The assignment is to be turned in before Midnight (by 11:59pm) on February 27. You should turn in the written parts of the solutions as a single pdf file through the TEACH website. The written solutions should be produced using editing software programs, such as LaTeX or Word, otherwise they will not be graded. You should turn in the source code and executable files for the programming assignment through the TEACH website.

1: Recovery and ARIES (4 points)

In this problem, you need to simulate the actions taken by ARIES algorithm. Consider the following log records and buffer actions:

time	LSN	Log	Buffer actions
0	00	update: T1 updates P7	P7 brought in to the buffer
1	10	update: T0 updates P9	P9 brought into the buffer; P9 flushed to disk
2	20	update: T1 updates P8	P8 brought into the buffer; P8 flushed to disk
3	30	begin_checkpoint	
4	40	end_checkpoint	
5	50	update: T1 updates P9	P9 brought into the buffer
6	60	update: T2 updates P6	P6 brought into the buffer
7	70	update: T1 updates P5	P5 brought into the buffer
8	80	update: T1 updates P7	P6 flushed to disk
9		CRASH RESTART	

(a) For the actions listed above, show Transaction Table (XT) and Dirty Page Table (DPT) after each action. Assume that DPT holds pageID and recLSN, and XT contains transID and lastLSN.

Solution:

Let XT denote xact table and DPT denote dirty page table.

After time 0:

XT

transaction	lastLSN	status
T1	00	active

DPT

page	recLSN
P7	00

After time 1:

XT

transaction	lastLSN	status
T1	00	active
T0	10	active

DPT

page	recLSN
P7	00

After time 2:

XT

transaction	lastLSN	status
T1	20	active
T0	10	active

DPT

page	recLSN
P7	00

After time 3: same as above.

After time 4: same as above.

After time 5:

XT

transaction	lastLSN	status
T1	50	active
T0	10	active

DPT

page	recLSN
P7	00
P9	50

After time 6:

XT

transaction	lastLSN	status
T1	50	active
T0	10	active
T2	60	active

DPT

page	recLSN
P7	00
P9	50
P6	60

After time 7:

XT

transaction	lastLSN	status
T1	70	active
T0	10	active
T2	60	active

DPT

page	recLSN
P7	00
P9	50
P6	60
P5	70

After time 8:

XT

transaction	lastLSN	status
T1	80	active
T0	10	active
T2	60	active

DPT

page	recLSN
P7	00
P9	50
P5	70

Note: P6 is written out with pageLSN = 80.

(b) Simulate Analysis phase to reconstruct XT and DPT after crash. Identify the point where the Analysis phase starts scanning log records and show XT and DPT after each action.

Solution:

Analysis phase begins by examining the most recent checkpoint and initializing the XP and DPT at the time of the checkpoint, i.e., after time 2 in solution to part (a).

After time 5:

XT

transaction	lastLSN	status
T1	50	active
T0	10	active

DPT

page	recLSN
P7	00
P9	50

After time 6:

XT

transaction	lastLSN	status
T1	50	active
T0	10	active
T2	60	active

DPT

page	recLSN
P7	00
P9	50
P6	60

After time 7:

XT

transaction	lastLSN	status
T1	70	active
T0	10	active
T2	60	active

DPT

page	recLSN
P7	00
P9	50
P6	60
P5	70

After time 8:

h!

LSN	Redone?	Why Not?
00	Yes	
10	No	affected page in DPT but recLSN is greater than 10
20	No	affected page not in DPT
30	No	checkpoint
40	No	checkpoint
50	Yes	
60	No	pageLSN 80 is greater than 60
70	Yes	
80	Yes	

Table 1: Redo operations

XT

transaction	lastLSN	status
T1	80	active
T0	10	active
T2	60	active

DPT

page	recLSN
P7	00
P9	50
P6	60
P5	70

(c) Simulate Redo phase: first identify where the Redo phase starts scanning the log records. Then, for each action identify whether it needs to be redone or not.

Solution:

Redo starts from the smallest recLSN in DPT at the end of Analysis, i.e., LSN=00. Table 1 shows whether each action needs to be redone and the reason.

(d) Simulate Undo phase: identify all actions that need to be undone. In what order will they be undone?

Solution:

As no transaction committed, all actions will be undone in the decreasing order of LSN. That is UNDO 80, 70, 60, 50, 20, 10, and 00.

2: Recovery (6 points)

Consider the following relations:

```
Dept (did (integer), dname (string), budget (double), managerid (integer))
Emp (eid (integer), ename (string), age (integer), salary (double))
```

Fields of types *integer*, *double*, and *string* occupy 4, 8, and 40 bytes, respectively. Each block can fit at most one tuple of an input relation. There are at most 22 blocks available to the join algorithm in the main memory. Assume that relations *Dept* and *Emp* are sorted according to attributes *managerid* and *eid*, respectively.

(a) Explain a join algorithm that efficiently computes $Dept \bowtie_{Dept.managerid=Emp.eid} Emp$ by merging these relations. We do *not* want to repeat the join from the beginning if a crash happens during the join computation. Instead, your algorithm must avoid recomputing the joint tuples that have been computed before the crash and continue the join computation after the database system restarts. The final output of your algorithm must be exactly the same as the result of the join as if no crash has happened during the join computation. You should explain the steps that your algorithm will follow during the normal execution of join and the techniques that it uses after a restart.

Solution:
In order to compute this join and enable it to be able to easily recover We should make the join as simple as possible. As such My algorithm will only read in one tuple at a time from each relation. We can do this because we are guaranteed that we are given the files in sorted order. This allows to keep the state of the join much more concise. Below I will explain how the join part of the algorithm works and then how the the logging and recovery part of the algorithm works.
The join algorithm works by reading in one tuple at a time from each relation. We then compare the two tuples. If they are equal on the join attribute then we merge them and out put the result keeping the employee tuple and reading in a department tuple. If the employee tuple is larger than the department tuple then a new department tuple is read in. If the department tuple is larger than the employee tuple then we load a new employee tuple. We do this until we run out of either department tuples or employee tuples.
In order to make sure that we can pick up where we left off we log the state of the join through out each step of the join. Directly following every read we write to the log file the offset of the begging of the tuple we just read in. After every write we record the offset of the end of the tuple that was just written. We make these log writes after the are completed to ensure that if the write or read is interrupted that we don't have partial reads or writes that have been recorded in the log. Logging after an operation ensures that ever operation we have in the log has been completed. It does run the risk of repeating some operations that have already been completed but only the most recent. However, This cost is going to be far less than going back and checking the last operation we have in the log is in some sort of dirty state.
In order to recover after we have crashed is a very simple process. First we read in the last entry to the log. This entry will contain all the offsets for the employee file, the department file and the output file. Once we have these offsets we can open the files and fseek to the offsets recorded in the log. With the files now set up to point exactly where they were when the system crashed we can restart the join algorithm using the existing join code we already wrote. As we continue the join we also append to the log making sure not to delete any of the log file until the join has been fully completed. The final step to ensuring that the recovery happens correctly is delete the log File once the join has been completed. We do this so that our program can tell when the join has been completed with out error. On start up the first thing we will do is check to see if there is a log file. If there is a log file that is non empty then our program was interrupted. If there is no log file then we know that the program exited normally so we need not recover.

Notes About Program:
To crash this program you may set the constant TEST_CRASH and STABILITY. TEST_CRASH Is a bool that will tell the program that you would like the random crash behavior on. Currently it is set to false. STABILITY will set how often a crash will happen the lower the stability the higher the chance of a crash. It is also important to note that my program will not restart itself. After a crash you will need to restart the program by typing main.out again.

(b) Implement your proposed algorithm in C++.

Requirements:

- Each input relation is stored in a separate CSV file, i.e., each tuple is in a separate line and fields of each record are separated by commas.
- The result of the join must be stored in a new CSV file. The files that store relations Dept and Emp are Dept.csv and Emp.csv, respectively.
- Your program must assume that the input files are in the current working directory, i.e., the one from which your program is running.
- The program must store the result in a new CSV file with the name join.csv in the current working directory.
- Your program must run on Linux. Each student has an account on *voltdb1.eecs.oregonstate.edu* server, which is a Linux machine. You may use this machine to test your program if you do not have access to any other Linux machine. You can use the following *bash* command to connect to *voltdb1*:

  ```
  > ssh your_onid_username@voltdb1.eecs.oregonstate.edu
  ```

  Then it asks for your ONID password and probably one another question. You can only access this server on campus.
- You can use following commands to compile and run C++ code:

  ```
  > g++ main.cpp -o main.out
  > main.out
  ```