Problem 1 -- Short Answer

Below is a function which inserts list element `what` after existing element `where` in a singly-linked list.

```
struct ll {
          struct ll *fwd;
          /* and other stuff */
};

void ll_insert(struct ll *where,struct ll *what)
{
          what->fwd = where->fwd;
          where->fwd = what;
}
```

A) Discuss why this function is not safe in a concurrent situation, i.e. where two or more threads try to execute `ll_insert` at the same time. Give specific examples of what can go wrong.

B) Modify the code above to make it safe for the STU case (as defined in the lecture notes) where our concern is only with a signal handler.

C) Now modify the original code in a different way so it can be safe for the MTU case, e.g. a multithreaded process, or multiple single-threaded processes sharing a data structure in shared memory.

Problem 2 -- A multi-layered synchronization programming problem

In this problem, you will explore synchronization issues on a simulated multi-processor, shared-memory environment. You will build a spinlock primitive using an atomic TAS instruction, then use that, along with signals which simulate interrupts, to build a semaphore library implementing the basic P,V and TRY operations. Finally, you will stress-test your semaphore library by an exercise establishing multiple simultaneous invocations of the P and V operations on the same semaphores.

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`, where `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 shellgame.c sem.c spinlock.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, while a non-zero value means locked. `tas` will atomically test the lock, and if it is currently unlocked, will lock it by setting it to 1. Since `tas` returns the *previous* value of `*lock`, this function returns 0 when the lock was acquired by the caller, and non-zero if the lock was already locked by another task.

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 the previous part. **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.

Note that you are required to implement all four operations above correctly, even if you do not wind up using all of them in the next part.

Problem 2C -- The Shell Game

No, we are not revisiting PS3! A well-known street con game known as the ‘‘shell game’’ or its cousin ‘‘3-card Monte’’ will be simulated using semaphores. There are three semaphores: A, B, C, representing the 3 shells, and the count of each semaphore represents how many (if any) pebbles are under the respective shell. There are 6 tasks which wait for the simulated pebble to appear under a particular shell and then move it to another shell. The choice of which is the "from" and "to" shell is fixed for a given task. E.g. task #1 waits on semaphore A, and moves the "pebble" to B. Task #2 moves from A to C, and so on.

We initialize the game by placing a certain number of pebbles under each shell. To be prototypical, that would be 1, although we can experiment with other numbers. If a task wants to move a pebble but the "from" shell is empty (i.e. the semaphore count is 0) then it must wait, because obviously we can't have a negative number of pebbles. On the other hand, in this simulation there is no limit to the number of pebbles under a given shell (i.e. the semaphore count has no upper bound), which might be an imaginative stretch, but just let it go! Now we can reason that if each task attempts the same number of moves and then exits, the system can not deadlock: there will always be at least one shell with at least one pebble, therefore at least one task will always be able to proceed. Further, after all tasks have concluded, the same number of pebbles is under each shell as when the game started. This is because each task has a complementary task which makes the opposite move, e.g. if task 1 moves A to C, task 5 moves from C to A.

This silly game is a simplification of problems which actually occur in synchronization. For example, we could think of the pebbles as network packet buffers, and the shells as various protocol layers. Our purpose is to use it to provide a real-world test of your semaphore library. For this last part of the assignment, write a program which implements this shell game, accepting two parameters, e.g. `shellgame 3 20000` initializes each semaphore to 3 and then spawns the 6 tasks, each of which makes 20000 moves.

A suggested structure of your shellgame program is a main process which establishes the shared mmap region and initializes everything, and then spawns 6 children. After this, the main process waits until each child has exited correctly.

Instrumentation: To gain further insight into the operation of the simulation, add instrumentation to your semaphore library: Within each semaphore, maintain an array of counters which tracks, for each virtual processor number, how many times it went to sleep waiting for that semaphore, and how many times that virtual processor number was woken up while waiting on that semaphore. Also keep track of how many times the signal handler was invoked in each task. Note that these numbers could wind up being either less than or greater than the number of iterations, depending on the amount of "contention" for the semaphore. These ratios might change from run to run and will certainly change depending on system load, the number of physical CPUs that you have, etc.

Here is a sample run (of course, your output can and will differ)

```
VCPU 0 starting, pid 19438
VCPU 1 starting, pid 19439
VCPU 3 starting, pid 19441
VCPU 2 starting, pid 19440
VCPU 4 starting, pid 19442
Main process spawned all children, waiting
VCPU 5 starting, pid 19443
Child 1 (pid 19439) done, signal handler was invoked 15097 times
VCPU 1 done
Child pid 19439 exited w/ 0
Child 4 (pid 19442) done, signal handler was invoked 15616 times
VCPU 4 done
Child pid 19442 exited w/ 0
Child 0 (pid 19438) done, signal handler was invoked 16472 times
Child 2 (pid 19440) done, signal handler was invoked 28008 times
VCPU 0 done
VCPU 2 done
Child 3 (pid 19441) done, signal handler was invoked 27737 times
Child 5 (pid 19443) done, signal handler was invoked 16357 times
VCPU 3 done
VCPU 5 done
Child pid 19438 exited w/ 0
Child pid 19440 exited w/ 0
Child pid 19443 exited w/ 0
Child pid 19441 exited w/ 0
Sem#     val      Sleeps   Wakes
0        2
 VCPU 0           16472    46112
 VCPU 1           15097    47489
 VCPU 2           0        0
 VCPU 3           0        0
 VCPU 4           0        0
 VCPU 5           0        0
1        2
 VCPU 0           0        0
 VCPU 1           0        0
 VCPU 2           28008    58402
 VCPU 3           27737    56901
 VCPU 4           0        0
 VCPU 5           0        0
2        2
 VCPU 0           0        0
 VCPU 1           0        0
 VCPU 2            0        0
 VCPU 3            0        0
 VCPU 4            15616    50182
 VCPU 5            16357    42895
```

I ran this on an 8-core system under light load, and all debugging messages were unbuffered (to stderr). Note how the tasks don't necessarily start in the order in which they were forked.