Operating Systems, Lab session 2

Robin Sommer (s2997592) Suzanne Maquelin (s2696320)

March 9, 2018

Shell

We improved our shell of the first lab by using an execv call instead of execvp. We added the following functionality to the shell: I/O redirection and background processes.
The main flow of the program is as follows:
First we parse the input from our shell. That input is passed to a distribution function which checks whether the process should be executed in the background and after that takes care of forking a child and letting the child(ren) execute the command(s).
The I/O redirection is handled within the child. We respectively close stdout and stdin and open the files they need to be redirected to.
The background process is handled within the parent process as following. If it is specified that it should be executed in the background, we will close the stdout and set the output to ”/dev/null”. Furthermore we do not wait for the child processes to exit, but just continue. The parent has a signal handler for the signal SIGCHLD, which is send upon termination of the child

We did not implement pipes because we did not have enough time, however we would do it in the following way: the parent process checks the parsed command for n pipes and forks n+1 children to distribute the programs over. We would make n pipes, such that the children can pass their output to their neighbours.

Simulating shared memory

WE SINCERELY SUGGEST YOU DO NOT RUN OUR PROGRAM FOR THIS PROBLEM.

We were not able to solve the problem as the question posed.
We ran into different problems trying to solve the problem in multiple ways.
In our first solution we changed the rights of the ”shared” memory of our children once. Then we let them communicate via pipes over the updated values of sharedTurnVariable. A consequence was that it was not really virtual shared memory, because the rights of the supposedly shared memory where changed only once. This meant both children had the access to the memory at the same time, which shouldn't happen.
We then implemented a mprotect statement modifying the rights, such that the children couldn't access the memory anymore after they did their block of code using the sharedTurnVariable.
This solved the problem of having continuous rights to access the memory, but still had the problem that the handler did not know who was supposed to have the rights, so it would always grant access to the memory.

Then we thought of letting the parent process keep hold of to who has the rights of writing. This meant we needed two different handlers, one for both the children and one for the parent.
For the handler of the children to know the access rights, they needed to contact the handler of the parent. We implemented this using SIGUSR1 and SIGUSR2. When the parent receives a signal it checks whether the child belonging to the signal should be granted access. It checks this by looking at its own turnVariable and the incoming signal.
When the child should get access, the parent sends a signal to the child and writes the current state of the memory to the pipe. The child receives this in its handler. Here the mprotect is called to set the rights to

writing. And the ”shared” memory (turnVariable) is updated by reading from the pipe.
When the child executed its turn in procPingPong, it writes the new status to the pipe and it notifies the parent, via signal SIGRTMIN, that the memory has been updated. Then the memory of the child process is set to protected again.

The problems we ran in to implementing this solution were mainly, not being sure the signals were used correctly in communication between parent and children, and not being able to access certain variables in updating the supposedly shared memory.