```
//370 Assignment 2 Answers - MBEA966
```

Question 9:

To execute assigned tasks, a queue structure is implemented which allows the dispatch queue to iterate through tasks. New tasks are appended to the tail of the queue and tasks to execute are taken from the head of the queue. Priority of the queue is strictly FIFO, this means that the head of the queue is the task which was most recently added out of all tasks remaining on the queue. Within a task structure, shown in Figure 1, a field called *next_task* stores a reference to the next task in the queue. A dispatch queue, structure shown in Figure 2, stores a reference to the task that is currently at the head of the queue. From the head, the dispatch queue can traverse to the next task by accessing the head's corresponding *next_task* field. Custom implementations of the standard queue access methods; push and pop were implemented as helper methods for easy access to queue elements.

```
typedef struct task {
    char name[64];              // to identify it when debugging
    void (*work)(void *);       // the function to perform
    void *params;               // parameters to pass to the function
    struct task *next_task;     // stores the task to be accessed next
    sem_t *task_semaphore;      // tracks whether a task has been completed
} task_t;
```

Figure 1: Task Structure

```
struct dispatch_queue_t {
    queue_type_t queue_type;            // the type of queue - serial or concurrent
    task_t *head;                       // pointer to the first task to be executed
    pthread_t *threads;                 // pointer to list of threads for queue to use
    sem_t *queue_semaphore;             // tracks how many tasks are ready to execute
    pthread_mutex_t *lock;              // used to a lock a queue to protect from concurrent access
    volatile int threads_executing;     // stores the number of threads currently executing tasks
};
```

Figure 2: Dispatch Queue Structure

As shown in Figure 2, a dispatch queue contains threads which are available to execute tasks on the queue. If the queue is serial, there is only a single thread available. If the queue is concurrent, there are as many threads as there are cores available. Since threads instantly begin executing a function upon creation, all threads are given a wrapper function to execute when initialised. The dispatch queue is passed as input to this function so that threads have access to necessary resources in the dispatch queue, including tasks needing execution, semaphores and locks. The role of the wrapper function is to constantly poll the queue semaphore, shown in Figure 3, which checks whether tasks need executing. When a task is added to the queue, the queue semaphore calls post which signals that a new task is ready to be executed. When any one of the threads becomes available and calls sem_wait, one (and only one) of those threads will extract the head of the queue and execute its associated task.

```
void *thread_wrapper_func(void *dispatch_queue) {
    dispatch_queue_t *queue_pointer = dispatch_queue;

    while (1) {
        //waits until there is a task for the thread to execute
        sem_wait(queue_pointer->queue_semaphore);
```

Figure 3: Wrapper Function Segment

In an environment where multithreading is occurring, concurrent access of resources can lead to unpredictable and undesired behaviour. To ensure that threads are not trying to currently add or remove elements from the queue, locking procedures are utilised. The field *lock* shown in the dispatch queue structure in Figure 1 was

used to execute the desired thread safe behaviour. In the event of multiple tasks being added to the queue in quick succession, multiple *sem_post* calls will be made which will cause multiple threads (if available) to attempt to pop off the task at the head of the queue. If the queue is not blocked from concurrent access, this could result in multiple threads accessing the same head value and thus a single task being executed unnecessarily, multiple times. As shown in the code snippet below in Figure 4, the queue is locked when the head of the queue is retrieved. This means that only one thread can pop off the queue at any given time, any thread trying to access the lock while it is in use will result in a blocking call until that lock becomes available. The same locking mechanism is also utilised when adding elements to the queue.

```
pthread_mutex_lock(queue_pointer->lock);

//increment the number of threads currently executing
queue_pointer->threads_executing++;
//retrieve the element at the front of the queue
task_t* current_task = pop(queue_pointer);

pthread_mutex_unlock(queue_pointer->lock);
```

Figure 4: Queue Locking Mechanism

The method *dispatch_async,* shown in Figure 5 adds a task to the end of the queue and immediately returns. To implement this, the given task was added to the queue using *push,* and *sem_post* was called by the queue semaphore to advertise the new task to execute to the queue's threads. If there is a thread waiting on the semaphore signal, it will retrieve the task from the head of the queue and execute it. Otherwise, the task will be retrieved and executed in the order it was added. No blocking code is added after the semaphore signal, so functionality will return immediately to the calling method.

```
int dispatch_async(dispatch_queue_t *dispatch_queue, task_t *task){
    //appends the given task to the queue
    pthread_mutex_lock(dispatch_queue->lock);
    push(dispatch_queue, task);
    pthread_mutex_unlock(dispatch_queue->lock);

    //increment semaphore count when a new task is added to the queue
    if (sem_post(dispatch_queue->queue_semaphore) == 0){
    } else {
        printf("sem_post unsuccessful\n");
    }
    return 0;
}
```

Figure 5: dispatch_async Method

When a task is dispatched using *dispatch_sync*, the code blocks until that task has been executed. To implement this, a semaphore is stored in the task. The value of semaphore is initialised to zero and after the input task has been added to the queue, the calling function waits on the value of the task's semaphore to change. The task's semaphore value will only change after it has been executed by a thread. After the execution of a task, the executing thread changes the status of the executed task's semaphore to 1 by calling *sem_post* which advertise that it has been completed. When this has been called, the previously blocking call to *sem_wait* in the calling function will stop blocking and control will return. Code is much the same as the above segment in Figure 5, although following the queue semaphore post call, there is a call to the task's semaphore to wait.

When implementing the function *dispatch_queue_wait,* the calling method should only return when all tasks on the queue have been executed. This means that the queue should be empty and all the threads should have completed their execution. To obtain this functionality, I stored a variable; *threads_executing,* shown in Figure 2, which shows the number of threads that are currently executing a task. This is incremented at the

start of a task execution and decremented afterwards. It is accessed using a lock so that threads cannot concurrently change the value. To check that there were no tasks left on the queue, all that was required was to check if the head of the queue was *NULL*. If the head has not been set, this means there are no tasks left to dispatch. This exit functionality is shown in the code segment below in Figure 6.

```
int dispatch_queue_wait(dispatch_queue_t *queue) {
    //poll to see if all threads have finished executing tasks and
    //there are no tasks left on the queue.
    while(1){
        if (queue->threads_executing == 0 && !queue->head){
            return 0;
        }
    }
}
```

Figure 6: dispatch_queue_wait Method

Question 11:

Test 4 - Concurrent

```
real    0m29.279s
user    0m54.177s
sys     0m0.211s
```

Test 5 - Serial

```
real    0m26.595s
user    0m49.488s
sys     0m0.136s
```

Timing of the test executions shows that the concurrent implementation takes longer than the serial. This seems like unexpected behaviour, as in the concurrent implementation tasks can run in parallel and serial must run one at a time. Logically this would suggest that by running tasks in parallel, the concurrent dispatch queue would be able to more quickly execute the queue of tasks. Since the task functions do not take long to execute, the overhead of concurrent threads communicating with each other is significant relative to the length of time taken to execute the tasks. In running a single thread, serial dispatch queues do not have the overhead of maintaining multiple threads concurrently. Over the course of the test, the effects of inter-thread communication in the concurrent implementation adds up and overall means it runs longer than the serial implementation in this scenario. Different results would likely be observed for tasks that take longer to execute as the effects of the inter-thread communication would be insignificant relative to the task execution times.