7th April 2020 Lab 7 Report K. Sai Anuroop (170030035)
Mandeep Bawa (170030038)

About immediate files:

In minix, the metadata of a file is stored in form of inodes. Inodes contain information such as last access time, modification time, file size, permissions etc. along with the pointers to the disk block where the data of a file is stores. These pointers either directly refer to a disk block, or they refer to a list of additional pointers to data blocks. The problem with a regular file is that even when it is very short, a complete disk block needs to be allocated. This wastes disk space.
In immediate files, the data is stored directly in the inode instead of the disk. An inode in Minix is 64 bytes long, and 40 bytes are used to hold pointers to data blocks. When no data blocks are used, these 40 bytes can be used to store the file content directly. Thus, for files up to 40 bytes, immediate files work and hence getting rid of fragmentation. Another important thing about immediate files is that the number of disk accesses is reduced for short files and hence reducing the access time. In our implementation, we limit the size of the immediate files to 32 bytes.

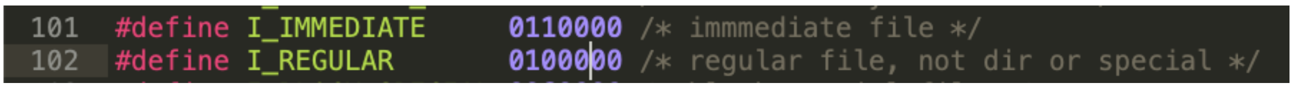
Files modified:

Following files have been modified to implement immediate files.

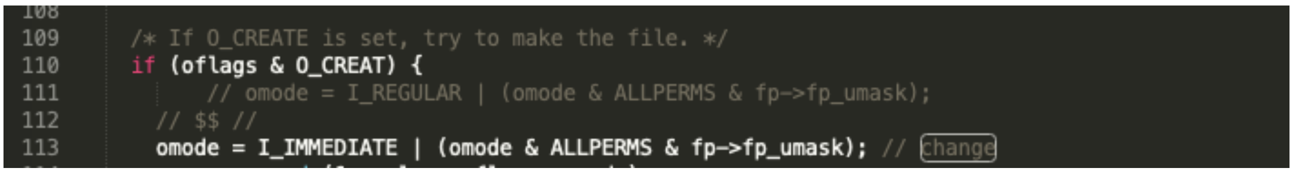
/minix/include/minix/const.h
/minix/fs/mfs/link.c
/minix/fs/mfs/misc.c
/minix/fs/mfs/read.c
/minix/fs/mfs/write.c
/minix/fs/mfs/table.c
/minix/fs/mfs/proto.h
/minix/servers/vfs/link.c
/minix/servers/vfs/open.c
/minix/lib/libc/gen/fslib.c

Here we list the major changes made in some of the above listed files.

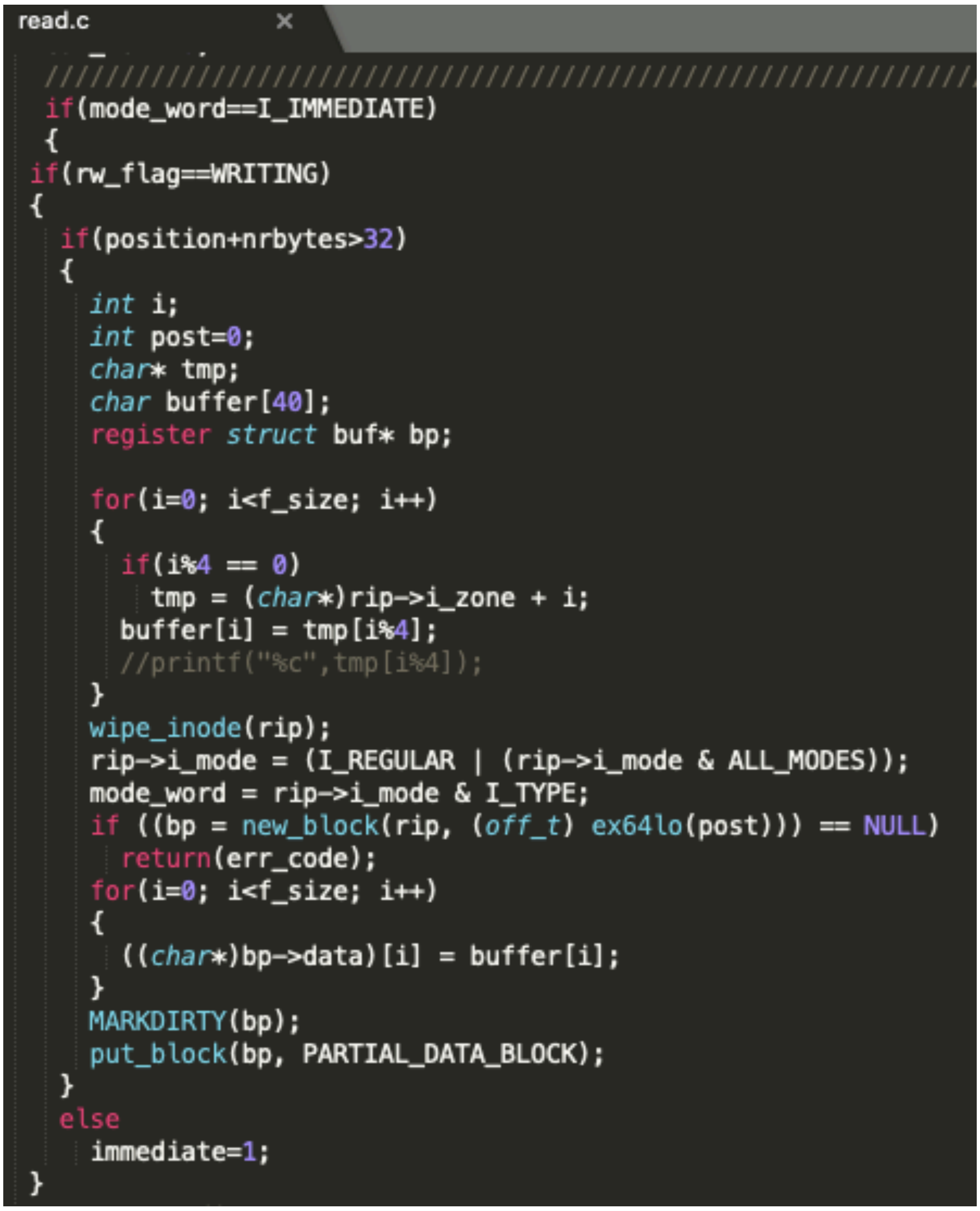
/minix/include/minix/const.h
Immediate flag is added, akin to regular flag.


```
#define I_IMMEDIATE      0110000 /* immmediate file */
#define I_REGULAR        0100000 /* regular file, not dir or special */
```

/minix/servers/vfs/open.c
In the common_open(char path[PATH_MAX], int oflags, mode_t omode) function of this file, we handle the creation of new node. We set the omode to I_IMMEDIATE, i.e., we create the file as an immediate and not a regular file. Then we do all the necessary work on the immediate file, as was earlier done on the regular file.


```
  /* If O_CREATE is set, try to make the file. */
  if (oflags & O_CREAT) {
        // omode = I_REGULAR | (omode & ALLPERMS & fp->fp_umask);
    // $$ //
    omode = I_IMMEDIATE | (omode & ALLPERMS & fp->fp_umask); // change
```

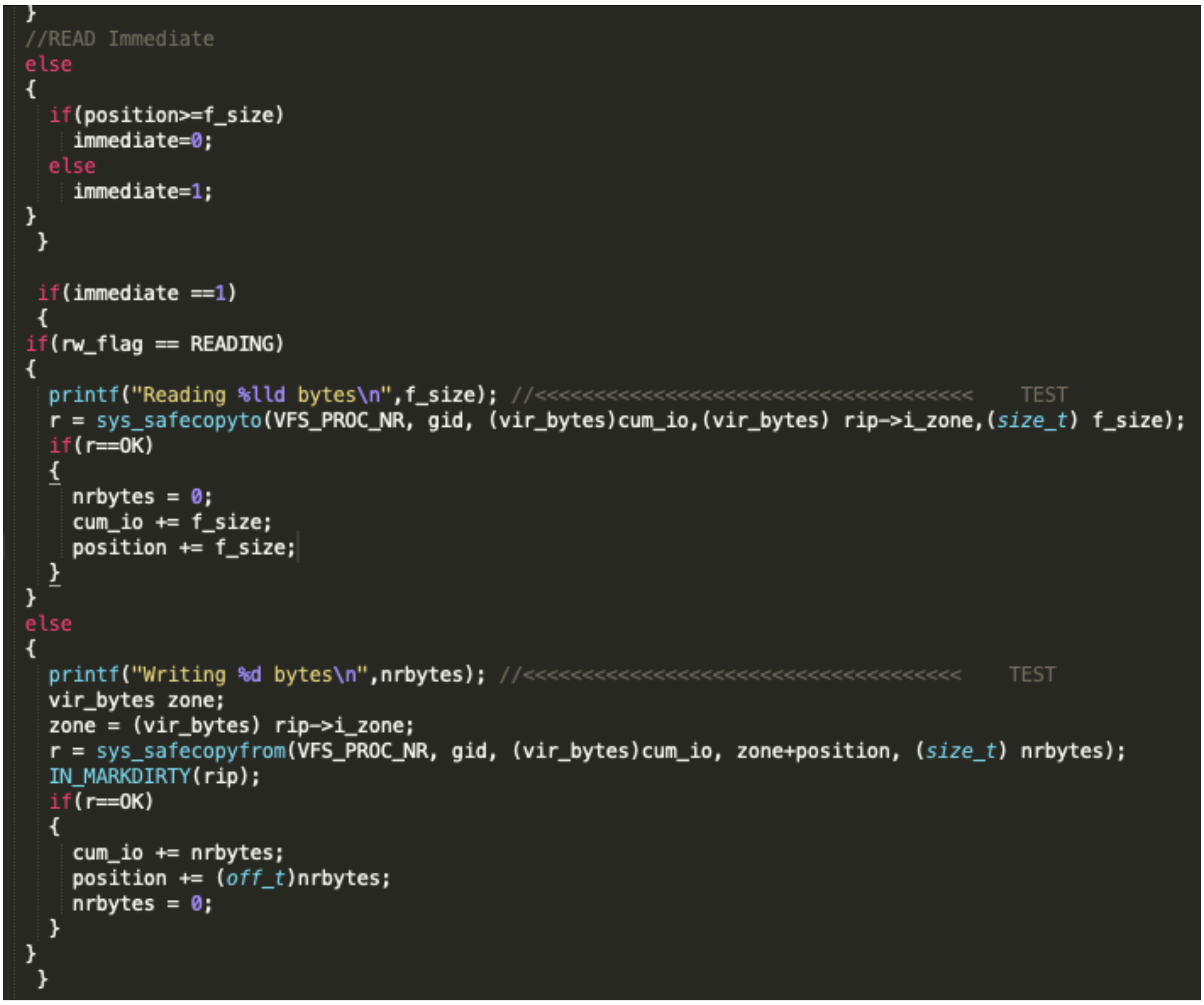
/minix/fs/mfs/read.c
In the fs_readwrite(void) function, we handle the reading and writing of the immediate files. If the file is in immediate mode and is to be written, if the file size is > 32 bytes, mode is changed to regular and the contents of the inode are copied to a new block which is marked dirty.
This achieves the immediate to regular transition.



read.c

```
  ////////////////////////////////////////////////////////////////////////
  if(mode_word==I_IMMEDIATE)
  {
 if(rw_flag==WRITING)
 {
   if(position+nrbytes>32)
   {
     int i;
     int post=0;
     char* tmp;
     char buffer[40];
     register struct buf* bp;

     for(i=0; i<f_size; i++)
     {
       if(i%4 == 0)
         tmp = (char*)rip->i_zone + i;
       buffer[i] = tmp[i%4];
       //printf("%c",tmp[i%4]);
     }
     wipe_inode(rip);
     rip->i_mode = (I_REGULAR | (rip->i_mode & ALL_MODES));
     mode_word = rip->i_mode & I_TYPE;
     if ((bp = new_block(rip, (off_t) ex64lo(post))) == NULL)
       return(err_code);
     for(i=0; i<f_size; i++)
     {
       ((char*)bp->data)[i] = buffer[i];
     }
     MARKDIRTY(bp);
     put_block(bp, PARTIAL_DATA_BLOCK);
   }
   else
     immediate=1;
 }
```

If the file is in immediate mode and wants to read or write, its size being <= 32 bytes, it fetches contents from and writes to inode itself.

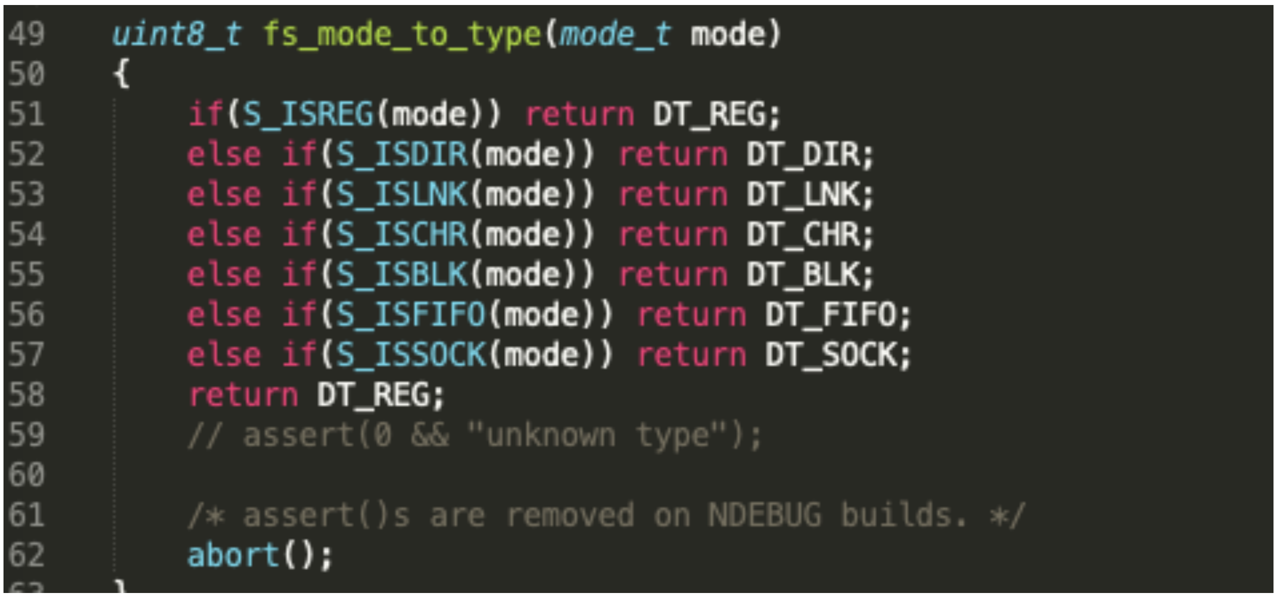


```
}
//READ Immediate
else
{
  if(position>=f_size)
    immediate=0;
  else
    immediate=1;
}
 }

 if(immediate ==1)
 {
if(rw_flag == READING)
{
  printf("Reading %lld bytes\n",f_size); //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<    TEST
  r = sys_safecopyto(VFS_PROC_NR, gid, (vir_bytes)cum_io,(vir_bytes) rip->i_zone,(size_t) f_size);
  if(r==OK)
  {
    nrbytes = 0;
    cum_io += f_size;
    position += f_size;
  }
}
else
{
  printf("Writing %d bytes\n",nrbytes); //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<    TEST
  vir_bytes zone;
  zone = (vir_bytes) rip->i_zone;
  r = sys_safecopyfrom(VFS_PROC_NR, gid, (vir_bytes)cum_io, zone+position, (size_t) nrbytes);
  IN_MARKDIRTY(rip);
  if(r==OK)
  {
    cum_io += nrbytes;
    position += (off_t)nrbytes;
    nrbytes = 0;
  }
}
 }
```

When the file is in regular mode, read and write operations are done as usual by looking up the zones. Also, no special changes are done to handle deletion of immediate files.

/minix/lib/libc/gen/fslib.c

In this file, we return the DT_REG flag in case the mode is not any of those listed in the function of fs_mode_to_type(mode_t mode), i.e., immediate mode is not recognised by this function and cribs by aborting the process. So, to overcome this, we return the DT_REG flag when we see that the mode is possibly immediate, so that the filesystem functions smoothly. An observed side-effect of this workaround is that vi editor does not recognise the files and throws up an error, hence vim should be used for viewing/editing files.

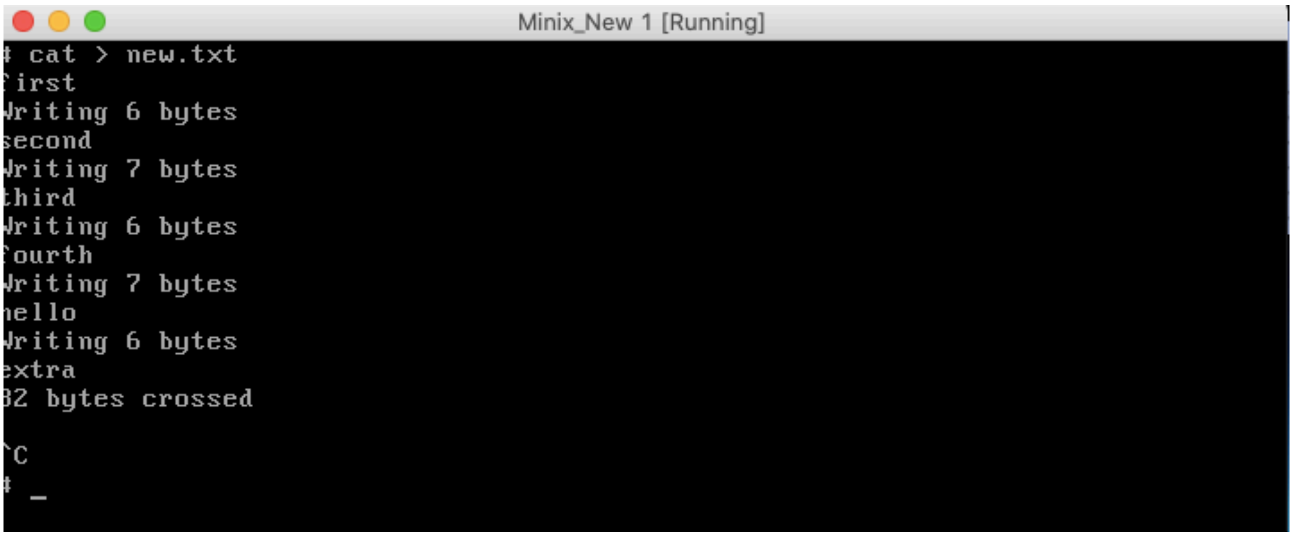


```
uint8_t fs_mode_to_type(mode_t mode)
{
    if(S_ISREG(mode)) return DT_REG;
    else if(S_ISDIR(mode)) return DT_DIR;
    else if(S_ISLNK(mode)) return DT_LNK;
    else if(S_ISCHR(mode)) return DT_CHR;
    else if(S_ISBLK(mode)) return DT_BLK;
    else if(S_ISFIFO(mode)) return DT_FIFO;
    else if(S_ISSOCK(mode)) return DT_SOCK;
    return DT_REG;
    // assert(0 && "unknown type");

    /* assert()s are removed on NDEBUG builds. */
    abort();
```

Working of the immediate files:

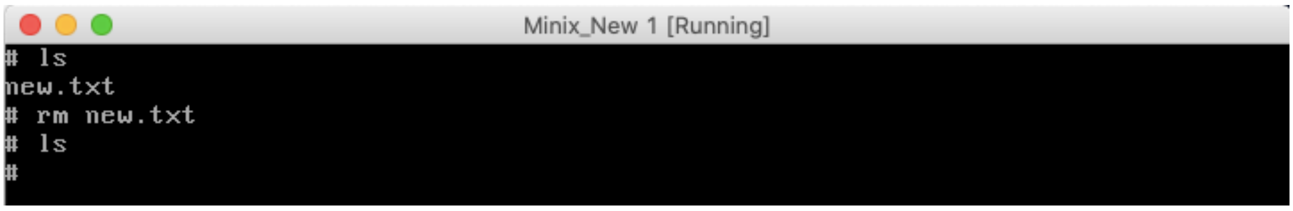
Here we show an overview of the working of immediate files that we had implemented.

One could observe that in this image, we create a file new.txt using cat command and write some content into it, until we reach the 32 byte mark, which is visible from the write <n> bytes lines which are printed from the fs_readwrite(void) function of /minix/fs/mfs/read.c file. We observe that once the file size exceeds 32 bytes, these statements are not printed as the file is now converted to a regular file.



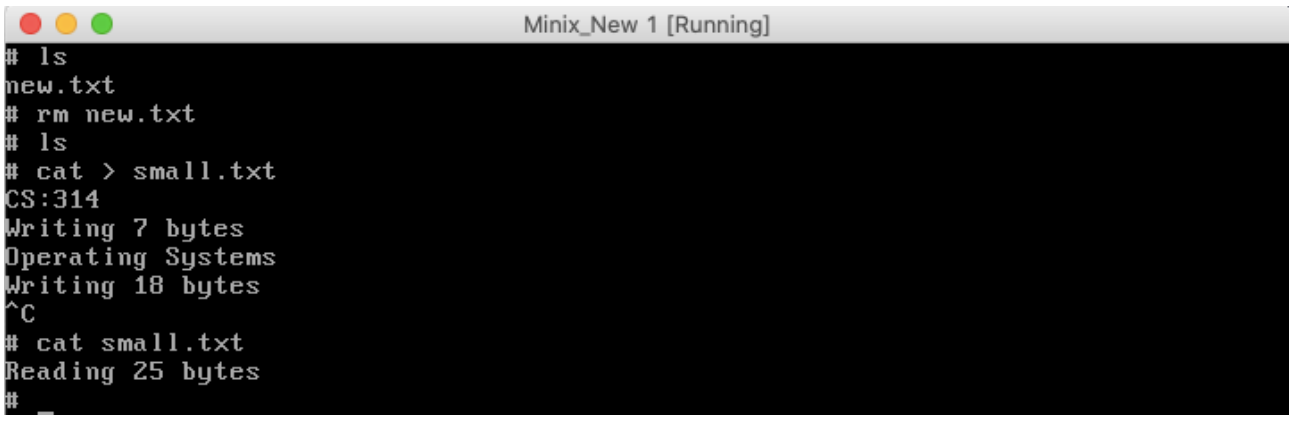
```
# cat > new.txt
first
Writing 6 bytes
second
Writing 7 bytes
third
Writing 6 bytes
fourth
Writing 7 bytes
hello
Writing 6 bytes
extra
32 bytes crossed

^C
# _
```

This screenshot shows that rm command works well when used to delete the file which is converted to regular from immediate type when file size exceeds 32 bytes.


Minix_New 1 [Running]

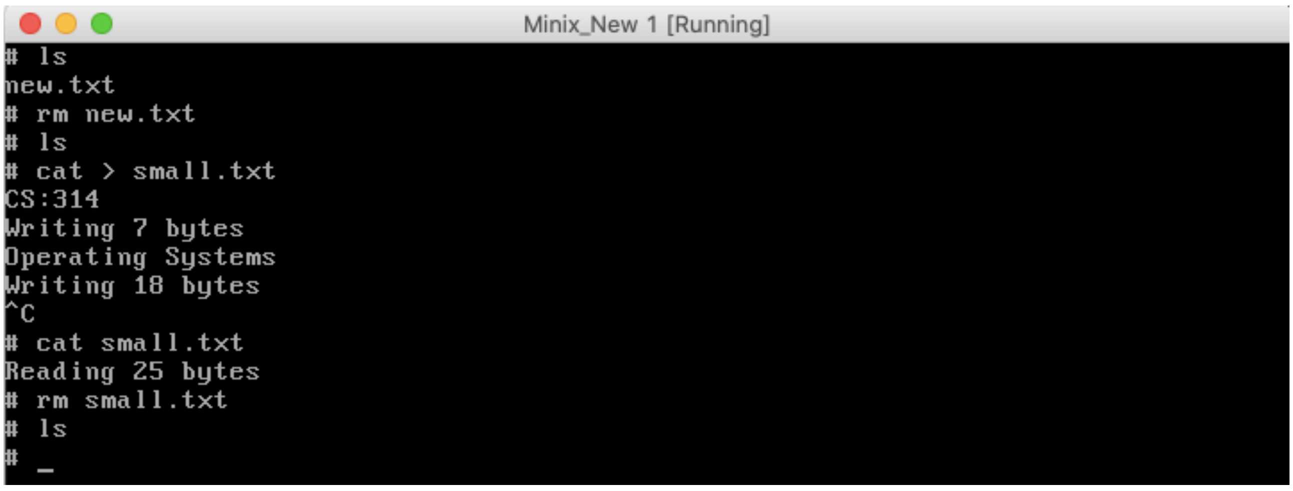
```
# ls
new.txt
# rm new.txt
# ls
#
```

This screenshot shows that cat command works well with the immediate file. Note that the size of immediate file is shown in the Reading <n> bytes line which is printed from the fs_readwrite(void) function of /minix/fs/mfs/read.c file.


Minix_New 1 [Running]

```
# ls
new.txt
# rm new.txt
# ls
# cat > small.txt
CS:314
Writing 7 bytes
Operating Systems
Writing 18 bytes
^C
# cat small.txt
Reading 25 bytes
# _
```

The next screenshot shows that rm command works well when used to delete this immediate file.


Minix_New 1 [Running]

```
# ls
new.txt
# rm new.txt
# ls
# cat > small.txt
CS:314
Writing 7 bytes
Operating Systems
Writing 18 bytes
^C
# cat small.txt
Reading 25 bytes
# rm small.txt
# ls
# _
```

Some important practical lessons learnt:

It is really important to take a snapshot/copy/.ova file of the working Minix OS in the VM, before making changes to the filesystem code, as the changes that are made to the filesystem may potentially wreak havoc on the OS which may render it unusable.

END OF REPORT