Assignment Three: Process Control and Writing a Simple Command Shell

EECE 4029: Operating Systems and Systems Programming
ELTN 4022: Operating Systems

Department of Electrical Engineering and Computer Science
University of Cincinnati, Cincinnati OH
Revision Date: July 26, 2021

Student Name: **Brian Culberson**

Task 2: Guided Programming Exercises

1.1: **Guided Programming Exercise:** Modify `HW_02_a.c` to use ONLY pointer arithmetic (no array expressions) and no for loops to do the same thing `HW_02_a.c` does. Be sure that you understand how the code works and how the pointer arithmetic relates to the array expression form. Put your modified code into the PDF of answers you are to turn in (I.E. fill it in below this question IN THE WORKSHEET). Provide liberal comments to explain what your pointer arithmetic is computing. **(5 points)**

```
// Program HW_02_a.c
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char **argv){
  int arg_count = 0; //still need to use this for the loop
  while(arg_count < argc){ //this wile loop replaces the for loop.
    printf("%s\n", *(argv + arg_count)); //changing the array logic to pointer
arithmetic.
    arg_count = arg_count + 1; //increment counter so while loop works
  }
}
```

1.2 Guided Programming Exercise: Modify HW_02_c.c to use `envp` instead of `environ`. Be sure that you understand how the code works. Provide liberal comments to explain what your pointer arithmetic is computing. Paste your source code for this into your answer document. Also, answer the following questions and explain your answers:

```
// Program HW_02_c.c
#include <stdlib.h>
#include <stdio.h>

// extern char **environ; // look into what “extern” means when applied to
                       // a C global variable :)
//dont need this anymore

int main(int argc, char **argv, char **envp){ // added envp
  while (*envp != NULL){ //you can change this to envp
    printf("%s\n", *envp); //print envp
    envp++; //increment envp
  }
  printf("\n");
}
```

(5 points for code)

WHERE in the process memory map do each of these variables exist at run time? **(2 points)**

- `environ`
- `envp`

both variables exist in the data segment since these are both global variables copied from the OS. Extern is used for environ. This is used to get it onto the program. Envp is passed into the program through the main in a similar way was environ.

WHERE in process memory do the actual strings containing the values of the environment variables exist at run time? **(3 points)**

This should still be the dat segment since they are global variables of the main.

1.3 Guided Programming Exercise: Answer the following questions and explain your answers:

- Presume that the code is instantiated in a process and it has JUST returned from its own call to `fork()` and there is now a clone of that process that is itself “just about” to pick up execution. Where will the clone child pick up its execution? **(3 points)**

The next line of execution would be the if statement after the fork. The child does copy itself but continues after the fork statement.

- How does the clone child know it's a clone? How does the parent process know it is not a clone, or at least not the clone if just made by calling `fork()?` **(2 points)**

  The child knows it's a clone because of the process id relative to the parent. In the code it does check the id to see if it is 0. This makes the child know it is a clone.

- Compile and run the program from the command line. Scroll through the whole history of outputs and describe how that two processes (original and clone) seem to be writing to the same terminal. Do you notice anything that is odd? Explain why you might be seeing this oddness. **(3 points)**

  The output at different times but the parent starts earlier. This could be because the child still needs to copy all the data in the executable before it can start but the parent is free to continue. This could explain the parent prints first then they combine outputs and then the child finishes up

1.4a Guided Programming Exercise: Answer the following questions and explain your answers:

- How are the outputs for `HW_02_e.c` and `HW_02_d.c` different? Explain, in your own words, why the output for `HW_02_e.c` appears as it does and must ALWAYS appear as it does. **(2 points)**

  Since there is a wait in the HW_02_e.c thie child can continue and print out, after this the parent can print out after the child has finished.

- What happens if a process that has no children calls `wait(NULL)` You may answer this question either using Internet research OR simply running some test code and examining the results. **(3 points)**

  You would get the same output as HW_02_d.c if the wait wasn't there. Both the child and the parent would print at the same time. Once the parent completes it operation and returns the child would also get to the end and return as well since the child is a clone of the parent.

- Using only `wait(NULL)` calls and any standard C programing constructs you like, write a program that prompts the user the enter a number of children to be created. Your program should create that many children and each child should print a message that says "Hello from child process XXX" where XXX is the process ID of the child. WHEN ALL OF THE CHILDREN FINISH the parent should then, and only then, print a message that says "Hello from parent process XXX" where XXX is the process ID of the parent. Note your program should work for any reasonable request for processes. Your code should be fully commented and you should explain, in comments, what each line of code is doing and why it is doing it. **(2 points)**

```
//Brian Culberson
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>

int main(){
    int children_count = 10;
    pid_t mychildren;

    //get children count
    printf("How many children should I create: ");
    scanf("%i", &children_count);
    printf("Creating %i children\n", children_count);

    for(int i = 0; i < children_count; i++){
        mychildren = fork(); //fork all the children
        if(mychildren < 0){ //error checking
            printf("The fork() didn't work! Terminate\n");
            return 0;
        }else if(mychildren == 0){ //if child go here
            printf("Hello from child ------- %i\n", getpid());
            return 0; //exit child so it does not keep looping
        }
    }
    wait(NULL); //wait for children to finish
    printf("Hello from parent ====== %i\n", getpid()); //parent process
finishes
    return 0;
}
```

1.4b Guided Programming Exercise: Answer the following questions and explain your answers:

- With regard to program `HW_02_f.c`, explain in your own words what each of these macros do: `WIFEXITED(return_status)` and `WEXITSTATUS(return_status)` **(2 points)**

  I would think of it like a error checking status. In try blocks you can add exceptions to check. Since these look at the status and returns true or false based off the status then that is how I would think of them.

- Compile and run `HW_02_f.c` and, presuming your executable is in `a.out`, run this command from the shell prompt:

  ```
  ./a.out ; echo "Parent Terminated Normally with a return v
  alue of $?"
  ```

  Explain, in your own words, why you see the screen output you do and explain how that output relates to both the contents of the program AND the nature of the shell command

used to invoke it. Hint: This has everything to do with how processes "communicate" their exit statues to other processes. **(3 points)**

Since the program returns a value, since this value can be anything you want you can return it to the terminal. Then with echo statements withing linux, you can print this value to the terminal.

1.5 Guided Programming Exercise: Answer the following questions and explain your answers:

Compile and run HW_02_g.c and, presuming your executable is in `a.out`, run this command from the shell prompt `./a.out ls -F` Explain, in your own words, why you see the screen output that you do and how that output related to both the content of the program AND the nature of the shell command used to invoke it. Be sure to comment on the pointer arithmetic done inside the line: `execvp(*(argv+1), argv+1);` **(5 points)**

The screen output changes because execvp rewrites the child and the child will execute a binary file/command when it hits the execvp line using the arguments passed. For this example, running ./a.out with ls -F, the child runs ls -F and the parent waits for the child to complete.

- Compile and run `HW_02_g.c` and, presuming your executable is in `a.out`, run this command from the shell prompt `./a.out` Explain, in your own words, why you see the screen output that you do and how that output related to both the content of the program AND the nature of the shell command used to invoke it. Be sure to comment on the pointer arithmetic done inside the line: `execvp(*(argv+1), argv+1);` **(5 points)**

  the child doesn't run anything because no arguments was passed through ./a.out. since the child rewrites itself for nothing it just exits and the parent follows after that

- Compile and run HW_02_g.c and, presuming your executable is in `a.out`, run this command from the shell prompt `./a.out hoopyfrood` Explain, in your own words, why you see the screen output that you do and how that output related to both the content of the program AND the nature of the shell command used to invoke it. Be sure to comment on the pointer arithmetic done inside the line: `execvp(*(argv+1), argv+1);` Of course, I'm assuming there is no executable file anywhere in your path called "hoopyfrood". If you happen to have an executable named "hoopyfrood" – then substitute a name that does NOT exist in your command path **(5 points)**

  Since the command doesn't exist the statement prints "this prints only if the previous calls fails". This is because the binary file hoopyfrood doesn't exist when it tries to rewrite the child.

1.6 Guided Programming Exercise: Write a C Program that Creates Children and Instructs them via Signals (15 points)

Write C code that does the following. **Turn in your C code as a SEPARATE C language text file with the name "`HW3_Task6.c`".** Your name should be embedded as a comment at the beginning of the file. Your program should:

- Launch five children. The five children should go into endless loops
- Each child should be written so it ignores the first SIGINT signal it gets, but honors the second SIGINT it gets by terminating execution. Each child should print a message to the screen identifying itself with its own PID and saying if it's ignoring or honoring the SIGINT request.
- Five seconds after launching the children, the parent should signal all of its children to terminate with a SIGINT signal. The children should, of course, have been written to ignore this first round of requests.
- Ten seconds after launching the children, the parent should signal all of its children to terminate with a SIGINT signal. In this round, all of the children should, of course, listen and actually terminate.

NOTE: THE PARENT SHOULD NOT IGNORE ANY SIGINT SIGNALS AND SHOULD MAINTAIN USE OF THE DEFAULT SIGNAL HANDLER FOR SIGINT.

Task 2: Write a Simple Command Shell

For this task, you are to modify the "sillyshell_template.c" program to produce a more functional shell. There is a video at this link that you can use to see a demo of a modified silly shell that meets specifications for this task. A full list of the specified capabilities you must provide, and their point values, follows:

- **Built-in Command: printenv (5/40 points)**
  You must add a built-in command that prints ALL of the environment variables of the current shell. The stub for this is in the sample code.
- **Built-in Commands: pwd and cd (10/40 points)**
  You must add build in command for pwd (print the current working directory to the screen) and cd (change the current working directory to what the user specifies). Note that you will need to do some research on how to change a directory. Note that processes can only change the current working directory of themselves. Any changes you make to working directory in the shell will not be reflected in any of the children. This is actually the behavior you want.
- **Launch of Commands as Child Processes (25 of 40 points)**
  You must add code to LAUNCH child processes that are named on via commands to the shell. These processes must exec() the name of the program to be launched according to the current setting of the PATH environment variable.
  - The named process must run with any and all command line parameters you passed it (10/25 points)
  - Control-c sends a terminate signal to the process currently controlling the terminal. YOUR SHELL SHOULD IGNORE control-c. That is, typing control-c at a shell prompt should have no effect. Typing in a control-c while a built-command is running should have no effect. Control-Z sends a SIGSTOP (pause) signal to the process currently controlling the terminal. YOUR SHELL SHOULD IGNORE control-z. That is, typing control-z at a shell prompt should have no effect. Likewise, typing a control-z while a built-in command is running should have no effect (5/25 points)
  - When a non-built-in command is running, typing a control-c should terminate the process. When a non-built-in command is running, a control-z should stop/pause the process. When a command terminates normally, the shell should print a message that says: <PID> terminated normally where <PID> is the process ID of the command that just exited. When a command is paused, the shell should print a message that says: <PID> is stopped. Note that when a process is stopped, it IS possible to restart it by sending it a SIGCONT command from this, or another shell, to that PID. You should test that you can restart stopped processes in this way either from the sillyshell that launched it or from another shell running in another window. (10/25 points)

Turn in your C code as a SEPARATE C language text file with the name "`MY_SHELL.c`". Your **name should be embedded as a comment** at the beginning of the file.