Student: Yu Cheng (Jade)

ICS 412

Homework #3

October 07, 2009

Homework #3

Exercise 1: Explain why a spin-lock is not a good idea on a single processor, single machine.

Answer: There is a bad side and a good side of spin-locks. The process holding the spin-lock does nothing but waiting when it is allocated by the CPU. So it burns the CPU cycles and slows down other processes. On the other hand, in the absence of tread priorities, spin-lock usually provides a faster responding time since it waits and constantly checks on the lock that this process is waiting for.

A single processor machine, however, does not benefit from the mechanism in the good side. This is due to the context swathing. In other words, the machine is pretending to be multithreaded. Whatever lock the spin-lock process is waiting for would not become available while the spin-lock process is allocated to run by the CPU because it would be the only process running in that time slice. The states for all other processes remain the same. The lock becomes available after the execution of the corresponding thread, and the spin-lock thread learns about this information the next time when it gets to run. This is same versus Blocked/Ready mechanism as far as CPU involvement goes. Therefore, a single processor machine does not gain a faster responding time by using the spin-lock and does not save the context switching overhead. The only thing it does is busy waiting, which is a waste of CPU cycles.

Exercise 2: Condition variable implementations typically provide a `signal_all()` function to wake up all threads blocked on the condition variable. (In Pthreads it's `pthread_cond_broadcast()`.) Consider the following code fragments for the Bounded Buffer Producer Consumer problem, for an arbitrary number of producers and consumers, written in OO Pseudo-Code. `mutex` and `cond` are a lock and a condition variable, respectively, both defined elsewhere in the code and visible by producers and consumers.

Producer:

```
...
mutex.lock();
while(buffer.isFull())
      cond.wait(mutex);
buffer.insert();
cond.signal_all();
mutex.unlock();
...
```

Consumer:

```
...
mutex.lock()
while(buffer.isEmpty());
      cond.wait(mutex);
buffer.remove();
cond.signal_all();
mutex.unlock();
...
```

Explain why if the calls to `signal_all()` were replaced by calls to `signal()` the code would not behave properly. What would be the problem? Give a concrete sequence of events for which this code would misbehave. Assume that condition variables are FIFO: upon a call to `signal()` the thread that had placed the first call to `wait()` is awakened.

Answer: Let's Assume function call `signal()` is used in the code, and we have a buffer of size one, two Consumer processes, and two Producer processes. We can simultaneously have multiple processes waiting in the blocked state, say two Producer processes. Now a Consumer process comes in and executes the signal call, which wakes up the first blocked Producer process. But before the wakened Producer process gets to run, the second Consumer is assigned to run by the CPU. This Consumer process checks for the `buffer.isEmpty` condition, since the previous Consumer just took away the only element in the buffer, this Consumer would go in to wait and block. Then the wakened Producer process executes and signal the second Producer process to wake up. The second Producer process would enter block state immediately since the buffer is still full. At this stage one Consumer process and one Producer are done, the other Consumer and Producer processes are both waiting. We stuck.

Let's consider the same situation with `signal_all()` function calls. Since once one process adds or deletes successfully, it wakes up all processes that are in the blocked wait state, the processes would stay in blocked state for a relatively shorter time, while enter and leave the wait state at a relatively faster rate. While in the awaked state, they continuously try to obtain the lock and further try to add or delete based on the buffer condition until they eventually accomplished the task.

The two scenarios can be represented using the following UML interaction diagrams:

```
Function call signal() is used:
-------------------------------------------------------------------------------------------------
Buffer              P-1                  P-2                  C-1                  C-2

Full                 :                    :                    :                    :
                    [ ] TRYLOCK           :                    :                    :
                     |                   [ ] TRYLOCK           :                    :
                     |                    |                   [ ] TRYLOCK           :
                     |                    |                    |                   [ ] TRYLOCK
                    [ ] LOCKED            |                    |                    |
                    [ ] WAIT              |                    |                    |
                    [ ] UNLOCK            |                    |                    |
                     |                   [ ] LOCKED            |                    |
                     |                   [ ] WAIT              |                    |
                     |                   [ ] UNLOCK            |                    |
                     |                    |                   [ ] LOCKED            |
Empty                |                    |                   [ ] DEL               |
                     |                    |                   [ ] SIGNAL            |
                     |                    |                   [ ] UNLOCK            |
                     |                    |                    :                   [ ] LOCKED
                     |                    |                    :                   [ ] WAIT
                     |                    |                    :                   [ ] UNLOCK
                    [ ] WAKEUP            |                    :                    |
                    [ ] TRYLOCK           |                    :                    |
Full                [ ] ADD               |                    :                    |
                    [ ] SIGNAL            |                    :                    |
                    [ ] UNLOCK            |                    :                    |
                     :                   [ ] WAKEUP            :                    |
                     :                   [ ] TRYLOCK           :                    |
                     :                   [ ] WAIT              :                    |
                     :                   [ ] UNLOCK            :                    |
                     :                    |                    :                    |
-------------------------------------------------------------------------------------------------

Function call signal_all() is used:
-------------------------------------------------------------------------------------------------
Buffer              P-1                  P-2                  C-1                  C-2

Full                 :                    :                    :                    :
                    [ ] TRYLOCK           :                    :                    :
                     |                   [ ] TRYLOCK           :                    :
                     |                    |                   [ ] TRYLOCK           :
                     |                    |                    |                   [ ] TRYLOCK
                    [ ] LOCKED            |                    |                    |
                    [ ] WAIT              |                    |                    |
                    [ ] UNLOCK            |                    |                    |
                     |                   [ ] LOCKED            |                    |
                     |                   [ ] WAIT              |                    |
                     |                   [ ] UNLOCK            |                    |
                     |                    |                   [ ] LOCKED            |
Empty                |                    |                   [ ] DEL               |
                     |                    |                   [ ] SIGNAL_ALL        |
                     |                    |                   [ ] UNLOCK            |
                     |                    |                    :                   [ ] LOCKED
                     |                    |                    :                   [ ] WAIT
                     |                    |                    :                   [ ] UNLOCK
                    [ ] WAKEUP            |                    :                    |
                    [ ] TRYLOCK           |                    :                    |
Full                [ ] ADD               |                    :                    |
                    [ ] SIGNAL_ALL        |                    :                    |
                    [ ] UNLOCK            |                    :                    |
                     :                   [ ] WAKEUP            :                    |
                     :                   [ ] TRYLOCK           :                    |
                     :                   [ ] WAIT              :                    |
                     :                   [ ] UNLOCK            :                    |
                     :                    |                    :                   [ ] WAKEUP
                     :                    |                    :                   [ ] TRYLOCK
                     :                    |                    :                   [ ] LOCKED
Empty                :                    |                    :                   [ ] DEL
                     :                    |                    :                   [ ] SIGNAL_ALL
                     :                    |                    :                   [ ] UNLOCK
                     :                   [ ] WAKEUP            :                    :
                     :                   [ ] TRYLOCK           :                    :
                     :                   [ ] LOCKED            :                    :
Full                 :                   [ ] ADD               :                    :
                     :                   [ ] SIGNAL_ALL        :                    :
                     :                   [ ] UNLOCK            :                    :
                     :                    :                    :                    :
-------------------------------------------------------------------------------------------------
```

Exercise 3: We consider a program in which we have two kinds of threads. *Adder* threads ad 1 to a shared counter. *Subtracter* threads subtract N from the shared counter (where N is an integer > 0). There can be an arbitrary numbers of *Adders* and *Subtracters* arrivals (possibly a finite number of them). The counter is set to zero initially. Here are the code's requirements:

- The counter should never be negative.
- The counter should never stay at a value $\geq N$ as long as there is a subtracter in the system.
- *Subtracters* need to block until they can safely do the subtraction (i.e., without making the counter negative).

Consider the following implementation based on Semaphores:

Three Semaphores with initial values:

```
mutex_counter: 1
mutex_sub: 1
notZero: 0
```

Adder:

```
     . . .
L1   P(mutex_counter);
L2   counter++;
L3   V(mutex_counter);
L4   V(notZero);
     . . .
```

Subtracter:

```
     . . .
L5   P(mutex_sub);
L6   for (i = 0; i < N; i++)
L7          P(notZero);
L8   V(mutex_sub);
L9   P(mutex_counter);
L10  counter -= N;
L11  V(mutex_counter);
     . . .
```

a. Would there be a problem if semaphore `mutex_counter` where used instead of semaphore `mutex_sub` in lines L5 and L8? If so, which one? If not, why?

Answer: Yes, there will be a dead-lock problem. Originally, when *subtracter* enters the `mutex_sub` protected section, it waits for the `notZero` counter to go up to N, and then puts `notZero` back to zero and exits the `mutex_sub` protected section. Later on it subtracts N. Since at this point the counter variable is at least N, this ensures the counter does not become negative.

After the modification, changing from `mutex_sub` to `mutex_counter`, the *subtracter* obtains the `mutex_counter` lock, and waits for `notZero` to go up to N before releasing the `mutex_counter` lock. This courses problems. If the *subtracter* holds the `mutex_coutner` lock, the *adder* wouldn't get to run, the `notZero` would never go up to N, which is what the *subtracter* is waiting for. This is, therefore, a dead lock.

The scenario can be represented using the following UML interaction diagrams:

```
L5, L8 are changed to mutex_counter:
-------------------------------------------------------------------------------------------------
         Subtracter                                          Adder

          :                                                   :
          :                                                   :
         [ ] TRY  mutex_counter = 1                           :
         [ ] SET  mutex_counter = 0                           :
         [ ] TRY  notZero < N                                 :
         [ ] WAIT notZero                                     :
          |                                                  [ ] TRY  mutex_counter = 0
          |                                                  [ ] WAIT mutex_counter
          |                                                   |
-------------------------------------------------------------------------------------------------
```

b. Would there be a problem if lines L8 and L9 were switched? If so, which one? If not, why?

Answer: No, I don't think there will be a serious problem. Originally, when *subtracter* enters the `mutex_sub` protected section, it waits for the `notZero` counter to go up to *N*, and then puts `notZero` back to zero, exits the `mutex_sub` protected section, and then obtain the `mutex_counter` semaphore. After the modification, even through L9 is within the `mutex_sub` protected block; the *subtracter* would still be waiting for `notZero` to go up to *N* before acquiring the `mutex_counter`. This allows the *adder* to continue working until `noZero` reaches *N*.

The scenario can be represented using the following UML interaction diagrams:

```
L8, L9 are changed to mutex_counter:
-------------------------------------------------------------------------------------------------
         Subtracter                                          Adder

          :                                                   :
          :                                                   :
         [ ] TRY  mutex_sub = 1                               :
         [ ] SET  mutex_sub = 0                               :
         [ ] TRY  notZero < N                                 :
         [ ] WAIT notZero                                     :
          |                                                  [ ] TRY  mutex_counter = 1
          |                                                  [ ] SET  mutex_counter = 0
          |                                                  [ ] counter ++
          |                                                  [ ] SET  mutex_counter = 1
          |                                                  [ ] SET  notZero ++
         [ ] TRY  notZero < N                                 :
          |                                                  [ ] TRY  mutex_counter = 1
          |                                                  [ ] SET  mutex_counter = 0
          |                                                  [ ] counter ++
          |                                                  [ ] SET  mutex_counter = 1
          |                                                  [ ] SET  notZero ++
         [ ] TRY  notZero < N                                 :
          |                                                  [ ] TRY  mutex_counter = 1
          |                                                  [ ] SET  mutex_counter = 0
          |                                                  [ ] counter ++
          |                                                  [ ] SET  mutex_counter = 1
          |                                                  [ ] SET  notZero ++
         [ ] TRY  notZero >= N                                :
         [ ] TRY  mutex_counter = 1                           :
         [ ] SET  mutex_counter = 0                           :
         [ ] SET  mutex_sub = 1                               :
         [ ] counter -= N                                     :
         [ ] SET  mutex_counter = 1                           :
          :                                                   :
-------------------------------------------------------------------------------------------------
```

c. Would there be a problem if lines L3 and L4 were switched? If so, which one? If not, why?

Answer: No, I don't think there will be any serious problem, other than the *subtracter* might need to wait a little extra time to obtain the `mutex_counter` semaphore after exiting the `mutex_sub` protected section and ready to subtract *N* from the counter.

The scenario can be represented using the following UML interaction diagrams:

```
L3, L4 are changed to mutex_counter:
-------------------------------------------------------------------------------------------------
            Subtracter                                        Adder

             :                                                 :
             :                                                 :
            [ ] TRY  mutex_sub = 1                             :
            [ ] SET  mutex_sub = 0                             :
            [ ] TRY  notZero < N                               :
            [ ] WAIT notZero                                   :
             |                                                [ ] TRY  mutex_counter = 1
             |                                                [ ] SET  mutex_counter = 0
             |                                                [ ] counter ++
             |                                                [ ] SET  notZero ++
             |                                                [ ] SET  mutex_counter = 1
            [ ] TRY  notZero < N                               :
             |                                                [ ] TRY  mutex_counter = 1
             |                                                [ ] SET  mutex_counter = 0
             |                                                [ ] counter ++
             |                                                [ ] SET  notZero ++
             |                                                [ ] SET  mutex_counter = 1
            [ ] TRY  notZero < N                               :
             |                                                [ ] TRY  mutex_counter = 1
             |                                                [ ] SET  mutex_counter = 0
             |                                                [ ] counter ++
             |                                                [ ] SET  notZero ++
             |                                                [ ] SET  mutex_counter = 1
            [ ] TRY  notZero >= N                              :
            [ ] SET  mutex_sub = 1                             :
            [ ] TRY  mutex_counter = 1                         :
            [ ] SET  mutex_counter = 0                         :
            [ ] counter -= N                                   :
            [ ] SET  mutex_counter = 1                         :
             :                                                 :
-------------------------------------------------------------------------------------------------
```

d. Using the same style of pseudo-code as in Exercise #2, write an equivalent implementation using one lock and one condition variable, and no semaphores.

Answer: In the following code, there is a lock, `mutex`, and a conditional variable, `cond`.

Adder:

```
      :
L1    mutex.lock();
L2    count++;
L3    cond.signal_all();
L4    mutex.unlock();
      :
```

Subtracter:

```
      :
L5    mutex.lock();
L6    while(count < N)
            cond.wait(mutex);
L7
L8    counter -= N;
L9    cond.signal_all();
L10   mutex.unlock();
      :
```