CP386: Assignment 1 – Spring 2022

Due on June 1, 2022 (Before 11:59 PM)

This is a group (of two) assignment, and we will try to practice the concept of the parent-child process, inter-process communication, and some related system calls.

General Instructions:

- Submit your source code files for each question file after adding .txt as its extension and name it as a combination of your Laurier ID number & your teammate's ID, an underscore, 'a' (for 'assignment'), and the assignment number in two digits, and question name. For example, if the students 100131001 and 100131233 are submitting Assignment 1 and question 1's part 2, it is named `100131001_100131233_a01_z_creator.c.txt`.
- For this assignment, you must use C99 language syntax. Your code must compile using make **without errors.** You will be provided with a makefile and instructions on using it.
- **Test your program thoroughly with the GCC compiler (version 5.4.0) in a Linux environment.**
- If your code does not compile, **then you will score zero**. Therefore, make sure that you have removed all syntax errors from your code.
- Please note that the submitted code will be checked for plagiarism. By submitting this zip file, you would confirm that you have not received unauthorized assistance in preparing the assignment. You also confirm that you are aware of course policies for submitted work.
- Marks will be deducted from any questions where these requirements are not met.
- Multiple attempts will be allowed, but only your last submission before the deadline will be graded. We reserve the right to take off points for not following directions.

Question 1

A zombie process is a process that has terminated but whose process control block has not been cleaned up from the main memory because the parent process had not waited for the child. In C, create a program ("z_creator.c") that forks a child process that ultimately becomes a zombie process. This zombie process must remain in the system for at least X seconds (value of X you can choose.).

To terminate a Zombie process, we must first identify it using the command <ps -l>. The process states are shown below the S column; processes with a state of Z are zombies. The process identifier (pid) of the child process is listed in the PID column, and that of the parent is listed in the PPID column.

Write a second C program ("z_terminator.c") to automate the process to obtain the status of each process. Identify a zombie process by listing all running processes. Identify and kill its parent process. Print the updated list of all the other processes with their status.

Run the binary file of the above program (z_creator.c) to create a zombie in the background using command <./z_creator &>. We can identify the parent process ID of the zombie process by using command <ps -l| grep -w Z|tr -s ' '|cut -d ' ' -f 5>.

Once we have ID, we can use the "kill -9 <parent process ID>" command to terminate the parent process. You can use the "system()" function for executing the Unix commands in a C program.

To invoke the program, first, use the command: `./z_creator` and use `./z_terminator` in the second terminal OR use command: `make runq1` via `makefile`.

The expected output for Question 1

```
osc@ubuntu:~/a1$ ./z_terminator
In Parent Process!
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
In Child process!
4 S  1000  1147  1072  0  80   0 -  5656 wait   tty1     00:00:00 bash
0 S  1000  1360  1147  0  80   0 -  1054 wait   tty1     00:00:00 z_terminator
0 S  1000  1362     1  0  80   0 -  1088 hrtime tty1     00:00:00 z_creator
0 S  1000  1363  1360  0  80   0 -  1125 wait   tty1     00:00:00 sh
0 R  1000  1364  1363  0  80   0 -  7228 -      tty1     00:00:00 ps
1 Z  1000  1365  1362  0  80   0 -     0 exit   tty1     00:00:00 z_creator <defunct>

The updated list of processes and their status is:
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000  1147  1072  0  80   0 -  5656 wait   tty1     00:00:00 bash
0 S  1000  1360  1147  0  80   0 -  1088 wait   tty1     00:00:00 z_terminator
0 S  1000  1372  1360  0  80   0 -  1125 wait   tty1     00:00:00 sh
0 R  1000  1373  1372  0  80   0 -  7228 -      tty1     00:00:00 ps
```

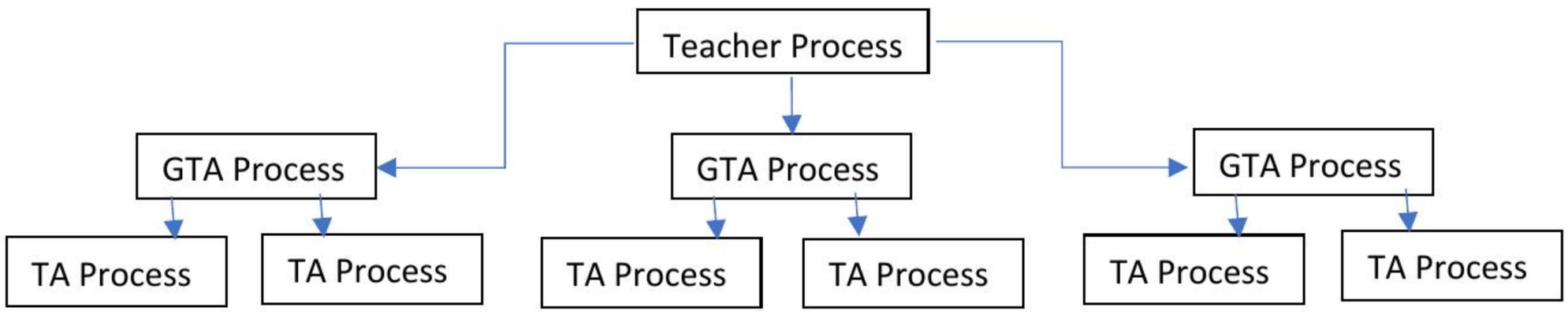
Note: When submitting source code files for Question 1, name them like:

- `FirstID_SecondID_a01_z_creator.c.txt`
- `FirstID_SecondID_a01_z_terminator.c.txt`

Question 2

Create a C program (name it "assignment_average.c") that finds the average grade of the course assignments. The teacher gives two assignments for every chapter. You must calculate the average grade for each assignment in all the chapters.

- The program should have a teacher process that reads the grades ("sample_in_grades.txt" grades file is available under the Assignment 1 section on Myls) of all assignments of all the chapters and creates a two-dimensional matrix of grades.
- Teacher process then creates a "GradTA" process. Each "GradTA" process takes care of solving a chapter.
- Each "GradTA" process would create "TA" processes and pass one assignment to each of them. "TA" process would calculate and print the average for that assignment. The process tree is shown below:



The input file "sample_in_grades.txt" contains data for three chapters and two assignments for each chapter (six columns). The first two columns are grades for two assignments for chapter 1, the following two columns are the grades for two assignments for chapter 2, and the last two columns are the grades for two assignments for chapter 3. The Grades file contains grades for 10 students.

The expected output for Question 2:

```
Assignment 1 - Average = 10.200000
Assignment 2 - Average = 11.600000
Assignment 3 - Average = 13.700000
Assignment 4 - Average = 13.600000
Assignment 5 - Average = 9.400000
Assignment 6 - Average = 11.600000
```

Note: When submitting source code files for Question 2, name them like:

- `FirstID_SecondID_a01_grades.c.txt`

Question 3

In C, write a program (name it "process_manangment.c") that includes the code to accomplish the following tasks:

1. A parent process use fork system calls for creating children processes, whenever required, and collecting the output of these. The following steps must be completed:
    a) Creation of a child process to read the content (A LINUX COMMAND PER LINE) of the input file. The file contents will be retrieved by the child process in the form of a string using a shared memory area.
    b) Creation of another child process that will execute these Linux commands one by one. The process will give the output using a pipe in the form of a string.
    c) Write the output after executing the commands in a file named "output.txt" by the parent process.
2. The parent process can use a fork system call for creating children processes whenever required.
3. Chapter 3 in the book discusses the POSIX API system calls for creating/using/clearing shared memory buffers.
4. The child process will read the contents of the file ("sample_in_process.txt"), not the parent process. The command-line arguments must be used by the parent process to read the file name.
    a) In this file, one shell command per line is present.
    b) The file's contents will be written to shared memory by the child process to allow the parent process to read it from there.
    c) After the file's contents are read, the termination of the child process will be performed.
5. Then the contents from the shared memory area will be copied to a dynamically allocated array.
6. Further, the following commands will be executed one by one during the child process:
    a) For executing shell commands, `execvp()` system call can be used.
    b) The output of the command to be written to a pipe by the child process, and the parent process will read it from the pipe and write it to a file via the output function.
    c) One child process **MUST** be used by repeatedly forking to execute all the commands.
    d) The given output function, `writeOutput()`, will be used by the parent process to write the output to a file ("output.txt"). The output function must be invoked iteratively for each command.

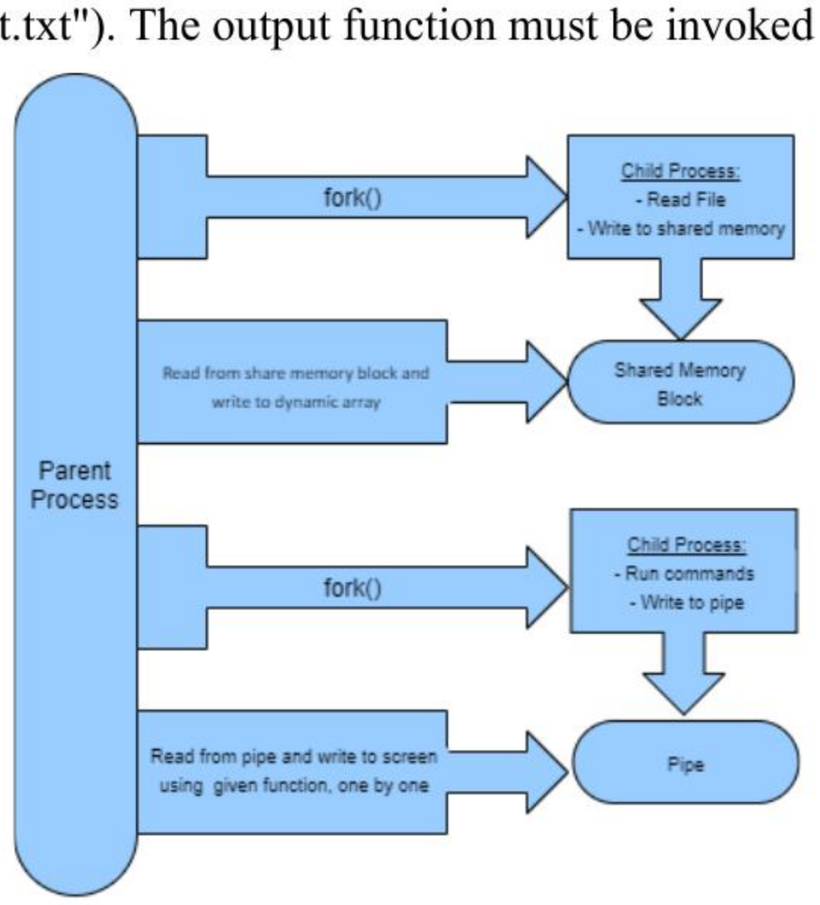


Figure 1: The flow chart

7. Figure 1 flow chart describes the flow of the program.

8. Use makefile to compile the program written above, the instructions to use and contents of the makefile have been provided on the Myls course page.
9. The other implementation details are at your discretion, and you are free to explore.

To invoke the program, first use the command: `./process_management sample_in_process.txt` in terminal OR use the command: `make runq2` via `makefile`.

Hint: The function for writing the results of executing the command you can use the function:

```
void writeOuput(char *command, char *output) {
        FILE *fp;
        fp = fopen("output.txt", "a");
        fprintf(fp, "The output of: %s : is\n", command);
        fprintf(fp, ">>>>>>>>>>>>>>>\n%s<<<<<<<<<<<<<<<\n", output);
        fclose(fp);

}
```

The expected output for Question 3 is:

```
The output of: uname -a : is
>>>>>>>>>>>>>>>
Linux ubuntu 4.4.0-87-generic #110-Ubuntu SMP Tue Jul 18 12:55:35 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
<<<<<<<<<<<<<<<
The output of: free -m : is
>>>>>>>>>>>>>>>
              total        used        free      shared  buff/cache   available
Mem:            740         221          85           2         432         365
Swap:           765          18         747
<<<<<<<<<<<<<<<
The output of: ls -l -a -F : is
>>>>>>>>>>>>>>>
total 36
drwxrwxr-x  2 osc osc  4096 Feb 10 15:36 ./
drwxr-xr-x 12 osc osc  4096 Feb 10 15:35 ../
-rwxrwxr-x  1 osc osc 14088 Feb 10 15:36 a.out*
-rw-rw-r--  1 osc osc   431 Feb 10 15:36 output.txt
-rwxrwxr-x  1 osc osc  3057 Feb 10 15:34 process_management.c*
-rwxrwxr-x  1 osc osc    39 Feb 10 15:34 sample_in_process.txt*
<<<<<<<<<<<<<<<
The output of: whoami : is
>>>>>>>>>>>>>>>
osc
<<<<<<<<<<<<<<<
The output of: pwd : is
>>>>>>>>>>>>>>>
/home/osc/Assignment1
<<<<<<<<<<<<<<<
```

Note: When submitting source code files for Question 3, name them like:

- `FirstID_SecondID_a01_process_management.c.txt`