计算机与操作系统实验二-实验报告

201250172 熊丘桓

1. 实验代码

1.1 main.cpp

```cpp
#include <stdint.h>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <regex>
#include <vector>

using namespace std;

// 参考代码：软件学院2019级-王一辉
#pragma pack(1)  // 指定按 1 字节对齐，否则 struct 和 class 的成员变量按 4 字节对齐

const unsigned BYTES_PER_ENTRY = 32;
const int MAX_CONTENT_SIZE = 10000;

const int FINAL_TAG = 0xFF8;  // FAT 表的坏簇标记

const char* COMMAND_ERROR = "ERROR: Cannot parse this command!\n";
const char* NO_FILE_ERROR = "ERROR: No such file!\n";
const char* FILE_TYPE_ERROR = "ERROR: File type error!!\n";
const char* EXIT_MESSAGE = "Bye!\n";
const string PATH_SEPARATOR = "/";

extern "C" {
void my_print(const char*, int);
void print(const char*);
void print_red(const char*);
}

inline void Print(const char* s, bool red = false) {
    const char* colored = red ? ("\033[31m" + string(s) + "\033[37m").c_str() : s;
#ifdef EagleBear
    printf("%s", colored);
#else
```

```
    // my_print(colored, strlen(colored));
    if (red) {
        print_red(s);
    } else {
        print(s);
    }
#endif
}

inline void readFromFile(void* dist, size_t size, FILE* fat12, uint32_t
pos) {
    fseek(fat12, pos, SEEK_SET);
    fread(dist, 1, size, fat12);
}

// 参考代码：https://blog.csdn.net/Mary19920410/article/details/77372828
vector<string> split(const string& str, const string& delim) {
    vector<string> res;
    if (str.empty())
        return res;

    char strs[str.length() + 1];
    strcpy(strs, str.c_str());

    char d[delim.length() + 1];
    strcpy(d, delim.c_str());

    char* p = strtok(strs, d);
    while (p) {
        string s = p;
        res.push_back(s);  //存入结果数组
        p = strtok(nullptr, d);
    }

    return res;
}

uint16_t bytesPerSector;     // 每扇区字节数，一般为 512
uint8_t sectorsPerCluster;   // 每簇扇区数，一般为 1
uint16_t reservedSectors;    // Boot 记录占用的扇区数
uint8_t FATCount;            // FAT 表个数，一般为 2
uint16_t directoryEntries;   // 根目录最大文件数
uint32_t sectorsPerFAT;      // FAT 扇区数
uint32_t FATBase;            // FAT1 的偏移量
uint32_t rootDirectoryBase;  // 根目录的偏移量
uint32_t dataBase;           // 数据区偏移量
uint32_t bytesPerCluster;    // 每簇字节数 = 每扇区字节数 * 每簇扇区数

```

```
82  inline int getFATValue(FILE* fat12, int num) {
83      uint32_t pos = FATBase + num / 2 * 3;
84      int FATValue = 0;
85      readFromFile(&FATValue, 3, fat12, pos);
86
87      if (num % 2) {  // pos id odd
88          FATValue = FATValue >> 12;
89      } else {  // pos is even
90          FATValue = FATValue & 0xfff;
91      }
92      return FATValue;
93  }
94
95  class BPB {
96     private:
97      uint16_t BPB_BytesPerSector;    // 每扇区字节数，一般为 512
98      uint8_t BPB_SectorsPerCluster;  // 每簇扇区数，一般为 1
99      uint16_t BPB_ReservedSectors;   // Boot record 占用的扇区数
100     uint8_t BPB_FATCount;           // FAT 表个数，一般为 2
101     uint16_t BPB_DirectoryEntries;  // 根目录文件数的最大值
102     uint16_t BPB_TotalSectors;      // 扇区数
103     uint8_t BPB_MediaDescriptor;    //
104     uint16_t BPB_SectorsPerFAT;     // FAT 扇区数，扇区数大于 65535 时该
    值为 0
105     uint16_t BPB_SectorsPerTrack;
106     uint16_t BPB_Heads;
107     uint32_t BPB_HiddenSectors;
108     uint32_t BPB_LargerSectorCount;  // 如果 BPB_SectorsPerFAT 为 0，该
    值为 FAT 扇区数
109
110    public:
111     BPB(FILE* fat12) {
112         readFromFile(this, 25, fat12, 11);
113
114         bytesPerSector = BPB_BytesPerSector;
115         sectorsPerCluster = BPB_SectorsPerCluster;
116         reservedSectors = BPB_ReservedSectors;
117         FATCount = BPB_FATCount;
118         directoryEntries = BPB_DirectoryEntries;
119
120         if (BPB_SectorsPerFAT != 0) {
121             sectorsPerFAT = BPB_SectorsPerFAT;
122         } else {
123             sectorsPerFAT = BPB_LargerSectorCount;
124         }
125
126         FATBase = reservedSectors * bytesPerSector;
```

```
127         rootDirectoryBase = FATBase + sectorsPerFAT * FATCount *
    bytesPerSector;
128         dataBase = rootDirectoryBase + (directoryEntries *
    BYTES_PER_ENTRY + bytesPerSector - 1) / bytesPerSector *
    bytesPerSector;  // dataBase = fileRootBase + ceil(1.0 *
    directoryEntries * BYTES_PER_ENTRY /bytesPerSector) * bytesPerSector;
129         bytesPerCluster = sectorsPerCluster * bytesPerSector;
130     };
131 };  // end of BPB
132
133 enum NodeType {
134     FILE_TYPE,
135     DIRECTORY_TYPE,
136     VIRTUAL,
137 };
138
139 struct Node {
140     const string name;
141     const string path;
142     size_t fileSize;
143     vector<Node*> children;
144     NodeType type;
145     int directoryCount = 0;
146     int fileCount = 0;
147     char content[MAX_CONTENT_SIZE]{};
148
149     void list(const bool listSize) const {
150         char tmp[522];
151         if (listSize) {
152             sprintf(tmp, "%s %d %d:\n", path.c_str(), directoryCount,
    fileCount);
153         } else {
154             sprintf(tmp, "%s:\n", path.c_str());
155         }
156
157         Print(tmp);
158
159         for (Node* child : children) {
160             if (child->type == VIRTUAL) {
161                 Print(child->name.c_str(), true);
162                 Print("  ");
163             } else if (child->type == FILE_TYPE) {
164                 if (listSize) {
165                     sprintf(tmp, "%s %d", child->name.c_str(), child-
    >fileSize);
166                 } else {
167                     sprintf(tmp, "%s  ", child->name.c_str());
168                 }
```

```
                Print(tmp);
            } else {  // DIRECTORY
                Print(child->name.c_str(), true);
                if (listSize) {
                    sprintf(tmp, " %d %d", child->directoryCount,
child->fileCount);
                } else {
                    sprintf(tmp, "  ");
                }
                Print(tmp);
            }

            if (listSize)
                Print("\n");
        }
        Print("\n");

        for (Node* child : children) {
            if (child->type == DIRECTORY_TYPE) {
                child->list(listSize);
            }
        }
    }

    Node(string name, string path, NodeType type, size_t fileSize)
        : name(name), path(path), type(type), fileSize(fileSize) {}

    inline void addChild(Node* child) {
        if (child->type == DIRECTORY_TYPE) {
            child->addChild(new Node(".", "", VIRTUAL, 0));
            child->addChild(new Node("..", "", VIRTUAL, 0));
            this->directoryCount++;
        } else if (child->type == FILE_TYPE) {
            this->fileCount++;
        }
        this->children.push_back(child);
    }

    string formatPath(string targetPath) const {
        vector<string> names = split(targetPath, PATH_SEPARATOR);
        stack<string> st;
        for (string name : names) {
            if (name == ".") {
                continue;
            } else if (name == "..") {
                if (!st.empty())
                    st.pop();
            } else {
```

```
                st.push(name);
            }
        }

        string res;
        while (!st.empty()) {
            res = PATH_SEPARATOR + st.top() + res;
            st.pop();
        }

        return res;
    }

    const Node* findNode(string targetPath) const {
        targetPath = formatPath(targetPath);

        // printf("targetPath = %s\n", targetPath.c_str());
        // printf("currentPath = %s\n", path.c_str());
        if (this->type == FILE_TYPE && targetPath == this->path + this->name) {
            // printf("find: %s\n", this->path.c_str());
            return this;
        }

        if (this->type == DIRECTORY_TYPE && this->path == targetPath + PATH_SEPARATOR) {
            return this;
        }

        if (targetPath.find(path) != 0) {
            return nullptr;
        }

        for (Node* child : children) {
            const Node* res = child->findNode(targetPath);
            if (res != nullptr) {
                return res;
            }
        }

        return nullptr;
    }

    void readContent(FILE* fat12, int startCluster);

    void readChildren(FILE* fat12, int startCluster);
};  // end of Node

```

```
struct DirectoryEntry {
    static const int NAME_LENGTH = 11;
    static const int DIRECTORY = 0x10;

    char fileName[NAME_LENGTH];  // "xxxxxxxxyyy" as xxxxxxxx.yyy, where xxxxxxxx is filename and yyy is extention
    uint8_t fileAttributes;
    char reserved[14];
    uint16_t firstCluster_low;
    uint32_t fileSize;

    DirectoryEntry() = default;

    inline bool invalidName() const {
        if (fileName[0] == '\0')
            return true;

        for (char ch : fileName)
            if (ch != ' ' && !isalnum(ch))
                return true;
        return false;
    }

    void initRootEntry(FILE* fat12, Node* root) {
        uint32_t base = rootDirectoryBase;

        for (int i = 0; i < directoryEntries; ++i) {
            readFromFile(this, BYTES_PER_ENTRY, fat12, base);
            base += BYTES_PER_ENTRY;
            if (this->invalidName())
                continue;

            string realName = this->transferName();
            Node* child;
            if (this->isFile()) {
                child = new Node(realName, root->path, FILE_TYPE, fileSize);
                root->addChild(child);
                child->readContent(fat12, firstCluster_low);
            } else {
                child = new Node(realName, root->path + realName + PATH_SEPARATOR, DIRECTORY_TYPE, 0);
                root->addChild(child);
                child->readChildren(fat12, firstCluster_low);
            }
        }
    }

```

```
307     inline bool isFile() const {
308         return (fileAttributes & DIRECTORY) == 0;
309     }
310 
311     string transferName() const {
312         string res;
313         for (int i = 0; i < NAME_LENGTH; ++i) {
314             if (i == 8 && this->isFile())
315                 res += '.';
316             if (fileName[i] != ' ')
317                 res += fileName[i];
318         }
319         return res;
320     }
321 
322     size_t getFileSize() const {
323         return fileSize;
324     }
325 };  // end of DirectoryEntry
326 
327 void Node::readContent(FILE* fat12, int startCluster) {
328     if (startCluster == 0)
329         return;
330 
331     char* pointer = this->content;
332     for (int currentCluster = startCluster; currentCluster < FINAL_TAG;
    currentCluster = getFATValue(fat12, currentCluster)) {
333         char tmp[bytesPerCluster];
334         uint32_t startByte = dataBase + (currentCluster - 2) *
    sectorsPerCluster * bytesPerSector;
335         readFromFile(tmp, bytesPerCluster, fat12, startByte);
336         memcpy(pointer, tmp, bytesPerCluster);
337         pointer += bytesPerCluster;
338     }
339 }
340 
341 void Node::readChildren(FILE* fat12, int startCluster) {
342     for (int currentCluster = startCluster; currentCluster < FINAL_TAG;
    currentCluster = getFATValue(fat12, currentCluster)) {
343         uint32_t startByte = dataBase + (currentCluster - 2) *
    sectorsPerCluster * bytesPerSector;  // 数据区的第一个簇的簇号是 2
344         for (int i = 0; i < bytesPerCluster; i += BYTES_PER_ENTRY) {
345             DirectoryEntry* rootEntry = new DirectoryEntry();
346             readFromFile(rootEntry, BYTES_PER_ENTRY, fat12, startByte +
    i);
347 
348             if (rootEntry->invalidName()) {
349                 continue;
```

```
            }

            string realName = rootEntry->transferName();
            if (rootEntry->isFile()) {
                Node* child = new Node(realName, this->path, FILE_TYPE,
rootEntry->getFileSize());
                addChild(child);
                child->readContent(fat12, rootEntry->firstCluster_low);
            } else {
                Node* child = new Node(realName, this->path + realName
+ PATH_SEPARATOR, DIRECTORY_TYPE, 0);
                addChild(child);
                child->readChildren(fat12, rootEntry-
>firstCluster_low);
            }
        }
    }
}

const char* handleCat(vector<string>& commands, const Node* root) {
    if (commands.size() != 2) {
        return COMMAND_ERROR;
    }

    string& path = commands[1];
    const Node* res = root->findNode(path);
    if (res == nullptr) {
        return NO_FILE_ERROR;
    }
    if (res->type != FILE_TYPE) {
        return FILE_TYPE_ERROR;
    }
    return res->content;
}

const char* handleList(vector<string>& commands, Node* root) {
    bool listSize = false, pathSet = false;
    const Node* directory = root;
    for (int i = 1; i < commands.size(); i++) {
        string& arg = commands[i];
        if (regex_match(arg.c_str(), regex("-l+"))) {
            listSize = true;
        } else if (!pathSet) {
            pathSet = true;
            directory = root->findNode(arg);
        } else {
            return COMMAND_ERROR;
        }
```

```
    }

    if (directory == nullptr)
        return NO_FILE_ERROR;
    if (directory->type != DIRECTORY_TYPE)
        return FILE_TYPE_ERROR;

    directory->list(listSize);
    return "";
}

int main() {
#ifdef EagleBear
    freopen("test.in", "r", stdin);
    freopen("test.out", "w", stdout);
#endif

    FILE* fat12 = fopen("./a2.img", "rb");  // 打开 FAT12 的映像文件
    BPB bpb(fat12);
    Node* root = new Node("", "/", DIRECTORY_TYPE, 0);
    DirectoryEntry* rootEntry = new DirectoryEntry();
    rootEntry->initRootEntry(fat12, root);

    while (true) {
        Print("> ");
        string inputLine;
        getline(cin, inputLine);
#ifdef EagleBear
        Print(inputLine.c_str());
        Print("\n");
#endif
        vector<string> commandLine = split(inputLine, " ");

        const char* res = COMMAND_ERROR;
        if (commandLine.size() == 0) {
            res = "";
        } else if (commandLine[0] == "exit") {
            if (commandLine.size() == 1) {
                Print(EXIT_MESSAGE);
                break;
            }
        } else if (commandLine[0] == "ls") {
            res = handleList(commandLine, root);
        } else if (commandLine[0] == "cat") {
            res = handleCat(commandLine, root);
        }
        Print(res);
    }
```

```
443
444     fclose(fat12);
445
446 #ifdef EagleBear
447     fclose(stdin), fclose(stdout);
448 #endif
449     return 0;
450 }
451
```

1.2 my_print.asm

```
1  ; 参考代码: https://blog.csdn.net/bedisdover/article/details/51287555
2  global print
3  global print_red
4  global my_print
5
6  section .data
7  color_red   db 1Bh,'[31m',0
8  .len        equ $-color_red
9  color_default   db 1Bh,'[37m',0
10 .len        equ $-color_default
11
12 section .text
13
14 print:
15     mov eax, 4
16     mov ebx, 1
17     mov ecx, color_default
18     mov edx, color_default.len
19     int 80h
20
21     mov ecx, [esp+4] ; ecx = str
22     mov eax, [esp+4]
23     call strlen
24     mov edx, eax ; edx = len
25     mov eax, 4
26     mov ebx, 1
27     int 80h
28
29     ret
30
31 print_red:
32     mov eax, 4
33     mov ebx, 1
34     mov ecx, color_red
35     mov edx, color_red.len
36     int 80h
```

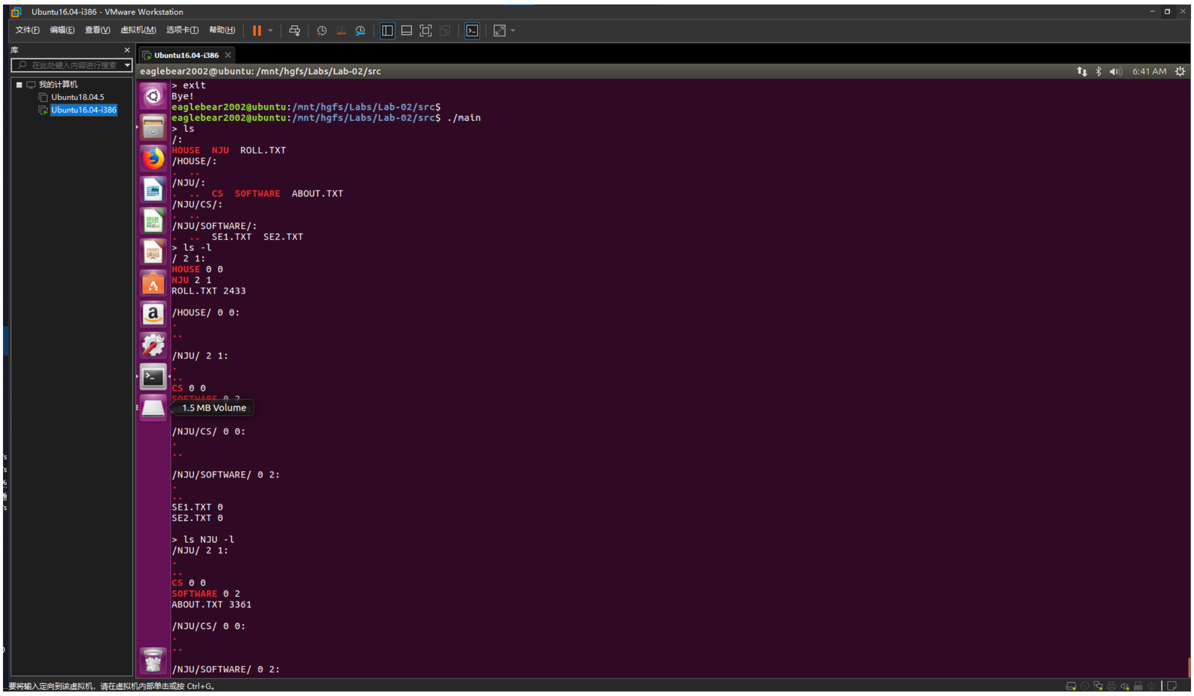
```
37
38      mov ecx, [esp+4]
39      mov eax, [esp+4]
40      call strlen
41      mov edx, eax
42      mov eax, 4
43      mov ebx, 1
44      int 80h
45
46      mov eax, 4
47      mov ebx, 1
48      mov ecx, color_default
49      mov edx, color_default.len
50      int 80h
51
52      ret
53
54  ;; strlen(str: eax) -> len: eax
55  strlen:
56      push ebx
57      mov ebx, eax
58
59      .nextchar:
60          cmp byte [eax], 0
61          jz .finished
62          inc eax
63          jmp .nextchar
64
65      .finished:
66          sub eax, ebx
67          pop ebx
68          ret
69
70  my_print:
71      push    ebp
72      mov     edx, [esp+12]
73      mov     ecx, [esp+8]
74      mov     ebx, 1
75      mov     eax, 4
76      int     80h
77      pop     ebp
78      ret
```

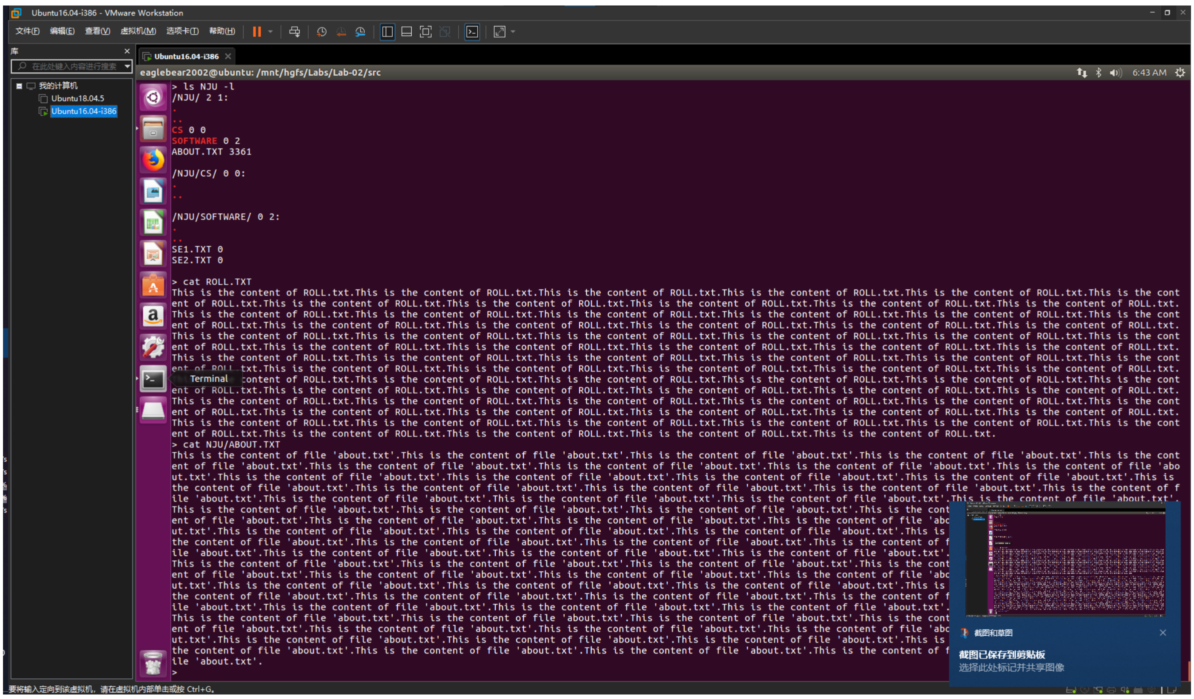
1.3 makefile

```makefile
Main: main.cpp my_print.asm
	nasm -felf32 my_print.asm -o my_print.o
	g++ main.cpp my_print.o -o main -std=c++11
	rm -rf my_print.o
clean:
	rm -rf main
```

2. 实验截图

该实验实现了附加功能： `cat` 命令支持输出超过 512 字节的文件。





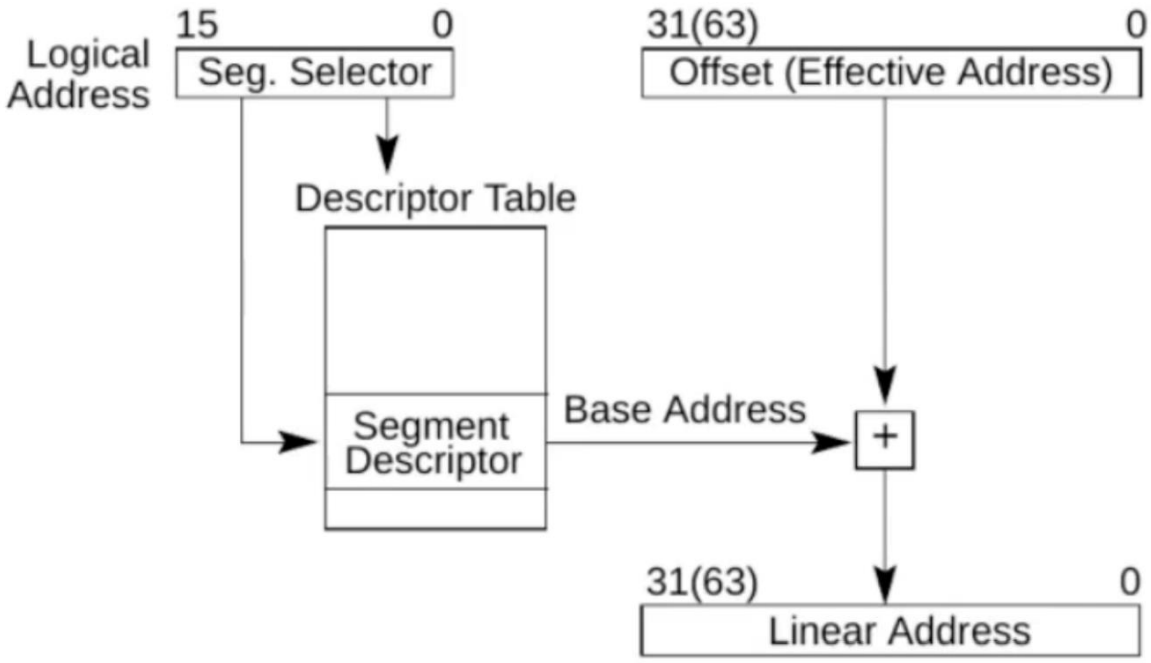
3. 实验困难

1. 使用 `#pragma pack(1)` 指令，编译时对成员变量按 1 字节对齐，而不是按 4 字节对齐。
2. 在汇编中实现输出红色字体，避免在 C++ 代码中实现输出红色字体带来的错误

4. 实验问题

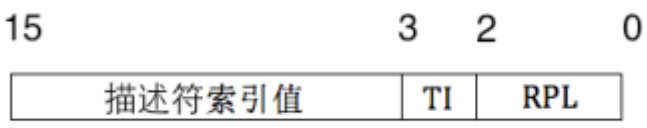
4.1 什么是实模式，什么是保护模式?

- 实模式：基地址+偏移量可以直接获得物理地址的模式。缺点：非常不安全
- 保护模式：不能直接拿到物理地址，需要进行地址转换。从 80286 开始，是现代操作系统的主要模式



4.2 什么是选择子?

- 选择子共 16 位，放在段选择寄存器里
- 低 2 位表示请求特权级
- 第 3 位表示选择 GDT 还是 LDT 方式
- 高 13 位表示在描述符表中的偏移
- 故描述符表的项数最多是 2^{13}



4.3 什么是描述符?

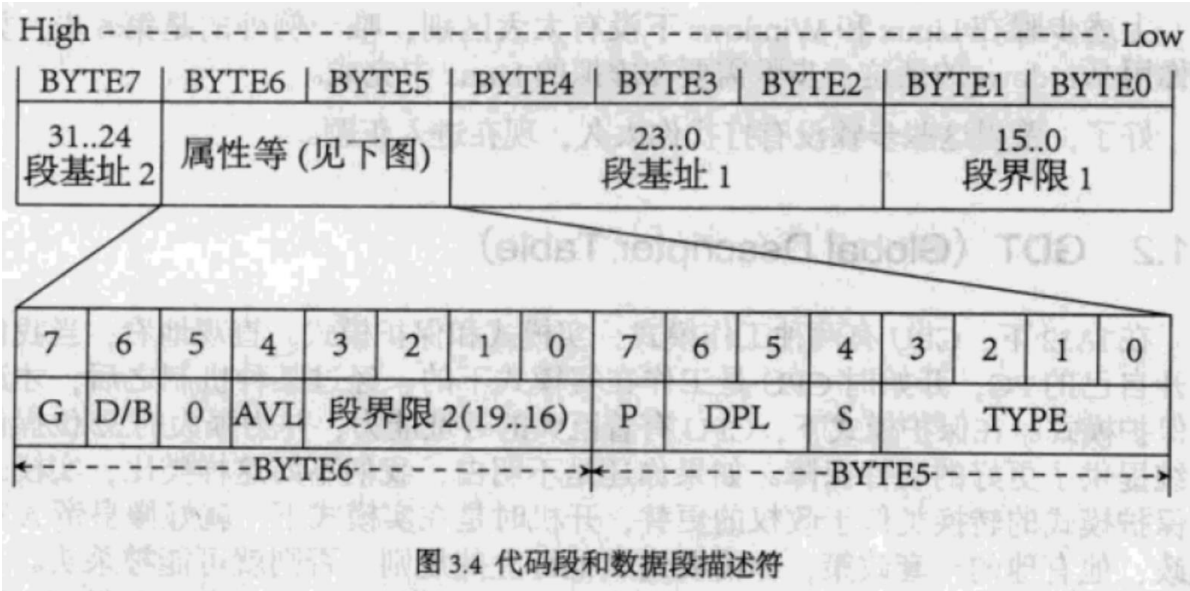


图 3.4 代码段和数据段描述符

4.4 什么是 GDT，什么是 LDT?

GDT：全局描述符表，是全局唯一的。存放一些公用的描述符，和包含各进程局部描述符表首地址的描述符。

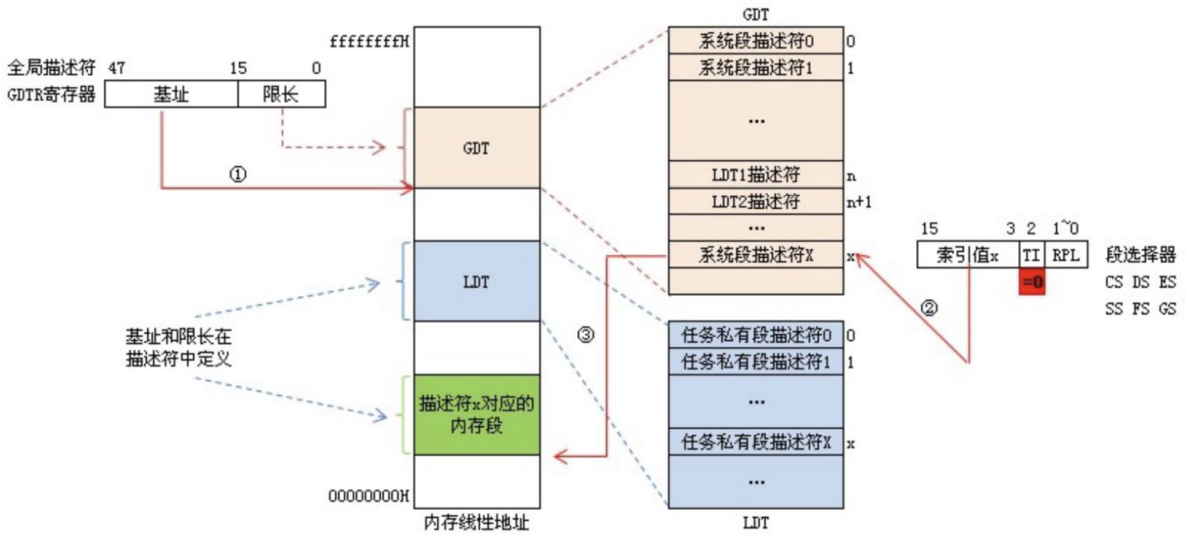
LDT：局部描述符表，每个进程都可以有一个。存放本进程内使用的描述符。

4.5 请分别说明 GDTR 和 LDTR 的结构

GDTR：48 位寄存器，高 32 位放置 GDT 首地址，低 16 位放置 GDT 限长。限长决定了可寻址的大小，注意低 16 位放的不是选择子

LDTR：16 位寄存器，放置一个特殊的选择子，用于查找当前进程的 LDT 首地址。

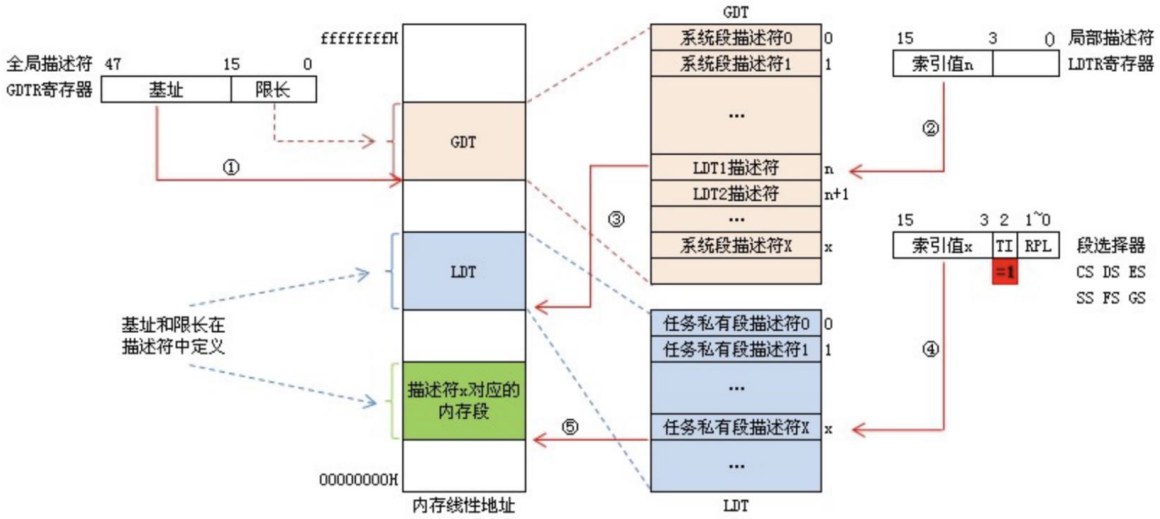
4.6 请说明 GDT 直接查找物理地址的具体步骤



1. 给出段选择子（放在段选择寄存器里）+ 偏移量

2. 若选择了 GDT 方式，则从 GDTR 获取 GDT 首地址，用段选择子中的 13 位做偏移，拿到 GDT 中的描述符
3. 如果合法且有权限，用描述符中的段首地址加上第 1 步中的偏移量找到物理地址，寻址结束

4.7 请说明通过 LDT 查找物理地址的具体步骤



1. 给出段选择子（放在段选择寄存器中）+ 偏移量
2. 若选择了 LDT 方式，则从 GDTR 获取 GDT 首地址，用 LDTR 中的偏移量做偏移，拿到 GDT 中的描述符 1
3. 从描述符 1 中获取 LDT 首地址，用段选择子中的 13 位做偏移，拿到 LDT 中的描述符 2
4. 如果合法且有权限，用描述符 2 中的段首地址加上第 1 步中的偏移量找到物理地址。寻址结束

4.8 根目录区大小一定么？扇区号是多少？为什么？

根目录区长度不是一定的，需要根据 BPB 中规定的根目录最大文件数计算。

下面代码求出根目录区的起始字节数。

```
FATBase = reservedSectors * bytesPerSector;
rootDirectoryBase = FATBase + sectorsPerFAT * FATCount * bytesPerSector;
```

一般来说，0 扇区为引导扇区，每个 FAT 表占用 9 个扇区，则根目录区从 19 扇区开始。

4.9 数据区第一个簇号是多少？为什么？

下面代码求出数据区的起始字节数。

```
dataBase = rootDirectoryBase + (directoryEntries * BYTES_PER_ENTRY + bytesPerSector - 1) / bytesPerSector * bytesPerSector;  // dataBase = fileRootBase + ceil(1.0 * directoryEntries * BYTES_PER_ENTRY /bytesPerSector) * bytesPerSector;
```

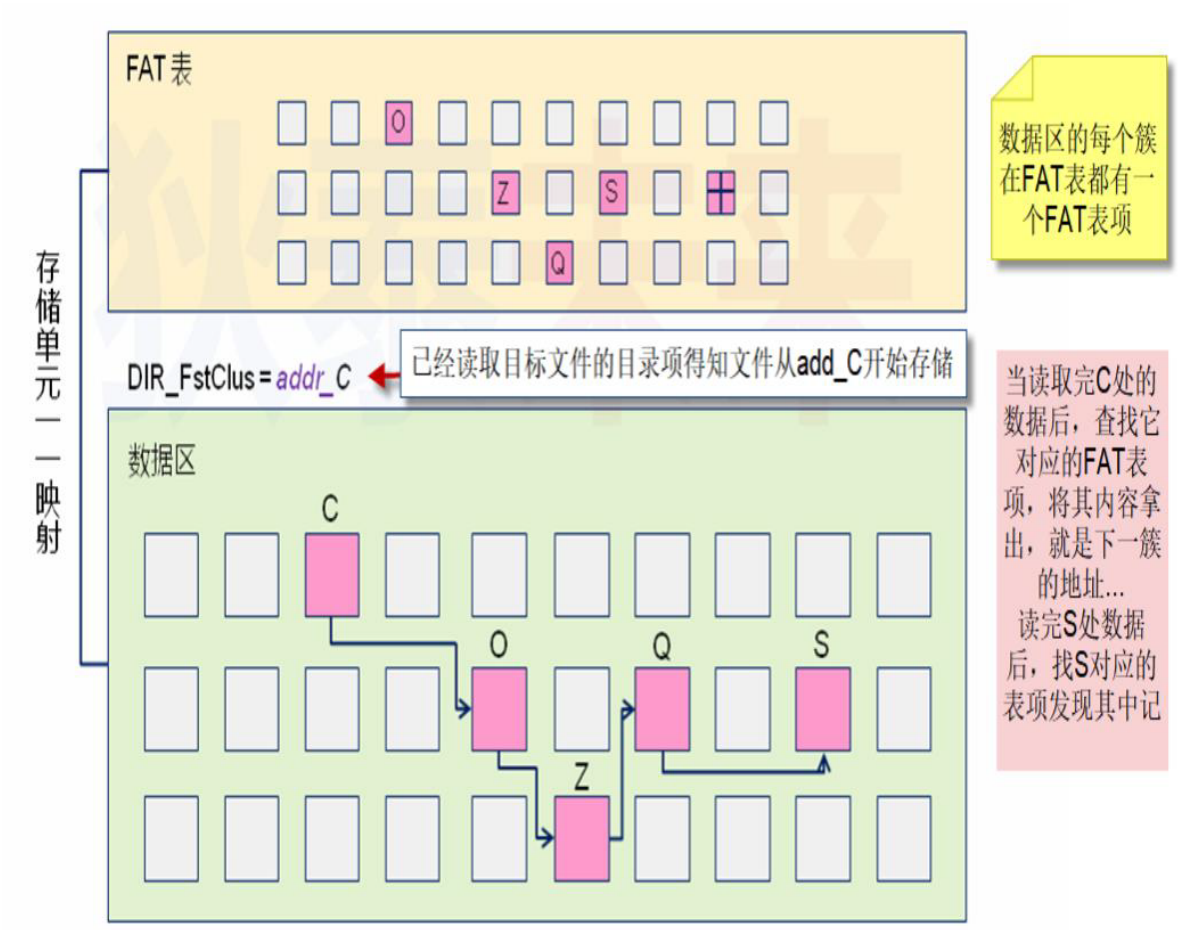
4.10 FAT 表的作用？

文件分配表被划分为紧密排列的若干个表项，每个表项都与**数据区**中的一个簇相对应，而且表项的序号也是与簇号一一对应的。

每 12 位成为一个 FAT 项（FATEntry），代表一个簇。所以 2 个 FAT 项会占用 3 个字节。

FAT 项的值代表文件的下一个簇号：

- 值大于或等于 `0xFF8`，表示当前簇已经是本文件的最后一个簇
- 值为 `0xFF7`，表示它是一个坏簇



4.11 解释静态链接的过程

静态链接是指在编译阶段直接把静态库加入到可执行文件中去，这样可执行文件会比较大。

静态链接时发生空间和地址分配，符号解析和重定位。

4.12 解释动态链接的过程

动态链接是指链接阶段仅仅只加入一些描述信息，而程序执行时再从系统中把相应动态库加载到内存中去。

动态链接分为装载时动态链接和运行时动态链接。

4.13 静态链接相关 PPT 中为什么使用 `ld` 链接而不是 `gcc`？

ld 是 gcc 工具链的一部分。

4.14 linux 下可执行文件的虚拟地址空间默认从哪里开始分配

从 `0x08048000` 开始分配。

> 在 386 系统上，文本基地址为 0x08048000，这允许文本下方有一个相当大的堆栈，同时仍保持在地址 0x08000000 上方，从而允许大多数程序使用单个二级页表。 （回想一下，在 386 上，每个二级表映射 0x00400000 个地址。）

4.15 BPB 指定字段的含义

```
class BPB {
   private:
    uint16_t BPB_BytesPerSector;    // 每扇区字节数，一般为 512
    uint8_t BPB_SectorsPerCluster;  // 每簇扇区数，一般为 1
    uint16_t BPB_ReservedSectors;   // Boot record 占用的扇区数
    uint8_t BPB_FATCount;           // FAT 表个数，一般为 2
    uint16_t BPB_DirectoryEntries;  // 根目录文件数的最大值
    uint16_t BPB_TotalSectors;      // 扇区数
    uint8_t BPB_MediaDescriptor;    // Media Descriptor 的种类
    uint16_t BPB_SectorsPerFAT;     // FAT 扇区数，扇区数大于 65535 时该值为 0
    uint16_t BPB_SectorsPerTrack;
    uint16_t BPB_Heads; // Number of heads or sides on the storage media.
    uint32_t BPB_HiddenSectors; // Number of hidden sectors.
    uint32_t BPB_LargerSectorCount;  // 如果 BPB_SectorsPerFAT 为 0，该值为 FAT 扇区数
```

```
15
16      ...
17  };
```

4.16 如何进入子目录并输出（说明方法调用）

在初始化时已经利用 `class Node` 构建了逻辑上的文件树。在处理命令时直接对构建好的文件树进行遍历，不再访问二进制文件。

4.17 如何获得指定文件的内容，即如何获得数据区的内容（比如使用指针等）

在初始化时已经利用 `class Node` 构建了逻辑上的文件树。在处理命令时直接对构建好的文件树进行遍历，不再访问二进制文件。

`findNode` 成员函数为根据目标路径，对当前节点及其子节点进行查找。

4.18 如何进行 C 代码和汇编之间的参数传递和返回值传递

使用 `[esp+4]` 获取函数的唯一一个参数。函数没有返回值。

4.19 汇编代码中对 I/O 的处理方式，说明指定寄存器所存值的含义

使用基本同实验一的方式进行 I/O，与输出颜色相关的代码如下：

```
1  section .data
2  color_red   db 1Bh,'[31m',0
3  .len        equ $-color_red
4  color_default   db 1Bh,'[37m',0
5  .len        equ $-color_default
```