REVIEW, CONCURRENCY 19

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

July 20, 2010

Review

For those of you who are looking for extra one on one help, go talk to Sesh! His 5:00-6:30 section had only handful of people. If you think about it, you're getting something on the order of 5-6 times more one on one time in his section than other ones.

MetaCircular Evaluator

First, implement `set!`, it should be remarkably similar to `define-variable!`.

We want to be able to dynamically evaluate problems. In other words, we want to be able to say something like `(set! (car ls) 3)` if the car is a symbol.

1. What changes would you need to make to the mc-eval code to accomplish this change?
2. How would you implement this instead as a separate primitive procedure called `eval-and-set!`

Environments / Local State

Let's look at something like this:

```
(define x 3)
(define (foo)
  (let ((bar (lambda () x)))
       (define x 5)
       (bar)))
```

What does this print? Why?

1 Overview

Many things we take for granted in ordinary programming become problematic when there is any kind of parallelism involved. These situations include

- multiple processors (hardware) sharing data
- software multithreading (simulated parallelism)
- operating system input/output device handlers

This is the most important topic in CS 162, the operating systems course; here in 61A we give only a brief introduction, in the hope that when you see this topic for the second time it'll be clearer as a result.

To see in simple terms what the problem is, think about the Scheme expression: `(set! x (+ x 1))`

As you'll learn in more detail in 61C, Scheme translates this into a sequence of instructions to your computer. The details depend on the particular computer model, but it'll be something like this:

```
Load x into local copy
Add 1 to local copy
Store local copy into x
```

Ordinarily we would expect this sequence of instructions to have the desired effect. If the value of `x` was 100 before these instructions, it should be 101 after them.

But imagine that this sequence of three instructions can be interrupted by other events that come in the middle. To be specific, let's suppose that someone else is also trying to add 1 to `x`'s value. Now we might have this sequence:

My Process	Other Process
`Load x into Local Copy`	
`Add 1 to Local Copy`	
	`Load x into Local Copy`
	`Add 1 to Local Copy`
	`Store Local Copy into X`
`Store Local Copy into X`	

The problem that we need to attack is the *critical section*, which means a sequence of instructions that mustn't be interrupted. The three instructions starting with the load and ending with the store are a critical section.

Actually, we don't have to say that these instructions can't be interrupted; the only condition we must enforce is that they can't be interrupted by another process that uses the variable `x`. It's okay if another process wants to add 1 to `y` meanwhile. So we'd like to be able to say something like:

```
reserve x
load x into local copy
add 1 to local copy
store local copy into x
release x
```

2 Levels of Abstraction

Computers don't really have instructions quite like `reserve` and `release`, but we'll see that they do provide similar mechanisms. A typical programming environment includes concurrency control mechanisms at three levels of abstraction:

SICP Name	Protects	Provided by
serializer	high level abstraction	programming language
mutex	critical section	operating system
test-and-set!	One atomic state transition	hardware

The serializer and the mutex are, in SICP, abstract data types. There is a constructor `make-serializer` that's implemented using a mutex, and a constructor `make-mutex` that's implemented using `test-and-set!`, which is a (simulated, in our case) hardware instruction.

2.1 Serializers

For now, let's look at how this idea can be expressed at a higher level of abstraction, in a Scheme program.

```
(define x-protector (make-serializer))
(define protected-increment-x
  (x-protector
    (lambda () (set! x (+ x 1)))))
> x
100
> (protected-increment-x)
> x
101
```

We introduce an abstraction called a *serializer*. This is a procedure that takes as its argument another procedure (call it `proc`). The serializer returns a new procedure (call it `protected-proc`). When invoked, `protected-proc` invokes `proc`, but only if the *same* serializer is not already in use by another protected procedure. `Proc` can have any number of arguments, and `protected-proc` will take the same arguments and return the same value.

There can be many different serializers, all in operation at once, but each one can't be doing two things at once. So if we say

```
(define x-protector (make-serializer))
(define y-protector (make-serializer))
(parallel-execute
  (x-protector (lambda () (set! x (+ x 1))))
  (y-protector (lambda () (set! y (+ y 1)))))
```

then both tasks can run at the same time. It doesn't matter how their machine instructions are interleaved. But if we say:

```
(parallel-execute
  (x-protector (lambda () (set! x (+ x 1))))
  (x-protector (lambda () (set! x (+ x 1)))))
```

then, since we're using the same serializer in both tasks, the serializer will ensure that they don't overlap in time.

I've introduced a new primitive procedure, `parallel-execute`. It takes any number of arguments, each of which is a procedure of no arguments, and invokes them them, in parallel rather than in sequence. This isn't a standard part of Scheme, but an extension for this section of the textbook.

You may be wondering about the need for all those `(lambda ()...)` notations. Since a serializer isn't a special form, it can't take an expression as argument. Instead we must give it a procedure that it can invoke.

2.2 Implementing Serializers

A serializer is a high-level abstraction. How do we make it work? Here is an *incorrect* attempt to implement serializers:

```
;;;;; In file cs61a/lectures/3.4/bad-serial.scm
(define (make-serializer)
  (let ((in-use? #f))
    (lambda (proc)
      (define (protected-proc . args)
        (if in-use?
            (begin (wait-a-while) ; Never mind how to do that.
                   (apply protected-proc args)) ; Try again.
            (begin (set! in-use? #t) ; Don't let anyone else in.
                   (apply proc args) ; Call the original procedure.
                   (set! in-use? #f)))) ; Finished, let others in again.
      protected-proc)))
```

This is a little complicated, so concentrate on the important parts. In particular, never mind about the *scheduling* aspect of parallelism—how we can ask this process to wait a while before trying again if the serializer is already in use. And never mind the stuff about `apply`, which is needed only so that we can serialize procedures with any number of arguments.

The part to focus on is this:

```
(if in-use?
  .......                    ; wait and try again
  (begin (set! in-use #t)    ; Don't let anyone else in.
         (apply proc args)   ; Call the original procedure.
         (set! in-use #f))) ; Finished, let others in again.
```

The intent of this code is that it first checks to see if the serializer is already in use. If not, we claim the serializer by setting `in-use` true, do our job, and then release the serializer.

The problem is that this sequence of events is subject to the same parallelism problems as the procedure we're trying to protect! What if we check the value of `in-use`, discover that it's false, and right at that moment another process sneaks in and grabs the serializer? In order to make this work we'd have to have another serializer protecting this one, and a third serializer protecting the second one, and so on.

There is no easy way to avoid this problem by clever programming tricks within the competing processes. We need help at the level of the underlying machinery that provides the parallelism: the hardware and/or the operating system. That underlying level must provide a *guaranteed atomic* operation with which we can test the old value of `in-use` and change it to a new value with no possibility of another process intervening. It turns out that there is a tricky software algorithm to generate guaranteed atomic test-and-set, but in practice, there is almost always hardware support for parallelism. Look up "Peterson's algorithm" in Wikipedia if you want to see the software solution.

The textbook assumes the existence of a procedure called `test-and-set!` with this guarantee of atomicity. Although there is a pseudo-implementation on page 312, that procedure won't really work, for the same reason that my pseudo-implementation of `make-serializer` won't work. What you have to imagine is that `test-and-set!` is a single instruction in the computer's hardware, comparable to the LOAD instruction and so on that I started with. This is a realistic assumption. Modern computers do provide some such hardware mechanism, precisely for the reasons we're discussing now.

2.3 The Mutex

The book uses an intermediate level of abstraction between the serializer and the atomic hardware capability, called a *mutex*. What's the difference between a mutex and a serializer? The serializer provides, as an abstraction, a protected operation, without requiring the programmer to think about the mechanism by which it's protected. The mutex exposes the sequence of events. Just as my incorrect implementation said:

```
(set! in-use #t)
(apply proc args)
(set! in-use #f)
```

The correct version uses a similar sequence:

```
(mutex 'acquire)
(apply proc args)
(mutex 'release)
```

By the way, all of the versions in these notes have another bug; I've simplified the discussion by ignoring the problem of return values. We want the value returned by `protected-proc` to be the same as the value returned by the original `proc`, even though the call to `proc` isn't the last step. Therefore the correct implementation is:

```
(mutex 'acquire)
(let ((result (apply proc args)))
     (mutex 'release)
     result)
```

as in the book's implementation on page 311.

3 Programming Considerations

Even with serializers, it's not easy to do a good job of writing programs that deal successfully with concurrency. In fact, all of the operating systems in widespread use today have bugs in this area; Unix systems, for example, are expected to crash every month or two because of concurrency bugs.

To make the discussion concrete, let's think about an airline reservation system, which serves thousands of simultaneous users around the world. Here are the things that can go wrong:

- **Incorrect results.** The worst problem is if the same seat is reserved for two different people. Just as in the case of adding 1 to `x`, the reservation system must first find a vacant seat, then mark that seat as occupied. That sequence of reading and then modifying the database must be protected.
- **Inefficiency.** One very simple way to ensure correct results is to use a single serializer to protect the entire reservation database, so that only one person could make a request at a time. But this is an unacceptable solution; thousands of people are waiting to reserve seats, mostly not for the same flight.

- **Deadlock.** Suppose that someone wants to travel to a city for which there is no direct flight. We must make sure that we can reserve a seat on flight A and a seat on connecting flight B on the same day, before we commit to either reservation. This probably means that we need to use *two* serializers at the same time, one for each flight. Suppose we say something like:

  ```
  (serializer-A (serializer-B (lambda () ...))))
  ```

  Meanwhile someone else says

  ```
  (serializer-B (serializer-A (lambda () ...))))
  ```

  The timing could work out so that we get serializer A, the other person gets serializer B, and then we are each stuck waiting for the other one.

- **Unfairness.** This isn't an issue in every situation, but sometimes you want to avoid a solution to the deadlock problem that always gives a certain process priority over some other one. If the high-priority process is greedy, the lower-priority process might never get its turn at the shared data.