PA1（上）-简易调试器

17307130178 宁晨然

一、概述

本次实验重心在于梳理 NEMU 框架，组织寄存器结构体，完成简易调试器中的几个功能。初步认识 NEMU 框架，活用 vim 进行编辑，以完成模拟调试的功能。

本次实验难度不大，主要困难在开始阅读代码时有些一头雾水，**实验内容全部完美完成，蓝色方框下的思考题全部做出**，用**“蓝框思考题 n”**标注。

二、实验过程

在开始之前，我认识到了虚拟化的过程。因为本次实验是建立在“windows 系统虚拟机中 linux 模拟的 NEMU”之上的，所以经历了两次模拟的过程。我在 windows10 上安装了 vmware15，并虚拟安装了 linux 系统 ubuntu18.0.4，在 linux 系统下模拟了 NEMU 系统，再在 NEMU 系统中运行程序，所以虚拟层次图如下。

——蓝框思考题 1：虚拟化的过程

“Hello World” Program
Micro operating system
Simulated x86 hardware
NEMU
GNU/Linux
Simulated computer hardware
VMware 15
Windows 10
Computer hardware

任务一：阅读框架代码

拿到 NEMU 这么大个项目，第一反应是无从下手，耗费了很多时间随意阅览 nemu 文件中的代码。虽然随意，但是大致清楚了每个文件下有哪些代码，摸清楚了整个框架的存放方式：**include 文件下几乎都是.h 文件的声名，src（source）文件下几乎都是.c 文件。**当然也看到了很多陌生的词汇，比如 watchpoint/expr/monitor，先置之不理。

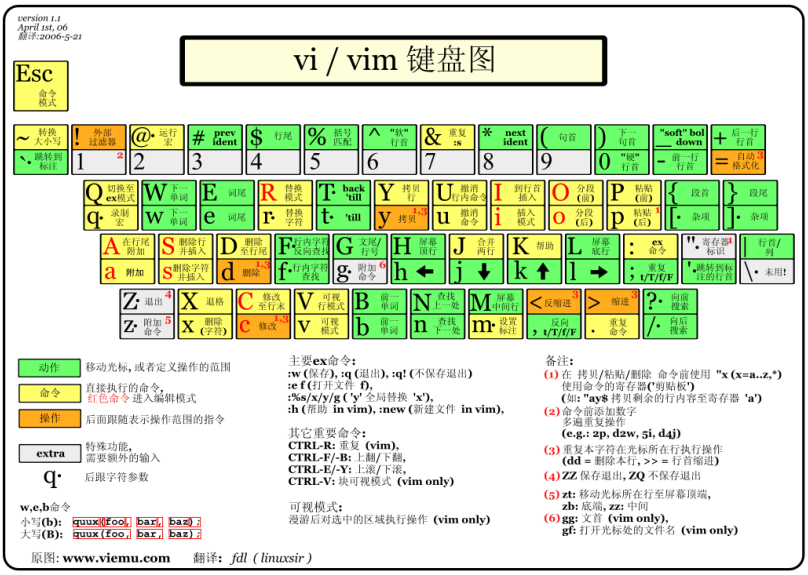
查看文件时，意外发现了 main.c，第二反应就猜到了“程序执行的开始”是从 main 函数执行。

——蓝框思考题 2：c 程序从哪里开始执行到哪里结束？

答：C/C++程序默认从 main 函数开始执行，但如果前方有类对象变量的声明，可能会先调用类的构造函数创建类对象。程序执行到 main 函数结束时结束。

——蓝框思考题 3：vim 快捷键盘图

答：掌握 vim 的快捷键着实增大了编辑文档效率。



（1）main 函数分析

main 函数很简洁，四个函数分工四项工作：①初始化 monitor②测试 CPU_state 结构启动③计算机启动④用户界面主循环。

先浅谈 restart()函数的理解。我的理解是，nemu 为了模拟计算机启动的环节，先调用 init_ramdisk()函数将真实磁盘中的可执行文件读入到了 ramdisk，而 ramdisk 为内存开始附近的一段磁盘。然后把 entry（不需要了解）读入到内存位置，并作为 eip 初值，最后做了 DRAM 的初始化。这里 restart 需要知道的是 eip 初值为 0x100000。

关键点是 ui_mainloop()函数的理解。我的理解是，nemu 类似于 linux 的命令台，每次会显示”nemu”等待用户输入指令，当输入指令（非结束指令 q）执行完毕后，又返回用户交互界面，继续等待下一个指令的输入，所以 nemu 界面是一个循环的主函数，即 ui_mainloop()。查阅 nemu/src/monitor/debug/ui.c 即可知道，这个函数使用 rl_gets()从键盘读入命令字符串，使用 strtok()函数将字符串拆分成命令，然后遍历 cmd_table.name 查找该命令，如果有则执行该命令，若返回值<0 则退出主循环，结束 nemu 界面，如果没有命令则输出错误。

此处解释一下 strtok()和 sscanf()函数：

①char *strtok(char s[], const char *delim);用于分解字符串为一组字符串，首次调用时，s 指向要分解的字符串，之后再次调用要把 s 设成 NULL，delim 为分隔符字符串。

②int sscanf(const char *buffer,const char *format,[argument]...);用于以固定字符串为输入源，转换为固定格式。

所以 ui_mainloop()函数的功能即：

①从键盘读入字符串②将字符串转换为命令字符串组③执行命令④结束或继续读入

（2）cpu_exec 函数分析

排除 ifdef debug 内容，可以得知主体成分是执行指令。

首先判断 nemu_state 是否为 running，即 nemu 是否在运作，如果已经停止工作，则需要先退出 nemu 再重新启动。然后是执行命令，此处调用 exec()函数来执行，之后我再分析。可以知道这里经历了，取指令、指令译码、最后执行三个步骤。再然后，是查看 watchpoints 的条件是否成立，这是后面需要完成的。最后，再修改 nemu_state，如果已经 stop 了，立刻返回，否则继续循环 for。for 循环结束后，将 nemu_state 置为 stop。

```
/* Execute one instruction, including instruction fetch,
 * instruction decode, and the actual execution. */
int instr_len = exec(cpu.eip);

cpu.eip += instr_len;
```

上图中是其中的执行命令步骤，可知先执行 eip 当前指向的指令，再将 eip 指向下一条指令。exec 即函数执行的函数了，它的位置需要细心找一下，在 nemu/src/cpu/exec/exec.c。

这里最关键的就是这个 volatile uint32_t n，它代表的是循环执行次数。可以在 cmd_c 中知道传入的参数为-1。如果传入 n 相当于执行 n 次，1 即单步执行。

——蓝框思考题 3：cmd_c()调用 cpu_exex()时传参-1？

答：-1 本身是带符号 int 类型，传入参数 uint32_t 类型时做了隐式类型转换，-1 即此时 uint32_t 的最大值，相当于循环最大次数，即继续执行接下来的所有指令，直到程序结束或遇到端点、watchpoints 条件不成立等，意思即 continue。

（3）exec 函数分析

这个函数其实是最难懂的，这应该也牵扯到 PA 后面的行程。我已经知道了 exec 是一个执行指令的函数，但是我找了半天都没找到一般的函数定义。只有一个类似的宏定义，在 nemu/include/cpu/helper.h 中：#define make_helper(name) int name(swaddr_t eip)。

```
make_helper(exec) {
    ops_decoded.opcode = instr_fetch(eip, 1);
    return opcode_table[ ops_decoded.opcode ](eip);
}
```

所以可以知道 make_helper(exec)的意思是 int exec(swaddr_t eip)，然后这个函数先取出 eip 所代表指令，保存在 opcode 中，可以推测 ops_decoded 是用于保存译码相关信息。然后执行 opcode_table[ops_decoded.opcode](eip)，并返回 int 值。此处不多纠缠这个函数，我相信之后的实验会对此处有详细的分析。相当于 opcode_table 是一个函数库，先执行函数，然后返回这个函数的返回值。[reference1]

——蓝框思考题 4：opcode_table 是个什么类型的数组？

*答:opcode_table 是 operationcode 操作码的映射数组。解释了各个汇编指令与 opcode 的关系。它的类型为 helper_fun，根据 grep 全局搜索发现，貌似并没有它的类型定义，除了一个奇怪的用法，typedef int (*helper_fun)(swaddr_t)。意思是 opcode_table 的数组中每个元素都是一个函数指针，这个函数为 int helper_fun(swaddr_t)，对应着某条指令的某种形式。说明类型是函数指针，都指向一个 helper 函数。*

之后的实验应该会实现所有指令，所以此处不纠缠。

```
chty627@ubuntu:~/ics2015$ grep -rn helper_fun*
nemu/src/cpu/exec/exec.c:6:typedef int (*helper_fun)(swaddr_t);
nemu/src/cpu/exec/exec.c:10:    static helper_fun concat(opcode_table_, name) [8] = { \
nemu/src/cpu/exec/exec.c:96:helper_fun opcode_table [256] = {
nemu/src/cpu/exec/exec.c:163:helper_fun _2byte_opcode_table [256] = {
```

（4）其他函数分析

三个对调试有用的宏，Log(),Assert(),panic()的定义和作用在讲义中已经很清晰，不予赘述。我查看了一下这三个宏的用处，并没有讲义中写的用这么广泛。只是之后的写代码，其实可以尝试着用一下。

```
src/monitor/cpu-exec.c:69:              Log_write("%s\n", asm_buf);
src/monitor/debug/expr.c:73:                    Log("match rules[%d] = \"%s\" at position %d
 with len %d: %.*s", i, rules[i].regex, position, substr_len, substr_len, substr_start);
```

```
include/memory/memory.h:16:     Assert(addr < HW_MEM_SIZE, "physical address(0x%08x) is out of bound
", addr); \
src/memory/dram.c:55:   Assert(addr < HW_MEM_SIZE, "physical address %x is outside of the physical m
emory!", addr);
src/memory/dram.c:76:   Assert(addr < HW_MEM_SIZE, "physical address %x is outside of the physical m
emory!", addr);
src/device/ide.c:134:   Assert(disk_fp, "Can not open '%s'", exec_file);
src/device/sdl.c:35:    Assert(ret == 0, "Can not set timer");
src/device/sdl.c:76:    Assert(ret == 0, "SDL_Init failed");
src/device/sdl.c:97:    Assert(ret == 0, "Can not set signal handler");
src/device/sdl.c:102:   Assert(ret == 0, "Can not set timer");
src/monitor/debug/expr.c:46:                    Assert(ret == 0, "regex compilation failed: %s\n%s",
 error_msg, rules[i].regex);
src/monitor/debug/elf.c:13:     Assert(argc == 2, "run NEMU with format 'nemu [program]'");
src/monitor/debug/elf.c:17:     Assert(fp, "Can not open '%s'", exec_file);
src/monitor/monitor.c:18:       Assert(log_fp, "Can not open 'log.txt'");
src/monitor/monitor.c:50:       Assert(fp, "Can not open '%s'", exec_file);
src/monitor/monitor.c:54:       Assert(file_size < ramdisk_max_size, "file size(%zd) too large", fil
e_size);
src/monitor/monitor.c:66:       Assert(fp, "Can not open 'entry'");
```

```
chty627@ubuntu:~/ics2015/nemu$ grep -rn panic*
include/debug.h:35:#define panic(format, ...) \
src/cpu/decode/decode-template.h:35:    panic("please implement me");
src/cpu/exec/logic/or-template.h:10:    panic("please implement me");
src/cpu/exec/logic/xor-template.h:10:   panic("please implement me");
src/cpu/exec/logic/and-template.h:10:   panic("please implement me");
src/cpu/exec/arith/dec-template.h:10:   panic("please implement me");
src/cpu/exec/arith/inc-template.h:10:   panic("please implement me");
src/device/i8259.c:46:          panic("uncomment the line above");
src/device/i8259.c:59:  panic("uncomment the line above");
src/monitor/debug/expr.c:82:                                    default: panic("please implement me"
);
src/monitor/debug/expr.c:105:   panic("please implement me");
```

其次是 swaddr_read() 和 swaddr_write()，在 nemu/src/memory/memory.c 中，其实这几个类型 hwaddr/lnaddr/swaddr 的实质类型都一样，只是模拟了不同的空间。为了保持变量统一方便，内存处理采用下面两个函数。

```c
uint32_t swaddr_read(swaddr_t addr, size_t len) {
#ifdef DEBUG
    assert(len == 1 || len == 2 || len == 4);
#endif
    return lnaddr_read(addr, len);
}

void swaddr_write(swaddr_t addr, size_t len, uint32_t data) {
#ifdef DEBUG
    assert(len == 1 || len == 2 || len == 4);
#endif
    lnaddr_write(addr, len, data);
}
```

——蓝框思考题 5：谁来指示程序的结束？

答：程序正常执行到 main()函数的 return 部分即结束，实质是先退栈 pop，再调用 ret 指令。例如下图是 hello.c 的汇编语言，可以发现退栈后 ret，将栈顶返回的地址弹出到 eip，然后 eip 按照该指令继续执行程序。但是 main 函数没有调用相，所以我猜测 eip 此时指向空指令或 NULL，此时无法进行继续程序，所以程序结束。（仅猜测，还需要多学知识）

```
Dump of assembler code for function main:
   0x00000000004004fd <+0>:     push   rbp
   0x00000000004004fe <+1>:     mov    rbp,rsp
   0x0000000000400501 <+4>:     mov    edi,0x4005a4
   0x0000000000400506 <+9>:     mov    eax,0x0
   0x000000000040050b <+14>:    call   0x4003f0 <printf@plt>
   0x0000000000400510 <+19>:    pop    rbp
   0x0000000000400511 <+20>:    ret
End of assembler dump.
```

任务二：重新组织寄存器结构体

这个任务非常简单，因为之前直接运行 nemu 会发现 assertion failure，实质是 struct 的结构有问题。首先通用寄存器们中，eax/ax/al 实质使用了同一个物理空间，很容易想得到应该是 union 类型的结构，然后由于八个寄存器相对独立，所以八个寄存器应该在 struct 中。还有程序计数器 eip，应该也是不同于八个通用寄存器的物理空间。在 nemu/include/cpu/reg.h 中的 CPU_state 就应该代表了寄存器的状态，只需要稍作修改一下空间分配情况，如下图，就可以实现 eax/ax/al 用于同一个物理空间，eax/ecx/edx/eip 等用不同的物理空间。这个 union 其实用了 9 个 32 位。

```c
typedef union {
    union {
        uint32_t _32;
        uint16_t _16;
        uint8_t _8[2];
    } gpr[8];

    /* Do NOT change the order of the GPRs' definitions. */
    struct
    {
        uint32_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
        swaddr_t eip;
    };

} CPU_state;
```

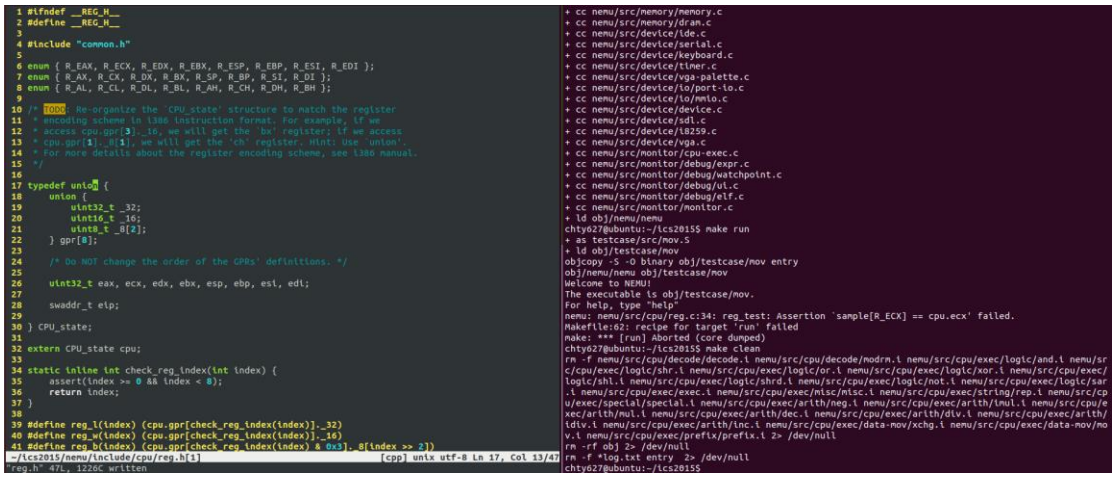
写完之后更新 git-log，先使用 git status 查看更新内容，再用 git commit -a 可以将这个修改的内容添加备注后放进 git 仓库中。使用 make run 运行 nemu 后，键入 c 可以得到正确结果，如右图。

```
commit 0c6817493d14377901e6600ad799a1375c0da5e9
Author: 17307130178-Ning Chenran <17307130178@fudan.edu.cn>
Date:   Sun Mar 24 13:53:27 2019 +0800

    modified cpu_state using union
```

```
chty627@ubuntu:~/ics2015$ make run
objcopy -S -O binary obj/testcase/mov entry
obj/nemu/nemu obj/testcase/mov
Welcome to NEMU!
The executable is obj/testcase/mov.
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x001002b1
```

任务三：为简易调试器添加功能

在开始之前，我发现任务三仅在 nemu/src/monitor/debug/ui.c 实现，那么现在使用分屏设置可以方便很多，左边屏幕查看 ui.c 进行更新，右边在主目录下进行 nemu 界面的测验。tmux 的插件使用起来极其方便。



该任务有三个小任务。①单步执行②打印寄存器③扫描内存

（1）单步执行 si N

需要添加两个部分，首先查看整个代码发现 cmd_table 中存放了所有指令的“名字/解释/函数”，所以先在 cmd_table 把所有的标准化的格式写好，可以省很多时间。

```c
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 },
    { "si", "Step N instructions exactly", cmd_si },
    { "info", "Print the state of registers or watchpoints", cmd_info},
    { "p", "Print value of expression EXPR", cmd_p},
    { "x", "Examine memory:x ADDRESS", cmd_x},
    { "w", "Set a watchpoint for an expression", cmd_w},
    { "d", "Delete N.th watchpoint", cmd_d},
    { "bt", "Print backtrace of all stack frames", cmd_bt}
    /* TODO: Add more commands */

};
```

第一次做的时候，关键点自然放在了 int (*handler) (char*)，这是一个函数，参数为字符串、返回值为 int，可以借鉴 cmd_help 和 cmd_c 看的出来这个函数应该怎么写，只需要模仿构造剩下的所有函数即可。我就先把函数一次性填充完整了。例如下图。由之前的分析可以知道这个 cmd_table 是用于 ui_mainloop 取指令用的，返回值告诉程序是否继续，只需要返回>=0 的数即可继续进行，所以先全部写返回 0。

```
static int cmd_bt(char *args)
{
    return 0;
}
```

si N 可以使用 gdb 中的 help si 查看解释，填写在 cmd_table 的解释中，其他的类似，均可以使用 help 查看手册中的解释。

cmd_si 代码如下：

```
static int cmd_si(char *args)
{
    char *arg = strtok(NULL," ");   //like cmd_help,get next instruction
    int n = 0, i = 0;
    if(arg == NULL) {cpu_exec(1);return 0;} //defalt state n = 1
    if(sscanf(arg,"%d",&n)!=0){
        if(n<=0)    printf("Invalid number \"%s\"\n",arg);
        else for(;i<n;i++) cpu_exec(1);
    }
    else printf("No symbol \"%s\" in current context.\n",arg);
    return 0;
}
```

思路：由 cmd_help 可以参考到 strtok/args 的用法，猜测到 args 应该是保存下来的命令字符串，需要先拆分第一个单词"si"，然后拆去" "，剩下如果有参数则转换为十进制，表示 N（单步执行的次数），如果没有参数则默认单步执行一次。

①当 si 后面没有参数时，默认 N=1。单步执行并返回 0。

②当 si 后面有参数时，读取 N 参数。

如果成功读取数字 N：若 N<=0 则报错"非法数字"；若 N>=0 则循环 N 次单步执行。

否则报错"非法字符串"。

所以分情况讨论 arg==NULL。如果是非空，则运用 sccanf 将其转换为十进制，用 for 循环实现单步执行 cpu_exec(1)。这里没有使用 cpu_exex(n)，怕这样会出些 bug，反正循环 n 次肯定不会错。最后返回值需要非负，因为没有强制结束程序。

报错：此处特意设计了报错系统。我在 gdb 中查看了如果发生错误的情况，有两种。

①数字错误　②字符串报错

更新 git/测试程序。

测试结果：

```
commit dad72cb9739595759f2e622b91f5f1740c8506b4 (HEAD -> master)
Author: 17307130178-Ning Chenran <17307130178@fudan.edu.cn>
Date:   Sun Mar 24 14:18:33 2019 +0800

    realize cmd_table cmd_si
```

```
(nemu) si
  100000:   b8 00 00 00 00                      movl $0x0,%eax
(nemu) si 2
  100005:   bb 00 00 00 00                      movl $0x0,%ebx
  10000a:   b9 00 00 00 00                      movl $0x0,%ecx
(nemu) si -1
Invalid number "-1"
(nemu) si jkl
No symbol "jkl" in current context.
```

（2）打印寄存器 info r

代码如下：

```c
static int cmd_info(char *args)
{
    char *arg = strtok(NULL," ");
    if(arg==NULL){
        printf("Try \"info r\" or \"info w\".\n");
        return 0;
    }
    if(strcmp(arg,"r") == 0)
    {
        int i;
        for(i=0;i<8;i++){
            if(strcmp(regsl[i],"ebp") == 0 || strcmp(regsl[i],"esp") == 0)
                printf("%s\t0x%08x\t0x%08x\n",regsl[i],reg_l(i),reg_l(i));
            else
                printf("%s\t0x%08x\t%d\n",regsl[i],reg_l(i),reg_l(i));
        }
        printf("%s\t0x%08x\t0x%08x\n\n","eip",cpu.eip,cpu.eip);
        for(i=0;i<8;i++)
            printf("%s\t0x%08x\t%d\n",regsw[i],reg_w(i),reg_w(i));
        for(i=0;i<8;i++)
            printf("%s\t0x%08x\t%d\n",regsb[i],reg_b(i),reg_b(i));
    }
    else if(strcmp(arg,"w") == 0)
    {
        //leave it now
    }
    else printf("Undefined info command: \"%s\".Try \"help info\".\n",arg);
    return 0;
}
```

思路：

本来这里需要实现 info r/w，暂时还没有涉及到 watchpoints，我看了半天也没怎么懂 watchpoints，放到之后实现。同样先用 strtok 把空格去掉。这里需要打印寄存器的状态，我先进入 reg.h 中看了看，果然有直接实现取值和取字符串的定义（见下图）。

```c
#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])

extern const char* regsl[];
extern const char* regsw[];
extern const char* regsb[];
```

可以简易实现取当前寄存器的名称和值。接下来只需要三次 for 循环格式输出即可，此处使用了 0x%0x8x 的格式，保证了十六进制数的规范化。当然 eip 是不包含在上面的，需要单独输出。条件分析如下：

①如果无子命令，提示“info r”或“info w”

②如果子命令为 r，打印寄存器。输出寄存器的十六进制与十进制。

32 位寄存器中 esp/ebp/eip 只输出十六进制。

③如果子命令为 w，打印 watchpoints。

④如果子命令非法，提示非法命令。

报错：需要单独处理无子命令和子命令非法的情况。

测试结果：

```
(nemu) info
Try "info r" or "info w".
(nemu) info w
(nemu) info ru
Undefined info command: "ru".Try "help info".
(nemu) info r
eax     0x013005f0      19924464
ecx     0x6efe0e72      1862143602
edx     0x097d47f5      159205365
ebx     0x7d89fb47      2106194759
esp     0x414299bb      0x414299bb
ebp     0x05c7268f      0x05c7268f
esi     0x6b7f7595      1803515285
edi     0x7b0a0223      2064253475
eip     0x00100000      0x00100000
```

```
ax      0x000005f0      1520
cx      0x00000e72      3698
dx      0x000047f5      18421
bx      0x0000fb47      64327
sp      0x000099bb      39355
bp      0x0000268f      9871
si      0x00007595      30101
di      0x00000223      547
al      0x000000f0      240
cl      0x00000072      114
dl      0x000000f5      245
bl      0x00000047      71
ah      0x00000005      5
ch      0x0000000e      14
dh      0x00000047      71
bh      0x000000fb      251
```

git 更新：（实质上更新了 3 次，后来添加修改了报错系统）

```
commit 74a64a0d235207c7ae2f21d4e1ae19ea2dd47f59
Author: 17307130178-Ning Chenran <17307130178@fudan.edu.cn>
Date:   Sun Mar 24 15:28:01 2019 +0800

    set the rest of the work frames

commit 2c2e5e4c12e55140bc8b9fa9a0be7e5b1e5dc16d
Author: 17307130178-Ning Chenran <17307130178@fudan.edu.cn>
Date:   Sun Mar 24 15:06:57 2019 +0800

    realize cmd_table cmd_info
```

（3）扫描内存 x

代码如下：

```
static int cmd_x(char *args)
{
    int n,i;
    swaddr_t addr;
    char *arg1 = strtok(NULL," ");
    char *arg2 = strtok(NULL," ");
    if(arg2==NULL){
        sscanf(arg1,"%x",&addr);
        printf("0x%08x:\n0x%08x",addr,swaddr_read(addr,4));
    }
    else{
        if(sscanf(arg1,"%d",&n)==0){
            printf("No symbol \"%s\" in current context.\n",arg1);
            return 0;
        }
        if(sscanf(arg2,"%x",&addr)==0){
            printf("A syntax error in expression, near '%s'.",arg2);
            return 0;
        }
        if(n<=0) {printf("Please enter a positive number.\n");return 0;}
        printf("0x%08x:\n",addr);
        for(i=0;i<n;i++,addr+=4){
            printf("0x%08x%c",swaddr_read(addr,4),i%4==3?'\n':' ');
        }
    }
    printf("\n");
    return 0;
}
```

思路：

之前提到过为了保持一致性，都使用 swaddr_t 作为取值比较方便。由于表达式还没写，先写一个立即数简单版本的，这里有两个 arg，所以需要两次 strtok 和 sscanf。之后取内存时只需要 swaddr_read()函数即可，同时每次需要把地址+4，用 for 循环即可实现。

①获取 x 后的两个参数 arg1 和 arg2

②如果 arg2 缺失，则默认扫描 arg1 附近的 1 个单位的内存

③如果 arg1 与 arg2 均合法，则按照格式输出内存。

若 arg1 不合法，报错“没有数字”或“数字非法”；若 arg2 不合法，报错“表达式错误”

报错：

这里需要处理 arg1/arg2 缺失和不合法的错误，分情况讨论即可。这里也是借鉴了 gdb 当中的错误提示。

测试结果：

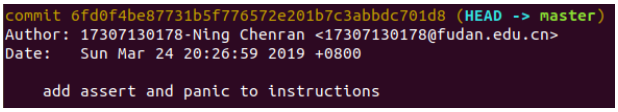
```
(nemu) x
Try "x fmt expr".
(nemu) x 0x100000
0x00100000:
0x000000b8
(nemu) x 2 0x100000
0x00100000:
0x000000b8 0x0000bb00
(nemu) x jk 0x100000
No symbol "jk" in current context.
(nemu) x 2 jk
A syntax error in expression, near 'jk'.
(nemu) x -1 0x100000
Please enter a positive number.
```

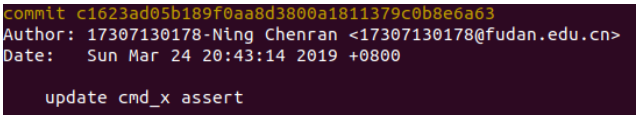
```
commit e63227432e29adb92f564873ef91e66065f5d39c
Author: 17307130178-Ning Chenran <17307130178@fudan.edu.cn>
Date:   Sun Mar 24 15:58:15 2019 +0800

    realize cmd_table cmd_x
```

同样也写进 git log。（实质做了三次更新，更新了报错系统）

（4）assert 和 panic

其实这个是后来才想起来一定要保证正数/字符串非空，所有后来做了两次更新。





后来发现如果只是使用 Assert 和 panic 函数的话，会使 nemu 直接退出 ui 界面，这其实与实际使用中的 gdb 不一样。gdb 遇到错误时，它会提醒你哪里错了，然后返回 ui 界面。所以我在第三次更新中，将报错系统全部替换成了 if 语句和 printf 语句，这样既可以报错又可以返回正常界面进行接下来的操作。并且优化了很多很多细节。包括最后了还在修改。

归纳特点即：

①系统 ui 会提示出错问题。

②系统 ui 会提示解决方法。

③系统 ui 会返回 ui 界面，让用户再次尝试/继续下次操作。

三、实验感受

任务一阅读框架代码，让我有机会认识到，从别人手中接手/开源代码开始，如何从零接触一段没有写过的框架代码。框架代码的核心就是看懂框架图，然后划分重要性，再按照顺序阅读，比如这次从 main 开始就可以一步一步了解 nemu。

任务二很简单，就是一个结构体的应用和 cpu 的理解。任务三则是直观理解了 gdb 的使用方法，因为这次的简易调试器功能基本都是模仿 gdb 做出来的。使用 gdb 的 help 可以很清晰的看到每个指令的深刻理解了 vim 的使用方法，这次实验全部在 ubuntu 的 terminal 的 tmux 情况下做出来的，熟悉了 vim 和 tmux 的操作，分屏查看代码和测试代码非常方便。然后是 git log 的使用，能够很清晰的查看到自己的修改痕迹，每次可以记录下修改的版本，以后要坚持使用。

PA 实验是介入别人的框架填写代码，阅读代码的能力不容忽视，相信在下次实验中也会更加理解这个框架代码的运作。

期待下次与你相见。

[reference]

reference1：nemu 如何执行指令？

https://max.book118.com/html/2017/0315/95583100.shtm