Machine Problem 4 - File System

CSIE3310 - Operating Systems
National Taiwan University

Total Points:	100
Release Date:	May 10
Due Date:	May 25, 23:59:00
TA e-mail:	`ntuos@googlegroups.com`
TA hours:	05/13 09:00-10:00, 05/19 09:00-10:00 Google Meet

Contents

1 Summary

In this MP, you will learn the fundamental knowledge of the file system by adding two features to xv6: large files and symbolic links. We strongly recommend you read Chapter 8 (file system) in xv6 hand book while you trace code. This gives you a quick overview of how xv6 implements its file system.

2 Environment Setup

We provide the whole xv6 code you need for this MP. All the necessary files are given, and you only need to modify the missing parts. The following instructions will help you complete the setup.

1. Download xv6 code from NTUCOOL.
2. Pull the docker image.

```
$ docker pull ntuos/mp4
```

3. Run the following command to start a container and mount the directory.

```
$ cd mp4
$ docker run -it --name mp4 -v $(pwd)/xv6:/home/xv6 ntuos/mp4
```

4. After getting into the container, run the command to start an xv6.

```
(container)$ make qemu
```

If everything is fine, then you are able to start working on the MP.

3 Problems 1: Large Files (30 points)

3.1 Description

In this problem, you have to increase the maximum size of an xv6 file. Currently xv6 files are limited to 268 blocks, or $268 * \texttt{BSIZE}$ bytes (`BSIZE` is 1024 bits in xv6). This limitation comes from the fact that an xv6 `inode` contains 12 "direct" block numbers and one "singly-indirect" block number, which refers to a block that holds up to 256 more block numbers, for a total of $12 + 256 = 268$ blocks.

Your task is to change the xv6 file system code to support large files. You need to implement **"doubly-indirect"** blocks, containing 256 addresses of singly-indirect blocks, each of which can contain up to 256 addresses of data blocks. By modifying one direct block into a "doubly-indirect" block, a file will be able to consist of up to 65803 blocks (11 from direct blocks, 256 from singly-indirect blocks and 256×256 from doubly-indirect blocks). However, it is still not sufficient to accomplish our goal. **You will need to implement extra doubly-indirect blocks to achieve up to** 66666 **blocks.**

3.2 Guidelines and Hints

1. Checkout `TODO` in the skeleton code.
2. `kernel/fs.h` describes the structure of an on-disk inode. The address of the data block is stored in `addrs`. Note that the length of `addrs` is always 13.
3. Make sure you understand `bmap()`. Write out a diagram of the relationships among `ip->addrs[]`, indirect blocks, doubly-indirect blocks, and data blocks. Make sure you understand why replacing a direct block with a doubly-indirect block increases the maximum file size by $256 \times 256 - 1$ blocks.
4. If you change the definition of `NDIRECT`, you probably have to change the declaration of `addrs[]` in struct inode in `file.h`. Make sure that struct inode and struct dinode have the same number of elements in their `addrs[]` arrays.
5. If you change the definition of `NDIRECT`, make sure to run `make clean` to delete `fs.img`. Then run `make qemu` to create a new `fs.img`, since `mkfs` uses `NDIRECT` to build the file system.
6. Don't forget to `brelse()` each block that you `bread()`.
7. You should allocate indirect blocks and doubly-indirect blocks only as needed, like the original `bmap()`.
8. Make sure `itrunc()` frees all blocks of a file, including doubly-indirect blocks.
9. You can pass problem 1 with modifying only: `fs.c`, `fs.h` and `file.h`.

3.3 How to Test

There are 2 public testcases and 3 private testcases. Passing each of them can get 6 points. You can run `make bigfile` in the container to check your grade for public testcases. The public testcases are implemented in `bigfile`. Feel free to check it out. As to private testcases, they will be added to `bigfile_private.c` while grading. Note that the public testcases do not reach 66666 blocks, so make sure your code can handle such large file to pass private testcases.

```
# make bigfile
........ qemu log........
Testing large files: (38.2s)
  Large files: public testcase 1 (6 points): OK
  Large files: public testcase 2 (6 points): OK
Score: 12/12
```

4 Problems 2: Symbolic Links to Files (30 points)

4.1 Description

In this problem you will add symbolic links to xv6. Symbolic links (or soft links) refer to a linked file by `pathname`; when a symbolic link is opened, the kernel follows the link to the referred file. Symbolic links resemble hard links, but hard links are restricted to pointing to files on the same disk, while symbolic links can cross disk devices. Although xv6 doesn't support multiple devices, implementing this system call is a good exercise to understand how `pathname` lookup works. You will implement the symlink(char *target, char *path) system call, which creates a new symbolic link at `path` that refers to a file named `target`. In addition, you also need to handle `open` when encountering symbolic links. If the target is also a symbolic link, you must recursively follow it until a non-link file is reached. If the links form a cycle, you must return an error code. You may approximate this by returning an error code if the depth of links reaches some threshold (e.g., 10). However, when a process specifies `O_NOFOLLOW` flags, `open` should open symbolic links (not targets).

4.2 Guidelines and Hints

1. Checkout `TODO` in the skeleton code.
2. Checkout `kernel/sysfile.c`. There is an unimplemented function `sys_symlink`. Note that system call symlink is already added in xv6, so you don't need to worry about that.
3. Checkout `kernel/stat.h`. There is a new file type `T_SYMLINK`, which represents a symbolic link.
4. Checkout `kernel/fcntl.h`. There is a new flag `O_NOFOLLOW` that can be used with the open system call.
5. The target does not need to exist for the system call to succeed.
6. You will need to store the target path in a symbolic link file, for example, in inode data blocks.
7. `symlink` should return an integer representing `0(success)` or `-1(failure)` similar to link and unlink.
8. Modify the `open` system call to handle paths with symbolic links. If the file does not exist, open must fail.
9. Don't worry about other system calls (e.g., `link` and `unlink`). They must not follow symbolic links; these system calls operate on the symbolic link itself.
10. You do not have to handle symbolic links to directories in this part.
11. You can pass problem 2 with modifying only: `sysfile.c`.

4.3 How to Test

There are 2 public testcases and 3 private testcases. Passing each of them can get 6 points. You can run `make symlinkfile` in the container to check your grade for public testcases. The public testcases are implemented in `symlinkfile`. Feel free to check it out. As to private testcases, they will be added to `symlinkfile_private.c` while grading.

```
# make symlinkfile
........ qemu log........
Testing symbolic links to files (public): (2.1s)
  Symbolic links to files: public testcase 1 (6 points): OK
  Symbolic links to files: public testcase 2 (6 points): OK
Score: 12/12
```

5 Problems 3: Symbolic Links to Directories (30 points)

5.1 Description

Instead of just implementing symbolic links to files, now you should also consider symbolic links to directories. We expect that a symbolic link to a directory should have these properties:

1. It can be part of a path, and will redirect to what it links to.
2. You can `cd` a symbolic link if it links to a directory.

For example, `symlink("/y/", "/x/a")` creates a symbolic link `/x/a` links to `/y/`. The actual path of `/x/a/b` should be `/y/b`. Thus, if you write to `/x/a/b`, you actually write to `/y/b`. Also, if you `cd` into /x/a, your working directory should become /y/.

5.2 Guidelines and Hints

1. Checkout `TODO` in the skeleton code.
2. You can leave `sys_symlink` function unchanged, since symbolic links store paths as strings. There is no difference between a file path and a directory path.
3. You have to handle paths that consist of symbolic links. Check `namex` function in `fs.c`.
4. You have to handle symbolic links in `sys_chdir` function. Like problem 2, you need to avoid infinite loops.
5. You can pass problem 3 with modifying only: `sysfile.c` and `fs.c`.

5.3 How to Test

There are 2 public testcases and 3 private testcases. Passing each of them can get 6 points. You can run `make symlinkdir` in the container to check your grade for public testcases. The testcases are implemented in `symlinkdir`. Feel free to check it out. As to private testcases, they will be added to `symlinkdir_private.c` while grading.

```
# make symlinkdir
........ qemu log........
Testing symbolic links to directories (public): (2.1s)
  Symbolic links to directories: public testcase 1 (6 points): OK
  Symbolic links to directories: public testcase 2 (6 points): OK
Score: 12/12
```

6 Report (10 points)

6.1 Description

Your report should include two parts:

1. Briefly explain how you solve each problem. If your solutions are reasonable and correct, you can get the points. Note that you can get points even if you fail to solve the problems. (10 points)
   (a) Problem 1: Large Files (4 points)
   (b) Problem 2: Symbolic Links to Files (3 points)
   (c) Problem 3: Symbolic Link to Directories (3 points)
2. We encouraged you to help other students. Please describe how you helped other students here. You should make the descriptions as short as possible, but you should also make them as concrete as possible (e.g., you can screenshot how you answered other students' questions on NTU COOL). Please note that you will not get any penalty if you leave empty here. Please also note that this bonus is not for you to do optimization, so we will not release the grading criteria and the grades. Regarding the final letter grades, it is very likely that this does not help — you will get promoted to the next level only if you are near the boundary of levels and you have significant contributions. (0 point)

6.2 Format

Your report should be a PDF file. Do not exceed two pages for the first part. Do not exceed two pages for the second part either.

7 Submissions

7.1 xv6 code

Please compress your xv6 source code as `<whatever>.zip` and upload to NTUCOOL. The filename does not matter since NTUCOOL will rename your submissions. Never compress files we do not request, such as `.o`, `.d`, `.asm` files. You can run `make clean` in the container before you compress. Make sure your xv6 can be compiled by `make qemu`.

We will unzip your submission by running `unzip *.zip`. Your folder structure **AFTER UNZIP** should be

```
<student_id>
|
+-- xv6
    |
    +-- kernel/
    |
    +-- user/
    |
    +-- Makefile
```

Note that **the English letters in the `<student_id>` must be lowercase**. E.g., it should be `d08922025` instead of `D08922025`.

7.2 Report

Please submit your report on Gradescope. There is an assignment called **MP4 Report**. Make sure you select the correct assignment.

8 Grading Policy

- You will get 0 points if we cannot compile your submission.

- You will be deducted 10 points if the folder structure is wrong. Using uppercase in the `<student_id>` is also a type of wrong folder structure.
- If your submission is late for n days, your score will be $max(raw\_score - 20 \times \lceil n \rceil, 0)$. Note that you will not get any points if $\lceil n \rceil >= 5$.