

Process Abstraction and Isolation in Linux

Hui Chen ^a

^aCUNY Brooklyn College

September 30, 2026

Outline

- 1 From Abstraction to Isolation
- 2 `fork()` and `clone()`
- 3 Namespaces and Isolation
- 4 Project Scope
- 5 The Design Principle
- 6 Next

Outline

- 1 From Abstraction to Isolation
- 2 `fork()` and `clone()`
- 3 Namespaces and Isolation
- 4 Project Scope
- 5 The Design Principle
- 6 Next

Process Isolation

- ▶ **Process isolation:** a single execution context (or a group) with a *restricted* view of the system.
- ▶ **Interface to the kernel mechanism:** the `clone()` system call, whose flags select what is shared and what is isolated.

Question: how does the kernel construct a different *view of the machine* for different processes?

Processes as Abstractions

Recall: a process consists of

- ▶ an **execution context** (registers, program counter, stack pointer), and
- ▶ an **address space** (text, data, heap, stack).

But from the user's perspective, a process is also an **abstraction of the machine itself**:

- ▶ it has “its own” CPU,
- ▶ it has “its own” memory,
- ▶ it has “its own” view of system resources (process IDs, hostname, mounts, network, ...).

Key idea: these views are *constructed by the kernel*, not given by the hardware. The kernel can construct different views for different processes.

Outline

- 1 From Abstraction to Isolation
- 2 **fork() and clone()**
- 3 Namespaces and Isolation
- 4 Project Scope
- 5 The Design Principle
- 6 Next

fork() vs. clone()

- ▶ `fork()` creates a new process that is a copy of the caller. The child gets its own address space and its own PCB, but the *same view of the system* (same PIDs, same hostname, same mounts, same network).
- ▶ `clone()` is the general form. It takes **flags** that select what the child *shares* with the parent and what it *isolates* from it.
- ▶ The same syscall therefore creates:
 - ▶ a **process**, with `clone(SIGCHLD)`;
 - ▶ an **isolated process** (a container), with `clone(CLONE_NEWPID | CLONE_NEWNS | ...)`;
 - ▶ and more ...

Outline

- 1 From Abstraction to Isolation
- 2 `fork()` and `clone()`
- 3 Namespaces and Isolation**
- 4 Project Scope
- 5 The Design Principle
- 6 Next

Linux Namespaces: Isolated Views of the System

A **namespace** wraps a global system resource and gives each process in the namespace its own view of it.

Namespace	Flag	Isolates
PID	CLONE_NEWPID	Process IDs
Mount	CLONE_NEWNS	Filesystem mounts
Network	CLONE_NEWNET	Network interfaces, routes
UTS	CLONE_NEWUTS	Hostname, domain name
IPC	CLONE_NEWIPC	SysV IPC, POSIX message queues
User	CLONE_NEWUSER	User and group IDs

Key idea: the kernel already had these global resources. Namespaces make them *per-process* instead of *system-wide*.

How the Kernel Represents a Process

The kernel represents each process with a data structure — on Linux, a `task_struct` (the general OS term is **Process Control Block**, or PCB).

The PCB holds more than registers and a program counter. It also holds **pointers** to the process's view of the system:

- ▶ `mm_struct` — the process's address space;
- ▶ `fs_struct` — filesystem context (root, current directory);
- ▶ `nsproxy` — the bundle of namespace pointers (PID, mount, network, UTS, IPC, user);
- ▶ `files_struct` — open file descriptors;
- ▶ scheduling and accounting information.

Key idea: the PCB is not just an execution context. It is also the *root* of the process's view of the system.

How Isolation Is Enforced

Every syscall that touches a shared resource resolves it *through the calling process's own PCB pointers*.

Syscall	Global resource	Resolved via
getpid()	process IDs	nsproxy->pid_ns
open()	filesystem root	fs_struct->root
sethostname()	hostname	nsproxy->uts_ns
socket()	network stack	nsproxy->net_ns

How clone(CLONE_NEWPID) works:

- ▶ the kernel allocates a new pid_namespace object;
- ▶ it creates a fresh nsproxy for the child that points at the new namespace;
- ▶ the child's task_struct points at the new nsproxy.

So: isolation is not a separate mechanism. It is what happens when the process's pointers point at different objects than everyone else's.

From Namespaces to a Container

A minimal container is a process with a restricted view:

- ▶ `clone()` with namespace flags gives the child its own PIDs, hostname, mounts, network, ...
- ▶ `chroot()` or `pivot_root()` gives the child its own filesystem root. Without a separate root, the containerized process would see the host's `/` and could access files outside the container.
- ▶ **cgroups** limit how much CPU, memory, and I/O the child can use.

Mental model: a container is not a virtual machine. It is a process (or group of processes) whose kernel-provided view of the system has been narrowed.

This is why containers share the host kernel: they are ordinary processes, just with restricted namespaces, mounts, and limits.

Outline

- 1 From Abstraction to Isolation
- 2 `fork()` and `clone()`
- 3 Namespaces and Isolation
- 4 Project Scope**
- 5 The Design Principle
- 6 Next

Project Scope: A Minimal Container

In the project assignment, you are asked to build a **minimal container launcher**:

- ▶ Use `clone()` to create a child in at least two new namespaces (e.g., PID and UTS).
- ▶ Demonstrate that the child sees a different view from the parent:
 - ▶ the child is PID 1 in its own PID namespace;
 - ▶ the child can set its own hostname without affecting the parent;
 - ▶ (optional) the child sees its own `/proc`.
- ▶ Explain, in writing, what each flag does and what would happen without it.

Not required for the minimum: image pulling, overlay filesystems, networking, resource limits. Some of these appear as optional tasks in the project page.

Outline

- 1 From Abstraction to Isolation
- 2 `fork()` and `clone()`
- 3 Namespaces and Isolation
- 4 Project Scope
- 5 The Design Principle**
- 6 Next

The Design Principle

- ▶ A process is not just “a program running.” It is a kernel-provided abstraction: an execution context, an address space, and a set of views of system resources.
- ▶ The kernel can *narrow* those views by pointing the process’s PCB at different objects.
- ▶ When it narrows them enough, we call the result a **container**.
- ▶ Namespaces, chroot, and cgroups are the *mechanisms*. The *principle* is isolation through controlled sharing.
- ▶ The same principle applies in the other direction too: processes can also *share* more rather than isolate more. We will see this when we discuss threads.

Outline

- 1 From Abstraction to Isolation
- 2 `fork()` and `clone()`
- 3 Namespaces and Isolation
- 4 Project Scope
- 5 The Design Principle
- 6 Next

Next

- ▶ The project assignment asks you to build a minimal container launcher using `clone()` and namespaces.
- ▶ After the project, we will discuss **threads**, where the same `clone()` mechanism is used to *share* rather than isolate.