Lab 1 - Group 58

William Leven (levenw@student.chalmers.se)
Vidar Magnusson (mvidar@student.chalmers.se)

1 Implementation of specifications

1.1 Simple commands

Simple commands are handled with the *execvp()* system call. The *v* is used to pass additional command line arguments to the program and *p* is used search the *$PATH* variable. The call to *execvp()* replaces the current process and therefore we make sure to *fork* before and only run *execvp()* in the child process. The only case that *execvp()* should return anything is when it has encountered an error, for that reason we have an *exit()* call immediately following the *execvp()* to make sure that the child process is cleaned up in such an event.

Problems encountered:
At first it took us some time to realize that execvp overwrote the current program (unless any errors occured) but after we figured that out we had no further issues with it.

1.2 Background commands

The implementation of background commands in the code is accomplished mostly by not waiting for child processes to complete their execution after all processes have been spawned. Furthermore, for background processes we simply ignore any *SIGINT* signals rather than going through and killing all the processes upon receiving such a signal.

Problems encountered:
We first had problems with zombie processes before we realized that we had to listed til child signals.

1.3 Pipes

Piping is handled initially by looping through the given linked list of commands in the given order (i.e. from right to left) and caching the file descriptor from the input of the current command to output of the next command (i.e. the command to the left of the current command). An exception to this is made for the last command (i.e. the leftmost command) which is either handled by the redirects, mentioned below, or comes from standard in.

Problems encountered:
At first we had a really hard time keeping track of the file descriptors in different processes as all had to be closed. After some major refactoring and manual debugging this became less of an issue.

1.4 Redirects

If the command has any redirects we make sure to open the applicable files before beginning the execution of any programs. After opening the files the returned file descriptors can be used in a similar way as the file descriptor for any potential pipes but for the first as well as the last command.

To redirect the out we use *creat()* which is identical to *open()* with the following flags *O_CREAT, O_WRONLY, O_TRUNC* which says that we should create the file if it doesn't exist, the file should be write only, and that we should truncate the file when writing to it. Furthermore we set permissions of the file to write: user and group, read: all.

To redirect the in we use *open()* with the read only setting.

Problems encountered:
We had some problems keeping track of all file descriptors in different processes but it was solved when we combined the logic with the one for pipes. We also had some problems with the permissions of the created file before we realized that we could set the permissions ourselves.

1.5 Built-in functions

Built-in functions are handled by checking if the current command is one of the two keywords for these and then handling them separately from the normal command handling code and does therefore not spawn any child processes. *exit* simply calls the function *exit()* to exit the program with status code 0 and *cd* uses *chdir()* to change the directory to the given argument.

Problems encountered:
The implementation of *exit* was fairly straight forward and no problems was encountered with this. For *cd* we initially put the code handling it in a child process meaning that the directory was changed for the child process and not for the main shell.

1.6 Signal handling

Initially the application is set to ignore any *SIGINT* or *SIGCHLD* signals in the main function, furthermore, each time a child process is spawned it is immediately also set to ignore any *SIGINT* signals. To have the ability to kill any currently running **foreground** command, during the time that the appliation waits for non-background commands to complete their execution, the application handles *SIGINT* signals differently. Upon recieving a *SIGINT* in this interval the application will go through all the child proccesses (whose PIDs are cached in an array after they are spawned) and kill all of them using the *kill()* function. After all the children are either done or have been killed, the application once more ignores any *SIGINT* signals.

Problems encountered:
For the signal handling we encountered multiple problems, one of them being that we initially didn't set the signal handling for *SIGINT* to *SIGIGN* when we spawned a child process. This led to the program sometimes shutting down upon <ctrl>+<c> when a command was running.

2 Result of self-tests

2.1 Simple commands

```
> date
Thu Sep 17 14:11:16 CEST 2020
>
```

```
> hello
Could not find executable: hello
>
```

Due to the *exit()* call after the execvp command the child process is killed and no zombies remain after the hello command.

2.2 Commands with parameters

```
> ls -al -p
total 149
drwxr-xr-x  5 levenw levenw    18 Sep 17 14:09 ./
drwx--x--x 33 levenw levenw    44 Sep 15 15:08 ../
drwxr-xr-x  8 levenw levenw    15 Sep 15 15:07 .git/
>
```

2.3 Redirection with in and out files

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

Seing as the files should be identical and that diff prints any difference between the files, the output is expected to be completely empty, which it was.

2.4 Background Processes

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

After pressing ctrl + c, the foreground process is stopped but the two background processes remain for the 60 seconds and then stops. Furthermore, killing the background processes with top does not make the prompt return, however, killing the foreground process does.

2.5 Process Communication (Pipes)

```
> ls -al | wc -w
164
> ls -al | wc
     19     164     997
> ls | grep lsh | sort -r
lsh.o
lsh.c
lsh
>
```

As can be seen, the prompt appeared again after the commands.

```
> ls | wc &
>      13      13     114

>
```

The prompt reappears before the output of wc, hence the slightly strange output. One can write the next command immediately but in this case we pressed enter once to get the prompt to look correctly first.

```
> cat < tmp.1 | wc > tmp.3
> cat tmp.1 | wc
     19     164     997
> cat tmp.3
     19     164     997
>
```

As can be seen, the outputs are the same. This is because we have run the command *wc* on the contents of tmp.1 and saved it the file tmp.3. Running it again on the same file without any modifications would hence give the same result as the contents of the file tmp.3.

```
> abf | wc
Could not find executable: abf
      0       0       0
> ls | abf
Could not find executable: abf
> grep apa | ls
lab1-levenw      lsh    lsh.o     parse.c  parse.o             tmp.1  tmp.3
lab1-levenw.zip  lsh.c  Makefile  parse.h  prepare-submission  tmp.2
```

The outputs is the output of ls, this is because the commands are run from right to left and ls is therefore the first command to be run. After that the program "hangs" due to grep expecting to get an input (/ waiting for EOF) but never getting it. Grep therefore never terminates unless ctrl+c (or another method of killing the process) is used.

2.6 Built-in Commands

```
> cd ..
> cd lab1
> cd tmp.tmp
No such path
>
```

As can be seen, the error "No such path" was generated while executing the commands.

```
> cd ..
> cd lab1 | abf
> Could not find executable: abf

> ls
lsh    lsh.o     parse.c  parse.o             tmp.1  tmp.3
lsh.c  Makefile  parse.h  prepare-submission  tmp.2
>
```

As can be seen, the command ls worked as expected.

```
> cd
Invalid argument
>
```

We did get the error "Invalid argument", the directory is therefore the same as before the command was run.

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

The shell did not exit, it did considered exit as a string literal to search for in the file.

```
>   exit
[levenw@remote11 repo]$
```

As can be seen, the shell was exited.

```
> grep exit | hej
Could not find executable: hej
```

As can be seen, no errors were generated and no ouput was printed also the prompt reappeared immediately after the command.

```
> grep cd | wc
```

No output appeared, however, after ctrl+D is pressed, the prompt reappears.

```
> exit
[levenw@remote11 repo]$
```

No zombies remain after exiting the shell, any background processes are left to complete their execution and is then terminated by the OS (as in bash).