

Contents

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

1 README

```
1  README
2  roeial, maayanimi
3  Roei Avrahami (304986714), Ma'ayan Mizrahi (304901259)
4  EX: 3
5
6  FILES:
7  blockChain.h - A multi threaded blockchain database manager.
8  blockChian.h - implementation of blockChain.h.
9  Block.h - A class that represent a single block.
10 Block.cpp - implementation of block.h
11 Makefile - generating the library libblockchain.a.
12 README - this file.
13
14 REMARKS:
15 using principles seen in class and demos, we implemented the desired interface.
16 for that purpose we used a class Block, containing all relevant information and
17 data for a single block.
18
19 database choices:
20 we saved the blocks that the deamon add to the blockChian in two lists that
21 used as queues: toAdd, toAddNow. The optional father are kept in the list: "childList",
22 and the all the blocks that where added to the blockChain are kept in a map. as indices we use the threads' id's.
23 We used pthread_mutex to each of these database, so When ever a thread try to access one of these database we
24 lock the database and after it finish using it, we unlock the database. This helps us avoid race conditions.
25
26 Library functions:
27 init_blockchain - In this function we initialize all the database that we use. we use the pthread_mutex initMutex
28 to lock the code that initialize the boolean flag "initialize" that indicate if the library
29 was initialize to avoid unexpected behavior of conditional race.
30 add_block -In this function we create a new block that contain the data before we did hashing to it, and push
31 this block to the "toAdd" list (using by the demon). we lock the toAdd list with pthread_mutex toAddMutex
32 to avoid unexpected behavior behavior of conditional race.
33 to_longes- This function change the flag of the given block's id to true so the deamon will know to attach it to
34 the longest chain.we don't using any mutex here.
35 attach_now- This function remove the given block (if exist) to "toAddNow" list of block that have an higher priority
36 in the deamon function. we lock the two lists "toAdd" and "toAddNow" with the pthread_mutexes toAddMutex,
37 and toAddNowMutex to avoid unexpected behavior of conditional race.
38 was_added- This function check if the given block was added. It checks if this block is in the blockMap, toAdd list
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,
41 chain_size- return the chain size. we don't using any mutex here.
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
44 are not in to save.
45 close_chain- change boolean flag "close" to true - than the deamon stop adding blocks and start to close the library.
46
47 private function:
48 daemonFunc(void*) - This function is the thread that running in the background and add the block to the blockChain after
49 it do hash to their data. While it use the databases: toAd, toAddNow, and BlockMap it lock them with
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.
52 int findLongestFather - Return the ID of one of the available father (the longest chains...) randomly. Using
53 pthread_mutexes childListMutex while it find it to avoid conditional race.
54 void deleteChild() -Called by "prune" delete all the child that where deleted in prune function. Using pthread_mutexes
55 childListMutex while it find it to avoid conditional race.
56
57
58
59
```

60 ANSWERS:
61 1.
62 The father of a block is decide before it really attach to the chain, and the daemon is the one that attach it. Because of
63 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 numberof multi-poiners.
66
67 2.
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.
70 This approach will enable only one long chain, thus relieving us from the boring pruning job.
71
72 3.
73 We don't deal the case when we are in the middle of hasing the data of a block (befor the daemon
74 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  /*
2  * Block.h
3  * This class represent one block in the cahinBlock.
4  * Created on: Apr 30, 2015
5  */
6
7  #ifndef BLOCK_H_
8  #define BLOCK_H_
9  #include <cstdio>
10 class Block{
11 private:
12     int _father;
13     int _id;
14     size_t _length;
15     int _depth;
16     char* _data;
17     char* _hash;
18     bool _toLongest;
19 public:
20
21     // constructor to Block.
22     Block(int father, int id, size_t length, int depth = 0, char* data = 0);
23
24     //Distructor to block.
25     ~Block();
26
27     //Return the data of the block.
28     char* getData() const;
29
30     //Set the hash data of the block.
31     void setHash(char* hash);
32
33     //Return the block's Id.
34     int getId() const;
35
36     //Return the block's deapth.
37     int getDepth() const;
38
39     //Return the stat length of the block.
40     size_t getLength() const;
41
42     //Return the block's father's id.
43     int getFather() const;
44
45     //Set the block's father's id.
46     void setFather(int father);
47
48     //Return true if this block was called by to_londest, false otherwise/
49     bool isToLongest() const;
50
51     //Set the block's toLongest field.
52     void setToLongest(bool toLongest);
53 };
54
55
56
57 #endif /* BLOCK_H_ */
```

3 Block.cpp

```
1  /*
2   * Block.cpp
3   *
4   * Created on: Apr 30, 2015
5   * Author: roeia1
6   */
7  #include "Block.h"
8  #include <stdlib.h>
9  #include <cstring>
10
11 /**
12  * constructor to Block.
13  */
14 Block::Block(int father, int id, size_t length, int depth, char* data):_father(father), _id(id), _length(length),
15 _depth(depth), _hash(0), _toLongest(false)
16 {
17     _data = (char*)malloc(length * sizeof(char));
18     strcpy(_data, data);
19 }
20
21 /**
22  * Destructor to block.
23  */
24 Block::~Block()
25 {
26     free (_data);
27     free (_hash);
28 }
29
30 /**
31  * Return the data of the block.
32  */
33 char* Block::getData() const
34 {
35     return _data;
36 }
37
38 /**
39  * Set the hash data of the block.
40  */
41 void Block::setHash(char* hash)
42 {
43     _hash = hash;
44 }
45
46 /**
47  * Return the block's Id.
48  */
49 int Block::getId() const
50 {
51     return _id;
52 }
53
54 /**
55  * Return the block's deapth.
56  */
57 int Block::getDepth() const
58 {
59     return _depth;
```

```

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

```

4 Makefile

```
1  all: libblockchain.a
2
3  blockchain.o: blockchain.cpp blockchain.h hash.h Block.h
4      g++ -Wall -Wextra -Wvla -std=c++11 -c blockchain.cpp
5
6  libblockchain.a: Block.o blockchain.o
7      ar rcs libblockchain.a Block.o blockchain.o
8
9  Block.o: Block.h Block.cpp
10     g++ -Wall -Wextra -Wvla -std=c++11 -c Block.cpp
11
12  tar:
13     tar -cvf ex3.tar Block.h Block.cpp blockchain.cpp README Makefile
14
15  clean:
16     rm -f blockchain.o Block.o libblockchain.o libblockchain.a
17
18
19  .PHONY: clean tar
```

5 blockchain.cpp

```
1  //=====
2  // Name      : ex3.cpp
3  // Author    :
4  // Version   :
5  // Copyright : Your copyright notice
6  // Description : Hello World in C++, Ansi-style
7  //=====
8
9  #include <queue>
10 #include <iostream>
11 #include <pthread.h>
12 #include <climits>
13 #include <map>
14 #include <list>
15 #include <set>
16 #include "blockchain.h"
17 #include "Block.h"
18 #include "hash.h"
19 using namespace std;
20
21 #define ERROR -1
22 #define FAILURE -1
23 #define NOT_EXIST -2
24 #define SUCCESS 0
25 #define ALLREADY_ATTACHED 0
26 #define ERROR_EXIT_STATUS -1
27 #define ALLREADY_EXIST 1
28 #define TRUE 1
29 #define FALSE 0
30
31 map<int, Block*> blockMap;
32 list<Block*> toAdd;
33 list<Block*> toAddNow;
34 list<Block*> childList;
35 bool closed;
36 bool initialized = false;
37 int numOfBlockes;
38 int availableID;
39 Block* currentDaemonBlock;
40 set<int> availableIDs;
41 pthread_t daemon;
42 pthread_mutex_t initMutexThread = PTHREAD_MUTEX_INITIALIZER;
43 pthread_mutex_t mapMutexThread;
44 pthread_mutex_t availbleIDMutexThread;
45 pthread_mutex_t toAddMutexThread;
46 pthread_mutex_t toAddNowMutexThread;
47 pthread_mutex_t childMutexThread;
48 pthread_mutex_t somethingAddedMutexThread;
49 pthread_cond_t condThread;
50
51 void* daemonFunc(void*);
52 void closing(int nonce);
53 int findLongestFather();
54 void deleteChild(int id);
55
56 /*
57  * DESCRIPTION: This function initiates the Block chain, and creates the genesis Block.
58  *             The genesis Block does not hold any transaction data or hash.
59  *             This function should be called prior to any other functions as a necessary precondition for their
```



```

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

```

```

128     }
129     pthread_mutex_unlock(&availableIDMutexThread);
130     int fatherID = findLongestFather();
131     pthread_mutex_lock(&toAddMutexThread);
132     toAdd.push_back(new Block(fatherID, newID, length, blockMap[fatherID]->getDepth() + 1 ,data));
133     pthread_cond_signal(&condThread);
134     pthread_mutex_unlock(&toAddMutexThread);
135     return newID;
136 }
137
138 /*
139  * DESCRIPTION: Without blocking, enforce the policy that this block_num should be attached
140  *              to the longest chain at the time of attachment of the Block. For clearance, this is
141  *              opposed to the original add_block that adds the Block to the longest chain during the time that add_block was called.
142  *              The block_num is the assigned value that was previously returned by add_block.
143  * RETURN VALUE: If block_num doesn't exist, return -2; In case of other errors, return -1; In case of success return 0;
144  *              In case block_num is already attached return 1.
145  */
146 int to_longest(int block_num)
147 {
148     //Error because we did not do init first, or if the block_num is the genesis.
149     if(!initialized || closed)
150     {
151         return ERROR;
152     }
153
154     //block_num is already attached.
155     if(blockMap.find(block_num) != blockMap.end())
156     {
157         return 1;
158     }
159     pthread_mutex_lock(&toAddMutexThread);
160     //block_num is in the waiting-list
161     for (list<Block*>::iterator it = toAdd.begin(); it != toAdd.end(); ++it)
162     {
163         if ((*it)->getId() == block_num)
164         {
165             (*it)->setToLongest(true);
166             pthread_mutex_unlock(&toAddMutexThread);
167             return 0;
168         }
169     }
170     pthread_mutex_unlock(&toAddMutexThread);
171     return NOT_EXIST; // the block_num doesn't exist.
172 }
173
174
175 /*
176  * DESCRIPTION: This function blocks all other Block attachments, until block_num is added to the chain.
177  *              that was previously returned by add_block.
178  * RETURN VALUE: If block_num doesn't exist, return -2;
179  *              In case of other errors, return -1; In case of success or if it is already attached return 0.
180  */
181 int attach_now(int block_num)
182 {
183     //Error because we did not do init first, or if the block_num is the genesis.
184     if(!initialized || closed)
185     {
186         return ERROR;
187     }
188
189     //block_num is already attached.
190     if(blockMap.find(block_num) != blockMap.end() || currentDaemonBlock->getId() == block_num)
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.
198     for (list<Block*>::iterator it = toAdd.begin(); it != toAdd.end(); ++it)
199     {
200         if ((*it)->getId() == block_num)
201         {
202             Block* blockToAttach = *it;
203             toAdd.erase(it);
204             toAddNow.push_back(blockToAttach);
205             pthread_mutex_unlock(&toAddMutexThread);
206             pthread_mutex_unlock(&toAddNowMutexThread);
207             return SUCCES;
208         }
209     }
210     pthread_mutex_unlock(&toAddMutexThread);
211     pthread_mutex_unlock(&toAddNowMutexThread);
212     return NOT_EXIST; // the block_num doesn't exist.
213 }
214
215 /*
216 * DESCRIPTION: Without blocking, check whether block_num was added to the chain.
217 * The block_num is the assigned value that was previously returned by add_block.
218 * RETURN VALUE: 1 if true and 0 if false. If the block_num doesn't exist, return -2;
219 * In case of other errors, return -1.
220 */
221 int was_added(int block_num)
222 {
223     //Error because we did not do init first, or if the block_num is the genesis.
224     if (!initialized || closed)
225     {
226         return ERROR;
227     }
228
229     //block_num was added.
230     if(blockMap.find(block_num) != blockMap.end())
231     {
232         return TRUE;
233     }
234
235     //block_num was not added yet.
236     for (list<Block*>::iterator it = toAddNow.begin(); it != toAddNow.end(); ++it)
237     {
238         if ((*it)->getId() == block_num)
239         {
240             return FALSE;
241         }
242     }
243     for (list<Block*>::iterator it = toAdd.begin(); it != toAdd.end(); ++it)
244     {
245         if ((*it)->getId() == block_num)
246         {
247             return FALSE;
248         }
249     }
250     return NOT_EXIST; // the block_num doesn't exist.
251 }
252
253 /*
254 * DESCRIPTION: Return how many Blocks were attached to the chain since init_blockchain.
255 * 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 int chain_size()
261 {
262     //Error because we did not do init first.
263

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```