·四川大学计算机学院学院

实 验 报 告

学号：2022141460176 姓名：杨一舟 专业：计算机科学与技术 第 13 周

课程名称	操作系统课程设计	实验课时	2
实验项目	Linux 文件系统管理	实验时间	2024.5.20
实验目的	1、熟悉 Linux 文件管理的常见命令 2、理解 Linux 文件管理机制 3、掌握位示图在文件系统中如何管理 4、熟悉目录的创建、维护和文件操作		
实验环境	Ubuntu 操作系统		
实验内容	**实验内容** **文件结构：** 位示图（Bitmap）：在外部存储器上创建位示图来记录文件存储器的使用情况，其中每个位对应一个物理块，用 0 和 1 表示空闲和占用。 Inode：每个文件或目录都有一个索引节点（Inode），它保存了文件系统中对象的元信息数据，如文件类型、大小和分配的磁盘块等。 **操作内容：** 模拟文件创建和删除：模拟文件系统的创建和删除操作，包括分配 Inode、更新目录、分配磁盘块等。 文件读写：文件系统的读写操作，如何组织管理磁盘上的文件。		

实验步骤

（以下代码在 Dev-C++中展示）

一、补全代码

1. 补充 BitMap 中未实现的函数；

```
void markBitAt(int vBitPosition, SBitMap& vioBitMap)
{
    vioBitMap.pMapData[vBitPosition / g_NumBitsInWord] |= 1 << (vBitPosition % g_NumBitsInWord);
}

void clearBitAt(int vBitPosition, SBitMap& vioBitMap)
{
    //此函数将位示图中指定位置的位清零。
    vioBitMap.pMapData[vBitPosition / g_NumBitsInWord] &= ~(1 << (vBitPosition % g_NumBitsInWord));
}

bool isAvailableBitAt(int vBitPosition, const SBitMap& vBitMap)
{
    //此函数检查位示图中指定位置的位是否为0（即是否可用）
    return (vBitPosition < vBitMap.NumBits) &&
        ((vBitMap.pMapData[vBitPosition / g_NumBitsInWord] & (1 << (vBitPosition % g_NumBitsInWord))) == 0);
}
```

```
int countClearBits(const SBitMap& vBitMap)
{
    //此函数统计位示图中所有清零（未占用）的位数。
    int count = 0;
    int NumWords = (int)ceil(vBitMap.NumBits / (double)g_NumBitsInWord);

    for (int i = 0; i < NumWords; ++i)
    {
        int word = vBitMap.pMapData[i];
        for (int j = 0; j < g_NumBitsInWord; ++j)
        {
            if ((word & (1 << j)) == 0)
            {
                ++count;
            }
        }
    }

    return count;
}
```

```
int findAndSetAvailableBit(SBitMap& voBitMap)
{
    //此函数找到并设置（置为1）位示图中第一个可用的位，如果找不到则返回-1。
    int NumWords = (int)ceil(voBitMap.NumBits / (double)g_NumBitsInWord);
    for (int i = 0; i < NumWords; ++i)
    {
        if (voBitMap.pMapData[i] != ~0)  // ~0表示所有位都是1
        {
            for (int j = 0; j < g_NumBitsInWord; ++j)
            {
                int bitMask = 1 << j;
                if ((voBitMap.pMapData[i] & bitMask) == 0)
                {
                    voBitMap.pMapData[i] |= bitMask;
                    return i * g_NumBitsInWord + j;
                }
            }
        }
    }

    return -1;
}
```

2. 补充通过 Inode 回收磁盘块的函数；

```
void deallocateDisk(const SInode& vInode, SBitMap& vioCBitMap)
{
    //回收磁盘块
    for (int i = 0; i < vInode.NumBlocks; ++i)
        {
            if (vInode.BlockNums[i] != -1) // 确保BlockNums数组中的有效块号
            {
                clearBitAt(vInode.BlockNums[i], vioBlockBitMap); // 将对应的块号在DataBlockBitMap中标记为未占用
            }
        }
}
```

3. 补充 Directory.cpp 中的函数；

```
int findFileIndex(const char* vFileName, const SDirectory& vDirectory)
{
    // 遍历目录中的文件
    for (int i = 0; i < g_MaxNumFiles; ++i)
    {
        // 如果当前目录项正在使用，并且文件名匹配，则返回索引
        if (vDirectory.FileSet[i].IsInUse && strcmp(vFileName, vDirectory.FileSet[i].FileName) == 0)
        {
            return i;
        }
    }
    // 如果没有找到匹配的文件名，返回-1
    return -1;
}

bool addFile2Directory(const char* vFileName, short vInodeNum, SDirectory& voDirectory)
{
    // 检查目录是否已满
    for (int i = 0; i < g_MaxNumFiles; ++i)
    {
        // 如果找到一个未使用的目录项，则添加新文件
        if (!voDirectory.FileSet[i].IsInUse)
        {
            voDirectory.FileSet[i].IsInUse = true;
            voDirectory.FileSet[i].InodeNum = vInodeNum;
            strncpy(voDirectory.FileSet[i].FileName, vFileName, g_MaxFileNameLen);
            voDirectory.FileSet[i].FileName[g_MaxFileNameLen] = '\0'; // 确保字符串以null字符结尾
            return true;
        }
    }
    // 如果目录已满，返回false
    return false;
}
```

4. 补充删除文件的函数；

```
bool removeFile(const char* vFileName, int vDirInodeNum)
{
    // 加载指定目录的目录结构
    SDirectory TempDirectory = loadDirectoryFromDisk(vDirInodeNum);

    // 使用findFileIndex找到要删除文件的目录项索引
    int FileIndex = findFileIndex(vFileName, TempDirectory);
    if (FileIndex == -1) return false; // 如果文件不存在，返回false

    // 获取对应的Inode编号
    int InodeNum = TempDirectory.FileSet[FileIndex].InodeNum;

    // 加载Inode信息
    SInode FileInode = loadInodeFromDisk(InodeNum);

    // 如果文件是目录，检查并更新Inode链接数
    if (FileInode.FileType == 'd')
    {
        if (FileInode.NumLinks > 1) {
            FileInode.NumLinks--; // 如果有其他链接，减少链接数
            saveInode2Disk(FileInode, InodeNum); // 保存更新后的Inode信息
        } else {
            // 如果没有其他链接，回收磁盘块
            SBitMap DataBlockBitMap;
            createEmptyBitMap(DataBlockBitMap, g_NumBlocks);
            memcpy(DataBlockBitMap.pMapData, g_Disk, g_BlockBitMapSize); // 从磁盘读取数据块位图
            deallocateDisk(FileInode, DataBlockBitMap); // 释放数据块
            memcpy(g_Disk, DataBlockBitMap.pMapData, g_BlockBitMapSize); // 更新磁盘上的数据块位图
            delete DataBlockBitMap.pMapData;
        }
```

```
    } else {
        // 如果文件是普通文件，直接回收磁盘块
        SBitMap DataBlockBitMap;
        createEmptyBitMap(DataBlockBitMap, g_NumBlocks);
        memcpy(DataBlockBitMap.pMapData, g_Disk, g_BlockBitMapSize); // 从磁盘读取数据块位图
        deallocateDisk(FileInode, DataBlockBitMap); // 释放数据块
        memcpy(g_Disk, DataBlockBitMap.pMapData, g_BlockBitMapSize); // 更新磁盘上的数据块位图
        delete DataBlockBitMap.pMapData;
    }

    // 删除目录项
    TempDirectory.FileSet[FileIndex].InodeNum = -1;
    TempDirectory.FileSet[FileIndex].IsInUse = false;
    memset(TempDirectory.FileSet[FileIndex].FileName, 0, g_MaxFileNameLen + 1);

    // 更新Inode位图
    SBitMap InodeBitMap;
    createEmptyBitMap(InodeBitMap, g_NumInodes);
    memcpy(InodeBitMap.pMapData, g_Disk + g_BlockBitMapSize, g_InodeBitMapSize); // 从磁盘读取Inode位图
    clearBitAt(InodeNum, InodeBitMap); // 清除对应Inode的位
    memcpy(g_Disk + g_BlockBitMapSize, InodeBitMap.pMapData, g_InodeBitMapSize); // 更新磁盘上的Inode位图
    delete InodeBitMap.pMapData;

    // 将更新后的目录写回磁盘
    saveDirectory2Disk(vDirInodeNum, TempDirectory);

    return true;
}
```

此函数确保了文件系统在删除文件时正确更新目录结构、释放相应的 Inode 和数据块，并将所有更改同步到磁盘。

二、测试验证

1. 编译源代码

在 Linux 环境下使用 make 命令（Makefile 已经配置好）来编译代码：

根据现有的 Makefile 配置文件，上述命令将编译 BitMap.cpp、Directory.cpp、FileSystem.cpp 和 main.cpp，生成可执行文件 FileSystem.out。

```
mountain@Lumous:~/OS_projects/文件创建和删除$ make
g++ -c BitMap.cpp
g++ -c Directory.cpp
g++ -c FileSystem.cpp
g++ -c main.cpp
g++ -o FileSystem.out BitMap.o Directory.o FileSystem.o main.o
```

2. 执行可执行文件

在终端中，运行编译后的可执行文件：

./FileSystem.out

```
mountain@Lumous:~/OS_projects/文件创建和删除$ ./FileSystem.out
Creating file test.txt(type:f) size 822 at /
Creating file test1.txt(type:f) size 822 at /
```

文件创建命令执行：

Creating file test.txt(type:f) size 822 at /

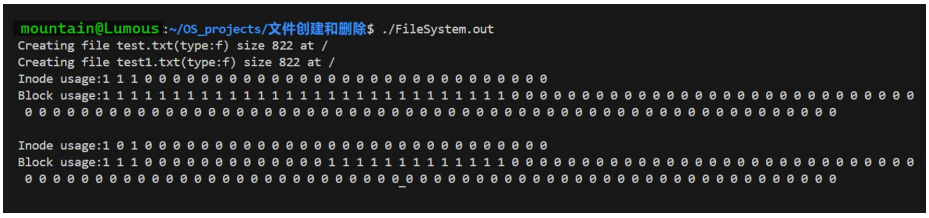
Creating file test1.txt(type:f) size 822 at /

这表明程序成功创建了两个普通文件（由 type:f 指示）：test.txt 和 test1.txt，大小为 822 字节，并都存储在根目录下。

Inode 使用情况：
初始情况：1 1 1 0 ... 表示有三个 Inode 被分配出去
（即两个文件和一个根目录），其余 Inode 为空闲。
删除文件后：1 0 1 0 ... 表示有一个 Inode 被释放（置为 0），
表明 test.txt 文件被删除，但 test1.txt 的 Inode 仍然被占用。

磁盘块使用情况
初始磁盘块使用情况：
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 ... 表示有多个块被分配出去，具体数量取决于文件大小和文件系统的块大小设置。

删除文件后的磁盘块使用情况：
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 ... 表示删除文件 test.txt 后，所占用的磁盘块被释放。



实验结果	文件创建：程序能够成功创建文件，并分配 Inode 和磁盘块。 Inode 管理：Inode 分配和释放正确，删除文件时能够释放 Inode。 磁盘块管理：删除文件后，文件所占用的磁盘块被释放。
小　结	本次实验让我深刻理解了文件系统的基本原理，特别是位示图和 Inode 在磁盘空间管理和文件识别中的作用。通过实践操作，我熟悉了目录和文件的管理方法，并通过编程实践加深了对文件系统工作机制的理解。这次实验不仅丰富了知识，也提高了我的实践能力。
指导老师评　议	成绩评定：　　指导教师签名：