Parallel Algorithms (15-499), Spring 09
Assignment #6 **Due: Not Due**

These problems will not be due but are meant as practice for the final.

Problem 1: Interleavings

Assuming a sequentially consistent memory give all possible outcomes of the following codes (i.e. for each list possible tuples of the final values of `x`, `y` and `z`).

```
x = 3;
in Parallel {
  y = x;   x = 22; y=11;
  x = 4;  y = 11; x = y;
}
```

```
x = 3;
in Parallel {
  y = x;  atomic{ x = 22; y= 11;}
  atomic{ x = 4;  y = 11;}  x = y;
}
```

Problem 2: Linearizable Queue

Here is an implementation of a FIFO queue.

```
struct queue {
  int back;    // position to put new elements
  item *itemArray;  // pointer to the array of elements
}

void enq(queue *q, item x) {
  int i = FetchAdd(&q->back,1);
  q->itemArray[i] = x; }

item dequeue(queue *q) {
  while (1) {
     int r = q->back;
     for (int i = 1; i < r; i++) {
        item x = Swap(&q->itemArray[i], EMPTY);
        if (x != EMPTY) return(x);  } } }
```

In the code we assume the array is initialized so all entries contain a special value `EMPTY`. We claim this implementation of queues is linearizable. Please answer the following questions. For each please explain in at least a few sentences.

1. Is it lock free?
2. Is it wait free?
3. Is it lockout free?
4. Since it is linearizable every operation must look like it occurred atomically somewhere between the start and end. It is tempting to argue that the linearization point of the `enq` is at the `FetchAdd`. Show that this is not correct (i.e. two enqueues might be serialized in a different order than the order of the `FetchAdd`s.).
5. Also show that the assignment `q.itemArray[i] = x` is not a linearization point. This indicates that the `enq` does not have a single linearization point and the point will depend on the context.
6. *Prove it is linearizable even though there is no fixed linearization point.

Problem 3: Reader Writer Locks

Below is an implementation of reader-writer locks. Recall that a reader-writer lock lets multiple reads into a region but only a single writer. In the code given up to n readers are allowed into the region.

```
int l = n;

void readLock() {
  if (l > 0) {
     if (FetchAdd(&l,-1) > 0) then return;
     else FetchAdd(&l,1); }}

void readUnlock() { FetchAdd(&l,1); }

void writeLock() {
  if (l > 0) {
     if (FetchAdd(&l,-n) > 0) then return;
     else FetchAdd(&l,n); }}

void writeUnlock() { FetchAdd(&l,n); }
```

The idea of the code is that as long as $l > 0$ we allow readers in. Also if $l = n$ then we have no readers so we allow a writer in, which then blocks readers since $l = 0$. Answer the following questions about the code.

1. If we just have readers is it fair (lockout free) for the readers assuming we have more than n of them? What if we have at most n readers?
2. If we have both readers and writers is it lockout free? (Assume at most n readers). If we have a lot of readers is it going to be a problem for the writers?
3. *Write a version of reader-writer locks based on `FetchAdd` which is lockout free for both readers and writers. Hint: consider the bakery algorithm based on `FetchAdd` and perhaps also look at the code for Room Synchronizations in the Asynchronous Algorithms notes.

Problem 4: Transactional Memory

Here is a scheme for implementing optimistic (delayed write) transactional memory. In the scheme, as with the scheme described in class, there is a computation part that writes all values to a local buffer and a commit part that tries to commit the changes to memory. If a transaction abort it is tried again. We assume every object has a write-lock associated with it and assume that if you read when the write lock is on it returns a special INVALID value (or raises an exception).

Computation part: During the computation part we keep track of every read location along with the version number that is read. If a read returns the INVALID value then abort the transaction. All writes are stored locally.

Commit part: During the commit phase we acquire a lock for all the values that need to be written. We then check that all locations we read still have the same version in them. If any versions have changed we abort the transaction and release the write locks. If the versions are the same we write all local values to the shared memory (incrementing the version number) and release the write locks.
Here are some questions about the scheme:

1. Can you get into a live-lock situation in which two transactions keep aborting each other and neither makes progress?
2. Is this scheme correct? If so prove it guarantees the transactions are serializable. It is probably useful to identify the serialization point.
3. Can this scheme lead to an infinite loop in some code that would normally always terminate if run serially?
4. For a write when should we grab the old version number—i.e. the one we will increment by 1 for the write? Should we do it when we first write locally or when we commit?