MCS: Identification, Authentication and Authorization 2020-21

Practical Exercises: Resource management and monitoring with `cgroups`

April 2, 2022 Due date: no date

Changelog

- v1.0 - Initial version.

1 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.

2 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`.

3 `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
```

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 descendance, thus encompassing all those processes in the same limitative scope.

4 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 following program to do it: a (limited) fork bomb (fork-bomb.c):

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
    printf( "Initial PID: %d\n", getpid() );

    for (int i = 0; i < 100; i++) {
        switch (fork()) {
        case 0:
             sleep( 10 );
             exit( 0 );
        case -1:
            printf( "\nError creating process %d\n", i );
            exit( errno );
        default:
            putchar( '.' );
            fflush( stdout );
            continue;
        }
    }

    printf( "\n" );

    return 0;
}
```

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

For a UID of 1000 (which you probably have in your Linux system), and cgroups version 1, create a pids cgroup, as follows:

```
sudo mkdir /sys/fs/cgroup/pids/user.slice/user-1000.slice/p_limit
```

Alternative, you can use this command, that works for whatever UID you have:

```
UID=‘id -u‘
sudo mkdir /sys/fs/cgroup/pids/user.slice/user-$UID.slice/p_limit
```

For cgroups version 2, use instead this command:

```
sudo mkdir /sys/fs/cgroup/user.slice/user-$UID.slice/p_limit
```

For simplicity, in this exercise we will create a symbolic link to this cgroup in order to reduce the length of the commands:

```
ln -s /sys/fs/cgroup/pids/user.slice/user-1000.slice/p_limit ~/p_limit
```

or, for cgroups version 2,

```
ln -s /sys/fs/cgroup/user.slice/user-1000.slice/p_limit ~/p_limit
```

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 ~/p_limit/pids.max
```

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

```
sudo bash -c "echo 10 > $HOME/p_limit/pids.max"
```

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

Now, to facilitate the inclusion of processes in the `p_limit cgroup`, change protection of the file that defines which processes belong to the `cgroup` (`cgroup.procs`). Assuming that `$USER` gives your user name (usually does), change the owner of the `cgroup cgroup.procs` file to be yourself:

```
sudo chown $USER ~/p_limit/cgroup.procs
```

Now, consider the following program (`cgroup.c`) that launches an arbitrary command within a given set of `cgroups`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <string.h>
#include <wait.h>

char * pid2str( pid_t pid )
{
    int len = 2;
    char * str;
    pid_t scout = pid;

    while (scout /= 10) len++;

    str = malloc( len );
    snprintf( str, len, "%u", pid );

    return str;
}

void usage( char * command )
{
    fprintf( stderr, "Usage: %s cgroup [cgroup list] -c command [command args]\n", command );
    exit( 1 );
}

int main( int argc, char ** argv )
{
    char ** command = 0;
    pid_t pid;
    char * pid_str;
    int pipe_fds[2];

    for (int i = 1; i < argc; i++) {
        if (strcmp( argv[i], "-c" ) == 0) {
            command = argv + i + 1;
            argv[i] = 0;
        }
    }
```

```
if (command == 0 || command == argv + 2) {
    fprintf( stderr, "No cgroups where provided\n");
    usage( argv[0] );
}

for (int i = 1; argv[i] != 0; i++) {
    char * cgroup_pids = malloc( strlen( argv[i] ) + 14 ); // "/cgroup.procs"
    sprintf( cgroup_pids, "%s/cgroup.procs", argv[i] );

    if (access( cgroup_pids, W_OK) == -1) {
        switch(errno) {
        case ENOENT:
            fprintf( stderr, "cgroup %s not found\n\t(looking for file %s)\n", argv[i],
                cgroup_pids );
            exit( 1 );
        case EACCES:
            fprintf( stderr, "No permission to add PID to cgroup %s\n\t(looking for file
                %s)\n", argv[i], cgroup_pids );
            exit( 1 );
        default:
            fprintf( stderr, "cgroup %s access error %d\n\t(looking for file %s)\n",
                argv[i], errno, cgroup_pids );
            exit( 1 );
        }
    }

    argv[i] = cgroup_pids;
}

printf( "Execute the command %s with these cgroups:\n", *command );
for (int i = 1; argv[i] != 0; i++) {
    printf( "\t%s\n", argv[i] );
}

pipe( pipe_fds ); // To sync parent and child

pid = fork();

if (pid == -1) { // error
    fprintf( stderr, "Could not fork, errno = %d\n", errno );
    exit( 2 );
}

if (pid == 0) { // child
    char c;
    read( pipe_fds[0], &c, 1 ); // indication to proceed from parent
    close( pipe_fds[0] );
    close( pipe_fds[1] );

    execv( command[0], command );

    fprintf( stderr, "Could not exec, errno = %d\n", errno );
    exit( 3 );
}

pid_str = pid2str( pid );

for (int i = 1; argv[i] != 0; i++) {
    int fd = open( argv[i], O_WRONLY );

    if (fd == -1) {
        fprintf( stderr, "Cannot open to write cgroup file %s (errno = %d)\n", argv[i],
            errno );
        exit( 1 );
    }

    if (write( fd, pid_str, strlen( pid_str ) ) == -1) {
        fprintf( stderr, "Child process coulnt not be included in cgroup\n" );
        kill( pid, SIGKILL );
    }
    close( fd );
}

write( pipe_fds[1], pid_str, 1 ); // Send indication to child to proceed
wait( 0 );

return 0;
```

```
}
```

Compile this program and use it to launch the fork bomb with it:

```
./cgroup ~/p_limit -c fork-bomb
```

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

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

```
cat ~/p_limit/cgroup.procs
```

Repeat this command until you see that all processes 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 frst 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 ~/p_limit/cgroup.procs‘/cgroup
```

Now run the fork bomb without the `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 $$ > ~/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 /sys/fs/cgroup/pids/user.slice/user-1000.slice/p_limit
```

or, for `cgroups` version 2,

```
sudo rmdir /sys/fs/cgroup/user.slice/user-1000.slice/p_limit
```

and you can also remove the symbolic link used:

```
rm ~/p_limit
```

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

5 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 4 KiB chunks.