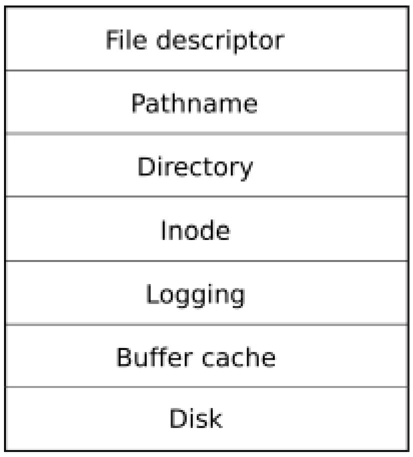
Design Description

1. Introduction

Our design of the tools to recovery program is based on the organization of the xv6 file system. By reviewing the xv6 file system (xv6: a simple, Unix-like teaching operating system chapter 8), we recognize that there are seven layers to implement the organization of the xv6 file system: file descriptor, pathname, directory, inode, logging, buffer cache and disk, see the figure below.



In the develop process, we are focusing on the layers of buffer cache, logging, inode and directory to accomplish a set of tools of recovery programs. The buffer cache is able to synchronize access to disk blocks to ensure that only one copy of a block is in memory and that only one kernel thread at a time uses that copy and cache popular blocks so that these blocks don't need to be re-read from the slow disk. The code is in `bio.c`. The logging layer is the critical parts of the recovery program on the xv6. The log stay in the fixed location, superblock. The logging layer allows higher layers to wrap updates to several blocks in a transaction. And ensures that the blocks are updated atomically the face of crashes. Xv6 solves the crashes problem by a simple form of logging. It places a description of all the disk writes it wishes to make in a log on the disk indicating that the log contains a complete operation. At that point the system call copies the writes to the on-disk file system data structures. After those writes have completed, the system erases the log on disk. The inode layer provides individual files, each represented as an inode with a unique i-number and some blocks holding the file's data. The directory layer implements each directory as a special kind of node whose content is a sequence of directory entries each of which contains a file's name and i-number.

2. Function implementation

2.1 `directoryWalker` system call

description:

A `directoryWalker` program is created in the `fs.c` to find all files and directories in a given directory and allocate the inodes of these files and directories. The program get the inode of the given path. By traversing the given directory path recursively, we record the inode number of each file and directory in an array named `directoryWalkerLog`. At the same time, we define two arrays `dirParentSet` and `fileDir` for future recovery. The former one is used to recored every directory inode number and its parent inode number, while the latter one to recored every file inode number and its directory inode number.

```
int
directoryWalker(char *path)
{
  // cprintf("path= %s", path);
  // find the inode for a given path name
  struct inode *dp = namei(path);

  // if the inode is null, means the path is invalid
  if(dp == 0)
    return -1;

  struct dirent de;
  ilock(dp);
  level++;

  if(dp->type == T_DIR) {

    for(uint offset = 0; offset < dp->size; offset += sizeof(de)) {

      // if there is a problem when reading
      if(readi(dp, (char*) &de, offset, sizeof(de)) != sizeof(de)) {
        panic("dirlookup read");
      }

      if(0 == offset && (strncmp(de.name,".", DIRSIZ) == 0)){
        struct inode* dirParent;
        char name[14] = "..";
        iunlock(dp);
        // record every directory inode number and its parents inode number
        dirParent = nameiparent(path,name);
        dirParentSet[dp->inum] = dirParent->inum;
        ilock(dp);
      }


      // find the current directory and its parent directory
      if((strncmp(de.name, ".", DIRSIZ) == 0) || (strncmp(de.name, "..", DIRSIZ)
== 0)) {

        directoryWalkerLog[de.inum] = 1;
        printSubDirInfo(level);
        cprintf("%s inode: %d\n", printName(de.name), de.inum);
        continue;
      }

      if(de.inum > 0) {
        struct inode *ip = dirlookup(dp, de.name, 0);
        ilock(ip);

        switch (ip->type) {
          case T_DIR:
            iunlock(ip);

            directoryWalkerLog[de.inum] = 1;
```

```
              printSubDirInfo(level);
              cprintf("%s inode: %d\n", printName(de.name), de.inum);

              iunlock(dp);

              char new_path[14] = {0};
              strcat(new_path,path);
              strcat(new_path,"/");
              strcat(new_path,de.name);

              directoryWalker(new_path);

              ilock(dp);
              break;

            case T_FILE:
              iunlock(ip);

              directoryWalkerLog[de.inum] = 1;
              fileDir[ip->inum] = dp->inum;
              printSubDirInfo(level);
              cprintf("%s inode: %d\n", printName(de.name), de.inum);

              break;

            case T_DEV:
              iunlock(ip);
              directoryWalkerLog[de.inum] = 1;
              break;
          }
        }
      }
    }
    level--;
    iunlock(dp);
    return 0;
  }
```

2.2 `inodeTBWalker` system call

Description:

This system call helps us to get all allocated inodes. Here we use an array named `inodeWalkerLog` to trace all allocated inodes.

```
// get all allocated inodes from buffer
int
inodeTBWalker(void) {
  for (int i = 0; i < 256; i++)
    inodeWalkerLog[i] = 0;

  struct buf *bp;
  struct dinode *dip;

  // get all allocated inodes
  for (int inum = 1; inum < sb.ninodes; inum++) {
    bp = bread(T_DIR, IBLOCK(inum, sb));
    dip = (struct dinode *) bp->data + inum % IPB;
    if (dip->type != 0 && dip->nlink > 0) {
      cprintf("Find allocoated inode: %d\n", inum);
      inodeWalkerLog[inum] = 1;
    }
    brelse(bp);
  }

  return 1;
}
```

2.3 `compareWalkersLog` system call

Description:

This system call helps us find whether two walkers return the same inode lists. In order to compare the inode results of the `directoryWalker` and `inodeTBWalker`, we create two helper function `checkInodesDir` and `checkInodes` We store the missing inode into `compareWalkersLog` for future recovery.

```
int
compareWalkers(void) {
  // inodes in both directory and buffer is free
  if ((checkInodesDir() == -1) || (checkInodes() == -1))
    return -1;

  // compare all inodes from inodeWalker and directoryWalker
  for (int i = 1; i < 256; i++) {
    // if check a difference
    if ((inodeWalkerLog[i] == 1 && directoryWalkerLog[i] == 0) ||
      (inodeWalkerLog[i] == 0 && directoryWalkerLog[i] == 1))
      cprintf("Inode#: %d missing\n", i);
    // record the missing inode for future recovery
    compareWalkersLog[i] = inodeWalkerLog[i] ^ directoryWalkerLog[i];
  }
  return 1;
}
```

2.4 `damageInode` system call

description:

The function of this system call is to help erase information in a directory inode.

```
int
damageInode(int inum) {
  if (inum <= 1) {
    cprintf("error: this is a root directory\n");
    return -1;
  }

  begin_op();
  struct inode *inodeToDel = iget(T_DIR, inum);
  if (inodeToDel->type != T_DIR) {
    cprintf("error: invalid directory\n");
    return -1;
  }
  damageDirInode[inum] = 1;

  // set locks
  ilock(inodeToDel);
  itrunc(inodeToDel);
  iunlockput(inodeToDel);
  end_op();

  cprintf("damange one inode.\n");

  // set all directoryWalkerLog to zero. We need to walker again to get an update
inode list
  for (int i = 0; i < 100; i++)
    directoryWalkerLog[i] = 0;
  return inum;
}
```

2.5 `recoverDirFile` system call

description:

The function of this system call is to help us recovery the damage directory and related files. The main idea is based on the result of `directoryWalkers`. If one or more of directory inodes of the file system is damaged, the file in this directory can not be read. Through `dirParentSet` and `firDir`, we could get basic information of a damaged inode. By linking the current damaged directory to the parent directory, we could recover the damage directory. By checking `compareWalkersLog`, we could know which inode is missing. For each missing inode, it belongs to two kind of file. One is a file and the other is a sub-directory. When it comes to recover files, we adopt a different recovery polices. If the damaged file A in the given directory is a sub-directory, we just need to `dirlink` them again. If the damage file is just a file, we check whether its directory is already in the missing inode set. If yes, we could need to hand it, otherwise, we need to `dirlink` this file to its directory.

```
void
recoverDir(int inum)
{
  begin_op();
  struct inode* dp;
  struct inode* dpp;
  dp = iget(T_DIR, inum);
  dpp = iget(T_DIR, dirParentSet[inum]);
  dirlink(dp, ".", dp->inum);
  dirlink(dp, "..", dpp->inum);
  end_op();
}

void
recover_file(int i)
{
  begin_op();
  struct inode* ip;
  char newName[14] = {0};
  strcat(newName,"recover_");
  char recoverNum[4];
  itoa(i,recoverNum);
  strcat(newName,recoverNum);
  ip = iget(1, fileDir[i]);
  // cprintf("ip.inum = %d\n", ip->inum);
  ilock(ip);
  dirlink(ip,newName,i);
  iunlock(ip);
  end_op();
}



int
recoverDirFile() {

  int missingInodeSet[256];

  for(int i = 1; i< 256;i++) {
    if(damageDirInode[i] == 1) {
      recoverDir(i);
      for(int j = 1; j< 256; j++) {
        if(dirParentSet[j] == i) {
          fileDir[j] = i;
        }
      }
    }
    missingInodeSet[i] = 0;
  }
  for(int i = 1; i <256; i++){
    // if(fileDir[i]!=0) {
    //   cprintf("%d file parent dir: %d\n", i, fileDir[i]);
    // }
    if (compareWalkersLog[i] == 1) {
      cprintf("missing inode#: %d\n", i);
      if(compareWalkersLog[fileDir[i]] != 1) {
        missingInodeSet[i] = 1;
      }
    }
  }
  for(int i =1; i < 256; i++) {
    if(missingInodeSet[i] == 1) {
      recover_file(i);
    }
  }
  return 1;
}
```