In this problem set, you will explore synchronization issues on a simulated multi-processor, shared-memory environment. We will not use threads-based programming, but instead will create an environment in which several UNIX processes share a memory region through `mmap`. Each process represents a parallel processor.

We will number each of these "virtual processors" with a small integer identifier which will be held in the global variable `my_procnum`. This is not the same as the UNIX process id, although you will probably need to keep track of the UNIX pids too. `my_procnum` ranges from 0 to `N_PROC-1`. `N_PROC` is the **maximum** number of virtual processors which your implementation is required to accept. For this project, `#define` it as 64.

To implement sleeping and waking in this project, the UNIX signal facility will be used to simulate *inter-processor interrupts*. Use signal SIGUSR1 and the system calls `sigsuspend` and `sigprocmask`, as discussed in class.

The starting point is an atomic test and set instruction. Since "some assembly is required," this will be provided to you in the file `tas.S` (32-bit), or `tas64.S` (64-bit). Use it with a makefile or directly with gcc, e.g. `gcc fifotest.c fifolib.c semlib.c tas.S` A .S file is a pure assembly language function. At the C level, it will work as:

```
int tas(volatile char *lock)
```

You will have to write your own prototype for the function since no header is provided. The tas function works as described in the lecture notes. A zero value means unlocked, and tas returns the *previous* value of `*lock`, meaning it returns 0 when the lock has been acquired, and 1 when it has not.

It is suggested that you implement a **spin lock** using this atomic TAS, and use that spin lock as a mutex to help implement the functions below. It will not be necessary to implement a full mutex lock with blocking/yielding, as that functionality will be built-in to the your semaphores.

Problem 1 -- Test the test-and-set

As a sanity check, write a simple test program that creates a shared memory region, spawns a bunch of processes sharing it, and does something non-atomic (such as simply incrementing an integer in the shared memory). Show that without mutex protection provided by the above TAS primitive, incorrect results are observed, and that with it, the program consistently works. Use a sufficient number of processes (typically at least equal to the number of CPUs/cores in your computer) and a sufficient number of iterations (millions) to create the failure condition. Of course, be mindful of silly things like overflowing a 32-bit int!

Problem 2 -- Implement semaphores

Create a module, called `sem.c`, with header file `sem.h`, which implements the semaphore operations defined below. You will need to make use of the spinlock mutex derived from the provided TAS function. **This is the only synchronization primitive** on which the semaphores should be based!

```
void sem_init(struct sem *s, int count);
        Initialize the semaphore *s with the initial count. Initialize
        any underlying data structures.  sem_init should only be called
        once in the program (per semaphore).  If called after the
        semaphore has been used, results are unpredictable.

int sem_try(struct sem *s);
        Attempt to perform the "P" operation (atomically decrement
        the semaphore).  If this operation would block, return 0,
        otherwise return 1.
```

```
void sem_wait(struct sem *s);
      Perform the P operation, blocking until successful.  Blocking
      should be accomplished by noting within the *s that the current
      virtual processor needs to be woken up, and then sleeping using
      the sigsuspend system call until SIGUSR1 is received.  Assume
      that the extern int variable my_procnum exists and contains
      the virtual processor id of the caller.  The implementation by
      which you keep track of waiting processors is up to you, e.g. it
      could be a linked list.  Be mindful however that any internal
      data structures such as a linked list of waiting processes can not
      be managed by malloc or anything which relies on malloc (such as the
      dreaded C++ standard data structures class template library), because
      such memory will be private to a specific process and not in the
      shared memory pool :(

void sem_inc(struct sem *s);
      Perform the V operation.  If any other processors were sleeping
      on this semaphore, wake them by sending a SIGUSR1 to their
      process id (which is not the same as the virtual processor number).
      If there are multiple sleepers (this would happen if multiple
     virtual processors attempt the P operation while the count is <1)
      then all must be sent the wakeup signal.
```

It is suggested that after you code up this semaphore library, you write a small testing framework to make sure it works correctly. Note that you are required to implement all four operations above correctly, even if you do not wind up using all of them in your FIFO below. You do not have to submit your test framework for this part.

Problem 3 -- A FIFO using semaphores

Now create a fifo module, `fifo.c` with associated header file `fifo.h`, which maintains a FIFO of unsigned longs using a shared memory data structure protected and coordinated **exclusively** with the semaphore module developed above. Take note of the word exclusive. If you have wait queues, fork, SIGUSR1 or similar code in your fifo.c module, you did not understand the word. Depending on your approach you may or may not need to use all of the semaphore functions above. However, if your FIFO implementation takes more than about 100 lines of code, you are probably over-complicating things.

```
void fifo_init(struct fifo *f);
            Initialize the shared memory FIFO *f including any
            required underlying initializations (such as calling sem_init)
            The FIFO will have a fifo length of MYFIFO_BUFSIZ elements,
            which should be a static #define in fifo.h (a value of 4K is
            reasonable).
```

```
void fifo_wr(struct fifo *f,unsigned long d);
               Enqueue the data word d into the FIFO, blocking
               unless and until the FIFO has room to accept it.
               Use the semaphore primitives to accomplish blocking and waking.
               Writing to the FIFO shall cause any and all processes that
               had been blocked because it was empty to wake up.

unsigned long fifo_rd(struct fifo *f);
               Dequeue the next data word from the FIFO and return it.
               Block unless and until there are available words
               queued in the FIFO.  Reading from the FIFO shall cause
               any and all processes that had been blocked because it was
               full to wake up.
```

NOTES: It is intended that the semaphores that you require to accomplish synchronizing this FIFO be part of the `struct fifo`. One single mmap area is sufficient for holding the `struct fifo` which in turn can hold the semaphores. Since we are artificially placing a small static limit on the number of virtual processors, there is no need to dynamically allocate anything.

The FIFO itself should be implemented as a fixed array of longs as a circular buffer with suitable pointers or indices for the next open write slot and the next available read slot. There are two distinct synchronization issues in the FIFO: 1) protecting the integrity of the `struct fifo` data structure during the enqueue and dequeue operations 2) coordinating the sleep/wakeup events of a reader waiting on an empty FIFO, or a write waiting for room in a full FIFO. These are symmetrical conditions. Issue #1 is short-lived and is an appropriate use of a spin-lock mutex (consider: how can you use a semaphore as a mutex?). Issue #2 is long-lived and will make use of the inherent sleep/wakeup mechanism of the semaphore.

Problem 4 -- Test your FIFO

Create a framework for testing your FIFO implementation. Establish a `struct fifo` in shared memory and create two virtual processors, one of which will be the writer and the other the reader. Have the writer send a fixed number of sequentially-numbered data using `fifo_wr` and have the reader read these and verify that all were received.

Next, give your system the acid test by creating multiple writers, but one reader. In a successful test, all of the writers' streams will be received by the reader complete, in (relative) sequence, with no missing or duplicated items, and all processes will eventually run to completion and exit (no hanging). A suggested approach is to treat each datum (32-bit word) as a bitwise word consisting of an ID for the writer and the sequence number. (It is not necessary to test under multiple readers, but your fifo code should work correctly for this case)

Use reasonable test parameters. Remember, an acid test of a FIFO where the buffer does not fill and empty quite a few times has a pH of 6.9, i.e. it isn't a very strong acid. You should be able to demonstrate **failure** by deliberately breaking something in your implementation, e.g. reversing the order of two locking operations. You should then be able to demonstrate success under a variety of strenuous conditions.

Submit all of the code comprising this final test system, i.e. your sem.[ch], fifo.[ch] and main.c files, as well as output from your test program showing it ran correctly. If the output is very verbose, you may trim the uninteresting stuff with an appropriate annotation.