Contents

1 Basic Test Results

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

2 README

```
akiva_s, roigreenberg
Akiva Sygal(305277220), Roi Greenberg(305571234)
Ex: 4

FILES:
README
CachingFileSystem.cpp -- a file that implements the Caching File System
makefile

REMARKS:
Most of the functins are implement same as the bbfs example and are very simple
so no need to expplain them.
We will explain how the read work
for the cache we have struct containing general information as block nmber,
directories path ect. nd also contain a vector Blocks.
The Blocks are second struct, contain information for single block.
This information is the path for the file the data took from, the number of the
block in this file, a counter of use and f course the data itself.
Each time a file beign read, we are going block by block.
For each block we first look for an exists block. If we can't find one, we
"activate" another block. If all the block are correntlly in use we search for
the least frequancy block, and replace it with the new block.
Then the function copy the data to the buffer and goes to the next rblock if
needed.


ANSWERS:

1 by default, heap memory is saved on RAM and so accessing it is much faster
than accessing the disk. *but* the heap is a virtual memory section and can
somtimes be mapped into disk - no garantuee that our heap-cache would really
be saved on RAM, so if it is actually saved on disk, we would get no better
performances.
2. no implementation is absolutely better: in Array implementation, reading
a specificf block is easier since we have random access to it, but when we
have to delete a block it would take O(n) time to find the ideal block to
delete, so this implementation is better when you know that your program would
read the same files (=blocks) a lot of times and would seldom read new files.
In the other hand, the list implementation can easily delete the less used
block but cannot access spesific blocks efficiently, so it should be preffered
 when we most of the time read new information and seldom read the same data
several times.,
3. the difference is that paging is used for *memory access*. memory access
treated in user-level and so happens all time, so updating the data structure
each time the memory is accessed (by user or so) is not realistic (veryy
expensive). but reading a block from disk can be treated only by kernel - it
requires a system call, takes much more time and so happens much seldom, so in
this case updating the data structure in each access to a filr block is
realistic.
4. LRU better: if we are treating each file after reading it, and only when
finishing the work with the current file we continue to read the next one.
In this case, deleting the less recently block would delete a block from the
oldest file imported, which most probabely is not needed antmore.
LFU better: if we are treating files with different information to be handled,
where some of the information is really important and have to be used a lot of
time, where somw of the other files contain unimportent information that after
reading once is not needed. in that case we want to delete the less important
information, which is most likely the information we have read the less number
of times.
```

both strategies are bad: when we are a file server that only provides file reading services to some user. we cannot know what is the next file the user will ask us to supply, so each of the algorithm is only a guess since we don’t know the logic of files reading.

5. the size of blocks that the OS read from the file system. if we read smaller blocks, it causes some "internal fragmentation" since when we want to read our small-size block, in any case the os would have to import the whole os-sized block from the memory and the rest of it would be spended. if we read more than the os block size each time, in each block importation we actually need to import more than 1 block, making the import action much expensive than needed.

3 CachingFileSystem.cpp

```
/*
 * CachingFileSystem.cpp
 *
 *  Created on: 15 April 2015
 *  Author: Netanel Zakay, HUJI, 67808  (Operating Systems 2014-2015).
 */

#define FUSE_USE_VERSION 26

#include <fuse.h>
#include <list>
#include <fcntl.h>
#include <vector>
#include <cstring>
#include <string>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <fstream>
#include <iostream>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <map>
#include <dirent.h>
#include <cassert>
#include <stdexcept>

#define BLOCK(x) (int)(x)/cache_data->block_size +1
#define SET_TIMER    if ((timer=time(nullptr)) == (time_t)(-1)) {return -errno;}

using namespace std;

struct fuse_operations caching_oper;

ofstream logfile;
time_t timer;

class MainError: public exception
{
    virtual const char* what() const throw ()
    {
        return "error in main";
    }
};

class SystemError: public exception
{
    virtual const char* what() const throw ()
    {
        return strerror(errno);
    }
};

typedef struct Block
{
    int num_of_accesses;
    char* data;
    list<int> cache_index;
    string file_path;
    size_t size;
    int block_num;
    Block(size_t size);
    ~Block()
    {
        delete[](data);
    }
    void rename(string new_file_path)
    {
        file_path = new_file_path;
    }
    void init_block(const string file_path, int block_num);
    void reset_block(void);

} Block;

Block::Block(size_t size)
{
    this->size = size;
    file_path = "";
    num_of_accesses = 0;
    data = new char[size];
    if (data == nullptr)
    {
        bad_alloc ex;
        throw ex;
    }
    for (size_t i = 0; i < size; i++)
    {
        data[i] = '\0';
    }
    block_num = -1;
}

void Block::init_block(string file_path, int block_num)
{
    this->file_path = file_path;
    num_of_accesses = 0;
    for (size_t i = 0; i < size; i++)
    {
        data[i] = '\0';
    }
    this->block_num = block_num;
}

void Block::reset_block(void)
{
    file_path = "";
    num_of_accesses = 0;
    for (size_t i = 0; i < size; i++)
    {
        data[i] = '\0';
    }
    block_num = -1;
}

typedef struct Cache
{
    vector<Block*> cache;

    char* rootdir;
    char* mountdir;
    int block_size;
    int num_of_blocks;

    string logfile_name = ".filesystem.log";
    unsigned int occupied_blocks;
    Cache(char* argv[]);
    ~Cache()
    {
        vector<Block*>::iterator it;
        for (it = cache.begin(); it != cache.end(); it++)
        {

            if ((*it) != nullptr)
            {
                delete(*it);
            }
        }
        free(rootdir);
        free(mountdir);
    }
    Block * find_block(string path, const int block_num);
    Block * assign_new_block();
} Cache;

Cache::Cache(char* argv[])
{
    struct stat buffer;
    rootdir = realpath(argv[1], NULL);
    if ((rootdir == NULL) || stat(rootdir, &buffer) == -1)
    {
        throw MainError();
    }
    if (S_ISDIR(buffer.st_mode) == false)
    {
        throw MainError();
    }
    mountdir = realpath(argv[2], NULL);
    if ((mountdir == NULL) || stat(mountdir, &buffer) == -1)
    {
        throw MainError();
    }
    if (S_ISDIR(buffer.st_mode) == false)
    {
        throw MainError();
    }
    if ((num_of_blocks = atoi(argv[3])) <= 0)
    {
        throw MainError();
    }
    if ((block_size = atoi(argv[4])) <= 0)
    {
        throw MainError();
    }
    cache.resize(num_of_blocks, nullptr);
    vector<Block*>::iterator it;
    for (it = cache.begin(); it != cache.end(); it++)
    {
        (*it) = new Block(block_size);
    }
    occupied_blocks = 0;
}

/*
 * searching for exists block
 * return pointer to the block or null if doesn't exists
 */
Block * Cache::find_block(string path, const int block_num)
{
    vector<Block*>::iterator it;
    for (it = cache.begin(); it != cache.end(); it++)
    {

        if ((*it)->file_path.compare(path) == 0
                && (*it)->block_num == block_num)
        {
            return *it;
        }
    }
    return nullptr;
}

/*
 * assigning new block.
 * if can't find empty block, delete the least used and create new instead
 */
Block * Cache::assign_new_block()
{

    vector<Block*>::iterator it;
    for (it = cache.begin(); it != cache.end(); it++)
    {
        if ((*it)->block_num == -1)
        {
            return *it ;
        }
    }
    //in case the cache is full. free the least accessed block and assigned new
    int i = 0;
    int min = 0;
    for (it = cache.begin(); it != cache.end(); it++, i++)
    {
        if ((*it)->num_of_accesses < cache.at(min)->num_of_accesses)
        {
            min = i;
        }
    }
    cache.at(min)->reset_block();
    return cache.at(min);
}

static Cache* cache_data;

string translate_path(string virtual_path)
{
    if (virtual_path.compare("/" + cache_data->logfile_name) == 0)
    {
        return "";
    }
    string real_path = cache_data->rootdir + virtual_path;

    return real_path;
}

string get_virtual_path(string absolute_path)
{
    unsigned int pos = absolute_path.find(cache_data->rootdir);
    if (pos != string::npos)
    {
        pos += strlen(cache_data->rootdir);
        absolute_path.erase(0, pos + 1);
    }
    return absolute_path;
}

/** Get file attributes.
 *
 * Similar to stat().  The 'st_dev' and 'st_blksize' fields are
 * ignored.  The 'st_ino' field is ignored except if the 'use_ino'
 * mount option is given.
 */
```

```
int caching_getattr(const char *path, struct stat *statbuf)
{
        SET_TIMER
        logfile << timer << " getattr\n";

        string real_path = translate_path(path);
        if (real_path.compare("") == 0)
        {
            return -ENOENT;
        }
        int retstat = 0;
        retstat = lstat(real_path.c_str(), statbuf);
        if (retstat != 0)
        {
            return -errno;
        }

        return retstat;
}

/**
 * Get attributes from an open file
 *
 * This method is called instead of the getattr() method if the
 * file information is available.
 *
 * Currently this is only called after the create() method if that
 * is implemented (see above).  Later it may be called for
 * invocations of fstat() too.
 *
 * Introduced in version 2.5
 */
int caching_fgetattr(const char *path, struct stat *statbuf,
        struct fuse_file_info *fi)
{
    SET_TIMER
    logfile << timer << " fgetattr\n";

    string real_path = translate_path(path);
    if (real_path.compare("") == 0)
    {
        return -ENOENT;
    }
    if (strcmp(path, "/") == 0)
    {
        return caching_getattr(path, statbuf);
    }
    int retstat = 0;
    retstat = fstat(fi->fh, statbuf);
    if (retstat != 0)
    {
        return -errno;
    }
    return retstat;
}

/**
 * Check file access permissions
 *
 * This will be called for the access() system call.  If the
 * ’default_permissions’ mount option is given, this method is not
 * called.
 *
 * This method is not called under Linux kernel versions 2.4.x
 *
 * Introduced in version 2.5
 */
int caching_access(const char *path, int mask)
```

```
{
    SET_TIMER
    logfile << timer << " access\n";

    string real_path = translate_path(path);
    if (real_path.compare("") == 0)
    {
        return -ENOENT;
    }
    int retstat = 0;
    retstat = access(real_path.c_str(), mask);
    if (retstat != 0)
    {
        return -errno;
    }
    return retstat;
}

/** File open operation
 *
 * No creation, or truncation flags (O_CREAT, O_EXCL, O_TRUNC)
 * will be passed to open().  Open should check if the operation
 * is permitted for the given flags.  Optionally open may also
 * return an arbitrary filehandle in the fuse_file_info structure,
 * which will be passed to all file operations.

 * pay attention that the max allowed path is PATH_MAX (in limits.h).
 * if the path is longer, return error.

 * Changed in version 2.2
 */
int caching_open(const char *path, struct fuse_file_info *fi)
{
    SET_TIMER
    logfile << timer << " open\n";

    int retstat = 0;
    int fd;
    string real_path = translate_path(path);
    if (real_path.compare("") == 0)
    {
        return -ENOENT;
    }
    if ((fi->flags & 3) != O_RDONLY)
    {
        return -EACCES;
    }

    fd = open(real_path.c_str(), fi->flags);
    if (retstat != 0)
    {
        return -errno;
    }

    fi->fh = fd;

    return retstat;
}

/*
 * search for data block.
 * if can't find the correct block, create new one and write the data
 * return pointer to the data
 */
char * read_block(int fd, string path, int block_num)
{
    int retstat = 0;
    Block * block = cache_data->find_block(path, block_num);
    if (block == nullptr)
    {
        block = cache_data->assign_new_block();
        block->init_block(path, block_num);
        retstat = pread(fd, block->data, cache_data->block_size,
                (block_num - 1) * cache_data->block_size);
        if (retstat == -1)
        {
            throw SystemError();
        }
    }
    block->num_of_accesses++;
    return block->data;
}

/** Read data from an open file
 *
 * Read should return exactly the number of bytes requested except
 * on EOF or error, otherwise the rest of the data will be
 * substituted with zeroes.
 *
 * Changed in version 2.2
 */
int caching_read(const char *path, char *buf, size_t size, off_t offset,
        struct fuse_file_info *fi)
{
    SET_TIMER;
    logfile << timer << " read\n";

    string real_path = translate_path(path);
    if (real_path.compare("") == 0)
    {
        return -ENOENT;
    }

    struct stat statbuf;
    if (lstat(real_path.c_str(), &statbuf) != 0)
    {
        return -errno;
    }
    if (offset > statbuf.st_size)
    {
        return -ENXIO;
    }
    if ((int) (size + offset) > statbuf.st_size)
    {
        size = statbuf.st_size - offset;
    }

    int retstat = 0;

    int first_block = BLOCK(offset);
    int last_block = BLOCK(offset + size);
    offset = offset % cache_data->block_size;
    char * tmp;
    try
    {
        for (int j = first_block; j < last_block; ++j)
        {

            tmp = read_block(fi->fh, real_path, j);
            memcpy(buf + retstat, tmp + offset, cache_data->block_size - offset);
            retstat += cache_data->block_size - offset;
            offset = 0;
        }
        int end = size - retstat;
        if (end > 0)
        {
            tmp = read_block(fi->fh, real_path, last_block);
            memcpy(buf + retstat, tmp + offset, end);
            retstat += end;
        }
    }
    catch (bad_alloc &e)
    {
        return -ENOMEM;
    }
    catch (SystemError &e)
    {
        return -errno;
    }

    return retstat;
}

/** Possibly flush cached data
 *
 * BIG NOTE: This is not equivalent to fsync().  It's not a
 * request to sync dirty data.
 *
 * Flush is called on each close() of a file descriptor.  So if a
 * filesystem wants to return write errors in close() and the file
 * has cached dirty data, this is a good place to write back data
 * and return any errors.  Since many applications ignore close()
 * errors this is not always useful.
 *
 * NOTE: The flush() method may be called more than once for each
 * open().  This happens if more than one file descriptor refers
 * to an opened file due to dup(), dup2() or fork() calls.  It is
 * not possible to determine if a flush is final, so each flush
 * should be treated equally.  Multiple write-flush sequences are
 * relatively rare, so this shouldn't be a problem.
 *
 * Filesystems shouldn't assume that flush will always be called
 * after some writes, or that if will be called at all.
 *
 * Changed in version 2.2
 */
int caching_flush(const char *path, struct fuse_file_info *fi)
{
    SET_TIMER
    logfile << timer << " flush\n";
    return 0;
}

/** Release an open file
 *
 * Release is called when there are no more references to an open
 * file: all file descriptors are closed and all memory mappings
 * are unmapped.
 *
 * For every open() call there will be exactly one release() call
 * with the same flags and file descriptor.  It is possible to
 * have a file opened more than once, in which case only the last
 * release will mean, that no more reads/writes will happen on the
 * file.  The return value of release is ignored.
 *
 * Changed in version 2.2
 */
int caching_release(const char *path, struct fuse_file_info *fi)
{
    SET_TIMER
    logfile << timer << " release\n";

    int retstat = close(fi->fh);
    if (retstat != 0)
    {
        return -errno;
    }
    return retstat;
}

/** Open directory
 *
 * This method should check if the open operation is permitted for
 * this  directory
 *
 * Introduced in version 2.3
 */
int caching_opendir(const char *path, struct fuse_file_info *fi)
{
    SET_TIMER
    logfile << timer << " opendir\n";

    DIR *dp;
    int retstat = 0;
    string real_path = translate_path(path);

    dp = opendir(real_path.c_str());
    if (dp == NULL)
    {
        return -errno;
    }

    fi->fh = (intptr_t) dp;

    return retstat;
}

/** Read directory
 *
 * This supersedes the old getdir() interface.  New applications
 * should use this.
 *
 * The readdir implementation ignores the offset parameter, and
 * passes zero to the filler function’s offset.  The filler
 * function will not return ’1’ (unless an error happens), so the
 * whole directory is read in a single readdir operation.  This
 * works just like the old getdir() method.
 *
 * Introduced in version 2.3
 */
int caching_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
        off_t offset, struct fuse_file_info *fi)
{
    SET_TIMER;
    logfile << timer << " readdir\n";

    int retstat = 0;
    DIR *dp;
    struct dirent *de;

    dp = (DIR *) (uintptr_t) fi->fh;
    int prev_errno = errno;

    while ((de = readdir(dp)) != NULL)
    {
        if ((strcmp(de->d_name, cache_data->logfile_name.c_str()) != 0)
        {
            if ((filler(buf, de->d_name, NULL, 0) != 0))
            {
                return -ENOMEM;
            }
        }
        prev_errno = errno;
    };

    if (prev_errno != errno)
    {
        return -errno;
    }

    return retstat;
}

/** Release directory
 *
 * Introduced in version 2.3
 */
int caching_releasedir(const char *path, struct fuse_file_info *fi)
{
    SET_TIMER
    logfile << timer << " releasedir\n";

    int retstat = 0;

    if (closedir((DIR *) (uintptr_t) fi->fh) == -1)
    {
        return -errno;
    }

    return retstat;
}

/** Rename a file */
int caching_rename(const char *path, const char *newpath)
{
    SET_TIMER;
    logfile << timer << " rename\n";

    int retstat = 0;
    string new_path = translate_path(newpath);
    string real_path = translate_path(path);
    if (real_path == "")
    {
        return -ENONET;
    }
    if (new_path == "")
    {
        return -EINVAL;
    }
    retstat = rename(real_path.c_str(), new_path.c_str());

    if (retstat != 0)
    {
        return -errno;
    }

    vector<Block*>::iterator it;
    for (it = cache_data->cache.begin(); it != cache_data->cache.end(); it++)
    {
        if ((*it)->file_path.compare(path) == 0)
        {
            (*it)->file_path = new_path;
        }
    }
    return retstat;
}

/**
 * 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.
 *
 * Introduced in version 2.3
 * Changed in version 2.6
 */
void *caching_init(struct fuse_conn_info *conn) {
    logfile << time(nullptr) << " init\n";
    return cache_data;
}

/**
 * Clean up filesystem
 *
 * Called on filesystem exit.
 *
 * Introduced in version 2.3
 */
void caching_destroy(void *userdata) {
    logfile << time(nullptr) << " destroy\n";

    logfile.close();
    delete(cache_data);
}

/**
 * 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)
{
    SET_TIMER;
    logfile << timer << " ioctl\n";

    for (vector<Block*>::iterator it = cache_data->cache.begin();
            it != cache_data->cache.end(); ++it)
    {
        if ((*it)->block_num != -1)
        {
            logfile << get_virtual_path((**it).file_path) << " " << (**it).block_num
                << " " << (**it).num_of_accesses << endl;
        }
    }
    return 0;
}

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

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

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

int main(int argc, char* argv[]) {
    try {
        if (argc != 5)
        {
            throw MainError();
        }
        cache_data = new Cache(argv);

        logfile.open("/" + string(cache_data->rootdir) +"/" + cache_data->logfile_name, ios::app);

        init_caching_oper();

        argv[1] = argv[2];
        for (int i = 2; i < (argc - 1); i++) {
            argv[i] = NULL;
        }
        argv[2] = (char*) "-s";
        argc = 3;

        int fuse_stat = fuse_main(argc, argv, &caching_oper, cache_data);
        return fuse_stat;
    }
    catch(MainError& e)
    {

        cout << "usage:  CachingFileSystem rootdir mountdir numberOfBlocks blockSize" << endl;
        free(cache_data);
        exit(0);
    }
    catch (exception &e) {
        cerr << "System Error " << e.what() << endl;;
        free(cache_data);
        exit(0);
    }
}
```

4 Makefile

```
CC = g++ -Wall
FLAG = -std=c++11 -D_FILE_OFFSET_BITS=64
FUSE = ‘pkg-config fuse --cflags --libs‘
LIBSRC = CachingFileSystem.cpp

all: CachingFileSystem

CachingFileSystem.o: CachingFileSystem.cpp
    g++ -std=c++11 -Wall -c -fmessage-length=0 -D_FILE_OFFSET_BITS=64 ‘pkg-config fuse --cflags‘ CachingFileSystem.cpp

CachingFileSystem: CachingFileSystem.o
    g++ -Wall -std=c++11 -D_FILE_OFFSET_BITS=64 CachingFileSystem.o ‘pkg-config fuse --libs‘ -o CachingFileSystem

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

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

clean:
    $(RM) CachingFileSystem.o CachingFileSystem *~ *core


.PHONY: CachingFileSystem CachingFileSystem.o
```