Assignment Report: Title

Zheng li - 120090155

1 Introduction [2']

This assignment uses xv6, a simple and Unix-like teaching operating system, as the platform to implement:

1. The function `bmap` with double and triple indirect blocks.
2. The function `itrunc` with double and triple indirect blocks.
3. The function `sys_symlink` which is called behind the `symlink(target, path)` system call, to create a new symbolic link file at the path that refers to the target.
4. Modification of `sys_open` in order to support opening a symbolic link file. If the linked file is also a symbolic link, recursively follow it until a non-link file is reached or return `-1` to indicate that the depth of links reaches some threshold.

2 Design [5']

The prework: the macro and data structure in `file.h` and `fs.h` doesn't support double and triple indirect blocks, so I created similar files which are `file_ec.h` and `fs_ec.h` with macro and struct `dinode` and `inode` changed to support double and triple indirect blocks in `fs_ec.c`. To be specific: The first 10 elements of `ip->addrs[]` should be direct blocks. The 11th element should be a singly-indirect block. The 12th element should be a doubly-indirect block. The 13th element should be a triple-indirect block. ($256 \times 256 \times 256 + 256 \times 256 + 256 + 10 \approx 1.6$ million + blocks)

2.1 For the 1. the function `bmap` with double and triple indirect blocks:

I implement the `bmap` supporting double indirect blocks in `fs.c` and I implement the `bmap` supporting both double and triple indirect blocks in `fs_ec.c`. I pick the design in `fs_ec.c` to be illustrated because it includes all the implementation of both double and triple indirect blocks.

My data structure: My data structure is a hierarchical block-based file system that utilizes multiple levels of indirect blocks (single, double, and triple) to manage large files.

Direct entry: The first 10 entries (`ip->addrs[0]` to `ip->addrs[9]`) are direct entry. Each directly points to a data block (Layer 0).

Single Indirect entry: The 11th entry (`ip->addrs[10]`) is a single indirect entry, points to a block holding 256 direct entry (Layer 1), each direct entry points to a data block, forming a entirety of 256 data blocks (Layer 0).

Double Indirect entry: The 12th entry (`ip->addrs[11]`) is a double indirect entry, points to a block holding 256 single indirect entry (Layer 2), each indirect entry points to a block, forming a entriety of 256 blocks (Layer 1), each direct entry in each Layer 1 block points to a data block, forming a entirety of 256*256 data blocks (Layer 0).

Triple Indirect entry: The 13th entry (`ip->addrs[12]`) is a triple indirect entry, points to a block holding 256 double indirect entry (Layer 3), each double indirect entry points to a block, forming a entriety of 256 blocks (Layer 2), each single indirect entry in each Layer 2 block points to a block,

forming a entirety of 256*256 blocks (Layer 1), each direct entry in each Layer 1 block points to a data block, forming a entirety of 256*256*256 data blocks (Layer 0).

Insight: take triple as example, the data structure is like a tree rooted from the `ip->addrs[12]`!

Steps of bmap Function: bmap maps a logical block number `bn` (a block within the file) to the physical address of a block on disk. It handles allocation when necessary by calling `balloc` to allocate new blocks if they are not yet allocated.

1. **Direct Blocks Handling:** if `bn` <`NDIRECT`, the function directly uses the `addrs[]` array in the inode (`ip->addrs[bn]`) to point to a data block. If inside the entry is 0 (i.e., the data block has not been allocated yet), it calls `balloc` to allocate a layer0 block, i.e., a data block, and `return` the address of the data block.
2. **Single Indirect Block Handling:** if `NDIRECT` <`bn` <`NDIRECT+NINDIRECT`, the function uses the single indirect entry, the 11th entry (`ip->addrs[10]`). If inside the entry is 0, it calls `balloc` to allocate a layer1 block. Then, it retrieves the related entry of `bn` in this block. If inside the entry is 0, it calls `balloc` to allocate a layer0 block, i.e., a data block, and returns the address of the data block.
3. **Double Indirect Block Handling:** if `NDIRECT+NINDIRECT` <`bn` <`NDIRECT+NINDIRECT+DNIDIRECT`, the function uses the double indirect entry, the 12th entry (`ip->addrs[11]`). If inside the entry is 0, it calls `balloc` to allocate a layer2 block. Then, it retrieves the related entry of `bn` in this block. If inside the entry is 0, it calls `balloc` to allocate a layer1 block. Then, it retrieves the related entry of `bn` in this block. If inside the entry is 0, it calls `balloc` to allocate a layer0 block, i.e., a data block, and returns the address of the data block.
4. **Triple Indirect Block Handling:** if `NDIRECT+NINDIRECT+DNINDIRECT` <`bn` <`NDIRECT+NINDIRECT+DNIDIRECT+TNINDIRECT`, the function uses the triple indirect entry, the 13th entry (`ip->addrs[12]`). If inside the entry is 0, it calls `balloc` to allocate a layer3 block. Then, it retrieves the related entry of `bn` in this block. If inside the entry is 0, it calls `balloc` to allocate a layer2 block. Then, it retrieves the related entry of `bn` in this block. If inside the entry is 0, it calls `balloc` to allocate a layer1 block. Then, it retrieves the related entry of `bn` in this block. If inside the entry is 0, it calls `balloc` to allocate a layer0 block, i.e., a data block, and returns the address of the data block.

2.2 For the 2. the function `itrunc` with double and triple indirect blocks:

I implement the `itrunc` supporting double indirect blocks in `fs.c` and I implement the `itrunc` supporting both double and triple indirect blocks in `fs_ec.c`. I pick the design in `fs_ec.c` to be illustrated because it includes all the implementation of both double and triple indirect blocks.

Steps of `itrunc` Function: `itrunc` frees the data blocks associated with the inode. It goes through all levels of the file system's block structure (direct, single indirect, double indirect, and triple indirect) and deallocates any blocks that the inode references.

1. **Direct Blocks Handling:** For `ip->addrs[0]` to `ip->addrs[9]` (direct entries). If a direct block is allocated, it calls `bfree(ip->dev, ip->addrs[i])` to free the block, then sets `ip->addrs[i] = 0`.
2. **Single Indirect Block / Double Indirect Block Handling / Triple Indirect Block Handling:** They are corresponding to `ip->addrs[10]` to `ip->addrs[12]`. Their logic is similar, so I just pick triple indirect handling to illustrate. The `bfree` step is first to free the three-layer left bottom tree (root is an entry of a layer 2 block), then free a one-layer large tree, then to the whole tree.

2.3 For the 3. the function `sys_symlink`:

The `sys_symlink()` function is a system call that creates a symbolic link in a filesystem. Here's an explanation of its design and flow:

1. **Input Parameters (Arguments)**: This function does not directly take arguments in its parameter list, but it retrieves them through the `argstr()` function calls. **Target**: The first

argument represents the target of the symbolic link (the destination file or directory). **Path**: The second argument represents the path where the symbolic link will be created.

2. **Argument Validation**: After fetching the arguments, the function checks if either of the arguments is invalid: `if (argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)`. If either argument cannot be fetched properly, the function returns -1, indicating failure. This prevents further execution with invalid or undefined parameters.
3. **Begin File Operation**: The `begin_op()` function is called to start the transaction, signaling that a filesystem operation is about to take place.
4. **Check If the Path Already Exists**: The function checks whether a file already exists at the given path (`path`) by calling `namei(path)`. If a file exists at the path, `namei()` will return a non-zero pointer (`ip`), indicating the file was found. If a file exists, it calls `end_op()` to finish the transaction and returns -1 because you cannot create a symbolic link at an existing path.
5. **Create the Symbolic Link**: If no existing file is found, the function proceeds to create a new symbolic link at the provided path. It does so by calling the `create()` function. The `create()` function attempts to create a new file (in this case, a symbolic link, represented by the type `T_SYMLINK`). If the creation fails, `ip` will be 0, and the function calls `end_op()` to finish the transaction and returns -1 because the symbolic link could not be created.
6. **Write the Target Path into the Symbolic Link**: Once the symbolic link inode is created, the target path (i.e., the file or directory being linked to) is written into the new symbolic link file: `if (writei(ip, 0, (uint64)target, 0, MAXPATH) < MAXPATH)`. The `writei()` function writes the target string into the symbolic link inode `ip`. If the number of bytes written is less than `MAXPATH`, it indicates an error occurred during the write operation. If so, the function:
   - Unlocks the inode (`iunlock(ip)`),
   - Decrements the reference count of the inode (`iput(ip)`),
   - Calls `end_op()` to end the transaction,
   - Returns -1 to indicate failure.
7. **Final Cleanup and End Operation**: If the write operation is successful, the function:
   - Unlocks the inode (`iunlock(ip)`),
   - Decrements the reference count of the inode (`iput(ip)`),
   - Calls `end_op()` to finalize the filesystem operation.

   Finally, the function returns 0, indicating that the symbolic link was successfully created.

2.4 For the 4. modification `sys_open`:

The part I modified within the loop structure: `if (ip->type == T_SYMLINK && (omode & O_NOFOLLOW) == 0) {}`

1. **Checking whether the inode `ip` is a symbolic link (`T_SYMLINK`) and if the `O_NOFOLLOW` flag is not set in the file open mode (`omode`)**
2. **Depth Limit for Symbolic Link Traversal:** To prevent infinite loops due to circular symbolic links (where a symlink points to another symlink that eventually points back to itself), a depth limit is set: `int depth = 10; // Set depth`
3. **Traverse Symbolic Links (Main Loop):** The main part of the code is a loop that attempts to follow symbolic links up to the specified depth (`depth = 10`): `for(int i = 0; i < depth; i++) {}` Inside the loop, each iteration represents one level of symbolic link traversal. The process is as follows:
4. **Reading the Target of the Symbolic Link:** Inside the loop, the function reads the target of the symbolic link (i.e., the file the symbolic link points to):
   - `memset(destination, 0, MAXPATH)`: clears the destination buffer (which will hold the path to the target of the symlink) to ensure there is no leftover data.

- `readi(ip, 0, (uint64)destination, 0, MAXPATH)`: reads the target path from the symbolic link inode `ip` into the destination buffer (not buffer cache, just kernel buffer). It uses `readi()` to read `MAXPATH` bytes from the inode. If `readi()` does not return `MAXPATH` bytes (i.e., if the read fails or the length of the target path is not correct), the function returns -1, indicating an error.

5. **Unlock Current Inode:** After reading the target path, the function unlocks and releases the current inode (`ip`) because it's about to switch to a new inode. This ensures that the current inode is not locked while the function switches to the new target inode.
6. **Resolving the Target Path:** The next step is to resolve the target path stored in `destination`: `if ((ip = namei(destination)) == 0)`
   - `namei(destination)`: This function searches the filesystem for the file or directory specified by `destination` and returns the corresponding inode (`ip`). If `namei()` returns 0, it means the target does not exist or there is an error in resolving the path, and the function returns -1.
7. **Lock the New Inode:** If the target path is successfully resolved and a new inode is found, the function locks the new inode (`ip`) to perform further operations on it.
8. **Handling the New Inode:** Once the new inode is locked, the function checks if it is still a symbolic link. If the new inode is not a symbolic link, it breaks out of the loop, as further traversal is unnecessary. This means the function has reached the final target (a regular file, directory, etc.), and no further symbolic link resolution is needed.
9. **Depth Limit Exceeded:** If the loop reaches the maximum depth (10), meaning that there are too many symbolic links, the function will stop. It releases the current inode (`ip`), calls `end_op()` to complete the filesystem operation, and returns -1, indicating an error.
10. **Reference Count Update:** If everything works fine and no errors occur, the function increments the reference count of the final inode (`ip`) because the current thread is holding this inode.

3 Environment and Execution [2']

This assignment uses xv6, a simple and Unix-like teaching operating system. You can test the correctness of your code using the following commands under the `/xv6-labs-2022` directory.

3.1 To Run Part 1 and Part 2:

Use the following commands:

```
make clean
make qemu
bigfile
```

3.2 To Run the Bonus Part:

To run the bonus part, replace the following files:

- `file.h`, `fs.c`, and `fs.h` with `file_ec.h`, `fs_ec.h`, and `fs_ec.c`, respectively.
- Change their names back to `file.h`, `fs.c`, and `fs.h`.

Then, run the same commands as for Part 1 and Part 2:

```
make clean
make qemu
bigfile
```

3.3 To Run Part 3 and Part 4:

Use the following commands:

```
make clean
make qemu
symlinktest
```

4 Conclusion [2']

Through these tasks, I gained a deeper understanding of how file systems work at a low level, particularly how files are managed using direct, indirect, double indirect, and triple indirect blocks. I also gained experience in handling symbolic links, including creating, resolving, and limiting the depth of symbolic link traversals.

I also understand how file system is physically made up on linux OS and how operating system treat them. Operating system treat FS system in a way of multiple layer from the upmost file descriptor layer to the downmost disk layer.

I get an idea of what exactly will happen when an user program call read, write and open. how these system call interact with underlying kernel function such as filewrite and writei. I learn the relationship between fd and struct file and inode. I have a thorough understanding of loggin layer and cache layer and dive into a lof of funcions in fs.c log.c and bio.c.

The assignment also allowed me to enhance my skills in kernel programming, working with system calls, file systems, and memory management in the context of a minimal Unix-like environment. Debugging and testing the code in the xv6 environment helped reinforce my understanding of system-level programming and provided hands-on experience with concepts like inode management, block allocation, and error handling.

Overall, this assignment improved my knowledge of file system internals and symbolic link management, as well as my ability to work within a Unix-like operating system to modify and extend its functionality.