

## PA1 目录-1613415 潘巧巧

关于 PA.....	2
NEMU 流程.....	3
4 个组成部件.....	3
NEMU 起点函数 main().....	3
Nemu 启动函数 init_monitor().....	3
用户界面主循环函数 ui_mainloop().....	4
简易调试器.....	5
实现流程.....	5
表达式求值 EXPR 相关知识.....	6
步骤简述 (expr()函数): .....	6
词法分析 (make_token()函数): 正则表达式。.....	6
递归求值 (evaluate()函数) .....	7
监视点相关 .....	9
PA1 实现过程.....	11
1. CPU_state.....	11
2. 单步调试 si[N] .....	11
3. 打印寄存器 info r.....	12
4. 算数表达式 EXPR 求值 (p EXPR) .....	13
5. 实现更复杂的表达式 EXPR 计算.....	15
6. 打印内存 (x N EXPR) .....	15
7. 设置监视点 (w EXPR) .....	16
8. 删除监视点 (d N) .....	17
9. 打印监视点 (info w) .....	17
10. 实现监视点变化则停止程序运行的效果.....	18
回答问题.....	20
遇到的坑.....	26
复习的知识和学到的经验.....	28

# 关于 PA

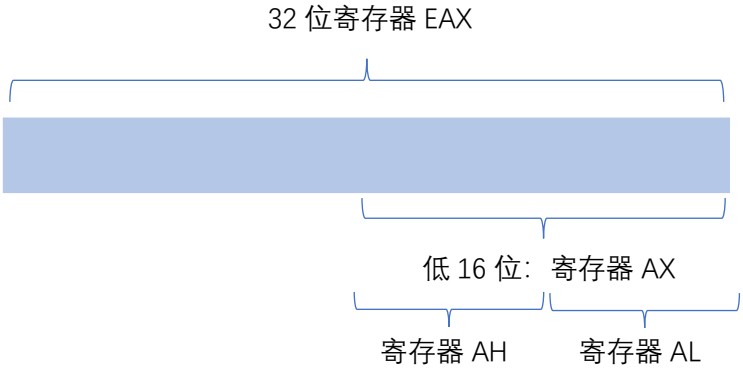
1、 PA 的目的是实现 NEMU（一款经过简化的 x86 全系统模拟器）和 Nanos-lite（简化版操作系统）。

2、 PA 实验本机设备层次

用户程序	Hello.c/马里奥/仙剑奇侠传
操作系统 3	Nanos-lite
硬件 3	NEMU
操作系统 2	Linux 操作系统
硬件 2	VM 虚拟机
操作系统 1	Windows10 操作系统
硬件 1	本机硬件

3、 图灵机核心结构部件和工作方式

- 1) 存储器：存放数据和程序。速度慢，容量大。
- 2) CPU：执行程序，负责处理数据的核心电路单元
- 3) 运算器：对数据进行处理
- 4) 寄存器：可以让 CPU 把正在处理中的数据暂时存放在其中。速度快，容量小。  
EAX, EDX, ECX, EBX, EBP, ESI, EDI, ESP: 32 位寄存器。  
AX, DX, CX, BX, BP, SI, DI, SP: 16 位寄存器。  
AL, DL, CL, BL, AH, DH, CH, BH 是 8 位寄存器。



- 5) 指令：指示 CPU 对数据进行何种处理。
- 6) 程序计数器 (PC/EIP)：指向下一条指令，使得当执行完一条指令之后，计算机就继续执行下一条指令。

# NEMU 流程

## 4 个组成部件

monitor+CPU+memory+device

monitor 主要用于监视这个虚拟计算机系统是否正确运行，负责与 GNU/Linux 进行交互(例如读写文件)，以及起到调试器的作用。

## NEMU 起点函数 main()

可以看见先调用 init\_monitor, 再将返回值当做参数运行用户界面主循环函数 ui\_mainloop()。

```
#include<stdio.h>
int init_monitor(int, char *[]);
void ui_mainloop(int);

int main(int argc, char *argv[]) {
    /* Initialize the monitor. */
    int is_batch_mode = init_monitor(argc, argv);

    /* Receive commands from user. */
    ui_mainloop(is_batch_mode);

    return 0;
}
```

## Nemu 启动函数 init\_monitor()

- 1) **reg\_test()**函数会生成一些随机的数据对寄存器实现的正确性进行测试。
- 2) **load\_img()**函数读入带有客户程序的镜像文件到一个固定的内存位置 0x100000, 这个程序是运行 NEMU 的一个参数, 在运行 NEMU 的命令中指定, 缺省时将把上文提到的 mov 程序作为客户程序。
- 3) **init\_difftest()**在 PA2 中实现, 用于对照。
- 4) **restart()**函数模拟了计算机启动的功能, 进行一些和计算机启动相关的初始化工作, 其中一个重要的工作就是将%eip 的初值设置为 0x100000。
- 5) **welcome()**函数输出欢迎信息和 NEMU 的编译时间。
- 6) monitor 的初始化工作结束, NEMU 进入用户界面主循环 ui\_mainloop(), 输出 NEMU 的命令提示符:(nemu)

```

int init_monitor(int argc, char *argv[]) {
    /* Perform some global initialization. */

    /* Parse arguments. */
    parse_args(argc, argv);

    /* Open the log file. */
    init_log();

    /* Test the implementation of the `CPU_state' structure. */
    reg_test();

#ifdef DIFF_TEST
    /* Fork a child process to perform differential testing. */
    init_difftest();
#endif

    /* Load the image to memory. */
    load_img();

    /* Initialize this virtual computer system. */
    restart();

    /* Compile the regular expressions. */
    init_regex();

    /* Initialize the watchpoint pool. */
    init_wp_pool();

    /* Initialize devices. */
    init_device();

    /* Display welcome message. */
    welcome();

    return is_batch_mode;
}

```

```

[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU successfully
y
[src/monitor/monitor.c,67,load_img] The image is /home/pqq/桌面/PA/ics2018/nanos
-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,32,welcome] Build time: 09:41:59, Jun  3 2019
For help, type "help"
(nemu) c

```

## 用户界面主循环函数 ui\_mainloop()

等待用户输入调试命令，并进行相应的分割，然后通过与 cmd\_table 中的元素逐个对比，执行用户希望执行的相关操作。已实现好的调试命令为 c、help、q，还需要实现更多功能。

```

// 用户界面主循环
void ui_mainloop(int is_batch_mode) {
    if (is_batch_mode) {
        cmd_c(NULL);
        return;
    }
    while (1) {
        char *str = rl_gets();
        char *str_end = str + strlen(str);
        /* extract the first token as the command */
        char *cmd = strtok(str, " ");
        if (cmd == NULL) { continue; }

        /* treat the remaining string as the arguments,
         * which may need further parsing
         */
        char *args = cmd + strlen(cmd) + 1;
        if (args >= str_end) {
            args = NULL;
        }

#ifdef HAS_IOE
        extern void sdl_clear_event_queue(void);
        sdl_clear_event_queue();
#endif

        int i;
        for (i = 0; i < NR_CMD; i++) {
            if (strcmp(cmd, cmd_table[i].name) == 0) {
                if (cmd_table[i].handler(args) < 0) { return; }
                break;
            }
        }

        if (i == NR_CMD) { printf("Unknown command '%s'\n", cmd); }
    }
}

```

# 简易调试器

命令	格式	使用举例	说明
帮助(1)	help	help	打印命令的帮助信息
继续运行(1)	c	c	继续运行被暂停的程序
退出(1)	q	q	退出 NEMU
单步执行	si [N]	si 10	让程序单步执行 N 条指令后暂停执行， 当 N 没有给出时，缺省为 1
打印程序状态	info SUBCMD	info r info w	打印寄存器状态 打印监视点信息
表达式求值	p EXPR	p \$eax + 1	求出表达式 EXPR 的值，EXPR 支持的 运算请见调试中的表达式求值小节
扫描内存(2)	x N EXPR	x 10 \$esp	求出表达式 EXPR 的值，将结果作为起始内存 地址，以十六进制形式输出连续的 N 个 4 字节
设置监视点	w EXPR	w *0x2000	当表达式 EXPR 的值发生变化时，暂停程序执行
删除监视点	d N	d 2	删除序号为 N 的监视点

## 实现流程

a) 在 cmd\_table 里添加新元素，包括命令、描述、操作函数的名字，如下图

```
static struct {
    char *name;
    char *description;
    int (*handler) (char *);
} cmd_table [] = {
    { "help", "Display informations about all supported commands", cmd_help },
    { "c", "Continue the execution of the program", cmd_c },
    { "q", "Exit NEMU", cmd_q },
    { "si", "Run 1 command of the program", cmd_si },
    { "info", "Print Info SUBCMD", cmd_info },
    { "p", "calculate the EXPR's value", cmd_p },
    { "x", "scan the memory", cmd_x },
    { "w", "set the watchpoint", cmd_w },
    { "d", "delete the watchpoint", cmd_d },
    /* TODO: Add more commands */
};
```

b) 在相同文件下实现对应的函数。

```
static int cmd_help(char *args);
static int cmd_c(char *args) {
}

static int cmd_q(char *args) {
    return -1;
}

static int cmd_si(char *args) {
}
extern void print_wp();
static int cmd_info(char *args){
}

extern uint32_t expr(char *e, bool *success);
static int cmd_p(char *args){
}
static int cmd_x(char *args){
}
static int cmd_w(char *args){
}
extern void free_wp(char* args);
static int cmd_d(char *args){
}
```

## 表达式求值 EXPR 相关知识

### 步骤简述 (expr()函数):

- 1) 识别 token
- 2) 递归求值

```
uint32_t expr(char *e, bool *success) {  
    /* (1) 识别token */  
    if (!make_token(e)) {  
        *success = false;  
        printf("make_token wrong\n");  
        return 0;  
    }  
  
    // 遍历tokens, 若遇到*判断是乘号还是指针符号  
    for(int i=0; i<nr_token; i++)  
        if(tokens[i].str[0]=='*' &&  
            (i==0 || (tokens[i-1].type!=TK_NUM && tokens[i-1].type!=TK_RIGHTBAR)))  
            tokens[i].type=TK_DEREF;  
  
    /* (2) 表达式求值 */  
    uint32_t result = evaluate(0, nr_token-1);  
  
    memset(&tokens, 0, sizeof(Token)*32);  
  
    return result;  
}
```

### 词法分析 (make\_token()函数): 正则表达式。

- 1) PA 实验中出现的 token 类型有:  
 十进制整数  
 +, -, \*, /  
 (, )  
 空格串(一个或多个空格)
- 2) 实现简述:
  - a) 使用 Token 结构体来记录 token 的信息

```
typedef struct token {  
    int type;  
    char str[32];  
} Token;  
  
Token tokens[32]; // 已识别出的token信息  
int nr_token=0; // 已识别出的token数目
```

- b) 用 position 变量来指示当前处理到的位置, 并且按顺序尝试用不同的规则 (rule) 来匹配当前位置的字符串。

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    /* TODO: Add more rules.
     * Pay attention to the precedence level of different rules.
     */
    {" +", TK_NOTYPE},      // spaces 1

    {"==", TK_EQ},         // equal== 2
    {"\\+", TK_PLUS},      // plus+ 3
    {"\\-", TK_MINUS},     // 减号- 4

    {"\\*", TK_MUL},       // 乘号* 5
    {"\\/ ", TK_DIV},      // 除号/ 6

    {"\\*", TK_DEREF},     // 指针符号* 7

    {"0x([0-9a-f])+", TK_HEXNUM}, //十六进制数字 8
    {"([0-9])+", TK_NUM},      //数字 9

    {"\\(\\$([a-z])+", TK_REGISTER}, //寄存器 10

    {"\\(", TK_LEFTBAR},      //左括号 11
    {"\\)", TK_RIGHTBAR},    //右括号 12
};
```

- c) 若一条规则匹配成功，并且匹配出的子串正好是当前待处理串的起始位置时，就成功地识别出一个 token。此时将识别出的 token 信息记录下来(一个例外是空格串)。

## 递归求值 (evaluate())函数

- 1、判断传入的 begin 和 end 值是否合法，如果 begin<end 则 panic

```
if(begin>end)
    panic("bad expression");
```

- 2、判断是否是个 single token，如果是，则判断是 16 进制数、寄存器值还是普通 10 进制数，然后按各自的处理返回。

```
else if(begin==end)//single token should be a number, return it
{
    if(tokens[begin].type==TK_HEXNUM) //16进制
    {
        uint32_t hex_num=0;
        sscanf(tokens[begin].str,"%x",&hex_num);
        return hex_num;
    }
    if(tokens[begin].type==TK_REGISTER) //寄存器
    {
        if(register_name_to_int(begin)==1)
            return atoi(tokens[begin].str);
    }
    else //普通10进制
        return atoi(tokens[begin].str);
}
```

- 3、括号匹配。check\_parentheses()函数用于判断表达式是否被一对匹配的括号包围着，同时检查表达式的左右括号是否匹配，如果不匹配，这个表达式肯定是不符合语法，也就不需要继续进行求值。关于判断是否匹配，我用了 climb 法，合法的括号匹配一定是连续上坡连续下坡且上下坡步数相同的。

```
//括号匹配
else if(check_parentheses(begin,end)==true)
    return evaluate(begin+1,end-1);
```

```

//查看最外层是否是匹配的括号
uint32_t check_parentheses(int begin, int end)
{
    //最左边是左括号，最右边是右括号
    if(tokens[begin].type==TK_LEFTBAR && tokens[end].type==TK_RIGHTBAR)
    {
        int climb=0;
        //用“上下坡”的概念实现判断是否合法
        for(int i=begin;i<end;i++)
        {
            if(tokens[i].type==TK_LEFTBAR)
                climb++;
            else if(tokens[i].type==TK_RIGHTBAR)
                climb--;
            if(i!=end-1 && climb==0)
                return false;
        }
        return true;
    }
    else
        return false;
}

```

- 4、普通运算。目标是找到当前式子中优先级最低的符号，然后根据找到的符号把运算一分为二，递归左右两个式子，递归调用继续求值。这里用 position 记录当前认为优先级最低的符合的位置，flag 用于判断是否在一对括号里，循环的过程中+-比\*/的优先级更低，

```

else//找到优先级最低的符号
{
    int position=0;
    bool flag=0;
    for(int i=begin;i<end;i++)
    {
        if(tokens[i].type==TK_LEFTBAR){
            flag=1;
            continue;
        }
        else if(tokens[i].type==TK_RIGHTBAR){
            flag=0;
            continue;
        }
        if(flag==0 && position==0 && (tokens[i].type==TK_PLUS ||
                                     tokens[i].type==TK_MINOR ||
                                     tokens[i].type==TK_MUL ||
                                     tokens[i].type==TK_DIV) )
            position=i;
        else if (flag==0 &&
                 (tokens[i].type==TK_MUL || tokens[i].type==TK_DIV) &&
                 !(tokens[position].type==TK_PLUS || tokens[position].type==TK_MINOR))
            position=i;
        else if(flag==0 && (tokens[i].type==TK_PLUS || tokens[i].type==TK_MINOR))
            position=i;
    }
    uint32_t val1=evaluate(begin,position-1);
    uint32_t val2=evaluate(position+1,end);
    switch(tokens[position].type)
    {
        case TK_PLUS: return val1+val2;
        case TK_MINOR: return val1-val2;
        case TK_MUL: return val1*val2;
        case TK_DIV: return val1/val2;
        default:assert(0);
    }
}
}

```



## 监视点相关

- 1、 监视点的功能是监视一个表达式的值何时发生变化。
- 2、 在 PA 中我们使用链表结构保存监视点，链表中的元素类型为 struct WP。成员有当前监视点的编号，指向下一个监视点的指针，当前监视点监视的表达式和表达式的值。

```
typedef struct watchpoint {
    int NO;
    struct watchpoint *next;

    /* TODO: Add more members if necessary */
    char expr[32];
    int value;
} WP;
```

- 3、 用“池”的数据结构来管理监视点结构体， head 和 free 指针辅助管理

```
static WP wp_pool[NR_WP];
static WP *head, *free_;
//head用于组织使用中的监视点结构
//free用于组织空闲中的监视点结构
```

- 4、 **Init\_wp\_pool()**函数对池进行初始化。初始化的结果是，池中的每一个节点都有自己的编号，彼此连接，free 指针指向池头，head 指针指向空。

```
//对head和free和wp池初始化
void init_wp_pool() {
    int i;
    for (i = 0; i < NR_WP; i++) {
        wp_pool[i].NO = i;
        wp_pool[i].next = &wp_pool[i + 1];
    }
    wp_pool[NR_WP - 1].next = NULL;

    head = NULL;
    free_ = wp_pool;
}
```

- 5、 **New\_wp()**函数的作用是申请新的监视点。该函数首先判断是否还有空闲监视点，如果有，则从 free 中取出头，使 free 指向 free->next，把取出来的监视点放入 head 中并更新相关信息。

```
WP* new_wp(char* args)
{
    if(free_==NULL)
    {
        assert(0);
        return NULL;
    }
    else
    {
        bool success=0;
        //从free_中取出一个监视点temp，并放到head中表示已被占用
        WP *temp=free_;
        free_=free_->next;

        if(head!=NULL)
            temp->next=head;
        else
            temp->next=NULL;
        head=&wp_pool[temp->NO];

        strcpy(head->expr,args);
        head->value=expr(args,&success);
        return temp;
    }
}
```

- 6、 **Free\_wp()**函数的作用是释放目标监视点。操作类似于链表删除节点，然后把从 head 开头的链表中拿出来的节点放回到 free 中。

```
void free_wp(char* args)
{
    WP *test=head;
    bool flag=0;
    int no = atoi(args);
    //查看第一个节点是不是目标节点
    if(test->NO == no)
    {
        head=head->next;
        test->next=free_;
        free_=test;
        return;
    }
    //第一个节点不是目标节点
    while(test->next!=NULL)
    {
        if(test->next->NO == no)
        {
            flag=1;
            break;
        }
        test=test->next;
    }
    if(flag==0)
    {
        printf("without this watchpoint, delete unsuccessfully\n");
        return ;
    }
    else
    {
        WP *temp=test->next;
        test->next=temp->next;
        temp->next=free_;
        free_=temp;
        printf("successfully delete watchpoint NO=%d\n", temp->NO);
    }
}
```

- 7、 **Print\_wp()**函数的作用是打印当前监视点。即从 head 开始遍历至 NULL，一直打印信息即可。

```
void print_wp()
{
    WP *test=head;
    if(test==NULL)
        printf("without watchpoint\n");
    else
    {
        while(test!=NULL)
        {
            printf("NO=%d\texpr=%s\tvalue=%d\n",test->NO,test->expr,test->value);
            test=test->next;
        }
    }
}
```

- 8、 **Check\_wp()**函数的作用是在程序运行的每一步检查 head 链表中是否有监视点的 value 发生变化，如果有，则打印相关信息，并返回 1，使程序暂停运行，否则返回 0。

```
bool check_wp()
{
    bool result=0;
    WP* test=head;
    while(test!=NULL)
    {
        bool success=0;
        uint32_t temp_value=expr(test->expr, &success);
        if(temp_value!=test->value)
        {
            result=1;
            printf("meet watchpoint %d whoes expr = %s\n",test->NO,test->expr);
            //更新这个wp的value
            test->value=temp_value;
        }
        test=test->next;
    }
    return result;
}
```

9、最后不要忘了在.h 文件中添加以上 4 个新实现的函数。

```
#ifndef WATCHPOINT_H
#define WATCHPOINT_H

#include "common.h"

typedef struct watchpoint {
    int NO;
    struct watchpoint *next;

    /* TODO: Add more members if necessary */
    char expr[32];
    int value;
} WP;

WP* new_wp(char *args);
bool check_wp();
void print_wp();
void free_wp(char *args);
#endif
```

## PA1 实现过程

### 1、 CPU\_state

使用 Union 正确地实现用于模拟寄存器的结构体 CPU\_state。

关于 Union：

- a) union 中可以定义多个成员，union 的大小由最大的成员的大小决定。
- b) union 成员共享同一块大小的内存，一次只能使用其中的一个成员。
- c) 对某一个成员赋值，会覆盖其他成员的值（也不奇怪，因为他们共享一块内存。但前提是成员所占字节数相同，当成员所占字节数不同时只会覆盖相应字节上的值，比如对 char 成员赋值就不会把整个 int 成员覆盖掉，因为 char 只占一个字节，而 int 占四个字节）
- d) 联合体 union 的存放顺序是所有成员都从低地址开始存放的。

```
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;
    };
};
vaddr_t eip;
```

### 2、 单步调试 si[N]

- 1、 在 cmd\_table 中添加 si

```
static struct {
    char *name;
    char *description;
    int (*handler) (char *);
} cmd_table [] = {
    { "help", "Display informations about all supported commands", cmd_help },
    { "c", "Continue the execution of the program", cmd_c },
    { "q", "Exit NEMU", cmd_q },
    { "si", "Run 1 command of the program", cmd_si },
    { "info", "Print info SUBCMD", cmd_info },
    { "p", "calculate the EXPR's value", cmd_p },
    { "x", "scan the memory", cmd_x },
    { "w", "set the watchpoint", cmd_w },
    { "d", "delete the watchpoint", cmd_d },
    /* TODO: Add more commands */
};
```

- 2、实现 cmd\_si: 模仿 cmd\_c 中使用 cpu\_exec 函数, 如果命令是 si 则参数为 1, 否则传入参数的 int 值。

```
static int cmd_si(char *args) {
    if(args==NULL) //si
        cpu_exec(1);
    else //si N
    {
        int value=atoi(strtok(NULL, " "));
        cpu_exec(value);
    }
    return 0;
}
```

- 3、效果展示

```
(nemu) si
100000: bd 00 00 00 00          movl $0x0,%ebp
(nemu) si 5
100005: bc 00 7c 00 00          movl $0x7c00,%esp
10000a: e8 29 00 00 00          call 100038
100038: 55                      pushl %ebp
100039: 89 e5                   movl %esp,%ebp
10003b: 56                      pushl %esi
```

### 3、 打印寄存器 info r

- 1、在 cmd\_table 中添加 info

```
static struct {
    char *name;
    char *description;
    int (*handler) (char *);
} cmd_table [] = {
    { "help", "Display informations about all supported commands", cmd_help },
    { "c", "Continue the execution of the program", cmd_c },
    { "q", "Exit NEMU", cmd_q },
    { "si", "Run 1 command of the program", cmd_si },
    { "info", "Print Info SUBCMD", cmd_info },
    { "p", "calculate the EXPR's value", cmd_p },
    { "x", "scan the memory", cmd_x },
    { "w", "set the watchpoint", cmd_w },
    { "d", "delete the watchpoint", cmd_d },
    /* TODO: Add more commands */
};
```

- 2、实现 cmd\_info, 目的是键入 info r 的时候打印所有寄存器的名字及相关的值

```
static int cmd_info(char *args){
    if(strcmp(args,"r")==0) //info r
    {
        for(int i=0;i<8;i++)
        {
            printf("%s\t",reg_name(i,4));
            printf("0x%x\n",reg_l(i));
        }
        for(int i=0;i<8;i++)
        {
            printf("%s\t",reg_name(i,2));
            printf("0x%x\n",reg_w(i));
        }
        for(int i=0;i<8;i++)
        {
            printf("%s\t",reg_name(i,1));
            printf("0x%x\n",reg_b(i));
        }
    }

    else if(strcmp(args,"w")==0)//info w
    {
        print_wp();
    }

    return 0;
}
```

### 3、效果展示

```
(nemu) info r
eax    0x7d192ad6
ecx    0x60c190a3
edx    0x45d107fe
ebx    0x20775bb9
esp    0x7bf4
ebp    0x7bf8
esi    0x32d84323
edi    0x2d2e077b
ax     0x2ad6
cx     0x90a3
dx     0x7fe
bx     0x5bb9
sp     0x7bf4
bp     0x7bf8
si     0x4323
di     0x77b
al     0xd6
cl     0xa3
dl     0xfe
bl     0xb9
ah     0x2a
ch     0x90
dh     0x7
bh     0x5b
```

## 4、 算数表达式 EXPR 求值 (p EXPR)

具体分析见上节，此处仅列出需要实现的代码

- 1、为算术表达式中的各种 token 类型添加规则，注意 C 语言字符串中转义字符的存在和正则表达式中元字符的功能。

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    /* TODO: Add more rules.
     * Pay attention to the precedence level of different rules.
     */
    {" +", TK_NOTYPE},      // spaces 1

    {"==", TK_EQ},         // equal== 2
    {"\\+", TK_PLUS},      // plus+ 3
    {"\\-", TK_MINUS},     // 减号- 4

    {"\\*", TK_MUL},       // 乘号* 5
    {"\\/ ", TK_DIV},      // 除号/ 6

    {"\\*", TK_DEREF},     // 指针符号* 7

    {"0x([0-9a-f])+", TK_HEXNUM}, //十六进制数字 8
    {"([0-9])+", TK_NUM},    //数字 9

    {"\\(\\$([a-z])+\\)", TK_REGISTER}, //寄存器 10

    {"\\(", TK_LEFTBAR},    //左括号 11
    {"\\)", TK_RIGHTBAR},   //右括号 12
};
```

- 2、在 make\_token 函数中将识别出的 token 信息记录下来(一个例外是空格串), 使用 Token 结构体来记录 token 的信息

```
static bool make_token(char *e) {
    int position = 0;
    int i;
    regmatch_t pmatch;
    nr_token = 0;

    while (e[position] != '\0') {
        /* Try all rules one by one. */
        for (i = 0; i < NR_REGEX; i++) {
            if (regexexec(&re[i], e + position, 1, &pmatch, 0) == 0 && pmatch.rm_so == 0) {
                char *substr_start = e + position;
                int substr_len = pmatch.rm_eo;
                /*Log("match rules[%d] = \"%s\" at position %d with len %d: %.*s",
                 i, rules[i].regex, position, substr_len, substr_len, substr_start);*/
                position += substr_len;

                /* TODO: Now a new token is recognized with rules[i]. Add codes
                 * to record the token in the array 'tokens'. For certain types
                 * of tokens, some extra actions should be performed.
                 */
                tokens[nr_token].type = rules[i].token_type;
                for (int i = 0; i < substr_len; i++)
                    tokens[nr_token].str[i] = (substr_start + i);
                nr_token++;

                break;
            }
        }
        if (i == NR_REGEX) {
            printf("no match at position %d\n%s\n%.*s^\n", position, e, position, "");
            return false;
        }
        return true;
    }
}
```

- 3、在 cmd\_table 添加 p 命令

```
static struct {
    char *name;
    char *description;
    int (*handler) (char *);
} cmd_table [] = {
    {"help", "Display informations about all supported commands", cmd_help },
    {"c", "Continue the execution of the program", cmd_c },
    {"q", "Exit NEMU", cmd_q },
    {"si", "Run 1 command of the program", cmd_si},
    {"info", "Print Info SURCMD", cmd_info},
    {"p", "calculate the EXPR's value", cmd_p},
    {"x", "scan the memory", cmd_x},
    {"w", "set the watchpoint", cmd_w},
    {"d", "delete the watchpoint", cmd_d},
    /* TODO: Add more commands */
};
```

- 4、实现 cmd\_p()函数。调用上节中实现的 expr()函数求值即可。

```
static int cmd_p(char *args){
    bool success;
    uint32_t result = expr(args, &success);
    printf("result=%d\n", result);
    return success;
}
```

5、效果展示

```
(nemu) p (1+2)*3+4/2
result=11
(nemu) p (1+2*3)
result=7
(nemu) p (1+2
bad expression
```

## 5、 实现更复杂的表达式 EXPR 计算

区分\*号是乘号还是指针取值。即在 expr 中进入 evaluate()之前先做一次循环判断。

```
uint32_t expr(char *e, bool *success) {
    /* (1) 识别token */
    if (!make_token(e)) {
        *success = false;
        printf("make_token wrong\n");
        return 0;
    }

    // 遍历tokens, 若遇到*判断是乘号还是指针符号
    for(int i=0; i<nr_token; i++)
        if(tokens[i].str[0]=='*' &&
            (i==0 || (tokens[i-1].type!=TK_NUM && tokens[i-1].type!=TK_RIGHTBAR)))
            tokens[i].type=TK_DEREF;

    /* (2) 表达式求值 */
    uint32_t result = evaluate(0, nr_token-1);

    memset(&tokens, 0, sizeof(Token)*32);

    return result;
}
```

## 6、 打印内存 (x N EXPR)

1、 在 cmd\_table 中添加 x

```
static struct {
    char *name;
    char *description;
    int (*handler) (char *);
} cmd_table [] = {
    { "help", "Display informations about all supported commands", cmd_help },
    { "c", "Continue the execution of the program", cmd_c },
    { "q", "Exit NEMU", cmd_q },
    { "si", "Run 1 command of the program", cmd_si },
    { "info", "Print Info SUBCMD", cmd_info },
    { "n", "calculate the EXPR's value", cmd_p },
    { "x", "scan the memory", cmd_x },
    { "w", "set the watchpoint", cmd_w },
    { "d", "delete the watchpoint", cmd_d },
    /* TODO: Add more commands */
};
```

2、 实现 cmd\_x()函数。实现过程利用了上节实现的 expr()函数对表达式进行求值，同时利用 vaddr\_read 函数访问虚拟地址。

```
static int cmd_x(char *args){
    char *len=strtok(args," ");
    char *addr=strtok(NULL," ");

    uint32_t length=0;
    sscanf(len,"%u",&length);

    bool success;
    uint32_t address = expr(addr,&success);

    for(int i=0;i<length;i++)
    {
        char copy_addr[20];
        memset(&copy_addr,0,20);
        sprintf(copy_addr,"%x",address);
        printf("%s\t",copy_addr);
        printf("%x\n", vaddr_read(address,4));
        address+=4;
    }
    return 0;
}
```

### 3、效果展示

```
(nemu) x 10 0x100000
100000 bd
100004 7c00bc00
100008 29e80000
10000c 90000000
100010 8be58955
100014 fdba084d
100018 90000003
10001c 7420a8ec
100020 3f8bafb
100024 c8880000
```

## 7、 设置监视点 (w EXPR)

### 1、 在 cmd\_table 中添加 w

```
static struct {
    char *name;
    char *description;
    int (*handler) (char *);
} cmd_table [] = {
    { "help", "Display informations about all supported commands", cmd_help },
    { "c", "Continue the execution of the program", cmd_c },
    { "q", "Exit NEMU", cmd_q },
    { "si", "Run 1 command of the program", cmd_si },
    { "info", "Print Info SUBCMD", cmd_info },
    { "p", "calculate the EXPR's value", cmd_p },
    { "x", "scan the memory", cmd_x },
    { "w", "set the watchpoint", cmd_w },
    { "d", "delete the watchpoint", cmd_d },
    /* TODO: Add more commands */
};
```

### 2、 实现 cmd\_w()函数，函数中调用一个 new\_wp 函数来设置监视点。

```
static int cmd_w(char *args){
    WP* wp = new_wp(args);
    printf("successfully got new point %d\n",wp->NO);
    return 0;
}
```

### 3、 根据需求，为 watchpoint 结构加入一些必要成员，包括监视的表达式的结构和表达式的值

```
typedef struct watchpoint {
    int NO;
    struct watchpoint *next;

    /* TODO: Add more members if necessary */
    char expr[32];
    int value;
} WP;
```



- 4、实现 new\_wp 函数，具体实现见上节。并且注意在.h 文件中添加函数声明。
- 5、效果演示。

```
(nemu) w 0x100000
successfully got new point 0
(nemu) w 0x100000+5
successfully got new point 1
```

## 8、 删除监视点 (d N)

- 1、在 cmd\_table 中添加 d

```
static struct {
    char *name;
    char *description;
    int (*handler) (char *);
} cmd_table [] = {
    { "help", "Display informations about all supported commands", cmd_help },
    { "c", "Continue the execution of the program", cmd_c },
    { "q", "Exit NEMU", cmd_q },
    { "si", "Run 1 command of the program", cmd_si },
    { "info", "Print Info SUBCMD", cmd_info },
    { "p", "calculate the EXPR's value", cmd_p },
    { "x", "scan the memory", cmd_x },
    { "w", "set the watchpoint", cmd_w },
    { "d", "delete the watchpoint", cmd_d },
};
/* TODO: Add more commands */
```

- 2、实现 cmd\_d 函数

```
extern void free_wp(char* args);
static int cmd_d(char *args){
    free_wp(args);
    return 0;
}
```

- 3、实现 free\_wp 函数，具体实现见上节，并且注意在.h 文件中添加函数声明。
- 4、效果演示。

```
(nemu) w 0x100000
successfully got new point 0
(nemu) info w
NO=0    expr=0x100000    value=1048576
(nemu) d 0
successfully delete watchpoint NO=0
(nemu) info w
without watchpoint
```

## 9、 打印监视点 (info w)

- 1、在 cmd\_info 函数中添加对 w 的支持，然后直接调用上节写好的 print\_wp 函数即可

```

extern void print_wp();
static int cmd_info(char *args){
    if(strcmp(args,"r")==0) //info r
    {
        for(int i=0;i<8;i++){
            printf("%s\t",reg_name(i,4));
            printf("0x%x\n",reg_l(i));
        }
        for(int i=0;i<8;i++){
            printf("%s\t",reg_name(i,2));
            printf("0x%x\n",reg_w(i));
        }
        for(int i=0;i<8;i++){
            printf("%s\t",reg_name(i,1));
            printf("0x%x\n",reg_b(i));
        }
    }
    else if(strcmp(args,"w")==0) //info w
    {
        print_wp();
    }
    return 0;
}

```

## 2、效果展示

```

(nemu) w 0x100000
successfully got new point 0
(nemu) w 0x100005
successfully got new point 1
(nemu) w 0x100010
successfully got new point 2
(nemu) info w
NO=2    expr=0x100010    value=1048592
NO=1    expr=0x100005    value=1048581
NO=0    expr=0x100000    value=1048576

```

## 10、 实现监视点变化则停止程序运行的效果

- 1、 每当 cpu\_exec()执行完一条指令，就对所有待监视的表达式进行求值，比较它们的值有没有发生变化。即上节提到的 check\_wp()的功能。
- 2、 若发生了变化，程序就因触发了监视点而暂停下来，需要将 nemu\_state 变量设置为 NEMU\_STOP 来达到暂停的效果。最后输出一句话提示用户触发了监视点，并返回到 ui\_mainloop()循环中等待用户的命令。

```

/* Simulate how the CPU works. */
void cpu_exec(uint64 t n) {
    if (nemu_state == NEMU_END) {
        printf("Program execution has ended. To restart the program, exit NEMU and run again.\n");
        return;
    }
    nemu_state = NEMU_RUNNING;

    bool print_flag = n < MAX_INSTR_TO_PRINT;

    for (; n > 0; n --) {
        /* Execute one instruction, including instruction fetch,
         * instruction decode, and the actual execution. */
        exec_wrapper(print_flag);

#ifdef DEBUG
        /* TODO: check watchpoints here. */
        if (check_wp() == 1)
        {
            nemu_state = NEMU_STOP;
        }
    }
#endif

#ifdef HAS_IOE
    extern void device_update();
    device_update();
#endif

    if (nemu_state != NEMU_RUNNING) { return; }

    if (nemu_state == NEMU_RUNNING) { nemu_state = NEMU_STOP; }
}

```

### 3、效果展示

```
(nemu) w ($ebx)
successfully got new point 0
(nemu) info w
N0=0      expr=($ebx)      value=31736
(nemu) si 10
meet watchpoint 0 whoes expr = ($ebx)
^
```

# 回答问题

1、

**初识虚拟化**

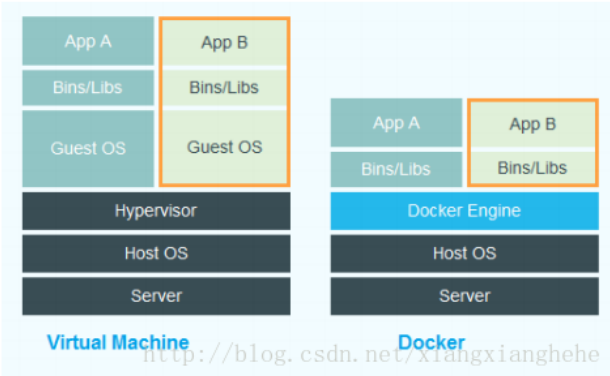
假设你在 Windows 中使用 Docker 安装了一个 GNU/Linux container, 然后在 container 中完成 PA, 通过 NEMU 运行 Hello World 程序. 在这样的情况下, 尝试画出相应的层次图.

嗯, 事实上在 Windows 中运行 Docker container 的真实情况有点复杂, 有兴趣的同学可以查找学习虚拟机和 container 的区别.

1) 层次图:

用户程序	Hello.c/ 马里奥/仙剑奇侠传
操作系统 3	Nanos-lite
硬件 3	NEMU
操作系统 2	GNU/Linux container
硬件 2	Docker
操作系统 1	Windows 操作系统
硬件 1	本机硬件

2) 虚拟机和 container 的区别:  
容器和虚拟机之间的主要区别在于虚拟化层的位置和操作系统资源的使用方式。



对比项	Docker	对比结果	虚拟化
快速创建、删除	启动应用	>>	启动Guest OS+启动应用
交付、部署	容器镜像	==	虚拟机镜像
密度	单Node 100~1000	>>	但Node 10~100
更新管理	迭代式更新, 修改Dockerfile, 对增量内容进行分发、存储、传输、节点启动和恢复迅速	>>	向虚拟机推送安装、升级应用软件补丁包
Windows的支持	不支持	<<	支持
稳定性	每月更新一个版本	<<	KVM, Xen, VMware都已很稳定
安全性	Docker具有宿主机root权限	<<	硬件隔离: Guest OS运行在非根模式
监控成熟度	还在发展过程中	<<	Host, Hypervisor, VM的监控工具在生产环境已使用多年
高可用性	通过业务本身的高可用性来保证	<<	武器库很丰富: 快照, 克隆, HA, 动态迁移, 异地容灾, 异地双活
管理平台成熟度	以k8s为代表, 还在快速发展过程中	<<	以OpenStack, vCenter, 汉柏OPV-Suite为代表, 已经在生产环境使用多年

2、

### 究竟要执行多久?

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?

进入 `cpu_exec()` 查看代码

```
void cpu_exec(uint64_t n) {
    if (nemu_state == NEMU_END) {
        printf("Program execution has ended. To restart the program, exit NEMU and run again.\n");
        return;
    }
    nemu_state = NEMU_RUNNING;

    bool print_flag = n < MAX_INSTR_TO_PRINT;

    for (; n > 0; n --) {
        /* Execute one instruction, including instruction fetch,
         * instruction decode, and the actual execution. */
        exec_wrapper(print_flag);
    }

#ifdef DEBUG
    /* TODO: check watchpoints here. */
    if (check_wp() == 1)
        nemu_state = NEMU_STOP;
#endif

#ifdef HAS_IOE
    extern void device_update();
    device_update();
#endif

    if (nemu_state != NEMU_RUNNING) { return; }

    if (nemu_state == NEMU_RUNNING) { nemu_state = NEMU_STOP; }
}
```

在执行  $n$  步代码的循环中,  $n$  是无符号整型, 故-1 表示最大的数。所以 for 循环可以执行最大次数的循环, 而 `exec_wrapper()` 函数就是执行 `%eip` 指向的当前指令并更新 `%eip`。最终就可以执行完所有指令。

3、

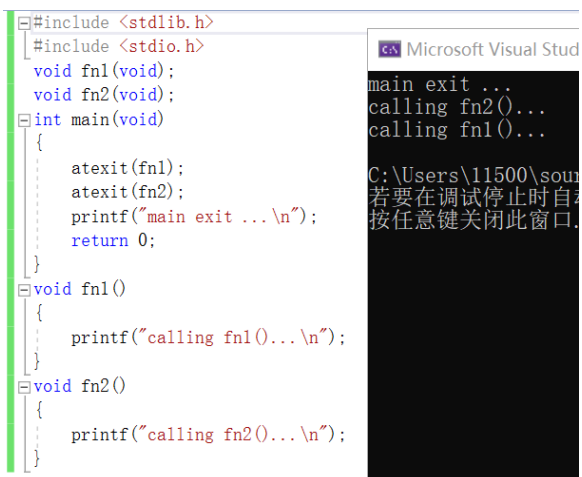
### 谁来指示程序的结束?

在程序设计课上老师告诉你, 当程序执行到 `main()` 函数返回处的时候, 程序就退出了, 你对此深信不疑。但你是否怀疑过, 凭什么程序执行到 `main()` 函数的返回处就结束了? 如果有人告诉你, 程序设计课上老师的说法是错误的, 你有办法来证明/反驳吗? 如果你对此感兴趣, 请在互联网上搜索相关内容。

有时候需要有一种与程序退出方式无关的方法来进行程序退出时的必要处理, 方法就是用 `atexit()` 函数来注册程序正常终止时要被调用的函数, `atexit()` 函数的参数是一个函数指针, 函数指针指向一个没有参数也没有返回值的函数。

它的原型是 `int atexit(void (*)(void));`

以下程序截图可证明和反驳。



The screenshot shows a C program in Microsoft Visual Studio. The code defines two functions, `fn1()` and `fn2()`, which print messages when called. In the `main` function, `atexit(fn1)` and `atexit(fn2)` are called to register these functions for execution at program termination. The `main` function then prints "main exit ..." and returns 0. The output window shows the execution sequence: "main exit ...", "calling fn2()", and "calling fn1()", demonstrating that the registered functions are called after `main` returns.

```
#include <stdlib.h>
#include <stdio.h>
void fn1(void);
void fn2(void);
int main(void)
{
    atexit(fn1);
    atexit(fn2);
    printf("main exit ...\n");
    return 0;
}
void fn1()
{
    printf("calling fn1()...\n");
}
void fn2()
{
    printf("calling fn2()...\n");
}
```

main exit ...  
calling fn2()...  
calling fn1()...

C:\Users\11500\source\...  
若要在调试停止时自动  
按任意键关闭此窗口。

4、

#### 温故而知新

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?

含义: 全局静态变量。

静态全局变量有以下特点:

- 1) 该变量在全局数据区分配内存;
- 2) 未经初始化的静态全局变量会被程序自动初始化为 0 (自动变量的值是随机的, 除非它被显式初始化);
- 3) 静态全局变量在声明它的整个文件都是可见的, 而在文件之外是不可见的;

原因: 静态全局变量不能被其它文件所用, 其它文件中可以定义相同名字的变量, 不会发生冲突。且使得变量可以被该文件的所有函数当做公共变量共同使用。

5、

#### 一点也不能长?

我们知道 `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, 文章中的断点机制还可以正常工作吗? 为什么?

`int3` 指令长度是一个字节, 换成两个字节之后就不能正常工作了。这种形式是有价值的, 因为它可以被用于代替第一指令字节中的任何一个的断点, 包括另一字节指令, 而无需重写其它代码。

6、

#### 随心所欲"的断点

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。

`GDB` 不能将断点设置在指令的中间或者末尾, 会出错。因为指向指令的指针 `EIP` 总是按规律增长, 只能指向指令头, 无法到达指令中间或指令尾部。

7、

#### NEMU 的前世今生

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?

调试器可以开始一些过程和系统进行调试, 或者自己附加到现有进程。它可以单一步骤通过代码, 设置断点和运行, 检查变量值和堆栈跟踪。

调试器有许多高级功能, 比如执行表达式和函数的调用 `debugged` 进程的地址空间, 并且即使改变的过程的代码在实时观看的效果。

模拟器跟调试器不同的地方在于环境的区别, 实现断点的方式不是软硬件断点是模拟出来的。我们所说的模拟器都是利用软件模拟相关的硬件, 相当于用程序去模拟, 而在硬件上运行的是运行这套模拟系统的程序, 而虚拟机或者说调试器是把系统的指令送到机器里, 用硬件来执行指令的, 所以效率高并且还会快很多。

8、

### 尝试通过目录定位关注的问题

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。

通过搜索目录查找到第五章第 96 页

CHAPTER 5 MEMORY MANAGEMENT .....	91
5.1 SEGMENT TRANSLATION .....	92
5.1.1 Descriptors.....	92
5.1.2 Descriptor Tables.....	94
5.1.3 <i>Selectors</i> .....	96
5.1.4 Segment Registers .....	97
5.2 PAGE TRANSLATION.....	98
5.2.1 Page Frame.....	98
5.2.2 Linear Address.....	98

9、

### 必答题

你需要在实验报告中回答下列问题:

①查阅 i386 手册理解了科学查阅手册的方法之后, 请你尝试在 i386 手册中查阅以下问题所在的位置, 把需要阅读的范围写到你的实验报告里面:

- ✧ EFLAGS 寄存器中的 CF 位是什么意思?
- ✧ ModR/M 字节是什么?
- ✧ `mov` 指令的具体格式是怎么样的?

②shell 命令完成 PA1 的内容之后, `nemu/` 目录下的所有 .c 和 .h 文件总共多少行代码? 你是使用什么命令得到这个结果的? 和框架代码相比, 你在 PA1 中编写了多少行代码? (Hint: 目前 2017 分支中记录的正好是做 PA1 之前的状态, 思考一下应该如何回到“过去”?) 你可以把这条命令写入 `Makefile` 中, 随着实验进度的推进, 你可以很方便地统计工程的代码行数, 例如敲入 `make count` 就会自动运行统计代码行数的命令。再来个难一点的, 除去空行之外, `nemu/` 目录下的所有 .c 和 .h 文件总共多少行代码?

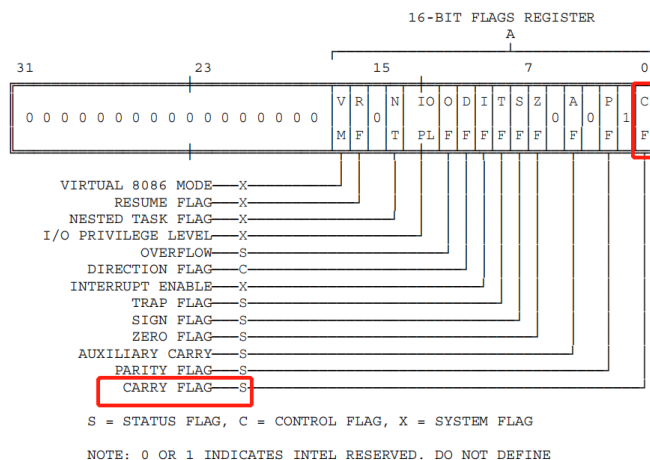
③使用 `man` 打开工程目录下的 `Makefile` 文件, 你会在 `CFLAGS` 变量中看到 `gcc` 的一些编译选项。请解释 `gcc` 中的 `-Wall` 和 `-Werror` 有什么作用? 为什么要使用 `-Wall` 和 `-Werror`?

①

1) 通过查找手册第 34 页知道 CF 是 CARRY FLAG, 百度翻译→进位标志位

### INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986

Figure 2-8. EFLAGS Register



2) 通过查找手册第 240 页知道 ModR/M 字节是“大多数可以在内存中引用操作数的指令在主操作码字节之后都有一个寻址形式的字节”。ModR/M 中 Mod 决定操作数, R/M 表示



Register (or/and) Memory, 跟 mod 一起确定源操作数。

## 17.2 Instruction Format

All instruction encodings are subsets of the general instruction format shown in Figure 17-1. Instructions consist of optional instruction prefixes, one or two primary opcode bytes, possibly an address specifier consisting of the ModR/M byte and the SIB (Scale Index Base) byte, a displacement, if required, and an immediate data field, if required.

Smaller encoding fields can be defined within the primary opcode or opcodes. These fields define the direction of the operation, the size of the displacements, the register encoding, or sign extension; encoding fields vary depending on the class of operation.

Most instructions that can refer to an operand in memory have an addressing form byte following the primary opcode byte(s). This byte, called the **ModR/M** byte, specifies the address form to be used. Certain encodings of the ModR/M byte indicate a second addressing byte, the SIB (Scale Index Base) byte, which follows the ModR/M byte and is required to fully specify the addressing form.

Addressing forms can include a displacement immediately following either the ModR/M or SIB byte. If a displacement is present, it can be 8-, 16- or 32-bits.

If the instruction specifies an immediate operand, the immediate operand always follows any displacement bytes. The immediate operand, if specified, is always the last field of the instruction.

### 3) 通过查找手册第 345 页可知 Mov 指令的具体格式

#### 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,offs8	4	Move byte at (seg:offset) to AL
A1	MOV AX,offs16	4	Move word at (seg:offset) to AX
A1	MOV EAX,offs32	4	Move dword at (seg:offset) to EAX
A2	MOV offs8,AL	2	Move AL to (seg:offset)
A3	MOV offs16,AX	2	Move AX to (seg:offset)
A3	MOV offs32,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
Ciiiiii	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

②

1. 在 nemu 下使用命令 `find . -name "*[.h|.cpp]" | xargs wc -l`

```
106 ./src/monitor/dlfi-test/gdb-host.c
215 ./src/monitor/debug/ui.c
126 ./src/monitor/debug/watchpoint.c
278 ./src/monitor/debug/expr.c
143 ./src/monitor/monitor.c
46 ./src/monitor/cpu-exec.c
3905 总用量
ppq@ppq-virtual-machine:~/桌面/PA/ics2018/nemu$
```

2. 在 nemu 下使用命令 `find . -name "*[.cpp|.h]" | xargs grep "^." | wc -l` , 除去空行共有 3215 行



```

pqq@pqq-virtual-machine:~/桌面/PA/ics2018/nemu$ find . -name "*[.h|.cpp]" | xargs grep "^." | wc -l
grep: .: 是一个目录
grep: ./build/obj/misc: 是一个目录
grep: ./build/obj/cpu/exec: 是一个目录
grep: ./src: 是一个目录
grep: ./src/misc: 是一个目录
grep: ./src/cpu/exec: 是一个目录
3215

```

### 3. 使用以下命令对比

```

pqq@pqq-virtual-machine:~/桌面/PA/ics2018/nemu$ git checkout pa0 && find . -name *.ch | xargs cat | wc -l
切换到分支 'pa0'
3487
pqq@pqq-virtual-machine:~/桌面/PA/ics2018/nemu$ git checkout pa1 && find . -name *.ch | xargs cat | wc -l
切换到分支 'pa1'
3866

```

- ③ -Wall 是打开 gcc 的所有警告, -Werror 要求 gcc 将所有的警告当成错误进行处理。  
 -Wall 作用是提供编译警告, 如果没有警告万一影响了程序运行不就尴尬了。位-Werror 可以防止函数定义未使用, 当定义未使用时, 会报错, 而不是警告, 保证了程序的正确运行。他将程序中所有 warning 都指示成为 error, 防止程序因为 warning 造成程序的不稳定性。

# 遇到的坑

- 1、实现 cmd\_si 时，先尝试实现单步调试，然后顺顺利利地发现调用 cpu\_exec(1)可以有效实现单步调试，于是在实现 si n 时使用了 for 循环进行 n 次 cpu\_exec(1)的方法而不是 cpu\_exec(n)的传参方式。

在刚实现时是没问题的，完全能跑通耶？！

但在实现监视点的时候，epu\_exec 函数进行了如下修改：

```
/* Simulate how the CPU works. */
void cpu_exec(uint64_t n) {
    if (nemu_state == NEMU_END) {
        printf("Program execution has ended. To restart the program, exit NEMU and run again.\n");
        return;
    }
    nemu_state = NEMU_RUNNING;

    bool print_flag = n < MAX_INSTR_TO_PRINT;

    for (; n > 0; n --) {
        /* Execute one instruction, including instruction fetch,
         * instruction decode, and the actual execution. */
        exec_wrapper(print_flag);

#ifdef DEBUG
        /* TODO0: check watchpoints here. */
        if (check_wp() == 1)
        {
            nemu_state = NEMU_STOP;
        }
    }
#endif

#ifdef HAS_IOE
    extern void device_update();
    device_update();
#endif

    if (nemu_state != NEMU_RUNNING) { return; }
}

if (nemu_state == NEMU_RUNNING) { nemu_state = NEMU_STOP; }
```

如果传参一直是 1 进来，在红框中碰到监视点变化被暂停（设置为 NEMU\_STOP），然后在蓝框中被返回。然后外层循环继续调用 cpu\_exec(1)进入函数，并把 nemu\_state 设置成 NEMU\_RUNNING，然后并运行 exec\_wrapper()，则无法起到 nemu\_state 状态调整的效果，也就无法实现监视点功能，程序将一直运行而不会停下。

```
static int cmd_si(char *args) {
    if (args == NULL)
        cpu_exec(1);
    else
    {
        int flag = 1;
        int value = 0;
        int length = strlen(args) - 1;
        for (int i = length; i >= 0; i--)
        {
            value += flag * ((int)args[i] - 48);
            flag *= 10;
        }
        for (int i = 0; i < value; i++)
            cpu_exec(1);
    }
    return 0;
}
```

(错误实现)

```

(nemu) si 2
100000: bd 00 00 00 00          movl $0x0,%ebp
100005: bc 00 7c 00 00          movl $0x7c00,%esp
(nemu) w ($ebx)
successfully got new point 0
(nemu) info w
NO=0   expr=($ebx)   value=31744
(nemu) si 5
10000a: e8 29 00 00 00          call 100038
meet watchpoint 0 whoes expr = ($ebx)
100038: 55                      pushl %ebp
meet watchpoint 0 whoes expr = ($ebx)
100039: 89 e5                   movl %esp,%ebp
10003b: 56                      pushl %esi
meet watchpoint 0 whoes expr = ($ebx)
10003c: 53                      pushl %ebx
meet watchpoint 0 whoes expr = ($ebx)

```

(错误效果)

```

static int cmd_si(char *args) {
    if(args==NULL)
        cpu_exec(1);
    else
    {
        int flag=1;
        int value=0;
        int length=strlen(args)-1;
        for(int i=length;i>=0;i--)
        {
            value+=flag*((int)args[i]-48);
            flag*=10;
        }
        cpu_exec(value);
    }
    return 0;
}

```

(正确实现)

```

Welcome to NEMU!
[src/monitor/monitor.c,32,welcome] Build time: 21:11:54, Jun  4 2019
For help, type "help"
(nemu) si 2
100000: bd 00 00 00 00          movl $0x0,%ebp
100005: bc 00 7c 00 00          movl $0x7c00,%esp
(nemu) w ($ebx)
successfully got new point 0
(nemu) si 5
10000a: e8 29 00 00 00          call 100038
meet watchpoint 0 whoes expr = ($ebx)
(nemu)

```

(正确效果)

2、不知道是在控制台输入 man readline……还去 i386 手册找……

关于 readline() 的功能和返回值等信息, 请查阅 [man readline](#)

```
pqq@pqq-virtual-machine: ~/桌面/test1$ man readline
```

```

pqq@pqq-virtual-machine: ~/桌面/test1
Library Functions Manual
NAME
    readline - get a line from a user with editing
SYNOPSIS
    #include <stdio.h>
    #include <readline/readline.h>
    #include <readline/history.h>

    char *
    readline (const char *prompt);
COPYRIGHT
    Readline is Copyright (C) 1989-2011 Free Software Foundation, Inc.
DESCRIPTION
    readline will read a line from the terminal and return it, using prompt
    as a prompt. If prompt is NULL or the empty string, no prompt is
    issued. The line returned is allocated with malloc(3); the caller must
    free it when finished. The line returned has the final newline
    removed, so only the text of the line remains.
Manual page readline(3readline) line 1 (press h for help or q to quit)

```

# 复习的知识和学到的经验

- 1、编译原理课的词法分析、句法分析相关，复习了正则表达式，练习了递归的使用。也唤醒了一些关于 yacc 和 lex 的惨痛回忆。
- 2、指针和链表。在实现 watchpoint 时熟悉了一下链表的插入删除等写法，对指针的运用有了更深的理解。
- 3、下载了好用的编辑器 sublime text，学会在 Linux 下输入命令行下载软件。
- 4、学习和使用对一些平时自己写代码比较少用的 C++ 保留字，如 extern, static, #ifdef, #endif 之类，以后自己写代码也可以多用了。
- 5、第一次参与这么多文件的代码工程的编写，一开始完全无法入手不知道从哪里开始读代码，然后慢慢适应，整个过程受益匪浅。
- 6、一开始不知道为什么 cpu\_exec() 要传入 -1 参数，然后复习了一波有符号无符号原码补码反码之类的概念。
- 7、复习了寄存器和他们之间的关系以及位数等。