# Contents

1	README	2
2	Block.h	4
3	Block.cpp	5
4	Makefile	7
5	blockchain.cpp	8

#### 1 README

57 58

```
README
1
2
     roeia1, maayanimi
     Roei Avrahami (304986714), Ma'ayan Mizrahi (304901259)
3
     EX: 3
4
     FILES:
6
     blockChain.h - A multi threaded blockchain database manager.
8
     blockChian.h - implementation of blockChain.h.
     Block.h - A class that represent a single block.
9
10
     Block.cpp - implementation of block.h
     Makefile - generating the library libblockchain.a.
11
     README - this file.
12
13
     REMARKS:
14
     using principles seen in class and demos, we implemented the desired interface.
15
16
     for that purpose we used a class Block, containing all relevant information and
     data for a single block.
17
18
     database choices:
19
     we saved the blocks that the deamon add to the blockChian in two lists that
20
21
     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.
22
23
     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.
24
25
     Library functions:
26
27
     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
28
                        was initialize to avoid unexpected behavior of conditional race.
29
     add_block -In this function we create a new block that contain the data before we did hashing to it, and push
30
                this block to the "toAdd" list (using by the demon). we lock the toAdd list with pthread_mutex toAddMutex
31
32
                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
33
34
                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
35
                 in the deamon function. we lock the two lists "toAdd" and "toAddNow" with the pthread_mutexes toAddMutex,
36
37
                  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
38
39
                or in toAddNow. While we are checking this information we lock this databases with the pthread_mutexes
40
                toAddMutex, toAddNowMutex and blockMapMutex to avoid unexpected behavior,
     chain_size- return the chain size. we don't using any mutex here.
41
42
     prune_chain- This function prune the chain: it choose randomly longest chain, and save all the id in this chin in
43
                  the set "toSave".we lock the blockMap with pthread_mutexes blockMapMutex and delete all the blocks that
                  are not in to save.
44
     close_chain- change boolean flag "close" to true - than the deamon stop adding blocks and start to close the library.
45
46
     private function:
47
     daemonFunc(void*) - This function is the thread that running in the background and add the block to the blockChain after
48
                          it do hash to their data. While it use the databases: toAd, toAddNow, and BlockMap it lock them with
49
50
                          their mutexes. when "closing" called the deamon start classing the library (calling closing func).
51
     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
52
                            pthread\_mutexes\ childListMutex\ while\ it\ find\ it\ to\ avoid\ conditional\ race.
53
    void deleteChild() -Called by "prune" delete all the child that where deleted in prune function. Using pthread_mutexes
54
55
                         childListMutex while it find it to avoid conditional race.
56
```

```
60 ANSWERS:
61 1.
62 The fath
```

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).

64 The period time that take to add a new block to chain (doing hasing to its data) is the main parameter that effects the 65 number of multi-poiners.

66 67

68 We can use this approach: each new block that will be create will be also call in to\_longest function.

69 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.

71 72 **3** 

70

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

75 the father of this block.

#### 2 Block.h

```
1
     * Block.h
     * This class represent one block in the cahinBlock.
3
     * Created on: Apr 30, 2015
4
    #ifndef BLOCK_H_
8
    #define BLOCK_H_
    #include <cstdio>
9
10
    class Block{
    private:
11
        int _father;
12
13
        int _id;
        size_t _length;
14
15
        int _depth;
        char* _data;
16
        char* _hash;
17
18
        bool _toLongest;
    public:
19
20
21
         // constructor to Block.
        Block(int father, int id, size_t length, int depth = 0, char* data = 0);
22
23
24
         //Distructor to block.
         ~Block();
25
26
27
        //Return the data of the block.
        char* getData() const;
28
29
        //Set the hash data of the block.
30
        void setHash(char* hash);
31
         //Return the block's Id.
33
34
        int getId() const;
35
         //Return the block's deapth.
36
37
         int getDepth() const;
38
        //Return the stat length of the block.
39
40
        size_t getLength() const;
41
42
        //Return the block's father's id.
        int getFather() const;
43
44
        //Set the block's father's id.
45
        void setFather(int father);
46
47
48
         //Return true if this block was called by to_londest, false otherwise/
        bool isToLongest() const;
49
50
         //Set the block's toLongest field.
51
         void setToLongest(bool toLongest);
52
    };
53
54
55
    #endif /* BLOCK_H_ */
```

# 3 Block.cpp

```
2
     * Block.cpp
3
     * Created on: Apr 30, 2015
          Author: roeia1
5
6
   #include "Block.h"
   #include <stdlib.h>
8
    #include <cstring>
9
10
11
12
     * constructor to Block.
13
    Block::Block(int father,int id, size_t length, int depth, char* data):_father(father), _id(id), _length(length),
14
15
    _depth(depth), _hash(0), _toLongest(false)
16
17
        _data = (char*)malloc(length * sizeof(char));
18
        strcpy(_data, data);
    }
19
20
21
     * Destructor to block.
22
23
    Block::~Block()
24
25
        free (_data);
26
        free (_hash);
27
28
29
30
31
    * Return the data of the block.
32
33
   char* Block::getData() const
34
        return _data;
35
36
37
38
39
    * Set the hash data of the block.
40
    void Block::setHash(char* hash)
41
42
        _hash = hash;
43
44
45
46
47
     * Return the block's Id.
48
^{49}
    int Block::getId() const
50
        return _id;
51
52
53
54
    * Return the block's deapth.
56
57
    int Block::getDepth() const
58
        return _depth;
59
```

```
60 }
61
62
    * Return the stat length of the block.
63
64
    size_t Block::getLength() const
65
66
        return _length;
67
68
69
70
    * Return the block's father's id.
*/
71
72
    int Block::getFather() const
73
74
           return _father;
75
    }
76
77
78
    * Return true if this block was called by to_londest, false otherwise.
80
    bool Block::isToLongest() const
81
82
        return _toLongest;
83
    }
84
85
86
    * Set the block's toLongest field.
*/
87
88
    void Block::setToLongest(bool toLongest)
89
90
         _toLongest = toLongest;
91
    }
92
93
94
    * Set the block's father's id.
95
96
    void Block::setFather(int father)
97
98
        _father = father;
99
100
```

## 4 Makefile

```
all: libblockchain.a
1
    blockchain.o: blockchain.cpp blockchain.h hash.h Block.h
    g++ -Wall -Wextra -Wvla -std=c++11 -c blockchain.cpp
4
    libblockchain.a: Block.o blockchain.o
6
        ar rcs libblockchain.a Block.o blockchain.o
8
    Block.o: Block.h Block.cpp
9
        g++ -Wall -Wextra -Wvla -std=c++11 -c Block.cpp
10
11
12
         tar -cvf ex3.tar Block.h Block.cpp blockchain.cpp README Makefile
14
    clean:
15
16
       rm -f blockchain.o Block.o libblockchain.o libblockchain.a
17
18
    .PHONY: clean tar
19
```

## 5 blockchain.cpp

```
//=----
    // Name
                : ex3.cpp
2
    // Author
3
   // Version
   // Version :
// Copyright : Your copyright notice
    // Description : Hello World in C++, Ansi-style
9
   #include <queue>
   #include <iostream>
10
   #include <pthread.h>
11
12
    #include <climits>
13 #include <map>
14 #include <list>
    #include <set>
   #include "blockchain.h"
16
17 #include "Block.h"
    #include "hash.h"
18
19
   using namespace std;
   #define ERROR -1
21
   #define FAILURE -1
22
23 #define NOT_EXIST -2
   #define SUCCES 0
24
25
   #define ALLREADY_ATTACHED 0
26
   #define ERROR_EXIT_STATUS -1
   #define ALLREADY_EXIST 1
27
28
    #define TRUE 1
   #define FALSE 0
29
30
    map<int, Block*> blockMap;
31
   list<Block*> toAdd;
32
33 list<Block*> toAddNow;
    list<Block*> childList;
34
    bool closed:
35
   bool initialized = false;
    int numOfBlockes;
37
38
   int availableID:
   Block* currentDaemonBlock;
   set<int> availableIDs;
40
41
    pthread_t daemon;
   pthread_mutex_t initMutexThread = PTHREAD_MUTEX_INITIALIZER;
42
    pthread_mutex_t mapMutexThread;
43
44
    pthread_mutex_t availbleIDMutexThread;
    pthread_mutex_t toAddMutexThread;
   pthread_mutex_t toAddNowMutexThread;
    pthread_mutex_t childMutexThread;
    pthread_mutex_t somethingAddedMutexThread;
48
49
    pthread_cond_t condThread;
50
    void* daemonFunc(void*);
51
52 void closing(int nonce);
    int findLongestFather();
53
    void deleteChild(int id);
54
56
    * DESCRIPTION: This function initiates the Block chain, and creates the genesis Block.
57
            The genesis Block does not hold any transaction data or hash.
58
            This function should be called prior to any other functions as a necessary precondition for their
59
```

```
60
             success (all other functions should return with an error otherwise).
 61
       * RETURN VALUE: On success O, otherwise -1.
 62
     int init_blockchain()
 63
 64
     {
          pthread_mutex_lock(&initMutexThread);
 65
          if (initialized)
 66
 67
          {
 68
              return -1;
 69
 70
         else
 71
          {
 72
              initialized = true;
         }
 73
 74
         pthread_mutex_unlock(&initMutexThread);
         pthread_mutex_init(&mapMutexThread,NULL);
 75
         {\tt pthread\_mutex\_init(\&availbleIDMutexThread,NULL);}
 76
         pthread_mutex_init(&toAddMutexThread,NULL);
 77
         pthread_mutex_init(&toAddNowMutexThread,NULL);
 78
         pthread_mutex_init(&childMutexThread,NULL);
 79
          currentDaemonBlock = NULL;
 80
          closed = false;
 81
          Block* genesis = new Block(-1, 0, 0);
 82
         blockMap[0] = genesis;
 83
 84
         numOfBlockes = 0;
 85
         childList.push_back(genesis);
         availableID = 1;
 86
 87
          int res;
         res = pthread_create(&daemon,NULL,daemonFunc,NULL);
 88
 89
 90
          if (res != 0)
 91
          ₹
              return ERROR;
 92
 93
         init_hash_generator();
 94
 95
          return SUCCES;
 96
     }
 97
 98
      * DESCRIPTION: Ultimately, the function adds the hash of the data to the Block chain.
 99
100
             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.
101
              Furthermore, the father Block should be determined before this function returns. The
102
103
              father Block should be the last Block of the current longest chain (arbitrary longest chain
              if there is more than one).
104
              Notice that once this call returns, the original data may be freed by the caller.
105
106
       * 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.
107
108
              On failure, -1 will be returned.
109
     int add_block(char *data , size_t length)
110
111
112
          //Error because we did not do init first.
          if (!initialized || closed || (availableID > INT_MAX && availableIDs.empty()))
113
114
              return ERROR;
115
116
         pthread_mutex_lock(&availbleIDMutexThread);
117
          int newID:
118
119
          if (!availableIDs.empty())
120
121
              newID = *(availableIDs.begin());
122
              availableIDs.erase(availableIDs.begin());
         }
123
124
         else
125
          {
              newID = availableID;
126
127
              availableID++;
```

```
128
         pthread_mutex_unlock(&availbleIDMutexThread);
129
          int fatherID = findLongestFather();
130
         pthread_mutex_lock(&toAddMutexThread);
131
          toAdd.push_back(new Block(fatherID, newID, length,blockMap[fatherID]->getDepth() + 1 ,data));
132
133
         pthread_cond_signal(&condThread);
134
         pthread_mutex_unlock(&toAddMutexThread);
         return newID:
135
136
     }
137
138
139
      * 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
140
              opposed to the original add_block that adds the Block to the longest chain during the time that add_block was called.
141
142
              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;
143
144
             In case block_num is already attached return 1.
145
     int to_longest(int block_num)
146
147
          //Error because we did not do init first, or if the block_num is the genesis.
148
149
          if(!initialized || closed)
150
          {
              return ERROR;
151
152
         }
153
          //block num is already attached.
154
155
          if(blockMap.find(block_num) != blockMap.end())
156
157
              return 1;
158
         pthread_mutex_lock(&toAddMutexThread);
159
160
          //block_num is in the waiting-list
          for (list<Block*>::iterator it = toAdd.begin(); it != toAdd.end(); ++it)
161
162
163
              if ((*it)->getId() == block_num)
164
165
                  (*it)->setToLongest(true);
                  pthread_mutex_unlock(&toAddMutexThread);
166
                  return 0:
167
              7
168
169
         pthread_mutex_unlock(&toAddMutexThread);
170
171
          return NOT_EXIST; // the block_num doesn't exist.
     }
172
173
174
175
176
      *\ \textit{DESCRIPTION: This function blocks all other Block attachments, until block\_num is added to the chain.}
177
             that was previously returned by add_block.
      * RETURN VALUE: If block_num doesn't exist, return -2;
178
              In case of other errors, return -1; In case of success or if it is already attached return 0.
179
180
181
     int attach_now(int block_num)
182
          //Error because we did not do init first, or if the block_num is the genesis.
183
184
         if(!initialized || closed)
185
          {
              return ERROR:
186
187
188
189
          //block_num is already attached.
          if(blockMap.find(block_num) != blockMap.end() || currentDaemonBlock->getId() == block_num)
190
          ₹
191
192
              return ALLREADY_ATTACHED;
193
194
195
         pthread_mutex_lock(&toAddMutexThread);
```

```
196
         pthread_mutex_lock(&toAddNowMutexThread);
197
          //block_num is in the waiting-list, is priority he will attached immediately.
          for (list<Block*>::iterator it = toAdd.begin(); it != toAdd.end(); ++it)
198
199
              if ((*it)->getId() == block_num)
200
201
                  Block* blockToAttach = *it;
202
                  toAdd.erase(it);
203
204
                  toAddNow.push_back(blockToAttach);
                  pthread_mutex_unlock(&toAddMutexThread);
205
                  pthread_mutex_unlock(&toAddNowMutexThread);
206
207
                  return SUCCES;
208
         }
209
210
         pthread_mutex_unlock(&toAddMutexThread);
         pthread_mutex_unlock(&toAddNowMutexThread);
211
212
          return NOT_EXIST; // the block_num doesn't exist.
213
     }
214
215
      * DESCRIPTION: Without blocking, check whether block_num was added to the chain.
216
            The block_num is the assigned value that was previously returned by add_block.
217
      * RETURN VALUE: 1 if true and 0 if false. If the block_num doesn't exist, return -2;
218
219
      * In case of other errors, return -1.
220
221
     int was_added(int block_num)
222
     {
          //Error because we did not do init first, or if the block_num is the genesis.
223
224
         if (!initialized || closed)
225
          {
226
              return ERROR;
227
228
229
          //block_num was added.
         if(blockMap.find(block_num) != blockMap.end())
230
231
232
              return TRUE;
         }
233
234
          //block num was not added uet.
235
          for (list<Block*>::iterator it = toAddNow.begin(); it != toAddNow.end(); ++it)
236
237
              if ((*it)->getId() == block_num)
238
239
                  return FALSE;
240
              }
241
242
         }
         for (list<Block*>::iterator it = toAdd.begin(); it != toAdd.end(); ++it)
243
244
              if ((*it)->getId() == block_num)
245
              {
246
247
                  return FALSE;
248
         }
249
         return NOT_EXIST; // the block_num\ doesn't\ exist.
250
     }
251
252
253
254
255
      * 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)
256
              again this function should return
257
              the new chain size.
258
      * RETURN VALUE: On success, the number of Blocks, otherwise -1.
259
260
261
     int chain_size()
262
     {
263
          //Error because we did not do init first.
```

```
264
         if (!initialized)
265
              return ERROR;
266
267
         }
         return numOfBlockes;
268
     }
269
270
271
272
      * DESCRIPTION: Search throughout the tree for sub-chains that are not the longest chain,
273
              detach them from the tree, free the blocks, and reuse the block_nums.
274
275
      * RETURN VALUE: On success O, otherwise -1.
276
277
     int prune_chain()
278
     {
          if (!initialized || closed)
279
280
              return ERROR;
281
282
283
         pthread_mutex_lock(&mapMutexThread);
          int longestID = findLongestFather();
284
          set<int> toSave;
285
286
         //update the set toSave that will save all the blocks of the chain that we will NOT prune.
287
288
         while(longestID != -1)
289
          {
              toSave.insert(longestID);
290
291
              longestID = blockMap[longestID]->getFather();
292
293
294
          //prune the chain
         for (map<int, Block*>::iterator it = blockMap.begin(); it != blockMap.end();)
295
296
297
              if(toSave.find(it->first) == toSave.end())
298
299
                  deleteChild(it->first);
300
                  delete (it->second);
                  it->second = NULL;
301
                  blockMap.erase(it++);
302
              }
303
304
              else
305
              {
                  ++it;
306
              }
307
308
          pthread_mutex_unlock(&mapMutexThread);
309
310
          return SUCCES;
     }
311
312
313
      * DESCRIPTION: Close the recent blockChain and reset the system, so that it is possible to call
314
315
              init\_blockchain again. Non-blocking. All pending Blocks should be hashed and printed to terminal (stdout).
316
              Calls to library methods which try to alter the state of the BlockChain are prohibited while closing the
317
              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.
318
319
320
     void close_chain()
321
          closed = true;
322
     }
323
324
325
      * DESCRIPTION: The function blocks and waits for close_chain to finish.
326
      * RETURN VALUE: If closing was successful, it returns 0.
327
              If close_chain was not called it should return -2. In case of other error, it should return -1.
328
329
330
     int return_on_close()
```

```
332
     {
          if (!initialized)
333
334
              return SUCCES;
335
          }
336
337
          if (!closed)
338
          {
              return NOT_EXIST;
339
340
         }
          void* retval;
341
342
          int res;
343
          res = pthread_join(daemon,&retval);
          if (res != 0)
344
345
346
              return ERROR;
347
348
          initialized = false;
349
350
351
          return SUCCES;
     }
352
353
354
355
      *\ \textit{This functuion is the deamon, its main propose is to handle all the background things: attach new blocks
356
      * to the blockChain, and when "close_chain" called - to close the library.
357
358
359
     void* daemonFunc(void*)
360
361
          int nonce;
362
          //Adding Blocks to the chain while "close_chain" did not called.
363
364
          while(!closed)
365
              //wait until new block is added to the waiting list.
366
367
              if (toAdd.empty() && toAddNow.empty())
368
                  pthread_mutex_lock(&somethingAddedMutexThread);
369
370
                  pthread_cond_wait(&condThread, &somethingAddedMutexThread);
371
372
373
                  pthread_mutex_unlock(&somethingAddedMutexThread);
              }
374
375
              pthread_mutex_lock(&toAddMutexThread);
              pthread_mutex_lock(&toAddNowMutexThread);
376
377
              if (!toAddNow.empty())
378
              {
                  currentDaemonBlock = toAddNow.front();
379
380
                  toAddNow.pop_front();
              }
381
              else
382
383
              {
384
                  currentDaemonBlock = toAdd.front();
385
                  toAdd.pop_front();
              }
386
              pthread_mutex_unlock(&toAddMutexThread);
387
388
              pthread_mutex_unlock(&toAddNowMutexThread);
389
              char* hashedData;
              bool addFlag = true;//this flag helps us
390
391
              nonce = generate_nonce(currentDaemonBlock->getId(), currentDaemonBlock->getFather());
              hashedData = generate_hash(currentDaemonBlock->getData(), currentDaemonBlock->getLength(), nonce);
392
393
              currentDaemonBlock->setHash(hashedData);
              pthread_mutex_lock(&mapMutexThread);
394
              // Checking if the father exists
395
              map<int, Block*>::iterator it = blockMap.find(currentDaemonBlock->getFather());
396
397
              if (it == blockMap.end())
              {
398
399
                  currentDaemonBlock->setFather(findLongestFather());
```

```
400
                  addFlag = false;
              }
401
402
              // Checking if to longest
403
              else if (currentDaemonBlock->isToLongest())
404
405
                  currentDaemonBlock->setToLongest(false);
406
                  int newFather = findLongestFather();
407
408
                  // If the father is different
                  if (blockMap[newFather]->getDepth() != blockMap[currentDaemonBlock->getFather()]->getDepth())
409
410
411
                      currentDaemonBlock->setFather(newFather);
                      addFlag = false;
412
                  }
413
414
              }
              //add the block to the chain.
415
416
              if (addFlag)
417
                  blockMap[currentDaemonBlock->getId()] = currentDaemonBlock;
418
                  numOfBlockes++;
419
                  pthread_mutex_lock(&childMutexThread);
420
                  for (list<Block*>::iterator it = childList.begin(); it != childList.end(); ++it)
421
422
423
424
                      if ((*it)->getId() == currentDaemonBlock->getFather())
425
                      {
                           childList.erase(it):
426
427
                           break;
                      }
428
429
                  }
430
                  childList.push_back(currentDaemonBlock);
                  pthread_mutex_unlock(&childMutexThread);
431
              }
432
433
              //add the block to toAddNow list if the block father was update by calling to_longest" func.
              else
434
435
              {
436
                  pthread_mutex_lock(&toAddMutexThread);
437
                  toAddNow.push_front(currentDaemonBlock);
                  pthread_mutex_unlock(&toAddMutexThread);
438
439
440
              pthread_mutex_unlock(&mapMutexThread);
441
442
443
          closing(nonce);
          pthread_exit(NULL);
444
     }
445
446
447
448
449
      * this func is calling by the deamon for closing.
450
451
     void closing(int nonce)
452
453
          //free all the elements in toAddNow
          pthread_mutex_lock(&toAddNowMutexThread);
454
          for (list<Block*>::iterator it = toAddNow.begin(); it != toAddNow.end();)
455
456
              nonce = generate_nonce((*it)->getId(), (*it)->getFather());
457
              char* hash = generate_hash((*it)->getData(), (*it)->getLength(), nonce);
458
459
              \verb"cout" << hash << endl; //printing the hash value.
460
              free (hash);
461
              delete (*it);
              toAddNow.erase(it++);
462
463
          {\tt pthread\_mutex\_unlock(\&toAddNowMutexThread);}
464
465
          //free all the elements in toAdd
466
467
          pthread_mutex_lock(&toAddMutexThread);
```

```
468
          for (list<Block*>::iterator it = toAdd.begin(); it != toAdd.end();)
469
              nonce = generate_nonce((*it)->getId(), (*it)->getFather());
470
              char* hash = generate_hash((*it)->getData(), (*it)->getLength(), nonce);
471
              cout << hash << endl;//printing the hash value.</pre>
472
473
              free (hash);
474
              delete (*it);
              toAdd.erase(it++);
475
476
          }
         pthread_mutex_unlock(&toAddMutexThread);
477
478
479
          //free all the element in blockMap
          pthread_mutex_lock(&mapMutexThread);
480
          for (map<int, Block*>::iterator it = blockMap.begin(); it != blockMap.end();)
481
482
              delete (it->second):
483
484
              blockMap.erase(it++);
485
          pthread_mutex_unlock(&mapMutexThread);
486
487
          //destroy the child list.
488
          pthread_mutex_lock(&childMutexThread);
489
490
          childList.clear();
          pthread_mutex_unlock(&childMutexThread);
491
492
493
          //destroy the availbleId structure.
          pthread_mutex_lock(&availbleIDMutexThread);
494
495
          availableIDs.clear();
          pthread_mutex_unlock(&availbleIDMutexThread);
496
497
498
          close_hash_generator();
          pthread_mutex_destroy(&mapMutexThread);
499
500
          pthread_mutex_destroy(&toAddMutexThread);
          pthread_mutex_destroy(&toAddNowMutexThread);
501
          pthread_mutex_destroy(&childMutexThread);
502
503
          pthread_mutex_destroy(&availbleIDMutexThread);
     }
504
505
506
507
      * Return the ID of one of the available father (the longest chains...) randomly.
508
509
     int findLongestFather()
510
511
          int maxDepth = 0;
512
513
          int maxDepthCounter = 0;
514
          pthread_mutex_lock(&childMutexThread);
          //find the max deapth and the number of the available fathers in this deapth.
515
516
          for (list<Block*>::iterator it = childList.begin(); it != childList.end(); ++it)
517
              if ((*it)->getDepth() == maxDepth)
518
519
              {
520
                  maxDepthCounter++;
521
              }
              if ((*it)->getDepth() > maxDepth)
522
              {
523
524
                  maxDepth = (*it)->getDepth();
525
                  maxDepthCounter = 1;
              }
526
527
         }
528
529
          //choose randomly one of the fathers.
          int randNum = rand() % maxDepthCounter + 1;
530
          list<Block*>::iterator it = childList.begin();
531
532
          int counter = 0;
533
          while (counter < randNum)
534
535
              if ((*it)->getDepth() == maxDepth)
```

```
536
              {
537
                  ++counter;
              }
538
539
              ++it;
         }
540
         it = --it;
541
         pthread_mutex_unlock(&childMutexThread);
542
         return (*it)->getId();
543
     }
544
545
546
547
     * This function update the child list if we did prune to the blockChain.
548
549
550
     void deleteChild(int id)
551
         {\tt pthread\_mutex\_lock(\&childMutexThread);}
552
553
         for (list<Block*>::iterator it = childList.begin(); it != childList.end();)
554
              if ((*it)->getId() == id)
555
556
                  childList.erase(it);
557
558
                  break;
              }
559
              it++;
560
561
         pthread_mutex_unlock(&childMutexThread);
562
     }
563
```