CS 111 Discussion Notes—Lab 3
Updated 6 May 2013

What you need to do for lab 3

1. **Read fully** the lab 2 assignment: http://www.read.cs.ucla.edu/111/lab3
2. **Read** through ospfs.h . This describes the structure of the file system.
3. Read through ospfsmod.c . This is where all of the exercises are.
4. **Solve** the exercises in ospfsmod.c:
    1. Reading: ospfs_read(), ospfs_dir_readdir()
    2. Blocks and indexing: allocate_block(), free_block(), add_block(), remove_block(), indir2_index(), indir_index(), direct_index()
    3. Writing: change_size(), ospfs_write()
    4. Directory manipulation: create_blank_direntry(), ospfs_create()
    5. Links: ospfs_link(), ospfs_unlink(), ospfs_symlink(), ospfs_follow_link()

Note: in this lab you must use copy_to_user() and copy_from_user() to copy data to/from user space.

Notes on the slides

Slide 2: A diagram of what a UNIX file system looks like on disk. File systems have some amount of metadata for the disk and the file system, and then they have the actual data. The boot sector block is used by BIOS to boot an operating system stored on disk. The superblock contain data about the file system as a whole.

The free block bitmap has one bit for every block, which includes bits for itself, the boot sector block, and the superblock. A **block** is the file system-level unit of space, just as a sector is the unit of space for a disk. This is usually 2^k sectors. In the Linux ext* file systems, the default is 8. In OSPFS, it's 2, so each block is 1024 bytes.

The file system data contains everything needed to hold files.

Slide 3: Just as the file system as a whole has metadata, so does each file have metadata. For UNIX file systems, the file metadata is stored in an **inode**, which references the data blocks that make up the file. In OSPFS, all of the inodes reside in an inode table that is a contiguous number of blocks. Inodes contain file metadata, such as the file's size, type, and the number of hard links, as well as pointers to the actual data blocks.

Direct pointers point directly to data blocks; the indirect pointer points to an "indirect" block, which contains 256 direct pointers; and the indirect^2 pointer points to an "indirect^2" block, which contains 256 indirect pointers, each of which points to an indirect block.

Each inode is 64 bytes, so each block has 16 inodes. We want inodes to be small, since the maximum number of files in the system is limited by the number of inodes.

Slide 4: An example that shows how inodes link to file data blocks. In this example, each inode has only four direct pointers, and the file has six blocks' worth of data. The inode is at some position in the inode table. The four direct pointers point to (contain the block number of) the first four data blocks of the file, which contain the first 4096 bytes of the file's data. The indirect pointer points to a block in the data area that contains direct pointers. That is, it points to a block that contains 256 pointers, each of which can point to a file data block. The first pointer in the pointer block (pointer #0) points to the fifth data block of the file, and the second points to the sixth.

Slide 6: Directories are also files, so they also have inodes. Their pointers are the same as for regular files, but the data is different. Each directory is a list of directory entries, each of which contains a filename and an inode number. When we look up a file in a directory, we find its name and then go to the referenced inode. Each directory entry is 128 bytes, so there are up to 8 directory entries per block.

Question 2: How can you delete a file (directory entry) from a directory without introducing a gap?

Slide 7: Again, a list of metadata for file systems. os_magic is a marker that lets the OS know that this file system is of type OSPFS. Then, we have the total number of blocks in the system, the number of inodes, and the first inode block (so that we know where the inode table begins).

At the file level, we track the file size, type (regular, directory, or symlink), number of hard links, and the file permissions. It's also worth mentioning that there are four types of blocks that can go in the data area: raw data (for regular files), directory entries (for directories), indirect blocks, and indirect^2 blocks. Symlinks don't have data blocks; all of their data is stored in the inode.

Slide 8: A couple of reasons why having links is useful.

Slide 9: The way that hard links work. One advantage to hard links over symlinks is that the lookup can be much faster: once we find the inode in a directory entry, we can jump directly to it. However, if a symlink points to a file nested several directories (e.g. `/a/b/c/d/e/f/g/h.txt`) then to reference the file we must first find directory "a" in the root directory, then find directory "b" within directory "a", and so on.

Question 4: Also in ext4 it's not possible to hard link across file systems—why not?

Slide 10: Symbolic links permit much more freedom than hard links, and in fact lab 3 asks you to make conditional symlinks, which are looked up differently depending on whether the process performing the lookup is running as root or not. A disadvantage of symlinks (other than speed) is that they can be invalid, that is, point to non-existent files.

Slide 11: In UNIX, nearly all OS objects are either files or have file interfaces. Two important filesystems on Linux are /dev and /proc , which abstract hardware devices and kernel objects as files. You can see that they're mounted as file systems by running mount:

```
$ mount
…
proc on /proc type proc (rw,noexec,nosuid,nodev)
…
udev on /dev type tmpfs (rw,mode=0755)
…
```

Slide 12: File systems can also store data indirectly. For example, you can use sshfs to mount a remote directory over SSH. This is useful for editing files on a SEASnet account with emacs running locally; rather than forwarding an X connection, it just sends the file over SSH to your account. For small files, this is much faster than using X.

To mount your home directory on lnxsrv to a subdirectory "seas" of the current directory, do

```
$ sshfs <username>@lnxsrv.seas.ucla.edu: seas
```

To unmount it, do

```
$ fusermount -u seas
```

Slide 13: File systems can also add functionality. For example, there are file systems that perform version control, such as ClearCase. This gives some of the version control functionality file system semantics; for example, files that aren't checked out are read-only. There are also checkpointing file systems such as NILFS that keep track of changes so that accidental writes can be undone. Some other file systems that add functionality are listed here.

Slide 14: Gives the major functions that you will need to write. There is an arrow from one function to another if the first function is intended to be used as a subroutine in the second. For example, the code is structured so that it will be natural to use add_block() when implementing change_size().

Slide 15: An example of how the inode eventually points to each block in a file. The ten direct pointers point to the first ten blocks (blocks 0 through 9) of the file data. The indirect pointer points to a block of 256 direct pointers. The first direct pointer in this block points to file block 10, and the last points to block number 265. The indirect^2 pointer in the inode points to a block of 256 indirect pointers. Each of these indirect pointers points to a block of 256 direct pointers, and each of the 256 direct pointers in such a block point to file data blocks. The first data block pointed to by (the first direct pointer in the block pointed to by (the first indirect pointer in the block pointed to by (the indirect^2 pointer of the inode)) is file data block number 266.

Slide 16: A table for what direct_index (d), indir_index (i), and indir2_index (i^2) should return when we want to access a specific data block. For file blocks 0–9, d = the block number and i = i^2 = -1. For file blocks 10–265, i = 0 and d = the number x such that direct pointer x in the pointer block points to the (x)th file block in the range 10–265. For the rest of the file blocks, i^2 = 0, i is an offset into the indirect^2 pointer block, and d is an offset into the indirect pointer block, so that (i^2,i,d) = (0,0,0) for file block 266 and (0,255,255) for the last possible file block.

Notes on the code—ospfsmod.c

ospfs_read: (*f_pos == 0) means the first byte. The number of characters read on success may be less than the number requested. Remember to use copy_to_user(). Remember to update the position f_pos.

Question 1: What should be done in the first exercise if (*fpos == oi_size)?

ospfs_dir_readdir: In ospfs_read(), a pointer was passed that pointed to our f_pos. Here, we use the file pointer. But we still need to update (filp->f_pos). You don't need to worry about what (dirent) does, just pass it to the filldir() call. Five of the six arguments to filldir() depend on the directory entry (the first is dirent). The fourth, the f_pos value corresponding to the directory entry, will always be two plus a multiple of 128, since each direntry is 128 bytes.

Line 428: The function makes a local copy of f_pos, then updated at the end prior to return.

Lines 432–444: This is where the function "reads" the current and parent directories. Also, it explains the "plus two". You can copy and paste the filldir() call and modify for the other directory contents.

Question 2: Lines 469–470: When does this happen?

indir2_index, indir_index, direct_index: These are just numerical computations.

Question 3: Line 621: What if the "offset of the relevant indirect block" is 0? How do we know whether to use the indirect block or the indirect^2 block?

allocate_block, free_block: These are both pretty straightforward. One thing to keep in mind is that you can (or should) only access the bitmap block by block. So for instance the bitvector operations, that each take a (void*) vector, can't point to any memory regions that cross block boundaries. Another thing to keep in mind is that these two functions are meant to operate only on the bitmap. They're used in add_block() and remove_block() to do the bitmap operations.

add_block, remove_block: The idea with add_block() and remove_block() is that you can grow the file one block at a time until there's at least enough room for the amount in change_size.

Question 4: This seems kind of slow. Why can't you add a lot at a time?

Lines 661–665: First sentence is a point to note for the change_size() function. For the third sentence, if the function succeeds, you want to clear the data blocks that you allocate.

Lines 669–673: This is the main difficulty.

Question 5: Last sentence of 669–673: When might this case occur?

change_size: Existing allocated blocks should not change from their original values, but it's okay to clear the new ones because they were already free; otherwise, they couldn't have been newly allocated. The idea is that change_size should be all-or-nothing with respect to -ENOSPC errors.

Question 6: Lines 759–761: What values do add_block() and remove_block() set it to?

ospfs_write: In ospfs_read(), we updated the count so that we didn't go past the end of the file. The main odd things to handle are the O_APPEND flag and the need to handle growing files.

create_blank_direntry: lines 980–982 just package the error number as a pointer (it's probably just a cast as void*). Lines 988 and 990 are kind of a hint. In ospfs_dir_readdir(), we need to handle gaps; this function can fill them. So directories only grow at the first moment that it has as many files as can fit, and a new one is added. We need to remember to clear out directory entries, since there can be gaps.
ospfs_create: For lines 1056–1057, remember that the convention in the ospfs_direntry struct is to null-terminate the filename. For line 1073, an empty inode is one for which there are no hard links.

Lines 1086–1095: This code should all come after what you write, and you shouldn't need to add anything after it for ospfs_create().

ospfs_link: (src_dentry->d_name.name) is the name of the file inside its (new) directory. (d_name.len) is the length of the filename; (d_name.name) is not null-terminated. (d_inode->i_ino) is the inode number for (**not** a pointer to) the file. Argument "dir" is a pointer to the directory inode.

Question 7: Line 1038: -ENOSPC is returned if the disk is full and the file can't be created. If we're linking a file, since it already exists, when could this case happen? (It **can** happen.)

opsfs_unlink: It's not stated where in this function to add your code. Part of this exercise is to read the code and check your understanding. Make sure you know what is being set to 0 in line 522 and what is being decremented in line 523. Remember that when a file's link count is zero, it's free; where should the data be freed? And what should happen when you unlink a symlink? Does the symlinked file also need to be deleted?

ospfs_symlink: The inputs look much the same as for ospfs_create(); it just has "symname" instead of "mode" and "nd". Also, the skeleton code is identical, except for an extra blank line. So, this function should be pretty similar. One caveat is that the helper function ospfs_inode() returns an ospfs_inode_t* and not an ospfs_inode_symlink_t*; you can just cast it, since they're the same size and have the header in the same place (so you know which is which).

opspfs_follow_link: Note that the ending period in line 1152 is not part of an example string. That is, the example is "`root?/path/1:/path/2`". As stated in the lab 3 description, this should use path/1 if the current process is running as root, and path 2 otherwise. You can use the "current" pointer to access the current process. You also need to be able to handle the case where the symlink is not conditional, and is just of the form "`some/path`". This can be a relative or absolute path.