Document Version: 1.0 (a new version means an update on the project)
Assignment Date: 2/5/2015
Due Date: 2/19/2014, 11:59pm

Objectives:

- Start writing non-trivial C programs
- Practice with process creation and multi-threading
- Exercise interprocess communication through message queues

1 Project Description

You will design and implement a keyword search server, that will take a single keyword and a directory name from a client and will search the keyword in all the files sitting in a given directory using multiple threads. You will also implement the client program. There may be several client processes requesting service from the server process. The client processes and the server will communicate using message queues. The client program will be named as **`ks_client`** (keyword search client) and it will take two parameters:

1. A keyword that is expressed as a single word, starting with a whitespace character and ending with a whitespace character, not including any whitespace characters in the middle. A whitespace character is either a <SPACE>, <TAB>, or <NEWLINE> character. There will not be any empty keywords specified.
2. A directory name (full path)

Hence a client program can be invoked as follows:

`./ks_client` *keyword dirpath*

A client process that is invoked with those parameters will first attach to the server master message queue identified by a **key** derived from a file *pathname* and a nonzero integer value. The server will use the master message queue to receive requests from the client processes. Since you are the programmer of both the client and the server, you will decide this *pathname* and the integer value that the **`ftok`** function expects to uniquely identify a message queue. For instance, you can use the name of the server program and the integer 1 in both client and server to create a key as follows:

```
ftok("ks_server.c", 1)
```

See the man page of the **`ftok`** for more details on obtaining a key. See also the **`msgget`** function to learn how to get a message queue identifier using the key returned by the **`ftok`** function.

After attaching the server master message queue, the client will generate a request message and include the *keyword* and *dirpath* in that message. The generated message will be sent to the server via this master message queue. The server must have created the master message queue earlier. Therefore, the server must be started earlier than the client.

The name of the server program must be **`ks_server`** (keyword search server). It will take no parameters. As in the client case, a key will be used to create the server master message queue via the **`msgget`** function. Parameters used in **`ftok`** function should be the same in both client and the server to connect to the same message queue. The server will be invoked as follows:

```
./ks_server
```

and it will run silently in the background. The server process will not produce any output to the screen. The output will go via message queues to the client processes. When the server process gets a request from a client process, it will create a child process that will handle the request. The request is just a message that is received via a message queue. The request includes the keyword and the full path of the directory which includes multiple files to be searched. The child process that is created will handle the request. **For every file in the directory, the child process will create a corresponding thread that will handle the search operation in that file.**

The thread will search the given keyword (i.e. will try to exactly match the entire word only) in the file that it is assigned. While searching the file, if the thread encounters the keyword as an entire word in a line, then the thread will record the file name, line number, keyword, and the line itself (excluding the newline character at the end) into a message. That message will be sent to the client process. If the keyword is seen, for example, in 5 separate lines, then 5 separate messages will be created. Received messages by the client will be printed to the screen in the following format:

<filename>:<linenumber>:<keyword>:<linestring>

For example, if the keyword is "*a*" and the content of the file to be searched (assume the file name is *file1.txt*) is as follows:

I raised my daughter in the American
fashion; I gave her freedom, but
taught her never to dishonor her
family. She found a boy friend,
not an Italian. She went to the
movies with him, stayed out late.
Two months ago he took her for a
drive, with another boy friend.
They made her drink whiskey and
then they tried to take advantage
of her. She resisted; she kept her
honor. So they beat her like an
animal. When I went to the hospital
her nose was broken, her jaw was
shattered and held together by
wire, and she could not even weep
because of the pain.

then the client will print the following output to the screen:

file1.txt:4:a:family. She found a boy friend,
file1.txt:7:a:Two months ago he took her for a

If the keyword is seen more than once in a line, then a single message will be created for that line. For example, if we had a line as "a a a", then the number of matches on this line would be 3. However,

we have to report such a line only once (i.e. generate a single message and print the line to the screen only once).

All the threads that are searching their assigned files will do the same thing. Hence, they will return their results to the client process who made that search request using messages via the help of message queue(s) again. For this purpose, you can create separate message queues between the server process and the corresponding client process who made that search request for sending replies. The number of message queues that you will create for this purpose is upto your design. (***TIP:*** *Each process has a unique process ID (PID). The client can create his own reply message queue using its own PID in ftok, and can pass his PID as a message type while sending the search messgae to the server. This way, the server can connect to this specific client's reply message queue to send the search result.*)

The client process will receive the reply messages from the server. When the server finishes searching the whole tree, it will send a special message to the client indicating the end of the search operation. Similarly, a special keyword called *exit* will be used to shut the server down. When the server receives the *exit* keyword, then it will not search that keyword but remove the master message queue and terminate. Similarly, the client will not wait output messages from the server if the *exit* keyword is sent to the server.

The user should be able to start up more than one client at the same time. In this way, several search requests may arrive to the server at the same time and the server has to create several child processes in this case. Each child process will handle one client's request. Handling a request may cause several threads to be created from that child process.

2 Constants and Tips

You can make the following assumptions on the keyword size, directory path size, line size, and the final result size in terms of the number of characters they include:

1. KEYWORDSIZE = 32
2. DIRPATHSIZE = 128
3. LINESIZE = 1024
4. RESULTSIZE = 2048

Also, remember to use thread-safe library functions inside your threads! (for instance **`strtok_r()`** instead of **`strtok()`**).

3 Development

You will develop your programs in a Unix environment using the C programming language. *gcc* will be used as the compiler. You will be provided a Makefile and your program should compile without any errors/warnings using this Makefile. Black-box testing will be applied. Clients' output will be sorted and redirected into a file, then it will be compared with the correct output. A black-box testing script will be provided on the class website, make sure that your program produces the success message in that test. A more complicated test (possibly more then one test) will be applied to grade your program.

4 Submission

Submission will be done through Blackboard strictly following the instructions below. Your work will be penalized 5 points out of 100 if the submission instructions are not followed. Memory leaks and compilation warnings will also be penalized if any (*Tip: make sure you use the* ***-Wall*** *flag of gcc*).

4.1 What to Submit

1. `ks_client.c`: including the source code of the client.
2. `ks_server.c`: including the source code of the server.
3. `README.txt`: including the following information:
   (a) Your name and ID.
   (b) Did you fully implement the project as described? If not, which parts are not implemented at all or not implemented as following the specified description (message queues, child creation, or multi-threading)? **Note that for this project, it is very easy to print out the required output to the screen and pass the black-box test by just using a search function in the client, without using any message queues, child creation, or multi-threading. Implementing the project without following the specified description will be considered as plagiarism if not properly described in this `README.txt` file.**

4.2 How to Submit

1. Create a directory and name it as your UofL ID number. For example, if a student's ID is 1234567, then the name of the directory will be 1234567.
2. Put all the files to be submitted (only the ones asked in the *What to Submit* section above) into this directory.
3. Zip the directory. As a result, you will have the file 1234567.zip.
4. Upload the file 1234567.zip to Blackboard using the "Attach File" option. You do not need to write anything to the "Submission" and "Comments" sections. NO LATE SUBMISSIONS WILL BE GRADED!
5. You are allowed to make multiple attempts of submission but only the latest attempt submitted before the deadline will be graded.

5 Grading

Grading of your program will be done by an automated process. Your programs will also be passed through a copy-checker program which will examine the source codes if they are original or copied. We will also examine your source file(s) manually.

6 Changes

No changes.