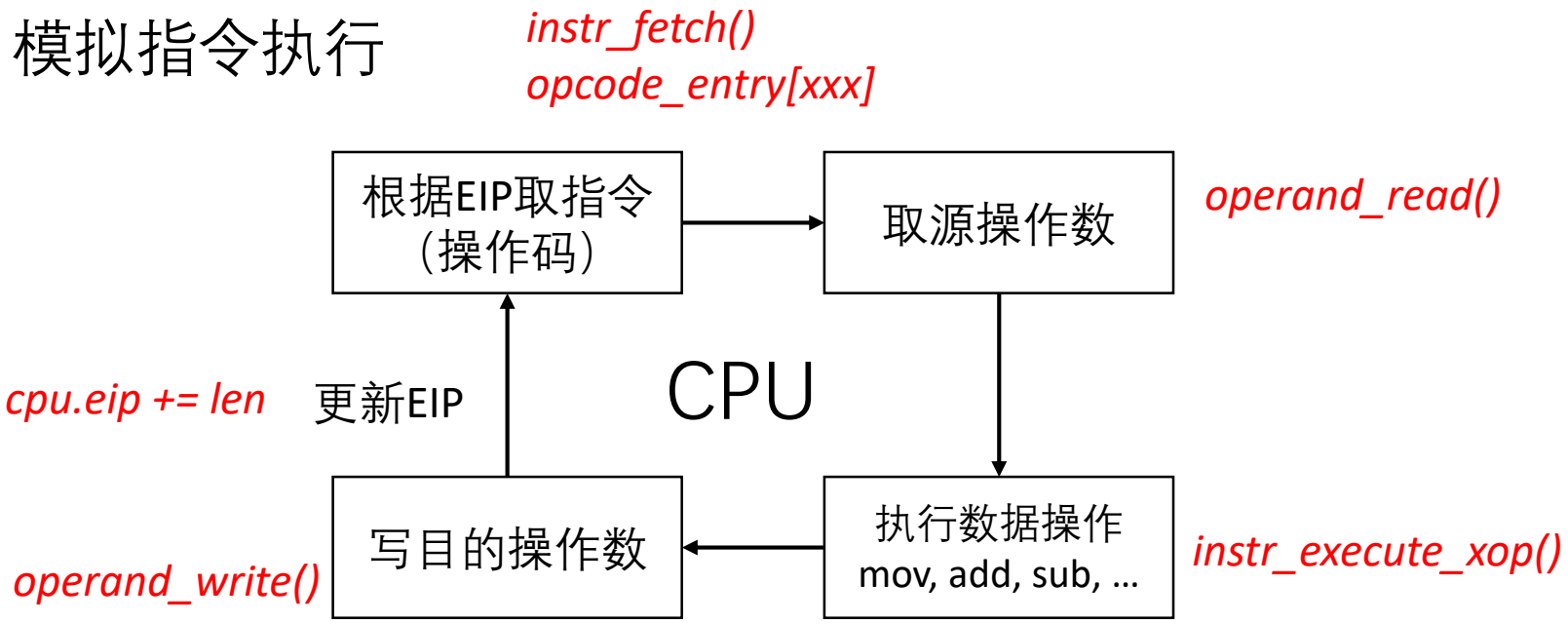
计算机系统基础
Programming Assignment

PA 2 程序的执行（第二课）
——PA 2-2 程序的装载
PA 2-3 调试器符号表解析

2017年10月20日

前情提要

- PA 1
  - ALU + FPU 基本运算功能
- PA 2-1
  - 模拟指令执行



先把指令依次在内存中排好，给EIP赋一个初始值，指向第一条指令，CPU就可以循环执行每一条指令了

前情提要

- 在实现的过程中出现了许多稀奇古怪的bug
- 基本的心理发展过程
  - 第一阶段：**不可能是我的错！**一定是框架代码、编译器、操作系统、虚拟机、CPU……里有bug！
  - 第二阶段：嗯……似乎这里有一点小问题，但是**不至于**吧~
  - 第三阶段：**当初这代码怎么能跑起来的！！！！？？？？**
- Debugging是码农们一生都要面对的问题
  - 基本过程
    - 重现错误（成功一半）：再跑一次、构造新的有针对性的测试用例……
    - 分离和定位root-cause：单步执行、断点……
    - 查看和分析：assert、printf……
    - 总结：不容易犯错的编码方式、构造对测试友好的代码……
  - 踩遍所有的坑，成就伟大程序员

补充说明

- 关于git记录过大的问题
  - 参见课程群中《关于文件过大》（sandhill.pdf）一文的说明

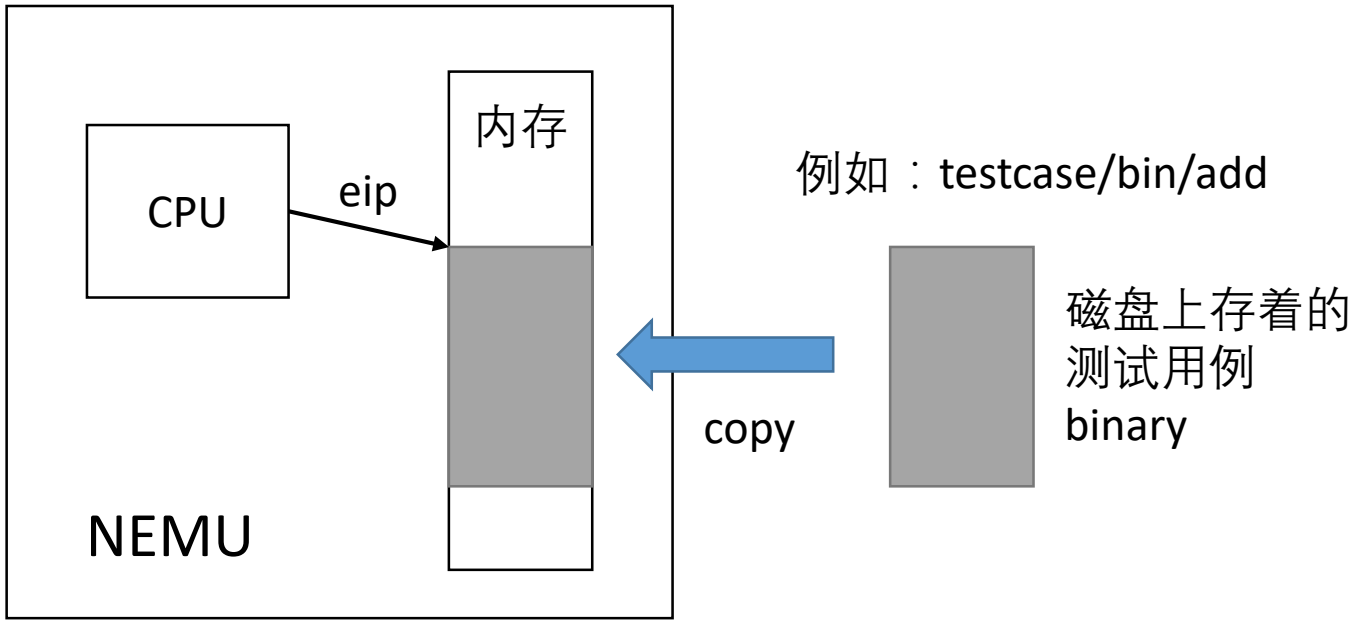
PA 2-2 程序的装载 & PA 2-3 调试器

- ELF文件的装载
- 符号表的解析

PA 2-2 程序的装载 & PA 2-3 调试器

- ELF文件的装载
- 符号表的解析

原先的NEMU是怎么装载程序的？



1. 把磁盘上存着的测试用例binary原封不动拷贝到NEMU的模拟内存里（从0x30000开始的地方）
2. 把CPU的eip初始化为0x30000
3. 模拟CPU通过执行exec()开始执行测试用例的binary

原先的NEMU是怎么装载程序的？

- 具体过程
  - testcase/Makefile

一个Makefile一般长这个样子

```
# 一堆变量赋值（:=与=的区别自行上网搜索）

xxx := yyy

# 一堆目标和规则

目标: 依赖
<TAB> 规则
```

可能是另外一组规则所要达成的目标（比如可执行文件add作为目标，add.o就是其依赖，而add.o又作为其它规则的目标，从add.c产生）

（在依赖被满足的前提下，通过怎样的规则来实现目标）

可能是个文件（add），也可能是个虚的目标（clean）；目标文件的依赖如果没有发生改变，那么不需要动用规则去产生新的目标文件

在命令行键入make之后

1. 搜索当前文件夹下的Makefile（还有另外三种可能的文件命名方式，自行搜索）
2. 如果没有在make后面跟目标名称，则默认第一个目标为最终目标，否则以目标名称对应的目标为最终目标（make clean）

```
# 一堆变量赋值（:=与=的区别自行上网搜索）
CC := gcc
LD := ld
CFLAGS := ……
LDFLAGS := -m elf_i386 -e start -Ttext=0x30000 # make run | make test
SFILES := $(shell find src/ -name "*.S")
CFILES := $(shell find src/ -name "*.c")
SOBJS := $(SFILES:.S=.o)
COBJS := $(CFILES:.c=.o)
SBINS := $(SFILES:.S=)
CBINS := $(CFILES:.c=)
# 一堆目标和规则

testcase: start.o $(SOBJS) $(COBJS) $(SBINS) $(CBINS)

%.o: %.c        这里就是gcc没啥好说的
	$(CC) $(CFLAGS) -c -o $@ $<

%.o: %.S
	$(CC) $(CFLAGS) -c -o $@ $<

%: %.o
	翻译过来（以add为例）：
	ld –m elf_i386 –e start –Ttext=0x30000 add src/start.o add.o ../include/newlib/libc.a
	$(LD) $(LDFLAGS) -o $@ src/start.o $< ../include/newlib/libc.a
	cp $@ $(addprefix bin/,$(notdir $@))  把testcase/src底下的可执行文件（如add）拷贝到testcase/bin底下去
	objcopy -S -O binary -B i386 $(addprefix bin/,$(notdir $@)) $(addprefix bin/,$(notdir $@)).img
```

链接时代码段起始位置设为0x30000

命令行在testcase/底下执行make就执行testcase目标，它依赖于一堆.o和可执行文件（没有后缀）

依赖

把这三个链接到一起得到add，代码段起始地址0x30000

又依赖（start.o的依赖没画，看Makefile）

这里采用objcopy把可执行文件中所有和执行不相干的内容剥离(-S)后，输出得到一个在testcase/bin文件夹底下，binary形式（-O binary）的.img文件，这个就是可执行文件的镜像

原先的NEMU是怎么装载程序的？

- 具体过程
  - 对比一下testcase/bin/add和testcase/bin/add.img

$ objdump –d add

```
00030000 <start>:
   30000:  e9 20 00 00 00        jmp    30025 <main>

00030005 <add>:
   30005:  55                    push   %ebp
   30006:  89 e5                 mov    %esp,%ebp
   30008:  83 ec 10              sub    $0x10,%esp
   3000b:  e8 b8 00 00 00        call   300c8 <…>
   30010:  05 f0 1f 00 00        add    $0x1ff0,%eax
   30015:  8b 55 08              mov    0x8(%ebp),%edx
   30018:  8b 45 0c              mov    0xc(%ebp),%eax
   3001b:  01 d0                 add    %edx,%eax
   3001d:  89 45 fc             mov    %eax,-0x4(%ebp)
   30020:  8b 45 fc              mov    -
0x4(%ebp),%eax
   30023:  c9                    leave
   30024:  c3                    ret


…
```

$ hexdump add.img

```
0000000 20e9 0000 5500 e589 ec83 e810 00b8 0000
0000010 f005 001f 8b00 0855 458b 010c 89d0 fc45
0000020 458b c9fc 55c3 e589 8353 10ec 9be8 0000
0000030 8100 cfc3 001f c700 f045 0000 0000 45c7
0000040 00f8 0000 eb00 c751 f445 0000 0000 3deb
0000050 458b 8bf4 8394 0020 0000 458b 8bf8 8384
0000060 0020 0000 5052 9ae8 ffff 83ff 08c4 c189

…
```

字节按小端序，所以add.img里面从0x0位置开始的内容对照可执行文件add是什么？

原先的NEMU是怎么装载程序的？

- 具体过程
  - 转到NEMU，在执行make run和make test时
  - 复习上节课讲到的NEMU运行测试用例的过程
    - 那个时候举的例子是mov，和add以及其它任何一个testcase没有什么不同

NEMU运行测试用例

- 从make run说起

Makefile

```
nemu:
        $(call git_commit, "nemu")
        cd nemu && make

run: nemu do_not_call_me_testcase
        $(call git_commit, "run")
        ./nemu/nemu -run mov
```

目标

前提依赖（编译nemu和testcase）

1. 打上git记录
2. 执行./nemu/nemu -run mov
   - 其中mov是测试用例的名称，对应testcase/src/里面测试用例源文件的文件名

NEMU运行测试用例

- 从make run说起
- 执行./nemu/nemu -run mov

nemu/src/main.c

```
int main(int argc, char* argv[]) {

        if(argc == 1) {  reg_test();  alu_test();  fpu_test();  return 0; }

        /* Read the arguments */
        if(argc == 3) {
                if(strcmp(argv[1], "-run")) {
                        printf("Error: %s %s %s\n", argv[0], argv[1], argv[2]);
                        printf("Usage: nemu -run <testcase>\n");
                        return 0;
                }
                strcpy(image_path, "./testcase/bin/");
                strcat(image_path, argv[2]);
                strcat(image_path, ".img");
                strcpy(elf_path, "./testcase/bin/");
                strcat(elf_path, argv[2]);
                single_run();

        }
        …
}
```

.img内存镜像路径

elf文件路径（现在不管）

NEMU运行测试用例

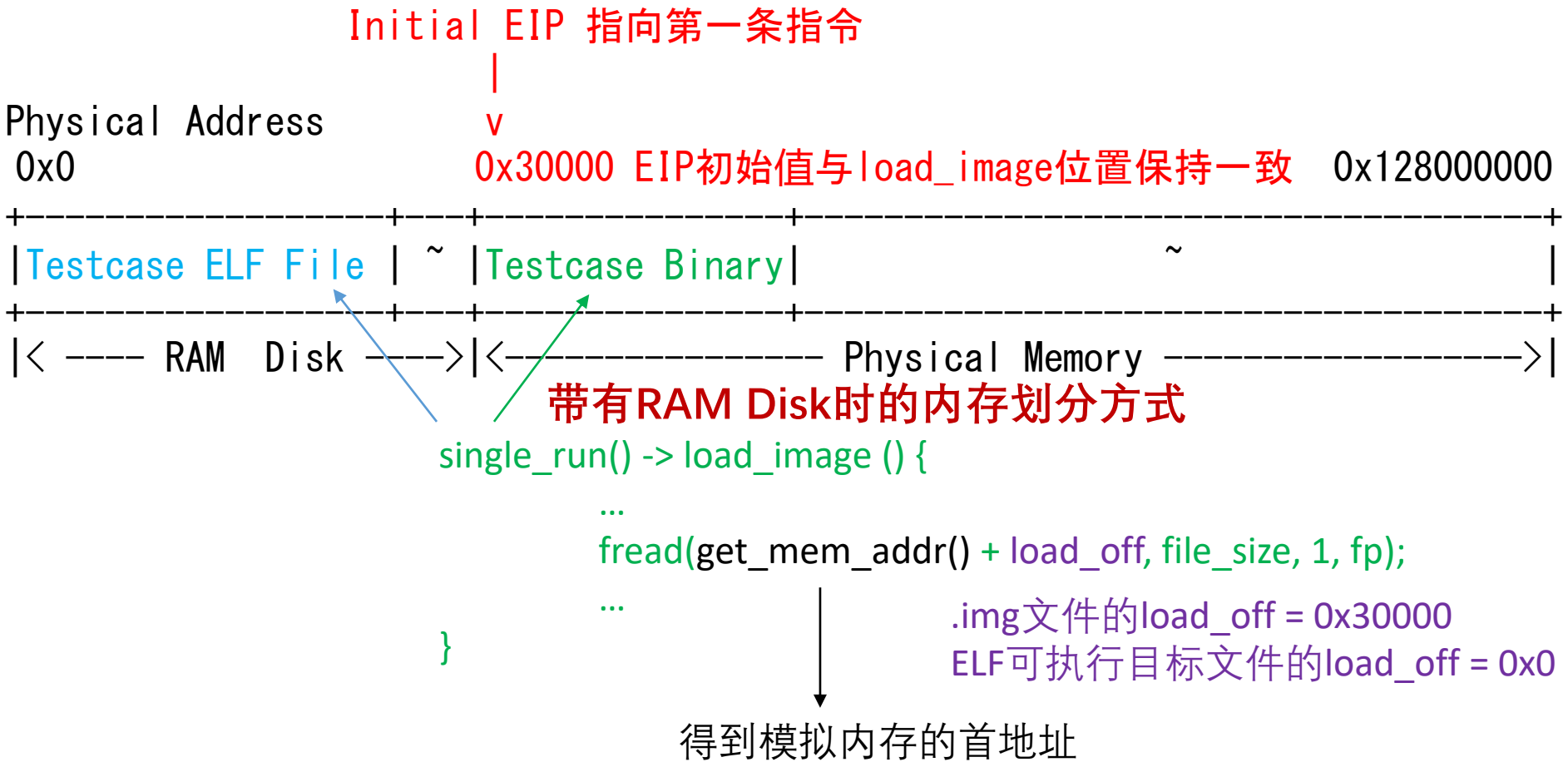
- 从make run说起
- 执行./nemu/nemu -run mov

nemu/src/main.c

```
static void single_run() {
        …
        restart(INIT_EIP);    INIT_EIP = LOAD_OFF = 0x30000
        load_image(image_path, LOAD_OFF);
        …
        load_image(elf_path, 0);
        …
        ui_mainloop(autorun);
}
```

NEMU运行测试用例

- 从make run说起
- 执行./nemu/nemu -run mov



ELF可执行目标文件

- 采用.img镜像文件加载可执行binary的方式看似完美无缺，但有时候却有一些问题
  - 好的方面
    - 装载过程简单直接，对裸机很友好（现代机器刚开机时也要通过这种方式加载一些最基本的引导程序）
  - 不好的方面
    - 存储效率问题
      - 除了必要的代码和数据，其它的一些运行时内存的内容，如栈、未初始化的全局变量，是不是要在镜像中保留相应的位置？
      - 这个好解决，不保存就行了
    - 链接困难
      - 最终的可执行目标文件很可能是由多个目标文件拼接（链接）而成的
        - 每个.c文件都会通过gcc产生一个.o文件（看看nemu在make的时候产生了多少个.o文件）
        - 最后通过ld把它们链接到一起得到最终的可执行目标文件（所有.o文件合到一起才得到了nemu这个可执行文件）
        - 不同的.o文件可能会共享一些全局变量和函数（instr/mov.c中的函数在cpu.c中被调用，cpu.c中的cpu全局变量被好多.c所使用）
        - 如果每个.o文件都是以.img镜像的方式来存储，拼接的时候困难重重，因此需要一个好的格式来描述目标文件

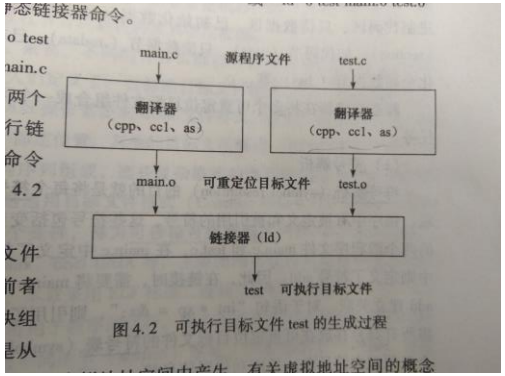


图 4.2 可执行目标文件 test 的生成过程

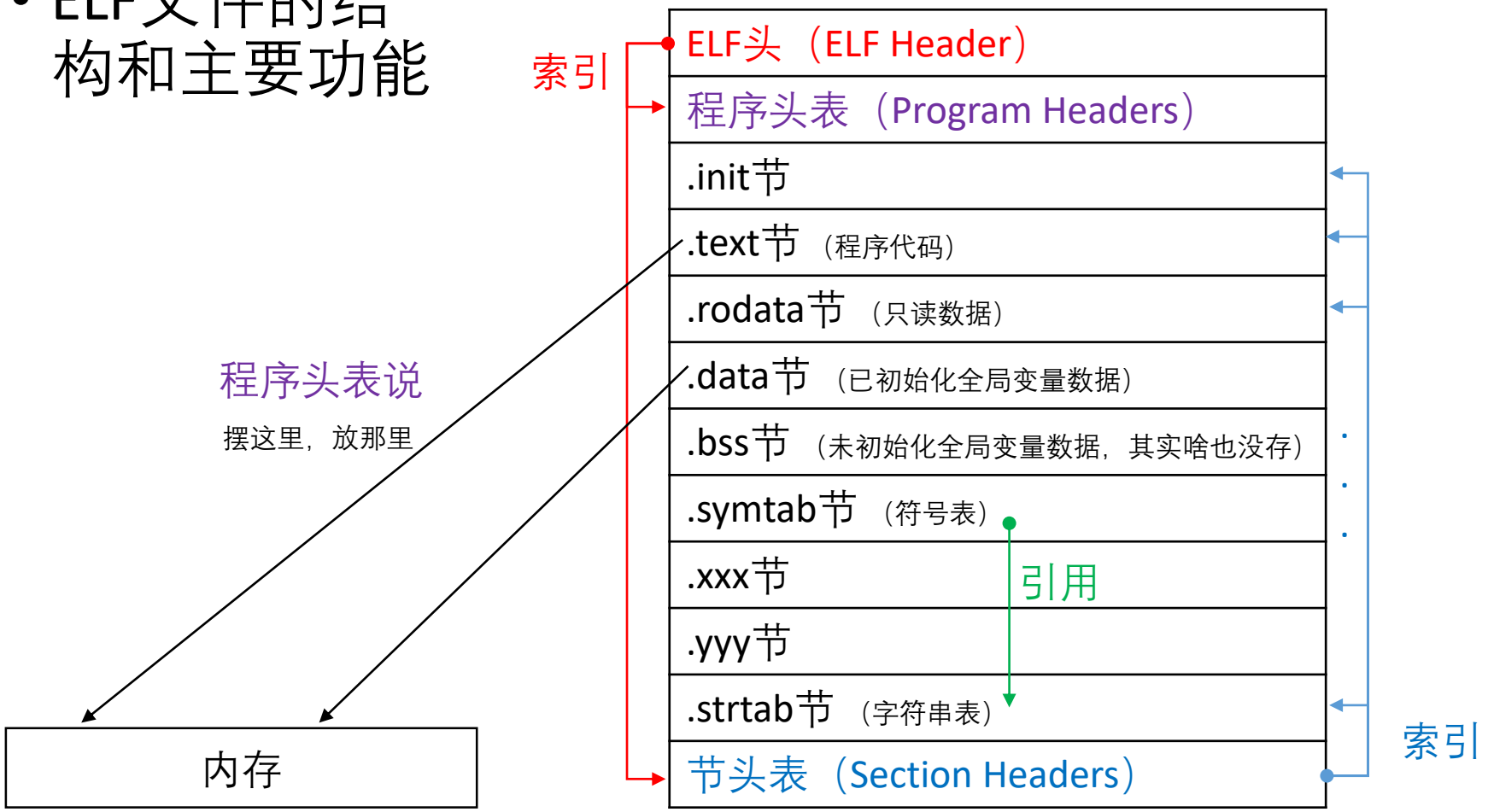
课本pg. 167

ELF可执行目标文件

- 一个好的目标文件应当包括的内容
  - 面向执行
    - 执行环境的要求：什么操作系统、什么架构……
    - 执行的内容：代码、数据的内容及其在内存中的位置
  - 面向链接
    - 函数名和全局变量所对应的符号
    - 可重定位信息
      - 拼接后有些跳转地址是不是要修改、全局变量的地址是不是要修改？
      - 用objdump观察nemu在make后产生的各个.o文件，似乎都是从0x0地址开始的，显然在链接时要把不同的.o文件的内容重新定位再拼接到一起
  - 以及相关的一些调试信息
  - 采用相对统一的格式存储.o、可执行目标文件、以后要被其它项目使用的库文件
- 在GNU/Linux系统中，采用可执行可链接格式（Executable and Linkable Format, ELF）来存储目标文件

ELF可执行目标文件

- ELF文件的结构和主要功能



ELF可执行目标文件

- 百闻不如一见
  - 以testcase/bin/add为例：readelf –a add | less
  - 出来好多内容，一块块地来看

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 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可执行目标文件

PA 2-2程序装载的核心内容

- 百闻不如一见
  - 以testcase/bin/add为例：readelf –a add | less
  - 出来好多内容，一块块地来看

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 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可执行目标文件

PA 2-2程序装载的核心内容

- 百闻不如一见
  - 以testcase/bin/add为例：readelf –a add | less
  - 出来好多内容，一块块地来看

通过ElfN_Ehdr中的e_phoff可以找到程序头表的首地址，而所谓的程序头表，在代码中就是Elf32_Phdr类型的一个数组，包含e_phnum个数组元素。数组中的每一项：Type – 该程序段的类型；Offset – 该程序段在ELF文件中的偏移量；VirtAddr – 该程序段在内存中的虚拟地址，此时等于PhysAddr，以后也不用看PhysAddr；FileSiz - 该程序段在文件中所占用的字节数；MemSiz – 该程序段在内存中所占用的字节数；其它。

```
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
```

看这个mapping，对比各个segment的Offset和Section Headers里面对应Section的Off

单独查看程序头表readelf –l add | less

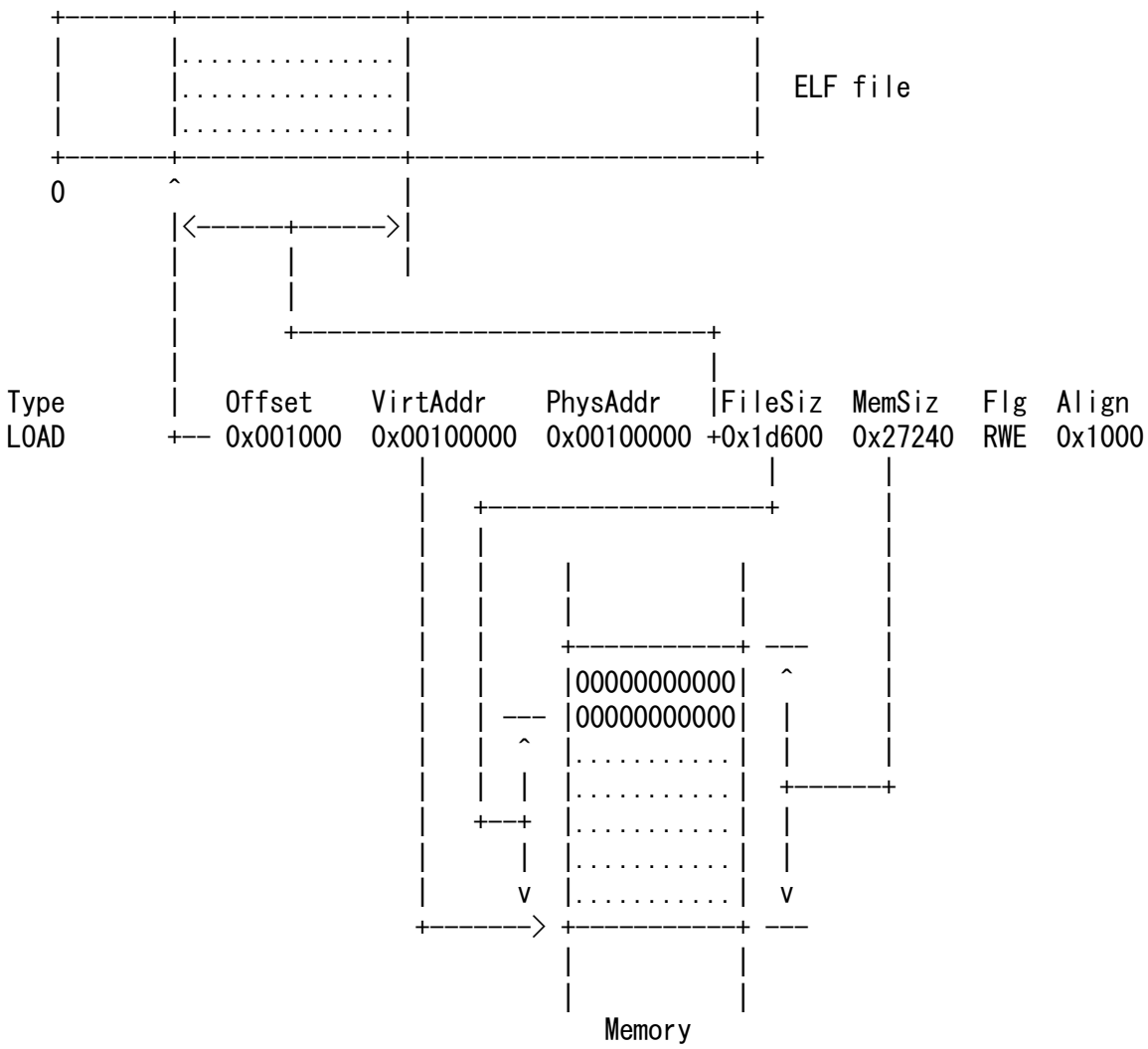
<elf.h>

```
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;
```

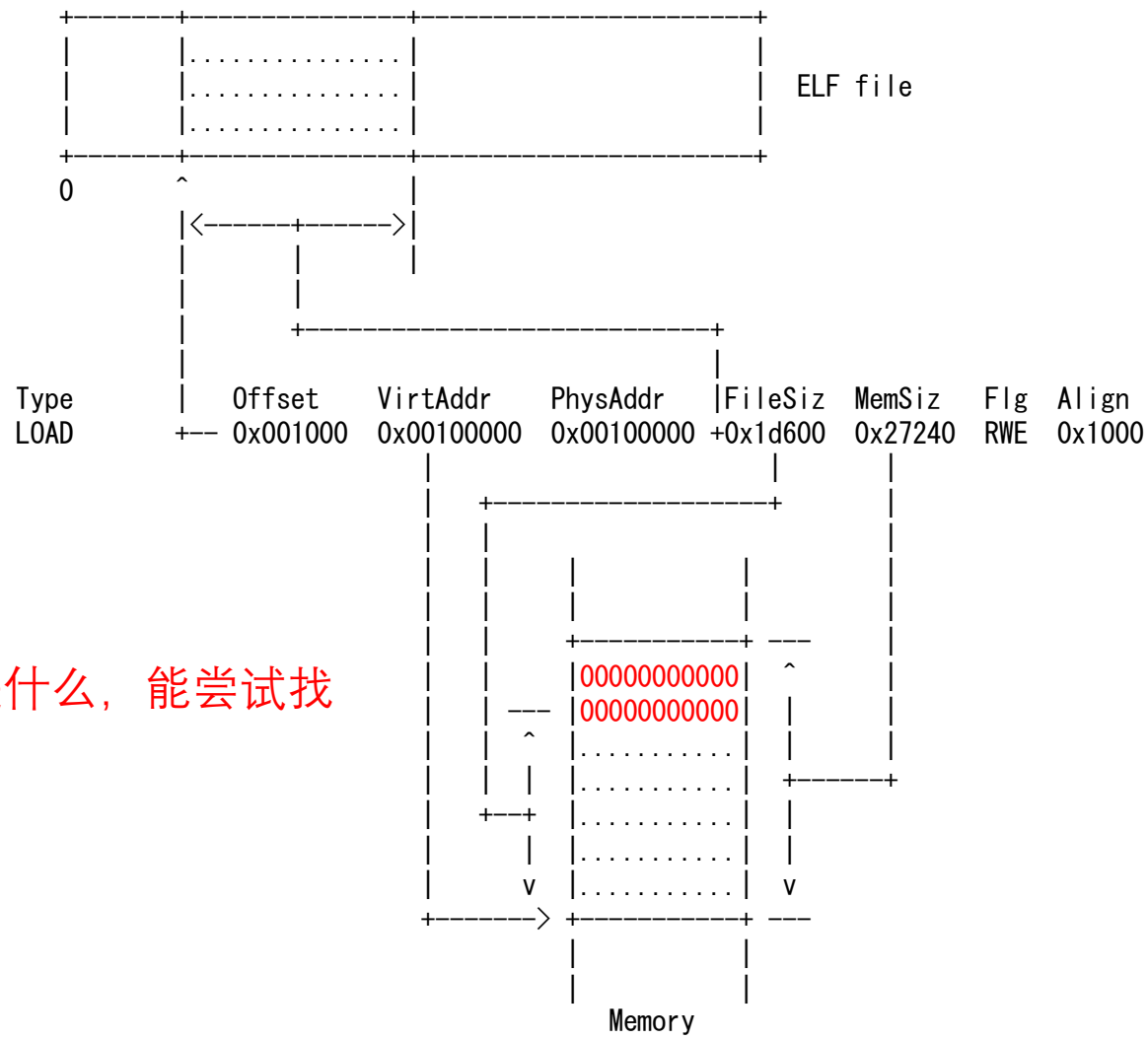
装载ELF可执行文件

- 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VirtAddr + [FileSiz, MemSiz]的区域清零

装载ELF可执行文件


ELF file
0
Type
LOAD
Offset
0x001000
VirtAddr
0x00100000
PhysAddr
0x00100000
FileSiz
0x1d600
MemSiz
0x27240
Flg
RWE
Align
0x1000
00000000000
00000000000
Memory

装载ELF可执行文件



这一块到底是什么，能尝试找出个例子吗？

装载ELF可执行文件

- 简单步骤
  - 一、读入ELF文件，前面若干字节解释为Elf32_Ehdr（强制类型转换即可）
  - 二、读Elf32_Ehdr中的e_phoff和e_phnum，得到程序头表在ELF文件中的起始地址和表项数
  - 三、循环读取每一个表项
    - 查看其type，如果是LOAD类型（LOAD类型在程序中等于几？通过man elf进行查看）
      - 则执行上一页中的拷贝操作
      - 如果不是LOAD类型则忽略
  - 四、程序头表读完了？加载完成

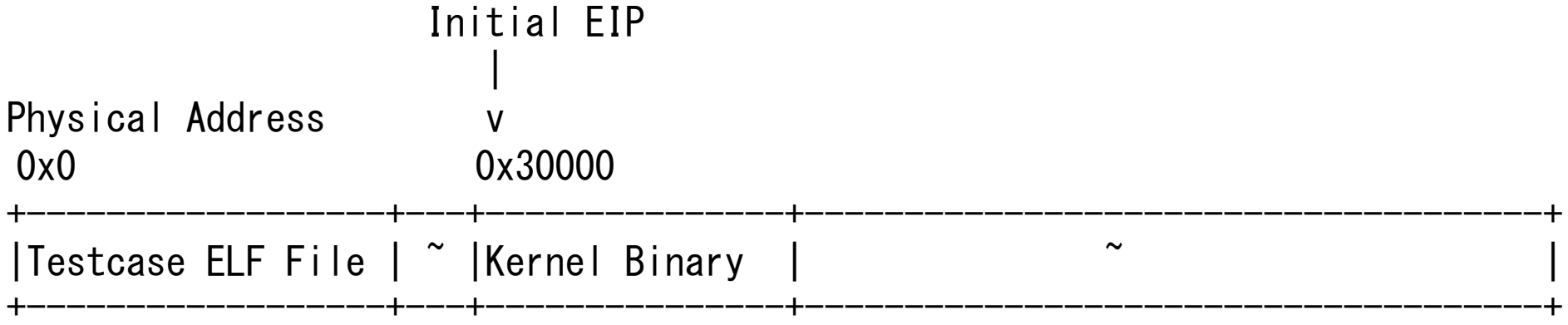
装载ELF可执行文件

- 谁来干这装载的工作
  - NEMU是不管的，它是裸机，只认指令序列，才不管你的可执行目标文件存成什么样
  - 既然可执行目标文件的格式是操作系统说了算的（Linux和Windows不一样），自然要有一个操作系统
  - 于是Kernel登场了！

$ make testkernel

装载ELF可执行文件

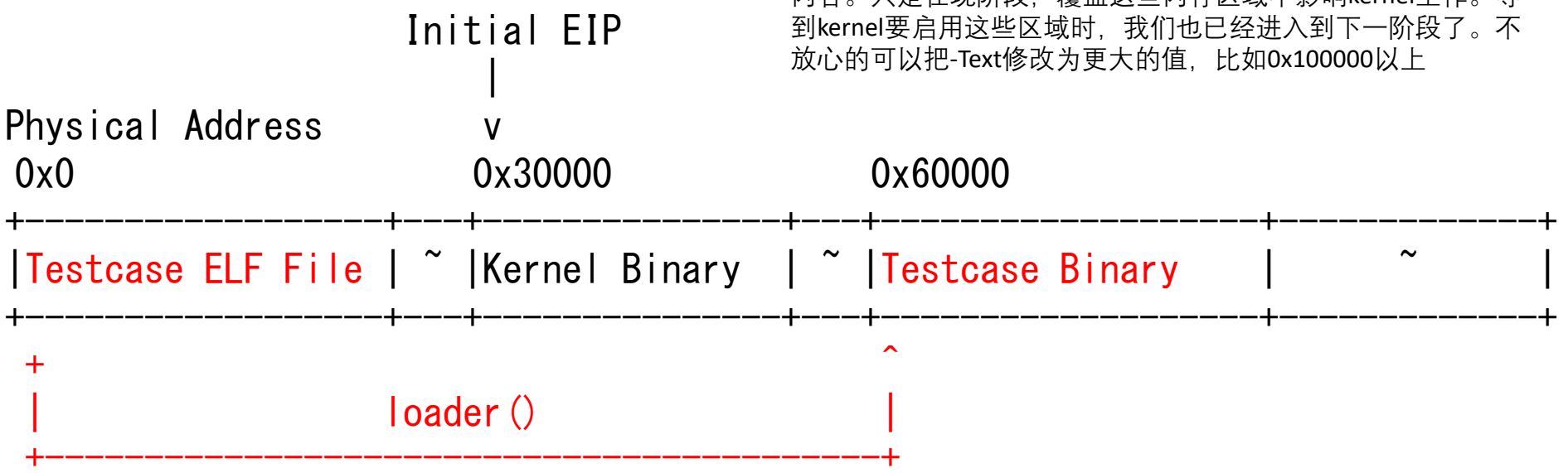
- make testkernel
  - 其行为和make test很相似
  - 区别在于，在物理内存0x30000处装载的不再是testcase binary image，而是kernel binary image
  - 物理内存0x0处开始，Ram Disk内摆放的仍然是testcase ELF file



装载ELF可执行文件

- make testkernel
  - NEMU从0x30000开始执行的第一条指令是Kernel的指令
  - Kernel负责将testcase ELF文件装载到内存里
  - 此时如果testcase在链接时仍然以0x30000作为起始地址，那么必然覆盖Kernel的内容，因此必须为kernel让出必要的空间
    - 在testcase/Makefile中，修改-Text=0x60000

实际上为Kernel留出0x30000字节的空间并不完全足够。通过readelf –a kernel仔细观察，发现其实在0x60000以后kernel还有内容。只是在现阶段，覆盖这些内存区域不影响kernel工作。等到kernel要启用这些区域时，我们也已经进入到下一阶段了。不放心的可以把-Text修改为更大的值，比如0x100000以上



装载ELF可执行文件

- 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:
        # Set up a stack for C code.
                movl $0, %ebp
                movl $(128 << 20), %esp
                jmp init                                # never return
```

装载ELF可执行文件

- Kernel的行为
  - 转到了kernel/src/main.c

```
void init() {
#ifdef IA32_PAGE
        /* We must set up kernel virtual memory first because our kernel thinks it
         * is located at 0xc0030000, which is set by the linking options in Makefile.
         * Before setting up correct paging, no global variable can be used. */
        init_page();

        /* After paging is enabled, transform %esp to virtual address. */
        asm volatile("addl %0, %%esp" : : "i"(KOFFSET));
#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);
}
```

装载ELF可执行文件

- Kernel的行为
  - 转到了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

}
```

装载ELF可执行文件

- Kernel的行为
  - loader()位于kernel/src/elf/elf.c
  - 详见后页

```
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)();
}
```

装载ELF可执行文件

- 如果loader()实现正确，且指令实现都正确
  - 执行make testkernel

```
Execute ./kernel/kernel.img ./testcase/bin/mov
hit breakpoint at eip = 0x00030000
(nemu) c
nemu trap output: [src/main.c,75,init_cond] {kernel} Hello, NEMU world!
nemu trap output: [src/elf/elf.c,29,loader] {kernel} ELF loading from ram disk.
nemu: HIT GOOD TRAP at eip = 0x000602b6
NEMU2 terminated
```

除了test-float，其它都是HIT GOOD TRAP

PA 2-2 程序的装载 & PA 2-3 调试器

- ELF文件的装载
- 符号表的解析

符号表的解析

- 在程序加载的过程中，我们解析了ELF文件的程序头表部分
- 而符号表解析的过程中，我们关注ELF文件的符号表部分
- 首先要找到名为.symtab的符号表和名为.strtab的字符串表
- 从读取节头表开始…

ELF可执行目标文件

找到名为.symtab的符号表和名为.strtab的字符串表

- 百闻不如一见
  - 以testcase/bin/add为例：readelf –a add | less
  - 出来好多内容，一块块地来看

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 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可执行目标文件

找到名为.symtab的符号表和名为.strtab的字符串表

- 百闻不如一见
  - 以testcase/bin/add为例：readelf –a add | less
  - 出来好多内容，一块块地来看

通过ElfN_Ehdr中的e_shoff可以找到节头表的首地址，而所谓的节头表，在代码中就是ELF32_Shdr类型的一个数组，包含e_shnum个数组元素。数组中的每一项：Addr – 该节在内存中的地址，Off – 该节在ELF文件中的位置，Size - 该节在文件中所占用的字节数。

```
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)

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

ELF可执行目标文件

找到名为.symtab的符号表和名为.strtab的字符串表

- 百闻不如一见
  - 以testcase/bin/add为例：readelf –a add | less
  - 出来好多内容，一块块地来看

通过ElfN_Ehdr中的e_shoff可以找到节头表的首地址，而所谓的节头表，在代码中就是ELF32_Shdr类型的一个数组，包含e_shnum个数组元素。数组中的每一项：Addr – 该节在内存中的地址，Off – 该节在ELF文件中的位置，Size - 该节在文件中所占用的字节数。

```
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_frame      PROGBITS        00000000 00249c 000040 00      0   0  4
  [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)

There are no section groups in this file.
```

符号表，及其对应的字符串表！

<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

NEMU中找到符号表的代码

- 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

- 符号表也是个数组，其类型为Elf32_Sym，首地址查节头表来获取

readelf –s add

```
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
```

这里的name是解析过的

<elf.h>

```
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, 0xfffffffe,
0xffffffff};
```

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|
```

'\0'

```
Symbol table '.symtab' contains 25 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
    24: 00032020    32 OBJECT  GLOBAL DEFAULT    4 6d
```

符号表的解析

- 符号表(.symtab)与字符串表(.strtab)结合
  - nemu/src/monitor/elf.c
  - 找到一个符号（使用全局变量名或函数名）在内存中的地址

```
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;
}
```

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

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
   ...
```

opcode.o

```
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
   ...
```

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？

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

实用小贴士

- 引入kernel之后
  - make run不能用了
  - make testkernel不进入交互模式了，调试好困难
  - 可以使用BREAK_POINT宏强制进入交互模式

```
#ifndef IA32_SEG

        .globl start
        start:
        # Set up a stack for C code.
                BREAK_POINT #每次一进Kernel就break，进入交互模式
                movl $0, %ebp
                movl $(128 << 20), %esp
                jmp init                                    # never return
```

任务

- PA 2-2 kernel装载程序
  - 2017年11月1日24时截止
  - 期间建议充分熟悉ELF文件以及相应的符号解析部分
  - 建议大家有能力的尽可能对照教程往前做，不要等，为配合理论课我们前面只能放得很慢
- PA 2-3的截止
  - 因为要等理论课进度，我们会再往后延一些
  - 延了这么久了，大家就努力把这个部分做完吧
- 下次课我们讲
  - 表达式求值
  - 同时为PA 3开个头

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

（量表一、二、三到PA 2截止时做一次，如果PA 2-2截止的时候也能做一次最好）