370 A2 Report:

(Cyrus Raitava-Kumar, crai897)

Question 9:

My implementation involves the usage of a singly-linked queue of 'nodes', which are structs that in and of themselves contain task structs. The head of this singly linked queue of nodes, is contained within the *dispatch_queue_t* struct.

Threads are created at first via the usage of a 'thread pool'; if the dispatch queue is serial, then tasks may only be done sequentially, and thus only one thread is needed for task execution (and therefore only one is spawned). In the case of a concurrent dispatch queue, my implementation calculates the number of existent cores/CPUs on the running machine, and spawns an equal number of threads.

These threads exist in the form of *dispatch_queue_thread_t* structs, that are spawned on creation of the dispatch queue itself. Upon creation, these *dispatch_queue_thread_t* structs contain references to pthreads, which are created and are assigned also within the *dispatch_queue_create()* method. These pthreads must have a function assigned to them at creation (to which they immediately execute); this comes in the form of the *thread_function()*, which has built into it a sem_wait() call, on the semaphore owned by the *dispatch_queue_t* struct itself. See below:

```c
// Wrapper function to aid in blocking until queue has a task/tasks to complete
void *thread_function(void *input)
{
    // Cast input to expected type (dispatch_queue_t)
    dispatch_queue_t *dispatchQueue = (dispatch_queue_t *)input;

    while (1)
    {
        // Wait until there is something on the queue to do (a sem_post() called)
        sem_wait(dispatchQueue->queue_semaphore);

        DEBUG_PRINTLN("========================\n");

        // Lock queue, and increment the number of executing threads field
        pthread_mutex_lock(dispatchQueue->lock);
        dispatchQueue->numExecutingThreads++;
        pthread_mutex_unlock(dispatchQueue->lock);

        DEBUG_PRINTLN("DOING TASK\n");

        // Pop the task off of the head of the queue
        // (note locking of queue occurs within pop() method)
        node_t *taskedNode = pop(dispatchQueue);
        DEBUG_PRINTLN("POPPED NODE HAS NAME: %s\n", taskedNode->nodeTask->name);

        // Save the task to execute
        task_t *task = taskedNode->nodeTask;

        // Execute work function of task
        taskedNode->nodeTask->work(taskedNode->nodeTask->params);

        // Use sem_post() to let dispatch_sync() method know execution of task has completed
        sem_post(task->taskSemaphore);

        DEBUG_PRINTLN("EXECUTED TASK W/ NAME: ");

        // Lock queue, and decrement the number of executing threads field
        pthread_mutex_lock(dispatchQueue->lock);
        dispatchQueue->numExecutingThreads--;
        pthread_mutex_unlock(dispatchQueue->lock);
    }
}
```

thread_function() blocks via the dispatchQueue's semaphore ,until a task has been added to the queue using the dispatch_sync() or dispatch_async() methods.

It is important to note that using the pop() and push() functions on the dispatch_queue_t structs, are *ensured* to be interleaved, as they utilise the lock field of the dispatch_queue_t. This ensures transactional consistency, in the face of concurrency.

When it comes to dispatching, a semaphore contained within the *dispatch_queue_t* struct is used to notify threads that a non-null task exists and has been safely added to the queue of nodes within the dispatch queue. Prior to this notification, the threads spawned on creation of the dispatch queue, block *within* the while loop shown above, on the *sem_wait()* line.

The notifying of threads of this, is triggered by the usage of *dispatch_async()* or *dispatch_sync()*, as shown below:

```c
// Method to synchronously dispatch task onto queue
int dispatch_sync(dispatch_queue_t *queue, task_t *task)
{
    DEBUG_PRINTLN("========================\n");
    DEBUG_PRINTLN("BEGINNING SYNC DISPATCH")

    // Push task onto queue
    push(queue, task);

    // Let all threads waiting on queue, know that there is another task on the queue
    sem_post(queue->queue_semaphore);

    // Wait until the task itself has been completed, to return
    sem_wait(task->taskSemaphore);
    return 0;
}
```

```c
// Method to asynchronously dispatch a task to the queue
int dispatch_async(dispatch_queue_t *queue, task_t *task)
{
    DEBUG_PRINTLN("========================\n");
    DEBUG_PRINTLN("BEGINNING ASYNC DISPATCH")

    // Push node onto end of queue
    push(queue, task);

    DEBUG_PRINTLN("POSTING TO SEMAPHORE, THAT TASK HAS BEEN QUEUED\n");

    // Post to the semaphore, that a task has been queued
    sem_post(queue->queue_semaphore);

    DEBUG_PRINTLN("ASYNC FINISHED PUSHING A NODE\n");
    return 0;
}
```

Both of these methods are used to add tasks to the queue dispatch queue in different ways, and both use the queue's semaphore, to let all existent threads know that an undone task exists on the queue.

In relation to contention and concurrency, semaphores are built naturally to be threadsafe; thus, when multiple threads are vying to execute a task whose existence was notified by a call to dispatch_async() or dispatch_sync(), one can be sure that inconsistent concurrency is highly unlikely, if not impossible.

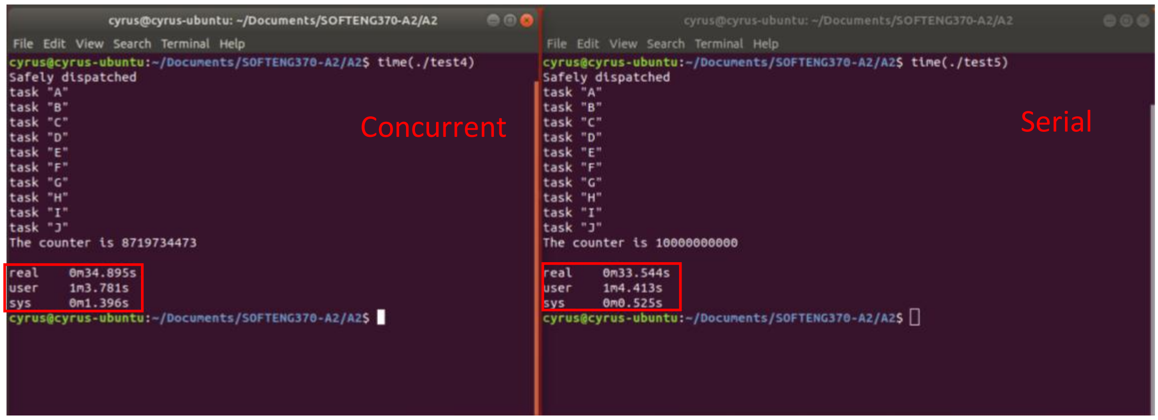
Furthermore, the usage of locks on the queue (characterized by the existence of a pthread_mutex_t lock within the dispatch_queue_t struct), ensure interleaving of access to the singly-linked list of nodes, from either the pop() or push() methods. This ensure that no tasks may be missed or left out, and no *lost updates* may occur, in the form of overwriting of popping the head off, or pushing another node onto the queue. See below:

```c
// Method to pop task/node struct off of beginning of singly-linked list
node_t *pop(dispatch_queue_t *dispatchQueue)
{
    DEBUG_PRINTLN("========================\n");

    // Locking queue, to pop off head
    pthread_mutex_lock(dispatchQueue->lock);

    // If the queue's head doesn't exist, unlock the queue and return NULL
    if (!dispatchQueue->head)
    {
        DEBUG_PRINTLN("ATTEMPTING TO POP TASK OFF QUEUE, BUT HEAD WAS NULL\n");
        pthread_mutex_unlock(dispatchQueue->lock);
        return NULL;
    }

    // Set the node to return, as the current head of the queue
    node_t *result = dispatchQueue->head;

    // If there are 2 or more nodes, set the second node to be the new head
    if (dispatchQueue->head->nextNode)
    {
        dispatchQueue->head = dispatchQueue->head->nextNode;
        DEBUG_PRINTLN("SET SECOND NODE TO BE HEAD\n");

        // Unlock the queue, and return result
        pthread_mutex_unlock(dispatchQueue->lock);
        return result;
    }
    else
    {
        DEBUG_PRINTLN("SET HEAD TO BE NULL\n");

        // If there is no second node, then you are popping off the last node,
        // and must set the head to now be null
        dispatchQueue->head = NULL;

        // Unlock queue and return result
        pthread_mutex_unlock(dispatchQueue->lock);
        return result;
    }
}
```

```c
// Method to append task/node struct combination onto end of singly-linked list
void push(dispatch_queue_t *dispatchQueue, task_t *newTask)
{
    DEBUG_PRINTLN("========================\n");

    // Lock the queue, prior to pushing anything onto it
    pthread_mutex_lock(dispatchQueue->lock);
    DEBUG_PRINTLN("LOCKED QUEUE NOW");

    DEBUG_PRINTLN("ATTEMPTING TO PUSH NODE W/ TASK NAME: %s\n", newTask->name);
    // If the queue's head doesn't exist, set it to be the newly made node
    if (!dispatchQueue->head)
    {
        dispatchQueue->head = node_create(newTask, NULL);
        DEBUG_PRINTLN("HEAD DIDN'T EXIST, SO HEAD IS NOW TASK W/ NAME: %s\n", dispatchQueue->head->nodeTask->name);

        // Unlock the queue, and return
        pthread_mutex_unlock(dispatchQueue->lock);
        return;
    }

    // If the head does exist, follow the pointers of nextNode
    // down the singly-linked list, to find the current tail
    node_t *position = dispatchQueue->head;
    while (position->nextNode)
    {
        position = position->nextNode;
    }

    // Append task wrapped in node, onto tail of singly-linked list
    // (note the nextNode pointer of the tail is set to NULL)
    position->nextNode = node_create(newTask, NULL);

    // Unlock the queue, and return
    pthread_mutex_unlock(dispatchQueue->lock);
    return;
}
```

Question 10:

Having run the *time* command on both tests 4 and 5, my results are the following:



Analysing this, one could be surprised; the serial dispatch queue, turns out to run with a *faster* run-time, than that of the concurrent dispatch queue. It would be normal for one to assume that with more threads, and the ability to execute multiple tasks simultaneously, concurrency would trump serial queueing.

Upon further inspection, one may realise that the *overhead* in communicating with multiple threads, and simultaneously ensuring data integrity, provides a far greater lag in communication and execution, than concurrency does provide a speed-up. This is especially true depending on how *granular* the tasks one is executing are, and - in the case of these tests -proves itself as such. A large pool of very small tasks (in terms of execution time), can thus be executed *faster* as a sum, through a serial queue, as opposed to a parallel queue.