
系统综合课程设计-实验报告

PA1：开天辟地的篇章，最简单的计算机

姓名：孙铭

学号：1711377

学院：计算机学院

专业：计算机科学与技术

时间：2020年3月15日

系统综合课程设计-实验报告

PA1：开天辟地的篇章，最简单的计算机

1. 概述

1.1 实验目的

1.2 实验内容

2. 环境配置

3. PA1——阶段1

3.1 简单计算机模型

3.1.1 NEMU的执行流程

3.1.2 代码：实现正确的寄存器结构体

3.1.3 问题：究竟执行了多久？

3.1.4 问题：谁来指示程序的运行结果？

3.2 基础设施：简易调试器

3.2.1 代码：单步执行、打印寄存器、扫描内存

4. PA1——阶段2

4.1 词法分析

4.1.1 代码：实现算术表达式的词法分析

4.2 表达式求值

4.2.1 代码：实现算术表达式的递归求值

4.2.2 代码：实现算术表达式的递归求值

4.2.3 代码：实现算术表达式的递归求值

4.2.3 代码：完善扫描内存的功能

5. PA1——阶段3

5.1 监视点

5.1.1 代码：实现监视点池的管理

5.1.2 问题：static的使用

5.1.3 代码：实现监视点

5.2 断点

5.2.1 问题：一点也不能长

5.2.2 问题：“随心所欲”的断点

5.2.3 问题：NEMU的前世今生

5.3 i386手册的学习

5.3.1 问题：通过目录定位关注的问题

5.3.2 必答题

1. 概述

1.1 实验目的

- 熟悉GUN/Linux平台
- 初步探究“程序在计算机上运行”的相关原理
- 初步学习GDB并在PA上实现简易调试器

1.2 实验内容

PA1的实验主要为简易调试器的实现，具体可以分成如下三个阶段。

- 第一阶段，模拟寄存器结构，实现调试器基本功能。
- 第二阶段，实现调试功能的表达式求值，并完善阶段一中的扫描内存函数。
- 第三阶段，实现调试功能中的监视点，学习断点相关知识与i386手册。

2. 环境配置

本次作业所使用计算机处理器为 Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80GHz，虚拟机平台为 VMware Workstation Pro，GUN/Linux开发平台版本为 ubuntu16.04。

ics2018工程目录文件内容如下。

```
ics2018
|--init.sh          # 初始化脚本
|--Makefile         # 用于工程打包提交
|--nanos-lite       # 微型操作系统内核
|--navy-apps        # 应用程序集
|--nemu             # NEMU
|--nexus-am         # 抽象计算机
```

NEMU工程目录源文件组织如下。

```
nemu
|--include          # 存放全局使用的头文件
|   |--common.h     # 公用的头文件
|   |--cpu
|       |--decode.h # 译码相关
|       |--exec.h   # 执行相关
|       |--reg.h    # 寄存器结构体的定义
|       |--rtl.h    # RTL指令
|       |--debug.h   # 一些方便调试用的宏
|       |--device    # 设备相关
|       |--macro.h   # 一些方便的宏定义
|       |--memory    # 访问内存相关
```

```
|   |--monitor
|   |   |--expr.h          # 表达式求值相关
|   |   |--monitor.h
|   |   |--watchpoint.h    # 监视点相关
|   |   |--nemu.h
|   |--Makefile           # 指示NEMU的编译和链接
|   |--Makefile.git        # git版本控制相关
|   |--runall.sh          # 一键测试脚本
|   |--src                 # 源文件
|   |   |--cpu
|   |   |   |--decode      # 译码相关
|   |   |   |--exec
|   |   |   |--intr.c      # 中断处理相关
|   |   |   |--reg.c       # 寄存器相关
|   |   |   |--device      # 设备相关
|   |   |   |--main.c
|   |   |   |--memory
|   |   |   |   |--memory.c # 访问内存的接口函数
|   |   |   |--misc
|   |   |   |   |--logo.c   # “i386”的logo
|   |   |   |--monitor
|   |   |   |   |--cpu-exec.c # 指令执行的主循环
|   |   |   |   |--diff-test
|   |   |   |   |--debug     # 简易调试器相关
|   |   |   |   |   |--expr.c # 表达式求值的实现
|   |   |   |   |   |--ui.c   # 用户界面相关
|   |   |   |   |   |--watchpoint.c # 监视点的实现
|   |   |   |   |--monitor.c
```

此外，参照PA0中的环境配置，安装实验所需要的包（此处不再赘述），在终端中编写代码所用的工具为vim编辑器，并添加了vim的配置文件`.vimrc`，在文件中加入了如下内容。

```
55 set number
56 set autoindent
57 set smartindent
58 set showmatch
59 set ruler
60 set incsearch
61 set tabstop=4
62 set shiftwidth=4
63 set softtabstop=4
64 set cindent
65 set clipboard+=unnamed
66 set fileencodings=utf-8,gdb2312,gbk,gb18030
67 set termencoding=utf-8
68 set encoding=prc
```

3. PA1——阶段1

3.1 简单计算机模型

PA的目的是要实现NEMU，一款经过简化的x86全系统模拟器。进一步，NEMU是一个虚拟出来的计算机系统，物理计算机中的基本功能，在NEMU中都是通过程序来实现的。我们可以将计算机看成由若干个硬件部件组成，这些部件之间相互协助，完成运行程序这件事情。在NEMU中，每一个硬件部件都由一个程序相关的数据对象来模拟，例如变量、数组、结构体等。而对这些部件的操作则通过对相应数据对象的操作来模拟。

3.1.1 NEMU的执行流程

参考实验指导，NEMU的执行流程如下。

```
init_monitor(argc, argv)      //和monitor相关的初始化工作
reg_test(); //生成随机数，测试寄存器结构的正确性
load_img(); //读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
restart(); //模拟“计算机启动”，并将eip初始值设为0x100000，以确保CPU从该位置开始执行程序
init_regex(); //正则式
init_wp_pool(); //监视点
init_device(); //设备的初始化
welcome(); //输出欢迎信息和 NEMU 编译时间
ui_mainloop(is_batch_mode); //用户界面主循环程序，实现交互功能
```

3.1.2 代码：实现正确的寄存器结构体

实现正确的寄存器结构体

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

根据实验指导可以知道，对于寄存器的结构，除程序计数器eip外，32位寄存器、16位寄存器、8位寄存器各8个，且这些寄存器物理结构相关联，如下图所示。



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。这些寄存器在物理上不是相互独立的，例如EAX的低16位是AX，而AX又分成AH和AL。

首先，观察这些寄存器在`nemu/include/cpu/reg.h`下的声明。

```
6 enum { R_EAX, R_ECX, R_EDX, R_EBX, R_ESP, R_EBP, R_ESI, REDI };
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 };
```

三类寄存器数组的声明方式如下。

```
47 extern const char* regsl[];
48 extern const char* regsw[];
49 extern const char* regsb[];
```

在`nemu/src/cpu/reg.c`中，可以找到寄存器数组的初始化代码如下。

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

根据前文对寄存器结构的分析，不难得到本题的答案，在文件`nemu/include/cpu/reg.h`下，寄存器结构体的实现代码如下图所示。

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

最后，将目录切至`nemu/`下，在终端中输入`make run`，成功输出`welcome`与编译时间，说明寄存器结构体代码实现正确。

```
sun@ests:~/Desktop/ics2018/nemu/src/cpu$ cd ../../
sun@ests:~/Desktop/ics2018/nemu$ make run
./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: 15:20:36, Mar 4 2020
For help, type "help"
(nemu)
```

3.1.3 问题：究竟执行了多久？

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？

根据实验指导，`cpu_exec()`函数定义在`nemu/src/monitor/cpu-exec.c`文件下。函数`cpu-exec()`部分代码如下。

```

17 void cpu_exec(uint64_t n) {
18     if (nemu_state == NEMU_END) {
19         printf("Program execution has ended. To restart the program,
20             n again.\n");
21     }
22     nemu_state = NEMU_RUNNING;
23
24     bool print_flag = n < MAX_INSTR_TO_PRINT;
25
26     for (; n > 0; n --) {
27         /* Execute one instruction, including instruction fetch,
28          * instruction decode, and the actual execution. */
29         exec_wrapper(print_flag);

```

注意到循环`for(; n > 0; n --)`, 这里即是函数`cpu-exec()`模仿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。

n的定义方式为`uint64_t`, 即`unsigned long long`(64位无符号整型)。当传入-1时, 由于无符号数的特性, 相当于`uint64_t`中的 $2^{64} - 1$, 即最大值。这样做可以保证函数能不断执行指令直到`nemu_trap`。

为了进一步证明, 我在`cpu-exec()`函数中加入了代码
`printf("%lu\n", n)`, 再次执行`make run`, 结果如下。可以看到, n是一个非常大的值。

```

Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 15:20:36, Mar 4 2020
For help, type "help"
(nemu) c
n = 18446744073709551615
nemu: HIT GOOD TRAP at eip = 0x00100026

```

3.1.4 问题: 谁来指示程序的运行结果?

谁来指示程序的结束?

在程序设计课上老师告诉你, 当程序执行到`main()`函数返回处的时候, 程序就退出了, 你对此深信不疑。但你是否怀疑过, 凭什么程序执行到`main()`函数的返回处就结束了?如果有人告诉你, 程序设计课上老师的说法是错的, 你有办法来证明/反驳吗?如果你对此感兴趣, 请在互联网上搜索相关内容。

程序执行到`main()`函数返回处时, 并没有直接退出。而是继续执行一个叫做`atexit()`的函数, 它是在正常程序退出时调用的注册函数。该函数的原型为
`int atexit (void (*)(void))`。

一个进程可以注册若干个函数, 这些函数由`exit`自动调用, 被称为终止函数。而`atexit`可以注册这些函数。`exit`调用终止处理函数的顺序与`atexit`注册的顺序相反。其原因是参数压栈, 参数的压栈是先进后出, 和函数的栈帧相同。若一个函数被多次注册, 也就会被多次调用。

3.2 基础设施: 简易调试器

根据实验指导, 我们需要实现的简易调试器应该具有的功能如下。

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

3.2.1 代码：单步执行、打印寄存器、扫描内存

实现单步执行、打印寄存器、扫描内存

熟悉了 NEMU 的框架之后，这些功能实现起来都很简单，同时我们对输出的格式不作硬性规定，就当做是熟悉 GNU/Linux 编程的一次练习吧。

不敢下手？别怕，放手去写！编译运行就知道写得对不对。代码改挂了，就改回来呗；代码改得面目全非，还 git 呀！

这个问题覆盖了三个子问题，即单步执行、打印寄存器、扫描内存。

解析命令使用 `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。简易调试器的第一步就是在 `cmd_table` 结构中，进行命令-描述信息-处理函数的定义。

在文件 `nemu/src/monitor/debug/ui.c` 中，对 `cmd_table` 添加代码如下。

```

53 static struct {
54     char *name;
55     char *description;
56     int (*handler) (char *);
57 } cmd_table [] = {
58     { "Help", "Display informations about all supported commands", cmd_help },
59     { "c", "Continue the execution of the program", cmd_c },
60     { "q", "Exit NEMU", cmd_q },
61     { "si", "args:[N];execute [N] instructions step by step", cmd_si }, // 单步执行程序
62     { "info", "args:r/w;print information about registers or watchpoint", cmd_info },
63     { "x", "x [N] [EXPR];scan the memory", cmd_x },
64     { "p", "expr", cmd_p },
65     { "w", "set the watchpoint", cmd_w },
66     { "d", "delete the watchpoint", cmd_d },
67     /* TODO: Add more commands */
68 };
69 
```

单步执行

继而，先来解决第一个问题：单步执行。该命令涉及到 `cmd_si()` 函数，比较简单，只需简单调用 `cpu_exec()` 函数即可。

```

96 static int cmd_si(char *args) {
97     uint64_t N = 0;
98     if(args == NULL){
99         N = 1;
100    }
101    else{
102        int nRet = sscanf(args, "%llu", &N);
103        if(nRet <= 0){
104            printf("args error in cmd_si\n");
105            return 0;
106        }
107    }
108    //printf("in cmd_si, N = %llu\n", N);
109    cpu_exec(N);
110    return 0;
111 }

```

测试样例如下。

```

[src/monitor/monitor.c,47,load_default_img] No image is given. Use the default b
uild-in image.
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 15:20:36, Mar 4 2020
For help, type "help"
(nemu) si 6
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
100017: 66 c7 84 99 00 e0 ff ff 01 00   movw $0x1,-0x2000(%ecx,%ebx,4)
(nemu) si
100021: b8 00 00 00 00          movl $0x0,%eax
(nemu) si
nemu: HIT GOOD TRAP at eip = 0x00100026
100026: d6                      nemu trap (eax = 0)
(nemu) 

```

打印寄存器

在打印寄存器这里，涉及到的函数是 `cmd_info()` 函数，结合前文提到的与寄存器相关的函数及数组，使用 `printf()` 函数进行打印输出。

```

113 static int cmd_info(char *args){
114     char s;
115     if(args==NULL){ //没有提供参数时
116         printf("args error in cmd_info\n");
117         return 0;
118     }
119     int nRet = sscanf(args, "%c", &s);
120     if(nRet <= 0){ //解析失败
121         printf("args error in cmd_info\n");
122         return 0;
123     }
124     if(s == 'r'){ //打印寄存器
125         int i;
126         //32位寄存器
127         for(i = 0; i < 8; i++){
128             printf("%s 0x%08x\n", regsl[i], reg_l(i)); //regsl[i]和reg_l(i)定义见reg.h和r
eg.c
129             printf("eip 0x%08x\n", cpu.eip);
130             //16位寄存器
131             for(i = 0; i < 8; i++)
132                 printf("%s 0x%04x\n", regsw[i], reg_w(i));
133             //8位寄存器
134             for(i = 0; i < 8; i++)
135                 printf("%s 0x%02x\n", regsb[i], reg_b(i));
136             return 0;
137         }
138     }
139     if(s == 'w'){ //打印监视点信息
140         print_wp();
141         return 0;
142     }
143     printf("args error in cmd_info\n");
144     return 0;
145 }

```

由于本报告是在PA1所有内容完成后书写的，因此 `cmd_info()` 函数中涉及到了后文监视点的内容，这里暂时先人工屏蔽。测试样例如下。

```
(nemu) info r
eax 0x0
ecx 0x100027
edx 0x2bfc1321
ebx 0x2
esp 0x154f3773
ebp 0x5512630a
esi 0x6c8a2bc5
edi 0x10556bec
eip 0x100027
ax 0x0
cx 0x27
dx 0x1321
bx 0x2
sp 0x3773
bp 0x630a
si 0x2bc5
di 0x6bec
al 0x0
cl 0x27
dl 0x21
bl 0x2
ah 0x0
ch 0x0
dh 0x13
bh 0x0
```

扫描内存

在 `nemu/src/memory/memory.c` 中，可以看到关于访问内存函数的定义。

```
14 uint32_t paddr_read(paddr_t addr, int len) {
15     return pmem_rw(addr, uint32_t) & (~0u >> ((4 - len) << 3));
16 }
17 //~为按位取反，u后缀为unsigned，即将32位的0按位取反得111...111后右移(4-len)
18 //<3位
19 //len取1 2 3 4,得pmem_rw(addr, uint32_t)的最后8 16 24 32位
20 void paddr_write(paddr_t addr, int len, uint32_t data) {
21     memcpy(guest_to_host(addr), &data, len);
22 }
23
24 uint32_t vaddr_read(vaddr_t addr, int len) {
25     return paddr_read(addr, len);
26 }
```

代码中，参数 `addr` 为内存首地址，根据 `len` 取值的不同，可以实现控制读取内存字节数的作用。这里为直观比较，先实现每次输出 N 个连续的单字节。连续的 4 字节同理，只需控制 `vaddr_read()` 函数的 `len=4`，并调节相应的 `addr` 即可。

```
147 static int cmd_x(char *args){
148     //注意内存8bit算一个
149     int nLen = 0;
150     vaddr_t addr;
151     int nRet = sscanf(args, "%d 0x%x", &nLen, &addr); //先设定addr为16进制数
152     if(nRet <= 0){ // 解析失败
153         printf("args error in cmd_x\n");
154         return 0;
155     }
156     //访问内存的函数uint32_t vaddr_read(vaddr_t addr, int len){return paddr_read(addr,
157     len);}
158     //设置一行打印4个（每个为uint_8类型，16进制的形式打印）
159     printf("Memory:");
160     for(int i = 0; i < nLen; i++){
161         if(i % 4 == 0)
162             printf("\n0x%02x: 0x%02x", addr + i, vaddr_read(addr + i, 1));
163         else
164             printf(" 0x%02x", vaddr_read(addr + i, 1));
165     }
166     printf("\n");
167 }
```

使用测试样例 `x 39 0x100000`，结果如下。

```
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 15:20:36, Mar 4 2020
For help, type "help"
(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
```

与 `nemu/src/monitor/monitor.c` 中的默认镜像做对比，可以验证结果的正确性。

```
34 static inline int load_default_img() {
35     const uint8_t img [] = {
36         0xb8, 0x34, 0x12, 0x00, 0x00,          // 100000: movl $0x1234,%eax
37         0xb9, 0x27, 0x00, 0x10, 0x00,          // 100005: movl $0x100027,%ecx
38         0x89, 0x01,                           // 10000a: movl %eax,(%ecx)
39         0x66, 0xc7, 0x41, 0x04, 0x01, 0x00,    // 10000c: movw $0x1,0x4(%ecx)
40         0xb0, 0x02, 0x00, 0x00, 0x00,          // 100012: movl $0x2,%ebx
41         0x66, 0xc7, 0x84, 0x99, 0x00, 0xe0,    // 100017: movw $0x1,-0x2000(%ecx,%ebx,4)
42         0xff, 0xff, 0x01, 0x00,                // 100021: movl $0x0,%eax
43         0xb8, 0x00, 0x00, 0x00, 0x00,          // 100026: nemu_trap
44     };
45 }
```

4. PA1——阶段2

根据实验指导，可以知道表达式求值主要包含两个部分。

- 其一，识别出表达式中的单元。
- 其二，根据表达式的归纳定义进行递归求值。

4.1 词法分析

词法分析使用正则表达式识别出单词token。除却实验指导中给出的算术表达式外，还扩充实现了负数和指针解引用。

4.1.1 代码：实现算术表达式的词法分析

实现算术表达式的词法分析

你需要完成以下的内容：

- 为算术表达式中的各种 token 类型添加规则，你需要注意 C 语言字符串中转义字符的存在和正则表达式中元字符的功能。
- 在成功识别出 token 后，将 token 的信息依次记录到 tokens 数组中。

首先实现token的定义与声明，按照正则表达式的方式描述。

```
9 enum { //多字符token的类型标识
10     TK_NOTYPE = 256, TK_NUMBER, TK_HEX, TK_REG,
11     TK_EQ, TK_NEQ, TK_AND, TK_OR,
12     TK_NEGATIVE, TK_DEREF,
13
14     /* TODO: Add more token types */
15
16 };
17 //TK_NEGATIVE,TK_DEREF分别为单目的-与*,
18 //词法分析中不进行区分，在表达式求值前进行判断
```

值得指出，由于ASCII码值范围在[0, 255]之间，因此定义TK_NOTYPE的枚举值为256，后面的枚举值相应累加。正则表达式如下。

```

20 static struct rule {
21     char *regex; // 正则表达式
22     int token_type; // token类型
23 } rules[] = {
24
25     /* TODO: Add more rules.
26      * Pay attention to the precedence level of different rules.
27      */
28     // '\\"进行转义，代表一个反斜杠字符'\
29     {"+", TK_NOTYPE}, // spaces, +是正规式符号
30     {"0|[1-9A-Fa-f][0-9A-Fa-f]*", TK_HEX}, //16进制，需写在10进制之前
31     {"0|[1-9][0-9]*", TK_NUMBER},
32     {"\$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},
33     {"==", TK_EQ},
34     {"!=", TK_NEQ},
35
36     {"&&", TK_AND},
37     {"||", TK_OR},
38     {"!", '!' },
39
40     {"\\|+", '+' },
41     {"\\|-", '-' },
42     {"\\|*", '*' },
43     {"\\|/", '/' },
44
45     {"\\(|", '(' },
46     {"\\|)", ')' },
47
48 };
49 
```

下一步，在`make_token`中针对该类函数进行处理。由于这一问题下只实现了`TK_NUMBER`，因此我只放出`TK_NUMBER`情况下的代码如下。这里需要注意，`strncpy()`函数与`strcpy()`函数不同，`strncpy()`函数不会在字符串末尾追加结束符`\0`，因此这里需要手动向复制的字符串末尾添加结束符`\0`。

```

105     if(substr_len > 32)
106         assert(0);
107     if(rules[i].token_type == TK_NOTYPE)    //空格丢弃
108         break;
109     else{ //不是空格
110         tokens[nr_token].type = rules[i].token_type;
111         switch(rules[i].token_type){
112             case TK_NUMBER:
113                 strncpy(tokens[nr_token].str, substr_start, substr_len); //注意>
是复制指定长度，记得加'\0'
114                 *(tokens[nr_token].str + substr_len) = '\0';
115                 break;
116         }
117     }
118 }
```

接下来，实现解析命令表达式求值。在文件
`nemu/src/monitor/debug/ui.c`中声明并定义`cmd_p()`函数。

```

169 static int cmd_p(char *args) { //表达式求值
170     bool success;
171     int res = expr(args, &success); //uint32_t expr(char *e, bool *success) 在expr.c>
中，需要取地址&
172     if(success == false)
173         printf("error in expr()\n");
174     else
175         printf("the value of expr is:%d\n", res);
176     return 0;
177 }
```

输入测试样例`p 1+2*` (33)，测试结果如下。

```

Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 15:20:36, Mar 4 2020
For help, type "help"
(nemu) p 1+2* (33)
[src/monitor/debug/expr.c,98,make_token] match rules[2] = "0|[1-9][0-9]*" at position 0 w
ith len 1: 1
Success record : nr_token=0,type=257,str=1
[src/monitor/debug/expr.c,98,nake_token] match rules[9] = "\+" at position 1 with len 1:
+
Success record : nr_token=1,type=43,str=
[src/monitor/debug/expr.c,98,nake_token] match rules[2] = "0|[1-9][0-9]*" at position 2 w
ith len 1: 2
Success record : nr_token=2,type=257,str=2
[src/monitor/debug/expr.c,98,nake_token] match rules[11] = "\*" at position 3 with len 1:
*
Success record : nr_token=3,type=42,str=
[src/monitor/debug/expr.c,98,make_token] match rules[0] = "+" at position 4 with len 2:
[src/monitor/debug/expr.c,98,nake_token] match rules[13] = "\(" at position 6 with len 1:
(
Success record : nr_token=4,type=40,str=
[src/monitor/debug/expr.c,98,nake_token] match rules[2] = "0|[1-9][0-9]*" at position 7 w
ith len 2: 33
Success record : nr_token=5,type=257,str=33
[src/monitor/debug/expr.c,98,make_token] match rules[14] = "\)" at position 9 with len 1:
)
Success record : nr_token=6,type=41,str=
the value of expr is:67

```

4.2 表达式求值

4.2.1 代码：实现算术表达式的递归求值

实现算术表达式的递归求值

由于 ICS 不是算法课，我们已经把递归求值的思路和框架都列出来了。你需要做的是理解这一思路，然后在框架中填充相应的内容。实现表达式求值的功能之后，`p` 命令也就不难实现了。需要注意的是，上述框架中并没有进行错误处理，在求值过程中发现表达式不合法的时候，应该给上层函数返回一个表示出错的标识，告诉上层函数“求值的结果是无效的”。例如在 `check_parentheses()` 函数中， $(4+3)*((2-1)$ 和 $(4+3)*(2-1)$ 这两个表达式虽然都返回 `false`，因为前一种情况是表达式不合法，是没有办法成功进行求值的；而后一种情况是一个合法的表达式，是可以成功求值的，只不过它的形式不属于 BNF 中的 "`(<expr>)" "，需要使用 dominant operator 的方式进行处理，因此你还需要想办法把它们区别开来。`

当然，你也可以在发现非法表达式的时候使用 `assert(0)` 终止程序。不过这样的话，你在使用表达式求值功能的时候就要十分谨慎了。

(1) `bool check_parentheses(int p, int q)`

要实现递归求值，首先需要利用 `check_parentheses()` 函数。该函数的作用是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，若不匹配，则该表达式必定不合语法，也就不需要继续求值。这里使用计数器 `count` 来统计未被匹配的左括号的数目，参数 `p` 为表达式左位，参数 `q` 为表达式右位。算法比较简单，这里不再赘述。代码如下。

```
144 //判断表达式是否为(expr)，即左右括号是否匹配，默认要求p<q
145 bool check_parentheses(int p, int q){
146     if(p >= q){
147         printf("error:p>=q in check_parentheses\n");
148         return false;
149     }
150     if(tokens[p].type != '(' || tokens[q].type != ')')
151         return false;
152     int count = 0; //利用计数器，count计数当前未匹配的左括号的数目
153     for(int curr = p + 1; curr < q; curr++){
154         if(tokens[curr].type == '(')
155             count += 1;
156         if(tokens[curr].type == ')'){
157             if(count != 0)
158                 count -= 1;
159             else
160                 return false;
161         }
162     }
163     if(count == 0)
164         return true;
165     else
166         return false;
167 }
```

(2) `int findDominantOp(int p, int q)`

下一步需要考虑的问题是，如何将一个长表达式正确地分裂成两个子表达式。这里需要定义 `dominant operator` 为表达式人工求值时，最后一步进行运算的运算符，它指示了表达式的类型。于是问题转化为，要正确地对一个长表达式进行分裂，就是要找到它的 `dominant operator`。

根据实验指导中给出的实例分析，在 `token` 表达式中寻找 `dominant operator` 的方法如下。

- 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 `+`。

于是，结合计数法，我构建了函数 `int findDominantOp(int p, int q)` 来寻找 `dominant operator`，为了照应到后文的指针解引用，负数以及位运算符，这里我首先构建了优先级查找函数 `int priority(int type)`，该函数将在后文给出。这里只给出 `findDominantOp()` 的代码如下。

```
179 //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
180 int findDominantOp(int p, int q){
181     int level = 0; //计算括号层数
182     int pos = -1; //位置
183     int min_pri = 5; //最低优先级
184     for(int i = p; i <= q; i++){
185         if(tokens[i].type == TK_NOTYPE || tokens[i].type == TK_NUMBER || tokens[i].t
186             ype == TK_HEX || tokens[i].type == TK_REG)
187             continue;
188         if(tokens[i].type == '(') level += 1;
189         if(tokens[i].type == ')'){
190             if(level > 0) level -= 1;
191             else{
192                 printf("error: number of brackets does match in 'findDominantOp'\n")
193             }
194         }
195         if(level != 0) continue; //还在括号内
196         else if(level == 0){
197             int pri = priority(tokens[i].type);
198             if(pri <= min_pri){
199                 min_pri = pri;
200                 pos = i;
201             }
202             else if(pri > min_pri) continue;
203         }
204     }
205     //printf("op_pos=%d\n", pos);
206     return pos; //由于这里pos的初值赋为-1，故索引位置不存在时pos值不改变，直接返回
207 }
```

代码中的 `min_pri` 为最低优先级，这里赋值为5对应了优先级查找函数中的内容。简要思路是，对表达式进行遍历，利用 `level` 参数进行括号判断，如果 `level != 0`，说明表达式还在括号内，我们认为出现在一对括号中的 token 不是 `dominant operator`，因此执行 `continue;`。对应的，如果 `level = 0`，表达式不在括号内，则与之前最低优先级的字符进行优先级的比较，如果优先级低，则将更低的优先级赋值给 `min_pri`，同时用参数 `pos` 记录该操作符的位置。

(3) int eval(int p, int q)

到这里，程序已经能成功找到 `dominant operator`，接下来，思路是先对分裂出来的两个子表达式进行递归求值，再根据 `dominant operator` 的类型对两个子表达式的值进行运算即可。需要对 `eval()` 函数进行实现，参考报告中给出的框架如下。

```

eval(p, q) {
    if (p > q) {
        /* Bad expression */
    }
    else if (p == q) {
        /* Single token.
         * For now this token should be a number.
         * Return the value of the number.
        */
    }
    else if (check_parentheses(p, q) == true) {
        /* The expression is surrounded by a matched pair of parentheses.
         * If that is the case, just throw away the parentheses.
        */
        return eval(p + 1, q - 1);
    }
    else {
        op = the position of dominant operator in the token expression;
        val1 = eval(p, op - 1);
        val2 = eval(op + 1, q);

        switch (op_type) {
            case '+': return val1 + val2;
            case '-': /* ... */
            case '*': /* ... */
            case '/': /* ... */
            default: assert(0);
        }
    }
}

```

参考上述框架，递归实现 `int eval(int p, int q)`，由于本报告是在所有代码完成之后才书写，本部分内容涉及到后文复杂表达式的实现，故本函数代码会在后文中一并给出。

(4) `uint32_t expr(char *e, bool *success)`

在实现了表达式递归求值之后，还需要把最顶层的表达式求值函数补充完整，即调用 `make_token` 词法分析之后，调用 `eval` 函数完成表达式求值，由于涉及到后文内容，这里只给出部分代码如下。

```

280 //最顶层：词法分析调用+表达式计算
281 uint32_t expr(char *e, bool *success) {
282     if (!make_token(e)) {
283         *success = false;
284         return 0;
285     }

307     //这里未进行错误处理，假设输入的表达式均合法
308     *success = true;
309     return eval(0, nr_token-1);
310 }

```

综上，实现了简单表达式的求值部分，下面给出一些测试样例（这里需要注意，实现中不进行错误处理，发现非法表达式时使用 `assert(0)` 终止程序）。

```
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 15:20:36, Mar 4 2020
For help, type "help"
(nemu) p 2-1
the value of expr is:1
(nemu) p 2+3*8
the value of expr is:26
(nemu) p (2+3)*8
the value of expr is:40
(nemu) (2+3)*(2-8)
Unknown command '(2+3)*(2-8)'
(nemu) p (2+3)*(2-8)
the value of expr is:-30
(nemu) p (2+18)/(5-3)-9
the value of expr is:1
(nemu) p -9
error: number of brackets does match in 'findDominantOp'
error:p>q in eval,p=0,q=-2
nemu: src/monitor/debug/expr.c:214: eval: Assertion `0' failed.
Makefile:46: recipe for target 'run' failed
make: *** [run] 已放弃 (core dumped)
```

4.2.2 代码：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

在上述实现中，我们并没有考虑负数的问题，例如“1+−1”，“−1”（我们不实现自减运算，这里应该解释成 $(-1)=1$ ），它们会被判定为不合法的表达式。为了实现负数的功能，你需要考虑两个问题：

- 负号和减号都是 $-$ ，如何区分它们？
- 负号是个单目运算符，分裂的时候需要注意什么？

你可以选择不实现负数的功能，但你很快就要面临类似的问题了。

分析题目，词法分析是对单一的token进行读取和识别，无法根据前一token进行负号和减号的区分。故应在expr()函数识别完token之后，调用eval()进行表达式求值之前，进行区分。在优先级的层面，单目运算符高于双目运算符。对于 $-1-2$ 和 $-(1-2)$ 这类负号操作数不同的表达式，应该通过寻找DominantOp，并在递归中进行求值的方式进行区分。

从结合性的角度考虑，不同于双目运算符，单目运算符为左结合。若要实现 -1 的运算顺序为 $-(1)$ ，则需根据DominantOp为表达式中的最低优先级，若**findDominant(p,q)**为负号，则说明该表达式中没有双目运算符，因此必有p为负号且为最终决定的运算符。对于指针解引用的处理方式，也与负号的处理方式相似。这部分代码将在后文一并给出。

4.2.3 代码：实现算术表达式的递归求值

前文实现了简单的算术表达式求值，这里需要把功能扩展到复杂的表达式，利用BNF说明需要增添的功能如下。

```
<expr> ::= <decimal-number>
| <hexadecimal-number>      # 以"0x"开头
| <reg_name>                # 以"$"开头
| "(" <expr> ")"
| <expr> "+" <expr>
| <expr> "-" <expr>
| <expr> "*" <expr>
| <expr> "/" <expr>
| <expr> "==" <expr>
| <expr> "!=" <expr>
| <expr> "&&" <expr>
| <expr> "||" <expr>
| "!" <expr>
| "*" <expr>                 # 指针解引用
```

要解决的问题如下。

实现更复杂的表达式求值

你需要实现上文BNF中列出的功能。一个要注意的地方是词法分析中编写规则的顺序，不正确的顺序会导致一个运算符被识别成两部分，例如!=被识别成!=和=。关于变量的功能，它需要涉及符号表和字符串表的查找，我们在PA中暂不实现。

上面的BNF并没有列出C语言中所有的运算符，例如各种位运算，<=等等，==，!=和逻辑运算符很可能在使用监视点的时候用到，因此要求你实现它们。如果你在将来的使用中发现由于缺少某一个运算符而感到使用不方便，到时候你再考虑实现它。

(1) enum结构体

在文件nemu/src/monitor/debug/expr.c中，对符号表进行如下拓展，拓展时要考虑优先级。如先识别16进制再识别10进制，先识别!=再识别!和=

```
9 enum { //多字符token的类型标识
10    TK_NOTYPE = 256, TK_NUMBER, TK_HEX, TK_REG,
11    TK_EQ, TK_NEQ, TK_AND, TK_OR,
12    TK_NEGATIVE, TK_DEREF,
```

(2) rules规则

接下来是符号表rules的正则表达式扩展，这一部分其实已经在前文给出，这里为了解决问题，进一步详细说明。`\N`为转义字符，`TK_NOTYPE`代表空格符，这里我只添加了空格，没有列出`\t`、`\n`、`\r`等空白符，之后用到时会继续补充。其中`TK_REG`代表寄存器名称。

```
29   {"+", TK_NOTYPE}, // spaces, +是正规式符号
30   {"0x[1-9A-Fa-f][0-9A-Fa-f]*", TK_HEX}, //16进制，需写在10进制之前
31   {"0|[1-9][0-9]*", TK_NUMBER},
32   {"\$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},
33
34   {"==", TK_EQ},
35   {"!="}, TK_NEQ},
36
37   {"&&", TK_AND},
38   {"|||", TK_OR},
39   {"!!"}, !},
40
41   {"\+\+", '+'},
42   {"\_-", '-'},
43   {"\*\*", '*'},
44   {"\\/", '/'},
45
46   {"\\(\", '('},
47   {"\\)\", ')'};
```

(3) make_token函数

由于这一步添加了16进制数和寄存器的类型，而16进制数在进行词法分析时应该去掉首位`0x`，寄存器应该去掉首位`$`，因此需要对make_token()函数进行如下修改。

首先，我定义了寄存器和16进制数的变量如下。

```
92     char *substr_start = e + position;
93     char *regstr_start = e + position + 1;
94     char *hexstr_start = e + position + 2;
```

之后，在switch语句中添加16进制数和寄存器词法分析规则如下。

```

113         switch(rules[i].token_type){
114             case TK_NUMBER:
115                 strncpy(tokens[nr_token].str, substr_start, substr_len); //注意是复
116                 *(tokens[nr_token].str + substr_len) = '\0';
117                 break;
118             case TK_HEX: //16进制
119                 //char *hexstr_start = e + position + 2;
120                 strncpy(tokens[nr_token].str, hexstr_start, substr_len - 2);
121                 *(tokens[nr_token].str + substr_len - 2) = '\0';
122                 break;
123             case TK_REG: //寄存器
124                 //char *regstr_start = e + position + 1;
125                 strncpy(tokens[nr_token].str, regstr_start, substr_len - 1);
126                 *(tokens[nr_token].str + substr_len - 1) = '\0';
127                 break;
128         }

```

需要注意的一点是，由于16进制数和寄存器都在首位去除了字符，因此在复制时需要相应地改变复制字符串的长度。如16进制数由于去除了首位的`0x`，因此利用`strncpy()`函数进行复制时，复制长度应为`substr_len-2`。

(4) findDominantOp函数

处理完`make_token`函数之后，需要回到`findDominantOp`函数，前一步对于简单表达式的实现，只添加了`+*/`基本的四则运算，这里考虑到更为复杂的运算，不可避免的问题就是符号优先级，这里我首先定义了优先级查找函数，思路比较简单，返回值是符号的优先级。

```

171 //计算字符优先级
172 int priority(int type{
173     if(type == TK_NEGATIVE || type == TK_DEREF || type == '!') return 4;
174     else if(type == '/' || type == '*') return 3;
175     else if(type == '+' || type == '-') return 2;
176     else if(type == TK_EQ || type == TK_NEQ) return 1;
177     else if(type == TK_AND || type == TK_OR) return 0;
178     else return 6;
179 }

```

再来看之前提到的`findDominantOp()`函数如下，函数实现思路已在前文给出，不再赘述。

```

179 //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
180 int findDominantOp(int p, int q){
181     int level = 0; //计算括号层数
182     int pos = -1; //位置
183     int min_pri = 5; //最低优先级
184     for(int i = p; i <= q; i++){
185         if(tokens[i].type == TK_NOTYPE || tokens[i].type == TK_NUMBER || tokens[i].t
186         type == TK_HEX || tokens[i].type == TK_REG)
187             continue;
188         if(tokens[i].type == '(') level += 1;
189         if(tokens[i].type == ')'){
190             if(level > 0) level -= 1;
191             else{
192                 printf("error: number of brackets does match in 'findDominantOp'\n")
193             }
194         }
195         if(level != 0) continue; //还在括号内
196         else if(level == 0){
197             int pri = priority(tokens[i].type);
198             if(pri <= min_pri){
199                 min_pri = pri;
200                 pos = i;
201             }
202             else if(pri > min_pri) continue;
203         }
204     }
205     //printf("op_pos=%d\n", pos);
206     return pos; //由于这里pos的初值赋为了-1，故索引位置不存在时pos值不改变，直接返回
207 -1
208 }

```

(5) eval()函数

之后，需要对递归求值函数进行补充`int eval(int p, int q)`，相比于前文的简单表达式计算的递归函数，这里主要进行了两部分的补充。

其一是对16进制数使用 `sscanf()` 函数进行读取，对寄存器，使用前文实现的寄存器数组进行寄存器的访问与识别。其二是对负数、指针解引用及但双目运算符的处理，对于指针解引用时，需要使用 `var_addr()` 函数从内存中读取4位 `uint32_t` 类型的数据，实现代码如下。

```

209 int eval(int p, int q){
210     if(p > q){
211         printf("error:p>q in eval,p=%d,q=%d\n",p,q);
212         assert(0);
213     }
214     if(p == q){
215         int num;
216         switch(tokens[p].type){
217             case TK_NUMBER:
218                 sscanf(tokens[p].str, "%d", &num);
219                 return num; // expr=NUM;
220             case TK_HEX:
221                 sscanf(tokens[p].str, "%x", &num);
222                 return num; // 16进制寄存器
223             case TK_REG: //寄存器
224                 for(int i = 0; i < 8; i++){
225                     if(strcmp(tokens[p].str, regsl[i]) == 0)
226                         return reg_l(i);
227                     if(strcmp(tokens[p].str, regsw[i]) == 0)
228                         return reg_w(i);
229                     if(strcmp(tokens[p].str, regsb[i]) == 0)
230                         return reg_b(i);
231                 }
232                 if(strcmp(tokens[p].str, "eip") == 0)
233                     return cpu.eip;
234                 else{
235                     printf("error in TK_REG in eval()\n");
236                     assert(0);
237                 }
238             }
239         } // if p == q
240         //p<q时
241         if(check_parentheses(p, q) == true)
242             return eval(p + 1, q - 1); //expr=(expr)
243         else{
244             int op = findDominantOp(p, q);
245             vaddr_t addr;
246             int result; //TK_DEREF中使用
247             //单目运算符,op=p
248             switch(tokens[op].type){
249                 case TK_NEGATIVE: //负数
250                     return -eval(p + 1, q);
251                 case TK_DEREF: //指针解引用
252                     addr = eval(p + 1, q); //vaddr_t = uint32_t
253                     result = vaddr_read(addr, 4); //4是因为需要从内存中取出uint32的数据
254                     printf("addr = %u(0x%08x)--->value=%d(0x%08x)\n", addr, addr, result,
255                     result);
256                     return result;
257                 case '!':
258                     result = eval(p + 1, q);
259                     if(result != 0)
260                         return 0;
261                     else
262                         return 1;
263             }
264             //双目运算符
265             int val1 = eval(p, op - 1); //left
266             int val2 = eval(op + 1, q); //right
267             switch(tokens[op].type){
268                 case '+': return val1 + val2;
269                 case '-': return val1 - val2;
270                 case '*': return val1 * val2;
271                 case '/': return val1 / val2;
272                 case TK_EQ: return val1 == val2;
273                 case TK_NEQ: return val1 != val2;
274                 case TK_AND: return val1 && val2;
275                 case TK_OR: return val1 || val2;
276             }
277         } //else
278     }

```

(6) `expr()` 函数

最后，需要对 `expr()` 函数进行补充。这里注意，由于指针解引用和负号的特殊性，需要在词法分析后，表达式求值前对两者进行补充与识别。若前一位不为 `TK_NUMBER` 及 `)`，则将该字符token类型置为 `TK_NEGATIVE` 或 `TK_DEREF`。完整代码如下。

```

282 //最顶层：词法分析调用+表达式计算
283 uint32_t expr(char *e, bool *success) {
284     if (!make_token(e)) {
285         *success = false;
286         return 0;
287     }
288
289     /* TODO: Insert codes to evaluate the expression. */
290     //nr_token记录总的token数目，所有token存储在数组[0, nr_token-1]处
291     //先区分负数和减号，乘法和指针
292     if(tokens[0].type == '-')
293         tokens[0].type = TK_NEGATIVE;
294     if(tokens[0].type == '*')
295         tokens[0].type = TK_DEREF; //先判断第一位
296     for(int i = 1; i < nr_token; i++){
297         if(tokens[i].type == '-'){
298             if(tokens[i-1].type != TK_NUMBER && tokens[i-1].type != ')')
299                 //注意加上TK_NEGATIVE, 如--1。注意矫正从左至右进行
300                 tokens[i].type = TK_NEGATIVE;
301         }
302         if(tokens[i].type == '*'){
303             if(tokens[i-1].type != TK_NUMBER && tokens[i-1].type != ')')
304                 //加上TK_NEGATIVE
305                 tokens[i].type = TK_DEREF;
306         }
307     } //for
308
309     //这里未进行错误处理，假设输入的表达式均合法
310     *success = true;
311     return eval(0, nr_token-1);
312 }

```

综上，实现了复杂表达式的求值。

为验证实验结果正确性，参考助教给出的测试样例进行测试。测试样例如下。

```

12
0x1A
$eip          //若在程序起始时计算，为0x100000(1048576)
$ax
$a1
$ah          //值可能改变，但注意$ax=$ah*(16*16)+$a1

1+12*3
-(2+3)+8
-1*5
-2
--2
12+6/3

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

```

测试结果如下。

```

[src/monitor/monitor.c,30,welcome] Build time: 15:20:36, Mar  4 2020
For help, type "help"
(nemu) p 12
the value of expr is:12
(nemu) p 0x1A
the value of expr is:26
(nemu) p $eip
the value of expr is:1048576
(nemu) p $ax
the value of expr is:2136
(nemu) p $a1
the value of expr is:88
(nemu) p $ah
the value of expr is:8

```

```
(nemu) p 1+12*3
the value of expr is:37
(nemu) p -(2+3)*8
the value of expr is:-3
(nemu) p -1*5
the value of expr is:-5
(nemu) p -2
the value of expr is:-2
(nemu) p --2
the value of expr is:2
(nemu) p 12+6/3
the value of expr is:14
(nemu) p 1+12*3
the value of expr is:37
```

```
(nemu) p 1+12*3==37
the value of expr is:1
(nemu) p 1+12*3!=37
the value of expr is:0
(nemu) p !(1-1)
the value of expr is:1
(nemu) p 3==3&(*0x100000==0x1234b8)
addr = 1048576(0x100000)---->value=1193144(0x001234b8)
the value of expr is:1
(nemu) p *0x100000
addr = 1048576(0x100000)---->value=1193144(0x001234b8)
the value of expr is:1193144
```

4.2.3 代码：完善扫描内存的功能

在原有基础上增加表达式求值函数expr的调用，代码如下。

```
147 static int cmd_x(char *args){
148     //注意内存8bit算一个
149     int nLen = 0;
150     vaddr_t addr;
151     char str[256];
152     int nRet = sscanf(args, "%d %s", &nLen, str); //先设定addr为16进制数
153     if(nRet <= 0){ // 解析失败
154         printf("args error in cmd_x\n");
155         return 0;
156     }
157     //访问内存的函数uint32_t vaddr_read(vaddr_t addr, int len){return paddr_read(addr
158     , len);}
159     bool success;
160     addr = expr(str, &success);
161     printf("Memory:");
162     for(int i = 0; i < nLen; i++){
163         printf("\n0x%08x: 0x%08x", addr + 4 * i, vaddr_read(addr + 4 * i, 4));
164     }
165     printf("\n");
166 }
```

测试样例结果如下。

```
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 18:35:29, Mar 15 20
For help, type "help"
(nemu) x 10 $eip
Memory:
0x100000: 0x001234b8
0x100004: 0x0027b900
0x100008: 0x01890010
0x10000c: 0x0441c766
0x100010: 0x02bb0001
0x100014: 0x66000000
0x100018: 0x009984c7
0x10001c: 0x01fffffe0
0x100020: 0x0000b800
0x100024: 0x00d60000
(nemu) x 10 0x100000
Memory:
0x100000: 0x001234b8
0x100004: 0x0027b900
0x100008: 0x01890010
0x10000c: 0x0441c766
0x100010: 0x02bb0001
0x100014: 0x66000000
0x100018: 0x009984c7
0x10001c: 0x01fffffe0
0x100020: 0x0000b800
0x100024: 0x00d60000
```

5. PA1——阶段3

5.1 监视点

监视点的功能是监视一个表达式的值何时发生变化。

5.1.1 代码：实现监视点池的管理

实现监视点池的管理

为了使用监视点池，你需要编写以下两个函数（你可以根据你的需要修改函数的参数和返回值）：

```
WP* new_wp();
void free_wp(WP *wp);
```

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。框架代码中定义了 32 个监视点结构，一般情况下应该足够使用，如果你需要更多的监视点结构，你可以修改 `NR_WP` 宏的值。这两个函数里面都需要执行一些链表插入、删除的操作，对链表操作不熟悉的同学来说，这可以作为一次链表的练习。

依照实验指导中给出的要求，建议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。

首先，需要定义监视点的结构及后续所需的相关函数。

对 `nemu/include/monitor/watchpoint.h` 文件进行修改如下。

```
6 typedef struct watchpoint {
7     int NO;
8     struct watchpoint *next;
9
10    /* TODO: Add more members if necessary */
11    int oldValue; //旧值
12    char e[32]; //表达式
13    int hitNum; //todo:记录触发次数
14 } WP;
15
16 bool new_wp(char *arg);
17 bool free_wp(int num);
18 void print_wp();
19 bool watch_wp();
```

其次，需要使用池的数据结构进行监视点管理。即设定正在使用的监视点按顺序编号，若删去某监视点使得其余监视点编号保持不变，那么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。为此，需要先在 `nemu/src/monitor/debug/watchpoint.c` 中定义全局变量如下。

```
4 #define NR_WP 32
5
6 static WP wp_pool[NR_WP];
7 static WP *head, *free_;
8 static int used_next; //记录head中下一个使用的wp的索引号。在new_wp与free_wp中进行维护
9 static WP *wpTemp; //辅助
```

利用 `init_wp_pool()` 实现结构初始化如下。

```

11 void init_wp_pool() {
12     int i;
13     for (i = 0; i < NR_WP; i++) {
14         wp_pool[i].NO = i;
15         wp_pool[i].next = &wp_pool[i + 1];
16         wp_pool[i].oldValue = 0;
17         wp_pool[i].hitNum = 0; //各种初始化
18     }
19     wp_pool[NR_WP - 1].next = NULL;
20
21     head = NULL;
22     free_ = wp_pool;
23     used_next = 0;
24 }

```

接下来需要实现 `new_wp()` 函数和 `free_wp()` 函数。

`new_wp()` 函数的作用是从 `free_` 链表中返回一个空闲的监视点结构，
`free_wp()` 函数的作用是将 `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他函数调用。在调用 `new_wp()` 函数时若出现没有空闲监视点结构的情况，则通过 `assert(0)` 终止程序。

`new_wp()` 函数实现代码如下。

```

28 bool new_wp(char *args){ //从free链表中返回一个空闲的监视点结构。arg为该监视点的表达式。成功
29     if(free_ == NULL)
30         assert(0);
31     WP* result = free_; //记录result
32     free_ = free_ -> next; //更新free链表
33
34     //设置新wp的相关信息
35     result -> NO = used_next;
36     used_next++; //记录索引信息
37     result -> next = NULL; //脱离free 组织，变为链表最尾端（则要求next=NULL）
38     strcpy(result -> e, args); //记录表达式字符串
39     result -> hitNum = 0; //触发次数 初始化
40     bool success;
41     result -> oldValue = expr(result -> e, &success); //计算旧值
42     if(success == false){
43         printf("error in new_wp:expression fault!\n");
44         return false;
45     }
46
47     //更新链表：将wp添加到链表最尾端（理论上添加到开头也可以，且不用遍历链表）
48     wptemp = head;
49     if(wptemp == NULL)
50         head = result; //第一
51     else{
52         while(wptemp -> next != NULL)
53             wptemp = wptemp -> next; //temp找到链表尾部
54         wptemp -> next = result;
55     }
56
57     printf("Success: set watchpoint %d, oldValue=%d\n", result -> NO, result -> oldValue);
58     return true;
59 }

```

`free_wp()` 函数实现代码如下。

```

61 //删除监视点：将索引号为num的wp从head中删除，并添加到free中。成功返回true，否则false
62 bool free_wp(int num){
63     WP *thewp = NULL; //记录要被删除的监视点
64     if(head == NULL){
65         printf("no watchpoint now\n");
66         return false;
67     }
68
69     //先判断head，随后在遍历中判断temp->next（便于删除操作），先执行head链表的删除操作，记录
70     thewp
71     if(head -> NO == num){
72         thewp = head;
73         head = head -> next; //更新head链表
74     }else{
75         wptemp = head;
76         while(wptemp != NULL && wptemp -> next != NULL){
77             if(wptemp -> next -> NO == num){ //找到该wp
78                 thewp = wptemp -> next;
79                 wptemp -> next = wptemp -> next -> next; //更新head链表
80                 break;
81             }
82             wptemp = wptemp -> next;
83         }
84     }
85
86     //在free链表中进行添加
87     if(thewp != NULL){
88         thewp -> next = free_;
89         free_ = thewp; //更新free链表
90     }
91     return true;
92 }
93 }

```

5.1.2 问题: static 的使用

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?

先来看定义 `wp_pool` 等变量的代码。

```
4 #define NR_WP 32
5
6 static WP wp_pool[NR_WP];
7 static WP *head, *free_;
8 static int used_next; //记录head中下一个使用的wp的索引号。在new_wp与free_wp中进行维护
9 static WP *wptemp; //辅助
```

这些变量使用了 `static` 关键字定义了全局静态变量, 这种做法使得这些变量只能在 `watchpoint.c` 文件中进行访问, 而其它源文件相对该文件中的这些变量进行读写操作时, 只能依靠封装好的函数来实现, 这样可以保证数据的安全性。防止在 `watchpoint.c` 被其它文件调用时可能造成的数据流失或者被篡改的问题。

5.1.3 代码: 实现监视点

实现监视点

你需要实现上文描述的监视点相关功能, 实现了表达式求值之后, 监视点实现的重点就落在了链表操作上。如果你仍然因为链表的实现而感到调试困难, 请尝试学会使用 `assertion`. 在同一时刻触发两个以上的监视点也是有可能的, 你可以自由决定如何处理这些特殊情况, 我们对此不作硬性规定。

根据实验指导, 需要实现的监视点功能如下。

实现了监视点池的管理之后, 我们就可以考虑如何实现监视点的相关功能了. 具体的, 你需要实现以下功能:

- ◆ 当用户给出一个待监视表达式时, 你需要通过 `new_wp()` 申请一个空闲的监视点结构, 并将表达式记录下来。每当 `cpu_exec()` 执行完一条指令, 就对所有待监视的表达式进行求值(你之前已经实现了表达式求值的功能了), 比较它们的值有没有发生变化, 若发生了变化, 程序就因触发了监视点而暂停下来, 你需要将 `nemu_state` 变量设置为 `NEMU_STOP` 来达到暂停的效果. 最后输出一句话提示用户触发了监视点, 并返回到 `ui_mainloop()` 循环中等待用户的命令。
- ◆ 使用 `info w` 命令来打印使用中的监视点信息, 至于要打印什么, 你可以参考 GDB 中 `info watchpoints` 的运行结果。
- ◆ 使用 `d` 命令来删除监视点, 你只需要释放相应的监视点结构即可。

还是在 `nemu/src/monitor/debug/watchpoint.c` 文件中, 首先需要实现打印监视点信息的辅助函数 `print_wp()` 如下。

```
95 //打印所有监视点信息
96 void print_wp(){
97     if(head == NULL){
98         printf("no watchpoint now\n");
99         return;
100    }
101   printf("watchpoint:\n");
102  printf("NO. expr           hitTimes\n");
103  wptemp = head;
104  while(wptemp != NULL){
105      printf("%d %s           %d\n", wptemp -> NO, wptemp -> e, wptemp -> hitNum);
106      wptemp = wptemp -> next;
107  }
108 }
```

其次, 实现判断监视点是否触发的辅助函数 `watch_wp()`, 该函数进行监视点各表达式的求值, 如果存在发生变化的值, 则打印相关信息, 返回 `false`。代码如下。

```

111 bool watch_wp(){
112     bool success;
113     int result;
114     if(head == NULL)
115         return true;
116
117     wptemp = head;
118     while(wptemp != NULL){
119         result = expr(wptemp -> e, &success); //uint32_t expr(char *e, bool *success)在expr.c中
120         if(result != wptemp -> oldValue){ //发生改变
121             wptemp -> hitNum += 1; //触发次数
122             printf("Hardware watchpoint %d:%s\n", wptemp -> NO, wptemp -> e);
123             printf("Old Value:%d\nNew value:%d\n", wptemp -> oldValue, result);
124             wptemp -> oldValue = result; //更新oldValue
125             return false; //设定触发一次就返回
126         }
127         wptemp = wptemp -> next;
128     }
129     return true;
130 }
```

下一步，需要对 `nemu/src/monitor/debug/ui.c` 文件进行修改，即添加监视点设置及删除的解析命令，以及相应的函数。

首先是监视点申请，比较简单，直接调用 `new_wp()` 即可。

```

179 static int cmd_w(char *args){ //设置监视点， args为expr
180     new_wp(args);
181     return 0;
182 }
```

对于监视点删除部分的代码实现如下。大体思路是当监视点解析成功时，调用 `free_wp()` 函数，将 `WP` 的资源归还到 `free_` 链表中。

```

184 static int cmd_d(char *args){ //删除监视点， args为监视点编号
185     int num = 0;
186     int nRet = sscanf(args, "%d", &num);
187
188     if(nRet <= 0){ //解析失败
189         printf("args error in cmd_si\n");
190         return 0;
191     }
192
193     int r = free_wp(num);
194     if(r == false)
195         printf("error: no watchpoint %d\n", num);
196     else
197         printf("Success delete watchpoint %d\n", num);
198
199     return 0;
200 }
201 }
```

最后，需要在 `cmd_info()` 函数中对查看监视点信息进行补充。加一个判断及 `print_wp` 函数调用即可。

```

139     if(s == 'w'){ //打印监视点信息
140         print_wp();
141         return 0;
142     }
```

上述功能完整实现之后，由于监视点触发时程序需要暂停，因此当 `watch_wp()` 函数值为 `false` 时，直接将 `nemu` 的运行状态赋值为 `NEMU_STOP` 即可。这里需要对 `nemu/src/monitor/cpu-exec.c` 文件进行修改。

添加头文件如下。

```

1 #include "nemu.h"
2 #include "monitor/monitor.h"
3 #include "monitor/watchpoint.h"
```

对 `cpu` 执行函数 `cpu_exec()` 进行修改如下。

```

32 #ifdef DEBUG
33     /* TODO: check watchpoints here. */
34     if(watch_wp() == false){
35         //触发的相关信息在watch_wp()中已经输出
36         nemu_state = NEMU_STOP;
37     }
38 #endif

```

接下来，按照助教给出的实验流程中的测试步骤进行测试，测试流程如下。

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

测试结果如下。

```

[src/monitor/monitor.c,47,load_default_img] No image is given. Use the default b
uild-in image.
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 15:20:36, Mar 4 2020
For help, type "help"
(nemu) w $eip==0x100000
Success: set watchpoint 0, oldValue=1
(nemu) info w
watchpoint:
NO.    expr                                hitTimes
0     $eip==0x100000                         0
(nemu) c
Hardware watchpoint 0:$eip==0x100000
Old Value:1
New value: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

```

5.2 断点

根据实验指导，断点的功能是让程序暂停下来，从而方便地查看程序某一时刻的状态。根据老师给出的[这篇文章](#)以及实验指导，我理解了断点的原理和基本的实现方式。

断点通过CPU的特殊指令 `int3` 来实现，`int3` 即为 `x86` 体系结构中的软中断（对预定义的中断处理例程的调用）。`int3` 指令会产生一个特殊的单字节操作码（`0xCC`），当进程执行到该指令时，将会调用调试异常处理例程，即 `trap to debugger`，从而实现中断。

断点的实现方式如下。

在调试器中设定断点时，具体操作为：

(1) 保存目标地址上的数据

(2) 将目标地址上的第一个字节替换为 `int3` 指令

当调试器向操作系统请求开始运行进程后，进程最终一定会遇到 `int3` 指令。此时进程停止，操作系统将发送一个 `SIGTRAP` 信号。随后调试器要做如下：

(1) 在目标地址上用原来的指令替换掉 `int3`

(2) 将被跟踪进程中的指令指针向后递减1。这么做是必须的，因为现在指令指针指向的是已经执行过的 `int3` 之后的下一条指令。

由于进程此时仍然是停止的，用户可以同时被调试进程进行某种形式的交互，如查看变量的值，检查调用栈等。

5.2.1 问题：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？假设有另一种 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？

首先，`int3` 指令长度是1个字节是必须的。这在上面给出的文章中可以找到答案。这种单字节的形式非常有价值，因为它可以利用断点（包括其他单字节指令）来替换任何指令的第一个字节，而不需要编写多余的代码。

在 x86 上的 `int` 指令占用两个字节 `-0xcd`，后面跟着中断号。`int3` 可以被编码为 `cd 03`，但是有一个特殊的单字节指令保留给它，即 `-0xcc`。至于这样的原因，因为这允许我们插入一格断点，而不需要覆盖多条指令。以下面代码为例：

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

假设我们想要在 `dec eax` 上放置一个断点，这恰好是一个单字节指令（操作码为 `0x48`）。若替换断点指令的长度，使之超过1个字节，则我们将被迫覆盖下一条指令（调用）的一部分，这会让程序产生混淆，并可能产生完全无效的结果。考虑分支 `jz foo`，这时进程可能不会在 `dec eax` 处停下来，而是 CPU 会继续执行无效的指令。而为 `int3` 提供一种特殊的单字节编码则可以解决这个问题。因为 x86 架构上指令最短的长度为1字节，这样可以保证只有我们希望停止的那条指令被修改。

5.2.2 问题：“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。

这里以简单的hello world汇编程序为基础进行实验，程序输出结果为我们熟悉的Hello,world!，汇编代码如下。

```
section    .text
; The _start symbol must be declared for the linker
(1d)
global _start

_start:

; Prepare arguments for the sys_write system call:
; - eax: system call number (sys_write)
; - ebx: file descriptor (stdout)
; - ecx: pointer to string
; - edx: string length
mov        edx, len1
mov        ecx, msg1
mov        ebx, 1
mov        eax, 4

; Execute the sys_write system call
int       0x80

; Now print the other message
mov        edx, len2
mov        ecx, msg2
mov        ebx, 1
mov        eax, 4
int       0x80

; Execute sys_exit
mov        eax, 1
int       0x80

section    .data

msg1      db      'Hello,' , 0xa
len1      equ     $ - msg1
msg2      db      'world!', 0xa
len2      equ     $ - msg2
```

首先先进行正常编译、链接、运行如下。

```
sun@ests:~/Desktop$ nasm -f elf32 hello.asm
sun@ests:~/Desktop$ ld hello.o -o hello
sun@ests:~/Desktop$ ./hello
Hello,
world!
```

接下来将断点设置在指令的首字节，利用objdump -d查看各指令对应地址。

```
sun@ests:~/Desktop$ objdump -d hello
hello:      文件格式 elf32-i386

Disassembly of section .text:
08048080 <_start>:
8048080:   ba 07 00 00 00        mov    $0x7,%edx
8048085:   b9 b4 90 04 08        mov    $0x80490b4,%ecx
804808a:   bb 01 00 00 00        mov    $0x1,%ebx
804808f:   b8 04 00 00 00        mov    $0x4,%eax
8048094:   cd 80                int    $0x80
8048096:   ba 07 00 00 00        mov    $0x7,%edx
804809b:   b9 bb 90 04 08        mov    $0x80490bb,%ecx
80480a0:   bb 01 00 00 00        mov    $0x1,%ebx
80480a5:   b8 04 00 00 00        mov    $0x4,%eax
80480aa:   cd 80                int    $0x80
80480ac:   b8 01 00 00 00        mov    $0x1,%eax
80480b1:   cd 80                int    $0x80
```

在`0x80480aa`处设置断点，可以看到程序正常运行如下。

```
(gdb) file hello
Reading symbols from hello...(no debugging symbols found)...done.
(gdb) break *0x80480aa
Breakpoint 1 at 0x80480aa
(gdb) r
Starting program: /home/sun/Desktop/hello
Hello,
Breakpoint 1, 0x080480aa in _start ()
(gdb) si
world!
0x080480ac in _start ()
(gdb)
0x080480b1 in _start ()
(gdb)
[Inferior 1 (process 5481) exited with code 01]
(gdb)
The program is not being run.
```

`0x80480aa`处的指令为双字节指令，故在下一个字节`0x80480ab`处设置断点。

```
(gdb) file hello
Reading symbols from hello...(no debugging symbols found)...done.
(gdb) break *0x80480ab
Breakpoint 1 at 0x80480ab
(gdb) r
Starting program: /home/sun/Desktop/hello
Hello,
Program received signal SIGSEGV, Segmentation fault.
0x080480aa in _start ()
```

调试程序报错`Program received signal SIGSEGV, Segmentation fault.`，即程序收到信号`SIGSEV`，出现段错误。通过查询，[这个网站](#)给出了段错误的定义。

所谓的段错误，就是指访问的内存超出了系统所给这个程序的内存空间，通常这个值是由`gdtr`来保存的，它是一个48位的寄存器，其中的32位是保存由它指向的`gdt`表，后13位保存相应于`gdt`的下标，最后3位包括了程序是否在内存中以及程序的在cpu中的运行级别，指向的`gdt`是由以64位为一个单位的表，在这张表中就保存着程序运行的代码段以及数据段的起始地址以及与此相应的段限和页面交换还有程序运行级别还有内存粒度等等的信息。一旦一个程序发生了越界访问，cpu就会产生相应的异常保护，于是`segmentation fault`就出现了。

简单来说，段错误就是访问了不可访问的内存，这个内存要么是不存在的，要么是受系统保护的。

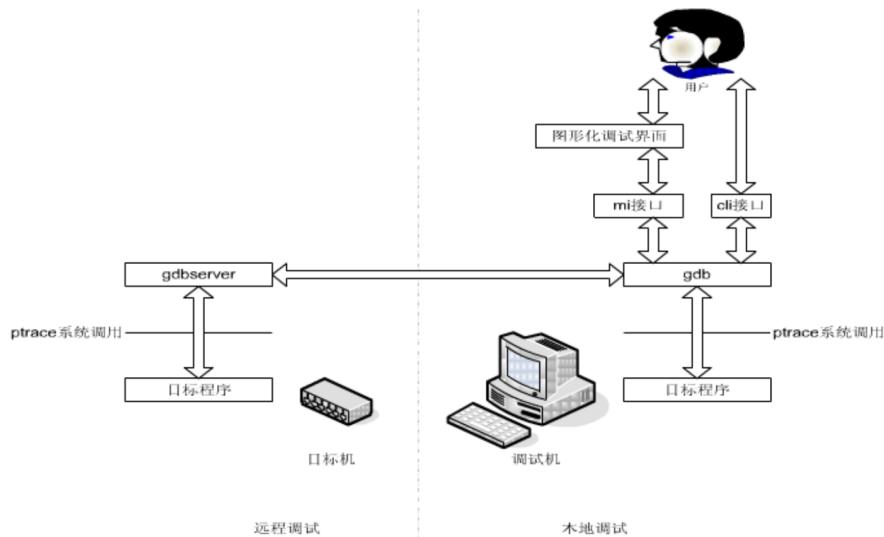
查阅了[大量资料](#)，这里之所以出现段错误，个人的想法是，在打印字符串的时候，实际上是打印某个地址开始的所有字符，但是当你想把整数当字符串打印的时候，这个整数被当成了一个地址，然后`printf()`从这个地址开始去打印字符，直到某个位置上的值为`\0`。所以，如果这个整数代表的地址不存在或者不可访问，自然也是访问了不该访问的内存，也就会报错`segmentation fault`。

5.2.3 问题：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？

根据实验指导，模拟器是模拟执行特定的硬件平台及其程序的软件程序，而调试器是用于测试和调试其他程序（“目标”程序）的计算机程序。从调试程序的方式上进行比较，PA1 中为 NEMU 实现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` 系统调用进行实现的。系统整体框架如下，图片来自[博客](#)。



`ptrace` 系统调用原型如下。`ptrace` 系统调用提供了一种方法，让父进程可以观察和控制其它进程的执行，检查和改变其核心映像及寄存器。主要用来实现断点调试和系统调用跟踪。

```
long ptrace(enum __ptrace_request request, \
pid_t pid,void *addr,void *data);
```

GDB 调试建立在信号的基础上的，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。

5.3 i386 手册的学习

5.3.1 问题：通过目录定位关注的问题

尝试通过目录定位关注的问题

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

在i386手册中搜索关键词 selector，发现484个结果.....



但目录中在第五章：寄存器管理中单独出现。

CHAPTER 5 MEMORY MANAGEMENT.....	91
5.1 SEGMENT TRANSLATION	92
5.1.1 Descriptors.....	92
5.1.2 Descriptor Tables.....	94
5.1.3 Selectors.....	96
5.1.4 Segment Registers.....	97
5.2 PAGE TRANSLATION.....	98
5.2.1 Page Frame.....	98
5.2.2 Linear Address.....	98

跳转至相应章节，找到关于 selector 的相关定义。

5.1.3 Selectors

The selector portion of a logical address identifies a descriptor by specifying a descriptor table and indexing a descriptor within that table. Selectors may be visible to applications programs as a field within a pointer variable, but the values of selectors are usually assigned (fixed up) by linkers or linking loaders. Figure 5-6 shows the format of a selector.

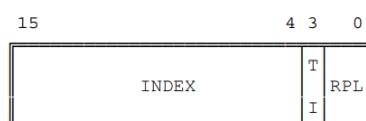
Index: Selects one of 8192 descriptors in a descriptor table. The processor simply multiplies this index value by 8 (the length of a descriptor), and adds the result to the base address of the descriptor table in order to access the appropriate segment descriptor in the table.

Table Indicator: Specifies to which descriptor table the selector refers. A zero indicates the GDT; a one indicates the current LDT.

Requested Privilege Level: Used by the protection mechanism. (Refer to Chapter 6.)

Because the first entry of the GDT is not used by the processor, a selector that has an index of zero and a table indicator of zero (i.e., a selector that points to the first entry of the GDT), can be used as a null selector. The processor does not cause an exception when a segment register (other than CS or SS) is loaded with a null selector. It will, however, cause an exception when the segment register is used to access memory. This feature is useful for initializing unused segment registers so as to trap accidental references.

Figure 5-6. Format of a Selector



TI - TABLE INDICATOR
RPL - REQUESTOR'S PRIVILEGE LEVEL

5.3.2 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 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？

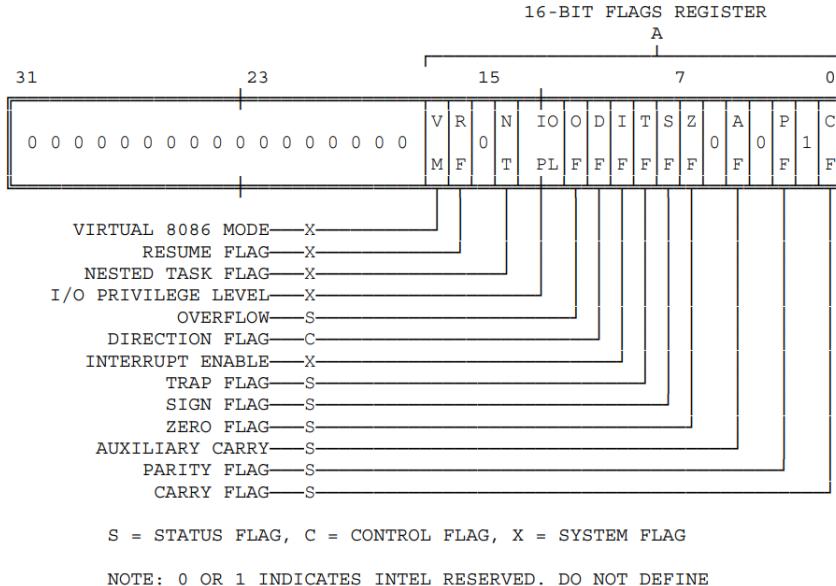
③使用 man 打开工程目录下的 Makefile 文件，你会在 CFLAGS 变量中看到 gcc 的一些编译选项。请解释 gcc 中的 -Wall 和 -Werror 有什么作用？为什么要使用 -Wall 和 -Werror？

问题1

首先解决第一个问题：**EFLAGS** 寄存器中的**CF**位是什么意思？

查找**EFLAGS Register**，在手册第34页可以，图2-8关于**EFLAGS**寄存器的结构示意图中可以找到CF位的含义如下。

Figure 2-8. **EFLAGS Register**



从图中可看出，CF位即**CARRY FLAG**，在**EFLAGS Register**中为进位标志位。

接下来解决第二个问题：**ModR/M**字节是什么？

搜索关键词**ModR/M**，由于问题设问点在“是什么”，即问**ModR/M**的定义，因此应该在偏前的章节中出现，向下查询关键词匹配结果，在i386手册第38页可以找到答案如下。

1. Most data-manipulation instructions that access memory contain a byte that explicitly specifies the addressing method for the operand. A byte, known as the modR/M byte, follows the opcode and specifies whether the operand is in a register or in memory. If the operand is in memory, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement. When an index register is used, the modR/M byte is also followed by another byte that identifies the index register and scaling factor. This addressing method is the most flexible.

翻译过来就是，大多数访问内存的数据操作指令都包含一个字节，该字节明确指定操作数的寻址方法。单字节，称为**modR/M**字节，跟在操作码之后，指定操作数是在寄存器中还是在内存中。

最后是第三个问题：**mov** 指令的具体格式是怎么样的？

同样，查询关键字**mov**，命令格式类的查询应在在i386关于指令介绍的章节中出现。手册第345页可以找到结果，问题答案参下。

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
C1iiiiii	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.

问题2

(1) 代码行数统计。

使用命令 `find . -name "*.[ch]" |xargs cat|wc -l`, 可知PA1中共有3952行代码。

```
sun@ests:~/Desktop/ics2018/nemu$ find . -name "*.[ch]" |xargs cat|wc -l
3952
```

使用 `find . -name "*.[ch]" |xargs cat|grep -v ^$|wc -l`, 过滤掉空格, 则PA1中共有3247行代码。

```
sun@ests:~/Desktop/ics2018/nemu$ find . -name "*.[ch]" |xargs cat|grep -v ^$|wc -l
3247
```

(2) 为使用 `make count` 计算总增添代码行数, 需要对 `makefile` 文件做以下修改。

```
56 count:
57     git checkout pa0
58     find . -name "*.[ch]" |xargs cat| wc -l
59     git checkout pa1
60     find . -name "*.[ch]" |xargs cat| wc -l
```

利用 `git add .` 和 `git commit` 命令提交修改后, 执行 `make count` 命令, 可以看到如下结果, 代码总共增加了 $3952 - 3487 = 465$ 行。

```
sun@ests:~/Desktop/ics2018/nemu$ make count
git checkout pa0
切换到分支 'pa0'
find . -name "*.[ch]" |xargs cat| wc -l
3487
git checkout pa1
切换到分支 'pa1'
find . -name "*.[ch]" |xargs cat| wc -l
3952
```

问题3

查看工程目录中 `Makefile` 文件中的 `CFLAGS` 参数如下。

```
15 CFLAGS    += -O2 -MMD -Wall -Werror -ggdb $(INCLUDES)
16
```

参数 `-Wall` 和 `-Werror` 的作用是提高代码安全性，便于代码维护和调试。其中，参数 `-Wall` 是开启所有警告，`-Werror` 是将所有警告当成错误处理，即在发生警告时停止编译操作，便于进行代码维护和 debug。

以上是本次实验流程，感谢批阅！
