SICP 3.4

October 26, 2007 at 11:05 Tags SICP

I'm going to use PLT Scheme for this section, because CLISP doesn't have thread support. PLT Scheme supports "green" threads[1] and synchronization elements.

This is `parallel-execute`:

```
(define (parallel-execute . thunks)
  (for-each thread thunks))
```

The `thread` function invokes its argument in a separate thread and returns immediately[2].

And this is `make-serializer`:

```
(define (make-serializer)
  (let ((mutex (make-semaphore 1)))
    (lambda (p)
      (define (serialized-p . args)
        (semaphore-wait mutex)
        (let ((val (apply p args)))
          (semaphore-post mutex)
          val))
      serialized-p)))
```

It uses PLT's *semaphore* object to do its work.

Exercise 3.38

a. To list the possible values, I'll map the possible orders of execution:

```
[peter, paul, mary] -> 45
[peter, mary, paul] -> 35
[mary, peter, paul] -> 40
[mary, paul, peter] -> 40
[paul, peter, mary] -> 45
[paul, mary, peter] -> 50
```

It is interesting to notice here that order between commutative operations (Peter and Paul) doesn't matter. The relative order of Mary does matter, however.

b. Consider one option: Peter's code runs first, fetches the `balance` of 100 and adds 10 to it, but doesn't get the chance to store it back before a task switch. Then comes a task switch and Paul's code runs fully, setting the balance to 80. Peter's code then comes back and stores the 110 it has

computed to `balance` (thus completely hiding Paul's operation). Mary's code then sets `balance` to 55.

Exercise 3.39

Note that this is not a full serialization of the computations. The second computation can still interfere between the computation and assignment of the first. Therefore, the possible results are:

- 101: P1 completes, then P2 completes
- 100: P1 computes `(* x x)`, then P2 completes (and sets `x` to 101), then P1 executes the assignment.
- 121: P2 completes, then P1 completes

Exercise 3.40

Now we have a full serialization, and the only thing that can differ between two executions is the order in the execution of P1 and P2. However, since the operations P1 and P2 are commutative, the same result will be produced in both cases: 1,000,000

Exercise 3.41

Since both `withdraw` and `deposit` make a single modification to the balance, I can't see how accessing it can result in anomalous behavior. Depending on the order of execution of the access relatively to `withdraw`, one can either see the old or the new value – but this is allowed, since the value is consistent with reality.

Perhaps, had `withdraw` did two assignments to the balance, for whatever reason, we could hit an intermediate state with an access.

Exercise 3.42

It is a safe change to make, and I can't see any kind of concurrency allowed by the original solution but not this one.

The reason for this is that the real work of the serializer is done in the call to protected procedures, and not in their creation. In their creation the function `serialized-p` is created and returned, and only when it's called it waits on the mutex.

Exercise 3.43

First, let's examine the serial case. The exchange operation given two accounts with balances A and B leaves the balances B and A, changing the order[3] but not the sums. The same principle applies to exchanging any number of accounts.

This will be violated in the first version of `exchange` defined in the book. Suppose that the accounts are: a1(10), a2(20), a3(30). Peter exchanges a1 and a2 while Paul concurrently exchanges a1 and a3. Let's examine the following scenario:

Peter's `exchange` computes the difference: -10. Now, Paul's `exchange` is switched to, and completes its whole work by leaving the accounts as: a1(30), a2(20), a3(10). Peter's `exchange` resumes its work, withdraws -10 from a1, and adds -10 to a2. The final state is: a1(40), a2(10), a3(10), which is completely inconsistent.

However, `exchange` always preserves the sum of the balances of its input accounts, by taking the same sum from one account and adding it to another.

Had the individual accesses not been serialized, we'd get back to the problems examined in the

beginning of the chapter. A call to `withdraw` could become intermixed with a call to `deposit` for the same account, leaving its balance completely incorrect.

Exercise 3.44

I'll take Ben's side here. Since the withdrawal and deposit are serialized, I can't see how the `transfer` operation can leave any of the account in an inconsistent state. Even if the transfer is interrupted between the withdraw and deposit, it still holds the correct sum to deposit into the target account and will do it eventually. In any given moment, an account holds its balance plus amounts "owed to it" by all pending `transfer` operations. Therefore, if all transfers complete, eventually the balances in all accounts will be correct.

There is an essential difference between the transfer problem and the exchange problem, and it is the lack of computation of `difference`, which may examine some intermediate state of the balance which no longer reflects reality if `exchange` is switched out and in between the computation of the difference and the account operations.

Exercise 3.45

When `serialized-exchange` is called, the serializers of both accounts are activated. Then, when `exchange` tries to call `deposit` or `withdraw`, it can't because these functions also try to use the serializer. It will block on the call of `withdraw` from account1 and stay in this state indefinitely.

Exercise 3.46

Suppose that p1 executes the test `(if (car cell)` on an untaken mutex. The test suceeds, but p1 is switched out. p2 executes the same test which also suceeds (since p1 still hasn't reached the acquiring code). In this way, both p1 and p2 will eventually acquire the mutex.

Exercise 3.47

Note that the implementation of `make-serializer` in PLT Scheme I posted above uses semaphores to implement mutexes. This is trivial, since a mutex is just a special case of a semaphore. In this exercise, we'll see how to implement semaphores in terms of mutexes, which is a little more complicated.

a. Here's the implementation using a mutex:

```
(define (make-semaphore-mtx maximal)
  (let ((count maximal)
        (mutex (make-mutex)))
    (define (the-sema m)
      (cond ((eq? m 'release)
             (mutex 'acquire)
             (unless (= count maximal)
               (set! count (+ 1 count)))
             (mutex 'release))
            ((eq? m 'acquire)
             (mutex 'acquire)
             (cond
               ((> count 0)
                 (set! count (- count 1))
                 (mutex 'release))
               (else
                 (mutex 'release)
                 (the-sema 'acquire))))
            (else
              (error "Unknown request -- " m))))
    the-sema))
```

This semaphore is a fancy counter that blocks when asked to acquire when empty. It uses a mutex to protect all accesses to the counter so that concurrent calls will leave it in a consistent state.

b. Using `test-and-set!` is almost identical, except that we'll have to implement the wait on a locked cell manually:

```
(define (loop-test-and-set! cell)
  (if (test-and-set! cell)
    (loop-test-and-set! cell)
    '()))

(define (make-semaphore-ts maximal)
  (let ((count maximal)
        (guard (cons #f '())))
    (define (the-sema m)
      (cond ((eq? m 'release)
             (loop-test-and-set! guard)
               (unless (= count maximal)
                 (set! count (+ 1 count)))
             (clear! guard))
            ((eq? m 'acquire)
              (cond
                (loop-test-and-set! guard)
                ((> count 0)
                  (set! count (- count 1))
                  (clear! guard))
                (else
                  (clear! guard)
                  (the-sema 'acquire))))
            (else
              (error "Unknown request -- " m))))
    the-sema))
```

Exercise 3.48

In the current implementation, given the exchange between a1 and a2, in parallel with the exchange between a2 and a1, we may have a situation where one process holds a lock on a1 while another holds a lock on a2. If we number the accounts, when both processes will first attempt to lock a1. Since a2 can be locked only when a1 is locked, we'll have no deadlock.

Here's the implementation:

```
(define (make-account number balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((balance-serializer (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            ((eq? m 'number) number)
            ((eq? m 'balance) balance)
            ((eq? m 'serializer) balance-serializer)
            (else (error "Unknown request -- MAKE-ACCOUNT"
                         m))))
    dispatch))
```

```
(define (serialized-exchange account1 account2)
  (let ((serializer1 (account1 'serializer))
        (serializer2 (account2 'serializer)))
    (if (< (account1 'number) (account2 'number))
      ((serializer2 (serializer1 exchange))
        account1 account2)
      ((serializer1 (serializer2 exchange))
        account1 account2))))
```

[1] Also called “interpreter threads”. These are threads implemented in the interpreter, unrelated to the OS native threads. This means that such threads can’t really be utilized to increase performance on parallel machines (since they run in a single process), and also that if one thread waits on a system call it blocks all the others. On the other hand, on uniprocessor computers, green threads have been found to be faster than native threads for some applications, because they're very light weight and context switching is very fast. Additionally, being implemented on the level of the interpreter, green threads behave exactly the same way on all platforms. *Thanks to Jens Axel Soegaard for bringing this trade off to my attention.*

[2] It actually returns the thread *descriptor* by which we can later access the thread, but we don’t use this feature here.

[3] Order is defined if we sort the accounts in some way unrelated to the balance in them. Say, lexicographically by the account’s object name.

Back to