Lab 3

Due Monday, February 28
Quick link: Getting kernel source / Useful kernel functions

Overview

In Lab 2, we used a RAM disk block device to teach you about the Linux kernel setting and about synchronization. In Lab 3, you'll learn about file systems by writing your own file system driver for Linux.

Operating systems generally support many different file systems, from old-school FAT (common in DOS machines) to new formats such as Linux's ext3. Each file system type is written to fit a common interface, called the *VFS layer*. (File systems are often implemented using object oriented techniques.) You will see how a real VFS layer works by filling out a simple file system implementation. You will load that file system module into the kernel and *mount* the file system, allowing you to access your file system with usual Unix commands like `cat`, `dd`, `emacs`, and even `firefox`. The file system driver stores its data in a built-in RAM disk, kind of like your RAM disk from Lab 2.

Like Lab 2, we expect you'll do this lab on Qemu using the `./run-qemu` script on your machine, or in the Linux lab. Alternatively, you can also use a Linux machine, such as the Linux lab machines.

Lab materials

The skeleton code you will get for this lab consists of the following files:

`ospfs.h`	Commented header file defining file system structures and constants.
`ospfsmod.c`	The file system module's source code. **All exercises in the lab involve changing this code.**
`answers.txt`	Fill this out before submitting.
`base/`	The contents of this directory are copied into your file system's initial RAM disk. You can change the contents of `base/`, then type `make clean; make`, to add files to or remove files from your RAM disk.
`./run-qemu`	This script starts up Qemu to run Lab 3.
`./run-direct`	This script runs Lab 3, and is intended for use on a Linux machine.
`test/`	The `./run-qemu` script changes `test/` into a view of your `ospfs` file system.
`Makefile`	
`ospfsformat.c`	Utility: Creates a RAM disk image from the `base/` directory.
`md5.c`, `md5.h`	Utility for `ospfsformat`.
`fsimgtoc.c`	Utility: Links the disk image into your module.
`truncate.c`	Utility: Truncates a file to a specified length.

When you unpack `lab3.tar.gz` (with the command `tar xvzf lab3.tar.gz`) you will get a directory `lab3` which contains these files. Do your work in this directory.

Handin procedure

When you are finished, edit the `answers.txt` file and follow the instructions to fill in your name(s), student ID(s), email address(es), short descriptions of any challenge problems you did, and any other information you'd like us to have. Then run "`make tarball`" which will generate a file `lab3-`*`yourusername`*`.tar.gz` inside the lab3 directory. Upload this file to CourseWeb using a web browser to turn in the project. (Please do not upload `.zip` files!)

Background information

File system preliminaries

The file system you will work with, OSPFS, is simpler than most "real" file systems, but it is powerful enough to read and write files organized in a hierarchical directory structure. OSPFS currently does not support timestamps, permissions, or special device files like most UNIX file systems do. Additionally, your module is only capable of using an in-memory file system, meaning that the data "on disk" is actually stored entirely in main memory. This means that changes made to the file system are not saved across reboots (or across module loads and unloads).

File system structure

Most UNIX file systems divide available disk space into two main types of regions: inode regions and data regions. UNIX file systems assign one inode to each file in the file system; a file's inode holds critical metadata about the file such as its attributes and pointers to its data blocks. The data regions are divided into large data blocks (typically 4KB or more), within which the file system stores file data and directory metadata. Directory entries contain file names and pointers to inodes; a file is said to be *hard-linked* if multiple directory entries in the file system refer to that file's inode. OSPFS is designed in much the same way.

Both files and directories logically consist of a series of data blocks, which may be scattered throughout the disk much like the pages of a process's virtual address space can be scattered throughout physical memory.

Sectors and blocks

Modern disks perform reads and writes in units of sectors, which today are almost universally 512 bytes each. File systems actually allocate and use disk storage in units of blocks. Be wary of the distinction between the two terms: sector size is a property of the disk hardware, whereas block size is an aspect of the operating system using the disk. A file system's block size must be at least the sector size of the underlying disk, but could be greater (often it is a power of two multiple of the sector size).

The original UNIX file system used a block size of 512 bytes, the same as the sector size of the underlying disk. Most modern file systems use a larger block size, however, because storage space has gotten much cheaper and it is more efficient to manage storage at larger granularities. Our file system will use a block size of 1024 bytes.

Superblocks

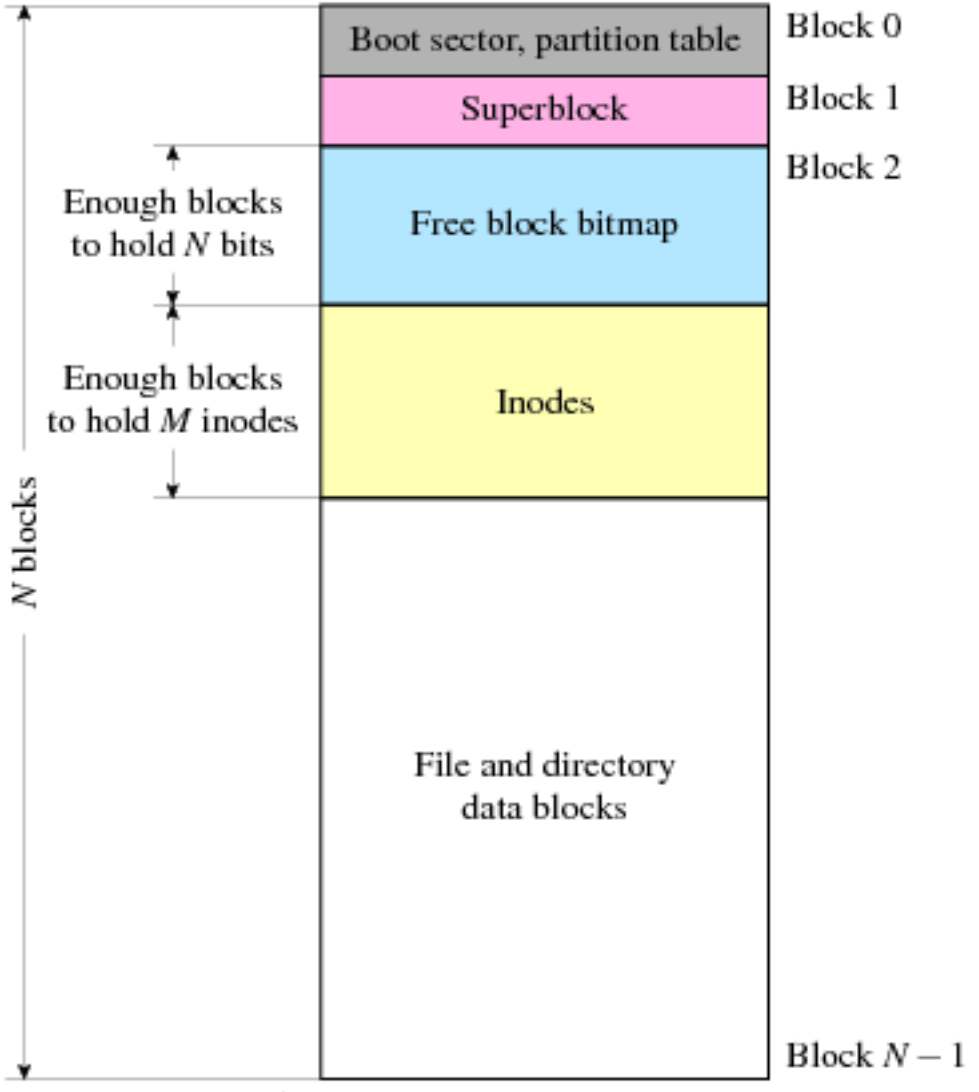
File systems typically reserve certain disk blocks, at "easy-to-find" locations on the disk (such as the very start or the very end), to hold meta-data describing properties of the file system as a whole: the block size, disk size, any meta-data required to find the root directory, the time the file system was last mounted, the time the file system was last checked for errors, and so on. These special blocks are called superblocks.

Our file system will have exactly one superblock, which will always be at block 1 on the disk. Its layout is defined by `struct ospfs_super` in `ospfs.h`. Block 0 is typically reserved to hold boot loaders and partition tables, so file systems generally never use the very first disk block. Most "real" file systems maintain multiple superblocks, replicated throughout several widely-spaced regions of the disk, so that if one of them is corrupted or the disk develops a media error in that region, the other superblocks can still be found and

used to access the file system. OSPFS omits this feature for simplicity.

The block bitmap: managing free disk blocks

Just as a kernel must manage the system's physical memory to ensure that each physical page is used for only one purpose at a time, a file system must manage the blocks of storage on a disk to ensure that each block is used for only one purpose at a time. In file systems it is common to keep track of free disk blocks using a bitmap rather than a linked list, because a bitmap is more storage-efficient than a linked list and easier to keep consistent on the disk. ("FAT" file systems use a linked list.) Searching for a free block in a bitmap can take more CPU time than simply removing the first element of a linked list, but for file systems this isn't a problem because the I/O cost of actually accessing the free block after we find it dominates performance.



To set up a free block bitmap, we reserve a contiguous region of space on the disk large enough to hold one bit for each disk block. For example, since our file system uses 1024-byte blocks, each bitmap block contains 1024*8 = 8192 bits, or enough bits to describe 8192 disk blocks. In other words, for every 8192 disk blocks the file system uses, we must reserve one disk block for the block bitmap. A given bit in the bitmap is set (value 1) if the corresponding block is free, and clear (value 0) if the corresponding block is in use. The block bitmap in our file system always starts at disk block 2, immediately after the superblock. We'll reserve enough bitmap blocks to hold one bit for each block in the entire disk, including the blocks containing the boot sector, the superblock, and the bitmap itself. We will simply make sure that the bitmap bits corresponding to these special, "reserved" areas of the disk are always clear (marked in-use).

Inodes

Each file and directory on the system corresponds to exactly one *inode*. The inode stores metadata information about the file, such as its size, its type (file or directory), and the locations of its data blocks. Directory information, such as the file's name and directory location, is *not* stored in the inode; instead, directory entries refer to inodes by number. This allows the file system to safely move files from one directory to another, and also allows for "hard links". Not all file systems have inodes, but OSPFS does.

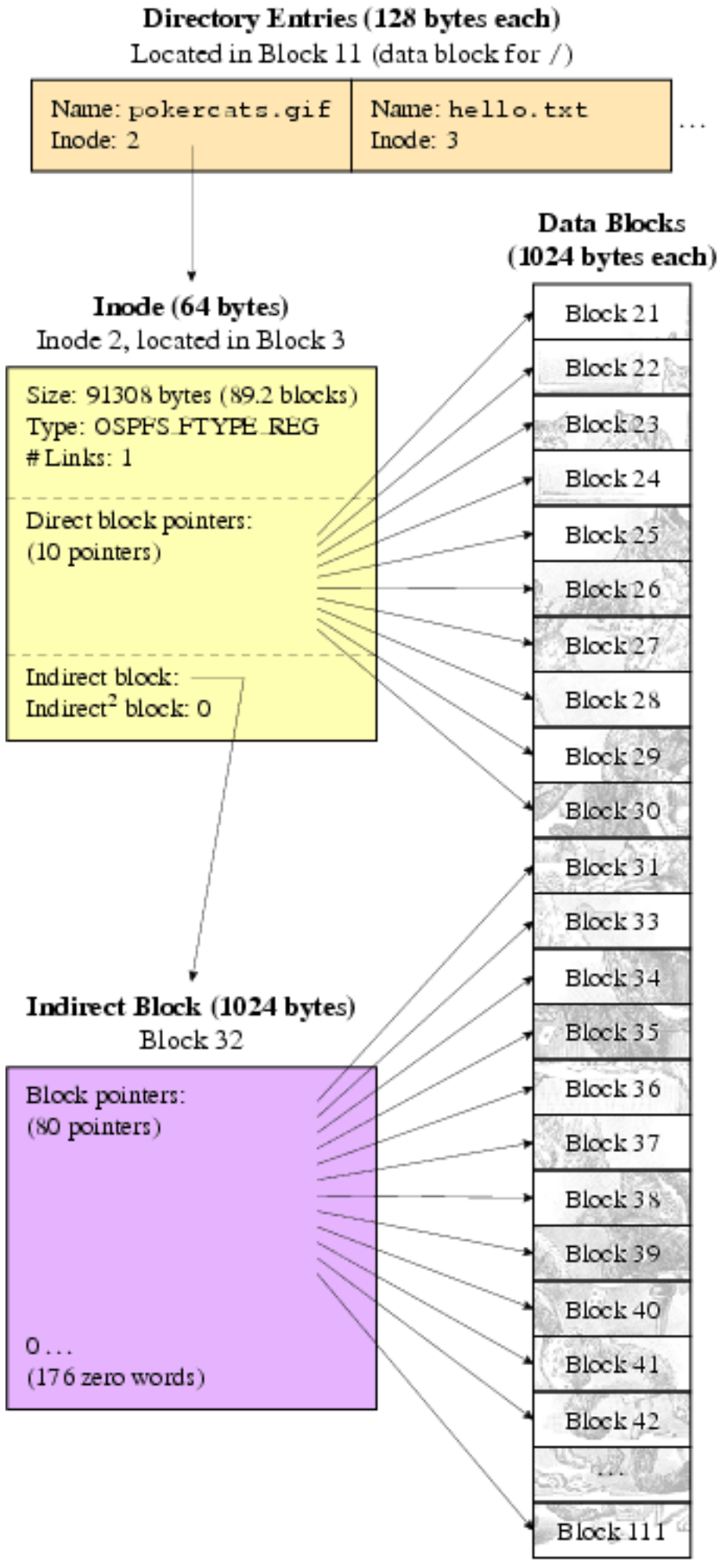
The layout of OSPFS's inodes is described by `struct ospfs_inode` in `ospfs.h`. The `oi_direct` array in `struct ospfs_inode` contains space to store the block numbers of the first 10 (`OSPFS_NDIRECT`) blocks of the file, which we call the file's direct blocks. For small files up to 10*1024 = 10KB in size, this means that the block numbers of all of the file's blocks will fit directly within the `ospfs_inode` structure itself. For larger files, however, we need a place to hold the rest of the file's block numbers. For any file greater than 10KB in size, therefore, we allocate an additional disk block, called the file's indirect block, to hold up to 1024/4 = 256 additional block numbers. The 10th block number is the 0th slot in the indirect block. This allows files to be up to 266 blocks, or a bit more than 256KB, in size. For files larger than this, the OSPFS file system supports doubly-indirect blocks as well. A doubly-indirect block points to indirect blocks, each of which points to 256 direct blocks. This allows files to be up to 10+256+256*256 = 65802 blocks. To support larger files, "real" file systems may support triply-indirect blocks as well. See the picture at right.

Inodes are relatively small structures; OSPFS's take up 64 bytes each. They are generally stored in a contiguous table on disk. The size of this table is specified when the file system is created, and in most file systems it

cannot later be changed. Just like blocks, an inode can be either in use or free, and just like blocks, a file system can run out of inodes. It is actually possible to run out of inodes before running out of blocks, and in this case no more files can be created even though there is space available to store them. For this reason, and because inodes themselves do not take up much space, the inode table is often created with far more available inodes than the file system is expected to need. (Remember, each file uses an inode, so the maximum number of files and directories the file system can store at once equals the number of inodes.)

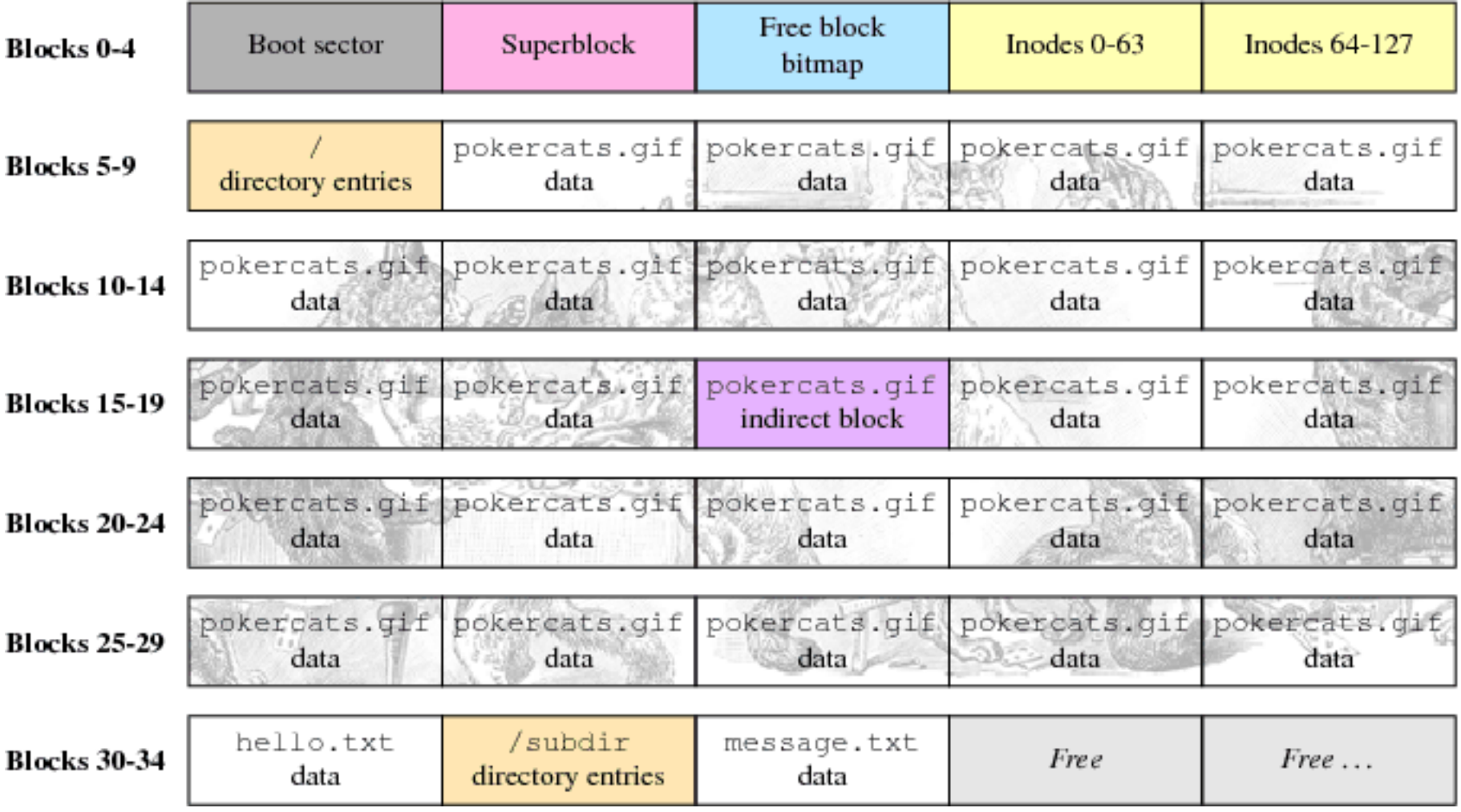
In OSPFS, the inode table is allocated immediately after the free block bitmap, and its blocks are also marked as in use in the free block bitmap. The number of inodes is stored in the superblock, and for convenience the block number of the first block of the inode table is also stored in the superblock. (Why is this only for convenience? How could we determine this value based only on the size of the disk?)

Even though your kernel module will not support creating new hard links (unless you do that challenge problem), each inode in OSPFS has a "link count", which is a number indicating how many directory entries refer to it. (The `ospfsformat` utility knows how to create hard links when it creates the initial OSPFS file system for your module to use.) If there is only one hard link to a file, then the link count will be 1: only one directory entry refers to the inode (the normal case). For each extra link created in a "real" UNIX file system, the link count increases by 1, and for each hard link removed, the link count decreases by 1. Technically speaking, even the file's original name is a hard link -- it is no different than later directory entries created for the inode except that it was created first. If the link count reaches 0, it means that the inode no longer has any hard links, and its storage space (both the blocks that make up the file and the inode itself) can be marked as free and reused. You will need to support this behavior when you delete files from OSPFS.



You may find the `-i` and `-l` options to the `ls` program useful. The `-i` option causes `ls` to display the inode number of each file. The `-l` option causes `ls` to display a "long listing" which includes, among other information, the link count and size of each file. These options can also be combined.

Here's a sample OSPFS file system. It **sort of** corresponds to the `base/` directory we handed out. (Note that the `hello.txt` data block is also linked to `world.txt` -- there is a file with two hard links!) The actual file system you observe will be different; your first free block will be around block #106.



Directories versus regular files

An `ospfs_inode` structure in our file system can represent either a regular file or a directory; these two types of "files" are distinguished by the type field in the `ospfs_inode` structure. The file system manages regular files and directory-files in exactly the same way, except that it does not interpret the contents of the data blocks associated with regular files at all, whereas the file system interprets the contents of a directory-file as a series of `ospfs_direntry` structures describing the files and subdirectories within the directory. Each `ospfs_direntry` structure just contains a filename and an inode number. All other information about the file -- including size, type, and pointers to data blocks -- is stored in the inode.

OSPFS directory entry structures are *fixed length*. Every directory entry takes up exactly 128 bytes. The directory entry design is explained in more detail in `ospfs.h`.

Inode number 1 in our file system is special: it is the inode for the *root directory* of the file system. Inode number 0 is reserved and must never be used.

Hard Links

Files can take up a lot of space, and you may want the same exact set of data to be stored in multiple directories. Hard links are a simple mechanism that lets you have multiple "`files`" link to the same `inode` structure. Suppose the user runs the following commands:

```
% echo foo >> foo.txt
% ln foo.txt gah.txt
% cat gah.txt
foo
% echo blurb >> gah.txt
% cat foo.txt
foo
blurb
```

In this code, the user is hard linking `gah.txt` to the same data that represents `foo.txt`. So whenever

either `gah.txt` or `foo.txt` is changed, both files see those changes, automatically. This may sound challenging, but it's actually really simple. The directory entries for `gah.txt` and `foo.txt` map to the same inode number. For this, you'll have to just fill out `osprd_link`, which gets called whenever the user tries to create a hard link in a directory. For simplicity, you do not need worry about or test hard links to symlinks. Only worry about hard links to regular files.

Symbolic links

OSPFS also supports *symbolic links*. This type of file adds a layer of indirection to the file system: A symbolic link file essentially points at another file. The command "`ln -s src.txt dst.txt`", will create a symbolic link file `dst.txt` that points to file `src.txt`. If an application asks to open `dst.txt`, the kernel will examine the symbolic link, *follow* the symbolic link, and return a file descriptor open to `src.txt`. For example, any changes to `src.txt` will show up immediately when a program examines `dst.txt`. In OSPFS, symbolic links' destination files are stored in their inodes themselves. See `ospfs.h` for more detail.

You are expected to add support for *symbolic link creation and deletion* to `ospfsmod.c`. Test it using code like this:

```
% ln -s hello.txt thelink
% diff hello.txt thelink && echo Same contents
Same contents
% echo "World" >> hello.txt
% diff hello.txt thelink && echo Same contents
Same contents
% rm thelink
```

Conditional Symlinks

One nice thing about symlinks is that the actual symlink file can contain anything. This allows the file system to interpret a symlink in unique ways. For instance, suppose you have a networked file system (like the one on the Seasnet machines) that connects to multiple different architectures. In order to use different versions of a binary, you currently need to set different values for the 'PATH' environment variable. It would be much nicer if the a symlink could link to two different paths, and select the right one based on some condition (in this case the cpu architecture). Doing this all the way is rather challenging, so we'll just start with a simple proof of concept; our conditional symlinks will be based on whether or not the current process is running as root.

Here's how they will work:

- The following command creates a conditional symlink: "`ln -s root?hello.txt:bye.txt test`". There are four important parts of this conditional symlink.
  - The conditional statement. For this lab, it will always 'root?' which is true if the current process is running as root and false in all other cases.
  - The symlink for when the condition is true. In this case it is 'hello.txt'. Whenever root accesses this symlink, it should be redirected to 'hello.txt' automatically.
  - The symlink for when the condition is false. In this case it is 'bye.txt'. Whenever any non-root process accesses this symlink, it should be automatically redirected to 'bye.txt'.
  - The symlink's name. In this case it is 'test'.

Handling this properly will require several things: determining the user id of the current running process, knowing when the running process is root, knowing when a conditional symlink is being accessed, and how to interpret a conditional symlink upon access. Some functions useful for this have been added to the Lab functions page.

Here's an example of how this should work:

```
# cd test
# echo "Not root" > notroot
# echo "Root" > root
# ln -s root?root:notroot amiroot
# cat amiroot
Root
# su user -c "cat amiroot"                          # run command as non-root user
Not root
```

What you have to do

Your assignment will be to implement the routines that handle free block management, changing file sizes, read/write to the file, reading a directory file, deleting symbolic links, creating files, and conditional symlinks.

You will need to implement exercises in the following functions for this lab. One plausible order (with **rough** grade percentages) is this:

- Support for reading files: `ospfs_read` (15%)
- Support for reading directories: `ospfs_dir_readdir` (10%)
- Free block bitmap bookkeeping: `allocate_block`, `free_block` (15%)
- Support for changing file sizes: `change_size` (15%)
- Support for writing files: `ospfs_write` (15%)
- Support for creating files: `ospfs_create` (10%)
- Support for creating hard links: `ospfs_link` (5%)
- Support for creating and deleting symbolic links: `ospfs_symlink` and `ospfs_unlink` (10%)
- Support for conditional symlinks (5%).

All of these functions have comments explaining what you must do. Also read the comments in `ospfs.h` and towards the top of `ospfsmod.c` to get oriented.

Building the lab

This lab is designed to run within the Qemu emulator, like Lab 2. So just run `./run-qemu` and press Enter at the `. go` prompt.

We have added new support for re-building your lab in the Qemu emulator (so you don't have to reboot Qemu to fix compilation errors and such). To reload your source, first type "`r`" in your shell window (where you ran `./run-qemu`), then type "`reload`" in the Qemu window. Do note that this process *wipes* your `test` directory, replacing it with a fresh version.

This builds an initial OSPFS image using the contents of the `base` directory. If you want to add files to your OSPFS image, simply add files and directories to `base`, then rerun Qemu. If `make` reports an error that the resulting file system is too big, raise the number `512` in the Makefile to something larger. By default, the image created by `ospfsformat` contains two directory entries which refer to the same file (that is, they are hard links): `hello.txt` and `world.txt`. You can add more files like this by adding hard links to the `base` directory using the `ln` command. Read its manual page by typing `man ln` at a shell prompt. (Note: the link counts on directories are always larger than one. Why might this be?) The Makefile also creates a symbolic link from file `link` to `hello.txt`.

The Qemu environment then *mounts* the file system on an existing directory, named `test`. This hides the directory's existing contents, turning the directory into a "gateway" into the new file system. When your file system is ready, `ls base` and `ls test` should return the same results (except for things like ownership,

permissions, and inode numbers).

It is worth discussing how the functionality of the script is implemented. The following list shows the commands used:

- Installing a module in the kernel: `insmod ospfs.ko`
- Checking to see if the module is installed: `cat /proc/modules` or `lsmod`
- Mounting the filesystem onto the `test` directory: `/bin/mount -t ospfs none test`
- Unmounting the filesystem from `test`: `umount test`
- Removing a module from the kernel: `rmmod ospfs`

If want to run Lab 3 code directly on some Linux machine, the script `./run-direct` will be of use. This script loads the module into the currently running kernel.

Testing

After implementing these functions, you should be able to do the following in your filesystem: read and write to files, change the size of files, append to existing files, read the contents of directories, and create and remove files.

Here are some hints for testing.

- We provide a large GIF file to test your handling of large files (with indirect blocks), `pokercats.gif`. To see whether your reading support works for large files, run `zgv test/pokercats.gif` at the Qemu prompt. You should see a picture. (Press return to get rid of the picture.) To test writing support for large files, try something like `cp test/pokercats.gif test/p.gif; zgv test/p.gif`.
- The `lab3-tester.pl` file, provided with the lab, works similarly to the testers from other labs. **You should as usual design some tests yourself.**

We have provided you with a minimal Lab 3 test script, `lab3-tester.pl`, to help you check a *few* of the following test cases. You are encouraged to add your own test cases to it as you complete parts of your lab. If you have questions on how to add test cases, feel free to email the TAs.

Here is a list of the test cases we will run on your code. Feel free to add these to the provided testing script so that it checks these things automatically for you! You will probably find the `dd` utility very useful to perform many of the tests. You can read its documentation (`man dd`) for details of how to use it: check out its `if=`, `of=`, `bs=`, `seek=`, `skip=`, and `conv=notrunc` options. Also the `truncate` utility we included with the skeleton code will help you test some of the cases. (Suggestion: compare the results of performing operations on your OSPFS file system to doing the same operations on the normal Linux file system.)

- **Directory reading tests:** listing files in directories
  - ls of the root directory
  - ls of the root directory including file types (ls -F)
  - ls of subdirectory
  - ls of subdirectory using more than one data block
  - ls of subdirectory using indirect data blocks
- **Reading tests:** reading various parts of a file
  - read the first byte of a file
  - read the first block of a file
  - read half of the first block of a file

  - read starting partway through the first block and into part of the next
  - read more than one block
  - try to read past the end of a file
  - try to read into an invalid buffer pointer
- **Overwrite tests:** all these tests overwrite part of a file without changing the rest of the file
  - overwrite the first few bytes of a file
  - overwrite the first two blocks of a file
  - overwrite the second block of a file
  - overwrite the middle part of the first block of a file
  - overwrite the second half of the first block of a file
  - overwrite the second half of the first block and the first half of the second block of a file
  - try to write from an invalid buffer pointer
- **Truncation tests:** truncating files to various lengths
  - truncate a single data block file to 0 length
  - truncate a file with only direct data blocks to 0 length
  - truncate a file with indirect data blocks to 0 length
  - truncate a single data block file to nonzero but smaller length
  - truncate a file with only direct data blocks to a smaller length that uses the same number of data blocks
  - truncate a file with only direct data blocks to a smaller length which uses fewer data blocks
  - truncate a file with indirect data blocks to a smaller length that still uses indirect data blocks
  - truncate a file with indirect data blocks to a smaller length which only uses direct data blocks
- **Appending tests:** adding data to the end of a file (not necessarily using O_APPEND)
  - append data to a file with a single data block without requiring a new block to be allocated
  - append more than one data block of data to a file
  - append several data blocks to a file that already has indirect data blocks
  - append data to a file with no indirect data blocks so that the appended data uses indirect data blocks
  - try to append data to a file so that there would be more than the maximum allowed data blocks
- **Block management tests:** checking to make sure the block bitmap is managed correctly
  - try to allocate a block when there are no free blocks
  - free a block and check that that block can be reallocated
  - try to allocate 3 blocks when only 2 are free
  - truncate a file so that it no longer has any indirect data blocks and ensure that the indirect pointer block is freed
- **Deletion tests:** checking that deleting files works correctly
  - delete a file from a directory
  - delete a file with more than one hard link from a directory
  - delete all hard links to an inode
- **Symbolic links:** checking that symbolic links work correctly
  - create a symbolic link
  - **DELETE A SYMBOLIC LINK** (you will need to check that this works!)

Design problems

Design problem guidelines

OSPFS Crash Testing

As we've discussed in class, file systems are expected to be **robust**: no matter when the computer crashes, a file system should leave stable storage correct (i.e., satisfying all four file system invariants), or at least *sufficiently* correct that no file system data is lost. Your OSPFS implementation is probably not robust!

This design problem asks you to design a mechanism to **test** file system robustness by **"crashing"** the OSPFS file system.

- Test: Introduce a per-OSPFS variable called `nwrites_to_crash`. OSPFS users can set this variable by making a system call on an open OSPFS file (an `ioctl`). When this variable is -1 (the default), OSPFS should act as usual. If the variable is 0, then the file system has "crashed": **every write to disk data should silently fail**. That is, any time that your OSPFS code writes to disk, whether in a data block, a superblock, an inode block, or whatever, the write should be ignored. If the variable is GREATER than 0, then the variable should be decremented by 1 for every write to a different block. Thus, after `nwrites_to_crash` writes, the OSPFS will "crash".
- Find bugs: Design a test program that demonstrates a bug with your OSPFS implementation. That is, your program should set `nwrites_to_crash` and then make a series of system calls -- writes, creates, links, unlinks, whatever -- so that after the "crash", the file system is left in an incorrect state. This will require that you understand incorrect states and figure out how to cause one. Demonstrate the problem by "uncrashing" the file system (setting `nwrites_to_crash` to -1), performing some more file system operations, and showing that the result is disaster (missing files, etc.).

OSPFS File System Fixer

Design a user-level program that analyzes an OSPFS file system image file, such as `fs.img`, and detects and fixes problems with that file. You should detect violations of the four invariants (1. All blocks used for exactly one purpose, 2. All blocks initialized before referenced, 3. All referenced blocks marked not free, 4. All unreferenced blocks marked free), as well as sanity-check disk structures like the superblock, inodes, and directory entries. Some file system problems (e.g. massive superblock corruption) will be unfixable; others easy to fix. Describe which problems fit into which category and why. Also design a way to test your file system image fixer, such as perhaps a program that generates bad file system images.

OSPFS Journaling

(This is probably the most challenging and interesting design problem for this lab.)

A *file system journal* is a special area of the disk used to increase robustness. Every change to the file system is first written to the journal area. When the journal entry representing a change is complete, the file system writes a "commit record" to the journal. Once this commit record is written, the change will definitely happen. Then, **after** writing the commit record, the file system implementation goes ahead and writes the change for real. Finally, it marks the journal transaction as having completed.

For example, say that OSPFS needed to extend a one-block file to two blocks long. This requires the following steps: 1. Allocate a block (pick a free block and mark it as used in the free block bitmap); 2. Write the block pointer into the inode; 3. Write the size into the inode. In your journal implementation, this will take several more steps:

1. Pick a free block. Say it's number *B*.
2. Write "Begin Transaction" into journal.

3. Write "allocate block *B*" into the journal. There are several ways to do this; you could write a small record that just says "allocate *B*", or you could write an entire copy of the relevant free-block-bitmap block after the change.
4. Write "set block *B* as direct block #1 of inode *I*, and set inode *I* size to *S*" into the journal. As above, there are several ways to do this.
5. Write "Commit Transaction" into journal.
6. Actually write the free-block-bitmap block and inode block.
7. Replace the relevant "Commit Transaction" element of the journal with "Completed Transaction".

Why do this? For crash safety: robustness! If the system crashes during step 6 (actually writing the file system), the file system can recover by *replaying* the journal. On reboot, the file system implementation will read the journal, and **re-perform** each action recorded in each of the "Committed Transactions". (It is not necessary to re-perform "Completed Transactions".) This will bring the file system back to the proper state. On the other hand, if the system crashes during steps 1-5, the journal record will not be "Committed". The file system can ignore that journal transaction as if it had never happened. Since no changes were made to the main part of the file system (that doesn't happen until Step 6), the main part of the file system is by definition OK!

You can read more about atomicity journals in your text, pages 9-47 - 9-60.

For this challenge problem, design a format for an OSPFS journal. Change the file system format to include space for a journal. Change the file system implementation to write change records to the journal before writing to the main file system. And design the program that replays the journal on reboot.

OSPFS Performance

The OSPFS file system has one free block bitmap area and one fixed-size inode area. This is easy, but not much like real file systems. Single free-block-bitmap and inode areas make the file system slow, because for many operations (such as allocating a block), the disk is forced to seek back and forth across large distances to (1) write the free-block-bitmap block and then (2) write the allocated block itself. Many real file systems therefore divide the disk into multiple **regions**. Each region has its own free block bitmap and its own inode area. This reduces the distance that the disk must seek to allocate a block: the disk head can stay within a single region. Additionally, most file systems don't artificially limit the number of files a file system can handle. OSPFS as currently specified can run out of inodes before the file system fills up.

- Redesign the OSPFS file system format so that an OSPFS disk is divided into regions. Change the superblock to record how big each region is, where each region begins, and/or anything else that is necessary.
- Redesign the `ospfsformat.c` code to generate a correctly-formatted "regionized" disk. (You may ask the professor's help with `ospfsformat.c`, since it is not very well commented yet.)
- Redesign the OSPFS file system code so that the file system attempts to keep all of a file's blocks within a single region. This means the disk will seek shorter distances when reading the file. Of course, as the disk gets full, it may be necessary to use blocks from another region.
- Redesign the OSPFS file system format and module code so that an OSPFS will never run out of inodes artifically. That is, the OSPFS file system won't run out of inodes unless the disk is full.

Good luck!