Computer Science 432
Spring 2022
Lab 7: Supporting large files. Due on Wednesday, May 4.

Objective. To appreciate the `inode` structure.

Discussion. Common to all unix implementations is the inode-based implementation of the on-disk file structure. Instead of storing files as large contiguous stretches of data, files are stored as collections of fixed sized *blocks*. The *block addresses* of the blocks associated with each file's data are stored directly or indirectly in a metadata on-disk structure called an *inode*.[1]

There are a fixed number of `inode` structures ever available on a disk, and the number is determined when the disk is initially formatted. The reason for this is because the `inode` is specifically designed to be a power-of-2 bytes long so that a whole number of them may be allocated within a single disk block. This makes `inode` access fast, computable, and efficient. The downside, of course, is that the structure of a file is rather inflexible.

How we make use of the metadata stored within the `inode` is entirely up to the implementor of the operating system. In particular, the block addresses of the blocks that hold the data for a file must be directly or indirectly accessable from the file's `inode`. Non-address data takes up 12 bytes of store (2 each for `type`, `major`, `minor`, and `nlink`, and 4 for `size`), so a 64-byte `inode` only has room for 13 4-byte block addresses. How ominous. If each of these addresses directly stored a pointer to a data block, the limit on the size of the file would be 13K. This is simply not reasonable. (For example, while the un-made `xv6` lab 6 has an average file size of 4K, it has 3 files larger than 13K.)

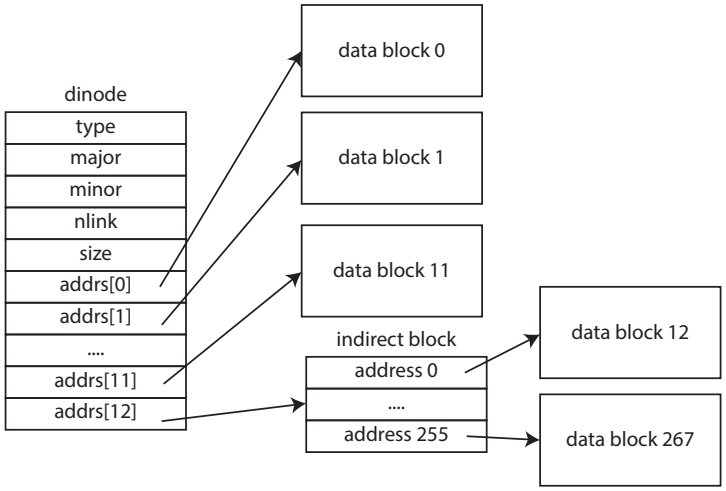

FIGURE 1. THE `dinode` STRUCTURE NATIVE TO `xv6`.

The approach used to by `xv6` (see Figure 1), is to have the first 12 addresses point directly to data blocks while the 13th points to a dedicated block of $1K/4b = 256$ addresses that *indirectly* point to logical blocks 12 through 267. By having the single indirect block appear last in the inode, it makes very small files quickly readable; these files use a single disk read for each block. Files with more than a dozen blocks access their data, indirectly, using up to two disk reads per block.

Sadly, even this is insufficient. If we were to work on even fairly small projects (like Lab 6 for `xv6`), the compiler would generate executables that could not be represented by this structure. Our goal this week is to develop a modified `inode` structure (see Figure 2) where the first 11 addresses are direct, the 12th is indirect, and the 13th is *doubly indirect*. A doubly indirect address is a pointer to a block of addresses to blocks that contain direct addresses. The total number of 1K blocks accessible in such a structure is $11 + 256 + 256 * 256 = 65803$, allowing for files that approach 65 megabytes in size. Incredible inodes, Batman!

[1] In `xv6` the `inode` structure that appears on disk is a `dinode`, while the larger cached version found in the kernel's memory is an `inode`. We will refer to the organizational structure universally as an '`inode`', but the reader is reminded that context—memory or disk—must be used to distinguish between the two related forms. Good luck!

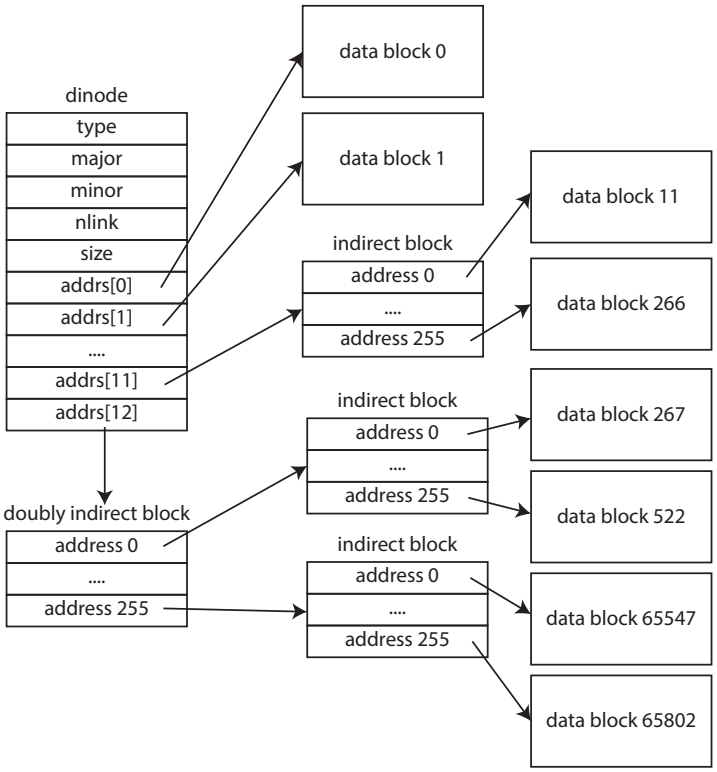

FIGURE 2. THE MODIFIED `dinode` STRUCTURE USED IN THIS LAB.

The Assignment. The basis for this lab is found in the `lab7` repository (use your user id rather than `22xyz`):

```
$ git clone ssh://22xyz@lohani.cs.williams.edu/~cs432/22xyz/lab7.git
```

Make sure that you've read Chapter 8 from the book. The sections on the buffer cache and inodes are particularly important, here.

The goal of this lab is to modify the `xv6` code so that it supports a version of `struct dinode` (and `struct inode`) whose `addrs` structure contains three types of block addresses:

- Direct block addresses. *Direct addresses* identify disk blocks that contain the data stored in files. This week, the first 11K of data is accessed through direct addressing.
- Indirect block addresses. Any *indirect address* points to a block of 256 direct addresses. Thus an indirect address allows one to access 256K of a file's data. As with the current `xv6` implementation, our modified `dinode` will contain a single indirect address, in the second-to-last `addrs` slot. The next 256K of data is accessed through this one link.
- Doubly indirect block addresses. A *doubly indirect address* points to a single block of *indirect* addresses. This is not a feature of the current `dinode`, but we would like to have one doubly indirect block address in the last `addrs` slot. The next 64M of data is accessed through this one link.

Here is an approach to implementing this revised version of the inode structuring mechanism:

1. The definition of the disk-based `struct dinode` is found in `kernel/fs.h` and the memory-based `struct inode` is found in `kernel/file.h`. You may find it useful to define or modify constants associated with these structures to support this new approach to file layout. Nothing you do should end up changing the *size* of the `addrs` array in these structures, just the *interpretation* of each of the array's addresses.
2. The function `bmap(ip,bn)` (see `kernel/fs.c`) is responsible for determining the block address of logical block `bn` of the `struct inode` pointed to by `ip`. This is the one location where the mapping of `addrs` entries to logical blocks is performed. First, look at the existing code and make sure you understand what each statement does. Note, in particular, how `bmap()` acts when the requested block number is out of range.

3. Now, with a sound plan in mind, make changes to `bmap(ip,bn)` that will allow it to compute and return disk block addresses for a much larger range of values in `bn`. Areas of focus:

   - Make sure that you always call `brelse` for any block that you `bread`. Doing this keeps the reference counting in the buffer cache accurate and avoids deadlock.
   - Make sure you only allocate block(s) as they are needed.
   - Handle errors with grace.

4. The `bmap()` method is responsible for *allocating* blocks associated with file growth. The `itrunc()` is responsible for recycling all the blocks associated with a file when it is deleted. Don't forget to deallocate indirect blocks, as well.

5. The `mkfs` system (the Ubuntu utility in the `mkfs` subdirectory) depends on your definition of `NDIRECT` and `NINDIRECT`. If you have made changes to these constants, it is best to remake everything from scratch with:

   ```
   $ make clean
   $ make qemu
   ```

   This will re-make the `mkfs` utility and cause it to re-make the file system image, `fs.img`.

   If you encounter logical problems with your code that cause it to crash, it is best if you remove `fs.img` and re-make it from scratch:

   ```
   $ rm fs.img
   $ make qemu
   ```

   Because the file system is *persistent*, you may have structural bugs that persist as well. Be careful!

6. The utility `bigfile` will create a very large file called `big.file`, limited only by the layout and interpretation of the `inode` structure. You should be able to run `bigfile` successfully, remove `big.file`, and then pass the `bigfile` test again.

7. If you have problems with debugging your allocations, I've provided a user utility (and system call), `df`, which reports the number of free blocks on the current disk.

8. When your ready, `make clean`, and then push your work to the server for review and grading!

Thought Questions. Please think about the following questions in preparation for meeting in small groups:

1. The contents of the `struct dinode` are read from the disk into a memory-resident version called `struct inode`. It's important to make sure that the `addrs` array is faithfully copied from one structure to the other. Where does that happen?

2. The source code for `mkfs.c` has not been changed from the release version of `xv6`. In particular, it does not know about your *doubly indirect addressing* mechanism. What assumptions are we making about the use of `mkfs`? Are these reasonable?

3. Why is it necessary that `itrunc` be correctly written? Suppose we did not make changes to `itrunc`. What would happen?

4. How does the `bigfile` test work?

⋆