Odin

Live migrating a container:

pros, cons and gotchas

Павел Емельянов

Linux-Piter, Санкт-Петербург, 2015

Agenda

- Why you might want to live migrate a container
- Why (and how) to avoid live migration
- Why is container live migration so complex

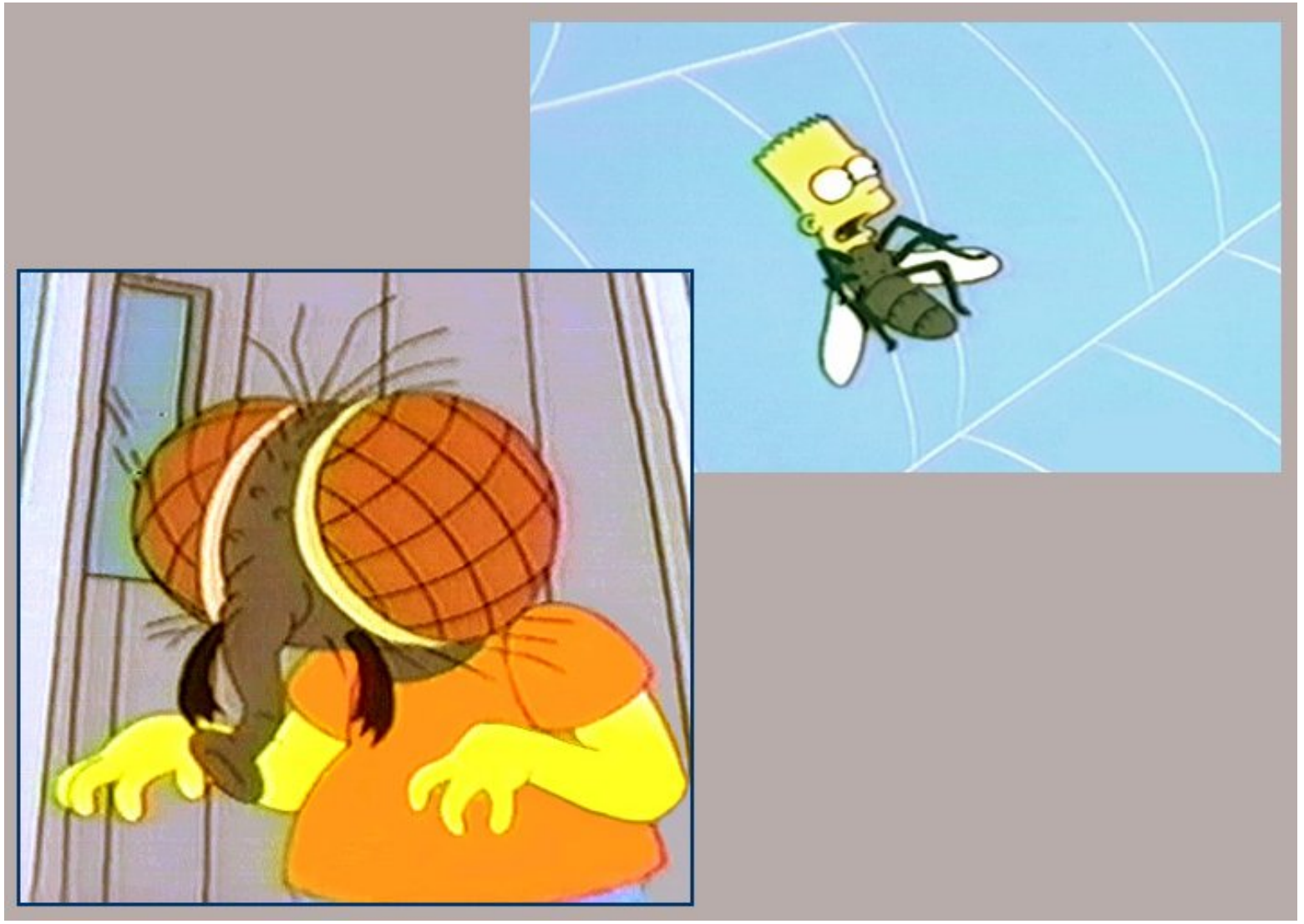
Migration in a nutshel

- Get object state
- Send state information
- Restore object copy from state

Why you might want to live migrate a container

- Load balancing
- Updating kernel
  - Can avoid live migration, just C/R
- Updaring or replacing hardware
- Spectacular

Why to avoid live migration



How to avoid live migration

- Balance reason for load, e.g. network traffic
- Microservices architecture
- Crash-driven updates
- Planned downtime

Making live migration live

- State saving, *transfering* and restoring happens with tasks frozen
  - “Scatter” container is too complex
- Save & Restore state quicker and quicker
- Avoid transfering big amount of data

Making live migration live

- Save/restore optimizations is a long-running task
  - “Dirty” bits
  - Less system calls
  - O(1) algorythms
- Big *memory* transfer can be easily moved out of the frozen period
  - Memory pre-copy
  - Memory post-copy

Pre-copy

- Copy memory while tasks are running
- Track memory changes
- goto again

Pre-copy

- Pros:
  - Safe: once migrated, source node can disappear
- Cons:
  - Unpredictable: iterations may take long
  - Non-guaranteed: “dirty” memory next round may remain big

Post-copy

- Migrate all but memory
- Turn on “network swap” on destination

Post-copy

Pros:

- Predictable: time to migrate can be well estimated

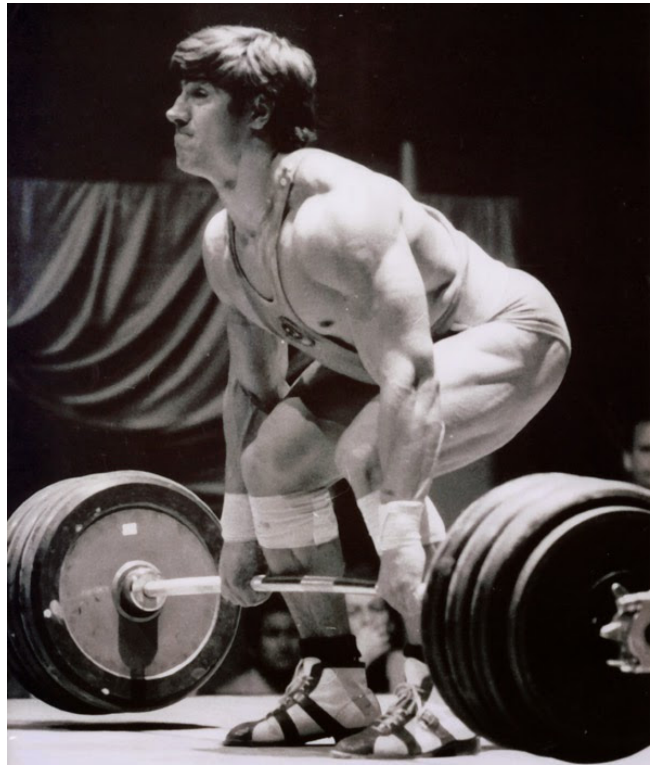
- Cons:
  - Unsafe: src node death means death of container on destination
  - Application slows down after migration

Live migration at length

- Memory pre-copy (iteratively, optional)
- Freeze + Save state
- Copy state
- Restore from state + Unfreeze and resume
- Memory post-copy (optional)

Gotchas

VS



Things to deal with

- VM
  - Environment: virtual hardware, paravirt
  - CPU
  - Memory
- Container
  - Environment: cgroups, namespaces
  - Processes and other animals
  - Memory

Memory pre-copy

- VM
  - All memory at hands
  - Plain address space
- Container
  - Memory
    - is scatered over the processes
    - can be (or can be not) shared
    - can be (or can be not) mapped to disk files

Save state

- VM
  - Hardware state
    - Tree of ~100 objects
    - Fixed amount of data per each
- Container
  - State of *all* objects
    - Graph of up to ~1000 objects
    - All have different amount of data, different reading API

Restore from state

- VM
  - Copy memory in place, write state into devices
- Container
  - Creation of many small objects
  - Not all have sane API for creation
    - Creation sequence can be non-trivial

Memory post-copy

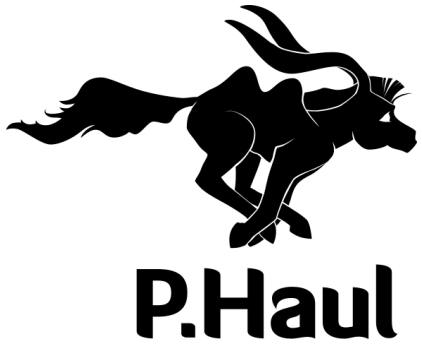
- UserfaultFD from Andrea Archangeli
- VM
  - Merged into 4.2
- Container
  - Non-cooperative work of uffd monitor and client, need further patching

And we also need this, this and this!

- Check for CPUs compatibility
- Check and load necessary kernel modules (iptables, filesystems)
- Non-shared filesystem should be copied
- Roll-back on source node if something fails in between
  - Keep tasks frozen after dump, kill after restore

Implementation

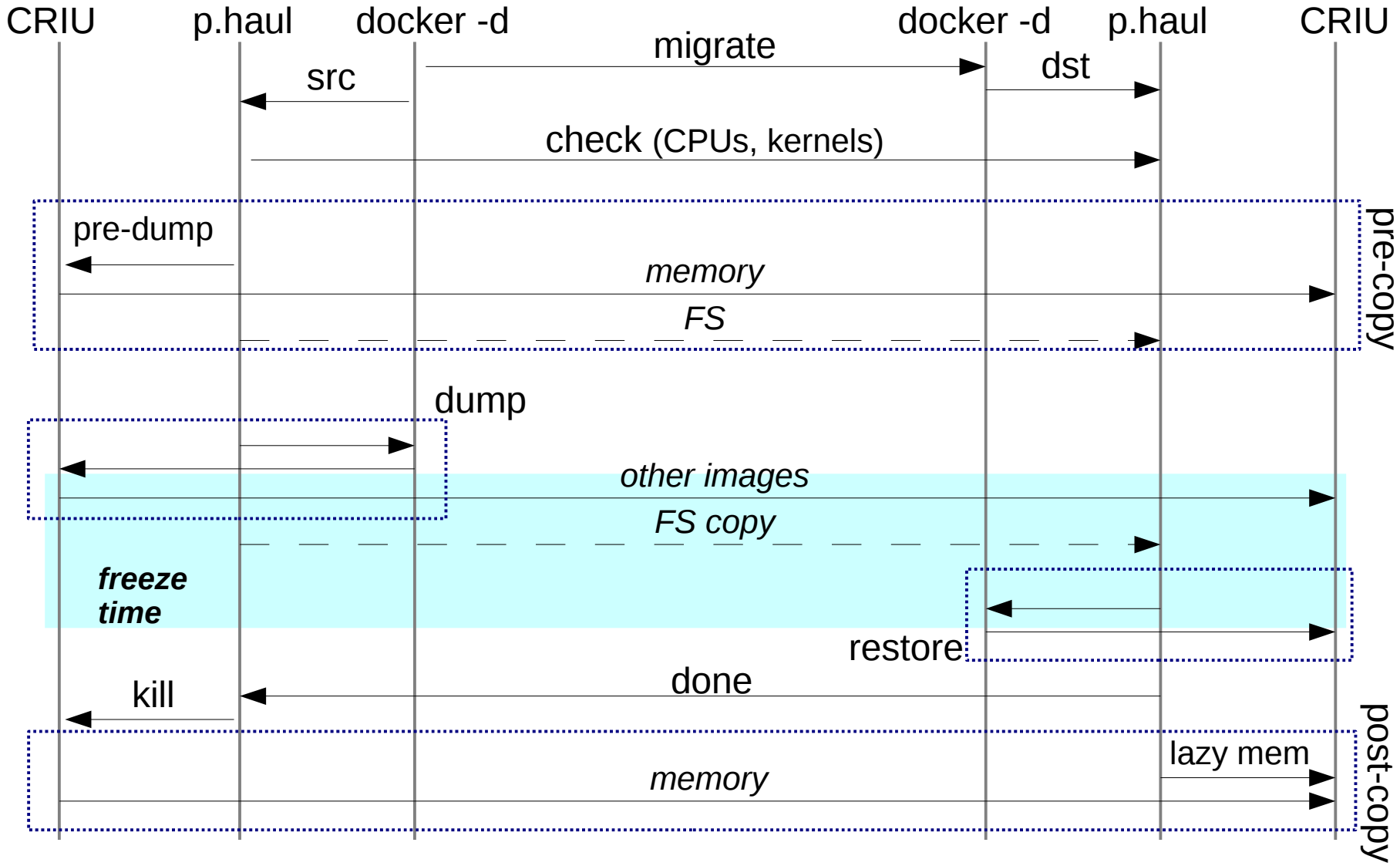
- CRIU: performance critical and low-level actions
  - Save & restore state
  - Memory pre/post copy
- P.Haul: puts everything together
  - Checks
  - Orchestrate all C/R steps
  - Deal with filesystem



P.Haul goals

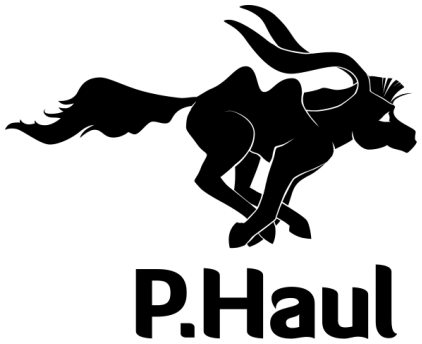
- Provide containers live miration engine using CRIU for OpenVZ/Docker/LXC/whatever
- Perform necessary pre-checks (e.g. CPU compatibility)
- Organize memory pre-copy and/or post-copy
- Take care of file-system migration (if needed)

Under the hood (top-level overview)



More info

- http://criu.org
- http://criu.org/P.Haul
- criu@openvz.org
- +CriuOrg / @__criu__
- https://github.com/xemul/(criu|p.haul)



Odin

Thank you!

xemul@openvz.org