CSci 4061: Introduction to Operating Systems

Programming project 5 **due: Tuesday December 7th, 2021**

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 11:59pm on Tuesday, December 7th.
Note: Do not publicize this assignment or your answer to the Internet, e.g., public GitHub repo.

Objectives: The main focus of this project is to let students get hands-on experience on synchronization and inter process communication in a server-client setting.

1 Project Description

In this project, you have to write a server and a client program where the server and the client can communicate using a message queue. Essentially, a number of client processes will be created by the client program and each of them will send the server some request using the message queue. The server will create multiple threads to handle the clients (one thread for each client). Each thread will receive requests from corresponding clients using the message queue. The server will maintain a global data structure, where results of all requests of all clients are combined (This is where synchronization comes into play). When all of the clients are done sending requests, the server threads will send the global result to corresponding clients.

1.1 Implementation details

Part 1: Client Program

Functionality The main process of the client program will traverse an input folder and get a list of all text files inside the folder. Then it will divide these text files among the client processes so that each gets almost the same number of files. The client processes will now send the paths of the textfiles (which are assigned to them) to the server using a message queue. The server will receive the paths and generate some statistics (described later) reading from the files and send it to the clients. Each of the clients will write the received result on a file. A detailed step-by step instruction is given below:

Program input: The client program should have two command line arguments:

- A relative or absolute path to a folder (say 'Sample')
- Number of clients (say 'c')

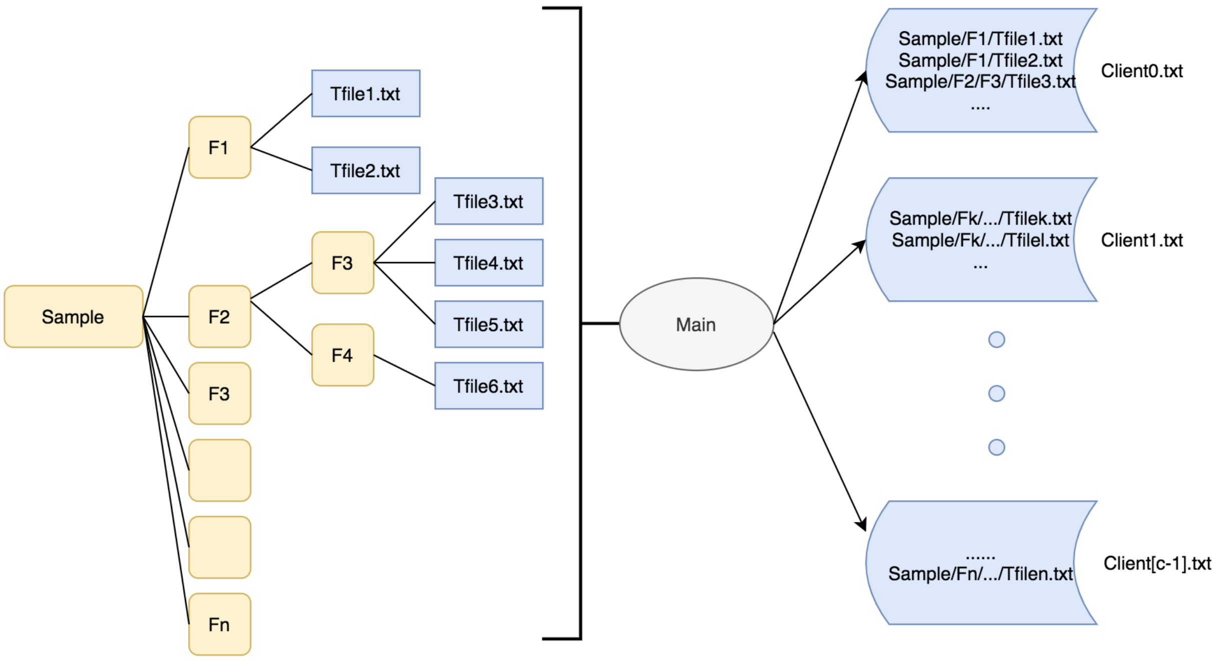

Figure 1: File Partitioning

File traversal: At first, the main process of the client program has to traverse the given folder 'Sample' and identify the paths of the text files which reside inside that folder (if there is any subfolder, then you have to recursively traverse that as well). Now the main process will partition the paths of these text files [not the content of files] equally into 'c' groups so that each group gets the same number of paths. For instance, if there are 50 text files inside the 'Sample' folder and the number of clients ('c') is 5, then divide the paths of the text files into c=5 groups so that each group gets 10 paths.

Next, the main process will create a folder 'ClientInput' in the current directory (client). And then write the list of paths (of text files) for each client i in a text file 'Clienti.txt' (i $\in$ [0,1,2,...,c-1]). The 'Clienti.txt' should be created inside the 'ClientInput' folder.

Note: Assume that the number of files is always greater than or equal to the number of clients except for the case of empty folder. You can also assume that the 'Sample' folder will not be inside the 'server' or 'client' source code folder.
After this part is done, you will have 'c' text files ('Clienti.txt') inside the ClientInput folder.

Generating client processes: The main process of the client program will now generate 'c' number of client processes (Hint: use fork()). Each client process 'i' will read its corresponding 'Clienti.txt' file line by line, present inside the 'ClientInput' folder. The client process will send each of these lines (each line is a path to a text file) to the server using the message queue. After sending each line, the client process will wait to receive an acknowledgement message ("ACK")

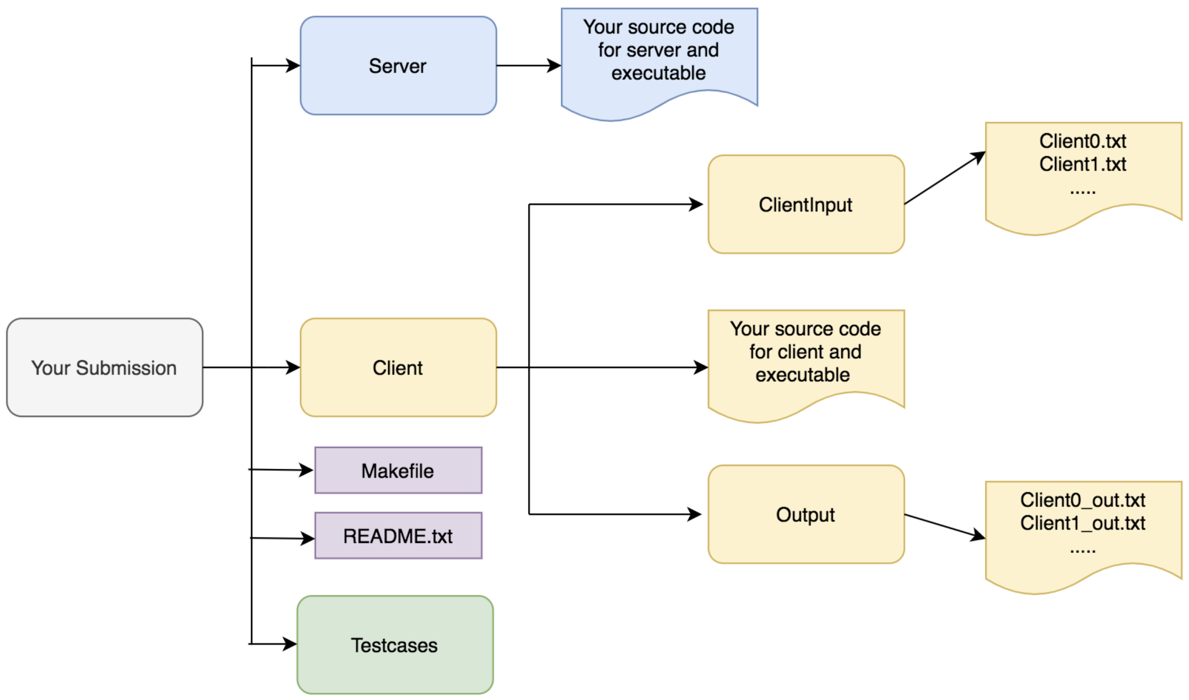

Figure 2: Folder hierarchy

from the server. It will send the next line after receiving acknowledgement for the previous line. Once it receives the acknowledgement for the last line, it will send an end message ("END") to the server. After that the client process will wait for the global result from the server and then will terminate.

Output Format: Each client process should receive a string representing the global word counts like this:
[number of words starting with 'a']#[number of words starting with 'b']# . . . #[number of words starting with 'z']
Example:

50#25#40#. . .#14

Note that the separator between two numbers should be '#'. And this global result should be the same for all client processes. Each client process 'i' should write this string to a text file named 'Clienti_out.txt' (where i=[0,1,2,...,c-1]) inside 'Output' folder. In essence, your final folder structure should follow Figure 2.

The client processes also need to print log statements in the terminal, which is described in section 1.2.

Part 2: Server Program

Functionality: The aim of the server is to maintain a global array which stores the word count for each letter (how many words start with ‘a’, how many words start with ‘b’, etc.). This array will be updated by its threads, where each thread receives some paths to text files from a client process. The threads will modify the array to reflect the number of words starting with a letter in those files. When all the client processes have finished sending up requests, the server threads send the final counts of the global array to the clients. The step-by-step details are as follows:

Program input: The server program should take in one argument, the number of client processes, say (‘c’).

Handling file requests from clients: The server will create ‘c’ threads to handle the requests from ‘c’ clients delivered via the message queue. Message queue is thread safe, so multiple threads can access the queue in a mutually exclusive manner, without explicit use of synchronization tools (mutex, semaphores). Thread ‘i’ will handle requests from client ‘i’. This means you will have to use the message type in the common queue to control the extraction of a request in thread ‘i’. Once a request, which is a file path, is extracted, the thread should open the file and read it line by line. The file consists of a single word per line and you are expected to increment a global integer array (size 26) maintained to keep track of alphabet letter count, based on the first letter of the word. For example, say the file contents are as below:

```
Apple
Orange
Train
Bus
Ants
```

After parsing the above example the integer array will have value as (first row with alphabets just for understanding, the array is 1D):

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
2	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0

Once a file is read completely, the thread creates a message “ACK” and sends it back to the client via the same message queue. Again, think how you should handle the message type to ensure that only corresponding client will extract the “ACK” message.
After completing all the requests from a client, the thread will receive an “END” message from the client. Upon receiving it, the thread will wait for other threads to complete their own processing. Then, it will go ahead and convert the global integer array to a character array. This array is sent back to the client before the thread exits.
Example of final integer array converted to character array:

51#32#68#32#56#78#13#69#13#32#37#26#76#53#12#15#27#38#67#84#25#79#24#78#25#47

<u>Executables:</u>
Your executables should be named as ‘server’ and ‘client’ and should be created inside the ‘server’ and ‘client’ folders respectively.

```
./client input_folder_path num_clients
./server num_clients
```

1.2 Logging

To check the flow of program execution, you need to log statements with timestamps and print on the terminal. You are free to add meaningful statements corresponding to events. The print statements are expected mainly for the following events:
Client:

- Client start
- File traversal and partitioning start
- For each file path send by Process 'i' to server (mention process id and file path)
- For each acknowlegement "ACK" received for a file path from server (mention process id and file path)
- For each "END" send to the server (mention process id)
- For each final result received from the server
- Client ends

Server:

- Server start
- For each file path received by a thread from client (mention thread id and file path)
- For each acknowlegement "ACK" send to the client for a file path (mention thread id and file path)
- For each "END" received from client (mention thread id)
- For global array send by thread 'i' after receiving "END" from client (mention thread id)
- Server ends

Here process id and thread id is not the ids provided by the operating system, but [i=0,1,2...n-1], where n is the number of clients or threads.

1.3 Assumptions

1. 'server' program will run before the 'client' program.
2. Both server and client will be executed on the same machine.
3. The files in the input directory of the client will have a single word per line not exceeding 50 characters and the number of words per file can vary.
4. The number of files present in the input directory to client should exceed the number of client processes unless it is an empty folder.
5. The argument for server program and the second argument for client program will be the same (number of client processes).

2 Deliverables:

One student from each group should upload to Canvas, a zip file containing their C source code files, a makefile, and a README that includes the following details:

1. The purpose of your program
2. How to compile the program
3. What exactly your program does
4. Any assumptions outside this document
5. Team names and x500
6. Your and your partners individual contributions

The README file does not have to be long, but must properly describe the above points. Proper in this case refers to – first-time user can answer the above questions without any confusion. Within your code you should use one or two comments 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 answer 'why', rather than 'how' you implement the said code. At the top of your README file and main C source file please include the following comment:

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

Note: Folders and files inside your submitted zip file should follow the hierarchy structure showed in Figure 2.

3 Grading Rubric

- 5% Correct README contents
- 5% Code quality such as using descriptive variable names and comments
- 10% Error checking and proper logging
- 20% Traversal and File operation
- 25% Message Queue usage
- 25% Synchronization
- 10% Output