Linux Isolation

Introduction

Linux provides several utilities to help isolate processes. Many of these utilities are the foundation for containers like Docker.

Isolation (namespaces)

There are several levels of isolation that can be achieved through the unshare utility/API…

```
$ unshare --help

Usage:
 unshare [options] <program> [<argument>...]

Run a program with some namespaces unshared from the parent.

Options:
 -m, --mount[=<file>]      unshare mounts namespace
 -u, --uts[=<file>]        unshare UTS namespace (hostname etc)
 -i, --ipc[=<file>]        unshare System V IPC namespace
 -n, --net[=<file>]        unshare network namespace
 -p, --pid[=<file>]        unshare pid namespace
 -U, --user[=<file>]       unshare user namespace
 -f, --fork                fork before launching <program>
     --mount-proc[=<dir>]  mount proc filesystem first (implies --mount)
 -r, --map-root-user       map current user to root (implies --user)
     --propagation slave|shared|private|unchanged
                           modify mount propagation in mount namespace
 -s, --setgroups allow|deny  control the setgroups syscall in user
namespaces

 -h, --help     display this help and exit
 -V, --version  output version information and exit

For more details see unshare(1).
```

Mount Isolation

The followings block makes it so that any mounts created will only be visible to your process and its children…

```
$ sudo unshare -m /bin/bash
# mkdir -p /tmp/mount_tmp
# mount -n -o size=1m -t tmpfs tmpfs /tmp/mount_tmp/
# cd /tmp/mount_tmp
# ls -al
total 8
drwxrwxrwt 2 root root   60 Mar  4 10:34 .
drwxrwxrwt 8 root root 4096 Mar  4 10:34 ..
-rw-r--r-- 1 root root    3 Mar  4 10:34 t.txt
```

The /tmp/mount_tmp mount won’t be visible to any other processes.

> **NOTE**: Adapted from https://www.endpoint.com/blog/2012/01/27/linux-unshare-m-for-per-process-private.

Mount Image

The following block will create a 50mb image and mount it to a directory as read/write…

```
$ dd if=/dev/zero of=file.img bs=1M count=50
$ mkfs -F file.img
$ sudo mkdir -p /tmp/mount_tmp/
$ sudo mount -o loop,rw,sync file.img /tmp/mount_tmp
$ sudo chown -R userA:groupA /tmp/mount_tmp
$ # now userA can write in the image in /tmp/mount_tmp
$ # unmount when finished
$ sudo umount file.img
$ # compress image, original will be gone unless -k flag used
$ gzip file.img
```

As long as the application writes out all relevant data to the mounted directory, you can checkpoint by compressing and stowing away the image file.

For applications that write a lot of temporary data into the mount/image, that data will still be there even if the files are removed. As such, consider filling the empty space on the mount with zeros prior to compressing (removing random garbage data increase compressibility). See https://superuser.com/a/528547 for a common Linux utility that zeros out space.

> **NOTE**: Aside from the mkfs executable, there are also executables unique to the filesystem types mkfs.ext3, mkfs.ntfs, mkfs.ext4, etc..
>
> **NOTE**: Apated from http://ubuntuhak.blogspot.ca/2012/10/how-to-create-format-and-mount-img-files.html.

Mount as Root

The following block will change the root directory to some other directory on the filesystem...

```
$ sudo cp -a /lib ./
$ sudo cp -a /lib64 ./
$ mkdir bin
$ sudo cp /bin/bash ./bin/
$ sudo chroot /tmp /bin/bash
# ls -al
bash: ls: command not found
# ls
bash: ls: command not found
# dir
bash: dir: command not found
# /
.ICE-unix/  .Test-unix/ .X11-unix/  .XIM-unix/  .font-unix/ bin/
lib/        lib64/
```

```
# /
.ICE-unix/  .Test-unix/ .X11-unix/  .XIM-unix/  .font-unix/ bin/
lib/        lib64/
# /
```

The major problem with this is that none of the Linux libraries/executables will be findable when you chroot. The typical way around this to copy /usr/bin /bin /lib /lib64 into the directory you're chrooting, such that you have access to all the basic Linux tools. This may cause problems with non-trivial stuff such as driver problems, access to /proc or /dev, etc…

In the above examples, /lib and /lib64 were copied, but only the bash executable was moved over. As such, none of the commands like echo, ls, and dir won't work.

> **NOTE**: Adapted from https://unix.stackexchange.com/a/416556.

PID Isolation

The following block allows you to isolate other PIDs from your process and its children…

```
$ sudo unshare --fork --pid --mount-proc /bin/bash
# ps uax
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.2  0.2  22340  5004 pts/5    S    10:14   0:00 /bin/bash
root          15  0.0  0.1  37364  3372 pts/5    R+   10:14   0:00 ps uax
# bash
# ps uax
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.2  22340  5004 pts/5    S    10:14   0:00 /bin/bash
root          16  0.0  0.2  22340  4888 pts/5    S    10:14   0:00 bash
root          30  0.0  0.1  37364  3268 pts/5    R+   10:14   0:00 ps uax
#
```

Other processes will still be able to see your PIDs (unless they're also isolated), but your processes will only see your PIDs.

> **NOTE**: Adapted from https://jvns.ca/blog/2016/10/10/what-even-is-a-container/.

Background Processes

One problem with starting a namespace is that nohup doesn't seem to work. If you create a new PID namespace, run a process through nohup, and then kill the bash shell that started that process, the process will not survive. It'll be gone…

```
$ sudo unshare --fork --pid --mount-proc /bin/bash
# nohup stress --cpu 1 --timeout 120 &  # won't be around if term killed
```

The workaround for this is to dump your command to a file and use nohup on the unshare command that starts the bash shell…

```
$ echo stress --cpu 1 --timeout 120 > a.sh
$ sudo echo hi # required so next sudo cmd doesn't ask for passwd
$ sudo nohup unshare --fork --pid --mount-proc /bin/bash /home/user/a.sh &
```

Killing Processes by SID

SID stands for session ID, which is the session leader (top-most parent process for the "session"). You can use bash and the ps tool to figure out what the SID is, then kill all the processes in that SID…

```
$ echo $$  # get pid
2277
$ ps -o 'user,pid,sid,args' --pid=2277  # get sid for pid
USER        PID   SID COMMAND
user       2277  2124 bash
$ ps -o 'user,pid,sid,args' --sid=2124  # get all pids in sid
USER        PID   SID COMMAND
user       2124  2124 /bin/bash
user       2277  2124 bash
user       2296  2124 ps -o user,pid,sid,args --sid=2124
```

Killing Processes by Namespace

You can use bash and the ps tool to figure out what the PID namespace is, then kill all the processes in that namespace…

```
$ sudo unshare --fork --pid --mount-proc /bin/bash
# ls -al /proc/self/ns/pid
lrwxrwxrwx 1 root root 0 Mar  5 11:59 /proc/self/ns/pid -> pid:[4026532189]
# readlink /proc/self/ns/pid
pid:[4026532189]
# ps -o 'user,pid,pidns,args'
USER        PID      PIDNS COMMAND
root          1 4026532189 /bin/bash
root         39 4026532189 ps -o user,pid,pidns,args
```

User Isolation

The following block allows you to hide the running user for your process and its children…

```
$ whoami
user
$ sudo unshare -U /bin/bash
$ whoami
nobody
$ exit
$ whoami
user
```

Hostname Isolation

The following block allows you to change the hostname (and domain name) of the machine but only for your process and its children…

```
$ hostname
user-VirtualBox
$ sudo unshare -u /bin/bash
# hostname
user-VirtualBox
# hostname my-new-hostname
# hostname
my-new-hostname
# exit
exit
$ hostname
user-VirtualBox
```

> **NOTE**: Adapted from https://medium.com/@teddyking/linux-namespaces-850489d3ccf.
> See https://unix.stackexchange.com/q/183717.

Resources (cgroups / control groups)

There are several resources that can be controlled through the /sys/fs/cgroup filesystem…

```
$ tree /sys/fs/cgroup
/sys/fs/cgroup
 ├──── blkio
 │     ├──── blkio.io_merged
 │     ├──── blkio.io_merged_recursive
 │     ├──── ...
 ├──── cpu -> cpu,cpuacct
 ├──── cpuacct -> cpu,cpuacct
```

```
├──── cpu,cpuacct
│     ├──── cpuacct.stat
│     ├──── cpuacct.usage
│     ├──── ...
├──── cpuset
│     ├──── cpuset.cpu_exclusive
│     ├──── cpuset.cpus
│     ├──── ...
├──── devices
│     ├──── devices.allow
│     ├──── devices.deny
│     ├──── ...
├──── memory
│     ├──── memory.failcnt
│     ├──── memory.force_empty
│     ├──── ...
├──── net_cls -> net_cls,net_prio
├──── net_cls,net_prio
│     ├──── net_cls.classid
│     ├──── net_prio.ifpriomap
│     ├──── ...
└──── net_prio -> net_cls,net_prio
```

A problem with resources is that, because of systemd, there’s multiple places from which you can control them. Keep this in mind when setting resources.

CPU Scheduler

There are 2 ways to control the share of processing power for a process and its children: shares and CFS.

Shares

The first way to control CPU scheduling is through cpu.shares…

```
$ sudo cgcreate -a user:user -t user:user -g cpu:test
$ echo 128 > /sys/fs/cgroup/cpu/test/cpu.shares
$ sudo cgexec -g cpu:test firefox
$ sudo cgdelete -g cpu:test  # delete
```

The number of shares is relative to all cgroup shares. In the example above, I specified 128, but there's another cgroup set to 1024. That means I'm guaranteed a minimum of 8% of the CPU power (128/1024). If the processes in the other cgroup are idle, my cgroup can be greedy and take those idle cycles (and vice versa), but my cgroup will always be guaranteed at least 8%.

Had I set to my cgroup to 1024, I'd get a minimum of 50% of the processing time. Had I set to to 2048, I'd get a minimum of 66% of the processing time. The algorithm to determine how much processing time you get is...

shareA / (shareA + shareB + shareC + ...)

The numerator is the share for your cgroup and the denominator is the sum of all cgroup shares.

/sys/fs/cgroup/cpu/cpu.shares specifies the shares for non-cgroup'd processes (defaults to 1024 I think). The scale at which your shares are set should be proportional to this value. For example, if you want non-cgroup'd processes to have a super low priority, you can scale your cgroup shares such that the non-cgroup share default will get computed to a 1% minimum.

> **NOTE**: the above paragraph of defaulting non-cgroup'd processes may be a systemd thing, see https://stackoverflow.com/a/49088329/1196226. I can't find anything in the systemd or cgroup filesystem that sets this to 1024.

CFS

The second way to set CPU shares is through cpu.cfs_period_us and cpu.cfs_quota.us...

```
$ sudo cgcreate -a user:user -t user:user -g cpu:test
$ cat /sys/fs/cgroup/cpu/cpu.cfs_period_us
100000
$ cat /sys/fs/cgroup/cpu/cpu.cfs_quota_us
-1
$ echo 2000 > /sys/fs/cgroup/cpu/test/cpu.cfs_quota_us
$ sudo cgexec -g cpu:test firefox
$ sudo cgdelete -g cpu:test  # delete
```

The important thing here is cfs_quota_us -- the cfs_period_us is something that you probably should never touch. In the example above, we force the scheduler to only ever give us 2% of the CPU (unless there's a higher priority task that needs it). The difference between this method and cpu.shares is that this method won't allow you to consume more than 2% ever, even if you have idle cycles available.

CPU Core Affinity

```
$ sudo cgcreate -a user:user -t user:user -g cpuset:cpuset_test
$ echo 0 > /sys/fs/cgroup/cpuset/cpuset_test/cpuset.cpus   # lock to core 0
$ echo 0 > /sys/fs/cgroup/cpuset/cpuset_test/cpuset.mems  # lock to numa
node 0
$ sudo cgexec -g cpuset:cpuset_test firefox   # run proc
$ sudo cgdelete -g cpuset:cpuset_test  # delete
```

This will lock to core 0. You can lock to multiple cores by range (e.g. 0-5) or commas (e.g. 1,2,5). You can leave out the NUMA locking line if you don't need it.

CPU Scheduler and Core Affinity

```
$ sudo cgcreate -a user:user -t user:user -g cpu,cpuset:test
$ echo 128 > /sys/fs/cgroup/cpu/test/cpu.shares
$ echo 0 > /sys/fs/cgroup/cpuset/test/cpuset.cpus  # you can use CFS
instead if needed
$ sudo cgexec -g cpu,cpuset:test firefox
$ sudo cgdelete -g cpu,cpuset:test  # delete
```

Mixing CPU core isolation and CPU shares has the following behaviour..

1. The processes will ONLY run on the specified cores
2. The processes will take up the specified CPU share from the overall processing power.

That means that if you specify you want 128/1024 shares on core 0 of a 2 core machine, your process will bind to core 0 and it'll take up 10% of the OVERALL processing power in the machine (not 10% of the core).

What happens if you assign more shares than a core is able to provide? The system will max out the core. So in the above example if my process wanted 3/4 shares but was only set to use cpu 0, it would only get 50% of the shares.

Memory

```
$ sudo cgcreate -a user:user -t user:user -g memory:test1
$ echo 33554432 > /sys/fs/cgroup/memory/test1/memory.limit_in_bytes
$ echo 0 > /sys/fs/cgroup/memory/test1/memory.swappiness # no swapping?
$ sudo cgexec -g memory:test1 bash
#
```

This makes about 32mb of memory available to processes in the cgroup. The swappiness value is linked to the likeness of the processes swapping out on OOM? I'm not sure. I haven't been able to decipher the documentation for this.

See swappiness section (section 5.3) of https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt.

See memory.swappiness section of https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-memory.

Device

The device cgroup will isolate processes to a certain set of devices.

See https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-devices for more information.

https://www.kernel.org/doc/html/v4.11/admin-guide/devices.html...

- https://askubuntu.com/a/809337. AMD and Intel devices would be listed under /dev/dri (probably).
- https://github.com/NVIDIA/nvidia-docker/wiki/GPU-isolation-(version-1.0). NVIDIA devices would be listed under /dev/nvidia*.

You need to do a lot more testing before choosing to use this. There's no guarantee that denying devices in this way will make them disappear from the OpenCL/OpenGL/CUDA APIs.