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.

Problem 1 -- The Spinlock Mutex

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 locktest.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 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.

Implement a **spin lock** using this atomic TAS, and use that spin lock as a mutex to help implement the other parts of this assignment. The following interface is suggested:

```
void spin_lock(int *lock);

void spin_unlock(int *lock);
```

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 spinlock 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 -- A slab allocator in shared memory

The Linux kernel uses a mechanism which was discussed briefly in Unit #5, known as a "slab" allocator. It is very useful when you have a number of fixed-size objects which need to be dynamically allocated and released. In this assignment, we'll use a slab allocator to allocate nodes of a circular, doubly-linked list, as described in part 4. A possible declaration of a slab is:

```
struct slab {
         char freemap[NSLOTS];
         struct dll slots[NSLOTS];
};
```

In this implementation, we'd use each char in the freemap to indicate if the given slot is in use or is free to be allocated. The Linux kernel saves space by using a bitmap. We'll also tremendously simplify the problem by allocating one fixed-size slab, and not allowing for additional slabs to be added on. Thus `NSLOTS` represents the maximum number of allocated objects.

Now, write two functions:

```
/* Allocate an object from the slab and return a pointer to it, or NULL
 * if the slab is full */
void *slab_alloc(struct slab *slab);

/* Free an object whose address is the second parameter, within the slab
 * pointed to by the first parameter.  Return -1 if the second parameter
 * is not actually the valid address of such an object, or if the object
 * pointer corresponds to a slot which is currently marked as free.
```

```
 * Return 1 on success
*/
int slab_dealloc(struct slab *slab, void *object);
```

The slab will be allocated at the start of your test program(s) by using `mmap` to create a shared, anonymous region. The address of that region will be the pointer to the slab, in other words, the region is the slab and the slab is the entire region.

Problem 3 -- Make the slab allocator thread-safe

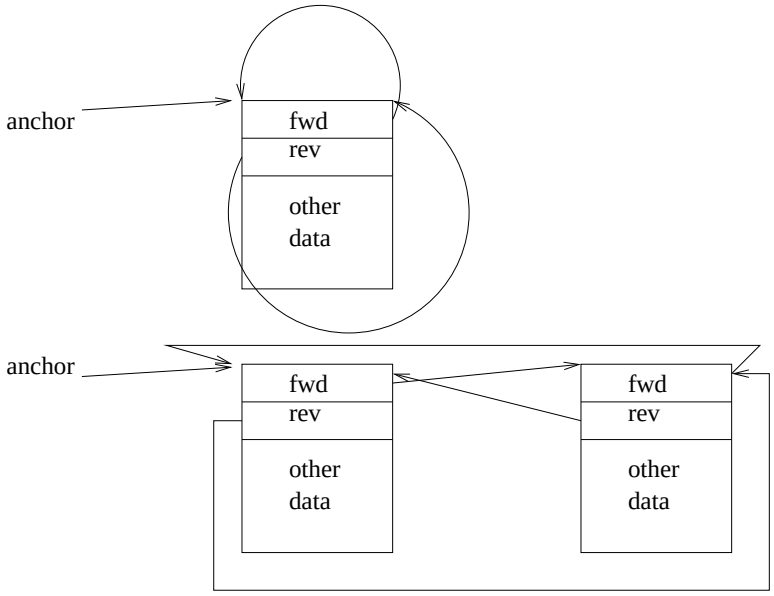
Now, introduce suitable mutex locking to eliminate any race conditions which might happen if multiple threads (aka tasks, implemented in our test environment by distinct processes) are simultaneously using the above two functions on the same slab. The locking should "live" inside of the slab, not in some external global variable, for reasons that *should* be obvious. It is not necessary to submit part 2 (the unsafe version) separate from part 3. I merely suggest part 2 as a milestone.

Problem 4 -- A Shared, Circular, Doubly-Linked List

We'll be working with a data structure that is often used inside the Linux kernel, known as the circular doubly-linked list. We define a node of this data structure (for the purposes of this exercise) as:

```
struct dll {
        int value;
        struct dll *fwd,*rev;
};
```

A property of the circular dll is that both the forward and reverse pointers always form a complete chain that closes upon itself. NULL pointers are never encountered, which simplifies the conditions that need to be considered in coding. Instead of a pointer to the start of the list, the list is defined by an "anchor" node, which is simply an ordinary node that does not store any meaningful data other than the forward/reverse pointers. An empty list is represented by the anchor node with both pointers pointing to itself. The diagram below depicts first an empty list, containing just the anchor and then a list with one meaningful element. *Note: unlike the Linux kernel implementation, the fwd and rev pointers point to the beginning of the appropriate struct dll, as illustrated*



Implement the following functions:

```
/* The first parameter is a pointer to the anchor of a circular, DLL
 * which is in ascending numerical order of the node->value.  The fwd
 * pointer of the anchor therefore points to the node with the lowest
 * value, and the rev pointer to the highest.  The second parameter
 * is the value of a new node, which is to be allocated using the
 * slab_alloc function, and inserted into the proper place in the list
 * The return value is a pointer to the inserted node, or NULL on failure
*/
struct dll *dll_insert(struct dll *anchor,int value,struct slab *slab);

/* Unlink the given node and free it using slab_dealloc.  Note that while
 * it is not strictly necessary to supply the anchor, having it might be
 * helpful, depending on the locking strategy you employ.  You can
 * simplify the problem by assuming  that the given node is in fact
 * contained on the given list.
 */

void dll_delete(struct dll *anchor, struct dll *node,struct slab *slab);

/* Find the node on the specified list with the specified value and
 * return a pointer to it, or NULL if no such node exists.  Optimize
 * the search with the knowledge that the list is stored in sorted order
 */
struct dll *dll_find(struct dll *anchor,int value);
```

The implementation above must be thread-safe! You must introduce appropriate locking to ensure that simultaneous/overlapping execution of any of the above functions will not result in a corruption of the data structure, or erroneous operations such as following a bad pointer or getting stuck in an endless loop.

Problem 5 -- Testing

Create a test framework for the above, which allocates a single slab (using the shared, anonymous mmap region), forks off a reasonable number of children, and then for each child, for some large number of iterations, performs a series of randomly-chosen insert, delete or find actions, with randomly chosen values. Suggestion: keep the range of random values small enough that the dll_find functions will actually find something most of the time. To further stress-test the code, set NSLOTS of the slab small enough that at least some of the inserts will fail temporarily for lack of space (if this happens, do not exit the entire test program! Just continue with your random action testing) At the conclusion of this stress test, you could walk the list (as a single-threaded task) and verify that the list is still intact and sorted.

Problem 6 -- Extra Credit +2 pts

Create an implementation of "seqlock" optimistic synchronization, built on top of your spinlock mutex. I don't necessarily recommend using the "X86 tricks" that the Linux kernel does. Use optimistic synchronization to improve the performance of the conflict between the dll_find function and the dll_insert and dll_delete functions. Your implementation must guarantee that if task A begins a find and task B has an overlapping insert or delete, that task A observes the effect of task B's action. Also, consider making the period of lock contention (the critical region)

among competing tasks as brief as possible, consistent with safety. Keep statistics to learn how often the optimistic method requires a retry. Compare the performance to that of the strict mutex-based approach.