운영체제 Project #3

2019082279 김영현

실행방법은 기존과 동일하다 make clean, make , make fs.img, ./bootxv6.sh

[1. Design]

1-1 Multi Indirect

Xv6는 Direct와 Single indirect를 통해 파일의 정보를 저장한다. 다만 기존 방식은 최대 140개 블록에 해당하는 파일만 쓸 수 있다는 한계가 있다. 기존의 xv6의 구조를 보면 On-disk inode structure인 dinode에는 해당 파일 data block들의 주소들이 담긴 addrs배열이 있다. 기존 addrs 배열엔 12개의 direct 주소와 1개의 indirect주소가 저장되어 있고, indirect주소를 통해서 128개의 주소를 더 다룰 수 있는 것이므로 실제로 140개의 블록까지 허용 가능하다는 것이다. 그래서 생각한 것은 이 addrs배열의 direct주소를 추가될 indirect 주소만큼 줄이고, double indirect 주소와 triple indirect 주소를 추가하는 것이다 그리고 이 추가된 indirect 주소도 기존의 xv6에서 indirect 주소가 처리되는 방식으로 구현한다면 충분히 만들 수 있다고 판단하였다.

1-2. Symbolic Link

Xv6는 hard link를 지원한다. Hard link의 방식은 sys_link를 통해 이루어진다. Sys_link를 통해 파일을 만들고, 원본 파일(old)의 inode를 공유하게 되며 이후에 dirlink를 통해 디렉토리에 추가한다.

원래는 sys_link의 코드를 조금만 변형하면 할 수 있지 않을까 했는데 그래도 중요한 것은 inode의 값을 정하는 방식이 sys_link와는 달라야 한다는 것이었다. Sys_link를 보면 inode ip 를 namei(old)를 통해 old path를 통해 접근해서 old inode의 값을 전달하고 dirlink를 통해 ip값 또한 정해주는 것을 볼 수 있었다. 그래서 기존의 sys_link 방식을 그대로 쓰기엔 구현하기 힘들어진다 판단했고 sysfile.c에 이미 있던 **create 함수를 이용하기로 판단했다. 또한 이론 수업 pdf에 있는 copy = create + readi + writei라고 적혀있는 내용에서 힌트를 얻었다.** Create과 sys_link의 차이점은 바로 create는 ialloc을 통해 type을 설정한 inode를 새로 할당할 수 있다는 것이다. Nameiparent를 통해 디렉토리를 찾고 dirlookup을 통해 이름을 검색하고 dirlink를 호출해서 파일을 만드는 것까진 동일했지만 inode를 새로 할당해서 inode를 return해준다는 것이 달랐다. 따라서 create방식을 이용하기로 했다. 다만 create를 통해서 inode를 새롭게 할당 받았을 때 symbolic link의 기능을 구현하기 위해선 inode에 가리키고자 하는 파일의 path를 기록해야 한다는 것이었다. 그래서 이 부분은 dirlink에서 디렉토리의 inode에 readi, writei를 통해 정보를 기록했다는 것에서 영감을 얻어서 writei를 하고 symbolic link를 sys_open을 통해 접근할 경우 readi를 해서 가리키는 파일의 path 정보를 받아와서 다시 한번 namei를 통해 해당 파일로 접근 가능하도록 하게 만들었다. 이렇게 하면 가리키고자 하는 파일이 사라져도 namei에서 없어진 파일로 인해서 문제가 발생하지 않으므로 접근이 불가능한 것도 구현이 가능해지는 것이었다.

1-3. sync

일반적으로 disk I/O가 컴퓨터 operation중에서 가장 느린 operation임에도 불구하고, 기존의 xv6는 write operation에 대해서 group flush, 즉 한 번에 flush를 하는 방식을 채용하고 있었다. 이러한 방식은 특정 프로세스가 다수의 write operation을 발생시킬 때 성능 저하시키는 문제점이 존재했다. 이걸 해결하기 위해 buffered I/O를 구현해서 sync 함수가 호출될 때만 flush하거나 buffer에 공간이 부족할 경우에 강제로 sync를 발생시키는 방식을 구현해야 했다. Xv6는 모든 FS system call이 호출될 때마다 begin_op가 수행되고, FS system call이 종료될 시에 end_op를 호출했다. Begin_op를 호출할 경우, log가 commit중이라면 sleep하고 아니면 outstanding operation 개수가 LOGSIZE를 초과할 경우에 sleep시킨다. 그 이외의 경우엔 log.outstanding을 증가시킨다. End_op에선 log.outstanding을 줄이고 log가 commit 중이 아니고 log.outstanding 개수가 0이 되고나서야 flag들을 setting하고 commit한다. Commit 에선 log를 disk에 기록하고 비운다. Sync를 통해서 flush가 이루어져야 한다면 begin_op에서는 LOGSIZE를 넘지 않으면 sleep시키고 end_op에선 outstanding operation이 없더라도 buffer가 꽉 차서 LOGSIZE를 넘는 게 아니라면 commit이 되지 않도록 했다. Sync 시스템콜에선 log가 commit이나 outstanding하지 않는 것을 기다리다가 commit하는 방식으로 구현했다.

[2. Code Summary]

```
#define FSSIZE       2100000  // size of file system in blocks
```

(param.h) 128*128*128을 고려해서 했다.

```
#define NDIRECT (12-2)//space for d_indir, t_indir
#define D_NINDIRECT_ADRS  11
#define T_NINDIRECT_ADRS 12
#define NINDIRECT (BSIZE / sizeof(uint))//일반 indirect
#define D_NINDIRECT ((NINDIRECT) * (NINDIRECT))//double indirect
#define T_NINDIRECT ((D_NINDIRECT) * (NINDIRECT))//triple indirect
#define MAXFILE ((NDIRECT) + (NINDIRECT) + (D_NINDIRECT) + (T_NINDIRECT))
```

(fs.h) multi-direct를 구현하기 위해 매크로를 설정했다. 기존 xv6 방식을 참고했다.

```
  uint addrs[NDIRECT+1+2];   // Data block addresses /for project03, d_ind,
t_ind
```

(fs.h) dinode의 address 배열을 수정했다.

```
static uint
bmap(struct inode *ip, uint bn) //bn = block number
```

(fs.c – bmap func) 기존 xv6의 indirect와 direct에서 balloc하는 방식을 기반으로 구현하고자 했다.

```
  bn -= NINDIRECT;

  //////////////////////////Double Indirect
  if(bn < D_NINDIRECT){
```

(fs.c – bmap func) block number에서 indirect만큼 빼고 다시 double indirect를 기존 indirect방식

과 유사하게 구현한다. 다만 차이점이라면 indirect를 두 번 거치기 때문에 그 부분을 고려했다.

```
if((addr = a[par_bn]) == 0)
 if((addr = a[bn]) == 0){
```

(fs.c – bmap func) double indirect를 위한 if문이다.

```
  bn -= D_NINDIRECT;

  //////////////////////////Triple Indirect
  if(bn < T_NINDIRECT){
```

(fs.c – bmap func) triple indirect도 indirect와 마찬가지로 구현했다.

```
    if((addr = a[par_par_bn]) == 0)
    if((addr = a[par_bn]) == 0){
    if((addr = a[bn]) == 0){
```

(fs.c -bmap func) triple indirect를 구현하기 위한 if문이다. 중요한 점은 block number를 식별하기 위해 double indirect의 개수와 indirect개수를 나누고 나머지를 구하는 방식을 통해서 해당 block number를 구하는 것이다.

```
static void
itrunc(struct inode *ip)
```

(fs.c – itrunc func) 기존 xv6에서 inode를 truncate할 때 사용하는 함수이며 기존의 방식을 참고하여 multi indirect를 구현하고자 했다.

```
  //for double_indirect : d -> ()
  if(ip->addrs[D_NINDIRECT_ADRS]){
```

(fs.c – itrunc func) double indirect의 블록을 free하기 위한 것이다. Double indirect이기 때문에 이중 for문을 통해서 접근한다.

```
  //for triple_indirect : t -> d -> ()
  if(ip->addrs[T_NINDIRECT_ADRS]){
```

(fs.c – itrunc func) triple indirect의 블록을 free하기 위한 것이다. Triple indirect이기 때문에 삼중 for문을 통해서 접근한다.

```
#define T_SYMLINK 4// Symbolic link
```

(stat.h) symbolic link를 위한 inode를 구별하기 위한 type이다.

```
//for project03 ..symlink
int
sys_symLink(void)
  if ((ip = create(new, T_SYMLINK, 0, 0)) == 0) {
  int old_len = strlen(old);
  writei(ip, (char*)&old_len, 0, sizeof(int)); // write to inode for reading
                                                 old path
  writei(ip, old, sizeof(int), old_len+1); // write old path
```

(sysfile.c – sys_symLink) symbolic link를 구현하기 위한 시스템 콜이다. create함수를 통해 파일을

생성한 후 inode를 받은 다음에 writei를 통해 inode에 old path를 기록한다. 다만 readi를 할 때 old path의 길이를 알고 있어야 하므로 old path의 길이를 먼저 writei 하여 정보를 습득할 수 있도록 한다. Strlen의 경우 널문자를 생략하기 대문에 1더한 길이를 보낸다. Writei는 off부터 n만큼 2번째 char* type 인자에 write한다.

```
if(ip->type == T_SYMLINK)
    {
      int read_len = 0;
      readi(ip, (char*)&read_len, 0, sizeof(int));//read pathlen in inode

      // Allocate memory for the target path
      char wanted_path[MAXPATH];

      readi(ip, wanted_path, sizeof(int), read_len+1);//read path in inode
      iunlockput(ip);

      if((ip = namei(wanted_path)) == 0){//get inode of path
```

(sysfile.c – sys_open) symbolic link를 했을 때 symbolic link용도인 파일을 open 했을 때, 해당 path의 inode를 namei를 통해서 전달받은 후, inode의 타입이 symbolic link인 것을 확인한 후, readi를 통해 inode에 적힌 old path를 전달 받는다. 전달받는 과정에서 project2 당시의 max path 50을 참고로 해서 선언한 param.h의 MAXPATH 크기의 배열 wanted_path에 old path를 복사한다. 이후, wanted_path에 기록된 old path와 namei를 통해서 타겟 inode의 정보에 접근해서 open할 수 있도록 한다.

```
  if(log.outstanding == 0){
    if(log.lh.n + MAXOPBLOCKS > LOGSIZE){/////for project03
      do_commit = 1;
      log.committing = 1;
    }
```

(log.c – end_op func) log.outstanding 개수가 0이 되어도 log buffer가 다 차지 않았으면 처리하지 않는다.

```
//for project03, sync function
int
sync(void)
  do{
    if(log.committing || log.outstanding){
      sleep(&log, &log.lock);
    }
    log.committing = 1;
    release(&log.lock);
    commit();
    acquire(&log.lock);
    log.committing = 0;
    wakeup(&log);
```

```
  }while(log.committing || log.outstanding);
```

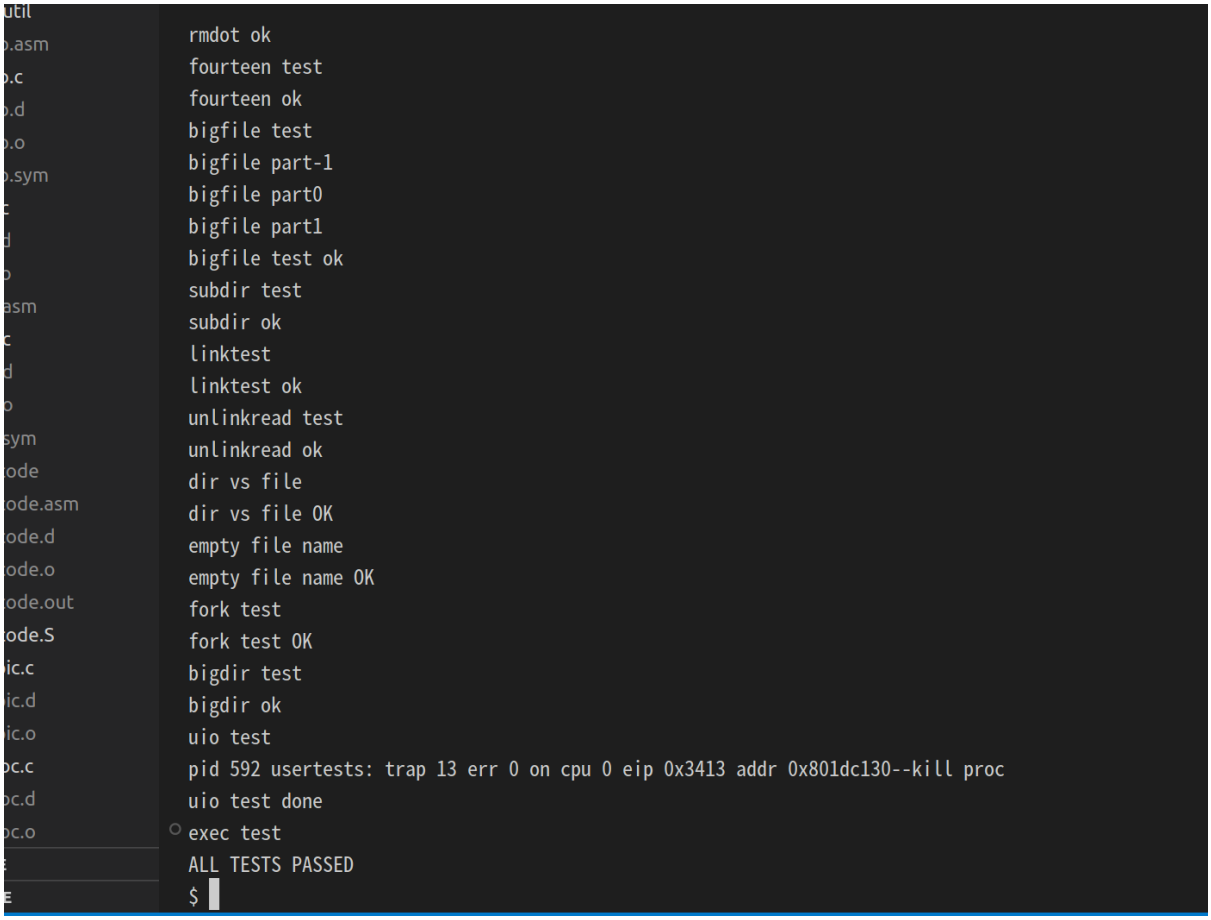
(log.c – sync func) sync가 호출되면 commit한다.

[3. Result]

```
$ stressfs
stressfs starting
write 0
write 1
write 2
write 3
write 4
read
read
read
read
read
$
```

Xv6 자체 stressfs 결과이다.

Stressfs를 한 후에 ls를 한 결과이다



xv6 자체 usertests를 한 결과이다.

[4. Trouble Shooting]

Multi-indirect를 구현하기 위해서 bmap에서 block number를 구하는 과정에서 혼동해서 bn을 잘못 구하는 바람에 많은 시간이 소요되었다. 또한 itrunc함수에서 삼중 루프를 작성할 때, block을 read하고 free하는 과정에서 여러 변수들을 작성하다보니 이것 또한 헷갈려서 시간이 지체되었다.

Symbolic link를 구현하는 부분에서 많은 고민을 했다. 원래 처음에는 link함수와 dirlink의 내용을 전부 가져와서 수정해서 했는데 헷갈려서 구현하기 힘들었다. 그러던 중에 OS 15 이론 pdf에서 copy = create + read + write를 발견했다. create함수를 보니 이용할 수 있을 것 같아서 create를 이용하여 구현했다.