Contents

1 README

```
matanmo, eyalc
Matan Modai (305097560), Eyal Cohen (301300372)
EX: 4

FILES:
CachingFileSystem.cpp - main caching file
CDE.h - CDE CLASS
CDE.cpp
CountChain.h - Count Chain Class
CountChain.cpp
LRUStack.h - LRU Stack class
LRUStack.cpp
Makefile
README - this file



REMARKS:

Design and implementation is exactly as explained in chapter 2.4 in the article:
"Data Cache Management Using Frequency-Based Replacement"


ANSWERS:

1.
==
Usually using the cache will be faster, but sometimes the OS saves some of the processs pages on
the disk and then when the process tries to retrieve the data, a page fault is encountered. In that
case, reading the data from the cache involves disk I/O and therefore the access speed is not
improved.

2.
==
When using simpler algorithms, the logic for managing the cache can be implemented on hardware
level, therefore leaving the CPU free to perform other tasks. However, when using more sophisticated
algorithms, the logic is performed at the CPU level, therefore making the pages management harder.

3.
==
When cache size is 3:
A,A,A,A,B,C,D,A,A,A,A,E,F,G,A,A,A,A - LFU is better than LRU because clearly A is used

A,A,A,A,A,B,C,D,E,B,C,D,E - LRU is better than LFU becase A is used at high frequency at first but
not at all afterwords.

A,B,C,D,A,B,C,D,A,B,C,D - MRU would be better than LRU and FRU

4.
==
Sometimes when using resources, a resource can have high locality frequency, meaning the resource
will be used a lot in a brief section of the program (e.g loops), but not very much used afterwords.
In those cases, we wouldnt want to give this resource too much significance because the frequency
is only relevant for a short section - high locality frequency.
When using the new section, we can have a better sense if the frequency of the resource is local.
Using this system, a high frequency count shows that the frequency does not refer only to a small
section, and therefore the probability of needing the resource in the future is higher than if it
the frequency was local.
```

2 CDE.h

```
#ifndef OS_EX4_CDE_H
#define OS_EX4_CDE_H

#include <string>
#include <cstdlib>
#include <stdlib.h>

using namespace std;

class CDE {
private:
    int _count;
    CDE * _countPrev;
    CDE * _countNext;
    CDE * _prev;
    CDE * _next;
    bool _isOld, _isNew;
    int _blockId;
    string _fileName;
    char * _blockData;
    size_t _dataSize;
public:
    CDE(int blockId, string fileName, size_t dataSize, char * blockData);
    ~CDE();

    char * getData() { return _blockData; };
    size_t getSize() { return _dataSize; };

    void setPrev(CDE * cde) { _prev = cde; };
    void setNext(CDE * cde) { _next = cde; };

    void setCountPrev(CDE * cde) { _countPrev = cde; };
    void setCountNext(CDE * cde) { _countNext = cde; };

    CDE * getPrev() { return _prev; };
    CDE * getNext() { return _next; };

    CDE * getCountPrev() { return _countPrev; };
    CDE * getCountNext() { return _countNext; };

    void increaseCount() { ++_count; };

    void setIsNew(bool b) { _isNew = b; };
    void setIsOld(bool b) { _isOld = b; };

    bool getIsOld() { return _isOld; };
    bool getIsNew() { return _isNew; };

    void setFileName(string fileName) { _fileName = fileName; };

    int getCount() { return _count; };

    int getBlockId() { return _blockId; };
    string getFileName() { return _fileName; };
};


#endif //OS_EX4_CDE_H
```

3 CDE.cpp

```
#include <cstring>
#include "CDE.h"
#include <iostream>

CDE::CDE(int blockId, string fileName, size_t dataSize,
          char * blockData) {
    _dataSize = dataSize;
    _fileName = fileName;
    _blockId = blockId;
    _blockData = (char *) malloc(sizeof(char) * dataSize);
    if(_blockData == NULL) {
        cout << "malloc ERROR" << endl;
    }
    memcpy(_blockData, blockData, dataSize);
    _isOld = false;
    _isNew = true;
    _count = 1;
    _countNext = nullptr;
    _countPrev = nullptr;
    _next = nullptr;
    _prev = nullptr;
};

CDE::~CDE() {
    free(_blockData);
};
```

4 CachingFileSystem.cpp

```cpp
/*
 * CachingFileSystem.cpp
 *
 *  Author: Netanel Zakay, HUJI, 67808  (Operating Systems 2015-2016).
 */

#define FUSE_USE_VERSION 26

#include <iostream>
#include <fuse.h>
#include <unistd.h>
#include <sstream>
#include <fstream>
#include <limits.h>
#include <errno.h>
#include <cstdlib>
#include <dirent.h>
#include <cstring>
#include <map>
#include <cmath>
#include <vector>

#include "CDE.h"
#include "LRUStack.h"
#include "CountChain.h"

using namespace std;

#define CMAX 3 // the maximum count list value
#define USAGE_ERROR "Usage: CachingFileSystem " \
                    "rootdir mountdir numberOfBlocks fOld fNew\n"
#define FILE_LOCATION "/.filesystem.log"

struct fuse_operations caching_oper;

struct {
    char* rootDir;
} typedef user_data;

static char* rootDir;
static char logPath[PATH_MAX];
static int blockSize;
static map<pair<string, int>, CDE*> cacheMap;
static LRUStack lru;
static CountChain countChain(CMAX);
static int numberOfBlocks;

std::ofstream logFile;
std::stringstream logBuffer;

/* Return the System block size*/
static int getBlockSize() {
    struct stat fi;
    stat("/tmp", &fi);
    return (int) fi.st_blksize;
}

static bool isLogFile(const char *path) {
    string logFile(".filesystem.log");
```

```
60      string str(path);
61      int pos = str.find(logFile);
62      if(pos == -1) {
63          return false;
64      }
65      return true;
66  }
67
68  /* Build absolute path from relative */
69  static void buildPath(char fpath[PATH_MAX], const char *path) {
70      strcpy(fpath, rootDir);
71      strncat(fpath, path, PATH_MAX);
72  }
73
74  /*
75   * Check that the given parameters are valid
76   */
77  bool isInputParamsValid(int argc, char* argv[]) {
78      // Usage error
79      bool usage_error = false;
80      if(argc != 6) {
81          usage_error = true;
82      } else {
83          struct stat sb;
84          // fetch params
85          char* rootdir = argv[1];
86          char* mountdir = argv[2];
87          int numberOfBlocks = atoi(argv[3]);
88          double fOld = atof(argv[4]);
89          double fNew = atof(argv[5]);
90
91          // check params are valid
92          if(fOld <= 0 || fOld >= 1 || fNew <= 0 ||
93                  fNew >= 1 || fOld + fNew > 1) {
94              usage_error = true;
95          } else if(numberOfBlocks < 0) {
96              usage_error = true;
97          } else if(stat(rootdir, &sb) != 0 || !S_ISDIR(sb.st_mode)) {
98              usage_error = true;
99          } else if(stat(mountdir, &sb) != 0 || !S_ISDIR(sb.st_mode)) {
100             usage_error = true;
101         }
102     }
103     if(usage_error) {
104         cout << USAGE_ERROR << endl;
105     }
106     return !usage_error;
107 }
108
109 /*
110  * Print system errors to cerr.
111  */
112 void sysError(std::string errFunc) {
113     std::cerr << "System Error: " <<
114     errFunc << " failed." << std::endl;
115     exit(1);
116 }
117
118 /**
119  * Write commands to log.
120  */
121 void writeToLog(string msg) {
122 //    logFile.open(logPath, std::ios_base::app);
123     if(logFile.fail()) {
124         sysError("open");
125     }
126
127     time_t t;
```

```
128     if((int) time(&t) < 0) {
129         sysError("time");
130     }
131 
132     logFile << t << " " << msg << "\n";
133 //    logFile.close();
134     if(logFile.fail()) {
135         sysError("close");
136     }
137 }
138 
139 
140 /**
141  * Remove block from cache
142  */
143 void removeFromCache() {
144     CDE * cde = countChain.getItemToRemove();
145     if(cde == nullptr) {
146        cde = lru.getTail();
147     }
148     lru.remove(cde);
149     countChain.remove(cde);
150     string fileName = cde->getFileName();
151     int blockId = cde->getBlockId();
152     cacheMap.erase({fileName, blockId});
153     delete cde;
154 }
155 
156 /*****************************************************************************/
157 
158 /** Get file attributes.
159  *
160  * Similar to stat().  The 'st_dev' and 'st_blksize' fields are
161  * ignored.  The 'st_ino' field is ignored except if the 'use_ino'
162  * mount option is given.
163  */
164 int caching_getattr(const char *path, struct stat *statbuf){
165     writeToLog("getattr");
166 
167     if(isLogFile(path)) {
168         return -ENOENT;
169     }
170 
171     char fpath[PATH_MAX];
172     buildPath(fpath, path);
173     int ret = stat(fpath, statbuf);
174     if(ret != 0) {
175         return -errno;
176     }
177     return ret;
178 }
179 
180 /**
181  * Get attributes from an open file
182  *
183  * This method is called instead of the getattr() method if the
184  * file information is available.
185  *
186  * Currently this is only called after the create() method if that
187  * is implemented (see above).  Later it may be called for
188  * invocations of fstat() too.
189  *
190  * Introduced in version 2.5
191  */
192 int caching_fgetattr(const char *path, struct stat *statbuf,
193                      struct fuse_file_info *fi){
194     writeToLog("fgetattr");
195 
```

```
196     if(isLogFile(path)) {
197         return -ENOENT;
198     }
199 
200     int ret = fstat((int) fi->fh, statbuf);
201     if(ret != 0) {
202         return -errno;
203     }
204     return ret;
205 }
206 
207 /**
208  * Check file access permissions
209  *
210  * This will be called for the access() system call.  If the
211  * 'default_permissions' mount option is given, this method is not
212  * called.
213  *
214  * This method is not called under Linux kernel versions 2.4.x
215  *
216  * Introduced in version 2.5
217  */
218 int caching_access(const char *path, int mask)
219 {
220     writeToLog("access");
221 
222     if(isLogFile(path)) {
223         return -ENOENT;
224     }
225 
226     char fpath[PATH_MAX];
227     buildPath(fpath, path);
228 
229     int ret = access(fpath, mask);
230     if(ret != 0) {
231         return -errno;
232     }
233     return ret;
234 }
235 
236 
237 /** File open operation
238  *
239  * No creation, or truncation flags (O_CREAT, O_EXCL, O_TRUNC)
240  * will be passed to open().  Open should check if the operation
241  * is permitted for the given flags.  Optionally open may also
242  * initialize an arbitrary filehandle (fh) in the fuse_file_info
243  * structure, which will be passed to all file operations.
244 
245  * pay attention that the max allowed path is PATH_MAX (in limits.h).
246  * if the path is longer, return error.
247 
248  * Changed in version 2.2
249  */
250 int caching_open(const char *path, struct fuse_file_info *fi){
251     writeToLog("open");
252 
253     if(isLogFile(path)) {
254         return -ENOENT;
255     }
256 
257     fi->direct_io = 1;
258     char fpath[PATH_MAX];
259     buildPath(fpath, path);
260     // if path too long
261     if(sizeof(*fpath) > PATH_MAX) {
262         return -ENAMETOOLONG;
263     }
```

```
264
265     // check the access is valid (read only)
266     if((fi->flags & 3) != O_RDONLY) {
267         return -EACCES;
268     }
269
270     int ret = open(fpath, (O_RDONLY | O_DIRECT | O_SYNC));
271     if(ret == -1) {
272         return -errno;
273     }
274     fi->fh = (uint64_t) ret;
275     return 0;
276 }
277
278
279 /** Read data from an open file
280  *
281  * Read should return exactly the number of bytes requested except
282  * on EOF or error. For example, if you receive size=100, offest=0,
283  * but the size of the file is 10, you will init only the first
284    ten bytes in the buff and return the number 10.
285
286    In order to read a file from the disk,
287    we strongly advise you to use "pread" rather than "read".
288    Pay attention, in pread the offset is valid as long it is
289    a multipication of the block size.
290    More specifically, pread returns 0 for negative offset
291    and an offset after the end of the file
292    (as long as the the rest of the requirements are fulfiiled).
293    You are suppose to preserve this behavior also in your implementation.
294
295  * Changed in version 2.2
296  */
297 int caching_read(const char *path, char *buf, size_t size,
298                  off_t offset, struct fuse_file_info *fi){
299     writeToLog("read");
300
301     if(isLogFile(path)) {
302         return -ENOENT;
303     }
304
305     int currentBlock = (int) offset / blockSize;
306     CDE * cde;
307     ssize_t b = 0;
308     string fileName = string(path);
309     int readTotal = 0;
310     off_t newOffset;
311     bool firstRead = true;
312     //TODO what if offset + size > file_size?
313     while(true) { // will end when pread returns 0
314         newOffset = offset + (off_t) readTotal;
315         if(newOffset%blockSize != 0 && !firstRead){
316             return readTotal;
317         }
318
319         cacheMap.empty();
320
321         if(cacheMap.count({fileName, currentBlock}) > 0) {
322             // cache hit
323             cde = cacheMap[{fileName, currentBlock}];
324
325             int readSize = 0;
326             int x = blockSize * currentBlock;
327             if(readTotal == (int) size) {
328                 return readTotal;
329             } else if((size_t) newOffset >= cde->getSize() + x) {
330                 return readTotal;
331             } else if(newOffset + size < cde->getSize() + x) {
```

```
332                 readSize = (int) size;
333             } else {
334                 readSize = (cde->getSize() + x) - (int) newOffset;
335             }
336 
337             if((int) size <= readTotal + readSize) {
338                 readSize = size - readTotal;
339             }
340 
341             int inBlockOffset = (int) newOffset - currentBlock * blockSize;
342 
343             memcpy(buf + readTotal, cde->getData() + inBlockOffset,
344                    (size_t) readSize);
345             countChain.increment(cde);
346             lru.reinsert(cde);
347             readTotal += readSize;
348 
349         } else {
350             // cache miss
351             char *blockData = (char *) aligned_alloc(blockSize,
352                                                      blockSize * sizeof(char));
353             off_t tmpOffset = newOffset - newOffset%blockSize;
354             b = pread((int) fi->fh, (void *) blockData, (size_t) blockSize,
355                       tmpOffset);
356 
357             if (b < 0) {
358                 cout << -errno << endl;
359             } else if (b == 0) {
360                 free(blockData);
361                 return readTotal;
362             }
363 
364 
365             cacheMap[{fileName, currentBlock}] = new CDE(currentBlock,
366                                                          fileName, b,
367                                                          blockData);
368             CDE *cde = cacheMap[{fileName, currentBlock}];
369 
370             free(blockData);
371             // add the new cde (which has count of 1 to CountChain[0]
372             countChain.insert(cde, 1);
373             if ((int) lru.getSize() < numberOfBlocks) {
374                 // there is empty place in cache
375                 lru.insert(cde);
376                 cde->setIsNew(true);
377             } else {
378                 // cache is full, use replacement policy
379                 if (numberOfBlocks != 0) {
380                     removeFromCache();
381                     lru.insert(cde);
382                 }
383             }
384 
385             int readSize = 0;
386             int x = blockSize * currentBlock;
387             if(readTotal == (int) size) {
388                 return readTotal;
389             } else if((size_t) newOffset >= cde->getSize() + x) {
390                 return readTotal;
391             } else if(newOffset + size < cde->getSize() + x) {
392                 readSize = (int) size;
393             } else {
394                 readSize = (cde->getSize() + x) - (int) newOffset;
395             }
396 
397             if((int) size <= readTotal + readSize) {
398                 readSize = size - readTotal;
399             }
```

```
400
401             int inBlockOffset = (int) newOffset - currentBlock * blockSize;
402
403             memcpy(buf + readTotal, cde->getData() + inBlockOffset,
404                    (size_t) readSize);
405
406             readTotal += readSize;
407         }
408         currentBlock++;
409         firstRead = false;
410     }
411 }
412
413 /** Possibly flush cached data
414  *
415  * BIG NOTE: This is not equivalent to fsync().  It's not a
416  * request to sync dirty data.
417  *
418  * Flush is called on each close() of a file descriptor.  So if a
419  * filesystem wants to return write errors in close() and the file
420  * has cached dirty data, this is a good place to write back data
421  * and return any errors.  Since many applications ignore close()
422  * errors this is not always useful.
423  *
424  * NOTE: The flush() method may be called more than once for each
425  * open().  This happens if more than one file descriptor refers
426  * to an opened file due to dup(), dup2() or fork() calls.  It is
427  * not possible to determine if a flush is final, so each flush
428  * should be treated equally.  Multiple write-flush sequences are
429  * relatively rare, so this shouldn't be a problem.
430  *
431  * Filesystems shouldn't assume that flush will always be called
432  * after some writes, or that if will be called at all.
433  *
434  * Changed in version 2.2
435  */
436 int caching_flush(const char *path, struct fuse_file_info *fi)
437 {
438     writeToLog("flush");
439
440     if(isLogFile(path)) {
441         return -ENOENT;
442     }
443
444     return 0;
445 }
446
447 /** Release an open file
448  *
449  * Release is called when there are no more references to an open
450  * file: all file descriptors are closed and all memory mappings
451  * are unmapped.
452  *
453  * For every open() call there will be exactly one release() call
454  * with the same flags and file descriptor.  It is possible to
455  * have a file opened more than once, in which case only the last
456  * release will mean, that no more reads/writes will happen on the
457  * file.  The return value of release is ignored.
458  *
459  * Changed in version 2.2
460  */
461 int caching_release(const char *path, struct fuse_file_info *fi){
462     writeToLog("release");
463
464     if(isLogFile(path)) {
465         return -ENOENT;
466     }
467
```

```
468     int ret = close(fi->fh);
469     if(ret != 0) {
470         return -errno;
471     }
472     return ret;
473 }
474 
475 /** Open directory
476  *
477  * This method should check if the open operation is permitted for
478  * this  directory
479  *
480  * Introduced in version 2.3
481  */
482 int caching_opendir(const char *path, struct fuse_file_info *fi){
483     writeToLog("opendir");
484 
485     if(isLogFile(path)) {
486         return -ENOENT;
487     }
488 
489     DIR *dp;
490     int ret = 0;
491 
492     char fpath[PATH_MAX];
493     buildPath(fpath, path);
494 
495     dp = opendir(fpath);
496     if(dp == NULL) {
497         ret = -errno;
498     } else {
499         fi->fh = (intptr_t) dp;
500     }
501     return ret;
502 }
503 
504 /** Read directory
505  *
506  * This supersedes the old getdir() interface.  New applications
507  * should use this.
508  *
509  * The readdir implementation ignores the offset parameter, and
510  * passes zero to the filler function's offset.  The filler
511  * function will not return '1' (unless an error happens), so the
512  * whole directory is read in a single readdir operation.  This
513  * works just like the old getdir() method.
514  *
515  * Introduced in version 2.3
516  */
517 int caching_readdir(const char *path, void *buf,
518                     fuse_fill_dir_t filler,
519                     off_t offset, struct fuse_file_info *fi){
520     writeToLog("readdir");
521 
522     if(isLogFile(path)) {
523         return -ENOENT;
524     }
525 
526     int ret = 0;
527     DIR *dp;
528     struct dirent *de;
529 
530     dp = (DIR *) fi->fh;
531     de = readdir(dp);
532 
533     if(de == 0) {
534         return -errno;
535     }
```

```
536
537         do {
538             if(filler(buf, de->d_name, NULL, 0) != 0) {
539                 return -ENOMEM;
540             }
541         } while((de = readdir(dp)) != NULL);
542
543         return ret;
544     }
545
546     /** Release directory
547      *
548      * Introduced in version 2.3
549      */
550     int caching_releasedir(const char *path, struct fuse_file_info *fi){
551         writeToLog("releasedir");
552
553         if(isLogFile(path)) {
554             return -ENOENT;
555         }
556
557         int ret = closedir((DIR *) fi->fh);
558         if(ret != 0) {
559             return -errno;
560         }
561         return ret;
562     }
563
564     /** Rename a file */
565     int caching_rename(const char *path, const char *newpath){
566         writeToLog("rename");
567
568         if(isLogFile(path)) {
569             return -ENOENT;
570         }
571
572         char fpath[PATH_MAX];
573         char fnewpath[PATH_MAX];
574
575         buildPath(fpath, path);
576         buildPath(fnewpath, newpath);
577
578         int ret = rename(fpath, fnewpath);
579         if(ret != 0) {
580             return -errno;
581         }
582
583         // rename block in cache
584         CDE * cde = lru.getHead();
585         while(cde != nullptr) {
586             string fileName = cde->getFileName();
587             int pos = fileName.find(string(path));
588             if(pos == 0 && (fileName[string(path).length()] == '/' ||
589                     fileName.length() == string(path).length())) {
590
591                 string suffix = fileName.substr(string(path).length(),
592                                                 fileName.length());
593                 string realNewPath = string(newpath) + suffix;
594                 cde->setFileName(realNewPath);
595                 cacheMap[{realNewPath, cde->getBlockId()}] = cde;
596                 cacheMap.erase({string(path), cde->getBlockId()});
597             }
598             cde = cde->getNext();
599         }
600         return ret;
601     }
602
603     /**
```

```
 * Initialize filesystem
 *
 * The return value will passed in the private_data field of
 * fuse_context to all file operations and as a parameter to the
 * destroy() method.
 *

If a failure occurs in this function, do nothing (absorb the failure
and don't report it).
For your task, the function needs to return NULL always
(if you do something else, be sure to use the fuse_context correctly).
 * Introduced in version 2.3
 * Changed in version 2.6
 */
void *caching_init(struct fuse_conn_info *conn){
    writeToLog("init");
    return NULL;
}


/**
 * Clean up filesystem
 *
 * Called on filesystem exit.

If a failure occurs in this function, do nothing
(absorb the failure and don't report it).

 * Introduced in version 2.3
 */
void caching_destroy(void *userdata){
    writeToLog("destroy");
    logFile.close();
}


/**
 * Ioctl from the FUSE sepc:
 * flags will have FUSE_IOCTL_COMPAT set for 32bit ioctls in
 * 64bit environment.  The size and direction of data is
 * determined by _IOC_*() decoding of cmd.  For _IOC_NONE,
 * data will be NULL, for _IOC_WRITE data is out area, for
 * _IOC_READ in area and if both are set in/out area.  In all
 * non-NULL cases, the area is of _IOC_SIZE(cmd) bytes.
 *
 * However, in our case, this function only needs to print
 cache table to the log file .
 *
 * Introduced in version 2.8
 */
int caching_ioctl (const char *, int cmd, void *arg,
        struct fuse_file_info *, unsigned int flags, void *data){
    CDE * cde = lru.getTail();
    while(cde != nullptr) {
        writeToLog(cde->getFileName() + " " +
                           to_string((cde->getBlockId() + 1)) + " " +
                           to_string(cde->getCount()));
        cde = cde->getPrev();
    }
    return 0;
}


// Initialise the operations.
// You are not supposed to change this function.
void init_caching_oper()
{
```

```
672     caching_oper.getattr = caching_getattr;
673     caching_oper.access = caching_access;
674     caching_oper.open = caching_open;
675     caching_oper.read = caching_read;
676     caching_oper.flush = caching_flush;
677     caching_oper.release = caching_release;
678     caching_oper.opendir = caching_opendir;
679     caching_oper.readdir = caching_readdir;
680     caching_oper.releasedir = caching_releasedir;
681     caching_oper.rename = caching_rename;
682     caching_oper.init = caching_init;
683     caching_oper.destroy = caching_destroy;
684     caching_oper.ioctl = caching_ioctl;
685     caching_oper.fgetattr = caching_fgetattr;
686
687
688     caching_oper.readlink = NULL;
689     caching_oper.getdir = NULL;
690     caching_oper.mknod = NULL;
691     caching_oper.mkdir = NULL;
692     caching_oper.unlink = NULL;
693     caching_oper.rmdir = NULL;
694     caching_oper.symlink = NULL;
695     caching_oper.link = NULL;
696     caching_oper.chmod = NULL;
697     caching_oper.chown = NULL;
698     caching_oper.truncate = NULL;
699     caching_oper.utime = NULL;
700     caching_oper.write = NULL;
701     caching_oper.statfs = NULL;
702     caching_oper.fsync = NULL;
703     caching_oper.setxattr = NULL;
704     caching_oper.getxattr = NULL;
705     caching_oper.listxattr = NULL;
706     caching_oper.removexattr = NULL;
707     caching_oper.fsyncdir = NULL;
708     caching_oper.create = NULL;
709     caching_oper.ftruncate = NULL;
710 }
711
712
713 int main(int argc, char* argv[]){
714
715     // Check input parameters
716     if(!isInputParamsValid(argc, argv)) {
717         exit(0);
718     }
719
720     init_caching_oper();
721
722     blockSize = getBlockSize();
723     rootDir = realpath(argv[1], NULL);
724
725     strcat(logPath, rootDir);
726     strcat(logPath, FILE_LOCATION);
727
728     numberOfBlocks = atoi(argv[3]);
729
730     lru.setNewIndex((int) (atof(argv[5]) * numberOfBlocks));
731     lru.setOldIndex(numberOfBlocks - (int) (atof(argv[4]) *
732             numberOfBlocks) + 1);
733
734     logFile.open(logPath, std::ios_base::app); // create/open log file
735
736     // arrange args for fuse_main call
737     argv[1] = argv[2];
738     for (int i = 2; i< (argc - 1); i++){
739         argv[i] = NULL;
```

```
740         }
741         argv[2] = (char*) "-s";
742         //argv[3] = (char*) "-f";
743         argc = 3;
744 
745         int fuse_stat = fuse_main(argc, argv, &caching_oper, NULL);
746         free(rootDir);
747         return fuse_stat;
748     }
```

5 CountChain.h

```
1  //
2  // Created by root on 5/23/16.
3  //
4
5  #ifndef OS_EX4_COUNTCHAIN_H
6  #define OS_EX4_COUNTCHAIN_H
7  #include <vector>
8  #include "CDE.h"
9  #include <iostream>
10
11 using namespace std;
12
13 class CountChain {
14 private:
15     vector<pair<CDE *, CDE *>*> _countChain;
16 public:
17     CountChain(int CMax);
18     void insert(CDE * cde, int pos);
19     void increment(CDE * cde);
20     void remove(CDE * cde);
21     CDE * getItemToRemove();
22     void printCountChain();
23 };
24
25
26 #endif //OS_EX4_COUNTCHAIN_H
```

6 CountChain.cpp

```cpp
//
// Created by root on 5/23/16.
//

#include "CountChain.h"

CountChain::CountChain(int CMax) {
    for(int i = 0; i < CMax; ++i) {
        auto myPair = new pair<CDE *, CDE*>;
        myPair->first = nullptr;
        myPair->second = nullptr;
        _countChain.push_back(myPair);
    }
}

void CountChain::insert(CDE * cde, int pos) {
    --pos;
    CDE * head = _countChain.at(pos)->first;
    CDE * tail = _countChain.at(pos)->second;

    if(head != nullptr) {
        cde->setCountNext(head);
        head->setCountPrev(cde);
    }
    _countChain.at(pos)->first = cde;

    if(tail == nullptr) {
        _countChain.at(pos)->second = cde;
    }
}

void CountChain::increment(CDE * cde) {
    if(!cde->getIsNew()) { // increase only if not new
        int count = cde->getCount();
        if((size_t) count <= _countChain.size()) {
            remove(cde);
            if((size_t) count < _countChain.size()) {
                insert(cde, count + 1);
            }
        }
        cde->setCountPrev(nullptr);
        cde->increaseCount();
    }
}

void CountChain::remove(CDE * cde) {
    if((size_t) cde->getCount() > _countChain.size()) {
        return;
    }
    if(cde->getCountPrev() != nullptr) {
        cde->getCountPrev()->setCountNext(cde->getCountNext());
    } else {
        _countChain.at(cde->getCount() - 1)->first = cde->getCountNext();
    }

    if(cde->getCountNext() != nullptr) {
        cde->getCountNext()->setCountPrev(cde->getCountPrev());
        cde->setCountNext(nullptr);
    } else {
```

```
60          _countChain.at(cde->getCount() - 1)->second = cde->getCountPrev();
61      }
62  }
63
64  CDE * CountChain::getItemToRemove() {
65      for(auto it : _countChain) {
66          if(it->second != nullptr) {
67              if(it->second->getIsOld()) {
68                  return it->second;
69              }
70          }
71
72      }
73      return nullptr;
74  }
```

7 LRUStack.h

```
1  //
2  // Created by root on 5/21/16.
3  //
4
5  #ifndef OS_EX4_LRUSTACK_H
6  #define OS_EX4_LRUSTACK_H
7
8  #include "CDE.h"
9  #include <iostream>
10
11 class LRUStack {
12 private:
13     CDE * _head;
14     CDE * _tail;
15     CDE * _newBoundary;
16     CDE * _oldBoundary;
17     size_t _size;
18     int _newIndex, _oldIndex;
19 public:
20     LRUStack();
21     ~LRUStack();
22
23     size_t getSize() { return _size; };
24     void insert(CDE * cde);
25     void remove(CDE * cde);
26     void reinsert(CDE * cde);
27
28     void setNewIndex(int newIndex) { _newIndex = newIndex; };
29     void setOldIndex(int oldIndex) { _oldIndex = oldIndex; };
30
31     void setNewBoundary(CDE * cde) { _newBoundary = cde; };
32     void setOldBoundary(CDE * cde) { _oldBoundary = cde; };
33
34     CDE * setNewBoundary() { return _newBoundary; };
35     CDE * setOldBoundary() { return _oldBoundary; };
36
37     CDE * getTail() { return _tail; };
38     CDE * getHead() { return _head; };
39
40     void printLru();
41 };
42
43
44 #endif //OS_EX4_LRUSTACK_H
```

8 LRUStack.cpp

```cpp
//
// Created by root on 5/21/16.
//

#include "LRUStack.h"

using namespace std;

LRUStack::LRUStack() {
    _size = 0;
    _head = nullptr;
    _tail = nullptr;
    _newBoundary = nullptr;
    _oldBoundary = nullptr;
}

LRUStack::~LRUStack() {
    CDE * cde = _head;
    while(cde != nullptr) {
        CDE* tmpCde = cde;
        cde = cde->getNext();
        delete tmpCde;
    }
}

void LRUStack::insert(CDE * newCde) {

    if(_size == 0) {
        _tail = newCde;
    } else {
        _head->setPrev(newCde);
        newCde->setNext(_head);
    }
    _head = newCde;

    ++_size;
    if(_size == (size_t) _newIndex) {
        _newBoundary = _tail;
    } else if(_size == (size_t) _oldIndex) {
        _oldBoundary = _tail;
        _oldBoundary->setIsOld(true);
        _oldBoundary->setIsNew(false);
    }

    if(_size > (size_t) _newIndex) {
        _newBoundary->setIsNew(false);
        _newBoundary = _newBoundary->getPrev();
    }
};

void LRUStack::remove(CDE * cde) {
    _oldBoundary = cde->getPrev(); // prev because insert is coming
    _oldBoundary->setIsOld(true);
    _oldBoundary->setIsNew(false);

    if(_head == _tail) {
        _head = nullptr;
        _tail = nullptr;
    } else if(cde == _head) {
```

```
60          _head = cde->getNext();
61          cde->getNext()->setPrev(nullptr);
62      } else if(cde == _tail) {
63          _tail = cde->getPrev();
64          cde->getPrev()->setNext(nullptr);
65      } else {
66          cde->getPrev()->setNext(cde->getNext());
67          cde->getNext()->setPrev(cde->getPrev());
68      }
69      --_size;
70  };
71
72  void LRUStack::reinsert(CDE * cde) {
73
74      // update new boundary
75      if(cde->getIsNew()) {
76          if(_newBoundary == cde) {
77              if(cde->getPrev() != nullptr) {
78                  _newBoundary = cde->getPrev();
79              }
80          }
81      } else {
82          if(_newBoundary->getPrev() == nullptr) {
83              _newBoundary->setIsNew(false);
84              _newBoundary = cde;
85          } else {
86              _newBoundary->setIsNew(false);
87              _newBoundary = _newBoundary->getPrev();
88          }
89      }
90
91      // update old boundary
92      if(cde->getIsOld()) {
93          _oldBoundary = _oldBoundary->getPrev(); // prev because insert is coming
94          _oldBoundary->setIsOld(true);
95          _oldBoundary->setIsNew(false);
96      }
97
98      cde->setIsOld(false);
99      cde->setIsNew(true);
100
101     if(_head != cde) {
102         if(_tail == cde) {
103             _tail = cde->getPrev();
104             _tail->setNext(nullptr);
105         } else {
106             cde->getPrev()->setNext(cde->getNext());
107             cde->getNext()->setPrev(cde->getPrev());
108         }
109         cde->setNext(_head);
110         _head->setPrev(cde);
111         _head = cde;
112         _head->setPrev(nullptr);
113     }
114 };
```

9 Makefile

```
CC = gcc
RANLIB = ranlib

LIBSRC = CachingFileSystem.cpp CDE.h CDE.cpp CountChain.h
LIBSRC2 = CountChain.cpp LRUStack.h LRUStack.cpp

CPPFILES = CDE.cpp LRUStack.cpp CountChain.cpp CachingFileSystem.cpp

PKGFLAGS = `pkg-config fuse --cflags --libs`

LIBOBJ = $(LIBSRC:.cpp=.o)

INCS = -I.
CFLAGS = -Wall -std=c++11 -g $(INCS)
LOADLIBES = -L./

TAR = tar
TARFLAGS = -cvf
TARNAME = ex4.tar
TARSRCS = $(LIBSRC) $(LIBSRC2) Makefile README

all: CDE.cpp LRUStack.cpp CountChain.cpp CachingFileSystem.cpp
    g++ -Wall -std=c++11 $(CPPFILES) $(PKGFLAGS) -o CachingFileSystem

clean:
    rm *.o *.tar

tar:
    $(TAR) $(TARFLAGS) $(TARNAME) $(TARSRCS)

```