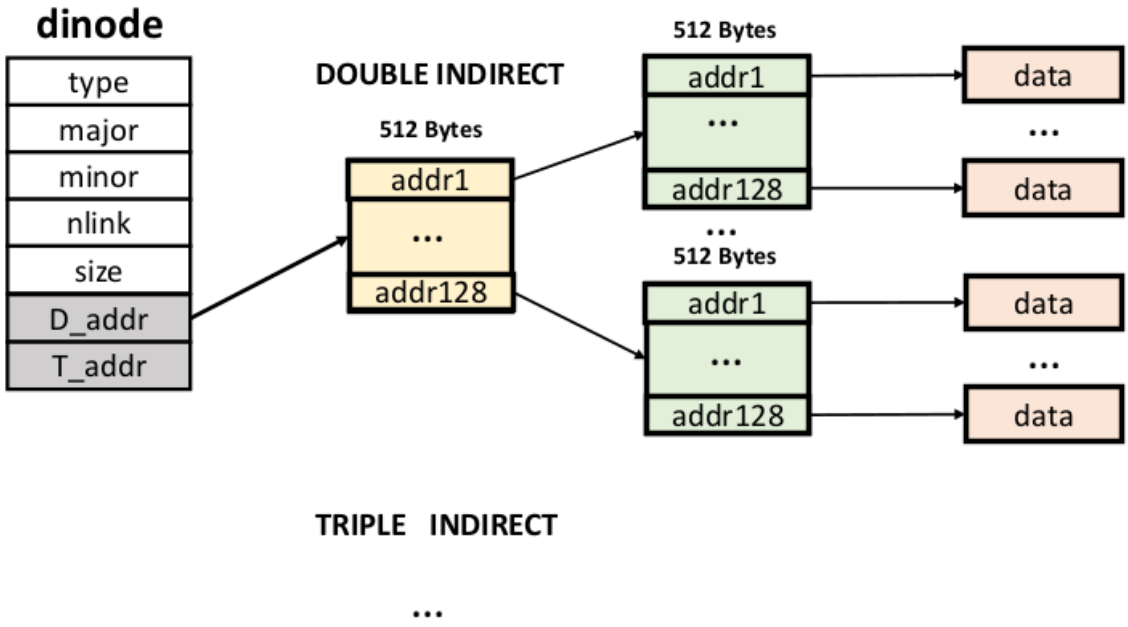
Proj3_Wiki

1. Multi Indirect 구현



변수 설정

```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define DINDIRECT (NINDIRECT * NINDIRECT)
#define TINDIRECT (NINDIRECT * NINDIRECT * NINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT + DINDIRECT+ TINDIRECT)

// On-disk inode structure
struct dinode {
  ...

  ...
  uint addrs[NDIRECT+3];   // Data block addresses
};
```

- `fs.h`

`NDIRECT` : direct로 접근하는 블록수

`NINDIRECT` : indirect로 접근하는 블록수(512 / 4 = 128개)

`DINDIRECT` : double indirect로 접근하는 블록수 (128 * 128 개)

`TINDIRECT` : double indirect로 접근하는 블록수 (128 * 128 * 128 개)

`MAXFILE` : xv6가 받을 수 있는 최대 파일 크기

`addrs` : 기존에서 double, triple indirect 구현을 위한 포인터 2개 추가 `[NDIRECT+3]`

```
// in-memory copy of an inode
struct inode {
  ...

  ...
  uint addrs[NDIRECT+3];
};
```

- `file.h`

`addrs` : 기존에서 double, triple indirect 구현을 위한 포인터 2개 추가 `[NDIRECT+3]`

```
#define FSSIZE       3,000,000  // size of file system in blocks
```

- `param.h`

`FSSIZE` : 받아들일 수 있는 최대 블록 수 (1000 → 3,000,000)

`bmap` 함수 수정

```
static uint bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;

  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0)
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
```

```
    if((addr = a[bn]) == 0){
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }
```

`fs.c` 파일에 있는 `bmap` 함수는 파일 시스템에서 블록 번호에 해당하는 데이터 블록을 찾는 역할을 합니다. `bmap` 함수는 주어진 파일의 블록 번호에 해당하는 데이터 블록이 있는지 확인하고, 데이터 블록이 존재하지 않는 경우 새로운 데이터 블록을 할당합니다.

`bmap` 함수의 동작은 다음과 같습니다:

1. `bn` 번째 블록 번호에 해당하는 데이터 블록의 번호를 계산합니다. 이는 데이터 블록이 direct, single indirect 블록 중 어디에 위치하는지에 따라 다르게 계산됩니다.
2. `bn` 번째 데이터 블록을 찾습니다. 이를 위해 먼저 `bn` 번째 direct 블록을 확인하고, 존재하지 않으면 single indirect 블록을 확인합니다. 데이터 블록이 존재하지 않는 경우 새로운 데이터 블록을 할당합니다.
3. `bn` 번째 데이터 블록의 번호를 반환합니다.

- Double Indirect 구현

```
bn -= NINDIRECT;

// double indirect
if (bn < DINDIRECT) {
  // 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);
    a = (uint*)bp->data;

    uint idx = bn / DINDIRECT;  // double indirect 블록 내에서의 인덱스
    uint idx2 = bn % DINDIRECT; // single indirect 블록 내에서의 인덱스

    if ((addr = a[idx]) == 0) {
      a[idx] = addr = balloc(ip->dev);
      log_write(bp);
    }

    brelse(bp);

    // Load single indirect block, allocating if necessary.
    bp = bread(ip->dev, addr);
    a = (uint *)bp->data;

    if ((addr = a[idx2]) == 0) {
```

```
      a[idx2] = addr = balloc(ip->dev);
      log_write(bp);
    }

    brelse(bp);
    return addr;
  }
```

우선 single indirect 블록을 넘을 경우 `bn -= NINDIRECT` 을 해준 뒤 double indirect을 진행하게 됩니다.

double indirect의 경우, single indirect 블록 내에서 double indirect 블록을 찾고, 그 안에서 다시 single indirect 블록을 찾아 해당 데이터 블록을 할당합니다.

이 때, double indirect안의 블록 내의 정확한 위치를 찾기 위해서, `DINDIRECT`로 나눈 몫과 나머지를 인덱스를 나타내는 변수로 활용합니다.

- Triple Indirect 구현

```
bn -= DINDIRECT

// triple indirect
  if (bn < NTINDIRECT) {
  // 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);
    a = (uint*)bp->data;

    uint idx = offset / TINDIRECT;       // triple indirect 블록 내에서의 인덱스
    uint idx2 = (offset % TINDIRECT) / DINDIRECT;  // double indirect 블록 내에서의 인덱스
    uint idx3 = (offset % TINDIRECT) % DINDIRECT; // single indirect 블록 내에서의 인덱스

    if ((addr = a[idx]) == 0) {
      a[idx] = addr = balloc(ip->dev);
      log_write(bp);
    }

    brelse(bp);

    // Load double indirect block, allocating if necessary.
    bp = bread(ip->dev, addr);
    a = (uint *)bp->data;

    if ((addr = a[idx2]) == 0) {
      a[idx2] = addr = balloc(ip->dev);
      log_write(bp);
    }

    brelse(bp);
```

```
    // Load single indirect block, allocating if necessary.
    bp = bread(ip->dev, addr);
    a = (uint *)bp->data;

    if ((addr = a[idx3]) == 0) {
      a[idx3] = addr = balloc(ip->dev);
      log_write(bp);
    }

    brelse(bp);

    return addr;
  }
```

double indirect와 비슷하게 `bn -= DINDIRECT` 을 해준 뒤 triple indirect을 진행하게 됩니다.

triple indirect의 경우, triple indirect 블록 내에서 double indirect 블록을 찾고, 그 안에서 single indirect 블록을 찾아 데이터 블록을 할당합니다.

이 때, triple indirect안의 블록 내의 정확한 위치를 찾기 위해서, `TINDIRECT`로 나눈 몫과 나머지를 인덱스를 나타내는 변수로 활용합니다.

`itrunc` 함수 수정

```
static void
itrunc(struct inode *ip)
{
  int i, j;
  struct buf *bp;
  uint *a;

  for(i = 0; i < NDIRECT; i++){
    if(ip->addrs[i]){
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }

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

  ip->size = 0;
```

```
  iupdate(ip);
}
```

`itrunc` 함수는 inode의 direct block 및 indirect block을 순회하면서 각 블록을 해제하고, inode의 크기를 0으로 설정한 후 변경 사항을 디스크에 반영합니다. 이렇게 함으로써 inode의 내용이 삭제되고 해당 공간이 해제됩니다.

1. `itrunc` 함수의 파라미터로 주어진 `struct inode *ip`는 truncate를 수행할 inode를 나타냅니다.
2. `for` 루프를 사용하여 inode의 `NDIRECT` 개의 direct block을 검사합니다. 각 direct block은 `addrs` 배열에 저장되어 있습니다.
   - `ip->addrs[i]`가 0이 아닌 경우, 해당 direct block을 해제합니다.
   - `ip->addrs[i]`를 0으로 설정하여 해당 direct block이 해제되었음을 나타냅니다.
3. `ip->addrs[NDIRECT]`에 indirect block이 저장되어 있는 경우(=NDIRECT 번째 direct block), 해당 indirect block을 검사합니다.
   - `bp = bread(ip->dev, ip->addrs[NDIRECT])`를 사용하여 indirect block을 읽어옵니다.
   - `a = (uint*)bp->data`를 사용하여 indirect block의 데이터를 가져옵니다.
   - `for` 루프를 사용하여 indirect block 내의 모든 indirect block entry를 검사합니다.
     - `a[j]`가 0이 아닌 경우, 해당 indirect block entry를 해제합니다.
   - `brelse(bp)`를 사용하여 indirect block의 버퍼를 해제합니다.
   - `bfree(ip->dev, ip->addrs[NDIRECT])`를 사용하여 indirect block을 해제합니다.
   - `ip->addrs[NDIRECT]`를 0으로 설정하여 indirect block이 해제되었음을 나타냅니다.
4. `ip->size`를 0으로 설정하여 inode의 크기를 0으로 만듭니다.
5. `iupdate` 함수를 호출하여 inode의 변경 사항을 디스크에 업데이트합니다.

- double, triple indirect 구현 시 삭제를 위한 수정

```
// for double indirect
  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]) {
        bp = bread(ip->dev, a[i]);
        b = (uint *)bp->data;
        for (j = 0; j < NINDIRECT; j++) {
```

```
          if (b[j])
            bfree(ip->dev, b[j]);
        }
        brelse(bp);
        bfree(ip->dev, a[i]);
      }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT+1]);
    ip->addrs[NDIRECT+1] = 0;
  }

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

추가적인 변수 `k` 와 `*b, *c`를 도입하여 루프를 제어합니다.

`NDIRECT` 이후의 다음 블록 `ip->addrs[NDIRECT + 1]`을 처리하기 위해 해당 부분의 코드를 추가하여 double indirect block을 처리합니다.

`ip->addrs[NDIRECT + 2]`을 처리하기 위해 해당 부분의 코드를 추가하여 triple indirect block을 처리합니다.

이중 루프를 사용하여 double indirect block과 triple indirect block의 각 entry를 순회하고 해제합니다.

각 block을 해제하기 전에 해당 block을 읽어오고, 해제한 후에는 `brelse`를 사용하여 해당 block의 버퍼를 해제합니다.

2. Symbolic Link 구현

- `ln.c` 파일 수정

```
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[])
{
  if(argc != 3){
    printf(2, "Usage: ln old new\n");
    exit();
  }
  if(link(argv[1], argv[2]) < 0)
    printf(2, "link %s %s: failed\n", argv[1], argv[2]);
  exit();
}
```

`ln.c` 파일의 `main` 함수는 다음과 같이 작동합니다.

1. `main` 함수는 두 개의 매개 변수 `argc`와 `argv[]`를 받습니다. `argc`는 명령행 인수의 개수를 나타내며, `argv[]`는 인수의 배열입니다.
2. `if(argc != 3)`는 명령행 인수의 개수가 3이 아닌 경우, 즉 `ln` 명령의 사용법에 맞지 않는 경우 오류 메시지를 출력하고 프로그램을 종료합니다.
3. `printf(2, "Usage: ln old new\n")`는 오류 메시지를 표준 에러 출력(`2`)으로 출력합니다. 메시지는 "Usage: ln old new"입니다.
4. `exit()` 함수를 호출하여 프로그램을 종료합니다.
5. `link(argv[1], argv[2])`는 `link` 시스템 호출을 사용하여 두 경로 간에 Hard 링크를 생성합니다. `argv[1]`은 원본 파일의 경로이고, `argv[2]`는 새 링크 파일의 경로입니다.
6. `if(link(argv[1], argv[2]) < 0)`는 링크 생성에 실패한 경우, 즉 `1`이 반환된 경우 오류 메시지를 출력합니다.
7. `printf(2, "link %s %s: failed\n", argv[1], argv[2])`는 오류 메시지를 표준 에러 출력으로 출력합니다. 메시지는 "link [원본 파일 경로] [새 링크 파일 경로]: failed" 형식입니다. `argv[1]`은 원본 파일의 경로이고, `argv[2]`는 새 링크 파일의 경로입니다.
8. `exit()` 함수를 호출하여 프로그램을 종료합니다.

Hard링크와 Symbolic 링크를 모두 생성할 수 있도록 수정합니다.

```
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[])
{
  if(argc != 4){ //
    printf(2, "Usage: ln [-s] or [-h] <old> <new>\n");
    exit();
  }
  int symbolic = 0;
  int opt;

  // -s 혹은 -h의 명령어인지 확인
  while ((opt = getopt(argc, argv, "sh")) != -1) {
    switch (opt) {
      case 's':
        symbolic = 1;
        break;
      case 'h':
        symbolic = 0;
        break;
      default:
        printf(2, "Unknown option: %c\n", opt);
        exit();
    }
  }

  if (symbolic) {
    if (symlink(argv[3], argv[4]) < 0)
      printf(2, "symlink %s %s: failed\n", argv[optind], argv[optind + 1]);
  } else {
    if (link(argv[3], argv[4]) < 0)
      printf(2, "link %s %s: failed\n", argv[optind], argv[optind + 1]);
  }
}
```

1. `if(argc != 4)`는 명령행 인수의 개수가 4가 아닌 경우, 즉 `ln` 명령의 사용법에 맞지 않는 경우 오류 메시지를 출력하고 프로그램을 종료합니다. 이 경우, 사용법 메시지는 "Usage: ln [-s] or [-h] <old> <new>"입니다.
2. `int symbolic = 0;`는 심볼릭 링크를 생성할 것인지를 나타내는 변수입니다. 초기값은 0으로 설정되어 하드 링크를 생성하도록 기본 설정됩니다.
3. `int opt;`는 `getopt` 함수를 사용하여 명령행 인수에서 옵션을 읽어오기 위한 변수입니다.

4. `while ((opt = getopt(argc, argv, "sh")) != -1)`는 `getopt` 함수를 사용하여 명령행 인수에서 옵션을 읽어오는 루프입니다. `s` 또는 `h` 옵션이 지정된 경우에는 해당하는 심볼릭 링크 생성 여부를 `symbolic` 변수에 설정합니다. 알 수 없는 옵션이 있는 경우 오류 메시지를 출력하고 프로그램을 종료합니다.

5. `if (symbolic)`은 `symbolic` 변수가 1로 설정된 경우, 즉 심볼릭 링크를 생성해야 하는 경우입니다.

6. `if (symlink(argv[3], argv[4]) < 0)`는 `symlink` 시스템 호출을 사용하여 심볼릭 링크를 생성합니다. `argv[3]`은 원본 파일의 경로이고, `argv[4]`는 새 심볼릭 링크 파일의 경로입니다. 심볼릭 링크 생성에 실패한 경우 오류 메시지를 출력합니다.

7. `printf(2, "symlink %s %s: failed\n", argv[optind], argv[optind + 1])`은 오류 메시지를 표준 에러 출력으로 출력합니다. 메시지는 "symlink [원본 파일 경로] [새 심볼릭 링크 파일 경로]: failed" 형식입니다.

8. `else`는 `symbolic` 변수가 0인 경우, 즉 하드 링크를 생성해야 하는 경우입니다.

9. `if (link(argv[3], argv[4]) < 0)`는 `link` 시스템 호출을 사용하여 하드 링크를 생성합니다. `argv[3]`은 원본 파일의 경로이고, `argv[4]`는 새 링크 파일의 경로입니다. 하드 링크 생성에 실패한 경우 오류 메시지를 출력합니다.

10. `printf(2, "hardlink %s %s: failed\n", argv[optind], argv[optind + 1])`은 오류 메시지를 표준 에러 출력으로 출력합니다. 메시지는 "hardlink [원본 파일 경로] [새 링크 파일 경로]: failed" 형식입니다.

이를 통해 주어진 코드에서는 `-s` 옵션이 지정되면 심볼릭 링크를 생성하고, 그렇지 않은 경우에는 하드 링크를 생성하는 기능을 구현하고 있습니다.

- `sys_symlink` system call 추가

In `sysfile.c` ,

기존의 hard 링크를 생서했던 `sys_link` 과 더불어 symbolic 링크를 생성하는 `sys_symlink` 를 추가

```
// Create the path new as a symbolic link to the same inode as old.
int
sys_symlink(void)
{
  char *target, *linkpath;

  if (argstr(0, &target) < 0 || argstr(1, &linkpath) < 0)
    return -1;

  begin_op();
```

```
  struct inode *ip = create(target, T_SYMLINK, 0, 0);
  if (ip == 0) {
    end_op();
    return -1;
  }

  if (writei(ip, linkpath, 0, strlen(linkpath)) < 0) {
    iput(ip);
    end_op();
    return -1;
  }

  iput(ip);
  end_op();

  return 0;
}
```

`target` 매개 변수는 링크 대상의 경로이고 `linkpath` 매개 변수는 심볼릭 링크의 경로입니다. `create` 함수를 통해 심볼릭 링크의 inode를 생성하고, `writei` 함수를 사용하여 심볼릭 링크의 내용을 씁니다. 이후 파일 시스템 동기화 작업을 수행하고, 성공적으로 생성된 경우 `0`을 반환합니다.

3. Sync 구현

- 죄송합니다.