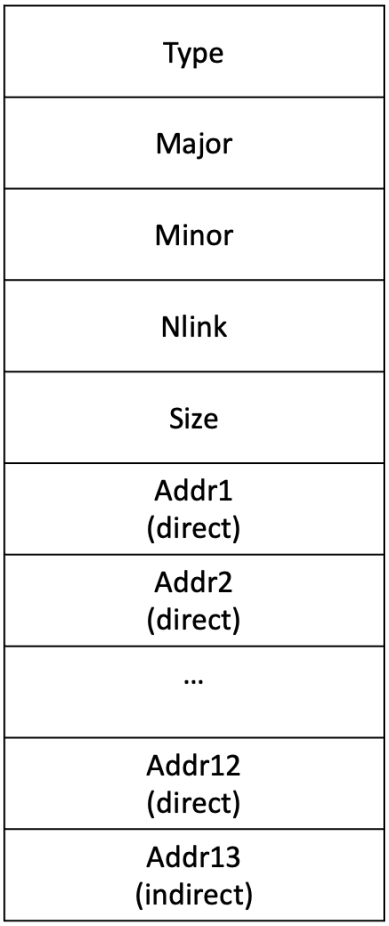
운영체제 Project 03 Wiki

1. Design

1) Multi Indirect

- 처음 xv6에서 구현되어 있는 **data blcok allocation 방식은 indexed allocation 방식**을 사용한다.



xv6 data block allocation

- 해당 **data block allocation 방식**을, 아래와 같은 그림으로 바꾸어 줄 것이다.

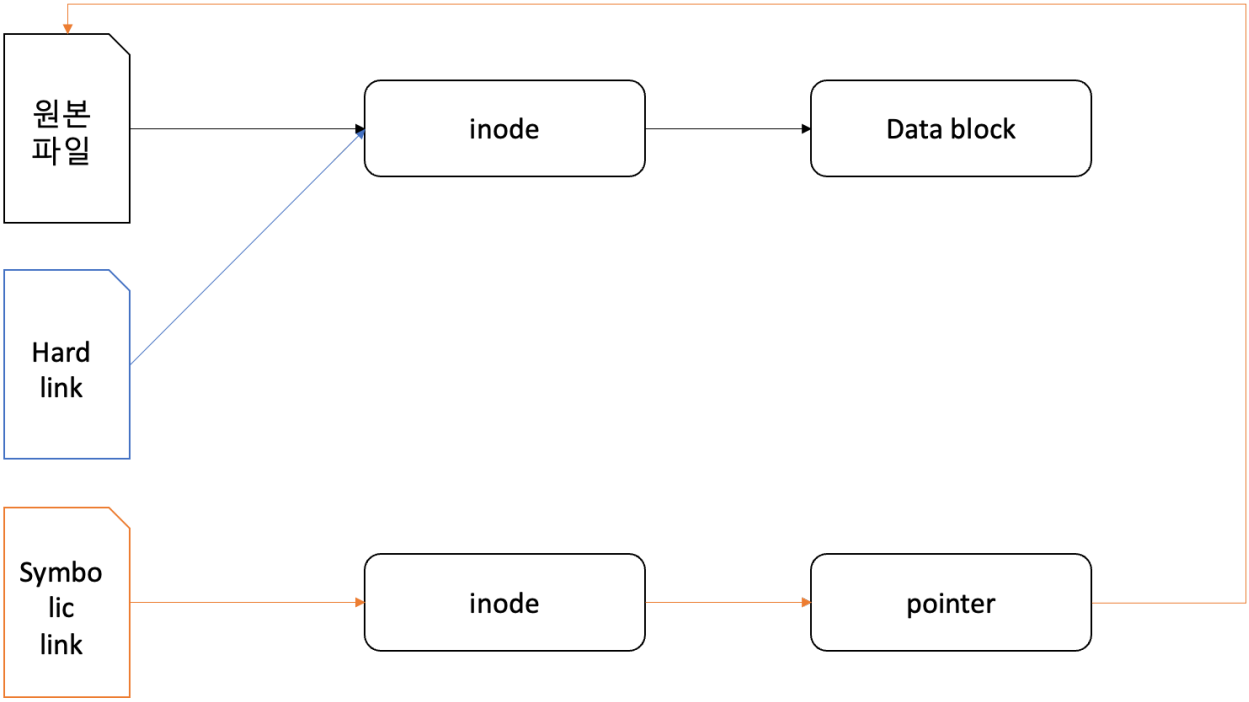
Type
Major
Minor
Nlink
Size
Addr1 (direct)
...
Addr10 (direct)
Addr11 (single indirect)
Addr12 (double indirect)
Addr13 (triple indirect)

수정된 data block allocation

- 이를 위해서는 **data block을 mapping해주는 bmap 함수와, 데이터 블럭을 해제 해주는 itrunc함수**를 수정 할 것이다.
- 또한, **inode 구조체, dinode 구조체**를 수정할 것 이다.

2) Symbolic Link

a) structure



ln linux link 구조

- linux에서의 hard link와 symbolic link구조를 그대로 차용 할 예정이다.
- hard link는 **원본파일의 inode를 그대로 참조하며**, inode에 대한 reference 숫자로 해당 inode를 참조하는 파일의 개수를 알 수 있는 방식이다.
- 하지만 symbolic link는 이와 다른 방식으로, **새로 만든 inode를 참조하며** 해당 inode에서 원본 파일을 가르키는 pointer을 통해서 redirecting하는 방식을 사용한다.
- 이와 같은 특징 때문에, symbolic link에 대한 접근은 내부에서의 redirecting을 통해 **원본 파일에 대한 접근이 되게 된다.**
- 이때, **file의 name을 통해서 pointing하는 방식이며**, 그 때문에 원본 파일의 name이 바뀌게 된다면 link가 제대로 동작하지 않는다.
- xv6에서의 hard link는 linux의 동작 방식과 유사하게 구현 되어 있으며, **새로운 inode를 가지고 원본 파일을 pointing하는 symbolic link를 새로 구현 할 것 이다.**

b) redirecting

- redirecting을 어디서 해주어야 하는 지도 추가적인 문제였다.
- 결론적으로 떠올린 2가지 방법이 있었다.
    - open, exec할 때 (symfile이 열린 순간)

    - readi, writei할 때 (file 내부에 접근하려 할 때)
- exec과 open에서 redirecting 했을 때의 문제점은 symbolic file의 metadata에도 접근을 할 수 없다는 점이었다.
- 그래서, **readi와 writei에서 redirecting**하는 방식을 차용할 것이다.

3) Sync

a) buffer I/O

Begin_op (process 1, outstanding = 1)
Begin_op (process 2, outstanding = 2)
End_op (process 1, outstanding = 1)
Begin_op (process 3, outstanding = 2)
End_op (process 2, outstanding = 1)
End_op (process 3, outstanding = 0)
Commit (grouping)

xv6의 group commit

- 기존 xv6의 commit 방식은, group commit 방식을 사용한다.
- 위의 그림과 같이, begin_op를 호출하면 I/O operation을 수행하는 프로세스의 개수를 늘려주고, end_op를 호출하면 I/O operation을 수행하는 프로세스의 개수를 줄여준다.
- 이때, 해당 **outstanding 변수가 0이 되는 상황에서 commit**이 발생한다.
- group commit 대신, sync를 호출 했을 때만 commit을 진행하게 해줄 것이다.
- **begin_op와 end_op가 기존에 수행하던 주된 역할은 outstanding의 개수를 세어주는 역할**이었다. buffer I/O를 위해 해당 역할을 제거하여 group commit를 제거 할 것이다.
- 또한, sync로 인한 commit 말고도 buffer가 더이상 가득 찼을 때, commit을 진행해 주어야 한다.

b) full buffer

Buffer cache	1(Dirty)	2(dirty)	3(dirty)	4(dirty)	5(dirty)

Log	1,2,3,4,5	1	2	3	4	5

buufer full == log full (log size = 5 && buffer size = 5)

- bget에서 **실제 buffer 공간을 할당 받기 전에, 예외처리를 해주어, commit**을 진행 해 줄 것이다.
- 이때, 예외 처리를 **log의 size를 보고 수행해 줄 것이다**.
  - log size와 buffer가 꽉 찬 것은 동일하기 때문이다.
  - 해당 근거는 buffer가 교체 기법 (LRU)를 사용해서도 buffer의 공간을 할당 할 수 없을 때, 즉 모든 buffer가 dirty bit로 설정된 경우 buffer의 공간이 부족한 것이다.
  - **모든 buffer가 dirty bit라면 log도 full인 것이기에 buffer가 full인 것과 buffer가 full인 것을 동일하게 생각하였다.**
- 그래서 해당 상황에서 **sync를 호출하여 commit**을 진행해 줄 것이다.

2. Implement

1) Multiple Indirect

a) dinode

```
#define NDIRECT 10
#define NINDIRECT (BSIZE / sizeof(uint)) //128개
#define D_INDIRECT (NINDIRECT * NINDIRECT) //128*128개
#define T_INDIRECT (D_INDIRECT * NINDIRECT) //128*128*128개
#define MAXFILE (NDIRECT + NINDIRECT + D_INDIRECT + T_INDIRECT)

// 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
};
```

- triple, double indirect를 구현하기 위해서 **direct block의 개수를 10개**로 만들어 주어, dobule과, triple indirect를 위한 block 크기 공간을 만들어 주었다.

b) inode

```
// 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+3];
};
```

- NDIRECT의 크기를 10으로 변경해 주었기 때문에, addrs 배열의 크기를 **NDIRECT + 3** 으로 변경해 주었다.

c) bmap

```
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;
  }
  //indirect
  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;
  }

  // double indirect
  bn -= NINDIRECT;
  if(bn < D_INDIRECT){
    if((addr = ip->addrs[NDIRECT + 1]) == 0){
      ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
    }
    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);

    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;
  }
  // triple indirect
  bn -= D_INDIRECT;
  if(bn < T_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/D_INDIRECT]) == 0){
      a[bn/D_INDIRECT] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
```

```
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[(bn%D_INDIRECT)/NINDIRECT]) == 0){
      a[(bn%D_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");
}
```

- bmap함수는 **data block을 mapping**해주는 역할을 한다.
- double indirect와 triple indirect를 구현하기 위해서, xv6에서 구현 되어 있던 **single indirect 매핑 방식을 차용하였다.**
- 한가지 다른 점이 있다면, double, triple indirect 방식은, **single indirect가 여러번 수행되는 것과 마찬가지 이기 때문에, divide연산과 modular 연산을 사용해서 정확한 위치를 찾아주어 매핑 해 주었다.**
- 해당 방식을 통해서 **총 (128*128*128 + 128*128 + 128 + 10)의 데이터 블럭을** 할당 할 수 있는 구조가 만들어졌다.

d) Itrunc

```
static void
itrunc(struct inode *ip)
{
  int i, j, k, p;
  struct buf *bp, *bp_1, *bp_2;
  uint *a, *a_2, *a_3;

  for(i = 0; i < NDIRECT; i++){
    if(ip->addrs[i]){
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
```

```
    }
  }

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

  //double indirect
  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]){
      bp_1 = bread(ip->dev, a[j]);
      a_2 = (uint*)bp_1->data;

      for(k = 0; k < NINDIRECT; k++){
        if(a_2[k]){
          bfree(ip->dev, a_2[k]);
        }
      }
      brelse(bp_1);
      bfree(ip->dev, a[j]);
      }
    }

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

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

    for(j = 0; j < NINDIRECT; j++){
      if(a[j]){
        bp_1 = bread(ip->dev, a[j]);
        a_2 = (uint*)bp_1->data;

        for(k = 0; k < NINDIRECT; k++){
```

```
          if(a_2[k]) {
            bp_2 = bread(ip->dev, a_2[k]);
            a_3 = (uint*)bp_2->data;

            for(p = 0; p < NINDIRECT; p++){
              if(a_3[p]){
                bfree(ip->dev, a_3[p]);
              }
            }

          brelse(bp_2);
          bfree(ip->dev, a_2[k]);
         }
        }

        brelse(bp_1);
        bfree(ip->dev, a[j]);
      }
    }

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

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

- itrunc는 **data block 할당을 해제해주는 역할**을 담당하는 역할을 한다.
- bmap과 마찬가지로 기존의 single indirect에서 사용하던 방식을 차용하였다.
- 다른 점은, double, triple indirect는 single indirect가 여러번 반복되는 방식이기에, **for문 이중, 삼중으로 사용해주어 정확한 data block을 찾아 주었다.**

2) Symbolic link

a) stat.h

```
#define T_DIR  1   // Director
#define T_FILE 2   // File
#define T_DEV  3   // Device
#define T_SYMLINK 4 // Symlink
```

- symbolic file을 관리해주기 위해서 **file type, T_SYMLINK type**을 추가해 주었다.

b) sys_symlink

```
int
sys_symlink(void)
{
  char *old, *new;
  char name[DIRSIZ];
  struct inode *temp, *dp;
  int fd, omode;
  struct file *f;

  if(argstr(0, &old) < 0 || argstr(1, &new) < 0){
    return -1;
  }
  cprintf("creating a sym link. old : %s new : %s\n", old, new);

  begin_op();

  //link 할 파일이 존재하는지 확인
  if((temp = namei(old)) == 0){
    cprintf("Not exist old file!\n");
    end_op();
    return -1;
  }

  //새로운 파일의 이름과 겹치는 것이 있는지 확인
  if((temp = namei(new)) != 0){
    cprintf("Already exist new file!\n");
    end_op();
    return -1;
  }

  //ip 할당
  struct inode *ip = create(new, T_FILE, 0, 0);
  if(ip == 0){
    end_op();
    return -1;
  }

  //새로 만든 ip의 data block에 데이터를 써준다.
  int len = strlen(old);
  char len_arr[sizeof(int)];
  memcpy(len_arr, &len, sizeof(int));
  writei(ip, len_arr, 0, sizeof(int));
  writei(ip, old, sizeof(int), len+1);
  ip->type = T_SYMLINK;

  iupdate(ip);
```

```
  iunlock(ip);
  end_op();
  return 0;
}
```

- symbolink를 수행하기 위해서 새로운 system call인 sys_symlink를 추가해 주었다.
- 해당 함수가 하는 일은 다음과 같다.
  - **link할 파일(old)가 존재하는지 확인한다.**
  - **새로 만들 파일(new)와 이름이 겹치는 파일이 있는지 확인한다.**
  - **create함수를 통해서 새로 inode를 할당 해주고, 해당 inode data block에 link 할 file 이름과, link 할 file 이름의 size를 저장해준다.**
  - **그런후 새로 만든 inode의 type을 T_SYMLINK로 저장해준후, iupdate를 통해 dinode에 update 해준다.**

c) readi && writei

```
int count = 0;
 //readi를 하는 주체를 바꾸어줌.
 while ((ip->type == T_SYMLINK && count <=20)){
     count++;
     struct inode *n_ip;
     ip->type = T_FILE;
     char len_arr[5];
     readi(ip, len_arr, 0, sizeof(int));
     int t;
     memcpy(&t, len_arr, sizeof(int));
     readi(ip, &path, sizeof(int), t+1);
     ip->type = T_SYMLINK;
     if((n_ip = namei(&path)) == 0){
       end_op();
       return -1;
     }
     else{
       ip = n_ip;
       if(ip == temp || count > 20){
         cprintf("There is a link cycle!\n");
         return -1;
       }
       ilock(ip);
       iunlock(ip);
```

```
    }
 }
```

- readi와 writei의 시작 부분에서, 만약 inode의 type이 T_SYMLINK라면 해당 inode의 data block에서 원본 파일의 file name 길이와, file name을 읽어 오도록 해주었다.
- 읽어온 **file name을 통해서, namei 함수를 호출하여 원본 파일의 ip**를 찾아주었다.
- 이때, symbolic link file을 계속해서 link하는 상황을 처리해주기 위해서, **count 변수와, while문을 통해서 총 20개의 연속적인 link를 처리 할 수 있게 해주었다.**
- 또한, **link파일 끼리 서로 link를 하는 "cycle"**이 생길 수 있다는 점에서 착안하여, 만약, 찾은 원본 파일의 ip가 symbolic file의 ip와 같다면 종료하게 예외처리 해주었다.

3) Sync

a) begin_op

```
void
begin_op(void)
{
  acquire(&log.lock);
  while(1){
    if(log.committing){
      sleep(&log, &log.lock);
    } else {
      release(&log.lock);
      break;
    }
  }
}
```

- begin_op에서 원래 group commit을 수행하기 위해 outstanding을 counting 해주던 부분을 삭제해 주었다.
- 사실, 이제 큰 의미가 없는 함수 이지만, **만약 commit을 수행중이라면 operation을 수행하면 안되기 때문에, log.commiting 변수가 1일 때, log를 sleep하게 해주었다.**

b) end_op

```
void
end_op(void)
```

```
{
  acquire(&log.lock);
  wakeup(&log);
  release(&log.lock);
}
```

- end_op에서도 begin_op와 마찬가지로 **group commit을 수행하기 위해 outstanding의 숫자를 보고 commit**을 수행해주던 부분을 삭제하였다.

c) sync

```
//sync
int sync(void){
  acquire(&log.lock);
  //아직 buffer가 안찼을 때
  if(check == 1){
    if (log.lh.n < LOGSIZE- MAXOPBLOCKS){
      release(&log.lock);
      return -1;
    }
  }
  //committing인 경우
  if(log.committing){
    release(&log.lock);
    return -1;
  }
  log.committing = 1;
  //commit 실행
  int temp = log.lh.n;
  release(&log.lock);
  commit();
  acquire(&log.lock);
  log.committing = 0;
  wakeup(&log);
  release(&log.lock);
  return temp;
}
```

- sync함수에서 하는 일은 다음과 같다.
  - **buffer가 full 상태 인지 아닌지 확인해준다.**
  - **만약, 현재 commit을 수행중인지 확인해준다.**
  - **temp 변수에 현재 log에 들어있는 개수를 저장해준다.**

    - **commit을 수행해준후, temp를 return해준다.**
- 이때, 항상 bget을 수행 할 때 마다 sync를 호출 해 주기에, check 변수가 1 (bget에서의 호출) 일 때, **log가 full인지 확인 해주고 그렇지 않다면 바로 return하게 해주었다.**
- bget에서 sync를 호출 한 것이 아니라면(user process에서 호출), **무조건 commit을 진행하게 해주었다.**

d) bget

```
//sync 부터 호출하여 buffer가 꽉 찼는지 확인
check = 1;
sync();
check = 0;
```

- 나의 구현에서는 buffer size check를 sync에서 수행해 준다.
- 이렇게 구현 했을 때의 문제점은 user process에서 sync를 호출 했는지, bget에서 sync를 호출 했는지 확인 할 방법이 없다는 점이였다.
- 그래서 sync를 호출하기 전에 **extern 변수 check를 1로 만들고 호출하여, sync에서 예외처리 해주게 하였다.**
- buffer size를 buffer를 할당 받기 전에 하기 때문에, **buffer를 할당 받기 위해 bget에 들어왔지만, 만약 가용한 buffer가 없다면 commit 후에 buffer를 할당 받기에 무조건 buffer**를 할당 받을 수 있다.

3. Result

1) Symbolic link

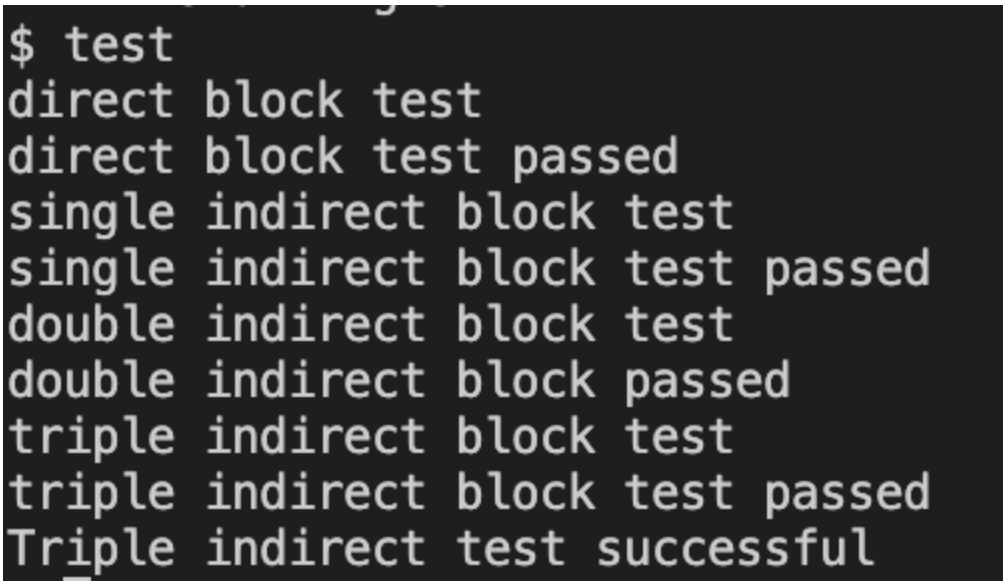
```
#include "param.h"
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fs.h"
#include "fcntl.h"
#include "syscall.h"
#include "traps.h"
#include "memlayout.h"
```

```
// 테스트 함수
void triple_indirect_test(void)
{
  char filename[] = "triple_indirect_test_file";

  // 파일 생성
  int fd = open(filename, O_RDWR | O_CREATE);
  if (fd < 0) {
    printf(2, "Failed to create file\n");
    return;
  }

  char buf[BSIZE];
  memset(buf, 'A', BSIZE);

  // 파일에 데이터 블록 할당 (triple indirect)
  int i, j, k;

  printf(2, "direct block test\n");
  for (i = 0; i < NDIRECT; i++) {
    if (write(fd, buf, BSIZE) != BSIZE) {
      printf(2, "Failed to write data block\n");
      close(fd);
      return;
    }
  }
  printf(2, "direct block test passed\n");

  // Single indirect block
  printf(2, "single indirect block test\n");
  for (i = 0; i < NINDIRECT; i++) {
    if (write(fd, buf, BSIZE) != BSIZE) {
      printf(2, "Failed to write data block\n");
      close(fd);
      return;
    }
  }
  printf(2, "single indirect block test passed\n");

  // Double indirect block
  printf(2, "double indirect block test\n");
  for (i = 0; i < NINDIRECT; i++) {
    int block_addr = 0;
    // printf(2, "double now %d\n", i);

    if (write(fd, &block_addr, sizeof(int)) != sizeof(int)) {
      printf(2, "Failed to write double indirect block address\n");
      close(fd);
      return;
    }
    for (j = 0; j < NINDIRECT; j++) {
      if (write(fd, buf, BSIZE) != BSIZE) {
```

```
        printf(2, "Failed to write data block\n");
        close(fd);
        return;
      }
    }
  }

  printf(2, "double indirect block passed\n");

  // Triple indirect block
  printf(2, "triple indirect block test\n");
  for (i = 0; i < NINDIRECT; i++) {
    int block_addr = 0;

    if (write(fd, &block_addr, sizeof(int)) != sizeof(int)) {
      printf(2, "Failed to write triple indirect block address\n");
      close(fd);
      return;
    }

    for (j = 0; j < NINDIRECT; j++) {
      int indirect_block_addr = 0;

      if (write(fd, &indirect_block_addr, sizeof(int)) != sizeof(int)) {
        printf(2, "Failed to write indirect block address\n");
        close(fd);
        return;
      }

      for (k = 0; k < NINDIRECT; k++) {
        if (write(fd, buf, BSIZE) != BSIZE) {
          printf(2, "Failed to write data block\n");
          close(fd);
          return;
        }
      }
    }
  }
  printf(2, "triple indirect block test passed\n");

  close(fd);

  printf(1, "Triple indirect test successful\n");
}

int main(void)
{
  triple_indirect_test();
```

```
  exit();
}
```

- triple indirect를 수행하기 위해서 해당 test case를 수행하였다.
- **총 1GB의 write**를 수행하는 test case이다.



성공적으로 수행되었다.

2) Symbolic link

a) basic link

```
$ ln -s test test1
creating a sym link. old : test new : test1
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15564
echo           2 4 14444
forktest       2 5 8880
grep           2 6 18400
init           2 7 15068
kill           2 8 14528
ln             2 9 14740
ls             2 10 16996
mkdir          2 11 14552
rm             2 12 14592
sh             2 13 28584
stressfs       2 14 15460
usertests      2 15 63124
wc             2 16 15980
zombie         2 17 14100
test           2 18 17008
test2          2 19 63024
console        3 20 0
test1          4 21 9
```

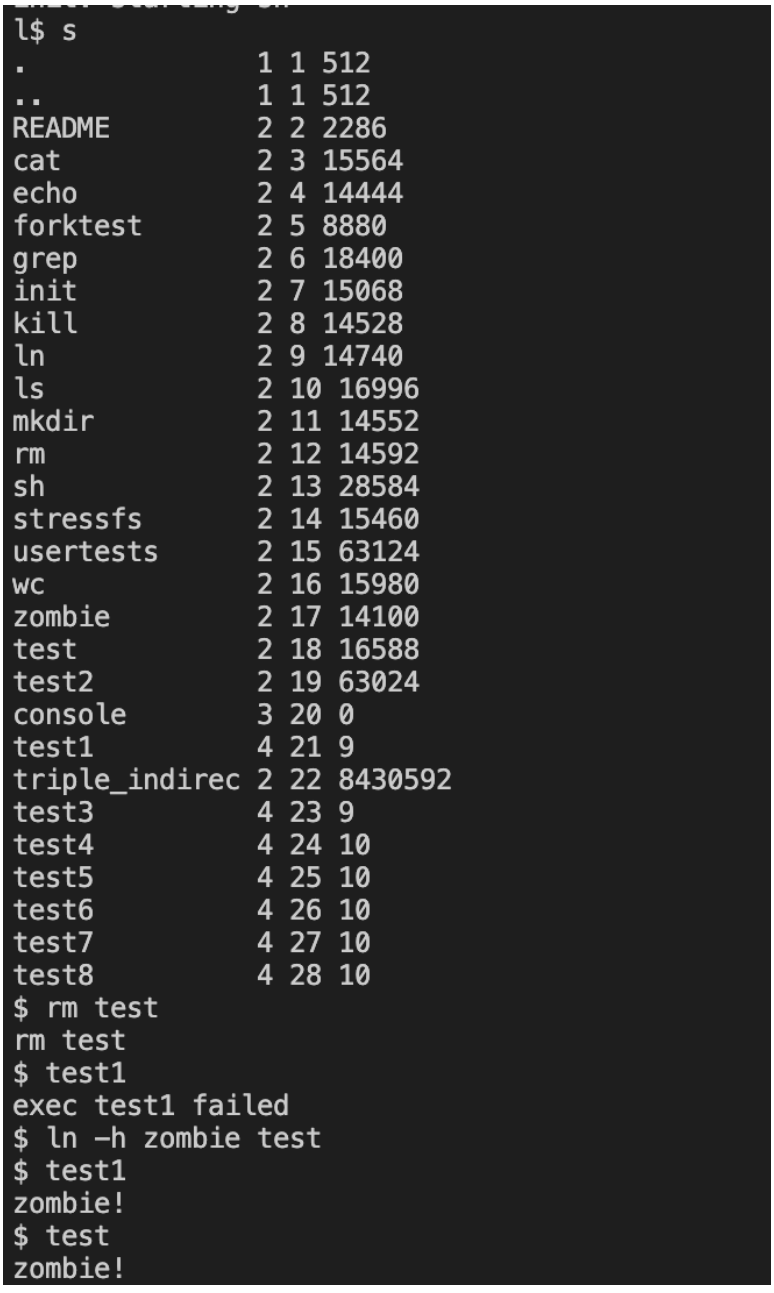
정상적으로 만들어진 link file

```
$ test1
direct block test
direct block test passed
single indirect block test
single indirect block test passed
double indirect block test
double indirect block passed
triple indirect block test
triple indirect block test passed
Triple indirect test successful
```

정상적으로 수행되는 모습

- 새로 만든 symbolic link file의 type은 T_SYMLINK (4)로 정상적으로 나온다.
- 또한, 해당 파일의 data block에는 원본 파일의 name, 원본 파일의 name 길이가 저장되어 있기에 총 9 byte의 크기가 나온다.
  - **name 길이 → int (4 byte)**
  - **원본 파일 name → ‘t’ + ‘e’ + ‘s’ + ‘t’ + ‘\0’ (5 byte)**

b) Delete original file



```
l$ s
.               1 1 512
..              1 1 512
README          2 2 2286
cat             2 3 15564
echo            2 4 14444
forktest        2 5 8880
grep            2 6 18400
init            2 7 15068
kill            2 8 14528
ln              2 9 14740
ls              2 10 16996
mkdir           2 11 14552
rm              2 12 14592
sh              2 13 28584
stressfs        2 14 15460
usertests       2 15 63124
wc              2 16 15980
zombie          2 17 14100
test            2 18 16588
test2           2 19 63024
console         3 20 0
test1           4 21 9
triple_indirec 2 22 8430592
test3           4 23 9
test4           4 24 10
test5           4 25 10
test6           4 26 10
test7           4 27 10
test8           4 28 10
$ rm test
rm test
$ test1
exec test1 failed
$ ln -h zombie test
$ test1
zombie!
$ test
zombie!
```

원본 파일 delete 후 같은 이름으로 재생성

- symbolic file test1이 link하고 있는 test를 삭제 후 test1을 실행하면 file name으로 리다이렉팅 하는 symbolic file이기에 실행이 실패한다.
- 이때, 만약 link하고 있는 **file name과 동일하게 다른 file을 만들어 주게 된다면, 정상적으로 리다이렉팅이 성공하는 모습이다.**

- 요약하자면, **원본파일 ‘test’를 삭제 후 ‘test’의 이름으로 zombie file을 hard link 해주었다.**

c) link and link (symlink)

```
$ ln -s test test3
creating a sym link. old : test new : test3
l$ n -s test3 test4
creating a sym link. old : test3 new : test4
$ ln -s test4 test5
creating a sym link. old : test4 new : test5
$ ln -s test5 test6
creating a sym link. old : test5 new : test6
$ ln -s test6 test7
creating a sym link. old : test6 new : test7
$ ln -s test7 test8
creating a sym link. old : test7 new : test8
$ test8
direct block test
direct block test passed
single indirect block test
single indirect block test passed
double indirect block test
```

link의 link (symbolic 만)

```
$ test7
direct block test
direct block test passed
single indirect block test
single indirect block test passed
double indirect block test
```

```
$ test6
direct block test
direct block test passed
single indirect block test
single indirect block test passed
double indirect block test
```

```
$ test5
direct block test
direct block test passed
single indirect block test
single indirect block test passed
double indirect block test
```

- symbolic link file을 계속해서 link 해주었다.
- 20번의 redirecting까지는 **허용하게 구현하였기 때문에 정상적으로 수행되는 모습이다.**

d) link and link (hardlink)

```
$ ln -h test8 test9
$ ln -s test9 test10
creating a sym link. old : test9 new : test10
$ ln -h test10 test11
$ ln -s test11 test12
creating a sym link. old : test11 new : test12
$ test12
direct block test
direct block test passed
single indirect block test
```

hardlink와 섞어서 수행

- hardlink와 symbolic link를 섞어서 **link를 진행해 보았다.**
- 결과는 성공적으로 수행 되었다.

e) link cycle

```
$ ln -s test c_test1
creating a sym link. old : test new : c_test1
$ ln -s c_test1 c_test2
creating a sym link. old : c_test1 new : c_test2
$ rm c_test1
rm c_test1
$ ln -s c_test2 c_test1
creating a sym link. old : c_test2 new : c_test1
```

cycle 생성

```
$ c_test1
There is a link cycle!
exec c_test1 failed
```

cycle시 수행 실패

```
$ c_test2
There is a link cycle!
exec c_test2 failed
```

cycle시 수행 실패

- symbolic link를 이용하여 cycle을 만들어 보았다.
- **서로가 서로를 redirecting하는 상황이기에, exec은 실패되어야 하고**, 정상적으로 감지하여 오류 메시지를 출력하였다.

f) file name error

```
$ ln -s notexist test3
creating a sym link. old : notexist new : test3
Not exist old file!
symbolic link -s notexist: failed
```

존재하지 않는 file에 대한 link

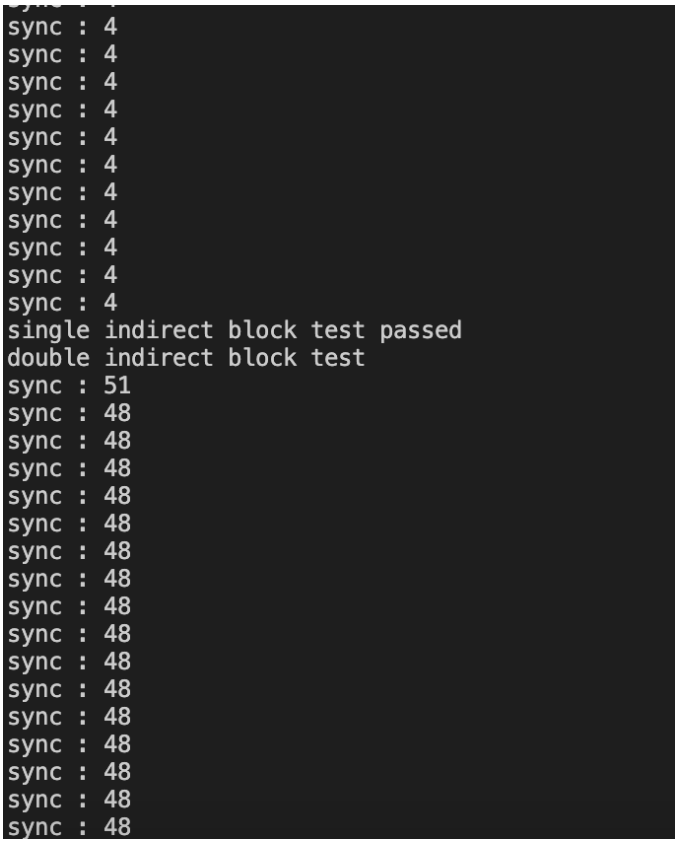
```
$ ln -s zombie test
creating a sym link. old : zombie new : test
Already exist new file!
symbolic link -s zombie: failed
```

이미 존재하는 file이름으로 link file 생성

- file name으로 인한 예외처리를 위해 2가지 test를 진행하였다.
- **첫 번째 test는 존재하지 않는 file에 대한 link를 요청 하였을 때다.**
  - 정상적으로 error 메시지를 출력 후 file이 생성되지 않는다.
- **두 번째 test는 이미 존재하는 file name으로 link file을 생성하려고 할 때이다.**
  - 정상적으로 error 메시지를 출력 후 file이 생성되지 않는다.

3) Sync

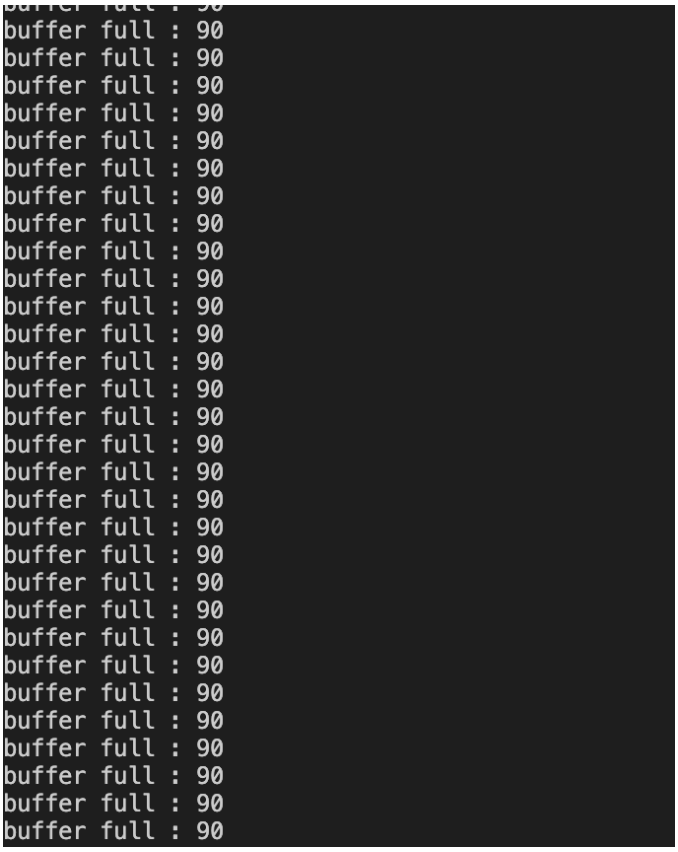
a) call sync



call sync

- triple indirect test에서 사용한 **test case에서 주기적으로 sync를 호출하고 commit한 data block을 출력하도록 수정해 주었다.**
- 정상적으로 sync가 수행되며, 값이 출력되는 것을 확인 할 수 있다.

b) buffer full

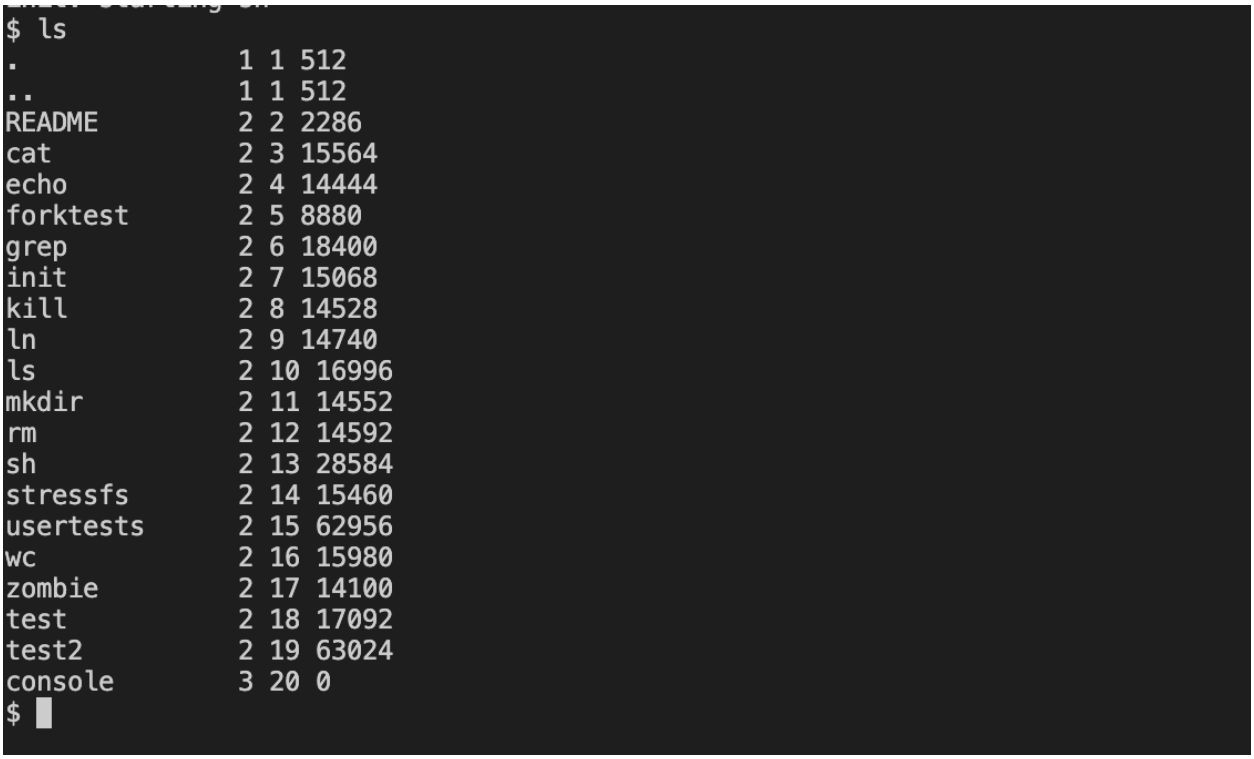


buffer full

- test case에서 sync를 호출 하지 않고, commit하기 전에, **commit 할 data block의 개수를 출력하게 해주었다.**
- 정상적으로 계속해서 90의 출력이 나오게 된다.
    - log size는 100이지만, MAXOPBLOCKS 만큼의 여유를 두고 수행하기에 해당 결과가 나오게 된다.

c) Not commit

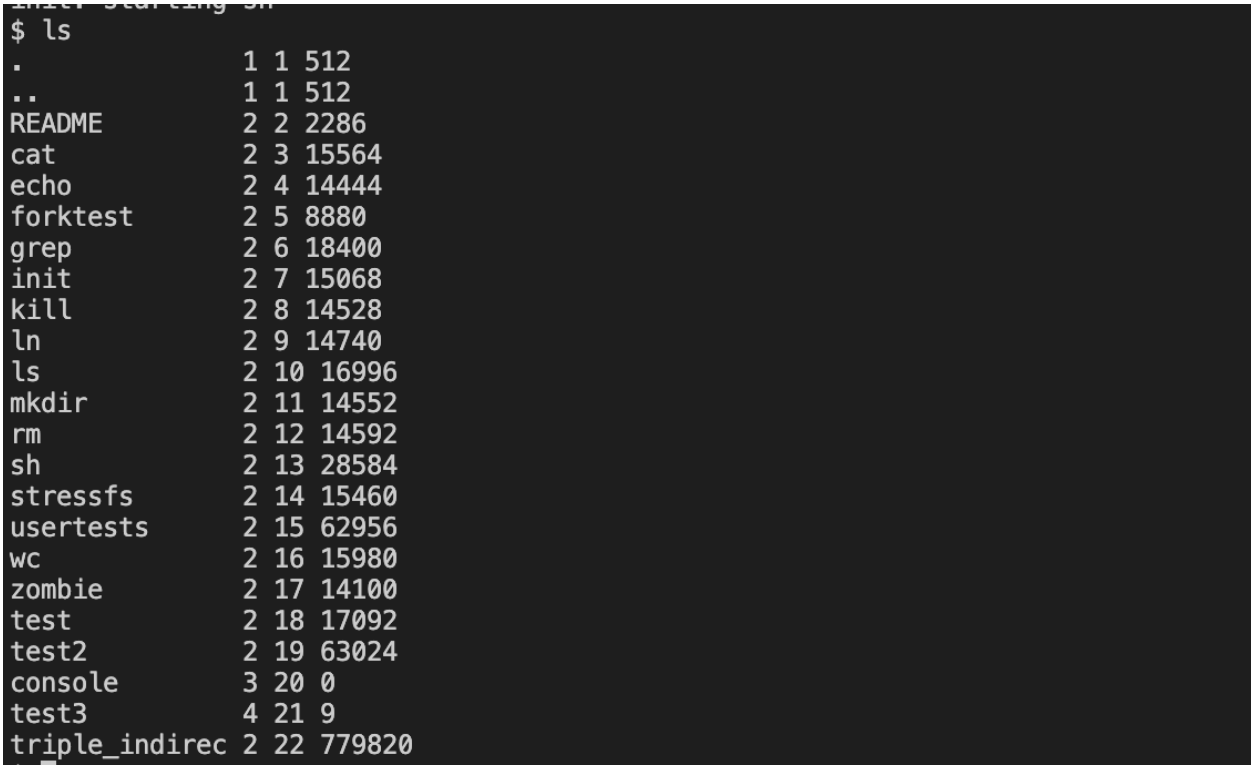
symbolic file 생성



재부팅 후 ls 명령어 수행

- sync가 호출되는 경우에는 파일 변경 사항이 disk에 저장되어 xv6 재부팅 후에도 해당 사항이 저장되어 있다.
- **group commit에는 operation이 끝나면 무조건 disk에 파일 변경 사항이 저장되지만, 수정한 buffer I/O는 그렇지 않다.**
- 그래서 link 파일을 만들고 buffer가 꽉 차기 전, 즉 **commit이 수행 되기 전에 xv6를 재부팅 하여 해당 link 파일이 남아있는지 확인하는 테스트를 진행하였다.**
- 결과는 disk에 파일 변경 사항이 저장되지 않아서 **ls 명령어 수행 시 해당 file metadata가 출력되지 않는 모습이다.**

symbolic file 생성 후, sync 수행



재부팅 후 ls 명령어 수행

- 반대로, **sync 호출이 정상적으로 되고, disk에 파일 변경사항이 정상적으로 반영되는지 확인 하기 위한 테스트를 진행하였다.**
- 해당 테스트는 symbolic link file을 만든 후, file I/O를 충분히 진행하여 sync를 호출되게 만들어주고, 재부팅 후에 ls 명령어 수행 시 file의 metadata가 출력 되는지 확인하는 것이다.
- 결과는 정상적으로 **disk에 파일 변경사항이 반영되어 ls 명령어 수행 시에 같이 출력되는 모습이다.**

4) usertests

- triple indirect와 sync를 가장 test하기 좋은 것은 usertests라는 생각이 들었다.
- 그 중 특히 **big files test는 MAXFILE의 크기만큼 file read, write를 수행하는데**, 해당 testcase를 통과하게 된다면 triple indirect, sync 모두 정상적으로 수행되는 것이다.

```
small file test
creat small succeeded; ok
writes ok
open small succeeded ok
read succeeded ok
small file test ok
big files test
big files ok
many creates, followed by unlink test
many creates, followed by unlink; ok
openiput test
openiput test ok
exitiput test
exitiput test ok
iput test
iput test ok
mem test
allocuvm out of memory
mem ok
pipe1 ok
preempt: kill... wait... preempt ok
exitwait ok
rmdot test
rmdot ok
fourteen test
fourteen ok
bigfile test
bigfile test ok
subdir test
subdir ok
linktest
linktest ok
unlinkread test
unlinkread ok
dir vs file
dir vs file OK
empty file name
empty file name OK
fork test
fork test OK
bigdir test
bigdir ok
uio test
pid 591 usertests: trap 13 err 0 on cpu 0 eip 0x35c3 addr 0x801dc130--kill proc
uio test done
exec test
ALL TESTS PASSED
```

usertests 중 file I/O는 성공

4. Trouble Shooting

1) redirecting in open, exec

- 처음에는 open, exec에서 원본 파일을 redirecting하는 방식으로 symlink를 구현하였다.
- 이때의 문제점은, symlink file 자체의 metadata에 접근 할 수 없다는 점이었다.
- 사실, **symbolic file 자체도 하나의 file이기에 해당 file의 metadata에 접근 할 수 있어야 하고** 그 뜻은 open, exec에서 redirecting하는 방식이 어느정도 잘못 되었다는 뜻이다.
- 이를 구현하기 위해서, 실제 file의 data block에 접근하는, **readi, writei에서만 redirecting하는 구조로 변경해 주었다.**

2) link cycle && link and link

- symbolic link를 처음 구현하였을 때, link의 link 그리고 link의 cycle을 고려하지 않았다.
- readi, writei를 수행 할 때, **단순히 count를 세어주어 cycle 여부를 판단하고, while 문을 사용하여 link의 link를 수행하게 해주었다.**

3) symlink file exec

- **symlink file을 만든 후, xv6를 종료하고 해당 file을 바로 수행 했을 때**, 정상적으로 수행 되지 않는 문제가 발생하였다.
- 이에 대한 문제 유추로 **symlink file의 metadata가 정상적으로 disk에 저장되지 않았다고** 생각하였다.
    - ilock에서 dinode와 inode의 정보를 동기화 시키는 부분이 있다.
    - symlink file에 대한 file I/O를 수행 했을 때, ilock을 호출하지 않아 정보 동기화가 되지 않을 것이라고 유추 하였다.
    - 그래서 file redirecting 후 ilock과 iunlock을 호출하게 해주어 해결하였다.

4) log size(1)

- 처음 구현할 때에는, begin_op 에서 log size가 찼는지 확인해주려 하였다.
- 이 구현에서 겪은 문제점은 begin_op를 통과해서 buffer를 할당 받게 될 때, **이미 log size가 찼지만, begin_op를 통과한 추가적인 작업이 많기에 buffer를 할당 받지 못하는 문제가 생겼다.**
- 그래서 **bget 즉, buffer를 할당받기 바로 직전에 sync를 호출 하여 buffer**가 꽉 찼는지 확인 해주는 구현으로 진행하였다.

5) log size(2)

- log size가 완전히 꽉 찼을때, 즉 정말로 buffer에 가용한 공간이 하나도 없을 때 commit을 수행하게 구현하였었다.
- 이때 구현의 문제점은, **commit 수행 시에 log내용을 buffer에 한번 저장한 후에 disk에 반영하는데, commit을 수행할 buffer의 공간조차 없다는 것이었다.**
- 그래서, **MAXOPBLOCKS만큼의 여유공간을 두어, 정상적으로 commit**을 수행 할 수 있도록 해주었다.