

上海交通大学试卷（期中）

（2020 至 2021 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: File System (25')

Panda recently learned an **inode-based file system** from the CSE lessons. Panda soon implemented his own file system, named PANDAFS.

In PANDAFS, the size of **each inode is 1KB**, and each **leaf node costs exactly one block (4KB)**. An inode contains **248 block pointers**, where the **first 247 pointers directly point to the first 247 blocks of the file**, while the last one **points to an indirect block** (indirect block only contains direct blocks).

The detailed structure of an inode in PANDAFS 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*248	Block pointers of this inode

- 1) Can PANDAFS support a file as large as 5MB? (4')
- 2) Panda wonders if he should add the **file name into the inode**. What is your suggestion? (4')
- 3) In CSE2020 classes, Panda downloaded many reading materials files, **one of which is a 128-byte file named "CSE2020.txt"**. Panda hopes to store it in **the "/" directory of PANDAFS on a hard disk**.
 - 3-1) Which order is preferred to **write this file to the disk**? **Why**? (4')
 - <1> update size, allocate a new block, write new data
 - <2> allocate a new block, write new data, update size
 - <3> allocate a new block, update size, write new data

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)									

3-2) How long does it take to read this file back from the disk? Given the following hard disk features:

- Sector size = 512 bytes (basic read/write unit)
- Seek time: 0.1 ms
- Read time: 0.2 ms/sector
- Write time: 0.3 ms/sector

Assume the file has been open for the first time, and its contents are continuous on the disk. (Note: it is better to give a detailed explanation of your calculation, e.g., Time = 1 seek root inode + 1 read root inode = 0.3ms) (4')

4) Given the formula:

$$\text{file utilization ratio} = \frac{\text{file size}}{\text{actually used disk size}},$$

Panda found that the file utilization ratio of PANDAFS is low when the file size is relatively small. Panda does not want significant changes to current inode data structure, otherwise the already stored files cannot be correctly read any more. How to improve file utilization ratio while remaining backward compatible? Please briefly describe your improved design. (Note: you may reuse the Padding field, or re-design FS operations such as create, read, and write, but anyhow the file utilization ratio should be $\geq 96\%$ for a 988-byte file.) (4')

5) Panda is glad to see his PANDAFS now supports good space-efficiency for small files. However, he still observes a very poor file lookup performance in the directory full of many small files. Please explain the reason, and help Panda with an optimization solution to PANDAFS. (5')

Problem 2: Transaction (26')

We have a transaction system which operates on a **key-value store**. Each transaction only **involves read(R)/write(W) operations on entries**. Now we have four transactions (T1-T4) which **access three entries (A, B, C)**.

1) What is final-state serializability, view serializability, and conflict serializability? Which **one is the strongest among them**? (4')

2) Consider the following schedule shown in the table, **which type of serializability introduced in class does this schedule achieve**? Please **explain your reasons**. (5')

T1	T2	T3	T4
W(B)			
W(C)			
	R(B)		
			R(B)
W(A)			
		R(A)	
			W(B)
		R(B)	
	R(C)		
	W(C)		
R(C)			
			R(A)

3) If the schedule above **cannot achieve conflict serializability**, can we just remove a **SINGLE action** in the schedule to **make it conflict-serializable**? If true, identify this action and give a **conflict equivalent serial ordering** of this **schedule**. (Eg. T1->T2->T3->T4) (4')

4) Now we want to implement conflict serializability in this system. There are two common methods: 2-phase locking (2PL) and optimistic concurrency control (OCC). Firstly, we use the 2PL mechanism in our system, now assume that there are two threads running T1 and T2 concurrently. After transaction execution we get the following schedule:

$W_1(B) \rightarrow W_1(C) \rightarrow W_1(A) \rightarrow R_2(B) \rightarrow R_1(C) \rightarrow R_2(C) \rightarrow W_2(C)$

Is this schedule possible? If so, please list the timepoints when two transactions acquire/release lock. (4')

5) Continue with the schedule produced by 2PL in question 4, if the thread running T1 crashes instantly after $R_2(B)$ which causes T1 abort, T2 must also be aborted because $R_2(B)$ read the value written by T1. We call this **Cascading Abort**. Can you modify the 2PL protocol to avoid cascading abort? Please give your design. (4')

6) Finally, we use the OCC mechanism in our system, this time we assume there are two threads running T2 and T4 concurrently. T2 and T4 are started at the same time. After transaction execution we get the following schedule:

$R_2(B) \rightarrow R_4(B) \rightarrow W_4(B) \rightarrow R_4(A) \rightarrow R_2(C) \rightarrow W_2(C)$

What will happen to the two transactions? List all possible situations. (5')

Problem 3: Fault Tolerance & Crash Consistency (29')

1. Please list at least commonly used mechanisms for fault tolerance. What are the commonalities of these methods? (4')

2. RAID5 allows one disk failure. If there are five disks in a RAID5 disk array system, and the MTTF of each disk is 100,000 hours, then:

(1) if we never repair the failure disk, what is the MTTF of the system? (4')

(2) if it takes 10 hours to repair the failure disk, what is the MTTF of the system? (3')

3. For the YFS in the lab, after it crashed, what should be checked during fsck? (3')

4. The disk space is limited, so log size cannot grow infinitely.

(1) If we can simply stop the world when doing truncating, how to truncate the log to save storage space? (3')

(2) For better performance, we want to continue accepting new transactions during truncating. How to truncate the log in this scene? (3')

5. Please describe the three modes (journal, ordered, writeback) of EXT4 file systems briefly. How to choose the mode in different scenes? (4')

6. Here are the steps when writing a file on EXT4 in ordered mode. Please describe the file system status if a crash happens at different times. (5')

```
write data ; flush ; log metadata ; (1)
flush ; (2)
log commit ; (3)
flush (4)
write metadata ; flush ; (5)
```

Problem 4: Before-or-after Atomicity (20')

You are the cook and owner for a popular food truck – business has been booming, however, your servers have just quit! So... you decide to use your programming skills to make an **automated ordering interface to automatically turn down or accept orders**. You come up with the following system:

At most **N orders can be accepted at a time**. If there are already N orders, **any extra orders are declined**. The chef will work **through one order at a time**.

```
extern int N;
int orders = 0;
pthread_mutex_t mutex;
pthread_cond_t cond;

void *order() {
    while(true){
        if (orders < N) {
            orders += 1;
            return;
        }
    }
}

void *cook(){
    while(true) {
        orders -= 1;
        cookOrder();
    }
}
```

To **synchronize the two functions**, you decide to use **lock and conditional variables**. Now, please answer following questions:

1. What is the problem of the following implementation of **order()**? (4')

```
void *order() {
    acquire(mutex);
    while(true){
        if (orders < N) {
            orders += 1;
            release(mutex);
            return;
        }
    }
}
```

2. Here are two implementations of **order()**, but both have problems. Please list the problems. After that, please give your solution. (6')

```
void *order() {
    acquire(mutex);
    while(true) {
        if (orders < N) {
            orders += 1;
            release(mutex);
            return;
        }
        release(mutex);
        yield();
        acquire(mutex);
    }
}
```

```
void *order() {
    acquire(mutex);
    while(true) {
        if (orders < N) {
            orders += 1;
            release(mutex);
            notify(have_order)
            return;
        }
        release(mutex);
        wait(not_full);
        acquire(mutex);
    }
}
```

3. To implement new **wait()** API, your partner proposes to use a modified version of Semaphore to implement the condition variable class as follows:

```
Semaphore.set(0); //Initialize semaphore to 0
Wait(CV cv, Lock lock) {
    Semaphore.P(cv, lock); //Special semaphore that releases lock if it waits
}
Notify(CV cv) {
    Semaphore.V(cv);
}
```

Do you think this proposal will work or not? Please explain your decision. (5')

Hint 1: A simple way to understand P(wait) and V(signal) operations is:

- **P()**: Decrements the value of semaphore variable by 1. If the **new** value of the semaphore variable is negative, the process executing wait is blocked. Otherwise, the process continues execution.
- **V()**: Increments the value of semaphore variable by 1. After the increment, if the **pre-increment** value was negative, it signals a blocked process to continue.

Hint 2: You can consider the different meaning of order of execution between semaphore and conditional variables.

4. Consider the following two functions implementing a producer and consumer:

<pre>// Producer void send(item) { acquire(lock); enqueue(item); printf("before signal()\n"); signal(lock); printf("after signal()\n"); release(lock); }</pre>	<pre>// Consumer item = get() { acquire(lock); while (queue.isEmpty()) { printf("before wait()\n"); wait(&lock); printf("after wait()\n"); } item = dequeue(); release(lock); }</pre>
--	---

Assume two threads T1 and T2, as follows:

T1	T2
send(item);	item = get();

What are the possible outputs? (5')