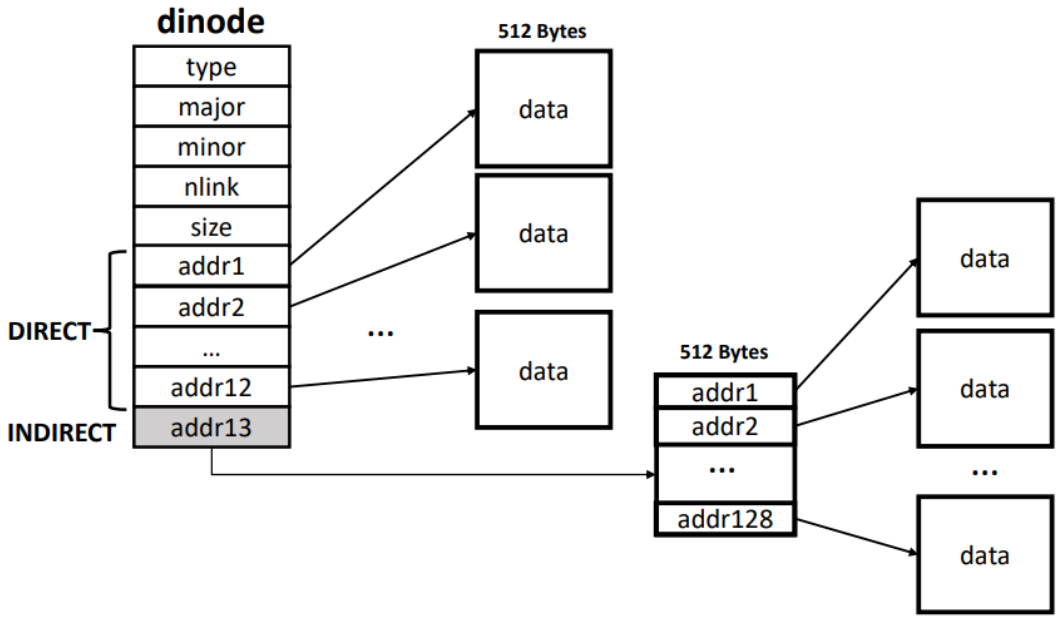
wiki

Design

Multi Indirect

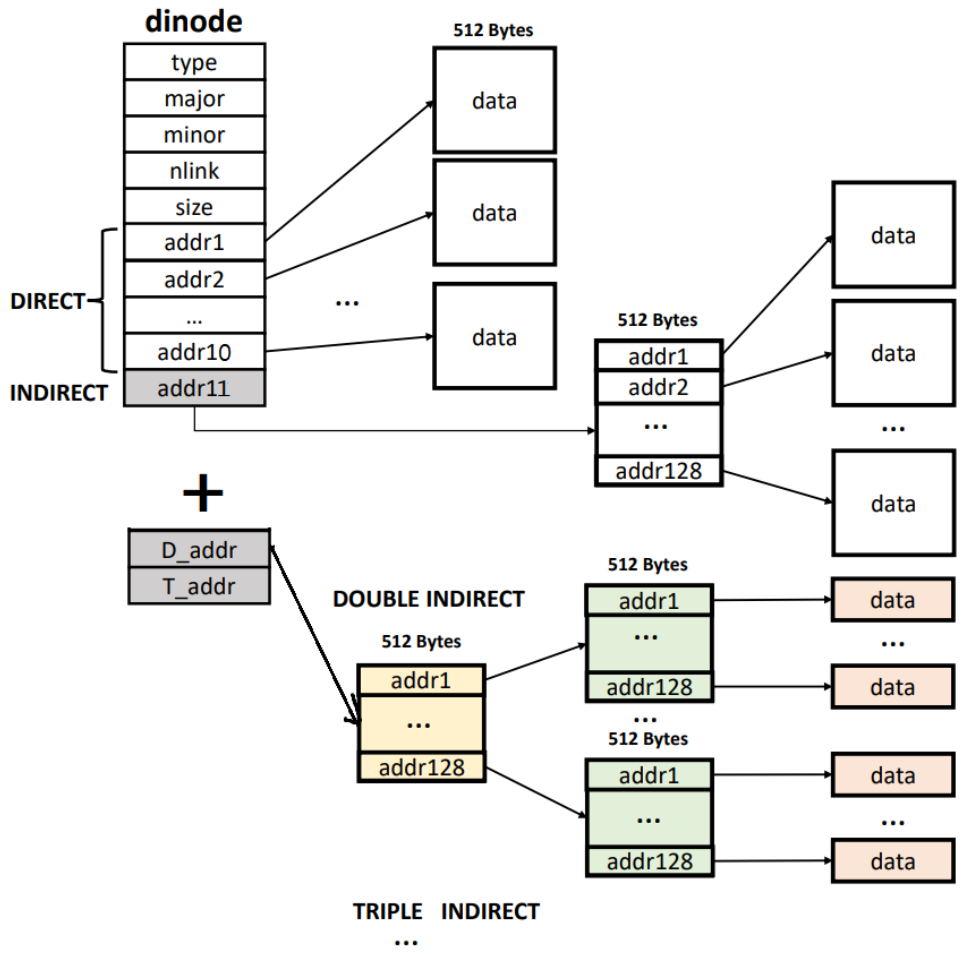
기존 xv6의 dinode 구조



기존의 xv6는 한 블록당 512Bytes의 데이터를 직접적으로 가리키는 direct 12개와 single indirect 1개를 통해 파일의 정보를 제공한다. 하지만, 이러한 방식으로는 최대 140(12+128)개 블록에 해당하는 파일만 쓸 수 있다. ($140 \times 512B = 70KB$)

변경된 dinode 구조

앞선 기존의 xv6의 문제점을 해결하기 위해, multi indirect, 그중에서도 double indirect 하나와 triple indirect 하나를 추가적으로 구현한다.



본 디자인에서는 기존의 direct와 single indirect를 제거하지는 않았다. 단, dinode 구조체의 전체 크기가 64bytes가 되는 것을 맞출 수 있도록 하기 위해, double indirect와 triple indirect를 위해 direct의 개수를 줄인다.

이에 따라 구조체 `dinode`(on disk inode), `inode`(in memory inode)의 addrs을 변경해줘야 할 것이다.

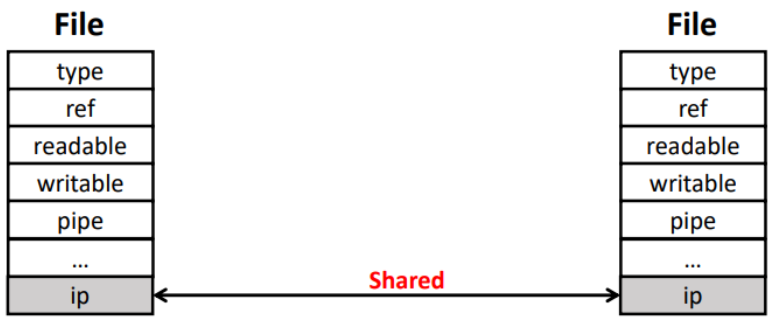
또한, 연관된 각종 상수값을 변경해줘야한다. `FSSIZE`나 `MAXBLOCKS`, `NDIRECT`등의 상수값을 조정해주어야 한다.

그리고, 원래 xv6 내부에는 single indirect를 읽는 것까지만 구현되어있는데, 더 나아가 double indirect, triple indirect까지 읽어올 수 있도록 추가적인 block number를 읽는 부분에 대한 처리를 추가해주어야 한다. 즉, `bmap`이나 `itrunc`와 같은 함수를 수정해주어야한다.

Symbolic link

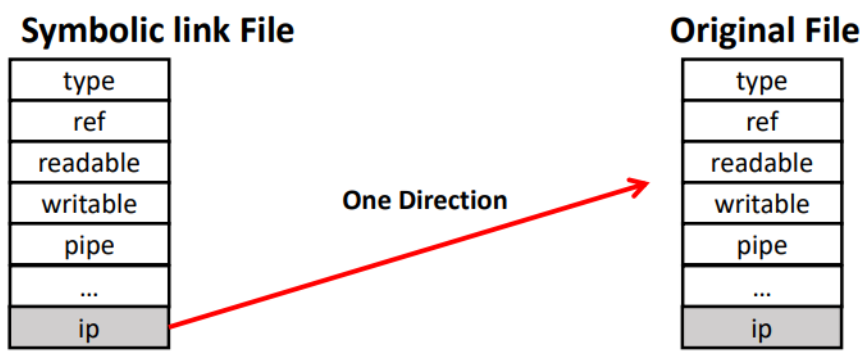
Hard link

Hard link 파일은 원본 파일의 i-node 자체를 공유하는 파일이다. 별개의 파일이지만, 같은 i-node 자체를 공유하므로 한쪽에서 파일을 변경하면 다른 한 쪽도 변경 내역을 공유하게 된다. 또한, 해당 i-node을 가리키는 파일이 몇개인지에 대한 값을 저장하여 한 파일이 삭제되어도 다른 파일에 영향을 주지 않고 잘 열려야 한다.

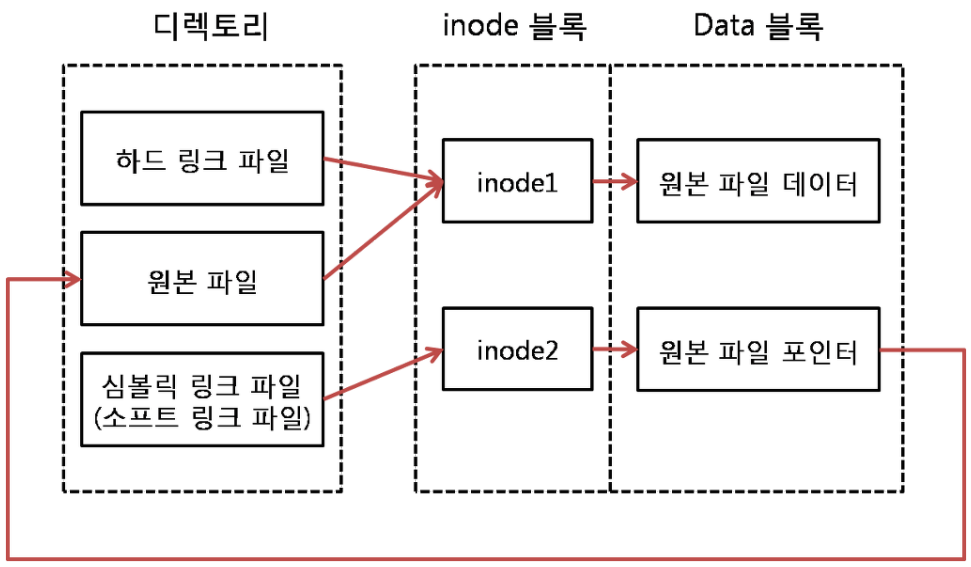


Symbolic link

'바로 가기' 기능과 비슷하다. Symbolic link는 고유의 i-node를 가지며, 그 i-node에서 다른 파일을 가리키게 된다. 즉, process가 symbolic link 파일에 접근하는 경우, original file로 redirection된다.



Symbolic link file



linux에서의 Hard link file과 symbolc link 파일의 비교

symbolic link 파일 만들기

```
ln -s {OLD} {NEW}
```

1. 우선, user program인 ln.c에서 기존의 아무 옵션도 주지 않을 때 하드 링크를 생성하게 하던 부분을 지우고, input 명령을 파싱하여 `-h` 옵션을 입력하면 하드 링크, `-s` 옵션을 입력하면 symbolic link를 생성하도록 수정한다.
2. symbolic link를 생성하는 시스템 콜(`sys_symlink`)를 만든다.

   symbolic link는 hard link처럼 link를 생성하려는 old file의 inode를 그대로 가리키는게 아니라, 새로운 inode를 가지고 원본 파일 포인터를 가리키는 형식이기 때문에, 우선적으로 inode를 생성해야할 것이다. 또한, 해당 파일이 symbolic link 파일임을 알 수 있도록 symbolic link 타입을 추가해서 해당되는 inode를 생성해야한다.

   그리고, 해당 새로운 inode의 data section에 원본 파일을 향한 경로 데이터를 저장하여 symbolic link 파일을 열 때 해당 원본 파일로 리다이렉션될 수 있도록 데이터를 쓰는 부분이 시스템 콜 안에 또한 필요할 것이다.

symbolic link 파일과 기존 system call과의 호환

1. open

   원래는 open하려는 file의 path의 inode를 얻어오고 그 inode로 이후의 작업이 진행되는데, 해당 파일 타입이 symbolic link인 경우 해당 link가 가지고 있는 데이터(원본 파일 path)를 읽어오고, 다시 돌아가 원본 파일의 path를 읽어온다(해당 작업은 링크가 다시 링크를 가리키는 등의 상황을 대비하여 반복적으로 이루어진다). 최종적으로 symbolic link 타입이 아닌 경우 해당 inode를 열 수 있게 된다.

2. exec

open과 마찬가지로, 원래는 입력받은 path에 해당하는 파일의 inode를 얻어 곧장 exec하였으나, 해당 파일이 symbolic link인지 확인하여 실제로 link가 가리키는 파일의 inode를 반환받은 뒤 exec하도록 수정한다.

3. write, read, close

   symbolic link를 write하거나 read할 때, close할 때는 파일을 open해야하는데, open에서 적절한 호환성을 갖춰주었다.

symbolic link 파일 정보 조회

추가적으로, UNIX의 symbolic link와 관련된 시스템 콜을 찾아보며 symbolic link 파일을 단순히 조회(예: `ls`) 할 때는, symbolic link파일을 open하는게 아니라 `readlink`라는 시스템 콜을 사용하여 정보를 읽어오는 것을 알 수 있었다. 만약 기존의 xv6처럼 stat을 통해 해당 파일의 정보를 읽어오려고 하게 되면, 내부적으로 `open` 시스템 콜이 호출되면서 symbolic link 파일 자체의 정보가 아닌 해당 링크 파일이 가리키는 원본 파일의 데이터가 출력되게 된다. 따라서, symbolic link파일인 경우 recursive하게 원본 파일을 찾지 않도록 readlink를 통해 symbolic link 파일 자체의 데이터를 읽도록 한다. (path에 해당하는 inode를 곧바로 반환하도록 함)

Sync

기존의 write operation에 관하여

기존의 xv6는 write operation에 대해 group flush를 진행한다. 해당 방식은 프로세스가 다수의 write operation을 발생시킬 때 성능을 저하시킬 수 있다(Disk I/O는 느리기 때문).

따라서, 해당 문제를 해결하기 위해 `sync`가 호출될때만 flush하는 Buffered I/O를 구현한다.

xv6에서는 file operation에 대해 transaction 단위로 작동한다. 해당 transaction을 시작할 때는 `begin_op`를 호출하고, 끝낼 때는 `end_op`를 호출한다. 이러한 transaction 단위의 동작에 기초하여, xv6에서는 logging system이 구현되어있다. logging은 carsh problem이 발생했을 때 이를 recovery하기 위한 장치이다. disk의 log section에 logging이 끝나고 나면, commit이 발생한다. xv6의 `begin_op`에서는 이러한 logging과 관련하여, 현재 logging system에 커밋중인 상태라면 sleep 상태로 대기하고, free log space가 없는 경우에도 commit이 될 때까지 기다리게 된다. 이러한 상태가 아니라면, log.outstanding을 증가시켜서 log space를 예약한다. 이후 `end_op`에서는 outstanding을 하나 감소시키고, outstanding 값이 0일 때(즉, 현재 logging중인 것이 없을 때) 현재의 transaction을 commit한다. 즉, FS system call의 outstanding이 없으면 group commit이 일어나게 된다.

Buffered I/O

기존의 Disk Operation은 많은 cost가 소모되는 작업이다. 따라서, sync가 호출될 때만 flush하도록 하는 Buffered I/O를 구현한다.

즉, 기존의 group flush 방식이 아니라, buffer cache에만 값을 쓰고, sync를 호출할 때만 실질적으로 disk에 로그와 데이터를 기록하도록 구현하였다.

1. end_op의 수정

   이를 위해, end_op에서 commit을 하는 조건에 'buffer가 가득 찼을 것'을 추가하였다. 이때, buffer가 가득 찼다는 것은, buffer의 DIRTY_BIT이 활성화되어 디스크에 써야하는(수정할 수 없는) 상태로 buffer 공간이 가득 찬 경우를 말한다. 그렇지 않다면, log 공간을 기다리는 begin_op가 있을 수 있으므로 wakeup(&log)를 호출하여 대기하던 로그 공간을 받을 수 있도록 한다. buffer가 가득차서 do_commit이 1로 설정되면 commit하여 사용가능한 buffer 공간을 확보한다(Dirty bit이 설정된 buffer의 로그 및 데이터를 디스크에 써서 buffer 공간을 확보한다).

   이때, begin_op가 아닌 end_op에서 확인하는 이유는 begin_op에서는 앞으로 어떤 write operation이 일어날지 모르기 때문이다.

2. sync 시스템콜 & sync 함수

   sync 시스템콜에서는 sync 함수를 호출하며, sync 함수 자체의 디자인은 간단하다. log를 commit 중이거나 여유 log space가 없는 경우가 아니라면(해당 경우는 sleep을 통해 대기하게 된다), commit 함수를 호출하여 버퍼의 데이터를 디스크에 로그와 데이터를 써준다.

Implementation

Multi Indirect

dinode, inode 구조체 및 상수값 변경

- file.h에서,
  - `inode` 구조체의 `addrs` 멤버 변수의 표현을 변경하였다.

```
// in-memory copy of an inode
// buffer cache 같은 역할을 수행
You, last week | 1 author (You)
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below he
  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]; // NDIRECT + 1 for original
  // 1 for D_addr, 1 for T_addr
};
```

기존에는 NDIRECT+1로, NDIRECT 12개에 SINGLE INDIRECT 하나를 나타냈으나, double indirect 하나와 triple indirect 하나가 추가되었으므로, (결국 현재 멤버 변수인 addrs의 length는 결과적으로는 12+1이 10+3이 되어 같긴 하나) NDIRECT가 10개, single indirect가 1개(기존에 있던 것), double indirect가 1개, triple indirect가 1개임을 내도록 수정하였다.

- fs.h에서,
  - NDIRECT를 10으로 감소시키고, MAXFILE의 크기를 Double indirect와 Triple indirect가 표현할 수 있는 범위까지 확장시켰다.

```
#define NDIRECT 10      You, last week • implement double & triple indirect
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT + NINDIRECT * NINDIRECT + NINDIRECT * NINDIRECT * NINDIRECT)
```

  - 또한, `dinode` 또한 NDIRECT+1로 표현되던 것을 NDIRECT+3으로 바꾸었다.

```
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_D
  short minor;          // Minor device number (T_D
  short nlink;          // Number of links to inode
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+3];   // Data block addresses
};
```

- param.h에서, 큰 파일을 읽고 쓰기 위해 FSSIZE를 변경하였다.

```
#define FSSIZE       2200000  // size of file system in blocks
```

`bmap` 변경

fs.c의 bmap에서,

`bmap`은 inode `ip`의 `bn`번째 데이터 블록의 block number를 리턴하는 함수이다. 원래는 single indirect의 블록 넘버까지 계산하는 로직이 있었으나, double indirect, triple indirect에 대해서도 블록 넘버를 구할 수 있도록 하는 코드를 추가해주었다. 구체적인 구현은 single indirect를 참고하였다.

- double indirect 구현

예를 들어, `bn`이 10 + 128 + 1인 경우,

첫번째 indirect block에서는 bn/NINDIRECT = 0의 index를 할당받게 된다. (컴퓨터 index상 0부터 시작하지만 이론적으로는 첫번째 블록 넘버)

이후, 1 % 128 = 1이므로, 두번째 depth의 indirection block에서는 1의 index를 할당받게 된다.

```
// double indirect
if (bn < NINDIRECT * NINDIRECT)
{
  // Load double indirect block, allocating if neces
  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 * NINDIRECT;
```

- triple indirect의 구현

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

  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;
  // load doubly indirect block, allocating if necessary
  if ((addr = a[bn % (NINDIRECT)]) == 0)
  {
    a[(bn % NINDIRECT)] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);
  return addr;
}
```

`bn`이 10 + 128 + 128*128 + 1인 경우,

첫번째 indirect block에서는 1 / 128*128 = 0의 index를 할당받는다.

이후, 두번째 depth의 indirect block에서는, 1 % (128*128) = 1의 index를 부여받는다.

마지막 세번째 depth의 indirect block에서는, (1 % (128*128)) % 128 = 1의 index를 부여받는다.

itrunc 변경

`itruc`는 inode의 모든 블록을 deallocate하고, size를 0으로 리셋하는 함수이다. 원래는 single indirect까지 할당 해제하는데, 이제 double indirect와 triple indirect가 추가되었으므로 해당 부분까지 할당되었는지 확인하고 할당되어있다면 할당 해제해주는 부분을 추가했다. 이부분은 관련 실습 2번째 시간에 굳이 수정하지 않아도 괜찮다고 언급해주셨으나, 이미 추가했었기 때문에 삭제하지 않았다.

- double indirect
- triple indirect

```c
// double indirect 할당 해제(할당 되어있는 경우)
if (ip->addrs[NDIRECT+1])
{
  bp = bread(ip->dev, ip->addrs[NDIRECT+1]); // indirect pointer로 block을 read
  a = (uint *)bp->data;
  for (j = 0; j < NINDIRECT; j++)
  {
    if (a[j]){
      bp2 = bread(ip->dev, a[j]);
      a2 = (uint *)bp2->data;

      for (k = 0; k < NINDIRECT; k++) {
        if (a2[k])
          bfree(ip->dev, a2[k]);
      }

      brelse(bp2);
      bfree(ip->dev, a[j]);
    }
  }
  brelse(bp);                          // indirect pointer 블록 닫기
  bfree(ip->dev, ip->addrs[NDIRECT+1]); // 해제된 indirect pointer로 block을 파괴
  ip->addrs[NDIRECT+1] = 0;
}
```

```c
// triple indirect 할당 해제(할당 되어있는 경우)      You, 1 second ago • Uncommi
if (ip->addrs[NDIRECT+2])
{
  bp = bread(ip->dev, ip->addrs[NDIRECT+2]); // indirect pointer로 block을 read
  a = (uint *)bp->data;
  for (j = 0; j < NINDIRECT; j++)
  {
    if (a[j]){
      bp2 = bread(ip->dev, a[j]);
      a2 = (uint *)bp2->data;

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

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

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

      brelse(bp2);
      bfree(ip->dev, a[j]);
    }
  }
  brelse(bp);                          // indirect pointer 블록 닫기
  bfree(ip->dev, ip->addrs[NDIRECT+2]); // 해제된 indirect pointer로 block을 파괴
  ip->addrs[NDIRECT+2] = 0;
}
```

Symbolic link

ln 유저 프로그램 수정

```c
int
main(int argc, char *argv[])
{
  if(argc != 4){
    printf(2, "Usage 1: ln [-h] [old] [new]\n");
    printf(2, "Usage 2: ln [-s] [old] [new]\n");

    exit();
  }

  if(!strcmp(argv[1], "-h")) {
    if(link(argv[2], argv[3]) < 0)
      printf(2, "link %s %s: failed\n", argv[2], argv[3]);
  }
  else if (!strcmp(argv[1], "-s")) {
    if(symlink(argv[2], argv[3]) < 0)
      printf(2, "symlink %s %s: failed\n", argv[2], argv[3]);
  }
  sync();
  exit();
}
```

다음과 같이 유저 프로그램인 `ln`을 수정하여, `ln -h old new`를 쉘에 입력하면 hard link를, `ln -s old new`를 입력하면 soft link(symbolic link)를 만들 수 있도록 하였다.

(optional) 추가적으로 프로젝트의 세번째 목표인 sync를 추가해서 실제로 디스크에 쓰는 과정까지 추가하였다.

symlink 시스템 콜 구현

```c
ublic > C stat.h > T_DIR
You, 19 hours ago | 1 author (You)
#define T_DIR  1   // Directory
#define T_FILE 2   // File
#define T_DEV  3   // Device
#define T_SYMLINK 4 // Symbolic link
```

symbolic link 파일이라는 것을 알 수 있도록 T 대응되는 타입을 추가해주었다.

```c
int sys_symlink(void)
{
  char *newLink, *oldPath;
  struct inode *ip;

  if (argstr(0, &oldPath) < 0 || argstr(1, &newLink) < 0)
    return -1;

  begin_op();

  // TODO: symbolic link는 hard link와 다르게 inode를 새롭게
  // 그러므로, namei를 통해 old path의 inode를 가져오는 것이 
  if ((ip = create(newLink, T_SYMLINK, 0, 0)) == 0)
  {
    end_op();
    return -1;
  }

  // old를 inode의 data block에 써줌
  writei(ip, oldPath, 0, strlen(oldPath));
  iupdate(ip);

  iunlockput(ip);

  end_op();

  return 0;
}
```

디자인에서 언급한 바와 같이, symbolic link는 원본 파일의 inode를 그대로 가리키는 hard link와 달리 자신만의 inode를 가지며 해당 inode에서 원본 파일에 대한 포인터를 저장하여 원본 파일의 inode로 리다이렉션될 수 있도록 한다. 따라서, 우선 inode를 `create`하는 과정부터 시작된다. T_SYMLINK 타입을 갖는 inode를 생성하고, 해당 inode에 입력받은 oldPath(원본 파일)의 데이터를 write한다. 그리고 이러한 수정을 dinode에 반영하는 `iupdate`를 호출한뒤, 잡았던 lock을 풀고 함수를 종료한다.

기존 시스템 콜과의 호환

- sys_open

```c
  }
  else
  {
  repeatSymlink:
    if ((ip = namei(path)) == 0)
    {
      end_op();
      return -1;
    }
    ilock(ip);

    // 열려는 파일이 symlink인 경우, ip를 symlink path가 가리키는 것으로 업데이트
    if (ip->type == T_SYMLINK)
    {
      char pathBuf[512];

      for (int bi = 0; bi < 512; bi++)
      {
        pathBuf[bi] = 0;
      }

      readi(ip, pathBuf, 0, 512);
      safestrcpy(path, pathBuf, 512);
      iunlockput(ip);
      goto repeatSymlink;
    }

    // 열려는 파일이 directory인데 write하려고 하면 에러
```

symbolic link 파일을 열려고 하는 경우, ip가 링크 파일이 아닌 실제 파일을 가리킬 때까지 goto문을 사용하여 recursive하게 해당 inode에 담긴 데이터를 readi하여 다시 해당 path의 파일을 열려고 시도한다. 계속 루프를 돌다가 최종적으로 symbolic link 타입이 아닌 경우 해당 inode를 열 수 있게 된다.

- exec

입력받은 path에 해당하는 inode를 바로 `namei`로 읽어오지 않고,

```
struct proc *curproc = myproc();

begin_op();

if((ip = getOriginInode(path)) == 0) {
  cprintf("exec: fail\n");
  return -1;
}

pgdir = 0;
```

symbolic link인 경우를 대비하여 원본 파일의 inode를 찾아온다.

원본 파일의 inode를 반환하는 `getOriginInode` 함수의 구현은 sys_open에서의 동작과 비슷하며, 구현은 다음과 같다.

```
You, 9 seconds ago | 1 author (You)
struct inode *getOriginInode(char *path)        You, 19 hours ago • implement
{
  struct inode *ip;
repeatSymlinkExec:
  if ((ip = namei(path)) == 0)
  {
    end_op();
    return 0;
  }
  ilock(ip);

  // 열려는 파일이 symlink인 경우, ip를 symlink path가 가리키는 것으로 업데이트
  if (ip->type == T_SYMLINK)
  {
    // safestrcpy(path, (char*)ip->addrs, strlen((char*)ip->addrs));
    char pathBuf[512];

    for (int i = 0; i < 512; i++)
    {
      pathBuf[i] = 0;
    }

    readi(ip, pathBuf, 0, 512);
    safestrcpy(path, pathBuf, 512);

    iunlockput(ip);
    goto repeatSymlinkExec;
  }

  return ip;
}
```

readline 시스템 콜 추가

`ls` 등으로 파일 정보를 조회할 때, 파일 정보를 알기 위해 내부적으로 `stat` 함수가 호출된다. 그런데, 이 `stat`에서는 파일을 `open`하여 정보를 조회하기 때문에, symbolic link 파일의 경우 앞서서 sys_open에서 원본 파일로 리다이렉션 되도록 했기 때문에 원본 파일의 정보를 반환하게 된다. 이로 인해, symbolic link 파일을 읽는 readlink 시스템 콜을 추가했다. 이로써 symbolic link 파일의 정보를 ls로 조회할 수 있게 된다.

```
int stat(const char *n, struct stat *st)        You, 19 ho
{
  int fd;
  int r;
  char buffer[512];

  char path[512];
  strcpy(path, n);

  if ((fd = readlink(path, buffer, strlen(path))) == -1)
  {
    fd = open(n, O_RDONLY);
  }

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

  return r;
}
```

stat 함수의 수정

```c
int sys_readlink(void)
{
  char *sympath, *buffer;
  int bufsize;
  int fd;
  struct file *f;
  struct inode *ip;

  if (argstr(0, &sympath) < 0 || argstr(1, &buffer) < 0 || argint(2, &bufsize))
    return -1;

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

  ilock(ip);

  if (ip->type != T_SYMLINK)
  {
    iunlockput(ip);
    end_op();
    return -1;
  }

  // TODO: symlink의 path에 원본 파일이 존재하는지 확인?
  // 필요없음. 어차피 symlink가 가리키는 곳에 파일이 있든 없든 symlink가 사라지는 건 아니니까.

  safestrcpy(buffer, (char *)ip->addrs, bufsize);

  if ((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0)
  {
    if (f)
      fileclose(f);
    iunlockput(ip);
    end_op();
    return -1;
  }
  iunlock(ip);
  end_op();

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

  return fd;
}
```

readlink 시스템 콜 함수

Sync

end_op의 수정

end_op에서, `do_commit`의 플래그를 설정해주는 조건문에, buffer가 다 찼는지 아닌지 확인하는 함수인 `bfull`의 조건을 추가했다. 따라서, 일반적인 상황에서는 기존처럼 commit이 일어나지 않게 될 것이다.

```c
int
bfull(void) {       You, 19 hours ago • implement project3 (s
  struct buf *b;

  acquire(&bcache.lock);

  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if((b->flags & B_DIRTY) == 0){
      release(&bcache.lock);
      return 0;
    }
  }

  release(&bcache.lock);
  return 1;
}
```

bio.c에 구현된 `bfull()`: B_DIRTY 플래그를 가지는 buffer로만 가득차있다면, 더이상 Buffer에 새롭게 뭔가를 쓸 수 없는 상황이 되므로, flush해주어야 한다.

```c
// called at the end of each FS system call.
// commits if this was the last outstanding operation.
void end_op(void)
{       You, 19 hours ago • implement project3 (squash merge) …
  // 로그에서 데이터를 뺌
  int do_commit = 0;

  acquire(&log.lock);
  log.outstanding -= 1; // 더이상 나는 하는게 없으므로 현재 transaction을 종료시킴
  if (log.committing)
    panic("log.committing");

  // outstanding이 0: 나를 제외하고 그 누구도 연산을 하고 있지 않고 나도 안쓰는중
  // 즉, 모든 transaction이 완료되었으므로, commit 작업을 수행
  if (log.outstanding == 0 &&
      ((log.lh.n + (log.outstanding + 1) * MAXOPBLOCKS > LOGSIZE) || bfull()))
  {
    do_commit = 1; // 이제 커밋해도 되겠다
  }
  else
  {
    // commit이 가능하지 않은 경우, log 공간을 기다리는 begin_op가 있을 수 있으므로,
    // wakeup(&log)를 통해 대기중인 프로세스가 깨어나도록 하여 로그 공간을 할당받음
    // 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)
  {
    commit();

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

sync의 구현

log.c의 commit()함수에서는 write_log, write_head, install_trans 등의 함수를 호출하여 디스크에 로그와 헤더 정보를 쓴 뒤, 실제 데이터를 디스크에 쓰게 된다. 하지만, 이제 end_op에서는 buffer cache가 다 찼을 때만 커밋을 해주도록 수정했기 때문에, sync를 호출해야 buffer cache에서 디스크로 commit이루어지게 된다.

```c
int sync(void)
{
  int blockCnt = 0;

  acquire(&log.lock);

  // 이미 commit 중이거나 다른 transaction이 동작 중이 때는 sleep으로 대기
  while (log.committing || log.outstanding)
  {
    sleep(&log, &log.lock);
  }

  blockCnt = log.lh.n;
  log.committing = 1;
  release(&log.lock);

  commit();

  acquire(&log.lock);

  log.committing = 0;
  wakeup(&log);

  release(&log.lock);

  return blockCnt;
}
```

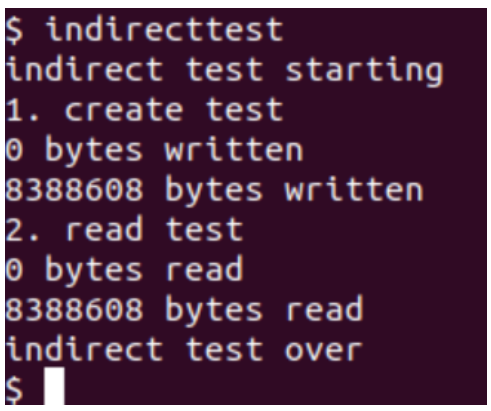
```c
int sys_sync(void)
{
  // 성공시 flush된 block의 수를, 실패시 -1를 반환
  return sync();
}
```

log.c에 구현된 `sync`: commit을 호출하여 디스크에 버퍼 캐시의 데이터를 쓴다. 성공한 경우 flush된 블록 수를 반환한다.

Result

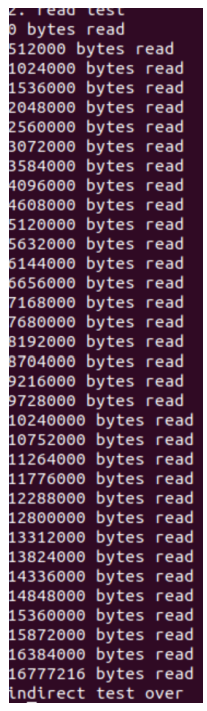
Multi Indirect: 용량이 큰 파일 읽어보는 테스트

1. 8MB짜리 파일 읽어보기



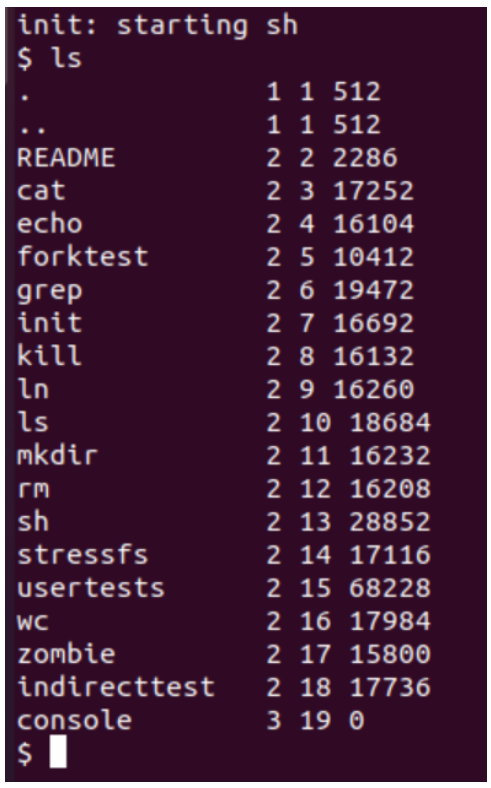
2. 16MB짜리 파일 읽어보기

16*1024*1024 byte의 파일을 잘 읽는 것을 확인할 수 있다. (다만, 테스트하는데에 다소 시간이 소요된다.)



Symbolic link: symlink 생성, 해당 파일 open, ls로 조회

1. xv6 부팅 후 `ls` 커맨드를 입력하였다.



2. `ln -s README sREADME`를 통해 symbolic link 생성하였다.

3. 생성된 symlink 조회(`ls`)

   내부적으로는, symbolic link file의 stat을 조회하기 위해서는 `readlink`라는 system call이 호출된다.

4. 기존의 README 파일과, symbolic link로 생성된 sREADME 파일 모두 잘 읽을 수 있는지 테스트하였다.

   또한, cat의 경우 내부적으로 read 시스템 콜을 호출하는데, 문제 없이 잘 읽어지는 것을 확인할 수 있었다.

```
$ cat README
NOTE: we have stopped maintaining the x86 version of xv6, and switched
our efforts to the RISC-V version
(https://github.com/mit-pdos/xv6-riscv.git)

xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
Version 6 (v6).  xv6 loosely follows the structure and style of v6,
but is implemented for a modern x86-based multiprocessor using ANSI C.

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14,
2000)). See also https://pdos.csail.mit.edu/6.828/, which
provides pointers to on-line resources for v6.

xv6 borrows code from the following sources:
    JOS (asm.h, elf.h, mmu.h, bootasm.S, ide.c, console.c, and others)
    Plan 9 (entryother.S, mp.h, mp.c, lapic.c)
    FreeBSD (ioapic.c)
```

```
$ cat sREADME
NOTE: we have stopped maintaining the x86 version of xv6, and switched
our efforts to the RISC-V version
(https://github.com/mit-pdos/xv6-riscv.git)

xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
Version 6 (v6).  xv6 loosely follows the structure and style of v6,
but is implemented for a modern x86-based multiprocessor using ANSI C.

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14,
2000)). See also https://pdos.csail.mit.edu/6.828/, which
provides pointers to on-line resources for v6.

xv6 borrows code from the following sources:
    JOS (asm.h, elf.h, mmu.h, bootasm.S, ide.c, console.c, and others)
    Plan 9 (entryother.S, mp.h, mp.c, lapic.c)
    FreeBSD (ioapic.c)
    NetBSD (console.c)
```

5. 새로운 symbolic link 파일을 지워도 기존 파일에는 영향이 없는 것 확인하였다.

```
$ ln -s README sREADME
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 17252
echo           2 4 16104
forktest       2 5 10412
grep           2 6 19472
init           2 7 16692
kill           2 8 16132
ln             2 9 16260
ls             2 10 18684
mkdir          2 11 16232
rm             2 12 16208
sh             2 13 28852
stressfs       2 14 17116
usertests      2 15 68228
wc             2 16 17984
zombie         2 17 15800
indirecttest   2 18 17736
console        3 19 0
sREADME        4 20 6
$ rm sREADME
$ cat README
NOTE: we have stopped maintaining the x86 versi
our efforts to the RISC-V version
(https://github.com/mit-pdos/xv6-riscv.git)
```

6. exec를 수정하여, 프로그램 실행도 symbolic link로 잘 되는 것을 확인하였다.

```
$ ln -s forktest sfork
$ sfork
fork test
fork test OK
$
```

7. 기존 파일을 지웠을 때, 실행되지 않는 것을 확인하였다.

```
$ rm forktest
$ sfork
exec: fail
exec sfork failed
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 17272
echo           2 4 16124
grep           2 6 19492
init           2 7 16712
kill           2 8 16156
ln             2 9 16312
ls             2 10 18708
mkdir          2 11 16252
rm             2 12 16232
sh             2 13 29236
stressfs       2 14 17140
usertests      2 15 68252
wc             2 16 18008
zombie         2 17 15824
indirecttest   2 18 17840
synctest       2 19 17324
console        3 20 0
sfork          4 21 8
```

8. 기존의 hard link도 잘 되는 것 확인하였다. (forktest가 내부적으로 write를 포함하는데, 문제없이 잘 실행되는 것을 확인할 수 있었다)

```
sfork          4 21 8
$ ln -h forktest hfork
$ hfork
fork test
fork test OK
$ rm forktest
$ hfork
fork test
fork test OK
$
```

Sync

1. sync를 내부적으로 호출하지 않고 file을 닫은 경우, 버퍼에 캐싱된 testfile이 출력된다.

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 17272
echo           2 4 16124
forktest       2 5 10440
grep           2 6 19492
init           2 7 16712
kill           2 8 16156
ln             2 9 16284
ls             2 10 18708
mkdir          2 11 16252
rm             2 12 16232
sh             2 13 28876
stressfs       2 14 17140
usertests      2 15 68252
wc             2 16 18008
zombie         2 17 15824
indirecttest   2 18 17840
synctest       2 19 17180
console        3 20 0
$ synctest
synctest: starting..
SyncTestFile 2 21 0

After filewrite
SyncTestFile 2 21 512
synctest: test over..
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 17272
echo           2 4 16124
forktest       2 5 10440
grep           2 6 19492
init           2 7 16712
kill           2 8 16156
ln             2 9 16284
ls             2 10 18708
mkdir          2 11 16252
rm             2 12 16232
sh             2 13 28876
stressfs       2 14 17140
usertests      2 15 68252
wc             2 16 18008
zombie         2 17 15824
indirecttest   2 18 17840
synctest       2 19 17180
console        3 20 0
SyncTestFile   2 21 512
$
```

- 다시 재부팅하자, file을 close하기 전에 sync를 호출하지 않았으므로 파일이 없어진 것을 확인할 수 있다.

```
sb: size 2200000 nblocks 219940
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 17272
echo           2 4 16124
forktest       2 5 10440
grep           2 6 19492
init           2 7 16712
kill           2 8 16156
ln             2 9 16284
ls             2 10 18708
mkdir          2 11 16252
rm             2 12 16232
sh             2 13 28876
stressfs       2 14 17140
usertests      2 15 68252
wc             2 16 18008
zombie         2 17 15824
indirecttest   2 18 17840
synctest       2 19 17180
console        3 20 0
$
```

2. sync를 내부적으로 호출한 경우, 재부팅해도 파일이 남아있는 것을 확인할 수 있다. (디스크에 썼기 때문)

```
$ synctest
synctest: starting..
SyncTestFile 2 21 0

After filewrite
SyncTestFile 2 21 512
flushCnt: 4
synctest: test over..
$ ls
.               1 1 512
..              1 1 512
README          2 2 2286
cat             2 3 17272
echo            2 4 16124
forktest        2 5 10440
grep            2 6 19492
init            2 7 16712
kill            2 8 16156
ln              2 9 16284
ls              2 10 18708
mkdir           2 11 16252
rm              2 12 16232
sh              2 13 28876
stressfs        2 14 17140
usertests       2 15 68252
wc              2 16 18008
zombie          2 17 15824
indirecttest    2 18 17840
synctest        2 19 17324
console         3 20 0
SyncTestFile    2 21 512
$
```

```
sb: size 2200000 nblocks 219940
init: starting sh
$ ls
.               1 1 512
..              1 1 512
README          2 2 2286
cat             2 3 17272
echo            2 4 16124
forktest        2 5 10440
grep            2 6 19492
init            2 7 16712
kill            2 8 16156
ln              2 9 16284
ls              2 10 18708
mkdir           2 11 16252
rm              2 12 16232
sh              2 13 28876
stressfs        2 14 17140
usertests       2 15 68252
wc              2 16 18008
zombie          2 17 15824
indirecttest    2 18 17840
synctest        2 19 17212
console         3 20 0
SyncTestFile    2 21 512
$
```

재부팅하여 ls를 실행한 모습

```
$ cat SyncTestFile
Hello world!
$
```

썼던 내용이 잘 읽어지는 모습

Trouble Shooting

Multi Indirect

구조체 외의 어떤 함수를 수정해야하는지 처음에 감이 오지 않았었으나, single indirect가 활용되는 부분을 따라가보면서 어느 부분의 코드를 더 수정해야할지 알 수 있었다.

그외에 블록 넘버를 설정하는 부분은 명세의 그림을 보면서 숫자를 계산해보았었다.

Symbolic Link

symbolic link에 멤버 변수 추가

inode에 symbolic link용으로 리다이렉트될 path를 가리킬 `char* path` 변수를 두었는데, 동시에 dinode 구조체에도 path 멤버 변수를 추가했었다. 그러나, dinode는 한 블럭(512Byte) 안에 8개가 들어갈 수 있도록 하나당 반드시 64Byte의 크기를 가져야했다. 그래서 멤버 변수 추가가 아닌 inode의 addrs를 이용해보는 것으로 수정했다.

symbolic link를 열 수 없는 문제

sys_open에서, inode의 type이 `T_SYMLINK`이면

```
safestrcpy(path, (char*)ip->addrs, strlen((char*)ip->addrs))
```

를 하도록 했다.

근데 ip→addrs를 strcpy로 내용을 복사해오는게 아니라 readi를 해야했다.

file을 open하는 경우 무조건 link가 가리키는 곳으로 redirect되는 문제

cat, ls(stat) 등에서는 파일 정보를 나타내기 위해 open 시스템 콜을 사용하는데,

```
repeatSymlink:
    if ((ip = namei(path)) == 0)
    {
      end_op();
      return -1;
    }
    ilock(ip);

    // 열려는 파일이 symlink인 경우, ip를 symlink path가 가리키는 것으로 업데이트
    if (ip->type == T_SYMLINK)
    {
      // safestrcpy(path, (char*)ip->addrs, strlen((char*)ip->addrs));
      char pathBuf[512];
      readi(ip, pathBuf, 0, 512);
      safestrcpy(path, pathBuf, 512);

      cprintf("sys_open: %d\n", path);
      iunlockput(ip);
      goto repeatSymlink;
    }
```

위와 같이 symbolic link일 때 해당하는 경로로 redirect 되도록 구성했을 때, ls나 cat 등 open system call을 사용하는 모든 곳에서 symbolic link의 정보가 아닌 symbolic link가 redirect하는 곳으로 바로 연결이 되어버려서 symbolic link에 대한 정보를 볼 수 없게 되는 문제가 생겼었다.

UNIX 계열 운영체제에서는, symbolic link를 open하는게 아니라 단순히 파일을 조회하려고 할 때는 `readlink`라는 시스템 콜을 사용한다고 한다. 그래서 본 프로젝트에서도 symbolic link 자체의 inode 정보를 조회하기 위한 `readlink` 시스템 콜을 추가하였다.

UNIX의 시스템 콜 종류를 참고한 곳은 다음과 같다.

> readlink(2): read value of symbolic link - Linux man page
>
> readlink() places the contents of the symbolic link path in the buffer buf, which has size bufsiz. readlink() does not append a null byte to buf. It will ...
>
> https://linux.die.net/man/2/readlink

Sync

Sync 자체의 디자인

워드 파일을 열심히 쓰다가(쓰면 곧장 쓴 내용이 보인다. 단, 디스크에 쓴게 아니라 메모리의 버퍼 캐시에 쓴 것이 보이게 되는 것이다.), 아직 저장을 안했는데 컴퓨터 전원이 나갔을 때, 다시 컴퓨터를 켜면 쓰던 파일 내용이 없어진 상황을 떠올리면서 sync의 동작 방식에 관해 고민해 보았다.

다만 추후 piazza를 통해 쉘로 돌아온 직후에 반영이 되지 않아야한다고 말씀해주신 답변을 보았는데, 해당 조건과 관련해서는 이론 시간에 배웠던 maintaining copy semantics을 유사하게 구현해야 될 것 같다는 생각이 들었었다. 즉, kernel buffer와 application buffer를 따로 두는 것이다. 유저 프로그램에서 file write가 일어나게 되면 kernel buffer가 아닌 application buffer에 데이터를 쭉 쓴다. 그리고 sync가 호출되면 kernel buffer와의 동기화가 일어나고, kernel buffer에서 디스크에 데이터를 써주게 된다. 쉘 등 다른 프로세스에서 read를 할 때는 기본적으로 kernel buffer의 값을 읽어온다. 하지만, 만약 user application에서 application buffer에만 데이터를 쭉 쓰다가 sync를 호출하지 않고 종료해버린 경우, 해당 프로세스가 종료되면 application buffer에 있던 데이터는 사라져버리고(disable되고), 쉘로 돌아올 경우 동기화되지 않은 kernel buffer cache의 데이터를 읽게 되므로 piazza 답변에 더 부합하도록 만들 수 있을 것이다. 해당 디자인은 과제 제출 기간 안에 제대로 부팅될 수 있을 정도로 구현하지 못하여, 우선은 아예 롤백하여 제출하게 되었다.

lock panic

본래 `begin_op`에서 buffer cache가 다 찼는지 확인하는 것이라고 생각했어서, system call에서 부르는 경우는 lock을 잡지 않은채로 진입하지만 begin_op에서 sync를 호출하는 경우에는 lock을 이미 잡고 들어오므로 lock을 잡았는지 안잡았는지에 대한 플래그를 sync의 input parameter로 설정했었다. 하지만, 좀더 생각해보니 begin_op에서는 아직 write를 하지 않았으므로, 새로운 buffer에 쓰는건지 cache의 내용

을 수정하게 되는 것인지 알 수 없다. 따라서, begin_op가 아니라 file system transaction이 끝날 때 호출되는 end_op에서, buffer를 다 썼는지 확인하도록 수정하여, 해당 panic은 아예 상관이 없어졌다.