Computer Science 432
Assignment 4 — The Williams Ultimate File System, Part I
Due Sunday, April 19.

Over the next few weeks we will be developing parts of the Williams Unix File System (WUFS). WUFS is a format suitable for organizing directory structures on smaller disks. We'll experiment with its use on files mounted by the `loop` device. This project is broken into two parts: in this lab we'll extend a utility to format an empty file system and the next lab we'll implement those extensions in a kernel module that supports interactive use of WUFS filesystems.

For the moment, we'll focus on a WUFS file system construction utility, `mkfs.wufs`, one of a suite of programs that support `mkfs`, an interface that system administrators use to make general file systems (see `mkfs(8)`). When this *userland* program targets a file, or block device,[1] it writes all the blocks on the target in a pre-determined format that can be interpreted as an empty file system.

Here's an outline of the process:

1. Determine the size of the file or device.
2. Write/reserve an empty block to hold 512 or more bytes of boot code.
3. Write a small description of the file system, its *superblock*.
4. Check the remaining data blocks of the disk for basic functionality. Any that fail are *bad*.
5. Write two free lists—represented by *bitmaps*—for inodes and data blocks.
6. Reserve space for metadata-carrying inodes.
7. Write an empty *root* directory, off of which the entire file system will hang. The root directory also includes `.` and `..` (both of which reference the root directory itself), and (possibly) files that hold any identified bad blocks.

There are a lot of technical details here, but they're all easily understood with just a little bit of study. Take your time to read this document and peruse the starter code. Once you have a good understanding of the strengths and weaknesses you'll be in a good position to understand any modern file system and propose reasonable extensions to WUFS.

What's Been Done.

I've started writing a version of `mkfs.wufs` that will lay out file systems according to a very simple protocol. As we'll see in a moment, some of the decisions I've made in the WUFS design are *unfortunate*. We'll think of these as *opportunities for improvement*.

Basics. The block size for the file system is as small as can be supported on most modern block devices: 1Kb or 1024 bytes. All I/O to the disk is accomplished with reads and writes in multiples of this block size. For example: we're required to reserve a boot sector of 512 bytes at the beginning of the disk for OS-specific boot code, so so we'll reserve the first full 1Kb (found at logical block address (lba) 0) for this purpose. In `mkfs.wufs` we will live with the abstraction that the *raw device* is a linear file of 1Kb blocks. We'll be using some I/O routines that allow us to update blocks of the raw device in arbitrary order. In reality, this abstraction is likely hiding a lot of details about how to read and write blocks on the actual physical device. The layout of most of the following structures is documented in `wufs_fs.h`.

[1] A block device natively transfers data in chunks or *blocks*. Disk drives and tape drives are the most common examples of block devices.

Superblock. The superblock (always found at lba 1) is identifiable by a pre-determined *magic number*. It keeps track of counts of inodes and blocks, sizes of inode and block bitmaps, and a limit on the size of files. File size limit is not arbitrary, but the indirect result of other important decisions. The number of blocks is easy to determine if we know the size of the underlying device. To compute the number of inodes, we'll use the rule of thumb that we typically need 1 inode for every 3 data blocks used (this is worth thinking about). The sizes of the bitmaps should be sufficiently large to refer to every inode and data block. Bitmaps are always padded out to full block sizes.

Bitmaps. As mentioned above, the file system maintains two free-lists as bitmaps. Each bitmap contains the 8 bits in each byte of one or more blocks. The first bitmap keeps track of available inodes. The first bit refers to inode 1, the second bit keeps track of inode 2, *etc.* A zero indicates the corresponding inode is unused, while a one suggests the inode is either allocated or, if part of the padding, illegal. There is no inode zero.

The second bitmap, which starts on a logical block boundary, keeps track of data blocks. The first bit refers to lba 0 (the boot block), the second bit corresponds to the superblock, *etc.* Blocks which are available for use on the disk have corresponding bits that are zero. All others are one.

The Inode Table. The 32 byte inode structure is designed to allow packing 32 inodes tightly into a single disk block. When we determine how many inodes we would like, we round the number upward to the next multiple of 32 so that the entire table fits perfectly into some number of blocks. This table is found beginning at the first block after the block bitmap, and often extends many blocks into the disk. Since the inodes describe the *metadata* associated with a file, these must be carefully initialized. **There are unfortunate limits to the inode structure I hope you will fix.**

The Data Blocks. Everything that follows is a data block. Before we format the file system we attempt to read every block in this area. Anything that's not readable is a bad block. (We'll mark these blocks used and collect them into special files we reserve; on some systems these bad-block-carrying files are found in a directory called `lost+found`.) All other data blocks free for whatever use we want.

The Directory Structure. One use for a data block is to hold a *directory*. A directory is simply a list of *directory entries*. This structure, as you will recall, is an inode-filename pairing. Because directories are sometimes written by very low level routines, there's quite a bit of pressure to make sure the size of the structure allows tight packing within a block. So, **I've made another unfortunate decision**: the directory entry structure is only 16 bytes long. Allowing for 2 byte inode numbers, only 14 bytes are left for saving a filename. This filename is padded out with zeros to fill the 14 bytes. This is sufficient for `JefferyAmherst` but not `EphraimWilliams` or `AbigailZimmermannNiefield`. By the way, since there is no inode 0, an empty directory entry is indicated by a zero in the inode field (aha!).

The Inode Structure. This is the 32 byte structure that describes a physical file. Currently this includes nine data block addresses. The *only* reason nine was chosen because that made the structure 32 bytes long. **Another unfortunate decision** is not the number, but the *interpretation* of these pointers. All nine pointers are direct, meaning that each refers to one 1Kb block of data. The maximum file size is thus (drum roll...): 9K. While you can recursively fit another WUFS file system in 9K, my recipe for gumbo would not fit easily in that small cranny.

Gathering Parts.

You can `wget` a tar file that will bootstrap this assignment, from `core`:

```
wget http://core/wufs-starter.tar.gz
```

This distribution contains sources for `mkfs.wufs` and other goodies.

Attached to this document I've dumped a very small WUFS file system. I would like you to annotate these pages as carefully as you can, to make sure your understanding of the layout is consistent with mine. I'm not going to grade these directly, but if you have questions, it will be useful to have this annotated sheet in hand. As you take a look at these bits you will find it useful to read through the `mkfs.wufs` code find answers to questions. If you remain confused, see me. Do not, however, make any decisions before you understand this unusual creature.[2]

Next, we'll be getting an assignment that will involve extending a working WUFS file system (a kernel module) to support changes you make this week. Be warned: future earnings reflect past decisions.

Getting to Work.
I would like you to perform the following tasks:

1. Create a file to use as a target for your tests. The easiest way is to use `dd` (device-to-device copy), to copy from `/dev/zero` (a source of zeros). The man page is quite helpful. To make a file, `a4.img`, that is 20 1K blocks long:

   ```
   dd if=/dev/zero of=a4.img bs=1024 count=20
   ```

   The result is a 20Kb file. It will neither grow nor shrink with time or use.

2. Get the code working. I accidentally deleted all the bitmap operations from the code in the file `wufs.c`; they need to be restored. We'll discuss these operations in lab in some detail. None is very difficult. Each is an opportunity to practice writing something beautiful from scratch.[3]

   To build `mkfs.wufs` type

   ```
   make
   ```

   You can target your disk image with the following command:

   ```
   mkfs.wufs a4.img
   ```

   You can dump the image in a (more) readable form with:

   ```
   od -A x -at <a4.img >a4.od
   ```

3. (Only) think about what would be necessary to raise the upper bound on the length of filenames. What is appropriate? I'm not sure, but you might think about how to figure out what most users would be happy with.

4. (Only) think about what would be necessary to raise the maximum size of files from 9K to, say, 100K or more. This change, like the lengthening of the file name, should be *simple* since you're going to be responsible for supporting these changes in our next lab.

5. Implement these changes in `mkfs.wufs`.

6. Document the new features of your file system in a file called `README-a4`.

7. Construct a small file system using your new file system format in an image called `a4.img`. After our next lab, you should then be able to mount this file system and verify your changes have worked.

Turning In.
Turn in a clean directory that contains your modified code, your first file system (`a4.img`), and `README-a4` as `a4.tar.gz`.

[2]One wonders if God had seen a snake before He implemented one, whether he might have improved the design.

[3]By now, most of you have learned that there's almost no code on the web worth copying, so don't. This is varsity Programming; let's go to Nationals.