Operating System Project 3 Wiki

2019092306 구본준

Design

1. Multi Indirect

기존에 존재하던 single indirect를 기반으로 double indirect와 triple indirect를 구현할 것이다. 가장 먼저 *dinode*와 *inode* 구조체를 변경하여 multi indirect를 위한 공간을 만들어 주어야 한다. 다음으로 블록을 할당해주는 *bmap()* 함수를 수정해야 하는데 만약 direct와 single indirect 공간을 모두 사용한 경우 double indirect, triple indirect 공간을 사용하도록 코드를 추가해준다. 마지막으로 이를 해제해주는 *itrunc()* 함수를 변경하여 multi indirect를 통해 할당한 블록을 찾아 모두 해제할 수 있도록 구현할 것이다.

2. Symbolic Link

symbolic link를 지원해주는 system call을 기존에 있던 *link()* system call을 기반으로 내용을 변경하여 만들 것이다. *link()* system call의 경우 부모 디렉토리를 확인하여 만들고 싶은 link 파일의 이름을 찾아 이를 link 하고 싶은 대상의 inode로 연결해주는 방식이다. Symbolic link의 경우 inode를 공유하는 것이 아닌 자신만의 inode를 가지며 open system call 호출 시 symbolic link 파일이 가리키는 파일을 열어주도록 구현할 것이다. 이를 위해 inode를 새롭게 할당하여 원본 대상 파일의 path를 symbolic link 파일의 inode에 write 한 후 open system call 호출 시 이를 읽어 다시 open 해주도록 만들 것이다.

2. Sync

xv6에서는 파일 관리를 트랜잭션처럼 처리하는데 이는 *begin_op(), end_op()* 함수를 통해 이루어진다. 이 때 *end_op()* 함수에서는 파일을 접근하고 있는 프로세스가 없는 경우 바로 *commit()* 함수를 통해 write를 진행하는데 이 부분을 들어내어 *sync()*라는 별도의 system call로 구현할 것이다. 추가적으로 버퍼에 더 이상 공간이 남아있지 않는 경우 버퍼를 비워주어야 하므로 *begin_op()* 함수 내에서 이를 확인하여 적절히 *sync()*를 호출하도록 만들 것이다.

Implement

1. dinode

```
#define NDIRECT 10
#define NINDIRECT (BSIZE / sizeof(uint))
#define DINDIRECT (128 * 128)
#define TINDIRECT (128 * 128 * 128)
#define MAXFILE (NDIRECT + NINDIRECT + DINDIRECT + TINDIRECT)

// 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
};
```

dinode 구조체이다. 구조체 자체에서 변한 것은 없으며 NDIRECT를 2 줄여 각각 double indirect와 triple indirect를 위한 공간으로 만들어주었다. 추가적으로 이와 같이 *file.h*에 있는 *inode* 구조체도 변경해주었다. 한 블록은 128개의 블록 포인터를 저장할 수 있으므로 DINDIRECT는 128^2로, TINDIRECT는 128^3으로 크기를 설정해주었다.

2. bmap()

```
// Double indirect block
  if(bn < DINDIRECT){
    // Load first indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT+1]) == 0)
      ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);

    // Load second indirect block, allocating if necessary.
    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);

    // Load direct block, allocating if necessary.
    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 -= DINDIRECT;

// Triple indirect block.
if(bn < TINDIRECT){
  // Load first indirect block, allocating if necessary.
  if((addr = ip->addrs[NDIRECT+2]) == 0)
    ip->addrs[NDIRECT+2] = addr = balloc(ip->dev);

  // Load second indirect block, allocating if necessary.
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  if((addr = a[bn / DINDIRECT]) == 0){
    a[bn / DINDIRECT] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);

  // Load third indirect block, allocating if necessary.
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  if((addr = a[(bn % DINDIRECT) / NINDIRECT]) == 0){
    a[(bn % DINDIRECT) / NINDIRECT] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);

  // Load direct block, allocating if necessary.
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  if((addr = a[(bn % DINDIRECT) % NINDIRECT]) == 0){
    a[(bn % DINDIRECT) % NINDIRECT] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);

  return addr;
}
```

bmap() 함수 내부에 추가한 코드이다. 기존에 있던 single indirect 코드를 활용하여 double indirect는 한 단계, triple indirect는 두 단계의 indirect 블록을 추가적으로 찾아가도록 구현하였다. Indexing의 경우 바깥쪽 블록 index는 현재 index를 안쪽 블록

크기로 나눈 몫으로, 안쪽 블록 index는 현재 index를 안쪽 블록 크기로 나눈 나머지로 설정하여 순서대로 블록을 할당할 수 있도록 하였다.

3. itrunc()

```
// Truncate double indirect block.
  if(ip->addrs[NDIRECT + 1]){
    bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
    a = (uint*)bp->data;
    for(i = 0; i < NINDIRECT; i++){
      if(a[i]) {
        bp2 = bread(ip->dev, a[i]);
        b = (uint*)bp2->data;
        for(j = 0; j < NINDIRECT; j++){
          if(b[j])
            bfree(ip->dev, b[j]);
        }
        brelse(bp2);
        bfree(ip->dev, a[i]);
      }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT + 1]);
    ip->addrs[NDIRECT + 1] = 0;
  }

  // Truncate triple indirect block.
  if(ip->addrs[NDIRECT + 2]){
    bp = bread(ip->dev, ip->addrs[NDIRECT + 2]);
    a = (uint*)bp->data;
    for(i = 0; i < NINDIRECT; i++){
      if(a[i]) {
        bp2 = bread(ip->dev, a[i]);
        b = (uint*)bp2->data;
        for(j = 0; j < NINDIRECT; j++){
          if(b[j]) {
            bp3 = bread(ip->dev, b[j]);
            c = (uint*)bp3->data;
            for(k = 0; k < NINDIRECT; k++){
              if(c[k])
                bfree(ip->dev, c[k]);
            }
            brelse(bp3);
            bfree(ip->dev, b[j]);
          }
        }
        brelse(bp2);
```

```
      bfree(ip->dev, a[i]);
    }
  }
  brelse(bp);
  bfree(ip->dev, ip->addrs[NDIRECT + 2]);
  ip->addrs[NDIRECT + 2] = 0;
}
```

itruct() 함수 내부에 추가한 코드이다. Single indirect를 truncate하는 코드를 기반으로 double indirect는 한 단계, triple indirect는 두 단계를 추가적으로 들어가 truncate하도록 구현하였다.

4. sys_slink()

```
// Allocate inode for the new file.
  nip = ialloc(dp->dev, T_SYM);

  // Link the inode to the new file name.
  if(dp->dev != ip->dev || dirlink(dp, name, nip->inum) < 0){
    iunlockput(dp);
    goto bad;
  }
  iunlockput(dp);
  iput(ip);

  // Write the old file path in the new file.
  ilock(nip);
  writei(nip, old, 0, strlen(old));
  nip->nlink = 1;
  iupdate(nip);
  iunlockput(nip);
```

symbolic link를 위한 system call인 *sys_slink()* 함수의 일부이다. 나머지 부분은 *sys_link()* 함수와 유사하며 다르게 구현한 부분은 inode를 새롭게 할당해주는 *ialloc()* 함수 호출 부분과 할당 받은 inode에 link 대상이 되는 파일의 path를 저장하는 *writei()* 함수 부분이 있다. inode의 type으로 symbolic link를 의미하는 T_SYM을 새로 만들었으며 4로 설정하였다. inode를 새로 할당 받았기 때문에 nlink를 1로 초기화해준다.

5. sys_open()

```
// Symbolic link: redirect to indicated file.
    if(ip->type == T_SYM){
      // Read the original file path.
      if((bytes = readi(ip, npath, 0, BSIZE - 1)) < 0){
        iunlockput(ip);
        end_op();
        return -1;
```

```
      }
      npath[bytes] = '\0';

      // Change inode to the original file's.
      iunlockput(ip);
      if((ip = namei(npath)) == 0){
        end_op();
        return -1;
      }
      ilock(ip);
    }
```

sys_open() 함수 내부에 추가한 코드이다. 만약 open한 파일의 type이 T_SYM이라면 symbolic link 파일이라는 의미이므로 symbolic link 파일을 읽어서 원본 파일의 path를 알아낸다. 이후 inode 변수의 값을 원본 파일의 것으로 바꿔준 후 나머지 open 과정을 진행한다.

6. openinfo()

```
int
stat(const char *n, struct stat *st)
{
  int fd;
  int r;

  fd = openinfo(n, O_RDONLY);
  if(fd < 0)
    return -1;
  r = fstat(fd, st);
  close(fd);
  return r;
}
```

ls 명령어와 같이 파일의 메타데이터를 확인해야 하는 경우 symbolic link 파일 자체를 open해야 하는 경우가 있을 수 있으므로 기존에 구현되어 있던 *sys_open()* 함수를 *sys_openinfo()*에 그대로 복사하여 만들었다. *stat()* 함수의 경우 사용자가 파일의 메타데이터를 알기 위해 호출하는 함수이므로 *open()* system call이 아닌 *openinfo()* system call을 호출하도록 바꿔주었다.

7. ln.c

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

```
  // Symbloic link.
  else if(!strcmp(argv[1], "-s")){
    if(slink(argv[2], argv[3]) < 0)
      printf(2, "link %s %s: failed\n", argv[1], argv[2]);
  }
```

ln.c 파일에 추가한 코드이다. Symbolic link도 지원할 수 있도록 "-h/-s" 인자를 받아 link의 종류를 구분한다.

8. sync()

```
int
sync(void)
{
  int ret = 0;

  // Check if the other process is commiting.
  acquire(&log.lock);
  while(log.committing)
    sleep(&log, &log.lock);
  log.committing = 1;
  release(&log.lock);

  // Commiting.
  if (log.lh.n > 0) {
    write_log();     // Write modified blocks from cache to log
    write_head();    // Write header to disk -- the real commit
    install_trans(); // Now install writes to home locations
    ret = log.lh.n;  // flushed block numbers.
    log.lh.n = 0;
    write_head();    // Erase the transaction from the log
  }

  // Commiting done.
  acquire(&log.lock);
  log.committing = 0;
  wakeup(&log);
  release(&log.lock);

  return ret;
}
```

새롭게 구현한 *sync()* 함수이다. System call로 만들었으며 기존에 *end_op()* 함수에 있던 commit 부분을 기반으로 구현하였다. 먼저 현재 다른 프로세스가 commit을 하고 있는지 확인하여 만약 그렇다면 sleep을, 아니라면 commit을 진행하게 된다. Flush된 블록의 개수를 return해주어야 하므로 *ret* 변수에 이를 저장하여 함수 종료 시 반환해준다.

9. begin_op()

```
else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
      release(&log.lock);
      sync();
      acquire(&log.lock);
}
```

begin_op() 함수 내부에 추가한 코드이다. 기존 코드의 경우 버퍼의 공간이 없을 때 단순히 sleep을 하여 공간이 생기기를 기다렸다. 하지만 이제 *sync()* 함수를 호출하지 않는 이상 버퍼가 비워지지 않으므로 sleep하지 않고 바로 *sync()*를 호출하도록 구현하였다. 이 때 *sync()* 함수 내부에서 자체적으로 log.lock을 잡기 때문에 호출 전에 release를 해주며 *sync()* 함수 종료 후 다시 log.lock을 acquire해준다.

Result

1. 컴파일 및 실행과정

```
make CPUS=1
make fs.img
```

컴파일 시 CPUS=1 인자를 통해 CPU 개수를 하나로 설정해주었다.

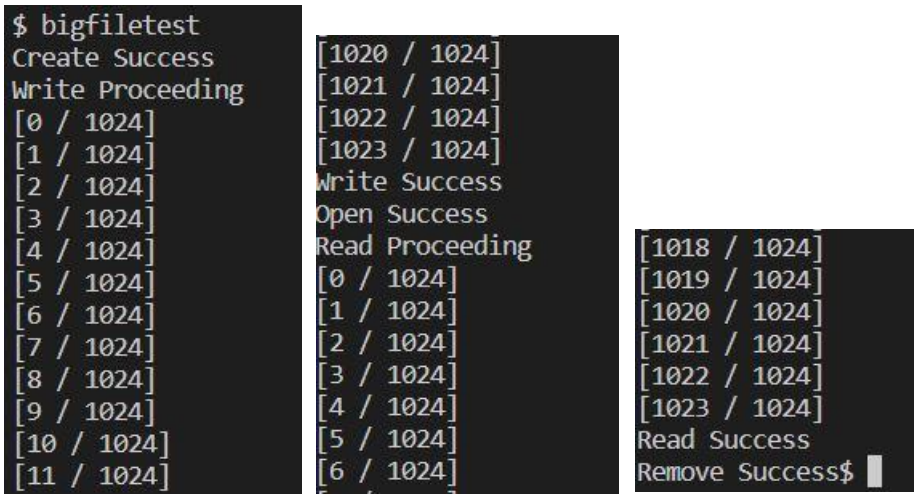
```
qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m
512
```

부팅을 위한 명령어이다.

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
```

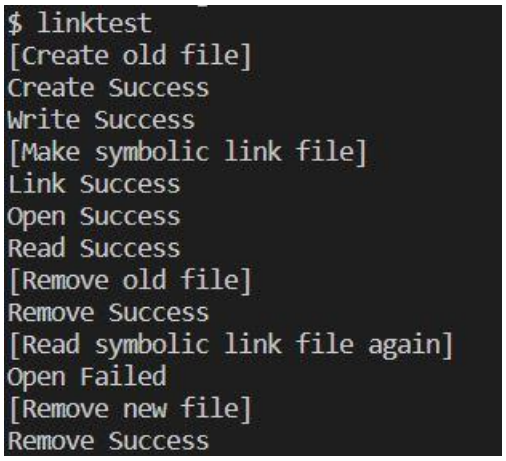
부팅이 정상적으로 완료된 모습이다. 이후 bigfiletest, linktest등의 테스트 파일 이름을 입력하면 정상적으로 테스트 코드가 실행된다.

2. Multi indirect test code



Multi indirection이 정상적으로 동작하는지 테스트하는 코드이다. 16MB의 파일을 생성하고 다시 읽은 후 삭제까지 진행하는 테스트 코드이다. Write와 read가 실행되는 중간 과정은 길어서 생략하였으며 정상적으로 크기가 큰 파일이 동작하는 것을 볼 수 있다.

3. Symbolic link test code



Symbolic link가 정상적으로 동작하는지 테스트하는 코드이다. Old 파일을 하나 생성한 후 이에 대한 symbolic link인 new 파일을 만들어준다. 이때 정상적으로 old 파일이 열리는 것을 확인할 수 있다. 이후 old 파일을 삭제하고 다시 new 파일을 open해 보면 이제는 열리지 않는 것을 확인할 수 있다.

3. Symbolic link ls

```
$ ln -s README link_file
```

```
link_file      4 23 6
```

Symbolic link 파일을 생성한 이후 ls 명령어를 입력하면 원본 파일의 메타데이터가 아닌 link 파일 자체의 메타데이터가 출력된다. 이 때 symbolic link 파일의 type은 T_SYM이기 때문에 4로 표시되는 것을 볼 수 있다.

4. Sync test code

```
$ syncwritetest
[Create a file with sync()]
Create Success
Write Success
File size: 131072
Sync: 85 blocks are written
[Create a file without sync()]
Create Success
Write Success
File size: 131072
```

```
$ syncreadtest
[Read a file written with sync()]
Open Success
File size: 131072
Read Success
[Read a file written without sync()]
Open Success
File size: 89088
Read Failed
Remove Success
Remove Success
```

sync() 함수가 정상적으로 동작하는지 테스트하는 코드이다. 먼저 두 개의 파일을 생성하는데 첫 번째 파일을 생성한 이후에는 *sync()*를 호출하며 두 번째 파일을 생성한 이후에는 바로 프로그램을 종료한다. Xv6를 재부팅 한 이후 이전에 만든 두 파일을 읽어보면 첫 번째 파일은 size가 그대로이지만 두 번째 파일은 그렇지 않다는 것을 볼 수 있다. 두 번째 파일의 사이즈가 0이 아닌 이유는 버퍼가 부족한 경우 자체적으로 *sync()* 함수를 호출하기 때문이다.

Trouble Shooting

1. define 우선순위 문제

```
#define NINDIRECT (BSIZE / sizeof(uint))
#define DINDIRECT (128 * 128)
#define TINDIRECT (128 * 128 * 128)
#define MAXFILE (NDIRECT + NINDIRECT + DINDIRECT + TINDIRECT)
```

처음에 define으로 설정한 수에 괄호를 넣지 않자 연산 우선순위 문제가 발생하였다. *bmap()* 함수에서는 DINDIRECT와 TINDIRECT를 연산에 사용하여 index를 알아내는데 이 때 괄호가 없어 '*'가 연산되기 전 다른 연산이 우선적으로 되어 잘못된 결과가 나왔다. 이를 발견 후 적절히 고쳐주었다.

2. symbolic link 생성시 nlink 변수 설정 문제

```
// Write the old file path in the new file.
  ilock(nip);
  writei(nip, old, 0, strlen(old));
  nip->nlink = 1;
  iupdate(nip);
  iunlockput(nip);
```

symbolic link는 기존의 hard link와 다르게 inode를 새로 만들어주어야 하는데 이 때 *nlink* 변수를 1로 초기화하지 않으니 *nlink*가 0으로 유지되어 제대로 동작하지 않는 문제가 발생하였다. 기존에 있던 static 함수인 *create()*를 참고하여 *nlink*를 적절히 설정해주어 문제를 해결하였다

3. sync 관련 lock 문제

```
else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
      release(&log.lock);
      sync();
      acquire(&log.lock);
```

sync() 함수 내부에서 자체적으로 lock을 잡는데 *begin_op()* 함수에서 lock을 미리 잡고 *sync()*를 호출하니 충돌하는 문제가 발생하였다. 이를 해결하기 위하여 호출 전에 미리 lock을 해제하도록 변경하여 문제를 해결하였다.