CSS430  
Martin Metke  
2017/06/25

Program 1 Report

# Part 1

Part 1 involved creating a file that used system calls in C++ to replicate the functioning of a Linux shell running the command line “ps -A | grep <some search string> | wc -l".

## Algorithm

Initially I considered writing an extensible, while-loop-based program that could be passed any arbitrary string of the form “<command1>[ <arg1>] … [<argN>] [| <command2>[ <arg1>] … [ <argN>][ | …]]” with any number of commands and arguments connected by pipe (“|”) symbols – a generalized solution applied to this specific problem – but I decided in the end to take a naïve approach in the interest of reducing extra testing and the additional burden of writing long REGEXes. If I had taken this route, the main() function would have contained one while loop that stripped commands and pipe symbols from a pre-generated “command line” and decided when and how to set up pipes based on the current process’s position (or PID) and how many arguments remained:  
 Parent: no pipes, just wait()  
 Child1: Pipe for child, no pipe to parent  
 Child2: Pipe to child, pipe from parent  
 …  
 ChildN-1: Pipe to child, pipe from parent  
 ChildN: No pipe to child, pipe from parent.  
This could be accomplished with the c++ regex library and regex groups, but again, a large amount of extra work and testing would be involved to make sure that this generalized solution operated correctly.

In my actual implementation, the **processes** command is passed one and only one argument. This can either be a Linux-style help flag (“-h” or “--help") or a no-spaces search string. Any other number of arguments will result in a failure message and failure return value.

Internally the main() function does some basic argument validation but is essentially a chain of “if/else if/else” statements, as these allowed a fairly simple and logical division of the main function into great-grand-child, grand-child, child, and parent sections.   
 Parent: no pipes, just wait()  
 Child1: Pipe to child, no pipe to parent (closes stdin and dup2’s to FD[RD] pipe end)  
 Child2: Pipe to child, pipe to parent (closes stdin, stdout; uses FD[WR] and FDGC[RD])  
 Child3: No pipe to child, pipe to parent (closes stdout, dup2’s to FDGC[WR])

If the shell command to be emulated had included even one more pipe symbol, I would have gone with the above generalized solution instead, however.

## Results

Output comparison of cmd “processes <string>” vs “ps -A | grep <string> | wc -l" on UWB Linux Lab machines:  
  
sleet01@uw1-320-15:~/git/CSS430/program1$ ./processes tty

2

sleet01@uw1-320-15:~/git/CSS430/program1$ ps -A | grep tty | wc -l

2

sleet01@uw1-320-15:~/git/CSS430/program1$ ./processes Sys

0

sleet01@uw1-320-15:~/git/CSS430/program1$ ps -A | grep Sys | wc -l

0

sleet01@uw1-320-15:~/git/CSS430/program1$ ./processes user

0

sleet01@uw1-320-15:~/git/CSS430/program1$ ps -A | grep user | wc -l

0

# Part 2

Part 2 implements a minimal command-running shell within ThreadOS.

## Algorithm

The main while loop executes continuously, accepting user input and parsing each line of user input into commands to be run:

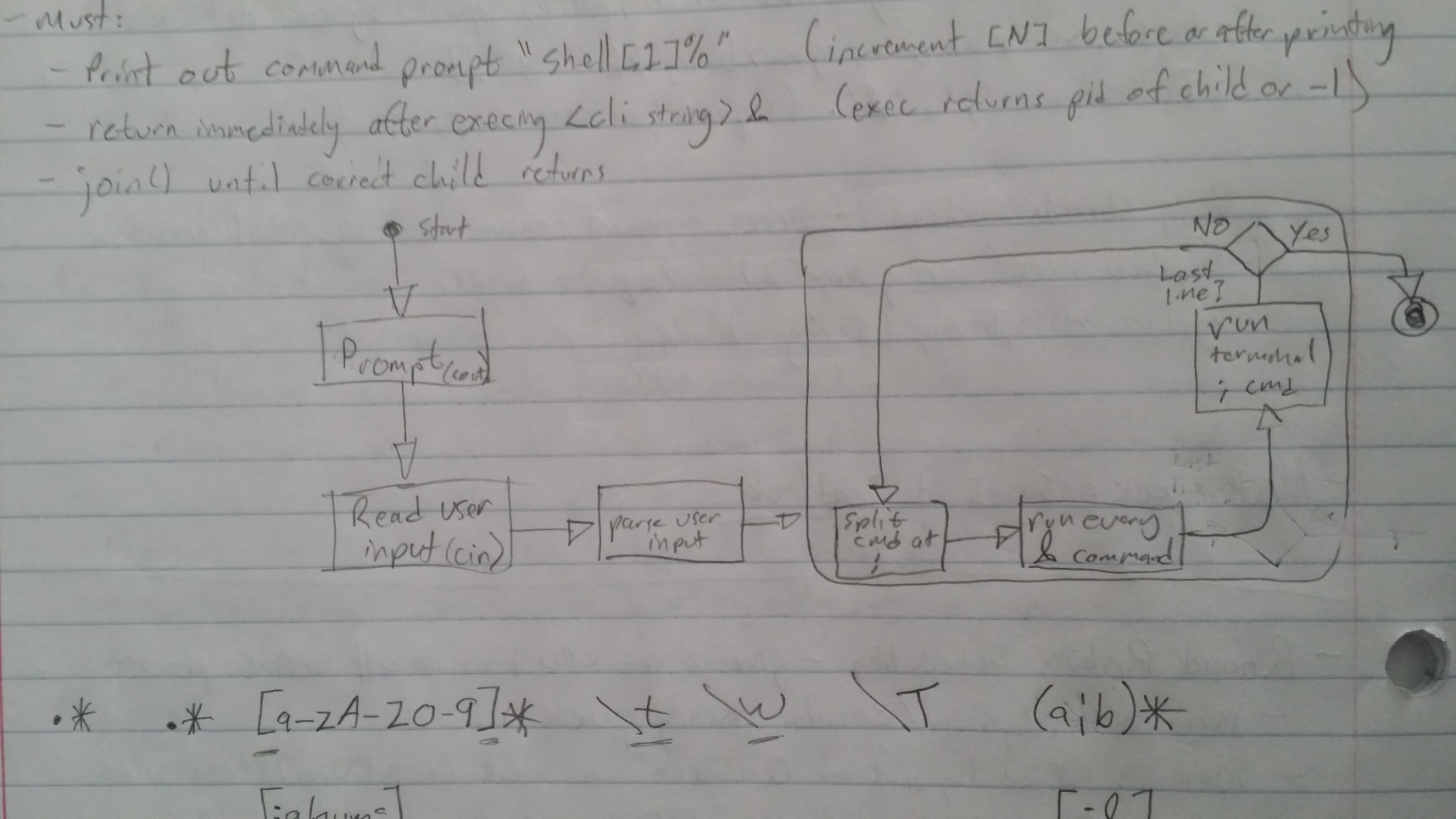
* The command line is broken down into sections ending with ‘;’ characters ( each command ending with a ‘;’ requires the shell to wait for its execution to complete ).
* Each sub-section is further decomposed into commands ending with the “&” delimiter (which can be executed and returned from immediately) and the lone “;”-delimited command (which the Shell thread must wait on).

This approach allows for a simplified loop which only needs to wait for one thread per sub-section of the command line input; each sub-section contains N (where N is 0 to infinity) commands to execute concurrently and zero or one command to wait on.

The Shell process only provides minimal input validation; each valid command is responsible for parsing and validating its own arguments (a la “bash” or other linux shells).

The main while loop will only exit if the user enters the “exit” command in a command line input string; this command may be executed as part of a set of delimited commands but commands following “exit” will not be executed.

## Algorithm (Diagram)



## Testing

In addition to running the suggested base and testing command lines, I input a number of degenerate commands:  
- all spaces - all delimiters - mangled commands  
- incorrect numbers of arguments - mixes of sequential and concurrent commands  
- concurrent commands with varying delays and one sequential command, to prove that join() was correctly waiting for the sequential thread.

## Sample Output

threadOS ver 1.0:  
Type ? for help  
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)  
-->l Shell  
l Shell  
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)

New Shell Created! Type 'exit' to exit Shell  
Shell[1]% PingPong 123 200 & PingPong ABC 400 & PingPong XYZ789 800 ;  
threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)  
threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=1)  
threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=1)

123 123 123 123 123 ABC 123 123 ABC 123 123 ABC 123 XYZ789 123 ABC 123 123 ABC 123 XYZ789 123 ABC 123 123 ABC 123 XYZ789 123 ABC 123 123 ABC 123 XYZ789 123 ABC 123 123 ABC 123 XYZ789 123 <snip>

Shell[2]%