Gebze Technical University

Department of Computer Engineering

CSE 312 /CSE 504

Operating Systems Spring 2022

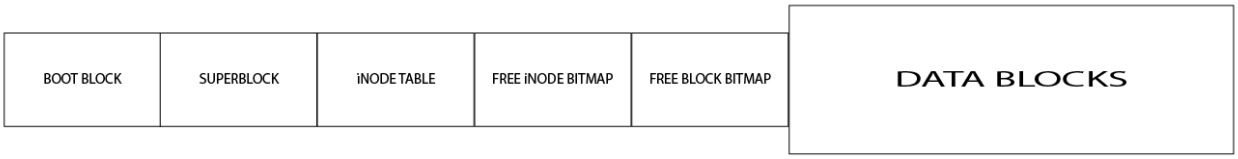
HW3

Due Date: May 31th. 2022

File Systems

Đuro Radusinović

UNIX version 6/7 file system's structure from textbook is followed during implementation. Regular file, directory and root directory itself are a same base file in its core. Main functionality is executed via methods from *memoryManagement.h,* implemented in *memoryManagement.c.* It's functionality is to abstract memory and make implementing methods such as *rmdir, mkdir, dir, write, read etc.* easier. Maximum files size depends on block size and implementation of indirect addressing makes possible for (9 + 512)*blockSize maximum size of the file. The last block is used as a block which stores block numbers pointing to real data. Each block is exactly 2bytes so it is possible to store up to 512 entires like this in a file system using 1024KB as block size.



Part1.

Part1.1

Directory structure contains inode number and a filename. Filename, as defined by UNIX v6/v7 has 14 characters with no trailing zero, meaning, each file name has at most 14 characters and if does have maximum number of characters, 14 then there is no trailing '\0' at the end of course. To do this, some string functions with respect to these limitations are written in *utils.h*. Directory structure and directory entries are as following

```
typedef struct DirectoryEntry{
    char filename[FILENAME_LENGTH];
    uint16_t inodeNumber;
}DirectoryEntry;
```

Directory structure is itself a file and files in UNIX v6/v7 are implemented using I-nodes. So, the structure of the directory itself would use Inode itself:

```
typedef struct Inode{
    FileAttributes attributes;
    FileType type;
    uint16_t diskBlocks[INODE_DB_N];
}Inode;
```

Distinguishing between a Regular file and a Directory file is done through Inode's **FileType** entry which itself is just a simple enumeration.

```
typedef enum FileType{DIRECTORY, REGULAR_FILE}FileType;
```

Another structure defining directory overall is **FileAttributes.** It stores last modification times, creation times and importantly file's size.

```
typedef struct FileAttributes{
    uint32_t fileSize;
    time_t timeCreation;
    time_t timeLastModification;
}FileAttributes;
```

Part1.2

To manage the file system, of course all data and all the inodes cannot be taken from the disk and manipulated in the RAM. Of course, since the file size of the system is just 16Mb, it would technically be possible to do that in today's computers though in the time when the file system implemented here existed that definitely would not be possible. This then requires a way to track free memory. Tracking free memory in this specific project is implemented through bitmaps. There are 2 bitmaps in total. One is for tracking free inodes and the other for tracking free blocks. Functions to manipulate the bitmap itself are provided through again implementation in *utils (utils.h, utils.c).* They are stored right after the table of inodes on the disk.

Part1.3

Super block is initialized at the beginning. Some calculations frequently used are directly provided through the super block itself. This provides better readability and conveniency. Root directory itself is stored in the second block, after the super block. Super block itself stores only initial size of the block entry.

```
typedef struct SuperBlock{
    uint32_t fileSystemSize;
    uint32_t blockSize;
    uint32_t usableBlockSizeInodeTable;

    uint16_t nBlocks;
    uint16_t nEmptyBlocks;
    uint16_t nFullBlocks;

    uint16_t nInodes;
    uint16_t sizeInode;
    uint16_t nEmptyInodes;
    uint16_t nFullInodes;
    uint32_t blockCountInodeTable;

    uint32_t blockCountInodeBitmap;
    uint32_t blockCountBlockBitmap;

    uint32_t firstDataBlock;
    uint32_t maxFileSizeDirectAddresing;
    uint32_t maxFileSizeIndirectAddresing;
}SuperBlock;
```

Part1.4

Function names that handle the operations file system provides are in fileSystem.h

```
#ifndef FILE_SYSTEM_H
#define FILE_SYSTEM_H
#include "utils.h"

bool dir(const char* path, FILE* fp);

bool mkdir(const char* path, FILE* fp);

bool rmdir(const char* path, FILE* fp);

void dumpe2fs(FILE* fp);

bool write(const char* path, const char* srcFile, FILE* fp);

bool read(const char* path, const char* destFileName, FILE* fp);

bool del(const char* path, FILE* fp);

#endif
```

These function extensively use functions provided by memoryManagement.h where the main abstraction logic for the file system is implemented. Method it provides are:

```
Inode createInode(FileType fileType);
DirectoryEntry createDirectoryEntry(uint16_t inodeNumber, const char* filename);

void syncInode(uint16_t inode_num, Inode modifiedInode);
void newInode();

void writeInode(uint16_t i, const Inode* inode, const char* fileSystemName);
void writeInodeOptimized(uint16_t i, const Inode* inode,  FILE* fp);//opening and closing the file is done by caller
void writeToBlock(uint16_t block_num, uint32_t offset, const void* src, uint16_t num_bytes, const char* fileSystemName);
void writeToBlockOptimized(uint16_t block_num, uint32_t offset, const void* src, uint16_t num_bytes, FILE* fileSystemName);//opening and
void fWriteToBlockOptimized(uint16_t block_num, uint32_t offset, const void* src, uint16_t num_bytes, int nmemb, FILE* fp);
void writeEmptyDirectory(uint16_t inode_num, const char* fileSystemName);
void writeToFile(uint16_t inode_num, const void* src, size_t size, const char* fileSystemName);//best to make size factor of 2
void writeToFileOptimized(uint16_t inode_num, const void* src, size_t size, FILE* fp);//best to make size factor of 2
void writeUsingDirectAddressing( Inode* inode, int newFileSize, const void* src, size_t size, FILE* fp);
void writeUsingIndirectAddressing( Inode* inode, int newFileSize, const void* src, size_t size, FILE* fp);//use factor of 2 for size whe
void writeToIndirectBlock( int indirectlyAddresedBlock, const Inode* inode, int newFileSize, const void* src, size_t size, FILE* fp);//a
int writeInitNewDirectory(uint16_t parentInode, FILE* fp);
int writeInitNewRegularFile(uint16_t parentInode, FILE* fp);//returns new inode number
void writeDirectoryEntry( const DirectoryEntry* directoryEntry, uint16_t inodeNumber, const char* fileSystemName );
void writeDirectoryEntryOptimized( const DirectoryEntry* directoryEntry, uint16_t inodeNumber, FILE* fp );
void syncDirectoryEntryOptimized( const Inode* parentInode, const DirectoryEntry* directoryEntry, FILE* fp );
int addNewDirectoryEntryOptimized(uint16_t inodeNumber, const char* fileName, FileType fileType, FILE* fp);

void readInode(FILE* fp, Inode* inode);
void readFromBlock(uint16_t blockNumber, uint32_t offset, void* dest, size_t size, const char* fileSystemName);
void readFromBlockOptimized(uint16_t blockNumber, uint32_t offset, void* dest, uint16_t size, FILE* fp);

Inode getInode( uint16_t inode_num, const char* fileSystemName );
Inode getInodeOptimized(uint16_t inode_num, FILE* fp);
uint16_t getInodeBlock(uint16_t inode_num);
uint32_t getInodeBlockOffset(uint16_t inode_num);
int getFreeBlockPosition(const Inode* inode);
int getPartiallyFilledBlockNumber(const Inode* inode);//returns last, partially filled block. When no blocks allocated returns -1
int getBlockNumberFromIndex(const Inode* inode, int i, FILE* fp);//filesystemname required since it can be indirectly addresses which w
int getBlockNumberFromIndirectBlock(const Inode* inode, int blockNumber, FILE* fp);//assumes previous check for boundaries
int getDirectoryEntryInodeNumber(const Inode* parentInode, const char* targetFileName, FILE* fp);
int getDirectoryEntryBlockIndex(int directoryEntryIndex);
int getDirectoryEntryBlockOffset(int directoryEntryIndex);
Inode getDirectoryEntryInode(int i, const Inode* inode, FILE* fp);
int getInodeByPath(char* path, FILE* fp);//returns inode number of directory, -1 otherwise
```

```
int getInodeByPath(char* path, FILE* fp);//returns inode number of directory, -1 otherwise
DirectoryEntry getDirectoryEntry( uint16_t blockNumber, int offset, const char* fileSystemName );
DirectoryEntry getDirectoryEntryFromPosition(const Inode* parentInode, int i, FILE* fp);
DirectoryEntry getDirectoryEntryFromFilename(const Inode* parentInode, const char* filename, FILE* fp);
DirectoryEntry getDirectoryEntryOptimized( uint16_t blockNumber, uint32_t offset, FILE* fp );
int getLastIndirectBlockNumber(const Inode* inode, FILE* fp);//assumes previous check for indirect addressing
int getLastBlockNumberFromIndirectBlock(int indirectBlock, FILE* fp);//assumes previous check for indirect addressing

void printDirectoryContent(uint16_t inodeNumber, const char* fileSystemName);
void printDirectoryContentOptimized(uint16_t inodeNumber, FILE* file);

void claimBlock(uint16_t block_num);
void claimInode(uint16_t inode_num);
void unclaimBlock(uint16_t block_num);
void unclaimInode(uint16_t inode_num);
void unclaimFile(int inodeNumber, FILE* fp);
int getAndClaimEmptyBlock();//gets and claims an empty block
int getAndClaimEmptyInode();//gets and claims an empty block
void syncAndUnclaimInode(int inodeNumber, const Inode* inode, FILE* fp);
void invalidateDirectoryEntry(const char* path , FILE* fp);

uint32_t countBlocks(const Inode* inode);
int countIndirectBlocks(int indirectBlock, FILE* fp);//assumes previous check for indirect addressing of the inode
uint32_t countDirectoryEntries(const Inode* inode);

int isInodeClaimed(uint16_t inode_num);
bool isIndirectlyAddreessed(const Inode* inode);

int fillPartialFilledBlock(int blockNumber, const Inode* inode,const void* src, size_t size, FILE* fp);
int fillNewBlocks(Inode* inode, int newBlocksCoount, const void* src, size_t size, FILE* fp);
int fillPartialFilledBlockDirectAddressing(const Inode* inode, const void* src, size_t size, FILE* fp);
int fillPartialFilledBlockIndirectAddressing(int indirectlyAddresedBlock, const Inode* inode, const void* src, size_t size, FILE* fp);
int fillNewBlocksIndirectAddressing(int indirectBlockNumber, int newBlocksCoount, const void* src, size_t size, FILE* fp);

bool fileNameCmp(const char* fn1, const char* fn2);
void fileNameCopy(char* dest, const char* src);
int fileNameLength(const char* fn);

bool containsFile(uint16_t inodeNumber, const char* filename, FILE* fp);
void printInode(const Inode* inode);
void printSuperBlock();
void printFileName(const char* filename);
```

Part2

In order to create a file system makeFileSystem related files (makeFileSystem.h, makeFileSystem.c) provide functions that are used for this. Also, some other basic functionality that is not directly related to filesystem's operations is given in here.

```
#ifndef MAKE_FILE_SYSTEM_H
#define MAKE_FILE_SYSTEM_H
#include <stdio.h>

void createFileSystem(int blockSize, const char* fileSystemName);//creates the file system and writes an empty file system to a file
void mountFileSystem(const char* fileSystemName);//reads the whole file system from give file
void syncFileSystem(const char* fileSystemName);//updates gobal variables like superblock and bitmaps
void unmountFileSystem(const char* fileSystemName);
void unmountFileSystemOptimized(FILE* fp);
void printFileSystemInfo();

#endif
```

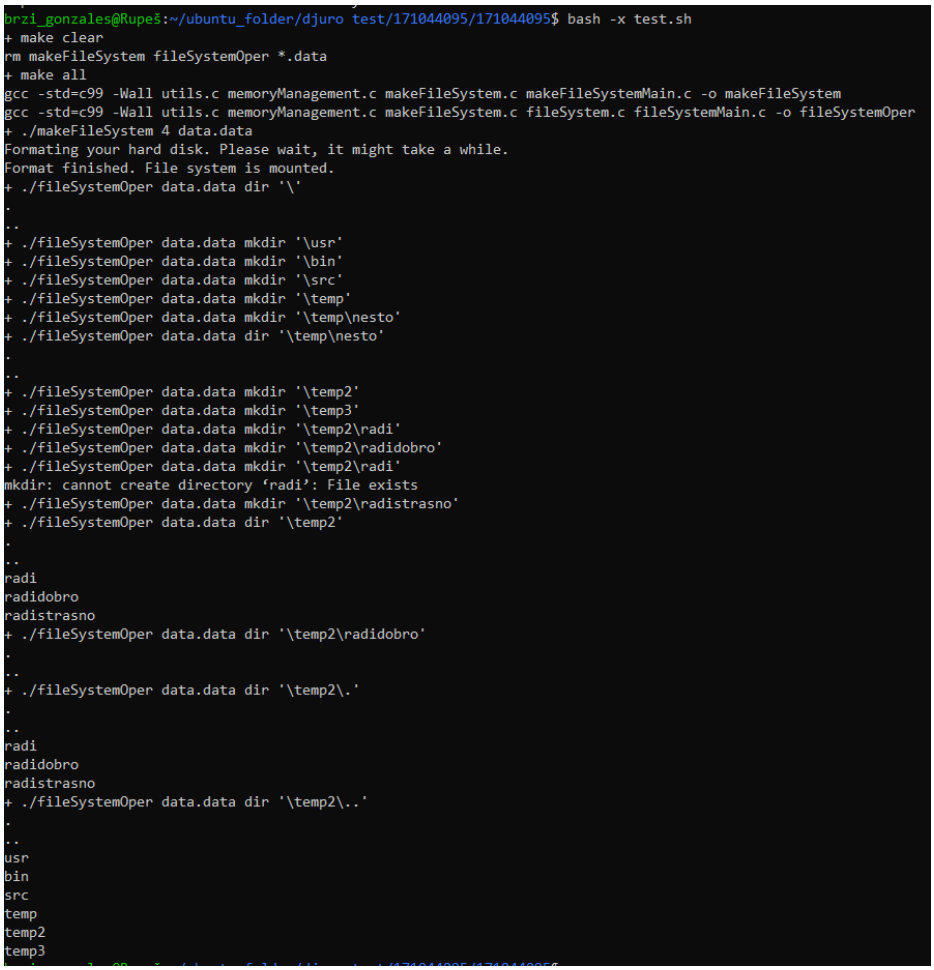
When mounting the system, only super block, free inode and free block bitmaps are loaded. All operations the filesystem is capable of performing are done through manipulating those 3 variables while using methods from memoryManagement.h.

Part3

Methods required are provided through *fileSystem.h.* There are four main test files *test.sh*, *test2.sh*, *test3.sh* and *test4.sh.* In *test4.sh* you will find test from the .pdf's homework.

Test results:

test.sh



test2.sh

```
brzi_gonzales@Rupeš:~/ubuntu_folder/djuro test/171044095/171044095$ bash -x test2.sh
+ make clear
rm makeFileSystem fileSystemOper *.data
+ make all
gcc -std=c99 -Wall utils.c memoryManagement.c makeFileSystem.c makeFileSystemMain.c -o makeFileSystem
gcc -std=c99 -Wall utils.c memoryManagement.c makeFileSystem.c fileSystem.c fileSystemMain.c -o fileSystemOper
+ ./makeFileSystem 4 data.data
Formating your hard disk. Please wait, it might take a while.
Format finished. File system is mounted.
+ ./fileSystemOper data.data dir '\'
.
..
+ ./fileSystemOper data.data mkdir '\usr'
+ ./fileSystemOper data.data mkdir '\bin'
+ ./fileSystemOper data.data mkdir '\bin\sth'
+ ./fileSystemOper data.data mkdir '\bin\temp'
+ ./fileSystemOper data.data mkdir '\bin\nice'
+ ./fileSystemOper data.data mkdir '\src'
+ ./fileSystemOper data.data dir '\bin'
.
..
sth
temp
nice
+ ./fileSystemOper data.data rmdir '\bin\nice'
+ ./fileSystemOper data.data dir '\bin'
.
..
sth
temp
+ ./fileSystemOper data.data dir '\'
.
..
usr
bin
src
+ ./fileSystemOper data.data rmdir '\bin'
+ ./fileSystemOper data.data dir '\bin'
dir: cannot access '\bin': No such file or directory
+ ./fileSystemOper data.data dir '\'
.
..
usr
src
brzi_gonzales@Rupeš:~/ubuntu_folder/djuro test/171044095/171044095$
```

test3.sh (tests mainly indirect addressing, for larger files)

```
brzi_gonzales@Rupeš:~/ubuntu_folder/djuro test/171044095/171044095$ bash -x test3.sh
+ make clear
rm makeFileSystem fileSystemOper *.data
+ make all
gcc -std=c99 -Wall utils.c memoryManagement.c makeFileSystem.c makeFileSystemMain.c -o makeFileSystem
gcc -std=c99 -Wall utils.c memoryManagement.c makeFileSystem.c fileSystem.c fileSystemMain.c -o fileSystemOper
+ ./makeFileSystem 1 data.data
Formating your hard disk. Please wait, it might take a while.
Format finished. File system is mounted.
+ ./fileSystemOper data.data dir '\'
.
..
+ ./fileSystemOper data.data mkdir '\usr'
+ ./fileSystemOper data.data mkdir '\bin'
+ ./fileSystemOper data.data mkdir '\bin\sth'
+ ./fileSystemOper data.data mkdir '\bin\temp'
+ ./fileSystemOper data.data mkdir '\bin\nice'
+ ./fileSystemOper data.data write '\bin\works' somefile.txt
+ ./fileSystemOper data.data read '\bin\works' thisWorks.txt
+ ./fileSystemOper data.data write '\bin\works2' somefile2.txt
+ ./fileSystemOper data.data read '\bin\works2' thisWorks2.txt
+ ./fileSystemOper data.data dir '\bin'
.
..
sth
temp
nice
works
works2
+ cmp thisWorks.txt somefile.txt
+ cmp thisWorks2.txt somefile2.txt
+ ./fileSystemOper data.data dumpe2fs
Block count: 16384
Free blocks: 15550
Used blocks: 834
Block size: 1024
Number of files and directories: 8
brzi_gonzales@Rupeš:~/ubuntu_folder/djuro test/171044095/171044095$
```

test4.sh

```
brzi_gonzales@Rupeš:~/ubuntu_folder/OS/HWK3$ sudo bash -x test4.sh
+ make clear
rm makeFileSystem fileSystemOper *.data
+ make all
gcc -std=c99 -Wall utils.c memoryManagement.c makeFileSystem.c makeFileSystemMain.c -o makeFileSystem
gcc -std=c99 -Wall utils.c memoryManagement.c makeFileSystem.c fileSystem.c fileSystemMain.c -o fileSystemOper
+ ./makeFileSystem 1 fileSystem.data
Formating your hard disk. Please wait, it might take a while.
Format finished. File system is mounted.
+ ./fileSystemOper fileSystem.data mkdir '\usr'
+ ./fileSystemOper fileSystem.data mkdir '\usr\ysa'
+ ./fileSystemOper fileSystem.data mkdir '\bin\ysa'
dir: cannot access '\bin\ysa': No such file or directory
+ ./fileSystemOper fileSystem.data write '\usr\ysa\file1' linuxFile.a
+ ./fileSystemOper fileSystem.data write '\usr\file2' linuxFile.a
+ ./fileSystemOper fileSystem.data write '\file3' linuxFile.a
+ ./fileSystemOper fileSystem.data dir '\'
.
..
usr
file3
+ ./fileSystemOper fileSystem.data del '\usr\ysa\file1'
+ ./fileSystemOper fileSystem.data dumpe2fs
Block count: 16384
Free blocks: 15664
Used blocks: 720
Block size: 1024
Number of files and directories: 5
+ ./fileSystemOper fileSystem.data read '\usr\file2' linuxFile2.a
+ cmp linuxFile2.a linuxFile.a
brzi_gonzales@Rupeš:~/ubuntu_folder/OS/HWK3$
```