

Project #3 : File System

0. Project #3

1. Multi Indirect

1-1. Design

1-2. Implementation

1. struct inode
2. struct dinode
3. bmap
4. itrunc

2. Symbolic Link

2-1. Design

2-2. Implementation

1. T_SYMLINK
2. sys_symlink
3. sys_open
4. ls

3. Sync

3-1. Design

3-2. Implementation

1. begin_op
2. end_op
3. sync

4. Result

4-1. multi indirect test

4-2. Symbolic link test

4-3. sync test

5. Trouble Shooting

5-1. Multi Indirect

1. inode, dinode struct
2. param.h

5-2. Symbolic Link

1. sys_symlink
2. open
3. ls

5-3. Sync

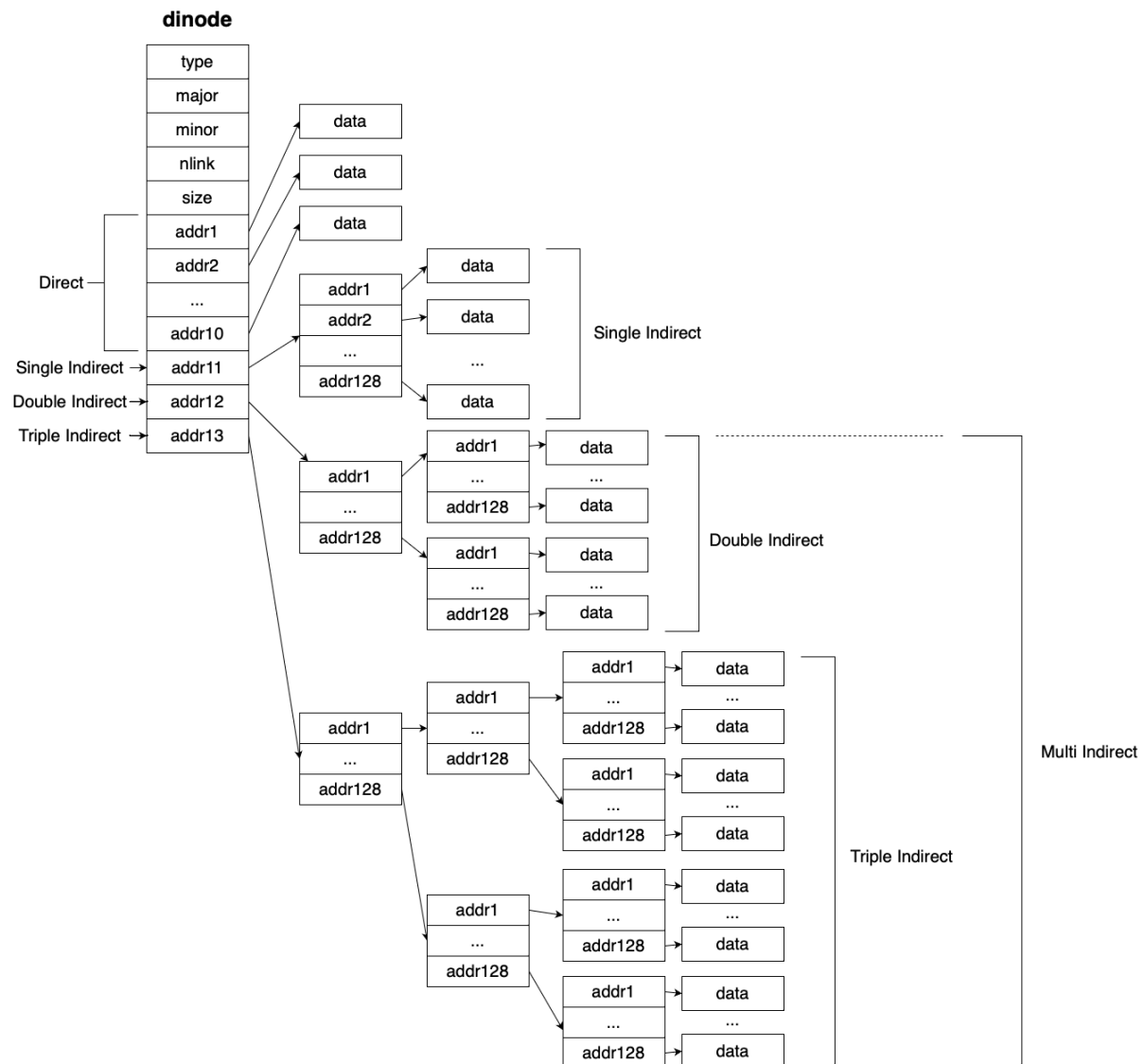
1. begin_op
2. sync

0. Project #3

- 이번 프로젝트의 목표는 xv6의 파일 시스템에 multi indirect, symbolic link, buffered I/O의 기능을 추가하여 xv6의 file system을 개선시키는 것이 목표이다.
 - 구현해야하는 목록은 다음과 같다.
1. multi indirect
 2. symbolic link
 3. buffered I/O

1. Multi Indirect

1-1. Design



- 나의 디자인은 위와 같은 그림으로 이루어져있다.
- 기존에 Directed는 10개, Single Indirect, Double Indirect, Triple Indirect는 각각 1개씩으로 디자인하였다.

1-2. Implementation

- 다음은 Multi Indirect를 구현할 때 영향을 미치는 구조체들이다.

1. struct inode

- 파일이나 디렉토리(디렉토리도 사실 특수한 파일이다.)의 metadata를 담는 구조체이며 xv6에서는 다음과 같이 구현되어있다. 실제로는 메모리 상에 저장되는 구조체이다.

```
// 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;
    uint addrs[NDIRECT+1];
};
```

2. struct dinode

- disk inode의 줄임말이며 실제로 디스크 상에 저장되는 inode를 나타낸다.
- xv6에는 다음과 같이 구현되어있다.

```
// 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
};
```

아래부터는 실제 구현에 대해서 다루겠다.

```
// main line 84 in mkfs.c
assert((BSIZE % sizeof(struct dinode)) == 0);
```

- 위를 보면 기존 `dinode` 구조체에서 더 늘리거나 더 줄일 경우 `assert`가 남을 확인하였고 기존 `direct block address`의 수를 줄여서 `double indirect`와 `triple indirect`를 위한 공간을 마련하였다.
- `direct`를 줄이고 `double`, `triple indirect`를 위한 공간을 마련한 구조체는 다음과 같다.

```
#define NDIRECT 10
#define NINDIRECT (BSIZE / sizeof(uint))
#define D_NINDIRECT (NINDIRECT * NINDIRECT)
#define T_NINDIRECT (D_NINDIRECT * NINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT + D_NINDIRECT + T_NINDIRECT)

// 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+3]; // Data block addresses
};
```

- `NDIRECT`는 위에서 설명한 것과 같은 이유로 10으로 줄였다.

- double indirect의 block address의 숫자는 다음과 같이 계산할 수 있다.

◦ # double indirect = # single indirect * # single indirect

- triple indirect의 block address의 숫자는 다음과 같이 계산할 수 있다.

◦ # triple indirect = # single indirect * # single indirect * # single indirect
= # double indirect * # single indirect

- 따라서 MAXFILE의 block address의 숫자도 다음과 같이 계산할 수 있다.

◦ total block address = # direct + # single indirect + # double indirect + # triple indirect

- `struct dinode`를 수정하였으니 `struct inode`도 또한 다음과 같이 수정해야한다.

```
// in file.h
// in-memory copy of an inode
struct inode {
    ...
    uint addrs[NDIRECT+3];
};
```

3. bmap

- 함수의 형식은 다음과 같다.

```
static uint bmap(struct inode *ip, uint bn);
```

- bmap은 inode ip에서 bn번째 block의 주소(disk)를 return하는 함수이다. 만약 해당 block이 할당되어있지 않다면 해당 block을 할당하여 해당 주소를 반환한다.

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

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

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

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

기존 bmap의 작동 방식은 다음과 같다.

1. 우선 각 direct / indirect type의 block에 들어 갈 수 있는 지 체크한다.
 2. direct인 경우 해당 block에 해당하는 주소를 반환한다. (만약, 할당이 필요한 경우 할당하고 해당 주소를 반환한다.)
 3. indirect인 경우 bn에 해당하는 block을 찾아서 반환하는데 bread를 통해서 ip → dev의 정보를 읽고 해당 위치의 block을 찾아서 주소를 반환한다.
- 기존에는 direct, single indirect만 존재하였고, 이번 project에서 double, triple등 multi indirect가 추가 되었으므로 다음과 같이 작성할 수 있다. (bmap의 indirect를 참고하였다.)
 - double Indirect는 다음과 같다.

```
bn -= NINDIRECT;

// TO-DO : Multi Indirect
// Implement double and triple indirect block

// double indirect block (MAX FILE SIZE OF DOUBLE INDIRECT : 8MiB)
// Implement double indirect
if(bn < D_NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NINDIRECT+1]) == 0)
        ip->addrs[NINDIRECT+1] = addr = balloc(ip->dev);

    // single indirect
    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);

    // double indirect
    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 data addr
    return addr;
}
```

- triple indirect는 다음과 같다.

```
bn -= D_NINDIRECT;

// triple indirect block (MAX FILE SIZE OF TRIPLE INDIRECT : 1GiB)
// Implement triple indirect
if(bn < T_NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NINDIRECT+2]) == 0)
        ip->addrs[NINDIRECT+2] = addr = balloc(ip->dev);

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

```

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

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

```

- multi indirect는 single indirect와 같은 logic으로 작동하도록 만들었다.

```

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

```

- 위의 구조가 반복 되면서 bn에 해당하는 block을 찾아서 주소를 반환하면 된다. 위의 구조가 3번 반복되면 triple, 2번 반복되면 double indirect이다.

4. itrunc

- itrunc 함수의 정의는 다음과 같다.

```
static void itrunc(struct inode *ip);
```

- ip에 존재하는 block들을 할당을 해제하는 함수이다. 주석에 따르면 inode에 대한 링크가 존재하지 않고 메모리 내 참조가 없는 경우에 호출되어 기존의 할당된 block들을 정리하는 함수라고 할 수 있다.
- 역시 기존 itrunc함수의 single indirect부분을 참고하여 작성하였다.

```

// TO-DO : Multi Indirect
// Implement double and triple indirect block

// Double indirect
// Find allocated blocks recursive and Free all of allocated blocks
if(ip->addrs[NINDIRECT+1]){
    bp = bread(ip->dev, ip->addrs[NINDIRECT+1]);
    a = (uint*)bp->data;
    for(i = 0; i < NINDIRECT; i++){
        // Check whether this block is allocated
        if (a[i]) {
            bp2 = bread(ip->dev, a[i]);
            a2 = (uint*)bp2->data;
            // free 2-depth blocks
            for(j = 0; j < NINDIRECT; j++){
                // Check whether this block is allocated
                if(a2[j])
                    bfree(ip->dev, a2[j]);
            }
        }
    }
}

```

```

        // free 1-depth blocks
        brelse(bp2);
        bfree(ip->dev, a[i]);
        a[i] = 0;
    }
}
brelse(bp);
bfree(ip->dev, ip->addrs[NDIRECT+1]);
ip->addrs[NDIRECT+1] = 0;
}

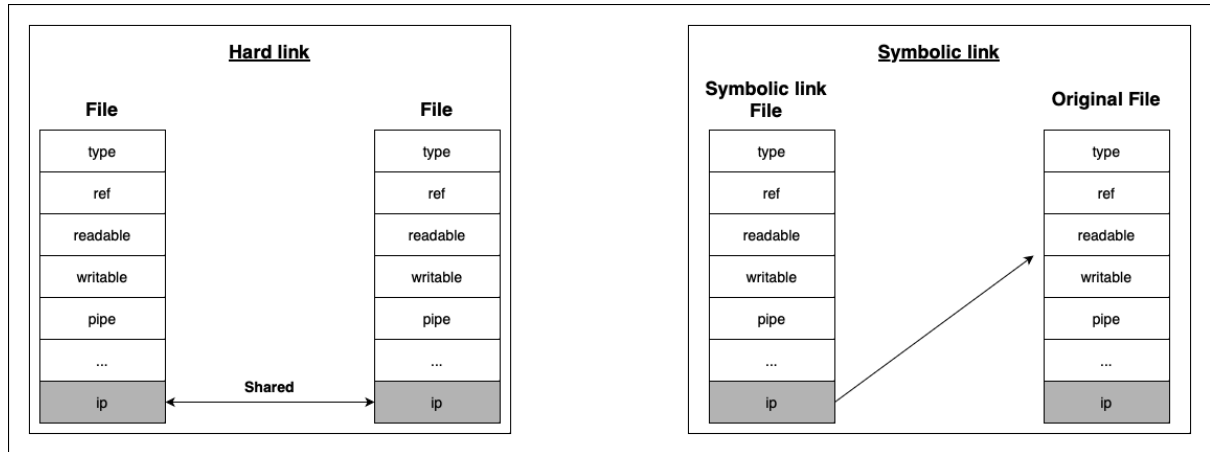
// Triple Indirect
// Find allocated blocks recursive and Free all of allocated blocks
if(ip->addrs[NDIRECT+2]){
    bp = bread(ip->dev, ip->addrs[NDIRECT+2]);
    a = (uint*)bp->data;
    for(i = 0; i < NINDIRECT; i++){
        // Check whether this block is allocated
        if (a[i]) {
            bp2 = bread(ip->dev, a[i]);
            a2 = (uint*)bp2->data;
            for(j = 0; j < NINDIRECT; j++){
                // Check whether this block is allocated
                if(a2[j]){
                    bp3 = bread(ip->dev, a2[j]);
                    a3 = (uint*)bp3->data;
                    // free 3-depth blocks
                    for(k = 0; k < NINDIRECT; k++){
                        // Check whether this block is allocated
                        if(a3[k])
                            bfree(ip->dev, a3[k]);
                    }
                    // free 2-depth blocks
                    brelse(bp3);
                    bfree(ip->dev, a2[j]);
                    a2[j] = 0;
                }
            }
            // free 1-depth blocks
            brelse(bp2);
            bfree(ip->dev, a[i]);
            a[i] = 0;
        }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT+2]);
    ip->addrs[NDIRECT+2] = 0;
}
}

```

- 구조는 할당된 block의 가장 깊은 depth까지 들어가서 밑에서부터 부터 bfree를 이용해서 block을 비워주는 함수이다. 마지막에는 ip → addrs[...]에 0을 넣어줌으로써 빈 공간임을 알려줄 수 있도록 한다.

2. Symbolic Link

2-1. Design



우선 Hard Link와 Symbolic Link에 대해서 간단하게 설명하도록 하겠다.

- Hard Link : 원본 파일의 I-node를 공유하는 파일이며, 별도의 파일이지만, 변경 내역은 공유된다. 그래서 한 파일이 제거되어도 다른 파일에게 영향을 주지 않는다.
- Symbolic Link : 일종의 바로가기 파일로 Symbolic Link파일에 접근하면 원본 파일로 Redirection되어야한다. 만약 원본 파일이 제거될 경우, Symbolic Link파일은 그대로 남아있으나, 접근하지 못한다.

To-Do 1:

기존 xv6에는 3가지 type의 파일이 존재한다.

```
// in stat.h
#define T_DIR 1 // Directory
#define T_FILE 2 // File
#define T_DEV 3 // Device
```

- 따라서 새로운 Type이 필요하게 되었다.

To-Do 2:

기존 Open system call을 보면 `if((ip = namei(path)) == 0)` 으로 path기반으로 파일을 연다는 것을 알 수 있다. 따라서 ip에 원본 파일의 이름(path)를 넣어서 보관하도록 구성하였다.

To-Do 3:

Symbolic link file을 열 경우, 원본 파일로 redirection하여 원본 파일을 여는 듯한 모습을 보여주어야한다. 따라서 기존의 Open도 또한 재귀적이나 반복적으로 호출하여 원본 파일을 열 수 있도록 하여야한다.

To-Do 4:

Symbolic link파일 끼리 cycle이 생성되는 경우 이를 탐지하기 위한 알고리즘이 필요하다.

2-2. Implementation

1. T_SYMLINK


```
// in stat.h
#define T_DIR 1 // Directory
#define T_FILE 2 // File
#define T_DEV 3 // Device
#define T_SYMLINK 4 // symbolic link
```

- 위를 통해서 우선 symbolic link type을 다른 type의 파일과 구분하였다.

2. sys_symlink

Symlink system call은 다음과 같이 작성하였다.

```
// system call : symlink (symbolic link)
// this system call
// in qemu, you can use this system call "ln -s old new"
int
sys_symlink(void)
{
    // local variable in system call
    char *new, *old;
    struct inode *ip;
    int len;

    // Use argstr, fetch the parameter of symlink
    if (argstr(0, &old) < 0 || argstr(1, &new) < 0)
        return -1;

    begin_op();
    // Make symlink type file named new (or pathname new)
    if((ip = create(new, T_SYMLINK, 0, 0)) == 0){
        end_op();
        return -1;
    }

    // Put the length and name of the old file in the ip of the new file
    len = strlen(old);

    // First, Put the length of the file named old.
    // This information is used to get the file named old exactly.
    writei(ip, (char *)&len, 0, sizeof(len));

    // Then, Put the name (path) of the file.
    writei(ip, old, sizeof(len), len + 1);
    iunlockput(ip);
    end_op();
    return 0;
}
```

- 작동과정은 다음과 같다.
1. 우선 새로운 type의 파일을 생성해야하므로 create함수를 통해서 new라는 path로 symbolic link type의 파일을 생성하였다.
 2. 그 후 old path의 길이와 이름을 ip에 writei라는 function을 이용해서 넣어주었다.
 - 길이를 넣어준 이유는 나중에 open에서 원본 파일을 찾기 위해 ip내부의 정보를 꺼내야하는데, 이때 old path의 길이를 알면 효율적으로 꺼내올 수 있기 때문에 old path의 length를 넣어주었다.

3. sys_open

```
...
// To-Do : Symbolic link file open
```

```

// if (!(omode & O_NOFOLLOW)) {
// Variable named depth : check the cycle between the symlink files
// If depth is greater than or equal to 20, It is determined that there is a cycle between symbolic link files.
depth = 0;
while(ip->type == T_SYMLINK && depth < 20) {
    // symbolic link file

    // Load length of file name and file name.
    readi(ip, (char *)&len, 0, sizeof(len));
    readi(ip, dest, sizeof(len), len+1);

    // Insert '\0' value to represent the end of the string.
    dest[len] = '\0';

    // Using namei function, load new ip
    iunlockput(ip);
    if((ip = namei(dest)) == 0){
        end_op();
        return -1;
    }
    ilock(ip);
    if(ip->type == T_DIR && omode != O_RDONLY){
        iunlockput(ip);
        end_op();
        return -1;
    }
    depth++;
}
// Probably a cycle between symlink files
if (ip->type == T_SYMLINK && depth >= 20) {
    iunlockput(ip);
    end_op();
    return -1;
}
// }
...

```

- symlink file을 open할경우 원본 파일로 redirection하는 코드이다.
- 작동과정은 다음과 같다.
 1. 만약 현재 ip의 type이 symbolic link이고 depth가 20이하일 경우 반복한다.
 2. sys_symlink에서 path의 길이와 이름을 넣어주었는데 이를 차례대로 꺼내온다.
 - a. 길이는 이름을 정확한 길이로 가져오기 위해서 사용된다.
 3. 그 후 namei를 이용해서 해당 pathname에 해당하는 ip를 load한다.
 4. 만약 directory인데 readonly가 아니면 -1을 반환한다.
 5. 그 후 depth를 증가시키고 2번으로 돌아간다.
- depth는 cycle이 존재하는지 판단하기 위해서 넣어주었고 20번을 넘게 돌 경우, cycle이 존재한다고 판단하였다.

4. ls

- 기존 ls함수에서는 symlink file의 stat이 제대로 출력되지 않은 것을 확인하였다.

`ln [-s] old new` 를 통해 생성된 파일은 inode number와 size가 달라야 하므로 다음과 과정을 통해서 수정하였다.

```

$ ln -s syncdatafile symlink
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat       2 3 16352
echo      2 4 15204
forktest  2 5 9520
grep      2 6 18572
init      2 7 15792
kill      2 8 15236
ln        2 9 15368
ls        2 10 17736
mkdir     2 11 15332
rm        2 12 15312
sh        2 13 27948
stressfs  2 14 16224
wc        2 15 17088
zombie    2 16 14904
test      2 17 17768
usertests 2 18 67400
symlinktest 2 19 19524
synctest  2 20 17984
bigfiletest 2 21 19744
console   3 22 0
syncdatafile 2 23 102400
symlink   2 23 102400

```

- 만약 출력하는 것이 t_symlink 파일일경우에 대해서 추가하였다.

```

...
case T_SYMLINK:
    printf(1, "%s %d %d %d\n", fmtname(path), st.type, st.ino, st.size);
    break;
...

```

- `stat(buf, &st)` 에서 ls의 하위 정보들을 가져오는 것을 확인하였고 이를 수정하면 symlink 파일의 inode number와 size등을 출력한다는 것을 확인하였다.
- stat을 호출하는 코드는 다음과 같다.

```

// st.type == T_DIR in ls.c
while(read(fd, &de, sizeof(de)) == sizeof(de)){
    if(de.inum == 0)
        continue;
    memmove(p, de.name, DIRSIZ);
    p[DIRSIZ] = 0;
    if(stat(buf, &st) < 0){
        printf(1, "ls: cannot stat %s\n", buf);
        continue;
    }
    printf(1, "%s %d %d %d\n", fmtname(buf), st.type, st.ino, st.size);
}

```

- stat 함수는 다음과 같다.

```

int
stat(const char *n, struct stat *st)
{
    int fd;
    int r;

    // T0-D0 : solve ls command for symlinkfile
    fd = open(n, 0);

```

```

if(fd < 0)
    return -1;
r = fstat(fd, st);
close(fd);
return r;
}

```

- 위의 `fd = open(n, 0)` 에서 `omode`를 `O_NOFOLLOW` mode를 넣어주는 것으로 수정하였다. 수정한 코드는 다음과 같다.

```

int
stat(const char *n, struct stat *st)
{
    int fd;
    int r;

    // T0-D0 : solve ls command for symlinkfile
    fd = open(n, O_NOFOLLOW);
    if(fd < 0)
        return -1;
    r = fstat(fd, st);
    close(fd);
    return r;
}

```

- `sys_open`도 다음과 같이 수정하였다. 만약 `open` mode가 `O_NOFOLLOW`가 아닌경우 원본 파일로 redirection을 시작한다.

```

if (!(omode & O_NOFOLLOW)) {
    // Variable named depth : check the cycle between the symlink files
    // If depth is greater than or equal to 20, It is determined that there is a cycle between symbolic link files.
    depth = 0;
    while(ip->type == T_SYMLINK && depth < 20) {
        ...
        depth++;
    }
    // Probably a cycle between symlink files
    if (ip->type == T_SYMLINK && depth >= 20) {
        iunlockput(ip);
        end_op();
        return -1;
    }
}
}

```

- 위에만 수정하면 `ls`가 작동하지 않는 문제가 발생하였고, 다음과 같은 코드를 추가해서 해결하였다. `omode`가 `O_RDONLY`, `O_NOFOLLOW`가 모두 아닐경우에만 오류를 발생하도록 수정함으로써 해결하였다.

```

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

```

- 수정한 코드를 바탕으로 같은 코드를 돌려보았을 때 다음과 같은 결과가 나옴을 확인하였고, 올바르게 구현되었음을 확인하였다.

```

$ ln -s syncdatafile symlink
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat       2 3 16352
echo      2 4 15204
forktest  2 5 9520
grep      2 6 18572
init      2 7 15792
kill      2 8 15236
ln        2 9 15368
ls        2 10 17736
mkdir     2 11 15332
rm        2 12 15312
sh        2 13 27948
stressfs  2 14 16224
wc        2 15 17088
zombie    2 16 14904
test      2 17 17768
usertests 2 18 67400
symlinktest 2 19 19524
synctest  2 20 17984
bigfiletest 2 21 19744
console   3 22 0
syncdatafile 2 23 102400
symlink   4 24 17

```

- stat만 주로 수정하고 ls는 기본 open omode는 수정하지 않았는데, 그 이유는 ls symlink을 하였을 때, symbolic link가 원본 파일을 잘 가리키고 있는지 확인하기 위해 수정을 하지 않았다.

```

$ ln -s syncdatafile symlink
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat       2 3 16352
echo      2 4 15204
forktest  2 5 9520
grep      2 6 18572
init      2 7 15792
kill      2 8 15236
ln        2 9 15368
ls        2 10 17736
mkdir     2 11 15332
rm        2 12 15312
sh        2 13 27948
stressfs  2 14 16224
wc        2 15 17088
zombie    2 16 14904
test      2 17 17768
usertests 2 18 67400
symlinktest 2 19 19524
synctest  2 20 17984
bigfiletest 2 21 19744
console   3 22 0
syncdatafile 2 23 102400
symlink   4 24 17
$ ls symlink
symlink   2 23 102400

```

- Is [symbolic link file]을 하였을 때 나오는 것은 원본파일을 출력하도록 하여서 symbolic link가 정상적으로 작동하였음을 확인 할 수 있었다.

3. Sync

3-1. Design

- 기존 xv6에는 group단위로 flush를 진행한다. 그 코드는 다음과 같다.

```
void
end_op(void)
{
    int do_commit = 0;

    acquire(&log.lock);
    log.outstanding -= 1;
    if(log.committing)
        panic("log.committing");
    if(log.outstanding == 0){
        do_commit = 1;
        log.committing = 1;
    } else {
        // begin_op() may be waiting for log space,
        // and decrementing log.outstanding has decreased
        // the amount of reserved space.
        wakeup(&log);
    }
    release(&log.lock);

    if(do_commit){
        // call commit w/o holding locks, since not allowed
        // to sleep with locks.
        commit();
        acquire(&log.lock);
        log.committing = 0;
        wakeup(&log);
        release(&log.lock);
    }
}
```

- outstanding이 0이 될 때 commit을 진행하는 형태이다. Disk I/O는 매우 느린 Operation이므로 이러한 부분을 개선시켜서 buffered I/O를 만드는 것이 목표이다.
- Begin_op의 기존 코드는 다음과 같다.

```
// called at the start of each FS system call.
void
begin_op(void)
{
    acquire(&log.lock);
    while(1){
        if(log.committing){
            sleep(&log, &log.lock);
        } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
            // this op might exhaust log space; wait for commit.
            sleep(&log, &log.lock);
        } else {
            log.outstanding += 1;
            release(&log.lock);
            break;
        }
    }
}
```

- begin_op에서는 buffer가 다 찼을 경우 다른 곳에서 commit 할때까지 기다린다.

sync를 만드는 것이 목적이고 sync는 user program이나 buffer가 가득 찼을 경우에 호출 되어질 수 있다.

3-2. Implementation

1. begin_op

- begin_op는 다음과 같이 수정하였다. buffer가 다 찼을 경우에는 sync를 호출하여 buffered i/o가 작동하도록 수정하였다.

```
void
begin_op(void)
{
    acquire(&log.lock);
    while(1){
        if(log.committing){
            sleep(&log, &log.lock);
        } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE - 1){
            // If buffer is full, then sync
            sync(1);
        } else {
            log.outstanding += 1;
            release(&log.lock);
            break;
        }
    }
}
```

2. end_op

- end_op는 commit하는 부분을 지웠고, outstanding이 0이면 자고있던 log를 깨워 sync가 작동할 수 있도록 하였다.

```
void
end_op(void)
{
    acquire(&log.lock);
    log.outstanding -= 1;
    if(log.outstanding < 0)
        panic("log.outstanding");

    // If outstanding of log is 0, wake up the log.
    if(log.outstanding == 0)
        wakeup(&log);
    release(&log.lock);
}
```

3. sync

- sync는 다음과 같이 작성하였다.

```
int
sync(int option)
{
    int fpage = -1;

    if (log.lh.n > 0) {
```

```

// [option 0] acquire log lock
if (!option)
    acquire(&log.lock);

while(log.outstanding > 0) {
    sleep(&log, &log.lock);
}

// If log is committing, Return -1
if (log.committing) {
    // [option 0] release log lock
    if (!option)
        release(&log.lock);
    return -1;
}

// Committing start and Set return value log.lh.n
log.committing = 1;
fpage = log.lh.n;

// Before call commit function, release log lock.
release(&log.lock);

// commit & set 0 committing var
// wake up lock
commit();
acquire(&log.lock);
log.committing = 0;
wakeup(&log);
// [option 0] release log lock
if(!option)
    release(&log.lock);
}
return fpage;
}

```

- 동작 과정은 다음과 같다.
 1. outstanding 0이 될 때까지 sleep을 시켜 end_op에서 0이 되어 깨울 때까지 재운다.
 2. 그 후 이미 committing이 수행중인지 체크하고 수행중이라면 -1을 반환한다. (commit 전에 lock을 해제하므로 committing중에 sync가 들어올 수 있다.)
 3. 반환 값 및 committing을 셋팅한다.
 4. commit 함수를 호출하고 committing 변수를 초기화하고 log를 깨운다. (for begin_op)
- option은 어디서 불렀는지에 따라 lock을 잡는다.
 - option이 1이면 begin_op에서 호출한것이므로 따로 lock을 잡지 않고 시작하며, option이 0인경우 밖에서 호출한 것이므로 lock을 잡고 마지막에 해제해야한다.
- sync는 parameter가 없으므로 system call wrapper function에 sync(0)을 전달 함으로써 해결하였다.

```

int
sys_sync(void)
{
    // The argument of sync function is used to hold the lock of log.
    // If value of argument is 0, Hold the lock of log.
    // Else If value of argument is 1, Do not have to hold the lock of log.
    return sync(0);
}

```

4. Result

4-1. multi indirect test

- usertests.c에서 big files test를 기반으로 테스트 코드를 작성하였다.

```
// bigfiletest.c
const int MiB = 2048;
int stdout = 1;
char buf[8192];

char name[7][15] = {
    "1MiBfile",
    "2MiBfile",
    "4MiBfile",
    "8MiBfile",
    "16MiBfile",
    "32MiBfile",
    "64MiBfile",
};

void
filetest(char *path, int size, int option)
{
    int i, fd, n;

    printf(stdout, "Start %d MiB file test\n", size / MiB);

    fd = open(path, O_CREATE|O_RDWR);
    if(fd < 0){
        printf(stdout, "[Error] creat failed!\n");
        exit();
    }

    for(i = 0; i < sizeof(buf); ++i) {
        buf[i]=i%26+97;
    }

    printf(stdout, "[Write start]\n");
    for(i = 0; i < size; i++){
        if(i % MiB == 0) {
            printf(stdout, "Write total %d MiB in file\n", i / MiB);
        }
        if(write(fd, buf, 512) != 512){
            printf(stdout, "[Error] write failed\n", i);
            exit();
        }
    }
    printf(stdout, "Write total %d MiB in file\n", size / MiB);
    close(fd);

    fd = open(path, O_RDONLY);
    if(fd < 0){
        printf(stdout, "[Error] open failed!\n");
        exit();
    }

    n = 0;

    printf(stdout, "[Read start]\n");
    for(;;){
        i = read(fd, buf, 512);
        if (n % MiB == 0){
            printf(stdout, "Read total %d MiB in file\n", n / MiB);
        }
        if(i == 0){
            if(n == size - 1){
                printf(stdout, "read only %d blocks from big", n);
                exit();
            }
            break;
        } else if(i != 512){
            printf(stdout, "[Error] read failed %d\n", i);
        }
        n += i;
    }
}
```

```

        exit();
    }
    n++;
}
if (sync() == -1) {
    printf(stdout, "[Error] sync failed\n");
    exit();
}
close(fd);
if(option && unlink(path) < 0){
    printf(stdout, "[Error] unlink failed\n");
    exit();
}
printf(stdout, "End %d MiB file test\n", size / MiB);
printf(stdout, "file size: %d bytes ok\n", size * 512);
}

void help()
{
    printf(stdout, "----- Test List ----- \n");
    printf(stdout, "- 0. 1 MiB file test \n");
    printf(stdout, "- 1. 2 MiB file test \n");
    printf(stdout, "- 2. 4 MiB file test \n");
    printf(stdout, "- 3. 8 MiB file test \n");
    printf(stdout, "- 4. 16 MiB file test \n");
    printf(stdout, "- 5. 32 MiB file test \n");
    printf(stdout, "- 6. 64 MiB file test \n");
    printf(stdout, "----- \n\n");
}

int pow2[] = {1, 2, 4, 8, 16, 32, 64};

int main(int argc, char *argv[])
{
    int n = 1, i;
    int cmd = 0;
    if (argc == 2) {
        cmd = argv[1][0] - '0';
    }

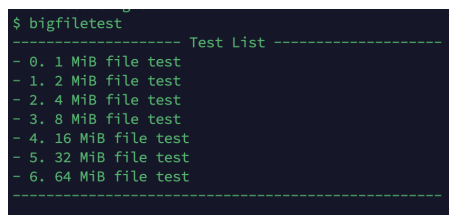
    help();

    for(i = 0; i < 7; i++, n *= 2) {
        printf(stdout, "[Test %d] %d MiB file test\n", i, n);
        filetest(name[i], n * MiB, cmd);
        printf(stdout, "[Test %d] %d MiB file ok\n\n", i, n);
    }

    printf(stdout, "All tests Passed\n");
    exit();
}

```

- 이 테스트 코드는 1, 2, 4, 8, 16, 32, 64 MiB 파일을 생성하는 코드이다. 테스트 결과는 다음과 같다.



```

$ bigfiletest
----- Test List -----
- 0. 1 MiB file test
- 1. 2 MiB file test
- 2. 4 MiB file test
- 3. 8 MiB file test
- 4. 16 MiB file test
- 5. 32 MiB file test
- 6. 64 MiB file test
-----

```

```
[Test 1] 2 MiB file test
Start 2 MiB file test
[Write start]
Write total 0 MiB in file
Write total 1 MiB in file
Write total 2 MiB in file
[Read start]
Read total 0 MiB in file
Read total 1 MiB in file
Read total 2 MiB in file
End 2 MiB file test
file size: 2097152 bytes ok
[Test 1] 2 MiB file ok
```

```
Test 3) 8 MIB file test
[Write start]
Write total 0 MIB in file
Write total 1 MIB in file
Write total 2 MIB in file
Write total 3 MIB in file
Write total 4 MIB in file
Write total 5 MIB in file
Write total 6 MIB in file
Write total 7 MIB in file
Write total 8 MIB in file
[Read start]
Read total 0 MIB in file
Read total 1 MIB in file
Read total 2 MIB in file
Read total 3 MIB in file
Read total 4 MIB in file
Read total 5 MIB in file
Read total 6 MIB in file
Read total 7 MIB in file
Read total 8 MIB in file
End 8 MIB file test
file size: 8386068 bytes ok
[Test 3] 8 MIB file ok
```

```
Test #4: 16 MB file test
Start 16 MB file test
[Write start]
Write total:0 MB in file
Write total 1 MB in file
Write total 2 MB in file
Write total 3 MB in file
Write total 4 MB in file
Write total 5 MB in file
Write total 6 MB in file
Write total 7 MB in file
Write total 8 MB in file
Write total 9 MB in file
Write total 10 MB in file
Write total 11 MB in file
Write total 12 MB in file
Write total 13 MB in file
Write total 14 MB in file
Write total 15 MB in file
Write total 16 MB in file
[Read start]
Read total 0 MB in file
Read total 1 MB in file
Read total 2 MB in file
Read total 3 MB in file
Read total 4 MB in file
Read total 5 MB in file
Read total 6 MB in file
Read total 7 MB in file
Read total 8 MB in file
Read total 9 MB in file
Read total 10 MB in file
Read total 11 MB in file
Read total 12 MB in file
Read total 13 MB in file
Read total 14 MB in file
Read total 15 MB in file
End 16 MB file test
Test #5: 16 MB file test
Test #4: 16 MB file ok
```

```
[Test 6] 64 MiB file test
Start 64 MiB file test
[Write start]
Write total 0 MiB in file
Write total 1 MiB in file
Write total 2 MiB in file
Write total 3 MiB in file
Write total 4 MiB in file
Write total 5 MiB in file
Write total 6 MiB in file
Write total 7 MiB in file
Write total 8 MiB in file
Write total 9 MiB in file
Write total 10 MiB in file
Write total 11 MiB in file
Write total 12 MiB in file
Write total 13 MiB in file
Write total 14 MiB in file
Write total 15 MiB in file
Write total 16 MiB in file
Write total 17 MiB in file
Write total 18 MiB in file
Write total 19 MiB in file
Write total 20 MiB in file
Write total 21 MiB in file
Write total 22 MiB in file
Write total 23 MiB in file
Write total 24 MiB in file
Write total 25 MiB in file
Write total 26 MiB in file
Write total 27 MiB in file
Write total 28 MiB in file
Write total 29 MiB in file
Write total 30 MiB in file
Write total 31 MiB in file
Write total 32 MiB in file
Write total 33 MiB in file
Write total 34 MiB in file
Write total 35 MiB in file
Write total 36 MiB in file
Write total 37 MiB in file
Write total 38 MiB in file
Write total 39 MiB in file
Write total 40 MiB in file
Write total 41 MiB in file
Write total 42 MiB in file
```

```

Read total 24 MiB in file
Read total 25 MiB in file
Read total 26 MiB in file
Read total 27 MiB in file
Read total 28 MiB in file
Read total 29 MiB in file
Read total 30 MiB in file
Read total 31 MiB in file
Read total 32 MiB in file
Read total 33 MiB in file
Read total 34 MiB in file
Read total 35 MiB in file
Read total 36 MiB in file
Read total 37 MiB in file
Read total 38 MiB in file
Read total 39 MiB in file
Read total 40 MiB in file
Read total 41 MiB in file
Read total 42 MiB in file
Read total 43 MiB in file
Read total 44 MiB in file
Read total 45 MiB in file
Read total 46 MiB in file
Read total 47 MiB in file
Read total 48 MiB in file
Read total 49 MiB in file
Read total 50 MiB in file
Read total 51 MiB in file
Read total 52 MiB in file
Read total 53 MiB in file
Read total 54 MiB in file
Read total 55 MiB in file
Read total 56 MiB in file
Read total 57 MiB in file
Read total 58 MiB in file
Read total 59 MiB in file
Read total 60 MiB in file
Read total 61 MiB in file
Read total 62 MiB in file
Read total 63 MiB in file
Read total 64 MiB in file
End 64 MiB file test
File size is 67108864 bytes ok
Test 64 MiB file ok

```

```
[Test 5] 32 MiB file test
Start 32 MiB file test
[Write start]
Write total 0 MiB in file
Write total 1 MiB in file
Write total 2 MiB in file
Write total 3 MiB in file
Write total 4 MiB in file
Write total 5 MiB in file
Write total 6 MiB in file
Write total 7 MiB in file
Write total 8 MiB in file
Write total 9 MiB in file
Write total 10 MiB in file
Write total 11 MiB in file
Write total 12 MiB in file
Write total 13 MiB in file
Write total 14 MiB in file
Write total 15 MiB in file
Write total 16 MiB in file
Write total 17 MiB in file
Write total 18 MiB in file
Write total 19 MiB in file
Write total 20 MiB in file
Write total 21 MiB in file
Write total 22 MiB in file
Write total 23 MiB in file
Write total 24 MiB in file
Write total 25 MiB in file
Write total 26 MiB in file
Write total 27 MiB in file
Write total 28 MiB in file
Write total 29 MiB in file
Write total 30 MiB in file
Write total 31 MiB in file
Write total 32 MiB in file
```

```
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 16352
echo       2 4 15204
forktest   2 5 9520
grep       2 6 18572
init       2 7 15792
kill       2 8 15236
ln         2 9 15420
ls         2 10 17720
mkdir      2 11 15332
rm         2 12 15312
sh         2 13 27948
stressfs   2 14 16224
wc         2 15 17088
zombie     2 16 14904
test       2 17 17768
usertests  2 18 67400
symlinktest 2 19 19524
syncstest  2 20 17984
bigfiletest 2 21 19744
console    3 22 0
1MiBfile   2 23 1048576
2MiBfile   2 24 2097152
4MiBfile   2 25 4194304
8MiBfile   2 26 8388608
16MiBfile  2 27 16777216
32MiBfile  2 28 33554432
64MiBfile  2 29 67108864
```

- 1 ~ 64 MiB파일이 성공적으로 잘 생성되었음을 확인하였다.

4-2. Symbolic link test

```
// symlinktest.c
#include "param.h"
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

#define BUFSIZE 100
#define TESTCNT 3

int symlinktest1();
int symlinktest2();
int symlinktest3();

int stdout = 1;
char buf[BUFSIZE];
char buf_link[BUFSIZE];

char *test_name_list[TESTCNT] = {
    "Simple Symlink",
    "Linked Symlink",
    "Miss Symlink"};

int (*test_list[TESTCNT])(void) = {
    symlinktest1,
    symlinktest2,
    symlinktest3};

int symlinktest1()
{
    int fd;

    fd = open("symlinkdummy", O_CREATE | O_RDWR);

    if (fd < 0)
    {
        printf(stdout, "error: create file for symbolic link failed!\n");
        exit();
    }
}
```

```

    }

    strcpy(buf, "Symbolic Test Running...");

    if (write(fd, buf, 512) != 512)
    {
        printf(stdout, "error: write file for symbolic link failed\n");
        exit();
    }

    sync();
    close(fd);

    symlink("symlinkdummy", "linksymlinkdummy");

    fd = open("linksymlinkdummy", O_RDONLY);

    read(fd, buf_link, sizeof(buf_link));

    close(fd);

    if (strcmp(buf, buf_link))
    {
        printf(2, "Test Fail...!\n[Original]: %s\n[Symbolic]: %s\n", buf, buf_link);
        return -1;
    }

    printf(2, " [Test Result]\n Original: %s\n Symbolic: %s\n", buf, buf_link);

    unlink("symlinkdummy");
    unlink("linksymlinkdummy");

    return 0;
}

int symlinktest2()
{
    int fd;
    char *file_name_list[5] = {
        "test1",
        "test2",
        "test3",
        "test4",
        "test5",
    };

    fd = open(file_name_list[0], O_CREATE | O_RDWR);

    if (fd < 0)
    {
        printf(stdout, "error: create file for symbolic link failed!\n");
        exit();
    }

    strcpy(buf, "Symbolic Recursive Test Success!!");

    if (write(fd, buf, 512) != 512)
    {
        printf(stdout, "error: write file for symbolic link failed\n");
        exit();
    }

    sync();
    read(fd, buf_link, sizeof(buf_link));
    close(fd);

    for (int i = 1; i < 5; ++i)
    {
        symlink(file_name_list[i - 1], file_name_list[i]);
    }

    fd = open(file_name_list[1], O_RDONLY);

```

```

    read(fd, buf_link, sizeof(buf_link));

    printf(2, "%s\n", buf_link);

    close(fd);

    if (strcmp(buf, buf_link))
    {
        printf(2, "Test Fail : [Original]: %s, \n[Symbolic]: %s\n", buf, buf_link);
        return -1;
    }
    printf(2, " [Test Result]\n    Original: %s\n    Symbolic: %s\n", buf, buf_link);

    for (int i = 0; i < 5; ++i)
    {
        unlink(file_name_list[i]);
    }
    return 0;
}

int symlinktest3()
{
    int fd;

    fd = open("missfile", O_CREATE | O_RDWR);

    if (fd < 0)
    {
        printf(stdout, "error: create file for symbolic link failed!\n");
        exit();
    }

    strcpy(buf, "Symbolic Missing Test Running...");

    if (write(fd, buf, 512) != 512)
    {
        printf(stdout, "error: write file for symbolic link failed\n");
        exit();
    }

    sync();
    close(fd);

    symlink("missfile", "linkfile");
    unlink("missfile");
    fd = open("linkfile", O_RDONLY);
    if (fd < 0)
    {
        unlink("linkfile");
        printf(2, " [Test Result]\n    Reference file is missed.\n", buf, buf_link);
        return 0;
    }

    return -1;
}

int main(int argc, char *argv[])
{
    printf(stdout, "soft symbolic test\n\n");

    for (int i = 0; i < TESTCNT; ++i)
    {
        printf(2, "[%s] Start\n", test_name_list[i]);
        if (test_list[i]() != 0)
        {
            printf(2, "%s failed\n", test_name_list[i]);
            exit();
        }
        printf(2, "[%s] Finish\n\n", test_name_list[i]);
    }

    exit();
}

```

- 테스트 목록은 다음과 같다.

1. 단순 symbolic link 테스트
2. 여러개의 symbolic link 파일이 연결되는 테스트
3. symbolic link의 원본 파일이 없어졌을 때의 테스트

```
$ symlinktest
soft symbolic test

[Simple Symlink] Start
[Test Result]
Original: Symbolic Test Running...
Symbolic: Symbolic Test Running...
[Simple Symlink] Finish

[Linked Symlink] Start
Symbolic Recursive Test Success!!
[Test Result]
Original: Symbolic Recursive Test Success!!
Symbolic: Symbolic Recursive Test Success!!
[Linked Symlink] Finish

[Miss Symlink] Start
[Test Result]
Reference file is missed.
[Miss Symlink] Finish
```

- ls 테스트, cat test 추가 예정

4-3. sync test

- sync를 확인하기 위해서 system call을 하나 추가하였다.

```
int
read_log(void)
{
    return log.lh.n;
}
```

- 테스트 코드는 다음과 같다.

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

```

#include "fs.h"
#include "fcntl.h"

#define FILESIZE      (200*512)
#define BUFFERSIZE    512
#define BUF_PER_FILE  ((FILESIZE) / (BUFFERSIZE))

int
main(int argc, char *argv[])
{
    char cmd = argv[1][0], data[BUFFERSIZE];
    int fd, i, cnt, ret_sync, old_log_num, now_log_num;

    for(i = 0; i < sizeof(data); ++i) {
        data[i]=i%26+97;
    }
    old_log_num = -1;

    printf(1, "Sync Test\n");
    switch (cmd)
    {
    case '1':
        /* code */
        printf(1, "[Test 1] Start buffered i/o test\n");
        fd = open("lostdatafile", O_CREATE | O_RDWR);
        for(i = 0; i < BUF_PER_FILE; i++) {
            if (i % 100 == 0) {
                printf(1, "[File %d] %d bytes written\n", fd, i * BUFFERSIZE);
            }
            if ((cnt = write(fd, data, sizeof(data))) != sizeof(data)) {
                printf(1, "[Error Test 1] Write returned %d\n", cnt);
                exit();
            }
            if ((now_log_num = read_log()) < 0) {
                printf(1, "[Error Test 1] read log : %d\n", now_log_num);
                exit();
            }
            printf(1, "[Log Info] %d to %d\n", old_log_num, now_log_num);
            old_log_num = now_log_num;
        }
        printf(1, "%d bytes written\n", BUF_PER_FILE * BUFFERSIZE);
        printf(1, "[Test 1] End buffered i/o test\n");
        close(fd);
        break;
    case '2':
        printf(1, "[Test 2] Start sync i/o test\n");
        fd = open("syncdatafile", O_CREATE | O_RDWR);
        for(i = 0; i < BUF_PER_FILE; i++) {
            if (i % 100 == 0) {
                printf(1, "[File %d] %d bytes written\n", fd, i * BUFFERSIZE);
            }
            if ((cnt = write(fd, data, sizeof(data))) != sizeof(data)) {
                printf(1, "[Write Error] Write returned %d\n", cnt);
                exit();
            }
            if ((old_log_num = read_log()) < 0) {
                printf(1, "[Log Error] read log : %d\n", old_log_num);
                exit();
            }
            if ((ret_sync = sync()) == -1) {
                printf(1, "[Sync Error] Sync failed\n");
                exit();
            }
            if ((now_log_num = read_log()) < 0) {
                printf(1, "[Log Error] read log : %d\n", now_log_num);
                exit();
            }
            printf(1, "[Log Info] %d to %d\n", old_log_num, now_log_num);
            printf(1, "Flushed %d of blocks\n", ret_sync);
        }
        printf(1, "%d bytes written\n", BUF_PER_FILE * BUFFERSIZE);
        printf(1, "[Test 2] End sync i/o test\n");
        close(fd);
    }
}

```



```

        break;
    default:
        printf(1, "WRONG CMD\n");
        break;
    }
    exit();
}

```

1. sync 하지 않고 종료되었을 때

- **synctest 1**으로 sync를 호출하지 않는 테스트 코드를 실행하였다.

```

$ synctest 1
Sync Test
[Test 1] Start buffered i/o test
[File 3] 0 bytes written

```

```

102400 bytes written
[Test 1] End buffered i/o test

```

- 종료 전 결과이다.

```

$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat       2 3 16352
echo      2 4 15204
forktest  2 5 9520
grep      2 6 18572
init      2 7 15792
kill      2 8 15236
ln        2 9 15368
ls        2 10 17736
mkdir     2 11 15332
rm        2 12 15312
sh        2 13 27948
stressfs  2 14 16224
wc        2 15 17088
zombie    2 16 14904
test      2 17 17768
usertests 2 18 67400
symlinktest 2 19 19524
synctest  2 20 17984
bigfiletest 2 21 19744
console   3 22 0
lostdatafile 2 23 102400

```

- 재부팅 후 결과이다.

```

$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat       2 3 16352
echo      2 4 15204
forktest  2 5 9520
grep      2 6 18572
init      2 7 15792
kill      2 8 15236
ln        2 9 15368
ls        2 10 17736
mkdir     2 11 15332
rm        2 12 15312
sh        2 13 27948
stressfs  2 14 16224
wc        2 15 17088
zombie    2 16 14904
test      2 17 17768
usertests 2 18 67400
symlinktest 2 19 19524
synctest  2 20 17984
bigfiletest 2 21 19744
console   3 22 0
lostdatafile 2 23 93184

```

- sync를 호출하지 않고 종료 후 qemu를 종료 후 다시 실행하였더니 file이 손실됨을 확인하였다. (102400 → 93184)

2. sync를 지속적으로 호출하는 test

- **synctest 2**으로 sync를 호출하는 테스트 코드를 실행하였다.

```

$ synctest 2
Sync Test
[Test 2] Start sync i/o test
[File 3] 0 bytes written

```

```

Flushed 4 of blocks
102400 bytes written
[Test 2] End sync i/o test

```

- 종료 전 결과이다.

- 재부팅 후 결과이다.

```
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 16352
echo       2 4 15204
forktest   2 5 9520
grep       2 6 18572
init       2 7 15792
kill       2 8 15236
ln         2 9 15368
ls         2 10 17736
mkdir      2 11 15332
rm         2 12 15312
sh         2 13 27948
stressfs   2 14 16224
wc         2 15 17088
zombie     2 16 14904
test       2 17 17768
usertests  2 18 67400
symlinktest 2 19 19524
synctest   2 20 17984
bigfiletest 2 21 19744
console    3 22 0
lostdatafile 2 23 93184
syncdatafile 2 24 102400
```

```
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 16352
echo       2 4 15204
forktest   2 5 9520
grep       2 6 18572
init       2 7 15792
kill       2 8 15236
ln         2 9 15368
ls         2 10 17736
mkdir      2 11 15332
rm         2 12 15312
sh         2 13 27948
stressfs   2 14 16224
wc         2 15 17088
zombie     2 16 14904
test       2 17 17768
usertests  2 18 67400
symlinktest 2 19 19524
synctest   2 20 17984
bigfiletest 2 21 19744
console    3 22 0
lostdatafile 2 23 93184
syncdatafile 2 24 102400
```

- sync를 호출한 파일은 꺾다켜도 file이 손실이 일어나지 않음을 확인하였다.

3. rm

나의 코드는 sync를 호출해야 반영이 된다 따라서 rm 명령어 작동후에도 sync가 호출되지 않았으면 파일이 지워지지 않는 것을 확인하였다.

- rm 수행후 종료 전 결과이다.
- 재부팅 후 결과이다.

```
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 16352
echo       2 4 15204
forktest   2 5 9520
grep       2 6 18572
init       2 7 15792
kill       2 8 15236
ln         2 9 15368
ls         2 10 17736
mkdir      2 11 15332
rm         2 12 15312
sh         2 13 27948
stressfs   2 14 16224
wc         2 15 17088
zombie     2 16 14904
test       2 17 17768
usertests  2 18 67400
symlinktest 2 19 19524
synctest   2 20 17984
bigfiletest 2 21 19744
console    3 22 0
lostdatafile 2 23 93184
syncdatafile 2 24 102400
```

```

$ rm syncdatafile
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 16352
echo       2 4 15204
forktest   2 5 9520
grep       2 6 18572
init       2 7 15792
kill       2 8 15236
ln         2 9 15368
ls         2 10 17736
mkdir      2 11 15332
rm         2 12 15312
sh         2 13 27948
stressfs   2 14 16224
wc         2 15 17088
zombie     2 16 14904
test       2 17 17768
usertests  2 18 67400
symlinktest 2 19 19524
syncctest  2 20 17984
bigfiletest 2 21 19744
console    3 22 0
lostdatafile 2 23 93184

```

- rm을 수행해도 sync가 호출되지 않았으면 파일이 지워지지 않았음을 확인하였다.

5. Trouble Shooting

5-1. Multi Indirect

1. inode, dinode struct

- 처음 구상했던 것은 direct와 single indirect 부분의 갯수는 그대로 나뉘게로 double, triple을 추가하려고 하였으나 그렇게 진행하니 assert가 나왔고 해당 부분을 찾았다.

```

// in mkfs.c
assert((BSIZE % sizeof(struct dinode)) == 0);
assert((BSIZE % sizeof(struct dirent)) == 0);

```

- 따라서 기존의 구조체를 사이즈를 유지한 채로 진행을 하였다. 그러기 위해서 NDIRECT의 개수를 2개 줄이고 해당 부분을 구현을 하였다.

2. param.h

- **FSSIZE**
 - 총 block의 개수는 MAXFILE인 2113674개이지만, FSSIZE를 적당히 50만개로 잡고 수행을 하였다.
- **MAXOPBLOCK**

- 64MiB Test를 하던 도중, FSSIZE는 충분하나 bread등의 panic이 발생하여서 다음과 같이 MAXOPBLOCKS를 늘려 buffer size를 늘려서 해결하였다.

```
#define MAXOPBLOCKS 20 // max # of blocks any FS op writes
#define LOGSIZE      (MAXOPBLOCKS*3) // max data blocks in on-disk log
#define NBUF          (MAXOPBLOCKS*3) // size of disk block cache
```

5-2. Symbolic Link

1. sys_symlink

- 처음에는 원본 파일의 path만 넣어주는 디자인으로 구현하였으나, 원본 파일을 얼마나 가져올 지 판단하지 못했고, 이로 인해서 제대로 symlink가 작동하지 않는 문제가 존재하였다. 따라서 원본 파일의 path를 넣어주기전에 원본 파일의 path의 길이를 미리 ip에 넣어줌으로써 해결하였다.

2. open

- TO-DO
 - open시 악의적인 경우로 symlink 파일들끼리의 cycle이 생길 수 있다고 판단하였고 이를 현재는 20번을 반복했는데, 좀 더 개선 사항이 있을 것 같다.

3. ls

- ls에서 symbolic link 파일을 만들어도 hard Link와 같이 원본 파일의 stat이 출력됨을 확인하였다. 이것을 위해서 설명한 것과 같이 stat function과 O_NOFOLLOW flag를 만들어 줌으로써 해결하였다. 기존 ls는 수정하지 않았는데 원본 파일과 잘 연결되었는지 확인하기 위해서 남겨두었다.

```
// in fcntl.h
#define O_RDONLY 0x000
#define O_WRONLY 0x001
#define O_RDWR 0x002
#define O_CREATE 0x200
#define O_NOFOLLOW 0x400
```

- open 수정

```
// sys_open in sysfile.c
if (!(omode & O_NOFOLLOW)) {
    ...
}
```

- stat 수정 (0에서 O_NOFOLLOW)로 수정

```
// TO-DO : solve ls command for symlinkfile
fd = open(n, O_NOFOLLOW);
if(fd < 0)
    return -1;
r = fstat(fd, st);
close(fd);
return r;
```

- DIR인 경우, O_NOFOLLOW flag이면 error가 나온 것을 확인하였고, NOFOLLOW인 경우도 DIR일 때도 작동하도록 수정하였다.

```

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

```

5-3. Sync

1. begin_op

- 다량, 큰 파일을 테스트 하던 도중 kernel쪽에서 에러가 났음을 확인하였고, 이는 공간이 없어서 kernel이 작동을 하지 않아 문제가 생겼음을 알게 되었고, 다음과 같이 수정하였다.

- `log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE-1`

2. sync

- 처음에는 sync시작할 때 lock을 잡고 시작하기 위해서 argument를 주어서 begin_op에서 수행하는지 그 외에서 수행하는지 판단하도록 하였고 상황에 맞춰서 lock을 잡고 끝나기 전에 해제하도록 하였다.
- commit 함수 호출 전 release를 시키는데 이 부분에서 다른 sync함수가 들어 올 수 있음을 생각하였고 다음 부분을 추가하여 예외처리를 하였다.

```

// If log is committing, Return -1
if (log.committing) {
    // [option 0] release log lock
    if (!option)
        release(&log.lock);
    return -1;
}

log.committing = 1;

```