ICS143B Project3 File System - Preliminary Document

Anbang Xu(35086995)

1. Introduction

Design and implement a simple file system using ldisk (a file to emulate a physical disk). This file system includes create, destroy, open, close, read, write, lseek, directory, save and init operations.

2. Data Structure

a. PackableMemory

```
+-----+
| size |
+-----+
| memory |
+-----+
```

The PackableMemory is a structure having an integer - '"size" and a byte array - "memory" for storage. It's used to pack/unpack an integer to/from a byte array. We can treat it as an integer array.

b. I/O System(IOSystem)

```
+-----+
| ldisk | → | PackableMemory1 | → | PackableMemory2 | → ….. → | PackableMemoryN |
+-----+
```

The I/O System is a structure having a linked list of "PackableMemory". The size L indicates the length of linked list and the size B indicates the number of bytes per PackageMemory. I/O system presents disk as a linear sequence of blocks. We can treat it as ldisk[L][B], L is the number of logical blocks on ldisk and B is the block length (in bytes). In this project, both L and B are 64.

c. Open File Table Entry(OFTEntry)

```
+-----+
| buffer |
+-----+
| currentPosition |
+-----+
| index |
+-----+
| whichBlock |
+-----+
```

The OFTEntry is a structure have a buffer, currentPostion, index and whichBlock. "buffer" is a byte array of length L and it's used to store one of blocks read from file(read-ahead). "currentPosition" indicates the position of current file pointer. "index" indicates the index of file descriptor. "whichBlock" indicates which block in file descriptor current buffer stores. In this project, each file descriptor is split into 4 blocks and each block occupies one integer. The first block stores the length of file. The other three blocks store the indices of data blocks. Thus, "whichBlock" could be assigned to 0, 1, 2, corresponding to three different data blocks.

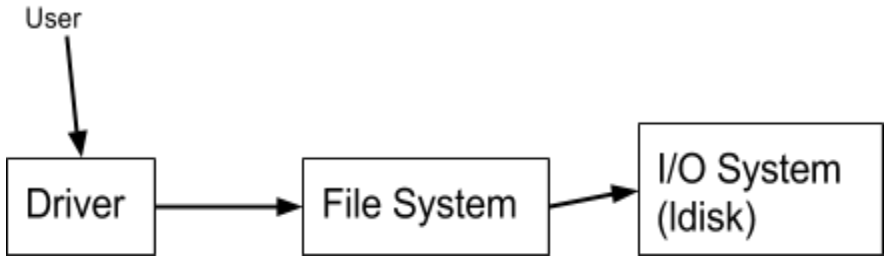
d. File System(FS)

```
+-----+
| IOSystem |
```

+-----+
| OFT | → | OFTEntry1 | → | OFTEntry2 | → ..... → | OFTEntryN |
+-----+
The File System is a structure having “IOSystem” and a linked list of “OFTEntry”. I describe the “IOSystem” and “OFTEntry” above. In this project, the OFT only has four entries, thus here N = 4.

3. System Architecture

a. Overall Organization



(1). I/O system: I/O system presents disk as a linear sequence of blocks. It has only two interfaces: read and write an entire block(B bytes). File System access ldisk using only these functions (no direct access to ldisk is allowed).
(2). File System: FS has a couple of user interfaces such as create, destroy, open, close, read, write, lseek, directory(list all files under input directory), init and save. The organization of file system includes:

A. Bit Map, which is kept in dedicated disk block 0
B. Directory, implemented as one regular file and organized as unsorted array of fixed-size slots. Each slot contains symbolic name(4 bytes max and almost use 4 bytes to store it) and index of descriptor(also use 4 bytes to store it)
C. File Descriptor, which is kept in dedicated k disk blocks. Each contains lengths(4 bytes) and disk map. Disk map is a fixed-list of max 3 disk blocks.
D. Data Block, which is used to store data.

(3). Driver: Driver processes user input and passes to FS. The user input can be either a set of commands such as a file or single shell command line.

b. Important function in IOSystem:
(1). Read Block
Input: block index(int), Output: data block(int[])
- Describe:
    A. read the corresponding block from ldisk based on block index
- What data structures may be changed? IOSystem

(2). Write Block
Input: block index(int), data block(int[])
- Describe:
    A. write data block into the corresponding block into ldisk
- What data structures may be changed? IOSystem

c. Important function in File System:
(1). Create File

Input: file name(string), Output: void
- Describe:
    A. find a free file descriptor
    B. find a free directory entry
    C. fill both entries
- What data structures may be changed? IOSystem, File System

(2). Destroy File
Input: file name(string), Output: void
- Describe:
    A. find a free file descriptor
    B. find a free directory entry
    C. fill both entries
- What data structures may be changed? IOSystem, File System

(3). Open File
Input: file name(string), Output: OFT entry index
- Describe:
    A. search directory to find index of file descriptor (i)
    B. allocate a free OFT entry (reuse deleted entries)
    C. fill in current position (0) and file descriptor index (i)
    D. read block 0 of file into the r/w buffer (read-ahead)
    E. adding a file length field (to simplify checking)
    F. adding whichblock filed(to record which block the buffer read from)
    G. return OFT index (j) (or return error)
- What data structures may be changed? IOSystem, File System, OFT

(4). Close File
Input: OFT entry index
- Describe:
    A. write buffer to disk
    B. update file length in descriptor
    C. free OFT entry
- What data structures may be changed? IOSystem, File System, OFT

(5). Read File
Input: OFT index(int), count(int)
- Describe:
    A. compute position in the r/w buffer
    B. copy from buffer to memory until
        a. desired count or end of file is reached:
            i. update current position, return status
        b. end of buffer is reached
            i. write the buffer to disk
            ii. read the next block
            iii. continue copying
- What data structures may be changed? IOSystem, File System, OFT

(6). Write File
Input: OFT index(int), count(int)
- Describe:
    A. compute position in the r/w buffer
    B. copy from memory into buffer until
        a. desired count or end of file is reached:
            i. update current pos, return status
        b. end of buffer is reached
            i. if block does not exist yet (file is expanding):
                1. allocate new block (search and update bit map)
                2. update file descriptor with new block number
            ii. write the buffer to disk block
            iii. continue copying
    C. update file length in descriptor
- What data structures may be changed? IOSystem, File System, OFT

(7). Lseek
Input: position, Output: status
- Describe:
    A. if the new position is not within the current block
        a. write the buffer to disk
        b. read the new block
        c. set the current position to the new position
    B. return status
- What data structures may be changed? IOSystem, File System, OFT

(8). List all files under directory
Input: directory, Output: void
- Describe:
    A. read directory file
    B. for each non-empty entry print file name
- What data structures may be changed? No

(9). Init(without parameter)
Input: void, Output: void
- Describe:
    A. initiate bitmap
    B. initiate directory
    C. initiate file descriptor for directory and file
- What data structures may be changed? IOSystem

(10). Init(with parameter) - Restore
Input: inputPath, Output: void
- Describe:
    A. restore ldisk from input file
- What data structures may be changed? IOSystem, File System, OFT

(11). Save
Input: outputPath, Output: void
- Describe:
    A. close all files
    B. write the ldisk into output file by binary format or text format
    C. clear Open File Table(OFT)
- What data structures may be changed? No

5. Pseudo Code

```
a. IOSystem.readBlock(int i) {
        PackableMemory pm = ldisk.get(i);
        int[] block = new int[16];
        for (int x = 0; x < 16; x++) {
                block[x] = pm.unpack(4 * x);
        }
        return block;
}

b. IOSystem.writeBlock(int i, int[] block) {
        boolean debug = true;
        PackableMemory pm = ldisk.get(i);
        for (int x = 0; x < block.length; x++) {
                pm.pack(block[x], 4 * x);
        }
}

c. FS.init() {
        // 1. initiate bitmap
        // file descriptor for directory
        for(int i = 0; i <= FILE_DESCRIPTOR_END_INDEX; i++){
                setBitMap(i);
        }
        // 2. initiate the directory
        initDirectory();
}

d. FS.create(String name) {
                // 1. find a free file descriptor
                int curBlockIdx = FILE_DESCRIPTOR_START_INDEX;
                int curReference = 0;
                int[] fdBlock = io.readBlock(curBlockIdx);
                while (curBlockIdx <= FILE_DESCRIPTOR_END_INDEX) {
                        if (fdBlock[curReference] < 0) { // find free
                                // update length
                                fdBlock[curReference] = 0;
                                break;
                        }
```

```
                curReference += FILE_DESCRIPTOR_SIZE;
                if (curReference > MAX_INDEX_WITHIN_BLOCK) {
                        // check next file descriptor block
                        curBlockIdx++;
                        fdBlock = io.readBlock(curBlockIdx);
                        curReference = 0;
                }
        }

        // 2. find a free directory entry
        int curDirIdx = DIRECTORY_START_INDEX;
        int curSlotIdx = 0; // 1st index {name, index}
        int[] dirBlock = io.readBlock(curDirIdx);
        while (curDirIdx <= DIRECTORY_END_INDEX) {
                if (dirBlock[curSlotIdx + 1] < 0) { // find free
                        // update name and index
                        dirBlock[curSlotIdx] = convertStringToInt(name);
                        dirBlock[curSlotIdx + 1] = (curBlockIdx -
FILE_DESCRIPTOR_START_INDEX)
                                        * 4 + curReference / 4;
                        break;
                }
                curSlotIdx += SLOT_SIZE;
                if (curSlotIdx > MAX_INDEX_WITHIN_BLOCK) {
                        // check next directory block
                        curDirIdx++;
                        dirBlock = io.readBlock(curSlotIdx);
                        curSlotIdx = 0;
                }
        }

        // 3. write back the updates to disk
        io.writeBlock(curBlockIdx, fdBlock);
        io.writeBlock(curDirIdx, dirBlock);
    }

e. FS.destroy(String name) {
        // 1. search directory to find file descriptor
        int curDirIdx = DIRECTORY_START_INDEX;
        int curSlotIdx = 0; // 1st index {name, index}
        int[] dirBlock = io.readBlock(curDirIdx);
        int nameToInt = convertStringToInt(name);
        int fdIdx = -1;
        int[] fdBlock = null;
        while (curDirIdx <= DIRECTORY_END_INDEX) {
                if (dirBlock[curSlotIdx] == nameToInt) { // find!
                        // 2. free file descriptor
                        fdIdx = curSlotIdx / 4 + 5;
```

```
                        int fdReference = curSlotIdx % 4;
                        fdBlock = io.readBlock(fdIdx);
                        fdBlock[fdReference] = -1;
                        // 3. update bitmap
                        for (int i = 1; i < 4; i++) {
                                setBitMap(fdBlock[fdReference + i]);
                        }
                        // 4. remove directory entry
                        dirBlock[curSlotIdx + 1] = -1;
                        break;
                }
                curSlotIdx += SLOT_SIZE;
                if (curSlotIdx > MAX_INDEX_WITHIN_BLOCK) {
                        // check next directory block
                        curDirIdx++;
                        dirBlock = io.readBlock(curSlotIdx);
                        curSlotIdx = 0;
                }
        }

        // 5. write back the updates to disk
        io.writeBlock(fdIdx, fdBlock);
        io.writeBlock(curDirIdx, dirBlock);
}
```

f.
```
FS.open(String name) {
        // 1. search directory to find file descriptor
        int slotIdx = getSlotIdx(name);
        if (slotIdx < 0) {
                System.out.println(name + " doesn't exist!");
                return -1;
        }

        // 2. allocate a free OPT entry
        int freeOPTIdx = getFreeOPTEntryIdx();

        // 3. fill in current position and file descriptor index
        OFT[freeOPTIdx].currentPosition = 0;
        OFT[freeOPTIdx].index = slotIdx;
        OFT[freeOPTIdx].whichBlock = 0;

        // 4 search first data block - update bitmap, update file descriptor
        int[] fdBlock = getFDBlockFromSlotIdx(slotIdx);
        if (fdBlock[slotIdx * 4 + 1] == -1) {
                int newBlockIdx = searchAndUpdateBitMap();
                fdBlock[slotIdx * 4 + 1] = newBlockIdx;
                int fdIdx = slotIdx / 4 + 5;
                io.writeBlock(fdIdx, fdBlock);
```

```
                }

                // 5. read block 0 of file into the r/w buffer(read-ahead)
                int fileLength = fdBlock[slotIdx * 4];
                int firstDataBlockIdx = fdBlock[slotIdx * 4 + 1];
                OFT[freeOPTIdx].length = fileLength;
                if (firstDataBlockIdx != -1)
                        OFT[freeOPTIdx].buffer = io.readBlock(firstDataBlockIdx);

                return freeOPTIdx;

g. FS.close(int OPTIdx) {
                // 1. write buffer to disk
                int dataBlockIdx = getDataBlockIdxFromOPTEntry(OFT[OPTIdx],
OFT[OPTIdx].whichBlock);
                io.writeBlock(dataBlockIdx, OFT[OPTIdx].buffer);

                // 2. update file length in descriptor
                int slotIdx = OFT[OPTIdx].index;
                int[] fdBlock = getFDBlockFromSlotIdx(slotIdx);
                fdBlock[slotIdx % 4] = OFT[OPTIdx].length;
                int fdIdx = slotIdx / 4 + 5;
                io.writeBlock(fdIdx, fdBlock);

                // 3. free OPT entry
                OFT[OPTIdx].index = -1;

                // 4. return status
                return true;
        }

h. FS.read(int OPTIdx, int count) {
                StringBuilder sb = new StringBuilder();
                while(count > 0 && OFT[OPTIdx].currentPosition < OFT[OPTIdx].length){
                        if(OFT[OPTIdx].currentPosition < 64 * (OFT[OPTIdx].whichBlock + 1)){
                                // read buffer
                                char c = OFT[OPTIdx].readCharFromBuffer(OFT[OPTIdx].currentPosition
% 64);
                                sb.append(c);
                                OFT[OPTIdx].currentPosition++;
                                count--;
                        } else{
                                // switch to next block
                                OFT[OPTIdx].whichBlock++;
                                int dataBlockIdx = getDataBlockIdxFromOPTEntry(OFT[OPTIdx],
OFT[OPTIdx].whichBlock);
                                if(dataBlockIdx == -1)
                                        break;
```

```
                        OFT[OPTIdx].buffer = io.readBlock(dataBlockIdx);
                    }
                }

                return sb.toString();
            }

I. FS.write(int OPTIdx, char c, int count) {
                int oldCount = count;
                while(count > 0){
                    if(OFT[OPTIdx].currentPosition < 64 * (OFT[OPTIdx].whichBlock + 1)){
                        // 1. write text to buffer
                        OFT[OPTIdx].writeCharToBuffer(c, OFT[OPTIdx].currentPosition % 64);
                        OFT[OPTIdx].currentPosition++;
                        OFT[OPTIdx].length++;
                        count--;
                    } else{
                        // 2. write the buffer to disk block
                        int dataBlockIdx = getDataBlockIdxFromOPTEntry(OFT[OPTIdx],
OFT[OPTIdx].whichBlock);
                        io.writeBlock(dataBlockIdx, OFT[OPTIdx].buffer);

                        // 3. update file length in descriptor
                        int slotIdx = OFT[OPTIdx].index;
                        int[] fdBlock = getFDBlockFromSlotIdx(slotIdx);
                        fdBlock[0] = OFT[OPTIdx].length;
                        int fdIdx = slotIdx / 4 + 5;

                        // 4. switch to next block
                        OFT[OPTIdx].whichBlock++;
                        //    search first data block - update bitmap, update file descriptor
                        if (fdBlock[slotIdx * 4 + OFT[OPTIdx].whichBlock + 1] == -1) {
                            int newBlockIdx = searchAndUpdateBitMap();
                            fdBlock[slotIdx * 4 + OFT[OPTIdx].whichBlock + 1] =
newBlockIdx;
                        }
                        io.writeBlock(fdIdx, fdBlock);

                        // 5. read block whichBlock of file into the r/w buffer(read-ahead)
                        int fileLength = fdBlock[slotIdx * 4];
                        int firstDataBlockIdx = fdBlock[slotIdx * 4 + OFT[OPTIdx].whichBlock + 1];
                        OFT[OPTIdx].length = fileLength;
                        if (firstDataBlockIdx != -1)
                            OFT[OPTIdx].buffer = io.readBlock(firstDataBlockIdx);
                    }
                }

                return oldCount;
```

```
	}

j. FS.seek(int OPTIdx, int target) {
		int curDataBlockNum = OFT[OPTIdx].currentPosition / 64;
		int targetDataBlockNum = target / 64;
		if (curDataBlockNum != targetDataBlockNum) { // if the new position is not within the
current block
			int slotIdx = OFT[OPTIdx].index;
			// 1. write the old buffer to disk
			int[] fdBlock = getFDBlockFromSlotIdx(slotIdx);
			int oldDataBlockIdx = fdBlock[slotIdx % 4 + curDataBlockNum + 1];
			io.writeBlock(oldDataBlockIdx, OFT[OPTIdx].buffer);

			// 2. read the new block to OPT
			int newDataBlockIdx = fdBlock[slotIdx % 4 + targetDataBlockNum + 1];
			OFT[OPTIdx].buffer = io.readBlock(newDataBlockIdx);
			OFT[OPTIdx].whichBlock = targetDataBlockNum;
		}
		OFT[OPTIdx].currentPosition = target;
	}

k. FS.save(String outputPath) throws Exception {
		for (int i = 1; i < 4; i++) {
			if (OFT[i].index != -1) {
				close(i);
			}
		}

		// convert array of bytes into file
		FileOutputStream fileOuputStream = new FileOutputStream(outputPath);
		byte[] temp = io.saveDiskToBytes();
		fileOuputStream.write(temp);
		fileOuputStream.close();

		// init OPT
		initOPT();
	}

l. FS.restore(String inputPath) throws Exception {
		byte[] bytes = new byte[64 * 64];
		FileInputStream fileInputStream = new FileInputStream(new File(
				inputPath));
		fileInputStream.read(bytes);
		fileInputStream.close();
		io.restoreDiskFromBytes(bytes);
		System.out.println();
	}
```