CSC3150-Instruction-A4

Introduction

This assignment uses xv6, a simple and Unix-like teaching operating system, as the platform to guide you in **implementing the indirect block to support big file management**. In existing implementation, singly-indirect blocks can handle limited blocks that are invalid for large file management. In this assignment, you'll increase the maximum size of an xv6 file by implementing doubly-indirect blocks for further extension.

We suggest you read Chapter 8: File system before writing code.

Preliminaries

The **mkfs** program creates the xv6 file system disk image and determines how many total blocks the file system has; this size is controlled by **FSSIZE** in kernel/param.h. You'll see that **FSSIZE** in the repository for this lab is set to 200,000 blocks. You should see the following output from mkfs/mkfs in the make output:

```
nmeta 70 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 25) blocks 1
```

This line describes the file system that mkfs/mkfs built: it has 70 meta-data blocks (blocks used to describe the file system) and 199,930 data blocks, totaling 200,000 blocks.

If at any point during the lab you find yourself having to rebuild the file system from scratch, you can run 'make clean', which forces make to rebuild fs.img.

Submission

- **Due on: 23:59, 06 Dec 2023**
- **Plagiarism is strictly forbidden.** Please note that TAs may ask you to explain the meaning of your program to ensure that the codes are indeed written by yourself. Please also note that we would check whether your program is too similar to your fellow students' code and solutions available on the internet using plagiarism detectors.
- Late submission: A late submission **within 15 minutes** will not induce any penalty on your grades. But **00:16 am-1:00 am: Reduced by 10%; 1:01 am-2:00 am: Reduced by 20%; 2:01 am-3:00 am: Reduced by 30% and so on.** (e.g., Li Hua submitted a perfect attempt at 2:10 a.m. He will get (100+10 (bonus))*0.7=77p

Format guide

The project structure is illustrated below. You can also use `ls` command to check if your structure is fine. Structure mismatch would cause grade deduction.

For this assignment, you don't need a specific folder for the bonus part. The source folder should contain four files: ***fs.c, file.h, fs.h, sysfile.c***

```
main@ubuntu:~/Desktop/Assignment_4_120010001$ ls
Report.pdf  source/

(One directory and one pdf.)
```

```
main@ubuntu:~/Desktop/Assignment_4_120010001/source$ ls
file.h  fs.c  fs.h  sysfile.c

(two .c files and two .h file)
```

Please compress the folder containing all required files into a single zip file and **name it using your student ID as the code shown below and above, for example, Assignment_4_120010001.zip.** The report should be submitted in pdf format, together with your source code. Format mismatch would cause grade deduction. Here is the sample step for compressing your code.

```
main@ubuntu:~/Desktop$
 zip -q -r Assignment_4_120010001.zip Assignment_4_120010001

main@ubuntu:~/Desktop$ ls
Assignment_4_120010001                 Assignment_4_120010001.zip
```

Instruction Guideline

We limit your implementation within ***fs.c, file.h, fs.h, sysfile.c*** four files, starting with "TODO:" comments. The entry (where you may start learning) of the test program is the main function in **bigfile.*c*, symlinktest.*c* (Bonus)** under the 'xv6-labs-2022/user' directory.

Sections with (*) are introduction sections. These sections **introduce** tools and functions that will help you understand what this system is about and how the system works with these components. You might need to use it for this assignment. Do **NOT CHANGE** them except the TODO parts.

1. For the introduction sections, please figure out how functions work and how to use them.
2. Be sure you have a basic idea of the content before starting your assignment. We believe that those would be enough for handling this assignment.
3. (option) For students who are interested in the xv6 system and want to learn more about it, you are welcome to read "xv6-book" to get more details.
    a. https://pdos.csail.mit.edu/6.828/2022/xv6/book-riscv-rev3.pdf

Sections **without** (*) are TODO sections. In these sections, the logic of how this component/ function should work is a detailed list. You should implement functions in given places.

1. However, no sample code will be shown here. You need to figure out the implementation based on the logic and APIs provided in the introduction sections.

Task1: Large Files

1. In this assignment, 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 in xv6).
    a. This limit 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.
2. The bigfile command creates the longest file it can and reports the size
    a. The template we provide will fail to write 256 blocks. The test fails because bigfile expects to be able to create a file with 65803 blocks, but unmodified xv6 limits files to 268 blocks.
3. You'll change the xv6 file system code to support a "doubly-indirect" block in each inode, containing 256 addresses of singly-indirect blocks, each of which can contain up to 256 addresses of data blocks.
    a. The result will be that a file can consist of up to 65803 blocks, or 256*256+256+11 blocks (11 instead of 12 because we will sacrifice one of the direct block numbers for the double-indirect block).

Definitions*

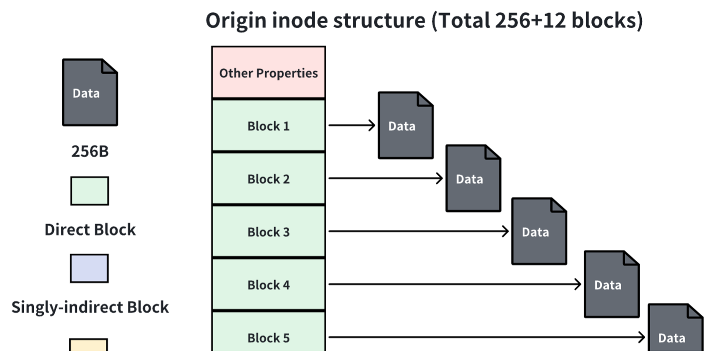
For more details, read <xv6-book> chapter 8.10

Following the hints and definitions above, we have provided you with the modified structure. Please read the comments on the codes.

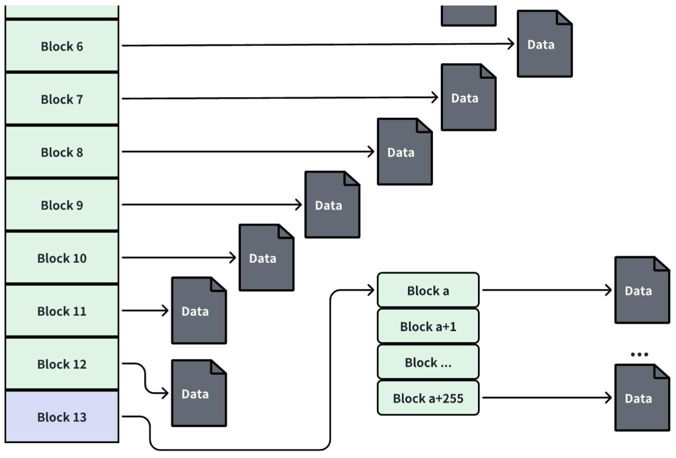
```
// Defined in kernel/fs.h
#define NDIRECT 11 // 12->11 By 3.a, we sacrifice 1 block for "doubly-indirec
#define NINDIRECT (BSIZE / sizeof(uint)) // = 1024/4 = 256
```

```c
#define DNINDIRECT (NINDIRECT * NINDIRECT) // = 256*256
#define MAXFILE (NDIRECT + NINDIRECT + DNINDIRECT) // = 256*256 + 256 + 11

///NOTE: Do not modify the structure
// On-disk inode structure
struct dinode {  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  ///NOTE: +2 instead of +1 because we NDIRECT is change from 12 to 11
  uint addrs[NDIRECT+2];   // Data block addresses
};

//Defined in kernel/file.h
///NOTE: Do not modify the structure
// in-memory copy of an inode
struct inode {  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+2]; ///NOTE: +2 instead of +1 because we NDIRECT is char
};
```

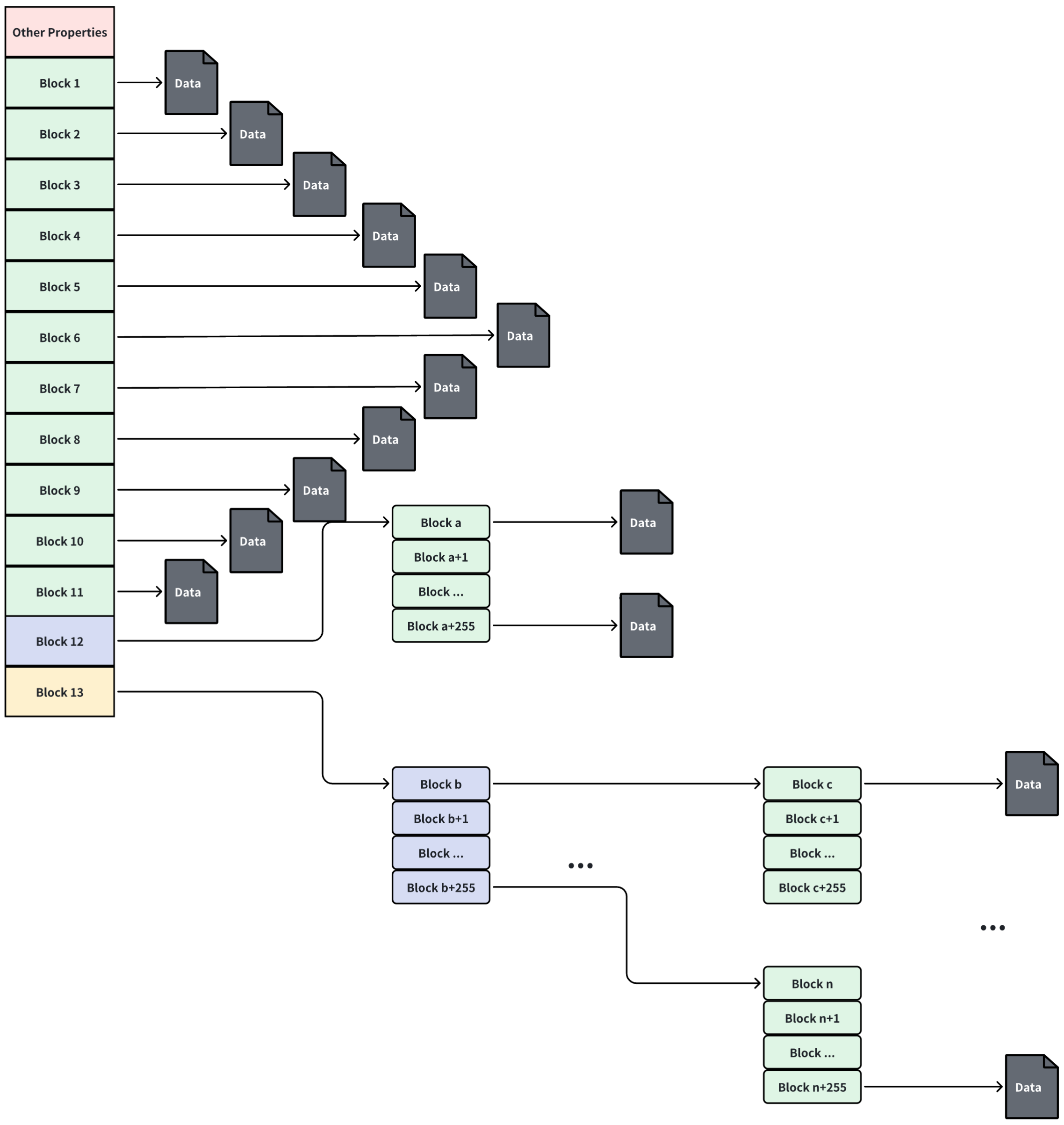
We provide a simple figure for you to have a basic idea of what we are going to achieve



🟨 Doubly-indirect Block



Target inode structure (Total 11+256+256*256 blocks)



APIs*

Block Management

```c
//Defined in fs.c

// Zero a block.
static void bzero(int dev, int bno);

// Blocks.
// Allocate a zeroed disk block.
// returns 0 if out of disk space.
static uint balloc(uint dev);

// Free a disk block.
static void bfree(int dev, uint b);
```

```c
//Defined in bio.c

// Return a locked buf with the contents of the indicated block.
struct buf* bread(uint dev, uint blockno);

// Release a locked buffer.
// Move to the head of the most-recently-used list.
void brelse(struct buf *b);
```

```c
//Defined in log.c

// Caller has modified b->data and is done with the buffer.
// Record the block number and pin in the cache by increasing refcnt.
// commit()/write_log() will do the disk write.
//
// log_write() replaces bwrite(); a typical use is:
//   bp = bread(...)
//   modify bp->data[]
//   log_write(bp)
//   brelse(bp)
void log_write(struct buf *b);
```

Learn from examples

For APIs provided above, they have been used to implement functions. You can learn how to use those functions to develop our system.

You may take a look at how it is used in **bmap(), itrunc(),** bzero(), balloc(), bfree()

Especially read existing code in **bmap() and itrunc()** as these two functions are where we need to modify, and they have already been implemented singly-indirect .

bmap()

See <xv6-book> chapter 8.10

```c
// Inode content
//
// The content (data) associated with each inode is stored
// in blocks on the disk. The first NDIRECT block numbers
// are listed in ip->addrs[].  The next NINDIRECT blocks are
// listed in block ip->addrs[NDIRECT].

// Return the disk block address of the nth block in inode ip.
// If there is no such block, bmap allocates one.
// returns 0 if out of disk space.

// TODO: implement doubly-indirect
static uint bmap(struct inode *ip, uint bn);
```

bmap() is called both when reading and writing a file. When writing, bmap() allocates new blocks as needed to hold file content, as well as allocating an indirect block if needed to hold block addresses.

bmap() deals with two kinds of block numbers. The bn argument is a "logical block number" -- a block number within the file, relative to the start of the file. The block numbers in ip->addrs[], and the argument to bread(), are disk block numbers. You can view bmap() as mapping a file's logical block numbers into disk block numbers.

itrunc()

See <xv6-book> chapter 8.10

itrunc frees a file’ s blocks, resetting the inode’ s size to zero. itrunc (kernel/fs.c:430) starts by freeing the direct blocks(kernel/fs.c:436-441), then the ones listed in the indirect block (kernel/fs.c:446- 449), and finally the indirect block itself (kernel/fs.c:451-452).

(TODO) Modify to support doubly-indirect block

```c
///TODO: modify it to support doubly-link
// Inode content
//
// The content (data) associated with each inode is stored
// in blocks on the disk. The first NDIRECT block numbers
// are listed in ip->addrs[].  The next NINDIRECT blocks are
// listed in block ip->addrs[NDIRECT].

// Return the disk block address of the nth block in inode ip.
// If there is no such block, bmap allocates one.
// returns 0 if out of disk space.
static uint bmap(struct inode *ip, uint bn);
```

Modify bmap() so that it implements a doubly-indirect block in addition to direct blocks and a singly-indirect block.

You'll have to have only 11 direct blocks, rather than 12, to make room for your new doubly-indirect block; you're not allowed to change the size of an on-disk inode.

> i.e., Do **NOT** modify the structure or size of *addrs* in **dinode** or **inode**. We have already set it up for you.

```c
///TODO: add discard of doubly-link correspondingly
// Truncate inode (discard contents).
// Caller must hold ip->lock.
void itrunc(struct inode *ip);
```

Hint

- The first 11 elements of ip->addrs[] should be direct blocks
- The 12th should be a singly-indirect block (just like the current one)
- The 13th should be your new doubly-indirect block. You are done with this exercise when bigfile writes 65803 blocks
- Remember that it needs modification to release Double-Indirect blocks (modify **itrunc()**)

Task2(Bonus): Symbolic links

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.

(TODO) Implementation of symlink

You will implement the symlink(char *target, char *path) system call, which creates a new symbolic link at path that refers to the file named by target. For further information, see the man page symlink.

Your solution is complete when you pass all cases in **symlinktest**.

Hints

- Add a new file type (T_SYMLINK) to kernel/stat.h to represent a symbolic link. (We already add it for you)
- Add a new flag to kernel/fcntl.h, (O_NOFOLLOW), that can be used with the open system call. Note that flags passed to open are combined using a bitwise OR operator, so your new flag should not overlap with any existing flags. This will let you compile user/symlinktest.c once you add it to the Makefile. (We already define it for you)
- Implement the symlink(target, path) system call to create a new symbolic link at the path that refers to target. Note that 'target' does not need to exist for the system call to succeed. You will need to choose somewhere to store the target path of a symbolic link, for example, in the inode's data blocks. symlink should return an integer representing success (0) or failure (-1), similar to link and unlink.
- Modify the **open** system call to handle the case where the path refers to a symbolic link. If the file does not exist, open must fail. When a process specifies O_NOFOLLOW in the flags to open, open should open the symlink (and not follow the symbolic link).
- If the linked file 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).
- Other system calls (e.g., link and unlink) must not follow symbolic links; these system calls operate on the symbolic link itself.
- You do not have to handle symbolic links to directories for this lab.

Grading Rules

You can test the correctness of your code using the following commands under '~/xv6-labs-2022' directory.

Test Task1

To run Task1, use the following command

```
make clean
make qemu
bigfile
```

By running the template we provide, you will receive the following information that tells you to implement functions for big file.

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

When you finish Task1 correctly, you should see the following output

```
$ bigfile
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
..........................
wrote 65803 blocks
done; ok
```

Test Task2

To run Task1, use the following command

```
make clean
make qemu
symlinktest
```

Template Output:

```
$ symlinktest
Start: test symlinks
FAILURE: symlink b -> a failed
Start: test concurrent symlinks
test concurrent symlinks: ok
```

Target Output:

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

Program part 90' + bonus 10'

bigfile	40p
Compile Success	50p
symlinktest (bonus)	10p

Report part 10'

You shall strictly follow **the provided latex template** for the report, where we have emphasized important parts and respective grading details. **Reports based on other templates will not be graded**.