Machine Problem 4 - File System

CSIE3310 - Operating Systems
National Taiwan University

Total Points:	100
Release Date:	May 14
Due Date:	May 27, 23:59:00
TA e-mail:	`ntuos@googlegroups.com`
TA hours:	Mon, Wed. 13:00-14:00 before the due date, CSIE Building B04

Contents

1 Overview

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. Downloadthe MP4.zip from NTUCOOL, unzip it, and enter the folder.

```
$ unzip MP4.zip
$ cd mp4
```

2. Pull the docker image.

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

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

```
$ docker run -it -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 Problem 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 bytes 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. **Your goal is to configure the inode structure such that a file can consist of exactly 67078 blocks, using a combination of direct, singly-indirect, and doubly-indirect blocks. Calculate the required number of each type of block to meet this exact number without exceeding or falling short.**

3.2 Guidelines and Hints

1. You can pass problem 1 with modifying only: `fs.h` and `file.h`, `fs.c` under `xv6/kernel/`.
2. Checkout `TODO` in the skeleton code.
3. `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.

4. If you change the definition of `NDIRECT`, you probably have to change the declaration of `addrs[]` in struct inode in `kernel/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. 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.
7. You should allocate indirect blocks and doubly-indirect blocks only as needed, like the original `bmap()`.
8. Don't forget to `brelse()` each block that you `bread()`.
9. Make sure `itrunc()` frees all blocks of a file, including doubly-indirect blocks.

3.3 How to Test

There are 2 public testcases (6% each) and 3 private testcases (6% each). You can run `make mp4_1` in the container to check your grade for public testcases. The public testcases are implemented in `mp4_1`. You can change variable `target` to test your implementation. As to private testcases, they will be added to `mp4_1_private` while grading. Note that the public testcases do not reach 67078 blocks, so make sure your code can handle such large file to pass private testcases.

```
# make mp4_1
........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 Problem 2: Symbolic Links to Files (30 points)

4.1 Description

In this problem, you have to 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., 20). However, when a process specifies `O_NOFOLLOW` flags, `open` should open symbolic links (not targets).

4.2 Guidelines and Hints

1. You can pass problem 2 with modifying only: `sysfile.c` under `xv6/kernel/`.
2. Checkout `TODO` in the skeleton code.
3. Checkout `kernel/sysfile.c`. There is an unimplemented function `sys_symlink()`. Note that the system call `symlink()` is already added in xv6, so you don't need to worry about that.
4. Checkout `kernel/stat.h`. There is a new file type `T_SYMLINK`, which represents a symbolic link.
5. Checkout `kernel/fcntl.h`. There is a new flag `O_NOFOLLOW` that can be used with the open system call.
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. The target does not need to exist for the symlink system call to succeed. However, If the path exists, symlink must fail.
9. In our test cases, the target and path in symlink system call are always `absolute paths`, starting from the root directory (/).
10. Modify the `open` system call to handle paths with symbolic links. If the file does not exist, open must fail.
11. 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.
12. You do not have to handle symbolic links to directories.
13. We provide a `symln` command that you can use for creating symbolic links easily during your tests. This command is available within the xv6 environment.

4.3 How to Test

There are 2 public testcases (9% each) and 3 private testcases (2%, 4%, 6%). You can run `make mp4_2` in the container to check your grade for public testcases. The public testcases are implemented in `mp4_2`. Feel free to check it out. As to private testcases, they will be added to `mp4_2_private.c` while grading.

```
# make mp4_2
........qemu log........
Testing symbolic links to files (public): (2.1s)
  Symbolic links to files: public testcase 1 (9 points): OK
  Symbolic links to files: public testcase 2 (9 points): OK
Score: 18/18
```

5 Problem 3: Reverse Symbolic Links Lookup (40 points)

5.1 Description

In this problem, you have to implement a system call `revreadlink`, which is short for `"reverse readlink"`. Unlike the `readlink` system call in most operating systems, which retrieves the target of a symbolic link, `revreadlink` aims to reverse this process by identifying all symbolic links that refer to a given file. The `revreadlink` system call will accept a target name as its input and return a collection of all symbolic links that `directly` point to the file.

5.2 Function Prototype

```
  int revreadlink(char *target, char *buf, int bufsize);
```

1. `target`: The pathname of the file whose links are to be identified, and you have to find all links that refer to `target`. In our testcases, `target` is always a file.
2. `buf`: A user-space buffer where the paths of the symbolic links will be stored. You must implement the functionality to write the resulting symbolic link paths directly into this buf. The buf should be managed carefully to handle string operations efficiently:
   - No leading whitespace at the beginning of the buf.
   - A single space must separate each path; ensure only one space is used.
   - No need of any trailing characters, spaces after the final path.
   - All paths must be absolute paths, beginning from the root directory (start with /).
   - If there are multiple paths, the order of these paths in the buffer can vary.
3. `bufsize`: The size of the buffer, which determines the maximum amount of data that can be returned. You do not need to worry about the returned data exceeding this size as test cases will be designed to fit within the buffer's capacity.

The function should return the length of the buffer actually used, which includes the lengths of all the paths and the spaces separating them. If there are no symbolic links that point to the `target`, the buffer(`buf`) provided to store the paths of symbolic links should remain empty, and the system call should return 0.

Below is an example illustrating the format of the buf obtained after invoking the `revreadlink` system call, along with its corresponding return value.

```
Assume there is a file named "/testtargetfile", which is the target of two symbolic
link files:
    /testa
    /testdir/testb
After calling "revreadlink("/testtargetfile", buf, bufsize)", the resulting content of
"buf" should be either of the following:
   "/testa /testdir/testb"
   "/testdir/testb /testa"
the return value of the system call should be 21
```

5.3 Guidelines and Hints

1. You can pass problem 3 with modifying only: `sysfile.c` under `xv6/kernel/`.
2. Implementation Strategy:
   - Begin your search at the root directory using `namei("/")` to obtain the root inode.
   - Traverse each directory from the root, checking each directory entry. Checkout the `struct dirent` from `fs.h`, which includes the inode number (`inum`) and name.
   - For each entry:
     - Use `iget()` to fetch the inode associated with each directory entry.
     - If the inode type is `T_SYMLINK`, compare its target with the specified target.
     - If the inode is a directory (`T_DIR`), recursively explore the directory, constructing absolute paths correctly.
   - Use `copyout()` for transferring data to user space.
   - Return the total bytes written to user buffer.
3. Manage file system operations with `begin_op()` and `end_op()`, and manage inode locks with `ilock()`, `iunlock()` and `iunlockput()`. Notice that `iget()` may increase the inode's reference count, so once done with the inode, use `iunlockput()` rather than just use `iunlock()` to release the lock and decrement the reference count.
4. We provide a revreadlink command that you can use during your tests. This command is available within the xv6 environment.

5.4 How to Test

There are 3 public testcases (8% each) and 2 private testcases (8% each). You can run `make mp4_3` in the container to check your grade for public testcases. The testcases are implemented in `mp4_3`. Feel free to check it out. As to private testcases, they will be added to `mp4_3_private.c` while grading.

```
# make mp4_3
........qemu log........
Testing reverse readlink (public): (15.1s)
  Reverse readlink: public testcase 1 (8 points): OK
  Reverse readlink: public testcase 2 (8 points): OK
  Reverse readlink: public testcase 3 (8 points): OK
Score: 24/24
```

6 Bonus Report (Optional)

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 it 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.

7 Submissions and scoring

Please submit two separate file packages, upload the code package to NTUCOOL, and submit the report to Gradescope.

7.1 xv6 code

Run this command to pack your code into a zip named in your `lowercase` student ID, for example, b12345678.zip. Submit this file to Machine Problem 4 section on NTUCOOL.

```
make STUDENT_ID=b12345678 zip # set your ID here
```

7.2 Report

Submit your report to Bonus Report on Gradescope in one PDF file.

7.3 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 $\lceil n \rceil \geq 5$.
- You can submit your work as many times as you want, but only the last submission will be graded. Previous submissions will be ignored.