Operating Systems Lab
Lab-1

Date: 16-05-2021
Group 9
Abhishek Raj (180010002)
Harsh Raj (180010017)
Sri Priya (180010025)

Design Decisions For implementation:

- All the background processes' pid is stored in a global array called `bg_pid`. The global array is handled similar to a stack, where after removing an element, the space is filled by the last element, and the size of the array decreases.
- The `backgroundHandler()` function reaps the process which just terminated. It ensures that the child process' resources are freed once the process completes. It gets called when the shell receives a `SIGCHLD` signal.
- The `killProcesses()` function kills all the child processes. This function is called when the shell is about to exit.
- The `execute()` function, executes a command. Depending on the type of execution (weather foreground, background, parallel, sequential etc) is required, it does other work, like pushing pids to the global arrays, etc.
- The `tokenize()` function takes a string, and breaks it into possible commands. It uses a global array `tokens`.
- We use `signal()` to handle the signals sent to the shell.

```
// for signal SIGINT, IGNORE IT
signal(SIGINT, SIG_IGN);

// for signal SIGCHLD, run the function backgroundHandler,
signal(SIGCHLD, backgroundHandler);
```

Screenshots:

```
harshraj22 in assignment 1 on master [!]
$ gcc my_shell.c

harshraj22 in assignment 1 on master [!]
$ ./a.out
$ sleep 5 & ls & pwd
$ /home/harshraj22/Desktop/lab/pro/Operating-Systems/assignment 1
Shell: Background process finished
a.out  commands.txt  fork.c  my_shell.c  readme.txt  shell.pdf  test.txt
Shell: Background process finished
Shell: Background process finished

$ sleep 5 && ls && pwd
a.out  commands.txt  fork.c  my_shell.c  readme.txt  shell.pdf  test.txt
/home/harshraj22/Desktop/lab/pro/Operating-Systems/assignment 1
$
$ sleep 5 &&& ls &&& pwd
/home/harshraj22/Desktop/lab/pro/Operating-Systems/assignment 1
a.out  commands.txt  fork.c  my_shell.c  readme.txt  shell.pdf  test.txt
$
$ exit
Shell: Goodbye.
```

Analysis:

1. Commands separated by &
   (parallel background execution)

   ```
   $ ./a.out
   $ sleep 5 & ls
   $ a.out  commands.txt  fork.c  my_shell.c  readme.txt  shell.pdf
   Shell: Background process finished
   pwd
   /home/harshraj22/Desktop/lab/pro/Operating-Systems/assignment 1
   $
   $ Shell: Background process finished
   ```

   As we see, the `ls` command gave output before sleep, thus ensuring parallel execution. While the `sleep` command is being executed, we were able to execute the `pwd` command, thus ensuring execution as a background process.

2. Commands separated by &&

(sequential and foreground execution)

```
$ ./a.out
$ ls && sleep 3 && pwd
a.out  commands.txt  fork.c  my_shell.c  readme.txt  shell.pdf  test.txt
/home/harshraj22/Desktop/lab/pro/Operating-Systems/assignment 1
$
```

The commands are executed sequentially, in the same order they are written.
The foreground execution does not let the shell execute any other command till they are executing.

3. Commands separated by &&&
   (foreground parallel execution)

```
$ sleep 5 &&& ls
a.out  commands.txt  fork.c  my_shell.c  readme.txt  shell.pdf  te
pwd
$ /home/harshraj22/Desktop/lab/pro/Operating-Systems/assignment 1
$
```

The foreground execution does not let the shell execute any other command till they are executing.
As we see, the `ls` command gave output before sleep, thus ensuring parallel execution.

4. Signal Handling
   (Ctrl + C)

```
$ ^C
$
$ exit
Shell: Goodbye.
```

The signal SIGINT is handled by the code (in this case, the signal is ignored). The shell terminates if the user enters `exit`.