PA1实验报告

PA讲义

南京大学匡亚明学院 刘志刚
学号：141242022

思考题

层次图

"Hello World" program
Micro operating system
Simulated Hardware(arch:IA-32)
NEMU
GNU/Linux
Simulated Hardware(arch:IA-32)
Virtual Box
Windows
Computer Hardware(arch:AMD64)

程序从哪里开始执行?

程序设计课程上说所有C语言程序都从main开始执行。但根据我读的CSAPP上面的解释，在Linux下，C语言程序的执行都是从一个固定的地址开始执行，首先执行一些初始化工作，主要是运行环境的初始化（C库等），而后才会从main开始执行。

关于opcode_table数组

以下是helper_fun和opcode_table的定义，我们可以发现helper_fun是一个函数指针类型，指向的函数接受一个地址，并返回一个int值。而opcode_table就是一个这样的函数指针的数组。

```
    typedef int (*helper_fun)(swaddr_t);
    helper_fun opcode_table [256];
```

指令的执行过程就是先取出eip所指指令的opcode，然后以此为下标到opcode_table里面去找相应的函数来执行这一条指令。因此我们可以猜测每一个或每一类指令都有一个函数来执行。而返回的int值应该是用来返回一些标志信息的，例如执行成功与否等。

```
make_helper(exec) {
    ops_decoded.opcode = instr_fetch(eip, 1);
    return opcode_table[ ops_decoded.opcode ](eip);
}
```

关于cpu_exec()函数

以下是cpu_exec函数的声明，我们可以发现它接受一个unsigned参数n，指定要执行多少条指令。

```
void cpu_exec(volatile uint32_t n);
```

以下是cpu_exec实现中最重要的那个循环，我们可以发现如果n=-1，由于n为unsigned，所以会执行4294967295步指令，相当于是直接执行到程序结束。但是这也引入了一个问题就是，对于死循环，如果使用c指令，程序并不能无限地跑下去。

```
for(; n > 0; n --) {
    /* Execute one instruction, including instruction fetch,
    * instruction decode, and the actual execution. */
    int instr_len = exec(cpu.eip);
    cpu.eip += instr_len;
    if(nemu_state != RUNNING) { return; }
}
```

谁来指示程序的结束？

在main函数return之后，还会有库和操作系统来执行一些收尾工作。

实现带有负数的算术表达式的求值

区分减号与负号的方法：

负号：

- 处在表达式的开头，前方没有表达式或前方为“(”
- 前方是非算术操作符

减号：

- 前方是表达式而非操作符

分裂负号时的注意点:
负号是单元运算符，因而要注意操作符的结合性

框架代码中使用的static关键字

使用static关键字可以限制变量或者函数的作用范围仅限于当前文件。这样可以保护变量，减少各文件之间的耦合。还可以减轻可能的命名空间污染问题。我们可以看到，在框架代码中大量使用了static关键字。

int3指令

int3指令长度为一字节，这是必须的。因为根据文章的介绍，在x86体系上，断点是通过software interrupts来实现的，使用int3覆盖目标指令的opcode字段，可以让程序在目标地址停下来。由于目标指令的长度最短可以为一个字节，如果int3指令长于一个字节的话，就会覆盖下一条指令。这个问题似乎并不算大，因为我们可以把int3覆盖的指令全都保存下来，但在这种情况下有一种问题不能解决，那就是断点设在一个跳转语句的后面，若多覆盖了任意一条指令，程序的执行会与我们设想的不同。所以int3指令长度必须是一字节。不能更长了。

随心所欲的断点

如果随心所欲的设置断点，可能不会触发断点，甚至会使程序运行不正确。
一个例子：
源程序

```
#include<stdio.h>

int main()
{
    int a=3;
    printf("%d\n",a);
    return 0;
}
```

`objdump`反汇编片段

```
080483fb <main>:
 804840c:   c7 45 f4 03 00 00 00    movl   $0x3,-0xc(%ebp)
 8048413:   83 ec 08                sub    $0x8,%esp
 8048416:   ff 75 f4                pushl  -0xc(%ebp)
 8048419:   68 d0 84 04 08          push   $0x80484d0
 804841e:   e8 ad fe ff ff          call   80482d0 <printf@plt>
```

其中我们可以发现0x804840c处的指令对应的就是`int a=3;`这条赋值语句。通过gdb来设置一个错误的断点，我们可以让`int3`指令的机器码错误地覆盖原有指令。我们可以发现，当讲断点设在0x804840f时，输出了204，而204正是0xcc！

```
Reading symbols from main...done.
(gdb) run
Starting program: /home/allen/test/main
3
[Inferior 1 (process 947) exited normally]
(gdb) break *0x804840f
Breakpoint 1 at 0x804840f: file main.c, line 5.
(gdb) run
Starting program: /home/allen/test/main
204
[Inferior 1 (process 951) exited normally]
```

Debuger与Emulator的区别

Debugger与Emulator都是与程序相关的，都要涉及到大量的底层内容。

> Debugger主要是与程序开发相关的，与程序的运行关系不是太大，Debugger并没有模拟运行程序的运行环境。Debugger是在受保护的环境下运行，权限受限，所以诸如读写内存等底层操作都必须通过系统调用来实现。被Debug的程序的运行环境是受保护的，正如这篇文章所提到的,Debugger对程序的追踪都必须要经过子进程的同意。所以Debugger是一个通过系统调用实现追踪子进程，是一个信号驱动的程序。
> Emulator与程序的执行相关。Emulator完全模拟了程序运行所需要的软硬件环境，所以Emulator并不需要像Debugger那样大量涉及到进程操作，信号量的处理。但Emulator更底层，每一条指令都需要你手写函数来处理。

通过目录定位关注的问题

如果想了解与selector相关的问题，我们需要阅读i386手册中5.1.3 `Selectors` 这一节。

必答题

查阅i386手册

- EFLAG寄存器中的CF位
  阅读范围：
  2.3.4 `Flags Registers`
- ModR/M字节
  阅读范围：
  17.2.1 `ModR/M and SIB Bytes`
- mov指令的具体格式
  阅读范围：
  3.1 `Data Movement Instructions`
  17.2.2.11 `Instruction Set Detail`

使用shell命令统计代码行数

以下是我与陈挚同学讨论完成的Makefile命令。

```
count:
    make -s count_1

count_1:
    echo "current"
    echo "all lines(blank line included) "
    cat `find nemu -name *\.[ch]` | wc -l  | tail -n 1
    echo "all lines(blank line excluded) "
    sed -e '/^[[:space:]]*$$/d' `find nemu -name *\.[ch]`  | wc -l | tail -n 1
    echo "original"
    echo "all lines(blank line included) "
    git checkout origin/master >/dev/null 2>&1
    cat `find nemu -name *\.[ch]` | wc -l  | tail -n 1
    echo "all lines(blank line excluded) "
    sed -e '/^[[:space:]]*$$/d' `find nemu -name *\.[ch]`  | wc -l | tail -n 1
    git checkout master >/dev/null 2>&1
```

输入make count即可统计代码行数以及空行数。输出如下：

```
allen@debian:~/ics/ics2015$ make count
make -s count_1
current
all lines(blank line included)
4645
all lines(blank line excluded)
3837
original
all lines(blank line included)
3696
all lines(blank line excluded)
2939
```

gcc的一些编译选项

`-Wall`意为开启所有类型的warning

`-Werror`意为将warning当作error

使用这两个编译选项之后，可以确保gcc对我们的代码进行最严格的检查，及早发现可能存在的问题。

实验进度

阶段1

- [x] 单步执行
- [x] 打印寄存器状态
- [x] 扫描内存
  - 不检查内存地址的有效性

阶段2

- [x] 十进制支持
  - 输入的十进制数范围为0-INT_MAX
- [x] 十六进制支持
  - 输入的十六进制数范围为0-INT_MAX
- [x] 访问寄存器支持
- [x] 解引用支持
  - 不检查输入地址的有效性
- [x] 算术操作符支持
- [x] 逻辑操作符支持
  - [x] 短路求值支持
- [x] 取负支持

阶段3

- [x] 插入监视点
  - [x] 监视点池满时返回错误信息
- [x] 删除监视点
- [x] 显示所有监视点
- [x] 监视点的触发

- 允许同时触发多个监视点，打印所有值发生改变的监视点

实现思路

首先感谢学长为我们写了框架代码，然后我们什么都好干了。
主体部分均参照PA讲义来完成，在eval中我开始将运算符根据它们的操作数个数以及结合性等分类处理，同时参照K&R，填写了运算符的信息。以下是operators的一些信息，以及经过修改的eval的大致框架。

```
// operator type
enum {unary,binary,unknown};
// associativity of operator
// l2r: left to right
// r2l: right to left
enum {l2r,r2l,no_asso};
// operator property: arithmetic or logic
enum {op_arith,op_logic,no_prop};

struct op_info
{
    int token_type;
    int priority;
    int op_type;
    int op_prop; // operator property: arithmetic or logic
    int asso;    //associativity
    bool sce;    //short-circuit evaluation
};

static unsigned eval(int p, int q,bool *success) {
    unsigned ret=0;
    *success=true;
    if(p > q)
    {
    }
    else if(p == q)
    {
        /* Single token.
         * now this token should be a number or a register.
         * Return the value of the number.
         */
```

```
32.             switch(tokens[p].type)
33.             {
34.             }
35.         }
36.         else if(paren(p, q))
37.         {
38.             // surrounded by parentheses
39.         }
40.         else
41.         {
42.             // splitting token
43.             int op = dominate(p,q);
44.             struct op_info a=info_op(tokens[op].type);
45.
46.             if(a.op_type==unary)
47.             {
48.                 if(a.asso==l2r)
49.                 {
50.                 }
51.                 else if(a.asso==r2l)
52.                 {
53.                 }
54.                 else
55.                 {
56.                 }
57.
58.             }
59.             else if(a.op_type==binary)
60.             {
61.                 if(a.asso==l2r)
62.                 {
63.                 }
64.                 else if(a.asso==r2l)
65.                 {
66.                 }
67.                 else
68.                 {
69.                 }
70.
71.             }
72.             else
73.             {
74.             }
75.         }
76.         return ret;
```

```
77.  }
```

框架代码的一个bug

bug描述：

位置：nemu/src/monitor/debug/expr.c

```
1.  void init_regex() {
2.      int i;
3.      char error_msg[128];
4.      int ret;
5.
6.      for(i = 0; i < NR_REGEX; i ++) {
7.          ret = regcomp(&re[i], rules[i].regex, REG_EXTENDED);
8.          if(ret != 0) {
9.              regerror(ret, &re[i], error_msg, 128);
10.             Assert(ret != 0, "regex compilation failed: %s\n%s", error_
    msg, rules[i].regex);
11.         }
12.     }
13. }
```

应该将第10行的`Assert(ret!=0,...)`改为`Assert(ret==0,...)`，不然在遇到错误的正则串时，不会报错，会导致后续的regexec匹配失败，对于某些错误正则串会导致后续的regexec触发段错误。

发现过程：

首先感谢祝于晴触发了这个bug。在调试时，我发现regexec会触发一个段错误，而且会导致之前的几个printf无法正常输出（abort之后，未刷新输出缓冲区）。我当时未注意到这是正则串的问题，我想即使匹配失败，也不应该触发一个段错误吧。我一度认为是regex库的bug。后来，我采取了对照未改动的框架代码，逐行添加的方式，最终发现是同学将"("的正则表达式写成了")"，这就导致了这个错误。
在发现之后，我仍然以为这是regex库的bug，我今天自己另外拿regcomp来编译")"，后来发现

regcomp正确地返回了非零值。后来，我再次检查框架代码中的init_regex()，才发现了这个问题的根源。修改之后，形如"("这样的错误正则串就不会编译通过，可以较早地定位问题。

问题及解决方法

- 我最初的打印寄存器是用sprintf实现的，打印一个char也是直接用sprintf("%x",a)来实现的，但我的同学发现了一个bug，如果把直接用其来打印一个char时，会出现一个不停输出1的死循环，后来我检查了一下，发现sprintf是将char转换成了一个int型再来打印的。对char执行符号扩展。后来我将char改成了unsigned char，这样就变成了零扩展，问题就得到了解决。看来参数类型的匹配以及unsigned与signed的差别与转换要小心！
- 在表达式求值时出现了同样的表达式序列会出现不同结果的问题，后来我检查了半天仍然不知道是哪里的问题，后来我在expr函数开始处加上了对所有全局变量的初始化操作，问题得到了解决。看来变量的初始化十分重要，同时对于频繁调用的程序段，要注意减少全局变量的量，并尽量减少对全局变量的写带来的副作用。
- gcc报错提示信息过长，导致在终端上无法显示完全。我查阅了一些资料，使用freopen函数，将stderr重定向到error文件，同时使用system()函数调用make，即可将报错信息输入到一个文件。可以看到全部的出错信息。
  后来我发现在可以直接在bash命令中重定向标准错误，这样就比用C语言重定向要简洁多了。可以这样写`make run 2>error`。
- 我在实现watchpoint功能时，最开始是使用`char*`来保存一个表达式的，但后来在调试时，我发现在一开始将其赋值时，是能够打印出表达式的，但后来在执行过程中表达式莫名其妙地消失了。后来我查阅了`readline`的手册，上面说readline返回`char*`是用malloc分配的，由使用者自行释放。后来我着重查找了一下free函数的调用，发现在`rl_gets()`函数中，每当一行处理函数之后，分配的空间就被丢弃了，后来我选择了将表达式拷贝到一个数组中。

```
1.   char* rl_gets() {
2.        static char *line_read = NULL;
3.        if (line_read) {
4.            free(line_read);
5.            line_read = NULL;
6.        }
7.        line_read = readline("(nemu) ");
```

```
        if (line_read && *line_read) {
            add_history(line_read);
        }
        return line_read;
    }
```

我的思考

- 减少模块间的耦合度的一个重要方法就是减少全局变量个数。
- 对于C程序，保护一个源文件中的数据与代码的方法就是使用static关键字，我们可以看到框架代码中对于每个源文件中全局变量和函数大量使用了static关键字。
- 采取模块测试，增量开发的方式，可以减少出现难定位的bug的几率。
- 框架设计很重要，设计得好，代码扩展和修改就很容易。否则，每次要增减功能，都有可能要改动大量代码，相当于是推倒重来。
- 一定要先学会基础知识，再来实现，先知道有哪些坑，并避开。否则会浪费很多时间在一些恼人的细节上。如果我的那位同学认真学好了正则表达式，也就不会浪费几个晚上调试那个regexec触发段错误的问题了。
- 程序在被abort之后，输出缓冲区的内容可能不会被刷新。而且由于输出缓冲区的刷新对于我们是透明的，我们并不知道它会不会刷新，有没有刷新。有可能导致这一次执行这个printf语句正确输出了，下一次同一条printf语句却没有输出的鬼畜情况。