上海交通大学试卷（期中）

（ 2023 至 2024 学年 第 1 学期）

班级号________________ 学号__________________姓名____________

课程名称　　　　计算机系统工程　　　　　　　　成绩____________

Problem 1: (Distributed) File System (30’)

1. NaiveFS is an inode-based filesystem. It is mounted to "/". NaiveFS has the following parameters:

- The sizes of inode and data block are 4KB.
- An inode contains 256 block pointers, with the first 255 pointers directly pointing to the data blocks, and the last block pointer pointing to an indirect block. The indirect block only contains block pointers pointing to the data blocks.
- The size of each data block pointer is 8 bytes.

a. Please calculate the max file size this file system can support. (Please also show how you calculate the result, list the formula) (5')

(255 + (4KB / 8B)) * 4KB

b. Suppose there are two directories "/A" and /B", and one regular file "a" under directory "/A". Please **describe** the process of creating a **hard link** from "/B/a-hard-link" to "/A/a". Please **describe** explain the process of creating a **symbolic link** fron "/B/a-symbolic-link" to "/A/a". (8’) Note that you only need to use the following procedures to describe the process:

- lookup(inode_id_t parent, string name) -> inode_id_t
- allocate_inode(FileType type) -> inode_id_t (type can be REGULAR/SYMLINK)
- write_file(inode_id_t id, string contents)
- add_dentry(inode_id_t parent, string name, inode_id_t child)

We assume that:

- The inode id of "/" is 1.

- The directory information is stored as a "name0:inode0/name1:inode1/ ..." string in the file blocks.

Hard link:

A_node_id = lookup (1, “A”)

... // Your process

hard link:

A_inode_id = lookup(1, "A")

B_inode_id = lookup(1, "B")

a_inode_id = lookup(A_inode_id, "a")

add_dentry(B_inode_id, "a-hard-link", a_inode_id)

symbolic link:

symlink_inode_id = allocate_inode(SYMLINK)

write_file(symlink_inode_id, "/A/a")

add_dentry(B_inode_id, "a-symbolic-link", symlink_inode_id)

c. Please calculate the latency of creating a **hard link** from "/B/a-hard-link" to "/A/a" given the process you described before. (Please also show how you calculate the result, list the formula) (7’) We assume that:

- The inodes of the three directories "/", "A" and "B" all use only one data block to store the directory entries. When adding new directory entries to these directories, there is no need to allocate new data blocks.
- There is no page cache (all disk operations will reach the disk).
- The basic read/write unit of the disk is a block.
- Seek time: 0.1ms
- Read time: 0.2ms/block (without seek)
- Write time: 0.3ms/block (without seek)

2 * (Fetch root inode (seek time + read time) + fetch data block of root inode (seek time + read time)) + fetch A inode (seek time + read time) + fetch data block of A inode (seek time + read time) + fetch B inode (seek time + read time) + write to data block of B inode (seek time + write time)

2 * ((0.1 + 0.2) * 2) + 3 * (0.1 + 0.2) + 0.1 + 0.3

2. Google File System (GFS) is a scalable distributed file system designed for large data-intensive applications.

a. Please explain the benefits (at least two) of replicating a chunk among multiple chunkservers. (4')

fault tolerance, scalable read

b. The two-phase process of writing to a chunk in GFS is listed below:

- Step 1: Master gives the client a list of chunkservers of current chunk, identifying the primary and secondaries
- Step 2 (Data flow): the client sends its modification of current chunk to all chunkservers
- Step 3 (Control flow): the client sends a commit request to the primary. The primary applies the modification of the client, and let the secondaries apply the modification. After all secondaries have applied the modification, the primary sends an ACK back to the client.

Please explain the benefits of splitting the writing process into two phases (data flow and control flow). (3')

data flow makes use of the network bandwidth, the single primary server will not be the bottolneck of transmitting data

control flow ensures the consistency of the chunk among all replicas

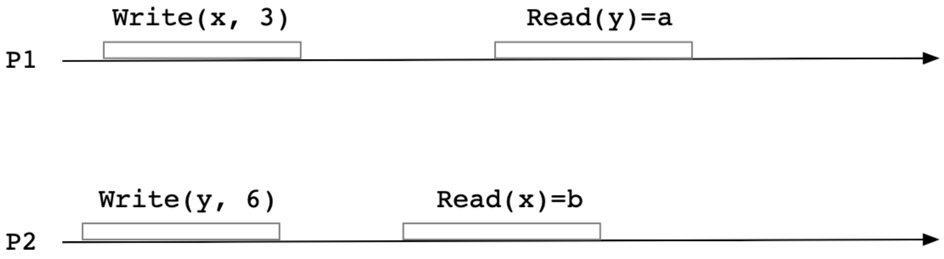
c. If many clients concurrently modify the same chunk, which step (step 1 / step 2 / step 3) mentioned in the previous question ensures that the modifications of the clients will be serialized? Explain how this step serializes the concurrent modifications from multiple clients. (3')

step 3

The primary chunkserver will determine the order of applying modifications from different clients, the secondaries will use the order determined by the primary chunkserver.

Problem 2: Consistency (20')

1. Please describe the differences between sequential consistency and linearizability. (4') Suppose you have two data, x and y, both are 0 initially. Fill in the following values at a and b, such that the execution satisfies sequential consistency but not linearizability. (4')



1.1 Sequencial consistency and linearizability both requires that the results of parallel operations be equivalent to some serial arrangement of their execution. But linearizability requires that the effects of the completion of an operation be visible to all operations that begin after the global physical time of the completion of that operation (for example, when a write is done at time t_1, all reads begin after t_1 will get the new value).
1.2 a = 0, b = 3 (3’) Or a = 6, b = 0 (3’) (Note: a=6, b=3 satisfies linearizable consistency; a=0, b=0 doesn’t satisfy sequencial consistency)

2. Lamport clock guarantees a total order between casually related events.

   Why? (2’) Sometimes, partial order is more appropriate, please describe why Lamport clock cannot give a partial order and how to extend it to do so. (4’)

2.1 Assuming that A and B have a causal relationship and that B depends on A, the server will update its local Lamport clock with the Lamport clock of A before applying the B operation, to ensure that its local Lamport Clock is greater than that of A. (3’)

2.2 Because Lamport Clock is always comparable, even for unrelated operations. Use vector timestamp, so that unrelated operations’ timestamp is uncomparable. (4’)

3. Suppose you are implementing a replicated key-value store that implements eventual consistency, which leverages the log to sync states between replicas. To truncate the log, you must determine which updates at a given replica’s log is stable. Assuming each update in the log is tagged with a lamport timestamp.

   a. We have described two approaches, namely, centralized, and decentralized approaches to determine which updates are stable. Please briefly describe them (3’).

Centralized: use the primary server to determine which ops can be turned into stable.
Decentralized: no primary server, the server decides whether to make the ops stable based on the Lamport Clock it received from other servers.

   b. Using the information in the table to find out stable writes of server1’s log. Assumption: server 1 uses lamport clock. (3’)

Approach	Server1’s all writes’	Server 1’s information on others	Primary’s response that Server 1 received	Server1’s stable writes
De-	<11, srv1>	Server 0’s Lamport	(no need to	<11, srv1>

centralized approach	<12, srv2> <14, srv1>	clock: 12 Server 1’s Lamport clock: 13 Server 2’s Lamport clock: 15	fill)	<12, srv2>
Centralized approach	<-, 11, srv1> <-, 12, srv2> <-, 14, srv1>	(no need to fill)	<1, 11, srv1> <2, 12, srv2> <3, 13, srv0> <4, 14, srv1>	<1, 11, srv1> <2, 12, srv2> <3, 13, srv0> <4, 14, srv1>

Problem 3: Paxos (20’)

Here is the pseudocode for Paxos.

```
state:
n_p (highest prepare seen)
n_a, v_a (highest accept seen)

propose(v):
   choose n, unique and higher than any n seen so far
   send prepare(n) to all servers including self
   if prepare_ok(n_a, v_a) from majority:
       v’ = v_a with highest n_a; choose own v otherwise
       send accept(n, v’) to all
       if accept_ok(n) from majority:
           send decided(v’) to all

acceptors’s prepare(n) handler:
   if n > n_p
       n_p = n
       reply prepare_ok(n_a, v_a)

acceptor’s accept(n, v) handler:
   if n >= n_p
       n_a = n
       v_a = v
       reply accept_ok(n)
```

1. Choose the earliest point that Paxos has agreed on a value, i.e., the agreed value will never change after the time. (2’) Please explain your reason. (6’)
   A. A leader receives prepare ok messages from a majority.
   B. A majority of acceptors accepts ok for the same proposal.
   C. A leader receives accept ok messages from a majority for a proposal.
   D. A majority of nodes receive decided messages.

B. Once a majority has received and accepted a given value v, then all future proposed values will also be v.

2. (1) A proposer sends out *prepare(4)* to five acceptors and receives back *prepare_ok(2, ‘abc’)*, *prepare_ok(2, ‘abc’)* and *prepare_ok(3, ‘xyz’)*. What would the proposer send during the send accept? (4’)

   Send *accecpt(4, ‘xyz’)* to all.

   (2) Suppose Line 9 is changed to:

```
9   v’ = any non-nil v_a; choose own v otherwise
```

Assume that the v_a variable in each acceptor starts with value nil. Describe a sequence of events where this change to Paxos would cause it to fail to agree correctly. (4’)

There are three acceptors, A1, A2, and A3. Proposer P1 sends propose messages to all, then accept(X) which only A1 receives, then proposer P1 stops. Proposer P2 sends propose messages to all, gets replies from just A2 and A3, sends accept(Y) which only A2 and A3 receive, then P2 crashes after hearing accept ok from A2 and A3. Thus the agreed value is Y, though not everyone knows it. Proposer P3 now sends propose messages to all, gets value X from A1 and value Y from A2 and A3, chooses X, and sends accept(X) to all three acceptors. That is an error: Y has already been agreed on, so all future proposals must be for Y.

A1:	p1	aX			p3	aX
A2:	p1		p2	aY	p3	aX
A3:	p1		p2	aY	p3	aX

3. Jack is trying to minimize the amount of storage stored on the disk. The pseudocode from lecture requires storing n_p, n_a, and v_a on disk. Jack decides that it may be sufficient to just store n_p and v_a on disk, and if a node reboots, the node sets n_a to the saved n_p value. Is this modified Paxos protocol, correct? If correct, please explain why. If not, give an example. (4’)

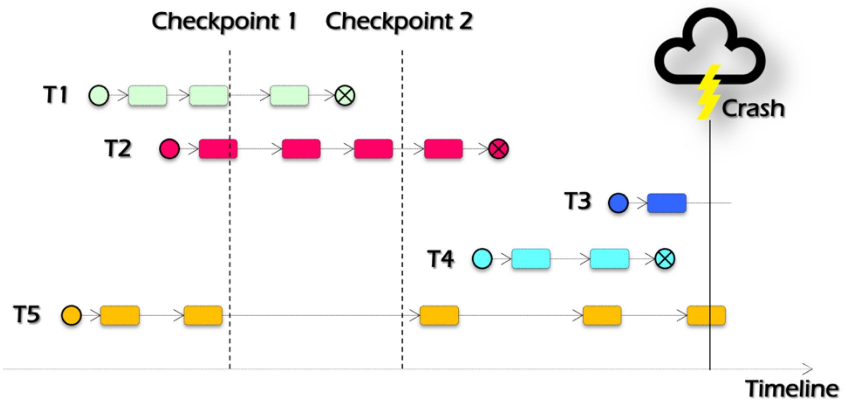
   No. (p for prepare, a for accept, v for value)

S1:	p1		p2	a2v2				
S2:	p1	a1v1			p3	reboot	p4	a4v1
S3:	p1		p2	a2v2			p4	a4v1

Problem 4: All-or-nothing and before-or-after (30’)

1. Assuming that we are now using redo-undo logging.

a. Which checkpoint time is correct (**1** or **2**)? (3’) Why? (4’)

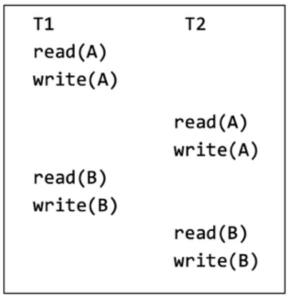


Checkpoint 1 is wrong. Because checkpoint in redo-undo logging should wait till no actions are in progress.

b. What do transactions (**T1~T5**) do when the system recovers from the crash? (5’)
T1: do nothing.
T2: redo its update based on the log in its checkpoint.
T3: undo its updates based on the log entries.
T4: redo its updates based on the log entries.
T5: undo its updates based on the log in both CKP & log entries.

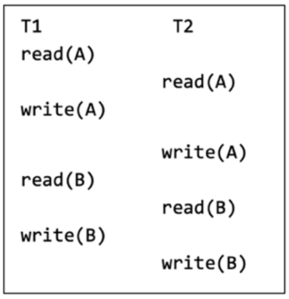
2. Given the following schedules, please use the conflict graph to determine which are conflict serializable, and which are not. (6’)

Schedule 1



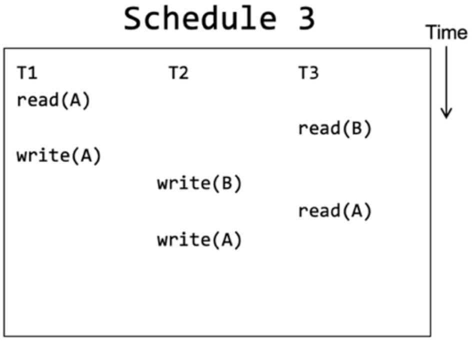
T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 2



T1	T2
read(A)	
	read(A)
write(A)	
	write(A)
read(B)	
	read(B)
write(B)	
	write(B)

Schedule 3



T1	T2	T3
read(A)		
		read(B)
write(A)		
	write(B)	
		read(A)
	write(A)	

S1: T1->T2. Yes.
S2: T1<->T2. No.
S3: T1->T3. T1->T2. T3->T2. Yes.

3. Given the following schedule, please determine whether it is conflict serializable or not. (2’) If yes, please describe your reason. If not, describe which serializability does it hold. (4’)

Time ↓

T1	T2	T3
read(A)		
		write(A)
	read(A)	
write(A)		
	write(A)	

View serializability. The conflict graph is cyclic. But when comparing it to running T1 then T3 then T2(serially), the final state is fine, and the intermediate reads are also fine. Also, final state serialzability is fine.

4. Assuming the following transactions are executed via OCC. After all transactions are finished, which of them commits? (2’) Which of them aborts (2’) What is the final state of the system (i.e., the value of A, B, C)? (2’) Note that the commit is executed atomically.

A	B	C
110	120	130

Time ↓

T1	T2	T3
read(C)		
	read(A)	
		read(C)
write(C,C+10)		
write(B,B+10)		
commit()		
	read(B)	
	write(B,B+10)	
		write(C,C+10)
		commit()
	commit()	

T1 and T2 is committed and T3 is aborted. A=110, B=140, C=140.

我承诺，我将严格遵守考试纪律。

承诺人：__________

题号									
得分									
批阅人(流水阅卷教师签名处)									