author: archychu titile: mips disk operation

第四部分

2013.11.26

陈译 汪乾文 朱霖潮

- **文件系统**
- **读写等文件操作**
- **执行一个文件**

Part 1 文件系统

几个参数

```
#define ONE_SECTOR          512
#define ONE_CLUSTER         ((1) * (ONE_SECTOR))
#define ONE_PAGE            ((1) * (ONE_CLUSTER))
```

磁盘布局

FAT_LIST能稍微快一点吧

```
             |--------------------------|
 SECTOR 0    |  number of fat table     | <- OFF 0
             |--------------------------|
             | cluster num of the first | <- OFF 4
             |fat table, and it is 'set'|
             |       to be 1.           |
             |--------------------------|
             |cluster num of the second | <- OFF 8
             | fat table                |
             |--------------------------|
             |                ..        |
             |                ..        |
             |  end of the first sector |
             +--------------------------+
```

也就是说我们总共有大约512 × (512 / 4)× 512 = 32M的硬盘 **够大了！**

SECTOR 1

第一个FAT表

- FAT中，结束符为0xffff_ffff，空为0x0000_0000
- fat项: 刚开始的1(fat_list) + 1(fat_table_1) + 1(根目录区) + 64(OS部分)个扇区已经被分配，初始化为-1,其他部分初始化为0

SECTOR 2

- 只有一个目录区
- 一个文件在目录区会留下这样一个记录:

```
#define FILE_NAME_LEN       15
typedef struct
{
    char dirName[FILE_NAME_LEN]; // 文件名
    char dirAttributes;          // 文件属性, 暂时没有用到
    int  dirStartCluster;        // 文件起始簇号
    int  dirFileSize;        // 表示文件的长度, 单位为byte
} DirEntry;                  // size of this is 24 bytes!
```

- 比如根目录区初始化为：

```
dirName         "/"
dirStartCluster 2
dirFileSize     16
```

SECTOR 3-66

- 所谓的os部分，包括.data, .text
- .data和.text分别为16K。
- 也就是说空闲的扇区是从67号开始的

内存分布

- ROM和RAM统一编址
- 开机时PC <- 0

```
$a0 = 512
$a1 = 3 * 512
$a2 = 64 * 512      ; 读取OS部分, 偏移扇区为3, 大小64个扇区
```

```
    jal read_disk
    j       512         ; 开始OS初始化
read_disk:
    ...
```

- **OS初始化, 几个数据结构：**
    - Kernel_buffer:

```
#define KERNEL_BUF_SECTOR   10
typedef struct
{
    /* buffer中这个cluster的磁盘中的偏移
       初始化为-1
     */
    int cluster_offset[KERNEL_BUF_SECTOR];
    /* 一个扇区的buffer */
    char cluster_buffer[KERNEL_BUF_SECTOR][ONE_SECTOR];
    /* 这个扇区打开后是否被写过
       初始化为0
     */
    int cluster_is_dirty[KERNEL_BUF_SECTOR];
    /* 指向下一个空的buffer */
    int next_free;
} Kernel_buf;
```

```
initialize  next_free = 0
     -----
0-> | F | - next_free(1) - next_free(4)
     -----
1-> | F | - next_free(2)
     -----
2-> | F | - next_free(3)
     -----
```

- **OS初始化, 几个数据结构：**
    - 保存所有fat表的地址

```
#define MAX_FAT_TABLE       (ONE_SECTOR / 4 - 1)
int fat_table_array[MAX_FAT_TABLE];
int fat_table_size;
```

- **OS初始化**

- 考虑到内存中会有两个用户程序，一个是接受输入，一个是我们要实现的应用程序，我们希望实现简单的页表实现j等使用绝对地址引用的指令，和.data段的地址取址。

```
#define MAX_PROCESS_NUM     2
/* 用于标示当前运行的程序 */
int cur_process = 0;

#define MAX_TEXT_PAGE_NUM           16
#define MAX_DATA_PAGE_NUM           4
#define USER_MAX_PAGE_NUM           (MAX_TEXT_PAGE_NUM \
                                    + MAX_DATA_PAGE_NUM)
/* 存放实际物理地址 */
int page_table[MAX_PROCESS_NUM][USER_MAX_PAGE_NUM];
```

大概的地址映射过程是这样的：

- 汇编的时候
  - .text起始地址0x0040_0000
  - .data起始地址0x1001_0000

- 传给MMU addr

```
int sign = addr & 0xffff0000;

if (sign == 0x00400000)

return ((page_table[cur_process][(addr >> 9)

- (TEXT_START >> 9)]) << 9 ) | (addr & 0x000001ff);

else if (sign == 0x10010000)

return ((page_table[cur_process][(addr >> 9)

- (TEXT_START >> 9) + MAX_TEXT_PAGE_NUM]) "<<" 9) | (addr &
0x000001ff);

else if (sign == 0xc000000)

; // 显存地址
```

```
#define OS_PAGE_NUM                 64
/* 所有的程序共享一个栈，即使是syscall调用也是，
   栈总共有4K，实际内存紧接着OS .data后
*/
#define GLOBAL_STACK_NUM            8

/* 因为只有1个应用程序加载，因此只需要这样两个变量 */
```

```
int old_free_page;
// 初始化空闲的page
int next_free_page = 1 + OS_PAGE_NUM + GLOBAL_STACK_NUM;
```

当执行一个文件，开始分配页的时候时：

```
old_free_page = next_free_page;
```

此后每分配一页的空间`next_free_page++;`

- **OS初始化**
    - 文件的概念

```
/* 如果打开的文件超过10就open失败 */
#define MAX_OPEN_FILE       10
typedef struct
{
    int free_file_count; // initialize with MAX_OPEN_FILE
    int is_free[MAX_OPEN_FILE];
    /* 根目录中的记录 */
    DirEntry file_entry[MAX_OPEN_FILE];
    /* 当前文件的读写位置，可以通过sys_lseek修改 */
    int fptr[MAX_OPEN_FILE];
    /* 该文件在根目录中的偏移，方便追加文件后修改文件大小 */
    int rootDirOffset[MAX_OPEN_FILE];
} DirEntry_M;
DirEntry_M dir_entry_m;
DirEntry curDirEntry;   // 保存根目录项的目录信息
```

- 文件操作
    - 创建文件(char *create_name)
        - 找到目录区的结束
        - 在fat表中找到一个空闲的簇k，并修改为0x0,设置is_dirty
        - 设置文件名，大小为0，起始簇为k
        - 设置is_dirty
        - 修改curDirEntry的大小

- 文件操作
    - 打开文件(char *file_name)
        - 确定打开文件个数在`MAX_OPEN_FILE`以内
        - 遍历根目录区，找到该项
        - 遍历文件标示符的数组，找到一个`free`项`i`
        - 修改`free_file_count--; file_entry`为找到的项; `fptr`初始为`0`;

is_free为0, rootDirOffset为找到的目录偏移
- 返回i为文件标示符

- 文件操作
  - seek(int fid, int offset, FILE_POS file_pos)

    typedef enum { M_SEEK_SETS, M_SEEK_CUR, } FILE_POS; * 直接修改fptr即可

- 文件操作
  - 读文件(int fid, char *buf, int size)
    - 这个buf是用户开的缓冲区，
    - 根据fptr找到开始簇，读到系统buffer，然后直接写到buf中， 因为此时的cur_process还是不变的,不需要做映射。同时递增fptr
    - 根据读的大小重复上述步骤

- 文件操作
  - 写文件(int fid, char *buf, int size)
    - 主要步骤与read类似
    - 不同的是，如果超过原来的文件大小，需要分配空间（修改fat，与create中拿到cluster类似） 同时修改文件大小

- 文件操作
  - 关闭文件(int fid)
    - 修改fid的is_free为0; free_file_count++
    - 注意此时不会进行写回磁盘的操作，只有在dirty的时候才会写回
    - 因此系统退出时有一个系统调用为force_write_back()

- 文件操作
  - 执行文件(char *file_name)
    - 打开文件
    - 读可执行文件的头，我们的可执行文件头：

```
0-3: text_seg start_offset(measure by bytes)
4-7: text_seg size(no more than 8k)
8-11: data_seg start_offset
12-15: data_seg size(no more than 2k)
16-19: debug_start_offset
20-23: debug size
```

```
*    开始分配页，并保存原来的`free_page`起始`old_free_page = next_free_page;
*    把文件内容读到新分配的页中
*    `cur_process++`; 执行`jal 0x0040_0000`
```

- 文件操作
  - 退出执行
    - `next_free_page = old_free_page;`
    - `cur_process--;`
    - `jr $ra` //此时$ra的物理地址能够被找到

- *NOTE:*
  - 我们所有的系统调用与函数调用类似，直接使用$ra, 并且$ra入栈(也可以不使用$ra)

谢谢