CSci 4061: Introduction to Operating Systems

Programming project 1: OS Basics **due: Thursday February 15th, 2024**

Ground Rules. You may choose to complete this project in a group of up to three students. Each group should turn in **one** copy with the names of all group members on it. The code must be originally written by your group. No code from outside the course texts and slides may be used—your code cannot be copied or derived from the Web, from past offerings, other students, programmer friends, etc. All submissions must compile and run on any CSE Labs machine located in KH 1-250. A zip file should be submitted through Canvas by 11:59 pm on Thursday, October 7th.

Note: Do not publicize any project document or your answer to the Internet, e.g., public GitHub repo.

1 Objectives

The main objectives of the project are

- Brush up 'C' programming skills
- Use Makefile to compile code
- Hands-on experience with child process creation(fork/exec/wait)
- Learn how to execute processes in sequential or in parallel

2 Tasks

The main tasks of the project are

- Fill in the blanks in the *main.c*
- Complete functions declaration in the *graph.h*
- Complete functions definition in the *graph.c*
- Complete the *makefile* to generate the executable file.

3 Introduction

In practice, we often need to finish a complicated computing task that consists of multiple jobs that are dependent on others. For example, we all will have encountered issues with the installation of some applications. Likely, some errors mention "Package X for installation of Package Y is not found". So we first install Package X and then go on to successfully install Package Y. This entire dependence between different packages can be represented using a dependency graph. During installation, the final product is installed only after all the nodes in the dependency graph are successfully downloaded and linked. In this project, you will be provided with a dependency graph that consists of multiple dependent nodes (each node represents one command). The root of the graph should only be executed after the child nodes are successful.

4 Project Description (Part 1 Sequential)

Given a directed graph, execute the commands in the nodes using fork/exec/wait calls using the Depth-First Search order. The input to the program executable will be a text file, as shown in Section 5. You may use any data structure to represent the graph according to your logic. Once the graph is constructed, processes should be spawned according to the traversal of nodes. The process corresponding to the first node to have zero out-degree in DFS (zero outward edges) should be executed first. For simplicity, the graph provided is a tree, which means there are no cycles, and the tree nodes have only one incoming edge. Before executing a node, you should first execute all the children of that node from left to right. On executing a node, the process should write "process_id parent_process_id executable arguments" into "output.txt". Parallel execution of nodes is not expected for part 1 but expected for part 2. The dependency graph should be executed as given in the sample example in Section 5.

A node of your data structure may have the following information. A sample structure is provided in the "graph.h" file in the project template. You are allowed to create your own node structure. **Please note that instructions in the .c and .h files are written based on the sample structure.**

```
struct node{
int visit; // DFS visit usage
struct connected_nodes *head; // nodes to be executed before this node
};
```

You may create a character array to store the executables and arguments

```
char cmds[32][550]; // check Assumptions section
```

Your code should consist of error-handling mechanisms for the system calls. The error printed out should be meaningful. Also, once the the program completes the graph execution, please make sure to free the dynamically allocated memory.

5 Sample Input

A sample input file is given below:

```
// Number of nodes in the graph
4
// Executable and arguments corresponding to each node
/bin/ls -l //Node 0
/bin/pwd // Node 1
echo Hello There //Node 2
echo Hello //Node 3
// source destination
0 1
0 2
2 3
```

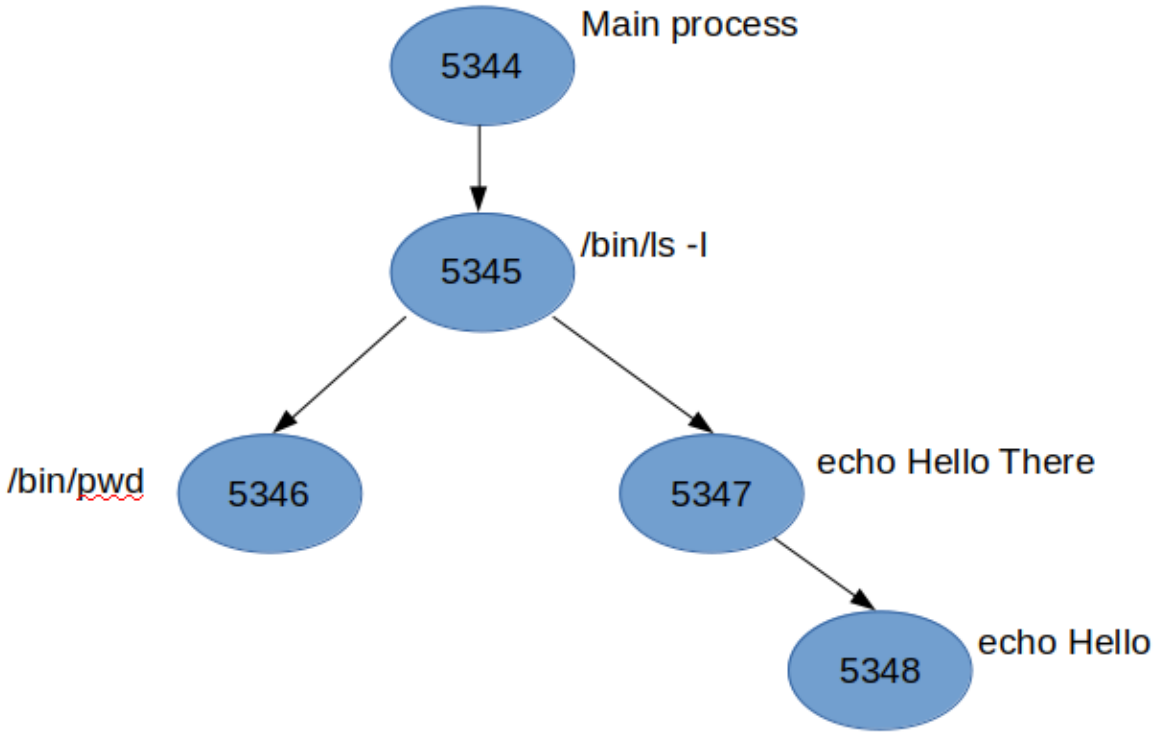

Figure 1: Dependency graph for given example

Note that the comments provided in the above example will not be present in your "input.txt". Please check the "Testcase" folder for more examples.

The 'input.txt' will consist of a maximum of 32 nodes, which implies your program will be spawning a maximum of 32 processes + 1 process for the main process. The content of the input.txt will have three sections. First, an integer value, 'V' represents the total number of nodes in the graph. Second, the next set of 'V' lines correspond to the executables and arguments corresponding to each node. Third, the directed edges between different nodes (source-destination). Each section is separated by a new line.

Note that the name of "input.txt" file can change. Do not hard code the name. Also, there can be a path with the input file name. For example

```
../Testcases/input1.txt
Testcases/input.txt
inputxyz.txt
```

6 Sample Output

The sample results.txt looks like

```
5346 5345 /bin/pwd
```

```
5348 5347 echo Hello
5347 5345 echo Hello There
5345 5344 /bin/ls -l
```

Note that the leftmost node in the graph is executed first and then the execution moves to the right. This is by the order in which node relations are given in the 3rd section of the "input.txt" file. The same output format should be followed. The delimiter is a single space.

The sample output on the terminal is as below:

```
/home/user1/F1
Hello
Hello There
total 12
-rw-rw-r-- 1 user1 user1 2 Feb 3 10:28 F1.txt
-rw-rw-r-- 1 user1 user1 2 Feb 3 10:28 F2.txt
-rw-rw-r-- 1 user1 user1 2 Feb 3 10:28 F3.txt
```

7 Enabling Parallel Execution (Part 2 Parallel)

Now, we assume that the children of a node are independent of each other. So we can execute all the children of a node in parallel (don't have to be from left to right). However, this does not mean that a child and its parent can be executed at the same time. **A node can be executed only after all the children have been executed.**

After we enable parallel execution, the order of terminal output may differ from each run. An output is valid as long as the children are executed before the parent. If you want to have the same output in results.txt even after parallel execution is enabled, this involves the locking mechanisms, which will be covered in the future.

For example, the following output in the terminal is valid for input1.txt:

```
Hello
/home/ta/csci4061/projects/p1/Solution
Hello There
total 12
-rw-r--r-- 1 ta ta 2 Sep 20 23:30 F1.txt
-rw-r--r-- 1 ta ta 2 Sep 20 23:30 F2.txt
-rw-r--r-- 1 ta ta 2 Sep 20 23:30 F3.txt
```

The following output is invalid because echo Hollo There is executed before echo Hello:

```
/home/ta/csci4061/projects/p1/Solution
Hello There
Hello
total 12
-rw-r--r-- 1 ta ta 2 Sep 20 23:30 F1.txt
-rw-r--r-- 1 ta ta 2 Sep 20 23:30 F2.txt
-rw-r--r-- 1 ta ta 2 Sep 20 23:30 F3.txt
```

Your program should execute nodes either in parallel or in DFS order sequentially based on whether a `-p` (for parallel) flag is set.

8 Execution Format

The executables name should be **depGraph**

DFS order:

```
./depGraph input.txt
```

Parallel execution:

```
./depGraph -p input.txt
```

9 Testing for DFS Order

Sample input files are provided in the "Testcase" folder. The expected output for each is provided in the "Testcase/ExpectedOutput" folder. Execute the input files separately and check the correctness with the given sample results. The process id is the only value that should be different. Note that there can be other cases that are not covered in the given input files.

TAs will be having a testing script to check your code for different input cases. Before the execution of each case, the "results.txt" the file will be deleted. So you do not have to write code for deleting the "results.txt" explicitly. Note that for your own testing purpose, you are free to do it.

10 Testing for Parallel Execution

Similar to Section 8 but the output order in the terminal may differ from each run. The output in results.txt may remain unchanged due to file lock.

However, ./depGraph -p input5.txt should have the output in terminal as below:

```
1
2
3
4
5
6
7
8
9
sleep sort done
```

11 Assumptions

- Maximum number of nodes is 32
- Maximum number of arguments for an executable is 10
- Maximum number of characters in an executable or an argument is 50

12 Deliverables (Please read carefully)

Directory structure

- template
  - F1
    - F1.txt
    - F2.txt
    - F3.txt
  - graph.c
  - graph.h
  - main.c
  - main.h
  - Makefile
  - README
- testcases
  - input1.txt
  - ...
  - input5.txt

One student from each group should zip the **template** folder and upload it to Canvas. The README in the template folder should include the following details:

- test machine, date, team member names, and x500
- Your and your partners' contributions
- How to compile the program
- The purpose of your program
- What exactly your program does
- Any assumptions outside this document (If you have)

The README file does not have to be long, but must properly describe the above points. Proper in this case refers to – the first-time user can answer the above questions without any confusion. **Within your code, you should remove the provinided comments and use your own words to describe each function you wrote.** You do not need to comment on every line of your code, but make sure your comments help graders understand your code easily. At the top of your README file and main C source file, please include a comment in the following form

```
/*test machine: CSELAB_machine_name
* date: mm/dd/yy
* name: full_name1, [full_name2]
* x500: id_for_first_name, [id_for_second_name]
*/
```

13 Grading Rubric

- 5% For correct README contents
- 5% Code quality, such as using descriptive variable names and comments
- 10% Error checking for all system calls
- 20% Correct use of fork/exec/wait
- 20% Solution can correctly execute at least one node.
- 20% Solution can execute dependency graph in DFS order correctly (10 test cases, each 2%)
- 20% Solution can execute dependency graph in parallel correctly (10 test cases, each 2%)
- 5 input files are given and we will have another 5. You could do more tests by yourself.