SOFTENG 370 A2 Answers

Question 9

Concurrent and serial dispatch queues are treated uniformly in my implementation, with the only difference between them being the number of threads in the thread pool (1 for serial, number of cores for concurrent). Tasks are not actually “dispatched” from the queue, rather each thread in the thread pool shares access to the same queue and extracts tasks from the top of the queue in a synchronised fashion.

Code snippet that gets executed by each thread in the thread pool

```
while (!queue->shutdown) {
    pthread_mutex_lock(&queue->queue_mutex);
    // wait for an item to be added to the queue if it's currently empty
    // need a while loop here in case of spurious wake-ups from cond_wait
    while (!queue->shutdown && !queue->waiting && queue->front == NULL) {
        pthread_cond_wait(&queue->queue_cond, &queue->queue_mutex);
    }
    // if the queue is shutdown, or is waiting and the queue is empty, the
    // thread should stop running
    if (queue->shutdown || (queue->waiting && queue->front == NULL)) {
        pthread_mutex_unlock(&queue->queue_mutex);
        break;
    }
    dispatch_queue_item_t *item = pop_item(queue);
    pthread_mutex_unlock(&queue->queue_mutex);
    item->task->work(item->task->params);
```

Code snippet for `dispatch_async`

```
pthread_mutex_lock(&queue->queue_mutex);
// ignore the task if the queue is waiting or shutdown
if (!queue->shutdown && !queue->waiting) {
    task->type = ASYNC;
    push_item(queue, task);
    pthread_cond_signal(&queue->queue_cond);
}
pthread_mutex_unlock(&queue->queue_mutex);
```

To synchronise access to the queue, a mutex (`queue->mutex`) is used so that different threads can’t extract tasks from the queue at the same time. This mutex is also used in conjunction with a condition variable (`queue->queue_cond`) so that threads can be signalled when a task is added to the queue, or when an event occurs (e.g. shutdown or waiting). The threads in the pool wait on this condition variable when the queue is empty so that busy waits are avoided. As seen above in the snippet for `dispatch_async`, the `queue_cond` variable is signalled whenever a task is added to the queue, and this causes exactly one thread in the pool to wake up (if any threads are waiting on `queue_cond`). Once a thread wakes up from the `pthread_cond_wait` call, it extracts the top task from the queue, executes it, and cleans up all memory allocated to the task.

Question 11

(2 core system)

test4		**test5**	
real	0m29.812s	real	0m21.622s
user	0m55.982s	user	0m21.445s
sys	0m0.408s	sys	0m0.084s

Test 4 takes more time to execute than test 5 even though the tasks are running in parallel. Also, the user time for test 4 is much higher than the user time for test5.

The disparity between the execution time between test4 and test5 can be explained by the contention that occurs when multiple threads try to access the same dispatch queue. Since the synchronisation mechanism used involves locking and unlocking a mutex, both the overhead involved in using a mutex and the fact one thread will block while waiting for the other to unlock the mutex, causes test4 to run ~8 seconds slower than test5, which doesn't have these contention problems.

Regarding the large difference in user time between test4 and test5, this is due to the fact that the time command measures the total time spent by all CPU cores (in user mode) while running the executable. Since test4 involves 2 separate threads that run on the 2 cores in the system, we see that user time is approximately double the real elapsed time. Since test5 only has one thread in the thread pool, user time approximately equals the real elapsed time.