Project 5b: The Small File Optimization

Notes

Start with this kernel, not the usual one: **~cs537-1/ta/xv6/xv6.fspatch.tar.gz**

5/2: The semantics of a write() should follow those of typical Unix systems. That is, if you try to write 100 bytes to a small file, the first 52 bytes should be written to the file, and the return value should indicate that 52 bytes are written. Similarly, if the file is already 10 bytes in size, and then someone issues a 100-byte write, 42 bytes should be written and 42 returned. Writes after the file is full (unless truncated) should fail and return an error (-1).

Overview

In this project, you'll be changing the existing xv6 file system to add high-performance support for small files. This type of optimization has been explored in the literature (see here for example) and now you can explore it, in some limited form, in this project.

The basic idea is simple: if you have a small file (say, just a few bytes in size), instead of allocating a data block for it, instead just store the data itself inside the inode, thus speeding up access to the small file (as well as saving some disk space). Because the inode has some number of slots for block addresses (specifically, NDIRECT for direct pointers, and 1 more for an indirect pointer), you should be able to store (NDIRECT + 1) * 4 bytes in the inode for these small files (52 bytes or less).

To make life a little easier for you, we will create a new file type that is specifically called a T_SMALLFILE. This will let your code recognize that the file being accessed is indeed a small file and act differently than the normal case. There will also be a new flag used to create these small files, O_SMALLFILE. More details below.

Thus, you'll have to be able to handle a new flag to open() (O_SMALLFILE), which, when passed to open(), should make sure to create the file as not a T_FILE (normal file) but rather as a T_SMALLFILE. Then, you'll have to modify the read and write paths to be able to read and write from these small files. You'll also have to check for errors; for example, you'll have to return errors when the user tries to make the small file bigger than the (NDIRECT+1)*4 bytes.

Details

To start, look in **include/stat.h.** You'll have to add the T_SMALLFILE in there, and define it to be 4, e.g.

```
#define T_SMALLFILE 4
```

This will let us determine if the file being accessed is indeed one of these small files.

You also need to poke around to find where a new flag to open should be defined. You'll find that **include/fcntl.h** has these definitions. You'll need to add:

```
#define O_SMALLFILE 0x400
```

in there. Note: these must be followed exactly, or tests will not work.

Then you need to go about following the read/write paths to see where to make your changes. Start in **kernel/sysfile.c** (looking at sys_open() and sys_write(), as well as create()), and follow through the read and write paths to **kernel/file.c** and eventually **kernel/fs.c.** In that last file, make sure to understand readi() and writei(), as well as the inode update routine iupdate().

There is no need to do any of this for directories; they should remain as is.

Note that a real file system likely wouldn't create a new file type (T_SMALLFILE) for this type of optimization; rather, it would store small files' data in the inode, and then, when the file grew beyond what fits in the inode, it would allocate data blocks and put all the data in there. In this project, small files always just use the inode to store data, and are not allowed to grow beyond what fits in there. This simplification is in place to make your life easier.

The Code

The code (and associated README) can be found in **~cs537-1/ta/xv6/** . Everything you need to build and run and even debug the kernel is in there, as before.