Singularity - Containers for Scientific Computing
ZID Workshop

Michael Fink   Universität Innsbruck

Innsbruck Nov 2018

Overview

Preliminaries

- Why containers
- Understanding Containers vs. Virtual Machines
- Comparison of Container Systesms (LXC, Docker, Singularity) - why Singularity?
- Containers and Host Resources

Using Singularity

- Singularity Workflow

1. **Manual Walkthrough Exercise**

Understanding Singularity

- Web resources
- Container Image Formats, Sources, Conversions
- Running Containers

Advanced Singularity

- Automating Creation of Containers
- Container Contents, Cache Files

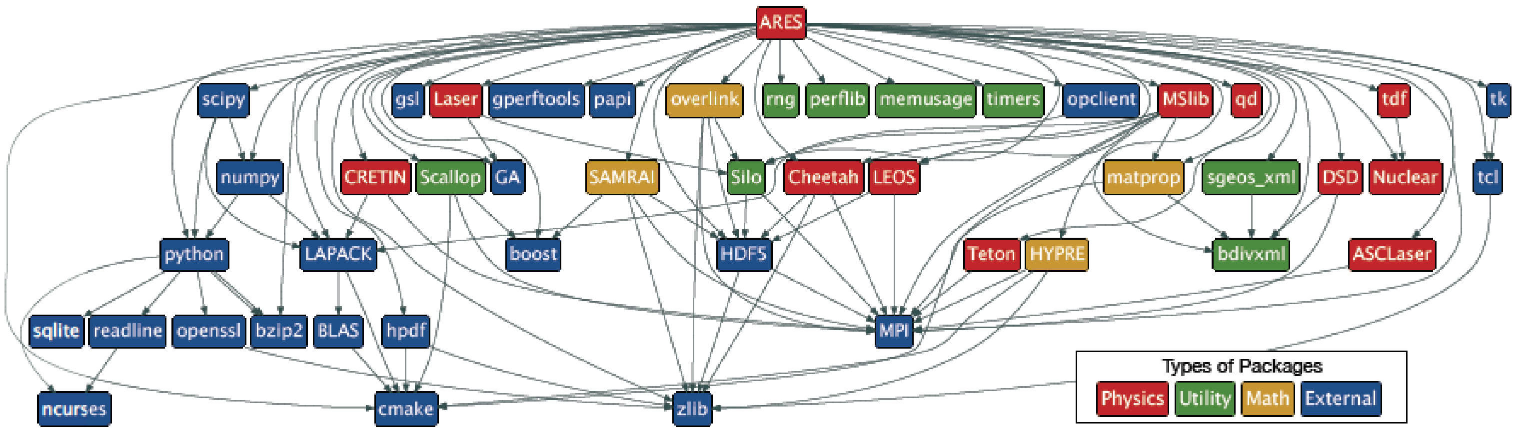
2. **Exercise - Build Container using Build File**

- Using MPI with Singularity

3. **Exercise - MPI Job with Container**

- Singularity Instances

Why Containers?



What is the problem?

- dependency hell
  complex (multiple + indirect + contradictory) software dependencies
- limited HPC team workforce
  always slow, always late
- conservative OS maintenance policy
  risk: upgrade breaks system
  HPC teams prefer stable over innovative OS
  e.g. Redhat/CentOS: backed by HW vendors but very slow adopting new developments
- user portability: differences between installations
  new computer → reinstall and test all software
- reproducibility of results
  recreate old computations for verification

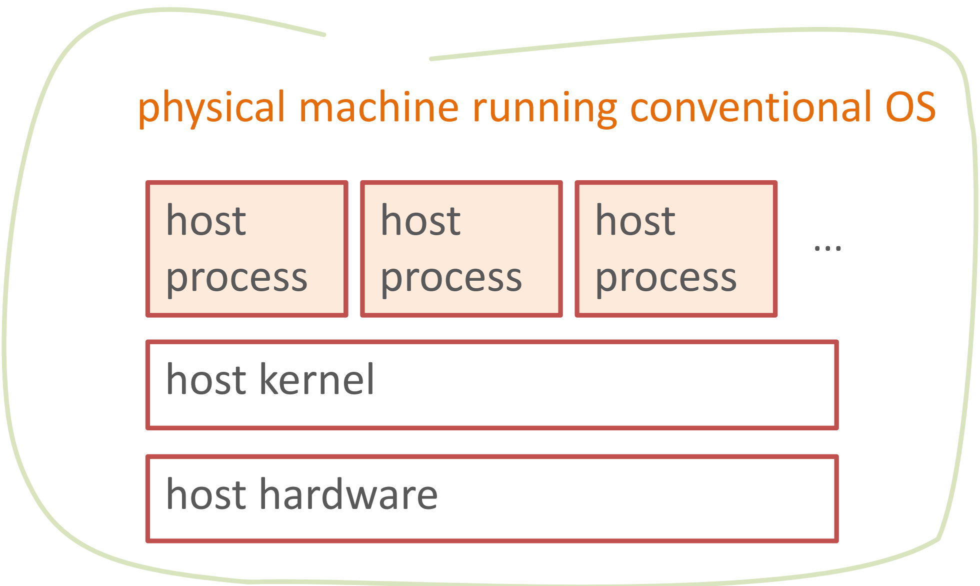
Solution: container: user-defined software environment in isolated, immutable, portable image

- contains user-defined copy of system and user software
- eliminate (most) system dependencies (but: host kernel and MPI must be compatible with container)
- encapsulate software
- long-term archive for reproducibility

Understanding Containers (1)

Conventional OS

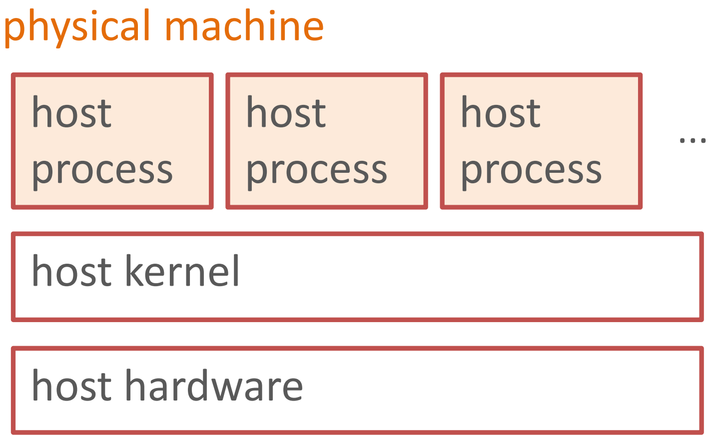
- Kernel runs on physical hardware
- All processes see host's resorces (file systems + files, network, memory etc.)

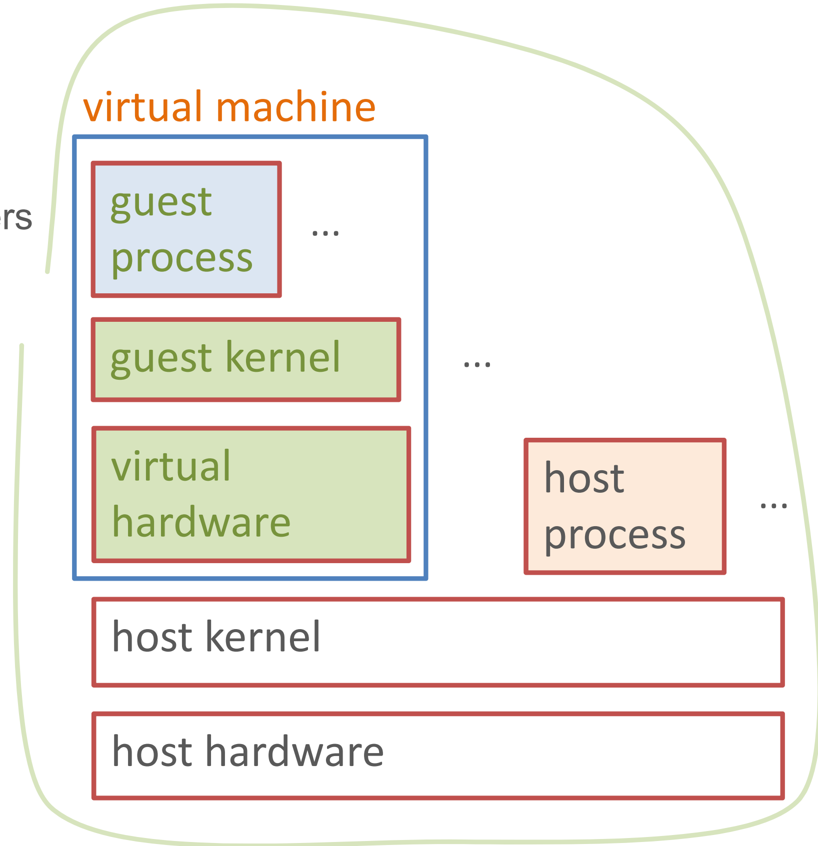


Understanding Containers (2)

Classical virtualization

- Host Kernel runs on physical hardware
- Hypervisor and virtual machines (guests) run as processes on host
- Each virtual machine (guest) has:
  - virtual hardware (processors, memory, network, ...)
  - its own kernel (same or different OS)
  - isolated set of processes, file systems + files etc.
- Virtualization overhead
  - Boot and shutdown, memory footprint, ...
  - Each system call (I/O, network, ...) has to go through all layers
  - 2 levels of multitasking, virtual memory management ...
  - Code instrumentation
  - ....

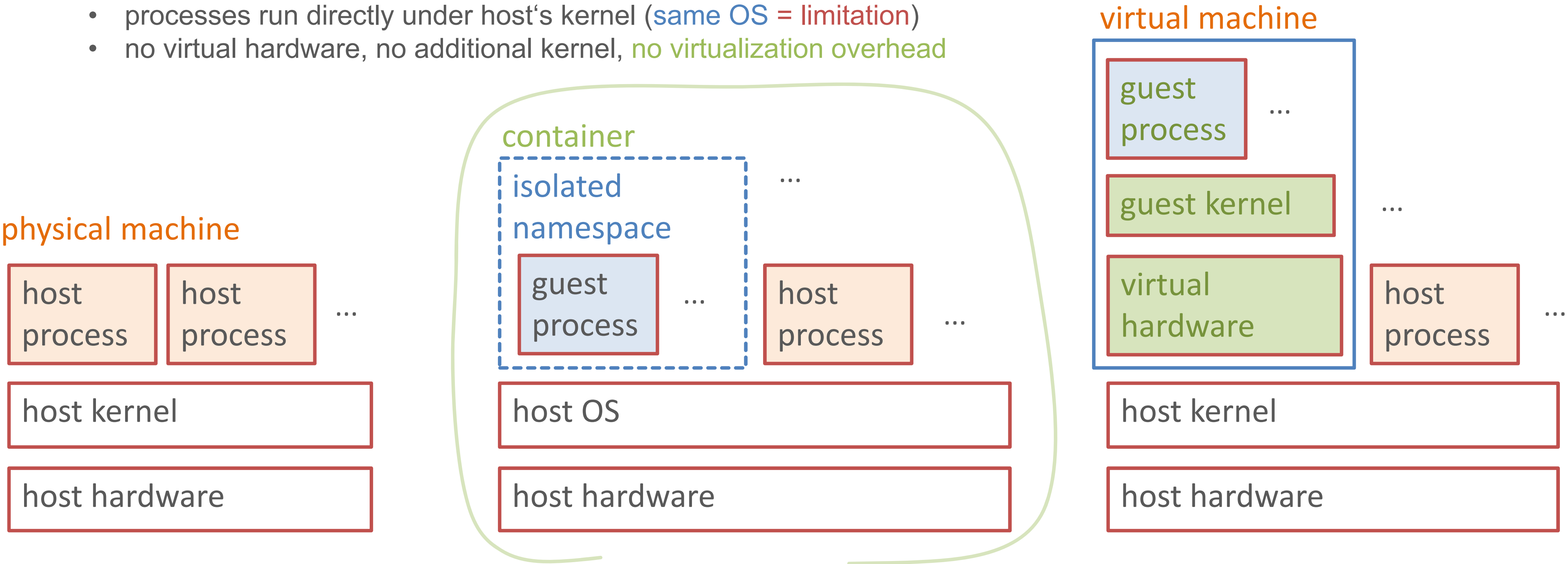




Understanding Containers (3)

Container (aka *OS Level Virtualization*)

- set of processes running on a host with manipulated namespaces = what resources a process can see
  - have private copy of
    - OS utilities and libraries, file systems and files, software, and data
    - other resources (PIDs, network, ...) - not relevant here
- similar to virtual machine, but:
  - processes run directly under host's kernel (same OS = limitation)
  - no virtual hardware, no additional kernel, no virtualization overhead



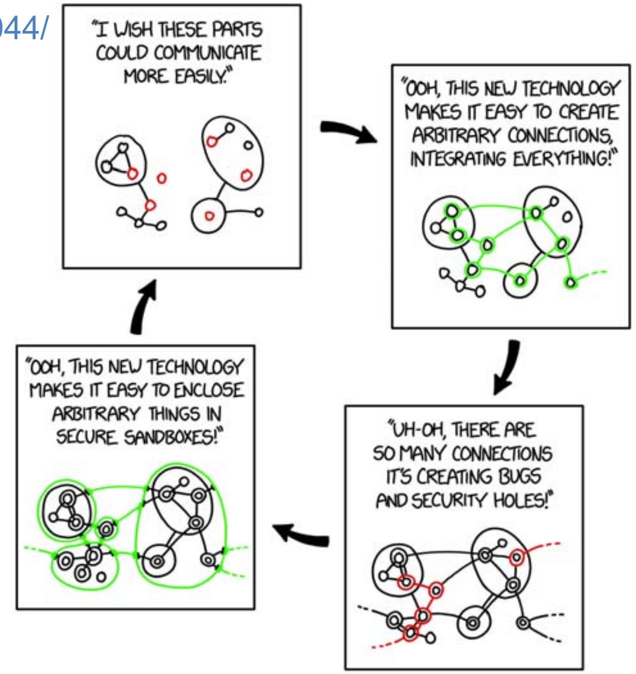
Overview of Container Solutions

- LXC (Linux Containers) linuxcontainers.org
  uses Linux namespaces and resource limitations (cgroups) to provide private, restricted environment for processes
  operation similar to virtual machines (boot, services)
  usage: OS containers (lightweight replacement for servers)
    - alternative to virtual machines
    - several applications per container
- Docker
  similar to LXC, similar purpose (often used for web and database services)
  client - server model:
    - containers run under dockerd
    - user controls operations with docker command
  usage: Application containers
    - typically only one program per container (microservices)
    - containers communicate over virtual network
  advantage:
    - very popular, huge collection of prebuilt containers on dockerhub

- Singularity
  uses Linux namespaces (no cgroups - resource limits should be responsibility of batch system)
  to provide private software environment for processes (user defined software environment)
  operation like running programs from a shell, access to all host resources except root file system
  developed for HPC cluster environments

Docker: why not for HPC?

(*) https://www.xkcd.com/2044/

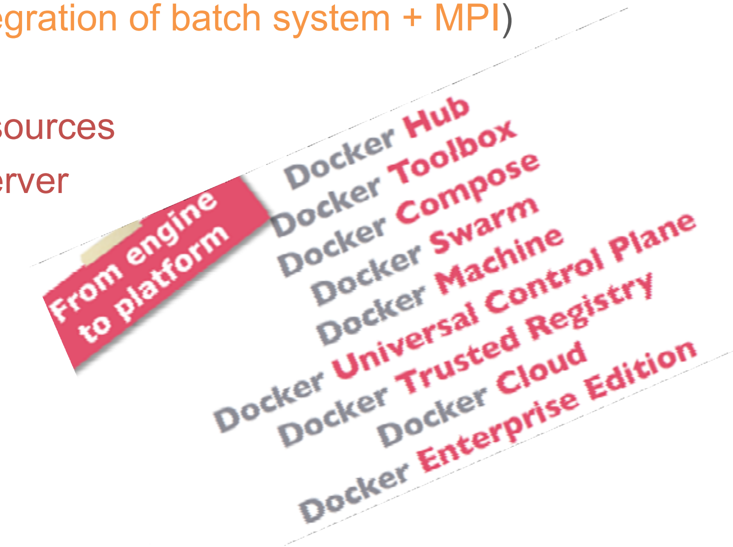


Docker

- de facto standard container solution for virtual hosting
- huge collection of prebuilt containers — repository: Docker Hub

- client-server model
  - containers run under Docker daemon
  - mimick virtual server (startup in background, separate network, ...)
  - docker user commands communicate with Docker daemon
  - breaks process hierarchy (no integration of batch system + MPI)
- need root privileges to run — unsuitable for multiuser
- containers completely isolated from host — no access to user data + host resources
- docker image hidden in obscure place — cannot copy image to arbitrary server

- complex orchestration (*) of multiple containers
- easy on PC, but very complex operation and deployment in cluster



Conclusion: Docker unsuitable for HPC

BUT: Leverage Docker Ecosystem

Why Singularity?

Singularity — easy to understand, use, and operate

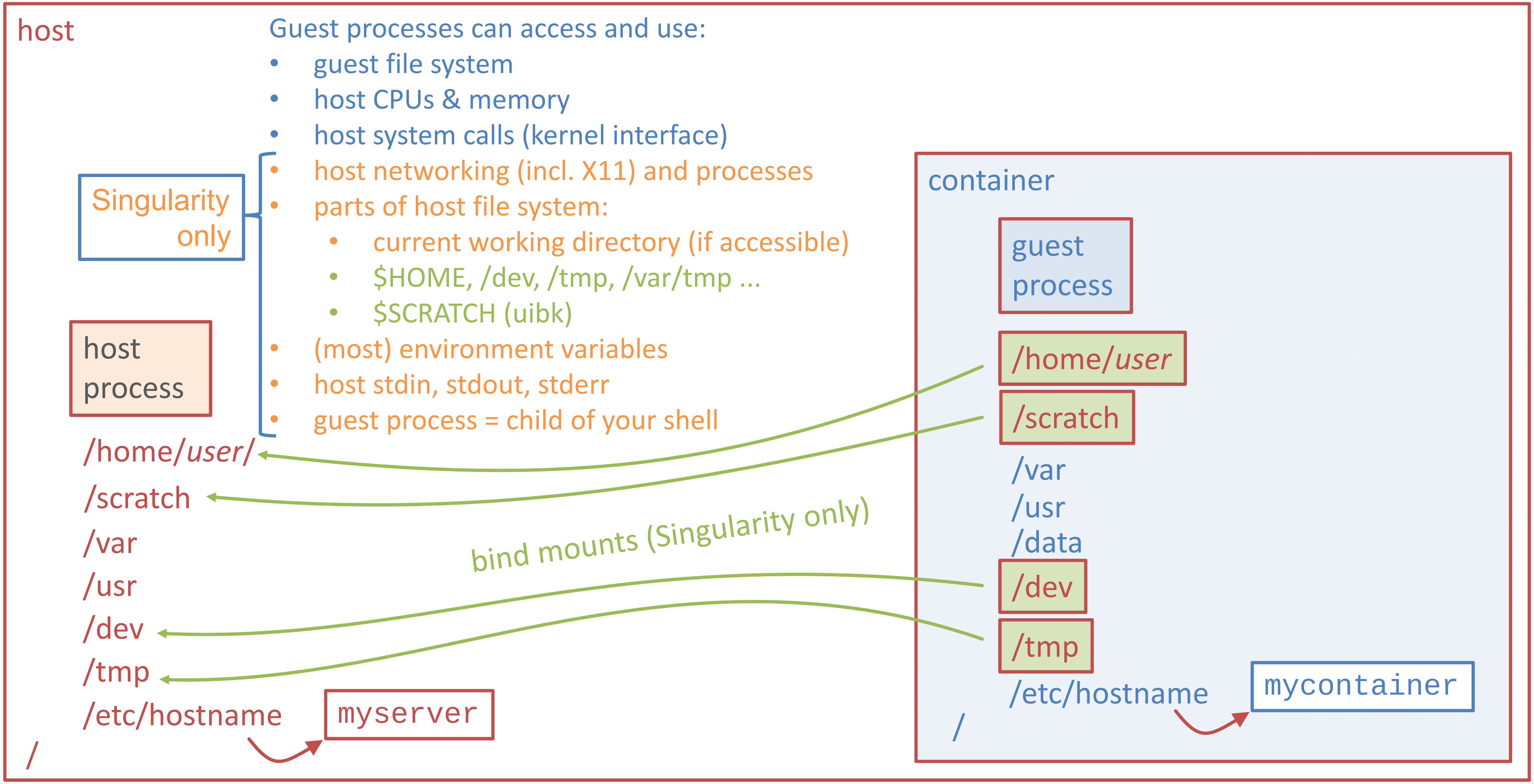
- designed to run in HPC envirunments
- use Docker containers or build your own — singularity can download containers from Docker Hub; no need to install Docker
- container processes run as children of current shell — trivial integration of shell tools (e.g. I/O redirection, pipelines, command line arguments), batch system and MPI
- secure: containers run with normal user privileges — suitable for multiuser
- by default, only replaces root file system — can provide different OS+SW environment, but: full access to all host resources (processors, network, infiniband, $HOME, $SCRATCH etc.)
- singularity image = single immutable file (squashfs) — easily copy / archive image anywhere

- emerges as new standard for HPC containers

note: Charliecloud, Shifter

- older competitors to Singularity
  - more complicated & less flexible
  - need Docker installation

Singularity Containers and Visibility of Host Resources



Singularity Containers and Visibility of Host Resources

On Test VM

```
Singularity test.simg:~/sing-test> df
Filesystem     1K-blocks    Used Available Use% Mounted on
OverlayFS           1024       0      1024   0% /
/dev/sda1       10253588 5107324   4605696  53% /tmp
udev             1989624       0   1989624   0% /dev
tmpfs            2019872   22984   1996888   2% /dev/shm
tmpfs              16384       8     16376   1% /etc/group
tmpfs             403976    1504    402472   1% /etc/resolv.conf
```

Note:

multiple mounts from same file system
(e.g. $HOME, /var/tmp on test VM) are not listed.

use df -a for complete output

On LCC2

```
Singularity test.simg:~> df
Filesystem                                           1K-blocks       Used  Available Use% Mounted on
OverlayFS                                                 1024          0       1024   0% /
hpdoc.uibk.ac.at:/hpc_pool/lcc2/scratch            10712179648 3839379136 6872800512  36% /scratch
/dev/mapper/vg00-lv_root                              25587500    5002364   20585136  20% /etc/hosts
devtmpfs                                               8121232          0    8121232   0% /dev
tmpfs                                                  8133636          0    8133636   0% /dev/shm
na1-hpc.uibk.ac.at:/hpc_home/qt-lcc2-home/home/cb01/cb011060   276901056   86282112  190618944  32% /home/cb01/cb011060
/dev/mapper/vg00-lv_tmp                               25587500      33032   25554468   1% /tmp
/dev/mapper/vg00-lv_var                               25587500    4978280   20609220  20% /var/tmp
tmpfs                                                    16384          8      16376   1% /etc/group
```

Overview

Preliminaries

- Why containers
- Understanding Containers vs. Virtual Machines
- Comparison of Container Sysetsms (LXC, Docker, Singularity) - why Singularity?
- Containers and Host Resources

Using Singularity

- Singularity Workflow

1. **Manual Walkthrough Exercise**

Understanding Singularity

- Web resources
- Container Image Formats, Sources, Conversions
- Running Containers

Advanced Singularity

- Automating Creation of Containers
- Container Contents, Cache Files

2. **Exercise - Build Container using Build File**

- Using MPI with Singularity

3. **Exercise - MPI Job with Container**

- Singularity Instances

Singularity Workflow

You need

- PC
  running Linux (or virtual Linux - e.g. on VirtualBox)
  with Singularity installed + root privilege

  build + configure your container in sandbox directory

  read/write directory hierarchy contains all container's files while building and modifying

  install software (OS utilities, third party software, data....)

  test container

  when finished testing locally - prepare for transfer to HPC system

  convert sandbox to squashfs image

  compressed readonly file system for linux

  automate build using build recipe file

- HPC system with Singularity installed + sufficient disk space (scratch)

  copy image to HPC server

  test + run your container

- If MPI is used

  OpenMPI versions of host (mpirun) and container (mpi libraries) must match

Setting up a VirtualBox Linux Instance

Singularity needs Linux to build containers: need Linux VM on Windows + OS X - recommended for Linux

- Download and install VirtualBox + Extension Pack from virtualbox.org
- VirtualBox: set up a Host Only Ethernet Adapter (Global Tools - Host Network Manager)

  allows you to ssh into your VM

For this workshop

- Download and import the uibk-singularity-workshop.ova demo VM image

For productive work

- Download Linux ISO. e.g. Ubuntu: releases.ubuntu.com
- Create new VM + install Linux ("minimal" is OK)

  ≥ 4 GB memory, ≥ 10 GB + estimated size of data virtual hard disk (dynamic)

  General/Advanced: bidirectional clipboard (need VB Guest Additions)

  Storage: IDE Optical Drive: select Linux ISO (this workshop: Ubuntu AMD64 18.04.1)

  Network: Adapter 1: Enable/NAT (default); recommended if local ssh access: Adapter 2: Enable/Host-only

  Shared Folders: optional (need VB Guest Additions)
- Start VM and install software

  ```
  sudo apt-get update ; sudo apt-get -y upgrade
  sudo apt-get -y install python gcc make libarchive-dev squashfs-tools
  ```

  install VirtualBox Guest Additions then restart machine

Install Singularity

On your Linux (virtual or physical) machine

- Follow steps on http://singularity.lbl.gov/install-linux
  or https://www.uibk.ac.at/zid/systeme/hpc-systeme/common/software/singularity24.html

- Example: version 2.5.2 (current as of June 2018; 2.6 has Nvidia + namespace enhancements)

```
VERSION=2.5.2
wget https://github.com/singularityware/singularity/releases/download/$VERSION/singularity-$VERSION.tar.gz
tar xvf singularity-$VERSION.tar.gz
cd singularity-$VERSION
./configure --prefix=/usr/local
make
sudo make install
```

- Keep installation directory - before installing a new version, remove existing old version:

```
OLDVERSION=x.y.z
cd singularity-$OLDVERSION
sudo make uninstall
cd ..
rm -rf singularity-$OLDVERSION
```

- Version 3.0.1 Nov 2018
  - Complete rewrite in Golang
  - CAUTION: new default container format, not compatible w/ V2.*X*

Singularity Workflow

How to use

simplest alternative - automation recommended

- develop (build and test) container on your PC (e.g. use latest ubuntu image from docker)
  - `sudo singularity build --sandbox myubuntu/ docker://ubuntu`
    create writable sandbox directory named *myubuntu*
  - `sudo singularity shell --writable myubuntu` — don't forget or silently lose all your changes
    work inside sandbox. install and test OS utilities & software — record steps to prepare automated build
- prepare container for use on HPC system
  - `sudo singularity build myubuntu.simg myubuntu/`
    convert sandbox to immutable (squashfs) production image. — better: prepare build script and automate build
  - `scp myubuntu.simg user@hpc-cluster:/scratch/user`
    ship container to HPC cluster
- test and use container on HPC system
  - `ssh user@hpc-cluster`
    `cd /scratch/user ; module load singularity [openmpi/version]`
    login and set up environment
  - `[mpirun -np n] singularity exec myubuntu.simg command [arg ...]`
    use container (interactive or batch). You have access to local directories e.g. $HOME and $SCRATCH

Manual Walkthrough Demo - Hands On (1)

Web page with details:

https://www.uibk.ac.at/zid/mitarbeiter/fink/singularity-2018/workshop-exercises.html#HDR1

- Prepare your PC and LCC account
  - Connect your laptop to Wifi
  - (Optional for Linux) start uibk-singularity-workshop virtual machine
  - Logon to your LCC account and create symlink $HOME/Scratch → $SCRATCH
  - Install Singularity

- Create and test your first container
  - Create container in sandbox
  - Start container writable shell as root
    - update OS
    - install OS utilities (vim, nano, less, gcc, make, wget)
    - download sample programs mycat.c and hello.c ; compile, and test in /data (SEE NEXT SLIDE)

- Transfer and use container on HPC system
  - Convert container to squashfs
  - Test container image on local machine (non-root)
  - Transfer to LCC2, test on remote machine

Manual Walkthrough Demo - Hands On (2)

Sample programs hello.c mycat.c

Trivial programs to demonstrate typical capabilities of UNIX programs

- process command line arguments
- read data from named files and stdin
- write data to stdout

hello.c

simply echoes all its command line arguments. Examples

```
$ hello a b c
hello a b c
$ ./hello one two three
./hello one two three
```

mycat.c

works like simplified version of cat(1) UNIX program, but prints header for each file read and line numbers

usage: `mycat [ file ... ]`

reads files (default / --): stdin

writes concatenated output to stdout with file headers and line numbers

Overview

Preliminaries

- Why containers
- Understanding Containers vs. Virtual Machines
- Comparison of Container Sysesms (LXC, Docker, Singularity) - why Singularity?
- Containers and Host Resources

Using Singularity

- Singularity Workflow

1. **Manual Walkthrough Exercise**

Understanding Singularity

- Web resources
- Container Image Formats, Sources, Conversions
- Running Containers

Advanced Singularity

- Automating Creation of Containers
- Container Contents, Cache Files

2. **Exercise - Build Container using Build File**

- Using MPI with Singularity

3. **Exercise - MPI Job with Container**

- Singularity Instances

Understanding Singularity: Documentation, Getting Help, Getting Software

Web resources

• https://www.sylabs.io/singularity/	Official Web Site
• https://www.sylabs.io/docs/	Singularity Documentation
• https://github.com/sylabs/singularity	Github: Software Download
• https://hub.docker.com/explore/	Docker Hub Repositories
• https://www.uibk.ac.at/zid/systeme/hpc-systeme/	UIBK HPC Home Page - search singularity
• https://www.uibk.ac.at/zid/mitarbeiter/fink/singularity-2018/	Material for present workshop

Singularity Command Help Utility

- singularity --help
- singularity *subcommand* --help

Understanding Singularity: Container Image Formats

Concept: container image

- Private copy of root file system: OS (except kernel) + utilities + libraries + permanent data for container

Types of container images (usable as *container path* in singularity commands)

- Sandbox directory
  - directory tree in host file system
  - mounted (read-only or read-write) as root inside container
  - create with `sudo singularity build --sandbox` *`name`* *`source`*
  - modify with `sudo singularity shell --writable` *`name`*
  - use to create, modify, and test user defined software environment

- Immutable squashfs image file
  - read-only image of root tree in single file
  - mounted read-only as root inside container
  - create with `sudo singularity build` *`name`*`.simg` *`source`*
  - use to test, deploy, and run software environment

- Legacy writable filesystem image file - deprecated
  - previously only available format - corruption + capacity issues
  - create with `sudo singularity build --writable` *`name`*`.img` *`source`*

Understanding Singularity: OS Image Sources

From where can singularity take OS images?

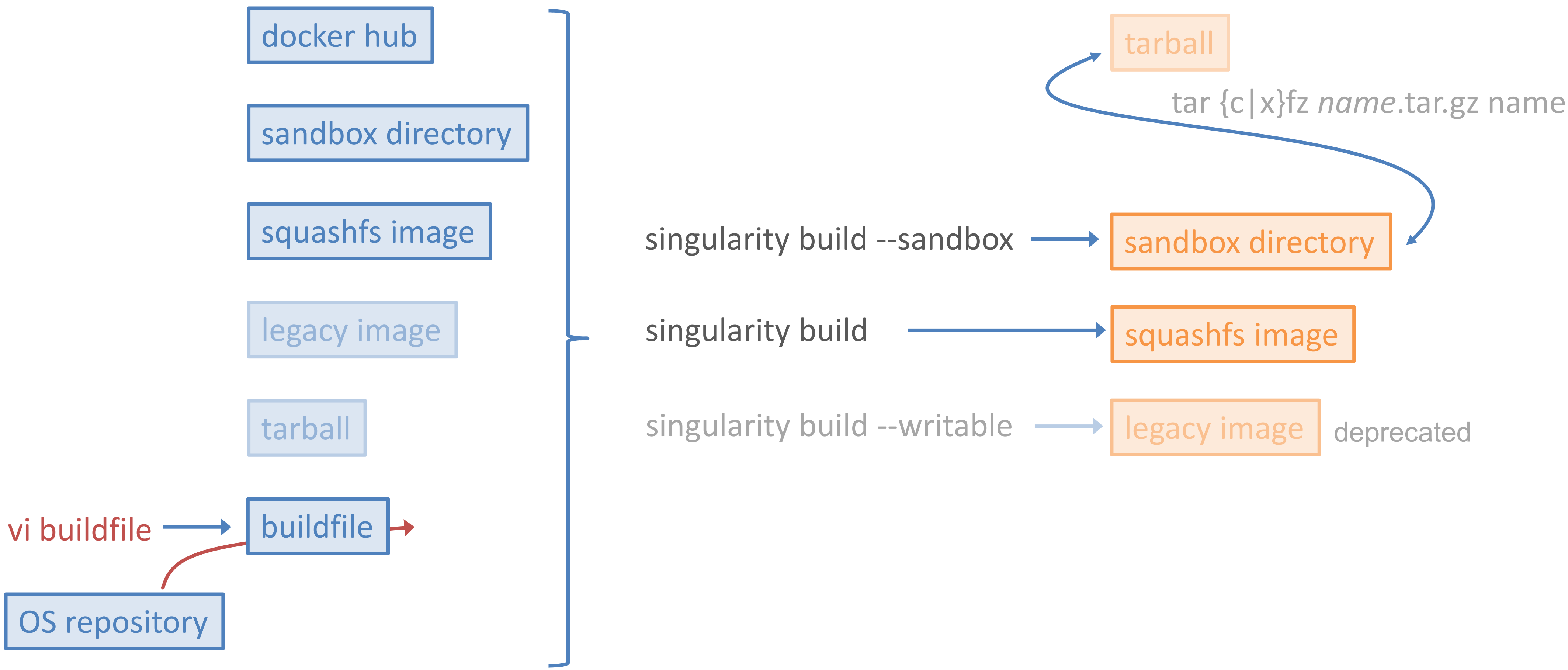
Types of image *sources* (confusing: Singularity documentation calls these *targets*)

source type	example		
• dockerhub	docker://*name*[:*release*]		points to a Docker registry (default Docker Hub)
e.g.	docker://ubuntu:bionic docker://centos:7.5		
• directory	*dirname*		Directory structure containing a root file system (typically from earlier build)
• image	*name*.simg	*name*.img	Local image on your machine (squashfs or legacy)
• tarball	*name*.tar.gz		Archive file containing above directory format (file name must contain "tar")
• buildfile	*name*.build	*name*.def	Buildfile = text file with recipe for building a container (explained later in *automation*)
• OS repository (Centos, Ubuntu, ...)			install OS from scratch - only via buildfile

- not available as source: ISO image for OS installation from scratch.
  Instead: use Dockerhub or OS repository to download clean OS image and then modify

Understanding Singularity: Converting Container Formats

singularity build [--option ...] *container-image-to-build* *source-spec*



Understanding Singularity: Running Containers

singularity exec [*options*] *container-path* *command* [*arg* ...]
execute any *command* in container

run [*options*] *container-path* [*arg* ...] purpose: define your own commands
start run-script /singularity [*arg* ...] within container

shell [*options*] *container-path*
start interactive shell in container

important common options

-w | --writable (only sandbox or legacy image) Container is writable (must use sudo)

-B | --bind *outsidepath*[:*insidepath*[:*options*]]
bind mount *outsidepath* (host) as *insidepath* (container) using *options* { [rw] | ro }

-H | --home *outsidehome*[:*insidehome*]
override mount of $HOME directory.
recommended: use subdirectory of $HOME
to prevent two-way leakage of config files / shell history, etc.

what happens

- command / runscript / shell is executed in container.
- I/O to $HOME, mounted directories, and /dev is possible
- Program has access to host's networking
- stdin, stdout and stderr are connected to running program

Using Singularity - Some Practical Considerations, esp. for UIBK

- Singularity activated by modules environment
  module load singularity/2.x   no access to older versions (security fixes)

- Which directories are mounted on UIBK clusters
  $HOME   (by default)
  $SCRATCH   (UIBK configuration)   need to create mount point /scratch in container for this to work

- CAVEAT: Re-Uploading Squashfs image may damage your jobs
  Be sure NOT to overwrite your Squashfs image while jobs are still running
  → jobs will malfunction and flood sysadmins with error messages

- Installing your own software inside container
  - Use tools like apt (Debian/Ubuntu) or yum (RedHat/CentOS) to install system components and prerequisites
  - Install your own stuff into unique directory (e.g. `/data`) which is unlikely to be clobbered by other components
  - Be sure NOT to install to $HOME (which is outside the container)

- Portability, compatibility, reproducibility
  Containers help mitigate the replication crisis
  But: still need compatible OS infrastructure on host (e.g. kernel, MPI libraries, ...)
  Example: OpenMPI 1.10.2 no longer compiles on Centos 7.4 → Ubuntu 16.04 MPI containers not usable

Overview

Preliminaries

- Why containers
- Understanding Containers vs. Virtual Machines
- Comparison of Container Systesms (LXC, Docker, Singularity) - why Singularity?
- Containers and Host Resources

Using Singularity

- Singularity Workflow

1. **Manual Walkthrough Exercise**

Understanding Singularity

- Web resources
- Container Image Formats, Sources, Conversions
- Running Containers

Advanced Singularity

- Automating Creation of Containers
- Container Contents, Cache Files

2. **Exercise - Build Container using Build File**

- Using MPI with Singularity

3. **Exercise - MPI Job with Container**

- Singularity Instances

Automating Creation of Containers Using Build Files

Build file := text file containing directives how to build containers

Use `singularity build help` to get template and valid combinations

Starts with BASEOS specs (specifies *source* of OS), e.g.

```
Bootstrap: {docker|yum|debootstrap|localimage}
From: source-spec
MirrorURL: http://location-of-installation-material
```

Continues with several sections, each having format

```
%section-header
lines with shell commands (scriptlet)
```

Sections are executed in host or container at different times during build process and runtime of container

%setup	executed during build	on host	after creation (bootstrap) of container.
	Use $SINGULARITY_ROOTFS to reference container contents.		
%post	executed during build	inside container	after %setup.
	add here all commands to setup your software		
%files	used during build - not a scriptlet, but pairs */path/on/host* */path/inside/container*.		
	executed too late to be useful - use %setup instead		
%test	executed during build	inside container	after %post
	define any test commands to test correct setup of container. Exit status used to determine success		

%environment	executed at start of runtime	inside container	when singularity { run | exec | shell } is used
	purpose: set all desired environment variables		
%runscript	executed at runtime	inside container	when singularity run is used
	purpose: define your own set of commands		

Container Directory Contents (Version 2.5)

After build, container root directory will contain:

1. Complete copy of OS installation from source
2. Singularity-specific data
   - you can modify these in sandbox after build while developing container, but...
   - ... remember to update your build file and rerun build for final version (consistency!)

`/.singularity.d/`

`actions/`	Implementation of Singularity commands
`env/`	Several environment scripts, including contents of your %environment scriptlet
`runscript`	Contents of your %runscript scriptlet
`test`	Contents of your %test scriptlet
`Singularity`	Your complete build file

`/environment -> .singularity.d/env/90-environment.sh`	Link to your %environment scriptlet
`/singularity -> .singularity.d/runscript`	Link to your %runscript scriptlet

```
/.exec       -> .singularity.d/actions/exec
/.run        -> .singularity.d/actions/run
/.shell      -> .singularity.d/actions/shell
/.test       -> .singularity.d/actions/test
```

Implementations of commands exec, run, shell, etc.
first, source all scripts in `/.singularity.d/env/*.sh`
then run your command, runscript, or shell

3. Results of all your %setup, %post, and %files actions

Cache and Temporary Directories

Singularity uses the following host directories

- $HOME/.singularity — cache of material downloaded from dockerhub + more
  override with export SINGULARITY_CACHEDIR=/alternate/directory
  disable with export SINGULARITY_DISABLE_CACHE=yes

- /tmp — temporary directories during squashfs image creation.
  must be large enough to hold entire tree
  override with export SINGULARITY_TMPDIR=/alternate/directory

Using Singularity - More Practical Considerations

- Warning
  Carefully inspect build file before running singularity build

  One error in build file can clobber your host (*)

  - All section header lines must begin with %*xxxx* - no blanks allowed

  - The %setup section is run as root user with no protection
  - (*) If the setup process continues into the %post section (for whatever reason - e.g. typo), all actions intended for container will affect host instead

  - This is why we recommend using a Linux VM even when building containers under Linux
  - Take VM snapshot before using untested build files

- Files in Sandbox directory created by `sudo singularity build -s` *`mydir`* are owned by root
  - Remove a sandbox directory *mydir* as root
    `sudo rm -rf` *`mydir`*

Exercise: Build Container using Build File

Web page with details:

https://www.uibk.ac.at/zid/mitarbeiter/fink/singularity-2018/workshop-exercises.html#HDR2

In this exercise, we will demonstrate automatic creation of a container using the build command and compile and use two sample programs to test shell integration of Singularity runs

- Download build file automate.build and sample programs mycat.c hello.c
- Create container squashfs image
- Repeat experiments of first exercise

Using MPI with Singularity

Concept

Normal MPI usage

- mpirun starts *n* MPI processes on same or different hosts using mpirun command
- MPI runtime uses MPI libraries linked into programs for interprocess communication

Singularity: use host's mpirun to start container processes and communicate with batch system

- advantage: Singularity batch and MPI integration is trivial (cf. competitors)
- but: MPI runtime on host needs to communicate with MPI libraries in container
  - → need identical MPI version in host and container

How to

- Build container using MPI from container's OS (or download and build MPI sources)
- Make sure identical MPI exists on target system (else request installation)
- Upload container image
- Create SGE batch script

```
#!/bin/bash
#$ -pe openmpi-xperhost y
#$ -cwd

module load openmpi/2.1.1 singularity/2.x
mpirun -np $NSLOTS singularity exec mycontainer.simg /data/bin/my-mpi-program
```

Exercise: Build and Run Container with MPI

Web page with details:

https://www.uibk.ac.at/zid/mitarbeiter/fink/singularity-2018/workshop-exercises.html#HDR3

In this exercise, we will download and compile one of the OSU benchmark programs to demonstrate MPI integration of Singularity

- Download build file   mpitest.build   and batch script   mpitest.sge
- Create container squashfs image and copy to LCC2

- Run test locally on login node
- Submit test job to SGE

- Compare test results

Singularity Instances (preview)

Problem

Memory congestion w/ multiple singularity processes on same host (e.g. MPI jobs)

- Each singularity exec separately mounts its container file system (squashfs)
- Buffer cache not shared → system memory flooded with identical copies of file system data

Solution

- Have several container processes share the same squashfs mount on each host → shared buffer cache

Singularity instance

- singularity instance.start [-H *homespec*] [-B *mountspec*] *container-path* *inst-name*
  start named instance of given container, create mounts and name spaces,
  but do not start any processes in container

- singularity exec instance://*inst-name* *command* [*arg* ...]
  start program in namespace of given container instance
- singularity instance.list
  list all active container instances
- singularity instance.stop [-f] [-s *n*] *inst-name*
  stop named instance
  -f force by sending KILL signal... -s *n* sent signal *n* ... to running processes (*n* numerical or symbolic)

Caveat: take extreme care that containers are cleaned up at end of job (limited resource - experiments underway)

Singularity Instances Example: MPI Job on SGE Cluster

```
#!/bin/bash
#$ -N mpi-inst
#$ -j yes
#$ -cwd
#$ -pe openmpi-2perhost 4
#$ -l h_rt=500

module load singularity/2.x openmpi/1.8.5
singularity=$(which singularity)
cwd=$(/bin/pwd)

awk '{print $1}' $PE_HOSTFILE |
   parallel -k -j0 "qrsh -inherit -nostdin -V {} $singularity instance.start $cwd/insttest.simg it.$JOB_ID"

sleep 5  # allow instances to settle

time mpirun -np $NSLOTS $singularity run instance://it.$JOB_ID my-program

sleep 5

time awk '{print $1}' $PE_HOSTFILE |
   parallel -k -j0 "qrsh -inherit -nostdin -V {} $singularity instance.stop it.$JOB_ID"
```

thank you