COSI 131a: Fall 2017

PA3: File System Design

Simulated Disk

The simulated disk uses a UNIX file named DISK to simulate a disk with NUM_BLOCKS blocks of BLOCK_SIZE bytes per block. It supports three methods:

```
/**
 * Read a block from the disk into a buffer.
 */
void read(int blockNum, byte[] buffer);

/**
 * Write a block from a buffer onto the disk.
 */
void write(int blockNum, byte[] buffer);

/**
 * Stop the disk and report how many read and writes took place.
 * If removeFile is true, it will also delete the DISK file.
 */
int stop(boolean removeFile);
```

In each case blockNum is required to be in range 0..Disk.NUM_BLOCKS-1 and buffer should be a byte array of **exactly** Disk.BLOCK_SIZE bytes. (There are also overloaded versions of read and write as described below.) Note that blocks must be read and written as a whole. If you need to read part of a block, you must read in the entire block and ignore the part you're not interested in. If you need to write part of a block, you must read in the whole block, modify the portion of interest, and write the whole block back out. The constructor looks for a file named DISK in the current directory. If it does not exist, the program will create it and initialize the first block to all zeros. Any other block must be written at least once before it can be read. The stop method prints statistics. It has an optional argument (default true) to indicate whether to remove the DISK file. Since the DISK file can be quite large, you should be sure to remove it before logging off.

The scheme we employ for implementing the file abstraction inside the simulated disk is very similar to the method used in the original UNIX. The file system is composed of a block allocator

(which tracks which blocks are free, i.e., *unused*) and an inode manager which tracks the blocks that belong to each file, identifying files by their inode.

Super Block

Block 0 of the disk is the so-called "super block", which contains information about the rest of the disk. You will want to keep a copy of this block in memory at all times. It should be read in when the file system starts up, and written back out before shutting down. The super block holds the following variables:

```
class SuperBlock {
    public int size;   // total size of file system (in blocks)
    public int msize;  // number of blocks used by the free space map
    public int isize;  // number of inode blocks
    public byte freeMap[] // first bits of free map
}
```

The size of the file system is recorded in the super block to allow the file system to use less than the whole disk and to support various disk sizes. In all the data structures on disk, a "pointer" to a disk block is an integer in the range 1..NUM_BLOCKS-1. Since block 0 is treated specially, you can use a block number of zero to represent a null pointer.
Remember that the super block (block 0), the free map blocks (1..msize), and the inode blocks (blocks msize+1..msize+isize) are at the beginning of your disk. The rest of the blocks are all data blocks. The contents of the data blocks will be determined by the write commands you issue in your program. When you go to read a block, you will have to know beforehand what type of block it is in order to read it correctly. Remember that "size" refers to the total number of blocks in the file system, so the number of data blocks unused by metadata in the file system is size - msize - isize - 1 ("-1" for the super block).

Free Space

You will manage free space by keeping track of which blocks are free (not a part of any file) in a *bitmap*. A bitmap is simply an array of bits; in our case, each bit corresponds to a block on the disk. We will call this the *free map*. If the bit corresponding to block n is 1, then block n is currently free. If the bit corresponding to page n is 0, then page n is not free.

The free space bitmap is stored in blocks 1..msize. For convenience, a free map block can be read into the following structure:

```
public class FreeMapBlock {
    public byte[] map;
}
```

File Structure and Indirect Blocks

Each file in the system is described by an *index node* (*inode* for short)[1].

```
class Inode {
    public final static int SIZE = 64; // size in bytes
    public int flags;
    public int owner;
    public int size;
    public int ptr[] = new int[13];
}
```

If the flags field is zero, the index block is unused. In a real file system, the bits of this int distinguish different types of file (directories, data files, etc.) and indicate permissions. You do not have to implement these features. Similarly, you may ignore the owner field. The size field indicates the current size of the file, in bytes.

Block 0 of the disk is the super block. Blocks msize+1..msize+isize are packed with inodes[1].

```
class InodeBlock {
      public static final int COUNT = Disk.BLOCK_SIZE / Inode.SIZE;
      public Inode inodes[] = new Inode[COUNT];
}
```

The remaining blocks may be unallocated, or allocated as *direct* or *indirect* blocks. They are collectively known as *data* blocks. Unallocated data blocks are marked as free in the free map, while allocated data blocks are marked as used.
The data blocks that contain the contents of the files are called *direct* blocks. The ptr fields in an inode point (directly or indirectly) to these blocks. The first 10 pointers point to the first 10 direct blocks. The 11th pointer (ptr[10]) points to an *indirect* block. This block contains pointers to the next Disk.BLOCK_SIZE/4 direct blocks of the file[1]. Pointer ptr[11] points to a "doubly indirect" block. It is filled with pointers to indirect blocks, each of which contains pointers to data blocks. Similarly, the final pointer points to a "triply indirect" block. The size of the file is determined by the size field of the inode, not by the pointers.

```
class IndirectBlock {
    public static final int COUNT = Disk.BLOCK_SIZE / 4;
    public int ptr[] = new int[COUNT];
}
```

You can calculate the maximum size of a file given the number of pointers that fit in an indirect block (IndirectBlock.COUNT). It is a good exercise to go ahead and try the calculation now.

Inode numbering

Inodes are numbered consecutively starting at 1 (not zero!). Since inodes are packed into blocks, each Inode block contains Disk.BLOCK_SIZE/Inode.SIZE or InodeBlock.COUNT Inodes (remember that Inode 1 will be located in the first Inode block, which is found after the last Free Map block).

Holes

A null pointer (either in the inode or in one of the indirect blocks) may indicate a *hole* in the file. For example, if the size field indicates that there should be five blocks, but ptr[2]==0, then the third block constitutes a hole. Similarly, if the file is large enough and ptr[10]==0, then blocks 11 through IndirectBlock.COUNT+10 are all holes. Attempts to read from a hole act as if the hole were filled with zeros; an attempt to write into a hole causes the hole to be "filled in": blocks are allocated as necessary and added to the file. Holes are created by seeking beyond the end of the file and then writing.

Other Disk Operations

The data structures SuperBlock, InodeBlock, and IndirectBlock are all the same size (i.e., all blocks are the same size), so any one of them can be written to or read from any disk block. For your convenience, we have added four overloaded versions of read and write to the Disk interface. These versions allow read and write to super blocks, Inode blocks, Indirect blocks, and Free Map blocks.

```
class Disk {
    public Disk() {
    public void read(int blocknum, byte[] buffer) {}
    public void read(int blocknum, SuperBlock block) {}
    public void read(int blocknum, InodeBlock block) {}
    public void read(int blocknum, IndirectBlock block) {}
    public void read(int blocknum, FreeMapBlock block) {}
    public void write(int blocknum, byte[] buffer) {}
    public void write(int blocknum, SuperBlock block) {}
    public void write(int blocknum, InodeBlock block) {}
    public void write(int blocknum, IndirectBlock block) {}
    public void write(int blocknum, FreeMapBlock block) {}
    public void stop() {}
}
```

File Table

This code is already provided in FileTable.java. However, make sure you understand how to use it from FileSystem. It is a data structure to keep track of open files. For each file, you will

need a pointer to an in-memory copy of its inode, its inumber, and the current seek pointer. Understand the methods for allocating and freeing slots in this table, determining whether a file descriptor is valid, and accessing the data associated with a file descriptor. FileTable.java has two classes: FileDescriptor and FileTable.
The FileDescriptor is an abstract data type (ADT). It contains private variables (inode, inumber, current seek pointer) and a series of get() and set() methods to either retrieve or update the object's private variables.

```
class FileDescriptor {
    public FileDescriptor(Inode newInode, int newInumber){}
    public Inode getInode(){}
    public int getInumber(){}
    public int getSeekPointer(){}
    public void setSeekPointer(int i){}
    public void setFileSize(int newSize){}
}
```

The file table uses a byte array called fileMap tells whether the specified index has a file open. fileMap is an array of 0's or 1's. 1 denoting an open file and 0 denoting an open slot. So if no files are open, then fileMap is an array of all zeros. The FileDescriptor array (fds) contains FileDescriptor objects. The index in fileMap corresponds to the fd object in fds array with the same index (denoted by parameter fd).
Each FileDescriptor object is an instance of the class FileDescriptor that maintains information about the specified file.

```
class FileTable {
    public FileTable(){}

    /**
     * Returns the index in the file table. Returns -1 if the
     * table is full.
     */
    public int allocate(){}

    /**
     * Add an entry to the table. This method will overwrite.
     *
     * Returns 0 for success and -1 for error
     */
    public int add(Inode inode, int inumber, int fd){}

    /**
     * Free an entry in the table.
```

```
 */
public void free(int fd){}

/**
 * Returns true if the file descriptor is valid, returns
 * false otherwise.
 */
public boolean isValid(int fd){}

/**
 * Given a file descriptor, this method returns the
 * corresponding inode.
 *
 * Returns null if no inode.
 */
public Inode getInode(int fd){}

/**
 * Given a file descriptor, this method returns the
 * inumber given a file descriptor or 0 if error.
 */
public int getInumber(int fd){}

/**
 * Given a file descriptor, this method returns the
 * corresponding seek pointer.
 */
public int getSeekPointer(int fd){}

/**
 * Updates the seek pointer given a file descriptor.
 *
 * Returns 1 if update is a success or 0 if failure.
 */
public int setSeekPointer(int fd, int newPointer) {}

/**
 * Updates the file size given a file descriptor.
 *
 * Returns 1 if update is a success and zero if failure.
 */
public int setFileSize(int fd, int newFileSize){}
```

```
    /**
     * Given an inumber it returns a file descriptor
     * or -1 if failure.
     */
    public int getFDfromInumber(int inumber){ }
}
```

File System Operations

The class **MyFileSystem** implements an interface **FileSystem**. The test programs and included shell all access the file system only through those methods in the FileSystem interface.

```
interface FileSystem {
    public int formatDisk(int size, int isize);
    public int shutdown();
    public int create();
    public int inumber(int fd);
    public int open(int inumber);
    public int read(int fd, byte[] buffer);
    public int write(int fd, byte[] buffer);
    public int seek(int fd, int offset, Whence whence);
    public int close(int fd);
    public int delete(int inumber);
}
```

In the tradition of C programming, each method returns an integer value, with -1 meaning "error" and a non-negative value (0 unless specified otherwise) meaning “success”[2].

- The method formatDisk initializes the disk to the state representing an empty file-system: It fills in the super block and links all the data blocks into the free map.
- The method shutdown closes all open files and shuts down the simulated disk.
- The method create creates a new empty file, and open locates an existing file. Each method returns an integer in the range from 0 through 20 (since, usually, a process has a limited number of files it have opened at a time, i.e., the size of the table containing the files descriptors is limited, and 20 is a reasonable number) called a "file descriptor" (fd for short). The fd is an index into an array called a "file descriptor table" representing open files. Each entry is associated with one file and also contains a "file pointer" (initially zero)[3].
    - The argument to open is the inumber of an existing file[4]. The method inumber returns the inumber of the file corresponding to an open file descriptor.
- The methods read, write, seek, and close behave similarly to their UNIX counterparts.
    - The method read reads up to buffer.length bytes starting at the current seek pointer. The return value is the number of bytes read. If there are fewer than buffer.length bytes between the current seek pointer and the end of the file (as

indicated by the *size* field in the inode), only the remaining bytes are read. In particular, if the current seek pointer is greater than or equal to the file size, then read returns zero and the buffer is unmodified. (The current seek pointer cannot be less than zero). The seek pointer is incremented by the number of bytes read.

- The method write transfers buffer.length bytes from buffer to the file starting at the current seek pointer and advances the seek pointer by that amount. It is *not* an error if the seek pointer is greater than the size of the file. In this case, holes may be created.
- The method seek modifies the seek pointer using offset and whence:
  - If whence == Whence.SEEK_SET, the seek pointer is set to offset relative to the beginning of the file.
  - If whence == Whence.SEEK_CUR, the seek pointer is adjusted by offset relative to its current position.
  - If whence == Whence.SEEK_END, the seek pointer is set to offset relative end of the file.

  The value of the offset can be positive or negative; however the resulting seekPointer must always be positive or zero. If a call to seek would result in a negative value for the seek pointer, the seek pointer is unchanged and the call returns -1. Otherwise, value returned is the new seek pointer (distance in bytes from the start of the file).

  Seek Example:
  file1 is 10 bytes in size. The file is of the string "<3COSI131a".

0	1	2	3	4	5	6	7	8	9
<	3	C	O	S	I	1	3	1	a

Let the current seek pointer be at the 3rd letter (whose value is "O") and buffer.length = 20 before each of the following examples.

- After seeking with offset = 2 and whence = Whence.SEEK_CUR, calling read(file1, buffer) will store "I131a" into buffer.
- After seeking with offset = -2 and whence = Whence.SEEK_CUR, calling read(file1, buffer) will store "3COSI131a" into buffer.
- After seeking with offset = -2 and whence = Whence.SEEK_END, calling read(file1, buffer) will store "1a" into buffer.
- After seeking with offset = -12 and whence = Whence.SEEK_END, the seek will return -1 and calling read(file1, buffer) will store "OSI131a" into buffer.
- After seeking with offset = 2 and whence = Whence.SEEK_SET, calling read(file1, buffer) will store "COSI131a" into buffer.

The method close writes the inode back to disk and frees the file table entry.
The method delete frees the inode and all of the blocks of the file. It is an error to delete a file that is currently open[5].
The method shutdown closes all open files, flushes all in-memory copies of disk structures out to disk, calls the stop() function on the disk, and prints any debugging or statistical information you deem worthwhile.

Footnotes

[1]There is also an artifact of Java here that would not be present in a real operating system. In Java, the Inode structure is stored in memory as three integers followed by a *pointer* to an array of thirteen more integers. There would also be additional information to indicate the type of the Inode structure and the size of the array. On disk, however, the Inode structure is simply 16 integers in a row, like this C structure:

```
struct inode {
    int flags;
    int owner;
    int size;
    int ptr[13];
};
```

Unfortunately, there's no easy way to create exactly this structure in memory in Java, but fortunately, you will probably never notice the difference. Similar remarks apply to InodeBlock and IndirectBlock.

[2]A real system would need some way to indicate what sort of error occurred. In UNIX, the nature of the error is indicated by an integer error code placed in a global variable called errno. For this project, you can just print an error message. A more "Java-like" design would use exceptions to indicate errors.

[3]In UNIX, this array is split into three parts. Each process has its own table of open files. There is a single system-wide table of so-called "in-core inodes" shared among all processes. Each entry in this table has a reference count so that it can be removed when the last process closes the file. Seek pointers are kept in yet another system-wide table so that there can be multiple seek pointers into the same file, and multiple processes can share a seek pointer. For this project, you can combine all this information into one table.

[4]In UNIX, the argument is a pathname. The file system uses the directories to translate this name into an inumber.