Project03 - File System Wiki

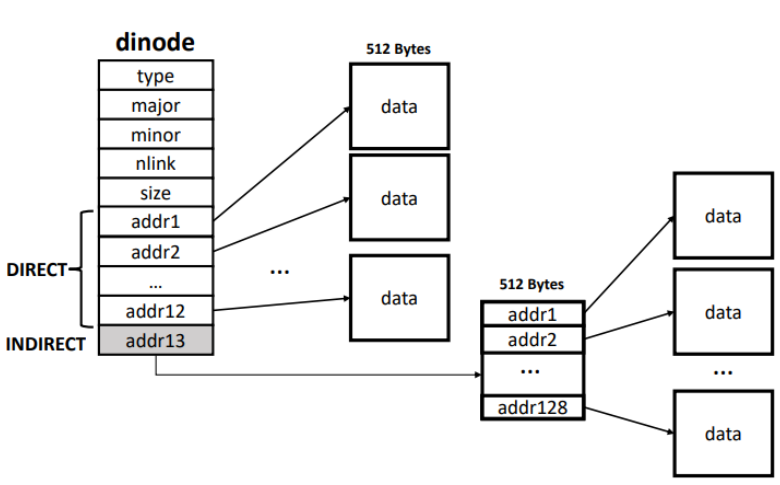
학과 : 컴퓨터소프트웨어학부

학번 : 20*****40

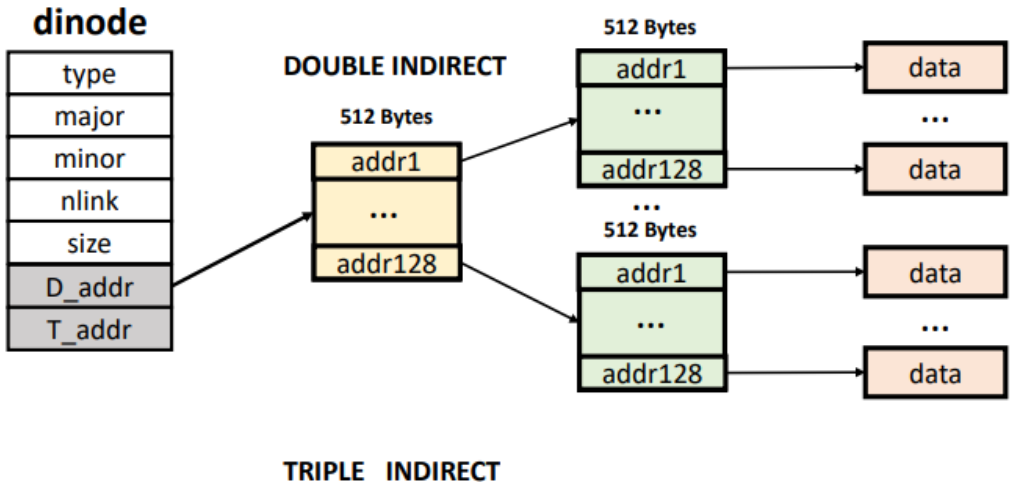
작성자 : 김휘수

1. Design

1-1. Mutil-Indirect



기본 xv6 File System

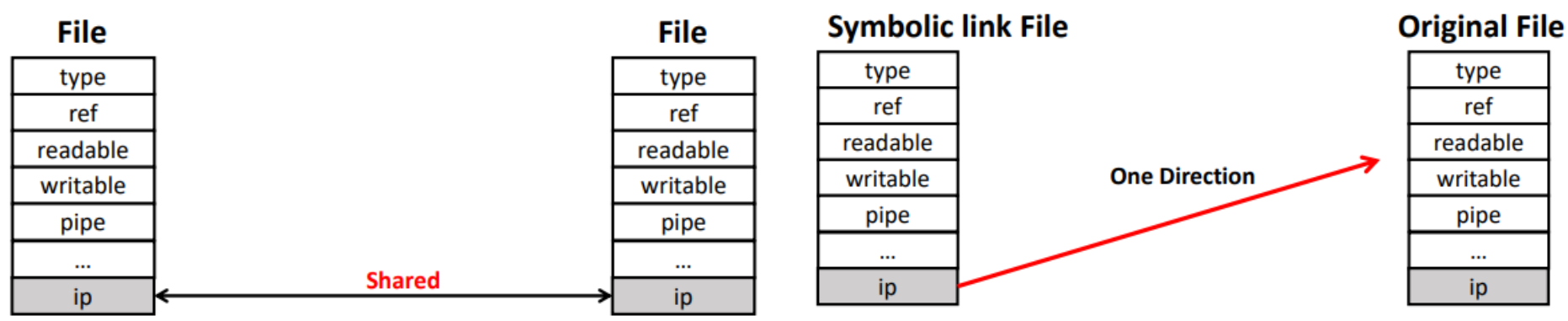


수정된 xv6 File System

Multi-Indirect의 구현을 위해서 기존의 Direct로 구현되어 있는 일부의 배열 칸을 **빌려서 Multi-Indirect를 위한 공간으로 활용**하면, 해당 공간에 대해 내부에 최대 128개의 노드가 존재할 수 있다.

Double Indirect를 이용하면 대략 8MB, Triple Indirect를 이용하면 대략 1GB정도까지의 File에 대한 I/O가 가능하다.

1-2. Symbolic Link (Soft Link)



[Hard Link에 대한 File의 Metadata] **[Soft Link에 대한 File의 Metadata]**

xv6에서는 Hard Link에 대해서는 지원하지만, Soft Link에 대해서는 지원하지 않는다.

Hard Link는 원본 파일의 Inode를 공유하는 파일이다. 각각의 File 자체는 개별적으로 존재하지만, 서로의 Inode를 공유하기 때문에, 변경 내용 등은 공유되면서, 다른 하나가 삭제되어도, 다른 File은 영향을 받지 않아야 한다.

반면, Soft Link File은 원본 파일에 대한 re-direction을 해주는 File이다.(Windows의 바로가기)

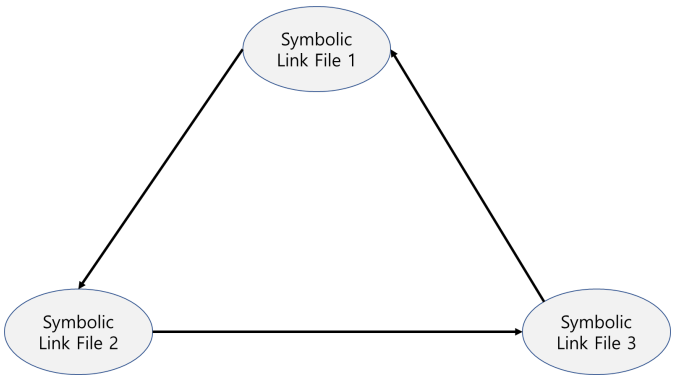
그렇기 때문에, Soft Link File은 자기 자신이 새로운 File을 가지지 않는다.

```
struct inode {

 ...
 ...
 ...
 short type;
 uint size;
 uint addrs[NDIRECT + 3];
 };
```

- 이를 위해서 기존의 xv6의 Inode 자료구조를 이용하여, 사용하지 않는 `addr`을 이용하여 원본 File에 대한 Path를 저장하여 해당 File에 접근할 때에 Re-direction을 해준다.
- Soft Link File의 Inode에서 `type`을 통해서 Inode가 soft link임을 나타낸다.
- Soft Link File의 Inode에서 `addr[NDIRECT + 3]`에 re-direction해주기 위한 Pathname과 그 길이를 저장했다.

1-2-1. Cyclic Symbolic Link



Symbolic Link는 file에 대한 metadata인 `inode`를 공유하는 방식이 아니라, 단순히 redirection하는 방법이다.

그렇기 때문에 만약 Symbolic Link를 잘못 사용하게 되면, 왼쪽과 같은 Cycle이 생겨 문제가 발생할 수 있다.

이러한 경우를 해결하기 위해서 Windows는 Symbolic File에 대한 Symbolic Link를 Original File에 대한 Symbolic File로 변경해서 관리한다.

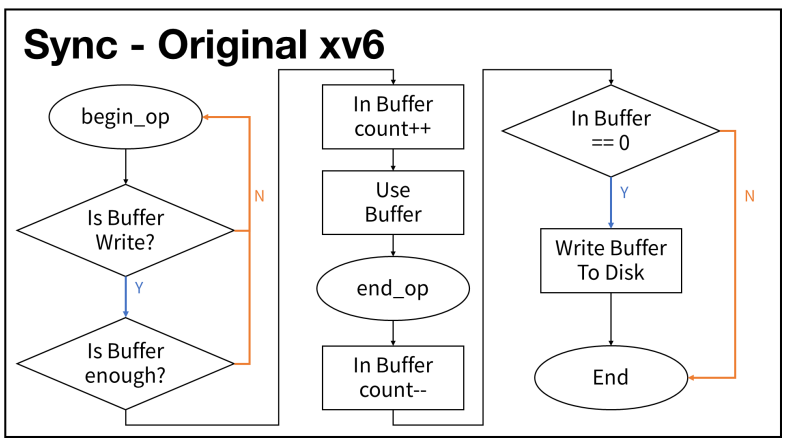
Linux의 경우에는 해당 관계는 허용하지만, 일정 길이 이상에 대해서는 허용하지 않는다.

이번 과제에서는 Linux의 정책을 따라서 길이가 10 이상에 대해서는 Cyclic Symbolic File이라고 가정했다.

1-3. Sync

기존의 xv6의 매커니즘은 Process Sync 수업시간에 배운 readers & writer problem과 상당히 유사하다.

[기존 xv6의 sync]



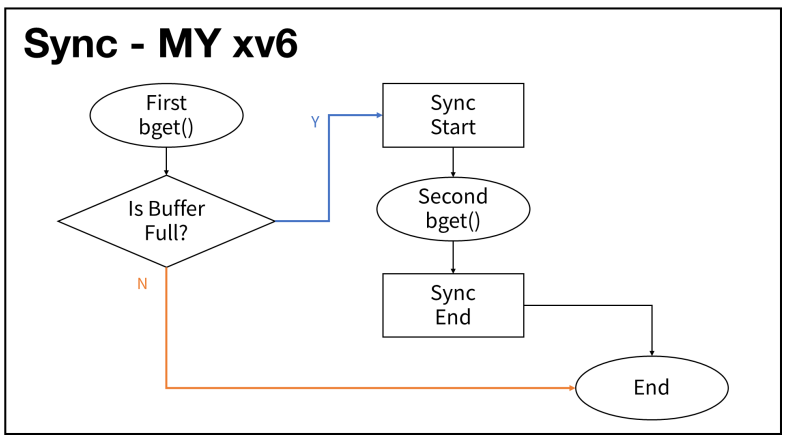
1. buffer에 대한 여유 공간 확인 & buffer를 disk에 쓰는 작업 중인지 확인
    a. 여유 공간이 부족하거나 쓰기 작업 중인 경우에는 대기 후, 과정 1 재시작
    b. 그렇지 않는 경우에는 과정 2 진행
2. buffer를 사용 중임을 표시
3. buffer 사용이 완료되면, 사용을 종료 했음을 표시
4. 어떤 process가 buffer 사용을 종료 했을 때, 아무도 buffer를 사용하고 있지 않다면, buffer를 disk에 쓰는 작업 진행

과제에서 변경해야 할 사항은 buffer에서 더 이상 쓰기가 불가능하면 buffer를 disk에 쓰는 것이다.

이를 위해서 buffer에 대한 요청이 들어온 시점에서 판단해야 했다.

아래는 변경된 xv6의 sync이다.

[변경된 xv6의 sync]



1. buffer에 대한 요청이 들어옴 - `bget()`의 첫 호출
2. buffer의 공간이 가득찼는지 확인
    a. 공간이 가득차지 않았으면, 요청 처리
    b. 그렇지 않는 경우에는 과정 3 진행
3. `sync` 시작
4. 내부적으로 `bget()`의 두 번째 호출
5. `sync` 종료
6. 첫 `bget()` 호출에 대한 buffer에 대한 요청 처리

2. Implement

2-1. Multi-Indirect

2-1-1. `dinode` & `inode`

[fs.h의 `dinode` 자료구조 및 상수]

```c
#define NDIRECT 10
#define NINDIRECT (BSIZE / sizeof(uint)) // 128
#define NDINDIRECT (NINDIRECT * NINDIRECT)
// Number of Double Indirect 128 * 128 = 16384
#define NTINDIRECT (NINDIRECT * NINDIRECT * NINDIRECT)
// Number of Triple Indirect 128 * 128 * 128 = 2'097'152
// #define MAXFILE (NDIRECT + NINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT + NTINDIRECT)
```

```c
// 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 + 1 + 1 + 1];   // Data block addresses
};
```

[file.h의 `inode`]

```c
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 + 1 + 1 + 1];
};
```

- xv6에서 File System과 관련된 자료구조와 상수값이다.
- `NDIRECT`는 inode와 연결되는 data block의 개수이다.
- `NINDIRECT`는 inode와 Single Indirect로 연결되는 data block의 개수이다.
- `NDINDIRECT`는 inode와 Double Indirect로 연결되는 data block의 개수이다.
- `NTINDIRECT`는 inode와 Triple Indirect로 연결되는 data block의 개수이다.
- `MAXFILE`은 inode와 연결될 수 있는 전체 data block의 개수이다.
- 10개의 direct와 1개의 single indirect, 1개의 double indirect, 1개의 triple indirect의 연결 구조를 갖고 있다.
- `dinode`와 `inode`의 기본적인 틀은 같아야하기 때문에, `addrs`의 수를 같도록 선언했다.
- 기존에 선언되어 있었던 `dinode`의 크기를 변경하면 system에 문제가 발생하기 때문에, `NDIRECT`의 수를 12에서 10으로 줄이고 2만큼에 대해서 Double Indirect와 Triple Indirect를 위한 공간으로 사용했다.

2-1-2. `bmap()`

```c
bn -= NINDIRECT;
if (bn < NDINDIRECT) { // double
  int inode_num = bn / NINDIRECT;
  int block_num = bn % NINDIRECT;
  // Load indirect block, allocating if necessary.
  // Allocate indirect inode pointer.
  if ((addr = ip->addrs[NDIRECT + 1]) == 0) { // outside inode
    ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
  }
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;

  if ((addr = a[inode_num]) == 0) { // inside inode
    a[inode_num] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);

  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  if ((addr = a[block_num]) == 0) { // data block
    a[block_num] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);
  return addr;
}
```

```c
bn -= NDINDIRECT;
if (bn < NTINDIRECT) { // triple
  int first_inode_num = bn / (NINDIRECT * NINDIRECT);
  int second_inode_num = (bn % (NINDIRECT * NINDIRECT)) / NINDIRECT;
  int block_num = (bn % (NINDIRECT * NINDIRECT)) % NINDIRECT;
  // Load indirect block, allocating if necessary.
  // Allocate indirect inode pointer.
  if ((addr = ip->addrs[NDIRECT + 1 + 1]) == 0) { // outside inode
    ip->addrs[NDIRECT + 1 + 1] = addr = balloc(ip->dev);
  }
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;

  if ((addr = a[first_inode_num]) == 0) { // first inside inode
    a[first_inode_num] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);

  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  if ((addr = a[second_inode_num]) == 0) { // second inside inode
    a[second_inode_num] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);

  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  if ((addr = a[block_num]) == 0) { // data block
    a[block_num] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);
  return addr;
}
```

- `inode` 안에 있는 block 번호에 해당하는 disk block의 주소를 반환하는 함수이다.
- 기존의 `bmap()` 함수에서 Double Indirect와 Triple Indirect를 mapping하는 부분을 추가했다.

  기본적인 틀은 Single Indirect와 같기 때문에 추가적인 자료구조나 별도의 알고리즘이 필요하지 않다.
- Block 번호를 이용해서 $2^7 = 128$에 대한 나눗셈 연산을 이용하면 해당 block의 inode에서의 위치를 결정할 수 있다.
- Double Indirect의 경우에는 내부에서 한 번 더 data block에 대해서 접근하는 과정이 필요하다.
- Triple Indirect의 경우에는 내부에서 두 번 더 data block에 대해서 접근하는 과정이 필요하다.

2-1-3. `itrunc()`

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

```

```
if (ip->addrs[NDIRECT + 1 + 1]) { // triple
  bp = bread(ip->dev, ip->addrs[NDIRECT + 1 + 1]);
  a = (uint*)bp->data;
  for (i = 0;i < NINDIRECT;i++) {
    if (a[i]) {
      dbp = bread(ip->dev, a[i]);
      da = (uint*)dbp->data;
      for (j = 0; j < NINDIRECT; j++) {
        if (da[j]) {
          tbp = bread(ip->dev, da[j]);
          ta = (uint*)tbp->data;
          for (int k = 0;k < NINDIRECT;k++) {
            if (ta[k])
              bfree(ip->dev, ta[k]);
          }
          brelse(tbp);
          bfree(ip->dev, da[j]);
        }
      }
      brelse(dbp);
      bfree(ip->dev, a[i]);
    }
  }
  brelse(bp);
  bfree(ip->dev, ip->addrs[NDIRECT + 1 + 1]);
  ip->addrs[NDIRECT + 1 + 1] = 0;
}

```

- `inode`의 content를 지울 때 호출되는 함수이다.
- 기존의 `itrunc()` 함수에서 Double Indirect와 Triple Indirect에 대한 truncate 부분을 추가했다.

  기본적인 틀은 Single Indirect와 같기 때문에 추가적인 자료구조나 별도의 알고리즘이 필요하지 않다.
- 내부의 data에 대해서 접근하면서 block을 free해준다.

2-2. Symbolic Link

2-2-1. `sys_symlink()`

```
1  int
2  sys_symlink(void)
3  {
4    char* new, * old;
5    struct inode* ip;
6
7    if (argstr(0, (char**)&old) < 0 || argstr(1, (char**)&new) < 0)
8      return -1;
9    begin_op();
10   // create new inode with SYMLINK type
11   // major, minor don't care
12   // Find ip with old pathname.
13   // If not exist, wrong request.
14   if ((ip = namei(old)) == 0) {
15     end_op();
16     return -1;
17   }
18   // Find ip with old pathname.
19   // If exist, wrong request.
20   if ((ip = namei(new)) != 0) {
21     end_op();
22     return -1;
23   }
24
25   // Create new ip in parent directory.
26   // If can't, wrong request.
27   // ip is locked after returned from create().
28   if ((ip = create(new, T_SYMLINK, 0, 0)) == 0) {
29     end_op();
30     return -1;
31   }
32
33   uint len = (uint)strlen(old);
34   // ip->addrs[0] = len;
35   // memmove((char*)(&ip->addrs[1]), old, len + 1);
36   ip->size = len;
37   memmove((char*)(&ip->addrs[0]), old, len + 1);
38   iupdate(ip);
39   iunlock(ip);
40
41   end_op();
42   return 0;
43 }
```

- Symbolic Link에 대한 system call의 구현이다.
- `old`는 link를 할 파일의 이름이다.
- `new`는 symbolic link를 통해서 만드는 파일의 이름이다.
- Line 14:17, `old`에 해당하는 파일이 있는지 확인한다.

  없으면 비정상적인 경우이므로 -1을 return한다.
- Line 20:23, `new`에 해당하는 파일이 있는지 확인한다.

  이미 존재한다면 비정상적인 경우이므로 -1을 return한다.
- Line 28:31, 새로운 inode를 `T_SYMLINK`타입으로 생성한다.
- Line 33:39, 만들어진 inode에 대해서 사용하지 않는 부분을 이용하여, 정보들을 저장한다. `inode` 구조체와 관련된 세부적인 고민 사항은 **Trouble Shooting**에서 설명이다.
- `stat.h`에서 Symbolic Link File을 구별하기 위해서, `T_SYMLINK`의 상수를 추가했다.

[stat.h]

```
1  #define T_DIR  1   // Directory
2  #define T_FILE 2   // File
3  #define T_DEV  3   // Device
4  #define T_SYMLINK 4 // Symbolic Link
```

2-2-2. `create()`

```
1  static struct inode*
2  create(char* path, short type, short major, short minor)
3  {
4    struct inode* ip, * dp;
5    char name[DIRSIZ];
6
7    if ((dp = nameiparent(path, name)) == 0)
8      return 0;
9    ilock(dp);
10
11   if ((ip = dirlookup(dp, name, 0)) != 0) {
12     ilock(ip);
13     iunlockput(dp);
14     if (type == T_FILE && ip->type == T_FILE)
15       return ip;
16     if (type == T_SYMLINK) { // new T_SYMLINK inode
17       ip->type = T_SYMLINK;
18       return ip;
19     }
20     iunlockput(ip);
21     return 0;
22   }
23
24   if ((ip = ialloc(dp->dev, type)) == 0)
25     panic("create: ialloc");
26
27   ilock(ip);
28   ip->major = major;
29   ip->minor = minor;
30   ip->nlink = 1;
31   if (type == T_SYMLINK && ip->type != T_SYMLINK) {
32     ip->type = T_SYMLINK;
33   }
34   iupdate(ip);
35
36   if (type == T_DIR) {  // Create . and .. entries.
37     dp->nlink++;  // for ".."
38     iupdate(dp);
39     // No ip->nlink++ for ".": avoid cyclic ref count.
40     if (dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
41       panic("create dots");
42   }
43   int t;
44   if ((t = dirlink(dp, name, ip->inum)) < 0) {
45     panic("create: dirlink");
46   }
47
48   iunlockput(dp);
49
50   return ip;
51 }
```

- 전달된 인자에 대한 `inode`를 만들어주는 함수이다.
- Line 7:22, 전달된 path에 대한 잘못된 상황을 체크해주는 부분이다.
  1. `path`에 대한 부모 directory가 없으면 -1을 return한다.
  2. directory에 전달된 `path`의 `name`에 대한 file이 이미 존재하면 -1을 return한다.
- Line 31:33, 생성하고자 하는 `inode`의 `type`이 `T_SYMLINK`일 때, 이를 반영해준다.
- 그 외의 부분은 `create()`에서 이미 구현되어 있는 부분이다.

2-2-3. `readi()` & `writei`

```
1  int
2  readi(struct inode* ip, char* dst, uint off, uint n)
3  {
4    uint tot, m;
5    struct buf* bp;
6    struct inode* oip = 0;
7    oip = ip; // Save Original ip
8    int cyclic = 0;
9    while (1) { // Symbolic Cycle Detection
10     if (ip->type != T_SYMLINK) {
11       break;
12     }
13     if (cyclic >= 10) {
14       // If the links are over 10 times,
15       // It is considered as "Symbolic Cycle".
16       goto bad;
17     }
18     cyclic++;
19     uint len = 0;
20     len = ip->size;
21     char path[MAXFLEN];
22     safestrcpy(path, (char*)(&ip->addrs[0]), len + 1);
23     iunlock(ip);
24     ip = namei(path);
25     if (ip == 0) {
26       ilock(oip);
27       return -1;
28     }
29     ilock(ip);
30   }
31
32   if (ip->type == T_DEV) {
33     if (ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read) {
34       goto bad;
35     }
36     if (oip != ip) {
37       iunlock(ip);
38       ilock(oip);
39     }
40     return devsw[ip->major].read(ip, dst, n);
41   }
42   if (off > ip->size || off + n < off) {
43     goto bad;
44   }
45   if (off + n > ip->size) {
46     n = ip->size - off;
47   }
48   for (tot = 0; tot < n; tot += m, off += m, dst += m) {
49     bp = bread(ip->dev, bmap(ip, off / BSIZE));
50     m = min(n - tot, BSIZE - off % BSIZE);
51     memmove(dst, bp->data + off % BSIZE, m);
52     brelse(bp);
53   }
54   if (oip != ip) {
55     iunlock(ip);
56     ilock(oip);
57   }
58   return n;
59 bad:
60   if (oip != ip) {
61     iunlock(ip);
62     ilock(oip);
63   }
64   return -1;
65 }
```

```
1  int
2  writei(struct inode* ip, char* src, uint off, uint n)
3  {
4    uint tot, m;
5    struct buf* bp;
6    struct inode* oip = 0;
7    oip = ip;
8    int cyclic = 0;
9    while (1) { // Symbolic Cycle Detection
10     if (ip->type != T_SYMLINK) {
11       break;
12     }
13     if (cyclic >= 10) {
14       // If the links are over 10 times,
15       // It is considered as "Symbolic Cycle".
16       goto bad;
17     }
18     cyclic++;
19     uint len = 0;
20     len = ip->size;
21     char path[MAXFLEN];
22     safestrcpy(path, (char*)(&ip->addrs[0]), len + 1);
23     iunlock(ip);
24     ip = namei(path);
25     if (ip == 0) {
26       ilock(oip);
27       return -1;
28     }
29     ilock(ip);
30   }
31
32   if (ip->type == T_DEV) {
33     if (ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write) {
34       goto bad;
35     }
36     if (oip != ip) {
37       iunlock(ip);
38       ilock(oip);
39     }
40     return devsw[ip->major].write(ip, src, n);
41   }
42   if (off > ip->size || off + n < off) {
43     goto bad;
44   }
45   if (off + n > MAXFILE * BSIZE) {
46     goto bad;
47   }
48   for (tot = 0; tot < n; tot += m, off += m, src += m) {
49     bp = bread(ip->dev, bmap(ip, off / BSIZE));
50     m = min(n - tot, BSIZE - off % BSIZE);
51     memmove(bp->data + off % BSIZE, src, m);
52     log_write(bp);
53     brelse(bp);
54   }
55
56   if (n > 0 && off > ip->size) {
57     ip->size = off;
58     iupdate(ip);
59   }
60   if (oip != ip) {
61     iunlock(ip);
62     ilock(oip);
63   }
64   return n;
65
66 bad:
67   if (oip != ip) {
68     iunlock(ip);
69     ilock(oip);
70   }
71   return -1;
72 }
```

- `readi()`는 `inode`에 대한 정보를 읽어들이는 함수이다.
- `writei()`는 `inode`에 대한 정보를 쓰는 함수이다.
- Line 9:30, Symbolic Link에 대한 Cycle여부를 판단하는 부분이다.

  Symbolic Link File인 경우에, `namei`함수를 통해서 변수가 가지고 있는 `pathname`을 이용해서 가리키고 있는 `inode`를 찾는다.

  만약 위의 과정이 10회이상 발생한다면, cycle로 간주하여 -1을 return한다.
- `bad` Label의 역할은 잘못된 실행에 대해서 `oip`와 `ip`가 다른경우에, `ip`의 lock을 풀어주고, `oip`의 lock을 잡아주는 것이다.

  이 과정이 필요한 이유는, Symbolic Link의 경우, 처음 함수를 호출할 때의 `ip`와 함수 종료시의 `ip`가 달라지기 때문에, lock을 유지하기 위해서이다.

2-2-4. `ln.c`

```
1  int
2  main(int argc, char* argv[])
3  {
4    if (argc != 4) {
5      printf(2, "Usage: ln -cmd old new\n");
6      exit();
7    }
8    char* cmd = argv[1];
9    if (!strcmp(cmd, "-h")) {
10     if (link(argv[2], argv[3]) < 0)
11       printf(2, "hard link %s %s: failed\n", argv[2], argv[3]);
12   }
13   else if (!strcmp(cmd, "-s")) {
14     if (symlink(argv[2], argv[3]) < 0)
15       printf(2, "soft link %s %s: failed\n", argv[2], argv[3]);
16   }
17   else {
18     printf(2, "Wrong link %s %s %s: failed\n", argv[1], argv[2], argv[3]);
19   }
20   exit();
21 }
```

- `ln`명령어를 실행하는 함수이다.
- `ln -h [old] [new]`를 입력하면 Hard Link를 실행한다.
- `ln -s [old] [new]`를 입력하면 Symbolic Link를 실행한다.
- Line 9:12, `-h`에 대해서 Hard Link를 실행하는 부분이다.
- Line 13:16, `-s`에 대해서 Symbolic Link를 실행하는 부분이다.
- Line 17:19, 잘못된 명령어에 대해서 오류 메시지를 출력하는 부분이다.

2-3. Sync

2-3-1. `sync()`

```
1  int is_flush;
2  int sync(void) {
3    int n = log.lh.n;
4    if (log.committing) {
5      while (log.committing) {
6        sleep(&log, &log.lock);
7      }
8      wakeup(&log);
9      return 0;
10   }
11
12   acquire(&log.lock);
13   log.committing = 1;
14   is_flush = 1;
15   release(&log.lock);
16
17   // call commit w/o holding locks, since not allowed
18   // to sleep with locks.
19   commit();
20
21   acquire(&log.lock);
22   log.committing = 0;
23   is_flush = 0;
24   wakeup(&log);
25   release(&log.lock);
26   return n;
27 }
```

- Buffer의 내용을 disk에 써주기 위한 `sync`함수이다.
- `n`은 buffer에서 disk로 쓴 개수를 저장하기 위한 변수이다.
- Line 4:10, buffer를 disk에 쓰는 중인 경우를 처리하기 위한 부분이다.

  buffer를 disk에 쓰는 작업이 끝날 때 까지 `sleep()`을 호출한다.

  이후 buffer에 쓰는 작업이 끝나게되면 `while`을 빠져나오게된다.

  이런 경우에 이 함수가 써준 buffer가 없으므로 0을 return한다.
- Line 13:14, 지금 부터 buffer를 disk에 쓰기 작업을 하겠다는 것을 표시한다.
- Line 19, buffer를 disk에 쓴다.
- Line 22:24, 쓰기 작업이 끝난 것을 표시하고 뒤에 대기 중인 process를 깨운다.
- `is_flush`에 대해서는 **Trouble Shooting**에서 자세히 설명한다.

2-3-2. `begin_op()` & `end_op()`

```
1  void
2  begin_op(void)
3  {
4    acquire(&log.lock);
5    while (1) {
6      if (log.committing) {
7        sleep(&log, &log.lock);
8      }
9      else {
10       log.outstanding += 1;
11       release(&log.lock);
12       break;
13     }
14   }
15 }
```

```
1  void
2  end_op(void)
3  {
4    acquire(&log.lock);
5    while (log.committing) {
6      sleep(&log, &log.lock);
7    }
8    log.outstanding -= 1;
9
10   wakeup(&log);
11   release(&log.lock);
12 }
```

- `begin_op()`는 transaction을 시작하기 위한 함수이다.
- `end_op()`는 transaction을 종료하기 위한 함수이다.
- 두 함수 모두 현재 buffer에 대해서 쓰기 작업이 진행 중이면 `sleep()`함수를 실행한다.

2-3-3. `bget()`

```
1  static struct buf*
2  bget(uint dev, uint blockno)
3  {
4    struct buf* b;
5
6    acquire(&bcache.lock);
7    if (!is_flush) {
8      int num_unused_buf = 0;
9      // Check that is there enough buffer.
10     for (b = bcache.head.prev; b != &bcache.head; b = b->prev) {
11       if (b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
12         num_unused_buf++;
13       }
14     }
15
16     // If not enough, does flush.
17     if (num_unused_buf <= 2) {
18       release(&bcache.lock);
19       flush();
20       acquire(&bcache.lock);
21     }
22   }
```

- `sync()`의 호출 뿐만 아니라, buffer가 가득차면 disk에 써주는 작업을 하기위해서 `bget()`안에 `flush()`함수를 추가하였다.
- `flush()`함수를 호출하는 조건은 buffer가 가득 찬 경우이다.
- buffer가 가득 차는 경우를 세어주기 위해서 내부에 사용하지 않는 buffer를 세어주고, 개수가 2 이하인 경우에 `flush()`를 호출하였다.
- 개수가 2 이하인 이유에 대해서는 **Trouble Shooting**에서 설명한다.
- `is_flush`는 현재 commit를 해야하는지 확인하기 위한 `extern int` 변수이다.

2-3-4. `flush()`

```
1  int flush(void) {
2    int n = log.lh.n;
3    if (log.committing) {
4      wakeup(&log);
5      return 0;
6    }
7    acquire(&log.lock);
8    log.committing = 1;
9    is_flush = 1;
10   release(&log.lock);
11
12   // call commit w/o holding locks, since not allowed
13   // to sleep with locks.
14   commit();
15   acquire(&log.lock);
16   log.committing = 0;
17   is_flush = 0;
18   wakeup(&log);
19   release(&log.lock);
20   return n;
21 }
```

- Line 3:6, 만약 쓰기 작업이 진행 중이면, 0을 return한다.

  이 과정이 필요한 이유는 **Trouble Shooting**에서 설명한다.
- Line 7:17은 `sync()`의 작업과 같기 때문에 설명은 생략한다.

2-4. System calls

```
1  #define SYS_symlink 22
2  #define SYS_sync    23
```

[syscall.h]

```
1  SYSCALL(symlink)
2  SYSCALL(sync)
```

[usys.S]

```
1  extern int sys_symlink(void);
2  extern int sys_sync(void);
```

```
1  [SYS_symlink] sys_symlink,
2  [SYS_sync] sys_sync,
```

[syscall.c]

```
1  int symlink(const char*, const char*);
2  int sync(void);
```

[user.h]

- Symbolic Link와 Sync에 대한 system call을 지원하기 위해서 추가한 내용이다.

2-5. Defines

```
#define LOGSIZE      (MAXOPBLOCKS*10)  // max data blocks in on-disk log
#define NBUF         (MAXOPBLOCKS*10)  // size of disk block cache
#define FSSIZE       2500000  // size of file system in blocks
#define MAXFLEN      1500  // size of file system in blocks
```

[param.h]

```
extern int is_flush;
int sync(void);
int flush(void);
```

[defs.h]

- 새롭게 정의한 상수들과 함수들이다.
- `FSSIZE`는 xv6의 file system의 전체 block의 수이다.
- `LOGSIZE`와 `NBUF`는 Buffer의 전체 크기이다.

---

3. Result

3-1. Test 1 - Tindirect_test

[Test 1의 소스 코드]

```
void triple_indirect_test(void)
{
    char filename[] = "triple_indirect_test_file";

    // 파일 생성
    int fd = open(filename, O_RDWR | O_CREATE);
    if (fd < 0) {
        printf(2, "Failed to create file\n");
        return;
    }

    char buf[BSIZE];
    memset(buf, 'A', BSIZE);

    // 파일에 데이터 블록 할당 (triple indirect)
    int i, j;

    printf(2, "direct block test\n");
    for (i = 0; i < NDIRECT; i++) {
        if (write(fd, buf, BSIZE) != BSIZE) {
            printf(2, "Failed to write data block\n");
            close(fd);
            return;
        }
    }
    printf(2, "direct block test passed\n");

    // Single indirect block
    printf(2, "single indirect block test\n");
    for (i = 0; i < NINDIRECT; i++) {
        if (write(fd, buf, BSIZE) != BSIZE) {
            printf(2, "Failed to write data block\n");
            close(fd);
            return;
        }
    }
    printf(2, "single indirect block test passed\n");

    // Double indirect block
    printf(2, "double indirect block test\n");
    for (i = 0; i < NINDIRECT; i++) {
        int block_addr = 0;

        if (write(fd, &block_addr, sizeof(int)) != sizeof(int)) {
            printf(2, "Failed to write double indirect block address\n");
            close(fd);
            return;
        }
        for (j = 0; j < NINDIRECT; j++) {
            if (write(fd, buf, BSIZE) != BSIZE) {
                printf(2, "Failed to write data block\n");
                close(fd);
                return;
            }
        }
    }
    printf(2, "double indirect block passed\n");

    // Triple indirect block
    printf(2, "triple indirect block test\n");
    for (i = 0; i < NINDIRECT; i++) {
        int block_addr = 0;

        if (write(fd, &block_addr, sizeof(int)) != sizeof(int)) {
            printf(2, "Failed to write triple indirect block address\n");
            close(fd);
            return;
        }
        for (j = 0; j < NINDIRECT; j++) {
            int indirect_block_addr = 0;

            if (write(fd, &indirect_block_addr, sizeof(int)) != sizeof(int)) {
                printf(2, "Failed to write indirect block address\n");
                close(fd);
                return;
            }
            for (int k = 0; k < NINDIRECT; k++) {
                if (write(fd, buf, BSIZE) != BSIZE) {
                    printf(2, "Failed to write data block\n");
                    close(fd);
                    return;
                }
            }
        }
    }
    printf(2, "triple indirect block test passed\n");
    close(fd);
    printf(1, "Triple indirect test successful\n");
}
```

[Test 1의 실행 결과]

```
$ Tindirect_test
direct block test
direct block test passed
single indirect block test
single indirect block test passed
double indirect block test
double indirect block passed
triple indirect block test
triple indirect block test passed
Triple indirect test successful
```

[분석]

- `triple_indirect_test`를 통해서 triple_indirect 까지 잘 접근할 수 있는지를 확인한다.

⇒ 모두 정상적으로 실행되었다.

3-2. Test 2 - Save & Load Test

[Test 2의 소스 코드]

```c
void save(char* filename) {
    int fd;
    struct test t;
    for (int i = 0; i < SIZE; i++) t.file[i] = '1';
    fd = open(filename, O_CREATE | O_RDWR);
    if (fd >= 0) {
        printf(1, "Create Success\n");
    }
    else {
        printf(1, "Error: Create failed\n");
        exit();
    }

    int size = sizeof(t);
    printf(1, "[%d]", size);
    for (int i = 0; i < 1024; i++) {
        for (int j = 0; j < 16; j++) {
            if (write(fd, &t, size) != size) {
                printf(1, "Error: Write failed\n");
                exit();
            }
        }
    }
    printf(1, "write ok\n");
    close(fd);
}
```

```c
void load(char* filename) {
    int fd;
    struct test t;
    fd = open(filename, O_RDONLY);
    if (fd >= 0) {
        printf(1, "Open Success\n");
    }
    else {
        printf(1, "Error: open failed\n");
        exit();
    }

    int size = sizeof(t);
    if (read(fd, &t, size) != size) {
        printf(1, "Error: read failed\n");
        exit();
    }
    printf(1, "Read Success\n");
    close(fd);
}
```

[Test 2의 실행 결과]

```
$ SandL
now 0
Create Success
[1024]1 sleep  init 80104757 801047ff 8010521d 80106401 80106143
2 sleep  sh 80104757 801047ff 8010521d 80106401 80106143
4 run    SandL
write ok
Open Success
Read Success
now 1
Create Success
[1024]write ok
Open Success
Read Success
now 2
Create Success
[1024]write ok
Open Success
Read Success
now 3
Create Success
[1024]write ok
Open Success
Read Success
$
```

[분석]

- 파일 을 열고 저장하는 테스트이다.
- 성공적으로 열고 저장이 되었다.

3-3. Test 3 - Symbolic Link

[Test 3의 실행 결과]

```
$ ln -h ls hard
$ ln -s ls soft
$ hard
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16352
echo           2 4 15204
forktest       2 5 9512
grep           2 6 18572
init           2 7 15792
kill           2 8 15232
ln             2 9 15480
ls             2 10 17720
mkdir          2 11 15332
rm             2 12 15308
sh             2 13 27952
stressfs       2 14 16224
usertests      2 15 67552
wc             2 16 17084
zombie         2 17 14900
Tindirect_test 2 18 18000
SandL          2 19 18864
console        3 20 0
hard           2 10 17720
soft           4 21 2
```

[그림1]

```
$ soft
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16352
echo           2 4 15204
forktest       2 5 9512
grep           2 6 18572
init           2 7 15792
kill           2 8 15232
ln             2 9 15480
ls             2 10 17720
mkdir          2 11 15332
rm             2 12 15308
sh             2 13 27952
stressfs       2 14 16224
usertests      2 15 67552
wc             2 16 17084
zombie         2 17 14900
Tindirect_test 2 18 18000
SandL          2 19 18864
console        3 20 0
hard           2 10 17720
soft           4 21 2
```

[그림2]

```
$ rm ls
$ ls
exec: fail
exec ls failed
$ hard
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16352
echo           2 4 15204
forktest       2 5 9512
grep           2 6 18572
init           2 7 15792
kill           2 8 15232
ln             2 9 15480
mkdir          2 11 15332
rm             2 12 15308
sh             2 13 27952
stressfs       2 14 16224
usertests      2 15 67552
wc             2 16 17084
zombie         2 17 14900
Tindirect_test 2 18 18000
SandL          2 19 18864
console        3 20 0
hard           2 10 17720
soft           4 21 2
$ soft
exec soft failed
```

[그림3]

```
$ ln -s ls soft0
$ ln -s soft0 soft1
$ ln -s soft1 soft2
$ rm soft0
$ ln -s soft2 soft0
$ soft0
It might be Cyclic Symbolic File
exec soft0 failed
$ soft1
It might be Cyclic Symbolic File
exec soft1 failed
$ soft2
It might be Cyclic Symbolic File
exec soft2 failed
$
```

[그림4]

[분석]

- Symbolic Link가 정상적으로 작동하는지 확인하는 테스트이다.
- **[그림 1]**에서 `ls`의 Hard Link 파일이 정상적으로 만들어져서 실행되었다.
- **[그림 2]**에서 `ls`의 Symbolic Link 파일이 정상적으로 만들어져서 실행되었다.
- **[그림 3]**에서 `ls`를 삭제한 뒤에, Hard Link 파일은 정상적으로 작동하지만, Symbolic Link 파일은 작동하지 않음을 알 수 있다.
- **[그림4]**에서 Cyclic Symbolic에 대한 테스트를 볼 수 있다. 실행이 되지 않고 `exec`이 종료됨을 알 수 있다.

3-4. Test 4 - Close without Sync (from Han Seung-Woo)

- 위 테스트의 아이디어는 유민수 교수님 반에서 운영체제를 수강 중인 한승우 학생에게 제공 받았다.
- 이 부분은 테스트 출력을 보이기 어려워서 캡쳐를 따로 하지 않았다.
- 테스트의 목적은 `sync()` 혹은 `flush()`를 호출하지 않은 상태에서 종료된 경우에 Disk에 쓰여지지 않음을 확인하기 위함이다.
- 테스트 과정은 다음과 같다.
  1. `ln -s ls soft`와 같은 코드로 임의의 파일을 생성한다.
  2. 바로 xv6를 종료한다.
  3. 다시 xv6를 실행한다.
  4. xv6에서 `ls`를 통해서 `soft` 파일이 존재하지 않음을 확인한다.
  5. 만약 존재한다면, `sync()` 혹은 `flush()` 함수를 호출하지 않았음에도, disk에 쓰여진 경우이므로 잘못된 구현을 한 것이다.
  6. 만약 존재하지 않는다면 올바르게 구현한 것이다.
- 위와 같은 테스트에서 정확히 실행되었으므로, 올바른 구현을 한 것으로 보인다.

---

4. Trouble shooting

4-1. Symbolic Link `inode`

4-1-1. 정보를 저장할 위치

Symbolic Link란 위에서도 여러 번 설명했지만, 어떤 metadata를 갖거나 공유하는 것이 아니라 단순히 redirection을 해주는 것이다. redirection을 해주기 위해서는 Original File에 대한 `pathname`을 저장해야 한다. 그렇지만 `dinode`와 `inode`에 새로운 field를 추가할 수가 없다. 왜냐하면, `dinode`의 크기가 변하게 되면, system에 문제가 발생하기 때문이다. 이에 대해 해결하기 위해서는 `inode`에서 쓰지 않는 field를 이용해서 저장해야 한다.

저장해야 하는 정보는 `pathname`과 그 길이이다. `inode`의 field에서 쓰지 않는 field는 `major, minor, size, addrs` 등이 있다.

처음에는 모든 정보를 `addrs`에 저장하려고 했다. `pathname`의 길이는 `addrs[0]`에 저장하고, `pathname`은 `addrs[1]`부터 나머지부분에 저장하려고 했고 실제로도 잘 작동했다.

나중에 code refactoring을 하면서 사실 `pathname`의 길이를 `addrs[0]`에 저장할 필요가 없다는 것을 알았다. `size`에 저장을 해도 문제가 없고, 오히려 `pathname`을 길이 4만큼 더 저장할 수 있다.

4-1-2. Redirection

Symbolic Link를 redirection을 어떻게 할 수 있을까에 대해서 고민을 했다. 우선 `sys_open()`함수에서 하는 방법을 이용하면, `ls`에서 redirection을 하면 표시되는 정보가 Original File이기 때문에, 과제의 명세에 맞지 않게 된다. 이 부분을 해결하기 위한 방법이 `readi()`와 `writei()`에서 Symbolic Link File인 경우에 내부적으로 Original File을 찾아가는 방법을 이용했다. 이렇게 하면, `ls`에서는 Symbolic Link File에 대한 정보가 나오지만, 실제로 data를 읽고 쓸때는 Original File에 대해서 이루어지게 된다.

4-2. `sync() & flush() - bget()`

4-2-1. Cyclic Call

```
int is_flush;
int sync(void) {
  int n = log.lh.n;
  if (log.committing) {
    while (log.committing) {
      sleep(&log, &log.lock);
    }
    wakeup(&log);
    return 0;
  }

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

  // call commit w/o holding locks, since not allowed
  // to sleep with locks.
  commit();

  acquire(&log.lock);
  log.committing = 0;
  is_flush = 0;
  wakeup(&log);
  release(&log.lock);
  return n;
}
```

```
int flush(void) {
  int n = log.lh.n;
  if (log.committing) {
    wakeup(&log);
    return 0;
  }
  acquire(&log.lock);
  log.committing = 1;
  is_flush = 1;
  release(&log.lock);

  // call commit w/o holding locks, since not allowed
  // to sleep with locks.
  commit();
  acquire(&log.lock);
  log.committing = 0;
  is_flush = 0;
  wakeup(&log);
  release(&log.lock);
  return n;
}
```

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

  acquire(&bcache.lock);
  if (!is_flush) {
    int num_unused_buf = 0;
    // Check that is there enough buffer.
    for (b = bcache.head.prev; b != &bcache.head; b = b->prev) {
      if (b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
        num_unused_buf++;
      }
    }

    // If not enough, does flush.
    if (num_unused_buf <= 2) {
      release(&bcache.lock);
      flush();
      acquire(&bcache.lock);
    }
  }
```

`sync(), flush()`과 `bget()`의 관계는, `bget()`에서 `flush()`를 호출하면 `flush()`안의 `commit()`에서 다시 `bget()`을 호출하는 cycle 형태로 함수가 반복적으로 호출된다.

즉, `bget()`에서 buffer를 disk에 써주기 위해서 `flush()`를 호출하면, 또 다시 `bget()`이 호출되고 다시 `flush()`가 호출되는 구조가 반복된다.

이를 막기 위해서, `flush()`안에서 `log.committing`을 체크하여서 안에서 추가적으로 `commit()`이 호출되지 않도록했다.

또한 `is_flush` 변수를 통해서 sync()안에서 flush()가 반복적으로 호출되지 않도록 만들었다.

`is_flush` 변수는 `extern int` 변수로 `bio.c`와 `log.c`에서 사용되며 flush를 해야 하는지 확인하는 flag 변수이다.

이를 사용하는 이유는 만약 `is_flush`가 없다면, buffer가 가득찬 상태에서 `sync()`요청이 들어오게 되면, `commit()`에서 `bget()`을 부를 때 마다 안에서 `flush()`를 해야 하는지 확인하고 buffer가 이미 부족한 상태이기 때문에 `flush()`를 호출하는 과정이 생긴다. 이러한 문제를 해결하기 위해서 `is_flush`를 도입했다.

4-2-2. `flush()`를 위해서 필요한 buffer의 수

```
if (num_unused_buf <= 2) {
  is_flush = 1;
  release(&bcache.lock);
  flush();
  acquire(&bcache.lock);
  is_flush = 0;
}
```

`bget()`함수에서 `flush()`를 호출해야 하는지 아닌지에 대한 판단을 위해서 사용하지 않는buffer의 수가 2 이하이면, 호출하도록 했다.

왜냐하면, 위에서 말했듯이, `flush()`내의 `commit()`함수가 `bget()`함수를 호출하기 때문이다.

조금 더 자세히 설명하자면 `commit()`내에는 다음과 같은 함수들이 있다.

- `wirte_log()`
- `write_head()`
- `install_trans()`

위의 함수 내부에서는 각각 반복문 안에서 매 loop마다 `bread()`함수를 2번, 1번, 2번씩 호출하고, `brelse()`로 buffer를 놓아준다.

`bread()`에서는 내부적으로 `bget()` 을 호출한다.

따라서, 최소한 2개의 buffer는 존재해야 `commit()`을 할 수 있다.

4-2-3. log vs buffer

사실 처음에는 buffer와 log의 수가 같다고 생각을 하여, buffer의 수가 아니라, log의 수를 보고 `flush()`하는 구현을 고려했었다.

그렇지만, 이런 방향은 의미 측면에서 buffer가 부족한 경우에 호출하는 것이 아니기도 하고, `bio.c`에서 `log.c`로 이동해서 결과를 보고 flow를 결정한다는 기분나쁜 구현이기도 했다.

그렇기 때문에 실질적으로 buffer의 수와 log의 수가 동일하기 때문에 문제가 발생하지 않더라도, 구현의 방향을 바꾸는 것이 맞다고 생각했다.

4-3. With Friend

이번 과제는 시험기간이라서 시간도 부족하고, 이해도 부족한 상태에서 과제를 진행하다보니, 혼자서 진행하기는 어려웠다.

다행히도 같이 운영체제를 수강하는 박연진 학생과 같이 서로 도와주면서 과제를 진행하여서, 기존의 다른 과제들보다 훨씬 빠르게 진행할 수 있었다.

물론 서로의 코드를 보거나 공유하지는 않았고, 이 과제가 무엇을 요구하는 과제인지, 어떻게 구현하는 방향성이 올바른 것인지에 대해서 서로 도와주었다.

혼자서 코딩을 하는 것도 물론 중요하지만, 다른사람과 구현을 비교하면서 어떤 방향이 맞는 방향인지 고민하는 과정도 중요하다는 것을 알 수 있는 과제였다.