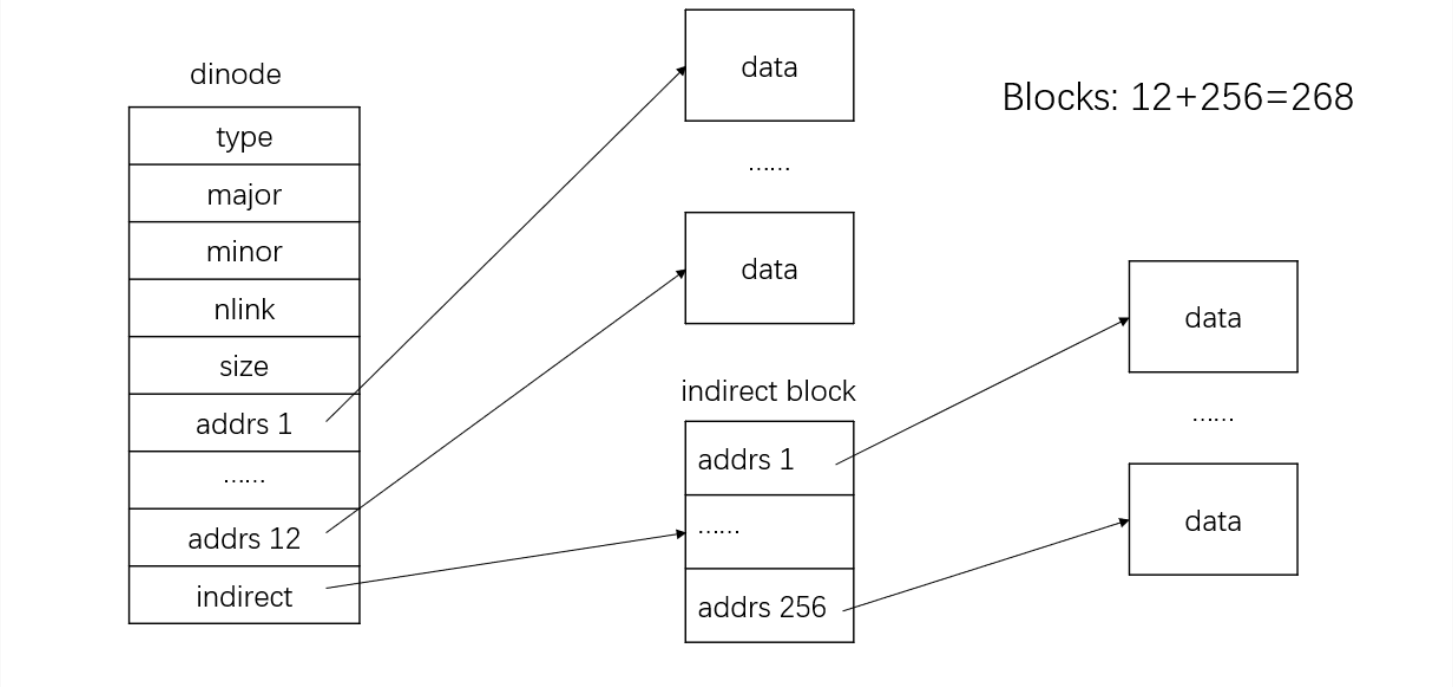
# 实验报告

## Lab 6

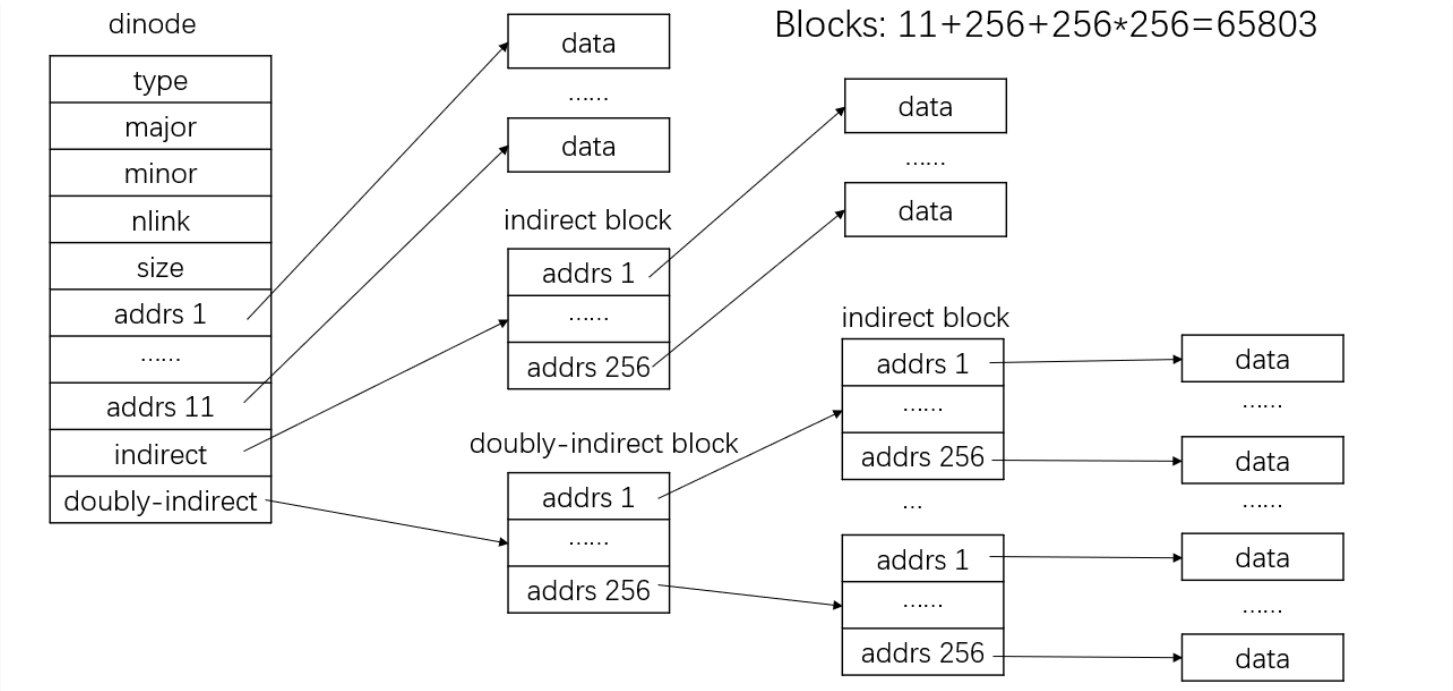
# 1. Large files

## 1.1 实验要求

我们需要增大xv6的文件的最大大小。之前xv6的文件大小最大为268块(Blocks)，其中一块是1024字节大小。这个限制是由于xv6的每个inode包含了12个直接块号和1个一级间接块号，其中这个一级间接块号指向一个包含256个块号的块，所以总共是12+256=268个块。inode结构如下图：




现在，我们需要更改xv6文件系统的相关代码，为每个inode支持二级间接块号。每个二级间接块号指向的块包含了256个一级间接块号，从而总共包含了256×256=65536个块。于是，现在每个文件的最大大小就变为11+256+65536=65803个块。这里，我们将12个直接块号的其中一个更改为了二级间接块号。更改后，inode结构如下图：




我们需要更改`fs.c`的`bmap()`函数以实现二级间接块。将一个直接块更改为一个二级间接块，且不能更改inode的大小。`ip->addrs[]`的前11个数组应该是直接块，第12个是一级间接块，第13个就应该是二级间接块。

# 1.2 实验基础

`mkfs`创建了xv6文件系统的磁盘镜像，并决定了文件系统总共的块数。文件系统的块数被定义于`kernel/param.h`的`FSSIZE`。目前，`FSSIZE`的值为200000。这200000个块中，有70个是元数据块，剩余的199930个都是数据块。

inode的格式定义于`fs.h`的`struct dinode`，如下：

```
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

inode在内存中的拷贝为`struct inode`，如下：

```
// 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];
};
```

如下是几个有用的参数，指示了直接块和间接块的个数。

```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))      // for xv6,it's 256.
#define MAXFILE (NDIRECT + NINDIRECT)         // 12 + 256 = 268
```

# 1.3 bmap()函数

如下是对`bmap()`函数的详细解析。

```
// Inode content
//
// The content (data) associated with each inode is stored
// in blocks on the disk. The first NDIRECT block numbers
// are listed in ip->addrs[].  The next NINDIRECT blocks are
// listed in block ip->addrs[NDIRECT].

// Return the disk block address of the nth block in inode ip.
// If there is no such block, bmap allocates one.
// returns 0 if out of disk space.
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;

  if(bn < NDIRECT){        // 如果需要的块是直接块(共12个)，直接查看addrs[bn]
    if((addr = ip->addrs[bn]) == 0){    // 如果没有这个块就分配一个
      addr = balloc(ip->dev);
      if(addr == 0)        // 分配失败
        return 0;
      ip->addrs[bn] = addr;
    }
    return addr;            // 返回该直接块的地址
  }
  bn -= NDIRECT;            // 不是直接块，减去直接块数

  // 是否是间接块(所指向的块,共256块)，
  // 如果是，查看间接块(addrs[NDIRECT])
  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0){    // 如果没有，也分配一个间接块
      addr = balloc(ip->dev);
      if(addr == 0)
        return 0;
      ip->addrs[NDIRECT] = addr;
    }
    bp = bread(ip->dev, addr);        // addr是间接块的地址，需要读取该块进一步查找
    // 获取间接块buffer：bp
    a = (uint*)bp->data;            // 对应块的数据(数组)
    if((addr = a[bn]) == 0){        // a[bn]是需要的磁盘块的地址
```

```
      addr = balloc(ip->dev);
      if(addr){
        a[bn] = addr;             // 分配成功，写入地址，
        log_write(bp);            // 将buffer写入log(见xv6 book 8.4-8.6),不展开
      }
    }
    brelse(bp);                 // 释放buffer的锁
    return addr;                // 返回获得的地址
  }

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

对12个直接块的读取很简单，后面256个间接块对应块的读取就不那么好理解了。为了理解`bmap()`的工作原理，我们还得进一步看看`bread()`函数的工作原理。

```
// Return a locked buf with the contents of the indicated block.
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;

  b = bget(dev, blockno);
  if(!b->valid) {
    virtio_disk_rw(b, 0);
    b->valid = 1;
  }
  return b;
}
```

`bread()`函数返回`buf*`类型。`buf`定义于`kernel/buf.h`，是xv6文件系统的第二层(见xv6 book 8.1)，它是一个双向链表，有一些相关的属性，每个链表节点的buffer对于着一个磁盘块的缓存，方便在内存中操作。其中我们关心的是它的`data`数组，保存的是磁盘块的数据。需要注意的是`data`数组类型为`uchar`，而`uchar`类型变量大小为1B，因此后面将`data`数组作为`uint`数组使用，相当于实际大小为1024/4=256。

```
struct buf {
  int valid;   // has data been read from disk?
  int disk;    // does disk "own" buf?
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
  struct buf *prev; // LRU cache list
  struct buf *next;
  uchar data[BSIZE];        // data[1024]
};
```

`bread()`函数通过`bget()`函数获取一个buffer，检查该buffer是否是对应磁盘块的拷贝，如果不是还要通过`virtio_disk_rw`去磁盘上读相应内容。之后返回这个buffer。

`bget()`函数根据给定的设备号`dev`和块号`blockno`扫描整个buffer链表，找到之后返回这个buffer。一些实现细节如该buffer是否已经被缓存，如果没有缓存还要进行LRU替换，不作展开。

因此，`bmap()`函数通过`bread()`函数获取了需要的间接块对应的buffer，这个buffer的data的内容就是间接块上保存的指向的块的地址。通过bn（已经减去了`NDIRECT`）就能得到对应块的磁盘地址。综上，`bmap()`函数接收一个`inode`和需要的块号，就能获得对应块的磁盘地址。

# 1.4 实现doubly-indirect block

因此，我们可以完全按照一级间接块实现的方式，类似地构建二级间接块。

首先，修改相应的定义:`NDIRECT`需要改为11，因为少了一个拿来当二级间接块了（实验要求不能修改`struct dinode`的大小)，同时新增参数`NDOUBLY_INDIRECT`指示二级间接块负责的块数，其值为256×256=65536。

一些相应的定义也要修改，如`MAXFILE`，如`struct dinode`和`struct inode`。

```
// kernel/fs.h
// my edit
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))    // 256
#define NDOUBLY_INDIRECT ((NINDIRECT) * (NINDIRECT))    // 65536
#define MAXFILE (NDIRECT + NINDIRECT + NDOUBLY_INDIRECT)  //65803

// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+2];   // Data block addresses
};
// edit ends

// kernel/file.h
// my edit
// 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+2];
};
// edit ends
```

接下来，修改`bmap()`函数：

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

  // DIRECT BLOCKS
  if(bn < NDIRECT){     // 0~10, total 11 blocks
    if((addr = ip->addrs[bn]) == 0){
      addr = balloc(ip->dev);
      if(addr == 0)
        return 0;
      ip->addrs[bn] = addr;
    }
    return addr;
  }
  bn -= NDIRECT;

  // SINGLY-INDIRECT BLOCKS
  if(bn < NINDIRECT){   // 11~267, total 256 blocks
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0){
      addr = balloc(ip->dev);
      if(addr == 0)
        return 0;
      ip->addrs[NDIRECT] = addr;
    }
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      addr = balloc(ip->dev);
      if(addr){
        a[bn] = addr;
        log_write(bp);
      }
    }
    brelse(bp);
    return addr;
  }

  // my edit
```

```
  bn -= NINDIRECT;

  // DOUBLY-INDIRECT BLOCKS
  if(bn < NDOUBLY_INDIRECT)   // 268~65802, total 65536 blocks
  {
    if((addr = ip->addrs[NDIRECT + 1]) == 0)
    {
      addr = balloc(ip->dev);
      if(addr == 0)
        return 0;
      ip->addrs[NDIRECT + 1] = addr;
    }
    // first disk read
    bp = bread(ip->dev,addr);
    a = (uint*)bp->data;
    if((addr = a[bn / NINDIRECT]) == 0)
    {
      addr = balloc(ip->dev);
      if(addr)
      {
        a[bn / NINDIRECT] = addr;
        log_write(bp);
      }
    }
    brelse(bp);

    // second disk read
    bp = bread(ip->dev,addr);
    a = (uint*)bp->data;
    if((addr = a[bn % NINDIRECT]) == 0)
    {
      addr = balloc(ip->dev);
      if(addr)
      {
        a[bn % NINDIRECT] = addr;
        log_write(bp);
      }
    }
    brelse(bp);
    return addr;
  }
  // edit ends
```

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

前面没有需要改的地方。处理二级间接块时，按理来说只需要将处理一级间接块的逻辑用两次即可。但需要注意一点：处理一级间接块时，只有这一个间接块，因此bn能够对应上这个间接块对应的256个块；但现在是二级间接块，我们知道的是bn，这是所有65536个最外层块的编号，如果要找对应的一级间接块编号，还要先除以`NINDIRECT`。同样的，对于一级间接块，编号范围也在0~255之间，bn所对应的内部编号应该是`bn % NINDIRECT`。

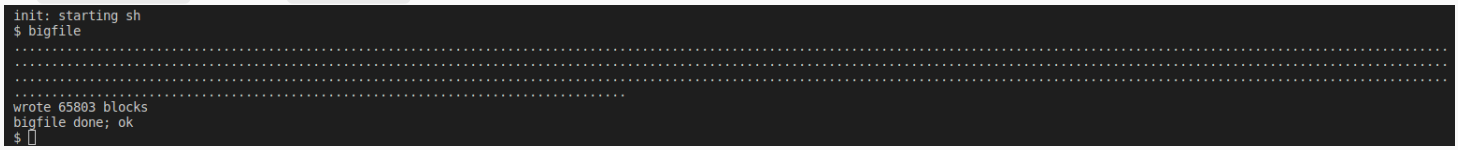
按照提示，还需要修改`itrunc()`函数。这个函数用来清空一个`inode`的所有内容。也很简单，仿照上面一样的写法就行。不过需要注意逐层清空，先清空外层的65536个实际块，再清空256个一级间接块，最后清空1个二级间接块。

```
// Truncate inode (discard contents).
// Caller must hold ip->lock.
void
itrunc(struct inode *ip)
{
  int i, j;
  struct buf *bp;
  uint *a;

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

  // SINGLY-INDIRECT BLOCKS
  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;
  }

  // my edit
  struct buf* bp2;
  uint *a2;
  int k;
  // DOUBLY-INDIRECT BLOCKS
  if(ip->addrs[NDIRECT + 1])
  {
    // buffer of doubly-indirect block
    bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT; j++)
```

```
    {
      // a[j] is an address of a singly-indirect block(totally 256 blocks)
      if(a[j])
      {
        // buffer of singly-indirect block
        bp2 = bread(ip->dev, a[j]);
        a2 = (uint*)bp2->data;
        for(k = 0; k < NINDIRECT; k++)
        {
          // a2[k] is an address of outest block(totally 65536 blocks)
          if(a2[k])
            bfree(ip->dev,a2[k]);
        }
        brelse(bp2);
        bfree(ip->dev, a[j]);
      }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT + 1]);
    ip->addrs[NDIRECT + 1] = 0;
  }
  // edit ends

  ip->size = 0;
  iupdate(ip);
}
```

# 1.5 实验结果

`make qemu`后输入`bigfile`，结果如图:




输入`usertests -q`，结果如图:

```
OK
test textwrite: usertrap(): unexpected scause 0x000000000000000f pid=6452
            sepc=0x0000000000002492 stval=0x0000000000000000
OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=6455
            sepc=0x0000000000005c62 stval=0x0000000000005c62
usertrap(): unexpected scause 0x000000000000000c pid=6456
            sepc=0x0000000000005c62 stval=0x0000000000005c62
OK
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
ALL TESTS PASSED
$
```

# 2. Symbolic links

## 2.1 实验要求

符号链接能通过路径名获取一个链接的文件。当符号链接被打开时，内核跟随链接至指定的文件。符号链接与硬链接很相似，但硬链接被限制只能指向在同一硬盘上的文件，但符号链接可以穿越磁盘设备。

我们需要为xv6文件系统添加符号链接(软链接)方式，通过实现`symlink()`系统调用。该系统调用会创建一个位于`path`的符号链接（本质上，就是一个字符串）指向路径`target`。

## 2.2 准备工作

先在`Makefile`中添加symlinktest。

```
UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $U/_grep\
    $U/_init\
    $U/_kill\
    $U/_ln\
    $U/_ls\
    $U/_mkdir\
    $U/_rm\
    $U/_sh\
    $U/_stressfs\
    $U/_usertests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_symlinktest\
```

然后按照标准流程注册一个系统调用，如下:

```
// 1. user/user.h
// my edit
int symlink(char*,char*);
// edit ends
```

注意这里`sigalarm`的第二个参数是一个函数指针`void(*)(void)`类型。

```
# 2. user/usys.pl
# my edit
entry("symlink");
# edit ends
```

```
// 3. kernel/syscall.h
// my edit
#define SYS_symlink 22
// edit ends
```

```
// 4. kernel/syscall.c
// Prototypes for the functions that handle system calls.
...
// my edit
extern uint64 sys_symlink(void);
// edit ends

// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = {
...
[SYS_symlink] sys_symlink,
};
```

```
//5. kernel/sysfile.c
// my edit
uint64 sys_symlink(void)
{

}
// edit ends
```

按照提示，在`kernel/stat.h`中增加新文件类型`T_SYMLINK`代表符号链接。

```
// my edit
#define T_SYMLINK 4   // Symbolic link
// edit ends
```

在`kernel/fcntl.h`中添加新flag `O_NOFOLLOW`。由于flag会参与位或运算，所以新flag不应与已存在的flag产生重叠。

现有的五个flag分别是:

```
#define O_RDONLY  0x000     // 0000 0000 0000
#define O_WRONLY  0x001     // 0000 0000 0001
#define O_RDWR    0x002     // 0000 0000 0010
#define O_CREATE  0x200     // 0010 0000 0000
#define O_TRUNC   0x400     // 0100 0000 0000
```

可以令`O_NOFOLLOW`=0x004，只要不重叠就行。这样`user/symbolic.c`能够成功被编译。

```
// my edit
#define O_NOFOLLOW 0x004    // 0000 0000 0100
// edit ends
```

## 2.3 实现`sys_symlink()`函数

首先需要了解初始函数原型`symlink()`。这个函数原型为`int symlink(char* target,char* path)`，作用是在`path`路径创建一个指向`target`路径的文件的符号链接。

首先获取参数。可以参考上面`sys_link()`的实现方式。

```
// my edit
uint64 sys_symlink(void)
{
  char target[MAXPATH],path[MAXPATH];
  if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
    return -1;
}
```

接下来具体实现`sys_symlink()`函数。在上面的`Large files`实验我们知道，inode中保存了一个`addrs`数组，内容是对应文件的数据块的磁盘地址（或间接块的磁盘地址），之后可以通过`bread()`函数，获取对应磁盘地址的buffer，从而查看需要的文件数据。`symlink()`函数做的工作很简单，在`path`创建一个指向`target`的符号链接，并将这个符号链接保存到对应文件的inode上。

根据xv6 book，先给出一些后面会用到的函数：

1. `create()`：为一个新inode，创建对应的文件名。具体细节上不展开，大致上，该函数检查对应路径是否存在，如果不存在就创建一个新inode。
2. `writei()`：向目标inode写入内容。具体地，从指定的off开始写入n字节的内容。
3. `begin_op()`和`end_op()`：用于保持一致性的log，在每个文件系统调用开头/结尾调用。
4. `ilock()`和`iunlock()`：将对应的inode加锁、解锁。
5. `iput()`：释放inode，并将对应inode的内容写回磁盘。(调用了`itrunc()`函数和`iupdate()`函数完成这两项功能)

基于上述函数，可以写出`sys_symlink()`函数:

```
// my edit
uint64 sys_symlink(void)
{
  char target[MAXPATH],path[MAXPATH];
  if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
    return -1;

  // create a new inode whose type is symbolic link
  struct inode* ip;
  begin_op();
  if((ip = create(path,T_SYMLINK,0,0)) == 0)
  {
    end_op();
    return -1;
  }

  // store the target into the inode we create
  if(writei(ip,0,(uint64)target,0,MAXPATH) < MAXPATH)
  {
    end_op();
    return -1;
  }

  // notice that create() has already lock the inode we create,so don't need to call ilock() ag
  iunlockput(ip);
  end_op();
  return 0;
}
// edit ends
```

首先调用`create()`函数在`path`创建符号链接类型的新inode，然后将`target`地址的内容(即`target`字符串)写入到inode `ip`上。最后，解锁、释放并更新`ip`(在`create()`时已经对`ip`上了锁，因此需要释放)。

## 2.4 修改`sys_open()`函数

根据提示，我们需要修改`open()`函数对应的系统调用以处理新添加的符号链接类型。我们想一想符号链接应该怎样处理：对应`T_SYMLINK`类型的inode，其数据应该是另一个目录`target`，得到这个`target`后再调用相应函数查找对应的inode，从而完成打开文件的工作。

这个对应的函数就是`namei()`。它的作用是获取路径对应的inode。对应的实现于`namex()`函数中，其中使用了`dirlookup()`和`skipelem()`函数，分别用于查找目录和解析目录，细节不作展开。

`sys_open()`函数的具体实现如下：

```
uint64
sys_open(void)
{
  ...
  // my edit
  // if the inode we open corresponds to a symbolic link file
  // and hope to follow the symbolic link
  if(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW))
  {
    int depth = 0;      // count for recursive depth
    char target[MAXPATH];
    // recursively follow
    while(ip->type == T_SYMLINK)
    {
      // depth greater than 10,must stop
      if(depth >= 10)
      {
        iunlockput(ip);
        end_op();
        return -1;
      }
      depth += 1;
      // get the content of inode:should be a path name
      readi(ip,0,(uint64)target,0,MAXPATH);
      iunlockput(ip);
      // find the inode that the path name points to.
      // if cannot,end
      if((ip = namei(target)) == 0)
      {
        end_op();
        return -1;
      }
      ilock(ip);
    }
    // else,just open the file directly.it will be done below.
  }
  // edit ends
  ...
}
```

当检测到inode类型为符号链接且omode不标识NOFOLLOW时（如果标识NOFOLLOW，表示需要直接打开这个符号链接，所以什么都不用做，这些工作在`sys_open()`后面的部分统一完成），此时对符号链接进行处理。

最外层是一个while循环，检测进一步获得的inode是不是还是符号链接，直到不是符号链接，或循环深度超过10时终止。如果`depth>=10`，表明很可能产生了链接循环，应该报错。

进入循环后，首先使用与之前的`writei()`函数对应的`readi()`函数，在相同的位置读出之前写入的路径`target`。此时需要执行`iunlockput()`函数，因为现在这个inode已经不需要了。然后用`namei()`函数获取`target`对应的inode，为这个inode上锁，进入下一次循环条件的判断。

# 2.5 实验结果

`make qemu`后输入`symlinktest`，结果如图：

```
init: starting sh
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
$
```

执行`usertests -q`的结果如图：

```
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=6457
            sepc=0x0000000000005c62 stval=0x0000000000005c62
usertrap(): unexpected scause 0x000000000000000c pid=6458
            sepc=0x0000000000005c62 stval=0x0000000000005c62
OK
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
ALL TESTS PASSED
```

退出后`make grade`的结果如图：

```
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (294.1s)
== Test time ==
time: OK
Score: 100/100
breezer@ubuntu:~/xv6-labs6-2022$
```

# 3. 实验总结

本次实验主要研究xv6的文件系统，从增长最大文件size和添加符号链接两个实验入手。这两个实验都是着眼于inode层的，而xv6文件系统总共分为7层，所以对于较低层(如log层、buffer层)的实现缺少了解，在使用相应函数完成实验时也需要花更多时间在xv6 book上查找更多相关资料。这些底层实现实际上是比较复杂的，需要花很多时间理解，不过要完成这两个实验的话实际上并不需要彻底理解底层，只需要大致了解函数抽象功能即可，总体来说难度中规中矩。