CSci 4061 Programming Assignment 3

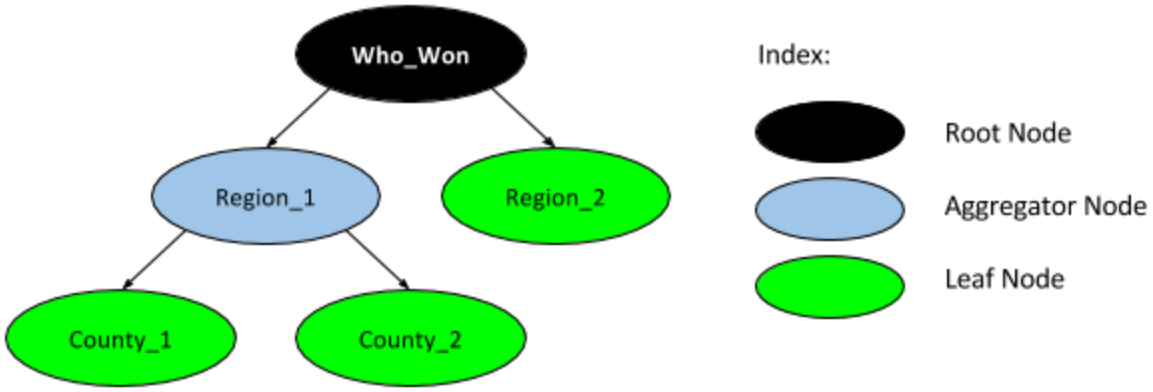
Due: Wednesday, 04/11/2018, 23:59.
Weight: 100 points, 10% of total grade.
Rule: It is **Highly recommended** to do the assignment in pairs.

1. Introduction

The purpose of this programming assignment is to get you started with thread programming and synchronization. You need to be familiar with POSIX threads, e.g., pthread_t, pthread_create, pthread_join, pthread_exit, pthread_self, pthread_mutex_t, pthread_mutex_lock, pthread_mutex_unlock, sem_t, sem_wait, sem_post, sem_destroy, pthread_cond_t, pthread_cond_signal, pthread_cond_wait.

2. The Overview of Our Existing Vote Count Application

Throughout this assignment specification we will consider the DAG example that can be seen in the figure below. Recall that our vote count application has three types of nodes: (1) root node, (2) aggregator node, and (3) leaf node.



Your role is to implement a local program for the vote count. The tool acts as a simple program that can read all leaf nodes' documents, and aggregate the result in the parent node. Forking new processes for each leaf document is not a cheap operation, so multithreading techniques is

much faster. You are going to implement a **votecounter** program as shown in Figure 1. In general, your program should handle the following main tasks:

(1) Reads a DAG text file named DAG.txt, which assist you to build the full graph data structure.
(2) Reads files (i.e., leaf files) from Input-Directory. This directory contains every leaf nodes' files (not sorted in any order). Files are encrypted for security.
(3) Decrypts leaf files from the Input-Directory and saves the decrypted version into the Output-Directory.
(4) Builds the directory structure in the Output-Directory that has the form of graph described in the DAG.txt. The leaf directories contain the decrypted files.
(5) Spawns one or more threads to do the aggregation of intermediate directories, i.e., non-leaf directories, and store the aggregated results files from their child/children. The root directory contains one folder and a log file which lists all your execution steps.

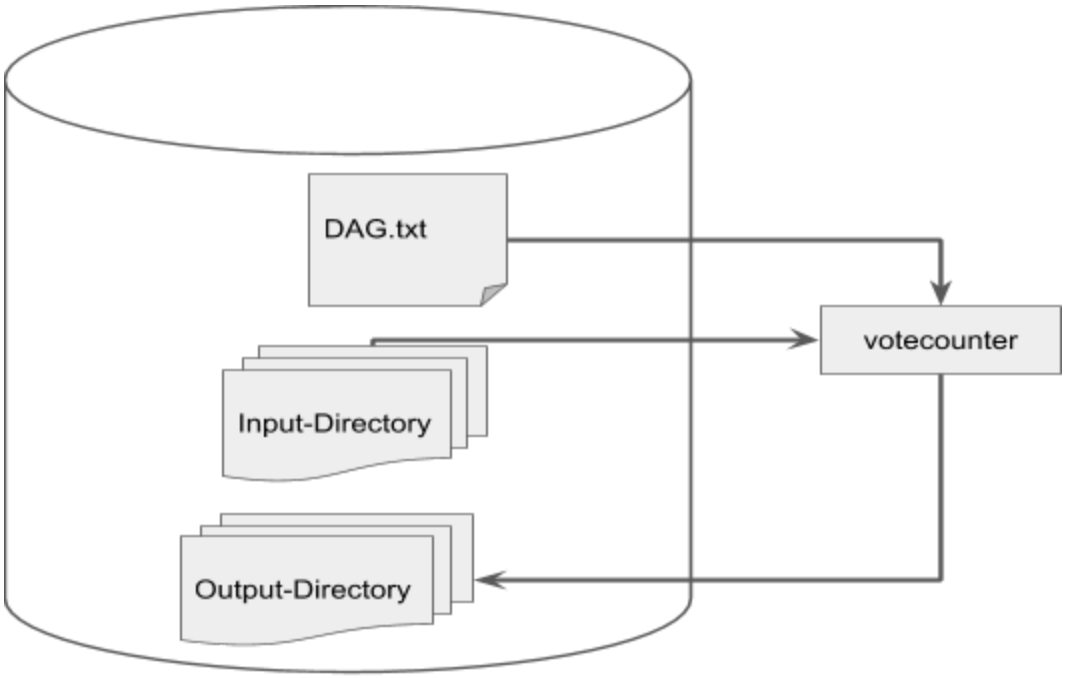


Figure 1: Overview

3. Your Assignment In Detail.

Your task is to implement a program that is called from the terminal as follows.

```
$./votecounter <DAG.txt> <input_dir> <output_dir>
```

```
Program Arguments:

<votecounter>: The executable file MUST be named votecounter (lower
case)

<DAG.txt>: A file that contains the graph.

<input_dir>: Path of the input directory. All leaf nodes' files  will
be stored inside this directory.

<output_dir>: Path of the output directory. All output will be
written inside this directory. You need to create this directory if
it does not exist.
```

The execution flow of your proofread program is summarized below.

Main Thread:

1. Read <DAG.txt> which contains the graph structure.
2. Create the <output_dir> if it doesn't exist. The output directory should have a similar structure of the <DAG.txt>. The output directory will contain a root directory and a log.txt that will be described further in Section 4.3.
3. Initialize a **dynamic shared queue**, which means the size of the queue can expand and shrink by inserting and removing elements, respectively.
4. Read the contents of the <input_dir>. For each file in the input directory, the main thread will insert the file name into a queue. Each file name in the queue is basically a request to be handled by a child thread later**.**
5. Create number threads <thread_num> equal to the number of files in the <input_dir>.
6. Each initialized thread should retrieve a file name from the shared queue and decrypt the file from <input_dir>. (Details below in Child Thread).
7. You **MUST** execute <num_threads> number of threads at once instead of executing one thread at a time.
8. Use a **synchronization mechanism** to manage the access to the shared queue.
9. Wait for all threads to finish before declare the winner and exiting the program.

Child Thread:

1. The child thread should retrieve a file name from the shared queue and remove the file name from the queue.
2. Appends to a log.txt file within the <ouput_dir>. Write the name of the processed file along with its thread id, and status in the format of "file_name:tid:status". Status should be [start]. For example, "County_1:23423:start".
3. Note that many threads will write to the log.txt **simultaneously** so you should **synchronize** your writes.
4. Reads the content of the leaf file from <input_dir>.
5. Invoke a **decryption** function to decrypt the file. (Section 3.1 will describe more details about the implementation of this function).
6. Once a child thread is finished with the decryption, store a decrypted version of the file under the <ouput_dir>. The decrypted file **MUST** be stored in the correct directory based on the DAG. The decrypted file name **MUST** have the same name as the encrypted file and ended with ".txt", e.g., County_1.txt is the decrypted file of County_1 in the <input_dir>.
7. Then, the child thread **aggregates** the results to its parent directory until you reach the root folder inside the <ouptut_dir> . If the aggregated file does not exist, then the child thread must create one. Otherwise, if the aggregated file exist in the parent node, then you are required to reflect updates to that file **recursively** until you reach the root of folder under <output_dir>. The name of the aggregated file **MUST** match the node name, i.e., the folder name. Be aware that every **thread start from the leaf node aggregate the votes, write the result to its parent node, and recursively write the result to all its ancestor nodes until it reach the root node**. (Section 3.2 will describe in details with an example the typical behavior of a child thread) .
8. Note that many threads may try to write their results to the one aggregated file **simultaneously** so you should **synchronize** your writes.
9. Once a child thread is finished with its decryption and aggregation, then the status of the file is appended to <log.txt> file. In particular, write the name of the file processed along with the thread id, and status , such as "file_name:tid:status". Status should be [end]. For example, when a thread finishes from processing County1, then it will append its status to the log file as "County1:23423:end".
10. Repeat steps (1-9) until all files are served. In other words, the shared queue is empty.

3.1 Decryption function

Each character in the file should be replaced by a character which is greater by **two** in terms of **alphabetical order** and we treat capital and lowercase letter independently. Example:

```
Input : YQCCJ
Output: ASEEL
```

Only English alphabets should be decrypted. This includes: [a-z] and [A-Z]. Please note that y, Y and z, Z are special cases which are replaced/encrypted with a, A and b, B respectively. Do not change white space.

3.2 Example.

Given the DAG below that consists of one root node (i.e., Who_Won), two parent nodes (i.e., Who_Won and Region_1), and three leaf nodes (i.e., Region_2, County_1, County_2).

```
$cat graph.txt
Who_Won:Region_1:Region_2
Region_1:County_1:County_2
```

The contents of all leaf nodes inside the <input_dir> are shown below.

$cat input_dir/Region_2	$cat input_dir/County_1	$cat input_dir/County_2
B	A	B
B	A	B
C	B	B
D	C	D

The main thread of your program creates nodes folders inside the output directory <output_dir>. After the folders are created, then the output directory should look like this. Note that leaf folders do not contain any files at this point.

```
$ ls -R ./output_dir/*
./Who_Won:
```

```
Region_1/  Region_2/

./Who_Won/Region_1:
County_1/  County_2/

./Who_Won/Region_1/County_1:

./Who_Won/Region_1/County_2:

./Who_Won/Region_2:
```

Now the main program should create a <numbr_thread> = the number of files inside the input directory, which is **equal 3**. Each thread will start to decrypt the leaf file and place where it belongs inside the output directory. For example, Let's assume three threads created Tid1, Tid2, and Tid3. Each thread is responsible for handling one leaf document, such as Tid1 is responsible for the County_1 file, Tid2 is responsible for County_2 file, and Tid3 is responsible for the Region_2 file. Essentially all threads start at the same time. Each thread decrypts its document, and place in its right position under the output directory. The status of each thread is written in the log.txt file. The output directory is shown below.

```
$ ls -R ./output_dir/*
./Who_Won:
Region_1/  Region_2/ log.txt

./Who_Won/Region_1:
County_1/  County_2/

./Who_Won/Region_1/County_1:
County_1.txt

./Who_Won/Region_1/County_2:
County_2.txt

./Who_Won/Region_2:
Region_2.txt
```

The contents of the files inside the output directory now are decrypted as shown below.

$cat Region_2.txt D D E F	$cat County_1.txt C C D E	$cat County_2.txt D D D F

Once threads finish from decryption files, then they will start to aggregate the votes, and write the aggregated result into their parents until each one of them reaches the root folder. For example, let's assume that both threads Tid1 and Tid2 started to aggregate votes each of its own leaf document. However, Tid1 finished aggregation before Tid2 and started to write the aggregated result into its parent node, i.e., Region_1 folder. First, Tid1 will acquire the lock before it starts writing the result into the Region_1 folder. Since this is the first thread that attempts to write the aggregated result in Region_1, then Tid1 will create the result file under Region_1 as shown below. Note that in the meantime thread Tid2 should wait until Tid1 release the lock.

```
$ ls -R ./output_dir/*
./Who_Won:
Region_1/  Region_2/  log.txt

./Who_Won/Region_1:
County_1/  County_2/ Region_1.txt

./Who_Won/Region_1/County_1:
County_1.txt

./Who_Won/Region_1/County_2:
County_2.txt

./Who_Won/Region_2:
Region_2.txt
```

The contents of the aggregated result at this point after Tid1 finish shown below.

```
$cat ./Who_Won/Region_1/Region_1.txt
C:2
D:1
E:1
```

At this point thread Tid1 releases the lock and thread Tid2 starts to write its result into Region_1. Similarly, thread Tid2 acquire the lock and then start writing the result into Region_1. Thread Tid2 finds that Region_1.txt file already exists. In this case, thread Tid2 should consider aggregating the votes inside Region_1.txt before writing its result. The contents of Region_1.txt shown below after Tid2 finish and release the lock.

```
$cat ./Who_Won/Region_1/Region_1.txt
C:2
D:4
E:1
F:1
```

The same synchronization takes place when Tid1 and Tid3 attempt to write the results in Who_Won directory. Let's assume that Tid3 is the first thread that attempts to write the result under Who_Won. Then this means Tid3 will create the Who_Won.txt file first. Meanwhile, Tid1 will wait until Tid3 release the lock. Below shown the aggregation results in the root directory (i.e., Who_Won) after both Tid3 and Tid1 respectively.

* After Tid3 finish

```
$cat ./Who_Won/Who_Won.txt
D:2
E:1
F:1
WINNER:D
```

* After Tid1 finish

```
$cat ./Who_Won/Who_Won.txt
C:2
D:3
E:2
F:1
WINNER:D
```

Finally, As shown below thread Tid2 will update the results within Who_Won.txt under the root directory.

* After Tid2 finish

```
$cat ./Who_Won/Who_Won.txt
C:2
D:6
E:2
F:2
WINNER:D
```

The final list of files and folders under the output directory shown below. The blue files are the decrypted leaf nodes files, while the read files are the aggregated files. The green file is the log file.

```
$ ls -R ./output_dir/*
./Who_Won:
Region_1/  Region_2/ Who_Won_1.txt log.txt

./Who_Won/Region_1:
County_1/  County_2/ Region_1.txt

./Who_Won/Region_1/County_1:
County_1.txt

./Who_Won/Region_1/County_2:
County_2.txt

./Who_Won/Region_2:
Region_2.txt
```

4. Assumptions and Expectation.

You must exactly follow assumptions and requirements in this section. If your program fail to follow instructions you might affects your grade, as we will have auto testing tools that test your submission with different test cases.

4.1. <DAG.txt>

The file name could be in any name. The content of the file specifies the graph of all nodes that needs to be created. The names of these nodes could be anything in alphanumeric, but containing at least one alphabet character. You should handle reading this file. **There can be any number of nodes and the name of each node can be in any length**. Your program should be dynamic to realize any number of nodes and any length of its name.

```
$cat graph.txt
Who_Won:Region_1:Region_2
Region_1:County_1:County_2
```

4.2. <input_dir>

You will be given a directory with this assignment as the second argument. You can assume that <input_dir> will be placed in the same current directory of your executable. All leaf nodes documents will be inside this directory.

```
$ls  input_dir
County_1
County_2
Region_2
```

Each file in <input_dir> is a leaf node that contains **encrypted** candidate names. Candidate Name could be any length and in any number of words.

(1) Example of a file in some given <input_dir> directory.

```
$cat input/County_1
Y
Z
```

```
A
B
```

4.3. <output_dir>

The <output_dir> should be created if it does not exist. In case the <output_dir> exists then your program should remove this directory with all its contents and create new <output_dir>. All output of your program should be written inside this directory. **Do not change** the name of vote files when your program writes it inside this directory. Create the DAG folders in this directory in terms of Folder and subfolders and so on. When you write aggregate values of vote to a parent node you **MUST** use the same parent node name ended with "**.txt**". Remember the log of your program also **MUST** be inside this directory with strict name **log.txt** (lowercase).

(2) Show list of directory entries of the output directory

```
$ ls output
Who_Won
log.txt
```

(3) Show a file of a leaf node .

```
$cat output/Who_Won/Region_1/County_1/County_1.txt
A
B
C
D
```

(4) Show the files of an intermediate node named Region_1 which contains two children, i.e. County_1 and County_2.

```
$ls  output/Who_Won/Region_1
County_1
County_2
Region_1.txt
```

(5) Show the aggregated file of intermediate nodes. **MUST follow the exact format** <candidateName:Count> without extra spaces or extra character. Each candidate stored in a single line.

```
$cat  output/Who_Won/Region_1/Region_1.txt
A:3
B:5
C:1
D:1
```

(6) Show the result in the root of the DAG node. **MUST follow the exact format** without extra spaces or extra character. For example, the following cases will be considered wrong.
(a) “winner:b”
(b) “WINNER :B” , similarly “WINNER: B”
(c) “WINNER:B.”

```
$cat  ./output/Who_Won/Who_Won.txt
A:12
B:20
C:10
D:1
WINNER:B
```

(7) Show the contents of the log file.

```
$cat  ./Who_Won/log.txt
County_1:111:start
County_2:222:start
County_2:222:end
Region_2:333:start
County_1:111:end
Region_2.:333:end
```

6. Extra Credits (10%)

To get extra credit, you need to implement two parts.

1. Your program should allow additional command line argument <num_threads>. The number of threads Specify the number of **thread pool**. This is the maximum number of threads that the server can handles while processing leaf files concurrently. If this argument is not given, then your program should use a **default value of 4**. The main thread of your program need to Initialize threads based on the given <num_threads> in the arguments.
2. Your program should modify the implementation of the shared queue with a shared priority queue. Recall in your basic implementation you only process files in <input_dir> into a shared queue, and then you have a certain number of threads from the shared queue in a First In First Out fashion. Now you will replace this shared queue with a **priority queue**. The queue should process files in the <input_dir> with higher priority before starting the lower priority. You can get the priority of files from the file name.

The priority of each file is given in the name such that <FileName_p_number> . If a document is not associated with a priority value then your program **MUST** assign it with a **the lowest priority**. The smallest number is the one with the highest priority, e.g., county_1_p_2 has higher priority than Region_2_p_3 . Be aware that file name could have '_' too.

```
$ls  ./input_dir_extra
County_1_p_2
County_2_p_1
Region_2_p_3
```

For simplicity you can consider that the DAG have similar nodes names.

```
$cat graph.txt
Who_Won:Region_1:Region_2_p_3
Region_1:County_1_p_2:County_2_p_1
```

Your task is to modify your program as follow.

```
$./votecounter DAG.txt <input_dir> <output_dir> <num_threads>
```

Program Arguments:

<votecounter>: The executable file **MUST** be named votecounter (**lower case**)

<DAG.txt>: A file that contains the graph.

<input_dir>: Path of the input directory. All leaf nodes' files will be stored inside this directory.

<output_dir>: Path of the output directory. All output will be written inside this directory. You need to create this directory if it does not exist.

<num_threads>: Specify the number of thread pool. This is the maximum number of threads that the server can handles while processing leaf files concurrently. If this argument is not given, then your program should use a **default value of 4.**

Note: Make a note in README if you are implementing this to receive the extra credit. Please see submission guidelines below for more details.

7. Assignment Rules.

1) Program must be written in C.
2) Make sure your program compiles using gcc compiler **GCC 4.9.2** on any of the **CSELab** ubuntu machines. We will use the same development Environment. No credits will be given if program does not compile or run as expected.
3) You are allowed to use **pthread** library **only**.
4) Your program must handle any number of threads.
5) You are allowed to use any **synchronization** mechanism, such as Mutual Exclusion (Mutex) Locks, Condition Variables, and Semaphores.
6) If the expected output does not match the expectation, then No credits will be given. This assignment will be evaluated based on the correctness of your code. **Make sure to follow the format of the output.**
7) If you decide to host your code on umn **github**., then make sure that the repository is **private** and only shared between you and your partner.

8. Error Handling.

1) 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.
2) 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.)
3) If there is a leaf node that does not have a file in the <input_dir>, then your program must not abort, and just **continue the execution normally** by ignoring that leaf.
4) If <input dir> is empty, then your program should print this exact message. **"error: input directory is empty".**
5) If leaf file inside <input_dir> is empty. Your program **should continue execution normally while ignoring the leaf**, without generating any errors.

9. Deliverables

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

1. Makefile
2. README
3. The source code of your submission (*.c and *.h) files.
4. **DO NOT** include your executables.

Makefile

Your Makefile should be able to be executed as follows:

1. Make your program **proofread** executable
2. make clean: remove all generated files including the <output_dir> and excluding the source codes and the <input.txt> and <input_dir>.

README

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

- Your x500 and the x500 of your partner.
- Your lecture section and your partner's lecture section.
- Mention if you implemented the extra credit .

- 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 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.

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

```
/*login: cselabs_login_name
  date: mm/dd/yy
  name: full_name1, full_name2
  id: id_for _first_name, id_for_second_name
   Extra credits [Yes or No]*/
```

10. Grading Rubric

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%)