A report on measuring the average time that it takes for the operating system to lock and unlock a semaphore.

Shah Dhruv Amitkumar

(Redid: 819653114)

Course Number: COMPE 571

Professor Name: Dr. Yusuf Ozturk

Date Submitted: 10-12-2015

Contents

Introduction

The main purpose of the project is to measure the average time needed to lock and unlock the semaphore. Before we do that let us first understand why semaphores are used.

Consider a system consisting of n processes {Po,P1,.....,Pn-1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important characteristic of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections is mutually exclusive in time by the processes. Purpose of this is to be sure that the current sequence of instructions would be allowed to execute in order without preemption. In a uniprocessor environment if we could disallow interrupts to occur while a shared variable is being modified the problem would be solved. But in the multiprocessor environment, disabling interrupts could be time consuming so semaphores can be used here.

A semaphore is a synchronization construct that can be used to provide mutual exclusion and conditional synchronization. One of the main problems for the semaphores are they take some time for locking and unlocking the semaphore even if application has one process going on. If the application doesn't need synchronization and mutual exclusion then may be semaphores are not the right way to go in the application.

According to the application need, one has to decide whether one should use semaphores or not. Because semaphore will consume some time to lock and unlock. In this fast and growing world everyone wants every application to be as fast as possible. So one should include semaphore in application when synchronization is a big part of application otherwise application will consume more time compared to that application where semaphore is not used. And in this chapter, you will find out how to create semaphores and lock and unlock the semaphore. Also you will learn how to calculate the average time it takes for system OS to lock and unlock the semaphores.

Discussion on the methodology to solve the problem

Algorithm to find the average time between locking and unlocking semaphores is given below.
1) Create a System V semaphore set with a single semaphore.
2) Call gettimeofday() to get the start time.
3) Call semop() to lock the semaphore.
4) Call semop() to unlock the semaphore.
5) Repeat steps 3 and 4 several thousand times.
6) Call gettimeofday() to get the end time.
7) Calculate the average time using data collected above.

First of all, we will understand how to create semaphore to begin the process to calculate the average time for locking and unlocking of the semaphore.

```
#define SEM_KEY 0x1234
...
struct sembuf g_lock_sembuf[1];
struct sembuf g_unlock_sembuf[1];
```

The first 2 line of code creates two arrays of structures of type sembuf, each with one element. One will be used to perform a lock the semaphore and other one will be used to perform an unlock the semaphore. These structures are initialized in the main function.

```
g_lock_sembuf[0].sem_num = 0;
g_lock_sembuf[0].sem_op = -1;
g_lock_sembuf[0].sem_flg = 0;
 g_unlock_sembuf[0].sem_num = 0;
g_unlock_sembuf[0].sem_op = 1;
g_unlock_sembuf[0].sem_flg = 0;
```

There are three elements to the structure, sem_num, sem_op, and sem_flg. For both structures.
1.Sem_num is initialized to zero defining that we are operating on the first semaphore in the set.
2.Sem_flg is initialized to zero stating that no special action is to be taken here.
3.Sem_op is initialized to -1 for the lock and 1 for the unlock.

```
if( ( g_sem_id = semget( SEM_KEY, 1, IPC_CREAT | 0666 ) ) == -1 )
{
fprintf(stderr,"semget() call failed, could not create semaphore!");
exit(1);
 }

if( semop( g_sem_id, g_unlock_sembuf, 1 ) == -1 )
{
fprintf(stderr,"semop() call failed, could not initalize semaphore!");
exit(1);
}
```

Next, the semaphore with key SEM_KEY is created using the semget() call. The permissions are set with 0666. As stated previously, immediately after creating the semaphore it must be initialize by unlocking it. The semop() call does this.

```
if( semctl( g_sem_id, 0, IPC_RMID, 0) != 0 )
{
fprintf(stderr,"Couldn't remove the semaphore!\n");
exit(1);
}
```

And in last, the semctl() call removes the semaphore from the system.
Now we will talk a little about timeval_diff function which will use timeval structure and will return measured time in microsecond.

```
long long
timeval_diff(struct timeval *difference, struct timeval *end_time, struct timeval *start_time )
{
  struct timeval temp_diff;

  if(difference==NULL)
  {
    difference=&temp_diff;
  }

  difference->tv_sec =end_time->tv_sec -start_time->tv_sec ;
  difference->tv_usec=end_time->tv_usec-start_time->tv_usec;
```

```
 while(difference->tv_usec<0)
 {
  difference->tv_usec+=1000000;
   difference->tv_sec -=1;
 }

 return 1000000LL*difference->tv_sec+
            difference->tv_usec;
}
```

To calculate time difference between locking and unlocking the semaphore we have to call gettimeoftheday() to start time.

```
if( gettimeofday( &earlier, NULL ) == -1 )
          {
               fprintf(stderr,"Couldn't get time of day, exiting.\n");
               exit(1);
          }
```

From this function we can find the start time which is time before locking semaphore and then we can find the end time after unlocking semaphore after repeating locking and unlocking process several thousand times.

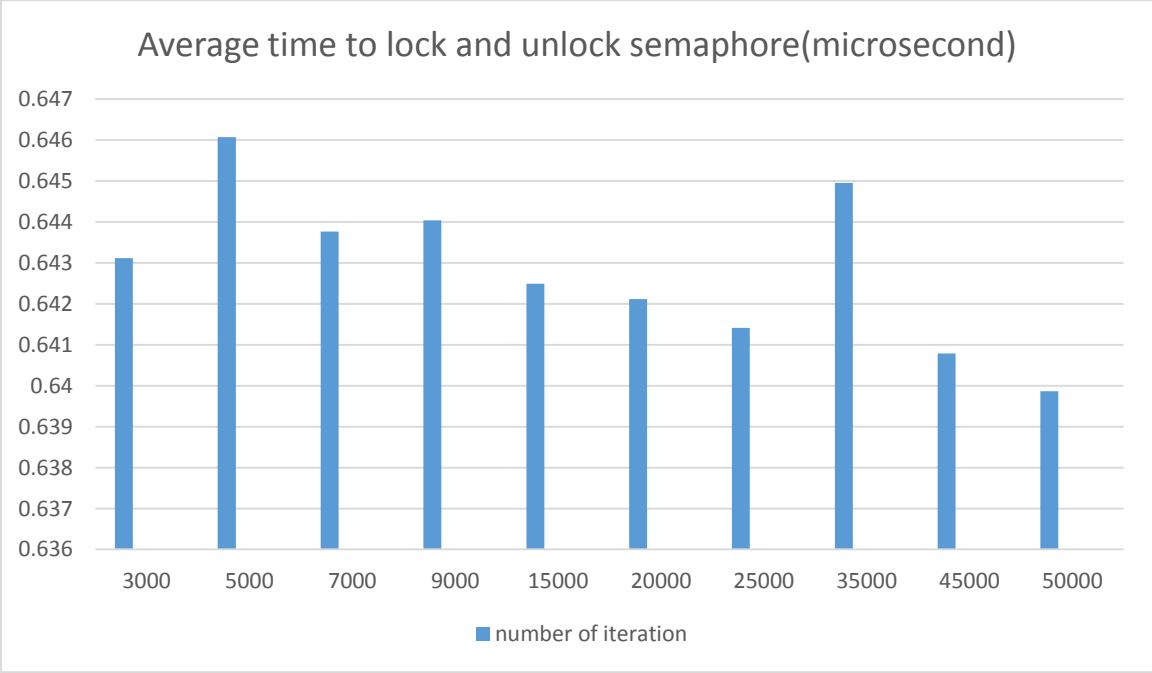
From the total time we get from the below equation, we just have to divide this number with the counter variable which is number of time repetition of locking and unlocking is done, we find the average time we need to calculate.

```
     n=timeval_diff(NULL,&l,&e);      //'e' is starting time and 'l'
                                      is ending time
     printf("\naverage: %f\n",n/count);   //calculating average time
```

Results of running program

1. Count vs Average time for operating system to lock and unlock the semaphore.

Count	Average time(Microsecond)
3000	0.643119
5000	0.646071
7000	0.643765
9000	0.644040
15000	0.642491
20000	0.642118
25000	0.641414
35000	0.644953
45000	0.640786
50000	0.639867



Analysis of Results

To gain average time it takes by system OS to lock and unlock the semaphore I did this experiment repeatedly. I tried to find out that with the increase in counter, output average time gets affected or not. With the experiments analyses I came to know that it does not matter if we increase or decrease the counter variable, output average time will be constant.

I did this experiments with the counter equals to 3000, 5000, 7000, 15000 and even for 50000. But the output average time remain same.

Average time = 0.64 microseconds

Here are some screenshots of output. i.e the average time of system OS to lock and unlock the semaphore.

Count = 3000

```
30 [volta]/home/student/dshah/dhruv/assi2> gcc -o assi2 assi2.c
31 [volta]/home/student/dshah/dhruv/assi2> ./assi2

average: 0.643119
32 [volta]/home/student/dshah/dhruv/assi2>
```

Count = 20000

```
22 [volta]/home/student/dshah/dhruv/assi2> gcc -o assi2 assi2.c
23 [volta]/home/student/dshah/dhruv/assi2> ./assi2

average: 0.642118
24 [volta]/home/student/dshah/dhruv/assi2>
```

Count = 50000

```
26 [volta]/home/student/dshah/dhruv/assi2> gcc -o assi2 assi2.c
27 [volta]/home/student/dshah/dhruv/assi2> ./assi2

average: 0.639747
28 [volta]/home/student/dshah/dhruv/assi2>
```

Conclusion

My assumption was that the system would take almost identical average time to lock and unlock semaphore irrespective of number times the repetition of locking and unlocking would be done. My experiments agrees with my assumption. The results show that total time would increase with the increase in repetition but overall average time would be almost same. So the purpose of the project was met obviously. While working on this project I learned when to use semaphore and when not to use it. Also I learned how to create semaphore, lock and unlock the semaphore.

Main thing that I came to know is that if the application doesn't need synchronization and mutual exclusion then semaphores should not be used. Because it will take more time to generate output compared to the application without semaphore as some time will be taken for locking and unlocking the semaphore. So one should choose wisely if one wants to include semaphore or not for the particular application. To conclude, according to the application need one can tradeoff between memory and operating system time.

Appendix

```
/*
 * Needed Includes
 */
#include <errno.h>              /* errno and error codes */
#include <sys/time.h>           /* for gettimeofday() */
#include <unistd.h>             /* for gettimeofday(), getpid() */
#include <stdio.h>              /* for printf() */
#include <unistd.h>             /* for fork() */
#include <sys/types.h>          /* for wait() */
#include <sys/wait.h>           /* for wait() */
#include <signal.h>             /* for kill(), sigsuspend(), others */
#include <sys/ipc.h>            /* for all IPC function calls */
#include <sys/shm.h>            /* for shmget(), shmat(), shmctl() */
#include <sys/sem.h>            /* for semget(), semop(), semctl() */
#include <stdlib.h>

/*
 * Needed constants
 */
#define SEM_KEY       0x1234
/*
*  function to find start and end time in the program
*/
long long
timeval_diff(struct timeval *difference,
```

```
             struct timeval *end_time,
             struct timeval *start_time
            )
{
  struct timeval temp_diff;

  if(difference==NULL)
  {
    difference=&temp_diff;
  }

  difference->tv_sec =end_time->tv_sec -start_time->tv_sec ;
  difference->tv_usec=end_time->tv_usec-start_time->tv_usec;

  /* Using while instead of if below makes the code slightly more robust.
*/

  while(difference->tv_usec<0)
  {
    difference->tv_usec+=1000000;
     difference->tv_sec -=1;
  }

  return 1000000LL*difference->tv_sec+
                  difference->tv_usec;

}
```

```
int        g_sem_id;



/*
 * The following two structures hold the semapore information needed to
lock
 * and unlock the semaphores.  They are initialized at the beginning of
the
 * main program and used in the PrintAlphabet() routine.  By initializing
them
 * at the beginning, they can easily be used at a later point without
 * reassignment or changes.
 */
struct sembuf    g_lock_sembuf[1];
struct sembuf    g_unlock_sembuf[1];


/*
*Main function
*/


int main( int argc, char *argv[] )
{



        int count=1;
         double x,y,z,n;
        struct timeval e;
         struct timeval l;
```

```
/*
 * Build the semaphore structures
 */
g_lock_sembuf[0].sem_num   =  0;
g_lock_sembuf[0].sem_op    = -1;
g_lock_sembuf[0].sem_flg   =  0;

g_unlock_sembuf[0].sem_num =  0;
g_unlock_sembuf[0].sem_op  =  1;
g_unlock_sembuf[0].sem_flg =  0;




/*
 * Create the semaphore
 */
if( ( g_sem_id = semget( SEM_KEY, 1, IPC_CREAT | 0666 ) ) == -1 )
{
      fprintf(stderr,"semget()  call  failed,  could  not  create
      semaphore!");
      exit(1);
}
if( semop( g_sem_id, g_unlock_sembuf, 1 ) == -1 )
{
      fprintf(stderr,"semop()  call  failed,  could  not  initalize
      semaphore!");
```

```
        exit(1);
    }



    /*
    *     getting start time
    */
    if( gettimeofday( &e, NULL ) == -1 ) {
                fprintf(stderr,"Couldn't get time of day, exiting.\n");
                exit(1);
        }

    while(count <=5000)
    {


    /*
    *     locking the semaphore
    */
    if( semop( g_sem_id, g_lock_sembuf, 1 ) == -1 )
              {
                    fprintf(stderr,"semop()  call  failed,  could  not
                    lock semaphore!");
                    exit(1);
              }



        count=count+1;          //increamenting counter
```

```
/*
*     unlocking the semaphore
*/
if( semop( g_sem_id, g_unlock_sembuf, 1 ) == -1 )
      {
            fprintf(stderr,"semop()   call   failed,   could   not
            initialize semaphore!");
      exit(1);
      }



}


if( gettimeofday( &l, NULL ) == -1 )
      {
            fprintf(stderr,"Couldn't get time of day, exiting.\n");
            exit(1);
      }
      n=timeval_diff(NULL,&l,&e);
      printf("\naverage: %f\n",n/count);//calculating average time



           /*
            * Delete the semaphore
            */
           if( semctl( g_sem_id, 0, IPC_RMID, 0) != 0 )
           {
```

```
                    fprintf(stderr,"Couldn't remove the semahpore!\n");
                        exit(1);
                }

}
```