CSCI 4061: Introduction to Operating Systems
Assignment 1: Vote Count Application

Due: February 21, 2017, 11:59 pm. We recommend you work in groups of up to 2 people.

1 Purpose:

The purpose of this lab is to understand Dependency Control flows using the concept of processes in C using fork, exec and wait.

Any Program dependency is composed of two parts:

1) A control dependency which specifies that a program cannot start until its predecessor(s) is finished

2) A data dependency specifies that a program requires input from its predecessor(s) before it can execute.

Many modern Big Data Computing systems such as Hadoop use these kind of workflows to solve complex data processing problems.

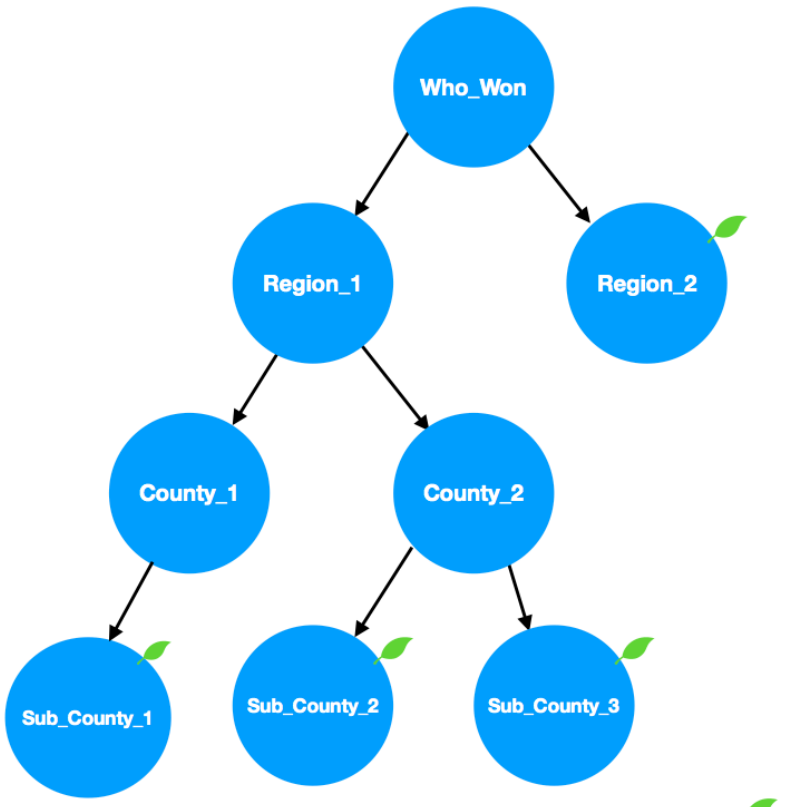

Figure 1

We have both dependencies in this scenario but we are only dealing with (1) in this lab and put this in perspective in multiple stages to solve our problem statement.

2 Description:

Your main program, 'votecounter' will be responsible for finding who won the election, after analyzing the stages in election vote counting, determining which ones are eligible to run at the moment in time. As stages of the vote counting system finish executing, your program will determine which other stages in the vote counting process have become eligible to run, execute them and continue this process untill we reach goal : Who won the election?

Each region may be subdivided further into multiple regions, which may further be divided into multiple regions. This would lead to an increase in stages of execution. A sample DAG with populated nodes for such a case would look as below:

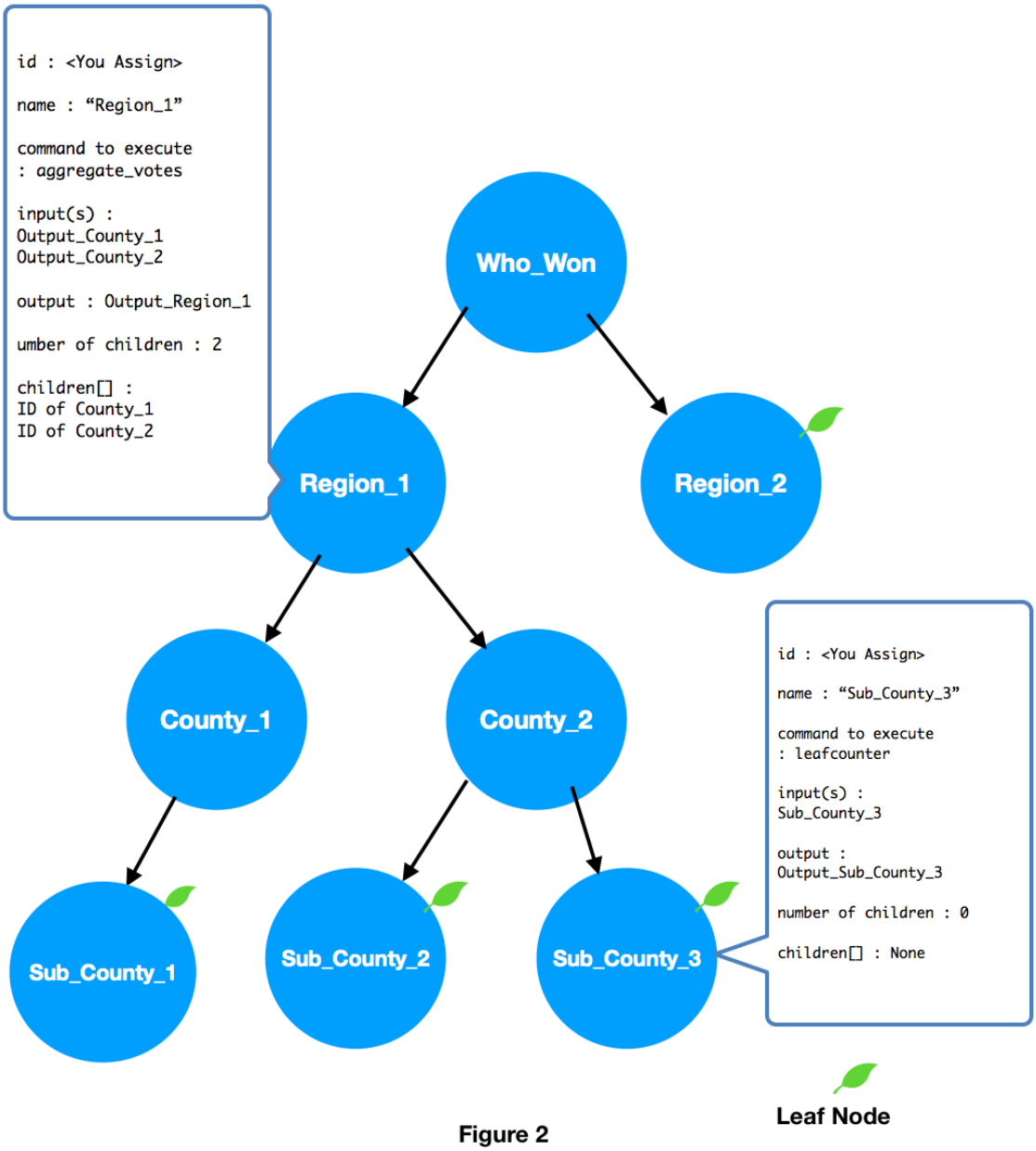

Figure 2

Each node in the graph represents one program to be executed. Each node will contain:
- The program name with its arguments
- "Pointers" to/Lists of child nodes
- The input files to be used as standard input for that program
- The output file to be used as standard output for that program

A node becomes eligible for execution once all of its child nodes have completed their own execution. Your main program will fork and exec each of the nodes as they become eligible to run.

3 Node Structure:

The below structure of a node for your DAGs is recommended. You are free to create your own structures.

```
typedef struct node{
      char name[1024];
      char prog[1024];
      char input[50][1024];
      char output[1024];
      int children[10];
      int num_children;
      int status;
      pid_t pid;
      int id;
}node_t;
```

4 High-level View:

Your program is supposed to parse the input file given in the first argument, ie, input.txt construct a data structure that models the graph, and then start executing the nodes. You are going to assign IDs & names to each of the nodes. Your program should determine which nodes are eligible to run, execute those nodes, wait for any node to finish, then repeat this process until all nodes have finished executing.

Each of the nodes perform a task, governed by program and it's arguments and generates an output. The parent's inputs will be it's children's outputs.

The data processing happening at leaf nodes is different to that happening at the parent nodes. The leaf nodes count individual votes as shown in the Section 'Input Format' above. A utility called 'leafcounter' is supposed to run this counting.

The utility supposed to run at the non-leaf except for the "Who_Won" node is called 'aggregate_votes'. It outputs the names of candidates with their aggregated votes.

The "Who_Won" node is supposed to call "Find_Winner" Utility. This compares against each of the candidates, their total votes and outputs:

1) The total votes of each candidate
2) The winner candidate's name
Please refer to the 'Utility Handbook' for more details on this.

5 Input File Format:

We will run your votecounter program as follows:
$ ./votecounter input.txt

Input.txt - Node Information

One text file will represent the information you need to know about the election.

1) Line 1 : How many candidates are participating in the election
2) Line 1 : The names of the candidates
3) Line 2: Specifies all the nodes which need to be created
4) Line 3 onwards : The next few lines are going to reflect on the DAG to be made.
5) Lines that start with # are commented out and to be ignored.
6) There will be an Input File each for the Leaf Nodes, which are the only nodes supposed to be counting individual votes. The rest of the nodes are only aggregating these inputs.
7) The names of these nodes *could be anything in alphanumeric, but containing atleast one alphabet character*
8) Refering to the sample input shown below, Line 3 is in two parts, separated with a colon ':'. The left side specifies a parent node, and it's children are specified to the right. You are supposed to create DAG's parent/child connections using this.
9) Only the parent nodes with their children are mentioned from Line 3 onwards
10) Any node can be a leaf node, except the "Who_Won" node.
11) The input file given to the program will always be named "input.txt"

This is a sample input text file supposed to make the DAG in Figure 3.

Sample

input.txt

```
3 A B C
Who_Won Region_1 Region_2 County_1
County_2 Sub_County_1 Sub_County_2
Sub_County_3
Who_Won : Region_1 Region_2
Region_1 : County_1 County_2
County_1 : Sub_County_1
County_2 : Sub_County_2 Sub_County_3
```

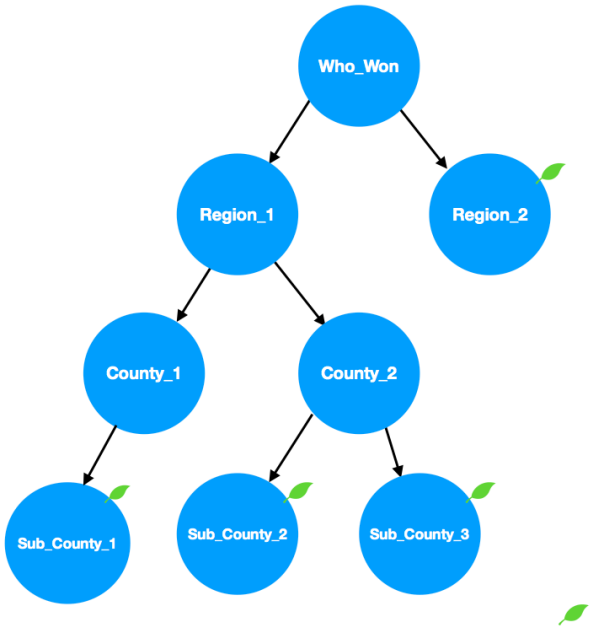

Figure 3

NodeName - Votes Information

1) These are the data files consumed by the program, and they should be as many as the number of leaf nodes.
2) These files contains individual votes (shown in Figure 4)
3) These files *are named after the leaf nodes*
4) The executable utility called 'leafcounter' is supposed to be calculating these votes
5) This utility will store the output in the filename you have specified as an argument while calling it (Please refer to the utility handbook).

The inputs to leaf nodes:
(A/B/C is the candidate's name as per the input file)
File name: Sub_County_1
A
A
A
B
C
B
B
B
C
…

As seen above, the file "Sub_County_1" pertains to the votes caste by each person in a separate line, for an individual candidate. *The files for all leaf nodes, will have similar format*. Their names could be different.

The utility 'leafcounter' will parse this and generate an output(the name of which you are supposed to specify) such as:
(A/B/C is the candidate's name as per the input file and the total number of votes are next to their names)
Ouput_Sub_County_1
A 54
B 2
C 10

Ouput_Sub_County_2
A 10
B 0
C 0

The utility ‘aggregate_votes’ will parse the above two inputs and generate an output(the name of which you are supposed to specify) such as:

Output_County_1

A 64
B 2
C 10

The utility ‘aggregate_votes’ will parse the inputs of node Region_1 and generate an output(the name of which you are supposed to specify) such as:

Output_Region_1

A 84
B 20
C 30

The utility ‘aggregate_votes’ will parse the inputs of node “Who_Won” and generate an output(the name of which you are supposed to specify) such as:

Output_Who_Won
A 140
B 120
C 160
Winner is C

6 Output:

The output *must be* be a simple text file named “**Output_Who_Won**” with :
1) The total votes of each candidate
2) The winner candidate’s name as shown below
Example:
Output_Who_Won
A 140
B 120
C 160
Winner is C

7 How To Run:

You are supposed to ensure that your program runs with a makefile. *You are supposed to write this makefile.*

$> make

[This command should execute your makefile without errors and will thus generate the executable]

$> ./votecounter input.txt

Please paste the votecounter file to TestCases folder to run it with different TestCases. If you want to try with your own TestCases, please make sure the utility files are in them.

8 Do's and Dont's

You can not use the library call system which invokes a shell command externally from your program. You must use fork and exec instead. You need to only execute a single graph) with each invocation of your votecounter program, but you must execute the graph in a stage-by-stage fashion. However, this does not mean that the processes which are independent of each other must not run in a parallel fashion.

That is, the direct children or dependents of a parent node must be forked one by one. Once one of them is done, then you should fork the next dependent. Do this until all the dependents are done. You MUST avoid fork bombs, and clean up any processes that linger as a result of broken code, especially zombies.

The command ps -u < userid > shows your processes and kill9 < processid > will get rid of them at the shell.

We have included a possible data-structure for your graph node. You are free to use fancy dynamic data-structures, however, if you wish. In general, you are free to use any or none of the provided code.

9 Useful System Calls & Functions

These are some useful calls: *(They are suggestions, by no means mandatory)*

fork, dup2, execvp, wait, strtok.

9.1 The Utilities Handbook

Some utilities which you might potentially want to use have been provided in makeargv.h and as executables. Please refer to the 'Utility Handbook' file provided to you.

10 Simplifying Assumptions

1. Command, parameters can only contain alphanumeric characters. Special characters such as n, <, >, & are not allowed.

2. You can assume that each file name will never be more than 1023 characters.
3. You can assume that every command will be on your PATH, thus you do not need to use absolute paths for a command.
4. You can assume that the number of nodes will not be more than 50.
5. You can assume that there are no FLAGS to the utilities and commands.
6. The input of Graph Information will always be called “input.txt”
7. The Final output file should be called “Output_Who_Won”
8. You may name the other outputs of nodes anything, as long as the functionality remains correct.
9. The leaf node input files are going to be named the same as the leaf nodes
10. All executables provided were compiled on CSELABS machine. It may be cross compatible with some Linux flavours, but not all. It is not compatible with Mac.

11 Error Handling:

You are expected to check the return value of all system calls that you use in your program to check for error conditions. Also, your main program should check to make sure the proper number of arguments is used when it is executed. If your program encounters an error, a useful error message should be printed to the screen. Your program should be robust; it should try to recover if possible. If the error prevents your program from functioning, then it should exit after printing the error message. (The use of the perror function for printing error messages is encouraged.)
You may assume that the Lines of the ‘input.txt’ file will contain information in the same pattern as has been shown in sample input, with Line 1, 2 and 3 onwards depicting respective data. However, do not assume that this data may always be correct. For this, you need proper error handling. Thus, if the first line not beggining with # and containing data, begins with a non-numeric value, then you are supposed to report error.
Cases such as what if there are no regions to vote, missing input files must be covered. This is not a list of sufficient Edge cases, but necessary ones.

11 Documentation:

You must include a README file, named "README" without extensions, which describes your program.
It needs to contain the following:

- The purpose of your program
- How to compile the program
- How to use the program from the shell (syntax)
- What exactly your program does

- Your x500 and the x500 of your partner
- Your and your partner's individual contributions

The README file does not have to be very long, as long as it properly describes the above points. Proper in this case means that a first-time user will be able to answer the above questions without any confusion. Within your code you should use one or two sentences to describe each function that you write. You do not need to comment every line of your code. However, you might want to comment portions of your code to increase readability.

At the top of your README file and main C source file please include the following comment:

```
/*login: itlabs_login_name
*  date: mm/dd/yy
*  name: full_name1, full_name2
*  id: id_for _first_name, id_for_second_name */
```

12 Grading:

5% README file

15% Documentation within code, coding, and style (indentations, readability of code, use of defined constants rather than numbers)

80% Test cases (correctness, error handling, meeting the specifications)

- Please make sure to pay attention to documentation and coding style. A perfectly working program will not receive full credit if it is undocumented and very difficult to read. It will not receive full points if your README lacks the correct name ("README") or does not include your (and your partner's) x500.
- You will be given most test cases up front. You are also encouraged to make up your own tests. If there is anything that is not clear to you, you should ask for clarification.
- We will use the GCC version installed on the CSELabs machines to compile your code. Make sure your code compiles and runs on the CSELabs machines.

13 Deliverables:

- Files containing your code
- A README file

- Do NOT include the executable

Note: Your makefile will be used by us to compile your program with the make utility. All files should be submitted using Canvas by only one group member. This is your official submission that we will grade.

14 Extra Credit

For up to 10 additional points, your `votecounter` program should find a cycle in the DAG. A cycle is when a node points to any of its ancestors. This would mean, if any node's children are pointing it either it, or to nodes which it's a child of.

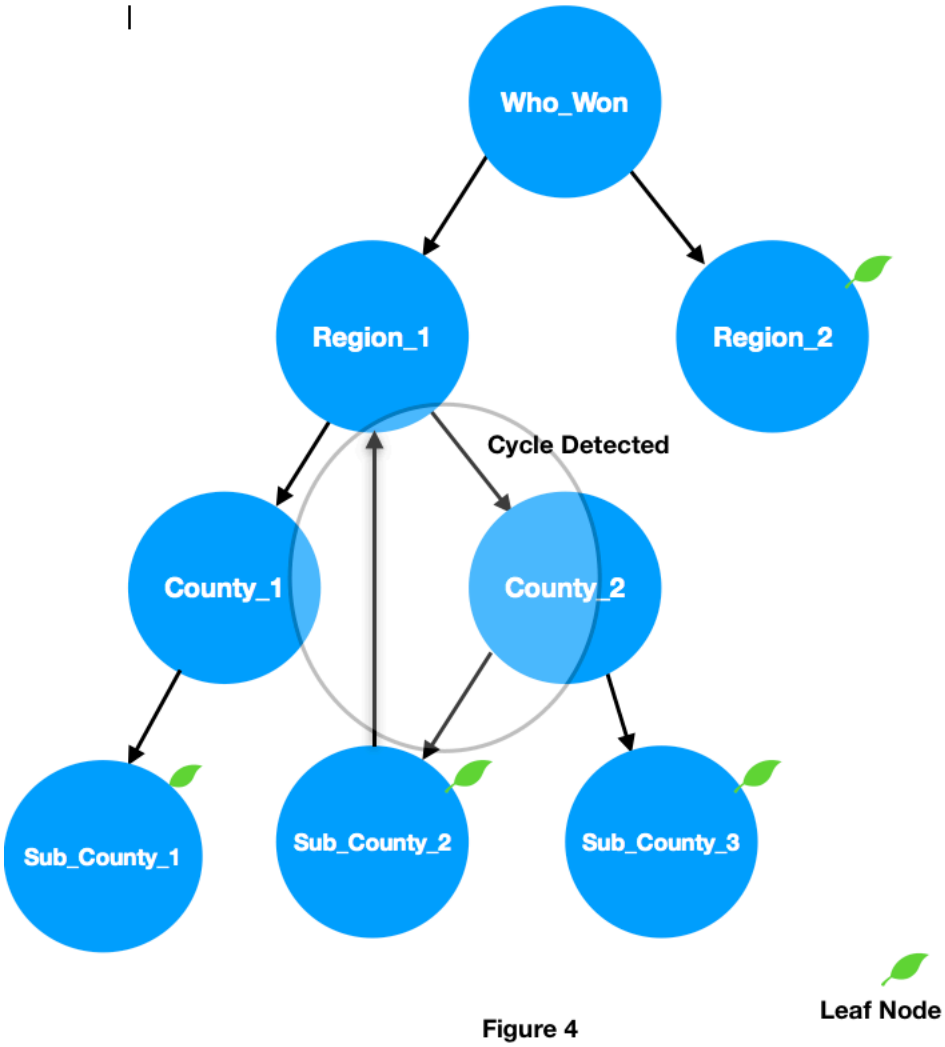

Figure 4