

CONTAINERDAYS 2026 · SINGAPORE · WORKSHOP

Build Your Own Container Runtime in Go

Namespaces, cgroups, capabilities and a pivoted root. Forty-five minutes, five tests to make green.

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,
```

```
}
```

Before we start

1. The Multipass VM from `docs/setup.md`. On your machine:

```
git clone https://github.com/asahasrabuddhe/cds-2026-singapore-crs-code.git crs
multipass mount crs crs:/home/ubuntu/crs
```



2. In the VM, `multipass shell crs`, then:

```
cd ~/crs && ./scripts/check.sh && make build && go test ./...
```



3. Your editor on the `crs` folder, on your machine.

Before we start

1. The Multipass VM from `docs/setup.md`. On your machine:

```
git clone https://github.com/asahasrabuddhe/cds-2026-singapore-crs-code.git crs
multipass mount crs crs:/home/ubuntu/crs
```



2. In the VM, `multipass shell crs`, then:

```
cd ~/crs && ./scripts/check.sh && make build && go test ./...
```



3. Your editor on the `crs` folder, on your machine.

Six packages fail. Five are the steps; the sixth goes green with step 4.

A container is a process wearing a costume

Step	The costume	You write	Minutes
1	Namespaces: what it can see	<code>CloneFlags()</code>	5
2	Rootfs: where it lives	the <code>pivot_root</code> sequence	10
3	cgroups: what it can use	<code>Files()</code> , <code>cpuMax()</code>	7
4	Capabilities: what it can do	<code>Default()</code>	5
5	Init: what PID 1 owes	<code>reap()</code>	7

Each step: I explain the idea, you write it, the test tells you when it is right.

A container is a process wearing a costume

Step	The costume	You write	Minutes
1	Namespaces: what it can see	<code>CloneFlags()</code>	5
2	Rootfs: where it lives	the <code>pivot_root</code> sequence	10
3	cgroups: what it can use	<code>Files()</code> , <code>cpuMax()</code>	7
4	Capabilities: what it can do	<code>Default()</code>	5
5	Init: what PID 1 owes	<code>reap()</code>	7

Each step: I explain the idea, you write it, the test tells you when it is right.

Stuck? The answer is on the `final` branch. `git restore --source origin/final internal/rootfs/rootfs.go` , and carry on.

Namespaces are bits on a clone

A namespace is a flag on `clone(2)`. It applies to the new process, never to the one asking.

```
// A process.  
cmd := exec.Command("/bin/sh")  
cmd.Stdin, cmd.Stdout = os.Stdin, os.Stdout  
cmd.Run()
```



Namespaces are bits on a clone

A namespace is a flag on `clone(2)`. It applies to the new process, never to the one asking.

The kernel has seven: PID, mount, UTS, IPC, network, cgroup and user. One bit each.

```
// A process in new namespaces.
cmd := exec.Command("/bin/sh")
cmd.SysProcAttr = &syscall.SysProcAttr{
    Cloneflags: syscall.CLONE_NEWUTS |
                syscall.CLONE_NEWPID,
}
cmd.Stdin, cmd.Stdout = os.Stdin, os.Stdout
cmd.Run()
```



Namespaces are bits on a clone

A namespace is a flag on `clone(2)`. It applies to the new process, never to the one asking.

The kernel has seven: PID, mount, UTS, IPC, network, cgroup and user. One bit each.

The container has to set itself up from inside, so the runtime starts itself again as the child.

Your job: turn the spec's fields into those bits, and hand them to the clone.

```
// The runtime, started again inside them.
cmd := exec.Command("/proc/self/exe", "child")
cmd.SysProcAttr = &syscall.SysProcAttr{
    Cloneflags: syscall.CLONE_NEWUTS |
        syscall.CLONE_NEWPID,
}
cmd.Stdin, cmd.Stdout = os.Stdin, os.Stdout
cmd.Run()

// In the child, PID 1 in the new namespace:
// the hostname, the rootfs, the limits, then
// the command.
```



Step 1 • Namespaces: what it can see

DO THIS

- 01 Open `internal/namespace/namespace.go` and find `TODO(step 1)`.
- 02 The PID flag is written for you. Add the other six the same way.
- 03 Run `go test ./internal/namespace/` until it passes.
- 04 Open `internal/container/container_linux.go`, find `TODO(step 1)`, and add one line.

HINTS

CHECK IT

```
make build && sudo ./mycontainer run -ns pid,uts -hostname box ubuntu /bin/sh
hostname; echo $$; ls /proc          # type these inside
```



Step 1 • Namespaces: what it can see

DO THIS

- 01 Open `internal/namespace/namespace.go` and find `TODO(step 1)`.
- 02 The PID flag is written for you. Add the other six the same way.
- 03 Run `go test ./internal/namespace/` until it passes.
- 04 Open `internal/container/container_linux.go`, find `TODO(step 1)`, and add one line.

CHECK IT

```
make build && sudo ./mycontainer run -ns pid,uts -hostname box ubuntu /bin/sh
hostname; echo $$; ls /proc          # type these inside
```



HINTS

Item 2, once more for each field:

```
if s.Mount {
    f |= syscall.CLONE_NEWNS
}
```



Mount is the odd one: `CLONE_NEWNS`. The rest are their own names.

Step 1 · Namespaces: what it can see

DO THIS

- 01 Open `internal/namespace/namespace.go` and find `TODD(step 1)`.
- 02 The PID flag is written for you. Add the other six the same way.
- 03 Run `go test ./internal/namespace/` until it passes.
- 04 Open `internal/container/container_linux.go`, find `TODD(step 1)`, and add one line.

CHECK IT

```
make build && sudo ./mycontainer run -ns pid,uts -hostname box ubuntu /bin/sh
hostname; echo $$; ls /proc          # type these inside
```



HINTS

Item 2, once more for each field:

```
if s.Mount {
    f |= syscall.CLONE_NEWNS
}
```



Mount is the odd one: `CLONE_NEWNS`. The rest are their own names.

Item 4 is this line:

```
attr.Cloneflags = s.Namespaces.CloneFlags()
```



Step 1 • Namespaces: what it can see

DO THIS

- 01 Open `internal/namespace/namespace.go` and find `TODD(step 1)`.
- 02 The PID flag is written for you. Add the other six the same way.
- 03 Run `go test ./internal/namespace/` until it passes.
- 04 Open `internal/container/container_linux.go`, find `TODD(step 1)`, and add one line.

CHECK IT

```
make build && sudo ./mycontainer run -ns pid,uts -hostname box ubuntu /bin/sh
hostname; echo $$; ls /proc          # type these inside
```



`box`, a small PID, and `/proc` still lists the VM's processes. Why?

HINTS

Item 2, once more for each field:

```
if s.Mount {
    f |= syscall.CLONE_NEWNS
}
```



Mount is the odd one: `CLONE_NEWNS`. The rest are their own names.

Item 4 is this line:

```
attr.Cloneflags = s.Namespaces.CloneFlags()
```



The root is a directory, swapped in

The container's `/` is a directory on the VM's disk, swapped in by `pivot_root(2)`. The order is the lesson.

```
/          the VM's root, shared
├─ proc    the VM's processes
├─ tmp/host-was-here
└─ .../rootfs/ubuntu a directory
```



The root is a directory, swapped in

The container's `/` is a directory on the VM's disk, swapped in by `pivot_root(2)`. The order is the lesson.

Make `/` private, so nothing leaks out. Bind the image onto itself: `pivot_root` wants a mount point.

```
/
├── proc
├── tmp/host-was-here
└── .../rootfs/ubuntu
```

the VM's root, private
the VM's processes
a mount point



The root is a directory, swapped in

The container's `/` is a directory on the VM's disk, swapped in by `pivot_root(2)`. The order is the lesson.

Make `/` private, so nothing leaks out. Bind the image onto itself: `pivot_root` wants a mount point.

Mount `/proc`, `/dev` and `/sys` inside it.

```

/
├─ proc
├─ tmp/host-was-here
└─ .../rootfs/ubuntu
    ├─ proc
    ├─ dev
    └─ sys
  
```



the VM's root, private
the VM's processes

a mount point
the container's processes
a tmpfs, a few devices
read-only

The root is a directory, swapped in

The container's `/` is a directory on the VM's disk, swapped in by `pivot_root(2)`. The order is the lesson.

Make `/` private, so nothing leaks out. Bind the image onto itself: `pivot_root` wants a mount point.

Mount `/proc`, `/dev` and `/sys` inside it.

Step in and pivot. The VM's root hangs off `/.old_root`.

```
/          the image
├─ proc    the container's processes
├─ dev     a tmpfs, a few devices
├─ sys     read-only
└─ .old_root the VM's root, still there
    └─ tmp/host-was-here
```



The root is a directory, swapped in

The container's `/` is a directory on the VM's disk, swapped in by `pivot_root(2)`. The order is the lesson.

Make `/` private, so nothing leaks out. Bind the image onto itself: `pivot_root` wants a mount point.

Mount `/proc`, `/dev` and `/sys` inside it.

Step in and pivot. The VM's root hangs off `/.old_root`.

Detach it and remove it. Nothing points out.

```
/
├─ proc
├─ dev
├─ sys
└─ tmp
```



the image
the container's processes
a tmpfs, a few devices
read-only
empty

Step 2 · Rootfs: where it lives

DO THIS

- 01 Open `internal/rootfs/rootfs.go` and find `TOD0(step 2)` in `Pivot`.
- 02 The comment lists seven things. Turn each into an entry in `steps`, in that order. The first is on the right.
- 03 Run `go test ./internal/rootfs/` after each one. It prints the call it wanted next, flags and all.
- 04 When it passes, run the check below.

CHECK IT

```
make build && touch /tmp/host-was-here && sudo ./mycontainer run ubuntu /bin/sh  
ps; ls /; ls /tmp; mount # type these inside
```



HINTS

Item 2, the first entry:

```
{ "make / private", func() error {  
    return m.Mount("", "/", "",  
        syscall.MS_REC|syscall.MS_PRIVATE, "")  
}, false},
```



Step 2 · Rootfs: where it lives

DO THIS

- 01 Open `internal/rootfs/rootfs.go` and find `TODD(step 2)` in `Pivot`.
- 02 The comment lists seven things. Turn each into an entry in `steps`, in that order. The first is on the right.
- 03 Run `go test ./internal/rootfs/` after each one. It prints the call it wanted next, flags and all.
- 04 When it passes, run the check below.

CHECK IT

```
make build && touch /tmp/host-was-here && sudo ./mycontainer run ubuntu /bin/sh  
ps; ls /; ls /tmp; mount # type these inside
```



HINTS

Item 2, the first entry:

```
{ "make / private", func() error {  
    return m.Mount("", "/", "",  
        syscall.MS_REC|syscall.MS_PRIVATE, "")  
}, false},
```



Items 3 to 5: `m.Mount` with `"proc"`, `"tmpfs"`, `"sysfs"` and `"cgroup2"`, and `populateDev` after the `tmpfs`.

Step 2 · Rootfs: where it lives

DO THIS

- 01 Open `internal/rootfs/rootfs.go` and find `TOD0(step 2)` in `Pivot`.
- 02 The comment lists seven things. Turn each into an entry in `steps`, in that order. The first is on the right.
- 03 Run `go test ./internal/rootfs/` after each one. It prints the call it wanted next, flags and all.
- 04 When it passes, run the check below.

CHECK IT

```
make build && touch /tmp/host-was-here && sudo ./mycontainer run ubuntu /bin/sh  
ps; ls /; ls /tmp; mount # type these inside
```



HINTS

Item 2, the first entry:

```
{ "make / private", func() error {  
    return m.Mount("", "/", "",  
        syscall.MS_REC|syscall.MS_PRIVATE, "")  
}, false},
```



Items 3 to 5: `m.Mount` with `"proc"`, `"tmpfs"`, `"sysfs"` and `"cgroup2"`, and `populateDev` after the `tmpfs`.

Item 6: `m.Chdir`, `m.PivotRoot`, `m.Chdir`. Item 7:

Step 2 · Rootfs: where it lives

DO THIS

- 01 Open `internal/rootfs/rootfs.go` and find `TODD(step 2)` in `Pivot`.
- 02 The comment lists seven things. Turn each into an entry in `steps`, in that order. The first is on the right.
- 03 Run `go test ./internal/rootfs/` after each one. It prints the call it wanted next, flags and all.
- 04 When it passes, run the check below.

CHECK IT

```
make build && touch /tmp/host-was-here && sudo ./mycontainer run ubuntu /bin/sh  
ps; ls /; ls /tmp; mount # type these inside
```



PID 1, your shell and `ps`. The image's tree. An empty `/tmp`. Only its own mounts.

HINTS

Item 2, the first entry:

```
{ "make / private", func() error {  
    return m.Mount("", "/", "",  
        syscall.MS_REC|syscall.MS_PRIVATE, "")  
}, false},
```



Items 3 to 5: `m.Mount` with `"proc"`, `"tmpfs"`, `"sysfs"` and `"cgroup2"`, and `populateDev` after the `tmpfs`.

Item 6: `m.Chdir`, `m.PivotRoot`, `m.Chdir`. Item 7:

A limit is a number in a file

cgroup v2 is a filesystem. A group is a directory, and the kernel fills it with files.

```
$ cd /sys/fs/cgroup && mkdir box && cd box  
$ ls  
cgroup.procs  cpu.max  memory.max  pids.max  ...
```



A limit is a number in a file

cgroup v2 is a filesystem. A group is a directory, and the kernel fills it with files.

Write a number to a file and the kernel enforces it. Without the swap file, the memory limit is a suggestion.

```
$ cd /sys/fs/cgroup && mkdir box && cd box
$ echo 20 > pids.max
$ echo 67108864 > memory.max          # 64 MiB
$ echo 0 > memory.swap.max
$ echo "50000 100000" > cpu.max      # half a CPU
```



A limit is a number in a file

cgroup v2 is a filesystem. A group is a directory, and the kernel fills it with files.

Write a number to a file and the kernel enforces it. Without the swap file, the memory limit is a suggestion.

A process joins by writing its PID. The runtime's child is born inside, so its first fork counts.

Your job: which file gets which number, and the flag that puts the child inside.

```
$ cd /sys/fs/cgroup && mkdir box && cd box
$ echo 20 > pids.max
$ echo 67108864 > memory.max          # 64 MiB
$ echo 0 > memory.swap.max
$ echo "50000 100000" > cpu.max      # half a CPU
$ echo $$ > cgroup.procs
$ for i in $(seq 40); do sleep 9 & done
# fork fails once the group holds 20
```



Step 3 • cgroups: what it can use

DO THIS

- 01 Open `internal/cgroup/cgroup.go` and write `cpuMax()` at `TODO(step 3)`.
- 02 Write `Files()` just below it: one file per limit that is set.
- 03 Run `go test ./internal/cgroup/` until it passes.
- 04 Open `internal/container/container_linux.go`, find `TODO(step 3)`, and add the lines on the right.

Keep `-pids` at sixteen or more. It counts the Go init's own threads.

HINTS

Item 2 is four files: `memory.max` in bytes, `memory.swap.max` as `0` beside it, `cpu.max`, `pids.max`. A zero limit writes nothing.

CHECK IT

```
make build && sudo ./mycontainer run -pids 20 ubuntu /bin/sh
cat /sys/fs/cgroup/pids.max; for i in $(seq 40); do sleep 2 & done      # inside
```



Step 3 • cgroups: what it can use

DO THIS

- 01 Open `internal/cgroup/cgroup.go` and write `cpuMax()` at `TODD(step 3)`.
- 02 Write `Files()` just below it: one file per limit that is set.
- 03 Run `go test ./internal/cgroup/` until it passes.
- 04 Open `internal/container/container_linux.go`, find `TODD(step 3)`, and add the lines on the right.

Keep `-pids` at sixteen or more. It counts the Go init's own threads.

CHECK IT

```
make build && sudo ./mycontainer run -pids 20 ubuntu /bin/sh
cat /sys/fs/cgroup/pids.max; for i in $(seq 40); do sleep 2 & done      # inside
```



HINTS

Item 2 is four files: `memory.max` in bytes, `memory.swap.max` as `0` beside it, `cpu.max`, `pids.max`. A zero limit writes nothing.

Item 1: `cpu.max` is a quota and a period in microseconds. The period is 100000, so half a CPU is `50000 100000`.

Step 3 • cgroups: what it can use

DO THIS

- 01 Open `internal/cgroup/cgroup.go` and write `cpuMax()` at `TODD(step 3)`.
- 02 Write `Files()` just below it: one file per limit that is set.
- 03 Run `go test ./internal/cgroup/` until it passes.
- 04 Open `internal/container/container_linux.go`, find `TODD(step 3)`, and add the lines on the right.

Keep `-pids` at sixteen or more. It counts the Go init's own threads.

CHECK IT

```
make build && sudo ./mycontainer run -pids 20 ubuntu /bin/sh
cat /sys/fs/cgroup/pids.max; for i in $(seq 40); do sleep 2 & done      # inside
```



HINTS

Item 2 is four files: `memory.max` in bytes, `memory.swap.max` as `0` beside it, `cpu.max`, `pids.max`. A zero limit writes nothing.

Item 1: `cpu.max` is a quota and a period in microseconds. The period is 100000, so half a CPU is `50000 100000`.

Item 4:

```
if group != nil {
    attr.UseCgroupFD = true
    attr.CgroupFD = group.FD
```

Step 3 • cgroups: what it can use

DO THIS

- 01 Open `internal/cgroup/cgroup.go` and write `cpuMax()` at `TODD(step 3)`.
- 02 Write `Files()` just below it: one file per limit that is set.
- 03 Run `go test ./internal/cgroup/` until it passes.
- 04 Open `internal/container/container_linux.go`, find `TODD(step 3)`, and add the lines on the right.

Keep `-pids` at sixteen or more. It counts the Go init's own threads.

CHECK IT

```
make build && sudo ./mycontainer run -pids 20 ubuntu /bin/sh
cat /sys/fs/cgroup/pids.max; for i in $(seq 40); do sleep 2 & done      # inside
```



20, then Cannot fork.

HINTS

Item 2 is four files: `memory.max` in bytes, `memory.swap.max` as `0` beside it, `cpu.max`, `pids.max`. A zero limit writes nothing.

Item 1: `cpu.max` is a quota and a period in microseconds. The period is 100000, so half a CPU is `50000 100000`.

Item 4:

```
if group != nil {
    attr.UseCgroupFD = true
    attr.CgroupFD = group.FD
```

Root is a set of bits

The kernel does not ask whether you are UID 0. It asks whether the process holds the capability for what it is doing.

```
# root on the VM  
CapBnd: 000001ffffffffffff bits 0 to 40, all of them
```



Root is a set of bits

The kernel does not ask whether you are UID 0. It asks whether the process holds the capability for what it is doing.

Docker keeps fourteen. `CAP_SYS_ADMIN`, bit 21, is not one of them, and `mount(2)` wants it.

```
# root in the container
```

```
CapBnd: 00000000a80425fb    fourteen bits
```

0 chown	7 setuid	27 mknod
1 dac_override	8 setpcap	29 audit_write
3 fowner	10 net_bind_service	31 setfcap
4 fsetid	13 net_raw	
5 kill	18 sys_chroot	
6 setgid		



Root is a set of bits

The kernel does not ask whether you are UID 0. It asks whether the process holds the capability for what it is doing.

Docker keeps fourteen. `CAP_SYS_ADMIN`, bit 21, is not one of them, and `mount(2)` wants it.

So root in the container cannot mount.

Your job: the fourteen. Dropping the rest is written.

```
# root in the container
```

```
CapBnd: 00000000a80425fb      fourteen bits
```

0 chown	7 setuid	27 mknod
1 dac_override	8 setpcap	29 audit_write
3 fowner	10 net_bind_service	31 setfcap
4 fsetid	13 net_raw	
5 kill	18 sys_chroot	
6 setgid	21 sys_admin	x

```
# mount -t tmpfs none /tmp
```

```
mount: /tmp: permission denied.
```



Step 4 • Capabilities: what it can do

DO THIS

- 01 Open `internal/caps/caps.go` and find `TODO(step 4)` in `Default()`.
- 02 Add the capabilities at the top of the file to the `Set`, all but `SysAdmin`. That is fourteen.
- 03 Run `go test ./internal/caps/ ./internal/cli/` until both pass.
- 04 Read `Child` in `internal/container/container_linux.go`. It locks the thread before the drop, and the warning says why.

CHECK IT

```
make build && sudo ./mycontainer run ubuntu /bin/sh  
grep CapBnd /proc/self/status; mount -t tmpfs none /tmp
```



HINTS

Item 2:

```
var s Set  
s.Add(Chown)  
// ...thirteen more  
return s
```



Capabilities are per thread. Drop on one, fork from

type these inside

Step 4 • Capabilities: what it can do

DO THIS

- 01 Open `internal/caps/caps.go` and find `TODO(step 4)` in `Default()`.
- 02 Add the capabilities at the top of the file to the `Set`, all but `SysAdmin`. That is fourteen.
- 03 Run `go test ./internal/caps/ ./internal/cli/` until both pass.
- 04 Read `Child` in `internal/container/container_linux.go`. It locks the thread before the drop, and the warning says why.

CHECK IT

```
make build && sudo ./mycontainer run ubuntu /bin/sh  
grep CapBnd /proc/self/status; mount -t tmpfs none /tmp
```



HINTS

Item 2:

```
var s Set  
s.Add(Chown)  
// ...thirteen more  
return s
```



Give one back with `-caps default,+CAP_SYS_ADMIN: a82425fb`, bit 21, and the mount works.

Capabilities are per thread. Drop on one, fork from

type these inside

Step 4 • Capabilities: what it can do

DO THIS

- 01 Open `internal/caps/caps.go` and find `TODO(step 4)` in `Default()`.
- 02 Add the capabilities at the top of the file to the `Set`, all but `SysAdmin`. That is fourteen.
- 03 Run `go test ./internal/caps/ ./internal/cli/` until both pass.
- 04 Read `Child` in `internal/container/container_linux.go`. It locks the thread before the drop, and the warning says why.

CHECK IT

```
make build && sudo ./mycontainer run ubuntu /bin/sh  
grep CapBnd /proc/self/status; mount -t tmpfs none /tmp
```



`a80425fb`, and root cannot mount.

HINTS

Item 2:

```
var s Set  
s.Add(Chown)  
// ...thirteen more  
return s
```



Give one back with `-caps`
`default,+CAP_SYS_ADMIN: a82425fb`, bit 21, and
the mount works.

Capabilities are per thread. Drop on one, fork from

type these inside

PID 1 waits for everyone

When a parent exits before its children, the kernel hands them to PID 1.

PID	PPID	STAT	CMD
1	0	S	mycontainer child
8	1	S	sh
9	8	S	sh -c 'sleep 2 &'
10	9	S	sleep 2



PID 1 waits for everyone

When a parent exits before its children, the kernel hands them to PID 1.

A child that exits stays in the table, a zombie, until its parent waits for it.

PID	PPID	STAT	CMD
1	0	S	mycontainer child
8	1	S	sh
10	1	S	sleep 2

9 exited. Its child is PID 1's now.



PID 1 waits for everyone

When a parent exits before its children, the kernel hands them to PID 1.

A child that exits stays in the table, a zombie, until its parent waits for it.

Only PID 1 can wait for an orphan. One that waits for its own child and nobody else collects zombies.

PID	PPID	STAT	CMD
1	0	S	mycontainer child
8	1	S	sh
10	1	Z	[sleep] <defunct>

10 exited. Nobody has waited for it.



PID 1 waits for everyone

When a parent exits before its children, the kernel hands them to PID 1.

A child that exits stays in the table, a zombie, until its parent waits for it.

Only PID 1 can wait for an orphan. One that waits for its own child and nobody else collects zombies.

Your job: wait for any child, as often as one exits, and never block.

PID	PPID	STAT	CMD
1	0	S	mycontainer child
8	1	S	sh

```
# PID 1 called wait4(-1, ..., WNOHANG).
```



Step 5 • Init: what PID 1 owes

DO THIS

- 01 Open `internal/initproc/initproc_linux.go` and find `TODO(step 5)` in `reap()`.
- 02 Replace the one `Wait4(main, ...)` with a loop on `Wait4(-1, ..., syscall.WNOHANG, nil)`.
- 03 Decide what each result means. The hints say.
- 04 Run `go test ./internal/initproc/`, then `go test ./...`.

CHECK IT

```
go test ./internal/initproc/ && go test ./...
```



HINTS

Item 2: any child, and never block.

```
for {  
    var ws syscall.WaitStatus  
    pid, err := syscall.Wait4(-1, &ws,  
        syscall.WNOHANG, nil)  
}
```



Step 5 · Init: what PID 1 owes

DO THIS

- 01 Open `internal/initproc/initproc_linux.go` and find `TODO(step 5)` in `reap()`.
- 02 Replace the one `Wait4(main, ...)` with a loop on `Wait4(-1, ..., syscall.WNOHANG, nil)`.
- 03 Decide what each result means. The hints say.
- 04 Run `go test ./internal/initproc/`, then `go test ./...`.

CHECK IT

```
go test ./internal/initproc/ && go test ./...
```



HINTS

Item 2: any child, and never block.

```
for {  
    var ws syscall.WaitStatus  
    pid, err := syscall.Wait4(-1, &ws,  
        syscall.WNOHANG, nil)  
}
```



Item 3. `pid == 0` : none has exited yet, return.

`ECHILD` : none left, return. `EINTR` : again.

Step 5 • Init: what PID 1 owes

DO THIS

- 01 Open `internal/initproc/initproc_linux.go` and find `TODD(step 5)` in `reap()`.
- 02 Replace the one `Wait4(main, ...)` with a loop on `Wait4(-1, ..., syscall.WNOHANG, nil)`.
- 03 Decide what each result means. The hints say.
- 04 Run `go test ./internal/initproc/`, then `go test ./...`.

CHECK IT

```
go test ./internal/initproc/ && go test ./...
```



HINTS

Item 2: any child, and never block.

```
for {  
    var ws syscall.WaitStatus  
    pid, err := syscall.Wait4(-1, &ws,  
        syscall.WNOHANG, nil)  
}
```



Item 3. `pid == 0` : none has exited yet, return.

`ECHILD` : none left, return. `EINTR` : again.

`pid == main` : keep `ExitCode(ws)` and set `done`.

Any other pid was an orphan. Loop either way.

Step 5 • Init: what PID 1 owes

DO THIS

- 01 Open `internal/initproc/initproc_linux.go` and find `TODD(step 5)` in `reap()`.
- 02 Replace the one `Wait4(main, ...)` with a loop on `Wait4(-1, ..., syscall.WNOHANG, nil)`.
- 03 Decide what each result means. The hints say.
- 04 Run `go test ./internal/initproc/`, then `go test ./...`.

CHECK IT

```
go test ./internal/initproc/ && go test ./...
```



Green means no zombie after an orphan exits, and nothing is red any more.

HINTS

Item 2: any child, and never block.

```
for {  
    var ws syscall.WaitStatus  
    pid, err := syscall.Wait4(-1, &ws,  
        syscall.WNOHANG, nil)  
}
```



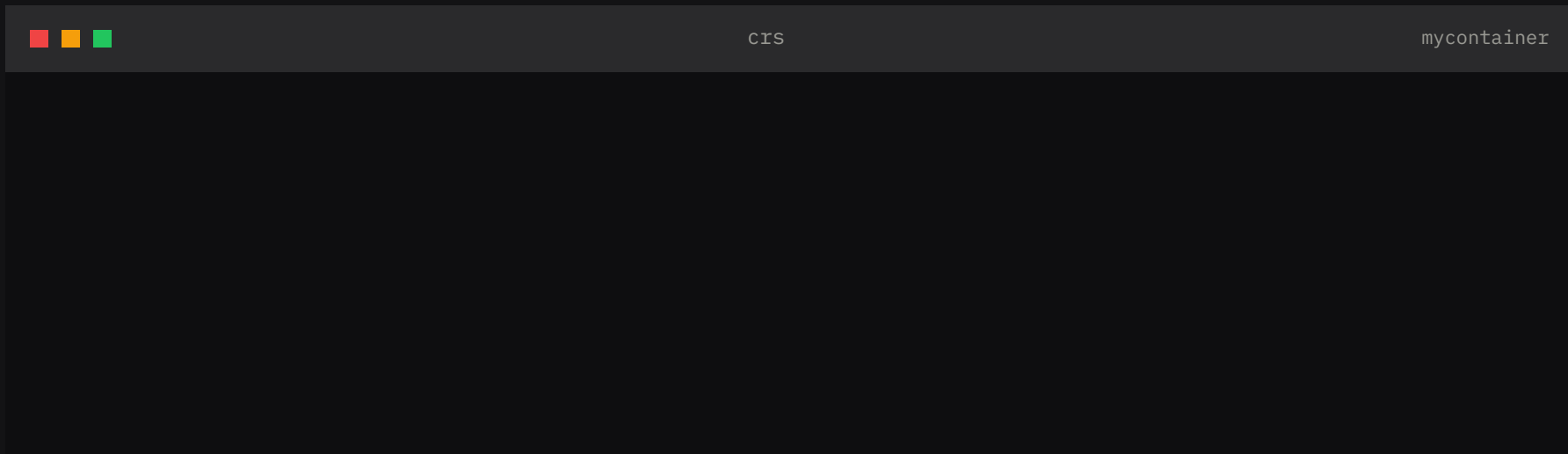
Item 3. `pid == 0` : none has exited yet, return.

`ECHILD` : none left, return. `EINTR` : again.

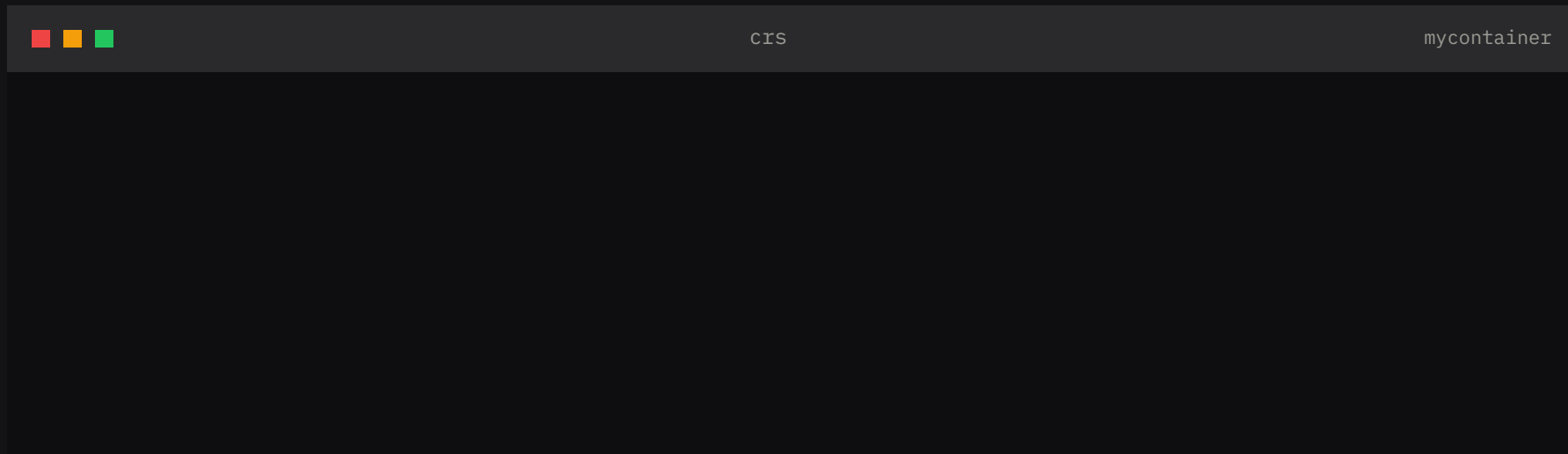
`pid == main` : keep `ExitCode(ws)` and set `done`.

Any other pid was an orphan. Loop either way.

What you just built, counted



What you just built, counted



1 clone3 · 1 sethostname · 12 mount · 1 pivot_root · 1 umount2 · 27 PR_CAPBSET_DROP · 3 execve

Take home

Take home

1. A container is a process wearing a costume: seven namespace bits, a pivoted mount, a cgroup directory, fourteen capability bits, and an init.

Take home

1. A container is a process wearing a costume: seven namespace bits, a pivoted mount, a cgroup directory, fourteen capability bits, and an init.
2. A PID namespace changes the numbers. `/proc` is what `ps` reads.

Take home

1. A container is a process wearing a costume: seven namespace bits, a pivoted mount, a cgroup directory, fourteen capability bits, and an init.
2. A PID namespace changes the numbers. `/proc` is what `ps` reads.
3. Root is a bounding set. Drop one bit and root cannot mount.

Take home

1. A container is a process wearing a costume: seven namespace bits, a pivoted mount, a cgroup directory, fourteen capability bits, and an init.
2. A PID namespace changes the numbers. `/proc` is what `ps` reads.
3. Root is a bounding set. Drop one bit and root cannot mount.
4. PID 1 owes the tree reaping and signals. Bring an init.

Take home

1. A container is a process wearing a costume: seven namespace bits, a pivoted mount, a cgroup directory, fourteen capability bits, and an init.
2. A PID namespace changes the numbers. `/proc` is what `ps` reads.
3. Root is a bounding set. Drop one bit and root cannot mount.
4. PID 1 owes the tree reaping and signals. Bring an init.
5. In Go: start the child through `/proc/self/exe`, lock the thread before you drop capabilities, and count your own threads against `pids.max`.

Take home

1. A container is a process wearing a costume: seven namespace bits, a pivoted mount, a cgroup directory, fourteen capability bits, and an init.
2. A PID namespace changes the numbers. `/proc` is what `ps` reads.
3. Root is a bounding set. Drop one bit and root cannot mount.
4. PID 1 owes the tree reaping and signals. Bring an init.
5. In Go: start the child through `/proc/self/exe`, lock the thread before you drop capabilities, and count your own threads against `pids.max`.

At home: three lines of UID mapping make it rootless. `docs/workshop.md` has them.

// THAT'S A WRAP

Thank You

The slides and the code are at:

github.com/asahasrabuddhe/cds-2026-singapore-crs-code

[cds-2026-singapore-crs-slides](#)

[cds-2026-singapore-crs-code](#)

[@asahasrabuddhe](#)



scan for slides + code