[ELE3021] Operating System

Project #3 Wiki

2019014266 Lim Kyu Min

Contents

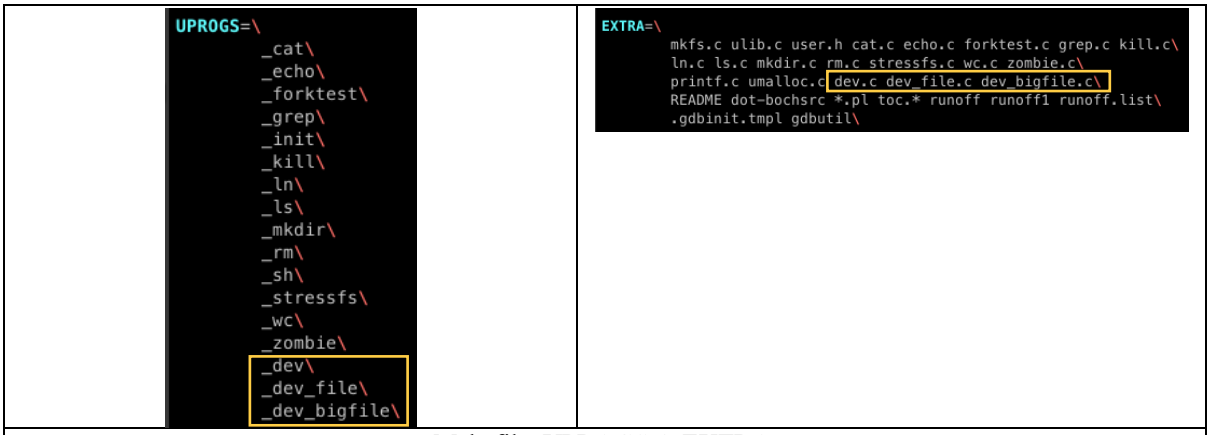
* 코드에서 //* 로 시작하는 주석은 해당 프로젝트 구현 도중 제가 단 주석이며, 이는 코드 구현의 의도와 내용을 이해하는 데에 도움을 줄 것입니다.

* Project #3의 Multi - Indirect를 테스트하는 Test Code는 Google에서의 xv6에 huge file을 write & read하는 코드를 참고하였습니다. 해당 파일의 이름은 dev_bigfile.c이며, dev_bigfile로 실행할 수 있습니다.

I. Introduction

Project #3 의 구현을 할 수 있는 시간이 많지 않아 최대한 간단하고 직관적인 로직으로 개발을 진행했습니다. 해당 Wiki 에서는 어떠한 로직과 방법으로 과제 명세들을 구현했는지 서술합니다. 이후에는 해당 기능을 시연하고, 마지막으로는 Trouble Shooting & Limitation 에서는 Project #2 와 마찬가지로 겪은 문제점과 어떻게 해결했는지, 예상되는 문제점들을 서술함으로써 해당 Wiki 를 마무리합니다.

- Makefile



Makefile, UPROGS & EXTRA.

- Project #3의 과제 구현 목록의 특성 상, 새로운 파일이 필요하지 않아 (즉, 기존 파일을 수정하는 방향으로 진행해도 되어서) 따로 OBJS에 추가된 파일은 존재하지 않습니다.
- Project #3에 추가된 User Program들은 과제 구현 도중에 symbolic link 테스트를 위한 dev, sync관련 기능을 확인하기 위한 dev_file, Multi-Indirect의 기능 구현을 테스트하기 위한 테스트 프로그램인 dev_bigfile이 있습니다.

II. Multi-Indirect

Project #3 의 첫번째 구현 대상인 Multi-Indirect 의 구현에 대해 서술하겠습니다.

Multi-Indirect 의 구현 내용은 Direct 와 Single Indirect 만을 지원하는 기존 xv6 를 확장하여 Double Indirect, Triple Indirect 을 적용하는 것입니다. 해당 내용을 구현하기 위해서 param.h, fs.h 의 정보와 기존의 Direct, Single Indirect 방식의 Allocation 을 담당하는 fs.c 를 주로 수정하였습니다.

I. param.h

```
#define NPROC        64  // maximum number of processes
#define KSTACKSIZE 4096  // size of per-process kernel stack
#define NCPU          8  // maximum number of CPUs
#define NOFILE       16  // open files per process
#define NFILE       100  // open files per system
#define NINODE       50  // maximum number of active i-nodes
#define NDEV         10  // maximum major device number
#define ROOTDEV       1  // device number of file system root disk
#define MAXARG       32  // max exec arguments
#define MAXOPBLOCKS  10  // max # of blocks any FS op writes
#define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
#define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
#define FSSIZE       100000  // size of file system in blocks
```

param.h – Changed parameter values

다음 parameter 가 수정되었습니다.

- FSSIZE: 1000 -> 100000
  - Double Indirect, Triple Indirect 를 구현하게 되면 기존 xv6 가 다룰 수 있는 것 이상의 크기의 파일에 대한 Operation 이 가능합니다. 해당 기능에 대한 제약이 없도록 FSSIZE 를 대폭 늘렸습니다.

II. fs.h

```
#define NDIRECT 10 //* Reduced NDIRECT num
                   //* Cannot Add multiple indirect without reducing Ndirect
                   //* Changed size of dinode will leads to error
                   //* mkfs.c:84: main: Assertion `(BSIZE % sizeof(struct dinode)) == 0' failed.
#define NINDIRECT (BSIZE / sizeof(uint))
#define DINDIRECT (128 * NINDIRECT)
#define TINDIRECT (128 * DINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT + DINDIRECT + TINDIRECT)

#define LAYERLIMIT 128
#define LARGELAYERLIMIT 16384
```

```
// 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 Indirect
  uint T_addr;           //* Triple Indirect
};
```

fs.h – Changed/New parameter Value & Changed dinode

다음과 같은 parameter 가 수정되거나 추가되었습니다.

- NDIRECT: 12 -> 10
  - xv6 의 dinode 의 크기 자체를 변경할 수는 없어서(크기가 변경되면 build 과정에서 오류가 나게 됩니다.), 기존 Direct 방식을 위한 공간 중 2 개를 각각 Double indirect 와

Triple Indirect 를 위한 공간으로 할당해주기로 했습니다. 다음과 같은 과정에서 Direct 의 개수를 12 개에서 10 개로 줄여야 했습니다.

- DINDIRECT: 128 * NINDIRECT (== 16384)
  - Double Indirect 의 수행을 위해 필요한 값입니다. Double Indirect 의 최대 Limit 를 나타내며, Double Indirect 가 handle 할 수 있는 file 의 크기를 대략적으로 나타냅니다.
- TINDIRECT: 128 * DINDIRECT (== 2097152)
  - Triple Indirect 의 수행을 위해 필요한 값이며, Triple Indirect 가 다룰 수 있는 파일의 최대 상한선을 나타냅니다.
- MAXFILE: (NDIRECT + NINDIRECT) -> (NDIRECT + NINDIRECT + DINDIRECT + TINDIRECT)
  - 처리할 수 있는 파일의 최대 상한선을 Triple Indirect 수준까지 높였습니다.
- LAYERLIMIT, LARGELAYERLIMIT: 128, 16384
  - Double Indirect, Triple Indirect 의 Allocation 시 사용됩니다. LargeLayerLimit 은 128 * LayerLimit 으로도 대체가 가능합니다.

Dinode structure 에서는 D_addr, T_addr 가 추가되었습니다. 각각 Double Indirect, Triple Indirect 를 담당하는 address variable 입니다.

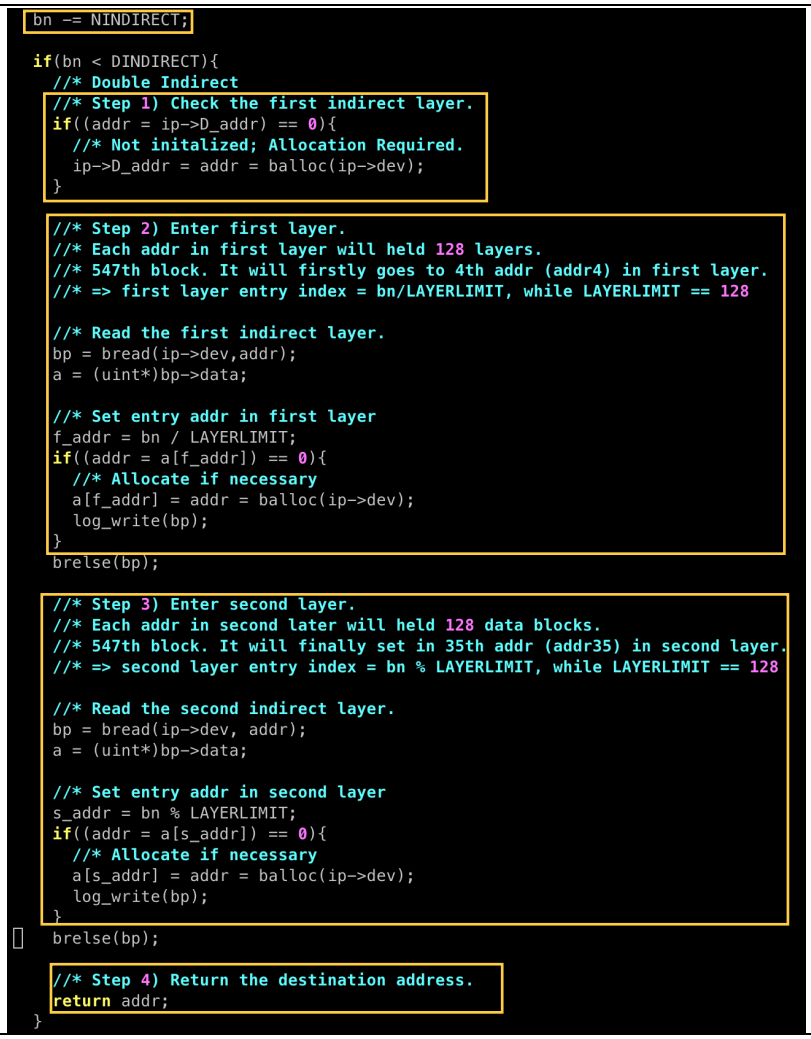
III. fs.c

본격적인 Multi-Indirect 의 구현 내용입니다. 해당 구현은 기존 allocation 을 담당하는 fs.c 의 bmap()을 수정함으로써 구현했습니다. 구현 내용을 3 부분으로 나누어 서술하겠습니다.

```
// Return the disk block address of the nth block in inode ip.
// If there is no such block, bmap allocates one.
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  uint f_addr, s_addr, t_addr; //* addr indicator of first layer and second layer
  struct buf *bp;

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

  //* Single Indirect
  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 bmap() [1/3] – Newly declared variables for Multi-Indirect

- uint f_addr, s_addr, t_addr
  - 각각 first_addr, second_addr, third_addr 를 뜻하며, Triple indirect 기준으로 첫번째 layer 의 주소, 두번째 layer 의 주소, 세번째 layer 의 주소를 가리킵니다.
  - 이하는 기존 xv6 에 구현된 기존 Direct, Single Indirect 의 구현 내용입니다. Double Indirect 와 Triple Indirect 는 해당 내용 하단에 구현되어있습니다.



```
  bn -= NINDIRECT;

  if(bn < DINDIRECT){
    //* Double Indirect
    //* Step 1) Check the first indirect layer.
    if((addr = ip->D_addr) == 0){
      //* Not initalized; Allocation Required.
      ip->D_addr = addr = balloc(ip->dev);
    }

    //* Step 2) Enter first layer.
    //* Each addr in first layer will held 128 layers.
    //* 547th block. It will firstly goes to 4th addr (addr4) in first layer.
    //* => first layer entry index = bn/LAYERLIMIT, while LAYERLIMIT == 128

    //* Read the first indirect layer.
    bp = bread(ip->dev,addr);
    a = (uint*)bp->data;

    //* Set entry addr in first layer
    f_addr = bn / LAYERLIMIT;
    if((addr = a[f_addr]) == 0){
      //* Allocate if necessary
      a[f_addr] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);

    //* Step 3) Enter second layer.
    //* Each addr in second later will held 128 data blocks.
    //* 547th block. It will finally set in 35th addr (addr35) in second layer.
    //* => second layer entry index = bn % LAYERLIMIT, while LAYERLIMIT == 128

    //* Read the second indirect layer.
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;

    //* Set entry addr in second layer
    s_addr = bn % LAYERLIMIT;
    if((addr = a[s_addr]) == 0){
      //* Allocate if necessary
      a[s_addr] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);

    //* Step 4) Return the destination address.
    return addr;
  }
```

fs.c bmap() [2/3] – Double Indirect

다음과 같은 방법으로 Double Indirect 는 진행이 됩니다.

- 0. 기존 bn 에서 증가된 NINDIRECT 만큼의 값을 빼주어 Double Indirect 의 첫번째 index 부터 indexing 하게 합니다.
- 1. Double Indirect 의 allocation 이 시작됩니다. D_addr 에 balloc()으로 새로운 address 를 할당합니다.
- 2. 현재 bn 의 값을 기준으로 LAYERLIMIT 의 몫을 기준으로 첫번째 layer 의 주소 (f_addr)를 결정합니다. 이는 기존의 pageing 방법의 scheme 을 그대로 가져온 것으로, 해당 값의 quotient 값은 첫번째 index, remaining value 는 두번째 index 로 할당하는 방법입니다. 첫번째 layer 주소가 결정되면 해당 자리에 balloc()으로 allocation 합니다.
- 3. 이후 Second Layer 의 indexing 을 시작합니다. LAYERLIMIT 의 나머지를 기준으로 두번째 layer 의 addr 를 결정하며 (s_addr), 결정 후 Step 2 와 같이 balloc()으로 할당해줍니다.
- 4. 모든 allocation 이 끝나면, 해당 마지막으로 할당된 두번째 layer 의 주소를 반환해줍니다. 다음과 같은 방법으로 Double Indirect 구현을 완료했습니다.

```
bn -= DINDIRECT;

if(bn < TINDIRECT){
  //* Triple Indirect
  //* It will be the same logic with double indrect but more layers.
  if((addr = ip->T_addr) == 0){
    //* Not initalized; Allocation Required.
    ip->T_addr = addr = balloc(ip->dev);
  }

  //* Step 2) Enter first layer.
  //* Each addr in first layer will held 128 second layers, which also held 128 layers
  //* => The single addr in first layer will hold 16384 daya blockd
  //* => first layer entry index = bn/LARGELAYERLIMIT, while LARGELAYERLIMIT == 16384

  //* Read the first indirect layer.
  bp = bread(ip->dev,addr);
  a = (uint*)bp->data;

  //* Set entry addr in first layer
  f_addr = bn / LARGELAYERLIMIT;
  if((addr = a[f_addr]) == 0){
    //* Allocate if necessary
    a[f_addr] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);

  //* Step 3) Enter second layer.
  //* Each addr in second later will held 128 layers.
  //* Its index will be computed same way for first layer in double indirect
  //* => second layer entry index = (bn % LARGELAYERLIMIT)/LAYERLIMIT

  //* Read the second indirect layer.
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;

  //* Set entry addr in second layer
  s_addr = (bn % LARGELAYERLIMIT)/LAYERLIMIT;
  if((addr = a[s_addr]) == 0){
    //* Allocate if necessary
    a[s_addr] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);
  //* Step 4) Enter third layer
  //* Each addr in third layer will held 128 data blocks
  //* Its index will be couputed in same way for second layer in double indirect
  //* => third layer enter index = (bn % LARGELAYERLIMIT)%LAYERLIMIT.

  //* Read the third indirect layer.
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;

  //* Set entry addr in third layer
  t_addr = (bn % LARGELAYERLIMIT) % LAYERLIMIT;
  if((addr = a[t_addr]) == 0){
    //* Allocate if necessary
    a[t_addr] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);

  //* Step 5) Return the destination address
  return addr;

 }

 panic("bmap: out of range");
}
```

fs.c bmap() [3/3] – Triple Indirect

Triple Indirect 는 Double Indirect 의 구현 방법과 매우 흡사합니다. Double Indirect 보다 한 단계 더 많이 indexing 작업을 한 것이라고 생각하면 됩니다. 로직 또한 매우 흡사하니 간단하게 서술하겠습니다.

- 0. 기존 bn 에서 증가된 DINDIRECT 만큼의 값을 빼주어 Triple Indirect 의 첫번째 index 부터 indexing 하게 합니다.
- 1. Triple Indirect 의 allocation 이 시작됩니다. T_addr 에 balloc()으로 새로운 address 를 할당합니다.
- 2. 현재 bn 의 값을 기준으로 LARGELAYERLIMIT 의 몫을 기준으로 첫번째 layer 의 주소(f_addr)를 결정하고, 해당 자리에 balloc()으로 allocation 합니다.
- 3. 이후 Second Layer 의 indexing 을 시작합니다. LARGELAYERLIMIT 의 나머지를 LAYERLIMIT 로 한번 더 나눈 후 해당 값으로 두번째 layer 의 addr(s_addr)를 결정하며, 결정 후 Step 2 와 같이 balloc()으로 할당해줍니다.

- 4. 마지막으로 Third Layer 의 indexing 을 시작합니다. Second Layer Index 연산의 나머지 값이 Third index ((bn % LARGELAYERLIMIT) %L AYERLIMIT)이므로, 해당 값을 third layer 의 address (t_addr)로 설정 후 balloc()으로 allocation 합니다.
- 5. 모든 allocation 이 끝나면, 해당 마지막으로 할당된 세번째 layer 의 주소를 반환해줍니다. 다음과 같은 방법으로 Triple Indirect 구현을 완료했습니다.

다음과 같은 방법으로 Double Indirect, Triple Indirect 의 구현을 모두 완료했고, Multi-Indirect 의 구현을 완료했습니다.

III. Symbolic Link

Project #3 의 두번째 구현 대상인 Symbolic Link 의 구현에 대해서 서술하겠습니다.

xv6 의 기본으로 구현되어있는, inode 로 접근하는 hardlink 와는 달리, soft link 는 path 정보로 접근을 하며, 그로 인해 original file 이 삭제되거나 이름이 바뀌면 open 과 같은 작업을 수행하지 못하고 오류를 발생시킵니다.

처음에는 readi(), writei()등을 수정하여 구현할 생각을 했지만, 모든 file operation (create, open, read, write …etc)는 open()호출을 통해 혹은 그 이후에 실행된다는 점에 주목을 했고, 그로 인해 open() systemcall 을 위주로 수정을 진행했습니다. 그 이외에 exec, ls 와 같은 systemcal 에서 symbolic link 구현을 위한추가적인 수정을 거쳐서 Symbolic Link 구현을 완료했습니다.

I. stat.h

```
#define T_DIR  1   // Directory
#define T_FILE 2   // File
#define T_DEV  3   // Device
#define T_SBLK 4   //* Symbolic Linked file
```

stat.h – Newly Added Symbolic Link File Type

T_SBLK (Symbolic Link)에 해당하는 새로운 File Type을 새로 정의합니다.

II. sys_link()

sys_link()에는 link와 직접적으로 연관되어 있는 System call임으로, Symbolic Link의 주된 내용이 구현되어 있습니다.

```
// Create the path new as a link to the same inode as old.
//* Link will now support symbolic link
int
sys_link(void)
{
  char name[DIRSIZ], *new, *old, *flag;
  int length;
  struct inode *dp, *ip;

  if(argstr(0, &flag) < 0 || argstr(1, &old) < 0 || argstr(2, &new) < 0){
    cprintf("link: read argument failed\n");
    return -1;
  }

  begin_op();
  if(flag[0] == '-' && flag[1] == 'h'){
    //* Standard scheme: Hard link - share I-node
    if((ip = namei(old)) == 0){ //* namei: get inode for current old path.
      end_op();
      return -1;
    }

    ilock(ip);
    if(ip->type == T_DIR){ //* If current inode is directory
      iunlockput(ip);
      end_op();
      return -1;
    }

    ip->nlink++; //* Increase inode's linked number.
    iupdate(ip); //* iupdate() copy a modified in-memory inode to disk
                 //- called after every changes of ip->xxx
    iunlock(ip);

    if((dp = nameiparent(new, name)) == 0) //* nameiparent: get inode of the parent, and copy.
      goto bad;
    ilock(dp);
    if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
      iunlockput(dp);
      goto bad;
    }
    iunlockput(dp);
    iput(ip);
```

sys_link() [1/2] – Newly Declared Variables & Handling for Hard-Link Request

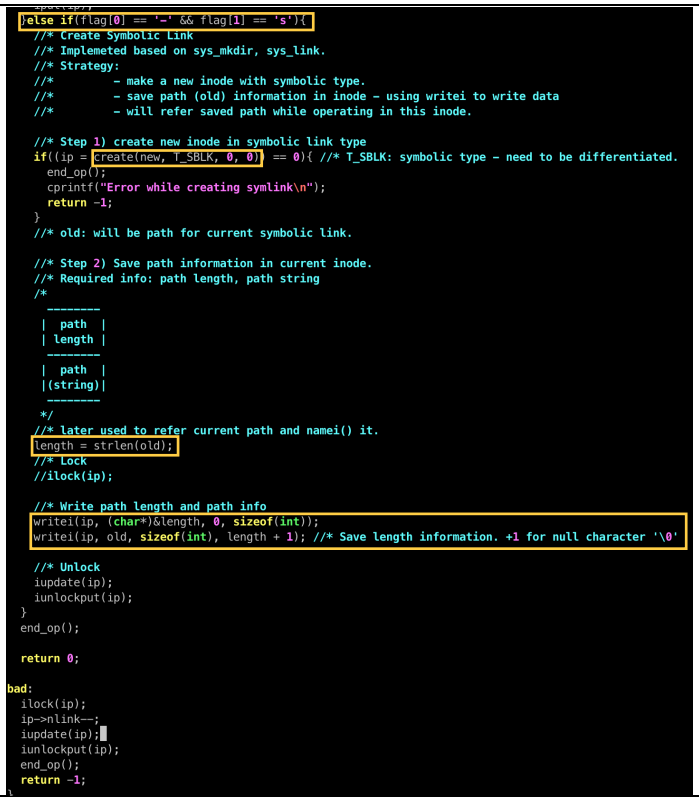
새로 추가된 변수는 다음과 같습니다.

- char *flag: 해당 request가 hard link인지, soft link인지 구분합니다.
- int length: path의 length 정보를 저장하며, 해당 정보는 symbolic link에서 사용이 됩니다.

Symbolic Link를 지원한다고 해서 기존의 hard link에 대한 기능을 지우는 것은 아닙니다. 다음과 같은 방법으로 hard link과 symbolic link에 대한 request를 구분합니다.

- ln -h [old] [new] : new는 old에 대한 hard link입니다.
- ln -s [old] [new]: new는 old에 대한 symbolic link입니다.

이때 -h, -s에 대한 정보는 flag에 저장이 되며 (argstr(1, &old) < 0), flag이 '-h'에 해당할 경우 기존의 hard link 형성 로직이 돌아가게 됩니다.

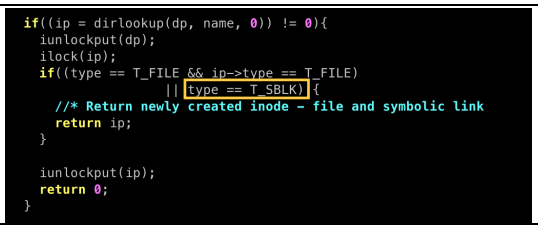


```
}else if(flag[0] == '-' && flag[1] == 's'){
  //* Create Symbolic Link
  //* Implemeted based on sys_mkdir, sys_link.
  //* Strategy:
  //*         - make a new inode with symbolic type.
  //*         - save path (old) information in inode - using writei to write data
  //*         - will refer saved path while operating in this inode.

  //* Step 1) create new inode in symbolic link type
  if((ip = create(new, T_SBLK, 0, 0)) == 0){ //* T_SBLK: symbolic type - need to be differentiated.
    end_op();
    cprintf("Error while creating symlink\n");
    return -1;
  }
  //* old: will be path for current symbolic link.

  //* Step 2) Save path information in current inode.
  //* Required info: path length, path string
  /*
    --------
   |  path  |
   | length |
    --------
   |  path  |
   |(string)|
    --------
   */
  //* later used to refer current path and namei() it.
  length = strlen(old);
  //* Lock
  //ilock(ip);

  //* Write path length and path info
  writei(ip, (char*)&length, 0, sizeof(int));
  writei(ip, old, sizeof(int), length + 1); //* Save length information. +1 for null character '\0'

  //* Unlock
  iupdate(ip);
  iunlockput(ip);
 }
 end_op();

 return 0;

bad:
 ilock(ip);
 ip->nlink--;
 iupdate(ip);
 iunlockput(ip);
 end_op();
 return -1;
}
```

sys_link() [2/2] – Handling for symbolic link request

flag가 '-s'에 해당할 경우, Symbolic Link를 만드는 routine에 들어가게 됩니다. 동작 방법은 다음과 같습니다.

- 1. create()를 통해 새로운 파일을 형성합니다. 이를 위해 create()함수를 sys_link() 위치상 위로 옮기고, T_SBLK라는 File Type이 올 경우 T_FILE일때와 같이 새로운 inode를 return하게 합니다.



```
if((ip = dirlookup(dp, name, 0)) != 0){
  iunlockput(dp);
  ilock(ip);
  if((type == T_FILE && ip->type == T_FILE)
                  || type == T_SBLK) {
    //* Return newly created inode - file and symbolic link
    return ip;
  }

  iunlockput(ip);
  return 0;
}
```

Part of create() – Return ip if (type == T_FILE or type == T_SBLK)

- 2. 해당 새로운 inode에 path 정보를 writei를 활용하여 입력합니다. 나중에 해당 path정보로 symbolic link의 redirection을 구현할 것입니다. (path 정보에 기반한 file redirection.)
  - 먼저 0번째 위치에는 length를 입력합니다. 해당 정보를 기반으로 이후에 readi를 호출할 시, 해당 length 정보를 기반으로 path 정보를 extract할 것입니다.
  - Length 정보를 저장한 후, 그 정보 바로 다음 자리에 (sizeof(int)) 바로 path를 적습니

다. 다음과 같이 path정보를 기록하여 이후에 open()과 같은 다른 함수에서 온전히 path 정보를 extract할 수 있게 했습니다. (0번째에 있는 length 정보를 먼저 읽은 후, 해당 위치 이후에 해당 length 만큼 읽어오면 path가 읽어와지게 됨으로.)

III. fcntl.h

open() system call의 수정사항을 설명하기 전에, fcntl.h에 새로 구현된 mode type을 먼저 설명하겠습니다.

```
#define O_RDONLY  0x000
#define O_WRONLY  0x001
#define O_RDWR    0x002
#define O_CREATE  0x200
#define O_NFSBLK  0x100 //* This flag stands for not following symbolic link.
```

setmemorylimit() – Receive and set memory limit to current process.

- O_NFSBLK (Not Follow Symbolic Link) : symbolic link 파일을 발견할 시, 해당 mode가 enable되어 있으면 Symbolic Link를 따라가지 않고 해당 file을 그대로 반환합니다.
- ls와 같이 모든 file 정보를 출력하는 system call과 같은 명령어에서, Symbolic Link를 따라가 원본의 inode를 출력하는 것이 아니라 Symbolic Link의 정보 (즉 new에 해당하는 path name과 정보)을 출력해야 하는 상황에 사용이 됩니다.

IV. sys_open()

```
if(omode & O_CREATE){
  ip = create(path, T_FILE, 0, 0);
  if(ip == 0){
    end_op();
    return -1;
  }
} else {
  if((ip = namei(path)) == 0){
    end_op();
    return -1;
  }
  ilock(ip);
  if(ip->type == T_DIR && (omode != O_RDONLY && omode != O_NFSBLK)){ //* not readonly nor not follow
    iunlockput(ip);
    end_op();
    return -1;
  }
  if(ip->type == T_SBLK && (omode != O_NFSBLK)){ //* will not follow if this omode is O_NFSBLK (not follow symbolic link).
    // * Current inode is symbolic link
    // * read path length, path information in inode.
    // * based on path information, reach inode until inode is non-symbolic link.

    do {
      // * read first info: path length
      readi(ip, (char*)&length, 0, sizeof(int));

      // * read path info based on length info
      // * update path info
      readi(ip, path, sizeof(int), length + 1);
      iunlockput(ip);

      // * get inode for current path, then update ip.
      if((ip = namei(path)) == 0){
        // * Error: cannot find current inode
        // * Original inode could be deleted.
        cprintf("Error: Inode cannot found. Original file could be deleted or possible inode corruption occured.\n");
        end_op();
        return -1;
      }
      ilock(ip); // * Lock again; readi.
      // * If newly updated ip is symbolic link,
      // * loop again until non-symbolic found.
    } while(ip->type == T_SBLK);
  }
}
```

sys_open() – Add Handler for Symbolic Link File

위 구현 방법에서 설명한 대로, symbolic link에 해당하는 파일이 올 경우 open에서는 path 정보를 기반으로 가리키는 inode를 찾는, redirection mechanism을 통해 가리키는 original inode를 찾습니다. 기존에 sys_open()에서 수정된 일부만 설명을 진행하겠습니다.

begin_op() 이후부터,

- 1. 해당 ip의 type이 T_DIR일 때 omode가 O_RDONLY가 아니라 O_NFSBLK일 경우에도 (즉, symbolic link를 not follow 할 경우) 오류로 판단하지 않고 다음 routine으로 넘어갑니다.
- 2. 만약 해당 ip의 type이 T_SBLK이고, omode가 O_NFSBLK가 아닐 경우에, symbolic link를 redirection하는 routine에 들어갑니다.

- i) 0번째 자리에 있는 length 정보를 readi()를 통해 읽어드립니다.
- ii) 읽어드린 length 정보를 기반으로 그 다음 위치에 있는 path 정보를 readi()를 통해 읽습니다.
- iii) 읽어온 path를 namei()를 통해 해당하는 inode를 읽어옵니다. 이때 다음과 같은 분기가 발생합니다.
  - Inode를 불러온 경우: 해당 inode의 type를 체크합니다. Type이 여전히 T_SBLK일 경우 (즉, 해당 symbolic link가 가리키는 file 역시 symbolic link일 경우), 다시 해당 routine을 돌립니다. 만일 다른 type의 경우, 해당 inode는 original file에 해당하는 inode이며, 해당 inode로 작업을 계속합니다
  - Inode를 불러오는데 실패한 경우: 이 경우는 path에 해당하는 inode를 찾을 수 없는 경우 (즉, original file이 삭제되거나 이름이 바뀐 경우)이며, 이 경우 오류 메세지를 출력하고 작업을 끝냅니다.
- iv) 정상적으로 original file의 inode를 불러온 경우, 해당 inode로 나머지 open() 작업을 지속합니다. 이후는 기존의 open()과 동일합니다.

V. exec()

```
if((ip = namei(path)) == 0){
  end_op();
  cprintf("exec: fail\n");
  return -1;
}

//* follow path if current ip is symbolic link
////* follow path if current ip is symbolic link;;
//ip = followip(ip, path);


ilock(ip);
ip = followip(ip, path);

if(ip == 0){
  //iunlock(ip);
  end_op();
  cprintf("exec: file corrupted.\n");
  return -1;
}
pgdir = 0;
```

Part of exec() – Check if current inode is symbolic link or not.

exec()에서 symbolic link의 파일을 실행할 경우, 해당 파일이 가리키는 original file을 return 해주어야지만 실행할 수 있습니다. 이 기능은 followip()에서 구현을 했고, return되는 inode가 없으면 이 역시 symbolic file이 가리키던 original file이 삭제되거나 이름이 바뀐 경우로 판단하여 오류 메세지를 출력한 후에 작업을 끝냅니다.

followip()는 다음과 같습니다. sys_open()에서 symbolic link의 original file을 찾는 과정과 동일합니다.

```
//* Modify Open - symbolic link
struct inode*
followip(struct inode* ip, char* path){
  int length;
  // * Check if current inode is symbolic link
  // * if symbolic link, go to directing link and return it.

  //ilock(ip);

  if(ip->type == T_SBLK) {
    // * Current inode is symbolic link
    // * read path length, path information in inode.
    // * based on path information, reach inode until inode is non-symbolic link
    do {
      // * read first info: path length
      readi(ip, (char*)&length, 0, sizeof(int));

      // * read path info based on length info
      // * update path info
      readi(ip, path, sizeof(int), length + 1);
      iunlockput(ip);

      // * get inode for current path, then update ip.
      if((ip = namei(path)) == 0){
        // * Error: cannot find current inode
        // * Original inode could be deleted.
        cprintf("Error: Inode cannot found. Original file could be deleted or possible inode corruption occured.\n");
        //end_op();
        return 0;
      }
      ilock(ip); // * Lock again; readi.
      // * If newly updated ip is symbolic link,
      // * loop again until non-symbolic found.
    } while(ip->type == T_SBLK);
  }

  return ip;
}
```

followip() – Find original file if current inode is symbolic link

VI. ls.c

마지막으로 ls에 대해서 symbolic link file이 있을 경우, 해당 file에 맞게 출력할 수 있도록 약간의 수정을 진행해줍니다.

```
void
ls(char *path)
{
  char buf[512], *p;
  int fd;
  struct dirent de
  struct stat st;

  if((fd = open(path, O_NFSBLK)) < 0){ //* list -> will not follow symbolic link
    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_SBLK: //* Symbolic Link
    printf(1, "%s %d %d %d (Symbolic Link)\n", fmtname(path), st.type, st.ino, st.size);
    break;
  case T_FILE:
    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(statnf(buf, &st) < 0){ //* statnf: same with stat, but with O_NFSBLK flag
        printf(1, "ls: cannot stat %s\n", buf);
        continue;
      }
      if(st.type == T_SBLK){
        printf(1, "%s %d %d %d (Symbolic Link)\n", fmtname(buf), st.type, st.ino, st.size);
      }else{
      printf(1, "%s %d %d %d\n", fmtname(buf), st.type, st.ino, st.size);
      }
    }
    break;
  }
  close(fd);
}
```

ls.c – Change open mode and add handle routine for symbolic link

다음과 같은 2가지의 주요 수정사항이 있습니다.

- 1. Open(): open mode를 O_RDONLY에서 O_NFSBLK로 바꿉니다. 기존 O_RDONLY로 ls를 호출할 경우, symbolic file에 대한 정보가 아닌, 해당 파일이 가리키는 original file의 정보를 출력하게 되어 ls의 기능을 제대로 수행하지 못하게 됩니다.
- 2. Case T_SBLK: Symbolic Link에 대해서는 (Symbolic Link)를 출력하여 해당 파일이 Symbolic link임을 출력하게 했습니다. 이를 case T_DIR: 이후에도 추가하여 ls . / ls .. 와 같이 directory를 출력하는 경우에도 출력할 수 있게 handling을 했습니다.

이상으로 symbolic link 구현에 대한 내용 서술을 마치겠습니다.

IV. Sync (Buffered I/O)

Project #3 의 세번째 요구사항인 sync 의 구현에 대해 서술합니다.

기존 xv6 에서 commit()을 통한 group flush 와 달리, 이제 flush 는 sync() system call 로 인해서만 발생합니다. 즉, 유저가 sync()를 통해 자유롭게 flush 할 수 있게 만들어 더욱 효율적으로 Disk I/O 를 수행할 수 있게 합니다.

sync()가 호출되거나 Buffer 의 공간이 다 찼을 경우에는 모든 Dirty Buffer 가 flush 됩니다. 다만, sync()를 호출하지 않고 file close 를 하는 경우, 변경 내용을 반영되지 않습니다.

수정은 log.c 에서만 일어났습니다. 이하는 log.c 의 어떠한 system call 이 수정되었고 추가되었는지 적겠습니다.

I. struct log

```
struct log {
  struct spinlock lock;
  int start;
  int size;
  int outstanding; // how many FS sys calls are executing.
  int committing;  // in commit(), please wait.i
  int flushing; //* wait until flushing ends - it has to be done without intervention.
  int dev;
  struct logheader lh;
};
struct log log;
```

log – new attribute flushing

Log structure 에 flushing 을 추가합니다. Flush 가 진행중일 때에 해당 값이 1 로 변하며, 이때 또 다른 sync() 호출로 인한 중복된 flush 시도를 abort 하는데 사용됩니다.

II. begin_op()

```
// called at the start of each FS system call.
//* Changed feature in Project #3 - Buffered I/O
//* begin_op will do one more job; call sync() if buffer fulled.
//* Standard of full: LOGSIZE - 2 (Some blocks need to be reserved for log operation. I set this reserve amount as 2 blocks.)
//* Based on log number and # outstanding blocks, check if the log full.
//* If the log full, call sync(), flush the buffer.
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 - 2){ //* Reserve for 2 block for log operation
      // this op might exhaust log space; wait for commit.
      //sleep(&log, &log.lock);
      sync();
    } else {
      log.outstanding += 1;
      release(&log.lock);
      break;
    }
  }
}
```

begin_op() – begin_op() will call sync() if buffer is fulled.

begin_op()는 다음과 같은 점이 수정되었습니다.

- 만약 buffer 가 full 되었을 경우 (outstanding 포함), sleep 이 하는 것이 아니라 바로 sync()를 호출하여 flush 를 진행합니다.
- 이때 Full 의 기준이 outstanding 을 감안하고도 여분의 2 개의 buffer 를 남겨두는데, (LOGSIZE - 2), 이를 처리하지 않으면 어떤 특정한 경우에 deadlock 이 걸리는 오류가 발생합니다. 이는 기본적인 logging operation 을 위한 여분의 spare space 가 필요해서 발생하는 오류로, spare space 2 칸을 항상 예비하도록 설정했습니다. 이를 적용하니 더 이상 deadlock 과 같은 오류가 발생하지 않게 되었습니다.
- 이 점은 이후 Trouble Shooting 에서 더 자세히 서술하겠습니다.

III. end_op()

```
// called at the end of each FS system call.
// commits if this was the last outstanding operation.
//* Changed feature in Project #3 - buffered I/O
//* Commit will NOT flush buffer.
//* Flush will be totally managed by systemcall sync();
//* end_op will now just resolve outstanding block; reduce the number if end_op called.
//* It will work like an 'marker'. (Mark the range of operation need to be logged.)
void
end_op(void)
{
  acquire(&log.lock);
  //* Resolve Outstanding Block
  log.outstanding -= 1;

  if (log.outstanding == 0){
    //* It won't call sync
    //* As all outstanding blocks are resloved,
    //* Wake the log wating the outstanding blocks to be resolved.
    wakeup(&log); //* This wakeup is for waiting log in sync();
  }

  //* for the case log.outstanding > 0
  //* begin_op() will now not sleep if the buffer is full.
  //* No need to wait such condition; begin_op will immediately flush the buffer.

  release(&log.lock);
}
```

end_op() – end_op() will now only increase & decrease outstanding value.

end_op()는 굉장히 구조가 간단해집니다. 일단 다음과 같은 기능이 사라지게 됩니다.

- 1. Commit: commit 을 통한 buffer flush 기능은 이제 sync()가 하게 됩니다. end_op 에서의 관련 호출 기능은 필요 없습니다.
- 2. Wakeup begin_op: 기존 end_op 에서는 begin_op()의 buffer 가 full 되었을 경우 해당 operation 이 waiting 하게 되는데, 이를 다시 깨우는 역할을 하는 wakeup()이 있었습니다. 하지만 sync()에서의 begin_op()는 buffer 가 full 되었다고 해서 wait 하지 않고 바로 flush 해버림으로 이 기능 또한 필요 없습니다.

다음과 같은 기능을 모두 제거하고, end_op 는 outstanding 의 수를 decrement 합니다. 추가로 outstanding 의 수가 0 이 되었을 시, wakeup()하는 로직이 있는데, 이는 sync()와 관련이 되어 있습니다.

IV. sync()

```
//* sync()
//* sync will check the buffer and flush it.
// Step 1) If the lock isn't held (e.g. called from user program.) it will internally acquire log.lock.
// Step 2) If there is still outstanding block, wait until it resolved (wait until end_op calls)
// Step 3) Flush the buffer, write the changes on the disk
// Step 4) if the log.lock acquired internally, release it.
int
sync(void){
  int logLocked = log.lock.locked; //* check log lock state
  int blockConsumed = 0; //* Value need to be returned; # block flushed.

  if(logLocked == 0){ //* If the lock isn't held
    acquire(&log.lock); //* Call Internally
  }

  while(log.outstanding > 0){ //* If there is an outstanding block
    sleep(&log, &log.lock); //* Wait for it
  }

  blockConsumed = flush(); //* Flush buffers

  if(logLocked == 0){ //* If the lock called internally
    release(&log.lock);   //* release the lock.
  }

  return blockConsumed; //* Return # block flushed
}
```

sync() – Check the buffer and flush them

Buffer 와 각종 조건을 확인하여 buffer flush 하는 sync() 함수에 대한 서술입니다. Sync 는 system call 임으로, User Program 에서도 호출할 수 있습니다. Sync 는 다음과 같은 순서로 동작합니다.

- 1. Check Log Lock: logLocked variable 을 통해 현재 lock 이 걸려있는지 확인합니다. Lock 이 걸려있지 않은 경우는 해당 sync()가 begin_op()가 아니라 User Program 에서 호출되었다는 뜻입니다. 해당 경우 lock 을 acquire 시켜줍니다.

- 2. Check Outstanding: Outstanding 의 값을 확인한 후, 만약 outstanding 한 block 이 여전히 남아있으면, sleep()을 호출하여 outstanding 값이 0 이 될 때까지 (즉 begin_op()를 호출한 모든 작업이 end_op 를 호출 할 때까지) wait 합니다.
  - ■ end_op()에서 마지막 outstanding block 이 resolve 되어 0 이 되었을 경우 wakeup()을 호출하게 되는데, 이때 해당 sync()가 wakeup 하게 됩니다.
- 3. Flush: flush()를 진행합니다. 해당 함수는 총 flush 된 block 의 수를 반환하며, 이를 blockConsumed 에 저장합니다.
- 4. Relinquish lock: 만약 Step 1 에서 lock 이 걸려있지 않아 임의로 lock 을 acquire 했을 경우, 다시 release 를 해줍니다.

모든 작업이 완료된 후, blockConsumed, 즉 flush 된 block 의 수를 반환합니다.

```
int
sys_sync(){
  if(log.flushing == 1){
    //* Currently flushing buffers!
    cprintf("sync: busy!\n");
    return -1;
  }
  return sync();
}
```

sys_sync() – Wrapper function for sync()

sync()에 대한 wrapper function 입니다. 만약 flushing 이 1, 즉 현재 flush 가 실행 중이면, busy 하다는 시스템 메세지를 출력한 후, 해당 시도를 abort 합니다.

V. flush()

```
//* flush()
//* flush will do the original commit job.
//* remove the all elements in buffer, and write those contents into disk.
int
flush(void){
  int flushed = 0;

  //* Prevent flush if currenlty commiting
  log.committing = 1;
  release(&log.lock); //* release the log lock
  log.flushing = 1; //* Flush start; Another sync() call will be ignored.

  if(log.lh.n > 0) { //* Original Commit Call
    //* Flush buffer.
    write_log();
    write_head();
    install_trans();
    flushed = log.lh.n;
    log.lh.n = 0;
    write_head();
  }

  acquire(&log.lock); //* acquire the log lock
  //* release & acquire are implemented in here based on original end_op call.
  //* without these calls, it will panic ("sched locks"), which is a deadlock in xv6.
  log.committing = 0;
  log.flushing = 0; //* Flush end; Now another sync() call will be allowed.

  return flushed;
}
```

flush() – flush all buffer if exist.

Buffer 에 dirty block 이 존재할 시 전체를 flush 하는 flush()에 대한 서술입니다. flush()는 end_op 와 commit 을 참고하여 구현하였습니다. 기존 end_op()에서 commit 을 하기 전, lock 을 release 하고 commit 이 끝난 후 다시 lock 을 acquire 하는 것을 참고하여 똑같이 작동하게 했습니다. (이 mechanism 이 없으면 sched locks 라는 panic()이 나오게 됩니다.)

Flushing 전 후에는 log.flushing 의 값을 각각 1 (enable), 0 (disable)을 해주며, 만약 dirty block 이 존재한다면, 기존의 commit()과 동일한 routine 을 실행합니다.

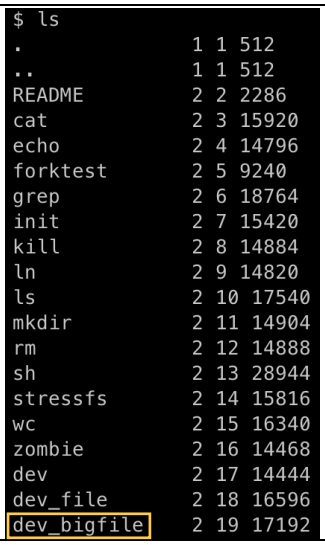
이상으로 sync (Buffered I/O) 구현에 대한 서술을 마무리하겠습니다.

V. Result Preview

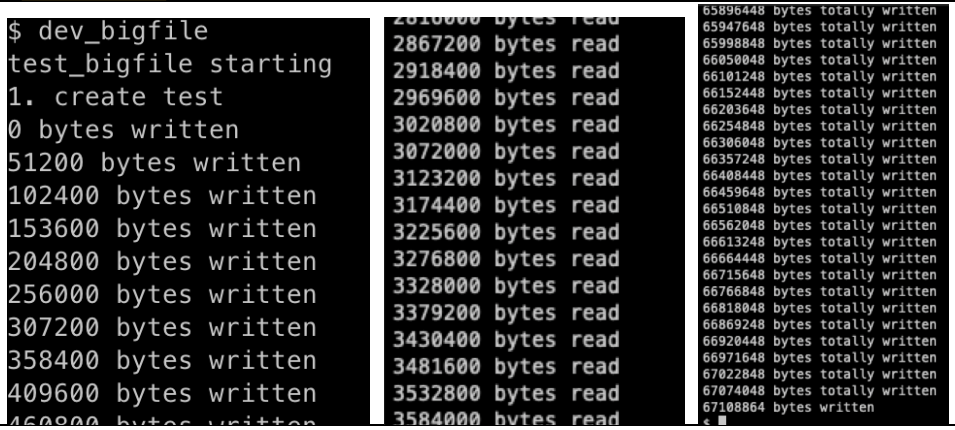
Project #3 의 Multi-Indirect, Symbolic Link, Sync (Buffered I/O)에 대한 기능 시연입니다.

I. Multi-Indirect

Multi-Indirect



- dev_bigfile 을 실행함으로 Multi-Indirect 에 대한 테스트를 실행합니다.
- dev_bigfile 은 약 16MB 정도의 대용량 데이터를 write&read 하는 테스트 코드로써, Goolge 에서 관련 코드를 찾아 테스트를 진행하였습니다.



- 전체 데이터가 성공적으로 write/read 가 실행되는 모습을 확인할 수 있습니다.
- 이로서 Double Indirect 와 Triple Indirect 까지 정상적으로 실행됨을 확인할 수 있습니다.

II. Symbolic Link

<table>
<tr><th colspan="2">Symbolic Link</th></tr>
<tr><td>

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

int
main(){
  printf(0, "Hello Link!\n");

  exit();
}
$ dev
Hello Link!
$
```

</td><td>

- dev.c 의 구조입니다. “Hello Link”를 호출하는 아주 간단한 User Program 입니다.

</td></tr>
<tr><td>

```
$ ln -s dev dev2
$ ls
.               1 1 512
..              1 1 512
README          2 2 2286
cat             2 3 15920
echo            2 4 14796
forktest        2 5 9240
grep            2 6 18764
init            2 7 15420
kill            2 8 14884
ln              2 9 14820
ls              2 10 17540
mkdir           2 11 14904
rm              2 12 14888
sh              2 13 28944
stressfs        2 14 15816
wc              2 15 16340
zombie          2 16 14468
dev             2 17 14444
dev_file        2 18 16596
dev_bigfile     2 19 17192
dev_test        2 20 16064
console         3 21 0
dev2            4 22 8 (Symbolic Link)
$ dev2
Hello Link!
$
```

</td><td>

- ln -s dev dev2 를 통해 dev 에 대한 symbolic link 를 dev2 라는 이름으로 실행합니다.
- 이후 ls 를 실행하면, 새로 형성된 dev2 가 보여지고, Symbolic Link 임이 명시되어 있습니다.

</td></tr>
<tr><td>

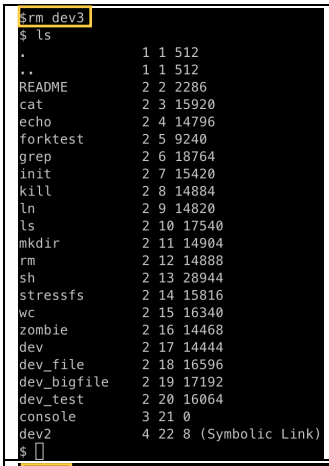
```
$ ln -s dev2 dev3
$ ls
.               1 1 512
..              1 1 512
README          2 2 2286
cat             2 3 15920
echo            2 4 14796
forktest        2 5 9240
grep            2 6 18764
init            2 7 15420
kill            2 8 14884
ln              2 9 14820
ls              2 10 17540
mkdir           2 11 14904
rm              2 12 14888
sh              2 13 28944
stressfs        2 14 15816
wc              2 15 16340
zombie          2 16 14468
dev             2 17 14444
dev_file        2 18 16596
dev_bigfile     2 19 17192
dev_test        2 20 16064
console         3 21 0
dev2            4 22 8 (Symbolic Link)
dev3            4 23 9 (Symbolic Link)
$ dev3
Hello Link!
$
```

</td><td>

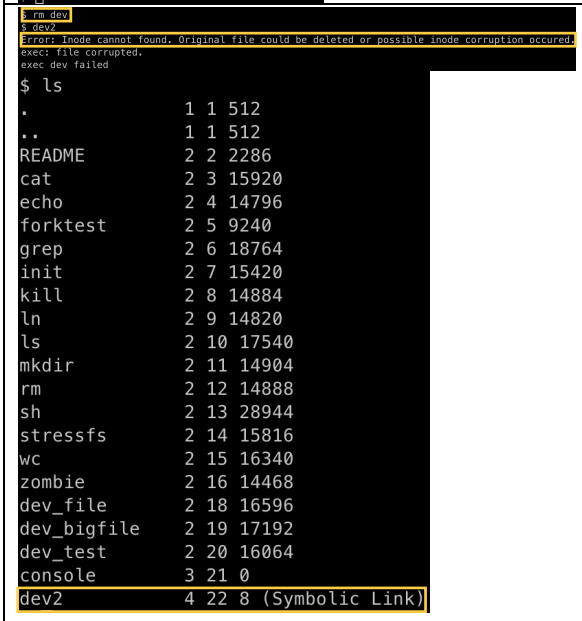
- ln -s dev2 dev3 를 통해 dev2 라는 symbolic link 에 대한 symbolic link 를 형성합니다.
- 이 또한 오류 없이 형성이 되었고, ls 호출 시 dev3 역시 보여지게 되며 Symbolic Link 임이 보여집니다.
- dev3 역시 실행 시 문제 없이 실행됩니다.
- Symbolic Link 에 대한 rm 또한 무리 없이 실행됩니다.

</td></tr>
</table>

<table>
<tr>
<td>



</td>
<td></td>
</tr>
<tr>
<td>



</td>
<td>

- 다음과 같이 original file 이 삭제된 경우, symbolic link 인 dev2 를 실행하려고 할 시 다음과 같은 Error 를 발생시킵니다.
- 이는 Original File 의 path 정보가 바뀌었을 때에도 발생하며, 모든 Operation 에 대해 error 를 출력합니다.
- 하지만 예외적으로 ls 에서의 open 은 허용합니다. Original File 이 삭제 되었을 때 모든 symbolic link 를 지우는 것은 바람직 하지 않다고 생각을 하였고, 또한 접근할 수 없다고 ls 에서도 해당 파일을 명시하지 않으면 유저가 알 수 있는 방법이 없을 것이라고 판단했습니다. 그러므로, ls 에 대해서는 original file 을 찾을 수 없는 symbolic link 의 정보 또한 출력하여, 유저가 이를 지울 수 있도록 합니다. 물론, 저 파일을 실행하려 하거나, 어떤 file operation 을 시도하려고 하면 동일한 에러 메세지를 출력하게 됩니다.

</td>
</tr>
</table>

III. Sync (Buffered I/O)

<table>
<tr><th colspan="2">Sync (Buffered I/O)</th></tr>
<tr><td>

```
int
sys_sync(){
  if(log.flushing == 1){
    //* Currently flushing buffers!
    cprintf("sync: busy!\n");
    return -1;
  }

  int ret = sync();
  cprintf("flushed: %d\n", ret);
  return ret;
}
```

```
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 - 2){
      // this op might exhaust log space; wait for commit.
      //sleep(&log, &log.lock);
      cprintf("Buffer Full! Flusing...\n");
      sync();
    } else {
      log.outstanding += 1;
      release(&log.lock);
      break;
    }
  }
}
```

</td><td>

- sync()에서 flush 되는 block 의 수를 출력하도록 잠시 수정했습니다.
- begin_op()에서 buffer 가 full 되었을 시, “Buffer Full! Flushing…”을 출력하도록 합니다.

</td></tr>
<tr><td>

```
void
save(void)
{
    int fd;
    char* str = "Hello World!!!!!\n";

    fd = open("filetest", O_CREATE | O_RDWR);
    if(fd >= 0) {
        printf(0, "ok: file creation succeed\n");
    } else {
        printf(0, "error: file creation failed\n");
        exit();
    }

    int size = sizeof(str);
    if(write(fd, &str, size) != size){
        printf(0, "error: write failed\n");
        exit();
    }
    printf(0, "write ok\n");
    sync();
    close(fd);
}
void
modify(void)
{
  int fd;
  char* str = "New String\n";

  fd = open("filetest", O_WRONLY);

  if(fd < 0){
    printf(0,"error: write failed in modify\n");
  }

  int size = sizeof(str);

  if(write(fd, &str, size) != size){
    printf(0,"write failed in modify.\n");
    exit();
  }

  printf(0, "write ok\n");
  sync();
  close(fd);

}
```

</td><td>

- dev_file.c 는 save, modify, load 와 같은 file read & write 작업을 실행합니다.
- 각 호출 마지막에 sync()를 호출하여 system call 와 flush 가 잘 호출되는지 확인합니다.

</td></tr>
</table>

```
void
load(void)
{
    int fd;
    char* str;

    fd = open("filetest", O_RDONLY);
    if(fd >= 0) {
        printf(0, "ok: read file succeed\n");
    } else {
        printf(0, "error: read file failed\n");
        exit();
    }

    int size = sizeof(str);
    if(read(fd, &str, size) != size){
        printf(0, "error: read string in file failed\n");
        exit();
    }
    printf(0, "file content (string): %s", str);
    printf(0, "read ok\n");
    close(fd);
}
```

```
int
main(void)
{
    save();
    modify();
    load();

    exit();
}
```

```
$ dev_bigfile
test_bigfile starting
1. create test
0 bytes written
Buffer Full! Flusing...
Buffer Full! Flusing...
Buffer Full! Flusing...
Buffer Full! Flusing...
Buffer Full! Flusing...
51200 bytes written
Buffer Full! Flusing...
Buffer Full! Flusing...
Buffer Full! Flusing...
```

```
$ dev_file
ok: file creation succeed
write ok
flushed: 1
write ok
flushed: 1
ok: read file succeed
file content (string): New String
read ok
$
```

- 먼저 dev_bigfile 을 호출하여 큰 파일을 write&read 할 경우, buffer full 이 되었을 경우, sync()가 호출되는 것이 확인되었습니다.
- 또한, dev_file 을 호출하였을 경우, sync()를 통해 변경사항이 적용된 점과 (“Hello World!”가 아닌 “New String”출력), sync()가 호출된 후 몇개의 block 이 flush 가 되었는지 출력하는 것을 볼 수 있습니다.

- sync()를 호출하지 않았을 때 변경사항이 적용되지 않는 테스트도 진행을 해야하나, 기존 xv6 는 disk 가 아니라 buffer 에서 데이터를 읽어와서 실제로 write 가 적용 안되어도 변경 이후의 데이터를 읽어 오게 됩니다. 이를 테스트 하기 위해서는 다른 프로세스를 통해 읽거나, file 을 따로 실제로 읽어오는 테스트 프로그램이 필요하나, 시간적 여유 부족으로 인해 해당 테스트는 skip 하게 되었습니다. 이론상 sync()이외에는 flush 하고 disk 에 데이터를 write 하는 기능은 호출되지 않으며, sync()가 호출되지 않으면 buffer 가 full 되지 않는 이상 flush 와 disk 에 write 가 일어나지 않으니, buffer 가 full 되지 않는 조건 하에 sync()가 호출이 되지 않으면 flush 와 disk 에 write 또한 일어나지 않게 될 것입니다.

VI. Trouble Shooting & Limitations

Project #3 구현 중 다음과 같은 문제점을 발견하였고, 다음과 같이 해결하였습니다

I. Multi-Indirect - MAXFILE

- Multi-Indirect 의 기능을 구현한 후, 첫 테스트 때에는 panic 이 발생했습니다. 처음에는 이 이유가 무엇인지 의문이었는데, 알고보니 MAXFILE 의 수 또한 Multi-Indirect 에서 구현되어 있는 Double ~ Triple Indirect 의 range 까지 cover 해야 한다는 것을 알게 되었습니다.
  - Before: #define MAXFILE (NDIRECT + NINDIRECT)
  - Updated: #define (NDIRECT + NINDIRECT + DINDIRECT + TINDIRECT)
- 다음과 같이 바꾸니 바로 해결이 되었습니다.

II. Symbolic Link - ls

- 기존에는 O_NFSBLK 라는 open mode 를 만들지 않았습니다. 즉, ls 에서 발생하는 open 또한 symbolic link 에 대한 redirection 을 실행을 했습니다. 그러자, 다음과 같은 현상이 발견되었습니다.
  - ln -s dev dev2 를 통해 dev2 라는 symbolic link 를 형성합니다.
  - ls 호출 시, dev2 는 dev2 가 아닌 dev2 가 가리키고 있는 original file 인 dev 를 출력하게 됩니다.
- 기능상에 문제가 있는 것은 아니지만, (dev2 를 실행하면 잘 실행이 되고, symbolic link 가 요구하는 특징을 모두 가지고 있음.) 이 현상이 정상적이라고 생각이 되지 않아 O_NFSBLK 라는 mode 를 만들어, ls 의 경우에는 symbolic link 의 redirection 을 허용하지 않도록 했습니다.
- 이 결과, 위 시연에서 보인 것과 같이 ls 시 symbolic link file 은 이 link 가 가리키는 original file 이 아닌, 자기 자신이 잘 출력되게 됩니다.

III. Sync (Buffered I/O) - panic("sched locks")

- 기존 flush()에는 log.lock 에 대한 release, acquire 코드가 없었고, 완성 후 실행을 하니 해당 panic 이 계속 뜨게 되었습니다.
- Google 에 검색하니, 해당 panic 은 xv6 에서 deadlock 조건에서 나오는 panic 이라는 것을 알게 되었고, lock 이 해당 panic 의 원인일 수 있겠다라는 생각을 하였습니다.
- end_op()와 commit()의 호출 순서와 이에 얽혀있는 lock 의 acquire, release 순서를 확인하여, flush()에 해당 순서에 맞춰 lock 을 release, acquire 시켜주니 해당 panic 은 발생하지 않게 되었습니다.

IV. Sync (Buffered I/O) - Deadlock

- 위의 panic("sched locks")와 달리, logging 도중 잘 실행이 되다가 특정 조건에서 xv6 가 얼어버리는 현상이 있었습니다.
- 이유를 찾지 못하다가, 조교님을 통해 log 기본적인 log operation 을 실행하기 위해서는 1~2 칸의 해당 operation 이 실행되기 위한 spare space 가 필요하다는 것을 알게 되었습니다. 즉, buffer 가 full 되면 flush 를 진행해야 하지만, 해당 full 의 기준이 outstanding 을 포함한 수가 LOGSIZE 와 같거나 큰 것이 아닌, LOGSIZE - 2 보다 같거나 크다는 것입니다.
- 즉, log buffer 에 있는 block + outstanding 이 LOGSIZE - 2 보다 같거나 클 때 Flush 를 하도록 설정했습니다. (LOGSIZE - 1 을 해도 문제는 없을 것 같지만, 최대한 안전하게

구현을 하고 싶었습니다.) 그 결과, xv6 가 logging 도중에 얼어버리는 현상은 일어나지 않았습니다.

Wrap Up

- 기능을 일단 모두 구현을 하긴 했으나, 완성도 면에서는 저 스스로도 만족스럽지가 않은 것 같습니다. 시간의 부족함을 많이 느낀 과제였으나, 최선을 다 했기에 후회는 되지 않았고, 가벼운 마음으로 해당 Project #3 을 마무리 짓습니다.

2019014266 임규민

ELE3021 Operating System – Project #3 Wiki

The End.