Name:

CS61as-\_\_

**Quiz 14b**

1. (3+2+2 points) Alice and Bob are both responsible for updating a database, which is divided into 5 sections. Each person has a “permission list” containing all the sections that they’re allowed to edit. Each section has a mutex associated with it to prevent multiple people from trying to edit it at once. When someone logs on to the database, the mutex for each section in their permission list is acquired:

(define (logon username permission-list)

(if (null? permission-list)

(start-session username)

(begin ((get-mutex (car permission-list)) 'acquire)

(logon username (cdr permission-list)))))

> (logon 'alice (list section1 section4 section5))

(welcome alice, you may now edit sections 1, 4, and 5)

Similarly, when the user logs out of the system, the mutexes for all of that person’s sections are released.

a. (3 points) Note that if one of Alice’s sections is also in Bob’s permission list, Bob will be stuck waiting for access to the database until Alice logs out. This is inconvenient, but there is a bigger problem with the system. Give a pair of permission lists that can cause deadlock when Alice and Bob try to log on simultaneously, and explain how the deadlock occurs.

Example answer:

Alice: (list section1 section4 section5)

Bob: (list section2 section5 section4)

Alice acquires section1 and section4, then Bob acquires section2 and section5. Now Alice is waiting for Bob to release section5, and Bob is waiting for Alice to release section4.

• Correct: 2 points

• Correct lists, explanation missing or off in wrong direction: 1 point

• Incorrect list, but explanation demonstrates understanding of deadlock: 1 point

b. (2 points) Give a different pair of permission lists that cause unfairness rather than deadlock, and explain how the unfairness occurs. In this case, unfairness means that if both people try to log on at the same time, one person is more likely to get access than the other.

Example answer:

Alice: (list section1 section4 section5)

Bob: (list section2 section3 section1)

Bob acquires section2 and section3, then Alice acquires section1 and section4. Now Bob is waiting for Alice to release section1, so he has to wait for Alice to finish updating the database even though he started logging on first.

• Correct: 2 points

• Correct lists, explanation missing or off in wrong direction: 1 point

c. (2 points) Suggest a way of making the database more efficient. Specifically, explain how Alice and Bob could avoid updating the same data simultaneously without completely locking each other out of the database. This is an open-ended question, and you don’t have to explain all of the implementation details of your system.

Example answers:

1. Have Alice and Bob request access to the sections individually, instead of all at once.
2. When a section of the database is in use, just ignore it and request the other ones.
3. Acquire each section just before updating its data and release it as soon as it has the new data
4. Give Alice and Bob local copies of their database sections when they log in, and update the main database when they log out.
5. Any other answer that prevents data races (specifically, one person writing while the other person reads or both people writing at once) and mutual exclusion (specifically, one person being locked out of the entire database while the other person is updating a single section).

• Any practical suggestion that prevents data races and mutual exclusion: 2 points

• Suggestion not possible to implement, but shows an attempt to avoid data races and mutual

exclusion: 1 point

• Suggestion clearly results in data races or mutual exclusion: 0 points

2. (3 points) Consider the procedure squares!, which squares every element of a list in place:

(define (squares! lst)

(if (not (null? lst))

(begin (set-car! lst (square (car lst)))

(squares! (cdr lst)))))

> (define lst '(3 4 6))

> (squares! lst)

> lst

(9 16 36)

Use parallel-execute to rewrite squares! so that the squaring is done in parallel. Your

new implementation should not impose any restrictions on the order in which squares are

computed.

(define (parallel-squares! lst)

(if (not (null? lst))

(parallel-execute (lambda () (set-car! lst (square (car lst))))

(lambda () (parallel-squares! (cdr lst))))))

• Correct: 2 points

• Calling squares! instead of parallel-squares!: 1.5 points

• Missing lambdas: 1 point (they should have seen this in the homework)

• parallel-execute called in the wrong place: 0 points