Problem 1 -- Short Answer

The following pseudo-code represents an attempt to enforce mutual exclusion in a critical region

```
void f()
{
        /* DISABLE_INTERRUPTS is a generic name for an assembly-level
           instruction that saves the current hardware interrupt mask
           and disables all interrupts */
        existing_mask=DISABLE_INTERRUPTS;
        /* Perform critical region operation */
        ENABLE_INTERRUPTS(existing_mask);     /* Restore prev IRQ mask */
}
```

Discuss why is (or why is not) this approach valid for the following situations:

a) user-level code, single-threaded, single CPU
b) user-level code, multi-threaded, single CPU
c) user-level code, multi-threaded, multi-CPU
d) kernel-level code, single CPU
e) kernel-level code, multi-CPU

Problem 2 -- A synchronization programming problem: spinlock, semaphores, fifos

In this problem, you will explore synchronization issues on a simulated multi-processor, shared-memory environment. You'll be building a FIFO that is shared in memory among multiple tasks (both readers and writers). The FIFO in turn will be built on top of a semaphores implementation that you will code, and these semaphores will rely on an atomic TAS (Test & Set) primitive that I will provide. This problem will therefore exercise both locking and event coordination.

Test Environment and Framework

We will not use threads-based programming, but instead will create an environment in which several single-threaded UNIX processes share a memory region through `mmap`. Each process represents a parallel processor. This approach is much easier to debug.

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 below.

Modular programming

This is a large programming assignment with building blocks. Each problem builds upon the previous problem. E.g. you'll build a spinlock "library" and then build your semaphore library, making use of your spinlock library. However, the libraries should be written so that they are generic, not restricted to this specific purpose. In a non-OO language such as C, this requires some discipline.

For each library, e.g. `spinlock.c`, you'll also write a header file e.g. `spinlock.h`. Header files contain

prototype declarations of the public functions in the library, struct definitions, #defines, and other declarative material. `.h` files never contain code (functions) or global variable declarations.

You can either use a `Makefile` to automate building this multi-part program, or you can compile manually each time.

Problem 2A -- Test and Set and Spin Lock

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)
```

The tas function works as described in the lecture notes. A zero value of `*lock` 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.

Now, implement a **spin lock** using this atomic TAS. It will not be necessary to implement a full mutex lock with blocking, as that functionality will later be built-in to your semaphores. Your spin lock library will consist of two functions, `spin_lock` and `spin_unlock` which will be similar to what is in the lecture notes. *Note: it may improve performance to use the* `sched_yield()` *system call within the spin lock retry loop*.

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 spinlock/TAS primitive, incorrect results are observed, and that with it, the program consistently works. Use a sufficient number of processes (typically 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!

Note about Macos & VMs: This assignment should be fine in both Linux and MacOs, although the assembly language .S file might require some minor tweaks. If you are running inside a Virtual Machine (e.g. VirtualBox or VMWare) you may have trouble generating a synchronization failure in a reasonable amount of time unless you allocate two or more CPU cores to your VM.

Note about Windows: This really isn't going to work at all on Windows.

Problem 2B -- Implement semaphores

Create a module, called `sem.c`, with header file `sem.h`, which implements the four semaphore operations defined below. You will need to make use of the spinlock mutex that you developed in part 1. **This is the only synchronization primitive** (other than the sleep/wakeup which is provided by the operating system) on which the semaphores should be based!

I am not stipulating what is inside `struct sem` -- that is your own design. I give some hints below of what you'll probably need.

```
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.
```

```
        Note that the return type is void so no errors are anticipated.
        The pointer s is assumed to point within an established
        area of shared memory.  This function does not allocate it!

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.  See below
        about how blocking and waking are to be implemented.

void sem_inc(struct sem *s);
        Perform the V operation.  If any other tasks 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.
```

Blocking and Waking

Each process is a task (simulated CPU) and signals are simulated interrupts. To block the task in `sem_wait`, you will use the `sigsuspend` system call, which has the useful property that it puts your process to sleep AND changes the blocked signals mask atomically. The task sleeps until any signal is delivered to it. Then another task which performs `sem_inc` will wake up the sleeping process at a later time, using `SIGUSR1`.

Now, inside of your `struct sem` you will have some kind of mechanism for keeping track of which tasks are waiting on the semaphore. It could be an array, it could be a bitmap indexed by "virtual processor number", it could be a linked list. Be very careful however: whatever data structures you use must be contained within the `struct sem`. You can't use `malloc` (or anything derived from it, such as `new` in C++) here because memory allocated that way is not part of the `MAP_SHARED` memory region and is therefore not actually shared among the various processes!

Common Problems

Need to protect wait list: Whatever you use for a waiting list, be mindful of using spinlock mutex protection as needed when adding to it (sem_wait) or removing from it (sem_inc). Let's say two tasks are in sem_wait at the same time and the semaphore value is 0 so they both want to wait. If you don't lock properly, they may both try to add to the wait list at the same time and corrupt it!

The insidious **lost wakeup**: After you put yourself on the wait list in sem_wait and release any mutex locking it, there is a brief race condition window where another task in sem_inc sees you on the wait list and sends a SIGUSR1. BUT, you haven't gotten to `sigsuspend` yet. So you'll go to sleep, waiting for a SIGUSR1 that may never come

again! To solve this, you'll need to use `sigprocmask` to block (mask) SIGUSR1 while you are adding yourself to the wait list, and take advantage of the atomic property of `sigsuspend` to unmask SIGUSR1 and go to sleep.

Testing your semaphore library

It is suggested that after you code up this semaphore library, you write a small testing framework to make sure it works correctly. For example, make several "consumers" that repeatedly call sem_wait, and several "producers" that call sem_inc a corresponding number of total times. Verify that all tasks complete and don't get stuck. Consider adding "instrumentation" to your semaphore library for debugging, e.g. counters for how many times a semaphore blocks and wakes up.

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 2C -- 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 an mmap'd 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 on the "empty" condition 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 on the "full"
                condition to wake up.
```

Important Notes about your FIFO library

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. You'll have to make sure the mmap area is big enough to hold `sizeof(struct fifo)` including its FIFO buffer. The FIFO itself should be implemented with a fixed-sized array of longs as a circular buffer with suitable pointers or indices for the next open write slot and the next available read slot. I make the number of elements in the FIFO a constant value governed by a `#define` to make the coding much easier.

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 spin-lock mutex as opposed to a blocking mutex?). Issue #2 is long-lived and will make use of the inherent sleep/wakeup mechanism of the semaphore.

Problem 2D -- Test your FIFO

Create a framework for testing your FIFO implementation. First, the baby steps: 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 (unsigned long) as a bitwise word consisting of an ID for the writer and the sequence number. The reader can keep track of the last sequence number seen from each writer. A datum received out of sequence, or duplicated, or missing, is a sign of corruption. Another sign of failure is if the system partially or totally hangs, e.g. one or more processes never complete.

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. Use a large number of total iterations! You should be able to **demonstrate failure** by deliberately breaking something in your implementation, perhaps by reversing the order of two locking operations, or neglecting a lock. 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, and output showing detection of failure when you deliberately broke your synchronization. The output may be very verbose; you may trim the uninteresting stuff with an appropriate annotation.