Assignment 4 Design Document
CMPS 111, Spring 2016

1 Goals

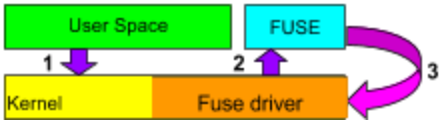
The goal of this assignment is to use the FUSE file system framework to implement a simple FAT-based file system in user space.This assignment will be run in user space and therefore, will give us experience in extending operating system functionality with a user program. Extending an operating system in such a way presents unique challenges, in that we must implement what is normally kernel logic without access to kernel level operations.

It is important that we learn how the operating system should interact with user space, since it restricts user programs to prevent them from crashing the machine or messing with any other programs that are in the kernel space, which teaches us about data protection.

2 Assumptions

We will be using a FAT (File Allocation Table) based file system and integrating it into FUSE; a user-space file system interface. For the disk structure, one can assume that the disk will be partitioned into 1 KB blocks. The file allocation table will occupy a number of these blocks. Since each block pointer in the FAT is 4 bytes, each block in the FAT can hold 256 block pointers, therefore we will need the number of blocks in the file systems divided by 256 FAT blocks. For the directories we will assume that each directory entry has a length of 64 bytes. The directory entries will carry the information specified in the assignment, but the only flag each directory entry will have to contain is the directory flag, which is a single bit that will indicate if the directory entry is for a regular file or a directory.

This is how we believe the fuse system works.



We are also assuming that if the amount of bytes to be read in read() goes over the amount actually present in the file, it is not an error and it will fill up the rest of the buffer with zeros.

3 Design

The first step to making the file system would be to create a disk image as specified in the assignment. It will contain the superblock, the file allocation table, and data blocks.

Superblock

The disk is divided into 1KB blocks and contains N blocks in total. The 0th block is reserved for the superblock, which will contain; the magic number, which is defined to be 0x2345beef, in the 0th word index, the total number of blocks in the file system (N) in the 1st word index, the number of blocks in the file allocation table (k or $1 + \frac{N}{256}$) in the 2nd word index, the block size which will be 1024 bytes for our file system in word index 3, and the starting block of the root directory in the last word index of 4. We will fill the rest of the super block with empty space.

File Allocation Table

The file allocation table will follow the superblock up until the kth block with k equaling $1 + \frac{N}{256}$. The first step to filling in the file allocation table would be to fill all the blocks that are already in use by the superblock and allocation table with a -1, indicating that they are invalid and can't be used. The root directory will be marked with a -2, which indicates that this block is the end of a file, and the root block is the one and only block needed for an empty file system. All the other blocks will be marked with zero because they are free. As information is written into the FAT the blocks will either point to other blocks that continue the file until it reaches an end of file, a -2. Any blocks that are in the file allocation table but aren't being used will be set to a -1.

Fuse:

To traverse through the file system we will be overriding Fuse operations with are own. We need to be able to use basic functionality and the following operations are the ones we need to implement to be able use them. Below are the main operations we are overriding and in the pseudocode we list the operations they are used for.

Operations we need to implement:

- Getattr
- Mkdir
- Unlink
- Rmdir
- Rename
- Open
- Read
- Write
- Release
- Readdir
- Opendir
- Main
- Create

4 Pseudocode

****

CreateDiskImage

Disk Image that will be created in the underlying file system. Split up into superblock, file allocation table, and file data (includes directory entries).

****

```
CreateDiskImage(imageName, nBlocks):
    magicNumber = 0x2345beef
    blockSize = 1024
    nBlocksInFileAllocationTable = (nBlocks/(blockSize/4))+1
    rootBlock = nBlocksInFileAllocationTable+1
    fileDescriptor = Open imageName as write only, create if doesn't exist
    //Creation of superblock
    write(fileDescriptor, address of magicNumber, 4)     //Write magic number 0x2345beef
    write(fileDescriptor, address of nBlocks, 4)   //Write number of Blocks to fileDescriptor
    write(fileDescriptor, address of nBlocksInFileAllocationTable, 4)
    write(fileDescriptor, address of blockSize, 4) //Write block size of 1024 to fileDescriptor\
    write(fileDescriptor, address of rootBlock, 4)
    //fill out rest of superblock
    emptyBuffer = empty array of size blockSize-(4*5)
    write(fileDescriptor, emptyBuffer, blockSize-20)
    // Fill in file allocation table
    allocationTable = array of size nBlocks
    for i from 0 to rootBlock
        allocationTable[i] = -1 // mark blocks in table and superblock as invalid
    allocationTable[rootBlock] = -2 // root block is occupied by root directory
    for i from rootBlock+1 to nBlocks
        allocationTable[i] = 0 //mark all other blocks as free
    write(fileDescriptor, allocationTable, nBlocks)
    //Fill in rest of last block in file allocation table
    buffer = array of size ((nBlocksInFileAllocationTable * blockSize) - nBlocks
    for i from 0 to length of buffer
        buffer[i] = -1 //mark rest of blocks invalid, since they don't exist
    write(fileDescripotor, buffer, size of buffer)
    // Fill out rest of disk image with empty data
    buffer = empty array of size blockSize
    for i from rootBlock to nBlocks
        write(fileDescriptor, emptyBuffer, blockSize)
```

Fuse Implementations

These are the following functions used in the implementation of fuse

---

Global Variables:

```
int diskImage
uint32_t nBlocks
uint32_t nBlocksInFileAllocationTable
uint32_t blockSize
int rootBlock
int * allocationTable
char * prevPath
```

Main(argc, argv):

```
if (argc < 2)
        error("Insufficient argument")
diskImage = open(argv[1], READ-WRITE)
if (open of diskImage failed)
        error("Unable to open: "+argv[1])
buffer = array of size 20 bytes (5 ints)
bytesRead=read(diskImage, buffer, 20)
if (bytesRead != 20 OR buffer[0] != 0x2345beef)
        error("Disk Image is not of proper format")
nBlocks = buffer[1]
nBlocksInFileAllocationTable = buffer[2]
blockSize = buffer[3]
rootBlock = buffer[4]
```

Struct DirectoryEntry:

```
Char filename [24]
Size_t creationTime
Size_t modificationTime
Size_t accessTime
Uint32_t fileLength
Int startBlock
Int flags
Int unused
```

Intermediate Functions

These following functions are used in the rest of the functions to traverse and do several operations in the file structure

---

FindOpenBlock()

Looks for an open space within the blocks. Checks to make sure there's enough space for a file to be written to, and returns the open block

---

```
FindOpenBlock():
        block = rootBlock
        while (allocationTable[block] != 0)
                block = block + 1
        If (block < nBlocks)
                byteArray [1024]
                for every entry in byteArray
                        byteArray[count] = 0;
                result=lseek(diskImage, block * blockSize, SET)
                result=write(diskImage, buffer, blockSize)
                return block
        else error("no empty blocks")
```

TraverseDirectoryTree

Goes through each of the blocks to find the block and file data specified by path

---

```
TraverseDirectoryTree(path):
        if((length of string is 1 and is equal to "/") == 0)
                if(offset isn't null)
                        return root directory
        block = rootBlock
        path = trim leading slash from path
        node = leading node of path
        directoryEntry = null
        while (path isn't null) //when path is null, we have found the start block of designated file
                offset = lseek(diskImage, blockSize * block, SET)
                buffer = array of size blockSize
```

```
        result=read(diskImage, buffer, blockSize)
        if (result < 0)
                error("Reading from disk image failed")
                return null
        i=0
        while (i < blockSize)  //find the given path node in the current block
                directoryEntry=GetDirectoryEntry(address of buffer[i])
                if (directoryEntry exists AND node equals directoryEntry.fileName)
                        if (directoryEntry.flags == DIRECTORY)
                                block = directoryEntry.startBlock
                                break loop
                        else
                                error("Not a directory")
                                return null
                i = i + 64
        if (i >= blockSize AND allocationTable[block] > 0) //reached end of this block, but
                                                                there's another
                block = allocationTable[block]
        else if (i < blockSize)
                path = trim leading slash from path, if no slash set to null
                node = leading node of path
        else
                error("Path not found")
                return null
    return directoryEntry
```

WriteAllocationTabletoDisk()

Writes the allocation table to disk

```
WriteAllocationTableToDisk():
    offset=lseek(diskImage, blockSize, SEEK_SET)
    result=write(diskImage, allocationTable, nBlocks*4)
    if (result < 0)
            error("Writing to disk image failed")
```

Directory Entry Functions

These following functions are used to read and write directory entries

---

```
//will receive a sequence of bytes and organize them into a DirectoryEntry structure
GetDirectoryEntry(array):
        dirEntry = new DirectoryEntry
        //read filename string into structure
        for i from 0 to 24
                dirEntry.fileName[i] = array[i]
        dirEntry.creationTime = *(size_t *)array[24]
        dirEntry.modificationTime = *(size_t *)array[32]
        dirEntry.accessTime = *(size_t *)array[40]
        dirEntry.fileLength = *(uint32_t*)array[48]
        dirEntry.startBlock = *(int *)array[52]
        dirEntry.flags = *(int *)array[56]
        dirEntry.unused = *(int *)array[60]
        return dirEntry
```

WriteDirectoryEntryToBuffer

Writes the directory entry struct to the buffer

```
WriteDirectoryEntryToBuffer(dirEntry, byteBuffer):
        intBuffer = (int*)byteBuffer
        longBuffer = (size_t *)byteBuffer
        uintBuffer = (uint32_t *)byteBuffer
        for i from 0 to 24
                byteBuffer[i] = dirEntry.fileName[i]
        longBuffer[3] = dirEntry.creationTime
        longBuffer[4] = dirEntry.modificationTime
        longBuffer[5] = dirEntry.accessTime
        uintBuffer[12] = dirEntry.fileLength
        intBuffer[13] = dirEntry.startBlock
        intBuffer[14] = dirEntry.flags
        intBuffer[15] = dirEntry.unused
```

---

removeDirectoryEntry

Removes the directory entry specified by path

```
removeDirectoryEntry (Path):
        check = traverseDirectoryTree(path)
        Previous = offset in block of check's directory entry
        Next = offset in block of entry after check
        currentEntry = check
        currentBlock = block containing currentEntry's directory entry
        done = false
        Buffer = array of size blockSize
        read currentBlock into memory
        while (not done)
                tempBuf = array of size 64
                if (next < blockSize)
                        nextDirectoryEntry = getDirectoryEntry(address of buffer[next])
                        if (nextDirectoryEntry exists)
                                Write directory entry to tempBuf
                        else
                                Fill tempBuf with zeros
                                done = true
                        write tempBuf to block at offset previous
                        previous = next
                        next = next + 64
                else if (allocationTable[currentBlock] > 0)
                        next = 0
                        nextBlock = allocationTable[currentBlock]
                        nextbuffer = array of size blockSize
                        read nextBlock's disk block into nextbuffer
                        nextDirectoryEntry = getDirectoryEntry(address of nextbuffer[next]);
                        if(nextDirectoryEntry exists)
                                Write directory entry to tempBuf
                        else
                                Fill tempBuf with zeros
                                done = 1
                        write tempBuf to current block at offset previous
                        previous = 0
                        next = next + 64
                        currentBlock = nextBlock
```

buffer = nextbuffer

Directory Functions

These following functions are used to be able traverse and read through directories

---

****

mkdir()

Creates a new directory by creating a new directory entry and writing to the disk used for mkdir operation.

****

mkdir(path):

```
origLoc =0
directoryEntry = TraverseDirectoryTree(path except trailing node) //get directory that will
                                                                   contain our new directory
block = newdirectoryEntry.startBlock
while (allocationTable[block] > 0) //get to block where we can write
        block = allocationTable[block]
offset = lseek(diskImage, blockSize * block, SEEK_SET)
buffer = array of size blockSize
result=read(diskImage,buffer, blocksize)
if(result < 0)
        error “can’t read diskImage”
read(diskImage, buffer, blockSize)
count=0
while (buffer[count] isn’t empty AND count < blockSize) //get to free space in block
count = count+ 64
if (count >= blockSize) //there is no free space in block, make a new one
        allocationTable[block] = findOpenBlock()
        block = allocationTable[block]
        allocationTable[block] = -2
        buffer = empty buffer of size blockSize
        count = 0
dirEntry = new DirectoryEntry
dirEntry.fileName = node
dirEntry.creationTime = time of now
dirEntry.modificationTime = dirEntry.creationTime
dirEntry.accessTime = dirEntry.creationTime
dirEntry.fileLength = 0
dirEntry.startBlock = findOpenBlock()
```

```
        dirEntry.flags = DIRECTORY
        allocationTable[dirEntry.startBlock] = -2
        WriteAllocationTableToDisk()
        WriteDirectoryEntryToBuffer(dirEntry, address of buffer[count])
        offset = lseek(diskImage, blockSize * block, SEEK_SET)
        result=write(diskImage,buffer, blockSize)
        If the result is less than zero
                Throw a “can’t read diskImage” error
                return -1
newDir.fileLength = new.fileLength + 64
newDir.modificationTime = time of now
newDir. accessTime = newDir.modificationTime

If (origLoc > 0)
        tempBuf = DirectoryEntry
        writeDirectoryEntryToBuffer(newDir, tempBuf)
        offset=lseek(diskImage, origLoc, SEEK_SET)
        result = write(diskImage, temoBuf, DirectoryEntry)
return 0
```

****

opendir()

Opens directory and shouldn’t have to check for permissions. Used for the cd operation

****

```
opendir(path, fi):
        Print “opendir”
        If (path matches “..”)
                Path = prevpath
        Int origloc
        Directoryentry check = traverseDirectoryTree(path, &origLoc);
        If (check is null AND check.flags does not equal IS_DIRECTORY)
                Return -ENOENT
        Check.accestime = time of now;
        If (origloc > 0)
                Char tempBuf
                writeDirectoryEntrytoBuffer(check, tempbuf)
                Offset = lseek(diskImage, origLoc, SEEK_SET);
                Result = write(diskImage, tempBuf, sizeof(struct DirectoryEntry));
        Prevpath = dirname(path);
        Return 0
```

****

readdir()

Reads directory entries and used for the ls operation

****

```
readdir(path,buf,filler,offset,fi)
        If (“..” == path) path = prevPath
        DirectoryEntry * check = TraverseDirectoryTree(path)
        If (check == NULL || check.flags != Directory)
                Return -enoent
        filler(buf, ".", NULL, 0);
        filler(buf, "..", NULL, 0);

        block = check.startBlock
        done = 0
        While (done == 0)
                offset = lseek(diskImage, blockSize * block, SET)
                buf = array of size blockSize
                result=read(diskImage, buf, blockSize)
                if (result < 0)
                        error(“Reading from disk image failed”)
                        return ERONENT
                i=0
                while (i < blockSize)  //find the given path node in the current block
                        directoryEntry=GetDirectoryEntry(address of buf[i])
                        filler(buf, directoryEntry.fileName, NULL, 0);
                        i = i + 64
                if (i >= blockSize AND allocationTable[block] > 0) //reached end of this block, but
                                                                    there’s another
                        block = allocationTable[block]
                Else done = 1;

        return 0
```

****

rmdir()

Removes directory and all files in it and updates the path. Used to rm operation for directories

****

```
rmdir(path):
        DirectoryEntry* check = TraverseDirectoryTree(path)
        filename = path of names of files in this directory
```

```
block = check.startBlock
While (done == 0)
        offset = lseek(diskImage, blockSize * block, SET)
        buffer = array of size blockSize
        result=read(diskImage, buffer, blockSize)
        if (result < 0)
                error("Reading from disk image failed")
                return ENOENT
        i=0
        Done = 0
        while (i < blockSize)  //find the given path node in the current block
                directoryEntry=GetDirectoryEntry(address of buffer[i])
                if(directoryEntry.flags == DIRECTORY)
                        rmdir(path+directoryEntry.filename)
                Else
                        unlink(path+directoryEntry.filename)
                i = i + 64
        if (i >= blockSize AND allocationTable[block] > 0) //reached end of this block, but
                                                                there's another
                block = allocationTable[block]
        Else done = 1;
removeDirectoryEntry(check, check.startBlock)
WriteAllocationTableToDisk()
```

File Functions

These following functions are used to be able to do all the operations to traverse and manipulate files

---

****

create()

Creates and opens a file used for any creating file operations

****

```
create(path, mode, fi)
        origLoc= 0
        DirectoryEntry * check = TraverseDirectoryTree(path)
        If (check != NULL)
                Return open(path,fi)

        directoryEntry  = TraverseDirectoryTree(path except trailing node) //get directory that will
                                                                            contain our new directory
        block = directoryEntry.startBlock
        while (allocationTable[block] > 0) //get to block where we can write
                block = allocationTable[block]
        offset = lseek(diskImage, blockSize * block, SET)
        buffer = array of size blockSize
        read(diskImage, buffer, blockSize)
        i=0
        while (buffer[i] isn't empty AND i < blockSize) //get to free space in block
                i = i + 64
        if (i >= blockSize) //there is no free space in block, make a new one
                allocationTable[block] = FindOpenBlock()
                block = allocationTable[block]
                allocationTable[block] = -2
                buffer = empty buffer of size blockSize
                i = 0
        dirEntry = new DirectoryEntry
        dirEntry.fileName = node
        dirEntry.creationTime = TODO: get time of now
        dirEntry.modificationTime = dirEntry.creationTime
        dirEntry.accessTime = dirEntry.creationTime
        dirEntry.fileLength = 0
        dirEntry.startBlock = FindOpenBlock()
        dirEntry.flags = 0
```

```
        allocationTable[dirEntry.startBlock] = -2
        WriteAllocationTableToDisk()
        WriteDirectoryEntryToBuffer(dirEntry, address of buffer[i])
        offset = lseek(diskImage, blockSize * block, SET)
        write(diskImage, buffer, blockSize)
        WriteAllocationTableToDisk()
        Return open(path,fi)
```

****

open()

File open operation

****

```
open(path, fi):
        //Set the open bit and then write the directory entry
        origLoc =0
        DirectoryEntry * check = TraverseDirectoryTree(path)
        If (check == NULL || check.flags == IS_DIRECTORY)
                return -enoent
        Check.flags = OPEN //Set the open flag on
        check.accessTime = null
        if(origLoc > 0)
                tempBuf = size of Directory entry
                writeDirectoryEntrytoBuffer(check, tempBuf)
                offset=lseek(diskImage,origLoc,SEEK_SET)
                result=write(diskImage,tempBuf,
        Return 0
```

****

release()

Releases an open file

****

```
release(path, fi):
        //Set the open bit and then write the directory entry
        origLoc = 0
        DirectoryEntry * check = TraverseDirectoryTree(path)
        If (check == NULL !! Check.flags != OPEN)
                return -enoent
        Check.flags = CLOSE //Set the open flag on
        writeDirectoryEntrytoBuffer(check, check.startblock)
        Return 0
```

****

read()

Reads data from an open file and returns exactly the numbers of bytes requested except on an end of file or error

****

```
read (path, buffer, size, offset, fi):
        origLoc = 0
        DirectoryEntry * check = TraverseDirectoryTree(path)
        If (check == NULL || Check.flags != IS_OPEN)
                for (int count = 0; count < size; count++) buf[count] = 0;
                return -enoent
        If (offset > check .fileLength)
                for (int count = 0; count < size; count++) buf[count] = 0;
                return 0;
        bytesRead = 0
        currentBlock = check.startBlock
        offsetCount = offset
        while (offsetCount > blockSize AND allocationTable[currentBlock] > 0)
                offsetCount = offsetCount - blockSize
                currentblock = allocationTable[currentBlock]
        if (offsetCount > blockSize)
                for (int count = 0; count < size; count++) buf[count] = 0;
                return 0
        while (bytesRead < size AND bytesRead+offset < check.fileLength)
                oft=lseek(diskImage, currentBlock * blockSize+offsetCount, SEEK_SET)
                result=read(diskImage, buffer+bytesRead, blockSize-offsetCount)
                offsetCount = 0
                bytesRead = bytesRead + result
        if(currentBlock == -2) break
                currentBlock = allocationTable[currentBlock]
        If (bytesRead < size)
                For (int count = bytesRead; count < size; count++) buf[count] = 0;
        Return bytesRead
```

****

write()

Writes data to open file and returns the exact number of bytes requested except on an error

****

```
write (path, buffer, size, offset, fi):
        DirectoryEntry * check = TraverseDirectoryTree(path)
        If (check == NULL || Check.flags != OPEN)
                Return -enoent          //If write breaks write for case for file doesn't exist
        bytesWritten = 0
        currentBlock = check.startBlock
        offsetCount = offset
        While (offsetCount > blockSize)
                offsetCount = offsetCount - blockSize
                allocationTable[block] = FindOpenBlock()
                currentBlock = allocationTable[block]
                allocationTable[block] = -2

        while (bytesWritten < size AND bytesWritten+offset < blockSize)
                result=lseek(diskImage, currentBlock * blockSize+offsetCount, SET)
                result=write(diskImage, buffer+bytesWritten, blockSize-offsetCount)
                offsetCount = 0
                bytesWritten +=  result
                allocationTable[block] = FindOpenBlock()
                currentBlock = allocationTable[block]
                allocationTable[block] = -2

        If (bytesWritten < size)
                Return error //Error fo r max size
        WriteAllocationTableToDisk()

        Return bytesWritten
```

****

getattr()

Gets the attributes of the file

****

```
getattr(const char* path, struct stat *st):
        DirectoryEntry *file = TraverseDirectoryTree(path)
        If (file == NULL)
                Return -ENOENT
        st.st_atime = file.accessTime
```

```
        st.st_ctime = file.creationTime
        st.st_mtime = file.modificationTime
        st.st_size  = file.fileLength
        st.st_mode  = DEFFILEMODE
        if(file.flags is IS_DIRECTORY) //checks to see if flag is set to directory or not
                st.st_mode |= S_IFDIR
                else
                st.st_mode |= S_IFREG
        st.st_nlink = 1
        return 0
```

****

rename()

Renames a file

****

```
rename(path, newName)
        origLoc= 0
        DirectoryEntry* check = TraverseDirectoryTree(path)
        If (check == NULL)
                Return -ERONENT
        check.fileName = newName
        //write directory entry to disk
```

****

unlink()

Removes(unlinks) a file Used for the rm functions to remove files

****

```
unlink(path):
        DirectoryEntry*  check = TraverseDirectoryTree(path)
        Int nextblock = 0;
        Int currentblock = check.startblock
        While (nextblock != -2)
                nextblock = allocationTable[currentblock]
                allocationTable[currentblock] = 0
                Currentblock = nextblock
        removeDirectoryEntry(check, check.startBlock)
        WriteAllocationTableToDisk()
```