CSE 5433

Lab 5 Documentation

Team #7 (Colin Drake and Ananth Mahadevan)

4/23/2013

### Lab5 file system design

We started our file system implementation by first learning and understanding buffer\_head, block devices and their corresponding I/O operations. We further split the milestones given in carmen into smaller milestones. Our first milestone was to create a module and test if it loads/unloads successfully. After this was completed, we moved onto mounting and unmounting the file system which included registering, unregistering the file system as well as creating a dummy superblock.

After we were confident about the above mentioned steps, the next major step was to design the layout of our filesystem (location and size of superblock, inode bitmap, data bitmap, inodes and data blocks). We looked into the layout of the BFS filesystem and modelled our filesystem based on that. We started with a bitmap for the inodes and data blocks, but realized that using a byte level representation for the inodes would be an equivalent solution that could be easier to implement. We started with a size of 1024 bytes each for the superblock, inode bitmap and the rest of the meta-data. However, we decided to go with a blocksize of 512 bytes after encountering numerous design concerns by having differing block sizes for various block segments.

Next, we created the mklab5fs utility that writes the initial meta-data to the disk. In order to help us with our testing, we created default entry for a couple of example files in the utility. After writing the superblocks to the device and reading it back successfully, we decided to display the current (“.”) and the parent (“..”) directories. These entries are not stored on the disk, but are displayed “on the run” in our implementation of readdir().

We faced quite a few issues in our next milestone which was to display the list of files using “ls”. We were initially unable to display the files, because we did not mark the superblock as dirty in another initialization function. After making the changes, we were able to display the files, but the permissions were not being displayed. Instead, there were questions marks (?) in its place. The solution was to add the mode to the files as well as writing the details of the files to the superblock.

After successfully displaying the files, we started working on creating a file (touch <filename>). We intentionally left one inode in between 2 inodes in the inode map free. Our program was able to find this empty inode, however the last file was not being displayed. This once again turned out to be a problem with not writing to the superblock. Lesson learnt! Once this was done, the next task of removing a file was quite straightforward and we were able to complete it without much difficulty. This implementation zeroes out the inode data and in addition removes the associated data blocks if there are no more inodes that link to it. This is done by checking the inode number, which is what the data blocks are derived from, and the inode->i\_nlinks variable, which signifies how many inodes are pointing to that same data. Thus, unused inode and data blocks are freed for later use by the system.

To support file I/O (manipulation of the data blocks), the generic read and write functions provided by the kernel were used. Unfortunately, due to time concerns we found ourselves only able to support writing to single data blocks. Finally, to implement hard links, our solution uses a different approach than most filesystems. Rather than having directory entries that point to inodes, our filesystem embeds equivalent data into the inodes themselves. This results in a slightly less elegant solution to the problem, but the functionality provided is equivalent. In essence, is\_hard\_link and data\_block\_to\_link\_to fields are stored. These are filled out when the link() function is called, and from there, get\_block() uses these to determine whether to calculate the data block itself or derive it from another inode’s. This allows read and write address space operations to work with regular files and hard links. In retrospect, the directory entry approach is cleaner, but didn’t occur to us at the time.

### Limitations of our file system:

Number of inodes - 512 (due to decision to use a byte-map instead of bit-map)

Number of disk blocks - 512 (likewise)

Files supported - 512

Size of a block - 512

#### **Previous Issues faced**

1. Unable to handle kernel paging request at virtual address 0x...

After filling in the relevant details into the superblock datastructure, in the function that writes the superblock to the disk, we had a kfree(sb) call which was the cause of this issue. Without allocating, we were trying to free a memory region which resulted in the above confusing (and misleading!) error.

References:

1. [www.haifux.org/lectures/120/writing-linux-2.4-fs/writing-linux-2.4-fs.html](http://www.haifux.org/lectures/120/writing-linux-2.4-fs/writing-linux-2.4-fs.html)

2. <http://www.tldp.org/LDP/lki/lki-3.html>

3. <http://en.wikipedia.org/wiki/Ext2>

4.<https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout>

5. <http://tldp.org/LDP/tlk/fs/filesystem.html>

6. <http://inglorion.net/documents/tutorials/tutorfs/>

7. <http://unix.stackexchange.com/a/4641>