计算机系统设计 PA1

1. 实验目的

- 熟悉指令集的架构
- 认识图灵机中程序运行的原理
- 熟悉存储器之间的存储关系
- 学习GDB，并在PA上实现简易调试器
- 学习表达式求值，监视点的实现方法

2. 实验内容

PA1 的实验主要为简易调试器的实现。其中有三个阶段：

阶段一：

- 实现正确的寄存器结构体
- 实现解析命令
- 实现单步执行
- 实现打印寄存器
- 实现内存扫描

阶段二：

- 实现词法分析
- 实现递归求值
- 实现调试中的表达式求值

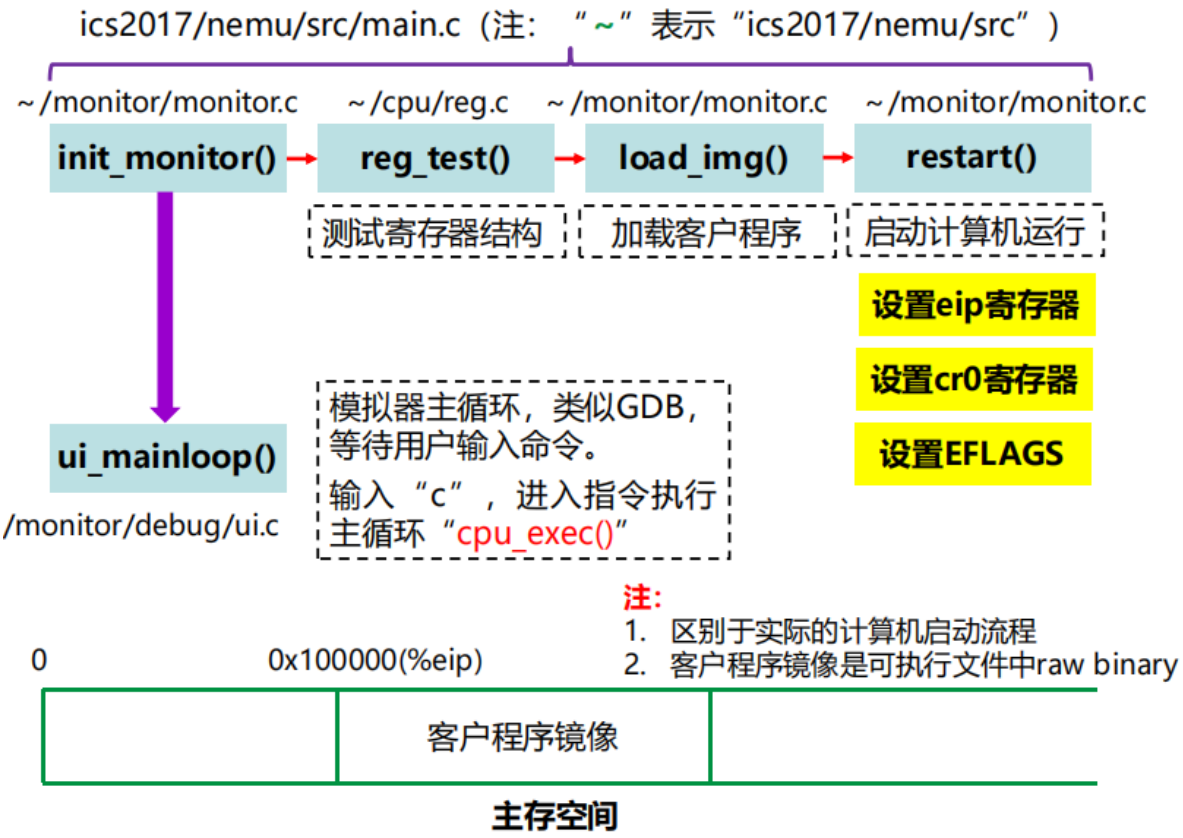
阶段三：

- 实现断点
- 实现监视点
- 学习断点相关知识与i386手册

3. 阶段一

3.1 NEMU执行流程

NEMU的主体代码框架组成了最简单的计算机。为了了解NEMU的原理，我们需要了解其执行流程。



上图为NEMU的主体代码框架示意图，接下来阅读源码：

nemu/src/main.c中，主要调用的`init_monitor`：

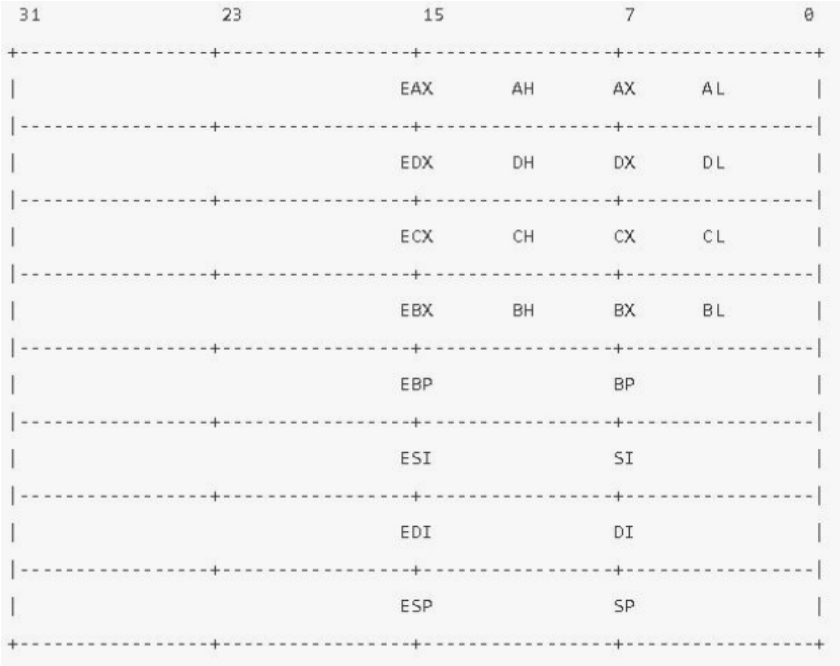
```
int init_monitor(int argc, char *argv[]) {
  /* Perform some global initialization. */

  /* Parse arguments. */
  parse_args(argc, argv);

  /* Open the log file. */
  init_log();

  /* Test the implementation of the `CPU_state' structure. */
  reg_test();   //生成随机数，测试寄存器结构的正确性

#ifdef DIFF_TEST
  /* Fork a child process to perform differential testing. */
  init_difftest();
#endif

  /* Load the image to memory. */
  load_img();   //载入带有客户程序的镜像文件，NEMU直接将客户镜像读入固定内存位置0x10000，缺省时使用mov程序作为客户程序

  /* Initialize this virtual computer system. */
  restart();    //模拟计算机启动，并将EIP初始值设置为0x100000以确保CPU从该位置开始执行程序

  /* Compile the regular expressions. */
  init_regex();     //正则式

  /* Initialize the watchpoint pool. */
  init_wp_pool();   //监视点
```

```
  /* Initialize devices. */
  init_device();    //设备初始化

  /* Display welcome message. */
  welcome();    //输出信息和NEMU编译时间

  return is_batch_mode;
}
```

执行完`init_monitor`之后，开始执行`ui_mainloop`函数，此函数主要用于从用户处接收指令，实现交互功能。

3.2 实现寄存器结构体

3.2.1 分析变量与函数



寄存器有以下几个重要的变量与函数：

1. 寄存器结构体：除了EIP之外，有8个32位、8个16位、8个8位寄存器。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。
2. 匿名Union结构：这里需要注意Union与Struct的不同。Union中的各变量互斥，共享同一内存首地址，而且各种变量名可以同时使用，寄存器的实现基于Union特性实现

   > 结构体和共用体的区别在于：结构体的各个成员会占用不同的内存，互相之间没有影响；而共用体的所有成员占用同一段内存，修改一个成员会影响其余所有成员。
   >
   > 结构体占用的内存大于等于所有成员占用的内存的总和（成员之间可能会存在缝隙），共用体占用的内存等于最长的成员占用的内存。共用体使用了内存覆盖技术，同一时刻只能保存一个成员的值，如果对新的成员赋值，就会把原来成员的值覆盖掉。
3. 相关变量：在`nemu/include/spu/reg.h`中，我们看到了三个外部变量：`regsl`、`regsw`、`regsb`，查看它们在reg.c中的定义：

```
const char *regsl[] = {"eax", "ecx", "edx", "ebx", "esp", "ebp", "esi",
"edi"};
const char *regsw[] = {"ax", "cx", "dx", "bx", "sp", "bp", "si", "di"};
const char *regsb[] = {"al", "cl", "dl", "bl", "ah", "ch", "dh", "bh"};
```

4. 相关函数：

```
#define reg_l(index) (cpu.gpr[check_reg_index(index)]._32)
#define reg_w(index) (cpu.gpr[check_reg_index(index)]._16)
#define reg_b(index) (cpu.gpr[check_reg_index(index) & 0x3]._8[index >> 2])
```

注：R_EAX = 0, R_EDI = 7

- `reg_l(index)`：$index \in \{0 \leq x \leq 7 \mid x \in Z\}$，返回`regsl[i]`所标识的寄存器
- `reg_w(index)`：$index \in \{0 \leq x \leq 7 \mid x \in Z\}$，返回`regsw[i]`所标识的寄存器
- `reg_b(index)`：$index \in \{0 \leq x \leq 7 \mid x \in Z\}$，返回`regsb[i]`所标识的寄存器
- `reg_test()`：这是寄存器的测试函数，它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。主要有两个部分：

  1. 从EAX开始，到EDI结束，检验所有的寄存器满足上图所示的结构。例如，要求EAX的低16位为AX，AX的高八位为AH，低8位为AL。

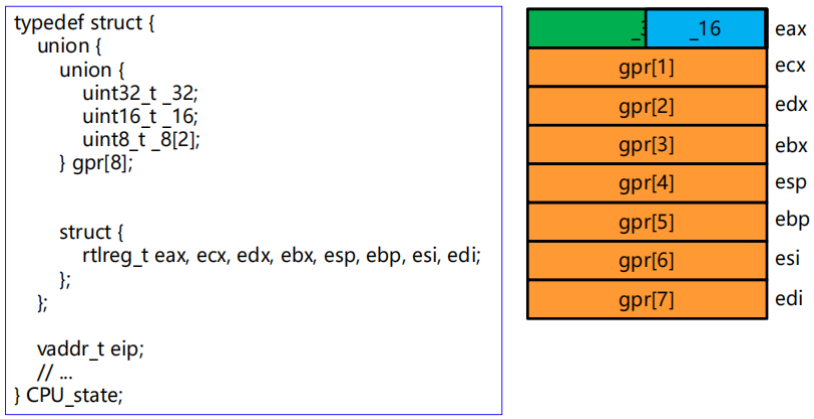
     ```
       //判断reg_w(0) ~ reg_w(7)是否等于sample[0] ~ sample[7]的低16位
       int i;
       for (i = R_EAX; i <= R_EDI; i ++) { // R_EAX = 0, R_EDI = 7
         sample[i] = rand(); // sample[0] ~ sample[7]
         reg_l(i) = sample[i]; // reg_l(0) ~ reg_l(7)
         assert(reg_w(i) == (sample[i] & 0xffff)); //判断reg_w(0) ~
     reg_w(7)是否等于sample[0] ~ sample[7]的低16位
       }

       // reg_b(0) ~ reg_b(7)是否等于sample[0] ~ sample[7]的低16位中的高低两
     个8位
       assert(reg_b(R_AL) == (sample[R_EAX] & 0xff));
       assert(reg_b(R_AH) == ((sample[R_EAX] >> 8) & 0xff));
       assert(reg_b(R_BL) == (sample[R_EBX] & 0xff));
       assert(reg_b(R_BH) == ((sample[R_EBX] >> 8) & 0xff));
       assert(reg_b(R_CL) == (sample[R_ECX] & 0xff));
       assert(reg_b(R_CH) == ((sample[R_ECX] >> 8) & 0xff));
       assert(reg_b(R_DL) == (sample[R_EDX] & 0xff));
       assert(reg_b(R_DH) == ((sample[R_EDX] >> 8) & 0xff));
     ```

  2. 确保可以通过寄存器名直接访问32位寄存器的内容，即使用`cpu.eax`，而不需要使用`reg_l(index)`等函数：

```
  assert(sample[R_EAX] == cpu.eax);
  assert(sample[R_ECX] == cpu.ecx);
  assert(sample[R_EDX] == cpu.edx);
  assert(sample[R_EBX] == cpu.ebx);
  assert(sample[R_ESP] == cpu.esp);
  assert(sample[R_EBP] == cpu.ebp);
  assert(sample[R_ESI] == cpu.esi);
  assert(sample[R_EDI] == cpu.edi);
```

3.2.2 代码实现

正确的寄存器结构应如下所示：



因此，正确的代码如下所示。修改`nemu/include/cpu/reg.h`：

```
typedef struct {
  union{
    /* data */
    union {
      uint32_t _32;
      uint16_t _16;
      uint8_t _8[2];
    } gpr[8];
    struct
    {
      rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
    };
  };
  /* 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.
   */
  vaddr_t eip;

} CPU_state;
```

3.2.3 运行结果

执行`make ISA=x86 run`指令，得到时间与结果：

```
./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: 18:42:19, Mar 13 2023
For help, type "help"
(nemu)
```

如图所示，成功运行成功。

3.2.4 究竟要运行多久?

> 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?

打开`nemu/src/cpu/exec.c`，观察相关函数。

NEMU 将不断执行指令,直到遇到以下情况之一,才会退出指令执行的循环:

- 达到要求的循环次数.
- 客户程序执行了 `nemu_trap` 指令.这是一条特殊的指令,机器码为 0xd6。它是为了在 NEMU 中让客户程序指示执行的结束而加入的.

定义在`nemu/src/monitor/debug/ui.c`中的`ui_mainloop`用户界面主循环，可以在这里输入很多指令来监控和调试。当命令键入c时，循环就调用`cmd_c`函数，里面`cpu_exec`函数传入的值是-1。

```
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026

(nemu)
```

为了表示程序是否正常结束，当触发nemu_trap时，NEMU将会根据这个结束状态参数来设置NEMU的结束状态, 并根据不同的状态输出不同的结束信息, 包括：

- HIT GOOD TRAP - 客户程序正确地结束执行
- HIT BAD TRAP - 客户程序错误地结束执行
- ABORT - 客户程序意外终止, 并未结束执行

在`cpu_exec.c()`函数中，有以下代码：

```
bool print_flag = n < MAX_INSTR_TO_PRINT;

for (; n > 0; n --) {
  /* Execute one instruction, including instruction fetch,
   * instruction decode, and the actual execution. */
  exec_wrapper(print_flag);
```

n是无符号整型，所以-1就是无符号最大的数字，那么函数里的for循环可以执行最大次数的循环，从而让cpu处理之后的指令。

3.2.5 谁来指示程序的结束

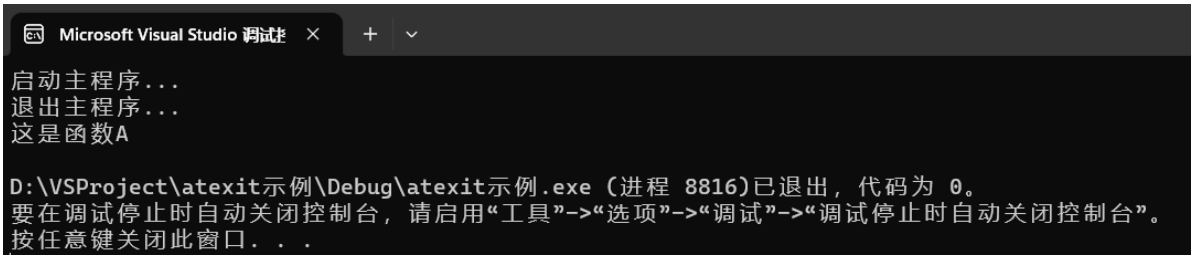
正常程序退出时调用`atexit`登记函数，1个进程可以登记若干个函数，这些函数由exit自动调用，这些函数被称为终止处理函数，`atexit`函数可以登记这些函数。如果函数成功注册，则该函数返回零，否则返回一个非零值。

以下是终止函数的区别：

1. 使用exit函数是会结束进程后自动刷新缓冲区，且是正常退出
2. 使用_exit函数是不会在进程结束后刷新缓冲区，且是立即终止进程（非正常退出）
3. atexit函数只由进程在正常退出情况下才能使用，因此atexit能与exit一起使用，但不能与_exit一起使用

以下实例演示了`atexit()`函数的用法：

```
#include <stdio.h>
#include <stdlib.h>

void functionA ()
{
   printf("这是函数A\n");
}

int main ()
{
   /* 注册终止函数 */
   atexit(functionA );
   printf("启动主程序...\n");
   printf("退出主程序...\n");
   return(0);
}
```



可以看出，`functionA`在最后才被调用，证明了return后才会调用`atexit`函数。

3.3 基础设施：简易调试器

调试器需要实现的功能如下图所示：

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
打印栈帧链(3)	`bt`	`bt`	打印栈帧链

已实现

正则表达式递归

链表

- 简易调试器：nemu/src/monitor/debug/ui.c
- 表达式求值： nemu/src/monitor/debug/expr.c
- 监视点： nemu/src/monitor/debug/watchpoint.c

3.3.1 实现解析命令

解析命令需要用到`cmd_table`结构。`ui_mainloop`函数使用`strtok()`函数解析命令，并根据输入 的命令，调用`cmd_table[i].handler(args)`来执行相关的处理函数，其格式为：`{命令，描述，处理函数}`。

在`nemu/src/monitor/debug/ui.c`中实现命令表，其代码如下：

```
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 },

  /* TODO: Add more commands */
  { "si", "args:[N]; exectue [N] instructions step by step", cmd_si}, //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
  { "info", "args:r/w;print information about register or watch point ", cmd_info}, //打印寄存器状态
  { "x", "x [N] [EXPR];sacn the memory", cmd_x }, //内存扫描
  { "p", "expr", cmd_p}, //表达式
  { "w", "set the watchpoint", cmd_w}, //添加监视点
  { "d", "delete the watchpoint", cmd_d} //删除监视点
};
```

执行效果如下所示：

```
(nemu) help
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - args:[N]; exectue [N] instructions step by step
info - args:r/w;print information about register or watch point
x - x [N] [EXPR];sacn the memory
p - expr
w - set the watchpoint
d - delete the watchpoint
(nemu)
```

3.3.2 实现单步执行

单步执行的格式为`si [N]`，程序单步执行`N`条指令后暂停, 当N没有给出时, 缺省为默认为1。

1. 传入`cmd_si()`函数的参数为字符串，现在需要利用一些方法将其分解为两部分，分别为`si`（空格）和`N`(N是字符串类型的数字），N的部分存到字符串`arg`中，此过程中需要用到`strtok()`库函数

   > **strtok( ) 的使用**
   > C 库函数 char *strtok(char *str, const char *delim) 分解字符串 str 为一组字符串，delim 为分隔符。

2. 根据字符串`arg`来判断需要执行的指令数`i`，需要使用`sscanf()`库函数，将字符串`arg`改为int型的数字`i`

   > **sscanf( ) 的使用**
   > C 库函数 int sscanf(const char *str, const char *format, ...) 从字符串读取格式化输入。
   > str – 这是 C 字符串，是函数检索数据的源。
   > format – 这是 C 字符串，包含了以下各项中的一个或多个：空格字符、非空格字符和format说明符。

在 nemu/src/monitor/debug/ui.c 添加 cmd_si 函数，其代码如下：

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

  else {
    int temp = sscanf(args, "%lu", &N);
    if(temp <= 0) {
      printf("args error in cmd_si\n");
      return 0;
    }
  }

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

执行效果如下所示：输入si和需要执行的步骤N，进行打印

```
(nemu) si 5
  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)
  100012:   bb 02 00 00 00                      movl $0x2,%ebx
(nemu) si 1
  100017:   66 c7 84 99 00 e0 ff ff 01 00       movw $0x1,-0x2000(%ecx,%ebx,4)
(nemu) si 1
  100021:   b8 00 00 00 00                      movl $0x0,%eax
(nemu) si 1
nemu: HIT GOOD TRAP at eip = 0x00100026

  100026:   d6                                  nemu trap (eax = 0)
(nemu)
```

3.3.3 实现打印寄存器

在 `nemu/src/monitor/debug/ui.c` 添加 `cmd_info` 函数，其代码如下，其主要作用是printf每一个寄存器的状态；输入参数 r 对寄存器直接进行打印，输入参数 w 对监视点进行打印：

```
static int cmd_info(char *args) { //监视点信息查看
  char s;
  if(args == NULL) {
    printf("args error in cmd_info (miss args)\n");
    return 0;
  }

  int temp = sscanf(args, "%c", &s);
  if(temp <= 0) {
    //解析失败
    printf("args error in cmd_info\n");
    return 0;
  }

  if(s == 'w') {
    //打印监视点信息
    print_wp();;
    return 0;
  }

  if(s == 'r') {
    //打印寄存器
    //32bit
    for(int i = 0; i < 8; i++) {
      printf("%s  0x%x\n", regsl[i], reg_l(i));
    }
    printf("eip  0x%x\n", cpu.eip);
    //16bit
    for(int i = 0; i < 8; i++) {
      printf("%s  0x%x\n", regsw[i], reg_w(i));
    }
    //8bit
    for(int i = 0; i < 8; i++)
    {
```

```
      printf("%s  0x%x\n", regsb[i], reg_b(i));
    }
    return 0;
  }

  //如果产生错误
  printf("args error in cmd_info\n");
  return 0;
}
```

执行效果如下：

```
(nemu) info r
eax  0xdb2e997
ecx  0x3986402d
edx  0x16f2217f
ebx  0xa134180
esp  0x2609a528
ebp  0x1ab72c3a
esi  0x6b7dba75
edi  0x7a0a0aaa
eip  0x100000
ax  0xe997
cx  0x402d
dx  0x217f
bx  0x4180
sp  0xa528
bp  0x2c3a
si  0xba75
di  0xaaa
al  0x97
cl  0x2d
dl  0x7f
bl  0x80
ah  0xe9
ch  0x40
dh  0x21
bh  0x41
(nemu)
```

在此处，我们能直观地感受到各个寄存器物理结构的关联。比如`eax`是0xdb2e991，而`ax`是0xe997，`al`为0x97，`ah`为0xe9.

3.3.4 实现内存扫描

在`nemu/src/memory/memory.c`中有以下函数的定义。可以看到，`read`函数需要传入两个参数，分别是起始地址和扫描长度。

```
uint32_t paddr_read(paddr_t addr, int len) {
  return pmem_rw(addr, uint32_t) & (~0u >> ((4 - len) << 3));
}

void paddr_write(paddr_t addr, int len, uint32_t data) {
  memcpy(guest_to_host(addr), &data, len);
}

uint32_t vaddr_read(vaddr_t addr, int len) {
  return paddr_read(addr, len);
```

```
}

void vaddr_write(vaddr_t addr, int len, uint32_t data) {
  paddr_write(addr, len, data);
}
```

任务中要求以16进制形式输出连续的N个4字节，因此，将address和4传入`vaddr_read`函数就可以得到，再用for循环循环nLen次，每4次打印一次当前地址，就可以实现内存的扫描。

在`nemu/src/monitor/debug/ui.c`添加`cmd_x`函数，其代码如下

```
static int cmd_x(char *args) {
  int nLen = 0;
  vaddr_t addr;
  int temp = sscanf(args, "%d 0x%x", &nLen, &addr);
  if(temp <= 0) {
    //解析失败
    printf("args error in cmd_si\n");
    return 0;
  }
  printf("Memory:");
  for(int i = 0; i < nLen; i++) {
    if(i % 4 == 0) {
      printf("\n0x%x:  0x%02x", addr + i, vaddr_read(addr + i, 1));
    }
    else {
      printf("  0x%02x", vaddr_read(addr + i, 1));
    }
  }
  printf("\n");
  return 0;
}
```

执行效果如下：

```
(nemu) x 39 0x100000
Memory:
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
(nemu)
```

与`load_default_img()`进行对比：

```
static inline int load_default_img() {
  const uint8_t img []  = {
    0xb8, 0x34, 0x12, 0x00, 0x00,        // 100000:  movl  $0x1234,%eax
```

```
    0xb9, 0x27, 0x00, 0x10, 0x00,        // 100005:  movl  $0x100027,%ecx
    0x89, 0x01,                          // 10000a:  movl  %eax,(%ecx)
    0x66, 0xc7, 0x41, 0x04, 0x01, 0x00,  // 10000c:  movw  $0x1,0x4(%ecx)
    0xbb, 0x02, 0x00, 0x00, 0x00,        // 100012:  movl  $0x2,%ebx
    0x66, 0xc7, 0x84, 0x99, 0x00, 0xe0,  // 100017:  movw
$0x1,-0x2000(%ecx,%ebx,4)
    0xff, 0xff, 0x01, 0x00,
    0xb8, 0x00, 0x00, 0x00, 0x00,        // 100021:  movl  $0x0,%eax
    0xd6,                                // 100026:  nemu_trap
  };

  Log("No image is given. Use the default build-in image.");

  memcpy(guest_to_host(ENTRY_START), img, sizeof(img));

  return sizeof(img);
}
```

完全一致，验证代码正确性。

4. 阶段二

阶段二为**表达式求值**，主要有两方面内容：

1. 识别出表达式中的单元
2. 根据表达式中的归纳定义递归求值

4.1 词法分析

想要求出表达式的值，第一步**要解决的问题**是识别字符串中的数字、符号、括号等等，**解决方法**是利用正则表达式刻画字符的组合规律，将字符串切割成一个个的有确定类型的token。

表达式求值中只能出现以下token：

- 十进制整数
- +, - ,*, /
- (,)
- 空格串（一个或多个空格）

需要完成：

- 为算术表达式中的各种token添加规则，需要注意C语言字符串中转义字符的存在和正则表达式中的元字符的功能
- 在成功识别出token之后，将token的信息一次记录到`tokens`数组中

4.1.1 分析变量和函数

- `static struct rule`：是由正则表达式和token类型组成的二元组

```
static struct rule {
  char *regex;
  int token_type;
}
```

- `enum { 单词类型 }`：多字符 token 的类型标识。单字符 token 直接使用它自身所对应的 ASCII 码作为该token 的类型标识，如+的 token 类型是'+'。由于 char 型（1个Byte）从 0-255，故 TK_NOTYPE从 256 开始编号。
- `struct token`：

  ```
  typedef struct token {
    int type;
    char str[32];
  } Token;
  ```

  其中 type 成员用于记录 token 的类型。大部分 token 只要记录类型就可以了，例如`+,-,*,/`，但这对于有些token 类型是不够的：如果我们只记录了一个十进制整数 token 的类型，在进行求值的时候我们还是不知道这个十进制整数是多少。这时我们应该将 token 相应的子串也记录下来,str 成员就是用来做这件事情的。
- `make_token`函数：
  - `position` 标记当前处理位置，使用 rule 中规则匹配字符串。失败时返回 false。
  - `struct Token tokens[32]`：顺序记录被识别的 token 信息。
  - `int nr_token`：记录已识别的 token 数目。
  - 如果尝试了所有的规则都无法在当前位置识别出 token，识别将会失败，这通常是待求值表达式并不合法造成的，make_token()函数将返回 false，表示词法分析失败。

4.1.2 代码实现

首先我们需要使用正则表达式分别编写用于识别这些 token 类型的规则。在框架代码中，一条规则是由正则表达式和 token 类型组成的二元组。框架代码中已经给出了+和空格串的规则，其中空格串的 token 类型是TK_NOTYPE，因为空格串并不参加求值过程，识别出来之后就可以将它们丢弃了；+的 token 类型是'+'。事实上token 类型只是一个整数，只要保证不同的类型的 token 被编码成不同的整数就可以了。

其中要特别注意，如果识别的符号为正则表达式的元符号则需要加上\符号进行转义。

代码如下：

```
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
  {"0x[0-9A-Fa-f][0-9A-Fa-f]*", TK_HEX},
  {"0|[1-9][0-9]*", TK_NUMBER}, //数字

{"\\$(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},
  {"==", TK_EQ},
  {"!=", TK_NEQ},
  {"&&", TK_AND},
  {"\\|\\|", TK_OR},
```

```
  {"!", '!'},
  {"\\+", '+'},     // plus
  {"-", '-'},
  {"\\*", '*'},
  {"\\/", '/'},
  {"\\(", '('},
  {"\\)", ')'},
};
```

扩充完正则表达式规则以后，需要做的就是对输入的字符串进行分析，对每一个符号进行分类，再将各个类型存储在`tokens[]`数组中，完成此操作的函数为`make_token`函数。

其中八进制数和十六进制数将会被提前处理，而寄存器则将会跳过开头的美元符号，需要补充的`switch`部分将表达式中每一个部分用对应的类型及具体值存储到`tokens[nr_token].str`中（如NUM类型里存具体的数字，RESITER类型里存具体的寄存器的名字等等），其代码如下：

```
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 (regexec(&re[i], e + position, 1, &pmatch, 0) == 0 && pmatch.rm_so ==
0) {
        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(substr_len > 32) { //检查是否超过最大长度
          assert(0);
        }
        if(rules[i].token_type == TK_NOTYPE) { //跳过空格
          break;
        }
        else {
          tokens[nr_token].type = rules[i].token_type;
          switch (rules[i].token_type) {
          case TK_NUMBER: //数字
            strncpy(tokens[nr_token].str, substr_start, substr_len);
            *(tokens[nr_token].str + substr_len) = '\0';
            break;
          case TK_HEX: //16进制数
            strncpy(tokens[nr_token].str, substr_start + 2, substr_len - 2); //跳
过开头的0x
```

```
            *(tokens[nr_token].str + substr_len - 2) = '\0';
            break;
          case TK_REG:  //寄存器
            strncpy(tokens[nr_token].str, substr_start + 1, substr_len - 1); //跳过开头的$
            *(tokens[nr_token].str + substr_len - 1) = '\0';
          }
          printf("Success record : nr_token = %d, dtype = %d, str = %s\n", nr_token, tokens[nr_token].type, tokens[nr_token].str);
          nr_token += 1; //记录成功后，nr_token加1
          break;
        }
      }
    }
    if (i == NR_REGEX) { //没有匹配到任何规则
      printf("no match at position %d\n%s\n%*.s^\n", position, e, position, "");
      return false;
    }
  }

  return true;
}
```

4.2 递归求值

4.2.1 分析变量与函数

词法分析部分我们已将`token`存入到了`tokens[]`数组中，接下来需要用递归的方法求出表达式的值，此功能在`eval()`函数中实现。

我们使用BNF来进行递归求值。

- `check_parentheses()`：用于判断表达式是否被一对匹配的括号包围着,同时检查表达式的左右括号是否匹配,如果不匹配,这个表达式肯定是不符合语法的,也就不需要继续进行求值了。
- `findDominantOp()`：在一个 token 表达式中寻找 DominantOp，返回其索引位置，不存在时返回-1。
- `eval()`：表达式求值函数

4.2.2 代码实现

首先需要考虑的是检查括号匹配。主要的思想是利用一个计数器来计算匹配的数目，如果最终计数器的值不为 0 则说明括号不匹配：

- 如果遍历期间 count<0 则说明出现多余的`')'`
- 如果遍历结束时 count>0 则说明出现多余的`'('`
- 只有当遍历结束后count=0才说明匹配成功，返回`true`，否则返回`false`

其代码如下：

```
//判断括号的匹配
bool check_parentheses(int p, int q) {
  if(p >= q) {
    //右括号个数少于左括号
    printf("error: p >= q in check_parntheses\n");
    return false;
```

```
  }
  if(tokens[p].type != '(' || tokens[q].type != ')'){
    //括号不匹配
    return false;
  }
  int cnt = 0; //记录当前未匹配的左括号的数目
  for(int curr = p + 1; curr < q; curr++) {
    if(tokens[curr].type == '(') { //遇到左括号，数目加1
      cnt++;
    }
    if(tokens[curr].type == ')') { //遇到右括号，数目减1
      if(cnt != 0) {
        cnt--;
      }
      else {
        //左右括号不匹配
        return false;
      }
    }
  }
  if(cnt == 0) {
    return true;
  }
  else {
    return false;
  }
}
```

然后需要利用一个函数来计算表达式优先级最低的运算符的位置，如果有同为最低优先级的预算符，返回最后被结合的运算符的索引位置。其主要思想为：

- 非运算符的 token 不是 dominant operator
- 出现在一对括号中的 token 不是 dominant operator。注意到这里不会出现有括号包围整个表达式的情况， 因为这种情况已经在 check_parentheses()相应的 if 块中被处理了。
- dominant operator 的优先级在表达式中是最低的，这是因为 dominant operator 是最后一步才进行的运算符。
- 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才是 dominant operator。一个例子是 1+2+3，它的 dominant operator 应该是右边的+。

要找出 dominant operator，只需要将 token 表达式全部扫描一遍,就可以按照上述方法唯一确定 dominant operator，然后再根据 dominant operator 的类型对两个子表达式的值进行运算即可。

其代码如下：

```
int findDominantOp(int p, int q) {
  int level=0;
  int pos[5]={-1, -1, -1, -1, -1};
  for(int i = p; i < q; i++){
     if(level == 0) {
        if(tokens[i].type == TK_AND || tokens[i].type == TK_OR) {
          pos[0] = i;
        }
        if(tokens[i].type == TK_EQ || tokens[i].type == TK_NEQ) {
          pos[1] = i;
        }
```

```
        if(tokens[i].type == '+' || tokens[i].type == '-') {
          pos[2] = i;
        }
        if(tokens[i].type == '*' || tokens[i].type == '/') {
          pos[3] = i;
        }
        if(tokens[i].type == TK_NEGATIVE || tokens[i].type == TK_DEREF ||
tokens[i].type == '!') {
          pos[4] = i;
        }
      }
      if(tokens[i].type=='(') {
        level++;
      }
      if(tokens[i].type==')') {
        level--;
      }
  }
  for(int i = 0; i < 5; i++) {
    if(pos[i] != -1) {
      return pos[i];
    }
  }
  printf("error in findDominantOp\n");
  printf("[p=%d, q=%d]\n",p,q);
  assert(0);
}
```

根据实验指导书编写 eval 函数，用于值的计算，其中的参数 p 和 q 分别代表这个子表达式的开始位置和结束为止，其代码如下：

```
uint32_t eval(int p, int q) {
  if(p > q) {
    printf("error:p>q in eval, p = %d, q = %d\n", p, q);
    assert(0);
  }
  if(p == q) {
    int num;
    switch (tokens[p].type){
      case TK_NUMBER: //数字
        sscanf(tokens[p].str, "%d", &num);
        return num;
      case TK_HEX: //16进制数
        sscanf(tokens[p].str, "%x", &num);
        return num;
      case TK_REG: //寄存器
        for(int i = 0; i < 8; i++) {
          if(strcmp(tokens[p].str, regsl[i]) == 0) {
            return reg_l(i);
          }
          if(strcmp(tokens[p].str, regsw[i]) == 0) {
            return reg_w(i);
          }
          if(strcmp(tokens[p].str, regsb[i]) == 0) {
            return reg_b(i);
```

```
          }
        }
        if(strcmp(tokens[p].str, "eip") == 0) {
          return cpu.eip;
        }
        else {
          printf("error in TK_REG in eval()\n");
          assert(0);
        }
    }
  }
  if(check_parentheses(p, q) == true) { //被括号包围
    return eval(p + 1, q - 1);  //递归子表达式
  }
  else {
    int op = findDominantOp(p, q);//找主运算符
    vaddr_t addr; //TK_DEREF的地址
    int result;
    // 单目运算符
    switch (tokens[op].type) {
      case TK_NEGATIVE: //负号
        printf("Operator= -.\n");
        printf("Value=%d.\n",result);
        return -eval(p + 1, q);
      case TK_DEREF: //指针求值
        addr = eval(p + 1, q);
        result = vaddr_read(addr, 4);
        printf("adddr=%u(0x%x)---->value=%d(0x%08x)\n", addr, addr, result,
result);
        return result;
      case '!':
        result = eval(p + 1, q);
        printf("Operator= !.\n");
        if(result != 0) {
          printf("Value=0.\n");
          return 0;
        }
        else {
          printf("Value=1.\n");
          return 1;
        }
    }
    uint32_t val1 = eval(p, op - 1);
    uint32_t val2 = eval(op + 1, q);
    switch(tokens[op].type) {
      case '+':
        printf("Operator= +.\n");
        printf("Value=%d.\n",val1+val2);
        return val1 + val2;
      case '-':
        printf("Operator= -.\n");
        printf("Value=%d.\n",val1-val2);
        return val1 - val2;
      case '/':
        if(val2==0){
```

```
          printf("Error: The val2 can't be 0.\n");
          assert(0);
        }
        printf("Operator= /.\n");
        printf("Value=%d.\n",val1/val2);
        return val1 / val2;
      case '*':
        printf("Operator= *.\n");
        printf("Value=%d.\n",val1*val2);
        return val1 * val2;
      case TK_EQ:
        printf("Operator= ==.\n");
        printf("Value=%d.\n",val1==val2);
        return val1 == val2;
      case TK_NEQ:
        printf("Operator= !=.\n");
        printf("Value=%d.\n",val1!=val2);
        return val1 != val2;
      case TK_AND:
        printf("Operator= &&.\n");
        printf("Value=%d.\n",val1&&val2);
        return val1 && val2;
      case TK_OR:
        printf("Operator= ||.\n");
        printf("Value=%d.\n",val1||val2);
        return val1 || val2;
      default:
        printf("Error: Invalid operator.\n");
        assert(0);
    }
  }
  return 1;
}
```

4.3 实现调试中的表达式求值

区分减法与负运算、乘法与指针引用:

减法与负运算、乘法与指针解引用在做词法分析阶段无法完成，因为词法分析只负责拆分出单词，负运算与指针解引用是单目运算符，所以只能在该函数中通过遍历tokens数组来区分，通过单目运算符前面的一个单词的类型来确定-与*符号的类型。

4.3.1 代码实现

`expr` 函数完善思路如下:

- 乘号或减号的前一单词为数字或寄存器： TK_NUM ， ） ， TK_HEX 。
- 指针解引用或负号的前一单词：除 TK_NUM 、 TK_HEX 与 ） 的其他单词，或 NULL。
- 优先级：单目运算符高于双目运算符。
- 结合性：单目运算符为左结合。

```
uint32_t expr(char *e, bool *success) {
  if (!make_token(e)) {
    *success = false;
    return 0;
```

```
  }
  /* TODO: Insert codes to evaluate the expression. */
  // TODO();

  // return 0;
  if(tokens[0].type == '-') { //处理-
    tokens[0].type = TK_NEGATIVE;
  }
  if(tokens[0].type == '*') { //处理-
    tokens[0].type = TK_DEREF;
  }
  for(int i = 1; i < nr_token; i++) {
    if(tokens[i].type == '-') {
      // 如果前一个不是数字或者右括号，那么就是负号
      if(tokens[i - 1].type != TK_NUMBER && tokens[i - 1].type != ')') {
        tokens[i].type = TK_NEGATIVE;
      }
    }
    if(tokens[i].type == '*') {
      // 如果前一个不是数字或者右括号，那么就是指针求值
      if(tokens[i - 1].type != TK_NUMBER && tokens[i - 1].type != ')') {
        tokens[i].type = TK_DEREF;
      }
    }
  }
  *success = true;
  return eval(0, nr_token - 1);
}
```

`cmd_p` 表达式求值：

直接调用expr函数。用success来标记是否成功，用r来记录运算结果。

代码如下：

```
static int cmd_p(char *args) {
  //表达式求值
  bool is_success;
  int temp = expr(args, &is_success);
  if(is_success == false) {
    printf("error in expr()\n");
  }
  else {
    printf("the value of expr is:%d\n", temp);
  }
  return 0;
}
```

4.3.2 运行结果

测试样例：

```
435
0x3D
$eip
```

```
$ax

3+34*1
-7-(5+8)
-4*3
$eax*40/4+4
--7

1+12*3==37
1+12*3!=37
3==3&&(*0x100000==0x1234b8)
*0x100000 //0x001234b8
```

测试结果如下所示：

```
[src/monitor/monitor.c,30,welcome] Build time: 18:42:19, Mar 13 2023
For help, type "help"
(nemu) p 425
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 0 with len 3: 425
Success record : nr_token = 0, dtype = 257, str = 425
the value of expr is:425
(nemu) p 0x3D
[src/monitor/debug/expr.c,91,make_token] match rules[1] = "0x[0-9A-Fa-f][0-9A-Fa-f]*" at position 0 with l
en 4: 0x3D
Success record : nr_token = 0, dtype = 258, str = 3D
the value of expr is:61
(nemu) p $eip
[src/monitor/debug/expr.c,91,make_token] match rules[3] = "\$(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)" at position 0 with len 4: $eip
Success record : nr_token = 0, dtype = 259, str = eip
the value of expr is:1048576
(nemu) p $ax
[src/monitor/debug/expr.c,91,make_token] match rules[3] = "\$(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)" at position 0 with len 3: $ax
Success record : nr_token = 0, dtype = 259, str = ax
the value of expr is:13335
```

```
(nemu) p 3+34*1
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 0 with len 1: 3
Success record : nr_token = 0, dtype = 257, str = 3
[src/monitor/debug/expr.c,91,make_token] match rules[9] = "\+" at position 1 with len 1: +
Success record : nr_token = 1, dtype = 43, str =
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 2 with len 2: 34
Success record : nr_token = 2, dtype = 257, str = 34
[src/monitor/debug/expr.c,91,make_token] match rules[11] = "\*" at position 4 with len 1: *
Success record : nr_token = 3, dtype = 42, str =
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 5 with len 1: 1
Success record : nr_token = 4, dtype = 257, str = 1
Operator= *.
Value=34.
Operator= +.
Value=37.
the value of expr is:37
(nemu) p -7-(5+8)
[src/monitor/debug/expr.c,91,make_token] match rules[10] = "-" at position 0 with len 1: -
Success record : nr_token = 0, dtype = 45, str = 3
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 1 with len 1: 7
Success record : nr_token = 1, dtype = 257, str = 7
[src/monitor/debug/expr.c,91,make_token] match rules[10] = "-" at position 2 with len 1: -
Success record : nr_token = 2, dtype = 45, str = 34
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "\(" at position 3 with len 1: (
Success record : nr_token = 3, dtype = 40, str =
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 4 with len 1: 5
Success record : nr_token = 4, dtype = 257, str = 5
[src/monitor/debug/expr.c,91,make_token] match rules[9] = "\+" at position 5 with len 1: +
Success record : nr_token = 5, dtype = 43, str =
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 6 with len 1: 8
Success record : nr_token = 6, dtype = 257, str = 8
[src/monitor/debug/expr.c,91,make_token] match rules[14] = "\)" at position 7 with len 1: )
Success record : nr_token = 7, dtype = 41, str =
Operator= -.
Value=0.
Operator= +.
Value=13.
Operator= -.
Value=-20.
the value of expr is:-20
```

```
(nemu) p -4*3
[src/monitor/debug/expr.c,91,make_token] match rules[10] = "-" at position 0 with len 1: -
Success record : nr_token = 0, dtype = 45, str = 3
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 1 with len 1: 4
Success record : nr_token = 1, dtype = 257, str = 4
[src/monitor/debug/expr.c,91,make_token] match rules[11] = "\*" at position 2 with len 1: *
Success record : nr_token = 2, dtype = 42, str = 34
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 3 with len 1: 3
Success record : nr_token = 3, dtype = 257, str = 3
Operator= -.
Value=0.
Operator= *.
Value=-12.
the value of expr is:-12
(nemu) p $eax*40/4+4
[src/monitor/debug/expr.c,91,make_token] match rules[3] = "\$(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)" at position 0 with len 4: $eax
Success record : nr_token = 0, dtype = 259, str = eax
[src/monitor/debug/expr.c,91,make_token] match rules[11] = "\*" at position 4 with len 1: *
Success record : nr_token = 1, dtype = 42, str = 4
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 5 with len 2: 40
Success record : nr_token = 2, dtype = 257, str = 40
[src/monitor/debug/expr.c,91,make_token] match rules[12] = "\/" at position 7 with len 1: /
Success record : nr_token = 3, dtype = 47, str = 3
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 8 with len 1: 4
Success record : nr_token = 4, dtype = 257, str = 4
[src/monitor/debug/expr.c,91,make_token] match rules[9] = "\+" at position 9 with len 1: +
Success record : nr_token = 5, dtype = 43, str =
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 10 with len 1: 4
Success record : nr_token = 6, dtype = 257, str = 4
adddr=40(0x28)---->value=0(0x00000000)
adddr=0(0x0)---->value=0(0x00000000)
Operator= /.
Value=0.
Operator= +.
Value=4.
the value of expr is:4
```

```
(nemu) p --7
[src/monitor/debug/expr.c,91,make_token] match rules[10] = "-" at position 0 with len 1: -
Success record : nr_token = 0, dtype = 45, str = eax
[src/monitor/debug/expr.c,91,make_token] match rules[10] = "-" at position 1 with len 1: -
Success record : nr_token = 1, dtype = 45, str = 4
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 2 with le
Success record : nr_token = 2, dtype = 257, str = 7
Operator= -.
Value=0.
Operator= -.
Value=0.
the value of expr is:7
(nemu)
```

```
(nemu) p 1+12*3==37
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 0 with len 1: 1
Success record : nr_token = 0, dtype = 257, str = 1
[src/monitor/debug/expr.c,91,make_token] match rules[9] = "\+" at position 1 with len 1: +
Success record : nr_token = 1, dtype = 43, str = 4
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 2 with len 2: 12
Success record : nr_token = 2, dtype = 257, str = 12
[src/monitor/debug/expr.c,91,make_token] match rules[11] = "\*" at position 4 with len 1: *
Success record : nr_token = 3, dtype = 42, str = 3
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 5 with len 1: 3
Success record : nr_token = 4, dtype = 257, str = 3
[src/monitor/debug/expr.c,91,make_token] match rules[4] = "==" at position 6 with len 2: ==
Success record : nr_token = 5, dtype = 260, str =
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 8 with len 2: 37
Success record : nr_token = 6, dtype = 257, str = 37
Operator= *.
Value=36.
Operator= +.
Value=37.
Operator= ==.
Value=1.
the value of expr is:1
```

```
(nemu) p 1+12*3!=37
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 0 with len 1: 1
Success record : nr_token = 0, dtype = 257, str = 1
[src/monitor/debug/expr.c,91,make_token] match rules[9] = "\+" at position 1 with len 1: +
Success record : nr_token = 1, dtype = 43, str = 4
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 2 with len 2: 12
Success record : nr_token = 2, dtype = 257, str = 12
[src/monitor/debug/expr.c,91,make_token] match rules[11] = "\*" at position 4 with len 1: *
Success record : nr_token = 3, dtype = 42, str = 3
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 5 with len 1: 3
Success record : nr_token = 4, dtype = 257, str = 3
[src/monitor/debug/expr.c,91,make_token] match rules[5] = "!=" at position 6 with len 2: !=
Success record : nr_token = 5, dtype = 261, str =
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 8 with len 2: 37
Success record : nr_token = 6, dtype = 257, str = 37
Operator= *.
Value=36.
Operator= +.
Value=37.
Operator= !=.
Value=0.
the value of expr is:0
```

```
(nemu) p 3==3&&(*0x100000==0x1234b8)
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 0 with len 1: 3
Success record : nr_token = 0, dtype = 257, str = 3
[src/monitor/debug/expr.c,91,make_token] match rules[4] = "==" at position 1 with len 2: ==
Success record : nr_token = 1, dtype = 260, str = 4
[src/monitor/debug/expr.c,91,make_token] match rules[2] = "0|[1-9][0-9]*" at position 3 with len 1: 3
Success record : nr_token = 2, dtype = 257, str = 3
[src/monitor/debug/expr.c,91,make_token] match rules[6] = "&&" at position 4 with len 2: &&
Success record : nr_token = 3, dtype = 262, str = 3
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "\(" at position 6 with len 1: (
Success record : nr_token = 4, dtype = 40, str = 1
[src/monitor/debug/expr.c,91,make_token] match rules[11] = "\*" at position 7 with len 1: *
Success record : nr_token = 5, dtype = 42, str =
[src/monitor/debug/expr.c,91,make_token] match rules[1] = "0x[0-9A-Fa-f][0-9A-Fa-f]*" at position 8 with l
en 8: 0x100000
Success record : nr_token = 6, dtype = 258, str = 100000
[src/monitor/debug/expr.c,91,make_token] match rules[4] = "==" at position 16 with len 2: ==
Success record : nr_token = 7, dtype = 260, str =
[src/monitor/debug/expr.c,91,make_token] match rules[1] = "0x[0-9A-Fa-f][0-9A-Fa-f]*" at position 18 with
len 8: 0x1234b8
Success record : nr_token = 8, dtype = 258, str = 1234b8
[src/monitor/debug/expr.c,91,make_token] match rules[14] = "\)" at position 26 with len 1: )
Success record : nr_token = 9, dtype = 41, str =
Operator= ==.
Value=1.
adddr=1048576(0x100000)---->value=1193144(0x001234b8)
Operator= ==.
Value=1.
Operator= &&.
Value=1.
the value of expr is:1
```

```
(nemu) p *0x100000
[src/monitor/debug/expr.c,91,make_token] match rules[11] = "\*" at position 0 with len 1: *
Success record : nr_token = 0, dtype = 42, str = 3
[src/monitor/debug/expr.c,91,make_token] match rules[1] = "0x[0-9A-Fa-f][0-9A-Fa-f]*" at position 1 with l
en 8: 0x100000
Success record : nr_token = 1, dtype = 258, str = 100000
adddr=1048576(0x100000)---->value=1193144(0x001234b8)
the value of expr is:1193144
```

经验证，上述运算结果均正确。

5. 阶段三

5.1 监视点

监视点的作用是监视一个表达式的值何时发生变化。简易调试器允许用户同时设置多个监视点，删除监视点，因此使用链表将监视点的信息组织起来。

为了实现监视点，需要`free_wp`和`free_wp`两个函数

- `free_wp`：从`free_`两次博爱中返回一个空闲的监视点结构
- `free_wp`：将`wp`归还到`free_`链表中

这两个函数作为监视点池的接口被其他函数调用。

5.1.1 分析代码和函数

- 定义监视点结构：修改`nemu/include/monitor/watchpoint.h`

```c
typedef struct watchpoint {
  int NO;   //监视点序号
  struct watchpoint *next;

  /* TODO: Add more members if necessary */
  int old;  //旧的值
  char e[32];  //表达式
  int hitNum;  //记录触发次数
} WP;
```

- 使用池的数据结构进行监视点管理：设定正在使用的监视点（即 head 连接的部分）按顺序编号，若删去某监视点，其余监视点编号保持不变，新定义的监视点不替补删去的监视点编号，而是在 当前最大监视点编号的基础上继续编号。

```
static WP wp_pool[NR_WP];
static WP *head, *free_; //head 用于组织使用中的监视点结构,free_用于组织空闲的监视点结构,init_wp_pool()函数会对两个链表进行了初始化
```

- 全局辅助变量：
  - `used_next`：记录`head`中下一个使用的watchpoint的编号
  - `wptemp`：辅助wp结构
- 全局函数：

```
bool new_wp(char *arg); //新建监视点
bool free_wp(int num);  //删除监视点
void print_wp();        //打印监视点
bool watch_wp();        //监视点值变化
```

其中`new_wp()`从`free_`链表中返回一个空闲的监视点结构，`free_wp()`将`wp`归还到`free_`链表中，这两个函数会作为监视点池的接口被其它函数调用。需要注意的是，调用`new_wp()`时可能会出现没有空闲监视点结构的情况，为了简单起见,此时可以通过 assert(0)马上终止程序。

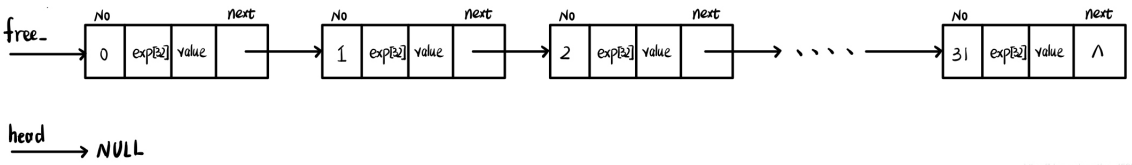
5.1.2 代码实现

首先实现全局变量：

```
static WP wp_pool[NR_WP];
static WP *head, *free_;  //head用于记录监视点链表的头指针，free_用于记录空闲监视点链表的头指针
static int used_next;  //用于记录在head中下一个使用的wp的index
static WP *wptemp;  //辅助wp结构
```

> 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?
>
> 框架代码中定义wp_pool等变量时使用了关键字static，在此处的含义是静态全局变量，该变量只能被本文件中的函数调用，并且是全局变量，而不能被同一程序其他文件中的函数调用。在此处使用static是为了避免它被误修改。
>
> 查阅资料后，我了解到了static有以下作用：
>
> - 在修饰变量的时候，static 修饰的静态局部变量只执行初始化一次，而且延长了局部变量的生命周期，直到程序运行结束以后才释放。
> - static 修饰全局变量的时候，这个全局变量只能在本文件中访问，不能在其它文件中访问，即便是 extern 外部声明也不可以。
> - static 修饰一个函数，则这个函数的只能在本文件中调用，不能被其他文件调用。static 修饰的变量存放在全局数据区的静态变量区，包括全局静态变量和局部静态变量，都在全局数据区分配内存。初始化的时候自动初始化为 0。
> - 不想被释放的时候，可以使用static修饰。比如修饰函数中存放在栈空间的数组。如果不想让这个数组在函数调用结束释放可以使用 static 修饰。
> - 考虑到数据安全性（当程序想要使用全局变量的时候应该先考虑使用 static）。

监视点池初始化会对两个链表 **free_** 和 **head** 进行了初始化，初始化以后的结果如图所示：



代码如下所示：

```
void init_wp_pool() {
  int i;
  for (i = 0; i < NR_WP; i ++) { //初始化wp_pool
    wp_pool[i].NO = i;  //记录索引信息
    wp_pool[i].next = &wp_pool[i + 1]; //链接wp
    wp_pool[i].old = 0; //初始化旧值
    wp_pool[i].hitNum = 0; //初始化命中次数
  }
  wp_pool[NR_WP - 1].next = NULL; //最后一个结点的next为NULL

  head = NULL;
  free_ = wp_pool;
  used_next = 0;
}
```

`new_wp()` 从 `free_` 链表中返回一个空闲的监视点结构。实现 `new_wp` 函数：

```
bool new_wp(char *args) {
  //从free链表中返回一个空闲监视点结构
  if(free_ == NULL) {
    //首先查看free链表是否存在，如果不存在则报错
    assert(0);
  }
  //记录取出的结构并更新链表
  WP* result = free_;
  free_ = free_ -> next;

  //设置新的wp相关信息
  result -> NO = used_next;
  used_next++; //记录索引信息
  result -> next = NULL; //从链表中取出
  strcpy(result -> e, args);
  result -> hitNum = 0; //初始化命中次数
  bool is_success;
  result -> old = expr(result -> e, &is_success);  //计算旧的值
  if(is_success == false) {
    printf("error in new_wp; expression fault!\n");
    return false;
  }

  //对head链表进行更新
  wptemp = head;
  if(wptemp == NULL) { //加入已用链表：如果head为空则直接赋值
    head = result;
  }
  else {
```

```
    while (wptemp -> next != NULL) //找到最后一个结点
    {
      wptemp = wptemp -> next;
    }
    wptemp -> next = result;
  }
  printf("Success: set watchpoint %d, oldvalue = %d\n", result -> NO, result ->
old);
  return true;
}
```

实现 `free_wp` 函数：

free_wp() 函数的参数为WP 类型的指针wp，free_wp() 的作用是将wp所指的结点归还到free_链表中。具体步骤如下：

1. 若 wp = NULL ，则说明输入有误
2. 若 wp = head ，说明wp指向head链表的头结点，只需让head指针指向下一个结点，再将wp所指的结点连到free_链表的第一个位置，并让free_指针指向该节点
3. 若 wp 为其它结点，则需要对head链表进行遍历找出wp所指的结点，再根据②中的步骤，将该结点归还到free_链表中

代码实现如下：

```
//删除监视点
bool free_wp(int num) {
  WP *chosen = NULL; //被选中删除的监视点
  if(head == NULL) { //如果head为空则直接返回
    printf("no watch point now\n");
    return false;
  }
  if(head -> NO == num) { //如果head是要删除的结点
    chosen = head;
    head = head -> next;
  }
  else { //如果head不是要删除的结点
    wptemp = head;
    while (wptemp != NULL && wptemp -> next != NULL) //找到要删除的结点
    {
      /* code */
      if(wptemp -> next -> NO == num) { //找到要删除的结点
        chosen = wptemp -> next;
        wptemp -> next = wptemp -> next -> next;
        break;
      }
      wptemp = wptemp -> next;
    }
  }
  //删除后在free链表中进行添加
  if(chosen != NULL) {
    chosen -> next = free_;
    free_ = chosen;
    return true;
  }
  return false;
```

```
}
```

`print_wp`为打印监视点信息，即调用命令中的info指令，用于查看监视点信息：

```
void print_wp() { //打印监视点信息
  if(head == NULL) { //如果head为空则直接返回
    printf("no watchpoint now\n");
    return;
  }
  printf("watchpoint:\n");
  printf("NO.  expr    hitTimes\n");
  wptemp = head; //从head开始遍历
  while (wptemp != NULL)
  {
    printf("%d  %s    %d\n", wptemp -> NO, wptemp -> e, wptemp -> hitNum);
    wptemp = wptemp ->next;
  }
}
```

`watch_wp`为判断监视点是否触发的辅助函数：

```
bool watch_wp() { //判断监视点是否触发的辅助函数
  bool is_success;
  int result;
  if(head == NULL) { //如果head为空则直接返回
    return true;
  }
  wptemp = head; //从head开始遍历
  while (wptemp != NULL)
  {
    /* code */
    result = expr(wptemp -> e, &is_success);
    if(result != wptemp -> old)
    {
      wptemp -> hitNum += 1;
      printf("Hardware watchpoint %d:%s\n", wptemp -> NO, wptemp -> e);
      printf("Old value:%d\nNew valus:%d\n\n", wptemp -> old, result);
      wptemp -> old = result;
      return false;
    }
    wptemp = wptemp -> next;
  }
  return true;
}
```

监视点的申请和删除：在`nemu/src/monitor/debug/ui.c`中。其中，`cmd_w`为监视点的申请，`cmd_d`为监视点的删除函数。

代码实现如下：

```
static int cmd_w(char *args) { //监视点的申请
  new_wp(args);
  return 0;
}
```

```
static int cmd_d(char* args) { //监视点的删除
  //删除监视点,args为监视点编号
  int num = 0;
  int nRet = sscanf(args, "%d", &num);
  if(nRet <= 0) {
    //解析失败
    printf("args error in cmd_si\n");
    return 0;
  }
  int r = free_wp(num); //删除监视点
  if(r == false) {
    printf("error: no watchpoint %d\n", num);
  }
  else {
    printf("Success delete watchpoint %d\n", num);
  }
  return 0;
}
```

监视点信息查看：补充`cmd_info`函数

```
static int cmd_info(char *args) { //监视点信息查看
  char s;
  if(args == NULL) {
    printf("args error in cmd_info (miss args)\n");
    return 0;
  }

  int temp = sscanf(args, "%c", &s);
  if(temp <= 0) {
    //解析失败
    printf("args error in cmd_info\n");
    return 0;
  }

  if(s == 'w') {
    //打印监视点信息
    print_wp();;
    return 0;
  }

  if(s == 'r') {
    //打印寄存器
    //32bit
    for(int i = 0; i < 8; i++) {
      printf("%s  0x%x\n", regsl[i], reg_l(i));
    }
    printf("eip  0x%x\n", cpu.eip);
    //16bit
    for(int i = 0; i < 8; i++) {
      printf("%s  0x%x\n", regsw[i], reg_w(i));
    }
    //8bit
    for(int i = 0; i < 8; i++)
```

```
    {
      printf("%s  0x%x\n", regsb[i], reg_b(i));
    }
    return 0;
  }

  //如果产生错误
  printf("args error in cmd_info\n");
  return 0;
}
```

修改`nemu/src/monitor/cpu-exec.c`，添加头文件，实现监视点的触发判断

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

#endif
```

5.1.2 运行结果

测试步骤：

- 设置监视点 `$eip==0x100000`
- 查看监视点状态 `hitTimes=0`
- `c`命令继续执行 可以看见触发监视点，程序暂停
- 查看监视点状态 `hitTimes=1`
- 删除监视点
- 查看监视点状态 无监视点
- `c` 命令继续执行 执行完毕，显示 `HIT GOOD TRAP` 信息

```
(nemu) w $eip==0x100000
[src/monitor/debug/expr.c,91,make_token] match rules[3] = "\$(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)" at position 0 with len 4: $eip
Success record : nr_token = 0, dtype = 259, str = eip
[src/monitor/debug/expr.c,91,make_token] match rules[4] = "==" at position 4 with len 2: ==
Success record : nr_token = 1, dtype = 260, str =
[src/monitor/debug/expr.c,91,make_token] match rules[1] = "0x[0-9A-Fa-f][0-9A-Fa-f]*" at position 6 with l
en 8: 0x100000
Success record : nr_token = 2, dtype = 258, str = 100000
Operator= ==.
Value=1.
Success: set watchpoint 0, oldvalue = 1
(nemu) info w
watchpoint:
NO.  expr    hitTimes
0  $eip==0x100000    0
(nemu) c
[src/monitor/debug/expr.c,91,make_token] match rules[3] = "\$(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)" at position 0 with len 4: $eip
Success record : nr_token = 0, dtype = 259, str = eip
[src/monitor/debug/expr.c,91,make_token] match rules[4] = "==" at position 4 with len 2: ==
Success record : nr_token = 1, dtype = 260, str =
[src/monitor/debug/expr.c,91,make_token] match rules[1] = "0x[0-9A-Fa-f][0-9A-Fa-f]*" at position 6 with l
en 8: 0x100000
Success record : nr_token = 2, dtype = 258, str = 100000
Operator= ==.
Value=0.
Hardware watchpoint 0:$eip==0x100000
Old value:1
New valus:0

(nemu) info w
watchpoint:
NO.  expr    hitTimes
0  $eip==0x100000    1
(nemu) d 0
Success delete watchpoint 0
(nemu) info w
no watchpoint now
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026

(nemu)
```

5.2 断点

断点的功能是让程序暂停下来，从而方便查看程序某一时刻的状态。事实上，我们可以很容易地用监视点来模拟断点的功能：`w $eip==ADDR`

其中 ADDR 为设置断点的地址。这样程序执行到 ADDR 的位置时就会暂停下来。

5.2.1 软件断点

在x86体系上实现断电要用到软中断的知识。首先需要分清中断和陷阱的概念：

中断即外中断，指来自处理机和内存外部的中断，包括 I/O 设备发出的 I/O中断、外部信号中断、各种定时器引起的时钟中断以及调试程序中设置的断点等引起的调试中断等。

陷阱即内中断，主要指在处理机和内存内部产生的中断。它包括程序运算引起的各种错误。软中断是通信进程之间用来模拟硬中断的一种信号通信方式。

中断和陷阱的主要区别：

1 、陷阱通常由处理机正在执行的现行指令引起，而中断则是由与现行指令无关的中断源引起的。

2 、陷阱处理程序提供的服务为当前进程所用，而中断处理程序提供的服务则不是为了当前进程的。

3 、CPU 在执行完一条指令之后，下一条指令开始之前响应中断，而在一条指令执行中也可以响应陷阱。

4 、在有的系统中，陷入处理程序被规定在各自的进程上下文中执行，而中断处理程序则在系统上下文中执行。

5.2.2 INT3 指令

x86 系列处理器从其第一代产品英特尔 8086 开始就提供了一条专门用来支持调试的指令，即 INT 3。简单地说，这条指令的目的就是使 CPU 中断（break）到调试器，以供调试者对执行现场进行各种分析。

当我们调试程序时，可以在可能有问题的地方插入一条 INT 3 指令，使 CPU 执行到这一点时停下来。这便是软件调试中经常用到的断点（breakpoint）功能，因此 INT 3 指令又被称为断点指令。

5.2.3 在调试器中设置断点

当我们在调试器中对代码的某一行设置断点时，调试器会先把这里的本来指令的第一个字节保存起来，然后写入一条 INT 3 指令。因为 INT 3 指令的机器码为 11001100b（0xCC），仅有一个字节，所以设置和取消断点时也只需要保存和恢复一个字节，这是设计这条指令时须考虑好的。

5.2.3 断点命中

当 CPU 执行到 INT 3 指令时，由于 INT 3 指令的设计目的就是中断到调试器，因此，CPU 执行这条指令的过程也就是产生断点异常（breakpoint exception，简称#BP）并会保存当前的执行上下文，转去执行异常处理例程的过程。对于 Windows来说，INT 3 异常的处理函数是操作系统的内核函数（KiTrap03）。

内核例程会把这个异常通过调试子系统以调试事件的形式分发给用户模式的调试器，并等待调试器的回复。

在调试器收到调试事件后，它会根据调试事件数据结构中的程序指针得到断点异常的发生位置，然后在自己内部的断点列表中寻找与其匹配的断点记录。如果能找到，则说明这是“自己”设置的断点，执行一系列准备动作后，便允许用户进行交互式调试。如果找不到，就说明导致这个异常的 INT 3 指令不是调试器动态替换进去的，因此会显示的对话框，意思是说一个“用户”插入的断点被触发了。

在调试器下，我们是看不到动态替换到程序中的 INT 3 指令的。大多数调试器的做法是在被调试程序中断到调试器时，会先将所有断点位置被替换为INT 3 的指令恢复成原来的指令，然后再把控制权交给用户。

5.2.4 恢复执行

当用户结束分析希望恢复被调试程序执行时，调试器通过调试 API 通知调试子系统，这会导致系统内核的异常分发函数返回到异常处理例程，然后异常处理例程通过IRET/IRETD 指令触发一个异常返回动作，使 CPU 恢复执行上下文，从发生异常的位置继续执行。

5.2.5 问题

> **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?

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INT 3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。

断点机制不能正常工作。因为INT3断点将被调试进程中对应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。

举个例子：

```
.. some code ..
    jz foo
    dec eax
foo:
    call bar
    .. some code ..
```

假设我们要在`dec eax`上设定断点。这恰好是条单字节指令（操作码是0×48）。如果替换为断点的指令长度超过1字节，我们就被迫改写了接下来的下一条指令（`call`），这可能会产生一些完全非法的行为。考虑一下条件分支`jz foo`，这时进程可能不会在`dec eax`处停止下来（我们在此设定的断点，改写了原来的指令），而是直接执行了后面的非法指令。

通过对`int 3`指令采用一个特殊的单字节编码就能解决这个问题。因为x86架构上指令最短的长度就是1字节，这样我们可以保证只有我们希望停止的那条指令被修改。

> **“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。

当将断点设置在指令的非首字节时，调试器将在该指令执行到该字节时停止程序的执行。

在这种情况下，程序可能已经执行了指令的前面部分，因此断点停止时，程序状态可能已经改变。这可能会导致调试器中断的上下文与程序实际执行的上下文不同，从而使得调试器中的变量值或调用栈可能与程序实际的状态不同。

此外，在某些情况下，将断点设置在指令的非首字节可能会导致程序在调试器停止时发生异常或崩溃，因为调试器可能会破坏指令的完整性或干扰指令的执行。

我写了一个汇编程序以完成非首字节断点的测试：

```
section .data
    message db "This is a test message", 0

section .text
    global _start

_start:
    ; 打印消息
    mov eax, 4
    mov ebx, 1
    mov ecx, message
    mov edx, 23
    int 0x80

    ; 在这里设置断点
    nop ; 用 nop 填充中间位置

    ; 执行一些无害指令
    nop
    nop
    nop

    ; 程序退出
    mov eax, 1
    xor ebx, ebx
```

```
    int 0x80
```

首先，编译、链接并运行：

```
myrrolinz@Myrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign$ nasm -f elf64 test.asm
myrrolinz@Myrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign$ ld -s -o test test.o
myrrolinz@Myrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign$ ./test
This is a test messagemyrrolinz@Myrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign$ gdb
```

可以看到程序会输出"This is a test message"

然后执行`objdump -d`，查看每条指令对应的位置：

```
myrrolinz@Myrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign$ objdump -d test

test:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <.text>:
  401000:       b8 04 00 00 00          mov    $0x4,%eax
  401005:       bb 01 00 00 00          mov    $0x1,%ebx
  40100a:       b9 00 20 40 00          mov    $0x402000,%ecx
  40100f:       ba 17 00 00 00          mov    $0x17,%edx
  401014:       cd 80                   int    $0x80
  401016:       90                      nop
  401017:       90                      nop
  401018:       90                      nop
  401019:       90                      nop
  40101a:       b8 01 00 00 00          mov    $0x1,%eax
  40101f:       31 db                   xor    %ebx,%ebx
  401021:       cd 80                   int    $0x80
```

可以看到`cd 80`词条语句的位置是`0x401014`，会输出"This is a test message"

进行gdb调试：

首先将断点设置在`0x401016`看看会发生什么：

```
(gdb) break *0x401016
Breakpoint 1 at 0x401016
(gdb) r
Starting program: /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/test
This is a test message
Breakpoint 1, 0x0000000000401016 in ?? ()
```

可以看到程序能够输出该语句。再尝试将断点设置在`0x401015`：

```
(gdb) break *0x401015
Breakpoint 2 at 0x401015
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/test

Program received signal SIGSEGV, Segmentation fault.
0x0000000000401014 in ?? ()
(gdb)
```

此时就产生了段错误，dbg无法继续运行。

根据汇编语言代码的注释可知，调用 int 80 进行字符串打印时，edx 存放的是字符串长度，如果尝试在该长度所在位置`0x401010`处设置断点，由于 0x17被 0xcc 代替，将输出其余多个字符：

```
(gdb) break *0x401011
Breakpoint 1 at 0x401011
(gdb) r
Starting program: /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/test
This is a test message.shstrtab.text.data
                                        @# @  [Inferior 1 (process 4959) exited normally]
(gdb)
```

总结：

- 如果`0xcc`在首字节处，读取指令时会被当做操作码，程序运行到此会产生`int 3`软中断
- 如果`0xcc`在非首字节处，它有可能被视为指令的操作数，解析出错误指令，无法继续运行

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

模拟器(Emulator)和调试器(Debugger)是两个不同的概念。

模拟器是一种软件，可以模拟运行另一种计算机系统的程序，例如在 x86 架构的计算机上运行 ARM 架构的程序。模拟器通过模拟另一种计算机系统的指令集、内存、输入/输出等硬件设备，来执行另一种计算机系统的程序。

调试器是一种工具，用于帮助程序员调试程序。调试器可以启动程序、暂停程序、单步执行程序、查看内存和寄存器的值等，以帮助程序员发现和修复程序中的错误。

NEMU是一个模拟器，它可以模拟运行一个类似于x86的指令集，并提供了调试功能。NEMU可以通过命令行界面或者图形界面来启动程序、单步执行程序、查看内存和寄存器的值等，以帮助程序员调试程序。

而GDB是一个调试器，它并不是模拟器，它不能够模拟运行其他指令集的程序。GDB是用于调试本地和远程程序的工具，它可以与程序交互，并提供了启动程序、暂停程序、单步执行程序、查看内存和寄存器的值等功能，以帮助程序员发现和修复程序中的错误。

具体在调试方面，NEMU的简易调试器是通过`ui_mainloop`获取相关命令后，执行对应程序并输出相关信息的。而在GDB中，调试是通过`ptrace`系统调用进行实现。在使用参数 ptrace 系统调用建立调试关系后，交付给目标程序的任何信号首先都会被 gdb 截获。 因此 gdb 可以先行对 信号进行相应处理，并根据信号的属性决定是否要将信号交付给目标程序。

在 Linux 系统中，GDB 使用的主要系统调用有：

- ptrace：用于控制被调试进程的执行，包括读写进程的内存、设置和删除断点等操作。
- waitpid：用于等待被调试进程结束，并获取其退出状态。
- execve：用于在新的进程空间中运行被调试程序。
- kill：用于向进程发送信号，包括 SIGTRAP 信号。

总的来说，GDB 的调试原理是通过在被调试程序中插入调试信息，以及使用系统调用来实现对被调试程序的控制和监视。

6. 必答题

6.1 i386手册

6.1.1 EFLAGS 寄存器中的 CF 位是什么意思?

EFLAGS 寄存器中的 CF 位是进位标志位，位于EFLAGS的第0位，用于标记在执行加法、减法、乘法或移位指令时是否有进位或借位发生。如果进位或借位发生，CF 置为 1，否则 CF 置为 0。

例如，在执行 add 指令时，如果两个操作数相加的结果超过了目标操作数的位数，就会发生进位，此时 CF 置为 1。在执行 sub 指令时，如果被减数小于减数，就会发生借位，此时 CF 置为 1。

在i386手册中，相关介绍如下：

i386-2.3Register(P33)

2.3.4 Flags Register

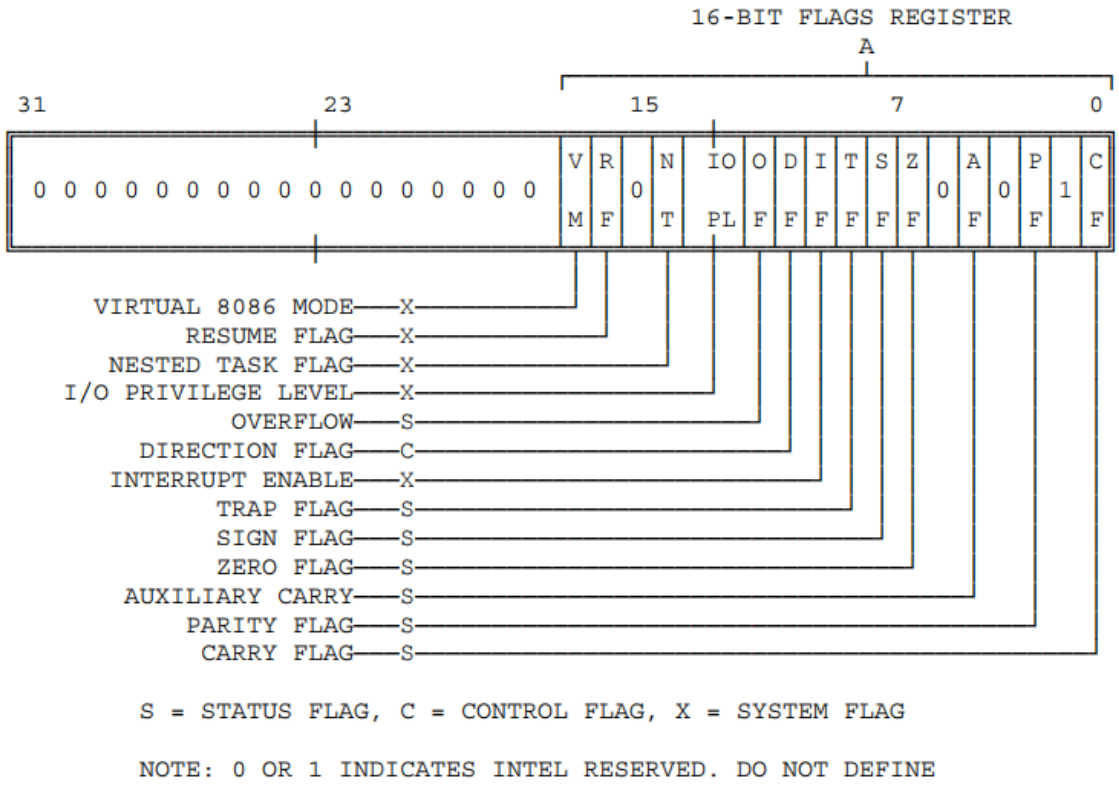
The flags register is a 32-bit register named EFLAGS. Figure 2-8 defines the bits within this register. The flags control certain operations and indicate the status of the 80386.

The low-order 16 bits of EFLAGS is named FLAGS and can be treated as a unit. This feature is useful when executing 8086 and 80286 code, because this part of EFLAGS is identical to the FLAGS register of the 8086 and the 80286.

The flags may be considered in three groups: the status flags, the control flags, and the systems flags. Discussion of the systems flags is delayed until Part II.

i386-附录 C(P419)

Figure 2-8. EFLAGS Register



6.1.2 ModR/M 字节是什么?

ModR/M相关内容在17.2节的INSTRUCTION FORMAT中。

ModR/M 字节是用于传递指令操作数的一种编码方式。它由三个部分组成：Mod、Reg/Opcode、R/M。其中：

- Mod (2 bits)：表示指令中的第二个操作数的寻址方式。Mod 字段的值可以是 00、01、10 或 11，分别对应不同的寻址方式。
- Reg/Opcode (3 bits)：当指令为 ALU 操作码时，表示操作码的低三位；当指令为寄存器操作时，表示寄存器编号。
- R/M (3 bits)：表示指令中的第一个操作数的寻址方式。

ModR/M 字节一般紧跟着操作码字节，在大多数情况下用于标识一个操作数，以及操作数所在的寄存器或存储器地址。在使用 ModR/M 编码时，指令的第一个操作数与第二个操作数的位置是固定的，即第一个操作数位于 ModR/M 字节中的 R/M 字段所表示的寄存器或存储器中，而第二个操作数的位置则由 Mod 字段所表示的寻址方式决定。

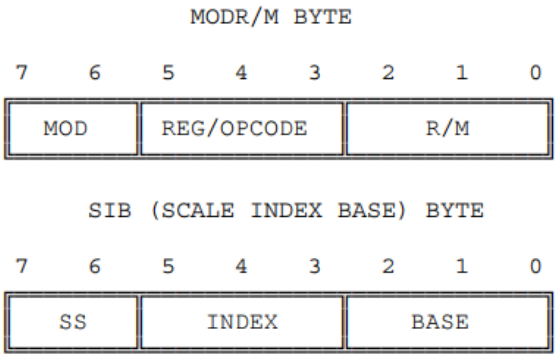
在指令编码过程中，根据不同的操作数类型和操作数寻址方式，ModR/M 字节会被不同的值所代替，这些值与不同的寻址方式和寄存器组合有关。

更具体来说：

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

具体在i386手册中如下：

Figure 17-2. ModR/M and SIB Byte Formats



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

- 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

6.1.3 mov 指令的具体格式是怎么样的?

mov指令相关内容在17.2.11节的 Instruction Set Detail中:

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
Ciiiiii	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.

Opcode	Instruction	Clocks	Description
0F 20 /r	MOV r32,CR0/CR2/CR3	6	Move (control register) to (register)
0F 22 /r	MOV CR0/CR2/CR3,r32	10/4/5	Move (register) to (control register)
0F 21 /r	MOV r32,DR0 -- 3	22	Move (debug register) to (register)
0F 21 /r	MOV r32,DR6/DR7	14	Move (debug register) to (register)
0F 23 /r	MOV DR0 -- 3,r32	22	Move (register) to (debug register)
0F 23 /r	MOV DR6/DR7,r32	16	Move (register) to (debug register)
0F 24 /r	MOV r32,TR6/TR7	12	Move (test register) to (register)
0F 26 /r	MOV TR6/TR7,r32	12	Move (register) to (test register)

Opcode	Instruction	Clocks	Description
A4	MOVS m8,m8	7	Move byte [(E)SI] to ES:[(E)DI]
A5	MOVS m16,m16	7	Move word [(E)SI] to ES:[(E)DI]
A5	MOVS m32,m32	7	Move dword [(E)SI] to ES:[(E)DI]
A4	MOVSB	7	Move byte DS:[(E)SI] to ES:[(E)DI]
A5	MOVSW	7	Move word DS:[(E)SI] to ES:[(E)DI]
A5	MOVSD	7	Move dword DS:[(E)SI] to ES:[(E)DI]

其中，`/r`表示 ModR/M 字节，`r/m8`、`r/m16`、`r/m32`、`r/m64`分别表示 8、16、32、64 位寄存器或内存地址，`r8`、`r16`、`r32`、`r64`分别表示 8、16、32、64 位寄存器，`imm8`、`imm16`、`imm32`、`imm64`分别表示 8、16、32、64 位立即数。

There are also variants of MOV that operate on segment registers. These are covered in a latersection of this chapter.:

- To a register from memory
- To memory from a register
- Between general registers
- Immediate data to a register
- Immediate data to a memory

参考 i386-17.3(P345)

6.2 问题2

> shell 命令完成 PA1 的内容之后,nemu/目录下的所有.c 和.h 和文件总共有多少行代码?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?

1. 行代码统计：在`nemu`在运行`find . -name "*.[ch]" |xargs cat|wc -l`

```
myrrolinz@Myrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu$ find . -name "*.[ch]"
  |xargs cat|wc -l
3991
```

2. 计算PA1编写的代码：只需要切换到PA0进行查看，然后再进行对比即可。将其写入makefile：

```
myrrolinz@Myrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu$ make count
git checkout pa0
M       nemu/Makefile
Switched to branch 'pa0'
Your branch is up to date with 'origin/pa0'.
find ./ -name "*.[ch]" |xargs cat|wc -l
3487
git checkout pa1
M       nemu/Makefile
Switched to branch 'pa1'
Your branch is ahead of 'origin/pa1' by 2 commits.
  (use "git push" to publish your local commits)
find ./ -name "*.[ch]" |xargs cat|wc -l
3991
```

   新增代码行数=3991-3487=504

3. 计算.c .h文件除去空格有多少行：

```
myrrolinz@Myrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu$ find . -name "*[.cpp|
  .h]" | xargs grep "^."| wc -l
grep: .: Is a directory
grep: ./build/obj/cpu/exec: Is a directory
grep: ./build/obj/misc: Is a directory
grep: ./src: Is a directory
grep: ./src/cpu/exec: Is a directory
grep: ./src/misc: Is a directory
3341
```

6.3 问题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` 是 GCC 的编译选项之一，它开启了许多警告选项，让编译器输出更多的警告信息，帮助程序员发现可能存在的问题。例如未声明的函数、不兼容指针类型的赋值、未使用的变量等等。

`-Werror` 是 GCC 的编译选项之一，它把编译器的所有警告都当成错误，也就是说一旦出现任何警告，就会停止编译并返回错误码。这样做可以确保程序员在编写代码时一定要仔细检查和处理所有的警告，从而避免潜在的问题。

使用 `-Wall` 和 `-Werror` 的主要目的是提高代码的质量和可靠性，防止因为一些低级错误而导致程序崩溃或者出现意料之外的行为。在实际开发中，建议使用这两个选项来编译代码，从而提高程序的稳定性和可维护性。

7. 实验感悟与遇到的问题

第一次入门PA还是感觉挺困难的，毕竟万事开头难，写代码和写报告都花了很多的时间，之前想图方便用虚拟机做，但是虚拟机的龟速让我忍无可忍，最终还是换了wsl:)至于图形化界面当时做ucore就已经踩了一次坑，可以用 `VcXsrv` + `xfce4` 来解决。

在"随心所欲"的断点部分，由于是我对汇编语言不太熟，光是把它调试正确就花了一些时间。我之前不知道语句会被加载到 `$edx` 里，也是费了很大功夫才弄清机器码具体每句话在说什么，最终把断点设在了我想要的位置。虽然折磨，但是这也加深了我对汇编的理解，以后编写汇编会更加顺手。

总之，PA1是一个开头难的过程，但积累了宝贵的经验，希望后续的PA能顺利进行。