CS 503 Lab 2

Experimenting with Mutual Exclusion

Shuheng Guo

Content

1. Introduction

This lab is a follow up of lab 1. In lab 1 I used semaphores to enforce exclusive access to the shared resource. In this lab, there will be three different method used to realize mutual exclusive. They are as follows:

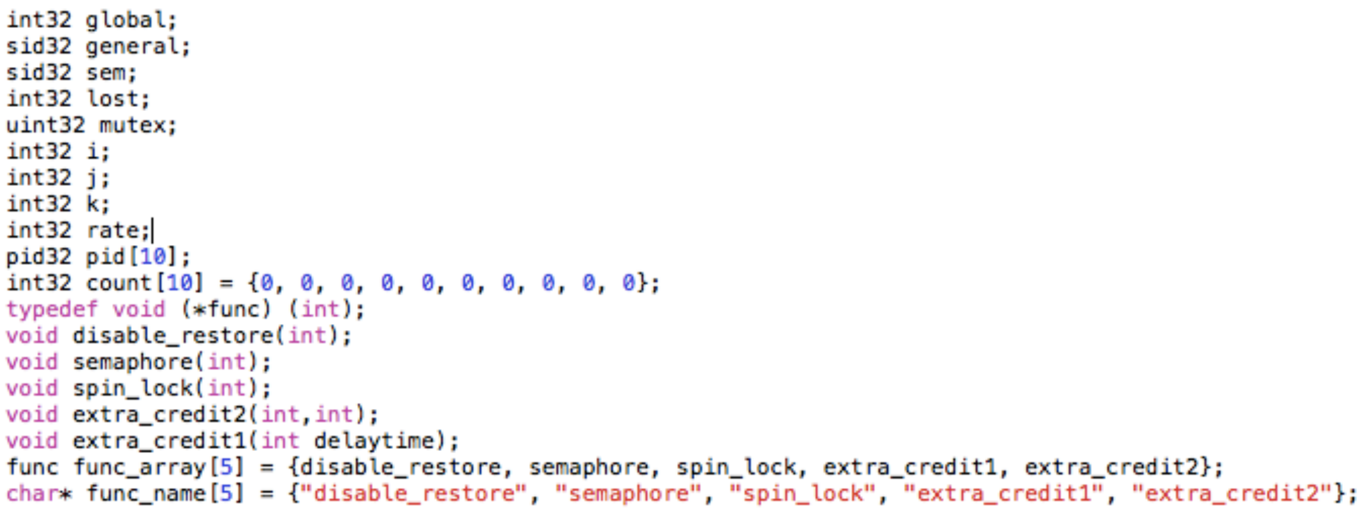
- Semaphores
- Spin lock, using the compare and exchange (CMPXCHG) instruction
- Disabling/Restoring interrupts.

Even though all of the three methods will do the job, but they may differ with each other in performance.

- Semaphores - Consider a semaphore with an initial value of 1, when a process waits on the semaphore the value is decremented and the process is allowed to continue executing if the value is >= 0. If the value is less than zero, the process is put to sleep until another process signals the semaphore.
- Disabling/Restoring interrupts - The XINU scheduler uses the clock interrupt to give each ready process a time slice of the CPU. When interrupts are disabled (using the disable() call), the clock interrupt is never sent so the XINU scheduler is never executed. As such, the process is allowed exclusive access to the CPU and any resources it wishes to use until interrupts are re-enabled.
- Spin lock using the CMPXCHG instruction - The compare and exchange instruction (CMPXCHG) is a special instruction in the x86 architecture that compares a register with a memory location and if the two are equal it sets the value of memory to a new value. This is done as an atomic operation.

2. Implementation and Explanation

2.1 variables



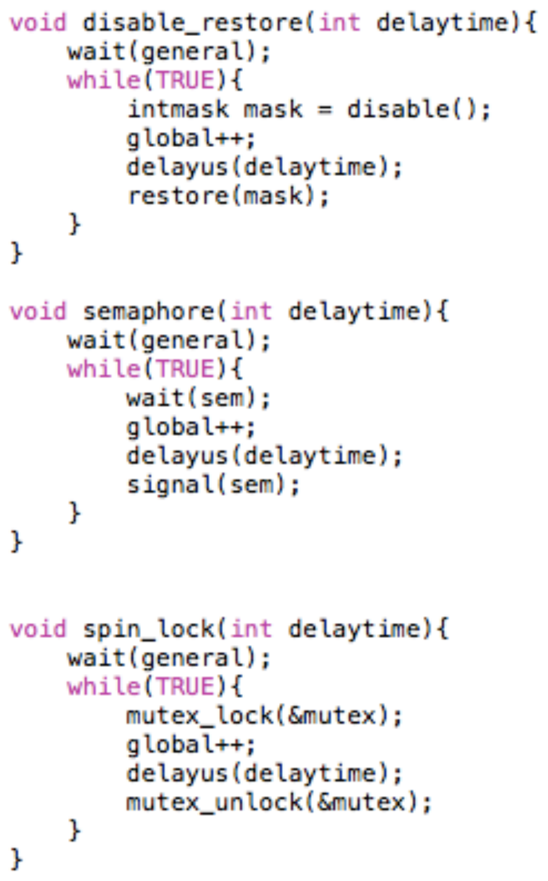
```
int32 global;
sid32 general;
sid32 sem;
int32 lost;
uint32 mutex;
int32 i;
int32 j;
int32 k;
int32 rate;
pid32 pid[10];
int32 count[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
typedef void (*func) (int);
void disable_restore(int);
void semaphore(int);
void spin_lock(int);
void extra_credit2(int,int);
void extra_credit1(int delaytime);
func func_array[5] = {disable_restore, semaphore, spin_lock, extra_credit1, extra_credit2};
char* func_name[5] = {"disable_restore", "semaphore", "spin_lock", "extra_credit1", "extra_credit2"};
```

Figure 1.   Variables in Lab2

Variables	Explanation
global	The variable to be added by three methods in every process
general	Semaphore used to start all
Sem	Semaphore for method use semaphore to realize mutual exclusive
Lost	time lost in problem extra credit 1
mutex	Mutex used for method spin lock
Rate	Iterate rate to measure performance of the three method
Count[]	Array of count by each process in problem extra credit 2
Func_array	Function array to restore different functions
Func_name	Array of function name
Pid[]	Array of process ID

Table 1. Variables in Lab2

2.2 Three Different Exclusive Functions



```
void disable_restore(int delaytime){
    wait(general);
    while(TRUE){
        intmask mask = disable();
        global++;
        delayus(delaytime);
        restore(mask);
    }
}

void semaphore(int delaytime){
    wait(general);
    while(TRUE){
        wait(sem);
        global++;
        delayus(delaytime);
        signal(sem);
    }
}


void spin_lock(int delaytime){
    wait(general);
    while(TRUE){
        mutex_lock(&mutex);
        global++;
        delayus(delaytime);
        mutex_unlock(&mutex);
    }
}
```

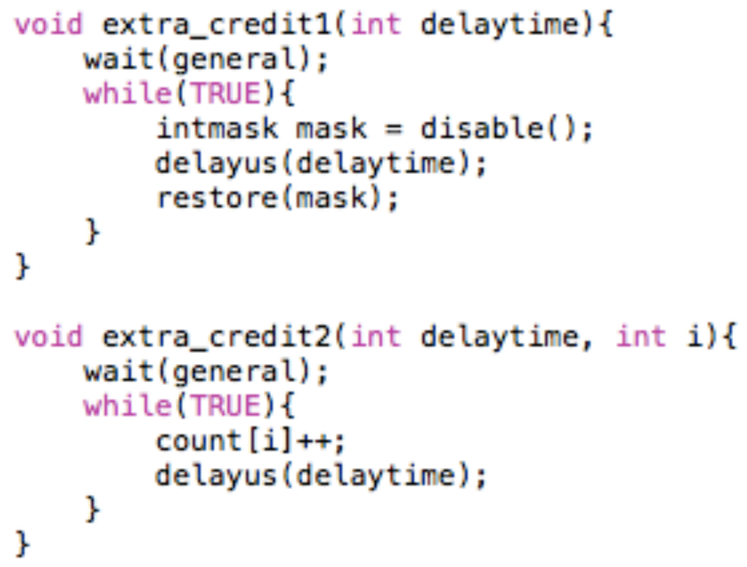
Figure 2. Three different Functions in Mutual Exclusive

The first function I used here is diable_restore. When interrupts are disabled (using the disable() call), the clock interrupt is never sent so the XINU scheduler is never executed. As such, the process is allowed exclusive access to the CPU and any resources it wishes to use until interrupts are re-enabled.

The second function I used here is semaphore. When a process waits on the semaphore the value is decremented and the process is allowed to continue executing if the value is >= 0. If the value is less than zero, the process is put to sleep until another process signals the semaphore.

The third function I used here is Spin lock. The compare and exchange instruction (CMPXCHG) is a special instruction in the x86 architecture that compares a register with a memory location and if the two are equal it sets the value of memory to a new value. This is done as an atomic operation.

2.3 Extra Credit

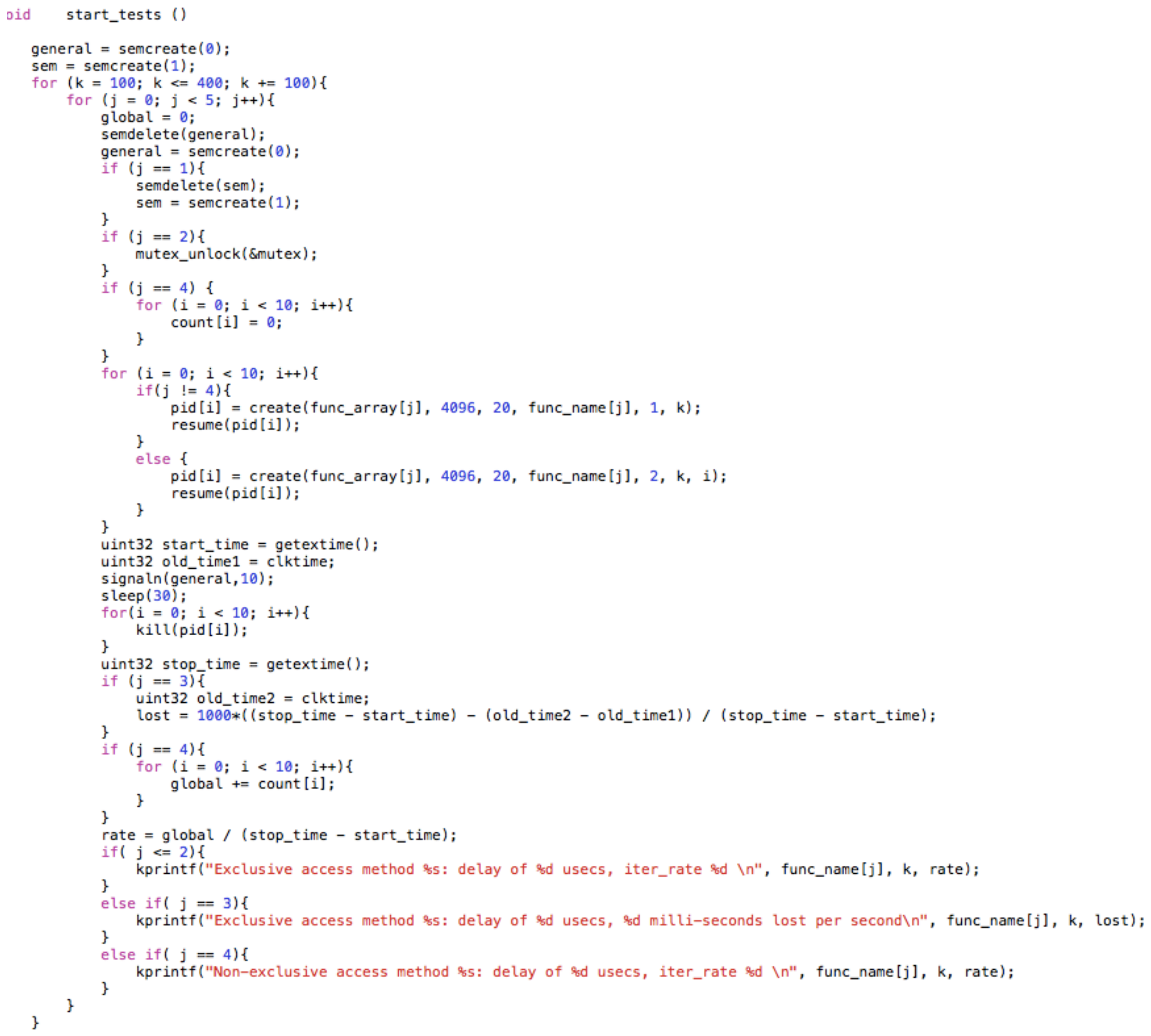


```
void extra_credit1(int delaytime){
    wait(general);
    while(TRUE){
        intmask mask = disable();
        delayus(delaytime);
        restore(mask);
    }
}

void extra_credit2(int delaytime, int i){
    wait(general);
    while(TRUE){
        count[i]++;
        delayus(delaytime);
    }
}
```

Figure 3. Extra Credit

The function extra_credit 1 is to cost the time lost in system clock. Every process just use disable-interrupt to contribute to the total time lost. The function extra_credit 2 is to calculate the overhead of every exclusive function and nonexclusive function. I just made every process to increase their own counter every time and sum them together to get the iterate rate of nonexclusive method. The result will be shown in chapter 3.

2.4 Start_tests



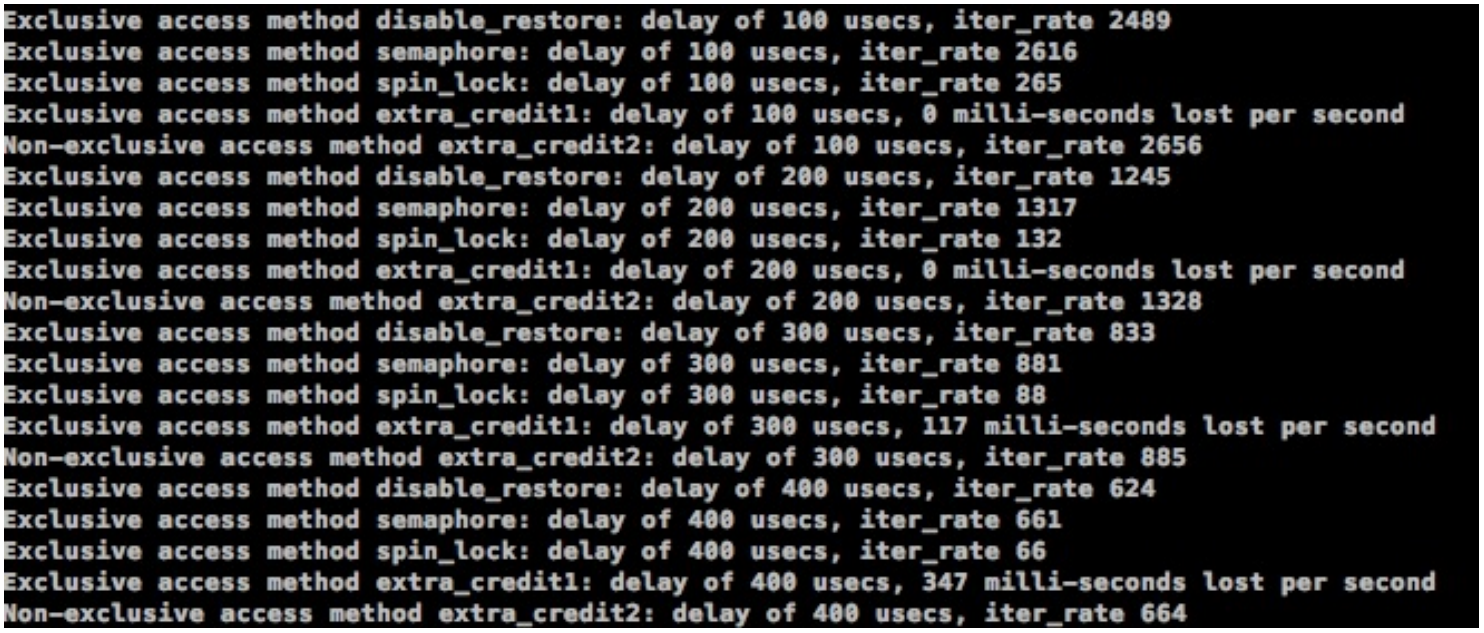
```
oid    start_tests ()

   general = semcreate(0);
   sem = semcreate(1);
   for (k = 100; k <= 400; k += 100){
       for (j = 0; j < 5; j++){
           global = 0;
           semdelete(general);
           general = semcreate(0);
           if (j == 1){
               semdelete(sem);
               sem = semcreate(1);
           }
           if (j == 2){
               mutex_unlock(&mutex);
           }
           if (j == 4) {
               for (i = 0; i < 10; i++){
                   count[i] = 0;
               }
           }
           for (i = 0; i < 10; i++){
               if(j != 4){
                   pid[i] = create(func_array[j], 4096, 20, func_name[j], 1, k);
                   resume(pid[i]);
               }
               else {
                   pid[i] = create(func_array[j], 4096, 20, func_name[j], 2, k, i);
                   resume(pid[i]);
               }
           }
           uint32 start_time = getextime();
           uint32 old_time1 = clktime;
           signaln(general,10);
           sleep(30);
           for(i = 0; i < 10; i++){
               kill(pid[i]);
           }
           uint32 stop_time = getextime();
           if (j == 3){
               uint32 old_time2 = clktime;
               lost = 1000*((stop_time - start_time) - (old_time2 - old_time1)) / (stop_time - start_time);
           }
           if (j == 4){
               for (i = 0; i < 10; i++){
                   global += count[i];
               }
           }
           rate = global / (stop_time - start_time);
           if( j <= 2){
               kprintf("Exclusive access method %s: delay of %d usecs, iter_rate %d \n", func_name[j], k, rate);
           }
           else if( j == 3){
               kprintf("Exclusive access method %s: delay of %d usecs, %d milli-seconds lost per second\n", func_name[j], k, lost);
           }
           else if( j == 4){
               kprintf("Non-exclusive access method %s: delay of %d usecs, iter_rate %d \n", func_name[j], k, rate);
           }
       }
   }
```

Figure 4. Function start_tests

This function starts the test. First, I initialize all the variables and semaphores. Using three for loops to create 10 processes for each function with different delay time. If conditioning sentences are used to deal with different situations of corresponding specific functions. Clktime and getextime() are used to get system time without/with interrupt disable time, respectively. The total time lost on interrupt disable can be calculated out by a simple expression. After 30 seconds, I killed all processes and calculate iterate rate of every function to be printed out.

3. Results

After running my lab2 code on XINU, I got the result as follows:



```
Exclusive access method disable_restore: delay of 100 usecs, iter_rate 2489
Exclusive access method semaphore: delay of 100 usecs, iter_rate 2616
Exclusive access method spin_lock: delay of 100 usecs, iter_rate 265
Exclusive access method extra_credit1: delay of 100 usecs, 0 milli-seconds lost per second
Non-exclusive access method extra_credit2: delay of 100 usecs, iter_rate 2656
Exclusive access method disable_restore: delay of 200 usecs, iter_rate 1245
Exclusive access method semaphore: delay of 200 usecs, iter_rate 1317
Exclusive access method spin_lock: delay of 200 usecs, iter_rate 132
Exclusive access method extra_credit1: delay of 200 usecs, 0 milli-seconds lost per second
Non-exclusive access method extra_credit2: delay of 200 usecs, iter_rate 1328
Exclusive access method disable_restore: delay of 300 usecs, iter_rate 833
Exclusive access method semaphore: delay of 300 usecs, iter_rate 881
Exclusive access method spin_lock: delay of 300 usecs, iter_rate 88
Exclusive access method extra_credit1: delay of 300 usecs, 117 milli-seconds lost per second
Non-exclusive access method extra_credit2: delay of 300 usecs, iter_rate 885
Exclusive access method disable_restore: delay of 400 usecs, iter_rate 624
Exclusive access method semaphore: delay of 400 usecs, iter_rate 661
Exclusive access method spin_lock: delay of 400 usecs, iter_rate 66
Exclusive access method extra_credit1: delay of 400 usecs, 347 milli-seconds lost per second
Non-exclusive access method extra_credit2: delay of 400 usecs, iter_rate 664
```

Figure 5. Lab 2 Result

From Figure 5 we can see that the result is reasonable. The iterate rate of method disable_restore and semaphore are almost the same but the method spin_lock is much lower than the first two method. As for extra_credit1 method, milli-seconds with delay time of 100us and 200us are too small to be shown in milli-seconds level. Lost time with delay time of 300us and 400us are 117ms and 347ms respectively. Iterate rate of nonexclusive method extra_credit2 should be larger than any exclusive method used before. And that is the case.

To calculate the overhead:

$$Overhead = 100 \times (1 - \left(\frac{exclusive}{nonexclusive}\right))$$

- Delay_time = 100us
  1) Disable_restore: 6.29%
  2) Semaphore: 1.51%

3) Spin_lock: 90.02%

- Delay_time = 200us

4) Disable_restore: 6.25%

5) Semaphore: 0.83%

6) Spin_lock: 90.06%

- Delay_time = 300us

7) Disable_restore: 5.88%

8) Semaphore: 0.45%

9) Spin_lock: 90.06%

- Delay_time = 400us

10) Disable_restore: 6.02%

11) Semaphore: 0.45%

12) Spin_lock: 90.06%

4. Conclusion and Discussion

After doing lab2, answer the following questions:

- Q: Which method for mutual exclusion provided the shortest wait time?
- Answer: From the lab result we can see that the performance of first two methods (disable_restore and semaphore) are almost the same. The method semaphore is a little bit better.
- Q: Can you think of other ways to measure the performance of mutual exclusion methods? Explain.
- Answer: To measure the performance of mutual exclusion we can get the variable global to a reasonable value and break the while loop whenever global reach to this preset value. Measure the total time spend in this process by function getextime(). The method with least time duration will have better performance.
- Q: Are there more methods for mutual exclusion that you can think of other than the three you tested? How would the performance compare to the three you tested?
- Answer: There are some other algorithms that can realize mutual exclusive:
    1. Dekker's algorithm: If one process is already in the critical section, the other process will busy wait for the first process to exit. This is done by the use of two flags, flag[0] and flag[1], which indicate an intention to enter the critical section and a turn variable that indicates who has priority between the two processes.
    2. Peterson's algorithm: Peterson's algorithm (AKA Peterson's solution) is a concurrent programming algorithm for mutual exclusion that allows two processes to share a single-use resource without conflict, using only shared memory for communication. It was formulated by Gary L. Peterson in 1981. While Peterson's original formulation worked with only two

processes, the algorithm can be generalized for more than two.

3. Lamport's bakery algorithm: Lamport's bakery algorithm is a computer algorithm devised by computer scientist Leslie Lamport, which is intended to improve the safety in the usage of shared resources among multiple threads by means of mutual exclusion.

Reference : http://en.wikipedia.org/wiki/Mutual_exclusion