PA1实验

1711333 蒋璋

2.1.2 实现正确的结构体

cpu_exec(-1)

return结束了吗？

基础设施的实现

单步执行

打印寄存器

表达式求值

扫描内存

监视点

必答题：

> *NEMU开始执行的时候,首先会调用init_monitor()函数(在nemu/src/monitor/monitor.c中定义)*
>
> *进行一些和monitor相关的初始化工作,我们对其中几项初始化工作进行一些说明.reg_test()函数(在nemu/src/cpu/reg.c中定义)会生成一些随机的数据,对寄存器实现的正确性进行测试.若不正确,将会触发assertionfail*

> > *阅读`reg_test()`的代码, 思考代码中的`assert()`条件是根据什么写出来的.*

测试方法:

给pc赋值

```
  uint32_t pc_sample = rand();
  cpu.pc = pc_sample;
```

给每一个寄存器赋值

```
int i;
  for (i = R_EAX; i <= R_EDI; i ++) {
    sample[i] = rand();
    reg_l(i) = sample[i];
    assert(reg_w(i) == (sample[i] & 0xffff));
  }
```

reg_l 为寄存器的32位值。reg_w为寄存器的低16位的值。通过给寄存器随机赋值并比较寄存器的低16位是否与随机数的低16位相同来判断寄存器是否正常工作。

2.1.2 实现正确的结构体

> *实现正确的寄存器结构体我们在PA0中提到,运行NEMU会出现assertionfail的错误信息,这是因为框架代码并没有正确地实现用于模拟寄存器的结构体CPU_state,现在你需要实现它了(结构体的定义在nemu/include/cpu/reg.h中)。关于i386寄存器的更多细节,请查阅i386手册。Hint:使用匿名union.*

由于我使用 的是南京大学的icss2019，文件在 nemu/src/isa/x86/reg.h中

```
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;
```

这个其实没什么好说的，就是八个寄存器加一个pc，每个寄存器是32位的。由于我们需要用不同的方式访问寄存器，所以使用union。union能够使得不同的变量名能占据相同的存储空间，方便了不同寄存器名的访问。另外使用匿名联合和结构可以在使用时少输入一个结构名或联合名。如我们想要访问寄存器eax可以 通过cpu.eax直接访问，而不用打上一堆的中间结构的名字。

cpu_exec(-1)

> *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?*

首先还是RTFC, 看看代码。

```
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;
  }

  for (; n > 0; n--)
  {
    __attribute__((unused)) vaddr_t ori_pc = cpu.pc;

    /* Execute one instruction, including instruction fetch,
     * instruction decode, and the actual execution. */
    __attribute__((unused)) vaddr_t seq_pc = exec_once();
```

可以看到cpu_exec的函数定义的参数是一个64位的无符号整数。函数内部的主循环是执行n条指令，n就是传入了参数。那为什么要传入-1呢？我们知道负数在计算机中是以补码的形式存储的，那么-1在内存中存储的值为0xffffffffffffffff 16个f，也就是64位无符号数能够表示的最大值。

return结束了吗？

> *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

上课时提到了atexit， 程序在执行完 return 以后会调用atexit函数这种终止注册函数。

基础设施的实现

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

监视点的相关代码在 src/monitor/debug/ 下

cmd_help 和 cmd_c已经实现好了，cmd_help遍历cmd_table，把相关的信息都打印出来。cmd_c 调用 cpu_exec(-1)， 也就是一直执行的意思。

从ui_mainloop()函数中可以知道命令的解析需要遍历 cmd_table，因此我们先补充cmd_table

```
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", "excute n step or null input excute 1 step", cmd_si},
    {"info", "print the state of register or the watch point", cmd_info},
    {"p", "calculate the value of the expr", cmd_p},
    {"x", "scan the memory and print the value at the physical addr", cmd_x},
```

```
    {"w", "add a watch point at the expr", cmd_w},
    {"d", "delete the Nth watch point ", cmd_d},
    /* TODO: Add more commands */
};
```

这是一个一次性的结构数组，在里面补充上我们的说明和函数定义即可。

单步执行

单步执行的功能十分简单, 而且框架代码中已经给出了模拟CPU执行方式的函数, 你只要使用相应的参数去调用它就可以了. 如果你仍然不知道要怎么做, RTFSC.

如果后面没跟数组那就是执行一步，否则就把数组用atoi解析输出来，然后执行相应的步数。

```
static int cmd_si(char *args)
{
  if (args == NULL)
  {
    cpu_exec(1);
  }
  else
  {
    int i = atoi(args);
    cpu_exec(i);
  }
  return 0;
}
```

打印寄存器

打印寄存器:cmd_info 函数

```
static int cmd_info(char *args)
{
  if (args[0] == 'w')
  {
    print_wp();
  }
  else if (args[0] == 'r')
  {
    for (int i = R_EAX; i < R_EDI; i++)
    {
      printf("%s 0x%08x\n", reg_name(i, 4),
cpu.gpr[i]._32);
    }
    printf("%s 0x%08x\n", "eip", cpu.pc);
    for (int i = R_AX; i < R_DI; i++)
```

```
    {
      printf("%s 0x%04x\n", reg_name(i, 2),
reg_w(i));
    }
    for (int i = R_AL; i < R_BH; i++)
    {
      printf("%s 0x%02x\n", reg_name(i, 1),
reg_b(i));
    }
  }
  return 0;
}
```

好像也是遍历一遍就行了。print_wp()的定义在后面介绍。

表达式求值

```
static int cmd_p(char *args)
{
  bool success;
  uint32_t result = expr(args, &success);
  if (success)
  {
    printf("the result is %u\n", result);
    return 0;
  }
  return -1;
}
```

这个接口倒是没什么计数含量，主要是表达式的实现，又复习了一遍编译原理。

不过仅仅是表达式的计算用不上语法分析，这里主要处理好词法分析的部分就好了。首先是单词的识别，这里给我们写了一个make_token的接口，需要我们去实现他。

正则表达式的函数已经有了，只要写好正则表达式就行。

```
static struct rule
{
  char *regex;
  int token_type;
} rules[] = {

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

```
  */

    {" +", TK_NOTYPE},                    // spaces
    {"[1-9][0-0]*", TK_DEC},              //dec
    {"0x[0-9a-f]+", TK_HEX},              //hex
    {"\\+", '+'},                         // plus
    {"\\-", '-'},                         // sub
    {"\\*", '*'},                         //mul
    {"\\/", '/'},                         // div
    {"==", TK_EQ},                        // equal
    {"\\$[eE][a-zA-Z][a-zA-Z]", TK_REG}, //REG
    {"\\(", '('},                         // LEF
    {"\\)", ')'},                         //RIG
    {"&&", TK_AND},                       //AND
    {"!=", TK_NE}                         // NOT EQUEL
};
```

正则式就是那些运算符和数字，包括10进制和16进制，另外还有寄存器名称的识别。

然后只要判断单词的类型给tokens正确的值即可。

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

  nr_token = 0;
  tokens[nr_token].type = '#';
  strncpy(tokens[nr_token].str, "#", 1);
  nr_token++;
  while (e[position] != '\0')
  {
    /* Try all rules one by one. */
    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;
        position += substr_len;
        /* TODO: Now a new token is recognized with rules[i]. Add codes
         * to record the token in the array `tokens'. For certain types
         * of tokens, some extra actions should be performed.
         */
```

```
        switch (rules[i].token_type)
        {
        case TK_NOTYPE:
          break;
        default:
          tokens[nr_token].type = rules[i].token_type;
          strncpy(tokens[nr_token].str, substr_start, substr_len);
          nr_token++;
          //TODO();
        }
        break;
      }
    }

    if (i == NR_REGEX)
    {
      printf("no match at position %d\n%s\n%*.s^\n", position, e, position, "");
      return false;
    }
  }
  tokens[nr_token].type = '#';
  strncpy(tokens[nr_token].str, "#", 1);
  nr_token++;
  return true;
}
```

然后是计算的实现。算术表达式最适合的文法是算符优先文法，实现的原理大概是：维护两个栈，数字栈和符号栈。按照输入流输入token，只要是数字一律移进。如果token[i]是符号，那么就得看优先级了，如果当前符号的优先级比符号栈中的上一个符号优先级高，那么该符号就移进，如果当前符号的优先级小于等于前面的优先级，那么就该归约了。归约的过程就是从数字栈中弹出操作数，从符号栈中弹出符号，计算，把计算结果压入数字栈。为了计算的方便，我们在第一个符号前面和最后一个符号后面各加入一个#，#的优先级最低。对于取地址运算符的处理，在词法分析的时候可以知道乘号的前一个符号，如果是运算符，那么就是取地址，如果是数字，那么就是乘号，然后给了取地址另外一个符号 %来识别。

下面是优先级函数：

分为前后两个函数，使用时比较栈顶符号的前置优先级和当前符号的后置优先级，根据结构做不同的操作。

```
int get_post_priority(int op)
{
  switch (op)
  {
  case '#':
```

```
    return -1;
  case '&':
    return 1;
  case '!':
  case '=':
    return 2;
  case '+':
  case '-':
    return 3;
  case '*':
  case '/':
    return 4;
  case '(':
    return 6;
  case '%': // 其实是取地址
    return 5;
  case ')':
    return 0;
  }
  return 0;
}

int get_pre_priority(int op)
{
  switch (op)
  {
  case '&':
    return 1;
  case '!':
  case '=':
    return 2;
  case '+':
  case '-':
    return 3;
  case '*':
  case '/':
    return 4;
  case '%': //其实是取地址
    return 5;
  case '(':
    return -1;
  case ')':
    return 6;
  case '#':
    return -2;
  }
  return 0;
}
```

```c
uint32_t expr(char *e, bool *success)
{
  if (!make_token(e))
  {
    *success = false;
    return 0;
  }
  values_top = -1;
  ops_top = -1;

  for (int i = 0; i < nr_token; i++)
  {
    // 括号应该加到符号栈里，这样就可以和前面的运算符隔开了
    // printf("%s\n", tokens[i].str);
    switch (tokens[i].type)
    {
    case TK_DEC:
    case TK_HEX:
    case TK_REG:
      values[++values_top] = eval(tokens[i]);
      break;
    default:
      if (ops_top == -1 || get_post_priority(eval(tokens[i])) > get_pre_priority(ops[ops_top]))
      {
        // printf("shift\n");
        if (i > 0 && strcmp(tokens[i].str, "*") == 0 && tokens[i - 1].type != TK_DEC && tokens[i - 1].type != TK_HEX && tokens[i - 1].type != TK_REG && tokens[i - 1].type != ')')
        {
          ops[++ops_top] = '%';
        }
        else
        {
          ops[++ops_top] = eval(tokens[i]);
        }
      }
      else
      {
        int tmp = 0;
        while (get_post_priority(eval(tokens[i])) <= get_pre_priority(ops[ops_top]) && ops_top > -1)
        {
          // printf("reduce\n");
          switch (ops[ops_top])
          {
          case '+':
            tmp = values[values_top] + values[values_top - 1];
            break;
```

```
          case '-':
            tmp = values[values_top] -
values[values_top - 1];
            break;
          case '%':
            tmp = vaddr_read(values[values_top], 4);
            values_top++;
            break;
          case '*':
            tmp = values[values_top] *
values[values_top - 1];
            break;
          case '/':
            tmp = values[values_top] /
values[values_top - 1];
            break;
          case ')':
            tmp = values[values_top];
            values_top++;
            ops_top--;
            break;
          case '=':
            tmp = values[values_top] ==
values[values_top - 1];
            break;
          case '!':
            tmp = values[values_top] !=
values[values_top - 1];
            break;
          case '&':
            tmp = values[values_top] &&
values[values_top - 1];
            break;
          }
          ops_top--;
          values[--values_top] = tmp;
          // printf("post : %d pre : %d\n",
get_post_priority(eval(tokens[i])),
get_pre_priority(ops[ops_top]));
        }
        ops[++ops_top] = eval(tokens[i]);
      }
    }
  }
  *success = true;
  /* TODO: Insert codes to evaluate the expression.
*/
  //TODO();
  return values[0];
}

```

扫描内存

表达式实现了以后就可以扫描内存。参数的识别用sscanf这个功能强大的函数来实现，识别出一个整数和后面的一个表达式，计算表达式的值得到地址，通过vaddr_read每次读取一个字节，以8位整形的16进制输出。

```
static int cmd_x(char *args)
{
  int n;
  char buf[128];
  sscanf(args, "%d %s", &n, buf);
  bool success;
  uint32_t addr = expr(buf, &success);
  if (!success)
    return -1;
  // 这里要把物理地址转虚拟地址
  for (int i = 0; i < n; i++)
  {
    printf("0x%08x : ", addr + i);
    printf("0x%02x\n", vaddr_read(addr + i, 1));
  }
  printf("\n");
  return 0;
}
```

```
(nemu) x 39 0x100000
0x00100000 : 0xb8
0x00100001 : 0x34
0x00100002 : 0x12
0x00100003 : 0x00
0x00100004 : 0x00
0x00100005 : 0xb9
0x00100006 : 0x27
0x00100007 : 0x00
0x00100008 : 0x10
0x00100009 : 0x00
0x0010000a : 0x89
0x0010000b : 0x01
0x0010000c : 0x66
0x0010000d : 0xc7
0x0010000e : 0x41
0x0010000f : 0x04
0x00100010 : 0x01
0x00100011 : 0x00
0x00100012 : 0xbb
0x00100013 : 0x02
0x00100014 : 0x00
0x00100015 : 0x00
0x00100016 : 0x00
```

```
0x00100017 : 0x66
0x00100018 : 0xc7
0x00100019 : 0x84
0x0010001a : 0x99
0x0010001b : 0x00
0x0010001c : 0xe0
0x0010001d : 0xff
0x0010001e : 0xff
0x0010001f : 0x01
0x00100020 : 0x00
0x00100021 : 0xb8
0x00100022 : 0x00
0x00100023 : 0x00
0x00100024 : 0x00
0x00100025 : 0x00
0x00100026 : 0xd6
```

结果需要与init.c中的img比较，结果一致即可。

监视点

监视点的功能是监视一个表达式的值何时发生变化.如果你从来没有使用过监视点,请在GDB中体验一下它的作用.简易调试器允许用户同时设置多个监视点,删除监视点,因此我们最好使用链表将监视点的信息组织起来.框架代码中已经定义了监视点的结构。

```
typedef struct watchpoint
{
  int NO;
  struct watchpoint *next;
  char args[64];
  /* TODO: Add more members if necessary */
  int value;
  bool isuse;
} WP;
```

监视点的结构是一个数组加链表的奇怪组合。我们应该需要保存它的表达式，以便计算出当前位置的值，另外还得要保存它上一次的值，以观察他是否变化。增加一位是否使用便于free 和 new 的时候的方便。

```
static int cmd_w(char *args)
{
  bool success;
  int value = expr(args, &success);
  if (!success)
  {
    printf("error expression!\n");
    return 0;
```

```
  }
  WP *wp = new_wp();
  strcpy(wp->args, args);
  wp->isuse = true;
  wp->value = value;
  return 0;
}
```

```
WP *new_wp()
{
  // 把free的头加到 head的前面
  if (free_ == NULL)
  {
    assert(0);
    return NULL;
  }
  WP *result = free_;
  free_ = free_->next;
  result->next = head;
  head = result;
  return result;
}
```

cmd_w用于从free_list中取头节点加入当前使用中的监视点链表

```
static int cmd_d(char *args)
{
  int n = atoi(args);
  del_wp(n);
  return 0;
}
```

```
void del_wp(int n)
{
  if (n >= 0 && n < NR_WP)
    free_wp(wp_pool + n);
}
```

```
void free_wp(WP *wp)
{
  if (!wp->isuse)
  {
    printf("error free number\n");
    return;
  }
  WP *temp = head;
  if (temp == wp)
  {
    head = head->next;
```

```
  }
  else
  {
    while (temp != NULL)
    {
      /* code */
      if (temp->next == wp)
      {
        WP *dd = temp->next->next;
        temp->next = dd;
        break;
      }
      temp = temp->next;
    }
  }
  wp->isuse = false;
  wp->next = free_;
  free_ = wp;
}
```

cmd_d就是相反的操作，把找到序号对应的node，把他取下来加入free即可。

然后就是监视点的检查了。

首先实现一个检查的函数，遍历一遍监视点的链表，看看每一个值是否与之前的相同，如果均不同返回false，否则返回true.

```
bool check_wp()
{
  bool flag = false;
  WP *wp = head;
  while (wp != NULL)
  {
    bool success;
    int result = expr(wp->args, &success);
    if (result != wp->value)
    {
      flag = true;
      printf("break point at %d : %s\n", wp->NO, wp->args);
    }
    wp->value = result;
    wp = wp->next;
  }
  return flag;
}
```

然后我们需要考虑检查的位置，自然需要在运行时检查，因此在cpu_exec中检查，在循环中加入

```
if (check_wp())
      nemu_state.state = NEMU_STOP;
```

就可以了。

> *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?*

主要是为了和全局变量区别，使用static后对其他文件不可见，只在当前文件内有效。

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?*

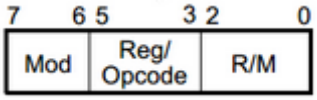
在手册的85-86页 4.1节

EFLAGS寄存器定义了系统的标志位。CF位是第0位，保存的是运算add、sub、mul、div、and、or等的进位信息。

> *ModR/M字节是什么?*

241页17.2

（2）、指令常规情况分析：
如果确定是不定长指令，则其后必定存在一个字节的ModR/M，而ModR/M的bit信息指出了通用形式的不定长指令的具体形式，ModR/M的格式如下所示：



其中第3、4、5位三位即Reg/Opcode来确定是哪一个通用寄存器G，（暂时仅考虑Reg/Opcode中reg的情况）；
其它两部分来确定E是什么（R/M）以及具体细节。
（Mod值有03四种情况、Reg/Opcode和R/M有07八种情况；Mod的00~10是内存，11是寄存器；R/M与Reg/Opcode的值即为寄存器的编号：eax/ax/al编号0、ecx/cx/cl编号为1...）

这里确实是没看的太懂。。。

> *mov指令的具体格式是怎么样的?*

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

可以用于寄存器立即数和内存之间的传输

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?

shell命令wc可以统计一个文件的行数或者是输入的行数。

使用fine -name 可以列举出匹配的文件名，然后使用xargs cat 可以打印出文件名里的内容

因此使用 find . -name "*.h" -or -name '*.c' | xargs cat | wc -l

可以打印出当前目录下c和h文件的代码行数。

然后如果要进行筛选，可以使用grep进行正则表达式的匹配，过滤空行

使用grep -v ^$ 过滤空行

find . -name "*.h" -or -name '*.c' | xargs cat | grep -v ^$ | wc -l

写进makefile中

```
count:
    find . -name "*.h" -or -name "*.c" | xargs cat | grep -v '^$$'  |  wc -l
```

这里真是个大坑，shell脚本中$不能转义，得用两个$$表示一个$，在这里错了好久，makefile加入这条规则以后就可以make count了。

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?

-Wall 打开gcc的所有警告

-Werror，它要求gcc将所有的警告当成错误进行处理

可以更及时的发现潜在的错误，使得代码更符合规范。