1 概述

1.1 实验目的

1.熟练掌握GNU/Linux系统的使用

2.提高使用vim编辑器的熟练度

3.理解"图灵机"(Turing Machine)工作原理

4.熟悉NEMU整体框架并充分理解与PA1相关的框架代码

5.完善NEMU寄存器结构体并实现NEMU简易调试器

1.2 实验内容

NEMU主要由4个模块构成，包括monitor、CPU、memory和设备。通过跟随PA实验流程中对NEMU框架代码的阅读指导阅读相应代码，理解了NEMU开始执行时的工作原理、NEMU monitor部分的工作原理和已经实现的帮助（help）、继续运行（c）和退出（q）命令的实现方式。在PA1部分需要实现NEMU的成功启动和NEMU简易调试器的所有命令并进行功能检验。共包括三个阶段：

阶段一：正确实现模拟寄存器和简易调试器单步执行、打印寄存器和扫描内存的功能

阶段二：正确实现简易调试器表达式求值的功能并将阶段一扫描内存的部分完善

阶段三：正确实现简易调试器监视点的功能，学习调试器断点工作原理和i386手册

2 阶段一

在开始所有的编程工作之前，我们需要跟随PA实验流程熟悉NEMU执行时的工作原理，NEMU main()函数中init_monitor()和ui_mainloop()具体功能以及init_monitor()函数内monitor各部分初始化功能说明如下：

```
init_monitor(int argc, char*argv[]) //monitor初始化
{
    reg_test();//生成随机数，测试寄存器结构的正确性
    load_img();//读入带有客户程序的镜像文件
               //缺省时调用载入默认镜像函数load_default_img()
    restart();//模拟“计算机启动，进行"计算机启动"相关的初始化工作，
              //将eip初始值设为0x100000，确保CPU从该位置开始执行程序
    init_regex();//输入命令正则表达检测
    init_wp_pool();//监视点初始化
    init_device();//设备初始化
    welcome();//输出欢迎信息和NEMU编译时间
    return is_banch_mode;
}
ui_mainloop(int is_batch_mode)//用户界面主循环程序，读入用户命令，实现交互功能
```

在NEMU目录下执行make run命令，会出现assertion fail的错误信息，这是因为reg_test()函数在monitor初始化过程中，会测试寄存器结构是否正确，否则无法进行monitor其他部分的初始化，更无法进入用户界面主循环ui_mainloop()。因此为了成功运行NEMU，需要参考PA实验指导中对寄存器的规定实现用于模拟寄存器的结构体CPU_state（目录: nemu/include/cpu/reg.h）。

实现模拟寄存器结构CPU_state后，我们就可以成功运行NEMU，并调用现有的帮助（help）、继续运行（c）和退出（q）命令来调试NEMU，为了在接下来的实验中在NEMU上调试所运行的程序，我们要将简易调试器剩余的命令实现，第一部分需要实现的命令为：单步执行（si [N]）、打印寄存器状态（info r）和扫描内存（x N EXPR）（目录：nemu/src/monitor/debug/u.c）。

2.1 实现方法及代码分析

为实现NEMU所要求的寄存器，需要使用union来完成寄存器内存的复用；阅读用户界面主循环函数ui_mainloop()内容可以明白MENU执行用户输入命令的具体方式，所有命令的名字、描述和调用句柄都在cmd_table[]结构体数组中定义，ui_mainloop()函数使用strtok()函数解析用户输入命令，并调用对应处理函数。因此想要实现其他命令，我们需要完善cmd_table[]结构体数组和对应处理函数。

2.1.1 寄存器结构实现

NEMU为兼容x86，所选择的寄存器结构体：

```
 31              23              15              7               0
+----------------+----------------+----------------+----------------+
|                                EAX       AH       AX       AL      |
|----------------+----------------+----------------+----------------|
|                                EDX       DH       DX       DL      |
|----------------+----------------+----------------+----------------|
|                                ECX       CH       CX       CL      |
|----------------+----------------+----------------+----------------|
|                                EBX       BH       BX       BL      |
|----------------+----------------+----------------+----------------|
|                                EBP                BP              |
|----------------+----------------+----------------+----------------|
|                                ESI                SI              |
|----------------+----------------+----------------+----------------|
|                                EDI                DI              |
|----------------+----------------+----------------+----------------|
|                                ESP                SP              |
+----------------+----------------+----------------+----------------+
```

```
 31              23              15              7               0
+----------------+----------------+----------------+----------------+
|                     EIP (INSTRUCTION POINTER)                      |
+----------------+----------------+----------------+----------------+
```

其中EAX，EDX，ECX，EBX，EBP，ESI，EDI，ESP 是32位寄存器；AX，DX，CX，BX，BP，SI，DI，SP 是16位寄存器；AL，DL，CL，BL，AH，DH，CH，BH 是8位寄存器。但部分寄存器存在内存空间的复用，例如EAX的低16位是AX，而AX低8位是AL，高8位是AH，其他服复用可参考寄存器结构图片。再结合PA实验指导中的提示（Hint：使用匿名union）所以我们在模拟寄存器的时候使用union。

具体实现代码及注释：

```
typedef struct {
  union //将union与struct共同包装为union可使eax、ecx、edx等32位寄存器与
        //gpr[0]、gpr[1]、gpr[2]等对应，方便其他函数对寄存器的访问
  {
    union //使用union实现8个32位寄存器，和寄存器的内存空间复用
          //32位Ubuntu是小端模式，所以16位寄存器必须放置在32位的前16位
          //低八位寄存器必须放置在16位的前8位
    {
      uint32_t _32;
      uint16_t _16;
      uint8_t _8[2];
    } gpr[8];
    struct //32位寄存器互相独立，需要包装为一个结构体
    {
      rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi; //rtlreg_t即uint32_t
    };
  };
  vaddr_t eip; //vaddr_t即uint32_t
} CPU_state;
extern CPU_state cpu;
```

此外在相同文件中还有reg_l，reg_w，reg_b的宏定义和regsl，regsw，regsb的声明，regsl，regsw，regsb的具体定义在reg.c文件中，可以利用这两部分方便的在其他文件中实现对寄存器名字和存储内容的访问。

```
#define reg_l(index) (cpu.gpr[check_reg_index(index)]._32)
#define reg_w(index) (cpu.gpr[check_reg_index(index)]._16)
#define reg_b(index) (cpu.gpr[check_reg_index(index) & 0x3]._8[index >> 2])

extern const char* regsl[];
extern const char* regsw[];
extern const char* regsb[];

//代码位置：nemu/src/cpu/reg.c
const char *regsl[] = {"eax", "ecx", "edx", "ebx", "esp", "ebp", "esi", "edi"};
const char *regsw[] = {"ax", "cx", "dx", "bx", "sp", "bp", "si", "di"};
const char *regsb[] = {"al", "cl", "dl", "bl", "ah", "ch", "dh", "bh"};
```

2.1.2 si，info r，x命令实现

NEMU简易调试器对命令格式要求：

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

其中所有命令和用户界面主循环函数ui_mainloop()均在nemu/src/monitor/debug/ui.c文件中，参考已实现的帮助（help）、继续运行（c）和退出（q）命令和ui_mainloop()函数内容我们可以了解MENU执行用户输入命令的具体方式，所有命令的名字、描述和调用句柄都在cmd_table[]结构体数组中定义，ui_mainloop()函数使用strtok()函数以空格为界解析用户输入命令，通过遍历cmd_table[]数组找到对应命令并调用对应cmd_table[i].handler(args)来执行相关的处理函数。因此想要实现其他命令，我们需要在cmd_table[]结构体数组中加入相应命令的名字、描述（help命令中展示）和调用句柄，声明并完成处理函数。

各命令调用函数声明及cmd_table[]结构体数组命令定义代码：

```
static int cmd_help(char *args);
static int cmd_si(char *args);
static int cmd_info(char *args);
static int cmd_x(char *args);
static int cmd_p(char *args);
static int cmd_w(char *args);
static int cmd_d(char *args);

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", "Single-step debug", cmd_si },
  { "info", "Print program status", cmd_info },
  { "x", "Scan memory", cmd_x },
  { "p", "Evaluate expression", cmd_p },
  { "w", "Set watchpoint", cmd_w },
  { "d", "Delete watchpoint", cmd_d },
};
```

继续运行（c）命令代码中调用了cpu_exec()函数，且通过阅读cpu_exec()函数代码（目录：nemu/src/monitor/cpu-exec.c），知道它的功能是模仿CPU的工作方式，不断执行n条指令，直到指令执行完毕或nemu_state不为NEMU_RUNNING状态，传入参数为uint64_t 类型，即unsigned long long（64位无符号整型），所以cmd_c()函数中传入的参数-1相当于2^64-1，可以确保NEMU将剩余所有的指令执行完。

```
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 --) { //执行n条命令
    exec_wrapper(print_flag);
    if (nemu_state != NEMU_RUNNING) { return; } //状态改变退出执行
  }
  if (nemu_state == NEMU_RUNNING) { nemu_state = NEMU_STOP; }
}
```

在cmd_si()函数的实现中只需要调用cpu_exec()函数并传入不同参数即可。

单步执行（si [N]）实现代码及注释：

```
static int cmd_si(char *args)
{
  int step;
  if (args == NULL) //无参数默认执行一步
    cpu_exec(1);
  else
  {
    step = atoi(args); //执行步数解析
    cpu_exec(step);
  }
  return 0;
}
```

打印寄存器状态函数的实现中直接调用regsl，regsw，regsb数组打印寄存器名字，调用reg_l，reg_w，reg_b宏定义打印寄存器状态，注意eip寄存器单独打印。cmd_info()中打印监视点部分在PA1的阶段三实现。

打印寄存器状态（info r）实现代码及注释：

```
static void print_register_status()
{
  int i,j;
  for(i = 0,j = 4; i < 4; i++,j++)
  { //按顺序以16进制形式打印前四个32位寄存器及复用的16位和两个8位寄存器状态
    printf("  %s  0x%08x | %s  0x%04x | %s  0x%02x | %s  0x%02x\n",
      regsl[i],reg_l(i),regsw[i],reg_w(i),regsb[i],reg_b(i),regsb[j],reg_b(j));
  }
  for(; i < 8; i++)
  { //按顺序以16进制形式打印后四个32位寄存器及复用的16位寄存器状态
    printf("  %s  0x%08x | %s  0x%04x\n",regsl[i],reg_l(i),regsw[i],reg_w(i));
  }
  printf("  eip  0x%08x\n",cpu.eip); //打印eip寄存器
}

static int cmd_info(char *args)
{
  if (args == NULL)
    return 0;
  switch(*args) //解析info命令
  {
    case 'r':
      print_register_status(); //调用打印寄存器状态函数
      break;
    case 'w':
      print_watchpoint_status();
      break;
    default:
      return 0;
  };
  return 0;
}
```

在阅读NEMU框架代码时我们知道了NEMU内存的实现方式——一个大小为128×1024×1024，数据类型为uint8_t的大数组（目录：nemu/src/memory/memory.c），在同目录下还有内存读取的函数uint32_t vaddr_read(vaddr_t addr, int len)实现从addr位置开始len个字节的读取。

```
#define pmem_rw(addr, type) *(type *)({\
    Assert(addr < PMEM_SIZE, "physical address(0x%08x) is out of bound", addr);
\
    guest_to_host(addr); \
    })
uint32_t paddr_read(paddr_t addr, int len) //addr为内存首地址，len(1,2,3,4)控制读取内
存的字节数
{  //将32位0(无符号数)全部取反为1，右移((4-len)<<3)位
   //len分别取1 2 3 4，可得pmem_rw(addr,uint32_t)的最后8 16 32 64位
  return pmem_rw(addr, uint32_t) & (~0u >> ((4 - len) << 3));
}
uint32_t vaddr_read(vaddr_t addr, int len)
{
  return paddr_read(addr, len);
}

//代码位置：nemu/include/memory/memory.h
#define guest_to_host(p) ((void *)(pmem + (unsigned)p)) //guest地址转换host地址
#define host_to_guest(p) ((paddr_t)((void *)p - (void *)pmem)) //host地址转换guest
地址
```

因此为了实现内存的扫描需要在cmd_x()函数中调用vaddr_read()函数并传入需要扫描的地址和步数参数，在实验过程中为了减少重复工作，此处先实现了除表达式求值外的函数框架，在阶段二表达式求值的的功能才完善了扫描内存命令。

扫描内存（x N EXPR）实现代码及注释：

```
static int cmd_x(char *args)
{
  char *arg = strtok(NULL, " "); //将N从字符串中分离
  if (arg == NULL)
    return 0;
  int num;
  sscanf(arg,"%d",&num); //解析N的数据
  char *exp = args+strlen(arg)+1; //将exp指向N后EXPR
  bool success;
  uint32_t addr=expr(exp,&success); //调用expr()函数计算表达式取值，expr()在阶段二实现
  for(int i=0;i<num*4;i++) //按顺序以16进制形式打印扫描内存地址和对应内容
  {
    if(i%4==0)
      printf("0x%x:  0x%02x",addr+i,vaddr_read(addr+i,1));
    else if(i%4==3)
      printf("  0x%02x\n",vaddr_read(addr+i,1));
    else
      printf("  0x%02x",vaddr_read(addr+i,1));
  }
  return 0;
}
```

2.2 运行结果

寄存器结构实现运行结果：

```
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 18:52:40, Mar 21 2022
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026

(nemu)
```

单步执行（si [N]）运行结果：

```
(nemu) si
  100000:   b8 34 12 00 00                        movl $0x1234,%eax
(nemu) si 1
  100005:   b9 27 00 10 00                        movl $0x100027,%ecx
(nemu) si 4
  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
  100017:   66 c7 84 99 00 e0 ff ff 01 00         movw $0x1,-0x2000(%ecx,%ebx,4)
(nemu)
```

打印寄存器状态（info r）运行结果：

```
(nemu) info r
  eax  0x00001234 | ax  0x1234 | al  0x34 | ah  0x12
  ecx  0x00100027 | cx  0x0027 | cl  0x27 | ch  0x00
  edx  0x58d674d9 | dx  0x74d9 | dl  0xd9 | dh  0x74
  ebx  0x00000002 | bx  0x0002 | bl  0x02 | bh  0x00
  esp  0x1838ce74 | sp  0xce74
  ebp  0x6e3cef47 | bp  0xef47
  esi  0x283ca2f4 | si  0xa2f4
  edi  0x703f7432 | di  0x7432
  eip  0x00100021
(nemu)
```

扫描内存（x N EXPR）运行结果：

```
(nemu) x 10 0x100000
0x100000:  0xb8  0x34  0x12  0x00
0x100004:  0x00  0xb9  0x27  0x00
0x100008:  0x10  0x00  0x89  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  0x00
(nemu) x 10 $eip
0x100000:  0xb8  0x34  0x12  0x00
0x100004:  0x00  0xb9  0x27  0x00
0x100008:  0x10  0x00  0x89  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  0x00
(nemu)
```

2.3 Bug总结

阶段一部分遇到的Bug数量较少，首先是在模拟寄存器结构体的部分要注意将8个32位寄存器打包位一个结构体，并且最外层的结构必须是union，这样才能确保可以使用cpu.eax，cpu.ecx等方式来访问寄存器；其次是在打印寄存器状态（info r）的函数中使用printf打印寄存器内容时输出16进制并使用%08x，%04x，%02x来限制输出长度，注意只有前四个32位寄存器各包含两个8位寄存器，顺序遍历8位寄存器输出会导致寄存器对应错误，不方便观察；最后在扫描内存（x N EXPR）部分要注意与前两个命令实现不同的部分在于扫描内存需要两个参数，N和EXPR，但strtok()函数会自动在截断处添加字符串结束符'\0'，所以在要将char *exp指到args+strlen(arg)+1，否则读出的仍是N。

3 阶段二

3.1 实现方法及代码分析

阶段二中需要实现的简易调试器功能为表达式求值（p EXPR），但想要最终实现表达式求值的功能，需要先实现token识别、括号匹配、dominant operator寻找的基础功能，在此基础上完善递归求值功能和各分支，最后在表达式求值函数expr()中调用以上模块就可完成。除此之外还需要像一阶段其他命令的实现一样在nemu/src/monitor/debug/ui.c文件中cmd_table[]结构体数组中加入表达式求值命令的名字"p"、功能描述和调用句柄cmd_p()，定义并完成处理函数。

实验过程中为了减少重复工作直接完成复杂表达式的求值（包括负号"-"和解引用"*"的实现）。

3.1.1 词法分析实现

词法分析（目录：nemu/src/monitor/debug/expr.c）需要实现表达式中每一个token的识别，并用一个数组来记录所输入的表达式，并且考虑到数字（包括十进制和十六进制）和寄存器参与表达式运算时作为一个整体，所以对在记录token类型以上三个token类型还需要记录对应值。因为事实上token类型只是一个整数，所以只需要保证不同的类型的token被编码成不同的整数就可以了，所以单符号的运算符包括"+"，"-"，"*"，"/"，"("，")"和"!"都不需要额外定义token类型（但额外定义也可以）。

比较特殊的是负号"-"和解引用"*"符号，因为这两个符号与减号和乘号相同，但运算时的处理不同，所以我们需要为这两个运算符添加token类型，而负号和减号的区别、解引用的乘号的区别则需要进行不同情况的判断。

词法分析实现代码及注释：

```
//所有token正则表达式和需要额外token类型token定义
enum {
  TK_NOTYPE = 256, TK_HEXNUM, TK_NUM, TK_REG, TK_LP, TK_RP,
  TK_EQ, TK_NOTEQ, TK_AND, TK_OR,
  TK_NEG, TK_DEREF
};

static struct rule {
  char *regex;
  int token_type;
} rules[] = {
  {" +", TK_NOTYPE},                // spaces
  {"0x[0-9a-f]+",TK_HEXNUM},        // hexnumber
  {"[0-9]+",TK_NUM},                // number
  {"\\$[a-z]{2,3}",TK_REG},         // register
  {"\\(", TK_LP},                   // '('
  {"\\)", TK_RP},                   // ')'
  {"\\+", '+'},                     // plus
  {"-", '-'},                       // minus
  {"\\*", '*'},                     // multiply
  {"\\/", '/'},                     // divide
  {"==", TK_EQ},                    // equal
  {"!=", TK_NOTEQ},                 // not equal
  {"&&", TK_AND},                   // and
  {"\\|\\|", TK_OR},                // or
  {"!", '!'},                       // not
};

typedef struct token {
  int type;  //记录token类型
```

```
  char str[32];  //记录数字（包括十进制和十六进制）和寄存器值
} Token;

Token tokens[32]; //利用数组记录表达式所有已识别的token
int nr_token; //记录已识别的token数目

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

  nr_token = 0;

  while (e[position] != '\0') {
  //依次识别当前字符串
    for (i = 0; i < NR_REGEX; i ++) {
      if (regexec(&re[i], e + position, 1, &pmatch, 0) == 0 && pmatch.rm_so ==
0)       {
        char *substr_start = e + position;
        int substr_len = pmatch.rm_eo;
        //成功识别token输出对应信息
        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;

        switch (rules[i].token_type) {
          case TK_NOTYPE: //空格识别
            break;
          case TK_HEXNUM: //16进制数字识别
            if(substr_len > 32) //Token结构体中记录对应值的str[]数组长度为32
                                //超过32位会出现识别的错误，需要丢出异常
            {
              printf("The hexadecimal number is too long\n");
              assert(0);
            }
            else
            {
              tokens[nr_token].type=TK_HEXNUM;
              //使用strncpy()函数复制，但strncpy()函数不添加结束符'\0',需要手动添加
              //需要去除数字前的0x
              strncpy(tokens[nr_token].str,substr_start+2,substr_len-2);
              *(tokens[nr_token].str+substr_len-2)='\0'; //添加结束符
              nr_token++; //成功识别nr_token++
            }
            break;
          case TK_NUM: //10进制数字识别
            if(substr_len > 32)
            {
              printf("The number is too long\n");
              assert(0);
            }
            else
            {
              tokens[nr_token].type=TK_NUM;
              strncpy(tokens[nr_token].str,substr_start,substr_len);
              *(tokens[nr_token].str+substr_len)='\0';
              nr_token++;
            }
```

```
            break;
          case TK_REG: //寄存器识别
            tokens[nr_token].type=TK_REG;
            //需要去除寄存器名称前$
            strncpy(tokens[nr_token].str,substr_start+1,substr_len-1);
            *(tokens[nr_token].str+substr_len-1)='\0';
            nr_token++;
            break;
          case TK_LP: //"("识别
            tokens[nr_token++].type=TK_LP;
            break;
          case TK_RP: //")"识别
            tokens[nr_token++].type=TK_RP;
            break;
          case '+': //"+"识别
            tokens[nr_token++].type='+';
            break;
          case '-': //减号和负号识别
          { //若"-"位于整个表达式第一位或是其前一位不是数字(10进制或16进制)和")",则为负号
            if(nr_token==0 || (tokens[nr_token-1].type!=TK_NUM &&
tokens[nr_token-1].type!=TK_HEXNUM && tokens[nr_token-1].type!=TK_RP))
            {
              tokens[nr_token++].type=TK_NEG;
              break;
            }
            else
            { //其他情况为减号
              tokens[nr_token++].type='-';
              break;
            }
          }
          case '*': //乘号和解引用识别
          { //若"*"位于整个表达式第一位或是其前一位不是数字(10进制或16进制)和")",则为解引用
            if(nr_token==0 || (tokens[nr_token-1].type!=TK_NUM &&
tokens[nr_token-1].type!=TK_HEXNUM && tokens[nr_token-1].type!=TK_RP))
            {
              tokens[nr_token++].type=TK_DEREF;
              break;
            }
            else
            { //其他情况为乘号
              tokens[nr_token++].type='*';
              break;
            }
          }
          case '/': //"/"识别
            tokens[nr_token++].type='/';
            break;
          case TK_EQ: //"=="识别
            tokens[nr_token++].type=TK_EQ;
            break;
          case TK_NOTEQ: //"!="识别
            tokens[nr_token++].type=TK_NOTEQ;
            break;
          case TK_AND: //"&&"识别
            tokens[nr_token++].type=TK_AND;
            break;
          case TK_OR: //"||"识别
```

```
            tokens[nr_token++].type=TK_OR;
            break;
          case '!': //"!"识别
            tokens[nr_token++].type='!';
            break;
          default: //若所有token都未匹配，则表达式中存在无法识别的token，抛出异常
            printf("The token type is wrong\n");
            assert(0);
            break;
        }
        break;
      }
    }
    if (i == NR_REGEX) {
      printf("no match at position %d\n%s\n%*.s^\n", position, e, position, "");
      return false;
    }
  }
  return true;
}
```

3.1.2 check_parentheses()实现

依据PA实验指导中表达式的定义和递归求值函数的分支，我们需要对表达式中的括号进行检查，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，倘若不匹配则无法进行求值。

check_parentheses()实现代码及注释：

```
bool check_parentheses(int p,int q)
{
  int p_number=0; //标记括号层级
  //若表达式左右不是都有括号直接返回false
  if(tokens[p].type!=TK_LP || tokens[q].type!=TK_RP)
    return false;
  for(int i=p;i<=q;i++) //遍历表达式
  {
    if(tokens[i].type==TK_LP)
      p_number++; //遇"("层级+1
    else if(tokens[i].type==TK_RP)
      p_number--; //遇")"层级-1
    if(i<q && p_number==0) //若未遍历完全部表达式括号层级降为0，代表表达式最左右括号不匹配
      return false;
  }
  if(p_number==0)
    return true;
  return false; //表达式括号不匹配
}
```

3.1.3 find_dominant_operator()实现

在面对最左边和最右边不同时是括号的合法长表达式时，我们需要正确地将它分裂成两个子表达式，PA实验流程中定义dominant operator为表达式人工求值时，最后一步进行运行的运算符。在实际的表达式求值中需要找到长表达式中的dominant operator，才能正确地将一个长表达式分裂成两个表达式来进行后续处理。

dominant operator寻找法则：

- 非运算符的token不是dominant operator
- 出现在一对括号中的token不是dominant operator
- dominant operator的优先级在表达式中是最低的
- 当有多个运算符的优先级都是最低时，根据结合性，最后被结合的运算符才是dominant operator

这里采用的表达式优先级为：

0　&& ||，1　== !=，2　+ -，3　* /，4　！负号 解引用

check_parentheses()实现代码及注释：

```
int find_dominant_operator(int p,int q)
{
  //p_number为括号层级，poi为dominant operator位置，pri为表达式中优先度最低运算符优先度
  int p_number=0,poi=0,pri=5;
  for(int i=p;i<=q;i++)
  {
    if (tokens[i].type==TK_LP)
      p_number++;
    else if (tokens[i].type==TK_RP)
      p_number--;
    if(p_number==0) //一对括号内的token不是dominant operator
    { //当遇到优先级相同或更低的运算符时更新poi与pri
      if (pri>=0 && (tokens[i].type==TK_AND || tokens[i].type==TK_OR))
        {poi=i;pri=0;}
      else if (pri>=1 && (tokens[i].type==TK_EQ || tokens[i].type==TK_NOTEQ))
        {poi=i;pri=1;}
      else if (pri>=2 && (tokens[i].type=='+' || tokens[i].type=='-'))
        {poi=i;pri=2;}
      else if (pri>=3 && (tokens[i].type=='*' || tokens[i].type=='/'))
        {poi=i;pri=3;}
      else if (pri>=4 && (tokens[i].type==TK_DEREF || tokens[i].type==TK_NEG ||
tokens[i].type=='!'))
        {poi=i;pri=4;}
    }
  }
  return poi;
}
```

此外根据运算规则，双目运算符应记录最后一个出现的的dominant operator，而单目运算符应记录第一个出现的dominant operator。但这里并未对单目运算符做特别的处理。因为若最终的dominant operator为单目运算符，则说明该表达式只有单目运算符，运算时直接从表达式的第一位开始分解即可具体操作在eval()函数中实现。

3.1.4 递归求值实现

根据PA实验流程中递归求值函数的结构：

```
eval(p, q) {
  if (p > q) {
    /* Bad expression */
  }
  else if (p == q) {
    /* Single token.
     * For now this token should be a number.
     * Return the value of the number.
     */
  }
  else if (check_parentheses(p, q) == true) {
    /* The expression is surrounded by a matched pair of parentheses.
     * If that is the case, just throw away the parentheses.
     */
    return eval(p + 1, q - 1);
  }
  else {
    op = the position of dominant operator in the token expression;
    val1 = eval(p, op - 1);
    val2 = eval(op + 1, q);
```

```
    switch (op_type) {
      case '+': return val1 + val2;
      case '-': /* ... */
      case '*': /* ... */
      case '/': /* ... */
      default: assert(0);
    }
  }
}
```

可以对eval()函数进行补全和完善，具体的分支实现在上图中已经非常清晰，p，q分别代表表达式两端的位置，p>q时抛出异常；p=q时意味着该表达式为数字（10进制或16进制）或寄存器，只需要读出对应的值即可；check_parentheses(p,q)==true时去掉括号；其他情况则需要将原表达式进行分解。只有dominant operator是单目运算符的情况需要补充，当dominant operator是单目运算符时，则意味着该表达式中只有单目运算符，运算时应从表达式的第一位开始分解。

递归求值实现代码及注释：

```
uint32_t eval(int p,int q)
{
  //printf("p=%d,q=%d\n",p,q);
  if(p>q) //p>q抛出异常
  {
    printf("The expression is wrong(p>q in eval),p=%d,q=%d\n",p,q);
    assert(0);
  }
  else if(p==q) //p=q时意味着该表达式为数字（10进制或16进制）或寄存器
  {
    int num;
    if(tokens[p].type==TK_NUM)
    {
      sscanf(tokens[p].str,"%d",&num); //读出10进制数字对应值
      return num;
    }
    else if(tokens[p].type==TK_HEXNUM)
    {
      sscanf(tokens[p].str,"%x",&num); //读出16进制数字对应值
```

```
      return num;
    }
    else if(tokens[p].type==TK_REG) //遍历所有寄存器，匹配则返回对应寄存器所存值
    {
      for(int i=0;i<8;i++)
      {
        if(strcmp(tokens[p].str,regsl[i])==0)
          return reg_l(i);
        if(strcmp(tokens[p].str,regsw[i])==0)
          return reg_w(i);
        if(strcmp(tokens[p].str,regsb[i])==0)
          return reg_b(i);
      }
      if(strcmp(tokens[p].str,"eip")==0)
        return cpu.eip;
      else
      { //若遍历所有寄存器仍未找出匹配则抛出异常
        //这是因为在正则表达式匹配时的寄存器表达式为"\\$[a-z]{2,3}"，非常宽松，可能出现不是
寄存器的token却被识别为寄存器
        printf("The registers name is wrong\n");
        assert(0);
      }
    }
    else
    {
      printf("wrong in eval(p==q)\n");
      assert(0);
    }
  }
  else if(check_parentheses(p,q)==true)
  {
    return eval(p+1,q-1); //去掉外层括号
  }
  else
  {
    int op=find_dominant_operator(p,q);
    //printf("op=%d\n",op);
    //op为单目运算符时的表达式分解，直接将第一位分解出去即可
    if(tokens[op].type==TK_NEG || tokens[op].type==TK_DEREF ||
tokens[op].type=='!')
    {
      if(tokens[p].type==TK_NEG) //符号分解
        return -eval(p+1,q);
      else if(tokens[p].type==TK_DEREF) //解引用分解，需调用vaddr_read()函数读取内存
      {
        vaddr_t addr;
        addr=eval(p+1,q);
        return vaddr_read(addr,4);
      }
      else if(tokens[p].type=='!') //"!"分解
      {
        if(eval(p+1,q)!=0)
          return 0;
        else
          return 1;
      }
    }
    //op为双目运算符时的表达式分解
```

```
    int val1=eval(p,op-1);
    int val2=eval(op+1,q);
    //printf("op.type=%d\n",tokens[op].type);
    switch (tokens[op].type)
    {
      case '+': return val1+val2;
      case '-': return val1-val2;
      case '*': return val1*val2;
      case '/': { if(val2==0)
                  { //检测除数是否为0，为0时抛出异常
                    printf("Can't divide 0\n");
                    assert(0);
                  }
                  else
                    return val1/val2;
                }
      case TK_EQ: return val1 == val2;
      case TK_NOTEQ: return val1 != val2;
      case TK_AND: return val1 && val2;
      case TK_OR: return val1 || val2;
      default: assert(0); //所有双目运算符均不匹配，抛出异常
    }
  }
}
```

最后再在expr()函数中调用eval()函数即可完成expr()函数的表达式求值功能。

```
uint32_t expr(char *e, bool *success) {
  if (!make_token(e)) { //make_token()失败直接返回
    *success = false;
    return 0;
  }
  *success = true;
  return eval(0,nr_token-1); //成功则调用eval()函数
}
```

3.1.5 p命令实现

表达式求值命令调用函数声明及cmd_table[]结构体数组命令定义均在一阶段代码段可见，此处仅展示cmd_p()函数代码。

表达式求值（p EXPR）实现代码及注释：

```
static int cmd_p(char *args)
{
  if (args == NULL)
    return 0;
  bool success;
  uint32_t result=expr(args,&success); //调用expr()函数求解表达式
  printf("exp result:  0x%08x  %d\n",result,result); //打印结果
  return 0;
}
```

完成expr()函数后在扫描内存处理函数cmd_x()函数中调用expr()函数将一阶段内存扫描功能完善。

3.2 运行结果

词法分析运行结果：

```
(nemu) p 2*(3+8/4)--0x10
[src/monitor/debug/expr.c,88,make_token] match rules[2] = "[0-9]+" at position 0
 with len 1: 2
[src/monitor/debug/expr.c,88,make_token] match rules[8] = "\*" at position 1 wit
h len 1: *
[src/monitor/debug/expr.c,88,make_token] match rules[4] = "\(" at position 2 wit
h len 1: (
[src/monitor/debug/expr.c,88,make_token] match rules[2] = "[0-9]+" at position 3
 with len 1: 3
[src/monitor/debug/expr.c,88,make_token] match rules[6] = "\+" at position 4 wit
h len 1: +
[src/monitor/debug/expr.c,88,make_token] match rules[2] = "[0-9]+" at position 5
 with len 1: 8
[src/monitor/debug/expr.c,88,make_token] match rules[9] = "\/" at position 6 wit
h len 1: /
[src/monitor/debug/expr.c,88,make_token] match rules[2] = "[0-9]+" at position 7
 with len 1: 4
[src/monitor/debug/expr.c,88,make_token] match rules[5] = "\)" at position 8 wit
h len 1: )
[src/monitor/debug/expr.c,88,make_token] match rules[7] = "-" at position 9 with
 len 1: -
[src/monitor/debug/expr.c,88,make_token] match rules[7] = "-" at position 10 wit
h len 1: -
[src/monitor/debug/expr.c,88,make_token] match rules[1] = "0x[0-9a-f]+" at posit
ion 11 with len 4: 0x10
```

```
(nemu) p 3!=3&&(*0x10000==0x1234b8)
[src/monitor/debug/expr.c,88,make_token] match rules[2] = "[0-9]+" at position 0
 with len 1: 3
[src/monitor/debug/expr.c,88,make_token] match rules[11] = "!=" at position 1 wi
th len 2: !=
[src/monitor/debug/expr.c,88,make_token] match rules[2] = "[0-9]+" at position 3
 with len 1: 3
[src/monitor/debug/expr.c,88,make_token] match rules[12] = "&&" at position 4 wi
th len 2: &&
[src/monitor/debug/expr.c,88,make_token] match rules[4] = "\(" at position 6 wit
h len 1: (
[src/monitor/debug/expr.c,88,make_token] match rules[8] = "\*" at position 7 wit
h len 1: *
[src/monitor/debug/expr.c,88,make_token] match rules[1] = "0x[0-9a-f]+" at posit
ion 8 with len 7: 0x10000
[src/monitor/debug/expr.c,88,make_token] match rules[10] = "==" at position 15 w
ith len 2: ==
[src/monitor/debug/expr.c,88,make_token] match rules[1] = "0x[0-9a-f]+" at posit
ion 17 with len 8: 0x1234b8
[src/monitor/debug/expr.c,88,make_token] match rules[5] = "\)" at position 25 wi
th len 1: )
```

```
(nemu) p !($eip==0x100000)
[src/monitor/debug/expr.c,88,make_token] match rules[14] = "!" at position 0 wit
h len 1: !
[src/monitor/debug/expr.c,88,make_token] match rules[4] = "\(" at position 1 wit
h len 1: (
[src/monitor/debug/expr.c,88,make_token] match rules[3] = "\$[a-z]{2,3}" at posi
tion 2 with len 4: $eip
[src/monitor/debug/expr.c,88,make_token] match rules[10] = "==" at position 6 wi
th len 2: ==
[src/monitor/debug/expr.c,88,make_token] match rules[1] = "0x[0-9a-f]+" at posit
ion 8 with len 8: 0x100000
[src/monitor/debug/expr.c,88,make_token] match rules[5] = "\)" at position 16 wi
th len 1: )
```

表达式求值（p EXPR）运行结果：

```
(nemu) p 2+3*8
exp result:  0x0000001a  26
(nemu) -(0x10+6)/2
Unknown command '-(0x10+6)/2'
(nemu) p 2+3*8
exp result:  0x0000001a  26
(nemu) p -(0x10+6)/2
exp result:  0xfffffff5  -11
(nemu) p --2*5
exp result:  0x0000000a  10
(nemu) p $eip
exp result:  0x00100000  1048576
(nemu) p $ecx
exp result:  0x61cb201e  1640701982
(nemu) p $cl
exp result:  0x0000001e  30
(nemu) p !(26-13*2)
exp result:  0x00000001  1
(nemu) p 3==3&&(*0x100000==0x1234b8)
exp result:  0x00000001  1
(nemu) p 0||1
exp result:  0x00000001  1
(nemu)
```

3.3 Bug总结

阶段二中的Bug主要集中在递归求值的实现，词法分析部分关键在于正则表达式的编写和负号以及解引用的识别，需要注意转义符"\"的使用来让在正则表达式中有额外含义的"+"，"*"等符号可以表示其原意，列举多种情况分出负号与减号、解引用和乘号的区别，还有就是strncpy()函数的使用，需要手动添加字符串结束符"\0"；递归求值部分括号匹配的实现比较简单，参考用栈实现括号匹配的思路即可，dominant operator寻找则需要注意单目运算符和双目运算符结合性的不同，尽管可以不在find_dominant_operator()函数处区分，但若是忽略这一区别，表达式计算的功能一定会出问题，PA实验流程中所给的递归求值的eval()函数结构已相当完备，只需补充dominant operator为单目运算符时的状况即可。

在调试NEMU时可以大量使用printf来打印部分变量信息，assert来抛出异常从而跟踪变量的变化，找到Bug位置。

4 阶段三

4.1 实现方法及代码分析

NEMU简易调试器允许用户同时设置多个监视点，删除监视点，PA实验流程中建议使用链表来组织监视点的信息。在框架代码nemu/include/monitor/watchpoint.h中已经定义了监视点的结构，但只包括NO和next两个成员，分别表示监视点的编号和指向下一个监视点的指针。参考GDB中监视点的格式，需要增加char expr[]和int value两个变量来记录监视点所监视的表达式以及对应值。

nemu/src/monitor/debug/watchpoint.c中代码定义了监视点结构的池wp_pool，还有两个链表head和free_，其中head用于组织使用中的监视点结构，free_用于组织空闲的监视点结构，init_wp_pool()函数会对两个链表进行了初始化。为了使用监视点池，需要添加bool new_wp(char* exp)函数和bool free_wp(int no)函数，其中new_wp()从free_链表摘取一个空闲的监视点结构并将其接入head，而free_wp()则将待删除的监视点归还到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。在实现这两个函数前需要先在watchpoint.h文件中添加声明，为了真正实现监视点的功能还实现需要检测监视点变化的函数和打印监视点信息的函数，并且完善nemu/src/monitor/debug/u.c文件中的设置监视点（w EXPR），打印监视点（info w）和删除监视点（d N）命令。

4.1.1 监视点池管理实现

首先在nemu/include/monitor/watchpoint.h文件中完善结构体，声明bool new_wp(char* exp)函数和bool free_wp(int no)函数。

WP结构体完善实现代码及注释：

```
typedef struct watchpoint {
  int NO;
  struct watchpoint *next;
  char expr[32]; //记录监视点所监视的表达式
  int value; //记录监视点所监视的表达式对应值
} WP;
```

new_wp()和free_wp()函数声明代码及注释：

```
bool new_wp(char* exp); //new_wp()函数声明，仿照GDB监视点格式，exp表示监视点表达式
bool free_wp(int no); //free_wp()函数声明，仿照GDB监视点格式，no表示待删除监视点编号
bool monitor_wppool_change(); //用于检测监视点变化，在4.1.2部分实现
void print_watchpoint_status(); //用于打印监视点状态，在4.1.2部分实现
```

其次在nemu/src/monitor/debug/watchpoint.c文件中完善监视点链表的初始化并定义new_wp()和free_wp()函数，这里监视点池的编号在GDB基础上稍作修改，第一个监视点编号为1，之后的监视点编号为当前编号最大的监视点编号+1。

init_wp_pool()函数完善代码及注释：

```
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[i].value = 0; //初始化value为0
  }
  wp_pool[NR_WP - 1].next = NULL;
  head = NULL;
  free_ = wp_pool;
}
```

bool new_wp(char* exp)函数实现代码及注释：

```
bool new_wp(char* exp)
{
  if(exp==NULL) //若表达式为空，抛出异常
  {
    printf("No expression(in new_wp)\n");
    assert(0);
  }
  if(free_==NULL) //若free_链表为空，则代表已无空监视点可分配，返回false
  {
    printf("No empty watchpoint\n");
    return false;
  }
  WP* new_wp=free_; //将free_链表第一个监视点摘下
  free_=free_->next;
```

```
  new_wp->next=NULL;
  bool success;
  uint32_t result=expr(exp,&success); //计算表达式对应值
  strcpy(new_wp->expr,exp); //监视点expr[]赋值
  new_wp->value=result; //监视点value赋值

  if(head!=NULL) //若head链表不为空
  {
    WP* temp=head;
    while(temp->next!=NULL) //找到当前head链表的最后一个节点
    {
        temp=temp->next;
    }
    temp->next=new_wp; //将new_wp链接在head链表尾部
    new_wp->NO=temp->NO+1; //new_wp.NO设置为当前编号最大的监视点编号+1
  }
  else //若head链表为空，则new_wp就是第一个监视点，new_wp.NO设置为1
  {
    head=new_wp;
    new_wp->NO=1;
  }
  //仿照GDB监视点格式输出监视点设置信息，返回true
  printf("Hardware watchpoint %d: %s\n",new_wp->NO,new_wp->expr);
  return true;
}
```

bool free_wp(int no)函数实现代码及注释：

```
bool free_wp(int no)
{
  if(head==NULL) //若head链表为空，则代表当前未设置监视点，返回false
  {
    printf("No watchpoint to delete\n");
    return false;
  }
  WP* delete_wp=head;
  while(delete_wp->NO!=no) //寻找NO等于no的待删除监视点delete_wp
    delete_wp=delete_wp->next;
  if(delete_wp==head) //如果待删除结点为head链表头节点
    head=head->next; //head指针更新
  else if(delete_wp==NULL) //若未找到NO等于no的监视点delete_wp，返回false
  {
    printf("No No.%d watchpoint in pool\n",no);
    return false;
  }
  else //找到了NO等于no的监视点delete_wp
  {
    WP* temp=head;
    while(temp->next!=delete_wp) //寻找delete_wp的前一个监视点
        temp=temp->next;
    temp->next=delete_wp->next; //将delete_wp从head链表中移除
  }
  if(delete_wp!=NULL) //若成功取出delete_wp
  {
    delete_wp->next=free_; //将delete_wp链接在free_链表头部
    free_=delete_wp; //更新free_指针，返回true
    return true;
```

```
  } //否则返回false
  return false;
}
```

4.1.2 监视点实现

monitor_wppool_change()函数和print_watchpoint_status()函数均已在watchpoint.h文件中声明，还需要需要在watchpoint.c文件中完成这两个函数的定义以实现对所有监视点的检测和依次打印全部监视点信息的功能。

monitor_wppool_change()函数实现代码及注释：

```
bool monitor_wppool_change()
{
  if(head==NULL) //若head链表为空，则无监视点，直接返回true
    return true;
  WP* temp=head;
  bool success;
  uint32_t result;
  while(temp!=NULL) //遍历所有监视点
  {
    result=expr(temp->expr,&success); //计算监视点记录表达式对应值
    //printf("result:%d\n",result);
    if(result!=temp->value) //若监视点记录表达式对应值改变
    { //仿照GDB监视点格式打印监视点信息
      printf("Hardware watchpoint %d: %s\n",temp->NO,temp->expr);
      printf("Old Value = %d\n",temp->value);
      printf("New Value = %d\n",result);
      temp->value=result; //更新监视点记录表达式对应值，返回false
      return false;
    }
    temp=temp->next;
  }
  return true; //否则返回true
}
```

print_watchpoint_status()函数实现代码及注释：

```
void print_watchpoint_status()
{
  if(head==NULL) //若head链表为空，则无需打印监视点
  {
    printf("No watchpoint in pool\n");
    return;
  }
  WP* temp=head;
  printf("NO.     expression\n");
  while(temp!=NULL) //遍历所有监视点，仿照GDB监视点格式并打印NO与expr[]
  {
    printf("%d      %s\n",temp->NO,temp->expr);
    temp=temp->next;
  }
}
```

为了实现监视点记录表达式对应值改变程序停止的效果，还需要在cpu_exec()函数中实现监视点的触发判断（目录：nemu/src/monitor/cpu-exec.c）。将monitor/watchpoint.h头文件添加到cpu-exec.c文件中，并在DEBUG宏定义下添加监视点触发判断。

监视点触发判断实现代码及注释：

```
//添加watchpoint.h头文件，否则无法调用monitor_wppool_change()
#include "monitor/watchpoint.h"

#ifdef DEBUG
  if(monitor_wppool_change()==false)
  //当某个监视点记录表达式对应值改变时设置nemu_state为NEMU_STOP，即可停止NEMU指令执行
  {
    nemu_state = NEMU_STOP;
  }
#endif
```

现在实现监视点所需函数均已实现，实现设置监视点（w EXPR），打印监视点（info w）和删除监视点（d N）命令只需在nemu/src/monitor/debug/ui.c文件中定义完善cmd_w()函数，cmd_info()函数和cmd_d()函数即可。

设置监视点（w EXPR）实现代码及注释：

```
static int cmd_w(char *args)
{
  if (args == NULL)
    return 0;
  bool new;
  new=new_wp(args); //调用new_wp()函数设置新监视点
  if(new==false)
    printf("Failed to new watchpoint\n");
  return 0;
}
```

打印监视点（info w）实现代码及注释：

```
static int cmd_info(char *args)
{
  if (args == NULL)
    return 0;
  switch(*args)
  {
    case 'r':
      print_register_status();
      break;
    case 'w':
      print_watchpoint_status(); //调用print_watchpoint_status()函数打印监视点信息
      break;
    default:
      return 0;
  };
  return 0;
}
```

删除监视点（d N）实现代码及注释：

```c
static int cmd_d(char *args)
{
  if (args == NULL)
    return 0;
  int num;
  bool delete;
  sscanf(args,"%d",&num); //解析命令中N
  delete=free_wp(num); //调用free_wp()函数删除标号为num的监视点
  if(delete==true)
    printf("Successfully delete watchpoint %d\n",num);
  else
    printf("Failed to delete watchpoint %d\n",num);
  return 0;
}
```

4.2 运行结果

监视点运行结果：

```
(nemu) w $eip==0x100008
Hardware watchpoint 1: $eip==0x100008
(nemu) w $eip==0x10000c
Hardware watchpoint 2: $eip==0x10000c
(nemu) info w
NO.     expression
1       $eip==0x100008
2       $eip==0x10000c
(nemu) c
Hardware watchpoint 2: $eip==0x10000c
Old Value = 0
New Value = 1
(nemu) d 1
Successfully delete watchpoint 1
(nemu) w $eip==0x100014
Hardware watchpoint 3: $eip==0x100014
(nemu) info w
NO.     expression
2       $eip==0x10000c
3       $eip==0x100014
(nemu) d 2
Successfully delete watchpoint 2
(nemu) d 3
Successfully delete watchpoint 3
(nemu) w $eip==0x100012
Hardware watchpoint 1: $eip==0x100012
(nemu) info w
NO.     expression
1       $eip==0x100012
(nemu)
```

4.3 Bug总结

阶段三中的Bug这要在于设置监视点和删除监视点时head和free_链表的维护，尽管此部分实现的是最简单的单向链表，但依然要注意各指针的维护，链表实现中节点位于链表头和链表尾，以及链表为空等特殊情况是一定要考虑到的；其次需要注意的点是在cpu_exec()函数中实现监视点的触发判断时一定要注意添加头文件；还有就是因为head指针被定义为static变量，所以print_watchpoint_status()函数需要在nemu/src/monitor/debug/watchpoint.c中定义。

5 手册必答题

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？

Figures

2-8 EFLAGS Register

4-1 Systems Flags of EFLAGS Register

Table of Contents

2.3 REGISTERS
2.3.4 Flags Register
2.3.4.1 Status Flags

APPENDIX C STATUS FLAG SUMMARY

- ModR/M字节是什么?

Figures

17-2 ModR/M and SIB Byte Formats

Table of Contents

17.2 INSTRUCTION FORMAT
17.2.1 ModR/M and SIB Bytes

- mov指令的具体格式是怎么样的?

Table of Contents

17.2 INSTRUCTION FORMAT
17.2.2 How to Read the Instruction Set Pages
17.2.2.11 Instruction Set Detail
MOV —— Move Data
MOV —— Move to/from Special Registers

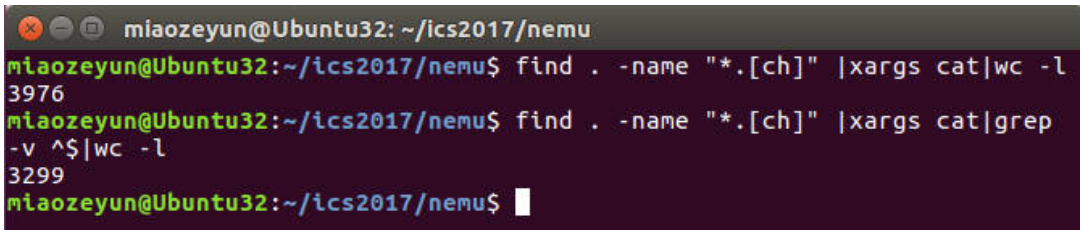
2. shell命令完成PA1的内容之后，nemu/目录下的所有.c和.h和文件总共有多少行代码？你是使用什么命令得到这个结果的？和框架代码相比，你在PA1中编写了多少行代码？（Hint：目前2017分支中记录的正好是做PA1之前的状态，思考一下应该如何回到“过去”？）你可以把这条命令写入Makefile中，随着实验进度的推进，你可以很方便地统计工程的代码行数，例如敲入make count就会自动运行统计代码行数的命令。再来个难一点的，除去空行之外，nemu/目录下的所有.c和.h文件总共有多少行代码?

   在ics2017/nemu目录分别执行

   find . -name "*.[ch]" |xargs cat|wc –l

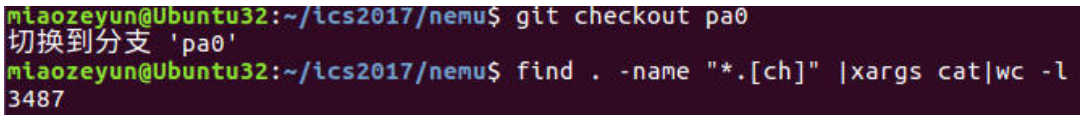
   find . -name "*.[ch]" |xargs cat|grep -v ^$|wc –l

   命令，可得完成PA1的内容之后nemu目录下的所有.c和.h文件总共有多少行代码（第二条为除去空行）



   可见完成PA1内容后nemu目录下的所有.c和.h文件共3976行代码，除去空格共3299行代码。

   想要查看在PA1中共编写多少行代码，可以使用git checkout命令切换回PA0分支，查看PA0全部代码行数，再用在PA1分支下统计的全部的代码行数减去PA0全部代码行数，即可得到在PA1中共编写代码行数。



计算可得在PA1中共编写代码行数为：489行。

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？

```
# Compilation flags
CC = gcc
LD = gcc
INCLUDES  = $(addprefix -I, $(INC_DIR))
CFLAGS   += -O2 -MMD -Wall -Werror -ggdb $(INCLUDES)
```

-Wall：编译后生成所有警告信息
-Werror：在发生警告时停止编译操作，要求GCC将所有警告当成错误进行处理

方便进行程序调试，也可以帮助检查出代码中尚未发现的Bug或是表达不严谨的地方，同时提高代码运行时的安全性，避免内存的非法读取或写入。

6 感悟与体会

1. 跟随PA实验指导阅读源代码可以快速理解需要在本次实验部分中用到的的NEMU框架代码，但一次阅读对代码的理解肯定是不充分的，随着实验过程再反复的阅读框架代码才能深刻理解代码含义，尽管不能在第一次阅读时就理解全部代码，但掌握代码的框架结构也是必须的。
2. 多查看手册，无论是Ubuntu、vim还是GCC，GDB，查阅手册是快速了解需要使用的工具或命令的最快方法。
3. 调试时多使用printf，assert或log来进行程序的调试，不要花大量的时间来肉眼检查代码错误，如果不能很快看出代码错误，就在可能出错的环节都用printf打印信息，再跟随信息变化查找Bug位置，如：在某个函数的调用前后都打印相关信息，若函数调用完后的结果不符合预期，则可锁定Bug就在此函数中。
4. 尽可能多地考虑特殊情况，包括编写代码和进行测试，并且尽可能为特殊情况的处理和异常输可定位的信息，assert也可，但assert会终止NEMU运行，倘若某些特殊情况的处理不必终止整个程序，就尽量不用assert。
5. 实验阶段全部完成后书写实验报告时梳理全部实验内容可以加深对代码的理解，也有助于发现原先所写的代码中尚未发现的Bug或是想到新的代码优化思路。