CS 521: Systems Programming

Containers

Lecture 16

Today's Schedule

- Container Background
- Playing with cgroups
- Docker

Today's Schedule

- **Container Background**
- Playing with cgroups
- Docker

What's a Container?

- Many of you have probably heard about or worked with containers already
- So, what are they?
  - *A stricter form of isolation from other processes*
  - They behave kind of like a VM, but run on the **same kernel** as the host
    - Can't virtualize another OS, like running Windows on Linux

Containing Processes

- You can run one or more processes in a container
- These processes behave just as they did before, except we impose more restrictions on them
  - They might not be allowed to see other processes outside the container
  - Maybe they aren't allowed to use all the CPU cores or RAM
  - Maybe we don't let them access the network
  - Or maybe they can't access certain files
- That's really it. Same as before, but with more **lies** 

A More Formal Description

- Containers are actually an evolution of the BSD *jail* concept
  - If you have an unreliable server process, put it in a jail
  - If compromised, the jailed process is less likely to be able to take over the entire system
- Processes in a container are often made to believe that they are running on their own machine
  - Isolated from all other processes on the host

Flexibility

- We can dynamically change resource allocations to the container, unlike VMs (usually)
  - CPU usage allowed, memory, network/disk I/O, etc.
- Containers are much faster to start, stop, and manipulate
  - No need to go through a long boot process
  - However, many containers start with an `init` process that will launch all background services as usual

cgroups, namespaces

- On linux, ***control groups*** and ***namespaces*** allow rapid changes to how resources are allocated
- What to limit disk write speed, CPU usage, etc.? Do it on the fly via the cgroups API
- A decent amount of this infrastructure was built at Google for their "Borg" project
  - Cluster orchestration at massive scale

Control Groups Functionality

- Resource limits
  - Example: placing upper bounds on memory
- Prioritization
  - Example: give certain groups higher CPU priority
- Accounting
  - Allows us to monitor resource consumption (can be used for billing, de-prioritizing groups, etc)
- Control
  - Checkpointing, freezing, restarting

Namespace Isolation

- Cgroups give us control over resources, but don't paint the full picture
- Namespaces give us **container isolation**
- Users, process IDs, hostnames, timezones can all be distinct from the host OS
  - (Even though they're running the same kernel)
- Can have separate mount points (both physical and virtual devices)
- Maybe most importantly: network isolation so the container appears to be its own network host

Today's Schedule

- Container Background
- **Playing with cgroups**
- Docker

Using cgroups

Let's play with `cgroups` functionality to limit the CPU usage of particular processes.

First, determine what controllers are active:

```
$ cat /sys/fs/cgroup/cgroup.subtree_control
memory pids
```

If you don't see the file above, then you aren't running cgroups2. (Are you on your VM?)

Enable CPU Controllers

Now let's enable CPU controllers. First, become root `sudo su -` and then:

```
$ echo '+cpu' > /sys/fs/cgroup/cgroup.subtree_control
$ echo '+cpuset' > /sys/fs/cgroup/cgroup.subtree_control
$ cat /sys/fs/cgroup/cgroup.subtree_control
cpuset cpu memory pids
```

Create a cgroup

```
$ cd /sys/fs/cgroup/
$ mkdir group-name-here
$ cd group-name-here
$ ls

cgroup.threads            memory.min
cpu.max                   memory.oom.group
cpu.pressure              memory.pressure
cpuset.cpus               memory.stat
cpuset.cpus.partition     memory.swap.events
cpu.uclamp.min            pids.max

... Whoa, where did all those files come from?! ...
```

Adding Processes to the Group

```
# Do 'echo PID' > cgroup.procs' to add a process

$ echo 1000 > cgroup.procs
$ echo 1001 > cgroup.procs
$ echo 1002 > cgroup.procs
$ echo 1003 > cgroup.procs
```

Adding Rules

```
# Which CPUs to use (comma-separated):
$ echo 1,2 > cpuset.cpus

# Set the maximum CPU usage to the default (100%):
$ echo "max 100000" > cpu.max

# Set maximum CPU usage to 50% (100000 * 0.5)
$ echo '50000 100000' > cpu.max
```

Tweaking the Rules

- You can add processes to group(s) whenever you'd like
- Resource limits can be changed dynamically
  - Try changing maximum CPU usage to 50% and watch the output of `top`
  - Change the `cpuset.cpus` and note how only the specific CPUs you choose are used
- And of course, CPU usage is only the most common example. You can always limit memory, disk usage, network, and more…

Today's Schedule

- Container Background
- Playing with cgroups
- **Docker**

So What About Docker?

- Wait, doesn't docker == containers?
- Well, not really.
- Docker provides an easy-to-use interface for working with Linux cgroups/namespaces
  - Btw: if you run Docker on macOS or Windows, it's virtualizing Linux!
- It makes it easy to compose containers by building off base images, adding packages and code, and isolating your container from the rest of the system

Interfacing with the Container API

- So… docker basically is an interface for working with containers
  - Alternatives: systemd-nspawn, podman, etc…
- We can use Docker to package up our software and deploy it anywhere
  - No need to hunt down packages or figure out how to install things. It's like a mini VM ready to go
    - With bare metal performance (on Linux at least)
  - Best of all, they're isolated. You control how much access they get.

An Aside: The Future of Docker

- Currently the future of docker looks pretty bleak, but we'll see what happens
  - Their tools/ecosystem are free, so now they're trying various avenues to "monetize" them
    - Like letting you stay on specific versions is now a "pro" feature
- Most of the "magic" Docker provides is in the Linux kernel
  - Their main advantage: good tooling/interfaces
    - Except others are copying them now…

One More Thing: Kubernetes

- You might have heard of Kubernetes…
- Open source version of the "Borg" cluster management built at Google
  - Or at least a close relative
- Kubernetes allows for orchestration of multiple containers over large sets of machines
  - If you're google, you don't want to worry about managing tons of physical OR virtual machines
  - Can be used in smallish deployments as well to manage all your containers

Installing Docker

Okay, to start off, let's try simply *running* some prepackaged software with Docker. First, we need to install it, then enable and start its *daemon*.

```
# Install docker:
$ sudo pacman -Sy docker

# Enable and start the docker daemon:
$ sudo systemctl enable docker
$ sudo systemctl start docker
```

Next, running software…

Running with Docker

- Let's say that we want to play some games
- We'll spin up a RetroArch container to do this:
  - `sudo docker run --rm -p 8080:80 \ inglebard/retroarch-web`
    - `inglebard/retroarch-web` : the software we're going to run. Docker will automatically download it
    - `--rm` : remove container after it finishes running
    - `-p` : map internal container **ports** to the outside machine (port 80 inside the container gets mapped to 8080 on the host)
- But… it's running on the VM. How do we get there?

Forwarding Ports with ssh

- We need a way of accessing our VM from our local development machines
- Let's forward the ports with `ssh`:
  - `ssh deltron -L 8080:localhost:8080`