Computer Science 432
Assignment 5 — The Williams United File System, Part II
Due Sunday, May 3.

By now you have a working knowledge of the Williams Undergraduate File System (WUFS), and you have modified `mkfs.wufs` to support longer file names and large files. With this utility you have probably already created a file system and used the unix `od` (*octal dump*) utility to diagnose problems with your implementation. Assuming your implementation works, you're probably going to want to try to mount the file system and use it.

Before we can do that, we need to make modifications to the WUFS kernel module. The module, as it stands now, is a slight rewrite of Linus Torvalds' original Minux file system. It surrently supports the original WUFS layout which is, as you know, limited to 9K or smaller files with filenames that are 14 characters or less. Your task, over the next two weeks, is to implement the WUFS kernel module modifications that parallel your extensions to `mkfs.wufs` from Assignment 4.

⋆ ⋆ ⋆

Warning: This assignment will typically require the writing of, perhaps, a few hundred lines of code. Still, the details of (1) getting to the point where the experiment can begin and (2) correctly implementing this code requires considerable care. *If you are insecure about your solution to Assignment 4 or you are insecure about the various steps in this assignment it is imperative that you see me.* You are not penalized for asking questions. However, backing out of late night actions will seem like a penalty if you do not proceed with care.

⋆ ⋆ ⋆

As with Assignment 3 (our `wisdom` syscall implementation), I will lead you through many of the steps needed to prepare and test your solution. I suggest plugging through the steps in order, and checking them off as you progress. This is a bit dangerous (I hung the kernel a dozen times, killing the `init` process by accident, used up all of memory, and trashed one (thankfully: virtual) drive in the process), so please execute this assignment on your favorite `panic`.

Fetching and Building a Kernel.

Make sure you have the handout for Assignment 3 in hand. (A version of that handout can be found in `/usr/cs-local/share/cs432/a3`.) There are details about kernel building and `vmplayer` you will need from that document.

I will assume that you will be working with a kernel that is not too different from version 2.6.32.28. You may either build upon your work from Assignment 3, or you may fetch a new copy of the kernel from `core`. For the purposes of this document, I'll assume the root of that distribution is at `~/linux`.

◇ Step -1. Cleaning up.

Whatever you do, we want to force a fresh kernel build, so you need to type

```
make clean
```

This removes any object files or kernel modules you may have built previously.

Step 0. Configuration.

We need to add our WUFS module to the kernel.

◇ Step 0a. Grab Revised Kernel Configuration File (Do Once).

The first step is to grab a new configuration file from core:

```
wget http://core/config.cs432-wufs
```

This file directs the kernel to build a few modules we had not included before, including the `loop` block device and the WUFS filesystem. Once you've downloaded the file, move it into place:

```
cp config.cs432-wufs ~/linux/.config
```

(The initial dot in the destination filename is, of course, important.)

◇ Step 0b. Grab the VM and Lab Starter Code (Do Once).

For this lab, I've built a new virtual machine that boots into a 2.6.32.28 kernel, by default. From your home directory, download and unpack it now:

```
wget http://core/cs432-wufs.tar.gz
tar xvfz cs432-wufs.tar.gz
```

The result is the `vmplayer` machine called `CS432-WUFS`. (You can, if you wish, remove the old appliance, `CS432`.)

Next, grab the `tar` file that contains the WUFS kernel module source:

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

Once it is downloaded, untar it into the `~/linux` directory for filesystems:

```
cd ~/linux/fs
tar xvfz ~/wufs-module-starter.tar.gz
```

You should now find the source in the directory `~/linux/fs/wufs`.

◇ Step 0c. Update Make (Do Once).

Add the WUFS module to the `make` system. Every directory in the kernel tree has a `Makefile`, and they work together to build exactly the modules you want. First, though, we need to tell the system about WUFS. Edit the `Makefile` in the filesystems directory:

```
cd ~/linux/fs
emacs Makefile
```

This file consists, for the most part, of a list of modules that have to be made. They're kept in a variable, `obj-m`. Each file and directory that must be included is appended to this variable. Order is important, so search for the line that builds the `minix` filesystem (the filesystem whose demands are closest to ours):

```
obj-$(CONFIG_MINIX_FS) += minix/
```

and add a new line following, that looks like:

```
obj-$(CONFIG_WUFS_FS) += wufs/
```

Save the `Makefile`.

Step 1. Building the Kernel, Proper (Do Once).

We need to build the kernel so that the dynamic modules know the actual addresses of kernel values and variables.

◇ Step 1a. Build the Kernel Image.

Recollect that this is accomplished with:

```
cd ~/linux
make bzImage
```

This will take 5 minutes, but you will only have to do this once. This image is essentially the kernel found as the choice `Working 2.6.32.28 Kernel` in the boot menu in this lab's virtual machine (see below). If all works out, you should see

```
Kernel: arch/x86/boot/bzImage is ready
```

◇ Step 1b. Configure WUFS (Do Once).

Now, edit the file `~/linux/include/config/auto.conf` and add the line

```
CONFIG_WUFS_FS=m
```

anywhere in this file. When we set the configuration variable to `m`, it causes this filesystem to be constructed as a standalone *module*. Modules have the great feature that they may be dynamically loaded and unloaded as they are needed (or, more likely, as they need to be repaird). No booting or rebooting will be required.

Step 2. Build the kernel modules.

This step compiles all the modules we will need to use (and many more). It's important that we build them all because there are interdependencies. Again, this fresh build only needs to be done once.

```
make modules
```

Watch. You should see compilation of files from `fs/wufs` early in the compile process. Later you'll see it mentioned again when the individual object files are linked together to form a *kernel object* file, `wufs.ko`.

◇ Step 3. Verify.

Verify the modules we need exist. You should be able to find

```
~/linux/drivers/block/loop.ko
~/linux/fs/wufs/wufs.ko
```

The first module is not normally built, but we need it for this assignment. It allows you to mount a file as a block device. A similar approach is to used to mount `.dmg` or `.iso` files on production operating systems.

You will also find a copy of my `0x0EEF` file system (the one you annotated) at

```
~/linux/fs/wufs/dab-wufs.img
```

We will need this in a moment, in Step 7.

Helpful hint. In the future, when you're looking to re-make just the `wufs.ko` module, you can type

```
make -C ~/linux M=~/linux/fs/wufs
```

The `-C` switch says *the kernel distribution is rooted here*, and the `M=` variable simply identifies a module directory that needs to be rebuilt. Typically, we'll be sitting in `/linux/fs/wufs` (we're editing the code), and so we simply type

```
make -C ~/linux M=$PWD
```

I've made this the default target in the WUFS `Makefile`.

⋆ Now is a good time for a break. ⋆

Mounting our first WUFS filesystem

We'll run a little experiment that will allow us to mount my `0x0EEF` version of the filesystem and test some of its characteristics.

◇ Step 4. Boot the Virtual Machine

Boot the `Working 2.6.32.28 Kernel` under `vmplayer`. You may be able to use other versions of the kernel, but the word *working* feels pretty good on some days of the week.

◇ Step 5. Upload Drivers

Log in as `root/rootuser` and write down the IP address it gives you. Recall how we *pull* files from the `panic` host. Copy the following files from `panic` to the home directory for `root`:

```
linux/drivers/block/loop.ko
linux/fs/wufs/wufs.ko
linux/fs/wufs/dab-wufs.img
```

You can use `scp`, or if you're facile with `sftp`, that works with fewer passwords. Later, you may want to copy up disk images *you've* create with your own `mkfs.wufs`.

◇ Step 6. Install the Kernel Modules.

Kernel modules are loaded with `insmod` and unloaded with `rmmod`. Let's load the two kernel modules we just built:

```
insmod loop.ko
insmod wufs.ko
```

Neither of these actually generates any output, but they do drop messages in the kernel log file. You can see the latest messages by typing:

```
dmesg | tail
```

You should see lines like:

```
loop: module loaded
WUFS: filesystem module loaded.
```

At any time you can see this list of loaded modules by `cat`-ing the file:

```
/proc/modules
```

and you can see the list of loaded filesystems by `cat`-ing

```
/proc/filesystems
```

The `/proc` directory is the result of the `proc` filesystem begin mounted on it! Way self-referential, man. (**Reversing this step.** Don't do this now, but uninstalling modules looks like

```
rmmod loop
```

without the `.ko` suffix.)

◇ Step 7. Setup the `loop` Device.

The `loop` devices is, essentially, a virtual hard drive whose content is backed by a file. When we installed the `loop.ko` module, several devices in the `/dev` directory (named `loop0`, `loop1`, etc.) become instances of this type of virtual disk. To associate a file with the device, we use the *loop setup* command, `losetup`:

```
losetup /dev/loop0 dab-wufs.img
```

We could have used any other block device, but they would probably arrive at our door with no data. We would have to copy data from a disk image (like `dab-wufs.img`) to the device with the `dd` command (that we're already familiar with). Using the `loop` device saves us this hassle.

Normally we would create the image file with `dd`, copying the requisite number of blocks to a blank image file. We would then use `mkfs` to layout the file system. The image `dab-wufs.img` is the product of `mkfs.wufs`, as you know.

(**Reversing this step.** Disassociating the `loop` device from the image is accomplished with

```
losetup -d /dev/loop0
```

Don't do this now.)

◇ Step 8. Mounting the Block Device.

Block devices (like `/dev/loop0`) which have been formatted correctly with some version of `mkfs` are *mounted* on an empty directory in another active file system. When the filesystem is mounted on directory `/mnt`, then all files within the file system appear to hang off of `/mnt`: a file, `.bad-0`, for example, would be accessible at `/mnt/.bad-0`. You need to be careful to make sure the target mount point is empty. If it isn't, the files within the directory become inaccessible until the filesystem is unmounted.

Most version of unix provide an empty directory, `/mnt`, used precisely for temporary mounting of filesystems. Let's do it:

```
mount /dev/loop0 /mnt
```

Things should be quiet. Looking at the kernel log, you'll see messages that indicate the filesystem is not `ext2` or `ext3`, but it *is* a version 0 `wufs`. Wowza: progress.

(**Reversing this step.** To unmount a the file system, we use `umount /mnt`. Your current directory may not be in the filesystem you're unmounting and you may not have any of the filesystem's files open when you do this. Typically, one moves to one's home directory before unmounting a filesystem. Note the silly spelling and, yeah, don't be unmounting now.)

Step 9. Experiment.

You.Are.A.Scientist: Kick tires. Poke things. Make zapping noises.

◇ Step 9a. Size Things Up.

You should be able to go to the `/mnt` directory and determine the file system size:

```
cd /mnt
df -h .
```

You're an expert at understanding `df`.

◇ Step 9b. Hog the Disk.

It looks like there is only `4K` left, so you should be able to do something like

```
dd if=/dev/zero of=/mnt/a bs=1k count=4
```

but trying something outrageous like

```
rm /mnt/a
dd if=/dev/zero of=/mnt/a bs=1k count=5
```

will get an I/O error after four 1K records: the filesystem is full.

Notice that you can

```
touch b
```

(which creates an empty file) on a full file system, and touch many more files as well. How many files can you create on a disk with no space? Good final exam question: How many empty files can be stored on a `0x0EFF`-version Wufs device with 4k available?

◇ Step 9c. Verify the Filename Length.

Should be, of course, 14.

◇ Step 10. Tear Down.

Take down the file system. Change to your home directory, unmount the `/mnt` partition, delete the `loop` association.

If you repeat Steps 7 and 8, you'll find that the filesystem is *persistent*; changes you make are saved to the disk image.

⋆ Now is a good time for a break. ⋆

Remember, if you shut down the virtual machine, you'll lose the kernel modules you've installed. (To combat the fatigue associated with setup, I like to create a little `makefile` that installs and uninstalls modules, creates file systems, etc. I `scp` this one `makefile` up to the virtual machine and then type an appropriate `make` command. It asks for a bunch of passwords, then makes and mounts the filesystem.)

Your mama is proud of you.

Upgrading Wufs.

Your ultimate goal is to build a version of `wufs.ko` that supports your `mkfs.wufs` extensions. The following steps will help you along the way. Still, much of the work is left for you to discover. You might read the remainder of this document before you start your attack.

◇ Step 11. Git Back.

Back on `panic`:

```
cd ~/linux/fs/wufs
```

If you're git-savvy, you might tell `git` to start a new branch and add the `wufs` source to the linux kernel tree:

```
git branch experimental
git checkout experimental
git add ~/linux/fs/wufs
git commit -m "Base release of wufs (version 0)."
```

This is not necessary, but you may find it useful. (If you're not `git`-savvy, why not read `man gittutorial`?)

◇ Step 12. Upgrade `wufs_fs.h`.

My version of `wufs_fs.h` is the copy we started with in Assigment 4. You should *copy* your final version of `wufs_fs.h` into place. This will break the system. You can see how bad things are with

```
make -C ~/linux M=$PWD
```

Errors? Now's a good time to learn how to compile and parse errors from `emacs` (check out `M-x compile` and `M-x next-error`). It is possible that the system will compile with no errors. (Did you complete Assignment 4?!)

The "main" file is `inode.c`. This file contains the module `init` and `exit` routine (which shouldn't be modified), as well as a routine to fill in superblock information. Make sure that your notions of file name length and maximum file size are represented in this file's logic. Once everything seems logically correct in `inode.c`, you can compile just that file with

```
make -C ~/linux M=$PWD inode.o
```

◇ Step 13. Data Blocks.

It is very likely that most other files will compile as well. The one file I know will break, logically, is `direct.c`. This file contains two important routines:

```
wufs_get_blk
wufs_truncate
```

These two routine are responsible for the (`wufs_get_blk`:) lookup/generation of blocks associated with inodes, and (`wufs_truncate`:) shrinking the size of the file, returning unused data blocks to the pool of free blocks. This file is worth printing (copy it to a lab machine, and `enscript` it from there).

If you limit yourself to very small files (viz. files that make use of only direct links), the module may work. To test this out, perform the above experiment with a filesystem generated by your solution to Assignment 4 and this new `wufs.ko` based on the version 1 (`0x1EEF`) `wufs_fs.h`.

◇ Step 14. Block Allocation.

Copy `direct.c` to `indirect.c` (I'm predicting you used indirect blocks in your solution to Assignment 4.) This will allow you to keep the version 0 approach around for reveiw. If you do this, edit the `makefile` to use `indirect.c` and *not* `direct.c`. `git`-ers will want to

```
git add indirect.c Makefile
```

and then commit.

◇ Step 15. The Long Haul.

Upgrade `wufs_get_blk`. This will take a long time. Take many breaks and write this code only when you're really awake. The purpose of this function is to determine the address of a logical block in a file. The caller will pass a `buffer_head`, and your ultimate goal is to call `map_bh`, which will assign the logical block address (type `block_t`) to the `buffer_head`.

We will discuss this in class. Probably several times. Please listen carefully and ask questions. The attached sheet contains descriptions of VFS routines you'll likely need to call.

◇ Step 16. Build and Test.

Build the module and, if you get no errors *or warnings*, upload it and test the filesystem on the virtual machine. You should be able to create large files (more than 9K; make sure your file system is large enough!) with non-trivial content. I like to use commands like:

```
dd if=/usr/share/dict/cracklib-small of=test bs=1k count=50
```

When you tail this file it should look like the middle of a password-cracking dictionary. Everything will work well (I hope!), but don't do anything that will write over or shrink the size of a file. The `rm` command, for example, will probably fail hard.

◇ Step 17. Cut it Short.

Upgrage `wufs_truncate`. The purpose of this function is to shrink the size of the file associated with an `inode`. Before this routine is called, the size has been adjusted downward, but the unused blocks have not been reclaimed. The `block_truncate_page` command will free all logical `buffer_heads` that have been associated with blocks that lie beyond the end of the file (you wil notice that `wufs_get_blk` is a parameter to this function), but it does not know how to return the logical block numbers to the pool. The remainder of this function should perform `wufs_free_block` for every block number that is no longer associated with the file. It is also responsible for "forgetting" any `buffer_heads` that were used to read indirect blocks.

This code requires great care.

◇ Step 18. Test.

Again, build the module and test it out. If it works, you should be able to create, shrink, and remove files without ill effect.

◇ Step 19. Turn It In.

When you are finished with your upgrades, clean the source directory, and turn in a `tar` file of the `wufs` source. The following commands will create a compact file, `a5.tar.gz`, in your top level directory:

```
cd ~/linux/fs/wufs
make -C ~/linux M=$PWD clean
cd ~/linux/fs
tar cvfz ~/a5.tar.gz --exclude .git wufs
```

Things to Think About.

It's important that you keep the following things in mind as you write your kernel code:

1. Remember that you're in the kernel. You do not have access to anything from the standard C library. Routines like `malloc` and `printf` are not available.
2. The kernel is multithreaded. You need to make sure that changes you make to data structures that are globally held (ie. non-local resources) are protected by appropriate locking:
    - There are a variety of locking mechanisms available, some of which I outline below. Generally, I would advise against creating new locks (the proliferation of locks increases the chance of deadlock), but you should feel free to use those locks that are available.
    - The expected path through a routine should avoid locking. If a path is high-traffic, locking will impact performance.
    - Try to avoid acquiring more than one lock at a time. Holding two locks invites the opportunity for deadlock.
    - Hold locks for as short a time as possible (but, as Einstein wished he said, *no less*).
    - If, after locking a resource, the state of the resource appears to have changed, back out of the critical setion, release all speculative resources, and retry the operation. See the `direct.c` version of `wufs_get_blk` for an example.

3. Realize that the `wufs_get_blk` and `wufs_truncate` are at odds (one builds up the file, the other tears it down), and that `wufs_truncate` indirectly calls `wufs_get_blk`.

4. Read up on `printk`, the primary tool for gross debugging of kernel problems.

5. All of the routines we call are defined somewhere in the kernel source tree. VFS-specific routines are often found in `~/linux/fs` files. The `grep -R` command can be your friend. Unfortunately, much of the linux source is very (very) poorly documented. *C'est la guerre.*

The following are some of the functions that may be of some help to you:

```
BLOCK ROUTINES FROM linux/fs/wufs/bitmap.c:
wufs_new_block(struct inode *inode)
wufs_free_block(struct inode *inode, unsigned long block)
  Logically allocate (and free) data in a disk's data blocks (based on the block bitmap).

BUFFER HEAD ROUTINES FROM linux/include/linux/buffer_head.h:
sb_bread(struct super_block *sb, sector_t block)
  Read a logical block from the disk associated with a filesystem's
  superblock. If the address is 2, for example, you'll get the first block of
  the inode bitmap. The result is a new buffer_head pointer.

sb_getblk(struct super_block *sb, sector_t block)
  Get a buffer_head associated with a logical block, but don't read it from
  disk. For example, you might call this if you're constructing a new
  block. In such cases, consider marking it new (below). Never fails.

char *bh->b_data and size_t bh->b_size
  These fields allow you direct access to the data associated with a
  buffer_head, bh.

map_bh(struct buffer_head *bh, struct super_block *sb, sector_t block)
  Map this buffer_head to the logical block address for a filesystem.

brelse(struct buffer_head *bh)
  Release a buffer_head. If it was dirty, it gets scheduled to be written back to disk.

bforget(struct buffer_head *bh)
  Like brelse, but throws away changes that might have happened.

set_buffer_new(struct buffer_head *)
  Indicate this block is new, as opposed to having been read from disk.

set_buffer_uptodate(struct buffer_head *)
  Mark this buffer ready to be written to disk

INODE UPDATING ROUTINE FROM linux/include/fs.h
mark_inode_dirty(struct inode *inode)
  Indicate an inode has data that should eventually be written to disk.

DIRTY INODE BUFFER ROUTINE FROM buffer_head.h
mark_buffer_dirty_inode(struct buffer_head *bh, struct inode *inode)
  Mark a buffer associated with an inode as dirty. This in-memory version is
  authoritative and should be written to disk.
```

LOCKING ROUTINES FROM linux/include/spinlock.h
read_lock(rwlock_t *lock), read_unlock(rwlock_t *lock)

write_lock(rwlock_t *lock), write_unlock(rwlock_t *lock)
Lock (and unlock) a kernel reader-writer lock for read or write access. A reader-writer lock is used in direct.c to control access to the direct pointers in the inode.

spin_lock(spinlock_t *lock), spin_unlock(spinlock_t *lock) (from spinlock.h)
Lock (and unlock) a spinlock. Spinlocks hurt performance, so hold them only for short periods of time. Spinlocks are used to control access to bitmaps by operations in bitmap.c.

lock_buffer(struct buffer_head *), unlock_buffer(struct buffer_head *)
Signal that the data associated with this buffer head is in transition. We lock the buffer head before, and release the lock after changes have been made to buffer heads.

BLOCK TRUNCATION ROUTINE FROM linux/fs/buffer.c
block_truncate_page(struct address_space *mapping, loff_t from, get_block_t *get_block)
Free all the buffers that have been mapped to blocks beyond offset from, using the get_block (typically wufs_get_blk) to generate the logical block addresses. Notice that wufs_get_blk and wufs_truncate can interact (possibly negatively) through this interface.