

GOPHERCON AFRICA 2026

Building a Container Runtime from Scratch

Namespaces, cgroups and a root filesystem, in Go

AS

Ajitem Sahasrabuddhe

Automattic · @asahasrabuddhe

```
package namespace
```

```
import (  
    "fmt"  
    "os"  
    "syscall"  
)
```

```
// NewNamespace creates a new Linux namespace  
// using the clone(2) system call with the  
// provided flags.
```

```
func NewNamespace(flags uintptr) error {  
    cmd := exec.Command("/proc/self/exe", "i
```

```
    cmd.SysProcAttr = &syscall.SysProcAttr{  
        Cloneflags: flags |  
            syscall.CLONE_NEWUTS |  
            syscall.CLONE_NEWPID |  
            syscall.CLONE_NEWNS,
```

```
}
```



Ajitem Sahasrabuddhe

Automattic

github.com/asahasrabuddhe

linkedin.com/in/ajitem

Go

Containers

Linux

Rust

- Maintainer of [urfave/cli](#), the Go package for building command line tools. 24k stars, used by over 90,000 repositories on GitHub.
- Building [wtfpdf.co](#) - free, private PDF tools that run in the browser and never upload your files. It is how I am learning Rust.

01

CHAPTER 01

What a container **is**

Before we build one.

What is a container runtime?

What is a container runtime?

- The software that runs containers on a host: it takes an image and a command, and turns them into a running, isolated process.

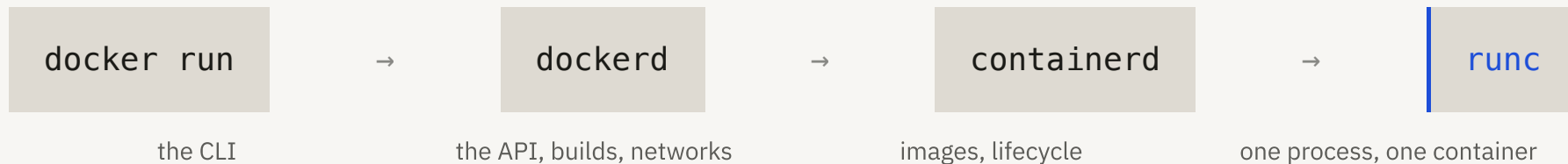
What is a container runtime?

- The software that runs containers on a host: it takes an image and a command, and turns them into a running, isolated process.
- Pulling images and managing their lifecycle sit on top of it.

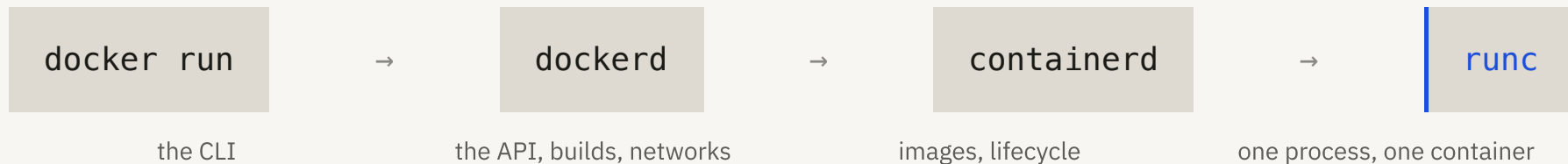
What is a container runtime?

- The software that runs containers on a host: it takes an image and a command, and turns them into a running, isolated process.
- Pulling images and managing their lifecycle sit on top of it.
- **Docker is not a container runtime.** It is a container engine, and it ships one.

Docker is not a runtime



Docker is not a runtime



Everything left of **runc** is management. Today we write the box on the right.

What is a container, really?

What is a container, really?

A process

What is a container, really?

A process

in an isolated environment namespaces and cgroups

What is a container, really?

A process

in an isolated environment namespaces and cgroups

with a root filesystem

The filesystem is **what** to run.
Namespaces and cgroups are **how**.

The filesystem is **what** to run.
Namespaces and cgroups are **how**.

cgroups set the budget. Namespaces set the view.

What are cgroups?

What are cgroups?

- A kernel feature for accounting and limiting resources.

What are cgroups?

- A kernel feature for accounting and limiting resources.
- Think of them as **budgets** for processes.

What are cgroups?

- A kernel feature for accounting and limiting resources.
- Think of them as **budgets** for processes.
- Memory, CPU, I/O, and how many processes a group may have.

What are cgroups?

- A kernel feature for accounting and limiting resources.
- Think of them as **budgets** for processes.
- Memory, CPU, I/O, and how many processes a group may have.
- Limits apply to a group, not to a process, and the kernel enforces them.

How cgroups work

```
/sys/fs/cgroup          root cgroup
├── system.slice
│   ├── sshd.service
│   └── nginx.service
└── mycontainer
    ├── 3f9c2a1b7e04
    └── 67557a8fc599
```



How cgroups work

```
/sys/fs/cgroup          root cgroup
├── system.slice
│   ├── sshd.service
│   └── nginx.service
└── mycontainer
    ├── 3f9c2a1b7e04
    └── 67557a8fc599
```



A tree of groups. Each process is in exactly one.

How cgroups work

```
/sys/fs/cgroup          root cgroup
├── system.slice
│   ├── sshd.service
│   └── nginx.service
└── mycontainer
    ├── 3f9c2a1b7e04
    └── 67557a8fc599
```



A tree of groups. Each process is in exactly one.

cgroup v2 is a [filesystem](#). A group is a directory, its limits are files.

How cgroups work

```
/sys/fs/cgroup          root cgroup
├── system.slice
│   ├── sshd.service
│   └── nginx.service
└── mycontainer
    ├── 3f9c2a1b7e04
    └── 67557a8fc599
```



A tree of groups. Each process is in exactly one.

cgroup v2 is a [filesystem](#). A group is a directory, its limits are files.

`memory.max`, `cpu.max`, `pids.max`. Write a number, and the kernel holds every process in the directory to it.

What are namespaces?

What are namespaces?

- A kernel feature for isolating what a process can see.

What are namespaces?

- A kernel feature for isolating what a process can see.
- Each namespace gives its processes their own view of one part of the system.

What are namespaces?

- A kernel feature for isolating what a process can see.
- Each namespace gives its processes their own view of one part of the system.
- Together they make a process feel like it has the machine to itself.

Namespaces

Namespace

Flag

pid

CLONE_NEWPID

mnt

CLONE_NEWNS

user

CLONE_NEWUSER

uts

CLONE_NEWUTS

ipc

CLONE_NEWIPC

net

CLONE_NEWNET

cgroup

CLONE_NEWCGROUP

Gives the process its own

process tree, starting at PID 1

mount table, and so its own /

user IDs: root inside, you outside

hostname

shared memory, semaphores, queues

interfaces, routes, ports

view of the cgroup tree

02

CHAPTER 02

Five namespaces, **then a budget**

Two processes, always

```
cmd := exec.Command("/proc/self/exe", append([]string{ChildCommand, root}, argv...)...)
cmd.Stdin, cmd.Stdout, cmd.Stderr = os.Stdin, os.Stdout, os.Stderr
cmd.SysProcAttr = &syscall.SysProcAttr{
    // The whole of 01-pid: the child gets its own process IDs, and the
    // first process in it is PID 1.
    Cloneflags: syscall.CLONE_NEWPID,
}
```



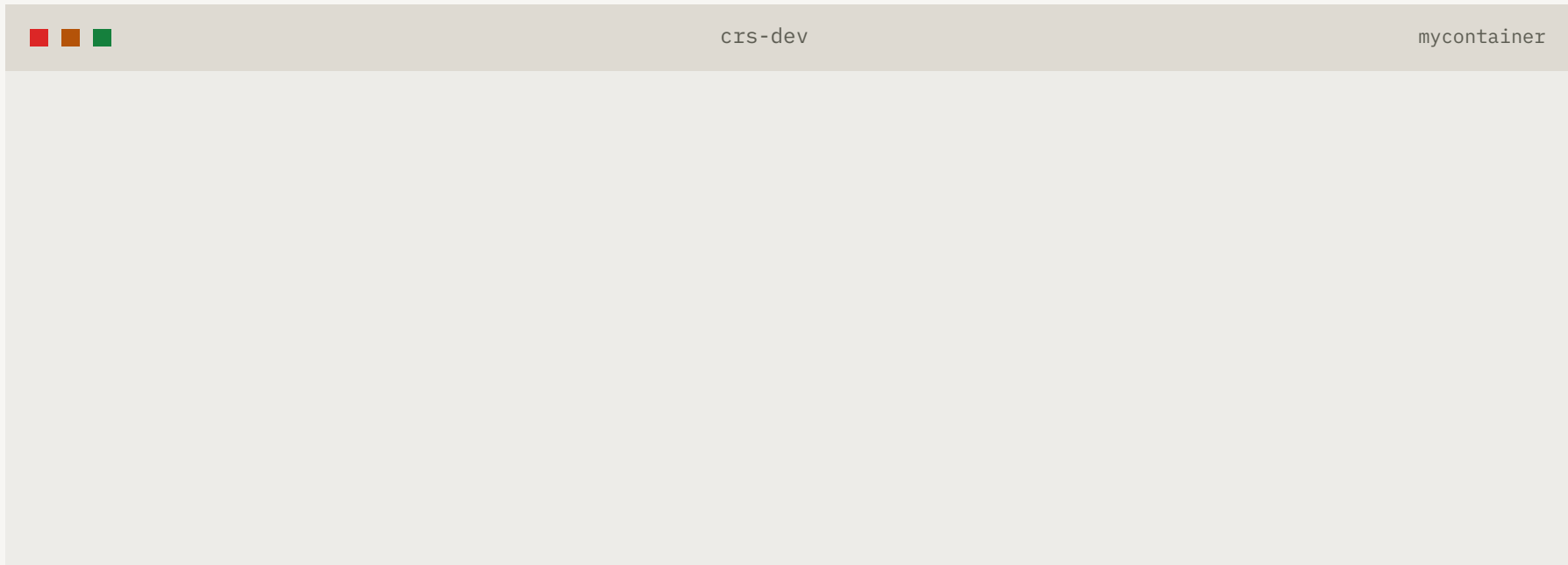
Two processes, always

```
cmd := exec.Command("/proc/self/exe", append([]string{ChildCommand, root}, argv...)...)
cmd.Stdin, cmd.Stdout, cmd.Stderr = os.Stdin, os.Stdout, os.Stderr
cmd.SysProcAttr = &syscall.SysProcAttr{
    // The whole of 01-pid: the child gets its own process IDs, and the
    // first process in it is PID 1.
    Cloneflags: syscall.CLONE_NEWPID,
}
```



Namespaces are set at `clone(2)`, for the process being created. The Go runtime is several threads before `main` runs. So the parent starts its own binary again, `/proc/self/exe child`, with the flag set, and that child is the container. `runc init` is the same trick.

01-pid: bash is PID 1



A PID namespace changes the numbers.

`/proc` is what `ps` reads.

And `/proc` is still the host's. That is the next step.

02-mnt: `pivot_root`, in order

1. `mount("", "/", MS_REC|MS_PRIVATE)`
2. `mount(rootfs, rootfs, MS_BIND|MS_REC)`
3. Mount `/proc` and `/dev`
4. `chdir(rootfs), pivot_root(".", ".old_root"), chdir("/")`
5. `umount2(".old_root", MNT_DETACH)`

02-mnt: `pivot_root`, in order

1. `mount("", "/", MS_REC|MS_PRIVATE)`
2. `mount(rootfs, rootfs, MS_BIND|MS_REC)`
3. Mount `/proc` and `/dev`
4. `chdir(rootfs), pivot_root(".", ".old_root"), chdir("/")`
5. `umount2(".old_root", MNT_DETACH)`

Every step is there because the one after it fails without it. A test pins the order.

02-mnt: a root of its own



03-user: root inside, you outside

```
cmd := exec.Command("/proc/self/exe", append([]string{ChildCommand, root}, argv...)...)
cmd.Stdin, cmd.Stdout, cmd.Stderr = os.Stdin, os.Stdout, os.Stderr
cmd.SysProcAttr = &syscall.SysProcAttr{
    // 01-pid: its own process IDs. 02-mnt: its own mount table.
    // 03-user: its own user IDs, so no sudo.
    Cloneflags: syscall.CLONE_NEWPID | syscall.CLONE_NEWNS | syscall.CLONE_NEWUSER,
    // One line each: root inside is the image's owner outside.
    UidMappings: []syscall.SysProcIDMap{{ContainerID: 0, HostID: uid, Size: 1}},
    GidMappings: []syscall.SysProcIDMap{{ContainerID: 0, HostID: gid, Size: 1}},
}
```



03-user: `id` says 0



04-uts: a hostname of its own

```
// Only this container sees it, because of CLONE_NEWUTS.  
if err := syscall.Sethostname([]byte(id)); err != nil {  
    return fmt.Errorf("sethostname: %w", err)  
}
```



05-ipc: shared memory, unshared



06-cgroups: limits are files

```
files := make(map[string]string)
if l.Memory > 0 {
    files["memory.max"] = strconv.FormatInt(l.Memory, 10)
    // Without this a process at memory.max keeps going into swap, and
    // the limit is a suggestion. Docker sets swap to the same value as
    // memory when you only pass --memory; zero is the honest version.
    files["memory.swap.max"] = "0"
}
if l.CPUs > 0 {
    files["cpu.max"] = cpuMax(l.CPUs)
}
if l.Pids > 0 {
    files["pids.max"] = strconv.Itoa(l.Pids)
}
```



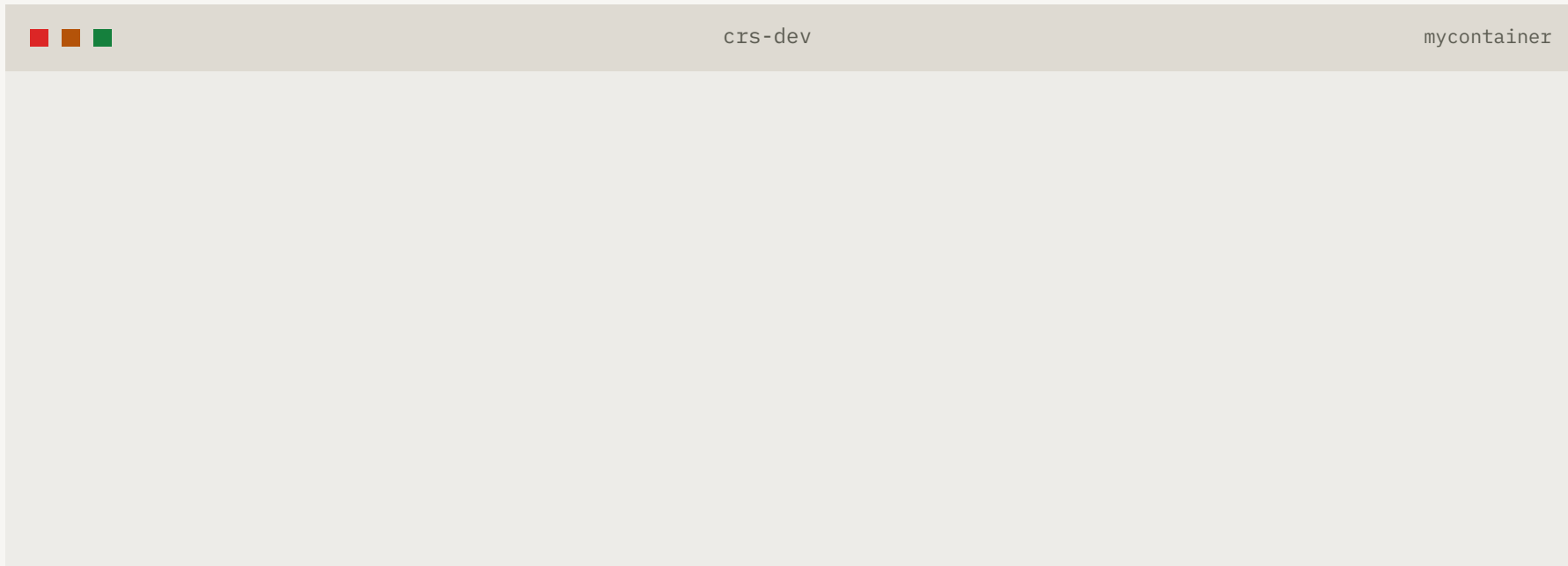
06-cgroups: limits are files

```
files := make(map[string]string)
if l.Memory > 0 {
    files["memory.max"] = strconv.FormatInt(l.Memory, 10)
    // Without this a process at memory.max keeps going into swap, and
    // the limit is a suggestion. Docker sets swap to the same value as
    // memory when you only pass --memory; zero is the honest version.
    files["memory.swap.max"] = "0"
}
if l.CPUs > 0 {
    files["cpu.max"] = cpuMax(l.CPUs)
}
if l.Pids > 0 {
    files["pids.max"] = strconv.Itoa(l.Pids)
}
```



Written into `/sys/fs/cgroup/mycontainer/<id>`. The child is born inside it with `CLONE_INT0_CGROUP`, and `CLONE_NEWCGROUP` lets it read its own limits.

06-cgroups: hitting the limit



That is a container



That is a container



1 clone3 · 1 sethostname · 12 mount · 1 pivot_root · 1 umount2 · 27 capability drops · 3 execve

Five things to take home

Five things to take home

1. A container is a process, in an isolated environment, with a root filesystem.

Five things to take home

1. A container is a process, in an isolated environment, with a root filesystem.
2. The filesystem is what to run. Namespaces and cgroups are how.

Five things to take home

1. A container is a process, in an isolated environment, with a root filesystem.
2. The filesystem is what to run. Namespaces and cgroups are how.
3. Each namespace is one flag, and covers only what it covers. A PID namespace alone still reads the host's `/proc`.

Five things to take home

1. A container is a process, in an isolated environment, with a root filesystem.
2. The filesystem is what to run. Namespaces and cgroups are how.
3. Each namespace is one flag, and covers only what it covers. A PID namespace alone still reads the host's `/proc`.
4. cgroup v2 is a directory, and limits are files the kernel enforces.

Five things to take home

1. A container is a process, in an isolated environment, with a root filesystem.
2. The filesystem is what to run. Namespaces and cgroups are how.
3. Each namespace is one flag, and covers only what it covers. A PID namespace alone still reads the host's `/proc`.
4. cgroup v2 is a directory, and limits are files the kernel enforces.
5. In Go, a container is your own binary started again through `/proc/self/exe`, because namespaces are set at `clone`.

Five things to take home

1. A container is a process, in an isolated environment, with a root filesystem.
2. The filesystem is what to run. Namespaces and cgroups are how.
3. Each namespace is one flag, and covers only what it covers. A PID namespace alone still reads the host's `/proc`.
4. cgroup v2 is a directory, and limits are files the kernel enforces.
5. In Go, a container is your own binary started again through `/proc/self/exe`, because namespaces are set at `clone`.

Docker, containerd and runc do far more than this. They do not do anything different.

// THAT'S A WRAP

Thank You

The slides and the code are at:

github.com/asahasrabuddhe/gca-2026-crs-code

[gca-2026-crs-slides](#)

[gca-2026-crs-code](#)

[@asahasrabuddhe](#)



scan for slides + code