COP290
Semester 2 2022-23
Lab 2 parts 1, 2, and 3

Submission Instructions

1. You can only use C for this Lab. You are to strictly adhere to the function signatures provided in the code template. Changes to the function signatures or data structures will lead to zero marks in the lab.
2. You will submit the source code in zip format to Moodle (Lab 2). The naming convention of the zip file should be <Entry_Number1>_<Entry_Number2>_<Entry_Number3>.zip .
3. There should be only one submission per team on moodle
4. You are allowed to form a group of not more than 3 members.
5. The Lab would be auto-graded. Therefore, follow the same naming conventions. Failing to adhere to these conventions will lead to zero marks in the Lab.
6. You should write the code without taking help from your peers except for group members or referring to online resources except for documentation. Not doing any of these will be considered a breach of the honor code, and would be severely penalised
7. You can use Piazza for any queries related to the Lab.
8. The code template is provided in Moodle
9. Kindly adhere to the directory structure

Note: Students are expected to follow the honour code guidelines. Any impermissible collaboration will be severely penalised.

References to get started with Git: https://www.coursera.org/learn/introduction-git-github This is a course offered by Google on Introduction to Git and Github. To start the course, click on “Enroll For Free” -> “Audit the Course”.

Part 1: Storing/restoring contexts

Learning objective: Stack, registers, gdb.
Suggested completion: 23rd January

Problem statement: We would like to save and restore “contexts”. The idea is to store a context (i.e, CPU registers) inside the conjecture method and when an assertion fails, restore the stored context and continue the process.

Implement the following functions in the template provided:

```
1. static void conjecture(int len, void* options, int sz, void fn(void*));
2. void assert(bool b);
```

About the code:

The main function has an array of numbers. Now it checks if the given number is less than 40 and not a prime number and returns the square of that number
Example: Input → {11, 23, 49, 85, 25}
Output → 625

Instructions:
You only have to modify `conjecture()` and `assert()` functions. Context switching and restoring is to be done using `ucontext`.

References: https://man7.org/linux/man-pages/man3/makecontext.3.html

Part 2: Concurrency

Learning objective: concurrency, heap, race conditions, cooperative scheduling, locking.

Problem Statement: Given N text files, you need to find the count of every word summed across all the files. This must be accomplished using "multithreading" and all the results should be stored in a hashmap.

Checkpoint 1: hashmap
Suggested completion: 30th January
Create a HashMap from scratch in C that supports all the basic hashmap operations like:

1. `void* hashmap_get(struct hashmap_s *const hashmap, const char* key);` → Fetch value of a key from hashmap
2. `int hashmap_put(struct hashmap_s *const hashmap, const char* key, void* data);` -> Set value of the key as data in hashmap. You can use any method to resolve conflicts. Also, write your own hashing function

Find the word count of all the files and store them in a hashmap using only a single thread.

Checkpoint 2: threads
Suggested completion: 5th February
Create a thread library from scratch using `ucontext` that supports basic thread operations like:

1. `void* mythread_create(void func(void*), void* arg);` → A function that creates a new thread.
2. `void mythread_join();` → Waits for other thread to complete. It is used in the case of dependent threads.

Now using your custom thread library and hashmap solve the problem and check the correctness of your result by comparing it with single thread results.

Checkpoint 3: locks
Suggested completion: 12th February

Now implement `void mythread_yield()` that switches the currently running thread to another thread. Then, in the `readFile()` function of the hashmap call `mythread_yield()` between `hashmap_get()` and `hashmap_put()`. Now re-run your code and answer the following questions in your report:

1. Does the output vary?
2. Why do you think the output varies?

Implement these two functions in order to resolve the race conditions
`void lock_acquire(struct lock* lk)` → Acquires lock. Yields if the lock is acquired by some other thread.
`int lock_release(struct lock* lk)` → Releases lock

Part 3: pThreads

Learning objective: Report writing (LaTex), Documentation (doxygen)
Suggested completion: 19th February

Problem statement: Now take the code from part 2 and place your files inside part 3 folder. Ensure that you are using `mythread.c` provided by us rather than your own `mythread.c` file.

- Do a comparative analysis between the runtimes of part 2 and part 3. What happens when you change the number of input files? Size of input files? What if each file has the same word repeated many times?
- Generate a documentation for your code using doxygen
- Write a report using LaTex where you would plot graphs (Time vs File Size) and also generate tables for the same.

You can download large text files for benchmarking from here:
https://www.gutenberg.org/browse/scores/top

Also mention in your report why you think differences in speeds are observed?

Submission guideline

Code templates for both questions are provided. Don't change the data structures and function signatures provided in the template. You are free to write helper functions, but autograder will only call the functions specified in the template.

In the final report, please also add a table that distributes 30 tokens among the 3 team members. These tokens describe the relative effor made by each team member. Please also describe which team member worked on what aspect.

Directory Structure for your submission
<Entry_Number1>_<Entry_Number2>_<Entry_Number3>

- → Part1
  - → include (contains the header files)
  - → src (contains the source code)
  - → data (contains data)
  - → obj (contains object files)
  - → makefile
- → Part2
  - → include (contains the header files)
  - → src (contains the source code)
  - → data (contains data)
  - → obj (contains object files)
  - → makefile
- → Part3
  - → include (contains the header files)
  - → src (contains the source code)
  - → data (contains data)
  - → obj (contains object files)
  - → doc (contains documentation generated by doxygen)
  - → makefile
  - → doxyfile
  - → tex/

Do note the header files are provided you need to write source code for the header files. Overall deadline: ***19th February, 11:59 P.M.***

Rubrics (40 marks)

Part 1: 4 marks
Part 2 CheckPoint 1: 3 marks
Part 2 CheckPoint 2: 6 marks
Part 2 CheckPoint 3: 5 marks
Part 3: 7 marks
Exam: 15 marks