# 文件系统格式说明

示例文件系统采用了非常简单的格式，主要由两个区域组成。可参考mkfs.c中的代码。

## 目录区域

目录区域记录了硬盘中所有文件的元信息。每个文件的元信息由如下一个结构体表示：

```
struct _dir {
        size_t offset;
        size_t size;
        char   name[128];
};
```

三个字段分别表示文件实际在硬盘中的起始位置、文件大小和文件名。需要注意的是，实际文件的元信息是从目录区域的第1个记录开始的，第0个记录实际使用`size`字段记录了文件的数目。

## 文件内容区域

文件内容区域根据元信息中提供的偏移信息，存储了文件的实际内容。

# 制作硬盘镜像

使用示例代码，在命令行中执行`make`即可编译生成相应的可执行文件*mkfs*和示例程序文件*test*，并将把多个文件打包存入硬盘镜像fs.img中。

如果要将自己的文件到放入fs.img，可使用如下命令（默认fs.img的大小只有500KB，如果需要更大的空间，请修改mkfs.c中`#define DISK_SIZE 500 * 1024`）：

```
./mkfs fs.img file1 file2 file3 ...
```

# 构建自己的用户态程序

示例的`test.c`和`asm.S`以及Makefile中相关的指令展示了如何构建自己的用户态可执行程序：

1. 将完全不涉及内核态操作的所有代码编译为`.o`文件（注意使用编译选项`-m32`）
2. 将多个`.o`文件链接（`ld`）成为可执行文件，使用如下选项

| 选项 | 说明 |
| --- | --- |
| `-m elf_i386` | 生成i386格式的二进制 |
| `-e main` | 执行代码起始位置（入口函数） |
| `-Ttext 0x80001000` | 指定代码段装载位置（此处为2GB处，可修改为其他位置） |

在构建自己的用户态程序时，可以将所有用户态代码放在一个.c源文件中，也可以如示例一般分做多个源文件，最后链接成一个Xinu上的可执行文件。

代码段装载位置（示例中的0x80001000）表示用户态程序中的所有指令和数据区域的偏移均以此作为最基础的基地址。通过`objdump -d test`可以看到main函数的起始地址（如下）。该值应根据实际用户态代码空间在虚拟内存空间中的起始点做出调整。如果用户态空间起始位置为0x80000000，用户态程序整个从这个位置开始加载，对于示例test而言，0x80001000刚好是该程序中代码段的实际线性地址。

```
80001000 <main>:
80001000:       f3 0f 1e fb             endbr32
80001004:       8d 4c 24 04             lea    0x4(%esp),%ecx
80001008:       83 e4 f0                and    $0xfffffff0,%esp
```

# ELF格式

示例test的文件头16进制表示如下：

```
00000000: 7f45 4c46 0101 0100 0000 0000 0000 0000  .ELF............
00000010: 0200 0300 0100 0000 0010 0080 3400 0000  ............4...
00000020: c831 0000 0000 0000 3400 2000 0500 2800  .1......4. ...(.
00000030: 0800 0700 0100 0000 0000 0000 0000 0080  ................
00000040: 0000 0080 d400 0000 d400 0000 0400 0000  ................
00000050: 0010 0000 0100 0000 0010 0000 0010 0080  ................
00000060: 0010 0080 5e00 0000 5e00 0000 0500 0000  ....^...^.......
00000070: 0010 0000 0100 0000 0020 0000 0020 0080  ......... ... ..
00000080: 0020 0080 6000 0000 6000 0000 0400 0000  . ..`...`.......
00000090: 0010 0000 0100 0000 0030 0000 0040 0080  .........0...@..
000000a0: 0040 0080 0c00 0000 0c00 0000 0600 0000  .@..............
000000b0: 0010 0000 51e5 7464 0000 0000 0000 0000  ....Q.td........
000000c0: 0000 0000 0000 0000 0000 0000 0700 0000  ................
000000d0: 1000 0000 0000 0000 0000 0000 0000 0000  ................
```

ELF文件头的数据结构可从elf.h中找到，主要包含以下两个：

```
// File header
struct elfhdr {
  uint magic;  // must equal ELF_MAGIC
  uchar elf[12];
  ushort type;
  ushort machine;
  uint version;
  uint entry;
  uint phoff;
  uint shoff;
  uint flags;
  ushort ehsize;
  ushort phentsize;
  ushort phnum;
  ushort shentsize;
  ushort shnum;
  ushort shstrndx;
};

// Program section header
struct proghdr {
  uint type;
  uint off;
  uint vaddr;
  uint paddr;
  uint filesz;
  uint memsz;
  uint flags;
  uint align;
};
```

ELF文件起始即为一个elfhdr结构，对照可得，从0x00到0x34的数据分别对应了elfhdr中的各个字段。关键字段包括：

| 字段 | 值 | 说明 |
| --- | --- | --- |
| type | 0200 | 0x02， 表示可执行文件 |
| entry | 0010 0080 | 0x80001000，代码段入口地址 |
| phoff | 3400 0000 | 0x00000034，ELF文件偏移0x34开始即为proghdr数据 |
| phentsize | 2000 | 0x0020，每个proghdr大小为0x20字节 |
| phnum | 0500 | 0x0005，有5个proghdr数据 |

5个proghdr数据分别位于[0x34, 0x54)、[0x54, 0x74)、[0x74, 0x94)、[0x94, 0xb4)、[0xb4, 0xd4)，每个数据结构表示ELF中一个有效的section。其中前4个的`type`均为1，根据xv6的代码，这种section是可

以加载的。这4个section的起始地址分别是0x80000000、0x80001000、0x80002000、0x80004000，长度分别是0xd4、0x5e、0x60、0x0c。第一个section即为ELF头部信息本身（elfhdr+proghdr），后边三个分别对应test中的`.text`、`.eh_frame`和`.got.plt`这3个部分，通过`objdump -D test`可以查看（如下）。

```
Disassembly of section .text:

80001000 <main>:
80001000:       f3 0f 1e fb             endbr32
        ...
8000105d:       c3                      ret

Disassembly of section .eh_frame:

80002000 <.eh_frame>:
80002000:       14 00                   adc    $0x0,%al
        ...
8000205c:       00 00                   add    %al,(%eax)
        ...

Disassembly of section .got.plt:

80004000 <_GLOBAL_OFFSET_TABLE_>:
        ...
```

这些section（尤其是后边3个）应该被加载到适当的虚拟内存位置（暂不考虑读/写/执行属性等），每个section由于是页对齐的，所以应该以页为单位分配内存，而proghdr中标明的长度，可作为从ELF文件中读取的数据的大小（没必要读取过多的无效数据进入内存）。

> 文件内偏移0x3000，但是地址实际上是0x80004000，在加载到虚拟内存空间时应注意此间区别。这里的_GLOBAL_OFFSET_TABLE_没有起到实际作用，而如果是`.data`、`.bss`之类的段如果有类似情况，加载地址错误会导致程序运行异常。

二进制文件test中偏移0x3000开始的部分，根据proghdr信息，在文件中偏移0x3000（`off`），地址0x80004000（`vaddr`和`paddr`），大小0xc（`filesz`和`memsz`），12字节全是数据0。
以16进制格式打开文件，后接的是`4743 433a ...`，在`objdump -D test`的结果中对应`.got.plt`之后的`.comment`段，可以看到是一段注释信息，如*GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0*，再后边一段则是文件中的字符串表，如*test.c*、*uprintf*等，在前述的proghdr中并没有列出此段的信息。

elfhdr中字段`shoff`表示二进制中的section header，相关数据结构没有列出，但其值为0x31c8，`shentsize`为0x28，`shnum`为0x8，也就是文件偏移0x31c8开始直到文件末尾（[0x31c8, 0x3308)）都是section header的内容。

通过`readelf -h test`可以得到elfhdr的信息，使用`-l`、`-S`选项则分别能获得proghdr和section header的信息。

> xv6只根据proghdr信息进行加载，而不考虑section header，因此只需按照proghdr中的信息读取文件并放到内存相应位置即可。另可在测试程序中加入全局变量，再编译后查看ELF文件头中的proghdr数据，并与二进制文件对照，以更清晰地理解哪些部分是有用的数据、各部分应放在内存什么位置。