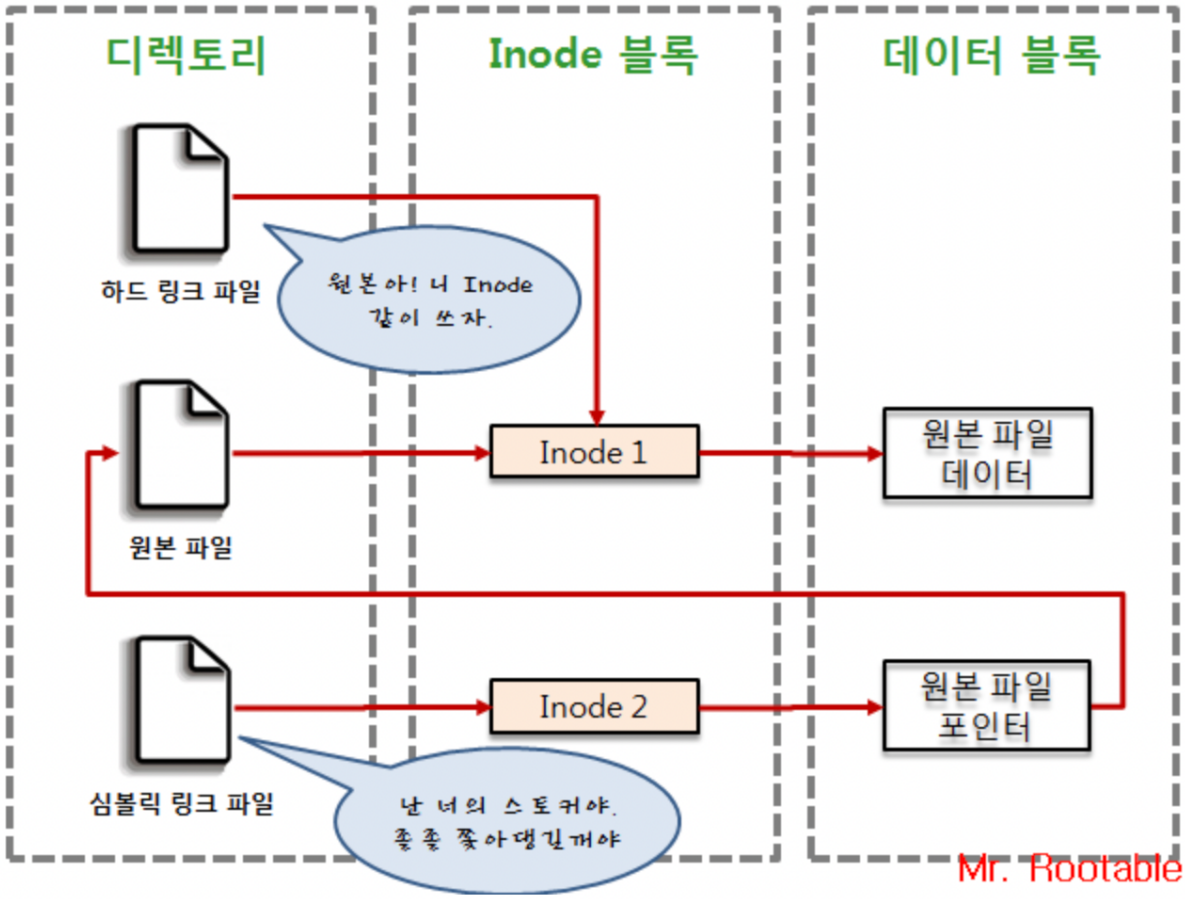
🎨 Design

1. Multi Indirect

이미 구현되어 있는 single indirect의 구현 내용을 분석하고 응용하여 구현하였습니다. single indirect는 dinode의 addrs 배열 마지막 요소를 사용하고 있는데, double, triple indirect를 위한 변수는 addrs 배열과 별개로 D_addr, T_addr 변수를 선언하여 사용하는 것으로 디자인 하였습니다.

2. Symbolic Link

hard link과 symbolic link의 차이점에 집중하여 구현하였습니다. 아래 하드링크과 심볼릭 링크의 디렉토리, inode, 데이터 블록 간의 관계를 설명하는 그림에서 힌트를 얻었습니다.



심볼릭 링크 파일은 별개의 inode를 갖는 파일로 선언하고, 원본 파일의 pathname을 inode에 저장하여 심볼링 링크 파일에 대한 open, read, write, close 작업이 있을 때 redirection 되도록 디자인하였습니다.

3. Buffered I/O

log.c의 begin_op와 end_op, commit 함수를 분석하여 group flush 의 동작 조건과 방식을 파악하고 sync 방식으로 수정하였습니다. sync는 로그 작성을 포함한 commit 과정으로, 기존에 end_op 아래쪽 do_commit으로 묶인 부분을 활용하였습니다. 과제 명세에 "버퍼에 공간이 부족한 경우" 강제로 sync를 발생시키도록 되어 있는데, writei에서 최대 MAXOPBLOCKS 만큼 묶어서 write 하는 부분을 그대로 유지하여, 로그 공간이 MAXOPBLOCKS 이하로 남았을 때 sync가 발생하도록

디자인하였습니다.

🛠️ Implement

1. Multi Indirect

먼저 dinode와 inode 구조체에 double indirect와 triple indirect의 주소를 담을 수 있는 변수를 추가 해 주었습니다.

```
12  // in-memory copy of an inode
13  struct inode {
14    uint dev;           // Device number
15    uint inum;          // Inode number
16    int ref;            // Reference count
17    struct sleeplock lock; // protects everything below here
18    int valid;          // inode has been read from disk?
19
20    short type;         // copy of disk inode
21    short major;
22    short minor;
23    short nlink;
24    uint size;
25    uint addrs[NDIRECT+1];
26    uint D_addr;          // Double index block
27    uint T_addr;          // Triple index block
28    char slink[16];       // slink path (original file path)
29  };
```

```
30  // On-disk inode structure
31  struct dinode {
32    short type;           // File type
33    short major;          // Major device number (T_DEV only)
34    short minor;          // Minor device number (T_DEV only)
35    short nlink;          // Number of links to inode in file system
36    uint size;            // Size of file (bytes)
37    uint addrs[NDIRECT+1];   // Data block addresses
38    uint D_addr;          // Double index block
39    uint T_addr;          // Triple index block
40    char slink[16];       // slink path (original file path)
41  };
```

D_addr, T_addr 변수를 통해 double indirect, triple indirect의 index block 주소를 저장합니다. 이때 mkfs.c에서 dinode 구조체의 크기에 대해 아래와 같은 제한을 두고 있기 때문에, 새로 할당한 변수에 대한 공간을 NDIRECT 값을 수정하여 마련하였습니다.

```
84    assert((BSIZE % sizeof(struct dinode)) == 0);
```

NDIRECT 값을 10으로 수정하였습니다. (cf. 이후 symbolic link의 구현에서 slink를 위한 공간을 할당하면서 16byte가 더 필요했고, 따라서 최종 과제물의 NDIRECT 값은 6으로 수정하였습니다.)

(1) bmap

inode에 대해 block을 로드하거나 할당하는 bmap 함수를 수정해야 합니다. double indirect는 direct, single indirect 공간을 모두 사용한 후 추가적인 공간을 할당하게 되므로, bn에서 NINDIRECT 값을 우선 제외하여 bn을 NDBLINDIRECT의 범위로 맞춥니다.

```
401    bn -= NINDIRECT;
```

D_addr가 가리키는 block에 128개씩 묶어 index를 저장하는 block의 주소를 써야 하므로, [bn / NINDIRECT] 연산을 하여 block을 할당합니다.

```
403    if(bn < NDBLINDIRECT){
404      if((addr = ip->D_addr) == 0) // first use of Double indirect
405        ip->D_addr = addr = balloc(ip->dev);
406      bp = bread(ip->dev, addr);
407      a = (uint*)bp->data;
408      if((addr = a[bn / NINDIRECT]) == 0){
409        a[bn / NINDIRECT] = addr = balloc(ip->dev);
410        log_write(bp);
411      }
412      brelse(bp);
```

마지막으로 실제 데이터 블록을 할당하여 파일 데이터가 쓰일 공간을 매핑합니다.

```
413     bp = bread(ip->dev, addr);
414     a = (uint*)bp->data;
415     if((addr = a[bn % NINDIRECT]) == 0){
416       a[bn % NINDIRECT] = addr = balloc(ip->dev);
417       log_write(bp);
418     }
419     brelse(bp);
420     return addr;
421   }
```

같은 매커니즘으로 triple indirect도 구현할 수 있습니다.

```
422   bn -= NDBLINDIRECT;
423
424   if(bn < NTRPLINDIRECT){
425     if((addr = ip->T_addr) == 0)
426       ip->T_addr = addr = balloc(ip->dev);
427     bp = bread(ip->dev, addr);
428     a = (uint*)bp->data;
429     if((addr = a[bn / NDBLINDIRECT]) == 0){
430       a[bn / NDBLINDIRECT] = addr = balloc(ip->dev);
431       log_write(bp);
432     }
433     brelse(bp);
434     bp = bread(ip->dev, addr);
435     a = (uint*)bp->data;
436     if((addr = a[bn / NINDIRECT]) == 0){
437       a[bn / NINDIRECT] = addr = balloc(ip->dev);
438       log_write(bp);
439     }
440     brelse(bp);
441     bp = bread(ip->dev, addr);
442     a = (uint*)bp->data;
443     if((addr = a[bn % NINDIRECT]) == 0){
444       a[bn % NINDIRECT] = addr = balloc(ip->dev);
445       log_write(bp);
446     }
447     brelse(bp);
```

triple indirect의 경우, T_addr에 128*128개씩의 indexing block 주소, 그 안에 각 128개씩의 data block 주소를 담을 수 있는 구조입니다. 따라서 T_addr에 할당 된 block에서 [bn / NDBLINDIRECT], 두번째 indexing block에서 [bn / NINDIRECT]로 인덱싱합니다. 마지막 data block은 double indirect와 마찬가지로 [bn % NINDIRECT]로 접근할 수 있습니다.

(3) param.h의 FSSIZE

최대 데이터 블록 제한을 높이기 위해 FSSIZE를 조정해야 합니다. multi indirect를 통해 할당 가능한 최대 블록 개수는 6+128+128*128+128*128*128 = 2113670 이므로, FSSIZE를 2200000로 조정하였습니다.

(2) itrunc

inode에 대한 사용이 끝났을 때 block 할당을 해제하는 itrunc 함수에도 동일하게 double indirect, triple indirect에 대한 처리를 해 주어야 합니다. single indirect를 참고하여 bfree에 대한 for loop

을 중첩시켜 구현하였습니다.

```
485     if(ip->D_addr){
486       bp = bread(ip->dev, ip->D_addr);
487       a = (uint*)bp->data;
488       for(j = 0; j < NINDIRECT; j++){
489         if(a[j]){
490           bp2 = bread(ip->dev, a[j]);
491           b = (uint*)bp2->data;
492           for(k = 0; k < NINDIRECT; k++){
493             if(b[k])
494               bfree(ip->dev, b[k]);
495           }
496           brelse(bp2);
497           bfree(ip->dev, a[j]);
498         }
499       }
500       brelse(bp);
501       bfree(ip->dev, ip->D_addr);
502       ip->D_addr = 0;
503     }
```

Double indirect는 for loop을 2번 중첩시켜 구현할 수 있습니다.

```
505     if(ip->T_addr){
506       bp = bread(ip->dev, ip->D_addr);
507       a = (uint*)bp->data;
508       for(j = 0; j < NINDIRECT; j++){
509         if(a[j]){
510           bp2 = bread(ip->dev, a[j]);
511           b = (uint*)bp2->data;
512           for(k = 0; k < NINDIRECT; k++){
513             if(b[k]){
514               bp3 = bread(ip->dev, b[k]);
515               c = (uint*)bp3->data;
516               for(l = 0; l < NINDIRECT; l++){
517                 if(c[l])
518                   bfree(ip->dev, c[l]);
519               }
520               brelse(bp3);
521               bfree(ip->dev, b[k]);
522             }
523           }
524           brelse(bp2);
525           bfree(ip->dev, a[j]);
526         }
527       }
528       brelse(bp);
529       bfree(ip->dev, ip->D_addr);
530       ip->T_addr = 0;
531     }
```

Triple indirect는 for loop을 3번 중첩시켜 구현할 수 있습니다.

2. Symbolic Link

(1) ln.c

-s, -h 옵션으로 symbolic link와 hard link를 구분하여 생성할 수 있도록 ln.c의 메인 함수를 수정해 주었습니다.

```c
int
main(int argc, char *argv[])
{
  if(argc != 4 || !(strcmp(argv[1], "-h") || strcmp(argv[1], "-s"))){
    printf(2, "Usage: ln [ -s | -h ] old new\n");
    exit();
  }
  if(!strcmp(argv[1], "-h")){
    if(link(argv[2], argv[3]) < 0) // -h, -s 인자 하나 더 받기
      printf(2, "hard link %s %s: failed\n", argv[2], argv[3]);
  }
  else{
    if(slink(argv[2], argv[3]) < 0)
      printf(2, "symbolic link %s %s: failed\n", argv[2], argv[3]);
  }
  sync();
  exit();
}

```

(2) file.h, fs.h의 inode, dinode 구조체

symbolic link 파일은 파일에 접근했을 때 link되어 있는 기존 파일로 redirection 되는 파일입니다. redirection은 pathname을 이용하여 포인팅 되도록 구현하였는데, 이를 위해 dinode와 inode 구조체에 link 된 파일 path를 저장할 변수를 할당하였습니다.

```c
// in-memory copy of an inode
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];
  uint D_addr;          // Double index block
  uint T_addr;          // Triple index block
  char slink[16];       // slink path (original file path)
};
```

```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];   // Data block addresses
  uint D_addr;          // Double index block
  uint T_addr;          // Triple index block
  char slink[16];       // slink path (original file path)
};
```

slink[16] 배열에 link할 파일 이름을 저장합니다. DIRSIZ가 14이고, dinode의 size 제한을 지키기 위해 size를 16으로 설정한 후 NDIRECT 값을 6으로 조정하였습니다. (4*4byte를 slink로 양보함)

(3) sysfile.c / sys_slink(void)

ln -s [old] [new]로 link를 생성했을 때, inode를 찾아 공유하는 hard link와 달리, 새로운 파일을 생성해야 합니다. 따라서 sys_slink에서는 새로운 inode를 할당하였습니다.

```
509  int
510  sys_slink(void)
511  {
512    char *new, *old;
513    struct inode *ip;
514
515    if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
516      return -1;
517
518    begin_op();
519    if((ip = create(new, T_SYM, 0, 0)) == 0)
520      goto bad;
521    safestrcpy(ip->slink, old, 16);
522    iupdate(ip);
523    iunlock(ip);
524    end_op();
525
526    return 0;
527
528  bad:
529    end_op();
530    return -1;
531  }
```

create 함수를 통해 새로운 inode를 생성하고, redirection 정보를 ip->slink에 저장합니다.

(4) fs.c / namex(char *path, int nameiparent, char *name, int redirection)

pathname으로 ip를 찾는 namex 함수가 symbolic link 파일에 대해 redirection 될 수 있도록 재귀적으로 구현 해 주었습니다. namex에 마지막으로 추가한 int redirection 변수는 redirection을 실행할지 여부를 받는 flag입니다. ls 명령어로 파일 목록을 읽는 경우 redirection 없이 파일 정보를 그대로 출력해야 하므로, redirection 변수를 0으로 설정 해 주었습니다.

```
756      ilock(next);
757      if(redirection && next->type == T_SYM){
758        iunlockput(next);
759        next = namex(next->slink, nameiparent, name, redirection);
760      }
761      else
762        iunlock(next);
```

path를 옮겨 가며 ip를 찾는 룹 안에서, next의 타입이 T_SYM인 경우 symbolic link 파일을 의미하므로, next inode의 slink 변수에 있는 path로 다시 namex 함수를 호출하여 next 값을 바꿔 줍니다.

3. Buffered I/O

(1) log.c / end_op(void)

buffered I/O를 구현하기 위해 먼저 end_op 함수를 수정하였습니다. 기존 end_op는 동시에 실행되는 write 연산에 대해 log.outstanding으로 표시한 후, 이 값이 0이 되어 모든 write 연산이 종료되었을 때 group commit을 수행합니다. buffered I/O는 이 과정을 "버퍼가 가득 찼을 때"의 조건으로 수행해야 하므로, do_commit 변수를 1로 바꾸는 조건을 아래와 같이 수정하여 end_op를 구현하였습니다.

```
143  // called at the end of each FS system call.
144  // commits if this was the last outstanding operation.
145  void
146  end_op(void)
147  {
148    int do_commit = 0;
149
150    acquire(&log.lock);
151    log.outstanding -= 1;
152    if(log.committing)
153      panic("log.committing");
154    if(log.outstanding != 0){
155      // begin_op() may be waiting for log space,
156      // and decrementing log.outstanding has decreased
157      // the amount of reserved space.
158      wakeup(&log);
159    }
160    else if(log.lh.n + MAXOPBLOCKS > LOGSIZE){
161      #ifdef DEBUG
162      cprintf("no buffer (log space) -> go commit");
163      #endif
164      do_commit = 1;
165      log.committing = 1;
166    }
167    release(&log.lock);
168
169    if(do_commit){
170      sync();
171    }
172  }
```

log.lh.n + MAXOPBLOCKS > LOGSIZE 조건이 통과되는 경우, 실제 bcache의 모든 block의 dirty bit가 on인, 더 이상 버퍼에 블럭을 올릴 수 없는 상태는 아닙니다. 하지만 writei에서 최대 MAXOPBLOCKS 개수만큼 묶어서 write 연산을 수행하고 있으므로, 하나의 writei 연산을 더 허용했을 때 버퍼 공간이 넘칠 위험이 있는 경우 sync를 호출하여 flush 해 주어야 합니다.

(2) log.c / sync(void)

```
238  int
239  sync(void)
240  {
241    // call commit w/o holding locks, since not allowed
242    // to sleep with locks.
243    int nblock;
244
245    if (log.committing == 0)
246      log.committing = 1;
247    nblock = commit();
248    acquire(&log.lock);
249    log.committing = 0;
250    wakeup(&log);
251    release(&log.lock);
252    return nblock;
253  }
```

end_op에서 do_commit == 1일 때 수행되는 부분이 disk I/O가 발생하며 버퍼를 flush 하는 과정이므로, sync 함수로 옮겨 구현할 수 있습니다.

(3) file write가 발생하는 user 프로그램 (쉘 프로그램)에 대한 수정

group commit 과정이 삭제되면서, 기존의 쉘 프로그램 중 file write가 필요한 프로그램에도 sync() 시스템 콜을 추가 해 주어야 재부팅 시 프로그램 수행 결과가 누락되지 않습니다. 따라서 ln, rm, mkdir 프로그램 종료 직전에 sync() 시스템 콜을 추가하여 파일 생성과 삭제 작업을 디스크와 동기화하도록 하였습니다.

🌈 Result (Test)

컴파일, 실행 과정에서 특별히 고려해야 할 사항은 없습니다.

make clean; make; make fs.img 로 컴파일합니다.

qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512 로 xv6를 실행합니다.

1. Multi Indirect

테스트 파일: file_test.c

과제 구현 후 각 indexing 방식에 따라 할당하는 블록 개수와 최대 용량은 다음과 같습니다.

indexing	block 개수	최대 용량
direct	6	3072B
single indirect	128	약 68KB
double indirect	128*128 = 16384	약 8MB
triple indirect	128*128*128 = 2097152	약 1GB

single indirect , double indirect, triple indirect의 세 범위에 해당하는 bytes로 테스트를 진행하였습니다.

(1) single indirect 범위, 약 65KB

```
$ file_test
Test 1: Write 65536 bytes
Test 1 passed
```

(2) double indirect 범위, 약 5MB

```
Test 2: Write 5185536 bytes
Test 2 passed
```

(3) triple indirect 범위, 약 13.5MB

```
Test 3: Write 13508608 bytes
Test 3 passed
```

write 작업에 대해 정상 동작함을 확인할 수 있습니다.

테스트 실행 후 파일 read, open, close에 대해서 ls, cat 프로그램을 실행하여 확인할 수 있습니다.

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15540
echo           2 4 14416
forktest       2 5 8860
grep           2 6 18376
init           2 7 15040
kill           2 8 14504
ln             2 9 14764
ls             2 10 16972
mkdir          2 11 14580
rm             2 12 14560
sh             2 13 28556
stressfs       2 14 15436
wc             2 15 15956
zombie         2 16 14080
file_test      2 17 17200
sync_test      2 18 16320
console        3 19 0
test.txt       2 20 65536
test2.txt      2 21 5185536
test3.txt      2 22 13508608
```

```
$ cat test.txt
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
$ cat test2.txt
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
$ cat test3.txt
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

2. Symbolic Link

쉘에서 ln -s [old] [new] 명령어를 입력하여 실행할 수 있습니다.

```
$ echo test > a
$ ln -s a b
$ cat b
test
```

파일 a에 대해 symbolic link 파일 b를 생성하면, b에 대한 read 작업 시 a 로 redirection 됩니다.

```
$ echo symbolic > b
$ cat b
symbolic
$ cat a
symbolic
```

write 작업에 대해서도 정상적으로 redirection 됨을 알 수 있습니다.

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15540
echo           2 4 14416
forktest       2 5 8860
grep           2 6 18376
init           2 7 15040
kill           2 8 14504
ln             2 9 14764
ls             2 10 16972
mkdir          2 11 14580
rm             2 12 14560
sh             2 13 28556
stressfs       2 14 15436
wc             2 15 15956
zombie         2 16 14080
file_test      2 17 17200
sync_test      2 18 16320
console        3 19 0
a              2 20 9
b              4 21 0
```

inode를 공유하는 hard link와 달리, symbolic link 파일은 별개의 inode를 가지므로, a와 b의 ls 명령어 결과가 다릅니다. 제 구현에 따르면, symbolic link 파일은 inode에 original file path 정보를 갖는 내용이 없는 파일입니다.

```
$ rm a
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15540
echo           2 4 14416
forktest       2 5 8860
grep           2 6 18376
init           2 7 15040
kill           2 8 14504
ln             2 9 14764
ls             2 10 16972
mkdir          2 11 14580
rm             2 12 14560
sh             2 13 28556
stressfs       2 14 15436
wc             2 15 15956
zombie         2 16 14080
file_test      2 17 17200
sync_test      2 18 16320
console        3 19 0
b              4 21 0
$ cat b
cat: cannot open b
```

original 파일인 a를 삭제하고 나면, b 파일은 삭제되지 않지만, redirection 할 파일이 삭제되었으므로 open, read, write, close 작업을 할 수 없습니다. cat b에 대해 파일 b를 open 할 수 없어 에러가 발생했음을 확인할 수 있습니다.

```
$ rm b
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15540
echo           2 4 14416
forktest       2 5 8860
grep           2 6 18376
init           2 7 15040
kill           2 8 14504
ln             2 9 14764
ls             2 10 16972
mkdir          2 11 14580
rm             2 12 14560
sh             2 13 28556
stressfs       2 14 15436
wc             2 15 15956
zombie         2 16 14080
file_test      2 17 17200
sync_test      2 18 16320
console        3 19 0
```

다만 rm b로 파일을 삭제하는 것은 정상적으로 동작하는데, 이는 파일 삭제를 수행하는 unlick 시스템콜이 dirlookup 내부의 iget을 이용하여 inode 값을 가져와 삭제하도록 구현되어 있기 때문입니다. (namex 호출을 이용하지 않으므로 redirection 없음)

3. Buffered I/O

테스트 파일: sync_test.c

버퍼 최대 용량보다 작은 5120 bytes의 파일을 쓴 후, sync 시스템 콜을 호출한 경우와 호출하지 않은 경우를 비교합니다.

```
$ sync_test
Test 1: Write 5120 bytes with sync
Test 1 file write done

Test 2: Write 5120 bytes without sync
Test 2 file write done

please reboot xv6 and check test2.txt
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15540
echo           2 4 14416
forktest       2 5 8860
grep           2 6 18376
init           2 7 15040
kill           2 8 14504
ln             2 9 14764
ls             2 10 16972
mkdir          2 11 14580
rm             2 12 14560
sh             2 13 28556
stressfs       2 14 15436
wc             2 15 15956
zombie         2 16 14080
file_test      2 17 17200
sync_test      2 18 16320
console        3 19 0
test.txt       2 20 5120
test2.txt      2 22 5120
```

```
$ cat test.txt
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrs
$ cat test2.txt
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrs
```

Test 1에서는 sync를 호출하고, Test 2에서는 sync를 호출하지 않습니다. sync_test를 실행한 후 xv6를 재부팅하지 않으면, buffer cache에 test2.txt 파일에 write한 내용이 남아 있으므로 test2.txt 파일이 존재하고 write 내용이 반영되어 있음을 확인할 수 있습니다.

이제 xv6를 종료한 후 재부팅하면,

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 2200000 nblocks 2199404 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15540
echo           2 4 14416
forktest       2 5 8860
grep           2 6 18376
init           2 7 15040
kill           2 8 14504
ln             2 9 14764
ls             2 10 16972
mkdir          2 11 14580
rm             2 12 14560
sh             2 13 28556
stressfs       2 14 15436
wc             2 15 15956
zombie         2 16 14080
file_test      2 17 17200
sync_test      2 18 16320
console        3 19 0
test.txt       2 20 5120
$
```

sync를 호출하지 않고 close 한 Test 2 의 test2.txt 파일 정보는 disk에 저장되지 않았음을 알 수 있습니다.

🤯 Trouble shooting

이번 과제에서는 symbolic link를 구현하는 과정에서 namex를 구현하는 데에 어려움을 겪었습니다. 처음 namex에 대한 디자인을 생각할 때, ilock, iunlock, iput 함수에 대한 이해가 없었고, next->type == T_SYM 인 inode에 대해 아래처럼 redirection 된 inode를 리턴하도록 구현하였습니다.

```
757     if(redirection && next->type == T_SYM){
758       return namex(next->slink, nameiparent, name, redirection);
759     }
```

이렇게 구현하고 나면 ip 변수에 대한 iput(ip) 가 누락되는 문제가 있습니다. 때문에 xv6 에서 symbolic link 파일의 namex 호출 이후 ip를 잘못 읽는 문제가 발생하는 듯 했습니다. 아래처럼 쉘에서 exec를 사용하는 프로그램을 수행했을 때, 오류가 발생합니다.

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15540
echo           2 4 14416
forktest       2 5 8860
grep           2 6 18376
init           2 7 15040
kill           2 8 14504
ln             2 9 14764
ls             2 10 16972
mkdir          2 11 14580
rm             2 12 14560
sh             2 13 28556
stressfs       2 14 15436
wc             2 15 15956
zombie         2 16 14080
file_test      2 17 17200
sync_test      2 18 16320
console        3 19 0
a              2 20 4
b              4 22 0
$ cat a
exec cat failed
$ cat b
exec cat failed
```

수정한 구조는 아래와 같습니다.

```
756       ilock(next);
757       if(redirection && next->type == T_SYM){
758         iunlockput(next);
759         next = namex(next->slink, nameiparent, name, redirection);
760       }
761       else
762         iunlock(next);
763       iput(ip);
764       ip = next;
765     }
766     if(nameiparent){
767       iput(ip);
768       return 0;
769     }
770     return ip;
771   }
```

next가 가리키는 파일에 대해 symbolic link 여부를 판단하고 redirection하는 일련의 과정 처음과 끝에 ilock, iunlock을 추가합니다. symbolic link임이 확인되면, redirection 하기 전에, next에 대한 lock을 풀고 reference를 해제한 후 새롭게 파일을 찾을 수 있도록 iunlockput(next)를 호출합니다. 재귀함수 호출이 마무리되고 나면 아래쪽 루틴이 실행되면서 iput(ip)가 실행되고, in-memory inode에 대한 누수 없이 namex() 작업을 마무리하게 됩니다.