MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1999

Recitation – Wednesday, April 7

1. Concurrency

Give all possible values of x that can result from executing the following:

```
(define x 10)
```

$$\texttt{(parallel-execute (lambda () (}\overbrace{\text{set!}}^{C}\texttt{ x (* }\overbrace{\texttt{x}}^{A}\ \overbrace{\texttt{x}}^{B}\texttt{)))}$$
$$\texttt{(lambda () (}\underbrace{\text{set!}}_{G}\texttt{ x (* }\underbrace{\texttt{x}}_{D}\ \underbrace{\texttt{x}}_{E}\ \underbrace{\texttt{x}}_{F}\texttt{))))}$$

`; Possible Values:` 10^2, 10^3, 10^4, 10^5, 10^6

Which of these possibilities remain if we instead use serialized procedures:

```
(define x 10)
(define s (make-serializer))
```

$$\texttt{(parallel-execute (s (lambda () (}\overbrace{\text{set!}}^{C}\texttt{ x (* }\overbrace{\texttt{x}}^{A}\ \overbrace{\texttt{x}}^{B}\texttt{)))}$$
$$\texttt{(s (lambda () (}\underbrace{\text{set!}}_{G}\texttt{ x (* }\underbrace{\texttt{x}}_{D}\ \underbrace{\texttt{x}}_{E}\ \underbrace{\texttt{x}}_{F}\texttt{))))}$$

`; Possible Values:` 10^6

2. More Concurrency: Bank Accounts

In class yesterday, we saw how we can use serializers to make account balances more safe for deposits and withdrawals. Consider the following procedure for making an account. We can define the procedure `get-balance` and a simple `deposit` procedure.

```
(define (make-account balance)
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((serializer (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'deposit) deposit)
            ((eq? m 'balance) balance)
            ((eq? m 'serializer) serializer)))
    dispatch))
```

```
(define (get-balance acct)
  (acct 'balance))

(define (deposit acct amount)
  (let ((d (acct 'deposit)))
    (d amount)))

(set a (make-account 100))
```

Notice that this `deposit` procedure is not safe because we do not use the serializer. For example,
`(parallel-execute (lambda () (deposit a 20)) (lambda () (deposit a 30)))`
could fail. How can we redefine the deposit procedure to make it safe if multiple deposits happen concurrently?

```
(define (deposit acct amount)
  (let ((s (acct 'serializer))
        (d (acct 'deposit)))
    ((s d) amount))
)
```

Do we need to redefine `get-balance` in the same way? Why or why not?
⇒No, because balance is always a reasonable value.

3. Exchanging Accounts

Now, consider the process of exchanging the amount of money in two accounts. For example,

```
(define a (make-account 100))
(define b (make-account 50))
(exchange a b)
(get-balance a)  ==>  50
(get-balance b)  ==> 100
```

Below is an `exchange` procedure that does not serialize. Using `exchange`, we can now write a serialized exchange that applies the serializers from both accounts before making the exchange:

```
(define (exchange a1 a2)
  (let ((diff (- (a1 'balance)
                 (a2 'balance))))
    ((a1 'deposit) (- diff))
    ((a2 'deposit) diff)))
```

```
(define (serialized-exchange a1 a2)
  (let ((s1 (a1 'serializer))
        (s2 (a2 'serializer)))
    ((s1 (s2 exchange)) a1 a2)
  ))
```

Does this fix everything? Not quite. This could make things worse! Now we have the concept of **deadlock**. One way of fixing deadlock is to give each serializer a unique id and insist every process acquire serializers in order of the unique ids.

Imagine that we've changed make-serializer so that it makes a serializer with a unique id, as show below, left. How could we change serialized-exchange to prevent deadlock?

```
(define s (make-serializer))
(define t (make-serializer))
(s proc) ==>  <serialized proc>
(s 'id)  ==>  1
(t 'id)  ==>  2
```

```
(define (serialized-exchange a1 a2)
  (let ((s1 (a1 'serializer))
        (s2 (a2 'serializer)))
    ((if (< (s1 'id) (s2 'id))
         (s1 (s2 exchange))
         (s2 (s1 exchange)))
     a1 a2)
  ))
```

4. Unique Ids

Let's try to add unique ids to our serializers. First, let's write a function called `tag-proc` that takes a procedure and returns a procedure that is tagged with a **unique** id. See the examples on the right.

```
(define tag-proc
  (let ((count 0))
    (lambda (f)
       (set! count (+ count 1))
       (let ((id count))
         (lambda (x)
           (if (eq? x 'id)
               id
               (f x))))))
)
```

```
(define f (tag-proc sqrt))
(define g (tag-proc sqrt))
(define h (tag-proc cube))
(f 'id) ⇒ 1     (unique id is 1)
(g 'id) ⇒ 2     (unique id is 2)
(h 'id) ⇒ 3     (unique id is 3)
(f 25)  ⇒ 5
(g 81)  ⇒ 9
(h 3)   ⇒ 27
```

How can we change our `make-serializer` procedure so that each serializer that is generated has a unique id? (Where do we put the call to `tag-proc`?)

```
(define (make-serializer)

 (let ((mutex (make-mutex)))
   (tag-proc
    (lambda (p)
      (define (serialized-p . args)
       ...
      )

      serialized-p))))
```