CS 503 - Fall 2014

Lab 2
Experimenting with Mutual Exclusion

Due Sunday, September 21st, 2014 at 11:59 PM

Objectives:

By the end of this lab you will be able to:

- Understand how to perform mutual exclusion using different locking methods
- Determine the performance trade-offs of using different mutual exclusion methods

Background

In lab1 you created a simulation for producers (farmers) and consumers (vegetarians). To successfully control access to the shared resource (field), you had to use semaphores to enforce exclusive access to the shared resource. In this lab you will experiment different methods for performing mutual exclusion using:

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

Your job is to write several test cases (details given below) that use the different methods for mutual exclusion and record the performance of each method.

How each mutual exclusion method works

- 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. You used this to control exclusive access to the field in lab 1.

- 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. You have seen in class how a spin lock guarantees mutual exclusion. You will find the implementation of spin lock in the file system/mutex.S

Setup

In /homes/cs503/xinu there is a file called xinu-fall2014-lab2.tar.gz that contains a start to the code for this lab. Unpack:

```
tar zxvf /u/u3/cs503/xinu/xinu-fall2014-lab2.tar.gz
```

This will create a directory called xinu-fall2014-lab2.

Along with the main code for XINU, this tarball contains the following files (additional explanation of the contents of the files is in the following sections).

- system/mutex.S - function declarations for the mutex functions that implement a spin lock
- system/getextime.c - the declaration for the getextime function. More on this function later.
- system/main.c - the main XINU process
- system/lab2tests.c - Your Lab2 implementation

What's in system/mutex.S

The file mutex.S contains the function declarations (in x86 assembly) for 2 mutex functions:

- void mutex_lock(uint32* mutex) - this function uses the CMPXCHG assembly instruction to lock a mutex. It does not return until the mutex is aquired. The parameter is a pointer to the mutex lock.
- void mutex_unlock(uint32* mutex) - this function unlocks the mutex. The parameter is a pointer to the mutex lock.

NOTE: The mutex lock must be initialized before any process uses it. It can be initialized using mutex_unlock() since this function sets the value of the lock to zero.

What's in system/getextime.c

The file getextime has the definition for the function getextime():

- `uint32  getextime(void);`

This function returns an unsigned integer, which is the number of seconds since the system initialization.

But, Xinu already has a global variable that counts the number of seconds (clktime) since system initialization. What is the difference between the two?

The variable clktime is incremented by the clock interrupt handler. But, if interrupts are disabled (one of the mutual exclusion methods that you will use in this lab) clktime will not be updated, and it will not be an accurate measure of the number of seconds since system initialization.

getextime() reads the Real-time clock, which is a separate battery powered device on the Intel Galileo boards. Disabling interrupts does not affect the Real-time clock. Using getextime(), you can measure number of seconds between two events accurately, without worrying about interrupts.

What's in system/main.c

The file main.c contains the main process function, which will start the tests. There will be a single function called from main:

- start_tests();

The teaching assistants may replace main.c during grading, please DO NOT put any implementation in main.c

What's in system/lab2tests.c

This file contains the definition of a single function:

- void start_tests(void) - you have to write your Lab 2 implementation inside of this function.

How do I write a test case to test the mutual exclusion methods?

Similar to what you did in lab 1, you will need a shared resource. For this lab a global unsigned integer variable will work. You will create ten (10) processes for each method of mutual exclusion and different delay times, which will do the following:

```
repeat the following:
        gain exclusive access
        increment the global variable
        do some non-blocking computations (eg. busy wait)**
        release exclusive access
```

** The delayus(m) function will do a busy wait for m micro-seconds. You can look at the implementation of delayus() in include/delay.h

As mentioned above, there are three methods to achieve mutual exclusion. You will write test cases for all the three methods. The start_tests() function should do the following:

```
global variable = 0
create 10 processes
sleep for 30 seconds
kill the 10 processes
print report (details below)
```

This will be done for all the three methods of mutual exclusion, and for different arguments of delayus() (details below)

Just like you did in Lab1, you must ensure that all the processes start executing at the same time. (Hint: signaln)

How do I measure performance of mutual exclusion methods?

In each iteration of the critical section the value of the variable is incremented by 1. Hence, the total amount of computation by a process in the critical section is directly proportional to the value of the global variable.

But, the process also spends some time waiting to gain access to the global variable. This time is considered as an overhead (because no useful computation is being done during this waiting period). By the end of the testing time (30 seconds), if you compare the value of the global variable for all the methods of exclusive access, you will be able to compare performance of each method of mutual exclusion.

You can use the function getextime() just before starting the processes and just after killing the processes to get the number of seconds between these two events.

You must also vary the amount of computation each process does in each interation, by varying the argument passed to the delayus() function.
The following delay values must be used: 100 micro-seconds, 200 micro-seconds, 300 micro-seconds and 400 micro-seconds.

You must compute the rate of iterations for each case. It is defined as:

```
iter_rate = (value of global variable) / time elapsed
```

After each run, the following must be printed:

```
Exclusive access method xxxx: delay of xxxx usecs, iter_rate: xxxx
```

With delay values mentioned above you would get an output that looks like this:

```
Exclusive access method disable/restore: delay of 100 usecs, iter_rate xxxx
Exclusive access method semaphore: delay of 100 usecs, iter_rate xxxx
Exclusive access method spin lock: delay of 100 usecs, iter_rate xxxx

Exclusive access method disable/restore: delay of 200 usecs, iter_rate xxxx
Exclusive access method semaphore: delay of 200 usecs, iter_rate xxxx
Exclusive access method spin lock: delay of 200 usecs, iter_rate xxxx

Exclusive access method disable/restore: delay of 300 usecs, iter_rate xxxx
Exclusive access method semaphore: delay of 300 usecs, iter_rate xxxx
Exclusive access method spin lock: delay of 300 usecs, iter_rate xxxx

Exclusive access method disable/restore: delay of 400 usecs, iter_rate xxxx
Exclusive access method semaphore: delay of 400 usecs, iter_rate xxxx
Exclusive access method spin lock: delay of 400 usecs, iter_rate xxxx
```

The higher the iter_rate, more time was spent on useful computations and less time was spent waiting for the lock.

Requirements and Tips:

In the 'system' directory include a PDF with a write up discussing your results, the details behind your implementation, and the answers to the following questions.

- Which method for mutual exclusion provided the shortest wait time?
- Can you think of other ways to measure the performance of mutual exclusion methods? Explain.
- 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?

Extra Credit

1. A clock interrupt in Xinu is generated every 1 ms. A clock interrupt is also called as a clock tick. The clock interrupt handler increments an internal milli-second counter and a second counter. When you disable interrupts to get mutual exclusion, the interrupt handler is never called and so we lose clock ticks.

   For extra credit, you have to write a code that counts the number of milli-seconds lost per minute. For this you will use the following test case:
   10 processes, each process runs:

   ```
   repeat the following:
           disable interrupts
           delayus(300) /* busy wait for 300 us */
           restore interrupts
   ```

   In this case, you have to count the number of milli-seconds lost per second [Hint: use getextime() intelligently]

   In the report, you must write:
   - How many milli-seconds are lost per second?
   - Details about your implementation - how you managed to count number of lost milli-seconds per second.
   - Why do you think your implementation gives the correct answer?
   - Try different values of delay and try to get a relation between delay time and lost milli-seconds per second.

2. In this lab implementation, we calculated the amount of useful work done by the processes and compared how fast each method of mutual exclusion is. But, to get the actual overhead, we must compare the useful computation rate with the maximum possible useful computation rate for the same period of time.
   For example, if the iter_rate with no mutual exclusion is 100 and with mutual exclusion is 90, then the estimate of overhead is 100 * (1 - (90/100)) = 10% i.e. 10% of the total time is spent trying to gain exclusive access.

You have to write a code to measure the iter_rate when there is no mutual exclusion. NOTE: This does not mean, incrementing the global variable without mutual exclusion. You have to find how many iterations would be executed if mutual exclusion was not needed at all. [Hint: have each process increment it's own counter so they do not need mutual exclusion]
Calculate the overheads for each of the three mutual exclusion methods you have used in this lab.

What to turn in

Submit using turnin command your complete source code (all of XINU) including the any files you added to complete the lab. Don't forget to include your write up in the 'system' directory.

To turn in your lab use the following command

```
turnin -c cs503 -p lab2 xinu-fall2014-lab2
```

assuming xinu-fall2014-lab2 is the name of the directory containing your code.

If you wish to, you can verify your submission by typing the following command:

```
turnin -v -c cs503 -p lab2
```

Do not forget the -v above, as otherwise your earlier submission will be erased (it is overwritten by a blank submission).

Note that resubmitting overwrites any earlier submission and erases any record of the date/time of any such earlier submission.

We will check that the submission timestamp is before the due date; we will not accept your submission if its timestamp is after the due date. Do NOT submit after 11:59 PM Eastern Time.