# Assignment 1 – Solution

The main bookkeeping of transaction state is done in `Cache(e).tId`. It is the negative transaction id of the transaction that updated the block, if the block has not yet been written to disk. It is made positive after being written to disk. And it is set to 0 immediately before the transaction commits or aborts.

a. Before looking at the ACID properties, consider the following scenarios:

1. Suppose transaction T1 updates cache blocks x and y and then commits. During the commit procedure, the diskWrite to x succeeds, but the diskWrite to y fails. Therefore, the transaction aborts. However, this means that x will not be restored correctly, because Commit previously set `Cache(x).oldBlock = Cache(x).newBlock`. The fix is to move this assignment into the next for-loop, at which point Commit can no longer fail.

2. Suppose transaction T1 updates `Cache(x).newBlock` and before calling `write(x, T1)` it aborts. Since `x` does not satisfy the condition of the first for-loop, "`Cache(x).newBlock = Cache(x).oldBlock`" does not execute. So `x` remains in cache but it has the wrong value for `Cache(x).newBlock`, which will be read by the next transaction that reads `x`. A fix for this would be to add the following for-loop to `abort`:

```
for (all cache entries e, where Cache(e).tId == tId)
   {Cache(e).newBlock = Cache(e).oldBlock
}
```

In view of these problems, let's look at the ACID properties one by one:

- Atomicity: The implementation is not atomic, due to scenarios 1 and 2 above. In these cases, T1 aborts but does have an effect, due to the erroneous value of Cache(x).oldBlock.
- Isolation: Transactions execute serially. So in the absence of aborts due to scenarios 1 and 2, the implementation is Isolated.
- Consistency: This is up to the transactions. The system has no effect, one way or the other.
- Durability: The implementation is durable. If a transaction commits, then all of its updates are on disk.

b. If the operating system fails while a transaction is executing Commit, then only some of its updates may have been written to disk, in which case the result is not atomic. The same problem can happen if the failure happens during an execution of Abort. There are no easy solutions to this. One solution, called shadow paging, will be described in class. Later in the course, we'll discuss another approach based on logging.

c. Given the issues in (a), this scenario doesn't make things much worse. So to make the problem more interesting, let's assume the fixes to part (a) are implemented. During Commit, if DiskWrite(b, a) succeeds (i.e. returns 0) and corrupts block b on disk, then the execution is not atomic. One way to fix this is to read back each block after it is written and diff it with the intended content, which is obviously a high price to pay for atomicity. Another way is to buy a disk that doesn't have this bad behavior ☺.

d. In the variation, suppose DiskWrite(b, a) fails (i.e. returns -1) and corrupts block b on disk. Then two lines later `abort` will be called, which will restore the previous value of the page on disk. So if the fixes to part (a) are implemented, the execution is still atomic. Isolation and durability are not affected, so the system is still durable and isolated.

e. Again, this scenario doesn't make things any worse than (a). So let's assume the fixes are implemented. In effect, this scenario deletes the first statement in Start(), thereby allowing a second transaction to start while another one is active. Let's look at the ACID properties one by one:

- Atomicity: It's hard to distinguish atomicity from isolation in this case. That said, if two transactions are running concurrently, one of them could overwrite a cache block that was previously updated by the other, causing its update to be lost and hence not be atomic. A variation is that the second transaction aborts. Since

its update didn't affect Cache(e).oldBlock, when it aborts it restores the first transaction's before-image of the block, instead of the first transaction's after-image. Another example: if $T_1$ updates a block, $T_2$ reads it, and then $T_1$ aborts, then $T_1$'s execution is not atomic. $T_2$'s read operation causes "Cache(e).tId = tId" to execute, which means that $T_1$'s update won't be undone.

- Isolation: Both transactions can update the same two blocks, say x and y, in different orders: $W_1(x)$ $W_2(y)$ $W_1(y)$ $W_2(x)$ $C_1$ $C_2$. Here if both transactions commit, the database state contains writes from each transaction: y from $T_1$, and x from $T_2$. This cannot occur in a serial execution. Hence the execution is not serializable.
- Consistency: Same as scenario (a), though you could argue it's even worse since the execution isn't isolated.
- Durability: This doesn't affect durability, although you could argue that the violations of atomicity and isolation mean the result isn't durable either.

One simple solution is the code in the assignment, where Start() returns an error when there's an active transaction. If you want transactions to run concurrently, then some form of per-block synchronization is needed, such as two-phase locking. We'll discuss this soon in class.