Lab - 2 - Resource management and monitoring with cgroups

Introduction

The goal of these exercises is to explore the functionalities of Linux `cgroups`. This mechanism allows to assign resource management and monitoring controllers to processes and, this way, limit their actions or monitor their activity.

This work requires a Linux host, which can be virtualised.

Resources used in the guide: [cgroups.zip]

Mounted `cgroups`

`cgroup` controllers are nowadays set up at boot time. The set of running `cgroup` containers is listed by file `/proc/cgroups`. In each line, you have a readable `cgroup` name, a file system hierarchy index, the number of `cgroups` in that hierarchy and its enabled status.

```
cat /proc/cgroups
```

Note: to increase the readability of the output you may change the terminal's tabbing schema with the command:

```
tabs 16
```

before listing the `cgroups`. To restore the normal tab spacing use the command

```
reset
```

`cgroups` form a hierarchy, which is observable and managed as a file system hierarchy. This hierarchy is created by mounting `cgroup` controllers on the file system mount point `/sys/fs/cgroup`. The list of mounted `cgroups` can be observed with:

```
mount -t cgroup
```

for version 1 `cgroups` or

```
mount -t cgroup2
```

for version 2 `cgroups`.

For version 1, each line represents a `cgroup` hierarchy, and some hierarchies have several controllers; this is the case of:

- `cgroups` `net_cls` and `net_prio`;
- `cgroups` `cpu` and `cpuacct`.

For version 2 there should be a single, unified hierarchy.

Each directory below each hierarchy represents a `cgroup` within that hierarchy. The processes that belong to that group have their PID listed in the `cgroup` file `cgroup.procs`. In each hierarchy, a process cannot belong to more than one `cgroup`.

`cgroups` of a process

The `cgroups` that a process belongs to are listed by the file `cgroup` in the process `/proc` directory. For your current shell, whose `PID` is given by `$$`, its `cgroups` can be listed as follows:

```
cat /proc/$$/cgroup
```

With `cgroups` version 1, you get many lines, since different controllers have different paths under `/sys/fs/cgroup`.

With `cgroups` version 2, which uses a unified tree, you get a single line, because all the controllers can be managed from a single directory (there is no splitting at a higher hierarchical level, as we have with version 1).

By default, new processes belong to the same `cgroups` of their ancestors. This way, any limitations imposed to a process by a `cgroup` will naturally extend to its process descendants, thus encompassing all those processes in the same limitative scope.

Creation and application of new `cgroups`

New `cgroups` can be created in the intended hierarchy, and below a given `cgroup`, with a simple `mkdir` command. However, you need privileges to do so.

Assume you have an application that you want to run with a given set of limits imposed by `cgroups`. We can use the a program to do it: a (limited) fork bomb (`fork-bomb.c`).

This program creates a high amount of processes (though limited to 100), which can be further limited with a `pids` `cgroup`.

Create a console (which runs a shell command interpreter) and check its `cgroup` with the command

```
cat /proc/$$/cgroup
```

The lines presented by the previous command start with a number (zero for `cgroups` version 2), followed by `:` and a `cgroup` name (empty for `cgroups` version 2), followed again by `:` and by the `cgroup` path.

The output should present one or more paths to the shell's `cgroups`, which should be reachable under `/sys/fs/cgroup`. We are interested only in controlling the number of processes in a `cgroup`, so for version 1 use only the path corresponding to the `pids` `cgroup` (should start with the path `/sys/fs/cgroup/pids`). For version 2 use the unique path that is presented.

Since this `cgroup` path can change from system to system, we will simply refer to it as the shell's `cgroup` path and we will use a shell variable (`SCGP`), to refer it. You can set this variable with this command:

```
SCGP=...
```

where the ellipsis stands for the shell's `cgroup` path.

Change the shell's current directory to its `cgroup` path:

```
cd $SCGP
```

and create a directory `p_limit` on it:

```
mkdir p_limit
```

and change the shell's current directory to it:

```
cd p_limit
```

If using `cgroups` version 2, check the contents of the file `cgroup.type`:

```
cat cgroup.type
```

This file indicates the type of the `cgroup`, which by default should be `domain invalid`. With this type, the `cgroup` does not accept threads (or processes) registered directly on it. To allow that, change its type to `threaded` with the following command:

```
echo threaded > cgroup.type
```

Check again the `cgroup` type, it should now be `threaded`.

If using `cgroups` version 2, check the contents of the file `cgroup.controllers`:

```
cat cgroup.controllers
```

This file lists all the controllers that are currently active in this `cgroup`. If it does not present the `pids` controller (the one that enables limiting the number of processes), you must add it with the following command:

```
echo +pids > cgroup.controllers
```

Check again the `cgroup` controllers, they should now include the `pids`.

Upon this command, you should be able to observe that a set of files starting by the stem `pids.` appeared:

```
ls -la
```

We will use this `cgroup` to limit the number of processes that can be created by the fork bomb. Let's say, 10. Thus, see first what is the limited imposed by the new group:

```
cat pids.max
```

You will be presented with the value `max`, which means the maximum value permitted by the `cgroup` above in the hierarchy. To change this value to `10`, run this command:

```
echo 10 > pids.max
```

Run the `cat` command again to verify the new limit of `10` processes in the `cgroup`.

Now, consider the program `cgroup.c`, that launches an arbitrary command within a given set of `cgroups` (this program is prepared to work with `cgroups` of versions 1 and 2):

Change the shell's current directory to one where you can store and compile the fork bomb and this other program.

After their compilation, use `cgroup` to run `fork-bomb` in the newly created `cgroup`:

```
./cgroup $SCGP/p_limit -c fork-bomb
```

Verify that the fork bomb cannot create more than `9` processes.

Since each process created by the fork bomb lasts for `10` seconds, you can observe the presence of their PID in the `cgroup` `cgroup.procs` file:

```
cat $SCGP/p_limit/cgroup.procs
```

Repeat this command until you see that all processes have left the `cgroup` (upon their termination).

Now, if you repeat the controlled launching of the fork bomb again several times, fast, you will see that you will probably succeed only the first time; in the next ones the fork bomb will not work at all. Explain why.

Upon launching the fork bomb with the `p_limit` `cgroup`, you can observe its use by one of the processes in that group (you need to be fast, or to increase the lifetime of the processes created by the fork bomb):

```
cat /proc/`head -1 $SCGP/p_limit/cgroup.procs`/cgroup
```

Now run the fork bomb without the limiting `cgroup`:

```
./fork-bomb
```

In this case, it will be able to create `100` new processes.

Add your actual shell to the `cgroup` we have been using:

```
echo $$ > $SCGP/p_limit/cgroup.procs
```

Execute again the fork bomb, without any control, and see what happens.

Answer this question: how can the shell continue to execute commands while the processes of the fork bomb are still running? All commands? Try a pipeline (a sequence of commands connected by a pipe). Did it work? Explain.

Once useless, you can remove the `cgroup` by acting on the `cgroups` file system:

```
sudo rmdir $SCGP/p_limit
```

Note: you may simply remove the directory (the `cgroup`) without having to remove all the files (`cgroup` attributes) that the `cgroup` contains.

Homework

Experiment to use the `memory` controller to create a `cgroup` that limits the amount of memory a process can use. The limit can be tested with the consecutive allocation of `4KiB` chunks.

2024

PREVIOUS
Project - Flexible, Risk Aware Authentication System

NEXT
Lab - 3 - File capabilities and files' extended attributes

Last updated on 8 Mar 2024