



南開大學  
Nankai University

计算机学院  
计算机系统设计实验报告

# PA1-开天辟地的篇章：最简单的计算机

姓名：杨馨仪

学号：2011440

专业：计算机科学与技术

2024 年 3 月 19 日

# 目录

<b>1 概述</b>	<b>2</b>
1.1 实验目的	2
1.2 实验内容	2
<b>2 阶段一</b>	<b>2</b>
2.1 简单计算机模型	2
2.1.1 NEMU 执行流程	2
2.1.2 代码：实现正确的寄存器结构体	3
2.1.3 问题：究竟要执行多久？	4
2.1.4 谁来指示程序的结束？	5
2.2 基础设施：简易调试器	5
2.2.1 代码：实现单步执行、打印寄存器、扫描内存	5
2.2.2 Bug1 主机与虚拟机间复制粘贴问题	8
<b>3 阶段二</b>	<b>8</b>
3.1 词法分析	8
3.1.1 代码：实现算术表达式的词法分析	8
3.2 表达式求值	9
3.2.1 代码：实现算术表达式的递归求值	9
3.2.2 代码：实现带有负数的算术表达式的求值	12
3.2.3 代码：实现更复杂的表达式求值	12
3.2.4 代码：调试完善	13
3.2.5 Bug2 Linux 没有 pow() 函数？	14
3.2.6 代码：完善扫描内存的功能	14
<b>4 阶段三</b>	<b>15</b>
4.1 监视点	15
4.1.1 代码：实现监视点池的管理	15
4.1.2 问题：static 的使用	16
4.1.3 代码：实现监视点	16
4.1.4 监测点功能验证	19
4.2 断点	20
4.2.1 断点的工作原理	20
4.2.2 断点功能验证	21
4.2.3 问题：“一点也不能长”	21
4.2.4 问题：随心所欲的断点	22
4.2.5 问题：NEMU 的前世今生	22
4.3 i386 手册的学习	22
4.3.1 问题：通过目录定位关注的问题	22
4.3.2 必答题	23
<b>5 实验结论与感想</b>	<b>24</b>

# 1 概述

## 1.1 实验目的

了解寄存器结构和指令执行原理，搭建好系统的基础设施——一个简易的调试器，实现单步执行、打印寄存器状态、打印监测点信息、表达式求值、扫描内存、设置和删除监测点的功能。

## 1.2 实验内容

1. 了解寄存器结构，根据寄存器结构完善代码，使系统支持 EAX、EDX、ECX、EBX、ESI、EDI、ESP（以上为 32 位寄存器）；AX、DX、CX、BX、BP、SI、DI、SP（以上为 16 位寄存器）；AL、DL、CL、DL、AH、DH、CH、BH（以上为 8 位寄存器）寄存器。
2. 了解指令执行原理，了解函数调用接口位置与参数，据此先实现一个简易调试器的单步执行、内存扫描、寄存器信息打印功能。
3. 实现表达式求值以实现更多的调试器功能。首先需要确定支持的 token 类别，完成词法分析，之后实现表达式的递归求值。
4. 在实现表达式求值的基础上实现监测点的功能，综合起来可以实现断点。监测点需要实现对监测点池的管理，支持新建、删除、打印监测点，之后再在程序运行时进行监测点检查。

# 2 阶段一

## 2.1 简单计算机模型

### 2.1.1 NEMU 执行流程

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。`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 `assertion fail`。

然后通过调用 `load_img()` 函数（在 `nemu/src/monitor/monitor.c` 中定义）读入带有客户程序的镜像文件。让 monitor 直接把一个有意义的客户程序镜像 `guest prog` 读入到一个固定的内存位置 `0x100000`，这个程序是运行 NEMU 的一个参数，在运行 NEMU 的命令中指定，缺省时将把上文提到的 `mov` 程序作为客户程序（参考 `load_default_img()` 函数）。

然后调用 `restart()` 函数（在 `nemu/src/monitor/monitor.c` 中定义），它模拟了“计算机启动”的功能，进行一些和“计算机启动”相关的初始化工作，一个重要的工作就是将 `%eip` 的初值设置为刚才我们约定的内存位置 `0x100000`，这样就可以让 CPU 从我们约定的内存位置开始执行程序了。

最后通过调用 `welcome()` 函数输出欢迎信息和 NEMU 的编译时间。monitor 的初始化工作结束后，NEMU 会进入用户界面主循环 `ui_mainloop()`（在 `nemu/src/monitor/debug/ui.c` 中定义），输出 NEMU 的命令提示符：`(nemu)`

代码已经实现了几个简单的命令，它们的功能和 GDB 是很类似的。输入 `c` 之后，NEMU 开始进入指令执行的主循环 `cpu_exec()`（在 `nemu/src/monitor/cpu-exec.c` 中定义）。`cpu_exec()` 模拟了 CPU 的工作方式：不断执行指令。`exec_wrapper()` 函数（在 `nemu/src/cpu/exec/exec.c` 中定义）的功能让 CPU 执行当前 `%eip` 指向的一条指令，然后更新 `%eip`。已经执行的指令会输出到日志文件 `log.txt` 中。

### 2.1.2 代码：实现正确的寄存器结构体

在本部分初步了解到 NEMU 架构四个主要模块是 monitor、CPU、memory、设备。NEMU 启动时如何对 monitor 做初始化。在 PA0 中运行 NEMU 发现由于没有正确实现寄存器结构会出现如下错误：

```
+ CC src/device/io/mmio.c
+ CC src/device/io/port-io.c
+ CC src/device/serial.c
+ CC src/device/keyboard.c
+ CC src/misc/logo.c
+ CC src/monitor/diff-test/gdb-host.c
+ CC src/monitor/diff-test/protocol.c
+ CC src/monitor/diff-test/diff-test.c
+ CC src/monitor/monitor.c
+ CC src/monitor/debug/expr.c
+ CC src/monitor/debug/ui.c
+ CC src/monitor/debug/watchpoint.c
+ CC src/monitor/cpu-exec.c
+ CC src/memory/memory.c
+ CC src/main.c
+ LD build/nemu
./build/nemu -l ./build/nemu-log.txt
nemu: src/cpu/reg.c:21: reg_test: Assertion `reg_w(i) == (sample[i] & 0xffff)' failed.
Makefile:46: recipe for target 'run' failed
make: *** [run] Aborted (core dumped)
yxy@ubuntu:~/ics2017/nemu$ |
```

图 2.1: 错误信息

因此需要代码实现正确的寄存器结构。之后继续阅读代码了解启动后的运行逻辑和当前已实现功能。

要想实现正确的寄存器结构首先需要了解本实验所选择的寄存器结构——CPU 中的寄存器有：EAX、EDX、ECX、EBX、ESI、EDI、ESP（以上为 32 位寄存器）；AX、DX、CX、BX、BP、SI、DI、SP（以上为 16 位寄存器）；AL、DL、CL、DL、AH、DH、CH、BH（以上为 8 位寄存器）

观察当前的错误代码：

```
typedef struct {
    struct {
        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.
     */
    rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;

    vaddr_t eip;
} CPU_state;
```

图 2.2: 错误代码

以 EAX 为例，当前寄存器结构相当于将 EAX、AH、AX、AL 这四个寄存器分隔开了，但实际上它们并不是独立的，AX 是 EAX 的低 16 位，AH 是 AX 的高 8 位、AL 是 AX 的低 8 位；使用 struct

并不能反映出它们的同地址关系；，因此需要借助 union 来实现寄存器结构。

以 EAX 为例，根据 AX 是 EAX 的低 16 位，AH 是 AX 的高 8 位、AL 是 AX 的低 8 位，首先需要 union 联合体把这四者关联到一个地址，然后再把 eax 的值与之联合到一起；这个步骤涉及到 EAX、EDX、ECX、EBX、ESI、EDI、ESP 八个寄存器。最后以上寄存器和 EIP 一起构成完整的寄存器结构。

```

17  typedef struct {
18  union {
19      union{
20          uint32_t _32;
21          uint16_t _16;
22          uint8_t _8[2];
23      } gpr[8];
24      struct {
25          rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
26      };
27  };
28  /* Do NOT change the order of the GPRs' definitions. */
29
30  /* In NEMU, rtlreg_t is exactly uint32_t. This makes RTL instructions
31   * in PA2 able to directly access these registers.
32   */
33  vaddr_t eip;
34  } CPU_state;

```

图 2.3: 修正后代码

补全代码后再运行 qemu 可以看到如下运行结果：

```

+ CC src/misc/logo.c
+ CC src/monitor/diff-test/gdb-host.c
+ CC src/monitor/diff-test/protocol.c
+ CC src/monitor/diff-test/diff-test.c
+ CC src/monitor/monitor.c
+ CC src/monitor/debug/expr.c
+ CC src/monitor/debug/ui.c
+ CC src/monitor/debug/watchpoint.c
+ CC src/monitor/cpu-exec.c
+ CC src/memory/memory.c
+ CC src/main.c
+ LD build/nemu
./build/nemu -l ./build/nemu-log.txt
[src/monitor/monitor.c,47,load_default_img] No image is given. Use the default build-in image.
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 06:52:13, Mar 18 2024
For help, type "help"
(nemu)

```

图 2.4: 寄存器结构正确实现后 qemu 运行

### 2.1.3 问题：究竟要执行多久？

**Q:** 在 cmd\_d() 函数中，调用 cpu\_exec() 是传入了参数-1，这是什么意思？

**A:** cpu\_exec() 函数模拟 CPU 的工作方式，参数 uint64\_t n 表示要执行的指令条数，运行时只有执行完 n 条指令或者执行到结束指令才会退出执行。对于 uint64\_t 来说-1 代表 64 位无符号整数最大值 0xFFFF FFFF FFFF FFFF，因此传参-1 相当于一直执行指令直到执行到结束指令，也就是指令全部执行完毕。

### 2.1.4 谁来指示程序的结束？

**Q:** 在程序设计课上老师告诉你, 当程序执行到 `main0` 函数返回处的时候, 程序就退出了, 你对此深信不疑. 但你提否怀疑过, 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?

**A:** 反例: `assert` `exit` 都可以让程序退出, 因此并不是程序执行到 `main()` 函数的返回处就结束。

## 2.2 基础设施：简易调试器

### 2.2.1 代码：实现单步执行、打印寄存器、扫描内存

NEMU 可以随时了解客户程序执行的所有信息, 但外部调试器并不好获取这些信息。为提高调试的效率, 在 `monitor` 中实现了一个具有如下功能的简易调试器:

命令	格式	使用举例	说明
帮助 (1)	<code>help</code>	<code>help</code>	打印命令的帮助信息
继续运行 (1)	<code>c</code>	<code>c</code>	继续运行被暂停的程序
退出 (1)	<code>q</code>	<code>q</code>	退出 NEMU
单步执行	<code>si [N]</code>	<code>si 10</code>	让程序单步执行 <code>N</code> 条指令后暂停执行, 当 <code>N</code> 没有给出时, 缺省为 1
打印程序状态	<code>info</code> <code>SUBCMD</code>	<code>info r</code> <code>info w</code>	打印寄存器状态 打印监视点信息
表达式求值	<code>p EXPR</code>	<code>p \$eax + 1</code>	求出表达式 <code>EXPR</code> 的值, <code>EXPR</code> 支持的运算请见调试中的表达式求值小节
扫描内存 (2)	<code>x N EXPR</code>	<code>x 10 \$esp</code>	求出表达式 <code>EXPR</code> 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 <code>N</code> 个 4 字节
设置监视点	<code>w EXPR</code>	<code>w *0x2000</code>	当表达式 <code>EXPR</code> 的值发生变化时, 暂停程序执行
删除监视点	<code>d N</code>	<code>d 2</code>	删除序号为 <code>N</code> 的监视点

图 2.5: NEMU 调试器功能

**解析命令:** 只需仿照已有的 `help`、`c`、`q` 命令, 补全调试信息指令和函数的对应, 实现初步的命令类型解析, 将具体的参数解析放到具体命令的实现函数中:

```
53  /* TODO: Add more commands */
54  //PA1-1
55
56  {"si", "single-step execution", cmd_si},
57  {"info", "Print information", cmd_info},
58  // {"p", "Evaluate the expression", cmd_p},
59  {"x", "Scan memory", cmd_x},
60  // {"w", "Set the monitoring points", cmd_w},
61  // {"d", "Remove the monitoring points", cmd_d}
```

图 2.6: 解析命令代码

其中表达式求值、设置监测点和删除监测点的调试功能，在实现表达式求值和监测点后再进行实现，故在此处暂时注释掉。

参数解析的部分，需要用到函数 `strtok(char *, const char)`，该函数的作用是用指定的常量字符分割字符串，调用一次函数返回的是第一个子串，之后想要再获取子字符串，传入的字符串参数应为 `NULL`，函数默认使用上一次未分割完的字符串从上一次分割的部分开始分割。

**单步执行：**首先需要使用 `strtok()` 获取要执行的步数，此时获取的步数是字符串形式，如果字符串为空也就是没有参数，默认执行 1 步；非空的话借助 `sscanf()` 函数将字符串转换为数字作为步数。执行时调用 `cpu_exec()` 函数，传的参数就是要执行的步数。

```
89  static int cmd_si(char *args){
90      char *arg = strtok(args, " ");
91      int N;
92      if(arg == NULL) cpu_exec(1);
93      else{
94          if(sscanf(arg, "%d", &N) == 1) cpu_exec(N);
95          else printf("Error parameter. \n");
96      }
97      return 0;
98  }
```

图 2.7: 单步执行代码

**打印寄存器：**打印寄存器是打印程序状态的一种情况，在打印程序状态函数中首先使用 `strtok()` 获取要打印的信息类型参数，对于参数为 `w` 的（打印监视点信息）先不做处理；对于参数为 `r` 的，打印寄存器状态，依次打印 `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, `edi`, `eip` 的值；对于参数既不是 `r` 也不是 `w` 的，输出错误提示信息。

```

100 static int cmd_info(char *args) {
101     char *arg = strtok(args, " ");
102     if(arg == NULL) printf("Lack of parameter. \n");
103     else {
104         if(strcmp(arg, "r") == 0) {
105             printf("----information of regs----\n");
106             printf(" eax : 0x%08x \n", cpu.eax);
107             printf(" ecx : 0x%08x \n", cpu.ecx);
108             printf(" edx : 0x%08x \n", cpu.edx);
109             printf(" ebx : 0x%08x \n", cpu.ebx);
110             printf(" esp : 0x%08x \n", cpu.esp);
111             printf(" ebp : 0x%08x \n", cpu.ebp);
112             printf(" esi : 0x%08x \n", cpu.esi);
113             printf(" edi : 0x%08x \n", cpu.edi);
114             printf(" eip : 0x%08x \n", cpu.eip);
115             printf("-----\n");
116         }
117         else if (strcmp(arg, "w") == 0) info_wplist();
118         else printf("Error parameter. \n");
119     }
120     return 0;
121 }

```

图 2.8: 打印寄存器代码

**扫描内存：**扫描内存需要获取扫描的范围 N 和起始地址 addr 两个参数，因此需要首先使用两次 strtok() 获取两个参数，然后借助 sscanf() 函数将获取到的参数字符串转换为数值，然后从 addr 开始使用 vaddr\_read() 函数获取当前地址的内存信息，每次地址递增 4，一共获取 N 条内存信息。这里的 addr 先按照数值来处理，在完成表达式求值的实现后，addr 可以为一个表达式。

```

123 static int cmd_x(char *args)
124 {
125     char *arg1 = strtok(args, " ");
126     char *arg2 = strtok(NULL, "");
127     if (arg1 == NULL || arg2 == NULL) printf("Lack of parameter. \n");
128     else {
129         int N;
130         vaddr_t addr;
131         if(sscanf(arg1, "%d", &N) == 1 && sscanf(arg2, "0x%08x", &addr) == 1){
132             int i;
133             if(N > 0){
134                 for (i = 0; i < N; i++) {
135                     printf("0x%06x : 0x%06x \n", addr, vaddr_read(addr, 4));
136                     addr = addr + 4;
137                 }
138             }
139             else printf("Error parameter. \n");
140         }
141         else printf("Error parameter. \n");
142     }
143     return 0;
144 }

```

图 2.9: 扫描内存代码

## 功能验证

运行 qemu, 依次对单步执行、寄存器信息打印、扫描内存功能进行测试：



```

(nemu) help
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - single-step execution
info - Print information
x - Scan memory
(nemu) si 4
100000: b8 34 12 00 00      movl $0x1234,%eax
100005: b9 27 00 10 00      movl $0x100027,%ecx
10000a: 89 01               movl %eax,(%ecx)
10000c: 66 c7 41 04 01 00    movw $0x1,0x4(%ecx)
(nemu) si -1
nemu: HIT GOOD TRAP at eip = 0x00100026

(nemu) info r
----information of regs----
eax : 0x00000000
ecx : 0x00100027
edx : 0x6329e76d
ebx : 0x00000002
esp : 0x6514f6fc
ebp : 0x2248e594
esi : 0x64c52dd8
edi : 0x2f036f0f
eip : 0x00100027
-----
(nemu) x 4 0x100000
0x100000 : 0x1234b8
0x100004 : 0x27b900
0x100008 : 0x1890010
0x10000c : 0x441c766
(nemu) 

```

图 2.10: 简易调试器功能验证

### 2.2.2 Bug1 主机与虚拟机间复制粘贴问题

对于这个问题，我根据同学们在课程群里的建议，下载了 Xterminal，使用 SSH 远程连接 Ubuntu 虚拟机，解决了复制粘贴的问题。

## 3 阶段二

### 3.1 词法分析

#### 3.1.1 代码：实现算术表达式的词法分析

通过阅读实验指导书，大致了解到表达式求值中至少要实现含括号的加减乘除的十进制运算。除运算符外，还有运算数的规则要设计，涉及到的有十进制的匹配规则。

因此在实现过程中首先在 rules 中添加所要识别的符号规则，并在枚举中添加对应的类型。添加的运算符有 +、-、\*、/、(、)；添加的运算数有十进制。由于匹配的过程中会依次遍历 rules 数组，因此应该将想要优先匹配的写在前面。

代码仅展示涉及到正则表达式和元字符的规则：

```

33     {"+", TK_NOTYPE},      // spaces
34     {"\\+", TK_PLUS},     // plus
35     {"-", TK_SUB},        // sub
36     {"\\*", TK_MUL},      // multi
37     {"\\/ ", TK_DIV},     // divide
38     {"\\(", TK_LBRACE},   // left
39     {"\\)", TK_RBRACE},   // right
40     {"==", TK_EQ},        // equal
41     {"[1-9][0-9]*", TK_DEC}, // dec number
42 };

```

接下来我们需要进行对 token 的识别并将其记录下来。在给出的框架中已有部分这部分的代码。可以通过阅读代码了解到 token 识别是遍历整个输入值字符串，对于已读未识别的部分逐一与所设计的规则做匹配判断，如果匹配上的话，就将识别到的 token 存入 Token 数组，更新识别开始的位置。

在 token 存储的过程中，会存储类型与内容，存储到 Token 数组的类型就是匹配上的规则对应的类型；对于运算符来说内容不需要存储，因为只要是这个类型的 token，类型就固定了，因此只需要存储十进制的内容，可以使用 strcpy() 也可以逐字符存储，但要记得在末尾加上结束符。

```

// PA1-2
int j;
switch (rules[i].token_type) {
    case TK_NOTYPE: break;
    case TK_DEC:
        tokens[nr_token].type = rules[i].token_type;
        for(j=0; j < substr_len; j++)
            tokens[nr_token].str[j] = e[position - substr_len + j];
        tokens[nr_token].str[j] = '\0';
        nr_token++;
        break;
    default:
        tokens[nr_token].type = rules[i].token_type;
        tokens[nr_token].str[0] = '\0';
        nr_token++;
        break;
}

```

## 3.2 表达式求值

### 3.2.1 代码：实现算术表达式的递归求值

**整体思路：**根据表达式的归纳定义特性，使用递归来进行表达式求值：长表达式由短表达式构成，求解长表达式就首先求解短表达式。在程序中使用两个整数 p、q 来指示子表达式的起始位置，对于  $p < q$  的，是错误的表达式；对于  $p = q$  的，是一个单一的数值，可能是十进制、十六进制和寄存器；对于  $p < q$  的，是一个规范完整的子表达式，其中如果该子表达式被一对匹配的括号包围，就要去计算括号中子表达式的结果来作为该子表达式的值，其他情况要去找子表达式中优先级最低的运算符，分别去计算两个运算数子表达式的值，最后根据运算符做计算得到该子表达式的值。

```

if (p>q)
{
    printf("Wrong Expression\n");
    assert(0);
}

```

单一的数值的表达式计算需要完成字符串到数值的转换；被括号包围的判断需要单独完成；优先级最低的运算符的查找需要遍历获取。

**单个 token：**对于十进制数字，使用 sscanf 函数 %d 完成转换。

```

else if(p == q)
{
    if(tokens[p].type == TK_DEC) sscanf(tokens[p].str, "%d", &val);
    else
    {
        printf("Wrong Expression\n");
        assert(0);
    }
    return val;
}

```

**被括号包裹的 token：**要判断子表达式是否被一对匹配的括号包围，首先要保证子表达式的起始位置和结束位置分别是前括号和后括号。但仅仅是不够的，因为可能是子表达式中包含了多个括号包围的子表达式，恰有两个括号一个在头一个在尾，因此还需要排除这种情况，在一般的括号检查的基础上，抛开起始和结束位置的括号来进行检查，初始计数为 0，遇到前括号 +1 后括号 -1，要求计数不能小于 0，这样就对上述情况实现了排除。

被括号包围的判断函数：

```

163 bool check_paretheses(int p, int q) {
164     if(!(tokens[p].type == TK_LBRACE && tokens[q].type == TK_RBRACE)) return false;
165     int num = 0;
166     for( int i=p+1;i<q;i++){
167         if(tokens[i].type == TK_LBRACE) num++;
168         else if(tokens[i].type == TK_RBRACE) num--;
169         if(num<0) return false;
170     }
171     if(num == 0) return true;
172     else return false;
173 }

```

该子表达式计算返回为递归调用计算起始位置为 p+1，结束位置为 q-1 的表达式值。

**一般的运算式 token：**

- 最低优先级符号查找

负数和指针解引用优先级比较高，由于它们的特殊性，对它们的处理单独进行，因此在进行优先级判断的时候，对于这两种情况不返回符号位置，而是返回 -1 作为标识，在后面进行单独的处理。

明确其余运算符的优先级：

$$\{ *, / \} > \{ +, - \}$$

```

133 int type_prior(int token_type)
134 {
135     if(token_type == TK_MUL || token_type == TK_DIV) return 1;
136     else if(token_type == TK_PLUS || token_type == TK_SUB) return 2;
137     else return -1;
138 }

```

首先括号和数值的优先级最高，最低优先级符号不应该位于括号中。因此需要对括号位置进行判断，初始计数为 0，遇到前括号 +1 后括号-1，这样只有计数为 0 时遍历到的符号才不在括号中。在遍历过程中如果遇到其他 token，首先判断是不是数值，以及计数是否为 0。如果确定是数值或者在括号中，直接继续向后遍历；否则的话就去与当前已经遍历到的最低优先级符号做比较，判断是否需要更改记录的最低优先级符号类型与位置。

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是我们要找的运算符。

```

140 int dominant_op(int p, int q) {
141     int cur_op = -1;
142     int op_position = -1;
143     int count = 0;
144     for(int i=p; i<=q; i++){
145         if(tokens[i].type == TK_LBRACE) count++;
146         else if(tokens[i].type == TK_RBRACE) count--;
147         else if(tokens[i].type == TK_DEC || count != 0) continue;
148         else if(count == 0){
149             int token_prior = type_prior(tokens[i].type);
150             int curop_prior = type_prior(cur_op);
151             if(token_prior > 0) {
152                 if(token_prior >= curop_prior){
153                     cur_op = tokens[i].type;
154                     op_position = i;
155                 }
156             }
157         }
158     }
159     return op_position;
160 }

```

- 数值计算

记找到的最低优先级符号位置为 op，这时找到的都是双目运算符，左右运算数分别为起始位置为 p，结束位置为 op-1 的表达式和起始位置为 op+1，结束位置为 q 的表达式。递归调用函数计算左右运算数的值，然后根据 token 的类型进行相应运算，得到运算结果就是子表达式的值。

```

else
{
    int op = dominant_op(p,q);
    int val1 = eval(p,op-1);
    int val2 = eval(op+1,q);
    switch(tokens[op].type)
    {
        case(TK_PLUS): return val1+val2;
        case(TK_SUB): return val1-val2;
        case(TK_MUL): return val1*val2;
        case(TK_DIV): return val1/val2;
        case(TK_EQ): return val1==val2;
        default: assert(0);
    }
}

```

### 3.2.2 代码：实现带有负数的算术表达式的求值

如果是负号与指针解引用，应当是在当前子表达式只有这两个符号的时候才需要做运算求值。首先不断向后遍历找到最后一个运算符，然后从它向前做计算。首先先算出符号的表达式值，然后根据符号判断是取相反数还是进行地址读取，更新运算值，不停向前迭代，直至到达该子表达式的开始。

```

if(op == -1)
{
    int k = p;
    while((tokens[k].type == TK_NEG || tokens[k].type == TK_DEREF)) k++;
    val = eval(k,q);
    for(int i=k-1; i >= p; i--)
    {
        if(tokens[i].type == TK_NEG) val = (-1)*val;
        else if(tokens[i].type == TK_DEREF) val = vaddr_read(val,4);
        return val;
    }
}

```

### 3.2.3 代码：实现更复杂的表达式求值

进一步实现了比较运算、逻辑运算等。除运算符外，还在之前代码的基础上补充了十六进制和寄存器的运算规则，对于这些新的运算符，也设置了新的优先级规则。

$$\{ *,/, \% \} > \{ +, - \} > \{ \ll, \gg \} > \{ <=, <, >=, > \} > \{ ==, != \} > \&\& > \parallel$$

```

{"\\%", TK_MOD}, // remainder
{"\\(", TK_LBRACE}, // left
{"\\)", TK_RBRACE}, // right
{"<<", TK_LS}, // left shift
{">>", TK_RS}, // right shift
{">=", TK_GE}, // greater than or equal
{">", TK_GT}, // greater than
{"<=", TK_LE}, // less than or equal
{"<", TK_LT}, // less than
{"!=", TK_NEQ}, // not equal
{"==", TK_EQ}, // equal
{"&&", TK_AND}, // and
{"\\|\\|", TK_OR}, // or
{"[1-9][0-9]*", TK_DEC}, // dec number
{"0[xX][0-9a-fA-F]+", TK_HEX}, // hex number
{"\\$(eax|ecx|edx|ebx|ebp|esi|edi|esp|eip)", TK_REG} // register

```

```

int type_prior(int token_type)
{
    if(token_type == TK_MUL || token_type == TK_DIV || token_type == TK_MOD) return 1;
    else if(token_type == TK_PLUS || token_type == TK_SUB) return 2;
    else if(token_type == TK_LS || token_type == TK_RS) return 3;
    else if(token_type == TK_GT || token_type == TK_GE || token_type == TK_LT || token_type == TK_LE) return 4;
    else if(token_type == TK_EQ || token_type == TK_NEQ) return 5;
    else if(token_type == TK_AND) return 6;
    else if(token_type == TK_OR) return 7;
    else return -1;
}

```

### 3.2.4 代码：调试完善

这部分的实现比较简单，需要在 ui.c 里增加 cmd\_p 函数。这里传参的时候不需要再使用 strtok 分隔参数，直接将 p 后面的整个字符串作为参数来调用 expr() 函数，直接得到计算结果并输出（以十进制和十六进制两种形式输出）。

```

static int cmd_p(char *args){
    bool *success = false;
    int val = expr(args, success);
    printf("%d( 0x%06x )\n",val, val);
    return 0;
}

```

**功能验证：**运行 qemu, 进行表达式求值测试：

#### 1. 算术表达式

```

(nemu) p (1+3)*2
[src/monitor/debug/expr.c,117,make_token] match rules[6] = "(" at position 0 with len 1: (
[src/monitor/debug/expr.c,117,make_token] match rules[18] = "[1-9][0-9]*" at position 1 with len 1: 1
[src/monitor/debug/expr.c,117,make_token] match rules[1] = "+" at position 2 with len 1: +
[src/monitor/debug/expr.c,117,make_token] match rules[18] = "[1-9][0-9]*" at position 3 with len 1: 3
[src/monitor/debug/expr.c,117,make_token] match rules[7] = ")" at position 4 with len 1: )
[src/monitor/debug/expr.c,117,make_token] match rules[3] = "*" at position 5 with len 1: *
[src/monitor/debug/expr.c,117,make_token] match rules[18] = "[1-9][0-9]*" at position 6 with len 1: 2
8( 0x000008 )

```

#### 2. 含负数的算数表达式

```

(nemu) p 1+-2
[src/monitor/debug/expr.c,117,make_token] match rules[18] = "[1-9][0-9]*" at position 0 with len 1: 1
[src/monitor/debug/expr.c,117,make_token] match rules[1] = "+" at position 1 with len 1: +
[src/monitor/debug/expr.c,117,make_token] match rules[2] = "-" at position 2 with len 1: -
[src/monitor/debug/expr.c,117,make_token] match rules[18] = "[1-9][0-9]*" at position 3 with len 1: 2
-1( 0xffffffff )

```

#### 3. 寄存器运算

```

(nemu) p $eip>>2
[src/monitor/debug/expr.c,117,make_token] match rules[20] = "\\$(eax|ecx|edx|ebx|ebp|esi|edi|esp|eip)" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,117,make_token] match rules[9] = ">>" at position 4 with len 2: >>
[src/monitor/debug/expr.c,117,make_token] match rules[18] = "[1-9][0-9]*" at position 6 with len 1: 2
262144( 0x040000 )

```

#### 4. 错误运算

```
(nemu) p (3+4))*((2-1)
[src/monitor/debug/expr.c,117,make_token] match rules[6] = "\" at position 0 with len 1: (
[src/monitor/debug/expr.c,117,make_token] match rules[18] = "[1-9][0-9]*" at position 1 with len 1: 3
[src/monitor/debug/expr.c,117,make_token] match rules[1] = "+" at position 2 with len 1: +
[src/monitor/debug/expr.c,117,make_token] match rules[18] = "[1-9][0-9]*" at position 3 with len 1: 4
[src/monitor/debug/expr.c,117,make_token] match rules[7] = ")" at position 4 with len 1: )
[src/monitor/debug/expr.c,117,make_token] match rules[7] = ")" at position 5 with len 1: )
[src/monitor/debug/expr.c,117,make_token] match rules[3] = "*" at position 6 with len 1: *
[src/monitor/debug/expr.c,117,make_token] match rules[6] = "\" at position 7 with len 1: (
[src/monitor/debug/expr.c,117,make_token] match rules[6] = "\" at position 8 with len 1: (
[src/monitor/debug/expr.c,117,make_token] match rules[18] = "[1-9][0-9]*" at position 9 with len 1: 2
[src/monitor/debug/expr.c,117,make_token] match rules[2] = "-" at position 10 with len 1: -
[src/monitor/debug/expr.c,117,make_token] match rules[18] = "[1-9][0-9]*" at position 11 with len 1: 1
[src/monitor/debug/expr.c,117,make_token] match rules[7] = ")" at position 12 with len 1: )
Wrong Expression
nemu: src/monitor/debug/expr.c:304: expr: Assertion `0' failed.
Makefile:46: recipe for target 'run' failed
make: *** [run] Aborted (core dumped)
```

均识别正确、运算结果正确。

### 3.2.5 Bug2 Linux 没有 pow() 函数？

编译时需要链接数学库，加上 -lm

### 3.2.6 代码：完善扫描内存的功能

此外还需要修改内存扫描函数 cmp\_x, 将第二个参数改为一个表达式，通过计算去获得地址值。因此重写了 cmp\_x 函数中 addr 的获取代码。

```
static int cmd_x(char *args)
{
    char *arg1 = strtok(args, " ");
    char *arg2 = strtok(NULL, "");
    if (arg1 == NULL || arg2 == NULL) printf("Lack of parameter. \n");
    else {
        int N;
        vaddr_t addr;
        if(sscanf(arg1,"%d",&N) == 1){
            int i;
            bool* success = false;
            int val = expr(arg2, success);
            addr = val;
            if(N > 0){
```

**功能验证：**进行内存扫描，将地址替换为表达式，得到的结果与直接使用运算结果做地址扫描一致。

```
(nemu) x 4 0x0ffffff +1
[src/monitor/debug/expr.c,117,make_token] match rules[19] = "0[xX][0-9a-fA-F]+" at position 0 with len 8: 0x0ffffff
[src/monitor/debug/expr.c,117,make_token] match rules[0] = "+" at position 8 with len 1:
[src/monitor/debug/expr.c,117,make_token] match rules[1] = "+" at position 9 with len 1: +
[src/monitor/debug/expr.c,117,make_token] match rules[18] = "[1-9][0-9]*" at position 10 with len 1: 1
0x100000 : 0x1234b8
0x100004 : 0x27b900
0x100008 : 0x1890010
0x10000c : 0x441c766
(nemu) |
```

图 3.11: 内存扫描部分的表达式求值



## 4 阶段三

### 4.1 监视点

#### 4.1.1 代码：实现监视点池的管理

除去代码框架中给出的序号和后项指针外，监测点中还应该存放监测的表达式，以及表达式的新值旧值，以实现新值旧值更新和表达式值变化的判断。

```
10      /* TODO: Add more members if necessary */
11      char expr[1024];
12      int new_val;
13      int old_val;
```

通过阅读代码可以得知，head 和 free\_ 两个指针分别链接着使用中的监测点和空闲的监测点，这些检测点统一存放在监测点池 wp\_pool 中。初始时，head 为空，说明现在没有设置监测点；free\_ 指向 wp\_pool，说明目前监测点池中的全部为空闲的监测点结构体。

#### 1. 获取空闲监视点结构体

如果要获取空闲监视点结构体，就要从 free\_ 指针指向的空闲监测点链表获取。

首先判断是否还有空闲的监测点结构，通过判断 free\_ 是否为空或者计数已有监测点看数量是否已达 NR\_WP。如果没有空闲的监测点结构，通过 assert(0) 终止。

在还有空闲的监测点结构的情况下进行结构分配。分配过程中分配的是 free\_ 指针指向的那个空闲结构。通过 free\_ 获取它指向的监测点结构在监测点池中的序号，更新 free\_ 指针（指向它的 next）。

然后将监测点池中对应序号的结构体插入 head 维护的链表：该结构体的 next 指向当前 head，然后将 head 指向该结构体，这样就将结构体插入了链表首部。

```
WP* new_wp()
{
    if(wpNUM == NR_WP){
        printf("Insufficient Memory\n");
        assert(0);
    }

    int no = free_>NO;
    free_ = free_>next;
    wp_pool[no].next = head;
    head = &wp_pool[no];
    wpNUM++;
    return head;
}
```

#### 2. 释放监视点结构体



释放结构体的时候会指定要释放的监测点，需要在 head 维护的使用中的监测点链表中找到这个监测点。

如果这个监测点在首部（也就是它是 head 指向的监测点），那么需要将 head 指针指向 head（也就是当前监测点）的 next，将监测点移除使用中的队列；然后将要释放的监测点的 next 指向 free\_，再将 free\_ 更新成当前监测点，将该监测点插入到空闲监测点链表的首部。

如果这个监测点不在首部，那就需要遍历使用中监测点链表，比较监测点序号来判断是否为目标监测点。由于链表中间监测点的删除需要让被删除监测点的前一个监测点的 next 指向被删除监测点的后一个监测点，而这又是一个单向链表，没有前向指针，因此在判断目标监测点的时候需要从 head 开始遍历比较遍历到节点的 next 是否为目标监测点。如果是的话就将当前遍历到的监测点的 next 指向目标监测点的 next，再按照监测点在首部情况中描述的方法将监测点插入空闲监测点链表的首部。

```
void free_wp(WP* wp){
    if(wp == head){
        head = head->next;
        wp->next = free_;
    }
    else{
        WP* p;
        p = head;
        while(p->next){
            if(p->next->NO == wp->NO){
                p->next = wp->next;
                wp->next = free_;
                break;
            }
            p = p->next;
        }
    }
    free_ = wp;
    wpNUM--;
}
```

#### 4.1.2 问题：static 的使用

**Q:** 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？

**A:** static 在此处表示全局静态变量。全局静态变量不能被其它文件访问，在这里使用 static 可以保证监测点不会被外部文件访问和更改，保证了调试监测的安全性。

#### 4.1.3 代码：实现监视点

##### 1. 设置监测点

设置检查点需要在 ui.c 里增加 cmp\_w 函数。直接将 w 后的字符串做参数，也就是检测点的 expr 去调用 watchpoint.c 中的监测点设置函数。

```
static int cmd_w(char *args){
    int no = set_wp(args);
    printf("Set Watch Point NO.%d Successfully!\n",no);
    return 0;
}
```

在监测点设置函数中首先通过 new\_wp 函数获取一个空闲的监测点结构；然后将调用 expr() 函数计算传入的表达式参数的值，作为监测点的旧值和新值；再将传入的表达式参数作为该监测点表达式成员变量 expr 的值。

```
int set_wp(char *e){
    WP* New_Wp = new_wp();
    bool *success = false;
    New_Wp->old_val = expr(e, success);
    New_Wp->new_val = New_Wp->old_val;
    strcpy(New_Wp->expr, e);
    printf("Set Watchpoint %d : \n",New_Wp->NO);
    printf("          expr  : %s \n",New_Wp->expr);
    printf("          value : 0x%06x \n",New_Wp->old_val);
    return New_Wp->NO;
}
```

## 2. 监测点信息打印

设置检查点需要在 ui.c 里的 cmp\_info 函数中 info w 情况的分支语句中补全，直接调用 watchpoint.c 中的监测点信息输出函数。

```
120         else if (strcmp(arg,"w") == 0)
121             info_wplist();
```

首先判断是否有使用中的监测点，也就是判断 head 是否为 NULL 或者判断使用中监测点数是否为 0。如果没有，输出提示“没有监测点”，停止打印。

在有使用中监测点的情况下，通过 next 指针遍历使用中监测点链表，逐一打印每个监测点的信息，包括序号、表达式、旧值和新值。

```
void info_wplist(){
    if(wpNUM == 0){
        printf("No Watch Point\n");
        return;
    }
    WP* wp =head;
    while(wp){
        printf("NO. %d : \n",wp->NO);
        printf("          expr:      %s \n",wp->expr);
        printf("          old value: 0x%06x \n",wp->old_val);
        printf("          new value: 0x%06x \n",wp->new_val);
        printf("\n");
        wp = wp->next;
    }
}
```

## 3. 删除监测点

设置检查点需要在 ui.c 里增加 cmd\_d 函数。借助 strtok() 和 scanf() 获取要删除监测点的序号，作为参数调用 watchpoint.c 中的监测点删除函数，并对参数不规范的情况做处理。

```
static int cmd_d(char *args){
    char *arg = strtok(args, " ");
    int N;
    if (arg == NULL) {
        printf("Lack of parameter. \n");
    }
    else {
        if(sscanf(arg,"%d",&N) == 1 && N >= 0){
            del_wp(N);
        }
        else printf("Error parameter. \n");
    }
    return 0;
}
```

首先判断是否有使用中的监测点，如果没有就中止退出。

如果有的话，先通过参数定位要删除的目标监测点，然后调用 free\_wp() 函数释放该监测点。

```
bool del_wp(int n){
    if(wpNUM == 0){
        printf("No Watch Point\n");
        assert(0);
    }
    WP *Del_Wp = &wp_pool[n];
    free_wp(Del_Wp);
    printf("Watchpoint %d has been deleted. \n",Del_Wp->NO);
    return true;
}
```

#### 4. 执行过程中的监测点检测

在执行过程中，每当 cpu\_exec() 执行完一条指令，就对所有待监视的表达式进行求值检查，如果有表达式的值发生变化，会触发监测点，程序暂停。因此需要在 cpu-exec.c 中的 DEBUG 部分调用 watchpoint.c 中的监测点扫描函数，获取扫描到的第一个表达式的值发生变化的监测点（也就是被触发的监测点），并输出该监测点的信息。

```
#ifdef DEBUG
    /* TODO: check watchpoints here. */
    WP *wp = scan_wp();
    if(wp != NULL){
        nemu_state = NEMU_STOP;
        printf("Watch point NO.%d's expr( %s ) value changes. \n",wp->NO,wp->expr);
        printf("old value is 0x%06x\n",wp->old_val);
        printf("new value is 0x%06x\n",wp->new_val);
    }
#endif
```

在监测点扫描函数中，首先判断是否有使用中的监测点，如果没有一定不会触发中断，可以直接返回空值表示没有监测点被触发。

如果有使用中的监测点，通过 next 指针遍历使用中监测点链表，首先将重新计算前也就是现在的新值赋给旧值，然后通过调用 `expr()` 函数计算该监测点表达式的值作为新值。比较新值旧值是否相等，如果不等，触发监测点，返回当前监测点到 `cpu-exec.c` 中，如果相等则继续向后遍历直至遇到表达式值发生变化的监测点或遍历完所有使用中的监测点。

```
WP* scan_wp(){
    if(wpNUM == 0){
        return NULL;
    }
    WP* wp = head;
    bool *success = false;
    while(wp){
        if(wp->new_val != wp->old_val){
            wp->old_val = wp->new_val;
        }
        wp->new_val = expr(wp->expr, success);
        if(wp->new_val != wp->old_val) return wp;
        wp = wp->next;
    }
    return NULL;
}
```

#### 4.1.4 监测点功能验证

运行 qemu, 尝试设置监测点：

```
(nemu) w $eip!=0x10000c
[src/monitor/debug/expr.c,117,make_token] match rules[20] = "\$(eax|ecx|edx|ebx|ebp|esi|edi|esp|eip)" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,117,make_token] match rules[14] = "=" at position 4 with len 2: !=
[src/monitor/debug/expr.c,117,make_token] match rules[19] = "[0-9a-fA-F]+" at position 6 with len 8: 0x10000c
Set Watchpoint 0 :
    expr : $eip!=0x10000c
    value : 0x000001
Set Watch Point NO.0 Successfully!
(nemu) w $ebp
[src/monitor/debug/expr.c,117,make_token] match rules[20] = "\$(eax|ecx|edx|ebx|ebp|esi|edi|esp|eip)" at position 0 with len 4: $ebp
Set Watchpoint 1 :
    expr : $ebp
    value : 0x4c9d46fe
Set Watch Point NO.1 Successfully!
```

图 4.12: 新建监测点

进行断点信息打印检测是否设置成功

```
(nemu) info w
NO. 1 :
    expr:      $ebp
    old value: 0x4c9d46fe
    new value: 0x4c9d46fe

NO. 0 :
    expr:      $eip!=0x10000c
    old value: 0x000001
    new value: 0x000001
```

图 4.13: 打印监测点信息

断点超过设置，信息正确打印。

根据断点信息删除监测点，并打印监测点信息进行验证：

```
(nemu) d 1
Watchpoint 1 has been deleted.
(nemu) info w
NO. 0 :
      expr:      $eip!=0x10000c
      old value:  0x000001
      new value:  0x000001
```

图 4.14: 删除监测点

成功删除监测点。

尝试执行并触发监测点，可以看到在监测点的值发生变化时程序中断：

```
(nemu) si 5
100000: b8 34 12 00 00          movl $0x1234,%eax
[src/monitor/debug/expr.c,117,make_token] match rules[20] = "$eax|ecx|edx|ebx|ebp|esi|edi|esp|eip)" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,117,make_token] match rules[14] = "=" at position 4 with len 2: !=
[src/monitor/debug/expr.c,117,make_token] match rules[19] = "[0-9a-fA-F]+" at position 6 with len 8: 0x10000c
100005: b9 27 00 10 00          movl $0x100027,%ecx
[src/monitor/debug/expr.c,117,make_token] match rules[20] = "$eax|ecx|edx|ebx|ebp|esi|edi|esp|eip)" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,117,make_token] match rules[14] = "=" at position 4 with len 2: !=
[src/monitor/debug/expr.c,117,make_token] match rules[19] = "[0-9a-fA-F]+" at position 6 with len 8: 0x10000c
10000a: 89 01                  movl %eax,%ecx
[src/monitor/debug/expr.c,117,make_token] match rules[20] = "$eax|ecx|edx|ebx|ebp|esi|edi|esp|eip)" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,117,make_token] match rules[14] = "=" at position 4 with len 2: !=
[src/monitor/debug/expr.c,117,make_token] match rules[19] = "[0-9a-fA-F]+" at position 6 with len 8: 0x10000c
Watch point NO.0's expr( $eip!=0x10000c ) value changes.
old value is 0x000001
new value is 0x000000
```

图 4.15: 触发监测点

## 4.2 断点

### 4.2.1 断点的工作原理

简单来说，CPU 通过一个名为 int 3 的特殊陷阱来实现断点。int 特指 x86 中的“陷阱指令”——调用预定义的中断处理函数。x86 支持 int 指令带有 8 个操作数，用于指定中断的种类，因此理论上支持 256 种陷阱。前 32 种由 CPU 为自己保留，int 3 被称为“调试器陷阱”。

INT 3 指令生成一个特殊的单字节操作码 (CC)，用于调用异常处理程序。（这种单字节形式很有价值，因为它可用于用断点替换任何指令的第一个字节，包括其他单字节指令，而不会覆盖其他代码）。

要在跟踪进程中的某个目标地址处设置断点，调试器将执行以下操作：

1. 记住存储在目标地址的数据
2. 用 int 3 指令替换目标地址的第一个字节

然后，当调试器要求 OS 运行该进程时，该进程将运行并最终触发 int 3。这是调试器再次进入的地址，接收到其子进程（或跟踪进程）已停止的信号。它可以：

1. 用原始指令替换目标地址处的 int 3 指令

2. 将跟踪过程的指令指针向后滚动一个。这是必需的，因为此时指令指针现在指向 int 3 之后，已经执行中断指令。
3. 允许用户以某种方式与进程交互，因为进程仍然在期望的目标地址处停止。此时调试器允许您查看变量值，调用堆栈等的部分。
4. 当用户想要继续运行时，除非用户要求取消断点，否则调试器将负责将断点放回目标地址（因为它已在步骤 1 中删除）。

#### 4.2.2 断点功能验证

设置监测点 \$eip==0x10000a, 执行程序:

```
(nemu) w $eip==0x10000a
[src/monitor/debug/expr.c,117,make_token] match rules[20] = "\$(eax|ecx|edx|ebx|ebp|esi|edi|esp|eip)" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,117,make_token] match rules[15] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,117,make_token] match rules[19] = "0[xX][0-9a-fA-F]+" at position 6 with len 8: 0x10000a
Set Watchpoint 0 :
    expr : $eip==0x10000a
    value : 0x000000
Set Watch Point NO.0 Successfully!
(nemu) si -1
[src/monitor/debug/expr.c,117,make_token] match rules[20] = "\$(eax|ecx|edx|ebx|ebp|esi|edi|esp|eip)" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,117,make_token] match rules[15] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,117,make_token] match rules[19] = "0[xX][0-9a-fA-F]+" at position 6 with len 8: 0x10000a
[src/monitor/debug/expr.c,117,make_token] match rules[20] = "\$(eax|ecx|edx|ebx|ebp|esi|edi|esp|eip)" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,117,make_token] match rules[15] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,117,make_token] match rules[19] = "0[xX][0-9a-fA-F]+" at position 6 with len 8: 0x10000a
Watch point NO.0's expr( $eip==0x10000a ) value changes.
old value is 0x000000
new value is 0x000001
```

图 4.16: 断点设置与触发

可以看到程序中断，输出此时的寄存器信息做检查

```
(nemu) info r
---information of regs---
eax : 0x00001234
ecx : 0x00100027
edx : 0x52a76e76
ebx : 0x3f99c30e
esp : 0x7629fdb6
ebp : 0x72e96b50
esi : 0x5da5ac82
edi : 0x44661838
eip : 0x0010000a
-----
```

断点成功设置。

#### 4.2.3 问题：“一点也不能长”

**Q:** 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？

**A:**

- 是必须的, 因为当我们在调试器中对代码的某一行设置断点时, 会把这里本来指令的第一个字节保存起来, 然后写入一条 `int3` 指令, 机器码为 `0xcc`, 仅有一个字节, 设置和取消断点时也需要保存和恢复一个字节。
- 不能正常工作, 因为 `int3` 断点将被调试进程中对地址处的字节替换为 `0xcc`, 当 `int3` 指令的长度变成 2 个字节, 其他指令同 x86, 会导致这里本来指令的第一个字节被 2 个字节所代替, 空间不够, 不正确。

#### 4.2.4 问题: 随心所欲的断点

**Q:** 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。

**A:** 将断点设置在指令的非首字节, 就无法检测到断点。因为会检测函数的地址, 读取它的第一个字节, 判断是否等于 `0xCCH`。

#### 4.2.5 问题: NEMU 的前世今生

**Q:** 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?

**A:**

Debugger 是一个命令行调试工具, 可以设置断点控制软件运行, 查看软件运行中信息, 修改软件执行流程。Emulator 指主要透过软件模拟硬件处理器的功能和指令系统的程序使计算机或者其他多媒体平台 (掌上电脑, 手机) 能够运行其他平台上的软件, 是一个载体, 就是虚拟机, 是一个独立的系统, 不会对电脑本身的系统造成影响, 可以借助它兼容不同的软件。

`gdb` 和我们在 `nemu` 中实现的功能好像差不多, 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, 可以随便查看变量当前的值。

### 4.3 i386 手册的学习

#### 4.3.1 问题: 通过目录定位关注的问题

**Q:** 假设你现在需要了解一个叫 `selector` 的概念, 请通过 i386 手册的目录确定你需要阅读手册中的哪些地方?

**A:** CHAPTER 5 MEMORY MANAGEMENT - 5.1 SEGMENT TRANSLATION - 5.1.3 Selectors



### 4.3.2 必答题

1. **Q:** 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?

**A:** 在手册第 34 页中 EFLAGS 寄存器结构处看到 CF 位属于状态标志位, 查看 2.3.4.1 状态标志位的介绍, 了解到 CF 被算术运算指令使用, 在执行算术指令之前, 有一些指令需要设置、清除和补充 CF。具体的定义在附录 C。在附录 C (P419) 中有对 CF 位的定义说明: CF 位是进位标志, 在高位有进位或借位时设置, 否则就清除该位。

- ModR/M 字节是什么?

**A:** 在手册第 38 页中提到大多数访问内存的数据操作指令都包含一个为操作数明确指定寻址方法的字节, 这个字节就是 ModR/M 字节。在第 241-243 页对 ModR/M 字节进行了具体介绍, 包括它包含的信息、格式等。

- mov 指令的具体格式是怎么样的?

**A:** 在第 347 页有对格式的说明, 格式是  $DEST \leftarrow SRC$ 。

2. **Q:** shell 命令完成 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? 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?

**A:**

完成 PA1 的内容之后,nemu/目录下的所有.c 和.h 文件总共有 3942 行, 使用命令

```
1 find . -name "*[.h|.c]" | xargs wc -l
```

得到结果。

和框架代码相比, 在 PA1 中编写了 416 行代码。

除去空行, nemu/目录下的所有.c 和.h 文件总共 3257 行, 使用命令

```
1 find . -name "*[.cpp|.h]" | xargs grep "^." | wc -l
```

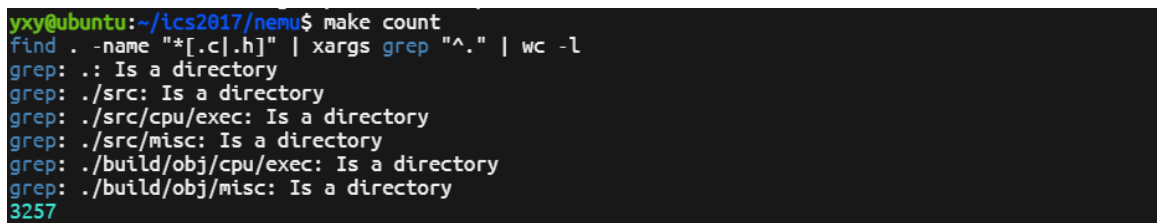
得到结果。

尝试在 makefile 文件中加入



```
1 count:
2 find . -name "*[.cpp|.h]" | xargs grep "^." | wc -l
```

在终端输入命令 `make count` 即可获知行数。



```
yxy@ubuntu:~/ics2017/nemu$ make count
find . -name "*[.c|.h]" | xargs grep "^." | wc -l
grep: ./src: Is a directory
grep: ./src/cpu/exec: Is a directory
grep: ./src/misc: Is a directory
grep: ./build/obj/cpu/exec: Is a directory
grep: ./build/obj/misc: Is a directory
3257
```

图 4.17: make count

3. Q: 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`?

A:

`-Wall` 的作用是让 `gcc` 产生尽可能多的警告信息, 在编译后显示所有警告。

`-Werror` 的作用是让 `gcc` 将所有的警告当成错误进行处理。

使用 `-Wall` 和 `-Werror` 是为了发现程序的错误, 便于 `Debug`, 找出错误点, 尽可能地避免程序运行出错, 优化程序。

## 5 实验结论与感想

这次实验对我来说是一次挑战。尤其是在处理词法分析和表达式求值时, 我花了很多时间来阅读指导手册, 理解实验的原理, 并用代码解决问题。特别是负号和解引用等特殊情况的处理, 让我花费了很长的时间去思考和调试代码。但在完成此次 PA1 之后, 我能明显的感觉到我对计算机系统认知上的提升。但此次实验, 也有许多我需要进一步改进的地方, 比如异常处理部分, 尽管我尽力考虑了各种情况, 但在一个真实的系统中, 异常处理需要更加周密, 而且在本次实验中我处理异常的方式就是简单的使用了 `assert(0)`, 这在之后的工作中都还需要进一步的完善, 以确保系统的稳定性和可靠性。再者对于我实现的系统的可靠性, 我只通过手动验证来确保输出的正确性, 但不能保证没有我没考虑到的情况发生, 这也许提示我应当尝试更多的试例, 或者采用自动化测试来检查程序在不同输入情况下的行为。另外, 我还会考虑使用调试工具来帮助捕获和解决潜在的错误。

总的来说, 这次实验对我来说是一个很好的学习机会。我相信通过克服这些挑战, 我将会获得更多的经验和收获, 并不断提升自己的解决问题和设计系统的能力。