Systems and Networks	**Project 2 Inode Implementation Notes**
CS3600 Fall 2014	**July 28, 2014**

In this project, you are free to use any disk layout you wish. If you feel up to a challenge, you can use an UNIX-like inode-based file system layout, as described below. The layout is harder understand and harder to debug, but makes the implementation of multiple directories (i.e., extra credit) easier.

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 seven different types of blocks: a single *volume control block* (VCB), *directory node* (DNODE) blocks, *directory entry* (DIRENT) blocks, *file node* (INODE) blocks, *indirect* (INDIRECT) blocks, *data blocks* (DB) and *free* (FREE) blocks. The VCB, DNODE, INODE, and INDIRECT 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.

When you initially format your disk, almost all blocks will be FREE blocks. As your disk receives `create` and `write` requests, the FREE blocks will slowly become other types of blocks (e.g., DNODE, INODE, INDIRECT, and DB blocks). When you receive a `delete`, you may free up some blocks, changing the state of certain blocks back to FREE.

2.1 Basic data types

Throughout the inode-based file system, you will be referencing block numbers (i.e., the blocks on which various pieces of data are stored). Many of these pointers will be statically allocated, and therefore many of them will be invalid when they are first created. Thus, I recommend using the following structure to represent block pointers:

```
typedef struct blocknum_t {
  int block:31;
  int valid:1;
} blocknum;
```

2.2 Volume control block (VCB)

The volume control block (VCB) is the first block in the file system, and contains metadata on the entire disk and the information needed to find the root directory DNODE. 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;

  // the block size
  int blocksize;

  // the location of the root DNODE
  blocknum root;

  // the location of the first free block
  blocknum free;

  // the name of your disk
  char name[496];
} vcb;
```

Note that the `name` variable was put in there simply to force the vcb structure to be 512 bytes (in general, making all of your block structures exactly `BLOCKSIZE` will make your life easier).

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));
```

2.3 Directory node block (DNODE)

Each directory in your system will have exactly one DNODE block. This block contains the metadata for the directory (i.e., the timestamps, owner, mode, etc), as well as pointers to the DIRENT blocks that actually contain the entries in the directory.

In more detail, you are going to want to store the number of entries in the directory (`size`), the user ID who owns the directory (`uid`), the group ID who owns the directory (`gid`), the permissions associated with the directory (`mode`), and the created/last accessed/last modified time of the directory (`create_time`, `access_time`, `modify_time`). You also need to store the pointers to the DIRENT blocks that actually have the entries; these are represented as an array of direct blocks, a single indirect block (i.e., a pointer to an INDIRECT block that has pointers to DIRENT blocks) and a double indirect block (a pointer to an INDIRECT block that has pointers to INDIRECT blocks that have pointers to DIRENT blocks).

Thus, you should create a directory entry structure that resembles

```
typedef struct dnode_t {
  // directory node metadata
  unsigned int size;
  uid_t user;
  gid_t group;
  mode_t mode;
  struct timespec access_time;
  struct timespec modify_time;
  struct timespec create_time;

  // the locations of the directory entry blocks
  blocknum direct[...];
  blocknum single_indirect;
  blocknum double_indirect;
} dnode;
```

Of course, you'll want to set the number of direct pointers so that your structure is BLOCKSIZE. Once you've got it set up, you can initialize it by doing things like

```
dnode mydnode;
mydnode.user = ...;
mydnode.access_time = ...;
```

```
...
mydnode.direct[1].valid = 0;
mydnode.single_indirect.valid = 0;
mydnode.double_indirect.valid = 0;
```

2.4 Indirect block (INDIRECT)

Indirect blocks are simply blocks that store more `blocknum` pointers to either DIRENT blocks, or other INDIRECT blocks. Thus, they can simply be represented as

```
typedef struct indirect_t {
  blocknum blocks[128];
} indirect;
```

2.5 Directory entry block (DIRENT)

Your directory entry blocks are blocks that actually contain the contents of directories. In general, they must contain the name of each entry, the type of each entry, and the blocknum where that entry's DNODE or INODE is stored. Addtionally, since the entries are statically allocated, you'll need to include a valid/invalid bit with each entry. So, you should first create a structure for a single directory entry

```
typedef struct direntry_t {
  char name[...];
  char type;
  blocknum block;
} direntry;
```

You can use the valid/invalid bit in the blocknum to indicate whether this entry is valid. In other words, you can use `mydirentry.block.valid` as your valid/invalid bit. Additionally, you should choose your name size to be something that makes direntry a power of 2 in size (e.g., 32 bytes). That way, you can pack them nicely into a DIRENT block.

The structure for your DIRENT block would then consist of an array of direntrys. In may look something like

```
typedef struct dirent_t {
  direntry entries[...];
} dirent;
```

Of course, it should be made to be BLOCKSIZE in size.

2.6 File inode block (INODE)

Each file in your system is represented by an INODE block. This entry should contain the file metadata including 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`). You also need to store the pointers to the DB blocks that actually have the file data; these are represented as an array of direct blocks, a single indirect block (i.e., a pointer to an INDIRECT block that has pointers to DB blocks) and a double indirect block (a pointer to an INDIRECT block that has pointers to INDIRECT blocks that have pointers to DB blocks).

Thus, you should create a file inode structure that resembles

```
typedef struct inode_t {
  // the file metadata
  unsigned int size;
  uid_t user;
  gid_t group;
  mode_t mode;
  struct timespec access_time;
  struct timespec modify_time;
  struct timespec create_time;

  // the locations of the file data blocks
  blocknum direct[...];
  blocknum single_indirect;
  blocknum double_indirect;
} inode;
```

Of course, similar to the directory node block, you'll want to set the number of direct pointers so that your structure is BLOCKSIZE.

2.7 Data block (DB)

File data blocks only contain user data. Thus, their structure is easy:

```
typedef struct db_t {
  char data[512];
} db;
```

You don't really need to have a separate structure for this, but I included just in case you wanted to.

2.8 Free block (FREE)

The final piece of the puzzle is where to store the information about the free blocks. You could maintain a separate data structure, but instead, you can simply use the free blocks themselves to store the data. To do so, you'll maintain a singly linked list, where each entry in the free block list has a pointer to the next entry. Thus, each free block actually will store data (just a blocknum pointer); it will look like

```
typedef struct free_t {
  blocknum next;
  char junk[508];
} free;
```

Of course, in your `3600mkfs` you will need to initialize all of these free blocks, with the first pointing to the next, pointing to the next, etc. Also, whenever you need to allocate a new block, you will need to “pop" the first free block off of the free block list (i.e., change the pointer in the VCB to point to the next element in the list). If the VCB pointer is invalid, then you've run out of disk space (no more free blocks).

3 Blank disk

To put it all together, when you go to format a blank disk, you're actually going to touch every block:

- **Block 0** You are going to create the VCB block as block 0, set the root `blocknum` to be 1 (and `valid`) and set the first element of the free list to be block 3 (and `valid`).
- **Block 1** You are going to create a DNODE block in block 1. The first `direct` block (`direct[0]`) will point to block 2 (and be `valid`), and all other elements of `direct` will be marked invalid. Both `single_indirect` and `double_indirect` will be invalid.
- **Block** 2 You are going to create a DIRENT block in block 2. There will be two entires: one for `.` and one for `..`. Both will have type "directory" and will have `blocknum` of 1 (pointers to the root DNODE) and be `valid`. All other entires will be `invalid`.
- **Block 3–N** The remainder of the blocks will be FREE blocks. The `next` value of each will point to the next block (and be `valid`), with the exception of block N (the final block), whose `next` will be invalid.

4 Implementation tips

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