Distributed Project

Abdelsalam ElTamawy 900170376

December 7, 2020

1 Usage

1.1 Initial Setup

This is meant to run on Linux based systems. Tested on Arch and Debian. The recommended setup is to create 2 Debian virtual machines.

The only form of setup required for this program to work is to setup an FTP server since this is what is being relied on to transfer files from one end point to another.

Simply download your preferred FTP server implementation, such as `vsftpd`. Then configure it so it allows anonymous access, in particular anonymous uploads so we do not need to worry about permissions later on. To enable anonymous access, open/create `/etc/vsftpd.conf` (Assuming you're using the vsftpd implementation). Then set these setting in the file:

```
anonymous_enable=YES
anon_upload_enable=YES
anon_mkdir_write_enable=YES
write_enable=YES
local_enable=YES
```

Then, simply run the ftp server on Linux with `systemd`.

1.2 Using the client and daemon

Then to actually run our fork. On both nodes, start the daemon with root permissions (again, so we don't have to worry about network permissions or anything similar), also to make sure `CRIU` has the permissions it might need. The client will create a simple text file in the home directory. To specify where the client file is to fork to, simply add the IP address as an argument. For instance, you should invoke the client executable as follows:

```
./client 192.168.1.5
```

Assuming the IP of the node to fork at is `192.168.1.5`.

1.3 Build from source

To build the project, navigate to the directory `./src/build_debug` and run `cmake --build ..`

1.4 Using the library

To extend this functionality to any compatible program, all that needs to be done is include the tiny `lib.cpp` in a project. It will give access to the minimal invocation of myfork so that the daemon can take over.

2 Initial attempts

For me at least, 90% of the project was researching the calls and technologies used behind processes management in Linux and how to go about using asynchronous networking libraries.

At first, I tried to create the process checkpoint and restore myself through lower level system calls. What I tried to do was to first pause the client process calling the fork and then the daemon attaching to it using the Linux system call `ptrace`, using it to capture the client's register state and information about it's execution such as active file descriptors or resources used.

We the, inspect the client's memory using the `/proc` directory. From the `/proc` directory we are to read where all the dynamic and static memory allocations have taken place. For each of these entires, we save the corresponding memory segments into an array.

Now we would have captured both the memory state and the register state. We then would bundle them up along with the client executable, also found in `proc`, into a tar ball and send them to the remote process. The remote daemon would intercept it and execute the executable which is immediately paused. Then, the executable would have its memory and register states replaces with what was sent through the tar ball, thus restoring the process to where it was at when the fork was called from the original client process.

However, all this was not used since we can use the `CRIU` library which stream lines the process.

3 How it works

The daemon starts 2 threads, each being responsible for a TCP listener. One listener to receive requests from the local process seeking to fork and another listener waiting for a request over the LAN to restore the process and complete the fork.

Each network call is made to be blocking to induce a form of thread safety so we do not attempt to do multiple restore and checkpoints at the same time, perhaps interfering with each other.

This allows to develop a sort of natural queue as requests build up at the socket and the daemon would go through them sequentially.

Upon receiving a fork request, the daemon anonymously transfers the image files created by `CRIU` to the `/tmp` directory of the remote machine from its own `/tmp` directory through the FTP server and protocol.

The local daemon then sends a message to notify the remote daemon that the file are ready and the process id to refer to. Now that the remote daemon is aware of the process `CRIU` image in its own file system. It would then access them and pass them to the `CRIU` API to restore them as a process.

4 Libraries used

Initially, I attempted to use the famous `asio` library for networking but it proved to be much too low level for out purposes. I went through many libraries to settle on something suitable.

I tried `gRPC`, `ZMQ`, `libcurl` and `QtNetwork`. Each spending days studying them and evaluating them.

What I ended up settling on is `SFML` and it's networking library which proved to be quite straight forward.

Also `CRIU` was used as a checkpoint/restore mechanism.

5 Limitations

It does not carry over environment variables. Nor can it carry over file descriptors. In essence, it is primarily there to carry over the code. Also, it requires for the both end points to use the same endian convention. Of course, the future system calls from the parent to the child will also be limited to what we implement that would relay to the remote end point.