**Document Version:** 1.0 (a newer version number means an update on the project)
**Assignment Date:** 2/9/2022, Noon
**Due Date - No Penalty:** 2/23/2022, 11:59pm
**Due Date - 10% Penalty:** 3/2/2022, 11:59pm
**Due Date - 20% Penalty:** 3/9/2022, 11:59pm
**No submission is accepted after 3/9/2022, 11:59pm**
**Group Info:** Individual assignment - no groups

### Objectives:

- Start writing non-trivial C programs
- Practice with thread creation/handling
- Exercise with multi-threaded programming and its performance evaluation

# 1 Project Description

In this project, you will develop an improved version of your project one by adding a keyword search functionality on it. This keyword search will be performed either in a sequential or parallel manner, depending on the related command-line argument being passed to your program. As a result, your program will have a similar output as in the first project, except now you will also include the number of occurrences of the keyword you searched. Your program will be called `pardirlist` and invoked using the following command-line arguments:

```
./pardirlist absolute_path keyword output_file ispar
```

`absolute_path` and `output_file` are the same as in project one, `keyword` is a single word to be searched and `ispar` can be 0 or 1, where, 0 indicates a sequential search and 1 indicates a parallel search.

As a result, your program will print *level*, *order*, *keyword_frequency*, and *absolute_path* for all the files and sub-directories residing in the given directory into the output file, **level by level, where within each level, order is determined alphabetically by the path name**, just like project one. Different than project one, now you also have to include the `keyword_frequency` value for each file being traversed, indicating the number of occurrences of the `keyword` being passed to your program. If it is a folder then the `keyword_frequency` value will be 0.

For example, assume the given directory path is `/home/naltipar/Downloads/final-src` and the content of this directory is:

**/home/naltipar/Downloads/final-src**

```
    README.txt
        int x;
    chap3
        Simulator
            dup.c
                int x;
                int y;
                int z;
        win32-pipe-parent.c
```

```
            int x;
            int y;
    chap4
        thrd-posix.c
            int x;
            int y;
        Driver.java
            int x;
            int y;
        thrd-win32.c
            int x;
            int y;
```

In the example above, the given directory **`final-src`** has two sub-directories **`chap3`** and **`chap4`**, and and a file `README.txt` with the content of:

*int x;*

**`chap3`** has a sub-directory **`Simulator`** and a file `win32-pipe-parent.c` with the content of:

*int x;*
*int y;*

**`Simulator`** has a file `dup.c` with the content of:

*int x;*
*int y;*
*int z;*

**`chap4`** has three files `thrd-posix.c`, `Driver.java`, and `thrd-win32.c`, all three with the same content of:

*int x;*
*int y;*

For the given directory path `/home/naltipar/Downloads/final-src`, when your program is invoked as:

```
./pardirlist /home/naltipar/Downloads/final-src int output.txt 0
```

or

```
./pardirlist /home/naltipar/Downloads/final-src int output.txt 1
```

then the content of the `output.txt` in both cases will be the same as follows:

```
1:1:0:/home/naltipar/Downloads/final-src
2:1:1:/home/naltipar/Downloads/final-src/README.txt
2:2:0:/home/naltipar/Downloads/final-src/chap3
2:3:0:/home/naltipar/Downloads/final-src/chap4
3:1:0:/home/naltipar/Downloads/final-src/chap3/Simulator
3:2:2:/home/naltipar/Downloads/final-src/chap3/win32-pipe-parent.c
3:3:2:/home/naltipar/Downloads/final-src/chap4/Driver.java
3:4:2:/home/naltipar/Downloads/final-src/chap4/thrd-posix.c
3:5:2:/home/naltipar/Downloads/final-src/chap4/thrd-win32.c
4:1:3:/home/naltipar/Downloads/final-src/chap3/Simulator/dup.c
```

To achieve this, your program will traverse the disk structure starting from the given directory path in a recursive manner, and build **a linked list data structure** the same way project one requires. While building this linked list, for every file being encountered, it will start a search process to find the number of occurrences of the given keyword in this file, and record this information inside the node struct representing this specific file. For the sequential version, this search will be performed by the main thread of this program, without creating any additional threads. For the parallel version, a new thread will be created for every file to be searched, and the search will be performed by this new thread while the main thread continues traversing and creating new threads for new files being encountered.

A match in a file will be an **exact match** of the **entire keyword only**, and the maximum length of a line in a file will never exceed 1024 characters. You can use `strtok_r()` to tokenize your file content into words, by making sure that both space and tab are considered as delimiters. Do not use `strtok()` in a multi-threaded program as it is not thread-safe. You can use `strcmp()` to compare tokens with keyword.

As a result, you will end up with a linked list that is sorted level by level, where each level is also sorted alphabetically, as in project one. In addition, every node in this linked list will include the `keyword_frequency` value that you calculated (will be 0 for folders). Finally, you will print this linked list in the specified format (level:order:frequency:path) into the specified output file. For the parallel version, you should make sure that the corresponding search thread is completed before printing its frequency value. For this purpose, you will need to store its thread id in the corresponding node, and perform a `pthread_join()` operation making sure that the thread is competed before it returns. To be able to join a thread, it should be created as joinable by setting its detach state as:

```
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
```

This is the default behavior in many systems but would not hurt to double check. See the man pages of `pthread_create`, `pthread_join`, `pthread_attr_setdetachstate`, etc. in your system to make sure of their behavior.

# 2 Development and Testing

**As in the first project, it is the requirement of this assignment that you have to use the specified data structures (only one linked list, no queues or trees), dynamic memory allocation, and `pthreads`. You cannot assume maximum number of levels, elements, or characters for the**

**paths, but you can assume that the maximum number of characters in each line of a file will be 1024.**

You will develop your program in a Unix environment using the C programming language. You can develop your program using a text editor (emacs, vi, gedit etc.) or an Integrated Development Environment available in Linux. *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 and your program's output will be compared to the correct output. A simple black-box testing script is provided to you for your own test; make sure that your program produces the success message in the provided test. A more complicated test (possibly more then one test) might be applied to grade your program. Submissions not following the specified rules here will be penalized.

The provided black-box test will require you enter your sudo password to be able to clear the cache before each execution so that the execution time of the parallel and the sequential versions can be calculated accurately. A sample output of the black-box test in our system is as follows:

```
Test 1 - Success----------seq-final-src-----------------Success
Elapsed time (seq-final-src): 9 ms

Test 2 - Success----------par-final-src-----------------Success
Elapsed time (par-final-src): 5 ms

Test 3 - Success--------seq-linux-master----------------Success
Elapsed time (seq-linux-master): 3191 ms

Test 4 - Success--------par-linux-master----------------Success
Elapsed time (par-linux-master): 1328 ms
```

In addition to the success/fail messages, you will also be provided the execution time of your program in milliseconds. The above values are obtained using an average laptop, where we expect to see similar values for your program as well. Your sequential program can run a little faster or slower based on how efficient you coded it, but there should be an obvious speed-up for the parallel version of it, especially for the `linux-master` test. Since the `final-src` test is too small, we do not expect to see a consistently high speed-up, but for the `linux-master` test, we expect to see at least 2x speed-up, which is calculated as dividing the execution time of the sequential version by the parallel version. For instance, the `linux-master` speed-up for the above example is: $\frac{3191}{1328} = 2.4$x.

For your reference, the specs of the machine that we used is as follows:

```
System:    Host: carbon Kernel: 4.4.0-210-generic x86_64 (64 bit gcc: 5.4.0)
           Desktop: Unity 7.4.5 (Gtk 3.20.8-1ubuntu0~ppa1) Distro: Ubuntu 16.04 xenial
Machine:   System: LENOVO (portable) product: 20A7002QUS v: ThinkPad X1 Carbon 2nd
           Mobo: LENOVO model: 20A7002QUS v: SDK0E50510 Pro Bios: LENOVO v: GRET32WW (1.09 ) date: 02/13/2014
CPU:       Dual core Intel Core i7-4600U (-HT-MCP-) cache: 4096 KB
           flags: (lm nx sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx) bmips: 10775
           clock speeds: max: 3300 MHz 1: 1844 MHz 2: 1610 MHz 3: 1722 MHz 4: 1770 MHz
Graphics:  Card: Intel Haswell-ULT Integrated Graphics Controller bus-ID: 00:02.0
           Display Server: X.Org 1.18.4 drivers: intel (unloaded: fbdev,vesa) Resolution: 1600x900@60.00hz
           GLX Renderer: Mesa DRI Intel Haswell Mobile GLX Version: 3.0 Mesa 18.0.5 Direct Rendering: Yes
Audio:     Card-1 Intel 8 Series HD Audio Controller driver: snd_hda_intel bus-ID: 00:1b.0
           Card-2 Intel Haswell-ULT HD Audio Controller driver: snd_hda_intel bus-ID: 00:03.0
           Sound: Advanced Linux Sound Architecture v: k4.4.0-210-generic
Network:   Card-1: Intel Ethernet Connection I218-LM driver: e1000e v: 3.2.6-k port: 3080 bus-ID: 00:19.0
```

```
           IF: eth0 state: down mac: 54:ee:75:10:41:db
           Card-2: Intel Wireless 7260 driver: iwlwifi bus-ID: 03:00.0
           IF: wlan0 state: up mac: e8:2a:ea:24:15:c6
Drives:    HDD Total Size: 256.1GB (28.4% used) ID-1: /dev/sda model: LITEONIT_LGT size: 256.1GB temp: 0C
Partition: ID-1: / size: 112G used: 68G (65%) fs: ext4 dev: /dev/sda5
RAID:      No RAID devices: /proc/mdstat, md_mod kernel module present
Sensors:   System Temperatures: cpu: 53.0C mobo: N/A
           Fan Speeds (in rpm): cpu: 2530
Info:      Processes: 331 Uptime: 6 days Memory: 4590.8/7674.9MB Init: systemd runlevel: 5 Gcc sys: 5.4.0
           Client: Shell (bash 4.3.481) inxi: 2.2.35
```

This is obtained by running the following command in terminal:

`inxi -Fx`

In the README.txt file, we will ask you to provide a similar output for your own machine, together with some performance values and analysis. You might need to install `inxi` for this purpose using the following command:

`sudo apt-get install inxi`

# 3 Checking Memory Leaks

You will need to make dynamic memory allocation. If you do not deallocate the memory that you allocated previously using **`free()`**, it means that your program has memory leaks. To receive full credit, your program should be memory-leak free. You can use `valgrind` to check the memory-leaks in your program. `valgrind` will output:

"All heap blocks were freed - no leaks are possible"

if your program is memory-leak free.

# 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. In addition, memory leaks will be penalized with a total of 5 points without depending on the amount of the leak. Similarly, compilation warnings will also be penalized with a total of 5 points without depending on the type or the number of warnings. You can check the compilation warnings using the **-Wall** flag of gcc.

## 4.1 What to Submit

1. `README.txt`: It should include your name, ID, and the average speed-up that you achieved for the `linux-master` test. You will calculate this average speed-up by running the provided black-box test 10 times, calculating the speed-up of each run as described above, and finding the average of these 10 speed-up values. **You are only required to provide average speed-up for the `linux-master` test.** In addition, you will provide the specs of your machine using the `inxi -Fx` command, and include a paragraph explaining if your speed-up values make sense or not compared to the ones provided in this document, by also commenting on the specs of both machines.

2. `pardirlist.c`: Source code of your program.
3. `Makefile`: Makefile used to compile your program. If different than the provided one, then you should inform the TA about your changes.

### 4.2 How to Submit

1. Create a directory and name it as the project number combined with your UofL ID number. For example, if a student's ID is 1234567, then the name of the directory will be ProjectX-1234567 for project X. If it is a group project, then use only one of the group member's ID and we will get the other student's ID from the README file.
2. Put all the files to be submitted (only the ones asked in the *What to Submit* section above) inside this directory.
3. Zip the directory. As a result, you will have the file ProjectX-1234567.zip.
4. Upload the file ProjectX-1234567.zip to Blackboard using the "Attach File" option. You do not need to write anything in the "Submission" and "Comments" sections.
5. **Your project will be graded only once!** If you make multiple attempts of submission before the deadline, then only your latest attempt will be graded. If your submission is late, then it will be graded immediately and you won't be allowed to resubmit once your project is 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 to check if you followed the specified implementation rules, if any.

## 6 Changes

- No changes.