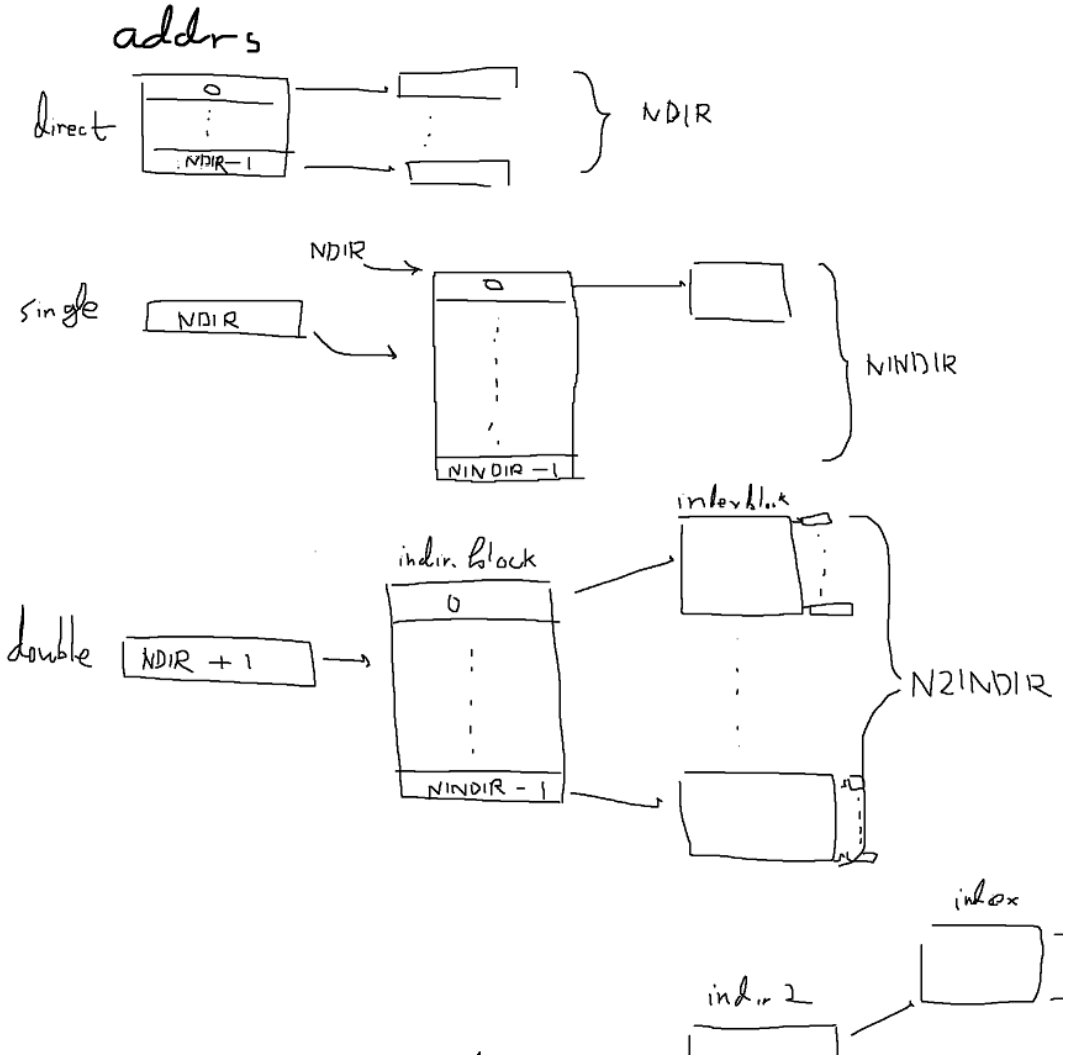
ELE3021 Project3 Wiki

File System

2020059152 오주영

1. Multi Indirect



초안: INODE의 ADDRS 구조

Multi indirect가 제대로 구현되기 위해서는 bmap 함수를 수정해주어야 한다. bmap 함수는 인자로 받은 bn (block number) 논리주소를 inode의 addrs 배열, 즉 index table의 entry로 변환해주어야 한다. Multilevel paging의 주소 변환과 비슷하게 일련의 연산으로 이런 변환을 수행할 수 있는데, 이를 위해 먼저 상수값을 몇몇 새로 정의 및 수정해주었다.

```
#define NDIRECT 10
#define NINDIRECT (BSIZE / sizeof(uint))  // ~ 128: single indirect
#define N2INDIRECT (NINDIRECT * NINDIRECT)
#define N3INDIRECT (N2INDIRECT * NINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT + N2INDIRECT + N3INDIRECT)

// 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];
};
```

Double indirect를 위해 N2INDIRECT를, triple indirect를 위해 N3INDIRECT 값을 fs.h에 정의해주었다. 또 각각에 대한 indirect block을 direct block들과 구분하기 위해 dinode와 inode 구조체의 addrs 배열 크기를 NDIRECT + 3으로 수정했다. 기존 NDIRECT 값인 12는 10으로 줄여줘야 했다. 이유는, 하나의 dinode 구조체의 크기가 BSIZE (Block size)의 약수가 되어야 하기 때문이다. 그래야 하는 정확한 이유는 모르겠지만, 추측을 해보자면, 한 block 내에서 dinode들을 관리할 때 dinode의 크기를 활용한 indexing을 사용하기 위함이 아닐까 싶다. 따라서 기존 direct block 두개를 각각 double, triple indirect block으로 쓰게끔 수정하였다.

bmap 함수에 추가한 double과 triple indirect 주소 변환 코드는 기존 bmap의 코드와 로직을 똑같이 여러번 반복하는 것일 뿐이기에 코드가 지저분하며 길어 설명은 생략하도록 하겠다. 유사하게, inode가 사용하던 모든 block들을 free해주는 함수인 itrunc도 multi indirect을 고려해 확장시켜주었지만, 마찬가지 이유로 설명을 생략한다.

2. Symbolic Link

바로가기 inode는 원본 inode의 정보를 저장하고, sys_open에서 열고자 하는 inode가 바로가기인 경우 원본 inode로 redirect되어야 한다. 따라서 바로가기 inode가 원본 inode의 경로를 저장하도록 구현하였다. 이 경로는 바로가기 inode의 block에 직접 write을 통해 저장한다. 즉 "hello"라는 파일에 "ho"라는 바로가기를 만들면, "ho"엔 단순히 "hello"라는 정보가 저장된다. 실제로 redirect을 하기 위해서 sys_open에서 바로가기인지 아닌지를 구분할 수 있어야 한다. 따라서 T_SYM이라는 inode type 상수를 새로 stat.h에 정의해주었다.

```
int
sys_symlink(void)
{
  char *old, *new;
  struct inode *ip;
  int pathlen;

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

  begin_op();

  if (namei(old) == 0) // 원본 파일이 없는 경우.
  {
    end_op();
    return -1;
  }

  if ((ip = create(new, T_SYM, 0, 0)) == 0)
  {
    end_op();
    return -1;
  }

  pathlen = strlen(old) + 1; // null 문자까지

  if (writei(ip, old, 0, pathlen) != pathlen)
    panic("symlink: writei");

  iupdate(ip);
  iunlock(ip);
  end_op();

  return 0;
}
```

sys_open에서는 ip가 T_SYM인 경우, readi를 통해 원본 경로를 읽어온다. 읽은 경로를 다시 namei 함수로 찾아 원본의 inode를 열도록 한다.

하지만 ls에서 각 파일의 정보를 출력할 때는 원본 파일의 정보가 출력되면 안된다. 바로가기를 만들고 원본을 지웠을때 ls을 실행해보면 stat 오류가 출력되는 문제가 있었다. 이유는, ls가 호출하는 stat 함수도 open 함수를 호출하기 때문이었다. 따라서 stat에서 호출하는 open은 redirection을 하지 않도록 조치가 필요했다.

이를 위해 기존 omode, 오픈모드 시스템을 활용하였다. O_STAT이라는 omode를 새로 fcntl.h에 정의해주었다. 따라서 stat함수는 open 함수의 인자로 이 O_STAT을 넘기며, sys_open에서는 O_STAT의 경우 redirection을 수행하지 않도록 해결하였다.

```
if (ip->type == T_SYM && omode != O_STAT)
{
  if (readi(ip, sympath, 0, ip->size) != ip->size)
    panic("symlink open: readi");

  iunlock(ip);
  if ((ip = namei(sympath)) == 0)
  {
    end_op();
    return -1;
  }
  ilock(ip);
}
```

3. Sync

기존 group commit 방식은 log_write 함수를 통해 commit할 block들의 정보를 log header에 저장해둔다. 어떤 operation이 수행을 마쳐 end_op 함수를 호출할 때, 다른 수행 중인 operation이 없다면 commit을 하는 방식이다. log header의 공간이 부족할 우려가 있으면 operation은 begin_op에서 sleep을 호출한다. 현재 수행 중인 operation들이 전부 끝나 group commit이 발생해 log header의 공간이 비워질 때까지 기다리는 것이다. 따라서 log header 공간이 부족해져도 commit을 통해 다시 공간이 확보된다는 것이 보장된다.

하지만 이번 프로젝트에서는 commit의 시점과 log header의 공간이 무관하다. Log header의 공간이 비워지는 상황은 sync가 시스템 콜로 호출되거나 buffer 공간이 없는 경우 강제로 호출되는 경우 뿐이다. 그렇다면 문제는 buffer 공간은 충분한데 log header의 공간이 부족한 경우다.

Sync는 결국 구현하지 못했다. 한 때 어느 정도 작동하는 수준까지는 구현을 했었으나, 이를 위해 기존 buffer io 로직을 많이 수정했기 때문에 sync의 버그가 전체 시스템에 극악한 문제를 일으키곤 했다. 따라서 최종 제출본에는 어느 정도 구현했던 sync의 로직을 전부 빼기로 했다. 그래도 이 섹션에서 나의 원래 계획에 대해 간략하게나마 코드와 함께 설명하도록 하겠다. Sync를 완전히 포기한 시점보다 오래된 코드이기에, 포기한 시점 당시에는 해결했던 자잘한 실수들도 섞여있다. 하지만 전반적인 흐름을 설명하기에는 충분할 듯 하다.

```
void
begin_op(void)
{
  acquire(&log.lock);

  while(1)
  {
    if(log.committing)
    {
      sleep(&log, &log.lock);
    }
    else
    {
      log.outstanding += 1;
      release(&log.lock);
      break;
    }
  }
}
```

begin_op에서는 기존과 달리 log header의 공간을 고려하지 않는다.

```
void
log_write(struct buf *b)
{

  b->flags |= B_DIRTY;
}
```

Operation 수행 중에 호출되는 log_write에서는 logheader의 리스트에 buf의 정보를 추가하지 말고, 단순히 buf의 dirty bit만 설정해주기 때문이다.

```
int
sync(void)
{
  struct buf **bufs;
  struct buf *b;
  int flushed = 0;

  acquire(&log.lock);

  log.committing = 1;

  bufs = get_dirty_bufs();

  ...
```

```
struct buf**
get_dirty_bufs(void)
{
  static struct buf *res[NBUF];
  struct buf *b;
  int i = 0;

  acquire(&bcache.lock);

  for (b = bcache.head.prev; b != &bcache.head; b = b->prev)
  {
    if (b->flags & B_DIRTY)
    {
      res[i++] = b;
    }
  }
  res[i] = 0;

  release(&bcache.lock);

  return res;
}
```

sync가 호출되면, bcache를 순회해 buffer의 모든 dirty buffer의 정보를 가져온다.

```
int
sync(void)
{
  ...

  bufs = get_dirty_bufs();

  for (b = *bufs; b; b++, flushed++)
  {
    logheader_append(b);

    if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
    {
      release(&log.lock);
      commit();
      acquire(&log.lock);
    }
  }

  ...
```

```
void
logheader_append(struct buf *b)
{
  int i;

  if (!holding(&log.lock))
    panic("lh_append: lock");

  if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
    panic("too big a transaction");

  for (i = 0; i < log.lh.n; i++)
  {
    if (log.lh.block[i] == b->blockno)
      break;
  }

  log.lh.block[i] = b->blockno;

  if (i == log.lh.n)
    log.lh.n++;

}
```

이 buffer들을 이제 실제로 차례차례 logheader의 리스트에 추가한다.

하지만 dirty buffer의 수가 log header의 크기보다 클 수 있으므로, 리스트에 추가하면서 log header가 가득 차면 commit을 호출한다.

```
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;

  acquire(&bcache.lock);

  ...

  for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
    if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
      b->dev = dev;
      b->blockno = blockno;
      b->flags = 0;
      b->refcnt = 1;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }

  release(&bcache.lock);

  if (sync() == 0)
    panic("bget: no buffers");

  return bget(dev, blockno);
}
```

Buffer의 공간이 없는 경우 강제로 sync가 호출되어야 한다. 따라서 bget 함수에서 buffer를 할당할 공간을 찾지 못하는 경우 sync를 호출한다.

계획은 이러했고, 실제로 어떤 파일을 만든 뒤 sync를 호출하지 않은채 xv6를 재부팅하면 그 파일이 없는 것까지 확인을 했다. 하지만 stressfs나 큰 파일을 만드는 코드를 실행하면 xv6가 스스로 재부팅을 하기도 하고, 페이지 폴트도 발생하고 등 다양한 문제가 발생해 결국 포기했다.

컴파일 및 실행

project03/xv6-public 디렉토리에서 다음 두 줄을 입력해 컴파일 및 실행을 할 수 있다.

1. ./makemake.sh
2. ./bootxv6.sh

실행 결과 – Multi Indirect

실행 인자로 n을 받아 block n개의 크기의 파일을 생성해주는 테스트 프로그램을 작성해보았다. 이를 사용해 bmap 함수가 Triple Indirect까지 정상적으로 처리할 수 있는지 32,768개의 block을 할당받는 파일을 만들어보았고, rm으로 itrunc의 작동 또한 문제 없어보인다.

```
$ makehugefile 32768
makehugefile success
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15484
echo           2 4 14364
forktest       2 5 8808
grep           2 6 18320
init           2 7 14984
kill           2 8 14452
ln             2 9 14816
ls             2 10 16916
mkdir          2 11 14472
rm             2 12 14452
sh             2 13 28500
stressfs       2 14 15416
wc             2 15 15900
zombie         2 16 14024
jason          2 17 15536
makehugefile   2 18 15248
vi             2 19 14840
console        3 20 0
hugefile       2 21 16777216
```

```
hugefile       2 21 16777216
$ rm hugefile
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15484
echo           2 4 14364
forktest       2 5 8808
grep           2 6 18320
init           2 7 14984
kill           2 8 14452
ln             2 9 14816
ls             2 10 16916
mkdir          2 11 14472
rm             2 12 14452
sh             2 13 28500
stressfs       2 14 15416
wc             2 15 15900
zombie         2 16 14024
jason          2 17 15536
makehugefile   2 18 15248
vi             2 19 14840
console        3 20 0
```

실행 결과 – Symbolic Link

vi라는 간단한 파일 생성 프로그램을 만들어 이를 통해 hello라는 파일을 만들어 보았다.

ln –s hello ho를 입력해 hello의 바로가기 파일 ho를 만들어 보았다.

ls에서 출력되는 ho의 정보는 hello의 것과 다르지만, cat을 통해 열어보면 같은 내용을 갖는다.

hello을 지워도 ls에 ho의 정보는 남아있지만, 내용은 접근할 수 없다.