Total Virtualization

Lab 1: Simple Containers

Before you start

You have VM accessible from the university network. IP addresses are there:
https://goo.gl/whAhx8 . Please,try to connect to your virtual machine.
Username is 'tv_lab'
Root password is 'godblessvirtualization'
After login please change your password in aim not to mix your work with others (accidentally, of course) by `passwd` command in console.

On these virtual machines you have all needed environment configured.
However, if you working on your own machine, you have to install all the needed packages:

```
> apt-get update && apt-get upgrade
> apt-get install gcc lxc
```

Docker installation manual is here:
https://docs.docker.com/engine/installation/linux/ubuntulinux/

Assignment

1. Create your own container that is able to run processes via command line using clone().
2. Isolate created container by creating new namespaces for PID, mount and networking.
3. Set up a network connection between parent namespace and child namespace.
4. Put container filesystem in a file by using **loop** and try to create file in your new mount point.
5. Run processor listing, network information and benchmark in your container, LXC and Docker guest containers.
6. Compare results and write a brief report about comparison results.

Assignment context & information

Linux Namespaces

Namespaces in Linux enable a process (or several processes) to have different views of the system than other processes.

There are currently 6 namespaces:

1. Process (processes)
2. Mount (mount points, filesystems)
3. Network (network stack)
4. IPC (System V IPC and POSIX messages)
5. UTS (hostname)
6. User (UIDs)

7. Control groups (hardware resource usage)

Three system calls are used for namespaces management:

- **clone().** Works like *fork()* and *exec()*, but creates a not only a new process, but also adds a new namespace; the process is attached to the created namespace.
- **unshare().** Creates a new namespace and attaches **the current process** to it.
- **setns().** A system call for joining an existing namespace.

More information: man clone, man unshare, man setns, man nsenter
https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces
https://ericchiang.github.io/post/containers-from-scratch/

Process Namespaces

Historically, the Linux kernel has maintained a single process tree. Every time when a system starts, a process with Process ID (PID) 1 is created. This process is a root of process tree and all the other processes are usually created with fork() and exec() system calls. A parent process is able to communicate with child processes and even kill them by sending signals.

Process namespaces allow us to maintain own isolated process trees with their own PID 1 process.

To create a new PID namespace we call the clone() system call with a flag CLONE_NEWPID.

Code example (you can use it as a framework for a project):

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

#define STACK_SIZE (1024 * 1024)

static char child_stack[STACK_SIZE];

static int child_fn() {
  printf("Clone internal PID: %ld\n", (long) getpid());
  pid_t child_pid = fork();
  if (child_pid) {
    printf("Clone fork child PID: %ld\n", (long) child_pid);
  }
  return EXIT_SUCCESS;
}

int main() {
  pid_t child_pid =
  clone(child_fn, child_stack + STACK_SIZE, CLONE_NEWPID | SIGCHLD, NULL);
  printf("clone() = %ld\n", (long) child_pid);

  waitpid(child_pid, NULL, 0);
  return EXIT_SUCCESS;
}
```

/proc filesystem

According to definition, /proc is a mount point of virtual filesystem, which contains information of all running processes and their properties, which generated by kernel in real time. Hence, after creating a new process with new PID and mount namespaces (with CLONE_NEWNS and CLONE_NEWPID flags specified in clone() call), you should provide new mount point for /proc.
If you skip this step, your containerized application will see all processes from parent namespace.

Network Namespace

The child process still has access to system resources, such as network and if it will listen on some port, this port will be inaccessible to other system processes. To create a different set of network interfaces, we can use **CLONE_NEWNET** flag with clone().

Append the following lines of code to child_fn() function to see how it works:

```
printf("Guest network namespace:\n");
system("ip link");
printf("\n\n");
```

After executing this program, you can see that new loopback device is created, but it's status is "DOWN"

Creating virtual network interface between host and guest

To create a network, you need to create two virtual interfaces – one for parent namespace and one for child. You can run the next command from the parent namespace:

```
> ip link add name veth0 type veth peer name veth1 netns <pid>
```

Where <pid> is PID of the process in child namespace observed by the parent.
Hint: to do it in C, use system() function

For enabling both network interfaces you can use `ifconfig` shell command in corresponding parts of system:

```
>ifconfig <interface_name> <ip> up
```

Where <interface_name> should be equal to veth0 or veth1 on host and guest correspondingly.
As <ip> you should put addresses in same subnet, e.g. 10.1.1.1/24 and 10.1.1.2/24

Mount namespace

Creating separate mount namespaces allows you to have a different root for each isolated process and other mount points.

You can create different mount namespace by using CLONE_NEWNS flag. Under a new mount namespace a child process can mount or unmount whatever is needed and it will not affect parent namespace.

For creating image file for guest file system you can use loop mechanism of UNIX. Read man losetup for more information about this approach.

Unintuitive behavior for new mount namespace.

Some Linux distributions, with util-linux version lower than 2.27 (you can check it in Ubuntu by command `> dpkg -l |grep util-linux`) has problem with CLONE_NEWNS flag (and with corresponding unshare command `> unshare --mount`). Problem is in "shared" property, which propagates from kernel to process. As result, it makes all mounts in new namespace shared with parent namespace, too. Starting from util-linux v2.27 it was fixed (according to patch info: https://github.com/karelzak/util-linux/commit/f0f22e9c6f109f8c1234caa3173368ef43b023eb).

Hence, in older versions of util-linux you should remount your root directory to prevent mount sharing between namespaces:

```
> mount --make-private -o remount /
```

And, moreover, call other mounts with --make-private argument, e.g.:

```
> mount -t proc proc /proc --make-private
```

Source:
https://unix.stackexchange.com/questions/246312/why-is-my-bind-mount-visible-outside-its-mount-namespace

Using loop for file system virtualization

For virtualization of filesystem, the easiest way - Linux loop mechanism.

The main idea - to create virtual storage device using file as image, which will contain file system information and all contents. Order of usage:

1. Create zeroed image file (hint: man dd)
2. Setup loop mechanism to use this file as image for virtual storage device (hint: man losetup)
3. Make filesystem on virtual storage device (hint: man mkfs)
4. Mount virtual storage device inside guest namespace for specific mount point (e.g. /home)
5. Create file inside mounted path in guest environment and check in host environment that this file is not exist

Hints: man dd, man mkfs, man mount

Read more: man loop, man losetup

(Bonus) Using ploop as filesystem image

However, loop mechanism has many drawbacks. This was a reason for Virtuozzo to create alternative approach - ploop, more flexible and useful than loop.
We can use ploop to create file systems inside files. Read **man ploop** or the online manual (https://openvz.org/Ploop/Getting_started) to get familiar.

(Bonus) CGroups mechanism - limiting resources for process

In many cases you need to have possibility to make borders for child processes and containers in their need for CPU, RAM, disk space, etc. Linux provides native mechanism for working with such restrictions called CGroups. As we know, in Linux everything is file, and CGroups mechanism is not exclusion - all control over control groups is making by editing files in virtual file system.

Start with creation of new control group for controlling process CPU availability:
```
> sudo mkdir /sys/fs/cgroup/cpu/demo
```

Then system will automatically create all needed “files” in this directory, so you can edit and correspondingly setup your new control group restrictions.
For example, let’s restrict your control group tasks by using only half of CPU in your system:
```
> echo 50000 > /sys/fs/cgroup/cpu/demo/cpu.cfs_quota_us
> echo 100000 > /sys/fs/cgroup/cpu/demo/cpu.cfs_period_us
```
(It means that for every 100.000us of single CPU time your cgroup processes will get only 50.000us)

Add some process to your cgroup:
```
> echo [pid] > /sys/fs/cgroup/cpu/demo/tasks
```

Remove control groups (**do not** use rm-rf):
```
> rmdir /sys/fs/cgroup/cpu/demo
```

Read more:
- Official documentation: https://elixir.bootlin.com/linux/v4.4/source/Documentation/cgroups/cgroups.txt
- More information from Red Hat: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01

LXC containers

LXC is preinstalled on your virtual machine, but for your personal usage you can get it by standard installer, for example:
```
> sudo apt-get install lxc
```

LXC also provides bunch of predefined containers, which you can use as base for your experiments and work. Just start from creating new one using command:
```
> sudo lxc-create -t download -n my-container-name
```

After creation you can start your container and attach console to it by commands:

```
> sudo lxc-start -n demo-container
> sudo lxc-attach -n demo-container
```

More information in official documentation: https://linuxcontainers.org/lxc/getting-started/

Docker containers

Docker is portable container engine, which initially was extension to LXC capabilities (it uses LXC functionality for some possibilities, along with cgroups and Linux kernel) which provides much more additional functions: powerful container orchestration, version control, client tools, huge image registry, support of different host os (Windows and Mac), etc. More information about comparison here:
https://robinsystems.com/blog/containers-deep-dive-lxc-vs-docker-comparison/

For testing you can use standard Ubuntu 16.04 image:

```
sudo docker run -it ubuntu:16.04 /bin/bash
```

Performance benchmarking

For understanding difference between host machine and container guest performance we can use sysbench util. Install it into container using:

```
apt-get update
apt-get install sysbench
```

After that you can run benchmark by command:

```
sysbench --test=cpu --cpu-max-prime=20000 run
```