**CS61A Notes – Week 10: Vectors, concurrency**

**A Concurrent March Through Programming Hell**

*On your computer, you often have multiple programs running at the same time – you might have your internet browser open browsing questionable pictures, your P2P software downloading non-pirated software, and your instant messaging client lying to a clueless middle-schooler across the country. But you have only one computer, and one CPU! How can you do so many things at once?*

*What actually happens is, the CPU switches between the different processes very quickly, doing work for each for a little while before moving on to the next, creating the illusion that the programs are running concurrently. The benefits are obvious for users like us so used to multitasking.*

*Unfortunately, parallelism is one of the biggest headaches you’ll encounter. We’ll attempt to give you a tiny migraine, but in CS162, you’ll be swimming in a large ocean of pain with no shore in sight and a leaking life jacket.*

A bit of syntax. To run things concurrently, we use a Scheme primitive called parallel-execute, a procedure that takes in any number of “thunks” – procedures that take no arguments – and executes the thunks in parallel. For example,

(define x 5)

(parallel-execute (lambda () (set! x (+ x 10)))

(lambda () (set! x (+ x 20))))

will attempt to set x to (+ x 10) and set x to (+ x 20) at the same time. Again, the computer “cheats” by interleaving operation between the different thunks. The answer that we want, of course, is 35 – we want the two thunks executed at the same time, but we still want the result to be *as if they executed consecutively*.

Now, consider this simple Scheme expression:

(set! x (+ x 10))

What looks like one Scheme operation is actually *three* operations:

1. lookup the value of x
2. add the value of x to 10
3. store the result into x

Thus, consider the above call to parallel-execute, and keep in mind that the two thunks can be interleaved arbitrarily:

|  |  |
| --- | --- |
| * lookup value of x * add 10 to the value of x * set x to the result | * lookup value of x * add 20 to the value of x * set x to the result |

If the operations were interleaved in the above manner (not interleaved at all), then the value of x at the end is 35.

|  |  |
| --- | --- |
| * lookup value of x * add 10 to the value of x * set x to the result | * lookup value of x * add 20 to the value of x * set x to the result |

In the above interleaving, the value of x ends up being 15. This is not what we wanted!

**QUESTION: What are the possible values of x after the below?**

**(define x 5)**

**(parallel-execute (lambda () (set! x (\* x 2)))**

**(lambda () (if (even? x)**

**(set! x (+ x 1))**

**(set! x (+ x 100)))))**

**Concurrency: The Series**

We use something called “serializers” to make sure that certain chunks of code are executed *together*. First, we need a way to create a serializer:

(define x-protector (make-serializer))

That was easy. A serializer takes in a procedure, and creates a *serialized* version of that procedure. So,

(define protected-plus-10 (x-protector (lambda () (set! x (+ x 10)))))

(define protected-plus-20 (x-protector (lambda () (set! x (+ x 20)))))

protected-plus-10 still does the same thing as the original thunk – take in no arguments, and add 10 to x. However, because protected-plus-10 and protected-plus-20 are created *with the same serializer*, their instructions *will not be interleaved*. Therefore, in doing,

(parallel-execute protected-plus-10 protected-plus-20)

you can always be sure that x will be set to 35 at the end.

There’s also a primitive object called a “mutex” that’s even lower level than serializers (in fact, serializers are implemented with mutexes). You can interact with a mutex this way:

(define m (make-mutex))

(m ‘acquire) ;; “reserves” the mutex

(m ‘release) ;; “releases” the mutex

Once one program has acquired a mutex, if another wants to acquire the same mutex, it must wait until the mutex is released. So we can do this to obtain the same result:

(define x-mutex (make-mutex))

(parallel-execute

(lambda () (x-mutex ‘acquire) (set! x (+ x 10)) (x-mutex ‘release))

(lambda () (x-mutex ‘acquire) (set! x (+ x 20)) (x-mutex ‘release)))

The calls to acquire and release a mutex marks the *critical sections* of the code – sections that should *not* be interleaved with other processes also needing the same mutex.

When working with concurrency, there are four potential kinds of problems:

1. **incorrectness –** like the second interleaving example above, the answer you get might just be wrong
2. **inefficiency** – you could lock up the whole computer and always run only one program at a time, but that's horribly inefficient
3. **deadlocks** – if two programs are competing for the same two resources, there can be deadlocks
4. **unfairness** – one program may be unfairly favored to do more work than another

**QUESTION: The Dining Politicians Problem. Politicians like to congregate once in a while, eat and spew nonsense. One slow Saturday afternoon, three politicians meet to have such wild fun. They sit around a circular table; however, due to the federal deficit (funny that these notes are timeless), they are provided with only three chopsticks, each lying in between two people. A politician will be able to eat only when both chopsticks next to him are not being used. If he cannot eat, he will just spew nonsense.**

1. **Here is an attempt to simulate this behavior:**

**(define (eat-talk i)**

**(define (loop)**

**(cond ((can-eat? i)**

**(take-chopsticks i)**

**(eat-a-while)**

**(release-chopsticks i))**

**(else (spew-nonsense)))**

**(loop)**

**(loop))**

**(parallel-execute (lambda () (eat-talk 0))**

**(lambda () (eat-talk 1))**

**(lambda () (eat-talk 2)))**

**;; a list of chopstick status, #t if usable, #f if taken**

**(define chopsticks ‘(#t #t #t))**

**;; does person i have both chopsticks?**

**(define (can-eat? i)**

**(and (list-ref chopsticks (right-chopstick i))**

**(list-ref chopsticks (left-chopstick i))))**

**;; let person i take both chopsticks**

**;; assume (list-set! ls i val) destructively sets the ith element of**

**;; ls to val**

**(define (take-chopsticks i)**

**(list-set! chopsticks (right-chopstick i) #f)**

**(list-set! chopsticks (left-chopstick i) #f))**

**;; let person i release both chopsticks**

**(define (release-chopsticks i)**

**(list-set! chopsticks (right-chopstick i) #t)**

**(list-set! chopsticks (left-chopstick i) #t))**

**;; some helper procedures**

**(define (left-chopstick i) (if (= i 2) 0 (+ i 1)))**

**(define (right-chopstick i) i)**

**Is this correct? If not, what kind of hazard does this create?**

1. **Here's a proposed fix:**

**(define protector (make-serializer))**

**(parallel-execute (protector (lambda () (eat-talk 0)))**

**(protector (lambda () (eat-talk 1)))**

**(protector (lambda () (eat-talk 2))))**

**Does this work?**

1. **Here’s another proposed fix: use one mutex per chopstick, and acquire both before doing anything:**

**(define protectors**

**(list (make-mutex) (make-mutex) (make-mutex)))**

**(define (eat-talk i)**

**(define (loop)**

**((list-ref protectors (right-chopstick i)) ‘acquire)**

**((list-ref protectors (left-chopstick i)) ‘acquire)**

**(cond ... ;; as before)**

**((list-ref protectors (right-chopstick i)) ‘release)**

**((list-ref protectors (left-chopstick i)) ‘release)**

**(loop))**

**(loop))**

**Does that work?**

1. **What about this:**

**(define m (make-mutex))**

**(define (eat-talk i)**

**(define (loop)**

**(m ‘acquire)**

**(cond ... ;; as before)**

**(m ‘release)**

**(loop))**

**(loop))**

1. **So what would be a good solution?**

***(Note: This problem is commonly referred to as “The Dining Philosophers” problem. However, here at Berkeley, we prefer to look down on politicians rather than philosophers.)***