CS 61A Summer 2009 Homework 6

Topic: Parallel Programming

Lectures: Monday 8/3, Tuesday 8/4, Wednesday 8/5, Thursday 8/6

Reading: Abelson and Sussman: 3.4, Lecture Notes on Mapreduce

1. (Graded by Correctness) Write a non-deterministic program `(factor num)` that takes a positive integer and successively returns all the factors of `num` not including 1 and the `num` itself. Each factor need only appear once although it's ok if it appears multiple times.

For example:

```
> (factor 18)
 2
> choose-another
 3
> choose-another
 6
> choose-another
 9
> choose-another
 There are no more choices
```

Be sure to test your program with ~`cs61a/lib/ambeval-final.scm` (which uses `choose` and `choose-another`) or ~`cs61a/lib/vambeval.scm` (which uses `amb` and `try-again`).

Note: in order to test your code with `parallel-execute` you must load ~`cs61a/lib/concurrency.scm` Be careful however that a couple of successful test case does not mean your code is correct. Some errors will only occur occasionally depending on how the atomic steps are interleaved.

2. Abelson and Sussman, 3.39, 3.40

3. Get the file ~`cs61a/lib/transfer-loop.scm` Consider the `transfer` procedure:

```
(define (transfer x)
    (if (<= x saving-acct)
  (begin (set! saving-acct (- saving-acct x))
         (set! checking-acct (+ checking-acct x)))
  "Don't have enough"))
```

We protected `transfer` by composing the serializers `(T (S transfer))`. However, this is somewhat inefficient. `transfer` really has two stages where it first deducts from the saving account and then adds to the checking account. While `transfer` is deducting from the savings account, other people should be able to access the checking account.
Fix this inefficiency by directly modifying the code in `transfer-loop.scm` You must also make sure that your changes will not create any new concurrency bugs.

4. Continuing with the file `transfer-loop.scm` from above. We would like to build an additional procedure `save-up`, which will take all the money in the `checking-acct` and move it to the `saving-acct`. The following is an attempted implementation.

```
(define (save-up)
   (let ((temp checking-acct))
     (serial-withdraw temp)   ;; serial-withdraw is (S check-withdraw)
     (serial-deposit temp)))  ;; serial-deposit is (T save-deposit)
```

Louis Reasoner claims that the above `save-up` procedure is already safe from concurrency errors since it's calling `serial-withdraw` and `serial-deposit`. Why is he wrong? Describe some potential concurrency errors that can still occur. How would you fix them?

5. Mutation procedures such as `set-car!` and `set-cdr!` are also vulnerable to concurrency errors. For example:

```
(define a-pair (cons 1 2))
(parallel-execute (lambda () (set-cdr! a-pair (+ 1 (cdr a-pair))))
                  (lambda () (set-cdr! a-pair (* 2 (cdr a-pair)))))
```

The above code can have errors in the same way that `set!` examples have errors.

In this problem, we are going to create a new concurrency control mechanism called a `lock`:

1. `lock` takes a pair as input and returns a protected version of the same pair
2. The procedure `make-lock` should return a new lock.
3. Define new procedures `safe-set-car!` and `safe-set-cdr!` that takes as a input protected pair it should only modify the pair if no other processes are calling `safe-set-car!` or `safe-set-cdr!` on a pair protected by the same lock

So, with `lock`, we can now write code like:

```
(define L (make-lock))
(define protected-p (L (cons 1 2)))

(parallel-execute (lambda () (safe-set-cdr! protected-p (+ 1 (cdr protected-p))))
                  (lambda () (safe-set-cdr! protected-p (* 2 (cdr protected-p)))))
```

And it should now be free of concurrency errors.

Use Mutex or Serializers to implement `make-lock`, `safe-set-car!`, and `safe-set-cdr!`

6. Louis Reasoner cannot understand why the Operating System and the hardware must provide `mutex`. Louis Reasoner believes that he can build his own mutex using just plain Scheme as such:

```
(define (make-mutex)
  (let ((taken #f))
    (define (dispatch m)
       (cond ((eq? m 'acquire)
                  (if (not taken) (set! taken #t)
                          (dispatch 'acquire))) ;; if not mutex already taken, try again
             ((eq? m 'release)
                 (set! taken #f))))
    dispatch))
```

Is Louis Reasoner correct? What is wrong with his "solution" ?

Note: The following question is optional relate to `mapreduce`. In order to test your code, you must first run `ssh cs61a-XX@icluster1.eecs.berkeley.edu` where you replace `XX` with your log-in.

7. (Optional) Do the `mapreduce` question from Lab 7B.