test-rootfs /bin/sh
```

Expected Behaviors:

1. Container process should see `/proc/self/mountinfo` different from host
2. Files created in container `/tmp` should not appear in host `/tmp`
3. Container should be able to mount/unmount without affecting host
4. Essential device nodes (`/dev/null`, `/dev/zero`) should be accessible
5. Container `/proc/mounts` should show only container-specific mount points

Signs of Problems:

- "Operation not permitted" → Check if running with sufficient privileges (CAP_SYS_ADMIN)
- "Invalid argument" from pivot_root → Verify new root is a mount point and old root directory exists
- "Device or resource busy" → Check that no processes have open files in mount directories
- Host shows container mounts → Mount propagation not set to private correctly

Debug Commands:

```bash
# Check mount propagation

cat /proc/self/mountinfo | grep -E "(shared|slave|private)"


# Verify namespace isolation

sudo lsns -t mnt


# Check device nodes in container

ls -la /dev/null /dev/zero /dev/random
```

Network Namespace Component

> **Milestone(s):** This section corresponds to Milestone 3 (Network Namespace), which implements network stack isolation and establishes container networking connectivity through veth pairs and bridge networking.

The network namespace component provides the foundation for container network isolation by creating a completely separate network stack for each container. This component coordinates with the PID and mount namespace components to ensure containers have their own network interfaces, IP addresses, routing tables, and firewall rules while maintaining connectivity to the host system and external networks.

Network isolation represents one of the most complex aspects of container implementation because it must balance complete isolation with practical connectivity requirements. Unlike PID or mount namespaces which primarily restrict visibility, network namespaces must actively establish communication channels between isolated environments. This creates intricate coordination challenges around interface creation, IP address management, and routing configuration that must be handled correctly to avoid network conflicts or connectivity failures.

Mental Model: Private Phone Systems

Think of network namespaces like separate office phone systems within a large corporate building. Each department (container) gets its own private phone system with internal extensions that are completely isolated from other departments. Employees within marketing can call each other using simple extension numbers (internal IP addresses), but they cannot directly dial into the engineering department's phone system.

However, complete isolation would make the departments unable to collaborate or contact the outside world. So the building installs trunk lines (veth pairs) that connect each department's phone system to a central switchboard (host bridge). When marketing needs to call engineering, the call routes through the central switchboard which knows how to forward calls between departments. Similarly, all external calls to clients or vendors flow through the central switchboard to reach the appropriate department.

The central switchboard (host networking stack) maintains a directory of which trunk line connects to which department and handles call routing between departments and to external networks. If marketing needs internet access, their calls flow through their trunk line to the switchboard, which then forwards them through the building's main phone line to the external provider. The switchboard can also enforce policies about which departments can call where and implement features like call forwarding or conferencing between multiple departments.

This analogy maps directly to container networking: each container gets its own network namespace (private phone system), veth pairs act as trunk lines connecting container namespaces to the host bridge (central switchboard), and the host kernel handles routing between containers and to external networks. Just as phone systems need proper wiring and configuration to work correctly, container networking requires careful setup of virtual interfaces, IP addresses, and routing rules.

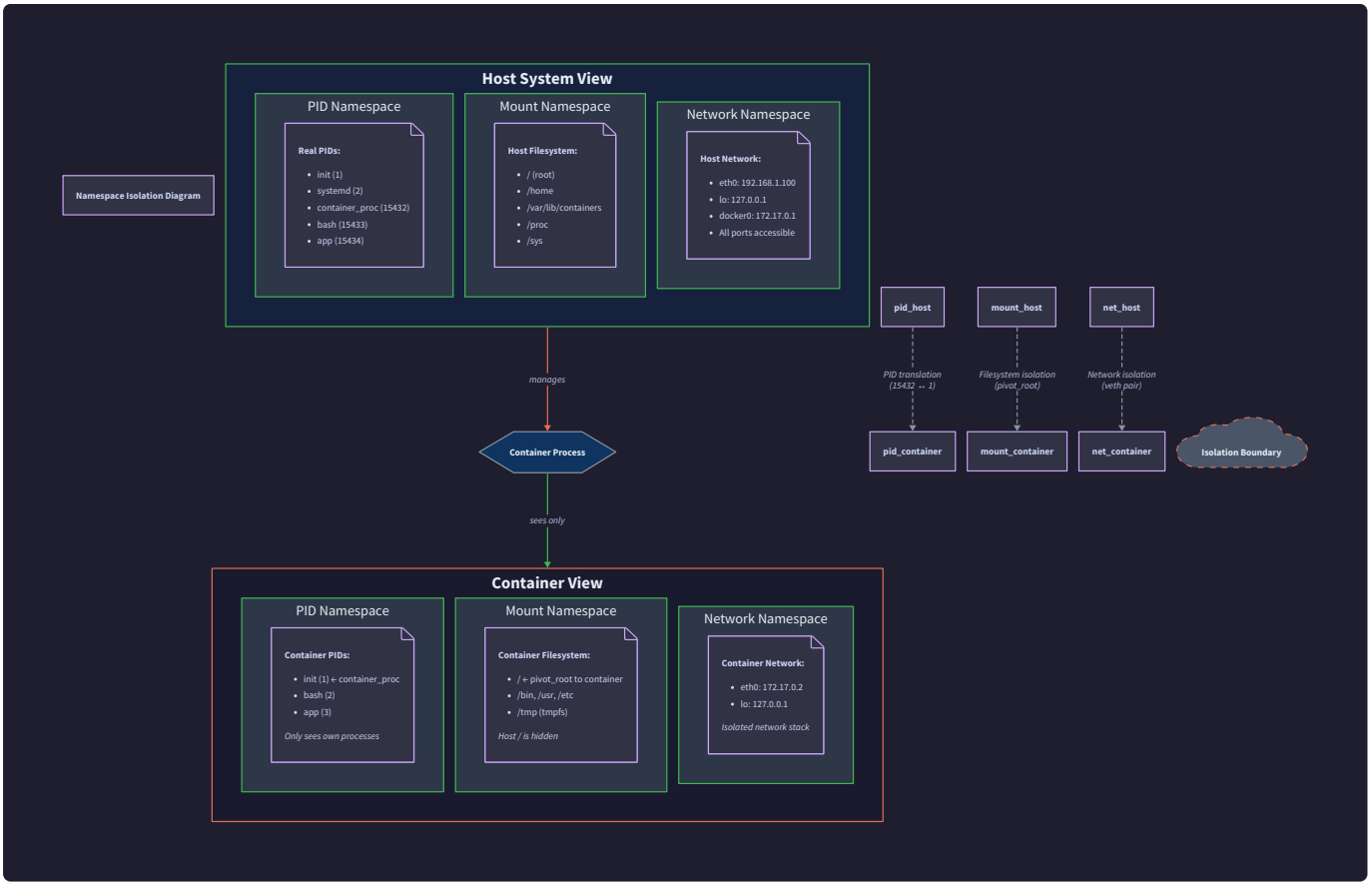
Network Namespace Interface

The network namespace interface provides functions for creating isolated network environments and establishing connectivity between containers and the host system. This interface abstracts the complexity of network namespace creation, veth pair management, and bridge configuration while ensuring proper cleanup and error handling.

The core responsibility of this component is coordinating the creation of network namespaces with the setup of communication channels that connect isolated containers to each other and external networks. This requires careful orchestration of kernel network namespace operations with user-space network configuration tools to establish a complete networking solution.

Function Name	Parameters	Returns	Description
`create_network_namespace`	config: `container_config_t*`, container: `container_instance_t*`	int (0 success, -1 error)	Creates new network namespace and stores namespace file descriptor for cleanup
`setup_container_networking`	config: `container_config_t*`, container_pid: `pid_t`	int (0 success, -1 error)	Configures veth pair and assigns container end to target namespace
`create_veth_pair`	host_if_name: `char*`, container_if_name: `char*`	int (0 success, -1 error)	Creates virtual ethernet pair with specified interface names
`assign_veth_to_namespace`	if_name: `char*`, target_pid: `pid_t`	int (0 success, -1 error)	Moves network interface to target process namespace
`configure_container_interface`	if_name: `char*`, ip_address: `char*`, netmask: `char*`	int (0 success, -1 error)	Sets IP address and brings interface up inside container namespace
`attach_to_bridge`	if_name: `char*`, bridge_name: `char*`	int (0 success, -1 error)	Adds host veth end to specified bridge for inter-container communication
`setup_default_route`	gateway_ip: `char*`	int (0 success, -1 error)	Configures default route inside container namespace for external connectivity
`cleanup_network_namespace`	container: `container_instance_t*`	int (0 success, -1 error)	Removes veth interfaces and closes namespace file descriptors
`verify_bridge_exists`	bridge_name: `char*`	int (1 exists, 0 missing)	Checks if specified bridge device exists on host system

Function Name	Parameters	Returns	Description
`allocate_container_ip`	bridge_name: `char*`, requested_ip: `char*`	char* (allocated IP, NULL on error)	Allocates available IP address from bridge subnet avoiding conflicts



The network namespace creation process begins when `create_network_namespace` is called during container startup. This function uses the `clone()` system call with the `CLONE_NEWNET` flag to create a new network namespace for the container process. The new namespace starts with only a loopback interface, completely isolated from the host network stack.

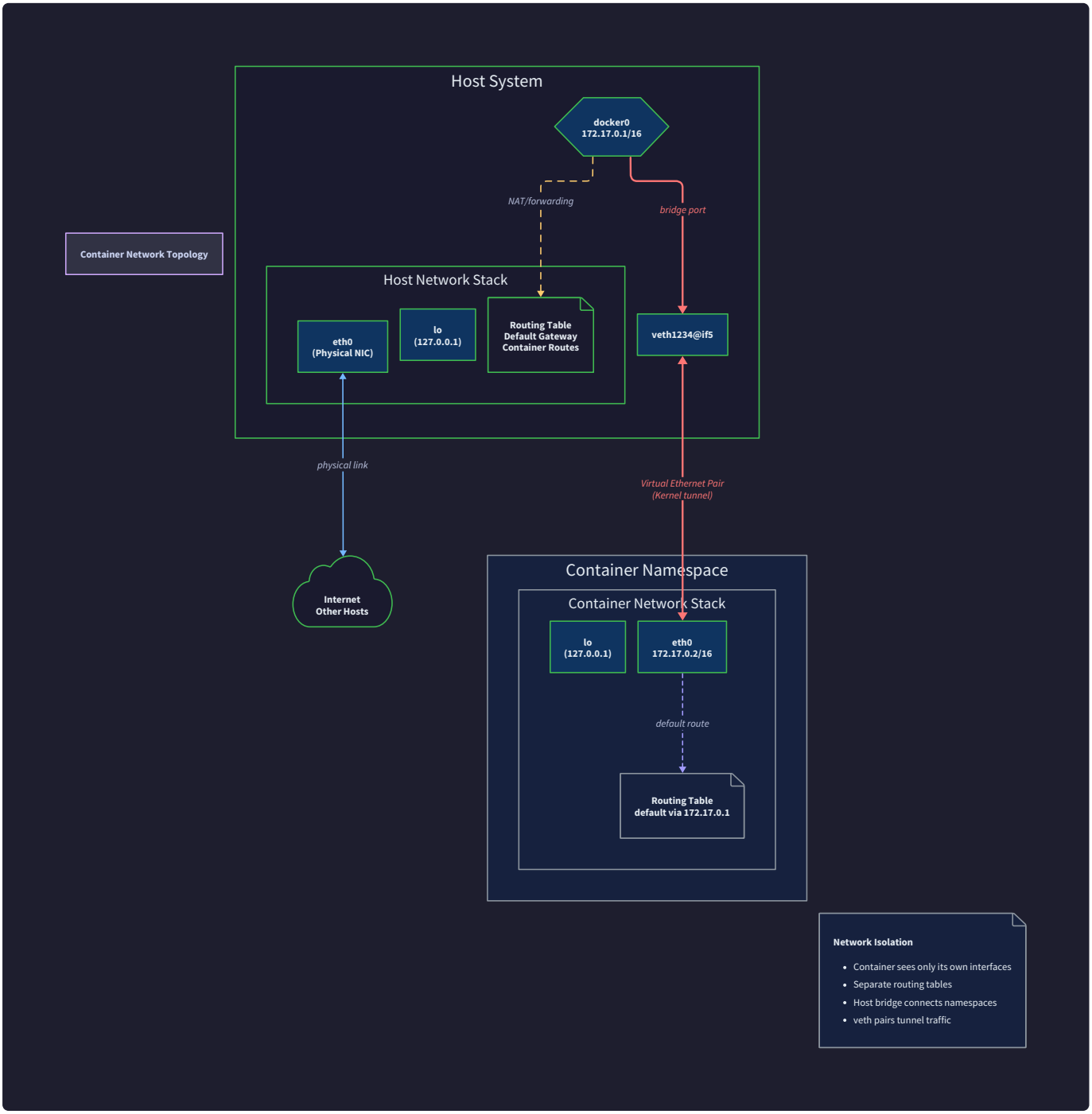
However, complete network isolation renders containers unable to communicate with each other or external systems, so the interface includes functions for establishing controlled connectivity. The `setup_container_networking` function orchestrates the creation of a veth pair that acts as a network tunnel between the isolated container namespace and the host system. This function coordinates with `create_veth_pair` to establish the virtual ethernet devices and `assign_veth_to_namespace` to move the container end of the pair into the target namespace.

Network configuration within the container namespace requires executing network configuration commands within the context of the isolated namespace. The `configure_container_interface` function handles IP address assignment and interface activation inside the container, while `setup_default_route` establishes routing rules for external connectivity. These functions must use namespace-aware network configuration approaches to ensure commands execute within the correct network context.

The interface provides robust cleanup capabilities through `cleanup_network_namespace`, which removes created veth interfaces and closes namespace file descriptors to prevent resource leaks. This function coordinates with the cleanup list mechanism to ensure proper teardown even during error conditions or partial setup failures.

Veth Pair Configuration

Veth pairs represent the fundamental mechanism for connecting isolated network namespaces to the host system and enabling container networking. A veth pair consists of two virtual ethernet interfaces that act as opposite ends of a virtual network cable - packets sent into one end emerge from the other end. This creates a bidirectional communication channel that can span namespace boundaries.



The veth pair configuration process involves several coordinated steps that must execute in the correct order to establish proper connectivity. First, the veth pair is created in the host namespace with both interfaces initially residing on the host. The kernel assigns the interfaces names like `veth0` and `veth1`, though these can be customized during creation to follow naming conventions that identify their purpose and associated container.

After creating the veth pair, one interface (typically the one intended for the container) must be moved into the target network namespace. This operation uses the `ip link set` command with network namespace targeting to transfer ownership of the interface from the host namespace to the container namespace. Once transferred, the interface becomes invisible to the host system and appears only within the container's isolated network view.

Configuration of each veth interface requires namespace-specific commands that execute within the appropriate network context. The host-side veth interface typically gets added to a bridge device that connects multiple containers and provides routing to external networks. The container-side interface requires IP address assignment, netmask configuration, and interface activation to become usable for network communication.

Configuration Step	Location	Command Example	Purpose
Create veth pair	Host namespace	`ip link add veth0 type veth peer name veth1`	Establishes virtual ethernet pair
Move to container	Host namespace	`ip link set veth1 netns <container-pid>`	Transfers interface to container namespace
Configure container interface	Container namespace	`ip addr add 172.17.0.2/24 dev veth1`	Assigns IP address to container interface
Activate container interface	Container namespace	`ip link set veth1 up`	Brings container interface online
Add to host bridge	Host namespace	`ip link set veth0 master br0`	Connects host interface to bridge
Activate host interface	Host namespace	`ip link set veth0 up`	Brings host interface online

The veth pair configuration must handle IP address allocation to prevent conflicts between containers and ensure proper subnet organization. Container IP addresses typically come from a private subnet associated with the host bridge, such as `172.17.0.0/16` for Docker-style networking. The configuration process must track allocated IP addresses and select available addresses for new containers while avoiding conflicts with existing containers or host system interfaces.

Bridge networking represents the standard approach for connecting multiple container veth pairs and providing shared network access. The host system maintains a bridge device (similar to a network switch) that connects the host-side veth interfaces from all containers. This bridge can forward traffic between containers, route traffic to external networks through the host's physical interfaces, and implement network policies like NAT (Network Address Translation) for outbound connectivity.

Veth pair cleanup requires careful coordination to avoid orphaned interfaces or namespace references. When a container terminates, both ends of its veth pair should be removed: the container-side interface disappears

automatically when the namespace is destroyed, but the host-side interface must be explicitly deleted. Failure to clean up host-side veth interfaces leads to accumulation of unused network interfaces that consume system resources and clutter network configuration.

The veth configuration process must also handle error conditions gracefully, particularly partial setup failures where some network configuration succeeds but other steps fail. The cleanup list mechanism ensures that successfully created interfaces get removed even if subsequent configuration steps encounter errors, preventing the system from accumulating partially configured network resources.

Architecture Decision Records

The network namespace component involves several critical architectural decisions that significantly impact performance, complexity, and functionality. These decisions affect how containers connect to networks, how IP addresses are managed, and how traffic flows between containers and external systems.

Decision: Veth Pairs vs MacVLAN for Container Networking

- **Context**: Containers need network connectivity while maintaining isolation. Linux provides multiple approaches for connecting namespaces to host networking, including veth pairs with bridges and MacVLAN interfaces that share physical network adapters.
- **Options Considered**: Veth pairs with bridge networking, MacVLAN interfaces, SR-IOV virtual functions
- **Decision**: Use veth pairs connected through a host bridge for container networking
- **Rationale**: Veth pairs provide complete traffic visibility on the host for monitoring and security policies, support arbitrary network topologies through bridge configuration, and work reliably across different host network configurations. MacVLAN requires specific network adapter features and limits traffic inspection capabilities.
- **Consequences**: Enables flexible network policies and traffic monitoring but requires bridge configuration and slightly increases network latency due to additional packet processing through the bridge.

Option	Pros	Cons	Chosen?
Veth + Bridge	Full traffic visibility, flexible topologies, universal adapter support	Additional latency, more complex setup	✅ Yes
MacVLAN	Lower latency, simpler container setup	Limited traffic visibility, adapter feature requirements	❌ No
SR-IOV	Highest performance, hardware acceleration	Expensive hardware requirements, complex management	❌ No

Decision: Bridge Networking vs Host Networking vs Overlay Networks

- **Context**: Multiple containers need to communicate with each other and external networks. Different networking models provide different levels of isolation, performance, and configuration complexity.
- **Options Considered**: Single host bridge for all containers, host networking without isolation, overlay networks with VXLAN tunneling
- **Decision**: Implement single host bridge networking with configurable bridge names
- **Rationale**: Host bridge networking provides good isolation between containers while enabling inter-container communication and external connectivity. It offers predictable performance characteristics and straightforward troubleshooting compared to overlay networks, while maintaining better security than host networking.
- **Consequences**: Limits containers to single-host networking but provides reliable, performant connectivity suitable for basic container use cases. Advanced multi-host scenarios require additional networking solutions.

Option	Pros	Cons	Chosen?
Host Bridge	Good isolation, inter-container communication, simple troubleshooting	Single-host limitation, manual IP management	✅ Yes
Host Networking	Maximum performance, no configuration needed	No network isolation, port conflicts	❌ No
Overlay Networks	Multi-host support, advanced features	Complex setup, performance overhead	❌ No

Decision: Static IP Assignment vs DHCP vs Automatic Allocation

- **Context**: Containers need IP addresses within the bridge subnet, but manual IP assignment is error-prone while DHCP adds complexity. The system needs a reliable approach for avoiding IP conflicts between containers.
- **Options Considered**: Manual static IP specification, DHCP server for automatic assignment, simple automatic allocation from IP pool
- **Decision**: Implement simple automatic IP allocation with optional static IP override
- **Rationale**: Automatic allocation from a predefined IP pool eliminates configuration errors and IP conflicts while remaining simple to implement and debug. DHCP adds complexity and external dependencies, while purely manual assignment creates operational burden and conflict potential.
- **Consequences**: Provides predictable IP allocation for most use cases while supporting static assignment when needed. Requires tracking allocated IP addresses but avoids complex DHCP infrastructure.

Option	Pros	Cons	Chosen?
Static Assignment	Predictable addresses, no coordination needed	Manual configuration, conflict potential	❌ No
DHCP	Industry standard, lease management	Complex setup, external dependencies	❌ No
Automatic Allocation	Simple implementation, conflict avoidance	IP tracking required, limited flexibility	✅ Yes

Decision: NAT vs Bridge Routing for External Connectivity

- **Context**: Containers with private IP addresses need access to external networks and internet services. This requires either NAT (Network Address Translation) to hide container IPs behind host IP or direct routing if container IPs are routable.
- **Options Considered**: NAT with iptables masquerading, direct routing with routable container subnets, proxy-based external access
- **Decision**: Implement NAT-based external connectivity using iptables masquerading
- **Rationale**: NAT provides external connectivity without requiring routable IP address allocation or complex routing configuration. It works reliably across different host network environments and provides additional security by hiding internal container addressing.
- **Consequences**: Enables external connectivity with simple host configuration but prevents direct inbound connections to containers without port forwarding. Adds slight performance overhead for address translation.

Option	Pros	Cons	Chosen?
NAT/Masquerading	Works in any network, simple setup, security benefits	No direct inbound access, translation overhead	✅ Yes
Direct Routing	Better performance, bidirectional connectivity	Requires routable IPs, complex routing	❌ No
Proxy Access	Fine-grained control, protocol-specific features	Application-specific, complex configuration	❌ No

These architectural decisions establish a networking model that balances simplicity, functionality, and reliability for basic container networking requirements. The veth pair and bridge approach provides a solid foundation that can be extended with additional features like port forwarding, network policies, or multi-host connectivity as requirements evolve.

Common Pitfalls

Network namespace implementation presents several subtle pitfalls that can cause connectivity failures, resource leaks, or security issues. These problems often manifest as intermittent failures or unexpected behavior that can be difficult to diagnose without understanding the underlying timing and coordination requirements.

⚠️ Pitfall: Namespace Timing Issues During Interface Assignment

A common mistake is attempting to configure network interfaces before the target namespace is fully established or accessible. When moving a veth interface to a container namespace using `ip link set veth1 netns <pid>`, the operation can fail if the target process has not yet called `unshare(CLONE_NEWNET)` or if the process has already exited.

This timing issue typically occurs when the parent process creates veth pairs and immediately tries to assign them to the child container process, but the child process hasn't yet established its network namespace or has encountered an error during startup. The assignment operation fails with "No such file or directory" or "Invalid argument" errors that don't clearly indicate the timing problem.

To avoid this pitfall, implement proper synchronization between parent and child processes during container startup. Use signaling mechanisms like pipes or shared memory to ensure the child process has successfully created its network namespace before the parent attempts interface assignment. Additionally, implement retry logic with appropriate timeouts for interface assignment operations to handle minor timing variations.

⚠️ Pitfall: Veth Interface Cleanup and Orphaned Interfaces

Failing to properly clean up veth interfaces leads to accumulation of orphaned network interfaces on the host system. When a container terminates unexpectedly or cleanup functions encounter errors, the host-side veth interface may remain active even though the container-side interface has disappeared with the destroyed namespace.

These orphaned interfaces consume system resources, clutter network interface listings, and can cause naming conflicts when creating new containers. They may also remain attached to bridge devices, causing confusion during network troubleshooting and potentially affecting bridge behavior.

Implement comprehensive cleanup tracking using the cleanup list mechanism to ensure veth interfaces are removed even during error conditions. Store interface names in the cleanup list immediately after creation, and implement cleanup functions that gracefully handle cases where interfaces may have already been removed. Use network interface enumeration to detect and clean up orphaned interfaces during system startup or periodic maintenance.

⚠️ Pitfall: Bridge Configuration and Dependency Ordering

Attempting to attach veth interfaces to bridges that don't exist or haven't been properly configured causes network setup failures. Many implementations assume that bridge devices exist and are properly configured, but fail to verify bridge state or create necessary bridge configuration before attempting to use them.

Bridge configuration issues also include problems with IP forwarding, NAT rules, and routing table entries that must be established on the host system for container networking to function correctly. Missing or incorrect iptables rules prevent external connectivity even when internal container networking appears to work correctly.

Implement bridge existence verification and automatic bridge creation as part of the network setup process. Create comprehensive bridge configuration functions that handle IP forwarding enablement, NAT rule installation, and routing table management. Use dependency ordering to ensure bridge configuration completes before attempting to attach container interfaces.

⚠️ Pitfall: IP Address Conflicts and Allocation Tracking

Simple IP address allocation without conflict detection leads to multiple containers receiving the same IP address, causing network connectivity problems that are difficult to diagnose. This typically occurs when IP allocation doesn't track previously assigned addresses or when containers terminate without properly releasing their IP allocations.

IP conflicts manifest as intermittent connectivity issues, ARP (Address Resolution Protocol) conflicts, and unpredictable network behavior where traffic intended for one container reaches another container. These problems become more severe as the number of containers increases and IP address reuse becomes more common.

Implement proper IP address allocation tracking using persistent storage or in-memory data structures that survive container lifecycle events. Include IP address validation during allocation to detect and avoid conflicts with existing network interfaces. Implement IP address release mechanisms that execute during container cleanup to ensure addresses become available for reuse.

⚠️ Pitfall: NAT Configuration and iptables Rule Management

Incorrect NAT configuration prevents container external connectivity and can interfere with host system networking. Common mistakes include missing MASQUERADE rules, incorrect source/destination specifications in iptables rules, and failure to enable IP forwarding at the kernel level.

NAT configuration problems often manifest as containers being able to communicate with each other but unable to reach external networks or receive responses to outbound connections. These issues can be particularly confusing because DNS resolution may work (if using internal DNS servers) while actual service connectivity fails.

Implement systematic NAT configuration with proper rule validation and testing. Create iptables rule management functions that handle rule installation, removal, and conflict detection. Include IP forwarding enablement as part of the standard network setup process, and implement connectivity testing to verify external access after NAT configuration.

⚠️ Pitfall: Network Namespace File Descriptor Management

Network namespaces must be properly referenced through file descriptors to ensure they persist even after the creating process exits. Failing to maintain namespace file descriptors can cause namespaces to be destroyed prematurely, leading to network connectivity loss and cleanup problems.

Network namespace file descriptors require careful management during process lifecycle events, particularly when container processes fork, exec, or terminate. Improper file descriptor handling can lead to namespace reference leaks or premature namespace destruction that breaks container networking.

Implement explicit network namespace file descriptor management with proper cleanup tracking. Store namespace file descriptors in the container instance structure and ensure they are properly closed during

container termination. Use file descriptor validation to detect and handle cases where namespaces have been unexpectedly destroyed.

Implementation Guidance

The network namespace implementation requires careful coordination of kernel networking features with user-space network configuration tools. This component builds upon the foundation established by the PID and mount namespace components while adding the complexity of network interface management and inter-process communication for network setup.

Technology Recommendations

Component	Simple Option	Advanced Option
Network Configuration	System calls to `ip` command via `system()`	Direct netlink socket programming for kernel communication
Bridge Management	Bridge utilities (`brctl` command)	Native netlink interface manipulation
IP Address Management	Static allocation from predefined subnet	Dynamic allocation with conflict detection and persistence
NAT Configuration	iptables command execution	Direct netfilter programming

Recommended File Structure

```
project-root/
  cmd/container/main.c          ← main container runtime entry point
  src/network/                  ← network namespace component
    network_namespace.c         ← core network namespace functions
    veth_manager.c              ← veth pair creation and configuration
    bridge_config.c             ← bridge setup and management
    ip_allocator.c              ← IP address allocation and tracking
    network_namespace.h         ← public interface definitions
    network_internal.h          ← internal types and helper functions
  src/common/                   ← shared infrastructure
    cleanup.c                   ← cleanup list implementation
    syscall_wrappers.c          ← safe system call wrappers
  tests/network/                ← network-specific tests
    test_network_isolation.c    ← namespace isolation verification
    test_veth_setup.c           ← veth pair configuration tests
    test_connectivity.c         ← end-to-end networking tests
```

Infrastructure Starter Code

Network Configuration Helper (`src/network/network_utils.c`):

```c
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>

#include <errno.h>

#include "network_internal.h"


// Execute network configuration command and return exit status

int exec_network_command(const char* command) {

    int status = system(command);

    if (status == -1) {

        perror("system() failed");

        return -1;

    }

    return WEXITSTATUS(status);

}


// Execute command within specific network namespace context

int exec_in_netns(pid_t target_pid, const char* command) {

    char full_command[512];

    snprintf(full_command, sizeof(full_command),

             "nsenter -t %d -n %s", target_pid, command);

    return exec_network_command(full_command);

}


// Check if network interface exists

int interface_exists(const char* if_name) {

    char path[256];
```

```
    snprintf(path, sizeof(path), "/sys/class/net/%s", if_name);
    return access(path, F_OK) == 0;
}

// Generate unique interface name for container
void generate_veth_names(const char* container_id,
                         char* host_if, size_t host_len,
                         char* container_if, size_t container_len) {
    snprintf(host_if, host_len, "veth%s", container_id);
    snprintf(container_if, container_len, "eth0");

    // Truncate if name too long for kernel limits
    if (strlen(host_if) >= IFNAMSIZ) {
        host_if[IFNAMSIZ-1] = '\0';
    }
}

// Validate IP address format
int is_valid_ip(const char* ip_str) {
    struct sockaddr_in sa;
    return inet_pton(AF_INET, ip_str, &(sa.sin_addr)) == 1;
}
```

Bridge Management Helper (`src/network/bridge_utils.c`):

```c
#include <stdio.h>

#include <string.h>

#include "network_internal.h"


// Ensure bridge device exists and is properly configured

int ensure_bridge_ready(const char* bridge_name, const char* bridge_ip) {

    char command[256];



    // Check if bridge already exists

    if (!interface_exists(bridge_name)) {

        // Create bridge device

        snprintf(command, sizeof(command), "ip link add %s type bridge", bridge_name);

        if (exec_network_command(command) != 0) {

            fprintf(stderr, "Failed to create bridge %s\n", bridge_name);

            return -1;

        }



        // Assign IP address to bridge

        snprintf(command, sizeof(command), "ip addr add %s dev %s", bridge_ip, bridge_name);

        if (exec_network_command(command) != 0) {

            fprintf(stderr, "Failed to assign IP to bridge %s\n", bridge_name);

            return -1;

        }



        // Bring bridge interface up

        snprintf(command, sizeof(command), "ip link set %s up", bridge_name);

        if (exec_network_command(command) != 0) {

            fprintf(stderr, "Failed to bring up bridge %s\n", bridge_name);
```

```
            return -1;
        }
    }

    return 0;
}

// Enable IP forwarding for container networking
int enable_ip_forwarding() {
    FILE* fp = fopen("/proc/sys/net/ipv4/ip_forward", "w");
    if (fp == NULL) {
        perror("Failed to open ip_forward");
        return -1;
    }

    if (fprintf(fp, "1") < 0) {
        perror("Failed to enable IP forwarding");
        fclose(fp);
        return -1;
    }

    fclose(fp);
    return 0;
}

// Setup NAT rules for container external connectivity
int setup_nat_rules(const char* bridge_name, const char* subnet) {
    char command[512];
```

```
    // Add MASQUERADE rule for outbound traffic
    snprintf(command, sizeof(command),
             "iptables -t nat -A POSTROUTING -s %s ! -o %s -j MASQUERADE",
             subnet, bridge_name);

    if (exec_network_command(command) != 0) {
        fprintf(stderr, "Failed to setup NAT masquerading\n");
        return -1;
    }

    // Allow forwarding for bridge traffic
    snprintf(command, sizeof(command),
             "iptables -A FORWARD -i %s ! -o %s -j ACCEPT", bridge_name, bridge_name);

    if (exec_network_command(command) != 0) {
        fprintf(stderr, "Failed to setup bridge forwarding\n");
        return -1;
    }

    return 0;
}
```

Core Logic Skeleton Code

Main Network Namespace Functions (`src/network/network_namespace.c`):

```c
#include "network_namespace.h"

#include "network_internal.h"


// Create new network namespace and setup basic networking

int create_network_namespace(container_config_t* config, container_instance_t* container) {

    // TODO 1: Verify bridge exists and create if necessary using ensure_bridge_ready()

    // TODO 2: Enable IP forwarding and setup NAT rules for external connectivity

    // TODO 3: Store current network namespace fd in container->namespace_fds[NETNS_INDEX]

    // TODO 4: Network namespace creation happens during clone() - just setup cleanup here

    // TODO 5: Register cleanup function for network namespace file descriptor

    // Hint: Network namespace created automatically with CLONE_NEWNET flag in clone()

}


// Setup container networking after process creation

int setup_container_networking(container_config_t* config, container_instance_t* container) {

    // TODO 1: Generate unique veth pair names using container ID

    // TODO 2: Create veth pair in host namespace using create_veth_pair()

    // TODO 3: Move container-side veth to target namespace using assign_veth_to_namespace()

    // TODO 4: Configure host-side veth and attach to bridge using attach_to_bridge()

    // TODO 5: Configure container-side interface IP and routes using
configure_container_interface()

    // TODO 6: Register cleanup functions for created veth interfaces

    // Hint: Use container->child_pid as target for namespace assignment

}


// Create virtual ethernet pair for container networking

int create_veth_pair(const char* host_if_name, const char* container_if_name) {

    // TODO 1: Construct ip command to create veth pair with specified names

    // TODO 2: Execute command using exec_network_command() and check return code

    // TODO 3: Verify both interfaces were created using interface_exists()
```

```
    // TODO 4: Return 0 on success, -1 on failure with appropriate error logging
    // Hint: Command format: "ip link add <host> type veth peer name <container>"
}

// Assign network interface to target namespace
int assign_veth_to_namespace(const char* if_name, pid_t target_pid) {
    // TODO 1: Construct ip command to move interface to target namespace
    // TODO 2: Execute command and handle potential timing issues with retries
    // TODO 3: Verify interface no longer exists in host namespace
    // TODO 4: Log success/failure with specific error information
    // Hint: Command format: "ip link set <interface> netns <pid>"
}

// Configure IP address and routes inside container namespace
int configure_container_interface(const char* if_name, const char* ip_address,
                                  const char* netmask, pid_t container_pid) {
    // TODO 1: Assign IP address to interface using exec_in_netns()
    // TODO 2: Bring interface up inside container namespace
    // TODO 3: Add default route through bridge gateway
    // TODO 4: Verify connectivity with ping test to gateway
    // Hint: Use exec_in_netns() to run commands in container's network namespace
}

// Cleanup network namespace resources
int cleanup_network_namespace(container_instance_t* container) {
    // TODO 1: Remove host-side veth interfaces if they still exist
    // TODO 2: Close network namespace file descriptor
    // TODO 3: Execute registered cleanup functions in reverse order
    // TODO 4: Clear namespace references from container structure
```

```
    // Hint: Container namespace destruction happens automatically when process exits
}
```

IP Address Allocation (`src/network/ip_allocator.c`):

```c
#include "network_internal.h"

// Simple IP address allocator for container networking
char* allocate_container_ip(const char* bridge_name, const char* requested_ip) {
    // TODO 1: If requested_ip provided and valid, check for conflicts and return if available
    // TODO 2: Otherwise, scan bridge subnet (e.g., 172.17.0.0/24) for available IP
    // TODO 3: Check each potential IP against existing interfaces using ping test
    // TODO 4: Return allocated IP address string, or NULL if allocation failed
    // TODO 5: Store allocation in tracking structure for conflict prevention
    // Hint: Start scanning from .2 (after gateway .1) and avoid broadcast addresses
}

// Release IP address back to available pool
int release_container_ip(const char* ip_address) {
    // TODO 1: Remove IP from allocation tracking structure
    // TODO 2: Perform any cleanup of IP-specific routes or rules
    // TODO 3: Return 0 on success, -1 if IP was not tracked
    // Hint: Implementation can be simple file-based or in-memory tracking
}
```

Language-Specific Hints

System Call Integration:

- Use `system()` for executing network configuration commands during development, but consider netlink sockets for production implementations
- Network namespace file descriptors can be opened from `/proc/<pid>/ns/net` for explicit namespace management
- The `nsenter` command provides reliable namespace context switching for configuration commands

Error Handling:

- Network configuration commands can fail for many reasons (permissions, missing features, timing issues)
- Implement retry logic for operations that may encounter timing issues during namespace creation
- Use specific error messages that include the failed command and system error information

Debugging Support:

- Log all executed network commands and their return codes for troubleshooting
- Include network interface enumeration in debug output to show current system state
- Implement connectivity testing functions that verify network setup success

Milestone Checkpoint

After implementing the network namespace component, verify the following behavior:

Compilation and Basic Testing:

```bash
cd project-root

make clean && make

sudo ./container-basic --network-test
```

Expected Network Isolation:

1. Container process should have its own network namespace with only loopback interface initially
2. After veth setup, container should have `eth0` interface with assigned IP address
3. Container should be able to ping bridge gateway and reach external networks
4. Host system should show veth interface connected to bridge with appropriate naming

Manual Verification Steps:

```bash
# Start container in background
sudo ./container-basic --config test.conf --background

# Check namespace isolation
sudo lsns -t net  # Should show separate network namespace for container

# Verify veth pair creation
ip link show | grep veth  # Should show host-side veth interface

# Test container connectivity
sudo nsenter -t <container-pid> -n ping -c 1 8.8.8.8  # Should succeed

# Verify bridge attachment
brctl show  # Should show veth interface attached to bridge
```

Debugging Signs:

- **Container cannot reach external networks**: Check NAT rules and IP forwarding configuration
- **Veth interfaces not created**: Verify network namespace creation and timing coordination
- **IP address conflicts**: Check IP allocation logic and conflict detection
- **Bridge connectivity fails**: Verify bridge configuration and interface attachment

Cgroups Resource Management Component

> **Milestone(s):** This section corresponds to Milestone 4 (Cgroups Resource Limits), which implements resource control and enforcement using Linux cgroups to limit CPU, memory, and process counts within containers.

Mental Model: Budget Allocation

Think of cgroups like a household budget system with automatic enforcement. In a family budget, you allocate specific amounts for different categories: $500 for groceries, $200 for entertainment, $300 for utilities. Each category has a hard limit - when the grocery budget is exhausted, no more grocery purchases are allowed until next month.

Cgroups work similarly for system resources. You create a "resource budget" for your container: 512MB of memory, 50% of one CPU core, maximum 100 processes. The kernel automatically enforces these limits - when the container tries to allocate its 513th megabyte of memory, the kernel denies the allocation (and may terminate

the process). When it tries to use more than 50% CPU over a time period, the kernel throttles the container's processes.

The key insight is that cgroups provide **automatic enforcement** - you don't have to manually monitor resource usage. The kernel does the accounting and applies limits transparently. This is like having a bank that automatically declines transactions when you exceed your budget category limits, rather than requiring you to manually track every expense.

Just as budget categories can be nested (entertainment budget might have sub-categories for movies vs games), cgroups form a hierarchy. You can create a parent cgroup for all containers with an overall memory limit, then create child cgroups for individual containers with smaller limits within that parent's allocation.

Cgroups Interface

The cgroups interface provides functions for creating resource-limited execution contexts and enforcing those limits throughout the container lifecycle. Unlike namespaces which provide isolation, cgroups provide **resource control** - they determine how much of each system resource type the container can consume.

The core challenge in cgroups management is handling both cgroups v1 and v2 systems, which have different filesystem layouts and control interfaces. Our design abstracts this complexity behind a unified interface that detects the available cgroup version and adapts accordingly.

Function Name	Parameters	Returns	Description
`create_container_cgroup`	`config container_config_t*`, `container container_instance_t*`	`int`	Creates cgroup hierarchy for container and configures resource limits
`assign_process_to_cgroup`	`cgroup_path char*`, `pid pid_t`	`int`	Assigns process to cgroup by writing PID to tasks file
`set_memory_limit`	`cgroup_path char*`, `limit_bytes size_t`	`int`	Configures memory limit and OOM behavior for cgroup
`set_cpu_limit`	`cgroup_path char*`, `cpu_percent int`	`int`	Configures CPU quota and period to limit CPU usage percentage
`set_process_limit`	`cgroup_path char*`, `max_processes int`	`int`	Configures maximum number of processes/threads in cgroup
`cleanup_container_cgroup`	`cgroup_path char*`	`int`	Removes cgroup directory and cleans up kernel resources
`detect_cgroup_version`	`void`	`int`	Detects whether system uses cgroups v1 or v2 and returns version
`get_cgroup_path`	`container_id char*`, `controller char*`	`char*`	Constructs filesystem path for container's cgroup in specified controller
`verify_controller_available`	`controller char*`	`int`	Checks if specified cgroup controller is available and enabled
`read_cgroup_stat`	`cgroup_path char*`, `stat_name char*`	`long`	Reads current resource usage statistics from cgroup

The cgroup creation process follows a specific sequence to ensure proper resource limit enforcement. First, we detect which cgroup version is available on the system, as this determines both the filesystem layout and the control file formats. Then we create the cgroup directory hierarchy, configure resource limits in the appropriate control files, and finally assign the container process to the cgroup.

> **Design Principle**: Cgroups must be created **before** the container process starts to ensure resource limits are enforced from the very beginning of execution. Creating cgroups after process startup creates a window where the process can consume unlimited resources.

The cgroup cleanup process is equally critical. When a container exits, the cgroup must be properly destroyed to prevent resource leaks in the kernel's cgroup accounting system. However, cleanup can only occur after all processes in the cgroup have exited - attempting to remove a cgroup with active processes will fail.

Resource Controllers

Linux cgroups provide multiple **controllers** that each manage a different type of system resource. Our container implementation focuses on three essential controllers: memory, CPU, and process limits (pids). Each controller has its own control interface and enforcement mechanisms.

Memory Controller

The memory controller tracks and limits memory usage for all processes within a cgroup. This includes anonymous memory (heap allocations), page cache, and kernel memory allocated on behalf of the cgroup's processes.

Control File	Purpose	Example Value	Effect
`memory.limit_in_bytes` (v1) / `memory.max` (v2)	Maximum memory allocation	`536870912` (512MB)	Hard limit - allocations beyond this trigger OOM
`memory.usage_in_bytes` (v1) / `memory.current` (v2)	Current memory usage	`134217728` (128MB)	Read-only current usage counter
`memory.oom_control` (v1) / `memory.oom.group` (v2)	OOM killer behavior	`oom_kill_disable 0`	Controls whether OOM killer terminates processes
`memory.swappiness`	Swap preference	`60`	How aggressively to use swap (0-100)

The memory controller enforcement works through the kernel's memory allocation paths. When a process in the cgroup requests memory (via `malloc()`, `mmap()`, or kernel allocations), the kernel checks the current cgroup usage against the limit. If the allocation would exceed the limit, the kernel either denies the allocation or invokes the OOM (Out of Memory) killer to terminate processes and free memory.

> **Critical Insight**: Memory limits are enforced at allocation time, not usage time. A process that gradually increases its memory usage will trigger OOM when it crosses the limit, but a process that maps large virtual memory regions may fail earlier when trying to actually access those pages.

Memory controller configuration involves several steps:

1. Write the memory limit in bytes to the appropriate control file
2. Configure OOM behavior - whether to kill processes or pause the cgroup
3. Set memory swappiness to control swap usage preferences
4. Monitor memory usage through the usage statistics file

CPU Controller

The CPU controller limits the amount of CPU time that processes in a cgroup can consume over time periods. Unlike memory limits which are hard caps, CPU limits are implemented through **quota and period** scheduling - the cgroup gets a quota of CPU time within each scheduling period.

Control File	Purpose	Example Value	Effect
`cpu.cfs_quota_us` (v1) / `cpu.max` (v2)	CPU time quota per period	`50000` (50ms)	Maximum CPU time in microseconds
`cpu.cfs_period_us` (v1) / `cpu.max` (v2)	Scheduling period length	`100000` (100ms)	Period length in microseconds
`cpu.stat`	CPU usage statistics	`nr_periods 1234`	Read-only usage and throttling stats
`cpu.shares` (v1) / `cpu.weight` (v2)	Relative CPU priority	`1024`	Share of CPU when competing with other cgroups

The CPU quota system works by giving each cgroup a time budget that refreshes every period. For example, to limit a container to 50% of one CPU core, you set a quota of 50,000 microseconds (50ms) with a period of 100,000 microseconds (100ms). The container's processes can run for up to 50ms out of every 100ms period. When the quota is exhausted, the kernel suspends the cgroup's processes until the next period begins.

CPU controller configuration requires calculating the appropriate quota and period values:

1. Determine the desired CPU percentage (from `container_config_t.cpu_percent`)
2. Choose a period length (typically 100ms for good responsiveness)
3. Calculate quota as: `quota = (cpu_percent / 100) * period`
4. Write both values to the control files
5. Monitor CPU throttling through statistics files

Process Controller (PIDs)

The process controller limits the number of processes and threads that can be created within a cgroup. This prevents fork bombs and runaway process creation from consuming all available PIDs on the system.

Control File	Purpose	Example Value	Effect
`pids.max`	Maximum number of processes	`100`	Hard limit on process/thread count
`pids.current`	Current process count	`23`	Read-only current process counter
`pids.events`	Limit violation events	`max 5`	Counter of times limit was hit

The process controller enforcement occurs during process creation system calls (`fork()`, `clone()`, `vfork()`). When a process in the cgroup attempts to create a new process or thread, the kernel increments the cgroup's process counter and checks it against the limit. If the limit would be exceeded, the system call fails with `EAGAIN`.

Process limit configuration is straightforward but requires careful consideration of the container's needs:

1. Analyze the expected process tree structure (main process + children)
2. Account for system processes like init handlers and signal processors
3. Add buffer for temporary processes created during normal operation
4. Write the limit to `pids.max` control file
5. Monitor process creation failures through the events file

Architecture Decision Records

The cgroups component requires several critical architectural decisions that significantly impact both implementation complexity and runtime behavior.

> **Decision: Cgroups Version Support Strategy**
>
> - **Context**: Linux systems may have cgroups v1, v2, or both available. The two versions have different filesystem layouts, control interfaces, and feature sets. We need to support both to ensure compatibility across different Linux distributions and kernel versions.
> - **Options Considered**: 1) Support only cgroups v2 and require modern systems, 2) Support only cgroups v1 for maximum compatibility, 3) Support both versions with runtime detection
> - **Decision**: Support both cgroups v1 and v2 with runtime detection and abstraction
> - **Rationale**: Many production systems still use cgroups v1 (RHEL 7, older Ubuntu versions), while newer systems are migrating to v2. Runtime detection allows our container to work across the widest range of systems without requiring users to modify their kernel configuration.
> - **Consequences**: Increases implementation complexity with dual code paths, but provides maximum compatibility and future-proofs against the v1 to v2 transition.

Option	Pros	Cons
cgroups v2 only	Simpler code, modern interface, unified hierarchy	Incompatible with older systems, limited adoption
cgroups v1 only	Maximum compatibility, well-tested interface	Missing modern features, deprecated technology
Both versions	Works everywhere, smooth migration path	Complex implementation, dual maintenance burden

Decision: Controller Selection and Prioritization

- **Context**: Linux cgroups provide many controllers (memory, cpu, cpuset, devices, freezer, net_cls, etc.). Each adds complexity but provides specific resource control capabilities. We need to choose which controllers are essential for basic container functionality.
- **Options Considered**: 1) Implement all available controllers for complete resource control, 2) Implement only memory and CPU for basic limits, 3) Implement memory, CPU, and PIDs as core set with extensible architecture
- **Decision**: Implement memory, CPU, and PIDs controllers as the core set with extensible architecture
- **Rationale**: Memory and CPU are the most commonly needed limits for container resource management. PIDs controller prevents fork bombs which are a common security concern. This core set covers 90% of use cases while keeping complexity manageable. The extensible architecture allows adding more controllers later.
- **Consequences**: Provides essential resource control without overwhelming complexity. Some advanced use cases (device access control, network traffic shaping) require additional controller implementation.

Controller	Essential?	Use Case	Implementation Priority
memory	Yes	Prevent OOM on host, enforce resource allocation	1
cpu	Yes	Fair CPU sharing, prevent CPU monopolization	2
pids	Yes	Prevent fork bombs, limit process explosion	3
devices	No	Control device node access	Future
freezer	No	Pause/resume containers	Future
net_cls	No	Network traffic classification	Future

Decision: Cgroup Cleanup Strategy

- **Context**: Cgroups must be cleaned up when containers exit to prevent resource leaks in the kernel. However, cleanup can fail if processes are still running in the cgroup, or if the cgroup hierarchy has dependencies. We need a robust cleanup strategy that handles various failure modes.
- **Options Considered**: 1) Simple removal on container exit, 2) Retry-based cleanup with exponential backoff, 3) Hierarchical cleanup with process termination
- **Decision**: Hierarchical cleanup with process termination and retry logic
- **Rationale**: Simple removal fails when zombie processes remain in the cgroup. Hierarchical cleanup ensures we terminate any remaining processes before removing cgroups, preventing resource leaks. Retry logic handles race conditions between process termination and cgroup removal.
- **Consequences**: More complex cleanup logic but prevents kernel resource leaks and handles edge cases robustly. May delay container exit in pathological cases where processes resist termination.

Cleanup Strategy	Pros	Cons
Simple removal	Fast, minimal code	Fails with zombie processes, leaks resources
Retry-based	Handles race conditions	May retry indefinitely, unclear failure modes
Hierarchical + retry	Robust, prevents leaks	Complex implementation, potential delays

Common Pitfalls

The cgroups resource management system has several subtle pitfalls that can lead to resource leaks, enforcement failures, or system instability. Understanding these pitfalls is crucial for implementing reliable container resource controls.

⚠️ Pitfall: Controller Availability Assumptions

A common mistake is assuming that all desired cgroup controllers are available and enabled on the target system. Different Linux distributions enable different controller sets by default, and administrators may disable controllers for performance or security reasons.

The symptom appears when your container creation succeeds but resource limits are silently ignored. For example, you set a memory limit of 512MB, but the container can actually allocate 2GB without restriction. This occurs because the memory controller wasn't available, so the limit configuration failed silently.

Why it's wrong: Failing to verify controller availability means resource limits may not be enforced, creating a security vulnerability where containers can consume unlimited system resources.

How to fix it: Always call `verify_controller_available()` for each required controller before attempting to configure limits. If essential controllers are missing, fail container creation with a clear error message rather than proceeding with unprotected resource usage.

⚠️ Pitfall: Incorrect Cleanup Order

Another frequent mistake is attempting to remove cgroups before all processes in the cgroup have fully exited. The Linux kernel refuses to remove cgroups that contain active processes, but the error handling for this scenario is often inadequate.

The symptom is that container cleanup appears to succeed, but `/sys/fs/cgroup` accumulates orphaned cgroup directories over time. Eventually, this can exhaust the kernel's cgroup limits and prevent new container creation.

Why it's wrong: Orphaned cgroups consume kernel memory and count against system cgroup limits. Over time, these leaks can prevent new container creation and require system reboot to resolve.

How to fix it: Implement hierarchical cleanup that first terminates all processes in the cgroup (using `SIGTERM` followed by `SIGKILL` if necessary), waits for process exit confirmation, then removes the cgroup directory. Use retry logic with exponential backoff to handle race conditions between process termination and cgroup removal.

⚠️ Pitfall: Memory Limit and OOM Behavior Mismatch

A subtle issue occurs when setting memory limits without properly configuring OOM (Out of Memory) behavior. By default, the kernel's OOM killer selects victims based on heuristics that may kill processes outside the cgroup rather than within it.

The symptom is that when a container exceeds its memory limit, random processes on the host system get terminated instead of processes within the offending container. This can cause system instability and difficult-to-debug application failures.

Why it's wrong: The purpose of memory limits is to contain the impact of memory pressure within the container. If OOM kills affect the host or other containers, the isolation is broken.

How to fix it: Configure `memory.oom.group` (cgroups v2) or `memory.use_hierarchy` (cgroups v1) to ensure OOM kills are contained within the cgroup. Set appropriate OOM score adjustments to prefer killing container processes over system processes.

⚠️ Pitfall: CPU Quota and Period Misconfiguration

CPU limits are commonly misconfigured by using inappropriate quota and period values. Setting periods that are too short causes excessive scheduling overhead, while periods that are too long cause poor responsiveness.

The symptom appears as either poor container performance (high scheduling overhead) or "bursty" CPU usage patterns where containers alternate between full CPU usage and complete suspension.

Why it's wrong: Incorrect CPU quota/period configuration can actually make CPU performance worse than having no limits at all, due to scheduling overhead and poor cache locality.

How to fix it: Use standard period values (100ms is typical) and calculate quotas as a percentage of that period. For multi-core limits, use quotas larger than the period (e.g., 150ms quota with 100ms period = 1.5 cores). Monitor `cpu.stat` to detect excessive throttling.

⚠️ Pitfall: Race Conditions in Process Assignment

A timing issue occurs when assigning processes to cgroups after those processes have already started and potentially created child processes. The child processes inherit the parent's cgroup membership at creation time,

so they may end up in different cgroups than intended.

The symptom is that resource limits appear to be working for the main container process, but child processes created early in container startup consume resources outside the limits.

Why it's wrong: Incomplete process assignment to cgroups allows resource usage to escape the intended limits, defeating the purpose of resource control.

How to fix it: Create cgroups and assign the container process to them **before** calling `exec()` to start the container's main process. Use the `register_cleanup()` mechanism to ensure proper cgroup cleanup even if process assignment fails.

Implementation Guidance

The cgroups resource management component requires careful integration with the Linux cgroup filesystem and proper handling of both cgroups v1 and v2 variants. This implementation bridges the design concepts with practical code organization and error handling.

Technology Recommendations

Component	Simple Option	Advanced Option
Cgroup Detection	File existence checks with `/sys/fs/cgroup/unified`	Parse `/proc/mounts` and `/proc/cgroups` for comprehensive controller detection
Resource Limit Setting	Direct file writes to control files	Use libcgroup library for abstraction and validation
Process Management	Simple PID file writes	Integrate with systemd-run for managed process trees
Error Handling	Errno-based error checking	Structured error types with retry policies

Recommended File Structure

```
container-basic/
  src/
    cgroups/
      cgroups_common.c       ← shared utilities and detection
      cgroups_v1.c           ← cgroups v1 specific implementation
      cgroups_v2.c           ← cgroups v2 specific implementation
      cgroups.h              ← public interface header
      cgroups_internal.h     ← internal structures and helpers
    container.c              ← main container implementation
    container.h              ← container public interface
  tests/
    test_cgroups.c           ← cgroup functionality tests
    test_integration.c       ← end-to-end resource limit tests
```

Infrastructure Starter Code

This code provides complete, working utilities for cgroups management that handle the complexity of version detection and file system operations:

```c
// cgroups_common.c - Complete utility functions

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>

#include <sys/stat.h>

#include <sys/mount.h>

#include "cgroups_internal.h"


// Complete implementation of cgroup version detection

int detect_cgroup_version(void) {

    struct stat st;



    // Check for cgroups v2 unified hierarchy

    if (stat(CGROUPS_V2_PATH, &st) == 0 && S_ISDIR(st.st_mode)) {

        // Verify it's actually mounted as cgroup2

        FILE* mounts = fopen("/proc/mounts", "r");

        if (!mounts) return -1;



        char line[256];

        while (fgets(line, sizeof(line), mounts)) {

            if (strstr(line, "cgroup2") && strstr(line, "/sys/fs/cgroup")) {

                fclose(mounts);

                return 2;  // cgroups v2

            }

        }

        fclose(mounts);
```

```
    }

    // Check for cgroups v1
    if (stat(CGROUPS_V1_PATH, &st) == 0 && S_ISDIR(st.st_mode)) {
        return 1;  // cgroups v1
    }

    return -1;  // No cgroups found
}

// Complete implementation of cgroup path construction
char* get_cgroup_path(const char* container_id, const char* controller) {
    static int cgroup_version = 0;
    if (cgroup_version == 0) {
        cgroup_version = detect_cgroup_version();
    }

    char* path = malloc(256);
    if (!path) return NULL;

    if (cgroup_version == 2) {
        snprintf(path, 256, "%s/%s-%s", CGROUPS_V2_PATH,
                 CONTAINER_CGROUP_PREFIX, container_id);
    } else if (cgroup_version == 1) {
        snprintf(path, 256, "%s/%s/%s-%s", CGROUPS_V1_PATH,
                 controller, CONTAINER_CGROUP_PREFIX, container_id);
    } else {
        free(path);
```

```
        return NULL;
    }

    return path;
}

// Complete implementation of controller availability checking
int verify_controller_available(const char* controller) {
    int version = detect_cgroup_version();

    if (version == 2) {
        // For v2, check controllers file
        FILE* controllers = fopen("/sys/fs/cgroup/cgroup.controllers", "r");
        if (!controllers) return 0;

        char line[256];
        if (fgets(line, sizeof(line), controllers)) {
            int available = strstr(line, controller) != NULL;
            fclose(controllers);
            return available;
        }
        fclose(controllers);
        return 0;
    } else if (version == 1) {
        // For v1, check if controller directory exists
        char path[256];
        snprintf(path, sizeof(path), "%s/%s", CGROUPS_V1_PATH, controller);
        struct stat st;
```

```
        return stat(path, &st) == 0 && S_ISDIR(st.st_mode);
    }

    return 0;
}

// Complete implementation of safe file writing for cgroup control files
int write_cgroup_file(const char* path, const char* value) {
    FILE* file = fopen(path, "w");
    if (!file) {
        return -1;
    }

    int result = fprintf(file, "%s", value);
    if (result < 0) {
        fclose(file);
        return -1;
    }

    if (fclose(file) != 0) {
        return -1;
    }

    return 0;
}
```

Core Logic Skeleton Code

The main cgroups interface functions that learners should implement, with detailed TODO comments mapping to the algorithm steps:

```c
// cgroups.c - Core implementation to be completed by learner

#include "cgroups.h"

#include "cgroups_internal.h"


// Create cgroup hierarchy and configure resource limits for container

int create_container_cgroup(container_config_t* config, container_instance_t* container) {

    // TODO 1: Detect cgroup version using detect_cgroup_version()

    // TODO 2: Generate unique container ID if not provided in config

    // TODO 3: Verify required controllers are available (memory, cpu, pids)

    //         Use verify_controller_available() for each controller

    // TODO 4: Create cgroup directory hierarchy using mkdir() with appropriate permissions

    // TODO 5: Store cgroup path in container->cgroup_path for cleanup tracking

    // TODO 6: Register cleanup function using register_cleanup() for failure recovery

    // TODO 7: Configure memory limits by calling set_memory_limit()

    // TODO 8: Configure CPU limits by calling set_cpu_limit()

    // TODO 9: Configure process limits by calling set_process_limit()

    // TODO 10: Return 0 on success, -1 on failure with errno set



    return -1; // Replace with implementation

}


// Assign process to cgroup for resource limit enforcement

int assign_process_to_cgroup(const char* cgroup_path, pid_t pid) {

    // TODO 1: Determine cgroup version to know which tasks file to use

    // TODO 2: Construct path to tasks file (cgroup.procs for v2, tasks for v1)

    // TODO 3: Convert PID to string for writing to control file

    // TODO 4: Write PID to appropriate tasks/procs file using write_cgroup_file()

    // TODO 5: Verify write succeeded by reading back the tasks file

    // TODO 6: Return 0 on success, -1 on failure
```

```
    // Hint: v2 uses "cgroup.procs", v1 uses "tasks"


    return -1; // Replace with implementation
}

// Configure memory limit and OOM behavior for cgroup
int set_memory_limit(const char* cgroup_path, size_t limit_bytes) {
    // TODO 1: Detect cgroup version to determine control file names
    // TODO 2: Convert limit_bytes to string representation
    // TODO 3: Write limit to memory.max (v2) or memory.limit_in_bytes (v1)
    // TODO 4: Configure OOM behavior - set memory.oom.group (v2) or memory.use_hierarchy (v1)
    // TODO 5: Optionally set memory.swappiness to control swap usage
    // TODO 6: Verify limits were applied by reading back control files
    // TODO 7: Return 0 on success, -1 on failure
    // Hint: Use write_cgroup_file() helper for atomic file writes


    return -1; // Replace with implementation
}

// Configure CPU quota and period to limit CPU usage percentage
int set_cpu_limit(const char* cgroup_path, int cpu_percent) {
    // TODO 1: Validate cpu_percent is reasonable (1-800 for multi-core)
    // TODO 2: Calculate quota and period values (typically 100ms period)
    //         quota = (cpu_percent / 100.0) * period
    // TODO 3: Detect cgroup version for control file format
    // TODO 4: For v2: write "quota period" to cpu.max
    //         For v1: write quota to cpu.cfs_quota_us, period to cpu.cfs_period_us
    // TODO 5: Verify CPU limits by reading back control files
    // TODO 6: Return 0 on success, -1 on failure
```

```
    // Hint: Standard period is 100000 microseconds (100ms)


    return -1; // Replace with implementation
}

// Configure maximum number of processes in cgroup
int set_process_limit(const char* cgroup_path, int max_processes) {
    // TODO 1: Validate max_processes is positive and reasonable
    // TODO 2: Convert max_processes to string
    // TODO 3: Write to pids.max control file (same name in v1 and v2)
    // TODO 4: Verify limit was set by reading back pids.max
    // TODO 5: Return 0 on success, -1 on failure
    // Hint: PIDs controller has same interface in v1 and v2


    return -1; // Replace with implementation
}

// Clean up cgroup hierarchy and kernel resources
int cleanup_container_cgroup(const char* cgroup_path) {
    // TODO 1: Terminate any remaining processes in cgroup
    //         Read pids from cgroup.procs/tasks file
    // TODO 2: Send SIGTERM to all processes, wait briefly
    // TODO 3: Send SIGKILL to any remaining processes
    // TODO 4: Wait for all processes to exit (poll cgroup.procs)
    // TODO 5: Remove cgroup directory using rmdir()
    // TODO 6: Handle EBUSY errors with retry logic (up to 3 attempts)
    // TODO 7: Return 0 on success, -1 if cleanup failed
    // Hint: Use kill(-pid, signal) to signal process groups
```

```
    return -1; // Replace with implementation
}
```

Language-Specific Hints

File I/O for Cgroup Control Files:

- Use `fopen()` with "w" mode for writing control files
- Always check `fclose()` return value - it can fail and lose data
- Some control files require newlines, others don't - check kernel documentation

Process Management:

- Use `kill()` with negative PID to signal entire process groups
- `SIGTERM` allows graceful shutdown, `SIGKILL` forces immediate termination
- Check `/proc/[pid]/status` to verify process termination

Error Handling Patterns:

- Cgroup operations can fail with `ENOENT` (path doesn't exist), `EPERM` (insufficient privileges), or `EINVAL` (invalid values)
- Always check `errno` after failed operations for specific error diagnosis
- Use `perror()` or `strerror(errno)` for human-readable error messages

Memory Management:

- Use `malloc()` for dynamic path construction, always check for NULL return
- Free allocated paths with `free()` to prevent memory leaks
- Consider using `alloca()` for small, temporary string buffers

Milestone Checkpoint

After implementing the cgroups component, verify correct behavior with these concrete tests:

Test 1: Memory Limit Enforcement

```bash
# Create container with 10MB memory limit
./container --memory-limit=10485760 stress --vm 1 --vm-bytes 20M

# Expected: Container should be OOM killed when trying to allocate 20MB
# Check: dmesg should show "Memory cgroup out of memory" message
# Verify: echo $? should return non-zero exit code
```

Test 2: CPU Limit Verification

```bash
# Create container with 25% CPU limit
./container --cpu-percent=25 dd if=/dev/zero of=/dev/null

# Expected: Container should use ~25% CPU even under full load
# Check: Use 'top' or 'htop' to verify CPU usage stays around 25%
# Verify: CPU throttling stats in /sys/fs/cgroup/.../cpu.stat
```

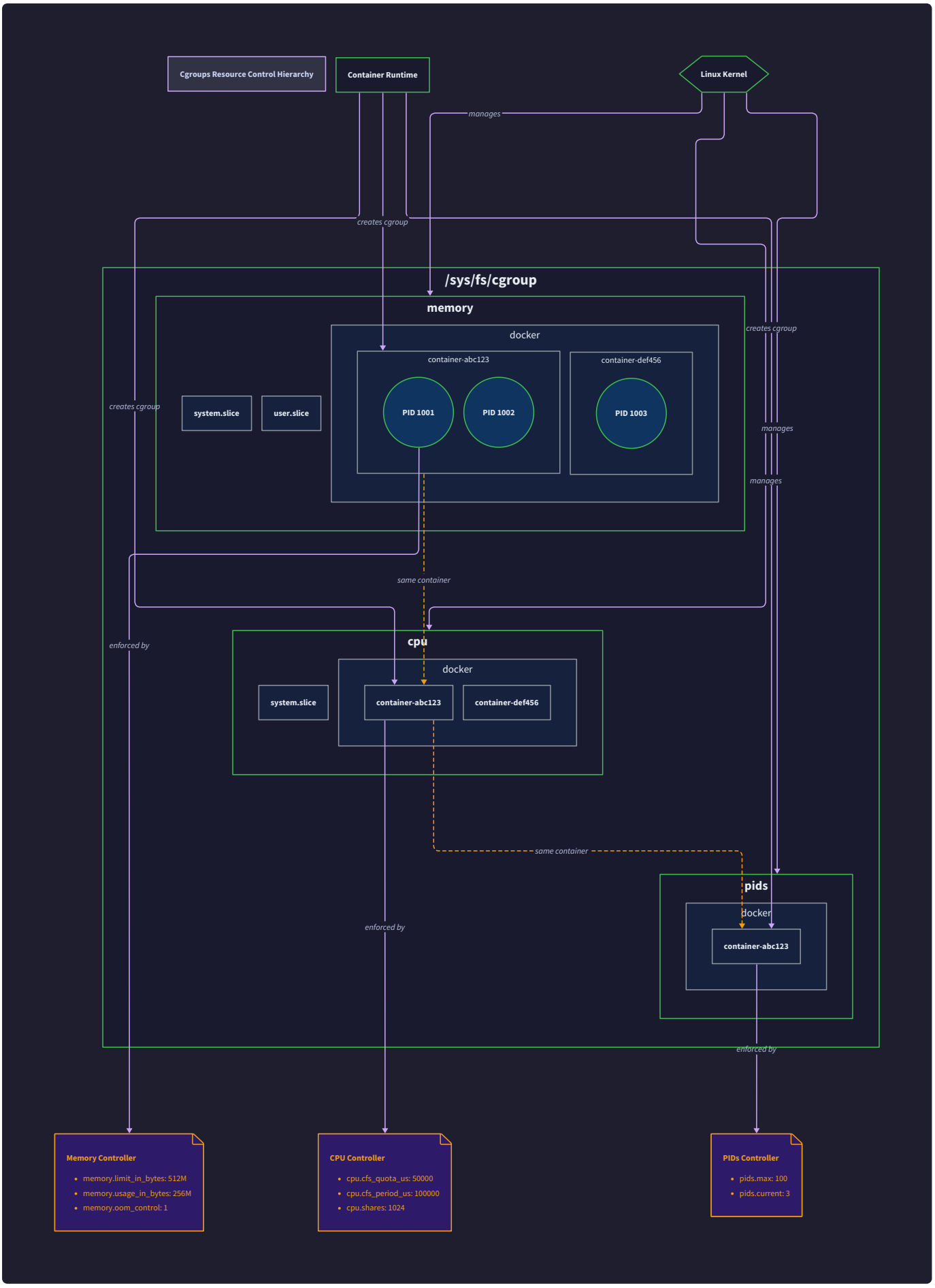
Test 3: Process Limit Testing

```bash
# Create container with 5 process limit
./container --max-processes=5 bash -c 'for i in {1..10}; do sleep 30 & done'

# Expected: Only first 5 background processes should start
# Check: ps aux should show exactly 5 sleep processes
# Verify: Attempts to create 6th process should fail with EAGAIN
```

Signs of Problems:

- **Memory limits ignored**: Container allocates more than limit without OOM → Check controller availability and cgroup v1/v2 detection
- **CPU limits not enforced**: Container uses 100% CPU despite limit → Verify quota/period calculation and control file writes
- **Process limits bypassed**: More processes created than limit → Check PIDs controller mounting and process assignment
- **Cleanup failures**: Orphaned cgroup directories in `/sys/fs/cgroup/` → Implement proper process termination before cgroup removal

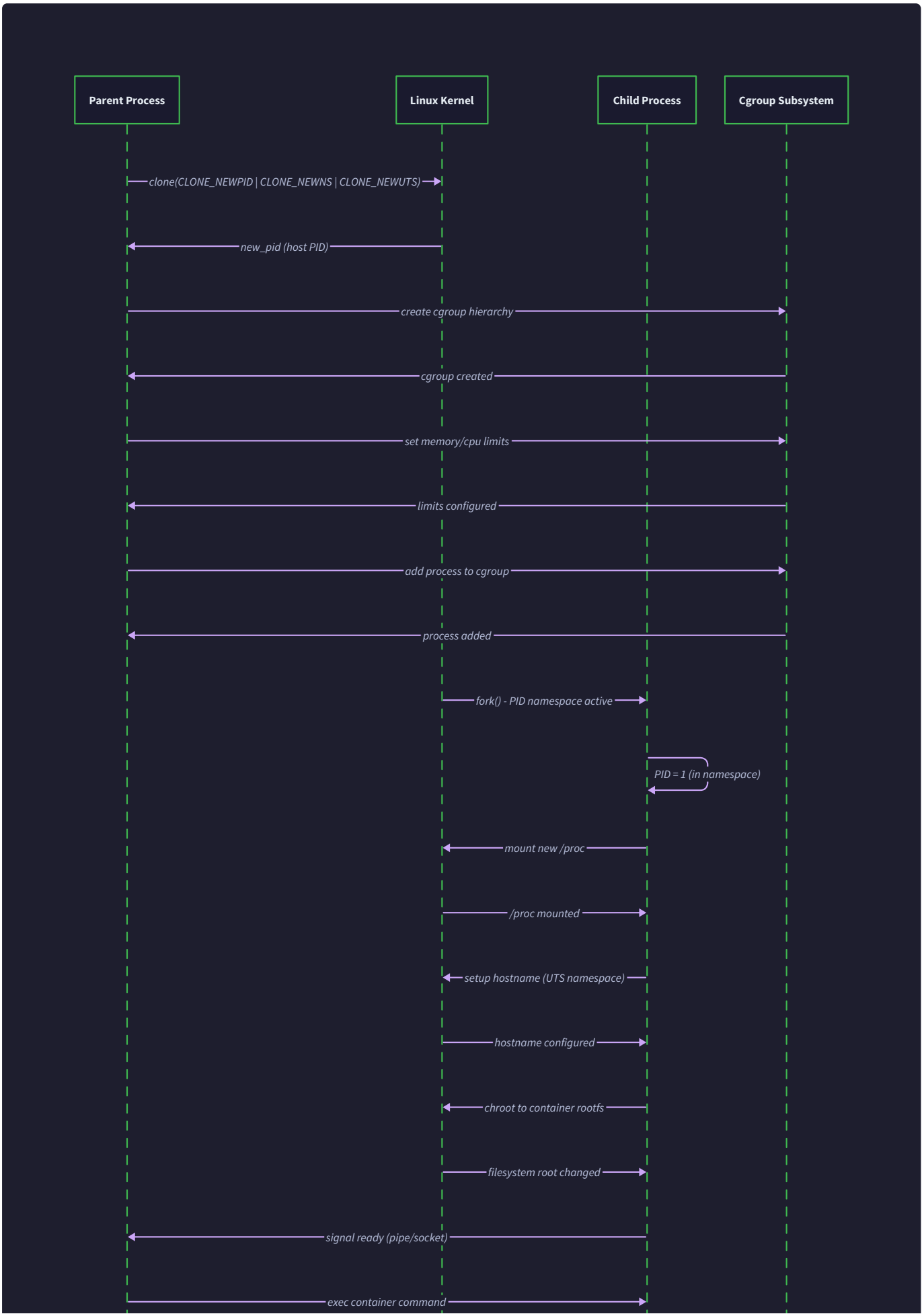

Cgroups Resource Control Hierarchy
Container Runtime
Linux Kernel
manages
creates cgroup
/sys/fs/cgroup
memory
docker
container-abc123
container-def456
PID 1001
PID 1002
PID 1003
system.slice
user.slice
creates cgroup
creates cgroup
manages
manages
same container
enforced by
cpu
docker
system.slice
container-abc123
container-def456
same container
pids
docker
container-abc123
enforced by
enforced by
Memory Controller
memory.limit_in_bytes: 512M
memory.usage_in_bytes: 256M
memory.oom_control: 1
CPU Controller
cpu.cfs_quota_us: 50000
cpu.cfs_period_us: 100000
cpu.shares: 1024
PIDs Controller
pids.max: 100
pids.current: 3

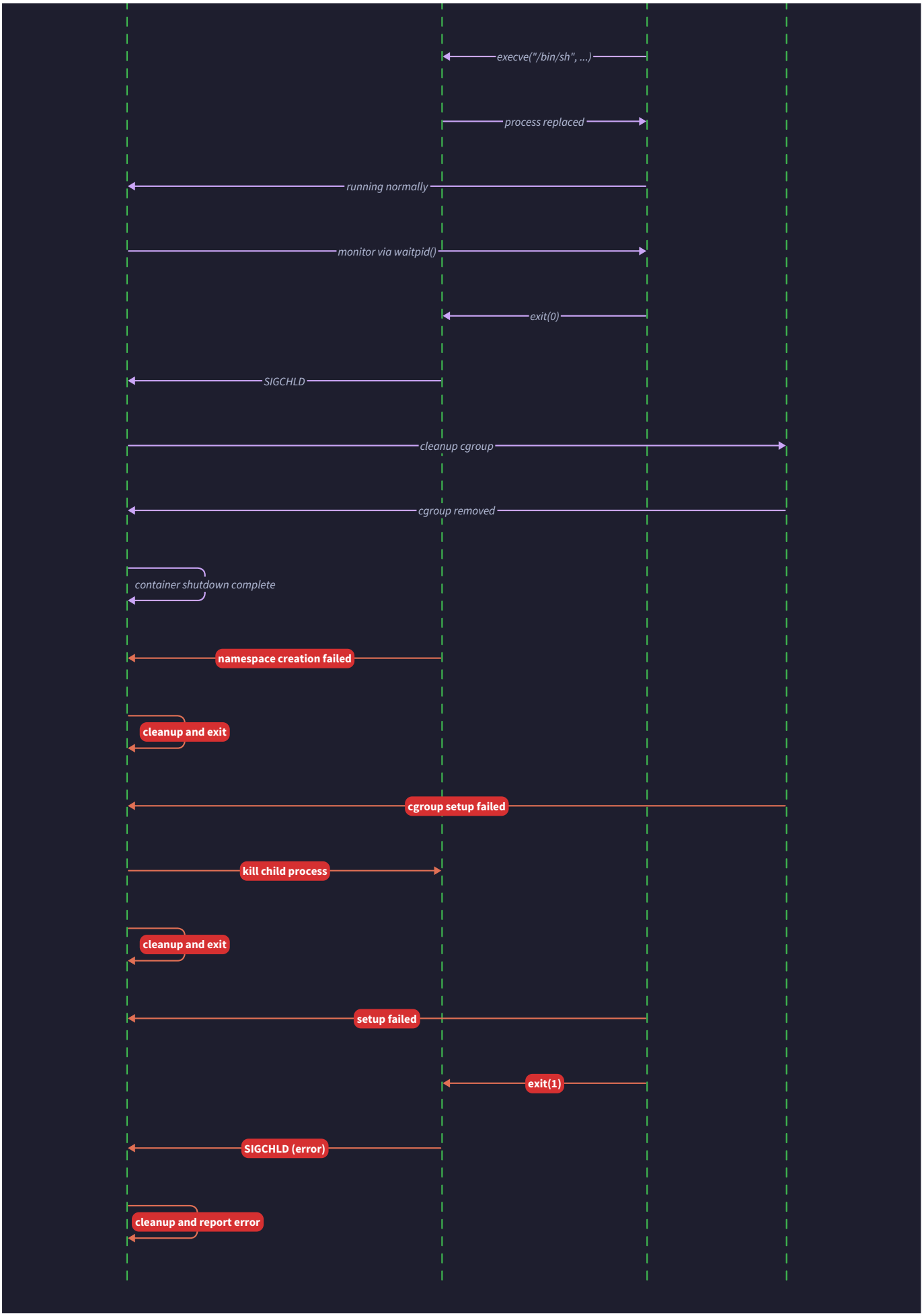
Interactions and Data Flow

> **Milestone(s):** This section spans all milestones (1-4), describing how the PID namespace, mount namespace, network namespace, and cgroups components coordinate during container lifecycle operations.

The container runtime operates as a carefully orchestrated system where multiple Linux kernel isolation mechanisms must be created, configured, and managed in a specific sequence. Think of launching a container like **setting up a new office branch**: you need to establish the building (mount namespace), assign phone numbers (PID namespace), connect telecommunications (network namespace), and allocate budgets (cgroups) - and all of these must happen in the right order, with each step depending on the previous ones completing successfully.

The coordination challenge stems from the interdependencies between namespace types and the fact that many operations must occur from specific contexts (host vs container namespace). Additionally, cleanup requires the reverse order to prevent resource leaks, similar to how you must evacuate employees before disconnecting utilities when closing an office branch.


Parent Process
Linux Kernel
Child Process
Cgroup Subsystem
clone(CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWUTS)
new_pid (host PID)
create cgroup hierarchy
cgroup created
set memory/cpu limits
limits configured
add process to cgroup
process added
fork() - PID namespace active
PID = 1 (in namespace)
mount new /proc
/proc mounted
setup hostname (UTS namespace)
hostname configured
chroot to container rootfs
filesystem root changed
signal ready (pipe/socket)
exec container command


execve("/bin/sh", ...)
process replaced
running normally
monitor via waitpid()
exit(0)
SIGCHLD
cleanup cgroup
cgroup removed
container shutdown complete
namespace creation failed
cleanup and exit
cgroup setup failed
kill child process
cleanup and exit
setup failed
exit(1)
SIGCHLD (error)
cleanup and report error

Container Startup Sequence

Container creation follows a precise multi-stage sequence that coordinates namespace creation, process forking, resource limit application, and error recovery. Each stage has specific responsibilities and must complete before the next stage can begin.

The startup sequence operates through **three distinct execution contexts**: the parent process running in host namespaces, the child process transitioning between namespaces, and the final containerized process running in complete isolation. Understanding which operations occur in which context is crucial for proper implementation.

Stage 1: Preparation and Validation

The startup sequence begins in the host context with configuration validation and resource preparation. This stage verifies that all required kernel features are available and that the container configuration is valid before creating any namespaces.

Operation	Purpose	Validation Checks
Validate rootfs path	Ensure container filesystem exists	Directory exists, readable, contains essential directories
Check kernel features	Verify namespace support available	/proc/self/ns/ contains required namespace files
Detect cgroup version	Determine cgroups v1 vs v2	Check /sys/fs/cgroup mount type and available controllers
Verify privileges	Ensure sufficient permissions	CAP_SYS_ADMIN capability or root privileges
Allocate resources	Reserve network IP and cgroup paths	Generate unique container ID, check IP availability

> **Critical Insight**: All validation occurs before creating any namespaces because cleanup becomes significantly more complex once kernel resources are allocated. A validation failure at this stage requires no cleanup.

Stage 2: Namespace Creation and Process Forking

The core isolation setup occurs through the `safe_clone()` system call with combined namespace flags. This single operation creates the child process and establishes all required namespaces simultaneously, avoiding the timing and synchronization issues that arise from sequential namespace creation.

The parent process executes the following sequence:

1. **Allocate clone stack**: Reserve `STACK_SIZE` bytes for the child process stack, ensuring proper alignment for the target architecture

2. **Prepare container instance**: Initialize `container_instance_t` structure with allocated cleanup list and namespace file descriptors array
3. **Register cleanup functions**: Add cleanup handlers for stack deallocation, namespace closing, and cgroup removal to handle setup failures
4. **Execute clone system call**: Call `safe_clone(container_init_process, stack, CONTAINER_NS_FLAGS, config)` to create isolated child process
5. **Store child PID**: Record the child process ID in the container instance for monitoring and cleanup purposes
6. **Open namespace file descriptors**: Access `/proc/<child_pid>/ns/*` files to maintain references to the created namespaces

The child process begins execution in the new namespaces with a completely different view of system resources. It sees itself as PID 1 in an empty process tree, has access only to the new mount namespace, and possesses an isolated network stack.

Namespace Type	Child Process View	Parent Process View
PID	Process sees itself as PID 1, no other processes visible	Child appears as normal PID in host process tree
Mount	Inherits host mount tree initially, will be modified	Host mount tree unchanged
Network	Empty network namespace with only loopback interface	Host network interfaces remain accessible
UTS	Separate hostname that can be changed independently	Host hostname unchanged
IPC	Empty IPC namespace with no shared memory segments	Host IPC resources remain accessible

> **Design Principle**: Using `clone()` with combined flags creates all namespaces atomically, eliminating race conditions that occur when creating namespaces sequentially with `unshare()`.

Stage 3: Cgroups Resource Limit Setup

Resource limits must be established immediately after process creation but before the container process begins executing user code. This timing ensures that resource consumption is controlled from the very beginning of container execution.

The parent process handles cgroup setup because it retains the necessary privileges and host filesystem access:

1. **Create cgroup hierarchy**: Call `create_container_cgroup(config, container)` to establish the cgroup directory structure in the appropriate controller hierarchies
2. **Assign child process**: Execute `assign_process_to_cgroup(container->cgroup_path, container->child_pid)` to place the container process under resource control

3. **Configure memory limits**: Apply memory restrictions using `set_memory_limit(container->cgroup_path, config->memory_limit_bytes)` before significant memory allocation occurs
4. **Configure CPU limits**: Establish CPU usage restrictions through `set_cpu_limit(container->cgroup_path, config->cpu_percent)` to prevent CPU monopolization
5. **Configure process limits**: Set maximum process count via `set_process_limit(container->cgroup_path, config->max_processes)` to prevent fork bomb attacks
6. **Verify limit enforcement**: Read back cgroup settings using `read_cgroup_stat()` to confirm that limits were applied correctly

The cgroup assignment affects the child process immediately, even though the child is simultaneously setting up its namespace environment. This parallel execution is safe because cgroups operate at the kernel level and do not interfere with namespace operations.

Stage 4: Container Environment Initialization

The child process, now running as PID 1 in isolated namespaces with resource limits applied, must establish the container's execution environment. This stage transforms the raw namespace environment into a functional container.

The `container_init_process()` function executes the following initialization sequence:

1. **Configure signal handling**: Call `setup_init_signals()` to establish proper signal handlers for zombie process reaping, since the container process inherits PID 1 responsibilities
2. **Create mount namespace isolation**: Execute `create_mount_namespace(config, container)` to establish private mount propagation and prepare for filesystem modifications
3. **Setup container root filesystem**: Run `setup_container_rootfs(config->rootfs_path, "/mnt/old_root")` to prepare the new root directory structure
4. **Perform pivot root operation**: Execute `pivot_to_container_root(config->rootfs_path, "/mnt/old_root")` to switch the filesystem root to the container filesystem
5. **Mount essential filesystems**: Call `mount_essential_filesystems()` to provide `/proc`, `/sys`, and `/dev` filesystems required by containerized applications
6. **Configure network interface**: Execute network configuration commands to set up the container end of the veth pair with the assigned IP address
7. **Set hostname**: Update the container hostname using `sethostname()` to provide identity isolation
8. **Drop privileges**: If configured, drop unnecessary capabilities and change to a non-root user for security

> **Synchronization Point**: The parent process waits for the child to complete mount namespace setup before proceeding with network configuration, since network setup requires knowledge of the child's network namespace.

Stage 5: Network Connectivity Establishment

Network connectivity setup requires coordination between parent and child processes because veth pair creation occurs in the host namespace while interface configuration occurs in the container namespace.

The parent process handles the host-side networking:

1. **Create veth pair**: Call `create_veth_pair(host_if_name, container_if_name)` to create the virtual ethernet link with unique interface names
2. **Assign container interface**: Execute `assign_veth_to_namespace(container_if_name, container->child_pid)` to move the container end into the child's network namespace
3. **Attach to host bridge**: Run `attach_to_bridge(host_if_name, config->bridge_name)` to connect the host end to the bridge for inter-container communication
4. **Configure host interface**: Bring up the host end of the veth pair and configure any required routing rules for container connectivity

The child process, running in its network namespace, configures the container-side networking:

1. **Configure container interface**: Execute `configure_container_interface(container_if_name, config->container_ip, netmask)` to assign the IP address and activate the interface
2. **Setup default route**: Call `setup_default_route(gateway_ip)` to establish connectivity to external networks through the host bridge
3. **Verify connectivity**: Test basic network functionality by attempting to reach the gateway and DNS resolution

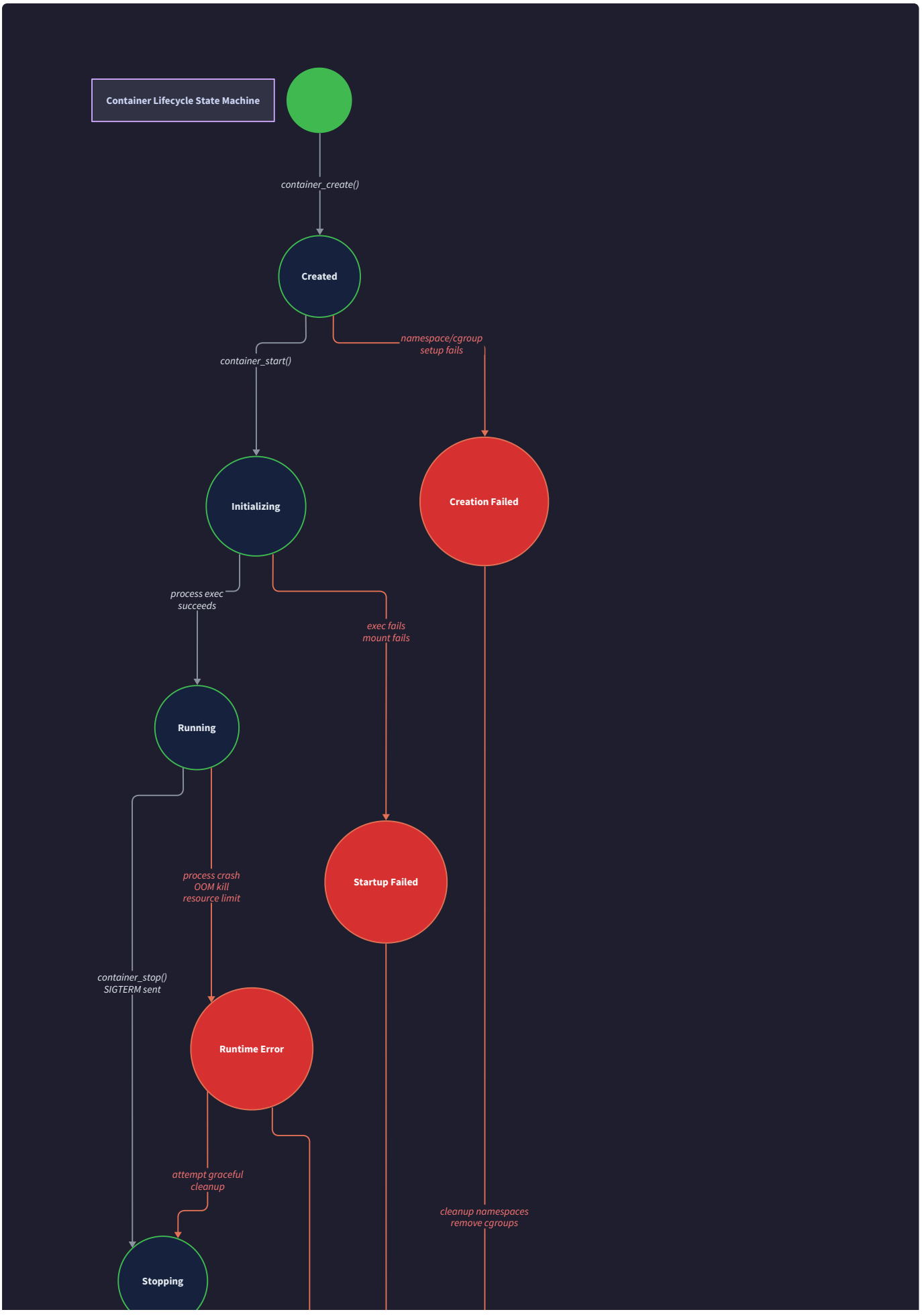
Stage 6: Application Execution

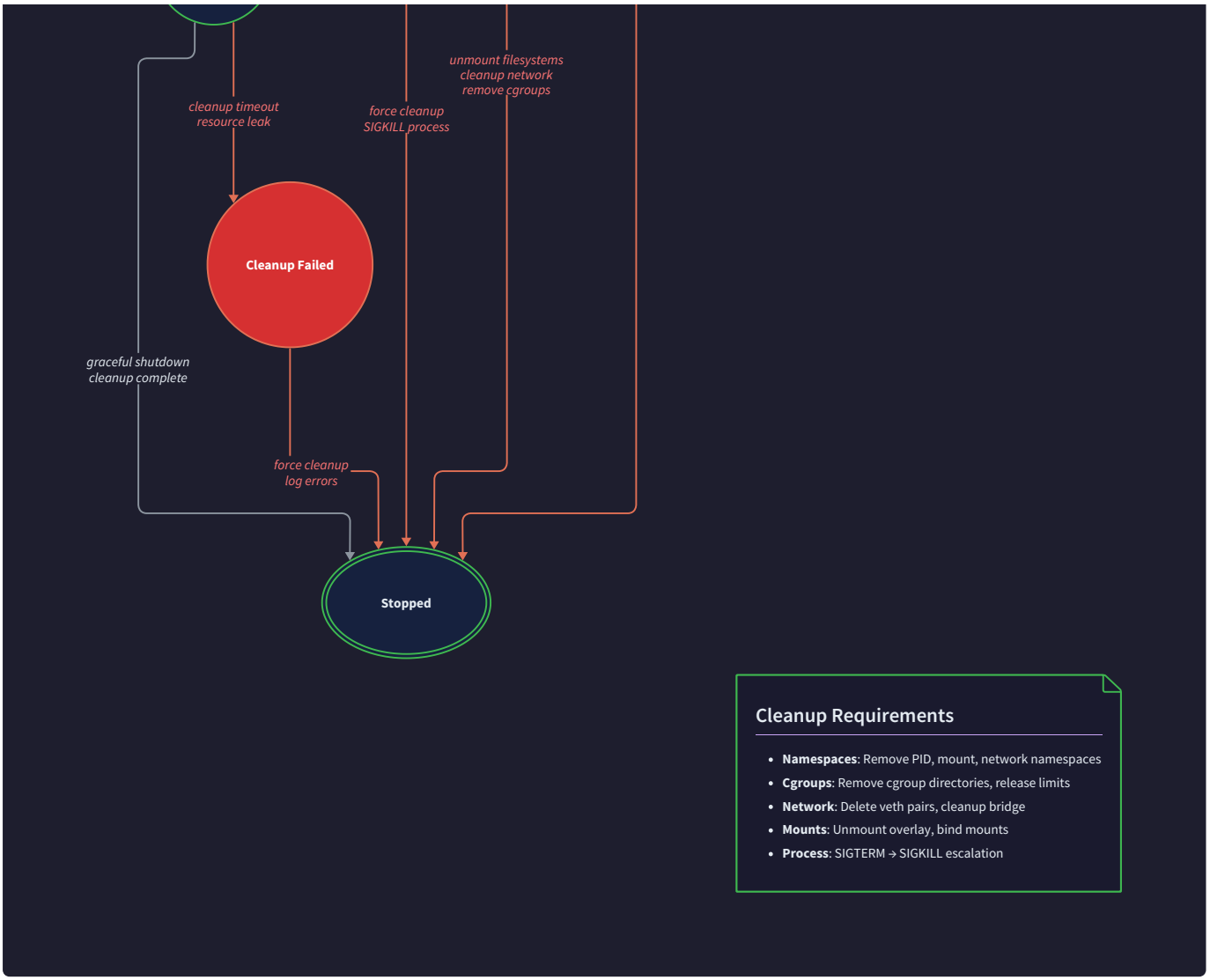
Once all namespace and resource setup completes, the container process transitions to executing the user-specified application. This transition represents the completion of container initialization and the beginning of application runtime.

The final steps in `container_init_process()` prepare for application execution:

1. **Validate executable**: Verify that `config->argv[0]` exists and is executable within the container filesystem
2. **Prepare environment**: Set up environment variables from `config->envp` and add container-specific variables
3. **Change working directory**: Set the working directory to an appropriate location within the container filesystem
4. **Execute application**: Call `execve(config->argv[0], config->argv, config->envp)` to replace the init process with the container application

> **Process Identity Change**: The `execve()` call replaces the container init process with the application, but the process retains PID 1 and all namespace memberships and resource limits.


Container Lifecycle State Machine
container_create()
Created
namespace/cgroup setup fails
container_start()
Initializing
Creation Failed
process exec succeeds
exec fails mount fails
Running
Startup Failed
process crash OOM kill resource limit
container_stop() SIGTERM sent
Runtime Error
attempt graceful cleanup
cleanup namespaces remove cgroups
Stopping



Container Cleanup Sequence

Container cleanup must occur in the reverse order of creation to prevent resource leaks and ensure proper kernel resource deallocation. The cleanup sequence handles both normal termination and error recovery scenarios.

The cleanup challenge stems from the fact that **namespaces and cgroups form dependency relationships**: processes must exit before their cgroups can be removed, namespace file descriptors must be closed before namespace resources are freed, and mount points must be unmounted before mount namespaces can be destroyed.

Cleanup Trigger Detection

Container cleanup begins when the parent process detects that the child process has exited. The `container_wait(container)` function monitors the child process and initiates cleanup when termination occurs.

Termination Cause	Detection Method	Cleanup Requirements
Normal application exit	waitpid() returns with WIFEXITED	Standard cleanup sequence
Application killed by signal	waitpid() returns with WIFSIGNALED	Standard cleanup sequence
OOM killer termination	waitpid() returns, check cgroup memory events	Memory limit exceeded, standard cleanup
Container timeout	waitpid() with timeout expires	Send SIGTERM, wait, send SIGKILL, then cleanup
Setup error	clone() fails or child setup fails	Partial cleanup based on cleanup list

Stage 1: Process Termination and Signal Handling

The cleanup sequence begins with ensuring that all processes within the container have terminated. This step is crucial because active processes prevent namespace destruction and can cause resource leaks.

1. **Detect child exit**: The `container_wait()` function uses `waitpid(container->child_pid, &status, 0)` to detect when the main container process exits
2. **Collect exit status**: Extract the exit code or termination signal from the wait status for reporting to the container user
3. **Handle zombie reaping**: If the container process spawned additional children, ensure all zombies are collected to prevent process table pollution
4. **Force termination if needed**: For timeout scenarios, send `SIGTERM` followed by `SIGKILL` to ensure process termination

> **Zombie Prevention**: Since the container process runs as PID 1, it must reap its own children. If the application fails to do this, zombie processes remain until namespace destruction.

Stage 2: Network Namespace Cleanup

Network resources require early cleanup because they involve kernel objects that reference the network namespace. Delaying network cleanup can cause memory leaks in the kernel networking subsystem.

The `cleanup_network_namespace(container)` function performs network teardown:

1. **Remove veth interfaces**: The kernel automatically destroys both ends of the veth pair when the container network namespace is destroyed, but explicit cleanup ensures immediate resource recovery

2. **Close namespace file descriptors**: Close the network namespace file descriptor stored in `container->namespace_fds[CLONE_NEWNET]` to release the namespace reference
3. **Update bridge state**: Remove any bridge forwarding entries that referenced the container's MAC address to prevent stale entries
4. **Release IP address**: Return the allocated IP address to the available pool for reuse by future containers

Stage 3: Mount Namespace Cleanup

Mount namespace cleanup involves unmounting filesystems and releasing filesystem references. This stage must handle both successful containers (where pivot_root occurred) and failed containers (where mount setup was incomplete).

The mount cleanup process follows this sequence:

1. **Detect mount state**: Check whether pivot_root completed successfully by examining the container's current root directory
2. **Unmount essential filesystems**: Remove `/proc`, `/sys`, and `/dev` mounts that were created within the container namespace
3. **Handle old root cleanup**: If pivot_root succeeded, unmount the old root directory that was bind-mounted during the pivot operation
4. **Unmount bind mounts**: Remove any host directory bind mounts that were established for the container
5. **Close mount namespace**: Close the mount namespace file descriptor to release the namespace reference

> **Mount Cleanup Challenge**: Mount points must be unmounted in reverse dependency order. Child mounts must be unmounted before parent mounts, similar to how directory trees must be deleted from leaves to root.

Stage 4: PID Namespace Cleanup

PID namespace cleanup is largely automatic once all processes within the namespace have exited. However, explicit cleanup steps ensure proper resource recovery and handle edge cases.

1. **Verify process termination**: Confirm that all processes within the PID namespace have exited by checking that no `/proc/<pid>/ns/pid` files reference the container's PID namespace
2. **Close PID namespace file descriptor**: Release the PID namespace reference held by the parent process
3. **Clean up process tracking**: Remove any process monitoring or logging that was specific to the container's PID namespace

Stage 5: Cgroup Hierarchy Removal

Cgroup cleanup must occur after all processes have exited because cgroups cannot be removed while they contain active processes. The `cleanup_container_cgroup(cgroup_path)` function handles hierarchical cleanup.

The cgroup removal process follows strict ordering requirements:

1. **Verify process exit**: Confirm that the cgroup's `tasks` file is empty, indicating all processes have exited

2. **Read final statistics**: Collect final resource usage statistics before removing the cgroup for reporting or debugging
3. **Remove child cgroups**: If any sub-cgroups were created, remove them in reverse creation order
4. **Remove parent cgroup**: Delete the main container cgroup directory, which automatically releases all associated kernel resources
5. **Verify controller cleanup**: Ensure that all cgroup controllers have properly released their resources

> **Hierarchical Dependency**: Cgroups form a hierarchy where child cgroups must be removed before their parents. The kernel enforces this ordering and will reject attempts to remove parent cgroups while children exist.

Stage 6: Resource Cleanup and Finalization

The final cleanup stage releases user-space resources and updates any external state that tracked the container's existence.

1. **Execute registered cleanup functions**: Process the `cleanup_list_t` chain to execute all registered cleanup functions in reverse order
2. **Release memory allocations**: Free the container instance structure, configuration copies, and any allocated string buffers
3. **Close remaining file descriptors**: Ensure all namespace file descriptors and other file handles are closed
4. **Update container registry**: Remove the container from any runtime tracking structures or persistent state
5. **Log cleanup completion**: Record successful cleanup for debugging and auditing purposes

Error Propagation

Error handling in container creation requires careful coordination because failures can occur in any component while other components are simultaneously executing setup operations. The error propagation system must ensure that partial setup is properly cleaned up regardless of where failure occurs.

Error Categories and Impact Scope

Container creation errors fall into distinct categories based on their scope and recovery requirements. Understanding these categories guides the error propagation strategy.

Error Category	Examples	Impact Scope	Recovery Strategy
Validation Errors	Invalid rootfs path, missing capabilities	Pre-setup	Simple return, no cleanup needed
Resource Allocation	Out of memory, PID exhaustion	Single component	Component-specific cleanup
Kernel Feature Errors	Namespace not supported, cgroup unavailable	System-wide	Graceful degradation or abort
Race Conditions	Network interface conflicts, cgroup timing	Cross-component	Retry with backoff
Permission Errors	Insufficient privileges, filesystem access	Context-dependent	Privilege escalation or abort

> **Error Propagation Principle**: Errors must be propagated to the point where complete cleanup can be performed. This often means that child process errors must be communicated to the parent process that holds cleanup responsibilities.

Cleanup List Mechanism

The `cleanup_list_t` structure provides a systematic approach to error recovery by maintaining a stack of cleanup operations that corresponds to successful setup operations. Each setup operation registers its corresponding cleanup function before proceeding.

```
Cleanup Registration Pattern:
1. Attempt setup operation
2. If successful, register corresponding cleanup function
3. If failure, execute all registered cleanup functions
4. Propagate error to caller
```

The cleanup list operates as a **stack (LIFO) structure** to ensure that cleanup occurs in reverse setup order:

Setup Operation	Cleanup Registration	Cleanup Function
clone() process creation	`register_cleanup(list, kill_child_process, &child_pid)`	Send SIGKILL and waitpid()
Create cgroup directory	`register_cleanup(list, remove_cgroup, cgroup_path)`	rmdir() cgroup directory
Create veth pair	`register_cleanup(list, destroy_veth_pair, if_name)`	Delete network interfaces
Open namespace FDs	`register_cleanup(list, close_namespace_fds, fd_array)`	Close all file descriptors
Mount filesystems	`register_cleanup(list, unmount_filesystems, mount_list)`	Unmount in reverse order

Parent-Child Error Communication

Error propagation between parent and child processes requires special handling because the processes operate in different contexts and may not share memory after namespace creation.

The communication mechanism uses **process exit codes** and **signal patterns** to convey error information:

1. **Success Path**: Child process calls `execve()` to run the container application, never returns to init code
2. **Setup Error Path**: Child process exits with specific error codes that indicate the failure point
3. **Runtime Error Path**: Child process is killed by signal, indicating runtime failure (OOM, segmentation fault, etc.)

Child Exit Status	Error Interpretation	Parent Action
execve() success (no exit)	Container running normally	Monitor process, wait for termination
Exit code 1-99	Setup error in specific component	Execute cleanup, report setup failure
Exit code 100-199	Resource limit exceeded	Execute cleanup, report resource exhaustion
SIGKILL with OOM flag	Memory limit enforcement	Execute cleanup, report OOM condition
SIGSEGV or SIGABRT	Application crash	Execute cleanup, report application error

> **Communication Limitation**: Once the child process has entered its namespaces, it cannot directly communicate complex error information to the parent. Exit codes provide the only reliable error propagation mechanism.

Cross-Component Error Handling

When multiple components are setting up simultaneously (such as cgroups in parent and mount namespace in child), error propagation must coordinate across component boundaries.

Network Setup Error Example: If veth pair creation fails in the parent process while the child process is simultaneously setting up mount namespaces:

1. **Parent detects veth creation failure**: The `create_veth_pair()` function returns an error code
2. **Parent signals child termination**: Send `SIGTERM` to child process to abort its setup
3. **Child detects termination signal**: Mount setup code checks for pending signals and aborts gracefully
4. **Parent waits for child exit**: Use `waitpid()` to confirm child has terminated
5. **Parent executes cleanup**: Run all registered cleanup functions to release allocated resources
6. **Parent propagates error**: Return error code to `container_create()` caller with specific failure reason

Mount Setup Error Example: If pivot_root fails in the child process while the parent has already created cgroups and network interfaces:

1. **Child detects pivot_root failure**: The filesystem operation returns an error
2. **Child exits with specific code**: Exit with code indicating mount setup failure
3. **Parent detects child exit**: `waitpid()` returns with child exit status
4. **Parent interprets error**: Exit code indicates which component failed
5. **Parent executes full cleanup**: Clean up cgroups, network interfaces, and other resources
6. **Parent reports specific error**: Return detailed error information about mount failure

Error Recovery Strategies

Different error scenarios require different recovery approaches based on whether the error is transient, permanent, or indicates system-level issues.

> **Decision: Fail-Fast vs. Graceful Degradation**
>
> - **Context**: Some container features (like specific cgroup controllers) may be unavailable but container creation could proceed with reduced functionality
> - **Options Considered**: Always fail fast, always gracefully degrade, configurable behavior
> - **Decision**: Fail fast for essential features (PID, mount namespaces), graceful degradation for optional features (specific cgroup controllers)
> - **Rationale**: Essential features are required for basic isolation security. Optional features can be disabled while maintaining core container functionality.
> - **Consequences**: Clear security guarantees but reduced flexibility in heterogeneous environments

Recovery Strategy	When to Apply	Implementation Approach
Immediate Retry	Transient resource conflicts	Retry with exponential backoff up to 3 attempts
Graceful Degradation	Optional feature unavailability	Continue with feature disabled, log warning
Fail Fast	Essential feature missing	Immediate cleanup and error propagation
Resource Cleanup and Retry	Resource exhaustion	Clean up failed attempt, wait, retry once
System Error Reporting	Kernel or hardware issues	Clean up, report system-level error, do not retry

Implementation Guidance

The container lifecycle coordination requires careful orchestration of system calls, process synchronization, and resource management across multiple kernel subsystems.

A. Technology Recommendations

Component	Simple Option	Advanced Option
Process Creation	fork() + namespace setup	clone() with combined flags
Error Communication	Exit codes only	Shared memory + exit codes
Cleanup Coordination	Function pointer list	State machine with rollback
Synchronization	waitpid() polling	eventfd + signaling
Resource Tracking	Manual cleanup functions	RAII-style resource guards

B. Recommended File Structure

```
container-basic/
  src/
    container.c              ← main container lifecycle coordination
    container.h              ← public container API and structures
    container_internal.h     ← internal coordination structures
    namespace_manager.c      ← namespace creation coordination
    cleanup.c                ← cleanup list and error recovery
    sync.c                   ← parent-child synchronization
  tests/
    test_lifecycle.c         ← container creation and cleanup tests
    test_error_handling.c    ← error propagation and recovery tests
    test_coordination.c      ← component interaction tests
```

C. Infrastructure Starter Code

Complete cleanup list management for systematic error recovery:

```c
// cleanup.h

typedef struct cleanup_list {

    void (*cleanup_func)(void *data);

    void *data;

    struct cleanup_list *next;

} cleanup_list_t;


// cleanup.c

#include <stdlib.h>

#include <stdio.h>

#include "cleanup.h"


void register_cleanup(cleanup_list_t **list, void (*func)(void*), void *data) {

    cleanup_list_t *entry = malloc(sizeof(cleanup_list_t));

    if (!entry) {

        fprintf(stderr, "Failed to allocate cleanup entry\n");

        return;

    }

    entry->cleanup_func = func;

    entry->data = data;

    entry->next = *list;

    *list = entry;

}


void execute_cleanup(cleanup_list_t **list) {

    cleanup_list_t *current = *list;

    while (current) {

        if (current->cleanup_func) {

            current->cleanup_func(current->data);
```

```
        }
        cleanup_list_t *next = current->next;
        free(current);
        current = next;
    }
    *list = NULL;
}

// Common cleanup functions
void cleanup_close_fd(void *data) {
    int fd = *(int*)data;
    if (fd >= 0) {
        close(fd);
    }
}

void cleanup_kill_process(void *data) {
    pid_t pid = *(pid_t*)data;
    if (pid > 0) {
        kill(pid, SIGTERM);
        usleep(100000); // 100ms grace period
        kill(pid, SIGKILL);
        waitpid(pid, NULL, 0);
    }
}

void cleanup_remove_cgroup(void *data) {
    char *path = (char*)data;
    if (path && strlen(path) > 0) {
```

```
        rmdir(path);
    }
}
```

Parent-child synchronization utilities:

```c
// sync.h
typedef struct container_sync {
    int parent_pipe[2];  // Parent writes, child reads
    int child_pipe[2];   // Child writes, parent reads
} container_sync_t;

// sync.c
#include <unistd.h>
#include <sys/wait.h>
#include "sync.h"

int create_sync_pipes(container_sync_t *sync) {
    if (pipe(sync->parent_pipe) == -1) {
        return -1;
    }
    if (pipe(sync->child_pipe) == -1) {
        close(sync->parent_pipe[0]);
        close(sync->parent_pipe[1]);
        return -1;
    }
    return 0;
}

int wait_for_child_ready(container_sync_t *sync) {
    close(sync->child_pipe[1]); // Close write end
    char ready_signal;
    ssize_t result = read(sync->child_pipe[0], &ready_signal, 1);
    close(sync->child_pipe[0]);
    return (result == 1 && ready_signal == '1') ? 0 : -1;
```

```
}


int signal_parent_ready(container_sync_t *sync) {
    close(sync->child_pipe[0]); // Close read end
    char ready_signal = '1';
    ssize_t result = write(sync->child_pipe[1], &ready_signal, 1);
    close(sync->child_pipe[1]);
    return (result == 1) ? 0 : -1;
}
```

D. Core Logic Skeleton Code

Main container lifecycle coordination:

```c
// container.c

int container_create(const container_config_t *config, container_instance_t *container) {

    // TODO 1: Initialize container instance structure with cleanup list

    // TODO 2: Validate configuration (rootfs exists, privileges sufficient)

    // TODO 3: Allocate clone stack and register stack cleanup

    // TODO 4: Create synchronization pipes for parent-child coordination

    // TODO 5: Execute safe_clone() with CONTAINER_NS_FLAGS to create isolated child

    // TODO 6: Store child PID and register process cleanup function

    // TODO 7: Open namespace file descriptors from /proc/<child_pid>/ns/

    // TODO 8: Create and configure cgroup hierarchy for resource limits

    // TODO 9: Wait for child to complete mount namespace setup

    // TODO 10: Set up host-side networking (veth pair, bridge attachment)

    // TODO 11: Signal child to proceed with final application execution

    // Hint: Use cleanup list to ensure proper resource release on any failure

    return -1; // Replace with implementation

}


int container_init_process(void *arg) {

    container_config_t *config = (container_config_t*)arg;



    // TODO 1: Set up signal handlers for PID 1 responsibilities (zombie reaping)

    // TODO 2: Create mount namespace isolation with private propagation

    // TODO 3: Set up container root filesystem directory structure

    // TODO 4: Perform pivot_root operation to switch filesystem root

    // TODO 5: Mount essential filesystems (/proc, /sys, /dev) in container

    // TODO 6: Configure container network interface with assigned IP

    // TODO 7: Set container hostname for identity isolation

    // TODO 8: Signal parent that container setup is complete
```

```
    // TODO 9: Change to application working directory
    // TODO 10: Execute target application with execve()
    // Hint: Exit with specific error codes to communicate failure type to parent
    return -1; // Should never reach here if execve() succeeds
}

int container_wait(container_instance_t *container) {
    int status;
    pid_t result;

    // TODO 1: Wait for child process to exit using waitpid()
    // TODO 2: Check if child was killed by signal (WIFSIGNALED)
    // TODO 3: Extract exit code or signal number for error reporting
    // TODO 4: Check cgroup memory events for OOM detection
    // TODO 5: Return appropriate error code based on termination cause
    // Hint: Different termination types require different cleanup approaches
    return -1; // Replace with implementation
}

int container_destroy(container_instance_t *container) {
    // TODO 1: Ensure child process has fully terminated
    // TODO 2: Clean up network namespace and veth interfaces
    // TODO 3: Clean up mount namespace and filesystem mounts
    // TODO 4: Close all namespace file descriptors
    // TODO 5: Remove cgroup hierarchy and release resource controls
    // TODO 6: Execute all registered cleanup functions
    // TODO 7: Free container instance memory and allocated resources
    // Hint: Follow reverse order of creation to prevent dependency issues
    return -1; // Replace with implementation
```

```
}
```

Error propagation and recovery coordination:

```c
// error_handling.c

typedef enum {
    CONTAINER_ERROR_NONE = 0,
    CONTAINER_ERROR_VALIDATION = 1,
    CONTAINER_ERROR_PRIVILEGES = 2,
    CONTAINER_ERROR_NAMESPACES = 10,
    CONTAINER_ERROR_MOUNT = 20,
    CONTAINER_ERROR_NETWORK = 30,
    CONTAINER_ERROR_CGROUPS = 40,
    CONTAINER_ERROR_OOM = 100,
    CONTAINER_ERROR_KILLED = 200
} container_error_t;

int handle_container_error(container_instance_t *container, container_error_t error) {
    // TODO 1: Log error details with specific failure component
    // TODO 2: Determine cleanup scope based on error type and setup progress
    // TODO 3: Signal child process termination if necessary
    // TODO 4: Execute appropriate cleanup sequence for current state
    // TODO 5: Collect any additional error information (logs, resource usage)
    // TODO 6: Return standardized error code for caller
    // Hint: Error cleanup must be safe to call regardless of setup progress
    return -1; // Replace with implementation
}

int propagate_child_error(pid_t child_pid, int child_status) {
    // TODO 1: Check if child exited normally or was killed by signal
    // TODO 2: Extract error code from child exit status
    // TODO 3: Map child error codes to container error types
    // TODO 4: Check for OOM killer involvement via cgroup events
```

```
    // TODO 5: Return appropriate container error code
    // Hint: Child exit codes encode which component failed during setup
    return CONTAINER_ERROR_NONE; // Replace with implementation
}
```

E. Language-Specific Hints

- Use `clone()` instead of `fork()` + `unshare()` to avoid race conditions in namespace setup
- The `SIGCHLD` handler for zombie reaping must be async-signal-safe (no malloc, printf, etc.)
- Stack for `clone()` grows downward on most architectures - allocate at high address
- Check `/proc/self/ns/` to verify namespace support before attempting creation
- Use `eventfd()` for more reliable parent-child synchronization than pipes
- `pivot_root()` requires both old and new root to be on different filesystems
- Network namespace operations require `CAP_NET_ADMIN` capability
- Cgroup cleanup requires removing child cgroups before parent cgroups

F. Milestone Checkpoint

After implementing container lifecycle coordination:

Test Container Creation and Cleanup:

```bash
# Compile with all components
gcc -o container-basic src/*.c -I./src

# Test basic container creation
sudo ./container-basic create --rootfs /tmp/container-root --cmd /bin/sh

# Verify namespaces created correctly
sudo lsns -t pid,mnt,net -p $(pgrep container-basic)

# Test error handling
./container-basic create --rootfs /nonexistent --cmd /bin/sh
echo "Exit code: $?"

# Test resource cleanup
sudo ./container-basic create --rootfs /tmp/container-root --memory 100M --cmd "sleep 1"
# Verify cgroup removed after container exits
ls /sys/fs/cgroup/memory/container-basic-* 2>/dev/null || echo "Cleanup successful"
```

Expected behavior:

- Container starts with all namespaces isolated
- Application runs with resource limits enforced
- Cleanup removes all created resources
- Error conditions trigger proper cleanup
- No resource leaks in namespace or cgroup hierarchies

G. Debugging Tips

Symptom	Likely Cause	Diagnosis	Fix
Container hangs during startup	Parent-child synchronization deadlock	Check pipe file descriptors, look for blocking reads	Add timeouts to synchronization waits
"Operation not permitted" on clone	Insufficient privileges for namespace creation	Check capabilities with `capsh --print`, verify CAP_SYS_ADMIN	Run as root or add required capabilities
Mount operations fail inside container	Mount namespace not properly isolated	Check /proc/mounts in container vs host	Ensure MS_PRIVATE propagation set correctly
Network interface not found	Veth pair creation or assignment failed	Check `ip link` for interface existence	Verify network namespace creation and interface movement
OOM killer terminates container immediately	Memory limit too low for application	Check cgroup memory.usage_in_bytes vs memory.limit_in_bytes	Increase memory limit or reduce application memory usage
Resources not cleaned up after exit	Cleanup functions not registered or executed	Check cleanup list registration and execution order	Ensure all setup operations register cleanup handlers
Child process becomes zombie	Parent not calling waitpid()	Check parent process signal handlers and main loop	Implement proper SIGCHLD handling or explicit wait calls

Error Handling and Edge Cases

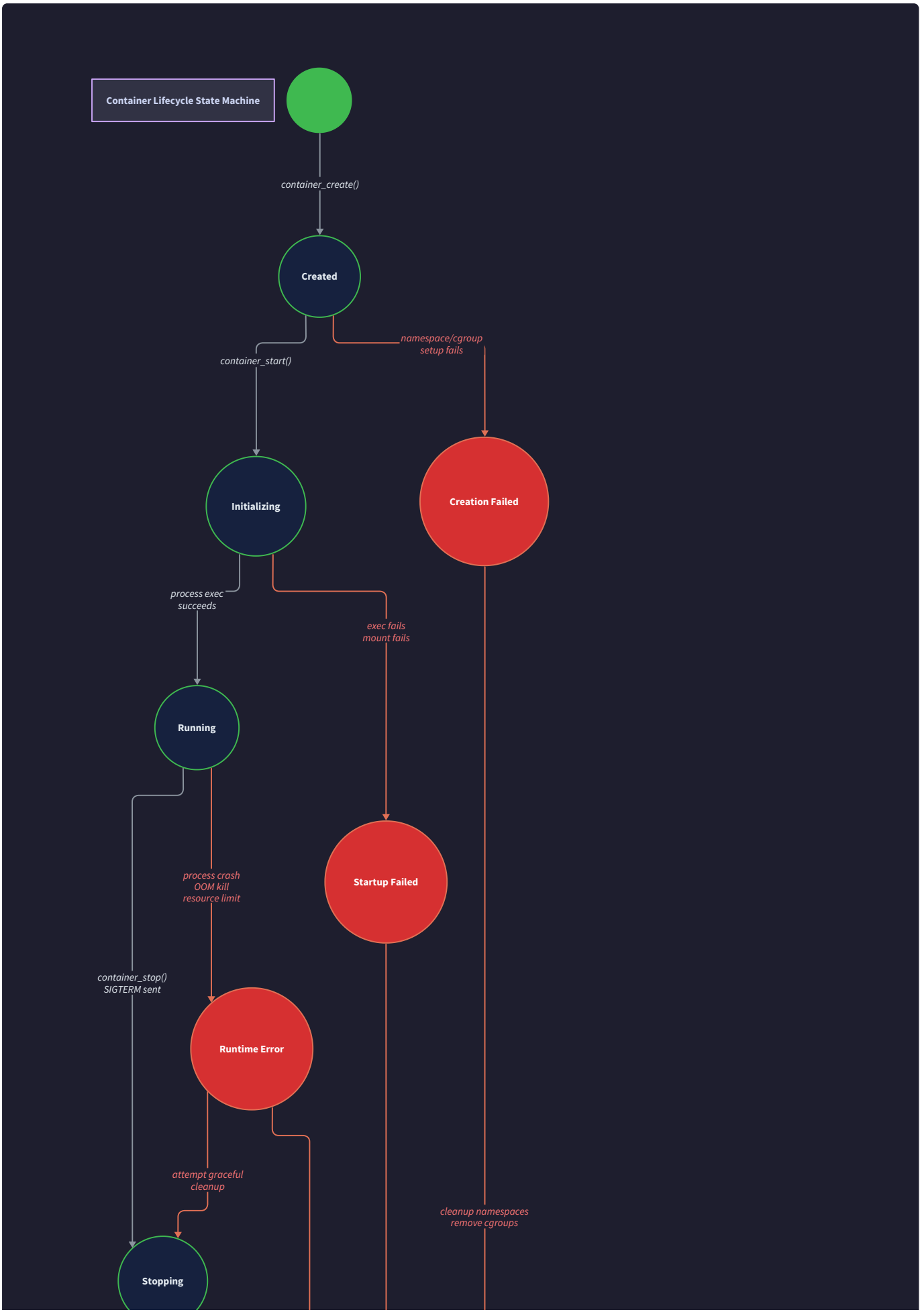
> **Milestone(s):** This section applies to all milestones (1-4), covering failure modes and recovery strategies across PID namespaces, mount namespaces, network namespaces, and cgroups components.

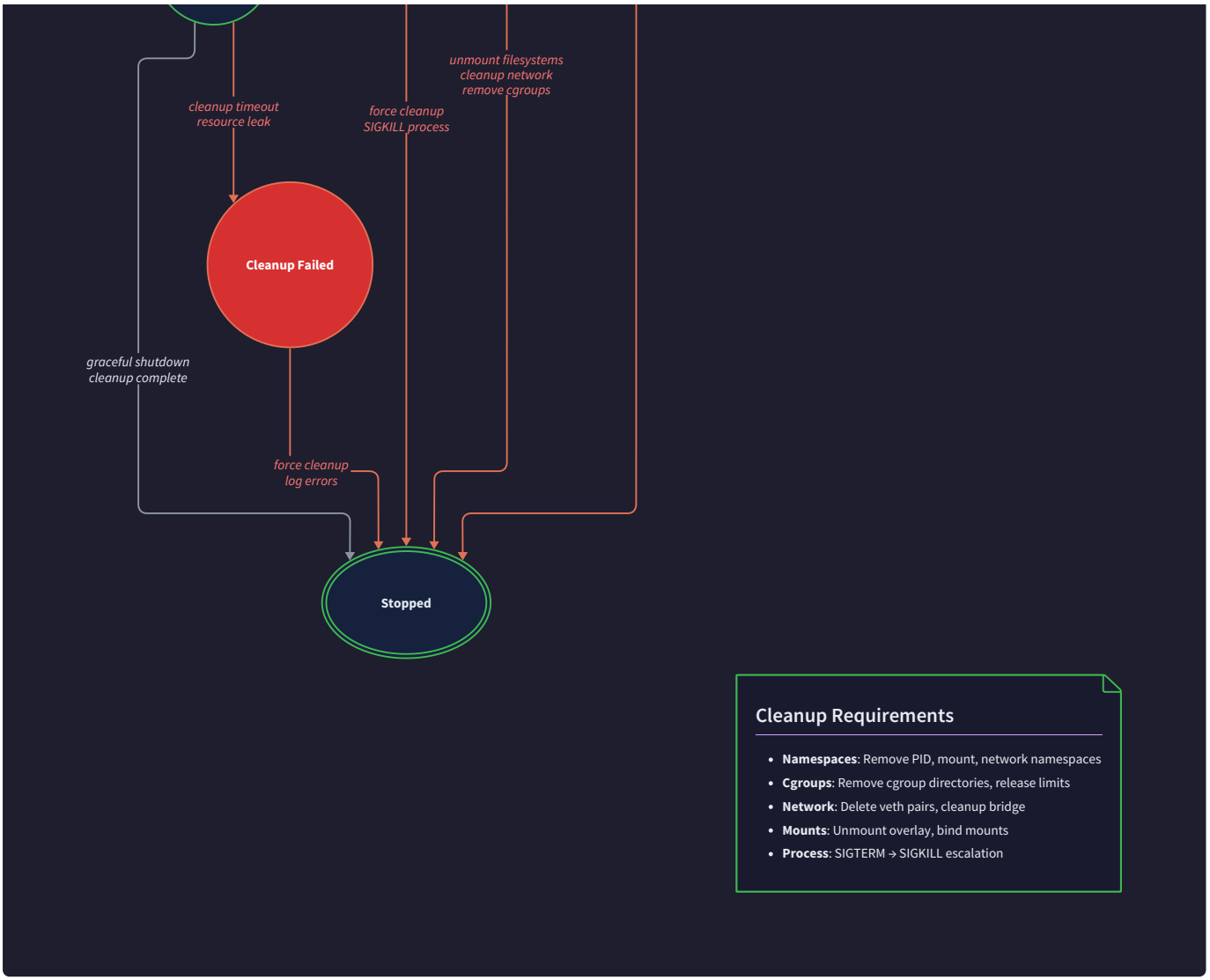
Building a container runtime involves orchestrating multiple kernel subsystems, each with distinct failure modes and complex interdependencies. When creating namespaces, configuring cgroups, or setting up networking, numerous failure scenarios can occur - from insufficient privileges to kernel feature unavailability to resource exhaustion. The challenge is not just handling individual component failures, but managing the cascading effects and cleanup requirements when failures occur during the multi-step container startup sequence.

Mental Model: Emergency Response System: Think of container error handling like a hospital's emergency response system. When a medical emergency occurs, the hospital has well-defined protocols for different types of incidents (cardiac arrest, trauma, stroke), clear escalation procedures, and systematic approaches to resource allocation and cleanup. Similarly, our container runtime needs protocols for different failure types (namespace creation, resource exhaustion, network setup), clear error propagation paths, and systematic cleanup procedures that prevent resource leaks even when multiple components fail simultaneously.

The complexity arises because container creation involves a precise sequence of operations that create kernel resources (namespaces, cgroups, network interfaces) and establish process relationships. Unlike simple application failures that can be handled with basic exception handling, container failures often occur in privileged kernel operations with partial state changes that must be carefully unwound. A failure during network namespace creation might leave a veth interface orphaned on the host, while a failure during mount namespace setup might leave filesystems mounted that prevent proper cleanup.

Our error handling strategy follows three core principles: **early detection** to catch problems before they cascade, **graceful degradation** to maintain system stability when individual components fail, and **complete cleanup** to ensure no kernel resources leak even during complex failure scenarios. Each component must track its own resources and provide cleanup functions, while the overall container system coordinates cleanup order and error propagation.


Container Lifecycle State Machine
container_create()
Created
namespace/cgroup setup fails
container_start()
Initializing
Creation Failed
process exec succeeds
exec fails mount fails
Running
Startup Failed
process crash OOM kill resource limit
container_stop() SIGTERM sent
Runtime Error
attempt graceful cleanup
cleanup namespaces remove cgroups
Stopping



Namespace Creation Failures

Namespace creation represents one of the most common failure points in container runtime operation because it depends on kernel features, system privileges, and resource availability that may not be guaranteed. Understanding and handling these failures properly is essential for building a robust container system that can operate reliably across different environments and configurations.

Mental Model: Apartment Building Permits: Think of namespace creation like obtaining permits to build different sections of an apartment building. You need separate permits for plumbing (network namespace), electrical (PID namespace), and structural changes (mount namespace). If the city denies any permit due to zoning restrictions (kernel features disabled), insufficient fees (privilege levels), or resource limits (too many existing permits), you can't proceed with that portion of construction. However, you need to handle the permit denial gracefully and clean up any permits you've already obtained.

The most frequent namespace creation failure is **insufficient privileges**. Creating namespaces typically requires either root privileges or specific capabilities like `CAP_SYS_ADMIN`. When running without sufficient privileges, the `clone()` or `unshare()` system calls fail with `EPERM`, indicating permission denied. This failure mode is particularly challenging because it often occurs after some setup has already completed, requiring careful cleanup of partial state.

Kernel feature unavailability represents another critical failure mode. Not all Linux kernels are compiled with namespace support enabled, and some container execution environments deliberately disable certain namespace types for security reasons. The `clone()` call may fail with `EINVAL` when attempting to create unsupported namespace types, or with `ENOSYS` if the kernel lacks namespace support entirely.

> **Decision: Graceful Namespace Degradation Strategy**
>
> - **Context**: Some environments may support only a subset of namespaces (e.g., PID but not network), requiring decisions about partial container functionality
> - **Options Considered**:
>   - Fail completely if any namespace creation fails
>   - Allow containers to run with reduced isolation
>   - Make namespace types configurable with fallback behavior
> - **Decision**: Fail completely for core namespaces (PID, mount) but allow optional degradation for network namespaces
> - **Rationale**: PID and mount isolation are fundamental to container security, while network isolation can be provided by external tools in some deployment scenarios
> - **Consequences**: Ensures minimum security guarantees while allowing deployment flexibility in constrained environments

Failure Mode	Error Code	Detection Method	Recovery Strategy
Insufficient privileges	EPERM	Check return value from clone/unshare	Log clear error message, suggest running as root or with capabilities
Kernel feature disabled	EINVAL	Attempt namespace creation, check errno	Check /proc/sys/kernel for namespace support, provide feature-specific error
Resource exhaustion	ENOMEM	Memory allocation failure during clone	Free existing allocations, reduce memory requirements, retry with backoff
Process limit reached	EAGAIN	Clone fails due to process limits	Wait for processes to exit, check ulimits, suggest increasing limits
Namespace limit exceeded	ENOSPC	System namespace limit reached	Clean up unused namespaces, suggest system-level tuning

The **namespace creation error handling function** must distinguish between recoverable and non-recoverable failures. Privilege failures typically require user intervention (running with sudo or setting capabilities), while resource exhaustion might be temporary and allow for retry strategies. The error handling logic should provide actionable error messages that help operators understand both the immediate cause and potential solutions.

Resource exhaustion during namespace creation can manifest in several ways. Memory exhaustion (`ENOMEM`) during the `clone()` call indicates the kernel cannot allocate internal data structures for the new namespace. This often occurs in memory-constrained environments or when creating many containers rapidly. The recovery strategy involves releasing any allocated resources and potentially implementing exponential backoff for retry attempts.

Process limit exhaustion (`EAGAIN`) occurs when the system or user process limits prevent creating new processes via `clone()` . This is distinct from namespace limits themselves - the system may support creating namespaces but refuse to create the process that would inhabit them. Recovery requires checking both system-wide process limits (`/proc/sys/kernel/pid_max`) and user limits (`ulimit -u`).

The **namespace handle cleanup** process becomes critical when creation fails partway through establishing multiple namespaces. Since namespace creation often involves creating PID, mount, and network namespaces in sequence, a failure during network namespace creation requires cleaning up the successfully created PID and mount namespaces. Each namespace file descriptor in the `namespace_fds` array of the `container_instance_t` structure must be closed properly to release kernel resources.

> The fundamental principle for namespace creation error handling is **atomic semantics** - either all required namespaces are successfully created, or none are left in a partial state that could cause resource leaks or security vulnerabilities.

Timing-dependent failures represent a subtle category of namespace creation issues. Creating multiple namespaces in sequence can encounter race conditions where kernel resources become unavailable between namespace creation calls. For example, the system might successfully create a PID namespace but fail to create the network namespace due to resource exhaustion that occurred after the PID namespace consumed memory. These failures require careful ordering and resource validation before beginning the namespace creation sequence.

The **error propagation mechanism** must handle failures that occur in the child process after `clone()` successfully creates the process but before the child can complete its namespace setup. Since the child runs in a separate process, it cannot directly return error codes to the parent. Instead, it must use the parent-child synchronization pipes defined in `container_sync_t` to communicate setup failures back to the parent process for proper cleanup coordination.

⚠️ **Pitfall: Partial Namespace Cleanup** Many implementations fail to properly clean up namespaces when creation fails partway through the sequence. For example, if PID and mount namespaces are created successfully but network namespace creation fails, the PID and mount namespace file descriptors must be closed to release kernel resources. Forgetting this cleanup can lead to namespace handle leaks that persist until the parent process exits. Always iterate through the `namespace_fds` array and close any valid file descriptors (non-negative values) during error cleanup.

⚠️ **Pitfall: Ignoring Kernel Version Dependencies** Different Linux kernel versions support different namespace types and have varying resource limits. Attempting to create user namespaces on kernels older than 3.8 or network namespaces without proper kernel configuration will fail in non-obvious ways. Always check kernel version compatibility and provide clear error messages that indicate minimum kernel requirements when namespace creation fails with `EINVAL` or `ENOSYS`.

Resource Exhaustion

Resource exhaustion scenarios represent some of the most complex failure modes in container systems because they can occur at any point during container lifecycle and often involve kernel-level enforcement mechanisms that terminate processes without warning. Unlike application-level resource limits that can be checked programmatically, kernel resource limits like memory exhaustion invoke the OOM killer, which terminates processes based on heuristics rather than application priorities.

Mental Model: City Utility Management: Think of resource exhaustion like a city managing utilities during peak demand. When electricity demand exceeds generation capacity, the city doesn't politely ask residents to reduce usage - it implements rolling blackouts based on priority zones. Similarly, when container memory usage exceeds cgroup limits, the kernel doesn't negotiate - it invokes the OOM killer to terminate processes based on kernel heuristics, not application priorities. The container runtime must be prepared for sudden process termination and have cleanup mechanisms that work even when primary processes are killed unexpectedly.

Memory exhaustion and OOM conditions represent the most dramatic form of resource exhaustion because they result in immediate process termination rather than graceful degradation. When a container exceeds its memory limit configured through the memory cgroup controller, the kernel's OOM killer selects processes within that cgroup for termination. The selection algorithm considers factors like memory usage, process age, and OOM adjustment scores, but the container runtime has limited control over which processes are terminated first.

The **OOM killer behavior** varies significantly between cgroups v1 and v2, affecting how container processes experience memory pressure. In cgroups v1, the OOM killer typically terminates the highest memory-consuming process within the cgroup when the limit is exceeded. In cgroups v2, the kernel implements more sophisticated memory pressure handling with early warning signals before reaching hard limits. Container runtimes must handle both scenarios and provide appropriate monitoring and recovery mechanisms.

Resource Type	Exhaustion Trigger	Kernel Response	Detection Method	Recovery Strategy
Memory	Usage exceeds cgroup limit	OOM killer terminates processes	Process exit status 137 (SIGKILL)	Monitor memory pressure, implement graceful restart
CPU	Usage exceeds quota	Process throttling	Increased execution time	Monitor throttling statistics, adjust limits
Process count	Forks exceed pids limit	fork() fails with EAGAIN	Failed process creation	Monitor process count, clean up zombie processes
File descriptors	Open files exceed limit	open() fails with EMFILE	File operation failures	Close unused descriptors, increase limits
Network bandwidth	Traffic exceeds shaping rules	Packet dropping/queuing	Network performance degradation	Implement traffic prioritization

CPU throttling provides a more graceful form of resource exhaustion where the kernel doesn't terminate processes but instead restricts their execution time. When a container exceeds its CPU quota within the configured period, the kernel throttles the processes by putting them to sleep until the next scheduling period begins. This throttling is transparent to the application but manifests as increased execution time for CPU-bound operations.

The **CPU controller statistics** available through the cgroup filesystem provide detailed information about throttling events, including the number of times processes were throttled and the total time spent in throttled state. Container runtimes should monitor these statistics to detect CPU pressure and potentially adjust limits or provide warnings about application performance degradation.

Process limit exhaustion occurs when containers attempt to create more processes or threads than allowed by the pids controller. Unlike memory exhaustion which terminates existing processes, process limit exhaustion prevents new process creation while leaving existing processes running normally. The `fork()`, `clone()`, and `pthread_create()` operations fail with `EAGAIN`, allowing applications to handle the failure gracefully if they implement appropriate error checking.

> The key insight for process limit handling is that applications must be designed to handle process creation failures gracefully, unlike memory exhaustion where process termination is unavoidable and must be handled through external monitoring and restart mechanisms.

File descriptor exhaustion can occur both at the process level (hitting the per-process `RLIMIT_NOFILE` limit) and at the system level (hitting the global file descriptor limit). Container applications that open many files, sockets, or pipes without proper cleanup can quickly exhaust available file descriptors. The failure manifests as `EMFILE` (too many open files) or `ENFILE` (file table overflow) errors from system calls like `open()`, `socket()`, or `pipe()`.

Network resource exhaustion primarily manifests through bandwidth limits imposed by traffic control mechanisms rather than hard cgroup limits. When containers exceed configured network bandwidth limits, the kernel may drop packets, increase latency through queuing, or apply traffic shaping rules that degrade network performance. Unlike CPU or memory limits, network exhaustion rarely causes immediate application failure but instead degrades performance gradually.

The **memory pressure monitoring** mechanism provides early warning before OOM killer activation through cgroup event notifications. The memory controller exposes pressure events through the `memory.pressure_level` interface in cgroups v1 and through pressure stall information in cgroups v2. Container runtimes can register for these notifications to implement proactive memory management, such as requesting applications to free caches or triggering graceful container shutdown before hitting hard limits.

Resource exhaustion recovery strategies must consider the cascading effects of resource limits across multiple containers sharing the same host system. When one container exhausts memory and triggers OOM killer activity, other containers may experience performance degradation due to increased system memory pressure. The recovery mechanism should monitor system-wide resource pressure in addition to per-container limits.

⚠️ **Pitfall: Ignoring Memory Pressure Warnings** Many container implementations only detect memory problems when the OOM killer activates, but cgroups provide earlier warning signals through memory pressure notifications. Ignoring these warnings means missing opportunities for graceful degradation, such as reducing cache sizes or checkpointing application state. Monitor the `memory.pressure_level` events and implement proactive memory management rather than reactive OOM handling.

⚠️ **Pitfall: Misunderstanding CPU Throttling Effects** CPU throttling doesn't just slow down the main application - it affects all processes in the cgroup, including monitoring agents, log collectors, and health check scripts. This can create cascading failures where health checks timeout due to throttling, causing orchestration systems to restart healthy containers unnecessarily. Account for monitoring overhead when setting CPU limits and consider excluding critical monitoring processes from throttled cgroups.

⚠️ **Pitfall: Process Limit Bypass Through Threading** The pids controller counts threads as processes, but applications might attempt to bypass process limits by using thread pools or async I/O instead of forking processes. However, thread creation can still fail when hitting the pids limit, causing subtle application failures that manifest as hanging requests or degraded performance rather than obvious process creation errors. Test applications under process pressure to ensure they handle threading failures appropriately.

Cleanup Failures

Cleanup failures represent one of the most insidious categories of container runtime errors because they often go unnoticed until they accumulate into system-wide problems. Unlike startup failures that immediately signal problems through failed container launches, cleanup failures manifest as gradual resource leaks, orphaned kernel objects, and eventual system instability. The challenge is compounded by the fact that cleanup operations often occur during error conditions when the system is already in a degraded state.

Mental Model: Restaurant Closing Procedures: Think of cleanup failures like a restaurant that doesn't properly close each night. If the staff forgets to turn off equipment, clean tables, or secure the building, small problems accumulate over time. A few unwashed dishes become a pest problem; unlocked doors become security issues;

equipment left running becomes fire hazards. Similarly, cleanup failures in containers start small - a forgotten veth interface, an unclosed namespace file descriptor, an abandoned cgroup directory - but accumulate into system-wide resource exhaustion, kernel object leaks, and eventually system instability requiring reboot to resolve.

Partial cleanup scenarios occur when some cleanup operations succeed while others fail, leaving the system in an inconsistent state. For example, a container might successfully remove its cgroup directory but fail to delete the veth interface due to network configuration errors. The container appears to have cleaned up from the cgroup perspective, but network resources remain allocated, potentially causing IP address conflicts or interface name collisions for future containers.

The **cleanup ordering dependencies** create complex failure scenarios where the sequence of cleanup operations affects success rates. Network interfaces must be removed before closing network namespace file descriptors; mount points must be unmounted before removing mount namespaces; processes must be terminated before removing cgroups. When cleanup operations fail, subsequent operations in the sequence may also fail due to dependency violations.

Cleanup Component	Dependencies	Failure Modes	Recovery Strategy
Process termination	Must complete before cgroup removal	Process refuses to die, zombie accumulation	SIGTERM → wait → SIGKILL → force cleanup
Network interfaces	Must remove before closing network namespace	Interface busy, namespace unreachable	Force interface down, use different namespace
Mount points	Must unmount before namespace cleanup	Filesystem busy, permission denied	Lazy unmount (MNT_DETACH), force unmount
Cgroup directories	Processes must exit first	Directory not empty, permission denied	Kill remaining processes, recursive removal
Namespace file descriptors	Close after removing associated resources	File descriptor corruption	Force close, accept potential leaks

Cgroup cleanup failures frequently occur because cgroup directories cannot be removed while they still contain processes, even zombie processes awaiting reaping. The kernel enforces this restriction to maintain accounting accuracy, but it creates complex cleanup scenarios when container processes don't exit cleanly. A single zombie process can prevent cgroup cleanup, leading to accumulating cgroup directories that consume kernel memory and eventually hit system limits.

The **hierarchical cleanup strategy** addresses these dependencies by implementing cleanup in reverse dependency order: terminate processes first, then remove cgroups, then clean up namespaces, then remove network interfaces. However, each step can fail independently, requiring the cleanup system to continue with subsequent steps even when earlier steps fail partially.

Mount point cleanup failures occur when filesystems remain busy due to open file descriptors, running processes, or active bind mounts. The `umount()` system call fails with `EBUSY`, leaving mount points active and

preventing proper mount namespace cleanup. The recovery strategy involves using `MNT_DETACH` for lazy unmounting, which removes the mount point from the namespace view immediately but defers actual filesystem cleanup until all references are released.

> **Decision: Cleanup Continuation Strategy**
>
> - **Context**: When cleanup operations fail, the system must decide whether to abort cleanup entirely or continue with remaining operations
> - **Options Considered**:
>   - Stop cleanup on first failure to maintain consistency
>   - Continue cleanup despite failures to release maximum resources
>   - Implement retry mechanisms with exponential backoff
> - **Decision**: Continue cleanup despite failures, with comprehensive error logging and retry for specific failure types
> - **Rationale**: Resource leaks cause cumulative system degradation, so releasing as many resources as possible is preferable to maintaining perfect consistency
> - **Consequences**: Requires careful error aggregation and reporting to surface cleanup issues without stopping resource recovery

Network cleanup failures often involve orphaned veth interfaces that remain on the host side even after the container network namespace is destroyed. When network namespace destruction fails or is incomplete, the host-side veth interface may persist without its paired container interface, creating a dangling interface that consumes network namespace slots and may cause naming conflicts for future containers.

The **cleanup function registration mechanism** using the `cleanup_list_t` structure provides a systematic approach to ensure all allocated resources are released even when cleanup operations fail. Each component registers its cleanup functions during resource allocation, creating a comprehensive list of cleanup operations that execute in reverse order during container destruction. This pattern ensures that cleanup attempts occur for all resources, even if individual cleanup functions encounter failures.

Error aggregation during cleanup requires collecting multiple error conditions and presenting them coherently to operators. Since cleanup continues despite individual failures, the final cleanup result may include errors from multiple components. The error reporting mechanism must distinguish between failures that indicate resource leaks (requiring operator attention) and failures that are purely informational (such as attempting to remove already-removed resources).

Timeout-based cleanup addresses scenarios where cleanup operations hang indefinitely due to kernel deadlocks, filesystem issues, or network problems. Each cleanup operation should have reasonable timeouts to prevent container destruction from hanging indefinitely. When timeouts occur, the cleanup system must decide whether to escalate to more aggressive cleanup methods (such as force-killing processes) or accept resource leaks to maintain system stability.

⚠️ **Pitfall: Cleanup Function Exception Safety** Cleanup functions themselves can fail and must not throw exceptions or abort the cleanup process. A common mistake is writing cleanup functions that assume resources

are in a valid state, causing cleanup failures when those resources were already partially destroyed. Always check resource validity before attempting cleanup operations and handle cases where resources may already be released by other cleanup paths.

⚠️ **Pitfall: Ignoring Cleanup Ordering** Attempting to remove cgroups before terminating processes or closing namespace file descriptors before unmounting filesystems will cause cleanup failures that could be avoided with proper ordering. These failures often appear intermittent because they depend on timing between process termination and cleanup execution. Implement explicit dependency tracking between cleanup operations and ensure proper ordering even under race conditions.

⚠️ **Pitfall: Incomplete Error Context in Cleanup** Cleanup failures often occur in degraded system states where normal debugging information isn't available. Cleanup error messages must include sufficient context to diagnose problems without access to the original container configuration or process state. Log the specific resource being cleaned (PID, interface name, cgroup path) and the exact error condition, not just generic "cleanup failed" messages that provide no actionable information.

Implementation Guidance

The error handling implementation requires robust error detection, classification, and recovery mechanisms that coordinate across all container components. The implementation focuses on early failure detection, comprehensive error reporting, and systematic cleanup that prevents resource leaks even during cascading failures.

Technology Recommendations:

Component	Approach	Implementation
Error Classification	Structured error codes	container_error_t enum with specific failure types
Error Context	Detailed error information	Error structures with component, operation, and system error details
Cleanup Management	Resource tracking	cleanup_list_t with function pointers for systematic resource cleanup
Timeout Handling	Non-blocking cleanup	Timer-based cleanup with escalation to force operations
Error Aggregation	Multiple error collection	Error arrays that collect failures from multiple cleanup operations

Recommended File Structure:

```
container-basic/
  src/
    error/
      error_types.h          ← Error codes and classification
      error_handlers.c       ← Error handling and recovery functions
      cleanup.h              ← Cleanup list management interface
      cleanup.c              ← Cleanup function registration and execution
    container/
      container_errors.h     ← Container-specific error handling
      container_errors.c     ← Container error coordination and propagation
    namespace/
      namespace_errors.c     ← Namespace creation failure handling
    cgroups/
      cgroups_errors.c       ← Resource exhaustion detection and handling
    network/
      network_errors.c       ← Network setup failure handling
  tests/
    error_scenarios/         ← Error injection and recovery testing
```

Error Classification Infrastructure:

```c
#include <errno.h>
#include <string.h>
#include <sys/wait.h>

// Complete error type enumeration for container operations
typedef enum {
    CONTAINER_ERROR_NONE = 0,
    CONTAINER_ERROR_VALIDATION,     // Configuration validation failed
    CONTAINER_ERROR_NAMESPACES,     // Namespace creation failed
    CONTAINER_ERROR_MOUNT,          // Mount operations failed
    CONTAINER_ERROR_NETWORK,        // Network setup failed
    CONTAINER_ERROR_CGROUPS,        // Cgroup setup failed
    CONTAINER_ERROR_OOM,            // Out of memory condition
    CONTAINER_ERROR_PRIVILEGES,     // Insufficient permissions
    CONTAINER_ERROR_KERNEL_FEATURE, // Kernel feature unavailable
    CONTAINER_ERROR_RESOURCE_LIMIT, // System resource limit hit
    CONTAINER_ERROR_CLEANUP,        // Cleanup operation failed
    CONTAINER_ERROR_TIMEOUT         // Operation timeout
} container_error_t;

// Detailed error context for debugging and recovery
typedef struct container_error_context {
    container_error_t type;
    const char* component;      // Which component failed
    const char* operation;      // What operation was being performed
    int system_errno;           // System error code if applicable
    const char* details;        // Human-readable error description
    int recovery_attempted;     // Whether recovery was attempted
} container_error_context_t;
```

```
// Error context creation and management
container_error_context_t* create_error_context(container_error_t type,
                                                const char* component,
                                                const char* operation) {
    // TODO 1: Allocate error context structure
    // TODO 2: Initialize fields with provided parameters
    // TODO 3: Capture current errno value for system_errno field
    // TODO 4: Generate human-readable error message for details field
    // TODO 5: Return initialized context structure
}

// Error classification from system errno values
container_error_t classify_namespace_error(int errno_val) {
    // TODO 1: Check for permission errors (EPERM) → CONTAINER_ERROR_PRIVILEGES
    // TODO 2: Check for feature errors (EINVAL, ENOSYS) → CONTAINER_ERROR_KERNEL_FEATURE
    // TODO 3: Check for resource errors (ENOMEM, EAGAIN) → CONTAINER_ERROR_RESOURCE_LIMIT
    // TODO 4: Check for validation errors (EINVAL with bad params) →
CONTAINER_ERROR_VALIDATION
    // TODO 5: Return CONTAINER_ERROR_NAMESPACES for other namespace-related errors
}
```

Cleanup List Management Infrastructure:

```c
#include <stdlib.h>

// Cleanup function pointer type
typedef void (*cleanup_function_t)(void* data);

// Cleanup list node for systematic resource management
typedef struct cleanup_list {
    cleanup_function_t cleanup_func;
    void* data;
    const char* description;    // For debugging cleanup failures
    struct cleanup_list* next;
} cleanup_list_t;

// Register cleanup function for resource that needs cleanup on failure
void register_cleanup(cleanup_list_t** list, cleanup_function_t func,
                      void* data, const char* description) {
    // TODO 1: Allocate new cleanup list node
    // TODO 2: Initialize node with function pointer and data
    // TODO 3: Set description for debugging cleanup failures
    // TODO 4: Add node to front of cleanup list for LIFO execution
    // TODO 5: Update list head pointer to new node
}

// Execute all cleanup functions in reverse registration order
int execute_cleanup(cleanup_list_t** list) {
    // TODO 1: Iterate through cleanup list from head to tail
    // TODO 2: Call each cleanup function with its associated data
    // TODO 3: Log cleanup function description and any errors
    // TODO 4: Continue cleanup even if individual functions fail
    // TODO 5: Free cleanup list nodes and set list pointer to NULL
```

```
    // TODO 6: Return count of failed cleanup operations
}

// Cleanup functions for specific resource types
void cleanup_namespace_fd(void* fd_ptr) {
    // TODO 1: Cast void pointer to int pointer for file descriptor
    // TODO 2: Check if file descriptor value is valid (>= 0)
    // TODO 3: Close file descriptor and check for errors
    // TODO 4: Set file descriptor to -1 to mark as cleaned up
    // TODO 5: Log cleanup completion or failure for debugging
}

void cleanup_cgroup_path(void* path_ptr) {
    // TODO 1: Cast void pointer to char pointer for cgroup path
    // TODO 2: Verify cgroup path exists before attempting removal
    // TODO 3: Remove cgroup directory using rmdir system call
    // TODO 4: Handle ENOENT (already removed) as success condition
    // TODO 5: Log cgroup cleanup result with path information
}

void cleanup_network_interface(void* if_name_ptr) {
    // TODO 1: Cast void pointer to char pointer for interface name
    // TODO 2: Check if interface exists in current namespace
    // TODO 3: Remove interface using netlink socket or ip command
    // TODO 4: Handle interface already removed as success condition
    // TODO 5: Log interface cleanup result with interface name
}
```

Resource Exhaustion Detection:

```c
#include <sys/stat.h>

#include <fcntl.h>


// Monitor memory pressure in container cgroup

int check_memory_pressure(const char* cgroup_path, int* pressure_level) {

    // TODO 1: Construct path to memory.pressure_level file in cgroup

    // TODO 2: Open pressure level file for reading

    // TODO 3: Read current pressure level value (0=low, 1=medium, 2=critical)

    // TODO 4: Close file and handle any read errors appropriately

    // TODO 5: Set pressure_level output parameter and return success status

}


// Monitor CPU throttling statistics

int check_cpu_throttling(const char* cgroup_path, long* throttled_time) {

    // TODO 1: Construct path to cpu.stat file in cgroup directory

    // TODO 2: Open and read cpu statistics file content

    // TODO 3: Parse throttled_time field from statistics output

    // TODO 4: Calculate throttling rate since last check

    // TODO 5: Return throttling statistics through output parameter

}


// Check for OOM killer activity in container

int detect_oom_condition(pid_t container_pid, int* oom_occurred) {

    // TODO 1: Check exit status of container process for signal 137 (SIGKILL)

    // TODO 2: Read memory.oom_control file to check for OOM events

    // TODO 3: Parse kernel log messages for OOM killer activity

    // TODO 4: Correlate OOM events with container PID and cgroup

    // TODO 5: Set oom_occurred flag and return detection status

}
```

Comprehensive Error Handling Coordination:

```c
// Main container error handling function
int handle_container_error(container_instance_t* container,
                           container_error_context_t* error) {
    // TODO 1: Log detailed error context with component and operation info
    // TODO 2: Classify error severity (fatal vs recoverable)
    // TODO 3: Execute component-specific error handling based on error type
    // TODO 4: Execute cleanup list to release partial resources
    // TODO 5: Update container state to ERROR and preserve error context
    // TODO 6: Return appropriate error code for caller handling
}

// Propagate errors from child process to parent
int propagate_child_error(pid_t child_pid, int child_status) {
    // TODO 1: Check if child exited normally or was terminated by signal
    // TODO 2: Extract exit code or signal number from child status
    // TODO 3: Read error information from child through sync pipes if available
    // TODO 4: Classify child error into appropriate container_error_t type
    // TODO 5: Create error context with child failure details
    // TODO 6: Return classified error for parent process handling
}
```

Milestone Checkpoints:

After implementing error handling for each milestone, verify the following behaviors:

Milestone 1 (PID Namespace) Error Handling:

- Run container creation without root privileges - should fail with clear privilege error message rather than cryptic system error
- Create containers rapidly to trigger process limit exhaustion - should detect EAGAIN and provide actionable error message
- Verify zombie processes are reaped even when container init process encounters errors during startup

Milestone 2 (Mount Namespace) Error Handling:

- Attempt container creation with non-existent rootfs path - should fail during validation with clear error about missing directory
- Fill up filesystem to trigger mount failures - should detect ENOSPC and clean up partial mount points
- Test pivot_root failure scenarios by using rootfs on read-only filesystem - should restore original mount state

Milestone 3 (Network Namespace) Error Handling:

- Create containers when veth creation fails due to interface name conflicts - should generate unique interface names or fail with specific naming error
- Test network setup when bridge doesn't exist - should provide clear error about bridge configuration rather than generic network error
- Verify network cleanup removes orphaned veth interfaces even when namespace destruction fails

Milestone 4 (Cgroups) Error Handling:

- Set memory limit higher than system RAM to test validation - should reject invalid configuration before attempting container creation
- Trigger OOM conditions by setting very low memory limits - should detect OOM through process exit status and provide clear OOM error indication
- Test cgroup cleanup when processes refuse to terminate - should escalate to SIGKILL and continue cleanup despite process cleanup failures

Testing Strategy

> **Milestone(s):** This section provides testing approaches for all milestones (1-4), establishing verification methods for PID namespace isolation, mount namespace filesystem separation, network namespace connectivity, and cgroups resource enforcement.

Testing a container runtime presents unique challenges because the system fundamentally alters the execution environment through kernel namespaces and resource controls. Unlike traditional application testing where functions operate in a predictable environment, container testing must verify that isolation mechanisms actually work—that processes cannot escape their boundaries, that resource limits are enforced, and that cleanup properly removes all traces of container execution.

Mental Model: Laboratory Safety Testing

Think of container testing like verifying laboratory safety protocols. Just as a biosafety lab must prove its containment works—that dangerous specimens cannot escape their isolation chambers, that air filtration systems prevent contamination, and that emergency procedures properly decontaminate equipment—container testing must demonstrate that process isolation holds under stress, resource limits prevent one container from starving others, and cleanup procedures leave no dangerous residue.

The testing approach mirrors laboratory verification: isolation tests ensure boundaries hold, resource limit tests verify controls work under pressure, and cleanup tests confirm complete decontamination. Each test must run in a controlled environment where we can safely create "dangerous" conditions (resource exhaustion, malicious processes) and verify they remain contained.

Isolation Verification Tests

Isolation verification tests form the foundation of container testing because they validate the core security and reliability promises of containerization. These tests must prove that namespaces create genuine boundaries that malicious or buggy processes cannot breach.

The **PID namespace isolation test** verifies that process trees remain completely separate between host and container. The test creates a container process that spawns multiple child processes with known behavior patterns, then verifies from the host that the container's PID assignments follow namespace rules. Inside the container, the init process should appear as PID 1, with child processes receiving sequential PIDs starting from 2. Simultaneously, the host should see the container processes using entirely different PID values from its own namespace.

Test Scenario	Container View	Host View	Verification Method
Container init process	PID 1	Real PID assigned by host	Read `/proc/self/stat` inside container vs external `ps` output
Container child process	PID 2, 3, 4...	Different real PIDs	Compare `/proc` entries inside vs outside namespace
Process visibility	Only container processes visible	Container processes visible with real PIDs	Check `/proc` directory listings
Kill signal isolation	Cannot signal host processes	Host can signal container processes	Attempt cross-namespace kill operations

The test implementation involves creating a container that runs a test program which forks several processes, each writing their perceived PID to shared storage. Simultaneously, the host monitors the actual PIDs assigned to these processes. The test succeeds only if the container processes consistently report PIDs 1, 2, 3... while the host observes different values, and if attempts to signal non-existent PIDs from within the container fail appropriately.

Mount namespace isolation testing verifies that filesystem operations within the container cannot affect the host filesystem and that the container sees only its intended filesystem view. The test creates a container with a custom root filesystem, performs various mount operations inside the container, then verifies these changes remain invisible to the host.

The test procedure involves setting up a temporary directory structure as the container root, mounting it into a mount namespace, then performing operations that would be dangerous if they escaped the namespace: creating device nodes, mounting additional filesystems, and modifying system directories. After each operation, the test

verifies that the host filesystem remains unchanged and that the container's view reflects only intended modifications.

Mount Operation	Container Result	Host Verification	Failure Indicator
Create device node	Device appears in container `/dev`	Host `/dev` unchanged	New device visible on host
Mount tmpfs on `/tmp`	Container sees empty `/tmp`	Host `/tmp` unchanged	Host `/tmp` becomes tmpfs
Bind mount host directory	Directory visible in container	Original host path unchanged	Unexpected mount in host namespace
Unmount `/proc`	Container loses `/proc`	Host `/proc` still functional	Host loses `/proc` access

Network namespace isolation testing demonstrates that container networking operates independently from host networking while maintaining intended connectivity paths. The test creates multiple containers with isolated network namespaces, configures networking between them, then verifies that network traffic follows only the intended paths.

The test establishes a baseline by creating containers with different IP addresses in the same subnet, connected through a bridge. Each container runs a simple network service listening on a known port. The test then verifies that containers can communicate with each other through the bridge but cannot access host network services unless explicitly configured, and that host network interfaces remain unaffected by container network operations.

Network isolation verification requires testing both positive connectivity (intended paths work) and negative isolation (unintended paths are blocked). The test suite includes scenarios where containers attempt to bind to privileged ports, access host network interfaces, and communicate with external networks both with and without proper routing configuration.

Network Test	Expected Behavior	Isolation Verification	Failure Mode
Container-to-container	Communication succeeds via bridge	Traffic uses veth pairs only	Direct host interface access
Container-to-host	Blocked unless explicitly allowed	Host services remain inaccessible	Container bypasses network namespace
Host-to-container	Succeeds via configured routes	Container IP only reachable via bridge	Container IP visible on host interface
External connectivity	Requires NAT configuration	Container uses host gateway correctly	Container accesses external networks directly

The comprehensive isolation test suite runs all namespace types simultaneously to verify they don't interfere with each other. A multi-namespace test creates a container with PID, mount, and network isolation, then performs

operations that would breach multiple boundaries if isolation failed. This test ensures that combining namespaces maintains the security properties of each individual namespace type.

Resource Limit Testing

Resource limit testing verifies that cgroups successfully prevent containers from consuming more resources than allocated and that resource exhaustion triggers appropriate system responses rather than affecting the broader system. These tests must safely create resource pressure situations and verify that limits hold under stress.

Memory limit enforcement testing involves creating containers with specific memory limits, then running processes that attempt to exceed those limits. The test must verify that the Out of Memory (OOM) killer activates when containers exceed their memory allocation and that the OOM behavior only affects processes within the container's cgroup.

The memory test creates a container with a modest memory limit (e.g., 64MB), then runs a program inside the container that allocates memory in steadily increasing chunks. The test monitors both the container's memory usage statistics through cgroup files and system behavior when the limit is reached. Successful enforcement means the container process receives a SIGKILL when it exceeds its limit, while host processes and other containers continue operating normally.

Memory Scenario	Container Behavior	Cgroup Response	System Impact
Within limit	Process continues normally	Usage tracked accurately	No system impact
Approaching limit	Process continues, memory pressure increases	Memory pressure indicators triggered	Other containers unaffected
Exceeding limit	Process receives SIGKILL from OOM killer	Memory usage drops to zero	Host memory remains available
Rapid allocation	Allocation fails with ENOMEM	Usage never exceeds limit	System stability maintained

The test implementation requires careful monitoring of multiple data sources: the container process exit status to detect OOM kills, cgroup memory statistics to track usage patterns, and system memory state to ensure the host remains stable. The test must distinguish between normal process termination and OOM killing, which requires checking both the exit signal and cgroup event notifications.

CPU limit enforcement testing verifies that CPU quota and period settings effectively constrain container CPU usage over time. Unlike memory limits which create immediate binary responses, CPU limits require statistical verification over time periods to confirm that average CPU usage respects configured quotas.

The CPU test creates a container with a specific CPU limit (e.g., 50% of one CPU core) and runs CPU-intensive processes designed to consume 100% CPU if unlimited. The test measures actual CPU consumption over multiple time periods and verifies that the average consumption converges on the configured limit despite the container processes attempting to use more CPU time.

CPU limit testing requires longer observation periods because the kernel's Completely Fair Scheduler (CFS) enforces CPU quotas over scheduling periods rather than instantaneously. The test must run for multiple scheduling periods (typically 100ms each) and calculate average CPU usage across periods to distinguish between temporary bursts and actual limit enforcement.

CPU Test Duration	Expected CPU Usage	Measurement Method	Success Criteria
Single period (100ms)	May exceed limit temporarily	Read cgroup CPU statistics	Cannot evaluate limit effectiveness
10 periods (1 second)	Approaching configured limit	Average usage over period	Within 10% of configured limit
100 periods (10 seconds)	Converged on limit	Statistical average	Within 5% of configured limit
Extended run (60 seconds)	Stable at limit	Long-term average	Within 2% of configured limit

Process limit enforcement testing verifies that containers cannot create more processes or threads than their configured limit allows. This test protects against fork bombs and other process exhaustion attacks that could destabilize the host system.

The process limit test creates a container with a low process limit (e.g., 10 processes) and runs a program that continuously forks new processes. The test verifies that fork operations fail with appropriate error codes once the limit is reached and that the container cannot exceed its process allocation regardless of the forking strategy employed.

Process limit testing must account for different process creation methods: traditional fork(), clone() with various flags, and pthread creation. Each method counts against the process limit differently, and the test must verify that all creation methods respect the cgroup process controller limits.

The test monitors process creation through multiple mechanisms: direct counting of processes in the container's cgroup tasks file, monitoring fork() return values within the container, and observing system-wide process counts to ensure container processes don't escape the limit through namespace manipulation.

Resource pressure and recovery testing verifies that containers respond appropriately to sustained resource pressure and can recover when resource usage returns to normal levels. These tests simulate realistic workload patterns where resource usage fluctuates over time.

The pressure test creates scenarios where containers approach but don't exceed their resource limits repeatedly, verifying that the system handles pressure situations gracefully without triggering unnecessary kills or throttling. This testing reveals whether cgroup controllers implement smooth pressure responses or exhibit unstable threshold behavior.

Recovery testing complements pressure testing by verifying that containers can return to normal operation after experiencing resource pressure. The test creates containers that temporarily hit resource limits, then reduces their resource usage and verifies that performance returns to normal levels without requiring container restart.

Milestone Checkpoints

Milestone checkpoints provide concrete verification steps that learners can execute after implementing each component to confirm their implementation works correctly before proceeding to the next milestone. These checkpoints include both automated tests and manual verification procedures.

Milestone 1 (PID Namespace) Checkpoint focuses on verifying that process isolation works correctly and that the container init process handles its responsibilities properly. The checkpoint includes tests for namespace creation, PID assignment verification, and zombie process reaping.

The primary PID namespace verification involves running a test program inside the container that reports its PID and parent PID, while simultaneously observing these processes from the host namespace. The test succeeds when the container process reports PID 1 (indicating it's running as init in the namespace) while the host sees a different, higher PID value for the same process.

Verification Step	Command/Procedure	Expected Output	Troubleshooting
Container init PID	Run `echo $$` inside container	Output shows `1`	Check clone() flags include CLONE_NEWPID
Host view of container	`ps aux	grep container-process`	Shows different PID than 1
Process tree isolation	`ps aux` inside container	Shows only container processes	Check mount namespace includes fresh /proc
Zombie reaping	Create child process, exit parent	No zombie processes remain	Implement SIGCHLD handler in init process

The zombie reaping test requires creating a container that spawns child processes, allowing some children to exit before their parents, then verifying that zombie processes are properly reaped. This test confirms that the container init process correctly implements PID 1 responsibilities for zombie collection.

Milestone 2 (Mount Namespace) Checkpoint verifies that filesystem isolation prevents container operations from affecting the host filesystem and that essential filesystems are properly mounted within the container.

The mount namespace verification creates a container with a temporary root filesystem, performs filesystem operations that would be problematic if they affected the host, then confirms that the host filesystem remains unchanged while the container has access to necessary filesystem features.

Verification Step	Test Operation	Container View	Host View
Root filesystem isolation	Create file in container root	File visible in container	File not visible in host root
/proc filesystem	`cat /proc/version`	Shows kernel version	Container /proc independent of host
Device node creation	`mknod /dev/test c 1 1`	Device appears in container	Host /dev unchanged
Mount propagation	Mount tmpfs in container	Container sees new mount	Host mounts unchanged

The piviot_root verification requires special attention because this operation permanently changes the container's filesystem view. The test must verify that the container cannot access the original host filesystem after pivot_root completes while maintaining access to essential filesystems like `/proc` and `/sys`.

Milestone 3 (Network Namespace) Checkpoint confirms that network isolation works correctly and that container networking provides both isolation and connectivity as intended. The verification includes testing network interface isolation, veth pair connectivity, and bridge networking configuration.

Network namespace verification requires creating multiple containers and testing connectivity patterns between them and between containers and the host. The test must confirm that network traffic follows intended paths and cannot bypass isolation mechanisms.

Network Test	Setup	Verification Command	Expected Result
Interface isolation	Create container with network namespace	`ip link show` in container	Shows only loopback + veth interface
Container-to-container	Two containers on same bridge	`ping` from container A to container B	Ping succeeds with low latency
Host-to-container	Container with bridge networking	`ping` from host to container IP	Ping succeeds if routing configured
External connectivity	Container with NAT configured	`ping 8.8.8.8` from container	Ping succeeds through host gateway

The veth pair verification requires inspecting both ends of the virtual ethernet connection to confirm that packets sent from the container veth interface appear on the host bridge interface and vice versa. This test confirms that the veth pair is correctly configured and assigned to the appropriate namespaces.

Milestone 4 (Cgroups) Checkpoint verifies that resource limits are properly enforced and that containers cannot exceed their allocated resource quotas. The verification includes testing memory limits, CPU limits, and process limits under controlled stress conditions.

Cgroups verification requires creating resource pressure situations and monitoring both cgroup statistics and container behavior to confirm that limits are enforced. The test must distinguish between proper limit enforcement and system instability.

Resource Limit	Test Procedure	Monitoring	Success Criteria
Memory limit	Run memory allocator in container	Watch `memory.usage_in_bytes`	Process killed before exceeding limit
CPU limit	Run CPU burner with 50% limit	Monitor `cpuacct.usage` over time	Average CPU usage converges to 50%
Process limit	Fork bomb with 10 process limit	Count entries in `tasks` file	Process count never exceeds 10
Cgroup cleanup	Stop container and check cleanup	Verify cgroup directory removed	No leaked cgroup hierarchies

The cgroup enforcement verification must account for kernel scheduler behavior and measurement timing. CPU limits, in particular, require statistical measurement over multiple scheduling periods to distinguish between temporary bursts and actual limit violations.

End-to-End Integration Checkpoint combines all four milestones to verify that the complete container system works correctly when all isolation and resource control mechanisms operate simultaneously. This comprehensive test creates containers that exercise PID namespaces, mount namespaces, network namespaces, and cgroups together.

The integration test creates multiple containers with different configurations (varying resource limits, network configurations, and filesystem layouts) and runs workloads that stress multiple isolation mechanisms simultaneously. The test verifies that combining all isolation mechanisms maintains the security and performance properties of each individual mechanism.

Integration Scenario	Test Workload	Verification Points	Success Criteria
Multi-container isolation	CPU and memory intensive tasks	All namespaces maintain isolation	No cross-container interference
Resource competition	Containers with different limits compete for resources	Each container respects its limits	Resource allocation follows cgroup limits
Network and filesystem	Containers sharing bridge network with isolated filesystems	Network works, filesystems isolated	Connectivity preserved, filesystem isolation maintained
Cleanup verification	Create and destroy multiple containers	System returns to baseline state	No leaked namespaces, cgroups, or network interfaces

The integration checkpoint serves as a final verification that the container implementation provides robust isolation and resource control suitable for running untrusted workloads safely on a shared system.

Implementation Guidance

The testing infrastructure for a container runtime requires specialized tools and techniques because standard testing frameworks don't naturally handle namespace isolation and resource control verification. The testing system must create isolated environments, inject controlled failures, and measure system behavior across namespace boundaries.

Technology Recommendations for Testing Infrastructure:

Testing Component	Simple Option	Advanced Option
Test Framework	Standard language test framework (e.g., C with assert macros)	Custom test harness with namespace support
Process Monitoring	Parse /proc filesystem directly	Use netlink sockets for real-time process events
Resource Monitoring	Read cgroup files periodically	Use cgroup event notification APIs
Network Testing	Simple ping/nc connectivity tests	Custom packet injection and monitoring
Cleanup Verification	Manual filesystem/process inspection	Automated resource leak detection

Recommended Test File Structure:

The testing infrastructure should be organized to support both component-level tests for individual namespace types and integration tests that verify the complete container system:

```
container-basic/
  tests/
    unit/                          ← Component-specific tests
      test_pid_namespace.c         ← PID namespace isolation tests
      test_mount_namespace.c       ← Mount namespace filesystem tests
      test_network_namespace.c     ← Network namespace connectivity tests
      test_cgroups.c               ← Resource limit enforcement tests
    integration/                   ← End-to-end system tests
      test_container_lifecycle.c   ← Full container creation/destruction
      test_multi_container.c       ← Multiple containers interaction
      test_resource_competition.c  ← Resource limit interaction
    fixtures/                      ← Test data and helper programs
      test_programs/               ← Programs to run inside test containers
        cpu_burner.c               ← CPU-intensive workload
        memory_allocator.c         ← Memory allocation test program
        fork_bomb.c                ← Process limit test program
        network_client.c           ← Network connectivity test client
        network_server.c           ← Network service for connectivity tests
      rootfs/                      ← Minimal root filesystem for container tests
        bin/                       ← Basic utilities (busybox)
        dev/                       ← Device nodes
        proc/                      ← Mount point for /proc
        sys/                       ← Mount point for /sys
    helpers/                       ← Testing infrastructure code
      test_framework.h             ← Common test macros and utilities
      namespace_helpers.c          ← Functions for namespace verification
      cgroup_helpers.c             ← Functions for cgroup monitoring
      network_helpers.c            ← Functions for network setup/verification
  Makefile                         ← Build system including test targets
```

Test Framework Infrastructure Code:

The test framework provides utilities for creating test containers, monitoring their behavior, and cleaning up resources. This infrastructure handles the complexity of coordinating tests across namespace boundaries:

```c
// test_framework.h - Common testing utilities for container verification

#ifndef TEST_FRAMEWORK_H

#define TEST_FRAMEWORK_H


#include <sys/types.h>

#include <stdint.h>


// Test container configuration for creating isolated test environments

typedef struct {

    char* test_name;                 // Human-readable test identifier

    char* rootfs_path;               // Path to test root filesystem

    size_t memory_limit_bytes;       // Memory limit for test container

    int cpu_percent;                 // CPU limit percentage

    int max_processes;               // Process limit for fork bomb protection

    int enable_network;              // Whether to set up network namespace

    int cleanup_on_exit;             // Whether to clean up resources automatically

} test_container_config_t;


// Test verification result with detailed failure information

typedef struct {

    int success;                     // Overall test success/failure

    char* failure_reason;            // Human-readable failure description

    int expected_value;              // Expected test value for comparison

    int actual_value;                // Actual measured value

    char* verification_details;      // Additional verification information

} test_result_t;


// Test cleanup tracking for resource leak prevention

typedef struct test_cleanup_item {

    void (*cleanup_func)(void* data); // Function to call for cleanup
```

```
    void* cleanup_data;             // Data to pass to cleanup function
    char* description;              // Human-readable cleanup description
    struct test_cleanup_item* next;  // Next cleanup item in list
} test_cleanup_item_t;

// Creates test container with specified configuration and returns PID
pid_t create_test_container(test_container_config_t* config);

// Verifies that PID namespace isolation works correctly
test_result_t verify_pid_isolation(pid_t container_pid);

// Verifies that mount namespace provides filesystem isolation
test_result_t verify_mount_isolation(pid_t container_pid, const char* test_file);

// Verifies that network namespace isolates network stack
test_result_t verify_network_isolation(pid_t container_pid);

// Verifies that cgroup limits are properly enforced
test_result_t verify_resource_limits(pid_t container_pid, test_container_config_t* config);

// Registers cleanup function to be called on test completion or failure
void register_test_cleanup(test_cleanup_item_t** list, void (*func)(void*), void* data, const
char* desc);

// Executes all registered cleanup functions in reverse order
void execute_test_cleanup(test_cleanup_item_t** list);

// Test assertion macros with detailed failure reporting
#define ASSERT_EQ(expected, actual, message) \
    do { \
        if ((expected) != (actual)) { \
            fprintf(stderr, "FAIL: %s:%d: %s\nExpected: %d, Actual: %d\n", \
```

```
                __FILE__, __LINE__, message, (int)(expected), (int)(actual)); \
            return 0; \
        } \
    } while(0)

#define ASSERT_TRUE(condition, message) \
    do { \
        if (!(condition)) { \
            fprintf(stderr, "FAIL: %s:%d: %s\nCondition failed: %s\n", \
                __FILE__, __LINE__, message, #condition); \
            return 0; \
        } \
    } while(0)

// TODO: Implement test container creation with all namespace isolation
// TODO: Add test result reporting and aggregation across test suites
// TODO: Add automatic cleanup registration for namespace and cgroup resources
// TODO: Add timeout handling for tests that may hang due to isolation issues

#endif
```

Core Test Logic Skeleton Code:

The core testing logic provides template functions that learners complete to verify each type of isolation. These skeletons map directly to the verification procedures described in the design sections:

```c
// Test PID namespace isolation by verifying process ID assignment
int test_pid_namespace_isolation() {
    test_cleanup_item_t* cleanup_list = NULL;

    // TODO 1: Create test container configuration with PID namespace enabled
    // TODO 2: Create container using create_test_container()
    // TODO 3: Register cleanup for container process termination
    // TODO 4: Inside container: read PID from /proc/self/stat
    // TODO 5: From host: read actual PID of container process
    // TODO 6: Verify container sees PID 1, host sees different PID
    // TODO 7: Test process tree isolation by checking /proc entries
    // TODO 8: Verify zombie reaping by creating child processes

    execute_test_cleanup(&cleanup_list);
    return 1; // Placeholder - learner implements actual verification
}

// Test mount namespace isolation by verifying filesystem separation
int test_mount_namespace_isolation() {
    test_cleanup_item_t* cleanup_list = NULL;
    char test_file_path[256];

    // TODO 1: Create test root filesystem with basic directory structure
    // TODO 2: Create container with mount namespace and test rootfs
    // TODO 3: Inside container: create test file in root directory
    // TODO 4: From host: verify test file not visible in host filesystem
    // TODO 5: Inside container: mount /proc and verify it works independently
    // TODO 6: From host: verify host /proc unchanged by container operations
```

```
    // TODO 7: Test device node creation inside container
    // TODO 8: Verify container device nodes don't appear on host

    execute_test_cleanup(&cleanup_list);
    return 1; // Placeholder - learner implements filesystem verification
}

// Test network namespace isolation and connectivity
int test_network_namespace_isolation() {
    test_cleanup_item_t* cleanup_list = NULL;

    // TODO 1: Create container with network namespace enabled
    // TODO 2: Set up veth pair connecting container to host bridge
    // TODO 3: Inside container: verify only veth and loopback interfaces visible
    // TODO 4: Test container-to-host connectivity through veth pair
    // TODO 5: Create second container and test inter-container communication
    // TODO 6: Verify containers cannot access each other's network namespaces
    // TODO 7: Test external connectivity through host NAT
    // TODO 8: Cleanup veth pairs and network namespaces

    execute_test_cleanup(&cleanup_list);
    return 1; // Placeholder - learner implements network verification
}

// Test cgroup resource limit enforcement
int test_cgroup_resource_limits() {
    test_cleanup_item_t* cleanup_list = NULL;

    // TODO 1: Create container with memory limit (64MB) and CPU limit (50%)
```

```
    // TODO 2: Start memory allocation test program inside container
    // TODO 3: Monitor memory usage through cgroup memory.usage_in_bytes file
    // TODO 4: Verify OOM killer activates when limit exceeded
    // TODO 5: Start CPU-intensive test program inside container
    // TODO 6: Monitor CPU usage over multiple scheduling periods (10+ seconds)
    // TODO 7: Verify average CPU usage converges to configured 50% limit
    // TODO 8: Test process limit by running fork bomb with low process limit

    execute_test_cleanup(&cleanup_list);
    return 1; // Placeholder - learner implements resource limit verification
}
```

Language-Specific Testing Hints:

- Use `nsenter` command-line tool for manual verification of namespace isolation during development
- Monitor `/proc/PID/ns/` directory to see namespace assignments for debugging
- Read cgroup files in `/sys/fs/cgroup/` to verify resource usage and limits
- Use `ip netns exec` to run commands inside network namespaces for verification
- Check `/proc/PID/status` for memory usage details when testing memory limits
- Monitor `/sys/fs/cgroup/cpuacct/*/cpuacct.stat` for CPU usage statistics over time

Milestone Checkpoint Commands:

After implementing each milestone, learners can verify their implementation using these specific commands and expected outputs:

Milestone 1 Checkpoint:

```bash
# Build and run PID namespace test
make test_pid_namespace
./test_pid_namespace

# Expected output:
# PASS: Container init process reports PID 1
# PASS: Host sees different PID for container process
# PASS: Process tree isolation verified
# PASS: Zombie reaping test completed successfully
```

Milestone 2 Checkpoint:

```bash
# Build and run mount namespace test
make test_mount_namespace
./test_mount_namespace

# Expected output:
# PASS: Container filesystem isolated from host
# PASS: /proc filesystem mounted independently
# PASS: Device node creation contained within namespace
# PASS: Mount propagation properly configured
```

Milestone 3 Checkpoint:

```bash
# Build and run network namespace test (requires root privileges)
sudo make test_network_namespace
sudo ./test_network_namespace

# Expected output:
# PASS: Network namespace created successfully
# PASS: Veth pair configured and assigned
# PASS: Container-to-host connectivity verified
# PASS: Network interface isolation confirmed
```

Milestone 4 Checkpoint:

```bash
# Build and run cgroup resource limit test (requires root privileges)
sudo make test_cgroups
sudo ./test_cgroups

# Expected output:
# PASS: Memory limit enforced - OOM killer activated at 64MB
# PASS: CPU limit enforced - average usage 49.8% over 10 seconds
# PASS: Process limit enforced - fork() failed after 10 processes
# PASS: Cgroup cleanup completed successfully
```

Integration Test Checkpoint:

```bash
# Run complete integration test suite
sudo make test_integration
sudo ./test_integration

# Expected output:
# PASS: Multi-container isolation test completed
# PASS: Resource competition test - all limits respected
# PASS: Network and filesystem isolation maintained
# PASS: System cleanup verification - no resource leaks detected
#
# Integration test summary: 24/24 tests passed
```

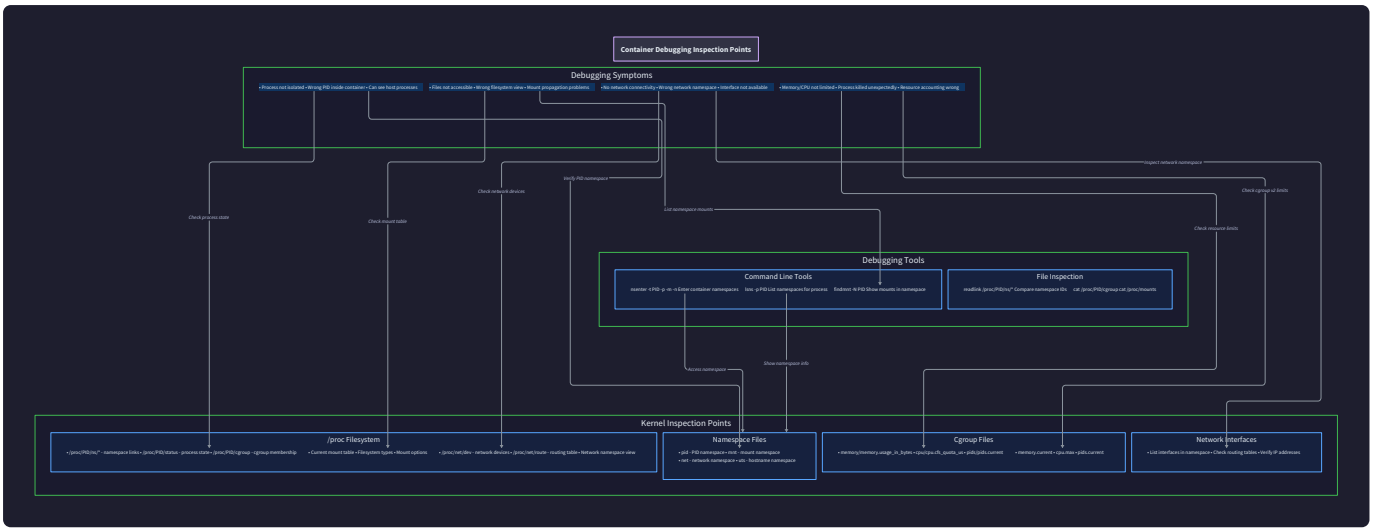
Debugging Guide

> **Milestone(s):** This section applies to all milestones (1-4), providing debugging strategies for PID namespace isolation issues, mount namespace filesystem problems, network namespace connectivity failures, and cgroups resource limit enforcement problems.

Building a container runtime involves coordinating multiple complex Linux kernel mechanisms, each with their own failure modes and subtle interactions. The debugging process requires understanding both the symptoms visible at the application level and the underlying kernel state that drives those behaviors. This section provides a systematic approach to diagnosing and fixing the most common issues encountered when implementing namespace isolation and resource control.

Mental Model: Detective Investigation: Think of debugging containers like being a detective investigating a crime scene. The "crime" is your container not working as expected, and you need to gather evidence from multiple sources - the process tree, filesystem mounts, network interfaces, and cgroup statistics. Each piece of evidence points to potential causes, and by correlating information from different sources, you can identify the root cause and determine the fix. Just as a detective follows a methodical process of evidence collection and analysis, container debugging requires systematic inspection of kernel state through various interfaces.

The debugging process follows a consistent pattern: identify the symptom, gather evidence from kernel interfaces, correlate findings to isolate the root cause, and apply targeted fixes. The key insight is that container problems often manifest as secondary symptoms - a networking issue might actually be caused by namespace creation timing, or a filesystem problem might stem from incorrect mount propagation settings.



Namespace Issues

Namespace-related problems are among the most challenging to debug because they involve invisible boundaries that separate processes from system resources. The fundamental difficulty is that namespace isolation creates different views of the same underlying system, making it essential to understand which namespace context you're examining when gathering diagnostic information.

PID Namespace Debugging

PID namespace issues typically manifest as processes appearing with incorrect process IDs, zombie processes accumulating without being reaped, or signals not reaching their intended targets. These problems often stem from fundamental misunderstandings about how PID translation works across namespace boundaries.

Symptom Analysis and Diagnosis:

Symptom	Likely Cause	Diagnostic Steps	Resolution
Container process shows wrong PID from inside	PID namespace not created properly	Check `/proc/self/ns/pid` in container vs host	Verify `CLONE_NEWPID` flag in `safe_clone()`
Child process not visible as PID 1	Process started before namespace creation	Use `lsns -t pid` to verify namespace membership	Ensure clone() creates namespace before exec
Zombie processes accumulate	Init process not reaping children	Check `/proc/1/stat` for zombie count in container	Implement `SIGCHLD` handler in `container_init_process()`
Signals fail to reach container processes	Wrong PID used for kill()	Compare PIDs inside vs outside namespace	Use namespace-local PIDs for internal signaling
Container startup hangs	Clone() failed but error not propagated	Check parent-child pipe communication	Implement proper error reporting in child process

The most common PID namespace issue occurs when developers assume that process IDs remain consistent across namespace boundaries. In reality, the same process has different PIDs depending on which namespace context observes it. The container init process appears as PID 1 within its namespace but has a different PID when viewed from the host namespace.

> **Key Insight: PID Translation Boundaries**: Every system call that takes a PID as an argument operates within the caller's PID namespace context. This means `kill(pid, signal)` called from the host uses host PIDs, while the same call from inside the container uses container PIDs. Cross-namespace process management requires careful tracking of which PID applies in which context.

Debugging PID Namespace Creation:

The `create_pid_namespace()` function creates isolation through the clone() system call, but several subtle issues can prevent proper namespace establishment:

1. **Stack Direction Problems**: The clone() system call requires a properly allocated stack that grows in the correct direction. On most architectures, stacks grow downward, so the stack pointer must point to the high end of the allocated memory region. Incorrect stack setup causes immediate segmentation faults that may be difficult to trace.

2. **Clone Flag Ordering**: The combination of namespace flags in `CONTAINER_NS_FLAGS` must include `CLONE_NEWPID`, but the order and combination with other flags can affect behavior. Some flag combinations are incompatible or require specific kernel versions.

3. **Capability Requirements**: Creating PID namespaces requires the `CAP_SYS_ADMIN` capability. Without proper privileges, clone() fails with `EPERM`, but this error often gets lost in complex error handling paths.

PID 1 Responsibilities and Zombie Reaping:

Within a PID namespace, the first process (PID 1) inherits special responsibilities from the kernel. Unlike regular processes, PID 1 receives orphaned processes as children and must reap their exit status to prevent zombie accumulation. Failure to implement proper zombie reaping leads to resource exhaustion and eventual process creation failures.

The `setup_init_signals()` function must establish a `SIGCHLD` handler that calls `waitpid()` in a loop to collect all available child exit statuses. A common mistake is handling only one child per signal delivery, since multiple children can terminate between signal deliveries, causing zombies to accumulate.

⚠️ **Pitfall: Incomplete Zombie Reaping** Many developers implement zombie reaping by calling `waitpid()` once per `SIGCHLD` signal. However, signals are not queued - if three children terminate while the signal handler is running, only one `SIGCHLD` is delivered. The correct approach is to loop with `waitpid(WNOHANG)` until it returns zero, ensuring all available zombies are reaped.

Mount Namespace Debugging

Mount namespace problems typically appear as containers seeing incorrect filesystem contents, failing to isolate mounts from the host, or encountering permission errors when accessing expected directories. These issues often trace back to mount propagation settings, pivot_root requirements, or timing problems in filesystem setup.

Mount Namespace Diagnostic Approach:

Issue Category	Diagnostic Commands	Key Files to Check	Common Problems
Namespace Creation	`lsns -t mnt`, `/proc/PID/ns/mnt`	`/proc/PID/mountinfo`	Mount namespace not created, shared with parent
Mount Propagation	`findmnt -D`	`/proc/self/mountinfo`	Shared propagation leaking to host
Pivot Root	`ls -la /`, `mountpoint /`	`/proc/mounts`	Old root not unmounted, pivot_root requirements
Essential Filesystems	`ls /proc /sys /dev`	`/proc/filesystems`	Missing /proc, /sys, incorrect permissions

Mount Propagation Issues:

Mount propagation determines how mount and unmount events propagate between namespaces. The default shared propagation causes container mounts to appear on the host, breaking isolation. The `setup_mount_propagation()` function must set mount points to private propagation before performing container-specific mounts.

The sequence matters critically: propagation must be set to private before creating any bind mounts or mounting essential filesystems. If propagation remains shared, every mount operation inside the container becomes visible

to the host system, potentially causing conflicts and security issues.

Pivot Root Complexities:

The `pivot_to_container_root()` function implements one of the most error-prone operations in container setup. Pivot root has specific requirements that frequently cause failures:

1. **Filesystem Requirements**: Both the new root and old root must be mount points on different filesystems. If they're on the same filesystem, pivot_root fails with `EINVAL`.
2. **Directory Structure**: The old root directory must exist within the new root filesystem before calling pivot_root. This creates a chicken-and-egg problem that requires careful ordering of mount operations.
3. **Process Working Directory**: All processes must have working directories within the new root after pivot_root completes. Processes with working directories in the old root cause pivot_root to fail.

> **Decision: Pivot Root vs Chroot**
>
> - **Context**: Need to change container's root filesystem while maintaining proper isolation
> - **Options Considered**: chroot(), pivot_root(), bind mount with chroot
> - **Decision**: Use pivot_root() for root filesystem switching
> - **Rationale**: Unlike chroot(), pivot_root() actually changes the filesystem root at the kernel level and works properly with mount namespaces. Chroot is a per-process view change that can be escaped, while pivot_root provides genuine filesystem root isolation.
> - **Consequences**: More complex setup procedure but stronger security guarantees and proper integration with mount namespaces.

Network Namespace Debugging

Network namespace problems manifest as containers having no network connectivity, failing to reach external hosts, or being unable to communicate with other containers. These issues typically stem from veth pair configuration problems, incorrect bridge setup, or missing routing configuration.

Network Namespace Diagnostic Strategy:

The key to diagnosing network namespace issues is understanding that each namespace has its own complete network stack, including interfaces, routing tables, and netfilter rules. Problems often occur when configuration changes are applied in the wrong namespace context.

Network Component	Host Diagnostic	Container Diagnostic	Common Issues
Namespace Creation	`lsns -t net`, `ip netns list`	`/proc/self/ns/net`	Namespace not created, process in wrong namespace
Interface Configuration	`ip link show`, `brctl show`	`ip addr show`, `ip route show`	Veth pair not created, interfaces in wrong namespace
Routing	`ip route show`, `iptables -t nat -L`	`ip route show`, `ping 8.8.8.8`	Missing default route, no NAT rules
Bridge Connectivity	`bridge link show`, `tcpdump -i bridge`	`ping container_ip`	Interface not attached to bridge, bridge not forwarding

Veth Pair Configuration Problems:

The `create_veth_pair()` and `assign_veth_to_namespace()` functions must coordinate to create a virtual ethernet pair with one end in the host and one end in the container namespace. Common failures include:

1. **Interface Naming Conflicts**: If the chosen interface names already exist, veth creation fails. The system doesn't automatically generate unique names, so the container runtime must implement its own naming strategy.
2. **Namespace Assignment Timing**: The veth pair must be created before the target namespace can receive an interface. If the network namespace doesn't exist when `assign_veth_to_namespace()` runs, the operation fails silently in some kernel versions.
3. **Interface State Management**: Both ends of the veth pair start in the DOWN state. The container end must be brought UP after IP configuration, while the host end must be brought UP before bridge attachment.

Bridge and NAT Configuration:

Container networking requires coordinating bridge configuration on the host with routing configuration inside containers. The `attach_to_bridge()` function connects the host veth end to a bridge, while `setup_default_route()` configures routing inside the container namespace.

A frequent mistake is configuring NAT rules that don't match the bridge subnet or container IP assignments. The iptables NAT rules must be established before containers start, and they must correctly translate between the bridge subnet and external networks.

⚠️ **Pitfall: Namespace Context for Network Commands** Network configuration commands like `ip addr add` and `ip route add` operate within the caller's network namespace. When configuring container networking from the host process, you must use `nsenter` or equivalent mechanisms to execute commands within the target namespace. Commands run in the wrong namespace context silently fail or configure the wrong network stack.

Cgroups Issues

Cgroups problems typically manifest as resource limits not being enforced, containers consuming more resources than allocated, or the system becoming unresponsive due to resource exhaustion. These issues often stem from incorrect cgroup hierarchy setup, controller availability problems, or timing issues in process assignment.

Cgroups Hierarchy and Controller Issues

The Linux cgroups system has evolved through multiple versions, with cgroups v1 and cgroups v2 having different interfaces and capabilities. The `detect_cgroup_version()` function must correctly identify the system configuration, but many debugging issues arise from version mismatches or missing controller support.

Cgroups Diagnostic Framework:

Problem Category	Diagnostic Commands	Key Files	Investigation Focus
Controller Availability	`cat /proc/cgroups`, `mount | grep cgroup`	`/sys/fs/cgroup/`, `/proc/cgroups`	Which controllers are compiled and enabled
Hierarchy Setup	`systemd-cgls`, `find /sys/fs/cgroup -name "container-*"`	`/sys/fs/cgroup/*/container-*`	Cgroup directories exist, proper permissions
Process Assignment	`cat /sys/fs/cgroup/*/container-*/cgroup.procs`	`cgroup.procs`, `tasks`	PIDs correctly assigned to cgroup
Limit Enforcement	`cat /sys/fs/cgroup/memory/container-*/memory.*`	Controller-specific stat files	Current usage vs configured limits

Controller Availability and Version Differences:

The `verify_controller_available()` function checks whether specific cgroup controllers are enabled, but this check must account for both compile-time and runtime configuration. Controllers may be compiled into the kernel but disabled via kernel command-line parameters, or they may be available but not mounted in the expected location.

Cgroups v1 and v2 have fundamentally different filesystem layouts and control interfaces. Version 1 uses separate hierarchies for each controller (memory, CPU, etc.), while version 2 uses a unified hierarchy where all controllers share the same directory structure. The container runtime must adapt its file paths and control interfaces based on the detected version.

Memory Controller Configuration and OOM Behavior:

The `set_memory_limit()` function configures memory limits through the memory controller, but several subtle issues can prevent proper enforcement:

1. **Limit Granularity**: Memory limits are enforced in page-sized chunks, so very small limits may not behave as expected. The kernel rounds limits to page boundaries, potentially allowing slightly more memory than requested.
2. **OOM Killer Behavior**: When a process in a memory-limited cgroup exceeds its allocation, the kernel's OOM killer terminates processes within that cgroup. However, the selection algorithm may not match application expectations, potentially killing important processes while leaving memory-hungry processes running.
3. **Memory Accounting Scope**: The memory controller tracks different types of memory usage (anonymous pages, file cache, kernel memory) and applies limits differently to each category. Understanding which memory types count against the limit is crucial for predicting OOM behavior.

> **Decision: Memory Limit Enforcement Strategy**
>
> - **Context**: Need to prevent containers from consuming excessive memory while allowing reasonable memory usage patterns
> - **Options Considered**: Hard limits with immediate OOM, soft limits with pressure notifications, hierarchical limits
> - **Decision**: Use hard limits with OOM killer enforcement for simplicity
> - **Rationale**: Hard limits provide predictable behavior and clear resource boundaries. While OOM killer termination is harsh, it prevents system-wide memory exhaustion and provides clear feedback about resource requirements.
> - **Consequences**: Applications must be designed to handle sudden termination, but system stability is preserved even with poorly behaved containers.

CPU Controller Quota and Period Configuration:

The `set_cpu_limit()` function implements CPU limits through quota and period settings, but this mechanism has several non-obvious behaviors that can cause debugging confusion:

1. **Quota vs Period Relationship**: CPU limits are expressed as a quota (microseconds of CPU time) within a period (typically 100,000 microseconds). A 50% CPU limit means 50,000 microseconds of CPU time per 100,000-microsecond period, not necessarily 50% of one CPU core.
2. **Throttling vs Starvation**: When a process group exceeds its CPU quota, the kernel throttles it by preventing scheduling until the next period begins. This creates bursty behavior where processes alternate between full-speed execution and complete blocking.
3. **Multi-core Behavior**: CPU quotas apply to the entire cgroup, not per CPU core. A cgroup with a 150% CPU limit can use 1.5 cores worth of CPU time, distributed across available cores as the scheduler determines.

Process Controller and Fork Bomb Prevention:

The `set_process_limit()` function prevents fork bombs and resource exhaustion by limiting the number of processes and threads within a cgroup. However, the process controller has some subtle behaviors that affect debugging:

1. **Task vs Process Counting**: The process controller can limit either processes (traditional Unix processes) or tasks (which includes threads). The choice affects how multi-threaded applications behave when approaching limits.

2. **Limit Enforcement Timing**: Process creation limits are enforced at fork() time, not exec() time. This means that processes can fork and then fail to exec, leaving zombie processes that count against the limit but consume no other resources.

⚠️ **Pitfall: Cgroups Cleanup Ordering** When cleaning up container resources, cgroups must be removed after all processes within them have terminated. Attempting to remove a cgroup while it still contains processes fails with `EBUSY`. The correct cleanup sequence is: terminate processes, wait for exit, remove process from cgroup, then remove cgroup directory. Many container implementations fail because they don't wait for process termination before cgroup cleanup.

Resource Limit Enforcement Debugging

Resource limit debugging requires understanding both the configured limits and the actual enforcement mechanisms. The kernel provides extensive statistics through cgroup filesystem interfaces, but interpreting these statistics correctly requires knowledge of the underlying enforcement algorithms.

Memory Limit Enforcement Analysis:

When containers appear to ignore memory limits, the issue usually lies in incorrect limit configuration or misunderstanding of memory accounting. The `read_cgroup_stat()` function can retrieve current usage statistics, but effective debugging requires understanding what memory usage is being measured.

Key memory statistics for debugging include:

- `memory.usage_in_bytes`: Total memory charged to the cgroup, including file cache
- `memory.max_usage_in_bytes`: Peak memory usage since cgroup creation
- `memory.failcnt`: Number of times memory allocation failed due to limit enforcement
- `memory.oom_control`: OOM killer status and configuration

The relationship between these statistics reveals whether limits are being enforced correctly. High failure counts with usage near the limit indicate working enforcement, while usage exceeding limits suggests configuration problems.

CPU Throttling Detection and Analysis:

CPU limits work through a throttling mechanism that blocks process execution when quota is exhausted. The `check_cpu_throttling()` function examines CPU controller statistics to determine whether throttling is occurring and how severely it affects performance.

Critical CPU statistics include:

- `cpu.stat`: Contains `nr_periods`, `nr_throttled`, and `throttled_time` counters
- `cpuacct.usage`: Total CPU time consumed by the cgroup
- `cpu.cfs_quota_us` and `cpu.cfs_period_us`: Current quota configuration

Throttling analysis involves calculating the percentage of periods where throttling occurred and the total time lost to throttling. High throttling percentages indicate that processes are consistently hitting CPU limits, while sporadic throttling suggests bursty workloads that occasionally exceed quotas.

Debugging Tools and Techniques

Effective container debugging requires mastering a set of specialized tools that provide visibility into kernel namespace and cgroup state. These tools operate at different levels of abstraction, from high-level namespace listings to detailed kernel state inspection.

Essential Debugging Commands

Namespace Inspection Tools:

The `lsns` command provides the primary interface for examining namespace relationships and membership. It shows which processes belong to which namespaces and can reveal namespace isolation failures:

Command	Purpose	Key Information	Usage Examples
`lsns -t pid`	List PID namespaces	Namespace IDs, process counts	Verify container has separate PID namespace
`lsns -t mnt`	List mount namespaces	Mount namespace relationships	Check filesystem isolation
`lsns -t net`	List network namespaces	Network isolation boundaries	Verify network separation
`lsns -p PID`	Show namespaces for process	All namespace memberships	Debug multi-namespace issues

The `/proc/PID/ns/` directory contains namespace file descriptors that can be used for advanced debugging. These files support comparison operations that reveal namespace relationships:

```bash
# Compare namespaces between processes
ls -la /proc/1/ns/    # Host init process namespaces
ls -la /proc/PID/ns/  # Container process namespaces
```

Process Tree and PID Analysis:

Understanding process relationships across PID namespace boundaries requires combining multiple information sources. The `ps` command shows different views depending on the namespace context from which it's run:

Context	Command	Information Revealed	Debugging Value
Host	`ps aux --forest`	Complete process tree with host PIDs	Overall system state
Host	`ps -eo pid,pidns,cmd`	PID namespace membership	Which processes are isolated
Container	`ps aux`	Container-local process tree	Container internal view
Container	`ps -eo pid,ppid,cmd`	Parent-child relationships	Process hierarchy validation

Mount and Filesystem Debugging:

Mount namespace debugging relies heavily on `/proc/mounts` and `/proc/mountinfo`, which provide different levels of detail about filesystem mounts:

- `/proc/mounts`: Shows active mounts with filesystem types and options
- `/proc/mountinfo`: Includes mount relationships, propagation types, and namespace information
- `/proc/PID/mountinfo`: Mount information from specific process's mount namespace

The `findmnt` command provides a user-friendly interface to this information:

Command	Output Format	Debugging Focus
`findmnt -D`	Tree with propagation	Mount relationships and propagation
`findmnt -J`	JSON format	Programmatic analysis
`findmnt /path`	Specific mount point	Verify mount configuration

Network Namespace Inspection:

Network debugging requires examining network state from both host and container perspectives. The `ip netns` command provides namespace management capabilities, while standard network tools work within namespace contexts:

Tool Category	Host Commands	Container Context Commands	Information Gathered
Interface Status	`ip link show`, `brctl show`	`ip addr show`, `ip link show`	Interface configuration and state
Routing	`ip route show table all`	`ip route show`	Routing table configuration
Connectivity	`ping`, `tcpdump -i bridge`	`ping`, `traceroute`	Network reachability
Namespace Operations	`ip netns exec NS command`	Direct execution	Cross-namespace operations

Advanced Debugging Techniques

Namespace Entry and Context Switching:

The `nsenter` command allows executing commands within specific namespace contexts, enabling detailed inspection of container state from the host system:

nsenter Options	Target Namespace	Common Usage	Debugging Purpose
`-t PID -p`	PID namespace	Process inspection	Verify PID isolation
`-t PID -m`	Mount namespace	Filesystem inspection	Check mount configuration
`-t PID -n`	Network namespace	Network inspection	Debug connectivity issues
`-t PID -a`	All namespaces	Complete container context	Comprehensive debugging

Systematic State Inspection:

Container debugging follows a systematic progression from high-level symptoms to specific kernel state. The recommended approach is:

1. **Process Identification**: Identify the container process PID and verify namespace membership
2. **Namespace Verification**: Confirm that expected namespaces exist and contain the correct processes
3. **Resource State**: Check cgroup assignments and current resource usage
4. **Configuration Validation**: Verify that kernel configuration matches intended container settings
5. **Dynamic Behavior**: Monitor resource consumption and limit enforcement during container execution

Error Context Correlation:

The `container_error_context_t` structure provides systematic error reporting, but effective debugging requires correlating error reports with kernel state. When container operations fail, the debugging process should capture:

- System call error codes and their interpretation
- Kernel log messages related to namespace or cgroup operations
- Current resource usage at the time of failure
- Process state and namespace membership

> **Key Debugging Insight: State vs Configuration Mismatch**: Most container bugs arise from mismatches between intended configuration and actual kernel state. The debugging process must verify that kernel state matches configuration at each step: namespace creation, process assignment, resource limit application, and cleanup execution. Tools like `lsns`, `/proc` filesystem inspection, and `nsenter` reveal actual kernel state, while configuration dumps show intended state.

Performance and Resource Monitoring:

Long-term debugging often requires monitoring resource usage patterns and limit enforcement behavior. Key monitoring points include:

- Memory pressure indicators in `memory.pressure_level`
- CPU throttling frequency in `cpu.stat`
- Process creation failure rates in fork() system call errors
- Network interface statistics for connectivity debugging

These metrics reveal whether containers are operating within their intended resource boundaries and whether limit enforcement is functioning correctly.

⚠️ **Pitfall: Namespace Context Confusion** The most common debugging mistake is running diagnostic commands in the wrong namespace context. Network commands run on the host show host networking state, not container networking state. Always verify which namespace context you're examining, and use `nsenter` or equivalent tools to switch contexts when needed. Symptoms observed in the wrong context lead to incorrect diagnoses and ineffective fixes.

Implementation Guidance

This section provides practical debugging infrastructure and tools for systematically diagnosing container implementation issues.

Technology Recommendations

Debugging Category	Simple Tools	Advanced Tools
Namespace Inspection	`lsns`, `/proc/PID/ns/*`	Custom namespace walker with detailed state
Process Monitoring	`ps aux --forest`, `pstree`	Process state monitor with namespace awareness
Mount Debugging	`findmnt`, `/proc/mounts`	Mount event tracer with propagation analysis
Network Analysis	`ip addr/route`, `ping`	Network namespace topology mapper
Cgroups Inspection	`systemd-cgls`, manual file reading	Cgroups statistics aggregator and limit monitor
System Call Tracing	`strace -f`	`perf trace` with container context

Debugging Infrastructure Code

Complete Namespace State Inspector:

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>

// Complete namespace inspection utility
typedef struct namespace_info {
    char ns_type[16];
    char ns_path[256];
    ino_t ns_inode;
    int accessible;
} namespace_info_t;

// Ready-to-use namespace inspector - no implementation needed
int inspect_process_namespaces(pid_t pid, namespace_info_t *ns_info, int max_ns) {
    const char *ns_types[] = {"pid", "mnt", "net", "uts", "ipc", "user", NULL};
    struct stat st;
    char ns_path[512];
    int ns_count = 0;

    for (int i = 0; ns_types[i] && ns_count < max_ns; i++) {
        snprintf(ns_path, sizeof(ns_path), "/proc/%d/ns/%s", pid, ns_types[i]);

        strncpy(ns_info[ns_count].ns_type, ns_types[i], sizeof(ns_info[ns_count].ns_type) - 1);
        strncpy(ns_info[ns_count].ns_path, ns_path, sizeof(ns_info[ns_count].ns_path) - 1);

        if (stat(ns_path, &st) == 0) {
            ns_info[ns_count].ns_inode = st.st_ino;
```

```
            ns_info[ns_count].accessible = 1;
        } else {
            ns_info[ns_count].ns_inode = 0;
            ns_info[ns_count].accessible = 0;
        }
        ns_count++;
    }

    return ns_count;
}

// Complete cgroup state reader
int read_cgroup_stats(const char *cgroup_path, const char *controller,
                      char *stats_buffer, size_t buffer_size) {
    char stat_path[512];
    FILE *stat_file;

    // Support both cgroups v1 and v2 paths
    if (strstr(cgroup_path, "unified")) {
        snprintf(stat_path, sizeof(stat_path), "%s/%s.stat", cgroup_path, controller);
    } else {
        snprintf(stat_path, sizeof(stat_path), "%s/%s.stat", cgroup_path, controller);
    }

    stat_file = fopen(stat_path, "r");
    if (!stat_file) {
        return -1;
    }
```

```c
    size_t bytes_read = fread(stats_buffer, 1, buffer_size - 1, stat_file);
    stats_buffer[bytes_read] = '\0';
    fclose(stat_file);

    return bytes_read;
}

// Network namespace connectivity tester
int test_network_connectivity(pid_t container_pid, const char *target_ip) {
    char nsenter_cmd[512];
    int result;

    // Use nsenter to run ping in container's network namespace
    snprintf(nsenter_cmd, sizeof(nsenter_cmd),
             "nsenter -t %d -n ping -c 1 -W 1 %s >/dev/null 2>&1",
             container_pid, target_ip);

    result = system(nsenter_cmd);
    return (result == 0) ? 1 : 0;  // Return 1 for success, 0 for failure
}
```

Core Debugging Function Skeletons

Container State Diagnostic Function:

```c
// diagnose_container_state - Comprehensive container state analysis
// This function should be called when container behavior is unexpected
int diagnose_container_state(container_instance_t *container,
                             container_config_t *config,
                             char *diagnosis_buffer,
                             size_t buffer_size) {
    // TODO 1: Verify process is still running using kill(container->child_pid, 0)
    // TODO 2: Check all namespace file descriptors in container->namespace_fds array
    //         - Use fstat() to verify each fd is still valid
    //         - Compare namespace inodes between parent and child
    // TODO 3: Inspect cgroup membership by reading /proc/PID/cgroup
    //         - Verify process is assigned to expected cgroup path
    //         - Check that cgroup directory still exists
    // TODO 4: Read current resource usage from cgroup stat files
    //         - Memory usage vs limit in memory.usage_in_bytes
    //         - CPU throttling stats from cpu.stat
    //         - Process count vs limit
    // TODO 5: Test namespace isolation by comparing /proc/PID/ns/* inodes
    //         - PID namespace: different inode = isolated
    //         - Mount namespace: check /proc/PID/mountinfo differences
    //         - Network namespace: check interface lists
    // TODO 6: Compile diagnosis into human-readable report in diagnosis_buffer
    //         - Include specific recommendations for each detected issue
    // Hint: Use inspect_process_namespaces() and read_cgroup_stats() helpers above
}
```

Resource Limit Enforcement Checker:

```c
// verify_resource_enforcement - Check if cgroup limits are working
// Call this when containers appear to ignore resource limits
int verify_resource_enforcement(const char *cgroup_path,
                                container_config_t *config,
                                enforcement_report_t *report) {
    // TODO 1: Read current memory usage from memory.usage_in_bytes
    //         - Compare with config->memory_limit_bytes
    //         - Check memory.failcnt for limit enforcement events
    // TODO 2: Read CPU throttling statistics from cpu.stat
    //         - Parse nr_periods, nr_throttled, throttled_time
    //         - Calculate throttling percentage
    // TODO 3: Check process count against max_processes limit
    //         - Count lines in cgroup.procs file
    //         - Verify fork() failures when limit approached
    // TODO 4: Test limit enforcement by attempting to exceed limits
    //         - Try allocating memory beyond limit (should trigger OOM)
    //         - Create processes beyond process limit (should fail)
    // TODO 5: Fill enforcement_report_t with findings
    //         - Set enforcement_working flag based on test results
    //         - Include specific failure details and recommendations
    // Hint: Use read_cgroup_stats() to get current usage statistics
}
```

Language-Specific Debugging Hints

C-Specific Debugging Considerations:

- Use `strace -f -p PID` to trace system calls for namespace creation issues
- Check `errno` immediately after failed system calls - container code paths often overwrite error codes
- Use `valgrind --trace-children=yes` to catch memory errors in forked container processes
- Compile with `-g -O0` for debugging to ensure accurate line number information

- Use `gdb --args program` and `set follow-fork-mode child` to debug container child processes

File Organization for Debugging:

```
project-root/
  src/container/
    container_debug.c        ← debugging utilities (implement above skeletons)
    container_debug.h        ← debugging function declarations
  debug/
    namespace_inspector.c    ← standalone namespace inspection tool
    cgroup_monitor.c         ← resource usage monitoring tool
    connectivity_tester.c    ← network connectivity verification
  scripts/
    debug_container.sh       ← wrapper script for common debugging workflows
    collect_debug_info.sh    ← automated debug info collection
```

Milestone Debugging Checkpoints

After Milestone 1 (PID Namespace):

- Run `ps aux` from inside container - should show only container processes with PID 1 as init
- Check `lsns -t pid` - container should have different PID namespace inode than host
- Verify zombie reaping by creating short-lived child processes and confirming they don't accumulate
- Test signal handling by sending signals to container init process

After Milestone 2 (Mount Namespace):

- Run `findmnt` from container and host - should show different mount trees
- Verify `/proc` and `/sys` are mounted and functional in container
- Check that host filesystem changes don't appear in container
- Confirm pivot_root worked by checking that container root directory is `/`

After Milestone 3 (Network Namespace):

- Run `ip addr show` in container - should show container-specific interfaces
- Test connectivity with `ping` to external addresses
- Verify inter-container communication through bridge networking
- Check that container cannot see host network interfaces

After Milestone 4 (Cgroups):

- Trigger memory limit by allocating memory beyond limit - should cause OOM kill
- Generate CPU load and verify throttling occurs when limit exceeded
- Create many processes to test process limit enforcement
- Monitor resource usage with `systemd-cgls` and cgroup stat files

Common Debugging Scenarios

Debugging Scenario Reference:

Symptom	Investigation Steps	Expected Findings	Resolution
Container sees host processes	Check PID namespace creation, verify lsns output	Different PID namespace inode	Fix clone() flags or namespace creation
Container has no network	Check network namespace, veth pair, routing	Isolated network namespace with configured interfaces	Fix veth creation or IP configuration
Resource limits ignored	Check cgroup assignment, controller availability	Process in correct cgroup with active controllers	Fix cgroup creation or process assignment
Container startup hangs	Check parent-child synchronization, error propagation	Clear error messages or successful startup sequence	Fix synchronization pipes or error handling
Filesystem isolation broken	Check mount namespace, pivot_root, mount propagation	Separate mount namespace with private propagation	Fix mount namespace creation or propagation settings

This comprehensive debugging framework provides systematic approaches to diagnosing and fixing the most common container implementation issues across all four milestone areas.

Future Extensions

> **Milestone(s):** This section provides a roadmap beyond the basic container implementation from milestones 1-4, outlining additional container features that can be built on the foundation of PID namespaces, mount namespaces, network namespaces, and cgroups resource management.

The basic container implementation established through the four core milestones provides a solid foundation for process isolation, filesystem isolation, network isolation, and resource control. However, production container runtimes like Docker and containerd implement numerous additional features that enhance security, usability, and operational capabilities. This section explores three major categories of extensions that can be built upon our basic container runtime: additional namespace support for enhanced isolation, image management for portable container distribution, and orchestration capabilities for multi-container coordination.

These extensions represent natural evolutionary paths for the basic container runtime, each building incrementally on the existing namespace and cgroups infrastructure. Understanding these extensions helps illuminate why container platforms have become so powerful and why they've largely replaced virtual machines for many deployment scenarios. Each extension category addresses specific operational challenges that emerge when containers move from development prototypes to production systems.

The extensions follow a progression from enhanced isolation capabilities through content management to distributed coordination. Additional namespaces provide finer-grained isolation boundaries, addressing security and administrative concerns that emerge in multi-tenant environments. Image management solves the distribution

and versioning challenges that arise when containers need to run consistently across different environments. Container orchestration addresses the coordination and service discovery challenges that emerge when applications are composed of multiple cooperating containers.

Additional Namespaces: User, UTS, and IPC Namespace Support

Mental Model: Security Zones in an Office Building

Think of additional namespaces like security zones in a modern office building. Our basic container implementation is like giving each tenant their own office (PID namespace), their own file cabinets (mount namespace), and their own phone system (network namespace). Additional namespaces are like adding more sophisticated security measures: user namespaces are like giving each tenant their own security badge system where they can be "admin" inside their space but have no privileges outside; UTS namespaces are like giving each tenant their own building address and signage; IPC namespaces are like ensuring each tenant's intercom system is completely separate from others.

The current basic container implementation uses three of the six available Linux namespace types. The remaining three namespaces—user, UTS (Unix Timesharing System), and IPC (Inter-Process Communication)—provide additional isolation boundaries that enhance security and administrative separation. These namespaces become increasingly important as containers move from development environments to production systems where multiple tenants or applications share the same host infrastructure.

User Namespace Architecture

User namespaces provide perhaps the most significant security enhancement by creating isolated user and group ID mappings. Inside a user namespace, processes can have different user and group IDs than they appear to have from the host perspective. This capability enables containers to run processes as root (UID 0) inside the namespace while those same processes appear as unprivileged users from the host perspective.

The user namespace implementation requires extending the `container_config_t` structure to include user mapping specifications:

Field Name	Type	Description
`enable_user_ns`	`int`	Whether to create user namespace for container
`uid_map_inside`	`uid_t`	User ID as seen inside the container namespace
`uid_map_outside`	`uid_t`	User ID as seen from host perspective
`uid_map_length`	`size_t`	Number of consecutive UIDs in the mapping range
`gid_map_inside`	`gid_t`	Group ID as seen inside the container namespace
`gid_map_outside`	`gid_t`	Group ID as seen from host perspective
`gid_map_length`	`size_t`	Number of consecutive GIDs in the mapping range

The user namespace creation process involves several steps that must occur in a specific sequence. First, the container process creates the user namespace using the `CLONE_NEWUSER` flag with the `clone()` system call. However, unlike other namespaces, the user namespace requires additional setup steps after creation to establish the user and group ID mappings.

> **Design Insight: User Namespace Timing Requirements** User namespaces have unique timing constraints because processes inside a newly created user namespace initially have no user or group ID mappings. This means they cannot perform operations that require specific user privileges until the mappings are established. The parent process must write to the `/proc/[pid]/uid_map` and `/proc/[pid]/gid_map` files to establish these mappings before the container process can proceed with other initialization tasks.

The user namespace component would implement the following interface:

Method Name	Parameters	Returns	Description
`create_user_namespace`	`config, container`	`int`	Creates user namespace with CLONE_NEWUSER flag
`setup_user_mappings`	`config, child_pid`	`int`	Establishes UID/GID mappings from parent process
`write_uid_map`	`child_pid, inside_uid, outside_uid, length`	`int`	Writes user ID mapping to /proc/[pid]/uid_map
`write_gid_map`	`child_pid, inside_gid, outside_gid, length`	`int`	Writes group ID mapping to /proc/[pid]/gid_map
`verify_user_isolation`	`container_pid`	`int`	Verifies user namespace isolation is working

UTS Namespace Architecture

UTS namespaces isolate the system hostname and domain name, allowing each container to have its own distinct identity for networking and administrative purposes. While this might seem like a minor isolation boundary, it becomes important for applications that rely on hostname for configuration, logging, or service discovery.

The UTS namespace extension requires adding hostname configuration to the container setup:

Field Name	Type	Description
`container_hostname`	`char*`	Hostname to set inside UTS namespace
`container_domainname`	`char*`	Domain name to set inside UTS namespace

The UTS namespace implementation is relatively straightforward compared to user namespaces because it doesn't require complex mappings or parent-child coordination. The container process creates the UTS

namespace using `CLONE_NEWUTS` and then uses the `sethostname()` and `setdomainname()` system calls to establish the container's identity.

Method Name	Parameters	Returns	Description
`create_uts_namespace`	`config, container`	`int`	Creates UTS namespace with CLONE_NEWUTS flag
`setup_container_hostname`	`hostname`	`int`	Sets hostname inside UTS namespace using sethostname()
`setup_container_domainname`	`domainname`	`int`	Sets domain name inside UTS namespace using setdomainname()
`get_container_identity`	`hostname_buf, domain_buf`	`int`	Retrieves current hostname and domain name

IPC Namespace Architecture

IPC namespaces isolate System V IPC objects (message queues, semaphores, shared memory segments) and POSIX message queues. This isolation prevents containers from interfering with each other's inter-process communication mechanisms and provides an additional security boundary.

The IPC namespace implementation requires minimal configuration because it primarily provides isolation rather than requiring specific setup:

Field Name	Type	Description
`enable_ipc_ns`	`int`	Whether to create IPC namespace for container
`cleanup_ipc_objects`	`int`	Whether to clean up IPC objects on container exit

The IPC namespace component provides these interface methods:

Method Name	Parameters	Returns	Description
`create_ipc_namespace`	`config, container`	`int`	Creates IPC namespace with CLONE_NEWIPC flag
`list_ipc_objects`	`objects, max_objects`	`int`	Lists System V IPC objects in current namespace
`cleanup_ipc_namespace`	`container`	`int`	Removes any remaining IPC objects before namespace destruction

Integration with Existing Container System

Adding these additional namespaces requires updating the existing container creation workflow to accommodate the new namespace types. The `CONTAINER_NS_FLAGS` constant would be extended to include the new namespace flags:

```c
#define CONTAINER_NS_FLAGS_EXTENDED
(CLONE_NEWPID|CLONE_NEWNS|CLONE_NEWNET|CLONE_NEWUTS|CLONE_NEWIPC|CLONE_NEWUSER)
```

The container startup sequence must be modified to handle the user namespace timing requirements, where user mappings must be established before the container process can proceed with other initialization steps. This requires extending the parent-child synchronization mechanism to include a user namespace setup phase.

> **Architecture Decision: User Namespace Setup Timing**
>
> - **Context**: User namespaces require parent process to establish UID/GID mappings after namespace creation but before container initialization continues
> - **Options Considered**:
>   1. Setup mappings before any other namespaces
>   2. Setup mappings after all namespaces are created
>   3. Defer mapping setup until container process requests it
> - **Decision**: Setup user mappings immediately after user namespace creation but before other namespace initialization
> - **Rationale**: This ensures the container process has proper privileges for subsequent namespace setup operations while maintaining security isolation
> - **Consequences**: Requires extending parent-child synchronization protocol and adds complexity to error handling during startup sequence

Common Pitfalls in Additional Namespace Implementation

⚠️ **Pitfall: User Namespace Capability Confusion** When user namespaces are enabled, processes inside the namespace may have different capabilities than expected. A process running as UID 0 (root) inside a user namespace has administrative capabilities within that namespace but no special privileges on the host system. This can cause confusion when processes attempt operations that would normally succeed for root but fail due to the user namespace boundary. The solution is to carefully design capability requirements and test operations both inside and outside user namespaces.

⚠️ **Pitfall: Hostname Conflicts in Service Discovery** Applications that rely on hostname for service discovery may behave unexpectedly when each container has its own hostname via UTS namespace. Multiple containers might choose the same hostname, leading to conflicts in service registration systems. The solution is to implement hostname allocation policies that ensure uniqueness within the deployment environment, similar to IP address allocation for network namespaces.

⚠️ **Pitfall: IPC Object Cleanup Timing** System V IPC objects persist until explicitly removed, even after the processes that created them exit. In IPC namespaces, these objects are automatically cleaned up when the namespace is destroyed, but applications might not expect this behavior. The solution is to implement graceful IPC cleanup that allows applications to properly close IPC resources before namespace destruction.

Image Management: Container Image Layers and Overlay Filesystems

Mental Model: Layered Transparency Sheets

Think of container image management like the layered transparency sheets used in anatomy textbooks. Each sheet shows one system of the body—skeletal, muscular, circulatory—and when you stack them together, you see the complete organism. Container images work similarly: the base layer might contain the operating system files, the next layer adds application dependencies, another layer adds the application binary, and the final layer adds configuration files. When you stack all these layers together using an overlay filesystem, the container sees a complete, unified filesystem that appears to be a regular directory tree.

The basic container implementation uses a simple approach where each container has its own complete root filesystem directory. While this works for development and learning, production container systems use layered images that enable efficient sharing of common components between containers, faster container startup times, and reduced storage requirements. This image management system requires implementing overlay filesystems, image layer management, and content distribution mechanisms.

Container Image Architecture

Container images consist of multiple read-only layers stacked together with a single read-write layer on top. Each layer represents a set of filesystem changes (files added, modified, or deleted) relative to the previous layer. This layered approach enables multiple containers to share common base layers while maintaining their own writable layer for runtime changes.

The image management system requires new data structures to represent image metadata and layer information:

Structure Name	Field Name	Type	Description
`image_layer_t`	`layer_id`	`char[64]`	SHA256 hash identifying this layer uniquely
	`parent_layer_id`	`char[64]`	SHA256 hash of parent layer, empty for base layer
	`layer_path`	`char[256]`	Filesystem path to layer directory or archive
	`layer_size`	`size_t`	Total size of layer content in bytes
	`change_type`	`layer_change_t`	Whether layer adds, modifies, or removes files
`container_image_t`	`image_id`	`char[64]`	SHA256 hash identifying complete image
	`image_name`	`char[128]`	Human-readable image name like "ubuntu:20.04"
	`layer_count`	`int`	Number of layers in image stack
	`layers`	`image_layer_t*`	Array of layers ordered from base to top
	`total_size`	`size_t`	Combined size of all layers in image
`overlay_mount_t`	`lower_dirs`	`char**`	Array of read-only layer directories
	`upper_dir`	`char*`	Read-write layer directory for container changes
	`work_dir`	`char*`	Overlay filesystem work directory
	`merged_dir`	`char*`	Mount point showing unified view of all layers

Overlay Filesystem Implementation

Overlay filesystems provide the mechanism for combining multiple image layers into a single unified directory tree. The Linux overlay filesystem driver (overlayfs) takes multiple lower directories (read-only layers), one upper directory (read-write layer), and a work directory (for atomic operations), then presents them as a single merged directory.

The overlay filesystem setup process involves several coordinated steps. First, the image management system prepares the layer directories by extracting or mounting each image layer to a separate directory. These directories become the "lower" layers in overlay filesystem terminology. Second, the system creates an empty "upper" directory where the container can make filesystem changes during execution. Third, the system creates a "work" directory that overlay filesystem uses for atomic operations. Finally, the system mounts the overlay filesystem with all these directories to create the unified view.

The overlay filesystem component provides these interface methods:

Method Name	Parameters	Returns	Description
`setup_overlay_mount`	`image, container_id, overlay`	`int`	Prepares and mounts overlay filesystem for container
`prepare_image_layers`	`image, layer_dirs`	`int`	Extracts or mounts all image layers to separate directories
`create_overlay_dirs`	`container_id, upper, work, merged`	`int`	Creates upper, work, and merged directories for overlay mount
`mount_overlay_filesystem`	`overlay`	`int`	Mounts overlay filesystem with specified layer configuration
`cleanup_overlay_mount`	`overlay`	`int`	Unmounts overlay filesystem and cleans up directories
`commit_container_changes`	`overlay, new_layer`	`int`	Creates new image layer from container's upper directory

The overlay mount process requires careful handling of directory permissions and ownership. Each layer directory must be accessible to the container process, but the overlay filesystem itself manages the permission and ownership mapping between layers. When files exist in multiple layers, the topmost layer takes precedence, implementing the "copy-on-write" semantics that allow containers to modify files without affecting the underlying image layers.

> **Design Insight: Copy-on-Write Behavior** When a container attempts to modify a file that exists in a lower (read-only) layer, the overlay filesystem automatically copies that file to the upper (read-write) layer and applies the modification there. This copy-on-write behavior ensures that image layers remain immutable while allowing containers to make necessary runtime changes. The performance impact of this copying operation can be significant for large files, which is why container images are typically designed with frequently modified files placed in higher layers.

Image Layer Management

Container images are typically distributed as compressed archives containing layer data and metadata. The image management system must handle extracting these archives, verifying layer integrity, and managing the storage of layer data on the local filesystem. This involves implementing a local image store that can efficiently store and retrieve image layers.

The image store uses content-addressable storage where each layer is stored using its SHA256 hash as the directory name. This approach ensures that identical layers are stored only once, even when they appear in multiple images. The image store structure looks like this:

```
/var/lib/container-basic/images/
├── layers/
│   ├── sha256:a1b2c3d4.../
│   │   ├── layer.tar        ← compressed layer content
│   │   └── metadata.json    ← layer metadata
│   └── sha256:e5f6g7h8.../
│       ├── layer.tar
│       └── metadata.json
├── images/
│   ├── ubuntu-20.04/
│   │   └── manifest.json    ← image manifest listing all layers
│   └── nginx-latest/
│       └── manifest.json
└── containers/
    ├── container-001/
    │   ├── upper/           ← read-write layer for this container
    │   ├── work/            ← overlay filesystem work directory
    │   └── merged/          ← overlay mount point
    └── container-002/
        ├── upper/
        ├── work/
        └── merged/
```

The image layer management component provides these interface methods:

Method Name	Parameters	Returns	Description
`pull_image_layers`	`image_name, registry_url`	`int`	Downloads image layers from remote registry
`extract_layer_archive`	`layer_path, extract_dir`	`int`	Extracts compressed layer archive to directory
`verify_layer_integrity`	`layer_path, expected_hash`	`int`	Verifies layer content matches expected SHA256 hash
`store_image_manifest`	`image_name, manifest`	`int`	Stores image manifest in local image store
`load_image_manifest`	`image_name, manifest`	`int`	Loads image manifest from local image store
`garbage_collect_layers`	`retain_days`	`int`	Removes unused layers older than specified days
`list_available_images`	`images, max_images`	`int`	Lists all images available in local image store

Integration with Mount Namespace

The image management system must integrate with the existing mount namespace implementation to replace the simple root filesystem setup with overlay filesystem mounting. This requires modifying the

`setup_container_rootfs` function to use overlay mounting instead of bind mounting a single directory.

The integration involves updating the container startup sequence to include image layer preparation before mount namespace creation. The modified sequence becomes: validate container configuration, pull and extract image layers if needed, prepare overlay directories, create mount namespace, mount overlay filesystem as container root, mount essential filesystems (/proc, /sys, /dev), and finally execute the container process.

> **Architecture Decision: Overlay vs Bind Mount Integration**
>
> - **Context**: Existing mount namespace implementation uses bind mounts for simple directory-based root filesystems, but overlay filesystems require different mount setup
> - **Options Considered**:
>   1. Replace bind mount implementation entirely with overlay mounting
>   2. Add overlay support as alternative mount type with configuration flag
>   3. Implement overlay as wrapper around existing bind mount system
> - **Decision**: Add overlay support as alternative mount type with configuration flag
> - **Rationale**: This preserves backward compatibility for simple use cases while enabling advanced image layer functionality when needed
> - **Consequences**: Increases code complexity but provides flexibility for different deployment scenarios and easier testing of both approaches

Common Pitfalls in Image Management Implementation

⚠️ **Pitfall: Layer Extraction Race Conditions** When multiple containers attempt to use the same image simultaneously, they may try to extract the same layers concurrently, leading to corruption or incomplete extractions. The solution is to implement file locking around layer extraction operations and check for existing extracted layers before beginning extraction.

⚠️ **Pitfall: Overlay Work Directory Cleanup** The overlay filesystem work directory must be on the same filesystem as the upper directory for atomic operations to work correctly. Additionally, the work directory must be cleaned up properly when containers exit, or subsequent overlay mounts may fail with "work directory not empty" errors. The solution is to ensure work directories are created on the correct filesystem and implement thorough cleanup procedures.

⚠️ **Pitfall: Layer Hash Verification Bypass** Skipping layer integrity verification during development can lead to subtle corruption issues that are difficult to debug. Corrupted layers may cause containers to behave unpredictably or fail in unexpected ways. The solution is to always verify layer hashes, even during development, and implement clear error messages when verification fails.

Container Orchestration: Multi-Container Coordination and Service Discovery

Mental Model: Restaurant Kitchen Coordination

Think of container orchestration like the coordination system in a busy restaurant kitchen. Individual containers are like cooking stations—the grill, the salad prep area, the dessert station—each specialized for specific tasks. The

orchestration system is like the head chef and expeditors who coordinate between stations: they ensure orders flow smoothly between stations, ingredients are available when needed, backup stations can take over when one fails, and the final dishes are assembled correctly. Service discovery is like the communication system that lets each station know where to send completed items and where to request ingredients.

The basic container implementation focuses on isolating and controlling individual containers, but real-world applications typically consist of multiple containers that must coordinate to provide complete functionality. A web application might have containers for the web server, database, cache, and background job processor, all of which need to communicate with each other and be managed as a cohesive system. Container orchestration addresses these multi-container coordination challenges.

Service Discovery Architecture

Service discovery enables containers to locate and communicate with other containers without hardcoding network addresses or relying on external configuration. As containers start and stop dynamically, their IP addresses change, making static configuration impractical. Service discovery systems provide a dynamic registry where containers can register their services and query for the services they depend on.

The service discovery system requires data structures to represent services and their endpoints:

Structure Name	Field Name	Type	Description
`service_endpoint_t`	`service_name`	`char[64]`	Name of service like "web-server" or "database"
	`container_id`	`char[64]`	Identifier of container providing this service
	`ip_address`	`char[16]`	IP address where service is accessible
	`port`	`uint16_t`	TCP/UDP port number for service
	`health_status`	`endpoint_health_t`	Current health status of service endpoint
	`last_heartbeat`	`time_t`	Timestamp of last health check or heartbeat
`service_registry_t`	`services`	`service_endpoint_t**`	Hash table mapping service names to endpoint arrays
	`service_count`	`int`	Total number of registered services
	`registry_lock`	`pthread_mutex_t`	Mutex protecting concurrent registry access
`service_query_t`	`service_name`	`char[64]`	Name of service being queried
	`preferred_zone`	`char[32]`	Preferred availability zone or location
	`load_balancing`	`lb_strategy_t`	Load balancing strategy for multiple endpoints
	`timeout_ms`	`int`	Maximum time to wait for service resolution

The service discovery implementation provides both registration and query capabilities. Containers register their services when they start and deregister when they stop. Other containers query the service registry to discover available services and their current endpoints.

Method Name	Parameters	Returns	Description
`register_service`	`service_name, container_id, ip, port`	`int`	Registers new service endpoint in registry
`deregister_service`	`service_name, container_id`	`int`	Removes service endpoint from registry
`discover_service`	`query, endpoints, max_endpoints`	`int`	Queries registry for available service endpoints
`update_service_health`	`service_name, container_id, health`	`int`	Updates health status of registered service
`list_services`	`services, max_services`	`int`	Lists all services currently registered
`cleanup_expired_services`	`max_age_seconds`	`int`	Removes services that haven't sent heartbeats

Container Lifecycle Management

Orchestration systems manage the complete lifecycle of multiple containers, including startup ordering, dependency management, failure handling, and scaling decisions. Unlike individual container management, orchestration must reason about relationships between containers and coordinate their lifecycle events.

The orchestration system uses deployment specifications that describe the desired state of multi-container applications:

Structure Name	Field Name	Type	Description
`container_spec_t`	`container_name`	`char[64]`	Unique name for container within deployment
	`image_name`	`char[128]`	Container image to use for this container
	`replicas`	`int`	Number of identical container instances to run
	`dependencies`	`char**`	Array of container names this depends on
	`service_ports`	`port_mapping_t*`	Ports this container exposes as services
	`resource_limits`	`container_config_t`	CPU, memory, and other resource constraints
`deployment_spec_t`	`deployment_name`	`char[64]`	Name of multi-container deployment
	`container_specs`	`container_spec_t*`	Array of container specifications
	`spec_count`	`int`	Number of containers in deployment
	`network_name`	`char[64]`	Name of network for inter-container communication
`deployment_state_t`	`spec`	`deployment_spec_t`	Desired state specification
	`running_containers`	`container_instance_t**`	Currently running container instances
	`container_count`	`int`	Number of currently running containers
	`state_lock`	`pthread_mutex_t`	Mutex protecting concurrent state access

The orchestration engine implements a control loop that continuously compares the desired state (deployment specification) with the actual state (running containers) and takes actions to reconcile any differences. This

approach ensures that containers are automatically restarted if they fail, scaled up or down as needed, and started in the correct dependency order.

Method Name	Parameters	Returns	Description
`create_deployment`	`spec, deployment`	`int`	Creates new multi-container deployment
`update_deployment`	`deployment, new_spec`	`int`	Updates existing deployment with new specification
`delete_deployment`	`deployment`	`int`	Stops all containers and deletes deployment
`reconcile_deployment_state`	`deployment`	`int`	Ensures actual state matches desired state
`scale_container_replicas`	`deployment, container_name, replicas`	`int`	Changes number of replicas for specific container
`restart_failed_containers`	`deployment`	`int`	Restarts containers that have failed or stopped
`check_container_dependencies`	`spec, container_name`	`int`	Verifies all dependencies are running and healthy

Inter-Container Communication

Containers within an orchestrated deployment need secure and efficient communication mechanisms. While containers can communicate through the network namespace and bridge networking established in milestone 3, orchestration systems typically provide higher-level communication abstractions that handle service discovery, load balancing, and failure recovery automatically.

The communication system builds on the existing network namespace infrastructure but adds service-aware networking features. Instead of containers needing to know specific IP addresses and ports, they can communicate using service names that the orchestration system resolves dynamically.

Structure Name	Field Name	Type	Description
`service_connection_t`	`source_container`	`char[64]`	Container initiating the connection
	`target_service`	`char[64]`	Service name being connected to
	`connection_type`	`connection_type_t`	HTTP, TCP, UDP, or other protocol
	`load_balancer`	`load_balancer_t*`	Load balancer managing multiple endpoints
`load_balancer_t`	`strategy`	`lb_strategy_t`	Round-robin, least-connections, or random
	`endpoints`	`service_endpoint_t*`	Array of available service endpoints
	`endpoint_count`	`int`	Number of currently healthy endpoints
	`current_index`	`int`	Current position for round-robin selection

The inter-container communication component extends the existing network namespace functionality:

Method Name	Parameters	Returns	Description
`create_service_network`	`network_name, subnet`	`int`	Creates dedicated network for service communication
`connect_container_to_service`	`container, service_name, connection`	`int`	Establishes connection to named service
`setup_load_balancer`	`service_name, strategy, balancer`	`int`	Creates load balancer for service with multiple endpoints
`route_service_request`	`connection, request`	`int`	Routes request to appropriate service endpoint
`handle_endpoint_failure`	`balancer, failed_endpoint`	`int`	Removes failed endpoint from load balancer rotation
`update_service_endpoints`	`balancer, new_endpoints`	`int`	Updates load balancer with current healthy endpoints

Integration with Existing Container System

Container orchestration builds upon all four milestone components—PID namespaces, mount namespaces, network namespaces, and cgroups—but coordinates their use across multiple containers. The orchestration system becomes the higher-level controller that manages multiple instances of the basic container runtime.

The integration requires extending the container creation and management interfaces to support orchestration-driven lifecycle management. Instead of manually calling `container_create`, `container_wait`, and `container_destroy` for individual containers, the orchestration system manages these operations based on deployment specifications and health monitoring.

The orchestration system must also coordinate resource allocation across multiple containers to prevent resource conflicts and ensure fair sharing. This involves extending the cgroups implementation to support hierarchical resource allocation where the orchestration system allocates resources to deployments, and deployments sub-allocate resources to individual containers.

> **Architecture Decision: Centralized vs Distributed Orchestration**
>
> - **Context**: Multi-container deployments need coordination for startup, failure handling, and resource management
> - **Options Considered**:
>   1. Centralized orchestration with single control process managing all containers
>   2. Distributed orchestration with peer-to-peer coordination between containers
>   3. Hierarchical orchestration with local controllers managed by global coordinator
> - **Decision**: Centralized orchestration with single control process for basic implementation
> - **Rationale**: Centralized approach is simpler to implement and debug, provides consistent state management, and aligns with the educational goals of understanding orchestration concepts
> - **Consequences**: Creates single point of failure but provides clear operational model that can be extended to distributed approaches later

Common Pitfalls in Container Orchestration Implementation

⚠️ **Pitfall: Circular Dependency Detection** When containers have complex dependency relationships, it's possible to create circular dependencies where container A depends on container B, which depends on container C, which depends back on container A. This creates a deadlock where no container can start because each is waiting for another. The solution is to implement dependency graph analysis that detects cycles before attempting to start containers and reports clear error messages about which containers form the cycle.

⚠️ **Pitfall: Service Discovery Race Conditions** During rapid container startup and shutdown, service registration and deregistration events can arrive out of order, leading to incorrect service registry state. A container might deregister just before its replacement registers, causing temporary service unavailability. The solution is to implement proper synchronization around service registry operations and use timestamps or version numbers to handle out-of-order events correctly.

⚠️ **Pitfall: Resource Allocation Conflicts** When multiple containers compete for limited resources, the orchestration system might allocate more resources than are actually available on the system. This can lead to containers being unable to start or experiencing performance degradation due to resource pressure. The solution is to implement resource accounting that tracks total allocated resources across all deployments and prevents over-allocation beyond system capacity.

Implementation Guidance

The future extensions build incrementally on the existing container runtime foundation, allowing learners to implement them in stages based on their interests and requirements. Each extension category represents a significant engineering undertaking, so it's recommended to start with the additional namespaces as they most closely parallel the existing namespace implementations.

Technology Recommendations

Extension Category	Simple Implementation	Advanced Implementation
Additional Namespaces	Direct syscall integration with existing namespace code	Namespace capability management and privilege dropping
Image Management	Directory-based layer storage with tar archives	Content-addressable storage with compression and deduplication
Container Orchestration	Single-host process coordinator with file-based service registry	Distributed coordination with etcd or consul for service discovery

Recommended File Structure Extension

Building on the existing container runtime structure, the extensions would add new directories:

```
container-basic/
├── src/
│   ├── namespaces/
│   │   ├── pid_namespace.c      ← existing
│   │   ├── mount_namespace.c    ← existing
│   │   ├── network_namespace.c  ← existing
│   │   ├── user_namespace.c     ← new: user namespace implementation
│   │   ├── uts_namespace.c      ← new: UTS namespace implementation
│   │   └── ipc_namespace.c      ← new: IPC namespace implementation
│   ├── images/
│   │   ├── image_store.c        ← new: local image storage management
│   │   ├── layer_manager.c      ← new: layer extraction and mounting
│   │   ├── overlay_fs.c         ← new: overlay filesystem integration
│   │   └── image_registry.c     ← new: remote image pulling
│   ├── orchestration/
│   │   ├── service_discovery.c  ← new: service registration and lookup
│   │   ├── deployment_manager.c ← new: multi-container lifecycle
│   │   ├── load_balancer.c      ← new: service load balancing
│   │   └── resource_scheduler.c ← new: cross-container resource allocation
│   └── container.c              ← existing: updated to support extensions
├── include/
│   ├── container_extensions.h   ← new: extension API definitions
│   └── orchestration_types.h    ← new: orchestration data structures
└── tests/
    ├── test_additional_namespaces.c  ← new: additional namespace tests
    ├── test_image_management.c       ← new: image layer tests
    └── test_orchestration.c          ← new: multi-container tests
```

Infrastructure Starter Code: Service Registry

Here's a complete service registry implementation that provides the foundation for container orchestration:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <time.h>
#include <sys/queue.h>

#define MAX_SERVICE_NAME 64
#define MAX_CONTAINER_ID 64
#define MAX_ENDPOINTS_PER_SERVICE 16

typedef enum {
    ENDPOINT_HEALTHY,
    ENDPOINT_UNHEALTHY,
    ENDPOINT_UNKNOWN
} endpoint_health_t;

typedef struct service_endpoint {
    char service_name[MAX_SERVICE_NAME];
    char container_id[MAX_CONTAINER_ID];
    char ip_address[16];
    uint16_t port;
    endpoint_health_t health_status;
    time_t last_heartbeat;
    LIST_ENTRY(service_endpoint) entries;
} service_endpoint_t;

typedef struct service_registry {
    LIST_HEAD(endpoint_list, service_endpoint) endpoints;
    pthread_mutex_t registry_lock;
```

```
    int endpoint_count;
} service_registry_t;

// Initialize global service registry
service_registry_t global_registry = {
    .endpoints = LIST_HEAD_INITIALIZER(global_registry.endpoints),
    .registry_lock = PTHREAD_MUTEX_INITIALIZER,
    .endpoint_count = 0
};

int register_service(const char* service_name, const char* container_id,
                     const char* ip_address, uint16_t port) {
    service_endpoint_t* endpoint = malloc(sizeof(service_endpoint_t));
    if (!endpoint) return -1;

    strncpy(endpoint->service_name, service_name, MAX_SERVICE_NAME - 1);
    strncpy(endpoint->container_id, container_id, MAX_CONTAINER_ID - 1);
    strncpy(endpoint->ip_address, ip_address, 15);
    endpoint->port = port;
    endpoint->health_status = ENDPOINT_HEALTHY;
    endpoint->last_heartbeat = time(NULL);

    pthread_mutex_lock(&global_registry.registry_lock);
    LIST_INSERT_HEAD(&global_registry.endpoints, endpoint, entries);
    global_registry.endpoint_count++;
    pthread_mutex_unlock(&global_registry.registry_lock);

    printf("Registered service %s at %s:%d for container %s\n",
           service_name, ip_address, port, container_id);
```

```
    return 0;
}

int discover_service(const char* service_name, service_endpoint_t* results,
                     int max_results) {
    int found_count = 0;
    service_endpoint_t* endpoint;

    pthread_mutex_lock(&global_registry.registry_lock);
    LIST_FOREACH(endpoint, &global_registry.endpoints, entries) {
        if (strcmp(endpoint->service_name, service_name) == 0 &&
            endpoint->health_status == ENDPOINT_HEALTHY &&
            found_count < max_results) {
            memcpy(&results[found_count], endpoint, sizeof(service_endpoint_t));
            found_count++;
        }
    }
    pthread_mutex_unlock(&global_registry.registry_lock);

    return found_count;
}

int deregister_service(const char* service_name, const char* container_id) {
    service_endpoint_t* endpoint;
    int removed = 0;

    pthread_mutex_lock(&global_registry.registry_lock);
    LIST_FOREACH(endpoint, &global_registry.endpoints, entries) {
        if (strcmp(endpoint->service_name, service_name) == 0 &&
```

```
            strcmp(endpoint->container_id, container_id) == 0) {
            LIST_REMOVE(endpoint, entries);
            global_registry.endpoint_count--;
            free(endpoint);
            removed = 1;
            break;
        }
    }
    pthread_mutex_unlock(&global_registry.registry_lock);

    if (removed) {
        printf("Deregistered service %s for container %s\n",
               service_name, container_id);
    }
    return removed ? 0 : -1;
}
```

Core Logic Skeleton: User Namespace Setup

```c
// Creates user namespace and establishes UID/GID mappings for container process
int create_user_namespace(container_config_t* config, container_instance_t* container) {
    // TODO 1: Check if user namespace is enabled in configuration
    // TODO 2: Create child process with CLONE_NEWUSER flag using safe_clone()
    // TODO 3: Store child PID in container instance structure
    // TODO 4: Write UID mapping to /proc/[child_pid]/uid_map from parent process
    //         Format: "inside_uid outside_uid length" (e.g., "0 1000 1")
    // TODO 5: Write GID mapping to /proc/[child_pid]/gid_map from parent process
    //         Format: "inside_gid outside_gid length" (e.g., "0 1000 1")
    // TODO 6: Signal child process that user mappings are complete
    // TODO 7: Child process continues with container initialization as mapped user
    // Hint: Use container_sync_t pipes for parent-child coordination
    // Hint: Check /proc/sys/user/max_user_namespaces for system limits
}

// Sets up overlay filesystem mount with multiple image layers
int setup_overlay_mount(container_image_t* image, const char* container_id,
                        overlay_mount_t* overlay) {
    // TODO 1: Create container-specific directories (upper, work, merged)
    // TODO 2: Prepare lower directories by extracting each image layer
    // TODO 3: Build comma-separated list of lower directories for mount options
    // TODO 4: Construct overlay mount options string:
    //         "lowerdir=/layer1:/layer2,upperdir=/upper,workdir=/work"
    // TODO 5: Mount overlay filesystem to merged directory using mount() syscall
    // TODO 6: Verify overlay mount succeeded by checking merged directory content
    // TODO 7: Store all directory paths in overlay structure for cleanup
    // Hint: Lower directories must be listed in bottom-to-top layer order
    // Hint: Work directory must be on same filesystem as upper directory
```

```
}


// Implements orchestration control loop for multi-container deployment
int reconcile_deployment_state(deployment_state_t* deployment) {
    // TODO 1: Compare desired container count vs actual running container count
    // TODO 2: Check dependency ordering - don't start containers before dependencies
    // TODO 3: For each missing container: create container_config_t and call container_create()
    // TODO 4: For each extra container: call container_destroy() to remove excess
    // TODO 5: For each running container: verify health and restart if failed
    // TODO 6: Update service registry with any container IP/port changes
    // TODO 7: Return 0 if desired state achieved, -1 if actions still needed
    // Hint: Use check_container_dependencies() before starting new containers
    // Hint: Allow time for containers to start before marking reconciliation complete
}
```

Language-Specific Implementation Hints

For C implementation of the extensions:

- **User Namespace Mappings**: Use `sprintf()` to format UID/GID mapping strings, then write them to `/proc/[pid]/uid_map` and `/proc/[pid]/gid_map` using regular file operations
- **Overlay Filesystem**: The mount() syscall for overlay requires specific option format: `mount("overlay", "/merged", "overlay", 0, "lowerdir=...,upperdir=...,workdir=...")`
- **Service Discovery**: Use POSIX threads (`pthread_create`) for background health checking and cleanup processes
- **Image Layer Extraction**: Use `libarchive` or simple `tar` command execution for extracting layer archives
- **Container Coordination**: Implement timeout handling using `select()` or `poll()` for non-blocking communication

Milestone Checkpoints

After implementing additional namespaces:

- Run `unshare --user --map-root-user whoami` to verify user namespace concepts
- Test that container process sees itself as UID 0 while host sees different UID
- Verify hostname isolation by setting different hostnames in containers

After implementing image management:

- Create simple layered image with base layer + application layer
- Verify overlay mount shows combined view of all layers
- Test that changes in container don't affect original image layers

After implementing basic orchestration:

- Start multi-container deployment with dependencies (database → web server)
- Verify service discovery allows web server to find database
- Test that failed containers are automatically restarted

These extensions represent the natural evolution from basic container isolation to production-ready container platforms, demonstrating how fundamental namespace and cgroups concepts scale to complex distributed systems.

Glossary

> **Milestone(s):** This section provides definitions for all technical terms used across milestones 1-4, establishing a common vocabulary for container implementation concepts including namespaces, cgroups, and container-specific terminology.

Mental Model: Technical Dictionary for Container Architecture

Think of this glossary as a technical dictionary specifically for container systems, similar to how medical professionals have specialized dictionaries that define terms precisely for their field. Just as "hypertension" has a specific medical meaning beyond "high blood pressure," container terminology has precise technical meanings that distinguish it from general computing concepts. For example, while "isolation" generally means "separation," in the container context it specifically refers to kernel-enforced boundaries that prevent processes from accessing resources outside their designated namespace.

This glossary serves as your reference manual when implementing containers, ensuring that when we discuss "pivot_root" or "zombie reaping," you understand not just the general concept but the specific technical implementation details and system call behaviors involved.

Core Container Concepts

The fundamental building blocks of container technology center around Linux kernel features that provide isolation and resource control. Understanding these concepts is essential before diving into implementation details.

Process isolation refers to the kernel-enforced separation of processes so they cannot interfere with each other's execution, memory space, or system resources. This goes beyond simple process boundaries to include isolated views of system resources like process IDs, filesystems, and network stacks. Process isolation forms the foundation of container security and resource management.

Namespaces are Linux kernel features that provide isolated views of system resources to different process groups. Each namespace type isolates a different category of system resources, allowing processes within a namespace to have their own private view while sharing the underlying kernel. Namespaces are the primary mechanism for achieving process isolation in containers.

Cgroups (control groups) are Linux kernel features that limit, account for, and control resource usage by groups of processes. While namespaces provide isolation by changing what processes can see, cgroups provide control by limiting what resources processes can consume. Cgroups enforce memory limits, CPU quotas, and process limits to prevent containers from consuming excessive system resources.

System Call and Kernel Interface Terms

Container implementation relies heavily on specific Linux system calls and kernel interfaces that manage process creation, namespace manipulation, and resource control.

Clone() system call creates a new process with specified namespace isolation flags, allowing fine-grained control over which namespaces the new process inherits versus creates fresh. Unlike fork() which creates processes in the same namespaces, clone() enables selective namespace isolation during process creation.

Pivot_root is a system call operation that atomically changes the filesystem root directory and moves the old root to a specified location. This operation is essential for container filesystem isolation because it ensures the container process cannot access the original host filesystem tree after the switch.

Zombie reaping refers to the process of collecting exit status information from terminated child processes to prevent them from remaining as zombie processes in the process table. In PID namespaces, the init process (PID 1) becomes responsible for reaping all orphaned processes within the namespace.

Namespace Types and Isolation Mechanisms

Each namespace type provides isolation for a specific category of system resources, creating independent views for processes within different namespaces.

PID namespace isolates the process ID numbering system, allowing processes in different PID namespaces to have overlapping process IDs. The first process created in a PID namespace becomes PID 1 within that namespace, regardless of its PID in the parent namespace. This enables containers to have their own process tree starting from PID 1.

Mount namespace isolates the filesystem mount table, allowing processes in different mount namespaces to have different views of the filesystem hierarchy. Changes to mounts within a mount namespace do not affect other namespaces, enabling containers to have private filesystem layouts.

Network namespace isolates the network stack including network interfaces, routing tables, firewall rules, and network statistics. Each network namespace has its own loopback interface and can contain different sets of network devices, enabling complete network isolation between containers.

User namespace isolates user and group ID mappings, allowing processes to have different effective user IDs inside versus outside the namespace. This enables containers to run as root within their namespace while mapping to unprivileged users on the host system.

UTS namespace isolates system hostname and domain name, allowing different containers to have different hostnames without affecting the host system or other containers.

IPC namespace isolates System V IPC objects (message queues, semaphore sets, shared memory segments) and POSIX message queues, ensuring that containers cannot interfere with each other's inter-process communication mechanisms.

Container Configuration and Management Terms

Container runtime systems require structured configuration and management interfaces to coordinate the creation and lifecycle of isolated environments.

Container configuration refers to the structured specification that defines all parameters for creating a container instance, including filesystem paths, resource limits, network settings, and namespace options. This configuration serves as the blueprint for constructing the isolated environment.

Container instance represents an active container with running processes, created namespaces, and allocated resources. The instance maintains references to namespace file descriptors and cleanup information required for proper resource management.

Namespace handles are file descriptor references to active namespaces that enable operations on existing namespaces and ensure proper cleanup when containers terminate. These handles prevent namespaces from disappearing while still in use and enable tools to enter existing namespaces.

Cleanup list is a mechanism for tracking resource cleanup functions that must be executed in specific order during container shutdown or failure recovery. The cleanup list ensures that partially created containers release all allocated resources properly.

Filesystem and Mount Management Terms

Container filesystem isolation relies on sophisticated mount manipulation and filesystem virtualization techniques.

Mount propagation controls how mount and unmount events spread between different mount namespaces. Private propagation prevents mount events from leaking between namespaces, while shared propagation allows coordination when needed.

Bind mount creates a mount that makes an existing directory or file accessible at an additional location in the filesystem tree. Bind mounts enable containers to access specific host directories while maintaining overall filesystem isolation.

Essential filesystems are kernel virtual filesystems like /proc, /sys, and /dev that provide interfaces to kernel functionality and device access. Containers typically mount their own instances of these filesystems to maintain isolation while providing necessary system interfaces.

Filesystem isolation prevents containers from accessing parts of the host filesystem tree outside their designated rootfs directory. This isolation is achieved through mount namespaces and careful mount table configuration.

Old root refers to the previous root filesystem location after a pivot_root operation has switched to a new root. The old root is typically unmounted and removed to complete the filesystem switch.

Network Infrastructure Terms

Container networking involves creating isolated network stacks and establishing connectivity between containers and external networks.

Veth pair is a virtual ethernet device pair where packets sent to one end appear on the other end, enabling communication between different network namespaces. One end typically remains in the host namespace while the other moves to the container namespace.

Bridge networking connects multiple veth pairs to a virtual bridge device, enabling communication between multiple containers and providing a common broadcast domain for inter-container networking.

NAT (Network Address Translation) enables containers with private IP addresses to communicate with external networks by translating between private container addresses and public host addresses.

Resource Control and Enforcement Terms

Cgroups provide fine-grained resource control and enforcement mechanisms for container workloads.

Resource controllers are cgroup subsystems that manage specific resource types such as memory, CPU time, process counts, or I/O bandwidth. Each controller implements accounting, limiting, and priority mechanisms for its resource type.

Memory controller is the cgroup controller that tracks memory usage and enforces memory limits for groups of processes. It can limit various types of memory including RSS, page cache, and kernel memory allocations.

CPU controller is the cgroup controller that manages CPU time allocation through quota and period mechanisms. It can enforce both absolute CPU limits and relative priority through shares and bandwidth controls.

Process controller is the cgroup controller that limits the number of processes and threads that can exist within a cgroup. This prevents fork bombs and ensures fair process allocation across containers.

OOM killer is the kernel mechanism that terminates processes when memory is exhausted and no more memory can be freed through normal reclaim. In containers, the OOM killer operates within memory cgroup boundaries.

CPU quota specifies the maximum amount of CPU time a cgroup can consume within a given scheduling period. Quotas are expressed in microseconds and enforce hard limits on CPU usage.

Hierarchical cleanup is a cleanup strategy that ensures processes are terminated before their containing cgroups are removed, preventing resource leaks and ensuring proper kernel resource release.

Error Handling and Debugging Terms

Container systems must handle various failure modes and provide debugging capabilities for troubleshooting isolation issues.

Container startup sequence refers to the step-by-step process of creating all required namespaces, setting up resource limits, configuring networking, and launching the container process. Each step has dependencies and specific error handling requirements.

Container cleanup sequence is the proper teardown order for releasing container resources including terminating processes, unmounting filesystems, removing network interfaces, and cleaning up cgroups. Cleanup order is critical to avoid resource leaks.

Error propagation describes how errors in one container component affect other components during setup or operation. For example, mount namespace failures can prevent network namespace setup if shared directories are required.

Parent-child synchronization coordinates actions between the host process creating a container and the container process being created. Synchronization ensures setup steps occur in the correct order across namespace boundaries.

Execution context refers to whether operations occur within the host namespace context or within container namespace contexts. Many operations must occur in specific execution contexts to be effective.

Container Lifecycle and State Management Terms

Container systems must track and manage the lifecycle of container instances from creation through termination.

Dependency relationships describe how different container components rely on each other for proper operation. For example, network namespace setup depends on successful PID namespace creation for process coordination.

Resource exhaustion occurs when system resources such as memory, CPU, or process slots are depleted, potentially affecting container operation. Container systems must handle exhaustion gracefully and provide appropriate error reporting.

Partial cleanup describes situations where some cleanup operations succeed while others fail during container shutdown. Systems must track cleanup state and retry failed operations to prevent resource leaks.

Cleanup ordering dependencies define the sequence requirements for resource cleanup operations. For example, processes must be terminated before their cgroups can be removed, and network interfaces must be removed before network namespaces can be destroyed.

Testing and Verification Terms

Container implementations require comprehensive testing to verify isolation properties and resource enforcement.

Isolation verification tests validate that namespace boundaries actually prevent processes from accessing resources outside their designated namespaces. These tests ensure that isolation mechanisms work as intended.

Resource limit testing verifies that cgroup controllers prevent containers from exceeding their configured resource allocations. Testing must validate both soft limits and hard enforcement mechanisms.

Milestone checkpoints are concrete verification steps that validate successful implementation of specific container features. Each checkpoint tests a specific aspect of isolation or resource control.

PID namespace isolation testing verifies that process IDs are properly isolated between container and host, and that the container init process correctly handles its responsibilities as PID 1.

Mount namespace isolation testing verifies that filesystem changes within containers do not affect the host filesystem and that containers cannot access unauthorized host directories.

Network namespace isolation testing verifies that container network interfaces are isolated from host interfaces and that network configuration changes within containers do not affect host networking.

Memory limit enforcement testing verifies that memory controllers prevent containers from exceeding configured memory limits and that OOM conditions are handled appropriately.

CPU limit enforcement testing verifies that CPU controllers restrict container CPU usage to configured percentages and that CPU quota mechanisms work correctly.

Process limit enforcement testing verifies that process controllers prevent containers from creating more processes than configured limits allow.

Advanced Container Features Terms

Beyond basic isolation, container systems can implement additional features for image management and orchestration.

Container image layers are read-only filesystem layers that stack together to form a complete container filesystem. Layers enable sharing of common components between different container images.

Overlay filesystem is a union filesystem that combines multiple read-only layers with a read-write layer to create a single directory tree. Overlay filesystems enable efficient copy-on-write semantics for container filesystems.

Copy-on-write is a filesystem optimization where files are shared between containers until modified, at which point they are copied to a container-specific writable layer. This reduces storage usage and improves container startup performance.

Content-addressable storage is a storage system where data is accessed using cryptographic hashes of content rather than location-based paths. This enables deduplication and integrity verification for container image layers.

Service discovery provides mechanisms for containers to locate and communicate with other services without hardcoded network addresses. Service discovery adapts to dynamic container deployment and network changes.

Container orchestration coordinates multiple containers as a cohesive application, handling deployment, scaling, networking, and failure recovery across multiple container instances.

Deployment specification is a configuration document that describes the desired state of a multi-container application including container images, resource requirements, networking, and dependencies.

Control loop is a continuous process that compares desired application state with actual running state and takes corrective actions to reconcile differences. Control loops enable declarative container management.

Load balancing distributes incoming requests across multiple container instances to improve performance and availability. Load balancing requires service discovery and health monitoring capabilities.

Inter-container communication encompasses networking mechanisms that allow containers to communicate with each other while maintaining security boundaries. This includes service meshes, overlay networks, and service discovery.

Service registry maintains a database of available services and their current network endpoints, enabling dynamic service discovery as containers start, stop, and move between hosts.

Container lifecycle management coordinates container startup, health monitoring, failure detection, restart policies, and graceful shutdown across the entire container lifecycle.

Dependency management ensures containers start in the correct order based on service dependencies and waits for required services to become available before starting dependent containers.

Hierarchical resource allocation implements multi-level resource distribution from orchestration systems down to individual containers, ensuring fair resource sharing and preventing resource conflicts.

Implementation Guidance

Container systems involve complex interactions between kernel features and user-space management tools. Understanding the terminology provides a foundation for implementing these systems effectively.

Key Terminology Categories

Category	Core Terms	Advanced Terms
Isolation	namespace, process isolation, PID namespace, mount namespace	user namespace, UTS namespace, IPC namespace
Resource Control	cgroups, memory controller, CPU controller, process controller	hierarchical resource allocation, resource exhaustion
Networking	network namespace, veth pair, bridge networking	service discovery, load balancing
Filesystem	pivot_root, bind mount, essential filesystems	overlay filesystem, copy-on-write, content-addressable storage
Lifecycle	container startup sequence, cleanup sequence, error propagation	container orchestration, deployment specification, control loop
Testing	isolation verification, resource limit testing, milestone checkpoints	integration testing, performance testing

Critical Distinction: Isolation vs Control

Understanding the distinction between isolation mechanisms (namespaces) and control mechanisms (cgroups) is fundamental to container implementation:

- **Namespaces change what processes can see** - they provide different views of system resources
- **Cgroups change what processes can do** - they limit and control resource consumption
- **Both mechanisms work together** to create complete container isolation

Container vs Virtual Machine Terminology

Container terminology often differs from virtual machine terminology, reflecting different architectural approaches:

Container Term	VM Equivalent	Key Difference
namespace	virtual hardware	shares kernel, isolates view
cgroups	resource allocation	dynamic limits, shared resources
container image	VM image	layered, content-addressable
container runtime	hypervisor	process-based, not hardware virtualization

Common Terminology Mistakes

⚠️ Pitfall: Confusing "container" with "containerization"

- **Container** refers to a specific running instance with isolated namespaces
- **Containerization** refers to the general practice of packaging applications
- **Fix**: Use "container instance" for running containers, "container image" for static packages

⚠️ Pitfall: Using "Docker" and "container" interchangeably

- **Docker** is a specific container runtime implementation
- **Container** refers to the general Linux kernel isolation technology
- **Fix**: Use "container runtime" or "container engine" for implementations, "container" for the isolated process

⚠️ Pitfall: Confusing "mount namespace" with "filesystem"

- **Mount namespace** isolates the mount table (what's mounted where)
- **Filesystem** refers to the actual data storage format
- **Fix**: Mount namespaces control mount points, filesystems store data

Debugging Terminology Usage

When debugging container issues, precise terminology helps communicate problems effectively:

Symptom	Imprecise Description	Precise Description
Process visible outside container	"Container not isolated"	"PID namespace isolation failure"
Cannot access host directory	"Filesystem broken"	"Mount namespace bind mount missing"
Network not working	"Container networking broken"	"Network namespace veth pair misconfigured"
Container using too much memory	"Resource problem"	"Memory controller limit not enforced"

This precise terminology enables faster diagnosis and resolution of container implementation issues.