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

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

Assignment - 11 *Marks: 10*
Assignment Out: *3^{rd} April, 2020*

This is an experiment similar to assignment 10. The difference is it is not for threads but for processes. So we have to use shared memory and change the implementation of the queue of suspended processes as `new` is not available on shared memory.

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 process after adding the process ID to its queue.

1. Create processes using `fork()` and use signals (assignment 9) to suspend and restart a process.

2. There is a new complication. The operations on the queue of suspended queue should be *atomic* as more than one process 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 *counter* in the shared memory initialized to *zero (0)*. The operations are *increment* and *decrement* of the counter.

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()` process takes an input n and creates n number of concurrent processes that perform increment operations on the counter. Similarly it creates another set of n number of concurrent processes to perform decrement operations on the counter.

6. The final value of the counter should be **zero (0)** as it was *initialized* to **zero (0)** at the beginning.

7. At the end of execution of $2 \times n$ processes, the result should be **zero (0)** if the operations are atomic (using your lock). Otherwise it can be anything arbitrary.

8. Following are the suggested data structures.

   (a) The data structure for the queue (similar to our produce-consumer i problem) is.

   ```
   #define MAX 100
   #define ERROR (-1)
   #define OK 0

   class queue {
     private:
         int front, rear, count ;
         int data[MAX];
     public:
         queue();
         bool isEmptyQ();
         bool isFullQ();
         int addQ(int); // return -1 on error, 0 on success
         int deleteQ(); // return -1 on error
   };
   ```

   (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`. The implementation is 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)`.

9. You should use a *Makefile* to compile your code. Prepare a .tar file of all required files and send it.

10. As it was mentioned earlier that the suggested implementation is incorrect (see the Google sheet for discussion). Nevertheless I learn something while writing this code and I wish you too.

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: 999

$ ./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: 960

$ ./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: -1000

$ ./a.out
Enter a small +ve integer: 10
lock? (1/0)
0
Data: 1007
```