Structure and Interpretation of Computer Programs
Second Edition
Sample Problem Set

Simulation and Concurrency

The programming assignment for this week explores two ideas: the simulation of a world in which objects are characterized by collections of state variables and the characteristics of systems involving concurrency. These ideas are presented in the context of a market game. In order not to waste too much of your own time, it is important to study the system and plan your work before coming to the lab.

This problem set begins by describing the overall structure of the simulation. The exercises will help you to master the ideas involved.

How England lost her Barings
or
an abject Leeson for the banking community

Earlier this year the English banking community was stunned by the failure of the 233-year-old Baring Brothers investment banking company. A 28-year-old trader in Singapore, Nick Leeson, lost over $1G in a rather short time. Mr. Leeson, who was originally involved in arbitrage trading on the differences between the prices of the Nikkei-225 average in the Osaka and Singapore stock exchanges incurred massive losses from an extremely leveraged position in the Nikkei average.

Markets

In our simplification, we model a market, such as the Osaka market in Nikkei-225 derivatives or the Singapore market in Nikkei-225 derivatives, as a message acceptor with internal state variables, `price` and `pending-orders`. The message acceptor can handle requests to get the current price, to update the price, to accept an order, and to process pending orders. We make these markets with a market-constructor procedure as follows:

```
;;; The initial price
(define nikkei-fundamental 16680.0)

(define Osaka
  (make-market "Osaka:Nikkei-225" nikkei-fundamental))

(define singapore
  (make-market "Singapore:Nikkei-225" nikkei-fundamental))
```

There are several messages accepted by a market. For example, one may obtain the current price on the Singapore market as follows:

```
(singapore 'get-price)
;;; Value: 16673.23
```

Traders interact with markets. A trader is modeled by an object that holds two kinds of assets: a monetary balance and a number of contracts. In this simulation there is only one kind of contract: A Nikkei-225 derivative contract (whatever that is! However, we may watch Mr. Leeson lose money with or without knowing what these contracts are about.)

The message `get-price` will be used by traders to obtain quotes of the current market price of a contract. The trader bases his decisions on this price.

Every so often, a trader may place a new order by sending a `new-order!` message to a market. An order is a procedure (of no arguments) supplied by the trader to be executed by the market in the near future. An order may modify the assets of the trader.

Our simulation system for the interaction of traders and markets sends certain system messages to the objects to model the flow of time. Thus, every so often a market receives, from the system, a message to change the price or to process orders. The markets are implemented as follows:

```
(define (make-market name initial-price)
  (let ((price initial-price)
        (price-serializer (make-serializer))
        (pending-orders '())
        (orders-serializer (make-serializer)))
    (define (the-market m)
      (cond ((eq? m 'new-price!)
             (price-serializer
              (lambda (update)
                (set! price (update price)))))
            ((eq? m 'get-price) price)
            ((eq? m 'new-order!)
             (orders-serializer
              (lambda (new-order)
                (set! pending-orders
                      (append pending-orders (list new-order))))))
            ((eq? m 'execute-an-order)
             (((orders-serializer
                (lambda ()
                  (if (not (null? pending-orders))
                      (let ((outstanding-order (car pending-orders)))
                        (set! pending-orders (cdr pending-orders))
                        outstanding-order)
                      (lambda () 'nothing-to-do)))))))
            ((eq? m 'get-name) name)
            (else (error "Wrong message" m))))
    the-market))
```

We can instantiate the general market to make a particular market, say the Osaka market in Nikkei-225 derivatives:

```
(define nikkei-fundamental 16680.0)

(define Osaka
  (make-market "Osaka:Nikkei-225" nikkei-fundamental))
```

Notice that in `make-market` the `price-serializer` is applied to a procedure that takes a procedure `update` and uses it to compute the new price from the old one. The *critical region* guarded by the serializer contains both the assignment to the protected variable and a read access of that variable.

The acceptance of a new order is similarly serialized. Execution of an order is also serialized with the `orders-serializer`, but this is a much more complicated situation.

Exercise 1: Why do we need two different serializers to implement a market? What bad result could we expect if we made only one serializer and used it to serialize all three of the guarded regions? Why must the `orders-serializer` be used for guarding two regions?

The message `new-price!` will be issued to each market every so often by a process that models random market forces. There are many factors that influence the price of a market commodity, and we cannot begin to model those factors. However, we can imagine that the commodity price will take a random walk starting with its fundamental value, and drifting. The procedure `nikkei-update` implements just such a strategy, whose details are probably not important, except in the way that it updates the prices when called (see the listing).

The `execute-an-order` message is used by the system to cause the market to execute one of the pending orders. If there are orders pending then the first is selected and executed, and removed from the list of orders.

Exercise 2: The code for handling an `execute-an-order` message is quite complicated. Louis Reasoner suggests that it could be simplified as follows:

```
((eq? m 'execute-an-order)
 ((orders-serializer
   (lambda ()
     (if (not (null? pending-orders))
         (begin ((car pending-orders))
                (set! pending-orders (cdr pending-orders))))))))
```

Unfortunately, Mr. Reasoner's suggestion is (as usual) not completely correct; it will work in the simple cases we have included in the problem set, but not in general. A slightly better idea, which is still not quite correct is:

```
((eq? m 'execute-an-order)
 (let ((current-order (lambda () 'nothing-to-do)))
   (if (not (null? pending-orders))
       ((orders-serializer
         (lambda ()
           (begin (set! current-order (car pending-orders))
                  (set! pending-orders (cdr pending-orders)))))))
   (current-order)))
```

Explain in no more than three short, clear sentences each, what is wrong with Louis's idea, and why the second try is better but still not quite right. Watch out – this question is subtle – the answer is not obvious. Try to draw timing diagrams showing how these methods may fail.

The code supplied defines a particular kind of trader, an arbitrager, Nick Leeson, who tries to make money on the difference of the value of a commodity on different exchanges. He buys on the low exchange and sells on the high one, pocketing the difference. The idea works, if the orders can be processed faster than the exchange moves. The arbitrager is implemented as follows:

```
(define (make-arbitrager name balance contracts authorization)
  (let ((trader-serializer (make-serializer)))
    (define (change-assets delta-money delta-contracts)
      ((trader-serializer
        (lambda ()
          (set! balance (+ balance delta-money))
          (set! contracts (+ contracts delta-contracts))))))
    (define (a<b low-place low-price high-place high-price)
      (if (> (- high-price low-price) transaction-cost)
          (let ((amount-to-gamble (min authorization balance)))
            (let ((ncontracts
                   (round (/ amount-to-gamble (- high-price low-price)))))
              (buy ncontracts low-place change-assets)
              (sell ncontracts high-place change-assets)))))
    (define (consider-a-trade)
      (let ((nikkei-225-Osaka (Osaka 'get-price))
            (nikkei-225-singapore (singapore 'get-price)))
        (if (< nikkei-225-Osaka nikkei-225-singapore)
            (a<b Osaka nikkei-225-Osaka
                 singapore nikkei-225-singapore)
            (a<b singapore nikkei-225-singapore
                 Osaka nikkei-225-Osaka))))
    (define (me message)
      (cond ((eq? message 'name) name)
            ((eq? message 'balance) balance)
            ((eq? message 'contracts) contracts)
            ((eq? message 'consider-a-trade) (consider-a-trade))
            (else
             (error "Unknown message -- ARBITRAGER" message))))
    me))
```

We can instantiate a particular trader as an instance of the arbtrager:

```
(define nick-leeson
  (make-arbitrager "Nick Leeson" 1000000000. 0.0 10000.))
```

So Nick is represented as a message acceptor that answers to a few messages. One may ask for his name, his monetary balance, his stock of contracts, and one may poke him to consider a trade. The simulation system will do this aperiodically, as part of the model of the flow of time.

Traders buy and sell contracts at a market using the procedure `transact`. A trader gives the market permission to subtract from the trader's monetary balance the cost of the contracts purchased and to add to the trader's stash the contracts he purchased. A sell order is just a buy of a negated number of contracts. The `permission` argument is just a procedure supplied by the trader that takes an amount of money and a number of contracts. (In the `arbitrager` trader above it is just the procedure `change-assets`.) `Permission` performs the action of modifying the trader's assets when the trade is executed.

```
(define (buy ncontracts market permission)
  (transact ncontracts market permission))

(define (sell ncontracts market permission)
  (transact (- ncontracts) market permission))
```

```
(define (transact ncontracts market permission)
  ((market 'new-order!)
   (lambda ()
     (permission (- (* ncontracts (market 'get-price)))
                 ncontracts))))
```

Exercise 3: What shared variables are protected by the `trader-serializer` in the `make-arbitrager` definition? Describe, in a few concise sentences, an example of a problem that is prevented by this serializer.

In this simulated market world there are a few other autonomous agents. There is a ticker for each market, which periodically prints the current price of a contract on that market, and there is an auditor, which aperiodically prints the assets of a trader. These minor players are just the following procedures:

```
(define (ticker market)
  (newline)
  (display (market 'get-name))
  (display "            ")
  (display (market 'get-price)))

(define (audit trader)
  (newline)
  (display (trader 'name))
  (display "    Balance: ")
  (display (trader 'balance))
  (display "    Contracts: ")
  (display (trader 'contracts)))
```

Finally, there is the system that we use to run our simulation. It is implemented by the procedure `start-world`, which you may find in the listing attached. `Start-world` uses a procedure `parallel-execute`, which starts up any number of independent processes. We will not try to tell you how `parallel-execute` works — it is not pretty. The system also provides a procedure `sleep-current-thread` that returns to its caller after waiting a number of milliseconds indicated by its argument. You will have to understand how to use this procedure, but you need not try to figure out how it works either.

Exercise 4: If you run the system long enough, you will see that the `audit` procedure occasionally prints out anomolous results. Sometimes, the balance/contracts for Nick are way out of line, but they get corrected soon after that:

```
Nick Leeson Balance: 1000098661.9071354 Contracts: 0.
Nick Leeson Balance: 1000135769.2921371 Contracts: 0.
Singapore:Nikkei-225 16723.26212008945
Tokyo:Nikkei-225 16719.39333592154
Singapore:Nikkei-225 16731.04299412824
Tokyo:Nikkei-225 16730.877484891316
Nick Leeson Balance: 1000179972.9031762 Contracts: -5207.
Nick Leeson Balance: 1145160434.4739444 Contracts: -8662.
Nick Leeson Balance: 1000235215.9529605 Contracts: 0.
Nick Leeson Balance: 1000584254.2787036 Contracts: 0.
...
Tokyo:Nikkei-225 16802.379854988434
Singapore:Nikkei-225 16805.91475593874
Tokyo:Nikkei-225 16803.939209184016
Singapore:Nikkei-225 16807.359095567597
Nick Leeson Balance: 922712662.5753176 Contracts: 0.
Tokyo:Nikkei-225 16805.338357625566
```

For tutorial, be prepared to explain, in a few simple sentences, what is causing this problem and to describe what must be done to fix the problem. Do *not* try to implement your changes.

Exercise 5: Do exercise 3.42 on page 289 of the notes. Is there any reason why our market simulation might have problems with deadlock? If not, why not? If so, explain how it might happen.

To do in the lab

When you load the problem set the market system will be ready to start. You may start it by executing `(start-world)`. It will produce a periodic time-sequence of market prices and aperiodic audits of Mr. Leeson, the arbitrager. You may stop the system by executing `(stop-world)`. To restart the system execute `(start-world)` again. If you are into macho programming you can patch the code while it is running (without stopping the system). This is commonly done by "real programmers" in debugging live operating systems.

Lab Exercise 1: Run the world for a bit. Does Mr. Leeson seem to lose money, gain money, or break even on the average? The time constants for the way the world runs are the numbers (of milliseconds) occuring in the procedure `start-world`. Also, Mr. Leeson's strategy depends on the value of the constant `transaction-cost`. Which of these constants could you change to make arbitrage extremely profitable? How do you think transaction-cost interacts with the timing constants? To support your argument, make a change and demonstrate the improved profits.

Lab Exercise 2: Make another arbitrager, say Bob Citron, who recently lost about \$1.7G of money invested by the taxpayers of Orange County, CA. Install him and demonstrate a system where both Mr. Leeson and Mr. Citron are executing trades on the Osaka and Singapore markets.

Lab Exercise 3: You will probably see a case where the parallel interleaving of the process threads fouls up the I/O, mixing the characters that are output by a ticker and the auditor. Can you explain this? Can you figure our a way to fix it? Write the code required to fix this bug. Be prepared to argue to your tutor that your code fixes exactly this bug and has no other consequences (such as preventing two markets from processing orders simultaneously.) (Hint: You can use a serializer, carefully.)

Lab Exercise 4: Your job is to to invent another kind of trader, such as one who tries to predict the future value of the commodity from analysis of its past behavior. You may try whatever strategy you think is effective, but you may not cheat by changing the code of any other component of the system or by diddling with the parameters of the nikki-update, or by setting the price of a market. Prizes will be awarded for the most interesting trader invented. Implement your trader, install the trader as a process, and demonstrate that the program works. Show output demonstrating that interesting trades are being made.