



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: 孟启轩

学号: 2212452

专业: 计算机科学与技术

- 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
 - 一、实验目的
 - 二、实验重要内容
 - (一) NEMU的执行流程
 - (二) 究竟要执行多久
 - (三) 谁来指示程序的结束
 - (四) 为什么要使用 static?
 - (五) 一点也不能长?
 - (六) 随心所欲的断点
 - (七) NEMU的前世今生
 - 三、实验方法与结果
 - (一) 阶段一
 - 1、实现正确的寄存器结构体
 - 2、基础设施: 简易调试器
 - (1) 单步执行si
 - (2) 打印寄存器info
 - (3) 扫描内存x
 - (二) 阶段二
 - 1、词法分析
 - 2、表达式求值
 - (1) 实现检查括号匹配的函数 `check_parentheses()`
 - (2) 操作符优先级及寻找主操作符
 - (3) 递归求值

- (4) 表达式求值
- (5) 表达式求值命令p
- (6) 完善扫描内存命令x
- (三) 阶段三
 - 实现监视点及相关的监视点管理功能
 - 实现监视点让程序暂停执行的效果(断点)
- 四、遇到的BUG及解决思路
- 五、必答题
- 六、总结与感悟

一、实验目的

理解框架代码，初步了解程序在计算机上运行的基本原理，并学习使用 GDB 调试工具。还需要在 monitor 中实现一个类似 GDB 的简易调试器。

- (1) 实现单步执行, 打印寄存器, 扫描内存。
- (2) 实现实现算术表达式的词法分析, 实现算术表达式的递归求值, 实现带有负数的算术表达式的求值。
- (3) 实现监视点池的管理。

二、实验重要内容

(一) NEMU的执行流程

初始化阶段通过 `init_monitor()` 验证寄存器并加载镜像到内存 `0x100000` , `restart()` 设置初始寄存器状态 (`%eip=0x100000`) 后进入用户界面主循环; 用户输入 `c` 后, `cpu_exec()` 启动指令执行循环, 通过 `exec_wrapper()` 逐条解析并执行指令 (利用 `opcode_table` 查找实现) , 记录日志并更新 `%eip` ; 程序通过触发非x86指令 `0xd6` (`nemu_trap`) 正常终止, 输出终止信息后返回交互界面; NEMU简化了BIOS流程, 直接跳过硬件初始化, 通过 `load_img()` 注入程序镜像至指定内存地址。

(二) 究竟要执行多久

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?

在 `nemu/src/monitor/cpu-exec.c` 文件中, `cpu_exec()` 函数模拟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。

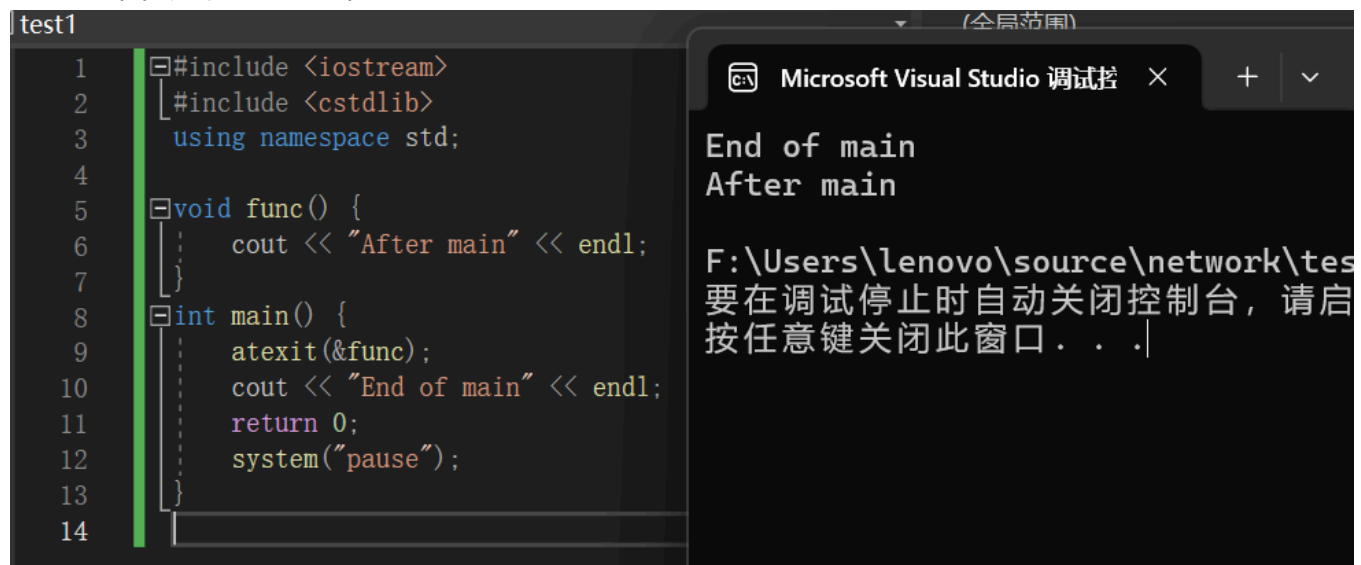
传入的参数为 `uint64_t` 类型, 即无符号长整型 (64位)。当我们传入-1时, 实际上是将其解释为 `uint64_t` 类型的最大值 (即 $2^{64}-1$), 从而确保函数能够持续执行指令直到遇到 `nemu_trap`。

(三) 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. 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.

写个简单程序测试一下即可。



The image shows a screenshot of a C++ program in a code editor and its debug output in the Visual Studio debugger. The code defines a function `func()` and a `main()` function. `main()` calls `func()`, prints "End of main", and then calls `system("pause")`. The debug output shows the program execution flow: "End of main" is printed, and then the program continues to execute after `main()` returns, demonstrating that the program does not end immediately at the `main()` return statement.

```
test1
1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4
5  void func() {
6      cout << "After main" << endl;
7  }
8  int main() {
9      atexit(&func);
10     cout << "End of main" << endl;
11     return 0;
12     system("pause");
13 }
14
```

Microsoft Visual Studio 调试控制 × + ▾

End of main
After main

F:\Users\lenovo\source\network\tes
要在调试停止时自动关闭控制台, 请启
按任意键关闭此窗口. . .|

通过测试我们可以发现, 程序并不在 `main()` 函数的返回处立即结束。 `exit()` 函数用于在程序运行的任意时刻结束程序, 其函数原型为 `void exit(int state)`。参数 `state` 是返回给操作系统的状态码, 返回 0 表示程序正常结束, 而非 0 表示程序非正常结束。 `main` 函数结束时会隐式调用 `exit()` 函数。当 `exit()` 函数被调用时, 它会首先执行由 `atexit()` 函数登记的所有函数, 然后刷新所有输出流、关闭所有打开的文件流, 并且删除通过标准 I/O 函数 `tmpfile()` 创建的临时文件。

`atexit()` 函数用于注册程序终止时需要执行的函数, 其函数原型为 `int atexit(void (*function)(void))`。由于程序退出的方式多种多样, 例如 `main()` 函数运行结束、在程序的某个地方调用 `exit()` 函数结束程序, 或者用户通过 `Ctrl+C` 或 `Ctrl+Break` 操作终止程序, 因此需要一种与退出方式无关的方法来进行程序退出时的必要处理。 `atexit()` 函数提供了一种注册程序正常终止时要被调用的函数的方式, 确保在程序退出时能够进行必要的清理和资源释放操作。

(四) 为什么要使用 static?

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访问, 从而保护了数据的安全性。这里使用 `static` 表示 `wp_pool` 是一个静态全局变量, 目的是为了 avoid 在程序运行过程中对其误操作。其他源文件若需对该变量进行操作, 必须通过封装的函数来实现。

(五) 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?

这是肯定的。在 `x86` 体系结构中, `int3` 指令的操作码为单字节 `0xCC`, 其固定长度为 1 字节, 这是处理器准确解码和执行的关键。然而在 `my-x86` 架构中, `int3` 指令的长度被修改为 2 字节, 这会导致原有调试器无法正确识别其长度, 进而破坏断点机制: 调试器可能错误读取指令数据, 导致断点失效或误执行后续指令。若要保证断点正常工作, 调试器需针对 `my-x86` 的特殊指令长度进行适配, 例如通过识别双字节 `int3` 格式并跳过完整指令长度; 反之, 若未进行适配, 则断点功能将因指令长度解析错误而失效。

(六) 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。

在 `x86` 指令体系中, 指令的非首字节通常代表操作数或修饰符而非指令起始位置, 因此若在 `GDB` 中将断点设置在指令的非首字节 (如通过 `b *0x080483f7+2` 设置在指令的第三个字节), 实际会干扰指令的操作数结构。当程序执行到该位置时, `CPU` 会误将操作数或修饰符当作指令执行, 可能导致程序无法停止调试而是触发不可预测的行为, 例如因执行无效地址的操作数而崩溃。

(七) 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?

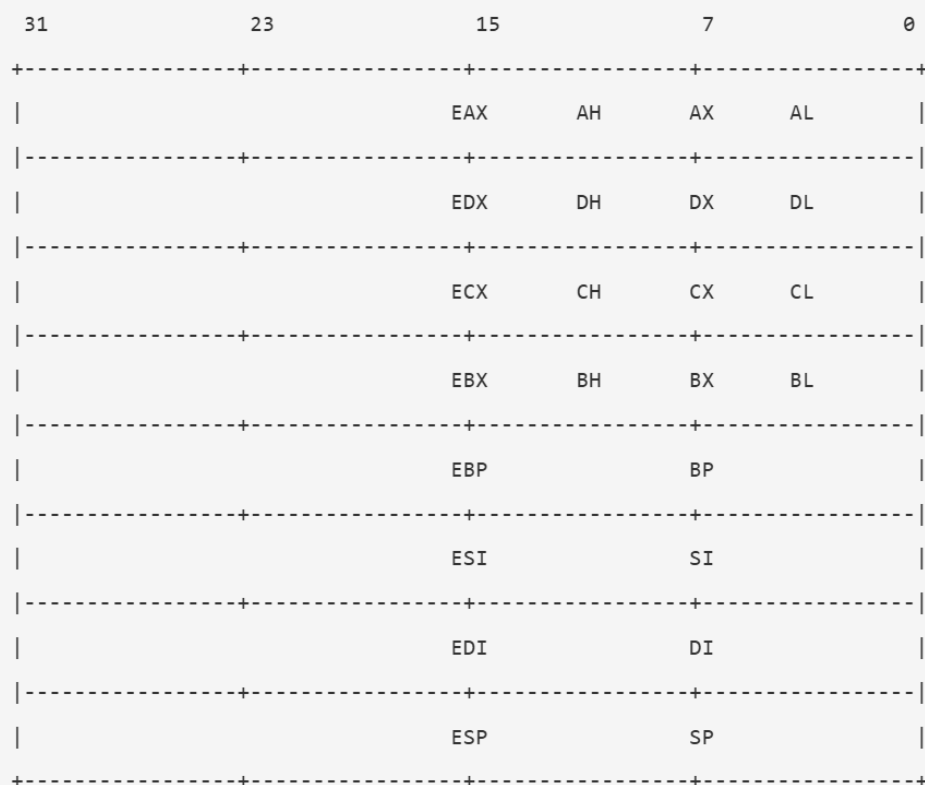
NEMU作为硬件模拟器, 通过软件模拟CPU等硬件运行客户程序, 其本质是将本地代码文件或内置指令作为输入, 通过 `cpu_exec()` 等函数模拟指令执行流程, 客户程序在NEMU的虚拟CPU中运行而非作为独立进程; 而GDB作为调试器, 依赖进程间通信机制, 通过 `ptrace` 系统调用将被调试程序作为子进程, 以读写其内存映像、捕获信号等方式实现调试功能, 例如断点通过向被调试进程内存写入 `int3` 指令实现, 两者的核心区别在于: NEMU通过模拟硬件直接执行代码, 而GDB通过操作系统级进程控制间接干预程序运行。

三、实验方法与结果

(一) 阶段一

1、实现正确的寄存器结构体

寄存器结构



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 . 这样的结构有时候在处理不同长度的数据时能提供一些便利.

结构体 CPU_state 的提示以及枚举常量的定义

```
ics2017 > nemu > include > cpu > C reg.h
```

```
1  #ifndef __REG_H__
2  #define __REG_H__
3
4  #include "common.h"
5
6  enum { R_EAX, R_ECX, R_EDX, R_EBX, R_ESP, R_EBP, R_ESI, R_EDI };
7  enum { R_AX, R_CX, R_DX, R_BX, R_SP, R_BP, R_SI, R_DI };
8  enum { R_AL, R_CL, R_DL, R_BL, R_AH, R_CH, R_DH, R_BH };
9
10 /* TODO: Re-organize the `CPU_state' structure to match the register
11  * encoding scheme in i386 instruction format. For example, if we
12  * access cpu.gpr[3]._16, we will get the `bx' register; if we access
13  * cpu.gpr[1]._8[1], we will get the 'ch' register. Hint: Use `union'.
14  * For more details about the register encoding scheme, see i386 manual.
15  */
16
```

reg_test()函数

```
24  assert(reg_b(R_AL) == (sample[R_EAX] & 0xff));
25  assert(reg_b(R_AH) == ((sample[R_EAX] >> 8) & 0xff));
26  assert(reg_b(R_BL) == (sample[R_EBX] & 0xff));
27  assert(reg_b(R_BH) == ((sample[R_EBX] >> 8) & 0xff));
28  assert(reg_b(R_CL) == (sample[R_ECX] & 0xff));
29  assert(reg_b(R_CH) == ((sample[R_ECX] >> 8) & 0xff));
30  assert(reg_b(R_DL) == (sample[R_EDX] & 0xff));
31  assert(reg_b(R_DH) == ((sample[R_EDX] >> 8) & 0xff));
32
33  assert(sample[R_EAX] == cpu.eax);
34  assert(sample[R_ECX] == cpu.ecx);
35  assert(sample[R_EDX] == cpu.edx);
36  assert(sample[R_EBX] == cpu.ebx);
37  assert(sample[R_ESP] == cpu.esp);
38  assert(sample[R_EBP] == cpu.ebp);
39  assert(sample[R_ESI] == cpu.esi);
40  assert(sample[R_EDI] == cpu.edi);
41
42  assert(eip_sample == cpu.eip);
43 }
```

根据提示和reg_test()函数要检查的内容，修改结构体CPU_state:

```

35 typedef struct {
36     union{
37         union {
38             uint32_t _32;
39             uint16_t _16;
40             uint8_t _8[2];
41         } gpr[8];
42
43         /* Do NOT change the order of the GPRs' definitions. */
44
45         /* In NEMU, rtlreg_t is exactly uint32_t. This makes RTL instructions
46          * in PA2 able to directly access these registers.
47          */
48         struct{
49             rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
50         };
51     };
52     vaddr_t eip;
53
54 } CPU_state;
55
56 extern CPU_state cpu;

```

由于8个通用寄存器对应的32位、16位和8位寄存器并不相互独立，因此将数组 `gpr[8]` 改为了匿名联合体（union）。这样可以通过该数组的某个元素访问对应的寄存器：

`_32` 可以访问到对应的32位寄存器（例如 `eax`）。

`_16` 可以访问到对应32位寄存器的低16位（例如 `ax`）。

`_8[0]` 可以访问到对应32位寄存器的低16位中的低8位（例如 `al`）。

`_8[1]` 可以访问到对应32位寄存器的低16位中的高8位（例如 `ah`）。

这种设计利用了联合体成员变量在同一时间只能使用其中一个、并且只能访问同一片内存的特性。在上图的 `reg_test` 函数的第24到31行，检查的就是这一机制。

外面的匿名联合体的作用是将 `gpr[8]` 数组与寄存器名相互对应，与上图中的枚举常量定义相对应。

此外，上图的 `reg_test()` 函数的 42 行表明 `eip` 的定义是要独立于这两个匿名 union 的。

测试结果


```

+ CC src/cpu/decode/decode.c
+ CC src/cpu/intr.c
+ CC src/cpu/reg.c
+ CC src/memory/memory.c
+ LD build/nemu
○ ics@e3b48c25a345:~/ics2017/nemu$ make run
./build/nemu -l ./build/nemu-log.txt
[src/monitor/monitor.c,47,load_default_img] No image is
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 15:10:25
For help, type "help"
(nemu) 

```

2、基础设施：简易调试器

我们需要实现的调试器功能如下：

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

当输入调试指令时，系统会调用 `cmd_table[i].handle(args)` 来执行相应的函数。因此，我们需要在 `cmd_table` 中添加命令与处理函数之间的对应关系。

```

246 static struct
247 {
248     char *name;
249     char *description;
250     int (*handler)(char *);
251 } cmd_table[] = {
252     {"help", "Display informations about all supported commands", cmd_help},
253     {"c", "Continue the execution of the program", cmd_c},
254     {"q", "Exit NEMU", cmd_q},
255     {"si", "args:[N];execute [N] instructions step by step", cmd_si},
256     {"info", "args:r/w;print infomation about registers or watchpoint", cmd_info},
257     {"p", "expr", cmd_p},
258     {"x", "x [N] [EXPR];scan the memory", cmd_x},
259     {"w", "set the watchpoint", cmd_w},
260     {"d", "delete the watchpoint", cmd_d},
261
262     /* TODO: Add more commands */
263
264 };

```

其中，handler 是一个返回值为 int 类型的函数指针，它接受一个 char* 类型的参数，并指向相应的命令处理函数。在 ui_mainloop() 函数中，系统通过这个 handler 参数调用用户输入命令所对应的处理函数。

(1) 单步执行si

单步执行指令si的实现

```

44 static int cmd_si(char *args)
45 {
46     int step;
47     if (args == NULL)
48         step = 1; // 默认执行1条指令
49     else
50         sscanf(args, "%d", &step); // 从参数字符串中解析步数
51     cpu_exec(step); // 执行step条指令
52     return 0;
53 }

```

- 用户输入si N：触发cmd_si函数，解析参数后调用cpu_exec(N)。
- 执行指令：cpu_exec循环执行N条指令，每条指令通过exec_wrapper处理。
- 返回控制权：执行完成后，返回到NEMU的命令行界面，等待用户输入下一个命令。

测试结果

```
For help, type "help"
(nemu) si
100000:  b8 34 12 00 00          movl $0x1234,%eax
(nemu) si 2
100005:  b9 27 00 10 00          movl $0x100027,%ecx
10000a:  89 01                   movl %eax,(%ecx)
(nemu) si 5
10000c:  66 c7 41 04 01 00       movw $0x1,0x4(%ecx)
100012:  bb 02 00 00 00          movl $0x2,%ebx
100017:  66 c7 84 99 00 e0 ff ff 01 00  movw $0x1,-0x2000(%ecx,%ebx,4)
100021:  b8 00 00 00 00          movl $0x0,%eax
nemu: HIT GOOD TRAP at eip = 0x00100026

100026:  d6                   nemu trap (eax = 0)
(nemu) q
ics@e3b48c25a345:~/ics2017/nemu$
```

(2) 打印寄存器info

打印寄存器info的实现

```

static int cmd_info(char *args)
{
    char *arg = strtok(NULL, " "); // 解析子命令（如"r"或"w"）
    if (args == NULL) {
        /* no argument given, show rules */
        printf("\ninfo\" must be followed by the name of an info command.\n");
        printf("List of info subcommands:\n\n");
        printf("info r -- List of integer registers and their contents\n");
        printf("info w -- Status of specified watchpoints (all watchpoints if no argument)\n");
    } else if (strcmp(arg, "r") == 0) { // 处理"info r"命令
        char *temp = strtok(NULL, " "); // 解析后续参数（指定的寄存器名）
        if (temp == NULL) { // 未指定寄存器名 → 打印所有寄存器
            /* 打印所有32位寄存器 */
            uint32_t i;
            for (i = 0; i < 8; i++) { // eax, ecx, edx, ebx, esp, ebp, esi, edi
                printf("%s\t0x%x\t%d\n", reg_name(i, 4), reg_l(i), reg_l(i));
            }
            printf("eip\t0x%x\t%d\n", cpu.eip, cpu.eip); // 打印指令指针eip
            /* 打印所有16位寄存器（如ax, cx等） */
            for (i = 0; i < 8; i++) {
                printf("%s\t0x%x\t%hd\n", reg_name(i, 2), reg_w(i), reg_w(i));
            }
            /* 打印所有8位寄存器（如al, cl等） */
            for (i = 0; i < 8; i++) {
                printf("%s\t0x%x\t%hhd\n", reg_name(i, 1), reg_b(i), reg_b(i));
            }
        } else { // 指定了寄存器名 → 逐个处理
            uint32_t i;
            uint32_t find = 0;
            while (temp != NULL) { // 处理多个参数（如"info r eax ecx"）
                find = 0;
                /* 检查32位寄存器名（如eax） */
                for (i = 0; i < 8; i++) {
                    if (strcmp(reg_name(i, 4), temp) == 0) { // reg_name(i,4)返回如"eax"
                        printf("%s\t0x%x\t%d\n", reg_name(i, 4), reg_l(i), reg_l(i));
                        find = 1;
                        break;
                    }
                }
                temp = strtok(NULL, " ");
            }
        }
    }
}

```

```

    if (find) { // 找到后继续处理下一个参数
        temp = strtok(NULL, " ");
        continue;
    }
    // 检查eip (单独处理)
    if (strcmp("eip", temp) == 0) {
        printf("eip\t0x%x\t%d\n", cpu.eip, cpu.eip);
        find = 1;
        temp = strtok(NULL, " ");
        continue;
    }
    // 检查16位寄存器名 (如ax)
    for (i = 0; i < 8; i++) {
        if (strcmp(reg_name(i, 2), temp) == 0) { // reg_name(i,2)返回如"ax"
            printf("%s\t0x%x\t%hd\n", reg_name(i, 2), reg_w(i), reg_w(i));
            find = 1;
            break;
        }
    }
    if (find) {
        temp = strtok(NULL, " ");
        continue;
    }
    // 检查8位寄存器名 (如al)
    for (i = 0; i < 8; i++) {
        if (strcmp(reg_name(i, 1), temp) == 0) { // reg_name(i,1)返回
            printf("%s\t0x%x\t%hhd\n", reg_name(i, 1), reg_b(i), reg_b(i));
            find = 1;
            break;
        }
    }
    if (!find) { // 未找到 → 提示错误
        printf("Invalid register `%s'\n", temp);
    }
    temp = strtok(NULL, " "); // 继续处理下一个参数
}

}

} else if (strcmp(arg, "w") == 0) { // 处理"info w"命令 (监视点)
    print_wp(); // 调用打印监视点的函数
}

```

```

    } else {
        printf("Undefined info command: \"%s\"\n", arg);
    }
    return 0;
}

```

- 通过strtok解析用户输入的多个参数，支持info r后跟多个寄存器名。
- 通过循环和reg_name函数，按位数（32/16/8位）分类打印所有寄存器。
- 对用户指定的寄存器名逐个匹配，覆盖32/16/8位寄。

测试结果

```

For help, type "help"
(nemu) info r
eax    0x39ab0fed    967512045
ecx    0x72e93e62    1927888482
edx    0x77e17b57    2011265879
ebx    0x4898aa2     76122786
esp    0x3ea3a007    1050910727
ebp    0x303d8021    809336865
esi    0x4a1a8aae    1243253422
edi    0x7946e2a5    2034688677
eip    0x100000      1048576
ax     0xfed    4077
cx     0x3e62   15970
dx     0x7b57   31575
bx     0x8aa2   -30046
sp     0xa007   -24569
bp     0x8021   -32735
si     0x8aae   -30034
di     0xe2a5   -7515
al     0xed     -19
cl     0x62     98
dl     0x57     87
bl     0xa2     -94
ah     0xf      15
ch     0x3e     62
dh     0x7b     123
bh     0x8a     -118
(nemu)

```

(3) 扫描内存x

扫描内存x的实现

```

static int cmd_x(char *args)
{
    char *arg = strtok(NULL, " ");
    uint32_t n = 0;
    if (arg == NULL)
    {
        /* no argument given */
        printf("Argument required (starting display address).\n");
    } else {
        char *temp1 = strtok(NULL, " ");
        char *temp2 = strtok(NULL, " ");
        uint32_t start_addr = 0;
        uint32_t mem_data = 0;
        bool success = true;
        if (temp2 == NULL) {
            // 解析参数: 假设格式为 "n address" (如 "4 0x1000")
            sscanf(arg, "%u", &n); // 解析n (行数)
            start_addr = expr(temp1, &success); // 解析起始地址表达式
            if (!success) {
                printf("Illegal expression.\n");
                return 0;
            }
            // 打印内存内容
            uint32_t i, j;
            for (i = 0; i < n; i++) {
                printf("0x%x:\t", start_addr); // 当前行的起始地址
                for (j = 0; j < 4; j++) { // 每行显示4个字节
                    mem_data = vaddr_read(start_addr, 1); // 读取1字节
                    printf("0x%02x\t", mem_data); // 以十六进制显示 (补零)
                    start_addr++; // 地址递增
                }
                printf("\n");
            }
        } else {
            printf("Syntax error: Too much arguments.\n");
        }
    }
    return 0;
}

```

上面有部分代码涉及阶段二内容，所以对参数解析做了修改。

- 支持输入x 4 addr的命令，显示从addr开始的4行内存（每行4字节）。
- 通过vaddr_read逐字节读取内存，并按十六进制格式显示。
- 每行固定显示4个字节，地址逐行递增。

测试结果

```
For help, type "help"
(nemu) x 10 0x100000
[src/monitor/debug/expr.c,104,make_token] match rules[1] = "0x[0-9A-Fa-f][0-9A-Fa-f]*" at position 0 with len 8: 0x100000
RESULT=1048576
0x100000:  0xb8  0x34  0x12  0x00
0x100004:  0x00  0xb9  0x27  0x00
0x100008:  0x10  0x00  0x89  0x01
0x10000c:  0x66  0xc7  0x41  0x04
0x100010:  0x01  0x00  0xbb  0x02
0x100014:  0x00  0x00  0x00  0x66
0x100018:  0xc7  0x84  0x99  0x00
0x10001c:  0xe0  0xff  0xff  0x01
0x100020:  0x00  0xb8  0x00  0x00
0x100024:  0x00  0x00  0xd6  0x00
(nemu) x 10 $eip
[src/monitor/debug/expr.c,104,make_token] match rules[3] = "\$(eax|ecx|edx|ebx|esp|ebp|esi|edi|eip|ax|cx|dx|bx|sp|bp|si|di)" at position 0 with len 4: $eip
RESULT=1048576
0x100000:  0xb8  0x34  0x12  0x00
0x100004:  0x00  0xb9  0x27  0x00
0x100008:  0x10  0x00  0x89  0x01
0x10000c:  0x66  0xc7  0x41  0x04
0x100010:  0x01  0x00  0xbb  0x02
0x100014:  0x00  0x00  0x00  0x66
0x100018:  0xc7  0x84  0x99  0x00
0x10001c:  0xe0  0xff  0xff  0x01
0x100020:  0x00  0xb8  0x00  0x00
0x100024:  0x00  0x00  0xd6  0x00
(nemu)
```

(二) 阶段二

1、词法分析

词法分析的作用是将输入的字符流识别成一个个单词 (token)。我们使用正则表达式来匹配子串。

定义token和识别规则


```

9  enum
10  { // start from 256 to avoid ascii
11      TK_NOTYPE = 256,
12      TK_HEX,
13      TK_DEC,
14      TK_REG,
15      TK_EQ,
16      TK_NEQ,
17      TK_AND,
18      TK_OR,
19      TK_NEG, // negative number
20      TK_POI, // use for get the content of pointer
21
22      /* TODO: Add more token types */
23
24  };

```

```

28  int token_type;
29  } rules[] = {
30
31      /* TODO: Add more rules.
32       * Pay attention to the precedence level of different rules.
33       */
34      {" +", TK_NOTYPE}, // 空
35      {"0x[0-9A-Fa-f][0-9A-Fa-f]*", TK_HEX},
36      {"0|[1-9][0-9]*", TK_DEC},
37      {"\\$(eax|ecx|edx|ebx|esp|ebp|esi|edi|eip|ax|cx|dx|bx|sp|bp|si|di|al|cl|dl|bl|ah|ch|dh|bh)", TK_REG},
38
39      {"\\+", '+'},
40      {"\\-", '-'},
41      {"\\*", '*'},
42      {"\\/", '/'},
43
44      {"\\(", '('},
45      {"\\)", ')'},
46
47      {"==", TK_EQ},
48      {"!=", TK_NEQ},
49
50      {"&&", TK_AND},
51      {"\\|\\|\\|", TK_OR},
52      {"!", '!'},
53  };

```

```

static bool make_token(char *e)
{
    int position = 0;
    int i;
    regmatch_t pmatch;
    nr_token = 0;
    while (e[position] != '\0')
    {
        /* Try all rules one by one. */
        for (i = 0; i < NR_REGEX; i++)
        {
            if (regexexec(&re[i], e + position, 1, &pmatch, 0) == 0 && pmatch.rm_so == 0) //完全匹配
            {
                //记录匹配结果
                char *substr_start = e + position;
                int substr_len = pmatch.rm_eo;
                Log("match rules[%d] = \"%s\" at position %d with len %d: %.*s",
                    i, rules[i].regex, position, substr_len, substr_len, substr_start);
                position += substr_len;

                /* 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.
                 */
                if (rules[i].token_type == TK_NOTYPE) //跳过空格
                    break;
                if (substr_len > 31) // avoid overflow
                    assert(0);
                strncpy(tokens[nr_token].str, substr_start, substr_len);
                tokens[nr_token].type = rules[i].token_type;
                nr_token = nr_token + 1;
                break;
            }
        }
        if (i == NR_REGEX)
        {
            printf("no match at position %d\n%s\n%*.s^\n", position, e, position, "");
            return false;
        }
    }
}

```

```
    return true;
}
```

- 逐字符遍历输入字符串，在每一个起始位置按规则数组顺序尝试所有预编译的正则表达式，按完全匹配成功的规则记录匹配结果。跳过空格。
- 移动扫描位置至当前匹配末尾，循环直到字符串结束

2、表达式求值

(1) 实现检查括号匹配的函数 check_parentheses()

```
137 bool check_parentheses(int p, int q)
138 {
139     if ((tokens[p].str[0] == '(' && (tokens[q].str[0] == ')')){
140         int count = 0;
141         for (int i = p; i < q; i++) {
142             if (tokens[i].str[0] == '(')
143                 count = count + 1;
144             if (tokens[i].str[0] == ')')
145                 count = count - 1;
146             if (count == 0){
147                 return false;
148             }
149         }
150         count = count - 1;
151         if (count != 0){
152             printf("Bad parentheses\n");
153             assert(0);
154         }
155         return true;
156     }
157     else{
158         return false;
159     }
160 }
```

我在这里仅用计数器实现了简单的检查，后面计算求值时进行了完善。

(2) 操作符优先级及寻找主操作符

代码如下

```

// 寻找主操作符 (dominant operator)
int op = 0;
int op_type = 0;
bool left = false;
int curr_level = 100; // 当前找到的最低优先级 (数值越大优先级越高)
for (int i = p; i <= q; i++){// 遍历所有token, 跳过括号内的内容
    if (tokens[i].str[0] == ')'){
        left = false;
        continue;
    }
    if (left)
        continue;
    if (tokens[i].str[0] == '('){
        left = true;
        continue;
    }

    switch (tokens[i].type){// 根据操作符类型更新主操作符
    case TK_OR:
        if (curr_level >= 1){
            curr_level = 1;
            op = i;
            op_type = TK_OR;
        }
        break;
    case TK_AND:
        if (curr_level >= 2){
            curr_level = 2;
            op = i;
            op_type = TK_AND;
        }
        break;
    case TK_EQ:
        if (curr_level >= 3){
            curr_level = 3;
            op = i;
            op_type = TK_EQ;
        }
        break;
    }
}

```

```

.....

case '<':
    if (curr_level >= 4){
        curr_level = 4;
        op = i;
        op_type = '<';
    }
    break;
.....

case '+':
    if (curr_level >= 5){// 左结合操作符，允许更新为更右侧的同级操作符
        curr_level = 5;
        op = i;
        op_type = '+';
    }
    break;
.....

case '*':
    if (curr_level >= 6){
        curr_level = 6;
        op = i;
        op_type = '*';
    }
    break;
.....

case '!':
    if (curr_level > 8){// 右结合操作符，使用 > 只保留第一个出现的
        curr_level = 8;
        op = i;
        op_type = '!';
    }
    break;
case TK_NEG:
    if (curr_level > 9){// 右结合操作符，使用 > 只保留第一个出现的
        curr_level = 9;

```

```

        op = i;
        op_type = TK_NEG;
    }
    break;

    .....
    default:
        break;
}
}

```

通过遍历 token，根据预设的优先级数值选择最低优先级的操作符作为主操作符。

从低到高优先级：

- 1: 逻辑或(TK_OR)
- 2: 逻辑与(TK_AND)
- 3: 等于/不等于(TK_EQ/TK_NEQ)
- 4: 比较(< >)
- 5: 加减
- 6: 乘除
- 7: 单目运算符(!, 负号, 解引用)

各操作符处理逻辑：

1. 使用 \geq 比较实现左结合性（选择最后出现的操作符）
2. 数值越大表示优先级越高
3. 优先级层级：
 - 逻辑或(1) < 逻辑与(2) < 等于/不等(3) < 比较(4)
 - 加减(5) < 乘除(6) < 单目(7)

(3) 递归求值

代码如下

```

uint32_t eval(int p, int q)
{
    if (p > q) return 0;           // 空表达式返回0
    else if (p == q) {             // 单个token的情况
        // 处理数字/寄存器等基本值
        uint32_t res;
        if (tokens[p].type == TK_HEX)
            sscanf(tokens[p].str, "%x", &res);
        else if (tokens[p].type == TK_DEC)
            sscanf(tokens[p].str, "%d", &res);
        else if (tokens[p].type == TK_REG){
            char tmp[3] = {tokens[p].str[1], tokens[p].str[2], tokens[p].str[3]};
            for (int i = 0; i < 8; i++){
                if (!strcmp(tmp, regsl[i])){
                    return cpu.gpr[i]._32;
                }
            }
            for (int i = 0; i < 8; i++){
                if (!strcmp(tmp, regsw[i])){
                    return cpu.gpr[i]._16;
                }
            }
            for (int i = 0; i < 8; i++){
                if (!strcmp(tmp, regsb[i])){
                    return cpu.gpr[i % 4]._8[i / 4];
                }
            }
            char teip[3] = "eip";
            if (strcmp(tmp, teip))
                return cpu.eip;
        }
        else
            assert(0);
        return res;
    }
    else if (check_parentheses(p, q) == true) // 完全被括号包裹的情况
    {
        return eval(p + 1, q - 1); // 递归处理括号内部
    }
    else { // 寻找主操作符 (dominant operator)
        .....
    }
}

```

```

// 递归求值并执行运算
uint32_t val1 = eval(p, op - 1); // 左子表达式
uint32_t val2 = eval(op + 1, q); // 右子表达式
switch (op_type){
case TK_OR:
    return val1 || val2;
case TK_AND:
    return val1 && val2;
...
...
...
default:
    assert(0);
}
}
}

```

- 对于基本情况，直接返回数字/寄存器值或处理空表达式
- 若表达式被合法括号包裹，则递归处理内部内容
- 通过优先级（数值越大优先级越高）和结合性（左结合选最后出现的同级操作符，右结合选首个）确定分裂点
- 将表达式拆分为左右子表达式分别求值，最终合并运算结果
- 特别处理单目运算符（如!/-号）时，仅计算右子树，而双目运算符（如+/*）同时计算左右子树。

(4) 表达式求值

```

377 uint32_t expr(char *e, bool *success)
378 {
379     if (!make_token(e))
380     {
381         *success = false;
382         return 0;
383     }
384     // 将减号(-)转换为负号的情况：出现在表达式开头，或前导token是操作符/左括号
385     if (nr_token != 1)
386     for (int i = 0; i < nr_token; i++)
387     if (tokens[i].type == '-' && (i == 0 || tokens[i - 1].type == '(' || tokens[i - 1].type == TK_NEG || tokens[i - 1].type == '-' || tokens[i - 1].type == TK_POI))
388     tokens[i].type = TK_NEG;
389     // 将星号(*)转换为解引用的情况：出现在表达式开头，或前导token不是数值/右括号
390     for (int i = 0; i < nr_token; i++)
391     if (tokens[i].type == '*' && (i == 0 || (tokens[i - 1].type != TK_DEC && tokens[i - 1].type != TK_HEX && tokens[i - 1].type != '=')))
392     tokens[i].type = TK_POI;
393
394     /* TODO: Insert codes to evaluate the expression. */
395     printf("RESULT=%d\n", eval(0, nr_token - 1));
396
397     return eval(0, nr_token - 1);
398 }
399

```


进行表达式求值，并处理负号和解引用的情况。

(5) 表达式求值命令p

实现处理函数 cmd_p()

```
179 static int cmd_p(char *args)
180 {
181     bool *success = false;
182     // char *arg = strtok(NULL, " ");
183     if (args == NULL)
184     {
185         printf("Lack of parameter!\n");
186         return 0;
187     }
188     uint32_t res = expr(args, success);
189     printf("the value of expr is %x\n", res);
190     return 0;
191 }
```

直接调用 expr() 函数并打印求值结果即可。

测试结果

```

For help, type "help"
(nemu) p 1+2*2
[src/monitor/debug/expr.c,104,make_token] match rules[2] = "0|[1-9][0-9]*" at position 0 with len 1: 1
[src/monitor/debug/expr.c,104,make_token] match rules[4] = "+" at position 1 with len 1: +
[src/monitor/debug/expr.c,104,make_token] match rules[2] = "0|[1-9][0-9]*" at position 2 with len 1: 2
[src/monitor/debug/expr.c,104,make_token] match rules[6] = "*" at position 3 with len 1: *
[src/monitor/debug/expr.c,104,make_token] match rules[2] = "0|[1-9][0-9]*" at position 4 with len 1: 2
RESULT=5
the value of expr is 5
(nemu) p -(1+1)
[src/monitor/debug/expr.c,104,make_token] match rules[5] = "-" at position 0 with len 1: -
[src/monitor/debug/expr.c,104,make_token] match rules[8] = "(" at position 1 with len 1: (
[src/monitor/debug/expr.c,104,make_token] match rules[2] = "0|[1-9][0-9]*" at position 2 with len 1: 1
[src/monitor/debug/expr.c,104,make_token] match rules[4] = "+" at position 3 with len 1: +
[src/monitor/debug/expr.c,104,make_token] match rules[2] = "0|[1-9][0-9]*" at position 4 with len 1: 1
[src/monitor/debug/expr.c,104,make_token] match rules[9] = ")" at position 5 with len 1: )
RESULT=-2
the value of expr is ffffffff
(nemu) p 1-(1+1)
[src/monitor/debug/expr.c,104,make_token] match rules[2] = "0|[1-9][0-9]*" at position 0 with len 1: 1
[src/monitor/debug/expr.c,104,make_token] match rules[5] = "-" at position 1 with len 1: -
[src/monitor/debug/expr.c,104,make_token] match rules[8] = "(" at position 2 with len 1: (
[src/monitor/debug/expr.c,104,make_token] match rules[2] = "0|[1-9][0-9]*" at position 3 with len 1: 1
[src/monitor/debug/expr.c,104,make_token] match rules[4] = "+" at position 4 with len 1: +
[src/monitor/debug/expr.c,104,make_token] match rules[2] = "0|[1-9][0-9]*" at position 5 with len 1: 1
[src/monitor/debug/expr.c,104,make_token] match rules[9] = ")" at position 6 with len 1: )
RESULT=-1
the value of expr is ffffffff
(nemu) p 6/2
[src/monitor/debug/expr.c,104,make_token] match rules[2] = "0|[1-9][0-9]*" at position 0 with len 1: 6
[src/monitor/debug/expr.c,104,make_token] match rules[7] = "/" at position 1 with len 1: /
[src/monitor/debug/expr.c,104,make_token] match rules[2] = "0|[1-9][0-9]*" at position 2 with len 1: 2
RESULT=3
the value of expr is 3
(nemu) 

```

```

For help, type "help"
(nemu) p 0&&1
[src/monitor/debug/expr.c,106,make_token] match rules[2] = "0|[1-9][0-9]*" at position 0 with len 1: 0
[src/monitor/debug/expr.c,106,make_token] match rules[12] = "&&" at position 1 with len 2: &&
[src/monitor/debug/expr.c,106,make_token] match rules[2] = "0|[1-9][0-9]*" at position 3 with len 1: 1
RESULT=0
the value of expr is 0
(nemu) p ((5*6+1)-(4*3))
[src/monitor/debug/expr.c,106,make_token] match rules[8] = "(" at position 0 with len 1: (
[src/monitor/debug/expr.c,106,make_token] match rules[8] = "(" at position 1 with len 1: (
[src/monitor/debug/expr.c,106,make_token] match rules[2] = "0|[1-9][0-9]*" at position 2 with len 1: 5
[src/monitor/debug/expr.c,106,make_token] match rules[6] = "*" at position 3 with len 1: *
[src/monitor/debug/expr.c,106,make_token] match rules[2] = "0|[1-9][0-9]*" at position 4 with len 1: 6
[src/monitor/debug/expr.c,106,make_token] match rules[4] = "+" at position 5 with len 1: +
[src/monitor/debug/expr.c,106,make_token] match rules[2] = "0|[1-9][0-9]*" at position 6 with len 1: 1
[src/monitor/debug/expr.c,106,make_token] match rules[9] = ")" at position 7 with len 1: )
[src/monitor/debug/expr.c,106,make_token] match rules[5] = "-" at position 8 with len 1: -
[src/monitor/debug/expr.c,106,make_token] match rules[8] = "(" at position 9 with len 1: (
[src/monitor/debug/expr.c,106,make_token] match rules[2] = "0|[1-9][0-9]*" at position 10 with len 1: 4
[src/monitor/debug/expr.c,106,make_token] match rules[6] = "*" at position 11 with len 1: *
[src/monitor/debug/expr.c,106,make_token] match rules[2] = "0|[1-9][0-9]*" at position 12 with len 1: 3
[src/monitor/debug/expr.c,106,make_token] match rules[9] = ")" at position 13 with len 1: )
[src/monitor/debug/expr.c,106,make_token] match rules[9] = ")" at position 14 with len 1: )
RESULT=19

```

(6) 完善扫描内存命令x

扫描内存与表达式求值进行结合。实现扫描的addr是一个表达式的情况。
代码在阶段一的x命令已给出。

完善前

```
[src/monitor/monitor.c,30,welcome] Build time: 15
For help, type "help"
(nemu) x 10 0x100000+0x000001
10
0x100000+0x000001
0x100000:      0xb8      0x34      0x12      0x00
0x100004:      0x00      0xb9      0x27      0x00
0x100008:      0x10      0x00      0x89      0x01
0x10000c:      0x66      0xc7      0x41      0x04
0x100010:      0x01      0x00      0xbb      0x02
0x100014:      0x00      0x00      0x00      0x66
0x100018:      0xc7      0x84      0x99      0x00
0x10001c:      0xe0      0xff      0xff      0x01
0x100020:      0x00      0xb8      0x00      0x00
0x100024:      0x00      0x00      0xd6      0x00
(nemu)
```

完善后

```
For help, type "help"
(nemu) x 10 0x100000+0x000001
[src/monitor/debug/expr.c,104,make_token] match rules[1] = "0x[0-9A-Fa-f][0-9A-Fa-f]*" at position 0 with len 8: 0x100000
[src/monitor/debug/expr.c,104,make_token] match rules[4] = "\\+" at position 8 with len 1: +
[src/monitor/debug/expr.c,104,make_token] match rules[1] = "0x[0-9A-Fa-f][0-9A-Fa-f]*" at position 9 with len 8: 0x000001
RESULT=1048577
0x100001:      0x34      0x12      0x00      0x00
0x100005:      0xb9      0x27      0x00      0x10
0x100009:      0x00      0x89      0x01      0x66
0x10000d:      0xc7      0x41      0x04      0x01
0x100011:      0x00      0xbb      0x02      0x00
0x100015:      0x00      0x00      0x66      0xc7
0x100019:      0x84      0x99      0x00      0xe0
0x10001d:      0xff      0xff      0x01      0x00
0x100021:      0xb8      0x00      0x00      0x00
0x100025:      0x00      0xd6      0x00      0x00
(nemu)
```

(三) 阶段三

实现监视点及相关的监视点管理功能

首先，我们需要完善监视点的结构体，新增监视点表达式、监视点的值、监视点命中次数三个成员变量，并在其中新增 new_wp(), free_wp(), print_wp(), check_wp() 四个成员变量。

```
6  typedef struct watchpoint {
7      int NO;
8      struct watchpoint *next;
9
10     /* TODO: Add more members if necessary */
11     int oldValue;
12     char wp[32];
13     int hitNum;
14 } WP;
15 bool new_wp(char *arg);
16 bool free_wp(int num);
17 void print_wp();
18 bool check_wp();
19
20 #endif
21
```

在初始化函数中，我们对新增的成员变量进行初始化，并且新增变量记录下一个监视点的编号以及监视点指针。

```
11 void init_wp_pool() {
12     int i;
13     for (i = 0; i < NR_WP; i++) {
14         wp_pool[i].NO = i;           // 初始化监视点编号 (0~31)
15         wp_pool[i].next = &wp_pool[i + 1]; // 链表化空闲池 (每个元素指向下一个)
16         wp_pool[i].oldValue = 0;      // 初始旧值为0
17         wp_pool[i].hitNum = 0;        // 初始命中次数为0
18     }
19     wp_pool[NR_WP - 1].next = NULL; // 最后一个元素的next置空
20
21     head = NULL; // 初始无活动监视点
22     free_ = wp_pool; // 空闲池初始化为整个对象池
23     used_next = 0; // 下一个监视点编号从0开始
24 }
```

然后我们便可实现监视点的新增与释放函数，new_wp() 以及 free_wp() 函数。
在 new_wp() 函数中，首先需要判断空闲链表中是否有剩余，有剩余的话将头部节点取用到监视点链表中，维护空闲链表头，对新监视点的信息进行初始化（根据所给的参数求值），当监视点列表为空时，将新监视点设为头，不为空时，则将其放到链表尾部。

```
26  /* 新建监视点 */
27  bool new_wp(char *args) {
28      if (free_ == NULL) {
29          assert(0); // 空闲池已耗尽，触发断言
30      }
31      WP *result = free_;
32      free_ = free_>next; // 从空闲池取出第一个元素
33      result->NO = used_next; // 分配唯一编号
34      used_next++;          // 编号递增
35      result->next = NULL;   // 新监视点初始无后续节点
36      strcpy(result->wp, args); // 复制表达式字符串（如"eax+ebx"）
37
38      bool success;
39      result->oldValue = expr(result->wp, &success); // 计算初始值
40      // if (!success) { ... } // 需处理表达式错误
41
42      // 将新监视点添加到活动链表末尾
43      if (head == NULL) {
44          head = result; // 空链表直接设为头节点
45      } else {
46          wptemp = head;
47          while (wptemp->next != NULL) wptemp = wptemp->next; // 遍历到尾节点
48          wptemp->next = result; // 尾部追加
49      }
50      printf("success: set watchpoint %d ,oldValue=%d\n", result->NO, result->oldValue);
51      return true;
52  }
```

在 free_wp 函数中，首先需要判断监视点链表中是否有剩余，有剩余的话将头部节点取用到监视点，将所给的编号与监视点链表中所有节点编号进行对比，若为头节点，则更新维护监视点链表头，若为中间节点，则维护链表连接。如果找到对应的监视点，则将其设为空闲链表的头部。

```

54  /* 删除指定编号的监视点 */
55  bool free_wp(int num) {
56      WP *thewp = NULL; // 待删除的监视点
57      if (head == NULL) {
58          printf("no watchpoint now\n");
59          return false;
60      }
61      // 检查头节点是否匹配
62      if (head->NO == num) {
63          thewp = head;
64          head = head->next; // 移除头节点
65      } else {
66          wptemp = head;
67          // 遍历链表寻找目标节点
68          while (wptemp != NULL && wptemp->next != NULL) {
69              if (wptemp->next->NO == num) {
70                  thewp = wptemp->next;
71                  wptemp->next = wptemp->next->next; // 跳过目标节点
72                  break;
73              }
74              wptemp = wptemp->next;
75          }
76      }
77      // 将释放的节点返回空闲池
78      if (thewp != NULL) {
79          thewp->next = free_;
80          free_ = thewp; // 空闲池头部指向新释放的节点
81          return true;
82      }
83      return false;
84  }

```

对于 print_wp() 函数，直接从头开始遍历 head 链表并打印每一个监视点的信息即可。

```

86  /* 打印所有监视点 */
87  void print_wp() {
88      if (head == NULL) {
89          printf("no watchpoint now\n");
90          return;
91      }
92      printf("watchpoint:\n");
93      printf("NO.   expr   hitTimes\n");
94      wptemp = head;
95      while (wptemp != NULL) {
96          printf("%d    %s    %d\n", wptemp->NO, wptemp->wp, wptemp->hitNum);
97          wptemp = wptemp->next;
98      }
99  }

```

对于 check_wp() 函数，从头开始遍历 head 链表，逐一检查每个监视点对应的表达式的新值和旧值是否相等，只要出现了一个不相等的情况就返回，表示命中。

```

102  /* 检查所有监视点是否命中 */
103  bool check_wp() {
104      bool success;
105      int result;
106      if (head == NULL) return true; // 无监视点时直接返回成功
107      wptemp = head;
108      while (wptemp != NULL) {
109          result = expr(wptemp->wp, &success); // 重新计算表达式值
110          if (result != wptemp->oldValue) { // 值变化
111              wptemp->hitNum += 1;
112              printf("Hardware watchpoint %d:%s\n", wptemp->NO, wptemp->wp);
113              printf("Old value:%d\nNew value:%d\n\n", wptemp->oldValue, result);
114              wptemp->oldValue = result; // 更新旧值
115              return false; // 有命中时返回false (可能触发中断)
116          }
117          wptemp = wptemp->next;
118      }
119      return true; // 所有监视点未命中
120  }
121

```

最后，我们需要实现命令 info w 和 d。对于 info w 命令，需要调用 print_wp() 函数打印所有监视点的信息。（代码在前面已经给出）

对于命令 d，调用 free_wp() 函数释放指定的监视点即可。

```

199  static int cmd_d(char *args)
200  {
201      int num = 0;
202      int nRet = sscanf(args, "%d", &num);
203      if (nRet <= 0)
204      {
205          printf("args error in cmd_si\n");
206          return 0;
207      }
208      int r = free_wp(num);
209      if (r == false)
210      {
211          printf("error: no watchpoint %d\n", num);
212      }
213      else
214      {
215          printf("success delete watchpoint %d\n", num);
216      }
217      return 0;
218  }

```

测试监视点

```

[src/monitor/debug/expr.c,111,make_token] match rules[15] = "\$(eax|ebx|ecx|edx|esi|edi|esp|ebp|eip|)" at position 0 with len 4: $eip
Watchpoint 0: $eip
Old value = 0x00100005
New value = 0x0010000a
(nemu) c
[src/monitor/debug/expr.c,111,make_token] match rules[15] = "\$(eax|ebx|ecx|edx|esi|edi|esp|ebp|eip|)" at position 0 with len 4: $eip
Watchpoint 0: $eip
Old value = 0x0010000a
New value = 0x0010000c
[src/monitor/debug/expr.c,111,make_token] match rules[15] = "\$(eax|ebx|ecx|edx|esi|edi|esp|ebp|eip|)" at position 0 with len 4: $eip
Watchpoint 0: $eip
Old value = 0x0010000c
New value = 0x00100012
[src/monitor/debug/expr.c,111,make_token] match rules[15] = "\$(eax|ebx|ecx|edx|esi|edi|esp|ebp|eip|)" at position 0 with len 4: $eip
Watchpoint 0: $eip
Old value = 0x00100012
New value = 0x00100017
[src/monitor/debug/expr.c,111,make_token] match rules[15] = "\$(eax|ebx|ecx|edx|esi|edi|esp|ebp|eip|)" at position 0 with len 4: $eip
Watchpoint 0: $eip
Old value = 0x00100017
New value = 0x00100021
[src/monitor/debug/expr.c,111,make_token] match rules[15] = "\$(eax|ebx|ecx|edx|esi|edi|esp|ebp|eip|)" at position 0 with len 4: $eip
Watchpoint 0: $eip
Old value = 0x00100021
New value = 0x00100026
nemu: HIT GOOD TRAP at eip = 0x00100026

[src/monitor/debug/expr.c,111,make_token] match rules[15] = "\$(eax|ebx|ecx|edx|esi|edi|esp|ebp|eip|)" at position 0 with len 4: $eip
Watchpoint 0: $eip
Old value = 0x00100026
New value = 0x00100027
(nemu) info w
watchpoint:
NO.  expr          hitTimes
0    $eip          0
(nemu) c
Program execution has ended. To restart the program, exit NEMU and run again.
(nemu) d 0
Free watchpoint 0
(nemu) info w
No watchpoint now
(nemu) q
ics@e3b48c25a345:~/ics2017/nemu$

```



```

(nemu) w $eip
[src/monitor/debug/expr.c,108,make_token] match rules[3] = "\$(eax|ecx|edx|ebx|esp|ebp|esi|edi|
len 4: $eip
RESULT=1048576
success: set watchpoint 0 ,oldValue=1048576
(nemu) info w
watchpoint:
NO.   expr   hitTimes
0     $eip   0
(nemu) d 0
success delete watchpoint 0
(nemu) w $eip
[src/monitor/debug/expr.c,108,make_token] match rules[3] = "\$(eax|ecx|edx|ebx|esp|ebp|esi|edi|
len 4: $eip
RESULT=1048576
success: set watchpoint 1 ,oldValue=1048576
(nemu) si 2
100000:  b8 34 12 00 00                                movl $0x1234,%eax
[src/monitor/debug/expr.c,108,make_token] match rules[3] = "\$(eax|ecx|edx|ebx|esp|ebp|esi|edi|
len 4: $eip
RESULT=1048581
Hardware watchpoint 1:$eip
Old value:1048576
New value:7367013

(nemu) info w
watchpoint:
NO.   expr   hitTimes
1     $eip   1
(nemu) si 2
100005:  b9 27 00 10 00                                movl $0x100027,%ecx

```

实现监视点让程序暂停执行的效果(断点)

如果检测到一个监视点表达式值发生变化 (check__wp() 返回值不为true), 则改变客户程序的执行状态为 NEMU__STOP 实现暂停客户程序的执行。

```

31  #ifdef DEBUG
32      /* TODO: check watchpoints here. */
33      if(!check_wp()){
34          nemu_state=NEMU_STOP;
35      }
36  #endif

```

```

For help, type "help"
(nemu) w $eip==0x100005
[src/monitor/debug/expr.c,106,make_token] match rules[3] = "\\$(eax|ecx|edx|ebx|esp|ebp|esi|edi|eip|ax|cx|dx|bx|sp|bp|si|di|a
len 4: $eip
[src/monitor/debug/expr.c,106,make_token] match rules[10] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,106,make_token] match rules[1] = "0x[0-9A-Fa-f][0-9A-Fa-f]*" at position 6 with len 8: 0x100005
RESULT=0
success: set watchpoint 0 ,oldValue=0
(nemu) c
[src/monitor/debug/expr.c,106,make_token] match rules[3] = "\\$(eax|ecx|edx|ebx|esp|ebp|esi|edi|eip|ax|cx|dx|bx|sp|bp|si|di|a
len 4: $eip
[src/monitor/debug/expr.c,106,make_token] match rules[10] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,106,make_token] match rules[1] = "0x[0-9A-Fa-f][0-9A-Fa-f]*" at position 6 with len 8: 0x100005
RESULT=1
Hardware watchpoint 0:$eip==0x100005
Old value:0
New value:1
(nemu) q

```

四、遇到的BUG及解决思路

- 遇到最主要的BUG就是在进行阶段二的表达式求值时，由于运算符的左右结合性不同，并且一开始对于运算符的优先级处理欠考虑，导致在执行p命令对表达式求值时，一部分的合法表达式会报错，比如a-b-c,这样同级的运算符之间的优先级没有处理好。
 - 解决：我通过修改主操作符选择逻辑，调整所有操作符的优先级数值，确保正确的优先级顺序。并且对于左结合的操作符，修改条件判断，允许在相同优先级时更新 op 为最后出现的位置。这样，单目运算符的优先级高于乘除，且右结合的情况下，遇到连续的单目运算符会选择第一个（右结合）。

```

For help, type "help"
(nemu) p 2-1-1
[src/monitor/debug/expr.c,516,make_token] match rules[2] = "0|[1-9][0-9]*" at position 0 with len 1: 2
[src/monitor/debug/expr.c,516,make_token] match rules[5] = "-" at position 1 with len 1: -
[src/monitor/debug/expr.c,516,make_token] match rules[2] = "0|[1-9][0-9]*" at position 2 with len 1: 1
[src/monitor/debug/expr.c,516,make_token] match rules[5] = "-" at position 3 with len 1: -
[src/monitor/debug/expr.c,516,make_token] match rules[2] = "0|[1-9][0-9]*" at position 4 with len 1: 1
RESULT=0
the value of expr is 0
(nemu) p 1*4-3*2

```

- 此外，在实现断点功能时，`$eip==addr` 的判断结果有误，尝试修改了很久，也一直不行，最后经过检查，是因为由于没有正确处理`®==addr`的判断，以及有个 `strcmp` 没加 `\0`，导致寄存器名称匹配的判断逻辑有问题。

五、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 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?

(1) 查阅 i386 手册

1. EFLAGS 寄存器中的 CF 位是什么意思?

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.

Appendix C Status Flag Summary

Status Flags' Functions

Bit	Name	Function
0	CF	Carry Flag — Set on high-order bit carry or borrow; cleared otherwise.
2	PF	Parity Flag — Set if low-order eight bits of result contain an even number of 1 bits; cleared otherwise.
4	AF	Adjust flag — Set on carry from or borrow to the low order four bits of AL; cleared otherwise. Used for decimal arithmetic.
6	ZF	Zero Flag — Set if result is zero; cleared otherwise.
7	SF	Sign Flag — Set equal to high-order bit of result (0 is positive, 1 if negative).
11	OF	Overflow Flag — Set if result is too large a positive number or too small a negative number (excluding sign-bit) to fit in destination operand; cleared otherwise.

CF 是 EFLAGS 寄存器的进位标志, 如果运算导致最高位产生进位或者借位则为 1, 否则为 0。

2. ModR/M 字节是什么?

17.2.1 ModR/M and SIB Bytes

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

INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986

- **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

大多数可以引用内存中的操作数的指令在主操作码字节之后都有一个寻址形式的字节，即为ModR/M字节。其包括MOD、REG/OPCODE、R/M三方面内容，分别表示索引类型或者寄存器编号、寻址模式编码等信息。mod域和r/m域共同决定其中一个操作数的寻址方式，mod域指明了从内存中还是从寄存器中寻址，r/m域可以指定一个寄存器作为寻址的位置。reg域是操作码的补充说明，它与操作码共同决定另一个操作数的寻址方式。

3. mov 指令的具体格式是怎么样子的？

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
C0 + ii	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

NOTES:

moffs8, moffs16, and moffs32 all consist of a simple offset relative to the segment base. The 8, 16, and 32 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 bits.

Operation

DEST ← SRC;

Description

MOV copies the second operand to the first operand.

MOV 目的寄存器 源操作数

(2) 统计代码行

我们可以通过如下指令获取代码行数，其中第一个为全部行数，第二个为去空行后行数。

```
● ics@e3b48c25a345:~/ics2017/nemu$ find . -name "*.ch" |xargs cat|wc -l
4135
● ics@e3b48c25a345:~/ics2017/nemu$ find . -name "*.ch" |xargs cat|grep -v ^$|wc -l
3451
```

为统计新增代码行数，我们可以将代码切到上一个实验的分支，统计行数，在切换回本实验的分支，统计行数，相减则可得出结果。

我们把这条命令写入Makefile文件，随着实验进度的推进，我们可以很方便地统计工程的代码行数，输入make count就会自动运行统计代码行数的命令。

```
count:
    git checkout pa0
    find . -name "*.ch" |xargs cat|wc -l
    git checkout pa1
    find . -name "*.ch" |xargs cat|wc -l
```

```
● ics@e3b48c25a345:~/ics2017/nemu$ make count
git checkout pa0
M      nemu/Makefile
Switched to branch 'pa0'
find . -name "*.ch" |xargs cat|wc -l
3487
git checkout pa1
M      nemu/Makefile
Switched to branch 'pa1'
find . -name "*.ch" |xargs cat|wc -l
4135
```

(3) gcc中的-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: 在发生警告时停止编译操作，即要求GCC将所有的警告当成错误进行处理，从而终止编译操作。

使用-Wall和-Werror 可以找出所有存在的或者潜在的错误，提高代码的安全性，优化程序，便于进行代码维护以及debug。

六、总结与感悟

本次实验不仅让我掌握了调试器的核心功能实现方法，更深化了对计算机系统底层运行机制的理解，也让我对i386有了初步认识。当然，实验中也遇到了很多难题，提升了我发现问题和解决问题的能力。