Brian Bauman

CSC 461

Homework 4

1.

In order to satisfy the third liveness condition, a process that would like to receive a token and enter a critical section must first request permission to do so from all other processes. Once those processes respond with permission, it sends a request to the coordinator and receives the token.

- If the process is not waiting for a token and receives a request for permission, it responds with permission

- If a process receives a request for permission and either already has the token or has made a request for the token, it waits until it receives and returns the token before responding to the requesting process with its permission.

- If the process receives a request for permission and is still waiting for its *own* permission requests to return, it compares the timestamp of the request that it sent to the requesting process with the timestamp of the request *from* the requesting process. If the requesting processes’ timestamp is lower, it responds with permission, otherwise, it waits to respond until it has received and returned the token.

This ensures that the order in which the coordinator receives the requests matches causal order. In the case in which two requests to enter the critical section are concurrent, the order no longer matters and the speed of message delivery will determine the order of token receipt. This strategy is identical to the one covered in class, but uses a centralized token and coordinator.

2.

T: x = read(i); write(j, 44);

U: write(i, 55); write(j, 66);

(a)

An interleaving that is not serialized could be:

- U: write(i, 55)

- T: x = read(i)

- T: write(j, 44)

- U: write(j, 66)

In this interleaving, one object (a\_i) sees U as the first transaction to execute, but another (a\_j) sees T as the first transaction.

(b)

An interleaving that is serialized could be:

- T: x = read(i)

- T: write(j, 44)

- U: write(i, 55)

- U: write(j, 66)

3.

(a)

Serialized: Yes

Locking schedule:

- U takes a lock on a\_i, a\_k, and a\_l.

- U writes to a\_i and releases the lock on a\_I

- T takes a lock on a\_i and a\_j

- T reads from a\_j and a\_i

- U writes to a\_k and releases the lock on a\_k

- T takes a lock on a\_k

- T writes to a\_k and releases the locks on a\_i, a\_j, and a\_k

- U writes to a\_l and releases the lock on a\_l

Strict/non-strict: Non-strict

Dirty reads: Yes

(b)

Serialized: No

(c)

Serialized: Yes

Locking schedule:

- T takes a lock on a\_j

- T reads a\_j

- U takes a lock on a\_i, a\_k, and a\_l

- U writes to a\_i and a\_k, and releases the locks on a\_i and a\_k

- T takes a lock on a\_i and a\_k

- T reads from a\_i, writes to a\_k, and releases the locks on a\_i and a\_k

- U writes to a\_l and releases the lock on a\_l

Strict/non-strict: Non-strict

Dirty reads: Yes

(d)

Serialized: Yes

Locking schedule:

- T takes a lock on a\_i, a\_j, and a\_k

- T reads from a\_j and a\_i, writes to a\_k, and releases all three locks

- U takes a lock on a\_i, a\_k, and a\_l

- U writes to a\_i, a\_k, and a\_l, and releases all three locks

Strict/non-strict: Strict

Dirty reads: No

(e)

Serialized: Yes

Locking schedule:

- U takes a lock on a\_i, a\_k, and a\_l

- U writes to a\_i and releases the lock on a\_i

- T takes a lock on a\_j

- T reads from a\_j

- U writes to a\_k and a\_l and releases the locks on a\_i, a\_k, and a\_l

- T takes a lock on a\_i and a\_k

- T reads from a\_i and releases the locks on a\_i and a\_j

- T writes to a\_k and releases the lock on a\_k

Strict/non-strict: Non-strict

Dirty reads: No (U commits before T reads a\_i)

4.

(a)

Yes, this is possible.

(b)

No, this is not possible.

(c)

No, this is not possible.

(d)

Yes, this is possible.

(e)

Yes, this is possible.

5.

P1 : W(x)0; W(x)1; R(y)1

P2 : R(x)1; W(y)1; R(x)0

There is no sequential ordering of P1 and P2 that would produce this behavior, therefore, they are not sequentially consistent.

6.

P1 : R(y)2; R(x)0

P2 : W(x)0; W(x)1

P3 : R(x)1; W(y)2

Because P3’s R(x)1 must precede its W(y)2, both of these operations must precede P1’s R(y)2. Also, P3’s R(x)1 must follow P2’s W(x)1, which implies that it must also follow P2’s W(x)0. With these constraints, the order of operations must be P2 —> P3 —> P1. However, in this scenario, P1 could not successfully perform a R(x)0, as P2’s W(x)1 has already occurred. Therefore, there is no ordering of these events that resolves into a causally consistent execution.