Lab 1: Developing a simple shell

Specifications

Specification 1

Execution of simple commands is done by using fork and the child runs execvp which runs the executable command from the PATH (environment variable). The return value of execvp indicates if there is a failure in execution and illegal commands are handled this way. The parent process waits for the child process to complete when it's not an background process.

```
> date
Mon 26 Sep 2022 09:09:47 AM CEST
> hello
Could not execute command: No such file or directory
> ls -al -p
total 109
drwxrwxrwx 2 shivel shivel    10 Sep 26 09:08 ./
drwxrwxrwx 5 shivel shivel     8 Sep 26 09:08 ../
-rwxrwxrwx 1 shivel shivel  6148 Sep 26 06:40 .DS_Store
-rwxr-x--- 1 shivel shivel 30040 Sep 26 09:08 lsh
-rw-r----- 1 shivel shivel  6298 Sep 26 07:43 lsh.c
-rw-r----- 1 shivel shivel 13720 Sep 26 09:08 lsh.o
-rwxrwxrwx 1 shivel shivel   383 Sep 26 06:40 Makefile
-rwxrwxrwx 1 shivel shivel  3152 Sep 26 06:40 parse.c
-rwxrwxrwx 1 shivel shivel   343 Sep 26 06:40 parse.h
-rw-r----- 1 shivel shivel 12224 Sep 26 09:08 parse.o
```

When programs fail, the stderr is printed on the screen. The child processes completes its execution.

Specification 2

Background processes are implemented by not waiting for the child process to finish as explained above. The parent process becomes ready for the next execution.

```
> sleep 60 &
> sleep 60 &
> sleep 60
> sleep 60
^C>
```

If the process is not running in background there are two ways to generate a prompt. We can kill the child process using the kill command in another terminal. Or we can use Ctrl+C to stop the execution of the child process and the implementation of the same will be explained in specification number 6. Only the foreground process will stop when pressing Ctrl+C and the background process continue to execute. When the background processes complete finishing, they die and there are no zombie processes left over. And the above is the expected behaviour.

Specification 3

Piping is done by first creating a pipe by the parent. The implementation was done without being aware that the list of commands are not stored in the order they are typed, so I've reversed the list before executing the commands one by one by the parent process. The first child will read from stdin and write into the write end of the pipe. The following children will read from the read end of the pipe and write to the write end of the pipe except the last child which will write to stdout. Note: stdin and stdout might change due to i/o redirection.

```
> ls -al | wc -w
110
> ls -al | wc
   13    110    638
> ls | grep lsh | sort -r
lsh.o
lsh.c
lsh
```

Yes the prompt returns after the above command.

```
> ls | wc &
>      9      9     61
```

The prompt appeared immediately after the pressing enter on the command since it's running as background processes but it finished execution quickly. I'm still able to type the next command and get the expected output.

```
> cat < tmp.1 | wc > tmp.3
> cat < tmp.1 | wc
   12    101    586
> cat tmp.3
   12    101    586
```

Both the outputs are same since the output of "cat < tmp.1 | wc" is being written to tmp.3 in the first command.

```
> abf | wc
Could not execute command: No such file or directory
     0      0      0
> ls | abf
Could not execute command: No such file or directory
> grep apa | ls
lsh  lsh.c  lsh.o  Makefile  parse.c  parse.h  parse.o  tmp.1  tmp.2  tmp.3
>
```

The prompts appear immediately for all commands except the one which has grep. That's because grep is waiting for data in stdin since not file has been given. Grep does not terminate by itself but Ctrl+D helps terminate it.

Specification 4

Input/Output redirection is done by creating file descriptors for them and the child process duplicates the read file descriptor to stdin and the write file descriptor to stdout. The opening of file descriptor is done in such a way that it can also create a new output file if needed.

```
> ls -al > tmp.1
> cat < tmp.1 > tmp.2
> diff tmp.1 tmp.2
```

No, the output is as expected since both files contain the same data and there is no difference.

Specification 5

When calling exit, exit(0) is called to end the process. When using cd, chdir method is used.

```
> cd ..
> cd lab1
cd failed due to: No such file or directory
> cd lab1-2022
> cd tmp.tmp
cd failed due to: No such file or directory
```

When specifying a wrong path and error will be thrown like above.

```
> cd ..
> cd lab1 | abf
cd failed due to: No such file or directory
Could not execute command: No such file or directory
> cd lab1-2022 | abf
Could not execute command: No such file or directory
> ls
code  docs  lab1-shivel  lab1-shivel.zip  prepare-submission.sh
```

Yes, the ls command worked as shown above.

```
> cd
> pwd
/chalmers/users/shivel
```

There is no error and it changes directory to HOME path.

```
> grep cd | wc
      0       0       0
```

Output only appears after Ctrl+D and does not appear before.

```
> grep exit < tmp.1
```

No, the shell did not quit and execute the grep properly by searching for exit as a string in tmp.1 file.

```
> ..exit
Could not execute command: No such file or directory
```

When using dots, it didn't exit but when using spaces it exited.

```
> grep exit | hej
Could not execute command: No such file or directory
```

There was an error here because hej is an illegal command and prompt was stuck waiting for stdin by grep. After Ctrl+D, it printed the error.

```
> exit
Goodbye
```

There no zombies left after exiting.

Specification 6
For Ctrl+C to terminate the foreground process and not the shell, a signal handler was introduced. A global variable was introduced to set the child process id when it gets created and when Ctrl+C is entered, the handler kills the child using the kill method and the global variable which has the process id. This way the handler only stops the child process and not the shell itself.

Specification 7
Ctrl+C will ignore background processes since the child process is being told to ignore SIGINT if the child is background process. This way Ctrl+C will not have an effect on it.

One issue that I faced when I'd used wait method instead waitpid (waiting for specific child) is that if there were other children running, wait will not stop execution until one child finished. By changing to waitpid, it was possible to wait for the specific child process and ignore about the other background process that might be running.