cds_s2_yz709.pdf

Question 1

Exercise 1 : (*Q3 from s1.pdf*)
A system using the generalised producer-consumer implementation suffers from priority inversion.

(a) A portion of the priority inversion arises from low-priority producers starving high-priority consumers waiting for one another via 'guard'. Why might using a mutex for mutual exclusion, rather than a semaphore, make this an easier problem to mitigate?

- For mutex, only one thread has exclusive access to it once it acquires the lock, but for semaphore, the value of semaphores can be changed by any threads.
- With a mutex, we can implement the priority inheritance protocol; the mutex would inherit the priority of the current owner. If a higher priority thread comes, it would temporarily increase the current priority to the highest of all blocked threads. Once the thread executes the critical region, its priority will be converted back. This avoids the cases where a lower-priority thread runs the critical region and blocks the higher-priority threads
- But with semaphores, when the low-priority producer and high-priority consumer share the same resource, the high-priority consumer has to wait until the low-priority consumer releases the resource by changing the semaphore back. Since all threads own the semaphore, setting a priority to the semaphore does not help.

(b) Another portion of the priority inversion arises from low-priority consumers starving high-priority consumers. How might this problem be addressed?

- If low-priority consumers and high priority consumers share the same critical section (e.g., both consumers want to consume elements from the buffer), the low-priority consumer enters the CS first. The high priority consumer comes and needs to wait until the low-priority consumer exists the CS.
- The priority inheritance protocol could be implemented to avoid this portion of priority inversion. Suppose a high-priority consumer comes to the critical section while the low-priority consumer is executing inside. In that case, we could temporarily boost the low-priority consumer to high-priority until the critical section exists.

(c) A final portion of the priority inversion arises from low-priority producers being starved by an unrelated medium-priority thread while high-priority consumers wait. How might this problem be mitigated?

- This case occurs when a low-priority producer obtains the lock and is preempted by a mid-priority thread. The high-priority consumer requiring the lock would not be able to move forward and needs to wait indefinitely.
- We could implement ceiling protocol by setting up a global highest priority for the critical section. The low-priority consumer in the critical section could temporarily raise its priority to the ceiling priority, preventing high-priority consumers from preempting it.

Comments:
I have some concerns about the question; here are my thoughts:
For (a) we have a low-priority producer and a high-priority consumer sharing a critical section;
For (b) we have a low-priority consumer and a high priority consumer sharing a critical section, I think we can implement priority inheritance on both to prevent priority inversion;
For (c), priority inheritance does not work since the medium-priority thread does not share the critical section, we have to implement the priority ceiling protocol instead.

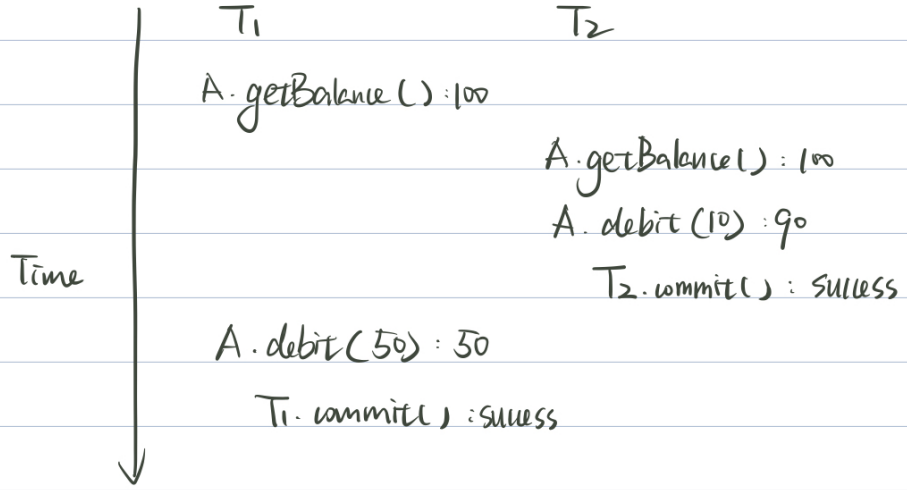
However, I don't know whether we need to implement (a) and (b) differently since one has producer-consumer and another has consumer-consumer.

Question 2

Exercise 2 :
On slide 178/231 three effects of bad schedules are described abstractly: lost-updates, dirty-reads, and unrepeatable-reads. For each of them provide an example consisting of transactions and an execution that results in the bad effect.

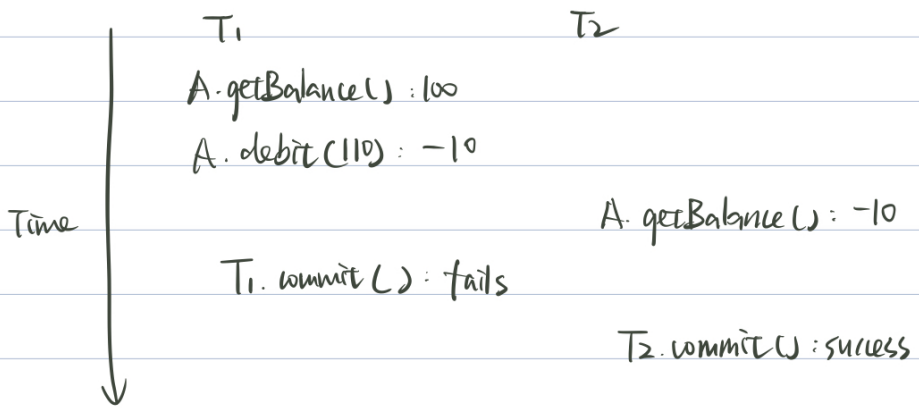
Q2

— For lost-updates: update done to the data item is lost since it is overwritten by the update done by another transaction.

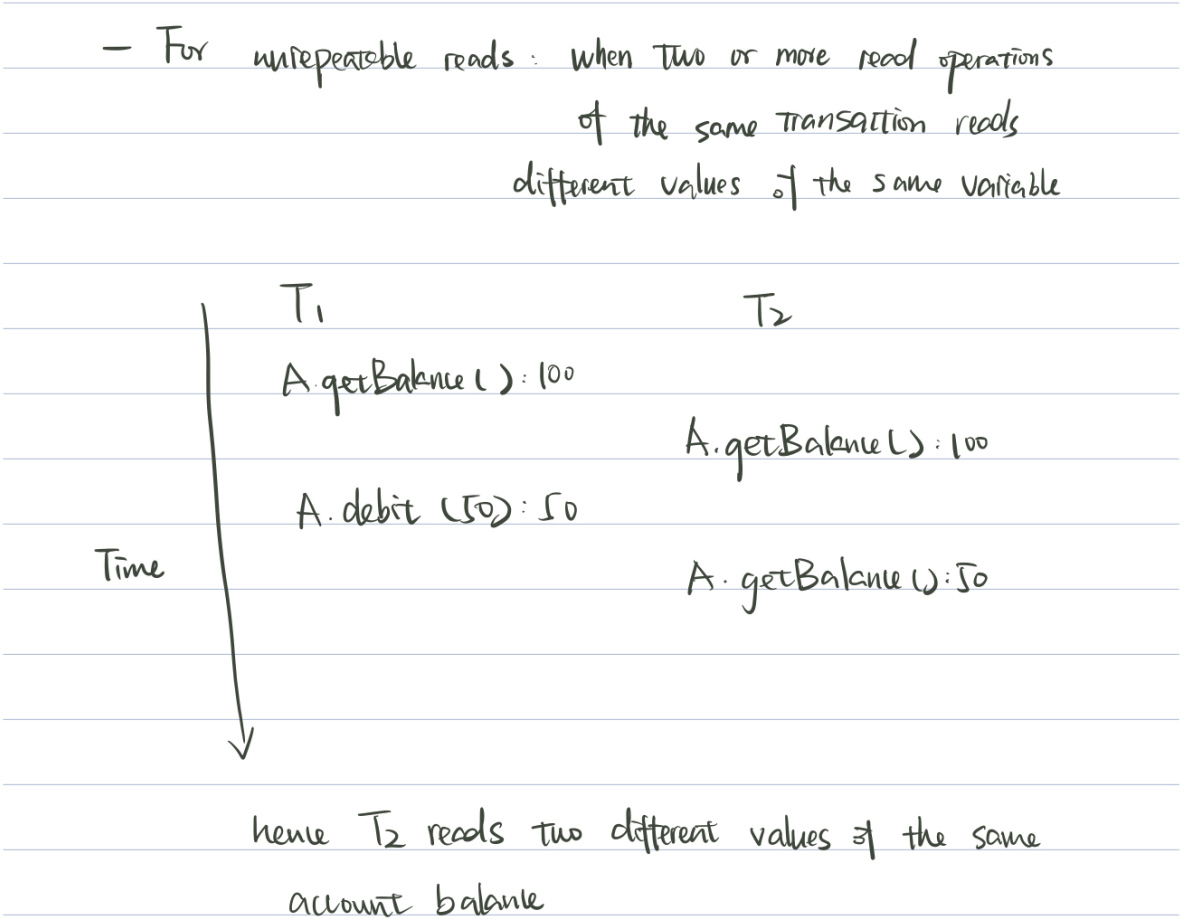


hence the updates done by T2 is overwritten by T2 and is lost.

— For dirty reads: one transaction updates an item and failed. But the updated item is read by another transaction before its value is converted back



hence T2 reads an incorrect value of account A's balance



Comments:

Question 3

Exercise 3 : (*adapted Q1.b from s2.pdf*)
Define the terms *serial* and *(conflict-)serialisible* from the lecture in your own words.
For the following two transactions, enumerate all interleavings under the assumption that individual operations on objects are atomic. For each, is it a serial execution? A conflict-serialisable execution?

```
transaction T1 {
    a = A.getBalance();
    A.credit(INTEREST*a);
}

transaction T2 {
    A.debit(100);
    B.credit(100);
}
```

- Serial: operations from two different transactions are not interleaved.
- Serialisables: the results of operations are identical to the case when serially executing the transactions.
- Conflict-serialisable: if we can turn a schedule S into another serial schedule T by swapping non-conflicting operations
  - Conflict operations belong to different transactions, operate on the same data item, and at least one write operation.
- Possible interleavings - $3 + 2 + 1 = 6$:
  - $R_{T1}(A), W_{T1}(A), W_{T2}(A), W_{T2}(B)$:
    - serial and conflict-serialisable
  - $R_{T1}(A), W_{T2}(A), W_{T1}(A), W_{T2}(B)$:
    - non-serial and non conflict-serialisable
  - $R_{T1}(A), W_{T2}(A), W_{T2}(B), W_{T1}(A)$:
    - non-serial and conflict-serialisable
    - e.g., exchange $R_{T1}(A)$ and $W_{T2}(B)$
  - $W_{T2}(A), R_{T1}(A), W_{T1}(A), W_{T2}(B)$:
    - non-serial and non conflict-serialisable
  - $W_{T2}(A), R_{T1}(A), W_{T2}(B), W_{T1}(A)$:
    - non-serial and conflict serialisable
    - e.g., exchange $R_{T1}(A)$ and $W_{T2}(B)$
  - $W_{T2}(A), W_{T2}(B), R_{T1}(A), W_{T1}(A)$:
    - serial and conflict serialisable

Comments:

Question 4

Exercise 4 : (*Q1.acd from s2.pdf*)
The guarantees of transactional systems are described by the ACID properties: atomicity, consistency, isolation, and durability. Transaction systems shift the burden of managing concurrency away from the end programmer into databases, language runtimes, and operating systems – and by masking its effects, allow flexibility in how the ACID properties are implemented.

(a) In the context of multi-threading, *atomicity* meant something different from its meaning in transaction systems. If we simply associate locks with objects in a transaction system to implement 2-phase locking (2PL), which ACID properties fall out naturally, and which require additional work? Why?

- Isolation would fall out naturally because two locks cannot guarantee a serialisable schedule; when transaction systems execute many transactions concurrently, a non-serialisable schedule will introduce inconsistent behaviours with a serialisable schedule.
- Hence we need to do additional work for isolation using protocols that concern the positioning of locking and unlocking operations.

(b) In transaction systems, 'atomicity' refers to transactions being committed fully or not at all. Why might it be easier to implement transaction atomicity with optimistic concurrency control than with 2PL?

- Optimistic concurrency control assumes conflicts are rare and execute transactions on data copies. We could have easy rollbacks for failing commits by deleting the copy; for successful commits, we could replace the original value with the data copy; these operations can be done at once, so more straightforward to ensure atomicity.
- With 2PL, there might be multiple transactions executing, and rollbacks involve cascading aborts, hence much more difficult when applying atomicity.

(c) One limitation of 2PL is that it can suffer *cascading abort*, when it implements *isolation* rather than *strict isolation*. In the above transactions of exercise 3, `B.credit(100)` may abort due to the resulting balance exceeding a yearly limit; illustrate a 2PL schedule in which aborting T2 triggers a cascading abort of T1. Why would strict 2PL have prevented this problem?

- Suppose T2 updates A, releases the write lock of A, then T1 can read the value A and update it by multiplying by 2 (i.e., $2A$); if T2 aborts, we have to abort T1 as well.
- With strict 2PL, all locks need to hold until the transaction ends, so T1 cannot be executed when T2 is running; thus, T1's fate is not tied to T2, avoiding the cascading abortion.

💡 Comments:

Question 5

Exercise 5 : (*Q2 from s1.pdf*)
2-phase locking (2PL) is one scheme for implementing isolation in the presence of concurrent transaction processing on multiple objects. It consists of two monotonic phases: lock expansion and lock contraction. For this question, we will work with the following transactions:

```
transaction T1 {
    x = A.read();
    B.write(x);
}

transaction T2 {
    x = B.read();
    C.write(x);
}

transaction T3 {
    x = C.read();
    A.write(x);
}
```

Imagine that the transaction scheduler, in an effort to offer fairness to transactions, interleaves scheduling with lines of code in a round-robin manner. For example, it would, without the intervention of locking, schedule two transactions as follows:

```
Epoch 1: T1.line1, T2.line1
Epoch 2: T1.line2, T2.line2
...
```

If a transaction is blocked on a lock, then the scheduler will skip that transaction and proceed to the next transaction in the epoch.

(a) If we use naïve 2PL to implement transactions T1, T2, and T3, what schedule will be selected?

- $R_1(A), R_2(B), R_3(C), W_1(B), W_2(C), W_3(A)$
- If only one lock is associated with one object, we would have a deadlock in Epoch 2 because none of the acquired locks has been released.

```
# Expanding phase
Epoch 1: T1 lock A, T2 lock B, T3 lock C
Epoch 2: T1 lock B, T2 lock C, T3 lock A -> deadlock
# Shrinking phase
```

```
Epoch 3: T1 unlock A, T2 unlock B, T3 unlock C
Epoch 4: T1 unlock B, T2 unlock C, T3 unlock A
```

(b) Deadlock presents a serious challenge to 2PL in the presence of composite operations. Imagine that the transaction system implements a simple deadlock detector in which, if there are in-flight transactions but none can be scheduled, then all in-flight transactions will be aborted (and their locks released) to restart in the next epoch. What schedule arises with the above three transactions?

```
Epoch 1: T1 lock A, T2 lock B, T3 lock C
Epoch 2: T1 lock B, T2 lock C, T3 lock A -> detect deadlock
Epoch 3: T1 unlock A, T2 unlock B, T3 unlock C
Epoch 4: T1 lock B, T2 lock C, T3 lock A
Epoch 5: T1 unlock B, T2 unlock C, T3 unlock A
```

(c) Livelock can also be a challenge for 2PL in the presence of a naïve deadlock resolution scheme. How might the scheme in (b) be modified to provide guarantees of progress in the event that a deadlock is detected?

- Livelock is when threads execute but make no progress
- Once the deadlock is detected, release all acquired locks and put all threads in a queue, only allowing one thread to continue executing at a time.

```
Epoch 1: T1 lock A, T2 lock B, T3 lock C
Epoch 2: T1 lock B, T2 lock C, T3 lock A -> detect deadlock
Epoch 3: T1 unlock A, T2 unlock B, T3 unlock C
Epoch 4: T1 lock B
Epoch 5: T1 unlock B
Epoch 6: T2 lock C
Epoch 7: T2 unlock C
Epoch 8: T3 lock A
Epoch 8: T3 unlock A
```

Comments:

Question 6

Exercise 6 : (*Q3.ab from s2.pdf*)
Timestamp ordering allows transactions to operate concurrently based on a serialisation selected as transactions start. Each transaction is issued a timestamp (sequence or ticket number) that is checked against object timestamps to detect conflicts, and to taint objects written or read to trigger detection of conflicts with other transactions that will touch the same object later.

(a) Timestamp ordering is conservative, in that it is committed to a particular serialisation of a transaction attempt at the moment that the timestamp is selected, causing it to potentially reject other valid serialisations. Give an example of two transactions, timestamp assignments, and an interleaved schedule in which a valid serialisation is rejected by TSO, leading to an unnecessary abort.

```
T1R(A): t = 1
T1W(A): t = 2
T2W(B): t = 3
T2W(A): t = 4

A possible non-serial but conflict-serialisable schedule:
T2W(A), T1R(A), T2W(B), T1W(A)

But with TSO, comparing the timestamp:
T2W(A) > tA -> tA = 4
T1R(A) < tA -> aborted schedule which is unnecessary
...
```

(b) TSO aborts transactions as conflicting operations are detected. Why could TSO could suffer from livelock under heavy contention?

- Under heavy contention, TSO will repeatedly have transactions aborting and retrying as it can miss serialisable schedules.
- Thus, if there is a livelock, there are no conflict operations. The states of the transactions are continuously changing, but none are making progress. TSO cannot detect such livelock as the operations do not have conflict timestamps.

💡 Comments:

Question 7

Exercise 7 : (*Q3.cdef from s2.pdf*)
Optimistic concurrency control likewise allows transactions to operate concurrently; unlike TSO, it 'searches' for a valid serialisation on transaction commit, rather than at transaction start. This is argued to give OCC greater flexibility in avoiding unnecessary aborts seen in TSO, which might exclude valid serialisations.

(a) Give an example of two transations and a schedule that would be accepted by OCC, but rejected by TSO.

```
A non-serial but conflict-serialisable schedule:
T1R(A), T2W(A), T2W(B), T1W(A)

With TSO:
  T1R(A): t = 1
  T1W(A): t = 2
  T2W(B): t = 3
  T2W(A): t = 4

  Comparing the timestamp:
  T1R(A) > tA : tA = 1
  T2W(A) > tA : tA = 4
  T2W(B) > tB : tB = 3
  T1W(A) < tA : aborted schedule

With OCC:
  Exchanging T2W(B) and T1R(A), we could have a serial schedule, hence accepted
```

(b) OCC executes transactions against local copies of data ('shadows') rather than globally visible original copies, which avoids the need to explicitly handle cascading aborts. However, copying all objects at the start of the transaction is problematic if the set of objects to be operated on is determined as part of the transaction itself. Why does on-demand copying of objects complicate transaction validation?

- On-demand copying of objects implies we don't know the exact objects at the start of the transaction and needs to devise all serialisable schedules with the set of objects at run-time.
- In the verification phase, despite serializability validation and read validation, we also have optimality matching, ensuring choosing a serialisation that commits as many as possible. Since each commit has a different set of objects acquired, it isn't easy to implement a heuristic strategy based on recent history.

(c) OCC aborts transactions as conflicting commits are detected. Why, informally, might we argue that this conflict behaviour is more resistant to whole-system livelock than TSO?

- OCC performs on data copies, where uncommitted transactions can be considered inconsistent states; livelock could occur because two transactions could repeatedly abort and restart. But in OCC, it can prevent livelock by adding a private write buffer for each transaction so that other transactions can only see committed transactions and their own writes.
- However, with TSO, we use timestamps to order operations, and these operations can be seen by all transactions, leading to a higher possibility of livelock.

(d) How could OCC's validation model lead to starvation? Describe a scenario in which a set of transactions and a serialisation cause starvation for a transaction.

- If there are sequences of short transactions with conflicting operations in a long-term transaction, OCC would repeatedly abort and restart the long-term transaction, leading to starvation. To avoid starvation, we need to temporarily block the conflicting transactions, letting the long-term transactions finish their execution.

(e) Why is OCC good for online booking on the web?

- The stateless nature of the web makes locking hard to implement, so OCC, which implements without locking, is preferred.
- For online booking, conflicts are rare because one customer usually only want to commit one booking at a time.
- Web data contention is low, OCC can find more serialisable schedules, no cascading aborts and no deadlock, so much easier to implement and roll back.
- The update is operated on the data copy rather than the original data, so OCC provides a much reliable system.

Comments:

Extra optional interesting content

💡 SQLite is probably one of the most used software libraries in the world. You might find it interesting to check out its documentation on atomicity and isolation. There's also an interesting podcast with its inventor.