CSE 2431 LAB 4 (***)

1. Goal

1) Learn how to use condition variables/semaphores
2) Learn how to test your multi-threaded code

2. Introduction

In this project, we will implement a bounded buffer, and use it to solve the producer consumer problem. A bounded buffer is a buffer with a size limit. It provides two core functions:

1) Push will add an item to the tail of the buffer, and if the buffer is full, wait till the buffer is not full.
2) Pop will delete an item from the head of the buffer, and if the buffer is empty, wait till the buffer is not empty.

I suggest you to implement it in three steps:

1) Implement a bounded buffer correctly, without considering synchronization
2) Add synchronization to protect the buffer
3) Create producer and consumer threads to test your buffer

2.1 Implement a non-synchronized bounded buffer

Implement the following two functions:
Push(…): it adds an item to the tail of the buffer, and if the buffer is already full, print "error".
Pop(): it returns the head of the buffer, and if the buffer is empty, print "error".

Apart from them, you will also need to implement an initialization function and a destroy function. You can either use the circular buffer approach, or use a linked list approach to implement the above functions. If you don't remember how to implement a bounded buffer, review your system I class or read wiki (https://en.wikipedia.org/wiki/Circular_buffer).

You should thoroughly test your code before you move to the next step.

2.2 Add synchronization

You should add synchronization code to protect your bounded buffer when multiple threads are accessing it concurrently.
Push(…): it adds an item to the tail of the buffer, and if the buffer is already full, it waits until the buffer is not full.
Pop(): it returns the head of the buffer, and if the buffer is empty, it waits until the buffer is not empty.

You can use either condition variable or semaphores. Our slides already contain answers.

2.3 Create producer and consumer threads to test your bounded buffer

You should create producer and consumer threads to test your buffer. Your producer thread should generate a number of messages and "push" them into a bounded buffer one by one.
Your consumer thread should "pop" from the buffer in an infinite loop and print the message.
You should check the followings:

1) If there are only producers, do they behave as you expected?
2) If there are only consumers, do they behave as you expected?
3) If there are both producers and consumers, have all "produced" messages been "consumed"?
4) Is any "produced" message "consumed" more than once?
5) Are messages "consumed" in the expected order?

3. Code skeleton

I have provided skeleton of code in three files: bounded_buffer.h bounded_buffer.c main.c
This project demonstrates how people build a complex program in practice: you cannot put all code in a single c file. You need to logically separate your program into multiple components, define the functionality (or interface) of each component in an h file, and implement the functionality in the c file. This brings several benefits:
1) Different programmers can collaborate by working on their own h and c files.
2) Even if one programmer changes its implementation in his/her c files, other programmers do not need to change their programs, as long as the definition in h files are not changed.

In the skeleton, bounded_buffer.h defines the functionalities of the queue. You should not change the function definitions in this h file, although you can add members in the struct. You should implement all defined functions in bounded_buffer.c. Besides, you should create producer and consumer threads in main.c to test your bounded_buffer.

Note: private functions (i.e., functions that you do not expect other programmers to use) do not need to be declared in the h file. If you need additional functions to implement your bounded buffer, just put them in your bounded_buffer.c. It's not necessary to put them in bounded_buffer.h.

4. Requirements and hints

1) You should not use any global variables in this lab, because in lab 5, you will need to create multiple bounded buffers and an implementation using global variables may be broken.
2) In your final submission, you should create 3 producers and 2 consumers in main.c. You should let each producer generate 10 different messages and you should set the size of the bounded buffer to be 5. You should test different settings in your tests, but you only need to submit this setting.
3) Be careful when you let producers push messages into the queue. Following code is wrong:

```
        for(int i=0; i<10; i++)
                push(&queue, &i);
```

   The reason is similar as the one when you pass arguments to pthread_create: objects you pushed into the queue may not be used immediately, so you'd better not modify it.

Solution is also similar: malloc an object before each push and push the malloced object into the queue. Once again, don't forget to free the object.

4) A tricky question for this lab is how you can know all threads have finished their jobs: you cannot simply use pthread_join because the consumer threads never finish, but if you don't call pthread_join, main thread will terminate quickly and then all threads are killed. We will learn a solution in Lab 5; in Lab 4, you can simply ask the main thread to sleep for a while (e.g. 5 seconds) and then call exit(), but remember this is not a good solution.
5) The grader may replace your main.c with his own main.c to test your bounded_buffer implementation. You need to make sure your implementation of bounded_buffer does not depend on anything in main.c.
6) Possible test cases: 1 producer and 1 consumer; 5 producers and 1 consumer; 1 producer and 5 consumers; set the max size of bounded buffer to be 1.

5. Compilation

You need to write your own Makefile. Since this program includes multiple files, you need to compile all of them into a single executable file.
If you need to use semaphores, you also need to add –lrt to the gcc command in Makefile.