CSci 4061 Programming Assignment 2

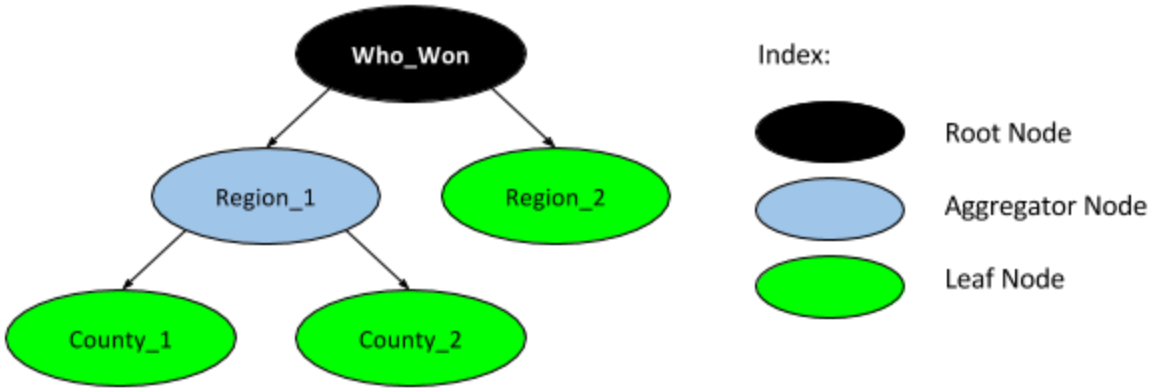
Due: Friday, 03/09/2018, 23:59.
Weight: 100 points, 10% of total grade.
Rule: The assignment must be done in a group of 1 or 2. However, group of 2 is recommended.

Introduction

The purpose of this lab is to learn using the Operating System's I/O and file system calls. In particular, you will do some practices with File I/O operations, File pointers and buffering, File descriptors and I/O redirections, file system, and directories. In the previous assignment, we provided several utility functions, e.g., to read, parse, and write a file. In this assignment, you will need to create your own functions to do those functions.

The Overview of Our Existing Vote Count Application

Recall that our vote count application has three types of node: (1) root node, (2) aggregator node, and (3) leaf node. An example of the application can be seen in the figure below:



In **our existing** vote count application:

1) The leaf node will execute a "`leafcounter`" executable which will read an input file and write out the summary of the input file into an output file.
2) The aggregator node will execute a "`aggregate_votes`" executable which will read *n* intermediate results and aggregate them into a single output file.
3) The root node has two main functions: (1) to parse an <input.txt> which contains the information of the DAG (Directed Acyclic Graph) and (2) to execute "`Output_Who_Won`" executable which will read *n* intermediate results, aggregate them, and output the winner to an output file.

Your Assignment

In this assignment, you will create a new breed of the Vote Counter application in which the structure of the graph is given in the form of a directory structure. For example, the figure above can be translated into a structure of 5 directories with a root node of Who_Won directory as follows:

```
Bob@foo:/$ ls
Who_Won

Bob@foo:/$ cd Who_Won; ls
Region_1 Region_2

Bob@foo:/Who_Won$ ls Region_2
votes.txt //Contains the votes for Region_2.

Bob@foo:/Who_Won$ cd Region_1; ls
County_1 County_2

Bob@foo:/Who_Won/Region_1$ ls County_1; ls County_2
votes.txt //Contain the votes for County_1.
votes.txt //Contain the votes for County_2.
```

Your application will need to traverse the directories and aggregate the votes as needed. Depending on your implementation, your application may traverse the directories in a depth-first search (DFS) or in a breadth-first search (BFS) manner. Furthermore, you will need to write your own function to read and write the result to a file. In general, your assignment is to create **three** executables, i.e., Leaf_Counter, Aggregator, and Vote_Counter that will be discussed below.

Setup

You may use your assignment 1 as a starting point for the project. However, it is **recommended** to start the assignment from scratch as there will be many changes to your assignment 1 code. However, you may use some functions from your assignment 1 to do this assignment, e.g., the Makeargv.h
However, you are **not** allowed to use any binaries that we have provided for programming assignment 1. There are three executables that you will need to create for this assignment. (**Note:** the name of the executables must be the same with the following):

Executable 1: Leaf_Counter

This executable will act as the leaf nodes.

Input:
This program has **one** argument and will be executed as follows:
`./Leaf_Counter <path>`

- `<path>`: is the relative path to a directory.

Main Functionality:

- This program has a similar functionality with your first programming assignment in which it will read the votes file located at the path, i.e., "`<path>/votes.txt`", aggregate the votes for each candidate, and output the results into a file that is also located in the path.
- The votes file will always be called "`votes.txt`" and the output file name will always be the path name with ".txt" appended to it. For example, if the `<path>="Who_Won/Region_1/County_1"`, the output file will be `"Who_Won/Region_1/County_1/County_1.txt"`.

Output:

- If the program is called on a directory that does **not** have votes.txt, it writes to the standard output the following message: "Not a leaf node.\n"
- Otherwise, the program has two outputs: (1) output the counting result into an output file whose name is the last directory name in the path appended with ".txt" and (2) write to the standard output the path to the output file ended with a new line character.
- The output file contains a counting result in the form of: "<candidate_1>:<count_1>,<candidate_2>:<count_2>,...,<candidate_n>:<count_n>\n"
- If the output file already exists, it will **replace** it with the new one.

Example: (Look at figure above for the structure of the DAG)

```
Bob@foo:/$ ./Leaf_Counter Who_Won/Region_1
Not a leaf node.

Bob@foo:/$ ls Who_Won/Region_1/County_1
votes.txt

Bob@foo:/$ cat Who_Won/Region_1/County_1/votes.txt
A
B
A
C

Bob@foo:/$ ./Leaf_Counter Who_Won/Region_1/County_1
Who_Won/Region_1/County_1/County_1.txt

Bob@foo:/$ ls Who_Won/Region_1/County_1
vote.txt County_1.txt

Bob@foo:/$ cat Who_Won/Region_1/County_1/County_1.txt
A:2,B:1,C:1
```

Executable 2: Aggregate_Votes

This program will act as the aggregator nodes.

Input:

This program has **one** argument and will be executed as follows:

`./Aggregate_Votes <path>`

- `<path>`: is the relative path to a directory.

Main Functionality:

- This program has a similar functionality with your first programming assignment in which it will aggregate all the voting results of **all subdirectories**, i.e., its children, that are defined in `<path>` and write the aggregation result to a file in `<path>`. The aggregation result file will have the same name as the current node appended with ".txt"
- If we call this program on a leaf node, then it will execute the Leaf_Counter program.
- Each subdirectory (or child) can either be: (1) a leaf node or (2) a non-leaf node. If the child is a leaf, then this program will spawn a child process to execute the Leaf_Counter program. If the child is a non-leaf, then this program will spawn a child process to execute another Aggregate_Votes program.
- The child process **must not** print their result into the standard output.
- If a directory is not a leaf node, however, contains a "votes.txt", then this program **must ignore** the "votes.txt".

Output:

- The program has two outputs: (1) output the aggregation result into an output file located in `<path>` with a file name of the directory name appended with ".txt" and (2) write to the standard output the output file path ended with a new line character.
- The output file contains a counting result in the form of: "<candidate_1>:<count_1>,<candidate_2>:<count_2>,...,<candidate_n>:<count_n>\n"
- If the output file already exists, it will **replace** it with the new one.

Example: (Look at figure above for the structure of the DAG)

```
Bob@foo:/$ ls Who_Won/Region_1/County_1
votes.txt County_1.txt

Bob@foo:/$ ./Aggregate_Votes Who_Won/Region_1/County_1
Who_Won/Region_1/County_1/County_1.txt

Bob@foo:/$ cat Who_Won/Region_1/County_1/County_1.txt
A:2,B:1,C:1

Bob@foo:/$ cat Who_Won/Region_1/County_2/votes.txt
A
D
```

```
Bob@foo/:$ ./Aggregate_Votes Who_Won/Region_1
Who_Won/Region_1/Region_1.txt

Bob@foo/:$ ls /Who_Won/Region_1/County_2 //Check the output of County_2
votes.txt County_2.txt

Bob@foo/:$ cat /Who_Won/Region_1/Region_1.txt
A:3,B:1,C:1,D:1
```

Executable 3: Vote_Counter

This program is the root node of the whole vote counting application.

Input:

Revision 03/06/18: This program has ~~**zero or**~~ **one** argument and will be executed as follows:

`./Vote_Counter <root_path>`

- `<root_path>`: the relative path from the current directory to the root.

Main Functionality:

- The working directory where the program is executed will be the root of the program.
- This program basically will call the Aggregate_Votes program, read output file of the aggregation result, and **append** the winner into the output file.
- Similarly, child process **should not** write any messages to the standard output.
- This program can also be called from any parts of the DAG.

Output:

- The program has one output: write to the standard output the output file path ended with a new line character.
- This program will alsol append the aggregation result with a winner as follows: "Winner: <candidate_n>\n"

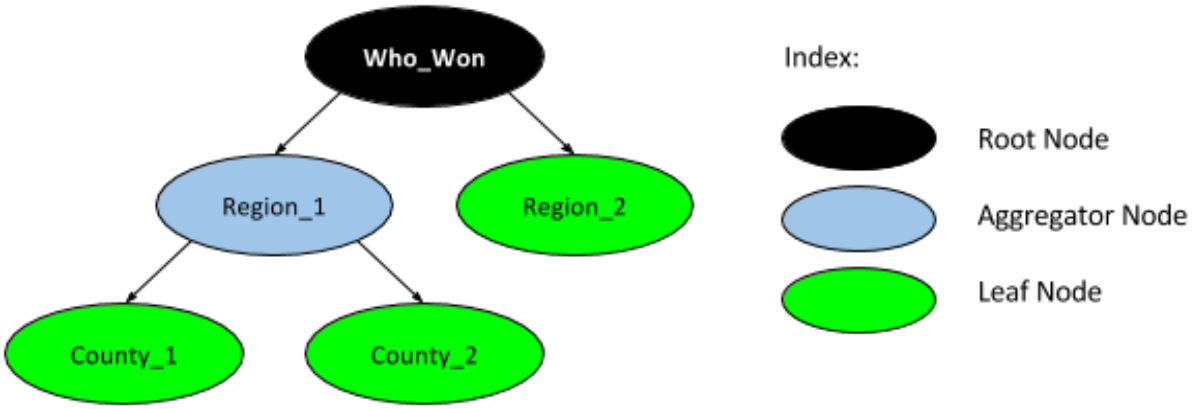
Example: (Look at figure above for the structure of the DAG)

```
Bob@foo:/$ ls Who_Won
Region_1 Region_2
Bob@foo:/$ cat Who_Won/Region_2/votes.txt
E
E
E
E
Bob@foo:/$ ./Vote_Counter Who_Won
Who_Won/Who_Won.txt
Bob@foo:/$ cat Who_Won/Who_Won.txt
A:3,B:1,C:1,D:1,E:4        //Written by the Aggregate_Votes on Who_Won
Winner:E                   //Appended by the Vote_Counter
```

Putting It Together

- Each executable must be able to be executed on its own without executing the Vote_Counter program. For example, we should be able to run Leaf_Counter and Aggregate_Votes independently.
- Both Aggregate_Votes and Vote_Counter can be executed on **any parts** of the DAG. For example, we can run an Aggregate_Votes or Vote_Counter on a leaf node.

Example



1) Calling the Leaf_Counter program on County_1 will count the data from the leaf node and print the result file name into the standard output.
2) Calling the Aggregate_Votes on Who_Won will spawn two processes, one for the Region_2 and one for the Region_1. For the Region_2, the Aggregate_Votes will execute a Leaf_Counter. For the Region_1, it will execute another Aggregate_Votes on Region_1. Then, the Who_Won aggregator will read the output of both children (the file names of the their output), parse the files and write out the aggregation result to the current directory. Lastly, it will print the aggregation result file name into the standard output.
3) Calling the Vote_Counter on Who_Won will execute the Aggregate_Votes on Who_Won and append the winner into the output_file.
4) Calling an Aggregate_Votes on County_1 will execute a Leaf_Counter program on County_1 because County_1 is a leaf node.
5) Calling a Vote_Counter on County_1 will execute the Aggregate_Votes program which will execute the Leaf_Counter program on County_1. Then, the Vote_Counter will append the winner to the output file.

Rules and Hints

Assignment Rules

- You cannot use the library call “system”.
- Every executable must follow the naming convention that is described by the documentation of this assignment, i.e., Leaf_Counter, Aggregate_Votes, and Vote_Counter.
- Both Aggregate_Votes and Vote_Counter executables must be able to be executed from any parts of the DAG.
- In contrast to your Programming Assignment 1, **there is no maximum number of children**. Your program must be dynamically able to handle any number of children.
- You **must not** use “`cd`” and “`ls`”

Useful System Calls and Functions

open, close, read, write, fopen, fgets, fclose, fork, execl, opendir, readdir
Please consult the manpage for the details on how to use them.

Simplifying Assumption

1. There is always a winner in the voting. (No ties).
2. The maximum size of a file name is 255 bytes.
3. The maximum size for each I/O operation message, e.g., each read or write, is 1024 bytes.

Error Handling

- You are expected to check the return value of all system calls that you use in your program and check for error conditions. Also, each 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 from library calls is encouraged.)
- Since we are going to redirect the standard output of some programs, you can try to debug your code by printing into the standard error instead.
- **Your program will be intentionally tested with errors.**

Deliverables

You must submit **single zip file** which contains **at least** the following files:

1. Makefile
2. README
3. At least 3 c files which contain the source code of your Leaf_Counter, Aggregate_Votes, and Vote_Counter

DO NOT include your executables. Your submission must be able to be compiled without additional files.
Revision 02/28/18: Please zip the files directly (instead of zipping the folder that contains your submission files) so that the c files are located in the root of the zip file.

Makefile

Your Makefile should be able to be executed as follows:

1. make Leaf_Counter: creates the Leaf_Counter executable.
2. make Aggregate_Votes: creates the Aggregate_Votes executable.
3. make Vote_Counter: creates the Vote_Counter executable.
4. make: creates all three executables.
5. make clean: remove all generated executables.

README

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
- Your x500 and the x500 of your partner
- Your lecture section and your partner's lecture section
- Your partner's and your individual contributions
- **Specify whether you are doing the extra credit or not.**

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

```
/*login: x500_1, x500_2
  date: mm/dd/yy
  name: full_name1, full_name2
  id: id_for _first_name, id_for_second_name */
```

Grading Rubric (Revision 03/06/18)

This the exact grading rubric:

- 10% Following the Deliverable Instructions (NO PARTIAL CREDIT)
  - All source codes, including .c, .h, and Makefile, are located at the **root of the zip** and **not** in a folder that you zip**.** Please **do not** zip the directory that contains your files, instead, select all your files then zip them directly.
  - All executables follow the correct naming convention
- 5% README and Makefile.
  - The Makefile can be compiled **with GCC 4.9.2**
  - The README is complete
- 15% Documentation within code, coding, and style. (5% each)
- 70% Test Cases.
  - Functionality test cases (55%)
  - Error handling test cases (15%)

Extra Credit (10 Additional Points)

For the extra credit, you will need to make the Vote_Counter program to be aware if the graph has a cycle. We will use symbolic link to create a cycle to the graph. In such case, the Vote_Counter program should work as before, however, it will append an additional information to the output file that specifies where the cycle is located. For example: assume there is a symbolic link from County_1 to Who_Won. Then, your Vote_Counter application should have the following output:

```
Bob@foo:/$ ./Vote_Counter Who_Won
Who_Won/Who_Won.txt

Bob@foo:/$ cat Who_Won/Who_Won.txt
A:3,B:1,C:1,D:1,E:4        //Written by the Aggregate_Votes on Who_Won
Winner:E                   //Appended by the Vote_Counter
There is a cycle from County_1 to Who_won. //Appended by the extra credit.
```

Clarification 03/08/18:

(1) In this extra credit, you will **not** given a **"bad symbolic link"**, i.e., a link that points to some random directories that are not part of the DAG.

(2) The link is also **guaranteed** to link the node's direct ancestors, e.g., its parent, grandparent, etc., and will not point to its sibling. For example, there can be a link from County_1 to Region_1 or Who_Won, however, there will **not** be a link from County_1 to Region_2.

(3) You **cannot** assume that there is only one symbolic link. For example, County_1 directory may have two symbolic links to Region_1 and Who_Won.