Operating System 2021 Machine Problem Site

The site provides the information for machine problems for Operating System Course offered in Spring 2021 at National Taiwan University

MP 4: File System

In this MP, you will learn the fundamental knowledge of file system by adding two features to xv6: large files and symbolic links.

Deadline

- 2021/06/14 23:59
- No early bird
- No late submission

Set Up

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 set up.

1. Download xv6 code from github classroom.
2. Pull docker image

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

3. Run the image and mount the directory The command below create a container from the image, and mount the xv6 code to it. Then, you directly interact with it via its shell.

   ```
   $ cd <xv6_directory>
   $ docker run -it --name <CONTAINER_NAME> -v "$(pwd)":/home/mp4 ntuos/mp4
   ```

4. Run xv6. Inside the container you run the command below. If everything is fine, then you are able to start up the xv6 now.

```
(container)$ cd /home/mp4
(container)$ make qemu
```

If xv6 starts up successfully, congrats! Now you can start to play with MP4.

Reminders

⭐ 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 implement its file system.

Problems

1. Large Files (4 points)

Preliminaries

Change the `FSSIZE` in `kernel/param.h` to 200000. This parameter controls the total number of blocks in xv6.

Description

In this problem you'll increase the maximum size of an xv6 file. Currently xv6 files are limited to 268 blocks, or 268*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.

The `bigfile` command creates the longest file it can, and reports the size. The command is already there, so you can try to run:

```
$ bigfile
..
wrote 268 blocks
bigfile: file is too small
```

The test fails because `bigfile` expects to create a file with 66666 blocks, but unmodified xv6 limits files to 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 (256 x 256 + 256 + 11). It is 11 because we have to sacrifice one direct block number for doubly-indirect block.

It is still not sufficient to accomplish our goal. You will need to implement an extra indirect block to achieve 66666 blocks.

⚠ `bigfile` may take a couple of minutes to run.

Test You Code

You can use `bigfile` to test your code. If you are able to get the following output, then you are done in this part.

```
$ bigfile
..........................................................................................
wrote 66666 blocks
done; ok
```

Where to Start & Hints

1. Checkout TODOs in the skeleton code.
2. `kernel/fs.h` describes the structure of an on-disk inode. The address of 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 between `ip->addrs[]`, the indirect block, the doubly-indirect block and the singly-indirect blocks it points to, and data blocks. Make sure you understand why adding a doubly-indirect block increases the maximum file size by 256*256-1 blocks.
4. If you change the definition of `NDIRECT`, you'll 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 double-indirect blocks.
9. You can pass by only modifying:
    - `fs.c`
    - `fs.h`
    - `file.h`

2. Symbolic Links to Files (4 points)

Description

In this exercise 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 resembles hard links, but hard links are restricted to pointing to file 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. When a process specifies `O_NOFOLLOW` flags, `open` should open symlinks (not targets). 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).

Test You Code

You can use `symlinktest` to test your code. If you are able to get the following output, then you are done in this part.

```
$ symlinktest
...
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
...
```

Where to Start & Hints

1. Checkout TODOs 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 `user/symlinktest.c`. We will use it for grading, so test your code by running it.
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. The target does not need to exist for the system call to succeed.

7. You will need to store the target path in a symbolic link file, for example, in inode data blocks.
8. `symlink` should return an integer representing success (0) or failure (-1) similar to link and unlink.
9. Modify the `open` system call to handle paths with symbolic links. If the file does not exist, open must fail.
10. 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.
11. You do not have to handle symbolic links to directories for this part.
12. You can pass by only modifying:
    - `sysfile.c`

3. Symbolic Links to Directories (2 points)

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 a 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`, yo actually write to `/y/b`. Also, if you `cd` in to `/x/a`, your working directory should become `/y/`.

You can check `testsymlinkdir` function in `symlinktest.c`. You will get this bonus point if you pass all tests.

Test You Code

You can use `symlinktest` to test your code. If you are able to get the following output, then you are done in this part.

```
$ symlinktest
...
Start: test symlinks to directory
test symlinks to directory: ok
...
```

Where to Start & Hints

1. Checkout TODOs in the skeleton code.

2. You can leave `sys_symlink` function unchanged, since symbolic links store path as a string. 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 may want to handle symbolic links in `sys_chdir` function.
5. You can pass by only modifying:
    - `sysfile.c`
    - `fs.c`

4. Report (1 point)

Your report should include two parts:

1. Briefly explain how you solve each problem.
    1. Problem 1: Large files
    2. Problem 2: Symbolic links to files
    3. Problem 3: Symbolic links to directories
2. As mentioned in the first lecture, 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.

The report should follow these format:

- Should be a PDF.
- Do not exceed two pages for the first part.
- Do not exceed two pages for the second part either.

Submit Rules

- Submit `xv6/` to github classroom.
    - Do not submit files generated by `Makefile`. You should `make clean` before you submit.
- Submit report to gradescope **MP4 Report**.

Grading Policy

Total (11 points):

- Large Files (4 points)
- Symbolic links to Files (4 points)
- Symbolic links to Directories (2 points)
- Report (1 point)

How we will grade your code:

- We will use the original `Makefile`. If you add any file or modify the structure of `xv6`, make sure it performs the same with the original `Makefile`. (We strongly recommend you keep it simple.)
- We will slightly modify the output of `bigfile.c` and `symlinktest.c` to prevent naive printing.
- Three problems will be graded by the same code, so you should handle them at the same time.
- To maintain grading consistency, please do not add or install any other packages outside the xv6. Otherwise, you should make sure your code perform the same in the original environment.
- You will get all points for each subproblem if you pass the testing commands. Otherwise, you don't get any point.

How we will grade your report:

- If your solutions are reasonable and correct, you get the following point:
  - Problem 1: Large files (0.4 point)
  - Problem 2: Symbolic links to files (0.3 point)
  - Problem 3: Symbolic links to directories (0.3 point)
- Note that even if you fail on the implementation, you still can get points if your reports are good.

For example, you pass **Large Files** (4) and **Symbolic Links to Files** (4), but you fail on **Symbolic Links to Directories** (0). You still hand in a complete **Report** (0.4+0.3+0.3). In this case, you get 9 points.

TA Hour and Discussion

If you have any question, there are two ways to ask:

- NTU COOL Discussion (Recommended)
- TA Hour
  - 6/4 10:30 ~ 11:30 (meet.google.com/iut-ofma-ykk)

Empirically, the efficiency of online TA hour is uncertain. We will host a short TA hour first. If the effect is good, then we may host additional TA hours. No matter what, we still strongly recommend you post your questions on NTU COOL.