本科生实验报告

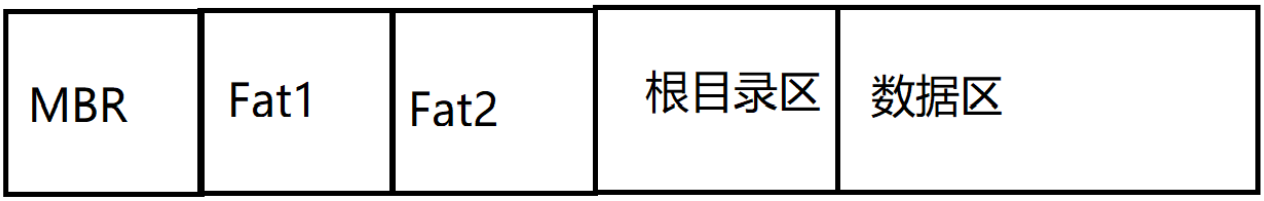
实验课程： 操作系统 实验名称： 期末大作业：文件系统
专业名称： 网络空间安全
学生学号： 20337021
实验地点： 我的家
实验成绩：
报告时间：

实验要求：

自己实现一个操作系统，并支持FAT 12 文件系统， 实现函数open, read, write, close, 并支持命令 cd, pwd, ls, cat, echo。

FAT 12 文件系统架构

FAT 12 文件系统时早期被Microsoft 发明的。在我们实验中使用一个虚拟的软盘（floppy disk) 来作为secondary storage。FAT12 的架构如下：



这个架构就是备用储存里面的储存架构，当中最低地址放着操作系统的应道主引导程序，之后时两个File allocation table（FAT）, 有两个的原因是因为当一个FAT损坏时，另外一个还可以当中备用。它的功能就是给出文件在数据区的位置。之后是根目录去和数据区。根目录存放着每一个文件或者文件夹的目录项，目录项存放着文件的元信息，包括文件名， 文件大小，文件位置。数据区就是放着所有文件里面的数据，通过目录项FAT表，就可以找到所需要文件的数据，并进行读写。

在MBR 里面放着文件系统组成信息，例如每簇扇区数 (bytes per sector) = 512 和 每簇扇区数（sectors per cluster) = 1。

```
Boot jump instruction
Oem identifier                  MSWIN4.1
Bytes per sector                512
Sectors per cluster             1
Reserved sectors                1
Fat count                       2
Dir entry count                 224
Total sectors                   2880
Media descriptor type           0F0h ;F0 = 3.5" floppydisk
Sectors per fat                 9    ; 9 sectors / fat
```

```
Sectors per track                   18
Heads                               2
Hidden sectors                      0
Large sector count                  0
```

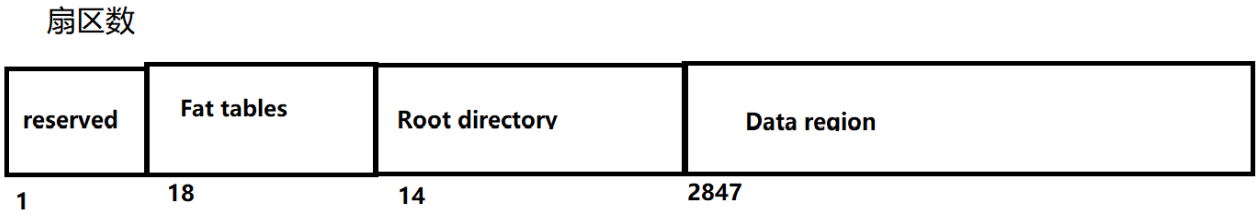
这里给出了MBR区占用了一个 1个扇区，么个FAT表占用9个扇区，因此FAT 区总共占用了2*9 = 18个三区，之后根目录所占用的扇区数

```
(Dir entry count * 32 + Bytes per sector - 1) / Bytes per sector = 14 个扇区
```

因此数据区占用了的扇区 = 总扇区 - 1 - 14 - 18 = 2880 - 33 = 2847，这也是簇的数量。

注意到FAT12当中的12是我们最小所需要给出每个扇区一个地址的比特数,因为我们有 2880个扇区，且

2^11 < 2880 < 2^12



如果一个文件大过一个簇的大小（512 bytes）， 那么就需要通过FAT 表找出它在512 bytes 以上的数据缩放在的簇的位置。注意FAT的文件系统都是链式分配的。我们现在讨论如何从一个在根目录里的目录项以及FAT表来获取文件数据。首先我们已知的一个文件名 name, 它有 8.3 格式（名，扩充）格式，需要转换成FAT长度位11的格式， 例如 "test.txt" => fat_name = "TEST TXT"(如果是文件架的话，FAT格式最后三个格会为空）。通过逐一读取在根目录的目录项，直到找到一个目录项direntry, 符合direntry.name == fat_name。找出对应的目录项后，通过开始簇direntry.firstcluster 计算出对应FAT。注意到fat12 是使用12位来作为数据的地址。

```
uint32_t FAT_NextCluster(uint32_t currentCluster)
{

   uint32_t fatIndex = currentCluster * 3 / 2;
   if(currentCluster % 2 == 0)
   {
    return (*(uint16_t far*)(g_Fat + fatIndex))& 0x0FFF;
   }
   else
   {
    return (*(uint16_t far*)(g_Fat + fatIndex)) >> 4;
   }
}
```

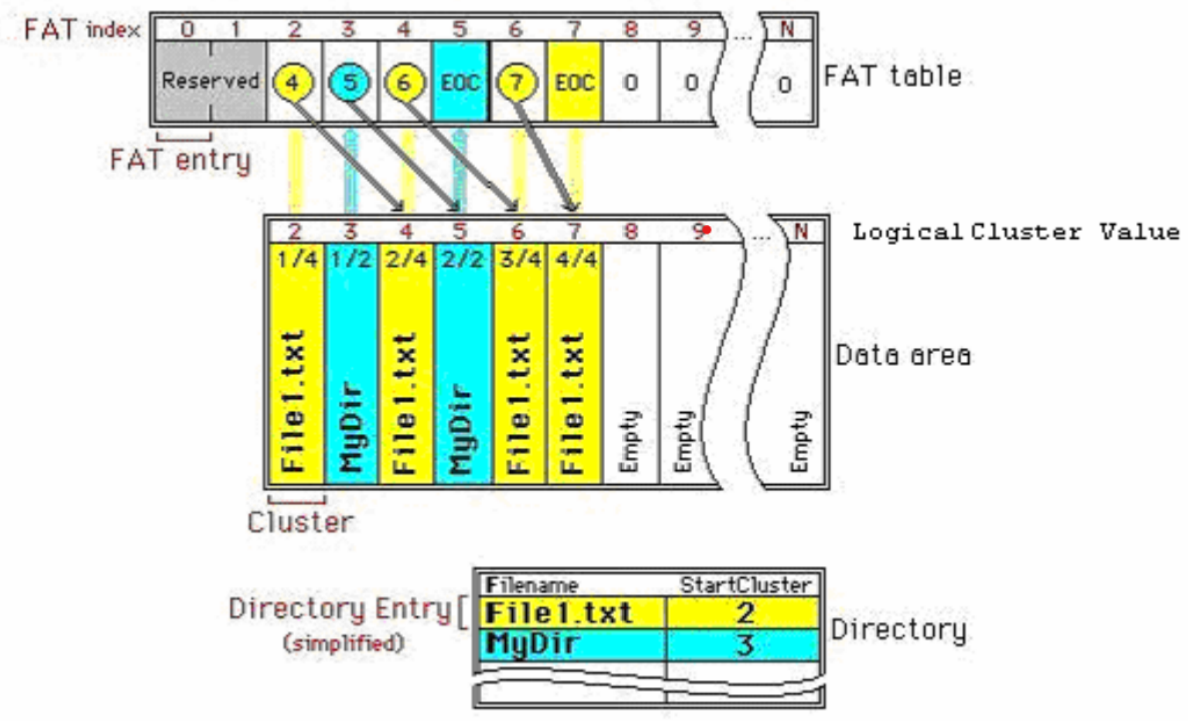
把FAT表看为一个数组FAT, 如果目前的簇位偶数，FAT表项 = FAT[currentCluster * 3/2] 的高四位 ；如是奇数，FAT表项 = FAT[currentCluster * 3/2] 的低四位。之前说过FAT12 是链式分配的，这是因为通过FAT表找出对应目前文件簇的项，该项的值就是对应目前文件簇的下一个簇的值。如果文件在目前的簇已经结束了，那么类似于链表使用NULL代表链表终止，如果对应的项为 0xFF8 - 0xFFF的值，就代表该簇为文件的最后一个簇。通过链式分配，解决到连续分配的问题。采用链接分配，每个文件是磁盘块（在这里叫做簇）的链表，磁盘块可能会散步在磁盘的任何地方。这解决了连续分配存在外来碎片的问题（external fragmentation), 空闲空间列表的任何块可以给用于满足请求。当创建文件时，并不需要说明文件的大小。只要有可用的空闲块，文件就可以继续郑家。因此，无需合并磁盘空间。

找到对应文件的簇以后，就需要计算每一个簇的地址。注意我们说过文件的数据是在数据区里面，因此簇的地址都是数据区的地址，也就是簇所在扇区，在这里叫做*物理扇区*，公式如下：

```
first_sector_of_cluster = (cluster - 2)*fat_boot->sectors_per_cluster =  +
first_data_sector;
```

注意到因为fat_boot->sectors_per_cluster = 1 (对于FAT12而言)，所以这个公式就是计算簇对应的扇区。first_data_sector 就是对应于数据区的第一个扇区。 而first_data_sector = 1 + 18+ 14 = 33 因此 first_sector_of_cluster = 33 + cluster - 2

我门以下例子来讲解以下FAT 12 的原理：



从图中，File1.txt的逻辑扇区为2， 4， 6 和 7。该文件的目录项的开始扇区FirstSector 为2， 也就是第一个数据扇区。在FAT表中，FAT表对应第2项的值为4，代表File1.txt的下一个数据扇区在逻辑扇区4。最后一个扇区为4, 这是显然的因为图上FAT表第7项写着EOC。对于文件夹MyDir 也是类似的。 注意到在文件的目录项里的 Attributes 就是用来分别出它是文件还是根文件里。

代码分析

我使用的是open watcom 的C编译器, nasm x86汇编器, 以及qemu模拟处理器，实现操作系统，并以文件系统为主要开发点实现文件系统 open, read, write, close 主要函数，以及相关cd, ls, cat, echo的命令。

在stage1文件夹里面定义了我们的引导程序boot.asm，它负责加载内核stage2.bin来执行。当中stage2.asm在根文件架里。在boot.asm里的load_kernel_loop 模块负责加载内核。

```
.load_kernel_loop:

    ; Read next cluster
    mov ax, [stage2_cluster]

    ; not nice :( hardcoded value
    add ax, 31                          ; first cluster = (stage2_cluster - 2) * sectors_per_cluster + start_sector
                                        ; start sector = reserved + fats + root directory size = 1 + 18 + 134 = 33
    mov cl, 1
    mov dl, [ebr_drive_number]
    call disk_read

    add bx, [bdb_bytes_per_sector]

    ; compute location of next cluster
    mov ax, [stage2_cluster]
    mov cx, 3
    mul cx
    mov cx, 2
    div cx                              ; ax = index of entry in FAT, dx = cluster mod 2

    mov si, buffer
    add si, ax
    mov ax, [ds:si]                     ; read entry from FAT table at index ax

    or dx, dx
    jz .even

.odd:
    shr ax, 4
    jmp .next_cluster_after

.even:
    and ax, 0x0FFF

.next_cluster_after:
    cmp ax, 0x0FF8                      ; end of chain
    jae .read_finish

    mov [stage2_cluster], ax
    jmp .load_kernel_loop
```

在stage2文件夹里定义了stage2.bin的代码，其中要执行的是main.c的main函数里面的代码。 其中所依赖的程序代码在：

x86.asm： 通过x86语言定义的读取/写入扇区，重置disk的函数

disk.c：封装x86.asm的代码为c代码

fat.c: 实现文件系统的函数open, read, write, close

shellCmd.c: 实现命令函指令cat, echo, cd, ls

1. x86.asm

x86_Disk_Reset: 通过中断指令 int 13h, 中断号 AH = 00h 重置disk

```
global _x86_Disk_Reset
_x86_Disk_Reset:
    ;;  make new call frame
    push bp         ; save old call frame
    mov bp, sp      ; initialize new call frame

    mov ah, 0
    mov dl, [bp+4]      ; dl - drive
    stc
    int 13h

    mov ax, 1
    sbb ax, 0       ; 1 on success, 0 on failure

    ;;  restore old call frame
    mov sp, bp
    pop bp
    ret
```

x86_Disk_Read: 通过中断指令 int 13h, 中断号 AH = 02h 读取扇区

```
 global _x86_Disk_Read
_x86_Disk_Read:
    ;;  make new call frame
    push bp         ; save old call frame
    mov bp, sp      ; initialize new call frame

    ;;  save modified regs
    push bx
    push es
    ;; setup args
    mov dl, [bp+4]      ; dl - drive

    mov ch, [bp+6]      ; ch - cylinder (lower 8 bits)
    mov cl, [bp+7]      ; cl - cylinder to bits 6-7
    shl cl, 6

    mov dh, [bp+10]     ; dh - head

    mov al, [bp+8]
    and al, 3Fh
    or cl, al       ; cl sectors to bits 0 -5

    mov al, [bp+12]     ; al - count

    mov bx, [bp+16]     ; es:bx - far pointer to data out
    mov es, bx
    mov bx, [bp+14]

    ;;  call int13h
    ;;  reset disk
    mov ah, 02h
    stc
    int 13h

    mov ax, 1
    sbb ax, 0       ; 1 on success, 0 on failure

    ;; restore regs
    pop es
    pop bx
    ;;  restore old call frame
    mov sp, bp
    pop bp
    ret
```

x86_Disk_Write： 通过中断指令 int 13h, 中断号 AH = 03h 写入扇区

```
global _x86_Disk_Write
_x86_Disk_Write:
        ;;  make new call frame
        push bp                 ; save old call frame
        mov bp, sp              ; initialize new call frame

        ;;  save modified regs
        push bx
        push es
        ;; setup args
        mov dl, [bp+4]          ; dl - drive

        mov ch, [bp+6]          ; ch - cylinder (lower 8 bits)
        mov cl, [bp+7]          ; cl - cylinder to bits 6-7
        shl cl, 6

        mov dh, [bp+10]         ; dh - head

        mov al, [bp+8]
        and al, 3Fh
        or cl, al               ; cl sectors to bits 0 -5

        mov al, [bp+12]         ; al - count

        mov bx, [bp+16]         ; es:bx - far pointer to data out
        mov es, bx
        mov bx, [bp+14]

        ;;  call int13h
        ;;  reset disk
        mov ah, 03h
        stc
        int 13h

        mov ax, 1
        sbb ax, 0               ; 1 on success, 0 on failure

        ;; restore regs
        pop es
        pop bx
        ;;  restore old call frame
        mov sp, bp
        pop bp
        ret
```

x86_Disk_GetDriveParams: 通过中断指令 int 13h, 中断号 AH = 08h, 传递给参数 柱面，磁头，扇区的总数量。

```
global _x86_Disk_GetDriveParams
_x86_Disk_GetDriveParams:
    ;;  make new call frame
    push bp             ; save old call frame
    mov bp, sp          ; initialize new call frame

    ;;  save regs
    push es
    push bx
    push si
    push di

    ;;  call int13h
    mov dl, [bp+4]          ; dl - disk drive
    mov ah, 08h
    mov di, 0           ; es:di - 0000:0000
    mov es, di
    stc
    int 13h

    ;; return
    mov ax, 1
    sbb ax, 0

    ;; out params
    mov si, [bp+6]          ; drive type from bl
    mov [si], bl

    mov bl, ch          ;cylinders - lower bits in ch
    mov bh, cl          ;cylinders - upper bits in cl (6-7)
    shr bh, 6
    mov si, [bp+8]          ; cylindersOut
    mov [si], bx


    xor ch, ch          ;sectors - lower 5 bits in cl
    and cl, 3Fh
    mov si, [bp+10]
    mov [si], cx

    mov cl, dh          ; heads -dh
    mov si, [bp+12]
    mov [si], cx

    ;; restore regs
    pop di
    pop si
    pop es
    pop es

    ;;  restore old call frame
    mov sp, bp
    pop bp
    ret
```

2. disk.c

我们有disk类，用来储存软盘 柱面，磁头，扇区的总数量。 DISK_Initialize （初始化disk类）：

```
bool DISK_Initialize(DISK *disk, uint8_t driveNumber)
{
    uint8_t driveType;
    uint16_t cylinders, sectors, heads;
    if(!x86_Disk_GetDriveParams(disk->id, &driveType, &cylinders, &sectors, &heads))
    {
        return false;
    }
    disk->id = driveNumber;
    disk->cylinders = cylinders+1;
    disk->heads = heads+1;
    disk->sectors = sectors;

    return true;
}
```

Disk_WriteSectors:

```
bool DISK_WriteSectors(DISK *disk, uint32_t lba, uint8_t sectors, void far *dataIn)
{
    uint16_t cylinder, sector, head;
    // 从 lba 转换成 chs,  以调用 x86_Disk_Write
    DISK_LBA2CHS(disk, lba, &cylinder, &head, &sector);

    //根据 Ralf Brown的interrupt list 网站所说，我们需要写入三次，
    // 每一次之间重置disk，才能够保证正确写入扇区
    for(int i = 0; i < 3; i++)
    {
        if(x86_Disk_Write(disk->id, cylinder, sector, head, sectors, dataIn))
        {
            return true;
        }
        x86_Disk_Reset(disk->id);
    }
    return false;
}
```

DISK_ReadSectors：读取扇区

```
bool DISK_ReadSectors(DISK *disk, uint32_t lba, uint8_t sectors, void far *dataOut)
{
    uint16_t cylinder, sector, head;
    // 从 lba 转换成 chs,  以调用 x86_Disk_Read
    DISK_LBA2CHS(disk, lba, &cylinder, &head, &sector);

    //根据 Ralf Brown的interrupt list 网站所说，我们需要读取三次，
    // 每一次之间重置disk，才能够保证正确读取扇区
    for(int i = 0; i < 3; i++)
    {
        if(x86_Disk_Read(disk->id, cylinder, sector, head, sectors, dataOut))
        {
            return true;
        }
        x86_Disk_Reset(disk->id);
    }
    return false;
}
```

DISK_LBA2CHS（从lba地址转换到chs)

```
void DISK_LBA2CHS(DISK *disk, uint32_t lba, uint16_t *cylinderOut, uint16_t *headOut, uint16_t *sectorOut)
{
    // sector = (LBA $ sectors per track + 1)
    *sectorOut = lba % disk->sectors + 1;
    // cylinders = (LBA / sectors per track) /heads
    *cylinderOut = (lba/disk->sectors) / disk->heads;

    // head = (LBA / sectors per track) % heads
    *headOut = (lba / disk->sectors) % disk->heads;
}
```

3. fat.c

目录项结构 (directory entry)

```
typedef struct {
    uint8_t Name[11]; // "8:3" filename (8: filename, 3 extension)
    uint8_t Attributes; //
    uint8_t _Reserved;
    uint8_t CreatedTimeTenths;
    uint16_t CreatedTime;
    uint16_t CreatedDate;
    uint16_t AccessedDate;
    uint16_t FirstClusterHigh; // high two bytes of first cluster number
    uint16_t ModifiedTime;
    uint16_t ModifiedDate;
    uint16_t FirstClusterLow; // start of a file in clusters
    uint32_t Size; // file size in bytes,  directories should be 0
} FAT_DirectoryEntry;
```

FAT_File: (对文件的信息)

- Handler：文件处理
- isDirectory: 是否文件架
- Position：目前在文件里的byte位置
- Size: 文件的byte大小

attributes: 文件的性质，例如只有读权限 (FAT_ATTRIBUTE_READ_ONLY), 是否文件夹 (FAT_ATTRIBUTE_DIRECTORY)

```
typedef struct
{
    int Handle;
    bool IsDirectory;
    uint32_t Position;
    uint32_t Size;
} FAT_File;


enum FAT_Attributes
{
    FAT_ATTRIBUTE_READ_ONLY = 0x01,
    FAT_ATTRIBUTE_HIDDEN= 0x02,
    FAT_ATTRIBUTE_SYSTEM = 0x04,
    FAT_ATTRIBUTE_VOLUME_ID = 0x08,
    FAT_ATTRIBUTE_DIRECTORY = 0x10,
    FAT_ATTRIBUTE_ARCHIVE = 0x20,
    FAT_ATTRIBUTE_LFN = FAT_ATTRIBUTE_READ_ONLY | FAT_ATTRIBUTE_HIDDEN | FAT_ATTRIBUTE_SYSTEM | FAT_ATTRIBUTE_VOLUME_ID
};
```

引导扇区结构：

```
// structure of boot sector
typedef struct
{

    uint8_t BootJumpInstruction[3];
    uint8_t OemIdentifer[8];
    uint16_t BytesPerSector;
    uint8_t SectorsPerCluster;
    uint16_t ReservedSectors;
    uint8_t FatCount;
    uint16_t DirEntryCount;
    uint16_t TotalSectors;
    uint8_t MediaDescriptorType;
    uint16_t SectorsPerFat;
    uint16_t SectorsPerTrack;
    uint16_t Heads;
    uint32_t HiddenSectors;
    uint32_t LargeSectorCount;

    // extended boot record
    uint8_t DriveNumber;
    uint8_t _Reserved;
    uint8_t Signature;
    uint32_t VolumeId; // serial number , value doesn't matter
    uint8_t VolumeLabel[11]; // 11 bytes, padded with spaces
    uint8_t SystemId[8];

    // 我们不理会代码部分
} FAT_BootSector;
```

读取引导扇区和FAT表

```
// 读取引导扇区
bool FAT_ReadBootSector(DISK *disk)
{
    return DISK_ReadSectors(disk, 0, 1, &g_Data->BS.BootSectorBytes);
}
// 读取FAT表
bool FAT_ReadFat(DISK *disk)
{
    return DISK_ReadSectors(disk, g_Data->BS.BootSector.ReservedSectors, g_Data->BS.BootSector.SectorsPerFat, g_Fat);
}
```

文件系统初始化：

```
bool FAT_Initialize(DISK *disk)
{
    g_Data = (FAT_Data far*)MEMORY_FAT_ADDR;


    // 读取引导扇区
    if(!FAT_ReadBootSector(disk))
    {
        printf("FAT: read boot sector error");
        return false;
    }
    // 读取FAT表

    uint32_t fatSize = g_Data->BS.BootSector.BytesPerSector * g_Data->BS.BootSector.SectorsPerFat;
    g_Fat = (uint8_t far *)g_Data + sizeof(FAT_Data);


    if(sizeof(FAT_Data) + fatSize >= MEMORY_FAT_SIZE)
    {
        printf("FAT: not enough memory to read FAT! Required %lu, only have %lu\r\n", sizeof(FAT_Data) + fatSize, MEMORY_FAT_SIZE);
        return false;
    }
    if(!FAT_ReadFat(disk))
    {
        printf("FAT: read FAT failed\r\n");
        return false;
    }

    // 开根文件

    uint32_t rootDirLba = g_Data->BS.BootSector.ReservedSectors + g_Data->BS.BootSector.SectorsPerFat * g_Data->BS.BootSector.FatCount;

    uint32_t rootDirSize =  sizeof(FAT_DirectoryEntry) * g_Data->BS.BootSector.DirEntryCount;

    g_Data->RootDirectory.Public.Handle = ROOT_DIRECTORY_HANDLE;
    g_Data->RootDirectory.Public.IsDirectory = true;
    g_Data->RootDirectory.Public.Position = 0;
    g_Data->RootDirectory.Public.Size = sizeof(FAT_DirectoryEntry) * g_Data->BS.BootSector.DirEntryCount;  // directory_entry_size * num of directory entry
    g_Data->RootDirectory.Opened = true;
    g_Data->RootDirectory.FirstCluster = rootDirLba;
    g_Data->RootDirectory.CurrentCluster = rootDirLba;
    g_Data->RootDirectory.CurrentSectorInCluster = 0;

    if(!DISK_ReadSectors(disk, rootDirLba, 1, g_Data->RootDirectory.Buffer))
    {
        printf("FAT: read root directory failed\r\n");
        return false;
    }

    // 计算数据区开始第一个扇区的位置
    uint32_t rootDirSectors = (rootDirSize + g_Data->BS.BootSector.BytesPerSector - 1)/g_Data->BS.BootSector.BytesPerSector;
    g_DataSectionLba = rootDirLba + rootDirSectors;

    // 从新设定已开文件
    for(int i = 0; i < MAX_FILE_HANDLES; i++)
    {
        g_Data->OpenedFiles[i].Opened = false;
    }

    return true;
}
```

FAT_ClusterToLba:

```
uint32_t FAT_ClusterToLba( uint32_t cluster)
{
    return g_DataSectionLba + (cluster - 2)  * g_Data->BS.BootSector.SectorsPerCluster;
}
```

FAT_OpenEntry:

```
FAT_File far *FAT_OpenEntry(DISK * disk, FAT_DirectoryEntry * entry)
{
    // 找出空的文件处理
    int handle = -1;
    for(int i = 0; i = MAX_FILE_HANDLES && handle < 0; i++)
    {
        if(!g_Data->OpenedFiles[i].Opened)
            handle = i;
    }
    // 没有空的文件处理
    if(handle < 0)
    {
        printf("FAT: out of file handlers\r\n");
        return false;
    }
    // 设置
    FAT_FileData far * fd = &g_Data->OpenedFiles[handle];
    fd->Public.Handle = handle;
    fd->Public.IsDirectory = (entry->Attributes & FAT_ATTRIBUTE_DIRECTORY) != 0;
    fd->Public.Position = 0;
    fd->Public.Size = entry->Size;
    fd->FirstCluster = entry->FirstClusterLow + ((uint32_t)entry->FirstClusterHigh << 16);
    fd->CurrentCluster= fd->FirstCluster;
    fd->CurrentSectorInCluster = 0;

    if(!DISK_ReadSectors(disk, FAT_ClusterToLba(fd->CurrentCluster), 1, fd->Buffer))
    {
        printf("FAT: read error\r\n");
        return false;
    }

    fd->Opened = true;
    return &fd->Public;
}
```

FAT_NextCluster:

```
uint32_t FAT_NextCluster(uint32_t currentCluster)
{

    uint32_t fatIndex = currentCluster * 3 / 2;
    if(currentCluster % 2 == 0) // n 为偶数
    {
        return (*(uint16_t far*)(g_Fat + fatIndex))& 0x0FFF;
    }
    else // n 为奇数
    {
        return (*(uint16_t far*)(g_Fat + fatIndex)) >> 4;
    }
}
```

FAT_Read:

```
uint32_t FAT_Read(DISK *disk, FAT_File far *file, uint32_t byteCount, void *dataOut)
{
    FAT_FileData far *fd = (file->Handle == ROOT_DIRECTORY_HANDLE)
    ? &g_Data->RootDirectory
    : &g_Data->OpenedFiles[file->Handle];

    uint8_t *u8DataOut = (uint8_t *) dataOut;

    // 获取文件数据
    if(!fd->Public.IsDirectory || (fd->Public.IsDirectory && fd->Public.Size != 0))
    {
        byteCount = min(byteCount, fd->Public.Size - fd->Public.Position);
    }

    while(byteCount > 0)
    {
        uint32_t leftInBuffer = SECTOR_SIZE - (fd->Public.Position % SECTOR_SIZE);
        uint32_t take = min(byteCount, leftInBuffer);

        memcpy(u8DataOut, fd->Buffer + fd->Public.Position % SECTOR_SIZE, take);
        u8DataOut += take;
        fd->Public.Position += take;
        byteCount -= take;

        //  看我们需不需要读取更多数据
        if(leftInBuffer == take)
        {
            // 对根文件的特定处理
            if(fd->Public.Handle == ROOT_DIRECTORY_HANDLE)
            {
                ++fd->CurrentCluster;
                // 读取下一个扇区
                if(!DISK_ReadSectors(disk, fd->CurrentCluster, 1, fd->Buffer))
                {
                    printf("FAT: read error!\r\n");
                    break;
                }
                }
                else
                {
                // 计算下一个要读的簇和扇区
                if(++fd->CurrentSectorInCluster >= g_Data->BS.BootSector.SectorsPerCluster)
                {
                    fd->CurrentSectorInCluster = 0;
                    fd->CurrentCluster = FAT_NextCluster(fd->CurrentCluster);
                }
                if(fd->CurrentCluster >= 0xFF8) // 文件结束
                {
                    // 记录文件结束位置
                    fd->Public.Size = fd->Public.Position;
                    break;
                }

                // 读取下一个扇区
                if(!DISK_ReadSectors(disk, FAT_ClusterToLba(fd->CurrentCluster) + fd->CurrentSectorInCluster,1, fd->Buffer))
                {
                    // 处理错误
                    printf("FAT: read error!\r\n");
                    break;
                }
            }
        }
    }
    return u8DataOut - (uint8_t*)dataOut;
}
```

FAT_ReadEntry:

```
bool FAT_ReadEntry(DISK *disk, FAT_File far * file, FAT_DirectoryEntry *dirEntry)
{
    return FAT_Read(disk, file, sizeof(FAT_DirectoryEntry), dirEntry) == sizeof(FAT_DirectoryEntry);
}
```

FAT_Write:

```
bool FAT_Write(DISK *disk, FAT_File far *file, void *dataIn)
{
    FAT_FileData far *fd = (file->Handle == ROOT_DIRECTORY_HANDLE)
    ? &g_Data->RootDirectory
    : &g_Data->OpenedFiles[file->Handle];
    // 写数据进入对应文件的扇区
    return DISK_WriteSectors(disk, FAT_ClusterToLba(fd->CurrentCluster), 1, dataIn);
}
```

FAT_FindFile

```
// 在根文件夹找出文件
bool FAT_FindFile(DISK *disk, FAT_File far* file, const char *name, FAT_DirectoryEntry *entryOut)
{

    char fatName[12];
    //转换文件名成 FAT格式
    convertToFATFilename(fatName, name);
    FAT_DirectoryEntry entry;
    while(FAT_ReadEntry(disk, file, &entry))
    {
        if(memcmp(fatName, entry.Name, 11) == 0)
        {
            *entryOut = entry;
            return true;
        }
    }
    return false;
}
```

FAT_Open:

```
FAT_File far *FAT_Open(DISK * disk, const char * path)
{
    char name[MAX_PATH_SIZE];
    // 略到斜线
    if(path[0] == '/')
        path++;
    FAT_File far *current  = &g_Data->RootDirectory.Public;

    while(*path)
    {
        // 获取路径的下一个文件名
        bool isLast = false;
        const char * delim = strchr(path, '/');
        if(delim != NULL)
        {
            memcpy(name, path, delim-path);
            name[delim-path + 1] = '\0';
            path = delim + 1;
        }
        else
        {
            unsigned len = strlen(path);
            memcpy(name, path, len);
            name[len+1] = '\0';
            path += len;
            isLast = true;
        }

        // 在目前文件家里找出对应文件名的目录项
        FAT_DirectoryEntry entry;
        if(FAT_FindFile(disk, current, name, &entry))
        {
            FAT_Close(current);
            // 判断是否文件夹
            if(!isLast && (entry.Attributes & FAT_ATTRIBUTE_DIRECTORY) == 0)
            {
            printf("FAT: %s not a directory\r\n", name);
            return NULL;
            }

            // open new directory entry
            current = FAT_OpenEntry(disk, &entry);
        }
        else
        {
            FAT_Close(current);
            printf("FAT %s not found\r\n", name);
            return NULL;
        }
    }
    return current;
}
```

4. shellCmd.c

ls:

```
void command_ls(DISK *disk, const char *currentWorkingDirectory)
{
    FAT_File far *fd = FAT_Open(disk, currentWorkingDirectory);

    FAT_DirectoryEntry entry;
    int i = 0;
    while(FAT_ReadEntry(disk, fd, &entry) && i++ < 5)
    {
        char filename[11];
        char extension[3];

        if(entry.Name[0] == 'N' && entry.Name[1] == 'B' && entry.Name[2] == 'O' && entry.Name[3] == 'S')
        {
            strcpy(filename, entry.Name);
            printf("%s", filename);
        }
        else if((entry.Attributes & 0x10))
        {
            convertToFileDirectoryName(filename, entry.Name);
            printf("%s", filename);
        }
        else
        {
            convertToFilename(filename, extension, entry.Name);
            printf("%s.%s", filename, extension);
        }
        printf("    ");
    }

    FAT_Close(fd);
    printf("\r\n");
}
```

cd：

```
bool command_cd(DISK *disk, char *currentDirectoryPath, char *directoryName)
{
    FAT_File far *fd = FAT_Open(disk, currentDirectoryPath);
    FAT_DirectoryEntry entry;
    if(FAT_FindFile(disk, fd, directoryName, &entry))
    {
        // not a directory
        if((entry.Attributes & 0x10) == 0)
        {
            printf("Error: The directory name is invalid\n");
            FAT_Close(fd);
            printf("\r\n");
            return false;
        }
        // is a directory

        if(!(strcmp(currentDirectoryPath, "/")== 0)) // not root directory
        {
            strcat(currentDirectoryPath, "/");
        }
        strcat(currentDirectoryPath, directoryName);
        FAT_Close(fd);
        printf("\r\n");
        return true;
    }
    else
    {
        printf("The system cannot find the path specified\n");
        FAT_Close(fd);
        printf("\r\n");
        return false;
    }
}
```

cat:

```
void command_cat(DISK *disk, char *currentDirectoryPath, char *directoryName)
{

    char fullFilePath[256];
    strcpy(fullFilePath, currentDirectoryPath);

    if(strcmp(currentDirectoryPath, "/") != 0) // not root directory
    {
        strcat(fullFilePath, "/");
    }

    strcat(fullFilePath, directoryName);

    char buffer[100];
    uint32_t read;
    FAT_File far *fd = FAT_Open(disk, fullFilePath);
    while((read = FAT_Read(disk, fd, sizeof(buffer), buffer)))
    {
        for(uint32_t i = 0; i < read; i++)
        {
            if(buffer[i] == '\n')
            putc('\r');
            putc(buffer[i]);
        }
    }
    FAT_Close(fd);
    printf("\r\n");
}
```

echo:

```
void command_echo(DISK *disk, char *currentDirectoryPath, char *filename, char *buffer)
{
    /*
    FAT_File far * fd = FAT_Open(disk, currentDirectoryPath);
    FAT_updateDirEntry(disk, fd, strlen(buffer) + 1, filename);
    FAT_Close(fd);
    */

    char fullFilePath[256];
    strcpy(fullFilePath, currentDirectoryPath);
    if(!(strcmp(currentDirectoryPath, "/") == 0)) // not root directory
    {
        strcat(currentDirectoryPath, "/");
    }
    strcat(fullFilePath, filename);

    FAT_File far * fd = FAT_Open(disk, fullFilePath);
    FAT_Write(disk, fd, buffer);
    FAT_Close(fd);
    printf("\r\n");
}
```

转换文件名到fat格式的函数：

```
void convertToFATFilename(char *FAT_Filename, const char *name)
{
    //convert from name to fat name
    memset(FAT_Filename, ' ', 11);
    FAT_Filename[11] = '\0';
    const char * ext = strchr(name, '.');
    if(ext == NULL)
    {
    ext = name + 11;
    }
    for(int i = 0; i < 8 && name[i] && name + i < ext ; i++)
    {
    FAT_Filename[i] = toupper(name[i]);
    }

    if(ext != name + 11)
    {
    for(int i = 0; i < 3 && ext[i+1]; i++)
    {
        FAT_Filename[i+8] = toupper(ext[i+1]);
    }
    }
}
```

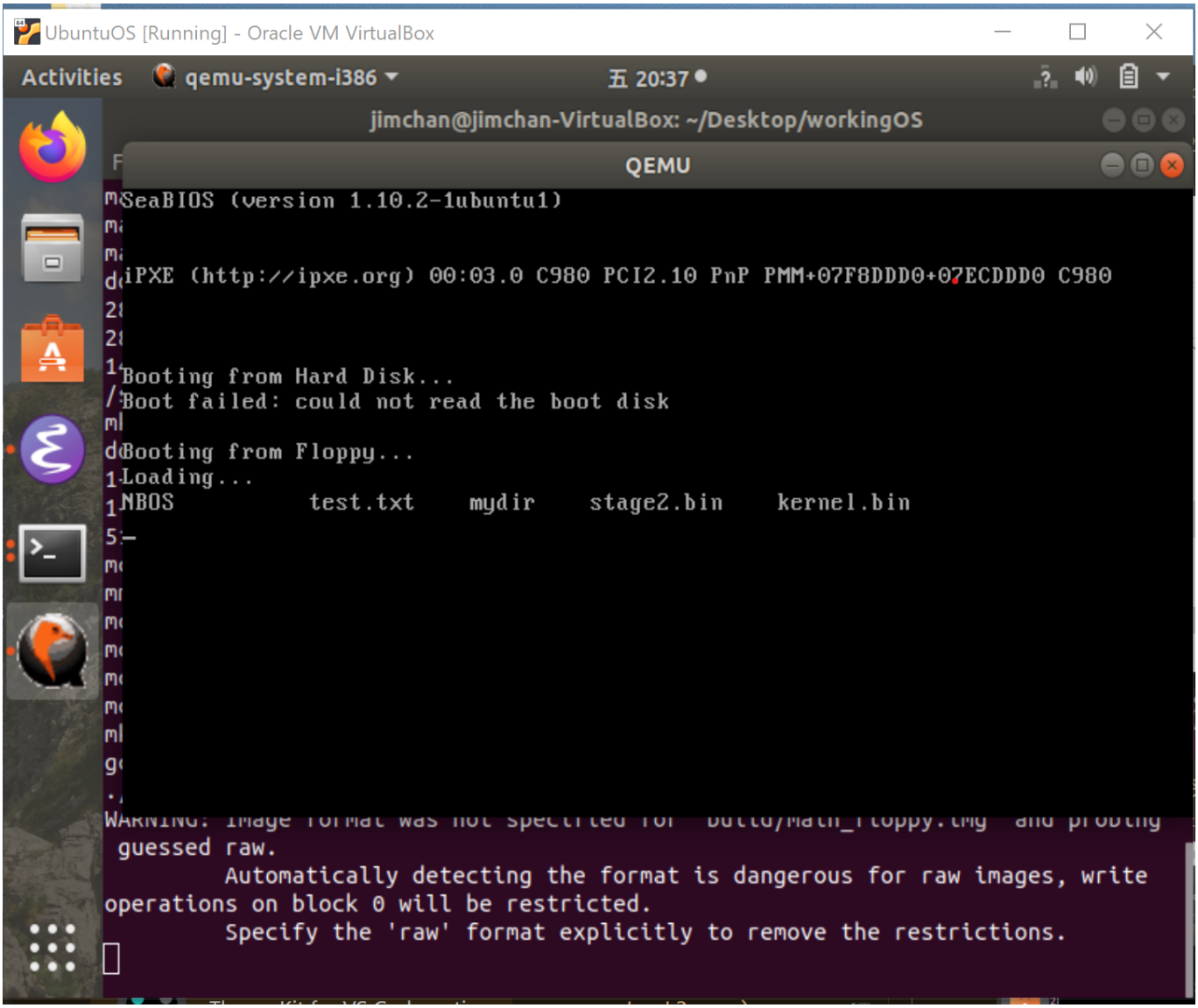
转换fat格式到文件名的函数

```
char *trimString(char *str)
{
    char *end;

    while(isspace((unsigned char) *str)) str++;

    if(*str == 0)
    {
    return str;
    }
    end = str + strlen(str) - 1;
    while(end > str && isspace((unsigned char) *end))
    end--;
    end[1] = '\0';

    return str;
}

void convertToFileDirectoryName(char *filename, const char *FAT_Filename)
{
    strncpy(filename, FAT_Filename, 8);
    filename[8] = '\0';
    trimString(filename);
    for(int i = 0; i < strlen(filename); i++)
    {
        filename[i] = tolower(filename[i]);
    }
}
```

实验演示

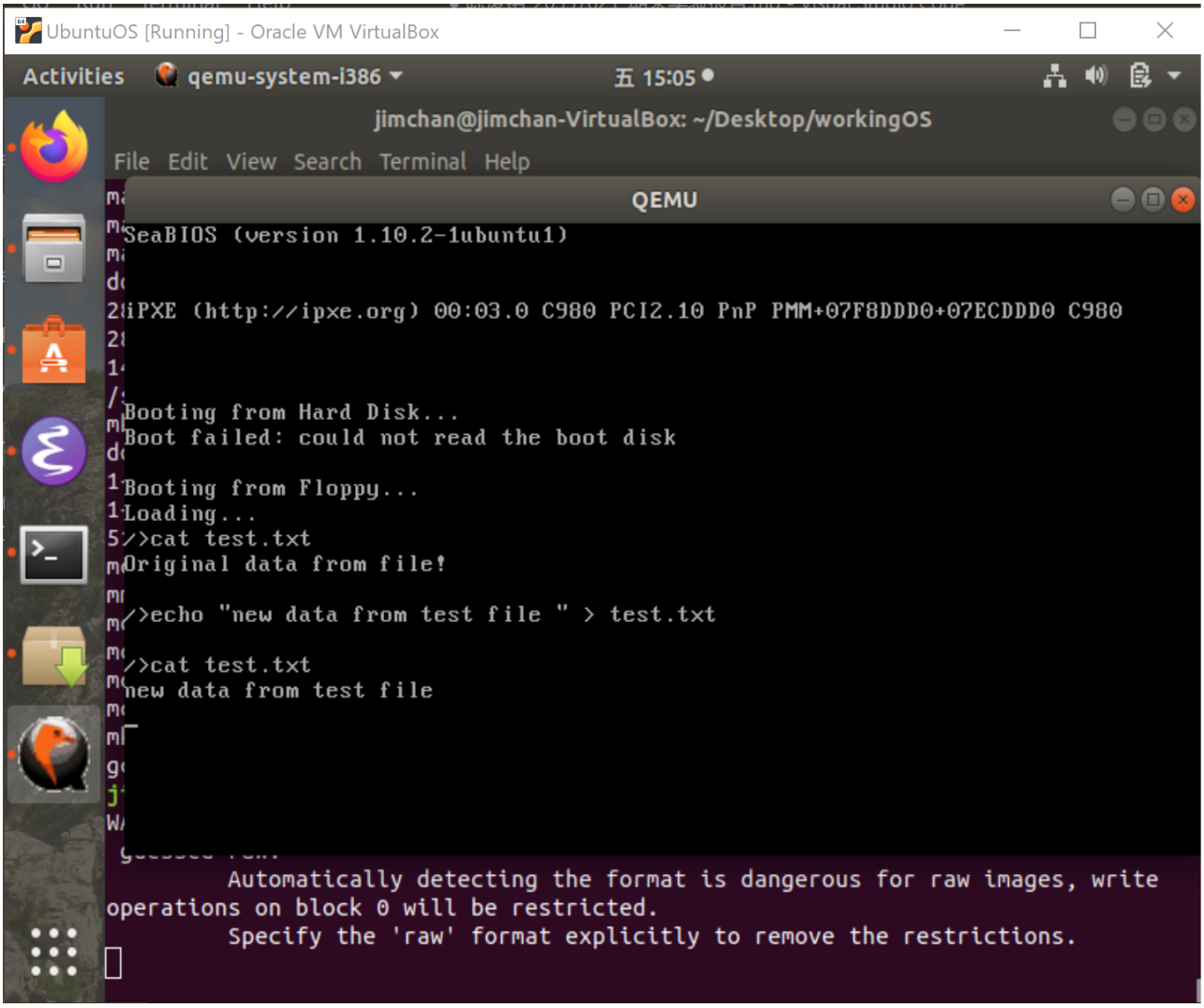
在stage2文件夹的main.c 测试。

根文件：


UbuntuOS [Running] - Oracle VM VirtualBox
Activities
qemu-system-i386
五 20:37
jimchan@jimchan-VirtualBox: ~/Desktop/workingOS
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDD0+07ECDDD0 C980
Booting from Hard Disk...
Boot failed: could not read the boot disk
Booting from Floppy...
Loading...
NBOS test.txt mydir stage2.bin kernel.bin
guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.

测试1

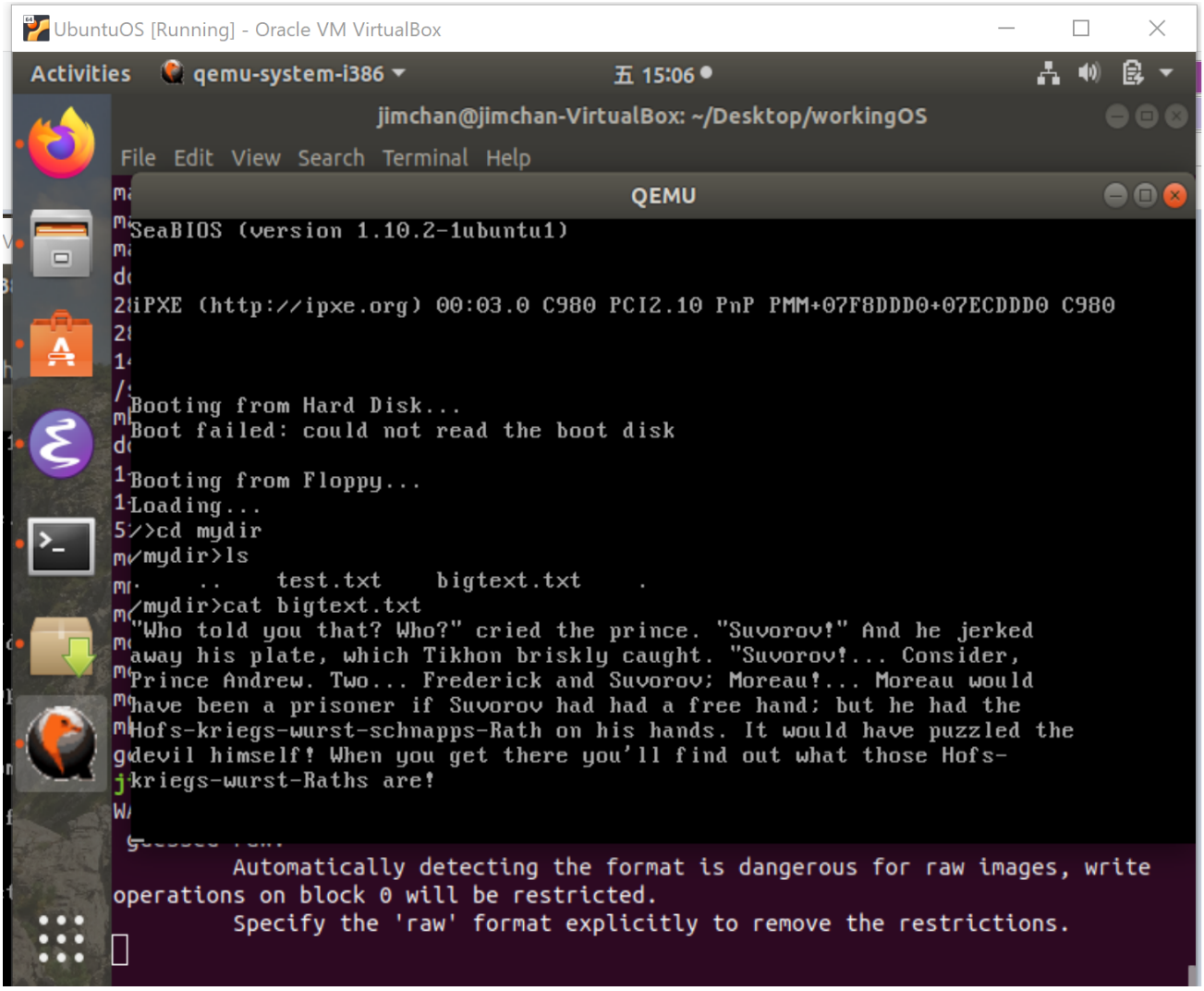
```
void test1(DISK *disk)
{
    const char *testStr = "new data from test file \0";
    printLocation();
    printf("cat test.txt\r\n");
    command_cat(disk,  currentWorkingDirectoryPath, "test.txt");
    printLocation();
    printf("echo \"%s\" > test.txt\r\n", testStr);
    command_echo(disk, currentWorkingDirectoryPath, "test.txt", testStr);
    printLocation();
    printf("cat test.txt\r\n");
    command_cat(disk,  currentWorkingDirectoryPath, "test.txt");
}
```



在这里演示的是cat 和 echo的命令，也就是读和写操作。从图中可看到文件内容从 “Original data from file!" 写成 "new data from text file"

测试2

```
void test2(DISK *disk)
{
    printLocation();
    printf("cd mydir\r\n");
    command_cd(disk, currentWorkingDirectoryPath, "mydir");
    printLocation();
    printf("ls\r\n");
    command_ls(disk, currentWorkingDirectoryPath);
    printLocation();
    printf("cat bigtext.txt\r\n");
    command_cat(disk, currentWorkingDirectoryPath, "bigtext.txt");
}
```



使用cd 进入文件夹mydir，ls列出所有文件夹里面的文件，并cat 输出里面的文件的内容

测试3

之后我们测试一个大文本（大小超过512 byte， 1 sector）的cat 调用：