南开大学
Nankai University

南　　开　　大　　学

计 算 机 学 院

PA1-最简单的计算机实验报告

蒋薇

年级：2021 级

专业：计算机科学与技术

2024 年 3 月 15 日

目录

一、 概述

(一) 实验目的

- **实现一个简易调试器**
- **完成基础表达式求值**
- **理解并完成监视点**

(二) 实验内容

1. stage0: 理解 nemu

学习 nemu 基本原理，使用匿名 union 实现正确的寄存器结构体

2. stage1: 实现基础解析命令

- **cmd_si**: 单步调试，无参数时默认执行一步；接受参数 n，则执行 n 步
- **cmd_info**: 打印寄存器或监视点信息
- **cmd_x**: 扫描内存，接收参数 n 和 addr，打印 addr 开始的 n 个字节内存

3. stage2: 实现表达式求值

- **cmd_p**: 表达式求值
- token 、make_token：识别 token，生成 tokens 数组
- check_parentheses：检查括号
- eval：递归求值

添加关系运算、逻辑运算、识别负号等。

4. stage3: 实现监视点的功能

- 完善 wp 结构体 - 实现 new_wp 和 free_wp，插入或删除监视点 - 在 cpu_exec 中检查监视点值的变化，并停止

二、 stage1

(一) 简单计算机模型

1. NEMU 执行流程

模块划分

NEMU 主要由 4 个模块构成:monitor,CPU,memory,device

- monitor: 调试器的必须部分，负责与 GNU/Linux 交互，也提供了方便的用户调试功能
- cpu: 译码、执行逻辑和寄存器结构体定义，以及相关的 rtl 指令和中断处理逻辑
- memory: 以数组形式表示内存，提供接口函数
- device: 设备相关

执行流程

NEMU 开始执行的时候, 首先会调用 init_monitor() 函数进行一些和 monitor 相关的初始化工作,reg_test() 函数会生成一些随机的数据, 对寄存器实现的正确性进行测试. 若不正确, 将会触发 assertion fail.

然后通过调用 load_img() 函数读入带有客户程序的镜像文件

然后调用 restart() 函数, 它模拟了” 计算机启动” 的功能

在 monitor 的其它初始化后, 最后通过调用 welcome() 函数输出欢迎信息和 NEMU 的编译时间.

monitor 的初始化工作结束后,NEMU 会进入用户界面主循环 ui_mainloop(), 输出 NEMU 的命令提示符:(nemu)

输入 c 之后,NEMU 开始进入指令执行的主循环 cpu_exec(),cpu_exec() 模拟了 CPU 的工作方式: 不断执行指令

exec_wrapper() 函数的功能让 CPU 执行当前%eip 指向的一条指令, 然后更新%eip

exec_real 是真正执行一条指令的函数，包括取值、查 opcode 表、译码执行

2. 代码：实现正确的寄存体结构体

问题：无法通过 reg_test，这是因为在 x86 中，不同位数的寄存器彼此之间可能共用一个寄存器；而 struct 则是用不同的内存空间模拟 x86 的寄存器，当其中一个寄存器的值发生变化，其他相关者不能同步变化，发生了错误

寄存器结构修改，使用匿名 union

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

我们想要让 rtlreg_t eax 对应 gpr 数组的第一个元素：gpr[0]，同时，_32、_16、_8 共用一个 32bit 的空间；所以 gpr[8] 和 rtlreg_t 需要联合，gpr[8] 内部也需要联合，这是一个嵌套的 union 结构

除了嵌套 union，还需要把 8 个 rtl 寄存器用 struct 包括起来，实现“gpr[0] 和 eax、gpr[1] 和 ecx.... 共用内存，否则成“gpr[8] 和 eax、gpr[8] 和 ecx.... 共用内存”。

3. 问题：究竟要执行多久

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. 如果你查阅 i386 手册, 你会发现 x86 中并没有这条指令, 它是为了在 NEMU 中让客户程序指示执行的结束而加入的.

- 当你看到 NEMU 输出以下内容时:nemu: HIT GOOD TRAP at eip = 0x00100026 说明客户程序已经成功地结束运行.

退出 cpu_exec() 之后,NEMU 将返回到 ui_mainloop(), 等待用户输入命令.

4. 问题：谁来指示程序的结束

通常，我们的 main 函数以 return 0 结束，这与调用 exit(0) 等价；status 为 0 是正常终止，否则是异常终止

exit() 会调用 _exit() 进入内核，二者区别是，exit() 在进入内核之前会做一些清理工作

- 调用 fclose 刷新缓冲区，把数据写入磁盘文件
- 调用终止函数，例如 atexit() 注册的函数
- 陷入内核，执行 _exit()

atexit() 用于注册终止函数，参数是一个函数指针，函数指针指向一个没有参数也没有返回值的函数。

main() 函数结束后，这些被注册的函数仍然会执行，也就意味着，我们的程序还没有结束

程序的终点，应该是 _exit() 结束，调用它，进程立即终止，而不是 main 函数中的 return 0。

(二) 基础设施: 简易调试器

1. 代码：实现单步执行、打印寄存器、扫描内存

单步执行

单步执行

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

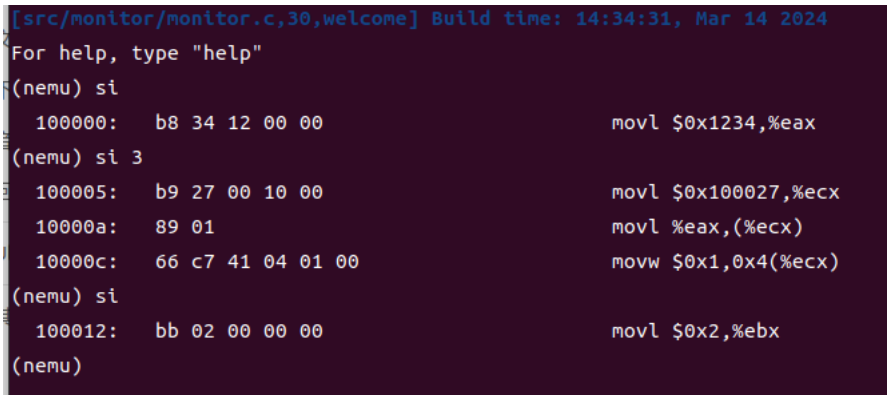

图 1: si

si 不带参数默认是单步执行，而带参数 n 则是执行 n 步

打印寄存器

打印寄存器 info r

```
    static int cmd_info(char *args)
{
...
    if (strcmp(arg, "r") == 0)
    {
      printf("\t%-16s\t%-16s\t%-16s\n", "REG", "HEX", "DEC");
      int i = 0;
      for (; i < 8; i++)
      {
        printf("\t%-16s\t%-16x\t%-16d\n", reg_name(i, 4), reg_l(i), reg_l(i))
            ;
      }
      for (i = 0; i < 8; i++)
      {
        printf("\t%-16s\t%-16x\t%-16d\n", reg_name(i, 2), reg_w(i), reg_w(i))
            ;
      }
      for (i = 0; i < 8; i++)
      {
        printf("\t%-16s\t%-16x\t%-16d\n", reg_name(i, 1), reg_b(i), reg_b(i))
            ;
      }
      printf("\t%-16s\t%-16x\t%-16d\n", "eip", cpu.eip, cpu.eip);
    }
    ...
}
```

需要读寄存器的值，使用如下函数

- reg_name() //获取寄存器名字
- reg_l() //获取寄存器 32bit
- reg_w() //获取寄存器低 16bit
- reg_b() //获取寄存器高、低 8bit

```
(nemu) info r
Get command: info r
        REG             HEX             DEC
        eax             1234            4660
        ecx             100027          1048615
        edx             53c0d95c        1405147484
        ebx             2               2
        esp             491065e9        1225811433
        ebp             7924c686        2032453254
        esi             369735e6        915879398
        edi             1efb20a3        519774371
        ax              1234            4660
        cx              27              39
        dx              d95c            55644
        bx              2               2
        sp              65e9            26089
        bp              c686            50822
        si              35e6            13798
```

图 2: info r

使用格式化输出 printf，可以分别打印十六进制和十进制数

扫描内存

扫描内存 x

```
   static int cmd_x(char *args)
{
  int N = atoi(strtok(NULL, " "));
char* expr = strtok(NULL, " "); //获取两个命令行参数
paddr_t addr;
sscanf(expr,"%x",&addr); //将输入的地址转换为十进制，便于参数传入到
paddr_read
int i = 0;
for(;i < N;i++){
if(i % 4 == 0){
printf("\naddr 0x%x:\n", addr + 4 * i);
}
printf("\t%x", paddr_read(addr+i,1)); //使用paddr_read循环读取
}
printf("\n");
return 0;
}
```

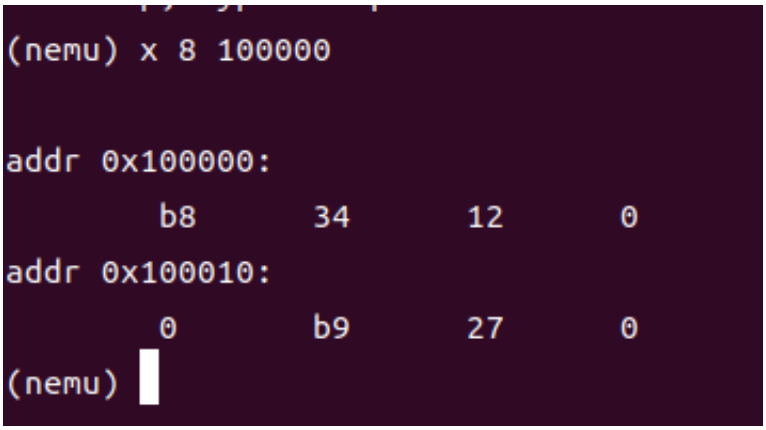

图 3: w

根据输入的两个参数 N 和 addr，打印从 addr 开始的 N 个字节的内存信息

三、 stage2

(一) 词法分析

为算术表达式中的各种 token 类型添加规则, 注意 C 语言字符串中转义字符的存在和正则表达式中元字符的功能

成功识别出 token 后, 将 token 的信息依次记录到 tokens 数组中

1. 代码：实现算术表达式的词法分析

补充词法分析

```
    rules[] = {

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

  {"␣+", TK_NOTYPE},    // spaces
   {"(0[xX])(0|([A-Fa-f1-9][A-Fa-f0-9]*))", TK_HEX},
  {"0|([1-9][0-9]*)", TK_INTEGER},
  {"\\$[a-z]+", TK_REG},
  {"[a-zA-Z1-9][a-zA-Z0-9]*", TK_VAR},
  {"\\(", '('},
  {"\\)", ')'},
  {"\\+", '+'},         // plus
   {"-", '-'},
  {"\\*", '*'},
  {"/", '/'},
};
```

- 能识别区分十进制整数、十六进制整数 (无论是否带 0x/X 前缀)

- 能识别寄存器，寄存器需以 $ 开头
- 能识别算术运算符、逻辑运算符、关系运算符
- 能识别变量名、空格

记录 tokens 数组

```
        case TK_INTEGER: tokens[nr_token].type=TK_INTEGER;break;
        case TK_HEX: tokens[nr_token].type=TK_HEX;break;
        case TK_REG: tokens[nr_token].type=TK_REG;break;
        case TK_VAR: tokens[nr_token].type=TK_VAR;break;
        case '+': tokens[nr_token].type='+';break;
        case '-': tokens[nr_token].type='-';break;
        case '*': tokens[nr_token].type='*';break;
        case '/': tokens[nr_token].type='/';break;
        case '!': tokens[nr_token].type='!';break;
        case '(': tokens[nr_token].type='(';break;
        case ')': tokens[nr_token].type=')';break;
        default: panic("make token: no such rule!\n"); break;
};
```

(二) 表达式求值

递归求值，BNF，长表达式是由短表达式构成的，我们就先对短表达式求值，然后再对长表达式求值。为了在 token 表达式中指示一个子表达式，我们可以使用两个整数 p 和 q 来指示这个子表达式的开始位置和结束位置.

1. 代码：实现算术表达式的递归求值

由给出的代码框架:

框架注释

```
    int eval(int p, int q){
if(p > q){
// a.bad expr:因为正确执行时不可能得到p>q的情况
}
else if(p == q){
// b.表明要对一个token求值，可以是十进制整数、十六进制整数、寄存器、变量
}
else if(checkparentheses(p, q)){
// c.去掉两端括号
return eval(p+1, q-1);
}
else{
// d.大部分情况的非token表达式，需要进一步拆分，形式如5+3，5-(3+2)，5+2*3
}
return 0;
}
```

expr() 调用 eval()，进行递归求值，eval 传入参数 p 和 q 是 expr 的下标，所以 eval 实现的关键在于如何找到优先级最低的运算符，从而拆分为 p index-1，index+1 q 两部分，再递归调用 eval()

p = q

p

```
if(tokens[p].type == TK_INTEGER){ // 十进制
return atoi(tokens[p].str);
}
else if(tokens[p].type == TK_HEX){ // 十六进制
int v = 0;
sscanf(tokens[p].str, "%x", &v);
return v;
}
else if(tokens[p].type == TK_REG){ // 寄存器
char reg[4];
sscanf(tokens[p].str, "$%s", reg);
int v = getRegValue(reg, strlen(reg));
return v;
}
return 0;
```

对于 10 进制的 INTEGER 和 16 进制的 HEX，分别用 atoi 和 sscanf 读入其 value，并返回即可

对于寄存器，用 ssacnf 读取寄存器名字，调用 getRegValue() 函数，参数是名字和长度

p

```
if(len == 3){
if(!strcmp(reg, "eax")){
return cpu.eax;
}
else if(!strcmp(reg, "ecx")){
//...
}
}
else if(len == 2){
if(!strcmp(reg, "ax")){
return cpu.eax & 0xffff;
}
// ...
else if(!strcmp(reg, "al")){
return reg_b(R_AL);
}
}
```

strcmp 比较字符串，结合全局变量 cpu 和 reg_b 等宏来读取寄存器的值，并返回

checkparentheses

识别从 p 到 q 的表达式子串是否被一对括号完整包含

用 cnt 计数，左括号 cnt++，右括号 cnt–，当出现如下情况时返回 false:
- p 和 q 的位置不是一对括号
- 还没有遍历到 q 时，左括号和右括号的数量相等了，即 cnt 减为 0
- 结束时 cnt 不为 0

p

```
bool checkparentheses(int p, int q){
int i,cnt = 0;
if(tokens[p].type != '(' || tokens[q].type != ')')
return false;
for(i = p; i <= q; i++){
if(tokens[p].type == '(')
cnt++;
else if(tokens[q].type == ')')
cnt--;
if(cnt == 0 && i<q)
return false;
}
if(cnt < 0) return false;
return true;
}
```

寻找 dominant operator

p

```
for(int i = p; i <= q; i++){
int c = tokens[i].type;
if(c == '(') lp++; // int lp, rp;
else if(c == ')') rp++;
if(isOp(c) && (lp == rp)){ //只有括号都匹配，即当前遍历的c不在一对括号中时，
    才更新
curOp
if(getOpPrior(curOp) >= getOpPrior(c)){
curOp = tokens[i].type; // curOp就是当前的dominant operator
index = i;
}
}
}

//找到dominant operator和它的index后，可以切分子串、递归调用eval求值

int vl = eval(p, index - 1);
int vr = eval(index + 1, q);
switch(curOp){
case '+':return vl + vr;break;
case '*':return vl * vr;break;
// ... 省略了其他运算符
default:return 0;break;
```

```
}
```

2. 代码：实现带负数的算术表达式求值

在 token 识别时，无法区分 *、-等符号的不同含义，可以在形成 tokens 数组后，观察相邻 token 来区分

区分乘号、解引用；区分减号、负号

如果当前 token 前一个 token 是操作符，那么此 token 就不可能是正常的乘号或减号，而是解引用或负号

p

```
for(int i = 0; i < nr_token; i++){
if(tokens[i].type == '*' && (i == 0 || isOp(tokens[i-1].type))){
tokens[i].type = TK_DRE; //这是一个解引用
}
if(tokens[i].type == '-' && (i == 0 || isOp(tokens[i-1].type))){
tokens[i].type = TK_NEG; //这是一个负号
}
}

if(curOp == TK_DRE){
paddr_t addr = 0;
sscanf(tokens[p+1].str, "%x", &addr);
uint32_t v = paddr_read(addr, 4); //解引用，调用paddr_read读内存
return v;
}
else if(curOp == TK_NEG){
int v = 0;
sscanf(tokens[p+1].str, "%d", &v); // 负号，返回负值
return -v;
}
else if(curOp == '!'){
if(eval(p+1,p+1)){ //判断！之后的token的值，如果非零就return 0，代表false
return 0; //如果是0，就返回1，代表true
}
else{
return 1;
}
}
```

3. 代码: 实现更复杂的表达式求值

添加关系运算符、逻辑运算符

添加关系运算符、逻辑运算符

```
    {"==", TK_EQ},          // equal
  {"!=", TK_NE},
```

```
  {"\\&\\&", TK_AND},
  {"\\|\\|", TK_OR},
  {"!", '!'}

  case TK_EQ: tokens[nr_token].type=TK_EQ;break;
        case TK_NE: tokens[nr_token].type=TK_NE;break;
         case TK_AND: tokens[nr_token].type=TK_AND;break;
         case TK_OR: tokens[nr_token].type=TK_OR;break;
```

```
(nemu) p 2 * (3 --2)
[src/monitor/debug/expr.c,88,make_token] match rules[2] = "0|([1-9][0-9]*)" at position 0 with len 1: 2
[src/monitor/debug/expr.c,88,make_token] match rules[0] = " +" at position 1 with len 1:
[src/monitor/debug/expr.c,88,make_token] match rules[9] = "\*" at position 2 with len 1: *
[src/monitor/debug/expr.c,88,make_token] match rules[0] = " +" at position 3 with len 1:
[src/monitor/debug/expr.c,88,make_token] match rules[5] = "\(" at position 4 with len 1: (
[src/monitor/debug/expr.c,88,make_token] match rules[2] = "0|([1-9][0-9]*)" at position 5 with len 1: 3
[src/monitor/debug/expr.c,88,make_token] match rules[0] = " +" at position 6 with len 1:
[src/monitor/debug/expr.c,88,make_token] match rules[8] = "-" at position 7 with len 1: -
[src/monitor/debug/expr.c,88,make_token] match rules[8] = "-" at position 8 with len 1: -
[src/monitor/debug/expr.c,88,make_token] match rules[2] = "0|([1-9][0-9]*)" at position 9 with len 1: 2
[src/monitor/debug/expr.c,88,make_token] match rules[6] = "\)" at position 10 with len 1: )
i:5, type=TK_NEG
start:2, end:), tokens:8
Op: *
start:2, end:2, tokens:1
INTEGER: 2
start:(, end:), tokens:6
start:3, end:2, tokens:4
Op: -
start:3, end:3, tokens:1
INTEGER: 3
start:-, end:2, tokens:2
Op: -(NEG)
cmd_p:val is 10
(nemu)
```

图 4: 表达式求值

四、 stage3

(一) 监视点

WP 结构体

WP 结构体

```
    typedef struct watchpoint {
int NO;
struct watchpoint *next;
int value;
char str[20]; // user's watchpoint input
int type; // type & 1 == 0: not used; else used;
// type & 2 == 0: reg; else number;
} WP;

bool wp_exists(int type){
```

```
return type & 1; // 是否存在
}

bool wp_reg(int type){
return type & 2; // 是否是单独的寄存器
}
```

添加了 value、str、type 三个域，作用如下：

- value：记录当前监视点的计算值，如果发生改变可以通过比较新旧值得知
- str：记录用户输入的表达式，用于 info w 打印
- type：类型，区分不同类型、不同状态的监视点，32bit 可以设计 32 种类型，每一个 bit 代表一种；目前只有两种：用最后 1bit 标识是否在使用，倒数第 2 个 bit 标识当前监视点是否是一个单独的寄存器

1. 代码：实现监视点池的管理

监视点链表实质上是一个数组, 四个变量，分别指向非空闲链表头尾、空闲链表头尾

监视点链表结构

```
    static WP wp_pool[NR_WP];
    static WP *head, *free_head, *tail, *free_tail;
```

用户申请监视点，总是能从空闲池中申请到序号较小的监视点

- 非空闲链表头，指向的一定是未被使用且序号最小的监视点，便于快速申请
- new_wp 时，分配序号最小监视点
- free_wp 时，释放对应序号的监视点到对应数组位置，并且检查能否更新链表头

2. 问题：static 的使用

wp_pool 变量定义时，使用了关键字 static

static 含义是静态全局变量，被 static 修饰的变量只能在本文件中作为全局变量使用，同一程序的其他文件不能使用

这样做提高了此变量的隔离性、安全性，避免被其他文件中的指令误修改

3. 代码：实现监视点

监视点链表 $\text{new}_wp()$

```
  WP* new_wp(char* str, int value){
if(WP_COUNT == NR_WP){ // 没有空闲监视点
panic("cannot set new watchpoint: full!\n");
return NULL;
}
tail->next = free_head; // 这里tail就是已分配的监视点的尾，new_wp需要在尾部追
    加
一个新
tail = tail->next; // 的监视点；所以tail要后移动
strcpy(tail->str, str); // 设置WP的str、value、type
tail->value = value;
```

```
tail->type = 1;
if(tail->str[0] == '$'){
tail->type += 2;
}
free_head = free_head->next; // 更新空闲链表头，因为头默认指向序号最小的，也就是被分配
的那
WP_COUNT++; // 个，所以直接向next移动即可
return tail;
```

监视点链表 free$_wp$()

```
    void free_wp(int no){ // 传入wp的no
    if(!wp_exists(wp_pool[no].type)){ // 检查是否正在使用
panic("cannot free watchpoint: not used!\n");
return;
}
WP* curWP = head;
WP* delWP = &wp_pool[no];
delWP->type = 0; // 清空type
while(curWP && wp_exists(curWP->type)){ //while遍历，直到遍历到尾或不存在
if(curWP->next == delWP){ // 通过数组寻址，找到了要删除的wp
curWP->next = delWP->next;
free_tail->next = delWP; // 此wp回归空闲链表
free_tail = free_tail->next;
free_tail->next = NULL;
WP_COUNT--;
return;
}
}
return;
}
```

Listing:

```
    bool wp_update_value(){
if(WP_COUNT == 0) // 没有监视点，不用更新值
return false;
WP* curWP = head;
if(!curWP){
panic("WP head is NULL!\n");
}
bool changed = false; // 假设没有值的变化
while(curWP && wp_exists(curWP->type)){
bool success = false;
int cv = expr(curWP->str, &success); // 调用expr，为每一个wp计算新的值
if(curWP->value != cv){
printf("wp:%s value changed: %d => %d\n", curWP->str, curWP->value, cv);
curWP->value = cv;
```

```
15 changed = true; // 存在wp更新了值，提示cpu_exec有监视点值得更新
16 }
17 curWP = curWP->next;
18 }
19 return changed;
20 }
21
22 //cpu_exec.c,如果有更新，则更改NEMU状态为STOP，此时就实现了监视点值改变时停止
       执行
23 int wp_count = get_wp_count();
24 if(wp_count){
25 printf("check watchpoint...\n");
26 if(wp_update_value()){
27 nemu_state = NEMU_STOP;
28 }
29 }
```

```
(nemu) w $eip
[src/monitor/debug/expr.c,88,make_token] match rules[3] = "\$[a-z]+" at position 0 with len 4: $eip
start:$eip, end:$eip, tokens:1
REG: eip, len=3, val=1048576
$eip watchpoint allot, now 1 watchpoint
value is 1048576
(nemu) c
check watchpoint...
[src/monitor/debug/expr.c,88,make_token] match rules[3] = "\$[a-z]+" at position 0 with len 4: $eip
start:$eip, end:$eip, tokens:1
REG: eip, len=3, val=1048581
wp:$eip value changed: 1048576 => 1048581
(nemu) info w
        NO                  EXPR                TYPE                VAL
        0                   $eip                3                   1048581
(nemu) d 0
0 watchpoint removed, now 0 watchpoint left.
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026

(nemu)
```

图 5: 监视点

(二) breaking point

1. 断点的工作原理

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. 调试器设置断点的工作方式和上述通过监视点来模拟断点的方法大相径庭.

2. 问题：“一点也不能长?”

int3 指令不带操作数，长度 1 字节，是必须的吗? 如果变成 2 字节，其他指令不变，文章中的断点机制还能正常工作吗?

必须的

- gdb 断点是基于 signal 实现的

- 添加断点就是保存该地址的一个字节，插入 INT3 指令，机器码 0xcc

- 执行到断点处，由 gdb 捕获 signal，将原来的字节写回，回到上一条指令 (被替换的指令) 重新执行

这时，程序的其他指令没有改变!

如果变成 2 字节，断点处的 1 个字节被 2 个字节替代，空间不足，而如果想要占用后一个字节空间，则又不能保证其他指令的完整性，所以断点机制不能正常工作

3. 问题：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

1. 首先，不能在该指令处“及时停下”：因为不能及时检测到第一个字节的 0xcc

2. 从断点恢复执行，也可能导致一些错误：如语法错误、跳转错误，因为从中间执行可能会忽略指令的部分执行结果

3. 会影响程序性能：执行指令需要额外处理，来跳过设置在非首字节的断点

综上，建议断点设置在指令首字节，充分利用 debugger，避免错误或性能损失

4. 问题：NEMU 的前世今生

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?

用途不同：Emulator 用于模拟硬件平台，Debugger 用于程序调试和错误排除

机理不同：Emulator 的底层实现与实际的硬件平台关联不大，只需要保证呈现给外部的行为跟目标系统一致（不需要保证内部运行原理一致)；而 Debugger 却是要完完全全根据硬件平台的实现，来进行调试工作

GDB 调试程序

系统启动 gdb 进程，调用系统函数 fork() 来创建 gdb 子进程，子进程需要做：1. 调用系统函数 ptrace(PTRACE_TRACEME，[其他参数])

2. 通过 execc 来加载、执行可执行程序，该程序就在 gdb 子进程中执行

ptrace 系统函数是 Linux 内核提供的一个用于进程跟踪的系统调用，通过它，一个进程 (tracer) 可以读写另外一个进程 (tracee) 的指令空间、数据空间、堆栈和寄存器的值, 并且 tracer 进程能截获 tracee 进程的所有信号；gdb 根据信号的属性来决定：在继续运行目标程序时是否把当前截获的信号转交给目标程序，如此一来，目标程序就在 gdb 发来的信号指挥下进行相应的动作

断点和单步执行

断点：将源码断点处对应的汇编代码存储到断点链表；在原处插入 INT3 执行到此处，os 发送 SIGTRAP 信号给 tracee，此时 gdb 能捕获，于是从断点链表中查找，替换回原汇编代码，并使得 PC 指针回退，等待用户的调试指令

单步执行：首先计算下一条源码对应汇编代码的 PC 值，当执行到此处停止，等待用户输入调试指令

(三) i386 手册

1. 问题：通过目录定位关注的问题

2. 必答题

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：位于 2.3.4 Flag Register，P33

ModR/M：位于 17.2.1 ModR/M and SIB Bytes，P241

mov：位于 17.2.2.11 Instruction Set Detail，P345 P351

- EFLAGS 寄存器中的 CF 位是什么意思?

在 x86 架构的 CPU 中，EFLAGS 寄存器是一个用于存储状态标志、控制标志和系统标志的寄存器。CF 位，也就是 Carry Flag（进位标志位），是 EFLAGS 寄存器中的第 0 位。

CF 位主要用于指示算术运算中的进位或借位。例如，在无符号算术运算中，当最高位产生进位时，CF 会被设置为 1；如果没有进位，CF 会被清除为 0。在减法或比较操作中，如果有借位发生，CF 同样会被设置为 1。

CF 位在以下情况中特别重要:

无符号数运算中表示进位或借位。多字节算术运算中传递进位或借位。一些位移和旋转指令中用于存储被移出的位

- ModR/M 字节是什么?

ModR/M 字节是 x86 指令集中用于指定操作数的一个字节。它存在于许多指令中，特别是那些需要指定一个或多个操作数的指令。ModR/M 字节由三部分组成:

Mod（修改）字段（最高的两位，位 6 和 7）：用于指示操作数的寻址模式或直接操作数。

Reg/Opcode 字段（中间的三位，位 3、4 和 5）：可以用来指定一个寄存器或扩展操作码。

R/M（寄存器/内存）字段（最低的三位，位 0、1 和 2）：用于指定一个寄存器作为操作数或一个内存地址。

- mov 指令的具体格式是怎么样的?

1. 寄存器到寄存器:

mov reg, reg 例如：mov eax, ebx（将 ebx 寄存器的值移动到 eax 寄存器）。

2. 内存到寄存器:

mov reg, [memory_address] 例如：mov eax, [ebx]（将存储在由 ebx 指向的内存地址的值移动到 eax 寄存器）。

3. 寄存器到内存:

mov [memory_address], reg 例如：mov [ebx], eax（将 eax 寄存器的值移动到由 ebx 指向的内存地址中）。

4. 立即数到寄存器:

mov reg, immediate 例如：mov eax, 5（将立即数 5 移动到 eax 寄存器）。

5. 立即数到内存:

mov [memory_address], immediate 例如：mov [ebx], 5（将立即数 5 移动到由 ebx 指向的内存地址中）。

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?

总计 4085

$$find. - name" * [.h|.cpp]"|xargswc - l$$

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.

```
fufu@fufu-virtual-machine:/mnt/hgfs/sharing/ics2017$ git diff --stat pa1 pa0
 nemu/include/cpu/reg.h              |  16 +---
 nemu/include/monitor/watchpoint.h   |  11 ---
 nemu/src/monitor/cpu-exec.c         |   8 --
 nemu/src/monitor/debug/expr.c       | 300 ++-----------------------------------
 nemu/src/monitor/debug/ui.c         | 113 ------------------------
 nemu/src/monitor/debug/watchpoint.c | 124 +-------------------------
 6 files changed, 9 insertions(+), 563 deletions(-)
```

图 6: change

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?

$$find. - name" * [.h|.cpp]"|xargsgrep"^"|wc - l$$

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：开启所有警告选项；编译时会打印出编译时所有错误或警告信息

-Werror：将警告视为错误；要求 GCC 将所有的警告当成错误进行处理，编译器立即停止编译，并报告错误信息

提高代码质量，尽早发现潜在问题：-Wall 检测个更多可能存在的问题，而-Werror 强制开发者修复所有警告内容

提升代码可移植性：使用不同编译器，会发现一些警告和错误信息，如果开发时就发现并修复这些信息，能帮助代码在不同编译器和 os 上正确编译运行

五、 实验总结

(一) Bug

1. Bug1: 主机与虚拟机间复制粘贴问题

在虚拟机内，选择 VMware 菜单的“VM”->“安装 VMware Tools...”。根据提示完成安装。重新启动虚拟机。

在 VMware 菜单中选择“编辑”->“虚拟机设置”->“选项”标签页。

在“选项”下，选择“来宾隔离”并确保“启用拖拽”和“启用剪贴板”选项均已勾选。

bug2:linux 没有 pow() 函数

忘记包含 <math.h> 头文件。

在编译时忘记链接数学库 libm。

系统可能缺少数学库或者数学库没有正确安装。

bug3 VMware Workstation 与 Device/Credential Guard 不兼容问题

1. 禁用 Hyper-V 如果你不需要 Hyper-V，可以尝试禁用它。打开命令提示符（以管理员身份）并运行以下命令：

$bcdedit/sethypervisorlaunchtypeoff$ 然后重启计算机。

2. 禁用 Device Guard 和 Credential Guard 如果你的系统上启用了 Device Guard 或 Credential Guard，你需要禁用它们。这些特性是为了增强系统安全性设计的，但它们可能会与 VMware 不兼容。禁用它们通常需要修改组策略或使用注册表编辑器。

使用组策略编辑器：

打开“本地组策略编辑器”（gpedit.msc）。导航到“计算机配置”-> “管理模板”-> “系统”-> “设备防护”。找到并禁用以下设置：“启用虚拟化基础的安全性”“启用 Credential Guard”

使用注册表编辑器：运行 regedit 打开注册表编辑器。找到名为 LsaCfgFlags 的 DWORD 值并将其设置为 0。

如果没有这个值，你可能需要创建它。