Contents

1 README

```
 README
 roeia1, maayanimi
 Roei Avrahami (304986714), Ma'ayan Mizrahi (304901259)
 EX: 3

 FILES:
 blockChain.h - A multi threaded blockchain database manager.
 blockChian.h - implementation of blockChain.h.
 Block.h - A class that represent a single block.
 Block.cpp - implementation of block.h
 Makefile - generating the library libblockchain.a.
 README - this file.

 REMARKS:
 using principles seen in class and demos, we implemented the desired interface.
 for that purpose we used a class Block, containing all relevant information and
 data for a single block.

 database choices:
 we saved the blocks that the deamon add to the blockChian in two lists that
 used as queues: toAdd, toAddNow. The optional father are kept in the list: "childList",
  and the all the blocks that where added to the blockChain are kept in a map. as indices we use the threads' id's.
 We used pthread_mutex to each of these database, so When ever a thread try to access one of these database we
 lock the database and after it finish using it, we unlock the database. This helps us avoid race conditions.

 Library functions:
 init_blockchain - In this function we initialize all the database that we use. we use the pthread_mutex initMutex
                   to lock the code that initialize the boolean flag "initialize" that indicate if the library
                   was initialize to avoid unexpected behavior of conditional race.
 add_block -In this function we create a new block that contain the data before we did hashing to it, and push
            this block to the "toAdd" list (using by the demon). we lock the toAdd list with pthread_mutex toAddMutex
            to avoid unexpected behavior behavior of conditional race.
 to_longes- This function change the flag of the given block's id to true so the deamon will know to attach it to
            the longest chain.we don't using any mutex here.
 attach_now- This function remove the given block (if exist) to "toAddNow" list of block that have an higher priority
             in the deamon function. we lock the two lists "toAdd" and "toAddNow" with the pthread_mutexes toAddMutex,
              and toAddNowMutex to avoid unexpected behavior of conditional race.
 was_added- This function check if the given block was added. It checks if this block is in the blockMap, toAdd list
            or in toAddNow. While we are checking this information we lock this databases with the pthread_mutexes
            toAddMutex, toAddNowMutex and blockMapMutex to avoid unexpected behavior,
 chain_size- return the chain size. we don't using any mutex here.
 prune_chain- This function prune the chain: it choose randomly longest chain, and save all the id in this chin in
              the set "toSave".we lock the blockMap with pthread_mutexes blockMapMutex and delete all the blocks that
              are not in to save.
 close_chain- change boolean flag "close" to true - than the deamon stop adding blocks and start to close the library.

 private function:
 daemonFunc(void*) - This function is the thread that running in the background and add the block to the blockChain after
                     it do hash to their data. While it use the databases: toAd, toAddNow, and BlockMap it lock them with
                     their mutexes. when "closing" called the deamon start classing the library (calling closing func).
 closing()- closing the library. using mutexs on the database when it closing it and delete its elements.
int findLongestFather - Return the ID of one of the available father (the longest chains...) randomly. Using
                        pthread_mutexes childListMutex while it find it to avoid conditional race.
void deleteChild() -Called by "prune" delete all the child that where deleted in prune function. Using pthread_mutexes
                    childListMutex while it find it to avoid conditional race.
```

```
ANSWERS:
1.
The father of a block is decide before it really attach to the chain, and the deamon is the one that attach it. Because of
that, a block can be the father of a few blocks,(non of those "sons" added yet, so the chain's size didn't change).
The period time that take to add a new block to chain (doing hasing to its data) is the main parameter that effects the
numberof multi-poiners.

2.
We can use this approach: each new block that will be create will be also call in to_longest function.
In that case each block will be attach to the longest father in running time.
This approach will enable only one long chain, thus relieving us from the boring pruning job.

3.
We don't deal the case when we are in the middle of hasing the data of a block (befor the deamon
attach this block to blockChain), and at the same time prune is called. In that case we may delete
 the father of this block.
```

2 Block.h

```
/*
 * Block.h
 * This class represent one block in the cahinBlock.
 *  Created on: Apr 30, 2015
 */

#ifndef BLOCK_H_
#define BLOCK_H_
#include <cstdio>
class Block{
private:
    int _father;
    int _id;
    size_t _length;
    int _depth;
    char* _data;
    char* _hash;
    bool _toLongest;
public:

    // constructor to Block.
    Block(int father, int id, size_t length, int depth = 0, char* data = 0);

    //Distructor to block.
    ~Block();

    //Return the data of the block.
    char* getData() const;

    //Set the hash data of the block.
    void setHash(char* hash);

    //Return the block's Id.
    int getId() const;

    //Return the block's deapth.
    int getDepth() const;

    //Return the stat length of the block.
    size_t getLength() const;

    //Return the block's father's id.
    int getFather() const;

    //Set the block's father's id.
    void setFather(int father);

    //Return true if this block was called by to_londest, false otherwise/
    bool isToLongest() const;

    //Set the block's toLongest field.
    void setToLongest(bool toLongest);
};



#endif /* BLOCK_H_ */
```

3 Block.cpp

```cpp
/*
 * Block.cpp
 *
 *  Created on: Apr 30, 2015
 *      Author: roeia1
 */
#include "Block.h"
#include <stdlib.h>
#include <cstring>

/**
 * constructor to Block.
 */
Block::Block(int father,int id, size_t length, int depth, char* data):_father(father), _id(id),  _length(length),
_depth(depth), _hash(0), _toLongest(false)
{
    _data = (char*)malloc(length * sizeof(char));
    strcpy(_data, data);
}

/**
 * Destructor to block.
 */
Block::~Block()
{
    free (_data);
    free (_hash);
}

/**
 * Return the data of the block.
 */
char* Block::getData() const
{
    return _data;
}

/**
 * Set the hash data of the block.
 */
void Block::setHash(char* hash)
{
    _hash = hash;
}

/**
 * Return the block's Id.
 */
int Block::getId() const
{
    return _id;
}

/**
 * Return the block's deapth.
 */
int Block::getDepth() const
{
    return _depth;
}

/**
 * Return the stat length of the block.
 */
size_t Block::getLength() const
{
    return _length;
}

/**
 * Return the block's father's id.
 */
int Block::getFather() const
{
        return _father;
}

/**
 * Return true if this block was called by to_londest, false otherwise.
 */
bool Block::isToLongest() const
{
    return _toLongest;
}

/**
 * Set the block's toLongest field.
 */
void Block::setToLongest(bool toLongest)
{
    _toLongest = toLongest;
}

/**
 * Set the block's father's id.
 */
void Block::setFather(int father)
{
    _father = father;
}
```

4 Makefile

```
all: libblockchain.a

blockchain.o: blockchain.cpp blockchain.h hash.h Block.h
    g++ -Wall -Wextra -Wvla -std=c++11 -c blockchain.cpp

libblockchain.a: Block.o  blockchain.o
    ar rcs libblockchain.a Block.o blockchain.o

Block.o: Block.h Block.cpp
    g++ -Wall -Wextra -Wvla -std=c++11 -c Block.cpp

tar:
    tar -cvf ex3.tar Block.h Block.cpp blockchain.cpp  README Makefile

clean:
    rm -f blockchain.o Block.o  libblockchain.o libblockchain.a


.PHONY: clean tar
```

5 blockchain.cpp

```cpp
//============================================================================
// Name        : ex3.cpp
// Author      :
// Version     :
// Copyright   : Your copyright notice
// Description : Hello World in C++, Ansi-style
//============================================================================

#include <queue>
#include <iostream>
#include <pthread.h>
#include <climits>
#include <map>
#include <list>
#include <set>
#include "blockchain.h"
#include "Block.h"
#include "hash.h"
using namespace std;

#define ERROR -1
#define FAILURE -1
#define NOT_EXIST -2
#define SUCCES 0
#define ALLREADY_ATTACHED 0
#define ERROR_EXIT_STATUS -1
#define ALLREADY_EXIST 1
#define TRUE 1
#define FALSE 0

map<int, Block*> blockMap;
list<Block*> toAdd;
list<Block*> toAddNow;
list<Block*> childList;
bool closed;
bool initialized = false;
int numOfBlockes;
int availableID;
Block* currentDaemonBlock;
set<int> availableIDs;
pthread_t daemon;
pthread_mutex_t initMutexThread = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mapMutexThread;
pthread_mutex_t availbleIDMutexThread;
pthread_mutex_t toAddMutexThread;
pthread_mutex_t toAddNowMutexThread;
pthread_mutex_t childMutexThread;
pthread_mutex_t somethingAddedMutexThread;
pthread_cond_t condThread;

void* daemonFunc(void*);
void closing(int nonce);
int findLongestFather();
void deleteChild(int id);

/*
 * 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 init_blockchain()
{
    pthread_mutex_lock(&initMutexThread);
    if (initialized)
    {
        return -1;
    }
    else
    {
        initialized = true;
    }
    pthread_mutex_unlock(&initMutexThread);
    pthread_mutex_init(&mapMutexThread,NULL);
    pthread_mutex_init(&availbleIDMutexThread,NULL);
    pthread_mutex_init(&toAddMutexThread,NULL);
    pthread_mutex_init(&toAddNowMutexThread,NULL);
    pthread_mutex_init(&childMutexThread,NULL);
    currentDaemonBlock = NULL;
    closed = false;
    Block* genesis = new Block(-1, 0, 0);
    blockMap[0] = genesis;
    numOfBlockes = 0;
    childList.push_back(genesis);
    availableID = 1;
    int res;
    res = pthread_create(&daemon,NULL,daemonFunc,NULL);

    if (res != 0)
    {
        return ERROR;
    }
    init_hash_generator();
    return SUCCES;
}

/*
 * 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 add_block(char *data , size_t length)
{
    //Error because we did not do init first.
    if (!initialized || closed || (availableID > INT_MAX && availableIDs.empty()))
    {
        return ERROR;
    }
    pthread_mutex_lock(&availbleIDMutexThread);
    int newID;
    if (!availableIDs.empty())
    {
        newID = *(availableIDs.begin());
        availableIDs.erase(availableIDs.begin());
    }
    else
    {
        newID = availableID;
        availableID++;
    }
    pthread_mutex_unlock(&availbleIDMutexThread);
    int fatherID = findLongestFather();
    pthread_mutex_lock(&toAddMutexThread);
    toAdd.push_back(new Block(fatherID, newID, length,blockMap[fatherID]->getDepth() + 1 ,data));
    pthread_cond_signal(&condThread);
    pthread_mutex_unlock(&toAddMutexThread);
    return newID;
}

/*
 * 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 to_longest(int block_num)
{
    //Error because we did not do init first, or if the block_num is the genesis.
    if(!initialized || closed)
    {
        return ERROR;
    }

    //block_num is already attached.
    if(blockMap.find(block_num) != blockMap.end())
    {
        return 1;
    }
    pthread_mutex_lock(&toAddMutexThread);
    //block_num is in the waiting-list
    for (list<Block*>::iterator it = toAdd.begin(); it != toAdd.end(); ++it)
    {
        if ((*it)->getId() == block_num)
        {
            (*it)->setToLongest(true);
            pthread_mutex_unlock(&toAddMutexThread);
            return 0;
        }
    }
    pthread_mutex_unlock(&toAddMutexThread);
    return NOT_EXIST; // the block_num doesn't exist.
}


/*
 * DESCRIPTION: This function blocks all other Block attachments, until block_num is added to the chain.
 *     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)
{
    //Error because we did not do init first, or if the block_num is the genesis.
    if(!initialized || closed)
    {
        return ERROR;
    }

    //block_num is already attached.
    if(blockMap.find(block_num) != blockMap.end() || currentDaemonBlock->getId() == block_num)
    {
        return ALLREADY_ATTACHED;
    }

    pthread_mutex_lock(&toAddMutexThread);
    pthread_mutex_lock(&toAddNowMutexThread);
    //block_num is in the waiting-list, is priority he will attached immediately.
    for (list<Block*>::iterator it = toAdd.begin(); it != toAdd.end(); ++it)
    {
        if ((*it)->getId() == block_num)
        {
            Block* blockToAttach = *it;
            toAdd.erase(it);
            toAddNow.push_back(blockToAttach);
            pthread_mutex_unlock(&toAddMutexThread);
            pthread_mutex_unlock(&toAddNowMutexThread);
            return SUCCES;
        }
    }
    pthread_mutex_unlock(&toAddMutexThread);
    pthread_mutex_unlock(&toAddNowMutexThread);
    return NOT_EXIST; // the block_num doesn't exist.
}

/*
 * 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)
{
    //Error because we did not do init first, or if the block_num is the genesis.
    if (!initialized || closed)
    {
        return ERROR;
    }

    //block_num was added.
    if(blockMap.find(block_num) != blockMap.end())
    {
        return TRUE;
    }

    //block_num was not added yet.
    for (list<Block*>::iterator it = toAddNow.begin(); it != toAddNow.end(); ++it)
    {
        if ((*it)->getId() == block_num)
        {
            return FALSE;
        }
    }
    for (list<Block*>::iterator it = toAdd.begin(); it != toAdd.end(); ++it)
    {
        if ((*it)->getId() == block_num)
        {
            return FALSE;
        }
    }
    return NOT_EXIST; // the block_num doesn't exist.
}


/*
 * 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 chain_size()
{
    //Error because we did not do init first.
    if (!initialized)
    {
        return ERROR;
    }
    return numOfBlockes;
}


/*
 * 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 (!initialized || closed)
    {
        return ERROR;
    }
    pthread_mutex_lock(&mapMutexThread);
    int longestID = findLongestFather();
    set<int> toSave;

    //update the set toSave that will save all the blocks of the chain that we will NOT prune.
    while(longestID != -1)
    {
        toSave.insert(longestID);
        longestID = blockMap[longestID]->getFather();
    }

    //prune the chain
    for (map<int, Block*>::iterator it = blockMap.begin(); it != blockMap.end();)
    {
        if(toSave.find(it->first) == toSave.end())
        {
            deleteChild(it->first);
            delete (it->second);
            it->second = NULL;
            blockMap.erase(it++);
        }
        else
        {
            ++it;
        }
    }
    pthread_mutex_unlock(&mapMutexThread);
    return SUCCES;
}

/*
 * 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.
 */
void close_chain()
{
    closed = true;
}

/*
 * 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 (!initialized)
    {
        return SUCCES;
    }
    if (!closed)
    {
        return NOT_EXIST;
    }
    void* retval;
    int res;
    res = pthread_join(daemon,&retval);
    if (res != 0)
    {
        return ERROR;
    }

    initialized = false;

    return SUCCES;
}


/*
 * This functuion is the deamon, its main propose is to handle all the background things: attach new blocks
 * to the blockChain, and when "close_chain" called - to close the library.
 */
void* daemonFunc(void*)
{
    int nonce;

    //Adding Blocks to the chain while "close_chain" did not called.
    while(!closed)
    {
        //wait until new block is added to the waiting list.
        if (toAdd.empty() && toAddNow.empty())
        {
            pthread_mutex_lock(&somethingAddedMutexThread);

            pthread_cond_wait(&condThread, &somethingAddedMutexThread);

            pthread_mutex_unlock(&somethingAddedMutexThread);
        }
        pthread_mutex_lock(&toAddMutexThread);
        pthread_mutex_lock(&toAddNowMutexThread);
        if (!toAddNow.empty())
        {
            currentDaemonBlock = toAddNow.front();
            toAddNow.pop_front();
        }
        else
        {
            currentDaemonBlock = toAdd.front();
            toAdd.pop_front();
        }
        pthread_mutex_unlock(&toAddMutexThread);
        pthread_mutex_unlock(&toAddNowMutexThread);
        char* hashedData;
        bool addFlag = true;//this flag helps us
        nonce = generate_nonce(currentDaemonBlock->getId(), currentDaemonBlock->getFather());
        hashedData = generate_hash(currentDaemonBlock->getData(), currentDaemonBlock->getLength(), nonce);
        currentDaemonBlock->setHash(hashedData);
        pthread_mutex_lock(&mapMutexThread);
        // Checking if the father exists
        map<int, Block*>::iterator it = blockMap.find(currentDaemonBlock->getFather());
        if (it == blockMap.end())
        {
            currentDaemonBlock->setFather(findLongestFather());
            addFlag = false;
        }

        // Checking if to longest
        else if (currentDaemonBlock->isToLongest())
        {
            currentDaemonBlock->setToLongest(false);
            int newFather = findLongestFather();
            // If the father is different
            if (blockMap[newFather]->getDepth() != blockMap[currentDaemonBlock->getFather()]->getDepth())
            {
                currentDaemonBlock->setFather(newFather);
                addFlag = false;
            }
        }
        //add the block to the chain.
        if (addFlag)
        {
            blockMap[currentDaemonBlock->getId()] = currentDaemonBlock;
            numOfBlockes++;
            pthread_mutex_lock(&childMutexThread);
            for (list<Block*>::iterator it = childList.begin(); it != childList.end(); ++it)
            {

                if ((*it)->getId() == currentDaemonBlock->getFather())
                {
                    childList.erase(it);
                    break;
                }
            }
            childList.push_back(currentDaemonBlock);
            pthread_mutex_unlock(&childMutexThread);
        }
        //add the block to toAddNow list if the block father was update by calling to_longest" func.
        else
        {
            pthread_mutex_lock(&toAddMutexThread);
            toAddNow.push_front(currentDaemonBlock);
            pthread_mutex_unlock(&toAddMutexThread);
        }
        pthread_mutex_unlock(&mapMutexThread);

    }
    closing(nonce);
    pthread_exit(NULL);
}


/**
 * this func is calling by the deamon for closing.
 */
void closing(int nonce)
{
    //free all the elements in toAddNow
    pthread_mutex_lock(&toAddNowMutexThread);
    for (list<Block*>::iterator it = toAddNow.begin(); it != toAddNow.end();)
    {
        nonce = generate_nonce((*it)->getId(), (*it)->getFather());
        char* hash = generate_hash((*it)->getData(), (*it)->getLength(), nonce);
        cout << hash << endl;//printing the hash value.
        free (hash);
        delete (*it);
        toAddNow.erase(it++);
    }
    pthread_mutex_unlock(&toAddNowMutexThread);

    //free all the elements in toAdd
    pthread_mutex_lock(&toAddMutexThread);
    for (list<Block*>::iterator it = toAdd.begin(); it != toAdd.end();)
    {
        nonce = generate_nonce((*it)->getId(), (*it)->getFather());
        char* hash = generate_hash((*it)->getData(), (*it)->getLength(), nonce);
        cout << hash << endl;//printing the hash value.
        free (hash);
        delete (*it);
        toAdd.erase(it++);
    }
    pthread_mutex_unlock(&toAddMutexThread);

    //free all the element in blockMap
    pthread_mutex_lock(&mapMutexThread);
    for (map<int, Block*>::iterator it = blockMap.begin(); it != blockMap.end();)
    {
        delete (it->second);
        blockMap.erase(it++);
    }
    pthread_mutex_unlock(&mapMutexThread);

    //destroy the child list.
    pthread_mutex_lock(&childMutexThread);
    childList.clear();
    pthread_mutex_unlock(&childMutexThread);

    //destroy the availbleId structure.
    pthread_mutex_lock(&availbleIDMutexThread);
    availableIDs.clear();
    pthread_mutex_unlock(&availbleIDMutexThread);

    close_hash_generator();
    pthread_mutex_destroy(&mapMutexThread);
    pthread_mutex_destroy(&toAddMutexThread);
    pthread_mutex_destroy(&toAddNowMutexThread);
    pthread_mutex_destroy(&childMutexThread);
    pthread_mutex_destroy(&availbleIDMutexThread);
}


/**
 * Return the ID of one of the available father (the longest chains...) randomly.
 */
int findLongestFather()
{
    int maxDepth = 0;
    int maxDepthCounter = 0;
    pthread_mutex_lock(&childMutexThread);
    //find the max deapth and the number of the available fathers in this deapth.
    for (list<Block*>::iterator it = childList.begin(); it != childList.end(); ++it)
    {
        if ((*it)->getDepth() == maxDepth)
        {
            maxDepthCounter++;
        }
        if ((*it)->getDepth() > maxDepth)
        {
            maxDepth = (*it)->getDepth();
            maxDepthCounter = 1;
        }
    }

    //choose randomly one of the fathers.
    int randNum = rand() % maxDepthCounter + 1;
    list<Block*>::iterator it = childList.begin();
    int counter = 0;
    while (counter < randNum)
    {
        if ((*it)->getDepth() == maxDepth)
```

```cpp
        {
            ++counter;
        }
        ++it;
    }
    it = --it;
    pthread_mutex_unlock(&childMutexThread);
    return (*it)->getId();
}


/**
 * This function update the child list if we did prune to the blockChain.
 */
void deleteChild(int id)
{
    pthread_mutex_lock(&childMutexThread);
    for (list<Block*>::iterator it = childList.begin(); it != childList.end();)
    {
        if ((*it)->getId() == id)
        {
            childList.erase(it);
            break;
        }
        it++;
    }
    pthread_mutex_unlock(&childMutexThread);
}
```