Contents

1 README

```
1  akiva_s,roigreenberg
2  Akiva Sygal(305277220), Roi Greenberg(305571234)
3  Ex: 3
4
5  FILES:
6  README
7  blockChain.cpp -- a file that implements the given blockChain.h
8  makefile
9
10 REMARKS:
11 we used the next DASTs: one map that holds the information about every block,
12 a list of all leaves in the tree-chain (which is enough to represent the chain
13 itself), a list of all deepest leaves from which we random each time we want
14 to get the longest branch in chain, and a queue of pending blocks that have
15 to be added to chain.
16 we manage 2 threads more than the main one: a deamon thread that tries all
17 the time to add the new blocks to chain, and a thread that is responsible
18 to close the chain when close_chain is called.
19 To make sure only one init can be called and also the init is the first to be
20 called, we use an ENUM that know the current state, destroyed,destroying or
21 initialized. this prevents also adding blocks to chain while destroying it.
22
23 we had a lock for every data structure, and we made the order of locking the
24 structures consistent through the whole program to prevent deadlocks.
25 In addition, we got a condition variable that announce the deamon that there
26 are new pending blocks that have to be added, and a lock for the
27 synchronization between the closing thread and "return to close" function.
28 another lock is used in order to let attach now stop the deamon thread
29 immediatly (without having to wait till deamon will open the pending lock so
30 "attach now" could take it. this way "attach now" can just actively block the
31 deamon till it ends its attachement).
32
33 ANSWERS:
34
35 1. That happend because 'add_block' is non-blocking. The number is affected by
36 the time take to calculate the hash.
37 If add_block will be blocking or hashing time will be 0 that will change.
38
39 2. We can mark every block with 'to_longest'. That way every block will add at
40 the bottom of the chain and there will be no 'sub-chain'.
41
42 3. If we wont block the chain while calculating the hash of a block with
43 'to_longest' flag, there is a chance that during the long hash calculation
44 other block added to the chain(one that start earlier, or in case attach_now
45 add it) or prune was called and delete the father and then we will need to
46 calculate it again. that scenerio can be every time up to infinity and the
47 block wont succeed to be addad.
```

2 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

```
1  //============================================================================
2  // Name        :
3  // Author      :
4  // Version     :
5  // Copyright   :
6  // Description :
7  //============================================================================
8
9  #include <iostream>
10 #include <deque>
11 #include <list>
12 #include <map>
13 #include <climits>
14 #include <pthread.h>
15 #include <string.h>
16 #include <algorithm>
17 #include <random>
18 #include <vector>
19 #include <unistd.h>
20 #include <exception>
21 #include "blockchain.h"
22 #include "hash.h"
23
24 using namespace std;
25
26 #define LOCK(x)     if(pthread_mutex_lock(&x) != 0){throw SystemError;}
27 #define UNLOCK(x) if(pthread_mutex_unlock(&x) != 0){throw SystemError;}
28 #define INIT_LOCK(x) if(pthread_mutex_init(&x,NULL) != 0){throw SystemError;}
29 #define DESTROY_LOCK(x) if(pthread_mutex_destroy(&x) != 0){throw SystemError;}
30
31
32
33 class SystemError: public exception
34 {
35   virtual const char* what() const throw()
36   {
37     return "System error happened";
38   }
39 } SystemError;
40
41
42 long chainSize = 0;
43 enum VALIDITY_MODE{destroyed,destroying,initialized}validity_mode = destroyed;
44
45 typedef struct Block
46 {
47     int num_block;
48     Block *father;
49     char * hash;
50     char* data;
51     int dataLength;
52     bool toLongest; //is "toLongest" was called on this block
53     bool attached; //is it attached to chain
54     int sonsCount;
55     int depth;
56
57     /**default constructor. */
58     Block()
59     {
```

```
60          father = NULL;
61          toLongest = false;
62          attached = false;
63          sonsCount = 0;
64          num_block = 0;
65          hash = NULL;
66          data = NULL;
67          dataLength = 0;
68          depth = 0;
69      }
70
71      /**default destructor. */
72      ~Block()
73      {
74          if (hash)
75          {
76              free(hash);
77          }
78          if (data)
79          {
80              free(data);
81          }
82      }
83
84      bool operator==(const Block& a) const
85      {
86          return (num_block == a.num_block);
87      }
88      bool operator==(const int  a) const
89      {
90          return (num_block == a);
91      }
92  }Block;
93
94  // map with all blocks, including the information in their fields
95  map<int, Block*> blocks;
96  deque<Block*> pendingList; //list of all blocks waiting to be added to chain
97  list<Block*> blockchain; //all blocks that are currently leaves in the chain
98  vector<Block*> deepestLeaves; //all leaves with the maximal depth in chain.
99
100 int longestDepth = 0;
101
102 //data structures' mutexes
103 pthread_mutex_t blocksLock;
104 pthread_mutex_t pendingListLock;
105 pthread_mutex_t blockchainLock;
106 pthread_mutex_t deepestLeavesLock;
107 pthread_mutex_t attachLock;
108
109 //a mutex identifies that we are in the middle of the closing process
110 pthread_mutex_t closingChainLock = PTHREAD_MUTEX_INITIALIZER;
111
112 //says weather there are some pending threads
113 pthread_cond_t pendingThreads;
114
115 pthread_t deamonThread;
116 pthread_t closingThread;
117
118 bool deamon_needed; //indicates weather we still need the deamon thread to run
119
120 /**
121  * when wanting to add a block to the longest chain, here we random one of
122  * the deepest leaves to be the the longest chain.
123  */
124 Block* random_longest(void)
125 {
126     Block* return_val;
127     try
```

```
128     {
129         LOCK(deepestLeavesLock)
130         int randomNum = rand() % deepestLeaves.size();
131         return_val = deepestLeaves.at(randomNum);
132         UNLOCK(deepestLeavesLock);
133     }
134     catch(exception& e)
135     {
136         throw;
137     }
138     return return_val;
139 }
140 
141 /**
142  *  assign's "block"'s father to be the longest chain.
143  */
144 void assign_father(Block * block)
145 {
146     try
147     {
148     LOCK(blocksLock);//till we finish updating all father/sons relations coherently.
149     if (block->father != NULL)
150     {
151         block->father->sonsCount--;
152     }
153     block->father = random_longest();
154     block->father->sonsCount++;
155     block->depth = block->father->depth + 1;
156     UNLOCK(blocksLock)
157     }
158     catch (exception& e)
159     {
160         throw;
161     }
162 }
163 
164 /**
165  * hashes block's data.
166  */
167 void calc_hash(Block *block)
168 {
169     int nonce = generate_nonce(block->num_block, block->father->num_block);
170     block->hash = generate_hash(block->data, block->dataLength, nonce);
171 }
172 
173 
174 /**
175  * adding "current" to the chain and updates data structures accordingly.
176  * this function is not chain-blocking! it must be called within a safe zone where
177  * the chain is locked and can be touched while execution.
178  */
179 void add_to_list(Block * current)
180 {
181     try
182     {
183         if (current->toLongest || current->father == NULL)
184         {
185             assign_father(current);
186         }
187         calc_hash(current);
188         if(find(blockchain.begin(), blockchain.end(), current->father) !=\
189                 blockchain.end())
190         {
191             blockchain.remove(current->father);
192             if (current->depth > longestDepth)
193             {
194                 longestDepth = current->depth;
195                 LOCK(deepestLeavesLock)
```

```
196                 deepestLeaves.clear();
197                 UNLOCK(deepestLeavesLock)
198             }
199         }
200         if (current->depth == longestDepth)
201         {
202             LOCK(deepestLeavesLock)
203             deepestLeaves.push_back(current);
204             UNLOCK(deepestLeavesLock)
205         }
206         blockchain.push_back(current);
207         current->attached=true;
208         chainSize++;
209     }
210     catch (exception& e)
211     {
212         throw;
213     }
214 }
215
216
217 /*
218  * return the first unused id or -1 if reaches to maximum threads
219  */
220 int find_first_free_block_num()
221 {
222     try
223     {
224         LOCK(blocksLock)
225         map<int, Block*>::iterator it = blocks.begin();
226         int i = 0;
227         // find first block_num
228         for ( ; it != blocks.end(); i++ , it++)
229         {
230             if (i != it->first)
231             {
232                 UNLOCK(blocksLock)
233                 return i;
234             }
235         }
236         UNLOCK(blocksLock)
237         if (i != INT_MAX)
238         {
239             return i;
240         }
241     }
242     catch (exception& e)
243     {
244         throw;
245     }
246     return -1;
247 }
248
249 /**the deamon thread who is responsible to add pending blocks into the block chain.*/
250 void* deamon (void*)
251 {
252     while(deamon_needed)
253     {
254         try
255         {
256             LOCK(pendingListLock)
257             while (pendingList.size() == 0)
258             {
259                 if(pthread_cond_wait(&pendingThreads, &pendingListLock)!=0)
260                     {
261                     throw SystemError;
262                     }
263                 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);
```

```
332         deamon_needed=true;
333
334
335         if (pthread_create(&deamonThread, NULL, deamon, NULL)!=0)
336         {
337             throw SystemError;
338         }
339         validity_mode = initialized;
340         UNLOCK(closingChainLock)//now this chain can be closed
341     }
342     catch (exception& e)
343     {
344         return -1;
345     }
346     return 0;
347 }
348
349
350 /*
351  * DESCRIPTION: Ultimately, the function adds the hash of the data to the Block chain.
352  * Since this is a non-blocking package, your implemented method should return as soon as possible, even before the Block was
353  * to the chain.
354  * Furthermore, the father Block should be determined before this function returns. The father Block should be the last Block
355  * longest chain (arbitrary longest chain if there is more than one).
356  * Notice that once this call returns, the original data may be freed by the caller.
357  * RETURN VALUE: On success, the function returns the lowest available block_num (> 0),
358  * which is assigned from now on to this individual piece of data.
359  * On failure, -1 will be returned.
360  */
361 int add_block(char *data , size_t length)
362 {
363     if (validity_mode != initialized)
364     {
365         cerr << " calling add_block while destroying the chain" << endl;
366         return -1;
367     }
368     try
369     {
370         Block *newBlock = new Block();
371         newBlock->num_block = find_first_free_block_num();
372         newBlock->data = (char*)malloc(length);
373         if(newBlock->data == nullptr)
374         {
375             throw SystemError;
376         }
377         memcpy(newBlock->data, data, length);
378         newBlock->dataLength = length;
379
380         assign_father(newBlock);
381         LOCK(pendingListLock)
382         LOCK(blocksLock)
383
384         blocks.insert(pair<int, Block*>(newBlock->num_block, newBlock));
385         UNLOCK(blocksLock)
386         pendingList.push_back(newBlock);
387         UNLOCK(pendingListLock)
388         if(pthread_cond_signal(&pendingThreads) != 0)
389         {
390             throw SystemError;
391         }
392         return newBlock->num_block;
393     }
394     catch (exception& e)
395     {
396         return -1;
397     }
398 }
399
```

```
400 /*
401  * DESCRIPTION: Without blocking, enforce the policy that this block_num should be attached to the longest chain at the time
402  * the Block. For clearance, this is opposed to the original add_block that adds the Block to the longest chain during the ti
403  * was called.
404  * The block_num is the assigned value that was previously returned by add_block.
405  * 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
406  * already attached return 1.
407  */
408 int to_longest(int block_num)
409 {
410     if (validity_mode != initialized)
411     {
412         cerr << "calling to_longest while destroying the chain" << endl;
413         return -1;
414     }
415     try
416     {
417         LOCK(blocksLock)
418         if (blocks.count(block_num) == 0)
419         {
420             UNLOCK(blocksLock)
421             return -2;
422         }
423         if (blocks.at(block_num)->attached)
424         {
425             UNLOCK(blocksLock)
426             return 1;
427         }
428         blocks.at(block_num)->toLongest = true;
429         UNLOCK(blocksLock)
430     }
431     catch (exception& e)
432     {
433         return -1;
434     }
435     return 0;
436 }
437
438 /*
439  * DESCRIPTION: This function blocks all other Block attachments, until block_num is added to the chain. The block_num is the
440  * that was previously returned by add_block.
441  * RETURN VALUE: If block_num doesn't exist, return -2;
442  * In case of other errors, return -1; In case of success or if it is already attached return 0.
443  */
444 int attach_now(int block_num)
445 {
446     if (validity_mode != initialized)
447     {
448         cerr << "calling a func while destroying the chain" << endl;
449         return -1;
450     }
451     try
452     {
453         LOCK(attachLock)
454         LOCK(blockchainLock)
455         LOCK(pendingListLock)
456         LOCK(blocksLock)
457         if (blocks.count(block_num) == 0)
458         {
459             UNLOCK(blocksLock)
460             UNLOCK(pendingListLock)
461             UNLOCK(blockchainLock)
462             return -2;
463         }
464         deque<Block*>::iterator it = find(pendingList.begin(), \
465                                   pendingList.end(), blocks.at(block_num));
466         UNLOCK(blocksLock)
467         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;
}
```

```
536
537 /*
538  * DESCRIPTION: Search throughout the tree for sub-chains that are not the longest chain,
539  *      detach them from the tree, free the blocks, and reuse the block_nums.
540  * RETURN VALUE: On success 0, otherwise -1.
541 */
542 int prune_chain()
543 {
544     if (validity_mode != initialized)
545     {
546         cerr << "calling a func while destroying the chain" << endl;
547         return -1;
548     }
549     try
550     {
551         LOCK(blockchainLock)
552
553         Block *block;
554         Block *temp;
555         Block* currentLongestChain = random_longest();
556
557         //emptying deepest leaves but the longest chosen chain
558         LOCK(deepestLeavesLock)
559         deepestLeaves.clear();
560         deepestLeaves.push_back(currentLongestChain);
561         UNLOCK(deepestLeavesLock)
562
563         while (blockchain.size() != 1)
564         {
565             block = blockchain.back();
566             blockchain.pop_back();
567
568             //skip the longest chain
569             if (block == currentLongestChain)
570             {
571                 blockchain.push_front(block);
572                 block = blockchain.back();
573                 blockchain.pop_back();
574             }
575
576             while (!block->sonsCount)
577             {
578                 temp = block;
579                 block = block->father;
580                 LOCK(blocksLock)
581                 blocks.erase(temp->num_block);
582                 delete(temp);
583                 block->sonsCount--;
584                 UNLOCK(blocksLock)
585             }
586         }
587         UNLOCK(blockchainLock)
588     }
589     catch (exception& e)
590     {
591         return -1;
592     }
593     return 0;
594 }
595
596 /**a thread that is doing the closing process of a chain*/
597 void* closing(void*)
598 {
599     try
600     {
601         LOCK(blockchainLock)
602         LOCK(pendingListLock)
603         for (deque<Block*>::iterator it = pendingList.begin();\
```

```
604                 it != pendingList.end(); it++)
605         {
606             calc_hash(*it);
607             cout << (*it)->hash << endl;
608         }
609 
610         close_hash_generator();
611         LOCK(blocksLock)
612         validity_mode = destroyed;
613         for (map<int, Block*>::iterator it = blocks.begin(); \
614                 it != blocks.end(); it++)
615         {
616             delete (it->second);
617         }
618         blocks.clear();
619         pendingList.clear();
620         blockchain.clear();
621         LOCK(deepestLeavesLock)
622         deepestLeaves.clear();
623         UNLOCK(deepestLeavesLock)
624 
625 
626         UNLOCK(blocksLock)
627         UNLOCK(pendingListLock)
628         UNLOCK(blockchainLock)
629 
630 
631         if(pthread_cond_destroy(&pendingThreads) != 0)
632         {
633             throw SystemError;
634         }
635 
636         DESTROY_LOCK(blocksLock);
637         DESTROY_LOCK(pendingListLock);
638         DESTROY_LOCK(blockchainLock);
639         DESTROY_LOCK(deepestLeavesLock);
640         DESTROY_LOCK(attachLock);
641 
642         UNLOCK(closingChainLock);//the only undestroyed allowed mutex.
643     }
644     catch (exception& e)
645     {
646         exit (EXIT_FAILURE);
647     }
648 
649     return NULL;
650 }
651 
652 
653 
654 /*
655  * DESCRIPTION: Close the recent blockchain and reset the system, so that it is possible to call init_blockchain again. Non-b
656  * All pending Blocks should be hashed and printed to terminal (stdout).
657  * Calls to library methods which try to alter the state of the Blockchain are prohibited while closing the Blockchain.
658  * e.g.: Calling chain_size() is ok, a call to prune_chain() should fail.
659  * In case of a system error, the function should cause the process to exit.
660  */
661 void close_chain()
662 {
663     if (validity_mode != initialized)
664     {
665         cerr << "the chain is already being closed!" << endl;
666         return;
667     }
668     try
669     {
670         LOCK(closingChainLock);
671         validity_mode = destroying;
```

```
672             deamon_needed = false;//finishes the deamon's running.
673             pthread_cond_signal(&pendingThreads);
674             if (pthread_create(&closingThread, NULL, closing, NULL) != 0)
675             {
676                 throw SystemError;
677             }
678         }
679         catch (exception& x)
680         {
681             exit(EXIT_FAILURE);
682         }
683     }
684
685     /*
686      * DESCRIPTION: The function blocks and waits for close_chain to finish.
687      * RETURN VALUE: If closing was successful, it returns 0.
688      * If close_chain was not called it should return -2. In case of other error, it should return -1.
689      */
690     int return_on_close()
691     {
692         if(validity_mode == initialized)
693         {
694             cerr << "calling return_on_close without close_chain" << endl;
695             return -2;
696         }
697         try
698         {
699             LOCK(closingChainLock);
700             pthread_join(deamonThread, NULL);
701             pthread_join(closingThread, NULL);
702             UNLOCK(closingChainLock);
703         }
704         catch (exception& x)
705         {
706             return -1;
707         }
708         return 0;
709     }
```