Section 9
November 30, 1027

** Read the caution at the top of the assignment, QEMU simulates persistent storage, so writes will persist between runs of QEMU. To mitigate this, run `make clean` to start with a fresh file system. On the other hand, if you want to maintain the state (for testing crashes), you *must* not remove the old image. You can run xk on the old disk image by running "make qemu".*

File Append:
This will require you to add a scheme to either (a) add more extents at the bottom of the file, or (b) implement direct and indirect blocks similar to how we discussed in class. (a) will be nice because you won't have to change how mkfs works. If you are looking for a challenge, the direct and indirect blocks will require some more work.

Looking at `struct dinode` you will see there is a field char `pad[44]`: this is free space where you can add new fields, just alter this number to keep it a multiple of 2. Right now inodes are 64 bytes, but you can easily extend that to 128 bytes if you like (although that shouldn't be necessary).

You need to implement a helper function to allocate and deallocate block on disk. To do that, you have to maintain information about what blocks are still free on the disk. You can do that via bitmap. Take a look at how "mkfs.c" does that.

File Create:
File create is going to require a few steps:
1) You will need to allocate space in the inode file for the inode, which may require allocating another block.
2) Once allocated, there needs to be a way to reference it from the shell. It will need to be added to the root directory. This requires finding the root file, and adding it to the end of the file in the form of a `struct dirent`.

Crash Safety:
It is paramount that persistent media is consistent. We want all file operations to be atomic. It is more favorable to undo partial work, rather than be up to date at the cost of consistency. That being said you will need to come up with a scheme to maintain consistency. Today we will offer up one suggestion, you may use it or another method to ensure consistency.

Write Ahead Log:
Simplified redo log: A simple logging system I will layout treats file operations as transactions, and transaction is completely atomic, either it completes or it does not. There is a log file, You can define a static number of blocks for the log anywhere where it will not conflict with new files (similar to the swap region). Log size of 30 blocks is enough.

The operations will be:
1) Start transaction/operation
2) Write to block(s) (with block number and the new data [just write the whole 512 bytes])
3) Keeping these block(s) in the block cache (by setting the dirty bit).
4) Write a commit log.
5) Flush data to disk, either evicting from block cache or just flushing and setting the "clean" bit.

* *Caveat* (that you can take advantage of): There won't be concurrent file writes, so you only have to deal with one transaction at a time. Once that transaction is finished, you can clear the log and get ready for the next operation.