实验报告-PA1

甘晨 181240014

2019 年 9 月 29 日

1 实验进程

1.1 PA1.1

ISA 选择了 x86，利用 Union 和 Struct 重新组织了寄存器的结构。实现了单步执行、打印寄存器和扫描内存功能，艰难地尝试着理解框架代码。**完成了 PA1.1 所有内容**。

1.2 PA1.2

实现了算数表达式的词法分析、匹配括号、寻找主操作符和递归求值功能，但没有是实现负数的运算。最后，实现了生成表达式工具，并借助此工具测试了表达式求值的功能。**完成了 PA1.2 所有内容**。

1.3 PA1.3

实现了拓展表达式求值的功能，能够实现指针解引用，取寄存器值和“&&”、“==”、“!=”运算功能，其余一些运算符在后面有需要再实现。**完成了 PA1.3 所有功能的实现，但只进行了少量测试，对于把 make count 写入 Makefile 文件这一点还未实现**

2 必答题

2.1 PA1.1

实现寄存器结构调整:

首先，通过 RTFSC 和阅读讲义中的 x86 寄存器组织结构，可以发现 32 位、16 位和 8 位寄存器需要共用地址空间，在提示的帮助下，这里可以使用 union 结构来重新组织这些寄存器，而使用匿名 union 的好处是方便直接访问联合类型的成员。然而，运行

make run 还会 abort，报错提示为（似乎是，记不太清了）：Assertion ”sample[R_EAX] == cpu.eax” failed”

通过 RTFSC，我发现在程序运行过程中没有给 cpu.eax 赋值的过程，所以说这个 assert（）会被触发。那么如何给这些寄存器赋值呢？这里卡了我很久，以至于想换成 riscv32，在做了各种尝试，甚至尝试在结构体定义的过程中直接赋值的操作（暴露了程序设计基础学的不扎实）. 后来在，在大佬的提示下，了解到了可以再用一个联合类型，让上面的 _32、_16 和 _8 寄存器和下面的 eax、edx 等寄存器共用空间，这样就可以实现对寄存器的赋值了，但是这边还需要注意的时，需要把 eax，edx 等通用寄存器整合为一个匿名 struct 结构，防止这些通用寄存器共用空间。

实现单步执行，打印寄存器，扫描内存:

```
nector@debian:~/ics2019/nemu$ make ISA=x86 run
Building x86-nemu
+ CC src/monitor/debug/watchpoint.c
+ LD build/x86-nemu
make -C /home/nector/ics2019/nemu/tools/qemu-diff
make[1]: Entering directory '/home/nector/ics2019/nemu/tools/qemu-diff'
make[1]: Nothing to be done for 'app'.
make[1]: Leaving directory '/home/nector/ics2019/nemu/tools/qemu-diff'
./build/x86-nemu -l ./build/nemu-log.txt -d /home/nector/ics2019/nemu/tools/qemu-diff/build/x86-qemu-so
[src/monitor/monitor.c,36,load_img] No image is given. Use the default build-in image.
[src/memory/memory.c,16,register_pmem] Add 'pmem' at [0x00000000, 0x07ffffff]
[src/device/io/mmio.c,14,add_mmio_map] Add mmio map 'argsrom' at [0xa2000000, 0xa2000fff]
[src/monitor/monitor.c,20,welcome] Debug: ON
[src/monitor/monitor.c,23,welcome] If debug mode is on, A log file will be generated to record every instruction
 NEMU executes. This may lead to a large log file. If it is not necessary, you can turn it off in include/common
.h.
[src/monitor/monitor.c,28,welcome] Build time: 21:09:23, Sep 28 2019
Welcome to x86-NEMU!
For help, type "help"
(nemu) si
  100000:   b8 34 12 00 00                      movl $0x1234,%eax
(nemu) si 5
  100005:   b9 27 00 10 00                      movl $0x100027,%ecx
  10000a:   89 01                               movl %eax,(%ecx)
  10000c:   66 c7 41 04 01 00                   movw $0x1,0x4(%ecx)
  100012:   bb 02 00 00 00                      movl $0x2,%ebx
  100017:   66 c7 84 99 00 e0 ff ff 01 00       movw $0x1,-0x2000(%ecx,%ebx,4)
(nemu) si 5
  100021:   b8 00 00 00 00                      movl $0x0,%eax
  100026:   d6                                  nemu trap
nemu: HIT GOOD TRAP at pc = 0x00100026

[src/monitor/cpu-exec.c,28,monitor_statistic] total guest instructions = 8
(nemu)
```

图 1: 单步执行

```
(nemu) info r
Register_id    Hexadecimal       Decimal
    eax:       0x00000000      000000000000D
    ecx:       0x00100027      000001048615D
    edx:       0x2db590b7      000766873783D
    ebx:       0x00000002      000000000002D
    esp:       0x7b984bd6      002073578454D
    ebp:       0x6d8c6756      001837918038D
    esi:       0x200cfd4e      000537722190D
    edi:       0x43ba55ca      001136285130D
(nemu)
```

图 2: 打印寄存器

```
(nemu) x 10 0x100000
0x00100000:      184           0x000000b8
0x00100001:      52            0x00000034
0x00100002:      18            0x00000012
0x00100003:      0             0x00000000
0x00100004:      0             0x00000000
0x00100005:      185           0x000000b9
0x00100006:      39            0x00000027
0x00100007:      0             0x00000000
0x00100008:      16            0x00000010
0x00100009:      0             0x00000000
(nemu)
```

图 3: 扫描内存

2.2 PA1.2

词法分析:

```
{"\\$[a-zA-Z]{2,3}",TK_REG},        // register
{" +", TK_NOTYPE},                  // spaces
{"\\+", '+'},                       // plus
{"==", TK_EQ},                      // equal
{"!=", TK_UEQ},                                 // unequal
{"\\-", '-'},                       // minus
{"\\*", '*' },                      // multiply
{"/", '/'},                         // divide
{"\\(", '('},                       // left_bracket
{"\\)", ')'},                       // right_bracket
{"0x[0-9a-fA-F]+",TK_HEX},   // hex_number
{"[0-9]+",TK_NUM},                  // number
{"[u]{1}",TK_CHAR},                 // character
{"&&", TK_AND}                     // and
```

图 4: 匹配规则

```
Welcome to x86-NEMU!
For help, type "help"
(nemu) p (*0x100000+$edx-12*(9-1)== 0)&&1
[src/monitor/debug/expr.c,148,make_token] match rules[8] = "\(" a
t position 0 with len 1: (

[src/monitor/debug/expr.c,148,make_token] match rules[6] = "\*" a
t position 1 with len 1: *

[src/monitor/debug/expr.c,148,make_token] match rules[10] = "0x[0
-9a-fA-F]+" at position 2 with len 8: 0x100000

[src/monitor/debug/expr.c,148,make_token] match rules[2] = "\+" a
t position 10 with len 1: +

[src/monitor/debug/expr.c,148,make_token] match rules[0] = "\$[a-
zA-Z]{2,3}" at position 11 with len 4: $edx
```

图 5: 识别信息

递归求值和生成表达式检测:

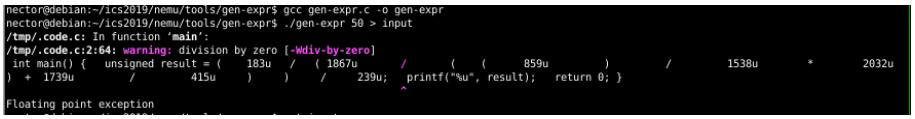

图 6: 生成表达式除 0 报错

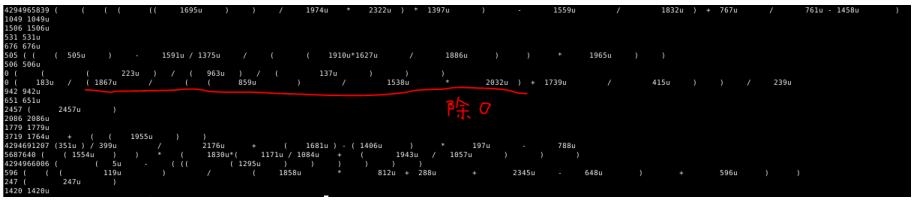

图 7: 生成的表达式

这里在生成表达是的时候可能会遇到除 0 的情况，此时编译器会报错，同时，我也在 eval () 函数里 assert 了除数为 0 的情况，但由于 assert 报错会影响测试继续执行，但往往在 abort 之前已经进行了相当多的样例测试了，此处没有去进一步处理。为了确保样例给的结果是无符号运算结果，我在数字后面都加了 u。

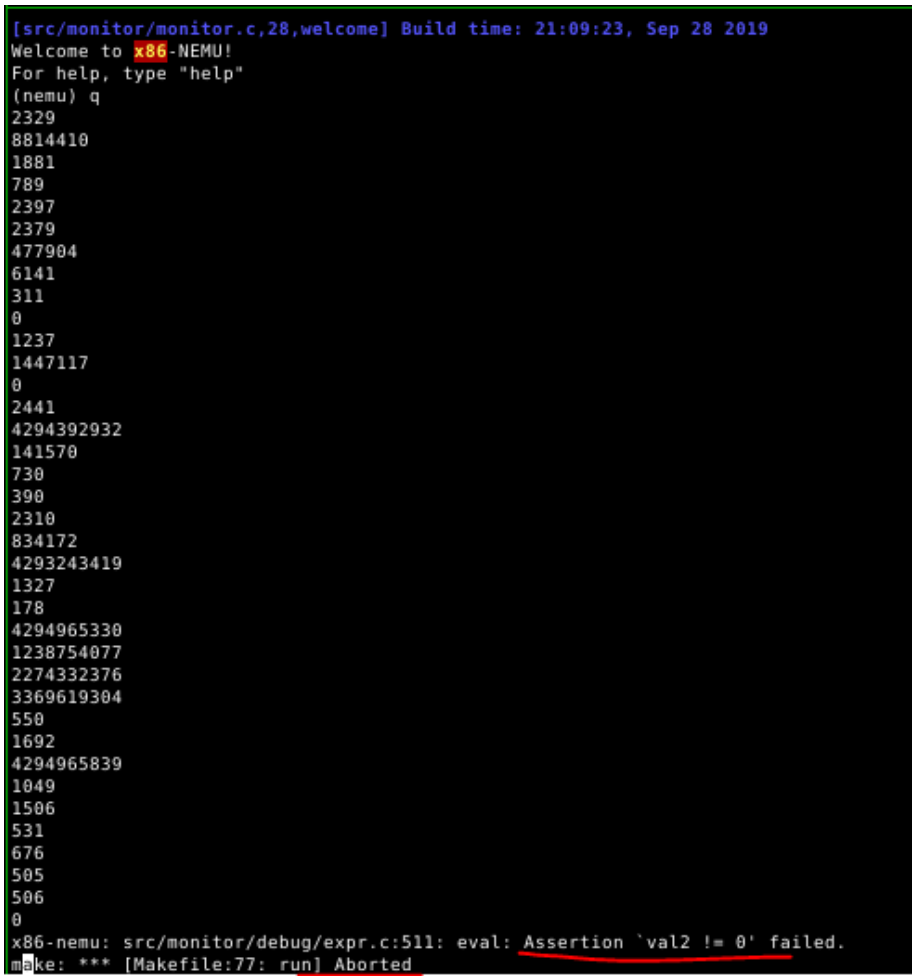

图 8: 表达式求值结果

此处是在简易调试器中运行计算生成的表达式，遇到除数是 0，触发了 assert，从而程序 abort

2.3 PA1.3

表达式求值功能拓展:

```
Welcome to x86-NEMU!
For help, type "help"
(nemu) p *0x100000 == 184
1
(nemu) p $edx+$ecx
2006231766
(nemu) p 1&&0
0
(nemu) p (12*12==144)&&(54-52==0)
0
(nemu) p (12*12==144)&&(54-52==2)
1
(nemu) p 0xff-255==0
1
(nemu)
```

图 9: 表达式求值功能拓展

监视点实现:

```
typedef struct watchpoint {
  int NO;
  bool work_state;
  char expression[32];
  uint32_t Old_Value;
  uint32_t New_Value;
  struct watchpoint *next;
```

图 10: 监视点结构

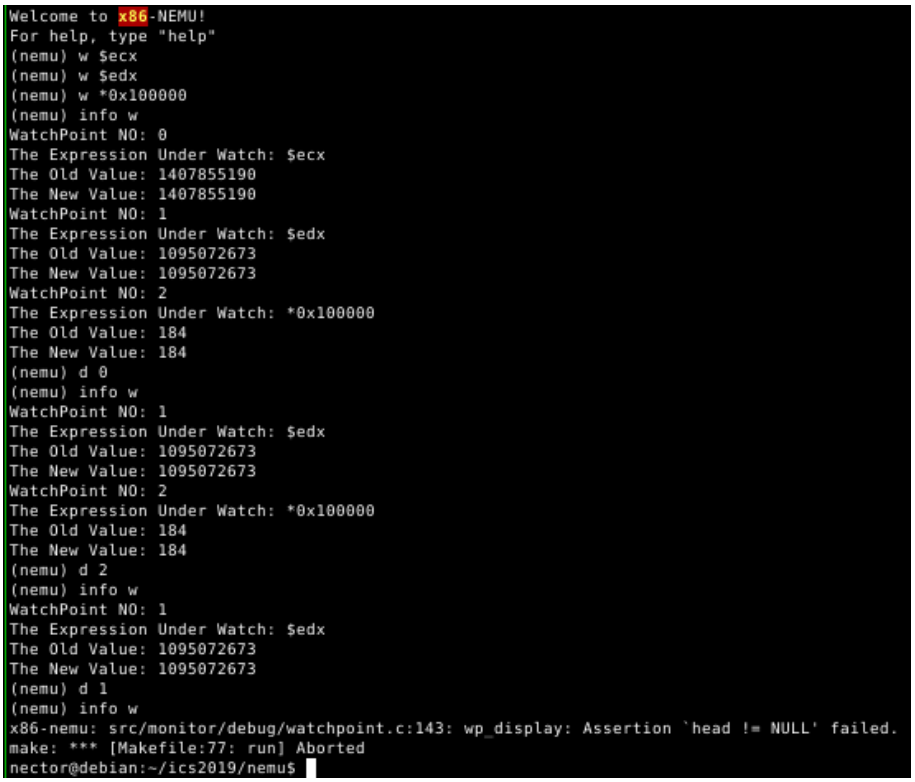

图 11: 监视点管理

目录定位:

Selector 的概念位于 i386 手册的 Chapter5 Memory Management 的 5.1Segment Translation 的 5.1.3Selectors

送分题:

我选择的 ISA 是 x86

理解基础设施:

基于讲义中的假设，450 次调试需要花费 9.375 天从 GDB 中获取并分析信息, 另一方面，如果市县实现了调试器，450 次调试需要花费 3.125 天从简易调试器中获取并分析信，也就是说简易调试器可以节约 2/3 的时间。

查阅手册:

CF 位首先表示 CARRY FLAG，是一种状态标志（Status Flag），这一标志受算术指令的影响，在执行算术指令之前，会改变 CF 的值；根据 Appendix C 中的描述，CF 指示高位的借位或进位，有借位或进位为 1（set），否则为 0（clear）。

ModR/M 会出现在操作码（opcode）后面，来确定获得操作数的寻址方式（specify the addressing form to be used)——内存或寄存器. 对于取内存操作数,还通过 ModR/M

来确定地址的计算方式。

根据 i386 手册，这里的 MOV 指令时 Intel 格式，不同于 AT&T 格式，其中 MOV A B 中，B 表示源操作数，A 表示目的操作数，与 AT&T 格式恰好相反。其中，源操作数可以使 8/16/32 位寄存器或内存、立即数、段基址和偏移地址做代表的内存中的内容，目的操作数也可以来自上述内容，但不能是立即数。

Shell 指令:

共有 6304 行代码，使用的命令是:（在 nemu/目录下）”find . -name “*.c”|xargs cat |wc -l” 和” find . -name ”*.h”|xargs cat |wc -l “，最后调用之前实现的 p 指令做一下表达式求值即可。

除去空行，共有 5143 行代码，使用的命令是:（在 nemu/目录下）”find . -name “*.c”|xargs cat|grep -v $|wc -l” 和” find . -name ”*.h”|xargs cat|grep -v $|wc -l “.

在 PA1 开始之前，共有 4970 行代码，除去空格有 4007 行代码，所以说，在 PA1 中，我一共写了 1136 行代码

使用 man:

-Wall 和-Werror 的作用是在编译程序时显示所有警告，并且把 warning 也当做 error 显示出来，正如 PA 讲义中所说的:” 调试是从 failure 回溯 fault 的过程 “，因而尽早的观测到 failure 有可能节省调试的时间，所以说，把 warning 出来的可能出错的地方在编译时警告显示出来，有可能避免当发现 failure 时已经很难在回溯的 fault 的情况发生。

3 实验心得

3.1 PA1.1

PA1.1 中给我留下最深的印象的就是 strtok() 函数，我在初次实现几个命令的功能时，是根据 cmd_help 指令依葫芦画瓢的，help 指令的第一步时提取第一个字符，因而我在后面的操作中学着提取了第一个字符，然而，我对传入参数 args 的理解不清，以为时包括命令标示（如 p、x、info 等等）的字符数组，因而，我谢了 char *arg = strtok(NULL,” ”) 这样一串代码，后来发现这行代码的意义不大，只是起到了分割字符数组的作用，如果指令字符后面只有一个参数，那么没必要执行这么一句，但对于 info 和 x 指令，就是必要的。但就是这一行代码，给我在 PA1.3 时带来了很多痛苦，以及消耗了我很多时间来找 bug，后面将会详述。

3.2 PA1.2

PA1.2 中的找主操作符是这一过程给我带来了体验到了一个道理：想偷懒反而可能会需要做的更多，一开始 PA 讲义上说可以把整个 tokens 数组遍历一遍来找，一开始，我想过可以给每个操作符一个优先级，方便比较，可想来还要把 tokens 数组遍历一遍，并且还要新定义一个结构体，有点麻烦，所以想另寻他法。最终，我想到了一个” 好办法

“, 我用递归的方式，先从 tokens 右边往左找，找到不在括号内的第一个运算符，如果是 + 或-，那么就返回这个运算符的 index，如果是 * 或/，那么再从左往右找不在括号里的第一个运算符；如果是 + 或-，那么返回包含左边第一个运算符但不包括右边第一个运算符的数组再去递归查找；如果是 * 或/，那么先记下右边那个运算符的 index，在递归查找包括右边运算符但不包括左边运算符的数组，若返回的 index 变了，那么就去改变后的 index，若不变，那就取那个 index；查找最后都会有 i == j, 若 i>j, 则 assert(0). 在 PA1.2 中，这一方法成功了，而且也通过了生成表达式的样例测试，也就是这成功的方法，使我在面对 PA1.3 时感觉非常糟糕。

3.3 PA1.3

好吧，似乎 PA1.1、PA1.2 都给 PA1.3 挖了坑，对于 PA1.2，我发现我找主操作符的方法在 PA1.3 中根本没法拓展，加了几个运算符之后，这个方法就很难行的通了，没办法了。在还是那位大佬（就是教我调整寄存器结构的那位大佬）的指点下，我还是利用了拷贝 tokens，并在结构体内加优先级的方法来做，出乎意料的是，这个方法竟然异常简单，我本来将近百行的代码瞬间缩短为了小几十行，我本来想偷懒的想法反而让我花费了更多时间。到 PA1.1 了，唉，没脸说了，我花了一个完晚上和周六一整个白天都在想这个 bug 是怎么回事。当时情况是这样的，我在拓展表达式求值的功能，因此，需要匹配寄存器，也就是需要匹配 $ 字符，我先是根据我的想法，写了正则表达式，我用 $edx 去测试，可报错为” position 0 no match “，而且这个 position 0 指向的是 “e”，当时我并没有注意到这个问题，又试了几种，还是不行，后来，我又上网查询，向大佬（还是那位大佬）求教，然而按照大佬的匹配方式，居然还是报错了！我都惊呆了！以为是虚拟机与真机有差别而产生了 bug!（似乎有点好笑），没办法，我后来只好先不实现寄存器，接着往下走，于是我开始搞其他的。偶然间，我打开了 ui.c 文件，就是写指令的文件，还记得我之前说的对 strtok 理解不清吗，我在取 p 指令的表达式的时候，也多次一举的又 strtok 了一下，我想我需要把后面的表达式与前面的操作符表示分开，而且后面的表达式需要时一整块（我当时还不知道传进来的字符数组已经去掉指令标志了），所以我不能按空格分割，而需要按一个不可能出现的字符来分割，你猜我选择了那个字符（QAQ），教训相当惨痛!!!

4 鸣谢

由衷感谢那位 x 姓大佬在我在 PA1 的泥潭中奄奄一息之时给予的帮助和指点，在此表示深深感谢。