Containers without Daemons

Podman Internals

Matt Heon
Software Engineer, Red Hat
A Brief History of Podman

- Started as a debug tool for CRI-O
- Expanded to become a full-featured runtime
- Branched off CRI-O to speed development
- First release in February ‘18
- Released 1.0 in January ‘19
Podman is a container engine
(Not a runtime)

- We don’t make containers - we tell a runtime to make them
- Open Containers Initiative standardized a way of talking to runtimes
- So, what do we do?
  - Storage
  - Networking
  - Lifecycle
Requirements

- Make a library that both CRI-O and Podman can use to launch containers
- Needs to run both with and without a daemon
- Re-use as much of CRI-O as possible
- Performance nearly identical to CRI-O
- Support for Pods
- Latecomers:
  - Remote API
  - Rootless Containers
Design Goals: Multiprocess

- Disk-backed database as single source of truth
- Strong emphasis on immutability
  - We can’t completely eliminate mutable state, but we can get pretty close
- Strong emphasis on safety over performance
  - Keep locking simple
  - Sync with DB early and often
  - Easier to fix performance later than to debug state corruption
Design Goals: Podman

- We split off from CRI-O, but we want to reintegrate easily
- Split core logic into libpod and CLI parsing into Podman
- Container management logic lives in libpod, provides an API for both CRI-O and libpod
- Podman translates CLI input into API calls - as lightweight as we can get away with
- Keep the two as separate as possible
Containers

- Managed by Conmon
- Strong dependencies allowed
- Some operations not possible because of immutability
  - Renaming
  - Adjusting resource limits
Pods

- A group of libpod containers that are managed together
- More loosely defined than Kubernetes pods
- In practice, the Podman CLI mostly makes pods very similar to Kubernetes
Images

- We already have good libraries for managing images
- API is not the best, though
- Main difficulty: parsing user input in a world with multiple registries

```
babybox
example.com/busybox
docker://example.com/busybox
example.com/busybox@sha256:0123456...
docker://example.com/busybox:latest
example.com/busybox:latest@sha256:0123456...
docker://example.com/busybox:latest
```
State

- We started off with SQLite, but dropped it quickly
- Moved to BoltDB - lightweight key-value store
- Serialize state via JSON and store in DB
- Very tolerant of rapid pace of development
At first, we used file locks. That was a mistake.
Locking

- Every container and pod has a lock
- We started with file locks - one file per lock
- Moved to mutexes in a shared memory segment
- Fixed number of locks, but we can reallocate the segment to increase
Podman

- Minimal layer on top of libpod
- Some glue logic to correct differences between CLI and libpod
- Most of the code is container creation
  - Parsing the CLI
  - Setting sane defaults where the CLI does not
  - Can’t really share this with CRI-O
- Starting to introduce indirection to allow CLI to also be used with Varlink
Varlink

- Varlink is an IPC protocol somewhat similar to DBus by the authors of systemd
- We were one of the first adopters
- We wanted to do an API without requiring a daemon
- And we (mostly) did it
  - Socket activation
  - Handle connection
  - Time out and exit after finishing
Rootless

- No added privileges - Podman is not setuid
- All we need is a user namespace
- Gets us surprisingly far - most things work without modification
- Mostly just manifests as a lot of conditions in the code - some operations can only be done by root
Walking through `podman run`

**Step 1: Pull an Image**
`libpod/image/image.go`

```go
func (ir *Runtime) New(ctx context.Context,
    name, signaturePolicyPath, authfile string,
    writer io.Writer,
    dockeroptions *DockerRegistryOptions,
    signingoptions SigningOptions,
    forcePull bool) (*Image, error)
```
Step 2: Parse the Command Line
cmd/podman/create.go

```go
func parseCreateOpts(ctx context.Context,
        c *cli.Context,
        runtime *libpod.Runtime,
        imageName string,
        data *inspect.ImageData) (*cc.CreateConfig, error)
```
func CreateConfigToOCISpec(config *CreateConfig) (*spec.Spec, error)

func (c *CreateConfig) GetContainerCreateOptions(
    runtime *libpod.Runtime,
    pod *libpod.Pod) ([][]libpod.CtrCreateOption, error)
Step 3: Make the Container
libpod/runtime_ctr.go

```go
func (r *Runtime) NewContainer(ctx context.Context,
    rSpec *spec.Spec,
    options ...CtrCreateOption) (*Container, error)

func (s *BoltState) AddContainer(ctr *Container) error
```
Step 4: Start the container
libpod/container_api.go

```go
func (c *Container) Start(ctx context.Context) error
func (c *Container) syncContainer() error
func (c *Container) prepare() error
func (c *Container) init(ctx context.Context) error
func (c *Container) start() error
```