

Contents

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

1 Basic Test Results

```
1  g++ -std=c++11 -Wall -c -fmessage-length=0 -D_FILE_OFFSET_BITS=64 'pkg-config fuse --cflags' CachingFileSystem.cpp
2  g++ -Wall -std=c++11 -D_FILE_OFFSET_BITS=64 CachingFileSystem.o 'pkg-config fuse --libs' -o CachingFileSystem
3  rm -f CachingFileSystem.o CachingFileSystem *~ *core
4  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 Ex: 4
4
5 FILES:
6 README
7 CachingFileSystem.cpp -- a file that implements the Caching File System
8 makefile
9
10 REMARKS:
11 Most of the functins are implement same as the bbfs example and are very simple
12 so no need to expplain them.
13 We will explain how the read work
14 for the cache we have struct containing general information as block nmber,
15 directories path ect. nd also contain a vector Blocks.
16 The Blocks are second struct, contain information for single block.
17 This information is the path for the file the data took from, the number of the
18 block in this file, a counter of use and f course the data itself.
19 Each time a file beign read, we are going block by block.
20 For each block we first look for an exists block. If we can't find one, we
21 "activate" another block. If all the block are correntlly in use we search for
22 the least frequancy block, and replace it with the new block.
23 Then the function copy the data to the buffer and goes to the next rblock if
24 needed.
25
26
27 ANSWERS:
28
29 1 by default, heap memory is saved on RAM and so accessing it is much faster
30 than accessing the disk. *but* the heap is a virtual memory section and can
31 somtimes be mapped into disk - no garantuee that our heap-cache would really
32 be saved on RAM, so if it is actually saved on disk, we would get no better
33 performances.
34 2. no implementation is absolutely better: in Array implementation, reading
35 a specifc block is easier since we have random access to it, but when we
36 have to delete a block it would take  $O(n)$  time to find the ideal block to
37 delete, so this implementation is better when you know that your program would
38 read the same files (=blocks) a lot of times and would seldom read new files.
39 In the other hand, the list implementation can easily delete the less used
40 block but cannot access spesific blocks efficiently, so it should be preffered
41 when we most of the time read new information and seldom read the same data
42 several times.,
43 3. the difference is that paging is used for *memory access*. memory access
44 treated in user-level and so happens all time, so updating the data structure
45 each time the memory is accessed (by user or so) is not realistic (very
46 expensive). but reading a block from disk can be treated only by kernel - it
47 requires a system call, takes much more time and so happens much seldom, so in
48 this case updating the data structure in each access to a filr block is
49 realistic.
50 4. LRU better: if we are treating each file after reading it, and only when
51 finishing the work with the current file we continue to read the next one.
52 In this case, deleting the less recently block would delete a block from the
53 oldest file imported, which most probably is not needed antmore.
54 LFU better: if we are treating files with different information to be handled,
55 where some of the information is really important and have to be used a lot of
56 time, where somw of the other files contain unimportant information that after
57 reading once is not needed. in that case we want to delete the less important
58 information, which is most likely the information we have read the less number
59 of times.
```

60 both strategies are bad: when we are a file server that only provides file
61 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
66 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

```
1  /*
2   * CachingFileSystem.cpp
3   *
4   * Created on: 15 April 2015
5   * Author: Netanel Zakay, HUJI, 67808 (Operating Systems 2014-2015).
6   */
7
8  #define FUSE_USE_VERSION 26
9
10 #include <fuse.h>
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>
21 #include <iostream>
22 #include <limits.h>
23 #include <stdio.h>
24 #include <stdlib.h>
25 #include <map>
26 #include <dirent.h>
27 #include <cassert>
28 #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
35 struct fuse_operations caching_oper;
36
37 ofstream logfile;
38 time_t timer;
39
40 class MainError: public exception
41 {
42     virtual const char* what() const throw ()
43     {
44         return "error in main";
45     }
46 };
47
48 class SystemError: public exception
49 {
50     virtual const char* what() const throw ()
51     {
52         return strerror(errno);
53     }
54 };
55
56 typedef struct Block
57 {
58     int num_of_accesses;
59     char* data;
```

```

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

```

```

128     unsigned int occupied_blocks;
129     Cache(char* argv[]);
130     ~Cache()
131     {
132         vector<Block*>::iterator it;
133         for (it = cache.begin(); it != cache.end(); it++)
134         {
135             if ((*it) != nullptr)
136             {
137                 delete(*it);
138             }
139         }
140         free(rootdir);
141         free(mountdir);
142     }
143     Block * find_block(string path, const int block_num);
144     Block * assign_new_block();
145 } Cache;
146
147 Cache::Cache(char* argv[])
148 {
149     struct stat buffer;
150     rootdir = realpath(argv[1], NULL);
151     if ((rootdir == NULL) || stat(rootdir, &buffer) == -1)
152     {
153         throw MainError();
154     }
155     if (S_ISDIR(buffer.st_mode) == false)
156     {
157         throw MainError();
158     }
159     mountdir = realpath(argv[2], NULL);
160     if ((mountdir == NULL) || stat(mountdir, &buffer) == -1)
161     {
162         throw MainError();
163     }
164     if (S_ISDIR(buffer.st_mode) == false)
165     {
166         throw MainError();
167     }
168     if ((num_of_blocks = atoi(argv[3])) <= 0)
169     {
170         throw MainError();
171     }
172     if ((block_size = atoi(argv[4])) <= 0)
173     {
174         throw MainError();
175     }
176     cache.resize(num_of_blocks, nullptr);
177     vector<Block*>::iterator it;
178     for (it = cache.begin(); it != cache.end(); it++)
179     {
180         (*it) = new Block(block_size);
181     }
182     occupied_blocks = 0;
183 }
184
185 /*
186  * searching for exists block
187  * 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 {
192     vector<Block*>::iterator it;
193     for (it = cache.begin(); it != cache.end(); it++)
194     {
195

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

536     {
537         return -errno;
538     }
539     return retstat;
540 }
541
542 /** Open directory
543  *
544  * This method should check if the open operation is permitted for
545  * this directory
546  *
547  * Introduced in version 2.3
548  */
549 int caching_opendir(const char *path, struct fuse_file_info *fi)
550 {
551     SET_TIMER
552     logfile << timer << " opendir\n";
553
554     DIR *dp;
555     int retstat = 0;
556     string real_path = translate_path(path);
557
558     dp = opendir(real_path.c_str());
559     if (dp == NULL)
560     {
561         return -errno;
562     }
563
564     fi->fh = (intptr_t) dp;
565
566     return retstat;
567 }
568
569 /** Read directory
570  *
571  * This supersedes the old getdir() interface. New applications
572  * should use this.
573  *
574  * The readdir implementation ignores the offset parameter, and
575  * passes zero to the filler function's offset. The filler
576  * function will not return '1' (unless an error happens), so the
577  * whole directory is read in a single readdir operation. This
578  * works just like the old getdir() method.
579  *
580  * Introduced in version 2.3
581  */
582 int caching_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
583                    off_t offset, struct fuse_file_info *fi)
584 {
585     SET_TIMER;
586     logfile << timer << " readdir\n";
587
588     int retstat = 0;
589     DIR *dp;
590     struct dirent *de;
591
592     dp = (DIR *) (uintptr_t) fi->fh;
593     int prev_errno = errno;
594
595     while ((de = readdir(dp)) != NULL)
596     {
597         if ((strcmp(de->d_name, cache_data->logfile_name.c_str())) != 0)
598         {
599             if ((filler(buf, de->d_name, NULL, 0) != 0))
600             {
601                 return -ENOMEM;
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  *
617  * Introduced in version 2.3
618  */
619 int caching_releasedir(const char *path, struct fuse_file_info *fi)
620 {
621     SET_TIMER;
622     logfile << timer << " releasedir\n";
623
624     int retstat = 0;
625
626     if (closedir((DIR *) (uintptr_t) fi->fh) == -1)
627     {
628         return -errno;
629     }
630
631     return retstat;
632 }
633
634 /** Rename a file */
635 int caching_rename(const char *path, const char *newpath)
636 {
637     SET_TIMER;
638     logfile << timer << " rename\n";
639
640     int retstat = 0;
641     string new_path = translate_path(newpath);
642     string real_path = translate_path(path);
643     if (real_path == "")
644     {
645         return -ENONET;
646     }
647     if (new_path == "")
648     {
649         return -EINVAL;
650     }
651     retstat = rename(real_path.c_str(), new_path.c_str());
652
653     if (retstat != 0)
654     {
655         return -errno;
656     }
657
658     vector<Block*>::iterator it;
659     for (it = cache_data->cache.begin(); it != cache_data->cache.end(); it++)
660     {
661         if ((*it)->file_path.compare(path) == 0)
662         {
663             (*it)->file_path = new_path;
664         }
665     }
666     return retstat;
667 }
668
669 /**
670  * Initialize filesystem
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
674  * destroy() method.
675  *
676  * Introduced in version 2.3
677  * Changed in version 2.6
678  */
679 void *caching_init(struct fuse_conn_info *conn) {
680     logfile << time(nullptr) << " init\n";
681     return cache_data;
682 }
683
684 /**
685  * Clean up filesystem
686  *
687  * Called on filesystem exit.
688  *
689  * Introduced in version 2.3
690  */
691 void caching_destroy(void *userdata) {
692     logfile << time(nullptr) << " destroy\n";
693
694     logfile.close();
695     delete(cache_data);
696 }
697
698 /**
699  * Ioctl from the FUSE sepc:
700  * flags will have FUSE_IOCTL_COMPAT set for 32bit ioctls in
701  * 64bit environment. The size and direction of data is
702  * determined by _IOC_*() decoding of cmd. For _IOC_NONE,
703  * data will be NULL, for _IOC_WRITE data is out area, for
704  * _IOC_READ in area and if both are set in/out area. In all
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  */
711 int caching_ioctl(const char *, int cmd, void *arg, struct fuse_file_info *,
712                  unsigned int flags, void *data)
713 {
714     SET_TIMER;
715     logfile << timer << " ioctl\n";
716
717     for (vector<Block*>::iterator it = cache_data->cache.begin();
718          it != cache_data->cache.end(); ++it)
719     {
720         if ((*it)->block_num != -1)
721         {
722             logfile << get_virtual_path((*it).file_path) << " " << (*it).block_num
723                 << " " << (*it).num_of_accesses << endl;
724         }
725     }
726     return 0;
727 }
728
729 // Initialise the operations.
730 // You are not supposed to change this function.
731 void init_caching_oper() {
732
733     caching_oper.getattr = caching_getattr;
734     caching_oper.access = caching_access;
735     caching_oper.open = caching_open;
736     caching_oper.read = caching_read;
737     caching_oper.flush = caching_flush;
738     caching_oper.release = caching_release;
739     caching_oper.opendir = caching_opendir;

```

```

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

```


4 Makefile

```
1 CC = g++ -Wall
2 FLAG = -std=c++11 -D_FILE_OFFSET_BITS=64
3 FUSE = `pkg-config fuse --cflags --libs`
4 LIBSRC = CachingFileSystem.cpp
5
6 all: CachingFileSystem
7
8 CachingFileSystem.o: CachingFileSystem.cpp
9     g++ -std=c++11 -Wall -c -fmessage-length=0 -D_FILE_OFFSET_BITS=64 `pkg-config fuse --cflags` CachingFileSystem.cpp
10
11 CachingFileSystem: CachingFileSystem.o
12     g++ -Wall -std=c++11 -D_FILE_OFFSET_BITS=64 CachingFileSystem.o `pkg-config fuse --libs` -o CachingFileSystem
13
14 TAR=tar
15 TARFLAGS=-cvf
16 TARNAME=ex4.tar
17 TARSRC=$(LIBSRC) Makefile README
18
19 tar:
20     $(TAR) $(TARFLAGS) $(TARNAME) $(TARSRC)
21
22 clean:
23     $(RM) CachingFileSystem.o CachingFileSystem *~ *core
24
25
26 .PHONY: CachingFileSystem CachingFileSystem.o
```