EECS 338 Homework 6: Synchronization

General requirements:

- See the posted due date.
- Create a typed report for discussion questions and program output.
- Upload a single, compressed file (e.g. zip) to Canvas that contains all required files (programs + report).
- All work should be your own, as explained in the Academic Integrity policy from the syllabus.
- Include either a single makefile that compiles all programs or a separate makefile for each.

1. Create two programs (separate .c files), as described below, that send and receive messages asynchronously using shared memory that is protected using a semaphore. Both programs should have identical user input loops that display the current message and allow the user to repeatedly quit, change the shared message, or redisplay the message. Each program should use two shared memory segments: one for the message, and one for the semaphore. Any code that writes a new message is a critical section and should be protected using the semaphore. **Include an example of the programs' output in your report.**

   a. *Server:* One program is a server that creates both shared memory segments and unlinks them before exiting. It also sets the original message.
   b. *Client:* The other program is a client that uses both shared memory segments for reading and writing. However, it should not use O_CREAT, ftruncate, or shm_unlink.
   c. *Execution order:* For simplicity, assume the server always stops *after* the client stops. In this way, the client can assume that shared memory has not been unlinked. The client should check for mmap() failures as usual.

   Below is an example of how your output might look. You can run the server and client in separate terminal shells. User input is shown in **bold**. Normally, the text would appear without any gaps, but each of the two users could be entering text at any time. However, the following output is shown with gaps between text to demonstrate time delays between user actions.

   Example where the server starts first:

```
Server starts:                      |
Message: Hello from the server!     |
0. Quit                             | Client starts after server:
1. Change message                   | Message: Hello from the server!
2. Check message                    | 0. Quit
                                    | 1. Change message
                                    | 2. Check message
Enter your choice: 1                |
Enter message: Hi client. How       |
are you?                            |
Message: Hi client. How are         | Enter your choice: 2
you?                                | Message: Hi client. How are you?
0. Quit                             | 0. Quit
1. Change message                   | 1. Change message
2. Check message                    | 2. Check message
                                    | Enter your choice: 1
                                    | Enter message: Hi server. I'm
                                    | fine.
Enter your choice: 2                |
Message: Hi server. I'm fine.       |
0. Quit                             |
```

`1. Change message`	
`2. Check message`	
`Enter your choice:` **`0`**	
	`Message: Hi server. I'm fine.`
	`0. Quit`
	`1. Change message`
	`2. Check message`
	`Enter your choice:` **`0`**

2. Create two programs (separate .c files), as described below, to allow one-way, asynchronous communication using a socket, P-threads, and a semaphore. The server will create two child threads, each of which will modify a global message buffer using strings sent from two client instances through the socket. **Include an example of the programs' output in your report.**

   a. *Server description:* One program is a server that creates the global message buffer, socket, semaphore, and child threads. It also sets the original message. It should have exactly one function that is executed on each child thread, described below.
   b. *Server child threads:* The function executed in each child thread should call "accept" for the socket and then repeatedly read new strings from the socket until "quit" is received. For each new string that is read (except "quit"), the function should use "sprintf" to write that string to the global message buffer, protecting this critical section operation with the semaphore.
   c. *Server parent thread:* The parent thread of the server should allow the user to repeatedly choose two options: quit or print the current value of the message buffer.
   d. *Client description:* The second .c program is a client that connects to the socket. It allows the user to repeatedly choose two options: quit or enter/send a new message to the server. You should run two instances of the client, each in a separate terminal shell.
   e. *Execution order:* The server and client should correctly respond to every possible execution order. A typical order would be: server starts, clients start, clients quit, server quits. Below are requirements for other possibilities:
      - *Client starts before server:* There will be no available socket available, and it should terminate with an error message.
      - *Server quits before clients:* If the server user chooses to quit before the clients quit, the server should not terminate until after the clients close their socket connections. Simply use pthread_join() for each thread before closing the primary socket file descriptor and destroying the semaphore.

Note that getting string input immediately after numerical input can be a problem. See the comment about this in problem #2 above. Below is an example of how your output might look. You can run the server and clients in separate terminal shells. User input is shown in **bold**. Normally, the text would appear without any gaps, but each of the three users could be entering text at any time. However, the following output is shown with gaps between text to demonstrate time delays between user actions.

Example:

Server starts:		
`Listening for`		
`clients...`		
	1st client starts after server:	
	`0. Quit`	
`Message: Server is`	`1. Send message.`	
`here.`	`Enter your choice:` **`1`**	
`0. Quit`	`Enter message:` **`1st`**	
`1. Check message.`	**`child is here!`**	

`Enter your choice: `**`1`**
`Message: 1st child is here!`
 	
`0. Quit`
`1. Check message.`
`Enter your choice: `**`1`**
`Message: 2nd child too!`
`0. Quit`
`1. Check message.`
`Enter your choice: `**`0`**
`Waiting for child threads to exit.`
	
	`0. Quit`
	`1. Send message.`
	`Enter your choice: `**`0`**
		2nd client starts after server:
		`0. Quit`
		`1. Send message.`
		`Enter your choice: `**`1`**
		`Enter message: `**`2nd child too!`**
		
		`0. Quit`
		`1. Send message.`
		`Enter your choice: `**`0`**

Tip (client/server design process): We recommend a gradual design process because there are so many different techniques involved (multithreading, sockets, separate client/server algorithms, and a semaphore). The following are suggestions for phases of increasing difficulty.

- Review the socket problem in HW #4. Then design a client loop to allow the user to repeatedly send messages to the server. Don't worry about multithreading or a global "message" buffer. Just have the server print whatever the client sends, over and over.
- Make the server create a child thread that handles all communication with the client.
- Make the server write the client messages to the message buffer.
- Determine whether your approach works with <u>two</u> clients, and revise as necessary.
- Include the semaphore for the critical section.

3. Implement a semaphore-based solution for the bounded-buffer problem presented in Section 5.7.1 of the textbook as described below. Be sure to use global variables and call "sem_destroy" where appropriate (e.g. semaphores). Remember that the algorithm given in Section 5.7.1 does not show the buffer indexes "in" and "out" that are necessary and described in Section 3.4.1.
   a. The parent thread initializes all semaphores and creates a child thread.
   b. The parent thread is a producer and uses a loop to store N values in a buffer of size N / 2. Use any values you wish. This will put some initial values in the buffer before the consumer begins.
   c. The child thread immediately sleeps when created such that the buffer has time to fill up with N / 2 values before proceeding. For example, sleep for 1 or 2 seconds. This is intended to allow a competition between the producer and consumer *after* the buffer has filled up.
   d. The child thread is a consumer. Upon awakening from the forced sleep, the child thread uses a loop to read and print N values (one at a time) from the buffer. After consuming each value, print a message indicating the value that was consumed. Below is an example where the child thread consumed a value "40":
      ```
                      Child consumed data = 40
      ```
      Note that, when this step begins, the child will signal(empty) after consuming its 1st value. This will unblock the parent and begin a competition between the producer and consumer.
   e. Create a way to print the theoretical value of the "mutex" semaphore before each call to `sem_wait(&mutex)` and `sem_post(&mutex)`. Assume that "mutex" is always decremented for wait() such that it may become negative. For example, you might create a global variable "S" that is decremented and incremented accordingly. Note that any changes to this "S" variable would also need to be mutually exclusive, so you would need *another* semaphore to ensure that the parent and child

threads cannot change its value concurrently. Though not required, you may wish to create helper functions that replace `sem_wait(&mutex)` and `sem_post(&mutex)` and incorporate this additional functionality. Below is an example of possible output for N = 5 and a producer that creates the sequence 10, 20, 30, 40, 50:

```
Parent wait: S=0
Parent post: S=1
Parent wait: S=0
Parent post: S=1
Child wait: S=0
Child consumed data = 10
Child post: S=1
Child wait: S=0
Child consumed data = 20
Parent wait: S=-1
Child post: S=0
Parent post: S=1
Parent wait: S=0
Parent post: S=1
Child wait: S=0
Child consumed data = 30
Child post: S=1
Child wait: S=0
Child consumed data = 40
Parent wait: S=-1
Child post: S=0
Parent post: S=1
Child wait: S=0
Child consumed data = 50
Child post: S=1
```

In the example above, the following synchronization occurs:
(a) The parent produces N / 2 = 2 values.
(b) The child consumes values 10 and 20. (NOTE: This ordering is <u>not</u> determined by the program.)
(c) The parent calls wait(&mutex) while the child is in its critical section consuming 20.
(d) The parent produces its 3rd and 4th values. (NOTE: This ordering is <u>not</u> determined by the program.)
(e) The child consumes values 30 and 40. (NOTE: This ordering is <u>not</u> determined by the program.)
(f) The parent calls wait(&mutex) while the child is in its critical section consuming 40.
(g) The parent produces its 5th value. (NOTE: This ordering is <u>not</u> determined by the program.)
(h) The child consumes values 50.

Provide the following in your report:

- **Explanation of how you implemented part (e).**
- **Program output.**
- **Description of the synchronization shown in your output. Did the producer ever wait on the consumer? Did the consumer ever wait on the producer?**