CS344 Assignment 5

Group-3, CSE

October 8, 2025

Team Members

- **230101024** (Baswa Dhanush Sai)
- **230101092** (Seella Mythresh Babu)
- **230101114** (Yeredla Koushik Reddy)
- **230101120** (Taes Padhihary)

Please find the Google Drive folder containing all the edited files here:

drive.google.com/drive/folders/106CkVtq2gCoXgV4hyZ2kui2GwBto0w_0?usp=sharing

Contents

---

1. Task 5.1: Doubly-Indirect Block Support

Reference used: xiayingp.gitbook.io/build_a_os/labs/lab-8-file-system-large-files

Synopsis of the problem Figure 1 shows the format of storing files in xv6 that is currently used. It supports $12 + 256 = 268$ blocks. This limits the total size of the files to around 268KiB. We need to increase this size.

1.1. Implementation

We need to support a "doubly-indirect" block in each inode, containing 256 addresses of singly indirect blocks, each of which can further contain upto 256 blocks. The new plan is shown in Figure 2. This increases the size to $11 + 256 + 256 \cdot 256 =$ **65803** blocks.

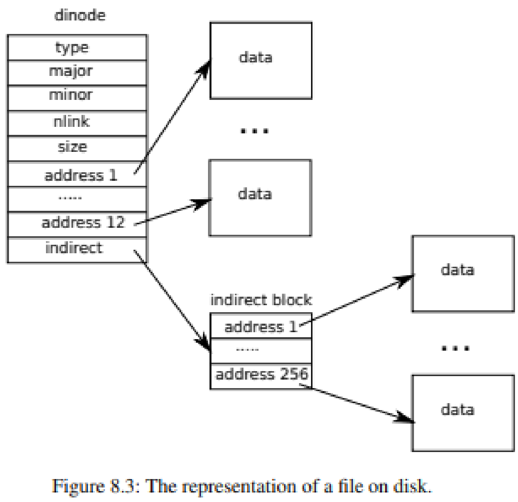

Figure 8.3: The representation of a file on disk.

Figure 1. Old file representation (source: xv6/book-riscv-rev4.pdf)

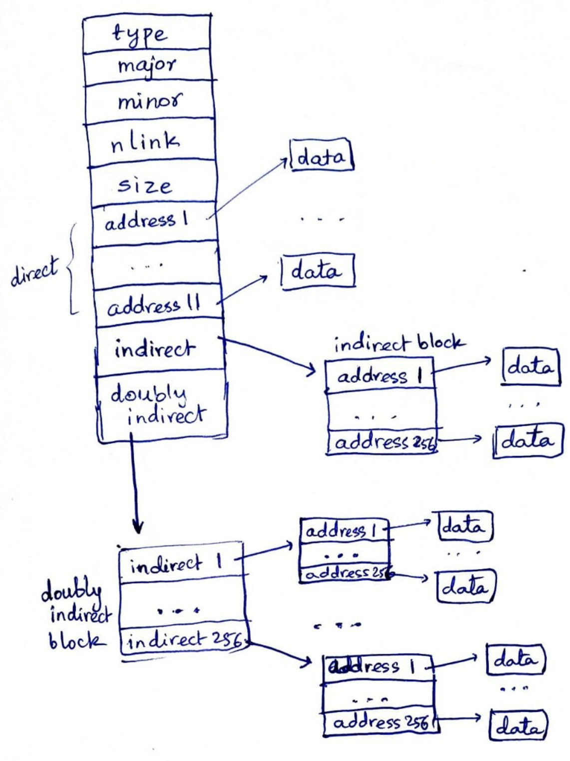

Figure 2. Implemented file representation

Mechanism

1. Update the FSSIZE macro in kernel/param.h to a large number like **200000** instead of 2000.
2. In kernel/fs.h, we update the value of NDIRECT to 11 (from the initial value of 12, because we should not change the disk addrs size), add the macro NDOUBLY_INDIRECT = NINDIRECT * NINDIRECT and update MAXFILE = NDIRECT + NINDIRECT + NDOUBLY_INDIRECT.
3. In kernel/file.h, change the size of addrs[] array of structure inode from NDIRECT+1 to NDIRECT+2 (addrs[NDIRECT] will have the indirect block and addrs[NDIRECT+1] will have the doubly-indirect block of Figure 2). This is the *in-memory* copy of the inode structure. Make the same changes to the *on-disk* copy by updating the size of addrs[] array to NDIRECT+2 in kernel/fs.h.
4. Finally, to implement doubly-ended indirection of files, we modify the functions bmap() and itrunc() of kernel/fs.c as shown in Listing 2.

Listing 1. changes in macros and structs

```
/* kernel/param.h */
#define FSSIZE      200000 // initially 2000, but
      now 200000

/* kernel/file.h */
// . . .
struct inode {
  // . . .
  uint addrs[NDIRECT+2];
};

/* kernel/fs.h */
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NDOUBLY_INDIRECT (NINDIRECT*
      NINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT
      + NDOUBLY_INDIRECT)

struct dinode {
  // . . .
  uint addrs[NDIRECT+2];
};
```

Listing 2. changes to kernel/fs.c for doubly-indirect block

```
/* kernel/fs.c */
static int bmap(struct inode *ip, uint bn){
  // . . .
  if (bn < NINDIRECT) { // calculate . . . }
  // Task 5.1: handle doubly indirect blocks
  bn -= NINDIRECT;
  if (bn < NDOUBLY_INDIRECT) {
    if ((addr = ip->addrs[NDIRECT+1]) == 0){
      addr = balloc(ip->dev);
      if (addr == 0) return 0;
      ip->addrs[NDIRECT+1] = addr;
    }
    bp = bread(ip->dev, ip->addrs[NDIRECT
      +1]);
    a = (uint*)bp->data;
    // now, go to the second layer
    uint outer_index = bn / NINDIRECT;
    if ((addr = a[outer_index]) == 0){
      addr = balloc(ip->dev);
      if (addr == 0){ brelse(bp); return 0; }
      a[outer_index] = addr; log_write(bp);
    }
    brelse(bp);

    // the final page from disk
    uint inner_index = bn % NINDIRECT;
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if ((addr = a[inner_index]) == 0){
      addr = balloc(ip->dev);
      if (addr == 0){ brelse(bp); return 0; }
      a[inner_index] = addr; log_write(bp);
    }
    brelse(bp); return addr;
  }
}

void itrunc(struct inode *ip){
  int i, j, k;
  struct buf *bp, *bp2;
  uint *a, *a2;
  // . . .
  // Task 5.1: free the doubly indirect blocks
  if (ip->addrs[NDIRECT + 1]){
    bp = bread(ip->dev, ip->addrs[NDIRECT
      +1]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT; j++){
      if (a[j]){
        bp2 = bread(ip->dev, a[j]);
```

```
      a2 = (uint*)bp2->data;
      for(k = 0; k < NINDIRECT; k++)
        if (a2[k]) bfree(ip->dev, a2[k]);
      brelse(bp2); bfree(ip->dev, a[j]);
    }
  }
```

```
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT+1]);
    ip->addrs[NDIRECT+1] = 0;
  }
}
```

1.2. Testing Doubly-Indirect Block Support

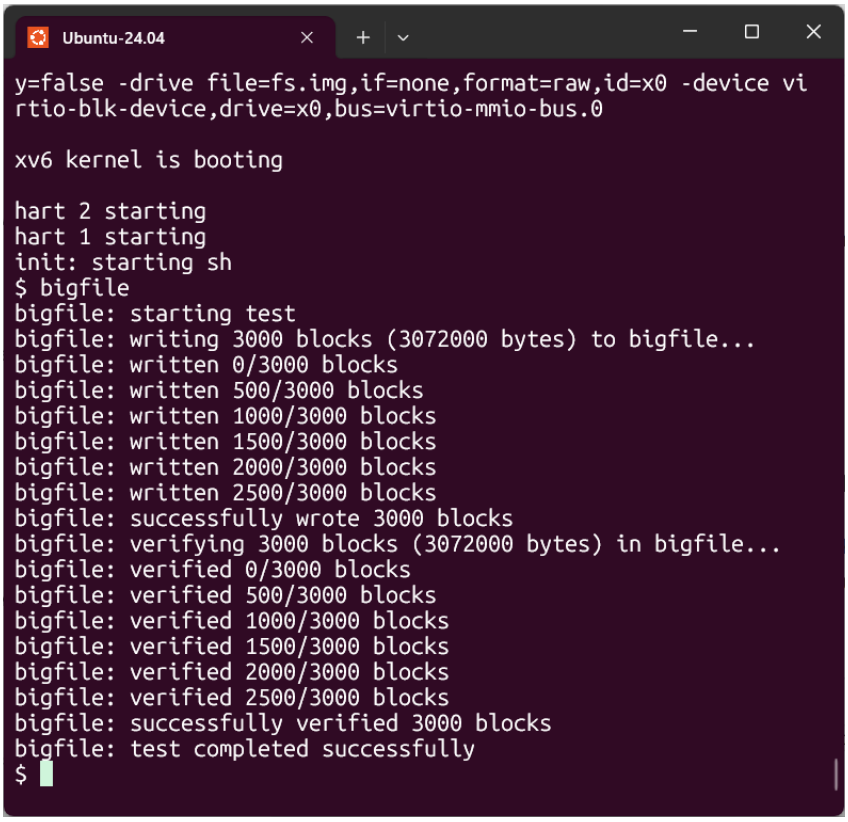

Figure 3. The test file user/bigfile.c allocates 3000 blocks (file size), fills them with distinct values and then checks the values at those 3000 blocks. The figure shows successful completion of the test.

2. Task 5.2: Symbolic Links (Soft Links)

Reference used: xiayingp.gitbook.io/build_a_os/labs/lab-8-file-system-symbolic-links

Synopsis of the problem Symbolic links refer to a linked file by pathname; when a symbolic link is opened, the kernel follows the link to the referred file. Symbolic links resembles hard links, but hard links are restricted to pointing to file on the same disk, while symbolic links can cross disk devices.

2.1. Implementation

1. Add T_SYMLINK in kernel/stat.h to denote another type of inode.

2. Define O_NOFOLLOW in kernel/fcntl.h.
3. Define the system call number in kernel/syscall.h for sys_symlink() and add it to the syscalls[] table in kernel/syscall.c.
4. Do modifications to sys_open() and sys_symlink() in kernel/sysfile.c as shown in Listing 4.
   - **sys_symlink()**: Implements the symlink system call to create a symbolic link named path that points to target. Receives both arguments; calls create(path, T_SYMLINK, 0, 0) to allocate an inode of type T_SYMLINK and stores the target inside the inode's data blocks using writei(). It unlocks and releases the inode (iunlockput(ip)) and finishes the transaction (end_op()).
   - Modified create() to include inode of type T_SYMLINK.
   - **sys_open()**: Retrieve the pathname and its flags. If the inode type is T_SYMLINK: (1) if O_NOFOLLOW is specified, open the symbolic link file itself instead of the target; and (2) if O_NOFOLLOW is not specified, then recursively resolve the path (till the depth < 10) using namei() and readi() until either a non-link file is found or the depth ≥ 10. Continue normal file descriptor allocation and setup once the correct inode is obtained.
5. Add the prototype of the syscall symlink in user/user.h and add it to user/usys.pl.

Listing 3. changes to variables

```c
/* kernel/stat.h */
// . . .
#define T_SYMLINK 4

/* kernel/fcntl.h */
// . . .
#define O_NOFOLLOW 0x800

/* kernel/syscall.h */
// . . .
#define SYS_symlink 22

/* kernel/syscall.c */
// . . .
extern uint64 sys_symlink(void);

static uint64 (*syscalls[])(void) = {
  // . . .
  [SYS_symlink] sys_symlink,
}

/* user/user.h */
// . . .
int symlink(const char *target, const char *
    path);
```

Listing 4. changes to sys_open() and creating sys_symlink() in kernel/sysfile.c

```c
static struct inode* create(char *path, short
    type, short major, short minor){
  // . . .
  if((ip = dirlookup(dp, name, 0)) != 0){
    iunlockput(dp);
    ilock(ip);
    if((type == T_SYMLINK) || (type == T_FILE
      && (ip->type == T_FILE || ip->type ==
      T_DEVICE))) // add T_SYMLINK
      recognition to inode -- Task 5.2
      return ip;
    iunlockput(ip);
    return 0;
  }
  // . . .
}

// Task 5.2 --- syscall: sys_symlink
uint64 sys_symlink(void){
  char target[MAXPATH], path[MAXPATH];
  if (argstr(0, target, MAXPATH) < 0 || argstr(1,
      path, MAXPATH) < 0) return -1;

  begin_op();
  struct inode *ip = create(path, T_SYMLINK,
      0, 0);
  if (ip == 0){ end_op(); return -1; }

  int len = strlen(target);
  if(writei(ip, 0, (uint64)target, 0, len + 1) != (len
      +1)){
    ip->nlink = 0; iupdate(ip); iunlockput(ip);
    end_op(); return -1;
  }

  iunlockput(ip);
  end_op();
  return 0;
}

uint64
sys_open(void)
```

```
{
 //. . .
 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);
  // Task 5.2
  if ((ip−>type == T_SYMLINK) && !(omode
    & O_NOFOLLOW)){
   int count = 0;
   while (ip−>type == T_SYMLINK && count
     < 10){
    int len = 0;
    if (readi(ip, 0, (uint64)path, 0, len + 1) != (
    len+1)){ iunlockput(ip); end_op(); return
    −1; }
    iunlockput(ip);

    if((ip = namei(path)) == 0){ end_op();
    return −1; }
    ilock(ip);
    count++;
   }
   if (count >= 10) { // recursive depth till 10
    printf("cycle for symlink, error\n");
    iunlockput(ip);
    end_op();
    return −1;
   }
  }
  //. . .
 }
 //. . .
```

2.2. Testing Symbolic links

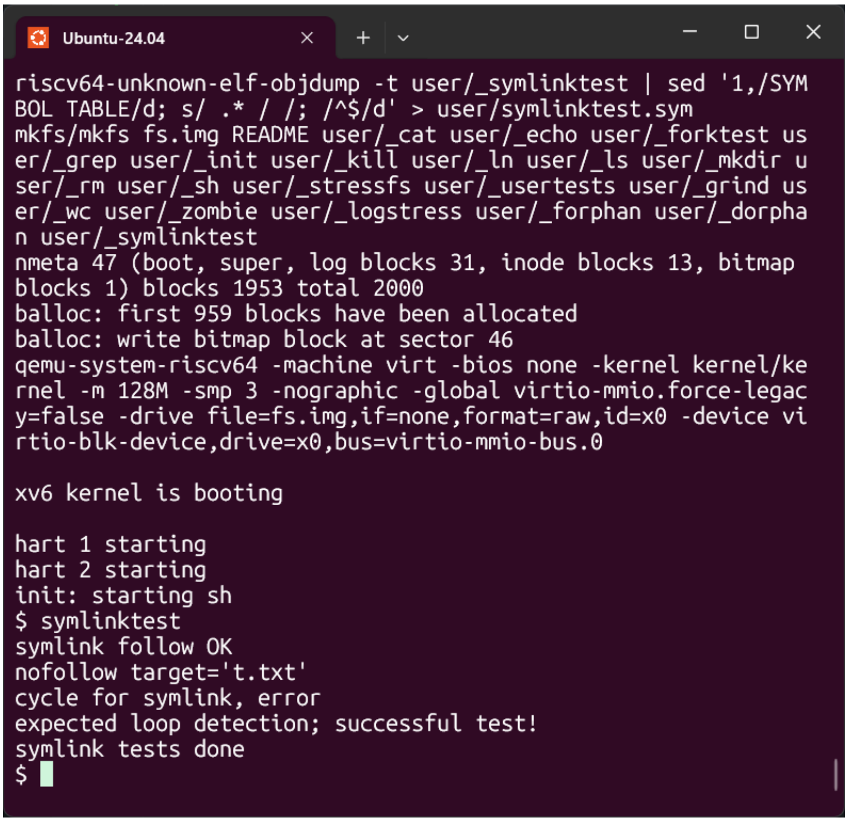

Figure 4. Testing the implementation using user/symlinktest.c: We create a base file (`t.txt`), link it symbolically (`l.txt`), verify content access through the link, test the O_NOFOLLOW flag to read the target path itself, and finally check loop detection with recursive links ($l_A \rightarrow l_B \rightarrow l_A$). The terminal output confirms all stages