编号：_____

| 实验 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 总评 | 教师签名 |
|---|---|---|---|---|---|---|---|---|---|---|
| 成绩 | | | | | | | | | | |

武汉大学国家网络安全学院

# 课程实验(设计)报告

题　　目：作业二：FAT32 文件系统数据安全删除

专业(班)：信息安全

学　　号：2021302181156

姓　　名：赵伯俣

课程名称：软件安全

任课教师：赵磊

2023 年 12 月 28 日

# 目 录

# 实验 6　FAT32 文件系统数据安全删除

# 1 实验名称

FAT32 文件系统数据安全删除

# 2 实验目的

（1）分析 FAT32 文件系统的数据结构
（2）对于给定的文件，要求能够通过分析 FAT 表识别出来文件所占的簇链
（3）对簇的内容重写，以达到安全删除的目的

# 3 实验原理

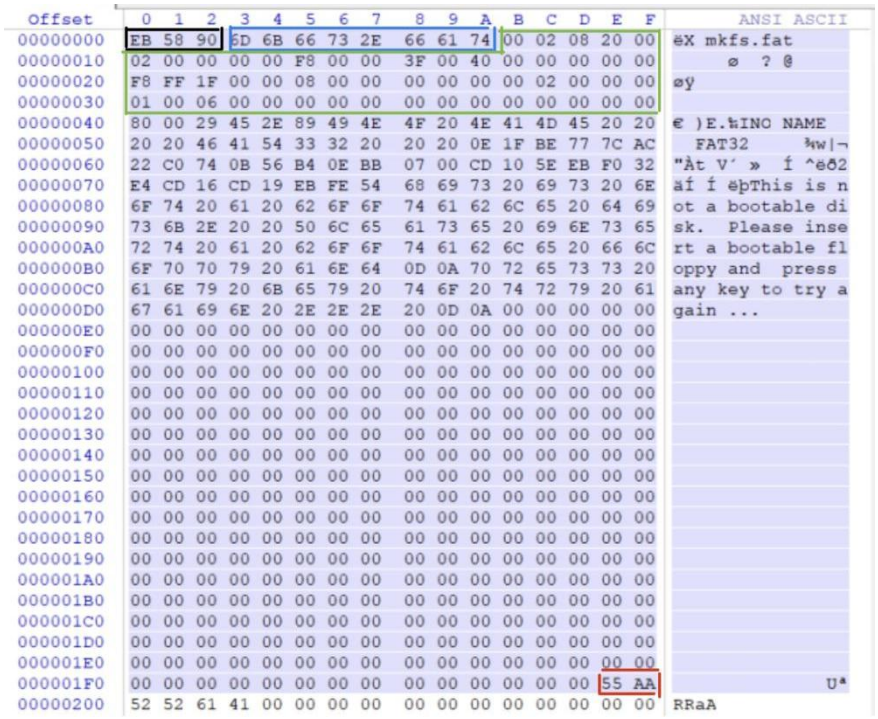
## 3.1 FAT32 文件系统格式

### 3.1.1 FAT32 文件系统整体格式。

（1）DBR 及其保留扇区：DBR 的含义是 DOS 引导记录，也称为操作系统引导记录，在 DBR 之后往往会有一些保留扇区。
（2）FAT1：FAT 的含义是文件分配表，FAT32 一般有两份 FAT，FAT1 是第一份，也是主 FAT。
（3）FAT2：FAT2 是 FAT32 的第二份文件分配表，也是 FAT1 的备份。
（4）DATA：DATA 也就是数据区，是 FAT32 文件系统的主要区域，其中包含目录区域。

### 3.1.2 DBR 格式分析

在本次实验中采用的软盘的 DBR 如下图所示




（1）跳转指令，为图中黑色矩形部分，本身占 2 字节它将程序执行流程跳转到引导程序处

（2）OEM 代号，为图中蓝色部分，这部分占 8 字节，其内容由创建该文件系统的 OEM 厂商具体安排

（3）结束标志。为图中红色矩形部分

（4）引导程序代码。为图中未框出的部分

（5）BPB，为图中绿色矩形部分，记录了文件的重要参数信息。其每个字段的含义如下表所示

| 字段长度 | 名称定义 | 值 |
|---|---|---|
| 2 | 扇区字节数 | 200h |
| 1 | 每簇扇区数 | 8h |
| 2 | 保留扇区数 | 20h |
| 1 | FAT 数 | 2h |
| 2 | 根目录项数（FAT12/16） | 0 |
| 2 | 小扇区数（FAT12/16） | 0 |
| 1 | 媒体描述符 | F8h |
| 2 | 每 FAT 扇区数（FAT12/16） | 0 |
| 2 | 每道扇区数 | 3Fh |
| 2 | 磁头数 | 40h |
| 4 | 隐藏扇区数 | 0 |
| 4 | 总扇区数 | FFFF8h |
| 4 | 每 FAT 扇区数（FAT32） | 800h |

| | | |
|---|---|---|
| 2 | 拓展标志 | 0 |
| 2 | 文件系统版本 | 0 |
| 4 | 根目录簇号 | 2h |
| 2 | 文件系统信息扇区号 | 1h |
| 2 | 备份引导扇区 | 6h |
| 12 | 保留 | 0 |

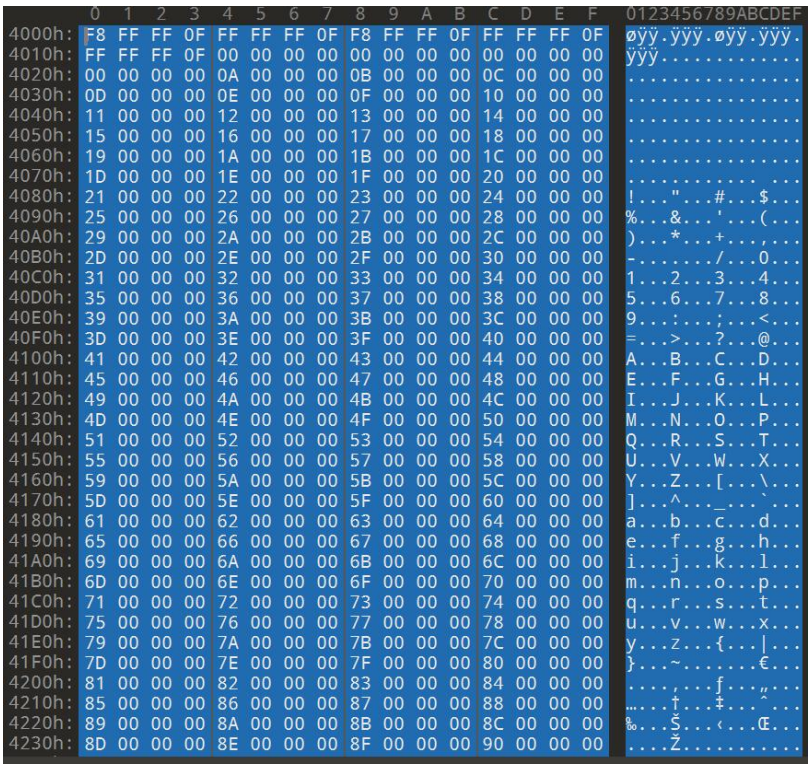
在 BPB 字段之后还存在有拓展 BPB 字段，其字段结构即定义如下表所示

| 字段长度 | 名称定义 | 值 |
|---|---|---|
| 1 | 物理驱动器号 | 80h |
| 1 | 保留 | 0 |
| 1 | 拓展引导标签 | 29h |
| 4 | 分区序号 | 49892E45h |
| 11 | 卷标 | NO NAME |
| 8 | 系统 ID | FAT32 |

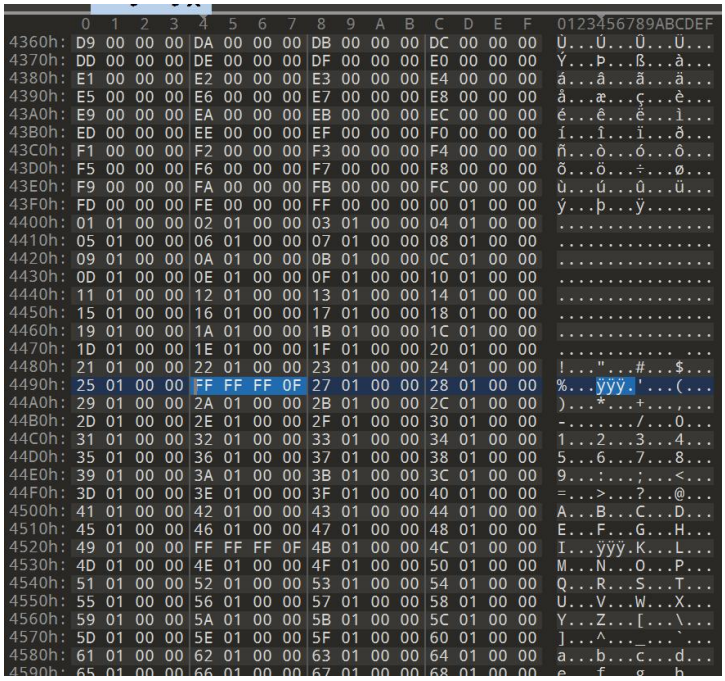
### 3.1.3 FAT 表

由 BPB 中的各项值可知，一簇为 8 个扇区，每个扇区 512 个字节，所以一簇共有 8*512=4KB

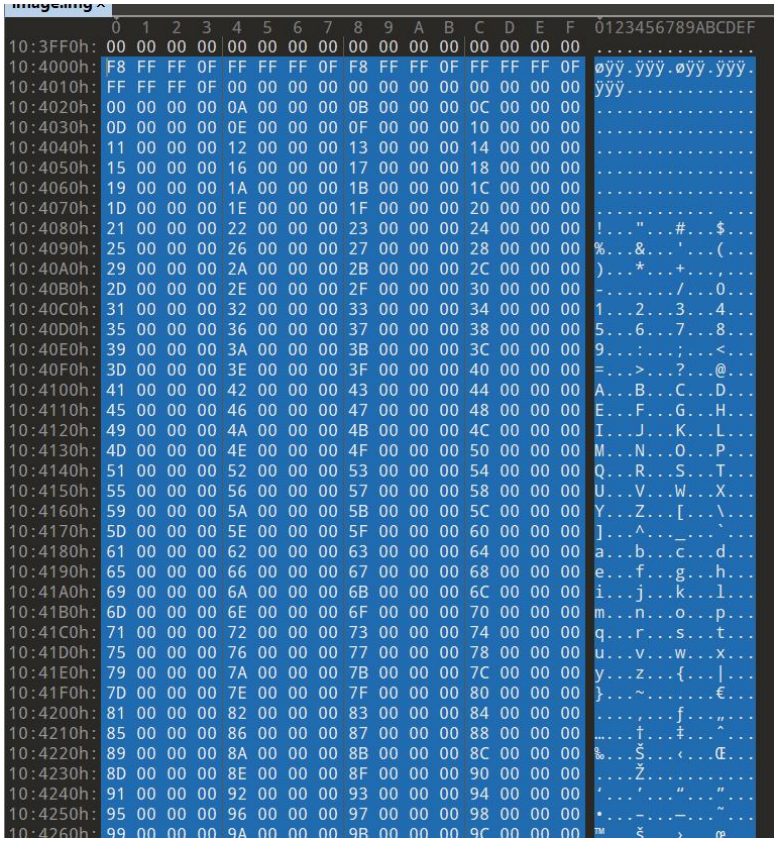
DBR 的保留扇区数为 20h=32 个，由此可以知道 FAT1 的起始位置为 DBR 所在位置+保留扇区数*扇区大小=32*512=4000h,跳转到对应的位置即可得到 FAT1 表的内容如下图所示



在该 FAT 表中从第 9 簇开始分析其内容为 0000000A，意为文件起始为第 9 簇，下一簇为第 10 簇，查看第 10 簇的内容为 0000000B,指向的下一簇为第 11 簇，由于该文件创建时为直接创建 1MB 的数据，所以文件的簇号应为连续的簇号，依次查看每一个簇指向的位置可以看出第 292 簇指向的位置为第 293 簇，而第 293 簇的位置值为 0FFFFFFF 为文件的结束标志，该文件的簇列表结束。

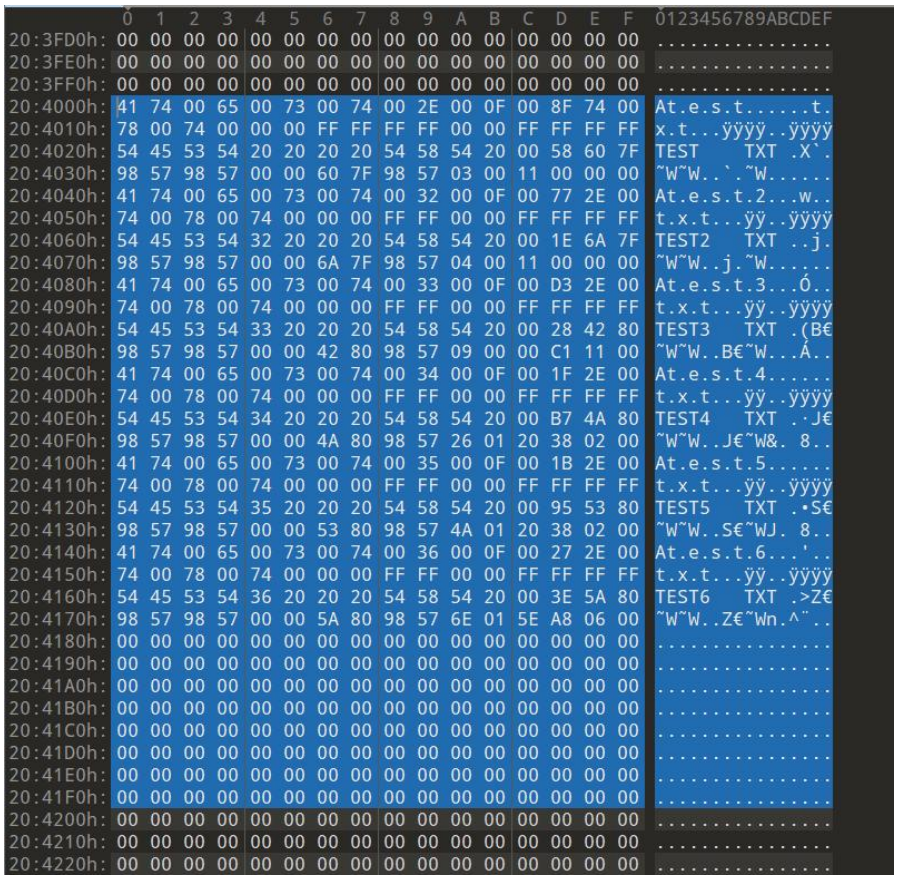


然后查找 FAT2 的起始位置=FAT1+FAT 扇区数*扇区大小=4000h+800h*512=104000h,跳转到对应的位置如下图所示，为 FAT2 表的内容

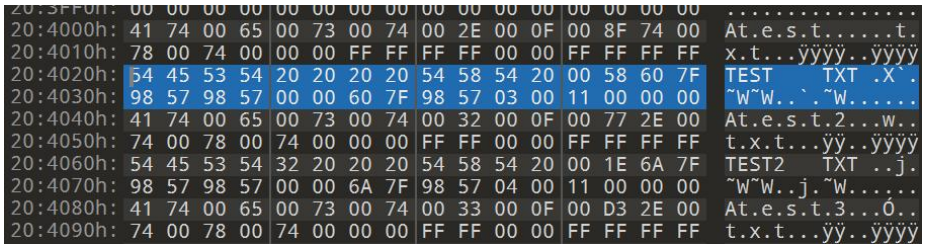


### 3.1.4 数据区

数据区的起始位置为 FAT2 起始位置+FAT 扇区数+扇区大小=104000h+800h*512=204000h,跳转到对应的位置如下图所示，为 FAT32 文件系统的数据区




每一个文件在数据区中都有其短目录项，其中文件 TEST 的短目录项内容如下图所示。




对于短目录项中每一个字段的解析如下表所示

| 字节偏移 | 字节数 | 定义 |
|---|---|---|
| 0x0-0x7 | 8 | 文件名 |
| 0x8-0xA | 3 | 拓展名 |
| 0xB | 1 | 属性字节 |
| 0xC | 1 | 系统保留 |
| 0xD | 1 | 创建时间的 10 毫秒位 |
| 0xE-0xF | 2 | 文件创建时间 |
| 0x10-0x11 | 2 | 文件创建日期 |
| 0x12-0x13 | 2 | 文件最后访问日期 |

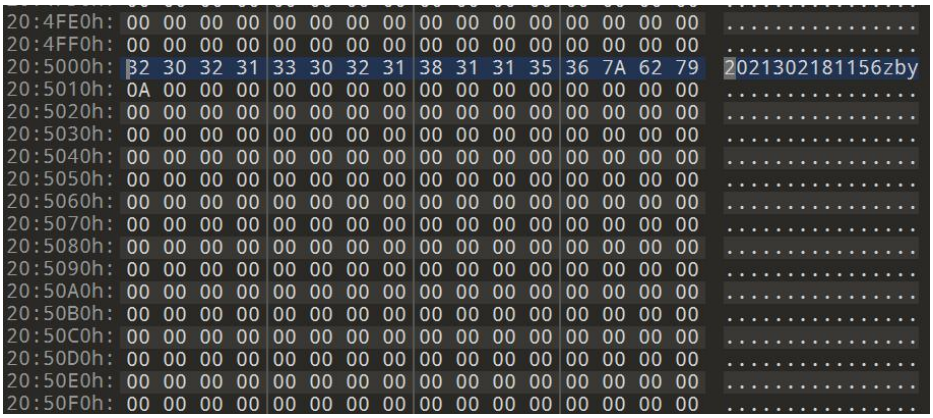
| | | |
|---|---|---|
| 0x14-0x15 | 2 | 文件起始簇号的高 16 位 |
| 0x16-0x17 | 2 | 文件的最近修改时间 |
| 0x18-0x19 | 2 | 文件的最近修改日期 |
| 0x1A-0x1B | 2 | 文件起始簇号的低 16 位 |
| 0x1C-0x1F | 4 | 表示文件的长度 |

在文件的属性字节中，各个字节值的含义如下表所示

| 字节值 | 含义 |
|---|---|
| 00000000 | 读写 |
| 00000001 | 只读 |
| 00000010 | 隐藏 |
| 00000100 | 系统 |
| 00001000 | 卷标 |
| 00010000 | 子目录 |
| 00100000 | 归档 |

通过观察第一个文件的目录项的起始簇号字节可以得到，该文件的起始簇号为：0000 0003，表示该文件的起始簇号为 3；文件的大小为 11h=17 个字节

因为该文件的起始簇号为 3，一簇 4KB，所以文件的内容放置位置为数据区起始位置+4KB=205000h 观察该位置的内容如下图所示




能够找到对应的文件的内容。

## 3.2 安全删除原理

通过分析 FAT32 文件的内容得知，如果想要安全删除一整个文件的内容，有以下几个步骤：

（1）需要首先找到文件的起始簇号，并根据起始簇号找到文件的簇链表。

（2）将所有簇链表指向的簇写入一个随机值。

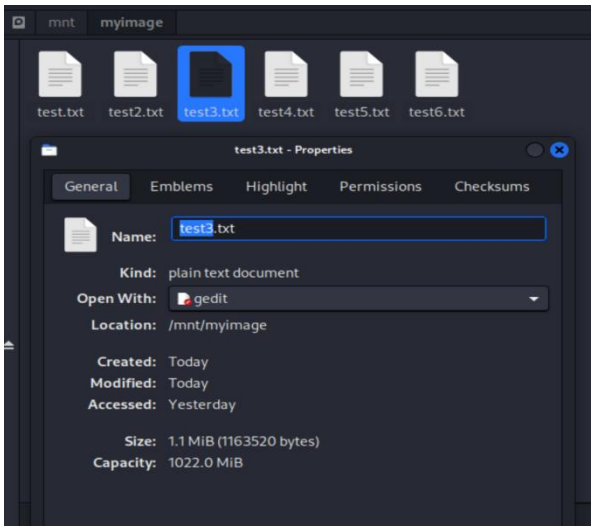
（3）然后将簇链表清空。

（4）最后将文件的目录项清空。

# 4 实验步骤

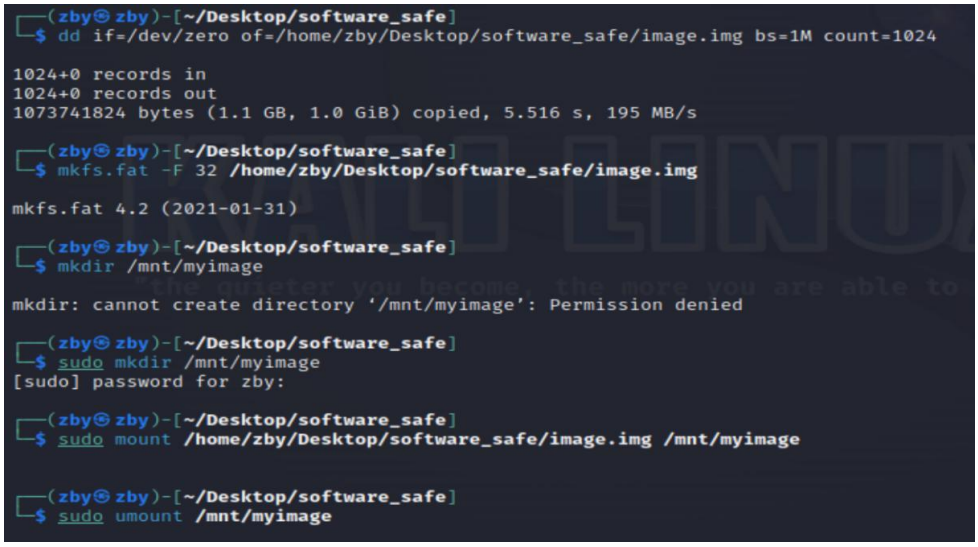
## 4.1 生成 FAT32 文件系统映像文件

首先执行如下指令创建一个空的映像文件
dd if=/dev/zero of=/home/zby/Desktop/software_safe/image.img bs=1M count=1024
然后执行如下指令创建 FAT32 文件系统
mkfs.fat -F 32 /home/zby/Desktop/software_safe/image.img
然后执行指令创建挂载点
sudo mkdir /mnt/myimage
之后挂载映像文件并添加文件
sudo mount /home/zby/Desktop/software_safe/image.img /mnt/myimage
在本次实验中向映像文件中添加了 6 个大小不同的文件如下图所示




最后执行指令卸载映像
sudo umount /mnt/myimage

最终命令行的执行结果如下图所示



## 4.2 编写安全删除程序

完整程序代码在实验目录下，以下对程序功能进行解释

### 4.2.1 读取映像文件 FAT32 参数

从 FAT32 映像文件中提取关键参数，以便其他函数能够根据这些参数来正确地读取、解析和操作 FAT32 文件系统。这些参数包括扇区大小、每簇的扇区数、保留扇区的数量、FAT 表的数量和大小，以及 FAT 表和数据区的起始位置。

首先读取 BIOS 参数块，然后计算 FAT 表和数据区的起始位置。

程序代码如下所示

```
def initialize_fat32_parameters(file_path):
    """初始化，获得映像文件参数信息"""
    global bytes_per_sector, sectors_per_cluster, data_start_sector, sectors_per_fat, \
        fat_start, reserved_sectors, number_of_fats

    with open(file_path, 'rb') as f:
        # 读取BPB（BIOS参数块）中的一些基本信息
        f.seek(11, 0)
        bytes_per_sector = struct.unpack('H', f.read(2))[0]
        sectors_per_cluster = struct.unpack('B', f.read(1))[0]
        reserved_sectors = struct.unpack('H', f.read(2))[0]
        number_of_fats = struct.unpack('B', f.read(1))[0]

        # 跳过一些不必要的信息
        f.seek(36, 0)
        sectors_per_fat = struct.unpack('I', f.read(4))[0]

        # 计算FAT表和数据区的起始位置
        fat_start = reserved_sectors
        data_start_sector = reserved_sectors + number_of_fats * sectors_per_fat
```

### 4.2.2 读取映像文件得到所有文件名

该步骤是从 FAT32 格式的磁盘映像文件中提取所有文件的文件名首先定位到根目录，之后每次读取根目录项 32 字节作为一个目录项，之后选取目录项中表示文件名称的字段保存。

该阶段的代码如下所示

```
def get_filename(file_path):
    """取得映像文件中所有的文件名"""
    global bytes_per_sector, sectors_per_cluster, reserved_sectors, number_of_fats, sectors_per_fat
    filename = []
```

```
    # 打开IMG文件
    with open(file_path, 'rb') as f:

        # 定位到根目录
        root_dir_sectors = (reserved_sectors + (number_of_fats * sectors_per_fat)
) * bytes_per_sector
        f.seek(root_dir_sectors, 0)

        # 读取根目录项
        while True:
            # 读取32字节的目录项
            dir_entry = f.read(32)
            if not dir_entry or dir_entry[0] == 0xE5:
                # 如果目录项为空或已删除，则跳过
                continue
            if dir_entry[0] == 0x00:
                # 如果目录项为0，则表示目录项列表结束
                break
            # 解析文件名
            filename.append(dir_entry[0:11].decode('ascii').rstrip())
    return filename
```

### 4.2.3 查找文件簇链

遍历根目录，查找与 filename_to_find 匹配的目录项，找到其起始簇号后遍历 FAT 表，从 start_cluster 开始，追踪文件所占据的簇使用全局变量 bytes_per_sector 和 fat_start 来定位 FAT 表中对应簇的条目。循环直到遇到簇链的结束标记（0x0FFFFFF8）。

该部分的代码如下所示。

```
def get_cluster_chain(f, start_cluster):
    """获取文件簇链"""

    global bytes_per_sector, fat_start
    cluster_chain = []
    current_cluster = start_cluster
    while current_cluster < 0x0FFFFFF8:
        cluster_chain.append(current_cluster)
        fat_offset = current_cluster * 4
        f.seek(fat_start * bytes_per_sector + fat_offset)
        current_cluster = struct.unpack('I', f.read(4))[0]
    return cluster_chain


```

```
def get_file_cluster_chain(file_path, filename_to_find):
    """找到文件名的簇链表"""
    global bytes_per_sector, sectors_per_cluster, reserved_sectors, number_of_fats, sectors_per_fat, fat_start
    with open(file_path, 'rb') as f:

        # 定位到根目录
        root_dir_sectors = (reserved_sectors + (number_of_fats * sectors_per_fat)) * bytes_per_sector
        f.seek(root_dir_sectors, 0)

        # 搜索文件
        while True:
            dir_entry = f.read(32)
            if not dir_entry or dir_entry[0] == 0xE5:
                continue
            if dir_entry[0] == 0x00:
                break

            filename = dir_entry[0:11].decode('ascii').rstrip()
            if filename == filename_to_find.upper():
                start_cluster = struct.unpack('H', dir_entry[26:28])[0] | (
                    struct.unpack('H', dir_entry[20:22])[0] << 16)
                cluster_chain = get_cluster_chain(f, start_cluster)
                return cluster_chain  # 返回簇链表

        return None  # 如果找不到文件，返回None
```

### 4.2.4 替换每一簇内容

在得到选定文件的簇链之后，将簇链中的每一簇内容都替换为相同长度的随机字节，以此实现对文件内容的删除。

生成随机内容簇和替换簇函数的程序代码如下所示

```
def write_cluster_data(file_path, cluster_number, new_data):
    """将数据写入簇号的指定位置"""
    global bytes_per_sector, sectors_per_cluster, data_start_sector

    cluster_offset = (cluster_number - 2) * sectors_per_cluster
    cluster_sector = data_start_sector + cluster_offset

    with open(file_path, 'r+b') as f:  # 'r+b' 模式用于读写
        f.seek(cluster_sector * bytes_per_sector)
        f.write(new_data)
def replace_with_random_bytes(cluster_data):
    """替换 cluster_data 的内容为相同长度的随机字节"""
```

```
    # 获取原始数据的长度
    data_length = len(cluster_data)

    # 生成相同长度的随机字节
    random_bytes = bytes(random.getrandbits(8) for _ in range(data_length))

    return random_bytes
```

### 4.2.5 清理文件目录表

首先定位到根目录，然后搜索并定位目录项最后清除匹配的目录项，将对应项的内容全部设置为 0

清理文件目录表的程序代码如下所示

```
def clear_directory_entry(file_path, filename_to_clear):
    """清理文件目录表"""
    global bytes_per_sector, sectors_per_cluster, reserved_sectors, number_of_fats, sectors_per_fat
    with open(file_path, 'r+b') as f:
        # 计算根目录的起始位置
        root_dir_sectors = (reserved_sectors + (number_of_fats * sectors_per_fat)) * bytes_per_sector
        f.seek(root_dir_sectors, 0)

        # 搜索文件
        while True:
            dir_entry_pos = f.tell()
            dir_entry = f.read(32)
            if not dir_entry or dir_entry[0] == 0xE5:
                continue
            if dir_entry[0] == 0x00:
                break

            filename = dir_entry[0:11].decode('ascii').rstrip()
            if filename == filename_to_clear.upper():
                f.seek(dir_entry_pos)
                f.write(b'\x00' * 32)  # 清零目录项
                print(filename_to_clear + "文件的目录表已经被清除")
                return

    print(filename_to_clear + "文件不存在")
```

### 4.2.6 清理 FAT 表中的簇链

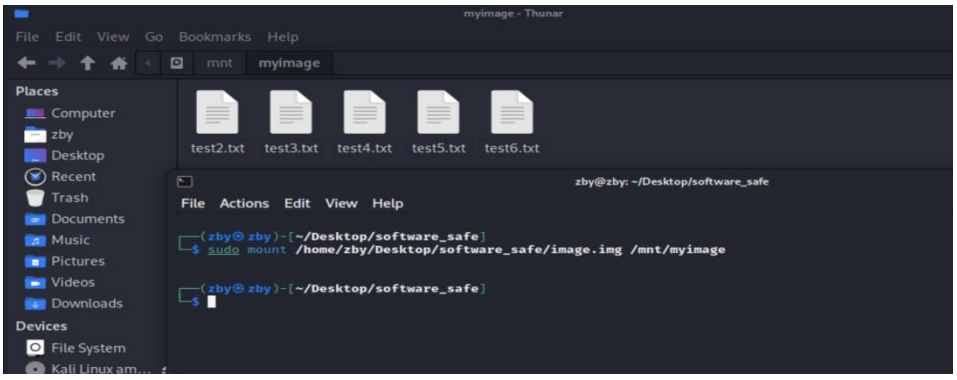
为了在实验中形成对比，在本次实验中只清理 FAT1 中的内容，遍历簇链然后将簇链中的每个值都清为 0。

清理程序如下所示

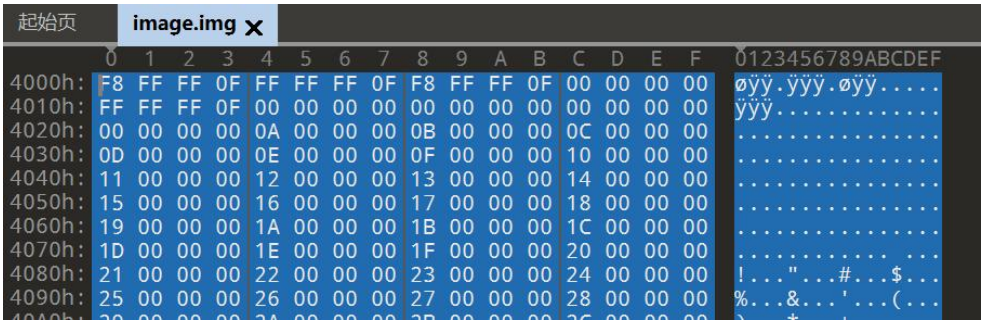
```
def clear_fat_chain(file_path, cluster_chain):
    """清理文件簇链表"""
    global bytes_per_sector, fat_start

    with open(file_path, 'r+b') as f:
        # 将选中文件的簇链表所有项替换为0
        for cluster in cluster_chain:
            # FAT 表中每个簇的条目是4 字节
            fat_offset = cluster * 4
            f.seek(fat_start * bytes_per_sector + fat_offset)
            f.write(b'\x00\x00\x00\x00')
```

# 5 实验结果

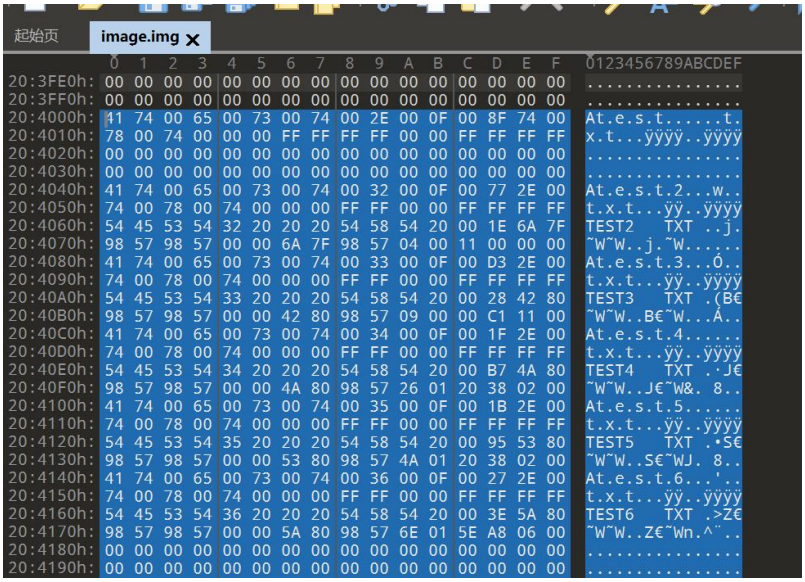
将程序修改之后的磁盘映像重新发送到虚拟机中将其挂载到文件目录下能够看到 test 已经被删除成功



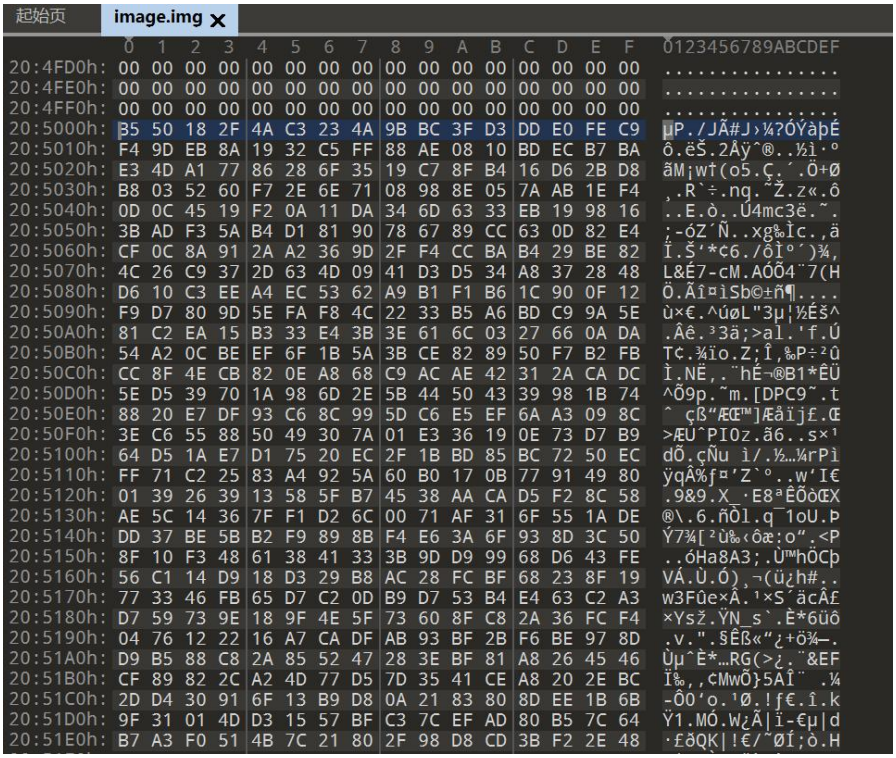

使用 010Editor 将该磁盘映像打开之后定位到簇链表之后得到的结果如下图所示，发现文件 test 的簇链已经被清 0



将磁盘映像定位到目录表中之后可以看到 TEST 所对应的表项被清 0




定位到该文件原本的数据区可以看到，数据区的内容被随机字符取代。




由此可得文件被成功安全删除

# 6 实验心得体会

在这次实验中，我深入了解了 FAT32 文件系统的结构和数据安全删除的重要性。通过操作 FAT 表和覆写数据簇，我不仅掌握了理论知识，还学会了如何实际应用这些技术。实验过程中的挑战使我意识到理论和实践之间的差异，但通过解决这些问题，我增强了问题解决能力和实际操作技能。