Operating Systems – Exercise 4

C/Unix Threads

General Guidelines:

Submission deadline is **Wednesday, May 20th at 23:55**
Submit your answers as **C, H files** and a **Makefile**, packed into a single ZIP file named Ex4-Name1-ID1-Name2-ID2.zip.
Post the ZIP file in the submission page in course website.
Do NOT pack the file as RAR or any other compression other than ZIP.

- No late submission will be accepted!
- You may work on this exercise in pairs
- Place both your names and IDs at the top of every code file (including headers).
- Send only your .c/.h files and a makefile. Do not send eclipse project files, etc.
- Your code must compile and run on Linux Ubuntu 32-bit.
- Your code must compile with no errors or warnings.
- Your code must never crash.
- You must supply a makefile with your code. The makefile must compile your code as instructed in this document. (missing makefile will be punished)

Important: If you have troubles understanding how to use C/UNIX system calls, read the appropriate manuals, see course textbooks and ask in course newsgroup for help.

Part 1 – Writing a Word Counting Service (95 points)

In this part you will implement a word counting service. You will create a program that can run in background and be supplied with file names to count words in - by any other processes in the system, using named pipes mechanism.
The number of words per file will be written to a log file.

In this exercise you will practice the following topics:

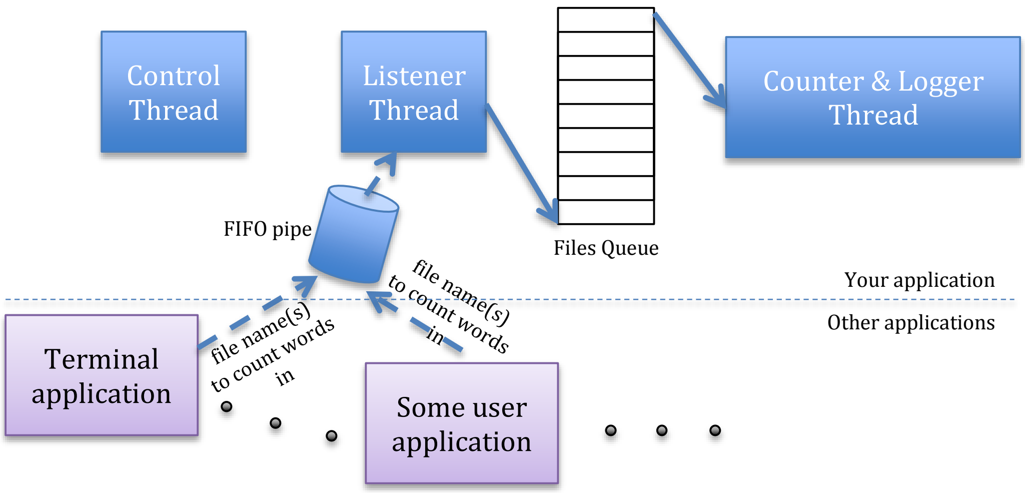
- UNIX file handling
- POSIX threads
- Mutex and condition
- FIFO (Named Pipes) as an Inter-Process Communication (IPC) tool

You can read about these topics in these links:
https://computing.llnl.gov/tutorials/pthreads/ - pthread tutorial
http://beej.us/guide/bgipc/output/html/singlepage/bgipc.html - UNIX IPC tutorial
http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html - another pthread tutorial

This material is also covered in the course textbooks (Tannenbaum; Silberschatz et al.)

A. System overview

We will implement a multi-threaded application with the following elements:



The control thread is the main thread of the process. It creates the files queue and the two other threads – the Listener and the Counter/Logger. Then, it reads user input (which can only be an exit command) and upon receiving an exit command, it shuts down the system (see next sections for more info).

The Listener thread should open a FIFO (named pipe) and wait for input on it. Once some input is received, it should parse it into file names and enqueue these file names to the file queue.

The Counter/Logger thread should dequeue files from the files queue and count words and write them to some specified log location.

B. Task: Implement a Synchronized Bounded Buffer

The files queue described above should be implemented as a thread-safe synchronized bounded buffer. It should support synchronized enqueue / dequeue operations of char pointers, and some other functionalities as described in the attached file `BoundedBuffer.h`.
You should implement the file `BoundedBuffer.c` that implements the functions defined in `BoundedBuffer.h`. You should follow the instructions provided in the header file.

C. Read the File `WordCounter.h`

The attached file `WordCounter.h` defines the functions you should implement in the file `WordCounter.c`.
Read the header file and follow it when reading the rest of the instructions in this exercise.
The file `WordCounter.c` is also attached, with some initial implementation and some helper functions you may use.

D. Task: Implement function `int countwords_in_file(char *file_name)` (in file `WordCounter.c`)

This function should receive the path of the file to read. It should simply read the file using an **`unsigned char`** buffer of size **`FILE_READ_BUFFER_SIZE`** (defined in WordCounter.c).
You are given the is_alphabetic() function, you must use it.
Verify that the function works correctly by supplying it some file to count words in check the results.

E. Task: Implement function `void log_count(WordCounterData *counter_data, char *file_name, int count);` (in file `WordCounter.c`)

This function should receive the counter data from where it should extract the output log file name (given when the program starts), the file name that was just counted, and the word count result.
It will write a log line to the file (log line should include the current time, file counted, and word count).
You can assume that the message being written will never exceed 1024 characters
Remember to append '\n' to the end of your lines so the log file will be readable.

F. Task: Implement function `void *run_listener(void *param)` (in file `WordCounter.c`)

This is the starting point of the Listener thread. You are provided with a skeleton of this function.

The Listener thread is responsible to "listen" to the named pipe file (using the `open(...)` system call) and when a client is connected, the listener should read and parse the data sent by this client (which is file paths to be word-counted).

This function receives two parameters using the **`ListenerData`** struct: files queue and pipe name.

The existing code creates a named pipe with a name as defined in the input parameter. The pipe is created with permission to read/write for everyone (octal 666).

Next, the function enters a loop that opens the pipe file (using `open(...)`) for reading (or "listening").
This call to `open(...)` blocks until some client is writing to the pipe. Once some data is received, it is being read using a buffer (named `filename`).
(You can assume that any chunk of received data is smaller than `READ_BUFF_SIZE`)

The content of the `filename` buffer may contain several file names, separated with a line break ('\n' character). You should create a mechanism that parses each file name, allocates memory for a copy of it (as we have to pass it to the WordCounter thread), copy the file name to the newly allocated buffer, and enqueue it to the files queue.

If the enqueue operation fails (returns 0), it means that the application is trying to exit. In this case, the thread should free the last allocated file name buffer (as it failed to enqueue it), remove the pipe file (using the `remove(...)` system call), and terminate (return NULL).

G. Task: Implement function `void *run_ wordcounter(void *param)` (in file `WordCounter.c`)

This is the starting point of the WordCounter thread.
This function should receive two parameters using the **`WordCounterData`** struct: files queue and destination log file name.
It should have some main loop that dequeues file names from the files queue, one by one.
For each dequeued file it should:

1. Count the words in the file and output the operation result to the screen.
   Use count_words_in_file() for counting.
2. Log the result to the file.
   Use log_count() for writing to file
3. Free the buffer that holds the file name (it was allocated by the Listener thread - this is the place to free it).

If the dequeue operation fails (returns NULL), it means that the application is trying to exit. In this case, the thread should terminate (return NULL).

H. Task: Implement function `int main(int argc, char *argv[])` (in file `WordCounter.c`)

This is the main starting point of the application.

The application is started through command line, as follows:

```
./WordCounter pipe_file_name destination_log_file_name
```

Where:

- `pipe_file_name` is the name of the FIFO pipe file to be used
- `destination_log_file_name` is the destination file name where the log should be written.

The function should:

1. Check its arguments, etc.
2. Create the files queue as a `BoundedBuffer`, initialize it.
3. Create the structs that are used as parameters for the WordCounter and Listener threads, and initialize them with the appropriate values.
4. Create and start the two threads (Listener and WordCounter).
5. Read input (lines) from the user (in a loop). After reading a line, check if it is equal to CMD_EXIT and if it is, call `bounded_buffer_finish(...)` to make the threads stop (possibly only when the next connection arrives). Also, this should make the main thread stop reading user input and go to the next step.
6. Join the threads.
7. Print some exit message.

I. Compile your program, run it and test it

You can test your program by running it, and meanwhile open a terminal window and use the following syntax (output redirection) to write file names into the pipe:

```
echo some_file_name > pipe_file
cat some_text_file_with_a_list_of_files.txt > pipe_file
```

Where:

- `some_file_name` is a name (or a full path) of a file to have its words counted
- `pipe_file` is the name of the pipe file (or the path, if you run the command from a different directory than where the pipe file is)
- `some_text_file_with_a_list_of_files.txt` is some text file that contains many file names to be word counted, one after each other, each one in a new line

Your program should be able to parse and count words in all requested files (unless the requested files are not accessible).

When testing your application, you should make sure that:

- Every requested file is being word counted and logged
- Requests for counting words can be sent one after each other
- Multiple files on one request (separated by a line break) are being word counted
- Files are being counted correctly (we will check for the results, use the supplied is_alphabetic)
- Your code does not crash (no Segmentation Faults / invalid free / etc.)
- Any other idea you may have to make sure your program works correctly

General Guidelines for Your Program:

- Your program must <u>never</u> crash
- Check any input from outside sources (user input, data from other processes, etc.)
- You may <u>not assume</u> any IO/OS operation always succeeds, <u>except</u> for the following calls: malloc, free, fclose, close, pthread_create, pthread_join, mutex/condition functions, printf/screen output functions, mknod.
- In case of an error when counting words in a file, you should simply display some error message in the log and skip the file you are counting (but not exit the whole application).
  In case of a fatal error (one that is not 'recoverable'), you should print some user-friendly error message and make your program exit peacefully.

(It is not very important how you do it, as long as it does not simply crash, and the error messages are explanatory)

- You are not obliged to use the given code or parts of it. However, it is recommended.
- Remember to add the –lpthread flag to your linking gcc call (or to Eclipse: right-click on the project, choose 'Properties', go to 'C/C++ Build' → 'Settings' → Linker → Libraries → Add a library (at the upper box) called 'pthread')

Sample Run:

Modify the attached `list.txt` to contain a list of files to test (for example, use files from previous exercises). Run your word counter and then use `cat` to write the content of this file to the pipe. This will copy the list of file names to the service, if the files are there you should see the word count of each file on the screen and in the log file:

On the screen:

```
vm@vm-ubuntu:~/workspace/OS13Ex6/Debug$ ./OS13Ex6 kuku out.txt
Logging results to file out.txt
Waiting for a connection
Connection received
Waiting for a connection
Counting words for file ../list.txt
File: ../list.txt || Words: 10
Counting words for file ../test.txt
File: ../test.txt || Words: 2
Counting words for file ../tlg3.txt
File: ../tlg3.txt || Words: 101142
Counting words for file ../tlg31.txt
File: ../tlg31.txt || Words: 101142
Counting words for file ../test.txt
File: ../test.txt || Words: 2
exit
Connection received
WordCounter has exited successfully
```

And in the log file:

```
2013-04-27 11:19:24 File: ../list.txt || Number of words: 10
2013-04-27 11:19:24 File: ../test.txt || Number of words: 2
2013-04-27 11:19:24 File: ../tlg3.txt || Number of words: 101142
2013-04-27 11:19:24 File: ../tlg31.txt || Number of words: 101142
2013-04-27 11:19:24 File: ../test.txt || Number of words: 2
```

- The time in the log file was written using the following format: "%Y-%m-%d %H:%M:%S", with the dateprintf() which is provided with the exercise files.

Do not forget the makefile

Part 2 - Makefile (5 points)

Note: missing makefile / make rules will be punished with more than 5 points!

Together with your C files, you are required to submit a makefile with the following rules:

```
all: wordcounter

clean:
	rm wordcounter *.o

wordcounter: WordCounter.o BoundedBuffer.o
	gcc -o wordcounter <..Object files required to be linked, and additional library
flags..>

BoundedBuffer.o: <... all files required by BoundedBuffer.c to compile ...>
	gcc -c BoundedBuffer.c

Wordcounter.o: <... all files required by Wordcounter.c to compile ...>
	gcc -c WordCounter.c
```

(replace the parts in `< ... >` and test your makefile before submitting it!)

To read more on makefiles see these links (or the previous exercise):
http://mrbook.org/tutorials/make/
http://web.mit.edu/gnu/doc/html/make_1.html