

南京大学计算机科学与技术系
Department of Computer Science and Technology Nanjing University

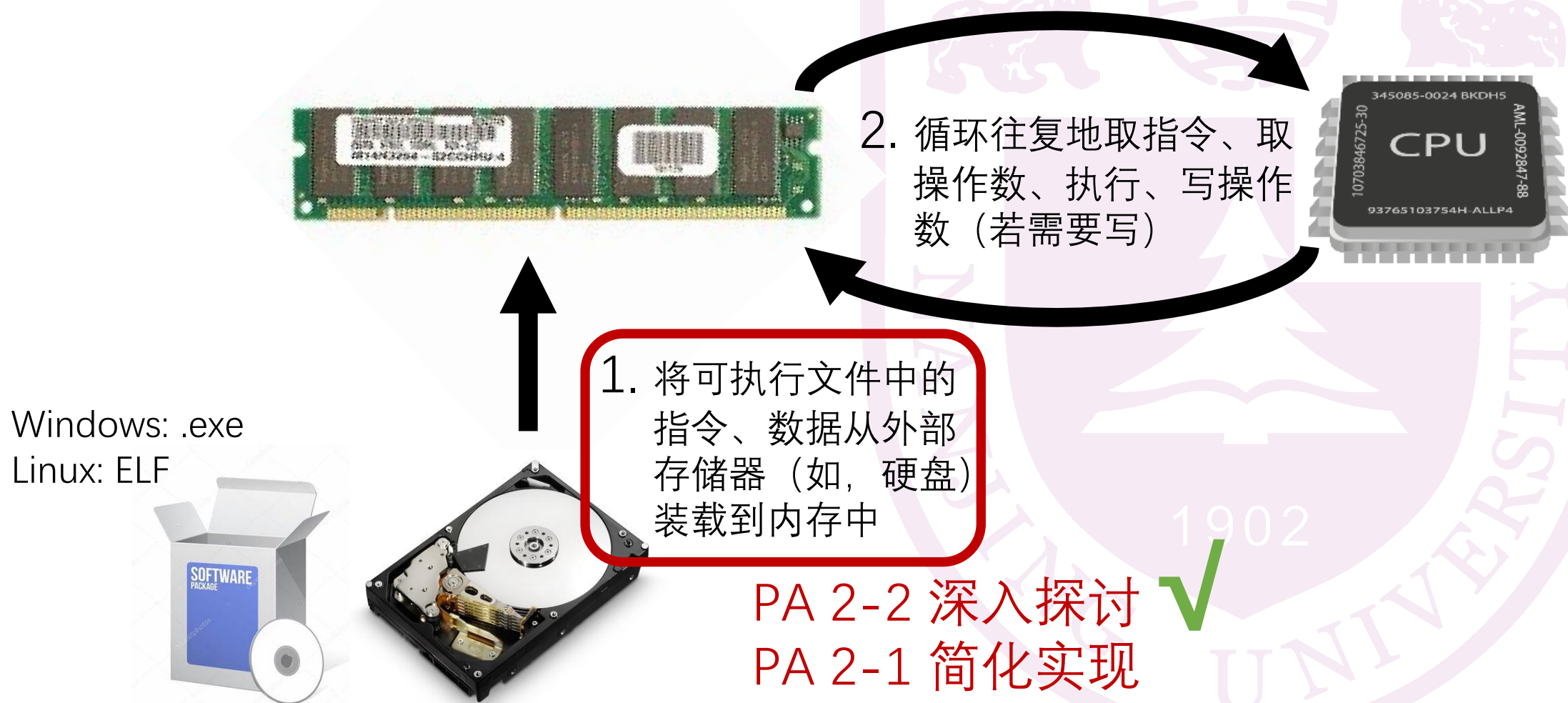
计算机系统基础
Programming Assignment

PA 2-2 – 程序的装载

PA 2-3 – 调试器符号表解析

2020年10月29日 / 2020年10月30日
南京大学《计算机系统基础》课程组

前情提要



目录

- 认识ELF文件
- PA 2-2 - 程序的装载
 - > ELF文件程序头表的解析
- PA 2-3 - 调试器符号表解析
 - > ELF文件符号表的解析

目录

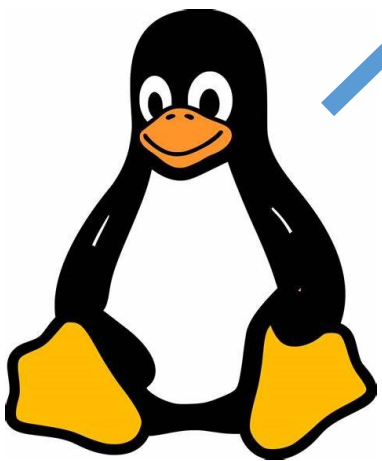
- 认识ELF文件

- PA 2-2 - 程序的装载
 - > ELF文件程序头表的解析
- PA 2-3 - 调试器符号表解析
 - > ELF文件符号表的解析

什么是ELF文件



ELF: Executable and Linkable Format
可执行可链接文件格式



运行

类Unix系统上的
标准格式

.exe

运行

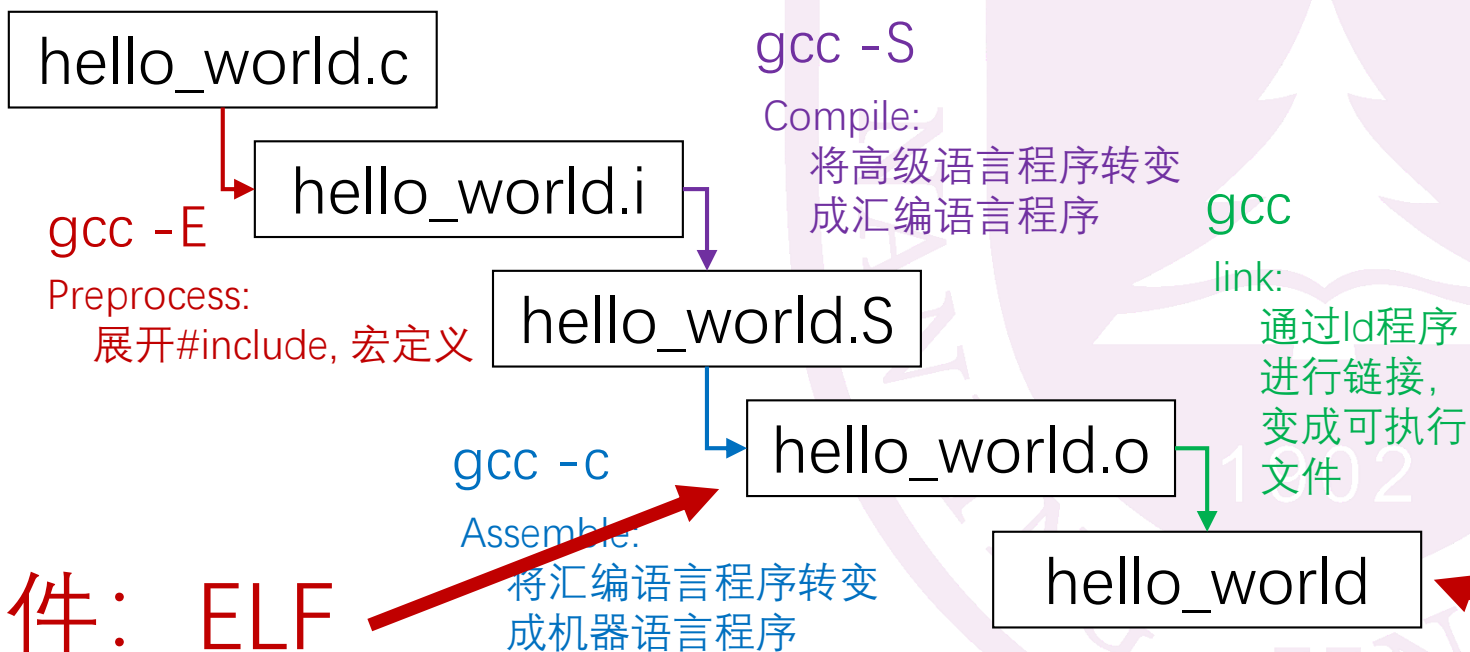


如何得到ELF文件

从高级语言到机器指令 – C语言程序编译的过程

```
$ gcc -o hello_world hello_world.c
```

一条命令执行了四个步骤



可重定位文件: ELF

可执行文件: ELF

ELF文件格式

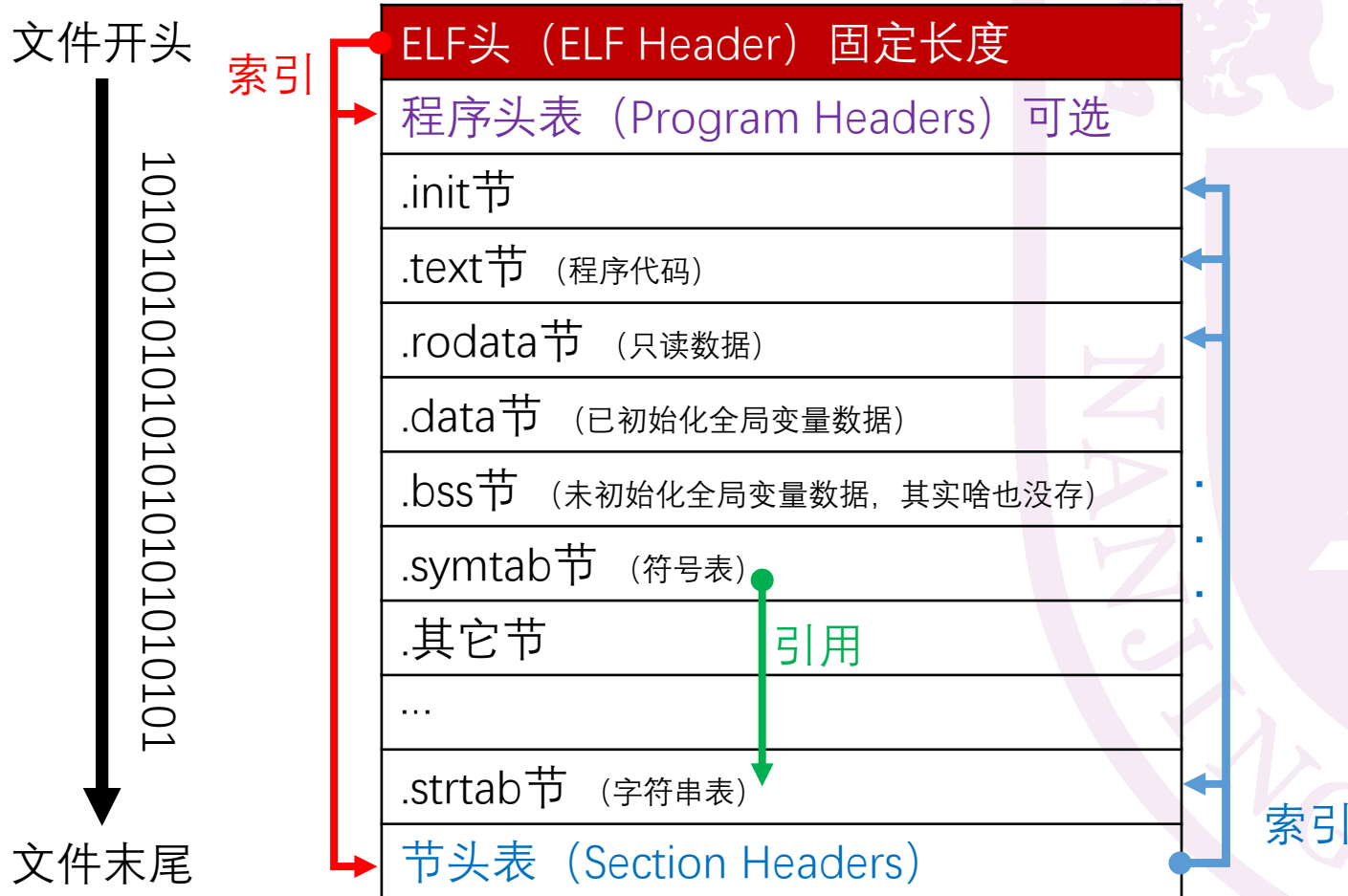


查看ELF文件

```
$ readelf
```

- a 查看所有
- h ELF头
- l 程序头表
- S 节头表
- s 符号表

ELF文件格式



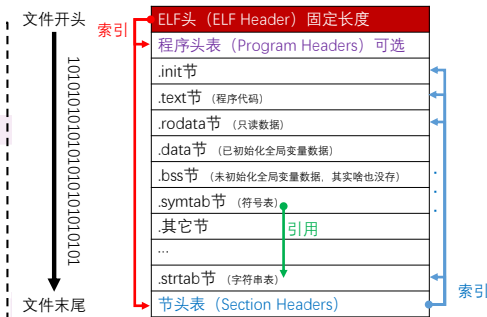
查看ELF文件

```
$ readelf
```

- a 查看所有
- h ELF头
- l 程序头表
- S 节头表
- s 符号表

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: 0x30000
Start of program headers: 52 (bytes into file)
Start of section headers: 18276 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 3
Size of section headers: 40 (bytes)
Number of section headers: 15
Section header string table index: 14

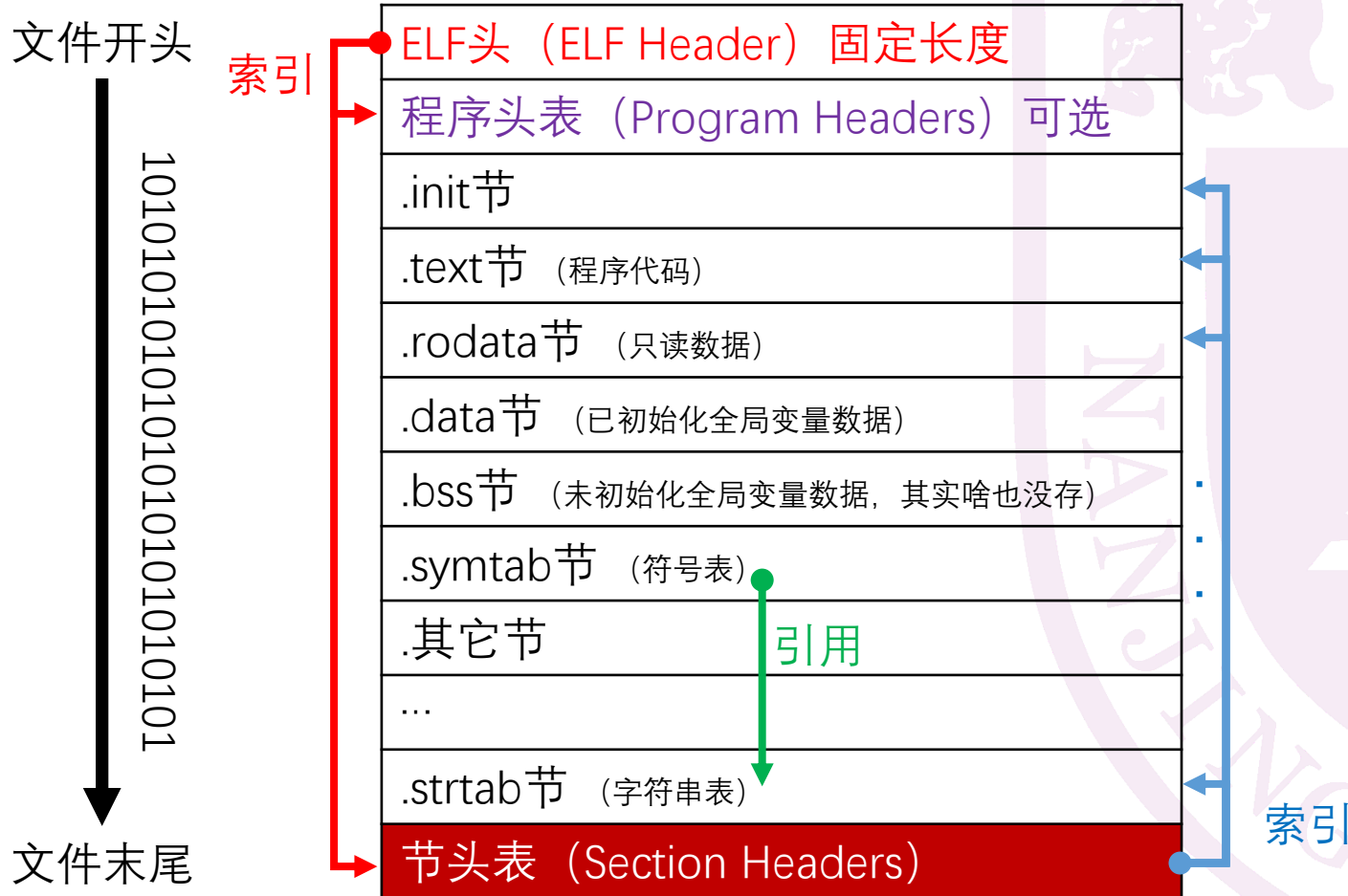


永远位于ELF
文件最开始的地
方, 固定长度
(ELF32,
ELF64)

`readelf -h filename`

ELF头

ELF文件格式



查看ELF文件

```
$ readelf
```

- a 查看所有
- h ELF头
- l 程序头表
- S 节头表
- s 符号表

节头表

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	00030000	001000	0000d0	00	AX	0	0	1
[2]	.eh_frame	PROGBITS	000300d0	0010d0	000084	00	A	0	0	4
[3]	.got.plt	PROGBITS	00032000	002000	00000c	04	WA	0	0	4
[4]	.data	PROGBITS	00032020	002020	000120	00	WA	0	0	32
[5]	.comment	PROGBITS	00000000	002140	000026	01	MS	0	0	1
[6]	.debug_aranges	PROGBITS	00000000	002168	000040	00		0	0	8
[7]	.debug_info	PROGBITS	00000000	0021a8	000142	00		0	0	1
[8]	.debug_abbrev	PROGBITS	00000000	0022ea	0000d6	00		0	0	1
[9]	.debug_line	PROGBITS	00000000	0023c0	0000dc	00		0	0	1
[10]	.debug_str	PROGBITS	00000000	00249c	001a37	01	MS	0	0	1
[11]	.debug_macro	PROGBITS	00000000	003ed3	0005f9	00		0	0	1
[12]	.symtab	SYMTAB	00000000	0044cc	000190	10		13	15	4
[13]	.strtab	STRTAB	00000000	00465c	000078	00		0	0	1
[14]	.shstrtab	STRTAB	00000000	0046d4	000090	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
p (processor specific)

```
readelf -S filename
```



节头表

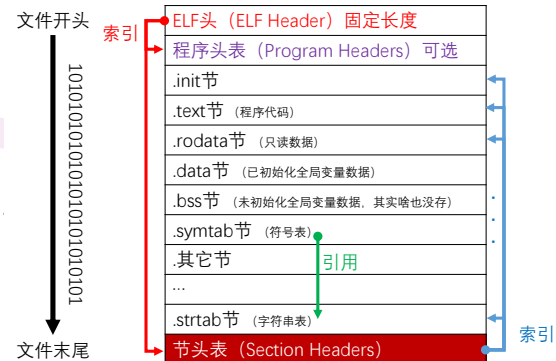
Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	00030000	001000	0000d0	00	AX	0	0	1
[2]	.eh_frame									
[3]	.got.plt									
[4]	.data									
[5]	.comment									
[6]	.debug_aranges									
[7]	.debug_info									
[8]	.debug_abbrev									
[9]	.debug_line									
[10]	.debug_str									
[11]	.debug_macro									
[12]	.symtab									
[13]	.strtab									
[14]	.shstrtab									

文件分为多少个节，每个节
(如.symtab节) 在文件中的什么地方

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
p (processor specific)



`readelf -S filename`

ELF文件格式



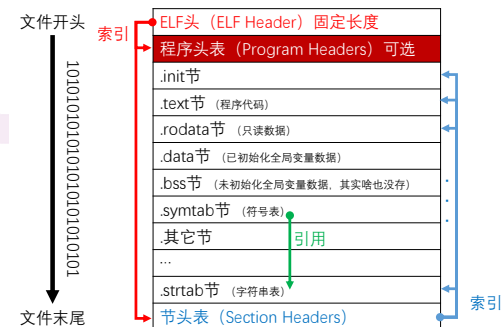
查看ELF文件

```
$ readelf
```

- a 查看所有
- h ELF头
- l 程序头表
- S 节头表
- s 符号表

程序头表

(可选, 仅可执行文件有, 可重定位文件没有)



Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x001000	0x00030000	0x00030000	0x00154	0x00154	R E	0x1000
LOAD	0x002000	0x00032000	0x00032000	0x00140	0x00140	RW	0x1000
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RWE	0x10

Section to Segment mapping:

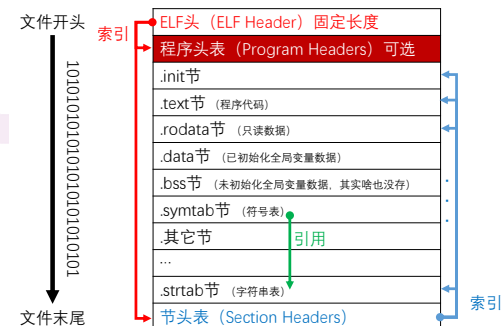
Segment Sections...

00	.text .eh_frame
01	.got.plt .data
02	

`readelf -l filename`

程序头表

(可选, 仅可执行文件有, 可重定位文件没有)



Program Headers:

Type

LOAD

LOAD

GNU_STACK

Section to Segme

Segment Section

00 .text .eh_frame

01 .got.plt .data

02

文件中的哪一部分 (节)
搬到内存中的哪个位置 (段)

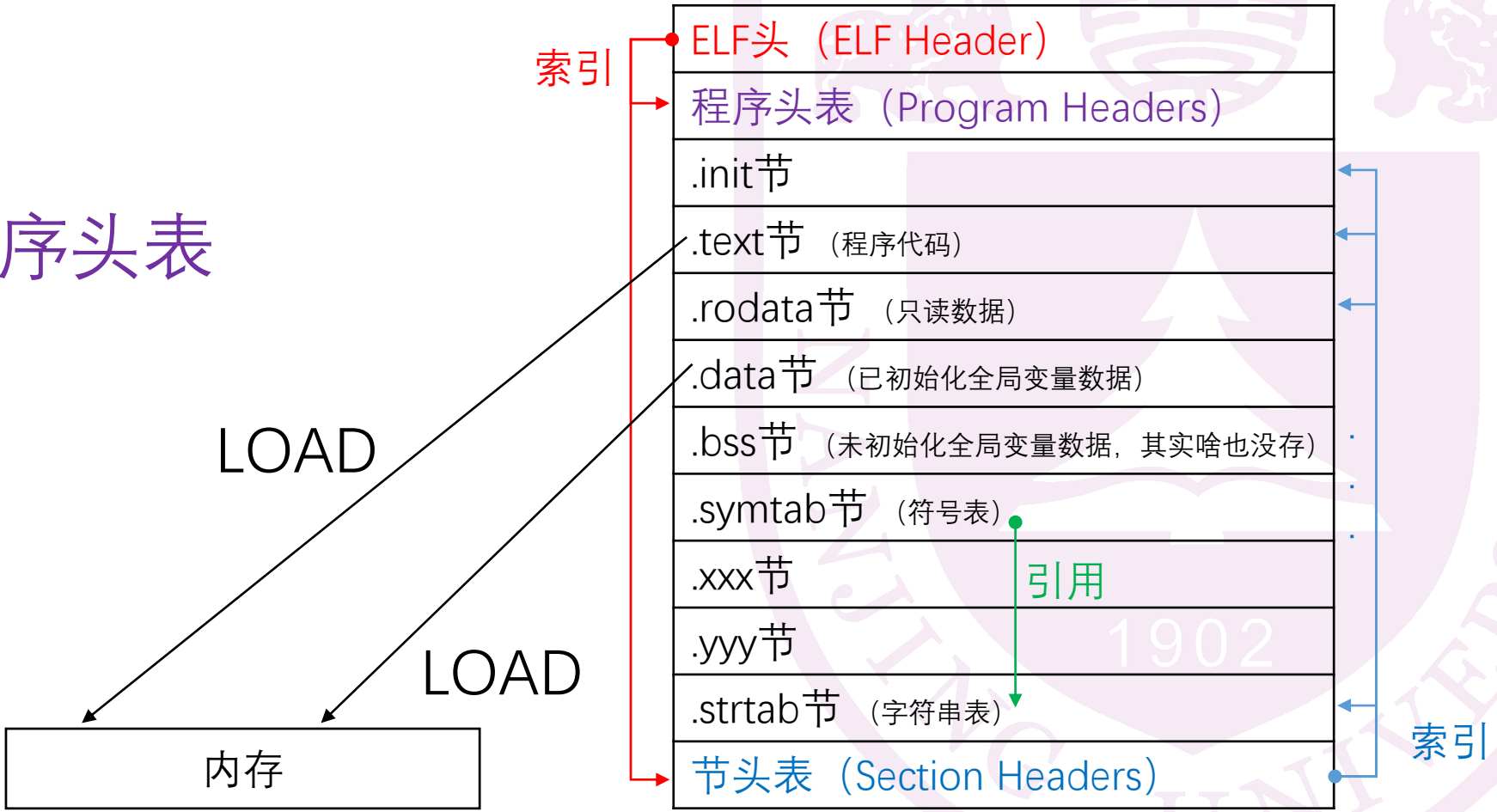
(PA 2-2 装载)

gn
000
000
0

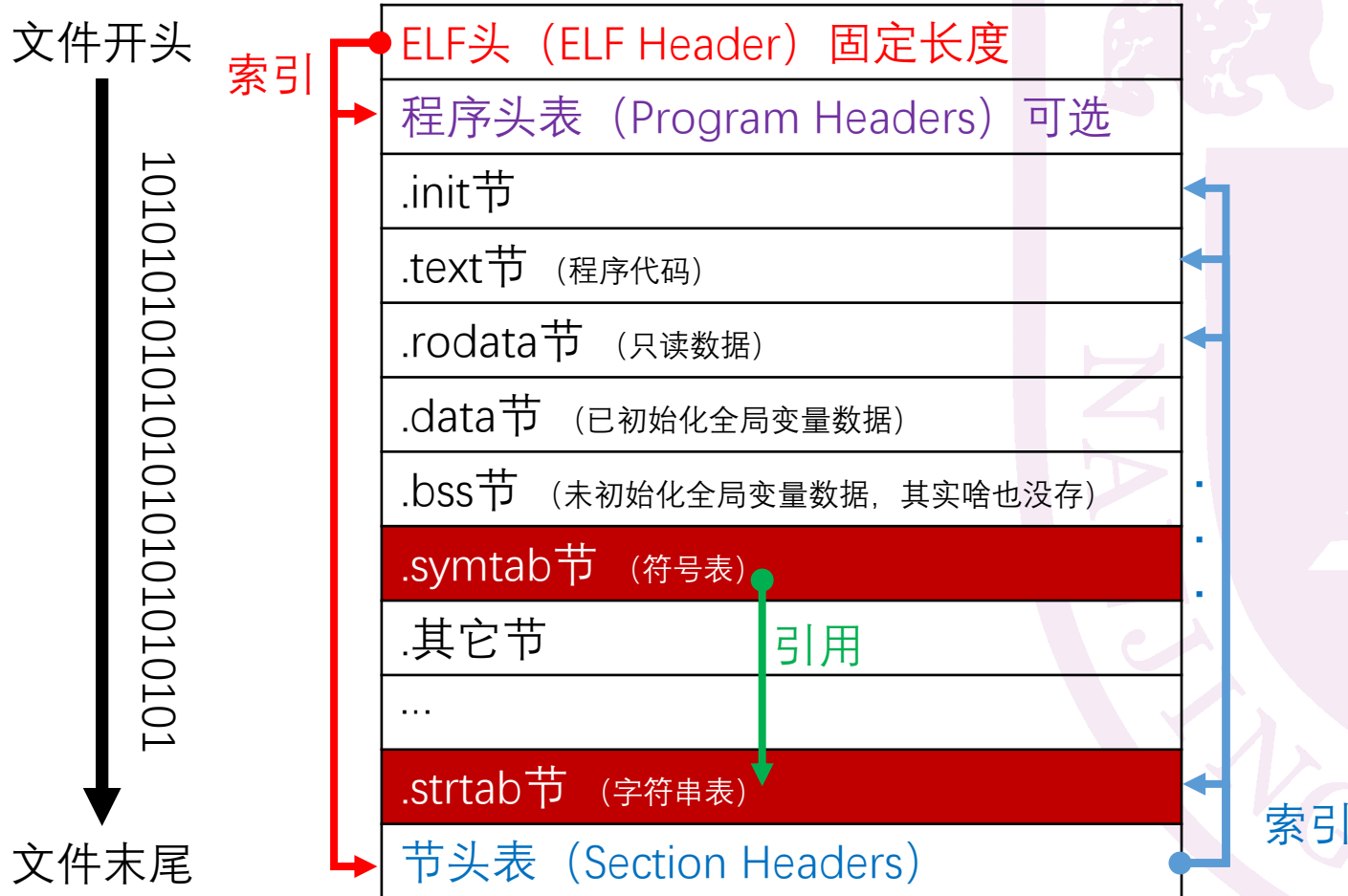
`readelf -l filename`

程序头表

程序头表



ELF文件格式



查看ELF文件

```
$ readelf
```

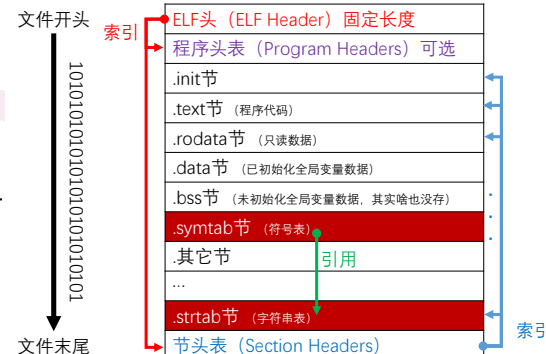
- a 查看所有
- h ELF头
- l 程序头表
- S 节头表
- s 符号表

符号表+字符串表

`readelf -s filename`

Symbol table '.symtab' contains 25 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00030000	0	SECTION	LOCAL	DEFAULT	1	
2:	000300d0	0	SECTION	LOCAL	DEFAULT	2	
...							
14:	00032000	0	OBJECT	LOCAL	DEFAULT	3	__GLOBAL_OFFSET_TABLE__
15:	000300c8	0	FUNC	GLOBAL	HIDDEN	1	__x86.get_pc_thunk.ax
16:	00030005	32	FUNC	GLOBAL	DEFAULT	1	add
17:	000300cc	0	FUNC	GLOBAL	HIDDEN	1	__x86.get_pc_thunk.bx
18:	00032140	0	NOTYPE	GLOBAL	DEFAULT	4	__bss_start
19:	00030025	163	FUNC	GLOBAL	DEFAULT	1	main
20:	00032040	256	OBJECT	GLOBAL	DEFAULT	4	ans
21:	00032140	0	NOTYPE	GLOBAL	DEFAULT	4	__edata
22:	00032140	0	NOTYPE	GLOBAL	DEFAULT	4	__end
23:	00030000	0	NOTYPE	GLOBAL	DEFAULT	1	start
24:	00032020	32	OBJECT	GLOBAL	DEFAULT	4	test_data



符号表+字符串表

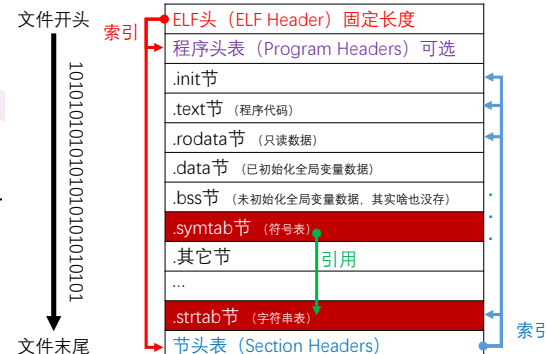
readelf -s filename

Symbol table '.symtab' contains 25 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00030000	0	SECTION	LOCAL	DEFAULT	1	
2:	000300d0						
...							
14:	00032000						
15:	000300c8						
16:	00030005						
17:	000300cc						
18:	00032140						
19:	00030025						
20:	00032040						
21:	00032140	0	NOTYPE	GLOBAL	DEFAULT	4	_edata
22:	00032140	0	NOTYPE	GLOBAL	DEFAULT	4	_end
23:	00030000	0	NOTYPE	GLOBAL	DEFAULT	1	start
24:	00032020	32	OBJECT	GLOBAL	DEFAULT	4	test_data

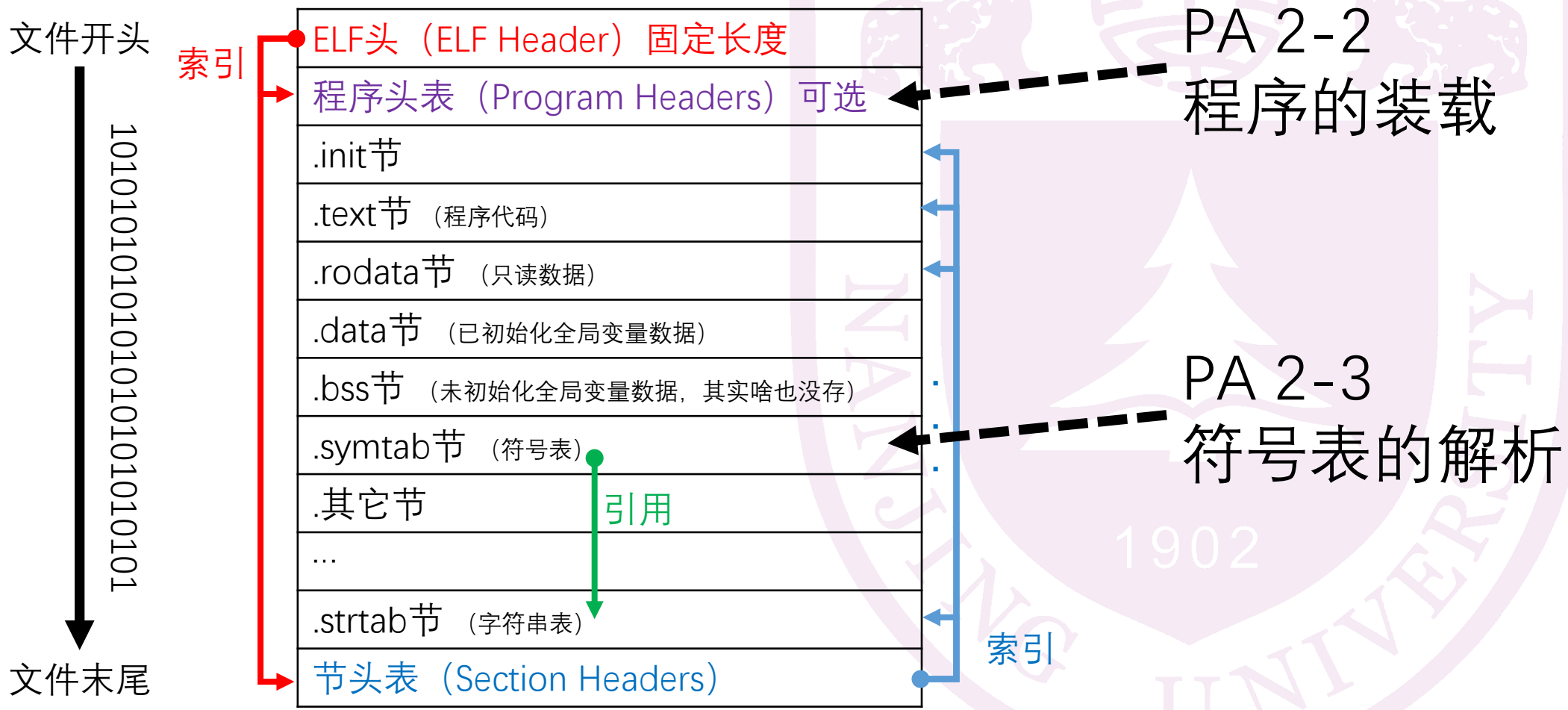
某全局变量或函数（如，'main'）
对应内存的哪个地址

(PA 2-3 符号表解析)



ET_TABLE_
_thunk.ax
_thunk.bx

ELF文件格式

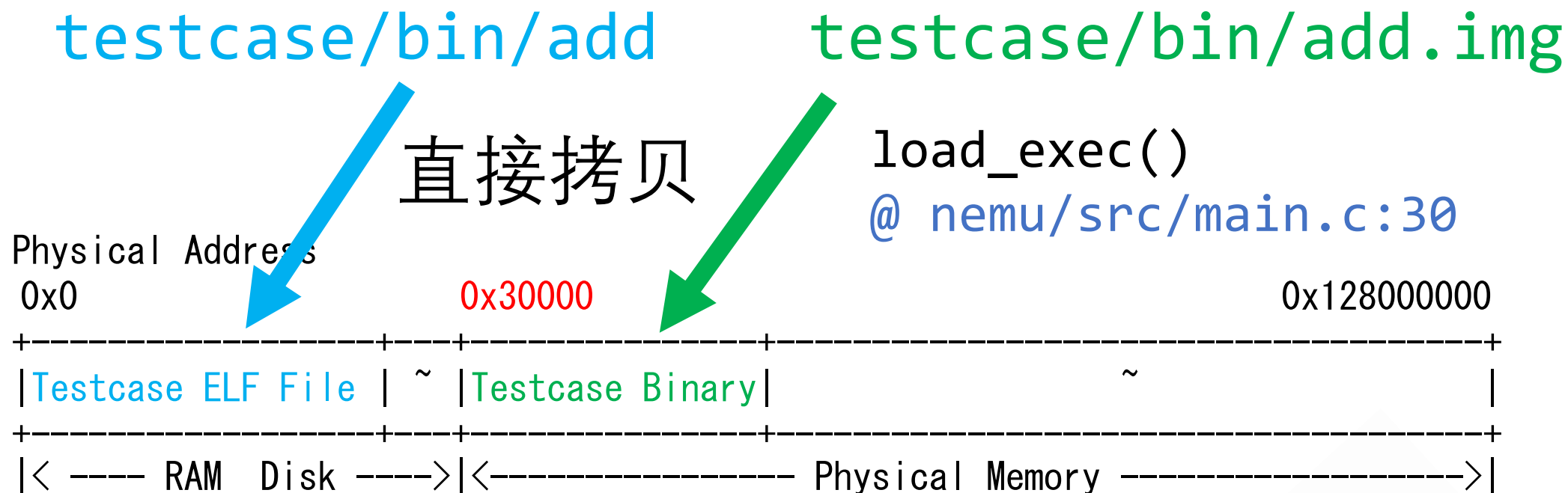


目录

- 认识ELF文件
- PA 2-2 - 程序的装载 → ELF文件程序头表的解析
- PA 2-3 - 调试器符号表解析 → ELF文件符号表的解析

原先的NEMU是怎么装载并运行程序的？

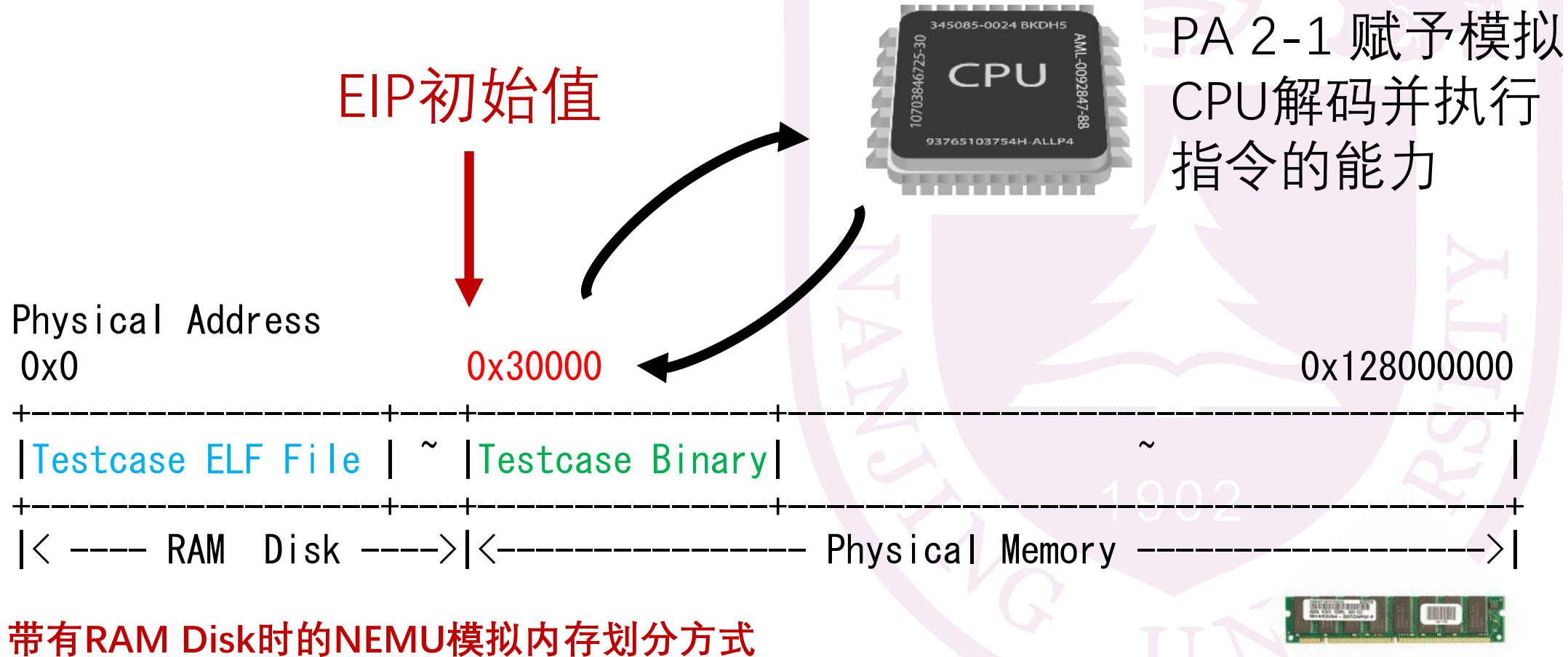
1.1 NEMU初始化模拟内存（PA 2-1的装载）



带有RAM Disk时的NEMU模拟内存划分方式



原先的NEMU是怎么装载并运行程序的？



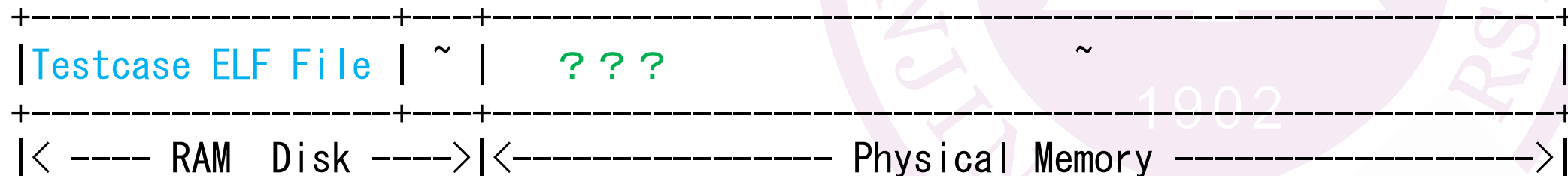
PA 2-2 以后NEMU是怎么装载并运行程序的?

Physical Address

0x0

0x30000

0x128000000

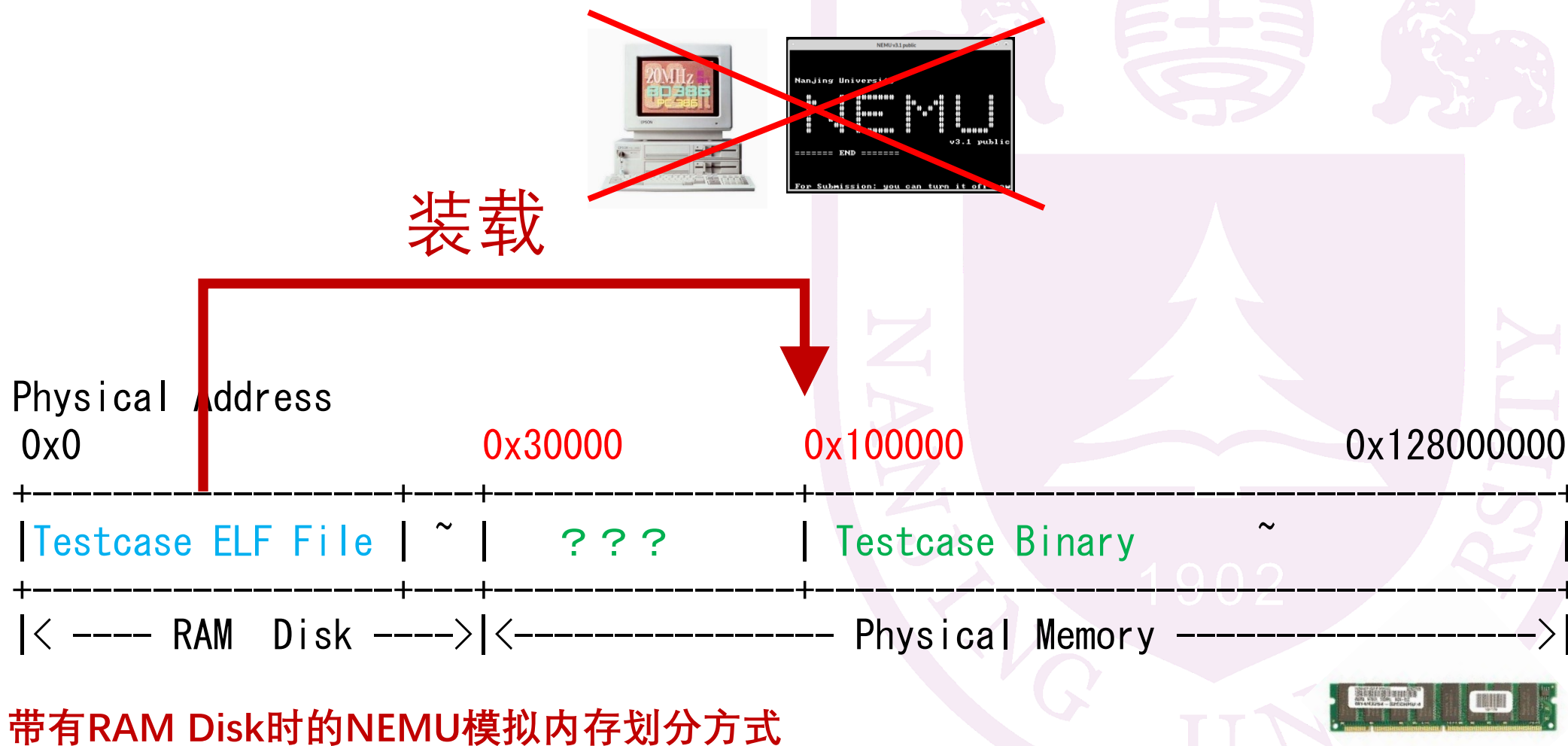


带有RAM Disk时的NEMU模拟内存划分方式

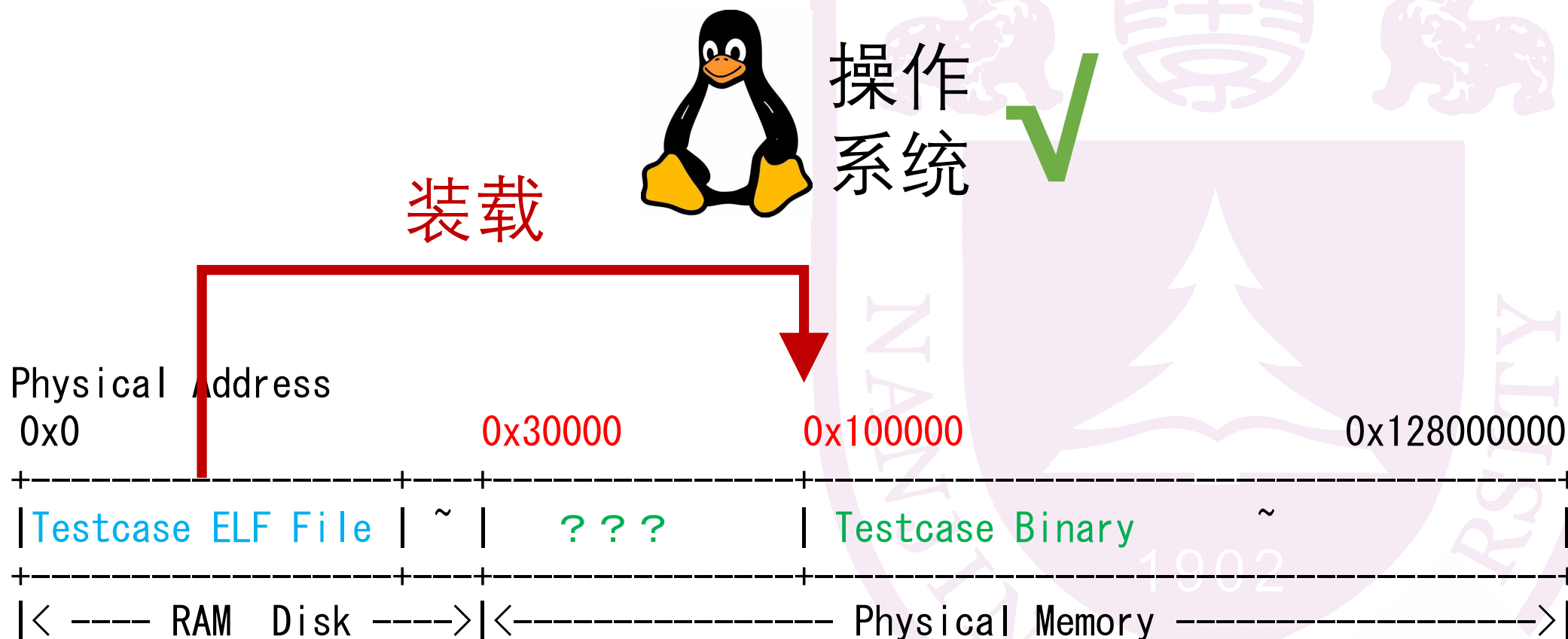




PA 2-2 以后NEMU是怎么装载并运行程序的?

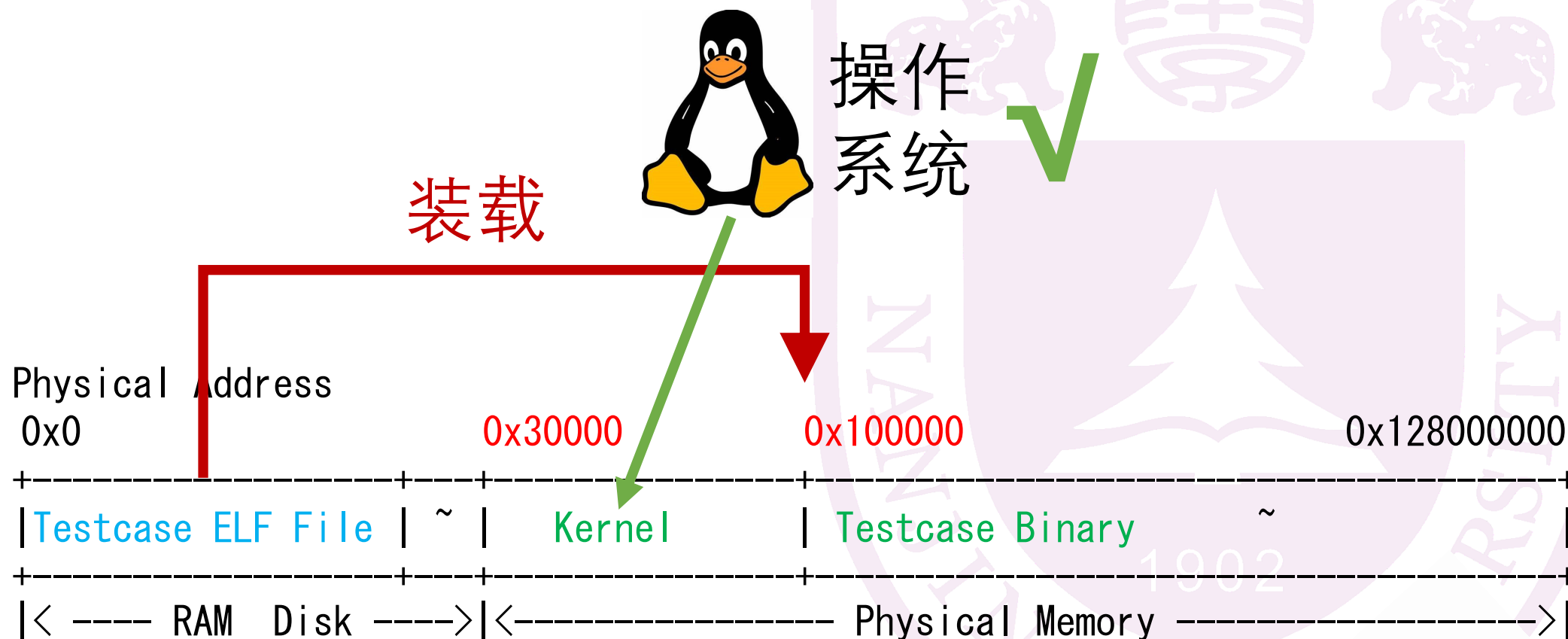


PA 2-2 以后NEMU是怎么装载并运行程序的?



帶有RAM Disk时的NEMU模拟内存划分方式

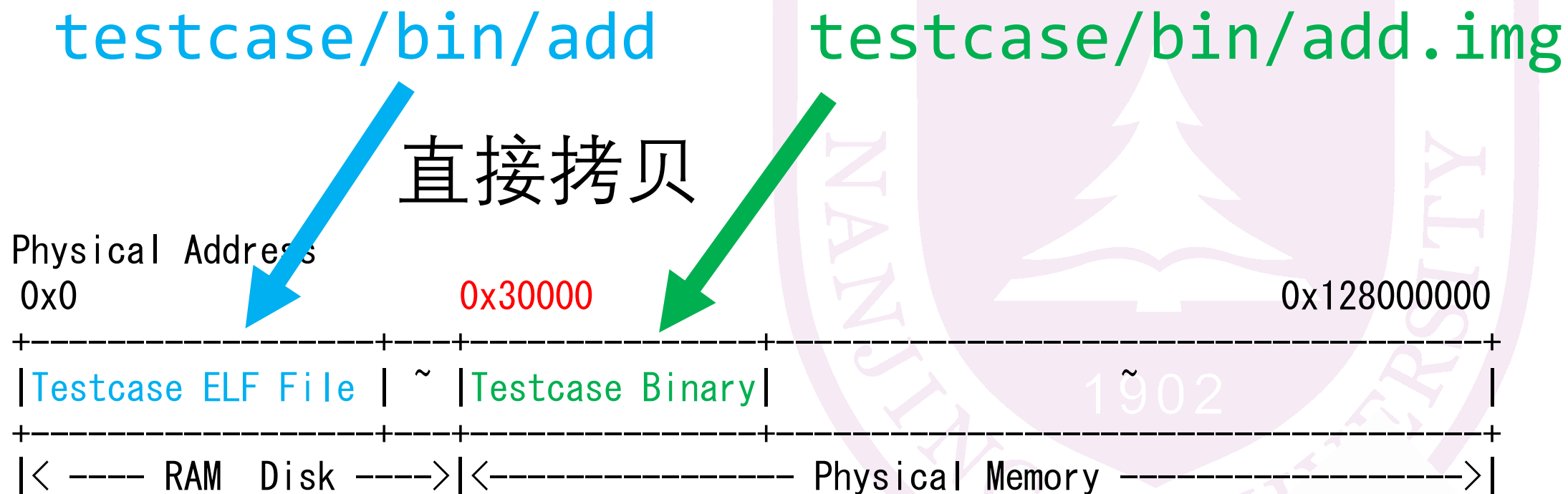
PA 2-2 以后NEMU是怎么装载并运行程序的？



带有RAM Disk时的NEMU模拟内存划分方式



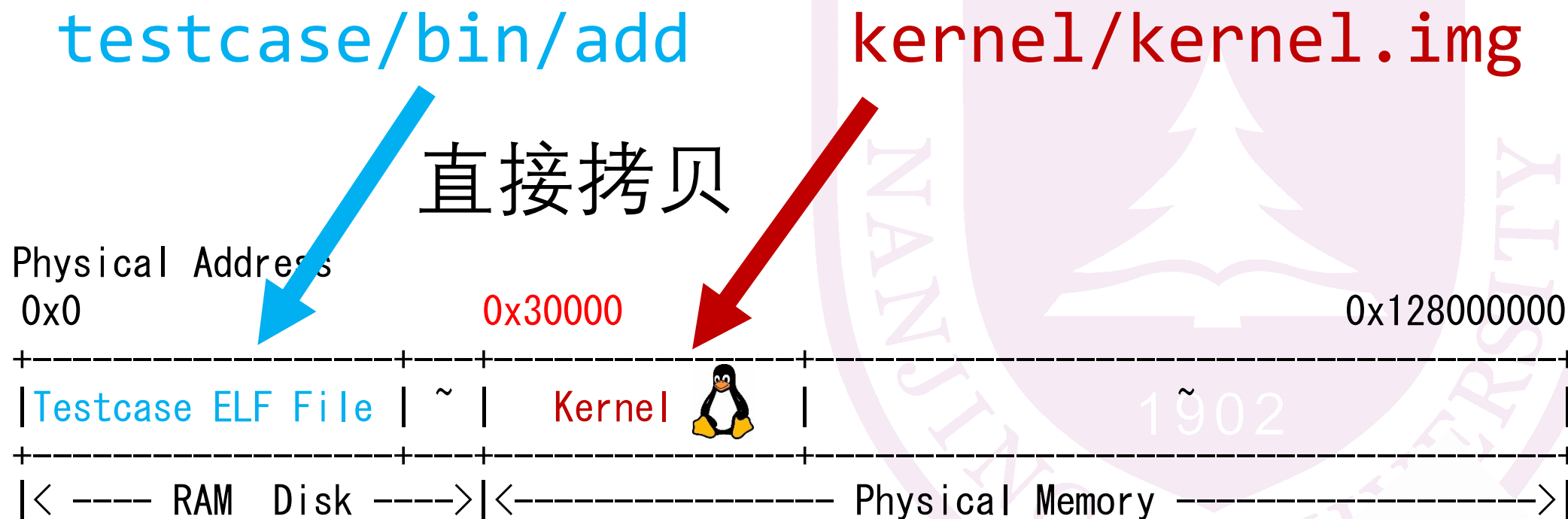
PA 2-2 以后NEMU是怎么装载并运行程序的?



带有RAM Disk时的NEMU模拟内存划分方式



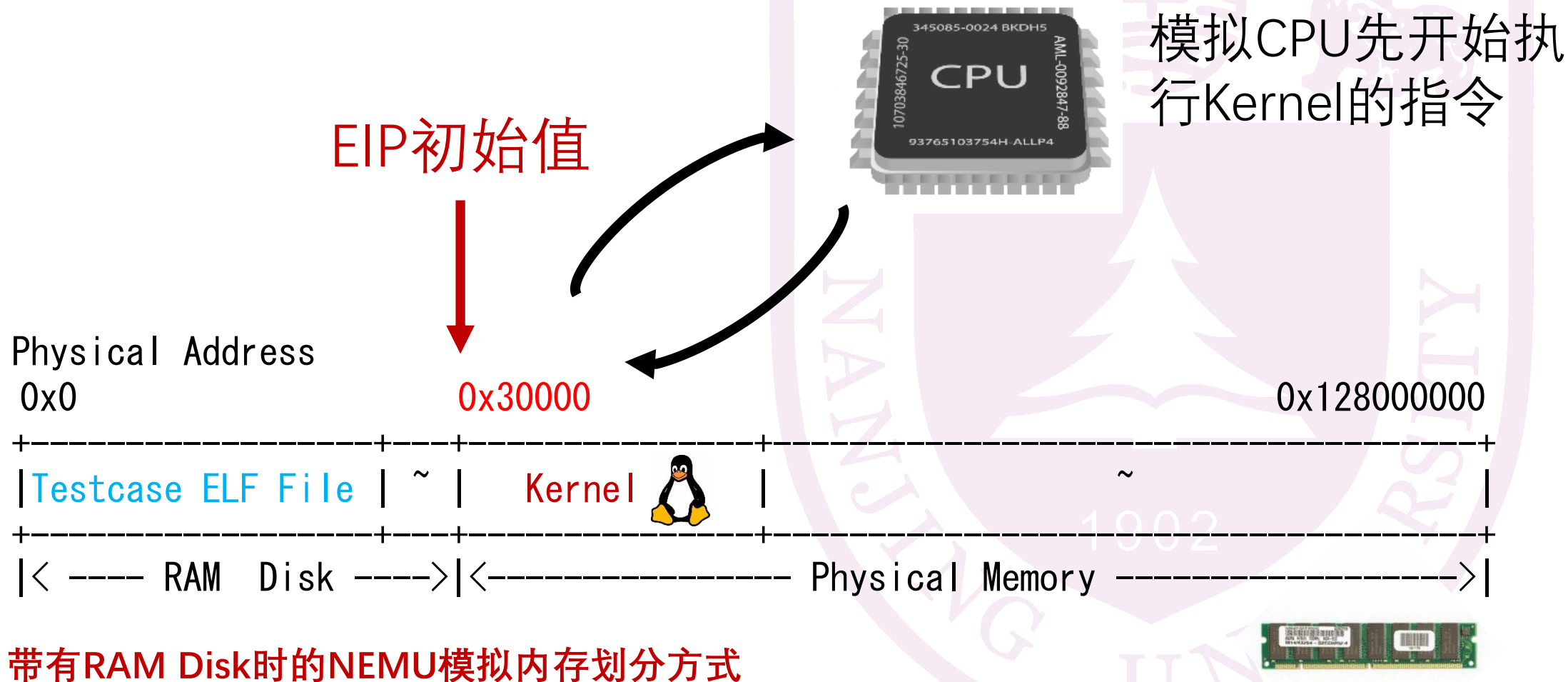
PA 2-2 以后NEMU是怎么装载并运行程序的?



带有RAM Disk时的NEMU模拟内存划分方式



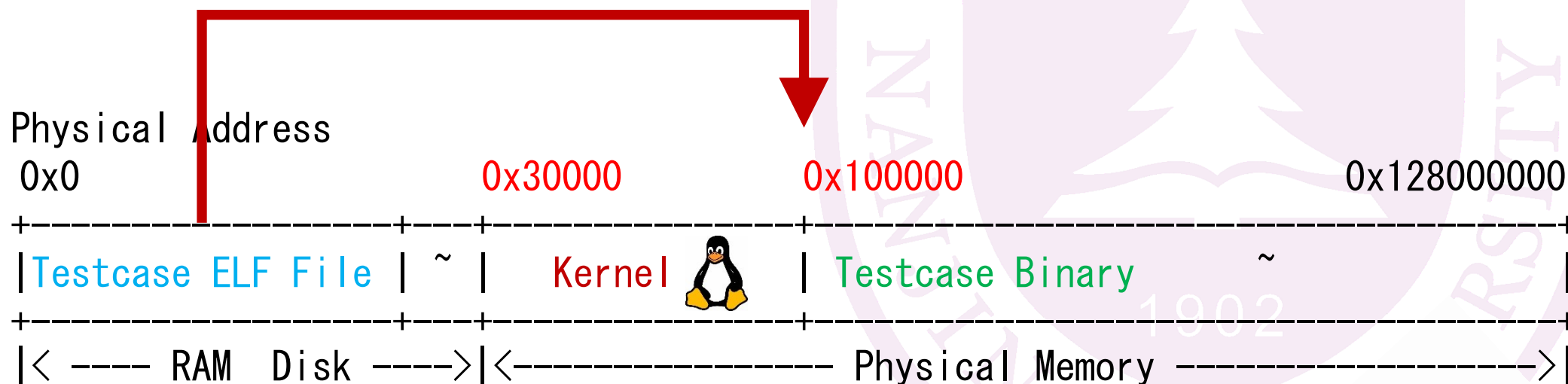
PA 2-2 以后NEMU是怎么装载并运行程序的?



带有RAM Disk时的NEMU模拟内存划分方式

PA 2-2 以后NEMU是怎么装载并运行程序的?

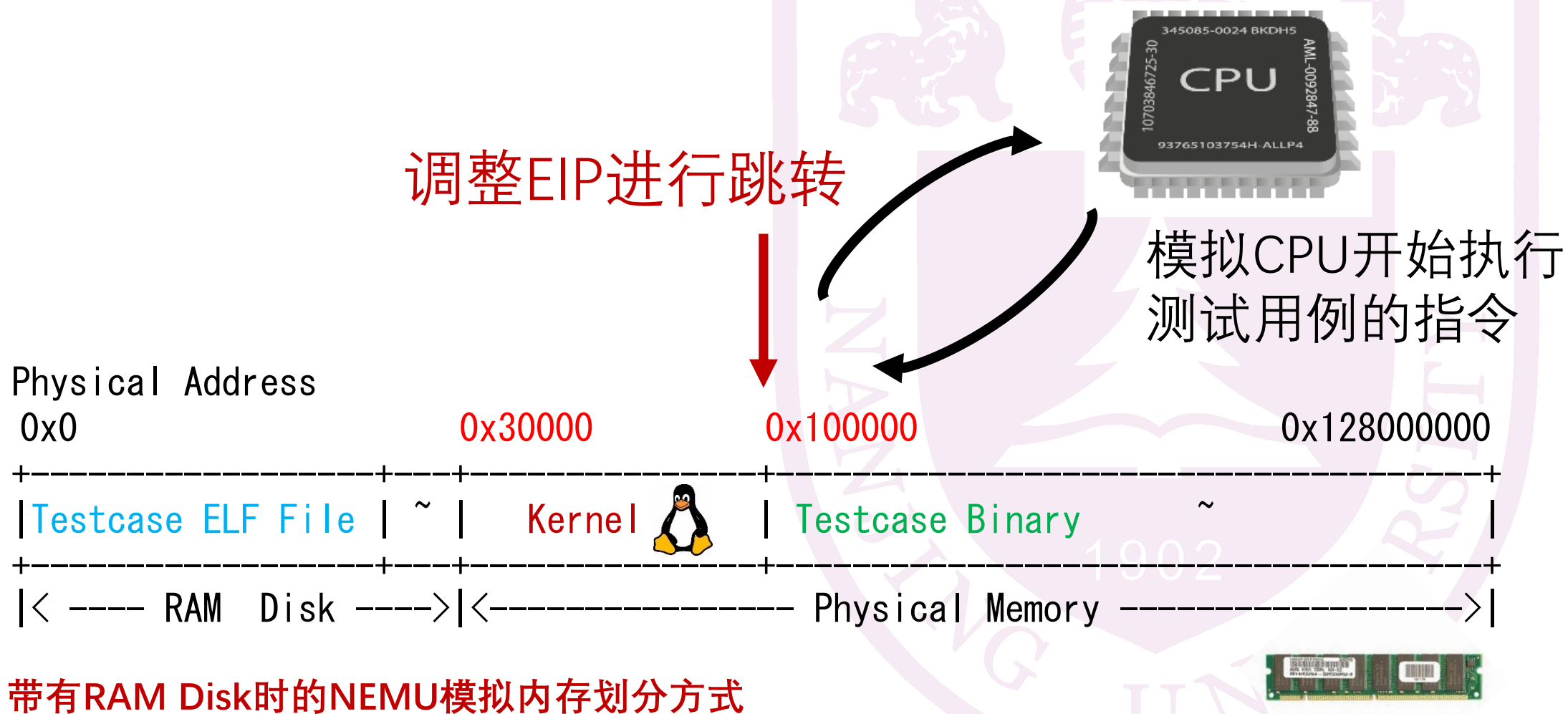
由Kernel解析测试用例的ELF文件并装载



带有RAM Disk时的NEMU模拟内存划分方式



PA 2-2 以后NEMU是怎么装载并运行程序的?



Kernel装载程序

- 如何实现对ELF文件的装载
- 如何在Kernel中加入这一功能



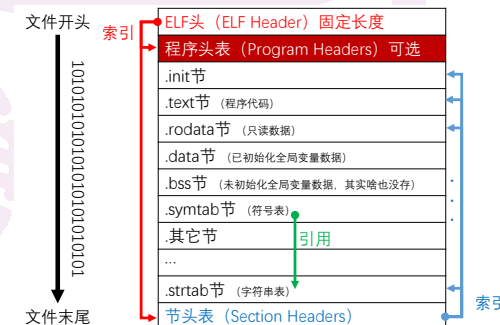
Kernel装载程序

- 如何实现对ELF文件的装载
- 如何在Kernel中加入这一功能



装载程序的实现：解析程序头表

文件中的哪一部分（节）
搬到内存中的哪个位置（段）



Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x001000	0x00030000	0x00030000	0x00154	0x00154	R E	0x1000
LOAD	0x002000	0x00032000	0x00032000	0x00140	0x00140	RW	0x1000
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RWE	0x10

Section to Segment mapping:

Segment Sections...

00	.text .eh_frame
01	.got.plt .data
02	

`readelf -l filename`

装载程序的实现：解析程序头表

*Type*指出ELF文件中节的类型, *Type*为
*LOAD*表示需要装载

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x001000	0x00030000	0x00030000	0x00154	0x00154	R E	0x1000
LOAD	0x002000	0x00032000	0x00032000	0x00140	0x00140	RW	0x1000
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RWE	0x10

Section to Segment mapping:

Segment Sections...

00	.text .eh_frame
01	.got.plt .data
02	

`readelf -l filename`

装载程序的实现：解析程序头表

若Type为LOAD，则ELF文件中从文件Offset开始位置，连续FileSiz个字节的内容需要被装载

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x001000	0x00030000	0x00030000	0x00154	0x00154	R E	0x1000
LOAD	0x002000	0x00032000	0x00032000	0x00140	0x00140	RW	0x1000
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RWE	0x10

Section to Segment mapping:

Segment Sections...

00	.text	.eh_frame
01	.got.plt	.data
02		

`readelf -l filename`

装载程序的实现：解析程序头表

MemSiz可能大于FileSiz

装载到内存VirtAddr开始，连续MemSiz个字节的区域中

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x001000	0x00030000	0x00030000	0x00154	0x00154	R E	0x1000
LOAD	0x002000	0x00032000	0x00032000	0x00140	0x00140	RW	0x1000
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RWE	0x10

Section to Segment mapping:

Segment Sections...

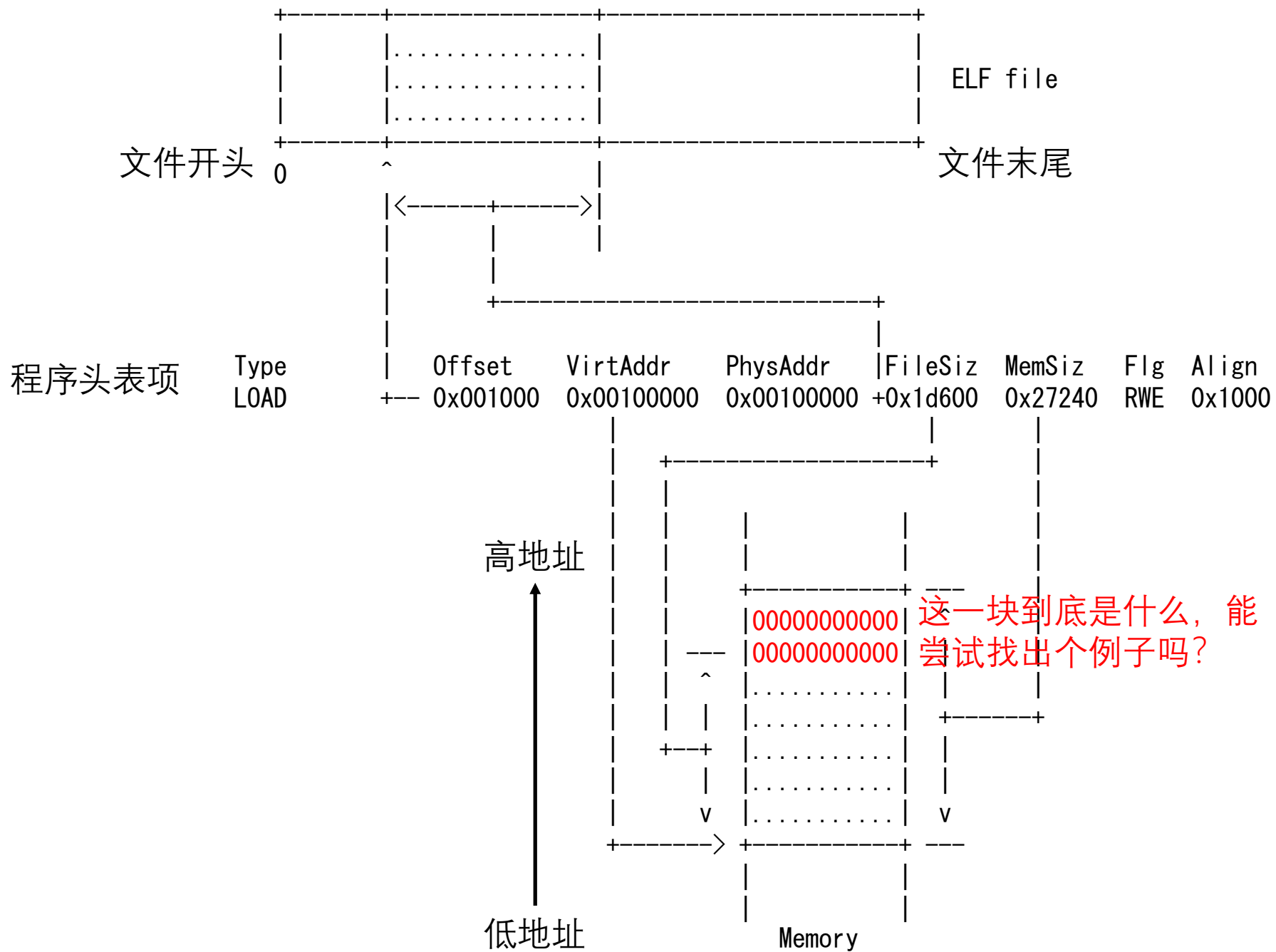
00	.text .eh_frame
01	.got.plt .data
02	

`readelf -l filename`

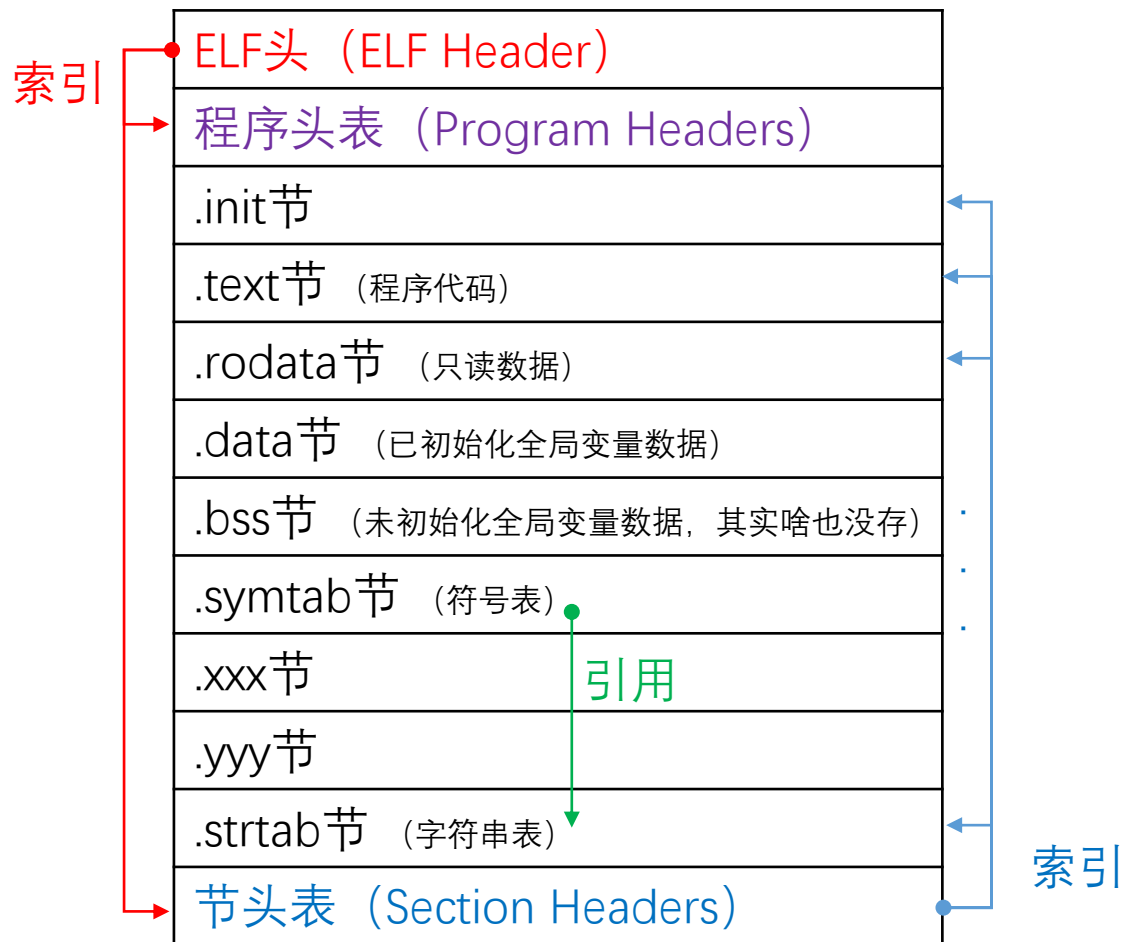
装载ELF可执行文件

- 所谓装载ELF可执行文件，就是把程序头表中，Type为LOAD类型的项目，把ELF文件中从Offset开始的FileSiz字节的内容，拷贝到内存VirtAddr开始的MemSiz大小的区域中
- 有时候MemSiz比FileSiz大？把内存中
$$\text{VirtAddr} + [\text{FileSiz}, \text{MemSiz}]$$

的区域清零



完整的装载逻辑

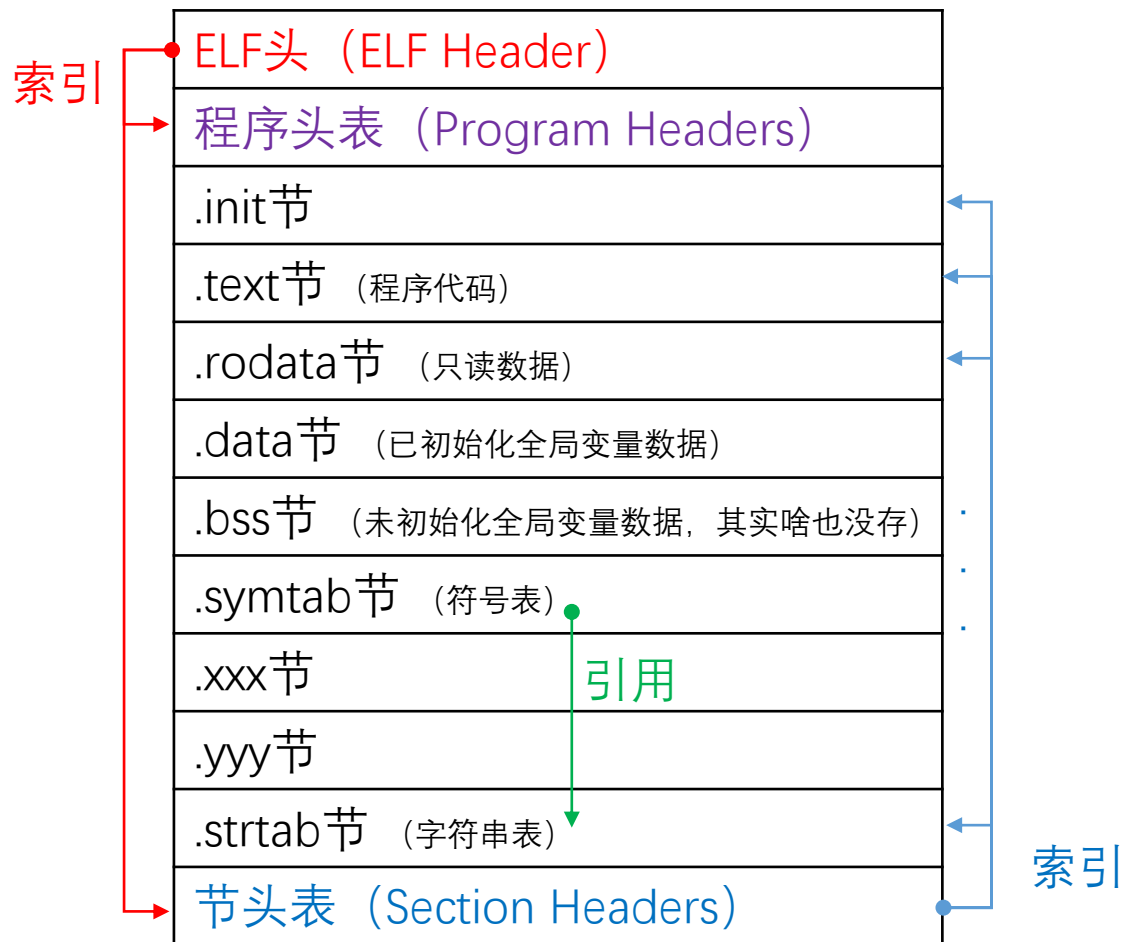


loader程序

输入: ELF文件
输出: 程序入口地址

1. 读入ELF头
2. 程序入口地址 = entry
3. 定位程序头表
4. for 程序头表中的每一项
5. if Type == Load then
6. 执行装载到内存
7. end if
8. end for
9. return 程序入口地址

完整的装载逻辑



loader程序

输入: ELF文件

输出: 程序入口地址

1. 读入ELF头
2. 程序入口地址 = entry
3. 定位程序头表
4. for 程序头表中的每一项
5. if Type == Load then
6. 执行装载到内存
7. end if
8. end for
9. return 程序入口地址

ELF头的编程解析

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:                       0x30000
Start of program headers:                  52 (bytes into file)
Start of section headers:                  18276 (bytes into file)
Flags:                                     0x0
Size of this header:                       52 (bytes)
Size of program headers:                   32 (bytes)
Number of program headers:                  3
Size of section headers:                   40 (bytes)
Number of section headers:                  15
Section header string table index: 14
```

```
#define EI_NIDENT 16
```

```
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

ELF头，它位于整个ELF文件最开始的地方。在32位Linux系统中，`<elf.h>`头文件中的`Elf32_Ehdr`数据结构与之对应。

ELF头的编程解析

```
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 + 入口地址)
Machine:                               Intel 80386
Version:                               0x1
Entry point address:                   0x30000
Start of program headers:               52 (bytes into file)
Start of section headers:               18276 (bytes into file)
Flags:                                  0x0
Size of this header:                    52 (bytes)
Size of program headers:                 32 (bytes)
Number of program headers:                3
Size of section headers:                 40 (bytes)
Number of section headers:                15
Section header string table index:       14
```

```
#define EI_NIDENT 16

typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

ELF头，它位于整个ELF文件最开始的地方。在32位Linux系统中，`<elf.h>`头文件中的`Elf32_Ehdr`数据结构与之对应。

ELF头的编程解析

```
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:                   0x30000
Start of program headers:               52 (bytes into file)
Start of section headers:               18276 (bytes into file)
Flags:                                  0x0
Size of this header:                     52 (bytes)
Size of program headers:                 32 (bytes)
Number of program headers:                3
Size of section headers:                 40 (bytes)
Number of section headers:                15
Section header string table index:       14
```

程序头表位置

```
#define EI_NIDENT 16

typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

ELF头，它位于整个ELF文件最开始的地方。在32位Linux系统中，<elf.h>头文件中的Elf32_Ehdr数据结构与之对应。

ELF头的编程解析

kernel/src/elf/elf.c

```
uint32_t loader() {  
    Elf32_Ehdr *elf;  
    Elf32_Phdr *ph, *eph;
```

```
#ifdef HAS_DEVICE_IDE  
... // 没有模拟硬盘
```

```
#else
```

```
elf = (void *)0x0;
```

```
// 模拟内存0x0处是RAM Disk, 存放的就是testcase ELF file, 最开始的部分是ELF头  
Log("ELF loading from ram disk.");
```

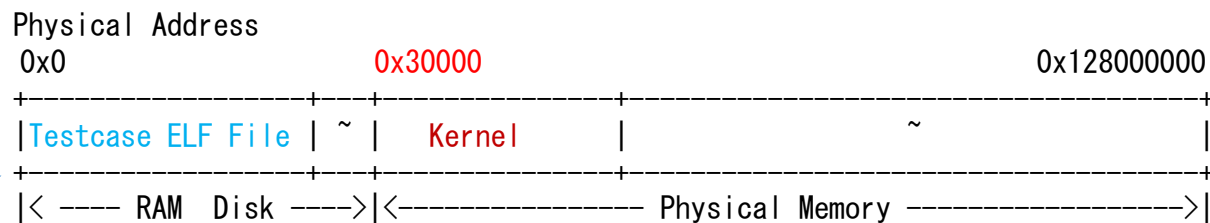
```
#endif
```

```
ph = (void *)elf + elf->e_phoff; // 找到ELF文件中的程序头表
```

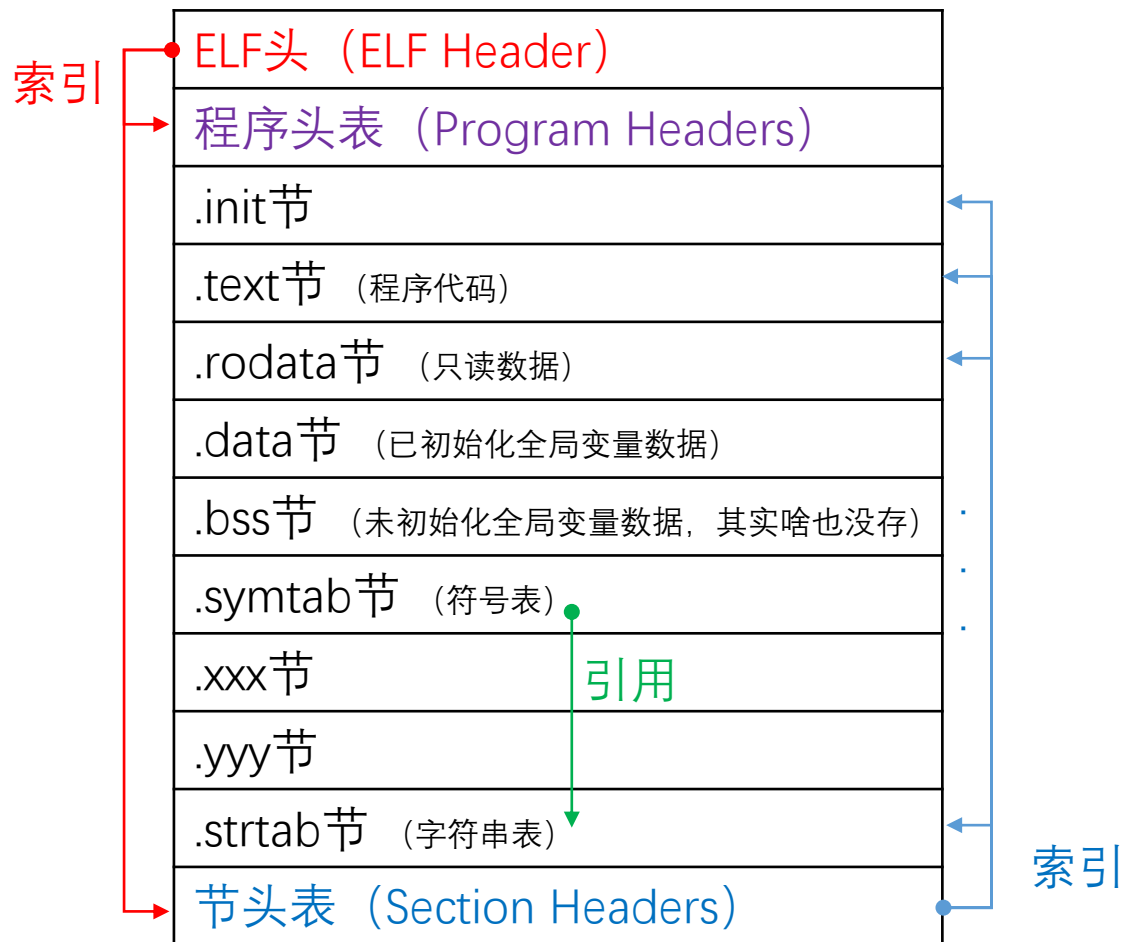
```
...
```

```
volatile uint32_t entry = elf->e_entry; // 头文件中指出的testcase起始地址, 应该是0x60000  
return entry; // 返回testcase起始地址, 在init_cond()后面执行((void (*)(void))eip)();
```

```
}
```



完整的装载逻辑



loader程序

输入: ELF文件
输出: 程序入口地址

1. 读入ELF头
2. 程序入口地址 = entry
3. **定位程序头表**
4. **for 程序头表中的每一项**
5. **if Type == Load then**
6. **执行装载到内存**
7. **end if**
8. **end for**
9. return 程序入口地址

程序头表的编程解析

kernel/src/elf/elf.c

```
uint32_t loader() {
    Elf32_Ehdr *elf;
    Elf32_Phdr *ph, *eph;

#ifdef HAS_DEVICE_IDE
    ... // 没有模拟硬盘
#else
    elf = (void *)0x0;
    // 模拟内存0x0处是RAM Disk, 存放的就是testcase ELF file, 最开始的部分是ELF头
    Log("ELF loading from ram disk.");
#endif
    ph = (void *)elf + elf->e_phoff; // 找到ELF文件中的程序头表
    ...

    volatile uint32_t entry = elf->e_entry; // 头文件中指出的testcase起始地址, 应该是0x60000
    return entry; // 返回testcase起始地址, 在init_cond()后面执行((void (*)(void))eip)();
}
```

程序头表的编程解析

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x001000	0x00030000	0x00030000	0x00154	0x00154	R E	0x1000
LOAD	0x002000	0x00032000	0x00032000	0x00140	0x00140	RW	0x1000
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RWE	0x10

Section to Segment mapping:

Segment Sections...

00	.text	.eh_frame
01	.got.plt	.data
02		

Elf32_Phdr类型的一个数组

Elf32_Phdr定义在<elf.h>

```
typedef struct {
    uint32_t    p_type;
    Elf32_Off   p_offset;
    Elf32_Addr  p_vaddr;
    Elf32_Addr  p_paddr;
    uint32_t    p_filesz;
    uint32_t    p_memsz;
    uint32_t    p_flags;
    uint32_t    p_align;
} Elf32_Phdr;
```

程序头表的编程解析

kernel/src/elf/elf.c

```
uint32_t loader() {  
    ...  
  
    /* Load each program segment */  
    ph = (void *)elf + elf->e_phoff; // 找到ELF文件中的程序头表  
    eph = ph + elf->e_phnum;  
    for(; ph < eph; ph++) { // 扫描程序头表中的各个表项  
        if(ph->p_type == PT_LOAD) { // 如果类型是LOAD, 那么就去装载吧  
  
            panic("Please implement the loader");  
  
            /* TODO: copy the segment from the ELF file to its proper memory area */  
  
            /* TODO: zeror the memory area [vaddr + file_sz, vaddr + mem_sz) */  
            #ifdef IA32_PAGE  
                ... // 没有开启分页  
            #endif  
        }  
    }  
    ...  
}
```

```
uint32_t loader() {
    Elf32_Ehdr *elf;
    Elf32_Phdr *ph, *eph;

#ifdef HAS_DEVICE_IDE
    ... // 没有模拟硬盘
#else
    elf = (void *)0x0;    // 模拟内存0x0处是RAM Disk, 存放的就是testcase ELF file, 最开始的部分是ELF头
    Log("ELF loading from ram disk.");
#endif

    /* Load each program segment */
    ph = (void *)elf + elf->e_phoff; // 找到ELF文件中的程序头表
    eph = ph + elf->e_phnum;
    for(; ph < eph; ph++) { // 扫描程序头表中的各个表项
        if(ph->p_type == PT_LOAD) { // 如果类型是LOAD, 那么就去装载吧
            panic("Please implement the loader");
            /* TODO: copy the segment from the ELF file to its proper memory area */
            /* TODO: zeror the memory area [vaddr + file_sz, vaddr + mem_sz) */
#ifdef IA32_PAGE
            ... // 没有开启分页
#endif
        }
    }

    volatile uint32_t entry = elf->e_entry; // 头文件中指出的testcase起始地址, 应该是0x60000
    ... //现在不管
    return entry; // 返回testcase起始地址, 在init_cond()后面执行((void*)(void))eip());
}
```

Kernel装载程序

- 如何实现对ELF文件的装载
- 如何在Kernel中加入这一功能



Kernel执行loader()

- Kernel的行为
 - 从kernel/start/start.S开始

```
#ifndef IA32_SEG      // 没在include/config.h中define IA32_SEG, 因此编译这个分支

    .globl start
    start:
        jmp init

    # never return
```

kernel/start/start.S

Kernel执行loader()

- Kernel的行为

- 转到了kernel/src/main.c

kernel/src/main.c

```
void init() {
#ifdef IA32_PAGE
    ...
#endif

    /* Jump to init_cond() to continue initialization. */
    // need to plus the offset 0xc0000000 if using gcc-6, strange
#ifdef IA32_PAGE
    asm volatile("jmp *%0" : : "r"(init_cond + 0xc0000000));
#else
    asm volatile("jmp *%0" : : "r"(init_cond)); // 就执行了这一句
#endif

    /* Should never reach here. */
    nemu_assert(0);
}
```

Kernel执行loader()

kernel/src/main.c

```
void init_cond() {  
    ... //前面全跳过  
  
    /* Output a welcome message.  
     * Note that the output is actually performed only when  
     * the serial port is available in NEMU.  
     */  
    Log("Hello, NEMU world!"); // 输出一下子，其工作原理直到PA 4才去弄明白  
  
    ... //中间全跳过  
  
    /* Load the program. */  
    uint32_t eip = loader(); // 装载测试用例，PA 2-2就是要实现这个  
  
    ... //中间全跳过  
  
    /* Here we go! */  
    ((void (*)(void))eip)(); // 包装成函数指针，跑去执行测试用例  
}
```



```
uint32_t loader() {
    Elf32_Ehdr *elf;
    Elf32_Phdr *ph, *eph;

#ifdef HAS_DEVICE_IDE
    ... // 没有模拟硬盘
#else
    elf = (void *)0x0;    // 模拟内存0x0处是RAM Disk, 存放的就是testcase ELF file, 最开始的部分是ELF头
    Log("ELF loading from ram disk.");
#endif

    /* Load each program segment */
    ph = (void *)elf + elf->e_phoff; // 找到ELF文件中的程序头表
    eph = ph + elf->e_phnum;
    for(; ph < eph; ph++) { // 扫描程序头表中的各个表项
        if(ph->p_type == PT_LOAD) { // 如果类型是LOAD, 那么就去装载吧
            panic("Please implement the loader");
            /* TODO: copy the segment from the ELF file to its proper memory area */
            /* TODO: zeror the memory area [vaddr + file_sz, vaddr + mem_sz) */
#ifdef IA32_PAGE
            ... // 没有开启分页
#endif
        }
    }

    volatile uint32_t entry = elf->e_entry; // 头文件中指出的testcase起始地址, 应该是0x60000
    ... //现在不管
    return entry; // 返回testcase起始地址, 在init_cond()后面执行((void*)(void))eip());
}
```

PA执行命令的改变，加入Kernel

修改testcase/Makefile中的链接选项，make clean后重新编译

```
#LDFLAGS := -m elf_i386 -e start -Ttext=0x30000
```

```
LDFLAGS := -m elf_i386 -e start -Ttext=0x100000
```

```
$ make run-kernel
```

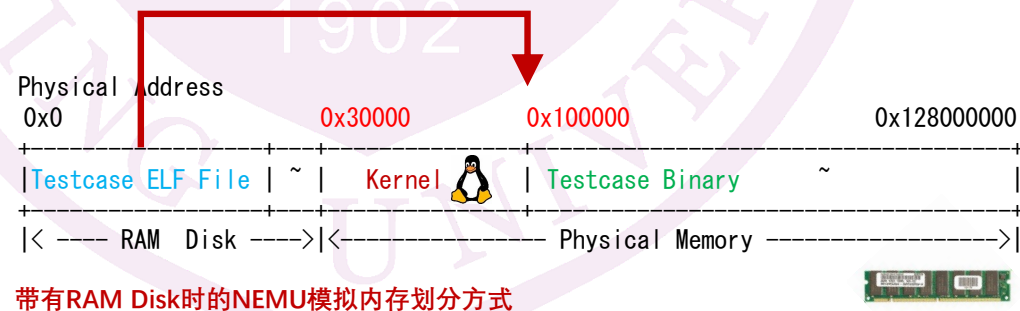
```
$ make test_pa-2-2
```

```
run-kernel: nemu
```

```
$(call git_commit, "run-kernel")
```

```
./nemu/nemu --kernel --testcase add
```

由Kernel解析测试用例的ELF文件并装载



PA 2-2 执行成功的标志

- 如果loader()实现正确，且指令实现都正确
 - 执行make run-kernel或make test_pa-2-2

```
icspa-public$ make clean
icspa-public$ make test_pa-2-2
...
./nemu/nemu --autorun --testcase string --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/string
nemu trap output: [src/main.c,82,init_cond] {kernel} Hello, NEMU world!
nemu trap output: [src/elf/elf.c,30,loader] {kernel} ELF loading from ram disk.
nemu: HIT GOOD TRAP at eip = 0x0010016a
NEMU2 terminated
./nemu/nemu --autorun --testcase hello-str --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/hello-str
nemu trap output: [src/main.c,82,init_cond] {kernel} Hello, NEMU world!
nemu trap output: [src/elf/elf.c,30,loader] {kernel} ELF loading from ram disk.
nemu: HIT GOOD TRAP at eip = 0x00100105
NEMU2 terminated
./nemu/nemu --autorun --testcase test-float --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/test-float
nemu trap output: [src/main.c,82,init_cond] {kernel} Hello, NEMU world!
nemu trap output: [src/elf/elf.c,30,loader] {kernel} ELF loading from ram disk.
nemu: HIT BAD TRAP at eip = 0x001000c8
NEMU2 terminated
```

除了test-float，其它都是HIT GOOD TRAP

实用小贴士

- 如果使用make test系列命令
 - 不会进入交互界面，调试有困难
 - 可以使用BREAK_POINT宏强制进入交互模式
 - 参考群里的ui.c，实现简易的x扫描内存命令

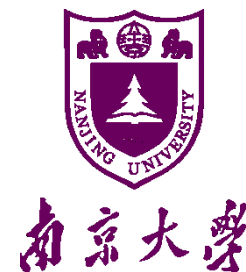
```
#ifndef IA32_SEG
```

```
.globl start  
start:
```

```
BREAK_POINT #每次一进Kernel就break，进入交互模式  
jmp init  
# never return
```

PA 2-1 Deadline

- 2020年11月11日24时（周三24时 = 周四0时）
- `make submit_pa-2-1`
- 在submit/目录下生成的压缩包上传到cslab系统备份窗口
- 实验报告提交到cslab实验报告提交窗口



PA 2-2到此结束

祝大家学习快乐，身心健康！

欢迎大家踊跃参加问卷调查

目录

- 认识ELF文件
- PA 2-2 - 程序的装载
 - > ELF文件程序头表的解析
- PA 2-3 - 调试器符号表解析
 - > ELF文件符号表的解析

符号表的解析

- 要解决的问题：

给定一个符号（如，全局变量）的名字，
返回其在内存中的值

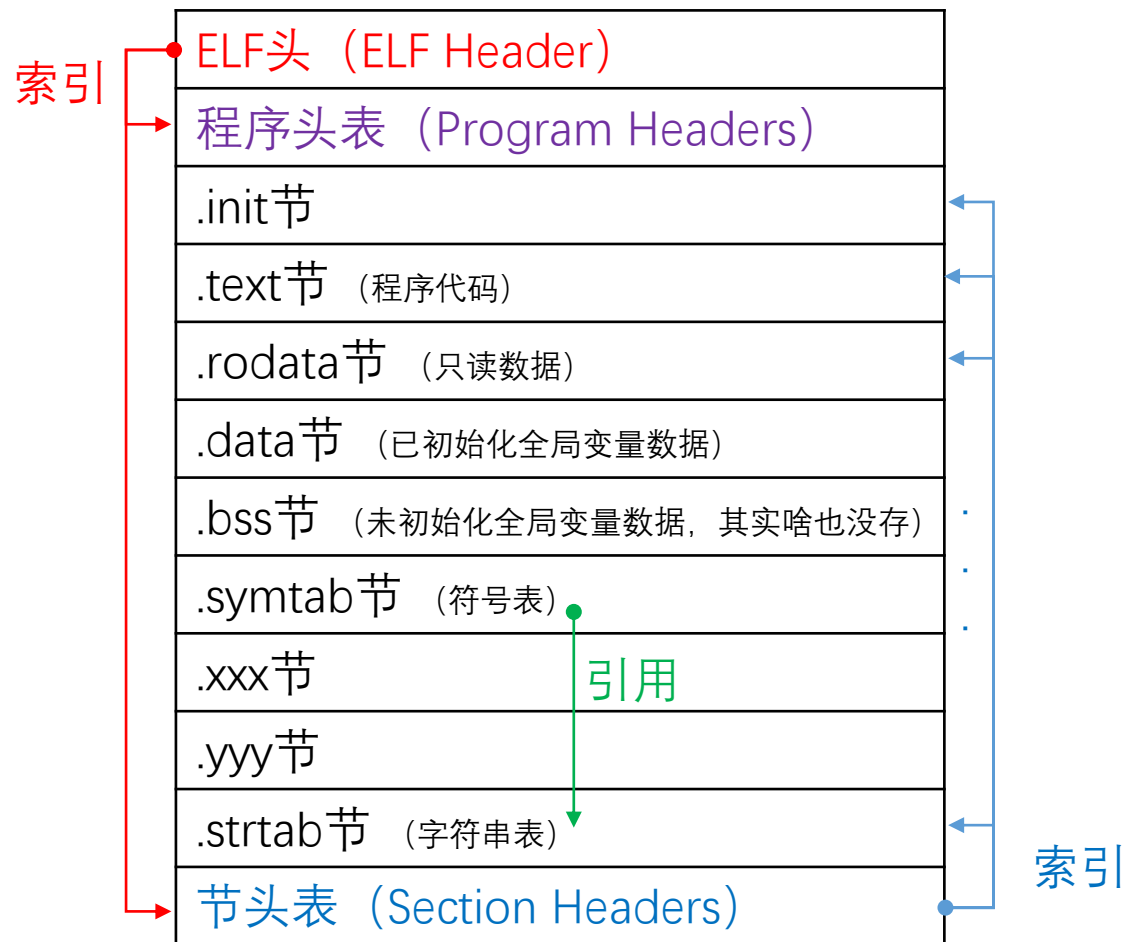
[testcase/src/add.c](#)

```
int test_data[] = {0, 1, 2, 0x7fffffff, 0x80000000, 0x80000001,
0xfffffffffe, 0xffffffffff};
```

NEMU中的交互式调试界面

```
(nemu) x 4 test_data
n = 4, expr = test_data
0x00032020:      0x00000000 0x00000001 0x00000002 0x7fffffff
```


符号表的解析



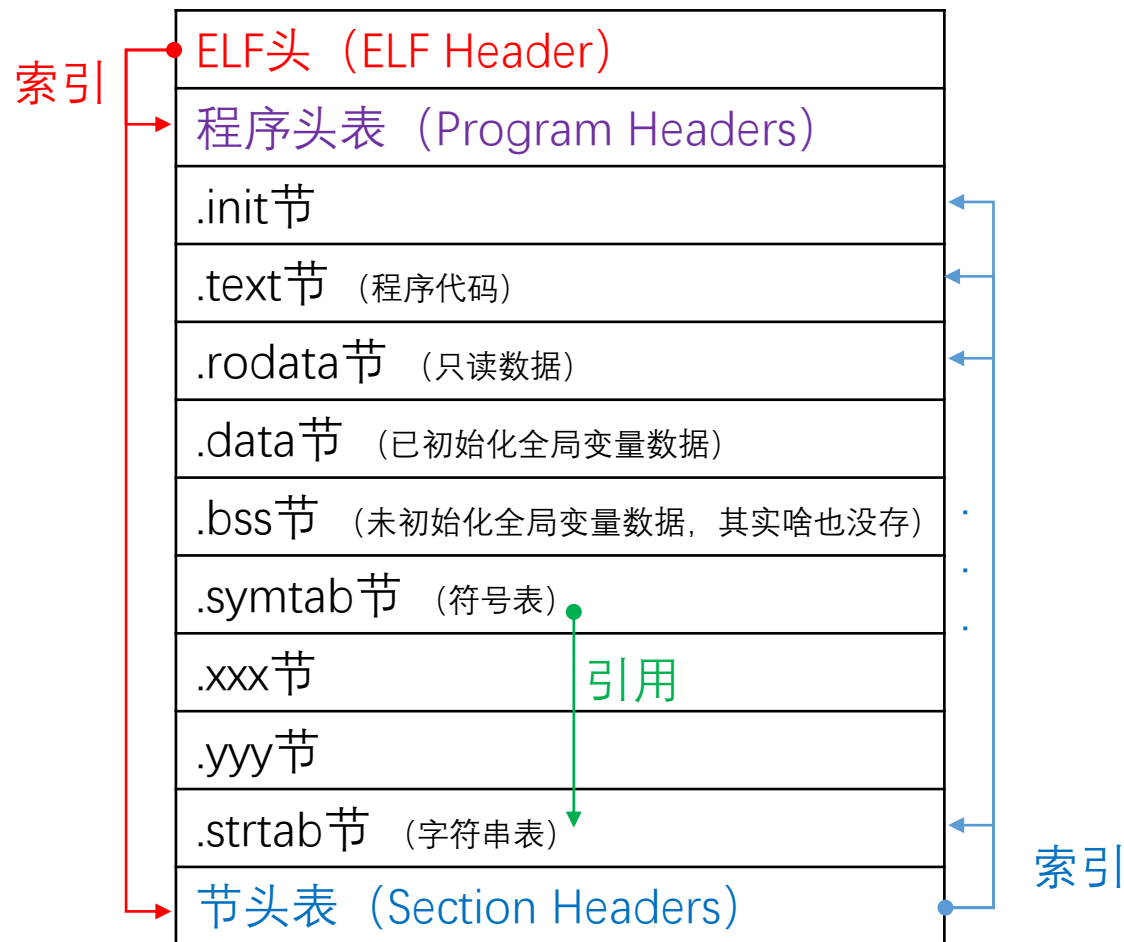
符号表解析程序

输入：ELF文件，带查询符号名

输出：符号在内存中的地址

1. 读入ELF头
2. 定位符号表
3. for 符号表中的每一项
4. if 该项名 == 带查询符号名
5. return 查找成功，符号的内存地址
6. end if
7. end for
8. return 查找失败

符号表的解析



符号表解析程序

输入: ELF文件, 带查询符号名

输出: 符号在内存中的地址

1. 读入ELF头
2. 定位符号表
3. for 符号表中的每一项
4. if 该项名 == 带查询符号名
5. return 查找成功, 符号的内存地址
6. end if
7. end for
8. return 查找失败

ELF头的编程解析

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: 0x30000
Start of program headers: 52 (bytes into file)
Start of section headers: 18276 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 3
Size of section headers: 40 (bytes)
Number of section headers: 15
Section header string table index: 14
```

节头表位置

节头表对应字符串表的索引

```
#define EI_NIDENT 16
```

```
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t e_type;
    uint16_t e_machine;
    uint32_t e_version;
    ElfN_Addr e_entry;
    ElfN_Off e_phoff;
    ElfN_Off e_shoff;
    uint32_t e_flags;
    uint16_t e_ehsize;
    uint16_t e_phentsize;
    uint16_t e_phnum;
    uint16_t e_shentsize;
    uint16_t e_shnum;
    uint16_t e_shstrndx;
} ElfN_Ehdr;
```

ELF头，它位于整个ELF文件最开始的地方。在32位Linux系统中，<elf.h>头文件中的Elf32_Ehdr数据结构与之对应。

解析节头表，找到符号表和字符串表

找到名为'.symtab'的符号表和名为'.strtab'的（对应节头表的）字符串表

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	00000000	00000000	00	AX	0	0	0
[1]	.text	PROGBITS	00030000	001000	0000d0	00	AX	0	0	1
[2]	.eh_frame	PROGBITS	000300d0	0010d0	000084	00	A	0	0	4
[3]	.got.plt	PROGBITS	00032000	002000	00000c	04	WA	0	0	4
[4]	.data	PROGBITS	00032020	002020	000120	00	WA	0	0	32
[5]	.comment	PROGBITS	00000000	002140	000026	01	MS	0	0	1
[6]	.debug_aranges	PROGBITS	00000000	002168	000040	00		0	0	8
[7]	.debug_info	PROGBITS	00000000	0021a8	000142	00		0	0	1
[8]	.debug_abbrev	PROGBITS	00000000	0022ea	0000d6	00		0	0	1
[9]	.debug_line	PROGBITS	00000000	0023c0	0000dc	00		0	0	1
[10]	.debug_str	PROGBITS	00000000	00249c	001a37	01	MS	0	0	1
[11]	.shstrtab	PROGBITS	00000000	000000	00005f9	00		0	0	1
[12]	.symtab	SYMTAB	00000000	00044cc	0000190	10		13	15	4
[13]	.strtab	STRTAB	00000000	000465c	0000078	00		0	0	1
[14]	.shstndx	STRTAB	00000000	00046d4	0000090	00		0	0	1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
p (processor specific)

There are no section groups in this file.

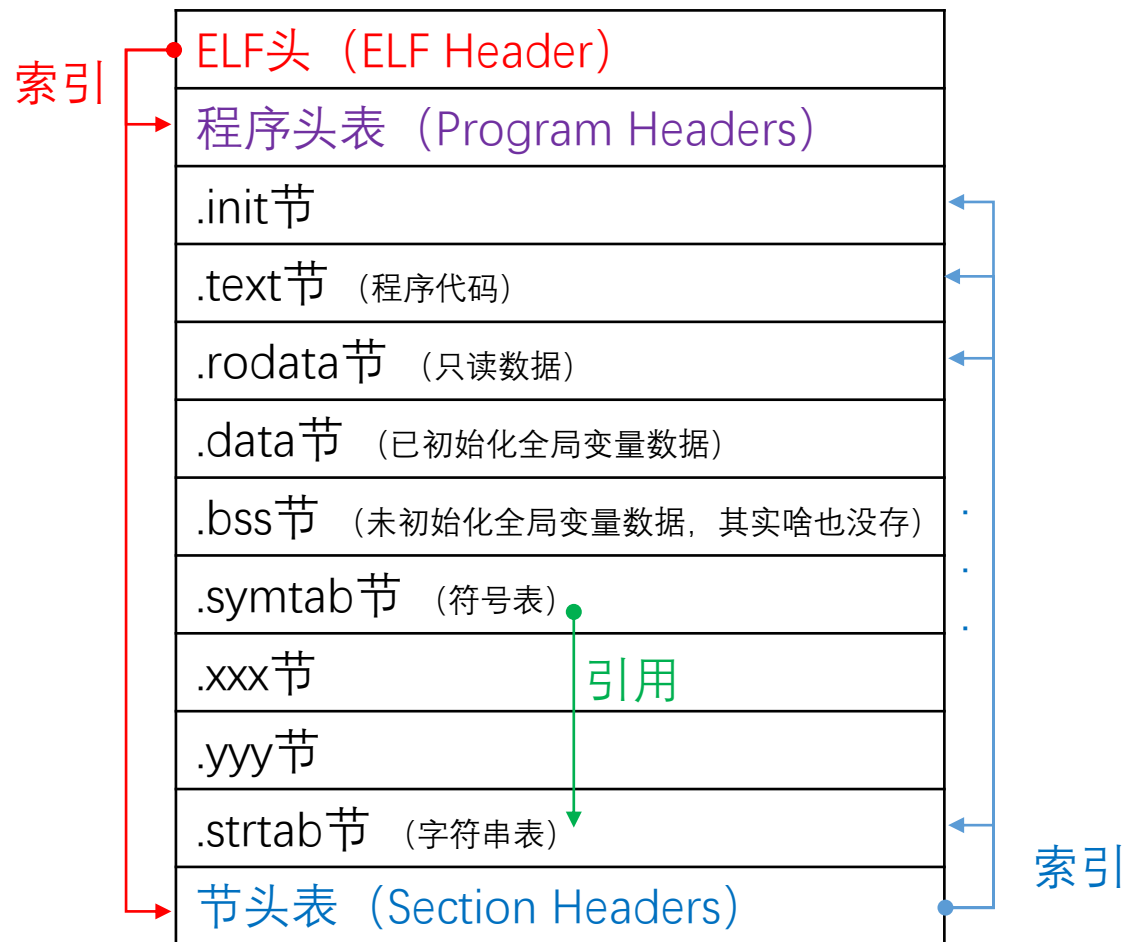
<elf.h>

```
typedef struct {
    uint32_t sh_name;
    uint32_t sh_type;
    uint32_t sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off sh_offset;
    uint32_t sh_size;
    uint32_t sh_link;
    uint32_t sh_info;
    uint32_t sh_addralign;
    uint32_t sh_entsize;
} Elf32_Shdr;
```

这里的sh_name只是一个索引值，只有用该索引值去查了.shstrtab之后才能得到.text, .data这样的字符串，而.shstrtab在哪里呢？ELF Header中的e_shstrndx变量告诉我们，它在Section Headers数组中的第14项

具体技术和符号表+字符串表解析方式一样

符号表的解析



符号表解析程序

输入：ELF文件，带查询符号名

输出：符号在内存中的地址

1. 读入ELF头
2. 定位符号表
3. **for 符号表中的每一项**
4. **if 该项名 == 带查询符号名**
5. **return 查找成功，符号的内存地址**
6. **end if**
7. **end for**
8. **return 查找失败**

符号表的解析

- 通过节头表定位'.symtab'节在ELF文件中的位置
- 符号表也是个数组，其类型为Elf32_Sym

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[12]	.symtab	SYMTAB	00000000	0044cc	000190	10		13	15	4

<elf.h>

Symbol table '.symtab' contains 25 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00030000	0	SECTION	LOCAL	DEFAULT	1	
2:	000300d0	0	SECTION	LOCAL	DEFAULT	2	
3:	00032000	0	SECTION	LOCAL	DEFAULT	3	
4:	00032020	0	SECTION	LOCAL	DEFAULT	4	
5:	00000000	0	SECTION	LOCAL	DEFAULT	5	
6:	00000000	0	SECTION	LOCAL	DEFAULT	6	
7:	00000000	0	SECTION	LOCAL	DEFAULT	7	
8:	00000000	0	SECTION	LOCAL	DEFAULT	8	
9:	00000000	0	SECTION	LOCAL	DEFAULT	9	
10:	00000000	0	SECTION	LOCAL	DEFAULT	10	
11:	00000000	0	SECTION	LOCAL	DEFAULT	11	
12:	00000000	0	FILE	LOCAL	DEFAULT	ABS	add.c
13:	00000000	0	FILE	LOCAL	DEFAULT	ABS	
14:	00032000	0	OBJECT	LOCAL	DEFAULT	3	__GLOBAL_OFFSET_TABLE__
15:	000300c8	0	FUNC	GLOBAL	HIDDEN	1	__x86.get_pc_thunk.ax
16:	00030005	32	FUNC	GLOBAL	DEFAULT	1	add
17:	000300cc	0	FUNC	GLOBAL	HIDDEN	1	__x86.get_pc_thunk.bx
18:	00032140	0	NOTYPE	GLOBAL	DEFAULT	4	__bss_start
19:	00030025	163	FUNC	GLOBAL	DEFAULT	1	main
20:	00032040	256	OBJECT	GLOBAL	DEFAULT	4	ans
21:	00032140	0	NOTYPE	GLOBAL	DEFAULT	4	_edata
22:	00032140	0	NOTYPE	GLOBAL	DEFAULT	4	_end
23:	00030000	0	NOTYPE	GLOBAL	DEFAULT	1	start
24:	00032020	32	OBJECT	GLOBAL	DEFAULT	4	test_data

```
typedef struct {
    uint32_t    st_name;
    Elf32_Addr  st_value;
    uint32_t    st_size;
    unsigned char st_info;
    unsigned char st_other;
    uint16_t    st_shndx;
} Elf32_Sym;
```

st_name – 符号名称，对应strtab中的偏移量
st_value – 符号的地址
st_size – 符号所占用的字节数
st_info – 包含了Type信息，man elf查看说明

testcase/src/add.c

```
int test_data[] = {0, 1, 2,
0x7fffffff, 0x80000000,
0x80000001, 0xffffffff,
0xffffffff};
```

readelf -s add

符号表的解析

- 符号表(.symtab)与字符串表(.strtab)结合, 获取符号的字符串形式的名称

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[13]	.strtab	STRTAB	00000000	00465c	000078	00		0	0	1

hexdump -C add

00004650	20 20 03 00 20 00 00 00	11 00 04 00 00 61 64 64	add
00004660	2e 63 00 5f 47 4c 4f 42	41 4c 5f 4f 46 46 53 45	.c._GLOBAL_OFFSE	
00004670	54 5f 54 41 42 4c 45 5f	00 5f 5f 78 38 36 2e 67	T_TABLE.__x86.g	
00004680	65 74 5f 70 63 5f 74 68	75 6e 6b 2e 61 78 00 61	et_pc_thunk.ax.a	
00004690	64 64 00 5f 5f 78 38 36	2e 67 65 74 5f 70 63 5f	dd.__x86.get_pc_	
000046a0	74 68 75 6e 6b 2e 62 78	00 5f 5f 62 73 73 5f 73	thunk.bx.__bss_s	
000046b0	74 61 72 74 00 6d 61 69	6e 00 61 6e 73 00 5f 65	tart.main.ans._e	
000046c0	64 61 74 61 00 5f 65 6e	64 00 74 65 73 74 5f 64	data._end.test_d	
000046d0	61 74 61 00 00 2e 73 79	6d 74 61 62 00 2e 73 74	ata...symtab..st	

Symbol table '.symtab' contains 25 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
24:	00032020	32	OBJECT	GLOBAL	DEFAULT	4	6d

```
typedef struct {
    uint32_t  st_name;
    Elf32_Addr st_value;
    uint32_t  st_size;
    unsigned char st_info;
    unsigned char st_other;
    uint16_t   st_shndx;
} Elf32_Sym;
```

符号表的解析

.strtab在ELF文件中起始位置 + 符号表项.st_name => 字符串

符号表解析程序

输入：ELF文件，带查询符号名
输出：符号在内存中的地址

1. 读入ELF头
2. 定位符号表
3. for 符号表中的每一项
4. **if 该项名 == 带查询符号名**
5. return 查找成功，符号的内存地址
6. end if
7. end for
8. return 查找失败

NEMU相关代码

[nemu/src/monitor/elf.c](#)

```
uint32_t look_up_symtab(char *sym, bool *success) {
    int i;
    for(i = 0; i < nr_symtab_entry; i++) {
        uint8_t type = ELF32_ST_TYPE(symtab[i].st_info);
        if((type == STT_FUNC || type == STT_OBJECT) &&
            strcmp(strtab + symtab[i].st_name, sym) == 0) {
            *success = true;
            return symtab[i].st_value;
        }
    }

    *success = false;
    return 0;
}
```

NEMU相关代码

nemu/src/monitor/elf.c

找到名为.symtab的符号表和
名为.strtab的字符串表

```
/* Load section header table 读取节头表 */
uint32_t sh_size = elf->e_shentsize * elf->e_shnum;
Elf32_Shdr *sh = malloc(sh_size);
fseek(fp, elf->e_shoff, SEEK_SET);
fread(sh, sh_size, 1, fp);

/* Load section header string table 读取节头表对应的字符串表 */
char *shstrtab = malloc(sh[elf->e_shstrndx].sh_size);
fseek(fp, sh[elf->e_shstrndx].sh_offset, SEEK_SET);
fread(shstrtab, sh[elf->e_shstrndx].sh_size, 1, fp);

int i;
for(i = 0; i < elf->e_shnum; i++) { /* 扫描节头表 */
    if(sh[i].sh_type == SHT_SYMTAB && /* 这一步和解析符号名称时的操作一样，等一下细讲
                                     strcmp(shstrtab + sh[i].sh_name, ".symtab") == 0) {
        /* Load symbol table from exec_file 得到符号表 */
        symtab = malloc(sh[i].sh_size);
        fseek(fp, sh[i].sh_offset, SEEK_SET);
        fread(symtab, sh[i].sh_size, 1, fp);
        nr_symtab_entry = sh[i].sh_size / sizeof(symtab[0]);
    }
    else if(sh[i].sh_type == SHT_STRTAB &&
            strcmp(shstrtab + sh[i].sh_name, ".strtab") == 0) {
        /* Load string table from exec_file 得到符号表对应的字符串表 */
        strtab = malloc(sh[i].sh_size);
        fseek(fp, sh[i].sh_offset, SEEK_SET);
        fread(strtab, sh[i].sh_size, 1, fp);
    }
}
```

符号表的解析

- 符号表解析了有啥用？
- 如果你想写一个链接器
 - 可以将处于不同.o文件中的全局变量或函数的调用和定义通过内存地址联系到一起
 - `readelf -s nemu/src/cpu/decode/opcode.o`
 - `readelf -s nemu/src/cpu/instr/mov.o`

Symbol table '.symtab' contains 165 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
...							
149:	000002a0	176	FUNC	GLOBAL	DEFAULT	39	mov_i2rm_b
151:	00000350	176	FUNC	GLOBAL	DEFAULT	39	mov_i2rm_v
...							

mov.o

opcode.o

如果发现符号表中有多个Type为FUNC或OBJECT，Bind类型为GLOBAL，其Ndx都显示在某一个section中被定义了的符号具有同样的名字：

multiple definition of xxx

去掉static void instr_execute_2op()前面的static就能够触发（比如尝试mov.c和sar.c，把static去掉），观察一下对应的符号表，是不是有什么变化？

Symbol table '.symtab' contains 249 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
...							
223:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	mov_i2r_b
224:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	mov_i2r_v
...							

符号表的解析

- 符号表解析了有啥用？
- 对于NEMU来说
 - 你可以使用 `x test_data` 来查看 `test_data` 的起始地址
 - 再使用 `x 起始地址+offset` 来查看 `test_data` 的内容
 - 也可以使用 `x *(test_data + offset)` 来查看 `test_data` 的内容
 - 你也可以使用 `b main` 来在 `main` 函数开始处设置断点
- 在NEMU中使用上述功能涉及对表达式求值功能的实现
 - 相应教程：看教程PA 2-3部分
 - 代码：nemu/src/monitor/expr.c
 - 我们下次课再讲