Calvin Nichols 0588016

Operating Systems Assignment #2: Walking on an Egg-”shell”

Problem:

Create a shell that allows for background processes, control directives, input and output redirection, efficiency enforcement and piping.

Solution:

Parsing:

I check each token returned from strtok for correct command line input. I implemented some error checking to ensure any symbols used are used correctly in the proper order and if not, I return a message specific to the problem. For example, the line “prog1 ! ^ infile” will return the statement “error: argument after ‘!’.” All the information found from the command line is saved in a node of type struct command of an array of linked list, one node per program, one list per command line.

Background Processes:

The flag “background” is set on the command node if “!” is found as the last token. During execution the flag will dictate the parent program using wait or not after the fork.

I/O Redirection:

Flag “infile” or “outfile” is set on the command node. During execution the function “io_redirection” will check these flags and alter the file descriptors accordingly.

Piping:

Piping flag is set on each program to be piped. The function “piping” which sets up the pipes in a loop and forks each child is called from the function “exec_cmdline”.

Efficiency Enforcement:

The percentage the cpu is required to be running at is saved to a command node. At execution an alarm is set for 5 seconds to allow the program to ramp up and then popen is used to check the output of ps. If the process meets the threshold alarm is reset to run again in 2 seconds else, SIGINT is sent to the process and killed.

Control Directives:

The control group is set in each node of the list. After the fork the parent saves the returned child pid to its corresponding node for later execution of jkill, jstop or jcont. When one of these commands are entered into the command line the first node of each list is searched for a corresponding job group, if found, sigstop, sigcont or sigkill is sent to each process.

Decisions/Assumptions/Justifications:

In order to keep track of all the commands being processed I used an array of linked lists and a struct containing all the variables I would need to execute each process correctly. This allows me to make a node for each program that needs to be run and a linked list is used for each command entered into the shell, except for control directives. This made it very easy for me to keep everything organized.

Calvin Nichols 0588016

My parsing method is fairly versatile but it is far from perfect. When a user inputs something that is an obvious mistake it should be caught but anything beyond that I let slide as this assignment is focused on the meat of the shell.

Limitations and Bugs:

My array of linked list is obviously unsustainable for a real shell and would ideally be replaced with a dynamic array or a 2d linked list but I ran out of time to implement this and it wasn't a priority concerning marks. Also, freeing of lists that have completed would need to be implemented.

When I catch mistakes at the command line made by the user I call exit. To be more in line with the real purpose of a shell I should be prompting the user to input the line again.

After a command setting a job group and efficiency and then stopping and continuing that job group the efficiency check will not trigger after that. Example:

```
"[1] prog1 > 90 !"
"jstop 1"
"jcont 1"
```

At this point the program will not be checked against its efficiency threshold.

When the prompt returns to the user, in certain situations the "My Shell > " will not print, but the shell will still be working correctly.

I would have also liked to implement defined debug statements and assert statements.

Testing:

I made various programs that allowed me to test all the components of this assignment. I am confident all requirements will work independently as specified by the assignment on the lab computers in 001A of Reynolds.

More specifically, I tested efficiency by running a program which contained an infinite loop doing some simple math and testing various efficiency requirements, some that should pass and some that should not. I used ps to check that my program was correct. All tests appeared to be working properly.

Job control is always tested with background processes on. I tested this by running a program that looped to 100 and printed out each increment. I ran 3 of these with the following commands:

```
[1] ./prog !
[2] ./prog !
[1] ./prog !
```

I then jstop 1 and jcont 1 which stopped and continued the group 1 processes properly.

For IO redirection I ran the following commands:

```
grep file1 ^ file2 $ out
ls $ out
cat file.txt -> uniq -c -> sort $ out
ls -> tr e f $ out
```

They all appeared to be working correctly.

Piping I ran commands such as:

```
cat file.txt -> uniq -c -> sort
ls -> tr e f
ls -> wc
```

Calvin Nichols 0588016

Everything ran as intended. Unfortunately, I do not know very many programs I can pipe together.

Bonus:
My shell supports:

- Multiple pipes, tested with:
  - cat file.txt -> uniq -c -> sort
  - cat file.txt -> uniq -c -> tr e f -> sort
- Multiple pipes with IO redirection, tested with:
  - cat file.txt -> uniq -c -> tr e f -> sort $ out
  - tr e f ^ scores -> wc $ out (note: input file must come immediately after the first program and its arguments, shown above and in readme)
- Job groups, background and efficiency, tested with:
  - [2] prog > 90 !
- Job groups, multiple pipes, IO redirection and background processes, tested with:
  - [2] tr e f ^ infile -> uniq -c -> tr z x -> sort $ outfile !

I do not support:

- Efficiency for job groups or multiple pipes, I decided not to implement this as it did not make sense to me.
- job group commands for a process not in the background.

References:
background process
http://stackoverflow.com/questions/1618756/how-do-i-exec-a-process-in-the-background-in-c
reapChildren.c seminar 3 scott dougan

shell input
http://linuxgazette.net/111/ramankutty.html

parsing
CIS2500, lab4 by Calvin Nichols (me)

piping
parentPipe.c & childPipe.c seminar 3 scott dougan
http://www.dgp.toronto.edu/~ajr/209/notes/procfiles/pipe-example.c

efficiency waiting
http://cboard.cprogramming.com/c-programming/138057-waitpid-non-blocking-fork.html

multiple pipes
http://stackoverflow.com/questions/8300301/chained-pipes-in-linux-using-pipe

linked list
http://www.c.happycodings.com/Data_Structures/code5.html