Lab 4: File System - Alejandro Vargas (avarg116)

Bigfile

Disk layout strategies like contiguous allocation, linked structures, and index structures improve file system design by modifying how files are stored, accessed, and managed on the disk. The UNIX inode is an indexed data structure for files that stores metadata including pointers to data blocks. In this lab, we are interested in increasing the bigfile or maximum file size by implementing a doubly-indirect block in the inode structure. This adds more indirection as each entry points to a single-indirect block (256 data blocks). As a result, it creates a hierarchical structure that efficiently organizes a large number of data blocks which is good for long data files. In xv6, the existing bigfile is limited to 268 blocks but we want 65803. The bmap function which is responsible for mapping logical block numbers to disk block numbers must be modified as well.

Hierarchy:

Direct blocks
- Singly-indirect
  - Direct blocks
- Doubly-indirect
  - Singly-indirect
    - Direct blocks

```
#define NDIRECT 11 //allocate room for doubly-indirect block
#define NINDIRECT (BSIZE / sizeof(uint))
#define NDOUBLY_INDIRECT (NINDIRECT*NINDIRECT) //(256*256)
#define MAXFILE (NDIRECT + NINDIRECT + NDOUBLY_INDIRECT) //65803
```

fs.h: first, we change NDIRECT 11 to 12 to allocate space for the doubly-indirect block without increasing the size of the on-disk inode. NDOUBLY_INDIRECT represents the number of block addresses in a doubly-indirect block (256*256). MAXFILE is updated to add NDOUBLY_INDIRECT (11+256+(256*256) = 65803).

file.h: we changed the definition of NDIRECT so we add 2 to addrs[] in struct inode. The same is done for struct dinode in fs.h.

```
bn -= NINDIRECT;
uint double_block = (bn / NINDIRECT); //index doubly-direct block

if(bn < NDOUBLY_INDIRECT){
  //load doubly-indirect block
  if((addr = ip->addrs[NDIRECT + 1]) == 0)
    ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  if((addr = a[double_block]) == 0){
    a[double_block] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);

  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  uint disk_block = (bn % NINDIRECT); //logical block num % num block adresses in single-indirect block
  if((addr = a[disk_block]) == 0){
    a[disk_block] = addr = balloc(ip->dev); //allocate new data block
    log_write(bp);
  }
  brelse(bp);
  return addr;
}

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

bmap: similarly to NINDIRECT, we load the doubly-indirect block, allocating if necessary. bn / NINDIRECT is used to calculate the index of the doubly-indirect block. NDIRECT +1 extends addrs to include the doubly-indirect block. The next segment of code allocates the data block in the doubly-indirect block. bn % NINDIRECT is the position of the data block. It is the logical block number % number of block addresses in a single-indirect block.

Symbolic Links

A link is a pointer to a file. All named files are hard links which reference a file object. A disadvantage of hard links is that they are restricted to the same file system. Symbolic links are indirect pointers to files, allowing them to cross disk devices. In this lab, I added symbolic links to xv6.

```
//your implementation goes here //
char name[MAXPATH], path[MAXPATH];
struct inode *ip;

if(argstr(0, name, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
  return -1;

begin_op(ROOTDEV);
if((ip = create(path, T_SYMLINK, 0, 0)) == 0){ //new symbolic link inode
  end_op(ROOTDEV);
  return -1; //faliure
}

//write length and target to symbolic link inode
int path_length = strlen(name);
writei(ip, 0, (uint64)&path_length, 0, sizeof(path_length));
writei(ip, 0, (uint64)name, sizeof(int), path_length);

iunlockput(ip);
end_op(ROOTDEV);
return 0; //success
```

sys_symlink: for the symlink system call, I referenced link. First, we check if the arguments are valid. Next, we create a new symbolic link inode at the target using type T_SYMLINK. The length of the path is written to the symbolic link inode. Either a 0 (success) or -1 (failure) are returned.

struct inode: to store the target path of a symbolic link, I added || ip->type == T_SYMLINK to the existing inode's data blocks.

```
//modifications
if(ip->type == T_SYMLINK && omode != O_NOFOLLOW){ //open symlink without following symbolic link
  int depth = 0;
  for(depth = 0; depth < 10 && ip->type == T_SYMLINK; depth++){
      int path_length;
      readi(ip, 0, (uint64)&path_length, 0, sizeof(path_length)); //read path length
      readi(ip, 0, (uint64)path, sizeof(int), path_length); //read target path
      iunlockput(ip);
      if((ip = namei(path)) == 0){ //follow symbolic link using target
          end_op(ROOTDEV);
          return -1;
      }
      ilock(ip);
  }
  if(depth >= 10){ //if depth of links > threshold 10
    iunlockput(ip);
    end_op(ROOTDEV);
    return -1;
  }
}
```

sys_open: we check if the file opened is a symbolic link and the 0_NOFOLLOW flag. A loop is used to handle the case where the path refers to a symbolic link. The target path and length are read from the symbolic link inode and the target is used to follow the link to the next inode.

Finally, we return an error code if the depth of links reaches 10. This indicates that the links form a cycle.

Testing