



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

系统综合课程设计实验报告

PA1 实验报告

周辰霏 1712991

年级：2017 级

专业：计算机科学与技术

2020 年 3 月 12 日

目录

一、 概述	1
(一) 实验目的	1
(二) 实验内容	1
二、 阶段一	1
(一) 实现 x86 的寄存器结构体	1
(二) 基础设施实现	2
1. 单步执行	2
2. 打印寄存器	3
3. 扫描内存	4
三、 阶段二	5
(一) 表达式词法分析	5
(二) 递归求值	6
1. 括号匹配	6
2. 主运算符	7
3. 负数	9
4. 随机测试	10
四、 阶段三	12
(一) 扩展表达式功能	12
(二) 实现监视点	13
五、 遇到的 bug 及解决	17
六、 思考题 && 必答题	18
(一) 思考题	18
(二) 必答题	20
七、 总结	21

一、 概述

(一) 实验目的

- 熟悉 GNU/Linux 平台
- 初步探究“程序在计算机上运行”的相关原理
- 初步学习 GDB 并在 PA 上实现简易调试器
- 学会自行阅读手册以及 RTFSC

PA 系列实验的终极目标是, 通过构建一个简单完整的计算机系统深入理解程序如何在计算机上运行。同时 PA 作为一个让我们更深一层了解计算机系统结构的实验, 它要教会我们在实验中自己思考并通过自己动手解决一系列的问题 (RTFM、STFW、RTFSC), 在遇到各式各样的 bug 和 error 的同时去尝试理解并掌握计算机系统的每一处细节, 并最终一步步完成 PA。在进行实验的过程中, 学会并养成独立思考并解决问题的习惯, 并最终理解每一处细节的设定, 通过在 PA 中编写一个用来执行其它程序的程序, 从而比较好的辅助课程掌握计算机系统。

PA1 作为开天辟地的新篇章, 其实只需要实现逻辑上的简单图灵机, 即从 EIP 指向位置取指并执行, 之后更新 EIP, 再对这一过程进行循环往复。而 GDB 作为一个优秀的调试器, 我们需要仿照 GDB 对 NEMU 实现简易调试器作为其基础设施, 以便 NEMU 可以随时了解客户程序执行的所有信息 (这是 GDB 不容易获悉的)。为了提高调试的效率, 同时也作为熟悉框架代码的练习, 我们需要在 monitor 中实现一个具有几项基本功能的简易调试器。

(二) 实验内容

PA1 真正作为代码的实验部分主要是简易调试器的实现, 可大致分为三个部分:

1. 模拟寄存器结构, 实现简易调试器基本功能: 单步执行、打印寄存器状态、扫描内存
2. 实现简易调试器调试功能中不可或缺的表达式求值, 并完善内存扫描函数
3. 实现简易调试器调试功能中的监视点, 学习断点相关知识与 i386 手册

二、 阶段一

(一) 实现 x86 的寄存器结构体

早先在 PA0 中运行 NEMU 会出现 assertion fail 的错误信息, 这是因为框架代码并没有正确地实现用于模拟 x86 寄存器的结构体 `CPU_state`, 所以正确实现改模拟寄存器的结构体即可 (`reg.h, reg.c`) 根据指导书的 x86 寄存器结构示意图, 其中 32 位寄存器 8 个: 4 个数据寄存器 EAX, EDX, ECX, EBX, 2 个变址和指针寄存器 ESI, EDI, 两个指针寄存器 ESP, EBP; 8 个为 32 位寄存器低 16 位的 16 位寄存器: AX, DX, CX, BX, BP, SI, DI, SP; 以及 16 位寄存器高 8 位低 8 位组成的 8 个 8 位寄存器: AL, DL, CL, BL, AH, DH, CH, BH。

x86 寄存器结构体

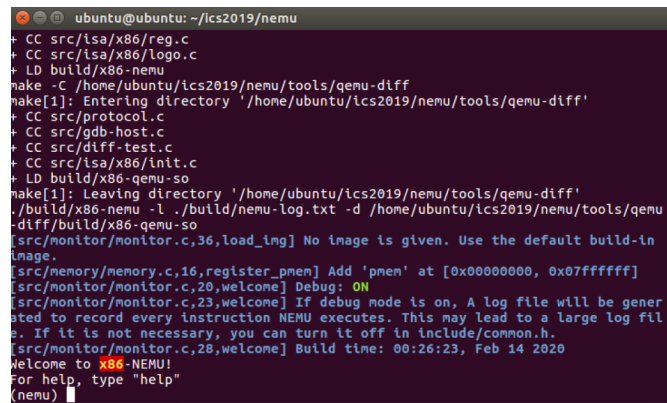
```
1 //nemu/src/isa/x86/include/isa/reg.h
2 typedef union { //struct -> union
3 union { //struct -> union
4     uint32_t _32;
```

```

5         uint16_t _16;
6         uint8_t _8[2];
7     } gpr[8];
8     struct {          // 对应CPU_state结构体
9         rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
10        vaddr_t pc;
11    };
12    } CPU_state;
13    extern CPU_state cpu; // 为了与后续调用相匹配

```

在经过上述实现后, 我们进行 make run 可以看到 NEMU 已经可以在 x86isa 下正常启动了。



```

ubuntu@ubuntu: ~/ics2019/nemu
+ CC src/isa/x86/reg.c
+ CC src/isa/x86/logo.c
+ LD build/x86-nemu
+ CC /home/ubuntu/ics2019/nemu/tools/qemu-diff
make[1]: Entering directory '/home/ubuntu/ics2019/nemu/tools/qemu-diff'
+ CC src/protocol.c
+ CC src/gdb-host.c
+ CC src/diff-test.c
+ CC src/isa/x86/init.c
+ LD build/x86-qemu-so
make[1]: Leaving directory '/home/ubuntu/ics2019/nemu/tools/qemu-diff'
./build/x86-nemu -l ./build/nemu-log.txt -d /home/ubuntu/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_pmen] Add 'pmem' at [0x00000000, 0x07ffffff]
[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: 00:26:23, Feb 14 2020
Welcome to x86-NEMU!
For help, type "help"
(nemu)

```

图 1: x86 寄存器

(二) 基础设施实现

经过了对 x86isa 的寄存器结构实现, NEMU 已经可以正常运行, 但为了实现提升开发效率的基础设施, 我们需要从简单的几个入手并熟悉代码框架: 单步执行、打印寄存器状态、扫描内存。

所需要实现的 monitor 调试功能均在 nemu/src/monitor/debug/ui.c 中, 我们需要去做的仅仅只是把已有框架下的既定功能实现并完善。

ui_mainloop 会通过 readline() 获取控制台的键入命令并使用 strtok(根据第二个参数将第一个参数传入的字符串进行分解并返回第一个子字符串) 识别和解析命令, 由于简易调试器中的命令如果跟有参数会与前面的命令指示符以空格划分, 所以可以使用 strtok(str, " ") 来获取命令字符串, 并将其与 cmd_table 中的内容相对比, 若有可以匹配的已实现命令则调用命令处理函数, 否则报错。

1. 单步执行

单步执行的实现非常简单, 因为框架中已经包含了一个 cmd_c 的继续执行函数, 我们只要仿照它的实现进行实现即可。即使用 cpu_exec() 函数, 该函数是模拟 CPU, 参数表示执行几条指令。同时还要在 cmd_table 里面加入相应描述字段和函数 handler。实验指导书的 sscanf 函数很好用, 它是将第一个参数格式化为第二个参数的形式, 并以引用的格式将值传递给第三个参数。

单步执行

```

1 // nemu/src/monitor/debug/ui.c
2 static int cmd_si(char *args){
3     int step=0; // 单步执行的步数

```

```

4      /* extract the first argument */
5      char *arg = strtok(NULL, " "); // 获取N
6      if (arg==NULL) // si 等价 si 1
7          cpu_exec(1);
8      else {
9          sscanf(arg, "%d", &step); // atoi
10         if (step <=0)
11             printf("Invalid number!\n");
12         else
13             cpu_exec(step); // 不要使用 for 循环
14     }
15     return 0;
16 }

```

```

Welcome to x86-EMUUI
For help, type "help"
(nemu) si 4
100000: b8 34 12 00 00      movl $0x1234,%eax
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)
(nemu) si
100012: bb 02 00 00 00      movl $0x2,%ebx
(nemu) si 0
nemu: HIT GOOD TRAP at pc = 0x00100026

```

图 2: 单步执行 si

2. 打印寄存器

ics2019 因为引入了三种不同的 ISA 所以这一项要在选定的 ISA 中实现为 API, 这样就可以屏蔽 ISA 之间的差异。借助前面构建好的寄存器结构循环输出其中的内容即可。

打印寄存器

```

1      //nemu/src/monitor/debug/ui.c
2      static int cmd_info(char *args){
3          /* extract the first argument */
4          char *arg = strtok(NULL, " "); // 判断 r 还是 w
5          if (strcmp(arg, "r")==0) // info r
6              isa_reg_display(); // 调用 reg.c 中的 API
7          else if (strcmp(arg, "w")==0){
8              printf("PA1.1 haven't implement this function yet.");
9          }
10         else // 没有键入 r 或 w 参数
11             printf("Unknown command:%s\n", arg);
12         return 0;
13     }
14     //nemu/src/isa/x86/reg.c
15     void isa_reg_display() {
16         for(int i=0; i<8; i++) // 8 个 32-bit reg
17             printf("%s: \t0x%08x\t\n", regsl[i], cpu.gpr[i]._32);
18         for(int i=0; i<8; i++) // 8 个 16-bit reg
19             printf("%s: \t\t\t0x%04x\t\n", regsw[i], cpu.gpr[i]._16);
20         for(int i=0; i<8; i++) // 8 个 8-bit reg

```

```

21         printf( "%s:\t\t\t\t0x%02x\t\n", regs_b[ i ], cpu.gpr[ i % 4 ]._8[ i / 4 ] );
22         printf( "eip:\t\t\t\t0x%08x\t\n", cpu.pc ); // eip
23     }

```

```
For help, type "help"
(nemu) info r
eax: 0x08cbc475
ecx: 0x352f8ad0
edx: 0x57c52924
ebx: 0x12ea88ea
esp: 0x52496588
ebp: 0x5d742cd8
esi: 0x2da64944
edi: 0x00088b4
ax: 0xc475
cx: 0x8ad0
dx: 0x2924
bx: 0x88ea
sp: 0x6588
bp: 0x2cd8
sl: 0x4944
di: 0x88b4
al: 0x75
cl: 0xd0
dl: 0x24
bl: 0xea
ah: 0xc4
ch: 0x8a
dh: 0x29
bh: 0x88
pc: 0x00100000
(nemu)
```

图 3: 打印寄存器 info r

3. 扫描内存

阶段一的扫描内存相对简单,只需要对 16 进制地址向后相应扫描就可以,函数主要是利用了 `vaddr_read` 来读指定地址的内容,阶段一只需要将地址传入并把长度调为 4 字节就可以了。虽然这个函数写的是 `vaddr` 线性地址,但是实际上它在 `memory.h` 被定义为 `isa_vaddr_read`,即根据不同 ISA 调用不同的读取 API,但是找到 `nemu/src/isa/x86/mmu.c` 中的定义,实际返回的还是 `paddr_read` 的结果,即物理地址的结果。

扫描内存

```

1 //nemu/src/monitor/debug/ui.c
2 static int cmd_x(char *args){
3     /*PA1.1 Simple version EXPR must be a hex number,
4     such as x 10 0x100000*/
5     //use memory/memory.c paddr_read which needs addr and len
6     /* extract the first argument */
7     int N, start;
8     vaddr_t temp;
9     char *arg0 = strtok(NULL, " "); //get N,4N
10    char *arg1 = strtok(NULL, " "); //get EXPR ,where to start
11    if(arg0==NULL||arg1==NULL){
12        printf("Lack of parameter!\n"); return 0;}
13    if(N==0){printf("Invalid parameter!\n"); return 0;}
14    sscanf(arg0, "%d",&N);
15    if(arg1[0]!='0'){ //not 0x hex address
16        //PA1.2
17    }
18    else{

```

```

19         sscanf(arg1, "%x", &start);
20         printf("address\tdata");
21         for (int i=0; i<N; i++){
22             if (i%4==0)
23                 printf("\n0x%x:\t0x%02x\t", start+i, vaddr_read(start+i, 1));
24             else
25                 printf("0x%02x\t", vaddr_read(start+i, 1));
26         }
27     }
28     printf("\n");
29     return 0;
30 }

```

NEMU 运行结果如下, 与 src/isa/x86init.c 中的默认镜像相对比一致说明功能实现正确。

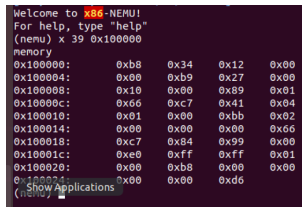


图 4: 扫描内存 1.0

```

const uint8_t isa_default_img[] = {
    0xb8, 0x34, 0x12, 0x00, 0x00, // 100000: movl $0x1234,%eax
    0xb9, 0x27, 0x00, 0x10, 0x00, // 100005: movl $0x100027,%ecx
    0x89, 0x01, // 10000a: movl %eax,%ecx
    0x66, 0xc7, 0x41, 0x04, 0x01, 0x00, // 10000c: movw $0x1,0x4(%ecx)
    0xbb, 0x02, 0x00, 0x00, 0x00, // 100012: movl $0x2,%ebx
    0x66, 0xc7, 0x84, 0x99, 0x00, 0xe0, // 100017: movw $0x1,-0x2000(%ecx,%ebx,4)
    0xff, 0xff, 0x01, 0x00,
    0xb8, 0x00, 0x00, 0x00, 0x00, // 100021: movl $0x0,%eax
    0xd6, // 100026: nemu_trap
};

```

图 5: 与默认镜像对比

三、 阶段二

(一) 表达式词法分析

再进行表达式运算之前, 首先需要识别出表达式中的 token 单元, 即编译原理中的词法分析阶段。在表达式计算的这一部分只需要实现基本四则运算并忽略空格即可。

词法分析

```

1 // nemu/src/monitor/debug/expr.c
2 enum {TK_NOTYPE = 256, TK_EQ, TK_NEQ, TK_REG, TK_DEC, TK_HEX, TK_NEG,
3       TK_AND, TK_OR, TK_LS, TK_RS, TK_LE, TK_GE, TK_POI};
4 // rules
5 {" +", TK_NOTYPE}, // spaces
6 {"0x[0-9A-F][0-9A-F]*", TK_HEX}, // hex
7 {"[0-9][1-9][0-9]*", TK_DEC}, // dec
8 {"\\+", '+'}, // plus
9 {"\\-", '-'}, // minus
10 {"\\*", '*'}, // multiple
11 {"\\/", '/'}, // divide
12 {"\\(", '('}, // LB
13 {"\\)", ')'}, // RB
14 // 之后在 make_tokens 中把 token 存到 tokens 数组中
15 if (rules[i].token_type==TK_NOTYPE) continue; // 忽略空格
16 memset(tokens[nr_token].str, '\\0', 32); // 避免溢出

```

```

17     strncpy(tokens[nr_token].str, substr_start, substr_len);
18     tokens[nr_token].type = rules[i].token_type;
19     nr_token++;
20     break;

```

第一步词法分析结束后可以看到 NEMU 已经可以正常输出词法分析的结果了

```

(nemu) p 3-8/4*(2+15)
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]*" a
t position 0 with len 1: 3
[src/monitor/debug/expr.c,94,make_token] match rules[5] = "-" at position 1 wit
h len 1: -
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]*" a
t position 2 with len 1: 8
[src/monitor/debug/expr.c,94,make_token] match rules[7] = "/" at position 3 wi
th len 1: /
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]*" a
t position 4 with len 1: 4
[src/monitor/debug/expr.c,94,make_token] match rules[6] = "*" at position 5 wi
th len 1: *
[src/monitor/debug/expr.c,94,make_token] match rules[8] = "(" at position 6 wi
th len 1: (
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]*" a
t position 7 with len 1: 2
[src/monitor/debug/expr.c,94,make_token] match rules[4] = "+" at position 8 wi
th len 1: +
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]*" a
t position 9 with len 2: 15
[src/monitor/debug/expr.c,94,make_token] match rules[9] = ")" at position 11 w
ith len 1: )

```

图 6: 打印寄存器 info r

(二) 递归求值

1. 括号匹配

递归求值的部分可以对应于编译原理的语法及语义分析, 首先要对特殊符号“()”进行处理即检查括号配对情况是否正确, 我使用了一种比较直观的方法, 即将左括号标记为 1 右括号标记为 -1, 那么整个表达式最终的加和应该为 0 才是一个正确的可以进行运算的表达式。

括号匹配

```

1 //nemu/src/monitor/debug/expr.c
2 bool check_parentheses(int p, int q){
3 //pq表示表达式字符串的起始和结束位置
4 int sum = 0; //sum最终为0 表示表达式括号可全部匹配
5 if(!(tokens[p].type=='(' && tokens[q].type=='))
6     return false;
7 else{
8     for(int i=p; i<q; i++){
9         {
10             if(tokens[i].type=='(') sum++;
11             else if(tokens[i].type==')') sum--;
12             if(sum==0) return false;
13             // 在迭代到最后一个符号前如果为0则说明匹配错误
14         }
15         sum--; //把q的)算上
16         if(sum==0)
17             return true;
18         return false;
19     }
20 }

```


2. 主运算符

之后的寻找主运算符的部分可以比作建立语法树,只需要对各个运算符的优先级进行标注即可。下附代码是监视点之后的完整版。

主运算符获取及计算

```

1 //nemu/src/monitor/debug/expr.c
2 uint32_t eval(int p,int q){
3     if(p>q) return 0; // 错误参数!
4     else if(p==q) // 只有一个token的情况: hex dec reg
5     {
6         uint32_t res;
7         if(tokens[p].type==TK_HEX) sscanf(tokens[p].str,"%x",&res); // 16进制
8         else if(tokens[p].type==TK_DEC) sscanf(tokens[p].str,"%d",&res); // 10进制
9         else if(tokens[p].type==TK_REG){ // 寄存器
10             char temp[3]={tokens[p].str[1],tokens[p].str[2],tokens[p].str[3]};
11             // 去掉寄存器前的$
12             for(int i=0;i<8;i++) // 32bit
13                 if(!strcmp(temp,regl[i])) return cpu.gpr[i]._32;
14             for(int i=0;i<8;i++) // 16bit
15                 if(!strcmp(temp,regw[i])) return cpu.gpr[i]._16;
16             for(int i=0;i<8;i++) // 8bit
17                 if(!strcmp(temp,regb[i])) return cpu.gpr[i%4]._8[i/4];
18             if(strcmp(temp,"eip")) return cpu.pc; // eip
19         }
20         else assert(0);
21         return res;
22     }
23     else if(check_parentheses(p,q)) // 检查括号是否匹配
24         return eval(p+1,q-1); // 是则去掉括号计算括号内部
25     else // 找中心算符的部分
26     {
27         int op=0,prev=10; // 优先级
28         bool lp=false; // 标记左括号
29         for(int i=p;i<=q;i++)
30         {
31             if(tokens[i].type==')'){lp=false;continue;}
32             if(lp) continue;
33             if(tokens[i].type=='('){lp=true;continue;}
34             switch(tokens[i].type) // prev 越大优先级越高
35             {
36                 case TK_OR: if(prev>1){prev=1;op=i;continue;}
37                 case TK_AND: if(prev>2){prev=2;op=i;continue;}
38                 case TK_NEQ: if(prev>3){prev=3;op=i;continue;}
39                 case TK_EQ: if(prev>3){prev=3;op=i;continue;}
40                 case TK_LE: if(prev>4){prev=4;op=i;continue;}
41                 case TK_GE: if(prev>4){prev=4;op=i;continue;}
42                 case '<': if(prev>4){prev=4;op=i;continue;}
43                 case '>': if(prev>4){prev=4;op=i;continue;}

```

```

44         case TK_LS: if (prev > 5) { prev = 5; op = i; continue; }
45         case TK_RS: if (prev > 5) { prev = 5; op = i; continue; }
46         case '+': if (prev > 6) { prev = 6; op = i; continue; }
47         case '-': if (prev > 6) { prev = 6; op = i; continue; }
48         case '*': if (prev > 7) { prev = 7; op = i; continue; }
49         case '/': if (prev > 7) { prev = 7; op = i; continue; }
50         case '!': if (prev > 8) { prev = 8; op = i; continue; }
51         case TK_NEG: if (prev > 9) { prev = 9; op = i; continue; }
52         case TK_POI: if (prev > 9) { prev = 9; op = i; continue; }
53         default: continue;
54     }
55 }
56 uint32_t val1 = eval(p, op - 1);
57 uint32_t val2 = eval(op + 1, q);
58 switch (tokens[op].type) // 计算
59 {
60     case TK_OR: return val1 || val2;
61     case TK_AND: return val1 && val2;
62     case TK_NEQ: return val1 != val2;
63     case TK_EQ: return val1 == val2;
64     case TK_LE: return val1 <= val2;
65     case TK_GE: return val1 >= val2;
66     case '<': return val1 < val2;
67     case '>': return val1 > val2;
68     case TK_LS: return val1 << val2;
69     case TK_RS: return val1 >> val2;
70     case '+': return val1 + val2;
71     case '-': return val1 - val2;
72     case '*': return val1 * val2;
73     case '/': return val1 / val2;
74     case '!': return !val2;
75     case TK_NEG: return -1 * val2;
76     case TK_POI: return vaddr_read(val2, 4);
77     default: assert(0);
78 }
79 }
80 }
81
82 //nemu/src/monitor/debug/expr.c——expr函数调用即可
83 return eval(0, nr_token - 1);

```

之后再在 ui.c 中完善 cmd_p 函数即可

表达式计算

```

1 //nemu/src/monitor/debug/ui.c
2 static int cmd_p(char *args){
3     bool success = false;
4     //char *arg = strtok(NULL, " "); //注释掉是因为expr中可能有空格
5     if (args == NULL) { printf("Lack of parameter!\n"); return 0; }

```

```

6      uint32_t res=expr(args,&success);
7      printf("Decimal:%u   Hex:0x%x\n",res,res);
8      return 0;
9  }

```

3. 负数

此时指导书又提出了新的刁难? 负数的实现, 难点在于分裂减号和负号, 要正确判断负号的出现, 之后对负号进行标记并使其获得最高优先级即可。不是很懂为什么要做负数, 指导书写的是你迟早会面临类似的问题, 所以做了, 但是后面的随机测试又要求无符号整型, 所以在cmd_p中的输出就写成了%u。然后过了几天老师发了等价于参考代码和样例的指导书, 于是我又把%d改回去了, 负数仍显示负数, 毕竟我还有个Hex 十六进制的表示。

负数

```

1 //nemu/src/monitor/debug/expr.c
2 {"-", "'-'}, // minus 识别token处
3 // 对于负号优先度的处理前面代码以及有过就不再赘述
4 // 在expr函数中进行标记
5 if(nr_token!=1) // 只有一个符号的时候不用区分
6     for(int i=0;i<nr_token;i++) // 检查负号还是减号
7     {
8         if(tokens[i].type=='-'&&(i==0||tokens[i-1].type=='('||
9         tokens[i-1].type==TK_NEG||tokens[i-1].type=='-'||
10        tokens[i-1].type=='+'||tokens[i-1].type=='*')
11        ||tokens[i-1].type=='/')
12        {
13            tokens[i].type=TK_NEG;
14            // 第一个符号、左边为 (、负号后跟+*/的时候都标记为负号
15        }
16    }

```

图 7: 负数无符号

```

(nemu) p 3*2
[src/monitor/debug/expr.c,94,make_token] match rules[5] = "-" at position 0 with
len 1: -
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]" a
t position 1 with len 1: 3
[src/monitor/debug/expr.c,94,make_token] match rules[6] = "(" at position 2 wi
th len 1: (
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]" a
t position 3 with len 1: 2
Decimal: 6 Hex:0xfffffa
(nemu)

```

图 8: 负数有符号

到此整个表达式求值的阶段二就算是功能实现完成了, 验证结果如下:

```

(nemu) p 21*(222-42*(3))/16
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]" a
t position 0 with len 2: 21
[src/monitor/debug/expr.c,94,make_token] match rules[4] = "(" at position 2 wi
th len 1: (
[src/monitor/debug/expr.c,94,make_token] match rules[8] = "(" at position 3 wi
th len 1: (
[src/monitor/debug/expr.c,94,make_token] match rules[0] = "+" at position 4 wi
th len 1: +
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]" a
t position 5 with len 3: 222
[src/monitor/debug/expr.c,94,make_token] match rules[5] = "-" at position 6 wi
th len 1: -
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]" a
t position 9 with len 2: 42
[src/monitor/debug/expr.c,94,make_token] match rules[6] = "(" at position 11 w
ith len 1: (
[src/monitor/debug/expr.c,94,make_token] match rules[8] = "(" at position 12 w
ith len 1: (
[src/monitor/debug/expr.c,94,make_token] match rules[0] = "+" at position 13 w
ith len 1: +
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]" a
t position 14 with len 1: 3
[src/monitor/debug/expr.c,94,make_token] match rules[9] = ")" at position 15 w
ith len 1: )
[src/monitor/debug/expr.c,94,make_token] match rules[9] = ")" at position 16 w
ith len 1: )
[src/monitor/debug/expr.c,94,make_token] match rules[7] = "/" at position 17 w
ith len 1: /
[src/monitor/debug/expr.c,94,make_token] match rules[0] = "+" at position 18 w
ith len 1: +
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]" a
t position 19 with len 2: 16
Decimal: 27 Hex:0x1b

```

图 9: 验证 1

```

(nemu) p 3-8/4*(2+15)
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]" a
t position 0 with len 1: 3
[src/monitor/debug/expr.c,94,make_token] match rules[5] = "-" at position 1 wi
th len 1: -
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]" a
t position 2 with len 1: 8
[src/monitor/debug/expr.c,94,make_token] match rules[7] = "/" at position 3 wi
th len 1: /
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]" a
t position 4 with len 1: 4
[src/monitor/debug/expr.c,94,make_token] match rules[6] = "(" at position 5 wi
th len 1: (
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]" a
t position 7 with len 1: 2
[src/monitor/debug/expr.c,94,make_token] match rules[4] = "+" at position 8 wi
th len 1: +
[src/monitor/debug/expr.c,94,make_token] match rules[2] = "[0-9][1-9][0-9]" a
t position 9 with len 2: 15
[src/monitor/debug/expr.c,94,make_token] match rules[9] = ")" at position 11 w
ith len 1: )
Decimal: 3 Hex:0x3

```

图 10: 验证 2

4. 随机测试

于是又有了新的难题,老师给的代码里并没有 nemu/tools 文件夹,但是 ics2019 代码中是有的,所以还是想尝试着做一下,但是差不多就好像是为了测试前一个程序的 bug 写了另一个更大的 bug。

框架也是事先做好的,只需要补充代码就可以了,整体上没有理解的难度。

随机测试

```

1 // nemu/tools/gen-expr/gen-expr.c
2 static char buf[65536]=""; // init
3 int prob[10]={0,0,1,1,2,2,2,2,2,2};
4 // 因为直接用rand均匀分布效果不好,为增加复杂性手动调整概率
5 static inline void gen_rand_op(){// 随机生成运算符
6     switch(rand()%4){
7         case 0:{strcat(buf,"+");break;}
8         case 1:{strcat(buf,"-");break;}
9         case 2:{strcat(buf,"*");break;}
10        case 3:{strcat(buf,"/");break;}
11        default:assert(0);
12    }
13 }
14 static int sign=0;// 记录迭代次数
15 static inline void gen_rand_expr(){// 随机生成表达式
16     if(sign>50){return;}
17     sign++;
18     switch(prob[rand()%10]){// 0 数字 1 括号 2 复杂表达式拼接(包含1、2)
19         case 0:{char s[4];int a=rand()%10;while(a==0)a=rand()%10;
20             while(prob[rand()%10])
21                 a+=rand()%10;sprintf(s,"%d",a);strcat(buf,s);break;}
22         case 1:{if(prob[rand()%10])strcat(buf," ");strcat(buf,"(");
23             if(!prob[rand()%10])strcat(buf," ");gen_rand_expr();
24             if(!prob[rand()%10])strcat(buf," ");strcat(buf,")");break;}
25         default:{if(!prob[rand()%10])strcat(buf," ");
26             gen_rand_expr();if(!prob[rand()%10])strcat(buf," ");
27             gen_rand_op();if(!prob[rand()%10])strcat(buf," ");
28             gen_rand_expr();break;}
29     }
30 }
31 // 使用sprintf写到文件中的时候加了判断,除数为0则continue
32 // 之后在main中调用并注意清0即可

```

```

main.c
2610 ( ( 59)) * 42+74+ (42) +25-27/41-9
410 25+ (21+ (14 ) * ( (11))+42* (5) )
4294939624 16- (36)*32 * ( ( (24 ))) -40
2 12/ ( 3* ( 2 ))
3 79/25
36 31+ (5)
30 (30)
2 34/ (16)
8 ( 8)
15 15
77 (77 )
41 41
29 7+17- (22* (48) ) / ( (3+ (12 - (36*7))+55))
27 27
0 7 /24
47 ( 47)
465 ( 51)* 9+ (6)
29 (29)
4 (4)
23 23
918 51*18
14 14
32 32
17956 1+ (13* ( 28 + ( 13*22*13-10)/22)* 22/28/103 *27* 35)
9 ( ( 9 ))
1 (29 ) /19

input
2610 ( ( 59)) * 42+74+ (42) +25-27/41-9
410 25+ (21+ (14 ) * ( (11))+42* (5) )
4294939624 16- (36)*32 * ( ( (24 ))) -40
2 12/ ( 3* ( 2 ))
3 79/25
36 31+ (5)
30 (30)
2 34/ (16)
8 ( 8)
15 15
77 (77 )
41 41
29 7+17- (22* (48) ) / ( (3+ (12 - (36*7))+55))
27 27
0 7 /24
47 ( 47)
465 ( 51)* 9+ (6)
29 (29)
4 (4)
23 23
918 51*18
14 14
32 32
17956 1+ (13* ( 28 + ( 13*22*13-10)/22)* 22/28/103 *27* 35)
9 ( ( 9 ))
1 (29 ) /19

```

图 11: 随机测试 input

之后再对 NEMU 的 main.c 进行相应修改就可以让 NEMU 一启动起来就进行自动计算 input 提供的算术表达式并与答案相对比输出结果。

随机测试

```

1 //nemu/src/main.c
2 // 添加头文件
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include "monitor/expr.h"
7 // to test p EXPR
8 FILE*fp;
9 fp=fopen("/home/ubuntu/ics2019/nemu/tools/gen-expr/input","r");
10 char buf[101];
11 while( fgets( buf, sizeof( buf), fp)){
12     char *p=strtok( buf, " ");
13     char *q=strtok( NULL, "\n");
14     bool success=true;
15     int res=expr(q,&success);
16     if(success){
17         if( atoi(p)==res){
18             printf("%s=%s, myresult=%d, equal=true\n",q,p,res);
19         } else {printf("%s=%s, myresult=%d, equal=false\n",q,p,res);}
20     }
21     else {printf("fail to eval");}
22 }
23 fclose(fp);

```

```
(15)=15, myresult=15, equal=true
( (28) )=28, myresult=28, equal=true
(30)=30, myresult=30, equal=true
( 38)=38, myresult=38, equal=true
15=15, myresult=15, equal=true
8* 21=168, myresult=168, equal=true
49*28=1372, myresult=1372, equal=true
48*4=192, myresult=192, equal=true
(20- 12)/22=0, myresult=0, equal=true
27=27, myresult=27, equal=true
25=25, myresult=25, equal=true
( 34*35 -2)=1188, myresult=1188, equal=true
(14)=14, myresult=14, equal=true
20 -7=13, myresult=13, equal=true
( ( 27) )=27, myresult=27, equal=true
( 86 -17)=69, myresult=69, equal=true
( 2)=2, myresult=2, equal=true
14+8=22, myresult=22, equal=true
(27)=27, myresult=27, equal=true
(28)+( 9 )/35=28, myresult=28, equal=true
(8/16 + (16) )=16, myresult=16, equal=true
25=25, myresult=25, equal=true
1+ ( 15)=16, myresult=16, equal=true
25+ (21+ (14)* ( (11))+42* (5) )=410, myresult=410, equal=true
```

图 12: 随机测试验证结果

其实整体来说, 随机测试并没有严格意义上的完整实现, 因为有几个小细节被我 skip 了, 以及学姐的段错误解决等详细想法见问题与解决。

之后再将表达式的部分加入到前面扫描内存当中实现完整的功能

扫描内存 2.0

```
1 //nemu/src/monitor/debug/ui.c cmd_x
2 if(arg1[0]!='0'){ //not 0x hex address
3     bool success=true;
4     temp=expr(arg1,&success);
5     if(!success){ printf("Illegal Address!\n"); return 0;}
6 }
```

```
(nemu) x 39 0x100000/0x2+0x25000*0x2
memory
0x100000: 0xb8 0x34 0x12 0x00
0x100004: 0x00 0xb9 0x27 0x00
0x100008: 0x10 0x00 0xb9 0x01
0x10000c: 0x66 0xc7 0x41 0x04
0x100010: 0x01 0x00 0xbb 0x02
0x100014: 0x00 0x00 0x00 0x66
0x100018: 0xc7 0x84 0x99 0x00
0x10001c: 0xe0 0xff 0xff 0x01
0x100020: 0x00 0xb8 0x00 0x00
0x100024: 0x00 0x00 0xd6
(nemu) █
```

图 13: 扫描内存 2.0

四、 阶段三

(一) 扩展表达式功能

在前面阶段二实现的基础上对优先级处理以及 token 添加即可, 唯一需要注意的是指针的 * 和负号的处理一样也是需要进行区分的

指针

```
1 //nemu/src/monitor/debug/expr.c
2 for(int i=0;i<n_token;i++)
3     if(tokens[i].type=='*'&&(i==0||(tokens[i-1].type!=TK_DEC
4         && tokens[i-1].type!=TK_HEX && tokens[i-1].type!='') ))
5         tokens[i].type=TK_POI;
```

然后很久以后老师又给了测试样例,(麻烦老师下次给早一点 QAQ), 不过也多亏老师的测试样例我才发现我的十六进制数正则表达式只认大写字母, 于是我又替换了一次图片为测试样例。

```
(nemu) info r
eax: 0x2a097062
ecx: 0x472a70cf
edx: 0x367e5aea
ebx: 0x5290fffa
esp: 0x7ca3b5ed
ebp: 0x27d9d913
esi: 0x7bf54f01
edi: 0x052c07cc
ax: 0x7062
cx: 0x70cf
dx: 0x5aea
bx: 0xffffa
sp: 0xb5ed
bp: 0xd913
si: 0x4f01
di: 0x 7cc
al: 0x62
cl: 0xcf
dl: 0xea
bl: 0xfa
ah: 0x70
ch: 0x70
dh: 0x5a
bh: 0xff
pc: 0x00100000
```

图 14: 此时的寄存器对应以验证正确性

```
(nemu) p 12
Decimal:12 Hex:0xc
(nemu) p 0x1a
Decimal:26 Hex:0x1a
(nemu) p $ebp
Decimal:1040576 Hex:0x100000
(nemu) p $ax
Decimal:28770 Hex:0x7062
(nemu) p $al
Decimal:98 Hex:0x62
(nemu) p $ah
Decimal:112 Hex:0x70
(nemu) p 1+12*3
Decimal:37 Hex:0x25
(nemu) p -(2+3)+8
Decimal:3 Hex:0x3
(nemu) p -1*5
Decimal:-5 Hex:0xfffffff5
(nemu) p -2
Decimal:-2 Hex:0xfffffff2
(nemu) p --2
Decimal:2 Hex:0x2
(nemu) p 12+6/3
Decimal:14 Hex:0xe
(nemu) p 1+12*3==37
Decimal:1 Hex:0x1
```

图 15: 测试样例 1

```
(nemu) p 1+12*3!=37
Decimal:0 Hex:0x0
(nemu) p !(1-1)
Decimal:1 Hex:0x1
(nemu) p 3==3 && (*0x100000==0x1234b8)
Decimal:1 Hex:0x1
(nemu) p *0x100000
Decimal:1193144 Hex:0x1234b8
```

图 16: 测试样例 2

(二) 实现监视点

首先需要在 nemu/include/monitor/watchpoint.h 中完善 wp_pool 结构, 之后再分别实现创建监视点, 查看监视点以及删除监视点的成员函数即可。

指针

```
1 //nemu/include/monitor/watchpoint.h
2 typedef struct watchpoint {
3     int NO;
4     struct watchpoint *next;
5     uint32_t init; // 旧值
6     char wpexpr[100]; // expr
7     int hit; // 触发次数
8
9 } WP;
10 WP* new_wp(); // 找到一个空闲的监视点
11 void free_wp(int no); // 释放
12 void view_wp(); // 查看
13 bool check_wp(); // 检查
14 void set_wp(char *args); // 建立
15 //nemu/src/monitor/debug/watchpoint.c
16 void init_wp_pool() {
17     int i;
18     for (i = 0; i < NR_WP; i++) {
```

```

19         wp_pool[i].NO = i;
20         wp_pool[i].next = &wp_pool[i + 1];
21         wp_pool[i].init=0; // 初始化
22         wp_pool[i].hit=0; // 初始化
23     }
24     wp_pool[NR_WP - 1].next = NULL;
25     head = NULL;
26     free_ = wp_pool;
27 }
28 WP* new_wp(){
29     if(head == NULL){
30         head = free_; // 直接从free 中拿
31         free_=free_>next;
32         head->next = NULL;
33         return head;
34     }
35     else {
36         if(free_ == NULL) // 无空闲
37             assert(0);
38         WP* temp = head;
39         while(temp->next!=NULL)
40             temp = temp->next;
41         temp->next = free_;
42         free_ = free_>next;
43         temp = temp->next;
44         temp->next = NULL;
45         return temp;
46     }
47 }
48 void free_wp(int no){ // 释放到 free_ 中
49     WP *wp=head;
50     while(wp!=NULL){
51         if(wp->NO == no) break;
52         wp = wp->next;
53     }
54     if(wp == NULL){
55         printf("Fail to free the wp");
56         return;
57     }
58     if(head == wp) head = head->next;
59     else {
60         WP* temp = head;
61         while(temp!=NULL){
62             if(temp->next == wp){
63                 temp->next = wp->next;
64                 break;
65             }
66             temp = temp->next;

```



```

67         }
68     }
69     wp->next = free_;
70     free_ = wp;
71 }
72 void view_wp(){
73     if(head==NULL) {printf("no watchpoint now!\n");}
74     else{
75         printf("NO\tExpr\tHit\n");
76         WP *temp = head;
77         while(temp!=NULL){
78             printf("%d\t%s\t%d\n",temp->NO,temp->wpexpr,temp->hit);
79             temp = temp->next;
80         }
81     }
82 }
83 bool check_wp(){ // 监视点有变动返回 true
84     WP *temp = head;
85     while(temp!=NULL) {
86         bool *success = false;
87         uint32_t new_val = expr(temp->wpexpr, success);
88         if(new_val != temp->init){
89             printf("Watchpoint %d: %s\n",temp->NO,temp->wpexpr);
90             printf("Old value = %d\nNew value = %d\n",
91                 temp->init, new_val);
92             temp->init = new_val; // 赋新值
93             return false;
94         }
95         temp = temp->next;
96     }
97     return true;
98 }
99 void set_wp(char *args){
100     WP* newwp=new_wp();
101     strcpy(newwp->wpexpr, args);
102     bool *success = false;
103     newwp->init=expr(newwp->wpexpr, success);
104     printf("Set watchpoint %d on %s successfully\n",
105         newwp->NO,newwp->wpexpr);
106 }

```

之后再在 ui.c 中调用函数即可实现监视点功能, 同时在 cpu_exec.c 中加入 check_wp 的内容即可实现监视点完整功能

监视点 ui

```

1 // nemu/src/monitor/debug/ui.c
2 static int cmd_w(char *args){ // 设置监视点
3     if(args==NULL){printf("Lack of parameter!\n");return 0;}
4     set_wp(args);

```

```

5         return 0;
6     }
7     static int cmd_d(char *args){ // 删除监视点
8         if(args==NULL){ printf("Lack of parameter!\n"); return 0;}
9         int num;
10        sscanf(args,"%d",&num);
11        if(num<0){ printf("Enter a positive number!\n"); return 0;}
12        free_wp(num);
13        printf("Delete watchpoint %d successfully\n",num);
14        return 0;
15    }
16    static int cmd_info(char *args){ // info w补充完善
17        char *arg = strtok(NULL, " "); // get r or w
18        if(strcmp(arg,"r")==0) // info r
19            isa_reg_display(); // reg.c API
20        else if(strcmp(arg,"w")==0){
21            view_wp();
22        }
23        else
24            printf("Unknown command:%s\n",arg);
25        return 0;
26    }
27    //nemu/src/monitor/cpu_exec.c
28    // 找到写着TODO check watchpoint的地方
29    if(check_wp()==false)
30        nemu_state.state=NEMU_STOP;
31    /*这里2019的代码和老师提供的2018不一样,因为
32    nemu_state在2019是个包含终止pc和ret值的结构体表示状态的
33    enum状态只是一个成员变量*/

```

由于我自己写的测试样例又和老师不一样,所以又一次进行了更新

```

Welcome to x86-NEMU!
For help, type "help"
(nemu) w $eip==0x100000
Set watchpoint 0 on $eip==0x100000 successfully
(nemu) info w
NO      Expr      Hit
0       $eip==0x100000 0
(nemu) c
Watchpoint 0: $eip==0x100000
Old value = 1
New value = 0
(nemu) info w
NO      Expr      Hit
0       $eip==0x100000 1
(nemu) d 0
Delete watchpoint 0 successfully
(nemu) info w
no watchpoint now!
(nemu) r
Unknown command 'r'
(nemu) c
nemu: HIT GOOD TRAP at pc = 0x00100026

[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 8
(nemu)

```

图 17: 设置、删除监视点

五、 遇到的 bug 及解决

终于完成了 PA1 的全部内容, 从 PA0 的环境配置到 PA1 的简易调试器真的是坎坷, 没有 STFW 的帮助真的寸步难行, PA 指导书与 ucore 指导书相比内容有趣了不少, 但是同样也有许多挖坑以及故意刁难的感觉, 以下是实验过程中遇到的机器和我本人的 bug 的不完全总结。

• 环境配置 PA0

- ✓ Vmware 在 32 位 ubuntu 上无法完成与本机之间的复制粘贴, 但 64 位可以。网上给出的解决方案是将 vmtools 重新安装: `sudo apt-get autoremove open-vm-tools sudo apt-get install open-vm-tools-desktop`, 重启虚拟机之后就可以正常复制粘贴但文件拖拽依然只能通过共享文件夹
- ✓ 必须要用复制粘贴是因为指导书提供的清华镜像源通过那条命令是更换 error 的, 所以直接上网搜如何更换阿里源, 32 位和 64 位以及版本不同会导致系统代号不同, 老师提供的 32 位镜像对应 xenial, 而我另外配置的最新 64 位对应 bionic。更换镜像之后, 所有的 apt-get 内容不超过 2 分钟。
- ✓ Vim 因为用过所以不觉得难受, 但是指导书给的例子必须要习惯 hjkl 代替上下左右确实比较痛苦。il<ESC>q1yyp<C-a>q98@1 这条 Vim 指令可以一行一个数字从 1-100 的输出, 其含义是 il 插入 1、yy 复制一整行 p 粘贴 ctrl+a(x) 是游标下字符 +1(-1)、q 到 q 之间是把这部分内容放寄存器、@1 重复一次; <C-v>24l4jd\$ 实现文本的左右颠倒, 其含义为 (确认光标在文本开头)ctrl+v 打开可复制 view 模式、24l 光标向右 24、4j 光标向下 4、d\$ 删除之后所有的 p 粘贴
- ✓ 直接在 32 位运行 mario 是不行的, 重新配置 ssr 和 X server 也不可以, 因为官方 18 的代码就要求是 64 位的, 会报错。所以 32 位的对应老师提供的代码, 而 18 或者 19 的代码在 64 位机器上运行是没有问题的。
- ✗ 将实验环境迁移为 MacOS 或 Windows 上的 Docker 以适应 2019 的代码

• 阶段一

- ✓ 2019 的代码把 eip 换成了 pc, 虽然不知道为什么要这么做, 但是秉持着不改动无关代码的原则, 我只对 pc 进行了显示上的 eip 转换。cpu.pc 的定义在 nemu/src/isa/x86/include/isa/reg.h 中, 即需要进行 union 定义的地方
- ✓ 最开始的在实现 si 单步执行的时候没有仔细看 cpu_exec 的源码, 所以执行 N 步的时候是直接使用的 for 循环, 目前来看是没有问题的, 但是到了 PA2 更新 PC 用循环就非常有趣了
- ✓ 扫描内存最早写的是直接 vaddr_read 按长度 4 字节的去读, 但是如果四字节一起读出来, 整个内存内容会和期望的 default_img 的内容是相反的, 因为 x86 是小端模式, 所以在一个一个字节取数据的时候就会先从小端取数, 而四个字节四个字节取的话就一个整体取出来, 所以将读取变成以 1 字节为单位读取, 这样结果就和预期一致了。

• 阶段二

- ✓ 词法分析的时候, 声明枚举符号 token 类型部分, 之所以 TK_NOTYPE 是从 256 开始, 是为了不与 ASCII 冲突, 但是稍微实验了下, 如果将值赋成一些特殊字符的编号也是没有影响的, 这一部分要很注重 KISS 法则, 不然后面怎么 bug 的都定位不了

- ✓ 在编译原理当中使用 yacc 工具编写词法分析的符号只需要一个反斜杠, 即 + 就可以了, 但是在 PA 里面却需要两个反斜杠, 这个问题的答案是在写报告的时候想到的, 就和 latex 一样, 反斜杠本身就是特殊符号, 所以还需要一个转义反斜杠来表示出它本身
 - ✓ cmd_p 的时候习惯性的写了 strtok 拆分, 然后报了挺长的错, 后来才想起来 expr 内部是允许有空格的, 一旦拆分会触发自己后面写的一个 assert。(典型的自己挖坑)
 - ✓ 随机测试确实理解和实现都没有难度, 但是有一个 bug 暴露了我的 C++ 代码能力, 拼接字符串我第一个想到的是笨拙的使用 char* 数组, 第二个想到的是复杂的用 strncpy, 错了好久以后才 STFW 了正确的 strcat, 太久没写过 C++ 了, 完全是试错
 - ✓ 随机测试有一个很神奇的地方, `fscanf(fp, "%d", &result);` 作为一段原有代码, 它的本意是读取程序生成的表达式运算结果, 但可能由于编译器担心没有 assert 的部分, 不能保证正确性, 所以报了 error, 而报错的原因很简单, 是 gcc 编译器将 warning 视作 error 了, 参考了一下 gen-expr 下的 makefile, 把其中 gcc 参数的 -Werror 去掉就行了, 虽然这种做法不正确, 但是可以最快解决问题
 - ✓ 按照 rand 的均匀分布容易让随机表达式总是落在简单和重复括号的区间, 所以直接自己写了一个适应程序的概率分布, 这样可以更多的生成复杂表达式; 还有一个是指导书提出的问题如何处理溢出, 通过加入迭代次数终止循环, 否则基本上只要开始生成复杂表达式就停不下来; 随机空格的实现就是在表达式生成中间通过随机概率插入;
 - ✗ 保证无符号运算我的理解是 %u 输出, 但是这并不是理想的结果, 那理想的方案是每运算一步即判断是否大于 0, 不是则舍去表达式; 过滤求值中有除 0 行为的表达式, 单纯除 0 的过滤也已经写到程序中了, 表达式中间结果为 0 作为除数的直接写在了 gen-expr 中, 在输出到文件的时候对这串表达式进行验证, 由于规定了无符号, 所以为负则可以跳过这次输出到文件。另外学姐的段错误我也报过, 第一次是因为除 0 表达式没有过滤完整, 第二次是因为 buf 溢出, 所以每打印完一个表达式对 buf 进行 memset 清 0 即可, 同样在代码中使用迭代次数来记录迭代生成表达式的次数, 防止迭代次数太多套死自己的情形发生, 间接的也避免了 buf 溢出的情形。
- 写完报告很久以后突然在准备做 PA2 的时候发现 PA1 的所有内容全都给做到 PA0 分支去了, 于是赶紧悬崖勒马又把代码重新挪了一遍, 幸亏我的报告比较详细才能较快的恢复, 虽然中间少了许多报错的 run 的 log 但是内容还是经检验无误的, 也让我彻底记住了开发中分支的重要性,(习惯于自己直接往 master 提交的严重后果)

六、 思考题 && 必答题

(一) 思考题

1. 如果没有寄存器, 计算机还可以工作吗? 如果可以, 这会对硬件提供的编程模型有什么影响呢?

可以工作, 虽然没有了寄存器, 但一些操作所依据的数据存储方式一定是要被定义的, 一些旧模型下的计算机使用 RAM 的第一页作为主要暂存空间, 因此前 256 个字节被称为“寄存器”, 即使它们是电子意义上的 DRAM。虽然可以工作, 但效率肯定远低于寄存器。编程模型似乎有个必要条件就是在寄存器中对指令的处理, 一周目没有很看懂。
2. 什么东西表征了 TRM 的状态? 在状态模型中, 执行指令和执行程序, 其本质分别是什么?

TRM 中存在一个叫做状态寄存器的组成部分,会存储图灵机当前的状态。执行指令和执行程序的本质区别是程序是许多指令的集合。此外程序寄存器仅用来存储程序的下一条指令所在地址。

3. 一个程序从哪里开始执行呢?

一般来说像 C/C++ 这种是从 main 函数开始执行的,但是汇编语言又似乎可以指定程序的入口地址,所以 STFW 到 `__attribute__` 机制,可以设置入口地址,而 gcc 是默认设定为 main 的

4. 阅读 `init_monitor()` 函数的代码,你会发现里面全部都是函数调用。按道理,把相应的函数体在 `init_monitor()` 中展开也不影响代码的正确性。相比之下,在这里使用函数有什么好处呢?

个人理解,方便调试和归类,全用源码替换,就算注释写得再好也会乱七八糟,如果把每一个不同的功能实现放在一个.c 文件里,虽然文件变多了,但是条理清晰了,对需要 RTFSC 的人很友好,对调试也友好

5. `parse_args` 这些参数是从哪里来的呢?

当然是 main 来的,调用逻辑是 `main`→`init_monitor`→`parse_args`,而 main 函数的参数是在运行的时候通过控制台输入的

6. 阅读 `reg_test()` 的代码,思考代码中的 `assert()` 条件是根据什么写出来的。

第一个 for 循环里面的 `assert` 检查 32 位和 16 位寄存器,即 EAX 低 16 位是否与 AX 一致;之后一部分验证 8 位是否与 16 位对应;最后验证是不是和 cpu 结构中的一致

7. 在 `cmd_c()` 函数中,调用 `cpu_exec()` 的时候传入了参数-1,你知道这是什么意思吗?

这个问题的答案在 `nemu/src/monitor/cpu-exec.c` 中 `cpu_exec` 的定义中,传入参数的类型是 `uint64_t`,所以-1 其实是最大的数 `0xffffffffffff`,所以这个 for 循环可以保证循环最多的次数,可以把所有的命令执行完而不至于说在我指令没有执行完之前程序提前结束

8. `opcode_table` 到底是个什么类型的数组?

是一个 `opcode_entry` 结构体类型的数组,通过 RTFSC 猜测是模拟指令码的结构体

9. 谁来指示程序的结束?

首先如果主函数里面有 `exit()` 的话也会直接结束,c 语言中有 `atexit()` 函数,如果将一组指向函数的指针传递给 `atexit()` 函数,那么在程序退出 `main()` 后,就会自动调用该函数;而且 main 函数返回值以后,操作系统还要根据返回值判断程序正常退出还是异常退出,之后程序才会结束运行;全局对象的析构函数会在 main 函数之后执行

10. 阅读 `nemu/Makefile`,尝试理解 NEMU 的编译过程。特别地,NEMU 是如何支持多种客户 ISA 的?你现在可以忽略这个问题,但你会在 PA2 中重新面对它。

这个 Makefile 相比 `ucore` 的真的简单了不少,基本就是包含文件,gcc 编译生成,链接,再之后就是 `run`、`clean` 这些的实现。多种 ISA,在最开头就定义了变量 `ISA ?= x86`,需要 isa 不同的时候调用 `$ISA` 变量即可

11. 为什么 `printf()` 的输出要换行?

不然所有的输出堆在一起吗,看着都难受。如果没有换行,那么缓存会一直填满,溢出是迟早的事情吧。而且这个 NEMU 只是一个模拟机,我以为他会和 `bash` 编译器一样自动换行,然而并没有,扫描内存的时候最后忘了写换行,(nemu) 真的就直接跟在后面了。。

12. 框架代码中定义 `wp_pool` 等变量的时候使用了关键字 `static`, `static` 在此处的含义是什么? 为什么要在此处使用它?

静态全局变量, 保证只在定义文件内有效, 既有全局变量的功能, 也避免了不必要的外部调用。其他源文件必须通过封装函数才可以实现对该变量的操作。

13. x86 的 `int3` 指令不带任何操作数, 操作码为 1 个字节, 因此指令的长度是 1 个字节。这是必须的吗? 假设有一种 x86 体系结构的变种 `my-x86`, 除了 `int3` 指令的长度变成了 2 个字节之外, 其余指令和 x86 相同。在 `my-x86` 中, 上述文章中的断点机制还可以正常工作吗? 为什么?

不能, 因为 `int3` 指令长度为 1 个字节时, 且存在想设置断点的指令的第一个字节处, 当程序执行到 `0xcc`, 就会出发异常进入调试, 但当 `int3` 指令的长度变为 2 个字节后, 会遇到如下问题: 因为 x86 架构上指令最短的长度就是 1 字节, 这样可以保证只有我们希望停止的那条指令被修改; 如果遇到单字节指令, 2 位长度的 `int3` 会影响到其他指令; 在运行过程中, 按每个字节的内容来解读, 改变后的 `int3` 指令就会被分开, 可能不会正确解读

14. 如果把断点设置在指令的非首字节 (中间或末尾), 会发生什么? 你可以在 GDB 中尝试一下, 然后思考并解释其中的缘由。

出现段错误

15. 你已经对 NEMU 的工作方式有所了解了。事实上在 NEMU 诞生之前, NEMU 曾经有一段时间并不叫 NEMU, 而是叫 NDB(NJU Debugger), 后来由于某种原因才改名为 NEMU。如果你想知道这一段史前的秘密, 你首先需要了解这样一个问题: 模拟器 (Emulator) 和调试器 (Debugger) 有什么不同? 更具体地, 和 NEMU 相比, GDB 到底是如何调试程序的?

Emulator 是用于分析研究目标系统本身, 其本身要跟 CPU 所有内部行为一致, Debugger 的最基本功能就是将一个飞速运行的程序中断下来, 并且使其按照用户的意愿执行, 主要靠异常实现

此外还有几个建议二周目思考的题目实在是想不出, STFW 没有结果以及不知道往哪个方向 RTFM, 所以暂时放下。

(二) 必答题

必答题只有如何阅读手册这部分是需要回答的, 其余必答题皆为代码实现。

1. 假设你现在需要了解一个叫 `selector` 的概念, 请通过 i386 手册的目录确定你需要阅读手册中的哪些地方。即使你选择的 ISA 并不是 x86, 也可以尝试去查阅这个概念。

打开手册 `ctrl+f` 查找 `selector`, 定位到-CHAPTER 5 MEMORY MANAGEMENT —5.1.3 Selectors

2. 我选择的 ISA 是 x86; 分析需要 $450 \times 20 \times 30$, 简易调试器花费 $450 \times 20 \times 10$, 节约了 $450 \times 20 \times 20 = 18000s = 5h$

3. i386 手册阅读

(a) EFLAGS 寄存器中的 CF 位是什么意思? 阅读内容: 3.2 Binary Arithmetic Instructions (50 页) EFLAGS 寄存器中的 CF 位是进位借位标志位的意思, 用于表示无符号数加减运算时的进/借位。

(b) ModR/M 字节是什么? 阅读内容: 2.5.3 Memory Operands

(c) mov 指令的具体格式是怎么样的?

MOV — Move Data

Opcode	Instruction	Clocks	Description
88 /r	MOV r/m8,r8	2/2	Move byte register to r/m byte
89 /r	MOV r/m16,r16	2/2	Move word register to r/m word
89 /r	MOV r/m32,r32	2/2	Move dword register to r/m dword
8A /r	MOV r8,r/m8	2/4	Move r/m byte to byte register
8B /r	MOV r16,r/m16	2/4	Move r/m word to word register
8B /r	MOV r32,r/m32	2/4	Move r/m dword to dword register
8C /r	MOV r/m16,Sreg	2/2	Move segment register to r/m word
8D /r	MOV Sreg,r/m16	2/5,pm=18/19	Move r/m word to segment register
A0	MOV AL,moffs8	4	Move byte at (seg:offset) to AL
A1	MOV AX,moffs16	4	Move word at (seg:offset) to AX
A1	MOV EAX,moffs32	4	Move dword at (seg:offset) to EAX
A2	MOV moffs8,AL	2	Move AL to (seg:offset)
A3	MOV moffs16,AX	2	Move AX to (seg:offset)
A3	MOV moffs32,EAX	2	Move EAX to (seg:offset)
B0 + rb	MOV reg8,imm8	2	Move immediate byte to register
B8 + rw	MOV reg16,imm16	2	Move immediate word to register
B8 + rd	MOV reg32,imm32	2	Move immediate dword to register
C4 + r/m8	MOV r/m8,imm8	2/2	Move immediate byte to r/m byte
C7	MOV r/m16,imm16	2/2	Move immediate word to r/m word
C7	MOV r/m32,imm32	2/2	Move immediate dword to r/m dword

图 18: MOV 格式

4. nemu/目录下的所有.c 和.h 文件总共有多少行代码? `5332 find -name "*"*.h|.c" -type f | xargs cat | wc -l`
5. 我在 PA1 编写的代码 427 行, 切到 master 分支, 加到 Makefile 最后就可以了
6. 除去空格,`find . -name "*"*.h|.c"|xargs cat|grep -v $|wc -l` 截止 pal 共 4415 行
7. 使用 man 打开工程目录下的 Makefile 文件, 你会在 CFLAGS 变量中看到 gcc 的一些编译选项. 请解释 gcc 中的 -Wall 和 -Werror 有什么作用? 为什么要使用 -Wall 和 -Werror? -Wall, 打开 gcc 的所有警告, -Werror, 它要求 gcc 将所有的警告当成错误进行处理。正如前面指导书说的, 更快找到 error, 防止等到了 fatal 的时候才开始 debug 的手忙脚乱, 比如说未定义使用函数等等。

七、 总结

30h 诚不我欺, 虽然写代码要不了那么久但是不停的 RTFSC 和为了思考题的 STFW,RTFM 加上写报告等等一系列确实很费时间, 虽然 PA 是给 NJU 大二同学使用的, 但对于我这种没有形成 RTFSC 体系的人, 确实是一个很好的培养习惯的平台, 本身的代码不难, 主要是涉及的点和面比较多, 这次在配置上也费了不少时间, 最开始老师没有说到底 32 还是 64 位的时候 (当然也是我自己没仔细看), 干脆把两个都配了, 虽然时间长了些, 但是结果是很好。

总的来说, 各方面都收获很多, 希望下次可以再快一点。对 PA 理解更深一些。