

实验二（PA2）指令系统

实验简介(请认真阅读以下内容, 若有违反, 后果自负)

预计平均耗时/代码量: 48 小时/约 750 行

实验内容: 本次实验的阶段安排如下:

- 阶段 1: 编写 helper 函数, 在 NEMU 中运行第一个 C 程序——mov-c.c
- 阶段 2: 实现更多的指令, 并通过测试
- 阶段 3: 完善简易调试器
- 阶段 4: 实现 loader
- 最后阶段: 实现黑客运行时劫持实验 (选做)

提交说明: 见[这里](#)

评分依据: 代码实现占 70%, 实验报告占 30%。代码实现中完成阶段 1 占 20%, 阶段 2 占 50%, 阶段 3 占 10%, 阶段 4 占 20%, 选做任务不计入成绩。


进行本实验前, 请在工程目录下执行以下命令进行分支整理, 否则将影响成绩:


```
git commit --allow-empty -am "before starting PA2"
```


```
git checkout master
```


```
git merge pa1
```


```
git checkout -b pa2
```

 普通阅读, 无需提交。

 必做任务, 需要提交

 选做任务, 可以不提交

 思考题, 需要提交

 实验简介, 请仔细阅读。

x86 指令系统简介

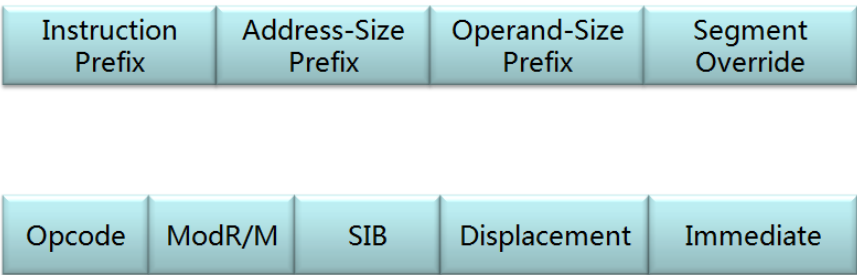
PA2 的任务是在 NEMU 中实现 x86 指令系统(的子集), 你不可避免地需要了解 x86 指令系统的细节。i386 手册有一章专门列出了所有指令的细节, 你需要在完成 PA2 的过程中反复阅读这一章的内容, 附录中的 opcode map 也很有用。在这一小节中, 我们对 x86 指令系统作一些简单的梳理。当你对于 x86 指令系统有任何疑问时, 请查阅 i386 手册, 关于指令系统的一切细节都在里面。

i386 手册勘误

我们在附件 6 列出目前找到的错误, 如果你在做实验的过程中也发现了新的错误, 请帮助我们更新勘误信息。

指令格式

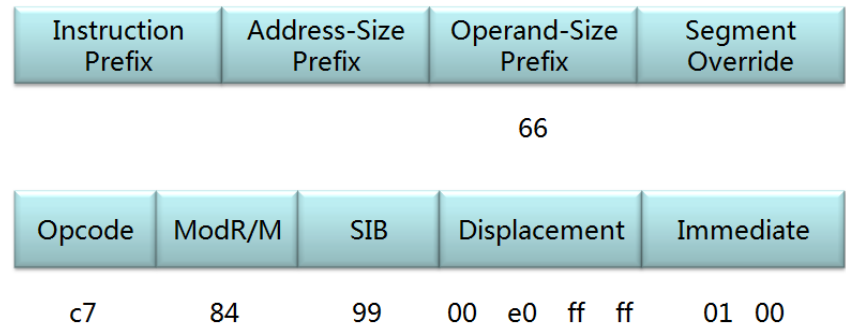
x86 指令的一般格式如下:



除了 opcode(操作码)必定出现之外, 其余组成部分可能不出现, 而对于某些组成部分, 其长度并不是固定的。但给定一条具体指令的二进制形式, 其组成部分的划分是有办法确定的, 不会产生歧义(即把一串比特串看成指令的时候, 不会出现两种不同的解释)。例如对于以下指令:

```
1000fe: 66 c7 84 99 00 e0 ff    movw    $0x1, -0x2000(%ecx,%ebx,4)
100105: ff 01 00
```

其组成部分的划分如下:

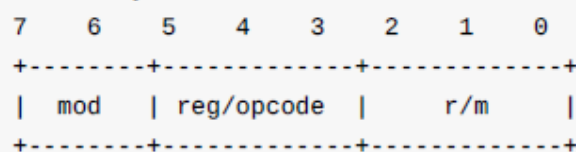


凭什么 0x84 要被解释成 ModR/M 字节呢？这是由 opcode 决定的，opcode 决定了这是什么指令的什么形式，同时也决定了 opcode 之后的比特串如何解释。如果你要问是谁来决定 opcode，那你就得去问 Intel 了。

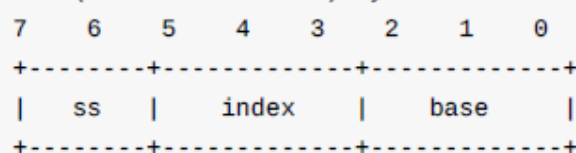
在我们的 PA 中，instruction prefix, address-size prefix 和 segment override prefix 都不会用到，因此 NEMU 也不需要实现这三者的功能。

另外我们在这里先给出 ModR/M 字节和 SIB 字节的格式，它们是用来确定指令的操作数的，详细的功能会在将来进行描述：

ModR/M byte



SIB (scale index base) byte



RISC - 与 CISC 平行的另一个世界

你是否觉得 x86 指令集的格式特别复杂？这其实是 CISC 的一个特性，不惜使用复杂的指令格式，牺牲硬件的开发成本，也要使得一条指令可以多做事情，从而提高代码的密度，减小程序的大小。随着时代的发展，架构师发现 CISC 中复杂的控制逻辑不利于提高处理器的性能，于是 RISC 应运而生。RISC 的宗旨就是简单，指令少，指令长度固定，指令格式统一，这和 KISS 法则有异曲同工之妙。[这里](#)有一篇对比 RICS 和 CISC 的小短文。

另外值得推荐的是[这篇文章](#)，里面讲述了一个从 RICS 世界诞生，到与 CISC 世界融为一体的故事，体会一下 RICS 的诞生对计算机体系结构发展的里程碑意义。

指令集细节

要实现一条指令，首先你需要知道这条指令的格式和功能，格式决定如何解释，功能决定如何执行。而这些信息都在 instruction set page (i386 手册第 17 章) 中，因此你务必知道如何阅读它们。我们以 mov 指令的 opcode 表为例来说明如何阅读：

Opcode	Instruction	Clocks	Description
< 1> 88 /r	MOV r/m8,r8	2/2	Move byte register to r/m byte
< 2> 89 /r	MOV r/m16,r16	2/2	Move word register to r/m word
< 3> 89 /r	MOV r/m32,r32	2/2	Move dword register to r/m dword
< 4> 8A /r	MOV r8,r/m8	2/4	Move r/m byte to byte register
< 5> 8B /r	MOV r16,r/m16	2/4	Move r/m word to word register
< 6> 8B /r	MOV r32,r/m32	2/4	Move r/m dword to dword register
< 7> 8C /r	MOV r/m16,Sreg	2/2	Move segment register to r/m word
< 8> 8D /r	MOV Sreg,r/m16	2/5,pm=18/19	Move r/m word to segment register
< 9> A0	MOV AL,moffs8	4	Move byte at (seg:offset) to AL
<10> A1	MOV AX,moffs16	4	Move word at (seg:offset) to AX
<11> A1	MOV EAX,moffs32	4	Move dword at (seg:offset) to EAX
<12> A2	MOV moffs8,AL	2	Move AL to (seg:offset)
<13> A3	MOV moffs16,AX	2	Move AX to (seg:offset)
<14> A3	MOV moffs32,EAX	2	Move EAX to (seg:offset)
<15> B0 + rb ib	MOV r8,imm8	2	Move immediate byte to register
<16> B8 + rw iw	MOV r16,imm16	2	Move immediate word to register
<17> B8 + rd id	MOV r32,imm32	2	Move immediate dword to register
<18> C6 /0 ib (*)	MOV r/m8,imm8	2/2	Move immediate byte to r/m byte
<19> C7 /0 iw (*)	MOV r/m16,imm16	2/2	Move immediate word to r/m word
<20> C7 /0 id (*)	MOV r/m32,imm32	2/2	Move immediate dword to r/m dword

NOTES:

moffs8, moffs16, and moffs32 all consist of a simple offset relative to the segment base. The 8, 16, and 32 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 bits.

注:

标记了(*)的指令形式的Opcode相对于i386手册有改动, 具体情况见下文的描述。

上表中的每一行给出了 **mov** 指令的不同形式, 每一列分别表示这种形式的 opcode, 汇编语言格式, 执行所需周期, 以及功能描述。由于 NEMU 关注的是功能的模拟, 因此 Clocks 一列不必关心。另外需要注意的是, i386 手册中的汇编语言格式都是 Intel 格式, 而 objdump (反汇编) 的默认格式是 AT&T 格式, 两者的源操作数和目的操作数位置不一样, 千万不要把它们混淆了! 否则你将会陷入难以理解的 bug 中。

首先我们来看 **mov** 指令的第一种形式:

Opcode	Instruction	Clocks	Description
< 1> 88 /r	MOV r/m8,r8	2/2	Move byte register to r/m byte

- 从功能描述可以看出, 它的作用是"将一个 8 位寄存器中的数据传送到 8 位的寄存器或者内存中", 其中 **r/m** 表示"寄存器或内存"。
- Opcode 一列中的编码都是用十六进制表示, **88** 表示这条指令的 opcode

的首字节是 0x88, /r 表示后面跟一个 ModR/M 字节, 并且 ModR/M 字节中的 reg/opcode 域解释成通用寄存器的编码, 用来表示其中一个操作数。通用寄存器的编码如下:

二进制编码	000	001	010	011	100	101	110	111
8位寄存器	AL	CL	DL	BL	AH	CH	DH	BH
16位寄存器	AX	CX	DX	BX	SP	BP	SI	DI
32位寄存器	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

- Instruction 一列中, r/m8 表示操作数是 8 位的寄存器或内存, r8 表示操作数是 8 位寄存器, 按照 Intel 格式的汇编语法来解释, 表示将 8 位寄存器(r8)中的数据传送到 8 位寄存器或内存(r/m8)中, 这和功能描述是一致的。至于 r/m 表示的究竟是寄存器还是内存, 这是由 ModR/M 字节的 mod 域决定的: 当 mod 域取值为 3 的时候, r/m 表示的是寄存器; 否则 r/m 表示的是内存。表示内存的时候又有多种寻址方式, 具体信息参考 i386 手册中的表格 17-3。

看明白了上面的第一种形式之后, 接下来的两种形式也就不难看懂了:

< 2> 89 /r	MOV r/m16,r16	2/2	Move word register to r/m word
< 3> 89 /r	MOV r/m32,r32	2/2	Move dword register to r/m dword

但你会发现, 这两种形式的 Opcode 都是一样的, 难道不会出现歧义吗? 不用着急, 还记得指令一般格式中的 operand-size prefix 吗? x86 正是通过它来区分上面这两种形式的。Operand-size prefix 的编码是 0x66, 作用是指当前指令需要改变操作数的长度。在 IA-32 中, 通常来说, 如果这个前缀没有出现, 操作数长度默认是 32 位; 当这个前缀出现的时候, 操作数长度就要改变成 16 位(也有相反的情况, 这个前缀的出现使得操作数长度从 16 位变成 32 位, 但这种情况在 IA-32 中极少出现)。换句话说, 如果把一个开头为 89 ... 的比特串解释成指令, 它就应该被解释成 MOV r/m32,r32 的形式; 如果比特串的开头是 66 89..., 它就应该被解释成 MOV r/m16,r16。

到现在为止, <4>-<6>三种形式你也明白了:

< 4> 8A /r	MOV r8,r/m8	2/4	Move r/m byte to byte register
< 5> 8B /r	MOV r16,r/m16	2/4	Move r/m word to word register
< 6> 8B /r	MOV r32,r/m32	2/4	Move r/m dword to dword register

<7>和<8>两种形式的 mov 指令涉及到段寄存器:

< 7> 8C /r	MOV r/m16,Sreg	2/2	Move segment register to r/m word
< 8> 8D /r	MOV Sreg,r/m16	2/5,pm=18/19	Move r/m word to segment register

现在 NEMU 中并没有加入段寄存器的功能, 因此这两种形式的 **mov** 指令还没有实现。但现在你可以先忽略它们, 你将会在 PA3 中加入这两种形式。

<9>-<14>这 6 种形式涉及到一种新的操作数记号 **moffs**:

< 9> A0	MOV AL,moffs8	4	Move byte at (seg:offset) to AL
<10> A1	MOV AX,moffs16	4	Move word at (seg:offset) to AX
<11> A1	MOV EAX,moffs32	4	Move dword at (seg:offset) to EAX
<12> A2	MOV moffs8,AL	2	Move AL to (seg:offset)
<13> A3	MOV moffs16,AX	2	Move AX to (seg:offset)
<14> A3	MOV moffs32,EAX	2	Move EAX to (seg:offset)

NOTES:

moffs8, moffs16, and moffs32 all consist of a simple offset relative to the segment base. The 8, 16, and 32 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 bits.

NOTES 中给出了 **moffs** 的含义, 它用来表示段内偏移量, 但 NEMU 现在还没有"段"的概念, 目前可以理解成"相对于物理地址 0 处的偏移量"。这 6 种形式是 **mov** 指令的特殊形式, 它们可以不通过 **ModR/M** 字节, 让 **displacement** 直接跟在 opcode 后面, 同时让 **displacement** 来指示一个内存地址。

<15>-<17>三种形式涉及到两种新的操作数记号:

<15> B0 + rb ib	MOV r8,imm8	2	Move immediate byte to register
<16> B8 + rw iw	MOV r16,imm16	2	Move immediate word to register
<17> B8 + rd id	MOV r32,imm32	2	Move immediate dword to register

其中:

- **+rb, +rw, +rd** 分别表示 8 位, 16 位, 32 位通用寄存器的编码。和 **ModR/M** 中的 **reg** 域不一样的是, 这三种记号表示直接将通用寄存器的编号按数值加到 **opcode** 中(也可以看成通用寄存器的编码嵌在 **opcode** 的低三位), 因此识别指令的时候可以通过 **opcode** 的低三位确定一个寄存器操作数。
- **ib, iw, id** 分别表示 8 位, 16 位, 32 位立即数。

最后 3 种形式涉及到一种新的操作码记号 **/digit**, 其中 **digit** 为 0 ~ 7 中的一个数字:

<18> C6 /0 ib (*) MOV r/m8,imm8	2/2	Move immediate byte to r/m byte
<19> C7 /0 iw (*) MOV r/m16,imm16	2/2	Move immediate word to r/m word
<20> C7 /0 id (*) MOV r/m32,imm32	2/2	Move immediate dword to r/m dword

注:

标记了(*)的指令形式的Opcode相对于i386手册有改动, 具体情况见下文的描述。

上述形式中的 /0 表示一个 ModR/M 字节, 并且 ModR/M 字节中的 reg/opcode 域解释成扩展 opcode, 其值取 0。对于含有 /digit 记号的指令形式, 需要通过指令本身的 opcode 和 ModR/M 中的扩展 opcode 共同决定指令的形式, 例如 80 /0 表示 add 指令的一种形式, 而 80 /5 则表示 sub 指令的一种形式, 只看 opcode 的首字节 80 不能区分它们。

注: 在 i386 手册中, 这 3 种形式的 mov 指令并没有 /0 的记号, 在这里加入 /0 纯粹是为了说明 /digit 记号的意思。但同时这条指令在 i386 中也比较特殊, 它需要使用 ModR/M 字节来表示一个寄存器或内存的操作数, 但 ModR/M 字节中的 reg/opcode 域却没有用到(一般情况下, ModR/M 字节中的 reg/opcode 域要么表示一个寄存器操作数, 要么作为扩展 opcode), i386 手册也没有对此进行特别的说明, 直觉上的解释就是"无论 ModR/M 字节中的 reg/opcode 域是什么值, 都可以被 CPU 识别成这种形式的 mov 指令"。x86 是商业 CPU, 我们无法从电路级实现来考证这一解释, 但对编译器生成代码来说, 这条指令中的 reg/opcode 域总得有个确定的值, 因此编译器一般会把这个值设成 0。在 NEMU 的框架代码中, 对这 3 种形式的 mov 指令的实现和 i386 手册中给出 opcode 保持一致, 忽略 ModR/M 字节中的 reg/opcode 域, 没有判断其值是否为 0。如果你不能理解这段话在说什么, 你可以忽略它, 因为这并不会影响实验的进行。

到此为止, 你已经学会了如何阅读大部分的指令集细节了。需要说明的是, 这里举的 mov 指令的例子并没有完全覆盖 i386 手册中指令集细节的所有记号, 若有疑问, 请参考 i386 手册。

除了 opcode 表之外, Operation, Description 和 Flags Affected 这三个条目都要仔细阅读, 这样你才能完整地掌握一条指令的功能。Exceptions 条目涉及到执行这条指令可能产生的异常, 由于 NEMU 不打算实现异常处理的机制, 你可以不用关心这一条目。

阅读源代码

上一小节中的内容全部出自 i386 手册, 现在我们结合框架代码来理解上面的内容。

在 PA1 中, 你已经阅读了 monitor 部分的框架代码, 了解了 NEMU 执行的粗略框架. 但现在你需要进一步弄明白, 一条指令是怎么在 NEMU 中执行的, 即我们需要进一步探究 `exec()` 函数中的细节。为了说明这个过程, 我们举了两个 `mov` 指令的例子, 它们是框架代码自带的用户程序 `mov(testcase/src/mov.S)` 中的两条指令(`mov` 的反汇编结果在 `obj/testcase/mov.txt` 中):

```
100014:    b9 00 80 00 00          mov     $0x8000,%ecx
.....
1000fe:    66 c7 84 99 00 e0 ff    movw   $0x1, -0x2000(%ecx,%ebx,4)
100105:    ff 01 00
```

helper 函数命名约定

对于每条指令的每一种形式, NEMU 分别使用一个 helper 函数来模拟它的执行。为了易于维护, 框架代码对 helper 函数的命名有一种通用的形式:

指令_形式_操作数后缀

例如对于 helper 函数 `mov_i2rm_b()`, 它模拟的指令是 `mov`, 形式是把立即数移动到寄存器或内存, 操作数后缀是 `b`, 表示操作数长度是 8 位。在 PA2 中, 你需要实现很多 helper 函数, 这种命名方式可以很容易地让你知道一个 helper 函数的功能。

一个特殊的操作数后缀是 `v`, 表示 variant, 意味着光看操作码的首字节, 操作数长度还不能确定, 可能是 16 位或者 32 位, 需要通过 `ops_decoded.is_operand_size_16` 成员变量来决定。其实这种 helper 函数做的事情, 就是在根据指令是否出现 `operand-size prefix` 来确定操作数长度, 从而决定最终的指令形式, 调用最终的 helper 函数来模拟指令的执行。

也有一些指令不需要区分形式和操作数后缀, 例如 `int3`, 这时可以直接用指令的名称来命名其 helper 函数。如果你觉得上述命名方式不易看懂, 你可以使用其它命名方式, 我们不做强制要求。

简单 mov 指令的执行

我们先来剖析第一条 `mov $0x8000, %ecx` 指令的执行过程。当 NEMU 执行到这条指令的时候(`eip = 0x100014`), 当前 `%eip` 的值被作为参数送进 `exec()` 函数(在 `nemu/src/cpu/exec/exec.c` 中定义)中。其中 `make_helper` 是个宏, 你需要编写一系列 helper 函数来模拟指令执行的过程, 而 `make_helper` 则定义了

helper 函数的声明形式:

```
#define make_helper(name) int name(swaddr_t eip)
```

从 `make_helper` 的定义可以看到, helper 函数都带有一个参数 `eip`, 返回值类型都是 `int`。从抽象的角度来说, 一个 helper 函数做的事情就是对参数 `eip` 所指向的内存单元进行某种操作, 然后返回这种操作涉及的代码长度。例如 `exec()` 函数的功能是"执行参数 `eip` 所指向的指令, 并返回这条指令的长度"; 框架代码中还定义了一些获取指令中的立即数的 `helper` 函数, 它们的功能是"获取参数 `eip` 所指向的立即数, 并返回这个立即数的长度"。

对于大部分指令来说, 执行它们都可以抽象成取指-译码-执行的指令周期。为了使描述更加清晰, 我们借助指令周期中的一些概念来说明指令执行的过程。

取指(instruction fetch, IF)

要执行一条指令, 首先要拿到这条指令。指令究竟在哪里呢? 还记得冯诺依曼体系结构的核心思想吗? 那就是"存储程序, 程序控制"。你以前听说这两句话的时候可能没有什么概念, 现在是实践的时候了。这两句话告诉你, 指令在存储器中, 由 PC(program counter, 在 x86 中就是 `%eip`) 指出当前指令的位置。事实上, `%eip` 就是一个指针! 在计算机世界中, 指针的概念无处不在, 如果你觉得对指针的概念还不是很熟悉, 就要赶紧复习指针这门必修课。取指令要做的事情自然就是将 `%eip` 指向的指令从内存读入到 CPU 中。在 NEMU 中, 有一个函数 `instr_fetch()` (在 `nemu/include/cpu/helper.h` 中定义) 专门负责取指令的工作。

译码(instruction decode, ID)

在取指阶段, CPU 拿到的是指令的比特串。如果想知道这串比特串究竟代表什么意思, 就要进行译码的工作了。我们可以把译码的工作作进一步的细化: 首先要决定具体是哪一条指令的哪一种形式, 这主要是通过查看指令的 `opcode` 来决定的。对于大多数指令来说, CPU 只要看指令的第一个字节就可以知道具体指令的形式了。在 NEMU 中, `exec()` 函数首先通过 `instr_fetch()` 取出指令的第一个字节, 然后根据取到的这个字节查看 `opcode_table`, 得到指令的 helper 函数, 从而调用这个 helper 函数来继续模拟这条指令的执行。以 `mov $0x8000, %ecx` 指令为例, 首先通过 `instr_fetch()` 取得这条指令的第一个字节 `0xb9`, 然后根据这个字节来索引 `opcode_table`, 找到了一个名为 `mov_i2r_v` 的 helper 函数, 这样就可以确定取到的是一条 `mov` 指令, 它的形式是将立即数移入寄存器(move immediate to register)。

事实上, 一个字节最多只能区分 256 种不同的指令形式, 当指令形式的数目大于 256 时, 我们需要使用另外的方法来识别它们。x86 中有主要有两种方法来

解决这个问题(在 PA2 中你都会遇到这两种情况):

- 一种方法是使用转义码(escape code), x86 中有一个 2 字节转义码 `0x0f`, 当指令 `opcode` 的第一个字节是 `0x0f` 时, 表示需要再读入一个字节才能决定具体的指令形式(部分条件跳转指令就属于这种情况)。后来随着各种 SSE 指令集的加入, 使用 2 字节转义码也不足以表示所有的指令形式了, x86 在 2 字节转义码的基础上又引入了 3 字节转义码, 当指令 `opcode` 的前两个字节是 `0x0f` 和 `0x38` 时, 表示需要再读入一个字节才能决定具体的指令形式。
- 另一种方法是利用 `ModR/M` 字节对 `opcode` 的长度进行扩充。有些时候, 读入一个字节也还不能完全确定具体的指令形式, 这时候需要读入紧跟在 `opcode` 后面的 `ModR/M` 字节, 把其中的 `reg/opcode` 域当做 `opcode` 的一部分来解释, 才能决定具体的指令形式。x86 把这些指令划分成不同的指令组(instruction group), 在同一个指令组中的指令需要通过 `ModR/M` 字节中的扩展 opcode 域来区分。

决定了具体的指令形式之后, 译码工作还需要决定指令的操作数。事实上, 在确定了指令的 `opcode` 之后, 指令形式就能确定下来了, CPU 可以根据指令形式来确定具体的操作数。我们还是以 `mov $0x8000, %ecx` 来说明这个过程, 但在这之前, 我们需要作一些额外的说明。在上文的描述中, 我们通过这条指令的第一个字节 `0xb9` 找到了 `mov_i2r_v()` 的 helper 函数, 这个 helper 函数的定义在 `nemu/src/cpu/exec/data-mov/mov.c` 中:

`make_helper_v(mov_i2r)`

其中 `make_helper_v()` 是个宏, 它在 `nemu/include/cpu/exec/helper.h` 中定义:

```
#define make_helper_v(name) \
    make_helper(concat(name, _v)) { \
        return (ops_decoded.is_operand_size_16 ? concat(name, _w) : concat(name, _l)) (eip); \
    }
```

进行宏展开之后, `mov_i2r_v()` 的函数体如下:

```
int mov_i2r_v(swaddr_t eip) {
    return (ops_decoded.is_operand_size_16 ? mov_i2r_w : mov_i2r_l) (eip); \
}
```

它的作用是 根据全局变量 `ops_decoded`(在 `nemu/src/cpu/decode/decode.c` 中定义)中的 `is_operand_size_16` 成员变量来决定操作数的长度, 然后从两个 helper 函数中选择一个进行调用。全局变量

`ops_decoded` 用于存放一些译码的结果, 其中的 `is_operand_size_16` 成员和指令中的 `operand-size prefix` 有关, 而且会经常用到, 框架代码把类似于 `mov_i2r_v()` 这样的功能抽象成一个宏 `make_helper_v()`, 方便代码的编写。关于 `is_operand_size_16` 成员的更多内容会在下文进行说明。根据指令 `mov $0x8000, %ecx` 的功能, 它的操作数长度为 4 字节, 因此这里会调用 `mov_i2r_l()` 的 `helper` 函数。`mov_i2r_l()` 的 `helper` 函数在 `nemu/src/cpu/exec/data-mov/mov-template.h` 中定义, 它的函数体是通过宏展开得到的, 在这里我们直接给出宏展开的结果, 关于宏的使用请阅读相应的框架代码:

```
int mov_i2r_l(swaddr_t eip) {
    return idex(eip, decode_i2r_l, do_mov_l);
}
```

其中 `idex()` 函数的原型为:

```
int idex(swaddr_t eip, int (*decode)(swaddr_t), void (*execute) (void));
```

它的作用是通过 `decode` 函数对参数 `eip` 指向的指令进行译码, 然后通过 `execute` 函数执行这条指令。

对于 `mov $0x8000, %ecx` 指令来说, 确定操作数其实就是确定寄存器 `%ecx` 和立即数 `$0x8000`。在 x86 中, 通用寄存器都有自己的编号, `mov_i2r` 形式的指令把寄存器编号也放在指令的第一个字节里面, 我们可以通过位运算将寄存器编号抽取出来。对于 `mov_i2r` 形式的指令来说, 立即数存放在指令的第二个字节, 可以很容易得到它。然而很多指令都具有 `i2r` 的形式, 框架代码提供了几个函数(`decode_i2r_l()` 等), 专门用于进行对 `i2r` 形式的指令的译码工作。`decode_i2r_l()` 函数会把指令中的立即数信息和寄存器信息分别记录在全局变量 `ops_decoded` 中的 `src` 成员和 `dest` 成员中, `nemu/include/cpu/helper.h` 中定义了两个宏 `op_src` 和 `op_dest`, 用于方便地访问这两个成员。

执行(execute, EX)

译码阶段的工作完成之后, CPU 就知道当前指令具体要做什么了, 执行阶段就是真正完成指令的工作。对于 `mov $0x8000, %ecx` 指令来说, 执行阶段的工作就是把立即数 `$0x8000` 送到寄存器 `%ecx` 中。由于 `mov` 指令的功能可以统一成“把源操作数的值传送到目标操作数中”, 而译码阶段已把操作数都准备好了, 所以只需要针对 `mov` 指令编写一个模拟执行过程的函数即可。这个函数就是 `do_mov_l()`, 它是通过在 `nemu/src/cpu/exec/data-mov/mov-template.h` 中

定义的 `do_execute()` 函数进行宏展开后得到的：

```
static void do_mov_l() {
    write_operand_l(&ops_decoded.dest), (&ops_decoded.src)->val);
    Assert(snprintf(assembly, 80, "movl %s,%s", (&ops_decoded.src)->str, (&ops_decoded.dest
}
```

其中 `write_operand_l()` 函数会根据第一个参数中记录的类型的不同进行相应的写操作，包括写寄存器和写内存。

更新 %eip

执行完一条指令之后，CPU 就要执行下一条指令。在这之前，CPU 需要更新 `%eip` 的值，让 `%eip` 指向下一条指令的位置。为此，我们需要确定刚刚执行完的指令的长度。在 NEMU 中，指令的长度是通过 `helper` 函数的返回值进行传递的，最终会传回到 `cpu_exec()` 函数中，完成对 `%eip` 的更新。

复杂 mov 指令的执行

对于第二个例子 `movw $0x1, -0x2000(%ecx,%ebx,4)`，执行这条指令还是分取指、译码、执行三个阶段。

首先是取指。这条 `mov` 指令比较特殊，它的第一个字节是 `0x66`，如果你查阅 i386 手册，你会发现 `0x66` 是一个 `operand-size prefix`。因为这个前缀的存在，本例中的 `mov` 指令才能被 CPU 识别成 `movw`。NEMU 使用 `ops_decoded.is_operand_size_16` 成员变量来记录操作数长度前缀是否出现，`0x66` 的 `helper` 函数 `operand_size()` 实现了这个功能。

`operand_size()` 函数对 `ops_decoded.is_operand_size_16` 成员变量做了标识之后，越过前缀重新调用 `exec()` 函数，此时取得了真正的操作码 `0xc7`，通过查看 `opcode_table` 调用了 `helper` 函数 `mov_i2rm_v()`。由于 `ops_decoded.is_operand_size_16` 成员变量进行过标识，在 `mov_i2rm_v()` 中将会调用 `mov_i2rm_w()` 的 `helper` 函数。到此为止才识别出本例中的指令是一条 `movw` 指令。

接下来是识别操作数。同样，我们先给出 `mov_i2rm_w()` 函数的宏展开结果：

```
int mov_i2rm_w(swaddr_t eip) {
    return idex(eip, decode_i2rm_w, do_mov_w);
}
```

这里使用 `decode_i2rm_w()` 函数来进行译码的工作，阅读代码，你会发现它最终会调用 `read_ModR_M()` 函数。由于本例中的 `mov` 指令需要访问内存，因

此除了要识别出立即数之外,还需要确定好要访问的内存地址。x86 通过 **ModR/M** 字节来指示内存操作数, 支持各种灵活的寻址方式。其中最一般的寻址格式是

displacement(R[base_reg], R[index_reg], scale_factor)

相应内存地址的计算方式为

addr = R[base_reg] + R[index_reg] * scale_factor + displacement

其它寻址格式都可以看作这种一般格式的特例, 例如

displacement(R[base_reg])

可以认为是在一般格式中取 **R[index_reg] = 0, scale_factor = 1** 的情况。这样, 确定内存地址就是要确定 **base_reg, index_reg, scale_factor** 和 **displacement** 这 4 个值, 而它们的信息已全部编码在 **ModR/M** 字节里面。

我们以本例中的 **movw \$0x1, -0x2000(%ecx,%ebx,4)** 说明如何识别出内存地址:

```
1000fe: 66 c7 84 99 00 e0 ff    movw    $0x1, -0x2000(%ecx,%ebx,4)
100105: ff 01 00
```

根据 **mov_i2rm** 的指令形式, **0xc7** 是 **opcode**, **0x84** 是 **ModR/M** 字节。在 i386 手册中查阅表格 17-3 得知, **0x84** 的编码表示在 **ModR/M** 字节后面还跟着一个 **SIB** 字节, 然后跟着一个 32 位的 **displacement**。于是读出 **SIB** 字节, 发现是 **0x99**。在 i386 手册中查阅表格 17-4 得知, **0x99** 的编码表示 **base_reg = ECX, index_reg = EBX, scale_factor = 4**。在 **SIB** 字节后面读出一个 32 位的 **displacement**, 发现是 **00 e0 ff ff**, 在小端存储方式下, 它被解释成 **-0x2000**。于是内存地址的计算方式为

addr = R[ECX] + R[EBX] * 4 - 0x2000

框架代码已经实现了 **load_addr()** 函数和 **read_ModR_M()** 函数(在 **nemu/src/cpu/decode/modrm.c** 中定义), 它们的函数原型为

```
int load_addr(swaddr_t eip, ModR_M *m, Operand *rm);
int read_ModR_M(swaddr_t eip, Operand *rm, Operand *reg);
```

它们将变量 **eip** 所指向的内存位置解释成 **ModR/M** 字节, 根据上述方法对 **ModR/M** 字节和 **SIB** 字节进行译码, 把译码结果存放到参数 **rm** 和 **reg** 指向的变量中, 同时返回这一译码过程所需的字节数。在上面的例子中, 为了计算出内

存地址, 用到了 `ModR/M` 字节, `SIB` 字节和 32 位的 `displacement`, 总共 6 个字节, 所以 `read_ModR_M()` 返回 6. 虽然 i386 手册中的表格 17-3 和表格 17-4 内容比较多, 仔细看会发现, `ModR/M` 字节和 `SIB` 字节的编码都是有规律可循的, 所以 `load_addr()` 函数可以很简单地识别出计算内存地址所需要的 4 个要素(当然也处理了一些特殊情况)。不过你现在可以不必关心其中的细节, 框架代码已经为你封装好这些细节, 并且提供了各种用于译码的接口函数。

本例中的执行阶段就是要把立即数写入到相应的内存位置, 这是通过 `do_mov_w()` 函数实现的。执行结束后返回指令的长度, 最终在 `cpu_exec()` 函数中更新 `%eip`。

结构化程序设计

细心的你会发现以下规律:

- 对于同一条指令的不同形式, 它们的执行阶段是相同的。例如 `add_i2rm` 和 `add_rm2r` 等, 它们的执行阶段都是把两个操作数相加, 把结果存入目的操作数。
- 对于不同指令的同一种形式, 它们的译码阶段是相同的。例如 `add_i2rm` 和 `sub_i2rm` 等, 它们的译码阶段都是识别出一个立即数和一个 `rm` 操作数。
- 对于同一条指令同一种形式的不同长度, 它们的译码阶段和执行阶段都是非常类似的。例如 `add_i2rm_b`, `add_i2rm_w` 和 `add_i2rm_l`, 它们都是识别出一个立即数和一个 `rm` 操作数, 然后把相加的结果存入 `rm` 操作数。

这意味着, 如果独立实现每条指令不同形式不同长度的 `helper` 函数, 将会引入大量重复的代码, 需要修改的时候, 相关的所有 `helper` 函数都要分别修改, 遗漏了某一处就会造成 bug, 工程维护的难度急速上升。一种好的做法是把译码, 执行和操作数长度的相关代码分离开来, 实现解耦, 也就是在程序设计课上提到的结构化程序设计。

在框架代码中, 实现译码和执行之间的解耦的是 `idex()` 函数, 它把译码和执行的 `helper` 函数的指针作为参数, 依次调用它们, 这样我们就可以分别编写译码和执行的 `helper` 函数了。实现操作数长度和译码, 执行这两者之间的解耦的是宏 `DATA_BYTE`, 它把不同操作数长度的共性抽象出来, 编写一份模板, 分别进行 3 次实例化, 就可以得到 3 分不同操作数长度的代码。

为了实现进一步的封装和抽象, 框架代码中使用了大量的宏, 我们在这里把相关的宏整理出来, 供大家参考。

宏	含义
<code>nemu/include/macro.h</code>	
<code>str(x)</code>	字符串 "x"
<code>concat(x, y)</code>	token xy
<code>nemu/include/cpu/reg.h</code>	
<code>reg_l(index)</code>	编码为 index 的32位GPR
<code>reg_w(index)</code>	编码为 index 的16位GPR
<code>reg_b(index)</code>	编码为 index 的8位GPR
<code>nemu/include/cpu/exec/template-start.h</code>	
<code>SUFFIX</code>	表示 DATA_BYTE 相应长度的后缀字母, 为 b, w, l 其中之一
<code>DATA_TYPE</code>	表示 DATA_BYTE 相应长度的无符号数据类型, 为 uint8_t, uint16_t, uint32_t 其中之一
<code>DATA_TYPE_S</code>	表示 DATA_BYTE 相应长度的有符号数据类型, 为 int8_t, int16_t, int32_t 其中之一
<code>REG(index)</code>	编码为 index, 长度为 DATA_BYTE 的GPR
<code>REG_NAME(index)</code>	编码为 index, 长度为 DATA_BYTE 的GPR的名称
<code>MEM_R(addr)</code>	从内存位置 addr 读出 DATA_BYTE 字节的数据
<code>MEM_W(addr)</code>	把长度为 DATA_BYTE 字节的数据 data 写入内存位置 addr
<code>OPERAND_W(op, src)</code>	把结果 src 写入长度为 DATA_BYTE 字节的目的地操作数 op
<code>MSB(n)</code>	取出长度为 DATA_BYTE 字节的数据 n 的MSB位
<code>nemu/include/cpu/helper.h</code>	
<code>make_helper(name)</code>	名为 name 的helper函数的原型说明
<code>op_src</code>	全局变量 ops_decoded 中源操作数成员的地址
<code>op_src2</code>	全局变量 ops_decoded 中2号源操作数成员的地址
<code>op_dest</code>	全局变量 ops_decoded 中目的操作数成员的地址
<code>nemu/include/cpu/exec/helper.h</code>	
<code>make_helper_v(name)</code>	名为 name_v 的helper函数的定义, 用于根据指令的操作数长度前缀进一步确定调用哪一个helper函数
<code>do_execute</code>	用于模拟指令真正的执行操作的函数名

<code>make_instr_helper(type)</code>	名为 指令_形式_操作数后缀 的helper函数的定义, 其中 <code>type</code> 为指令的形式, 通过调用 <code>idex()</code> 函数来进行指令的译码和执行
<code>print_asm(...)</code>	将反汇编结果的字符串打印到缓冲区 <code>assembly</code> 中
<code>print_asm_template1()</code>	打印单目操作数指令的反汇编结果
<code>print_asm_template2()</code>	打印双目操作数指令的反汇编结果
<code>print_asm_template3()</code>	打印三目操作数指令的反汇编结果
<code>nemu/src/cpu/exec/*/*-template.h</code>	
<code>instr</code>	指令的名称, 被 <code>do_execute</code> , <code>make_instr_helper(type)</code> 和 <code>print_asm_template?()</code> 使用

计算机世界处处都是 tradeoff, 有好处自然需要付出代价。由于处理宏的时候不会进行语法检查, 因为宏而造成的错误很可能不会马上暴露。例如以下代码:

```
#define N 10;
int a[N];
```

在编译的时候, 编译器会提示代码的第 2 行有语法错误, 但如果你光看第 2 行代码, 你很难发现错误, 甚至会怀疑编译器有 bug。因此如果你对宏不太熟悉, 可能会对阅读框架代码带来困难。我们准备了命令, 用来专门生成 `nemu/src/cpu/decode` 目录和 `nemu/src/cpu/exec` 目录下源文件的预处理结果, 键入

```
make cpp
```

会在这些目录中生成 `.i` 的预处理结果, 它们可以帮助你阅读框架代码, 调试与宏相关的错误。键入

```
make clean-cpp
```

可以移除这些预处理结果。

源文件组织

最后我们来聊聊 `nemu/src/cpu/exec` 目录下源文件的组织方式。

```

nemu/src/cpu/exec
├── all-instr.h
├── arith
│   └── ...
├── data-mov
│   ├── mov.c
│   ├── mov.h
│   ├── mov-template.h
│   ├── xchg.c
│   ├── xchg.h
│   └── xchg-template.h
├── exec.c
├── logic
│   └── ...
├── misc
│   ├── misc.c
│   └── misc.h
├── prefix
│   ├── prefix.c
│   └── prefix.h
├── special
│   ├── special.c
│   └── special.h
└── string
    ├── rep.c
    └── rep.h

```

- `exec.c` 中定义了操作码表 `opcode_table` 和 helper 函数 `exec()`, `exec()` 根据指令的 `opcode` 首字节查阅 `opcode_table`, 并调用相应的 helper 函数来模拟相应指令的执行。除此之外, 和 2 字节转义码相关的 2 字节操作码表 `_2byte_opcode_table`, 以及各种指令组表也在 `exec.c` 中定义。
- `all-instr.h` 中列出了所有用于模拟指令执行的 helper 函数的声明, 这个头文件被 `exec.c` 包含, 这样就可以在 `exec.c` 中的 `opcode_table` 直接使用各种 helper 函数了。
- 除了 `exec.c` 和 `all-instr.h` 两个源文件之外, 目录下还有若干子目录, 这些子目录分别存放用于模拟不同功能的指令的源文件。i386 手册根据功能对所有指令都进行了分类, 框架代码中对相关文件的管理参考了手册中的分类方法 (其中 `special` 子目录下模拟了和 NEMU 相关的功能, 与 i386 手册无关)。以 `nemu/src/cpu/exec/data-mov` 目录下与 `mov` 指令相关的文件为例, 我们对其文件组织进行进一步的说明:
 - `mov.h` 中列出了用于模拟 `mov` 指令所有形式的 helper 函数的声明, 这个头文件被 `all-instr.h` 包含。
 - `mov-template.h` 是 `mov` 指令 helper 函数定义的模板, `mov` 指令 helper 函数的函数体都在这个文件中定义。模板的功能通过宏来实现的: 对于一条指令, 不同操作数长度的相近形式都有相似行为, 可以将它们

公共行为用宏抽象。 `mov-template.h` 开头包含头文件 `nemu/include/cpu/exec/template-start.h`, 结尾包含了头文件 `nemu/include/cpu/exec/template-end.h`, 它们包含了一些在模板头文件中使用的宏定义, 例如 `DATA_TYPE`, `REG()`等, 使用它们可以编写出简洁的代码。

- `mov.c` 中定义了 `mov` 指令的所有 helper 函数, 其中分三次对 `mov-template.h` 中定义的模板进行实例化, 进行宏展开之后就可以得到 helper 函数的完整定义了; 另外操作数后缀为 `v` 的 helper 函数也在 `mov.c` 中定义。

在 PA2 中, 你需要编写很多 helper 函数, 好的源文件组织方式可以帮助你方便地管理工程。

实现新指令

我们对实现一条新指令的流程进行整理。如果指令的形式比较规整, 适合使用模板(适合大部分的指令), 可以按照以下流程来实现

- 编写指令模板文件 `xxx-template.h`
 - 在文件头尾分别包含 `cpu/exec/template-start.h` 和 `cpu/exec/template-end.h`
 - 定义宏 `instr` 为指令名称
 - 定义函数 `static void do_execute()`, 实现该指令的通用执行过程
 - 定义 helper 函数
 - ✓ 若指令的译码方式在 `nemu/include/cpu/decode/decode.h` 中已经存在, 那么可以考虑使用宏 `make_instr_helper()`来构造 helper 函数 (大部分 helper 函数都可以通过这种方式构造)
 - ✓ 否则
 - ✧ 可以考虑添加相应的译码函数
 - ✧ 或者不使用 `make_instr_helper()`, 而是直接使用 `make_helper()`来定义 helper 函数, 在函数体中直接进行译码, 并调用 `do_execute()`(可以参考 `nemu/src/cpu/exec/data-mov/xchg-template.h` 中的 `xchg_a2r` 指令类型)
- 编写指令实例化文件 `xxx.c`
 - 包含 `cpu/exec-helper.h`

- 通过分别将宏 `DATA_BYTE` 定义成 `1`, `2`, `4`, 分别对指令模板文件 `xxxtemplate.h` 进行实例化
- 若一个 `helper` 函数只会在某些操作数长度中用到, 可以在 `xxx-template.h` 中通过条件编译的功能来指定 (可以参考 `nemu/src/cpu/exec/data-mov/xchg-template.h` 中的 `xchg_a2r` 指令类型)
- 必要时通过宏 `make_helper_v()` 定义相应的重载函数, 根据指令的操作数长度前缀确定调用哪一个 `helper` 函数
- 编写指令头文件 `xxx.h`
 - 声明 `helper` 函数的原型
- 在 `nemu/src/cpu/exec/all-instr.h` 中包含 `xxx.h`
- 在 `nemu/src/cpu/exec/exec.c` 中的 `opcode_table` 中填写相应的 `helper` 函数

如果指令形式不易抽象成模板(例如 `ret`), 那么可以不采取模板的方式实现, 直接在 `xxx.c` 中通过宏 `make_helper()` 定义函数体, 并编写译码和执行的过程。

初出茅庐：运行第一个 C 程序

说了这么多，现在到了动手实践的时候，你在 PA2 的第一个任务，就是编写几条指令的 helper 函数，使得第一个简单的 C 程序可以在 NEMU 中运行起来。这个简单的 C 程序的代码是 `testcase/src/mov-c.c`，它做的事情十分简单，对数组的某些元素进行赋值，然后马上读出这些元素的值，检查它们是否被正确赋值。

使用 assertion 进行验证

要怎么证明 `mov-c` 程序正确运行了呢？你可能马上想到把元素的值输出到屏幕上看看。但是，输出一句话是一件很复杂的事情(没错！的确是一件很复杂的事情，尽管你天天都在用)，由于现在 NEMU 的功能十分简陋，不足以支持用户程序进行输出。事实上，做 PA 的最终目标之一，就是让用户程序成功输出一句话，回过头来你就能够理解，程序要输出一句话其实也不容易。

既然用户程序不能输出数组元素，那就用简易调试器中的扫描内存功能，把数组元素所在的内存区域打印出来看看吧！这是一个可行的方法，但你很快就会因为把时间花费在人工检查当而感到厌倦了。

有没有一种方法能够让程序自动进行检查呢？当然有！那不就是帮你拦截了无数 bug 的 assertion 吗？assertion 的功能就是当检查条件为假时，马上终止程序的执行，并汇报违反 assertion 的地方。先别着急，终止程序是需要操作系统的帮助的，目前 NEMU 中并没有运行操作系统，是不能直接使用标准库中的 assertion 功能的。幸运的是，框架代码早就已经考虑到这点了，还记得在 PA1 中提到的 `nemu_trap` 这条特殊的指令吗？我们只需要对这条特殊的指令稍作包装，就可以把 assertion 的功能移植到用户程序中了！

移植后的 assertion 通过 `nemu_assert()` 来使用，它是个宏，在 `lib-common/trap.h` 中定义。`lib-common/trap.h` 专门定义了一些用于测试的宏：

```
#define HIT_GOOD_TRAP \
    asm volatile(".byte 0xd6" : : "a" (0))

#define HIT_BAD_TRAP \
    asm volatile(".byte 0xd6" : : "a" (1))

#define nemu_assert(cond) \
    do { \
        if( !(cond) ) HIT_BAD_TRAP; \
    } while(0)
```

其中 `HIT_GOOD_TRAP` 是一条内联汇编语句，内联汇编语句允许我们在 C 代码中嵌入汇编语句。这条指令和我们常见的汇编指令不一样(例如 `movl`

\$1, %eax), 它是直接通过指令的编码给出的, 它只有一个字节, 就是 0xd6。如果你在 nemu/src/cpu/exec/exec.c 中查看 opcode_table, 你会发现, 这条指令正是那条特殊的 nemu_trap! 这其实也说明了为什么要通过编码来给出这条指令, 如果你使用

```
asm volatile("nemu_trap" : : "a" (0))
```

的方式来给出指令, 汇编器将会报错, 因为这条特殊的指令是我们人为添加的, 标准的汇编器并不能识别它。如果你查看 objdump 的反汇编结果, 你会看到 nemu_trap 指令被标识为 (bad), 原因是类似的: objdump 并不能识别我们人为添加的 nemu_trap 指令。"a"(0) 表示在执行内联汇编语句给出的汇编代码之前, 先将 0 读入 %eax 寄存器。这样, 这段汇编代码的功能就和 nemu/src/cpu/exec/special/special.c 中的 helper 函数 nemu_trap() 对应起来了。此外, volatile 是 C 语言的一个关键字, 如果你想了解 volatile 的更多信息, 请查阅相关资料。HIT_BAD_TRAP 的功能是类似的, 这里就不再进行叙述了。

最后来看看 nemu_assert(), 它做的事情十分简单, 当条件为假时, 就执行 HIT_BAD_TRAP。这样几行代码就实现了 assertion 的功能, 我们就可以在用户程序中使用 assertion 了。

上述三个宏都有相应的汇编版本, 在汇编代码中包含头文件 trap.h, 你就可以使用它们了。不过汇编版本的 nemu_assert() 功能比较简陋, 它只能判断某个通用寄存器是否与给定的一个立即数相等。

最后, 用户程序从 main 函数返回之后, 会通过 HIT_GOOD_TRAP 强行结束用户程序的运行, 同时也提示我们用户程序通过了所有的 assertion。

思考题 1: main 函数返回到哪里

查看 testcase 的相关代码, 你知道用户程序从 main 函数返回之后会跳转到哪里吗? 如果用户程序在 GNU/Linux 中运行, 问题的答案又是什么?

运行时环境与交叉编译

在让 NEMU 运行用户程序之前, 我们先来讨论 NEMU 需要为用户程序的运行提供什么。在你运行 hello world 程序时, 你敲入一条命令(或者点击一下鼠标), 程序就成功运行了, 但这背后其实隐藏着操作系统开发者和库函数开发者的无数汗水。一个事实是, 应用程序的运行都需要运行时环境的支持, 包括加载, 销毁程序, 以及提供程序运行时的各种动态链接库(你经常使用的库函数就是运行时环境提供的)等。现在轮到你来为用户程序提供运行时环境的支持了, 不用担心,

由于 NEMU 目前的功能并不完善, 我们必定无法向用户程序提供 GNU/Linux 般的运行时环境。目前, 我们约定 NEMU 提供的运行时环境有:

1. 物理内存有 128MB(当然, 这是我们模拟出来的物理内存), 所有内存地址都是物理地址。

2. 程序入口位于地址 0x100000, 程序总是从这里开始执行。

3. %ebp 的初值为 0, %esp 的初值为 0x8000000 附近。需要注意的是, 这个地址是物理内存的最大值, 是一个非法的物理地址, 不能直接访问。

4. 程序通过 nemu_trap 结束运行。

5. 不提供库函数的动态链接, 但提供静态链接, 故实际上对用户程序来说, 库函数的使用与运行时环境无关。库函数的静态链接是通过框架代码中提供的函数库 uclibc 实现的, 相应的文件有 lib-common/uclibc/lib/libc.a 和 lib-common/uclibc/include 目录下的头文件, Makefile 中已经有相应的设置了。uclibc 是一个专门为嵌入式系统提供的 C 库, 库中的函数对运行时环境的要求极低, 其中一些函数甚至不需要任何运行时环境的支持(例如 memcpy 等), 这正好符合 NEMU 的情况。这样, 你就可以在用户程序中使用一些不需要运行时环境支持的库函数了。但类似于 printf() 这种需要运行时环境支持的库函数目前还是无法使用, 否则将会发生链接错误。

在让 NEMU 运行 mov-c 用户程序之前, 我们需要将用户程序的代码 mov-c.c 编译成可执行文件。需要说明的是, 我们不能使用 gcc 的默认选项直接编译 mov-c.c, 因为默认选项会根据 GNU/Linux 的运行时环境将代码编译成运行在 GNU/Linux 下的可执行文件。但此时的 NEMU 并不能为用户程序提供 GNU/Linux 的运行时环境, 在 NEMU 中运行上述可执行文件会产生错误, 因此我们不能使用 gcc 的默认选项来编译用户程序。

解决这个问题的方法是交叉编译, 我们需要在 GNU/Linux 下根据 NEMU 提供的运行时环境编译出能够在 NEMU 中运行的可执行文件。框架代码已经把相应的配置准备好了: gcc 先将源文件编译成目标文件, 然后让 ld 根据链接脚本 testcase/user.ld 将目标文件链接成可执行文件。根据链接脚本 testcase/user.ld 的指示, 可执行程序重定位后的节从 0x100000 开始, 首先是 .text 节, 其中又以 obj/testcase/start.o 中的 .text 节开始, 然后接下来是其它目标文件的 .text 节, 未列出的节将以默认顺序往后依次放置。这样, 可执行程序 0x100000 处总是放置 testcase/src/start.S 的代码, 而不是其它代码, 保证用户程序总能从 0x100000 开始正确执行。

修改工程目录下的 Makefile 文件, 更换 NEMU 的用户程序:

注：
“-”表示删除该行或注释该行。
“+”表示将删除的行替换为该行

make run

```
invalid opcode(eip = 0x0010000a): e8 06 00 00 00 b8 00 00 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x0010000a is not implemented.
2. Something is implemented incorrectly.

Find this eip value(0x0010000a) in the disassembling result to distinguish which case it is

If it is the first case, see

  _ _ _ _ _
( ) _ \ / _ \ / / | \ \ |
_ _ ) | ( ) / / _ | \ / | _ _ _ _ _ _ _ _ |
| | _ < > _ < | ' _ \ | | \ | / _ | ' _ \ | | / _ | | | | |
| | _ ) | ( ) | ( ) | | | ( | | | | | ( | |
| | _ / \ _ / \ _ / | | | \ _ , - | | | \ _ , - | \ _ , - |

for more details.

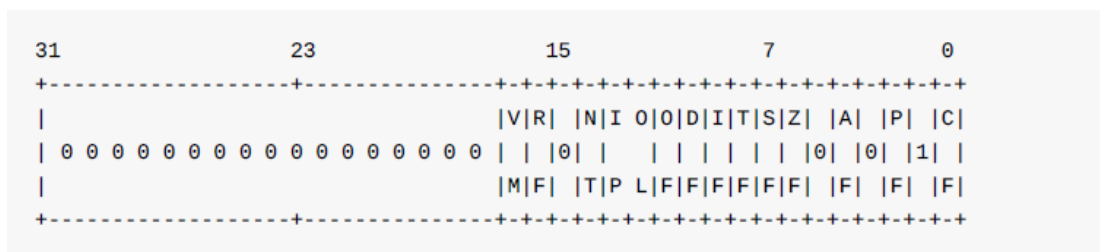
If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!

nemu: nemu/src/cpu/exec/special/special.c:24: inv: Assertion `0' failed.
```

实现最少的指令

要实现哪些指令才能让 `mov-c` 在 NEMU 中运行起来呢？答案就在其反汇编结果(`obj/testcase/mov-c.txt`)中。查看反汇编结果, 你发现只需要添加 `call`, `push`, `test`, `je`, `cmp`, `pop`, `ret` 七条指令就可以了。每一条指令还有不同的形式, 根据 KISS 法则, 你可以先实现只在 `mov-c` 中出现的指令形式, 通过指令的 `opcode` 可以确定具体的形式。

- **call**: **call** 指令有很多形式, 不过在 PA 中只会用到其中的几种, 现在只需要实现 **CALL rel32** 的形式就可以了。
- **push**: 现在只需要实现 **PUSH r32** 的形式就可以了。
- **test**: 在实现 **test** 指令的时候需要使用 **EFLAGS** 寄存器, 定义在寄存器结构体中 (位于 **nemu/include/cpu/reg.h**)。 **EFLAGS** 是一个 32 位寄存器, 它的结构如下:



关于 EFLAGS 中每一位的含义, 请查阅 i386 手册。在 NEMU 中, 我们只会用到 EFLAGS 中以下的 7 个位: **CF, PF, ZF, SF, IF, DF, OF**。其余位的功能可暂不实现。EFLAGS 寄存器的定义使用了结构体的位域(bit field)功能, 如果你从未听说过位域, 请查阅相关资料。关于 EFLAGAS 初值, 遵循 i386 手册中提到的约定, 需要在 i386 手册的**第 10 章**中找到这一初值, 然后在 **restart()**函数中对 EFLAGS 寄存器进行初始化。基于 EFLAGS 寄存器就可以实现 **test** 指令了。此外, NEMU 还定义了函数 **update_eflags_pf_zf_sf**, 位于 **nemu/include/cpu/eflags.h** 和 **nemu/src/cpu/eflags.c** 中, 用于根据指令执行结果更新 PF, ZF 和 SF 位, 在后续指令实现过程中可直接调用。

- **je**: **je** 指令是 **jcc** 的一种形式。
- **cmp**: 要注意被减数和减数的位置。
- **pop, ret**: 阅读 i386 手册吧。

必做任务 1: 运行用户程序 mov-c

编写相应 helper 函数实现上文提到的指令, 具体细节务必参考 i386 手册。实现成功后, 在 NEMU 中运行用户程序 **mov-c**, 会看到 **HIT GOOD TRAP** 的信息。

温馨提示

PA2 阶段 1 到此结束。

融会贯通：实现更多的指令

为了让 NEMU 支持大部分程序的运行，目前你需要实现以下指令：

- Data Move Instructions: `mov xchg movsx movzx cwtl` (在 i386 手册中为 `cbw/cwde/cltd`(在 i386 手册中为 `cwd/cdq`) `push pop leave`
- Binary Arithmetic Instructions: `add cmp` `adc sub sbb inc dec neg` `mul imul div idiv`
- Logical Instructions: `not and or xor sal(shl) shr sar` `setcc test`
- Control Transfer Instructions: `jmp` `call ret jcc`
- String and Character Translation Instructions: `movs stos scas rep` `lods`
- Miscellaneous Instructions: `lea nop`

你只需要实现上述带下划线的指令，它们不多不少都和以下的某些内容相关：EFLAGS, 堆栈, 整数扩展, 加减溢出判断。我们需要针对 `lods` 这条字符串操作指令进行补充说明：这条指令和 `stos`、`scas` 和 `movs` 指令的执行涉及到段寄存器，但我们目前并没有在 NEMU 中实现段寄存器，因此我们先忽略和段寄存器相关的描述。例如 i386 手册中对 `movs` 指令有如下描述：

Move DS:[(E)SI] to ES:[(E)DI]

目前我们只需要理解成

Move [(E)SI] to [(E)DI]

即可

框架代码已经把其它指令实现好了，但有些没有填写 `opcode_table`。此外，某些需要更新 EFLAGS 的指令，以及有符号立即数的译码函数（在 `nemu/src/cpu/decode/decode-template.h` 中定义）并没有完全实现好（框架代码中已经插入了 `panic()` 作为提示），你还需要编写相应的功能。

断点(2)

我们在 PA1 中介绍了如何通过监视点来模拟断点，不过这种方法需要提前知道设置断点的地址，下面来介绍一种不需要提前知道断点地址的设置方法。

在 x86 中有一条叫 `int3` 的特殊指令，它是专门给调试器准备的，一般的程序不应该使用这条指令。当程序执行到 `int3` 指令的时候，CPU 将会抛出一个含义为“程序触发了断点”的异常(exception)，操作系统会捕捉到这个异常，然后操作系统会通过信号机制(signal)，向程序发送一个 SIGTRAP 信号，这样程序就知

道自己触发了一个断点。如果你现在觉得上述过程很难理解, 不必担心, 你只需要知道**当程序执行到 int3 指令的时候, 调试器就能够知道程序触发了断点。**

设置断点, 其实就是在程序中插入 `int3` 指令。`int3` 指令的机器码为 `0xcc`, 长度为一个字节。如果你看过 PA1 中关于断点的阅读材料, 你会发现断点真正的工作原理比较复杂, 根据 KISS 法则, 我们采用一种简单的方法来实现断点的功能。在 `lib-common/trap.h` 中提供了一个函数 `set_bp()`, 它的功能就是马上执行 `int3` 指令。当 NEMU 发现程序执行的是 `int3` 指令时, 输出一句话提示用户触发了断点, 最后返回到 `ui_mainloop()` 循环中等待用户的命令。

框架代码已经实现上述功能了。要使用断点, 你只要在用户程序的源代码中调用 `set_bp()` 函数, 就可以达到设置断点的效果了。你可以在 `testcase/src/mov-c.c` 中插入断点, 然后重新编译 `mov-c` 程序并运行 NEMU 来体会这种断点的设置方法。需要注意的是, 这种简化的做法其实是对 `int3` 指令的滥用, 因为在真实的操作系统中, 一般的程序不应该使用 `int3` 指令, 否则它将会在运行时异常终止。

不过这种方法也有不足之处: 一行 C 代码可能会对很多条机器指令, 因此上述方法并不能在任意位置设置断点。例如我们熟知的函数调用语句, 其对应的机器指令分为压入实参和控制转移两部分, 我们很难使用 `set_bp()` 在程序压入实参后, 执行 `call` 指令前设置断点, 不过这对监视点来说就不在话下了。

测试用例

未测试代码永远是错的, 你需要足够多的测试用例来测试你的 NEMU。我们在 `testcase` 目录下准备了一些测试用例, 需要更换测试用例时, 修改工程目录下 `Makefile` 中的 `USERPROG` 变量, 改成测试用例的可执行文件 (`obj/testcase/xxx`, 不是 C 源文件) 即可。

必做任务 2: 实现更多指令

你需要实现上文中提到的更多指令, 以支持 `testcase` 目录下更多程序的运行。实现的时候尽可能使用框架代码中的宏(参考 `include/cpu/exec/helper.h` 和 `include/cpu/exec/template-start.h`), 它们可以帮助你编写出简洁的代码。

你可以自由选择按照什么顺序来实现指令。经过 PA1 的训练之后, 你应该不会实现所有指令之后才进行测试了, 要养成尽早做测试的好习惯, 一般原则都是"实现尽可能少的指令来进行下一次的测试"。你不需要实现所有指令的所有形式, 只需要通过 `testcase` 目录下的测试就可以了。由于部分测试用例需要实现较多指令, 建议按照以下顺序进行测试:

1. 其它; 2. struct; 3. string; 4. hello-str

`testcase` 目录下的大部分测试用例都可以直接在 NEMU 上运行, 除了以下几个测试用例:

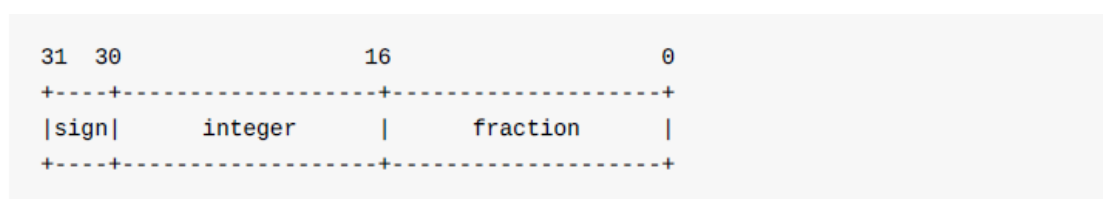
- `hello-inline-asm`
- `hello`
- `integral`
- `quadratic-eq`
- `print-FLOAT`

其中运行 `hello-inline-asm` 和 `hello` 需要系统调用的支持, 在 PA2 中我们无法提供系统调用的功能, 这两个测试用例将会在 PA4 中用到, 目前你可以忽略它们; 要运行 `print-FLOAT` 需要使一些与运行时代码相关的小技巧, 我们在 PA2 的最后再来讨论如何运行它; 而 `integral` 和 `quadratic-eq` 涉及到浮点数的使用, 我们先来讨论如何处理浮点数。

实现 binary scaling (浮点数定点化)

要在 NEMU 中实现浮点指令也不是不可能的事情。但实现浮点指令需要涉及 x87 架构的很多细节, 而且我们并不打算在用户程序中直接使用浮点指令。为了在保持程序逻辑的同时不引入浮点指令, 我们通过整数来模拟实数的运算, 这样的方法叫 binary scaling (定点化)。

我们先来说明如何用 32 位整数来表示一个实数。为了方便叙述, 我们称用 binary scaling 方法表示的实数的类型为 `FLOAT`。我们约定最高位为符号位, 接下来的 15 位表示整数部分, 低 16 位表示小数部分, 即约定小数点在第 15 和第 16 位之间(从第 0 位开始)。从这个约定可以看到, `FLOAT` 类型其实是实数的一种定点表示。



这样, 对于一个实数 `a`, 它的 `FLOAT` 类型表示 $A = a * 2^{16}$ (截断结果的小数部分)。例如实数 `1.2` 和 `5.6` 用 `FLOAT` 类型来近似表示, 就是

$$1.2 * 2^{16} = 78643 = 0x13333$$

+---+-----+-----+-----+			
0	1	3333	
+---+-----+-----+-----+			

$$5.6 * 2^{16} = 367001 = 0x59999$$

+---+-----+-----+-----+			
0	5	9999	
+---+-----+-----+-----+			

而实际上, 这两个 **Float** 类型数据表示的数是:

$$0x13333 / 2^{16} = 1.19999695$$

$$0x59999 / 2^{16} = 5.59999084$$

对于负实数, 我们用相应正数的相反数来表示, 例如 **-1.2** 的 **Float** 类型表示为:

$$-(1.2 * 2^{16}) = -0x13333 = 0xfffecccd$$

思考题 2: 比较 **Float** 和 **float**

Float 和 **float** 类型的数据都是 32 位, 它们都可以表示 2^{32} 个不同的数, 但由于表示方法不一样, **Float** 和 **float** 能表示的数集是不一样的。思考一下, 我们用 **Float** 来模拟表示 **float**, 这其中隐含着哪些取舍?

接下来我们来考虑 **Float** 类型的常见运算, 假设实数 **a, b** 的 **Float** 类型表示分别为 **A, B**。

- 由于我们使用整数来表示 **Float** 类型, **Float** 类型的加法可以直接用整数加法来进行:

$$A + B = a * 2^{16} + b * 2^{16} = (a + b) * 2^{16}$$

- 由于我们使用补码的方式来表示 **Float** 类型数据, 因此 **Float** 类型的减法用整数减法来进行。

$$A - B = a * 2^{16} - b * 2^{16} = (a - b) * 2^{16}$$

- **Float** 类型的乘除法和加减法就不一样了：

$$A * B = a * 2^{16} * b * 2^{16} = (a * b) * 2^{32} \neq (a * b) * 2^{16}$$

也就是说, 直接把两个 **Float** 数据相乘得到的结果并不等于相应的两个浮点数乘积的 **Float** 表示。为了得到正确的结果, 我们需要对相乘的结果进行调整: 只要将结果除以 2^{16} , 就能得出正确的结果了。除法也需要对结果进行调整, 至于如何调整, 当然难不倒聪明的你啦!

- 如果把 $A = a * 2^{16}$ 看成一个映射, 那么在这个映射的作用下, 关系运算是保序的, 即 $a \leq b$ 当且仅当 $A \leq B$, 故 **Float** 类型的关系运算可以用整数的关系运算来进行。

有了这些结论, 要用 **Float** 类型来模拟实数运算就很方便了。除了乘除法需要额外实现之外, 其余运算都可以直接使用相应的整数运算来进行。例如

```
float a = 1.2;
float b = 10;
int c = 0;
if(b > 7.9) {
    c = (a + 1) * b / 2.3;
}
```

用 **Float** 类型来模拟就是

```
Float a = f2F(1.2);
Float b = int2F(10);
int c = 0;
if(b > f2F(7.9)) {
    c = F2int(F_div_F(F_mul_F((a + int2F(1)), b), f2F(2.3)));
}
```

其中还引入了一些类型转换函数来实现和 **Float** 相关的类型转换。

必做任务 3：实现 binary scaling

框架代码已经将测试用例中涉及浮点数的部分用 `FLOAT` 类型来模拟，你需要实现一些和 `FLOAT` 类型相关的函数：

```
/* lib-common/FLOAT.h */
int32_t F2int(FLOAT a);
FLOAT int2F(int a);
FLOAT F_mul_int(FLOAT a, int b);
FLOAT F_div_int(FLOAT a, int b);
/* lib-common/FLOAT/FLOAT.c */
FLOAT f2F(float a);
FLOAT F_mul_F(FLOAT a, FLOAT b);
FLOAT F_div_F(FLOAT a, FLOAT b);
FLOAT Fabs(FLOAT a);
```

其中 `F_mul_int()` 和 `F_div_int()` 用于计算一个 `FLOAT` 类型数据和一个整型数据的积/商，这两种特殊情况可以快速计算出结果，不需要将整型数据先转化成 `FLOAT` 类型再进行运算。`lib-common/FLOAT/FLOAT.c` 中的 `pow()` 函数目前不会用到，我们会在 PA4 再提到它。实现成功后，你还需要在 `lib-common/FLOAT/Makefile.part` 中编写用于分别生成 `FLOAT.o` 和 `FLOAT_vfprintf.o` (`FLOAT_vfprintf.o` 的生成为选做) 的规则，要求如下：

- 只编译不链接
- 使用 `-m32`, `-O2` 和 `-fno-builtin` 编译选项。此外，对于 `FLOAT_vfprintf.o` 还需要额外添加两个编译选项 `-fno-stack-protector` 和 `-D_FORTIFY_SOURCE=0`
- 添加 `lib-common` 目录作为头文件的搜索路径
- 把 `FLOAT.o` 和 `FLOAT_vfprintf.o` 生成到在 `obj/lib-common` 目录下，若目录不存在，可以先通过 `mkdir -p` 目录路径名 创建

`FLOAT_vfprintf.c` 中的代码会在运行 `print-FLOAT` 测试用例时被用到，我们在 PA2 的最后再进一步说明，目前你只需要通过相应的规则将其编译为目标文件，`Makefile` 就会将其加入到 `FLOAT.a` 中，将来链接的时候就能够找到相应的函数。编写规则后，修改工程目录下的 `Makefile` 文件：

```
--- Makefile
+++ Makefile
@@ -12,2 +12,2 @@
LIBC = $(LIBC_LIB_DIR)/libc.a
-#FLOAT = obj/$(LIB_COMMON_DIR)/FLOAT/FLOAT.a
+#FLOAT = obj/$(LIB_COMMON_DIR)/FLOAT/FLOAT.a
```

让 `FLOAT.a` 参与链接，这样你就可以在 NEMU 中运行 `integral` 和 `quadratic-eq` 这两个测试用例了。

事实上，我们并没有考虑计算结果溢出的情况，不过我们的测试用例中的浮点数结果都可以在 `float` 类型中表示，所以你可以不关心溢出的问题。如果你不放心，你可以在上述函数的实现中插入 `assertion` 来捕捉溢出错误。

编写自己的测试用例

从测试的角度来说，`testcase` 目录下的测试用例还不够完备，很多指令可能都没有被覆盖到。想象一下你编写了一个 `if` 语句，但程序运行的时候根本就没有进入过这个 `if` 块中，你怎么好意思说你写的这个 `if` 语句是对的呢？因此我们鼓励你编写自己的测试用例，尽可能地覆盖到你写的所有代码。用户程序的来源有很多，例如程序设计作业中的小程序，或者是已经完成的数据结构作业等等。但你需要注意的是 NEMU 提供的运行时环境，用户程序不能输出，只能通过 `nemu_assert()` 来进行验证。你可以按照以下步骤编写一个测试用例：

- 先使用 `printf()` 根据数组的格式输出测试结果，此时你编写的是一个运行在 GNU/Linux 下的程序，直接用 `gcc` 编译即可。
- 运行程序，得到了数组格式的输出，然后把这些输出结果作为全局数组添加到源代码中，你可以通过 `>>` 将输出重定向追加到源代码中。
- 去掉代码中 `printf()` 和头文件，包含 `trap.h`，使用 `nemu_assert()` 进行验证。
- 把修改后的 `.c` 文件放到 `testcase/src` 目录下即可。

这样你就成功添加了一个测试用例了，按照上文提到的步骤更换测试用例，就可以使用你的测试用例进行测试了。

我们还鼓励你把测试用例分享给大家，同时也可以使用其它同学提供的测试用例进行测试。你的程序通过越多的测试，程序的健壮性就越好，越有希望通过“在 NEMU 上运行仙剑奇侠传”的终极考验。

温馨提示

PA2 阶段 2 结束。此阶段需要实现较多指令，请大家耐心阅读 i386 手册，尽量使用代码框架中提供的宏。每个同学独立完成，不要畏惧困难。

第三视点: 简易调试器(2)

接下来, 我们将会对简易调试器的功能进行扩展, 为调试提供更多的手段。

运行用户程序 add

在继续之前, 请保证用户程序 add 可以在 NEMU 中正确运行。你将使用 add 程序来测试简易调试器的新功能。

添加变量支持

你已经在 PA1 中实现了简易调试器, 现在你已经将用户程序换成了 C 程序。和之前的 `mov.S` 相比, C 程序多了变量和函数的要素, 那么在表达式求值中如何支持变量的输出呢?

```
(nemu) p test_data
```

换句话说, 我们怎么从 `test_data` 这个字符串找到这个变量在运行时刻的信息? 下面我们就来讨论这个问题。

符号表(symbol table)是可执行文件的一个 section, 它记录了程序编译时刻的一些信息, 其中就包括变量和函数的信息。为了完善调试器的功能, 我们首先需要了解符号表中都记录了哪些信息。

以 add 这个用户程序为例, 使用 `readelf` 命令查看 ELF 可执行文件的信息:

```
readelf -a add
```

你会看到 `readelf` 命令输出了很多信息, 这些信息对了解 ELF 的结构有很好的帮助, 我们建议你在课后仔细琢磨。目前我们只需要关心符号表的信息就可以了, 在输出中找到符号表的信息:

Symbol table '.symtab' contains 10 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00100000	0	SECTION	LOCAL	DEFAULT	1	
2:	0010009c	0	SECTION	LOCAL	DEFAULT	2	
3:	00100100	0	SECTION	LOCAL	DEFAULT	3	
4:	00000000	0	SECTION	LOCAL	DEFAULT	4	
5:	00000000	0	FILE	LOCAL	DEFAULT	ABS	add.c
6:	00100084	22	FUNC	GLOBAL	DEFAULT	1	add
7:	00100000	129	FUNC	GLOBAL	DEFAULT	1	main
8:	00100120	256	OBJECT	GLOBAL	DEFAULT	3	ans
9:	00100100	32	OBJECT	GLOBAL	DEFAULT	3	test_data

其中每一行代表一个表项, 每一列列出了表项的一些属性, 现在我们只需要关心 **Type** 属性为 **OBJECT** 的表项就可以了。仔细观察 **Name** 属性之后, 你会发现这些表项正好对应了 **add.c** 中定义的全局变量, 而相应的 **Value** 属性正好是它们的地址(你可以与 **add.txt** 中的反汇编结果进行对比), 而找到地址之后就可以找到这个变量了。

思考题 3: 消失的符号

我们在 **add.c** 中定义了宏 **NR_DATA**, 同时也在 **add()** 函数中定义了局部变量 **c** 和形参 **a, b**, 但你会发现在符号表中找不到和它们对应的表项, 为什么会这样? 思考一下, 什么才算是一个符号(symbol)?

太好了, 我们可以通过符号表建立变量名和其地址之间的映射关系! 别着急, **readelf** 输出的信息是已经经过解析的, 实际上符号表中 **Name** 属性存放的是字符串在字符串表(string table)中的偏移量。为了查看字符串表, 我们先查看 **readelf** 输出中 Section Headers 的信息:

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	00100000	001000	00009a	00	AX	0	0	4
[2]	.eh_frame	PROGBITS	0010009c	00109c	000058	00	A	0	0	4
[3]	.data	PROGBITS	00100100	001100	000120	00	WA	0	0	32
[4]	.comment	PROGBITS	00000000	001220	00001c	01	MS	0	0	1
[5]	.shstrtab	STRTAB	00000000	00123c	00003a	00		0	0	1
[6]	.symtab	SYMTAB	00000000	0013b8	0000a0	10		7	6	4
[7]	.strtab	STRTAB	00000000	001458	00001e	00		0	0	1

从 Section Headers 的信息可以看到, 字符串表在 ELF 文件偏移为 **0x1458** 的位置开始存放. 在终端中可以通过以下命令直接输出 ELF 文件的十六进制形式:

```
hd add
```

查看输出结果的最后几行, 我们可以看到, 字符串表只不过是把标识符的字符串拼接起来而已。现在我们可以厘清符号表和字符串表之间的关系了:

```
Section Headers:
[Nr] Name                Type           Addr      Off      Size    ES Flg Lk Inf Al
[ 7] .strtab              STRTAB        00000000  001458  00001e  00      0  0  1

                                |
                                +-----+
                                V                V
00001450  20 00 00 00 11 00 03 00 00 61 64 64 2e 63 00 61 | .....add.c.a| |
00001460  64 64 00 6d 61 69 6e 00 61 6e 73 00 74 65 73 74 |dd.main.ans.test| |
00001470  5f 64 61 74 61 00      ^                ^      |_data.| |
                                |                |
                                |                +-----+
                                |                |
                                +-----+
Symbol table '.symtab' contains 10 entries:
Num:   Value      Size Type   Bind  Vis      Ndx Name | |
5: 00000000      0 FILE   LOCAL DEFAULT ABS 1  | |
6: 00100084     22 FUNC   GLOBAL DEFAULT 1 7  ---+---+
7: 00100000    129 FUNC   GLOBAL DEFAULT 1 11  | |
8: 00100120    256 OBJECT GLOBAL DEFAULT 3 16  ---+
9: 00100100     32 OBJECT GLOBAL DEFAULT 3 20  -----+
```

一种解决方法已经呼之欲出了: 在表达式递归求值的过程中, 如果发现 token 的类型是一个标识符, 就通过这个标识符在符号表中找到一项符合要求的表项(表项的 **Type** 属性是 **OBJECT**, 并且将 **Name** 属性的值作为字符串表中的偏移所找到的字符串和标识符的命名一致), 找到标识符的地址, 并将这个地址作为结果返回。在上述 add 程序的例子中:

```
(nemu) p test_data
0x100100
```

需要注意的是, 如果标识符是一个基本类型变量, 简易调试器和 GDB 的处理会有所不同: 在 GDB 中会直接返回基本类型变量的值, 但我们在表达式求值中并没有实现类型系统, 因此我们无法区分一个标识符是否基本类型变量, 所以我们统一输出变量的地址。如果对于一个整型变量 **x**, 我们可以通过以下方式输出它的值:

```
(nemu) p *x
```

而对于一个整型数组 `A`, 如果想输出 `A[1]` 的值, 可以通过以下方式:

```
(nemu) p *(A + 4)
```

必做任务 4: 为表达式求值添加变量的支持

根据上文提到的方法, 向表达式求值添加变量的支持, 为此, 你还需要在表达式求值的词法分析和递归求值中添加对变量的识别和处理。框架代码提供的 `load_table()` 函数已经为你从可执行文件中抽取出符号表和字符串表了, 其中 `strtab` 是字符串表, `symtab` 是符号表, `nr_symtab_entry` 是符号表的表项数目, 更多的信息请阅读 `nemu/src/monitor/debug/elf.c`。

头文件 `<elf.h>` 已经为我们定义了与 ELF 可执行文件相关的数据结构, 为了使用符号表, 请查阅

`man 5 elf`

实现之后, 你就可以在表达式中使用变量了。在 NEMU 中运行 `add` 程序, 并打印全局数组某些元素的值。

打印栈帧链

我们知道函数调用会在堆栈上形成栈帧, 记录和这次函数调用有关的信息。若干次连续的函数调用将会在堆栈上形成一条栈帧链, 为调试提供了很多有用的信息: `%eip` 可以让你知道程序现在的位置, 栈帧链则可以告诉你, 程序是怎么运行到现在的位置的。我们需要在简易调试器中添加 `bt` 命令, 打印出栈帧链的信息, 如果你从来没有使用过 `bt` 命令, 请先在 GDB 中尝试。

在堆栈中形成的栈帧链结构如下:



可以看到, `%ebp` 在栈帧链的组织中起到了非常重要的作用, 通过 `%ebp`, 我们就可以找到每一个栈帧的信息了。聪明的你也许一眼就看出来, 这不就是数据结构中学过的链表吗? 我们可以定义一个结构体来进一步厘清其中的奥妙:

```
typedef struct {
    swaddr_t prev_ebp;
    swaddr_t ret_addr;
    uint32_t args[4];
} PartOfStackFrame;
```

其中 `prev_ebp` 就类似于 next 指针, 不过我们没有将它定义成指针类型, 这是因为它表示的地址是用户程序的地址, 直接把它作为 NEMU 的地址来进行解引用就会发生错误, 所以这个结构体中的每一个成员都需要通过 `swaddr_read()` 来读取。 `args` 成员数组表示函数的实参, 实际上实参的个数不一定是 4 个, 但我们仍然可以将它们强制打印出来, 说不定可以从中发现一些有用

的调试信息。链表的表头存储在 `%ebp` 寄存器中, 所以我们可以从 `%ebp` 寄存器开始, 像遍历链表那样逐一扫描并打印栈帧链中的信息。链表通过 `NULL` 指示链表的结束, 在栈帧链中也是类似的。还记得 NEMU 提供的运行时环境吗? `%ebp` 寄存器的初值为 `0`, 当我们发现栈帧中 `%ebp` 的信息为 `0` 时, 就表示已经达到最开始运行的函数了。

由于缺乏形参和局部变量的具体信息, 我们只需要打印地址, 函数名, 以及前 4 个参数就可以了, 打印格式可以参考 GDB 中 `bt` 命令的输出。如何确定某个地址落在哪一个函数中呢? 这就需要符号表的帮助了:

```
Symbol table '.symtab' contains 10 entries:
Num:   Value   Size Type   Bind   Vis     Ndx Name
  0: 00000000    0 NOTYPE  LOCAL  DEFAULT UND
  1: 00100000    0 SECTION LOCAL  DEFAULT 1
  2: 0010009c    0 SECTION LOCAL  DEFAULT 2
  3: 00100100    0 SECTION LOCAL  DEFAULT 3
  4: 00000000    0 SECTION LOCAL  DEFAULT 4
  5: 00000000    0 FILE    LOCAL  DEFAULT ABS add.c
  6: 00100084   22 FUNC    GLOBAL DEFAULT 1 add
  7: 00100000  129 FUNC    GLOBAL DEFAULT 1 main
  8: 00100120  256 OBJECT GLOBAL DEFAULT 3 ans
  9: 00100100   32 OBJECT GLOBAL DEFAULT 3 test_data
```

对于 `Type` 属性为 `FUNC` 的表项, `Value` 属性指示了函数的起始地址, `Size` 属性指示了函数的大小, 通过这两个属性就可以确定函数的范围了。由于函数的范围是互不相交的, 因此我们可以通过扫描符号表中 `Type` 属性为 `FUNC` 的每一个表项, 唯一确定一个地址在哪个函数中。为了得到函数名, 你只需要根据表项中的 `Name` 属性在字符串表中找到相应的字符串就可以了。

选做任务 1: 打印栈帧链

为简易调试器添加 `bt` 命令, 实现打印栈帧链的功能。实现之后, 在 `add` 的 `add()` 函数中设置断点, 触发断点之后, 在 `monitor` 中测试 `bt` 命令的实现是否正确。

温馨提示

PA2 阶段 3 到此结束。

三生万物: 实现加载程序的 loader

loader 是一个用于加载程序的模块, 实现了足够多的指令之后, 你也可以实现一个很简单的 loader, 来加载其它用户程序。

可执行文件的组织

我们知道程序中包括代码和数据, 它们都是存储在可执行文件中。加载的过程就是把可执行文件中的代码和数据放置在正确的内存位置, 然后跳转到程序入口, 程序就开始执行了。更具体的, 我们需要解决以下问题:

- 可执行文件在哪里?
- 代码和数据在可执行文件的哪个位置?
- 代码和数据有多少?
- "正确的内存位置"在哪里?

我们在 PA1 中已经提到了以下内容:**在一个完整的模拟器中, 程序应该存放在磁盘中, 但目前我们并没有实现磁盘的模拟, 因此 NEMU 先把内存开始的位置作为 ramdisk 来使用。**现在的 ramdisk 十分简单, 它只有一个文件, 也就是我们将来加载的用户程序, 访问内存位置 0 就可以得到用户程序的第一个字节。这其实已经回答了上述第一个问题: 可执行文件位于内存位置 0。为了回答剩下的问题, 我们首先需要了解可执行文件是如何组织的。

代码和(静态)数据是程序的必备要素, 可执行文件中自然需要包含这两者。但仅仅包含它们还是不够的, 我们还需要一些额外的信息来告诉我们"代码和数据分别有多少", 否则我们连它们两者的分界线在哪里都不知道。这些额外的信息描述了可执行文件的组织形式, 不同组织形式形成了不同格式的可执行文件, 例如 Windows 主流的可执行文件是 [PE\(Portable Executable\)](#) 格式, 而 GNU/Linux 主要使用 [ELF\(Executable and Linkable Format\)](#) 格式, 因此一般情况下, 你不能在 Windows 下把一个可执行文件拷贝到 GNU/Linux 下执行, 反之亦然。ELF 是 GNU/Linux 可执行文件的标准格式, 这是因为 GNU/Linux 遵循 [System V ABI\(Application Binary Interface\)](#)。

思考题 4: 堆和栈在哪里?

我们提到了代码和数据都在可执行文件里面, 但却没有提到堆(heap)和栈(stack)。为什么堆和栈的内容没有放入可执行文件里面? 那程序运行时刻用到的堆和栈又是怎么来的?

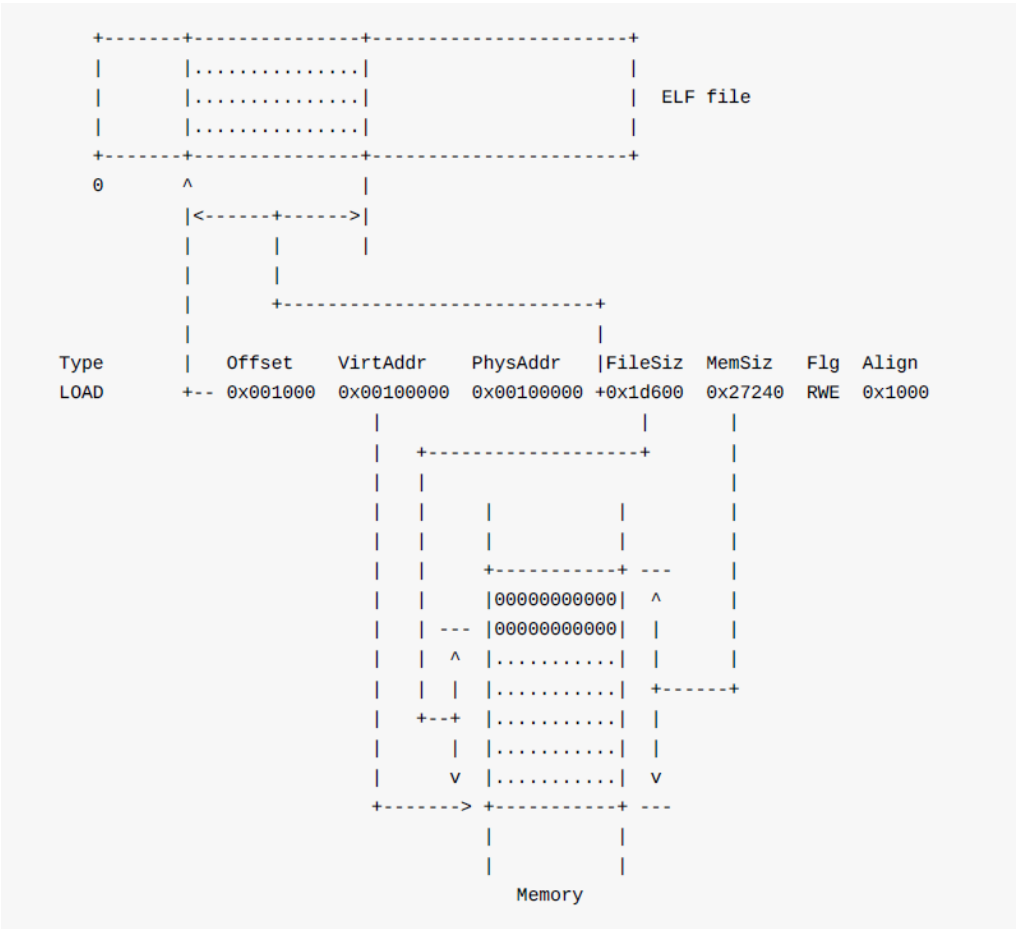
思考题 5: 如何识别不同格式的可执行文件?

如果你在 GNU/Linux 下执行一个从 Windows 拷过来的可执行文件, 将会报告"格式错误"。思考一下, GNU/Linux 是如何知道"格式错误"的?

你应该已经学会使用 `readelf` 命令来查看 ELF 文件的信息了。ELF 文件提供了两个视角来组织一个可执行文件, 一个是面向链接过程的 section 视角, 这个视角提供了用于链接与重定位的信息(例如符号表); 另一个是面向执行的 segment 视角, 这个视角提供了用于加载可执行文件的信息。通过 `readelf` 命令, 我们还可以看到 section 和 segment 之间的映射关系: 一个 segment 可能由 0 个或多个 section 组成, 但一个 section 可能不被包含于任何 segment 中。

我们现在关心的是如何加载程序, 因此我们重点关注 segment 的视角。ELF 中采用 `program header table (程序头表)` 来管理 segment, `program header table` 的一个表项描述了一个 segment 的所有属性, 包括类型, 虚拟地址, 标志, 对齐方式, 以及文件内偏移量和 segment 大小。根据这些信息, 我们就可以知道需要加载可执行文件的哪些字节了, 同时我们也可以看到, 加载一个可执行文件并不是加载它所包含的所有内容, 只要加载那些与运行时刻相关的内容就可以了 (Type 为 LOAD), 例如调试信息和符号表就不必加载。由于运行时环境的约束, 在 PA 中我们只需要加载代码段和数据段, 如果你在 GNU/Linux 下编译一个 Hello world 程序并使用 `readelf` 查看, 你会发现它需要加载更多的内容。

我们通过下面的图来说明如何根据 segment 的属性来加载它:



思考题 6：冗余的属性？

使用 `readelf` 查看一个 ELF 文件的信息，你会看到一个 segment 包含两个大小的属性，分别是 `FileSiz` 和 `MemSiz`，这是为什么？再仔细观察一下，你会发现 `FileSiz` 通常不会大于相应的 `MemSiz`，这又是为什么？

你需要找出每一个 program header 的 `Offset`, `VirtAddr`, `FileSiz` 和 `MemSiz` 这些参数。其中相对文件偏移 `Offset` 指出相应 segment 的内容从 ELF 文件的第 `Offset` 字节开始，在文件中的大小为 `FileSiz`，它需要被分配到以 `VirtAddr` 为首地址的虚拟内存位置，在内存中它占用大小为 `MemSiz`。但现在 NEMU 还没有虚拟地址的概念，因此你只需要把 `VirtAddr` 当做物理地址来使用就可以了，也就是说，这个 segment 使用的内存就是 `[VirtAddr, VirtAddr + MemSiz)` 这一连续区间，然后将 segment 的内容从 ELF 文件中读入到这一内存区间，并将 `[VirtAddr + FileSiz, VirtAddr + MemSiz)` 对应的物理区间清零。

关于程序从何而来，可以参考一篇文章：[COMPILER, ASSEMBLER, LINKER AND LOADER: A BRIEF STORY](#)。

在 kernel 中实现 loader

理解了上述内容之后，你就可以在 kernel 中实现 loader 了。在这之前，我们先对 kernel 作一些简单的介绍。

在 PA 中，kernel 是一个单任务微型操作系统的内核，其代码在工程目录的 kernel 目录下。kernel 已经包含了后续 PA 用到的所有模块，换句话说，我们现在就要在 NEMU 上运行一个操作系统了(尽管这是一个十分简陋的操作系统)，同时也将带领你根据课堂上的知识剖析一个简单操作系统内核的组成。由于 NEMU 的功能是逐渐添加的，kernel 也要配合这个过程，你会通过 `kernel/include/common.h` 中的一些与实验进度相关的宏来控制 kernel 的功能。随着实验进度的推进，我们会逐渐讲解所有的模块，kernel 做的工作也会越来越多。因此在阅读 kernel 的代码时，你只需要关心和当前进度相关的模块就可以了，不要纠缠于和当前进度无关的代码。

另外需要说明的是，虽然不会引起明显的误解，但在引入 kernel 之后，我们还是会在某些地方使用“用户进程”的概念，而不是“用户程序”。如果你现在不能理解什么是进程，你只需要把进程作为“正在运行的程序”来理解就可以了。还感觉不出这两者的区别？举一个简单的例子吧，如果你打开了记事本 3 次，计算机上就会有 3 个记事本进程在运行，但磁盘中的记事本程序只有一个。进程是操作系统中一个重要的概念，有关进程的详细知识会在操作系统课上进行介绍。在工程目录下执行 `make kernel` 来编译 `kernel`。`kernel` 的源文件组织如下：

```

kernel
├── include
│   ├── common.h
│   ├── debug.h
│   ├── memory.h
│   └── x86
│       ├── cpu.h
│       ├── io.h
│       └── memory.h
│   └── x86.h
├── Makefile.part
└── src
    ├── driver
    │   ├── ide          # IDE驱动程序
    │   └── ...
    │   └── ramdisk.c    # ramdisk驱动程序
    ├── elf              # loader相关
    │   └── elf.c
    ├── fs               # 文件系统
    │   └── fs.c
    ├── irq              # 中断处理相关
    │   ├── do_irq.S     # 中断处理入口代码
    │   ├── i8259.c      # intel 8259中断控制器
    │   ├── idt.c        # IDT相关
    │   └── irq_handle.c # 中断分发和处理
    ├── lib
    │   ├── misc.c       # 杂项
    │   ├── printk.c
    │   └── serial.c     # 串口
    ├── main.c
    ├── memory           # 存储管理相关
    │   ├── kvm.c        # kernel虚拟内存
    │   ├── mm.c         # 存储管理器MM
    │   ├── mm_malloc.o  # 为用户程序分配内存的接口函数，不要删除它！
    │   └── vmem.c       # video memory
    ├── start.S          # kernel入口代码
    ├── syscall          # 系统调用处理相关
    └── do_syscall.c

```

一开始 `kernel/include/common.h` 中所有与实验进度相关的宏都没有定义，此时 `kernel` 的功能十分简单。我们先简单梳理一下此时 `kernel` 的行为：

1. 第一条指令从 `kernel/src/start.S` 开始，设置好堆栈之后就跳转到 `kernel/src/main.c` 的 `init()` 函数处执行。

2. 由于 `NEMU` 还没有实现分段分页的功能，此时 `kernel` 会跳过很多初始化工作，直接跳转到 `init_cond()` 函数处继续进行初始化。

3. 继续跳过一些初始化工作之后，会通过 `Log()` 宏输出一句话。需要说明的是，`kernel` 中定义的 `Log()` 宏并不是 `NEMU` 中定义的 `Log()` 宏，`kernel` 和 `NEMU` 的代码是相互独立的，因此编译某一方时都不会受到对方代码的影响，你在阅读代码的时候需要注意这一点。在 `kernel` 中，`Log()` 宏通过 `printk()` 输出。阅读

`printk()`的代码,发现此时 `printk()`什么都不做就直接返回了,这是由于现在 NEMU 还不能提供输出的功能,因此现在 kernel 中的 `Log()`宏并不能成功输出。

4. 调用 `loader()`加载用户程序, `loader()`函数会返回用户程序的入口地址。
5. 跳转到用户程序的入口执行。

理解上述过程后,你需要在 `kernel/src/elf/elf.c` 的 `loader()`函数中定义正确 ELF 文件魔数,然后编写加载 segment 的代码,完成加载用户程序的功能。你需要使用 `ramdisk_read()`函数来读出 `ramdisk` 中的内容, `ramdisk_read()`函数的原型如下:

```
void ramdisk_read(uint8_t *buf, uint32_t offset, uint32_t len);
```

它负责把从 `ramdisk` 中 `offset` 偏移处的 `len` 字节读入到 `buf` 中。

我们之前让用户程序直接在 `0x100000` 处运行,而现在我们希望先从 `0x100000` 处运行 kernel,让 kernel 中的 `loader` 模块加载用户程序,然后跳转到用户程序处运行。为此,我们需要修改用户程序的链接选项:

```
--- testcase/Makefile.part
+++ testcase/Makefile.part
@@ -8,2 +8,2 @@
testcase_START_OBJ := $(testcase_OBJ_DIR)/start.o
-testcase_LDFLAGS := -m elf_i386 -T testcase/user.ld
+testcase_LDFLAGS := -m elf_i386 -e main -Ttext-segment=0x00800000
```

我们让用户程序从 `0x800000` 附近开始运行,避免和 kernel 的内容产生冲突。最后我们还需要修改工程目录下的 Makefile,把 kernel 作为 `entry`:

```
--- Makefile
+++ Makefile
@@ -56,2 +56,2 @@
USERPROG = obj/testcase/mov-c
-ENTRY = $(USERPROG)
+ENTRY = $(kernel_BIN)
```

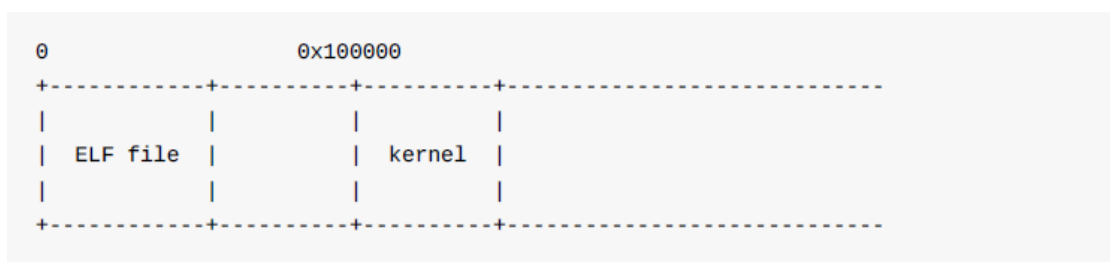
我们从运行时刻的角度来描述 NEMU 中物理内存的变化过程:

- 刚开始运行 NEMU 时, NEMU 会进行一些全局的初始化操作,其中有一项内容是初始化 `ramdisk`(由 `nemu/src/monitor/monitor.c` 中的 `init_ramdisk()`函数完成): 将用户程序的 ELF 文件从真实磁盘拷贝到模拟内存中地址为 0 的位置。目前 `ramdisk` 中只有一个文件,就是用户程

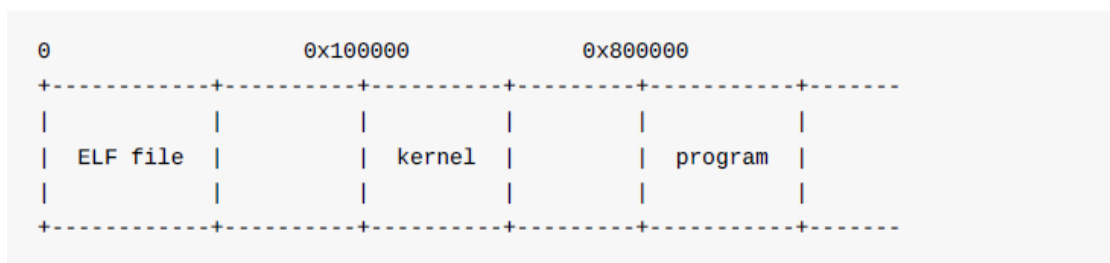
序的 ELF 文件:



- 第二项初始化操作是将 `entry` 加载到内存位置 `0x100000`(由 `nemu/src/monitor/monitor.c` 中的 `load_entry()` 函数完成)。之前 `entry` 就是用户程序本身, 引入 kernel 之后, `entry` 就变成 kernel 了。换句话说, 在我们的 PA 中, kernel 是由 NEMU 的 `monitor` 模块直接加载到内存中的。这一点与真实的计算机有所不同, 在真实的计算机中, 计算机启动之后会首先运行 BIOS 程序, BIOS 程序会将 MBR (引导代码) 加载到内存, MBR 再加载别的程序... 最后加载 kernel。要模拟这个过程需要一个完善的模拟磁盘, 而且需要涉及较多的细节, 根据 KISS 法则, 我们不模拟这个过程, 而是让 NEMU 直接将 kernel 加载到内存。这样, 在 NEMU 开始执行第一条指令之前, kernel 就已经在内存中了:



- NEMU 执行的第一条指令就是 kernel 的第一条指令。如上文所述, kernel 会完成一些自身的初始化工作, 然后从 ramdisk 中的 EFL 文件把用户程序加载到内存位置 `0x800000` 附近, 然后跳转到用户程序中执行:



必做任务 5：实现 loader

你需要在 kernel 中实现 loader 的功能，让 NEMU 从 kernel 开始执行。kernel 的 loader 模块负责把用户程序加载到正确的内存位置，然后执行用户程序。如果你发现你编写的 loader 没有正常工作，你可以使用 `nemu_assert()` 和简易调试器进行调试。

实现 loader 之后，你就可以使用 `test.sh` 脚本进行测试了，在工程目录下运行

make test

脚本会将 `testcase` 中的测试用例逐个进行测试，并报告每个测试用例的运行结果，这样你就不需要手动切换测试用例了。如果一个测试用例运行失败，脚本将会保留相应的日志文件；当使用脚本通过这个测试用例的时候，日志文件将会被移除。

温馨提示

PA2 阶段 4 到此结束。

扭转乾坤：改变程序的行为（选做任务 2）

`print-FLOAT` 程序的功能是通过 `sprintf()` 函数的 `%f` 功能来格式化一个 `FLOAT` 类型的变量，然后使用 `strcmp()` 函数来检查字符串是否正确。等等，`%f` 不是用来格式化一个 `float` 类型的变量的吗？那该如何实现让 `%f` 来格式化一个 `FLOAT` 类型的变量？

你可能会想修改 `uclibc` 的源代码，然后重新编译。这是一个能行的方法，但需要进行库函数的交叉编译，代价比较大，有兴趣的同学可以到 `uclibc` 的官网上下载源代码并尝试修改。

既然不修改 `uclibc` 的源代码，那么就从运行时的代码下手吧！这正是这次任务最有魅力的地方：你将要通过对 `libc.a` 进行攻击，劫持相应的执行流，来改变 `%f` 的行为！

为了方便调试，我们先在 GNU/Linux 中实施攻击，成功后再将程序移植到 NEMU 中运行。框架代码已经为我们准备好生成相应文件的配置了（见 `testcase/Makefile.part`）。在工程目录下执行

```
make pa2-7
```

就会生成 `obj/testcase/print-FLOAT-linux` 这一可以在 GNU/Linux 中直接运行的可执行文件。生成它时，`lib-common/uclibc/lib` 目录下的 `crt?.o` 会参与链接，它们用于提供与 GNU/Linux 相兼容的运行环境。编译过程会通过 `gcc` 定义宏 `LINUX_RT`，从而控制 `testcase/src/print-FLOAT.c` 采用 `printf()` 来输出结果。代码期望首先通过 `init_FLOAT_vfprintf()`（在 `FLOAT.a` 中定义）对 `libc.a` 进行劫持攻击，之后通过 `%f` 进行格式化时就会执行我们编写的劫持目标代码，该代码会根据 `FLOAT` 的编码方式进行格式化，从而实现通过 `%f` 对 `FLOAT` 变量进行格式化的功能。

运行 `obj/testcase/print-FLOAT-linux`，你会发现输出的结果为 `0.000000`，这是因为我们还没有对 `libc.a` 进行劫持，无法正确输出 `FLOAT` 类型的数据。你的第一个任务就是要对 `libc.a` 中负责进行字符串格式化的 `_vfprintf_internal()` 函数进行劫持。在劫持前，格式化 `%f` 时 `vfprintf_internal()` 会运行如下代码：

```

else if (ppfs->conv_num <= CONV_A) { /* floating point */
    ssize_t nf;
    nf = _fpmxtostr(stream,
        (__fpmx_t)
        (PRINT_INFO_FLAG_VAL(&(ppfs->info), is_long_double)
        ? *(long double *) *argptr
        : (long double) (* (double *) *argptr)),
        &ppfs->info, FP_OUT );
    if (nf < 0) {
        return -1;
    }
    *count += nf;

    return 0;
} else if (ppfs->conv_num <= CONV_S) { /* wide char or string */

```

其中会调用uclibc中的 `_fpmxtostr()` 函数来实现 `float` 类型数据的格式化, 其函数原型说明如下:

```

extern ssize_t _fpmxtostr(FILE * fp, __fpmx_t x, struct printf_info *info,
    __fp_outfunc_t fp_outfunc) attribute_hidden;

```

我们的目标是实现 `lib-common/FLOAT/FLOAT_vfprintf.c` 中的 `modify_vfprintf()` 函数, 它负责在运行时刻修改与上述 c 代码对应的二进制代码, 使其调用 `libcommon/FLOAT/FLOAT_vfprintf.c` 中的 `format_FLOAT()` 函数, 而不是 `_fpmxtostr()` 函数。要如何下手呢? 首先当然需要找到相应的二进制代码的位置。

选做任务: 知己知彼

对 `obj/testcase/print-FLOAT-linux` 进行反汇编, 找到与上述 c 代码对应的二进制代码的地址范围。

我们期望劫持后的代码如下:

```

else if (ppfs->conv_num <= CONV_A) { /* floating point */
    ssize_t nf;
    nf = format_FLOAT(stream, *(FLOAT *) *argptr);
    if (nf < 0) {
        return -1;
    }
    *count += nf;

    return 0;
} else if (ppfs->conv_num <= CONV_S) { /* wide char or string */

```

分析我们期望的代码, 我们只要做到以下几点即可:

- 将函数调用的目标改为 `format_FLOAT()`
- 设置好正确的函数调用参数
- 清理因为调用 `_fpmxtostr()` 而留下的浮点指令

首先我们来修改函数调用的目标。我们知道 `e8` 开头的是 `call REL32` 形式的指令, 因此后面跟的是一个相对于当前 `eip` 的偏移量。因此, 我们只需要修改这一偏移量, 就可以达到修改函数调用目标的目的了。为此, 我们需要得知这条 `call` 指令的地址。假设从反汇编结果中得知该指令的地址为 `p`, 那么 `p+1` 就指向了我们需要修改的偏移量。该如何修改这一偏移量呢? 其实我们只需要让它加上 `_fpmxtostr()` 的首地址和 `format_FLOAT()` 的首地址之间的差即可。由于 `format_FLOAT()` 就在 `lib-common/FLOAT/FLOAT_vfprintf.c` 中定义, 所以可以直接引用它来得到其首地址。但 `_fpmxtostr()` 却在 `libc.a` 中定义, 需要先通过外部引用来对其进行声明。事实上, 框架代码中已经对其进行声明了。

理解了上述内容之后, 你就可以在 `modify_vfprintf()` 中编写代码, 来修改函数调用的目标了。不过可别高兴太早了, 我们编写代码重新编译之后, 由于代码的大小发生了变化, 可能会导致我们修改的那条 `call` 指令在链接重定位阶段后的地址不再是之前从反汇编结果中看到的 `p`! 这下可麻烦了, 难道每次编写完代码之后, 都要重新回过头来调整 `p` 的值? 更麻烦的是, 如果链接的用户程序改变了, `p` 也同样会改变! 这意味着, 这种把 `p` 写死的方法来修改指令是不能拓展到其它用户程序中的。但天无绝人之路, 虽然这条 `call` 指令的位置会变化, 但它相对 `fprintf_internal()` 函数首地址的偏移量是不变的! 这样, 我们只需要先计算出这一偏移量 `o`, 我们就可以通过 `fprintf_internal + o` 的方式找到这条 `call` 指令了。由于 `fprintf_internal()` 函数的首地址是在链接重定位时确定的, 在运行时刻通过这种方式必定能找到 `call` 指令正确的位置。

选做任务: 偷龙转凤

在 `modify_vfprintf()` 中编写代码, 修改函数调用的目标, 使得 `fprintf_internal()` 在对 `%f` 进行格式化的时候调用 `format_FLOAT()`, 而不是 `_fpmxtostr()`。

编写代码后, 编译并运行 `obj/testcase/print-FLOAT-linux`, 你会发现程序触发了段错误! 这是因为在 GNU/Linux 下, 程序的代码是只读的, 而我们的劫持需要在运行时刻修改程序的代码, 进行了违规的写操作, 从而触发了段错误。为了使得我们的劫持可以顺利进行, 我们需要借助系统调用 `mprotect()` 的帮助。 `mprotect()` 可以改变一段内存区间的访问权限, 详细信息请参考 `man 2`

`mprotect`。我们需要在 `modify_vfprintf()` 修改代码之前为相应的内存区间设置可写权限：

```
#include <sys/mman.h>

mprotect((void *)((??? & 0xfffff000), 4096*2, PROT_READ | PROT_WRITE | PROT_EXEC);
```

其中，你需要填写 `???` 中的内容，内容为 `call` 指令所在地址的往前 (backward) `100` 字节所在的地址。往前 `100` 字节是因为我们接下来还需要修改这一区间的代码，而 `100` 字节已经足够覆盖这一区间。重新编译后运行，如果你的代码实现正确，你应该能看到目前通过 `%f` 输出了一个十六进制数。

但输出的这个十六进制数并不是我们在调用 `printf()` 时传入的 `FLOAT` 数据，这是因为我们仅仅改变了函数调用的目标，`_vfprintf_internal()` 仍然按照调用 `fpmxtostr()` 那样压入参数，但 `format_FLOAT()` 却以为 `vfprintf_internal()` 已经压入了正确的参数。

接下来我们需要修改传入参数的代码了。

选做任务：知己知彼（2）

阅读相应的汇编代码，并回答以下问题：

- 调用 `_fpmxtostr()` 的参数是如何压栈的？
- 变量 `argptr` 存放在哪一个寄存器中？
- 变量 `argptr` 指向的内容是什么？
- 变量 `stream` 的地址在哪里？
- 调用 `_fpmxtostr()` 返回后会清理栈上若干字节的内容，这若干字节都有些什么内容？

对比劫持前后的代码，我们发现第一个参数 `stream` 都是一样的，因此传入 `stream` 的代码不需要进行改动。为了让 `format_FLOAT()` 拿到正确的实参 `f`，我们需要在 `_vfprintf_internal()` 中将它的值放置到栈中正确的位置上。原来的代码中是通过一条占用三个字节浮点指令来放置第二个参数的，我们需要修改它，将它换成一条 `push` 指令，用于将 `argptr` 指向的内容压栈即可。正所谓细节决定成败，我们还需要处理以下的细节问题：

- 使用的 `push` 指令不能超过三个字节。
- 如果使用的 `push` 指令不足三个字节，剩下的字节要通过 `nop` 来覆盖，

保证原来那条三个字节的浮点指令"不留任何痕迹"。

- 由于使用的 `push` 指令改变了栈的深度, 为了正确维护栈的深度, 我们还需要修改前一条用于分配栈空间大小的指令。
- 由于 `format_FLOAT()` 只使用两个参数, 传入的其余参数可以暂时忽略。

选做任务: 偷龙转凤 (2)

在 `modify_vfprintf()` 中编写代码, 修改 `_vfprintf_internal()` 的代码, 使其压入正确的参数, 让 `format_FLOAT()` 正确输出 `FLOAT` 数据的十六进制表示。如果你仍然觉得毫无头绪, 你需要重新思考 **知己知彼(2)** 中的问题。

选做任务: 头龙转凤 (3)

现在的 `format_FLOAT()` 已经可以正确地获得 `FLOAT` 类型的实参了。你需要修改 `format_FLOAT()` 的实现, 把 `FLOAT` 类型数据的十进制小数表示作为格式化的结果。我们约定格式化结果通过截断的方式保留 6 位小数。

最后, 我们需要清除这段代码中的其它浮点指令, 来让 `print-FLOAT` 可以在 NEMU 中运行。清除的方式很简单, 只需要用 `nop` 指令覆盖它们即可。

另外在 NEMU 中运行之前, 记得去掉代码中的 `mprotect()` 系统调用, 因为目前 NEMU 还不能支持系统调用的执行。

选做任务: 偷龙转凤 (4)

在 NEMU 中运行 `print-FLOAT`, 你会发现仍然遇到浮点指令。仔细分析后, 你会发现这次遇到的浮点指令位于 `_ppfs_setargs()` 中。你的最后一个任务就是编写 `libcommon/FLOAT/FLOAT_vfprintf.c` 中的 `modify_ppfs_setargs()` 函数, 让其对 `_ppfs_setargs()` 函数的运行时二进制进行修改, 达到绕过上述浮点指令的目的。

主要的相关代码是一个 `switch-case` 语句, 通过不同的格式说明符来获取不同长度的数据。`%f` 对应的是一个 64 位的数据(float 类型不是 32 位吗?), 事实上也可以把它看成一个 `long long` 类型的数据来获取, 这样就能绕过 `float` 相应分支的浮点指令来获取 64 位的数据了。你的修改目标就是在 `float` 分支的开头放置一条 `jmp` 指令, 跳转到 `long long` 分支。更多代码细节请阅读 `modify_ppfs_setargs()` 函数中的注释。有了修改 `_vfprintf_internal()` 函数的经验, 这次的任务当然也难不倒聪明的你啦!

实现正确后, 你就可以在 NEMU 中成功运行用户程序 `print-FLOAT` 了。

不过在真实的操作系统中, 要进行这种劫持代码的攻击其实并非易事。回顾我们在 GNU/Linux 的实现, 我们需要先通过 `mprotect()` 系统调用打开代码的可写权限, 才能进行劫持。而一般的程序并不会主动打开代码的可写权限, 因此黑客们也在绞尽脑汁思考其它更巧妙的攻击方式。不过可以肯定的是, 没有对计算机系统的深入理解, 是不可能写出这样的代码的。

和代码玩游戏

在各门计算机专业课上, 你学会了如何对数据做一些基本操作, 学会了 `play with the data`, 例如算阶乘, 用链表组织学生信息, 但你从来都不知道代码是什么东西。而上述练习其实就是在 `play with the code`, 你会发现修改程序的行为犹如探囊取物。和代码玩游戏的最高境界就要数 `self-modifying code` 了, 它们能够在运行时刻修改自己, 但这种代码极难读懂, 维护更是难上加难, 因此它们成为了黑客们炫耀的一种工具。事实上, 你刚才已经编写了 `self-modifying code`!

这一切是否引起你的思考: 代码的本质是什么? 代码和数据之间的区别究竟在哪里?

温馨提示

PA2 到此结束。

任务自查表

序号	是否已完成
必做任务 1	
必做任务 2	
必做任务 3	
必做任务 4	
必做任务 5	
选做任务 1	
选做任务 2	

实验提交说明

提交地址

阶段性工程提交至自建 git 远程仓库

最终工程和实验报告提交智慧树平台

提交方式

1. 实验报告命名为“学号-PA2.pdf”，上传至智慧树平台；
2. 教师将通过脚本自动检查运行结果，因此除非实验手册特别说明，不要修改工程中的任何脚本，包括 Makefile、test.sh。
 - 我们会清除中间结果，使用原来的编译选项重新编译(包括 -Wall 和 -Werror)，若编译不通过，本次实验你将得 0 分(编译错误是最容易排除的错误，我们有理由认为你没有认真对待实验)。
3. 学生务必使用 `make submit` 命令将整个工程打包，导出到本地机器，再上传到智慧树平台。
 - `make submit` 命令会用你的学号来命名压缩包，然后修改压缩包的名称为“学号-PA2.zip”。此外，不要修改压缩包内工程根目录的命名。为了防止出现编码问题，压缩包中所有文件名都不要包含中文。

git 版本控制

请使用 git 管理你的项目，并合理地手动提交的 git 记录。请你不定期查看自己的 git log，检查是否与自己的开发过程相符。git log 是独立完成实验的最有力证据，完成了实验内容却缺少合理的 git log，会给抄袭判定提供最有力的证据。

如果出现 git head 损毁现象，说明你拷贝了他人的代码，也按抄袭处理。

实验报告内容

你必须在实验报告中描述以下内容：

- **实验进度。**按照“任务自查表”格式在线制作表格并填写。缺少实验进度的描述，或者描述与实际情况不符，将被视为没有完成本次实验。

● 思考题

你可以自由选择报告的其它内容。你不必详细地描述实验过程, 但我们鼓励你在报告中描述如下内容:

- 你遇到的问题和对这些问题的思考、解决办法
- 实验心得

如果你实在没有想法, 你可以提交一份不包含任何想法的报告, 我们不会强求。但请不要

- 大量粘贴讲义内容
- 大量粘贴代码和贴图, 却没有相应的详细解释(让我们明显看出是凑字数的)

来让你的报告看起来十分丰富, 编写和阅读这样的报告毫无任何意义, 你也不会因此获得更多的分数, 同时还可能带来扣分的可能。