Computer Science and Engineering & Information Technology (2^{nd} Year B.Tech.) IIIT Kalyani, West Bengal

Operating System Lab (CS 411): (Spring: 2019-2020)

Assignment - 10 *Marks: 10*

Assignment Out: *27th March, 2020*

In this experiment you will try to implement a *lock* that will not only make the *critical section atomic* but instead of *busy wait* on it, it will suspend its thread after adding the thread ID to its queue.

1. Create the threads using `clone()` and use signals (assignment 9) to suspend and restart a thread.
2. There is a new complication. The operations on the queue of suspended threads should be *atomic* as more than one thread may try to update it concurrently. The *atomicity* of the operations on the queue is ensured by busy wait.
3. The shared data structure is simply a *global counter* initialized to *zero (0).* The operations are *increment* and *decrement.*
4. The *lock* will make the increment and decrement operations on the counter *atomic.* **Note:** introduce delay to magnify the possibility of race in the critical sections of increment and decrement operations.
5. The `main()` thread takes an input n and creates n number of threads that concurrently perform increment operations on the counter.
6. Similarly it creates another set of n number of concurrent threads to perform decrement operations on the counter.
7. The counter was *initialized* to **zero (0)** at the beginning.
8. At the end of execution of $2 \times n$ threads, the result should be **zero (0)** if the operations are atomic (using your lock). Otherwise it can be anything arbitrary.
9. Following are the suggested data structures.
   (a) The data structure for the queue is same as *assignment 9.*

   ```
   typedef struct node {
           int data;
           struct node *next;
   } node_t;

   class queue{
         node_t *front, *rear;
         public:
         queue();
         bool isEmptyQ();
         void addQ(int n);
         int deleteQ();
   };
   ```

   (b) The suggested data structure for the lock is as follows:

   ```
   typedef struct mylock_t{
           int mylock;
           int guard;
           queue q;
   } mylock_t;

   void mylockInit(mylock_t &, int); // 2nd param for initial value
   void mylock(mylock_t &);
   void myunlock(mylock_t &);
   ```

(c) You may put both of then in a header file `myLock.h` and implement in `myLock.c++`.

(d) The `int mylock;` field of the data type `myloc_t` is the actual lock variable.

(e) The `int guard;` is the *local lock* used to make the operations on `queue q;` atomic. This one is actually a *spin lock*.

(f) The operations on `mylock_t` are as usual. But they relay on our old (assignment 8)
`void tasLock(int *lp)`,
`void tasUnlock(int &lck)` and
`void tasInitlock(int &lck)`.

10. You should use a *Makefile* to compile your code.

11. Is your implementation correct? Try to find a possibility of race condition and magnify it if there is one!

Input/Output:

```
$ ./a.out
Enter a small +ve integer: 1
lock? (1/0)
1
Data: 0

$ ./a.out
Enter a small +ve integer: 1
lock? (1/0)
0
Data: 1000

$ ./a.out
Enter a small +ve integer: 4
lock? (1/0)
1
Data: 0

$ ./a.out
Enter a small +ve integer: 4
lock? (1/0)
1
Data: 0

$ ./a.out
Enter a small +ve integer: 4
lock? (1/0)
0
Data: -994

$ ./a.out
Enter a small +ve integer: 4
lock? (1/0)
0
Data: 905

$ ./a.out
Enter a small +ve integer: 10
lock? (1/0)
1
Data: 0

$ ./a.out
Enter a small +ve integer: 10
```

```
lock? (1/0)
0
Data: 1070

$ ./a.out
Enter a small +ve integer: 10
lock? (1/0)
0
Data: -1001
```