OS222: Assignment 4

File Systems

June 11, 2022

Responsible TAs: Yuval Gitlitz

1 Introduction

In this assignment you are required to extend the xv6 file system. Xv6 implements a Unix-Like inode based file system, and when running on top of QEMU, stores its data on a virtual IDE disk (fs.img) for persistence. To get familiar with xv6's file system design and capabilities it is recommended to read Chapter 8 of the xv6 book, available in the course's website.

The assignment is composed of two main parts:

1. Increasing Xv6 maximum file size.
2. Adding support for symbolic links.

Use the following git repository:
`https://github.com/BGU-CS-OS/xv6-riscv`

Before writing any code, make sure you read the **whole** assignment.

2 Expanding the maximum file size

Xv6, as a variant of Unix, uses the same file system architecture based on i-nodes. In i-node based file systems, the maximum file size is determined by the i-node structure. In xv6, an inode contains 12 direct links to data blocks and another single indirect link. Each data block is 1024 bytes, totaling in 12KB for the 12 direct links. The indirect link points at a block containing $1024/4 = 256$ additional links to the actual data. This single level of indirection therefore gives access to 256KB of additional data. Thus, overall, the maximum file size in xv6 is 268KB.

In this part of the assignment, you will change the i-node structure to support files of size up to $\approx$ 64MB by adding a double indirection link to the i-node structure. Note that in xv6, an i-node is referred to as a **dinode** (disk i-node) and an (in-memory) cached i-node is referred to as an **inode**.

- The file **mkfs/mkfs.c** is written using standard C libraries and is built and executed by the makefile outside of xv6 to create the virtual drive, fs.img. One of its tasks is to write the superblock, which contains metadata characterizing the file system. You will have to modify the content of the superblock to be consistent with the changes you make. The virtual disk, fs.img, should contain at least 2^{17} blocks (totaling 128MB)
- The size of a **dinode** structure should be a divisor of the block size. In other words, an integer number of **dinodes** should fit in a block. You may want to add padding for this purpose.

2.1 Sanity Test

Write a simple user application that creates a text file of size 10MB and writes notification messages to the screen:

1. After writing all the single 12 direct blocks.
2. After writing all the single indirect blocks.
3. After writing the rest of the file in the double indirect block.

The output should look something like:

```
Finished writing 12KB (direct)
Finished writing 268KB (single indirect)
Finished writing 10MB
```

3 Adding support for symbolic links

Xv6 supports hard links via the user space program *ln*. Hard links allow different pathnames to reference the same actual file by using the same i-node number. For example, when a hard link named “b.txt” is created for a file named “a.txt”, both “a.txt” and “b.txt” refer to the same file (data) on disk. Changes made to “a.txt” are reflected in “b.txt” and vice versa. Deleting one will not affect the other (it will only decrease the link count).

In this task you will extend the program *ln* to support symbolic links (also referred to as soft links). When a new symbolic link is created, it does not share the same i-node as the pointed file (target). Instead, a new file is created, and a new i-node is assigned to it. **The content of the new file will contain a path to the target**.

Notice that symbolic links are a special type of files and should be assigned a unique enumeration value in the file type enum (currently supported types are: T_FILE, T_DIR and T_DEV). Also, symbolic links can point to any type of file: a regular file, a directory and even other symbolic links. Furthermore, they can be either absolute or relative. Naturally, moving a relative link to a different location may result in a broken link.

The following syntax should be used to create a symbolic link:

```
ln -s old_path new_path
```

Example:

```
ln -s /cat /new_cat
```

This example will create a new symbolic link names *new_cat* linking to the program cat. Running *new_cat* will then run cat itself.

The following system calls should be implemented to support symbolic links:

```c
int symlink (const char *oldpath, const char * newpath)
```

- **oldpath** The path the new file contains. This will be written as the content of the file. Note that it is OK to create a symbolic link to a file that doesn’t exist.
- **newpath** The new symbolic link file path. If successful, a new symbolic link file will be created in this path.

The function returns 0 upon success and -1 on failure. Note that if newpath exists the function will fail.

```c
int readlink (const char * pathname, char * buf, int bufsize)
```

- **pathname** A path to a symbolic link file.
- **buf** A pointer to a buffer into which the content of pathname symbolic link will be read.
- **bufsize** the size of the buf buffer.

The function returns 0 upon success and -1 on failure. The function will fail if either: 1) pathname does not exists, 2) it is not a symbolic link, or 3) bufsize is smaller than the length of the resolved path.

3.1 Extending sym-link support

Extend symbolic link support so that user application can receive them as their operands. For example, applying the cat command to a symbolic link should display the contents of the target file to which the symbolic link points.

To do so, you will need to change the system such that system calls such as *open*, *chdir*, *exec*, *fstat*, *unlink* work well with symbolic links:

- **fstat** should return with a special type for symbolic link.
- **chdir / exec / open** should dereference symbolic links.
- **unlink** should remove the symbolic link and not the referenced file.

Note that to avoid infinite dereferencing loops (A is a symbolic link to B and B is a symbolic link to A) you will need to add a constant *(MAX_DEREFERENCE)* and set it to 31. Each call to *namex* should not allow more then *MAX_DEREFERENCE* symbolic links. This is how Linux avoids dereferencing infinite loops.

In addition to updating the implementation of relevant system calls, you are required to update the usermode application *ls*. On a symlink, it must print the file name and the path, similarly to the output of "ls -l linkname" in your linux machine. For example if *lnk* is a link to *./cat*, the output of ls in the root directory should look similar to:

```
.              1 1 1024
..             1 1 1024
README         2 2 2654
cat            2 3 24104
echo           2 4 22928
forktest       2 5 13168
grep           2 6 27408
init           2 7 23664
kill           2 8 22888
ln             2 9 22728
ls             2 10 26296
mkdir          2 11 23016
rm             2 12 23008
sh             2 13 41840
stressfs       2 14 23880
usertests      2 15 156904
grind          2 16 38032
wc             2 17 25208
zombie         2 18 22264
console        3 19 0
lnk->./cat     2 20 0
```

4 Submission Guidelines

Make sure that your Makefile is properly updated and that your code compiles with no warnings whatsoever. We strongly recommend documenting your code changes with comments - these are often handy when discussing your code with the graders. Due to our constrained resources, assignments are only allowed in pairs. Please note this important point and try to match up with a partner as soon as possible. Submissions are only allowed through the submission system. To avoid submitting a large number of Xv6 builds you are required to submit a patch (i.e. a file which patches the original Xv6 and applies all your changes). You may use the following instructions to guide you through the process:

1. Backup your work before proceeding!

2. Before creating the patch review the change list and make sure it contains all the changes that you applied and nothing more. Modified files are automatically detected by git but new files must be added explicitly with the "git add" command:

   ```
   > git add . –Av
   > git commit –m "commit message"
   ```

3. At this point you may examine the differences (the patch)

   ```
   > git diff origin
   ```

4. Once you are ready to create a patch simply make sure the output is redirected to the patch file:

   ```
   > git diff origin > ID1_ID2.patch
   ```

5. Finally, you should note that the graders are instructed to examine your code on lab computers only! We advise you to test your code on lab computers prior to submission, and in addition after submission to download your assignment, create a clean Xv6 folder (by using the git clone command), apply the patch, compile it, and make sure everything runs and works. The following command will be used by the testers to apply the patch:

   ```
    > patch –p1 < ID1_ID2.patch
   ```