ECS150
WQ 2016
Programming Assignment Number 1
Unix System Calls
Due: Friday, January 29, 2016, 10 AM, via SmartSite
Submission and grading instructions to be posted but assume we will do interactive grading on Friday, Jan 29, starting at 10 AM

As I have indicated, you are permitted to work in groups up to three students.

Your programs are to be written in C and use system calls. Unix commands, such as `grep`, are not permitted.

You can use any UNIX systems, but the programs you submit should also work on MINIX 3.

Think hard about how you want to demonstrate your programs so you convince us graders that you have a correct solution. You will have approximately 3 minutes per problem to convince us. You will be doing lots of demonstrations in your careers, so try to make the demonstrations interesting. Also, be sure to be prepared to show us your programs in addition to demonstrating them or allow us to suggest input that might cause your programs to fail because you did not perform error checking.

Problem 1: Write a program `pipe_test` which creates three processes, P1, P2, P3 and pipes as needed. The first process displays its pid, prompts the user for a string and then sends it to the second process. The second process displays its pid, displays the string it received, reverses the string, displays the reversed string, and sends it to the third process. The third process displays its pid, displays the string it received, converts the string to uppercase, displays the string the string it generated, and sends it back to the first process. When the first process gets the processed string, it displays its pid and displays the string it received it to the terminal. When all this is done, all processes terminate but display their pid and a message announcing their imminent death.

Here is an example of the program in action:

```
%pipe_test

I am process P1
My pid is p
Please enter a string: help

I am process P2
I just received the string help
I am sending pleh to P3
```

```
I am process P3
…

I am process x
My pid is p
I am about to die

…
```

Problem 2. Write a program `AlrmTest` that does the following. Creates two child processes that each increments a process-local variable `count` in an infinite loop. The parent is just a `for` loop of 5 iterations, where in each interation it sleeps for a second and then sends the `SIGALRM` signal to each child. Upon receiving this signal, each child is to print out the value of `count` along with its process id and reset the count to zero. Your output might look like:

```
Count is 11161817, my pid is 472
Count is 10956178, my …
Count is …
Count is …
Count is ……
```

Be prepared to justify the values returned.

Problem 3: Write a program `directory_traverse_breadth_first` that takes as argument a path to a directory searching for a file name. To convince us that your program is working as intended, print out the directory and file names it encounters in the search. For example, the nodes explored could be printed out as follows:

```
/
/dirC
/dirA
/dirC/my3.dat
/dirA/dirB
/dirA/my1.dat
/dirA/my2.dat
/dirA/dirB/my1.dat
```

Be sure to let us know if you cannot figure out the directory tree we have in mind.

Hint: breadth-first can be implemented with a queue. As the program encounters each directory node a particular level, it enqueues the complete pathname for later examination.

Problem 4: Write a program `process_tree` that takes a single integer argument `n` from the command line and creates a binary tree of processes of depth `n`. When the tree is created, each process, each process should display the phrase

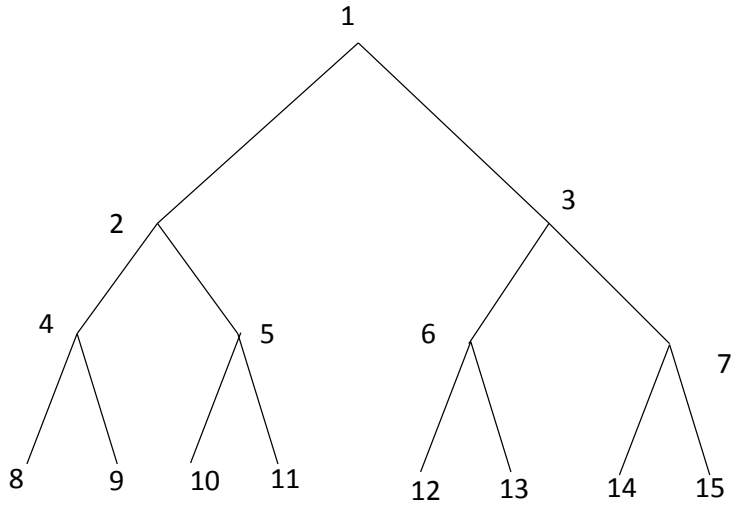
`I am process x; my process pid is p` (the pid for process x);
`My parents pid is p1` (the pid for this process' parent)

After printing as indicated above, the process terminates.

For example, if the user enters

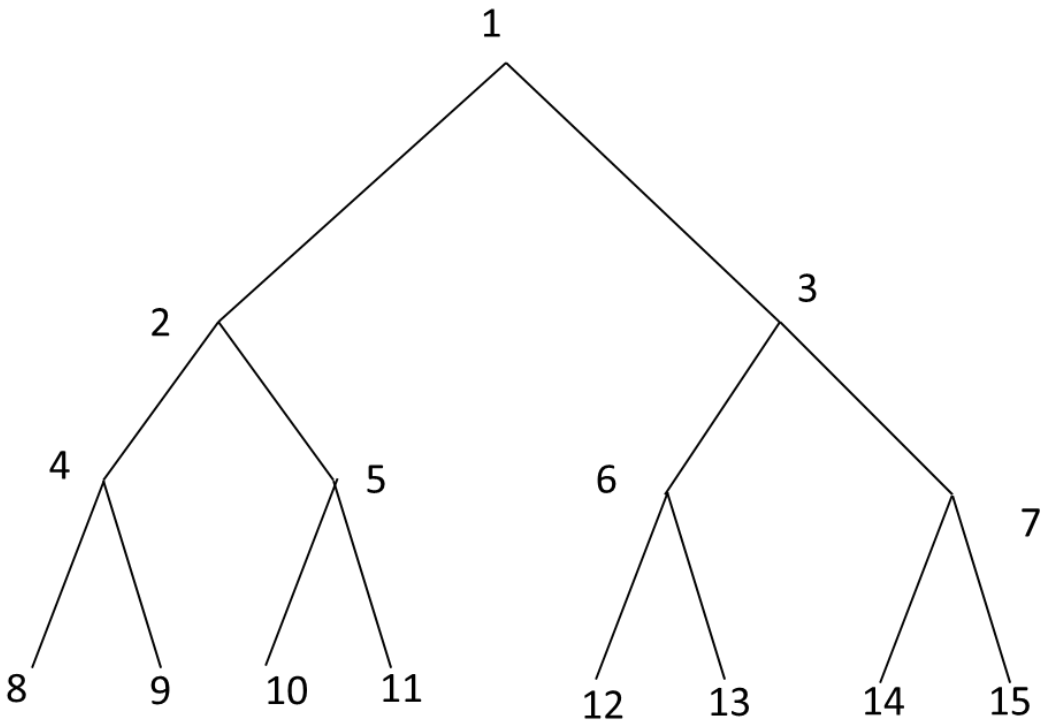
```
%tree 4
```

the process tree would correspond to the following:



Make sure the original process does not terminate until all of its children have died. This will permit you to terminate the parent and its children from the terminal.

Problem 5: Extend `smallsh`, so that is handles signals to kill processes. Make sure background processes and the shell itself are not killed by either `SIGINT` or `SIGQUIT`. Come with a creative demonstration. You might write a program that is an infinite loop and run if first as a foreground process and then in the background. You should be able to kill the foreground process with a signal but not the background one. To make the demonstration more interesting, you could have the action of `SIGINT` give the option to the user to change his/her mind about killing the foreground process or confirming that the process is to be killed.


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15