PA1实验报告-1711346-李明旗

目录

1 概述

PA的最终目的是要实现NEMU,一款经过简化的x86全系统模拟器。

1.1 实验目的

(1) 熟悉 GNU/Linux 平台

(2) 初步探究“程序在计算机上运行”的相关原理

(3) 初步学习 GDB 并在 PA上实现简易调试器

1.2 实验内容

PA1 的实验主要为简易调试器的实现。主要分为以下下三个阶段：

第一阶段，模拟寄存器结构，实现调试器基本功能。

第二阶段，实现调试功能的表达式求值，并完善阶段一中的扫描内存函数。

第三阶段，实现调试功能中的监视点，学习断点相关知识与 i386 手册。

2 阶段一

2.1 简单计算机模型

2.11 NEMU执行流程

NEMU 起点函数 nemu/src/main.c，

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) 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。

4) init_regex();/正则式

5) init_wp_pool();//监视点

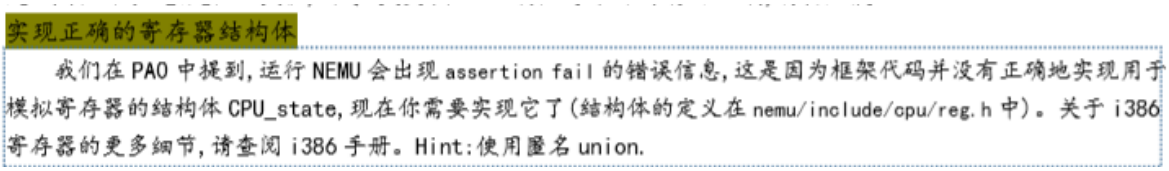
6) init_device();设备的初始化

7) welcome()函数输出欢迎信息和 NEMU 的编译时间。

8) ui_mainloop(is_batch_mode)//用户界面主循环程序，实现交互功能

```
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;
}
```

2.12 代码：实现真正的寄存器结构体



相关文件：nemu/src/cpu/reg.c 与 nemu/include/cpu/reg.h

第一部分，需实现 gpr[8]的 Union 结构。第二部分，要求 eax,ecx,edx 等 32 位寄存器与 gpr[8]开始于同一内存首地址。

```
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;
```

make run后

```
uild-in image.
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 08:54:38, Mar 11 2020
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026
```

2.1.3 问题:究竟要执行多久?

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?

查看代码发现:

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

n是无符号整型，-1表示最大的数。所以for 循环可以执行最大次数的循环，而ecex_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。用 printf("%llu\n",n)来查看 n=-1 时的取值如下：比老师的多输出了几行n的取值。

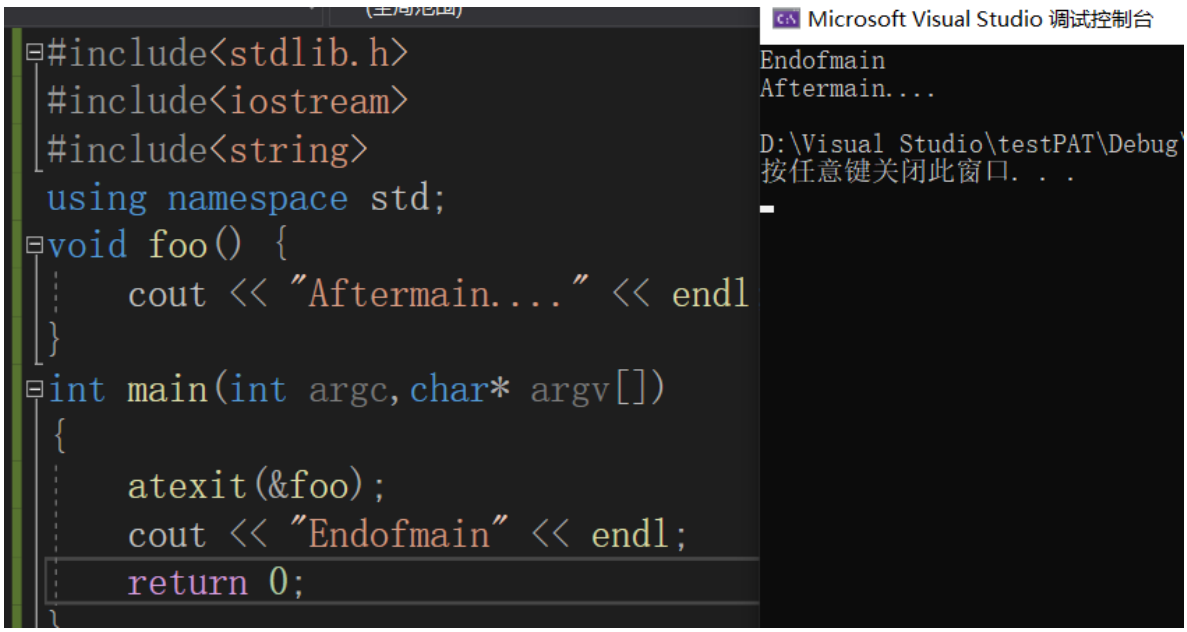
```
For help, type "help"
(nemu) c
18446744073709551615
18446744073709551614
18446744073709551613
18446744073709551612
18446744073709551611
18446744073709551610
18446744073709551609
18446744073709551608
nemu: HIT GOOD TRAP at eip = 0x00100026
```

2.1.4 问题:谁来指示程序的结束?

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.

在main()函数返回后，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。



这个例子就说明了在main()函数返回后，又调用了atexit()函数。

2.2 基础设施：简易调试器

2.2.1 代码：实现单步执行，打印寄存器，扫描内存

可以知道是用 readline 读取我们输入的命令之后，用 strtok()分解第一个字符串（以空格分开），然后与cmd_table[]中的 name 比较，执行对应的函数。

实现单步执行:

在 cmd_table 中添加 si

```
{ "si","Run 1 command of the program", cmd_si},
```

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;
}
```

效果展示:

```
(nemu) si
  100000:   b8 34 12 00 00                  movl $0x1234,%eax
(nemu) si 4
  100005:   b9 27 00 10 00                  movl $0x100027,%ecx
  10000a:   89 01                           movl %eax,(%ecx)
  10000c:   66 c7 41 04 01 00               movw $0x1,0x4(%ecx)
  100012:   bb 02 00 00 00                  movl $0x2,%ebx
(nemu)
```

打印寄存器:

在 cmd_table 中添加 info

```
{ "info", "Print Info SUBCMD", cmd_info},
```

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));
            }
        }
```

效果展示：

```
(nemu) info r
eax     0x51265319
ecx     0x7bb6d168
edx     0x2760746f
ebx     0x5de445b3
esp     0x4e38be58
ebp     0x3111a056
esi     0x5b727c0d
edi     0x3514699f
ax      0x5319
cx      0xd168
dx      0x746f
bx      0x45b3
sp      0xbe58
bp      0xa056
si      0x7c0d
di      0x699f
al      0x19
cl      0x68
dl      0x6f
bl      0xb3
ah      0x53
ch      0xd1
dh      0x74
bh      0x45
(nemu)
```

扫描内存：

在 cmd_table 中添加 x

```
{ "x", "scan the memory", cmd_x},
```

实现 cmd_x()函数，打印 0x100000 附近的内存。

```c
static int cmd_x(char *args){
  vaddr_t start_point;
  int i,time,temp;
  char *arg=strtok(NULL," ");
  char *arg_1=strtok(NULL," ");
  sscanf(arg,"%d",&time);
  sscanf(arg_1,"%x",&start_point);
  printf("address    data\n");
  for(i=1;i<=time;++i){
  printf("%#x    ",start_point);
  temp=vaddr_read(start_point,4);
  printf("%#x\n",temp);
  start_point+=4;
}
  return 0;
}
```

效果展示:

```
(nemu) x 10 0x100000
address    data
0x100000   0x001234b8
0x100004   0x0027b900
0x100008   0x01890010
0x10000c   0x0441c766
0x100010   0x02bb0001
0x100014   0x66000000
0x100018   0x009984c7
0x10001c   0x01ffffe0
0x100020   0x0000b800
0x100024   0x00d60000
(nemu)
```

3 阶段二

3.1 词法分析

3.1.1 代码:实现算术表达式的词法分析

为算术表达式中的各种 token 类型添加规则,

```c
enum {  //从256开始,为了避开ascii
  TK_NOTYPE = 256, TK_HEX, TK_DEC, TK_REG, TK_EQ, TK_NEQ,
  TK_AND, TK_OR,
  TK_NEG,      //-代表负数
  TK_POI,       //指针解引用
  TK_LS, TK_RS, TK_BOE, TK_LOE

  /* TODO: Add more token types */

};
```

```
{" +", TK_NOTYPE},    // spaces
{"0x[0-9A-Fa-f][0-9A-Fa-f]*", TK_HEX},
{"0|[1-9][0-9]*", TK_DEC},
{"\\$(eax|ecx|edx|ebx|esp|ebp|esi|edi|eip|ax|cx|d

{"\\+", '+'},         // 使用单引号
{"-", '-'},
{"\\*", '*'},
{"\\/", '/'},

{"\\(", '('},
{"\\)", ')'},

{"==", TK_EQ},
{"!=", TK_NEQ},

{"&&", TK_AND},
{"\\|\\|", TK_OR},
{"!", '!'},
// 注意前缀问题 >=识别应在>前面
// 类似的 十进制和十六进制位置
{"<<", TK_LS},
{">>", TK_RS},
{">=", TK_BOE},
{">", '>'},
{"<=", TK_LOE},
{"<", '<'}

};
```

在make_token 函数中针对该类型进行处理，copy字符串时除了注意溢出的问题，还有就是别忘了末尾追加结束符.

```
if(rules[i].token_type == TK_NOTYPE) //空格直接舍弃
    break;
if(substr_len>32)  //str溢出 false报错
    assert(0);
memset(tokens[nr_token].str,'\0',32); //以防万一
strncpy(tokens[nr_token].str, substr_start, substr_len);// 类似上面的%.*s
*(tokens[nr_token].str+substr_len)='\0';
tokens[nr_token].type = rules[i].token_type;
// Log("Save in type=%d, str=%s",tokens[nr_token].type,tokens[nr_token].str);
nr_token = nr_token + 1;

break;
```

3.2 表达式求值.

3.2.1 代码:实现算术表达式的递归求值.

递归求值（evaluate()函数），在token表达式中指示一个子表达式,我们可以使用两个整数p和q来指示这个子表达式的开始位置和结束位置。

```c
uint32_t eval(int p,int q){
    if(p>q){   //3+缺省为3+0 --1缺省为0--1
        // printf("Bad expression\n");
        return 0;
        // assert(0);
    }
    else if(p==q){
        uint32_t res;
        if(tokens[p].type == TK_HEX) sscanf(tokens[p].str,"%x",&res);
        else if(tokens[p].type == TK_DEC) sscanf(tokens[p].str,"%d",&res);
        else if(tokens[p].type == TK_REG){
            char tmp[3] = {tokens[p].str[1],tokens[p].str[2],tokens[p].str[3]};
            for(int i=0;i<8;i++)
                if(!strcmp(tmp,regsl[i])){return cpu.gpr[i]._32;}
            for(int i=0;i<8;i++)
                if(!strcmp(tmp,regsw[i])){return cpu.gpr[i]._16;}
            for(int i=0;i<8;i++)
                if(!strcmp(tmp,regsb[i])){return cpu.gpr[i%4]._8[i/4];}
            char teip[3]="eip";
            if(strcmp(tmp,teip))return cpu.eip;
        }
        else assert(0);
        return res;
    }
    else if(check_parentheses(p,q) == true){
        return eval(p+1,q-1);
    }
        uint32_t val1 = eval(p,op-1);
        uint32_t val2 = eval(op+1,q);
        switch(op_type){
            case TK_OR:return val1||val2;
            case TK_AND:return val1&&val2;
            case TK_NEQ:return val1!=val2;
            case TK_EQ:return val1==val2;
            case TK_LOE:return val1<=val2;
            case TK_BOE:return val1>=val2;
            case '<':return val1<val2;
            case '>':return val1>val2;
            case TK_RS:return val1>>val2;
            case TK_LS:return val1<<val2;
            case '+':return val1+val2;
            case '-':return val1-val2;
            case '*':return val1*val2;
            case '/':return val1/val2;
            case '!':return !val2;
            case TK_NEG:return -1*val2;
            case TK_POI:return vaddr_read(val2,4);
            default:assert(0);
        }
    }
```

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。

```
bool check_parentheses(int p,int q){
    if((tokens[p].str[0]=='(') && (tokens[q].str[0]==')')){
        //左括号记为1 右括号记为-1
        //总和应该为0 且遍历完之前总和一定不为0，以确保最左和最右匹配
        int count = 0;
        for(int i=p;i<q;i++) //前n-1个数总和应不为0
        {
            if(tokens[i].str[0] == '('
                count = count + 1;
            if(tokens[i].str[0] == ')')
                count = count - 1;
            if(count == 0)
            {
                //printf("Leftmost and rightmost are not matched\n");
                return false;
            }
        }
        count = count -1; //最后一个右括号
        if(count !=0) //总和应该为0
        {
            printf("Bad parentheses\n");
            assert(0);
        }
        return true;
```

在 cmd_table 添加 p 命令

```
{ "p", "calculate the EXPR's value", cmd_p},
```

实现 cmd_p()函数。调用实现的 expr()函数求值

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

效果展示:

```
For help, type "help"
(nemu) p (1+2)*3-4/2
result=7
(nemu)
```

```
(nemu) p (2+3)*8
result=40
(nemu) p (2+18)/(5-3)-9
result=1
(nemu)
```

3.2.2 代码:实现带有负数的算术表达式的求值

区分-号是减号还是负号。即在 expr 中进入 evaluate()之前先做一次循环判断。

```
if(nr_token!=1)  //只有一个符号时没必要区分
  for(int i=0;i<nr_token;i++)  //负号的判断 当其为第一个符号，或左边为(时,或按照讲义左边可能为负号(--1)
      if(tokens[i].type == '-' &&(i==0||tokens[i-1].type == '('||tokens[i-1].type == TK_NEG
                                                              ||tokens[i-1].type == '-'
                                                              ||tokens[i-1].type == '+'
                                                              ||tokens[i-1].type == '*'
                                                              ||tokens[i-1].type == '/'))
          tokens[i].type = TK_NEG;
```

效果展示:

```
(nemu) p --8
result=8
(nemu) p -8*(2+5)
result=-56
(nemu)
```

3.2.3 代码:实现更复杂的表达式求值

```
switch(tokens[i].type){
    case TK_OR:if(curr_prev>1){curr_prev=1;op=i;op_type=TK_OR;continue;}
    case TK_AND:if(curr_prev>2){curr_prev=2;op=i;op_type=TK_AND;continue;}
    case TK_NEQ:if(curr_prev>3){curr_prev=3;op=i;op_type=TK_NEQ;continue;}
    case TK_EQ:if(curr_prev>3){curr_prev=3;op=i;op_type=TK_EQ;continue;}
    case TK_LOE:if(curr_prev>4){curr_prev=4;op=i;op_type=TK_LOE;continue;}
    case TK_BOE:if(curr_prev>4){curr_prev=4;op=i;op_type=TK_BOE;continue;}
```

和负号一样，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。

```
    for(int i=0;i<nr_token;i++)
        if(tokens[i].type == '*' &&(i==0||(tokens[i-1].type!=TK_DEC && tokens[i-1].type!=TK_H
kens[i-1].type!=')')))
            tokens[i].type = TK_POI;
```

效果展示:

```
(nemu) p 3==3&&(*0x100000==0x1234b8)
result=1
(nemu)
```

3.2.4 代码:完善扫描内存的功能

利用实现的 expr()函数对表达式进行求值

```
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;
```

效果展示:

```
(nemu) x 10 0x100000
100000  1234b8
100004  27b900
100008  1890010
10000c  441c766
100010  2bb0001
100014  66000000
100018  9984c7
10001c  1ffffe0
100020  b800
100024  d60000
(nemu) x 10 $eip
100000  1234b8
100004  27b900
100008  1890010
10000c  441c766
100010  2bb0001
100014  66000000
100018  9984c7
10001c  1ffffe0
100020  b800
100024  d60000
```

4 阶段三

4.1 监视点

4.1.1 代码:实现监视点池的管理

实现监视点池的管理

为了使用监视点池,你需要编写以下两个函数(你可以根据你的需要修改函数的参数和返回值):

```
WP* new_wp();
void free_wp(WP *wp);
```

其中 new_wp()从 free_链表中返回一个空闲的监视点结构,free_wp()将 wp 归还到 free_链表中,这两个函数会作为监视点池的接口被其它函数调用.需要注意的是,调用 new_wp()时可能会出现没有空闲监视点结构的情况,

定义监视点的结构及后续所需的相关函数

```
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
```

init_wp_pool()实现结构初始化

```c
//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;
}
```

实现 new_wp 函数

```c
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
                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;
        }
}
```

实现free_wp 函数

```
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;
                printf("successfully delete watchpoint NO=%d\n", test->NO);
                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);
```

4.1.2 问题: static 的使用

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。

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。

4.1.3 代码:实现监视点

打印监视点信息的辅助函数 print_wp()

```
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;
                }
        }
}
```

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;
}
```

监视点的申请

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

监视点的删除

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

监视点信息查看：在原 cmd_info 函数中进行补充。

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

return 0;
```

cpu_exec()函数中实现监视点的触发判断：

```
/* TODO: check watchpoints here. *,
if(check_wp()){
    nemu_state = NEMU_STOP;
}
```

效果展示：

```
(nemu) w $eip==0x100000
successfully got new point 0
(nemu) info w
NO=0    expr=$eip==0x100000     value=1
(nemu) c
meet watchpoint 0 whoes expr = $eip==0x100000
(nemu) info w
NO=0    expr=$eip==0x100000     value=0
(nemu) d 0
successfully delete watchpoint NO=0
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026
```

4.2 断点

4.2.1 断点的工作原理S

断点通过 CPU 的特殊指令——int3 来实现的，INT3 指令产生一个特殊的单字节操作码（0xCC），当进程执行到该指令时， 将会调用调试异常处理例程 即“traptodebugger”，实现中断。

4.2.2 问题:“一点也不能长?

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。

4.2.3 问题:随心所欲的断点

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.

如果将断点设置在指令的首字节处，在读取指令时，0xcc 将被视作指令的操作码，程序执行到此时，产生 int3 软中断。而设置在指令的非首字节处时，0xcc可能被作为指令的操作数，解析出错误指令，产生意料之外的 错误。

4.2.4 问题: NEMU的前世今生

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?

模拟器是模拟执行特定的硬件平台及其程序的软件程序，而调试器是用于测试和调试其他程序的计算机程序。

我们NEMU 实现了简易调试器，它是通过 ui_mainloop 获取相关命令后，执行对应程序并输出相关信息的。 而在 GDB 中，调试是通过 ptrace 系统调用进行实现的。

4.3 i386手册的学习

4.3.1 问题:通过目录定位关注的问题

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

4.3.2 必答题

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 和文件总共有多少行代码？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 文件总共有多少行代码？

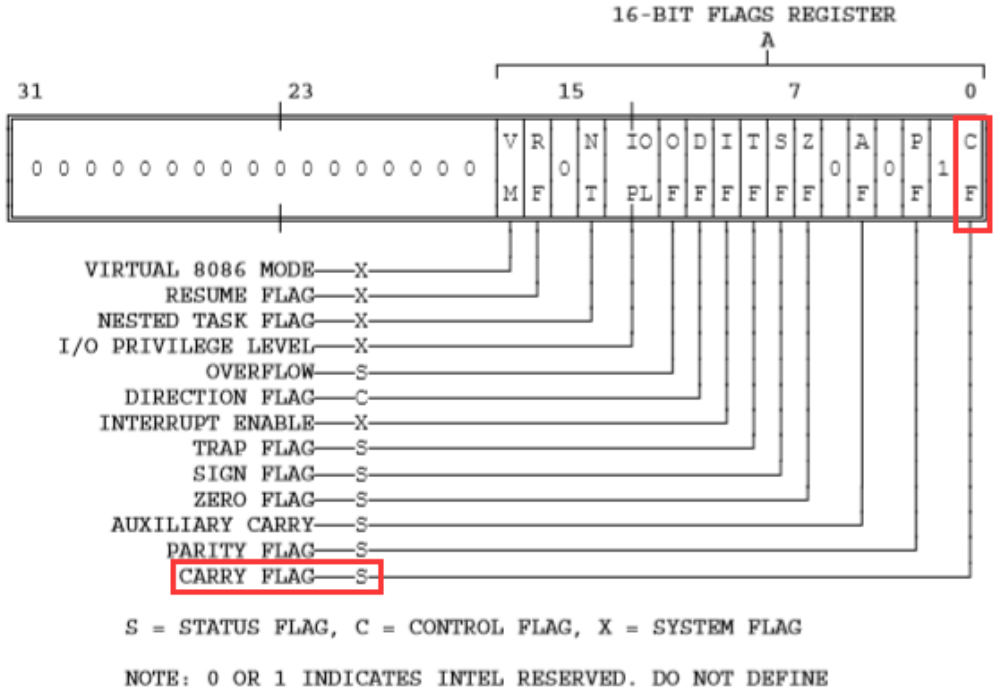
③使用 man 打开工程目录下的 Makefile 文件，你会在 CFLAGS 变量中看到 gcc 的一些编译选项。请解释 gcc 中的 -Wall 和 -Werror 有什么作用？为什么要使用 -Wall 和 -Werror？

①

□ EFLAGS 寄存器中的 CF 位是什么意思？

INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986

Figure 2-8. EFLAGS Register



所以，CF为 FLAGS 寄存器的状态标记，若算术操作的结果在高阶位产生 借位或进位，则置 CF 位为 1，否则为 0。

□ ModR/M 字节是什么？

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.

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) Mermory，跟 mod 一起确定源操作数。

□ mov 指令的具体格式是怎么样的?

通过查找发现MOV指令如下：

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
C6 /0	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 在 nemu 下使用命令 find . -name "*[.h|.cpp]" | xargs wc -l ，PA1中共有3895行代码；

```
qiqi@qiqi-virtual-machine:~/ics2018/nemu$ find . -name "*.[ch]" |xargs cat|wc
-l
3859
qiqi@qiqi-virtual-machine:~/ics2018/nemu$
```

不含空格的统计：find.-name"*.[ch]"|xargscat|grep-v^$|wc–l，得 PA1中 共 3180 行代码。

```
qiqi@qiqi-virtual-machine:~/ics2018/nemu$ find . -name "*.[ch]" |xargs cat|gre
p -v ^$|wc -l
3180
qiqi@qiqi-virtual-machine:~/ics2018/nemu$
```

使用 git branch 可查看分支情况, git checkout[分支]可进行分支切换。

```
qiqi@qiqi-virtual-machine:~/ics2018/nemu$ git branch
  master
  pa0
* pa1
qiqi@qiqi-virtual-machine:~/ics2018/nemu$
```

故要查看 PA1编写了多少代码，只需要：切换至 PA0分支，查看当前代码行 数；再切换回原分支, 查看此时的代码行数。将以上操作写入 makefile：

```
count:
	git checkout pa0
	find . -name "*.[ch]" |xargs cat|wc -l
	git checkout pa1
	find . -name "*.[ch]" |xargs cat|wc -l
                                                    59,1-8        底端
```

输入make count: 在 PA1中编写的代码行数为 3859-3487=372 行。

```
qiqi@qiqi-virtual-machine:~/ics2018/nemu$ make count
git checkout pa0
正在检出文件: 100% (755/755), 完成.
切换到分支 'pa0'
find . -name "*.[ch]" |xargs cat|wc -l
3487
git checkout pa1
切换到分支 'pa1'
find . -name "*.[ch]" |xargs cat|wc -l
3859
qiqi@qiqi-virtual-machine:~/ics2018/nemu$
```

③查看CFLAGS：

```
CFLAGS   += -O2 -MMD -Wall -Werror -ggdb $(INCLUDES)
```

-wall：生成所有警告信息

-werror：在发生警告时停止编译操作，即要求 gcc 将所有警告当成错误进行 处理。

作用是为了提高代码的安全性，便于进行代码维护和 debug

5 实验结论与感想

在进行表达式求值时复习了，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。在实现 watchpoint 时对指针的运用有了更深的理解，还有一开始不知道为什么 cpu_exec() 要传入-1 参数，然后复习了有符号和无符号，原码补码反码之间的关系。

在实现过程中也遇到了奇奇怪怪的问题，比如在make file实现make count 后，在终端输入会报这种错，网上找让git clean,但发现git clean好像禁用了？？ 最后发现make run一下在make count 就好了。。。应该是make run之后才能把修改的makefile文件读入吧!

```
qiqi@qiqi-virtual-machine:~/ics2018/nemu$ make count
git checkout pa0
error: Your local changes to the following files would be overwritten by checkou
t:
        nemu/Makefile
Please, commit your changes or stash them before you can switch branches.
Aborting
Makefile:56: recipe for target 'count' failed
make: *** [count] Error 1
qiqi@qiqi-virtual-machine:~/ics2018/nemu$
```

总之，觉得pa1很有挑战性，不过老师给的指导很详细，但也需要花费大量的时间和耐心！