Week 11: Concurrency

This week's material isn't really thoroughly covered in CS61A, but later on (for those who are going to continue with CS) in CS162 this material will make more sense, but we'll touch on this now and also a little in CS61C =)

So what is this whole concurrency business...a new currency? No no no, it's the ability to run multiple things at once also known as parallelism. Just think if you couldn't play music at the same time while you're writing up cs61a notes...what kind of world would that be?

Parallelism

So what happens in your computer when say you want to listen to music and type up notes? A processor can only execute one process at a time. The operating system's job to schedule the processes accordingly, meaning it says for every 1/1000000 of a second send a the next piece of encoding from winamp to the soundcard and then for every 2/1000000 of a second send the next piece of encoding from the sound card to the speaker, then for every 1/1000 of a second poll the cursor and see where it is....etc etc etc.

In CS61C you'll learn that everything can be broken down into assembly. A computer can only execute one instruction per cycle. Let's take a look at a Scheme instruction: (set! x (+ x 1)) . This actually takes three machine instructions:

1) **(set! x (+ x 1))**
   Find the current value for x and save it
2) **(set! x (+ x 1))**
   Add 1 to the value stored for x
3) **(set! x (+ x 1))**
   Store the return value from the addition as x

So now back to the idea of parallelism, or running things at the same time. I can finally let you in on the secret... running parallel processes A and B means interleaving the execution of A's instruction set and B's instruction set. Let's look at this:

Let process A be (set! x (+ x 1)) and B be (set! x (+ x 2)). Here are A and B's instruction set:

A	B
1) Find the current value for x and save it	**1) Find the current value for x and save it**
2) Add 1 to the value stored for x	**2) Add 2 to the value stored for x**
3) Store the return value (from above) as x	**3) Store the return value (from above) as x**

Now where could the problem lie? Well lets see. Suppose x initially has the value 30. After executing A and B (in any order) x's value should be 33. Let's try running A and B in parallel. To do this we use a function parallel-execute that interleaves instructions of procedures with no arguments. To run it we can simply type:

```
(parallel-execute (lambda () A)
                  (lambda () B))
```

Now for running A and B in parallel. Any permutation involving A's instructions and B's instructions is plausible (as long as A1 < A2 < A3 and B1 < B2 < B3). For example here is a plausible permutation: Let's assume x is initially 30

Time		

1	**A1) Find the current value for x and save it (x = 30)**	
2		**B1) Find the current value for x and save it (x = 30)**
3	**A2) Add 1 to the value stored for x (31)**	
4	**A3) Store the return value (from above) as x (x = 31)**	
5		**B2) Add 2 to the value stored for x (32)**
6		**B3) Store the return value (from above) as x (x = 32)**

If these instructions are executed in this order, the final value of x is 32, not 33! Also we could think of a situation where the return value is 31 (try the sequence A1 B1 B2 B3 A2 A3). These are incorrect results. Even while running processes in parallel, we still require correct results! Lets think of a case where this really affects you. Lets say you have a joint bank account with your friend Jojo. Lets say process A is you going to the

bank and depositing $1.00 into your account. Lets say process B is Jojo depositing your start-up $1,000,000 in the same account. Suppose you both go to different ATM machines and try depositing at the same time, so the ATM OS calls parallel-execute on your deposits. Wouldn't you like to have $1,000,001 added to your account rather than $1. This is why we need to serialize.

Serializers

A serializer is a procedure that takes a procedure as an argument and returns a "protected" version of that procedure. The protected version of this procedure comes equipped with a mutex, or mutually exculsive procedure. A mutex can be acquired (or grabbed) and released. The serialized procedure can only continue execution when it "has" or has acquired the mutex. The mutex "guards" the part of the program we call the critical section. Above the changing of the bank account balance is the critical section. For example, if we serialized the example above (with x starting at 30), the only 2 possibilities for interleaving the order of instructions would be A1 A2 A3 B1 B2 B3 or B1 B2 B3 A1 A2 A3. Now with the power of serialization, we are able to avoid a large problem: Incorrect Results. This is the main reason we would ever need to serialize.

There is still a problem with the mutex. What if process A looks at the mutex and sees it is available. Then process B sees the same mutex is available before process A can grab it. Thus A and B both believe they have the mutex and once again we have incorrect results. We have to insure that once we see a mutex is available we immediately grab it. A piece of hardware implementing test-and-set! takes care of this for us, it both tests and sets a mutex to taken in one instruction. This is necessary to ensure the correctness of our serializer.

Even if we serialize we are not guaranteed to succeed in terms of results or efficiency. Let's take a look at our make-account procedure from previous homeworks:

```
(define (make-account balance)
  (define (withdraw amount)
    (begin
          (set! balance (- balance amount))
          balance)))
  (define (deposit amount)
    (begin
       (set! balance (+ balance amount)))
          balance))
  (define (dispatch msg)
```

```
  (cond ((eq? msg 'withdraw) withdraw)
        ((eq? msg 'deposit) deposit)
        ((eq? msg 'balance) (lambda () balance))
        (else (error: Unknown.....))))
 dispatch)
```

Now, what if we had joint accounts (remember that homework question...) and two people tried to access the same variable balance at the same time? Well, we had better hope that that variable is serialized. Thus we need to serialize accesses to balance. This is our critical section. There are multiple options for serializing. Should we serialize withdraw? deposit? dispatch? or balance? Well only one of these work and lets look at why:

* **withdraw?**
  What if 2 people deposit money simultaneously, this is not protected -> incorrect.
* **deposit?**
  What if 2 people withdraw money simultaneously, this is not protected -> incorrect.
* **withdraw and deposit each are serialized?**
  Since each has a different serializer, each has a different mutex. Thus you only need one mutex to access balance and a parallel-executed withdraw and deposit can still have both access to the same unprotected balance. -> incorrect
* **make-account**
  Well this would ensure correct results, however what if there are 2 different accounts?
  Its fine to interleave two different accounts withdraws and deposits. Here we are extremely inefficient because the critical section is too large.

This leaves us with serializing dispatch, which makes sense because whenever we call a dispatch we want to lock down that balance, nothing more and nothing less since all calls to dispatch access balance. Here is our correct implementation of make-account.

```
(define (make-account balance)
 (define (withdraw amount)
  (begin
       (set! balance (- balance amount))
       balance)))
 (define (deposit amount)
  (begin
    (set! balance (+ balance amount)))
       balance))
 (let ((protected (make-serializer)))
```

```
(define (dispatch msg)
  (cond ((eq? msg 'withdraw) (protected withdraw))
        ((eq? msg 'deposit) (protected deposit))
        ((eq? msg 'balance) (lambda () balance))
        (else (error: Unknown.....))))
  dispatch))
```

Things that can go wrong:

So far we know of 2 problems involving serialization: Inefficiency (critical section too large), and Incorrect Results (critical section too small). There is (classically) one more problem we might encounter: deadlock. Deadlock is a problem with a cyclic dependency between 2 or more serializers. If processes A and B are both protected by serializers s and t. If A has s and B has t, then both A and B will wait forever for the other mutex. This is called deadlock. Here are some more examples. Try them on your own:

What are the possible values for baz after each of the following sequences?

(a)

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

(parallel-execute (s (lambda () (set! baz (/ baz 2))))
                  (s (lambda () (set! baz (+ baz baz)))))
```

(b)

```
(define baz 10)

(parallel-execute (lambda () (set! baz (/ baz 2)))
                  (lambda () (set! baz (+ baz baz))))
```

(c)

```
(define baz 10)
(define s (make-serializer))
(define t (make-serializer))

(parallel-execute (s (lambda () (set! baz (/ baz 2))))
                   (t (lambda () (set! baz (+ baz baz)))))
```

(d)

```
(define baz 10)
(define s (make-serializer))
(define t (make-serializer))

(parallel-execute (t (s (lambda () (set! baz (/ baz 2)))))
                  (s (t (lambda () (set! baz (+ baz baz))))))
```

Answers:
(a) Only 10, this is correct
(b) 10 (correct interleaving), or 5, 20, 15 (all wrong)
(c) 10 (correct) or 5, 20, 15. The key point is that because the 2 processes do not share a serializer, they never prevent the other from accessing data. In other words, two unrelated serializers do not affect each other!
(d) 10 (correct) or deadlock. If one gets s and the other gets t they must wait forever to get the other!

*** thanks to Todd Segal for these notes =) * Super Monkey Ball rules!**