EX5 - Containers

Due: 16.06.2022

Start early, and write code that you won't mind debugging

Part 1: Theoretical Questions (10 pts)

The following questions are here to help you understand the material, not to trick you. Please provide straightforward answers.
All questions are worth an equal number of points.

1. Describe one general use of Linux pid *namespaces*.
2. How can Linux mount namespaces be used to help isolate a process?
3. Describe one general use of Linux *cgroups*.
4. Explain the use of the clone command, and how it is different from the fork command
5. What does the chroot command do?
6. What is the purpose of procfs? Give an example of its use.

Part 2: Containers (50 pts)

Introduction

The purpose of this assignment is for you to develop a better understanding of how containers such as docker work under the hood, and what levels of isolation such containers can provide. As such, you will construct a container that will run a (somewhat) isolated program inside.

In this assignment, you will create an *executable* that constructs a container and runs a program within the container.

The Container

Your task is as follows:

Create a container that has its own:

1. hostname
2. Process IDs (from within the container the process ID view should only show the processes running within the container with the first process in the container being process 1)
3. root directory of new filesystem
4. proc filesystem

After compilation you should have an executable file container that can be run as follows:

./container <new_hostname> <new_filesystem_directory> <num_processes> <path_to_program_to_run_within_container> <args_for_program>

Verify that you manage to run a terminal within the container by using the above command. Said terminal is run with "/bin/bash". This will be one of our tests. Additionally, note that the *<new_filesystem_directory>* should also be the new root directory and it is a path relative to the root of the host.

<path_to_program_to_run_within_container> should be relative to the root of the container's filesystem, and should be within it.

Finally, *<args_for_program>* is a non-negative number of command-line arguments for the program to be run inside the container, specified by *<path_to_program_to_run_within_container>*. It may contain 0 or more arguments.

You can assume that all the arguments values to the executable *container* are valid.

We recommend using this system image for the procfs and filesystem when debugging. You should unzip it and provide the path to your code when debugging. You're also welcome to try other system images if you wish. We have tested this image on the aquarium computers and it may not work for users working on their own pcs.

From within the container, the view should be only of the first container process (with pid 1) and any processes created from within the container – meaning children/descendants of the first container process.

Additionally, any changes within the container should not propagate out of the container. For example, changing the hostname or chrooting within the container should not affect the environment outside the container.

What to do

1. Create a new process with flags to separate its namespaces from the parents. Allocate a stack for the new process, of size 8192 bytes.
2. From within the new process:
    1. Change the hostname and root directory
    2. Limit the number of processes that can run within the container by generating the appropriate cgroups
    3. Change the working directory into the new root directory
    4. Mount the new procfs
    5. Run the terminal/new program
3. When shutting down the container make sure to unmount the container's filesystem from the host.
4. Delete the files directories you created when you defined the cgroup for the container.

An important point: make sure to call wait() from the process creating the container, after finishing startup of the container, so as not to finish before we are done with the container.

Useful functions

You most likely will need to use the following functions. Read their man pages.
clone()
mount()
chroot()
chdir()
umount()
exec()

sethostname()
mkdir()

Variants of these functions may also be relevant.

Additionally, read up on the following flags, you most likely will find them useful:
CLONE_NEWUTS | CLONE_NEWPID | CLONE_NEWNS | SIGCHLD

Error Messages

The following error messages should be emitted to *stderr*.
Nothing else should be emitted to *stderr* or *stdout*.

When a system call fails (such as memory allocation) you should print a **single line** in the following format:
"system error: *text*\n"
Where *text* is a description of the error, and then *exit(1).*

~~When a function in your code fails (such as invalid input), you should print a **single line** in the following format:~~
~~"container builder error: *text*\n"~~
~~Where *text* is a description of the error, and then return the appropriate return value.~~

Coding Environment

To run and compile your binary you will need sudo permissions. As such permissions are not available to standard cse users, to run your code you will have to use the standard *rundeb10* vm provided on the aquarium computers. We recommend running it using the below command from the terminal for easy mounting of your development folder:
rundeb10 -cow /cs/course/current/os/ex5/ex5-deb10.qcow2 -bind /cs/usr/<CSE_LOGIN>/<PATH_TO_PROJECT_DIR>/ -serial -root -snapshot -- -net user,hostfwd=tcp:127.0.0.1:2222-:22 -net nic,model=virtio

Note that *<CSE_LOGIN>* should be replaced with your cse username and *<PATH_TO_PROJECT_DIR>* with the path of your project files.

You can read more about the vm here.
Do not save anything in the vm, it resets when shut down. Note that to run the binary within the vm you will need to be su. GDB and g++ are both installed within the vm.

For those who wish to develop at home, running a debian 10 vm or ubuntu 20.04 vm should be fine (you will still need to run your code as sudo). We highly recommend testing your code in the *rundeb10* vm as that is the environment in which we will be testing your code.

Part 3: Sockets (40 pts)

Introduction

The goal in this part of the assignment is for you to open a socket between a server and a client.

Your task is to create an *executable* which based on command line arguments will run either a client or server at a port given at the command line argument.

The server will execute terminal commands that the client passes to it through the socket.

After compilation you should have an executable file sockets that can be run as follows:

./sockets client <port> <terminal_command_to_run>

./sockets server <port>

What to do

1. Receive arguments.
2. Connect to port/open socket at port based on arguments
3. If server: run received terminal commands from port. If client: send terminal command to server

Important notes:

1. you should allocate a buffer of size 256B for both the client and server, for sending and reading the data sent through the network.
2. The server should listen to at most 5 clients

Useful functions

You most likely will need to use the following functions. Read their man pages.

system()
connect()
socket()
bind()

Error Messages

The following error messages should be emitted to *stderr*.
Nothing else should be emitted to *stderr* or *stdout*.

When a system call fails (such as memory allocation) you should print a **single line** in the following format:
"system error: *text*\n"
Where *text* is a description of the error, and then *exit(1).*

Submission

Submit a tar file named ex5.tar that contains:

1. README file built according to the course guidelines.
2. All relevant source code files
3. Makefile - your makefile should generate two **executable** files named: *container* and *sockets* when running 'make' with no arguments.

Make sure that the tar file can be extracted and that the extracted files compile.

Guidelines

1. **Read the course guidelines**.
2. Design your program carefully before you start writing it. Think about how to run a terminal from within the container.
3. Make sure your container is actually isolated to the level defined in this assignment.
4. Always check the return value of the system calls that you use.
5. Test your code thoroughly - write test programs and cross-test programs with other groups.
6. During development, use asserts and debug printouts, but make sure to remove all asserts and any debug output from your code before submitting.

Late Submission Policy

Submission time	**16.06, 23:55**	**19.06, 23:55**	**20.6, 23:55**	**21.06, 23:55**	**22.06, 23:55**	**23.06, 23:55**
Penalty	**0**	**-3**	**-10**	**-25**	**-40**	**-100**

Good luck!