* 1. Atomicity is achieved as long as rollback is implemented. Consistency is achieved by allowing invariants to be restored before committing changes. Isolation is not inherent in the system but can be achieved using strict 2PL which holds locks for longer. Durability is not inherent in the protocol but instead depends on the implementation of the “commit” function.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Instruction 1 | Instruction 2 | Instruction 3 | Instruction 4 | Serial? | Conflict Serialisable? |
| a = A.getBalance(); | A.credit(INTEREST\*a); | A.debit(100); | B.credit(100); | Yes | Yes |
| a = A.getBalance(); | A.debit(100); | A.credit(INTEREST\*a); | B.credit(100); | No | No |
| a = A.getBalance(); | A.debit(100); | B.credit(100); | A.credit(INTEREST\*a); | No | No |
| A.debit(100); | a = A.getBalance(); | A.credit(INTEREST\*a); | B.credit(100); | No | Yes |
| A.debit(100); | a = A.getBalance(); | B.credit(100); | A.credit(INTEREST\*a); | No | Yes |
| A.debit(100); | B.credit(100); | a = A.getBalance(); | A.credit(INTEREST\*a); | Yes | Yes |

* 1. With OCC the data is written to a copy of the data, so it is very easy to simply discard the copy if something goes wrong, without the risk of any of the original data having been modified. This means that rollback is built into the system, as opposed to 2PL in which rollback is an additional protocol to implement.
  2. If the operations were run in the following order:
* A.debit(100);
* a = A.getBalance();
* A.credit(INTEREST\*a);
* B.credit(100);

then the “a” variable from T1 would be a dirty read, and so T1 would have to abort when T2 aborts. Strict 2PL would have prevented this by allowing T2 to hold on to the A write lock for longer, stopping T1 from making the dirty read.

* 1. This might occur if the data is half-way through being written to disk and then encounters a failure such as a power outage. In this case only half of the changes would be written to disk and the operation would be marked as unfinished.
     1. A bigger maximum log size means that more of the finite persistent storage space is used up by the log. This means that much of the data is redundant/outdated
     2. The log is append-only so varying the size of the log shouldn’t affect transaction throughput.
     3. Having a larger log means that the chain of transactions to undo or redo might be longer, but again hopefully the chain is short enough that it makes up a small proportion of the log itself so the recovery time should not be affected
     4. A log entry might only be partially written, making it impossible to undo or redo the transaction
     5. Log entries might be written out of order, and so non-commutative transactions might be recorded differently in the log than on the disk. This would again cause issues recovering after a crash
* T1.L1
* T2.L1
* T3.L1
* -- DEADLOCK --  
  1. The same schedule will be selected and will repeat every time a deadlock is reached.
  2. Each time a deadlock occurs, systematically reorder the transactions such that if necessary, they will try executing in every possible permutation.

transaction T1 { *// V(T1) = 0*

x = A.read(); *// V(A) := 0*

B.write(x); *// V(T1) < V(B) so abort*

}

transaction T2 { *// V(T2) = 1*

x = B.read(); *// V(B) := 1*

A.write(x); *// V(T2) >= V(A) so V(A) := 1*

}

* 1. TSO can often reject serializable schedules because it decides on an order a priori. This means that a set of heavily contentious transactions might be repeated many times. This is starvation, which is a form of livelock.

transaction T1 { *// V(T1) = 0*

x = A.read(); *// V(A) := 0*

A.write(10); *// V(T1) < V(A) so abort*

}

transaction T2 { *// V(T2) = 1*

y = A.read(); *// V(B) := 1*

}

Schedule:

* x = A.read();
* y = A.read();
* A.write(10);
  1. It complicated transaction validation because it is unclear at the start of the transaction which data needs to be copied, and by the time it is known, some of that data might have been modified by another transaction. This means that if a transaction needs two objects, the first might be copied before the second transaction has committed, and the second might be copied afterwards. This can violate some of the data invariants.
  2. OCC aborts at commit-time, whereas TSO chooses an ordering when the transactions start. This means that TSO will abort more often (sometimes even if not necessary) and so it is more likely to create livelock.
  3. Starvation might occur when set of transactions is repeatedly retried. This is likely when the mixture of transactions is highly conflicting.
  4. OCC is good for online booking because it is unlikely that there will be much conflict – the scenario where two people try to book the last available spot is a rare occurrence – and so the optimism is justified. This means that the risk of starvation is outweighed by the increased likelihood of finding a serializable schedule.