

# Operating System Project3 Wiki

12300

2018079116

정진성

## Contents

- 1. Design**
- 2. Implement**
- 3. Result**
- 4. Trouble shooting**

## 1. Design

xv6의 파일은 inode에 구조체로 관리됩니다.

```
// On-disk inode structure
struct dinode {
    short type;          // File type
    short major;         // Major device number (T_DEV only)
    short minor;         // Minor device number (T_DEV only)
    short nlink;         // Number of links to inode in file system
    uint size;           // Size of file (bytes)
    uint addrs[NDIRECT+1]; // Data block addresses
};
```

[fs.c]

이 중 addrs에 파일의 실제 주소가 관리됩니다. NDIRECT 개수 만큼 직접적으로

데이터 블록을 가리키고, 1은 간접적으로 데이터를 가리키는 블록을 관리합니다. 이러한 파일 시스템은 최대  $12 \times 512 + 512 \times 128 = 6144 + 65536 = 71680B$ , 약 70KB의 파일 관리가 최대입니다. 더 큰 파일을 관리하기 위해 DIRECT의 계층을 늘려 TRIPLE INDIRECT를 만들어 약 1GB의 파일 크기를 관리하도록 하려고 합니다. addrs를 변경해 direct 데이터블록을 감소시키고 그 만큼 indirect 블록을 늘려 이를 구현할 수 있습니다.

```
// Triple indirect implement
#define NDIRECT 10 // To simplify implementation reduce NDIRECT
#define NINDIRECT (BSIZE / sizeof(uint)) // 128
#define DOUBLE_INDIRECT (NINDIRECT * NINDIRECT) // 128 * 128
#define TRIPLE_INDIRECT (DOUBLE_INDIRECT * NINDIRECT) // 128 * 128 * 128
#define MAXFILE (NDIRECT + NINDIRECT + DOUBLE_INDIRECT + TRIPLE_INDIRECT)
||| // 10 + 128 + 128 * 128 + 128 * 128 * 128
||| // 2113674 * 512 = 1GB
```

[fs.c]

기본 xv6는 link 기능을 지원하는 데 이는 hard link입니다. 원본 파일을 복사한 개념입니다. symbolic link를 구현하기 위해서는 hard link와 다르게 기존 inode를 그대로 사용하는 것이 아닌 새로 inode를 만들고 이를 원본을 가리키도록 하여 구현하려 했습니다. 따라서 새로운 파일 타입인 T\_SYM을 정의하고, inode에 target이라는 변수를 추가하여 T\_SYM일 경우 target을 활용하도록 만들었습니다.

```
#define T_DIR 1 // Directory
#define T_FILE 2 // File
#define T_DEV 3 // Device
#define T_SYM 4 // Symbolic link file
```

[stat.h]

```

// On-disk inode structure
struct dinode {
    short type;          // File type
    short major;         // Major device number (T_DEV only)
    short minor;         // Minor device number (T_DEV only)
    short nlink;         // Number of links to inode in file system
    uint size;           // Size of file (bytes)
    char target[64];     // Symbolic target

    uint addrs[NDIRECT+3]; // Data block addresses and triple indirect
};

```

[fs.h]

## 2. Implement

### - Multi Indirect

dinode구조체를 변경했을 때 최대한 다른 기능에 영향이 없도록 assert((BSIZE % sizeof(struct dinode)) == 0);를 만족하도록 하기 위해 NDIRECT의 크기를 10으로 조정하고 triple indirect를 위해 +3을 해주어 공간을 만들었습니다.

```

// On-disk inode structure
struct dinode {
    short type;          // File type
    short major;         // Major device number (T_DEV only)
    short minor;         // Minor device number (T_DEV only)
    short nlink;         // Number of links to inode in file system
    uint size;           // Size of file (bytes)
    char target[64];     // Symbolic target

    uint addrs[NDIRECT+3]; // Data block addresses and triple indirect
};

```

[fs.h]

inode 구조체 또한 똑같이 변경해 주었습니다.

```

// in-memory copy of an inode
struct inode {
    uint dev;           // Device number
    uint inum;          // Inode number
    int ref;            // Reference count
    struct sleeplock lock; // protects everything below here
    int valid;          // inode has been read from disk?

    short type;         // copy of disk inode
    short major;
    short minor;
    short nlink;
    uint size;
    char target[64];

    uint addrs[NDIRECT+3]; // triple indirect
};


```

[file.h]

xv6에서 bmap함수는 파일 시스템에서 데이터 블록의 물리적인 위치를 찾는 데 사용됩니다. 파일 시스템은 파일의 데이터를 여러 블록에 저장하고, 각 블록은 디스크의 일부 공간에 저장됩니다. 따라서 bmap함수는 주어진 파일의 블록 번호에 해당하는 블록의 물리적인 주소를 계산하기 때문에 변경된 파일 시스템에서도 올바르게 블록을 계산하여 할당하도록 변경해 주어야합니다.

기존 bmap 함수에서의 구조와 동일하게 구현하였습니다. bn이 NDIRECT 보다 작으면 바로 addrs의 direct로 가게 저장하였고 그 크기보다 크다면 NINDIRECT, DOUBLE INDIRECT를 각각 단계를 지날 때마다 빼주어 bn의 크기에 따라 어느 계층에 할당해야할지 계산해 주었습니다. double indirect는 ip->addrs의 11번째 인덱스에 할당해주며 그 인덱스에는 double에 맞게 아래 단계의 블록의 주소를 할당해 더 깊은 블록을 찾아가게 구현하였습니다. 따라서 bp와 a를 사용하는 단계가 두번 구성되었고, triple은 동일하게 세단계로 구성하여 bn을 쪼개서 구현하였습니다.

```
bn -= NINDIRECT;

if(bn < DOUBLE_INDIRECT){
    // Load double indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT+1]) == 0)
        ip->addrs[NDIRECT+1] = addr = malloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;

    if((addr = a[bn/NINDIRECT]) == 0){
        a[bn/NINDIRECT] = addr = malloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);

    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn%NINDIRECT]) == 0){
        a[bn%NINDIRECT] = addr = malloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);
    return addr;
}
bp -= DOUBLE_INDIRECT;
```

[fs.c]

```

}

bp -= DOUBLE_INDIRECT;

if(bn < TRIPLE_INDIRECT){
    // Load triple indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT+2]) == 0)
        ip->addrs[NDIRECT+2] = addr = balloc(ip->dev);

    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn/DIRECT]) == 0){
        a[bn/DIRECT] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);

    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[(bn%DOUBLE_INDIRECT)/NINDIRECT]) == 0){
        a[(bn%DOUBLE_INDIRECT)/NINDIRECT] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);

    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn%NINDIRECT]) == 0){
        a[bn%NINDIRECT] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);
    return addr;
}
panic("bmap: out of range");
}

```

[fs.c]

또한 xv6에서 `itrunc`함수는 파일의 데이터를 초기화하고 파일 크기를 변경하는 데 사용됩니다. 파일 시스템에서 파일을 삭제할 때나 파일 크기를 줄일 때 호출되어 파일의 데이터를 삭제하고 해당 파일의 inode 정보를 갱신하기 때문에 변경이 필요한 함수입니다.

`triple indirect`에서는 레벨이 3이기 때문에 버퍼가 3개씩 필요하여 이를 더 추가해 주었고, 나머지는 기존의 `itrunc` 구조와 동일하게 구현하였습니다.

```
static void
itrunc(struct inode *ip)
{
    int i, j, k;
    struct buf *bp, *bp2, *bp3;
    uint *a, *a2, *a3;
```

[fs.c]

addrs[12]가 0이 아니면 데이터가 저장되어 있다는 뜻이므로 bp에 저장하고 a에 주소를 저장하여 a를 탐색합니다. 같은 구조로 a를 탐색하며 0인지 검사하고 0이 아니라면 bp에 저장하고 a에 주소를 저장하는 단계를 레벨 3까지 들어가며 이후 블록을 해제 시켜줍니다. 이를 통해 triple indirect 구조를 지원하는 itrunc함수를 구현할 수 있습니다.

```

if(ip->addrs[NDIRECT+2]){
    bp = bread(ip->dev, ip->addrs[NDIRECT+2]);
    a = (uint*)bp->data;

    for(i = 0; i < NINDIRECT; i++){
        if(a[i]){
            bp2 = bread(ip->dev, a[i]);
            a2 = (uint*)bp2->data;

            for(j = 0; j < NINDIRECT; j++){
                if(a2[j]){
                    bp3 = bread(ip->dev, a2[j]);
                    a3 = (uint*)bp3->data;

                    for(k = 0; k < NINDIRECT; k++){
                        if(a3[k])
                            bfree(ip->dev, a3[k]);
                    }

                    brelse(bp3);
                    bfree(ip->dev, a2[j]);
                }
            }

            brelse(bp2);
            bfree(ip->dev, a[i]);
        }
    }

    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT+2]);
    ip->addrs[NDIRECT+2] = 0;
}

```

[fs.c]

마지막으로 시스템이 큰 파일구조를 허용하도록 param.h의 FSIZE를 늘려 1GB정도의 파일을 관리할 수 있도록 해주었습니다.

```
1 #define NPROC      64 // maximum number of processes
2 #define KSTACKSIZE 4096 // size of per-process kernel stack
3 #define NCPU       8 // maximum number of CPUs
4 #define NOFILE     16 // open files per process
5 #define NFILE      100 // open files per system
6 #define NINODE     50 // maximum number of active i-nodes
7 #define NDEV        10 // maximum major device number
8 #define ROOTDEV    1 // device number of file system root disk
9 #define MAXARG     32 // max exec arguments
10 #define MAXOPBLOCKS 10 // max # of blocks any FS op writes
11 #define LOGSIZE    (MAXOPBLOCKS*3) // max data blocks in on-disk log
12 #define NBUF       (MAXOPBLOCKS*3) // size of disk block cache
13
14 // about 2113674 blocks are possible
15 #define FSSIZE     3000000 // size of file system in blocks
16
17
```

[param.h]

### - Symbolic link

기존의 link 함수를 -h와 -s 옵션으로 나누어 하드링크와 심볼릭 링크를 구분하여 생성할 수 있도록 수정하였습니다. 심볼릭 링크를 구현하는 시스템 콜 symlink 함수도 추가해 주었습니다.

```

// Adding symbolic link
int
main(int argc, char *argv[])
{
    if(argc != 4){
        printf(2, "Usage: ln [-o] old new\n");
        exit();
    }

    // Hard link
    if (strcmp(argv[1], "-h") == 0) {
        if(link(argv[2], argv[3]) < 0)
            printf(2, "Hard link %s %s: failed\n", argv[2], argv[3]);
    }
    // Symbolic link
    else if(strcmp(argv[1], "-s") == 0) {
        if(symlink(argv[2], argv[3]) < 0)
            printf(2, "Symbolic link %s %s: failed\n", argv[2], argv[3]);
    }

    exit();
}

```

[ln.c]

기존 hard 링크를 지원하는 sys\_link의 구조를 참고하여 구현하였습니다. 새로운 sym\_ip를 할당하기 위해 create를 사용하여 T\_SYM타입의 inode를 생성하였고 sym\_ip의 target에 현재 old의 path name을 저장하여 접근시 이를 따라가도록 설정하였습니다.

```

// Create the path symbolic as a link to the old inode.
int
sys_symlink(void)
{
    char *new, *old;
    struct inode *sym_ip, *ip ;
    if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
        return -1;

    begin_op();
    if(ip = namei(old)) == 0){
        end_op();
        return -1;
    }

    if(ip->type == T_DIR){
        end_op();
        return -1;
    }

    sym_ip = create(new, T_SYM, 0, 0); // Create T_SYM type inode
    if(sym_ip == 0) {
        iunlock(sym_ip);
        end_op();
        return -1;
    }
    iunlock(sym_ip);

    ilock(sym_ip);
    strncpy(sym_ip->target,old,strlen(old));
    sym_ip->size = ip->size;
    iupdate(sym_ip);
    iunlockput(sym_ip);

    end_op();
    return 0;
}

```

[sysfile.c]

또한 이 T\_SYM 타입의 파일을 열 때 target의 path로 이동해야하므로 sys\_open 파일의 일부를 수정하였습니다. path에 target의 이름을 저장하여 namei함수를 통해 inode를 얻고 T\_SYMBOL이 아닐때까지 탐색한 후 그 파일을 열어 주는 방식으로 구현하였습니다.

```

while(ip->type == T_SYM) { // Considering recursive
    strncpy(path,ip->target,64);

    if((next_ip = namei(path)) == 0) { // Get inode
        iunlockput(ip);
        end_op();
        return -1;
    }

    iunlockput(ip);
    ip = next_ip;
    ilock(ip);

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

```

[sysfile.c]

### 3. Result

컴파일과정은 이전과 동일합니다. 하지만 편하게 명령어를 입력하기 위해 shell script를 작성했기에 sh makemake.sh, sh start.sh 를 입력하면 편하게 컴파일 및 실행할 수 있습니다.

```

1  #!/bin/sh
2
3  make clean
4  make
5  make fs.img

```

[makemake.sh]

```

1  #!/bin/sh
2
3  qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512

```

[start.sh]

FSIZE를 늘린 탓인지 컴파일 속도가 이전보다 눈에 띄게 늘었지만 조금만 기다리면 완료됩니다.

### - Multi Indirect

16MB의 파일을 읽고 쓰는 테스트 파일을 작성하였습니다. 버퍼에 알파벳을 저장하고 이를 포함하는 파일을 만들고 읽습니다.

```
#define NUM_BYTES 16 * 1024 * 1024 // 16 MB

char buf[NUM_BYTES], buf2[NUM_BYTES];
char filename[16] = "test_file0";

void failed(const char *msg)
{
    printf(1, msg);
    printf(1, "Test failed!!\n");
    exit();
}

void test1()
{
    int fd;

    printf(1, "Test 1: Write %d bytes\n", NUM_BYTES);
    fd = open(filename, O_CREATE | O_WRONLY);
    if (fd < 0)
        failed("File open error\n");
    if (write(fd, buf, NUM_BYTES) < 0)
        failed("File write error\n");
    if (close(fd) < 0)
        failed("File close error\n");
    printf(1, "Test 1 passed\n\n");
}
```

[file\_test.c]

```

void test2()
{
    int i, fd;
    for (i = 0; i < NUM_BYTES; i++)
        buf2[i] = 0;

    printf(1, "Test 2: Read %d bytes\n", NUM_BYTES);
    fd = open(filename, O_RDONLY);
    if (fd < 0)
        failed("File open error\n");
    if (read(fd, buf2, NUM_BYTES) < 0)
        failed("File read error\n");
    for (i = 0; i < NUM_BYTES; i++) {
        if (buf2[i] != (i % 26) + 'a') {
            printf(1, "%dth character, expected %c, found %c\n", i, (i % 26) + 'a', buf2[i]);
            failed("");
        }
    }
    if (close(fd) < 0)
        failed("File close error\n");
    if (unlink(filename) < 0)
        failed("File unlink error\n");
    if ((fd = open(filename, O_RDONLY)) >= 0)
        failed("File not erased\n");
    printf(1, "Test 2 passed\n\n");
}

int main(int argc, char *argv[])
{
    int i;
    for (i = 0; i < NUM_BYTES; i++)
        buf[i] = (i % 26) + 'a'; // alpahbet

    test1();
    test2();

    exit();
}

```

[file\_test.c]

```

$ file_test
Test 1: Write 16777216 bytes
Test 1 passed

Test 2: Read 16777216 bytes
Test 2 passed

```

\$

결과를 확인해보면 wirte와 read 모두 잘 동작함을 확인할 수 있습니다.

## - Symbolic link

symbolic link 기능을 테스트하기 위해 symbolic\_test.c를 작성하였습니다.

원본 파일을 생성하고 symlink함수를 통해 symbolic link를 만들고 그 링크 파일을 읽었을 때의 결과를 살펴보는 과정을 거칩니다. 이후 원본파일이 삭제 되었을 경우 파일에 접근할 수 없어야하는데 이를 확인하는 과정은 주석처리를 해제하여 확인할 수 있습니다.

```
void create_file(const char* path, const char* content) {
    int fd = open(path, O_WRONLY | O_CREATE);
    if (fd < 0) {
        printf(1, "Error creating file: %s\n", path);
        return;
    }
    write(fd, content, strlen(content));
    close(fd);
}

void read_file(const char* path) {
    int fd = open(path, O_RDONLY);
    if (fd < 0) {
        printf(1, "Error opening file: %s\n", path);
        return;
    }

    char buffer[100];
    int bytesRead = read(fd, buffer, sizeof(buffer)-1);
    if (bytesRead < 0) {
        printf(1, "Error reading file: %s\n", path);
    } else {
        buffer[bytesRead] = '\0';
        printf(1, "Content of %s: %s\n", path, buffer);
    }

    close(fd);
}
```

[symbolic\_test.c]

```

int main(void) {
    // Create a symbolic link
    char* target = "/file.txt";
    char* symlinkfile = "/symlinkfile.txt";
    // Create the target file
    char* content = "Hello, world!";
    create_file(target, content);

    if (symlink(target, symlinkfile) < 0) {
        printf(1, "Error creating symbolic link\n");
        exit();
    }

    // Read from the symbolic link
    read_file(symlinkfile);

    // Remove the original file
    if (unlink(target) < 0) {
        printf(1, "Error removing file: %s\n", target);
    }

    // Attempt to read from the symbolic link again
    //read_file(symlinkfile);

    exit();
}

```

[symbolic\_test.c]

```

$ symbolic_test
Content of /file.txt: Hello, world!
$ 

```

결과를 확인해 보면 원본파일에 잘 접근하였고 Hello, world!라는 contents 또한 잘 읽었음을 확인할 수 있었고 unlink 또한 오류가 발생하지 않았음을 확인할 수 있었습니다.

이제 원본파일을 지우고 link 파일에 접근하기 위해 밑의 read\_file 주석을 제거하

여 실행하여 보겠습니다.

```
$ symbolic_test  
Content of /file.txt: Hello, world!  
Error opening file: /file.txt  
$
```

linkfile의 inode에 target으로 들어가 보았지만 파일이 없어 오류가 발생하는 것을 확인할 수 있었습니다.

## 4. Trouble shooting

symbolic link의 테스트 파일을 작성하여 확인하면 잘 동작함을 확인할 수 있었지만 xv6의 shell에서 ln -s 명령어로 생성한 파일은 이름이 원본파일의 이름과 같아지는 현상을 확인하였습니다. 이를 위해 NO\_FOLLOW 모드를 만들어 ls 명령어에 활용하려 하였습니다.

ls시 open을 할때는 NOFOLLOW로 open하면 target을 쫓지 않기 때문에 새로 만들어진 이름의 파일을 확인할 수 있을거라 생각했지만 생각대로 되지 않았습니다.

ls.c의 while문 안에서 T\_SYM을 확인할 시 NOFOLLOW를 하려 했지만 이미 open을 한 상황이었기 때문에 이 방법으로는 구현하기 힘들었습니다.

```
if (!(omode & O_NOFOLLOW)) {
    while(ip->type == T_SYM) { // Considering recursive
        strcpy(path,ip->target,64);

        if( (next_ip = namei(path)) == 0) { // Get inode
            iunlockput(ip);
            end_op();
            return -1;
        }

        iunlockput(ip);
        ip = next_ip;
        ilock(ip);

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

[NOFOLLOW를 활용한 sys\_open]