Contents

| 1 | Basic Test Results | 2 |
|---|-----------------------|----|
| 2 | README | 3 |
| 3 | CachingFileSystem.cpp | 5 |
| 4 | Makefile | 17 |

1 Basic Test Results

- g++ -std=c++11 -Wall -c -fmessage-length=0 -D_FILE_OFFSET_BITS=64 'pkg-config fuse --cflags' CachingFileSystem.cpp g++ -Wall -std=c++11 -D_FILE_OFFSET_BITS=64 CachingFileSystem.o 'pkg-config fuse --libs' -o CachingFileSystem rm -f CachingFileSystem.o CachingFileSystem ** *core Reading /tmp/bodek.PddBJI/os/EX4_Submission/roigreenberg/presubmission/testdir/testTmp/README

2 README

```
1
    akiva_s, roigreenberg
2
    Akiva Sygal(305277220), Roi Greenberg(305571234)
3
4
    FILES:
    README
6
    CachingFileSystem.cpp -- a file that implements the Caching File System
9
10
    REMARKS:
    Most of the functins are implement same as the bbfs example and are very simple
11
12
    so no need to expplain them.
    We will explain how the read work
    for the cache we have struct containing general information as block nmber,
14
15
    directories path ect. nd also contain a vector Blocks.
    The Blocks are second struct, contain information for single block.
    This information is the path for the file the data took from, the number of the
17
    block in this file, a counter of use and f course the data itself.
    Each time a file beign read, we are going block by block.
19
    For each block we first look for an exists block. If we can't find one, we
20
21
    "activate" another block. If all the block are correntlly in use we search for
    the least frequancy block, and replace it with the new block.
22
23
    Then the function copy the data to the buffer and goes to the next rblock if
24
    needed.
25
26
27
    ANSWERS:
28
    1 by default, heap memory is saved on RAM and so accessing it is much faster
29
    than accessing the disk. *but* the heap is a virtual memory section and can
30
    somtimes be mapped into disk - no garantuee that our heap-cache would really
31
    be saved on RAM, so if it is actually saved on disk, we would get no better
    performances.
33
34
    2. no implementation is absolutely better: in Array implementation, reading
    a specificf block is easier since we have random access to it, but when we
35
36
    have to delete a block it would take O(n) time to find the ideal block to
    delete, so this implementation is better when you know that your program would
37
    read the same files (=blocks) a lot of times and would seldom read new files.
38
39
    In the other hand, the list implementation can easily delete the less used
    block but cannot access spesific blocks efficiently, so it should be preffered
    when we most of the time read new information and seldom read the same data
41
    several times.,
42
    3. the difference is that paging is used for *memory access*. memory access
43
    treated in user-level and so happens all time, so updating the data structure
44
    each time the memory is accessed (by user or so) is not realistic (veryy
    expensive). but reading a block from disk can be treated only by kernel - it
46
47
    requires a system call, takes much more time and so happens much seldom, so in
    this case updating the data structure in each access to a filr block is
    realistic.
49
    4. LRU better: if we are treating each file after reading it, and only when
50
    finishing the work with the current file we continue to read the next one.
51
    In this case, deleting the less recently block would delete a block from the
52
    oldest file imported, which most probabely is not needed antmore.
53
    LFU better: if we are treating files with different information to be handled,
54
55
    where some of the information is really important and have to be used a lot of
    time, where somw of the other files contain unimportent information that after
    reading once is not needed. in that case we want to delete the less important
57
    information, which is most likely the information we have read the less number
```

- $\,\,$ both strategies are bad: when we are a file server that only provides file
- $\,$ reading services to some user. we cannot know what is the next file the user
- 62 will ask us to supply, so each of the algorithm is only a guess since we don't
- $63\,$ $\,$ know the logic of files reading.
- 64 5. the size of blocks that the OS read from the file system. if we read smaller
- 65 blocks, it causes some "internal fragmentation" since when we want to read our
- small-size block, in any case the os would have to import the whole os-sized
- 67 block from the memory and the rest of it would be spended. if we read more than
- 68 the os block size each time, in each block importation we actually need to
- 69 import more than 1 block, making the import action much expensive than needed.

3 CachingFileSystem.cpp

```
2
     * CachingFileSystem.cpp
3
     * Created on: 15 April 2015
     * Author: Netanel Zakay, HUJI, 67808 (Operating Systems 2014-2015).
5
6
    #define FUSE_USE_VERSION 26
8
9
   #include <fuse.h>
10
11
   #include <list>
12
    #include <fcntl.h>
13 #include <vector>
14 #include <cstring>
15
    #include <string>
16 #include <string.h>
17 #include <errno.h>
18
   #include <stdio.h>
19 #include <unistd.h>
20 #include <fstream>
    #include <iostream>
21
22
   #include <limits.h>
23 #include <stdio.h>
   #include <stdlib.h>
24
25
   #include <map>
26 #include <dirent.h>
27
   #include <cassert>
    #include <stdexcept>
29
30
   # define BLOCK(x) (int)(x) / cache_data->block_size +1
31
    #define SET\_TIMER if ((timer=time(nullptr)) == (time\_t)(-1)) \{return - errno;\}
32
33
   using namespace std;
34
    struct fuse_operations caching_oper;
35
37
    ofstream logfile;
38
    time_t timer;
    class MainError: public exception
40
41
        virtual const char* what() const throw ()
42
43
44
            return "error in main";
45
    };
46
47
    class SystemError: public exception
48
^{49}
        virtual const char* what() const throw ()
50
51
52
            return strerror(errno);
53
    };
54
55
    typedef struct Block
56
57
        int num_of_accesses;
58
        char* data:
59
```

```
list<int> cache_index;
 60
 61
         string file_path;
 62
         size_t size;
 63
          int block_num;
         Block(size_t size);
 64
 65
          ~Block()
 66
              delete[](data);
 67
 68
         }
         void rename(string new_file_path)
 69
 70
 71
              file_path = new_file_path;
 72
          void init_block(const string file_path, int block_num);
 73
 74
          void reset_block(void);
 75
     } Block;
 76
 77
     Block::Block(size_t size)
 78
 79
 80
          this->size = size;
         file_path = "";
 81
         num_of_accesses = 0;
 82
         data = new char[size];
 83
          if (data == nullptr)
 84
 85
              bad_alloc ex;
 86
 87
              throw ex;
         }
 88
         for (size_t i = 0; i < size; i++)</pre>
 89
 90
              data[i] = '\0';
 91
 92
 93
          block_num = -1;
     }
94
 95
     void Block::init_block(string file_path, int block_num)
 96
97
 98
          this->file_path = file_path;
         num_of_accesses = 0;
99
          for (size_t i = 0; i < size; i++)</pre>
100
101
              data[i] = '\0';
102
103
          this->block_num = block_num;
104
     }
105
106
     void Block::reset_block(void)
107
108
          file_path = "";
109
         num_of_accesses = 0;
110
          for (size_t i = 0; i < size; i++)
111
112
              data[i] = '\0';
113
114
          block_num = -1;
115
     }
116
117
     typedef struct Cache
118
119
          vector<Block*> cache;
120
121
122
          char* rootdir;
         char* mountdir;
123
          int block_size;
124
125
          int num_of_blocks;
126
         string logfile_name = ".filesystem.log";
127
```

```
128
          unsigned int occupied_blocks;
          Cache(char* argv[]);
129
130
          ~Cache()
131
              vector<Block*>::iterator it;
132
              for (it = cache.begin(); it != cache.end(); it++)
133
134
              {
135
136
                  if ((*it) != nullptr)
137
                  {
                      delete(*it);
138
139
                  }
140
              free(rootdir);
141
142
              free(mountdir);
143
144
          Block * find_block(string path, const int block_num);
          Block * assign_new_block();
145
     } Cache;
146
147
     Cache::Cache(char* argv[])
148
149
          struct stat buffer;
150
         rootdir = realpath(argv[1], NULL);
151
152
          if ((rootdir == NULL) || stat(rootdir, &buffer) == -1)
153
          {
              throw MainError();
154
155
         }
          if (S_ISDIR(buffer.st_mode) == false)
156
157
          {
158
              throw MainError();
159
         mountdir = realpath(argv[2], NULL);
160
161
          if ((mountdir == NULL) || stat(mountdir, &buffer) == -1)
162
          {
163
              throw MainError();
164
          if (S_ISDIR(buffer.st_mode) == false)
165
166
          {
              throw MainError();
167
         }
168
          if ((num_of_blocks = atoi(argv[3])) <= 0)</pre>
169
170
          {
171
              throw MainError();
          }
172
          if ((block_size = atoi(argv[4])) <= 0)</pre>
173
174
          {
              throw MainError();
175
176
         }
177
          cache.resize(num_of_blocks, nullptr);
          vector<Block*>::iterator it:
178
179
          for (it = cache.begin(); it != cache.end(); it++)
180
181
              (*it) = new Block(block_size);
          }
182
          occupied_blocks = 0;
183
     }
184
185
186
187
      * searching for exists block
      * return pointer to the block or null if doesn't exists
188
189
190
     Block * Cache::find_block(string path, const int block_num)
191
          vector<Block*>::iterator it;
192
          for (it = cache.begin(); it != cache.end(); it++)
193
194
195
```

```
196
             if ((*it)->file_path.compare(path) == 0
197
                      && (*it)->block_num == block_num)
198
              {
199
                  return *it;
             }
200
         }
201
         return nullptr;
202
     }
203
204
205
      * assigning new block.
206
207
      * if can't find empty block, delete the least used and create new instead
208
     Block * Cache::assign_new_block()
209
210
211
212
         vector<Block*>::iterator it;
213
         for (it = cache.begin(); it != cache.end(); it++)
214
215
             if ((*it)->block_num == -1)
             {
216
217
                  return *it;
218
219
         }
          //in case the cache is full. free the least accessed block and assigned new
220
221
         int i = 0;
         int min = 0;
222
          for (it = cache.begin(); it != cache.end(); it++, i++)
223
224
225
             if ((*it)->num_of_accesses < cache.at(min)->num_of_accesses)
226
             {
                  min = i;
227
             }
228
229
         }
         cache.at(min)->reset_block();
230
231
         return cache.at(min);
232
233
     static Cache* cache_data;
234
235
     string translate_path(string virtual_path)
236
237
          if (virtual_path.compare("/" + cache_data->logfile_name) == 0)
238
239
             return "";
240
241
^{242}
         string real_path = cache_data->rootdir + virtual_path;
243
244
         return real_path;
     }
^{245}
246
247
     string get_virtual_path(string absolute_path)
248
249
         unsigned int pos = absolute_path.find(cache_data->rootdir);
250
          if (pos != string::npos)
251
              pos += strlen(cache_data->rootdir);
252
             absolute_path.erase(0, pos + 1);
253
254
255
         return absolute_path;
     }
^{256}
257
258
     /** Get file attributes.
259
      * Similar to stat(). The 'st_dev' and 'st_blksize' fields are
260
      * ignored. The 'st_ino' field is ignored except if the 'use_ino'
261
      * mount option is given.
262
263
```

```
264
     int caching_getattr(const char *path, struct stat *statbuf)
265
             SET TIMER
266
267
             logfile << timer << " getattr\n";</pre>
268
             string real_path = translate_path(path);
269
             if (real_path.compare("") == 0)
270
             {
271
272
                  return -ENOENT;
             }
273
             int retstat = 0:
274
275
             retstat = lstat(real_path.c_str(), statbuf);
276
             if (retstat != 0)
277
             {
278
                  return -errno;
279
280
281
             return retstat;
     }
282
283
284
      * Get attributes from an open file
285
286
      * This method is called instead of the getattr() method if the
287
288
      * file information is available.
289
      * Currently this is only called after the create() method if that
290
      * is implemented (see above). Later it may be called for
291
      * invocations of fstat() too.
292
293
294
      * Introduced in version 2.5
295
     int caching_fgetattr(const char *path, struct stat *statbuf,
296
297
             struct fuse_file_info *fi)
298
299
         SET_TIMER
         logfile << timer << " fgetattr\n";</pre>
300
301
         string real_path = translate_path(path);
302
         if (real_path.compare("") == 0)
303
304
         {
             return -ENOENT;
305
         }
306
         if (strcmp(path, "/") == 0)
307
308
         {
             return caching_getattr(path, statbuf);
309
310
         }
         int retstat = 0;
311
         retstat = fstat(fi->fh, statbuf);
312
         if (retstat != 0)
313
314
         {
315
             return -errno;
316
         }
317
         return retstat;
     }
318
319
320
      * Check file access permissions
321
322
      * This will be called for the access() system call. If the
323
      * 'default_permissions' mount option is given, this method is not
324
325
      * called.
326
      * This method is not called under Linux kernel versions 2.4.x
327
328
329
      * Introduced in version 2.5
330
331
    int caching_access(const char *path, int mask)
```

```
332
     {
333
          SET_TIMER
          logfile << timer << " access\n";</pre>
334
335
          string real_path = translate_path(path);
336
          if (real_path.compare("") == 0)
337
338
              return -ENOENT;
339
340
         }
          int retstat = 0;
341
         retstat = access(real_path.c_str(), mask);
342
343
          if (retstat != 0)
344
345
              return -errno;
346
          }
         return retstat;
347
348
     }
349
     /** File open operation
350
351
      * No creation, or truncation flags (O_CREAT, O_EXCL, O_TRUNC)
352
      * will be passed to open(). Open should check if the operation
353
       * is permitted for the given flags. Optionally open may also
354
       st return an arbitrary filehandle in the fuse_file_info structure,
355
       * which will be passed to all file operations.
356
357
      * pay attention that the max allowed path is PATH_MAX (in limits.h).
358
359
       * if the path is longer, return error.
360
361
      * Changed in version 2.2
362
     int caching_open(const char *path, struct fuse_file_info *fi)
363
364
     {
365
         logfile << timer << " open\n";</pre>
366
367
          int retstat = 0;
368
          int fd:
369
          string real_path = translate_path(path);
370
          if (real_path.compare("") == 0)
371
372
          {
              return -ENOENT;
373
         }
374
          if ((fi->flags & 3) != O_RDONLY)
375
376
          {
              return -EACCES;
377
378
          }
379
380
         fd = open(real_path.c_str(), fi->flags);
          if (retstat != 0)
381
382
          {
383
              return -errno;
384
385
          fi->fh = fd;
386
387
388
          return retstat;
     }
389
390
391
      * search for data block.
392
393
      * if can't find the correct block, create new one and write the data
394
      * return pointer to the data
395
     char * read_block(int fd, string path, int block_num)
396
397
          int retstat = 0;
398
399
          Block * block = cache_data->find_block(path, block_num);
```

```
400
         if (block == nullptr)
401
              block = cache_data->assign_new_block();
402
403
              block->init_block(path, block_num);
              retstat = pread(fd, block->data, cache_data->block_size,
404
                     (block_num - 1) * cache_data->block_size);
405
              if (retstat == -1)
406
              {
407
408
                  throw SystemError();
409
         }
410
411
         block->num_of_accesses++;
412
         return block->data;
     }
413
414
     /** Read data from an open file
415
416
417
      * Read should return exactly the number of bytes requested except
      * on EOF or error, otherwise the rest of the data will be
418
419
      * substituted with zeroes.
420
421
      * Changed in version 2.2
422
     int caching_read(const char *path, char *buf, size_t size, off_t offset,
423
424
              struct fuse_file_info *fi)
425
         SET TIMER:
426
427
         logfile << timer << " read\n";</pre>
428
429
         string real_path = translate_path(path);
430
          if (real_path.compare("") == 0)
431
         {
432
              return -ENOENT;
433
434
435
          struct stat statbuf;
         if (lstat(real_path.c_str(), &statbuf) != 0)
436
437
          {
438
              return -errno;
         }
439
440
         if (offset > statbuf.st_size)
441
         {
              return -ENXIO:
442
443
         }
         if ((int) (size + offset) > statbuf.st_size)
444
445
446
              size = statbuf.st_size - offset;
447
448
         int retstat = 0;
449
450
451
          int first_block = BLOCK(offset);
452
          int last_block = BLOCK(offset + size);
         offset = offset % cache_data->block_size;
453
         char * tmp;
454
455
         try
456
          {
              for (int j = first_block; j < last_block; ++j)</pre>
457
458
459
                  tmp = read_block(fi->fh, real_path, j);
460
461
                  memcpy(buf + retstat, tmp + offset, cache_data->block_size - offset);
462
                  retstat += cache_data->block_size - offset;
                  offset = 0;
463
              }
464
              int end = size - retstat;
465
              if (end > 0)
466
467
              {
```

```
468
                  tmp = read_block(fi->fh, real_path, last_block);
469
                  memcpy(buf + retstat, tmp + offset, end);
470
                  retstat += end:
             }
471
472
          catch (bad_alloc &e)
473
474
          {
             return -ENOMEM;
475
476
         }
         catch (SystemError &e)
477
478
479
              return -errno;
480
481
482
         return retstat;
     }
483
484
     /** Possibly flush cached data
485
486
      * BIG NOTE: This is not equivalent to fsync(). It's not a
487
      * request to sync dirty data.
488
489
      * Flush is called on each close() of a file descriptor. So if a
490
      * filesystem wants to return write errors in close() and the file
491
      * has cached dirty data, this is a good place to write back data
492
493
       * and return any errors. Since many applications ignore close()
      * errors this is not always useful.
494
495
      * NOTE: The flush() method may be called more than once for each
496
497
      st open(). This happens if more than one file descriptor refers
498
      * to an opened file due to dup(), dup2() or fork() calls. It is
       * not possible to determine if a flush is final, so each flush
499
      st should be treated equally. Multiple write-flush sequences are
500
501
      * relatively rare, so this shouldn't be a problem.
502
503
      * Filesystems shouldn't assume that flush will always be called
      * after some writes, or that if will be called at all.
504
505
      * Changed in version 2.2
506
507
     int caching_flush(const char *path, struct fuse_file_info *fi)
508
509
         SET TIMER
510
         logfile << timer << " flush\n";</pre>
511
         return 0;
512
     }
513
514
     /** Release an open file
515
516
      * Release is called when there are no more references to an open
517
      * file: all file descriptors are closed and all memory mappings
518
519
      * are unmapped.
520
      * For every open() call there will be exactly one release() call
521
      st with the same flags and file descriptor. It is possible to
       * have a file opened more than once, in which case only the last
523
      * release will mean, that no more reads/writes will happen on the
524
      * file. The return value of release is ignored.
525
526
527
      * Changed in version 2.2
528
529
     int caching_release(const char *path, struct fuse_file_info *fi)
530
531
         logfile << timer << " release\n";</pre>
532
533
         int retstat = close(fi->fh);
534
535
         if (retstat != 0)
```

```
536
537
              return -errno;
538
539
          return retstat;
     }
540
541
     /** Open directory
542
543
544
      * This method should check if the open operation is permitted for
      * this directory
545
546
547
      * Introduced in version 2.3
548
     int caching_opendir(const char *path, struct fuse_file_info *fi)
549
550
     {
551
          logfile << timer << " opendir\n";</pre>
552
553
          DIR *dp;
554
555
          int retstat = 0;
          string real_path = translate_path(path);
556
557
          dp = opendir(real_path.c_str());
558
          if (dp == NULL)
559
560
561
              return -errno;
562
563
          fi->fh = (intptr_t) dp;
564
565
566
          return retstat;
     }
567
568
569
     /** Read directory
570
571
      * This supersedes the old getdir() interface. New applications
572
      * should use this.
573
      * The readdir implementation ignores the offset parameter, and
574
      * passes zero to the filler function's offset. The filler
* function will not return '1' (unless an error happens), so the
575
576
       * whole directory is read in a single readdir operation. This
577
       * works just like the old getdir() method.
578
579
       * Introduced in version 2.3
580
581
582
     int caching_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
             off_t offset, struct fuse_file_info *fi)
583
584
          SET_TIMER;
585
          logfile << timer << " readdir\n";</pre>
586
587
588
          int retstat = 0;
          DIR *dp;
589
          struct dirent *de;
590
591
          dp = (DIR *) (uintptr_t) fi->fh;
592
          int prev_errno = errno;
593
594
          while ((de = readdir(dp)) != NULL)
595
596
              if ((strcmp(de->d_name, cache_data->logfile_name.c_str())) != 0)
597
598
                   if ((filler(buf, de->d_name, NULL, 0) != 0))
599
600
                  {
                       return -ENOMEM;
601
602
              }
603
```

```
604
             prev_errno = errno;
605
         };
606
607
          if (prev_errno != errno)
608
          {
609
              return -errno;
          }
610
611
612
          return retstat;
     }
613
614
615
     /** Release directory
616
      * Introduced in version 2.3
617
618
     int caching_releasedir(const char *path, struct fuse_file_info *fi)
619
620
621
         logfile << timer << " releasedir\n";</pre>
622
623
         int retstat = 0;
624
625
          if (closedir((DIR *) (uintptr_t) fi->fh) == -1)
626
627
         {
628
              return -errno;
629
630
631
          return retstat;
     }
632
633
634
      /** Rename a file */
     int caching_rename(const char *path, const char *newpath)
635
636
     {
637
          SET_TIMER;
         logfile << timer << " rename\n";</pre>
638
639
          int retstat = 0;
640
          string new_path = translate_path(newpath);
641
642
          string real_path = translate_path(path);
          if (real_path == "")
643
644
          {
             return -ENONET;
645
         }
646
          if (new_path == "")
647
648
          {
              return -EINVAL;
649
650
         }
         retstat = rename(real_path.c_str(), new_path.c_str());
651
652
          if (retstat != 0)
653
654
          {
655
              return -errno;
656
657
658
          vector<Block*>::iterator it;
         for (it = cache_data->cache.begin(); it != cache_data->cache.end(); it++)
659
660
              if ((*it)->file_path.compare(path) == 0)
661
              {
662
663
                  (*it)->file_path = new_path;
664
          }
665
666
          return retstat;
     }
667
668
669
      * Initialize filesystem
670
671
```

```
672
      * The return value will passed in the private_data field of
673
      * fuse_context to all file operations and as a parameter to the
      *\ destroy()\ method.
674
675
      * Introduced in version 2.3
676
677
      * Changed in version 2.6
678
     void *caching_init(struct fuse_conn_info *conn) {
679
680
         logfile << time(nullptr) << " init\n";</pre>
          return cache_data;
681
     }
682
683
684
      * Clean up filesystem
685
686
      * Called on filesystem exit.
687
688
      * Introduced in version 2.3
689
690
     void caching_destroy(void *userdata) {
691
         logfile << time(nullptr) << " destroy\n";</pre>
692
693
          logfile.close();
694
          delete(cache_data);
695
     }
696
697
698
699
      * Ioctl from the FUSE sepc:
      * flags will have FUSE_IOCTL_COMPAT set for 32bit ioctls in
700
701
      st 64bit environment. The size and direction of data is
702
      * determined by _IOC_*() decoding of cmd. For _IOC_NONE,
      * data will be NULL, for _IOC_WRITE data is out area, for
703
      * \_IOC\_READ in area and if both are set in/out area. In all
704
705
      * non-NULL cases, the area is of _IOC_SIZE(cmd) bytes.
706
707
      * However, in our case, this function only needs to print cache table to the log file .
708
709
      * Introduced in version 2.8
710
     int caching_ioctl(const char *, int cmd, void *arg, struct fuse_file_info *,
711
712
              unsigned int flags, void *data)
713
          SET TIMER:
714
          logfile << timer << " ioctl\n";</pre>
715
716
          for (vector<Block*>::iterator it = cache_data->cache.begin();
717
718
                  it != cache_data->cache.end(); ++it)
          ₹
719
720
              if ((*it)->block_num != -1)
721
              {
                  logfile << get_virtual_path((**it).file_path) << " " << (**it).block_num</pre>
722
                      << " " << (**it).num_of_accesses << endl;
723
724
              }
          }
725
726
          return 0;
     }
727
728
     // Initialise the operations.
729
     // You are not supposed to change this function.
730
731
     void init_caching_oper() {
732
733
          caching_oper.getattr = caching_getattr;
734
          caching_oper.access = caching_access;
          caching_oper.open = caching_open;
735
          caching_oper.read = caching_read;
736
          caching_oper.flush = caching_flush;
737
          caching_oper.release = caching_release;
738
739
          caching_oper.opendir = caching_opendir;
```

```
740
          caching_oper.readdir = caching_readdir;
          caching_oper.releasedir = caching_releasedir;
741
742
          caching_oper.rename = caching_rename;
743
          caching_oper.init = caching_init;
          caching_oper.destroy = caching_destroy;
744
745
          caching_oper.ioctl = caching_ioctl;
          caching_oper.fgetattr = caching_fgetattr;
746
747
748
          caching_oper.readlink = NULL;
          caching_oper.getdir = NULL;
749
          caching_oper.mknod = NULL;
750
751
          caching_oper.mkdir = NULL;
          caching_oper.unlink = NULL;
752
753
          caching_oper.rmdir = NULL;
754
          caching_oper.symlink = NULL;
          caching_oper.link = NULL;
755
756
          caching_oper.chmod = NULL;
          caching_oper.chown = NULL;
757
          caching_oper.truncate = NULL;
758
759
          caching_oper.utime = NULL;
          caching_oper.write = NULL;
760
761
          caching_oper.statfs = NULL;
          caching_oper.fsync = NULL;
762
          caching_oper.setxattr = NULL;
763
          caching_oper.getxattr = NULL;
764
          caching_oper.listxattr = NULL;
765
          caching_oper.removexattr = NULL;
766
767
          caching_oper.fsyncdir = NULL;
          caching_oper.create = NULL;
768
769
          caching_oper.ftruncate = NULL;
770
     }
771
772
     int main(int argc, char* argv[]) {
773
          try {
              if (argc != 5)
774
775
              {
                  throw MainError();
776
              }
777
              cache_data = new Cache(argv);
778
779
              logfile.open("/" + string(cache_data->rootdir) +"/" + cache_data->logfile_name, ios::app);
780
781
              init_caching_oper();
782
783
              argv[1] = argv[2];
784
              for (int i = 2; i < (argc - 1); i++) {
785
786
                  argv[i] = NULL;
787
788
              argv[2] = (char*) "-s";
789
              argc = 3;
790
791
              int fuse_stat = fuse_main(argc, argv, &caching_oper, cache_data);
792
              return fuse_stat;
         }
793
          catch(MainError& e)
794
795
796
              cout << "usage: CachingFileSystem rootdir mountdir numberOfBlocks blockSize" << endl;</pre>
797
              free(cache_data);
798
799
              exit(0);
800
801
          catch (exception &e) {
              cerr << "System Error " << e.what() << endl;;</pre>
802
              free(cache_data);
803
804
              exit(0);
          }
805
     }
806
```

4 Makefile

```
CC = g++ -Wall
1
    FLAG = -std=c++11 -D_FILE_OFFSET_BITS=64
    FUSE = 'pkg-config fuse --cflags --libs'
   LIBSRC = CachingFileSystem.cpp
4
    all: CachingFileSystem
6
8
    CachingFileSystem.o: CachingFileSystem.cpp
        g++ -std=c++11 -Wall -c -fmessage-length=0 -D_FILE_OFFSET_BITS=64 'pkg-config fuse --cflags' CachingFileSystem.cpp
9
10
    CachingFileSystem: CachingFileSystem.o
11
        g++ -Wall -std=c++11 -D_FILE_OFFSET_BITS=64 CachingFileSystem.o 'pkg-config fuse --libs' -o CachingFileSystem
12
13
    TAR=tar
14
    TARFLAGS=-cvf
15
    TARNAME=ex4.tar
16
    TARSRCS=$(LIBSRC) Makefile README
17
19
        $(TAR) $(TARFLAGS) $(TARNAME) $(TARSRCS)
20
21
    clean:
22
        (RM) CachingFileSystem.o CachingFileSystem ** *core
23
24
25
    .PHONY: CachingFileSystem CachingFileSystem.o
```