CP386: Assignment 3 – Spring 2022

Due on July 6, 2022

Before 11:59 PM

This is a group assignment. In this assignment we will try to practice the concept of synchronization with semaphores, deadlocks, and starvation. The assignment extends the concepts of multithreading used in A02.

General Instructions:

- Submit a PDF file question 1 and source code file for question 2, 3 and name these as Question_1.pdf, Question_2.c.txt, Question_3.c.txt.
- For this assignment, you must use C99 language syntax. Your code must compile using make **without errors.** There will be a small bonus if it compiles without warnings. You will be provided with a make file and instructions on how to use 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 get zero**. Therefore, make sure that you have removed all syntax errors from your code.
- Please note that the submitted code would be checked for plagiarism and by submitting this file you would confirm that you have not received any unauthorized assistance in the preparation of 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.

Question 1

In dining philosopher synchronization problem if each of the five philosophers, p[i], executes the following code:

```
while(1){
    think
    wait(mutex)
       wait(f[i])
       wait(f[i+1 mod 5])
    signal(mutex)
    eat
    signal(f[i])
    signal(f[i+1 mod 5])
}
```

Does this code solution satisfy all requirements of the dining philosophers' problem? Describe the scenarios when the concurrency requirement is possible and when it is violated?

Question 2

1. Write a C program to create a function "inc_dec()" that acts like a critical section. Three threads execute this function and trying to access and update the global data of "a" and "b" variables. To avoid race condition and inconsistent results, apply and implement an appropriate locking mechanism (using a semaphore and a mutex) for global data. The image below shows the expected output:

```
Read value of 'a' global variable is: 5
Read value of 'b' global variable is: 7
Updated value of 'a' variable is: 6
Updated value of 'b' variable is: 6
Read value of 'a' global variable is: 6
Read value of 'b' global variable is: 6
Updated value of 'a' variable is: 7
Updated value of 'b' variable is: 5
Read value of 'a' global variable is: 7
Read value of 'b' global variable is: 5
Updated value of 'a' variable is: 8
Updated value of 'b' variable is: 4
```

Question 3

In this question, a process will create multiple threads at different times, as in Question 3 of assignment 2. These threads may have different `start_time` but there is no lifetime. Each thread after its creation runs a small critical section and then terminates. All threads perform same action/code. Most of the code such as reading the input file, creating the threads etc. is provided. Your task is to implement following synchronization logic with the help of POSIX semaphores:

- Only one thread can be in its critical section at any time in this process.
- The first thread, in terms of creation time, enters first in its critical section.
- After those threads are permitted to perform their critical section based on their ID.
    - Threads are given IDs in the format `txy` where `x` and `y` are digits (0-9). Thread IDs are unique. Threads may have same or different start_time. Thread entries in the input file can be in any order.
    - The "`y`" in thread IDs thus will either be an even digit or odd digit.
    - After the first thread, the next thread that will be permitted to perform its critical section must be the one in which "`y`" is different i.e. if "`y`" was even in first thread then in the next it must be odd or vice versa.
    - For the rest of the process, you must follow the same scheme i.e. two threads with odd "`y`" or even "`y`" can not perform critical section simultaneously.
- Since synchronization may lead to deadlock or starvation, you must make sure that your solution is deadlock free i.e. your program must terminate successfully, and all the threads must perform their critical section.
- One extended form of starvation will be that towards the end, we have all odd or all even processes left, and they are all locked. Once the process reaches to that stage, you must let them perform their critical section to avoid starvation. However, you must make sure that there are no other threads coming in future which could help avoid this situation.

Description of Question 3:

For this Question, you are provided a skeleton code in the file `Sample_Code_Skeleton_Q2.c`. Some functions are completely implemented, and some are partially implemented. Additionally, you can write your own functions if required. Complete the program as per following details so that we can have functionality as described in the Synopsis above. Write all the code in single C file:

2. The code provided reads the content of file for you and populate the threads information in a dynamic array of type `struct thread`. You may some more members to this data structure. If you want to initialize those members, then you can possibly do that during the file read.
3. The `main()` function already contains the code to create and invoke threads. However, there is no synchronization logic added to it. If required, you will add some suitable code in the while loops to perform the tasks required.
4. The `threadRun()` function also contains the code that a thread must run. However, again the synchronization logic is missing. Add the suitable code before and after the critical section.
5. You will need to create and use POSIX semaphore(s) to implement the required logic.
6. The image below shows the expected output for the sample input file provided with this assignment:

[1] New Thread with ID t00 is started.
[1] Thread t00 is in its critical section
[1] Thread with ID t00 is finished.
[2] New Thread with ID t03 is started.
[2] Thread t03 is in its critical section
[2] Thread with ID t03 is finished.
[3] New Thread with ID t07 is started.
[4] New Thread with ID t05 is started.
[5] New Thread with ID t02 is started.
[5] Thread t02 is in its critical section
[5] Thread with ID t02 is finished.
[5] Thread t07 is in its critical section
[5] Thread with ID t07 is finished.
[20] New Thread with ID t01 is started.
[20] Thread t01 is in its critical section
[20] Thread with ID t01 is finished.
[20] Thread t05 is in its critical section
[20] Thread with ID t05 is finished.