CS450
Progamming Assignment 4: Report

November 29, 2017

Ling Xing Zheng
A20343483

Venkata Krishna Chaitanya
Polavarapu
A20378279

Question 1

Extent based file system

The following files are modified to implement extent based file system.

```
fcntl.h     : Modified to include #define O_EXTENT  0x020,
                          used in create function call.
fs.c        : Modified to implement changes for extent based file system.
stat.h      : Modified to include new file type (#define T_EXT_FILE 4).
              Changes made to "struct stat" to extend the functionality of fstat()
sysfile.c   : Changes made to sys_open to appropriately pass
                          the file type during file creation.
```

Data allocation strategy

1. The idea is that we use existing inode pointers to store the (block address, length). The first 24 bits holds the address and the last 8 bits hold the length.

2. Whenever a data block of a file is needed, function `bmap` is called. This function maps the file block number to disk block number. Most of the changes are made here.

3. When ever `bmap()` is called, we first check if the requested file block is on disk. We do this by checking if the given block number is within the `[ip->addrs, ip->addr+len)`. Repeat the previous check for all the available extents in the inode. If found, we return the corresponding block address. The corresponding code is shown below (`bmap` from `fs.c`).

```
 if (ip->type == T_EXT_FILE){
387     // Check if the block is already in use
388     int i = 0;
389     uint currLen = 0;
391     while(ip->addrs[i]){
392       // Extract the last 8 bits, which gives the length
393       uint length = EXT_LEN(ip->addrs[i]);
394
395       // Check of the requested block is within the extent
396       if (bn >= currLen && bn < length + currLen){
398             addr = EXT_ADDR(ip->addrs[i]) +  bn - currLen;
399             return addr;
400       }
401
402       // if not move to next extent
403       currLen += length;
```

4. If the block is not found, we ask for a new block. If the new block is contiguous with respect to the last extent in the inode, we increase its length. If not we add a new entry in the inode address table and return the disk address. If however the entries are $\geq$ `NDIRECT` we throw a kernel `panic()`. The corresponding code of `bmap` from `fs.c`.

```
410     if(i >= NDIRECT){
411        panic("Extent file allocation exceeded\n");
412     }
414     uint newDiskBlock = balloc(ip->dev);
415     // Check if newAddr is contiguos w.r.t to the previous extent
416       uint len = EXT_LEN(ip->addrs[i-1]);
417       if( (newDiskBlock == EXT_ADDR(ip->addrs[i-1]) + len)
                && EXT_LEN(ip->addrs[i-1]) < 0xff){
419         ip->addrs[i-1]  += 1;
421       }
422       else{
423      // if not, add a new entry in the table
424       ip->addrs[i] = GEN_ADDR(newDiskBlock, 1);

426       }
427    return newDiskBlock;
```

Data de-allocation

1. When a file is to be deleted, the system call `unlink` eventually calls `itrunc` which frees up the inodes (by calling `bfree`).

2. Our changes for data de-allocation happens in `itrunc()`. If the file type is `T_EXT_FILE`, we read each extent, for each block in the extent we call `bfree()`. The code below(from`itrunc()` in `fs.c`) shows what we said above

```
  // Code to clean Extent files
467   if (ip->type == T_EXT_FILE){
468     for(i = 0; i < NDIRECT; i++){
469       if(ip->addrs[i]){
470         uint len = EXT_LEN(ip->addrs[i]);
471         for(j = 0; j < len; j++){
472           bfree(ip->dev, EXT_ADDR(ip->addrs[i]) + j);
474         }
475         ip->addrs[i] = 0;
476       }
477     }
479     return;
480   }
```

Question 2

`fstat()` Implementation details

The following files are changed for implementing changes to `fstat()`

```
fs.c  : Modified to implement changes to stati()
stat.h: Changes made to "struct stat" to extend the functionality of fstat()
```

1. The `struct stat` is changed in the following way to hold additional details for extent based files.

```
 6 struct stat {
 7   short type;  // Type of file
 8   int dev;     // File system's disk device
 9   uint ino;    // Inode number
10   short nlink; // Number of links to file
11   uint size;   // Size of file in bytes
12   struct extents_details{
13     uint addr:24, // Extent Pointers
14          len:8 ;  // Extent Length
15   }extents[12];
16 };
```

2. The `stati()` was changed to collect more information if the file is extent based.

```
505 // Copy stat information from inode.
506 // Caller must hold ip->lock.
507 void
508 stati(struct inode *ip, struct stat *st)
509 {
510   st->dev = ip->dev;
511   st->ino = ip->inum;
512   st->type = ip->type;
513   st->nlink = ip->nlink;
514   st->size = ip->size;
515
516   // Get more info for extent files
517   int i = 0;
518   if (st->type == T_EXT_FILE){
519     for(i = 0; i < NDIRECT; i++)
520     {
521       st->extents[i].addr = EXT_ADDR(ip->addrs[i]);
522       st->extents[i].len  = EXT_LEN(ip->addrs[i]);
523     }
524   }
525 }
```

Testing Extent files/`fstat()`

We write the user code `assign4.c` that calls various function including `create, write`. We also use existing utilities like `wc, rm` for our testing purposes.

Test Case 1:

The default files in xv6 system can have a maximum size of 70 KB. However extent based files hold file sizes $>$ 70 KB. We test this by writing 80KB to the file.

Test Case 2:

We ensure that the default file system is not affected by our changes by writing to 2 files each of 1KB.

Test Case 3:

We test our modified `fstat()` on extent based files to if it works as expected.

Results of Testcase 1,2,3:

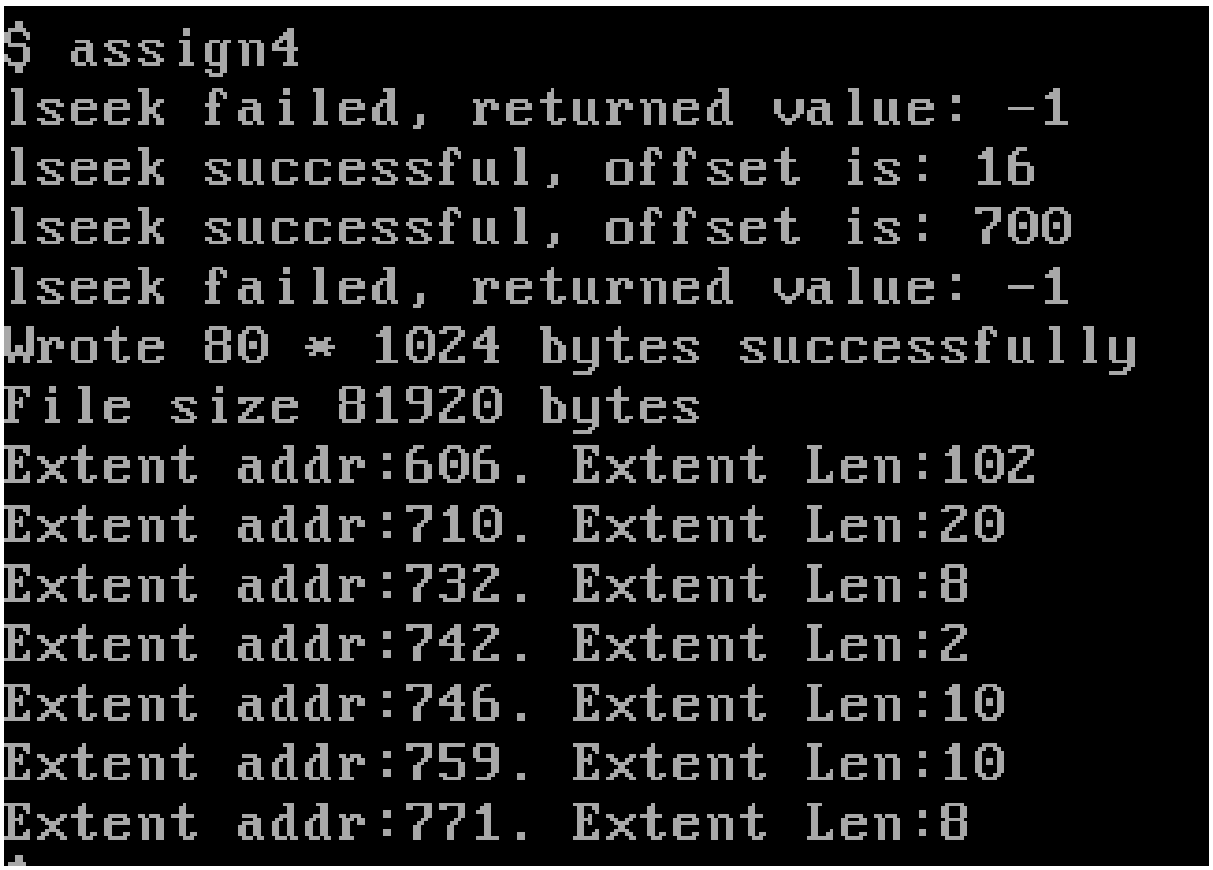

Figure 1: output of `assign4.c`: We see the output written successfully. `fstat` displaying values correctly.

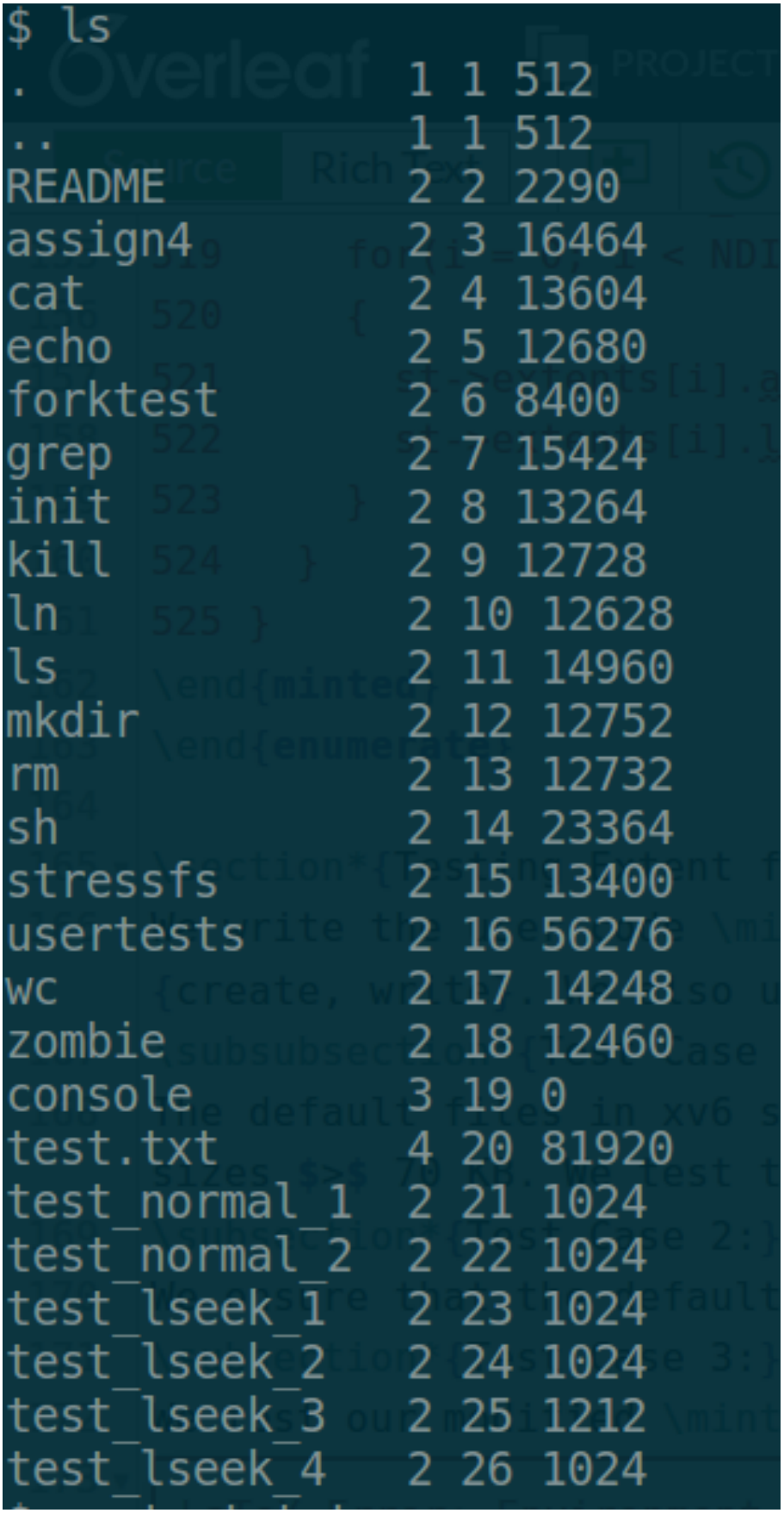

Figure 2: `ls` output showing that our file is indeed 80 KB

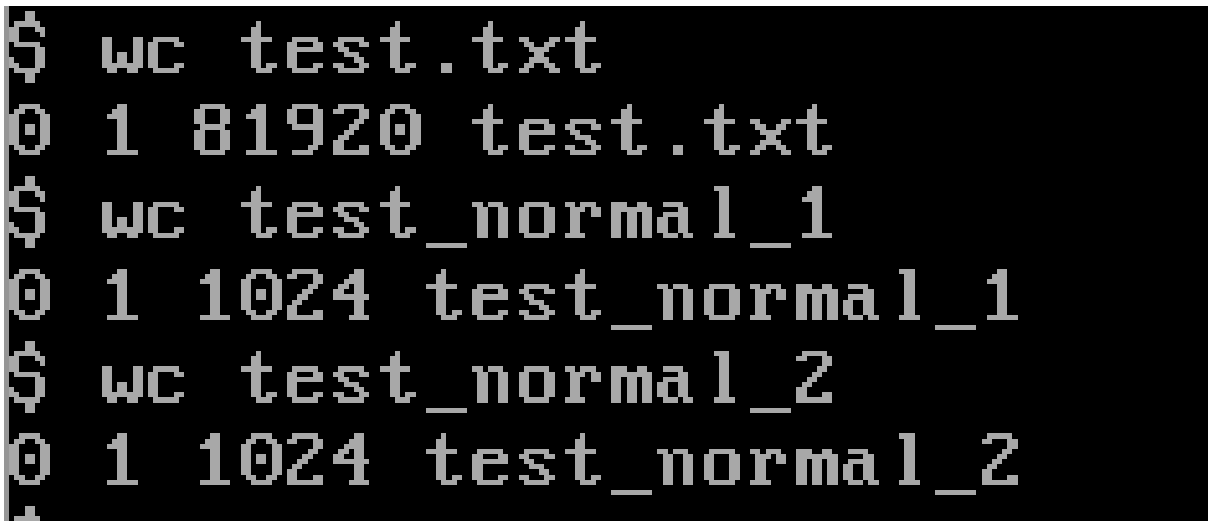

Figure 3: `wc` output showing that the existing file/ extent file read writes work as expected