In this project, you are free to use any disk layout you wish. However, we suggest that you use a file-allocation-table (FAT) layout, as described below. The layout is conceptually the easiest to understand, as well as the easiest to debug. If you wish to take a more challenging approach and implement an inode-based solution, we strongly recommend that you first consult the course staff in order to understand the scope of such a design.

1 Disk model

Recall that you will be given a disk (actually a file on the real file system) that you will interact with only via `dread` and `dwrite`. When reading, you must read one and exactly one block; when writing, you must write one and exactly one block. Thus, your disk is really just an array of blocks

```
+---+---+---+---+---+---+-------+---+
| 0 | 1 | 2 | 3 | 4 | 5 |  ...  | N |
+---+---+---+---+---+---+-------+---+
```

where each of the blocks are all exactly `BLOCKSIZE` (512 bytes).

2 Disk layout

We now describe the suggested disk layout for this project. At a high level, your will divide the disk into four different types of blocks: a single *volume control block* (VCB), a number of *directory entry* (DE) blocks, a number of *file allocation table* (FAT) blocks, and a number of *data blocks* (DB). The VCB, DE blocks, and FAT blocks all represent *overhead*, meaning the blocks are only used to store metadata, not actual user data. The user data for files is only stored in DBs. Thus, in designing you file system, you'd like to keep as many blocks as DBs as possible.

Note that you should assign blocks to a category when you format the disk in 3600`mkfs` and not change its category once assigned. Thus, you should select, up front, how many DE blocks will exist, how many DBs, etc.

2.1 Volume control block

The volume control block (VCB) is the first block in the file system, and contains the information needed to find the rest of the file system structures, as well as metadata for the root directory. The file system layout information should contain the block size of the disk, pointers to where the DE blocks begin and end, where the FAT blocks begin and end, and where the DB blocks begin and end. Finally, you should also include a "magic number", that's always the same for disks you format (simply pick a constant). This will allow you to quickly identify whether a disk is yours when you go to mount it.

You may wish to define a C structure to represent your volume control block. If so, this might resemble something like

```
typedef struct vcb_s {
  // a magic number of identify your disk
  int magic;

  // description of the disk layout
  int blocksize;
  int de_start;
  int de_length;
  int fat_start;
  int fat_length;
  int db_start;

  // metadata for the root directory
  uid_t user;
  gid_t group;
  mode_t mode;
  struct timespec access_time;
  struct timespec modify_time;
  struct timespec create_time;
} vcb;
```

Obviously, you will need to be able to write the initial VCB to block 0 of the disk when formatting the disk in `3600mkfs`, and then read it back when you are mounting the filesystem in `vcb_mount`. To read and write it, you will first need to create a temporary memory location that is BLOCKSIZE bytes, copy the VCB you wish to write into that location, and then finally call `dwrite`. An example of this is

```
// first, set up your VCB
vcb myvcb;
myvcb.blocksize = BLOCKSIZE;
myvcb.de_start = ...;
...

// then, copy it to a BLOCKSIZE-d location
char tmp[BLOCKSIZE];
memset(tmp, 0, BLOCKSIZE);
memcpy(tmp, &myvcb, sizeof(vcb));

// finally, actually write it to disk in the 0th block
dwrite(0, tmp);
```

You should use similar code to read and write directory entry blocks and file allocation table blocks. For example, if you at some point later wish to read in the VCB from disk, you can do something like

```
// first, allocate up your VCB
vcb myvcb;
```

```
// now, set up a temporary BLOCKSIZE-d location
char tmp[BLOCKSIZE];
memset(tmp, 0, BLOCKSIZE);

// read it in from disk
dread(0, tmp);

// and copy it into your VCB structure
memcpy(&myvcb, tmp, sizeof(vcb));
```

2.2 Directory entries

Each file in your system is represented by a directory entry. This entry should contain the name of the file and the index of the first block that contains the file's data (basically, an index into the FAT). Additionally, you are going to want to store other file metadata in the directory entry, include the size of the file in bytes (`size`), the user ID who owns the file (`uid`), the group ID who owns the file (`gid`), the permissions associated with the file (`mode`), and the created/last accessed/last modified time of the file (`create_time`, `access_time`, `modify_time`). Thus, you should create a directory entry structure that resembles

```
typedef struct dirent_s {
  unsigned int valid;
  unsigned int first_block;
  unsigned int size;
  uid_t user;
  gid_t group;
  mode_t mode;
  struct timespec access_time;
  struct timespec modify_time;
  struct timespec create_time;
  char name[...];
} dirent;
```

To make your life easy, you should make the sure that BLOCKSIZE is divisible by the size of this structure. That way, you can pack a number of `dirents` into a single block, and no `direct` will span multiple disk blocks. Note that you can manipulate the size of the structure by changing the size of the `name` variable.

With this layout, you'll be deciding up front how many files your file system is able to support (this is determined by the number of directory entries, which is predetermined). Moreover, since there is only one directory in your filesystem, all of these files are by definition in the root directory.

2.3 File allocation table (FAT)

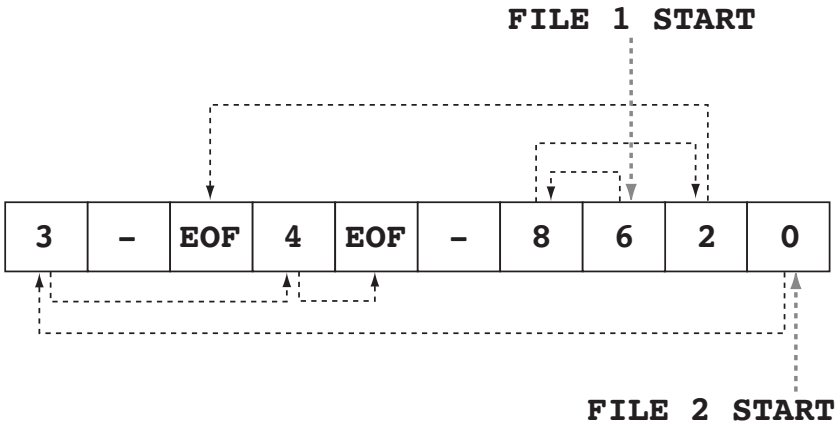
The final piece of metadata that you will need to create is your file allocation table (FAT). The FAT is a series of pointers that say which data blocks follow which in the layout of the files. Thus, your FAT should contain one entry for each data blocks that exist on your disk.

It is recommended that you have a structure for each entry in your FAT. This structure should look something like

```
typedef struct fatent_s {
  unsigned int used:1;
  unsigned int eof:1;
  unsigned int next:30;
} fatent;
```

This would use 32 bits per entry, with one bit reserved for whether or not the block is used or free, another bit to represent whether this block is the last block in the file (i.e., there is no next block), and the remaining 30 bits to encode the identifier (pointer) of the next block.

An example of a file allocation table with two files is shown below. File 1's start block is block 7. Following the pointers, the data in file 1 is stored on blocks {7,6,8,2}. Similarly, file 2's start block is block 9, and the entire contents of the file are on blocks {9,0,3,4}. Note that blocks 1 and 5 and currently unallocated and available to be assigned.



3 Implementation tips

Start by making the `dirent` structure the same size as BLOCKSIZE. This means that each DE block will contain exactly one `dirent`, making your code easier to debug. Later, once you have got it working reliably, reduce the size of the directory entries so that you can pack multiple entries into a single disk block to make your file system more efficient.

Overall, follow the maxim of first making your code correct and then making it fast. Premature optimization is the root of all evil.