实验四 FAT文件系统的实现 Part2

> 提示：本文档涉及一些对Part1问题的更正。所以只要你计划完成Part1，就建议你阅读本文档的部分内容。

实验目标

- 熟悉文件系统的基本功能与工作原理（理论基础）
- 熟悉FAT16的存储结构，利用FUSE实现一个FAT文件系统：
  - 文件目录与文件的读（只读文件系统，基础）
  - 文件、目录的创建与删除（基础）
  - 文件的写（进阶）
- 在实现fat16文件系统的基础上，优化文件系统性能（进阶，开放性）

实验环境

- VMware / VirtualBox
- OS: Ubuntu 20.04 LTS
- Linux内核版本: 5.9.0+
- libfuse3

实验时间安排

注：此处为实验发布时的安排计划，请以课程主页和课程群内最新公告为准

- 6月6日晚实验课，讲解第一部分及检查实验
- 6月13日晚实验课，讲解第二部分及检查实验
- 6月20日晚实验课，检查实验
- 6月27日晚实验课，检查实验
- 7月2日 18:00 前，提交实验报告及代码

实验报告及实验代码提交

- 提交代码文件 `simple_fat16.c`
- 提交你的实验报告（推荐为 PDF 格式）
- 上传至 **BB系统**

第一部分：对Part1的补充说明

1.0 关于代码版本的说明

我们在Part2中发布了完整的实验代码和文档包 `lab4-all.tar.gz`，其中包括测试文件在内的多个文件相比于Part1发布的版本稍有修改，我们建议你根据目前的实验情况，选择以下一种方式处理你的代码：

- 如果你**还未开始进行实验**：请进行忽略之前发布的 `lab4-all.tar.gz`，**使用 `lab4-all.tar.gz` 中的文件完成实验即可。**
- 如果你**已经开始了Part1的实验**：请用你的 `simple_fat16.c` 替换解压后 `lab4-all` 内的相同文件，然后再使用 `lab4-all` 进行实验。

代码更新说明

本部分说明了新发布的 `lab4-all.tar.gz` 相对于 Part1 发布代码的区别，如无特殊情况，你**不需要**关注本节内容，直接使用新的代码完成实验即可。

- 增加了 `hello.c`，相应修改了Makefile，帮助大家理解和测试fuse安装。
- 更新了 `pytest` 的调用方式，通过 `python3 -m pytest` 调用，防止使用 `pip` 安装了 `pytest`，在运行测试时仍提示找不到 `pytest` 命令的问题。
- 更新了测试文件生成代码，将 `tree` 目录下所有文件名均改为小写，防止大小写不匹配的问题。
- 更新了测试代码，将测试中的二级子目录由 `subsubdir` 改为 `ssdir`，防止目录名超出8字符长度导致的问题。
- 将每磁道的物理扇区数由 `2048` 减至 `512`。
- 增加了性能测试相关脚本。

1.1 实验环境配置

除了 Part1 中提到的安装 `libfuse3-dev`，还需要单独安装 `fuse3` 包，即运行：

```
sudo apt install fuse3
```

完整实验环境配置过程如下：

```
sudo apt update # 从软件源更新索引
sudo apt install fuse3 libfuse3-dev pkg-config python3 python3-pip # 安装可能需要的软件包
pip install pytest # 安装pytest
```

本次实验提供的代码中增加了 `hello.c`，这是一个 `fuse3` 自带的样例程序，实现了一个只能读取的文件系统，文件系统下只有一个 hello 文件，可以用于测试 fuse 是否正确安装。测试方法如下：

```
make hello # 编译 hello 文件系统
mkdir hi   # 建立一个空目录作为挂载点
./hello -d hi  # 运行 hello，挂载至 hi 目录
```

上述程序运行后，在另一个终端中运行：

```
ls hi
# 输出 hello
cat hi/hello
# 输出 Hello World!
```

说明 FUSE 安装成功。感兴趣的同学也可以通过查看 `hello.c` 来了解 FUSE 是如何实现一个文件系统的。（`hello.c` 的代码相比本实验代码更为简单，所以阅读它可能更容易理解FUSE的使用方式。）

1.2 实验运行及调试指南

本次实验中概念较多，不熟悉的同学在完成实验、运行代码和调试的过程中可能会遇到各种问题，本部分对代码运行和调试做进一步说明，帮助同学们更好地完成实验。另外，附录中提供了我们给出的原始镜像文件中的文件结构，你可以将程序输出与该结构对比，来确认程序正确性。

挂载和挂载点

Part1 中我们提到，运行 `simple_fat16` 后，文件系统被挂载到一个目录下。对 Linux 不熟悉的同学可能会疑惑什么是**挂载**。“挂载”实际上是指 linux 上使存储设备（例如硬盘、CD等）上的文件和目录可供用户通过计算机的文件系统访问的过程。这个过程把一个文件系统连接到了另一个文件系统的某个目录下。

为什么需要挂载呢？想想我们在使用 Windows 时，可以有多个磁盘，通过磁盘标识符（`C:`、`D:`...）来访问不同的磁盘。但在 linux 中，我们只有一个根目录，根目录下文件也并不是按磁盘分类的，那如果我们有两块磁盘怎么办？挂载就是解决这个问题的一种方式，例如根目录在磁盘 A 上（相当于 Windows 中的 C 盘），则我们存储在 `/` 下的所有文件实际都存放在磁盘 A 上。然后假设我们将磁盘 `B` 挂载到 `/data` 下，注意，这里 `/data` 需要是磁盘 A 中的一个 **空目录**，称为**挂载点**。之后，我们访问 `/data` 下的文件时，实际上就访问了磁盘 B（和上面的文件系统），创建的文件实际也会存储在 B 中。这样，我们就能通过唯一的根目录 `/` 访问不同的磁盘或者其它存储设备。

在本次实验中，我们的镜像文件就是一个存储设备，其中保存了 FAT16 格式的文件数据。实验中，我们会将其挂载到 `./fat16` 目录下，此时访问该目录，就能够访问到我们的文件系统。运行我们的程序时，fuse实际上就进行了挂载操作，而程序结束时，该操作被卸载。因此，在程序运行时，我们能访问镜像文件中的数据，而程序结束后，`./fat16` 只是根文件系统下一个空文件夹而已，此时我们再访问这个文件夹，再里面创建文件等，就与 `simple_fat16` 无关了。（因为挂载要保证挂载点是空目录，所以在程序结束后，不要再 `./fat16` 中创建文件哦，这样会导致下次运行程序失败。如果不小心创建了文件，只需要清空该文件夹在运行程序即可。）

如何打印调试信息

在 Part1 的说明中，我们提到可以用以下方法运行程序：

```
./simple_fat16 -s -d ./fat16/ --img=fat16.img
```

其中 `-d` 参数表示以 Debug 模式运行程序，`./fat16/` 是上一节提到的**挂载点**，`--img` 指示镜像位置。

为什么我们要以 `-d` 模式运行程呢？实际上该参数会将程序保持在前台运行。如果去掉该参数，我们的文件系统将自动运行在后台，我们无法判断程序的运行状态（例如程序是否异常终止，是陷入死循环还是返回了错误，在程序中printf时，我们也无法看到输出）。通过 -d 参数，我们能像运行普通 C 程序一样运行我们的文件系统程序，并观察程序输出。

但是 `-d` 参数会自带一些 FUSE 输出的，难以理解的调试信息，这可能将我们需要的信息（如自己printf的程序状态）淹没在其它调试信息里。为了避免这种情况，我们可以使用 `-f` 而不是 `-d` 来运行程序：

```
./simple_fat16 -s -f ./fat16/ --img=fat16.img
```

`-f` 的意义是前台运行程序，但不输出调试信息，这样，FUSE本身将不再输出，只有我们在代码中输出的内容会显示在终端里，于是我们可以非常方便的调试我们的代码。

文件操作到 FUSE 函数的转换

我们需要在终端中对文件系统操作来调试我们的程序，但我们不能直接在终端中调用程序里的函数，而只能用shell命令来间接调用不同函数。但单一的shell命令通常会调用多个FUSE里的函数，来实现目标。例如，`ls` 命令会根据文件类型（目录还是文件）以及文件权限将输出染成不同颜色，但 `readdir` 本身只提供了每个文件（或目录）的文件名。因此，`ls` 命令在获得目录下每个文件名后，会依次访问每个文件，获得其属性。如下是一次 `ls ./fat16` 的访问流程（其中 `getattr` 是获取文件属性的函数，在提供的代码中已经实现）：

```
readdir(path='/')
getattr(path='/large.txt')
getattr(path='/small')
getattr(path='/tree')
getattr(path='/')
```

可以看到，一次 `ls` 操作实际上调用了5个函数。如果其中某些函数出错，就会使得命令出现奇怪的错误。例如，`readdir` 正常返回了三个文件名，但 `getattr` 时，文件系统却提示找不到 `large.txt`，这时候 `ls` 就可能出现显示了 `large.txt` 但同时提示找不到 `large.txt` 的错误。（虽然 `getattr` 代码已经实现，但其中调用了 `find_entry`，所以如果 `find_entry_internal` 没有正确实现，该函数还可能提示错误。）

因此，在调试时，我们要首先搞清楚我们的操作对应到哪些程序中的函数。我们可以通过在每个函数前加上 `printf`，来判断哪些函数被调用了。（在提供代码的大部分函数前，已经写好了这样一条 `printf`，但也有些函数没有，可以自行加入。）你可以使用上节提到的 `-f` 而不是 `-d` 参数运行程序，使得输出更清晰。

一个较为基础的经验是，大部分操作都会调用 `getattr`，因此遇到奇怪的问题时，记得检查一下 `find_entry` 是否正确。

另外，为了方便调试，我们给出以下用到不同函数对应的 shell 命令：

1. `readdir`：`ls`，`tree`
2. `read`：`cat`，`head`，`tail`
3. `mknod`：`touch`
4. `unlink`：`rm`
5. `mkdir`：`mkdir`
6. `rmdir`：`rmdir`，`rm -r`
7. `write`：`echo somthing > [file]`
8. `truncate`：`truncate`

1.3 目录项文件名判断

本次实验简化了 FAT 中的文件名判断，判断目录项中 8+3 格式文件名是否是用户传入的需要查找的文件名，可以使用 已经实现的 `checkname` 函数。这个将用户传入的长文件名转换为 8+3 文件名，如果转换后的结果相同，则认为是相同文件。所以，本次实验中文件名不区分大小写，且不区分长度超过8的文件名。例如，通过 `LARGE.TXT` 也可以访问 `large.txt`；创建一个 `longlonglongname.txt` 后，可以通过

`longlong.txt` 或者 `longlonganything.txt` 来访问相同文件。

如果你更新到了 `lab4-all`，提供的测试代码中不包含超过 8 字符的文件名，或者大小写不匹配的情况，你可以忽略该条，按你自己的设计比较文件名。如果你使用 part1 提供的测试代码，则你需要严格按照上述要求（推荐直接使用 `checkname` 函数），实现文件名比较，否则会出现大小写不匹配，和 `subsubdir` 无法找到的问题。

第二部分：实验任务

2.1 任务三：实现FAT文件系统写操作

为了支持 FAT16 文件的写操作，需要实现如下两个函数：

```
int fat16_write(const char *path, const char *data, size_t size, off_t offset, struct
fuse_file_info *fi);
int fat16_truncate(const char *path, off_t size, struct fuse_file_info* fi)
```

函数解释说明：

- `fat16_write`：将 `data[0..size-1]` 的数据写入到 `path` 对应的文件的偏移为 `offset` 的位置，如果写入的数据超出了文件末尾，则应相应增大文件大小。`fi` 参数在本次实验中可忽略。
- `fat16_truncate`：将 `path` 对应的文件大小设置为 `size`。如果 `size` 比原文件的大小要小，则从末尾处截断文件；如果 `size` 大于原文件的大小，那么将新增加部分的数据初始化为 `0`。同样 `fi` 参数可忽略。

当我们在终端中使用类似 `echo hello >> file` 的命令，通过 `>>` 重定向符向文件末尾追加写时，只会调用 `fat16_write` 函数。若我们使用 `>` 重定向符，如运行 `echo hello > file` 时，则会先调用 `fat16_truncate` 将文件大小设置为0（或如果文件不存在，会创建文件），再调用 `write` 向文件中写入。所以为了支持文件写，两个函数都需要实现。我们也可以使用 `truncate` 命令来直接测试 `fat16_truncate` 的功能。

2.1.1 文件写入

首先看一下 `fat16_write` 的实现思路：

1. 通过 `path` 获取到对应的目录项，即 `DIR_ENTRY` 结构体，通过调用 `find_entry` 即可（任务一中实现）。
2. 比较新写入的数据和文件原来的数据所需的簇数量，如果新写入的数据所需的簇数量大于原来所需的簇数量，则需要为文件扩容簇数量。即在FAT表中查找未分配的簇，并链接在文件末尾。如果你完成了前置任务中的 `alloc_clusters`，你可以在此处使用它，为你分配你所需要的簇数量，你只需要正确地将分配好的簇链连接在你文件的末尾即可。（如果文件原来没有任何簇，你要把簇链连接在目录项的 `DIR.FstClusLo`，否则，连接在原文件簇链的末尾。）
3. 按需扩容后，只需要实现写文件操作即可，通过 `DIR.FstClusLo` 获取第一个簇号，之后可以通过链接依次遍历，找到要写入簇的位置，并在读取相应的扇区，修改扇区的正确位置，并写回扇区。（这个过程和 `read` 的实现很类似。）
4. 更新对应的目录项，上述过程中，可能修改了目录项的 `FstClusLo`，最后，你还需要更新 `FileSize`，然后将更改后的目录项写回镜像文件。（使用 `dir_entry_write`）
5. 返回成功写入的数据字节数。

`fat16_write` 函数的流程有些复杂，为了简化代码逻辑，在提供的代码中，我们将这一过程拆分成了以下多个函数：

```
// 要实现的函数本身，完成流程的第一步，并正确调用write_file
int fat16_write(const char *path, const char *data, size_t size, off_t offset,
                struct fuse_file_info *fi);

// 给文件分配新簇，将文件拓展至可容纳size的大小
int file_reserve_clusters(DIR_ENTRY* dir, size_t size);

// 相簇的相应位置写入长度为size的数据data
ssize_t write_to_cluster_at_offset(cluster_t clus, off_t offset, const char* data,
size_t size);
```

你可以在我们提供的代码中找到这些函数的详细功能以及参数说明。你可以通过正确补充这些函数来实现文件写入的功能。当然，和前面的任务一样，你也可以不实现这些函数，而通过自己喜欢的逻辑实现 `fat16_write` 。

2.1.2 文件截断/拓展

`truncate` 意为“截断”，但实际上这个函数也能实现文件拓展，所有下文我们也用“截断”一词统称改变文件大小的操作。

接下来是 `fat16_truncate` 的实现思路：

1. 通过 path 获取到对应的目录项，即 `DIR_ENTRY` 结构体，通过调用 `find_root` 即可
2. 计算并比较原文件拥有的簇数量 `n1` ，和文件截断后需要的新簇数量 `n2` 。通过比较 `n1` 和 `n2` 的大小，来判断文件是否需要新增或释放簇：
    1. `n1 == n2` ：此时不需要任何操作
    2. `n1 < n2` ：此时需要扩容文件大小，可以参照 `fat_write` 实现思路。
    3. `n1 > n2` ：此时需要截断文件，找到文件的第 `n2` 个簇，更改FAT表项，将此处改为文件末尾，并释放后续所有簇。（释放可使用Part1中的 `free_clusters` 函数。）
3. 按需写入，比较新文件的大小 `new_size` （就是参数中的 `size` ）和原来的文件大小 `old_size` ：
    1. `new_size > old_size` ：需要从 `old_size` 位置开始，将后续文件数据清空为 `0x00` 。如果你正确实现了 `alloc_clusters` ，那么新分配的簇应该已经清空，而已有的最后一个簇的末尾也在上一次分配时清空过。
2. `new_size <= old_size` ：不需要进行任何操作
3. 实现目录项的更新，通过 `find_root` 可以获取到目录项对应的偏移量，更新对应的目录项数据写入即可
4. 返回 0 表示正常结束，否则表示异常

2.2 任务四：FAT 文件系统性能优化

2.2.1 磁盘寻道时间

我们的文件系统通过读写镜像文件模拟对硬盘的读写，但实际硬盘运行中，对硬盘不同磁道进行读取时，会产生较长的寻道时间。为了模拟该寻道时间，我们为 `sector_read` 和 `sector_write` 增加了寻道等待时间功能，在运行时，给 `simple_fat16` 提供 `seek_time` 参数，读写扇区时，就会根据读写的上一个扇区和当前扇区所在磁道，模拟不同的寻道时间。如下：

```
./simple_fat16 -f ./fat16 --img=./fat16.img --seek_time=10
```

`seek_time` 指定的是磁头每移动一个磁道所需的时间，单位为微秒。由于本次实验的镜像文件较小，当前设置中，每个磁道包含512个扇区（即 256KB）。所以，当读取 0 号扇区后再读取 512 号扇区时，就会有 10 微妙的寻道延迟。通过这个延迟，我们模拟了磁盘顺序访问速度快、随机访问速度慢的特性。

2.2.2 实验内容

本实验的任务四，目标是在上述模拟磁盘环境下，优化我们实现的 FAT16 文件系统性能。具体而言，我们为文件系统写了简单的工作负载，并且提供了助教实现的标准文件系统的二进制程序。你的程序需要在模拟磁盘环境（`--seek_tim=10`）下运行特定工作负载下，并与助教提供的标准程序在相同环境和负载下的运行时间对比，达成性能优化目标。（注：这部分内容可能会花费较长时间，我们推荐有学有余力的同学尝试完成这部分内容。）

实验目标

- 运行时间在标准代码运行时间的两倍以内。（1分）
- 运行时间在标准代码运行时间的 1/2 以内。（额外 +1 分）

工作负载

本任务的工作负载在 `test/fat16_bench.py` 内实现，具体工作过程如下：

1. 在指定目录随机创建40个文件。（每个文件可能在随机的多级子目录下。）
2. 随机进行2000次写入，每次写入随机挑选上述文件中的一个，向文件中追加写入 2333 字节的字符。
3. 随机进行2000次读取，每次读取随机挑选上述文件中的一个，在文件内随机位置开始，读取 2333 字节的内容。

优化思路

本次实验的设计思路可以由同学们自由探索，鼓励同学们自行对文件系统进行测试，找出影响性能的主要因素，并自行设计方案进行优化。这里仅提供一些开放性的思考方向：

1. 本实验的各个操作中，是否有哪些总是需要重复完成的部分？我们能不能减少这些部分的重复次数？
2. 工作负载中有大量随机读写，会导致寻道时间增加，我们有没有办法减少一部分寻道时间？
3. 我们有没有简单的方式来利用内存加速我们的文件系统，但不影响文件系统完整性和一致性？（注：文件系统一致性，即文件系统每部分数据之间要互不冲突。例如，若FAT中某文件长度需要3个簇来容纳，FAT表中却只给其分配了两个簇，此时文件系统就并不一致。）

这些思路带来并不和优化方案一一对应，只是作为一些思考方向，我们不限制优化的实现方式，只要你能保证文件系统的正确性，并且达到实验目标即可。但是，请你在**实验报告中详述你的设计思路和优化方案**（包括可能进行的对文件系统的测试等），如果你不能说明你的优化方案，你将无法获得性能 200% 的额外加分。

测试方法

我们实现了 `run_bench.sh` 脚本来进行自动化性能评估。`run_bench.sh` 的过程中，会自动调用标准程序和你的程序分别运行一次工作负载，并记录运行时间，并打印。最终输出的结尾可能如下：

```
...
Read 1500/2000 times.
Read 1600/2000 times.
Read 1700/2000 times.
Read 1800/2000 times.
Read 1900/2000 times.
26.715387110991287
std time:
27.06488428299781
your time:
26.715387110991287
```

注意最后四行分别输出了标准时间和你的程序的运行时间（单位为秒）。标准程序大约需要30s完成上述工作负载（在助教的电脑上），所以整个脚本可能需要一分钟甚至更长的运行时间。脚本运行过程中会显示运行进度（如上图中的`Read .../2000 times.`），你可以观察进度来判断程序是否卡死。

你也可以手动运行工作负载脚本。假设你已将你的文件系统挂载至`./fat16`目录，运行以下命令即可在该目录测试性能：

```
python3 test/fat16_bench.py ./fat16
```

要注意，你的程序性能可能和文件系统内文件的数量、文件系统以使用空间的大小相关，所以请使用未修改的，我们提供的原始镜像文件来进行测试，以得到较为准确的性能测试结果。（建议将原始镜像复制进行备份，需要还原时将镜像复制回来即可。）

第三部分 检查内容 & 评分标准

注：为平衡实验难度，减轻同学们的实验压力，本次实验任务一、二难度较低分值较高，任务三、四较为复杂但分值较低，同学们可根据自身时间情况选择性完成。

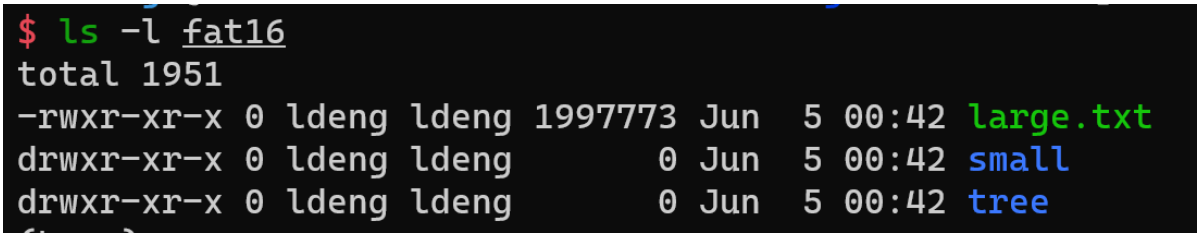
1. 任务一：实现FAT文件系统读操作（**满分5分**）
    - 能够运行`tree`，`ls`命令查看文件目录结构，通过`TestFat16List`下的一个测试（**3分**）
    - 能够正确读取小于一个簇的文件，通过`TestFat16ReadSmall`下的两个测试（**1分**）
    - 能够正确读取长文件，通过`TestFat16ReadLarge`下的三个测试（**1分**）
    - 讲解你的代码
2. 任务二：实现FAT文件系统创建/删除文件、目录操作（**满分2分**）
    - 能够运行`touch`、`mkdir`命令创建新文件、目录，要保证文件属性的正确填写
    - 能够运行`rm`、`rmdir`、`rm -r`命令删除已有文件、目录，要保证簇的正确释放
    - （只检查这个）通过`TestFat16CreateRemove`下的8个测试（**2分**）
    - 讲解你的代码
3. 任务三：实现FAT文件系统写操作（**满分2分**）
    - 能够正确写入文件，通过文件写入、截断测试（`TestFat16Write`下的四个测试）（**2分**）
    - 讲解你的代码
4. 任务四：FAT 文件系统性能优化（**满分1分**）
    - 在模拟磁盘环境下，性能达到基准测试 50%，以`test/run_bench.sh`显示的时间为准（**1分**）
    - 在模拟磁盘环境下，性能达到基准测试的 200%，以`test/run_bench.sh`显示的时间为准。（**额外+1分**）

- 向助教说明你的优化设计，并提交相应实验报告

附录

镜像文件结构

为了方便大家调试，这里给出我们镜像文件中的目录结构。镜像文件的根目录下，包含一个大文件 `large.txt` 和两个目录 `small` 和 `tree` 。



`large.txt` 是一个大文本文件，文件大小为 1997773 字节，其中包括199778行，除了最后一行外，每行10个字符，由行号和下划线组成。最后一行只有三个字符 `end` 。文件的开头和结尾如下：

```
$ head fat16/large.txt -n 5
0________
1________
2________
3________
4________


$ tail fat16/large.txt -n 5
199773___
199774___
199775___
199776___
end
```

`small` 目录下是20个小文件，文件名分别为 `s00.txt` ~ `s19.txt` ，第 `i` 个小文件的大小是 `4 * i` ，文件中的内容是将4位文件序号（ `i-1` ）重复 `i` 次。下面展示了 `small` 目录其中一部分内容，和几个文件中的内容：

```
$ ls -l fat16/small
total 0
-rwxr-xr-x 0 ldeng ldeng  4 Jun  5 00:42 s00.txt
-rwxr-xr-x 0 ldeng ldeng  8 Jun  5 00:42 s01.txt
-rwxr-xr-x 0 ldeng ldeng 12 Jun  5 00:42 s02.txt
-rwxr-xr-x 0 ldeng ldeng 16 Jun  5 00:42 s03.txt
...
-rwxr-xr-x 0 ldeng ldeng 76 Jun  5 00:42 s18.txt
-rwxr-xr-x 0 ldeng ldeng 80 Jun  5 00:42 s19.txt

$ cat fat16/small/s00.txt
0000

$ cat fat16/small/s01.txt
00010001

$ cat fat16/small/s06.txt
```

```
000600060006000600060006000600060006

$ cat fat16/small/s17.txt
00170017001700170017001700170017001700170017001700170017001700170017001700170017
```

`tree` 目录是一个较深的大目录，里面包含了很多级子目录和文件。每一级子目录都由上一级目录名为前缀。结构如下（绿色为文件，蓝色为目录）：