운영체제 Project 3 WIKI

2019008813 문원찬

1. Multi Indirect

a. Design

큰 파일을 다루기 위해 Double indirect와 Triple indirect를 구현하고자 한다. 기존의 xv6의 12개의 NDIRECT를 2개 줄여서 각각 Double indirect와 Triple indirect를 구현하기 위한 공간으로 만들 것이다. Double indirect는 이미 정의된 indirect의 indirect로 구현할 예정이며 Triple indiect는 indirect의 indirect의 indirect로, 계층적으로 구현한다.

b. Implement

가장 우선적으로 param.h의 FSSIZE를 충분히 큰 수인 100000으로 설정했다. 이후 Design에 맞게 fs.h를 다음과 같이 수정해주었다.

fs.h

```
#define NDIRECT 10
#define NINDIRECT (BSIZE / sizeof(uint))
#define NINDIRECT_2 (NINDIRECT*NINDIRECT)
#define NINDIRECT_3 (NINDIRECT_2*NINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT + NINDIRECT_2 + NINDIRECT_3)

// 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
};
```

NDIRECT를 12에서 10으로 줄이고 Double indirect와 Triple indirect 각각의 사이즈는 NINDIRECT*NINDIRECT와 NINDIRECT*NINDIRECT*NINDIRECT으로 정의했다. NDIRECT가 2개 줄었으므로 dinode 구조체의 addrs의 사이즈를 NDIRECT+1에서 NDIRECT+3으로 수정했다. file.h의 inode구조체에서도 addrs의 사이즈를 같은 방식으로 수정했다.

fs.c – bmap 함수

```
if(bn < NINDIRECT_2){ // addrs[NDIRECT+1] 사용
    // Load double indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT+1]) == 0)
      ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr); // 첫 계층
    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); // 두번째 계층
    a = (uint*)bp->data;
    if((addr = a[bn%NINDIRECT]) == 0){
      a[bn%NINDIRECT] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }
if(bn < NINDIRECT_3){ // addrs[NDIRECT+2] 사용
    // 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); // 첫번째 계층
    a = (uint*)bp->data;
    if((addr = a[bn/NINDIRECT_2]) == 0){
      a[bn/NINDIRECT_2] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    bp = bread(ip->dev, addr); // 두번째 계층
    a = (uint*)bp->data;
    if((addr = a[(bn%NINDIRECT_2)/NINDIRECT]) == 0){
      a[(bn%NINDIRECT_2)/NINDIRECT] = addr = balloc(ip->dev);
      log_write(bp);

    }
    brelse(bp);
    bp = bread(ip->dev, addr); // 세번째 계층
    a = (uint*)bp->data;
    if((addr = a[(bn%NINDIRECT_2)%NINDIRECT]) == 0){
      a[(bn%NINDIRECT_2)%NINDIRECT] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }
```

다음은 Double indirect와 Triple indirect가 잘 작동하기 위해 fs.c의 bmap 함수를 수정한 함수의 일부이다. 기존에 bmap 함수처럼 bn의 크기에 따라 Double indirect를 사용할지 Triple indirect를 사용할지 정했다. Double이면 addrs[NDIRECT+1]을 사용하고 bn을 NINDIRECT로 나눈 몫과 나머지를 이용해 block을 mapping했다. bn의 몫을 첫번째 계층의 index로, 나머지를 두번째 계층의 index로 사용해 겹치는 block이 없도록 구현했다. Triple의 경우엔 addrs[NDIRECT+2]를 사용하며 bn을 NINDIRECT_2로 나눈 몫이 첫번째 계층의 index, 나머지를 다시 NINDIRECT로 나눈 몫을 두번째 계층의 index로 사용했으며, 그에 따른 나머지는 세번째 계층의 index로 사용했다.

bmap 함수 뿐만 아니라 파일 삭제에 쓰이는 itrunc 함수도 수정해야 한다. 계층 구조로 저장하였기 때문에 가장 끝 계층부터 free 해주어야 한다. 수정한 내용은 다음과 같다.

fs.c – itrunc 함수

```
if(ip->addrs[NDIRECT+1]){ // For double indirect
    bp = bread(ip->dev, ip->addrs[NDIRECT+1]);
    a = (uint*)bp->data;
    for(i = 0; i < NINDIRECT; i++){
      if(a[i]){
        bp_2 = bread(ip->dev, a[i]);
        a_2 = (uint*)bp_2->data;
        for(j = 0; j < NINDIRECT; j++){
          if(a_2[j])
            bfree(ip->dev, a_2[j]); // 두번째 계층 free
        }
        brelse(bp_2);
        bfree(ip->dev, a[i]); // 첫번째 계층 free
      }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT+1]);
    ip->addrs[NDIRECT+1] = 0;
}
if(ip->addrs[NDIRECT+2]){ // For triple indirect
    bp = bread(ip->dev, ip->addrs[NDIRECT+2]);
    a = (uint*)bp->data;
    for(i = 0; i < NINDIRECT; i++){
      if(a[i]){
        bp_2 = bread(ip->dev, a[i]);
        a_2 = (uint*)bp_2->data;
        for(j = 0; j < NINDIRECT; j++){
          if(a_2[j]){
            bp_3 = bread(ip->dev, a_2[j]);
            a_3 = (uint*)bp_3->data;
            for(k = 0; k < NINDIRECT; k++){
              if(a_3[k])
                bfree(ip->dev, a_3[k]); //세번째 계층 free
            }
            brelse(bp_3);
            bfree(ip->dev, a_2[j]); // 두번째 계층 free
          }
        }
        brelse(bp_2);
        bfree(ip->dev, a[i]); // 첫번째 계층 free
      }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT+2]);
    ip->addrs[NDIRECT+2] = 0;
}
```

c. Result

Result – triple_indirect_test

```
$ triple_indirect_test
triple indirect test
[1] write test
0 bytes written
51200 bytes written
102400 bytes written
153600 bytes written
```

```
16742400 bytes written
16777216 bytes written
[2] read test
0 bytes read
51200 bytes read
102400 bytes read
153600 bytes read
```

```
16588800 bytes read
16640000 bytes read
16691200 bytes read
16742400 bytes read
16777216 bytes read
$
$
```

Triple Indirect가 작동하는지 보기 위해 16MB를 한 파일에 쓰고 읽는 테스트를 진행하였다. 진행 결과는 위 사진과 같다. 모두 16MB를 쓰고 읽는 모습을 볼 수 있다.

d. Trouble Shooting

bn을 적절히 나누어 각 계층에 맞는 index를 할당하는 부분이 어려웠던 점 빼곤 잘 해결했다.

2. Symbolic Link

a. Design

symlink 시스템콜을 추가해 심볼릭 링크를 구현할 예정이다. 해당 시스템콜은 링크파일을 만들고 안에 연결할 원본 파일의 path와 그 크기를 담아두는 역할을 한다. 이때 링크파일임을 표시하기 위해 T_SYM이라는 타입을 추가한다. 링크파일을 열 때 링크파일을 읽어서 해당 path에 해당하는 원본 파일을 리다이렉션할 것이다. ls명령어 시 파일을 open해서 파일의 타입을 확인하는데 링크파일을 open할 때 리다이렉션하지 않게 open 모드를 따로 추가한다.

b. Implement

파일의 타입을 추가하기 위해 stat.h에 T_SYM을 4로 정의했다. 또, ls를 위해 open 모드를 fcntl.h에 O_SYM을 0x003으로 정의했다. 이후 심볼릭 링크를 위한 시스템콜을 구현한 함수는 다음과 같다.

sysfile.c – sys_symlink 함수

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

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

  if(namei(old)==0) //링크할 파일이 없으면 -1
    return -1;
  begin_op();
  ip = create(new, T_SYM, 0, 0); //링크파일 생성
  if(ip==0){
    end_op();
    return -1;
  }
  //링크파일의 inode에 old의 path 저장
  ip->type = T_SYM;
  int slen = strlen(old);
  if(writei(ip, (char *)&slen, 0, sizeof(slen)) != sizeof(slen)) {
    iunlockput(ip);
    end_op();
    return -1;
  }
  if(writei(ip, old, sizeof(slen), slen+1)<0){
    iunlockput(ip);
    end_op();
    return -1;
  }

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

  return 0;
}
```

전반적으로 sys_link 함수의 형태를 참고했다. 인자로 old엔 원본파일의 path, new엔 만들 링크파일의 path를 받는다. old란 파일이 없으면 실패이며 찾았으면 우선 링크파일을 T_SYM타입으로 create 함수를 통해 만든다. 이후 파일의 타입을 T_SYM으로 지정하여 open을 대비했다. 이후엔 링크파일에 old의 path의 크기를 우선 형변환을 통해 저장했고 그 이후엔 old의 path를 저장한다. type을 바꿨으니 iupdate 함수를 호출하고 0을 반환해 링크파일 생성의 성공을 알린다.

sysfile.c – create 함수

```
static struct inode*
create(char *path, short type, short major, short minor)
{
  struct inode *ip, *dp;
  char name[DIRSIZ];

  if((dp = nameiparent(path, name)) == 0)
    return 0;
  ilock(dp);
```

```
  if((ip = dirlookup(dp, name, 0)) != 0){
    iunlockput(dp);
    ilock(ip);
    if(type == T_FILE && ip->type == T_FILE)
      return ip;
    if(type == T_SYM) // symlink면 return ip
      return ip;
    iunlockput(ip);
    return 0;
  }
  . . .
```

위 사진은 create 함수에서 T_SYM을 처리하는 부분을 추가한 모습이다.

sysfile.c – sys_open 함수

```
link:
    if(ip->type == T_SYM && omode != O_SYM){ // type이 symlink면 ip를 변경. ls에서 구분을 위해 omode를 추가
      ilock(ip);
      readi(ip, (char *)&path_size, 0, sizeof(int)); // path의 size 불러오기
      readi(ip, path_sym, sizeof(int), path_size+1); // path의 사이즈에 맞게 path 불러오기
      iunlock(ip);
      path_sym[path_size]=0; // 끝에 null 확실히 넣어주기
      if((ip = namei(path_sym)) == 0){
        cprintf("maybe file is deleted\n"); // 못 찾았으면 삭제된 걸수도
        end_op();
        return -1;
      }
      if(ip->type == T_SYM)
        goto link;
    }
    if(omode==O_SYM) // ls는 ip변경 x
      omode = O_RDONLY;
```

위 사진은 파일을 열 때 리다이렉션하기 위한 코드이다. 파일이 T_SYM 타입이며 O_SYM을 제외한 omode이면 readi를 통해 path의 길이를 불러오고 다시 readi를 통해 path의 크기만큼 path_sym에 path를 불러온다. O_SYM은 ls와 stat 함수에서만 사용되는 모드다. 이후 path에 해당하는 ip를 불러옴으로써 리다이렉션이 마무리된다. 만약에 원본 파일이 링크파일이라면 다시 리다이렉션을 하기위해 goto 문법을 사용해 지금까지의 일을 다시 반복한다. 이는 원본파일이 링크파일이 아닐 때 까지 반복된다.

ls.c – ls 함수

```
void
ls(char *path)
{
  char buf[512], *p;
```

```
  int fd;
  struct dirent de;
  struct stat st;

if((fd = open(path, O_SYM)) < 0){
    printf(2, "ls: cannot open %s\n", path);
    return;
  }

  if(fstat(fd, &st) < 0){
    printf(2, "ls: cannot stat %s\n", path);
    close(fd);
    return;
  }

  switch(st.type){
  case T_FILE:
    printf(1, "%s %d %d %d\n", fmtname(path), st.type, st.ino, st.size);
    break;

  case T_SYM:
    printf(1, "%s %d %d %d\n", fmtname(path), st.type, st.ino, st.size);
    break;

  case T_DIR:
    if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){
      printf(1, "ls: path too long\n");
      break;
    }
    strcpy(buf, path);
    p = buf+strlen(buf);
    *p++ = '/';
    while(read(fd, &de, sizeof(de)) == sizeof(de)){
      if(de.inum == 0)
        continue;
      memmove(p, de.name, DIRSIZ);
      p[DIRSIZ] = 0;
      if(stat(buf, &st) < 0){
        printf(1, "ls: cannot stat %s\n", buf);
        continue;
      }
      printf(1, "%s %d %d %d\n", fmtname(buf), st.type, st.ino, st.size);
    }
    break;
  }
. . .
```

위는 ls.c의 일부이다. open의 모드를 O_SYM으로 설정하여 open시 리다이렉션이 일어나지 않게 설정했다. 만약 shell에서 ls만 쳤을 때엔 현재 디렉토리인 . 대해 ls를 하는데 이때 각각에 대해 stat 함수가 호출된다. 따라서 stat 함수의 open의 모드도 O_SYM으로 설정했다.

ln.c

```c
int
main(int argc, char *argv[])
{
  if(argc != 4){
    printf(2, "Usage: ln [option] old new\n");
    exit();
  }
  if(argv[1][1] == 'h'){ // h이면 하드링크
    if(link(argv[2], argv[3]) < 0)
      printf(2, "hard link %s %s: failed\n", argv[2], argv[3]);
    exit();
  }
  if(argv[1][1] == 's'){ // s이면 심볼릭링크
    if(symlink(argv[2], argv[3]) < 0)
      printf(2, "symbolic link %s %s: failed\n", argv[2], argv[3]);
    exit();
  }

  printf(2, "link failed\n");
  exit();
}
```

다음은 ln 명령어에 옵션을 추가하기 위해 ln.c를 수정한 모습이다. 받는 인자를 늘리고 옵션 자리에 h면 기존의 하드링크, s면 심볼릭 링크파일을 만드는 것으로 구현했다.

c. Result

Result – symlink_test

```
$ symlink_test
Successed to create target file
Successed to create symbolic link
Successed to open symbolic link
Successed to read symbolic link
Symbolic Link Contents: Hello, World!
```

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15496
echo           2 4 14376
forktest       2 5 8812
grep           2 6 18332
init           2 7 15000
kill           2 8 14460
ln             2 9 14652
ls             2 10 16984
mkdir          2 11 14484
rm             2 12 14464
sh             2 13 28516
stressfs       2 14 15392
wc             2 15 15912
zombie         2 16 14032
triple_indirec 2 17 15876
symlink_test   2 18 15592
sync_test      2 19 16168
console        3 20 0
hugefile       2 21 16777216
file.txt       2 22 15
symlink.txt    4 23 13
```

첫번째 사진은 symlink_test.c 안에서 심볼릭 링크를 테스트한 내용이다. symlink_test.c의 내용은 다음과 같다. file.txt를 생성하고 'Hello, World!₩n'을 write하고, symlink 시스템콜을 호출해 symlink.txt를 file.txt의 링크파일로 만든다음, symlink.txt를 read한다. 첫번째 사진의 마지막줄에 Hello, World! 가 출력되는 것으로 보아 파일을 잘 리다이렉션함을 알 수 있었다. ls 명령어를 통해 링크파일인 symlink.txt를 살펴보면 다른 파일과는 다른 타입인 T_SYM인 4를 출력하고 파일의 크기 또한 원본 파일의 크기와는 다른 모습을 볼 수 있다.

Result – 원본 파일 삭제의 경우

```
$ rm file.txt
$ ls symlink.txt
symlink.txt    4 23 13
$ cat symlink.txt
maybe file is deleted
cat: cannot open symlink.txt
```

다음은 원본 파일이 삭제 됐을 때 링크파일이 ls엔 잡히지만 cat하면 열리지 않는 모습이다.

Result – ln -s 명령어

```
file.txt       2 25 15
symlink.txt    4 26 13

$ ln -s file.txt sym.txt
$ ls sym.txt
sym.txt        4 27 13
$ cat sym.txt
Hello, World!
$ ln -s symlink.txt symsym.txt
$ ls symsym.txt
symsym.txt     4 28 16
$ cat symsym.txt
Hello, World!
```

다음은 원본 파일인 file.txt에 ln 명령어를 통해 심볼릭 링크 파일(sym.txt)을 만들고 ls, cat한 내용과, 링크 파일(symlink.txt)의 링크파일(symsym.txt)을 만들고 ls, cat한 내용이다.
ln -s 명령어로 심볼릭 링크파일이 만들어지고 ls, cat 또한 의도한 방향으로 잘 작동했다. 또한 링크파일의 링크파일인 symsym.txt에 ls, cat 명령어 또한 의도한 방향으로 잘 작동했다.

Result – ln -h 명령어

```
$ ln -h file.txt hard1.txt
$ ls file.txt hard1.txt
file.txt       2 25 15
hard1.txt      2 25 15
```

하드링크에 대해선 ln.c에서 옵션 처리만 해주었고 하드링크를 구현하는 link 함수는 수정한 사항이 없으므로 잘 작동했다.

d. Trouble Shooting

sys_symlink 함수를 구현할 때 path의 길이를 생각 못하고 writei와 readi에 적당히 큰 길이를 인자로 넣어서 저장하고 읽는 부분에서 문제가 발생했다. 저장에는 문제가 없었는데 읽을 때 trap14가 발생해서 너무 큰 길이는 오류가 생김을 깨달아 path의 길이 또한 저장하고 읽음으로써 이를 해결했다.

3. Sync

a. Design

xv6에선 begin_op, end_op, commit 함수를 통해 Disk I/O가 진행된다. group flush이 아닌 buffered I/O를 구현하기 위해 기존에 end_op에서 commit하던걸 더 이상 하지 않게 한다. sync를 하는 시스템콜 sys_sync를 구현하고 begin_op에서 buffer에 공간이 부족할지 판단하고 부족하다면 미리 sync하도록 구현한다. 또는 sync 시스템콜이 호출되었을 때 sync하도록 구현한다.

b. Implement

log.c – end_op 함수

```
void
end_op(void)
{
  int do_commit = 0;

  acquire(&log.lock);
  log.outstanding -= 1;
  if(log.committing)
    panic("log.committing");
  if(log.outstanding == 0){
    // do_commit = 1;
    // log.committing = 1;
```

우선 더이상 end_op이 commit을 하지 않도록 해당 내용을 주석 처리했다.

log.c – commit 함수

```
static void
commit()
{
  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
    log.lh.n = 0;
    write_head();    // Erase the transaction from the log
  }
}
```

기존의 commit 함수로 buffer cache에서 log로, disk로 데이터가 이동하게 하는 함수이다. commit을 해야 변경된 데이터를 log에도 쓰고 disk에도 최종적으로 저장하는 것이다. 따라서 commit 함수 자체를 sync_ 함수에 적용했다.

log.c – sync_ 함수

```
int
sync_(void)
{

  int block_num = log.lh.n; // sync될 block들
```

```
  if(block_num == 0)
    return -1;
  else
    commit(); // disk에 저장

  return block_num;
}
```

log.lh.n은 현재 cache의 block수를 의미하므로 저장해두고 이 block의 수가 0이면 sync할 내용이 없으므로 -1을 리턴한다. 아닐 경우 commit 함수를 통해 cache의 block들을 log에 쓰고 disk에 쓴다. 리턴 값은 flush에 성공된 block수인 block_num이다.

log.c – begin_op 함수

```
void
begin_op(void)
{
  acquire(&log.lock);
  while(1){
    if(log.committing){
      sleep(&log, &log.lock);
    } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
      // this op might exhaust log space; wait for commit.
      log.committing = 1;
      release(&log.lock);
      sync_();
      acquire(&log.lock);
      log.outstanding = 0; // commit했으니 초기화
      log.committing = 0;
      wakeup(&log);
    } else {
      log.outstanding += 1;
      release(&log.lock);
      break;
    }
  }
}
```

다음은 수정한 begin_op 함수의 모습이다. else if문은 기존의 begin_op을 그대로 사용했다. 해당 else if문에 걸리면 더 이상 버퍼에 쓸 수 없다고 판단하고 sync를 진행한다.

sysfile.c – sys_sync 함수

```
int
sys_sync(void)
{
  return sync_();
}
```

기존에 sync를 하는 sync_ 함수를 다시 return 함으로써 sync 시스템콜을 구현했다.

c. Result

Result – sync_test

```
$ sync_test
[Test] file before sync(file_before_sync.txt)
Successed to create target file
Successed to open target file
Successed to read target file
target file Contents: Hello, World!

[Test] file with sync(file_sync.txt)
sync blocks 5
Successed to create target file
Successed to open target file
Successed to read target file
target file Contents: Hello, World!

[Test] file after sync(file_after_sync.txt)
Successed to create target file
Successed to open target file
Successed to read target file
target file Contents: Hello, World!
```

[sync를 하기 전에 write하고 close, read한 파일-1]과 [write, sync, close, read한 파일-2], [sync 후에 write, close, read한 파일-3] 3개로 구성했다. sync를 하지 않았음에도 read를 할 수 있었던 건 read 시 disk가 아닌 캐싱된 데이터를 사용했기 때문이라 판단했다. 따라서 sync를 한 시점에 buffer에 있던 파일-1, 파일-2는 디스크에 저장됐을 것이고 sync이후인 파일-3는 disk에 저장되지 않았을 것이다. 이는 간단히 xv6를 껐다가 다시 킴으로써 확인할 수 있었다.

Result – Rebooting xv6

```
file_before_sy 2 21 15
file_sync.txt  2 22 15
file_after_syn 2 23 15
```

```
file_before_sy 2 21 15
file_sync.txt  2 22 15
$
```

첫번째 사진은 sync_test 직후의 ls한 사진이고 두번째는 xv6를 재부팅 후의 ls한 사진이다. sync되지 않은 파일-3이 디스크에 저장되지 않았음을 확인했다.

Result – triple_indirect_test –> Rebooting xv6

```
sync_test      2 19 16684
console        3 20 0
hugefile       2 21 16774656
$
```

기존의 triple_indirect_test가 sync없이 512B씩 16MB까지 쓰는 test였기에 begin_op에서 제대로 buffer가 가득차기 전에 sync해주는지 테스트 할 수 있었다. 사진은 재부팅 후 테스트 파일이

16MB인 16,777,216B에서 16,774,656B로 줄은 모습이다. buffer가 가득 차기 전 sync를 반복하다가 마지막에 buffer가 다 차지 못해 2560B(2.5MB)의 sync되지 못한 데이터들이 손실된 것을 파악할 수 있었다.

d. Trouble Shooting

begin_op에서 기존의 else if문을 썼기 때문에 else if 문이 정말 buffer가 가득 차기 직전에 sync를 하게 해주는지 정확히 의미를 파악하지 못했다. 하지만 정확히 buffer가 다 차는 시점은 아니더라도 그 전에 sync를 하게 되기에 의미가 있다고 생각한다.