Assignment Report: File System

Meng Ziqi - 120020153

1 Introduction [2']

This assignment involves modifying the file system to enhance its functionality. Specifically, the task focuses on implementing support for doubly-indirect blocks for large file management (Task 1) and handling symbolic links (Task 2).

2 Design [5']

2.1 task 1

To support large files, the inode structure was modified, and critical file system functions like 'bmap' and 'itrunc' were enhanced. The design changes aimed to increase the maximum file size from 268 blocks to 65,803 blocks.

Inode Structure Modifications: Following the instruction from assignment description, on-disk `dinode` and the in-memory `inode` structures were modified. `NDIRECT` was reduced from 12 to 11, freeing space to store the address of a doubly-indirect block. This doubly-indirect block points to singly-indirect blocks, which in turn point to data blocks. The `addrs` array has been updated accordingly, using `NDIRECT + 2` to include the singly-indirect and doubly-indirect blocks.

bmap() Function: Previously, `bmap` only handled direct and singly-indirect blocks. The newly added logic introduces an additional layer to resolve block addresses when the logical block number exceeds the range of direct and singly-indirect blocks:

1. If the logical block number exceeds the range of direct and singly-indirect blocks, it is adjusted by subtracting the number of singly-indirect blocks (`NINDIRECT`). The remaining logical block number now corresponds to the doubly-indirect block range.
2. The doubly-indirect block address is retrieved from the inode's `addrs[NDIRECT+1]`. If this address is unallocated (i.e., zero), a new doubly-indirect block is allocated, and its address is stored in the inode.
3. The doubly-indirect block is read into memory, and its contents are interpreted as an array of addresses. The index of the singly-indirect block within the doubly-indirect block is calculated as `bn / NINDIRECT`. If the address for the desired singly-indirect block is unallocated, a new block is allocated, and its address is stored in the doubly-indirect block.
4. The singly-indirect block is then read into memory, and its contents are interpreted as an array of addresses. The specific data block within the singly-indirect block is identified by calculating `bn % NINDIRECT`. If the address for the desired data block is unallocated, a new block is allocated, and its address is stored in the singly-indirect block.
5. Once the appropriate data block address is resolved (and allocated if necessary), the function returns this address to the caller.

The addition of doubly-indirect blocks allows the file system to support files requiring up to 65,803 blocks, compared to the previous limit of 268 blocks.

itrunc() Function: The `itrunc` function was enhanced to support the release of doubly-indirect blocks, allowing the file system to properly clean up resources associated with larger files:

1. The function first checks if the inode has a doubly-indirect block by examining the address stored in `ip->addrs[NDIRECT+1]`. If it exists, the doubly-indirect block is read into memory for further processing.
2. The doubly-indirect block contains addresses pointing to singly-indirect blocks. For each non-null entry in the doubly-indirect block, similar to previous code, the corresponding singly-indirect block is loaded into memory.
3. Similar to previous code, for every non-null entry in a singly-indirect block, the corresponding data block is freed using the `bfree` function.
4. After processing all data blocks within a singly-indirect block, the singly-indirect block itself is freed using `bfree`, ensuring all associated resources are released.
5. After all singly-indirect blocks have been processed and freed, the doubly-indirect block itself is freed, and the inode's doubly-indirect block pointer (`ip->addrs[NDIRECT+1]`) is set to zero.

These modifications allow the file system to properly release resources associated with files that use doubly-indirect blocks.

2.2 Task 2:

The second task involved implementing symbolic links (symlinks) in the file system. Symbolic links are special file types that store the path of another file or directory, allowing users to create references to other resources. The implementation required creating a new system call, `sys_symlink`, and enhancing the behavior of the `sys_open` function to handle symlinks.

sys_symlink System Call: The `sys_symlink` function was implemented to create symbolic links:

1. The input parameters `link_target` and `link_path` are validated to ensure they are valid paths and within the maximum allowable path length.
2. The `create` function is called with the `T_SYMLINK` type to allocate an inode for the symbolic link. The new inode is initialized to represent a symlink.
3. The `writei` function is used to store the target path (`link_target`) as the content of the newly created symbolic link file. This allows the symlink to store the reference to the actual target.
4. If the target path cannot be written or if inode creation fails, the function handles these errors gracefully by releasing resources and returning an appropriate error code.
5. The function returns 0 on successful creation of the symlink and -1 in case of any failure.

sys_open Function: The `sys_open` function was enhanced to support symbolic link resolution. Specifically, it was modified to handle the following cases:

- If the file being opened is a symbolic link and the `O_NOFOLLOW` flag is not set, the function recursively resolves the symlink until the actual file or directory is reached.
- If the `O_NOFOLLOW` flag is set, the symlink itself is opened, and no further resolution is performed.
- To prevent infinite loops caused by cyclic symbolic links, a depth limit of 10 was introduced. If this limit is reached, the function returns an error to indicate the cycle.

The recursive resolution logic involves:

1. Reading the target path of the symlink using `readi`.
2. Resolving the target path to a new inode using `namei`.
3. Repeating this process until a non-symlink file is reached or the depth limit is exceeded.

This logic ensures that symbolic links are resolved correctly and efficiently, while avoiding potential issues like infinite recursion.

3 Environment and Execution [2']

3.1 Environment

The implementation and testing of this assignment were conducted in the following environment:

- **Operating System:** Ubuntu 20.04 running on a virtual machine.
- **Kernel Version:** Linux version 5.4.0, built with GCC 9.4.0.
- **Development Tools:**
  - `make`: For building the xv6 kernel and managing dependencies.
  - `qemu`: A hardware emulator to run and test the modified xv6 kernel.
- **Editor:** VS Code with SSH extensions for remote development and debugging.

3.2 Execution

The following commands were used to clean, compile, and execute the xv6 kernel:

```
make clean
make qemu
```

To verify the implementation of doubly-indirect blocks, the test case was run using the command: `bigfile`. And the output is as follows:

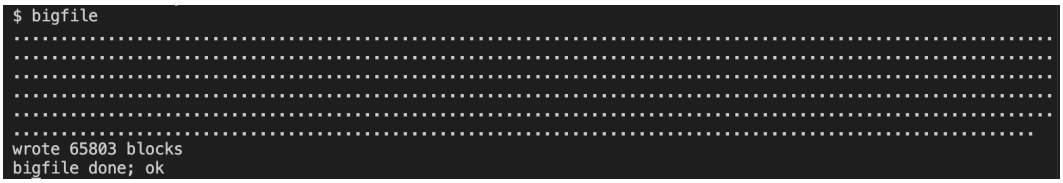

Figure 1: Task 1 Test Result

To verify the implementation of symbolic links, the following command was run: `symlinktest`. The output is:

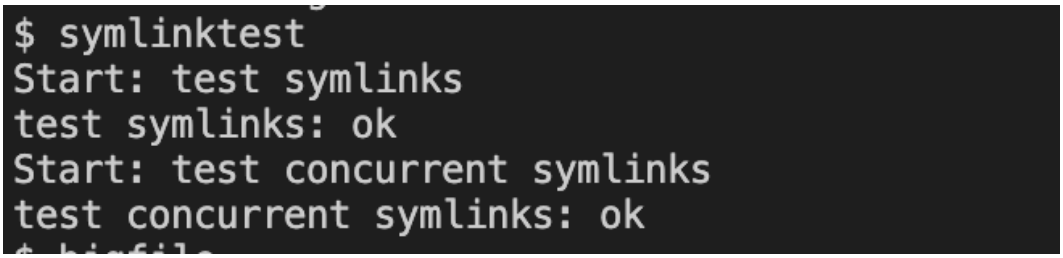

Figure 2: Task 2 Test Result

The successful execution of the test cases demonstrates that the modified xv6 kernel supports the newly added functions.

4 Conclusion [2']

This assignment provided a comprehensive understanding of file system design and implementation. Through Task 1, I learned how to extend file system capabilities by implementing doubly-indirect blocks, enabling support for significantly larger files. Task 2 introduced symbolic links, which required careful design to handle recursive resolution and edge cases.