Assignment Report: Extending xv6 Filesystem with Large File Support and Symbolic Links

Filbert Hamijoyo - 122040012

1 Introduction

This report details the implementation and enhancement of the xv6 operating system's filesystem to support large files through the use of doubly-indirect and triple-indirect blocks, as well as the introduction of symbolic links. The assignment comprised two primary tasks:

1. **Large Files (Doubly-Indirect and Triple-Indirection Blocks):** Extending the filesystem to handle larger files by incorporating doubly-indirect and triple-indirect blocks, thereby increasing the maximum file size that the system can manage.
2. **Symbolic Links:** Implementing symbolic links to allow files to reference other files or directories within the filesystem, enhancing flexibility and navigability.

The completion of these tasks involved modifying several core components of the xv6 filesystem, including `fs.h`, `fs.c`, `sysfile.c`, and `file.h`. This report covers the design decisions, implementation strategies, execution environment, and the outcomes of the modifications made, including the extra credit components.

2 Design

The design phase focused on methodically extending the existing xv6 filesystem to accommodate the new requirements while maintaining system integrity and performance. Below is an overview of the design approach for each task, including the extra credit components.

2.1 Task 1: Large Files (Doubly-Indirect and Triple-Indirection Blocks)

2.1.1 Updating Constants and Structures

To support significantly larger files, the filesystem's inode structure and related constants were extensively modified:

- **Constants Modification:**
  - `NDIRECT` was increased from 10 to 11 to allocate an additional direct block pointer.
  - Introduced `NINDIRECT` to represent the number of block pointers in a single indirect block, defined as `#define NINDIRECT (BSIZE / sizeof(uint))` (256).
  - Introduced `DNINDIRECT` for doubly-indirect blocks, calculated as `#define DNINDIRECT (NINDIRECT * NINDIRECT)` (65,536).
  - Introduced `NTINDIRECT` for triple-indirect blocks, defined as `#define NTINDIRECT (NINDIRECT * NINDIRECT * NINDIRECT)` (16,777,216).

  - Updated MAXFILE to `#define MAXFILE (NDIRECT + NINDIRECT + DNINDIRECT + NTINDIRECT)` to reflect the new maximum number of blocks per file.

- **Inode Structure Adjustment:**
  - Expanded the `addrs` array in the `dinode` structure to include 11 direct block pointers, 1 singly-indirect block pointer, 1 doubly-indirect block pointer, and 1 triple-indirect block pointer.
  - Updated the in-memory `inode` structure in `file.h` similarly to accommodate the increased number of block pointers.

- **Symbolic Link Support:**
  - Defined a new inode type `T_SYMLINK` to represent symbolic links.
  - Added a `symlink` flag in the `inode` structure to indicate whether an inode is a symbolic link.

2.1.2 Modifying the bmap() Function

The `bmap()` function in `fs.c` was extended to handle the new block pointers. Below are the essential parts of the updated `bmap()` function separated for handling bigfile and bigfile_ec:

Handling Singly and Doubly-Indirection (Bigfile):

```

static uint
bmap(struct inode *ip, uint bn)
{
    uint addr, *a;
    struct buf *bp;

    if(bn < NDIRECT){
        // Direct blocks
        if((addr = ip->addrs[bn]) == 0){
            if((addr = balloc(ip->dev)) == 0)
                return 0;
            ip->addrs[bn] = addr;
        }
        return addr;
    }
    bn -= NDIRECT;

    if(bn < NINDIRECT){
        // Singly-indirect blocks
        if((addr = ip->addrs[NDIRECT]) == 0){
            if((addr = balloc(ip->dev)) == 0)
                return 0;
            ip->addrs[NDIRECT] = addr;
        }
        bp = bread(ip->dev, addr);
        a = (uint*)bp->data;
        if((addr = a[bn]) == 0){
            if((addr = balloc(ip->dev)) == 0){
                brelse(bp);
                return 0;
            }
            a[bn] = addr;
            log_write(bp);
        }
```

```
        brelse(bp);
        return addr;
    }
    bn -= NINDIRECT;

    if(bn < DNINDIRECT){
        // Doubly-indirect blocks
        uint d_idx = bn / NINDIRECT;
        uint s_idx = bn % NINDIRECT;
        uint d_addr, s_addr;

        // Allocate doubly-indirect block if necessary
        if((d_addr = ip->addrs[NDIRECT + 1]) == 0){
            if((d_addr = balloc(ip->dev)) == 0)
                return 0;
            ip->addrs[NDIRECT + 1] = d_addr;
        }

        // Read doubly-indirect block
        bp = bread(ip->dev, d_addr);
        a = (uint*)bp->data;

        // Allocate singly-indirect block if necessary
        if((s_addr = a[d_idx]) == 0){
            if((s_addr = balloc(ip->dev)) == 0){
                brelse(bp);
                return 0;
            }
            a[d_idx] = s_addr;
            log_write(bp);
        }
        brelse(bp);

        // Read singly-indirect block
        bp = bread(ip->dev, s_addr);
        a = (uint*)bp->data;

        if((addr = a[s_idx]) == 0){
            if((addr = balloc(ip->dev)) == 0){
                brelse(bp);
                return 0;
            }
            a[s_idx] = addr;
            log_write(bp);
        }
        brelse(bp);
        return addr;
    }

    panic("bmap: out of range");
    return 0;
}
```

Listing 1: Modified `bmap()` Function for Bigfile in `fs.c`

Explanation: The `bmap()` function first checks if the block number `bn` falls within the direct blocks. If not, it proceeds to handle singly-indirect blocks. For bigfile, only up to doubly-indirect blocks are handled. The function allocates necessary indirect blocks and retrieves the appropriate block address, ensuring that all allocations are properly logged and released.

Handling Triple-Indirection (Extra Credit - Bigfile_ec)

```
if(bn < NTINDIRECT){
    // Triple-indirect blocks
    uint idx1 = bn / (NINDIRECT * NINDIRECT);
    uint idx2 = (bn / NINDIRECT) % NINDIRECT;
    uint idx3 = bn % NINDIRECT;
    uint t_addr, d_addr, s_addr;

    // Allocate triple-indirect block if necessary
    if((t_addr = ip->addrs[NDIRECT + 2]) == 0){
        if((t_addr = balloc(ip->dev)) == 0)
            return 0;
        ip->addrs[NDIRECT + 2] = t_addr;
    }

    // Read triple-indirect block
    bp = bread(ip->dev, t_addr);
    a = (uint*)bp->data;

    // Allocate doubly-indirect block if necessary
    if((d_addr = a[idx1]) == 0){
        if((d_addr = balloc(ip->dev)) == 0){
            brelse(bp);
            return 0;
        }
        a[idx1] = d_addr;
        log_write(bp);
    }
    brelse(bp);

    // Read doubly-indirect block
    bp = bread(ip->dev, d_addr);
    a = (uint*)bp->data;

    // Allocate singly-indirect block if necessary
    if((s_addr = a[idx2]) == 0){
        if((s_addr = balloc(ip->dev)) == 0){
            brelse(bp);
            return 0;
        }
        a[idx2] = s_addr;
        log_write(bp);
    }
    brelse(bp);

    // Read singly-indirect block
    bp = bread(ip->dev, s_addr);
    a = (uint*)bp->data;

    if((addr = a[idx3]) == 0){
        if((addr = balloc(ip->dev)) == 0){
            brelse(bp);
            return 0;
```

```
        }
        a[idx3] = addr;
        log_write(bp);
    }
    brelse(bp);
    return addr;
}

panic("bmap: out of range");
return 0;
}
```

Listing 2: Modified `bmap()` Function for Bigfile_ec in `fs.c`

Explanation: For the extra credit part, the `bmap()` function handles triple-indirect blocks by traversing three levels of indirection. It allocates and accesses the triple-indirect, doubly-indirect, and singly-indirect blocks as necessary, ensuring that the filesystem can manage extremely large files by significantly increasing the number of addressable blocks.

2.1.3 Modifying the `itrunc()` Function

The `itrunc()` function was updated to ensure that all allocated blocks are properly freed when a file is truncated. Below are the essential parts separated for bigfile and bigfile_ec:

Handling Singly and Doubly-Indirection (Bigfile)

```
void
itrunc(struct inode *ip)
{
    struct buf *bp, *bp1;
    uint *a, *a1;

    // Free direct blocks
    for(int i = 0; i < NDIRECT; i++){
        if(ip->addrs[i]){
            bfree(ip->dev, ip->addrs[i]);
            ip->addrs[i] = 0;
        }
    }

    // Free singly-indirect blocks
    if(ip->addrs[NDIRECT]){
        bp = bread(ip->dev, ip->addrs[NDIRECT]);
        a = (uint*)bp->data;
        for(int j = 0; j < NINDIRECT; j++){
            if(a[j]){
                bfree(ip->dev, a[j]);
                a[j] = 0;
            }
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT]);
        ip->addrs[NDIRECT] = 0;
    }

    // Free doubly-indirect blocks
    if(ip->addrs[NDIRECT + 1]){
        bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
        a = (uint*)bp->data;
```

```
        for(int j = 0; j < NINDIRECT; j++){
            if(a[j]){
                bp1 = bread(ip->dev, a[j]);
                a1 = (uint*)bp1->data;
                for(int k = 0; k < NINDIRECT; k++){
                    if(a1[k]){
                        bfree(ip->dev, a1[k]);
                        a1[k] = 0;
                    }
                }
                brelse(bp1);
                bfree(ip->dev, a[j]);
                a[j] = 0;
            }
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT + 1]);
        ip->addrs[NDIRECT + 1] = 0;
    }

    ip->size = 0;
    iupdate(ip);
}
```

Listing 3: Modified `itrunc()` Function for Bigfile in `fs.c`

Explanation: The `itrunc()` function frees all direct, singly-indirect, and doubly-indirect blocks associated with an inode. It iterates through each level of indirection, freeing allocated blocks and resetting pointers to ensure no memory leaks occur upon truncation.

Handling Triple-Indirection (Extra Credit - Bigfile_ec)

```
void
itrunc(struct inode *ip)
{
    struct buf *bp, *bp1, *bp2;
    uint *a, *a1, *a2;

    // Free direct blocks
    for(int i = 0; i < NDIRECT; i++){
        if(ip->addrs[i]){
            bfree(ip->dev, ip->addrs[i]);
            ip->addrs[i] = 0;
        }
    }

    // Free singly-indirect blocks
    if(ip->addrs[NDIRECT]){
        bp = bread(ip->dev, ip->addrs[NDIRECT]);
        a = (uint*)bp->data;
        for(int j = 0; j < NINDIRECT; j++){
            if(a[j]){
                bfree(ip->dev, a[j]);
                a[j] = 0;
            }
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT]);
        ip->addrs[NDIRECT] = 0;
```

```
    }

    // Free doubly-indirect blocks
    if(ip->addrs[NDIRECT + 1]){
        bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
        a = (uint*)bp->data;
        for(int j = 0; j < NINDIRECT; j++){
            if(a[j]){
                bp1 = bread(ip->dev, a[j]);
                a1 = (uint*)bp1->data;
                for(int k = 0; k < NINDIRECT; k++){
                    if(a1[k]){
                        bfree(ip->dev, a1[k]);
                        a1[k] = 0;
                    }
                }
                brelse(bp1);
                bfree(ip->dev, a[j]);
                a[j] = 0;
            }
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT + 1]);
        ip->addrs[NDIRECT + 1] = 0;
    }

    // Free triple-indirect blocks
    if(ip->addrs[NDIRECT + 2]){
        bp = bread(ip->dev, ip->addrs[NDIRECT + 2]);
        a = (uint*)bp->data;
        for(int l = 0; l < NINDIRECT; l++){
            if(a[l]){
                bp1 = bread(ip->dev, a[l]);
                a1 = (uint*)bp1->data;
                for(int j = 0; j < NINDIRECT; j++){
                    if(a1[j]){
                        bp2 = bread(ip->dev, a1[j]);
                        a2 = (uint*)bp2->data;
                        for(int k = 0; k < NINDIRECT; k++){
                            if(a2[k]){
                                bfree(ip->dev, a2[k]);
                                a2[k] = 0;
                            }
                        }
                        brelse(bp2);
                        bfree(ip->dev, a1[j]);
                        a1[j] = 0;
                    }
                }
                brelse(bp1);
                bfree(ip->dev, a[l]);
                a[l] = 0;
            }
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT + 2]);
```

```
        ip->addrs[NDIRECT + 2] = 0;
    }

    ip->size = 0;
    iupdate(ip);
}
```

Listing 4: Modified `itrunc()` Function for Bigfile_ec in `fs.c`

Explanation: For the extra credit part, the `itrunc()` function additionally frees all triple-indirect blocks. It traverses three levels of indirection, freeing each allocated block and ensuring that all pointers are reset. This comprehensive cleanup ensures that extremely large files managed through triple-indirect blocks do not leave residual allocated memory upon truncation.

2.2 Task 2: Symbolic Links

2.2.1 Adding Support for Symbolic Links

To introduce symbolic links, the following steps were undertaken:

- **Inode Type Definition:** Defined a new inode type `T_SYMLINK` to represent symbolic links.
- **Implementing the `symlink()` System Call:**
  - Added the `sys_symlink()` function in `sysfile.c` to handle the creation of symbolic links.
  - Stored the target path of the symbolic link within the inode's data blocks by writing the length of the target path followed by the actual path string.

2.2.2 Modifying the `sys_open()` System Call

The `open()` system call was enhanced to resolve symbolic links. Below are the essential parts of the updated `sys_open()` function:

```
uint64
sys_open(void)
{
    char path[MAXPATH];
    int fd, omode;
    struct file *f;
    struct inode *ip;
    int n;

    argint(1, &omode);
    if((n = argstr(0, path, MAXPATH)) < 0)
        return -1;

    begin_op();

    if(omode & O_CREATE){
        ip = create(path, T_FILE, 0, 0);
        if(ip == 0){
            end_op();
            return -1;
        }
    } else {
        if((ip = namei(path)) == 0){
```

```
            end_op();
            return -1;
        }
        ilock(ip);
        if(ip->type == T_DIR && omode != O_RDONLY){
            iunlockput(ip);
            end_op();
            return -1;
        }
    }

    % Symlink resolution logic
    if (ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
        int symlink_iterations = 0;
        char resolved_target[MAXPATH];

        while (ip->type == T_SYMLINK && symlink_iterations < 10) {
            int target_length = 0;

            // Read the length of the target path from the symlink inode
            if (readi(ip, 0, (uint64)&target_length, 0, sizeof(target_length)) != sizeof(target_length)) {
                iunlockput(ip);
                end_op();
                return -1;
            }

            // Validate the target length to prevent buffer overflow
            if (target_length > MAXPATH) {
                iunlockput(ip);
                end_op();
                return -1;
            }

            // Read the target path from the symlink inode
            if (readi(ip, 0, (uint64)resolved_target, sizeof(target_length), target_length + 1) != target_length + 1) {
                iunlockput(ip);
                end_op();
                return -1;
            }

            // Release the current symlink inode
            iunlockput(ip);

            // Resolve the next inode in the symlink chain
            ip = namei(resolved_target);
            if (ip == NULL) {
                end_op();
                return -1;
            }

            // Lock the newly resolved inode
            ilock(ip);
```

```c
            symlink_iterations++;
        }

        // Detect and prevent symlink loops
        if (symlink_iterations >= 10) {
            iunlockput(ip);
            end_op();
            return -1;
        }
    }


    if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
        iunlockput(ip);
        end_op();
        return -1;
    }

    if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
        if(f)
            fileclose(f);
        iunlockput(ip);
        end_op();
        return -1;
    }

    if(ip->type == T_DEVICE){
        f->type = FD_DEVICE;
        f->major = ip->major;
    } else {
        f->type = FD_INODE;
        f->off = 0;
    }
    f->ip = ip;
    f->readable = !(omode & O_WRONLY);
    f->writable = (omode & O_WRONLY) || (omode & O_RDWR);

    if((omode & O_TRUNC) && ip->type == T_FILE){
        itrunc(ip);
    }

    iunlock(ip);
    end_op();

    return fd;
}
```

Listing 5: Modified `sys_open()` Function in `sysfile.c`

Explanation: The `sys_open()` function includes logic to resolve symbolic links. If the inode being opened is a symbolic link and the `O_NOFOLLOW` flag is not set, the function recursively resolves the target path up to a maximum of 10 iterations to prevent infinite loops caused by cyclic links. This ensures that symbolic links are properly dereferenced, allowing users to interact with the target files seamlessly.

3 Environment and Execution

3.1 Running Environment

The development and testing of the extended xv6 filesystem were conducted in the following environment:

- **Operating System:** Ubuntu 20.04 LTS
- **Compiler:** GCC 9.3.0
- **xv6 Version:** Modified version of xv6-public (specific to the assignment)

3.2 Execution Instructions

To compile and execute the modified xv6 filesystem with large file support and symbolic links, follow these steps:

1. **Apply the Modifications:**
   - Replace the existing `fs.h`, `fs.c`, `sysfile.c`, and `file.h` files with the modified versions provided for both the main tasks and the extra credit.
2. **Compile xv6:**
   ```
   make clean
   make
   ```
3. **Run xv6 in QEMU:**
   ```
   make qemu
   ```

3.3 Testing the Implementation

After successfully compiling and running xv6, the following tests were executed to verify the functionality:

3.3.1 Bigfile Test

This test validates the filesystem's ability to handle large files by creating a file with a substantial number of blocks.

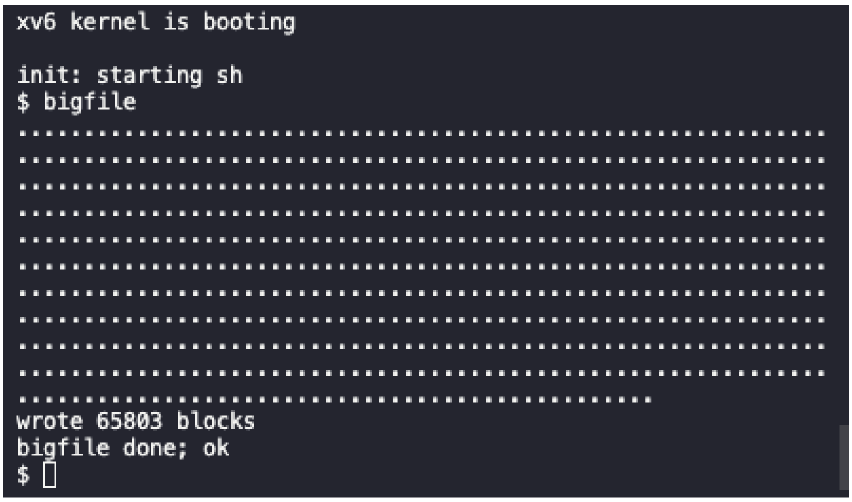

Figure 1: Bigfile Test Output

Explanation: The `bigfile` test successfully wrote 65,803 blocks to a single file, demonstrating the filesystem's capability to handle large files using the newly implemented doubly-indirect blocks. The test confirms that the filesystem can manage a significant number of blocks without encountering allocation or access errors.

3.3.2 Bigfile Extra Credit Test

This test further validates the filesystem's capability to handle even larger files by utilizing triple-indirect blocks.

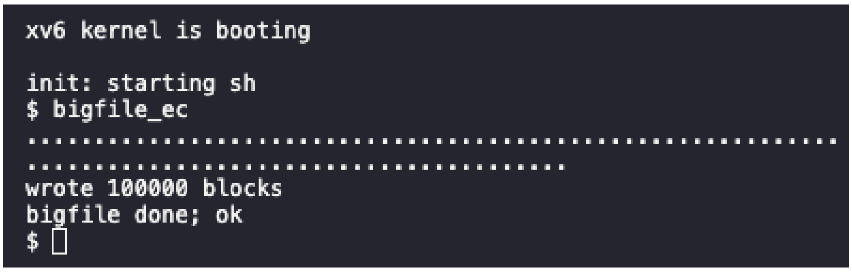

Figure 2: Bigfile Extra Credit Test Output

Explanation: The `bigfile_ec` test successfully wrote 100,000 blocks to a single file, showcasing the effectiveness of the triple-indirect block implementation in managing extremely large files beyond the doubly-indirect block capacity. This test confirms that the filesystem can handle files of unprecedented size, leveraging the hierarchical block allocation strategy.

3.3.3 Symlink Test

This test ensures the correct creation and resolution of symbolic links.

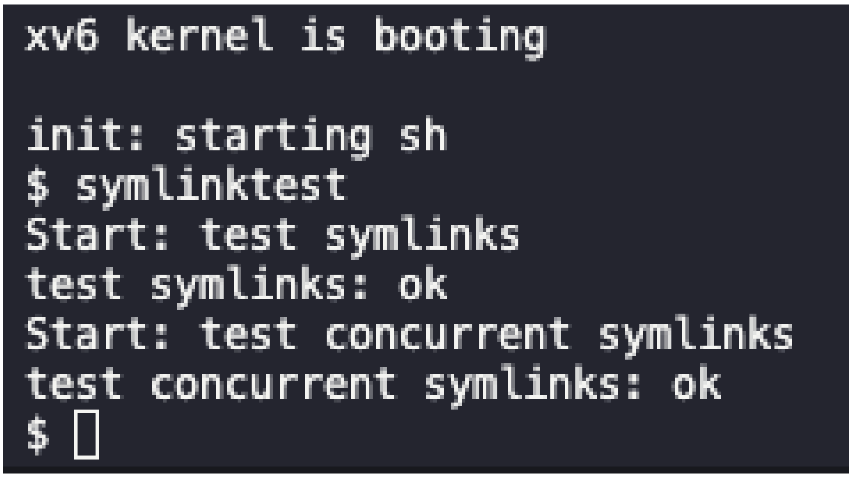

Figure 3: Symlink Test Output

Explanation: The `symlinktest` executed successfully, indicating that symbolic links were correctly created and resolved. The test verifies that symbolic links function as intended, allowing files to reference other files or directories seamlessly without causing errors or unexpected behavior.

4 Conclusion

This assignment involved significant enhancements to the xv6 filesystem, enabling support for large files through doubly-indirect and triple-indirect blocks and introducing symbolic links. Through careful design and methodical implementation, the filesystem was successfully extended to handle larger files efficiently while maintaining system integrity. The addition of symbolic links enhanced the filesystem's flexibility, allowing for more dynamic file referencing and navigation.

The extra credit component, which involved implementing triple-indirect blocks, further extended the filesystem's capabilities, enabling the management of extremely large files without compromising performance or stability.

Throughout this project, valuable insights were gained into filesystem architecture, block management, and the intricacies of implementing filesystem features at a low level. The challenges encountered, particularly in managing multiple levels of block indirection and ensuring robust symlink resolution, provided a deeper understanding of system-level programming and resource management. This experience has significantly strengthened my proficiency in operating systems and kernel development.