Excercise 4 - syscalls libraries

In this experiment I used c language for my experiments, in your experiment I used 6 different system calls, the purpose of which is to create a background daemon, the whole process is done on the docker container.

First you can run the code with the following command.

```
cd ex4
docker build -t lab1_ex4 .
docker run --rm -ti -v $(pwd)/:/root/lab lab1_ex4 /bin/bash -c "cd /root/lab; gcc -o ex4 ex4.c;    ./ex4 ;
ps ajx "
```

After running the above code you should see the details of the processes running on the system, and you should see a file named. /ex4, which is exactly what I wanted to create.

In my experiment I used 7 different system calls, they are fork(), setsid(), chdir(), umask(), close(), open(), dup().

The daemon is always running in the background and is present when the system starts and is not associated with any terminal. But in essence daemons are also processes, and here I will implement the creation of daemons by modifying existing processes. Processes have some hierarchical relationships, such as sessions and process groups. Therefore, a process will have the following IDs, PID, PGID and SID. For parent-child processes, the new process inherits the PGID and SID of the parent process, but by calling the ps axjf command you can see that there are many different sessions and many different process groups under the sessions. This is because the system has the ability to change and set the PGID and SID, otherwise a system would only exist in one session and process, and there would be no background processes anymore, which would not facilitate the shell job management.

So, if necessary, when we need to create a daemon, we need to use the fork function to detach the child process and exit the parent process, so that the current child process is not controlled by the parent process and starts the first step of independence. This is the reason I use fork() firstly.

The parent and child processes are created and the exit() function is used on the parent process, at which point the child process is released from the control of the parent process.

Since the fork system call creates a child process, the child process has the session duration, process group and control terminal of the parent process, etc. Although the parent process exits, the child process is not completely independent, as the session duration and terminal have not changed. In order to make the child process fully independent and become a new process, a new session needs to be created, which is where the setsid() system call is used.

Then call chdir() to change the working directory of the current process, which I set to /home, The main purpose of this step is to avoid the trouble of having the current process in the same directory as the parent previous process.

Similarly, the child process inherits the file permissions of the parent process, and again to avoid trouble, I reset the permissions of that process by using umask().

For background processes there is no need for terminal input nor error reporting and output, so here I will deal with standard input, standard output, and standard errors. The first argument in dup2() means that the file descriptor of the first argument is copied to the file descriptor of the second argument, which is obtained with the open() system call, RDWR means readable and writable, and the path is ‘dev/null’ . This is a very special and commonly used path in Linux to handle unnecessary data.

Finally use while(1) to keep the process running in the background.

After executing the program chase you can see via ps ajx that the process is running in the background, indicating successful creation.