OS 2015

Ex3: Block Chain

Due on: Monday, 11 May 2015, 23:55

Part I: Coding Assignment (90 pts)

Background

In early 2009, some anonymous entity using the name *Satoshi Nakamoto* published an open-source software implementing a decentralized online payment method. The basic unit of this system was named *Bitcoin*. This decentralized digital currency (often named cryptocurrency) is said to be the money built for the age of the internet.
One of the most revolutionary aspects of the Bitcoin system is it's ledger, the heart of the Bitcoin system, the *Block chain*.
The Block chain is a public ledger (ספר חשבונות) that records the bitcoin transactions (TXs), without relying on any trusted central authority. This is done by broadcasting each and every TX ("Bob sends *C* Bitcoins to Alice") to all the nodes in the network. The nodes validate the TX and add it to their ledger, and then broadcast the recent changes in the ledger to other nodes.
Thus the block chain, the ledger, is a distributed database, where every node holds a copy of it, validates its data, and updates the database.
As a revolutionary data structure, many ideas (and derived startups) try to utilize the nature of the block chain in a range of different uses other than bitcoin, thus you may find it interesting to jump along to the *Further Reading* section for deeper and wider explanations about the Bitcoin system and the block chain.

The Block Chain

A block chain is a transaction database shared by all nodes participating in a decentralised trust system (such as Bitcoin). The blockchain is a public ledger of all

transactions that have ever been executed, and is constantly growing as 'completed' blocks are added chronologically.
Each block contains a hashed piece of data, generated by using a cryptography hash function.
The block chain consists of, surprisingly, a linked chain of data blocks. The block chain begins with a genesis (root) block. Each block points only to one preceding block, it's 'father'. Every block will be connected to the genesis block, either directly or via a chain of the genesis descendants. Ideally, they form a linear chain, but because of the distributed nature of the system, several new blocks may be added at the same time, and then all of them will point at the same 'father' block. If you find it hard to imagine try to think of a backwards linked list, or rather a backwards pointing tree.
As more blocks are added, the longest branch becomes the 'official' one. One of the strongest aspects of Bitcoin's security is the reliance on the longest chain in the block chain. For example if two blocks 'argue' whether X paid 5 bitcoin to Y (as written in block K) or X paid 5 bitcoin to Z (as written in block L), the 'winning' block is the one within the longest chain. The reason is quite innate in the Bitcoin system itself and we will try to keep this property.

Cryptographic Hash Functions and hashing

These are a set of functions that deterministically map strings (of any length) to a fixed-size string. Crypto hash functions also have the following properties:

- Efficiency: It is easy to compute some x's hash(x).
- Collision Resistance: It is computationally hard to find a pair x,y such that hash(x)=hash(y).
- Hash(x) reveals little about x.
- Hash(x) "looks" like a completely random string.

In this exercise one of your tasks will be to use a hash function to hash given data (in order to attach it to the block chain). Hashing the data shouldn't be a blocking operation, as the hashing itself is simply computing some function. You will use a supplied crypto hash function in order to hash the block's data.

Note: Supplementary hash functions will be uploaded to moodle in the upcoming week. Meanwhile you may use <functional> library hash function, or leave it out for later on.

Further reading

The web is full with additional data about Bitcoin and block chain, technology enthusiasts and curious students may find this subjects quite challenging and interesting.

Furthermore, as the Bitcoin system and the block chain combine many CS and Mathematics areas, so there is exciting academic research going on.
In the Hebrew U., Dr. Aviv Zohar is conducting research regarding these fields. You may find interesting courses and projects under his supervision.

Bitcoin:

- http://en.wikipedia.org/wiki/Bitcoin
- https://bitcoin.org/en/how-it-works
- http://www.coindesk.com/bitcoin-explained-five-year-old/

Block Chain:

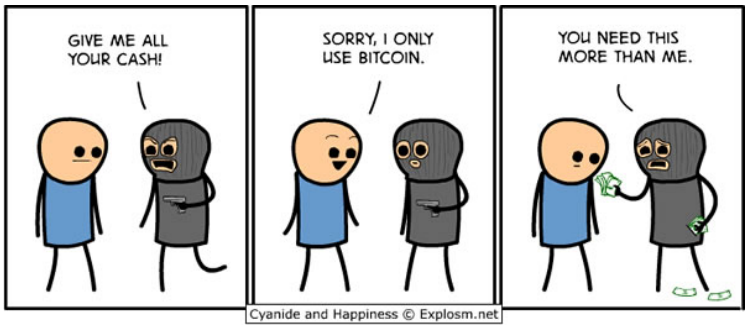
- http://en.wikipedia.org/wiki/Bitcoin#Block_chain
- http://gizmodo.com/whats-the-blockchain-and-why-does-bitcoin-depend-on-it-1698025216
- http://www.youtube.com/watch?v=QzDO44oZWtE
- Different uses for block chain

Satoshi Nakamoto's original paper:

- https://bitcoin.org/bitcoin.pdf

Cryptographic Hash Functions

- http://en.wikipedia.org/wiki/Cryptographic_hash_function



TL;DR *(your responsibility!)*

You will implement an asynchronous simplified block chain management library, that will store hashed pieces of data in a chain-like structure.

Assignment

Note: Block as part of the *block chain* will be written as *Block*, block as in blocking threads by mutex will be written as block.

Consider the introduced *Block chain* (BC) data structure, used in a multi-threaded system that accepts pieces of data from several client threads concurrently. Each of these client thread tasks comprises of adding their piece of data to the currently longest chain as a single uninterrupted atomic operation.
You will not implement a trivial solution of locking the whole BC whenever a client wishes to add a piece of data. You will rather implement an efficient non-blocking package, as a static library.
In order to implement the library you will need to create a daemon thread that manages the asynchronous data addition operations. Daemon threads are essentially service providers for other threads running in the same process as the daemon thread .
The public interface of your library is given by *blockchain.h* and is explained below. Obviously, there are some internal functions (and data structures) that you may find necessary to implement. These functions are not visible outside the library; therefore, you are not restricted in their number, signatures, and content, as they are part of your private implementation. Your code should not have any memory leaks, but you are allowed to use one (and only one!) statically initialized mutex that will not be destroyed by your library. Also, you may assume that no more than INT_MAX *Blocks* will be written to the *Block chain*.

Library functions

```
int init_blockchain();
```

DESCRIPTION: This function initiates the *Block chain*, and creates the genesis *Block*. The genesis *Block* does not hold any transaction data or hash.
This function should be called prior to any other functions as a necessary precondition for their success (all other functions should return with an error otherwise).
RETURN VALUE: On success 0, otherwise -1.

```
int add_block(char *data , size_t length);
```

OVERVIEW: For each added *Block* you will need to immediately find a “father” *Block* to point to; this will be the current end of the longest chain. Thus, each *Block* that you create will point

to a *Block* that is already attached to the *Block chain*.
You will need to hash the given data before attaching the *Block*. Hashing may take some time, so don't be surprised, and don't think of hashing the data during the add_block execution.
DESCRIPTION: Ultimately, the function adds the hash of the data to the *Block* chain.
Since this is a non-blocking package, your implemented method should return as soon as possible, even **before the *Block* was actually attached to the chain**.
Furthermore, the father *Block* should be determined before this function returns. The father *Block* should be the last *Block* of the current longest chain (arbitrary longest chain if there is more than one).
Notice that once this call returns, the original data may be freed by the caller.
RETURN VALUE: On success, the function returns the lowest available block_num (> 0), which is assigned from now on to this individual piece of data.
On failure, -1 will be returned.

```
int to_longest(int block_num);
```

OVERVIEW: Due to the design of this package (cf. theoretical part) there may be multiple chains pointing back at the same "father" *Block*. In some cases the user/s would like to attach a specific *Block* to the end of the real-time longest chain.
They may constrain it by applying this function over a specific block_num, such that whenever this *Block* will be attached, it will be attached to the real-time longest chain.
DESCRIPTION: Without blocking, enforce the policy that this block_num should be attached to the longest chain at the time of attachment of the Block. For clearance, this is opposed to the original add_block that adds the *Block* to the longest chain during the time that add_block was called.
The block_num is the assigned value that was previously returned by add_block.
RETURN VALUE: If block_num doesn't exist, return -2; In case of other errors, return -1; In case of success return 0; In case block_num is already attached return 1.

```
int attach_now(int block_num);
```

OVERVIEW: In some scenarios we would like some piece of data to be added immediately to the database, at the instant that the function is called, e.g. some sort of 'real-time' relevant data.
DESCRIPTION: This function blocks all other *Block* attachments, until block_num is added to the chain. The block_num is the assigned value that was previously returned by add_block.
RETURN VALUE: If block_num doesn't exist, return -2;
In case of other errors, return -1; In case of success or if it is already attached return 0.

```
int was_added(int block_num);
```

DESCRIPTION: Without blocking, check whether block_num was added to the chain.
The block_num is the assigned value that was previously returned by add_block.
RETURN VALUE: 1 if true and 0 if false. If the block_num doesn't exist, return -2; In case of other errors, return -1.

```
int chain_size();
```

DESCRIPTION: Return how many *Blocks* were attached to the chain since init_blockchain.
If the chain was closed (by using close_chain) and then initialized (init_blockchain) again this function should return the new chain size.
RETURN VALUE: On success, the number of *Blocks*, otherwise -1.

```
int prune_chain();
```

OVERVIEW: Since additional 'short' sub-chains (cf. theoretical part) make our data structure heavy in memory as well as lowers the number of blocks that we can add to the blockchain, some sort of pruning (גיזום) should be made, so as to free the block_nums and the memory. Moreover, we wish to keep a 'consensus' type data, which is defined in the bitcoin protocol as the longest chain.
Intuitively, you can think of it as a tree growing to all directions (short sub-chains), but as the maintainer of the tree you want it to grow to some specific direction, therefore you prune all these short sub-chains out of the tree.
DESCRIPTION: Search throughout the tree for sub-chains that are not the longest chain, detach them from the tree, free the blocks, and reuse the block_nums.
RETURN VALUE: On success 0, otherwise -1.

```
void close_chain();
```

DESCRIPTION: Close the recent blockchain and reset the system, so that it is possible to call init_blockchain again. Non-blocking.
All pending *Blocks* should be hashed and printed to **terminal** (stdout).
Calls to library methods which try to alter the state of the Blockchain are prohibited while closing the Blockchain. e.g.: Calling chain_size() is ok, a call to prune_chain() should fail.
In case of a system error, the function should cause the process to exit.

```
int return_on_close();
```

DESCRIPTION: The function blocks and waits for close_chain to finish.

RETURN VALUE: If closing was successful, it returns 0.
If close_chain was not called it should return -2. In case of other error, it should return -1.

Assumptions & Clarifications

1. By default, each *Block*'s father is determined at the time of "submission" to the chain (i.e. when add_block is called). The determined father is therefore the last *Block* in the longest chain, at the time the client called add_block. Exceptions to this rule are introduced when using *to_longest*. In this case, the father is determined in the same way, but at the time of actual insertion to the chain (meaning, when the current *Block* is clearly the next *Block* that will join the *Block* chain).
2. Whenever there is more than one (>1) longest chains, you should **randomly** refer to one of them. You may think of it as: attach *Block* to a random element i in the set {longest chains}.
3. There will be no more than INT_MAX *Blocks* attached to the *Block chain*.
4. Data extraction from the *Block chain* is not of your concern, meaning you should not worry about the lack of functions to access the *Block chain*.

PART II: Theoretical Questions (10 pts)

The following questions are here to help you understand the material. We don't try to trick or fail you, so please provide a short straightforward answer and not a full thesis.

1. What is the library feature that enables multi-pointing toward an arbitrary father *Block*, what is the main parameter that affects the number of multi-pointer *Blocks*, can we change it? [**No more than 3 lines**] (4 pts)
2. In the ex we allow creation of 'short' sub-chains, and therefore we need to add a pruning functionality. Think of an approach that will enable only one long chain, thus relieving us from the boring pruning job. [**No more than 3 lines**] (4 pts)
3. Can you identify any other race conditions that we didn't deal with? name one. (2 pts)

Background reading and Resources

1. Carefully read the pthread man-pages, which are grouped into 3 major classes:
   1. Thread management:
      Working directly on threads.
   2. Mutexes:
      Dealing with synchronization ("mutual exclusion").
   3. Condition variables:
      Communication between threads that share a mutex.

2. POSIX Threads Programming a good tutorial

Submission

Submit a tar file online, containing the following:

- A *README* file. The README should be structured according to the course guidelines, and contain an explanation on how and why your library functions are built the way they are. Specifically you should explain any locking that you use, or alternatively why locking is not needed.
- The source files for your implementation of the library. In particular, *blockchain.c* or *blockchain.cpp*.
- Your *Makefile*. Running ***make*** with no arguments should generate the *libblockchain.a* library.

Do not change the header file. Your exercise should work with our version of blockchain.h.

Guidelines

- This exercise objective is multi-threaded programming; therefore, make sure you protect all shared resources. You may want to use a multi-processor like ***river*** to test your code.
- You may find using C++ advantageous in this exercise, due to STL's abstract data types.
- Make sure to check the exit status of all the system calls and pthread calls you use.
- Use the Q&A forum, and be sure to check the existing topics, as some of your questions might have already been answered.
- Coding guidelines are obligatory.

Needless to say, start early.

Good Luck