ELE3021_project03_12298_2019042497

Design

1. Multi Indirect

   우선 기존에 dinode에서 addrs의 NDIRECT로 할당되던 2칸을 줄여 10칸으로 만들고, 뒤에 double과 triple indirect를 위한 칸을 추가해줄 것입니다. 그리고 기존에 NDIRECT와 NINDIRECT가 사용된 bmap()과 itrunc()를 수정해 double indirect와 triple indirect가 동작하도록 수정해줄 것입니다.

2. Symbolic Link

   옵션이 추가되었기 때문에 ln.c에서 argc를 4개 받는 것으로 바꿔줄 것입니다. 그리고 symbolic_link를 하는 syscall 함수를 만들어서 '-h' 옵션이 들어오면 기존의 link를 진행해주고, '-s' 옵션이 들어오면 새로 만든 symbolic_link를 진행하도록 할 것입니다. 그리고 symbolic link로 생성된 file이라면 일단 flag를 줘서 나중에 구분할 수 있도록 하고, writei와 readi에서 redirection을 통해 원본 파일에 접근하도록 해줄 것입니다. 기존에 생각했던 sys_open()에서 해주면 ls와 exec 등이 동작하지 않고, namei()에서 redirection을 해주면 ls가 제대로 동작하기 않기 때문에, open까지는 본인의 메타데이터 정보를 가지고 있다가, 실제로 write과 read를 할 때 해준다면 exec, ls, cat 등이 다 잘 동작하게 할 수 있을 것 같다고 판단했습니다.

3. Sync

   기존의 group commit을 수정해서 구현할 것입니다. end_op()에서 log.outstanding을 1 감소시켜 실행 중인 FS syscalls의 개수를 하나 줄이고, 현재 실행 중인 FS syscalls가 없다면 commit하는 게 기존 구현입니다. 이때, 이 commit을 하는 부분을 지우고 sync() 함수로 따로 분리해낼 것입니다. 이때 중요한 점은 기존에 log space의 남은 공간이 부족하면 commit이 수행되어 공간이 확보될 때까지 sleep하는데, sync()로 따로 분리할 것이기 때문에 공간 확보가 되지 않아 여기서 오류가 날 수 있습니다. 따라서 해당 오류를 처리하기 위해 log space의 공간이 부족해질 때 sync()를 호출해 disk에 flush 해줄 것입니다. Xv6에서 log는 buffer를 거쳐서 disk에 저장되기 때문에 해당 처리를 해주면 buffer space가 가득 찼을 때 flush하는 것과 같은 의미가 되어 해결 가능합니다.

Implement

먼저, Multi Indirect의 구현입니다.

```
#define NDIRECT 10 // 기존에 12에서 double과 triple을 위해 10으로 줄임
#define NINDIRECT (BSIZE / sizeof(uint))
#define DOUBLE_NINDIRECT (BSIZE / sizeof(uint)) * (BSIZE / sizeof(uint))
#define TRIPLE_NINDIRECT (BSIZE / sizeof(uint)) * (BSIZE / sizeof(uint)) * (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT + DOUBLE_NINDIRECT + TRIPLE_NINDIRECT) // 총 file의 개수

// 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, single+double+triple
};
```

먼저 fs.h 안의 값을 다음과 같이 수정해주었습니다. addrs에서 2칸을 각각 double indirect와 triple indirect를 위해 할당하기 위해 기존의 NDIRECT의 값 12에서 2만큼 줄여 10으로 만들었습니다. 그리고 각각의 사이즈를 define 해줘 나중에 사용하기 편하도록 했습니다. MAXFILE 수 또한 이에 맞춰 증가시켜주었습니다.

```
bn -= NINDIRECT; // block number에서 이미 할당한 NINDIRECT만큼 값을 빼줌 (double에서 0번 index부터 채워넣기 위함)
// double indirect
if(bn < DOUBLE_NINDIRECT){ // bn이 double indirect 범위 보다 작은 경우
  // Load indirect block, allocating if necessary.
  if((addr = ip->addrs[NDIRECT + 1]) == 0) // NDIRECT + 1 (double)의 값이 0이라면 (할당되지 않았다면)
    ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev); // 새로운 data block을 할당
  bp = bread(ip->dev, addr); // 만든 data block을 읽어와 bp에 저장
  a = (uint*)bp->data; // a에 block 내용을 가져옴
  if((addr = a[bn / (NINDIRECT)]) == 0){ // double indirect의 첫 번째 주소 공간을 담는 공간이 할당되지 않았다면
    a[bn / (NINDIRECT)] = addr = balloc(ip->dev); // 새로운 data block을 할당
    log_write(bp); // log에 변경내용 기록
  }
  brelse(bp); // 사용이 끝난 data block을 반환
  bp = bread(ip->dev, addr); // 만든 data block을 읽어와 bp에 저장
  a = (uint*)bp->data; // a에 block 내용을 가져옴
  if((addr = a[bn % (NINDIRECT)]) == 0){ // double indirect의 두 번째 주소 공간을 담는 공간이 할당되지 않았다면
    a[bn % (NINDIRECT)] = addr = balloc(ip->dev); // 새로운 data block을 할당
    log_write(bp); // log에 변경내용 기록
  }
  brelse(bp); // 사용이 끝난 data block을 반환
  return addr; // 할당 받은 block 반환
}
```

다음은 fs.c의 bmap() 함수 안에서의 추가 부분입니다. Double indirect를 위해 기존에 할당된 만큼을 먼저 빼 주었습니다. 그리고는 addrs에서 [NDIRECT + 1]의 index를 사용해주었습니다. 여기에 값이 할당되지 않았다면 block을 할당해주고, 정보를 읽어오도록 했습니다. 그리고는 [bn / (NINDIRECT)]의 index를 사용해 double indirect의 첫번째 block table에서의 index를 판단해주도록 했습니다. 이후 똑같이 block을 할당하고, 정보를 읽어오도록 했습니다. 그리고 여기서 [bn % (NINDIRECT)]의 index를 사용해 최종적으로 두번째 block table에서 data가 할당되도록 해주었습니다. 기존의 Single indirect 부분에서 가져와 수정했습니다.

```
bn -= DOUBLE_NINDIRECT; // block number에서 이미 할당한 DOUBLE_NINDIRECT만큼 값을 빼줌 (triple에서 0번 index부터 채워넣기 위함)
// triple indirect
if(bn < TRIPLE_NINDIRECT){ // bn이 triple indirect 범위 보다 작은 경우
  // Load indirect block, allocating if necessary.
  if((addr = ip->addrs[NDIRECT+2]) == 0) // NDIRECT + 2 (triple)의 값이 0이라면 (할당되지 않았다면)
    ip->addrs[NDIRECT+2] = addr = balloc(ip->dev); // 새로운 data block을 할당
  bp = bread(ip->dev, addr); // 만든 data block을 읽어와 bp에 저장
  a = (uint*)bp->data; // a에 block 내용을 가져옴
  if((addr = a[bn / (DOUBLE_NINDIRECT)]) == 0){ // triple indirect의 첫 번째 주소 공간을 담는 공간이 할당되지 않았다면
    a[bn / (DOUBLE_NINDIRECT)] = addr = balloc(ip->dev); // 새로운 data block을 할당
    log_write(bp); // log에 변경내용 기록
  }
  brelse(bp); // 사용이 끝난 data block을 반환
  bn %= (DOUBLE_NINDIRECT); // 계산하기 쉽게 첫 번째 주소 공간 할당한 만큼의 나머지를 구해서 사용함
  bp = bread(ip->dev, addr); // 만든 data block을 읽어와 bp에 저장
  a = (uint*)bp->data; // a에 block 내용을 가져옴
  if((addr = a[bn / (NINDIRECT)]) == 0){ // triple indirect의 두 번째 주소 공간을 담는 공간이 할당되지 않았다면
    a[bn / (NINDIRECT)] = addr = balloc(ip->dev); // 새로운 data block을 할당
    log_write(bp); // log에 변경내용 기록
  }
  brelse(bp); // 사용이 끝난 data block을 반환
  bp = bread(ip->dev, addr); // 만든 data block을 읽어와 bp에 저장
  a = (uint*)bp->data; // a에 block 내용을 가져옴
  if((addr = a[bn % (NINDIRECT)]) == 0){ // triple indirect의 세 번째 주소 공간을 담는 공간이 할당되지 않았다면
    a[bn % (NINDIRECT)] = addr = balloc(ip->dev); // 새로운 data block을 할당
    log_write(bp); // log에 변경내용 기록
  }
  brelse(bp); // 사용이 끝난 data block을 반환
  return addr; // 할당 받은 block 반환
}
```

Triple indirect도 마찬가지로 동작하는데, 이때 한 개의 block table이 더 있기 때문에 처음에 [bn / (DOUBLE_NINDIRECT)]의 index로 나눠서 첫번째 block table에서 어디로 들어갈지를 정해주도록 했고, 계산의 편의를 위해 bn % (DOUBLE_NINDIRECT) 값을 가지고 그 다음을 계산하도록 해주었습니다. 이렇게 Multi Indirect의 block mapping을 해주었습니다.

```
// double indirect
if(ip->addrs[NDIRECT + 1]){ // inode의 addrs에서 double indirect를 담은 index에 할당이 되어 있다면
  bp = bread(ip->dev, ip->addrs[NDIRECT + 1]); // 해당 data block을 bp로 읽어옴
  a = (uint*)bp->data; // a에 block 내용을 가져옴
  for(i = 0; i < NINDIRECT; i++){ // 해당 block의 칸을 전부 순회
    if(a[i]){ // 만약에 할당되어 있다면
      bp1 = bread(ip->dev, a[i]); // 해당 data block을 bp1으로 읽어옴
      a1 = (uint*)bp1->data; // a1에 bp1의 내용을 가져옴
      for(j = 0; j < NINDIRECT; j++){ // bp1의 칸을 전부 순회
        if(a1[j]) // 만약에 할당되어 있다면
          bfree(ip->dev, a1[j]); // a1[j] block을 free
      }
      brelse(bp1); // 사용이 끝난 bp1을 반환
      bfree(ip->dev, a[i]); // a[i] block을 free
    }
  }
  brelse(bp); // 사용이 끝난 bp를 반환
  bfree(ip->dev, ip->addrs[NDIRECT + 1]); // ip->addrs[NDIRECT + 1] block을 free
  ip->addrs[NDIRECT + 1] = 0; // 0으로 초기화
}
```

다음은 fs.c의 itrunc() 함수의 추가 부분입니다. bmap() 수정과 마찬가지로 기존의 함수에서 진행되는 순서를 지키며 함수가 실행되도록 했습니다. Double indirect가 할당되어 있는 [NDIRECT + 1]의 index의 값을 읽어오도록 했고, 해당 block이 가지고 있는 block들을 전부 순회해서 할당되어 있다면 free를 해주도록 했습니다. 처음 block table의 block이 가진 block들의 순회를 마치면 해

당 block 또한 free가 되도록 했습니다. 이를 반복한 후 마지막에 0 값으로 아무것도 없다고 초기화를 해서 오류가 나지 않도록 처리했습니다. 마찬가지로 기존의 single indirect 부분을 가져와서 수정해주었습니다.

```
// triple indirect
if(ip->addrs[NDIRECT + 2]){ // inode의 addrs에서 triple indirect를 담은 index에 할당이 되어 있다면
  bp = bread(ip->dev, ip->addrs[NDIRECT + 2]); // 해당 data block을 bp로 읽어옴
  a = (uint*)bp->data; // a에 block 내용을 가져옴
  for(i = 0; i < NINDIRECT; i++){ // 해당 block의 칸을 전부 순회
    if(a[i]){ // 만약에 할당되어 있다면
      bp1 = bread(ip->dev, a[i]); // 해당 data block을 bp1으로 읽어옴
      a1 = (uint*)bp1->data; // a1에 bp1의 내용을 가져옴
      for(j = 0; j < NINDIRECT; j++){ // bp1의 칸을 전부 순회
        if(a1[j]){ // 만약에 할당되어 있다면
          bp2 = bread(ip->dev, a1[j]); // 해당 data block을 bp2으로 읽어옴
          a2 = (uint*)bp2->data; // a2에 bp2의 내용을 가져옴
          for(k = 0; k < NINDIRECT; k++){ // bp2의 칸을 전부 순회
            if(a2[k]) // 만약에 할당되어 있다면
              bfree(ip->dev, a2[k]); // a2[k] block을 free
          }
          brelse(bp2); // 사용이 끝난 bp2를 반환
          bfree(ip->dev, a1[j]); // a1[j] block을 free
        }
      }
      brelse(bp1); // 사용이 끝난 bp1을 반환
      bfree(ip->dev, a[i]); // a[i] block을 free
    }
  }
  brelse(bp); // 사용이 끝난 bp를 반환
  bfree(ip->dev, ip->addrs[NDIRECT + 2]); // ip->addrs[NDIRECT + 2] block을 free
  ip->addrs[NDIRECT + 2] = 0; // 0으로 초기화
}
```

Triple indirect도 [NDIRECT + 2]의 index를 읽어오고, 모든 block들을 하나하나 free 하기 위해 첫 번째부터 세 번째 block table까지 하나하나 순회를 하도록 해서 free를 해주었습니다. Double indirect에서 한 block table만큼 더 생겼기 때문에 해당 구현만 추가로 신경써주었습니다.

```
#define FSSIZE       100000  // size of file system in blocks (> 32768)
```

마지막으로 param.h에서 FSSIZE parameter를 여유롭게 10만으로 늘려서 최대 데이터 블록 개수의 제한을 확 늘려주었습니다. 테스트로 요구된 16MB 정도를 넘는 크기를 설정하기 위해 넉넉하게 10만으로 구현했습니다.

다음은 Symbolic Link의 구현입니다.

```
int
main(int argc, char *argv[])
{
  if(argc != 4){ // 옵션이 추가되었기 때문에 4개 인자인지 체크
    printf(2, "Usage: ln [-h or -s] [old] [new]\n");
    exit();
  }
  if(!strcmp(argv[1], "-h")){ // 기존의 hard link라면
    if(link(argv[2], argv[3]) < 0)
      printf(2, "hard link %s %s: failed\n", argv[2], argv[3]);
    exit();
  }
  else if(!strcmp(argv[1], "-s")){ // symbolic link라면
    if(symbolic_link(argv[2], argv[3]) < 0) // symbolic link 실행
      printf(2, "symbolic link %s %s: failed\n", argv[2], argv[3]);
    exit();
  }
}
```

먼저, ln.c입니다. 기존의 hard link만 되었을 때는 argc가 3일 때만 실행되도록 했지만, 옵션을 추가하기 때문에 argc가 4개일 때 실행되도록 바꿔주었습니다. 그리고 '-h' 옵션이 들어오면 기존에 실행하던 코드를 넣어주었고, '-s' 옵션이 들어오면 기존 코드에서 link() 대신 symbolic_link가 실행되도록 해주었습니다.

```
#define T_SYM  4   // Symbolic link
```

다음은 symbolic link 파일을 만들었을 때, symbolic link 파일임을 표시하기 위해 stat.h에서 T_SYM을 4로 주었습니다.

```
  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+3];
  char old[50]; // symbolic link로 연결한 old file의 name
```

또한, file.h의 inode의 구조체에 symbolic link로 연결한 old file의 이름을 담는 변수를 추가해주었습니다.

```
// symbolic link를 만드는 함수
int
sys_symbolic_link(void)
{
  char *new, *old;
  struct inode *ip;
  struct file *f;

  if(argstr(0, &old) < 0 || argstr(1, &new) < 0) // 첫 번째, 두번째 인자가 올바른지 확인
    return -1;

  begin_op();

  ip = create(new, T_SYM, 0, 0); // symbolic link file을 생성하고 해당 file의 inode를 반환
  if(ip == 0){
    end_op();
    return -1;
  }

  if((f = filealloc()) == 0){ // file 구조체를 할당하고, 오류가 난다면
    if(f)
      fileclose(f);
    iunlockput(ip);
    end_op();
    return -1;
  }
  strncpy(ip->old, old, strlen(old)); // ip의 old에 old를 그대로 복사
  iunlock(ip);
  end_op();

  f->type = FD_INODE;
  f->ip = ip;
  f->off = 0;
  f->readable = 1;
  f->writable = 0;
  return 0;
}
```

syscall 함수인 sys_symbolic_link()입니다. sys_link()와 sys_open()에서 코드를 가져와 변형했습니다. 먼저 old와 new의 인자가 올바른 지 체크하고, T_SYM을 갖는 file을 생성합니다. 이 file의 inode에서 만들어왔던 ip->old에 주어진 인자 old를 복사해서 넣었습니다. 여기까지 된 이후, 실제로 symbolic에 대한 File Operation은 나중에 readi()와 writei()에서 old의 이름을 가지고 redirection 되어 실행됩니다.

```
if(ip->type == T_SYM){ // 만약 symbolic link라면
  struct inode *ip2; // 새로 inode의 정보를 담을 변수
  if((ip2 = namei((char*)ip->old)) != 0) // symbolic link가 가리키는 old가 있다면
    ip = ip2; // old로 ip를 대체
  else
    return -1;
  ilock(ip);
  iunlock(ip);
}
```

다음은 readi()와 writei()에서 redirection하는 부분입니다. 두 함수 다 똑같이 해당 코드가 초반에 들어가 있습니다. 새로운 inode의 정보를 담을 변수를 만들고, 여기에 namei(ip->old)를 통해 symbolic link가 가지고 있던 old의 name을 넣어 old의 inode를 ip2에 받습니다. 그리고 이게 성공을 한다면 ip를 ip2로 대체해 redirection 되도록 구현했습니다.

마지막으로 Sync 구현입니다.

```
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);
      log.now_lock = 1; // log.lock을 획득했음을 체크
      sync();           // log와 buffer flush
      log.now_lock = 0; // log.lock 획득 체크 해제
    }
    else {
      log.outstanding += 1;
      release(&log.lock);
      break;
    }
  }
}
```

먼저, log.c 에서 begin_op() 함수입니다. 여기서 문제가 생기는 부분이 기존에 log 공간이 다 찼을 때 sleep 을 시킴으로써 flush 되기를 기다리게 해주었던 부분인데 그냥 지우면 log 공간이 꽉 찰 수 있기 때문에 여기서 flush 를 강제로 하도록 해주었습니다. 또한, log.lock 을 획득한 상태이기 때문에 구조체 안에 now_lock 변수를 선언해서 획득했음을 표시해주었고, sync()가 끝나고 나왔을 때 해당 상태를 변경해주었습니다.

```
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);
  }*/
}
```

end_op() 함수에서는 기존에 있던 do_commit 관련 부분을 다 지워서 매번 commit 되지 않도록 했습니다.

```
// 현재 Buffer에 있는 모든 Dirty Buffer를 Flush 하는 함수
int
sync(void)
{
  int flush_num = log.lh.n; // flush된 block의 수 출력
  if (!log.now_lock) // log.lock을 얻지 않았다면
    acquire(&log.lock); // log.lock 획득
  while(log.outstanding || log.committing) // 실행중인 FS syscall이 있다면
    sleep(&log, &log.lock);

  log.committing = 1; // commit 중임을 표시
  release(&log.lock); // log.lock 해제

  commit(); // commit
  acquire(&log.lock); // log.lock 획득
  log.committing = 0; // commit 중 표시를 해제
  wakeup(&log); // log sleep을 깨움

  if (!log.now_lock) // 기존에 log.lock을 얻지 않았다면
    release(&log.lock); // log.lock 해제

  if(log.lh.n != 0) // 개수가 0이 아니면 오류
    return -1;
  return flush_num; // flush한 개수 출력
}

// sync 함수의 system call 함수
int
sys_sync(void)
{
  return sync(); // sync()에서 return 된 값을 return
}
```

sync()와 sys_sync() 함수입니다. begin_op()에서 강제로 commit 할 때 log.lock 을 획득해서 오기 때문에 미리 now_lock 이라는 변수를 구조체 안에 만들어 해당 경우에는 log.lock 을 획득하지 않도록 해주었습니다. 그 후, while 문을 계속 돌면서 현재 실행 중인 FS syscall 이 있다면 sleep 을 해주었습니다. 이후, 차례가 오면 log.committing 을 1 로 표시해 commit 중임을 표시해주었고, 기존에 end_op()에서 do_commit 하던 부분을 가져와 flush 를 해주었습니다. 이후, 기존에 log.lock 을 얻지 않았다면 log.lock 을 해제해 주었습니다. 마지막으로 오류로 정상적인 flush 가 되지 않았다면 -1 을 반환하고, 성공을 했다면 flush 된 block 수를 반환하도록 해주었습니다.

마지막으로 system call 함수를 등록한 과정을 설명하겠습니다.

```
#define SYS_symbolic_link 22
#define SYS_sync   23
```

```
extern int sys_symbolic_link(void);
extern int sys_sync(void);
```

```
[SYS_symbolic_link] sys_symbolic_link,
[SYS_sync]    sys_sync,
```

syscall.h 과 syscall.c 에 만들어 놓은 wrapper function 을 등록해주었습니다.

```
int symbolic_link(const char*, const char*);
int sync(void);
```

```
SYSCALL(symbolic_link)
SYSCALL(sync)
```

user program 이 사용할 수 있도록 user.h 에 해당 함수들을 넣고 usys.S 에도 넣어주었습니다

Result

먼저 Multi Indirect 테스트입니다.

```
$ test_test
huge file test starting
1. create test
0 bytes written          16435200 bytes written
51200 bytes written      16486400 bytes written
102400 bytes written     16537600 bytes written
153600 bytes written     16588800 bytes written
204800 bytes written     16640000 bytes written
256000 bytes written     16691200 bytes written
307200 bytes written     16742400 bytes written
358400 bytes written     16777216 bytes written
409600 bytes written     1. create test finished
```

16MB 크기 정도의 file을 buffer 단위로 create하는 test입니다.

Double indirect까지만 구현이 된다면 약 8MB쯤에서 멈추지만 16MB 끝까지 돌아가는 것을 확인할 수 있었습니다.

```
2. read test
0 bytes read          16486400 bytes read
51200 bytes read      16537600 bytes read
102400 bytes read     16588800 bytes read
153600 bytes read     16640000 bytes read
204800 bytes read     16691200 bytes read
256000 bytes read     16742400 bytes read
307200 bytes read     16777216 bytes read
358400 bytes read     2. read test finished
```

16MB 크기 정도의 file을 buffer 단위로 read하는 test입니다. 마찬가지로 16MB 끝까지 돌아가는 것을 확인했습니다.

```
3. stress test
0 bytes written
51200 bytes written   16486400 bytes written
102400 bytes written  16537600 bytes written
153600 bytes written  16588800 bytes written
204800 bytes written  16640000 bytes written
256000 bytes written  16691200 bytes written
307200 bytes written  16742400 bytes written
358400 bytes written  16777216 bytes written
409600 bytes written  3. stress test finished
```

마지막으로 unlink했다가 다시 하는 stress test입니다. 마찬가지로 잘 동작하는 것을 확인했습니다.

다음은 Symbolic Link 테스트입니다.

```
$ echo hi > hello
$ ln -h hello hard
$ ln -s hello sym
```

먼저 echo로 hi를 가지고 있는 hello file을 생성했습니다. 그리고 hard로 hello를 hard linking 했고, sym으로 hello를 symbolic linking 했습니다.

```
hello           2 21 3
hard            2 21 3
sym             4 22 0
$ cat hello
hi
$ cat hard
hi
$ cat sym
hi
```

ls로 출력을 했을 때 hello와 hard file은 정확히 같은 metadata를 가지는 것을 확인할 수 있고, sym은 자신만의 metadata를 가지고 있는 것을 확인할 수 있었습니다.

또한, cat를 통해 각 file 안의 내용을 출력해보았을 때 hard는 당연히 hello와 같은 값을 출력하는 것을 확인할 수 있었고, sym은 정상적으로 hello로 redirection 되어 그 값을 출력하는 것을 확인할 수 있었습니다.

```
$ rm hello
$ cat hard
hi
$ cat sym
cat: read error
```

다음에 hello를 삭제를 했을 때 hard는 그대로 동작하는 것을 확인할 수 있었고, sym은 원본 file이 사라졌기 때문에 에러가 나는 것을 확인할 수 있었습니다.

이외에도 exec()도 잘 되는 지 확인하기 위해 ls 명령어를 symbolic linking 한 결과 정상적으로 동작하는 것을 확인했습니다.

다음은 Sync 테스트입니다.

```
$ echo hi > hello
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16328
echo           2 4 15180
forktest       2 5 9488
grep           2 6 18544
init           2 7 15764
kill           2 8 15208
ln             2 9 15356
ls             2 10 17696
mkdir          2 11 15308
rm             2 12 15288
sh             2 13 27928
stressfs       2 14 16196
usertests      2 15 67304
wc             2 16 17064
zombie         2 17 14876
test_test      2 18 17484
sync_test      2 19 14828
console        3 20 0
hello          2 21 3
$ QEMU: Terminated
```

```
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16328
echo           2 4 15180
forktest       2 5 9488
grep           2 6 18544
init           2 7 15764
kill           2 8 15208
ln             2 9 15356
ls             2 10 17696
mkdir          2 11 15308
rm             2 12 15288
sh             2 13 27928
stressfs       2 14 16196
usertests      2 15 67304
wc             2 16 17064
zombie         2 17 14876
test_test      2 18 17484
sync_test      2 19 14828
console        3 20 0
$
```

왼쪽을 보면 먼저 hello file을 만들고 (ctrl+a) 이후 x를 눌러 강제 종료한 과정입니다.

이후 다시 xv6를 실행했습니다. 결과는 오른쪽과 같이 아직 sync()를 통한 flush가 일어나지 않았기에, 이전에 메모리 상에서만 존재했던 data로 disk 상에 저장되지 못해 다시 실행했을 때 만들어 놓은 hello file은 사라졌음을 확인할 수 있습니다.

```c
int
main(void)
{
  sync();
  exit();
};
```

이제 sync_test.c를 만들어 sync() syscall을 불러 강제로 disk에 data를 남기는 test를 실행해볼 것입니다.

왼쪽은 hhh file을 만들고, 바로 sync_test를 해서 sync()를 호출해 disk에 저장한 후, 이전과 같이 (ctrl+a) 이후 x를 눌러 강제 종료한 과정입니다.

이후 다시 xv6를 실행했습니다. 결과는 오른쪽과 같이 disk로 flush 되었기 때문에 hhh file이 잘 남아있는 것을 확인할 수 있었습니다.

Trouble shooting

1. symbolic link를 구현할 때 sys_open()에서 redirection을 구현했을 때, ls와 exec() 둘 다 정상적으로 작동하지 않는 문제가 발생한 경우

해결: 처음에는 open()에서 구현하지 않고, namei() 함수로 들어가서 redirection 되도록 해주었습니다. 이때, exec()에서는 정상적으로 작동하는 것을 볼 수 있었습니다. 하지만 ls에서 symbolic file이 본인의 metadata를 출력하지 않고, redirection된 원본 file의 metadata를 출력하는 것을 확인했습니다.

이후 고민 끝에 read와 write 할 때 redirection을 하게 된다면 모든 게 정상적으로 동작할 수 있겠다는 생각을 하였고, readi()와 writei()에서 redirection을 해주면서 정상적으로 동작하도록 만들어주었습니다.

2. sync()를 구현할 때, log가 꽉 차기 전에 비워줘야 하는 데 그렇지 못해 too big a transaction 오류가 출력된 경우

해결: 기존에 group commit과 관련해서 log.outstanding과 do_commit 부분을 다 지우다 보니 발생한 문제임을 발견했습니다. 따라서 무작정 다 없애는 것이 아니라 log space가 꽉 차는 것을 방지하기 위한 예외 처리를 해줘야 했습니다.

따라서 다시 log.outstanding을 사용하여 현재 실행 중인 FS syscalls가 어느 정도 있는 지 체크하고, 기존에 begin_op()에서 log가 다 찼을 때 조건문에서 sync()를 해주면서 해결해주었습니다.

3. sync()를 구현하면서 log.lock을 획득했는데 또 획득하려고 하여 오류가 난 경우

해결: log.lock을 획득한 경우가 위의 2번 문제를 해결하는 과정에서 발생해 sync()를 호출하고 있음을 찾았고, 이를 해결하기 위해 구조체에 now_lock이라는 변수를 만들어서 flag처럼 다뤄주면서 해결했습니다.