Synchronization

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. In effect, we are simulating the synchronization issues that an actual kernel would have, but from the relative comfort and safety of user-land.

You will be building in layers: first build a spinlock mutex (using an atomic TAS function that I provide), then after testing it, build our customized version of Condition Variables using only the spinlock primitive for locking. Finally, build a FIFO using your condition variables library, and build a test framework to prove that your entire implementation is correct.

Problem 1 -- 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). Make sure you use the correct one for your architecture. Use it with a makefile or directly with gcc, e.g. `gcc spinlock.c spintest.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. Note that the tas function is defined for a single char, since it uses the XCHGB (exchange Byte) opcode, but it returns an int. The use of the `volatile` keyword tells the compiler to avoid optimizing memory accesses to the lock.

Now, implement a tiny library `spinlock.c` and associated header file `spinlock.h` using this provided TAS primitive. Create a data type such as `struct spinlock` which will contain the char that is used by TAS as the primitive lock. For debugging purposes, your struct could also contain the pid of the current holder of the lock, or a count of how many lock/unlock operations were performed, etc. Provide two functions in the library:

```
void spin_lock(struct spinlock *l);
void spin_unlock(struct spinlock *l);
```

These are *spin* locks meaning there is no sleep/wakeup or other mutex contention other than spinning on TAS until the lock is clear. Clearly, as explained in lecture notes, the spin lock only makes sense when the period of contention is brief. **NOTE:** Your performance may improve on Linux systems if you use the `sched_yield` system call inside the spin wait loop. You will be trading off useless user-mode CPU cycles for the expense of a system call. Try it.

Problem 2 -- Test the test-and-set

Now we can build a simple testing framework just to prove that our spinlocks work. Write a simple program that creates a shared mmap region (use MAP_ANONYMOUS | MAP_SHARED), spawns a number of child processes (accept the number as a command-line argument) each of which does something non-atomic (e.g. incrementing an integer in that shared memory area). Specify the number of iterations per child process as a command-line argument.

Show that without protecting the non-atomic operation, incorrect results are obtained (e.g. the final value of the number does not equal nchild * niter). Show that with a spin lock protecting the atomic operation, correct results are consistently obtained. Run a number of trials with at least as many child processes as there are CPUs/cores on your PC, or 8, whichever is smaller, and a sufficient number of total iterations to demonstrate failure when it is expected. Be mindful of silly things like overflowing a 32-bit int counter!

Problem 3 -- Implement Condition Variables

Now create a module called `cv.c` with associated header file `cv.h`. It will provide a modified and simplified implementation of Condition Variables (the lecture notes give examples using the POSIX Threads Library pthreads) and will rely solely on spinlocks as the locking primitive, plus certain UNIX system calls for blocking and unblocking as described below. The following 4 functions must be written:

```
void cv_init(struct cv *cv);
/* Initialize any internal data structures in cv so that it is ready for
 * use.  The initial condition is that nobody is waiting for this cv.
 * You can probably arrange your struct cv so that all-0 bytes is
 * the initialization condition.
 */

void cv_wait(struct cv *cv, struct spinlock *mutex);
/* This will be called with the spinlock mutex held by the caller (otherwise
 * results will be undefined).  Atomically record within the internals
 * of cv that the caller is going to sleep, release the mutex, and
 * go to sleep (see text below).  After waking up, re-acquire the mutex
 * before returning to the caller
 */

int cv_broadcast(struct cv *cv);
/* Wake up any and all waiters (sleepers) on this cv.  If there are no waiters
 * the call has no effect and is not "remembered" for the next time that
 * someone calls cv_wait.  cv_broadcast should be called with the same mutex
 * held that protects cv_wait, as discussed in lecture notes under "Lost
 * Wakup", but note that cv_broadcast does not take a mutex as a parameter.
 * Return value: the number of sleepers that were awoken.
 */

int cv_signal(struct cv *cv);
/* Exactly the same as cv_broadcast except at most one sleeper is awoken.
 * Your choice how to pick which one if more than one candidate
 */
```

The condition variables create blocking semantics (sleep and wakeup.) To implement this, you will use the `sigsuspend` system call in `cv_wait` which will block that process until a signal is received. You can think of this as analogous to a physical CPU entering a power-save mode because it has no useful work to do until the condition that it is waiting for is satisfied, and the signal as analogous to an inter-processor interrupt from the waker CPU to the sleeper. I recommend using `SIGUSR1` for this signal. `sigsuspend` has the useful semantics guaranteed by the kernel that it atomically changes the blocked signals mask to the requested value and puts the process to sleep. This allows us to keep `SIGUSR1` blocked, and to un-block it just when we use `sigsuspend`. In this way, we don't have to worry about the race condition of the signal coming in just as we are going to sleep and missing it. You need to establish a signal handler (because IGNoring the signal causes sigsuspend not to wake up, and the DeFauLt behavior is to terminate on SIGUSR1) but it need not do anything. The signal handler could be established in your cv module, or in the testing framework.

Note however that there are additional challenges. You will almost certainly need to introduce another spinlock

which is internal to the `struct cv` to protect its innards during critical non-atomic operations, such as putting the caller on the wait list. You may wish to introduce other debugging counters, such as the total number of waits, wakeups and spurious wakeups (wakeup when nobody is sleeping). **Static limits:** to simplify coding, you can have a static limit `CV_MAXPROC` which is the maximum number of these simulated virtual processors that your library needs to handle. Define this to be at least 64. **Malloc forbidden!** All parts of the `struct cv` must be in the shared memory region. Do not use malloc since this will allocate memory from the heap of the individual child process, which will not be shared. By using a static limit, you can avoid all dynamic allocation within your cv module. The initial allocation of the shared mmap region is handled by the testing program, not your library.

It is suggested that after you code up this condition var 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 4 -- A FIFO using condition variables

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 with the spinlock and condition variable modules developed above.

```
void fifo_init(struct fifo *f);
/* Initialize the shared memory FIFO *f including any required underlying
 * initializations (such as calling cv_init).  The FIFO will have a static
 * fifo length of MYFIFO_BUFSIZ elements.   #define this in fifo.h.
 * A value of 1K 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. (i.e. block until !full)
 */

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.  (i.e. block until !empty)
 */
```

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. Since reading and writing the FIFO involves non-atomic operations, you will need one or more spinlock mutexes to protect that. You could use the same mutex that protects the condition variable.

Problem 5 -- 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)

TIPS: receiving a writer's stream out-of-order indicates improper locking protection of the FIFO during read/write operations. Failure of all the writers to run to completion without hanging and/or failure of the reader to see all of the streams complete indicates an issue with the full/empty conditions and/or the condition variables module.

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 spinlock.[ch] cv.[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.

```
#Sample output....your output is not required to match this!
$ time ./ftest -w40 -n30000
Beginning test with 40 writers, 30000 items each
Writer 0 completed
Reader stream 0 completed
Writer 39 completed
Reader stream 39 completed
Writer 13 completed
Reader stream 13 completed
Writer 5 completed
Reader stream 5 completed
Reader stream 38 completed
All streams done
Waiting for writer children to die

real    0m2.275s
user    0m4.145s
sys     0m11.758s

#This was using sched_yield in the spinlock loop, hence the high systime
#Note that U+S > R.  This was run on a system with 8 independent
#execution units (what Linux calls "cpus")  and (U+S) ~= 8R
#system load from other tasks was low at the time of execution
```