File System Design - Team 42

Nabeel Rana
Leigh Ann Apotheker
Ethan Zheng
Bryan Mendez

What your Volume Control Block looks like (what fields and information it might contain)

The *signature* is a unique identifier to verify the file system type. The *volumeSize* field is the size of the volume in bytes. The *blockSize* field is the size of each block in bytes. The *numOfBlocks* field is the total number of blocks in the volume. The *fsBlockNum* is the block number where the filesystem metadata is. The *rootDirBlockNum* is the block number where the root directory is. The *fatStartBlock* is the block number where the free space management starts and *fatBlockNum* is the number of blocks that it occupies.

```
typedef struct VolumeControlBlock
{
   int signature; // unique identifier to verify the file system type
   int volumeSize; // size of the volume in bytes
   int blockSize; // size of each block in bytes
   int numOfBlocks; // total number of blocks in the volume
   int fsBlockNum; // block number where the filesystem metadata is
   int rootDirBlockNum; // block number where the root directory is
   int fatStartBlock; // block number where the free space management (FAT) starts
   int fatBlockNum; //number of blocks FAT occupies
} VCB;
```

How you will track the free space (include function prototypes for how the rest of the filesystem will use the freespace system).

We'll use a File Allocation Table (FAT) to track the free space. It'll consist of an array of integers, where each element represents a block. We'll have a special number that represents a free block (-1) and the end of a file (-2). Otherwise, each used block will be the number to the next block of the file. When a file is deleted, we'll iterate through blocks starting from the start block and setting them to free until the end of the file block.

```
int initFreeSpace(int numOfBlocks);
int allocateBlocks(int numOfBlocks, int startBlock);
int freeBlocks(int startBlock);
```

What your directory entry will look like (include a structure definition).

The *DirectoryEntry* structure represents an entry in the filesystem. The *name* field is the name of the file or directory, which supports up to 100 characters. The *size* field is the size of the file in bytes. The *location* field is the block number where the file starts. The *isDirectory* field is a flag that indicates whether the entry is a file (0) or directory (1). The *modifiedAt* field is the Epoch time in seconds of the last time the file was modified. The *createdAt* field is the Epoch time in seconds of the time the file was created. The *accesesedAt* field is the Epoch time in seconds of the last time the file was accessed.

```
typedef struct DirectoryEntry
{
   char name[100]; // name of the file or directory
   int size; // size of the file in bytes
   int location; // block number where the file starts
   int isDirectory; // flag that indicates whether the entry is a file or directory
   int modifiedAt; // Epoch time in seconds of the last time the file was modified
   int createdAt; // Epoch time in seconds of the time the file was created
   int accessedAt; // Epoch time in seconds of the last time the file was accessed
} DE;
```

Are your files going to be stored contiguous or dis-contiguous?

Our files will be stored dis-contiguously because our choice for free space management, FAT, supports it. Additionally, through dis-contiguous allocation, we would avoid running into external fragmentation issues as files are allocated and deleted. There is no external fragmentation with linked allocation with FAT. As such, there is no need for compaction, which is used to approach external fragmentation in contiguous allocation. The cost for compaction can be considerable. While it may be simpler to initially implement contiguous memory allocation, dis-contiguous memory allocation allows for more ease in modifying the size farther in our implementation of our file system. Furthermore, with contiguous allocation, there is difficulty in finding the space for a new file and knowing the file size. To create a new file with linked allocation, we can create a new entry in the directory, and a file size can continue to expand as long as free blocks are available.