Computer Science 162
Anthony D. Joseph
University of California, Berkeley
Spring 2014
Midterm 2
April 28, 2014

Name	
SID	
Login	
TA Name	
Section Time	

This is a closed book and one 2-sided handwritten note examination. You have 80 minutes to answer as many questions as possible. The number in parentheses at the beginning of each question indicates the number of points for that question. You should read all of the questions before starting the exam, as some of the questions are substantially more time consuming. Write all of your answers directly on this paper. Make your answers as concise as possible. If there is something in a question that you believe is open to interpretation, then please ask us about it!

Grade Table (for instructor use only)

Question	Points	Score
1	27	
2	23	
3	25	
4	25	
Total:	100	

Foreword

After projects 3 & 4, you and your group realize that the world needs a better key-value store. As an intelligent group of students with a good idea, you all, of course, decide to start a company! Over the next 80 minutes you will use the knowledge you've gained in this class to design a more scalable, fault tolerant key-value store. (Unfortunately, you still have to take this test by yourself...)

1. (27 points) **Bleeding Edge Security**

   (a) (3 points) You want to design a secure username/password login scheme. You remember from CS162 lecture that it is a bad idea to store passwords in plaintext, so instead you decide to store SHA256(password) on your server's database. Kelvin, your security guru friend, is skeptical about this scheme but wants to keep it fast (i.e., no slow hashing allowed). *In no more than three sentences*, explain how could you improve it without imposing any requirements on the password?

   Solution: Use a salt to fight against confirmation/amortized attacks

   (b) (4 points) The public key for your server is distributed by the certificate authority George's Certificates For Yiu. *In no more than three sentences*, explain what could go wrong if George's private key is compromised, and how would the situation be fixed? *Be specific.*

   Solution: Certificate could be modified for attacker to give your server an impersonated public key. Would want to put on revocation list, and get a new certificate.

   (c) (8 points) You are discussing security schemes with Nick and he asks you about the pros and cons of various methods. Pick true or false for the following statements.

   i. (2 points) Asymmetric cryptography is much slower than symmetric cryptography
      - √ **True**
      - ◯ False

   ii. (2 points) Digital certificates bind a host's identity with its private key
      - ◯ True
      - √ **False**

   iii. (2 points) The Heartbleed OpenSSL vulnerability was a virus
      - ◯ True
      - √ **False**

   iv. (2 points) Setting the no-execute bit on the stack prevents all buffer overflow attacks
      - ◯ True
      - √ **False**

(d) You now decide to consider schemes of encryption for messages sent back and forth between the server. You aim to provide *Confidentiality*, *Integrity*, *Authentication*, and *Non-repudiation*.

For each scheme, fill in all of the properties that will hold in the presence of a man-in-the-middle attacker. If an approach fails entirely (ie. does not make sense, cannot be computed, or a message cannot be read by the intended recipient), fill in only Broken. Fill in only None if the scheme is possible but none of the properties hold.

Assume all keys have been securely distributed, and the following constants:

- KS = the server's public key
- $KS_{PRIVATE}$ = the server's private key
- KC = the client's public key
- $KC_{PRIVATE}$ = the client's private key
- $M1$ = the client's message to send
- $M2$ = the server's message to reply with
- E_K is a secure encryption algorithm, using K as a key.
- $X|Y$ denotes a concatenation of X and Y

i. (3 points) The client sends message $E_{KC}(M1)$, and the server replies with $E_{KS}(M2)$

- √ **Broken**
- ◯ Confidentiality
- ◯ Integrity
- ◯ Authentication
- ◯ Non-Repudiation
- ◯ None

Solution: Broken - cannot decrypt

ii. (3 points) The client sends message $E_{KS}(M1)$, and the server replies with $E_{KC}(M2)$

- ◯ Broken
- √ **Confidentiality**
- ◯ Integrity
- ◯ Authentication
- ◯ Non-Repudiation
- ◯ None

Solution: Man in the middle can still encrypt the same packets pretending to be S or C.

iii. (3 points) The client sends message $E_{KS_{PRIVATE}}(M1)$, and the server replies with $E_{KC_{PRIVATE}}(M2)$

- √ **Broken**
- ◯ Confidentiality
- ◯ Integrity
- ◯ Authentication
- ◯ Non-Repudiation
- ◯ None

iv. (3 points) The client sends message $E_{KS}(len(N1)|N1|M1)$, and the server replies with $E_{KC}(len(N2)|N2|M2)$. $N1$ and $N2$ are independently and randomly generated nonces.

- ◯ Broken
- √ **Confidentiality**
- ◯ Integrity
- ◯ Authentication
- ◯ Non-Repudiation
- ◯ None

Solution: Confidentiality only. The nonces are different so they don't stop anything.

2. (23 points) **The Onion Network**
Now, the next point is to make certain that we are able to connect to all our clients.

Since all of our engineers graduated from Berkeley with 162 experience, we want to make certain that we are able to connect to all our clients, and to optimize the KVStore networking performance. We decided to have our own implementation of the system which will improve networking performance. The only application that uses our router is our KVStore.

Each question is independent of each other.

(a) (5 points) Since we're currently a super small startup, all of our servers reside in a single room and are directly connected to the same gateway router. All incoming and outgoing traffic pass through this router. To make routing faster, we add an extra tag to each request (both client and server side). We then upgrade our router with our own custom firmware that forwards our KVServer packets by reading this application-level tag. You can think of this as clients sending requests with serverIDs as the tag and our router forwarding on these serverIDs directly.

Is this design feasible? Why or why not?

Solution:
It will work with the existing layering system. This is because each layer is independent of each other and the application layer is at the very top. So we can do anything we want in that layer and as long as our internal router supports it, it will be able to forward it correctly.

(b) (5 points) Our startup wants to make sure that we move fast and break things. Instead of using and understanding TCP, we designed our own transport layer protocol (George Datagram Protocol, GDP) that gives no guarantees since we want to fail fast. We decide to assign it a random protocol number, 42, since the OS needs this protocol within the IP field to determine how to handle it. Unless we receive a success reply back from our KVStore, we assume the transaction failed. Our protocol only includes the two port numbers (source and destination) within the transport layer. Even less overhead than UDP!

Is this design feasible? Why or why not?

Solution: This will not work with the existing system. This is because any existing hosts will not be able to match this protocol to the real protocol with the number 42. The OS will be confused as to how to parse this packet and it will fail.

Now, we must consider the latency of our network. Using TCP connections, we want to optimize our KVStore for Cal students, so we analyze connections from Berkeley. Here are some constants and assumptions:

- The latency between Berkeley and the datacenter is 5 milliseconds.
- No connection has been made to start with.
- Each packet can contain up to 1500 Bytes.
- The window size is 100 packets.
- The server does not need time to process the information and can send the correct response instantly.
- Kilo is 10^3 and Mega is 10^6
- The SYN/ACK by itself **takes up 0 bytes** in data transfer.
- Since the ACK is zero bytes, SYNs and data can be sent along with it.

We want to be able to figure out the time it takes for the client to send over data for 1000 bytes, and receive a response from the server that will also be 1000 bytes. Assume that this includes all memory it needs for headers. **Round to the nearest ms**

(c) (4 points) What is the time that it takes for you to get confirmation that your request has gone through if you were using AT&T from Berkeley and have 500 Kbps bandwidth.

Solution: It takes 5 ms to send the SYN, and then another 5 ms to send the ACK back. Now, we want to calculate how long this packet stays in the pipe. With 500 Kbps bandwidth, it takes 16 milliseconds to transfer data. This means that it will take 21 ms for the client to send to the server, and assuming no processing time, it will be 21 ms to get a response back. This means, that it is a total of 52 ms to get a confirmation including the three way handshake.

(d) (4 points) What is the time that it takes for you to get confirmation that your request has gone through if you were using Berkeley dorm internet and have 100 Mbps bandwidth?

Solution:
It still takes 5 ms to send the SYN and another 5 ms to send the ACK back. At this speed, it takes roughly 0.08 ms to transfer. So each side sending information is 5.08 ms. It will take 20.16 ms to send the data. We will accept 20 ms, since it is sort of negligible

(e) (5 points) We have now decided that for a KV store with just strings would be too boring. Ignoring the security concerns, we will now let anybody be able to store any type of file of any size that is mapped to a specific key. This allows one person to upload the latest Game of Thrones episode and have his friend download it as long as they have the correct key.

Using the same assumptions as above, we now have a file that is 300 MB. How long does it take to upload at 500 Kbps? And the server has been upgraded to only send back a hash of each packet, so the ACK for each data packet will be neglible, so we can treat it as zero bytes.

Round to the nearest second.

Solution: 500 Kbps

The initial latency SYN and ACK will take 10 ms.

We need to send 200,000 packets. Each packet will take 24 ms to send over. And it will take 5 ms for the server to send a response back. Due to the fact that our connection speed is so slow, we won't actually ever be able to utilize our window size since we can only have one thing transferring before we already get the ACK for the previous packet. It will take 4800 seconds to send over everything, and then an extra 10 ms to receive the last ACK. Since we have the 5 ms delay of us sending over the wire, and then the 5 ms delay of the server sending us back. But these milliseconds are negligible.

3. (25 points) **Synchronization Hard is**
Your kvstore service is getting popular! In order to keep things fast, you're having to schedule multiple transactions at once.

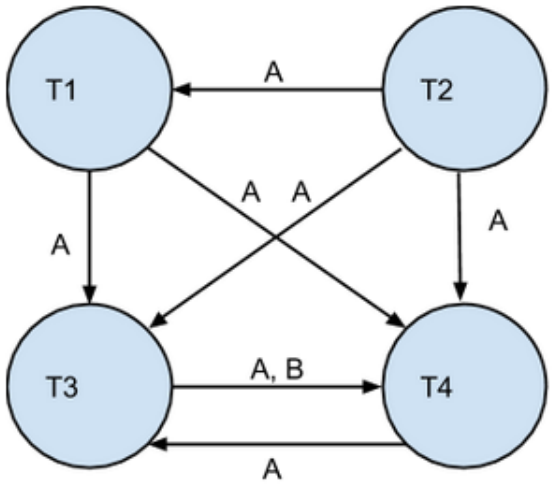
Consider the following schedule of transactions (T1, T2, T3, T4) below, operating on data objects A and B.

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

Time →

(a) (5 points) Draw the dependency graph for this schedule. Be sure to label all nodes and edges.

Solution:



(b) (4 points) Is this schedule conflict-serializable? If so, provide a candidate schedule. Please explain your answer for full credit.

Solution: No, dependency graph is cyclic.

(c) (4 points) Is this schedule serializable? If so, provide a candidate schedule. Please explain your answer for full credit.

Solution: Yes, while the dependency graph is cyclic, there aren't any reads after the last writes so it doesn't matter. Possible schedules: 2, 1, 4, 3

In the spirit of completely "bulletproofing" your system, Isaac convinces you to consider 2-phase locking for your kvstore scheduling. To refresh your memory, he asks you some questions:

(d) (4 points) Does the following schedule correctly implement strict 2-phase locking? If not, please explain your answer. Assume D, E, and F are separate data objects.

Transaction 1	**Transaction 2**
Acquire X(D)	Acquire X(E)
R(D);	R(E);
D = D + 10;	E = E + 10;
W(D);	W(E);
Release X(D)	Acquire X(F)
	R(F);
	F = F - E;
	W(F);
	Release X(E)
	Release X(F)

Solution: Yes, even though the locks are independent it still follows 2PL (no locks are released until the end of a transaction)

(e) (8 points) Schedule the following transactions according to (normal) 2-phase locking, for the minimum possible execution time.

Assume each instruction takes one time unit, and acquiring/releasing a lock takes zero time units. **Once a transaction acquires a lock it cannot upgrade or downgrade.**

Transaction 1	**Transaction 2**
R(A)	R(A)
R(B)	R(C)
A = A + B	C = C - A
W(A)	W(C);
B = B - A	A = A + C
W(B)	W(A)

Complete your schedule in the table below, and at each stage specify which locks the transaction holds during that time unit. Each row represents one time unit.

The first row has been completed for you. You may not need all of the rows to finish.

T1 instructions	**T1 locks**	**T2 instructions**	**T2 locks**
R(A);	X(A)		

Solution:

T1 INSTRUCTIONS	T1 LOCKS	T2 INSTRUCTIONS	T2 LOCKS
R(A);	X(A)		
R(B);	X(A), X(B)		
A = A + B;	X(A), X(B)		
W(A);	X(A), X(B)		
B = B - A;	X(B)	R(A);	X(A)
W(B);	X(B)	R(C);	X(A), X(C)
		C = C - A;	X(A), X(C)
		W(C);	X(A), X(C)
		A = A + C;	X(A)
		W(A);	X(A)

This page has intentionally been left blank

DO NOT WRITE ANSWERS ON THIS PAGE

4. (25 points) **Chunkey Monkey**

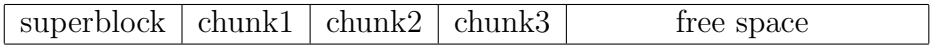
   In order to further optimize KV store performance we are going to fit our servers with a custom file system.

   What makes this file system different is that because our key value store is mostly in memory, and we are going to use our disk as a best effort backup so we only want to do large sequential writes.

   With those considerations, we are going to adopt the following simple design.

   - Each file will represent one KV pair
   - We will abandon the idea of inodes entirely, instead we will have fnodes which contain all the metadata about file and the key and value (which will be the only data stored in our files anyway).
   - An fnode will be exactly one block large
   - We will never write individual files, instead we will have a notion of a "chunk" which will be a batch of a 1000 files and writes will occur chunk at a time allowing us to use the full bandwidth of our disk.
   - Disk utilization will only be measured chunk-wise, if a part of a chunk is used the super block will say all the bytes in the chunk are used.
   - Writes don't actually flush to disk until a **timeout** occurs or a **quarter** of the files in that chunk have been updated.

   Here is a visual representation of our file system layout:



   The superblock will contain all the metadata needed for the file system to function. The superblock is the first block in the file system so its easy to find, the data in the super block doesn't have to take up the size of a block, but that block is not used for anything else.

   The computers that run this file system are 32 bit intel x86 machines with a page size of 4 kB and 8 gb of ram.

(a) (4 points) Given a disk with 4 Tib capacity (1 Tib is 2^{40} bytes), what block size will optimize the number of fnodes you can store (given that you don't care how small an fnode can be). Remember you must be able to actually address all your blocks. Why is this the best you can do?

Solution: 1 Kib (because 32 bit operating system)

(b) (4 points) Given the same disk size, what is the block size that will optimize the size of an fnode (given that you don't care about how many fnodes will be stored). Why is this the best that you can do? (Or why larger values don't make sense)

Solution: 4 Kib (or one page)

(c) (4 points) In this filesystem, we save an extra seek in reading the file by storing the file metadata and file data in the same on-disk structure. What similar thing does the Fast File System (FFS) architecture do to minimize seek time when reading a file?

Solution: FFS stores inodes NEAR the data (same cylinder group)

(d) (13 points) Lets write some code! You need to implement the write function for this file system. Kelvin has copy pasted some code from the Linux kernel for us, we've deleted all the parts that don't work for this usecase. You can assume the functions that are declared but not defined are defined else where and work properly. Remember if you are going to write you write the ENTIRE chunk.

Please fill in all the blanks on the next page.

```
#define FS_SIZE 4398046511104

// Block size is file system dependent.
// This is here so that you know it exists.
#define BLOCK_SIZE <some number here>

//This structure is laid out exactly like this on disk!
struct super_block {
    int32_t last_chunk_num_used;  // the largest used chunk block number
    int32_t disk_space_used // amount of disk space used
};

// This structure is laid out exactly like this on disk!
struct fnode {
    int32_t key;
    int32_t value;
    int8_t meta_data[______________________]; // Fill me in!
};

// This is just a logical in memory representation of a chunk
// chunks on disk will just be a series of 1000 fnodes
struct chunk {
   int32_t bnum_begin; // the block number this chunk starts at
   struct fnode files[1000];
   int32_t updated; // count of how many files in this chunk are updated
};

// This is just a logical in memory representation of our fs for easy use
struct kv_fs {
    struct super_block* sb;
    // How many chunks are there?
    struct chunk chunks[FS_SIZE/______________________]; // Fill me in
};

// Writes raw block data to specified block offset on device
void write_to_disk(void* block_data, long b_num);

// Maps key to a particular chunk
int key_to_chunk_num(int32_t key);

// Maps key to a block offset WITHIN a particular chunk
// (if it does not exist it will create an empty fnode for it)
int key_to_block_num(int32_t key);
```

```
/* This method updates the in memory data structures of the
   File system to reflect a particular kvput operation
   It will update the superblock and update/create the necessary
   fnode. */

void kv_put(struct kv_fs* fs, int32_t key, int32_t value) {

    int chunk_num = ____________________________________;

    // If new chunk update superblock

    if (chunk_num > ____________________________________) {

         ____________________________________________;
        fs->sb->disk_space_used += BLOCK_SIZE*1000;
    }
    fs->chunks[chunk_num].files[____________________].value = value;

    ____________________________________________________________;

    write(fs, &fs->chunks[chunk_num], 0);
}

/* This method will check the state of the file system and write to
   disk if the chunk has enough changed fnodes or the timeout has
      been reached */

void write(struct kv_fs* fs, struct chunk* chunk, int timeout) {
    if (_____________________________________________ || timeout) {

        for (int i = 0; i < 1000 ; i++) {

            write_to_disk(&chunk->files[i],

                          ____________________________________);
            chunk->updated = 0;
        }

        write_to_disk(____________________________, _________________________);
    }
}
```

Solution:

```
#define FS_SIZE 4398046511104


// Block size is file system dependent.
// This is here so that you know it exists.
#define BLOCK_SIZE 4096

struct super_block {
    int32_t last_chunk_num_used;
    int32_t disk_space_used;
};

// This structure is laid out exactly like this on disk!
struct fnode {
    int32_t key;
    int32_t value;
    int8_t meta_data[BLOCK_SIZE - 8];
};

// This is just a logical in memory representation of a chunk
// chunks on disk will just be a series of 1000 fnode blocks

struct chunk {
   int32_t bnum_begin;
   struct fnode files[1000];
   int32_t  updated;

};

struct kv_fs {
    struct super_block* sb;
    struct chunk chunks[FS_SIZE/(1000*BLOCK_SIZE)];
};

int write_to_disk(void* block_data, long b_num);

int key_to_chunk_num(int32_t key);

int key_to_block_num(int32_t key);

void kv_put(struct kv_fs* fs, int32_t key, int32_t value) {
    int chunk_num =  key_to_chunk_num(key);
     if (chunk_num > fs->sb->last_chunk_num_used) {
        fs->sb->last_chunk_num_used++;
```

```
        fs->sb->disk_space_used += BLOCK_SIZE*1000;
    }
    fs->chunks[chunk_num].files[key_to_block_num(key)].value =
                        value;
    fs->chunks[chunk_num].updated++;

}

void write(struct kv_fs* fs, struct chunk* chunk, int timeout) {
    if (chunk->updated > 250 || timeout) {
        for (int i = 0; i < 1000 ; i++) {
            write_to_disk(&chunk->files[i],
                          chunk->bnum_begin + i);
            chunk->update = 0;
        }
        write_to_disk(fs->sb, 0);
    }
}
```

This page has intentionally been left blank

DO NOT WRITE ANSWERS ON THIS PAGE