Problem 1: File System (25’)

Xiaohong wrote an inode-based file system called NaiveFS. She formats one of her disk partitions as NaiveFS and mounts it to "/". NaiveFS matches the inode-based file system we learned in CSE classes, with the following parameters:

- The size of each inode is 256 Bytes.
- The size of each data block is 512 Bytes.
- An inode contains 56 pointers, with the first 54 pointers directly pointing to the data blocks, the next 1 pointer pointing to an indirection block and the last 1 pointer pointing to a double-indirect block.
- The size of each data block pointer is 4 bytes.

The detailed structure of an inode in NaiveFS is shown in the following table:

Name	**Size (Bytes)**	**Description**
inode ID	8	inode identifier
Size	8	File size
Type	1	File Type
Padding	3	Alignment padding
RefCount	4	Reference count of this inode
Atime	4	Last access time of this inode
Mtime	4	Last data modification time of this inode
Block Pointers	4*56	Block pointers of this inode

Table-1 inode design in NaiveFS

1. Xiaohong has a file "/mnt/hdd/cse-notes.txt", where "/mnt/hdd" is mounted as an ext4 file system. She hopes to create a hard link from "/cse-notes.txt" to "/mnt/hdd/cse-notes.txt". Can she succeed? Why? (4’)

2. Xiaohong creates a symbolic link from "/lab-symbolic.tar.gz" to "/lab.tar.gz". Please describe the read and write sequence of inode and data during `open("/lab-symbolic.tar.gz")`. E.g., read XXX inode, write XXX inode, read XXX data, write XXX data. (Suppose the file system is mounted using the options to record atime for files, but not to record atime for directories) (4’)

3. Suppose the file write order in NaiveFS is: update size → allocate new blocks → write new data. Please list at least **two** inconsistent cases after crash. (4’)

4. Alice tries to copy a video (~20MB) to NaiveFS, but she found that NaiveFS does not support files of such size currently. Can you provide at

least **two** solutions to make NaiveFS support storing the video? You are free to change any parameter of NaiveFS. (4')

5. For a file with a content length of 224,256 Bytes, please calculate the metadata size caused by NaiveFS. You should consider the space occupied by the inode and all the indirect blocks. (4')

6. Reducing file system metadata size is important for performance. The smaller the metadata is, the more possible it is cached in the DRAM and does not need to load from the slow disk. For large files, many indirection blocks would be used as metadata. Can you figure out a way to optimize metadata size for large files? (Hint: if the file can be stored sequentially on the disk...) (5')

Problem 2: MapReduce and Graph (24')

Xiaohong hopes to use MapReduce (MR) to run a Social Arithmetic algorithm for a very large graph, which is stored as an adjacent list (vertex, weight, [friends]).

Below is a sequential version of the Social Arithmetic program.

```
Compute(v):
1   foreach n in v.friends:
2           like += n.weight / n.edges;
```

1. Please write the Map and Reduce function, respectively, in order to exploit parallelism over MR. You may also use pseudocode. (8')

2. MR workers communicate with the master via Remote Procedure Calls (RPCs). Should RPCs in MR guarantee idempotence? Why? (4')

3. Failures are common in datacenters. How can MR detect and tolerate worker failures? (4')

4. Xiaohong uses GFS for MR. A GFS typically consists of a master and multiple chunkservers. How does GFS handles master and chunkserver failures? (4')

5. Bob suggests Xiaohong to use a graph-friendly framework such as Pregel. What are Pregel's advantages for graph processing? (at least two) (4')

Problem 3: Paxos (25')

Here is the pseudocode for **Paxos**, we assume all the servers are both acceptors and proposers.

```
                              States
n0: the proposal number of the proposer
v0: the chosen value of the proposer
np: highest prepare seen
na, va: highest accept seen
                            Proposer
propose(v):
1  choose n0 > np
2  send prepare(n0) to all servers including itself
3  if prepare_ok(na; va) from majority:
4     v0 = va with highest na; choose own v otherwise
5     send accept(n0; v0) to all
6     if accept_ok(n0) from majority:
7          send decided(v0) to all
                            Acceptor
acceptor's prepare(n) handler:
1  if n > np
2     np = n
3     reply prepare_ok(na; va)
acceptor's accept(n; v) handler:
1  if n >= np
2     np = n
3     na = n
4     va = v
5     reply accept_ok(n)
```

1. Could Paxos successfuly decide a value when 6 servers fails in a cluster of 9 servers? How many server failures could be tolerated **at most** by Paxos in a cluster of 9 servers? (5')

2. The CAP theorem states that it is impossible for a distributed system to simultaneously guarantee **C**onsistency, **A**vailability, and **P**artition tolerance.

   a) What should the proposer do if there is no majority replying accept_ok in propose(v) line 5? (2')

   b) Which property of CAP will be sacrificed using your solution in a)? Please give an example to explain your answer. (4')

3. Which states (*n0, v0, np, na, va*) **must** be persisted for consistency? Please describe how to restore the states above that do **not** need to be persisted? (5')

4. A server lost its persisted state on the disk, and it tries to restore the states by copying from another server. Is it correct? If so, why? If not, please give an example to explain your answer. (5')

5. A learner wants to get a decided value by asking the value of *va* from a randomly picked server. Is it correct? If so, why? If not, how to get the decided value? (5')

Problem 4: Transaction Concurrency Control (25')

Xiaohong is developing a ticketing system for a railway company, the main business of which includes ticket purchasing and refunding. When purchasing a ticket, a user would select one seat and the amount of the remaining ticket will be declined. The refunding is the opposite.

Assuming xiaohong has developed one basic function to handle the requests of ticket purchase (T_1). The read of remaining ticket amount (A) in transaction T_i is annoted as $R_i(A)$, and $W_i(A)$ as write event.

1. Here's one implementation of *ticket_purchase*. Please point out the problems when different threads run them in parallel. After that, give some solutions you've learned in class to manage the transaction concurrency control. (4')

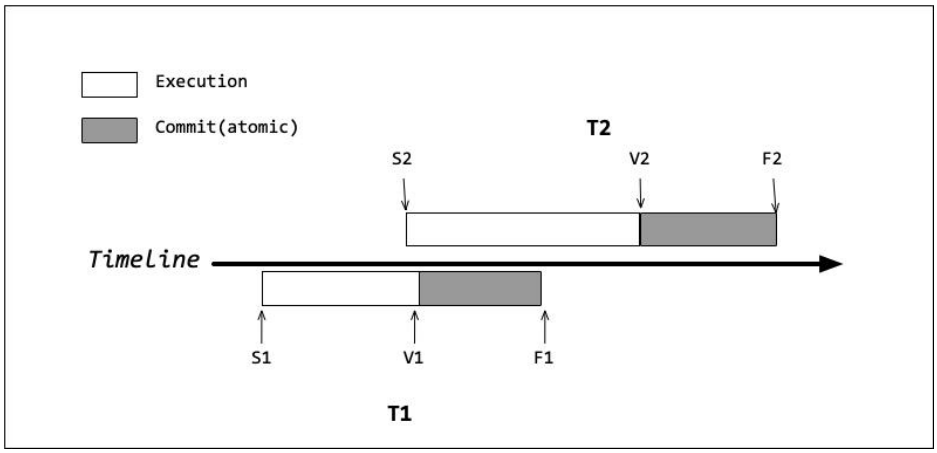
```
void ticket_purchase() { // purchase one ticket
    tmp = read(ticket_amount)
    if (tmp == 0) {  return Fail }
    write(ticket_amount, tmp - 1)
}
```

2. Now Xiaohong uses transactions to handle the ticketing system, and adopts conflict graph to show the data dependency of the transactions, assisting to check out whether one schedule is conflict serializable. We denote *ticket_num* as A and if we have two conflict transaction operations $W_1(A)W_2(A)$, one directed edge would start from T_1 to T_2 in the conflict graph.
   Now please draw the conflict graph of $R_1(A)R_2(A)R_3(B)W_2(A)R_1(B)W_3(B)W_1(A)$, and explain why this schedule is not conflict serializable (4')

3. Xiaohong wants to adopt 2PL to control the concurrency in the system, while 2PL may have introduce deadlock. Please give two solutions to avoid deadlock and explain why these solutions can help to avoid deadlock (4').

4. There are 3 transactions T_1, T_2, T_3 executing under a OCC protocol we've learned in class. We assume each transaction will perform three steps in order:

start the transaction(***S***), start validation (***V***) , finish commit (***F***). That is, transaction T_i must perform $S_i; V_i; F_i$ in order.
For example, the below figure shows the process, and the events sequence is $S_1; S_2; V_1; F_1; V_2; F_2$. Note that the commit opearation is atomic (i.e., execute in a critical section). If one transaction meets the requirement of OCC protocol, this transaction could commit (i.e., validate successfully).



Suppose three transactions have the following read/write sets:

	Read set	Write set
T_1	$\{B, G\}$	$\{C, E, G\}$
T_2	$\{E\}$	$\{A, B\}$
T_3	$\{A, C, D\}$	$\{C\}$

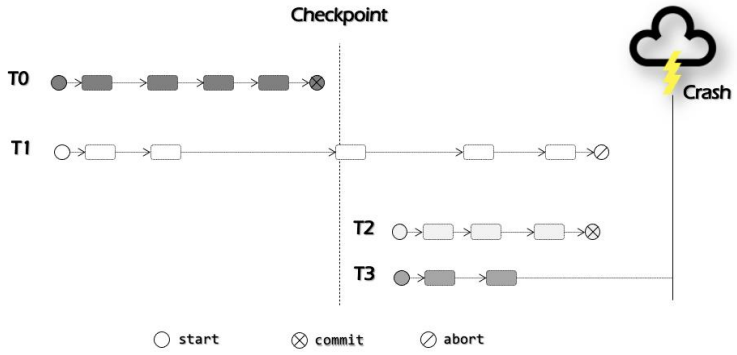
Now consider the following sequence of events $S_1; S_2; V_2; V_1; S_3; F_2; F_1; V_3; F_3$ .Please answer which transactions commit successfully and which do not.
(Note that the changes is invisible until the transactions come to ***F*** stage.)

a) Does T_1 commit successfully ? **If not, point out the conflict transaction.** (3')

b) Does T_2 commit successfully ? **If not, point out the conflict transaction.** (3')

c) Does T_3 commit successfully ? **If not, point out the conflict transaction.** (3')

5. Now Xiaohong is going to log all of the actions in transactions. Assume one crash occurs, and Xiaohong notices the event sequence is shown as the figure below:



a) With the help of undo/redo log, the transactions could recover from the crash by conducting undo/redo operations. Please explain why we need checkpoint in logging, and briefly describe how to add one checkpoint while logging. (4')

b) According to the figure, T_0 could **do nothing** to recover from the crash. Now please point out what operations other three transaction should do (e.g., T1: do nothing). (3')