Assignment #4: Synchronization

CS3010 Fall 2024
25 points
due Saturday, Oct. 26th, 11:59 pm
second chance (for 22.5 points): due TBA, 11:59 pm

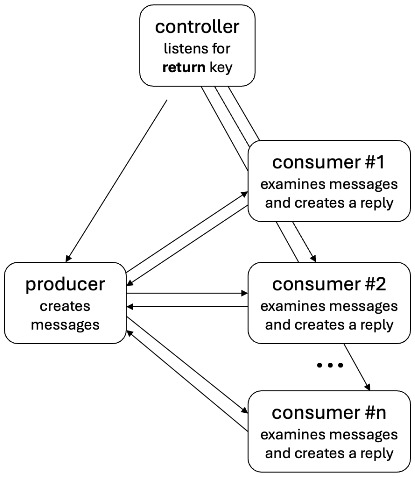
1 Threads and Synchronization

You'll write a multithreaded producer-consumer program in which threads synchronize their actions using the mechanism of mutex locks. There will be one producer thread, n consumer threads, and a controller thread.

The producer thread writes a message for a specific consumer into a shared variable and notifies the consumers when the value is available. Consumers examine the shared data, and the consumer for whom the message is intended processes the message, creates a response, and then notifies the producer that a response is ready.

The controller thread waits for the return key to be pressed on the keyboard and then notifies the other threads that it's time to exit.

Here's a diagram of the structure of the program:



You may work individually or with a partner.

1.1 Example code

First, get `simple-mutex-example.c` from the class gitlab repo. Compile this and run it. This program shows an example of how to create pthreads, pass data to them, and use a mutex lock. On silk, you will need to compile this in the following way:

```
$ gcc simple-mutex-example.c -lpthread
```

You can also run this on macOS or on Windows. To compile it at the command line on macOS or Windows:

```
$ gcc simple-mutex-example.c
```

1.2 Structures

Define these structures:

```
#define NUM_RECEIVERS 4
#define MAX_MESSAGE_LEN 31

typedef struct {
  bool done;              // whether or not the program is done
  pthread_mutex_t *mutex; // synchronization for this variable
} ControlInfo;

typedef struct {
  int recipientID;                 // id of intended recipient
  char message[MAX_MESSAGE_LEN+1]; // the message for the recipient
  char reply[MAX_MESSAGE_LEN+1];   // the reply from the recipient
  bool messageReady;               // whether data is ready for the recipient
  bool replyReady;                 // whether reply is ready for the sender
  pthread_mutex_t *mutex;          // synchronization for these variables
} DataInfo;

typedef struct {
  int myID;             // id of sender or receiver
  DataInfo *data;       // the data
  ControlInfo *control; // control
} ThreadInfo;
```

In the `main()`, create and initialize a single instance of `ControlInfo` and a single instance of `DataInfo`.

Create 1 + `NUM_RECEIVERS` instances of `ThreadInfo`.

Create a single sender (producer) thread and `NUM_RECEIVERS` receiver (consumer) threads. Pass a unique instance of `ThreadInfo` to each. Set the `myID` field for the sender thread to zero, and set the `myID` field of the consumers to the values 1, 2, ..., `NUM_RECEIVERS`.

Create a single controller thread and pass the `ControlInfo` to it.

1.3 Implementation

Implement these functions:

```
void *receiver(void *);
void *sender(void *);
void *controller(void *);
```

1.3.1 Spinning

Implement this function as well:

```
void spin(int val);
```

This function consists of a doubly-nested for loop, over the range 0 to $val - 1$ in each case. The body of the inner loop is empty. This will serve as a delay mechanism.

1.3.2 Rules

Do not use `sleep()` in your program!

Do not use any global variables in your program!

Do not read or modify a shared variable outside of a lock-unlock region!

1.3.3 Controller

The controller will merely do a `getchar()` call. When this call returns, meaning that the user has pressed the return key, the producer will set the `done` field in the `ControlInfo` instance to `true`.

1.3.4 Sender

The sender will loop until the `done` field of the `ControlInfo` instance has been set to `true`. In this loop it will do the following:

```
call spin() with a value such as 10000
if replyReady is true {
  print "reply: [message]", where message is the string in the message field
  set replyReady to false
}
if messageReady is false {
  generate a random integer in the range 1 to NUM_RECEIVERS
  put the integer in the recipientID field of the data struct
  put this string in the message field of the data struct:
      "message for i", where i is the random integer that was generated
  set messageReady to true
}
```

1.3.5 Receiver

Each receiver will wait until `messageReady` is `true`. Each receiver will then check the `recipientID` field to see whether it matches that receiver's id. If so, the receiver will consume the message and create a reply of the form `[message] - read by i`, where `[message]` is the original message and `i` is that receiver's id. It will then set `replyReady` to `true`.

A receiver can process a message only when `messageReady` has been set to `true` by the sender.

1.4 General comments

There is a single instance of the `DataInfo` structure, and the sender and all receivers have a reference to this instance. Similarly, there is a single instance of the `ControlInfo` structure.

In each iteration of their loops, the sender and the receivers need to check whether the controller has set the `done` flag to true; if the flag has been set, then they exit, by callingl `pthread_exit(NULL)`.

1.5 Output

You should see sender and the receivers alternate their printing, like this:

```
message for 2
reply: 'message for 2 - read by 2'
message for 4
reply: 'message for 4 - read by 4'
message for 2
reply: 'message for 2 - read by 2'
message for 2
```

```
reply: 'message for 2 - read by 2'
message for 2
reply: 'message for 2 - read by 2'
message for 4
reply: 'message for 4 - read by 4'
message for 1
reply: 'message for 1 - read by 1'
message for 3
reply: 'message for 3 - read by 3'
message for 1
reply: 'message for 1 - read by 1'
```

and this should continue until you press the return key.

2 Key Points

Here are the key points and things to remember for this assignment:

- any access to shared data by a thread must be protected by a lock/unlock pair
- we cannot predict the relative order in which threads will execute their statements

3 What to Submit

Submit your `sync.netid.c` and `sync.netid.h` files.

4 Extra Credit

For a bit of extra credit: implement the same behavior, except now use two condition variables, as described in the "Condition Variables: Bounded Buffer" example Lecture #5.

You might have to modify the data structures from the base assignment. The sender and the receivers must now react to two different conditions (message ready por not ready; program done).

Put your code in a file named `sync-ec.netid.c`