Docker

from the inside out

David Calavera

Agenda

Execution drivers

Storage drivers

Logging drivers

Volume drivers

Container

Isolation

Namespaces

CLONE_NEWUSER

CLONE_NEWPID

CLONE_NEWUTS

CLONE_NEWNS

Forking in Go

Re-exec yourself with a different name! 😎[1]

```
// newns.go
func main() {
    if os.Args[0] == "fork-newns" {
        os.Exit(fork())
    }
    reExec()
}
```

[1] The code in this presentation doesn't check errors properly 😱

Forking in Go

```
func fork() int {
    name, err := exec.LookPath(os.Args[1])
    if err != nil {
        return 1
    }
    // system calls inside the container
    syscall.Exec(name, os.Args[1:], os.Environ())
    return 0
}
```

CLONE_NEWUSER

```
run := []string{"sh", "-c", "echo", "hello camp!"}
cmd := &exec.Cmd{
    Path: os.Args[0],
    Args: append([]string{"fork-newns"}, run...),
}

cmd.SysProcAttr = &syscall.SysProcAttr{
    Cloneflags: syscall.CLONE_NEWUSER,
    UidMappings: []syscall.SysProcIDMap{{0, os.Getuid(), 1}},
    GidMappings: []syscall.SysProcIDMap{{0, os.Getgid(), 1}},
}
cmd.Run()
```

CLONE_NEWPID

```
run := []string{"sh", "-c", "echo", "hello camp!"}
cmd := &exec.Cmd{
    Path: os.Args[0],
    Args: append([]string{"fork-newns"}, run...),
}
cmd.SysProcAttr = &syscall.SysProcAttr{
    Cloneflags: syscall.CLONE_NEWUSER | syscall.CLONE_NEWPID,
    UidMappings: []syscall.SysProcIDMap{{0, os.Getuid(), 1}},
    GidMappings: []syscall.SysProcIDMap{{0, os.Getgid(), 1}},
}
cmd.Run()
```

CLONE_NEWUTS

```
cmd.Cloneflags = syscall.CLONE_NEWUSER | syscall.CLONE_NEWPID |
        syscall.CLONE_NEWUTS

func fork() int {
    // ...
    // system calls inside the container
    syscall.SetHostname("my-little-container")
    syscall.Exec(name, os.Args[1:], os.Environ())
    return 0
}
```

CLONE_NEWNS

pivot_root(8)

pivot_root **moves the root file system of the current process to the directory *put_old* and makes *new_root* the new root file system.**

CLONE_NEWNS

Filesystem isolation

```
cmd.Cloneflags = syscall.CLONE_NEWUSER | syscall.CLONE_NEWPID |
        syscall.CLONE_NEWUTS | syscall.CLONE_NEWNS

func mount(newRoot string) {
    defFlags := syscall.MS_NOEXEC | syscall.MS_NOSUID | syscall.MS_NODEV
    suidFlags := syscall.MS_NOSUID | syscall.MS_STRICTATIME
    syscall.Mount("/proc", newRoot+"/proc", "proc", defFlags, "")
    syscall.Mount("tmpfs", newRoot+"/dev", "tmpfs", suidFlags, "mode=755")
}
```

CLONE_NEWNS

Filesystem isolation

```
func pivotRoot(newRoot, putOld string) {
    flags := syscall.MS_BIND | syscall.MS_REC
    syscall.Mount(newRoot, newRoot, "bind", flags, "")
    syscall.Mount(putOld, putOld, "bind", flags, "")
    syscall.PivotRoot(newRoot, putOld)
    syscall.Chdir("/")
}
```

CLONE_NEWNS

Filesystem isolation

```
func fork() int {
    // ...
    // system calls inside the container
    mount(newRoot)
    pivotRoot(newRoot, putOld)
    syscall.SetHostname("my-little-container")
    syscall.Exec(name, os.Args[1:], os.Environ())
    return 0
}
```

What about...

Docker on Windows?

or...

Docker on FreeBSD?

Execution drivers

```
// Driver is an interface for drivers to implement
// including all basic functions a driver should have
type Driver interface {
    Run(*Command, *Pipes, StartCallback) (ExitStatus, error)
    Exec(*Command, *ProcessConfig, *Pipes, StartCallback) (int, error)
    Kill(*Command, int) error
    Pause(*Command) error
    Unpause(*Command) error
    Name() string
    Info(string) Info
    GetPidsForContainer(string) ([]int, error)
    Terminate(*Command) error
    Clean(string) error
    Stats(string) (*ResourceStats, error)
}
```

Storage drivers

UFS / COW

Overlay

ZFS

Union filesystems

All of the use cases we are interested in basically boil down to the same thing: having an image or filesystem that is used read-only (either because it is not writable, or because writing to the image is not desired), and pretending that this image or filesystem is writable, storing changes somewhere else.

— Andreas Gruenbacher

Copy-on-Write

Copy-on-write (COW) is a data storage technique in which you make a copy of the data block that is going to be modified, rather than modify the data block directly. You then update your pointers to look at the new block location, rather than the old. You also free up the old block, so it can be available to the application.

— Aaron Toponce

Docker image storage

Layered file system to reduce footprint

Copy on Write changes inside containers

Storage drivers

Docker selects the one that's available to you

Unless you explicitly say so:

```
$ docker daemon --storage-driver zfs
```

Storage drivers

```
func init() {
    graphdriver.Register("overlay", Init)
}

// Init returns a native diff driver for overlay filesystem.
// If overlay filesystem is not supported on the host,
// graphdriver.ErrNotSupported is returned as error.
// If a overlay filesystem is not supported over a existing filesystem,
// then error graphdriver.ErrIncompatibleFS is returned.
func Init(home string, options []string) (graphdriver.Driver, error) {
    ...
}
```

Storage drivers

```
// Driver is the interface for layered/snapshot file system drivers.
type Driver interface {
    ProtoDriver
    Diff(id, parent string) (archive.Archive, error)
    Changes(id, parent string) ([]archive.Change, error)
    ApplyDiff(id, parent string, diff archive.Reader) (size int64, err error)
    DiffSize(id, parent string) (size int64, err error)
}
```

Storage drivers

```
// ProtoDriver defines the basic capabilities of a driver.
type ProtoDriver interface {
    String() string
    Create(id, parent string) error
    Remove(id string) error
    Get(id, mountLabel string) (dir string, err error)
    Put(id string) error
    Exists(id string) bool
    Status() [][2]string
    GetMetadata(id string) (map[string]string, error)
    Cleanup() error
}
```

Overlay

Two layers filesystem

Lower level read only

Upper level read and write

Docker on Overlay

Creating the file system

```
// Get creates and mounts the required file system
// for the given id and returns the mount path.
func (d *Driver) Get(id string, mountLabel string) (string, error) {
    dir := d.dir(id)
    lowerID, _ := ioutil.ReadFile(path.Join(dir, "lower-id"))

    lowerDir := path.Join(d.dir(string(lowerID)), "root")
    upperDir := path.Join(dir, "upper")
    workDir := path.Join(dir, "work")
    mergedDir := path.Join(dir, "merged")

    o := fmt.Sprintf("lowerdir=%s,upperdir=%s,workdir=%s", lowerDir, upperDir, workDir)
    syscall.Mount("overlay", mergedDir, "overlay", 0, label.FormatMountLabel(o, mountLabel))

    return mergedDir
}
```

ZFS

ZFS is a combined file system and logical volume manager

ZFS uses a copy-on-write transactional object model.

When ZFS writes new data, the blocks containing the old data can be retained, allowing a snapshot version of the file system to be maintained.

Docker on ZFS

/var/lib/docker is a ZFS dataset

Docker clones the filesystem for every container

Docker on ZFS

Creating the file system

```
func (d *Driver) Get(id, mountLabel string) (string, error) {
    mountpoint := d.MountPath(id)
    filesystem := d.ZfsPath(id)
    options := label.FormatMountLabel("", mountLabel)

    err := mount.Mount(filesystem, mountpoint, "zfs", options)
    if err != nil {
        return "", ErrZFSCreate{filesystem, mountpoint}
    }

    return mountpoint, nil
}
```

Logging drivers

Docker will die in production from all the logs it accumulates

— A friend of mine, summer 2013

Logging drivers

JSON log

Syslog

Journald

Gelf

Fluentd

Logging drivers

```
// Message is datastructure that represents record from some container.
type Message struct {
    ContainerID string
    Line        []byte
    Source      string
    Timestamp   time.Time
}

// Logger is the interface for docker logging drivers.
type Logger interface {
    Log(*Message) error
    Name() string
    Close() error
}
```

Logging drivers

```
func init() {
    logger.RegisterLogDriver("journald", New)
}

// New creates a journald logger using the configuration passed in on
// the context.
func New(ctx logger.Context) (logger.Logger, error) {
    jmap := map[string]string{
        "CONTAINER_ID":      ctx.ContainerID,
        "CONTAINER_NAME":    ctx.ContainerName}
    return &journald{Jmap: jmap}, nil
}
```

Logging drivers

```
func (s *journald) Log(msg *logger.Message) error {
    if msg.Source == "stderr" {
        return journal.Send(string(msg.Line), journal.PriErr, s.Jmap)
    }
    return journal.Send(string(msg.Line), journal.PriInfo, s.Jmap)
}
```

Logging drivers

Copying logs

```
func (container *Container) startLogging() error {
    cfg := container.getLogConfig()
    l, _ := container.getLogger()

    sources := map[string]io.Reader{
        "out": container.StdoutPipe(),
        "err": container.StderrPipe()}
    copier := logger.NewCopier(container.ID, sources, l)
    container.logCopier = copier
    copier.Run()
    container.logDriver = l

    return nil
}
```

Logging drivers

Copying logs

```
// Writes are concurrent.
// You need implement some sync in your logger
type Copier struct {
    ...
    copyJobs sync.WaitGroup
}

// Run starts logs copying
func (c *Copier) Run() {
    for name, reader := range c.srcs {
        c.copyJobs.Add(1)
        go c.copySrc(name, reader)
    }
}
```

Logging drivers

Copying logs

```
func (c *Copier) copySrc(name string, src io.Reader) {
    defer c.copyJobs.Done()
    reader := bufio.NewReader(src)

    for { //ever
        line, err := reader.ReadBytes('\n')
        line = bytes.TrimSuffix(line, []byte{'\n'})

        // ReadBytes can return full or partial output even when it failed.
        // e.g. it can return a full entry and EOF.
        if len(line) > 0 {
            c.dst.Log(&Message{ContainerID: c.cid, Line: line,
                       Source: name, Timestamp: time.Now().UTC()})
        }
        if err != nil { return }
    }
}
```

Volume drivers

```
$ docker run --volume-driver foo
```

Volume drivers

Architecture

RPC calls

- VolumeDriver.Create
- VolumeDriver.Remove
- VolumeDriver.Mount
- VolumeDriver.Unmount
- VolumeDriver.Path

Volume drivers

GlusterFS[2]

Scalable network filesystem.

[2] https://www.gluster.org

Volume drivers

GlusterFS plugin[3]

```
func (d glusterfsDriver) Create(r dkvolume.Request) dkvolume.Response {
    ...
    exist, _ := d.restClient.VolumeExist(r.Name)
    if !exist {
        if err := d.restClient.CreateVolume(r.Name, d.servers); err != nil {
            return dkvolume.Response{Err: err.Error()}
        }
    }
    return dkvolume.Response{}
}
```

[3] https://github.com/calavera/docker-volume-glusterfs

Volume drivers

GlusterFS plugin

```
func (d glusterfsDriver) Mount(r dkvolume.Request) dkvolume.Response {
    mountPoint := d.mountpoint(r.Name)
    server := d.servers[rand.Intn(len(d.servers))]
    cmd := fmt.Sprintf("glusterfs --volfile-id=%s --volfile-server=%s %s",
                r.Name, server, mountPoint)
    exec.Command("sh", "-c", cmd).CombinedOutput()
    return dkvolume.Response{Mountpoint: mountPoint}
}
```

Volume drivers

Vault[4]

A tool for managing secrets.

[4] https://vaultproject.io

Volume drivers

Vault plugin[5]

```
func (d *driver) Create(r dkvolume.Request) dkvolume.Response {
    return dkvolume.Response{}
}
```

[5] https://github.com/calavera/docker-volume-vault

Volume drivers

Vault plugin

```
func (d *driver) Mount(r dkvolume.Request) dkvolume.Response {
    mountPoint := d.mountpoint(r.Name)
    conf := api.DefaultConfig()
    client, _ := api.NewClient(conf)
    fs, root := NewFs(client, newOwnership("root", "root"))
```

Volume drivers

Vault plugin

```
    mountOptions := &fuse.MountOptions{
        AllowOther: true,
        Name:       fs.String(),
        Options:    []string{"default_permissions"},
    }
    conn := nodefs.NewFileSystemConnector(root, &nodefs.Options{})
    server, _ := fuse.NewServer(conn.RawFS(), mountPoint, mountOptions)
    go server.Serve()
    return dkvolume.Response{Mountpoint: mountPoint}
}
```

Docker is extensible

in more ways than you think

And we haven't even talked about networking

Thank you

and thanks to Alex Morozov and Jérôme Petazzoni for the insights on namespaces and storage on Linux.