CS 354 Spring 2024

Lab 5: File System [200 pts]

Due: 4/21/2024 (Sun.), 11:59 PM

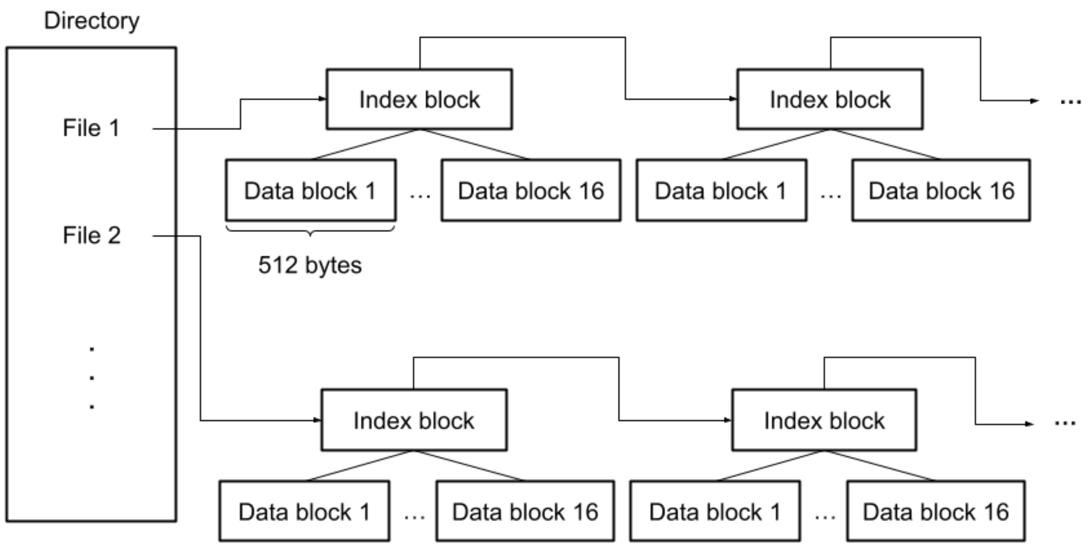
1. Introduction

In this lab, you will be tasked with modifying the i-block implementation in the Xinu file system.

2. Background

2.1. Xinu File System Overview

In Xinu's file system, a flat, non-hierarchical linked index block structure is used to store file content. Each index block contains a fixed number of pointers to data blocks where the actual file content is stored. Each index block also contains a pointer to the next index block. These pointers form a linked list, where their corresponding data blocks represent the continuous content in the file. This is illustrated in the image below.



Read Chapter 19 of the XINU textbook to get a basic understanding of the structures used to implement the XINU file system. In particular, be familiar with the purpose and usage of the following data structures:

- `lfiblk` - an index block (iblock)
- `lflcblk` - an "opened" file in memory

*The rest of this lab assumes that you are familiar with the concepts covered in chapter 19 of the XINU textbook. Please make sure that you do have a solid grasp of the chapter before proceeding. It can **save you a lot of time**.*

2.2. Linux File System – The "Indirect" pointers

In Xinu's file system, a flat, non-hierarchical linked index block structure is used to store file content efficiently. However, this structure is different from the hierarchical file structure commonly found in operating systems like Linux.

In a typical Linux file system, rather than having a linked list of i-blocks, indirect blocks are used to extend the storage capacity of a file beyond what can be directly addressed by the inode itself. As a file grows in its size, the inode allocates blocks of disk space to store the file's data. Initially, the inode contains pointers to a certain number of data blocks, known as direct blocks. However, if the file size exceeds the capacity of these direct blocks, the inode employs indirect blocks to access additional data blocks.

Here's how indirect blocks work with the Linux inode structure:

- Direct Blocks: The inode contains pointers to a certain number of direct data blocks. Each direct block can store a fixed amount of data.
- Indirect Block: When the file size exceeds the capacity of the direct blocks, the inode uses an indirect block. This indirect block contains pointers to additional data blocks, known as singly indirect blocks.
- Singly Indirect Blocks: Singly indirect blocks are blocks that contain pointers to data blocks. Each pointer in the singly indirect block points to a data block containing file data.
- Doubly Indirect Blocks: If the file size continues to grow and requires more data blocks, the inode employs doubly indirect blocks. Doubly indirect blocks contain pointers to singly indirect blocks.
- Triply Indirect Blocks: In extremely large files, where even doubly indirect blocks are not sufficient, the inode utilizes triply indirect blocks. Triply indirect blocks contain pointers to doubly indirect blocks.

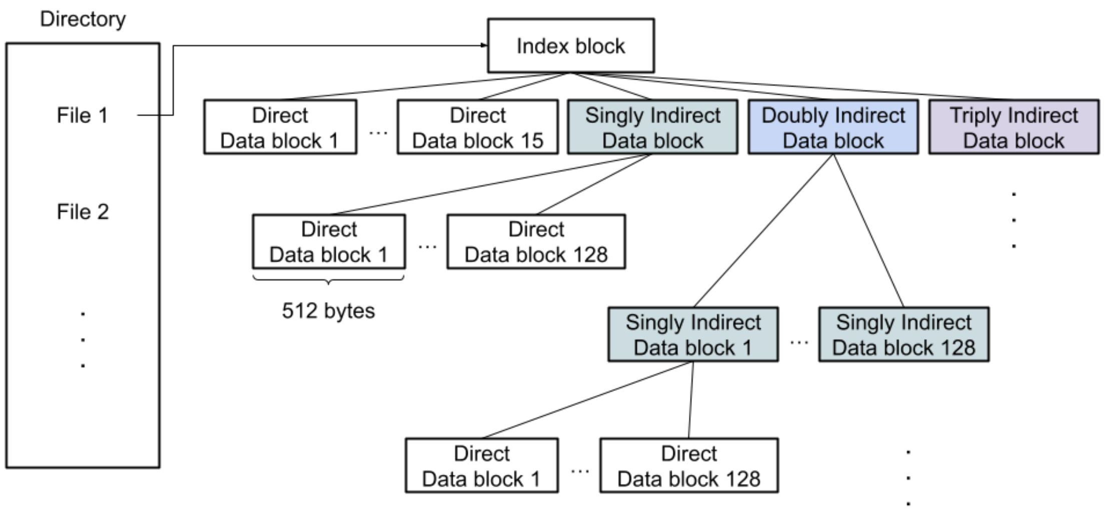
This hierarchical structure allows the Linux file system to efficiently manage large files by extending the storage capacity beyond the direct blocks provided by the inode.

We will adapt the i-block structure in XINU following the inode structure used in Linux.

The original i-block structure in XINU contains 18 pointer-sized integers, where the first two contain the pointer to the next i-block and meta-information about the file (the offset). With a hierarchical design, however, where each file has its unique i-block, we no longer need the first two fields in the i-block.

In order to keep the inode the same size, we choose to utilize the first 15 pointers in the index block as direct data blocks, followed by three more pointers, utilized as singly, doubly, and triply indirect blocks respectively.

Below is an illustration of the modified i-block structure that we will be using to implement the file system. Note that each file only has a single index block now, with direct, indirect, doubly indirect, and triply indirect data blocks.



3. The Alternative i-block Implementation (200 pts)

The objective of this lab is to implement a unix-like inode structure, where indirect blocks are employed to replace the flat, non-hierarchical linked index block structure.

3.1. Getting Started

*Please use a **modified copy of XINU, lab5.tar.gz**, as a starting place of this lab.*

You can obtain the copy by running

```
tar xzvf /homes/cs354/lab5.tar.gz
```

In the provided starter code, you will find a directory “device/lifs”, which is cloned from the original “device/lfs” directory that comes with XINU (the added “i” is supposed to stand for “indirect”). The “indirect” version has been adapted to facilitate implementation according to the hierarchical i-block structure mentioned above to support the alternative, unix-like i-block implementation.

For example, you can compare the following structures to see how they are adapted exactly:

- `lfiblk` v.s. `lifiblk`
- `lflcblk` v.s. `liflcblk`

You will find some test code in main.c to help get you started. The test code given to you demonstrates that common operations such as `putc`, `write`, `read` already work correctly, as long as the file size does not exceed the value of `LIF_AREA_DIRECT`.

With files larger than the direct data block threshold, XINU will trigger an unimplemented error. And your goal is to implement support for larger files that make use of the indirect blocks.

As noted before, main() serves as an app for your own testing purposes. The TAs when evaluating your code will use their own main.c to evaluate your XINU kernel modifications. If you need to add declarations, please add them in include/lab5.h.

3.2. Key threshold values

You will need to refer to a few key threshold file size values to decide when it is necessary to utilize singly, doubly, and triply indirect data blocks. Here are some questions that might help guide your thought:

1. What is the maximum size of a file using only direct data blocks (recall there are 15 of them)? Which macro in `lifilesys.h` is set to this value?
2. How much is the maximum size of a file increased by utilizing singly indirect data blocks? Which macro is set to this value?
3. How much is the maximum size of a file increased by utilizing doubly indirect data blocks on top of singly indirect data blocks? Which macro is set to this value?
4. How much is the maximum size of a file increased by adding triply indirect data blocks? Which macro is set to this value?

3.3 Implementation

To get full points for this section, you only need to implement support for **up to singly indirect data block**. You will only be evaluated on the correct functionality of the following high-level generic operations on files:

- `putc`
- `getc`
- `seek`
- `read`

- `write`

According to the declaration in `config/Configuration`, for our `lifs` device, the above interface functions map to the following implementation functions:

- `liflputc`
- `liflgetc`
- `liflseek`
- `liflread`
- `liflwrite`

Since `reliflread` and `liflwrite` delegate their tasks to `liflgetc` and `liflputc` under-the-hood, and both `liflgetc` and `liflputc` uses `lifsetup` to synchronize between the in-memory structure and the randisk, **it suffices to have most of the modification made in `device/lifs/lifsetup.c` for this lab**.

To test your implementation, you should use different sized files and invoke the above functions interleavingly to verify that the indirect blocks are fetched and updated correctly.

4. Bonus problem (25 pts)

As a bonus challenge, you have the option to implement support for doubly and triply indirect blocks in the Xinu file system. By implementing these blocks, you will gain experience in handling complex data structures and optimizing file system performance for large files.

Note: The bonus problem is optional and is intended to provide an additional challenge for students who wish to explore advanced topics in file system design and implementation. While completing the bonus problem is not required, it offers an opportunity to earn extra credit and demonstrate mastery of the course material beyond the basic requirements.

Turn-in instructions

General instructions:

When implementing code in the labs, please maintain separate versions/copies of code so that mistakes such as unintentional overwriting or deletion of code is prevented. This is in addition to the efficiency that such organization provides. You may use any number of version control systems such as GIT and RCS. Please make sure that your code is protected from public access. For example, when using GIT, use git that manages code locally instead of its on-line counterpart github. If you prefer not to use version control tools, you may just use manual copy to keep track of different versions required for development and testing. More vigilance and discipline may be required when doing so.

The TAs, when evaluating your code, will use their own test code (replacing your main.c and main()) to drive your XINU code. The code you put inside main() is for your own testing and will, in general, not be considered during evaluation.

Electronic turn-in instructions:

i) Please make sure to follow the file and function naming convention as mentioned in the lab handout.

ii) Go to the xinu-spring2024/compile directory and run "make clean".

iii) Go to the directory where lab5 (containing xinu-spring2024/) is a subdirectory.

For example, if /homes/alice/cs354/lab5/xinu-spring2024 is your directory structure, go to /homes/alice/cs354

iv) Type the following command

```
turnin -c cs354 -p lab5 lab5
```

You can check/list the submitted files using

```
turnin -c cs354 -p lab5 -v
```

Please make sure to disable all debugging output before submitting your code.