

华中科技大学

2017

系统能力综合训练 课程设计报告

题 目:	X86 模拟器设计
专 业:	计算机科学与技术
班 级:	CS1406
学 号:	U20151465
姓 名:	梅纽因
电 话:	15158575337
邮 件:	jeff.mei@outlook.com
完成日期:	2017-10-31 周二上午



计算机科学与技术学院

华中科技大学课程设计报告

目 录

1	课程设计概述.....	2
1.1	课设背景	2
1.2	设计任务	2
2	简易调试器.....	3
2.1	功能实现要点.....	3
2.2	必答题	10
2.3	主要故障与调试.....	11

1 课程设计概述

1.1 课设背景

理解"程序如何在计算机上运行"的根本途径是从"零"开始实现一个完整的计算机系统。华中科技大学计算机科学与技术系计算机系统基础课程的小型项目 (Programming Assignment, PA) 将提出 x86 架构的一个教学版子集 n86, 指导学生实现一个功能完备的 n86 模拟器 NEMU(NJU EMUlator), 最终在 NEMU 上运行游戏"仙剑奇侠传", 来让学生探究"程序在计算机上运行"的基本原理。

1.2 设计任务

NEMU 是一个基于 QEMU(Quick Emulator)的项目, 并去除了大量与课程内容差异较大的部分。PA 包括一个准备实验(配置实验环境)以及 5 部分连贯的实验内容, 包括:

- 简易调试器
- 冯诺依曼计算机系统
- 批处理系统
- 分时多任务
- 程序性能优化

实验环境为:

- CPU 架构: x64
- 操作系统: GNU/Linux
- 编译器: GCC
- 编程语言: C 语言

2 简易调试器

2.1 功能实现要点

PA1 的主要任务是实现一个简易调试器，为后续实验提供一个 debug 工具。调试器的主要功能包括：

- 单步执行 N 条指令 (si)
- 表达式求值 (p)
- 创建、删除监视点 (w/d)
- 打印寄存器和监视点状态 (info)

上述的命令均在 `nemu/src/monitor.debug/ui.c` 中的 `cmd_table` 结构体中有定义。对应每一种命令在该文件下均有一个专门的入口函数来执行相应的操作。

2.1.1 实现寄存器结构体

根据 x86 手册, CPU 内共有 8 个通用寄存器。我们即可以通过其位次访问 `gpr[i]`, 亦可以使用其别名访问。因此我们需要正确修改相关结构, 保证两种访问方式等价。

由于计算机中寄存器公用一片内存, 因此我们使用 `union` 实现内存公用, 定义如下结构体和枚举。(在 `nemu/include/cpu/reg.h` 中定义)

```
const char *regsl[] = {"eax", "ecx", "edx", "ebx", "esp", "ebp",  
"esi", "edi"};  
  
union {  
    union{  
        uint32_t _32;  
        uint16_t _16;  
        uint8_t _8[2];  
    } gpr[8];  
    struct{  
        rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;  
    };  
};
```

华中科技大学课程设计报告

实现 gpr 结构体后，运行 NEMU。当通过 `nemu/src/cpu/reg.c` 中的 `reg_test` 的断言测试时，说明该结构体的实现是正确的。

2.1.2 基础设施

1. 单步执行 N 条指令

当调试器读取到“si”字符串时，则会调用 `cmd_si` 函数进行相应的处理。若参数缺省，则默认为单步执行；若有参数则将参数转换为 `int` 类型后再调用 `cpu_exec()` 函数，参数则为执行的步数即可。测试结果如图 4.1 所示。

```
static int cmd_si(char *args) {
    char *arg = strtok(NULL, ""); //extract the first argument
    if (arg == NULL) {
        cpu_exec(1); //no argument given
    }
    else {
        int count = atoi(arg);
        cpu_exec(count);
    }
    return 0;
}
```



The screenshot shows the NEMU terminal interface. It starts with a welcome message and a prompt for help. The user enters 'si', which triggers a single-step execution. The terminal displays the current instruction address and its assembly code. The first instruction is at address 100000, which is a 'movl \$0x1234,%eax' instruction. The user then enters 'si 2', which triggers two more single-step executions. The second instruction is at address 100005, which is a 'movl \$0x100027,%ecx' instruction. The third instruction is at address 10000a, which is a 'movl %eax,(%ecx)' instruction. The terminal shows the state of the registers and the instruction being executed at each step.

图 4.1 单步调试测试结果

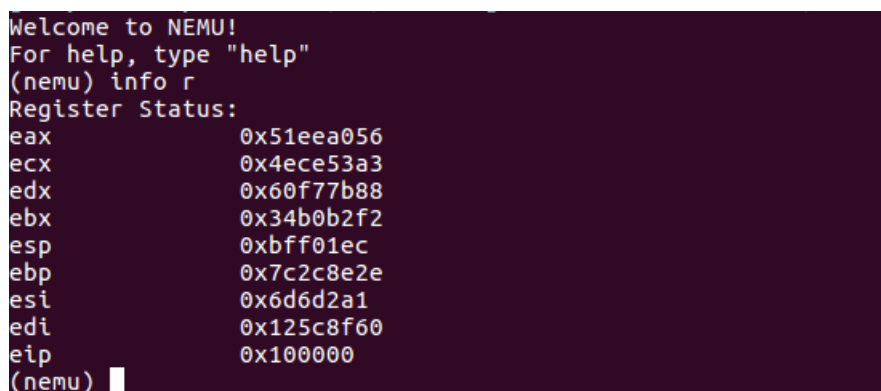
2. 打印寄存器和监视点状态

当调试器读取到“info”字符串时，则会调用 `cmd_info` 函数进行相应的处理。若参数为“r”，则根据寄存器的别名顺序依次打印出每一个寄存器内当前的状态；若参数

华中科技大学课程设计报告

为“w”，则会调用 `print_wp` 打印监视点当前的状态。这部分内容在下一小节之后会实现。测试效果如图 4.2 所示。

```
static int cmd_info(char *args) {
    char *arg = strtok(NULL, ""); //extract the first argument
    if (arg == NULL) {
        printf("Wrong argument!\n");
        return 0;
    }
    else if (!strcmp(arg, "r")) {
        printf("Register Status:\n");
        for(int i = R_EAX; i <= R_EDI; i++){
            printf("%s\t\t0x%x\n", reg_name(i, 4), reg_l(i));
        }
        printf("eip\t\t0x%x\n", cpu.eip);
        return 0;
    }
    else if (!strcmp(arg, "w")) {
        print_wp();
        return 0;
    }
    else {
        printf("Wrong argument!\n");
        return 0;
    }
}
```



```
Welcome to NEMU!
For help, type "help"
(nemu) info r
Register Status:
eax          0x51eea056
ecx          0x4ece53a3
edx          0x60f77b88
ebx          0x34b0b2f2
esp          0xbff01ec
ebp          0x7c2c8e2e
esi          0x6d6d2a1
edi          0x125c8f60
eip          0x100000
(nemu) 
```

图 4.2 打印寄存器状态测试

3. 扫描内存

当调试器读取到“x”字符时，则会调用 `cmd_x` 函数做相应的处理。参数内容则为一个整形 `N` 和一个表达式 `EXPR`。函数功能为：首先求出表达式 `EXPR` 的值，将结果作为起始内存地址，随后以十六进制形式输出连续的 `N` 个 4 字节。

进入 `cmd_x` 函数后，首先提取第一个和第二个参数并转换成可处理的整形和表达式结果。若转换失败，则输出错误信息；若转换成功，以表达式 `EXPR` 的值作为内存地址 `addr`，随后以十六进制形式输出连续的 `N` 个 4 字节。

```
static int cmd_x(char *args){
    char *arg1 = strtok(NULL, " ");
    char *arg2 = arg1 + strlen(arg1) + 1;
    int num = atoi(arg1);
    bool success;
    vaddr_t addr = expr(arg2, &success);
    if(!success){
        printf("Invalid input!\n");
        return 0;
    }
    for(int i = 0; i < num; i++){
        vaddr_t addr_temp = addr + 4 * i;
        uint32_t data = vaddr_read(addr_temp, 4);
        if(i % 4 == 0){
```

华中科技大学课程设计报告

```
printf("0x%x:\t", addr_temp);  
}  
printf("0x%x\t", data);  
if((i + 1) % 4 == 0){  
    printf("\n");  
}  
}  
  
printf("\n");  
return 0;  
}
```

2.1.3 表达式求值

这一部分是实现简易调试器中最为重要的一环。在 `nemu/src/debug/expr.c` 中的 `rule` 结构中，我们给出了以下 `token`，我们需要将他们的正则表达式按如下识别（而非运算）的优先级存放在该结构中。

表 4-1 token 及识别顺序表

优先级	运算符	正则表达形式
1	空格	"+"
2	+ - * /	"\\+" "\\-" "*" "/"
3	()	"\\(" "\\)"
4	十六进制数字	"0[xX][0-9a-fA-F]+"
5	十进制数字	"[0-9][0-9]*"
6	寄存器	"\\\$[a-zA-Z][a-zA-Z]+"
7	== !=	"==" "!="
8	&&	"&&" "\\ \\ \\ "
9	!	"!"

当每一个 `token` 均被成功识别之后，他们会被存放在 `tokens` 数组，进入 `expr` 函数继续进行处理。大致过程可以分为 3 步：

华中科技大学课程设计报告

1.对减与负(-), 乘与指针(*)进行区分。若‘-’或‘*’出现在表达式的开头, 或他们之前的一个 token 并非数值类型, 此时他们为负或指针; 否则他们为减或乘。

2.在 eval 函数中进行递归求值。

3.若在之前有任何的不匹配发生, 输出错误信息; 否则返回运算结果。

递归求值的核心思想为, 先在表达式中找到主运算符, 根据主运算符的位置将表达式拆分为左右两个子表达式; 随后再将左右两个子表达式继续做如下拆分, 直至表达式中仅剩下一个 token 时进行求值, 随后一步步向上回溯得到整个表达式的值。在实验指导中已经给出了框架逻辑, 这里不再赘述。

找到主运算符的 dominant_operator 函数的逻辑为, 我们首先遍历整个输入的表达式。若该 token 不是运算符或者在一个括号内时, 指针向下一个 token 移动; 若找到了符合要求的 token 时, 暂且存放当前的位置 i, 令取一个新的指针向后遍历。该函数的伪代码如下。

```
dominant_operator(p, q) {  
    i = p;  
    //go on traverse  
    while(i <= q && (!is_operator(tokens[i].type) || inside_par(i, p, q))){}  
    //invalid  
    if(i > q) {}  
    dominant_pos = i;  
    for(++i; i <= q; ++i){  
        if(!is_operator(tokens[i].type) || inside_par(i, p, q))  
            continue;  
        cur_type = tokens[i].type;  
        dominant_type = tokens[dominant_pos].type;  
        //if the pointer hit a more dominant operator  
        if(check_priority(cur_type, dominant_type) > 0)    dominant_pos = i;  
    }  
    return dominant_pos;  
}
```

华中科技大学课程设计报告

若新的指针指向的 token 满足以下条件时,则会取代当前存放的主运算符的位置:

1. 是一个运算符且不在括号内
2. 运算优先级比当前的运算符高

找到主运算符后,我们便可以递归地在 `eval` 函数中求值得到结果。

在上述过程中会使用到以下辅助函数:

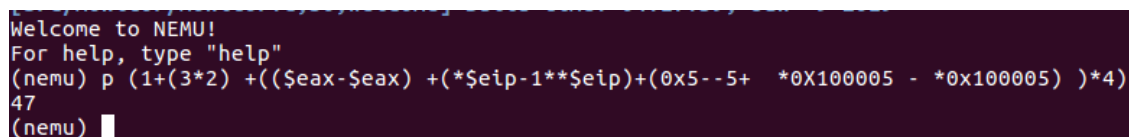
`check_parentheses`: 检查表达式的左右括号是否匹配,并返回相应的 bool 值。

`associate`: 检查该运算符的结合性,返回 0 代表左结合,返回 1 代表右结合。

`check_priority`: 根据输入的运算符返回其运算的优先级。数值越小代表运算优先级越高。

`read_reg`: 若当前的 token 类型为寄存器类型,则需要根据字符串的内容,与 `regsl`, `regsw`, `regsb` 三个数组内的寄存器别名字符串进行比较后找到相应的寄存器并读取结果。

表达式求值测试结果如图 4.3 所示。



```
Welcome to NEMU!
For help, type "help"
(nemu) p (1+(3*2) +(($eax-$eax) +(*$eip-1**$eip)+(0x5--5+ *0X100005 - *0x100005) )*4)
47
(nemu) █
```

图 4.3 表达式求值测试

2.1.4 监视点

在框架代码中,监视点是作为两个链表(分别代表空闲监视点和被占用监视点)进行存放的。

首先对 `nemu/include/monitor/watchpoint.h` 中的 `WP` 监视点数组进行修改。两个成员,分别为表达式结果 `value` 和表达式字符串本身 `expr`。

```
typedef struct watchpoint {
    int NO;
    struct watchpoint *next;
    /* TODO: Add more members if necessary */
    uint32_t value;                // original value of the expr
}
```

华中科技大学课程设计报告

```
char expr[MAX_LENGTH_OF_EXPR + 1];    // the expression
} WP;
```

随后在 `nemu/src/monitor/debug/watchpoint.c` 实现以下辅助函数：

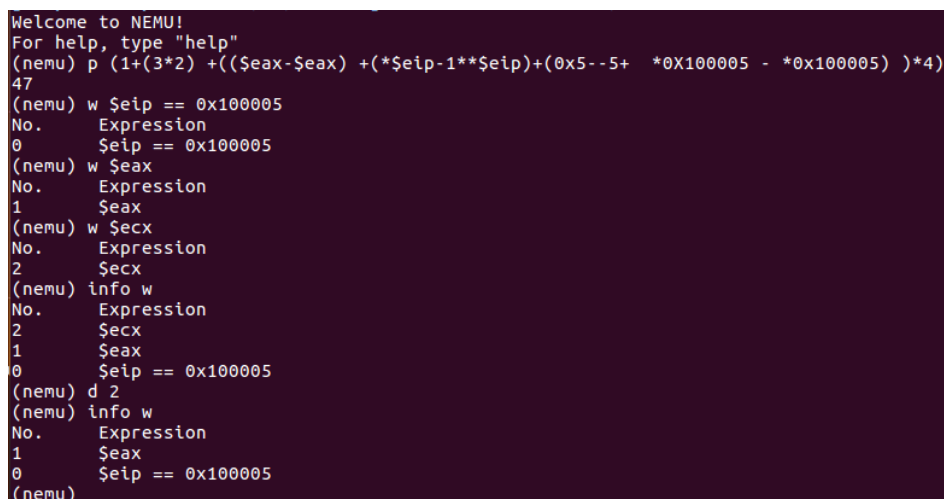
new_wp: 从空闲监视点链表中取出一个节点分配给新的监视点，返回新分配监视点的指针。

free_wp: 当一个监视点使用完毕时，将其从被占用监视点链表中转移到空闲监视点链表中。

check_wp: 被 `nemu/src/cpu/exec.c` 所调用。遍历被占用监视点链表，并重新计算表达式内容本身和最初的值 `value` 进行比较。如出现变化，则将表达式旧值和新值打印供调试用。

print_wp: 遍历被占用监视点链表，打印每一个监视点的状态。

监视点测试结果如图 4.4 所示。



```
Welcome to NEMU!
For help, type "help"
(nemu) p (1+(3*2) +(($eax-$eax) +(*$eip-1*$eip)+(0x5--5+  *0x100005 - *0x100005) )*4)
47
(nemu) w $eip == 0x100005
No.    Expression
0      $eip == 0x100005
(nemu) w $eax
No.    Expression
1      $eax
(nemu) w $ecx
No.    Expression
2      $ecx
(nemu) info w
No.    Expression
2      $ecx
1      $eax
0      $eip == 0x100005
(nemu) d 2
(nemu) info w
No.    Expression
1      $eax
0      $eip == 0x100005
(nemu)
```

图 4.4 监视点测试

2.2 必答题

1.理解基础设施

如果没有简易调试器，则花费的调试时间为 $500 \times 0.9 \times 30 \times 20 = 270,000s = 75h$ 。使用简易调试器可以节约 $1/3$ 的时间。

2.查阅 i386 手册

EFLAGS 中的 CF 位

位于 Appendix C – Status Flag Summary

华中科技大学课程设计报告

ModR/M

位于 17.2 Instruction Format

mov

位于 17 80836 Instruction Set

3.shell 命令

使用 `find . -name "*.c|h" |xargs cat|wc -l` 命令统计代码行数。PA1 部分共有 4336 行代码，PA0 共有 3824 行代码，共有 512 行的差别/

使用 `find . -name "*.c|h" |xargs cat|grep -v ^$|wc -l` 命令统计非空行代码行数。PA1 分支共有 3594 行。

4.使用 man

-wall 表示输出 warning 信息。

-werror 表示将所有的 warning 变为 error 使程序停止编译。

这种方法可以更好的帮我们调试代码，查看错误信息。

2.3 主要故障与调试

1.表达式 token 识别顺序错误

最开始在 `rule` 结构中加入每一个 token 时，我没有注意到每一个 token 存放的顺序其实和识别过程的优先级有关。因此我没有在意，将识别十进制的正则表达式放置于识别十六进制的 token 之前。这样当出现十六进制数时，对于其中的“0x”部分，表达式引擎会先识别 0 作为一个十进制数，随后将“x”作为一个非法 token。我在 `make_token` 函数中将每一个 token 识别的过程单步打印出来后才发现问题，注意到了框架代码中对于 precedence 的提示。

2.表达式长度问题

在进行最终的表达式测试时，对于教师给出的测试案例

$(1+(3*2)+((\$eax-\$eax)+(*\$eip-1**\$eip)+(0x5--5+ *0X100005 - *0x100005))*4)$

如果将表达式拆开进行求解结果合法，但是输入完整的表达式时无法输出正确的答案。经过单步调试后我发现在 `nemu/src/monitor/debug/expr.c` 中有如下定义：

```
Token tokens[64];
```

```
int nr_token;
```

这个定义说明了 tokens 数组最多只能存放 32 个 token，而该表达式由于过长，之后的 token 无法被正确识别，才导致了没有任何结果输出。将数组长度从 32 改为 64 即可修复该问题。

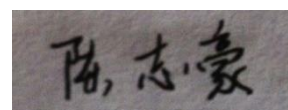
• 指导教师评定意见 •

一、原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签字：



二、对课程设计的学术评语（教师填写）

三、对课程设计的评分（教师填写）

评分项目 (分值)	报告撰写 (30 分)	课设过程 (70 分)	最终评定 (100 分)
得分			

指导教师签字：_____