OS assignment3

Design

- Multi Indirect

MAXFILE의 크기를 증가시키고 addrs의 블록을 증가시켜 DINDIRECT 와 TIDIRECT를 추가해준뒤

iappend와 bmap에서 Triple indirect 까지 가능하도록 수정한다

- Symbolic link

심볼릭 링크 system call이 호출되면 link의 대상을 addr의 첫번째 주소에 넣어준다

그 후 파일을 읽거나 쓸 때 target주소에 해당하는 inode를 가져와 수행한다

즉 바로가기 개념을 쓴것

- Sync

기존 begin_op에서 수정하던 commit 을 sync() 에서 수행한다

어짜피 디스크에 적기전에 log를 적어도 되기 때문에

기존에 log_write를 호출하던 곳을 다 없애고 sync()가 호출된 순간에만 log를 적는다

버퍼의 크기를 체크하는 부분을 버퍼를 할당받는 bget에서 수행하고 만약 꽉찼다면 sync()를 호출해준다

implement

1. Multi Indirect

```
param.h

#define FSSIZE       2000000  // size of file system in blocks
```

512* 10 + 512*128 + 512 * 128^2 + 512 * 128^3 해준거를 512로 나누면 2113674이나온다

10%를 메타데이터라고 잡고 넉넉하게 2000000으로 잡아주었다.

```
fs.h

...

#define NDIRECT 10
#define NINDIRECT (BSIZE / sizeof(uint))
#define DINDIRECT 128 * 128 //double indirect
#define TINDIRECT 128 * 128 * 128 //triple indirect
#define MAXFILE (NDIRECT + NINDIRECT + DINDIRECT + TINDIRECT) //MAXFILE 크기증가

// 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
};

...
```

fs.h 에서 DINDIRECT 랑 TINDIRECT 추가하고 NIDIRECT는 2개 줄여줌

MAXFILE 바꿔줌

dinode 구조체에서 data block addresses 에서 블록을 2개 추가해줌

```
file.h

...
```

```
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]; //addrs 증가
};

...
```

inode 구조체도 동일하게 data block addresses 에서 블록을 2개 추가해줌

```
mkfs.c

void
iappend(uint inum, void *xp, int n)
{
...

// double indircet block을 추가하는 부분
  } else if(fbn < (NDIRECT + NINDIRECT + DINDIRECT)){
    if(xint(din.addrs[NDIRECT+1]) == 0){
      //double indirect block이 할당되지 않았으면 새로운 block할당
      din.addrs[NDIRECT+1] = xint(freeblock++);
    }

    uint index1 = (fbn - NDIRECT - NINDIRECT) / NINDIRECT;
    rsect(xint(double_indirect[index1]), (char*)indirect);

    if(indirect[index1] == 0){
      //double indirect block 내부의 indirect block이 아직 할당되지 않았으면
//새로운 block할당
      indirect[index1] = xint(freeblock++);
      wsect(xint(double_indirect[index1]), (char*)indirect);
    }

    x = xint(indirect[(fbn - NDIRECT - NINDIRECT) % NINDIRECT]);

  // triple indirect block을 추가하는 부분
   } else {
    if(xint(din.addrs[NDIRECT+2]) == 0){
      //triple indirect block이 아직 할당되지 않았으면 새로운 block할당
```

```
      din.addrs[NDIRECT+2] = xint(freeblock++);
    }

    uint index1 = (fbn - NDIRECT - NINDIRECT - DINDIRECT) / (NINDIRECT * NINDIRECT);
    rsect(xint(din.addrs[NDIRECT+2]), (char*)triple_indirect);

    if(triple_indirect[index1] == 0){
      //triple indirect block 내부의 double indirect block이 아직 할당되지 않았으면
// 새로운 block할당
      triple_indirect[index1] = xint(freeblock++);
      wsect(xint(din.addrs[NDIRECT+2]), (char*)triple_indirect);
    }

    rsect(xint(triple_indirect[index1]), (char*)double_indirect);
    uint index2 = ((fbn - NDIRECT - NINDIRECT - DINDIRECT) / NINDIRECT) % NINDIRECT;
    if(double_indirect[index2] == 0){
      //triple indirect block 내부의 indirect block이 아직 할당되지 않았으면
//새로운 block할당
      double_indirect[index2] = xint(freeblock++);
      wsect(xint(triple_indirect[index1]), (char*)double_indirect);
    }

    rsect(xint(double_indirect[index2]), (char*)indirect);
    uint index3 = (fbn - NDIRECT - NINDIRECT - DINDIRECT) % NINDIRECT;
    if(indirect[index3] == 0){
      //triple indirect block 내부의 indirect block이 아직 할당되지 않았으면
//새로운 block할당
      indirect[index3] = xint(freeblock++);
      wsect(xint(double_indirect[index2]), (char*)indirect);
    }

    x = xint(indirect[index3]);
  }

...

}
```

기존 INDIRECT까지 블록을 할당하던것을 TINDIRECT까지 가능하도록 변경

```
fs.c

static uint
bmap(struct inode *ip, uint bn)
{

...

//double indirect block load, 필요하다면 새로 할당
```

```
  if(bn < DINDIRECT){
    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);
      bp->flags|=B_DIRTY;
    }
    brelse(bp);
    bp2 = bread(ip->dev, addr);
    a = (uint*)bp2->data;
    if((addr = a[bn % NINDIRECT]) == 0){
      a[bn % NINDIRECT] = addr = balloc(ip->dev);
      // log_write(bp2);
      bp2->flags|=B_DIRTY;
    }
    brelse(bp2);
    return addr;
  }

  bn -= DINDIRECT;
  //triple indirect block load, 필요하다면 새로 할당
  if(bn < TINDIRECT){
  if((addr = ip->addrs[NDIRECT + 2]) == 0)
    ip->addrs[NDIRECT + 2] = addr = balloc(ip->dev);
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  uint firstIndex = bn / (NINDIRECT * NINDIRECT);
  if((addr = a[firstIndex]) == 0){
    a[firstIndex] = addr = balloc(ip->dev);
    // log_write(bp);
    bp->flags|=B_DIRTY;
  }
  brelse(bp);

  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  uint secondIndex = (bn / NINDIRECT) % NINDIRECT;
  if((addr = a[secondIndex]) == 0){
    a[secondIndex] = addr = balloc(ip->dev);
    // log_write(bp);
    bp->flags|=B_DIRTY;
  }
  brelse(bp);

  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  uint thirdIndex = bn % NINDIRECT;
  if((addr = a[thirdIndex]) == 0){
    a[thirdIndex] = addr = balloc(ip->dev);
    // log_write(bp);
    bp->flags|=B_DIRTY;
```

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

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

기존 INDIRECT까지 블록매핑하던것을 TINDIRECT까지 가능하도록 변경

2. Symbolic Link

```
stat.h

#define T_SYMLINK 4  // Symbolic link
```

심볼릭 링크 타입을 위해 추가

```
sysfile.c

int
sys_symlink(void)
{
  char *link, *target;
  struct inode *dp, *ip;

  // 문자열 형태의 target과 link에 대한 주소를 가져옴
  if(argstr(0, &target) < 0 || argstr(1, &link) < 0)
    return -1;

  begin_op();
  // target의 inode를 가져온다.
  if((ip = namei(target)) == 0){
    end_op();
    return -1;
  }

  ilock(ip);
  // 만약 target이 디렉토리라면 에러를 반환한다.
  if(ip->type == T_DIR){
    iunlockput(ip);
```

```
    end_op();
    return -1;
  }

  // inode의 상태를 디스크에 업데이트한다.
  iupdate(ip);
  iunlock(ip);

  // link 이름으로 새 inode를 생성하고, 이 inode는 T_SYMLINK 타입이다.
  if((dp = create(link, T_SYMLINK, 0, 0)) == 0)
    return -1;

  // 생성된 inode의 addrs 부분에 target의 경로를 복사한다.
  memmove((char*)(dp->addrs), target, strlen(target));

  // inode의 상태를 디스크에 업데이트한다.
  iupdate(dp);
  iunlock(dp);

  end_op();

  return 0;
}
```

`namei` 함수를 사용하여 `target`의 inode를 가져온다. `ip->type == T_DIR`를 통해 `target`이 디렉토리인지 확인한다. 만약 디렉토리라면, 이 함수는 -1을 반환하고 종료합니다. `iupdate(ip)`를 통해 inode의 상태를 디스크에 업데이트. `create` 함수를 통해 `link` 이름으로 새 inode를 생성하고, 이 inode는 `T_SYMLINK` 타입이다 `memmove`를 통해 생성된 inode의 `addrs` 부분에 `target`의 경로를 복사하고. `iupdate(dp)`를 통해 심볼릭 링크 inode의 상태를 디스크에 업데이트한다.

이를통해 실제 경로를 알 필요 없이 대상 파일이나 디렉토리에 접근할 수 있게 됨

```
fs.c

int
readi(struct inode *ip, char *dst, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  // 만약 inode가 symbolic link 타입이라면,
  // addrs에 저장된 이름으로 실제 inode를 찾는다.
  if(ip->type == T_SYMLINK){
```

```
    if((ip = namei((char*)(ip->addrs)))==0) return -1;
  }

...

}
```

파일을 읽을 때 심볼릭 링크라면 addrs에 저장된 이름으로 실제 inode를 찾는다

```
fs.c

int
writei(struct inode *ip, char *src, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

   // 만약 inode가 symbolic link 타입이라면,
  // addrs에 저장된 이름으로 실제 inode를 찾는다.
  if(ip->type == T_SYMLINK){
    if((ip = namei((char*)(ip->addrs)))==0) return -1;
  }

 ...
}
```

readi와 동일하게 파일을 쓸 때 심볼릭 링크라면 addrs에 저장된 이름으로 실제 inode를 찾는다

3. Sync

```
param.h

#define LOGSIZE      (MAXOPBLOCKS*10)  // max data blocks in on-disk log
#define NBUF         (MAXOPBLOCKS*10)  // size of disk block cache
```

LOGSIZE와 NBUF 사이즈를 늘려주었다

```
bio.c

struct {
  struct spinlock lock;
  struct buf buf[NBUF];
  int isfull; //beg 함수의 재귀적호출을 막기 위한 flag

  // Linked list of all buffers, through prev/next.
  // head.next is most recently used.
  struct buf head;
} bcache;
```

beg 함수의 재귀적호출을 막기 위한 flag로 isfull 추가

```
log.c

// called at the start of each FS system call.
void
begin_op(void)
{
  acquire(&log.lock);
  // 로그가 커밋중이지 않을때만 진행
  while(1){
    if(log.committing){
      sleep(&log, &log.lock);
    } else {
      release(&log.lock);
      break;
    }
  }
}
```

로그가 현재 커밋 중이지 않은 경우에만 연산을 계속 진행할 수 있도록 한다. 만약 로그가 커밋 중이라면, 이 함수는 커밋이 완료될 때까지 대기

```
log.c

// called at the end of each FS system call.
```

```
void
end_op(void)
{
  //파일시스템 연산이 끝나면 대기중인 연산 진행
  acquire(&log.lock);
  wakeup(&log);
  release(&log.lock);
}
```

기존의 커밋 부분을 삭제하고 그냥 다른 대기 중인 연산이 계속 진행할 수 있도록 `wakeup()`을 호출해 해당 로그를 깨운다

```
log.c


int
sync(void)
{
  int temp = 0;

  acquire(&log.lock);

  if (log.committing) { // 이미 commit이 진행중이면
    release(&log.lock);
    return -1; // 실패로 간주
  }

  log.committing = 1;
  release(&log.lock);

  temp = logDirtyBuffer();

  // Dirty buffer가 없으면 이미 sync되어 있다는 것이므로 성공으로 간주
  if (temp == 0) {
    acquire(&log.lock);
    log.committing = 0;
    release(&log.lock);
    return 0;
  }

  write_log();
  write_head();
  install_trans();

  temp = log.lh.n;
  log.lh.n = 0;
```

```
  write_head();

  acquire(&log.lock);
  log.committing = 0;
  release(&log.lock);

  return temp; // 동기화한 블록 수를 반환
}
```

실패한다는 것을 커밋중일때 호출된것이라고 하였다

기존에 log_write를 하는 모든 부분은

```
// log_write(bp);
  bp->flags|=B_DIRTY;
```

이런식으로 flag처리를 해주고 로그에 적지 않았다

그리고 sync가 호출될때 ogDirtyBuffer를 통해 dirty buffer를 찾아 log에 기록한다

만약 값이 0이면 dirty buffer가 없는 것이므로 0반환

아니면 commit 동작 수행하고

동기화한 블록 수를 반환한다

```
bio.c

// dirty버퍼들을 찾아 로그에 기록하고 개수 반환
int
logDirtyBuffer(void)
{
  struct buf *b;

  int cnt = 0;
  acquire(&bcache.lock);
  // 버퍼가 dirty 상태면 로그에 기록
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if((b->flags & B_DIRTY)){
      log_write(b);
      cnt++;
    }
  }
  release(&bcache.lock);
```

```
  return cnt; //dirtybuffer 개수 반환
}
```

버퍼 캐시를 돌면서 dirtybuffer를 찾아 log_write를 호출해 log를 적는다

log를 적은 dirtybuffer 개수를 반환한다

```
bio.c

//버퍼 캐시가 가득 찼는지 확인
int
buffer_isfull(void)
{
  struct buf* b = 0;
  int cnt = 0;
  acquire(&bcache.lock);
  //버퍼가 dirty 상태면 cnt증가
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if((b->flags & B_DIRTY)){
      cnt++;
    }
  }
  release(&bcache.lock);
  if(cnt >= NBUF-3) return 1; //dirty 버퍼가 가득차있으면 1반환
  else return 0;
}
```

버퍼 캐시를 돌며 dirtybuffer의 개수를 센다

만약 dirtybuffer의 개수가 NBUF-3이상이라면 꽉찬것으로 간주하고 1을 return한다

여기서 NBUF-3을 꽉찬것으로 간주한 이유는 `write_log` 함수가 한 번에 최대 3개의 버퍼를 사용하기 때문이다. `write_log` 함수는 로그 블록(`to`)과 캐시 블록(`from`)에 각각 하나씩의 버퍼를 사용하고, 또한 `bwrite` 함수 내부에서 버퍼 한 개를 사용할 수 있다. 따라서 `write_log` 함수가 동시에 실행되는 경우를 대비해 `buffer_isfull` 함수에서는 최대 버퍼 사용량보다 3개 적게 버퍼가 사용되었는지 확인하였다

```
bio.c

static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;
  //버퍼가 가득찼다면 sync호출 beget의 재귀적 호출을 막기 위해 isfull flag를 둠
  if(bcache.isfull == 0 && buffer_isfull())
  {
    acquire(&bcache.lock);
    bcache.isfull = 1;
    release(&bcache.lock);

    sync();

    acquire(&bcache.lock);
    bcache.isfull = 0;
    release(&bcache.lock);
  }

...
}
```

버퍼 캐시가 가득 찼을 때 (`buffer_isfull` 함수의 반환 값이 참일 때) sync를 호출하도록 `bget` 함수를 수정하였다. 이를 통해 버퍼 캐시가 가득 찼을 때 버퍼의 내용을 디스크에 동기화한다.

다만, 이 과정에서 `bget` 함수 자체가 sync를 호출하므로, sync 과정 중에 다시 `bget` 함수가 호출되면서 재귀적으로 sync가 호출되는 상황을 막기 위해 `isfull` 플래그를 사용한다. 버퍼 캐시가 가득 찼을 때 처음 sync를 호출하기 전에 `isfull` 플래그를 1로 설정하고, sync 호출이 완료된 후에 `isfull` 플래그를 다시 0으로 설정한다. 이를 통해 `bget` 함수가 sync를 호출하는 동안에는 추가적인 sync 호출을 방지하고, sync 호출이 완료된 후에는 다시 sync 호출이 가능하도록 한다.

---

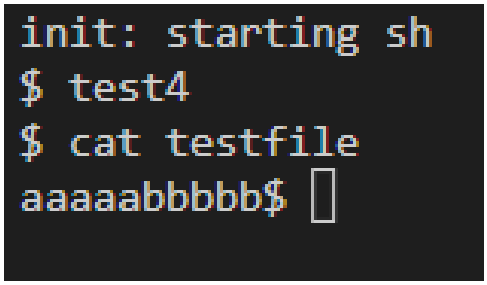
result

```
$ test1
Successfully written 6MB to file: 6mbfile
Successfully read 6MB from file: 6mbfile
Successfully written 16MB to file: 16mbfile
Successfully read 16MB from file: 16mbfile
$
```

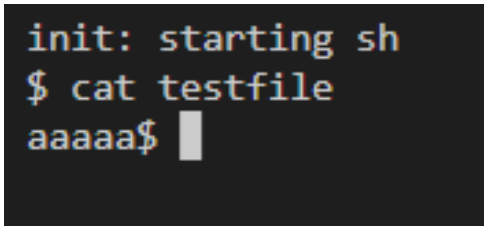
test1을 실행하여 6mb파일 쓰고읽기, 16mb파일 쓰고읽기 테스트 결과

```
$ ln -s ls symlink
$ symlink
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16324
echo           2 4 15176
forktest       2 5 9480
grep           2 6 18540
init           2 7 15760
kill           2 8 15204
ln             2 9 15332
ls             2 10 17688
mkdir          2 11 15300
rm             2 12 15280
sh             2 13 27920
stressfs       2 14 16192
usertests      2 15 67296
wc             2 16 17056
zombie         2 17 14868
test1          2 18 17420
test2          2 19 18760
test3          2 20 17696
test4          2 21 15652
console        3 22 0
testfile       2 23 5
triple_indirec 2 24 8525824
symlink        4 25 0
$
```

ls 와 symbolic link로 연결 후 실행



test4를 실행하여 testfile을 생성하고 ‘a’를 다섯번 쓴다음 sync() 호출하고 ‘b’를 다섯 번 작성



xv6를 재실행 후 testfile을 출력해보면 디스크에 적히지 않은 ‘b’ 5개가 출력되지 않는 것을 볼 수 있다

---

trouble shooting

- addrs 의 사이즈를 2를 늘려주니까 Bsize와 dinode가 나누어떨어지지 않아서 오류가 발생하였다

```
mkfs: mkfs.c:84: main: Assertion `(BSIZE % sizeof(struct dinode)) == 0' failed.
make: *** [Makefile:187: fs.img] 중지됨 (메모리 덤프됨)
```

→ NDirect 사이즈를 2줄여줌으로써 해결하였다

- 처음에 버퍼의 크기를 확인하고 가득차있을 경우 `sync()`를 호출하는 과정을 `begin_op()`에서 진행하였다

그랬더니 log_write를 해주는과정에서 “too big a transaction” panic이 발생하였다

고민해보니 begin_op()와 end_op() 사이에서 버퍼 할당을 한번만 받지 않아서 발생한다고 생각하였고

→그래서 버퍼 할당을 받는 `bget()`에서 버퍼의 크기를 확인하고 `sync()`를 호출해주어서 해결하였다