ELE3021_project03

Path of Project03 & How to execute xv6

- project 3의 위치

  Project_3/xv6-public에 구현했습니다. (노란색 형관펜으로 표시된 곳)

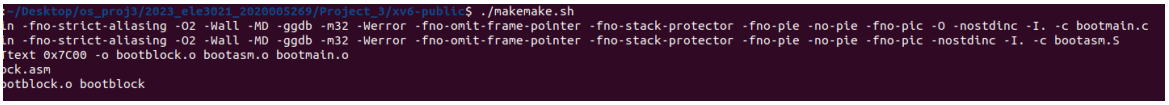
Name	Last commit	Last update
.vscode	UDT: mlfq 갈아엎고 임시 커밋	2 months ago
Project_1/xv6-public	FIN: L2 queue	1 month ago
Project_2	UDT: exit() 코드 수정	2 weeks ago
Project_3	UDT: symbolic real final	5 days ago
lab4_systemcall	FIX: lab4 - system call - comment fix	2 months ago
lab6-sync	CRET: mlfq_test 추가, 시스템콜 이름 변경, 내일 다시 고쳐야함	2 months ago
xv6-public	FIX: proc_ticks 초기화 오류 fix, mlfq 완성(?)	2 months ago

- 실행방법
  Project_3/xv6-public 디렉토리에서

```
make clean
./makemake.sh
./run-xv6.sh
```

  를 쉘에 입력하여 실행합니다.

  빌드 화면 예시



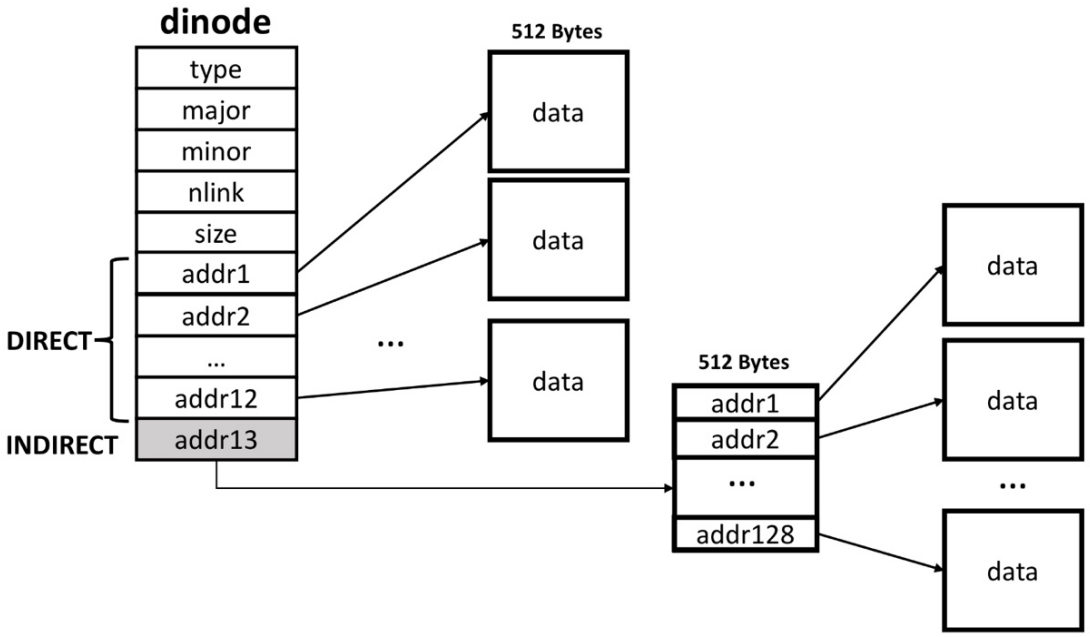
Design

Project 3은 double, tripple indirect table과 symbolic link, sync() system call을 이용한 buffer i/o를 구현하는 것이었으나, sync()는 구현하지 못해 double, tripple indirect table과 sybolic link만 구현이 되어있는 상태입니다.

다만 sync() buffer i/o를 어떻게 구현하려고 했는지 디자인과 implementation은 적어보려고 합니다.

- double, tripple indirect table

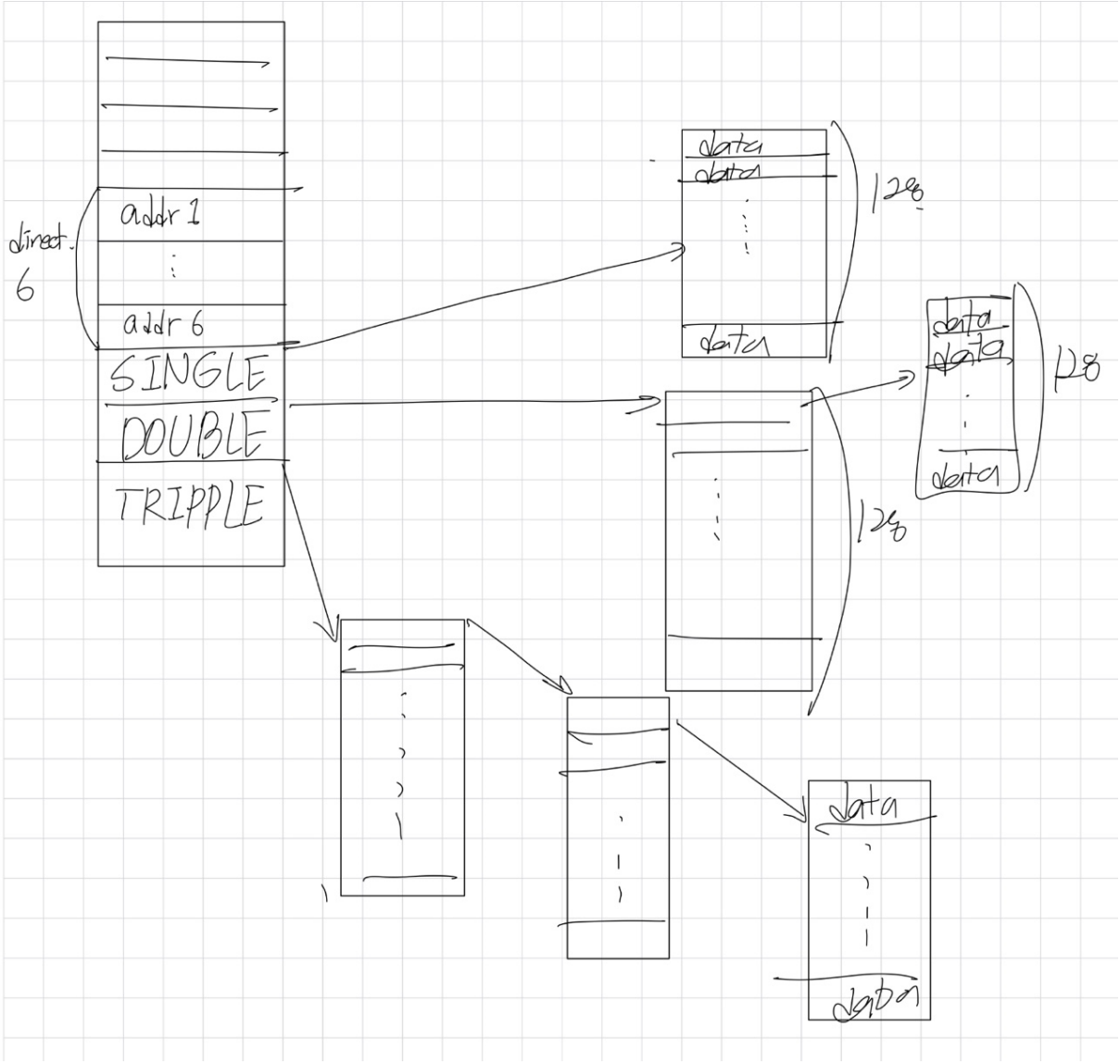
기존 xv6의 파일 시스템은 아래 그림과 같이 12개의 direct table, 1개의 single indirect table로 구성 되어있었습니다.



위와 같은 구조로는 한 파일이 최대 140개의 블록을 할당 받을 수 있어 약 70KB까지의 파일만 읽고 쓸 수 있다는 단점이 있었습니다. 저는 아래 그림과 같이 direct link의 개수를 12개→6개로 줄이고, 싱글 인다이렉트 테이블 아래에 더블 인다이렉트, 트리플 인다이렉트 테이블을 넣을 계획입니다.

direct의 개수를 10개로 줄이지 않고 6개로 줄인 이유는 dinode에 다른 변수를 더 선언했기 때문입니다.

아래는 디자인을 그림으로 나타낸 것입니다.



- symbolic link

기존 xv6는 두 파일이 inode를 공유하는 hard link만을 제공했습니다.

symbolic link는 복사한 파일이 원본 파일과 inode를 공유하지 않고 각각의 inode를 가지고 있고 원본 파일의 바로가기처럼 기능하는 link입니다.

symbolic link 파일을 B, 원본 파일을 A라고 할 때, 파일 B가 지워지더라도 A에는 영향이 없어야하며, 만약 원본인 A가 지워졌다면 'cat'등의 명령어를 실행했을 때 A에 접근되지 않아야합니다.

저는 'sys_open' 함수 내에서 심볼릭 링크 파일과 일반 파일을 구분하고, 심볼릭이라면 심볼릭 파일 대신에 원본 파일을 open할 수 있도록 구현할 예정입니다.

- sync()

기존 xv6는 begin_op 후에 어떤 block이 변경되면 그 block 번호를 log.lh에 저장해두었다가 end_op 호출시에 그 블록들의 변경사항을 디스크에 반영해주는 commit 방법을 제공했습니다.

이 commit 방식 대신 sync()하는 시스템 콜을 구현하여 bcache의 버퍼가 가득 차면 자동으로 sync()가 호출되어 변경 사항이 디스크로 내려가고, 유저가 원할 때 sync()를 호출하여 변경 사항을 디스크에 적을 수 있게 하는 시스템 콜을 구현하려고 합니다.

Implementation

- multi indirect table의 구현
  - fs.h

    해당 기능을 구현하기 위해 NDIRECT 매크로를 수정하고, NINDIRECT_D(double indirect table이 수용할 수 있는 block 개수), NINDIRECT_T(tripple indirect table이 수용할 수 있는 블록 개수)를 추가했습니다.

    ```
    #define NDIRECT 6
    #define NINDIRECT (BSIZE / sizeof(uint))
    #define NINDIRECT_D (NINDIRECT * NINDIRECT)
    #define NINDIRECT_T (NINDIRECT * NINDIRECT * NINDIRECT)

    #define MAXFILE (NDIRECT + NINDIRECT + NINDIRECT_D + NINDIRECT_T)
    ```

  - param.h

    16MB이상의 파일을 읽고 쓰기 위해서는 블록이 최소 32768개가 필요합니다. 따라서 'FSSSIZE'의 매크로를 변경하여 늘려주었습니다.

    ```
    #define FSSIZE       70000  // size of file system in blocks
    ```

  - fs.c
    - bmap 함수 수정

      추가한 multi indirect table에 맞게 블록을 할당할 수 있도록 함수를 다음과 같이 수정하였습니다. blockno가 다이렉트 테이블에 할당되는 범위라면 다이렉트에 할당아니라면 싱글 인다이렉트에 할당될 수 있는지 확인하고, 그 범위를 넘어선다면 더블 인다이렉트에 할당되는 범위인지 확인하고, 그것도 넘어선다면 트리플 인다이렉트 테이블에 할당하는 방식으로 구현했습니다.

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

        // cprintf("line 378 bn: %d\n", bn);

        // 다이렉트에 매핑 -> 원래 있었음
        if(bn < NDIRECT){
          if((addr = ip->addrs[bn]) == 0)
            ip->addrs[bn] = addr = balloc(ip->dev);
      ```

```
    return addr;
  }
  bn -= NDIRECT;

  // 싱글 인다이렉트에 매핑 -> 원래 있었음
  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;
  }

  bn -= NINDIRECT;

  // 더블 인다이렉트에 매핑 -> project 3에서 추가
  if(bn < NINDIRECT_D) {
    // outer table block부터 할당
    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;
  }
```

```
bn -= NINDIRECT_D;

// 트리플 인다이렉트에 매핑 -> project 3에서 추가
if(bn < NINDIRECT_T) {
  // table 1 (outer)
  if((addr = ip->addrs[NDIRECT+2]) == 0) {
    ip->addrs[NDIRECT+2] = addr = balloc(ip->dev);
  }

  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  // table 2 (outer)
  if((addr = a[bn/NINDIRECT_D]) == 0) {
    a[bn/NINDIRECT_D] = addr = balloc(ip->dev);
    log_write(bp);
  }

  brelse(bp);
  bn %= NINDIRECT_D;

  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  // table 3 (outer)
  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;

  bn %= NINDIRECT;

  // table 4 (inner)
  if((addr = a[bn]) == 0) {
    a[bn] = addr = balloc(ip->dev);
    log_write(bp);
  }

  brelse(bp);
  return addr;
}
```

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

- itrunc 함수 수정

  파일의 블록을 모두 해제하는 함수인 itrunc을 다음과 같이 수정했습니다.

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

  // 다이렉트 먼저 해제
  for(i = 0; i < NDIRECT; i++){
    if(ip->addrs[i]){
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }

  // 싱글 인다이렉트 해제
  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;
  }

  // 더블 인다이렉트 해제
  if(ip->addrs[NDIRECT+1]) {
    remove_children(ip, ip->addrs[NDIRECT+1], 2);
    ip->addrs[NDIRECT+1] = 0;
  }


  // 트리플 인다이렉트 해제
  if(ip->addrs[NDIRECT+2]) {
    remove_children(ip, ip->addrs[NDIRECT+1], 3);
    ip->addrs[NDIRECT+2] = 0;
  }
```

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

- remove_children 함수 구현

  멀티 인다이렉트에 매핑된 블록들을 쉽게 free하기 위해서 remove_children을 구현했습니다.

```
// project 3. iturc에서 호출하는 함수
// 인다이렉트 테이블에 계층적으로 매핑돼있는 블록들을 없애준다.
void
remove_children(struct inode *ip, uint addr, int lev) {
  int i;
  struct buf *bp;
  uint *a;

  bp = bread(ip->dev, addr); // 현재 블록(테이블)을 읽기
  a = (uint*)bp->data;
  for(i=0; i < NINDIRECT; i++) {
    if(a[i]) {
      if(lev > 1) {
        remove_children(ip, a[i], lev-1);
      }

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

- symbolic link
  - fs.h과 file.h

    심볼릭 링크를 구현하기 위해, fs.h의 dinode, file.h의 inode struct를 다음과 같이 변경했습니다.
    (inode의 내부 필드는 dinode의 구성과 동일합니다.)

    'char path[12]'는 해당 파일이 심볼링 링크 파일 일때만 의미가 있으며, 원본파일의 path를 의미합니다.

```
// 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
```

```
  // Project 3에서 추가
  char path[12];         // 심볼릭(바로가기 버튼)일때는 이 패스에 연결해줘야하는
  int file_type;      // 파일의 타입. 0이면 일반 파일이고, 1이면 심볼릭이다.

};
```

- sysfile.c

  sysfile.c 파일의 sys_open 함수에 다음 부분을 추가했습니다. 만약 현재 open하려는 파일이 심볼릭 링크 파일이라면 중간에 path를 바꾸어서 심볼릭 링크가 가리키는 원본 파일의 inode를 읽어올 수 있도록 구현했습니다.

  만약 원본 파일이 삭제되어 읽어오는 것에 실패했다면, 그냥 현재 심볼릭 링크의 inode를 읽어올 수 있도록 하였습니다.

```
// 위에서 읽었는데 만약 바로가기 버튼이었다면?
    if(ip->file_type == SYMBOLIC) {
      flag = 1;
      inum_s = ip->inum;

      safestrcpy(sym_path, path, 12);
      path = ip->path; // 기존 패스를 끼워줌

      // 여기서 원본파일이 삭제된 경우를 처리
       if((ip = namei((path))) == 0) {
          if((ip = namei((sym_path))) == 0) {
            end_op();
            return -1;
          }
          ip->size = 0;
       }
    }
```

  또한 syslink 시스템 콜을 구현했습니다.

```
// Project 3
int
sys_symlink(void) {
  char *new, *old;
  struct inode *dp, *ip;
  // char name[DIRSIZ];

  if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
    return -1;

  begin_op();
  if((ip = namei(old)) == 0){ // old를 읽어옴
    end_op();
```

```
    return -1;
  }

  ilock(ip);
  if(ip->type == T_DIR){ // 디렉토리는 바로가기가 안됨
    iunlockput(ip);
    end_op();
    return -1;
  }
  strncpy((ip->path), old, 12); // origin path 끼워주고
  iunlock(ip);

  // 바로가기 아이콘을 만들어주자
  if((dp = create(new, T_FILE, 0, 0)) == 0) {
    end_op();
    return -1;
  }
  // if((dp = nameiparent(old, name)) == 0) {
  //   end_op();
  //   return -1;
  // }
  // ilock(dp);
  strncpy((dp->path), old, 12); // origin path 끼워주고
  dp->file_type = SYMBOLIC;
  iupdate(dp);
  iunlock(dp); // put하면 안되는듯..?

  end_op();

  return 0;
}
```

- ln.c

  '-s', '-h' 옵션을 구분하기 위해 ln.c를 다음과 같이 수정했습니다.

```
int
main(int argc, char *argv[])
{
  if(argc != 4){ // project 3에서 고침. 이제는 옵션까지 인자를 4개 받아야한다. 0
    printf(2, "Usage: ln old new\n");
    exit();
  }
  if(!strcmp(argv[1], "-h")) {
    if(link(argv[2], argv[3]) < 0)
      printf(2, "hard link %s %s: failed\n", argv[2], argv[3]);
  } else if(!strcmp(argv[1], "-s")) {
    if(symlink(argv[2], argv[3]) < 0) {
      printf(1, "symbolic link %s %s: falied\n", argv[2], argv[3]);
```

```
    }
  } else { // 옵션이 잘못됨
    printf(1, "ln faild. invalid option.\n");
  }

  exit();
}
```

- ls.c

  ls 명령어를 입력했을 때 원본 파일과 심볼릭 파일의 출력 정보가 달라야합니다. 따라서 다음과 같은 부분을 추가하여 각자의 inode 정보가 출력될 수 있도록 하였습니다.

```
if(st.ino != de.inum)
        printf(1, "%s %d %d %d\n", fmtname(buf), st.type, de.inum, 0);
      else
        printf(1, "%s %d %d %d\n", fmtname(buf), st.type, st.ino, st.si
```

- sync()

  기존 xv6는 버퍼가 부족하다면 패닉 상태가 됩니다. (bio.c의 bget함수에서)

  따라서 sync()를 구현해고, bget에서 남은 버퍼가 없다면 sync()를 호출하여 모든 변경사항을 디스크로 내려 버퍼를 비워주고 다시 버퍼 하나를 읽어오는 식으로 구현을 하려고 했습니다.

Result

- indirect table

16MB 이상의 파일을 write, read 할 수 있는지 확인하기 위해 stressfs.c를 다음과 같이 수정했습니다.

블록 하나의 크기가 512bytes이기에 32768개의 블록이 16MB입니다.

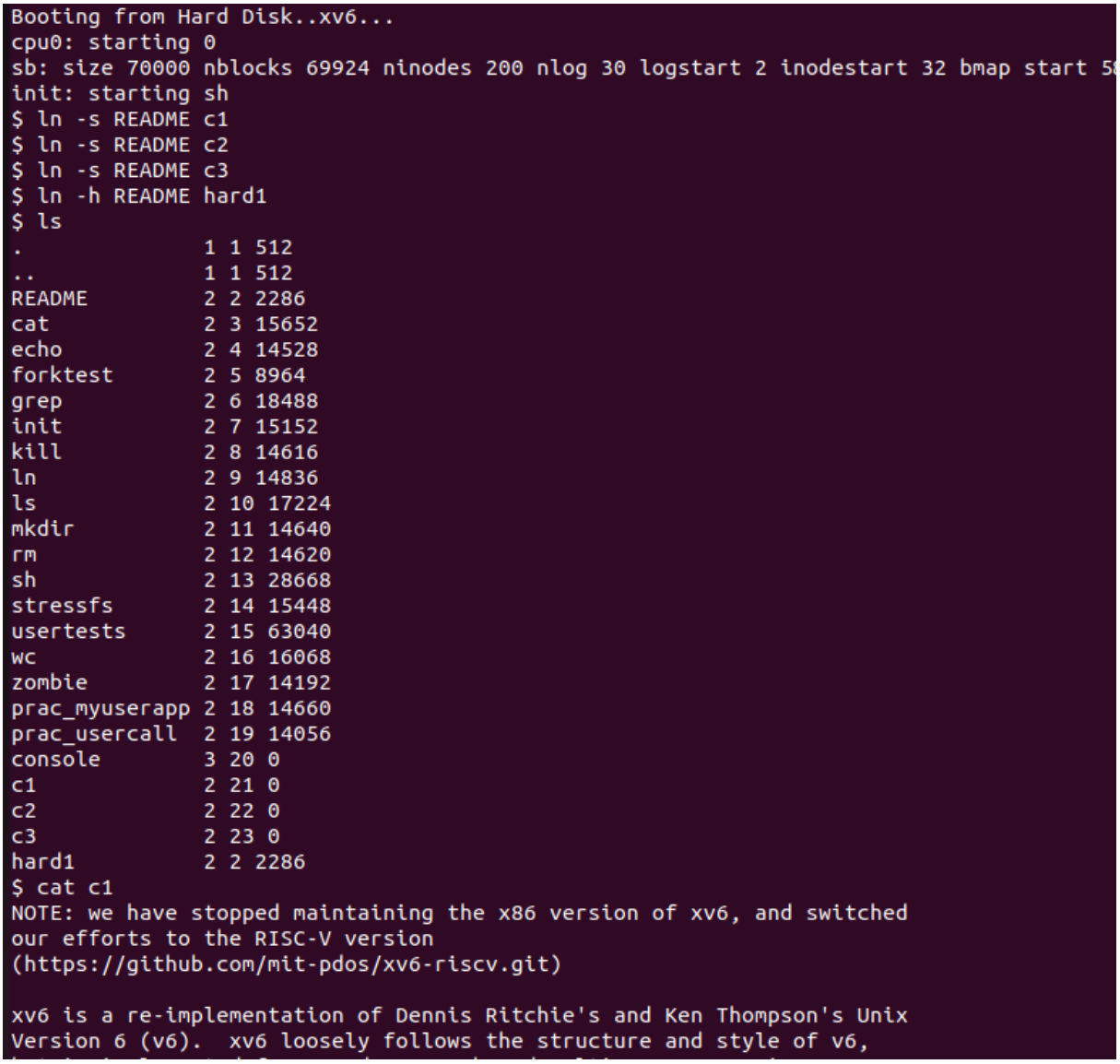
16MB를 초과하는 파일을 만들기 위해 32769개의 블록을 읽고 쓰도록 테스트코드를 변경했습니다.

```c
int
main(int argc, char *argv[])
{
  int fd, i = 0;
  char path[] = "stressfs0";
  char data[512];

  printf(1, "stressfs starting\n");
  memset(data, 'a', sizeof(data));

  // for(i = 0; i < 1; i++)
  //   if(fork() > 0)
  //     break;

  printf(1, "write %d\n", i);

  path[8] += i;
  fd = open(path, O_CREATE | O_RDWR);
  for(i = 0; i < 32769; i++) {
    // if(i % 30000 == 0)
    //   printf(1, "writing %d\n", i);
    write(fd, data, sizeof(data));
  }
  close(fd);

  printf(1, "read\n");

  fd = open(path, O_RDONLY);
  for (i = 0; i < 32769; i++)
    read(fd, data, sizeof(data));
  close(fd);

  wait();

  exit();
}
```

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 50000 nblocks 49929 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ stressfs
stressfs starting
write 0
read
$
```

성공적으로 읽고 써지는 것을 확인할 수 있었습니다.

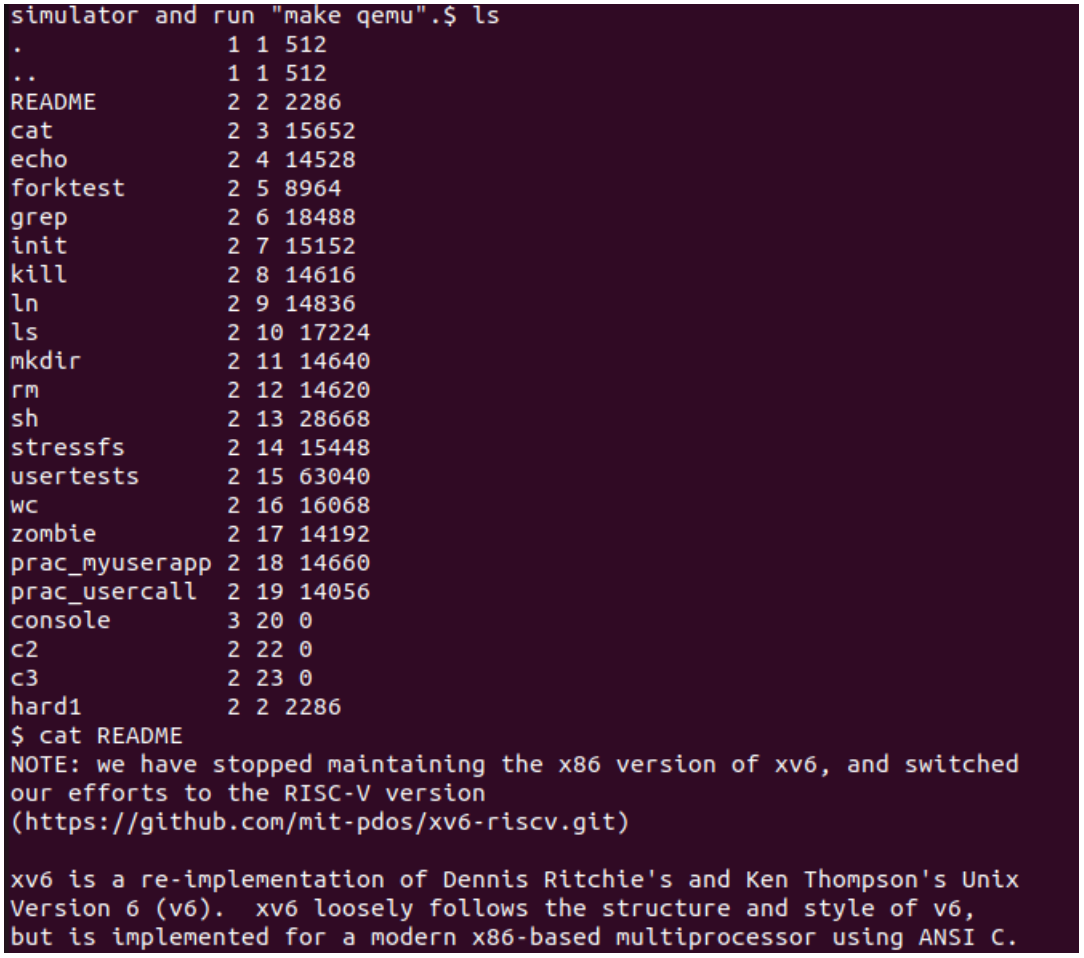
- symbolic link



README파일의 심볼릭 링크 c1, c2, c3를 만든 모습입니다.

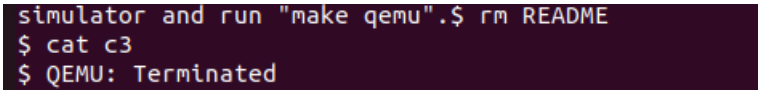
inode를 공유하지 않으므로 inode의 번호도 원본파일과 별개(원본: 2, 심볼릭들: 21, 22, 23)이며, 파일사이즈도 0으로 출력하였습니다.

'cat c1'시에도 README의 내용이 잘 출력됩니다.



위 사진은 'rm c1'을 한뒤 'ls'를 입력해보고 'cat README'한 실행 결과입니다.

rm c1을 하면 원본파일에 관계없이 심볼릭 링크 c1만 삭제되며, cat README시에도 README의 내용이 잘 출력됩니다.



'rm README'하여 원본파일을 없애고 'cat c3'이렇게 심볼릭 링크를 읽으려고 하면 아무것도 출력되지 않도록 구현했습니다. (아래 xv6의 종료는 제가 시킨 것이고 시스템 오류도 종료된 것이 아닙니다.)

```
$ ln -s stressfs1 stress
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15652
echo           2 4 14528
forktest       2 5 8964
grep           2 6 18488
init           2 7 15152
kill           2 8 14616
ln             2 9 14836
ls             2 10 17224
mkdir          2 11 14640
rm             2 12 14620
sh             2 13 28668
stressfs       2 14 15448
usertests      2 15 63040
wc             2 16 16068
zombie         2 17 14192
prac_myuserapp 2 18 14660
prac_usercall  2 19 14056
console        3 20 0
pp             2 21 0
stressfs1      2 22 16777728
stressfs0      2 23 16777728
stress         2 24 0
$ cat stress
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

대용량 파일의 심볼릭도 정상적으로 동작합니다.

Trouble Shooting

- 심볼릭 링크를 만들고 ls시에 원본파일과 심볼릭 링크의 정보가 동일한 것

  심볼릭과 원본파일은 각자의 inode를 가지므로 ls의 결과(inode 번호, size등)가 달라야합니다. 하지만 저는 둘이 같은 문제를 겪었었고, 'ls.c' 파일에서 해당 처리를 해주며 해결했습니다.

```
if(st.ino != de.inum)
  printf(1, "%s %d %d %d\n", fmtname(buf), st.type, de.inum, 0);
else
  printf(1, "%s %d %d %d\n", fmtname(buf), st.type, st.ino, st.size);
```

- 버퍼 아이오 구현 시 모든 프로세스가 sleep 상태가 되는 현상
  버퍼 아이오 구현 시에 모든 프로세스가 sleep 상태가 되어 멈춰버리는 현상을 겪었습니다. 이는 원인을 찾지 못하여 해결하지 못하였고, 결국 인다이렉트 테이블, 심볼릭 링크만 구현된 상태로 제출합니다.