**北京科技大学实验报告**

学院： 计通学院 专业： 计算机科学与技术 班级： 计172

姓名： 金玉卿 学号： 41724051 实验日期： 2019 年 12 月 10 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读写文件实现系统**

在 EOS 主函数中，函数开始执行，首先调用CreatFile函数代开要读取的文件，将读取 的数据写入文件，如果未成功，则将失败结果输出到控制台，成功的话仍调用CreatFile打开需要写入的文件，判断是否增加写，如果是则将文件指针移动到文件的末尾，然后将写入文件句柄作为输出句柄，否则直接输出句柄。执行完后调用 ReadFile 函数读取文件，再判断是否已读取，读取成功关闭文件，结束函数，否则继续读取文件。

与FAT12相关的几个结构体可以在fat12.h文件中查看，fat12.h中定义了目录项结构体（DIRENT）、卷控制块（VCB）、文件控制块（FCB）等结构体。

* 1. **打开文件**

应用程序可以调用API函数CreateFile来打开要操作的文件，该函数的定义在api/eosapi.c中可以找到。

EOSAPI HANDLE CreateFile(

IN PCSTR FileName,

IN ULONG DesiredAccess,

IN ULONG ShareMode,

IN ULONG CreationDisposition,

IN ULONG FlagsAndAttributes

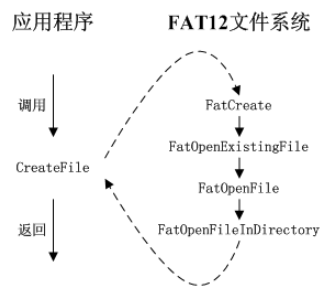
)

第一个参数 FileName 是文件的全路径，例如，要打开软盘根目录中的文件 a.txt，则这个参数就应该为“A:\\a.txt”。第二个参数 DesiredAccess 指定要打开文件的访问方式，如果这个参数的值为 GENERIC\_READ（在文件 inc/eosdef.h 中定义），则打开的文件为只读；如果这个参数的值为 GENERIC\_WRITE ，则打开的文件为只写的；如果这个参数的值为GENERIC\_READ|GENERIC\_WRITE，则打开的文件既是可读的，又是可写的。第三个参数 ShareMode 和第五个参数 FlagsAndAttributes 通常被设置为 0。

函数在执行的过程中会调用 FAT12 文件系统提供的相关函数，在main函数中通过以下语句实现对CreateFile函数的调用。

hFileRead = CreateFile(argv[1], GENERIC\_READ, 0, OPEN\_EXISTING, 0);

该函数的调用流程如下图所示。



CreateFile 函数会首先进入 IO 模块内的 IoCreateFile 函数（在文件io/io.c中定义）创建一个文件对象，然后再由 IO 模块进入 FatCreate 函数来打开文件。

如果CreateFile函数打开文件失败，会返回INVALID\_HANDLE\_VALUE。如果该函数成功打开了文件，会返回文件的句柄，该文件句柄供应用程序在调用其它操作文件的 API函数时使用，判断是否增加写，如果是将文件指针移动到文件的末尾，然后将写入文件句柄输出句柄，否则直接输出句柄。CreateFile函数在执行的过程中会调用 FAT12文件系统提供的相关函数，在FatOpenFileInDirectory 函数中，会初始化 FCB 并将之插入所在目录的文件链表中。

打开文件相关FAT12中的函数可以在fat12.c文件中找到，分析函数代码可以发现打开文件即创建文件对象和文件控制块，将文件的相关信息在内存中准备好，方便后续对文件的读写操作。

* 1. **关闭文件**

通过调用CloseHandle函数实现文件的关闭，CloseHandle函数的定义在api/eosapi.c中可以找到。

EOSAPI BOOL CloseHandle(

IN HANDLE Handle

)

参数 Handle 是由 CreateFile 函数返回的被打开文件的句柄。如果该函数成功关闭文件，返回 TRUE，否则返回 FALSE。该函数在关闭文件时同样会调用FAT12文件系统提供的相关函数，在main函数中通过以下语句实现文件的关闭。

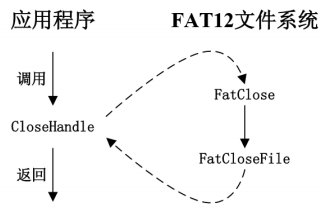
if (hFileRead != INVALID\_HANDLE\_VALUE)

CloseHandle(hFileRead);

if (hFileWrite != INVALID\_HANDLE\_VALUE)

CloseHandle(hFileWrite);

该函数的调用流程如下图所示。



在 FatCloseFile 函数中完成关闭文件的操作，主要工作是从内存中删除文件控制块。实际上，CloseHandle 函数会首先进入对象模块内的 ObCloseHandle 函数（在文件 ob/obhandle.c 中定义）尝试从内存中删除文件对象（如果文件对象的引用计数变为 0）。关闭文件的过程和打开文件正好相反，本质就是删除文件对象和文件控制块的过程。

* 1. **读文件**

在成功打开文件后，调用读文件API函数ReadFile将文件的数据（保存在软盘的数据区）读入内存，从而允许处理器对文件的数据进行特定的操作（例如打印输出到屏幕上）。ReadFile 函数在api/eosapi.c 文件中定义如下：

EOSAPI BOOL ReadFile(

IN HANDLE Handle,

OUT PVOID Buffer,

IN ULONG NumberOfBytesToRead,

OUT PULONG NumberOfBytesRead

)

参数 Handle 是由 CreateFile 函数返回的被打开文件的句柄。参数 Buffer 指向一块内存缓冲区，从软盘数据区读取到的文件数据就会放入此块内存缓冲区中。参数 NumberOfBytesToRead 指定本次调用希望读取到的字节数，显然该参数的值应该小于等于 Buffer 缓冲区的大小。参数 NumberOfBytesRead 是一个输出参数，用于返回本次调用实际读取到的字节数，显然该参数返回的值会小于等于 NumberOfBytesToRead。如果该函数成功从文件中读取到数据，就返回 TRUE，否则返回 FALSE。

在main函数中对函数的调用代码如下：

while (TRUE)

{

ReadFile(hFileRead, Buffer, BUFFER\_SIZE, &m);

// 将实际读取到的 m 个字节写入输出句柄。

if (!WriteFile(hOutput, Buffer, m, &n))

{

printf("Write file error: %d\n", GetLastError());

goto RETURN;

}

// 如果实际读取的字节数少于预期，说明文件读取完毕。

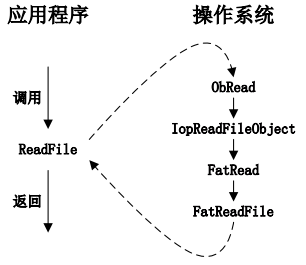
if (m < BUFFER\_SIZE)

break;

}

在调用CreateFile 函数以只读的方式打开软盘根目录中的文件后，在一个死循环中进行读文件操作，每轮循环都尝试从文件中读取BUFFER\_SIZE个字节的数据到 Buffer 缓冲区中，并将缓冲区中的数据打印输出到标准输出（屏幕）。当文件读取完毕后应该结束死循环。由参数m返回实际读取的字节数，当实际读取的字节数小于希望读取的字节数时，说明文件读取完毕。

ReadFile函数调用函数流程如下图：



其中的 ObRead 函数在文件ob/obmethod.c文件中定义，属于对象模块；IopReadFileObject 函数在文件io/file.c 文件中定义，属于 IO 模块；FatRead 函数和 FatReadFile 函数在文件 io/driver/fat12.c 文件中定义，属于 FAT12 文件系统模块。

* 1. **写文件**

在打开文件后，可以调用 API 函数 WriteFile 将内存中的数据写入文件。WriteFile 函数在api/eosapi.c 文件中定义如下：

EOSAPI BOOL WriteFile(

IN HANDLE Handle,

IN PVOID Buffer,

IN ULONG NumberOfBytesToWrite,

OUT PULONG NumberOfBytesWritten

)

参数 Handle 是由 CreateFile 函数返回的被打开文件的句柄。参数 Buffer 指向一块内存缓冲区，该缓冲区中的全部或部分数据会被写入文件中。参数 NumberOfBytesToWrite 指定本次调用希望写入文件的字节数，显然该参数的值应该小于等于 Buffer 缓冲区的大小。参数 NumberOfBytesWritten 是一个输出参数，用于返回本次调用实际写入文件的字节数，显然该参数返回的值会小于等于 NumberOfBytesToWrite。如果该函数成功向文件中写入数据，就返回 TRUE，否则返回 FALSE。

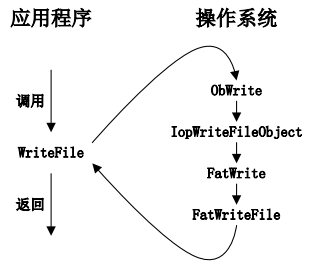
在main函数中对函数的调用代码如下：

SetFilePointer(hFileWrite, GetFileSize(hFileWrite), FILE\_BEGIN);

WriteFile(hOutput, Buffer, m, &n)

首先调用 CreateFile 函数以只写的方式打开软盘根目录中的文件。然后，调用 API 函数 SetFilePointer 将文件指针移动到文件的末尾。最后，调用WriteFile函数将字符串写入文件的末尾。

WriteFile函数调用函数流程如下图：

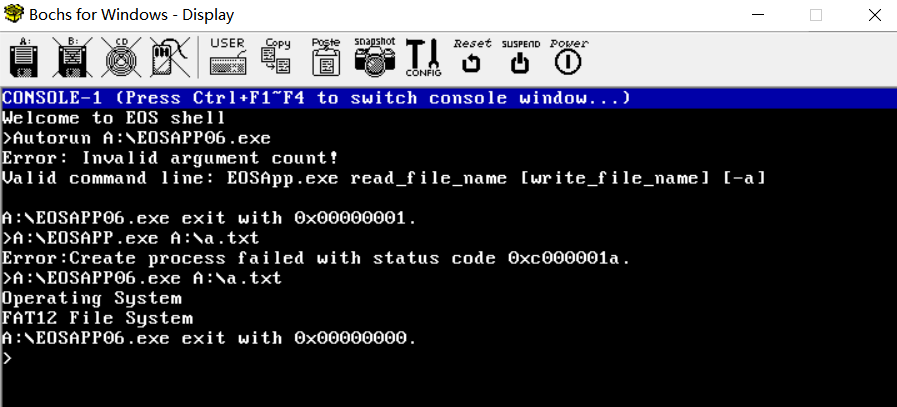


**2）EOS中FAT12文件系统读文件过程的跟踪**

（简要说明在本部分实验过程中完成的主要工作，包括对读文件的跟踪等，总结EOS中读文件的实现方法）

**2.1 FatReadFile函数**

1. 应用程序在执行时，会自动运行EOS应用程序EOSApp.exe，在控制台输入命令“A:\EOSApp.exe A:\a.txt”后按回车，EOSApp.exe 会读取 a.txt 文件，读文件时调用的API函数ReadFile最终会调用FatReadFile函数。



1. FatReadFile函数在 io/driver/fat12.c 文件中定义如下:

STATUS FatReadFile(

IN PVCB Vcb,

IN PFCB File,

IN ULONG Offset,

IN ULONG BytesToRead,

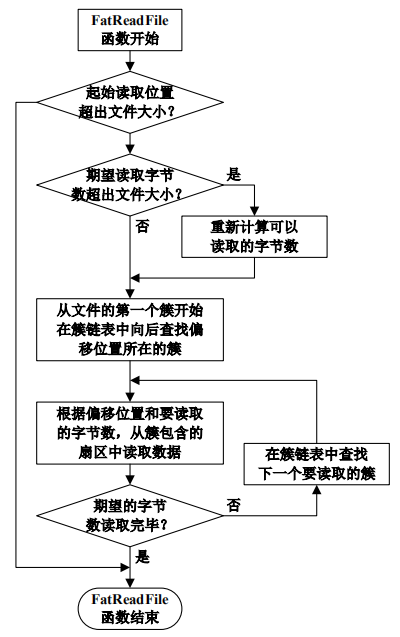
OUT PVOID Buffer,

OUT PULONG BytesRead

)

参数 Vcb 和 File 分别指向卷控制块和文件控制块，为读文件操作提供所需要的全部信息。参数 Offset 是读文件的起始位置（以字节为单位），在 FatRead 函数中可以看到是将文件对象的 CurrentByteOffset 域传给了此参数。文件对象中的 CurrentByteOffset 域即为文件指针，创建文件对象时初始化此域为 0，表示从文件的开始位置读取。在读文件的过程中会将已经读取的字节数增加到 CurrentByteOffset域中，从而保证文件指针始终指向下一次要读取的起始位置。增加文件指针的代码在 IopReadFileObject函数中。参数 BytesToRead、Buffer、BytesRead 分别与 ReadFile 中的参数 NumberOfBytesToRead 、Buffer、NumberOfBytesRead 对应。

FatReadFile函数的流程图如下图：



首先是要定位读取的起始位置所在的簇号，由于保存文件数据的簇是由 FAT表中的簇链串起来的，只能顺序的获取簇号，如果读取的起始位置不在文件的第一个簇中，就需要顺着簇链向后找到起始位置所在的簇。第二个问题是，在定位要读取的第一个簇后，就可以读取数据了，不过由于要读取的数据可能保存在多个连续的簇中，而且一个簇可以包括多个扇区（虽然这里使用的 FAT12 文件系统一个簇只包括一个扇区），所以需要使用一个双重循环来遍历多个簇，并且在簇中遍历多个扇区。最后一个问题是，只能通过调用函数 IopReadWriteSector 从单个扇区中读取数据，这就需要将要读取的数据以扇区为单位进行分块，并将簇号转换为对应的扇区号。

1. 调试FatReadFile函数，在“快速监视”对话框输入”\*Vcb”，参数Vcb提供的信息如下：

{

DiskDevice = 0x803fc578,

Bpb = {

BytesPerSector = 512,

SectorsPerCluster = 1 '\001',

ReservedSectors = 1,

Fats = 2 '\002',

RootEntries = 224,

Sectors = 2880,

Media = 240 '?,

SectorsPerFat = 9,

SectorsPerTrack = 18,

Heads = 2,

HiddenSectors = 0,

LargeSectors = 0

},

Fat = 0x803f7fc8,

FirstRootDirSector = 19,

RootDirSize = 7168,

FileListHead = {

Next = 0x803fab74,

Prev = 0x803faff4

},

FirstDataSector = 33,

NumberOfClusters = 2847

}

1. 在“快速监视”对话框输入“\*File”提供信息如下：

{

……

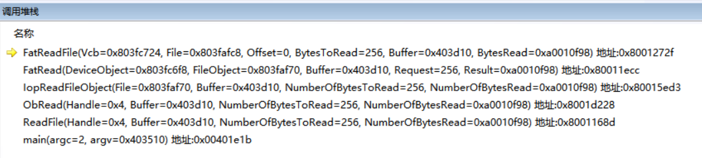
FirstCluster = 3,

FileSize = 35,

……

}

1. 在“调用堆栈”窗口可以看到FatReadFile函数的执行过程和之前的流程图一致。



1. 在本次调试过程中，读取文件的起始偏移位置（0）未超过文件的大小（35），且预期读取的字节数（256）大于文件的大小（35）所以实际可读取的字节数为35。偏移位置Offset为0所以开始读取的簇Cluster是文件的第一个簇。

Cluster = File->FirstCluster;

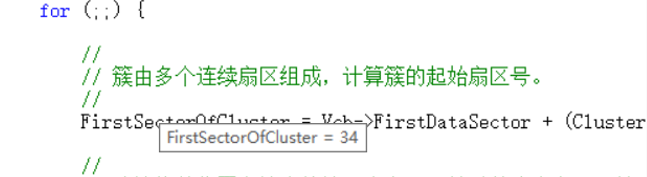
for (i = Offset / FatBytesPerCluster(&Vcb->Bpb); i > 0; i--) {

Cluster = FatGetFatEntryValue(Vcb, Cluster);

}

1. 计算簇的起始扇区号，由于文件第一个簇为3，所以计算结果FirstSectorOfCluster 为34。

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



1. 接下来使用双重循环读取扇区中的数据，外层循环是遍历文件簇链中的所有簇，内层循环是遍历一个簇中的所有扇区。由于该文件大小只有 35 个字节，都存储在第一个簇的第一个扇区中，另外，这里使用的 FAT12 文件系统每个簇只有一个扇区，所以并没有循环执行。

for (i = ((Offset + ReadCount) / Vcb->Bpb.BytesPerSector) % Vcb->Bpb.SectorsPerCluster;

i < Vcb->Bpb.SectorsPerCluster; i++ ) {

OffsetInSector = (Offset + ReadCount) % Vcb->Bpb.BytesPerSector;

// 计算需要在这个扇区内读取的字节数。

if (BytesToRead - ReadCount > Vcb->Bpb.BytesPerSector - OffsetInSector) {

BytesToReadInSector = Vcb->Bpb.BytesPerSector - OffsetInSector;

} else {

BytesToReadInSector = BytesToRead - ReadCount;

}

Status = IopReadWriteSector( Vcb->DiskDevice,

FirstSectorOfCluster + i,

OffsetInSector,

(PCHAR)Buffer + ReadCount,

BytesToReadInSector,

TRUE );

if (!EOS\_SUCCESS(Status)) {

return Status;

}

ReadCount += BytesToReadInSector;

if (ReadCount == BytesToRead) {

\*BytesRead = ReadCount;

return STATUS\_SUCCESS;

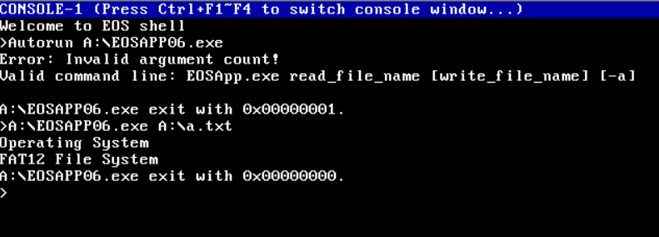
}

}

Cluster = FatGetFatEntryValue(Vcb, Cluster);

}

1. 激活虚拟机窗口查看执行结果：



**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）

3.1 实现方法简要描述

实现目标：

使 FatWriteFile 函数写入文件的数据可以跨越多个扇区的边界。

功能描述：

在文件指定的偏移位置开始写数据，如果偏移位置小于文件大小则覆盖原有内容，如果写范围超出文件大小则自动增加文件大小，如果文件大小增加后超过文件占用的磁盘空间大小则自动为文件分配新的簇，增加文件占用的磁盘空间。

实现方法：

根据起始写入的位置和写入数据的大小将要写入的数据分割为不跨越扇区边界的多块数据。当没有跨越扇区时，直接使用 IopReadWriteSector 函数写入文件，若写入数据需要跨扇区，即使用循环来处理写入数据跨越多个扇区边界的问题。每次循环时只将合适的数据写入当前簇，在后面的循环中将余下的数据写入簇链中后面的簇，直到所有数据写入完毕。每次向簇中写数据之前都需要判断是否需要分配新簇。最后，如果文件长度增加了则必须修改文件的长度，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。

3.2 源代码

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;

ULONG i;

ULONG WriteCount=0;

// 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;

//如果偏移量+要写的字节数超过了一个扇区的大小，说明要跨越这个要写的扇区，写入后面的一个或多个扇区中

if(OffsetInSector + BytesToWrite > Vcb->Bpb.BytesPerSector){

//先计算第一个扇区要写多少个字节，扇区大小减偏移量

ULONG BytesToWriteFirstTime = Vcb->Bpb.BytesPerSector - OffsetInSector;

//计算最后一次写入的扇区要写多少数据

ULONG BytesToWriteLastTime = (BytesToWrite - BytesToWriteFirstTime) % Vcb->Bpb.BytesPerSector;

Status = IopReadWriteSector( Vcb->DiskDevice,

FirstSectorOfCluster,

OffsetInSector,

(PCHAR)Buffer,

BytesToWriteFirstTime,

FALSE );

if (!EOS\_SUCCESS(Status))

return Status;

WriteCount += BytesToWriteFirstTime;

//数据去掉开始的部分和最后剩余的部分，中间的部分一定是扇区大小的整数倍，计算出这个倍数，就是要写几次中间的扇区

INT sharpTimes = (BytesToWrite - BytesToWriteFirstTime - BytesToWriteLastTime) / Vcb->Bpb.BytesPerSector;

INT j = 0;

for (j = 0; j<sharpTimes; ++j) {

//顺着簇链向后查找下一个簇的簇号，如果没有就先分配

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;

//写入当前扇区，因为写入完整的一个扇区的数据，所以偏移量为 0，写入字节数为一个扇区的大小

//写入数据缓冲区(PCHAR)Buffer的指针要移动到对应的位置，地址！即起始位置+已经写完的字节数

Status = IopReadWriteSector(Vcb->DiskDevice,

FirstSectorOfCluster,

0,

(PCHAR)(Buffer + WriteCount),

Vcb->Bpb.BytesPerSector,

FALSE);

if (!EOS\_SUCCESS(Status))

return Status;

WriteCount += Vcb->Bpb.BytesPerSector;

}

//如果最后还剩下一些不够一个扇区的数据，写入

if (BytesToWriteLastTime != 0) {

//同样先找下一个簇，如果没有就分配一个

PrevClusterNum = CurrentClusterNum;

CurrentClusterNum = FatGetFatEntryValue(Vcb, PrevClusterNum);

if (0 == CurrentClusterNum || CurrentClusterNum >= 0xFF8) {

FatAllocateOneCluster(Vcb, &NewClusterNum);

if (0 == File->FirstCluster)

File->FirstCluster = NewClusterNum;

else

FatSetFatEntryValue(Vcb, PrevClusterNum, NewClusterNum);

CurrentClusterNum = NewClusterNum;

}

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

Status = IopReadWriteSector(Vcb->DiskDevice,

FirstSectorOfCluster,

0,

(PCHAR)(Buffer + WriteCount),

BytesToWriteLastTime,

FALSE);

if (!EOS\_SUCCESS(Status))

return Status;

}

}

//如果数据不会跨越扇区就一次写入

else {

Status = IopReadWriteSector(Vcb->DiskDevice,

FirstSectorOfCluster,

OffsetInSector,

(PCHAR)Buffer,

BytesToWrite,

FALSE);

if (!EOS\_SUCCESS(Status))

return Status;

}

// 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

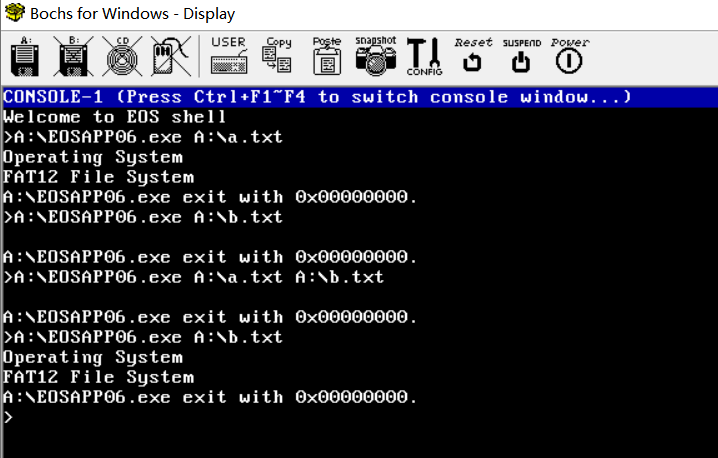
\*BytesWriten = BytesToWrite;

return STATUS\_SUCCESS;

}

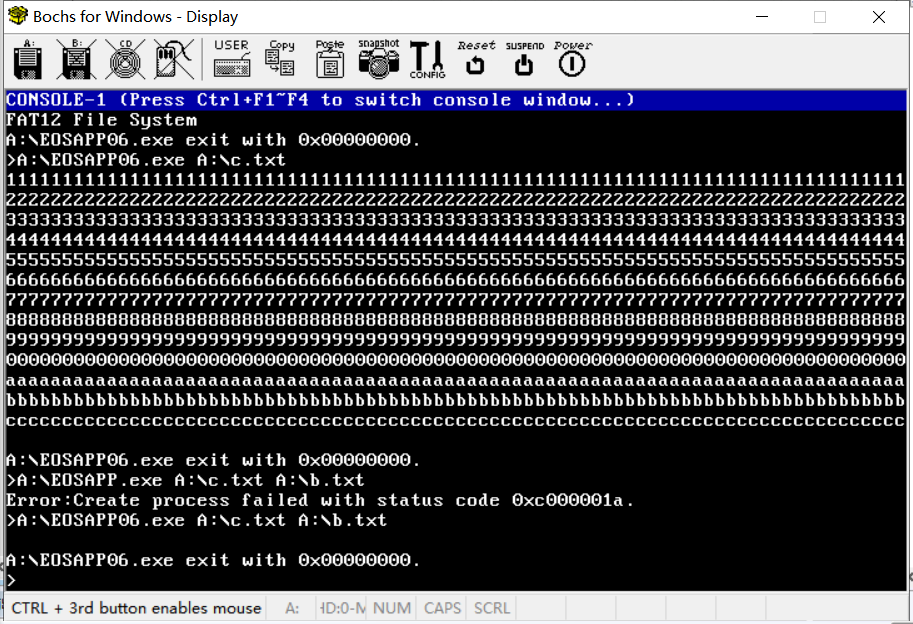
3.3 测试及结果

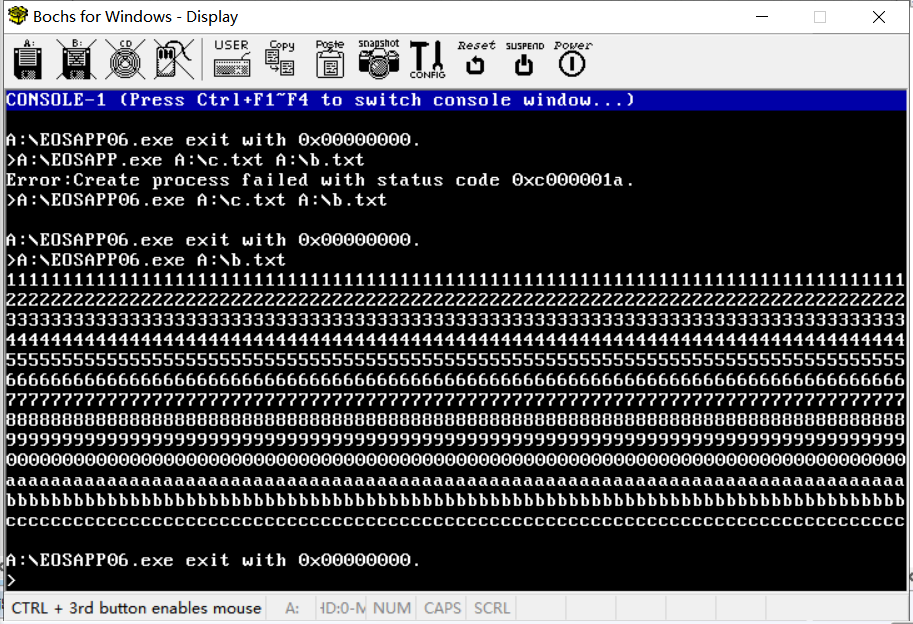
a）将a.txt 文件内容写入 b.txt, b.txt 一开始为空，写入后读出数据与 a.txt 相同



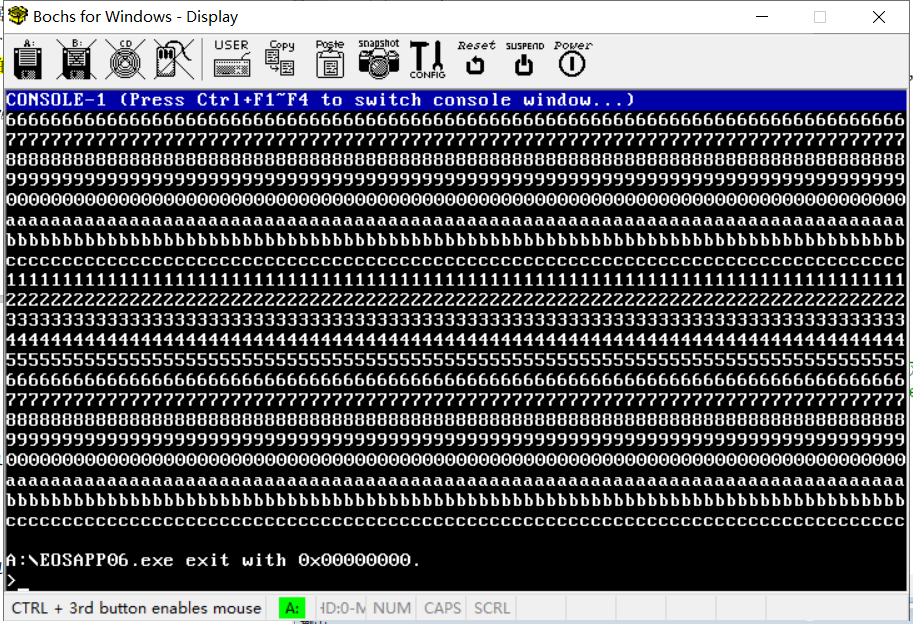
b）将 c.txt 文件内容写入 b.txt,写入后读出 b.txt 数据与 c.txt 相同(覆盖原来 b.txt 中内

容)文件c.txt大小为 1024B，为两个扇区的大小，因此这里跨了一个扇区边界。

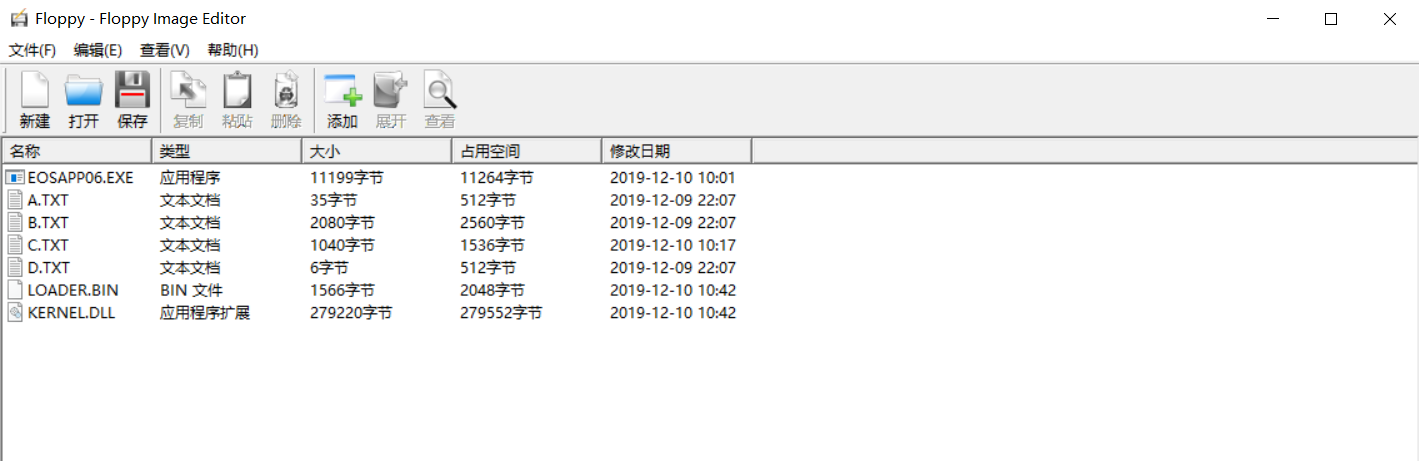




c）将c.txt文件内容增加写入b.txt文件，输出b.txt文件内容，为两份c.txt文件的内容



通过查看Floppy.img中b.txt的文件的大小为2080字节恰好为c.txt大小1040字节的两倍，从而证明结果正确。



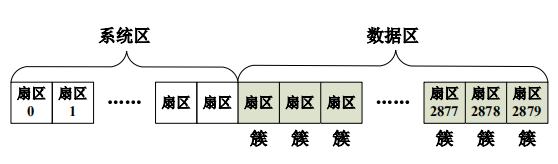
**结果分析：**

（对本实验所做工作及结果进行分析，包括EOS中FAT12文件系统实现方法的特点、不足及改进意见；结合EOS对文件系统实现相关问题提出自己的思考；分析写文件实现方法的有效性、不足和改进意见，如果同时采用了多种实现方法，则进行对比分析；其他需要说明的问题）

1、EOS中FAT12文件系统实现方法的特点及不足

文件系统不关心用于存储文件的各个数据块的物理位置，而总是将这些数据块抽象为与其大小相同的逻辑扇区，并将这些逻辑扇区组成了一个线性的、可随机访问的、从 0 开始计数的数组。EOS 使用的FAT12 文件系统中虽然 1.44M 软盘上的 2880 个物理扇区分布在不同的盘面和磁道上，但是 FAT12 文件系统只是将这些物理扇区抽象为编号从 0 到 2879 的逻辑扇区。当 FAT12 文件系统需要访问 9 号逻辑扇区时，就将编号 9 做为参数传入块设备层的 IopReadWriteSector 函数即可，软盘驱动程序负责将逻辑扇区编号映射到相应的物理扇区。

簇包括一组连续的扇区，文件系统将簇做为数据区中存储文件数据的基本单位，在EOS 使用的 FAT12 文件系统中，一个簇只包含一个扇区。其结构如图所示。



文件系统解析给出的路径，将文件相对于其所管辖设备（可以简单理解为磁盘上的某个分区）的扇区偏移计算出来，将这个偏移传递给磁盘驱动，磁盘驱动将这个偏移转换成相对于整个磁盘的偏移，然后根据读写指令向磁盘控制器发送信号，从而实现对文件的各种操作。

FAT12 文件系统特点：FAT12 住磁盘分为引导扇区、文件分配表、根目录区、数据区四个部分，其中引导扇区位于第一个扇区，完成了 FAT12 文件系统的初始化，并将其绑定到了软盘驱动器上。紧接着引导扇区的是两个完全相同的 FAT 表，每个 FAT 表占用 9 个扇区。它用于将数据区的磁盘空间分配给文件，将文件占用的簇连接成一个簇链，链头由目录项中的起始簇号确定。FAT 表之后是根目录区，根目录区中最多能装 224 个目录项，因此 FAT12 最多只能有 224 个文件/文件夹。根目录后是数据区，存放文件中具体内容。

FAT12 文件系统的不足：磁盘利用效率低，文件碎片严重,文件存储收到了限制，当文件被删除并且在同一位置被写入新数据，他们的片段通常是分散的，减慢了读写速度。

2、写文件实现方法的分析

FatWriteFile 函数中使用了 for 循环实现跨扇区写入。基本思想是数据去掉开始的部分和最后剩余的部分，中间的部分一定是扇区大小的整数倍，计算出这个倍数，就是要写几次中间的扇区，中间的倍数 sharpTimes 作为 for 循环的循环次数，实现中间扇区的写入，在这之前要先计算出文件写入的偏移位置与扇区大小之差，先写满上一个簇，接着写中间扇区，然后将文件大小与之前写入的数据长度相减，将剩下部分写入下一个簇。需要注意的是如果文件长度增加了则必须修改文件的长度。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。经验证该方法可行，能够正确实现跨扇区写入数据。

3、思考

（1）结合 FAT12 文件系统，说明“文件大小”和“文件占用磁盘空间大小”的区别，并举例说明文件的这两个属性值变化的方式有什么不同。

答：文件的大小其实就是文件内容实际具有的字节数，它以 Byte 为衡量单位，只要文件 内容和格式不发生变化，文件大小就不会发生变化。但文件在磁盘上的所占空间却不是 以 Byte 为衡量单位的，它最小的计量单位是“簇(Cluster)”。所以，当我们存储非常多的小文件的时候，其所占用的空间就越大，但很可能其实际的文件大小却很小，也就是 这个道理。同理，一个文件越大，其占用的空间也就越小。

（2） EOS 应用程序在读写文件时，缓冲区大小设置为 512 的倍数比较合适，说明原因。

答：因为 FAT12 系统一个扇区的大小为 512B,即一个簇大小为 512B，而簇又是文件数据区存储空间的最小单位，因此将缓冲区大小设置为 512 的倍数可以方便读写一个或多个 簇。