Department of Computer Science and Engineering
Institute of Technical Education & Research, SOA, Deemed to be University

MAJOR ASSIGNMENT-3

UNIX Systems Programming (CSE 3041)

Working with argument array, process creation and use of `fork-exec-wait` combination for the child to execute a new program

The assigment contains three parts. **In the first part**, students will be able to design their own argument array as like **`**argv`** parameter in `main` and be able to use the argument array in various to `exec`-family of functions. **In the second part** student will be able to create a chain of n processes by calling the **`fork()`** in a loop. **In the third part** students will be able to create a hybrid structure by combing the chain and fan of processes.

PART-1

This part aims to use **`execvp`** function to execute a different program that is different from that of the parent. The `exec` family of functions provides a facility for overlaying the process image of the calling process with a new image, and the **`execvp`** is one of them. The function prototype of **`execvp`** is given as;

SYNOPSIS:

```
int execvp(const char *file, char *const argv[]);
```

The arguments to the **`execvp`** will be set from command-line argumnets. The number of command line argument must be 2. Your code must supply an error checking to avoid more or less number(s) of command-line argument(s). An argumnet array named **`myargv`** will be constructed from the command-line string passed to the program. For example if you run the code **`$ ./a.out ``This is a test''`**, the argument array **`myargv`** will be displayed as follows;

```
myargv[0] ---- > This
myargv[1] ---- > is
myargv[2] ---- > a
myargv[3] ---  > test
```

Now, create a user-defined function named **`makeargv`** to create an argument array from a string passed on the command line. The following prototype shows a **`makeargv`** function that has a delimiter set parameter.

```
int makeargv(const char *s, const char *delimiters, char ***argvp)
   ;
```

The `const` qualifier means that the function does not modify the memory pointed to by the first two parameters.

Design your C code called **`ImageOverlay.c`** that creates a child process to execute a command string passed as the first command-line arguments. The program **`ImageOverlay.c`** will call the function **`makeargv`** to create the argumnet array and the created argument array must be used to set the parameter in the **`execvp`** call. The code must handle the error checking such as (1) if **`makeargv`** function returns -1, (2) fork() fails, (3) execvp fails etc. The code will be compiled as **`gcc ImageOverlay.c -o loadnew`** and run as **`./loadnew ``argument list''`**

Tesing

(a) **`./loadnew ``ls -l''`** To show the long listing of files in the current directory

(b) **`./loadnew ``ls -l *.c''`** To show all C files in the current directory

(c) **`./loadnew ``wc''`** to run **`wc`** command as like shell.

(d) **`./loadnew ``grep <patternname> <filename>''`** To search a pattern in the given file

(e) **`./loadnew ``cp <filename1> <filename2>''`** To copy a file

PART-2

Create a chain of n processes. It takes a single command-line argument that specifies the number of processes to create. Before exiting, each process outputs its i value, its process ID, its parent process ID and the process ID of its child. The parent does not execute wait. If the parent exits before the child, the child becomes an orphan. In this case, the child process is adopted by a special system process called `init`. As a result, some of the processes may indicate a different parent process ID. The Sample code for chain of n processes is given below;

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (int argc, char *argv[]) {
   pid_t childpid = 0;
   int i, n;
   if (argc != 2){
     /* check for valid number of command-line arguments */
     fprintf(stderr, "Usage: %s processes\n", argv[0]);
     return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
     if (childpid = fork())
       break;
   fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld
      \n",i, (long)getpid(), (long)getppid(), (long)childpid);
   return 0;
}
```

(a) Run Program above sample code and observe the results for different numbers of processes.

(b) Fill in the actual process IDs of the processes in the chain diagram constructed from the sample code for a run with command-line argument value of 4.

(c) Experiment with different values for the command-line argument to find out the largest number of processes that the program can generate. Observe the fraction that are adopted by `init`.

(d) Place **`sleep(10);`** directly before the final **`fprintf`** statement in the sample code. What is the maximum number of processes generated in this case?

(e) Put a loop around the final fprintf in the given sample code. Have the loop execute **k** times. Put **`sleep(10);`** inside this loop after the **`fprintf`**. Pass **k** and **m** on the command line. Run the program for several values of **n**, **k** and **m**. Observe the results.

(g) Modify the sample code by putting a **`wait`** function call before the final **`fprintf`** statement. How does this affect the output of the program?

(h) Modify the sample code by replacing the final **`fprintf`** statement with **four `fprintf`** statements, one each for the four integers displayed. Only the last one should output a newline. What happens when you run this program? Can you tell which process generated each part of the output? Run the program several times and see if there is a difference in the output.

(i) Modify the sample code by replacing the final **`fprintf`** statement with a loop that reads **nchars** characters from standard input, one character at a time, and puts them in an array called **mybuf**. The values of **n** and **nchars** should be passed as command-line arguments. After the loop, put a **null** character in entry **nchars** of the array so that it contains a string. Output to standard error in a single **`fprintf`** the process ID followed by a colon followed by the string in **mybuf**. Run the program for several values of **n** and **nchars**. Observe the results. Press the Return key often and continue typing at the keyboard until all of the processes have exited.

PART-3

Create a hybrid of chain and fan of n processes. It takes a single command-line argument that specifies the number of processes to create. Design the C code to construct the below process tree structure. Use the wait/waitpid system call so that parent-child relationship will hold and no process will be adopted by the system special process.