CSCI 4210 — Operating Systems
CSCI 6140 — Computer Operating Systems
Homework 3 (document version 1.4)
Multi-threading in C using Pthreads

Overview

- This homework is due by 11:59:59 PM on Friday, November 10, 2017.
- This homework will count as 8% of your final course grade.
- This homework is to be completed **individually**. Do not share your code with anyone else.
- You **must** use C for this homework assignment, and your code **must** successfully compile via `gcc` with absolutely no warning messages when the `-Wall` (i.e., warn all) compiler option is used. We will also use `-Werror`, which will treat all warnings as critical errors.
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v16.04.3 LTS. Note that the `gcc` compiler is version 5.4.0 (`Ubuntu 5.4.0-6ubuntu1~16.04.4`).

Homework Specifications

In this third assignment, the goal is to work with threads and focus on synchronization. You will use C and the POSIX thread (Pthread) library to implement a single-process multi-threaded system that parallelizes the processing of multiple input files.

As with Homework 2, a key requirement here is that you **must** parallelize all processing to the extent possible, carefully following the specifications below.

In brief, your program must first scan a given directory (i.e., `argv[1]`) for all regular files. Each regular file is assigned to a child thread, which is responsible for reading and identifying valid words from the file. For each valid word identified, the child thread must store the word in the fixed-size buffer shared by all threads. When the shared buffer becomes full, the main (i.e., top-level) thread writes the valid words to an output file (i.e., `argv[3]`) and clears the buffer.

The fixed size is specified via the second command-line argument (i.e., `argv[2]`). The main thread dynamically allocates this memory. Further, each child thread is responsible for allocating memory for each individual word. And since there is dynamic memory allocation required, be sure to use `free()` as appropriate.

Command-Line Arguments and Program Execution

As noted above, the first command-line argument (i.e., `argv[1]`) specifies the directory to open. This could be specified as either an absolute or relative path; you should be able to simply plug this into the `opendir()` system call.

The second command-line argument (i.e., `argv[2]`) specifies the size of the fixed-size buffer. More specifically, you should dynamically allocate an array of `char *` of the given size.

The third command-line argument (i.e., `argv[3]`) specifies the output file to write all collected valid words to. Create this file (truncating any existing file if necessary) with the following permissions: `rw-r--r--`.

Below is an example of how you could execute your program and the expected `stdout` output. Here, the example directory contains three regular files `abc.txt`, `def.txt`, and `ghi.txt`, each of which contains exactly eight valid words.

(v1.4) Please note that the output format for the main thread "`Joined`" lines has changed to include a `':'` character before the child thread ID is printed.

```
bash$ ./a.out ../somedir/subdir 5 hw3-output01.txt
MAIN: Dynamically allocated memory to store 5 words
MAIN: Opened "../somedir/subdir" directory
MAIN: Created child thread for "abc.txt"
MAIN: Created child thread for "def.txt"
MAIN: Created child thread for "ghi.txt"
MAIN: Closed "../somedir/subdir" directory
MAIN: Created "hw3-output01.txt" output file
TID 123: Opened "abc.txt"
TID 123: Stored "abc" in shared buffer at index [0]
TID 123: Stored "abc" in shared buffer at index [1]
TID 123: Stored "abc" in shared buffer at index [2]
TID 123: Stored "abc" in shared buffer at index [3]
TID 123: Stored "abc" in shared buffer at index [4]
MAIN: Buffer is full; writing 5 words to output file
TID 123: Stored "abc" in shared buffer at index [0]
TID 123: Stored "abc" in shared buffer at index [1]
TID 123: Stored "abc" in shared buffer at index [2]
TID 123: Closed "abc.txt"; and exiting
MAIN: Joined child thread: 123
TID 456: Opened "def.txt"
TID 456: Stored "def" in shared buffer at index [3]
TID 456: Stored "def" in shared buffer at index [4]
MAIN: Buffer is full; writing 5 words to output file
TID 456: Stored "def" in shared buffer at index [0]
TID 789: Opened "ghi.txt"
TID 789: Stored "ghi" in shared buffer at index [1]
TID 789: Stored "ghi" in shared buffer at index [2]
TID 789: Stored "ghi" in shared buffer at index [3]
```

```
TID 789: Stored "ghi" in shared buffer at index [4]
MAIN: Buffer is full; writing 5 words to output file
TID 789: Stored "ghi" in shared buffer at index [0]
TID 456: Stored "def" in shared buffer at index [1]
TID 456: Stored "def" in shared buffer at index [2]
TID 456: Stored "def" in shared buffer at index [3]
TID 456: Stored "def" in shared buffer at index [4]
MAIN: Buffer is full; writing 5 words to output file
TID 456: Stored "def" in shared buffer at index [0]
TID 456: Closed "def.txt"; and exiting
MAIN: Joined child thread: 456
TID 789: Stored "ghi" in shared buffer at index [1]
TID 789: Stored "ghi" in shared buffer at index [2]
TID 789: Stored "ghi" in shared buffer at index [3]
TID 789: Closed "ghi.txt"; and exiting
MAIN: Joined child thread: 789
MAIN: All threads are done; writing 4 words to output file
```

Note that some interleaving of output is expected here. Further, the index values shown may vary. Match the above output format **exactly as shown above**.

(v1.1) Further, each word that is written to the output file (e.g., `hw3-output01.txt`) by the main thread should be written on a line of its own. From the above example, the `hw3-output01.txt` file would contain:

```
abc
abc
abc
abc
abc
abc
abc
abc
def
def
def
ghi
ghi
ghi
ghi
ghi
def
def
def
def
def
ghi
ghi
ghi
```

Synchronization

All threads share the same memory space. As part of this assignment, you will have to come up with a way to synchronize all of the threads, in particular as each of the threads adds words to the shared buffer. Consider using one or more `mutex` variables to ensure that there are no synchronization errors. Further, another aspect of synchronization is to ensure a specific sequence of events in relation to multiple threads. For this assignment, use `pthread_exit()` and `pthread_join()` to ensure the main thread writes any remaining words to the output file when all threads have terminated.

Starter Code

To ensure that all homework submissions are using the same structure (and therefore facing the same synchronization issues), you are required to use the following `hw3.h` header file, which is also available on the course website. This required header file declares two global variables that your threads must make use of.

```
#ifndef _HW3_H
#define _HW3_H

/* global/shared array of character pointers */
/*  that is used to store all of the words   */
/*   (this should be allocated in main() by  */
/*     using the calloc() system call)       */
char ** words = NULL;

/* global/shared integer specifying the size */
/*  of the words array (from argv[2])        */
int maxwords;

#endif
```

(v1.1) What is a Word?

As with Homework 1, words are defined as containing only alphanumeric characters and being at least two characters in length. Further, words are case sensitive (e.g., "`Lion`" is different than "`lion`"). Consider using `fopen()`, `fscanf()`, `fgetc()`, `isalnum()`, and other such string and character functions here. And be sure to check out the details of each function by reviewing the corresponding `man` pages from the terminal.

Handling Command-Line Argument Errors

Your program must ensure that the correct number of command-line arguments are included. If not, display an error message and usage information exactly as follows on `stderr`:

```
ERROR: Invalid arguments
USAGE: ./a.out <input-directory> <buffer-size> <output-file>
```

Handling System Call Errors

In general, if a system call fails, use `perror()` to display the appropriate error message on `stderr`, then exit the program by returning `EXIT_FAILURE`. If a system or library call does not set the global `errno`, use `fprintf()` instead of `perror()` to write an error message to `stderr`. See the various examples on the course website.

Submission Instructions

To submit your assignment (and also perform final testing of your code), please use Submitty, the homework submission server. The specific URL is on the course website.

Note that this assignment will be available on Submitty a few days before the due date. Please do not ask on Piazza when Submitty will be available, as you should perform adequate testing on your own Ubuntu platform.

That said, to make sure that your program does execute properly everywhere, including Submitty, use the techniques below.

First, as discussed in class (on 9/7), output to standard output (`stdout`) is buffered. To ensure buffered output is properly flushed to a file for grading on Submitty, use `fflush()` after every set of `printf()` statements, as follows:

```
printf( ... );     /* print something out to stdout */
fflush( stdout );  /* make sure that the output is sent to a */
                   /*  redirected output file, if specified  */
```

Second, also discussed in class (on 8/31), use the `DEBUG_MODE` technique to make sure you do not submit any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
    printf( "the value of x is %d\n", x );
    printf( "the value of q is %d\n", q );
    printf( "why is my program crashing here?!" );
    fflush( stdout );
#endif
```

And to compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE homework3.c -pthread
```

Finally, be sure you properly use `pthread_create()`, `pthread_join()`, `pthread_self()`, and `pthread_exit()` calls to implement this homework assignment.