Date due: Monday, May 9, 11:59:59 p.m.

1 Introduction and purpose

In this project you will write a concurrent (multithreaded) program to perform a simple task. The purpose of the project is to get some experience with concurrency using Pthreads. The different lecture examples illustrating Pthreads show all of the features that you will need to write this project. You just have to figure out how to put the pieces together, so the first step will be to carefully study those lecture examples. If you don't understand them perfectly, ask questions about them in the TAs' office hours **before** trying to write code.

This is a smaller project that will not be graded for style. So there is a shorter time for it to be done in.

Important: after the semester is over, be sure to save anything you want to from ELMS, and to clean up your TerpConnect account, as described in Appendix D.

2 Problem to be solved

The project tarfile contains a program `grep.c`. When compiled, it should be run with command line arguments consisting of a word followed by the names of zero or more files, for example `grep.x banana file1 file2 file3`. The program opens each file whose name follows the word, and reads the file's lines. In the process it searches in the file's contents for the word that was the command–line argument. It keeps a running total of the number of lines in which the word was found in the collection of files, and prints the total at the end.

`grep` is actually a very frequently–used UNIX command, which searches for a word or string in files. Our program's behavior is a bit different than that of real `grep`, just to make the project simpler. The actual `grep` command has a number of arguments to control what it does, but in its default mode, instead of printing the count of occurrences of the search string in files, it prints the actual lines of the files that contain the string. As an example to see this, `cd` to your `project08` directory and try commands like these:

```
grep Fakefile fakefile.c
grep Fakefile fakefile.c fakefile-datastructure.c
grep Fakefile *.c *.h
```

Our program just prints the number of lines that the actual `grep` command would print in each case, not the lines themselves. Note that the `-c` option to grep will cause it to behave more like the behavior of our program, in which it prints the number of occurrences of the word in each file. For example, try the command `grep -c Fakefile fakefile.c fakefile-datastructure.c` in your `project08` directory for illustration.

(Note that the actual `grep` command can search for any string of text in files, not just a single word, but if the text to be searched for has characters that have special effects in shell commands, such as `*`, or spaces, the string has to be surrounded by quotes.)

As far as the results it produces are concerned, our program `grep.c` is wonderful and works perfectly. The only problem is that we forgot that we had intended it to be multithreaded. Your task is to rectify this.

In particular, you must **change** the program so it creates **one thread per filename appearing on its command line**, so the number of threads created will equal the number of arguments on the command line that follow the program name and the word to search for. Each thread needs to be passed the word to search for, as well as one of the filenames on the command line. The thread should open the file whose name was passed to it, search in that file for the word, and return the number of lines in that file that contain the word. (If the file does not even exist the thread should return 0.)

The code that is currently in the `main()` function of `grep.c`, for reading the lines of a single file and searching for the argument word in it, must be turned into a function that is invoked by each thread. `main()` will have to create the threads and pass arguments to them (each thread is passed the word to search for and a filename), access the threads' return values, and sum the number of times that each thread found the word in its file (meaning add up the values returned by all the threads). When all the threads have finished, `main()` must print (as it currently does) the total occurrences of the argument word, but now this means the sum of the occurrences that the threads found and returned.

The output of your multithreaded program will be exactly the same as the original single–threaded version of `grep.c` given to you. The only difference is that it will use multiple threads, one for each filename argument.

Your program **must** use multiple threads, one for each filename argument. You could submit the original (single–threaded) program `grep.c` that is given to you, and it would pass all of the tests. But we will be checking during grading whether your program uses multiple threads. If it does not you will get **zero points for the entire project**. (Consequently, due to the minimum requirements policy for projects, you **would not pass the course** unless you were to later submit a version by the end of the semester that passes at least half of the public tests and does use threads. Furthermore, just submitting `grep.c` as your work for this project could be considered an academic integrity violation.)

Note that our original `grep.c` program ignores invalid files (i.e., command–line arguments that are not the names of existing files). Your multithreaded version should do the same.

A Development procedure review

A.1 Obtaining the project files and compiling your program

Log into the Grace machines and use commands similar to those from before:

```
cd ~/216
tar -zxvf ~/216public/project10/project10.tgz
```

This will create a directory `project10` with the necessary files for the project, including our `grep.c` and the public tests. You **must** have your coursework in your course disk space for this class. `cd` to the `project10` directory, **copy** `grep.c` to `grep-threaded.c` using `cp`, and modify `grep-threaded.c` to use threads as described above. (Do not modify the original `grep.c` because the submit server is expecting your program to be in `grep-threaded.c`.)

In most projects this semester you wrote functions that were called from our main programs. Some projects were exceptions, in that your code was the main program. This project is one of those exceptions. Like `grep.c`, your `grep-threaded.c` will be a complete, standalone program (although it will be threaded).

The public test inputs are just text files that your program will be run with as command–line arguments. The public tests are also C programs, but they will **not** be linked together with your program, because your `grep-threaded.x` will be a standalone executable. The tests just run your compiled `grep-threaded.x` program, with different command–line arguments, using the C library `system()` function. For the tests to work you need to compile your program to use the name `grep-threaded.x` for the compiled program.) And keep in mind that when you compile each public test it will not be linked with any other file.

Because `grep-threaded.x` and each public test executable are formed from only one source file you don't have to write a makefile, and you can just compile your `grep-threaded.c` program and the public tests by hand. (You are welcome to write a makefile if you want; it will just be ignored on the submit server.) By now you should know how to use the `gcc` compiler (look in the UNIX tutorial for information if you need to), but don't forget when compiling `grep-threaded.c` to add the `-lpthread` option necessary to compile (actually link) programs using Pthreads.

So to run the public tests you will first have to compile your `grep-threaded.c` to form `grep-threaded.x`, then compile each public test like `public1.c` to form an executable like `public1.x`.

A.2 Running your program, checking your results, and submitting

Each public test like `public1.x` is run as a standalone executable; it does not read any input or take any command–line arguments. Each public test runs your `grep-threaded.x` with different words to search for and filenames as command–line arguments, but the public tests themselves don't have any command–line arguments. (We could have just written the public tests as shell scripts, because they just run your `grep-threaded.x` with different arguments, but we wrote them as C programs for reasons related to the secret tests.)

Some notes about the tests:

- They use the fact, which was not explained in class, that the compiler joins adjacent string literals together.
- Since the public tests will be executable C programs, the `run-tests` script that your TA showed you can run all the tests at once.
- If you want to run the debugger on your program, the comment at the top of each test shows what arguments it runs `grep.threaded.x` with. You can run `gdb` on your `grep.threaded.x` with the same arguments. (Note that you need to run the debugger on `grep.threaded.x` with the filenames shown in the comment– you will **not** be debugging the public tests `public1.x`, `public2.x`, etc., since they just run your program.)

- Your `grep.threaded.x` should always produce the same results as our provided `grep.c` would produce when compiled. If you want to see the results produced by the original `grep.c` on any test you can just make a copy of the test using `cp` and change `./grep-threaded.x` to `./grep.x` in the copy, and compile and run that copy, after compiling `grep.c` to form `grep.x` as an executable. (Make this change in a **copy** of a public test; do not modify the public tests themselves.)
- **Since there is not a required makefile for this project, if you make any changes to your `grep-threaded.c` program, be sure to recompile it to form a new `grep-threaded.x` before running the tests!**

Running `submit` from the project directory will submit your project, but **before** you submit you **must** make sure you have passed all the public tests, by compiling and running them yourself.

A.2.1 Checking whether your program is concurrent

If you run one of the public tests and your program prints the right number as output (`diff` says there are no differences between the test's output and the expected output) then your program must be computing its result properly. But it might not be using threads correctly, which could cause you to lose significant credit during grading, even though your program's output is right. One error that is sometimes made by students without much experience with concurrency is writing programs that only run one thread at a time, which completely defeats the purpose of even using concurrency. There are various ways that things can be done wrong so that only one thread runs at a time, so it is difficult to explain every type of mistake that could be made. But since you would lose **considerable** credit if your program only runs one thread at a time, you should take a few minutes to make sure your program is not doing this.

The easy way to check for this is to run your program under `gdb`, because `gdb` will print messages when each thread begins and finishes. You do not even need to set any breakpoints in `gdb`, just run `grep-threaded.x` with a word to search for and some filenames (for example `grep-threaded.x word file1 file2 file3 file4 file5 file6`). If you get output that looks like the example on the left below, **and** different times you run the program you see different orders of threads starting vs. exiting, your program is running multiple threads concurrently. (the hexadecimal numbers are the thread's ID just in hex, LWP stands for "lightweight process", and the LWP ID is a number that the kernel uses to refer to threads.) However, if you get output like that on the right below, and even when you run the program different times you still always see one thread being created and exiting before the next one is created and exits, etc., then your program is **not** using threads correctly.

```
[New Thread 0x7ffff77f0700 (LWP 11671)]
[New Thread 0x7ffff6fef700 (LWP 11672)]
[New Thread 0x7ffff67ee700 (LWP 11673)]
[Thread 0x7ffff77f0700 (LWP 11671) exited]
[New Thread 0x7ffff5fed700 (LWP 11674)]
[Thread 0x7ffff6fef700 (LWP 11672) exited]
[New Thread 0x7ffff57ec700 (LWP 11675)]
[New Thread 0x7ffff4feb700 (LWP 11676)]
[Thread 0x7ffff67ee700 (LWP 11673) exited]
[Thread 0x7ffff57ec700 (LWP 11675) exited]
[Thread 0x7ffff4feb700 (LWP 11676) exited]
```

```
[New Thread 0x7ffff77b9700 (LWP 20342)]
[Thread 0x7ffff77b9700 (LWP 20342) exited]
[New Thread 0x7ffff77b9700 (LWP 20343)]
[Thread 0x7ffff77b9700 (LWP 20343) exited]
[New Thread 0x7ffff77b9700 (LWP 20344)]
[Thread 0x7ffff77b9700 (LWP 20344) exited]
[New Thread 0x7ffff77b9700 (LWP 20345)]
[Thread 0x7ffff77b9700 (LWP 20345) exited]
[New Thread 0x7ffff77b9700 (LWP 20346)]
[Thread 0x7ffff77b9700 (LWP 20346) exited]
[New Thread 0x7ffff77b9700 (LWP 20347)]
```

By the way, even if you do see that your threads are running and exiting in different arrangements when you run the program different times, this does not guarantee that you are doing everything correctly with concurrency. You could somehow be unnecessarily limiting the concurrent execution of threads inside the code where the thread function is reading from the file. But although this does not guarantee that your code is perfect, it will let you know, before you submit, if you are making several common types of mistakes, so you can fix them.

A.3 Grading criteria

Your grade for this project will be based on:

public tests	60 points
secret tests	40 points

B Project–specific requirements, suggestions, and other notes

- If your `grep-threaded.c` doesn't use multiple threads– one for each file argument– you will **not receive any credit for the project**. The entire purpose of this project is to use concurrency and threads.
- Your threads **must** return values. There are different ways of returning values from threads in Pthreads, but you **cannot just have the threads store their word count occurrences in global variables or other shared variables**. **There must be code in your program after creating and running each thread, when the threads finish, that gets their return values and add them to a cumulative sum, and this must be done outside the threads (after threads finish).** You will **lose significant credit** unless each thread returns the count of occurrences of lines with the word in the file that it is reading.
- Your program must use the **minimum synchronization necessary** to achieve correct results, but **no more synchronization than that**. At the extreme, if you were to only allow one thread at a time to do anything, your program would work fine– but as described you will not have used concurrency. Minimum synchronization just means that the program's threads must be able to run concurrently as much as possible, and a thread should only wait for another one to do something when **absolutely necessary** to ensure correct results (meaning threads should only wait for others in cases where correct results cannot be achieved without that).
- Your program code can **not call `sleep()` anywhere**. Some lecture examples of concurrency used `sleep()` in order to cause the results of small concurrent programs to be more random, so that you could see concurrency working. However, a more realistic concurrent program like this one will exhibit different behavior on its own; you do not need to make this happen artificially. Things have been set up so your program will not even compile on the submit server if it calls `sleep()`.

  Also do **not** use a loop to make a thread wait for another one thread to do something. This is called busy–waiting.
- You can **only** use the Pthreads features that have been covered in class, or you will lose credit.
- When your program creates threads it will have to save their IDs. You don't know in advance (during coding) how many threads your program will have to create, because you can't see the secret tests. Situations where you have to store data and don't know how much data there will be until the program is running, this is where memory allocation must be used. (If your program just creates a giant non–dynamically–allocated array, without allocating memory, you will **lose credit**.)
- All your code **must** be in the file `grep-threaded.c`. No other user–written source (.c) or header files can be added. (Otherwise things probably won't compile on the submit server.)
- The project style guide disallows using (in general) global variables in projects. If you want you may use global variables in this project, but (as stated above) you **cannot use them to "return" values from thread functions, and you cannot use them to keep track of the cumulative count of lines in files that contain occurrences of the search word**. We stress that the project can be written **without using any global variables**. If you are trying to use global variables you are at the minimum making things more difficult for yourself than they need to be, **or** you are violating the requirements above in a way that would cause you to lose significant credit. **Instead** of using global variables, ask for help in the TAs' office hours to see why you don't need to use them.
- If your program has to allocate any memory, you will **lose credit** if you cast the return value of any memory allocation functions. Besides being completely unnecessary, in some cases this can mask certain errors in code.
- Your program should check whether any memory allocations are successful; if any are not, it should print some sort of explanatory message and quit. It doesn't matter how you accomplish this.
- Your program must free any dynamically–allocated memory once it is no longer in use. One of the public tests tests this, so you will fail that test if you are not freeing allocated memory, and secret tests may also test this.
- For this project you will **lose one point** from your final project score for every submission that you make in excess of five submissions. You will also **lose one point** for every submission that does not compile, in excess of two noncompiling submissions. Therefore be sure to compile, run, and test your project's results **before** submitting.

C Academic integrity

Please **carefully read** the academic honesty section of the syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, publicly providing others access to your project code online, or unauthorized use of computer accounts, **will be submitted** to the Office of Student Conduct, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to projects. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. More information is in the course syllabus – please review it now.

The academic integrity requirements also apply to any test data for projects, which must be **your own original work**. Exchanging test data or working together to write test cases is also prohibited.

D End–of–semester TerpConnect account cleanup

The class ELMS space will become inaccessible at some point in the future, so if there's any information you want from it that you don't already have, be sure to download and save it after finals are over. The instructional staff will not be able to provide copies of coursework or assignments in the future, so you need to save copies yourself if you might ever want them later.

A couple weeks after the semester ends you will lose the ability to log in to the Grace systems, although you will still be able to log into `terpconnect.umd.edu` and access your course disk space on that machine. However, early the next semester you will lose permission to access your course disk space, including all your projects, as well as everything in `~/216public`, because the class space will be automatically deleted then by DIT (the Division of Information Technology). And after a year, course projects are also removed from the CMSC submit server. So if you want to save any of your projects or other coursework, or anything that we provided (lecture or discussion examples, secret tests, etc.), you will have to do so yourself over the summer. Sometimes companies want to see examples of class projects during job or internship interviews, and students taking upper–level CMSC courses often want to go back and look at relevant projects and materials from earlier courses. The instructional staff will not be able to provide copies of these in the future, so you'll need to save them yourself if you might ever want them. Recall that the `-r` option to the `cp` command recursively copies a directory and all its contents, including subdirectories, so a command like

```
cp -r ~/216/ ~/216.sp22
```

would copy everything in your extra course disk space (which the symbolic link `216` in your home directory points to) to a directory in your home directory named "`216.sp22`" (use a different name if you like). (Of course you have to have enough free disk space in your TerpConnect account to store the files.) Although you will lose login permission to the Grace systems after the semester (unless you're taking another course using them), you will still be able to log into your TerpConnect account as long as you're associated with the University, via `terpconnect.umd.edu`.

You can also download files to your own computer; if you're using Windows the left pane of MobaXterm will allow you to do this. On a Mac or Linux system you should be able to just open a terminal, cd to where you want to copy the files, and use a command like the following, where *loginID* is your directory ID, and the final period refers to the current directory:

```
scp -r loginID@grace.umd.edu:216 .
```

After this project and the semester are over and you've copied the files you want to save, undo the changes to your account that you made during an early discussion section as described below, since later courses may use the Grace systems, and the changes you made for this course may conflict with changes necessary for them. **However, it may not be possible to provide the secret tests for some of our projects until the early part of the summer, for reasons outside of my control.** (The submit server is using the secret tests and you are getting credit for them, but the tests themselves for some projects may not be able to be put into `~/216public` or visible to you on the submit server until later.) If you want to see the secret tests then **wait** to perform the **rest** of these cleanup steps until **after** the secret tests are available. (As mentioned, even after you lose your login permission for `grace.umd.edu` you will still be able to log into `terpconnect.umd.edu` and see the class files and secret tests there, until the beginning of the next semester.)

Once you have saved everything you want and looked at or copied the secret tests (if you want) when they are available, to undo the changes to your account:

1. Remove our directory from your path in your file `~/.path` by editing it and just removing the line beginning "`setenv PATH`" that you added earlier. (Of course, if you've added any directories of your own to your path then you would not want to remove those.)
2. Remove the symbolic link named `216` that you created from your home directory to your extra disk space, by just removing the symlink, as in `rm ~/216`. You could still reach the files in your extra disk space after that if you wanted to (until you lose access to them) by just using the full pathname, for example, instead of `cd ~/216` you could still use a command like:

   `cd /afs/glue/class/spring2022/cmsc/216/0101/student/`*loginID*
3. Also remove the symbolic link named `216public` that you created in your home directory, which points to the class public directory, after copying any files you want from there, as in `rm ~/216public`
4. Remove the line `source ~/216public/.216settings` that you added to your `~/.cshrc.mine` file.
5. Remove the line (starting with `load`) that was added to your `~/.emacs` file, because it is referring to a file in `216public` that will cease to exist when DIT removes the class files, which can lead to errors.

Make sure you can log out and log back in successfully, and are able to list your files and access things in your directory after that, to ensure that you didn't make any mistakes performing the changes.