北京科技大学实验报告

学院：计算机与通信工程学院　专业：计算机科学与技术　班级：计 1703

姓名：　张宝丰　　学号：41724081　　实验日期：2019 年　12 月　1 日

实验名称：操作系统实验 6　FAT12 文件系统（3 分）

实验目的：以一个教学型操作系统 EOS 为例，理解磁盘存储器管理的基本原理与文件系统的实现方法；能对核心源代码进行分析和修改，具备实现一个简单文件系统的基本能力；训练分析问题、解决问题以及自主学习能力，通过 6 个实验的实践，达到能独立对小型操作系统的部分功能进行分析、设计和实现。

实验环境：EOS 操作系统及其实验环境。

实验内容：

通过调用 EOS API 读取文件数据，跟踪 FAT12 文件系统的读文件功能，分析 EOS 中 FAT12 文件系统的相关源代码，理解并阐述 EOS 实现 FAT12 文件系统的方法；修改 EOS 的源代码，为 FAT12 文件系统添加写文件功能，。

实验步骤：

1）EOS 中 FAT12 文件系统相关源代码分析

（分析 EOS 中 FAT12 文件系统的相关源代码，简要说明 EOS 实现 FAT12 文件系统的方法，包括主要数据结构与文件基本操作的实现等）

FAT12 是文件分配表，每个表项占用 12 位，对应着一个簇（簇是一个逻辑概念），EOS 操作系统中的每个簇只对应一个磁盘扇区。表项存放着对应簇的具体信息以及该文件下一个簇的簇号，相当于把一个文件占用的内存空间用一个链表组织起来。那么如何识别一个文件到了末尾呢？FAT12 是通过规定一些特殊的簇号来实现的，比如 0x00 的簇号表示该簇未被占用，而大于等于 0xFF8 的簇号表示当前表项对应的簇是文件占用的最后一个簇。

基于上面的簇分配思路，EOS 为用户提供了打开文件/关闭文件,读/写文件等处理文件的方法。

首先是打开文件，EOS 提供 CreateFile 作为用户打开文件的 API，其定义如下：

```
CreateFile(
    IN PCSTR FileName,
    IN ULONG DesiredAccess,
    IN ULONG ShareMode,
    IN ULONG CreationDisposition,
    IN ULONG FlagsAndAttributes
    )
```

在打开文件的过程中，FAT12 户根据 FileName 创建文件对象并获取句柄，截止处理访问权限等其他信息。打开文件会经历如下的调用过程：CreateFile -> FatCreate -> FatOpenExistingFile -> FatOpenFile -> FatOpenFileInDiretory。虽然 CreateFile 的参数中提供了创建文件、删除文件、截断文件的 API，但是在 FatCreateFile 只实现了打开文件 OpenExisting，这会影响后面的实验当中用小文件覆盖大文件的实现。

```
STATUS
FatCreate(
    IN PDEVICE_OBJECT DeviceObject,
    IN PCSTR FileName,
    IN ULONG CreationDisposition,
    IN OUT PFILE_OBJECT FileObject
    )
{
    //
    // 检查路径是否有效。
    //
    if (!FatCheckPath(FileName, FALSE)) {
        return STATUS_PATH_SYNTAX_BAD;
    }

    //
    // 目前仅实现了打开已有文件。
    //
    if (OPEN_EXISTING == CreationDisposition) {

        return FatOpenExistingFile(DeviceObject, FileName, FileObject);

    } else if (CREATE_NEW == CreationDisposition) {

        return STATUS_NOT_SUPPORTED;

    } else if (CREATE_ALWAYS == CreationDisposition) {

        return STATUS_NOT_SUPPORTED;

    } else if (OPEN_ALWAYS == CreationDisposition) {
```

```
            return STATUS_NOT_SUPPORTED;

        } else if (TRUNCATE_EXISTING == CreationDisposition) {

            return STATUS_NOT_SUPPORTED;
        }
}
```

关闭文件的实现就要简单许多，在 eosapi.c 中定义了给用户关闭文件的 API：

```
EOSAPI
BOOL
CloseHandle(
    IN HANDLE Handle
    )
{
    STATUS Status;

    Status = ObCloseHandle(Handle);

    PsSetLastError(TranslateStatusToError(Status));

    return EOS_SUCCESS(Status);
}
```

关闭文件实际上是从内存中删除文件对象和文件控制块（FCB）的过程，可以从上面代码中看到，关闭文件只需要获取文件的 Handle，接着调用 ObCloseHandle -> ObDerefObject，令文件引用数-1，当文件的引用数减小为 0 时，则会在 ObDerefObject 函数中删除对象，并调用 FatClose -> FatCloseFile 来关闭文件。

读文件：

从文件系统中读取一个文件需要经历 3 个模块的过程：通过文件路径从 Object 模块获取文件对象句柄，接着通过 IO 模块的 IopReadFileObject 通过句柄读取文件对象，最后通过 FAT12 文件系统的 FatRead 和 FatReadFile 来读取实际的文件内容。

主要的逻辑在 FatReadFile 中实现：

```
STATUS
FatReadFile(
    IN PVCB Vcb,
    IN PFCB File,
    IN ULONG Offset,
    IN ULONG BytesToRead,
```

OUT PVOID Buffer,

OUT PULONG BytesRead

)

参数列表：

Offset -- 读取的文件内起始偏移地址。

BytesToRead -- 期望读取的字节数。

Buffer -- 用于存放读取数据的缓冲区的指针。

BytesRead -- 实际读取的字节数。

FatReadFile 会首先通过 Offset 和簇的大小计算出读起始位置对应的簇号，接着通过循环不断读取簇的内容，直到达到文件末尾（读取大小==目标大小||下一个簇号>=0xFF8）：

```
for (i = Offset / FatBytesPerCluster(&Vcb->Bpb); i > 0; i--) {
        Cluster = FatGetFatEntryValue(Vcb, Cluster);
    }
```

写文件：

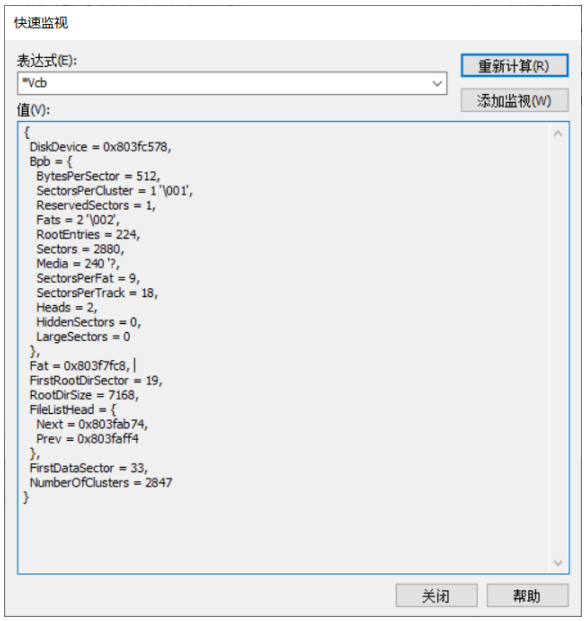
写文件的思路和读取文件类似，但是 EOS 只实现了单扇区内的写入，若文件跨越多个扇区，则超过一个扇区的内容会丢失。在 3）中我仿照 FatReadFile 的逻辑扩展了 FatWriteFile 的功能，使其支持跨扇区读写。

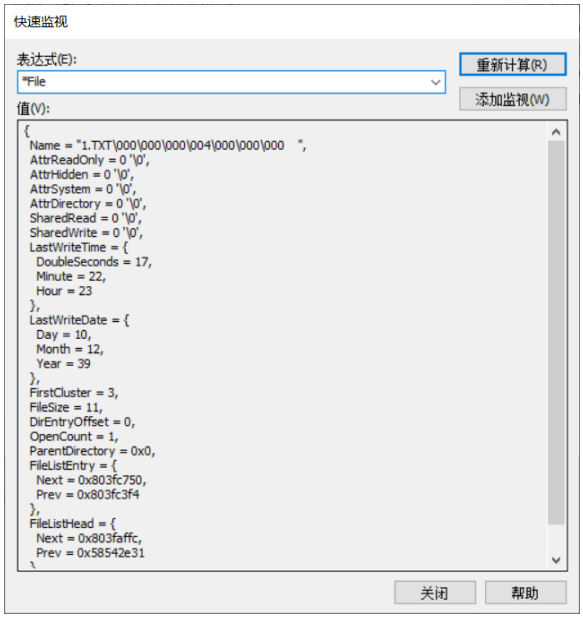
2）EOS 中 FAT12 文件系统读文件过程的跟踪

```
124     }
125     else
126     {
127         hOutput = GetStdHandle(STD_OUTPUT_HANDLE);
128     }
129
130     while (TRUE)
131     {
132         //
133         // 尝试读取 BUFFER_SIZE 个字节，实际读取到的字节数由 m 返回。
134         //
135         ReadFile(hFileRead, Buffer, BUFFER_SIZE, &m);
136
137         //
138         // 将实际读取到的 m 个字节写入输出句柄。
139         //
140
141         if (4 == argc && 0 == stricmp(argv[3], "-o"))
142         {
143             m = m + 0x10000000;
144         }
145
146         if (!WriteFile(hOutput, Buffer, m, &n))
```

```
    //
    // 由对象句柄得到对象指针
    //
    Status = ObRefObjectByHandle(Handle, NULL, &Object);

    if (EOS_SUCCESS(Status)) {

        //
        // 由对象指针得到注册的对象类型
        //
        Type = OBJECT_TO_OBJECT_TYPE(Object);

        if (NULL != Type->Read) {
            //
            // 调用对象类型中注册的 Read 操作
            //
            Status = Type->Read(Object, Buffer, NumberOfBytesToRead, NumberOfBytesRead);
        } else {
            Status = STATUS_INVALID_HANDLE;
        }

        ObDerefObject(Object);
    }
```

```
功能描述：
    在文件的指定偏移处读取指定字节的数据，实际读取的字节数可能受到文件长度的限制
    而小于期望值。

参数：
    Vcb -- 卷控制块指针。
    File -- 文件控制块指针。
    Offset -- 读取的文件内起始偏移地址。
    BytesToRead -- 期望读取的字节数。
    Buffer -- 用于存放读取数据的缓冲区的指针。
    BytesRead -- 实际读取的字节数。

返回值：
    如果成功则返回STATUS_SUCCESS。

--*/
{
    STATUS Status;
    ULONG i;
    ULONG ReadCount = 0;
    USHORT Cluster;
    ULONG FirstSectorOfCluster;
    ULONG OffsetInSector;
```





FirstSectorOfCluster = Vcb->FirstDataSector + (Cluster - 2) * Vcb->Bpb.SectorsPerCluster;

3）为 EOS 的 FAT12 文件系统添加写文件功能

（给出实现方法的简要描述、源代码、测试及结果等）

写文件可以仿照读文件的逻辑实现，通过 Offset 计算出文件起始的簇号，并依据文件大小、下一簇号判断是否已经读完；若没有读取完毕，则继续读出下一个簇中的内容。下面是我修改的 FatWriteFile 的源代码，支持跨扇区写入、覆盖写入（清空原来的文件）。

```
STATUS
FatWriteFile(
    IN PVCB Vcb,
    IN PFCB File,
    IN ULONG Offset,
    IN ULONG BytesToWrite,
    IN PVOID Buffer,
    OUT PULONG BytesWriten
    )
{
    STATUS Status;

    // 由于在将新分配的簇插入簇链尾部时，必须知道前一个簇的簇号，
```

```
// 所以定义了“前一个簇号”和“当前簇号”两个变量。
USHORT PrevClusterNum, CurrentClusterNum;
USHORT NewClusterNum;
ULONG ClusterIndex;
ULONG FirstSectorOfCluster;
ULONG OffsetInSector;

// ---------- add code ----------
ULONG WriteCount = 0;        // 已经写入的字节数
ULONG BytesToWriteInSector; // 在当前扇区要写入的字节数

ULONG i;
USHORT now;
USHORT next;
// 这里通过 BytesToWrite 的大小来判断是否进行覆盖写
// 如果 BytesToWrite 过大，说明是覆盖写模式，则将文件簇清空，再从头写入
if( BytesToWrite > 0x10000000){
    BytesToWrite -= 0x10000000;

    next = 0;
    now = File->FirstCluster;
    if(now!=0){
        File->FirstCluster=0;
        while(next<0xFF8){
            next = FatGetFatEntryValue(Vcb,now);
            FatSetFatEntryValue(Vcb,now,0);
            now = next;
            if(next=0xFF8)
                break;
        }
        File->FileSize = 0;
        if(!File->AttrDirectory)
            FatWriteDirEntry(Vcb,File);

    }
}
```

```
// 写入的起始位置不能超出文件大小（并不影响增加文件大小或增加簇，想想原因？）
if (Offset > File->FileSize)
    return STATUS_SUCCESS;

// 根据簇的大小，计算写入的起始位置在簇链的第几个簇中（从 0 开始计数）
ClusterIndex = Offset / FatBytesPerCluster(&Vcb->Bpb);

// 顺着簇链向后查找写入的起始位置所在簇的簇号。
PrevClusterNum = 0;
CurrentClusterNum = File->FirstCluster;
for (i = ClusterIndex; i > 0; i--) {
    PrevClusterNum = CurrentClusterNum;
    CurrentClusterNum = FatGetFatEntryValue(Vcb, PrevClusterNum);
}

// 如果写入的起始位置还没有对应的簇，就增加簇
if (0 == CurrentClusterNum || CurrentClusterNum >= 0xFF8) {

    // 为文件分配一个空闲簇
    FatAllocateOneCluster(Vcb, &NewClusterNum);

    // 将新分配的簇安装到簇链中
    if (0 == File->FirstCluster)
        File->FirstCluster = NewClusterNum;
    else
        FatSetFatEntryValue(Vcb, PrevClusterNum, NewClusterNum);

    CurrentClusterNum = NewClusterNum;
}

// 计算当前簇的第一个扇区的扇区号。簇从 2 开始计数。
// FirstSectorOfCluster = Vcb->FirstDataSector + (CurrentClusterNum - 2) *
Vcb->Bpb.SectorsPerCluster;

// 计算写位置在扇区内的字节偏移。
OffsetInSector = Offset % Vcb->Bpb.BytesPerSector;
```

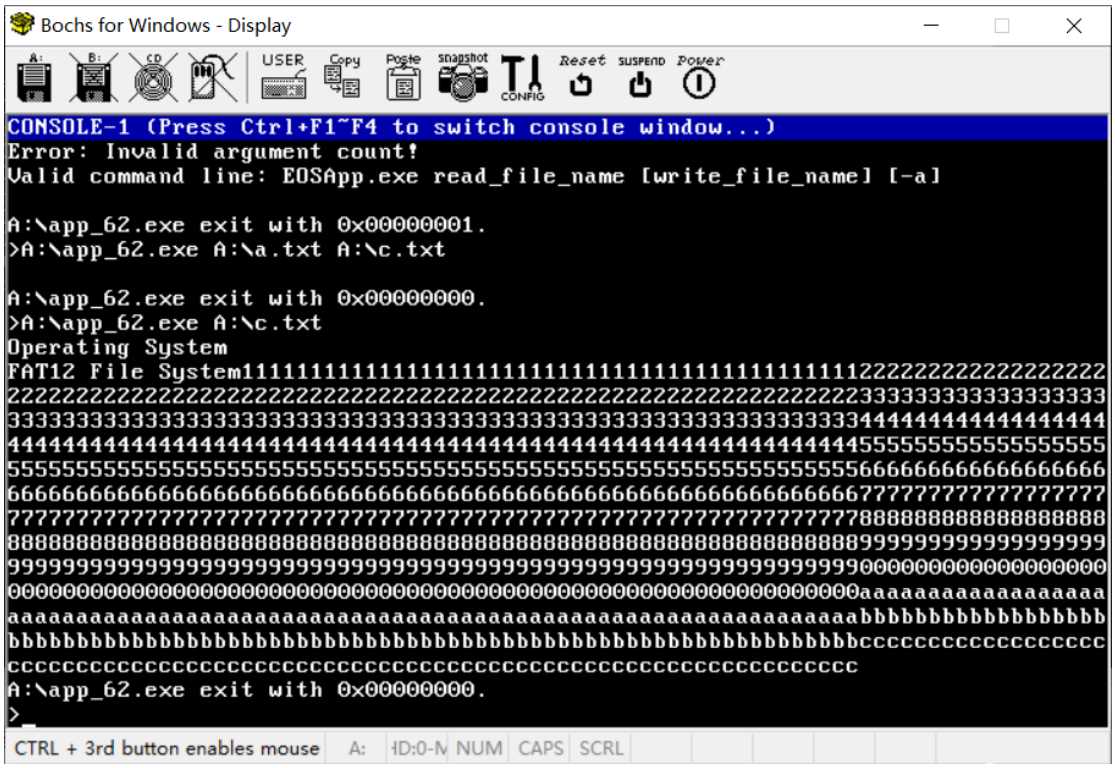
```
    //
    // 从偏移位置所在的簇开始向后写文件的簇，直到全部写完
    //
    for (;;) {

        //
        // 簇由多个连续扇区组成，计算簇的起始扇区号。
        //
        FirstSectorOfCluster = Vcb->FirstDataSector + (CurrentClusterNum - 2) *
Vcb->Bpb.SectorsPerCluster;

        //
        // 计算偏移位置在簇内的第几个扇区，然后从这个扇区开始读取簇内的连续扇区。
        //
        for (i = ((Offset + WriteCount) / Vcb->Bpb.BytesPerSector) %
Vcb->Bpb.SectorsPerCluster;
             i < Vcb->Bpb.SectorsPerCluster; i++ ) {

            //
            // 计算写入位置在扇区内的字节偏移。
            //
            OffsetInSector = (Offset + WriteCount) % Vcb->Bpb.BytesPerSector;

            //
            // 计算需要在这个扇区内写入的字节数。
            //
            if (BytesToWrite - WriteCount > Vcb->Bpb.BytesPerSector - OffsetInSector) {
                BytesToWriteInSector = Vcb->Bpb.BytesPerSector - OffsetInSector;
            } else {
                BytesToWriteInSector = BytesToWrite - WriteCount;
            }

            //
            // 写入扇区数据。
            //
            Status = IopReadWriteSector( Vcb->DiskDevice,
                                         FirstSectorOfCluster + i ,
```

```
                                        OffsetInSector,
                                        (PCHAR)Buffer + WriteCount,
                                        BytesToWriteInSector,
                                        FALSE );

            if (!EOS_SUCCESS(Status)) {
                return Status;
            }

            //
            // 如果写入完成则返回。
            //
            WriteCount += BytesToWriteInSector;
            if (WriteCount == BytesToWrite) {
                // 如果文件长度增加了则必须修改文件的长度。
                if (Offset + BytesToWrite > File->FileSize) {
                    File->FileSize = Offset + BytesToWrite;

                    // 如果是数据文件则需要同步修改文件在磁盘上对应的 DIRENT 结构
                    // 体。目录文件的 DIRENT 结构体中的 FileSize 永远为 0，无需修改。
                    if (!File->AttrDirectory)
                        FatWriteDirEntry(Vcb, File);
                }

                // 返回实际写入的字节数量
                *BytesWriten = BytesToWrite;

                return STATUS_SUCCESS;
            }
        }

        //
        // 继续读文件的下一个簇。
        //
        PrevClusterNum = CurrentClusterNum;
```

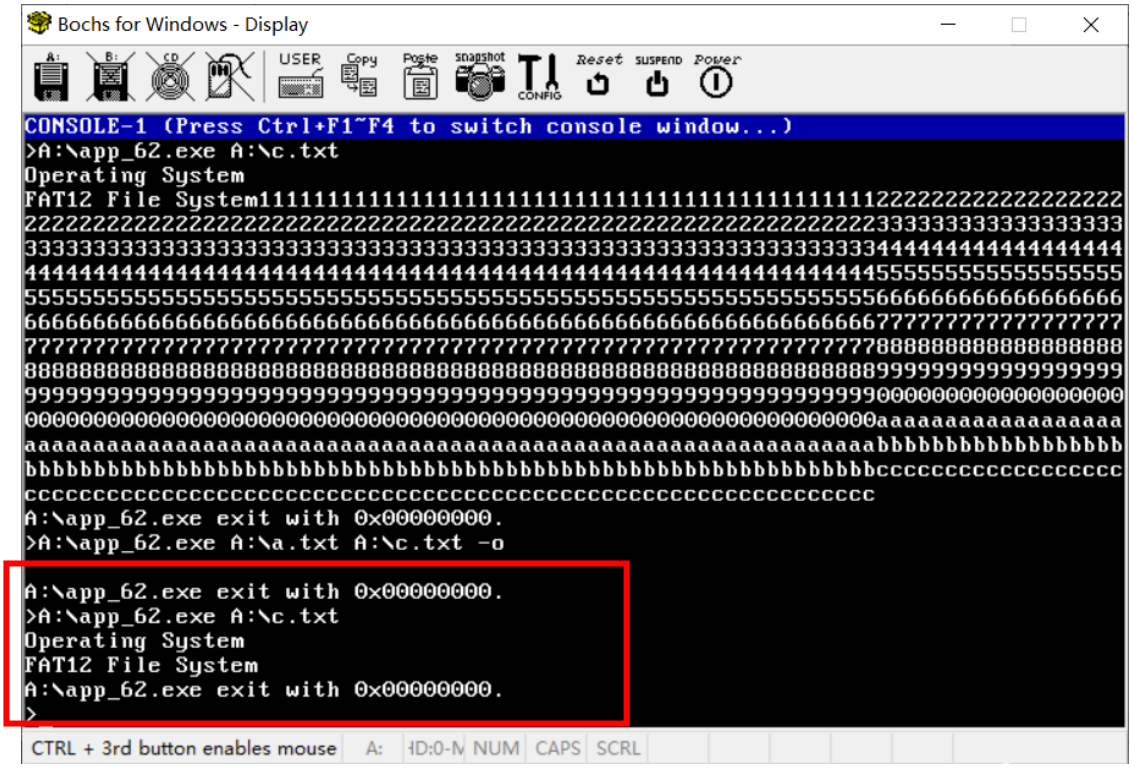
```
            CurrentClusterNum = FatGetFatEntryValue(Vcb, PrevClusterNum);

            // 如果文件尚未写完，但簇不够了，则分配一个新的簇
            if (0 == CurrentClusterNum || CurrentClusterNum >= 0xFF8) {

                // 为文件分配一个空闲簇
                FatAllocateOneCluster(Vcb, &NewClusterNum);

                // 将新分配的簇安装到簇链中
                if (0 == File->FirstCluster)
                    File->FirstCluster = NewClusterNum;
                else
                    FatSetFatEntryValue(Vcb, PrevClusterNum, NewClusterNum);

                CurrentClusterNum = NewClusterNum;
            }
        }

        // 如果文件长度增加了则必须修改文件的长度。
        if (Offset + BytesToWrite > File->FileSize) {
            File->FileSize = Offset + BytesToWrite;

            // 如果是数据文件则需要同步修改文件在磁盘上对应的 DIRENT 结构
            // 体。目录文件的 DIRENT 结构体中的 FileSize 永远为 0，无需修改。
            if (!File->AttrDirectory)
                FatWriteDirEntry(Vcb, File);
        }

        // 返回实际写入的字节数量
        *BytesWriten = BytesToWrite;

        return STATUS_SUCCESS;
    }
```

这里实现覆盖写利用了 BytesToWrite 传递信息，若在控制台命令的末尾加入了-o，我的程序会将 BytesToWrite 加上 0x10000000，使其变为一个很大的值，再在 FatWriteFile 中判断其是否为覆盖写，如果是，则将 BytesToWrite 还原为正常值，并将文件簇清空并释放，再重新写入。

下面是两种读写的演示，其一为正常写入，可以看到 A 中的内容直接写在了 C 里面，但是不影响 C 后面字节的内容。



第二是覆盖写，可以看到 A 中的内容覆盖了 C 的内容，C 中多余的数据被删除。



结果分析：

本次实验我研究了 EOS 实现文件系统的方式，EOS 使用 FAT12 文件系统管理文件，每个

FAT 目录项占 32 个字节，最多存储 244 个目录项，也就是说只能软盘之中最多只能有 244 个文件。每个目录项对应一个文件的起始簇，一个文件的全部簇用链表进行组织。在读写文件的过程中，EOS 会根据路径打开现有的文件，主要的步骤有根据路径获取文件句柄、打开文件对象，再通过 FAT 文件系统对文件进行操作。在阅读源码的过程中，我发现每一层的设计都会有异常处理，比如识别不合法的文件路径、判断文件对象是否存在等，这些步骤为操作系统的稳定性提供了保障。

在实现写文件的过程中，我借鉴了读取文件的逻辑，但是不同的是，待写入的文件有可能是个空文件，也就是还没有为它分配簇，或者写入的内容超过了当前簇的容量，这时需要为文件分配额外的簇。接下来就是通过 Offset 计算文件的起始簇，通过双重循环写入了。我的代码中在 While 循环内外各写了一次为文件分配簇的操作，有些代码冗余，可以考虑将这步操作封装为函数，以优化代码逻辑。