Quiz 14a Rubric

1. (2+1+1 points) Another way of dealing with concurrency is to mark sections of code as critical. It’s pretty simple – a critical section of code cannot be executed in parallel with anything else. So, a critical section of code and a non-critical section of code cannot run in parallel.
   1. Suppose we have a procedure critical that takes as input a procedure of no arguments, and returns a new procedure of no arguments, which is now a critical section of code. In the code fragment below, add calls to critical so that we only get “correct” values of x. (Correct means that we could have gotten that value of x by executing the procedures sequentially in some order.) You should use critical as little as possible.

; x has been defined to be 10

(parallel-execute (lambda () (set! x 1))

(lambda () (set! x 4))

**(critical** (lambda () (set! x (+ x 2)))**)**)

Notice that the “critical” must be on the last procedure, because that one reads and writes x. The others write constants to x, which are assumed to be atomic (the hardware usually enforces this, and this is the assumption SICP uses).

2 points for the correct answer

1 point for making any two of the procedures critical

0.5 points for making all three procedures critical.

0 points for making none of them critical.

* 1. Now do the same thing using one serializer. (Again, minimize the use of serializers.)

(define x-serializer (make-serializer))

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

**(x-serializer** (lambda () (set! x 4))**)**

**(x-serializer** (lambda () (set! x (+x 2)))**)**)

1 point, all or nothing (but don’t mark off for minor syntax errors and typos)

* 1. Explain one disadvantage of critical sections (as compared to serializers or mutexes). (Hint: Think about the three problems – incorrectness, deadlock and inefficiency.)

Note: Parts a and b are **not** a hint for this problem.

Critical sections force the section of code to be executed alone, even though it may be safe to run it with certain other sections of code. We aren’t allowed to say “don’t run the code in parallel with A and B, but you can run it in parallel with C”, which we can do with serializers and mutexes. This can lead to inefficiency.

1 point, probably all or nothing. They only need to get the idea right – the fact that this could lead to inefficiency.

1. (1+1+2 points) Here is a procedure that tries to compute a \* (a+1) \* … \* (b-1) \* b quickly by taking advantage of parallelism:

(define (fast-product a b)

; base cases hidden for compactness

(let ((mid (/ (+ a b) 2))

(result 1))

(parallel-execute

(lambda () (set! result (\* result (fast-product a mid))))

(lambda () (set! result (\* result (fast-product mid b)))))

result))

* 1. Give at least one scenario in which this procedure could give incorrect results. (A timing diagram or a description of a timing diagram in words is sufficient.)

The first lambda reads result, then the second lambda reads result, then the first lambda writes to result, and then the second lambda writes to result. (There are other solutions as well.) (1 point, all or nothing)

* 1. Suppose to fix this we just serialized both the lambdas entirely:

(define (fast-product a b)

; base cases omitted for simplicity

(let ((mid (/ (+ a b) 2))

**(s (make-serializer))**

(result 1))

(parallel-execute

**(s** (lambda () (set! result (\* result (fast-product a mid))))

**(s** (lambda () (set! result (\* result (fast-product mid b)))))

result))

This is inefficient (essentially, it’s the same as if we hadn’t parallelized at all). However, Cy D Fect argues that it is also incorrect! The reason is that for each recursive call, we create an entirely new serializer. Taking (fast-product 0 8) as an example, the recursive call (fast-product 0 2) has a different serializer than (fast-product 4 8). So, they could execute concurrently, and both of them could try to set! result at the same time, which could lead to the problem in the previous part.

The oracles have informed you that Cy is incorrect. What’s wrong with his argument?

The problem is that the result set by (fast-product 0 2) is different from the one set by (fast-product 4 8) – they are in different frames. So, there’s no problem in setting them concurrently – they can’t interfere with each other.

Another answer: (fast-product 0 2) was a recursive call from (fast-product 0 4), which was serialized with the same serializer as (fast-product 4 8), so it must finish first. (Essentially, all of the recursive calls of the first thunk must be done before the second thunk even starts.)

* 1. Rewrite fast-product to fix this inefficiency (you can omit the base cases as done previously). Here’s the rubric:

2 points: Correct, efficient procedure

1 point: Correct, inefficient procedure

0 points: Incorrect procedure

parallel-execute returns okay, so you do actually need to use set! inside the lambdas.

Hint: What was the problem in a? Presumably it had something to do with set!. How could you change it so that the calls to set! no longer cause problems?

(define (fast-product a b)

; base cases omitted for simplicity

(let ((mid (/ (+ a b) 2))

(res1 1)

(res2 1))

(parallel-execute

(lambda () (set! res1 (fast-product a mid))))

(lambda () (set! res2 (fast-product mid b)))))

(\* res1 res2)))

If the procedure is incorrect, but not because of concurrency problems, then you may want to take off fewer points. Use your best judgment (and remember that half-points are allowed).

1. (2 points) Remember our Tic-Tac-Toe game from discussion 7? Now we want to turn it into a client/server program, so that players don’t have to be at the same computer to play. Here are the classes and their methods that we had previously:

(define-class (board)

(instance-vars (grid (make-grid)))

(method (piece x y) (get-piece grid x y))

(method (play-move piece x y)

(set! grid (next-grid grid piece x y))))

; Initially, a player is not playing a game. To play a game, the start-game method must be called.

(define-class (player name)

(instance-vars (game #f) (player-piece #f))

(method (start-game piece board)

(set! game board) (set! player-piece piece))

(method (make-move)

(let ((move (read)))

(ask game 'play-move player-piece (car move) (cadr move)))))

1. Where should the two classes be implemented? Each answer should be “Client” or “Server”.

Board: Server

Person: Client

1. Give at least one new request type that should be added. (A request type is something like receive-msg, broadcast, etc.) For each request type, explain who sends it (client or server), and what should happen when the request is received. The first one has been done for you.

Start-game: Sent by the server to the client. When the client receives it, the client starts a new game (corresponding to the start-game method of the Player class above).

Make-move: Sent by the client to the server. The request includes a move. When the server receives it, it makes a note of the move (corresponding to the play-move method).

Play-move: Sent by the server to the client. No data (or possibly the board). When the client receives it, it sends back a make-move request.

Either one of these gets full credit. If they put some other request type that makes sense, that should get full credit as well.

If they have at least one request with the right idea, they get at least 0.5 points. Ways they could lose the other 0.5 points:

* Having redundant requests (such as a make-move and a play-move both from the client to the server), but it’s fine if they have something that encodes more functionality, if it makes sense for that to be a request
* Sending a move from server to client instead of client to server
* Other such things.