Programming Assignment 3
Bi-directional IPC through Shared Bounded Buffers

Due: Monday, Dec 1 at 11:59pm on Canvas.

Learning Objectives

The objective of this assignment is to apply key knowledge about concurrency and synchronization through implementing a bi-directional inter-process communication (IPC) mechanism using shared bounded buffer.

Problem Description

This project is based on Project 2, which already provided a working program that uses shared memory to send messages to a chat server. You need to extend the program in the following ways.

1. Make the program able to both send messages to, and receive messages from server. You need to use multiple threads for this. Use the POSIX thread library to create threads: pthread_create().
2. You will need to create two bounded buffers in the shared memory, one for each message direction. The data structure is defined as follows.

```
#define MAX_MSG_LEN 1024
#define NUMBER_OF_MSG 5

struct message_buffer {
      pthread_mutex_t mutex;
      pthread_cond_t full, empty;
      int front, tail;
      char messages[NUMBER_OF_MSG][MAX_MSG_LEN];
};

struct shm_data {
      struct message_buffer s2c;
      struct message_buffer c2s;
};
```

   The header file that contains these definitions is provided for you.

3. The threads in both the server and the client use pthread mutex (lock) and condition variables for synchronization. The mutex and condition variables are part of the shared data structure.
4. Like in the previous project, the server's binary is provided for you. Your program is the client that communicates with the server through the following protocol.

a. Establish an IPC channel with the server through shared memory. Both the server and client programs are invoked through command line, with an integer provided as the key of the shared memory.

   e.g.,
   ```
   ./project3_server 365
   ./client 365
   ```

   Here `365` is the key of the shared memory. It serves as a channel identifier. Processes that attach a shared memory segment with the same key will have the corresponding page table entr[ies] mapped to the same physical frame[s]. Pick an arbitrary key when testing your program, to avoid interfering with other students testing on the same lab machine.

b. The client and server will communicate using the shared data structure above. User of your program types in a message that is up to `MAX_MSG_LEN` long (including string terminator). The messages are buffered in the two-dimensional array `messages`. Each message buffer can store up to `NUMBER_OF_MSG` messages. Sender puts messages in the buffer, and receiver gets the messages from the buffer. Note that you no longer need to divide messages into fixed chunks. Both the client and server can send and receive messages. Thus two message buffers are required. `s2c` is for messages sent from server to client, and `c2s` is for messages sent from client to server. Use the method we learned in the class to implement this bounded buffer. Apply appropriate synchronization methods to prevent data races.

c. When the server starts, it initializes the data structures including the mutex and condition variables. When testing, always start the server program first. The client program can assume all the fields of the data structures have been properly initialized.

d. The server follows the bounded buffer example we discussed in the class. You can find the pseudo code in the lecture slide.

Requirements

There shall be no race conditions or memory errors from your program. All messages must be sent and received correctly.

Testing your program

Manually interacting with the program may not reveal any race conditions that may exist. We have provided a testing script `test_script.sh` for you to use. You may have to first make the file executable by typing `chmod  u+x  test_script.sh`. Then use pipe to send the output to the client and/or server to see if your program behaves as expected.

e.g.,
```
./test_script.sh 10|./client 365
```

Like in Project 2, you need to test your program on one of the lab machines **cselx01.cse.usf.edu-cselx37.cse.usf.edu**. The server binary is located at

`/home/x/xou/cop4600/project3_server`.

Copy the binary to your folder on the lab machine and test it there.

Important: When compiling your client program on the lab machines, you must provide the -pthread option.

```
gcc -pthread -o client client.c
```

What to Submit

Name your C file as `client.c` and upload it on CANVAS.

Background Knowledge on POSIX Thread Synchronization

The POSIX thread (pthread) library provides the standard locking (mutex) and condition variable operations. The library functions you'll likely need include: `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `pthread_cond_wait()`, and `pthread_cond_signal()`. Use the man pages to understand their semantics and correct usage. These synchronization mechanisms can work among threads from different processes (like in this project). The server code has already initialized the locks and condition variables properly to this effect.

Extra Credit (up to 50 points):

When a user is in the middle of typing a message, if a new message arrives it will mingle into the half message that is being typed in. Please research a way to postpone the client program displaying a new message if the user is in the middle of typing one. The new message will be immediately displayed once the user types "enter" indicating finishing the message.

Submit your extra credit in a separate program named `extra_credit.c`, along with a README file explaining what/how you have accomplished in the extra credit, and how to run the program. You must still submit the baseline version `client.c`. Please put both files into a zip or tar file for submission.

Warning: We ourselves have not found a working solution for this yet. So this is an open-ended research endeavor. We cannot guarantee that a solution exists that satisfies our requirement. Even if you can only make it work partially, please still submit it and we may award some partial extra credit points!

Note

- All work must be independent and yours. *Using code found anywhere is prohibited.*
- All programming projects have a nominal point of 100. The total point of all programming projects is a weighted average based on each project's amount of work.