Homework for Week 10
(Due Date: Check Canvas)

1. The following is an implementation of a stack data structure for a multithreaded application. Identify the bugs in this implementation and correct them.

```
(01) #include "Exception.h"
(02) #include "my_lock.h"
(03) #include <iostream.h>

(04) const MaxStackSize = 100;

(05) class Stack { // throws an exception when popping an empty or pushing into full stack
(06) private:
(07)   int s[MaxStackSize];
(08)   int stackp;     // stack pointer
(09)   Exception * e;  // for error handling
(10)   Lock lock;      // for mutual exclusion
(11) public:
(12)   Stack();
(13)   ~Stack() {}
(14)   int Pop();
(15)   void Push(int item);
(16) };
(17)
(18) Stack::Stack() {
(19)   stackp = MaxStackSize;
(20)   e = new Exception();
(21) }
(22)
(23) int Stack::Pop() {
(24)   lock.lock();
(25)   if (stackp == MaxStackSize) {
(26)     e->SetErrorMsg("Popping empty stack");
(27)     throw(e);
(28)   }
(29)   lock.unlock();
(30)   return s[stackp++];
(31) }
(32)
(33) void Stack::Push(int item) {
(34)   lock.lock();
(35)   if (stackp == 0) {
(36)     e->SetErrorMsg("Pushing to a full stack");
(37)     throw(e);
(38)   }
(39)   s[--stackp] = item;
(40)   lock.unlock();
(41) }
```

2. Consider Peterson's solution to the mutual exclusion problem described in Lesson "Software Solutions to the Critical Section Problem". Suppose we make a small change to the solution and change the line

   ```
   turn = other;
   ```

   to

   ```
   turn = self;
   ```

   This change will cause the program to no longer be correct. Describe what can go wrong if this change is made. That is, describe a sequence of events that leads to a situation where either:

   - Both processes can be in their critical section at the same time, or
   - A process wants to enter its critical section, but it cannot and loops infinitely waiting to enter.

3. John Hacker is a hardware designer who came up with a great idea for a hardware instruction that he claims can help in solving the critical section problem. It is called atomic counters. An atomic counter is a variable in memory that can be incremented and then sampled in one atomic operation. Also, it can be reset to 0. That is, the two operations allowed on the shared variable a are:

   ```
   j = a++; // execute in one atomic operation; j is value after increment
   a = 0;
   ```

   Either come up with a solution to the C.S. problem using this facility or show that it cannot help.

4. In Lesson "Concurrent Data Structures" we learned how the function `contains()` is implemented for class `LazyList`. Provide the code for function `contains()` for the class `FineList`, which was also discussed in the lesson.

5. [6] Your new employee claims that the lazy list's validation method (described in Lesson "Concurrent Data Structures") can be simplified by dropping the check that `pred.next` is equal to `curr`. After all, the code always sets `pred` to the old value of `curr`, and before `pred.next` can be changed, the new value of `curr` must be marked, causing the validation to fail. Explain the error in this reasoning.

References

[1] A. Silberschatz, P. Galvin, and G. Gagne, *Applied Operating Systems Concepts*, John Wiley & Sons, Inc., New York, NY, 2000.

[2] Deitel, Deitel, and Choffnes, *Operating Systems*, Pearson / Prentice Hall, 2004.

[3] A. S. Tanenbaum, *Modern Operating Systems*, Pearson / Prentice Hall, 2008.

[4] L. F. Bic, A. C. Shaw, *Operating Systems Principles*, Prentice Hall 2003.

[5] C. Crowley, *Operating Systems, A Design-Oriented Approach*, Irwin 1997.

[6] M. Herlihy, N. Shavit, *The Art of Multiprocessor Programming*, Elsevier, 2008