Synchronisation 2

Reference Counting

You are to implement a few functions and a collection of references, with each entry of the collection containing a reference and counter (however you are free to implement this in another way if you feel, as long as it is flexible). Reference may also contain a dependency on another reference (parent) that they are attached to and that if the parent reference is deallocated then the reference counter for the dependent objects must be decremented.

```
/**
 * Request an allocation of memory based on the size parameter
 * and specify if it is dependency of another reference. If the dep argument
 * is NULL then it is a root and does not  depend on any other object.
 */
void* new_ref(size_t size, void* dep)

/**
 * Increments the reference count for the given reference.
 * If the reference is not valid, NULL should be returned
 * If the reference is valid, the same reference should be returned
 * and the counter incremented
 * If NULL is passed, NULL should be returned and no increment operation occurs
 */
void* inc_ref(void* ref);


/**
 * Decrements the reference count for a given reference.
 * If the reference is not valid or NULL, NULL should be returned
 * and no decrement operation should occur.
 * If the reference is valid, the reference should be decremented,
 *     If the counter is 0, the allocation should be deallocated
 *          and NULL returned.
 */
void* dec_ref(void* ref);
```

- What purposes of reference counting?
- Produce an example where reference counting would prevent unintentional deallocation of memory
- When designing concurrent data structures, why would reference counters be important?

Atomic Reference Counting

After implementing a non-atomic reference counter, implement an atomic reference counter. Switch the counter variables to be atomic integers instead of regular integers and use the atomic operations

- Mix an existing data structure with atomic reference counting and observe the benefits in a multi-threaded context.
- What other method could be used and also incorporate the benefits observed?

Atomic Stack

Construct a linked stack data structure which is where the last node pushed wouldbe the first node to be popped. Try and make your stack a generic stack that can be associated with any data type.

```
struct stack_node;
struct stack;
/**
* @return stack, returns a new stack object on the heap
*/
struct stack* stack_new();
/**
* Pushes a new object on the stack, this will be at the
* top of the stack.
* @param stack
* @param data
*/
void stack_push(struct stack*, void* data);
/**
* Pops the last element from the top of the stack.
* @param stack
* @return object, object at the top of the stack
*/
void* stack_pop(struct stack*);
/**
* Destroys the stack, deallocates any memory associated with it.
* @param stack
*/
void stack_destroy(struct stack*);
```

Once you have constructed stack, consider how you can apply atomic operations to the stack? How could you make your stack usable from multiple threads without using locks.

- You will need to use a `atomic_compare_exchange_*` function for this task
- What should you do if the operation fails?

- Discuss how you would implement the push and pop operation of your program

Read-Write Locking On Dynamic Array

You have been tasked with implementing a read-write dynamic array. Any add or remove function will be a write operation while a get function will be considered a read operation.

You will need to arrange different test cases that will look into different read-write ratios. Your test cases should check that you can correctly add, remove, insert and retrieve elements from the dynamic array and your functions are thread safe.

Afterwards, implement benchmarks with the following read-write ratios with your data structure and compare against your coarse grained implementation. Consider giving your data structure a set number of elements at the beginning.

- 0% Write, 90% Read
- 40% Write, 60% Read
- 50% Write, 50% Read
- 60% Write, 40% Read
- 90% Write, 10% Read

```
struct dynamic_array;
struct dynamic_array* dynamic_array_new(size_t initial_capacity);
void dynamic_array_add(struct dynamic_array *ary, void*value);
void* dynamic_array_get(struct dynamic_array *v,
size_t index);
void* dynamic_array_remove(struct dynamic_array *ary,
size_t index);
void dynamic_array_destroy(struct dynamic_array* ary);
```

Fine-Grained Locking - Hand-over-Hand Locking - Optional

In a previous tutorial, you were asked to construct a simple mutex around a linked list, however, we want to apply more fine-grained locking onto our linked list.

Add a mutex field to each node and ensure that on any access of a linked list node, the predecessor is also locked and updated accordingly.

- For any get/read operation, the value returned is assumed to have been borrowed. Discuss with your tutor the lifetime of this value, particularly what would happen to the value if we were to destroy the linked list.

Have a further discussion regarding how we could amend the lifetime issue of the value upon destruction of the linked list.

- For any removal or put operation, the nodes must be locked for the modification to ensure that nodes
- destroy function are assumed to not be thread-safe and requires the user to ensure that other threads are no longer modifying the data structure.