目录

目录

概述

选择的指导书及代码版本，平台

选择的是 nju19 的 pa 指导书，原因是课程给到的 17 版本采用 32 位系统，虚拟机连不上 vscode 啊（而且虚拟机莫名崩溃），询问后可以采用较新版本，就采用了 19 的代码框架。

19 版的可以选择不同的架构来实现，这里仍然选择的是 x86 架构的

实验内容

完善寄存器结构
完善调试器其中包括:
单步执行;
打印程序状态
表达式求值
扫描内存(需要表达式求值作为前置)
设置监视点(需要表达式求值作为前置)
删除监视点

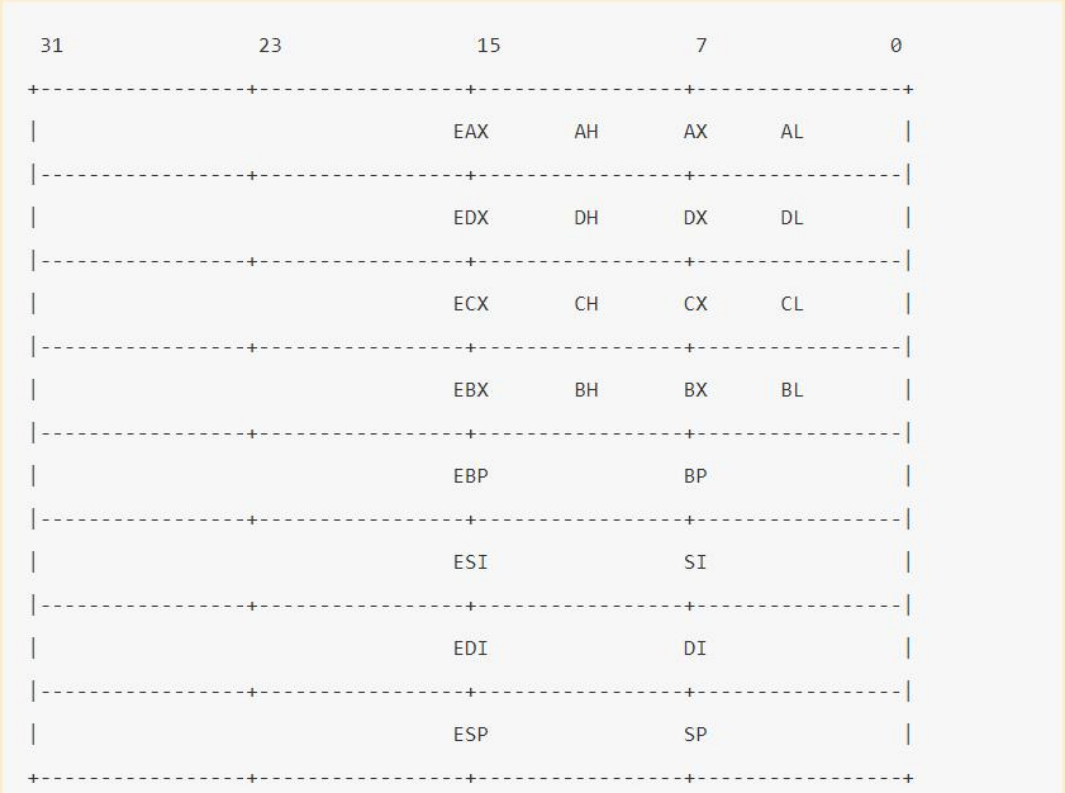
回答相关问题

完善寄存器结构

x86 寄存器的结构:

x86 除 pc 寄存器外，有 32 位，16 位，8 位寄存器各八个，其结构如下:

x86的寄存器结构如下:



其中

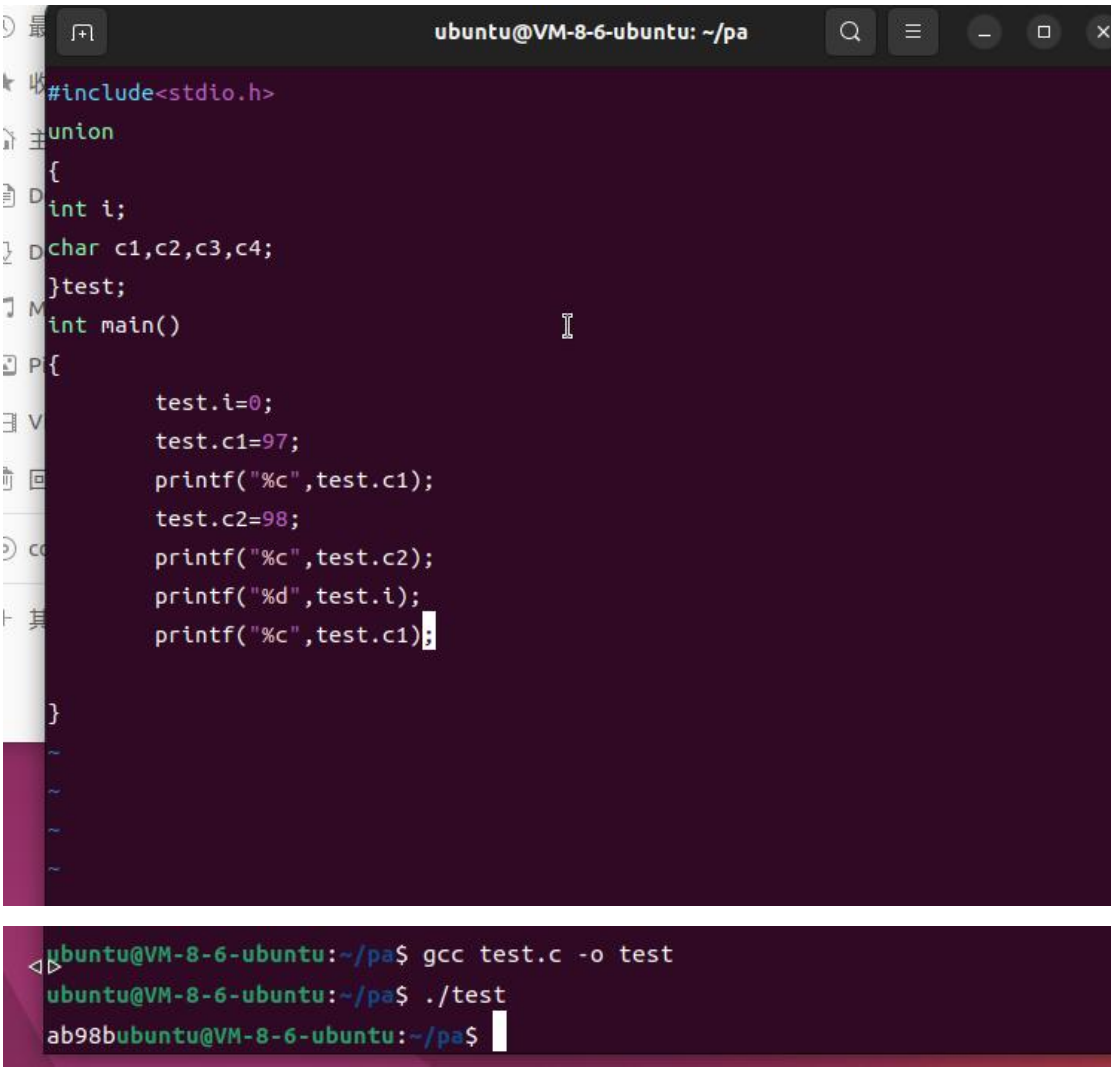
- `EAX`，`EDX`，`ECX`，`EBX`，`EBP`，`ESI`，`EDI`，`ESP`是32位寄存器;
- `AX`，`DX`，`CX`，`BX`，`BP`，`SI`，`DI`，`SP`是16位寄存器;
- `AL`，`DL`，`CL`，`BL`，`AH`，`DH`，`CH`，`BH`是8位寄存器.

但它们在物理上并不是相互独立的, 例如 `EAX` 的低16位是 `AX`，而 `AX` 又分成 `AH` 和 `AL`.

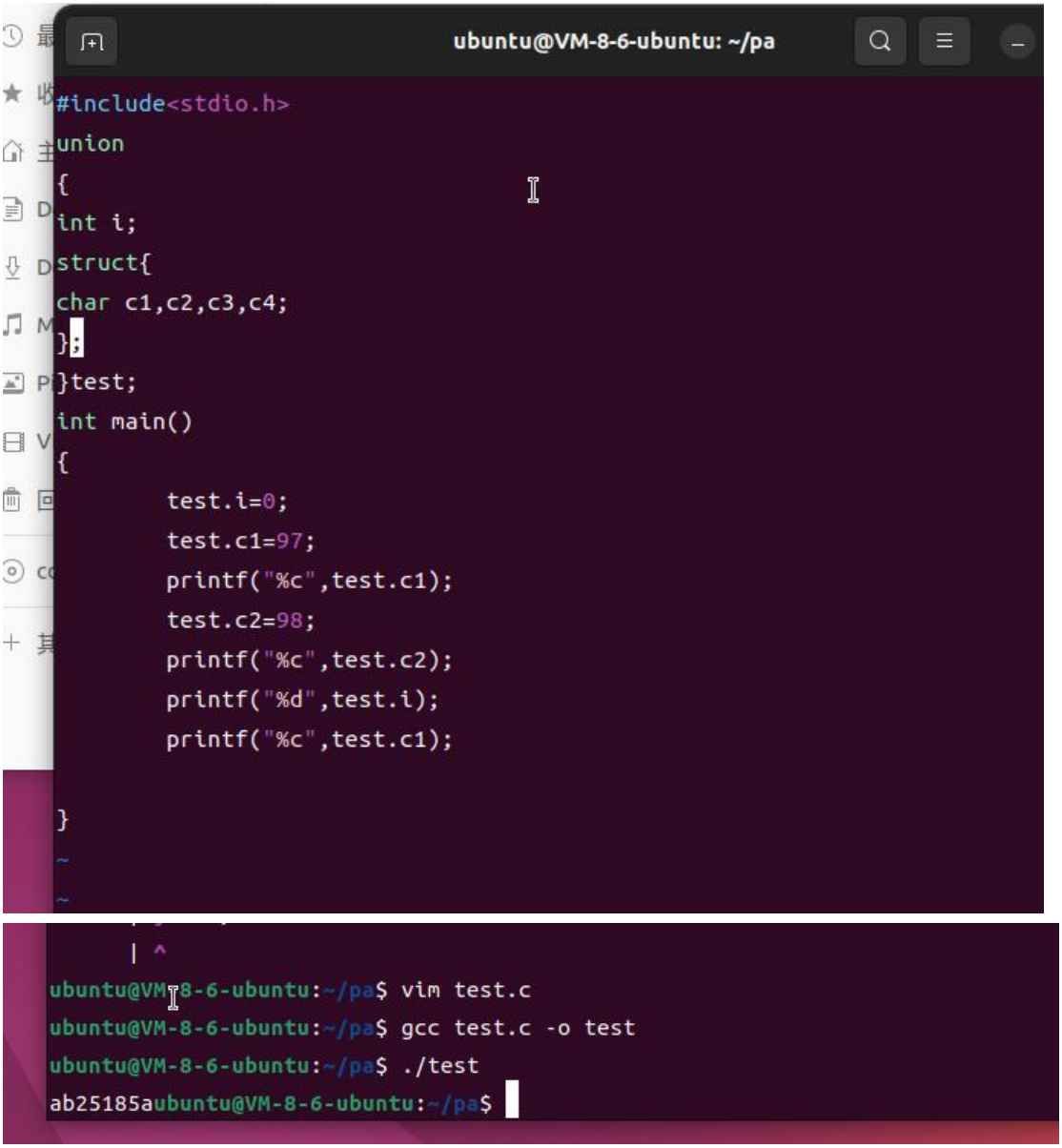
那么我们为了模拟这种内存共用的效果，就需要使用 union

union 结构:

与 Struct 不同，Union 中的各变量互斥，共享同一内存首地址，寄存器的结构实现就是基于该特性实现的。
关于 union 的用法:



如上图，我们看到如此写的话 c1,c2,c3,c4 都是共享的低 8 位的地址



使用 struct 和 union 嵌套就可以很好的让 c1,c2,c3,c4 是独立的内存，但和 i 又是不独立的。

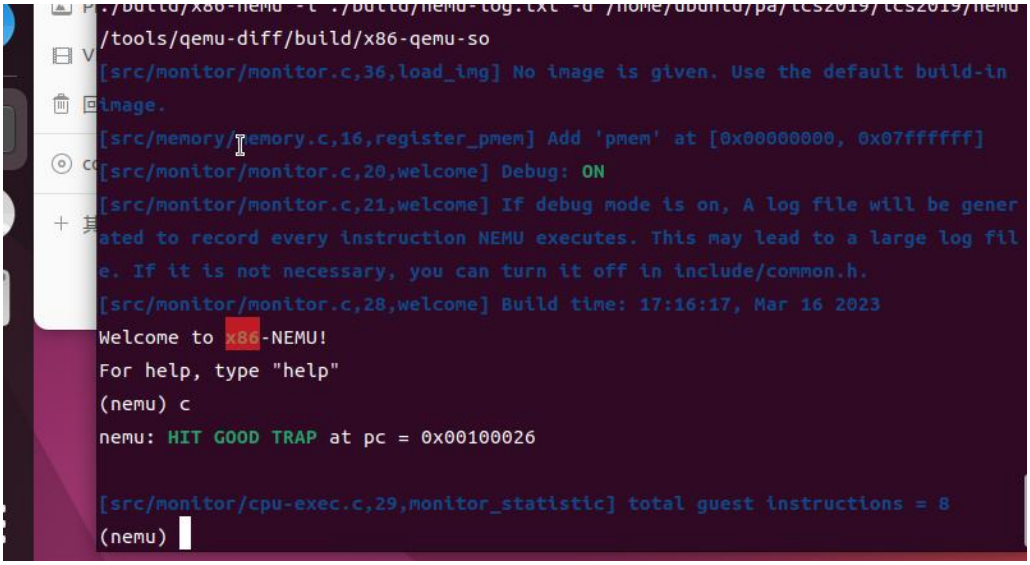
寄存器具体完善

于是根据上面对 x86 寄存器结构的了解和 union 结构的了解，我们如此完善寄存器结构:

```c
typedef struct {
  union{
  union {
    uint32_t _32;
    uint16_t _16;
    uint8_t _8[2];
  } gpr[8];

  /* Do NOT change the order of the GPRs' definitions. */

  /* In NEMU, rtlreg_t is exactly uint32_t. This makes RTL instructions
   * in PA2 able to directly access these registers.
   */
   struct{
  rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
   };
  };
  vaddr_t pc;

} CPU_state;
```

运行结果:

成功跑通



单步执行

寄存器结构完成后，我们正式开始完善调试器，首先我们完善单步调试功能。

调试器命令是如何调用的

想要完善功能，我们首先需要知道他是如何调用的，我们的指令是在 ui.c 的 mainloop 函数中调用的，我们首先发

现其分辨指令类型是通过

```
/* extract the first token as the command */
char *cmd = strtok(str, " ");
```

将传入的字符串切割得到第一个空格前的内容来辨别的。
然后通过

```
 which may need further parsing
 */
char *args = cmd + strlen(cmd) + 1;
if (args >= str_end) {
  args = NULL;
}
```

来将分割掉命令类型的字符串作为将要执行的命令的函数的参数。

```
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;
  }
}
```

接下来通过遍历 cmd_table 结构，判断 cmd 字符串是否和某个命令匹配，进而调用其对应的处理函数，来执行命令。
此外我们发现，当函数的执行返回值<0 时，我们会退出 mainloop。
那么我们想要完善我们调试器的功能，首先需要将我们要添加的功能补充至 cmd_table

cmd_table 结构

```
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", "格式 si [N] 使用举例 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",cmd_si},
  { "info", "打印程序状态 格式 info SUBCMD 使用举例 info r打印寄存器状态info w打印监视点信息", cmd_info },
  { "p", "表达式求值 格式 p EXPR 举例: p $eax + 1 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", cmd_p },
  { "x", "扫描内存 格式 x N EXPR 使用举例 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", cmd_x },
  { "w", "设置监视点  格式 w EXPR 举例: w *0x2000 当表达式EXPR的值发生变化时，暂停程序执行", cmd_w },
  { "d", "删除监视点  格式 d N 举例: d 2 删除序号为N的监视点备注", cmd_d },

  /* TODO: Add more commands */

};
```

根据已经写好的一些指令内容和 mainloop 的调用，我们可以看到 cmd_table 包含三个部分，字符串，name 其为我们的指令对应的字符串，description 为对应指令的描述，这将在调用 help 的时候使用，一个返回值为 int，参数为 char*的函数指针，这指向我们实现对应指令的函数。

实现单步调试

根据上面分析，我们首先需要创建
static int cmd_si(char*args)函数，并将对应信息补充至 cmd_table（后续函数都需要有类似操作，不再赘述）

```
static int cmd_si(char *args) {
  char* arg=strtok(args," ");
  uint64_t N=0;
  if(arg==NULL)
```

```
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", "格式 si [N] 使用举例 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",cmd_si},
  { "info", "打印程序状态 格式 info SUBCMD 使用举例 info r打印寄存器状态info w打印监视点信息", cmd_info },
  { "p", "表达式求值 格式 p EXPR 举例: p $eax + 1 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", cmd_p },
  { "x", "扫描内存 格式 x N EXPR 使用举例 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", cmd_x },
  { "w", "设置监视点  格式 w EXPR 举例: w *0x2000 当表达式EXPR的值发生变化时, 暂停程序执行", cmd_w },
  { "d", "删除监视点  格式 d N 举例: d 2 删除序号为N的监视点备注", cmd_d },

  /* TODO: Add more commands */
```

接下来我们需要实现我们的单步调试

```
static int cmd_si(char *args) {
  char* arg=strtok(args," ");
  uint64_t N=0;
  if(arg==NULL)
  {
    N=1;
  }
  else{
    N=atoi(arg);

  }
cpu_exec(N);
  return 0;
```

很简单，我们只需要将字符串转换为数字 N，如果没有参数，则将 N 赋值为 1，然后调用 cpu__exec(N)来执行 N 步即可

打印程序状态

实现打印程序状态

参数为空的时候我们需要输出错误提示，当参数为 r 时候我们直接将 cpu 对应寄存器的内容输出即可，当参数为 w 的时候我们调用 print_wp 函数，这将在后面监视点部分介绍，此处不展开，其他情况提示参数错误

```c
static int cmd_info(char *args) {
  char* arg=strtok(args," ");
  if(arg==NULL)
  {
    printf("cmd_info没有参数");
    return 0;
  }
  if (strcmp(arg,"r")==0)
  {
        printf("eax is %x\n",cpu.eax);
        printf("ecx is %x\n",cpu.ecx);
        printf("edx is %x\n",cpu.edx);
        printf("ebx is %x\n",cpu.ebx);
        printf("esp is %x\n",cpu.esp);
        printf("ebp is %x\n",cpu.ebp);
        printf("esi is %x\n",cpu.esi);
        printf("edi is %x\n",cpu.edi);
        printf("pc is %x\n",cpu.pc);
        return 0;
  }
  if(strcmp(arg,"w")==0)
  {
    print_wp();
    return 0;
  }
  printf("info命令未输入参数\n");
  return 0;
}
```

运行结果

```
(nemu) info r
eax is 1234
ecx is 21e2cc31
edx is 33d18583
ebx is 5ff1962e
esp is 77229ac1
ebp is 63fb9151
esi is 57baf76d
edi is 60bb122e
pc is 100005
```

表达式求值

在这里我们有两大步分别是
1：识别出表达式中的单元
2：根据表达式的归纳定义进行递归求值

其中第一步我们通过正则表达式来辅助实现

词法分析

token 的构建

这部分代码在 src/monitor/debug/expr 中
我们要补充我们需要实现的 token 以及其对应的匹配规则，根据监视点部分的指导手册，我们具体要实现能满足以下表达式的 token

```
<expr> ::= <decimal-number>
  | <hexadecimal-number>    # 以"0x"开头
  | <reg_name>              # 以"$"开头
  | "(" <expr> ")"
  | <expr> "+" <expr>
  | <expr> "-" <expr>
  | <expr> "*" <expr>
  | <expr> "/" <expr>
  | <expr> "==" <expr>
  | <expr> "!=" <expr>
  | <expr> "&&" <expr>
  | "*" <expr>              # 指针解引用
```

具体代码如下:
token:

```
enum {
  TK_NOTYPE = 256,
  TK_HEX,
  TK_DEC,
  TK_REG,
  TK_EQ,
  TK_NEQ,
  TK_AND,
  TK_OR,
  TK_DEREF


  /* TODO: Add more token types */
```

rule:

```
} rules[] = {

  /* TODO: Add more rules.
   * Pay attention to the precedence level of different rules.
   */

  {" +", TK_NOTYPE},    // spaces
  {"0[xX][0-9A-Fa-f]+",TK_HEX},
  {"0|[1-9][0-9]*",TK_DEC},
  {"\\$(eax|ebx|ecx|edx|esp|ebp|esi|edi|pc|ax|bx|cx|dx|sp|bp|si|di|al|cl|dl|bl|ah|ch|dh|bh)",TK_REG},
  {"==", TK_EQ},
  {"!=", TK_NEQ},
  {"&&",TK_AND},
  {"\\|\\|",TK_OR},
  {"\\+", '+'},
  {"-", '-'},
  {"\\*", '*'},
  {"\\/", '/'},
  {"\\(", '('},
  {"\\)", ')'}
};
```

其中解引用是一种比较特殊的类型，因为需要将它与乘法区分。而乘法是双目运算符，解引用为单目，那么我们可以利用此特征得出：

当'*'出现在整个表达式的头部时，他一定是解引用。

当'*'的左侧不是')',10 进制，寄存器，16 进制的时候，一定是解引用。

其他时候'*'的左右两侧都为 expr，一定是表示乘法

据此我们可以得到处理的代码

```
uint32_t expr(char *e, bool *success) {
  if (!make_token(e)) {
    *success = false;
    return 0;
  }
  if(tokens[0].type=='*')
  {
    tokens[0].type=TK_DEREF;
  }
  for(int i=1;i<nr_token;i++)
{
  if(
  tokens[i].type=='*'&&
  tokens[i-1].type!=TK_HEX&&
  tokens[i-1].type!=TK_DEC&&
  tokens[i-1].type!=TK_REG&&
  tokens[i-1].type!=')'
  )
  {
    tokens[i].type=TK_DEREF;
  }
```

类型和值的保存

对于寄存器，16 进制，10 进制这种非符号 token，我们在保留其类型的同时，我们还要保留他的数值，处理部分如下：

```
case TK_NOTYPE:
  break;
case TK_DEC:
  if(substr_len>=32)
  {
    Log("错误的表达式参数, 超过了32字节! \n");
    return false;
  }
  strncpy(tokens[nr_token].str,substr_start,substr_len);
  *(tokens[nr_token].str+substr_len)='\0';
  tokens[nr_token].type=TK_DEC;
  nr_token++;
  //printf("%s\n",tokens[nr_token].str);
  break;
case TK_HEX:
  if(substr_len>=32)
  {
    Log("错误的表达式参数, 超过了32字节! \n");
    return false;
  }
  strncpy(tokens[nr_token].str,substr_start+2,substr_len-2);
  *(tokens[nr_token].str+substr_len-2)='\0';
  tokens[nr_token].type=TK_HEX;
  nr_token++;
  //printf("%s\n",tokens[nr_token].str);
  break;
case TK_REG:
  strncpy(tokens[nr_token].str,substr_start+1,substr_len-1);
```

而对于+-*/之类的符号我们只需保存其类型即可：

```
      case '(':
      tokens[nr_token++].type='(';
      break;
      case ')':
      tokens[nr_token++].type=')';
      break;
      case '+':
      tokens[nr_token++].type='+';
      break;
      case '-':
      tokens[nr_token++].type='-';
      break;
      case '*':
      tokens[nr_token++].type='*';
      break;
      case '/':
      tokens[nr_token++].type='/';
      break;
      case TK_EQ:
      tokens[nr_token++].type=TK_EQ;
      break;
      case TK_NEQ:
```

表达式求值

表达式求值的整体思路：

根据表达式的归纳定义特性，我们可以很方便地使用递归来进行求值. 首先我们给出算术表达式的归纳定义:

```
<expr> ::= <number>    # 一个数是表达式
  | "(" <expr> ")"     # 在表达式两边加个括号也是表达式
  | <expr> "+" <expr>  # 两个表达式相加也是表达式
  | <expr> "-" <expr>  # 接下来你全懂了
  | <expr> "*" <expr>
  | <expr> "/" <expr>
```

根据上述 BNF 定义，一种解决方案已经逐渐成型了：既然长表达式是由短表达式构成的，我们就先对短表达式求值，然后再对长表达式求值.
为了在 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. 这样我们就可以很容易把求值函数的框架写出来了:

```
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 主运算符 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);
    }
  }
}
```

其中 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的，也就不需要继续进行求值了. 我们举一些例子来说明

check_parentheses()函数的功能:

```
"(2 - 1)"             // true
"(4 + 3 * (2 - 1))"   // true
"4 + 3 * (2 - 1)"     // false, the whole expression is not surrounded by a matched
                      // pair of parentheses
"(4 + 3)) * ((2 - 1)" // false, bad expression
"(4 + 3) * (2 - 1)"   // false, the leftmost '(' and the rightmost ')' are not matched
```

那么，我们接下来具体要实现的便是括号匹配函数，token 数值的计算，主运算符位置寻找函数。

括号匹配

bool check_parentheses(int p,int q)
功能：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。若符号要求函数返回 true，否则函数返回 false
首先，当我们出现 p>=q 的情况，此情况为错误情况，报错之。

```
bool check_parentheses(int p,int q)
{
  if(p>=q)
  {
  Log("括号匹配出现了错误,p>=q的情况\n");
  return false;
  }
  if (tokens[p].type!='('||tokens[q].type!=')')
  {
    return false;
  }
```

接下来我们用一个整形 un_left 记录未匹配的左括号数目
那么当我们没有完全遍历完整个表达式（i<q）但 un_left=0 的情况，那么说明他一定是没有被一对括号正确包裹的表达式（因为被一对括号包裹的表达式一定是开头和结尾的括号匹配，但未遍历完就匹配上了说明结尾没有括号/结尾的括号与非开头括号匹配/错误括号格式）
接下来当遍历完后，如果全部匹配成功(un_left==0)，即返回 true 否则依然为 false

```
int un_left=0;
for(int i=p;i<=q;i++)
{
  if(tokens[i].type=='(')
  {
    un_left++;
  }
  if(tokens[i].type==')')
  {
    un_left--;
  }
  if(un_left==0&&i<q)
  {
    return false;
  }

}
if(un_left!=0)
{
  return false;
}
return true;
```

主运算符位置寻找

当我们消除了被括号包裹的表达式的括号后，一个式子一定能表达为 expr1 mainop expr2
那么我们如何找到 mainop 呢，我们知道 mainop 是表达式中最后计算的运算符，那么
首先 mainop 一定是不在一对括号内的（包裹整个表达式的括号已经被我们消去），因为在括号内一定会在括号内式子计算完之前/之时计算，但因为包裹整个表达式的括号已经被我们消去，所以在括号外一定有比他计算的更晚的运算符
都在括号外面的运算符，优先级最低的最后计算，很好理解， 先计算*/，在计算+-
同级别同在括号外的运算符，越靠右的优先级越低
那么我们据此规则便可以写出以下函数
int find_dominant_operator(int p,int q)
功能：在一个 token 表达式中寻找 DominantOp，返回其索引位置

```
int find_dominant_operator(int p,int q)
{
  int i=0,left=0,pr=100,position=p;
  for(i=p;i<=q;i++)
  {
    if(tokens[i].type=='(')
    {
      left++;
      i++;
      while(1)
      {
        if(tokens[i].type=='(')
        {
          left++;
        }
        if(tokens[i].type==')')
        {
          left--;
        }
        i++;
        if(left==0)
        {
          break;
        }
        if(i>q)
        {
          Log("括号匹配错误，错误的括号数量\n");
          assert(0);
        }
      }


    }

    if(tokens[i].type==TK_HEX||tokens[i].type==TK_DEC||tokens[i].type==TK_REG)
    {
      continue;
    }
    int temp_pr=get_pr(i);
    if(temp_pr<=pr)
    {
      position=i;
      pr=temp_pr;
    }
  }
  return position;

}
```

其中 get_pr（N）为获取运算符优先级的函数，如下

```c
int get_pr(int i)
{
  if(tokens[i].type=='*'||tokens[i].type=='/')
  return 3;
  if(tokens[i].type=='+'||tokens[i].type=='-')
  return 2;
  if(tokens[i].type==TK_EQ||tokens[i].type==TK_NEQ)
  return 1;
  if(tokens[i].type==TK_AND||tokens[i].type==TK_OR)
  return 0;
  return 999;
}
```

token 数值计算

完成了上面的函数后，我们只需要再完善一下对 16,10 进制，reg 的计算即可，计算很简单，16，10 进制算一下即可，reg 直接读出来数据，代码如下：

```c
else if(p==q)
{
  switch(tokens[p].type)
  {
    case TK_DEC:
    for (int i=0;i<len;i++) val=val*10+tokens[p].str[i]-'0';
    Log("%d到%d为10进制，计算结果为%u\n",p,q,val);
    return val;
    case TK_HEX:
    for (int i=0;i<len;++i) val=val*16+hex_cal(tokens[p].str[i]);
    Log("%d到%d为16进制，计算结果为%u\n",p,q,val);
    return val;
    case TK_REG:
    for(int i=0;i<8;i++)
    {
      if(strcmp(tokens[p].str,regsl_c[i])==0)

      {val= reg_l(i);
      Log("%d到%d为reg_l计算结果为%u\n",p,q,val);
      return val;
      }
      if(strcmp(tokens[p].str,regsw_c[i])==0)
      {
        val= reg_w(i);
      Log("%d到%d为reg_w计算结果为%u\n",p,q,val);
      return val;
      }
      if(strcmp(tokens[p].str,regsb_c[i])==0)
      {val= reg_b(i);
      Log("%d到%d为reg_b计算结果为%u\n",p,q,val);
      return val;
```

16 进制计算函数如下

```c
uint32_t hex_cal(char ch)
{
  return (('a'<=ch&&ch<='f')?ch-'a'+10:(('A'<=ch&&ch<='F')?ch-'A'+10:ch-'0'));
}
```

最终整体代码

```c
uint32_t eval(int p,int q)
{uint32_t val=0,len=strlen(tokens[p].str);
  if(p>q)
  {
    Log("表达式计算出现错了，p>q\n");
    assert(0);
  }
  else if(p==q)
  {
    switch(tokens[p].type)
    {
      case TK_DEC:
      for (int i=0;i<len;i++) val=val*10+tokens[p].str[i]-'0';
      Log("%d到%d为10进制，计算结果为%u\n",p,q,val);
      return val;
      case TK_HEX:
      for (int i=0;i<len;++i) val=val*16+hex_cal(tokens[p].str[i]);
      Log("%d到%d为16进制，计算结果为%u\n",p,q,val);
      return val;
      case TK_REG:
      for(int i=0;i<8;i++)
      {
        if(strcmp(tokens[p].str,regsl_c[i])==0)

        {val= reg_l(i);
        Log("%d到%d为reg_l计算结果为%u\n",p,q,val);
        return val;
        }
        if(strcmp(tokens[p].str,regsw_c[i])==0)
        {
          val= reg_w(i);
        Log("%d到%d为reg_w计算结果为%u\n",p,q,val);
        return val;
        }
        if(strcmp(tokens[p].str,regsb_c[i])==0)
        {val= reg_b(i);
        Log("%d到%d为reg_b计算结果为%u\n",p,q,val);
        return val;
        }

      }
      if(strcmp(tokens[p].str,"pc")==0)
      {
        Log("%d到%d为pc计算结果为%u\n",p,q,cpu.pc);
        return cpu.pc;
      }
      else
      {
        printf("错误的寄存器名字%s\n",tokens[p].str);
        assert(0);
      }
      default:
    }

  }
```

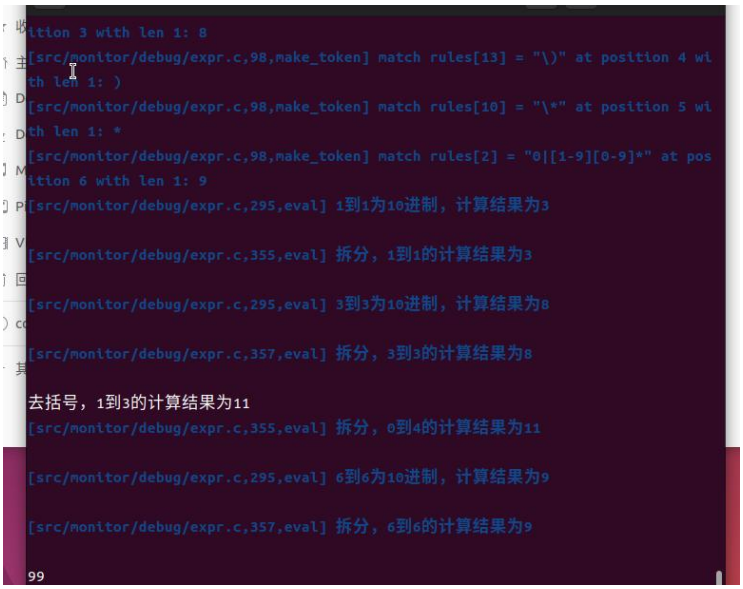
```c
  else if(check_parentheses(p,q)==true)
  {uint32_t result=0;
     result=eval(p+1,q-1);
     printf("去括号，%d到%d的计算结果为%d\n",p+1,q-1,result);
     return result;
  }
  else
  {
   int op=find_dominant_operator(p,q);
   if(tokens[op].type==TK_DEREF)
   {
     uint32_t addr=eval(p+1,q);
     val=vaddr_read(addr,4);
     Log("%d到%d的计算结果为解引用结果，结果为%u\n",p,q,val);
     return val;
   }

   uint32_t val1=eval(p,op-1);
   Log("拆分，%d到%d的计算结果为%u\n",p,op-1,val1);
   uint32_t val2=eval(op+1,q);
   Log("拆分，%d到%d的计算结果为%u\n",op+1,q,val2);
   switch(tokens[op].type)
   {
     case '+':
     return val1+val2;
     case '-':
     return val1-val2;
     case '*':
     return val1*val2;
     case '/':
     return val1/val2;
     case TK_AND:
     return val1&&val2;
     case TK_OR:
     return val1||val2;
     case TK_EQ:
     return val1==val2;
     case TK_NEQ:
     return val1!=val2;
     default:
     printf("错误的运算符类型,%d\n",tokens[op].type);
     assert(0);
   }

  }
}
```

```c
}
uint32_t expr(char *e, bool *success) {
  if (!make_token(e)) {
    *success = false;
    return 0;
  }
  if(tokens[0].type=='*')
  {
    tokens[0].type=TK_DEREF;
  }
  for(int i=1;i<nr_token;i++)
{
  if(
  tokens[i].type=='*'&&
  tokens[i-1].type!=TK_HEX&&
  tokens[i-1].type!=TK_DEC&&
  tokens[i-1].type!=TK_REG&&
  tokens[i-1].type!=')'
  )
  {
    tokens[i].type==TK_DEREF;
  }
}
 //printf("nr_token为%d\n",nr_token);
  return eval(0,nr_token-1);

  /* TODO: Insert codes to evaluate the expression. */

  return 0;
}
```

运行结果:

```
(nemu) p 3+2
[src/monitor/debug/expr.c,98,make_token] match rules[2] = "0|[1-9][0-9]*" at pos
ition 0 with len 1: 3
[src/monitor/debug/expr.c,98,make_token] match rules[8] = "\+" at position 1 wit
h len 1: +
[src/monitor/debug/expr.c,98,make_token] match rules[2] = "0|[1-9][0-9]*" at pos
ition 2 with len 1: 2
[src/monitor/debug/expr.c,295,eval] 0到0为10进制，计算结果为3

[src/monitor/debug/expr.c,355,eval] 拆分，0到0的计算结果为3

[src/monitor/debug/expr.c,295,eval] 2到2为10进制，计算结果为2

[src/monitor/debug/expr.c,357,eval] 拆分，2到2的计算结果为2

5
```

```
(nemu) p (3+8)*9
```



内存扫描

我们在上面已经完成了表达式求值的功能，只需要在内存扫描的时候分割参数，然后求解表达式最后调用 vaddr_read 读取之即可

代码实现

```c
static int cmd_x(char *args) {
  if(args==NULL)
  {
    printf("扫描内存缺少参数\n");
    return 0;
  }
  int arglen=strlen(args);
  char *argN=strtok(args," ");
  if(argN==NULL)
    {
    printf("扫描内存缺少参数\n");
    return 0;
  }
  int N=atoi(argN);
  if(N<=0)
  {
    printf("错误的参数N\n");
    return 0;
  }
  //printf("N的值为%d\n",N);
  char *exprs=args+strlen(argN)+1;
  if(exprs>=args+arglen)
  {
    printf("没有输入表达式\n");
    return 0;
  }
  //printf("expr为%s\n",expr);
  bool success=true;
  vaddr_t expr_result=expr(exprs,&success);
  //vaddr_t expr_result=atoi(expr);
  if(success==false)
  {
    printf("表达式求解过程中出现了错误\n");
```

```c
  printf("Memory\n");
  for(int i=0;i<N;i++)
  {
    uint32_t data = vaddr_read(expr_result + i * 4,4);
        printf("0x%08x  " , expr_result + i * 4 );
        for(int j =0 ; j < 4 ; j++){
            printf("0x%02x " , data & 0xff);
            data = data >> 8 ;
        }
        printf("\n");


  }
  return 0;
```

运行结果

监视点

完成监视点功能我们需要实现以下几个函数:
WP* new_wp(char *args);//创建一个监视点，其表达式为 args
void free_wp(int N);// 将一个监视点释放
bool check_wp();//判断监视点值是否改变
void print_wp();//打印监视点状态

监视点结构:

```
6  typedef struct watchpoint {
7    int NO;
8    struct watchpoint *next;
9
10   /* TODO: Add more members if necessary */
11   char exp[128];
12   uint32_t old_val;
13   int hit_num;
14
15
16 } WP;
```

除了链表结构和编号，我们还需要以下的数据:
exp，记录其将要监视的表达式
old_val，记录上一时刻监视点表达式的计算数值
hit_num 记录监视点数值改变的次数

创建监视点:

我们用一个全局静态变量来记录下一个监视点的该赋予的编号

```
static int W_id;
```

接下来就是从 freelist 里获取闲置监视点，将其加入监视点队列，并设置其表达式，调用表达式求值计算其值即可，只是一些简单的链表的插入删除和初始值的赋予。

```
WP* new_wp(char *args)//get a free watchpoint from the list 'free_'
{
  if (free_==NULL)
  {
    Log("没有足够的监视点\n");
    assert(0);
    return NULL;
  }
  bool success=true;
  uint32_t val=expr(args,&success);
  if(success==false)
  {
    Log("监视点设置时表达式计算错误\n");
    val=0;
    return NULL;
  }
  if (head==NULL)
  {
    head=free_;
    free_=free_->next;
    head->next=NULL;
    head->NO=++W_id;
    head->hit_num=0;
    strcpy(head->exp,args);
    head->old_val=val;
    return head;
  }
  WP *now=head;
  while (now->next!=NULL) now=now->next;
```

```
  now->next=free_;


  free_=free_->next;
  now->next->next=NULL;
  now->next->NO=++W_id;
  now->next->hit_num=0;
  strcpy(now->next->exp,args);
  now->next->old_val=val;
  return now->next;
}
```

最后我们会在 cmd_w 中调用此函数创建监视点

运行结果：

```
(nemu) w 1+2
[src/monitor/debug/expr.c,98,make_token] match rules[2] = "0|[1-9][0-9]*" at pos
ition 0 with len 1: 1
[src/monitor/debug/expr.c,98,make_token] match rules[8] = "\+" at position 1 wit
h len 1: +
[src/monitor/debug/expr.c,98,make_token] match rules[2] = "0|[1-9][0-9]*" at pos
ition 2 with len 1: 2
[src/monitor/debug/expr.c,295,eval] 0到0为10进制，计算结果为1

[src/monitor/debug/expr.c,355,eval] 拆分，0到0的计算结果为1

[src/monitor/debug/expr.c,295,eval] 2到2为10进制，计算结果为2

[src/monitor/debug/expr.c,357,eval] 拆分，2到2的计算结果为2

插入检查点成功，表达式为1+2,目前值为3，编号为1
(nemu) w2+3
```

```
插入检查点成功，表达式为2+3,目前值为5，编号为2
(nemu) info w
监视点1，表达式为1+2，  值为3，hitnum为0
监视点2，表达式为2+3，  值为5，hitnum为0
```

监视点删除

我们给定一个编号，遍历活跃监视点，寻找并将对应监视点放回到 freelist 中（没有对应编号则提示，不做任何操作），代码实现如下；

```
void free_wp(int N)//make wp free and return to the list 'free_'
{
  WP *now=head,*wp;
  if (now!=NULL&&now->NO==N)
  {
    head=head->next;
    now->next=free_;
    free_=now;
    return;
  }
  while (now!=NULL&&now->next->NO!=N) now=now->next;
  if (now==NULL)return;
  if(now->NO!=N)
  {
    printf("没有编号为%d的监视点\n",N);
    return;
  }
  wp=now->next;
  now->next=wp->next;
  wp->next=free_;
  free_=wp;
}
```

然后我们在 ui.c 中用 cmd_d 调用即可

运行结果：

```
d - 删除监视点  格式 d N 举例：d 2 删除序号为N的监视点备注
(nemu) d 1
(nemu) info w
监视点2，表达式为2+3，  值为5，hitnum为0
```

判断监视点是否改变

每当执行一步的时候，我们都要重新计算监视点的值，并且比对存储的老值，并将新的计算值存储如果有监视点发生了改变，则返回 1，无改变则返回 0，代码实现如下：

```
bool check_wp()
{
  bool _suc,change=0;
  uint32_t now_val;
  for (WP *now=head;now!=NULL;now=now->next)
  {
      now_val=expr(now->exp,&_suc);
      if (now->old_val!=now_val)
      {
       now->hit_num++;
       change=1;
       Log("监视点%d，表达式%s的值由%u变为了%u\n",now->NO,now->exp,now->old_val,now_val);
       now->old_val=now_val;
      }
      else
      {
  //printf("监视点%d，表达式%s的值为%u，未改变\n",now->NO,now->exp,now->old_val);
      }

  }
  return change;
}
void print_wp()
```

然后我们应该在每执行一步指令的时候调用一次更新操作，并且当有监视点的值改变的时候应该将系统暂停，命令的执行在 src/monitor/cpu_exec.c 的 cpu_exec 函数执行，我们根据代码注释找到修改的位置，修改，调用 check_wp 函数并根据返回值决定是否要将 state 改为 STOP

```
    log_clearbuf();

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

#endif
```

运行结果:

```
(nemu) w $pc
[src/monitor/debug/expr.c,98,make_token] match rules[3] = "\$(eax|ebx|ecx|edx|es
p|ebp|esi|edi|pc|ax|bx|cx|dx|sp|bp|si|di|al|cl|dl|bl|ah|ch|dh|bh)" at position 0
 with len 3: $pc
[src/monitor/debug/expr.c,325,eval] 0到0为pc计算结果为1048576

插入检查点成功，表达式为$pc,目前值为1048576，编号为3
(nemu)
```

```
(nemu) si
  100000:   b8 34 12 00 00                      movl $0x1234,%eax
[src/monitor/debug/expr.c,98,make_token] match rules[3] = "\$(eax|ebx|ecx|edx|es
p|ebp|esi|edi|pc|ax|bx|cx|dx|sp|bp|si|di|al|cl|dl|bl|ah|ch|dh|bh)" at position 0
 with len 3: $pc
[src/monitor/debug/expr.c,325,eval] 0到0为pc计算结果为1048581

[src/monitor/debug/watchpoint.c,97,check_wp] 监视点3，表达式$pc的值由1048576变为
了1048581
```

```
(nemu) si 3
  100005:   b9 27 00 10 00                      movl $0x100027,%ecx
[src/monitor/debug/expr.c,98,make_token] match rules[3] = "\$(eax|ebx|ecx|edx|es
p|ebp|esi|edi|pc|ax|bx|cx|dx|sp|bp|si|di|al|cl|dl|bl|ah|ch|dh|bh)" at position 0
 with len 3: $pc
[src/monitor/debug/expr.c,325,eval] 0到0为pc计算结果为1048586

[src/monitor/debug/watchpoint.c,97,check_wp] 监视点3，表达式$pc的值由1048581变为
了1048586
```

$pc 每一步都会变，可以看到，正确的检测到了改变，并且将程序暂停了.

打印监视点

我们只需要遍历活跃监视点链表将其信息都输出即可，代码如下:

```
}
void print_wp()
{
  for (WP*now=head;now!=NULL;now=now->next)
  {
    printf("监视点%d，表达式为%s，  值为%u，hitnum为%d\n",now->NO,now->exp,now->old_val,now->hit_num);
  }
};
```

然后我们在 cmd_info 中调用 print_wp 即可

运行结果

```
(nemu) info w
监视点3，表达式为$pc，  值为1048586，hitnum为2
```

断点

断点我们可以通过执行 w $pc==addr 即可

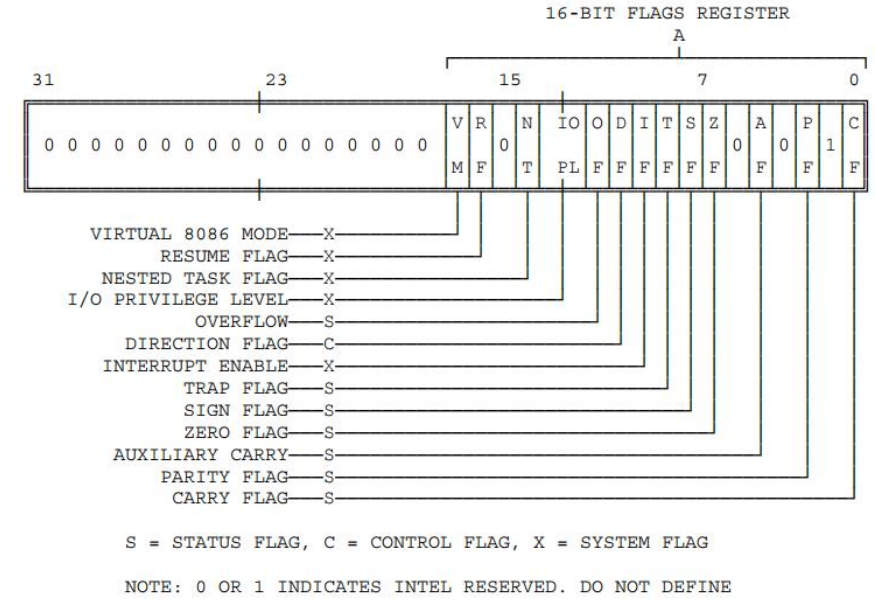
必答题

> **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？

CF

查阅 i386 手册，搜索 EFLAGS，我们找到下图：

Figure 2-8. EFLAGS Register



S = STATUS FLAG, C = CONTROL FLAG, X = SYSTEM FLAG

NOTE: 0 OR 1 INDICATES INTEL RESERVED. DO NOT DEFINE

然后我们找到 CF 位的介绍：

2.3.4.1 Status Flags

The status flags of the EFLAGS register allow the results of one instruction to influence later instructions. The arithmetic instructions use OF, SF, ZF, AF, PF, and CF. The SCAS (Scan String), CMPS (Compare String), and LOOP instructions use ZF to signal that their operations are complete. There are instructions to set, clear, and complement CF before execution of an arithmetic instruction. Refer to Appendix C for definition of each status flag.

我们再查找 **Appendix C** 找到 CF 寄存器之含义：

```
0    CF      Carry Flag — Set on high-order bit carry or borrow; cleared
             otherwise.
```

故可知 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。

阅读的范围

2.3.4 Flags Register
Appendix C Status Flag Summary

ModR/M

```
    The ModR/M and SIB bytes follow the opcode byte(s) in many of the 80386
instructions. They contain the following information:
    The indexing type or register number to be used in the instruction
    The register to be used, or more information to select the instruction
    The base, index, and scale information
    The ModR/M byte contains three fields of information:
    The mod field, which occupies the two most significant bits of the byte,
combines with the r/m field to form 32 possible values: eight registers and 24
indexing modes
    The reg field, which occupies the next three bits following the mod field,
specifies either a register number or three more bits of opcode information. The
meaning of the reg field is determined by the first (opcode) byte of the
instruction.
    The r/m field, which occupies the three least significant bits of the byte,
can specify a register as the location of an operand, or can form part of the
addressing-mode encoding in combination with the field as described above
```

当 Mod = 00 时，ModR/M 字节通过寄存器直接进行内存寻址
当 Mod = 01 时，ModR/M 字节通过寄存器+I8 进行内存寻址(I 为立即数，即 8 位立即数)
当 Mod = 10 时，ModR/M 字节通过寄存器+I32 进行内存寻址
当 Mod = 11 时，ModR/M 字节直接操作两个寄存器

阅读范围

17.2 INSTRUCTION FORMAT
17.2.1 ModR/M and SIB Bytes

mov 指令的具体格式

MOV {条件}{S} 目的寄存器，源操作数

阅读范围

17.2 INSTRUCTION FORMAT
17.2.2.11 Instruction Set Detail
MOV —— Move Data.
MOV —— Move to/from Special Registers

shell 命令

代码行数统计

使用 wc 进行行数统计：find . -name "*.h" -or -name "*.c" | xargs grep -ve "^$" | wc -l

```
ubuntu@VM-8-6-ubuntu:~/pa/ics2019/ics2019/nemu$ find . -name "*.h" -or -name "*.c" | xargs grep -ve "^$" | wc -l
4556
```

一共有 4556 行代码
我们 gitcheck out pa0，

```
ubuntu@VM-8-6-ubuntu:~/pa/ics2019/ics2019_pa1/nemu$ find . -name "*.h" -or -name "*.c" | xargs grep -ve "^$" | wc -l
4008
```

我们一共编写了 4556-4008=538 行（忽略空行）

make count

```
COUNT_L := $(shell  find . -name "*.h" -or -name "*.c" | xargs grep -ve "^$$" | wc -l)

count:
    @echo $(COUNT_L) lines in nemu
```

```
ubuntu@VM-8-6-ubuntu:~/pa/ics2019/ics2019/nemu$ make count
Building x86-nemu
4556 lines in nemu
```

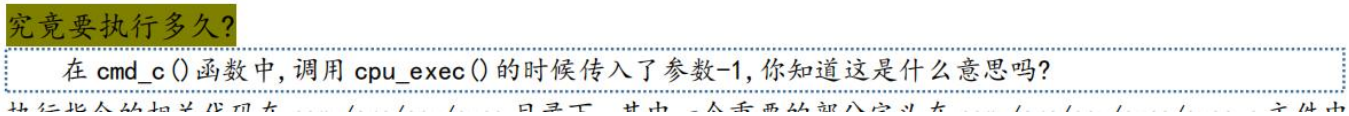
GCC 编译选项

-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。

两者并用的话相当于在出现任何警告的时候都报错，终止编译，这可以提高代码的安全性，并且让我们在早期就发现可能在后期触发 bug 的隐患，让我们提早 debug，方便代码维护。

思考题

究竟要运行多久



我们打开 nemu/src/monitor/cpu-exec.c 文件，观察 cpu_exec()函数，这个函数模仿 CPU 的工作方式，不断执行 n 条指令直到指令执行完毕或进入 nemu_trap 才退出指令执行的循环。
然后我们又发现其传入参数为一个无符号 64 位整型

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

那么其实传入-1，相当于 uint64_t 中的 2^64-1（最大值），意味着执行所有的指令直到 nemu_trap

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.

atexit 函数指示函数的结束，atexit 是一个特殊的函数，它是正常程序退出时调用的注册函数，用以下代码验证

问题 输出 调试控制台 终端 端口

```c
#include<stdio.h>
#include<stdlib.h>
void after_main()
{
        printf("在main函数之后调用了函数\n");
}
int main()
{       atexit(&after_main);
        printf("main函数结束\n");
        return 0;

}
```

```
bash: ./test.c: 权限不够
ubuntu@VM-8-6-ubuntu:~/pa$ sudo ./test
main函数结束
在main函数之后调用了函数
ubuntu@VM-8-6-ubuntu:~/pa$
```

static 的意义

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？

static 定义全局静态变量，该变量只能在本文件中进行访问，保护了数据的安全性。其他源文件想对该变量进行操作时，需要通过封装的函数来实现。

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?

是必须的，改了不可以正常运行，因为调试器设置一个断点时，它会将原始代码的第一个字节替换为 0xCC，并在恢复执行时还原。如果 int3 指令长度为两个字节，那么调试器就需要替换和还原两个字节，那么当我们打断点的指令长度为 1 的时候，他就会导致下一条指令也被修改，这会导致错误的产生。

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.

如果把断点设置在指令的非首字节，那么可能会导致一些错误。因为软件断点是通过将指令的第一个字节替换为中断指令（如 int3）来实现的，如果替换了非首字节，那么原始指令就会被破坏或改变，而同时这种改变因为并不是放在首字节，所以并不会被视为操作码，而会被视为操作数等，导致也不会被恢复，这可能会使程序执行错误的操作或者无法继续执行

NEMU 的前世今生

NEMU 的前世今生

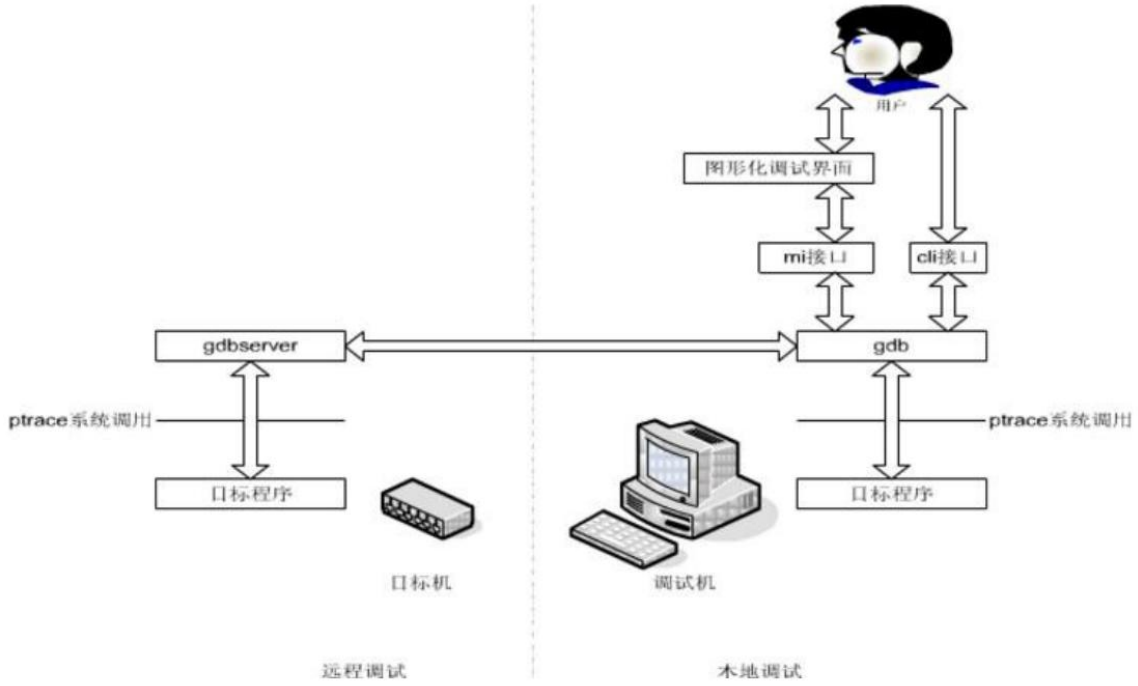
你已经对 NEMU 的工作方式有所了解了.事实上在 NEMU 诞生之前,NEMU 曾经有一段时间并不叫 NEMU,而是叫 NDB(NJU Debugger),后来由于某种原因才改名为 NEMU.如果你想知道这一段史前的秘密,你首先需要了解这样一个问题:模拟器(Emulator)和调试器(Debugger)有什么不同?更具体地,和 NEMU 相比,GDB 到底是如何调试程序的?

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（“目标”程序）的计算机程序。

从调试程序的方式上进行比较:

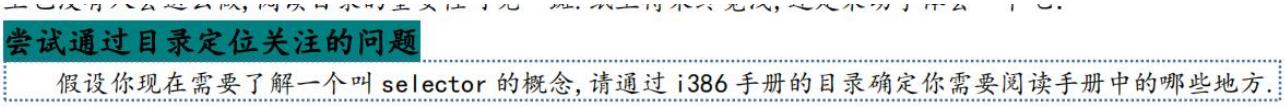
PA1 中，我们为 NEMU 实现了简易调试器，它是通过 ui_mainloop 获取相关 命令后，执行对应程序并输出相关信息的。

而在 GDB 中，调试是通过 ptrace 系统调用进行实现的。其整体框架如下



GDB 调试建立在信号的基础上的，在使用参数 ptrace 系统调用建立调试关 系后，交付给目标程序的任何信号首先都会被 gdb 截获。 因此 gdb 可以先行对 信号进行相应处理，并根据信号的属性决定是否要将信号交付给目标程序。

尝试通过目录定位关注的问题



直接在手册里搜索 selector,得到需要阅读的以下章节:
CHAPTER 5 MEMORY MANAGEMENT
5.1 SEGMENT TRANSLATION
    5.1.3 Selector
CHAPTER 17 80386 INSTRUCTION
17.2 INSTRUCTION FORMAT
    17.2.2 How to Read the Instruction Set Page 17.2.2.11 Instruction Set Detail ARPL —— Adjust RPL Field of Selector
Figures
    5-6 Format of a Selector