Contents

1 README

```
akiva_s,roigreenberg
Akiva Sygal(305277220), Roi Greenberg(305571234)
Ex: 3

FILES:
README
blockChain.cpp -- a file that implements the given blockChain.h
makefile

REMARKS:
we used the next DASTs: one map that holds the information about every block,
a list of all leaves in the tree-chain (which is enough to represent the chain
itself), a list of all deepest leaves from which we random each time we want
to get the longest branch in chain, and a queue of pending blocks that have
to be added to chain.
we manage 2 threads more than the main one: a deamon thread that tries all
the time to add the new blocks to chain, and a thread that is responsible
to close the chain when close_chain is called.
To make sure only one init can be called and also the init is the first to be
called, we use an ENUM that know the current state, destroyed,destroying or
initialized. this prevents also adding blocks to chain while destroying it.

we had a lock for every data structure, and we made the order of locking the
structures consistent through the whole program to prevent deadlocks.
In addition, we got a condition variable that announce the deamon that there
are new pending blocks that have to be added, and a lock for the
synchronization between the closing thread and "return to close" function.
another lock is used in order to let attach now stop the deamon thread
immediatly (without having to wait till deamon will open the pending lock so
"attach now" could take it. this way "attach now" can just actively block the
deamon till it ends its attachement).

ANSWERS:

1. That happend because 'add_block' is non-blocking. The number is affected by
the time take to calculate the hash.
If add_block will be blocking or hashing time will be 0 that will change.

2. We can mark every block with 'to_longest'. That way every block will add at
the bottom of the chain and there will be no 'sub-chain'.

3. If we wont block the chain while calculating the hash of a block with
'to_longest' flag, there is a chance that during the long hash calculation
other block added to the chain(one that start earlier, or in case attach_now
add it) or prune was called and delete the father and then we will need to
calculate it again. that scenerio can be every time up to infinity and the
block wont succeed to be addad.
```

2 Makefile

```makefile
CC=g++
RANLIB=ranlib

LIBSRC=blockchain.cpp
LIBOBJ=$(LIBSRC:.cpp=.o)


INCS=-I.
CFLAGS = -Wall -g -std=c++11 $(INCS)
CPPFLAGS = -Wall -g -std=c++11
LOADLIBES = -L./

COMPILE.cc=$(CC) $(CPPFLAGS) -c

OSMLIB = libblockchain.a
TARGETS = $(OSMLIB)

TAR=tar
TARFLAGS=-cvf
TARNAME=ex3.tar
TARSRCS=$(LIBSRC) Makefile README

all: $(TARGETS)

$(LIBOBJ): $(LIBSRC) $(LIBSRC:.cpp=.h) hash.h
    $(COMPILE.cc) $% $<

$(TARGETS): $(LIBOBJ)
    $(AR) $(ARFLAGS) $@ $^
    $(RANLIB) $@

clean:
    $(RM) $(TARGETS) $(OSMLIB) $(OBJ) $(LIBOBJ) *~ *core

depend:
    makedepend -- $(CFLAGS) -- $(SRC) $(LIBSRC)

tar:
    $(TAR) $(TARFLAGS) $(TARNAME) $(TARSRCS)
```

3 blockchain.cpp

```
//============================================================================
// Name        :
// Author      :
// Version     :
// Copyright   :
// Description :
//============================================================================

#include <iostream>
#include <deque>
#include <list>
#include <map>
#include <climits>
#include <pthread.h>
#include <string.h>
#include <algorithm>
#include <random>
#include <vector>
#include <unistd.h>
#include <exception>
#include "blockchain.h"
#include "hash.h"

using namespace std;

#define LOCK(x)     if(pthread_mutex_lock(&x) != 0){throw SystemError;}
#define UNLOCK(x) if(pthread_mutex_unlock(&x) != 0){throw SystemError;}
#define INIT_LOCK(x) if(pthread_mutex_init(&x,NULL) != 0){throw SystemError;}
#define DESTROY_LOCK(x) if(pthread_mutex_destroy(&x) != 0){throw SystemError;}



class SystemError: public exception
{
  virtual const char* what() const throw()
  {
    return "System error happened";
  }
} SystemError;


long chainSize = 0;
enum VALIDITY_MODE{destroyed,destroying,initialized}validity_mode = destroyed;

typedef struct Block
{
    int num_block;
    Block *father;
    char * hash;
    char* data;
    int dataLength;
    bool toLongest; //is "toLongest" was called on this block
    bool attached; //is it attached to chain
    int sonsCount;
    int depth;

    /**default constructor. */
    Block()
    {
        father = NULL;
        toLongest = false;
        attached = false;
        sonsCount = 0;
        num_block = 0;
        hash = NULL;
        data = NULL;
        dataLength = 0;
        depth = 0;
    }

    /**default destructor. */
    ~Block()
    {
        if (hash)
        {
            free(hash);
        }
        if (data)
        {
            free(data);
        }
    }

    bool operator==(const Block& a) const
    {
        return (num_block == a.num_block);
    }
    bool operator==(const int  a) const
    {
        return (num_block == a);
    }
}Block;

// map with all blocks, including the information in their fields
map<int, Block*> blocks;
deque<Block*> pendingList; //list of all blocks waiting to be added to chain
list<Block*> blockchain; //all blocks that are currently leaves in the chain
vector<Block*> deepestLeaves; //all leaves with the maximal depth in chain.

int longestDepth = 0;

//data structures' mutexes
pthread_mutex_t blocksLock;
pthread_mutex_t pendingListLock;
pthread_mutex_t blockchainLock;
pthread_mutex_t deepestLeavesLock;
pthread_mutex_t attachLock;

//a mutex identifies that we are in the middle of the closing process
pthread_mutex_t closingChainLock = PTHREAD_MUTEX_INITIALIZER;

//says weather there are some pending threads
pthread_cond_t pendingThreads;

pthread_t deamonThread;
pthread_t closingThread;

bool deamon_needed; //indicates weather we still need the deamon thread to run

/**
 * when wanting to add a block to the longest chain, here we random one of
 * the deepest leaves to be the the longest chain.
 */
Block* random_longest(void)
{
    Block* return_val;
    try
    {
        LOCK(deepestLeavesLock)
        int randomNum = rand() % deepestLeaves.size();
        return_val = deepestLeaves.at(randomNum);
        UNLOCK(deepestLeavesLock);
    }
    catch(exception& e)
    {
        throw;
    }
    return return_val;
}

/**
 *  assign's "block"'s father to be the longest chain.
 */
void assign_father(Block * block)
{
    try
    {
    LOCK(blocksLock);//till we finish updating all father/sons relations coherently.
    if (block->father != NULL)
    {
        block->father->sonsCount--;
    }
    block->father = random_longest();
    block->father->sonsCount++;
    block->depth = block->father->depth + 1;
    UNLOCK(blocksLock)
    }
    catch (exception& e)
    {
        throw;
    }
}

/**
 * hashes block's data.
 */
void calc_hash(Block *block)
{
    int nonce = generate_nonce(block->num_block, block->father->num_block);
    block->hash = generate_hash(block->data, block->dataLength, nonce);
}


/**
 * adding "current" to the chain and updates data structures accordingly.
 * this function is not chain-blocking! it must be called within a safe zone where
 * the chain is locked and can be touched while execution.
 */
void add_to_list(Block * current)
{
    try
    {
        if (current->toLongest || current->father == NULL)
        {
            assign_father(current);
        }
        calc_hash(current);
        if(find(blockchain.begin(), blockchain.end(), current->father) !=\
                blockchain.end())
        {
            blockchain.remove(current->father);
            if (current->depth > longestDepth)
            {
                longestDepth = current->depth;
                LOCK(deepestLeavesLock)
                deepestLeaves.clear();
                UNLOCK(deepestLeavesLock)
            }
        }
        if (current->depth == longestDepth)
        {
            LOCK(deepestLeavesLock)
            deepestLeaves.push_back(current);
            UNLOCK(deepestLeavesLock)
        }
        blockchain.push_back(current);
        current->attached=true;
        chainSize++;
    }
    catch (exception& e)
    {
        throw;
    }
}


/*
 * return the first unused id or -1 if reaches to maximum threads
 */
int find_first_free_block_num()
{
    try
    {
        LOCK(blocksLock)
        map<int, Block*>::iterator it = blocks.begin();
        int i = 0;
        // find first block_num
        for ( ; it != blocks.end(); i++ , it++)
        {
            if (i != it->first)
            {
                UNLOCK(blocksLock)
                return i;
            }
        }
        UNLOCK(blocksLock)
        if (i != INT_MAX)
        {
            return i;
        }
    }
    catch (exception& e)
    {
        throw;
    }
    return -1;
}

/**the deamon thread who is responsible to add pending blocks into the block chain.*/
void* deamon (void*)
{
    while(deamon_needed)
    {
        try
        {
            LOCK(pendingListLock)
            while (pendingList.size() == 0)
            {
                if(pthread_cond_wait(&pendingThreads, &pendingListLock)!=0)
                    {
                    throw SystemError;
                    }
                if (validity_mode != initialized)
                {
                    UNLOCK(pendingListLock)
                    return NULL;
                }
            }
            Block * current = pendingList.front();
            pendingList.pop_front();
            UNLOCK(pendingListLock)

            LOCK(blockchainLock)
            add_to_list(current);
            UNLOCK(blockchainLock)

            LOCK(attachLock)
            UNLOCK(attachLock)
        }
        catch (exception& e)
        {
            exit (EXIT_FAILURE);
        }
    }
    return NULL;
}



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

int init_blockchain()
{
    if (validity_mode != destroyed)
    {
        cerr << "the chain is already initialized" << endl;
        return -1;
    }
    try
    {
        init_hash_generator();
        LOCK(closingChainLock)//verifies we are not in the middle of the closing process

        INIT_LOCK(blocksLock);
        INIT_LOCK(pendingListLock);
        INIT_LOCK(blockchainLock);
        INIT_LOCK(deepestLeavesLock);
        INIT_LOCK(attachLock);

        if(pthread_cond_init(&pendingThreads,NULL)!=0)
        {
            throw SystemError;
        }

        chainSize = 0;

        Block *genesis = new Block();
        genesis->num_block = 0;
        genesis->attached = true;
        genesis->depth = 0;
        longestDepth = 0;

        deepestLeaves.push_back(genesis);
        blocks.insert(pair<int, Block*>(genesis->num_block, genesis));
        blockchain.push_back(genesis);
        deamon_needed=true;


        if (pthread_create(&deamonThread, NULL, deamon, NULL)!=0)
        {
            throw SystemError;
        }
        validity_mode = initialized;
        UNLOCK(closingChainLock)//now this chain can be closed
    }
    catch (exception& e)
    {
        return -1;
    }
    return 0;
}


/*
 * 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
 * to the chain.
 * Furthermore, the father Block should be determined before this function returns. The father Block should be the last Block
 * 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 add_block(char *data , size_t length)
{
    if (validity_mode != initialized)
    {
        cerr << " calling add_block while destroying the chain" << endl;
        return -1;
    }
    try
    {
        Block *newBlock = new Block();
        newBlock->num_block = find_first_free_block_num();
        newBlock->data = (char*)malloc(length);
        if(newBlock->data == nullptr)
        {
            throw SystemError;
        }
        memcpy(newBlock->data, data, length);
        newBlock->dataLength = length;

        assign_father(newBlock);
        LOCK(pendingListLock)
        LOCK(blocksLock)

        blocks.insert(pair<int, Block*>(newBlock->num_block, newBlock));
        UNLOCK(blocksLock)
        pendingList.push_back(newBlock);
        UNLOCK(pendingListLock)
        if(pthread_cond_signal(&pendingThreads) != 0)
        {
            throw SystemError;
        }
        return newBlock->num_block;
    }
    catch (exception& e)
    {
        return -1;
    }
}
```

```
/*
 * DESCRIPTION: Without blocking, enforce the policy that this block_num should be attached to the longest chain at the time
 * the Block. For clearance, this is opposed to the original add_block that adds the Block to the longest chain during the ti
 * 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 c
 * already attached return 1.
 */
int to_longest(int block_num)
{
    if (validity_mode != initialized)
    {
        cerr << "calling to_longest while destroying the chain" << endl;
        return -1;
    }
    try
    {
        LOCK(blocksLock)
        if (blocks.count(block_num) == 0)
        {
            UNLOCK(blocksLock)
            return -2;
        }
        if (blocks.at(block_num)->attached)
        {
            UNLOCK(blocksLock)
            return 1;
        }
        blocks.at(block_num)->toLongest = true;
        UNLOCK(blocksLock)
    }
    catch (exception& e)
    {
        return -1;
    }
    return 0;
}

/*
 * DESCRIPTION: This function blocks all other Block attachments, until block_num is added to the chain. The block_num is the
 * 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 attach_now(int block_num)
{
    if (validity_mode != initialized)
    {
        cerr << "calling a func while destroying the chain" << endl;
        return -1;
    }
    try
    {
        LOCK(attachLock)
        LOCK(blockchainLock)
        LOCK(pendingListLock)
        LOCK(blocksLock)
        if (blocks.count(block_num) == 0)
        {
            UNLOCK(blocksLock)
            UNLOCK(pendingListLock)
            UNLOCK(blockchainLock)
            return -2;
        }
        deque<Block*>::iterator it = find(pendingList.begin(), \
                                pendingList.end(), blocks.at(block_num));
        UNLOCK(blocksLock)
        if(it != pendingList.end())
        {
            pendingList.erase(it);
            add_to_list(*it);
        }
        UNLOCK(pendingListLock)
        UNLOCK(blockchainLock)
        UNLOCK(attachLock)
    }
    catch (exception& e)
    {
        return -1;
    }
    return 0;
}


/*
 * 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 was_added(int block_num)
{
    try
    {
        if (validity_mode == destroyed)
        {
            cerr << "calling was_added when no chain is existed" << endl;
            return -1;
        }
        LOCK(blocksLock);
        if (blocks.count(block_num) == 0)
        {
            UNLOCK(blocksLock)
            return -2;
        }
        if (blocks.at(block_num)->attached)
        {
            UNLOCK(blocksLock)
            return 1;
        }
        else
        {
            UNLOCK(blocksLock)
            return 0;
        }
    }
    catch (exception& e)
    {
        return -1;
    }
}

/*
 * 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 th
 * RETURN VALUE: On success, the number of Blocks, otherwise -1.
 */
int chain_size()
{
    if (validity_mode == destroyed)
    {
        cerr << "calling chain_size when no chain is existed" << endl;
        return -1;
    }
    return chainSize;
}

```

```

/*
 * 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.
*/
int prune_chain()
{
    if (validity_mode != initialized)
    {
        cerr << "calling a func while destroying the chain" << endl;
        return -1;
    }
    try
    {
        LOCK(blockchainLock)

        Block *block;
        Block *temp;
        Block* currentLongestChain = random_longest();

        //emptying deepest leaves but the longest chosen chain
        LOCK(deepestLeavesLock)
        deepestLeaves.clear();
        deepestLeaves.push_back(currentLongestChain);
        UNLOCK(deepestLeavesLock)

        while (blockchain.size() != 1)
        {
            block = blockchain.back();
            blockchain.pop_back();

            //skip the longest chain
            if (block == currentLongestChain)
            {
                blockchain.push_front(block);
                block = blockchain.back();
                blockchain.pop_back();
            }

            while (!block->sonsCount)
            {
                temp = block;
                block = block->father;
                LOCK(blocksLock)
                blocks.erase(temp->num_block);
                delete(temp);
                block->sonsCount--;
                UNLOCK(blocksLock)
            }
        }
        UNLOCK(blockchainLock)
    }
    catch (exception& e)
    {
        return -1;
    }
    return 0;
}

/**a thread that is doing the closing process of a chain*/
void* closing(void*)
{
    try
    {
        LOCK(blockchainLock)
        LOCK(pendingListLock)
        for (deque<Block*>::iterator it = pendingList.begin();\
                it != pendingList.end(); it++)
        {
            calc_hash(*it);
            cout << (*it)->hash << endl;
        }

        close_hash_generator();
        LOCK(blocksLock)
        validity_mode = destroyed;
        for (map<int, Block*>::iterator it = blocks.begin(); \
                it != blocks.end(); it++)
        {
            delete (it->second);
        }
        blocks.clear();
        pendingList.clear();
        blockchain.clear();
        LOCK(deepestLeavesLock)
        deepestLeaves.clear();
        UNLOCK(deepestLeavesLock)


        UNLOCK(blocksLock)
        UNLOCK(pendingListLock)
        UNLOCK(blockchainLock)


        if(pthread_cond_destroy(&pendingThreads) != 0)
        {
            throw SystemError;
        }

        DESTROY_LOCK(blocksLock);
        DESTROY_LOCK(pendingListLock);
        DESTROY_LOCK(blockchainLock);
        DESTROY_LOCK(deepestLeavesLock);
        DESTROY_LOCK(attachLock);

        UNLOCK(closingChainLock);//the only undestroyed allowed mutex.
    }
    catch (exception& e)
    {
        exit (EXIT_FAILURE);
    }

    return NULL;
}



/*
 * DESCRIPTION: Close the recent blockchain and reset the system, so that it is possible to call init_blockchain again. Non-b
 * 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.
 */
void close_chain()
{
    if (validity_mode != initialized)
    {
        cerr << "the chain is already being closed!" << endl;
        return;
    }
    try
    {
        LOCK(closingChainLock);
        validity_mode = destroying;
        deamon_needed = false;//finishes the deamon's running.
        pthread_cond_signal(&pendingThreads);
        if (pthread_create(&closingThread, NULL, closing, NULL) != 0)
        {
            throw SystemError;
        }
    }
    catch (exception& x)
    {
        exit(EXIT_FAILURE);
    }
}

/*
 * 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.
 */
int return_on_close()
{
    if(validity_mode == initialized)
    {
        cerr << "calling return_on_close without close_chain" << endl;
        return -2;
    }
    try
    {
        LOCK(closingChainLock);
        pthread_join(deamonThread, NULL);
        pthread_join(closingThread, NULL);
        UNLOCK(closingChainLock);
    }
    catch (exception& x)
    {
        return -1;
    }
    return 0;
}
```