Lab

Forking

The purpose of this lab is to get continue getting acquainted with `C` by performing some familiar tasks. Recall that you completed a shell lab in CSPB 2400, so this is all review (only now we will be using `C` instead of `C++`). After you complete this lab, you should have a good start on the shell programming assignment.

Warmup: Tokenizing User Input

Write a `C` program that gets a string from the user via `stdin`, then saves each word of the string (separated by white space) into an array. For example, if `"This is a sentence."` was entered as input via `stdin`, then your program should create a new array with elements `"This"`, `"is"`, `"a"`, and `"sentence."`. You should find the functions `getline` and `strtok` useful for this task. Please visit their respective `man` pages for more details. Below is some sample usage of the latter function.

Example: `strtok`

```
#include <stdio.h>
#include <string.h>

int main (int argc, char *argv[]) {
  char str[] = "This is a sentence.";
  char *pch;
  printf ("Splitting string \"%s\" into tokens:\n",str);
  pch = strtok(str," ,.-"); // second arg. is a string of delimiting chars
  while (pch != NULL) // while there are more tokens to read
  {
    printf ("%s\n",pch);
    pch = strtok(NULL, " ,.-");
  }
  return 0;
}
```

Note: for the shell assignment you may not want to implement the array of strings data-structure for your solution; however, this data-structure will be useful for other assignments in this course, and it is something that we should all know how to implement in `C`.

Forking Processes

For our next task, compile and execute the following `C` program.

Example: `fork()`

```
#include <stdio.h>

int main (int argc, char *argv[]) {

 int pid = fork();

 if ( pid == 0 ) {
   printf( "This is being printed from the child process\n" );
 }
 else {
   printf( "This is being printed in the parent process:\n"
     " - the process identifier (pid) of the child is %d\n", pid);
   // we can also get the pid of the parent process using getpid()
   printf( "This is being printed in the parent process:\n"
     " - the process identifier (pid) of the parent is %d\n",(int) getpid());
 }
 return 0;
}
```

After you compile and run this program, note what the PIDs of the parent and child processes are. Now wait a minute or so, then run your program again and note the new PIDs. There should be a gap between these two sets of numbers, explained by the fact that the operating system (and other programs you might be running) are spawning new processes. For a comprehensive and real-time display of the processes running on your system, run the `top` command in your terminal. You can throw a `while(1);` statement in your child and recompile your program if you'd like to see a process you made spin its wheels in `top`. Once you do this, you'll need to kill that process, so you can hit `k` while `top` is spinning, then specify the PID of your child. Type `q` to quit `top`.

Note that it is possible that the parent process can execute before the child or vice versa depending on how the operating system schedules the processes. For instance, when we ran this code, the child code was executed before the parent code the vast majority of the time. This behavior is a consequence of the fact that the parent is not invoking `wait` to wait on the child process to terminate before it executes its code. When we fork processes, we must `wait` on the child processes in order to know the exit conditions of the children, as shown in the code example below.

Example: `execvp(char* cmd, char *args[])`

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main (int argc, char *argv[]) {

 char *args[3]; // arguments for the child process.

 args[0] = "ls";
 args[1] = "-l";
 args[2] = NULL; // indicates the end of arguments.

 pid_t pid = fork();

 if ( pid == (pid_t) 0 ) { // child process
   if (execvp(args[0], args) == -1) { // report if error
      perror("execvp: error");
    }
     /* the child goes off, runs our program, then exits. */
    /* the child process will not reach this point */
   /* unless there is an error with execvp */
   exit(1);
 }
 else {

  if (pid == (pid_t) (-1)) { // check if the fork was successful
   perror("Fork failed");
   exit(1);
  }
   int cstatus;
   pid_t c = wait(&cstatus); // wait for the child to complete the command
   printf("Parent: Child %ld exited with status = %d\n", (long) c, cstatus);
 }
  return 0;
}
```

Mini-shell

To finish the lab, combine your tokenizing code from the first part of the lab with the example above so that your program

1. prompts the user to type in a single Unix command at most 64 characters long,
2. runs that command,
3. reports the exit status of the command,
4. repeats steps 1-3 until the user enters DONE.

This should give you a good start on the shell programming assignment.