Assignment #5: Synchronization

CS201 Spring 2020
10 points
due Sunday, Mar. 29th, 11:59 pm
second-chance (for 9 points) due Sunday, Apr. 5th, 11:59 pm

1 Synchronization

You'll write a multithreaded C program that will solve the Readers/Writers Problem of the First Kind, as we discussed in class.

1.1 Algorithm

Here's the complete solution, in pseudocode:

```
// initialization
semaphore rw_sem = 1;
semaphore mutex = 1;
int num_readers = 0;

// reader thread does this
for (i=0; i<NUM_READS; ++i) {
  wait(mutex); // lock for accessing num_readers
  ++num_readers;
  // print a message with the current value of num_readers
  if (num_readers == 1) // if I am first reader
    wait(rw_sem); // lock for shared buffer
  signal(mutex); // done accessing num_readers
  // print a message that this reader is reading
  // perform reading
  wait(mutex); // lock for accessing num_readers
  --num_readers;
  // print a message with the current value of num_readers
  if (num_readers == 0) // if I am last reader
    signal(rw_sem); // release lock for shbuf
  signal(mutex); // done accessing num_readers
}

// writer thread does this
for (i=0; i<NUM_WRITES; ++i) {
  wait(rw_sem); // wait
  // print a message that this writer is writing
  // do the writing
  signal(rw_sem); // done
}
```

1.2 Named semaphores

On Linux and macOS, a named semaphore is a system-wide resource. The name lets a user uniquely identify a semaphore (so that my semaphores don't get mixed up with your semaphores).

The call `sem_open()` initializes a named semaphore. By specifying `O_CREAT` and `O_EXCL`, you can create a named semaphore so that only you can use it.

The call `sem_close()` tells the system that you're done using the semaphore, and `sem_unlink()` actually removes the semaphore.

If your program exits (or crashes) after `sem_open()` but before `sem_unlink()`, you'll get an error the next time you try to `sem_open()` it. In this case, clean up the semaphore with `sem_unlink()` and try again.

The example program `semaphore-example.c`, under Examples in the class gitlab space, shows how to create and use and clean up a named semaphore.

1.3 Declarations

You will have to do this assignment on Linux or macOS. You should run your code on kaladin though to verify that it works correctly.

Declare these structures:

```
typedef struct {
  pthread_mutex_t mutex;
  sem_t *rw_sem;
  int readerCount;
} SyncInfo;

typedef struct {
  SyncInfo *syncInfo;
  char myName[32];
} ThreadInfo;
```

You should write these two functions:

```
void *reader(void *data);
void *writer(void *data);
```

The `data` parameter will point to a `ThreadInfo` struct. Instead of the `while (true)`, the `reader()` function will have a for-loop in which it tries to perform reading. Similarly he `writer()` function will have a for-loop in which it tries to perform writing. The reading and writing actions will actually just be this call: `sleep(1)`.

1.4 Compiling and linking

You'll have to include these header files:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
```

On Linux, you will have to compile your program this way:

```
$ gcc rw1.netid.c -lpthread
```

On macOS, you do not need the `-lpthread`.

2 What to Do

Do the actual implementations in C of the program described above. Put your structure definitions and function prototypes in `rw1.netid.h`, and put your code in `rw1.netid.c`; submit these two files to Blackboard.

You will need to write the `main()` function. Your `main()` will create the threads and then wait for them to finish. Leave your `main()` in your file when you submit it.

Create ten reader threads and two writer threads. Use these `#define` statements:

```
#define NUM_READERS 10
#define NUM_WRITERS 2
#define NUM_READS 5
#define NUM_WRITES 5
```

Define this also:

```
#define SEM_NAME "/SEM_netid"
```

and use your actual netid.

For an example of how to use a semaphore, see the sample program `semaphore-example.c`, in the Examples directory of the class gitlab site (https://gitlab.uvm.edu/Jason.Hibbeler/ForStudents/tree/master/CS201/).

Each thread will have its own `ThreadInfo` instance. Create an array of these, as you did in Assignment #4.

`NUM_READS` is the number of times that each reader goes through its loop, and `NUM_WRITES` is the number of times that each writer goes through its loop.

You need only a single instance of SyncInfo.

Give each thread a name. In the for-loop that you will use for creating the reader threads, do this:

```
sprintf(threadInfo[idx].myName, "Reader␣%d", i);
```

Print out `hello from Reader n` for each reader when a reader thread starts; print a similar message for each writer rhread

Put your code in a file `rw1.netid.c` and your declarations in `rw1.netid.h`.

3 Testing

How will you know this is working correctly? You should see that only one writer at a time can write, but that more than one reader can read. Print a message each time a reader thread modifies the `numReaders` variable. This output will help you see the program is correct.