

操作系统实验四：FAT 文件系统的实现

——任务一实验文档

姓名：李昱祁

学号：PB18071496

实验任务：

我完成的实验内容为任务一的第一部分，要求如下：

支持读操作的 FAT16 文件系统

- 补全 `simple_fat16.c` 代码文件中的五个 TODO
- 利用 `bench.h` 文件中给出的测试函数进行测试
- 进行 FUSE 功能测试
- 回答相关问题

代码补全：

首先介绍补全的 5 个 TODO

1. `path_split`

```
char **path_split(char *pathInput, int *pathDepth_ret)
{
    int i = 0, j = 0, k = 0;
    int pathDepth = 0, pathLenth = 0;
    pathLenth = strlen(pathInput);
```

```

for (i = 0; i < pathLenth; i++)
{
    if(pathInput[i] == '/')
        pathDepth++;
}

char **paths = malloc(pathDepth * sizeof(char *));

for(i = 0; i < pathDepth; i++)
{
    paths[i] = malloc(11 * sizeof(char));
}

int isExtension = 0; //用于记录是否为扩展名

for (i = 0, j = 0; i < pathLenth; i++)
{
    if (pathInput[i] == '/')
    {
        strcpy(paths[j], " ");
        k = 0;
        isExtension = 0;
        if (pathInput[i + 1] == '.' && pathInput[i + 2] == '.' &&
(pathInput[i + 3] == '/' || i + 3 >= pathLenth))
        {
            // "../.../"
            paths[j][0] = '.';
            paths[j][1] = '.';
            i += 2;
        }
    }
}

```

```

        else if (pathInput[i + 1] == '.' && (pathInput[i + 2] == '/' || i
+ 2 >= pathLenth))
        {
            paths[j][0] = '.';
            i += 1;
        }
        j++;
        continue;
    }

    if (pathInput[i] == '.')
    {
        k = 8;
        isExtension = 1;
        continue;
    }
    if ((k < 11 && isExtension == 1) || (k < 8 && isExtension == 0))
    {
        if(pathInput[i] >= 'a' && pathInput[i] <= 'z')
            paths[j - 1][k] = pathInput[i] - 'a' + 'A';
        else
            paths[j - 1][k] = pathInput[i];
        k++;
    }

}

*pathDepth_ret = pathDepth;
return paths;
}

```

此函数主要功能为：将一个表示绝对路径名的字符串进行分割处理，将分割后的每段放入一个二维数组中。

依据要求，先根据 ‘/’ 的个数统计出路径的深度；再把每两个 ‘/’ 之间的内容依次拷贝至二维数组 paths 中，具体做法是：每读到一个 ‘/’，就完成对这个 ‘/’ 到下一个 ‘/’（或者是最终的文件/目录）之间内容的拷贝。需要注意的是：

‘.’ 符号既可能表示文件名与扩展名之间的分隔，也可能表示 ‘/.’ 或者 ‘/..’，需要进行单独判断；在拷贝英文字母时，如果是小写则需要转换成大写。

2. pre_init_fat16

TODO 部分补充如下：

```
sector_read(fd, 0, buffer);

memcpy(&fat16_ins->Bpb.BS_jmpBoot, buffer, 3);
memcpy(&fat16_ins->Bpb.BS_OEMName, buffer + 3, 8);
memcpy(&fat16_ins->Bpb.BPB_BytsPerSec, buffer + 11, 2);
memcpy(&fat16_ins->Bpb.BPB_SecPerClus, buffer + 13, 1);
memcpy(&fat16_ins->Bpb.BPB_RsvdSecCnt, buffer + 14, 2);
memcpy(&fat16_ins->Bpb.BPB_NumFATS, buffer + 16, 1);
memcpy(&fat16_ins->Bpb.BPB_RootEntCnt, buffer + 17, 2);
memcpy(&fat16_ins->Bpb.BPB_TotSec16, buffer + 19, 2);
memcpy(&fat16_ins->Bpb.BPB_Media, buffer + 21, 1);
memcpy(&fat16_ins->Bpb.BPB_FATSz16, buffer + 22, 2);
memcpy(&fat16_ins->Bpb.BPB_SecPerTrk, buffer + 24, 2);
memcpy(&fat16_ins->Bpb.BPB_NumHeads, buffer + 26, 2);
memcpy(&fat16_ins->Bpb.BPB_HiddSec, buffer + 28, 4);
memcpy(&fat16_ins->Bpb.BPB_TotSec32, buffer + 32, 4);
memcpy(&fat16_ins->Bpb.BS_DrvNum, buffer + 36, 1);
memcpy(&fat16_ins->Bpb.BS_Reserved1, buffer + 37, 1);
memcpy(&fat16_ins->Bpb.BS_BootSig, buffer + 38, 1);
memcpy(&fat16_ins->Bpb.BS_VollID, buffer + 39, 4);
```

```

memcpy(fat16_ins->Bpb.BS_VollLab, buffer + 43, 11);
memcpy(fat16_ins->Bpb.BS_FilSysType, buffer + 54, 8);
memcpy(fat16_ins->Bpb.Reserved2, buffer + 62, 448);
memcpy(&fat16_ins->Bpb.Signature_word, buffer + 510, 2);

/* 根目录偏移 = FAT1 的偏移 + 所有 FAT 表的长度 */
fat16_ins->FirstRootDirSecNum = fat16_ins->Bpb.BPB_RsvdSecCnt +
fat16_ins->Bpb.BPB_NumFATS * fat16_ins->Bpb.BPB_FATSz16;

/* 数据区偏移：在根目录的基础上进行计算 */
fat16_ins->FirstDataSector = fat16_ins->FirstRootDirSecNum +
fat16_ins->Bpb.BPB_RootEntCnt * 32 / fat16_ins->Bpb.BPB_BytsPerSec;

return fat16_ins;
}

```

先把 DBR 扇区的数据读至 buffer 中，之后用其中 BPB 参数块来补充 fat16 结构体中 BPB_BS 结构体的各项参数；之后，就可以用 BPB 块中的参数来计算

fat16_ins->FirstRootDirSecNum 与
fat16_ins->FirstDataSector:

fat16_ins->FirstRootDirSecNum 为根目录第一个扇区的偏移，用“保留区扇区数”+“FAT 表数目”*“每个 FAT 表的扇区数”即可

fat16_ins->FirstDataSector 为数据区第一个扇区的偏移，用之前算出的根目录第一个扇区偏移，加上根目录扇区数即可。根目录扇区数计算如下：

目录条目数 * 每个目录条目字节数 / 每个扇区字节数

3. fat_entry_by_cluster

```
WORD fat_entry_by_cluster(FAT16 *fat16_ins, WORD ClusterN)
```

```

{
    BYTE sector_buffer[BYTES_PER_SECTOR];

    WORD offset_by_byte, SEC_offset;

    WORD Sec_Num, FatClusEntryVal;

    offset_by_byte = 2 * (ClusterN - 2) + 4; // 计算要找的 FAT 项在 FAT 表中偏移了
多少字节

    Sec_Num = offset_by_byte / fat16_ins->Bpb.BPB_BytsPerSec;
    SEC_offset = offset_by_byte % fat16_ins->Bpb.BPB_BytsPerSec;

    sector_read(fat16_ins->fd, fat16_ins->Bpb.BPB_RsvdSecCnt + Sec_Num,
sector_buffer);

    FatClusEntryVal = *((WORD*)&sector_buffer[SEC_offset]);

    return FatClusEntryVal;
}

```

首先根据簇号，计算出要找的簇的信息在 FAT 表中**偏移的字节数**：其中减 2 表示根目录后第一个簇编号为 2；乘 2 表示在 FAT 表中每个簇用 2 个字节来记录相关信息；加 4 表示 FAT 表以“F8 FF FF FF”开头占用了 4 个字节的空間。

之后根据这个偏移量，我们可以算出它在 FAT 表中的第几个扇区，以及在扇区中的**偏移量**。再根据保留区中扇区数目，即可读出要找的 FAT 项所在的扇区至 sector_buffer，再根据该项在扇区中的偏移即可读出该项。

注意 sector_buffer 为字节数组，而一个 FAT 表项要占用 2 字节。可以将指针类型转换为 WORD* 来从该地址读取 2 字节内容。

4. find_subdir

```

int find_subdir(FAT16 *fat16_ins, DIR_ENTRY *Dir, char **paths, int
pathDepth, int curDepth)
{
    int i, j;

    int DirSecCnt = 1; /* 用于统计已读取的扇区数 */
    BYTE buffer[BYTES_PER_SECTOR];

    WORD ClusterN, FatClusEntryVal, FirstSectorofCluster;

    int Dir_per_sec = fat16_ins->Bpb.BPB_BytsPerSec / BYTES_PER_DIR;
    first_sector_by_cluster(fat16_ins, Dir->DIR_FstClusL0, &FatClusEntryVal,
&FirstSectorofCluster, buffer);

    i = 0;
    while(true)
    {
        if(i % (Dir_per_sec * fat16_ins->Bpb.BPB_SecPerClus) == 0 && i != 0)
        {
            DirSecCnt = 1;
            ClusterN = FatClusEntryVal;
            first_sector_by_cluster(fat16_ins, ClusterN, &FatClusEntryVal,
&FirstSectorofCluster, buffer);
        }

        else if(i % Dir_per_sec == 0 && i != 0 )
        {
            DirSecCnt++;
            sector_read(fat16_ins->fd, FirstSectorofCluster + (DirSecCnt -
1) % fat16_ins->Bpb.BPB_SecPerClus, buffer);
        }

        if(FatClusEntryVal == 0x00)

```

```

{
    break;
}

memcpy(Dir, buffer + i % Dir_per_sec * BYTES_PER_DIR, BYTES_PER_DIR);

if(strncmp(paths[curDepth], Dir->DIR_Name, 11) == 0)
{
    if(curDepth == pathDepth - 1)
    {
        return 0;
    }
    else
    {
        return(find_subdir(fat16_ins, Dir, paths, pathDepth, curDepth +
1));
    }
}

i++;
}

return 1;
}

```

首先在循环开始前，先计算出每个扇区的目录条目个数，以及将当前目录的起始扇区读入 buffer 中。

在循环中，变量 i 含义为我们读的目录条目的序号。

先判断是否需要更新 buffer：如果需要跨越簇当然要查 FAT 表更新为下一个簇的起始扇区；否则，若仅仅是读完了一个扇区则直接线性读取下一个扇区；若更新完扇区后发现，当前

FAT 表项中的值已经为 0x00，说明函数入口处给出的当前目录下没有此文件/目录，查找失败，break 跳出循环，返回 1

之后将当前 i 对应的目录条目填充到 Dir 当中；若此目录条目记录的文件名/目录名与要找的下一级相同，则进行如下判断：

1. 若已经找到了最后一级路径，则返回 0
2. 否则，递归调用 find_subdir，此时的 Dir 直接可以作为下一级函数调用入口的 Dir，只用将 curDepth 加 1 表示我们已经又找到了一级

5. read_path

```
int read_path(FAT16 *fat16_ins, const char *path, size_t size, off_t offset,
char *buffer)
{
    long long i, j;
    int malloc_size = (size + offset) > BYTES_PER_SECTOR ? (size + offset) :
BYTES_PER_SECTOR;
    BYTE *sector_buffer = malloc(malloc_size * sizeof(BYTE));

    /* 文件对应目录项，簇号，簇对应FAT表项的内容，簇的第一个扇区号 */
    DIR_ENTRY Dir;
    WORD ClusterN, FatClusEntryVal, FirstSectorofCluster;

    if(find_root(fat16_ins, &Dir, path) == 1 || Dir.DIR_FileSize <= offset)
    {
        return 0;
    }

    ClusterN = Dir.DIR_FstClusL0;
    first_sector_by_cluster(fat16_ins, ClusterN, &FatClusEntryVal,
&FirstSectorofCluster, buffer);
```

```

for (i = 0, j = 0; i < size + offset; i += BYTES_PER_SECTOR, j++)
{
    sector_read(fat16_ins->fd, FirstSectorofCluster + j, sector_buffer +
i);

    if ((j + 1) % fat16_ins->Bpb.BPB_SecPerClus == 0)
    {
        ClusterN = FatClusEntryVal;

        first_sector_by_cluster(fat16_ins, ClusterN, &FatClusEntryVal,
&FirstSectorofCluster, buffer);

        j = -1;
    }
}

memcpy(buffer, sector_buffer + offset, size);
free(sector_buffer);

return size;
}

```

该函数的设计比较简单：

首先用已经设计完乘的 find_root 函数来将 path 对应的文件信息读至 Dir 结构体中，之后我们便可以通过 Dir 来使用这些信息；

若 find_root 失败，或 offset < Dir. DIR_FileSize，则直接返回 0；

之后开始一个扇区一个扇区读入，循环控制变量 i 记录读入 buffer 的总字节数，每读入一个扇区 i 的值就会增加 BYTES_PER_SECTOR，即 512 字节；变量 j 记录在该簇中读的扇区数，当读完一簇后，通过 first_sector_by_cluster 函数来找到下一个扇区的起始扇区号，继续读入

当读完后，将 sector_buffer 中 offset 以后的内容拷贝至 buffer，返回读入的大小 size

测试结果：

测试一：

```
parallels@parallels-vm: ~/lab4-code
parallels@parallels-vm:~$ cd lab4-code
parallels@parallels-vm:~/lab4-code$ make clean
rm -f simple_fat16 *.o
parallels@parallels-vm:~/lab4-code$ make
gcc -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse -g -c -o simple_fat16.o simple_fat16.c
gcc -g -o simple_fat16 simple_fat16.o -lfuse -pthread
parallels@parallels-vm:~/lab4-code$ ./simple_fat16 --test
-----
running test
-----
#1 running test_path_split
test case 1: OK
test case 2: OK
test case 3: OK
success in test_path_split

#2 running test_pre_init_fat16
success in test_pre_init_fat16

#3 running test_fat_entry_by_cluster
test case 1: OK
test case 2: OK
test case 3: OK
success in test_fat_entry_by_cluster

#4 running test_find_subdir
test case 1: OK
test case 2: OK
test case 3: OK
success in test_find_subdir

parallels@parallels-vm:~/lab4-code$
```

Make 编译后，运行 main 函数中 test 部分的内容，成功通过了测试集

测试二：

以 fat16.img 作为磁盘镜像文件：

```
char *FAT_FILE_NAME = "fat16.img";
```

在这里，我还对 simple_fat16.c 文件中的 fat16_readdir 函数做了相应的修改，使得根目录下已经被删除的文件不被显示

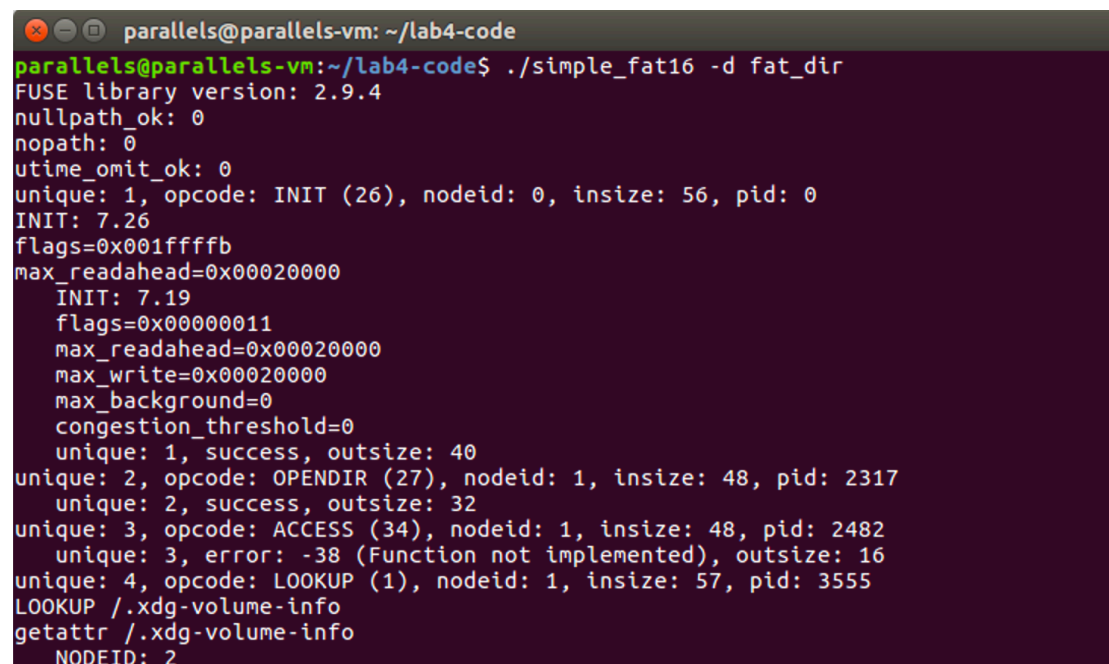
```

/* If we find a file or a directory, fill it into the pathInput */
    if ((Root.DIR_Attr == ATTR_ARCHIVE || Root.DIR_Attr ==
ATTR_DIRECTORY) && Root.DIR_Name[0] != 0xe5)
    {
        const char *filename = (const char
*)path_decode(Root.DIR_Name);
        filler(pathInput, filename, NULL, 0);
    }

```

即还要对 Root 条目中文件名的第一位（偏移为 0x0）的 1 个字节进行判断，如果其值为 0xe5 则表示它已经被删除，这一点在课堂教学中也有涉及到。

之后运行如下：

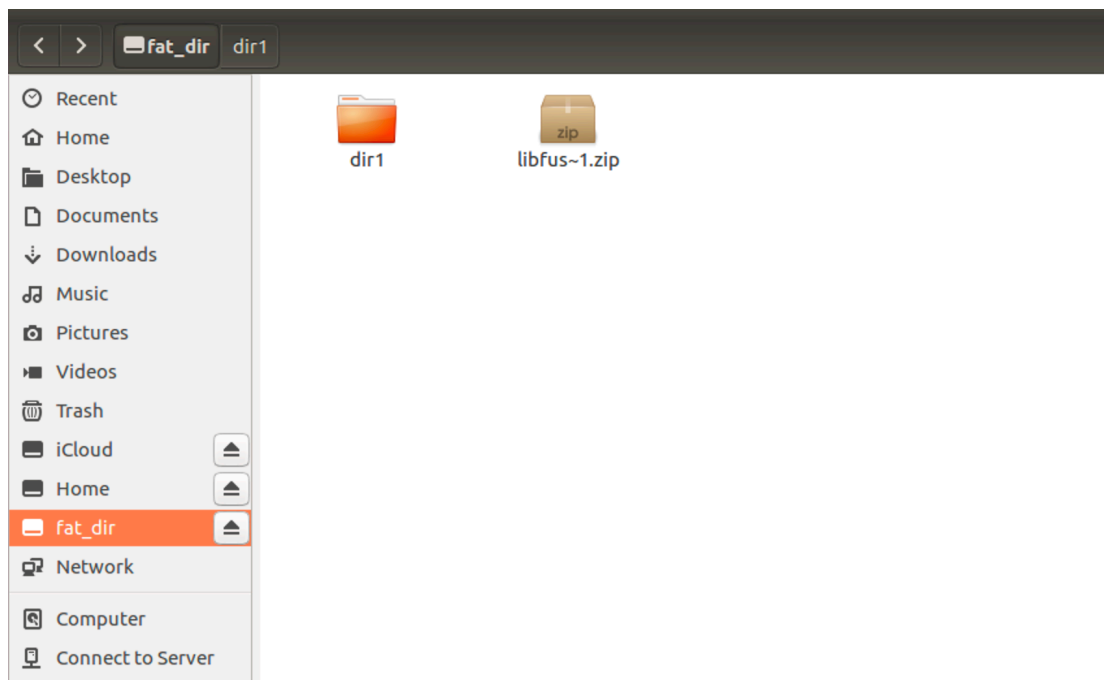


```

parallels@parallels-vm: ~/lab4-code
parallels@parallels-vm:~/lab4-code$ ./simple_fat16 -d fat_dir
FUSE library version: 2.9.4
nullpath_ok: 0
nopath: 0
utime_omit_ok: 0
unique: 1, opcode: INIT (26), nodeid: 0, insize: 56, pid: 0
INIT: 7.26
flags=0x001ffffb
max_readahead=0x00020000
    INIT: 7.19
    flags=0x00000011
    max_readahead=0x00020000
    max_write=0x00020000
    max_background=0
    congestion_threshold=0
    unique: 1, success, outsize: 40
unique: 2, opcode: OPENDIR (27), nodeid: 1, insize: 48, pid: 2317
    unique: 2, success, outsize: 32
unique: 3, opcode: ACCESS (34), nodeid: 1, insize: 48, pid: 2482
    unique: 3, error: -38 (Function not implemented), outsize: 16
unique: 4, opcode: LOOKUP (1), nodeid: 1, insize: 57, pid: 3555
LOOKUP /.xdg-volume-info
getattr /.xdg-volume-info
    NODEID: 2

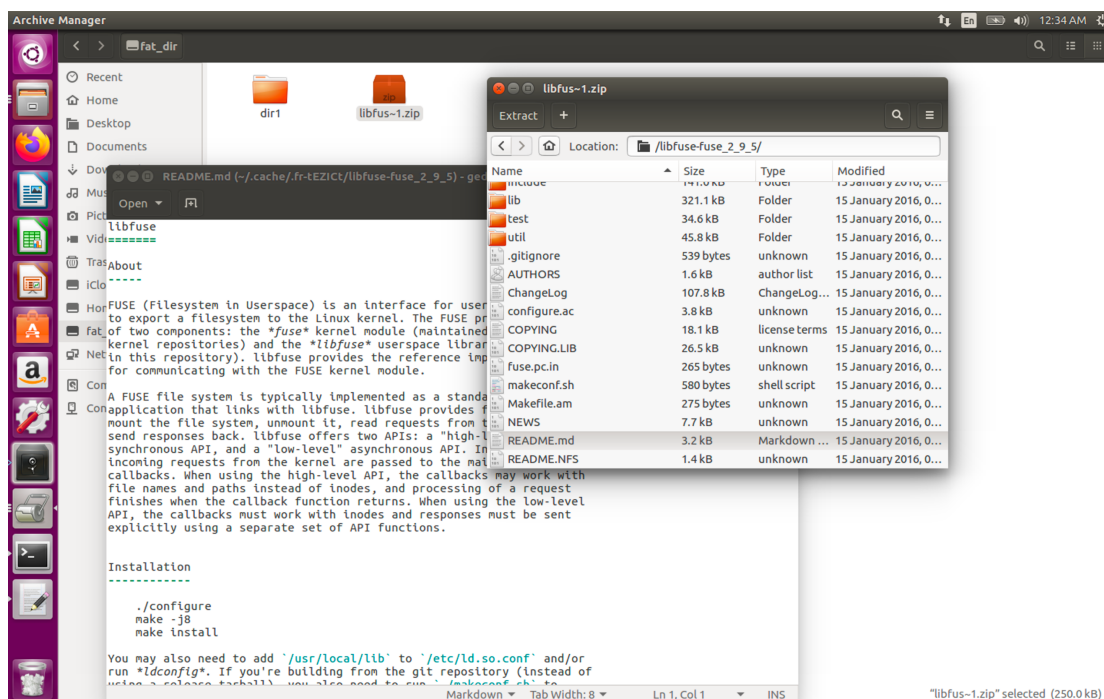
```

可以看到：

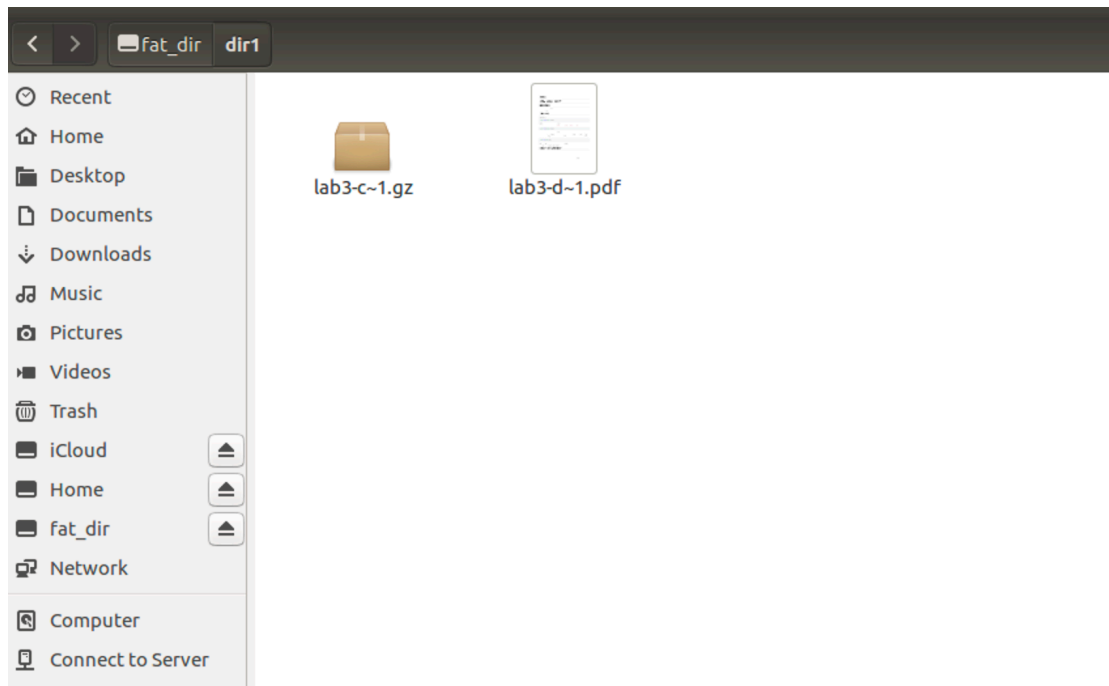


此时，无效编码文件就不会再显示了。

压缩包中的文件也都能正常打开：



dir1 目录下的内容为：



满足设计要求，通过了测试二

回答问题：

1. 简要描述代码定义的结构体 `fat16_oper` 中每个元素所对应的操作和操作执行的流程

答：

a. `fat16_init`:

主要操作为调用 `fuse_get_context` 函数，来构建 `context` 指针所指向的结构体。之后，返回所指结构体的相关数据。

b. `fat16_destory`:

直接 `free` 释放相应的指针，实际上也只是对 `free` 函数进行了封装

c. `fat16_getattr`:

主要功能为把文件相应的属性拷贝到 `stbuf` 所指向的结构体中

具体来说，部分属性可以直接进行拷贝，如文件大小等；一些属性通过函数调用来获取，比如其所属的用户、组等信息；其余一些如文件读/写/运行权限等，需要进行相应的修改。

d. `fat16_readdir`:

主要功能为在 FAT16 文件系统，寻找接口处 `path` 参数表示的目录，把该目录下所有的文件、目录路径信息记录在 `pathInput` 中，提供给 `fuse` 使用

分两种情况进行：如果 `path` 表示根目录，则直接搜索根目录区(根据 `fat16_ins` 所指结构体中记录的根目录扇区偏移来进行索引)，把根目录的所有条目装入 `pathInput` 中即可；如果 `path` 表示子目录，则先调用 `find_root` 函数填充相应目录的信息值 `Dir` 所指的结构体，之后再进行把该目录下的条目装入 `pathInput`.

e. `fat16_read`:

从 `path` 对应的文件 `offset` 偏移处开始读 `size` 大小的文件，至 `pathInput` 中，供 `fuse` 使用。

操作执行时，实际上只是对之前已经实现的 `read_path` 函数进行了封装

2. 阅读 `libfuse` 源码，试解释本实验中使用到的 `fuse_main()` 函数

答:

`fuse_main` 函数实际上没有定义。根据 `fuse.h` 中的宏定义，`fuse_main` 被替换为了 `fuse_main_compat2`；

```
# if FUSE_USE_VERSION == 21
#     define fuse_operations fuse_operations_compat2
#     define fuse_main fuse_main_compat2
```

而阅读 `fuse_main_compat2` 定义后发现，它实际上只是调用了 `fuse_main_common` 函数，并且为其固定增添了参数。所以，`fuse_main` 实际上执行的是 `fuse_main_common`

`fuse_main_common` 函数定义如下：

```
static int fuse_main_common(int argc, char *argv[],
                            const struct fuse_operations *op, size_t op_size,
                            void *user_data, int compat)
{
    struct fuse *fuse;
    char *mountpoint;
    int multithreaded;
    int res;

    fuse = fuse_setup_common(argc, argv, op, op_size, &mountpoint,
                            &multithreaded, NULL, user_data, compat);
    if (fuse == NULL)
        return 1;

    if (multithreaded)
        res = fuse_loop_mt(fuse);
    else
        res = fuse_loop(fuse);

    fuse_teardown_common(fuse, mountpoint);
    if (res == -1)
        return 1;
```



```
    return 0;
}
```

它主要是调用了 `fuse_setup_common` 函数，并且根据其返回值进行了相应的判断。

根据 lab4 实验讲义中所给的关于 fuse 的介绍博客，`fuse_setup_common` 函数则调用了 `fuse_mount_common`，然后使用创建子进程执行 `fusermount` 程序，确保 fuse 模块已经被加载；`fuse_setup_common` 函数还调用了 `fuse_new_common` 函数，它完成分配 fuse 数据结构的工作，存储并维护一个文件系统数据镜像缓存 `cached`，返回到 `fuse_main_common`

之后，根据返回值（fuse 和 `multithreaded`），`fuse_main_common` 会进行判断，执行 `fuse_loop` 或者 `fuse_loop_mt`，其中后缀 **mt** 是多线程 `multithreaded` 的缩写。这两个函数都可以从设备 `/dev/fuse` 读取文件系统调用，调用 `fuse_main()` 之前调用存储在 `fuse_operations` 结构体中的用户态函数。这些调用的结果回写到 `/dev/fuse` 设备（这个设备可以转发给系统调用）。

其中 `fuse_loop` 定义如下：

```
int fuse_loop(struct fuse *f)
{
    if (!f)
        return -1;

    if (lru_enabled(f))
        return fuse_session_loop_remember(f);

    return fuse_session_loop(f->se);
}
```

如果执行的是 `fuse_loop`，那么接下来会执行 `fuse_session_loop`：fuse 通过 `fuse_session_loop`（单线程）来启动 fuse 守护程序，守护程序不断的从 `/dev/fuse` 上读取请求，并进行相应的处理。

实验总结：

1. 通过本次试验，我不仅复习了 FAT 文件系统的许多特性，还学习到了关于 FAT 使用的一些细节问题：比如 BPB 参数块记录了哪些信息、FAT 表怎么存储下一个簇的编号、具体一些未使用的簇如何定义等等……
2. 在补全代码后进行第一个测试时出现过问题，原因是 `find_subdir` 函数中一个 `if` 中的 `==` 写成了 `=`；我前期主要在 mac OS X 下编程，使用的编辑器是 vs code，由于项目中有许多宏、结构体的相关定义，导致编辑器反应有些迟钝，没有在编辑过程中第一时间发现问题，而是编译运行后才发现问题
3. 我使用 gdb 进行 debug，在 Makefile 中为 gcc 编译添加了 `-g` 选项；需要给 main 传参的 debug 已经很久没有用到；即使在 linux 的命令行中 debug 没有 GUI，还是很顺利的发现了问题：出错的第二个测试，它显示的错误文件名是上级目录名，我就很快对应到，是在“判断是否达到了最终深度”时出错，进行了相应的修改
4. 第二个测试中，pdf 打开不正常。在确认 5 个 TODO 部分代码补充无误后，我开始认为是 fuse 相关函数（具体是 `fat16_readdir`）出错，还查了很多相关资料想要改正。后来助教通知 pdf 显示不正常可以接受，是浏览器的问题，于是我就忽略了它。