---- GROUP ----

Daniel Finlay dsf77
Khalid Jaber ksa68
Daniel Hollasch djh201

---- PRELIMINARIES ----

BASE:
Implemented all basic methods, applied indirect pointers to mathematically support files of size 65,536 bytes for the extension base.

EXTRA CREDIT:
For Extra credit implemented:
Directory Support: opendir(), closedir(),
Extended Directory Operations: mkdir(), rmdir()

---- DATA STRUCTURES ----

1)This is our Bitmap structure that we used for mapping whether a specific inode/datablock is being used/occupied at any given time on the entire file system, so we can find free space for allocation of files. We implemented set(),clear(),get(),allocat(),deallocate(), & print() methods to manipulate the inode & databitmaps

```
typedef struct {
  int bits, numPartitions;
  bitmap_type *container;
} BitMap;
```

2)This is our ByteBuffer structure that we used for serialization/marshalling of our data structures to allow for persistency of our variables/structs throughout the use of the filesystem calls. We implemented allocate(), allocate_n(), getByte(), mergeBuffers(), writeByte(), writeShort(), writeInt(), writeLong(), writeString(),

readByte, readShort(), readInt(), readLong(), readString(), for the manipulation/functionality of our ByteBuffer.

```
typedef struct {
  int readerPosition, writerPosition;
  Byte *buffer;
} ByteBuffer;
```

3)This is our Inode structure, of size 488 Bytes, that we used for keeping track of the possible details of the file an Inode represents. In order to support the minimum number of files we allocated blocks 1-128 of our file system, with one Inode per block, to store our Inodes in the file system. Our alternative inode structure allows for large files by instead of have just indirect or doubly indirect pointers, it has blockLinks[200] list that can potentially create files as large as the File System data blocks section allows, and discontinuously. So inodeX can have a file structure that points to data blocks 1->255->3->257->..., while inodeY can have a file structure that points to data blocks 2->256->4->599->..., This allows for very large file sizes. The maximum size of a file supported by our system is: Indirect: 1 direct block * 128 Inode blocks * 512 bytes/block = 65,536 bytes

```
typedef struct {
  /**
   *  The user id of the file's owner.
   */
  uid_t userId; // TODO find out what owner represents (more clarification needed)
  /**
   * The group id of the file's owner.
   */
  gid_t groupId;
  /**
   * Type (directory, file, device)
   */
  mode_t st_mode;
  /**
```

```
    * The identifer of the i-node.
    */
    ino_t id;
    /**
    * Last FILE modified time.
    */
    timestruc_t lastFileModTime;
    /**
    * Last accessed time.
    */
    timestruc_t lastAccessTime
    /**
    * Last iNode modified time.
    */
    timestruc_t lastModifiedTime
    /**
    * Number of links to the file.
    */
    nlink_t numFileLinks;
    /**
    * The size of the file in bytes.
    */
    size_t fileSize;
    /**
    * The list of blocks to support.
    */
    short blockLinks[200];
} INode;
```

4)This is our SuperBlock structure that we used for tracking the necessary metadata of the file system. This SuperBlock is stored in block 0 of our filesystem. It hold bitmaps of the free inodes/blocks in the file system, along with the number of free inodes/blocks that exist on the filesystem.

```
typedef struct {
  /**
  * List of free blocks.
  */
  BitMap *blockBitMap;
  /**
   * List of free I-nodes.
   */
  BitMap *iNodeBitMap;
  /**
   * Number of free blocks
   */
  long numFreeBlocks;
  /**
   * Number of free I-nodes
   */
  short numFreeINodes;
} SuperBlock;
```

5)This is our Directory Entry structure that we used for storing the string representation of a file to its corresponding inode number mapping.

```
typedef struct {
  /**
   * The number of the i-node.
   */
```

```
  ino_t ino;
  /**
   * The name of the entry.
   */
  char entryName[NAME_MAX];
} DirectoryEntry;
```

6)This is our Directory structure that we used for managing our nested directories in a First-Child, Sibling Linked List tree structure, with each Directory node holding the Directory Entry struct from above of the filename <--> Inode number mapping.

```
/**
* Contains a list of (entry name, inode number) pair
*/
typedef struct Directory {
  /**
   * The entry of this directory.
   */
  DirectoryEntry *entry;
  /**
   * The parent of this directory.
   */
  struct Directory *parent;
  /**
   * The next sibling of this directory.
   */
  struct Directory *sibling;
  /**
   * The next child of this directory.
   */
  struct Directory *child;
} Directory;
```

7)This is our Reserve Block structure that we used for reserving a given block of data so when a file is created/opened no other files mess with these reserved data block regions

```
typedef struct {
  int nextDataBlock;
  int nextLink;
} ReserveBlock;
```

---- ALGORITHMS ----

1)Traversing relative/absolute paths

Our implementation utilizes helper functions to traverse through the Tree data structure representing the file paths. ParseDirectory() traverses through the given path through LCRS tree recursively.

We implemented a findDirectory() method to help with the sfs_create and sfs_open methods. Utilizing this method assists us to find the parent directory of where to open/create/unlink the desired file. Once again we recursively traverse through the directory LCRS tree to find the specified absolute path passed in by parameters. FindDirectory() is used to find the parent directory for a given absolute path.

2) Inodes

We have several helper functions dedicated for Inode uses in our project. The first to mention is the node_stat() that is used to access information about the Inode of file. This includes the user/group IDs, last access/modified times, and the mode of the Inode.

To find an Inode we have the method findInode() given the absolute path of the directory. We once again call the findDirectory() method passing the absolute path to find the inode Id for the given path. We add what is returned to the "InodeList" location to get the correct Inode id we are looking for.

To check if an Inode in the list of Inodes is free or in use, we have three functions to do so. The function node_reserve() reserves an inode using the Bitmap data structure in conjunction with the function bitmap_set(). To unreserve an Inode we have the function node_unreserve(), again using the Bitmap data structure, but this time with bitmap_clear. In both these methods we utilize the node_stat() to get the ID of any inode passed as a parameter. Lastly we have the isReservedNode() method to check if a given Inode parameter is currently reserved, i.e if it is currently in use holding a file's metadata.

----METHODS----

// Implemented/Finished
void *sfs_init(struct fuse_conn_info *conn)
SPECIAL NOTES For optional Fuse function:

This function is responsible for initializing the filesystem.The first step of this method is to open the disk to be written to. By opening the disk, we prepare the system to be written to for initialization. We implement a mutex lock to protect this section of code to ensure init is thread safe. Following the lock, we allocate the SuperBlock and bitmap_allocate() the SuperBlock for holding the metadata of the filesystem. Utilizing the bitmap_allocate() again, we allocate the Inodes bitmap and free block. So the superblock now holds metadata regarding the state of the filesystem; the location of blocks, the number of free blocks, and free inodes.

Once the preliminary initialization is complete, we then loop through all the Inode blocks to initialize their metadata , the user/group ID, access/modified times, and mode types.

// Implemented/Finished
int sfs_getattr(const char *path, struct stat *st)

//Implemented/Finished
int sfs_create(const char *path, mode_t mode, struct fuse_file_info *fi)

// Implemented/Finished
int sfs_open(const char *path, struct fuse_file_info *fi)

//Implemented/Finished
int sfs_unlink(const char *path)

```
// Implemented/Finished
int sfs_mkdir(const char *path, mode_t mode)

// Implemented/Finished
int sfs_rmdir(const char *path)

// Implemented/Finished
int sfs_opendir(const char *path, struct fuse_file_info *fi)

//Implemented/Finished
int sfs_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info
*fi)
```