PA1实验报告

1811***_秦凧

Part1

1.1实验目的

1. 学习git和vim的使用
2. 了解menu架构
3. 了解寄存器的结构和实现方式
4. 学习简易调试器的搭建

1.2 实验内容

1. 实现正确的寄存器结构
2. 实现单步执行、打印寄存器、扫描内存指令

1.3 实验步骤

1.3.1 实现正确寄存器结构

用于实现模拟寄存器的结构被定义在 `nemu/include/cpu/reg.h` 中，框架中原始的寄存器结构定义有误，会导致nemu出现 `assertion fail` 错误，我们需要修改寄存器结构定义是nemu能正确运行。

查看 `nemu/include/cpu/reg.h` 中原始的定义如下，其中寄存器结构定义中使用了struct，这导致无法通过 `gpr` 数组和寄存器名访问同一内存，也无法单独方位寄存器的高位和低位。为解决这一问题，我们修改寄存器结构的定义，使用union结构，union联合体中定义的变量共享同一段内存，如此在 `CPU_state` 中我们就可以使用寄存器名和 `gpr` 数组访问同一段内存，也可以通过 `_32` 、 `_16` 、 `_8` 访问寄存器的高位和低位。修改后的代码如下：

```
typedef struct {
  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;
} CPU_state;
```

1.3.2 实现单步执行、打印寄存器、扫描内存指令

实验指导书中已经指出nemu是如何解析命令的：使用 `readline()` 函数从键盘上读入命令，并通过一系列的字符串处理函数来完成解析命令工作。 实现题目中的三向指令需要在 `nemu/src/monitor/debug/ui.c` 中补全代码，该文件中已经提供了 `cmd_help` 等命令的实现样例，只需要仿照提供的样例编写代码即可。

实现上述指令首先要补充 `cmd_table` 中的内容，添加命令名、命令描述和对用的处理函数名。之后根据样例，在处理函数中获取命令参数可以使用 `strtok` 函数，该函数原型为 `char *strtok(char s[], const char *delim)`，其将一个字符串分解为一组字符串，参数中 `s` 为要分解的字符串， `delim` 为分隔符字符串。

以下介绍各个处理函数的具体实现

1.3.2.1 单步执行

在单步执行指令 `si` 的处理函数中，首先使用 `strtok` 函数获取指令参数，如果获取的参数为空，该指令默认单步执行一条指令后暂停；如果获取到指令参数，首先将字符串参数转化成数值N，之后执行N条指令后暂停。指令的执行可以仿照样例中 `cmd_c` 函数，使用 `cpu_exec` 实现。具体代码如下：

```
static int cmd_si(char *args) {
  char *arg = strtok(NULL, " ");
  if (arg == NULL) {
    cpu_exec(1);
  }
  else {
    int n = 0;
    for (int i = 0; args[i]; i++) {
      n = n * 10 + args[i] - 48;
    }
    cpu_exec(n);
  }
  return 0;
}
```

1.3.2.2 打印寄存器

在 `info` 指令中处理函数中，使用 `strtok` 函数获取指令参数，如果参数为空输出错误信息并返回，否则根据具体参数进行不同操作：

- r-打印寄存器状态：只需使用 `printf` 语句打印寄存器值即可；
- w-打印监视点信息：调用负责打印监视点信息的函数 `printWatchpointInfo` ，该函数将在part3中与监视点功能一同实现；
- 其他：输出错误信息并返回。

代码如下：

```
static int cmd_info(char *args) {
  char *arg = strtok(NULL, " ");
  if (arg == NULL) {
    printf("info r: print register status. info w: print monitor information.");
```

```
    return -1;
  }
  else {
    switch (args[0]){
      case 'r': {
        //eax, ecx, edx, ebx, esp, ebp, esi, edi, eip
        printf("eax\t%#x\n", cpu.eax);
        ... ... //打印其他寄存器
        printf("eip\t%#x\n", cpu.eip);
        return 0;
      }
      case 'w':{
        printWatchpointInfo();
        return 0;
      }
      default:{
        printf("info r: print register status. info w: print monitor
information.\n");
        return -1;
      }
    }
  }
  return 0;
}
```

1.3.2.3 扫描内存

在扫描内存指令 `x` 处理函数中，同样首先获得指令的两个参数并检查参数是否有误，有误则输出错误信息并返回，否则开始处理指令参数，将第一个参数N转化为数值，并计算第二个表达式参数的值，参数二表达式的求值使用函数 `expr` ，该函数的功能将在part2中实现，调用 `expr` 后首先要根据该函数执行结果判断表达式是否有误；有误输出错误信息并返回，否则开始读取内存，实验指导书中提到"在客户程序运行的过程中,总是使用 `vaddr_read()` 和 `vaddr_write()` 访问模拟的内存"，因此在读取内存时我们调用 `vaddr_read` 函数，并传入起始地址和要读取的内存数目。代码如下：

```
static int cmd_x(char *args) {
  char *chOutNum = strtok(NULL, " ");
  char *e = strtok(NULL, " ");
  if (chOutNum == NULL || e == NULL) {
    printf("x N EXPR. Find out the value of EXPR, take the result as the starting
memory address, and output n consecutive 4 bytes in hexadecimal form the starting
memory. eg: x 10 $esp\n");
    return -1;
  }
  //计算开始输出的地址
  bool success = true;
  uint32_t addr = expr(e, &success);
  if (success == false) {
    printf("'ui.c' wrong expr.\n");
    return -1;
  }
```

```
  //计算输出的字节数并输出连续的N个4字节
  int numLen = strlen(chOutNum);
  int intOutNum = 0;//N
  for (int i = 0; i < numLen; i++) {
    intOutNum = intOutNum * 10 + chOutNum[i] - 48;
  }
  printf("%#x\t%#x\n", addr, vaddr_read(addr, 4*intOutNum));
  return 0;
}
```

1.4 问答

1.4.1 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?

cpu_exec 函数的参数类型是无符号数 uint64_t ，传入-1相当于传入 uint64_t 的最大值，即让程序执行 uint64_t 的最大值条指令后停下，因为一般的程序不会有那么多条指令，也就相当于让程序一直执行直到结束。

1.4.2 谁来指示程序的结束?

通过查阅资料，程序并不总是在 main 函数返回处结束。在Linux系统中，main函数执行完后，还需要调用 atexit 注册的函数，以及 __attribute__ （destructor）标记的函数，最后进行 exit 系统调用，系统会清理掉所有的资源，并且等待父进程调用 wait 获取状态（main里面的return 0）。

Part2

2.1实验目的

1. 复习词法分析
2. 复习语法分析

2.2 实验内容

1. 实现词法分析
2. 实现表达式求值
3. 实现表达式求值指令

2.3 实验步骤

2.3.1 实现词法分析

此方法分析就是是辈出表达式中的单元即token，在pa1中我们使用正则表达式识别token。首先需要使用正则表达式分别编写用于识别这些token类型的规则，示例代码中已经给出了空格和+的识别方法，并给出识别token的代码，只需要仿照示例编写其他识别规则即可。增加规则主要在 expr.c 开头的 enmu 部分和 rules[] 部分添加，添加后内容如下：

```
enum {
  TK_NOTYPE = 256,
```

```
  TK_NUM,//数字
  TK_LBRACKET, TK_RBRACKET, // ( )
  TK_ADD, TK_SUB, TK_MUL, TK_DIV, //+-*/
  TK_DEREFERENCE, TK_MINUS, //解引用，负号
  TK_G, TK_L, TK_GE, TK_LE, //> < >= <=
  TK_EQ, TK_NEQ, // == !=
  TK_AND, TK_OR, TK_NOT, // && || !
  TK_REG,//寄存器
  TK_ID//变量标识符
};

static struct rule {
  char *regex;
  int token_type;
} rules[] = {
  {" +", TK_NOTYPE},
  {"0|[1-9][0-9]*", TK_NUM},
  {"\\$(e[a,b,c,d]x|e[s,b,i]p|e[s,d]i)", TK_REG},
  {"[a-zA-Z][0-9a-zA-Z_]*", TK_ID},
  {"\\(", TK_LBRACKET}, {")", TK_RBRACKET},
  {">=", TK_GE}, {">=", TK_LE},
  {"==", TK_EQ}, {"!=", TK_NEQ}, {"!", TK_NOT},
  {"\\*", TK_MUL}, {"\\/", TK_DIV}, {"\\+", TK_ADD}, {"-", TK_SUB},
  {">", TK_G}, {"<", TK_L},
  {"&&", TK_AND}, {"||", TK_OR}
};
```

除了补充识别规则外，还需要按照提示补全 `make_token` 函数的内容，该函数用于根据上述编写规则识别token把那个储存起来，其中识别token部分已经给出，我们只需要补全后续的储存token部分即可，即将 `rules` 数组中识别出的token信息转存到 `tokens` 数组中，其中部分token如运算符等只需要转存token的类型即可，对于数字、寄存器名、变量名则需要将其对应的字符串一并转存，通知转存时还要注意不能超出 `token.str` 的最长限制，这里对超出最长限制的字符串的处理方式是输出错误信息之后返回false。补充的部分代码如下：

```
... ...
tokens[nr_token].type = rules[i].token_type;
switch (rules[i].token_type) {
  case TK_NUM:
  case TK_ID:
  case TK_REG:
    if (substr_len <= MAX_TOKEN_LEN){
      for (int j = 0; j < substr_len; j++) {
        tokens[nr_token].str[j] = *(substr_start+j);
      }
    }
    else{
      printf("The token is too long(need to be < %d).\n", MAX_TOKEN_LEN);
      return false;
    }
    break;
}
nr_token++;
```

```
  break;
  ... ...
```

2.3.2 实现表达式递归求值

我们借助BNF来进行表达式的递归求值，在2.3.1中我们已经获得了表达式中得的token并储存在数组中，按照指导书中的提示，进行递归表达式求值时可以使用该数组的下标表示要就散的部分表达式。实现递归求值有以下几个部分：补全对外的接口函数 `expr` ，编写递归函数 `eval` ，编写与用于检查左右括号匹配的函数 `check_parentheses` ，编写用于查找 `dominant operator` 的函数 `findDominantOp` ，下面将以此介绍这几个部分的实现过程。

2.3.2.1 完善 `expr` 函数

该函数用于向外提供计算表达式的接口，在函数内首先调用 `make_token` 函数识别表达式中的token，成功继续进行下一步，否则将 `*success` 的值改为false并返回。之后就是我们需要补充的部分，表达式中的部分算符-和*既可以作为双目运算符的减法和乘法，也可以作为单目运算符的取负和取地址，在词法分析中我们无法分辨这两个运算符的正确函数，故全部当做减法和乘法处理了，因此在调用 `expr` 函数开始递归求值前需要首先修正token数组中可能出现的错误的-和*，我们通过检查这两个运算符的前一个token来判断其含义：如果这两个运算符是数组中的第一个token或他们前面是洽谈运算符，则它们是单目的否则是双目的。如果判断出它们是单目运算符则更改该token的type为响应单目运算符的type。更正-和*之后调用 `eval` 函数开始递归求值，具体代码如下：

```
uint32_t expr(char *e, bool *success) {
  if (!make_token(e)) {
    *success = false;
    return 0;
  }
  for (int i = 0; i < nr_token; i++) { //判断单目运算符*-
    if (tokens[i].type == TK_MUL || tokens[i].type == TK_SUB) {
      if (i == 0 ||
          (tokens[i-1].type != TK_NUM && tokens[i-1].type != TK_REG &&
           tokens[i-1].type != TK_RBRACKET && tokens[i-1].type != TK_ID)) {
        if (tokens[i].type == TK_SUB) {
          tokens[i].type = TK_MINUS;
        }
        else {
          tokens[i].type = TK_DEREFERENCE;
        }
      }
    }
  }
  return eval(0, nr_token - 1, success);
}
```

2.3.2.2 编写 `eval` 函数

`eval` 函数用于对表达式递归求值，其参数为：

1. `int p`：表达式最左端在 `tokens` 数组中的下标；
2. `int q`：表达式最右端在 `tokens` 数组中的下标；
3. `bool *success`：指示表达式求值是否成功。

根据指导指导书中给出的框架，`eval` 函数中应分为四种情况：

1. p>q时表达式有误，返回错误信息；
2. p==q时说明只有单独的一个token，该token应是一个可用于计算的数字或储存数字的寄存器名或字符串；
3. 如果p和q对应的token是一对匹配的括号，则直接丢弃括号；
4. 除以上三种情况，第四种情况中首先获得 `dominant operator`，之后分别递归计算其两边的表达式的结果，并将两边的结果进行相应运算。

pa1中已经给出了 `eval` 函数的框架，只需要填充该框架即可。需要注意的是在上述第二种情况即p==q时，需要判断该单独的运算数是数字还是储存在寄存器或变量中中内容，对于变量值的获取需要使用到符号表或其他辅助结构，此处还不能实现，暂时略过。其他地方使用到的 `check_parentheses` 函数和 `findDominantOp` 函数将在后续内容中说明。`eval` 函数代码如下：

```
uint32_t eval(int p, int q, bool* success) {
  if (p > q) {//表达式有误
    *success = false;
    return -1;
  }
  else if (p == q) {//单独的数字/寄存器/id
    if (tokens[p].type == TK_NUM) {//数字
      int numLen = strlen(tokens[p].str);
      int num = 0;
      for (int i = 0; i < numLen; i++) {
        num = num*10 + tokens[p].str[i] - '0';
      }
      return num;
    }
    else if (tokens[p].type == TK_REG) {//寄存器
      switch (tokens[p].str[2]) {
        case 'a': return cpu.eax;
        case 'b':
          if (tokens[p].str[3] == 'x')
            return cpu.ebx;
          else if (tokens[p].str[3] == 'p')
            return cpu.ebp;
          else {
            *success = false;
            return -1;
          }
        case 'c': return cpu.ecx;
        case 'd':
          if (tokens[p].str[3] == 'x')
            return cpu.edx;
          else if (tokens[p].str[3] == 'i')
            return cpu.edi;
          else {
```

```
            *success = false;
            return -1;
          }
        case 's':
          if (tokens[p].str[3] == 'p')
            return cpu.esp;
          else if (tokens[p].str[3] == 'i')
            return cpu.esi;
          else {
            *success = false;
            return -1;
          }
        case 'i': return cpu.eip;
        default:
          *success = false;
          return -1;
      }
    }
    else if (tokens[p].type == TK_ID) {//id
      //TODO:
      return 0;
    }
  }
  else if (check_parentheses(p, q, success)) {
    if (*success == false) {//括号不匹配，停止递归直接返回
      return -1;
    }
    return eval(p+1, q-1, success);//去掉括号继续递归
  }
  else {
    int opPos = findDominantOp(p, q);
    switch (tokens[opPos].type) {
      case TK_ADD: return eval(p, opPos-1, success) + eval(opPos+1, q, success);
      case TK_SUB: return eval(p, opPos-1, success) - eval(opPos+1, q, success);
      case TK_MUL: return eval(p, opPos-1, success) * eval(opPos+1, q, success);
      case TK_DIV: return eval(p, opPos-1, success) / eval(opPos+1, q, success);
      case TK_G:   return eval(p, opPos-1, success) > eval(opPos+1, q, success);
      case TK_L:   return eval(p, opPos-1, success) < eval(opPos+1, q, success);
      case TK_GE:  return eval(p, opPos-1, success) >= eval(opPos+1, q, success);
      case TK_LE:  return eval(p, opPos-1, success) <= eval(opPos+1, q, success);
      case TK_EQ:  return eval(p, opPos-1, success) == eval(opPos+1, q, success);
      case TK_NEQ: return eval(p, opPos-1, success) != eval(opPos+1, q, success);
      case TK_AND: return eval(p, opPos-1, success) && eval(opPos+1, q, success);
      case TK_OR:  return eval(p, opPos-1, success) || eval(opPos+1, q, success);
      case TK_NOT: return !eval(opPos+1, q, success);
      case TK_MINUS: return -eval(opPos+1, q, success);
      case TK_DEREFERENCE: return *(int*)(eval(opPos+1, q, success));
      default:
        *success = false;
        assert("Wrong in 'expr.c' eval().");
        return -1;
    }
  }
}
```

2.3.2.3 编写 `check_parentheses` 函数

`check_parentheses` 函数用于检查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，匹配则返回true，否则说明表达式有误，返回错误信息。该函数的参数与 `eval` 函数相同。

检查括号是否匹配可以使用栈的结构，在这里为简便起见，我们只使用一个变量 `simuStack` 模拟简易栈，从p到q逐一检查token类型，如果是左括号就将其入栈，即将 `simuStack` 的值加1，如果是右括号将一个左括出栈，即将 `simuStack` 的值减1。每读取完一个token都检查栈的情况，如果 `simuStack` 为0说明此时栈为空，如果栈为空时正好已检查完p到q的所有token，说明表达式左右的括号是一对且表达式无误，返回true；如果此时还未检查为所有token，说明表达式左右的括号不是一对，返回false；如果某一次检查栈是发现 `simuStack` 的值小于0，说明表达式有误，将 `*success` 的值置为false，但是返回true，此处返回true是因为 `eval` 中会检查 `*success` 的值，返回true进入if语句可以直接结束递归。具体代码如下：

```
bool check_parentheses(int p, int q, bool* success){//检查括号匹配
  if (tokens[p].type != TK_LBRACKET || tokens[q].type != TK_RBRACKET) {
    return false;
  }
  int simuStack = 0;//模拟栈
  for (int i = p; i <= q; i++) {
    if (tokens[i].type == TK_LBRACKET) {
      simuStack++;
    }
    else if (tokens[i].type == TK_RBRACKET) {
      simuStack--;
    }
    if (simuStack == 0 && i != q) { //最左最右括号不是同一对
      return false;
    }
    if (simuStack < 0) {//表达式有误
      *success = false;
      return true; //为了在eval中停止递归，返回true
    }
  }
  if (simuStack != 0){//左括号数大于右括号，表达式有误
    *success = false;
  }
  return true;
}
```

2.3.2.3 编写 `findDominantOp` 函数

`findDominantOp` 函数用于寻找 `dominant operator` 即表达式人工求值时最后一步进行运行的运算符。这里寻找 `dominant operator` 采取的方法是大小不同的整数代表每一个运算符的优先级，用变量 `opType` 代表此时找到的优先级最低的运算符，用 `opPos` 表示该运算符的位置，之后遍历表达式，对每一个运算符比较该运算符与 `opType` 的优先级，优先级低于 `opType` 则更新 `opType` 和 `opPos` ，最后返回 `opPos` 。需要注意的是，因为之前已经将表达式两侧的匹配的括号丢弃（ `eval` 函数第三种情况），故传入 `findDominantOp` 函数中的部

分表达式的左右不会是匹配的括号，故当遇到左括号时可以将其后的token丢弃（不检查），直到遇到下一个右括号，这样可以提高效率。函数代码如下：

```
int findDominantOp(int p, int q) {
  int opType = -1, opPos = 0;// 0 取地址 负 !, 1 * /, 2 + -, 3 > < >= <=, 4 == !=,
5 &&, 6 ||
  for (int i = p; i <= q; i++) {
    int tempType = 0;//运算符类型
    switch (tokens[i].type) {
      case TK_DEREFERENCE:
      case TK_MINUS:
      case TK_NOT:
        tempType = 0; break;
      case TK_MUL:
      case TK_DIV:
        tempType = 1; break;
      case TK_ADD:
      case TK_SUB:
        tempType = 2; break;
      case TK_G:
      case TK_L:
      case TK_GE:
      case TK_LE:
        tempType = 3; break;
      case TK_EQ:
      case TK_NEQ:
        tempType = 4; break;
      case TK_AND:
        tempType = 5; break;
      case TK_OR:
        tempType = 6; break;
      case TK_LBRACKET:
        while (i <= q && tokens[i].type != TK_RBRACKET) {
          //因前面处理过括号，这里不会是一整个括号，且DominantOp不会在括号中，直接跳过括
号包含的部分
          i++;
        }
      default: continue;
    }
    if (tempType > opType) {
      opType = tempType;
      opPos = i;
    }
  }
  return opPos;
}
```

2.3.3 实现表达式求值指令

实现表达式求值功能之后我们就可以补全 `ui.c` 中的表达式求值指令的处理函数。只需要按照之前的方法获得指令参数，之后调用 `eval` 函数即可，需要注意的是调用函数时要传入一个 `bool*` 类型的参数用于指示表达式

是否计算成功。实现代码如下：

```
static int cmd_p(char *args) {
  char *e = strtok(NULL, " ");
  if (e == NULL) {
    printf("p EXPR. Find the value of EXPR. eg: p $eax+1\n");
    return -1;
  }
  bool success = true;
  uint32_t result = expr(e, &success);
  if (success == false) {
    printf("'ui.c' wrong expr.\n");
    return -1;
  }
  else {
    printf("%d\n", result);
    return 0;
  }
}
```

2.4 问答

2.4.1 实现带有负数的算术表达式的求值

2.4.1.1 负号和减号都是-,如何区分它们?

可以通过该符号的位置以及其前面一个符号来判断其是单目还是双目运算符：如果该符号是表达式中的第一个符号，或该符号前为其他运算符，则其为符号。

2.4.1.2 负号是个单目运算符,分裂的时候需要注意什么?

需要注意单目运算符的符号是右结合的，应对符号后面的数字生效，即将后面的一个数字取负，但前面的内容没有影响。

Part3

3.1实验目的

1. 了解监视点的功能
2. 学习监视点的实现原理

3.2 实验内容

1. 实现监视点功能
2. 实现设置监视点、删除监视点指令

3.3 实验步骤

3.3.1 实现监视点功能

监视点的功能是监视一个表达式的值何时发生变化，并在表达式发生变化时将程序停下。pa1中给出了监视点结构的部分内容以及用于组织使用中的监视点的 `head` ，和用于组织空闲监视点的 `free_` ，并完成了初始化工作。 根据实验指导书的指示，我们需要实现获取新监视点和删除旧监视点两个函数：`new_wp` 和 `free_wp` 。此外，在part1中info指令的处理函数中，对于打印监视点信息的部分，我们调用了 `printWatchpointInfo` 函数，该函数也需要在监视点部分补充实现。最后，为了完成监视点每当表达式的值发生改变就停止程序的功能，还需要实现 `checkWatchpoint` 函数用于检查是否有监视点对应的表达式值改变。以下将介绍各个部分的实现过程。

3.3.1.2 获取监视点的 `new_wp` 函数

`new_wp` 函数从 `free_` 链表中返回一个空闲的监视点结构，该函数首先检查 `free_` 是否为 `NULL` ，是的话说明已经没有空闲的监视点，输出信息并返回 `NULL` ，否则使用 `tempWP` 记录此时的 `free_` 指向的监视点，将 `free_` 指它的下一个监视点，再将 `tempWP` 加入 `head` 链表，最后返回 `tempWP` 。代码如下：

```
WP* new_wp() {
  if (free_ == NULL) {
    printf("There is no more watchpoint.");
    return NULL;
  }
  WP* tempWP = free_;
  free_ = free_->next;//获取一个未使用监视点
  tempWP->next = head;
  head = tempWP;//加入以使用列表
  printf("Get a new watchpoint with id %d.\n", tempWP->NO);
  return tempWP;
}
```

3.3.1.2 删除监视点的 `free_wp` 函数

`free_wp` 函数将一个监视点归还到 `free_` 链表中。指导书中给出的函数原型为 `void free_wp(WP* wp)`，而删除监视点的指令中给出的参数是该监视点的id，如果 `free_wp` 的参数为 `WP*` 类型，则在删除监视点的函数中需要通过监视点id获取该监视点，即需要遍历 `head` 链表。但原本的框架中 `head` 和 `free_` 被定义在 `watchpoint.c` 中，并被 `static` 修饰，如果使用实验指导书中给出的 `free_wp` 函数的原型需要更改 `head` 和 `free_` 的定义，且在 `watchpoint.c` 之外操作 `head` 和 `free_` 不利于保持 `watchpoint.c` 部分的封装性。故在此不使用实验指导书中给出的函数原型，而将 `free_wp` 函数的参数改为要释放的监视点的id，即将根据id查找监视点的工作放入 `free_wp` 函数中。

实现 `free_wp` 时，首先在 `head` 链表中根据id查找要删除的监视点，查找时应注意记录正在对比id的监视点的上一个监视点，找到后将该监视点从 `head` 链表中摘除并放入 `free_` 链表中。如果没有找到参数id对应的监视点则输出错误信息。代码如下：

```
void free_wp(int id) {
  WP* temp = head;
  if (head->NO == id){
    //head是删除的监视点
```

```
    head = head->next;
    temp->next = free_;
    free_ = temp;
    printf("Watchpoint with id %d has been deleted.\n", id);
    return;
  }
  else {
    //查找要删除的监视点
    WP* curPos = head;
    while(curPos->next != NULL) {
      if (curPos->next->NO == id) {
        temp = curPos->next;
        curPos->next = curPos->next->next;
        temp->next = free_;
        free_ = temp;
        printf("Watchpoint with id %d has been deleted.\n", id);
        return;
      }
      curPos = curPos->next;
    }
  }
  printf ("Watchpoint with id %d has not been set.\n", id);
}
```

3.3.1.3 输出监视点信息的 `printWatchpointInfo` 函数

该函数被 `ui.c` 中info指令的处理函数抵用，用于输出各个监视点的信息。大户出的信息包括该监视点的id（`wp.NO`）、该监视点对用的表达式、以及表达式此时的值，为了输出后面两个信息，还需要在 `watchpoint.h` 中监视点结构体中添加用于储存二者的变量，这两个变量在后续实现表达式改变即让程序停止的功能时也会使用到。修改后的监视点结构如下：

```
typedef struct watchpoint {
  int NO;
  struct watchpoint *next;
  char expr[MAX_EXPR_LEN];//用于储存表达式
  uint32_t lastVal;//上一条指令执行之后的值
} WP;
```

函数功能的实现较为简单，只需要遍历 `head` 链表，依次输出链表中储存的监视点信息即可，代码如下：

```
void printWatchpointInfo(){
  WP* curPos = head;
  if (head == NULL){
    printf("There is no watchpoint set.\n");
    return;
  }
  int num = 0;
  while (curPos != NULL) {
```

```
    printf("Watchpoint id: %d\texpr: %s\tnow value: %d\n", curPos->NO, curPos-
>expr, curPos->lastVal);
    num++;
    curPos = curPos->next;
  }
  printf("total num: %d\n", num);
}
```

3.3.1.4 监视点出现变化时停下程序

为了在监视点有变化时通下程序，我们首先要实现 `checkWatchpoint` 函数，该函数用于检查各个监视点对应的表达式是否发生变化，以便于在发生变化时停下程序。检查监视点时遍历 `head` 链表，对于每一个监视点调用 `expr` 函数计算结果，之后与储存在监视点结构中的 `lastVal` 比较，如果有变化则输出提示信息，如果所有的电视监视点都无变化返回false，否则返回true。代码如下：

```
bool checkWatchpoint(){
  bool isChange = false;
  WP* curPos = head;
  while (curPos != NULL) {
    bool success = true;
    uint32_t curVal = expr(curPos->expr, &success);
    if (success == false) {
      printf("cpu-exec.c watchpoint which id %d wrong expr.\n", curPos->NO);
    }
    else if (curPos->lastVal != curVal){
      isChange = true;
      curPos->lastVal = curVal;
      printf("A watchpoint which id is %d and expr is '%s' has been triggered, now
value is %d.\n",
             curPos->NO, curPos->expr, curPos->lastVal);
    }
    curPos = curPos->next;
  }
  return isChange;
}
```

实现了检查监视点的检查函数之后还需要在让程序在监视点出现变化时停下，根据实验指导书，应该在 `src\monitor\cpu-exec.c` 中添加代码。只需要调用上面实现的函数判断返回值，如果返回值为true则将 `nemu_state` 设置为 `NEMU_STOP` 并返回，添加的代碼如下：

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

3.3.2 实现设置监视点、删除监视点指令

3.3.2.1 实现设置监视点指令

设置监视点指令的处理函数中首先获取指令参数，之后计算输入的表达式的值，如果输入的表达式有误，输出错误信息直接返回，否则调用 `new_wp` 函数获取一个空闲的监视点，如果获取失败输出错误信息并返回，否则将表达式和表达式此时的值存入该监视点。代码如下：

```
static int cmd_w(char *args) {
  char *e = strtok(NULL, " ");
  if (e == NULL) {
    printf("w EXPR. Suspend the program execution when the value of expr changes.
eg: w *0x20000\n");
    return -1;
  }
  //计算表达式值
  if (strlen(e) > MAX_EXPR_LEN) {//表达式太长
    printf("The expr is too long, max %d\n", MAX_EXPR_LEN);
    return -1;
  }
  bool success = true;
  int lastVal = expr(e, &success);
  if (success == false) {//表达式有误
    printf("ui.c wrong expr\n");
    return -1;
  }
  //申请和初始化监视点
  WP* wp = new_wp();
  if (wp != NULL) {
    strcpy(wp->expr, e);
    wp->lastVal = lastVal;
    return 0;
  }
  return -1;
}
```

3.3.2.2 实现删除监视点指令

删除监视点首先获取指令参数，将改参数转变为数值类型，并检查该数值大小是否大于监视点总数，是的话输出错误信息，否则调用 `free_wp` 函数删除监视点。代码如下：

```
static int cmd_d(char *args){
  char* numStr = strtok(NULL, " ");
  if (numStr == NULL) {
    printf("d N. Delete monitor point with serial number n. eg: d 2\n");
  }
  //计算要删除的监视点id
  int numLen = strlen(numStr);
  int watchpointID = 0;
```

```
    for (int i = 0; i < numLen; i++){
      watchpointID = watchpointID * 10 + numStr[i] - '0';
    }
    if (watchpointID > NR_WP) {
      printf("Watchpoint id should be less than %d.\n", NR_WP);
      return -1;
    }
    //删除监视点
    free_wp(watchpointID);
    return 0;
  }
```

3.4 问题

3.4.1 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?

static 在此处用于将 wp_pool 等变量隐藏，即保证这些变量不可以在其他文件中通过 extern 关键字获取，避免在其他源文件中引用这些变量导致错误。起到了对其他源文件进行隐藏与隔离错误的作用，有利于模块化程序设计。

3.4.2 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?

是必须的，Intel手册中写到：This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code，即这样的单字节可以重写其他指令，包括单字节指令，而不会覆盖其他指令。在 my_x86 中断点机制不能正常执行，因为中断执行的机制是将设置中断处的代码前一个字节改为 0xcc ，如果替换断点指令的长度不再是 0xcc 而超过1字节，同时被覆盖的指令（设置断点处的指令）长度只有一个字节，将可能被迫覆盖下一条指令的一部分，这将使其发生乱码，并可能产生无效的代码。

3.4.3 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?

会不能正常工作，因为检测断点时是检测指令的首字节是否等于特定值，将断点设置在非首字节会导致检测不到断点。

3.4.4 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?

调试器是一个命令行调试工具，可以设置断点、调试程序、测试bug等，然后模拟器是一个载体，类似虚拟机，是一个独立的系统，不会对电脑本身的系统造成影响，可以兼容不同软件。

gdb和nemu类似，但是gdb是可以直接在函数某一行或是函数入口处设置断点，可以随意查看变量当前的值。

4 必答题

4.1 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：

4.1.1 EFLAGS 寄存器中的 CF 位是什么意思？（P50 3.2）

许多算术指令同时操作有符号整数和无符号整数,CF包含与无符号整数相关的信息。如果整数是无符号的，可以在这些算术操作之一之后测试CF，以确定该操作是否需要在目标操作数的高阶位置进位或借位，是的话将CF置位。完成CF的设置。

4.1.2 ModR/M 字节是什么？（P241-243 17.2.1）

ModR/M由Mod，Reg/Opcode，R/M三部分组成。Mod是前两位，提供寄存器寻址和内存寻址，Reg/Opcode为345位，如果是Reg表示使用哪个寄存器，Opcode表示对group属性的Opcode进行补充；R/M为678位，与mod结合起来查图得8个寄存器和24个内存寻址。

4.1.3 mov 指令的具体格式是怎么样的？（P347）

格式是 MOV DEST SRC ，将 DEST 中的内容储存到 SRC中。

4.2

1. 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?
    - 共4949行代码。
2. 你是使用什么命 令得到这个结果的?
    - 使用的命令是：

```
find . -type f | xargs cat | wc -l
```

3. 和框架代码相比,你在PA1中编写了多少行代码?
    - 编写了4949-4452=497行代码。
4. 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"?
    - 使用命令 git checkout pa0 切换git分支到pa0中。
5. 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?
    - 共3562行代码。

4.3 请解释 gcc 中的 -Wall 和 -Werror 有什么作用?为什么要使用 -Wall 和 -Werror?

-Wall 使GCC产生尽可能多的警告信息，取消编译操作，打印出编译时所有错误或警告信息。

-Werror 要求GCC将所有的警告当成错误进行处理，取消编译操作。

使用 -Wall 和 -Werror 就是为了找出存在的错误，尽可能地避免程序运行出错，优化程序。