OS 344 - Lab Assignment 5

Instructions

- This assignment must be completed in a group.
- Group members should ensure that one member submits the completed assignment before the **deadline**.
- Each group should submit a **report**, with relevant screenshots, necessary data, and findings related to the assignments.
- We expect a sincere and fair effort from your side. All submissions will be checked for **plagiarism** through software, and plagiarised submissions will be penalised heavily, irrespective of the source and copy.
- There will be a **viva associated with the assignment**. Attendance of all group members is mandatory.
- **Assignment code, report, and viva all will be considered for grading.**
- An early start is recommended; any extension to complete the assignment will not be given.
- If any required information is missing, **reasonable assumptions may be made**. All such assumptions must be mentioned in the report to ensure transparency and consistency.
- Before you start coding, read **Chapter 8** of the xv6 book, paying close attention to the File System implementation and inode management.

Task 5.1: Implement Doubly-Indirect Block Support

In this task, you will modify the xv6 file system to **support larger files** using a **doubly-indirect block pointer**. This extends the maximum file size beyond what is currently supported by xv6.

Objective:

The xv6 file system supports **12 direct data block addresses and one single indirect address per inode.** This limits the maximum file size to approximately 268 KB. Your goal is to **extend the file system to allow much larger files by adding a doubly-indirect pointer**.

1. Modify inode Structure

In **fs.h**, modify the inode layout as follows:

- Reduce the number of direct block pointers (e.g., from 12 to 10 or 11).
- Retain one entry for a single indirect block.
- Add one new entry for a doubly-indirect block.

The new inode structure should still fit within the 128-byte size limit.

2. Update Block Mapping (bmap)

Modify **bmap()** in **fs.c** to support doubly-indirect addressing:

- If the logical block number exceeds the single indirect range, traverse the doubly-indirect pointer
- Allocate new indirect blocks as needed.

Each doubly-indirect block should contain 256 pointers to indirect blocks, each of which points to 256 data blocks.

3. Update Block Deallocation (itrunc)

- Extend **itrunc()** to recursively free all blocks associated with the doubly-indirect pointer.
- Ensure that data blocks, indirect blocks, and doubly-indirect blocks are all freed properly.

4. Testing

Create a user program **bigfile.c** to verify your implementation.

- The program should create a large file that exceeds the previous xv6 limit.
- Write data sequentially and verify integrity by reading it back.

Deliverables:

1. Modified kernel files (**fs.c, file.c, mkfs.c, fs.h**).
2. The test program **bigfile.c**.
3. Mention in the report:
    - Explanation of how doubly-indirect addressing is implemented.
    - The new theoretical maximum file size.
    - Screenshots of successful compilation and execution.

Expected Outcome:

Your xv6 file system should now support much larger files and correctly handle both reading and writing beyond the single indirect block limit.

Task 5.2: Implement Symbolic Links (Soft Links)

In this task, you will extend the xv6 file system to support symbolic (soft) links. Symbolic links are special files that point to other files or directories.

Objective:

The goal is to implement a new system call **symlink(target, path)** that creates a symbolic link at path referring to target. When opening a symbolic link, xv6 should resolve the link and open the target file transparently.

1. Define a New File Type

- Add a new file type **T_SYMLINK** in **stat.h**.
- Update **mkfs.c** and related file system components to recognize symbolic links.

2. Implement symlink System Call

Create a new system call int **symlink(const char *target, const char *path)**.

- It should create a new inode of type **T_SYMLINK**.
- Store the target path as data within the symbolic link file.
- Return **0** on success and **-1** on failure.

3. Modify open() to Handle Symbolic Links

Update the **open()** implementation in **sysfile.c**:

- When opening a symbolic link, read the stored target path and resolve it recursively.
- Limit recursion depth to **10** to prevent infinite loops.
- Support **O_NOFOLLOW** flag to open the symlink itself rather than the target.

4. Additional Considerations

- Ensure that **stat()** on a symbolic link returns its own information, not that of the target.
- Ensure **unlink()** removes the symbolic link, not the target.
  Optional: Support symbolic links to directories.

5. Testing

Implement a test program **symlinktest.c** to validate your implementation.

- Create symbolic links to files and directories.
- Test recursive links and detect loops.
- Test **O_NOFOLLOW** behavior.

Deliverables:

1. Modified kernel files (**sysfile.c, fs.c, file.c, syscall.c, syscall.h, user.h**).
2. Test program **symlinktest.c**.
3. Include in your report:
   - Explanation of the symlink mechanism.
   - Recursive resolution handling and depth limit.
   - Screenshots of successful execution.

Expected Outcome:

Your implementation should allow users to create, resolve, and manage symbolic links. Programs using symbolic links should behave as expected when reading, writing, or executing linked files.