Contents

1 README

```
itamakatz, liavst2
Itamar Katz (555792977) , Liav Steinberg (203630090)
EX: 4

==================================
=             FILES              =
==================================

- README - this file.
- Makefile
- CachingFileSystem.cpp
- Cache.cpp
- LogFile.cpp
- Cache.h
- LogFile.h
- Block.h
- Block.cpp


==================================
=            DESIGN             =
==================================

Our Cache is an independent unit in our implementation.
In CachingFileSystem.cpp the functions that call the cache
are read and ioctl only.
Read calls the "collect" function which retrieves the wanted
information from the cache. Given the offset and the size
to read, collect calculates the starting and ending block
indexes to search in the cache. If the block doesnt exists,
collect calls the private function _cache_add to add this block
to the cache. If the block exists, collect calls _cache_replace
to place the current block on top of the Cache container.
All these blocks are placed in array, ordered by their
sequential access. Then we start to copy the content of the
blocks to "buf" using memcpy, and the wanted offset and size.

In ioctl, the cache simply summons the logfile on each block
in its repository. The logfile writes the block information
down.

==================================
=            ANSWERS             =
==================================


1. We know that accessing the disk is way slower than accessing
     the memory because the memory is physically closer to the
     cpu. In this excersice we store our information on the heap.
     If the heap is located in the RAM, it will be more efficient to
     read from it. But if the heap is located elsewhere, then our
     cache advantage will be lost. Also, searching in the cache may
     take time too if we are handling large amount of filele


2. In class, we saw there is a trade-off between sophisticated
     algorithm and a fast cache management. We require minimum
     time to handle a buffer cache managed by the OS because it
     is constantly changing.
```

```
3. LRU is better:
     files -> test1, test2, test3
     read ->
     test1,
     test1,
     test2,
     test2,
     test3,
     test3,

     LFU is better:
     Files -> test 1, test 2, test 3, test 4.
     read ->
     test 1
     test 2
     test 3
     test 1
     test 1
     test 1
     test 1
     test 1
     test 2
     test 3
     test 4


     Niether is better:
     Files -> test 1, test 2, test 3, test 4.

     Read:
     test 1
     test 2
     test 3
     test 4
     test 1
     test 2
     test 3
     test 4



4. Because there can be an intensive use in the same
     segment for a short period of time(This is called
     the locality principle as we saw) and then wont be
     in use anymore. That can happen to a block in the
     new section, increasing the block's reference count
     wont achieve this goal.
```

2 Block.h

```
#ifndef EX4_BLOCK_H
#define EX4_BLOCK_H

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

#define REF_COUNT_INIT 1


class Block {
public:
    int _refCount;
    std::string* _path;
    int _index;
    char* _content;

    Block(const char* path, int index, size_t blksize);

    ~Block();

    static bool comparator(const Block* , const Block*);
};



#endif //EX4_BLOCK_H
```

3 Block.cpp

```cpp
#include "Block.h"



Block::Block(const char* path, int index, size_t blksize):
        _refCount(REF_COUNT_INIT),
        _index(index){
    _path = new std::string(path);
    _content = (char*)aligned_alloc(blksize, blksize);
    memset(_content, 0, blksize);
}


Block::~Block(){
    delete(_path);
    free(_content);
    _path = NULL;
    _content = NULL;
}


bool Block::comparator(const Block* d1, const Block* d2) {
    return d1->_refCount > d2->_refCount;
}
```

4 Cache.h

```
#ifndef EX4_CACHE_H
#define EX4_CACHE_H

#include <cstdio>
#include <sys/stat.h>
#include <sys/types.h>
#include <map>
#include <queue>
#include <fcntl.h>
#include <math.h>
#include <unistd.h>
#include <string.h>
#include <algorithm>

#include "LogFile.h"
#include "Block.h"

#define FALIURE 1
#define SUCCESS 0


class Cache{
public:

    Cache(const double num_args[]);

    ~Cache();

    int collect(const char *path, char *buf, size_t size,
                off_t offset, struct fuse_file_info *fi);

    int ioctl_info(LogFile** lf, std::string rootdir);

    void inner_rename(std::string* path, std::string* newpath);

private:
    blksize_t _blksize;
    size_t _TotalSize;
    size_t _OldSize;
    size_t _MidSize;
    size_t _NewSize;

    std::deque<Block*> _Cache;

    void _cache_clean();

    char* _cache_replace(const char* path, int index);

    char* _cache_add(const char* path, struct fuse_file_info *fi, int index);

    void _cache_evict();

    void _sort_old(void);

    bool _find_existing_block(const char *path, int index);

    int _fsize(const char *path);

};
```

```

#endif //EX4_CACHE_H
```

5 Cache.cpp

```cpp
#include <iostream>
#include "Cache.h"



Cache::~Cache(){
    _cache_clean();
}



Cache::Cache(const double num_args[]){
    _TotalSize = (size_t) num_args[0];
    _NewSize = (size_t) std::floor((double)_TotalSize * num_args[1]);
    _OldSize = (size_t) std::floor((double)_TotalSize * num_args[2]);
    if (!_NewSize || !_OldSize){
        throw FALIURE;
    }
    _MidSize = _TotalSize - _NewSize - _OldSize;
    struct stat fi;
    stat("/tmp", &fi);
    _blksize = fi.st_blksize;
}



int Cache::collect(const char *path, char *buf, size_t size,
                   off_t offset, struct fuse_file_info *fi){

    int total_size = _fsize(path);
    if (total_size < 0){
        return total_size;
    }

    if (size > (size_t) (total_size - offset)){ // adjust the size
        size = (size_t) (total_size - offset);
    }

    if (total_size < offset || size <= 0){
        return SUCCESS;
    }

    // check if its in the cache
    int start_index = (int) std::floor((double)offset / (double)_blksize);
    int end_index = (int) std::floor
        ((((double)size + (double)offset) - 1)  / (double)_blksize);
    int num_of_blocks = end_index - start_index + 1;

    char* ret[num_of_blocks];
    for (int index = start_index, i = 0; i < num_of_blocks; index++, i++){

        if (!_find_existing_block(path, index)) {
            try {
                ret[i] = _cache_add(path, fi, index);
            } catch (std::string& err){
                return -errno;
            } catch (std::bad_alloc& err){
                return -FALIURE;
            }
            continue;
        }
        ret[i] = _cache_replace(path, index);
    }

    if (num_of_blocks == 1){
        memcpy(buf, ret[0] + offset - start_index*_blksize, size);
        return (int)size;
    }

    size_t buf_offset = (size_t) ((start_index + 1)*_blksize - offset);
    memcpy(buf, ret[0] + offset - start_index*_blksize, buf_offset);


    for (int i = 1; i < num_of_blocks - 1; i++){
        memcpy(buf + buf_offset, ret[i], (size_t) _blksize);
        buf_offset += (size_t) (_blksize);
    }

    memcpy(buf + buf_offset, ret[num_of_blocks - 1], size - buf_offset);
    return (int)size;
}


bool Cache::_find_existing_block(const char *path, int index){
    for (Block* block: _Cache){
        if (block->_index == index && !block->_path->compare(path)){
            return true;
        }
    }
    return false;
}



int Cache::ioctl_info(LogFile** lf, std::string rootdir){
    for (size_t i = 0; i < _Cache.size(); i++){
        std::string file(*(_Cache[i]->_path));
        size_t pos = file.find(rootdir);
        std::string path = file.substr(pos + rootdir.length() + 1);
        (*lf)->cache_ioctl(path, _Cache[i]->_index, _Cache[i]->_refCount);
    }
    return SUCCESS;
}



void Cache::inner_rename(std::string* path, std::string* newpath) {

    for (Block* item: _Cache){
        if (path->length() > item->_path->length()){
            continue;
        }
        std::string prefix_path = item->_path->substr(0, path->length());
        if (!prefix_path.compare(*path)) {
            item->_path->replace(0, path->length(), *newpath);
        }
    }
    return;
}



void Cache::_cache_clean(){
    for (Block* item: _Cache){
        delete(item);
    }
    _Cache.clear();
    return;
}



char* Cache::_cache_replace(const char* path, int index){

    auto iter = _Cache.begin();
    for (size_t i = 1; i <= _Cache.size(); i++, iter++){
        if(!(*iter)->_path->compare(path) && (*iter)->_index == index) {
            Block* block = std::move((*iter));
            _Cache.erase(iter);
            _Cache.push_front(block);

            if (i > _NewSize){ // was not in new section
                block->_refCount++;
                if(i > _NewSize + _MidSize){ // was in old section
                    _sort_old();
                }
            }

            return block->_content;
        }
    }

    return NULL;
}



char* Cache::_cache_add(const char* path,struct fuse_file_info*fi, int index){

    Block* newBlock = new Block(path, index, (size_t) _blksize);
    ssize_t amount = pread((int)fi->fh, newBlock->_content,
                           (size_t) _blksize, (off_t)(index*_blksize));
    if (amount < 0){
        throw "error";
    }

    _Cache.push_front(newBlock);
    if (_Cache.size() == _TotalSize + 1){
        _cache_evict();
    }

    return newBlock->_content;
}



void Cache::_cache_evict(){
    Block* evicted = std::move(_Cache.back());
    _Cache.pop_back();
    delete(evicted);
    _sort_old();
}



void Cache::_sort_old(void){
    auto iter = _Cache.begin();
    std::advance(iter, _TotalSize - _OldSize);
    std::stable_sort(iter, _Cache.end(), Block::comparator);
}



int Cache::_fsize(const char *file){
    struct stat st;
    return stat(file, &st) ? (-errno): (int)st.st_size;
}
```

6 CachingFileSystem.cpp

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

#define FUSE_USE_VERSION 26
#define FUSE_ARG_NUM 3
#define MINIMUM_ARGS_NUM 6
#define SUCCESS 0
#define USAGE "Usage: CachingFileSystem rootdir mountdir\
 numberOfBlocks fOld fNew"
#define DIR_ERROR "System Error: Can't evaluate given directories"
#define ALLOCATION_FAIL "System Error: Allocation Failure"

#include <errno.h>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <unistd.h>
#include <dirent.h>

#include "Cache.h"

struct fuse_operations g_caching_oper;
static Cache* cache;
static char rootdir[PATH_MAX];
static LogFile* lf;


bool LogFileAccess(const char* path){
    return !strcmp(path, LOG_FILE_PATH);
}


/**
 * returns the absolute path of the file
 */
char* absolute_path(const char *path){
    char g_abs_path[PATH_MAX];
    strcpy(g_abs_path, rootdir);
    return strncat(g_abs_path, path, PATH_MAX);
}


/**
 * implements getattr
 */
int caching_getattr(const char *path, struct stat *statbuf){

    lf->command("getattr");
    if (LogFileAccess(path)){ // trying to reach our logfile
        return -ENOENT;
    }

    int ret = lstat(absolute_path(path), statbuf);
    return (ret < 0) ? (-errno): ret;
}

```

```c

/**
 * implements fgetattr
 */
int caching_fgetattr(const char *path, struct stat *statbuf,
                     struct fuse_file_info *fi){
    lf->command("fgetattr");
    if (LogFileAccess(path)){ // trying to reach our logfile
        return -ENOENT;
    }

    if (!strcmp(path, "/")){
        return caching_getattr(path, statbuf);
    }

    int ret = fstat(fi->fh, statbuf);
    return (ret < 0) ? (-errno): ret;
}


/**
 * implements access
 */
int caching_access(const char *path, int mask){

    lf->command("access");
    if (LogFileAccess(path)){ // trying to reach our logfile
        return -ENOENT;
    }

    int ret = access(absolute_path(path), mask);
    return (ret < 0) ? (-errno): ret;
}


/**
 * implements open
 */
int caching_open(const char *path, struct fuse_file_info *fi){

    lf->command("open");
    if (LogFileAccess(path)){ // trying to reach our logfile
        return -ENOENT;
    }

    fi->direct_io = 1;
    if ((fi->flags & 3) != O_RDONLY) {
        return -EACCES;
    }

    int fd = open(absolute_path(path), O_RDONLY|O_DIRECT|O_SYNC);
    if (fd < 0){
        return -ENOENT;
    }

    fi->fh = fd;


    return SUCCESS;
}


/**
 * implements read
 */
int caching_read(const char *path, char *buf, size_t size,
                 off_t offset, struct fuse_file_info *fi){

```

```c
    lf->command("read");
    if (LogFileAccess(path)){ // trying to reach our logfile
        return -ENOENT;
    }

    int ret = cache->collect(absolute_path(path), buf, size, offset, fi);
    return ret;
}


/**
 * implements flush
 */
int caching_flush(const char *path, struct fuse_file_info*){

    lf->command("flush");
    if (LogFileAccess(path)){ // trying to reach our logfile
        return -ENOENT;
    }

    return SUCCESS;
}


/**
 * implements release
 */
int caching_release(const char *path, struct fuse_file_info *fi){

    lf->command("release");
    if (LogFileAccess(path)){ // trying to reach our logfile
        return -ENOENT;
    }

    int ret = close((int)fi->fh);
    return (ret < 0) ? (-errno): ret;
}


/**
 * implements opendir
 */
int caching_opendir(const char *path, struct fuse_file_info *fi){

    lf->command("opendir");
    if (LogFileAccess(path)){ // trying to reach our logfile
        return -ENOENT;
    }

    DIR* dir = opendir(absolute_path(path));

    if (dir == NULL){
        return -errno;
    }

    fi->fh = (intptr_t) dir;
    return SUCCESS;
}


/**
 * implements readdir
 */
int caching_readdir(const char* path, void *buf,
                    fuse_fill_dir_t filler, off_t ,struct fuse_file_info *fi){

    lf->command("readdir");
    if (LogFileAccess(path)){ // trying to reach our logfile
```

```
        return -ENOENT;
    }

    DIR* dp = (DIR*) (uintptr_t) fi->fh;
    struct dirent *de;
    if ((de = readdir(dp)) == 0){
        return -errno;
    }

    do
    {
        if (LogFileAccess(de->d_name)){
            continue;
        }
        if (filler(buf, de->d_name, NULL, 0)){
            return -ENOMEM;
        }
    } while ((de = readdir(dp)) != NULL);

    return SUCCESS;
}

/**
 * implements releasedir
 */
int caching_releasedir(const char* path, struct fuse_file_info *fi){

    lf->command("releasedir");
    if (LogFileAccess(path)){ // trying to reach our logfile
        return -ENOENT;
    }

    int ret = closedir((DIR*) (uintptr_t) fi->fh);
    return (ret < 0) ? (-errno): ret;
}

/**
 * implements rename
 */
int caching_rename(const char *path, const char *newpath){

    lf->command("rename");
    if (LogFileAccess(path)){ // trying to reach our logfile
        return -ENOENT;
    }


    std::string oldPath(absolute_path(path));
    std::string newPath(absolute_path(newpath));

    int ret = rename(oldPath.c_str(), newPath.c_str());
    if (ret < 0){
        return -errno;
    }

    // also change in the cache
    cache->inner_rename(&oldPath, &newPath);
    return ret;
}

/**
 * implements init
 */
void* caching_init(struct fuse_conn_info*){

    lf->command("init");
    return NULL;
}
```

```

/**
 * implements destroy
 */
void caching_destroy(void*){

    lf->command("destroy");
    delete(cache);
    delete(lf);
    return;
}

/**
 * implements ioctl
 */
int caching_ioctl (const char*, int, void*,
                   struct fuse_file_info *, unsigned int, void*){
    lf->command("ioctl");
    cache->ioctl_info(&lf, rootdir);
    return SUCCESS;
}



void init_caching_oper() {

    g_caching_oper.getattr = caching_getattr;
    g_caching_oper.access = caching_access;
    g_caching_oper.open = caching_open;
    g_caching_oper.read = caching_read;
    g_caching_oper.flush = caching_flush;
    g_caching_oper.release = caching_release;
    g_caching_oper.opendir = caching_opendir;
    g_caching_oper.readdir = caching_readdir;
    g_caching_oper.releasedir = caching_releasedir;
    g_caching_oper.rename = caching_rename;
    g_caching_oper.init = caching_init;
    g_caching_oper.destroy = caching_destroy;
    g_caching_oper.ioctl = caching_ioctl;
    g_caching_oper.fgetattr = caching_fgetattr;


    g_caching_oper.readlink = NULL;
    g_caching_oper.getdir = NULL;
    g_caching_oper.mknod = NULL;
    g_caching_oper.mkdir = NULL;
    g_caching_oper.unlink = NULL;
    g_caching_oper.rmdir = NULL;
    g_caching_oper.symlink = NULL;
    g_caching_oper.link = NULL;
    g_caching_oper.chmod = NULL;
    g_caching_oper.chown = NULL;
    g_caching_oper.truncate = NULL;
    g_caching_oper.utime = NULL;
    g_caching_oper.write = NULL;
    g_caching_oper.statfs = NULL;
    g_caching_oper.fsync = NULL;
    g_caching_oper.setxattr = NULL;
    g_caching_oper.getxattr = NULL;
    g_caching_oper.listxattr = NULL;
    g_caching_oper.removexattr = NULL;
    g_caching_oper.fsyncdir = NULL;
    g_caching_oper.create = NULL;
    g_caching_oper.ftruncate = NULL;
}


/**
```

```
 * checks main args
 */
void check_args(const char* mountdir, const double num_args[]){

    if (*std::min_element(num_args, num_args+2) <= 0 ||
            num_args[1]+num_args[2] > 1){
        std::cout << USAGE << std::endl;
        exit(FALIURE);
    }

    DIR* d1 = opendir(rootdir);
    DIR* d2 = opendir(mountdir);

    if (errno == ENOENT){ // one or both directories don't exist
        std::cout << USAGE << std::endl;
        exit(FALIURE);
    }

    closedir(d1);
    closedir(d2);
    return;
}



int main(int argc, char* argv[]){

    if (argc < MINIMUM_ARGS_NUM) {
        std::cout << USAGE << std::endl;
        exit(FALIURE);
    }

    char mountdir[PATH_MAX];
    if (realpath(argv[1], rootdir) == NULL ||
        realpath(argv[2], mountdir) == NULL){
        std::cerr << DIR_ERROR << std::endl;
        exit(FALIURE);
    }

    const double num_args[] =
            {(double) atoi(argv[3]), atof(argv[4]), atof(argv[5])};
    check_args(mountdir, num_args);

    try {
        cache = new Cache(num_args);
    } catch (int err){
        std::cout << USAGE << std::endl;
        exit(FALIURE);
    } catch (std::bad_alloc& err){
        std::cerr << ALLOCATION_FAIL << std::endl;
        exit(FALIURE);
    }

    try {
        lf = new LogFile(rootdir);
    } catch (std::bad_alloc& err){
        delete(cache);
        std::cerr << ALLOCATION_FAIL << std::endl;
        exit(FALIURE);
    }

    char* fuse_args[] = {argv[0], mountdir, (char*)"-s"};
    int number_of_args = FUSE_ARG_NUM;

    init_caching_oper();
    int fuse_stat =fuse_main(number_of_args, fuse_args, &g_caching_oper,NULL);
    return fuse_stat;
}
```

7 LogFile.h

```
#ifndef EX4_LOGFILE_H
#define EX4_LOGFILE_H


#include <iosfwd>
#include <fstream>
#include <fuse.h>

#define LOG_FILE_PATH "/.filesystem.log"


class LogFile{
private:
    std::ofstream __lfile__;

public:
    LogFile(std::string);

    ~LogFile();

    void command(std::string);

    void cache_ioctl(std::string, int, int);
};


#endif //EX4_LOGFILE_H
```

8 LogFile.cpp

```cpp
#include "LogFile.h"


LogFile::LogFile(std::string fullpath) {
    fullpath.append(LOG_FILE_PATH);
    __lfile__.open(fullpath, std::ios_base::app);
}


LogFile::~LogFile(){
    __lfile__.close();
}

void LogFile::command(std::string syscall) {
    __lfile__ << time(NULL) << " " << syscall << std::endl;
}

void LogFile::cache_ioctl(std::string path, int index, int ref){
    __lfile__ << path << " " << (index + 1) << " " << ref << std::endl;
}
```

9 Makefile

```makefile
CFLAGS = -g -std=c++11 -DNDEBUG -Wall -D_FILE_OFFSET_BITS=64
FUSE_FLAGS = `pkg-config fuse --cflags --libs`

TAR_NAME = ex4.tar
SOURCES = CachingFileSystem.cpp LogFile.cpp Cache.cpp Block.cpp
HEADERS = LogFile.h Cache.h Block.h
OBJS = $(SOURCES:.cpp=.o)
EXTRA_FILES = README Makefile
TAR_FILES = $(SOURCES) $(HEADERS) $(EXTRA_FILES)

EXEC = CachingFileSystem

.DEFAULT_GOAL = $(EXEC)


all: $(EXEC) tar

$(EXEC): $(OBJS)
    $(CXX) $(FUSE_FLAGS) $^ -o $@


%.o: %.cpp $(HEADERS)
    $(CXX) -c $(CFLAGS) $<

tar:
    tar -cvf $(TAR_NAME) $(TAR_FILES)


clean:
    rm -rf $(TAR_NAME) $(OBJS) $(EXEC)


.PHONY:
    all tar clean
```