Exercises for Hands-on 7: Write Ahead Log System

Make sure to fill in questions in the space provided underneath the question text.

Using wal-sys:

Start *wal-sys* with a reset:

```
athena% ./wal-sys.py -reset
```

and run the following commands (sequence 1):

```
begin 1
create_account 1 studentA 1000
commit 1
end 1
begin 2
create_account 2 studentB 2000
begin 3
create_account 3 studentC 3000
credit_account 3 studentC 100
debit_account 3 studentA 100
commit 3
show_state
crash
```

Wal-sys should print out the contents of the DB and LOG files, and then exit.

Use a text editor to examine the DB and LOG files and answer the following questions (do not run *wal-sys* again until you have answered these questions):

Question 1: *Wal-sys* displays the current state of the database contents after you type show_state. Why doesn't the database show *studentB*?

---

1) Because studentB's account is neither committed or ended. The `end` command is what triggers the forced_write_to_disk, which writes the db/persistent storage. The sequence above crashes before the task (2) of writing studentB's account info. The only student in the database is studentA, since only task 1 had ended prior to the crash.

Question 2: When the database recovers, which accounts should be active, and what values should they contain?

Question 3: Can you explain why the DB file does not contain a record for *studentC* and contains the pre-debit balance for *studentA*?

---

2) When the database recovers, studentA and studentC should be active since they both have actions which were committed. Though, studentB's account information will be lost, as no changes are committed before the crash. StudentA should have $900 and studentC should have 3100, as the credit and debit commands were committed.

3) Because those actions take place in task (3). Task (3) executes the account creation of studentC and the debit of studentA, then commits these changes. But, since these actions are only committed and not ended, this means that they have only been logged persistently. The database only writes these changes after the `end` keyword, but a crash happened before this.

Recovering the database

When you run wal-sys without the **-reset** option it recovers the database "DB" using the "LOG" file. To recover the database and then look at the results, type:

```
athena% ./wal-sys.py
> show_state
> crash
```

Question 4: What do you expect the state of DB to be after *wal-sys* recovers after this new crash? Why?

Question 5: Run *wal-sys* again to recover the database. Examine the DB file. Does the state of the database match your expectations? Why or why not?

Question 6: During recovery, *wal-sys* reports the *action_id*s of those recoverable actions that are "Losers", "Winners", and "Done". What is the meaning of these categories?

---

4) After this new (2nd) crash, I would expect the database to be as it is in the show_state call. This is different than in (1), as committed actions were ended after they were recovered. There are no other modifications to the DB.
On-disk DB contents:
Account: studentC Value: 3100
Account: studentA Value: 900

5) Yes, the state of the DB matches my expectation. No additional actions happened in the snippet above, so recovering after that crash shouldn't change the DB.

6)

- **Done** seems to be tasks (such as task 1) which commits and finishes before the crash, and all of its data is already moved to persistent storage thus requires no recovery action
- **Winner** seems to mean tasks which were not finished, but were committed and have adequate logs to reconstruct the state at its commit point. During recovery, the state at the commit point is restored, and the task is then ended so these changes can be reflected in the disk. The textbook says its any all-or-nothing action that recorded an OUTCOME in the log.
- **Loser** seems to be tasks which were initiated, but never made it to their commit point. These actions are basically lost, and are not re-executed. The textbook says that they are all-or-nothing actions that were still in progress at the time of the crash

Checkpoints

Start wal-sys with a reset:

```
        athena% ./wal-sys.py -reset
```

and run the following commands (sequence 2):

```
        begin 1
        create_account 1 studentA 1000
        commit 1
        end 1
        begin 2
        create_account 2 studentB 2000
        checkpoint
        begin 3
        create_account 3 studentC 3000
        credit_account 3 studentC 100
        debit_account 2 studentB 100
        commit 3
        show_state
        crash
```

Note: *the remainder of this assignment is only concerned with sequence 2. We will ask you to crash and recover the system a few times, but you should not run the sequence commands again. (Also note that in sequence 2, the command* ***debit_account 2 studentB 100*** *refers to action_id 2, not action_id 3! This is not a typo).*

Question 7: Why are the results of recoverable action 2's **create_account 2 studentB 2000** command not installed in "DB" by the **checkpoint** command on the following line?

---

7) The `checkpoint` keyword does not trigger a write to the DB. The checkpoint serves as a summary, and supplies information about every line up to that point. This allows for better reasoning about the environment during recovery. Only the `end` keyword triggers a write to the DB.

Examine the LOG output file. In particular, inspect the CHECKPOINT entry. Also, count the number of entries in the LOG file. Run wal-sys again to recover the database.

Question 8: How many lines were rolled back? What is the advantage of using checkpoints?

---

8) The log was rolled back 6 lines (to the checkpoint) in this case, compared to 11 lines in the first sequence (to the beginning). Checkpoints have advantages and disadvantages based on the needs of the DB. They add additional complexity to recovery, but adding more types of information to the log, but this extra complexity usually increases recovery performance in practice. A checkpoint basically summarizes all losers up to that point, so that the log does not need to be rolled all the way back. Specifically, if a checkpoint is encountered early in the backtrack through a long long, and the checkpoint claims that there are no losers prior to that point, then recovery can stop without reading the entire log.

Note down the *action_id*s of "Winners", "Losers", and "Done". Use the **show_state** command to look at the recovered database and verify that the database recovered correctly. Crash the system, and then run *wal-sys* again to recover the database a second time.

Question 9: Does the second run of the recovery procedure (for sequence 2) restore "DB" to the same state as the first run? What is this property called?

Question 10: Compare the *action_id*s of "Winners", "Losers", and "Done" from the second recovery with those from the first. The lists are different. How does the recovery procedure guarantee the property from Question 9 even though the recovery procedure can change? (Hint: Examine the "LOG" file).

Question 11: How long did this hands-on take you?

---

9) Yes, the second run will recover the DB to the same state as the first. In fact, it will do this after any number of crash/recover cycles, or even if the program crashes during recovery. This property is known as being *idempotent*. Specifically, a system is idempotent if it can recover from any number of crash-restart cycles without compromising the correctness of the ultimate result.

10) **First Recovery:**
Winners: id: 3 Losers: id: 2 Done: id: 1
Second Recovery:
Winners: Losers: id: 2 Done: id: 1 id: 3

We see above that the 2 sets of ids have different labels. Specifically, the winner (3) has been moved to Done. This is because that during the recovery process, the system logs END records for all winners because winners are known to have recorded an OUTCOME in the log. The recovery procedure did not *change* as in both recoveries any winners would be ended, and by the end of the first recovery action (3) had an END record. By the end of the 2nd recovery action (3) still had an END record, thus the state after recovery was the same and the the process was idempotent.

11) 4 hours.