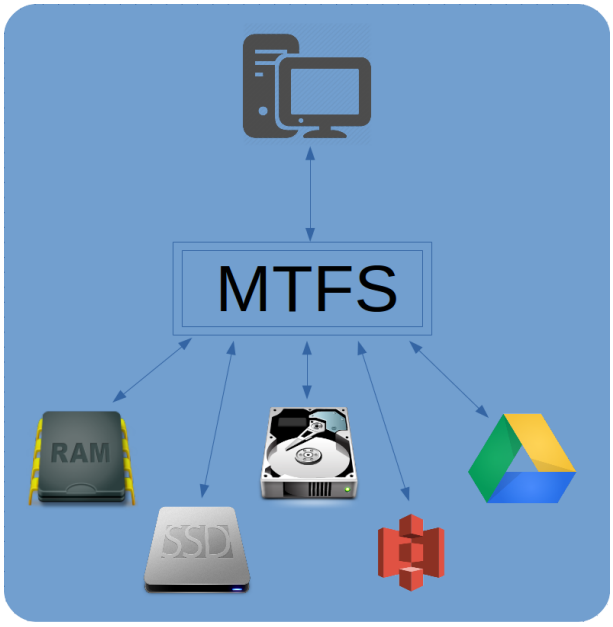
h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·SO GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

Multi Tier Filesystem



Thèse de Bachelor (Annexes) présentée par

Monsieur David WITTWER

pour l'obtention du titre Bachelor of Science HES-SO en

Ingénierie des technologies de l'information avec orientation en Logiciels et systèmes complexes

Septembre 2017

Professeur HES responsable TB
Florent GLÜCK

Table des matières

Chapitre 1

Mode d'emploi

1.1 Pré requis

Le système n'est pas livré sous forme d'exécutable pour l'instant il faut donc le compiler à partir des sources. Pour ce fait il faut disposer d'un compilateur C++ capable de compiler au minimum la version C++14 comme g++, et il faut aussi disposer de CMake[1] version 3.5 au minimum.

Les libraires suivantes doivent être installées :

— fuse version 3.0.* https://github.com/libfuse/libfuse ;

— boost filesystem http://www.boost.org/doc/libs/1_64_0/more/getting_started/unix-variants.html ;

— aws-sdk-cpp-s3 https://github.com/aws/aws-sdk-cpp ;

Outre ces librairies il est fortement recommandé de créer un utilisateur pour MTFS. Il est indispensable de créer un home pour MTFS, par défaut le home `/home/mtfs` est utilisé.

1.2 Compilation

La compilation est effectuée à l'aide de CMake, qui permet de compiler chaque exécutable y compris les tests, et de les installer. Les sources du projet sont disponibles à l'adresse https://github.com/Dawen18/mtfs. Pour la suite `SRC_PATH` correspond au dossier contenant les fichiers récupérés sur GitHub. Il est recommandé de créer un dossier pour les *build* de MTFS. Pour la suite de ces explications, `BUILD_PATH` correspond à ce dossier de *build*, et `HOME_DIR` au dossier home de MTFS (par defaut /home/mtfs). les commandes suivantes sont à exécuter pour installer MTFS :

— `cd <BUILD_PATH>` ;

— `cmake -DMTFS_HOME_DIR=<HOME_DIR> <SRC_PATH>` ;

— Pour compiler l'intégralité utilisez `make` ou alors spécifiez l'exécutable à compiler avec `make <exécutable>`

— `sudo cp mtFS/mtfsMount mtfs/mtfsCreate /usr/local/bin/.` : copie des deux exé-

cutables dans le dossier des binaires.

Il reste maintenant à installer les *plugins*. Avant tout il faut créer le dossier Libs dans le home de MTFS. `mkdir HOME_DIR/Libs`

1.2.1 *Plugin* block

La commande suivante peut être passée si la totalité du projet a été compilé au point précédent.

`make block`

Puis copier la libraire dans le dossier home de MTFS

`cp Plugin/BlockDevide/libblock.so /home/mtfs/Libs`

1.2.2 *Plugin* S3

La commande suivante peut être passée si la totalité du projet a été compilé en section 1.2.

`make s3`

Puis copier la libraire dans le dossier home de MTFS

`cp Plugin/S3/libs3.so /home/mtfs/Libs`

1.3 Création d'un espace de stockage

Les commandes suivantes sont celles utilisées pour la configuration utilisée lors des tests. En cas de doute sur une option, consultez l'aide de l'exécutable avec `mtfsCreate -h`.

— `mtfsCreate -n test`

— `mtfsCreate -i 0 -d 0 -b 0 -s 512 -r 0 test`

— `mtfsCreate -add test`

— `mtfsCreate -p 1 -add -c '{"highLimit": 2, "plName": "block", "params": {"devicePath": "/dev/sdb7", "fsType": "ext4"}}' test`

— `mtfsCreate -p 1 -add -c '{"lowLimit": 1, "plName": "s3", "params": {"bucket": "mtfs", "region": "eu-central-1"}}' test`

— `mtfsCreate -install test`

1.4 Montage de l'espace de stockage

Pour monter un espace de stockage voici la commande :

`mtfsMount [options] <mountpoint> <configname>`

Par exemple pour monter la configuration de test paramétrée ci-dessus la commande suivante est utilisée :

`mtfsMount -d -o allow_other /tmp/mtfs test`

Chapitre 2

Conventions de codage

Le code écrit au cours de ce projet suit des conventions afin de conserver un aspect homogène et faciliter ainsi sa lisibilité :

- Les noms des objets sont écrit en *UpperCamelCase* ;
- Les fonctions, types et variables sont écrits en *lowerCamelCase* ;
- Les structs sont post fixée avec "_st" et les typedef avec "_t"
- Les constantes et macro-définition sont écrites en majuscule et leurs mots sont sé- parés par des *underscore* (exemple : MTFS_HOME_DIR) ;
- Le code est documenté avec Doxygen :
  - Les fonctions publiques sont documentées uniquement dans le *header*. Ces commentaires ne sont pas dupliqués dans le fichier source.
  - A l'inverse les fonctions privées sont documentées dans le fichier source et non dans le *header*.
-

Chapitre 3

Code Source

3.1 *Config*

3.1.1 mtfsCreate.cpp

```
/**
 * \file mtfsCreate.cpp
 * \brief
 * \author David Wittwer
 * \version 0.0.1
 * \copyright GNU Publis License V3
 *
 * This file is part of MTFS.

    MTFS is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Foobar is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
 */

//#include <mtfs/structs.h>
#include <option/optionparser.h>
#include <iostream>
#include <utils/Fs.h>
#include <rapidjson/document.h>
#include <mtfs/Mtfs.h>
#include <rapidjson/stringbuffer.h>
#include <rapidjson/prettywriter.h>
#include <fstream>
#include <rapidjson/istreamwrapper.h>
#include <mtfs/TimeRule.h>
#include <mtfs/PoolManager.h>
#include <algorithm>
#include <zconf.h>
#include <boost/filesystem.hpp>
#include <pluginSystem/PluginManager.h>
```

```
40
41 using namespace std;
42 using namespace mtfs;
43 using namespace rapidjson;
44
45 struct Arg : public option::Arg {
46   static void printError(const char *msg1, const option::Option &opt, const
     char *msg2) {
47     fprintf(stderr, "ERROR: %s", msg1);
48     fwrite(opt.name, (size_t) opt.namelen, 1, stderr);
49     fprintf(stderr, "%s", msg2);
50   }
51
52   static option::ArgStatus Numeric(const option::Option &option, bool msg) {
53     char *endptr = nullptr;
54     if (option.arg != nullptr && (strtol(option.arg, &endptr, 10) != 0)) {};
55     if (endptr != option.arg && *endptr == 0)
56       return option::ARG_OK;
57
58     if (msg) printError("Option '", option, "' requires a numeric argument\n");
59     return option::ARG_ILLEGAL;
60   }
61
62   static option::ArgStatus NonEmpty(const option::Option &option, bool msg) {
63     if (option.arg != nullptr && option.arg[0] != 0)
64       return option::ARG_OK;
65
66     if (msg) printError("Option '", option, "' requires a non-empty argument\n"
     );
67     return option::ARG_ILLEGAL;
68   }
69
70   static option::ArgStatus Migration(const option::Option &option, bool msg) {
71     return option::ARG_OK;
72   }
73 };
74
75 enum optionIndex {
76   UNKNOWN,
77   NEW,
78   ICACHE,
79   DCACHE,
80   BCACHE,
81   BSIZE,
82   REDUNDANCY,
83   MIGRATION,
84   POOL,
85   VOLUME,
86   ADD,
87   DEL,
88   CONFIG,
89   INSTALL,
90   HELP
91 };
92 const option::Descriptor usage[] =
93     {
94         {UNKNOWN,      0, "",          "",              Arg::None,          "USAGE:
     mtfsCreate [options] <system_name>"},
```

```
95          {NEW,          0, "n",     "new",          Arg::NonEmpty,      "  -n, \t--
    new    \tCreate a new storage."},
96          {ICACHE,       0, "i",     "inode-cache", Arg::Numeric,       "  -i[SIZE],
     \t--inode-cache[=SIZE]  \tSet inode cache size."},
97          {DCACHE,       0, "d",     "dir-cache",    Arg::Numeric,       "  -d[SIZE],
     \t--dir-cache[=SIZE]  \tSet directory entry cache size."},
98          {BCACHE,       0, "b",     "block-cache", Arg::Numeric,       "  -b[SIZE],
     \t--block-cache[=SIZE]  \tSet block cache size."},
99          {BSIZE,        0, "s",     "block-size",  Arg::Numeric,       "  -s[SIZE],
     \t--block-size[=SIZE]  \tSet block size. Default 4096 octets"},
100         {REDUNDANCY, 0, "r",     "redundancy",  Arg::Numeric,       "  -r[NB], \
    t--redundancy[=NB]  \tSet number of redundancy for each block or inode."},
101         {MIGRATION,    0, "m",     "migration",    Arg::Migration,     "  -m[TYPE],
     \t--migration[=TYPE]  \tSet migration type in system or pool if -p is used"
    },
102         {POOL,         0, "p",     "pool",         Arg::Numeric,       "  -p[ID], \
    t--pool[=ID]    \tSpecify pool."},
103         {VOLUME,       0, "v",     "volume",       Arg::Numeric,       "  -v[ID], \
    t--volume[=ID]    \tSpecify volume in pool."},
104         {ADD,          0, "",      "add",          Arg::None,          "  \t--add
    \tAdd a pool or volume."},
105         {DEL,          0, "",      "del",          Arg::None,          "  \t--del
    \tDelete a pool or volume."},
106         {CONFIG,       0, "c",     "config",       Arg::NonEmpty,      "  -c[CONFIG
    ], \t--config[=CONFIG]  \tSet config for plugin or migration (json string or
     filename which contains de json config."},
107         {INSTALL,      0, "",      "install",      Arg::None,          "  -i,  \t--
    install  \tInstall the config."},
108         {HELP,         0, "h",     "help",         option::Arg::None, "  -h, \t--
    help \tPrint this help and exit."},
109         {UNKNOWN,      0, "",      "",             option::Arg::None, "\nExamples
    :\n"
110                                                 " Create system:\n"
111                                                 "  mtfsCreate -n mtfsRoot\n"
112                                                 " Add pool:\n"
113                                                 "  mtfsCreate -p mtfsRoot\n"
114                                                 " Add volume in pool:\n"
115                                                 "  mtfsCreate -v mtfsRoot 1\n"},
116         {0,            0, nullptr, nullptr,        nullptr,            nullptr}
117     };
118
119 bool configExist(const string &name);
120
121 void loadConfig(superblock_t &sb, const string &name);
122
123 bool writeConfig(superblock_t &superblock, const string &confName);
124
125 uint32_t addVolume(pool_t &pool, volume_t &volume);
126
127 uint32_t addPool(superblock_t &sb, pool_t &pool);
128
129 uint32_t findMissing(std::vector<uint32_t> &x, uint32_t number);
130
131 void installConfig(superblock_t &superblock, const string &name);
132
133
134 int main(int argc, char **argv) {
135   std::cout << MTFS_CONFIG_DIR << std::endl;
136   argc -= (argc > 0);
```

```
137   argv += (argc > 0);
138   option::Stats stats(usage, argc, argv);
139   option::Option options[stats.options_max], buffer[stats.buffer_max];
140   option::Parser parse(usage, argc, argv, options, buffer);
141
142   if (parse.error())
143     return 1;
144
145   if ((nullptr != options[HELP]) || 0 == argc) {
146     option::printUsage(cerr, usage);
147     return 0;
148   }
149
150   for (option::Option *opt = options[UNKNOWN]; opt; opt = opt->next())
151     cerr << "Unknown option: " << opt->name << "\n";
152   if (options[UNKNOWN] != nullptr)
153     return -1;
154
155 #ifdef DEBUG
156   for (int i = 0; i < parse.nonOptionsCount(); ++i)
157     std::cout << "Non-option #" << i << ": " << parse.nonOption(i) << "\n";
158 #endif
159
160   if (0 != chdir(MTFS_HOME_DIR)) {
161     return errno;
162   }
163
164   superblock_t superblock;
165   memset(&superblock, 0, sizeof(superblock_t));
166   superblock.pools.clear();
167
168   string confName;
169
170 //  if option new.
171   if (options[NEW] != nullptr) {
172 #ifndef DEBUG
173     if (configExist(options[NEW].arg)) {
174       cerr << "Config '" << options[NEW].arg << "' already exist" << endl;
175       return -1;
176     }
177 #endif
178
179     confName = options[NEW].arg;
180
181     superblock.iCacheSz = superblock.bCacheSz = superblock.dCacheSz =
      superblock.blockSz = 4096;
182     superblock.redundancy = 1;
183     superblock.migration = Rule::TIME_MIGRATION;
184
185   } else {
186     confName = parse.nonOption(0);
187     if (!configExist(confName)) {
188       cerr << "Config not exist" << endl;
189       return -1;
190     }
191
192 #ifdef DEBUG
193     cout << "Load config " << confName << endl;
194 #endif
```

```
195
196       loadConfig(superblock, confName);
197     }
198
199     if (nullptr != options[ICACHE]) {
200       superblock.iCacheSz = static_cast<size_t>(stoi(options[ICACHE].arg));
201     }
202
203     if (nullptr != options[DCACHE]) {
204       superblock.dCacheSz = static_cast<size_t>(stoi(options[DCACHE].arg));
205     }
206
207     if (nullptr != options[BCACHE]) {
208       superblock.bCacheSz = static_cast<size_t>(stoi(options[BCACHE].arg));
209     }
210
211     if (nullptr != options[BSIZE]) {
212       superblock.blockSz = static_cast<size_t>(stoi(options[BSIZE].arg));
213     }
214
215     if (nullptr != options[REDUNDANCY]) {
216       superblock.redundancy = static_cast<size_t>(stoi(options[REDUNDANCY].arg));
217     }
218
219     if (nullptr != options[MIGRATION] && nullptr == options[POOL]) {
220       string migration = options[MIGRATION].arg;
221
222       if ("time" == migration) {
223         superblock.migration = Rule::TIME_MIGRATION;
224       } else if ("user" == migration) {
225         superblock.migration = Rule::RIGHT_MIGRATION;
226       } else {
227         cerr << "unknow migration";
228         return -1;
229       }
230     } else if (nullptr != options[MIGRATION] && nullptr != options[POOL]) {
231     }
232
233     if (nullptr != options[ADD] && nullptr == options[POOL]) {
234 //    ADD one pool
235 #ifdef DEBUG
236       cout << "add a pool" << endl;
237 #endif
238       pool_t pool;
239       pool.migration = Rule::TIME_MIGRATION;
240       pool.volumes.clear();
241
242       if (nullptr != options[CONFIG]) {
243         string arg = options[CONFIG].arg;
244         Document tmpDoc;
245 #ifdef DEBUG
246         cout << "pool arg: " << arg << endl;
247 #endif
248
249         if (".json" == arg.substr(arg.length() - 5)) {
250 //          TODO Parse json file
251         } else
252           tmpDoc.Parse(arg.c_str());
253
```

```
254       if (0 > Rule::rulesAreValid(superblock.migration, tmpDoc)) {
255         cerr << "Invalid config!" << endl;
256         return -1;
257       }
258
259       if (tmpDoc.HasMember(Rule::MIGRATION))
260         pool.migration = tmpDoc[Rule::MIGRATION].GetInt();
261
262       if (tmpDoc.HasMember(TimeRule::TIME_LOW_LIMIT)) {
263         tmpDoc[TimeRule::TIME_LOW_LIMIT].SetUint(tmpDoc[TimeRule::
    TIME_LOW_LIMIT].GetUint() * 60);
264       }
265
266       if (tmpDoc.HasMember(TimeRule::TIME_HIGH_LIMIT))
267         tmpDoc[TimeRule::TIME_HIGH_LIMIT].SetUint(tmpDoc[TimeRule::
    TIME_HIGH_LIMIT].GetUint() * 60);
268
269       pool.rule = Rule::buildRule(superblock.migration, tmpDoc);
270     } else {
271 //      cerr << "config needed!" << endl;
272 //      return -1;
273       Document d;
274       d.SetObject();
275       Value v;
276       v.SetInt(0);
277       d.AddMember(StringRef("timeHighLimit"), v, d.GetAllocator());
278       pool.rule = Rule::buildRule(superblock.migration, d);
279     }
280
281     int newPoolId = addPool(superblock, pool);
282     if (0 < newPoolId)
283       cout << "new pool:" << newPoolId << endl;
284
285   } else if (nullptr != options[ADD] && nullptr != options[POOL]) {
286 //    ADD one volume.
287     if (superblock.pools.end() == superblock.pools.find(stoul(options[POOL].arg
    ))) {
288       cerr << "pool '" << options[POOL].arg << "' not found" << endl;
289       return -1;
290     }
291
292     uint32_t poolId = (uint32_t) stoul(options[POOL].arg);
293     volume_t volume;
294     pool_t *pool = &superblock.pools[poolId];
295
296     if (nullptr != options[CONFIG]) {
297       string arg = options[CONFIG].arg;
298       Document tmpDoc;
299 #ifdef DEBUG
300       cout << "pool arg: " << arg << endl;
301 #endif
302
303       if (".json" == arg.substr(arg.length() - 5)) {
304 //        TODO Parse json file
305       } else
306         tmpDoc.Parse(arg.c_str());
307
308       if (0 > Rule::rulesAreValid(pool->migration, tmpDoc)) {
309         cerr << "Invalid config!" << endl;
```

```
310         return -1;
311       }
312
313       if (!tmpDoc.HasMember(pluginSystem::Plugin::TYPE)) {
314         cerr << R"(config need a plugin name eg: {"plName":"block"})" << endl;
315         return -1;
316       }
317       volume.pluginName = tmpDoc[pluginSystem::Plugin::TYPE].GetString();
318
319       if (tmpDoc.HasMember(pluginSystem::Plugin::PARAMS)) {
320         for (auto &&param: tmpDoc[pluginSystem::Plugin::PARAMS].GetObject()) {
321           volume.params.insert(make_pair(param.name.GetString(), param.value.
    GetString()));
322         }
323       }
324
325       if (tmpDoc.HasMember(TimeRule::TIME_LOW_LIMIT)) {
326         tmpDoc[TimeRule::TIME_LOW_LIMIT].SetUint(tmpDoc[TimeRule::
    TIME_LOW_LIMIT].GetUint() * 60);
327       }
328
329       if (tmpDoc.HasMember(TimeRule::TIME_HIGH_LIMIT))
330         tmpDoc[TimeRule::TIME_HIGH_LIMIT].SetUint(tmpDoc[TimeRule::
    TIME_HIGH_LIMIT].GetUint() * 60);
331
332
333       volume.rule = Rule::buildRule(pool->migration, tmpDoc);
334     } else {
335       cerr << "config needed!" << endl;
336       return -1;
337     }
338
339     int newVolumeId = addVolume(*pool, volume);
340     if (0 < newVolumeId)
341       cout << "new volume:" << newVolumeId << endl;
342   }
343
344 //  TODO catch -p, -v anc -c options
345
346   if (nullptr != options[INSTALL]) {
347     installConfig(superblock, confName);
348     return EXIT_SUCCESS;
349   }
350
351   writeConfig(superblock, confName);
352
353   return 0;
354 }
355
356 //
    ////////////////////////////////////////////////////////////////////////////////
357 ////////////////////////////////////////////////Definitions
    ////////////////////////////////////////////////////////
358 //
    ////////////////////////////////////////////////////////////////////////////////
359
360 bool configExist(const string &name) {
```

```
361   return Fs::fileExists(MTFS_CONFIG_DIR, name + ".json");
362 }
363
364 void loadConfig(superblock_t &sb, const string &name) {
365   string filename = string(MTFS_CONFIG_DIR) + name + ".json";
366   ifstream file(filename);
367   if (!file.is_open()) {
368     return;
369   }
370
371   IStreamWrapper wrapper(file);
372   Document confFile;
373
374   confFile.ParseStream(wrapper);
375
376   Mtfs::jsonToStruct(confFile, sb);
377 }
378
379 bool writeConfig(superblock_t &superblock, const string &confName) {
380   Document d;
381   Document::AllocatorType &allocator = d.GetAllocator();
382   d.SetObject();
383
384   Value v;
385
386   Mtfs::structToJson(superblock, d);
387
388   StringBuffer sb;
389   PrettyWriter<StringBuffer> pw(sb);
390   d.Accept(pw);
391
392   string filename = string(MTFS_CONFIG_DIR) + confName + ".json";
393   ofstream configFile(filename);
394   configFile << sb.GetString() << endl;
395   configFile.close();
396
397   return true;
398 }
399
400 /**
401  * @brief Add volume in pool
402  *
403  * @param pool
404  * @param volume
405  *
406  * @return The volume Id
407  */
408 uint32_t addVolume(pool_t &pool, volume_t &volume) {
409   uint32_t volumeId = 0;
410   vector<uint32_t> ids;
411   ids.clear();
412
413   for (auto &item  : pool.volumes) {
414     ids.push_back(item.first);
415   }
416   if (ids.empty())
417     volumeId = 1;
418   else
419     volumeId = findMissing(ids, 0);
```

```
420
421   if (pool.volumes.end() != pool.volumes.find(volumeId))
422     return 0;
423
424   pool.volumes.insert(make_pair(volumeId, volume));
425
426   return volumeId;
427 }
428
429 /**
430  * @brief add a pool in system
431  *
432  * @param sb  The superblock
433  * @param pool  The pool to add
434  *
435  * @return the pool id
436  */
437 uint32_t addPool(superblock_t &sb, pool_t &pool) {
438   uint32_t poolId;
439   vector<uint32_t> ids;
440   ids.clear();
441
442   for (auto &item  : sb.pools) {
443     ids.push_back(item.first);
444   }
445   if (ids.empty())
446     poolId = 1;
447   else
448     poolId = findMissing(ids, 0);
449
450   if (sb.pools.end() != sb.pools.find(poolId))
451     return 0;
452
453   sb.pools.insert(make_pair(poolId, pool));
454
455   return poolId;
456 }
457
458 /**
459  * Function find on stackOverflow
460  *
461  * http://stackoverflow.com/questions/28176191/find-first-missing-element-in-a-
    vector
462  *
463  * @param x vector with content
464  * @param number Number from which to search
465  * @return The missing number.
466  */
467 uint32_t findMissing(std::vector<uint32_t> &x, uint32_t number) {
468   std::sort(x.begin(), x.end());
469   auto pos = std::upper_bound(x.begin(), x.end(), number);
470
471   if (*pos - number > 1)
472     return number + 1;
473
474   std::vector<int> diffs;
475   std::adjacent_difference(pos, x.end(), std::back_inserter(diffs));
476   auto pos2 = std::find_if(diffs.begin() + 1, diffs.end(), [](int x) { return x
    > 1; });
```

```
477   return *(pos + (pos2 - diffs.begin() - 1)) + 1;
478
479 }
480
481 void installConfig(superblock_t &superblock, const string &name) {
482 //  create ident for rootInode.
483   superblock.rootInodes.clear();
484   const int rootRedundancy = max(3, (int) superblock.redundancy);
485   int i = 0;
486   for (auto &&pool: superblock.pools) {
487     for (auto &&volume: pool.second.volumes) {
488       if (rootRedundancy > i)
489         superblock.rootInodes.emplace_back(0, volume.first, pool.first);
490       else
491         break;
492       i++;
493     }
494   }
495
496   writeConfig(superblock, name);
497
498   string filename = "superblock.json";
499   boost::filesystem::create_directory(MTFS_INSTALL_DIR + name);
500   string src = string(MTFS_CONFIG_DIR) + name + ".json";
501   string dst = string(MTFS_INSTALL_DIR) + name + "/" + filename;
502   boost::filesystem::copy_file(src, dst, boost::filesystem::copy_option::
      overwrite_if_exists);
503   string rootFilename = string(MTFS_INSTALL_DIR) + name + "/root.json";
504
505   inode_t rootInode;
506   Mtfs::createRootInode(rootInode);
507
508   Document rootJSON(kObjectType);
509   rootInode.toJson(rootJSON);
510
511 //  Attach all plugins for write superblock and rootInode
512 //  Only 3 firsts for rootInode
513   pluginSystem::PluginManager *manager = pluginSystem::PluginManager::
      getInstance();
514   i = 0;
515   for (auto &&pool: superblock.pools) {
516     for (auto &&volume: pool.second.volumes) {
517       volume.second.params["home"] = MTFS_PLUGIN_HOME;
518       volume.second.params["blockSize"] = to_string(superblock.blockSz);
519
520       pluginSystem::Plugin *plugin;
521
522       plugin = manager->getPlugin(volume.second.pluginName);
523       plugin->attach(volume.second.params);
524
525       if (rootRedundancy > i)
526         plugin->put(0, &rootInode, INODE, false);
527       plugin->putSuperblock(superblock);
528
529       i++;
530
531       plugin->detach();
532     }
533   }
```

```
534
535   StringBuffer sb;
536   PrettyWriter<StringBuffer> pw(sb);
537   rootJSON.Accept(pw);
538
539   ofstream configFile(rootFilename);
540   configFile << sb.GetString() << endl;
541   configFile.close();
542 }
```

Listing 3.1 – "mtfsCreate.cpp

3.1.2 mtfsMount.cpp

```
1  /**
2   * \file mtfsMount.cpp
3   * \brief
4   * \author David Wittwer
5   * \version 0.0.1
6   * \copyright GNU Publis License V3
7   *
8   * This file is part of MTFS.
9
10     MTFS is free software: you can redistribute it and/or modify
11     it under the terms of the GNU General Public License as published by
12     the Free Software Foundation, either version 3 of the License, or
13     (at your option) any later version.
14
15     Foobar is distributed in the hope that it will be useful,
16     but WITHOUT ANY WARRANTY; without even the implied warranty of
17     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18     GNU General Public License for more details.
19
20     You should have received a copy of the GNU General Public License
21     along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22  */
23
24 #define FUSE_USE_VERSION 30
25
26 #include <iostream>
27 #include <unistd.h>
28 #include <rapidjson/document.h>
29 #include <rapidjson/istreamwrapper.h>
30 #include <fstream>
31 #include <pluginSystem/PluginManager.h>
32 #include <mtfs/PoolManager.h>
33 #include <mtfs/Mtfs.h>
34 #include <wrapper/MtfsFuse.h>
35 #include <utils/Fs.h>
36 #include <boost/filesystem.hpp>
37 #include <utils/Logger.h>
38
39
40 using namespace std;
41 using namespace rapidjson;
42
43 int main(int argc, char **argv) {
44   if (!Fs::dirExists(MTFS_HOME_DIR)) {
```

```
45      cerr << "Sorry no configured system found." << endl;
46      cerr << "Please configure one or recover with -r device_in_system "
47          "where device_in_system is a device wich was in the configuration." <<
        endl;
48      return -1;
49    }
50
51
52    string sysName = argv[argc - 1];
53    string filename = sysName + ".json";
54    string filepath = string(MTFS_HOME_DIR) + "/" + mtfs::Mtfs::CONFIG_DIR + "/"
       + filename;
55    argc--;
56
57    if (!boost::filesystem::exists(filepath)) {
58      cerr << "File not found" << endl;
59      return -1;
60    }
61
62    chdir(MTFS_HOME_DIR);
63    ifstream file(string(mtfs::Mtfs::CONFIG_DIR) + "/" + filename);
64    if (!file.is_open()) {
65      cerr << "error openning file " << strerror(errno) << endl;
66      return -1;
67    }
68
69    IStreamWrapper wrapper(file);
70    Document d;
71    d.ParseStream(wrapper);
72
73  //  validate config file.
74    if (!mtfs::Mtfs::validate(d)) {
75      cerr << "Invalid or corrupted JSON!" << endl;
76      return -1;
77    }
78
79  //  build mtfs
80    mtfs::Mtfs::start(d, MTFS_HOME_DIR, sysName);
81
82  //  build mtsfFuse.
83    wrapper::MtfsFuse *mtfsFuse;
84    mtfsFuse = new wrapper::MtfsFuse();
85
86    int ret = mtfsFuse->run(argc, argv);
87
88    mtfs::Mtfs::stop();
89
90    return ret;
91  }
```

Listing 3.2 – "mtfsMount.cpp

3.2 *Plugin System*

3.2.1 Plugin.h

```
1  /**
```

```
2  * \file Plugin.h
3  * \brief
4  * \author David Wittwer
5  * \version 0.0.1
6  * \copyright GNU Publis License V3
7  *
8  * This file is part of MTFS.
9
10    MTFS is free software: you can redistribute it and/or modify
11    it under the terms of the GNU General Public License as published by
12    the Free Software Foundation, either version 3 of the License, or
13    (at your option) any later version.
14
15    Foobar is distributed in the hope that it will be useful,
16    but WITHOUT ANY WARRANTY; without even the implied warranty of
17    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18    GNU General Public License for more details.
19
20    You should have received a copy of the GNU General Public License
21    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22 */
23 #ifndef PLUGINSYSTEM_PLUGIN_H
24 #define PLUGINSYSTEM_PLUGIN_H
25
26 #include <string>
27 #include <vector>
28 #include <list>
29 #include <iostream>
30 #include <cassert>
31
32 #include <mtfs/structs.h>
33 #include <map>
34
35 namespace pluginSystem {
36   typedef struct {
37     std::string name;
38     std::vector<std::string> params;
39   } pluginInfo_t;
40
41   class Plugin {
42   public:
43     static constexpr const char *TYPE = "plName";
44     static constexpr const char *PARAMS = "params";
45
46     enum statusCode {
47       SUCCESS,
48     };
49
50     /**
51      * Get the name of plugin
52      *
53      * @return name of plugin
54      */
55     virtual std::string getName()=0;
56
57     virtual bool attach(std::map<std::string, std::string> params)=0;
58
59     virtual bool detach()=0;
60
```

```
61      virtual int add(uint64_t *id, const mtfs::blockType &type)=0;
62
63      virtual int del(const uint64_t &id, const mtfs::blockType &type)=0;
64
65      virtual int get(const uint64_t &id, void *data, const mtfs::blockType &type
        , bool metas)=0;
66
67      virtual int put(const uint64_t &id, const void *data, const mtfs::blockType
        &type, bool metas)=0;
68
69      virtual bool getSuperblock(mtfs::superblock_t &superblock)=0;
70
71      virtual bool putSuperblock(const mtfs::superblock_t &superblock)=0;
72
73    };
74
75  }  // namespace Plugin
76  #endif
```

Listing 3.3 – "pluginSystem/Plugin.h

3.2.2 PluginManager

PluginManager.h

```
1   /**
2    * \file PluginManager.h
3    * \brief
4    * \author David Wittwer
5    * \version 0.0.1
6    * \copyright GNU Publis License V3
7    *
8    * This file is part of MTFS.
9
10      MTFS is free software: you can redistribute it and/or modify
11      it under the terms of the GNU General Public License as published by
12      the Free Software Foundation, either version 3 of the License, or
13      (at your option) any later version.
14
15      Foobar is distributed in the hope that it will be useful,
16      but WITHOUT ANY WARRANTY; without even the implied warranty of
17      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18      GNU General Public License for more details.
19
20      You should have received a copy of the GNU General Public License
21      along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22   */
23
24  #ifndef TRAVAIL_BACHELOR_PLUGINMANAGER_H
25  #define TRAVAIL_BACHELOR_PLUGINMANAGER_H
26
27  #include <map>
28  #include <pluginSystem/Plugin.h>
29
30  namespace pluginSystem {
31    typedef struct {
32      pluginSystem::Plugin *(*createObj)();
33
```

```
34      void (*destroyObj)(Plugin *plugin);
35
36      pluginInfo_t (*getInfo)();
37    } plugin_t;
38
39    class PluginManager {
40    public:
41      static const int SUCCESS = 0;
42      static const int PLUGIN_FOUND = 0;
43      static const int PLUGIN_NOT_FOUND = -1;
44
45      /**
46       * @brief Get manager instance
47       *
48       * @return Instance of PluginManager
49       */
50      static PluginManager *getInstance();
51
52      /**
53       * @brief Get plugin
54       *
55       * @param pluginName Name of plugin
56       * @return nullptr if no plugin found
57       */
58      Plugin *getPlugin(std::string pluginName);
59
60      /**
61       * @brief destroy plugin object
62       *
63       * @param pluginName Name of plugin.
64       * @param plugin Tthe object to destroy.
65       */
66      void freePlugin(std::string pluginName, Plugin *plugin);
67
68      /**
69       * @brief Get plugin info.
70       *
71       * @param pluginName Name of plugin.
72       * @param info Struct who contains the info @see pluginInfo_t.
73       * @return SUCCESS or PLUGIN_NOT_FOUND
74       */
75      int getInfo(std::string pluginName, pluginInfo_t &info);
76
77      /**
78       * @brief Get last error code.
79       *
80       * @return The error code.
81       */
82      int getError();
83
84    private:
85      static constexpr const char *PLUGIN_DIR = "Libs";
86
87      std::map<std::string, plugin_t> pluginMap;
88      static PluginManager *instance;
89      int lastError;
90
91
92      PluginManager();
```

```
93
94      bool pluginExist(std::string name);
95
96      Plugin *loadPlugin(std::string name);
97
98    };
99  }
100
101
102 #endif //TRAVAIL_BACHELOR_PLUGINMANAGER_H
```

Listing 3.4 – "pluginSystem/PluginManager.h"

PluginManager.cpp

```
1  /**
2   * \file PluginManager.cpp
3   * \brief
4   * \author David Wittwer
5   * \version 0.0.1
6   * \copyright GNU Publis License V3
7   *
8   * This file is part of MTFS.
9
10     MTFS is free software: you can redistribute it and/or modify
11     it under the terms of the GNU General Public License as published by
12     the Free Software Foundation, either version 3 of the License, or
13     (at your option) any later version.
14
15     Foobar is distributed in the hope that it will be useful,
16     but WITHOUT ANY WARRANTY; without even the implied warranty of
17     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18     GNU General Public License for more details.
19
20     You should have received a copy of the GNU General Public License
21     along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22  */
23
24 #include <dlfcn.h>
25 #include <boost/filesystem.hpp>
26 #include <boost/range/iterator_range.hpp>
27 #include <pluginSystem/PluginManager.h>
28 #include <pluginSystem/Plugin.h>
29
30 #include "pluginSystem/PluginManager.h"
31 #include "../../../Plugin/S3/S3.h"
32
33 using namespace std;
34 using namespace boost::filesystem;
35
36 namespace pluginSystem {
37   PluginManager *PluginManager::instance = 0;
38
39   /*
     ********************************************************************************
     */
40   /*                              Publics                                       */
41   /*
     ********************************************************************************
```

```
 */

PluginManager *PluginManager::getInstance() {
  if (!instance)
    instance = new PluginManager();

  return instance;
}

Plugin *PluginManager::getPlugin(string pluginName) {
  map<string, plugin_t>::iterator it;

  it = pluginMap.find(pluginName);
  if (it != pluginMap.end()) {
    this->lastError = PLUGIN_FOUND;
    return it->second.createObj();
  }

  if (!pluginExist(pluginName)) {
    this->lastError = PLUGIN_NOT_FOUND;
    return nullptr;
  }

  return loadPlugin(pluginName);
}

void PluginManager::freePlugin(std::string pluginName, Plugin *plugin) {
  if (this->pluginMap.find(pluginName) == this->pluginMap.end()) {
    this->lastError = PLUGIN_NOT_FOUND;
    return;
  }

  this->pluginMap[pluginName].destroyObj(plugin);
  this->lastError = SUCCESS;
}

int PluginManager::getError() {
  return this->lastError;
}

int PluginManager::getInfo(std::string pluginName, pluginInfo_t &info) {
  if (this->pluginMap.find(pluginName) == this->pluginMap.end()) {
    this->lastError = PLUGIN_NOT_FOUND;
    return PLUGIN_NOT_FOUND;
  }

  this->lastError = SUCCESS;
  info = this->pluginMap[pluginName].getInfo();

  return SUCCESS;
}

/*
 ******************************************************************************
 */
/*                              Private                                       */
/*
 ******************************************************************************
 */
```

```
96
97  PluginManager::PluginManager() {
98  }
99
100 bool PluginManager::pluginExist(std::string name) {
101   string dir("./");
102   dir += PLUGIN_DIR;
103
104   if (is_directory(dir)) {
105     for (auto &entry  :boost::make_iterator_range(directory_iterator(dir))) {
106       if (entry.path().filename() == (string("lib") + name + ".so")) {
107         return true;
108       }
109     }
110   }
111   return false;
112 }
113
114 Plugin *PluginManager::loadPlugin(string name) {
115   plugin_t plugin{};
116   string path = string("./") + PLUGIN_DIR + "/lib" + name + ".so";
117
118 //    Open plugin
119   void *library = dlopen(path.c_str(), RTLD_LAZY);
120   if (nullptr == library) {
121     cerr << "Cannot load plugin '" << name << "': " << dlerror() << endl;
122     return nullptr;
123   }
124   dlerror();
125
126 //    Load createObj symbol
127   plugin.createObj = (pluginSystem::Plugin *(*)()) dlsym(library, "createObj"
    );
128   char *dlsym_error = dlerror();
129   if (nullptr != dlsym_error) {
130     cerr << "Cannot load symbol create: " << dlsym_error << endl;
131     return nullptr;
132   }
133   dlerror();
134
135   plugin.destroyObj = (void (*)(Plugin *)) dlsym(library, "destroyObj");
136   dlsym_error = dlerror();
137   if (dlsym_error) {
138     cerr << "Cannot load symbol destroy: " << dlsym_error << endl;
139     return nullptr;
140   }
141   dlerror();
142
143   plugin.getInfo = (pluginInfo_t (*)()) dlsym(library, "getInfo");
144   dlsym_error = dlerror();
145   if (dlsym_error) {
146     cerr << "Cannot load symbol info: " << dlsym_error << endl;
147     return nullptr;
148   }
149   dlerror();
150
151   dlclose(library);
152
153   this->pluginMap.insert(make_pair(plugin.getInfo().name, plugin));
```

```
154
155     return plugin.createObj();
156   }
157 }
```

Listing 3.5 – "pluginSystem/PluginManager.cpp

3.2.3 BlockDevice

BlockDevice.h

```
1  /**
2   * \file BlockDevice.h
3   * \brief
4   * \author David Wittwer
5   * \version 0.0.1
6   * \copyright GNU Publis License V3
7   *
8   * This file is part of MTFS.
9
10    MTFS is free software: you can redistribute it and/or modify
11    it under the terms of the GNU General Public License as published by
12    the Free Software Foundation, either version 3 of the License, or
13    (at your option) any later version.
14
15    Foobar is distributed in the hope that it will be useful,
16    but WITHOUT ANY WARRANTY; without even the implied warranty of
17    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18    GNU General Public License for more details.
19
20    You should have received a copy of the GNU General Public License
21    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22  */
23
24 #ifndef PLUGINSYSTEM_BLOCK_DEVICE_H
25 #define PLUGINSYSTEM_BLOCK_DEVICE_H
26
27 #include <string>
28 #include <vector>
29 #include <list>
30 #include <iostream>
31 #include <cassert>
32
33 #include <pluginSystem/Plugin.h>
34 #include <mtfs/structs.h>
35 #include <map>
36 #include <queue>
37 #include <mutex>
38
39 namespace pluginSystem {
40   class BlockDevice : public Plugin {
41     static constexpr const char *INODES_DIR = "inodes";
42     static constexpr const char *BLOCKS_DIR = "blocks";
43     static constexpr const char *DIR_BLOCKS_DIR = "dirBlocks";
44     static constexpr const char *METAS_DIR = "metas";
45     static constexpr const char *INODE_METAS_DIR = "metas/inodes";
46     static constexpr const char *DIR_BLOCK_METAS_DIR = "metas/dirBlocks";
47     static constexpr const char *BLOCK_METAS_DIR = "metas/blocks";
```

```
48
49      static const int SUCCESS = 0;
50
51    public:
52      static constexpr const char *NAME = "block";
53
54      BlockDevice();
55
56      virtual ~BlockDevice();
57
58      bool attach(std::map<std::string, std::string> params) override;
59
60      bool detach() override;
61
62      int add(uint64_t *id, const mtfs::blockType &type) override;
63
64      int del(const uint64_t &id, const mtfs::blockType &type) override;
65
66      int get(const uint64_t &id, void *data, const mtfs::blockType &type, bool
        metas) override;
67
68      int put(const uint64_t &id, const void *data, const mtfs::blockType &type,
        bool metas) override;
69
70      bool getSuperblock(mtfs::superblock_t &superblock) override;
71
72      bool putSuperblock(const mtfs::superblock_t &superblock) override;
73
74      std::string getName() override;
75
76    private:
77      int blockSize;
78      std::string mountpoint;
79      std::string devicePath;
80      std::string fsType;
81
82      std::mutex inodeMutex;
83      std::vector<uint64_t> freeInodes;
84      uint64_t nextFreeInode;
85
86      std::mutex dirBlockMutex;
87      std::vector<uint64_t> freeDirBlocks;
88      uint64_t nextFreeDirBlock;
89
90      std::mutex blockMutex;
91      std::vector<uint64_t> freeBlocks;
92      uint64_t nextFreeBlock;
93
94
95      int addInode(uint64_t *inodeId);
96
97      int addDirBlock(uint64_t *id);
98
99      int addBlock(uint64_t *blockId);
100
101     int delInode(const uint64_t &inodeId);
102
103     int getInode(const uint64_t &inodeId, mtfs::inode_st &inode);
104
```

```
105     int putInode(const uint64_t &inodeId, const mtfs::inode_st &inode);
106
107     int getInodeMetas(const uint64_t &inodeId, mtfs::blockInfo_t &metas);
108
109     int putInodeMetas(const uint64_t &inodeId, const mtfs::blockInfo_t &metas);
110
111     int delDirBlock(const uint64_t &id);
112
113     int getDirBlock(const uint64_t &id, mtfs::dirBlock_t &block);
114
115     int putDirBlock(const uint64_t &id, const mtfs::dirBlock_t &block);
116
117     int getDirBlockMetas(const uint64_t &id, mtfs::blockInfo_t &metas);
118
119     int putDirBlockMetas(const uint64_t &id, const mtfs::blockInfo_t &metas);
120
121     int delBlock(const uint64_t &blockId);
122
123     int getBlock(const uint64_t &blockId, std::uint8_t *buffer);
124
125     int putBlock(const uint64_t &blockId, const uint8_t *buffer);
126
127     int getBlockMetas(const uint64_t &blockId, mtfs::blockInfo_t &metas);
128
129     int putBlockMetas(const uint64_t &blockId, const mtfs::blockInfo_t &metas);
130
131
132     void initDirHierarchie();
133
134     void initInodes();
135
136     void initDirBlocks();
137
138     void initBlocks();
139
140     void writeMetas();
141
142     void logError(std::string message);
143
144     bool dirExists(std::string path);
145
146     int createFile(std::string path);
147
148     int deleteFile(std::string path);
149
150     int getMetas(const std::string &filename, mtfs::blockInfo_t &infos);
151
152     int putMetas(const std::string &filename, const mtfs::blockInfo_t &infos);
153
154   };
155
156 }  // namespace Plugin
157 #endif
```

Listing 3.6 – "pluginSystem/BlockDevice.h"

BlockDevice.cpp

```cpp
/**
 * \file BlockDevice.cpp
 * \brief
 * \author David Wittwer
 * \version 0.0.1
 * \copyright GNU Publis License V3
 *
 * This file is part of MTFS.

    MTFS is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Foobar is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <iostream>
#include <sys/mount.h>
#include <map>
#include <sys/stat.h>
#include <fcntl.h>
#include <rapidjson/document.h>
#include <rapidjson/stringbuffer.h>
#include <rapidjson/prettywriter.h>
#include <rapidjson/istreamwrapper.h>
#include <fstream>
#include <memory>
#include <utils/Logger.h>

#include "BlockDevice.h"

#define DEBUG

using namespace std;
using namespace rapidjson;
using namespace mtfs;

namespace pluginSystem {
  BlockDevice::BlockDevice() : nextFreeBlock(0), nextFreeDirBlock(0),
    nextFreeInode(1) {
    this->freeBlocks.clear();
    this->freeDirBlocks.clear();
    this->freeInodes.clear();
  }

  BlockDevice::~BlockDevice() {
    this->freeBlocks.clear();
    this->freeDirBlocks.clear();
    this->freeInodes.clear();
  }

  string BlockDevice::getName() {
```

```
59      return NAME;
60    }
61
62    bool BlockDevice::attach(std::map<string, string> params) {
63      if (params.find("devicePath") == params.end() || params.find("home") ==
       params.end() ||
64          params.find("fsType") == params.end())
65        return false;
66
67      string home;
68      home = params["home"];
69      if (home.back() != '/')
70        home += '/';
71      string parentDir = home + "BlockDevices/";
72      string bs = params["blockSize"];
73      this->blockSize = stoi(params.at("blockSize"));
74      this->fsType = params["fsType"];
75      this->devicePath = params["devicePath"];
76      this->mountpoint = parentDir + this->devicePath.substr(this->devicePath.
       find('/', 1) + 1) + '/';
77
78  #ifdef DEBUG
79      Logger::getInstance()->log("BLOCK", "create dir in home", Logger::L_INFO);
80  #endif
81
82      if (!dirExists(parentDir)) {
83        if (mkdir(parentDir.c_str(), 0700) != 0) {
84          logError("mkdir error " + parentDir + " " + strerror(errno));
85          return false;
86        }
87      }
88
89      if (!dirExists(this->mountpoint)) {
90        if (mkdir(this->mountpoint.c_str(), 0700) != 0) {
91          logError("mkdir error " + this->mountpoint + " " + strerror(errno));
92          return false;
93        }
94      }
95
96  #ifndef DEBUG
97      //    Mount device
98      mount(this->devicePath.c_str(), this->mountpoint.c_str(), this->fsType.
       c_str(), 0, nullptr);
99  #endif
100
101     initDirHierarchie();
102
103     initInodes();
104     initDirBlocks();
105     initBlocks();
106
107     return true;
108   }
109
110   bool BlockDevice::detach() {
111     writeMetas();
112
113 #ifndef DEBUG
114     if (umount2(this->mountpoint.c_str(), MNT_DETACH) != 0) {
```

```
115         cerr << "ERROR Failed to umount: " << this->mountpoint << endl;
116         cerr << "reason: " << strerror(errno) << " [" << errno << "]" << endl;
117         return false;
118       }
119 #endif
120 
121       return true;
122     }
123 
124     int BlockDevice::add(uint64_t *id, const blockType &type) {
125       int ret;
126       switch (type) {
127         case INODE:
128           ret = this->addInode(id);
129           break;
130         case DIR_BLOCK:
131           ret = this->addDirBlock(id);
132           break;
133         case DATA_BLOCK:
134           ret = this->addBlock(id);
135           break;
136         default:
137           ret = ENOSYS;
138           break;
139       }
140       return ret;
141     }
142 
143     int BlockDevice::del(const uint64_t &id, const mtfs::blockType &type) {
144       int ret;
145 
146       switch (type) {
147         case INODE:
148           ret = this->delInode(id);
149           break;
150         case DIR_BLOCK:
151           ret = this->delDirBlock(id);
152           break;
153         case DATA_BLOCK:
154           ret = this->delBlock(id);
155           break;
156         default:
157           ret = ENOSYS;
158           break;
159       }
160 
161       return ret;
162     }
163 
164     int BlockDevice::get(const uint64_t &id, void *data, const mtfs::blockType &
      type, const bool metas) {
165       int ret;
166 
167       switch (type) {
168         case INODE:
169           if (metas)
170             ret = this->getInodeMetas(id, *(blockInfo_t *) data);
171           else
172             ret = this->getInode(id, *(inode_t *) data);
```

```
173         break;
174       case DIR_BLOCK:
175         if (metas)
176           ret = this->getDirBlockMetas(id, *(blockInfo_t *) data);
177         else
178           ret = this->getDirBlock(id, *(dirBlock_t *) data);
179         break;
180       case DATA_BLOCK:
181         if (metas)
182           ret = this->getBlockMetas(id, *(blockInfo_t *) data);
183         else
184           ret = this->getBlock(id, (uint8_t *) data);
185         break;
186       default:
187         ret = ENOSYS;
188         break;
189     }
190
191     return ret;
192   }
193
194   int BlockDevice::put(const uint64_t &id, const void *data, const mtfs::
      blockType &type, const bool metas) {
195     int ret;
196
197     switch (type) {
198       case INODE:
199         if (metas)
200           ret = this->putInodeMetas(id, *(blockInfo_t *) data);
201         else
202           ret = this->putInode(id, *(inode_t *) data);
203         break;
204       case DIR_BLOCK:
205         if (metas)
206           ret = this->putDirBlockMetas(id, *(blockInfo_t *) data);
207         else
208           ret = this->putDirBlock(id, *(dirBlock_t *) data);
209         break;
210       case DATA_BLOCK:
211         if (metas)
212           ret = this->putBlockMetas(id, *(blockInfo_t *) data);
213         else
214           ret = this->putBlock(id, (uint8_t *) data);
215         break;
216       default:
217         ret = ENOSYS;
218         break;
219     }
220
221     return ret;
222   }
223
224   int BlockDevice::addInode(uint64_t *inodeId) {
225     unique_lock<mutex> lk(this->inodeMutex);
226     if (this->freeInodes.empty()) {
227       *inodeId = this->nextFreeInode;
228       this->nextFreeInode++;
229     } else {
230       *inodeId = this->freeInodes[0];
```

```
231       this->freeInodes.erase(freeInodes.begin());
232     }
233     lk.unlock();
234 
235     string filename = this->mountpoint;
236     if ('/' != filename.back())
237       filename += '/';
238 
239     filename = filename + INODES_DIR + "/" + to_string(*inodeId);
240 
241     return createFile(filename);
242   }
243 
244   int BlockDevice::delInode(const uint64_t &inodeId) {
245     unique_lock<mutex> lk(this->inodeMutex);
246     if (this->nextFreeInode - 1 == inodeId)
247       this->nextFreeInode--;
248     else {
249       this->freeInodes.push_back(inodeId);
250     }
251     lk.unlock();
252 
253     string filename = this->mountpoint;
254     if ('/' != filename.back())
255       filename += '/';
256 
257     filename = filename + INODES_DIR + "/" + to_string(inodeId);
258 
259     return deleteFile(filename);
260   }
261 
262   int BlockDevice::getInode(const uint64_t &inodeId, mtfs::inode_st &inode) {
263     string filename = this->mountpoint;
264     if ('/' != filename.back())
265       filename += '/';
266 
267     filename = filename + INODES_DIR + "/" + to_string(inodeId);
268     ifstream file(filename);
269     if (!file.is_open())
270       return errno != 0 ? errno : ENOENT;
271 
272     IStreamWrapper wrapper(file);
273 
274     Document d;
275     d.ParseStream(wrapper);
276 
277     assert(d.IsObject());
278     assert(d.HasMember(IN_MODE));
279     assert(d.HasMember(IN_UID));
280     assert(d.HasMember(IN_GID));
281     assert(d.HasMember(IN_SIZE));
282     assert(d.HasMember(IN_LINKS));
283     assert(d.HasMember(IN_ACCESS));
284     assert(d.HasMember(IN_BLOCKS));
285 
286     inode.accesRight = (uint16_t) d[IN_MODE].GetUint();
287     inode.uid = (uint16_t) d[IN_UID].GetUint();
288     inode.gid = (uint16_t) d[IN_GID].GetUint();
289     inode.size = d[IN_SIZE].GetUint64();
```

```
290     inode.linkCount = (uint8_t) d[IN_LINKS].GetUint();
291     inode.atime = d[IN_ACCESS].GetUint64();
292
293     const Value &dataArray = d[IN_BLOCKS];
294     assert(dataArray.IsArray());
295     inode.dataBlocks.clear();
296     for (auto &a : dataArray.GetArray()) {
297       vector<ident_t> redundancy;
298
299       assert(a.IsArray());
300       for (auto &v : a.GetArray()) {
301         ident_t ident;
302
303         assert(v.IsObject());
304         assert(v.HasMember(ID_POOL));
305         assert(v.HasMember(ID_VOLUME));
306         assert(v.HasMember(ID_ID));
307
308         ident.poolId = (uint16_t) v[ID_POOL].GetUint();
309         ident.volumeId = (uint16_t) v[ID_VOLUME].GetUint();
310         ident.id = v[ID_ID].GetUint64();
311
312         redundancy.push_back(ident);
313       }
314
315       inode.dataBlocks.push_back(redundancy);
316     }
317
318     return this->SUCCESS;
319   }
320
321   int BlockDevice::putInode(const uint64_t &inodeId, const inode_st &inode) {
322     Document d;
323     d.SetObject();
324     Document::AllocatorType &alloc = d.GetAllocator();
325
326     inode.toJson(d);
327
328     StringBuffer sb;
329     PrettyWriter<StringBuffer> pw(sb);
330     d.Accept(pw);
331
332     string filename = this->mountpoint;
333     if ('/' != filename.back())
334       filename += '/';
335
336     filename = filename + BlockDevice::INODES_DIR + "/" + to_string(inodeId);
337     ofstream inodeFile(filename);
338     inodeFile << sb.GetString() << endl;
339     inodeFile.close();
340
341     return this->SUCCESS;
342   }
343
344   int BlockDevice::getInodeMetas(const uint64_t &inodeId, mtfs::blockInfo_t &
      metas) {
345     string filename = this->mountpoint;
346     if ('/' != filename.back())
347       filename += '/';
```

```
348
349     filename = filename + INODE_METAS_DIR + "/" + to_string(inodeId) + ".json";
350
351     return getMetas(filename, metas);
352   }
353
354   int BlockDevice::putInodeMetas(const uint64_t &inodeId, const mtfs::
        blockInfo_t &metas) {
355     string filename = this->mountpoint;
356     if ('/' != filename.back())
357       filename += '/';
358
359     filename = filename + INODE_METAS_DIR + "/" + to_string(inodeId) + ".json";
360
361     return putMetas(filename, metas);
362   }
363
364   int BlockDevice::addDirBlock(uint64_t *id) {
365     unique_lock<mutex> lk(this->dirBlockMutex);
366     if (this->freeDirBlocks.empty()) {
367       *id = this->nextFreeDirBlock;
368       this->nextFreeDirBlock++;
369     } else {
370       *id = this->freeDirBlocks.front();
371       iter_swap(this->freeDirBlocks.begin(), this->freeDirBlocks.end());
372       this->freeDirBlocks.pop_back();
373     }
374     lk.unlock();
375
376     string filename = this->mountpoint;
377     if ('/' != filename.back())
378       filename += '/';
379
380     filename = filename + DIR_BLOCKS_DIR + "/" + to_string(*id);
381
382     return createFile(filename);
383   }
384
385   int BlockDevice::delDirBlock(const uint64_t &id) {
386     unique_lock<mutex> lk(this->dirBlockMutex);
387     if (id == this->nextFreeDirBlock - 1)
388       this->nextFreeDirBlock--;
389     else
390       this->freeDirBlocks.push_back(id);
391     lk.unlock();
392
393     string filename = this->mountpoint;
394     if ('/' != filename.back())
395       filename += '/';
396
397     filename = filename + DIR_BLOCKS_DIR + "/" + to_string(id);
398
399     return deleteFile(filename);
400   }
401
402   int BlockDevice::getDirBlock(const uint64_t &id, dirBlock_t &block) {
403     string filename = this->mountpoint;
404     if ('/' != filename.back())
405       filename += '/';
```

```
406
407     filename = filename + DIR_BLOCKS_DIR + "/" + to_string(id);
408
409     ifstream file(filename);
410     if (!file.is_open())
411       return errno != 0 ? errno : ENOENT;
412
413
414     IStreamWrapper wrapper(file);
415
416     Document d;
417     d.ParseStream(wrapper);
418
419     assert(d.IsObject());
420
421     for (auto &&item: d.GetObject()) {
422       assert(item.value.IsArray());
423
424       vector<ident_t> ids;
425
426       for (auto &&ident: item.value.GetArray()) {
427         ident_t i;
428         i.fromJson(ident);
429         ids.push_back(i);
430       }
431       block.entries.insert(make_pair(item.name.GetString(), ids));
432     }
433     return this->SUCCESS;
434   }
435
436   int BlockDevice::putDirBlock(const uint64_t &id, const dirBlock_t &block) {
437     Document d;
438     d.SetObject();
439
440     Document::AllocatorType &allocator = d.GetAllocator();
441
442     for (auto &&item: block.entries) {
443       Value r(kArrayType);
444       for (auto &&ident: item.second) {
445         Value v(kObjectType);
446         ident.toJson(v, allocator);
447         r.PushBack(v, allocator);
448       }
449       d.AddMember(StringRef(item.first.c_str()), r, allocator);
450     }
451
452     StringBuffer sb;
453     PrettyWriter<StringBuffer> pw(sb);
454     d.Accept(pw);
455
456     string filename = this->mountpoint;
457     if ('/' != filename.back())
458       filename += '/';
459
460     filename = filename + BlockDevice::DIR_BLOCKS_DIR + "/" + to_string(id);
461     ofstream file(filename);
462     if (!file.is_open())
463       return errno != 0 ? errno : ENOENT;
464     file << sb.GetString() << endl;
```

```
465     file.close();
466
467     return this->SUCCESS;
468   }
469
470   int BlockDevice::getDirBlockMetas(const uint64_t &id, mtfs::blockInfo_t &
      metas) {
471     string filename = this->mountpoint;
472     if ('/' != filename.back())
473       filename += '/';
474
475     filename = filename + DIR_BLOCK_METAS_DIR + "/" + to_string(id) + ".json";
476     return getMetas(filename, metas);
477   }
478
479   int BlockDevice::putDirBlockMetas(const uint64_t &id, const mtfs::blockInfo_t
      &metas) {
480     string filename = this->mountpoint;
481     if ('/' != filename.back())
482       filename += '/';
483
484     filename = filename + DIR_BLOCK_METAS_DIR + "/" + to_string(id) + ".json";
485     return putMetas(filename, metas);
486   }
487
488   int BlockDevice::addBlock(uint64_t *blockId) {
489     unique_lock<mutex> lk(this->blockMutex);
490     if (this->freeBlocks.empty()) {
491       *blockId = this->nextFreeBlock;
492       this->nextFreeBlock++;
493     } else {
494       *blockId = this->freeBlocks[this->freeBlocks.size() - 1];
495       this->freeBlocks.pop_back();
496     }
497     lk.unlock();
498
499     string filename = this->mountpoint;
500     if ('/' != filename.back())
501       filename += '/';
502
503     filename = filename + BLOCKS_DIR + "/" + to_string(*blockId);
504
505     return createFile(filename);
506   }
507
508   int BlockDevice::delBlock(const uint64_t &blockId) {
509     unique_lock<mutex> lk(this->blockMutex);
510     if (this->nextFreeBlock - 1 == blockId)
511       nextFreeBlock--;
512     else
513       this->freeBlocks.push_back(blockId);
514     lk.unlock();
515
516     string filename = this->mountpoint;
517     if ('/' != filename.back())
518       filename += '/';
519
520     filename = filename + BLOCKS_DIR + "/" + to_string(blockId);
521
```

```
522     return deleteFile(filename);
523   }
524
525   int BlockDevice::getBlock(const uint64_t &blockId, std::uint8_t *buffer) {
526     string filename = this->mountpoint;
527     if ('/' != filename.back())
528       filename += '/';
529
530     filename = filename + BLOCKS_DIR + "/" + to_string(blockId);
531
532     ifstream file(filename);
533     if (!file.is_open())
534       return errno != 0 ? errno : ENOENT;
535
536     file.read((char *) buffer, this->blockSize);
537     file.close();
538     return this->SUCCESS;
539   }
540
541   int BlockDevice::putBlock(const uint64_t &blockId, const uint8_t *buffer) {
542     string filename = this->mountpoint;
543     if ('/' != filename.back())
544       filename += '/';
545
546     filename = filename + BLOCKS_DIR + "/" + to_string(blockId);
547     ofstream file(filename);
548     if (!file.is_open())
549       return errno != 0 ? errno : ENOENT;
550
551     file.write((const char *) buffer, this->blockSize);
552     file.close();
553     return this->SUCCESS;
554   }
555
556   int BlockDevice::getBlockMetas(const uint64_t &blockId, mtfs::blockInfo_t &
      metas) {
557     string filename = this->mountpoint;
558     if ('/' != filename.back())
559       filename += '/';
560
561     filename = filename + BLOCK_METAS_DIR + "/" + to_string(blockId) + ".json";
562
563     return getMetas(filename, metas);
564   }
565
566   int BlockDevice::putBlockMetas(const uint64_t &blockId, const blockInfo_t &
      metas) {
567     string filename = this->mountpoint;
568     if ('/' != filename.back())
569       filename += '/';
570
571     filename = filename + BLOCK_METAS_DIR + "/" + to_string(blockId) + ".json";
572
573     return putMetas(filename, metas);
574   }
575
576   bool BlockDevice::getSuperblock(mtfs::superblock_t &superblock) {
577     return false;
578   }
```

```
579
580   bool BlockDevice::putSuperblock(const superblock_t &superblock) {
581     string filename = this->mountpoint;
582     if ('/' != filename.back())
583       filename += '/';
584
585     filename = filename + METAS_DIR + "/superblock";
586     ofstream sbFile(filename, ios::binary);
587     sbFile.write((const char *) &superblock, sizeof(superblock));
588     sbFile.close();
589
590     return true;
591   }
592
593   //////////////Private method///////////////
594
595   void BlockDevice::initDirHierarchie() {
596     if (!dirExists(this->mountpoint + BlockDevice::INODES_DIR))
597       mkdir((this->mountpoint + BlockDevice::INODES_DIR).c_str(), 0700);
598     if (!dirExists(this->mountpoint + BlockDevice::DIR_BLOCKS_DIR))
599       mkdir((this->mountpoint + BlockDevice::DIR_BLOCKS_DIR).c_str(), 0700);
600     if (!dirExists(this->mountpoint + BlockDevice::BLOCKS_DIR))
601       mkdir((this->mountpoint + BlockDevice::BLOCKS_DIR).c_str(), 0700);
602     if (!dirExists(this->mountpoint + BlockDevice::METAS_DIR))
603       mkdir((this->mountpoint + BlockDevice::METAS_DIR).c_str(), 0700);
604     if (!dirExists(this->mountpoint + BlockDevice::INODE_METAS_DIR))
605       mkdir((this->mountpoint + BlockDevice::INODE_METAS_DIR).c_str(), 0700);
606     if (!dirExists(this->mountpoint + BlockDevice::DIR_BLOCK_METAS_DIR))
607       mkdir((this->mountpoint + BlockDevice::DIR_BLOCK_METAS_DIR).c_str(),
    0700);
608     if (!dirExists(this->mountpoint + BlockDevice::BLOCK_METAS_DIR))
609       mkdir((this->mountpoint + BlockDevice::BLOCK_METAS_DIR).c_str(), 0700);
610   }
611
612   void BlockDevice::initInodes() {
613     string filename = this->mountpoint;
614     if ('/' != filename.back())
615       filename += '/';
616
617     filename = filename + METAS_DIR + "/inodes.json";
618
619     ifstream inodeFile(filename);
620     if (!inodeFile.is_open())
621       return;
622
623     IStreamWrapper isw(inodeFile);
624
625     Document d;
626     d.ParseStream(isw);
627
628     assert(d.IsObject());
629     assert(d.HasMember("nextFreeInode"));
630     assert(d.HasMember("freeInodes"));
631     this->nextFreeInode = d["nextFreeInode"].GetUint64();
632     const Value &inodeArray = d["freeInodes"];
633
634     this->freeInodes.clear();
635     assert(inodeArray.IsArray());
636     for (SizeType i = 0; i < inodeArray.Size(); i++) {
```

```
637       this->freeInodes.push_back(inodeArray[i].GetUint64());
638     }
639   }
640
641   void BlockDevice::initDirBlocks() {
642     string filename = this->mountpoint;
643     if ('/' != filename.back())
644       filename += '/';
645
646     filename = filename + METAS_DIR + "/dirBlocks.json";
647
648     ifstream inodeFile(filename);
649     if (!inodeFile.is_open())
650       return;
651
652     IStreamWrapper isw(inodeFile);
653
654     Document d;
655     d.ParseStream(isw);
656
657     assert(d.IsObject());
658     assert(d.HasMember("nextFreeDirBlock"));
659     assert(d.HasMember("freeDirBlocks"));
660     this->nextFreeDirBlock = d["nextFreeDirBlock"].GetUint64();
661     const Value &array = d["freeDirBlocks"];
662
663     this->freeDirBlocks.clear();
664     assert(array.IsArray());
665     for (SizeType i = 0; i < array.Size(); i++) {
666       this->freeDirBlocks.push_back(array[i].GetUint64());
667     }
668   }
669
670   void BlockDevice::initBlocks() {
671     string filename = this->mountpoint;
672     if ('/' != filename.back())
673       filename += '/';
674
675     filename = filename + METAS_DIR + "/blocks.json";
676
677     ifstream inodeFile(filename);
678     if (!inodeFile.is_open())
679       return;
680
681     IStreamWrapper isw(inodeFile);
682
683     Document d;
684     d.ParseStream(isw);
685
686     assert(d.IsObject());
687     assert(d.HasMember("nextFreeBlock"));
688     assert(d.HasMember("freeBlocks"));
689     this->nextFreeBlock = d["nextFreeBlock"].GetUint64();
690     const Value &inodeArray = d["freeBlocks"];
691
692     this->freeBlocks.clear();
693     assert(inodeArray.IsArray());
694     for (SizeType i = 0; i < inodeArray.Size(); i++) {
695       this->freeBlocks.push_back(inodeArray[i].GetUint64());
```

```
696     }
697   }
698
699   void BlockDevice::writeMetas() {
700     {
701 //    writeInodeMeta
702       Document d;
703       d.SetObject();
704
705       Document::AllocatorType &allocator = d.GetAllocator();
706
707       Value value(kObjectType);
708       value.SetUint64(this->nextFreeInode);
709       d.AddMember(StringRef("nextFreeInode"), value, allocator);
710
711       Value inodeList(kArrayType);
712       Value inode(kObjectType);
713
714       for (unsigned long &freeInode : this->freeInodes) {
715         inode.SetUint64(freeInode);
716         inodeList.PushBack(Value(freeInode), allocator);
717       }
718
719       d.AddMember(StringRef("freeInodes"), inodeList, allocator);
720
721       StringBuffer strBuff;
722       PrettyWriter<StringBuffer> writer(strBuff);
723       d.Accept(writer);
724
725       string filename = this->mountpoint;
726       if ('/' != filename.back())
727         filename += '/';
728
729       filename = filename + METAS_DIR + "/inodes.json";
730
731       ofstream inodeFile;
732       inodeFile.open(filename);
733       inodeFile << strBuff.GetString() << endl;
734       inodeFile.close();
735     }
736
737     {
738 //    writeDirBlocksMeta
739       Document d;
740       d.SetObject();
741
742       Document::AllocatorType &allocator = d.GetAllocator();
743
744       Value freeInode(kObjectType);
745       freeInode.SetUint64(this->nextFreeDirBlock);
746       d.AddMember(StringRef("nextFreeDirBlock"), freeInode, allocator);
747
748       Value inodeList(kArrayType);
749       Value inode(kObjectType);
750
751       for (unsigned long &freeDirBlock : this->freeDirBlocks) {
752         inode.SetUint64(freeDirBlock);
753         inodeList.PushBack(Value(freeDirBlock), allocator);
754       }
```

```
755
756       d.AddMember(StringRef("freeDirBlocks"), inodeList, allocator);
757
758       StringBuffer strBuff;
759       PrettyWriter<StringBuffer> writer(strBuff);
760       d.Accept(writer);
761
762       string filename = this->mountpoint;
763       if ('/' != filename.back())
764         filename += '/';
765
766       filename = filename + METAS_DIR + "/dirBlocks.json";
767
768       ofstream inodeFile;
769       inodeFile.open(filename);
770       inodeFile << strBuff.GetString() << endl;
771       inodeFile.close();
772     }
773
774     {
775 //    Write block metas
776       Document db;
777       db.SetObject();
778
779       Document::AllocatorType &blAllocator = db.GetAllocator();
780
781       Value freeBlock(kObjectType);
782       freeBlock.SetUint64(this->nextFreeBlock);
783       db.AddMember(StringRef("nextFreeBlock"), freeBlock, blAllocator);
784
785       Value blockList(kArrayType);
786
787       for (auto b :this->freeBlocks) {
788         blockList.PushBack(Value(b), blAllocator);
789       }
790
791       db.AddMember(StringRef("freeBlocks"), blockList, blAllocator);
792
793       StringBuffer bStrBuff;
794       PrettyWriter<StringBuffer> bWriter(bStrBuff);
795       db.Accept(bWriter);
796
797       string filename = this->mountpoint;
798       if ('/' != filename.back())
799         filename += '/';
800
801       filename = filename + METAS_DIR + "/blocks.json";
802
803       ofstream blockFile;
804       blockFile.open(filename);
805       blockFile << bStrBuff.GetString() << endl;
806       blockFile.close();
807     }
808   }
809
810   void BlockDevice::logError(string message) {
811     cerr << message << endl;
812   }
813
```

```
814  bool BlockDevice::dirExists(string path) {
815    struct stat info{};
816
817    if (stat(path.c_str(), &info) != 0)
818      return false;
819    return (info.st_mode & S_IFDIR) != 0;
820  }
821
822  int BlockDevice::createFile(string path) {
823    if (creat(path.c_str(), 0600) < 0) {
824      string message = "create " + path + " error " + string(strerror(errno)) +
     " [" + to_string(errno) + "]";
825      logError(message);
826      return errno;
827    }
828    return this->SUCCESS;
829  }
830
831  int BlockDevice::deleteFile(std::string path) {
832    if (remove(path.c_str()) != 0) {
833      string message = "ERROR: deleting file " + path;
834      logError(message);
835      return errno;
836    }
837    return SUCCESS;
838  }
839
840  int BlockDevice::getMetas(const std::string &filename, mtfs::blockInfo_t &
     infos) {
841    ifstream file(filename);
842    if (!file.is_open())
843      return 0 != errno ? errno : EAGAIN;
844
845    IStreamWrapper wrapper(file);
846
847    Document d;
848    d.ParseStream(wrapper);
849
850    assert(d.HasMember(BI_REFF));
851    assert(d[BI_REFF].IsArray());
852    assert(d.HasMember(BI_ACCESS));
853
854    for (auto &&id :d[BI_REFF].GetArray()) {
855      assert(id.IsObject());
856      assert(id.HasMember(ID_POOL));
857      assert(id.HasMember(ID_VOLUME));
858      assert(id.HasMember(ID_ID));
859
860      uint32_t poolId = id[ID_POOL].GetUint();
861      uint32_t volumeId = id[ID_VOLUME].GetUint();
862      uint64_t i = id[ID_ID].GetUint64();
863
864      infos.referenceId.emplace_back(i, volumeId, poolId);
865    }
866
867    infos.lastAccess = d[BI_ACCESS].GetUint64();
868
869    return 0;
870  }
```

```
871
872   int BlockDevice::putMetas(const std::string &filename, const mtfs::
        blockInfo_t &infos) {
873     Document d;
874     d.SetObject();
875     Document::AllocatorType &allocator = d.GetAllocator();
876
877     Value v;
878
879     Value a(kArrayType);
880     for (auto &&id : infos.referenceId) {
881       v.SetObject();
882
883       v.AddMember(StringRef(ID_POOL), Value(id.poolId), allocator);
884       v.AddMember(StringRef(ID_VOLUME), Value(id.volumeId), allocator);
885       v.AddMember(StringRef(ID_ID), Value(id.id), allocator);
886
887       a.PushBack(v, allocator);
888     }
889     d.AddMember(StringRef(BI_REFF), a, allocator);
890
891     v.SetUint64(infos.lastAccess);
892     d.AddMember(StringRef(BI_ACCESS), v, allocator);
893
894     StringBuffer bStrBuff;
895     PrettyWriter<StringBuffer> bWriter(bStrBuff);
896     d.Accept(bWriter);
897
898     ofstream blockFile;
899     blockFile.open(filename);
900     if (!blockFile.is_open())
901       return -1;
902
903     blockFile << bStrBuff.GetString() << endl;
904     blockFile.close();
905
906     return 0;
907   }
908
909
910 }  // namespace Plugin
911
912 extern "C" pluginSystem::Plugin *createObj() {
913   return new pluginSystem::BlockDevice();
914 }
915
916 extern "C" void destroyObj(pluginSystem::Plugin *plugin) {
917   delete plugin;
918 }
919
920 extern "C" pluginSystem::pluginInfo_t getInfo() {
921   vector<string> params;
922   params.emplace_back("devicePath");
923   params.emplace_back("fsType");
924
925   pluginSystem::pluginInfo_t info = {
926       .name = pluginSystem::BlockDevice::NAME,
927       .params = params,
928   };
```

```
929
930   return info;
931 }
```

Listing 3.7 – "pluginSystem/BlockDevice.cpp

3.2.4 S3

S3.h

```
1  /**
2   * \file S3.h
3   * \brief
4   * \author David Wittwer
5   * \version 0.0.1
6   * \copyright GNU Publis License V3
7   *
8   * This file is part of MTFS.
9   *
10     MTFS is free software: you can redistribute it and/or modify
11     it under the terms of the GNU General Public License as published by
12     the Free Software Foundation, either version 3 of the License, or
13     (at your option) any later version.
14
15     Foobar is distributed in the hope that it will be useful,
16     but WITHOUT ANY WARRANTY; without even the implied warranty of
17     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18     GNU General Public License for more details.
19
20     You should have received a copy of the GNU General Public License
21     along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22   */
23
24 #ifndef MTFS_S3_H
25 #define MTFS_S3_H
26
27 #include <pluginSystem/Plugin.h>
28 #include <aws/core/Aws.h>
29 #include <aws/s3/S3Client.h>
30
31 namespace pluginSystem {
32   class S3 : public Plugin {
33   public:
34
35     static constexpr const char *NAME = "s3";
36
37     S3();
38
39     std::string getName() override;
40
41     bool attach(std::map<std::string, std::string> params) override;
42
43     bool detach() override;
44
45     int add(uint64_t *id, const mtfs::blockType &type) override;
46
47     int del(const uint64_t &id, const mtfs::blockType &type) override;
48
```

```
49      int get(const uint64_t &id, void *data, const mtfs::blockType &type, bool
        metas) override;
50
51      int put(const uint64_t &id, const void *data, const mtfs::blockType &type,
        bool metas) override;
52
53      bool getSuperblock(mtfs::superblock_t &superblock) override;
54
55      bool putSuperblock(const mtfs::superblock_t &superblock) override;
56
57    private:
58      static constexpr const char *INODE_PREFIX = "inodes/";
59      static constexpr const char *DIRECTORY_PREFIX = "directories/";
60      static constexpr const char *DATA_PREFIX = "datas/";
61
62      Aws::S3::S3Client *s3Client;
63      Aws::SDKOptions *options;
64      std::string home;
65      size_t blocksize;
66      Aws::String bucket;
67
68      std::mutex inodeMutex;
69      std::vector<uint64_t> freeInodes;
70      uint64_t nextFreeInode;
71
72      std::mutex dirBlockMutex;
73      std::vector<uint64_t> freeDirectory;
74      uint64_t nextFreeDirectory;
75
76      std::mutex blockMutex;
77      std::vector<uint64_t> freeDatas;
78      uint64_t nextFreeData;
79
80      int writeTmpFile(const std::string &filename, const void *data, const mtfs
        ::blockType &type);
81
82      int writeInode(const std::string &filename, const mtfs::inode_t *inode);
83
84      int writeDirBlock(const std::string &filename, const mtfs::dirBlock_t *
        dirBlock);
85
86      int writeDataBlock(const std::string &filename, const uint8_t *datas);
87
88      int writeSuperblock(const std::string &filename, const mtfs::superblock_t *
        superblock);
89
90      int dlObj(const std::string &filename, const std::string &key);
91
92      int upObj(const std::string &filename, const std::string &key);
93
94      bool dirExists(std::string path);
95
96      void initIds();
97
98      void writeMetas();
99
100     void delObj(const std::string &key);
101   };
102 }
```

```
103
104 #endif //MTFS_S3_H
```

Listing 3.8 – "pluginSystem/S3.h"

S3.cpp

```
1  /**
2   * \file S3.cpp
3   * \brief
4   * \author David Wittwer
5   * \version 0.0.1
6   * \copyright GNU Publis License V3
7   *
8   * This file is part of MTFS.
9
10     MTFS is free software: you can redistribute it and/or modify
11     it under the terms of the GNU General Public License as published by
12     the Free Software Foundation, either version 3 of the License, or
13     (at your option) any later version.
14
15     Foobar is distributed in the hope that it will be useful,
16     but WITHOUT ANY WARRANTY; without even the implied warranty of
17     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18     GNU General Public License for more details.
19
20     You should have received a copy of the GNU General Public License
21     along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22  */
23
24 #include <aws/s3/model/PutObjectRequest.h>
25 #include <aws/s3/model/GetObjectRequest.h>
26 #include <aws/s3/model/DeleteObjectRequest.h>
27 #include <iostream>
28 #include <fstream>
29 #include <rapidjson/stringbuffer.h>
30 #include <rapidjson/prettywriter.h>
31 #include <sys/stat.h>
32 #include <utils/Logger.h>
33 #include <zconf.h>
34 #include <rapidjson/istreamwrapper.h>
35 #include <mtfs/structs.h>
36
37 #include "S3.h"
38
39 using namespace std;
40 using namespace Aws::S3;
41 using namespace rapidjson;
42
43 namespace pluginSystem {
44
45   S3::S3() : nextFreeInode(1), nextFreeDirectory(0), nextFreeData(0) {
46     this->freeDatas.clear();
47     this->freeDirectory.clear();
48     this->freeInodes.clear();
49   }
50
51   std::string S3::getName() {
52     return S3::NAME;
```

```
53  }
54
55  bool S3::attach(std::map<std::string, std::string> params) {
56    if (params.end() == params.find("home") || params.end() == params.find("
    blockSize") ||
57      params.end() == params.find("region") || params.end() == params.find("
    bucket"))
58      return false;
59
60    this->blocksize = static_cast<size_t>(stoi(params["blockSize"]));
61    this->bucket = params["bucket"].c_str();
62    this->home = params["home"] + this->bucket.c_str() + "/";
63
64    this->options = new Aws::SDKOptions();
65    Aws::InitAPI(*this->options);
66
67    Aws::Client::ClientConfiguration configuration;
68    configuration.region = params["region"].c_str();
69    this->s3Client = new S3Client(configuration);
70
71    if (!dirExists(this->home)) {
72      if (mkdir(this->home.c_str(), 0700) != 0) {
73        Logger::getInstance()->log("S3", "mkdir error " + this->home + " " +
    strerror(errno), Logger::L_ERROR);
74        return false;
75      }
76    }
77
78    this->initIds();
79
80    return true;
81  }
82
83  bool S3::detach() {
84    this->writeMetas();
85
86    Aws::ShutdownAPI(*this->options);
87
88    return true;
89  }
90
91  int S3::add(uint64_t *id, const mtfs::blockType &type) {
92
93    mutex *mu = nullptr;
94    vector<uint64_t> *ve = nullptr;
95    uint64_t *nextId;
96
97    switch (type) {
98      case mtfs::INODE:
99        mu = &this->inodeMutex;
100       ve = &this->freeInodes;
101       nextId = &this->nextFreeInode;
102       break;
103     case mtfs::DIR_BLOCK:
104       mu = &this->dirBlockMutex;
105       ve = &this->freeDirectory;
106       nextId = &this->nextFreeDirectory;
107       break;
108     case mtfs::DATA_BLOCK:
```

```
109         mu = &this->blockMutex;
110         ve = &this->freeDatas;
111         nextId = &this->nextFreeData;
112         break;
113       default:
114         return ENOSYS;
115     }
116
117     unique_lock<mutex> lk(*mu);
118     if (ve->empty()) {
119       *id = *nextId;
120       (*nextId)++;
121     } else {
122       *id = (*ve)[0];
123       ve->erase(ve->begin());
124     }
125     lk.unlock();
126
127     return 0;
128   }
129
130   int S3::del(const uint64_t &id, const mtfs::blockType &type) {
131     mutex *mu = nullptr;
132     vector<uint64_t> *ve = nullptr;
133     uint64_t *nextId;
134
135     string key;
136
137     switch (type) {
138       case mtfs::INODE:
139         mu = &this->inodeMutex;
140         ve = &this->freeInodes;
141         nextId = &this->nextFreeInode;
142         key = INODE_PREFIX;
143         break;
144       case mtfs::DIR_BLOCK:
145         mu = &this->dirBlockMutex;
146         ve = &this->freeDirectory;
147         nextId = &this->nextFreeDirectory;
148         key = DIRECTORY_PREFIX;
149         break;
150       case mtfs::DATA_BLOCK:
151         mu = &this->blockMutex;
152         ve = &this->freeDatas;
153         nextId = &this->nextFreeData;
154         key = DATA_PREFIX;
155         break;
156       default:
157         return ENOSYS;
158     }
159
160     unique_lock<mutex> lk(*mu);
161     if (id == *nextId - 1) {
162       (*nextId)--;
163     } else {
164       ve->push_back(id);
165     }
166     lk.unlock();
167
```

```
168     key += to_string(id);
169     this->delObj(key);
170 
171     return 0;
172   }
173 
174   int S3::get(const uint64_t &id, void *data, const mtfs::blockType &type, bool
      metas) {
175     string filename = this->home, key;
176     switch (type) {
177       case mtfs::INODE:
178         filename += "in";
179         key += INODE_PREFIX;
180         break;
181       case mtfs::DIR_BLOCK:
182         filename += "di";
183         key += DIRECTORY_PREFIX;
184         break;
185       case mtfs::DATA_BLOCK:
186         filename += "da";
187         key += DATA_PREFIX;
188         break;
189       default:
190         return ENOSYS;
191     }
192 
193     if (metas) {
194       filename += "me";
195       key = "metas/" + key;
196     }
197 
198     key += to_string(id);
199 
200     if (0 != this->dlObj(filename, key))
201       return 1;
202 
203     if (mtfs::DATA_BLOCK == type) {
204       if (metas) {
205         ifstream file(filename);
206         if (!file.is_open())
207           return errno != 0 ? errno : ENOENT;
208 
209         IStreamWrapper wrapper(file);
210 
211         Document d;
212         d.ParseStream(wrapper);
213 
214         ((mtfs::blockInfo_t *) data)->fromJson(d);
215       } else {
216         ifstream file(filename);
217         if (!file.is_open())
218           return errno != 0 ? errno : ENOENT;
219 
220         file.read((char *) data, this->blocksize);
221         file.close();
222       }
223     } else {
224       ifstream file(filename);
225       if (!file.is_open())
```

```
226         return errno != 0 ? errno : ENOENT;
227
228       IStreamWrapper wrapper(file);
229
230       Document d;
231       d.ParseStream(wrapper);
232
233       switch (type) {
234         case mtfs::INODE:
235           if (metas)
236             ((mtfs::blockInfo_t *) data)->fromJson(d);
237           else
238             ((mtfs::inode_t *) data)->fromJson(d);
239           break;
240         case mtfs::DIR_BLOCK:
241           if (metas)
242             ((mtfs::blockInfo_t *) data)->fromJson(d);
243           else
244             ((mtfs::dirBlock_t *) data)->fromJson(d);
245           break;
246         case mtfs::SUPERBLOCK:
247           break;
248         default:
249           return ENOSYS;
250       }
251
252       file.close();
253     }
254
255     unlink(filename.c_str());
256     return 0;
257   }
258
259   int S3::put(const uint64_t &id, const void *data, const mtfs::blockType &type
      , bool metas) {
260     string filename = this->home, key;
261     switch (type) {
262       case mtfs::INODE:
263         filename += "in";
264         key += INODE_PREFIX;
265         break;
266       case mtfs::DIR_BLOCK:
267         filename += "di";
268         key += DIRECTORY_PREFIX;
269         break;
270       case mtfs::DATA_BLOCK:
271         filename += "da";
272         key += DATA_PREFIX;
273         break;
274       default:
275         return ENOSYS;
276     }
277
278     if (metas) {
279       filename += "me";
280       key = "metas/" + key;
281     }
282
283     key += to_string(id);
```

```
284
285     if (mtfs::DATA_BLOCK == type) {
286       if (metas) {
287         Document d;
288
289         ((mtfs::blockInfo_t *) data)->toJson(d);
290
291         StringBuffer sb;
292         PrettyWriter<StringBuffer> pw(sb);
293         d.Accept(pw);
294
295         ofstream file(filename);
296         if (!file.is_open())
297           return errno != 0 ? errno : ENOENT;
298         file << sb.GetString() << endl;
299         file.close();
300       } else {
301         ofstream file(filename);
302         if (!file.is_open())
303           return errno != 0 ? errno : ENOENT;
304
305         file.write((char *) data, this->blocksize);
306         file.close();
307       }
308     } else {
309       Document d;
310
311       switch (type) {
312         case mtfs::INODE:
313           if (metas)
314             ((mtfs::blockInfo_t *) data)->toJson(d);
315           else
316             ((mtfs::inode_t *) data)->toJson(d);
317           break;
318         case mtfs::DIR_BLOCK:
319           if (metas)
320             ((mtfs::blockInfo_t *) data)->toJson(d);
321           else
322             ((mtfs::dirBlock_t *) data)->toJson(d);
323           break;
324         case mtfs::SUPERBLOCK:
325           break;
326         default:
327           return ENOSYS;
328       }
329
330       StringBuffer sb;
331       PrettyWriter<StringBuffer> pw(sb);
332       d.Accept(pw);
333
334       ofstream file(filename);
335       if (!file.is_open())
336         return errno != 0 ? errno : ENOENT;
337       file << sb.GetString() << endl;
338       file.close();
339     }
340
341     if (0 != this->upObj(filename, key))
342       return 1;
```

```
343
344     unlink(filename.c_str());
345     return 0;
346   }
347
348   bool S3::getSuperblock(mtfs::superblock_t &superblock) {
349     return false;
350   }
351
352   bool S3::putSuperblock(const mtfs::superblock_t &superblock) {
353     return false;
354   }
355
356   int S3::writeTmpFile(const string &filename, const void *data, const mtfs::
      blockType &type) {
357     switch (type) {
358       case mtfs::INODE:
359         return this->writeInode(filename, (mtfs::inode_t *) data);
360         break;
361       case mtfs::DIR_BLOCK:
362         return this->writeDirBlock(filename, (mtfs::dirBlock_t *) data);
363         break;
364       case mtfs::DATA_BLOCK:
365         return this->writeDataBlock(filename, (uint8_t *) data);
366         break;
367       case mtfs::SUPERBLOCK:
368         return this->writeSuperblock(filename, (mtfs::superblock_t *) data);
369         break;
370       default:
371         return ENOSYS;
372     }
373   }
374
375   int S3::writeInode(const std::string &filename, const mtfs::inode_t *inode) {
376     Document d;
377     d.SetObject();
378     Document::AllocatorType &alloc = d.GetAllocator();
379
380     inode->toJson(d);
381
382     StringBuffer sb;
383     PrettyWriter<StringBuffer> pw(sb);
384     d.Accept(pw);
385
386     ofstream inodeFile(filename);
387     inodeFile << sb.GetString() << endl;
388     inodeFile.close();
389
390     return this->SUCCESS;
391   }
392
393   int S3::writeDirBlock(const std::string &filename, const mtfs::dirBlock_t *
      dirBlock) {
394     Document d;
395     d.SetObject();
396
397     Document::AllocatorType &allocator = d.GetAllocator();
398
399     for (auto &&item: dirBlock->entries) {
```

```
400         Value r(kArrayType);
401         for (auto &&ident: item.second) {
402           Value v(kObjectType);
403           ident.toJson(v, allocator);
404           r.PushBack(v, allocator);
405         }
406         d.AddMember(StringRef(item.first.c_str()), r, allocator);
407       }
408
409       StringBuffer sb;
410       PrettyWriter<StringBuffer> pw(sb);
411       d.Accept(pw);
412
413       ofstream file(filename);
414       if (!file.is_open())
415         return errno != 0 ? errno : ENOENT;
416       file << sb.GetString() << endl;
417       file.close();
418
419       return this->SUCCESS;
420     }
421
422     int S3::writeDataBlock(const std::string &filename, const uint8_t *datas) {
423       ofstream file(filename);
424       if (!file.is_open())
425         return errno != 0 ? errno : ENOENT;
426
427       file.write((const char *) datas, this->blocksize);
428       file.close();
429       return this->SUCCESS;
430     }
431
432     int S3::writeSuperblock(const std::string &filename, const mtfs::superblock_t
       *superblock) {
433       ofstream sbFile(filename, ios::binary);
434       sbFile.write((const char *) superblock, sizeof(*superblock));
435       sbFile.close();
436     }
437
438     int S3::dlObj(const std::string &filename, const std::string &key) {
439       Model::GetObjectRequest objectRequest;
440       objectRequest.WithBucket(this->bucket).WithKey(key.c_str());
441
442       auto getObjectOutcome = this->s3Client->GetObject(objectRequest);
443
444       if (getObjectOutcome.IsSuccess()) {
445         Aws::OFStream localFile;
446         localFile.open(filename, ios_base::out | ios_base::binary);
447         localFile << getObjectOutcome.GetResult().GetBody().rdbuf();
448         return 0;
449       }
450
451       return 1;
452     }
453
454     int S3::upObj(const std::string &filename, const std::string &key) {
455       Model::PutObjectRequest objectRequest;
456       objectRequest.WithBucket(this->bucket).WithKey(key.c_str());
457
```

```
458     auto inputData = Aws::MakeShared<Aws::FStream>("PutObjectInputStream",
459                                 filename.c_str(), std::ios_base::in);
460
461     objectRequest.SetBody(inputData);
462
463     auto putObjectOutcome = this->s3Client->PutObject(objectRequest);
464
465     if (putObjectOutcome.IsSuccess()) {
466 //    std::cout << "Done!" << std::endl;
467       return 0;
468     }
469     string message = string("PutObject error: ") + putObjectOutcome.GetError().
    GetExceptionName().c_str() + " " +
470               putObjectOutcome.GetError().GetMessage().c_str();
471     Logger::getInstance()->log("S3", message, Logger::L_ERROR);
472
473     return 1;
474   }
475
476   void S3::delObj(const string &key) {
477     Aws::S3::Model::DeleteObjectRequest objectRequest;
478     objectRequest.WithBucket(this->bucket).WithKey(key.c_str());
479
480     auto deleteObjectOutcome = this->s3Client->DeleteObject(objectRequest);
481
482     if (deleteObjectOutcome.IsSuccess()) {
483       Logger::getInstance()->log("S3", "Delete Success", Logger::L_INFO);
484     } else {
485       string message =
486           string("PutObject error: ") + deleteObjectOutcome.GetError().
    GetExceptionName().c_str() + " " +
487           deleteObjectOutcome.GetError().GetMessage().c_str();
488       Logger::getInstance()->log("S3", message, Logger::L_ERROR);
489     }
490   }
491
492   bool S3::dirExists(string path) {
493     struct stat info{};
494
495     if (stat(path.c_str(), &info) != 0)
496       return false;
497     return (info.st_mode & S_IFDIR) != 0;
498   }
499
500   void S3::initIds() {
501     string keyName = "ids.json";
502     string filename = this->home + keyName;
503
504     if (0 != dlObj(filename, keyName)) {
505       return;
506     }
507
508     ifstream inodeFile(filename);
509     if (!inodeFile.is_open())
510       return;
511
512     IStreamWrapper isw(inodeFile);
513
514     Document d;
```

```
515     d.ParseStream(isw);
516
517     assert(d.IsObject());
518
519 //  Init inode ids
520     assert(d.HasMember("inode"));
521     Value o = d["inode"].GetObject();
522     assert(o.HasMember("next"));
523     this->nextFreeInode = o["next"].GetUint64();
524     assert(o.HasMember("frees"));
525     for (auto &&id : o["frees"].GetArray()) {
526       this->freeInodes.push_back(id.GetUint64());
527     }
528
529 //  Init Dir block ids
530     assert(d.HasMember("directory"));
531     o = d["directory"].GetObject();
532     assert(o.HasMember("next"));
533     this->nextFreeDirectory = o["next"].GetUint64();
534     assert(o.HasMember("frees"));
535     for (auto &&id : o["frees"].GetArray()) {
536       this->freeDirectory.push_back(id.GetUint64());
537     }
538
539 //  Init data Block Ids
540     assert(d.HasMember("data"));
541     o = d["data"].GetObject();
542     assert(o.HasMember("next"));
543     this->nextFreeData = o["next"].GetUint64();
544     assert(o.HasMember("frees"));
545     for (auto &&id : o["frees"].GetArray()) {
546       this->freeDatas.push_back(id.GetUint64());
547     }
548
549
550     unlink(filename.c_str());
551
552     Logger::getInstance()->log("S3", "INIT SUCCESS", Logger::L_INFO);
553   }
554
555   void S3::writeMetas() {
556     Document d;
557     d.SetObject();
558
559     Document::AllocatorType &allocator = d.GetAllocator();
560
561     Value o(kObjectType);
562     Value a(kArrayType);
563     Value v;
564     v.SetUint64(this->nextFreeInode);
565     o.AddMember(StringRef("next"), v, allocator);
566     for (auto &&id :this->freeInodes) {
567       v.SetUint64(id);
568       a.PushBack(v, allocator);
569     }
570     o.AddMember(StringRef("frees"), a, allocator);
571     d.AddMember(StringRef("inode"), o, allocator);
572
573     o.SetObject();
```

```
574     a.SetArray();
575     v.SetUint64(this->nextFreeDirectory);
576     o.AddMember(StringRef("next"), v, allocator);
577     for (auto &&id :this->freeDirectory) {
578       v.SetUint64(id);
579       a.PushBack(v, allocator);
580     }
581     o.AddMember(StringRef("frees"), a, allocator);
582     d.AddMember(StringRef("directory"), o, allocator);
583 
584     o.SetObject();
585     a.SetArray();
586     v.SetUint64(this->nextFreeData);
587     o.AddMember(StringRef("next"), v, allocator);
588     for (auto &&id :this->freeDatas) {
589       v.SetUint64(id);
590       a.PushBack(v, allocator);
591     }
592     o.AddMember(StringRef("frees"), a, allocator);
593     d.AddMember(StringRef("data"), o, allocator);
594 
595     StringBuffer strBuff;
596     PrettyWriter<StringBuffer> writer(strBuff);
597     d.Accept(writer);
598 
599     string key = "ids.json";
600     string filename = this->home + key;
601     ofstream file;
602     file.open(filename);
603     file << strBuff.GetString() << endl;
604     file.close();
605 
606     this->upObj(filename, key);
607 
608     unlink(filename.c_str());
609   }
610 }
611 
612 extern "C" pluginSystem::Plugin *createObj() {
613   return new pluginSystem::S3();
614 }
615 
616 extern "C" void destroyObj(pluginSystem::Plugin *plugin) {
617   delete plugin;
618 }
619 
620 extern "C" pluginSystem::pluginInfo_t getInfo() {
621   vector<string> params;
622   params.emplace_back("bucket");
623   params.emplace_back("region");
624 
625   pluginSystem::pluginInfo_t info = {
626       .name = pluginSystem::S3::NAME,
627       .params = params,
628   };
629 
630   return info;
631 }
```

Listing 3.9 – "pluginSystem/S3.cpp

3.3 *Wrapper*

3.3.1 FuseBase

FuseBase.h

```
/**
 * \file FuseBase.h
 * \brief
 * \author David Wittwer
 * \version 0.0.1
 * \copyright GNU Publis License V3
 *
 * This file is part of MTFS.

    MTFS is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Foobar is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
 */

#ifndef MTFSFUSE_FUSE_BASE_H
#define MTFSFUSE_FUSE_BASE_H

#include <fuse3/fuse_lowlevel.h>
#include <string>
#include <vector>
#include <list>
#include <iostream>
#include <assert.h>

#include "wrapper/FuseCallback.h"


namespace wrapper {
  class FuseCallback;

  class FuseBase {

  public:
    FuseBase();

    virtual ~FuseBase() = 0;

    int run(int argc, char **argv);

```

```
49
50  protected:
51    friend class FuseCallback;
52
53    virtual bool runPrepare(int argc, char **argv)=0;
54
55    virtual void init(void *userdata, fuse_conn_info *conn);
56
57    virtual void destroy(void *userdata);
58
59    virtual void lookup(fuse_req_t req, fuse_ino_t parent, const char *name);
60
61    virtual void forget(fuse_req_t req, fuse_ino_t ino, uint64_t nlookup);
62
63    virtual void getAttr(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi);
64
65    virtual void setattr(fuse_req_t req, fuse_ino_t ino, struct stat *attr, int
      to_set, fuse_file_info *fi);
66
67    virtual void readlink(fuse_req_t req, fuse_ino_t ino);
68
69    virtual void mknod(fuse_req_t req, fuse_ino_t parent, const char *name,
    mode_t mode, dev_t rdev);
70
71    virtual void mkdir(fuse_req_t req, fuse_ino_t parent, const char *name,
    mode_t mode);
72
73    virtual void unlink(fuse_req_t req, fuse_ino_t parent, const char *name);
74
75    virtual void rmdir(fuse_req_t req, fuse_ino_t parent, const char *name);
76
77    virtual void symlink(fuse_req_t req, const char *link, fuse_ino_t parent,
    const char *name);
78
79    virtual void
80    rename(fuse_req_t req, fuse_ino_t parent, const char *name, fuse_ino_t
    newparent, const char *newname,
81          unsigned int flags);
82
83    virtual void link(fuse_req_t req, fuse_ino_t ino, fuse_ino_t newparent,
    const char *newname);
84
85    virtual void open(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi);
86
87    virtual void read(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
    fuse_file_info *fi);
88
89    virtual void write(fuse_req_t req, fuse_ino_t ino, const char *buf, size_t
    size, off_t off, fuse_file_info *fi);
90
91    virtual void flush(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi);
92
93    virtual void release(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi);
94
95    virtual void fsync(fuse_req_t req, fuse_ino_t ino, int datasync,
    fuse_file_info *fi);
96
97    virtual void opendir(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi);
98
```

```
99      virtual void readdir(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off
        , fuse_file_info *fi);
100
101     virtual void releasedir(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi)
        ;
102
103     virtual void fsyncdir(fuse_req_t req, fuse_ino_t ino, int datasync,
        fuse_file_info *fi);
104
105     virtual void statfs(fuse_req_t req, fuse_ino_t ino);
106
107     virtual void
108     setxattr(fuse_req_t req, fuse_ino_t ino, const char *name, const char *
        value, size_t size, int flags);
109
110     virtual void getxattr(fuse_req_t req, fuse_ino_t ino, const char *name,
        size_t size);
111
112     virtual void listxattr(fuse_req_t req, fuse_ino_t ino, size_t size);
113
114     virtual void removexattr(fuse_req_t req, fuse_ino_t ino, const char *name);
115
116     virtual void access(fuse_req_t req, fuse_ino_t ino, int mask);
117
118     virtual void create(fuse_req_t req, fuse_ino_t parent, const char *name,
        mode_t mode, fuse_file_info *fi);
119
120     virtual void getlk(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi,
        struct flock *lock);
121
122     virtual void setlk(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi,
        struct flock *lock, int sleep);
123
124     virtual void bmap(fuse_req_t req, fuse_ino_t ino, size_t blocksize,
        uint64_t idx);
125
126     virtual void ioctl(fuse_req_t req, fuse_ino_t ino, int cmd, void *arg,
        fuse_file_info *fi, unsigned int flags,
127                const void *in_buf, size_t in_bufsz, size_t out_bufsz);
128
129     virtual void poll(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi,
        fuse_pollhandle *ph);
130
131     virtual void write_buf(fuse_req_t req, fuse_ino_t ino, fuse_bufvec *bufv,
        off_t off, fuse_file_info *fi);
132
133     virtual void retrive_reply(fuse_req_t req, void *cookie, fuse_ino_t ino,
        off_t offset, fuse_bufvec *bufv);
134
135     virtual void forget_multi(fuse_req_t req, size_t count, fuse_forget_data *
        forgets);
136
137     virtual void flock(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi, int
        op);
138
139     virtual void
140     fallocate(fuse_req_t req, fuse_ino_t ino, int mode, off_t offset, off_t
        length, fuse_file_info *fi);
141
```

```
142     virtual void readdirplus(fuse_req_t req, fuse_ino_t ino, size_t size, off_t
        off, fuse_file_info *fi);
143
144   private:
145     FuseCallback *callbacks;
146
147   };
148
149 }  // namespace wrapper
150 #endif
```

Listing 3.10 – "wrapper/FuseBase.h"

FuseBase.cpp

```
1  /**
2   * \file FuseBase.cpp
3   * \brief
4   * \author David Wittwer
5   * \version 0.0.1
6   * \copyright GNU Publis License V3
7   *
8   * This file is part of MTFS.
9
10    MTFS is free software: you can redistribute it and/or modify
11    it under the terms of the GNU General Public License as published by
12    the Free Software Foundation, either version 3 of the License, or
13    (at your option) any later version.
14
15    Foobar is distributed in the hope that it will be useful,
16    but WITHOUT ANY WARRANTY; without even the implied warranty of
17    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18    GNU General Public License for more details.
19
20    You should have received a copy of the GNU General Public License
21    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22  */
23
24 #include <utils/Logger.h>
25 #include <fstream>
26 #include "wrapper/FuseBase.h"
27
28 namespace wrapper {
29
30   FuseBase::FuseBase() {
31     callbacks = FuseCallback::getInstance();
32   }
33
34   FuseBase::~FuseBase() = default;
35
36   int FuseBase::run(int argc, char **argv) {
37
38     struct fuse_args args = FUSE_ARGS_INIT(argc, argv);
39     struct fuse_session *se;
40     struct fuse_cmdline_opts opts{};
41     int ret = -1;
42
43     if (!runPrepare(argc, argv))
44       return -1;
```

```
45 //    std::cout << "run" << std::endl;
46
47     callbacks->setBase(this);
48
49     if (fuse_parse_cmdline(&args, &opts) != 0)
50       return 1;
51     if (0 != opts.show_help) {
52       std::cerr << "usage: " << argv[0] << " [options] <mountpoint> <configName
       >" << std::endl << std::endl;
53       fuse_cmdline_help();
54       fuse_lowlevel_help();
55       ret = 0;
56       goto err_out1;
57     } else if (0 != opts.show_version) {
58       std::cerr << "FUSE library version " << fuse_pkgversion() << std::endl;
59       fuse_lowlevel_version();
60       ret = 0;
61       goto err_out1;
62     }
63
64     if (0 != opts.debug) {
65       std::ofstream log("/tmp/mtfs.log");
66       Logger::init(std::cerr, Logger::L_DEBUG);
67     } else {
68       std::ofstream log;
69       log.open("/var/log/mtfs.log");
70       Logger::init(log, Logger::L_ERROR);
71     }
72
73     se = fuse_session_new(&args, &FuseCallback::ops, sizeof(FuseCallback::ops),
       nullptr);
74
75     if (se == nullptr)
76       goto err_out1;
77
78     if (fuse_set_signal_handlers(se) != 0)
79       goto err_out2;
80
81     if (fuse_session_mount(se, opts.mountpoint) != 0)
82       goto err_out3;
83
84     fuse_daemonize(opts.foreground);
85
86     /* Block until ctrl+c or fusermount -u */
87     if (0 != opts.singlethread)
88       ret = fuse_session_loop(se);
89     else
90       ret = fuse_session_loop_mt(se, opts.clone_fd);
91
92     fuse_session_unmount(se);
93     err_out3:
94     fuse_remove_signal_handlers(se);
95     err_out2:
96     fuse_session_destroy(se);
97     err_out1:
98     free(opts.mountpoint);
99     fuse_opt_free_args(&args);
100
101    return 0 != ret ? 1 : 0;
```

```
102   }
103
104   void FuseBase::init(void *userdata, fuse_conn_info *conn) {
105     (void) userdata, conn;
106   }
107
108   void FuseBase::destroy(void *userdata) {
109     (void) userdata;
110   }
111
112   void FuseBase::lookup(fuse_req_t req, fuse_ino_t parent, const char *name) {
113     (void) parent, name;
114     fuse_reply_err(req, ENOSYS);
115   }
116
117   void FuseBase::forget(fuse_req_t req, fuse_ino_t ino, uint64_t nlookup) {
118     (void) ino, nlookup;
119     fuse_reply_err(req, ENOSYS);
120   }
121
122   void FuseBase::getAttr(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) {
123     (void) ino, fi;
124     fuse_reply_err(req, ENOSYS);
125   }
126
127   void FuseBase::setattr(fuse_req_t req, fuse_ino_t ino, struct stat *attr, int
        to_set, fuse_file_info *fi) {
128     (void) ino, attr, to_set, fi;
129     fuse_reply_err(req, ENOSYS);
130   }
131
132   void FuseBase::readlink(fuse_req_t req, fuse_ino_t ino) {
133     (void) ino;
134     fuse_reply_err(req, ENOSYS);
135   }
136
137   void FuseBase::mknod(fuse_req_t req, fuse_ino_t parent, const char *name,
       mode_t mode, dev_t rdev) {
138     (void) parent, name, mode, rdev;
139     fuse_reply_err(req, ENOSYS);
140   }
141
142   void FuseBase::mkdir(fuse_req_t req, fuse_ino_t parent, const char *name,
       mode_t mode) {
143     (void) parent, name, mode;
144     fuse_reply_err(req, ENOSYS);
145   }
146
147   void FuseBase::unlink(fuse_req_t req, fuse_ino_t parent, const char *name) {
148     (void) parent, name;
149     fuse_reply_err(req, ENOSYS);
150   }
151
152   void FuseBase::rmdir(fuse_req_t req, fuse_ino_t parent, const char *name) {
153     (void) parent, name;
154     fuse_reply_err(req, ENOSYS);
155   }
156
```

```
157  void FuseBase::symlink(fuse_req_t req, const char *link, fuse_ino_t parent,
      const char *name) {
158    (void) link, parent, name;
159    fuse_reply_err(req, ENOSYS);
160  }
161
162  void
163  FuseBase::rename(fuse_req_t req, fuse_ino_t parent, const char *name,
      fuse_ino_t newparent, const char *newname,
164            unsigned int flags) {
165    (void) parent, name, newparent, newname, flags;
166    fuse_reply_err(req, ENOSYS);
167  }
168
169  void FuseBase::link(fuse_req_t req, fuse_ino_t ino, fuse_ino_t newparent,
      const char *newname) {
170    (void) ino, newparent, newname;
171    fuse_reply_err(req, ENOSYS);
172  }
173
174  void FuseBase::open(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) {
175    (void) ino, fi;
176    fuse_reply_err(req, ENOSYS);
177  }
178
179  void FuseBase::read(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
      fuse_file_info *fi) {
180    (void) ino, size, off, fi;
181    fuse_reply_err(req, ENOSYS);
182  }
183
184  void FuseBase::write(fuse_req_t req, fuse_ino_t ino, const char *buf, size_t
      size, off_t off, fuse_file_info *fi) {
185    (void) ino, buf, size, off, fi;
186    fuse_reply_err(req, ENOSYS);
187  }
188
189  void FuseBase::flush(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) {
190    (void) ino, fi;
191    fuse_reply_err(req, ENOSYS);
192  }
193
194  void FuseBase::release(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) {
195    (void) ino, fi;
196    fuse_reply_err(req, ENOSYS);
197  }
198
199  void FuseBase::fsync(fuse_req_t req, fuse_ino_t ino, int datasync,
      fuse_file_info *fi) {
200    (void) ino, datasync, fi;
201    fuse_reply_err(req, ENOSYS);
202  }
203
204  void FuseBase::opendir(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) {
205    (void) ino, fi;
206    fuse_reply_err(req, ENOSYS);
207  }
208
```

```
209  void FuseBase::readdir(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off
       , fuse_file_info *fi) {
210    (void) ino, size, off, fi;
211    fuse_reply_err(req, ENOSYS);
212  }
213
214  void FuseBase::releasedir(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi)
       {
215    (void) ino, fi;
216    fuse_reply_err(req, ENOSYS);
217  }
218
219  void FuseBase::fsyncdir(fuse_req_t req, fuse_ino_t ino, int datasync,
       fuse_file_info *fi) {
220    (void) ino, datasync, fi;
221    fuse_reply_err(req, ENOSYS);
222  }
223
224  void FuseBase::statfs(fuse_req_t req, fuse_ino_t ino) {
225    (void) ino;
226    fuse_reply_err(req, ENOSYS);
227  }
228
229  void
230  FuseBase::setxattr(fuse_req_t req, fuse_ino_t ino, const char *name, const
       char *value, size_t size, int flags) {
231    (void) ino, name, value, size, flags;
232    fuse_reply_err(req, ENOSYS);
233  }
234
235  void FuseBase::getxattr(fuse_req_t req, fuse_ino_t ino, const char *name,
       size_t size) {
236    (void) ino, name, size;
237    fuse_reply_err(req, ENOSYS);
238  }
239
240  void FuseBase::listxattr(fuse_req_t req, fuse_ino_t ino, size_t size) {
241    (void) ino, size;
242    fuse_reply_err(req, ENOSYS);
243  }
244
245  void FuseBase::removexattr(fuse_req_t req, fuse_ino_t ino, const char *name)
       {
246    (void) ino, name;
247    fuse_reply_err(req, ENOSYS);
248  }
249
250  void FuseBase::access(fuse_req_t req, fuse_ino_t ino, int mask) {
251    (void) ino, mask;
252    fuse_reply_err(req, ENOSYS);
253  }
254
255  void FuseBase::create(fuse_req_t req, fuse_ino_t parent, const char *name,
       mode_t mode, fuse_file_info *fi) {
256    (void) parent, name, mode, fi;
257    fuse_reply_err(req, ENOSYS);
258  }
259
```

```
260 void FuseBase::getlk(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi,
    struct flock *lock) {
261   (void) ino, fi, lock;
262   fuse_reply_err(req, ENOSYS);
263 }
264
265 void FuseBase::setlk(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi,
    struct flock *lock, int sleep) {
266   (void) ino, fi, lock, sleep;
267   fuse_reply_err(req, ENOSYS);
268 }
269
270 void FuseBase::bmap(fuse_req_t req, fuse_ino_t ino, size_t blocksize,
    uint64_t idx) {
271   (void) ino, blocksize, idx;
272   fuse_reply_err(req, ENOSYS);
273 }
274
275 void FuseBase::ioctl(fuse_req_t req, fuse_ino_t ino, int cmd, void *arg,
    fuse_file_info *fi, unsigned int flags,
276             const void *in_buf, size_t in_bufsz, size_t out_bufsz) {
277   (void) ino, cmd, arg, fi, flags, in_buf, in_bufsz, out_bufsz;
278   fuse_reply_err(req, ENOSYS);
279 }
280
281 void FuseBase::poll(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi,
    fuse_pollhandle *ph) {
282   (void) ino, fi, ph;
283   fuse_reply_err(req, ENOSYS);
284 }
285
286 void FuseBase::write_buf(fuse_req_t req, fuse_ino_t ino, fuse_bufvec *bufv,
    off_t off, fuse_file_info *fi) {
287   (void) ino, bufv, off, fi;
288   fuse_reply_err(req, ENOSYS);
289 }
290
291 void FuseBase::retrive_reply(fuse_req_t req, void *cookie, fuse_ino_t ino,
    off_t offset, fuse_bufvec *bufv) {
292   (void) cookie, ino, offset, bufv;
293   fuse_reply_err(req, ENOSYS);
294 }
295
296 void FuseBase::forget_multi(fuse_req_t req, size_t count, fuse_forget_data *
    forgets) {
297   (void) count, forgets;
298   fuse_reply_err(req, ENOSYS);
299 }
300
301 void FuseBase::flock(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi, int
    op) {
302   (void) ino, fi, op;
303   fuse_reply_err(req, ENOSYS);
304 }
305
306 void FuseBase::fallocate(fuse_req_t req, fuse_ino_t ino, int mode, off_t
    offset, off_t length, fuse_file_info *fi) {
307   (void) ino, mode, offset, length, fi;
308   fuse_reply_err(req, ENOSYS);
```

```
309   }
310
311   void FuseBase::readdirplus(fuse_req_t req, fuse_ino_t ino, size_t size, off_t
        off, fuse_file_info *fi) {
312     (void) ino, size, off, fi;
313     fuse_reply_err(req, ENOSYS);
314   }
315
316
317 }  // namespace wrapper
```

Listing 3.11 – "wrapper/FuseBase.cpp"

3.3.2 FuseCallback

FuseCallback.h

```
1  /**
2   * \file FuseCallback.h
3   * \brief
4   * \author David Wittwer
5   * \version 0.0.1
6   * \copyright GNU Publis License V3
7   *
8   * This file is part of MTFS.
9
10    MTFS is free software: you can redistribute it and/or modify
11    it under the terms of the GNU General Public License as published by
12    the Free Software Foundation, either version 3 of the License, or
13    (at your option) any later version.
14
15    Foobar is distributed in the hope that it will be useful,
16    but WITHOUT ANY WARRANTY; without even the implied warranty of
17    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18    GNU General Public License for more details.
19
20    You should have received a copy of the GNU General Public License
21    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22  */
23
24 #ifndef MTFSFUSE_FUSE_CALLBACK_H
25 #define MTFSFUSE_FUSE_CALLBACK_H
26
27 #include <fuse3/fuse_lowlevel.h>
28 #include <string>
29 #include <vector>
30 #include <list>
31 #include <iostream>
32
33 #include "wrapper/FuseBase.h"
34
35
36 namespace wrapper {
37   class FuseBase;
38
39   class FuseCallback {
40   public:
41     static FuseCallback *getInstance();
```

```
42
43      void setBase(FuseBase *base);
44
45      static struct fuse_lowlevel_ops ops;
46
47
48    private:
49      FuseCallback();
50
51      static void init(void *userdata, struct fuse_conn_info *conn);
52
53      static void destroy(void *userdata);
54
55      static void lookup(fuse_req_t req, fuse_ino_t parent, const char *name);
56
57      static void forget(fuse_req_t req, fuse_ino_t ino, std::uint64_t nlookup);
58
59      static void getAttr(fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *
        fi);
60
61      static void setattr(fuse_req_t req, fuse_ino_t ino, struct stat *attr, int
        to_set, struct fuse_file_info *fi);
62
63      static void readlink(fuse_req_t req, fuse_ino_t ino);
64
65      static void mknod(fuse_req_t req, fuse_ino_t parent, const char *name,
        mode_t mode, dev_t rdev);
66
67      static void mkdir(fuse_req_t req, fuse_ino_t parent, const char *name,
        mode_t mode);
68
69      static void unlink(fuse_req_t req, fuse_ino_t parent, const char *name);
70
71      static void rmdir(fuse_req_t req, fuse_ino_t parent, const char *name);
72
73      static void symlink(fuse_req_t req, const char *link, fuse_ino_t parent,
        const char *name);
74
75      static void
76      rename(fuse_req_t req, fuse_ino_t parent, const char *name, fuse_ino_t
        newparent, const char *newname,
77             unsigned int flags);
78
79      static void link(fuse_req_t req, fuse_ino_t ino, fuse_ino_t newparent,
        const char *newname);
80
81      static void open(fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi)
        ;
82
83      static void read(fuse_req_t req, fuse_ino_t ino, std::size_t size, off_t
        off, struct fuse_file_info *fi);
84
85      static void
86      write(fuse_req_t req, fuse_ino_t ino, const char *buf, std::size_t size,
        off_t off, struct fuse_file_info *fi);
87
88      static void flush(fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi
        );
89
```

```
90      static void release(fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *
        fi);
91
92      static void fsync(fuse_req_t req, fuse_ino_t ino, int datasync, struct
        fuse_file_info *fi);
93
94      static void opendir(fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *
        fi);
95
96      static void readdir(fuse_req_t req, fuse_ino_t ino, std::size_t size, off_t
         off, struct fuse_file_info *fi);
97
98      static void releasedir(fuse_req_t req, fuse_ino_t ino, struct
        fuse_file_info *fi);
99
100     static void fsyncdir(fuse_req_t req, fuse_ino_t ino, int datasync, struct
        fuse_file_info *fi);
101
102     static void statfs(fuse_req_t req, fuse_ino_t ino);
103
104     static void
105     setxattr(fuse_req_t req, fuse_ino_t ino, const char *name, const char *
        value, std::size_t size, int flags);
106
107     static void getxattr(fuse_req_t req, fuse_ino_t ino, const char *name, std
        ::size_t size);
108
109     static void listxattr(fuse_req_t req, fuse_ino_t ino, std::size_t size);
110
111     static void removexattr(fuse_req_t req, fuse_ino_t ino, const char *name);
112
113     static void access(fuse_req_t req, fuse_ino_t ino, int mask);
114
115     static void create(fuse_req_t req, fuse_ino_t parent, const char *name,
        mode_t mode, struct fuse_file_info *fi);
116
117     static void getlk(fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi
        , struct flock *lock);
118
119     static void setlk(fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi
        , struct flock *lock, int sleep);
120
121     static void bmap(fuse_req_t req, fuse_ino_t ino, std::size_t blocksize, std
        ::uint64_t idx);
122
123     static void
124     ioctl(fuse_req_t req, fuse_ino_t ino, int cmd, void *arg, struct
        fuse_file_info *fi, unsigned int flags,
125         const void *in_buf, std::size_t in_bufsz, std::size_t out_bufsz);
126
127     static void poll(fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi,
         struct fuse_pollhandle *ph);
128
129     static void
130     write_buf(fuse_req_t req, fuse_ino_t ino, struct fuse_bufvec *bufv, off_t
        off, struct fuse_file_info *fi);
131
132     static void retrive_reply(fuse_req_t req, void *cookie, fuse_ino_t ino,
        off_t offset, struct fuse_bufvec *bufv);
```

```
133
134     static void forget_multi(fuse_req_t req, std::size_t count, struct
        fuse_forget_data *forgets);
135
136     static void flock(fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi
        , int op);
137
138     static void
139     fallocate(fuse_req_t req, fuse_ino_t ino, int mode, off_t offset, off_t
        length, struct fuse_file_info *fi);
140
141     static void readdirplus(fuse_req_t req, fuse_ino_t ino, std::size_t size,
        off_t off, struct fuse_file_info *fi);
142
143
144     static FuseCallback *self;
145
146     static FuseBase *base;
147
148   };
149
150 }  // namespace wrapper
151 #endif
```

Listing 3.12 – "wrapper/FuseCallback.h"

FuseCallback.cpp

```
1  #include <string>
2  #include <vector>
3  #include <list>
4  /**
5   * \file FuseCallback.cpp
6   * \brief
7   * \author David Wittwer
8   * \version 0.0.1
9   * \copyright GNU Publis License V3
10  *
11  * This file is part of MTFS.
12
13     MTFS is free software: you can redistribute it and/or modify
14     it under the terms of the GNU General Public License as published by
15     the Free Software Foundation, either version 3 of the License, or
16     (at your option) any later version.
17
18     Foobar is distributed in the hope that it will be useful,
19     but WITHOUT ANY WARRANTY; without even the implied warranty of
20     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
21     GNU General Public License for more details.
22
23     You should have received a copy of the GNU General Public License
24     along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
25  */
26
27 #include <iostream>
28 #include <assert.h>
29
30 #include <wrapper/FuseCallback.h>
31
```

```
32 namespace wrapper {
33   FuseCallback *FuseCallback::self = 0;
34   FuseBase *FuseCallback::base = 0;
35
36   struct fuse_lowlevel_ops FuseCallback::ops =
37       {
38           .init = FuseCallback::init,
39           .destroy=FuseCallback::destroy,
40           .lookup=FuseCallback::lookup,
41           .forget=FuseCallback::forget,
42           .getattr=FuseCallback::getAttr,
43           .setattr=FuseCallback::setattr,
44           .readlink=FuseCallback::readlink,
45           .mknod=FuseCallback::mknod,
46           .mkdir=FuseCallback::mkdir,
47           .unlink=FuseCallback::unlink,
48           .rmdir=FuseCallback::rmdir,
49           .symlink=FuseCallback::symlink,
50           .rename=FuseCallback::rename,
51           .link=FuseCallback::link,
52           .open=FuseCallback::open,
53           .read=FuseCallback::read,
54           .write=FuseCallback::write,
55           .flush=FuseCallback::flush,
56           .release=FuseCallback::release,
57           .fsync=FuseCallback::fsync,
58           .opendir=FuseCallback::opendir,
59           .readdir=FuseCallback::readdir,
60           .releasedir=FuseCallback::releasedir,
61           .fsyncdir=FuseCallback::fsyncdir,
62           .statfs=FuseCallback::statfs,
63           .setxattr=FuseCallback::setxattr,
64           .getxattr=FuseCallback::getxattr,
65           .listxattr=FuseCallback::listxattr,
66           .removexattr=FuseCallback::removexattr,
67           .access=FuseCallback::access,
68           .create=FuseCallback::create,
69           .getlk=FuseCallback::getlk,
70           .setlk=FuseCallback::setlk,
71           .bmap=FuseCallback::bmap,
72           .ioctl=FuseCallback::ioctl,
73           .poll=FuseCallback::poll,
74           .write_buf=FuseCallback::write_buf,
75           .retrieve_reply=FuseCallback::retrive_reply,
76           .forget_multi=FuseCallback::forget_multi,
77           .flock=FuseCallback::flock,
78           .fallocate=FuseCallback::fallocate,
79           .readdirplus=FuseCallback::readdirplus,
80       };
81
82   FuseCallback::FuseCallback() {
83     this->base = 0;
84   }
85
86   FuseCallback *FuseCallback::getInstance() {
87     if (!self)
88       self = new FuseCallback();
89
90     return self;
```

```
91  }
92
93  void FuseCallback::setBase(FuseBase *base) {
94    this->base = base;
95  }
96
97  void FuseCallback::init(void *userdata, struct fuse_conn_info *conn) {
98    if (base)
99      base->init(userdata, conn);
100 }
101
102 void FuseCallback::destroy(void *userdata) {
103   if (base)
104     base->destroy(userdata);
105 }
106
107 void FuseCallback::lookup(fuse_req_t req, fuse_ino_t parent, const char *name
      ) {
108   if (base)
109     base->lookup(req, parent, name);
110   else
111     fuse_reply_err(req, ENOSYS);
112 }
113
114 void FuseCallback::forget(fuse_req_t req, fuse_ino_t ino, uint64_t nlookup) {
115   if (base)
116     base->forget(req, ino, nlookup);
117   else
118     fuse_reply_err(req, ENOSYS);
119 }
120
121 void FuseCallback::getAttr(fuse_req_t req, fuse_ino_t ino, struct
      fuse_file_info *fi) {
122   if (base)
123     base->getAttr(req, ino, fi);
124   else
125     fuse_reply_err(req, ENOSYS);
126 }
127
128 void
129 FuseCallback::setattr(fuse_req_t req, fuse_ino_t ino, struct stat *attr, int
      to_set, struct fuse_file_info *fi) {
130   if (base)
131     base->setattr(req, ino, attr, to_set, fi);
132   else
133     fuse_reply_err(req, ENOSYS);
134 }
135
136 void FuseCallback::readlink(fuse_req_t req, fuse_ino_t ino) {
137   if (base)
138     base->readlink(req, ino);
139   else
140     fuse_reply_err(req, ENOSYS);
141 }
142
143 void FuseCallback::mknod(fuse_req_t req, fuse_ino_t parent, const char *name,
      mode_t mode, dev_t rdev) {
144   if (base)
145     base->mknod(req, parent, name, mode, rdev);
```

```
146       else
147         fuse_reply_err(req, ENOSYS);
148     }
149
150     void FuseCallback::mkdir(fuse_req_t req, fuse_ino_t parent, const char *name,
          mode_t mode) {
151       if (base)
152         base->mkdir(req, parent, name, mode);
153       else
154         fuse_reply_err(req, ENOSYS);
155     }
156
157     void FuseCallback::unlink(fuse_req_t req, fuse_ino_t parent, const char *name
          ) {
158       if (base)
159         base->unlink(req, parent, name);
160       else
161         fuse_reply_err(req, ENOSYS);
162     }
163
164     void FuseCallback::rmdir(fuse_req_t req, fuse_ino_t parent, const char *name)
          {
165       if (base)
166         base->rmdir(req, parent, name);
167       else
168         fuse_reply_err(req, ENOSYS);
169     }
170
171     void FuseCallback::symlink(fuse_req_t req, const char *link, fuse_ino_t
          parent, const char *name) {
172       if (base)
173         base->symlink(req, link, parent, name);
174       else
175         fuse_reply_err(req, ENOSYS);
176     }
177
178     void
179     FuseCallback::rename(fuse_req_t req, fuse_ino_t parent, const char *name,
          fuse_ino_t newparent, const char *newname,
180               unsigned int flags) {
181       if (base)
182         base->rename(req, parent, name, newparent, newname, flags);
183       else
184         fuse_reply_err(req, ENOSYS);
185     }
186
187     void FuseCallback::link(fuse_req_t req, fuse_ino_t ino, fuse_ino_t newparent,
          const char *newname) {
188       if (base)
189         base->link(req, ino, newparent, newname);
190       else
191         fuse_reply_err(req, ENOSYS);
192     }
193
194     void FuseCallback::open(fuse_req_t req, fuse_ino_t ino, struct fuse_file_info
          *fi) {
195       if (base)
196         base->open(req, ino, fi);
197       else
```

```
198       fuse_reply_err(req, ENOSYS);
199   }
200
201   void FuseCallback::read(fuse_req_t req, fuse_ino_t ino, size_t size, off_t
      off, struct fuse_file_info *fi) {
202     if (base)
203       base->read(req, ino, size, off, fi);
204     else
205       fuse_reply_err(req, ENOSYS);
206   }
207
208   void FuseCallback::write(fuse_req_t req, fuse_ino_t ino, const char *buf,
      size_t size, off_t off,
209                struct fuse_file_info *fi) {
210     if (base)
211       base->write(req, ino, buf, size, off, fi);
212     else
213       fuse_reply_err(req, ENOSYS);
214   }
215
216   void FuseCallback::flush(fuse_req_t req, fuse_ino_t ino, struct
      fuse_file_info *fi) {
217     if (base)
218       base->flush(req, ino, fi);
219     else
220       fuse_reply_err(req, ENOSYS);
221   }
222
223   void FuseCallback::release(fuse_req_t req, fuse_ino_t ino, struct
      fuse_file_info *fi) {
224     if (base)
225       base->release(req, ino, fi);
226     else
227       fuse_reply_err(req, ENOSYS);
228   }
229
230   void FuseCallback::fsync(fuse_req_t req, fuse_ino_t ino, int datasync, struct
       fuse_file_info *fi) {
231     if (base)
232       base->fsync(req, ino, datasync, fi);
233     else
234       fuse_reply_err(req, ENOSYS);
235   }
236
237   void FuseCallback::opendir(fuse_req_t req, fuse_ino_t ino, struct
      fuse_file_info *fi) {
238     if (base)
239       base->opendir(req, ino, fi);
240     else
241       fuse_reply_err(req, ENOSYS);
242   }
243
244   void FuseCallback::readdir(fuse_req_t req, fuse_ino_t ino, size_t size, off_t
       off, struct fuse_file_info *fi) {
245     if (base)
246       base->readdir(req, ino, size, off, fi);
247     else
248       fuse_reply_err(req, ENOSYS);
249   }
```

```
250
251   void FuseCallback::releasedir(fuse_req_t req, fuse_ino_t ino, struct
       fuse_file_info *fi) {
252     if (base)
253       base->releasedir(req, ino, fi);
254     else
255       fuse_reply_err(req, ENOSYS);
256   }
257
258   void FuseCallback::fsyncdir(fuse_req_t req, fuse_ino_t ino, int datasync,
       struct fuse_file_info *fi) {
259     if (base)
260       base->fsyncdir(req, ino, datasync, fi);
261     else
262       fuse_reply_err(req, ENOSYS);
263   }
264
265   void FuseCallback::statfs(fuse_req_t req, fuse_ino_t ino) {
266     if (base)
267       base->statfs(req, ino);
268     else
269       fuse_reply_err(req, ENOSYS);
270   }
271
272   void FuseCallback::setxattr(fuse_req_t req, fuse_ino_t ino, const char *name,
       const char *value, size_t size,
273                 int flags) {
274     if (base)
275       base->setxattr(req, ino, name, value, size, flags);
276     else
277       fuse_reply_err(req, ENOSYS);
278   }
279
280   void FuseCallback::getxattr(fuse_req_t req, fuse_ino_t ino, const char *name,
       size_t size) {
281     if (base)
282       base->getxattr(req, ino, name, size);
283     else
284       fuse_reply_err(req, ENOSYS);
285   }
286
287   void FuseCallback::listxattr(fuse_req_t req, fuse_ino_t ino, size_t size) {
288     if (base)
289       base->listxattr(req, ino, size);
290     else
291       fuse_reply_err(req, ENOSYS);
292   }
293
294   void FuseCallback::removexattr(fuse_req_t req, fuse_ino_t ino, const char *
       name) {
295     if (base)
296       base->removexattr(req, ino, name);
297     else
298       fuse_reply_err(req, ENOSYS);
299   }
300
301   void FuseCallback::access(fuse_req_t req, fuse_ino_t ino, int mask) {
302     if (base)
303       base->access(req, ino, mask);
```

```
304       else
305         fuse_reply_err(req, ENOSYS);
306     }
307
308     void
309     FuseCallback::create(fuse_req_t req, fuse_ino_t parent, const char *name,
        mode_t mode, struct fuse_file_info *fi) {
310       if (base)
311         base->create(req, parent, name, mode, fi);
312       else
313         fuse_reply_err(req, ENOSYS);
314     }
315
316     void FuseCallback::getlk(fuse_req_t req, fuse_ino_t ino, struct
        fuse_file_info *fi, struct flock *lock) {
317       if (base)
318         base->getlk(req, ino, fi, lock);
319       else
320         fuse_reply_err(req, ENOSYS);
321     }
322
323     void FuseCallback::setlk(fuse_req_t req, fuse_ino_t ino, struct
        fuse_file_info *fi, struct flock *lock, int sleep) {
324       if (base)
325         base->setlk(req, ino, fi, lock, sleep);
326       else
327         fuse_reply_err(req, ENOSYS);
328     }
329
330     void FuseCallback::bmap(fuse_req_t req, fuse_ino_t ino, size_t blocksize,
        uint64_t idx) {
331       if (base)
332         base->bmap(req, ino, blocksize, idx);
333       else
334         fuse_reply_err(req, ENOSYS);
335     }
336
337     void FuseCallback::ioctl(fuse_req_t req, fuse_ino_t ino, int cmd, void *arg,
        struct fuse_file_info *fi,
338                 unsigned int flags,
339                 const void *in_buf, size_t in_bufsz, size_t out_bufsz) {
340       if (base)
341         base->ioctl(req, ino, cmd, arg, fi, flags, in_buf, in_bufsz, out_bufsz);
342       else
343         fuse_reply_err(req, ENOSYS);
344     }
345
346     void FuseCallback::poll(fuse_req_t req, fuse_ino_t ino, struct fuse_file_info
        *fi, struct fuse_pollhandle *ph) {
347       if (base)
348         base->poll(req, ino, fi, ph);
349       else
350         fuse_reply_err(req, ENOSYS);
351     }
352
353     void FuseCallback::write_buf(fuse_req_t req, fuse_ino_t ino, struct
        fuse_bufvec *bufv, off_t off,
354                  struct fuse_file_info *fi) {
355       if (base)
```

```
356       base->write_buf(req, ino, bufv, off, fi);
357     else
358       fuse_reply_err(req, ENOSYS);
359   }
360
361   void
362   FuseCallback::retrive_reply(fuse_req_t req, void *cookie, fuse_ino_t ino,
      off_t offset, struct fuse_bufvec *bufv) {
363     if (base)
364       base->retrive_reply(req, cookie, ino, offset, bufv);
365     else
366       fuse_reply_err(req, ENOSYS);
367   }
368
369   void FuseCallback::forget_multi(fuse_req_t req, size_t count, struct
      fuse_forget_data *forgets) {
370     if (base)
371       base->forget_multi(req, count, forgets);
372     else
373       fuse_reply_err(req, ENOSYS);
374   }
375
376   void FuseCallback::flock(fuse_req_t req, fuse_ino_t ino, struct
      fuse_file_info *fi, int op) {
377     if (base)
378       base->flock(req, ino, fi, op);
379     else
380       fuse_reply_err(req, ENOSYS);
381   }
382
383   void
384   FuseCallback::fallocate(fuse_req_t req, fuse_ino_t ino, int mode, off_t
      offset, off_t length,
385               struct fuse_file_info *fi) {
386     if (base)
387       base->fallocate(req, ino, mode, offset, length, fi);
388     else
389       fuse_reply_err(req, ENOSYS);
390   }
391
392   void
393   FuseCallback::readdirplus(fuse_req_t req, fuse_ino_t ino, size_t size, off_t
      off, struct fuse_file_info *fi) {
394     if (base)
395       base->readdirplus(req, ino, size, off, fi);
396     else
397       fuse_reply_err(req, ENOSYS);
398   }
399
400
401 }  // namespace wrapper
```

Listing 3.13 – "wrapper/FuseCallback.cpp"

3.3.3 MtfsFuse

MtfsFuse.h

```
/**
 * \file MtfsFuse.h
 * \brief
 * \author David Wittwer
 * \version 0.0.1
 * \copyright GNU Publis License V3
 *
 * This file is part of MTFS.

    MTFS is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Foobar is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
 */

#ifndef MTFSFUSE_MTFS_FUSE_H
#define MTFSFUSE_MTFS_FUSE_H

#include <string>
#include <vector>
#include <list>
#include <iostream>
#include <cassert>

#include "wrapper/FuseBase.h"

namespace wrapper {
  class MtfsFuse : public FuseBase {

  public:

    ~MtfsFuse() override;

  protected:
    bool runPrepare(int argc, char **argv) override;


    void init(void *userdata, fuse_conn_info *conn) override;

    void destroy(void *userdata) override;

    void lookup(fuse_req_t req, fuse_ino_t parent, const char *name) override;

    void getAttr(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) override;

    void setattr(fuse_req_t req, fuse_ino_t ino, struct stat *attr, int to_set,
     fuse_file_info *fi) override;

    void mknod(fuse_req_t req, fuse_ino_t parent, const char *name, mode_t mode
    , dev_t rdev) override;

```

```
58      void open(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) override;
59
60      void release(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) override;
61
62      void opendir(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) override;
63
64      void access(fuse_req_t req, fuse_ino_t ino, int mask) override;
65
66      void readdirplus(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
        fuse_file_info *fi) override;
67
68      void releasedir(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi)
        override;
69
70      void readdir(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
        fuse_file_info *fi) override;
71
72      void mkdir(fuse_req_t req, fuse_ino_t parent, const char *name, mode_t mode
        ) override;
73
74      void
75      write(fuse_req_t req, fuse_ino_t ino, const char *buf, size_t size, off_t
        off, fuse_file_info *fi) override;
76
77      void write_buf(fuse_req_t req, fuse_ino_t ino, fuse_bufvec *bufv, off_t off
        , fuse_file_info *fi) override;
78
79      void read(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
        fuse_file_info *fi) override;
80
81      void unlink(fuse_req_t req, fuse_ino_t parent, const char *name) override;
82
83      void rmdir(fuse_req_t req, fuse_ino_t parent, const char *name) override;
84    };
85
86  }  // namespace wrapper
87  #endif
```

Listing 3.14 – "wrapper/MtfsFuse.h"

MtfsFuse.cpp

```
1   /**
2    * \file MtfsFuse.cpp
3    * \brief
4    * \author David Wittwer
5    * \version 0.0.1
6    * \copyright GNU Publis License V3
7    *
8    * This file is part of MTFS.
9
10      MTFS is free software: you can redistribute it and/or modify
11      it under the terms of the GNU General Public License as published by
12      the Free Software Foundation, either version 3 of the License, or
13      (at your option) any later version.
14
15      Foobar is distributed in the hope that it will be useful,
16      but WITHOUT ANY WARRANTY; without even the implied warranty of
17      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
```

```
18      GNU General Public License for more details.
19
20      You should have received a copy of the GNU General Public License
21      along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22   */
23
24  #include <string>
25  #include <mtfs/Mtfs.h>
26
27  #include "wrapper/MtfsFuse.h"
28
29
30  namespace wrapper {
31
32
33    MtfsFuse::~MtfsFuse() {
34
35    }
36
37    bool MtfsFuse::runPrepare(int argc, char **argv) {
38      (void) argc, argv;
39      return true;
40    }
41
42    void MtfsFuse::init(void *userdata, fuse_conn_info *conn) {
43      mtfs::Mtfs::getInstance()->init(userdata, conn);
44    }
45
46    void MtfsFuse::destroy(void *userdata) {
47      mtfs::Mtfs::getInstance()->destroy(userdata);
48    }
49
50    void MtfsFuse::lookup(fuse_req_t req, fuse_ino_t parent, const char *name) {
51      mtfs::Mtfs::getInstance()->lookup(req, parent, name);
52    }
53
54    void MtfsFuse::getAttr(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) {
55      mtfs::Mtfs::getInstance()->getAttr(req, ino, fi);
56    }
57
58    void MtfsFuse::setattr(fuse_req_t req, fuse_ino_t ino, struct stat *attr, int
       to_set, fuse_file_info *fi) {
59      mtfs::Mtfs::getInstance()->setAttr(req, ino, attr, to_set, fi);
60    }
61
62    void MtfsFuse::mknod(fuse_req_t req, fuse_ino_t parent, const char *name,
       mode_t mode, dev_t rdev) {
63      mtfs::Mtfs::getInstance()->mknod(req, parent, name, mode, rdev);
64    }
65
66    void MtfsFuse::mkdir(fuse_req_t req, fuse_ino_t parent, const char *name,
       mode_t mode) {
67      mtfs::Mtfs::getInstance()->mkdir(req, parent, name, mode);
68    }
69
70    void MtfsFuse::unlink(fuse_req_t req, fuse_ino_t parent, const char *name) {
71      mtfs::Mtfs::getInstance()->unlink(req, parent, name);
72    }
73
```

```
74  void MtfsFuse::rmdir(fuse_req_t req, fuse_ino_t parent, const char *name) {
75    mtfs::Mtfs::getInstance()->rmdir(req, parent, name);
76  }
77
78  void MtfsFuse::open(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) {
79    mtfs::Mtfs::getInstance()->open(req, ino, fi);
80  }
81
82  void MtfsFuse::release(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) {
83    mtfs::Mtfs::getInstance()->release(req, ino, fi);
84  }
85
86  void MtfsFuse::opendir(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) {
87    mtfs::Mtfs::getInstance()->opendir(req, ino, fi);
88  }
89
90  void MtfsFuse::access(fuse_req_t req, fuse_ino_t ino, int mask) {
91    mtfs::Mtfs::getInstance()->access(req, ino, mask);
92  }
93
94  void MtfsFuse::readdir(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off
      , fuse_file_info *fi) {
95    mtfs::Mtfs::getInstance()->readdir(req, ino, size, off, fi);
96  }
97
98  void MtfsFuse::readdirplus(fuse_req_t req, fuse_ino_t ino, size_t size, off_t
       off, fuse_file_info *fi) {
99    mtfs::Mtfs::getInstance()->readdirplus(req, ino, size, off, fi);
100 }
101
102 void MtfsFuse::releasedir(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi)
       {
103   mtfs::Mtfs::getInstance()->releasedir(req, ino, fi);
104 }
105
106 void MtfsFuse::write(fuse_req_t req, fuse_ino_t ino, const char *buf, size_t
      size, off_t off, fuse_file_info *fi) {
107   mtfs::Mtfs::getInstance()->write(req, ino, buf, size, off, fi);
108 }
109
110 void MtfsFuse::write_buf(fuse_req_t req, fuse_ino_t ino, fuse_bufvec *bufv,
      off_t off, fuse_file_info *fi) {
111   mtfs::Mtfs::getInstance()->write_buf(req, ino, bufv, off, fi);
112 }
113
114 void MtfsFuse::read(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
      fuse_file_info *fi) {
115   mtfs::Mtfs::getInstance()->read(req, ino, size, off, fi);
116 }
117
118 }  // namespace wrapper
```

Listing 3.15 – "wrapper/MtfsFuse.cpp"

3.4 *Core*

3.4.1 structs.h

```
/**
 * \file structs.h
 * \brief
 * \author David Wittwer
 * \version 0.0.1
 * \copyright GNU Publis License V3
 *
 * This file is part of MTFS.

    MTFS is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Foobar is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
 */

#ifndef TRAVAIL_BACHELOR_STRUCTS_H
#define TRAVAIL_BACHELOR_STRUCTS_H

#include <map>
#include <rapidjson/document.h>

#define IN_MODE "mode"
#define IN_UID "uid"
#define IN_GID "gid"
#define IN_SIZE "size"
#define IN_LINKS "linkCount"
#define IN_ACCESS "atime"
#define IN_BLOCKS "dataBlocks"

#define ID_POOL "poolId"
#define ID_VOLUME "volumeId"
#define ID_ID "id"

#define RU_MIGRATION "migration"

#define PO_POOLS "pools"

#define SB_INODE_CACHE "inodeCacheSize"
#define SB_DIR_CACHE "directoryCacheSize"
#define SB_BLOCK_CACHE "blockCacheSize"
#define SB_BLOCK_SIZE_ST "blockSize"
#define SB_REDUNDANCY "redundancy"

#define BI_REFF "referenceId"
#define BI_ACCESS "lastAccess"

namespace mtfs {
  class Rule;

  enum blockType {
    INODE,
```

```
60      DIR_BLOCK,
61      DATA_BLOCK,
62      SUPERBLOCK
63    };
64
65    typedef struct ruleInfo_st {
66      uid_t uid;
67      gid_t gid;
68      uint64_t lastAccess;
69
70      ruleInfo_st() : ruleInfo_st(0, 0, 0) {}
71
72      ruleInfo_st(uid_t uid, gid_t gid, uint64_t atime) : uid(uid), gid(gid),
      lastAccess(atime) {}
73    } ruleInfo_t;
74
75    typedef struct ident_st {
76      std::uint32_t poolId;
77      std::uint32_t volumeId;
78      std::uint64_t id;
79
80      explicit ident_st(std::uint64_t id = 0, std::uint32_t vid = 0, std::
      uint32_t pid = 0) : id(id),
81                                                volumeId(vid),
82                                                poolId(pid) {}
83
84      ~ident_st() = default;
85
86      ident_st &operator=(const ident_st &id) {
87        this->poolId = id.poolId;
88        this->volumeId = id.volumeId;
89        this->id = id.id;
90
91        return *this;
92      }
93
94      bool operator==(const ident_st &i) const {
95        return (poolId == i.poolId && volumeId == i.volumeId && id == i.id);
96      }
97
98      bool operator!=(const ident_st &i) const {
99        return (poolId != i.poolId || volumeId != i.volumeId || id != i.id);
100     }
101
102     bool operator<(const ident_st &i) const {
103       if (poolId < i.poolId)
104         return true;
105
106       if (volumeId < i.volumeId)
107         return true;
108
109       return id < i.id;
110     }
111
112     void toJson(rapidjson::Value &dest, rapidjson::Document::AllocatorType &
      alloc) const {
113       rapidjson::Value v;
114
115       v.SetUint(this->poolId);
```

```
116       dest.AddMember(rapidjson::StringRef(ID_POOL), v, alloc);
117
118       v.SetUint(this->volumeId);
119       dest.AddMember(rapidjson::StringRef(ID_VOLUME), v, alloc);
120
121       v.SetUint64(this->id);
122       dest.AddMember(rapidjson::StringRef(ID_ID), v, alloc);
123     }
124
125     void fromJson(rapidjson::Value &src) {
126       assert(src.IsObject());
127       assert(src.HasMember(ID_POOL));
128       assert(src.HasMember(ID_VOLUME));
129       assert(src.HasMember(ID_ID));
130
131       this->poolId = src[ID_POOL].GetUint();
132       this->volumeId = src[ID_VOLUME].GetUint();
133       this->id = src[ID_ID].GetUint64();
134     }
135
136     std::string toString() {
137       return "p:" + std::to_string(this->poolId) + " v:" + std::to_string(this
    ->volumeId) + " i:" +
138              std::to_string(this->id);
139     }
140
141   } ident_t;
142
143   typedef struct inode_st {
144     mode_t accesRight{0};
145     uid_t uid{0};
146     gid_t gid{0};
147     uint64_t size{0};
148     uint32_t linkCount{1};
149     std::uint64_t atime;
150     std::vector<std::vector<ident_t>> dataBlocks;
151
152     inode_st() : atime((uint64_t) time(nullptr)) {
153       this->dataBlocks.clear();
154     }
155
156     bool operator==(const inode_st &rhs) const {
157       if (accesRight != rhs.accesRight)
158         return false;
159       if (uid != rhs.uid)
160         return false;
161       if (gid != rhs.gid)
162         return false;
163       if (size != rhs.size)
164         return false;
165       if (linkCount != rhs.linkCount)
166         return false;
167       if (atime != rhs.atime)
168         return false;
169
170       if (dataBlocks.size() != rhs.dataBlocks.size()) {
171 //        std::cout << "size differ " << dataBlocks.size() << " " << rhs.
    dataBlocks.size() << std::endl;
172         return false;
```

```
173       }
174       for (auto lIter = dataBlocks.begin(), rIter = rhs.dataBlocks.begin();
175          lIter != dataBlocks.end(); lIter++, rIter++) {
176         std::vector<ident_t> red = *lIter, rhsRed = *rIter;
177
178         if (red.size() != rhsRed.size())
179           return false;
180         for (auto lb = red.begin(), rb = rhsRed.begin();
181            lb != red.end(); lb++, rb++) {
182           ident_t li = *lb, ri = *rb;
183
184           if (li != ri)
185             return false;
186         }
187       }
188
189       return true;
190     }
191
192     void toJson(rapidjson::Document &dest) const {
193       dest.SetObject();
194
195       rapidjson::Value v;
196       rapidjson::Document::AllocatorType &alloc = dest.GetAllocator();
197
198       v.SetInt(this->accesRight);
199       dest.AddMember(rapidjson::StringRef(IN_MODE), v, alloc);
200
201       v.SetInt(this->uid);
202       dest.AddMember(rapidjson::StringRef(IN_UID), v, alloc);
203
204       v.SetInt(this->gid);
205       dest.AddMember(rapidjson::StringRef(IN_GID), v, alloc);
206
207       v.SetUint64(this->size);
208       dest.AddMember(rapidjson::StringRef(IN_SIZE), v, alloc);
209
210       v.SetUint(this->linkCount);
211       dest.AddMember(rapidjson::StringRef(IN_LINKS), v, alloc);
212
213       v.SetUint64(this->atime);
214       dest.AddMember(rapidjson::StringRef(IN_ACCESS), v, alloc);
215
216       rapidjson::Value a(rapidjson::kArrayType);
217       for (auto &&blocks : dataBlocks) {
218         rapidjson::Value red(rapidjson::kArrayType);
219
220         for (auto &&block : blocks) {
221           rapidjson::Value ident(rapidjson::kObjectType);
222
223           block.toJson(ident, alloc);
224
225           red.PushBack(ident, alloc);
226         }
227
228         a.PushBack(red, alloc);
229       }
230       dest.AddMember(rapidjson::StringRef(IN_BLOCKS), a, alloc);
231     }
```

```
232
233     void fromJson(rapidjson::Document &d) {
234       assert(d.IsObject());
235       assert(d.HasMember(IN_MODE));
236       assert(d.HasMember(IN_UID));
237       assert(d.HasMember(IN_GID));
238       assert(d.HasMember(IN_SIZE));
239       assert(d.HasMember(IN_LINKS));
240       assert(d.HasMember(IN_ACCESS));
241       assert(d.HasMember(IN_BLOCKS));
242
243       this->accesRight = (uint16_t) d[IN_MODE].GetUint();
244       this->uid = (uint16_t) d[IN_UID].GetUint();
245       this->gid = (uint16_t) d[IN_GID].GetUint();
246       this->size = d[IN_SIZE].GetUint64();
247       this->linkCount = (uint8_t) d[IN_LINKS].GetUint();
248       this->atime = d[IN_ACCESS].GetUint64();
249
250       const rapidjson::Value &dataArray = d[IN_BLOCKS];
251       assert(dataArray.IsArray());
252       this->dataBlocks.clear();
253       for (auto &a : dataArray.GetArray()) {
254         std::vector<ident_t> redundancy;
255
256         assert(a.IsArray());
257         for (auto &v : a.GetArray()) {
258           ident_t ident;
259
260           assert(v.IsObject());
261           assert(v.HasMember(ID_POOL));
262           assert(v.HasMember(ID_VOLUME));
263           assert(v.HasMember(ID_ID));
264
265           ident.poolId = (uint16_t) v[ID_POOL].GetUint();
266           ident.volumeId = (uint16_t) v[ID_VOLUME].GetUint();
267           ident.id = v[ID_ID].GetUint64();
268
269           redundancy.push_back(ident);
270         }
271
272         this->dataBlocks.push_back(redundancy);
273       }
274     }
275   } inode_t;
276
277   typedef struct dirBlock_st {
278     std::map<std::string, std::vector<mtfs::ident_t>> entries;
279
280     void fromJson(rapidjson::Document &d) {
281       assert(d.IsObject());
282
283       for (auto &&item: d.GetObject()) {
284         assert(item.value.IsArray());
285
286         std::vector<ident_t> ids;
287
288         for (auto &&ident: item.value.GetArray()) {
289           ident_t i;
290           i.fromJson(ident);
```

```
291             ids.push_back(i);
292           }
293           this->entries.emplace(item.name.GetString(), ids);
294         }
295       }
296
297       void toJson(rapidjson::Document &d) {
298         d.SetObject();
299
300         rapidjson::Document::AllocatorType &allocator = d.GetAllocator();
301
302         for (auto &&item: this->entries) {
303           rapidjson::Value r(rapidjson::kArrayType);
304           for (auto &&ident: item.second) {
305             rapidjson::Value v(rapidjson::kObjectType);
306             ident.toJson(v, allocator);
307             r.PushBack(v, allocator);
308           }
309           d.AddMember(rapidjson::StringRef(item.first.c_str()), r, allocator);
310         }
311       }
312
313     } dirBlock_t;
314
315     typedef struct volume_st {
316       std::string pluginName;
317       Rule *rule;
318       std::map<std::string, std::string> params;
319     } volume_t;
320
321     typedef struct pool_st {
322       int migration;
323       Rule *rule;
324       std::map<uint32_t, volume_t> volumes;
325     } pool_t;
326
327     typedef struct superblock_st {
328       size_t iCacheSz;
329       size_t dCacheSz;
330       size_t bCacheSz;
331       size_t blockSz;
332       size_t redundancy;
333       int migration;
334       std::map<uint32_t, pool_t> pools;
335       std::vector<ident_t> rootInodes;
336     } superblock_t;
337
338     typedef struct blockInfo_st {
339       ident_t id;
340       std::vector<ident_t> referenceId;
341       uint64_t lastAccess{0};
342
343       bool operator==(const blockInfo_st &other) const {
344         if (other.id != this->id)
345           return false;
346
347         if (!(other.referenceId == this->referenceId))
348           return false;
349
```

```
350
351       return (other.lastAccess == this->lastAccess);
352     }
353
354     blockInfo_st() : id(ident_t()) {};
355
356     void toJson(rapidjson::Document &d) {
357       d.SetObject();
358       rapidjson::Document::AllocatorType &allocator = d.GetAllocator();
359
360       rapidjson::Value v;
361
362       rapidjson::Value a(rapidjson::kArrayType);
363       for (auto &&id : this->referenceId) {
364         v.SetObject();
365
366         v.AddMember(rapidjson::StringRef(ID_POOL), rapidjson::Value(id.poolId),
      allocator);
367         v.AddMember(rapidjson::StringRef(ID_VOLUME), rapidjson::Value(id.
     volumeId), allocator);
368         v.AddMember(rapidjson::StringRef(ID_ID), rapidjson::Value(id.id),
     allocator);
369
370         a.PushBack(v, allocator);
371       }
372       d.AddMember(rapidjson::StringRef(BI_REFF), a, allocator);
373
374       v.SetUint64(this->lastAccess);
375       d.AddMember(rapidjson::StringRef(BI_ACCESS), v, allocator);
376     }
377
378     void fromJson(rapidjson::Document &d) {
379       assert(d.IsObject());
380       assert(d.HasMember(BI_REFF));
381       assert(d[BI_REFF].IsArray());
382       assert(d.HasMember(BI_ACCESS));
383
384       for (auto &&id :d[BI_REFF].GetArray()) {
385         assert(id.IsObject());
386         assert(id.HasMember(ID_POOL));
387         assert(id.HasMember(ID_VOLUME));
388         assert(id.HasMember(ID_ID));
389
390         uint32_t poolId = id[ID_POOL].GetUint();
391         uint32_t volumeId = id[ID_VOLUME].GetUint();
392         uint64_t i = id[ID_ID].GetUint64();
393
394         this->referenceId.push_back(ident_st(i, volumeId, poolId));
395       }
396
397       this->lastAccess = d[BI_ACCESS].GetUint64();
398     }
399
400   } blockInfo_t;
401
402
403 }  // namespace mtFS
404
405 #endif //TRAVAIL_BACHELOR_STRUCTS_H
```

Listing 3.16 – "core/structs.h"

3.4.2 Acces.h

```
/**
 * \file Access.h
 * \brief
 * \author David Wittwer
 * \version 0.0.1
 * \copyright GNU Publis License V3
 *
 * This file is part of MTFS.

    MTFS is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Foobar is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
 */

#ifndef FILESTORAGE_INODE_ACCES_H
#define FILESTORAGE_INODE_ACCES_H

#include <string>
#include <vector>
#include <list>
#include <iostream>
#include <assert.h>
#include <mtfs/structs.h>

namespace mtfs {
  typedef struct ruleInfo_st ruleInfo_t;

  class Acces {
  public:

    virtual ~Acces() {};

    /**
     * Add a block in a volume
     *
     * @param [in] info ruleInfo_t To choose a pool or volume in which add the
    block
     * @param [out] ids Id for new blocks
     * @param [in] type Type of block (see enum blockType)
     * @param [in] nb Numbre of block to add.
     *
     * @return 0 if success else std linus error code.
     */
```

```
52      virtual int add(const ruleInfo_t &info, std::vector<ident_t> &ids,
        blockType type, size_t nb)=0;
53
54      /**
55       * Dele a block in a volume
56       *
57       * @param [in] id Id of block to delete
58       * @param [in] type Type of block to delete
59       *
60       * @return 0 if success else std linus error code.
61       */
62      virtual int del(const ident_t &id, blockType type)=0;
63
64      /**
65       * Get a block in a volume
66       *
67       * @param [in] id Id of block
68       * @param [out] data Pointer on memory already allocate
69       * @param [in] type Type of block
70       *
71       * @return 0 if success else std linus error code.
72       */
73      virtual int get(const ident_t &id, void *data, blockType type)=0;
74
75      /**
76       * Put a block in a volume
77       *
78       * @param [in] id Id of block
79       * @param [in] data Pointer on memory who contains the data.
80       * @param [in] type Type of block
81       *
82       * @return 0 if success else std linus error code.
83       */
84      virtual int put(const ident_t &id, const void *data, blockType type)=0;
85
86      /**
87       * Get metas block info
88       *
89       * @param [in] id Id of block
90       * @param [out] metas Datas
91       * @param [in] type Type of block
92       *
93       * @return 0 if success else std linus error code.
94       */
95      virtual int getMetas(const ident_t &id, blockInfo_t &metas, blockType type)
        =0;
96
97      /**
98       * Put metas block info
99       *
100      * @param id Id of block
101      * @param metas Datas
102      * @param type Type of block
103      *
104      * @return 0 if success else std linus error code.
105      */
106     virtual int putMetas(const ident_t &id, const blockInfo_t &metas, blockType
        type)=0;
107 };
```

```
108
109 }  // namespace mtfs
110 #endif
```

Listing 3.17 – "core/Acces.h"

3.4.3 Mtfs

Mtfs.h

```
1  /**
2   * \file Mtfs.h
3   * \brief
4   * \author David Wittwer
5   * \version 0.0.1
6   * \copyright GNU Publis License V3
7   *
8   * This file is part of MTFS.
9
10     MTFS is free software: you can redistribute it and/or modify
11     it under the terms of the GNU General Public License as published by
12     the Free Software Foundation, either version 3 of the License, or
13     (at your option) any later version.
14
15     Foobar is distributed in the hope that it will be useful,
16     but WITHOUT ANY WARRANTY; without even the implied warranty of
17     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18     GNU General Public License for more details.
19
20     You should have received a copy of the GNU General Public License
21     along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22  */
23
24 #ifndef FILESTORAGE_MTFS_H
25 #define FILESTORAGE_MTFS_H
26
27 #include <thread>
28
29 #include <mtfs/Rule.h>
30 #include <mtfs/Acces.h>
31 #include <rapidjson/document.h>
32 #include <boost/threadpool.hpp>
33 #include <fuse3/fuse_lowlevel.h>
34 #include "structs.h"
35 #include "Migrator.h"
36 #include <mutex>
37 #include <condition_variable>
38 #include <utils/Semaphore.h>
39
40 namespace mtfs {
41   struct internalInode_st;
42   struct dl_st;
43
44   class Mtfs {
45   public:
46 //     static constexpr const char *SYSTEMS_DIR = "Systems";
47     static constexpr const char *CONFIG_DIR = "Configs";
48
```

```
49      static constexpr const char *INODE_CACHE = "inodeCacheSize";
50      static constexpr const char *DIR_CACHE = "directoryCacheSize";
51      static constexpr const char *BLOCK_CACHE = "blockCacheSize";
52      static constexpr const char *BLOCK_SIZE_ST = "blockSize";
53      static constexpr const char *REDUNDANCY = "redundancy";
54      static constexpr const char *ROOT_INODES = "rootInodes";
55
56
57    private:
58  //    CONFIG
59      static const size_t SIMULT_DL = 2;
60      static const size_t SIMULT_UP = 2;
61      static const int INIT_DL = 2;
62      static constexpr const double ATTR_TIMEOUT = 1.0;
63
64  //    REQUEST STATUS CODES
65      static const int SUCCESS = 0;
66      static const int PENDING = 9999;
67
68      static Mtfs *instance;
69      static boost::threadpool::pool *threadPool;
70      static std::string systemName;
71
72      size_t redundancy;
73      size_t blockSize;
74      int maxEntryPerBlock;
75      internalInode_st *rootIn;
76
77      Acces *inodes;
78      Acces *dirBlocks;
79      Acces *blocks;
80
81      std::thread migratorThr;
82      Migrator::info_st migratorInfo;
83
84    public:
85      static Mtfs *getInstance();
86
87      /**
88       * Validate system config
89       *
90       * @param system JSON config of system
91       *
92       * @return true if validelse false
93       */
94      static bool validate(const rapidjson::Value &system);
95
96      /**
97       * Create the root inode
98       *
99       * @param [out] inode The root inode
100      *
101      * @return true if success
102      */
103     static bool createRootInode(inode_t &inode);
104
105     /**
106      * Start MTFS
107      *
```

```
108      * @param system System config
109      * @param homeDir MTFS home dir
110      * @param sysName System name
111      *
112      * @return true if success
113      */
114     static bool start(rapidjson::Document &system, std::string homeDir, std::
        string sysName);
115
116     /**
117      * Stop MTFS
118      */
119     static void stop();
120
121     /**
122      * Convert struct superblock to JSON document
123      *
124      * @param [in] sb Superblock struct
125      * @param [out] d JSON document
126      */
127     static void structToJson(const superblock_t &sb, rapidjson::Document &d);
128
129     /**
130      * Convert JSON document to struct superblock
131      *
132      * @param [in] d JSON document
133      * @param [out] sb Struct superblock
134      */
135     static void jsonToStruct(rapidjson::Document &d, superblock_t &sb);
136
137     /*            Fuse handlers           */
138
139     /**
140      * Init handler
141      *
142      * @param userdata
143      * @param conn
144      */
145     void init(void *userdata, fuse_conn_info *conn);
146
147     /**
148      * Destroy handler
149      *
150      * @param userdata
151      */
152     void destroy(void *userdata);
153
154     /**
155      * Lookup handler
156      *
157      * @param req
158      * @param parent
159      * @param name
160      */
161     void lookup(fuse_req_t req, fuse_ino_t parent, const std::string &name);
162
163     /**
164      * Getattr handler
165      *
```

```
166         * @param req
167         * @param ino
168         * @param fi
169         */
170        void getAttr(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi);
171
172        /**
173         * Setattr handler
174         *
175         * @param req
176         * @param ino
177         * @param attr
178         * @param toSet
179         * @param fi
180         */
181        void setAttr(fuse_req_t req, fuse_ino_t ino, struct stat *attr, int toSet,
           fuse_file_info *fi);
182
183        /**
184         * Mknod handler
185         *
186         * @param req
187         * @param parent
188         * @param name
189         * @param mode
190         * @param rdev
191         */
192        void mknod(fuse_req_t req, fuse_ino_t parent, const std::string &name,
           mode_t mode, dev_t rdev);
193
194        /**
195         * Mkdir handler
196         *
197         * @param req
198         * @param ino
199         * @param name
200         * @param mode
201         */
202        void mkdir(fuse_req_t req, fuse_ino_t ino, const std::string &name, mode_t
           mode);
203
204        /**
205         * Unlink handler
206         *
207         * @param req
208         * @param parent
209         * @param name
210         */
211        void unlink(fuse_req_t req, fuse_ino_t parent, const std::string &name);
212
213        /**
214         * Rmdir handler
215         *
216         * @param req
217         * @param parent
218         * @param name
219         */
220        void rmdir(fuse_req_t req, fuse_ino_t parent, const std::string &name);
221
```

```
222     /**
223      * Open handler
224      *
225      * @param req
226      * @param ino
227      * @param fi
228      */
229     void open(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi);
230 
231     /**
232      * Read handler
233      *
234      * @param req
235      * @param ino
236      * @param size
237      * @param off
238      * @param fi
239      */
240     void read(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
    fuse_file_info *fi);
241 
242     /**
243      * Write handler
244      *
245      * @param req
246      * @param ino
247      * @param buf
248      * @param size
249      * @param off
250      * @param fi
251      */
252     void write(fuse_req_t req, fuse_ino_t ino, const char *buf, size_t size,
    off_t off, fuse_file_info *fi);
253 
254     /**
255      * Release handler
256      *
257      * @param req
258      * @param ino
259      * @param fi
260      */
261     void release(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi);
262 
263     /**
264      * Opendir handler
265      *
266      * @param req
267      * @param ino
268      * @param fi
269      */
270     void opendir(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi);
271 
272     /**
273      * Readdir handler
274      *
275      * @param req
276      * @param ino
277      * @param size
278      * @param off
```

```
279         * @param fi
280         */
281        void readdir(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
       fuse_file_info *fi);
282
283        /**
284         * Releasedir handler
285         *
286         * @param req
287         * @param ino
288         * @param fi
289         */
290        void releasedir(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi);
291
292        /**
293         * Access handler
294         *
295         * @param req
296         * @param ino
297         * @param mask
298         */
299        void access(fuse_req_t req, fuse_ino_t ino, int mask);
300
301        /**
302         * Write_buf handler
303         *
304         * @param req
305         * @param ino
306         * @param bufv
307         * @param off
308         * @param fi
309         */
310        void write_buf(fuse_req_t req, fuse_ino_t ino, fuse_bufvec *bufv, off_t off
       , fuse_file_info *fi);
311
312        /**
313         * Readdirplus handler
314         *
315         * @param req
316         * @param ino
317         * @param size
318         * @param off
319         * @param fi
320         */
321        void readdirplus(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
       fuse_file_info *fi);
322
323
324      private:
325
326        Mtfs();
327
328
329        inode_t getRootInode();
330
331        bool build(const superblock_t &superblock);
332
333        void readRootInode();
334
```

```
335     void writeRootInode();
336
337     int addEntry(internalInode_st *parentInode, std::string name, std::vector<
    ident_t> &inodeIds);
338
339     int insertInode(const inode_t &inode, std::vector<ident_t> &idents);
340
341     void dlBlocks(const inode_t &inode, dl_st *dlSt, const blockType type,
    const int firstBlockIdx);
342
343     void dlDirBlocks(std::vector<ident_t> &ids, std::queue<dirBlock_t> *q, std
    ::mutex *queueMutex, Semaphore *sem);
344
345     void dlInodes(dl_st *src, dl_st *dst);
346
347     void initMetas(const internalInode_st &parentInode, const std::vector<
    ident_t> &ids, const blockType &type,
348             boost::threadpool::pool *thPool = nullptr);
349
350     void
351     dlInode(std::vector<ident_t> &ids, std::queue<std::pair<std::string,
    inode_t>> *queue, std::mutex *queueMutex,
352         Semaphore *sem,
353         std::string &key);
354
355
356     ////              UTILS             ////
357
358     internalInode_st *getIntInode(fuse_ino_t ino);
359
360     void
361     doReaddir(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
    fuse_file_info *fi, const bool &plus = false);
362
363     size_t dirBufAdd(fuse_req_t &req, char *buf, size_t &currentSize, std::
    string name, internalInode_st &inode,
364             const bool &plus);
365
366     void buildParam(const internalInode_st &inode, fuse_entry_param &param);
367
368     void buildStat(const internalInode_st &inode, struct stat &st);
369
370     ruleInfo_t getRuleInfo(const inode_t &inode);
371
372     static internalInode_st *newInode(const mode_t &mode, const fuse_ctx *ctx);
373
374     static uint64_t now();
375
376     int doUnlink(internalInode_st *parent, const std::string name);
377
378     int delEntry(std::vector<ident_t> &ids, dirBlock_t &blk, const std::string
    &name);
379   };
380
381 }  // namespace mtfs
382 #endif
```

Listing 3.18 – "core/Mtfs.h"

Mtfs.cpp

```
/**
 * \file Mtfs.cpp
 * \brief
 * \author David Wittwer
 * \version 0.0.1
 * \copyright GNU Publis License V3
 *
 * This file is part of MTFS.

    MTFS is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Foobar is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <mtfs/Mtfs.h>
#include <mtfs/Cache.h>
#include <pluginSystem/PluginManager.h>
#include <fstream>
#include <rapidjson/istreamwrapper.h>
#include <rapidjson/stringbuffer.h>
#include <rapidjson/prettywriter.h>
#include <boost/filesystem.hpp>
#include <utility>
#include <grp.h>
#include <pwd.h>
#include <utils/Logger.h>

#define GROUPS_TO_SEARCH 30
//TODO compute this value in function of entry size
#define ENTRY_PER_BLOCK 20

namespace mtfs {
  using namespace std;
  using namespace rapidjson;
  using namespace boost::threadpool;

  struct internalInode_st {
    inode_t inode;
    vector<ident_t> idents;
  };

  struct dl_st {
    mutex *fifoMu{nullptr};
    Semaphore *sem{nullptr};
    union {
      queue<pair<string, inode_t>> *inodeQueue;
      queue<dirBlock_t> *dirQueue;
      queue<uint8_t *> *blkQueue;
```

```
58      } fifo;
59      bool end{false};
60      mutex *endMu{nullptr};
61      pool *dlThPool{nullptr};
62
63      dl_st() {
64      }
65    };
66
67    struct fd_st {
68      enum bt {
69        DATA,
70        DIR,
71      };
72
73      bool firstCall{true};
74      bt blockT;
75      dl_st blkDl;
76      dl_st inoDl;
77      boost::thread *blkThr{nullptr};
78      boost::thread *inoThr{nullptr};
79
80      fd_st() : blkDl(dl_st()), inoDl(dl_st()) {}
81
82      ~fd_st() {
83        switch (this->blockT) {
84          case DATA:
85            delete blkDl.fifo.blkQueue;
86            break;
87          case DIR:
88            delete blkDl.fifo.dirQueue;
89            break;
90        }
91      }
92    };
93
94
95    Mtfs *Mtfs::instance = nullptr;
96    pool *Mtfs::threadPool = nullptr;
97    string Mtfs::systemName;
98
99    //
      ////////////////////////////////////////////////////////////////////////////////
100   ////                          STATICS                                       ////
101   //
      ////////////////////////////////////////////////////////////////////////////////
102
103   Mtfs *Mtfs::getInstance() {
104     if (instance == nullptr)
105       instance = new Mtfs();
106
107     return instance;
108   }
109
110   bool Mtfs::validate(const Value &system) {
111     if (!system.IsObject())
112       throw invalid_argument("Not a object!");
```

```
113
114     if (!system.HasMember(INODE_CACHE))
115       throw invalid_argument("Inode cache missing!");
116
117     if (!system.HasMember(DIR_CACHE))
118       throw invalid_argument("Directory cache missing!");
119
120     if (!system.HasMember(BLOCK_CACHE))
121       throw invalid_argument("Block cache missing!");
122
123     if (!system.HasMember(BLOCK_SIZE_ST))
124       throw invalid_argument("Block size missing!");
125
126     if (!system.HasMember(REDUNDANCY))
127       throw invalid_argument("Redundancy missing!");
128
129     if (!system.HasMember(Pool::POOLS))
130       throw invalid_argument("Pools missing!");
131     if (!system[Pool::POOLS].IsObject())
132       throw invalid_argument("Pool is not a object!");
133
134     int migration = -1;
135     if (system[Pool::POOLS].MemberCount() <= 0)
136       throw invalid_argument("Number of pool invalid!");
137     else if (system[Pool::POOLS].MemberCount() != 1) {
138       if (!system.HasMember(Rule::MIGRATION))
139         throw invalid_argument("Migration missing!");
140
141       migration = system[Rule::MIGRATION].GetInt();
142     }
143
144     for (auto &m: system[Pool::POOLS].GetObject()) {
145       if (Rule::rulesAreValid(migration, m.value) != Rule::VALID_RULES)
146         throw invalid_argument(string("Rules invalid for pool '") + m.name.
    GetString() + "'");
147
148       if (!Pool::validate(m.value))
149         throw invalid_argument(string("Pool '") + m.name.GetString() + "'
    invalid!");
150     }
151
152     return true;
153   }
154
155   bool Mtfs::createRootInode(inode_t &inode) {
156     inode.accesRight = S_IFDIR | 0775;
157     inode.uid = 0;
158     inode.gid = 1001;
159     inode.size = 0;
160     inode.linkCount = 2;
161     inode.atime = (uint64_t) time(nullptr);
162     inode.dataBlocks.clear();
163
164     return true;
165   }
166
167   bool Mtfs::start(Document &system, std::string homeDir, string sysName) {
168     (void) homeDir;
169     systemName = std::move(sysName);
```

```
170
171     superblock_t superblock;
172     jsonToStruct(system, superblock);
173
174     if (!getInstance()->build(superblock))
175       return false;
176
177     auto nbThread = (unsigned int) (thread::hardware_concurrency() * 1.25);
178     threadPool = new pool(nbThread);
179
180     return true;
181   }
182
183   void Mtfs::stop() {
184     delete threadPool;
185
186     delete instance;
187   }
188
189   void Mtfs::structToJson(const superblock_t &sb, rapidjson::Document &d) {
190     rapidjson::Document::AllocatorType &allocator = d.GetAllocator();
191
192     d.SetObject();
193
194     rapidjson::Value v;
195
196     v.SetUint((unsigned int) sb.iCacheSz);
197     d.AddMember(rapidjson::StringRef(Mtfs::INODE_CACHE), v, allocator);
198
199     v.SetUint((unsigned int) sb.dCacheSz);
200     d.AddMember(rapidjson::StringRef(Mtfs::DIR_CACHE), v, allocator);
201
202     v.SetUint((unsigned int) sb.bCacheSz);
203     d.AddMember(rapidjson::StringRef(Mtfs::BLOCK_CACHE), v, allocator);
204
205     v.SetUint((unsigned int) sb.blockSz);
206     d.AddMember(rapidjson::StringRef(Mtfs::BLOCK_SIZE_ST), v, allocator);
207
208     v.SetUint((unsigned int) sb.redundancy);
209     d.AddMember(rapidjson::StringRef(Mtfs::REDUNDANCY), v, allocator);
210
211     v.SetInt(sb.migration);
212     d.AddMember(rapidjson::StringRef(Rule::MIGRATION), v, allocator);
213
214     rapidjson::Value pools(rapidjson::kObjectType);
215     for (auto &&item: sb.pools) {
216       rapidjson::Value pool(rapidjson::kObjectType);
217
218       Pool::structToJson(item.second, pool, allocator);
219
220       string id = to_string(item.first);
221       Value index(id.c_str(), (SizeType) id.size(), allocator);
222       pools.AddMember(index, pool, allocator);
223     }
224     d.AddMember(rapidjson::StringRef(Pool::POOLS), pools, allocator);
225
226     v.SetArray();
227     for (auto &&id: sb.rootInodes) {
228       Value ident(kObjectType);
```

```
229       id.toJson(ident, allocator);
230       v.PushBack(ident, allocator);
231     }
232     d.AddMember(StringRef(ROOT_INODES), v, allocator);
233   }
234
235   void Mtfs::jsonToStruct(rapidjson::Document &d, superblock_t &sb) {
236     assert(d.HasMember(INODE_CACHE));
237     sb.iCacheSz = d[INODE_CACHE].GetUint();
238
239     assert(d.HasMember(DIR_CACHE));
240     sb.dCacheSz = d[DIR_CACHE].GetUint();
241
242     assert(d.HasMember(BLOCK_CACHE));
243     sb.bCacheSz = d[BLOCK_CACHE].GetUint();
244
245     assert(d.HasMember(BLOCK_SIZE_ST));
246     sb.blockSz = d[BLOCK_SIZE_ST].GetUint();
247
248     assert(d.HasMember(REDUNDANCY));
249     sb.redundancy = d[REDUNDANCY].GetUint();
250
251     assert(d.HasMember(Rule::MIGRATION));
252     sb.migration = d[Rule::MIGRATION].GetInt();
253
254     assert(d.HasMember(Pool::POOLS));
255     for (auto &&item :d[Pool::POOLS].GetObject()) {
256       string sId = item.name.GetString();
257       auto id = (uint32_t) stoul(sId);
258       pool_t pool;
259       memset(&pool, 0, sizeof(pool_t));
260       pool.volumes.clear();
261
262       pool.rule = Rule::buildRule(sb.migration, item.value);
263
264       Pool::jsonToStruct(item.value, pool);
265
266       sb.pools.insert(make_pair(id, pool));
267     }
268
269     assert(d.HasMember(ROOT_INODES) && d[ROOT_INODES].IsArray());
270     for (auto &&ident: d[ROOT_INODES].GetArray()) {
271       ident_t id = ident_t();
272       id.fromJson(ident);
273       sb.rootInodes.push_back(id);
274     }
275   }
276
277   //
     ////////////////////////////////////////////////////////////////////////////////
278   ////                          MEMBERS                                       ////
279   //
     ////////////////////////////////////////////////////////////////////////////////
280
281   //
     ////////////////////////////////////////////////////////////////////////////////
```

```
282   ////                          FUSE fcts                                ////
283   //
       ////////////////////////////////////////////////////////////////////////////////////////
284
285   void Mtfs::init(void *userdata, fuse_conn_info *conn) {
286     (void) userdata, conn;
287
288     getInstance()->readRootInode();
289
290     this->migratorInfo.poolManager = (PoolManager *) this->blocks;
291
292     this->migratorThr = std::thread(Migrator::main, &this->migratorInfo);
293
294 #ifdef DEBUG
295     cerr << "[MTFS]: End init" << endl;
296 #endif
297   }
298
299   void Mtfs::destroy(void *userdata) {
300     (void) userdata;
301
302     unique_lock<mutex> lk(*this->migratorInfo.endMutex);
303     this->migratorInfo.end = true;
304     lk.unlock();
305     this->migratorInfo.condV.notify_all();
306
307     this->migratorThr.join();
308
309     Acces *iptr = this->inodes;
310     Acces *dptr = this->dirBlocks;
311     Acces *bptr = this->blocks;
312
313     if (iptr != dptr && iptr != bptr)
314       delete this->inodes;
315
316     if (dptr != bptr)
317       delete this->dirBlocks;
318     delete this->blocks;
319   }
320
321   void Mtfs::getAttr(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) {
322     (void) fi;
323
324     internalInode_st *inode = this->getIntInode(ino);
325
326     if (nullptr == inode)
327       return (void) fuse_reply_err(req, ENOENT);
328
329     struct stat st{};
330     this->buildStat(*inode, st);
331
332     fuse_reply_attr(req, &st, this->ATTR_TIMEOUT);
333   }
334
335   void Mtfs::setAttr(fuse_req_t req, fuse_ino_t ino, struct stat *attr, int
       toSet, fuse_file_info *fi) {
336     (void) fi;
337
```

```
338     internalInode_st *inInode = this->getIntInode(ino);
339
340     if (0 != (FUSE_SET_ATTR_MODE & toSet))
341       inInode->inode.accesRight = attr->st_mode;
342
343     if (0 != (FUSE_SET_ATTR_UID & toSet))
344       inInode->inode.uid = attr->st_uid;
345
346     if (0 != (FUSE_SET_ATTR_GID & toSet))
347       inInode->inode.gid = attr->st_gid;
348
349     if (0 != (FUSE_SET_ATTR_SIZE & toSet))
350       inInode->inode.size = (uint64_t) attr->st_size;
351
352     if (0 != (FUSE_SET_ATTR_ATIME & toSet))
353       inInode->inode.atime = (uint64_t) attr->st_atim.tv_sec;
354
355     buildStat(*inInode, *attr);
356     fuse_reply_attr(req, attr, 1.0);
357
358     pool upThPool(this->SIMULT_UP);
359     for (auto &&id: inInode->idents) {
360       upThPool.schedule(bind(&Acces::put, this->inodes, id, &inInode->inode,
    INODE));
361     }
362   }
363
364   void Mtfs::lookup(fuse_req_t req, fuse_ino_t parent, const string &name) {
365     Logger::getInstance()->log("LOOKUP", name, Logger::L_DEBUG);
366
367     internalInode_st *parentInode = this->getIntInode(parent);
368
369 //     dl All blocks
370     dl_st blkDl = dl_st();
371     blkDl.dlThPool = new pool(this->SIMULT_DL);
372     blkDl.sem = new Semaphore();
373     blkDl.fifoMu = new mutex;
374     blkDl.fifo.dirQueue = new queue<dirBlock_t>();
375     blkDl.endMu = new mutex;
376
377     this->dlBlocks(parentInode->inode, &blkDl, blockType::DIR_BLOCK, 0);
378
379     vector<ident_t> inodeIds;
380
381     unique_lock<mutex> endLk(*blkDl.endMu, defer_lock);
382     unique_lock<mutex> fifoLk(*blkDl.fifoMu, defer_lock);
383     lock(endLk, fifoLk);
384     while (!(blkDl.end && blkDl.fifo.dirQueue->empty())) {
385       endLk.unlock();
386       fifoLk.unlock();
387
388       blkDl.sem->wait();
389
390       lock(endLk, fifoLk);
391       if (blkDl.end && blkDl.fifo.dirQueue->empty())
392         break;
393       endLk.unlock();
394
395       dirBlock_t block = blkDl.fifo.dirQueue->front();
```

```
396       blkDl.fifo.dirQueue->pop();
397       fifoLk.unlock();
398
399 //      if entry is find
400       if (block.entries.end() != block.entries.find(name)) {
401         blkDl.dlThPool->clear();
402         inodeIds = block.entries[name];
403
404         lock(endLk, fifoLk);
405         break;
406       }
407
408       lock(endLk, fifoLk);
409     }
410     endLk.unlock();
411     fifoLk.unlock();
412
413
414 //    TODO Doit etre un pointeur car lookup avant open donc inode doit etre en
    memoire.
415     auto *inode = new internalInode_st();
416     inode->idents = inodeIds;
417     int ret;
418
419     if (inodeIds.empty()) {
420       fuse_reply_err(req, ENOENT);
421       return;
422     }
423
424     do {
425       ret = this->inodes->get(inodeIds.back(), &inode->inode, blockType::INODE)
    ;
426       inodeIds.pop_back();
427       if (inodeIds.empty())
428         break;
429     } while (0 != ret);
430
431 //    Send reply to fuse
432     if (0 != ret)
433       fuse_reply_err(req, ret);
434     else {
435       fuse_entry_param param = fuse_entry_param();
436       this->buildParam(*inode, param);
437       fuse_reply_entry(req, &param);
438     }
439
440 //    free all datas;
441     blkDl.dlThPool->wait();
442
443     delete (blkDl.dlThPool);
444     delete (blkDl.sem);
445     delete (blkDl.fifoMu);
446     delete (blkDl.endMu);
447
448     while (!blkDl.fifo.dirQueue->empty())
449       blkDl.fifo.dirQueue->pop();
450
451     delete (blkDl.fifo.dirQueue);
452   }
```

```
453
454   void Mtfs::mknod(fuse_req_t req, fuse_ino_t parent, const string &name,
       mode_t mode, dev_t rdev) {
455     (void) rdev;
456
457     int ret;
458
459     internalInode_st *parentInode = this->getIntInode(parent);
460
461 //    Create and write new inode
462     internalInode_st *inode = this->newInode(mode, fuse_req_ctx(req));
463     vector<ident_t> inodeIdents;
464     if (0 != (ret = this->insertInode(inode->inode, inodeIdents))) {
465       delete inode;
466       fuse_reply_err(req, ret);
467       return;
468     }
469
470     pool mPool(SIMULT_UP);
471     this->initMetas(*parentInode, inodeIdents, blockType::INODE, &mPool);
472
473 //      Add entry in dir
474     if (0 != (ret = this->addEntry(parentInode, name, inodeIdents))) {
475       delete inode;
476       fuse_reply_err(req, ret);
477       return;
478     }
479
480 //      reply to fuse
481     fuse_entry_param param{};
482     memset(&param, 0, sizeof(fuse_entry_param));
483
484     this->buildParam(*inode, param);
485
486     fuse_reply_entry(req, &param);
487   }
488
489   void Mtfs::mkdir(fuse_req_t req, fuse_ino_t ino, const string &name, mode_t
       mode) {
490     int ret;
491     internalInode_st *parentInode = this->getIntInode(ino);
492
493 //    Create and write new inode
494     internalInode_st *inode = this->newInode(mode, fuse_req_ctx(req));
495     inode->inode.accesRight |= S_IFDIR;
496     vector<ident_t> inodeIdents;
497     if (0 != (ret = this->insertInode(inode->inode, inodeIdents))) {
498       delete inode;
499       fuse_reply_err(req, ret);
500       return;
501     }
502
503     pool mPool(SIMULT_UP);
504     this->initMetas(*parentInode, inodeIdents, blockType::INODE, &mPool);
505
506 //      Add entry in dir
507     if (0 != (ret = this->addEntry(parentInode, name, inodeIdents))) {
508       delete inode;
509       fuse_reply_err(req, ret);
```

```
510       return;
511     }
512
513 //      reply to fuse
514     fuse_entry_param param{};
515     memset(&param, 0, sizeof(fuse_entry_param));
516
517     this->buildParam(*inode, param);
518
519     fuse_reply_entry(req, &param);
520   }
521
522   void Mtfs::unlink(fuse_req_t req, fuse_ino_t parent, const string &name) {
523     internalInode_st *parentInode = this->getIntInode(parent);
524
525     int ret;
526     if (SUCCESS != (ret = this->doUnlink(parentInode, name)))
527       fuse_reply_err(req, ret);
528     else
529       fuse_reply_err(req, SUCCESS);
530   }
531
532   void Mtfs::rmdir(fuse_req_t req, fuse_ino_t parent, const string &name) {
533     (void) parent, name;
534
535     fuse_reply_err(req, ENOSYS);
536   }
537
538   void Mtfs::access(fuse_req_t req, fuse_ino_t ino, int mask) {
539
540     const struct fuse_ctx *context = fuse_req_ctx(req);
541
542     if (FUSE_ROOT_ID == ino) {
543       int ret = EACCES;
544       inode_t inode = instance->getRootInode();
545       if (context->uid == inode.uid) {
546         if ((mask << 6 & inode.accesRight) != 0)
547           ret = 0;
548         else {
549           cerr << "[MTFS]: bad user right: " << (inode.accesRight & (mask << 6)
    ) << endl;
550         }
551       } else {
552
553 //        get all user groups;
554         auto bufsize = (size_t) sysconf(_SC_GETPW_R_SIZE_MAX);
555         if (bufsize == -1)
556           bufsize = 16384;
557
558         auto *buf = (char *) malloc(bufsize * sizeof(char));
559         if (nullptr == buf) {
560           fuse_reply_err(req, ENOMEM);
561           free(buf);
562           return;
563         }
564
565         struct passwd pwd{}, *result;
566         int s = getpwuid_r(context->uid, &pwd, buf, bufsize, &result);
567         if (nullptr == result) {
```

```
568             int err;
569             if (0 == s)
570               err = EAGAIN;
571             else
572               err = s;
573             fuse_reply_err(req, err);
574             free(buf);
575             return;
576           }
577 
578           int ngroups = GROUPS_TO_SEARCH;
579           auto *groups = (gid_t *) malloc(ngroups * sizeof(gid_t));
580           if (getgrouplist(pwd.pw_name, pwd.pw_gid, groups, &ngroups) == -1) {
581             cerr << "get group list error. ngoups " << ngroups << endl;
582             free(buf);
583             free(groups);
584           }
585 
586           if (find(groups, groups + ngroups, inode.gid) != groups + ngroups) {
587             if ((mask << 3 & inode.accesRight) != 0)
588               ret = 0;
589           } else {
590             if ((mask & inode.accesRight) != 0)
591               ret = 0;
592           }
593 
594           free(buf);
595           free(groups);
596         }
597 
598         fuse_reply_err(req, ret);
599 
600 
601       } else {
602         fuse_reply_err(req, SUCCESS);
603       }
604     }
605 
606     void Mtfs::open(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) {
607       (void) ino;
608 //      internalInode_st *inode = getIntInode(ino);
609 
610 //      if (!inode->inode.dataBlocks.empty()) {
611 //        uint8_t *firstBlock = (uint8_t *) malloc(this->blockSize * sizeof(
    uint8_t));
612 //
613 //        this->blocks->get(inode->inode.dataBlocks.front().front(), &firstBlock,
     blockType::DATA_BLOCK);
614 //        fi->fh = (uint64_t) firstBlock;
615 //      } else
616 //        fi->fh = 0;A
617 
618       auto *fd = new fd_st();
619       fd->blockT = fd->DATA;
620       fd->blkDl.dlThPool = new pool(this->SIMULT_DL);
621       fd->blkDl.sem = new Semaphore();
622       fd->blkDl.fifoMu = new mutex();
623       fd->blkDl.endMu = new mutex();
624       fd->blkDl.fifo.blkQueue = new queue<uint8_t *>();
```

```
625     fd->inoDl.dlThPool = new pool(this->SIMULT_UP);
626
627     fi->fh = (uint64_t) fd;
628
629     fuse_reply_open(req, fi);
630   }
631
632   void Mtfs::release(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) {
633     (void) ino;
634 //    internalInode_st *intIno = this->getIntInode(ino);
635
636     delete (uint8_t *) fi->fh;
637     fi->fh = 0;
638
639     fuse_reply_err(req, SUCCESS);
640   }
641
642   void Mtfs::opendir(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) {
643
644     internalInode_st *intInode = this->getIntInode(ino);
645
646     auto *fd = new fd_st();
647     fd->blockT = fd->DIR;
648     fd->blkDl.dlThPool = new pool(this->SIMULT_DL);
649     fd->blkDl.sem = new Semaphore();
650     fd->blkDl.fifoMu = new mutex();
651     fd->blkDl.endMu = new mutex();
652     fd->blkDl.fifo.dirQueue = new queue<dirBlock_t>();
653
654     int i = 0;
655     for (auto &&blks: intInode->inode.dataBlocks) {
656
657 //      bind(&Mtfs::dlDirBlocks, this, blks, &blkQueue, &queueMutex, &semaphore
    ));
658       fd->blkDl.dlThPool->schedule(
659           bind(&Mtfs::dlDirBlocks, this, blks, fd->blkDl.fifo.dirQueue, fd->
    blkDl.fifoMu, fd->blkDl.sem));
660
661       if (this->INIT_DL <= i)
662         break;
663     }
664
665     fi->fh = (uint64_t) fd;
666
667     fuse_reply_open(req, fi);
668 //    fuse_reply_err(req, ENOSYS);
669   }
670
671   void Mtfs::readdir(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
     fuse_file_info *fi) {
672     this->doReaddir(req, ino, size, off, fi, false);
673   }
674
675   void Mtfs::readdirplus(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off
    , fuse_file_info *fi) {
676     this->doReaddir(req, ino, size, off, fi, true);
677   }
678
679   void Mtfs::releasedir(fuse_req_t req, fuse_ino_t ino, fuse_file_info *fi) {
```

```
680     (void) ino;
681
682     auto *fd = (fd_st *) fi->fh;
683     delete fd;
684     fi->fh = 0;
685
686     fuse_reply_err(req, SUCCESS);
687   }
688
689   void Mtfs::write(fuse_req_t req, fuse_ino_t ino, const char *buf, size_t size
      , off_t off, fuse_file_info *fi) {
690     (void) ino, buf, size, off, fi;
691 //    internalInode_st *inode = this->getIntInode(ino);
692 //
693 //    int startBlock = (int) (off / this->blockSize);
694 //
695 //    fuse_reply_write(req, size);
696     fuse_reply_err(req, ENOSYS);
697   }
698
699   void Mtfs::write_buf(fuse_req_t req, fuse_ino_t ino, fuse_bufvec *bufv, off_t
      off, fuse_file_info *fi) {
700     int ret = 0;
701
702     auto *fd = (fd_st *) fi->fh;
703
704     internalInode_st *inode = this->getIntInode(ino);
705
706     size_t size = bufv->buf[0].size;
707     auto firstBlock = (int) (off / this->blockSize);
708     auto lastBlock = (int) ((off + size - 1) / this->blockSize);
709
710     ruleInfo_t newBlockInfo = ruleInfo_st();
711     newBlockInfo.gid = inode->inode.gid;
712     newBlockInfo.uid = inode->inode.uid;
713     newBlockInfo.lastAccess = this->now();
714
715     size_t rem = size;
716     size_t write = 0;
717
718     pool metasTp(SIMULT_UP);
719
720     vector<uint64_t> toFree;
721
722     for (int blockToWrite = firstBlock; blockToWrite <= lastBlock; ++
      blockToWrite) {
723       auto *block = (uint8_t *) calloc(sizeof(uint8_t), this->blockSize);
724       memset(block, 0, this->blockSize);
725
726       vector<ident_t> blockIdents;
727       if (inode->inode.dataBlocks.empty() || inode->inode.dataBlocks.size() <=
      blockToWrite) {
728         if (SUCCESS !=
729           (ret = this->blocks->add(newBlockInfo, blockIdents, blockType::
      DATA_BLOCK,
730                                    (const int) this->redundancy))) {
731           fuse_reply_err(req, ret);
732           return;
733         }
```

```
734
735         this->initMetas(*inode, blockIdents, blockType::DATA_BLOCK, &metasTp);
736
737         inode->inode.dataBlocks.push_back(blockIdents);
738       } else {
739         blockIdents = inode->inode.dataBlocks[blockToWrite];
740
741         for (auto &&id: blockIdents) {
742           ret = this->blocks->get(id, block, blockType::DATA_BLOCK);
743           if (SUCCESS == ret)
744             break;
745         }
746         if (SUCCESS != ret)
747           return (void) fuse_reply_err(req, ret);
748       }
749
750       size_t startPos = blockToWrite == firstBlock ? off % this->blockSize : 0;
751       size_t endPos = (blockToWrite == lastBlock ? rem % (this->blockSize + 1)
    : this->blockSize) - startPos;
752
753       memcpy(block + startPos, bufv->buf->mem, endPos);
754       write += endPos;
755       rem -= endPos;
756
757       for (auto &&id: blockIdents) {
758         fd->blkDl.dlThPool->schedule(bind(&Acces::put, this->blocks, id, block,
     blockType::DATA_BLOCK));
759       }
760
761       toFree.push_back((uint64_t) block);
762     }
763
764     auto *data = (uint8_t *) malloc(size * sizeof(uint8_t));
765     memcpy(data, bufv->buf[0].mem, bufv->buf[0].size);
766
767     inode->inode.size = off + size;
768
769 //    TODO supprimer les block en trop.
770     vector<ident_t> toDel;
771     if (inode->inode.dataBlocks.size() > lastBlock + 1) {
772       for (auto &dataBlock : inode->inode.dataBlocks) {
773         for (auto &&id: dataBlock) {
774           toDel.push_back(id);
775         }
776       }
777
778       inode->inode.dataBlocks.erase(inode->inode.dataBlocks.begin() + lastBlock
     + 1,
779                     inode->inode.dataBlocks.end());
780     }
781
782     for (auto &&id: inode->idents) {
783       fd->inoDl.dlThPool->schedule(bind(&Acces::put, this->inodes, id, &inode->
    inode, blockType::INODE));
784     }
785
786     fd->blkDl.dlThPool->wait();
787     fd->inoDl.dlThPool->wait();
788
```

```
789     fuse_reply_write(req, write);
790
791     pool delPool(this->redundancy);
792     for (auto &&id: toDel) {
793       delPool.schedule(bind(&Acces::del, this->blocks, id, blockType::
    DATA_BLOCK));
794     }
795
796     for (auto &&ptr: toFree) {
797       free((uint8_t *) ptr);
798     }
799   }
800
801   void Mtfs::read(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
     fuse_file_info *fi) {
802     (void) fi;
803     internalInode_st *inode = this->getIntInode(ino);
804
805
806     size = min(size, inode->inode.size);
807     int firstBlock, lastBlock;
808     firstBlock = (int) (off / this->blockSize);
809     lastBlock = (int) ((off + size - 1) / this->blockSize);
810
811     size_t rem = size;
812     size_t read = 0;
813
814     char *buf;
815     buf = new char[size];
816     char *p = buf;
817
818     for (int blockToRead = firstBlock; blockToRead <= lastBlock; blockToRead++)
      {
819       uint8_t block[this->blockSize];
820       if (SUCCESS != this->blocks->get(inode->inode.dataBlocks[blockToRead][0],
     block, blockType::DATA_BLOCK)) {
821         delete[] buf;
822         fuse_reply_err(req, EAGAIN);
823
824         return;
825       }
826
827       size_t startPos = blockToRead == firstBlock ? off % this->blockSize : 0;
828       size_t endPos = blockToRead == lastBlock ? rem % (this->blockSize + 1) :
     this->blockSize;
829
830       size_t toCopy = endPos - startPos;
831       memcpy(p, block + startPos, toCopy);
832
833       rem -= toCopy;
834       read += toCopy;
835       p += toCopy;
836     }
837
838     fuse_reply_buf(req, buf, read);
839     delete[] buf;
840   }
841
```

```
842 //
    ////////////////////////////////////////////////////////////////////////////////
843 ////                          PRIVATE                                       ////
844 //
    ////////////////////////////////////////////////////////////////////////////////
845
846   Mtfs::Mtfs() : redundancy(1), maxEntryPerBlock(ENTRY_PER_BLOCK) {
847     this->inodes = nullptr;
848     this->dirBlocks = nullptr;
849     this->blocks = nullptr;
850     this->rootIn = new internalInode_st();
851   }
852
853   /**
854    * Get the root inode
855    *
856    * @return the inode
857    */
858   inode_t Mtfs::getRootInode() {
859     return this->rootIn->inode;
860 //    return this->rootInode;
861   }
862
863   /**
864    * Build MTFS system
865    *
866    * @param superblock
867    * @return
868    */
869   bool Mtfs::build(const superblock_t &superblock) {
870
871     this->redundancy = superblock.redundancy;
872     this->blockSize = superblock.blockSz;
873     this->rootIn->idents = superblock.rootInodes;
874
875     pluginSystem::PluginManager *manager = pluginSystem::PluginManager::
      getInstance();
876
877     PoolManager *poolManager;
878     poolManager = new PoolManager();
879     for (auto &&poolSt: superblock.pools) {
880       Pool *pool;
881       pool = new Pool(this->blockSize);
882
883       for (auto volSt: poolSt.second.volumes) {
884         volSt.second.params.insert(make_pair("home", MTFS_PLUGIN_HOME));
885         volSt.second.params.insert(make_pair("blockSize", to_string(this->
      blockSize)));
886
887         pluginSystem::Plugin *plugin = manager->getPlugin(volSt.second.
      pluginName);
888
889         if (!plugin->attach(volSt.second.params)) {
890           cerr << "Failed attach plugin " << volSt.second.pluginName << endl;
891         }
892
893         pool->addVolume(volSt.first, new Volume(plugin), volSt.second.rule);
```

```
894         }
895
896         poolManager->addPool(poolSt.first, pool, poolSt.second.rule);
897       }
898
899       this->inodes = poolManager;
900       this->dirBlocks = poolManager;
901       this->blocks = poolManager;
902
903       return true;
904     }
905
906     /**
907      * Read the root inode
908      */
909     void Mtfs::readRootInode() {
910       string filename = string(MTFS_INSTALL_DIR) + "/" + systemName + "/root.json
          ";
911       ifstream file(filename);
912       if (!file.is_open())
913         return;
914
915       IStreamWrapper wrapper(file);
916
917       Document d(kObjectType);
918       d.ParseStream(wrapper);
919
920       assert(d.HasMember(IN_MODE));
921       this->rootIn->inode.accesRight = d[IN_MODE].GetUint();
922
923       assert(d.HasMember(IN_LINKS));
924       this->rootIn->inode.linkCount = (uint8_t) d[IN_LINKS].GetUint();
925
926       assert(d.HasMember(IN_UID));
927       this->rootIn->inode.uid = d[IN_UID].GetUint();
928
929       assert(d.HasMember(IN_GID));
930       this->rootIn->inode.gid = d[IN_GID].GetUint();
931
932       assert(d.HasMember(IN_SIZE));
933       this->rootIn->inode.size = d[IN_SIZE].GetUint64();
934
935       assert(d.HasMember(IN_ACCESS));
936       this->rootIn->inode.atime = d[IN_ACCESS].GetUint64();
937
938       assert(d.HasMember(IN_BLOCKS));
939       for (auto &&item: d[IN_BLOCKS].GetArray()) {
940         vector<ident_t> blocksRedundancy;
941         for (auto &&block: item.GetArray()) {
942           ident_t ident = ident_t();
943
944           assert(block.HasMember(ID_POOL));
945           ident.poolId = block[ID_POOL].GetUint();
946
947           assert(block.HasMember(ID_VOLUME));
948           ident.volumeId = block[ID_VOLUME].GetUint();
949
950           assert(block.HasMember(ID_ID));
951           ident.id = block[ID_ID].GetUint64();
```

```
952
953         blocksRedundancy.push_back(ident);
954       }
955       this->rootIn->inode.dataBlocks.push_back(blocksRedundancy);
956     }
957   }
958
959   /**
960    * Write the root inode
961    */
962   void Mtfs::writeRootInode() {
963     Document d;
964
965     this->rootIn->inode.toJson(d);
966
967     StringBuffer sb;
968     PrettyWriter<StringBuffer> pw(sb);
969     d.Accept(pw);
970
971     string filename = string(MTFS_INSTALL_DIR) + "/" + systemName + "/root.json
      ";
972     ofstream rootFile(filename);
973     rootFile << sb.GetString() << endl;
974     rootFile.close();
975   }
976
977   /**
978    * @brief Add entry in directory.
979    *
980    * This function add a new entry in directory and update parent inode if
      necessary.
981    *
982    * @param parentInode   Directory to add entry
983    * @param name      Name of entry
984    * @param inodeIds    Inodes od entry
985    *
986    * @return          0 if success else errno.
987    */
988   int Mtfs::addEntry(internalInode_st *parentInode, std::string name, vector<
      ident_t> &inodeIds) {
989     int ret = 0;
990
991     dirBlock_t dirBlock = dirBlock_t();
992
993     vector<ident_t> blockIdents;
994
995     pool iPool(parentInode->idents.size());
996     pool mPool(SIMULT_UP);
997
998 //    if directory is empty add one block
999     if (parentInode->inode.dataBlocks.empty()) {
1000       if (0 !=
1001         (ret = this->dirBlocks->add(getRuleInfo(parentInode->inode),
      blockIdents, blockType::DIR_BLOCK,
1002                       this->redundancy))) {
1003         return ret;
1004       }
1005
1006       dirBlock.entries.clear();
```

```
1007
1008        this->initMetas(*parentInode, blockIdents, blockType::DIR_BLOCK, &mPool);
1009
1010        parentInode->inode.atime = this->now();
1011        parentInode->inode.dataBlocks.push_back(blockIdents);
1012        for (auto &&ident: parentInode->idents) {
1013          iPool.schedule(bind(&Acces::put, this->inodes, ident, &parentInode->
    inode, blockType::INODE));
1014        }
1015        if (parentInode == this->rootIn)
1016          this->writeRootInode();
1017
1018      } else {
1019//        else get the first block.
1020        blockIdents = parentInode->inode.dataBlocks.back();
1021
1022        for (const auto &blockIdent : blockIdents) {
1023          if (0 == (ret = this->dirBlocks->get(blockIdent, &dirBlock, blockType::
    DIR_BLOCK)))
1024            break;
1025        }
1026
1027        if (ret != SUCCESS)
1028          return ret;
1029      }
1030
1031
1032//      if block is full, allocate new block.
1033      if (this->maxEntryPerBlock == dirBlock.entries.size()) {
1034        ruleInfo_t info = getRuleInfo(parentInode->inode);
1035        info.lastAccess = this->now();
1036        if (0 != (ret = this->dirBlocks->add(info, blockIdents, blockType::
    DIR_BLOCK, this->redundancy))) {
1037          return ret;
1038        }
1039
1040        this->initMetas(*parentInode, blockIdents, blockType::DIR_BLOCK, &mPool);
1041
1042        dirBlock.entries.clear();
1043
1044        parentInode->inode.atime = this->now();
1045        parentInode->inode.dataBlocks.push_back(blockIdents);
1046        for (auto &&ident: parentInode->idents) {
1047          iPool.schedule(bind(&Acces::put, this->inodes, ident, &parentInode->
    inode, blockType::INODE));
1048        }
1049        if (parentInode == this->rootIn)
1050          this->writeRootInode();
1051      }
1052
1053
1054//      write block
1055      dirBlock.entries.insert(make_pair(name, inodeIds));
1056      pool wpool(blockIdents.size());
1057      for (auto &&ident: blockIdents) {
1058//        TODO try again if block not write.
1059        wpool.schedule(bind(&Acces::put, this->dirBlocks, ident, &dirBlock,
    blockType::DIR_BLOCK));
1060      }
```

```
1061
1062      return 0;
1063    }
1064
1065    /**
1066     * @brief Insert inode in system.
1067     *
1068     * This function get free inode Id and put inode in volumes
1069     *
1070     * @param[in] inode Inode to put
1071     * @param[out] idents Ids to new inode
1072     *
1073     * @return 0 if SUCCESS else @see satic const vars.
1074     */
1075    int Mtfs::insertInode(const inode_t &inode, vector<ident_t> &idents) {
1076      int ret;
1077
1078  //    get new inode idents
1079      if (0 != (ret = this->inodes->add(this->getRuleInfo(inode), idents,
     blockType::INODE, this->redundancy))) {
1080        return ret;
1081      }
1082
1083  //    write inode in volumes
1084      {
1085        pool thPool(idents.size());
1086        for (auto &&ident: idents) {
1087          thPool.schedule(bind(&Acces::put, this->inodes, ident, &inode,
     blockType::INODE));
1088        }
1089      }
1090
1091      return ret;
1092    }
1093
1094    /**
1095     * Download all blocks in inode
1096     *
1097     * @param inode Inode who containe the blocks
1098     * @param dlSt Download struct
1099     * @param type Type of block to dowload DIR_BLOCK | DATA_BLOCK
1100     * @param firstBlockIdx Index of first block to download
1101     */
1102    void Mtfs::dlBlocks(const inode_t &inode, dl_st *dlSt, const blockType type,
     const int firstBlockIdx) {
1103  //    TODO implements
1104
1105      if (firstBlockIdx < inode.dataBlocks.size()) {
1106        for (auto idents = inode.dataBlocks.begin() + firstBlockIdx; idents !=
     inode.dataBlocks.end(); idents++) {
1107          switch (type) {
1108            case DIR_BLOCK:
1109              dlSt->dlThPool->schedule(
1110                  bind(&Mtfs::dlDirBlocks, this, *idents, dlSt->fifo.dirQueue,
     dlSt->fifoMu, dlSt->sem));
1111              break;
1112            case DATA_BLOCK:
1113              break;
1114            default:
```

```
1115 //           TODO Log error
1116              break;
1117            }
1118          }
1119        }
1120
1121        dlSt->dlThPool->wait();
1122
1123        unique_lock<mutex> endLk(*dlSt->endMu);
1124        dlSt->end = true;
1125        endLk.unlock();
1126        dlSt->sem->notify();
1127      }
1128
1129      /**
1130       * @brief download or get a directory block.
1131       *
1132       * This function is create for work in a other thread than worker.
1133       *
1134       * @param[in] ids   Block ids
1135       * @param[out] q  Queue
1136       * @param[in] queueMutex
1137       * @param[in] sem
1138       */
1139      void Mtfs::dlDirBlocks(vector<ident_t> &ids, queue<dirBlock_t> *q, mutex *
          queueMutex, Semaphore *sem) {
1140 //   TODO gerer le cas ou aucun bloc n'a pu etre dl.
1141        int ret = 1;
1142        dirBlock_t db = dirBlock_t();
1143
1144        for (auto &&id: ids) {
1145          ret = this->dirBlocks->get(id, &db, blockType::DIR_BLOCK);
1146          if (SUCCESS == ret)
1147            break;
1148        }
1149
1150        if (SUCCESS == ret) {
1151          unique_lock<mutex> lk(*queueMutex);
1152          q->push(db);
1153          sem->notify();
1154        }
1155      }
1156
1157      /**
1158       * Download all inodes from dir entries.
1159       *
1160       * @param src dl dir entries struct
1161       * @param dst dl inode struct
1162       */
1163      void Mtfs::dlInodes(dl_st *src, dl_st *dst) {
1164        pool inodesPool(this->SIMULT_DL);
1165        unique_lock<mutex> srcEndLk(*src->endMu, defer_lock);
1166        unique_lock<mutex> srcFifoLk(*src->fifoMu, defer_lock);
1167        std::lock(srcEndLk, srcFifoLk);
1168        while (!(src->end && src->fifo.dirQueue->empty())) {
1169          srcEndLk.unlock();
1170          srcFifoLk.unlock();
1171
1172          src->sem->wait();
```

```
1173
1174        std::lock(srcEndLk, srcFifoLk);
1175        if (src->end && src->fifo.dirQueue->empty())
1176          break;
1177        srcEndLk.unlock();
1178
1179        dirBlock_t dirBlock = src->fifo.dirQueue->front();
1180        src->fifo.dirQueue->pop();
1181        srcFifoLk.unlock();
1182
1183        for (auto &&entry: dirBlock.entries) {
1184          inodesPool.schedule(
1185              bind(&Mtfs::dlInode, this, entry.second, dst->fifo.inodeQueue, dst
    ->fifoMu, dst->sem,
1186                   entry.first));
1187        }
1188
1189        std::lock(srcEndLk, srcFifoLk);
1190      }
1191      srcEndLk.unlock();
1192      srcFifoLk.unlock();
1193
1194      inodesPool.wait();
1195
1196      unique_lock<mutex> dstEndLock(*dst->endMu);
1197      dst->end = true;
1198      dstEndLock.unlock();
1199
1200      dst->sem->notify();
1201    }
1202
1203    /**
1204     * Download inodes
1205     *
1206     * @param ids
1207     * @param queue
1208     * @param queueMutex
1209     * @param sem
1210     * @param key
1211     */
1212    void
1213    Mtfs::dlInode(vector<ident_t> &ids, queue<pair<string, inode_t>> *queue,
     mutex *queueMutex, Semaphore *sem,
1214            string &key) {
1215      int ret = 1;
1216      inode_t in = inode_t();
1217
1218      for (auto &&id: ids) {
1219        ret = this->inodes->get(id, &in, blockType::INODE);
1220        if (SUCCESS == ret)
1221          break;
1222      }
1223
1224      if (SUCCESS == ret) {
1225        unique_lock<mutex> lk(*queueMutex);
1226        queue->push(make_pair(key, in));
1227        sem->notify();
1228      }
1229    }
```

```
1230
1231  internalInode_st *Mtfs::getIntInode(fuse_ino_t ino) {
1232    if (FUSE_ROOT_ID == ino)
1233      return this->rootIn;
1234
1235    return (internalInode_st *) ino;
1236  }
1237
1238  void Mtfs::doReaddir(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
      fuse_file_info *fi, const bool &plus) {
1239    (void) off;
1240    internalInode_st *inode = this->getIntInode(ino);
1241    fd_st *fd;
1242    fd = (fd_st *) fi->fh;
1243
1244    char *buf, *p;
1245    size_t rem, currSize;
1246
1247    buf = (char *) calloc(size, 1);
1248    if (nullptr == buf)
1249      return (void) fuse_reply_err(req, ENOMEM);
1250
1251    p = buf;
1252    rem = size;
1253    currSize = 0;
1254
1255    if (fd->firstCall) {
1256      fd->firstCall = false;
1257      size_t entrySize = this->dirBufAdd(req, p, currSize, ".", *inode, plus);
1258      p += entrySize;
1259      rem -= entrySize;
1260      currSize += entrySize;
1261
1262//      Dl other blocks and inodes
1263      int idx = this->INIT_DL;
1264      fd->blkThr = new boost::thread(
1265          bind(&Mtfs::dlBlocks, this, inode->inode, &fd->blkDl, blockType::
      DIR_BLOCK, idx));
1266
1267      fd->inoDl.fifoMu = new mutex();
1268      fd->inoDl.endMu = new mutex();
1269      fd->inoDl.sem = new Semaphore();
1270      fd->inoDl.fifo.inodeQueue = new queue<pair<string, inode_t>>();
1271
1272      fd->inoThr = new boost::thread(bind(&Mtfs::dlInodes, this, &fd->blkDl, &
      fd->inoDl));
1273    }
1274
1275    unique_lock<mutex> inoEndLock(*fd->inoDl.endMu, defer_lock);
1276    unique_lock<mutex> inoFifoLock(*fd->inoDl.fifoMu, defer_lock);
1277    lock(inoEndLock, inoFifoLock);
1278    while (!(fd->inoDl.end && fd->inoDl.fifo.inodeQueue->empty())) {
1279      inoEndLock.unlock();
1280      inoFifoLock.unlock();
1281
1282      fd->inoDl.sem->wait();
1283
1284      internalInode_st *intInode;
1285      intInode = new internalInode_st();
```

```
1286 //       shared_ptr<internalInode_st> intInode(new internalInode_st());
1287       lock(inoEndLock, inoFifoLock);
1288       if (fd->inoDl.end && fd->inoDl.fifo.inodeQueue->empty())
1289         break;
1290       inoEndLock.unlock();
1291 
1292       string entryName = fd->inoDl.fifo.inodeQueue->front().first;
1293       intInode->inode = fd->inoDl.fifo.inodeQueue->front().second;
1294       fd->inoDl.fifo.inodeQueue->pop();
1295       inoFifoLock.unlock();
1296 
1297 
1298       size_t entrySize = this->dirBufAdd(req, p, currSize, entryName, *intInode
     , plus);
1299 
1300       if (entrySize > rem) {
1301         lock(inoEndLock, inoFifoLock);
1302         break;
1303       }
1304 
1305 //       p += entrySize;
1306       rem -= entrySize;
1307       currSize += entrySize;
1308 
1309       lock(inoEndLock, inoFifoLock);
1310       break;
1311     }
1312     inoEndLock.unlock();
1313     inoFifoLock.unlock();
1314     fuse_reply_buf(req, buf, size - rem);
1315   }
1316 
1317   size_t Mtfs::dirBufAdd(fuse_req_t &req, char *buf, size_t &currentSize, std::
     string name, internalInode_st &inode,
1318                          const bool &plus) {
1319     size_t entSize = 0;
1320 
1321     if (plus) {
1322       fuse_entry_param *p;
1323       p = new fuse_entry_param();
1324       buildParam(inode, *p);
1325 
1326       entSize += fuse_add_direntry_plus(req, nullptr, 0, name.c_str(), nullptr,
     0);
1327       fuse_add_direntry_plus(req, buf, entSize, name.c_str(), p, currentSize +
     entSize);
1328     } else {
1329       struct stat stbuf{};
1330       memset(&stbuf, 0, sizeof(stbuf));
1331       buildStat(inode, stbuf);
1332 
1333       entSize += fuse_add_direntry(req, nullptr, 0, name.c_str(), nullptr, 0);
1334       fuse_add_direntry(req, buf, entSize, name.c_str(), &stbuf, currentSize +
     entSize);
1335     }
1336     return entSize;
1337   }
1338 
```

```
1339  void Mtfs::buildParam(const internalInode_st &inode, fuse_entry_param &param)
        {
1340    param.ino = (fuse_ino_t) &inode;
1341
1342    this->buildStat(inode, param.attr);
1343
1344    param.generation = 1;
1345    param.attr_timeout = 0.0;
1346    param.entry_timeout = 0.0;
1347  }
1348
1349  void Mtfs::buildStat(const internalInode_st &inode, struct stat &st) {
1350    st.st_dev = 0;
1351    st.st_ino = (__ino_t) &inode;
1352
1353    st.st_mode = inode.inode.accesRight;
1354    st.st_nlink = inode.inode.linkCount;
1355    st.st_uid = inode.inode.uid;
1356    st.st_gid = inode.inode.gid;
1357    st.st_size = inode.inode.size;
1358    time_t time = inode.inode.atime;
1359    st.st_atim.tv_sec = time;
1360    st.st_ctim.tv_sec = time;
1361    st.st_mtim.tv_sec = time;
1362    st.st_blksize = this->blockSize;
1363    st.st_blocks = inode.inode.dataBlocks.size();
1364  }
1365
1366  ruleInfo_t Mtfs::getRuleInfo(const inode_t &inode) {
1367    return {inode.uid, inode.gid, inode.atime};
1368  }
1369
1370  internalInode_st *Mtfs::newInode(const mode_t &mode, const fuse_ctx *ctx) {
1371    internalInode_st *inode;
1372    inode = new internalInode_st();
1373
1374    inode->inode.accesRight = mode;
1375    inode->inode.uid = ctx->uid;
1376    inode->inode.gid = ctx->gid;
1377    if ((mode & S_IFDIR) != 0)
1378      inode->inode.linkCount = 2;
1379
1380    return inode;
1381  }
1382
1383  uint64_t Mtfs::now() {
1384    return (uint64_t) time(nullptr);
1385  }
1386
1387  /**
1388   * Init metas blocs
1389   *
1390   * @param parentInode
1391   * @param ids
1392   * @param type
1393   * @param thPool
1394   */
1395  void Mtfs::initMetas(const internalInode_st &parentInode, const vector<
       ident_t> &ids, const blockType &type,
```

```
1396                boost::threadpool::pool *thPool) {
1397     blockInfo_t metas = blockInfo_t();
1398     metas.lastAccess = (uint64_t) time(nullptr);
1399     metas.referenceId = parentInode.idents;
1400
1401     for (auto &&id :ids) {
1402       metas.id = id;
1403
1404       switch (type) {
1405         case INODE:
1406           if (nullptr != thPool)
1407             thPool->schedule(bind(&Acces::putMetas, this->inodes, id, metas,
     type));
1408           else
1409             this->inodes->putMetas(id, metas, type);
1410           break;
1411         case DIR_BLOCK:
1412           if (nullptr != thPool)
1413             thPool->schedule(bind(&Acces::putMetas, this->dirBlocks, id, metas,
      type));
1414           else
1415             this->dirBlocks->putMetas(id, metas, type);
1416           break;
1417         case DATA_BLOCK:
1418           if (nullptr != thPool)
1419             thPool->schedule(bind(&Acces::putMetas, this->blocks, id, metas,
     type));
1420           else
1421             this->blocks->putMetas(id, metas, type);
1422           break;
1423         default:
1424           break;
1425       }
1426     }
1427   }
1428
1429   int Mtfs::doUnlink(internalInode_st *parent, const std::string name) {
1430
1431     for (auto &&dBlkIds :parent->inode.dataBlocks) {
1432       dirBlock_t dBlk = dirBlock_t();
1433
1434       this->dirBlocks->get(dBlkIds.front(), &dBlk, blockType::DIR_BLOCK);
1435
1436       if (dBlk.entries.end() != dBlk.entries.find(name))
1437         return this->delEntry(dBlkIds, dBlk, name);
1438     }
1439     return ENOENT;
1440   }
1441
1442   int Mtfs::delEntry(std::vector<ident_t> &ids, dirBlock_t &blk, const std::
     string &name) {
1443     vector<ident_t> inodeIds = blk.entries.find(name)->second;
1444
1445     inode_t inode = inode_t();
1446     this->inodes->get(inodeIds.front(), &inode, blockType::INODE);
1447
1448     if (0 < (inode.accesRight & S_IFDIR))
1449       return EISDIR;
1450
```

```
1451      blk.entries.erase(name);
1452      pool upPool(SIMULT_UP);
1453
1454      for (auto &&id :ids) {
1455        upPool.schedule(bind(&Acces::put, this->dirBlocks, id, &blk, blockType::
      DIR_BLOCK));
1456      }
1457
1458      for (auto &&dataBlkIds :inode.dataBlocks) {
1459        for (auto &&blkId :dataBlkIds) {
1460          this->blocks->del(blkId, blockType::DATA_BLOCK);
1461        }
1462      }
1463
1464      for (auto &&inodeId :inodeIds) {
1465        this->inodes->del(inodeId, blockType::INODE);
1466      }
1467
1468      return 0;
1469    }
1470
1471
1472  }  // namespace mtfs
```

Listing 3.19 – "core/Mtfs.cpp"

3.4.4 Pool

Pool.h

```
1  /**
2   * \file Pool.h
3   * \brief
4   * \author David Wittwer
5   * \version 0.0.1
6   * \copyright GNU Publis License V3
7   *
8   * This file is part of MTFS.
9
10     MTFS is free software: you can redistribute it and/or modify
11     it under the terms of the GNU General Public License as published by
12     the Free Software Foundation, either version 3 of the License, or
13     (at your option) any later version.
14
15     Foobar is distributed in the hope that it will be useful,
16     but WITHOUT ANY WARRANTY; without even the implied warranty of
17     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18     GNU General Public License for more details.
19
20     You should have received a copy of the GNU General Public License
21     along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22   */
23
24 #ifndef FILESTORAGE_POOL_H
25 #define FILESTORAGE_POOL_H
26
27 #include <string>
28 #include <vector>
```

```
29 #include <mtfs/structs.h>
30 //#include <mtfs/Volume.h>
31 #include <mtfs/Rule.h>
32
33 namespace mtfs {
34   class Volume;
35
36   class Pool {
37
38   public:
39     static constexpr const char *POOLS = "pools";
40
41 //     STATUS CODE
42     static const int SUCCESS = 0;
43     static const int VOLUME_ID_EXIST = 1;
44     static const int NO_VALID_VOLUME = 2;
45
46
47   public:
48     Pool(const size_t &blkSize);
49
50     virtual ~Pool();
51
52     static bool validate(const rapidjson::Value &pool);
53
54     static void
55     structToJson(const pool_t &pool, rapidjson::Value &dest, rapidjson::
     Document::AllocatorType &allocator);
56
57     static void jsonToStruct(rapidjson::Value &src, pool_t &pool);
58
59     int addVolume(uint32_t volumeId, Volume *volume, Rule *rule);
60
61     int add(const ruleInfo_t &info, std::vector<ident_t> &idents, const
     blockType &type, const int &nb = 1);
62
63     int del(const uint32_t &volumeId, const uint64_t &id, const blockType &type
     );
64
65     int get(const uint32_t &volumeId, const uint64_t &id, void *data, const
     blockType &type);
66
67     int put(const uint32_t &volumeId, const uint64_t &id, const void *data,
     const blockType &type);
68
69     int getMetas(const uint32_t &volumeId, const uint64_t &id, blockInfo_t &
     metas, const blockType &type);
70
71     int putMetas(const uint32_t &volumeId, const uint64_t &id, const
     blockInfo_t &metas, const blockType &type);
72
73     void
74     doMigration(std::map<ident_t, ident_t> &movedBlk, std::vector<ident_t> &
     unsatisfyBlk, const blockType &type);
75
76
77   private:
78     const size_t blockSize;
79
```

```
80      std::map<uint32_t, Volume *> volumes;
81      std::map<uint32_t, Rule *> rules;
82
83
84      int getValidVolumes(const ruleInfo_t &info, std::vector<uint32_t> &
        volumeIds);
85    };
86
87  }  // namespace mtfs
88  #endif
```

Listing 3.20 – "core/Pool.h"

Pool.cpp

```
1   /**
2    * \file Pool.cpp
3    * \brief
4    * \author David Wittwer
5    * \version 0.0.1
6    * \copyright GNU Publis License V3
7    *
8    * This file is part of MTFS.
9
10     MTFS is free software: you can redistribute it and/or modify
11     it under the terms of the GNU General Public License as published by
12     the Free Software Foundation, either version 3 of the License, or
13     (at your option) any later version.
14
15     Foobar is distributed in the hope that it will be useful,
16     but WITHOUT ANY WARRANTY; without even the implied warranty of
17     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18     GNU General Public License for more details.
19
20     You should have received a copy of the GNU General Public License
21     along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22   */
23
24  #include <utils/Logger.h>
25  #include "mtfs/Pool.h"
26
27  using namespace std;
28
29  namespace mtfs {
30    Pool::Pool(const size_t &blkSize) : blockSize(blkSize) {}
31
32    Pool::~Pool() {
33      for (auto &&vol: this->volumes) {
34        delete vol.second;
35      }
36      for (auto &&rule: this->rules) {
37        delete rule.second;
38      }
39    }
40
41    bool Pool::validate(const rapidjson::Value &pool) {
42      if (!pool.HasMember(mtfs::Volume::VOLUMES))
43        throw invalid_argument("Volumes missing!");
44      if (!pool[mtfs::Volume::VOLUMES].IsObject())
```

```
45        throw invalid_argument("Volumes is not a object");
46
47      int migration = -1;
48      if (pool[mtfs::Volume::VOLUMES].MemberCount() <= 0)
49        throw invalid_argument("Number of volumes invalid!");
50      else if (pool[mtfs::Volume::VOLUMES].MemberCount() != 1) {
51        if (!pool.HasMember(Rule::MIGRATION))
52          throw invalid_argument("Migration missing!");
53
54        migration = pool[Rule::MIGRATION].GetInt();
55      }
56
57      for (auto &m: pool[mtfs::Volume::VOLUMES].GetObject()) {
58        if (Rule::rulesAreValid(migration, m.value) != Rule::VALID_RULES)
59          throw invalid_argument(string("Rules invalid for volume '") + m.name.
      GetString() + "'!");
60
61        if (!Volume::validate(m.value))
62          throw invalid_argument(string("Volume '") + m.name.GetString() + "'
      invalid!");
63      }
64
65      return true;
66    }
67
68    void Pool::structToJson(const pool_t &pool, rapidjson::Value &dest, rapidjson
      ::Document::AllocatorType &allocator) {
69      dest.AddMember(rapidjson::StringRef(Rule::MIGRATION), rapidjson::Value(pool
      .migration), allocator);
70
71      pool.rule->toJson(dest, allocator);
72
73      rapidjson::Value volumes(rapidjson::kObjectType);
74      for (auto &&item : pool.volumes) {
75        rapidjson::Value volume(rapidjson::kObjectType);
76
77        Volume::structToJson(item.second, volume, allocator);
78
79        string id = to_string(item.first);
80        rapidjson::Value index(id.c_str(), (rapidjson::SizeType) id.size(),
      allocator);
81        volumes.AddMember(index, volume, allocator);
82      }
83
84      dest.AddMember(rapidjson::StringRef(Volume::VOLUMES), volumes, allocator);
85    }
86
87    void Pool::jsonToStruct(rapidjson::Value &src, pool_t &pool) {
88      assert(src.HasMember(Rule::MIGRATION));
89      pool.migration = src[Rule::MIGRATION].GetInt();
90
91      assert(src.HasMember(Volume::VOLUMES));
92      for (auto &&item :src[Volume::VOLUMES].GetObject()) {
93        uint32_t id = (uint32_t) stoul(item.name.GetString());
94        volume_t volume;
95        memset(&volume, 0, sizeof(volume_t));
96        volume.params.clear();
97
98        volume.rule = Rule::buildRule(pool.migration, item.value);
```

```
99
100       Volume::jsonToStruct(item.value, volume);
101
102       pool.volumes.insert(make_pair(id, volume));
103     }
104
105   }
106
107   int Pool::addVolume(uint32_t volumeId, Volume *volume, Rule *rule) {
108     if (this->volumes.find(volumeId) != this->volumes.end() && this->rules.find
      (volumeId) != this->rules.end())
109       return VOLUME_ID_EXIST;
110
111     this->volumes[volumeId] = volume;
112     this->rules[volumeId] = rule;
113
114     rule->configureStorage(volume);
115
116     return 0;
117   }
118
119   int Pool::add(const ruleInfo_t &info, std::vector<ident_t> &idents, const
      blockType &type, const int &nb) {
120     int ret;
121     vector<uint32_t> volumeIds;
122     if (0 != (ret = this->getValidVolumes(info, volumeIds))) {
123       return ret;
124     }
125
126     if (0 == volumeIds.size())
127       return NO_VALID_VOLUME;
128
129     if (nb <= volumeIds.size()) {
130       for (int i = 0; i < nb; i++) {
131         uint64_t id = 0;
132         const uint32_t vid = volumeIds[i];
133
134         this->volumes[vid]->add(id, type);
135
136         idents.push_back(ident_t(id, vid));
137       }
138     } else {
139       int blkPerPool = (int) (nb / volumeIds.size());
140       int remainder = (int) (nb % volumeIds.size());
141       for (auto &&vid: volumeIds) {
142         vector<uint64_t> tmpIds;
143         int nbToAllocate = blkPerPool;
144
145         if (0 < remainder) {
146           nbToAllocate++;
147           remainder--;
148         }
149
150         this->volumes[vid]->add(tmpIds, nbToAllocate, type);
151
152         for (auto &&tmpId: tmpIds) {
153           idents.push_back(ident_t(tmpId, vid));
154         }
155       }
```

```
156     }
157
158     return ret;
159   }
160
161   int Pool::del(const uint32_t &volumeId, const uint64_t &id, const blockType &
      type) {
162     return this->volumes[volumeId]->del(id, type);
163   }
164
165   int Pool::get(const uint32_t &volumeId, const uint64_t &id, void *data, const
       blockType &type) {
166     return this->volumes[volumeId]->get(id, data, type);
167   }
168
169   int Pool::put(const uint32_t &volumeId, const uint64_t &id, const void *data,
       const blockType &type) {
170     return this->volumes[volumeId]->put(id, data, type);
171   }
172
173   int Pool::getMetas(const uint32_t &volumeId, const uint64_t &id, blockInfo_t
      &metas, const blockType &type) {
174     return this->volumes[volumeId]->getMetas(id, metas, type);
175   }
176
177   int Pool::putMetas(const uint32_t &volumeId, const uint64_t &id, const
      blockInfo_t &metas, const blockType &type) {
178     return this->volumes[volumeId]->putMetas(id, metas, type);
179   }
180
181   int Pool::getValidVolumes(const ruleInfo_t &info, vector<uint32_t> &volumeIds
      ) {
182     for (auto &&item: this->rules) {
183       if (item.second->satisfyRules(info))
184         volumeIds.push_back(item.first);
185     }
186     return 0;
187   }
188
189   void
190   Pool::doMigration(std::map<ident_t, ident_t> &movedBlk, std::vector<ident_t>
      &unsatisfyBlk, const blockType &type) {
191     for (auto &&volume :this->volumes) {
192       Logger::getInstance()->log("Pool.doMigration", "do migration for volume:
      " + to_string(volume.first),
193                                  Logger::L_DEBUG);
194       vector<blockInfo_t> unsatisfy;
195       unsatisfy.clear();
196       volume.second->getUnsatisfy(unsatisfy, type);
197
198       for (auto &&blkInfos :unsatisfy) {
199         ident_t oldIdent(blkInfos.id.id, volume.first);
200         blkInfos.id.volumeId = volume.first;
201
202         ruleInfo_t info = ruleInfo_t();
203         info.lastAccess = blkInfos.lastAccess;
204
205         vector<uint32_t> newVolumes;
206         this->getValidVolumes(info, newVolumes);
```

```
207
208         ident_t newIdent(0, newVolumes.front());
209         this->volumes[newIdent.volumeId]->add(newIdent.id, type);
210
211         if (oldIdent.volumeId == newIdent.volumeId) {
212           continue;
213         }
214
215         void *datas = nullptr;
216         switch (type) {
217           case INODE:
218             datas = new inode_t();
219             break;
220           case DIR_BLOCK:
221             datas = new dirBlock_t();
222             break;
223           case DATA_BLOCK:
224             datas = new uint8_t[this->blockSize];
225             break;
226           default:
227             continue;
228         }
229         this->volumes[oldIdent.volumeId]->get(oldIdent.id, datas, type);
230
231         this->volumes[newIdent.volumeId]->put(newIdent.id, datas, type);
232         this->volumes[newIdent.volumeId]->putMetas(newIdent.id, blkInfos, type)
    ;
233
234         movedBlk.emplace(oldIdent, newIdent);
235
236 //         this->volumes[oldIdent.volumeId]->del(oldIdent.id, type);
237
238         switch (type) {
239           case INODE:
240             delete ((inode_t *) datas);
241             break;
242           case DIR_BLOCK:
243             delete ((dirBlock_t *) datas);
244             break;
245           case DATA_BLOCK:
246             delete[]((uint8_t *) datas);
247             break;
248           default:
249             break;
250         }
251       }
252
253       unsigned long nb = unsatisfy.size();
254
255       if (0 != nb)
256         Logger::getInstance()->log("Pool.doMigration", "Block need to move: " +
     to_string(nb), Logger::L_DEBUG);
257     }
258   }
259
260
261 }  // namespace mtfs
```

Listing 3.21 – "core/Pool.cpp"

3.4.5 PoolManager

PoolManager.h

```
/**
 * \file PoolManager.h
 * \brief
 * \author David Wittwer
 * \version 0.0.1
 * \copyright GNU Publis License V3
 *
 * This file is part of MTFS.

    MTFS is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Foobar is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
 */

#ifndef FILESTORAGE_POOL_MANAGER_H
#define FILESTORAGE_POOL_MANAGER_H

#include <string>
#include <vector>
#include <list>
#include <set>
#include <iostream>
#include <mutex>
#include <assert.h>

#include "mtfs/Acces.h"
#include "mtfs/Pool.h"

namespace mtfs {
  class Pool;

  class Acces;

  class PoolManager : public Acces {

  public:
    static const int SUCCESS = 0;
    static const int POOL_ID_EXIST = 1;
    static const int NO_VALID_POOL = 2;
    static const int IS_LOCKED = 3;

  private:

    std::map<uint32_t, Pool *> pools;
    std::map<uint32_t, Rule *> rules;

    std::mutex inodeMutex;
```

```
57      std::set<ident_t> lockedInodes;
58      std::recursive_mutex inodeTransMutex;
59      std::map<ident_t, ident_t> inodeTranslateMap;
60
61      std::mutex dirMutex;
62      std::set<ident_t> lockedDirBlock;
63      std::recursive_mutex dirTransMutex;
64      std::map<ident_t, ident_t> dirBlockTranslateMap;
65
66      std::mutex blockMutex;
67      std::set<ident_t> lockedBlocks;
68      std::recursive_mutex blockTransMutex;
69      std::map<ident_t, ident_t> blockTranslateMap;
70
71
72    public:
73      virtual ~PoolManager();
74
75      int addPool(uint32_t poolId, Pool *pool, Rule *rule);
76
77      int add(const ruleInfo_t &info, std::vector<ident_t> &ids, const blockType
        type, const size_t nb) override;
78
79      int del(const ident_t &id, const blockType type) override;
80
81      int get(const ident_t &id, void *data, const blockType type) override;
82
83      int put(const ident_t &id, const void *data, const blockType type) override
        ;
84
85      int getMetas(const ident_t &id, blockInfo_t &metas, const blockType type)
        override;
86
87      int putMetas(const ident_t &id, const blockInfo_t &metas, const blockType
        type) override;
88
89      void doMigration(const blockType type);
90
91    private:
92
93      bool isLocked(const ident_t &id, const blockType &type);
94
95      bool lock(const ident_t &id, const blockType &type);
96
97      bool unlock(const ident_t &id, const blockType &type);
98
99      bool hasMoved(const ident_t &id, ident_t &newId, const blockType &type);
100
101     int getValidPools(const ruleInfo_t &info, std::vector<uint32_t> &poolIds);
102
103     void dumpTranslateMap(const int &nb, const blockType &type);
104
105     const int moveBlk(const ident_t &old, const ident_t &cur, const blockType &
        type);
106   };
107
108 }  // namespace mtfs
109 #endif
```

Listing 3.22 – "core/PoolManager.h"

PoolManager.cpp

```
/**
 * \file PoolManager.cpp
 * \brief
 * \author David Wittwer
 * \version 0.0.1
 * \copyright GNU Publis License V3
 *
 * This file is part of MTFS.

    MTFS is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Foobar is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <mtfs/PoolManager.h>
#include <utils/Logger.h>

namespace mtfs {
  using namespace std;

  PoolManager::~PoolManager() {
//    dumpTranslateMap(INT_MAX, INODE);
//    dumpTranslateMap(INT_MAX, DIR_BLOCK);
//    dumpTranslateMap(INT_MAX, DATA_BLOCK);
    for (auto &&pool: this->pools) {
      delete pool.second;
    }

    for (auto &&rule: this->rules) {
      delete rule.second;
    }
  }

  int PoolManager::addPool(uint32_t poolId, Pool *pool, Rule *rule) {
    if (pools.find(poolId) != pools.end())
      return POOL_ID_EXIST;

    pools[poolId] = pool;
    rules[poolId] = rule;

    return SUCCESS;
  }

```

```
53  int PoolManager::add(const ruleInfo_t &info, std::vector<ident_t> &ids, const
      blockType type,
54              const size_t nb) {
55    int ret;
56    vector<uint32_t> poolIds;
57    if (0 != (ret = this->getValidPools(info, poolIds))) {
58      return ret;
59    }
60    vector<ident_t> tmpIdent;
61
62    if (nb <= poolIds.size()) {
63      for (auto &&id: poolIds) {
64        tmpIdent.clear();
65
66        this->pools[id]->add(info, tmpIdent, type);
67
68        for (auto &&ident: tmpIdent) {
69          ident.poolId = id;
70          ids.push_back(ident);
71        }
72      }
73    } else {
74      int blkPerPool = (int) (nb / poolIds.size());
75      int remainder = (int) (nb % poolIds.size());
76      for (auto &&id: poolIds) {
77        tmpIdent.clear();
78
79        int nbToAlloc = blkPerPool;
80        if (0 < remainder) {
81          nbToAlloc++;
82          remainder--;
83        }
84
85        this->pools[id]->add(info, tmpIdent, type, nbToAlloc);
86
87        for (auto &&ident: tmpIdent) {
88          ident.poolId = id;
89          ids.push_back(ident);
90        }
91      }
92    }
93
94    return ret;
95  }
96
97  int PoolManager::del(const ident_t &id, const blockType type) {
98    int ret = IS_LOCKED;
99
100   ident_t newId = id;
101   this->hasMoved(id, newId, type);
102
103   if (!this->isLocked(newId, type))
104     ret = this->pools[newId.poolId]->del(newId.volumeId, newId.id, type);
105
106   return ret;
107 }
108
109 int PoolManager::get(const ident_t &id, void *data, const blockType type) {
110   int ret = IS_LOCKED;
```

```
111
112     ident_t newId = id;
113     this->hasMoved(id, newId, type);
114
115     if (!this->isLocked(newId, type)) {
116       string message = "get ";
117
118       switch (type) {
119         case INODE:
120           message += "inode ";
121           break;
122         case DIR_BLOCK:
123           message += "dir block ";
124           break;
125         case DATA_BLOCK:
126           message += "dat block ";
127           break;
128         case SUPERBLOCK:
129           message += "superblock ";
130           break;
131       }
132
133       message += newId.toString();
134
135       Logger::getInstance()->log("POOL_MANAGER", message, Logger::L_DEBUG);
136       ret = this->pools[newId.poolId]->get(newId.volumeId, newId.id, data, type
    );
137     }
138
139     return ret;
140   }
141
142   int PoolManager::put(const ident_t &id, const void *data, const blockType
     type) {
143     int ret = IS_LOCKED;
144
145     ident_t newId = id;
146     this->hasMoved(id, newId, type);
147
148     if (this->pools.end() == this->pools.find(newId.poolId))
149       throw out_of_range("Invalid pool id " + newId.poolId);
150
151     if (this->lock(newId, type)) {
152       ret = this->pools[newId.poolId]->put(newId.volumeId, newId.id, data, type
    );
153       this->unlock(newId, type);
154     }
155
156     return ret;
157   }
158
159   int PoolManager::getMetas(const ident_t &id, blockInfo_t &metas, const
     blockType type) {
160     int ret = IS_LOCKED;
161
162     ident_t newId = id;
163     this->hasMoved(id, newId, type);
164
165     if (!this->isLocked(newId, type))
```

```
166       ret = this->pools[newId.poolId]->getMetas(newId.volumeId, newId.id, metas
    , type);
167
168     return ret;
169   }
170
171   int PoolManager::putMetas(const ident_t &id, const blockInfo_t &metas, const
    blockType type) {
172     int ret = IS_LOCKED;
173
174     ident_t newId = id;
175     this->hasMoved(id, newId, type);
176
177     if (this->pools.end() == this->pools.find(newId.poolId))
178       throw out_of_range("Invalid pool id " + newId.poolId);
179
180     if (this->lock(newId, type)) {
181       ret = this->pools[newId.poolId]->putMetas(newId.volumeId, newId.id, metas
    , type);
182       this->unlock(newId, type);
183     }
184
185     return ret;
186   }
187
188   void PoolManager::doMigration(const blockType type) {
189     vector<ident_t> unsatisfyBlk;
190
191 //     this->dumpTranslateMap(10, type);
192
193     for (auto &&pool: this->pools) {
194       map<ident_t, ident_t> tmpMovedBlk;
195       pool.second->doMigration(tmpMovedBlk, unsatisfyBlk, type);
196       for (auto &&item :tmpMovedBlk) {
197         ident_t key = item.first;
198         ident_t val = item.second;
199
200         key.poolId = pool.first;
201         val.poolId = pool.first;
202         recursive_mutex *mu = nullptr;
203         map<ident_t, ident_t> *map = nullptr;
204         switch (type) {
205           case INODE:
206             mu = &this->inodeTransMutex;
207             map = &this->inodeTranslateMap;
208             break;
209           case DIR_BLOCK:
210             mu = &this->dirTransMutex;
211             map = &this->dirBlockTranslateMap;
212             break;
213           case DATA_BLOCK:
214             mu = &this->blockTransMutex;
215             map = &this->blockTranslateMap;
216             break;
217           case SUPERBLOCK:
218             continue;
219             break;
220         }
221         unique_lock<recursive_mutex> lk(*mu);
```

```
222         map->emplace(key, val);
223       }
224
225       Logger::getInstance()->log("Poolmanager", "endMig", Logger::level::
    L_DEBUG);
226     }
227   }
228
229   bool PoolManager::isLocked(const ident_t &id, const blockType &type) {
230     mutex *mu = nullptr;
231     set<ident_t> *lSet = nullptr;
232
233     switch (type) {
234       case INODE:
235         mu = &this->inodeMutex;
236         lSet = &this->lockedInodes;
237         break;
238       case DIR_BLOCK:
239         mu = &this->dirMutex;
240         lSet = &this->lockedDirBlock;
241         break;
242       case DATA_BLOCK:
243         mu = &this->blockMutex;
244         lSet = &this->lockedBlocks;
245         break;
246     }
247
248     if (nullptr == mu || nullptr == lSet)
249       return false;
250
251     unique_lock<mutex> lk(*mu);
252     return lSet->find(id) != lSet->end();
253   }
254
255   bool PoolManager::lock(const ident_t &id, const blockType &type) {
256     mutex *mu = nullptr;
257     set<ident_t> *lSet = nullptr;
258
259     switch (type) {
260       case INODE:
261         mu = &this->inodeMutex;
262         lSet = &this->lockedInodes;
263         break;
264       case DIR_BLOCK:
265         mu = &this->dirMutex;
266         lSet = &this->lockedDirBlock;
267         break;
268       case DATA_BLOCK:
269         mu = &this->blockMutex;
270         lSet = &this->lockedBlocks;
271         break;
272       default:
273 //        TODO log error
274         return false;
275         break;
276     }
277
278     unique_lock<mutex> lk(*mu);
279     if (lSet->find(id) != lSet->end())
```

```
280       return false;
281
282     lSet->insert(id);
283
284     return true;
285   }
286
287   bool PoolManager::unlock(const ident_t &id, const blockType &type) {
288     mutex *mu = nullptr;
289     set<ident_t> *lSet = nullptr;
290
291     switch (type) {
292       case INODE:
293         mu = &this->inodeMutex;
294         lSet = &this->lockedInodes;
295         break;
296       case DIR_BLOCK:
297         mu = &this->dirMutex;
298         lSet = &this->lockedDirBlock;
299         break;
300       case DATA_BLOCK:
301         mu = &this->blockMutex;
302         lSet = &this->lockedBlocks;
303         break;
304       default:
305 //        TODO log error
306         return false;
307         break;
308     }
309
310     unique_lock<mutex> lk(*mu);
311     if (lSet->find(id) == lSet->end())
312       return false;
313
314     lSet->erase(id);
315
316     return true;
317   }
318
319   bool PoolManager::hasMoved(const ident_t &id, ident_t &newId, const blockType
       &type) {
320     recursive_mutex *mu = nullptr;
321     map<ident_t, ident_t> *transMap = nullptr;
322
323     switch (type) {
324       case INODE:
325         mu = &this->inodeTransMutex;
326         transMap = &this->inodeTranslateMap;
327         break;
328       case DIR_BLOCK:
329         mu = &this->dirTransMutex;
330         transMap = &this->dirBlockTranslateMap;
331         break;
332       case DATA_BLOCK:
333         mu = &this->blockTransMutex;
334         transMap = &this->blockTranslateMap;
335         break;
336       default:
337         return false;
```

```
338     }
339
340     assert(mu != nullptr);
341     assert(transMap != nullptr);
342
343     unique_lock<recursive_mutex> lk(*mu);
344     if (transMap->find(id) != transMap->end()) {
345       newId = (*transMap)[id];
346       return true;
347     }
348
349     return false;
350   }
351
352   int PoolManager::getValidPools(const ruleInfo_t &info, std::vector<uint32_t>
       &poolIds) {
353     for (auto &&item: this->rules) {
354       if (item.second->satisfyRules(info))
355         poolIds.push_back(item.first);
356     }
357
358     return SUCCESS;
359   }
360
361   void PoolManager::dumpTranslateMap(const int &nb, const blockType &type) {
362
363     recursive_mutex *mu;
364     map<ident_t, ident_t> *tm;
365     switch (type) {
366       case INODE:
367         mu = &this->inodeTransMutex;
368         tm = &this->inodeTranslateMap;
369         break;
370       case DIR_BLOCK:
371         mu = &this->dirTransMutex;
372         tm = &this->dirBlockTranslateMap;
373         break;
374       case DATA_BLOCK:
375         mu = &this->blockTransMutex;
376         tm = &this->blockTranslateMap;
377         break;
378       default:
379         return;
380     }
381
382     unique_lock<recursive_mutex> lk(*mu);
383     int i = 0;
384     for (auto &&item :*tm) {
385       if (nb == i)
386         break;
387       if (SUCCESS == this->moveBlk(item.first, item.second, type)) {
388         tm->erase(item.first);
389       }
390       i++;
391     }
392   }
393
394   const int PoolManager::moveBlk(const ident_t &old, const ident_t &cur, const
       blockType &type) {
```

```
395     blockInfo_t metas{};
396     this->pools[cur.poolId]->getMetas(cur.volumeId, cur.id, metas, type);
397     dirBlock_t dirBlock{};
398 
399     switch (type) {
400       case INODE:
401         this->get(metas.referenceId.front(), &dirBlock, DIR_BLOCK);
402         for (auto &&item :dirBlock.entries) {
403           for (auto &i : item.second) {
404             if (old == i)
405               i = cur;
406           }
407         }
408         for (auto &&ref :metas.referenceId) {
409           this->put(ref, &dirBlock, DIR_BLOCK);
410         }
411         break;
412       case DIR_BLOCK:
413       case DATA_BLOCK:
414         inode_t inode{};
415         this->get(metas.referenceId.front(), &inode, INODE);
416 
417         for (auto &&blks :inode.dataBlocks) {
418           for (auto &blk :blks) {
419             if (old == blk)
420               blk = cur;
421           }
422         }
423         for (auto &&ref :metas.referenceId) {
424           this->put(ref, &inode, INODE);
425         }
426         break;
427     }
428 
429     return SUCCESS;
430   }
431 
432 }  // namespace mtfs
```

Listing 3.23 – "core/PoolManager.cpp"

3.4.6 Rule

Rule.h

```
1  /**
2   * \file Rule.h
3   * \brief
4   * \author David Wittwer
5   * \version 0.0.1
6   * \copyright GNU Publis License V3
7   *
8   * This file is part of MTFS.
9   *
10    MTFS is free software: you can redistribute it and/or modify
11    it under the terms of the GNU General Public License as published by
12    the Free Software Foundation, either version 3 of the License, or
13    (at your option) any later version.
```

```
14
15      Foobar is distributed in the hope that it will be useful,
16      but WITHOUT ANY WARRANTY; without even the implied warranty of
17      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18      GNU General Public License for more details.
19
20      You should have received a copy of the GNU General Public License
21      along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22  */
23
24 #ifndef FILESTORAGE_RULE_H
25 #define FILESTORAGE_RULE_H
26
27 #include <string>
28 #include <vector>
29 #include <list>
30 #include <iostream>
31 #include <assert.h>
32 #include <rapidjson/schema.h>
33 #include <mtfs/structs.h>
34 #include <mtfs/Volume.h>
35
36 /**
37  *
38  */
39 namespace mtfs {
40   class Pool;
41
42   class Volume;
43
44   /**
45    * @interface
46    */
47   class Rule {
48   public:
49     static constexpr const char *MIGRATION = "migration";     ///< Migration
      name in the config file
50
51     static const int NO_MIGRATION = -1;     ///< No migration value
52     static const int TIME_MIGRATION = 0;     ///< Time migration value
53     static const int RIGHT_MIGRATION = 1;     ///< User right migration
54
55     static const int SUCCESS = 0;     ///< Success code.
56     static const int VALID_RULES = 0;     ///< Valid rule code.
57     static const int INVALID_RULES = -1;     ///< Invalid rule code.
58     static const int UNKNOW_MIGRATION = -2;     ///< Unknow migration code.
59   public:
60
61     /**
62      * @brief Copy JSON config.
63      *
64      * @param migration   Migration type.
65      * @param source    JSON source config.
66      * @param destination   JSON destination config.
67      * @param allocator   Allocator for destination.
68      *
69      * @return SUCCESS if no error or corresponding code.
70      */
```

```
71      static int copyConfig(int migration, rapidjson::Document &source, rapidjson
        ::Value &destination,
72                  rapidjson::Document::AllocatorType &allocator);
73
74      /**
75       * @brief Check if rule config is valid.
76       *
77       * @param migration Migration type.
78       * @param value Json config.
79       * @return VALID_RULES or INVALID_RULES
80       */
81      static int rulesAreValid(int migration, const rapidjson::Value &value);
82
83      /**
84       * @brief Build a Rule object.
85       *
86       * @param migration Migration type.
87       * @param value Json Config.
88       *
89       * @return Pointer on new Rule or nullptr if fail.
90       */
91      static Rule *buildRule(int migration, const rapidjson::Value &value);
92
93      /**
94       * Dump object to Json.
95       *
96       * @param json
97       * @param allocator
98       * @return
99       */
100     virtual bool toJson(rapidjson::Value &json, rapidjson::Document::
        AllocatorType &allocator)=0;
101
102     /**
103      * @brief Check if the block or inode satisfy rules.
104      * @param info Info of Block/Inode
105      *
106      * @return true or false.
107      */
108     virtual bool satisfyRules(mtfs::ruleInfo_t info)=0;
109
110     /**
111      * Configure volume storage
112      * @param volume
113      * @return
114      */
115     virtual int configureStorage(Volume *volume);
116
117     /**
118      * Configure pool storage
119      * @param volume
120      * @return
121      */
122     virtual int configureStorage(Pool *pool);
123   };
124
125 }  // namespace mtfs
126 #endif
```

Listing 3.24 – "core/Rule.h"

Rule.cpp

```
/**
 * \file Rule.cpp
 * \brief
 * \author David Wittwer
 * \version 0.0.1
 * \copyright GNU Publis License V3
 *
 * This file is part of MTFS.

    MTFS is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Foobar is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <mtfs/TimeRule.h>
#include <mtfs/UserRightRule.h>
#include <pwd.h>

namespace mtfs {

  int Rule::copyConfig(int migration, rapidjson::Document &source, rapidjson::
    Value &destination,
              rapidjson::Document::AllocatorType &allocator) {
    switch (migration) {
      case TIME_MIGRATION:
        return TimeRule::copyConfig(source, destination, allocator);
      case RIGHT_MIGRATION:
        return UserRightRule::copyConfig(source, destination, allocator);
      default:
        return UNKNOW_MIGRATION;
    }
  }

  int Rule::rulesAreValid(int migration, const rapidjson::Value &value) {
    switch (migration) {
      case NO_MIGRATION:
        return SUCCESS;
      case TIME_MIGRATION:
        return TimeRule::rulesAreValid(value);
      case RIGHT_MIGRATION:
        return UserRightRule::rulesAreValid(value);
      default:
        return UNKNOW_MIGRATION;
    }
```

```
53   }
54
55   Rule *Rule::buildRule(int migration, const rapidjson::Value &value) {
56     Rule *rule = nullptr;
57     if (migration == NO_MIGRATION)
58       rule = nullptr;
59     else if (migration == TIME_MIGRATION) {
60       uint64_t ll = 0, hl = 0;
61
62       if (value.HasMember(TimeRule::TIME_LOW_LIMIT))
63         ll = value[TimeRule::TIME_LOW_LIMIT].GetUint64();
64
65       if (value.HasMember(TimeRule::TIME_HIGH_LIMIT))
66         hl = value[TimeRule::TIME_HIGH_LIMIT].GetUint64();
67
68       rule = new TimeRule(ll, hl);
69     } else if (migration == RIGHT_MIGRATION) {
70       UserRightRule *uRule = new UserRightRule();
71
72       if (value.HasMember(UserRightRule::ALLOW_USER))
73         for (auto &ua: value[UserRightRule::ALLOW_USER].GetArray())
74           uRule->addAllowUid(getpwnam(ua.GetString())->pw_uid);
75
76       if (value.HasMember(UserRightRule::DENY_USER))
77         for (auto &ud: value[UserRightRule::DENY_USER].GetArray())
78           uRule->addDenyUid(getpwnam(ud.GetString())->pw_uid);
79
80       if (value.HasMember(UserRightRule::ALLOW_GROUP))
81         for (auto &ga: value[UserRightRule::ALLOW_GROUP].GetArray())
82           uRule->addAllowGid(getpwnam(ga.GetString())->pw_gid);
83
84       if (value.HasMember(UserRightRule::DENY_GROUP))
85         for (auto &ga: value[UserRightRule::DENY_GROUP].GetArray())
86           uRule->addDenyGid(getpwnam(ga.GetString())->pw_gid);
87
88       rule = uRule;
89     }
90     return rule;
91   }
92
93   int Rule::configureStorage(Volume *volume) {
94     volume->setIsTimeVolume(false);
95
96     return ENOSYS;
97   }
98
99   int Rule::configureStorage(Pool *pool) {
100    (void) pool;
101
102    return ENOSYS;
103  }
104
105 }
```

Listing 3.25 – "core/Rule.cpp"

3.4.7 TimeRule

TimeRule.h

```
/**
 * \file TimeRule.h
 * \brief
 * \author David Wittwer
 * \version 0.0.1
 * \copyright GNU Publis License V3
 *
 * This file is part of MTFS.

    MTFS is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Foobar is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
 */

#ifndef FILESTORAGE_TIME_RULE_H
#define FILESTORAGE_TIME_RULE_H

#include <string>
#include <vector>
#include <list>
#include <iostream>
#include <assert.h>

#include <mtfs/Rule.h>

namespace mtfs {
  class TimeRule : public Rule {
  public:
    static constexpr const char *TIME_LOW_LIMIT = "lowLimit";
    static constexpr const char *TIME_HIGH_LIMIT = "highLimit";

  private:
    uint64_t lowerLimit;
    uint64_t higerLimit;

  public:
    static int copyConfig(rapidjson::Document &source, rapidjson::Value &
    destination,
                rapidjson::Document::AllocatorType &allocator);

    static int rulesAreValid(const rapidjson::Value &value);

    TimeRule(uint64_t lowerLimit, uint64_t higerLimit);

    bool satisfyRules(ruleInfo_st info) override;

```

```
55      bool toJson(rapidjson::Value &json, rapidjson::Document::AllocatorType &
        allocator) override;
56
57      int configureStorage(Volume *volume) override;
58
59      int configureStorage(Pool *pool) override;
60
61    private:
62      uint64_t now();
63    };
64
65  }  // namespace mtfs
66  #endif
```

Listing 3.26 – "core/TimeRule.h"

TimeRule.cpp

```
1   /**
2    * \file TimeRule.cpp
3    * \brief
4    * \author David Wittwer
5    * \version 0.0.1
6    * \copyright GNU Publis License V3
7    *
8    * This file is part of MTFS.
9
10     MTFS is free software: you can redistribute it and/or modify
11     it under the terms of the GNU General Public License as published by
12     the Free Software Foundation, either version 3 of the License, or
13     (at your option) any later version.
14
15     Foobar is distributed in the hope that it will be useful,
16     but WITHOUT ANY WARRANTY; without even the implied warranty of
17     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18     GNU General Public License for more details.
19
20     You should have received a copy of the GNU General Public License
21     along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22   */
23
24  #include "mtfs/TimeRule.h"
25
26  namespace mtfs {
27    TimeRule::TimeRule(uint64_t lowerLimit, uint64_t higerLimit) : lowerLimit(
       lowerLimit),
28                                                    higerLimit(higerLimit) {}
29
30    bool TimeRule::satisfyRules(ruleInfo_st info) {
31      uint64_t now = this->now();
32      uint64_t delay = now - info.lastAccess;
33
34      if (0 == this->lowerLimit && 0 == this->higerLimit)
35        return true;
36      else if (0 == this->lowerLimit)
37        return delay <= this->higerLimit;
38      else if (0 == this->higerLimit)
39        return delay >= this->lowerLimit;
40      else
```

```
41      return this->lowerLimit <= delay <= this->higerLimit;
42  }
43
44  int TimeRule::copyConfig(rapidjson::Document &source, rapidjson::Value &
    destination,
45              rapidjson::Document::AllocatorType &allocator) {
46
47    if (source.HasMember(TIME_LOW_LIMIT))
48      destination.AddMember(rapidjson::StringRef(TIME_LOW_LIMIT), source[
    TIME_LOW_LIMIT], allocator);
49
50    if (source.HasMember(TIME_HIGH_LIMIT))
51      destination.AddMember(rapidjson::StringRef(TIME_HIGH_LIMIT), source[
    TIME_HIGH_LIMIT], allocator);
52
53    return SUCCESS;
54  }
55
56  int TimeRule::rulesAreValid(const rapidjson::Value &value) {
57    if (value.HasMember(TIME_LOW_LIMIT) || value.HasMember(TIME_HIGH_LIMIT)) {
58      return VALID_RULES;
59    } else {
60      return INVALID_RULES;
61    }
62  }
63
64  bool TimeRule::toJson(rapidjson::Value &json, rapidjson::Document::
    AllocatorType &allocator) {
65
66    rapidjson::Value v;
67
68    if (this->lowerLimit != 0) {
69      v.SetUint64(this->lowerLimit);
70      json.AddMember(rapidjson::StringRef(TIME_LOW_LIMIT), v, allocator);
71    }
72
73    if (this->higerLimit != 0) {
74      v.SetUint64(this->higerLimit);
75      json.AddMember(rapidjson::StringRef(TIME_HIGH_LIMIT), v, allocator);
76    }
77
78    return true;
79  }
80
81  uint64_t TimeRule::now() {
82    return (uint64_t) time(NULL);
83  }
84
85  int TimeRule::configureStorage(Volume *volume) {
86    volume->setMinDelay(this->lowerLimit);
87    volume->setMaxDelay(this->higerLimit);
88    volume->setIsTimeVolume(true);
89
90    return 0;
91  }
92
93  int TimeRule::configureStorage(Pool *pool) {
94    return 0;
95  }
```

```
96
97 }  // namespace mtfs
```

Listing 3.27 – "core/TimeRule.cpp"

3.4.8 UserRightRule

UserRightRule.h

```
1  /**
2   * \file UserRightRule.h
3   * \brief
4   * \author David Wittwer
5   * \version 0.0.1
6   * \copyright GNU Publis License V3
7   *
8   * This file is part of MTFS.
9
10     MTFS is free software: you can redistribute it and/or modify
11     it under the terms of the GNU General Public License as published by
12     the Free Software Foundation, either version 3 of the License, or
13     (at your option) any later version.
14
15     Foobar is distributed in the hope that it will be useful,
16     but WITHOUT ANY WARRANTY; without even the implied warranty of
17     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18     GNU General Public License for more details.
19
20     You should have received a copy of the GNU General Public License
21     along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22  */
23
24 #ifndef FILESTORAGE_USER_GROUP_RULE_H
25 #define FILESTORAGE_USER_GROUP_RULE_H
26
27 #include <string>
28 #include <vector>
29 #include <list>
30 #include <iostream>
31 #include <assert.h>
32
33 #include "mtfs/Rule.h"
34
35 namespace mtfs {
36   class UserRightRule : public Rule {
37   public:
38     static constexpr const char *ALLOW_USER = "allowUsers";
39     static constexpr const char *DENY_USER = "denyUsers";
40     static constexpr const char *ALLOW_GROUP = "allowGroups";
41     static constexpr const char *DENY_GROUP = "denyGroups";
42
43   private:
44     std::vector<uid_t> uidAllowed;
45
46     std::vector<uid_t> uidDenied;
47
48     std::vector<gid_t> gidAllowed;
49
```

```
50      std::vector<gid_t> gidDenied;
51
52    public:
53      static int copyConfig(rapidjson::Document &source, rapidjson::Value &
      destination,
54                  rapidjson::Document::AllocatorType &allocator);
55
56      static int rulesAreValid(const rapidjson::Value &value);
57
58      void addAllowUid(uid_t uid);
59
60      void addDenyUid(uid_t uid);
61
62      void addAllowGid(gid_t gid);
63
64      void addDenyGid(gid_t gid);
65
66      bool toJson(rapidjson::Value &json, rapidjson::Document::AllocatorType &
      allocator) override;
67
68      bool satisfyRules(ruleInfo_st info) override;
69
70    };
71
72  }  // namespace mtfs
73  #endif
```

Listing 3.28 – "core/UserRightRule.h"

UserRightRule.cpp

```
1   /**
2    * \file UserRightRule.cpp
3    * \brief
4    * \author David Wittwer
5    * \version 0.0.1
6    * \copyright GNU Publis License V3
7    *
8    * This file is part of MTFS.
9
10     MTFS is free software: you can redistribute it and/or modify
11     it under the terms of the GNU General Public License as published by
12     the Free Software Foundation, either version 3 of the License, or
13     (at your option) any later version.
14
15     Foobar is distributed in the hope that it will be useful,
16     but WITHOUT ANY WARRANTY; without even the implied warranty of
17     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18     GNU General Public License for more details.
19
20     You should have received a copy of the GNU General Public License
21     along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22   */
23
24  #include <pwd.h>
25  #include "mtfs/UserRightRule.h"
26
27  namespace mtfs {
```

```
28  int UserRightRule::copyConfig(rapidjson::Document &source, rapidjson::Value &
      destination,
29                    rapidjson::Document::AllocatorType &allocator) {
30  //    TODO copy config
31      if (source.HasMember(ALLOW_USER))
32        destination.AddMember(rapidjson::StringRef(ALLOW_USER), source[ALLOW_USER
      ], allocator);
33
34      if (source.HasMember(DENY_USER))
35        destination.AddMember(rapidjson::StringRef(DENY_USER), source[DENY_USER],
       allocator);
36
37      if (source.HasMember(ALLOW_GROUP))
38        destination.AddMember(rapidjson::StringRef(ALLOW_GROUP), source[
      ALLOW_GROUP], allocator);
39
40      if (source.HasMember(DENY_GROUP))
41        destination.AddMember(rapidjson::StringRef(DENY_GROUP), source[DENY_GROUP
      ], allocator);
42
43      return SUCCESS;
44    }
45
46    int UserRightRule::rulesAreValid(const rapidjson::Value &value) {
47      if (value.HasMember(ALLOW_USER) || value.HasMember(ALLOW_GROUP) ||
48          value.HasMember(DENY_USER) || value.HasMember(DENY_GROUP)) {
49        return VALID_RULES;
50      } else {
51        return INVALID_RULES;
52      }
53    }
54
55    void UserRightRule::addAllowUid(uid_t uid) {
56      uidAllowed.push_back(uid);
57    }
58
59    void UserRightRule::addDenyUid(uid_t uid) {
60      uidDenied.push_back(uid);
61    }
62
63    void UserRightRule::addAllowGid(gid_t gid) {
64      gidAllowed.push_back(gid);
65    }
66
67    void UserRightRule::addDenyGid(gid_t gid) {
68      gidDenied.push_back(gid);
69    }
70
71    bool UserRightRule::satisfyRules(ruleInfo_st info) {
72      return true;
73    }
74
75    bool UserRightRule::toJson(rapidjson::Value &json, rapidjson::Document::
      AllocatorType &allocator) {
76
77      rapidjson::Value v;
78      rapidjson::Value a(rapidjson::kArrayType);
79
80      struct passwd *pw;
```

```
81      char buffer[50];
82      memset(buffer, 0, 50 * sizeof(char));
83
84      if (this->uidAllowed.size() != 0) {
85        for (auto &&allowed : this->uidAllowed) {
86          pw = getpwuid(allowed);
87          int len = sprintf(buffer, "%s", pw->pw_name);
88          v.SetString(buffer, (rapidjson::SizeType) len, allocator);
89          a.PushBack(v, allocator);
90        }
91        json.AddMember(rapidjson::StringRef(ALLOW_USER), a, allocator);
92      }
93
94      if (this->uidDenied.size() != 0) {
95        for (auto &&denied : this->uidDenied) {
96          pw = getpwuid(denied);
97          int len = sprintf(buffer, "%s", pw->pw_name);
98          v.SetString(buffer, (rapidjson::SizeType) len, allocator);
99          a.PushBack(v, allocator);
100       }
101       json.AddMember(rapidjson::StringRef(DENY_USER), a, allocator);
102     }
103
104     if (this->gidAllowed.size() != 0) {
105       for (auto &&allowed : this->gidAllowed) {
106         pw = getpwuid(allowed);
107         int len = sprintf(buffer, "%s", pw->pw_name);
108         v.SetString(buffer, (rapidjson::SizeType) len, allocator);
109         a.PushBack(v, allocator);
110       }
111       json.AddMember(rapidjson::StringRef(ALLOW_GROUP), a, allocator);
112     }
113
114     if (this->gidDenied.size() != 0) {
115       for (auto &&denied : this->gidDenied) {
116         pw = getpwuid(denied);
117         int len = sprintf(buffer, "%s", pw->pw_name);
118         v.SetString(buffer, (rapidjson::SizeType) len, allocator);
119         a.PushBack(v, allocator);
120       }
121       json.AddMember(rapidjson::StringRef(DENY_GROUP), a, allocator);
122     }
123
124     return true;
125   }
126
127 }  // namespace mtfs
```

Listing 3.29 – "core/UserRightRule.cpp"

3.4.9 Volume

Volume.h

```
1 /**
2  * \file Volume.h
3  * \brief
4  * \author David Wittwer
```

```
 * \version 0.0.1
 * \copyright GNU Publis License V3
 *
 * This file is part of MTFS.

    MTFS is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Foobar is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
 */

#ifndef FILESTORAGE_VOLUME_H
#define FILESTORAGE_VOLUME_H

#include <string>
#include <vector>
#include <climits>

#include <rapidjson/document.h>
#include <mtfs/structs.h>
#include <pluginSystem/Plugin.h>
#include <mutex>

namespace mtfs {
  class Volume {

  public:
    static constexpr const char *VOLUMES = "volumes";

  private:
    static const uint64_t NEW_DELAY = 50;

    pluginSystem::Plugin *plugin;

    bool isTimeVolume;
    uint64_t minDelay;
    uint64_t maxDelay;

    std::mutex iaMutex;
    std::map<uint64_t, uint64_t> inodesAccess;
    std::mutex daMutex;
    std::map<uint64_t, uint64_t> dirBlockAccess;
    std::mutex baMutex;
    std::map<uint64_t, uint64_t> blocksAccess;

    std::mutex niMutex;
    std::map<uint64_t, uint64_t> newInode;
    std::mutex ndMutex;
    std::map<uint64_t, uint64_t> newDir;
    std::mutex nbMutex;
    std::map<uint64_t, uint64_t> newData;
```

```
64
65    public:
66      virtual ~Volume();
67
68      void setMinDelay(uint64_t minDelay);
69
70      void setMaxDelay(uint64_t maxDelay);
71
72      void setIsTimeVolume(bool b);
73
74      static bool validate(const rapidjson::Value &volume);
75
76      static void
77      structToJson(const volume_t &volume, rapidjson::Value &dest, rapidjson::
     Document::AllocatorType &allocator);
78
79      static void jsonToStruct(rapidjson::Value &src, volume_t &volume);
80
81      explicit Volume(pluginSystem::Plugin *plugin);
82
83      int add(uint64_t &id, const blockType &type);
84
85      int add(std::vector<uint64_t> &ids, const int &nb, const blockType &type);
86
87      int del(const uint64_t &id, const blockType &type);
88
89      int get(const uint64_t &id, void *data, const blockType &type);
90
91      int put(const uint64_t &id, const void *data, const blockType &type);
92
93      int getMetas(const uint64_t &id, blockInfo_t &metas, const blockType &type)
     ;
94
95      int putMetas(const uint64_t &id, const blockInfo_t &metas, const blockType
     &type);
96
97      int getUnsatisfy(std::vector<blockInfo_t> &unsatisfy, const blockType &type
     , int limit = INT_MAX);
98
99    private:
100     int getOutOfTime(std::vector<uint64_t> &blocks, const blockType &type);
101
102     bool updateLastAccess(const uint64_t &id, const blockType &type);
103
104     void purgeNewMap(const blockType &type);
105   };
106
107 }  // namespace mtfs
108 #endif
```

Listing 3.30 – "core/Volume.h"

Volume.cpp

```
1 /**
2  * \file Volume.cpp
3  * \brief
4  * \author David Wittwer
5  * \version 0.0.1
```

```
 * \copyright GNU Publis License V3
 *
 * This file is part of MTFS.

    MTFS is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Foobar is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <pluginSystem/PluginManager.h>
#include <thread>
#include <boost/thread.hpp>
#include <boost/threadpool/pool.hpp>
#include "mtfs/Volume.h"
#include <mtfs/Rule.h>

using namespace std;

namespace mtfs {

  Volume::Volume(pluginSystem::Plugin *plugin) : plugin(plugin), minDelay(0),
   maxDelay(0) {}

  Volume::~Volume() {
    plugin->detach();

    pluginSystem::PluginManager::getInstance()->freePlugin(this->plugin->
   getName(), this->plugin);
  }

  bool Volume::validate(const rapidjson::Value &volume) {
    if (!volume.HasMember(pluginSystem::Plugin::TYPE))
      return false;

    if (!volume.HasMember(pluginSystem::Plugin::PARAMS))
      return false;

    string volType = volume[pluginSystem::Plugin::TYPE].GetString();

    pluginSystem::PluginManager *manager = pluginSystem::PluginManager::
   getInstance();

    pluginSystem::Plugin *plugin = manager->getPlugin(volType);
    if (nullptr == plugin)
      return false;

    pluginSystem::pluginInfo_t info;

    if (manager->getInfo(volType, info) != pluginSystem::PluginManager::SUCCESS
   )
```

```
61        return false;
62
63      for (auto &&inf : info.params) {
64        if (!volume[pluginSystem::Plugin::PARAMS].HasMember(inf.c_str()))
65          return false;
66      }
67
68  //    vector<string> infos = plugin->getInfos();
69  //    infos.erase(infos.begin());
70  //    for (auto i:infos)
71  //      if (!volume.HasMember(i.c_str()))
72  //        return false;
73
74      return true;
75    }
76
77    void Volume::structToJson(const volume_t &volume, rapidjson::Value &dest,
78                  rapidjson::Document::AllocatorType &allocator) {
79      volume.rule->toJson(dest, allocator);
80
81      rapidjson::Value v;
82
83      v.SetString(rapidjson::StringRef(volume.pluginName.c_str()));
84      dest.AddMember(rapidjson::StringRef(pluginSystem::Plugin::TYPE), v,
       allocator);
85
86      rapidjson::Value p(rapidjson::kObjectType);
87      for (auto &&item : volume.params) {
88        v.SetString(rapidjson::StringRef(item.second.c_str()));
89        p.AddMember(rapidjson::StringRef(item.first.c_str()), v, allocator);
90      }
91      dest.AddMember(rapidjson::StringRef(pluginSystem::Plugin::PARAMS), p,
       allocator);
92    }
93
94    void Volume::jsonToStruct(rapidjson::Value &src, volume_t &volume) {
95      assert(src.HasMember(pluginSystem::Plugin::TYPE));
96      volume.pluginName = src[pluginSystem::Plugin::TYPE].GetString();
97
98      assert(src.HasMember(pluginSystem::Plugin::PARAMS));
99      for (auto &&item : src[pluginSystem::Plugin::PARAMS].GetObject()) {
100       volume.params.insert(make_pair(item.name.GetString(), item.value.
       GetString()));
101     }
102   }
103
104   int Volume::add(uint64_t &id, const blockType &type) {
105     int ret;
106     ret = this->plugin->add(&id, type);
107
108     if (ENOSYS == ret) {
109 //      TODO log noimplemented
110     }
111     mutex *mu = nullptr;
112     map<uint64_t, uint64_t> *ma = nullptr;
113
114     switch (type) {
115       case INODE:
116         mu = &this->niMutex;
```

```
117         ma = &this->newInode;
118         break;
119       case DIR_BLOCK:
120         mu = &this->ndMutex;
121         ma = &this->newData;
122         break;
123       case DATA_BLOCK:
124         mu = &this->nbMutex;
125         ma = &this->newData;
126         break;
127       default:
128         return ENOSYS;
129     }
130
131     //      add id in newBlock map
132     unique_lock<mutex> lk(*mu);
133     ma->emplace(id, time(nullptr));
134     lk.unlock();
135
136     return ret;
137   }
138
139   int Volume::add(std::vector<uint64_t> &ids, const int &nb, const blockType &
      type) {
140     boost::threadpool::pool thPool((size_t) nb);
141
142     vector<uint64_t *> tmp;
143     for (int i = 0; i < nb; ++i) {
144       uint64_t *id;
145       id = new uint64_t();
146       tmp.push_back(id);
147       thPool.schedule(bind(&pluginSystem::Plugin::add, this->plugin, id, type))
      ;
148     }
149
150     thPool.wait();
151
152     mutex *mu = nullptr;
153     map<uint64_t, uint64_t> *ma = nullptr;
154
155     switch (type) {
156       case INODE:
157         mu = &this->niMutex;
158         ma = &this->newInode;
159         break;
160       case DIR_BLOCK:
161         mu = &this->ndMutex;
162         ma = &this->newData;
163         break;
164       case DATA_BLOCK:
165         mu = &this->nbMutex;
166         ma = &this->newData;
167         break;
168       default:
169         return ENOSYS;
170     }
171
172     for (auto &&item: tmp) {
173 //      add id in newBlock map
```

```
174       unique_lock<mutex> lk(*mu);
175       ma->emplace(*item, time(nullptr));
176       lk.unlock();
177
178       ids.push_back(*item);
179       delete item;
180     }
181
182     return 0;
183   }
184
185   int Volume::del(const uint64_t &id, const blockType &type) {
186     int ret;
187     ret = this->plugin->del(id, type);
188
189     if (ENOSYS == ret) {
190 //        TODO log noimplemented
191     }
192
193     map<uint64_t, uint64_t> *access = nullptr;
194     mutex *mu;
195     switch (type) {
196       case INODE:
197         mu = &this->iaMutex;
198         access = &this->inodesAccess;
199         break;
200       case DIR_BLOCK:
201         mu = &this->daMutex;
202         access = &this->dirBlockAccess;
203         break;
204       case DATA_BLOCK:
205         mu = &this->baMutex;
206         access = &this->blocksAccess;
207         break;
208       default:
209         return ENOSYS;
210     }
211
212     unique_lock<mutex> lk(*mu);
213     access->erase(id);
214     lk.unlock();
215
216     return ret;
217   }
218
219   int Volume::get(const uint64_t &id, void *data, const blockType &type) {
220     int ret;
221     ret = this->plugin->get(id, data, type, false);
222
223     if (ENOSYS == ret) {
224 //        TODO log noimplemented
225     }
226
227     this->updateLastAccess(id, type);
228
229     return ret;
230   }
231
```

```
232   int Volume::put(const uint64_t &id, const void *data, const blockType &type)
      {
233     int ret;
234     ret = this->plugin->put(id, data, type, false);
235
236     if (ENOSYS == ret) {
237 //        TODO log noimplemented
238     }
239
240     this->updateLastAccess(id, type);
241
242     return ret;
243   }
244
245   int Volume::getMetas(const uint64_t &id, blockInfo_t &metas, const blockType
      &type) {
246     int ret;
247     if (EXIT_SUCCESS != (ret = this->plugin->get(id, &metas, type, true))) {
248       return ret;
249     }
250
251     map<uint64_t, uint64_t> *access = nullptr;
252     mutex *mu;
253     switch (type) {
254       case INODE:
255         mu = &this->iaMutex;
256         access = &this->inodesAccess;
257         break;
258       case DIR_BLOCK:
259         mu = &this->daMutex;
260         access = &this->dirBlockAccess;
261         break;
262       case DATA_BLOCK:
263         mu = &this->baMutex;
264         access = &this->blocksAccess;
265         break;
266       default:
267         return ENOSYS;
268     }
269
270     unique_lock<mutex> lk(*mu);
271     if (access->end() == access->find(id))
272       (*access)[id] = metas.lastAccess;
273
274     return ret;
275   }
276
277   int Volume::putMetas(const uint64_t &id, const blockInfo_t &metas, const
      blockType &type) {
278     if (0 == id && INODE == type)
279       return EXIT_SUCCESS;
280
281     int ret;
282     if (EXIT_SUCCESS != (ret = this->plugin->put(id, &metas, type, true))) {
283       return ret;
284     }
285
286     map<uint64_t, uint64_t> *access = nullptr;
287     mutex *mu;
```

```
288     switch (type) {
289       case INODE:
290         mu = &this->iaMutex;
291         access = &this->inodesAccess;
292         break;
293       case DIR_BLOCK:
294         mu = &this->daMutex;
295         access = &this->dirBlockAccess;
296         break;
297       case DATA_BLOCK:
298         mu = &this->baMutex;
299         access = &this->blocksAccess;
300         break;
301       default:
302         return ENOSYS;
303     }
304
305     unique_lock<mutex> lk(*mu);
306     (*access)[id] = metas.lastAccess;
307
308     return ret;
309   }
310
311   bool Volume::updateLastAccess(const uint64_t &id, const blockType &type) {
312     blockInfo_t metas = blockInfo_t();
313     this->getMetas(id, metas, type);
314
315     metas.lastAccess = static_cast<uint64_t>(time(nullptr));
316     this->putMetas(id, metas, type);
317
318     return true;
319   }
320
321   void Volume::setMinDelay(uint64_t minDelay) {
322     this->minDelay = minDelay;
323   }
324
325   void Volume::setMaxDelay(uint64_t maxDelay) {
326     this->maxDelay = maxDelay;
327   }
328
329   void Volume::setIsTimeVolume(bool b) {
330     this->isTimeVolume = b;
331   }
332
333   int Volume::getUnsatisfy(vector<blockInfo_t> &unsatisfy, const blockType &
       type, const int limit) {
334     this->purgeNewMap(type);
335
336     int nb = 0;
337     if (this->isTimeVolume) {
338       vector<uint64_t> under;
339       this->getOutOfTime(under, type);
340
341       for (auto &&blk :under) {
342         blockInfo_t info = blockInfo_t();
343         this->getMetas(blk, info, type);
344         info.id.id = blk;
345
```

```
346         unsatisfy.push_back(info);
347
348         nb++;
349         if (limit == nb)
350           return 0;
351       }
352     }
353
354     return 0;
355   }
356
357   int Volume::getOutOfTime(vector<uint64_t> &blocks, const blockType &type) {
358     uint64_t now;
359     now = (uint64_t) time(nullptr);
360     const uint64_t maxTimestamp = now - this->minDelay;
361     const uint64_t minTimestamp = 0 == this->maxDelay ? 0 : now - this->
      maxDelay;
362
363     mutex *mu;
364     map<uint64_t, uint64_t> *mp;
365     mutex *newMut;
366     map<uint64_t, uint64_t> *newMap;
367     switch (type) {
368       case INODE:
369         mu = &this->iaMutex;
370         mp = &this->inodesAccess;
371         newMut = &this->niMutex;
372         newMap = &this->newInode;
373         break;
374       case DIR_BLOCK:
375         mu = &this->daMutex;
376         mp = &this->dirBlockAccess;
377         newMut = &this->ndMutex;
378         newMap = &this->newDir;
379         break;
380       case DATA_BLOCK:
381         mu = &this->baMutex;
382         mp = &this->blocksAccess;
383         newMut = &this->ndMutex;
384         newMap = &this->newData;
385         break;
386       default:
387         return ENOSYS;
388     }
389
390     unique_lock<mutex> lk(*mu);
391     for (auto &&item :*mp) {
392       if (0 == item.first && INODE == type)
393         continue;
394       unique_lock<mutex> nlk(*newMut);
395       if (newMap->end() != newMap->find(item.first)) {
396         nlk.unlock();
397         continue;
398       }
399       nlk.unlock();
400
401       if (!(maxTimestamp > item.second && minTimestamp < item.second))
402         blocks.push_back(item.first);
403     }
```

```
404
405     return 0;
406   }
407
408   void Volume::purgeNewMap(const blockType &type) {
409     mutex *mu = nullptr;
410     map<uint64_t, uint64_t> *ma = nullptr;
411
412     switch (type) {
413       case INODE:
414         mu = &this->niMutex;
415         ma = &this->newInode;
416         break;
417       case DIR_BLOCK:
418         mu = &this->ndMutex;
419         ma = &this->newData;
420         break;
421       case DATA_BLOCK:
422         mu = &this->nbMutex;
423         ma = &this->newData;
424         break;
425       default:
426         return;
427     }
428
429     uint64_t newTimestamp = time(nullptr) - NEW_DELAY;
430
431     unique_lock<mutex> lk(*mu);
432     for (auto &&item :*ma) {
433       if (newTimestamp > item.second)
434         ma->erase(item.first);
435     }
436   }
437
438 }  // namespace mtfs
```

Listing 3.31 – "core/Volume.cpp"

3.5 *Migrator*

3.5.1 Migrator.h

```
1  /**
2   * \file Migrator.h
3   * \brief
4   * \author David Wittwer
5   * \version 0.0.1
6   * \copyright GNU Publis License V3
7   *
8   * This file is part of MTFS.
9
10    MTFS is free software: you can redistribute it and/or modify
11    it under the terms of the GNU General Public License as published by
12    the Free Software Foundation, either version 3 of the License, or
13    (at your option) any later version.
14
15    Foobar is distributed in the hope that it will be useful,
```

```
16      but WITHOUT ANY WARRANTY; without even the implied warranty of
17      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18      GNU General Public License for more details.
19
20      You should have received a copy of the GNU General Public License
21      along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22   */
23
24
25  #ifndef MTFS_MIGRATOR_H
26  #define MTFS_MIGRATOR_H
27
28  #include <mutex>
29  #include <condition_variable>
30  #include "PoolManager.h"
31
32  namespace mtfs {
33    class Migrator {
34    private:
35      static const int MIGRATION_DELAY = 10;
36
37    public:
38      struct info_st {
39        std::mutex *endMutex;
40        std::condition_variable condV;
41        bool end;
42        PoolManager *poolManager;
43
44        info_st() : endMutex(new std::mutex), end(false) {};
45
46        ~info_st() {
47          delete endMutex;
48        }
49      };
50
51      static void main(info_st *infos);
52    };
53  }
54
55
56  #endif //MTFS_MIGRATOR_H
```

Listing 3.32 – "migrator/Migrator.h"

3.5.2 Migrator.cpp

```
1   /**
2    * \file Migrator.cpp
3    * \brief
4    * \author David Wittwer
5    * \version 0.0.1
6    * \copyright GNU Publis License V3
7    *
8    * This file is part of MTFS.
9
10     MTFS is free software: you can redistribute it and/or modify
11     it under the terms of the GNU General Public License as published by
12     the Free Software Foundation, either version 3 of the License, or
```

```
13      (at your option) any later version.
14
15      Foobar is distributed in the hope that it will be useful,
16      but WITHOUT ANY WARRANTY; without even the implied warranty of
17      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18      GNU General Public License for more details.
19
20      You should have received a copy of the GNU General Public License
21      along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22  */
23
24 #include <zconf.h>
25 #include <iostream>
26 #include <utils/Logger.h>
27 #include "mtfs/Migrator.h"
28
29 #define MIGRATION_DELAY chrono::seconds(30)
30
31 using namespace std;
32
33 void mtfs::Migrator::main(info_st *infos) {
34
35   unique_lock<mutex> lk(*infos->endMutex);
36   while (!infos->end) {
37     lk.unlock();
38
39     lk.lock();
40     infos->condV.wait_for(lk, MIGRATION_DELAY);
41     if (infos->end)
42       break;
43     lk.unlock();
44
45 //    Do migration;
46     Logger::getInstance()->log("MIGRATOR", "start inode migration", Logger::
    L_DEBUG);
47     infos->poolManager->doMigration(blockType::INODE);
48     Logger::getInstance()->log("MIGRATOR", "start dirblock migration", Logger::
    L_DEBUG);
49     infos->poolManager->doMigration(blockType::DIR_BLOCK);
50     Logger::getInstance()->log("MIGRATOR", "start datablock migration", Logger
    ::L_DEBUG);
51     infos->poolManager->doMigration(blockType::DATA_BLOCK);
52     Logger::getInstance()->log("MIGRATOR", "end migration", Logger::L_DEBUG);
53
54     lk.lock();
55   }
56   lk.unlock();
57
58 }
```

Listing 3.33 – "migrator/Migrator.cpp"

3.6 Tests

3.7 Migration test

Listing 3.34 – "migrationTests.py"

3.8 test de vitesse

```
/**
 * \file main.cpp
 * \brief
 * \author David Wittwer
 * \version 0.0.1
 * \copyright GNU Publis License V3
 *
 * This file is part of MTFS.

    MTFS is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Foobar is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <iostream>
#include <boost/filesystem.hpp>
#include <fcntl.h>

#define FIRST_SIZE 512
#define LAST_SIZE 131072
#define SAMPLE 5
#define LOOP 50

#define FILENAME "/tmp/mtfs/test24"

using namespace std;

int main(int argc, char **argv) {
  cout << "Start Mtfs perfs tests" << endl;

  ofstream file("sample.csv");

  creat(FILENAME, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);

  file << "octets ";
  for (int i = 1; i <= LOOP; ++i) {
    file << to_string(i) << " ";
  }
  file << endl;

  string readTable, writeTable;

  for (size_t i = FIRST_SIZE; i <= LAST_SIZE; i *= 2) {
```

```
53      string readRow, writeRow;
54      readRow = to_string(i);
55      writeRow = to_string(i);
56
57      for (int j = 0; j < LOOP; ++j) {
58        char buffer[i + 1];
59        char *rbuffer = static_cast<char *>(calloc(sizeof(char), i + 1));
60        for (int k = 0; k < i; ++k) {
61          buffer[k] = 'a';
62        }
63
64        clock_t writeSum = 0;
65        clock_t readSum = 0;
66
67        for (int l = 0; l < SAMPLE; ++l) {
68          int fd = open(FILENAME, O_SYNC | O_RDWR);
69          if (fd <= 0) {
70            cerr << "open error " << strerror(errno) << endl;
71            break;
72          }
73          ssize_t nb = 0;
74          clock_t start;
75
76          start = clock();
77          do {
78            nb += write(fd, buffer, i);
79          } while (nb < i);
80          writeSum += clock() - start;
81
82          if (nb < 0) {
83            cerr << "write error " << strerror(errno) << endl;
84            break;
85          }
86
87          lseek(fd, 0, SEEK_SET);
88
89          start = clock();
90          do {
91            nb += read(fd, buffer, i);
92          } while (nb < i);
93          readSum += clock() - start;
94          close(fd);
95        }
96        float wm = (float) writeSum / SAMPLE;
97        float rm = (float) readSum / SAMPLE;
98  //      cout << to_string(m) << " " << to_string((m / CLOCKS_PER_SEC) * 1000)
      << endl;
99        writeRow += " " + to_string(wm);
100       readRow += " " + to_string(rm);
101 //      file << to_string(m) << " ";
102     }
103
104     writeTable += writeRow + "\n";
105     readTable += readRow + "\n";
106     cout << "size " << to_string(i) << endl;
107
108   }
109
110   file << readTable << endl << writeTable << endl;
```

```
111
112 }
```

Listing 3.35 – "speedTest.cpp"

3.9 Plugin tests

3.9.1 block device

```
1  /**
2   * \file blockDeviceTests.cpp
3   * \brief
4   * \author David Wittwer
5   * \version 0.0.1
6   * \copyright GNU Publis License V3
7   *
8   * This file is part of MTFS.
9
10     MTFS is free software: you can redistribute it and/or modify
11     it under the terms of the GNU General Public License as published by
12     the Free Software Foundation, either version 3 of the License, or
13     (at your option) any later version.
14
15     Foobar is distributed in the hope that it will be useful,
16     but WITHOUT ANY WARRANTY; without even the implied warranty of
17     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18     GNU General Public License for more details.
19
20     You should have received a copy of the GNU General Public License
21     along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22  */
23
24 #include <gtest/gtest.h>
25 #include <BlockDevice/BlockDevice.h>
26 #include <mtfs/Mtfs.h>
27 //#include <mtfs/structs.h>
28
29
30 using namespace std;
31 using namespace pluginSystem;
32
33 #define HOME "/home/david/Cours/4eme/Travail_bachelor/Home/Plugins/"
34 #define BLOCK_SIZE 4096
35 #define DEVICE "/dev/tests"
36 #define FS_TYPE "ext4"
37
38 TEST(BlockDevice, attachDetach) {
39 #ifndef DEBUG
40   if (setuid(0) != 0)
41     cout << "fail setuid" << endl;
42 #endif
43
44   BlockDevice blockDevice;
45   map<string, string> params;
46   params.insert(make_pair("home", HOME));
47   params.insert(make_pair("blockSize", to_string(BLOCK_SIZE)));
48   params.insert(make_pair("devicePath", DEVICE));
```

```
49    params.insert(make_pair("fsType", FS_TYPE));
50    ASSERT_TRUE(blockDevice.attach(params));
51    ASSERT_TRUE(blockDevice.detach());
52  }
53
54  class BlockDeviceFixture : public ::testing::Test {
55  public:
56    BlockDeviceFixture() {
57      rootIdent.poolId = 0;
58      rootIdent.volumeId = 0;
59      rootIdent.id = 0;
60    }
61
62    virtual void SetUp() {
63      map<string, string> params;
64      params.insert(make_pair("home", HOME));
65      params.insert(make_pair("blockSize", "4096"));
66      params.insert(make_pair("devicePath", DEVICE));
67      params.insert(make_pair("fsType", FS_TYPE));
68      blockDevice.attach(params);
69    }
70
71    virtual void TearDown() {
72      blockDevice.detach();
73    }
74
75    ~BlockDeviceFixture() {
76    }
77
78    BlockDevice blockDevice;
79    mtfs::ident_t rootIdent;
80
81  };
82
83  TEST_F(BlockDeviceFixture, addInode) {
84    uint64_t inode = 0;
85    ASSERT_EQ(0, blockDevice.add(&inode, mtfs::blockType::INODE));
86    ASSERT_NE(0, inode);
87
88    uint64_t inode2 = 0;
89    ASSERT_EQ(0, blockDevice.add(&inode2, mtfs::blockType::INODE));
90    ASSERT_NE(0, inode2);
91    ASSERT_GT(inode2, inode);
92
93    uint64_t inode3 = 0;
94    ASSERT_EQ(0, blockDevice.add(&inode3, mtfs::blockType::INODE));
95  }
96
97  TEST_F(BlockDeviceFixture, delInode) {
98    uint64_t inode;
99    blockDevice.add(&inode, mtfs::blockType::INODE);
100   ASSERT_EQ(0, blockDevice.del(inode, mtfs::blockType::INODE));
101
102   uint64_t inode2;
103   blockDevice.add(&inode2, mtfs::blockType::INODE);
104   blockDevice.add(&inode2, mtfs::blockType::INODE);
105   blockDevice.add(&inode2, mtfs::blockType::INODE);
106   blockDevice.del(inode, mtfs::blockType::INODE);
107 }
```

```
108
109 TEST_F(BlockDeviceFixture, writeInode) {
110   uint64_t inodeId;
111   blockDevice.add(&inodeId, mtfs::blockType::INODE);
112
113   mtfs::ident_t oIdent;
114   oIdent.id = 42;
115   oIdent.volumeId = 1;
116   oIdent.poolId = 1;
117
118   mtfs::inode_t ino;
119   ino.accesRight = 0666;
120   ino.uid = 0;
121   ino.gid = 0;
122   ino.size = 1024;
123   ino.linkCount = 1;
124   ino.atime = (unsigned long &&) time(nullptr);
125
126   for (int i = 0; i < 4; ++i) {
127     vector<mtfs::ident_t> blocks;
128
129     for (uint j = 0; j < 3; ++j) {
130       mtfs::ident_t ident = mtfs::ident_t(1, j, i * 3 + j);
131
132       blocks.push_back(ident);
133     }
134     ino.dataBlocks.push_back(blocks);
135   }
136
137   ASSERT_EQ(0, blockDevice.put(inodeId, &ino, mtfs::blockType::INODE, false));
138 }
139
140 TEST_F(BlockDeviceFixture, readInode) {
141   mtfs::inode_t original, inode;
142   memset(&original, 0, sizeof(mtfs::inode_t));
143   memset(&inode, 0, sizeof(mtfs::inode_t));
144
145   original.accesRight = 0644;
146   original.uid = 1;
147   original.gid = 1;
148   original.size = 1024;
149   original.linkCount = 1;
150   original.atime = (unsigned long &&) time(nullptr);
151
152   for (int i = 0; i < 4; ++i) {
153     vector<mtfs::ident_t> blocks;
154
155     for (uint j = 0; j < 3; ++j) {
156       mtfs::ident_t ident = mtfs::ident_t(1, j, i * 3 + j);
157
158       blocks.push_back(ident);
159     }
160     original.dataBlocks.push_back(blocks);
161   }
162
163   uint64_t inodeId;
164   blockDevice.add(&inodeId, mtfs::blockType::INODE);
165   blockDevice.put(inodeId, &original, mtfs::blockType::INODE, false);
```

```
166   ASSERT_EQ(0, blockDevice.get(inodeId, &inode, mtfs::blockType::INODE, false))
        ;
167
168 //  cout << original.accesRight << " " << inode.accesRight << endl;
169   ASSERT_EQ(original, inode);
170   original.size = 2048;
171   blockDevice.put(inodeId, &original, mtfs::blockType::INODE, false);
172   ASSERT_EQ(0, blockDevice.get(inodeId, &inode, mtfs::blockType::INODE, false))
        ;
173 }
174
175 TEST_F(BlockDeviceFixture, addBlock) {
176   uint64_t block = 0;
177   ASSERT_EQ(0, blockDevice.add(&block, mtfs::blockType::DATA_BLOCK));
178   ASSERT_NE(0, block);
179
180   uint64_t block2 = 0;
181   ASSERT_EQ(0, blockDevice.add(&block2, mtfs::blockType::DATA_BLOCK));
182   ASSERT_NE(0, block2);
183   ASSERT_GT(block2, block);
184
185   uint64_t block3 = 0;
186   ASSERT_EQ(0, blockDevice.add(&block3, mtfs::blockType::DATA_BLOCK));
187 }
188
189 TEST_F(BlockDeviceFixture, delBlock) {
190   uint64_t block;
191   blockDevice.add(&block, mtfs::blockType::DATA_BLOCK);
192   ASSERT_EQ(0, blockDevice.del(block, mtfs::blockType::DATA_BLOCK));
193
194   uint64_t block2;
195   blockDevice.add(&block2, mtfs::blockType::DATA_BLOCK);
196   blockDevice.add(&block2, mtfs::blockType::DATA_BLOCK);
197   blockDevice.add(&block2, mtfs::blockType::DATA_BLOCK);
198   blockDevice.del(block, mtfs::blockType::DATA_BLOCK);
199 }
200
201 TEST_F(BlockDeviceFixture, writeBlock) {
202   uint8_t block[BLOCK_SIZE];
203   memset(block, 0, BLOCK_SIZE);
204   for (uint8_t i = 0; i < 50; ++i) {
205     block[i] = 'a';
206   }
207
208   uint64_t blockId;
209   blockDevice.add(&blockId, mtfs::blockType::DATA_BLOCK);
210   ASSERT_EQ(0, blockDevice.put(blockId, &block, mtfs::blockType::DATA_BLOCK,
        false));
211 }
212
213 TEST_F(BlockDeviceFixture, readBlock) {
214   uint8_t block[BLOCK_SIZE];
215   memset(block, 0, BLOCK_SIZE);
216
217   uint8_t readBlock[BLOCK_SIZE];
218   memset(readBlock, 0, BLOCK_SIZE);
219
220   for (int i = 0; i < 500; ++i) {
221     block[i] = (uint8_t) to_string(i)[0];
```

```
222   }
223
224   uint64_t blockId;
225   blockDevice.add(&blockId, mtfs::blockType::DATA_BLOCK);
226   blockDevice.put(blockId, &block, mtfs::blockType::DATA_BLOCK, false);
227
228   ASSERT_EQ(0, blockDevice.get(blockId, &readBlock, mtfs::blockType::DATA_BLOCK
        , false));
229   ASSERT_TRUE(0 == memcmp(block, readBlock, BLOCK_SIZE));
230 }
231
232 TEST_F(BlockDeviceFixture, rootInode) {
233   mtfs::inode_t inode;
234   inode.accesRight = 0444;
235   inode.uid = 1000;
236   inode.gid = 1000;
237   inode.size = 1024;
238   inode.linkCount = 2;
239   inode.atime = (uint64_t) time(nullptr);
240   inode.dataBlocks.clear();
241
242   mtfs::inode_t readInode;
243
244   ASSERT_EQ(0, blockDevice.put(0, &inode, mtfs::blockType::INODE, false));
245   ASSERT_EQ(0, blockDevice.get(0, &readInode, mtfs::blockType::INODE, false));
246   ASSERT_EQ(inode, readInode);
247 }
248
249 TEST_F(BlockDeviceFixture, superblock) {
250   mtfs::superblock_t superblock;
251   superblock.iCacheSz = superblock.dCacheSz = superblock.bCacheSz = superblock.
        blockSz = 4096;
252   superblock.migration = superblock.redundancy = 1;
253   superblock.pools.clear();
254   for (int i = 0; i < 5; ++i) {
255     mtfs::pool_t pool;
256     pool.migration = 0;
257     pool.rule = nullptr;
258     pool.volumes.clear();
259     superblock.pools[i] = pool;
260   }
261
262
263   ASSERT_TRUE(blockDevice.putSuperblock(superblock));
264 }
265
266 TEST_F(BlockDeviceFixture, putMetas) {
267   mtfs::blockInfo_t blockInfo;
268
269   mtfs::ident_t ident1(1, 1, 1);
270   mtfs::ident_t ident2(2, 2, 2);
271
272   blockInfo.referenceId.push_back(ident1);
273   blockInfo.referenceId.push_back(ident2);
274   blockInfo.lastAccess = (uint64_t) time(nullptr);
275
276   ASSERT_EQ(0, blockDevice.put(1, &blockInfo, mtfs::blockType::INODE, true));
277   ASSERT_EQ(0, blockDevice.put(1, &blockInfo, mtfs::blockType::DIR_BLOCK, true)
        );
```

```
278   ASSERT_EQ(0, blockDevice.put(1, &blockInfo, mtfs::blockType::DATA_BLOCK, true
        ));
279 }
280
281 TEST_F(BlockDeviceFixture, getMetas) {
282   mtfs::blockInfo_t blockInfo = mtfs::blockInfo_t(), receiveInfo = mtfs::
        blockInfo_t();
283 //  memset(&blockInfo, 0, sizeof(mtfs::blockInfo_t));
284 //  memset(&receiveInfo, 0, sizeof(mtfs::blockInfo_t));
285
286   mtfs::ident_t ident1(1, 1, 1);
287   mtfs::ident_t ident2(2, 2, 2);
288
289   blockInfo.referenceId.push_back(ident1);
290   blockInfo.referenceId.push_back(ident2);
291   blockInfo.lastAccess = (uint64_t) time(nullptr);
292
293   ASSERT_EQ(0, blockDevice.put(2, &blockInfo, mtfs::blockType::INODE, true));
294   ASSERT_EQ(0, blockDevice.get(2, &receiveInfo, mtfs::blockType::INODE, true));
295   ASSERT_EQ(blockInfo, receiveInfo);
296
297   receiveInfo = mtfs::blockInfo_t();
298   ASSERT_EQ(0, blockDevice.put(2, &blockInfo, mtfs::blockType::DIR_BLOCK, true)
        );
299   ASSERT_EQ(0, blockDevice.get(2, &receiveInfo, mtfs::blockType::DIR_BLOCK,
        true));
300   ASSERT_EQ(blockInfo, receiveInfo);
301
302   receiveInfo = mtfs::blockInfo_t();
303   ASSERT_EQ(0, blockDevice.put(2, &blockInfo, mtfs::blockType::DATA_BLOCK, true
        ));
304   ASSERT_EQ(0, blockDevice.get(2, &receiveInfo, mtfs::blockType::DATA_BLOCK,
        true));
305   ASSERT_EQ(blockInfo, receiveInfo);
306 }
307
308 TEST_F(BlockDeviceFixture, putDirBlock) {
309   mtfs::ident_t id1, id2;
310   id2.poolId = id1.poolId = 1;
311   id1.volumeId = 1;
312   id1.id = 2;
313   id2.volumeId = 2;
314   id2.id = 42;
315
316   vector<mtfs::ident_t> ids;
317   ids.push_back(id1);
318   ids.push_back(id2);
319
320   mtfs::dirBlock_t block = mtfs::dirBlock_t();
321   block.entries.clear();
322   block.entries.insert(make_pair("baz", ids));
323   block.entries.insert(make_pair("test", ids));
324
325   uint64_t dId = 0;
326
327   ASSERT_EQ(0, blockDevice.add(&dId, mtfs::blockType::DIR_BLOCK));
328   EXPECT_LE(0, dId);
329   ASSERT_EQ(0, blockDevice.put(dId, &block, mtfs::blockType::DIR_BLOCK, false))
        ;
```

```
330
331   mtfs::dirBlock_t readBlock;
332   ASSERT_EQ(0, blockDevice.get(dId, &readBlock, mtfs::blockType::DIR_BLOCK,
        false));
333   ASSERT_EQ(block.entries.size(), readBlock.entries.size());
334   for (auto &&item: block.entries) {
335     ASSERT_NE(readBlock.entries.end(), readBlock.entries.find(item.first));
336     int i = 0;
337     for (auto &&ident: item.second) {
338       ASSERT_EQ(ident, readBlock.entries[item.first][i]);
339       i++;
340     }
341   }
342
343   ASSERT_EQ(0, blockDevice.del(dId, mtfs::blockType::DIR_BLOCK));
344 }
345
346 int main(int argc, char **argv) {
347   ::testing::InitGoogleTest(&argc, argv);
348   return RUN_ALL_TESTS();
349 }
```

Listing 3.36 – "blockDeviceTests.cpp"

3.9.2 S3

```
1  /**
2   * \file s3Tests.cpp
3   * \brief
4   * \author David Wittwer
5   * \version 0.0.1
6   * \copyright GNU Publis License V3
7   *
8   * This file is part of MTFS.
9
10    MTFS is free software: you can redistribute it and/or modify
11    it under the terms of the GNU General Public License as published by
12    the Free Software Foundation, either version 3 of the License, or
13    (at your option) any later version.
14
15    Foobar is distributed in the hope that it will be useful,
16    but WITHOUT ANY WARRANTY; without even the implied warranty of
17    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18    GNU General Public License for more details.
19
20    You should have received a copy of the GNU General Public License
21    along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
22  */
23
24 #include <gtest/gtest.h>
25 #include <S3/S3.h>
26 #include <mtfs/Mtfs.h>
27 //#include <mtfs/structs.h>
28
29 #define BLOCK_SIZE 4096
30
31 using namespace std;
32 using namespace pluginSystem;
```

```
33
34 #define HOME "/home/david/Cours/4eme/Travail_bachelor/Home/Plugins/"
35
36 TEST(S3, attachDetach) {
37 #ifndef DEBUG
38   if (setuid(0) != 0)
39   cout << "fail setuid" << endl;
40 #endif
41
42   S3 device;
43   map<string, string> params;
44   params.insert(make_pair("home", HOME));
45   params.insert(make_pair("blockSize", to_string(BLOCK_SIZE)));
46   params.emplace("region", "eu-central-1");
47   params.emplace("bucket", "mtfs");
48   ASSERT_TRUE(device.attach(params));
49   ASSERT_TRUE(device.detach());
50 }
51
52 class S3Fixture : public ::testing::Test {
53 public:
54   S3Fixture() {
55     rootIdent.poolId = 0;
56     rootIdent.volumeId = 0;
57     rootIdent.id = 0;
58   }
59
60   virtual void SetUp() {
61     map<string, string> params;
62     params.emplace("home", HOME);
63     params.emplace("blockSize", to_string(BLOCK_SIZE));
64     params.emplace("region", "eu-central-1");
65     params.emplace("bucket", "mtfs");
66     s3.attach(params);
67   }
68
69   virtual void TearDown() {
70     s3.detach();
71   }
72
73   ~S3Fixture() {
74   }
75
76   S3 s3;
77   mtfs::ident_t rootIdent;
78
79 };
80
81 TEST_F(S3Fixture, addInode) {
82   uint64_t inode = 0;
83   ASSERT_EQ(0, s3.add(&inode, mtfs::blockType::INODE));
84   ASSERT_NE(0, inode);
85
86   uint64_t inode2 = 0;
87   ASSERT_EQ(0, s3.add(&inode2, mtfs::blockType::INODE));
88   ASSERT_NE(0, inode2);
89   ASSERT_GT(inode2, inode);
90
91   uint64_t inode3 = 0;
```

```
92    ASSERT_EQ(0, s3.add(&inode3, mtfs::blockType::INODE));
93  }
94
95  TEST_F(S3Fixture, delInode) {
96    uint64_t inode;
97    s3.add(&inode, mtfs::blockType::INODE);
98    ASSERT_EQ(0, s3.del(inode, mtfs::blockType::INODE));
99
100   uint64_t inode2;
101   s3.add(&inode2, mtfs::blockType::INODE);
102   s3.add(&inode2, mtfs::blockType::INODE);
103   s3.add(&inode2, mtfs::blockType::INODE);
104   s3.del(inode, mtfs::blockType::INODE);
105 }
106
107 TEST_F(S3Fixture, writeInode) {
108   uint64_t inodeId = 0;
109   s3.add(&inodeId, mtfs::blockType::INODE);
110
111   mtfs::ident_t oIdent;
112   oIdent.id = 42;
113   oIdent.volumeId = 1;
114   oIdent.poolId = 1;
115
116   mtfs::inode_t ino;
117   ino.accesRight = 0666;
118   ino.uid = 0;
119   ino.gid = 0;
120   ino.size = 1024;
121   ino.linkCount = 1;
122   ino.atime = (unsigned long &&) time(nullptr);
123
124   for (int i = 0; i < 4; ++i) {
125     vector<mtfs::ident_t> blocks;
126
127     for (uint j = 0; j < 3; ++j) {
128       mtfs::ident_t ident = mtfs::ident_t(1, j, i * 3 + j);
129
130       blocks.push_back(ident);
131     }
132     ino.dataBlocks.push_back(blocks);
133   }
134
135   ASSERT_EQ(0, s3.put(inodeId, &ino, mtfs::blockType::INODE, false));
136 }
137
138 TEST_F(S3Fixture, readInode) {
139   mtfs::inode_t original, inode;
140   memset(&original, 0, sizeof(mtfs::inode_t));
141   memset(&inode, 0, sizeof(mtfs::inode_t));
142
143   original.accesRight = 0644;
144   original.uid = 1;
145   original.gid = 1;
146   original.size = 1024;
147   original.linkCount = 1;
148   original.atime = (unsigned long &&) time(nullptr);
149
150   for (int i = 0; i < 4; ++i) {
```

```
151     vector<mtfs::ident_t> blocks;
152
153     for (uint j = 0; j < 3; ++j) {
154       mtfs::ident_t ident = mtfs::ident_t(1, j, i * 3 + j);
155
156       blocks.push_back(ident);
157     }
158     original.dataBlocks.push_back(blocks);
159   }
160
161   uint64_t inodeId;
162   s3.add(&inodeId, mtfs::blockType::INODE);
163   s3.put(inodeId, &original, mtfs::blockType::INODE, false);
164   ASSERT_EQ(0, s3.get(inodeId, &inode, mtfs::blockType::INODE, false));
165
166 //  cout << original.accesRight << " " << inode.accesRight << endl;
167   ASSERT_EQ(original, inode);
168   original.size = 2048;
169   s3.put(inodeId, &original, mtfs::blockType::INODE, false);
170   ASSERT_EQ(0, s3.get(inodeId, &inode, mtfs::blockType::INODE, false));
171 }
172
173 TEST_F(S3Fixture, addBlock) {
174   uint64_t block = 0;
175   ASSERT_EQ(0, s3.add(&block, mtfs::blockType::DATA_BLOCK));
176 //  EXPECT_NE(0, block);
177
178   uint64_t block2 = 0;
179   ASSERT_EQ(0, s3.add(&block2, mtfs::blockType::DATA_BLOCK));
180   ASSERT_NE(0, block2);
181   ASSERT_GT(block2, block);
182
183   uint64_t block3 = 0;
184   ASSERT_EQ(0, s3.add(&block3, mtfs::blockType::DATA_BLOCK));
185 }
186
187 TEST_F(S3Fixture, delBlock) {
188   uint64_t block;
189   s3.add(&block, mtfs::blockType::DATA_BLOCK);
190   ASSERT_EQ(0, s3.del(block, mtfs::blockType::DATA_BLOCK));
191
192   uint64_t block2;
193   s3.add(&block2, mtfs::blockType::DATA_BLOCK);
194   s3.add(&block2, mtfs::blockType::DATA_BLOCK);
195   s3.add(&block2, mtfs::blockType::DATA_BLOCK);
196   s3.del(block, mtfs::blockType::DATA_BLOCK);
197 }
198
199 TEST_F(S3Fixture, writeBlock) {
200   uint8_t block[BLOCK_SIZE];
201   memset(block, 0, BLOCK_SIZE);
202   for (uint8_t i = 0; i < 50; ++i) {
203     block[i] = 'a';
204   }
205
206   uint64_t blockId = 0;
207   s3.add(&blockId, mtfs::blockType::DATA_BLOCK);
208   ASSERT_EQ(0, s3.put(blockId, &block, mtfs::blockType::DATA_BLOCK, false));
209 }
```

```
210
211 TEST_F(S3Fixture, readBlock) {
212   uint8_t block[BLOCK_SIZE];
213   memset(block, 0, BLOCK_SIZE);
214
215   uint8_t readBlock[BLOCK_SIZE];
216   memset(readBlock, 0, BLOCK_SIZE);
217
218   for (int i = 0; i < 500; ++i) {
219     block[i] = (uint8_t) to_string(i)[0];
220   }
221
222   uint64_t blockId = 0;
223   s3.add(&blockId, mtfs::blockType::DATA_BLOCK);
224   s3.put(blockId, &block, mtfs::blockType::DATA_BLOCK, false);
225
226   ASSERT_EQ(0, s3.get(blockId, &readBlock, mtfs::blockType::DATA_BLOCK, false))
      ;
227   ASSERT_TRUE(0 == memcmp(block, readBlock, BLOCK_SIZE));
228 }
229
230 TEST_F(S3Fixture, rootInode) {
231   mtfs::inode_t inode;
232   inode.accesRight = 0444;
233   inode.uid = 1000;
234   inode.gid = 1000;
235   inode.size = 1024;
236   inode.linkCount = 2;
237   inode.atime = (uint64_t) time(nullptr);
238   inode.dataBlocks.clear();
239
240   mtfs::inode_t readInode;
241
242   ASSERT_EQ(0, s3.put(0, &inode, mtfs::blockType::INODE, false));
243   ASSERT_EQ(0, s3.get(0, &readInode, mtfs::blockType::INODE, false));
244   ASSERT_EQ(inode, readInode);
245 }
246
247 TEST_F(S3Fixture, superblock) {
248   mtfs::superblock_t superblock;
249   superblock.iCacheSz = superblock.dCacheSz = superblock.bCacheSz = superblock.
      blockSz = 4096;
250   superblock.migration = superblock.redundancy = 1;
251   superblock.pools.clear();
252   for (int i = 0; i < 5; ++i) {
253     mtfs::pool_t pool;
254     pool.migration = 0;
255     pool.rule = nullptr;
256     pool.volumes.clear();
257     superblock.pools[i] = pool;
258   }
259
260
261   ASSERT_TRUE(s3.putSuperblock(superblock));
262 }
263
264 TEST_F(S3Fixture, putMetas) {
265   mtfs::blockInfo_t blockInfo;
266
```

```
267   mtfs::ident_t ident1(1, 1, 1);
268   mtfs::ident_t ident2(2, 2, 2);
269
270   blockInfo.referenceId.push_back(ident1);
271   blockInfo.referenceId.push_back(ident2);
272   blockInfo.lastAccess = (uint64_t) time(nullptr);
273
274   ASSERT_EQ(0, s3.put(1, &blockInfo, mtfs::blockType::INODE, true));
275   ASSERT_EQ(0, s3.put(1, &blockInfo, mtfs::blockType::DIR_BLOCK, true));
276   ASSERT_EQ(0, s3.put(1, &blockInfo, mtfs::blockType::DATA_BLOCK, true));
277 }
278
279 //TEST_F(S3Fixture, getMetas) {
280 //   mtfs::blockInfo_t blockInfo = mtfs::blockInfo_t(), receiveInfo = mtfs::
      blockInfo_t();
281 ////   memset(&blockInfo, 0, sizeof(mtfs::blockInfo_t));
282 ////   memset(&receiveInfo, 0, sizeof(mtfs::blockInfo_t));
283 //
284 //   mtfs::ident_t ident1(1, 1, 1);
285 //   mtfs::ident_t ident2(2, 2, 2);
286 //
287 //   blockInfo.referenceId.push_back(ident1);
288 //   blockInfo.referenceId.push_back(ident2);
289 //   blockInfo.lastAccess = (uint64_t) time(nullptr);
290 //
291 //   ASSERT_EQ(0, s3.put(2, &blockInfo, mtfs::blockType::INODE, true));
292 //   ASSERT_EQ(0, s3.get(2, &receiveInfo, mtfs::blockType::INODE, true));
293 //   ASSERT_EQ(blockInfo, receiveInfo);
294 //
295 //   receiveInfo = mtfs::blockInfo_t();
296 //   ASSERT_EQ(0, s3.put(2, &blockInfo, mtfs::blockType::DIR_BLOCK, true));
297 //   ASSERT_EQ(0, s3.get(2, &receiveInfo, mtfs::blockType::DIR_BLOCK, true));
298 //   ASSERT_EQ(blockInfo, receiveInfo);
299 //
300 //   receiveInfo = mtfs::blockInfo_t();
301 //   ASSERT_EQ(0, s3.put(2, &blockInfo, mtfs::blockType::DATA_BLOCK, true));
302 //   ASSERT_EQ(0, s3.get(2, &receiveInfo, mtfs::blockType::DATA_BLOCK, true));
303 //   ASSERT_EQ(blockInfo, receiveInfo);
304 //}
305
306 TEST_F(S3Fixture, putDirBlock) {
307   mtfs::ident_t id1, id2;
308   id2.poolId = id1.poolId = 1;
309   id1.volumeId = 1;
310   id1.id = 2;
311   id2.volumeId = 2;
312   id2.id = 42;
313
314   vector<mtfs::ident_t> ids;
315   ids.push_back(id1);
316   ids.push_back(id2);
317
318   mtfs::dirBlock_t block = mtfs::dirBlock_t();
319   block.entries.clear();
320   block.entries.insert(make_pair("baz", ids));
321   block.entries.insert(make_pair("test", ids));
322
323   uint64_t dId = 0;
324
```

```
325   ASSERT_EQ(0, s3.add(&dId, mtfs::blockType::DIR_BLOCK));
326   EXPECT_LE(0, dId);
327   ASSERT_EQ(0, s3.put(dId, &block, mtfs::blockType::DIR_BLOCK, false));
328
329   mtfs::dirBlock_t readBlock;
330   ASSERT_EQ(0, s3.get(dId, &readBlock, mtfs::blockType::DIR_BLOCK, false));
331   ASSERT_EQ(block.entries.size(), readBlock.entries.size());
332   for (auto &&item: block.entries) {
333     ASSERT_NE(readBlock.entries.end(), readBlock.entries.find(item.first));
334     int i = 0;
335     for (auto &&ident: item.second) {
336       ASSERT_EQ(ident, readBlock.entries[item.first][i]);
337       i++;
338     }
339   }
340
341   ASSERT_EQ(0, s3.del(dId, mtfs::blockType::DIR_BLOCK));
342 }
343
344 int main(int argc, char **argv) {
345   ::testing::InitGoogleTest(&argc, argv);
346   return RUN_ALL_TESTS();
347 }
```

Listing 3.37 – "s3Tests.cpp"

Chapitre 4

CMakeLists

4.1 CMakeLists général

```
cmake_minimum_required(VERSION 3.5)
project(mtFS)

set(CMAKE_CXX_STANDARD 14)

include_directories(library)
include_directories(mtFS/include)

add_subdirectory(mtFS)
add_subdirectory(mtFS_Tests)
add_subdirectory(Plugin)
```

Listing 4.1 – "CMakeLists.txt"

4.2 CMakeLists mtfs

```
if (NOT DEFINED MTFS_HOME_DIR)
  set(MTFS_HOME_DIR /home/mtfs)
endif ()
add_definitions(-DMTFS_HOME_DIR="${MTFS_HOME_DIR}")
add_definitions(-DMTFS_PLUGIN_HOME="${MTFS_HOME_DIR}/Plugins/")
add_definitions(-DMTFS_CONFIG_DIR="${MTFS_HOME_DIR}/Configs/")
add_definitions(-DMTFS_INSTALL_DIR="${MTFS_HOME_DIR}/Systems/")

set(FS_HEAD
    include/mtfs/Cache.h
    include/mtfs/Acces.h
    include/mtfs/Mtfs.h
    include/mtfs/Pool.h
    include/mtfs/PoolManager.h
    include/mtfs/Rule.h
    include/mtfs/structs.h
    include/mtfs/TimeRule.h
    include/mtfs/UserRightRule.h
    include/mtfs/Volume.h
    include/mtfs/Migrator.h)

set(FS_SRC
    src/mtfs/Cache.cpp
```

```
24      src/mtfs/Mtfs.cpp
25      src/mtfs/Pool.cpp
26      src/mtfs/PoolManager.cpp
27      src/mtfs/Rule.cpp
28      src/mtfs/TimeRule.cpp
29      src/mtfs/UserRightRule.cpp
30      src/mtfs/Volume.cpp
31      src/mtfs/Migrator.cpp)
32
33  set(MTFUSE_HEAD
34      include/wrapper/FuseBase.h
35      include/wrapper/FuseCallback.h
36      include/wrapper/MtfsFuse.h
37      )
38
39  set(MTFUSE_SRC
40      src/wrapper/FuseBase.cpp
41      src/wrapper/FuseCallback.cpp
42      src/wrapper/MtfsFuse.cpp
43      )
44
45  set(CONF_HEAD
46      include/pluginSystem/PluginManager.h
47      include/pluginSystem/Plugin.h
48      src/utils/Semaphore.cpp include/utils/Semaphore.h src/mtfs/Migrator.cpp
        include/mtfs/Migrator.h)
49
50  set(CONF_SRC
51      src/pluginSystem/PluginManager.cpp
52      )
53
54  set(UTILS
55      include/utils/Fs.h
56      src/utils/Fs.cpp
57      src/utils/Semaphore.cpp
58      include/utils/Semaphore.h
59      src/utils/Logger.cpp
60      include/utils/Logger.h
61      )
62
63  set(CMAKE_PREFIX_PATH /opt/aws-build)
64  find_package(aws-sdk-cpp)
65  add_definitions(-DUSE_IMPORT_EXPORT)
66
67  add_executable(mtfsCreate src/mtfsCreate.cpp
68      ${FS_SRC} ${CONF_SRC} ${UTILS})
69
70  target_link_libraries(mtfsCreate fuse3 dl boost_system boost_filesystem
        boost_thread pthread)
71
72
73  #add_definitions(-DDEBUG)
74
75  add_executable(mtfsMount src/mtfsMount.cpp
76      ${CONF_HEAD} ${CONF_SRC}
77      ${FS_HEAD} ${FS_SRC}
78      ${MTFUSE_HEAD} ${MTFUSE_SRC}
79      ${UTILS}
80      )
```

```
81
82 target_link_libraries(mtfsMount fuse3 dl boost_system boost_filesystem
       boost_thread pthread)
```

Listing 4.2 – "mtFS/CMakeLists.txt"

4.3 CMakeLists plugin

```
1  add_subdirectory(BlockDevice)
2  add_subdirectory(S3)
3
4  set(PLUGIN_H ../mtFS/include/pluginSystem/Plugin.h)
5
6  #set(BLOCK_SRCS BlockDevice/BlockDevice.h BlockDevice/BlockDevice.cpp ${
       PLUGIN_H})
7  #add_executable(Block ${BLOCK_SRCS})
8  #
9  #set(S3_SRCS S3/S3.h S3/S3.cpp ${PLUGIN_H})
10 #add_executable(S3exe ${S3_SRCS})
11 #target_link_libraries(S3exe aws-cpp-sdk-s3)
```

Listing 4.3 – "Plugin/CMakeLists.txt"

4.3.1 CMakeLists block

```
1  project(blockDevice)
2
3  set(CMAKE_BUILD_TYPE Release)
4
5  add_library(block SHARED BlockDevice.cpp ../../mtFS/src/utils/Logger.cpp)
```

Listing 4.4 – "Plugin/BlockDevice/CMakeLists.txt"

4.3.2 CMakeLists S3

```
1  project(S3)
2
3  set(CMAKE_BUILD_TYPE Release)
4
5  add_library(s3 SHARED S3.cpp ../../mtFS/src/utils/Logger.cpp)
6  target_link_libraries(s3 aws-cpp-sdk-s3)
```

Listing 4.5 – "Plugin/S3/CMakeLists.txt"

4.4 CMakeLists Tests

```
1  project(mtFS_Tests)
2
3  add_subdirectory(PluginTests)
4  add_subdirectory(PerformanceTests)
```

Listing 4.6 – "mtFS_Tests/CMakeLists.txt"

4.4.1 CMakeLists Performance tests

```
add_executable(perfTests main.cpp)

target_link_libraries(perfTests boost_system boost_filesystem boost_thread)
```

Listing 4.7 – "mtFS_Tests/PerformanceTests/CMakeLists.txt"

4.4.2 CMakeLists PluginTests

```
include_directories(${gtest_SOURCE_DIR}/include ${gtest_SOURCE_DIR})
include_directories(../../mtFS/include ../../Plugin/)

set(BLOCK_SRCS ../../Plugin/BlockDevice/BlockDevice.h ../../Plugin/BlockDevice/
    BlockDevice.cpp)

add_definitions(-DDEBUG)
add_executable(blockTests blockDeviceTests.cpp ${BLOCK_SRCS} ../../mtFS/src/
    utils/Logger.cpp)

target_link_libraries(blockTests gtest gtest_main boost_system)

set(CMAKE_PREFIX_PATH /opt/aws-build)

find_package(aws-sdk-cpp)
add_definitions(-DUSE_IMPORT_EXPORT)
set(S3_SRCS ../../Plugin/S3/S3.h ../../Plugin/S3/S3.cpp ../../mtFS/src/utils/
    Logger.cpp)
add_executable(s3Tests s3Tests.cpp ${S3_SRCS})

target_link_libraries(s3Tests gtest gtest_main boost_system aws-cpp-sdk-s3)
```

Listing 4.8 – "mtFS_Tests/PluginTests/CMakeLists.txt"

Bibliographie

[1] CMake. Compilateur c/c++. https://cmake.org/download/, 2017. [En ligne ; Page disponible le 16-août-2017].