

Contents

1	README	2
2	Block.h	4
3	Block.cpp	5
4	Cache.h	6
5	Cache.cpp	8
6	CachingFileSystem.cpp	12
7	LogFile.h	18
8	LogFile.cpp	19
9	Makefile	20

1 README

```
1  itamakatz, liavst2
2  Itamar Katz (555792977) , Liav Steinberg (203630090)
3  EX: 4
4
5  =====
6  =                FILES                =
7  =====
8
9  - README - this file.
10 - Makefile
11 - CachingFileSystem.cpp
12 - Cache.cpp
13 - LogFile.cpp
14 - Cache.h
15 - LogFile.h
16 - Block.h
17 - Block.cpp
18
19
20 =====
21 =                DESIGN                =
22 =====
23
24 Our Cache is an independent unit in our implementation.
25 In CachingFileSystem.cpp the functions that call the cache
26 are read and ioctl only.
27 Read calls the "collect" function which retrieves the wanted
28 information from the cache. Given the offset and the size
29 to read, collect calculates the starting and ending block
30 indexes to search in the cache. If the block doesn't exist,
31 collect calls the private function _cache_add to add this block
32 to the cache. If the block exists, collect calls _cache_replace
33 to place the current block on top of the Cache container.
34 All these blocks are placed in array, ordered by their
35 sequential access. Then we start to copy the content of the
36 blocks to "buf" using memcpy, and the wanted offset and size.
37
38 In ioctl, the cache simply summons the logfile on each block
39 in its repository. The logfile writes the block information
40 down.
41
42 =====
43 =                ANSWERS                =
44 =====
45
46
47 1. We know that accessing the disk is way slower than accessing
48    the memory because the memory is physically closer to the
49    cpu. In this exercise we store our information on the heap.
50    If the heap is located in the RAM, it will be more efficient to
51    read from it. But if the heap is located elsewhere, then our
52    cache advantage will be lost. Also, searching in the cache may
53    take time too if we are handling large amount of filele
54
55
56 2. In class, we saw there is a trade-off between sophisticated
57    algorithm and a fast cache management. We require minimum
58    time to handle a buffer cache managed by the OS because it
59    is constantly changing.
```

```

60
61
62 3. LRU is better:
63     files -> test1, test2, test3
64     read ->
65     test1,
66     test1,
67     test2,
68     test2,
69     test3,
70     test3,
71
72     LFU is better:
73     Files -> test 1, test 2, test 3, test 4.
74     read ->
75     test 1
76     test 2
77     test 3
78     test 1
79     test 1
80     test 1
81     test 1
82     test 1
83     test 2
84     test 3
85     test 4
86
87
88     Niether is better:
89     Files -> test 1, test 2, test 3, test 4.
90
91     Read:
92     test 1
93     test 2
94     test 3
95     test 4
96     test 1
97     test 2
98     test 3
99     test 4
100
101
102
103 4. Because there can be an intensive use in the same
104     segment for a short period of time(This is called
105     the locality principle as we saw) and then wont be
106     in use anymore. That can happen to a block in the
107     new section, increasing the block's reference count
108     wont achieve this goal.
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124

```

2 Block.h

```
1  #ifndef EX4_BLOCK_H
2  #define EX4_BLOCK_H
3
4  #include <stdio.h>
5  #include <string.h>
6  #include <stdlib.h>
7  #include <string>
8
9  #define REF_COUNT_INIT 1
10
11
12  class Block {
13  public:
14      int _refCount;
15      std::string* _path;
16      int _index;
17      char* _content;
18
19      Block(const char* path, int index, size_t blksize);
20
21      ~Block();
22
23      static bool comparator(const Block* , const Block*);
24  };
25
26
27
28  #endif //EX4_BLOCK_H
```

3 Block.cpp

```
1  #include "Block.h"
2
3
4
5  Block::Block(const char* path, int index, size_t blksize):
6      _refCount(REF_COUNT_INIT),
7      _index(index){
8      _path = new std::string(path);
9      _content = (char*)aligned_alloc(blksize, blksize);
10     memset(_content, 0, blksize);
11 }
12
13
14 Block::~Block(){
15     delete(_path);
16     free(_content);
17     _path = NULL;
18     _content = NULL;
19 }
20
21
22 bool Block::comparator(const Block* d1, const Block* d2) {
23     return d1->_refCount > d2->_refCount;
24 }
```

4 Cache.h

```
1  #ifndef EX4_CACHE_H
2  #define EX4_CACHE_H
3
4  #include <stdio>
5  #include <sys/stat.h>
6  #include <sys/types.h>
7  #include <map>
8  #include <queue>
9  #include <fcntl.h>
10 #include <math.h>
11 #include <unistd.h>
12 #include <string.h>
13 #include <algorithm>
14
15 #include "LogFile.h"
16 #include "Block.h"
17
18 #define FALIURE 1
19 #define SUCCESS 0
20
21
22 class Cache{
23 public:
24
25     Cache(const double num_args[]);
26
27     ~Cache();
28
29     int collect(const char *path, char *buf, size_t size,
30                off_t offset, struct fuse_file_info *fi);
31
32     int ioctl_info(LogFile** lf, std::string rootdir);
33
34     void inner_rename(std::string* path, std::string* newpath);
35
36 private:
37     blksize_t _blksize;
38     size_t _TotalSize;
39     size_t _OldSize;
40     size_t _MidSize;
41     size_t _NewSize;
42
43     std::deque<Block*> _Cache;
44
45     void _cache_clean();
46
47     char* _cache_replace(const char* path, int index);
48
49     char* _cache_add(const char* path, struct fuse_file_info *fi, int index);
50
51     void _cache_evict();
52
53     void _sort_old(void);
54
55     bool _find_existing_block(const char *path, int index);
56
57     int _fsize(const char *path);
58
59 };
```

```
60
61  #endif //EX4_CACHE_H
```

5 Cache.cpp

```
1  #include <iostream>
2  #include "Cache.h"
3
4
5
6  Cache::~Cache(){
7      _cache_clean();
8  }
9
10
11
12  Cache::Cache(const double num_args[]){
13      _TotalSize = (size_t) num_args[0];
14      _NewSize = (size_t) std::floor((double)_TotalSize * num_args[1]);
15      _OldSize = (size_t) std::floor((double)_TotalSize * num_args[2]);
16      if (!_NewSize || !_OldSize){
17          throw FALIURE;
18      }
19      _MidSize = _TotalSize - _NewSize - _OldSize;
20      struct stat fi;
21      stat("/tmp", &fi);
22      _blksize = fi.st_blksize;
23  }
24
25
26
27  int Cache::collect(const char *path, char *buf, size_t size,
28                    off_t offset, struct fuse_file_info *fi){
29
30      int total_size = _fsize(path);
31      if (total_size < 0){
32          return total_size;
33      }
34
35      if (size > (size_t) (total_size - offset)){ // adjust the size
36          size = (size_t) (total_size - offset);
37      }
38
39      if (total_size < offset || size <= 0){
40          return SUCCESS;
41      }
42
43      // check if its in the cache
44      int start_index = (int) std::floor((double)offset / (double)_blksize);
45      int end_index = (int) std::floor
46          (((double)size + (double)offset) - 1) / (double)_blksize;
47      int num_of_blocks = end_index - start_index + 1;
48
49      char* ret[num_of_blocks];
50      for (int index = start_index, i = 0; i < num_of_blocks; index++, i++){
51
52          if (!_find_existing_block(path, index)) {
53              try {
54                  ret[i] = _cache_add(path, fi, index);
55              } catch (std::string& err){
56                  return -errno;
57              } catch (std::bad_alloc& err){
58                  return -FALIURE;
59              }

```



```

60         continue;
61     }
62     ret[i] = _cache_replace(path, index);
63 }
64
65 if (num_of_blocks == 1){
66     memcpy(buf, ret[0] + offset - start_index*_blksize, size);
67     return (int)size;
68 }
69
70 size_t buf_offset = (size_t) ((start_index + 1)*_blksize - offset);
71 memcpy(buf, ret[0] + offset - start_index*_blksize, buf_offset);
72
73
74 for (int i = 1; i < num_of_blocks - 1; i++){
75     memcpy(buf + buf_offset, ret[i], (size_t) _blksize);
76     buf_offset += (size_t) (_blksize);
77 }
78
79 memcpy(buf + buf_offset, ret[num_of_blocks - 1], size - buf_offset);
80 return (int)size;
81 }
82
83
84 bool Cache::_find_existing_block(const char *path, int index){
85     for (Block* block: _Cache){
86         if (block->index == index && !block->_path->compare(path)){
87             return true;
88         }
89     }
90     return false;
91 }
92
93
94
95 int Cache::ioctl_info(LogFile** lf, std::string rootdir){
96     for (size_t i = 0; i < _Cache.size(); i++){
97         std::string file(*(_Cache[i]->_path));
98         size_t pos = file.find(rootdir);
99         std::string path = file.substr(pos + rootdir.length() + 1);
100         (*lf)->cache_ioctl(path, _Cache[i]->_index, _Cache[i]->_refCount);
101     }
102     return SUCCESS;
103 }
104
105
106
107 void Cache::inner_rename(std::string* path, std::string* newpath) {
108
109     for (Block* item: _Cache){
110         if (path->length() > item->_path->length()){
111             continue;
112         }
113         std::string prefix_path = item->_path->substr(0, path->length());
114         if (!prefix_path.compare(*path)) {
115             item->_path->replace(0, path->length(), *newpath);
116         }
117     }
118     return;
119 }
120
121
122
123 void Cache::_cache_clean(){
124     for (Block* item: _Cache){
125         delete(item);
126     }
127     _Cache.clear();

```

```

128     return;
129 }
130
131
132
133 char* Cache::_cache_replace(const char* path, int index){
134
135     auto iter = _Cache.begin();
136     for (size_t i = 1; i <= _Cache.size(); i++, iter++){
137         if(!(*iter)->_path->compare(path) && (*iter)->_index == index) {
138             Block* block = std::move(*iter);
139             _Cache.erase(iter);
140             _Cache.push_front(block);
141
142             if (i > _NewSize){ // was not in new section
143                 block->_refCount++;
144                 if(i > _NewSize + _MidSize){ // was in old section
145                     _sort_old();
146                 }
147             }
148
149             return block->_content;
150         }
151     }
152
153     return NULL;
154 }
155
156
157
158 char* Cache::_cache_add(const char* path, struct fuse_file_info*fi, int index){
159
160     Block* newBlock = new Block(path, index, (size_t) _blksize);
161     ssize_t amount = pread((int)fi->fh, newBlock->_content,
162                           (size_t) _blksize, (off_t)(index*_blksize));
163     if (amount < 0){
164         throw "error";
165     }
166
167     _Cache.push_front(newBlock);
168     if (_Cache.size() == _TotalSize + 1){
169         _cache_evict();
170     }
171
172     return newBlock->_content;
173 }
174
175
176
177 void Cache::_cache_evict(){
178     Block* evicted = std::move(_Cache.back());
179     _Cache.pop_back();
180     delete(evicted);
181     _sort_old();
182 }
183
184
185
186 void Cache::_sort_old(void){
187     auto iter = _Cache.begin();
188     std::advance(iter, _TotalSize - _OldSize);
189     std::stable_sort(iter, _Cache.end(), Block::comparator);
190 }
191
192
193
194 int Cache::_fsize(const char *file){
195     struct stat st;

```

```
196     return stat(file, &st) ? (-errno): (int)st.st_size;
197 }
```

6 CachingFileSystem.cpp

```
1  /*
2   * CachingFileSystem.cpp
3   *
4   * Author: Netanel Zakay, HUJI, 67808 (Operating Systems 2015-2016).
5   */
6
7  #define FUSE_USE_VERSION 26
8  #define FUSE_ARG_NUM 3
9  #define MINIMUM_ARGS_NUM 6
10 #define SUCCESS 0
11 #define USAGE "Usage: CachingFileSystem rootdir mountdir\
12     numberOfBlocks fold fNew"
13 #define DIR_ERROR "System Error: Can't evaluate given directories"
14 #define ALLOCATION_FAIL "System Error: Allocation Failure"
15
16 #include <errno.h>
17 #include <cstring>
18 #include <iostream>
19 #include <algorithm>
20 #include <unistd.h>
21 #include <dirent.h>
22
23 #include "Cache.h"
24
25 struct fuse_operations g_caching_oper;
26 static Cache* cache;
27 static char rootdir[PATH_MAX];
28 static LogFile* lf;
29
30
31 bool LogFileAccess(const char* path){
32     return !strcmp(path, LOG_FILE_PATH);
33 }
34
35
36 /**
37  * returns the absolute path of the file
38  */
39 char* absolute_path(const char *path){
40     char g_abs_path[PATH_MAX];
41     strcpy(g_abs_path, rootdir);
42     return strncat(g_abs_path, path, PATH_MAX);
43 }
44
45
46 /**
47  * implements getattr
48  */
49 int caching_getattr(const char *path, struct stat *statbuf){
50
51     lf->command("getattr");
52     if (LogFileAccess(path)){ // trying to reach our logfile
53         return -ENOENT;
54     }
55
56     int ret = lstat(absolute_path(path), statbuf);
57     return (ret < 0) ? (-errno): ret;
58 }
59
```

```

60
61 /**
62  * implements fgetattr
63  */
64 int caching_fgetattr(const char *path, struct stat *statbuf,
65                     struct fuse_file_info *fi){
66     lf->command("fgetattr");
67     if (LogFileAccess(path)){ // trying to reach our logfile
68         return -ENOENT;
69     }
70
71     if (!strcmp(path, "/")){
72         return caching_getattr(path, statbuf);
73     }
74
75     int ret = fstat(fi->fh, statbuf);
76     return (ret < 0) ? (-errno): ret;
77 }
78
79
80 /**
81  * implements access
82  */
83 int caching_access(const char *path, int mask){
84
85     lf->command("access");
86     if (LogFileAccess(path)){ // trying to reach our logfile
87         return -ENOENT;
88     }
89
90     int ret = access(absolute_path(path), mask);
91     return (ret < 0) ? (-errno): ret;
92 }
93
94
95 /**
96  * implements open
97  */
98 int caching_open(const char *path, struct fuse_file_info *fi){
99
100     lf->command("open");
101     if (LogFileAccess(path)){ // trying to reach our logfile
102         return -ENOENT;
103     }
104
105     fi->direct_io = 1;
106     if ((fi->flags & 3) != O_RDONLY) {
107         return -EACCES;
108     }
109
110     int fd = open(absolute_path(path), O_RDONLY|O_DIRECT|O_SYNC);
111     if (fd < 0){
112         return -ENOENT;
113     }
114
115     fi->fh = fd;
116
117     return SUCCESS;
118 }
119
120
121
122 /**
123  * implements read
124  */
125 int caching_read(const char *path, char *buf, size_t size,
126                 off_t offset, struct fuse_file_info *fi){
127

```

```

128     lf->command("read");
129     if (LogFileAccess(path)){ // trying to reach our logfile
130         return -ENOENT;
131     }
132
133     int ret = cache->collect(absolute_path(path), buf, size, offset, fi);
134     return ret;
135 }
136
137
138 /**
139  * implements flush
140  */
141 int caching_flush(const char *path, struct fuse_file_info*){
142
143     lf->command("flush");
144     if (LogFileAccess(path)){ // trying to reach our logfile
145         return -ENOENT;
146     }
147
148     return SUCCESS;
149 }
150
151
152 /**
153  * implements release
154  */
155 int caching_release(const char *path, struct fuse_file_info *fi){
156
157     lf->command("release");
158     if (LogFileAccess(path)){ // trying to reach our logfile
159         return -ENOENT;
160     }
161
162     int ret = close((int)fi->fh);
163     return (ret < 0) ? (-errno): ret;
164 }
165
166
167 /**
168  * implements opendir
169  */
170 int caching_opendir(const char *path, struct fuse_file_info *fi){
171
172     lf->command("opendir");
173     if (LogFileAccess(path)){ // trying to reach our logfile
174         return -ENOENT;
175     }
176
177     DIR* dir = opendir(absolute_path(path));
178
179     if (dir == NULL){
180         return -errno;
181     }
182
183     fi->fh = (intptr_t) dir;
184     return SUCCESS;
185 }
186
187
188 /**
189  * implements readdir
190  */
191 int caching_readdir(const char* path, void *buf,
192                    fuse_fill_dir_t filler, off_t ,struct fuse_file_info *fi){
193
194     lf->command("readdir");
195     if (LogFileAccess(path)){ // trying to reach our logfile

```

```

196         return -ENOENT;
197     }
198
199     DIR* dp = (DIR*) (uintptr_t) fi->fh;
200     struct dirent *de;
201     if ((de = readdir(dp)) == 0){
202         return -errno;
203     }
204
205     do
206     {
207         if (LogFileAccess(de->d_name)){
208             continue;
209         }
210         if (filler(buf, de->d_name, NULL, 0)){
211             return -ENOMEM;
212         }
213     } while ((de = readdir(dp)) != NULL);
214
215     return SUCCESS;
216 }
217
218 /**
219  * implements releasedir
220  */
221 int caching_releasedir(const char* path, struct fuse_file_info *fi){
222
223     lf->command("releasedir");
224     if (LogFileAccess(path)){ // trying to reach our logfile
225         return -ENOENT;
226     }
227
228     int ret = closedir((DIR*) (uintptr_t) fi->fh);
229     return (ret < 0) ? (-errno): ret;
230 }
231
232 /**
233  * implements rename
234  */
235 int caching_rename(const char *path, const char *newpath){
236
237     lf->command("rename");
238     if (LogFileAccess(path)){ // trying to reach our logfile
239         return -ENOENT;
240     }
241
242
243     std::string oldPath(absolute_path(path));
244     std::string newPath(absolute_path(newpath));
245
246     int ret = rename(oldPath.c_str(), newPath.c_str());
247     if (ret < 0){
248         return -errno;
249     }
250
251     // also change in the cache
252     cache->inner_rename(&oldPath, &newPath);
253     return ret;
254 }
255
256 /**
257  * implements init
258  */
259 void* caching_init(struct fuse_conn_info*){
260
261     lf->command("init");
262     return NULL;
263 }

```

```

264
265 /**
266  * implements destroy
267  */
268 void caching_destroy(void*){
269
270     lf->command("destroy");
271     delete(cache);
272     delete(lf);
273     return;
274 }
275
276 /**
277  * implements ioctl
278  */
279 int caching_ioctl (const char*, int, void*,
280                    struct fuse_file_info *, unsigned int, void*){
281     lf->command("ioctl");
282     cache->iocctl_info(&lf, rootdir);
283     return SUCCESS;
284 }
285
286
287
288 void init_caching_oper() {
289
290     g_caching_oper.getattr = caching_getattr;
291     g_caching_oper.access = caching_access;
292     g_caching_oper.open = caching_open;
293     g_caching_oper.read = caching_read;
294     g_caching_oper.flush = caching_flush;
295     g_caching_oper.release = caching_release;
296     g_caching_oper.opendir = caching_opendir;
297     g_caching_oper.readdir = caching_readdir;
298     g_caching_oper.releasedir = caching_releasedir;
299     g_caching_oper.rename = caching_rename;
300     g_caching_oper.init = caching_init;
301     g_caching_oper.destroy = caching_destroy;
302     g_caching_oper.ioctl = caching_ioctl;
303     g_caching_oper.fgetattr = caching_fgetattr;
304
305
306     g_caching_oper.readlink = NULL;
307     g_caching_oper.getdir = NULL;
308     g_caching_oper.mknod = NULL;
309     g_caching_oper.mkdir = NULL;
310     g_caching_oper.unlink = NULL;
311     g_caching_oper.rmdir = NULL;
312     g_caching_oper.symlink = NULL;
313     g_caching_oper.link = NULL;
314     g_caching_oper.chmod = NULL;
315     g_caching_oper.chown = NULL;
316     g_caching_oper.truncate = NULL;
317     g_caching_oper.utime = NULL;
318     g_caching_oper.write = NULL;
319     g_caching_oper.statfs = NULL;
320     g_caching_oper.fsync = NULL;
321     g_caching_oper.setxattr = NULL;
322     g_caching_oper.getxattr = NULL;
323     g_caching_oper.listxattr = NULL;
324     g_caching_oper.removexattr = NULL;
325     g_caching_oper.fsyncdir = NULL;
326     g_caching_oper.create = NULL;
327     g_caching_oper.ftruncate = NULL;
328 }
329
330
331 /**

```



```

332  * checks main args
333  */
334  void check_args(const char* mountdir, const double num_args[]){
335
336      if (*std::min_element(num_args, num_args+2) <= 0 ||
337          num_args[1]+num_args[2] > 1){
338          std::cout << USAGE << std::endl;
339          exit(FALIURE);
340      }
341
342      DIR* d1 = opendir(rootdir);
343      DIR* d2 = opendir(mountdir);
344
345      if (errno == ENOENT){ // one or both directories don't exist
346          std::cout << USAGE << std::endl;
347          exit(FALIURE);
348      }
349
350      closedir(d1);
351      closedir(d2);
352      return;
353  }
354
355
356
357  int main(int argc, char* argv[]){
358
359      if (argc < MINIMUM_ARGS_NUM) {
360          std::cout << USAGE << std::endl;
361          exit(FALIURE);
362      }
363
364      char mountdir[PATH_MAX];
365      if (realpath(argv[1], rootdir) == NULL ||
366          realpath(argv[2], mountdir) == NULL){
367          std::cerr << DIR_ERROR << std::endl;
368          exit(FALIURE);
369      }
370
371      const double num_args[] =
372          {(double) atoi(argv[3]), atof(argv[4]), atof(argv[5])};
373      check_args(mountdir, num_args);
374
375      try {
376          cache = new Cache(num_args);
377      } catch (int err){
378          std::cout << USAGE << std::endl;
379          exit(FALIURE);
380      } catch (std::bad_alloc& err){
381          std::cerr << ALLOCATION_FAIL << std::endl;
382          exit(FALIURE);
383      }
384
385      try {
386          lf = new LogFile(rootdir);
387      } catch (std::bad_alloc& err){
388          delete(cache);
389          std::cerr << ALLOCATION_FAIL << std::endl;
390          exit(FALIURE);
391      }
392
393      char* fuse_args[] = {argv[0], mountdir, (char*)" -s"};
394      int number_of_args = FUSE_ARG_NUM;
395
396      init_caching_oper();
397      int fuse_stat =fuse_main(number_of_args, fuse_args, &g_caching_oper,NULL);
398      return fuse_stat;
399  }

```

7 LogFile.h

```
1  #ifndef EX4_LOGFILE_H
2  #define EX4_LOGFILE_H
3
4
5  #include <iosfwd>
6  #include <fstream>
7  #include <fuse.h>
8
9  #define LOG_FILE_PATH "/.filesystem.log"
10
11
12  class LogFile{
13  private:
14      std::ofstream __lfile__;
15
16  public:
17      LogFile(std::string);
18
19      ~LogFile();
20
21      void command(std::string);
22
23      void cache_ioctl(std::string, int, int);
24  };
25
26
27  #endif //EX4_LOGFILE_H
```

8 LogFile.cpp

```
1  #include "LogFile.h"
2
3
4  LogFile::LogFile(std::string fullpath) {
5      fullpath.append(LOG_FILE_PATH);
6      __lfile__.open(fullpath, std::ios_base::app);
7  }
8
9
10 LogFile::~LogFile(){
11     __lfile__.close();
12 }
13
14 void LogFile::command(std::string syscall) {
15     __lfile__ << time(NULL) << " " << syscall << std::endl;
16 }
17
18 void LogFile::cache_ioctl(std::string path, int index, int ref){
19     __lfile__ << path << " " << (index + 1) << " " << ref << std::endl;
20 }
```

9 Makefile

```
1  CFLAGS = -g -std=c++11 -DNDEBUG -Wall -D_FILE_OFFSET_BITS=64
2  FUSE_FLAGS = `pkg-config fuse --cflags --libs`
3
4  TAR_NAME = ex4.tar
5  SOURCES = CachingFileSystem.cpp LogFile.cpp Cache.cpp Block.cpp
6  HEADERS = LogFile.h Cache.h Block.h
7  OBJS = $(SOURCES:.cpp=.o)
8  EXTRA_FILES = README Makefile
9  TAR_FILES = $(SOURCES) $(HEADERS) $(EXTRA_FILES)
10
11 EXEC = CachingFileSystem
12
13 .DEFAULT_GOAL = $(EXEC)
14
15
16 all: $(EXEC) tar
17
18 $(EXEC): $(OBJS)
19     $(CXX) $(FUSE_FLAGS) $^ -o $@
20
21
22 %.o: %.cpp $(HEADERS)
23     $(CXX) -c $(CFLAGS) $<
24
25 tar:
26     tar -cvf $(TAR_NAME) $(TAR_FILES)
27
28
29 clean:
30     rm -rf $(TAR_NAME) $(OBJS) $(EXEC)
31
32
33 .PHONY:
34     all tar clean
```