Project Report

Assignment 5
Name: Thanaji Rao Thakkalapelli

Part 0

In my previous implementation I used to allocate head and tail which cannot be freed. Now I removed all not needed allocations first.

Added threadmap.c with instructions and created simple functions with three threads to check it is working properly.

Part 1:

Created new kernel thread with clone with the instructions given.

Faced segmentation faults since the thread switching not exactly working for different kernel threads.

Part 2

Implemented spin lock and unlock tested with given spinlock_test.c got success.
In this part I got same issues as above since we didn't jump threads and use test and set locks.

Part 3

For this part I implemented two kernel threads, runs on our application.

With the given instructions I changed Scheduler.c, assembly program switch.s to jump to thread wrap.

Used spin locks to protect
scheduler_begin
scheduler_end
yield
thread_fork

I initialized new spinlock variable in my main.c program to protect printf.
Before protecting I faced issues printing characters interchanged and added to different output etc.

I wrote number of different threads and checked through GDB how kernel threads switching in between and executing same instruction twice, each kernel by once. With which there is race condition in output and some cases deadlocks occurred.

I wrote simple test function that runs by different threads and incrementing a global variable.

Which is there are 100 threads incrementing a shared variable and prints the value of shared variable after all threads completed.

So you can see there are different outputs, since there are two Kernel threads running on the application and critical section not protected. I tested with mutex locks. Since mutex locks are global locks, which are not protected with spin locks, we can see race condition in output.
Executing same program 40 times produces different results as shown below.

```
thakkal@ada:~/Desktop/OS/proj5/part3_1$ for i in `seq 40`; do ./a.out | tail -n
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 9
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 15
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 97
Shared Variable 97
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 3
Shared Variable 100
Shared Variable 100
Shared Variable 15
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 30
Shared Variable 100
Shared Variable 100
```

For this part my program executing properly and with tests we can understand there should be more implementation required to protect critical sections

Part 4:

Created a new spinlock variable to lock mutex variables

With instructions created a function block works as yield but make sures the lock on thread exists till the completion.

Since we locked mutex and the instructions are atomic in mutex lock.

Used same test case as part 3 and now I got same result in all 40 executions as shown below.

```
thakkal@ada:~/Desktop/OS/proj5/part4$ gcc -I ~kstew2/local/include switch.s scheduler.c queue.c main.c threadmap.c -lrt -g
thakkal@ada:~/Desktop/OS/proj5/part4$ for i in `seq 40`; do ./a.out | tail -n 1; done
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
Shared Variable 100
```

Part 5

Performance of application is not fixed and with the increase of kernel threads. the performance varied.

My test consist of calculating Fibonacci series and a producer consumer

So I tested with different number of kernel threads got time output as below. Since the spin is not fixed the result vary depends on the time takes each time spinning.

Which exactly explains that the execution time is not fixed.
It's varying, not increasing with number of kernel threads and nor decreasing.
Different number of executions shows different result.

2 kernel threads
real 0m0.007s
user 0m0.000s
sys 0m0.009s

4 kernel threads
real 0m0.008s
user 0m0.013s
sys 0m0.003s

8 threads
real 0m0.028s
user 0m0.149s
sys 0m0.009s

16 threads
real 0m0.011s
user 0m0.002s
sys 0m0.025s

In the current approach, there is a separation between identifying that the lock has been released and attempting to acquire the lock with test-and-set instruction. It allows more than one processor to see that the lock has been released. So all these processors pass the test and proceed to try a test-and-set instruction. Ideally, if one processor could notice the change that lock has been released and acquire the lock before any other processor committed to doing test-and-set, the performance would be better.

Cache copies of the lock value are invalidated by a test-and-set instruction even if the value is not changed. If this were not the case, invalidations would occur only when the lock is released and then again when it is reacquired.

Invalidation based cache-coherence requires O (P) of network or bus cycles to broadcast a value to P waiting processors. While solution to any of these three problems by itself would result in better performance, any single solution would still require bus activity that grows linearly with the number of processors.

Below algorithm cuts contention and invalidations by adding latency between retries, which provides good performance if there few number of spinning threads and delay is short, delay is long but there are many number of spinning threads.

```
Delay Only on Attempted Set
while(lock=BUSY or TestAndSet(lock)=BUSY)
begin
        while (lock=BUSY);   /* spin on read without delay */
        delay();              /* delay before TestAndSet */
end;
<criticial section>
```

Coming to next approach that generates contention while spin on read. This is good approach when there is just need to check lock less frequently.

```
While(lock=BUSY or TestAndSet(lock)=BUSY)
delay();
<criticial section>
```

With increase in user level threads working on single method, performance decreases. Delay is helpful when our program has big computation and working on large number of threads.