CP386: Assignment 2 – Spring 2022

Due on June 22, 2022 (Before 11:59 PM)

This is a group (of two) assignment, and we will try to practice the concept of the parent-child process, inter-process communication, and some related system calls.

General Instructions:

- Submit your source code files for each question file after adding .txt as its extension and name it as a combination of your Laurier ID number & your teammate's ID, an underscore, 'a' (for 'assignment'), and the assignment number in two digits, and question name. For example, if the students 100131001 and 100131233 are submitting Assignment 2 and question 1's, it is named `100131001_100131233_a02_QUESTION_NAME.c.txt`.
- For this assignment, you must use C99 language syntax. Your code must compile using make without errors. You will be provided with a makefile and instructions on using it.
- Test your program thoroughly with the **GCC** compiler (version 5.4.0) in a Linux environment.
- If your code does not compile, then you will score zero. Therefore, make sure that you have removed all syntax errors from your code.
- Please note that the submitted code will be checked for plagiarism. By submitting this zip file, you would confirm that you have not received unauthorized assistance in preparing the assignment. You also confirm that you are aware of course policies for submitted work.
- Marks will be deducted from any questions where these requirements are not met.
- Multiple attempts will be allowed, but only your last submission before the deadline will be graded. We reserve the right to take off points for not following directions.

Question 1

Write a multithreaded C program, wherein the main function, you need to create a thread descriptor variable and method. The program should tell you whether a thread is created successfully or not. The thread function executes if the thread is successful; otherwise, you exit the program.

- In the custom thread function, a for loop is created, which iterates 5 times [0–4] and prints a statement ("I am a Custom Thread Function Created By user."). It sleeps for one second after the iteration of every print statement. The sleep(…) function is available in the “unistd.h” library.
- The main function also creates a loop that iterates 5 times [0–4] and prints a statement ("I am the process thread created by the compiler by default."). It also sleeps for one second after the iteration of every print statement.

Do you witness any ambiguity in the order of the printed statements of the output of the program (maybe you have to run the program multiple times)? If yes, state it in the comment section of the program file. The expected output for this question is:

```
This program would create threads
I am a Custom Thread Function Created By user.
Custom thread created Successfully
I am the process thread created by compiler By default.
I am a Custom Thread Function Created By user.
I am the process thread created by compiler By default.
I am a Custom Thread Function Created By user.
I am the process thread created by compiler By default.
I am a Custom Thread Function Created By user.
I am the process thread created by compiler By default.
I am the process thread created by compiler By default.
I am a Custom Thread Function Created By user.
```

Note: When submitting source code file for this question, name it like:

- `FirstID_SecondID_threads.c.txt`

Question 2

Write a C program to validate the solution to the Sudoku puzzle. A Sudoku puzzle uses a 9 × 9 grid in which each column and row, as well as each of the nine 3×3 sub-grids, must contain all the digits 1 ··· 9. Figure below presents an example of a valid 9 x 9 Sudoku puzzle solution. This program implements a multithreaded application that reads a Sudoku puzzle solution from a file ***("sample1_in.txt")*** and validates it.

2	7	6	3	1	4	9	5	8
8	5	4	9	6	2	7	1	3
9	1	3	8	7	5	2	6	4
4	6	8	1	2	7	3	9	5
5	9	7	4	3	8	6	2	1
1	3	2	5	9	6	4	8	7
3	2	5	7	8	9	1	4	6
6	4	1	2	5	3	8	7	9
7	8	9	6	4	1	5	3	2

There are several different ways of multi-threading this application (students are encouraged to explore). One suggested strategy is to create threads that check the following criteria:

- A thread to check that each column contains the digits 1 through 9
- A thread to check that each row contains the digits 1 through 9
- Nine threads to check that each of the 3 × 3 sub-grids contains the digits 1 through 9

This would result in a total of eleven separate threads for validating a Sudoku puzzle. However, you are welcome to create even more threads for this problem. For example, rather than creating one thread that checks all nine columns, you could create nine separate threads and have each of them check one column.

1. Passing Parameters to Each Thread: The parent thread will create the worker threads, passing each worker the location that it must check in the Sudoku grid. This step will require passing several parameters to each thread. The easiest approach is to create a data structure using a struct. For example, a structure to pass the row and column where a thread must begin validating would appear as follows:

```
/* structure for passing data to threads */
        typedef struct
        {
            int row;
            int column;
        } parameters;
```

Pthreads will create worker threads using a strategy similar to that shown below:

```
parameters *data = (parameters *) malloc(sizeof(parameters));
data->row = 1;
data->column = 1;
/* Now create the thread passing it data as a parameter */
```

The data pointer will be passed to the `pthread's create()` (Pthreads) function, which in turn will pass it as a parameter to the function that is to run as a separate thread.

2. Returning Results to the Parent Thread: Each worker thread is assigned the task of determining the validity of a particular region of the Sudoku puzzle. Once a worker has performed this check, it must pass its results back to the parent. One good way to handle this is to create an array of integer values that is visible to each thread. The i^{th} index in this array corresponds to the i^{th} worker thread. If a worker sets its corresponding value to 1, it is indicating that its region of the Sudoku puzzle is valid. A value of 0 indicates otherwise. When all worker threads have completed, the parent thread checks each entry in the result array to determine if the Sudoku puzzle is valid.

The expected output for this question is:

```
Sudoku Puzzle input is:
2 7 6 3 1 4 9 5 8
8 5 4 9 6 2 7 1 3
9 1 3 8 7 5 2 6 4
4 6 8 1 2 7 3 9 5
5 9 7 4 3 8 6 2 1
1 3 2 5 9 6 4 8 7
3 2 5 7 8 9 1 4 6
6 4 1 2 5 3 8 7 9
7 8 9 6 4 1 5 3 2
Sudoku puzzle is valid
```

Note: When submitting source code file for this question, name it like:

- `FirstID_SecondID_sudoku.c.txt`

Question 3

Write a C program that will create multiple threads at different times. The threads may have different start_time and lifetime. Once all the threads are over, the process will terminate. The program should perform the following tasks:

- Read the input file that contains thread properties: ID, start time and lifetime.
- Create a thread whenever it is the start time of some thread.
- All the threads implement/run the same function. Implement this function so that the thread should terminate when it has consumed time units equal to its lifetime.

For this question, a code skeleton code has been shared with you (in the file `Question3_Code_Skeleton.c`). Some functions are completely implemented, some are partially, and for some, only header is provided. Additionally, you can write your functions if required. Complete the program as per the following details to have functionality as described above. Write all the code in a single C file for this question:

3. The program has its local clock mechanism. Program must invoke `startClock()` when it is ready to service threads. The file reading should be performed before this. Once the `startClock()` is invoked, the `getCurrentTime()` can be invoked to see how much time units are passed since `startClock()` was invoked. This will provide a local clock ticks behaviour to service your threads.
4. The thread information will be provided to this program in a text file. Program must accept the name of this file as a command-line argument. In this file, there will be one thread information per line. Each thread information has three attributes provided in order: `thread_id; start_time; lifetime`. It is not necessary that the individual thread lines are in any specific order. The `start_time` indicates that at what time a thread should be created and started. The `lifetime` is the time units for which a thread must stay alive. Use the methods in step 1 in a suitable way to implement this functionality.
5. To read the file, `readFile()` function is provided. However, it has a small part missing that you must complete so that the thread attributes can be suitably stored in memory. This also requires the completion of `struct thread`. Add suitable members to `struct thread` and the store the content of the file in a suitable data structure.

6. Once you have thread information in memory, start servicing the threads using the suitable POSIX library calls. The `main()` function is provided but you have to complete it suitably. The program will keep running if there are some threads which are not yet created or if there are some running threads.
7. The `threadRun()` is the function that each thread must run. Implement this function so that each thread takes care of its `lifetime` and terminates when that is over.
8. The image below shows the expected output for the sample input file provided:

```
[1] New Thread with ID t10 is started.
[2] New Thread with ID t2 is started.
[5] New Thread with ID t1 is started.
[9] Thread with ID t2 is finished.
[10] Thread with ID t1 is finished.
[16] Thread with ID t10 is finished.
[20] New Thread with ID t22 is started.
[25] Thread with ID t22 is finished.
```

Note: When submitting source code file for this question,name it like:

- `FirstID_SecondID_thread_timing.c.txt`