

# 第9课 程序的加载与执行

《跟着瓦利哥学写OS》



# 联系方式

- 自己动手写操作系统QQ群：82616767。
- 申请考试邮箱：sangwf@gmail.com
- 所有课程的代码都在Github：
  - <https://github.com/sangwf/walleclass>



# 思考

- 操作系统如何执行应用程序的？如双击一个应用，到底发生了什么？



2014/9/11 Android APP可以在Chrome OS上运行





# 可执行程序

- 一种特殊格式的文件，里面包含机器指令，可以被操作系统加载执行。



# 常见可执行文件的格式

- ELF — Executable and Linkable Format, Linux/Unix的默认格式。
- PE — Portable Execute, Windows的默认格式。
- Macho — Mac OS的默认格式。



# ELF格式

Linking View

ELF header
Program header table <i>optional</i>
Section 1
...
Section $n$
...
...
Section header table

Execution View

ELF header
Program header table
Segment 1
Segment 2
...
Section header table <i>optional</i>



# ELF Header

```
typedef struct {  
    unsigned char    e_ident[EI_NIDENT];  
    Elf32_Half       e_type;  
    Elf32_Half       e_machine;  
    Elf32_Word       e_version;  
    Elf32_Addr       e_entry;  
    Elf32_Off        e_phoff;  
    Elf32_Off        e_shoff;  
    Elf32_Word       e_flags;  
    Elf32_Half       e_ehsize;  
    Elf32_Half       e_phentsize;  
    Elf32_Half       e_phnum;  
    Elf32_Half       e_shentsize;  
    Elf32_Half       e_shnum;  
    Elf32_Half       e_shstrndx;  
} Elf32_Ehdr;
```



# ELF Header

- `e_ident` : ELF文件标识, 以`0x7f`开头。
- `e_type` : 文件具体类型, 可重定向文件/可执行文件/Core文件等。
- `e_entry` : 可执行程序执行时的入口地址。



# 可重定向文件

- 代码的绝对地址在编译时无法确定，需要在链接时确定和调整。



# 实验1：认识ELF文件

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[])
4 {
5     printf("Hello, world!\n");
6
7     return 0;
8 }
```



# 实验1：认识ELF文件

- `$ i586-pc-linux-gcc helloworld.c`
- `$ i586-pc-linux-readelf -h a.out`
- `$ i586-pc-linux-objdump -d a.out`



# 实验1：认识ELF文件

## ELF Header:

Magic:	7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:	ELF32
Data:	2's complement, little endian
Version:	1 (current)
OS/ABI:	UNIX - System V
ABI Version:	0
Type:	EXEC (Executable file)
Machine:	Intel 80386
Version:	0x1
Entry point address:	0x80482f0
Start of program headers:	52 (bytes into file)
Start of section headers:	3776 (bytes into file)
Flags:	0x0



# 实验1：认识ELF文件

```
a.out:      file format elf32-i386
```

```
Disassembly of section .init:
```

```
08048274 <_init>:
```

8048274:	55	push	%ebp
8048275:	89 e5	mov	%esp,%ebp
8048277:	53	push	%ebx
8048278:	83 ec 04	sub	\$0x4,%esp
804827b:	e8 00 00 00 00	call	8048280 <_init+0xc>
8048280:	5b	pop	%ebx



# 回顾bootloader的加载

- 读取第0号扇区，加载到0x7C00地址，然后从0x7C00执行。



# 程序的加载

File Offset	File	Virtual Address
Program header table	0 ELF header	
	Other information	
0x100	Text segment ... 0x2be00 bytes	0x8048100  0x8073eff
0x2bf00	Data segment ... 0x4e00 bytes	0x8074f00  0x8079cff
0x30d00	Other information ...	



# 程序的加载步骤 (`exec`系统调用)

- ① 1, 读取ELF Header;
- ② 2, 读取各个段并放到内存指定位置 (或设置好页表映射);
- ③ 3, 修改栈上的函数返回地址EIP为`e_entry`;
- ④ 4, 中断返回后, 即从`e_entry`开始执行。



# 实验2：exec系统调用

- 实现一个简单的exec系统调用
- 如果传入文件名为“print\_hello”，则执行print\_hello程序，否则，执行print\_world程序。
- print\_hello/print\_world均为Binary格式而非ELF格式，简化实现。



# 代码1

```
1 #include "system.h"
2
3 int my_entrance()
4 {
5     if (fork_syscall() == 0) {
6         // son
7         exec("print_hello");
8         while(1) ;
9     } else {
10        // dad
11        print_char('D');
12        while(1) ;
13    }
14
15    return 0;
16 }
```



# 代码2

```
29 int exec(char *filename)
30 {
31     int ret = -1;
32     if (strncmp(filename, "print_hello", 11) == 0) {
33         ret = exec_syscall(0x2600, 0x200); // print_hello存放在0x2600处
34     } else {
35         ret = exec_syscall(0x2A00, 0x200); // print_world存放在0x2A00处
36     }
37
38     return ret;
39 }
```



# 代码3：没有库函数可用了！

```
19 int strncmp(char *str1, char *str2, int size)
20 {
21     while (--size && *str1 && *str1 == *str2) {
22         str1++;
23         str2++;
24     }
25
26     return (*str1 - *str2);
27 }
```



# 代码4

```
11 int exec_syscall(int file_offset, int file_size)
12 {
13     int ret = -1;
14     __asm__ volatile ("int $0x83": "=a" (ret): "a"(file_offset), "c"(file_size));
15
16     return ret;
17 }
```



# 代码5

```
250 int_exec: ;eax是起始内存地址, ecx是文件大小
251     cli ;关闭中断, 避免中间被打断
252     push ebp
253     mov ebp, esp
254     push es
255     push ds
256     push ebx
257
258     mov ebx, DATA_SEL
259     mov ds, bx
260     mov es, bx
261
```

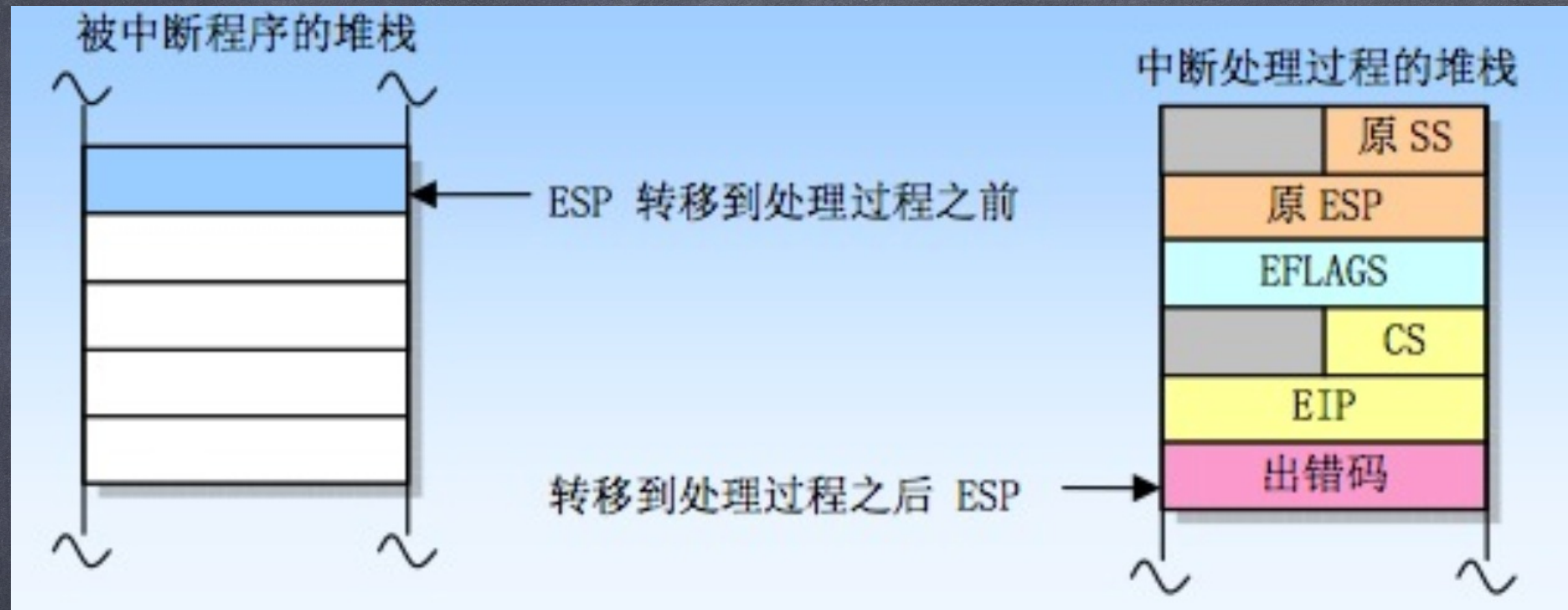


# 代码6

```
262      ;复制应用程序到 0x2500
263      lea si, [eax]
264      lea di, [0x2500]
265      rep movsb
266
267      mov dword [ss: ebp + 4], 0x2500 ;修改 eip
268
269      pop ebx
270      pop ds
271      pop es
272      leave
273      sti
274      iret
```



# EIP在内核栈上的位置





# 代码7

```
1 #include "system.h"
2
3 int my_entrance()
4 {
5     print_string("hello");
6     while(1) ;
7
8     return 0;
9 }
```

```
1 #include "system.h"
2
3 int my_entrance()
4 {
5     print_string("world");
6     while(1) ;
7
8     return 0;
9 }
```



# 代码8

```
11 #define print_string(string) ({ \
12     __asm__ volatile ("int $0x84"::"d" (string) ); \
13 })
```

```
277 int_print_string:
278     call func_write_string
279     iret
```



# 代码9

```
.mark_ps_while:  
cmp byte [edx], 0  
je .mark_ps_ret  
mov al, [edx]  
call func_write_char  
inc edx  
jmp .mark_ps_while  
  
.mark_ps_ret:
```



# 代码10

```
9 i586-pc-linux-gcc -c print_hello.c
10 i586-pc-linux-ld -Ttext=2500 -emy_entrance print_hello.o system.o -o print_hello.elf
11 i586-pc-linux-objcopy -O binary print_hello.elf print_hello.bin
12
13 i586-pc-linux-gcc -c print_world.c
14 i586-pc-linux-ld -Ttext=2500 -emy_entrance print_world.o system.o -o print_world.elf
15 i586-pc-linux-objcopy -O binary print_world.elf print_world.bin
```

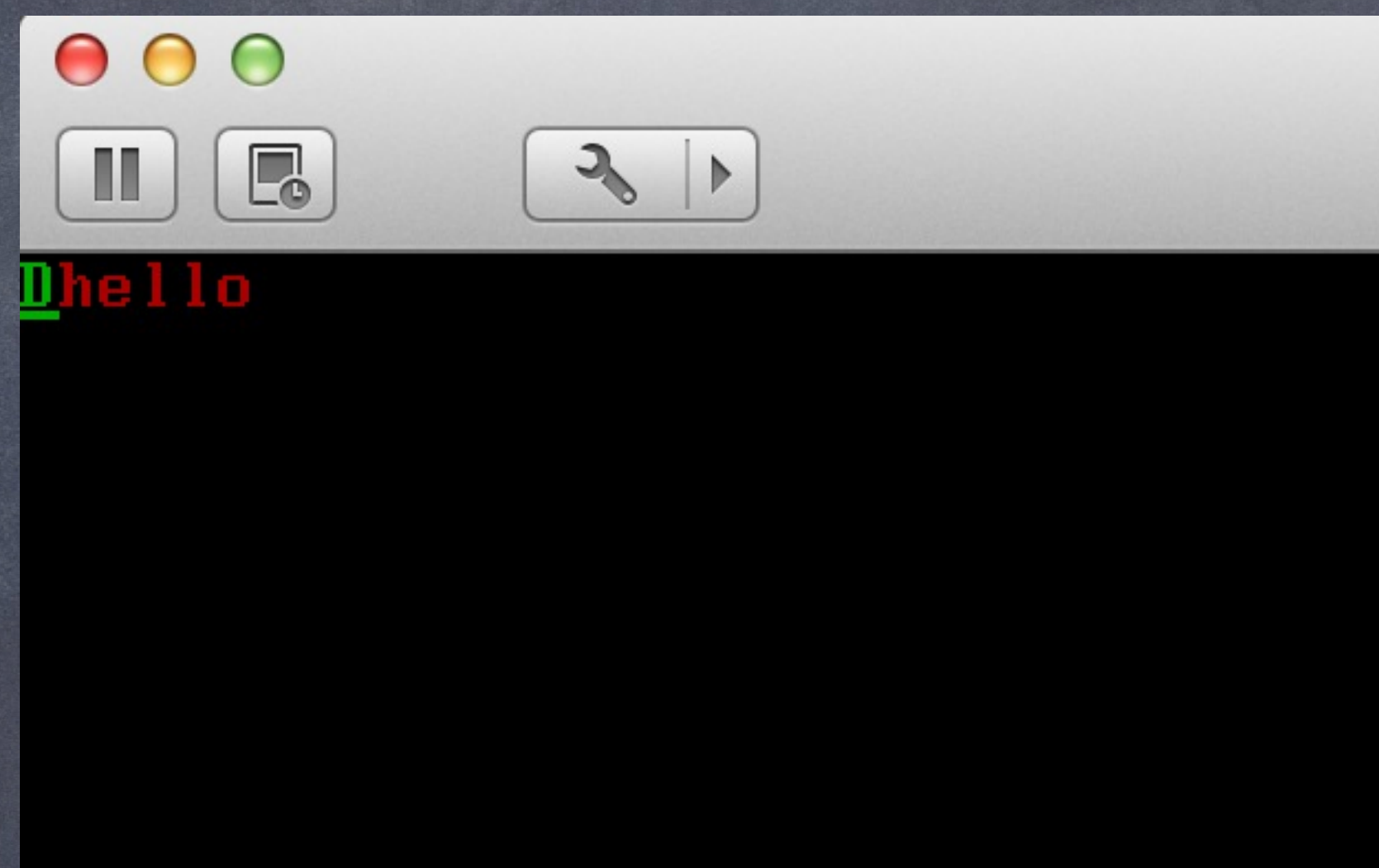


# 代码11

```
17 cat boot.bin print_ab.bin usermode.bin > system.bin
18 dd conv=sync if=system.bin of=system.inter bs=0x2800 count=1
19
20 # write print_hello at 0x2800
21 cat system.inter print_hello.bin > sys_hello.bin
22 dd conv=sync if=sys_hello.bin of=sys_hello.inter bs=0x2C00 count=1
23
24 # write print_world at 0x2C00
25 cat sys_hello.inter print_world.bin > sys_world.bin
26 dd conv=sync if=sys_world.bin of=print_ab.img bs=1440k count=1
```



运行结果:



```
hello
```



# 思考

- Android APP的APK格式和ELF格式有何差异？



谢谢！