

# 华中科技大学

## 2019

### 系统能力综合训练 课程设计报告

题 目： X86 模拟器设计

专 业： 计算机科学与技术

班 级： CS1705

学 号： U201714740

姓 名： 张智璐

电 话： 15619796392

邮 件： 910688714@qq.com

完成日期： 2021-1-11



# 目 录

1	课程设计概述 .....	1
1.1	课设目的 .....	1
1.2	课设任务 .....	1
1.3	实验环境 .....	1
2	实验过程 .....	2
2.1	PA0 .....	2
2.2	PA1 .....	2
2.2.1	总体设计 .....	2
2.2.2	详细设计 .....	3
2.2.3	运行结果 .....	7
2.2.4	问题解答 .....	8
2.3	PA2 .....	9
2.3.1	总体设计 .....	9
2.3.2	详细设计 .....	10
2.3.3	运行结果 .....	12
2.3.4	问题解答 .....	13
2.4	PA3 .....	14
2.4.1	总体设计 .....	14
2.4.2	详细设计 .....	14
2.4.3	运行结果 .....	17
2.4.4	问题解答 .....	19
3	设计总结与心得 .....	20
3.1	课设总结 .....	20
3.2	课设心得 .....	20
	参考文献 .....	22

# 1 课程设计概述

## 1.1 课设目的

探究“程序在计算机上运行”的机理，掌握计算机软硬协同的机制，进一步加深对计算机分层系统栈的理解，梳理大学 3 年所学的全部理论知识，提升学生计算机系统能力。

## 1.2 课设任务

在代码框架中实现一个简化的 x86 模拟器 (NJU EMUlator)

- 可解释执行 X86 执行代码
- 支持输入输出设备
- 支持异常流处理
- 支持精简操作系统---支持文件系统
- 支持虚存管理
- 支持进程分时调度

最终在模拟器上运行“仙剑奇侠传”。

## 1.3 实验环境

- 系统：Ubuntu 18.04.5 LTS
- gcc 版本：9.1.0
- 内存：4GB
- 处理器内核总数：8

## 2 实验过程

### 2.1 PA0

PA0 是配置实验环境，由于我是用 vmware 自建虚拟机，没有使用老师给的 vdi，所以需要自己去按照指导手册里面的步骤安装工具。另外还更新了 gcc 到 9.1.0 版本解决了 unrecognized command line option ‘-mmanual-endbr’ 的问题。一开始我偷了个懒，工具没有全部安装就跳过了 PA0，结果做到 PA2 的时候 diff-test 发现缺少了 qemu，当时调试了好久，最后才发现是 PA0 的时候跳过了根本就没有装 qemu。所以自建还是不如直接用老师给的 vdi 镜像(后来同学问我，我都是直接推荐他们装 virtualbox 使用 vdi 镜像)。

### 2.2 PA1

#### 2.2.1 总体设计

PA1 的内容分为 3 个部分，不过核心是实现手册里面的基础设施中的调试器的一些功能，具体功能如表 2.1。

表 2.1 PA1 命令

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

(1) 命令已实现

(2) 与 GDB 相比，我们在这里做了简化，更改了命令的格式

### 2.2.2 详细设计

在正式开始 PA1 前，由于我选择的是 ISA 是 x86，因此需要修改 reg.h 中对 CPU\_state 的定义：

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

因为原先定义的是 struct，gpr 数组和 8 个寄存器并没有关联，物理上是顺序独立的排列，为了让他们相关联起来，就要使用 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;
        };
    };
    vaddr_t pc;
} CPU_state;
```

修改好以后，在 menu 目录下执行 make run，看到欢迎提示说明修改正确。

PA1 要在 ui.c 中补全表 2.1 的命令，增加一条新的命令需要先写一个对应的函数，然后在 cmd\_table 中增加命令的名称，说明，函数。

#### (1) PA1.1

##### 1) si 命令

si 命令用 strtok 函数读入输入的 N，转换为数字后，作为参数传给 cpu\_exec()，cpu\_exec(n) 是执行 n 步程序。没有输入 N 的话默认调用 cpu\_exec(1) 执行 1 步。

##### 2) x 命令

x 命令需要用到 vaddr\_read(addr,n) 函数，其中参数 addr 是内存地址，n 是读出的字节数，思路和刚刚一样，先用 strtok 函数读出命令的 2 个参数 N 和 exp，这里暂时不考虑复杂的表达式，仅限于 16 进制数字，所以可以直接使用 sscanf(arg2, "%x", &addr); 将第 2 个参数地址得到，然后再用 vaddr\_read(addr,4) 每次读出 4 个字节的内存，循环 N 次即可。

##### 3) info r 命令

在 reg.c 中实现 isa\_reg\_display()，其实只需要使用 printf 按照一定格式打印

所有寄存器值。当判断 info 命令的第 2 个参数为 r 的时候调用 isa\_reg\_display 即可。

(2) PA1.2

上面 3 个命令的实现是 PA1.1 的内容，相对比较简单，PA1.2 实现最难的 p 命令，对表达式求值。

首先对于一个表达式，要用正则表达式识别其中的符号 token 才能进一步分析。用到的 token 如表 2.2 所示。

表 2.2 所有 token 一览

名称	正则表达式	意义	优先级
TK_NOTYPE	+	空格	0
TK_NUM	[0-9]+	十进制数字	0
TK_HEX	0[Xx][0-9a-fA-F]+	十六进制数字	0
TK_REG	\\$[a-zA-Z]{2,3}	寄存器	0
TK_OR	\ \		1
TK_AND	&&	&&	2
TK_EQ	==	==	3
TK_PLUS	\+	+	4
TK_SUB	-	-	4
TK_MUL	\*	*	5
TK_DIV	/	/	5
TK_NEGTIVE	-	负数	6
TK_DEREF	\*	引用 (例:*\$eax)	6
TK_LBR	\(	(	7
TK_RBR	\)	)	7

名称用于在 expr.c 中作为枚举常量，优先级是为了方便后续符号之间的运算顺序比较引入，为此需要修改源代码中 rule 和 Token 结构体的定义，增加 priority 成员。按照表 2.2 的正则表达式，符号名，优先级填写好 rules 数组，在 make\_token 中会对表达式挨个字符进行分析，从第一条规则开始遍历 rules，匹配成功以后，首先对匹配字符串的长度检测，判断是否大于 32，如果大于 32 个字符 assert(0)，否则的话，根据匹配的规则，分为 3 类处理：

- 空格：不处理。
- 数值类：TK\_NUM,TK\_HEX,TK\_REG 三个，新增一项 tokens 数组成员，保存类型和优先级，并保存匹配的字符串等待后续解析。
- 其他：新增一项 tokens 数组成员，保存类型和优先级。

到这里一个表达式已经被解析为一个一个符号并保存在了 tokens 数组中，接下来需要按照手册提示完成求值函数。

完成手册中提示的 eval() 函数之前，还需要写 2 个辅助函数——check\_parentheses()和 get\_main\_op()。

int check\_parentheses(int p, int q)，这里将返回类型由原来的 bool 改为 int，因为需要返回 3 种不同情况。参数 p 和 q 分别是 tokens 数组的起始和结束位置。

函数的功能是根据 `tokens` 数组从 `p` 到 `q` 这部分所代表的表达式返回以下 3 种：

- 返回 1 如果表达式是一个由一对左右括号完整包裹的一个有效表达式。
- 返回 0 如果表达式是一个有效的表达式，但是并没有被一对左右括号包裹。
- 返回-1 如果表达式不是一个有效的表达式。

思路是从 `p` 位置开始到 `q` 位置，用 `n` 进行记录（`n` 初始为 0），如果遇到左括号 `n` 自增，如果遇到右括号 `n` 自减，在循环过程中，只要 `n` 变为负数可以立即退出返回-1，如果循环结束 `n` 不为 0 也返回-1，当然还要检测是否 `p` 和 `q` 位置是一对括号。

另外值得一提的是这里需要考虑一种特殊情况，就是虽然是一个有效的表达式，同时左右分别是左括号和右括号，但是这两个括号并不是一对，例如：

$(4 + 3) * (3 + 4)$

对于这样的表达式，应该返回 0 而不是 1。也就是说需要对括号的匹配性进行检测，思路与上面类似，不过从 `p+1` 到 `q-1` 开始循环记录 `n` 值。

`int get_main_op(int p, int q)`，函数的作用是从 `p` 位置开始扫描到 `q` 位置，找到一个优先级最低的运算符且该运算符不能在括号包裹的表达式里面，同优先级的情况下取最右边的运算符作为主符号，最后将主符号的位置返回。因为 `tokens` 里面有存各个符号的优先级，所以这里就可以非常简单的实现。判断括号内的方法可以参考上面的用 `n` 记录的方法，只有当 `n` 等于 0 的时候说明不在括号内，这时候再进行对运算符优先级的比较。

两个辅助函数写完的情况下，可以开始写 `eval` 函数求值了，但是在此之前还有 2 个 `token` 没有识别，分别是负数和引用，因为他们的符号与减和乘相同，因此需要联系前面的符号判断。在 `expr` 函数中对 `tokens` 数组扫描，对于其中的\* 也就是 `type` 是 `TK_MUL`，如果位置在第一个，或者他的前一个符号不是数字(可以用优先级简单的判断，因为数字的优先级是 0)，且不是右括号，那么将这个 `TK_MUL` 重新解释为 `TK_DEREF`，优先级为 6。同理对 `TK_NEGTIVE` 也进行类似的判断识别。现在所有符号都正确的识别了，可以使用 `eval` 对其进行求值了。

`uint32_t eval(int p, int q, bool *success)`，求出 `tokens` 数组从 `p` 到 `q` 所代表的表达式的值，如果求值成功 `*success==true`，失败 `*success==false`，返回求值的结果。`eval` 的大致框架和思路在手册里面有写，算法流程图如图 2.1。

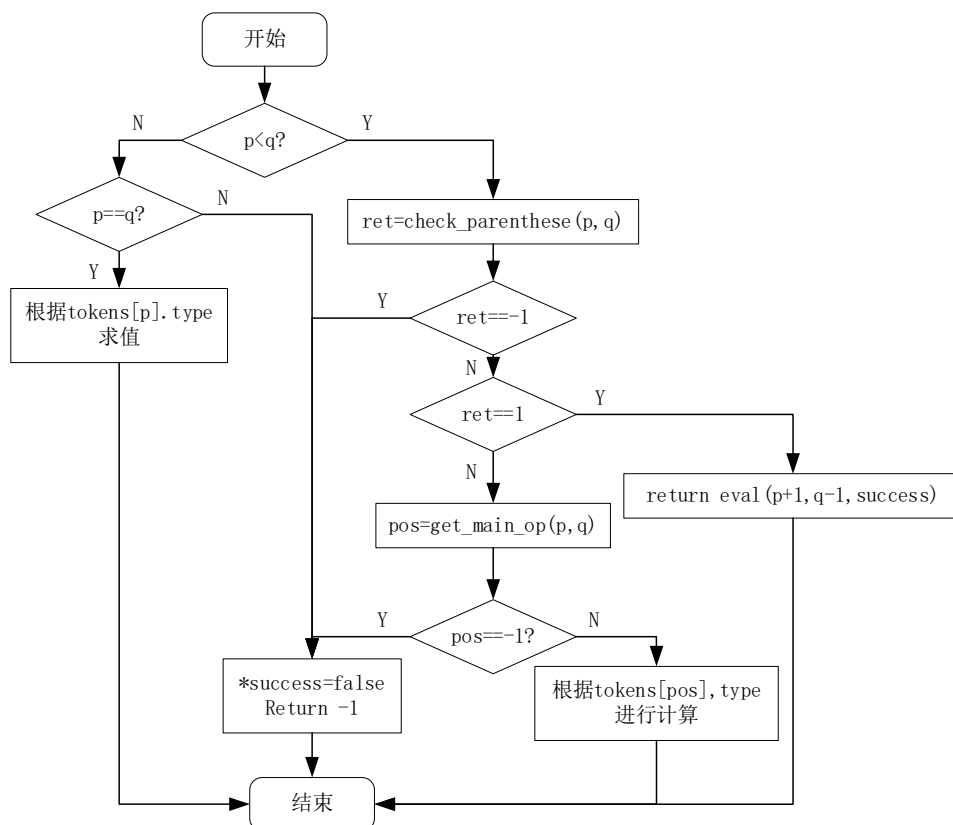


图 2.1 eval 算法流程图

大致思路和流程图中一样，首先是  $p==q$  的时候，这时候说明这个表达式没有运算符，其本身是一个数值或者寄存器，如果是 `TK_NUM` 或者 `TK_HEX`，那么直接 `strtoul(tokens[p].str, NULL, 0)` 就可以把字符串转化为数值，如果是 `TK_REG`，那么需要使用 `isa_reg_str2val(tokens[p].str + 1, success)` 获取寄存器的值。`isa_reg_str2val()` 函数在 `reg.c` 中，实现方法是遍历所有的寄存器，比对传入参数 `s` 和寄存器的名字，找到 `s` 对应的寄存器然后返回寄存器中的值。

另外处理一下除以 0 的异常行为。也就是在处理主运算符是除法的时候，如果 `val2==0`，输出一段提示后，`*success=false`。

最后 `TK_DEREF` 和 `TK_NEGTIVE` 可以提前单独运算，因为没有用到 `val1`，分别是 `val=vaddr_read(val2,4)` 和 `val=-val2`。

为了检测表达式求值的正确性，手册要求完成 `gen-expr.c` 实现一个随机测验，原理在手册中已经给出，而需要补充实现的是其中的随机生成表达式部分。

`void gen_rand_expr(int l, int r)` 在字符串位置 `l` 和 `r` 之间随机生成表达式的 `token`，如果 `r==l` 生成一个 1 位数字，如果 `r==l+1` 生成一个两位数，此外，从下面 2 种情况中生成一种：

- 在 `l+1` 到 `r-1` 中随机选取一个位置作为主符号的位置 `op`，并从加乘除当中随机选择一个符号。递归调用 `gen_rand_expr(l, op-1)` 和 `gen_rand_expr(op+1, r)` 生成两边的表达式。
- 在 `l` 和 `r` 的位置生成一对括号，递归调用 `gen_rand_expr(l+1, r-1)`，生成被一对括号包裹的表达式。

最后修改下面的 `main` 函数中对 `gen_rand_expr()` 的调用，加上参数，就完成



了。

检测没问题的情况下，只需要在 `p` 命令的函数中提取命令的参数调用 `expr` 函数求值即可。

### (3) PA1.3

PA1.3 要求实现监视点的创建,删除和展示。监视点采用链表池的结构管理,所有监视点保存在 `wp_pool` 数组中,而正在使用的监视点由 `head` 指针串起的一个链表,空闲的监视点链表由 `free_` 指向。监视点结点中除了保存序号 `NO` 和 `next` 指针,还需要额外增加 3 个成员:

- `char expr[128]`: 用于记录创建的监视点的表达式,例如 `$eax`。
- `uint32_t value`: 求出当前的表达式的值,表达式是不变的,但是其值会变,因为寄存器值会变,当表达式值改变的时候就要暂停程序,算是命中。
- `int hit`: 记录该监视点的命中次数。

主要操作都是链表的基本操作,创建一个监视点,就是将 `free_` 指向的首结点从空闲链表摘除,加进 `head` 指向的链表。并设置其 `expr, hit`, 计算 `expr`。而删除一个监视点,就是遍历 `head` 指向的链表,找到一个结点其序号 `NO` 与给定的相符,将其从 `head` 链表中删除,移到 `free_` 链表。调用相关函数即可完成 `w` 命令和 `d` 命令。而 `info w` 就是遍历 `head` 链表,按一定格式输出结点中保存的信息。

除此之外,当监视点监视的表达式值变化时,需要暂停程序。首先实现 `check_wp()`, 它的功能是检查所有监视点,看是否有监视点的表达式值发生了变化。方法很简单,遍历 `head` 指向的所有正在使用的监视点,然后重新计算其 `expr` 的值,与之前保存在 `value` 当中的值比对,如果不同的话,说明表达式的值发生了变化。然后需要暂停程序,这里手册给了提示,只要把 `nemu_state.state` 赋值为 `NEMU_STOP` 即可。

这样,所有命令就完成了, PA1 结束。

### 2.2.3 运行结果

按照 PA1 测试用例进行测试,测试结果如图 2.2 图 2.3 图 2.4 所示。运行正常。

```
Welcome to x86-NEMU!
For help, type "help"
(nemu) si
100000: b8 34 12 00 00          movl $0x1234,%eax
(nemu) si 2
100005: b9 27 00 10 00          movl $0x100027,%ecx
10000a: 89 01                  movl %eax,(%ecx)
(nemu) info r
eax      0x1234          4660
ecx      0x100027        1048615
edx      0x23e41e2b      602152491
ebx      0xdf64c6        14640326
esp      0x267982a3      645497507
ebp      0x688ac12b      1753923883
esi      0x85e2c3f       140389439
edi      0x26afa01c      649043996
pc       0x10000c        1048588
(nemu) q
yuirito@ubuntu:~/ics2019/nemu$
```

图 2.2 PA1 测试 1

```

(nemu) w $eip == 0x100005
watchpoint 0 : $eip
(nemu) w $eax
watchpoint 1 : $eax
(nemu) w $ecx
watchpoint 2 : $ecx
(nemu) info w
Num      What      Value
2        $ecx      68949755(0x41c16fb)
1        $eax      1377791271(0x521f6d27)
0        $eip      1048576(0x100000)
(nemu) d 2
(nemu) si 2
    100000:  b8 34 12 00 00                movl $0x1234,%eax
Watchpoint 1: $eax
Old value = 1377791271
New value = 4660
Watchpoint 0: $eip
Old value = 1048576
New value = 1048581
(nemu) si 2
    100005:  b9 27 00 10 00                movl $0x100027,%ecx
Watchpoint 0: $eip
Old value = 1048581
New value = 1048586

```

图 2.3 PA1 测试 2

```

(nemu) p (1+2)*(4/3)
0x3(3)
(nemu) p (3/3)+(123*4
[src/monitor/debug/expr.c,242,eval] check_parentheses return -1
Invalid expr
(nemu) p (1+(3*2) +(($eax-$eax) +(*$eip-1**$eip)+(0x5--5+ *0x100005 - *0x100005
) ) *4)
0x2f(47)
(nemu)

```

图 2.4 PA1 测试 3

## 2.2.4 问题解答

### 必答题

- 送分题 我选择的 ISA 是 x86。
- 理解基础设施
  - 假设完成 PA 需要编译 500 次，90%用于调试，即 450 次调试，如果没有实现简易调试器，需要花费 30s 从 GDB 获取并分析一个信息，并且 20 个信息才能排除一个 bug，那么排除一个 bug 所需时间为  $20 \times 30 = 600s$ ，如果一次调试排除一个 bug，总共需要  $450 \times 600 = 270000s = 75h$ ，即在调试方面需要花费 75 小时。
  - 使用简易调试器，可以节约 2/3 的时间，也就是  $75 \times 2/3 = 50h$ 。
- 查阅手册
  - 2.3.4 Flags Register
  - 17.2.1 ModR/M and SIB Bytes
  - 3.1 Data Movement Instructions
- shell 命令
  - nemu/ 下共有 5479 行代码。用的命令是 `find . -name`



## 2.3.2 详细设计

### (1) PA2.1

在 PA2.1 中根据反汇编结果需要实现 5 条指令, 分别是 call, push, sub, xor, ret。

#### 1) call 指令

查看反汇编结果, 可以看到有 2 条 e8 cd 的形式的 call 指令, e8 是 opcode, 后面 4 个字节是偏移量, 跳转的地址是 pc+指令长度+偏移量。

首先要去填写 opcode\_table[0xe8], 指示它的译码函数和执行函数, 这个需要提前查看所有的译码函数选择恰当的一个, 这里选择 decode\_J(), 执行函数也是类似, 选择 exec\_call(), 位宽是 4 个字节, 因此

```
opcode_table[0xe8]=IDEX(J, call)
```

接下来补充译码函数 decode\_op\_SI(), 取 width 宽度的字节作为偏移量, 然后如果 width 小于 4 的话, 还要进一步进行符号位扩展, 最后保存在 op->simm 中。其中符号位扩展使用 rtl\_sext(), rtl\_sext() 需要自己实现, 方法也比较简单, 左移 32-width\*8 位后再移回来即可。

call 执行函数的任务是先把下一条指令的地址(seq\_pc)压栈, 然后跳转到译码函数计算出来的地址(jmp\_pc)。跳转的 rtl\_j() 已经实现, 需要补充压栈函数 rtl\_push(), 先将 esp-4, 然后使用 rtl\_sm() 作为客户访存保存数据, 把地址压栈。

#### 2) ret 指令

ret 指令整个只有一个字节的 opcode: 0xc3,

```
opcode_table[0xc3]=EX(ret)
```

ret 用于从函数返回, 主要实现 rtl\_pop() 出栈函数, 与上述 rtl\_push() 类似, 使用 rtl\_lm 作为客户访存取数据, 然后 esp+4。

#### 3) push 指令

push 指令有 2 条他们的 opcode 分别是 0x68 和 0x55。

```
opcode_table[0x68]=IDEX(push_SI, push)
```

译码函数和 call 类似, 不过不用计算和填写 jmp\_pc, 只需要取到立即数。

rtl\_push 已经实现过了, 所以执行函数 exec\_push 直接调用即可。

```
opcode_table[0x55]=IDEX(r, push)
```

与上面类似, 区别是操作数来源不同, 不过原代码中已经实现了, 只需要填写 opcode\_table 即可。

#### 4) xor 指令

xor 指令的 opcode 是 0x31

```
opcode_table[0x31]=IDEX(G2E, xor)
```

根据手册提示, xor 会把 CF 和 OF 设置为 0, 另外需要更新 ZF 和 SF。

为了增加符号位寄存器, 需要再次修改 CPU\_state 的定义, 不过只需要关心其中的 CF, ZF, SF, IF, OF。

#### 5) sub 指令

sub 指令的 opcode\_table 已经填好, 但只是对其进行分组到了 gp1, 根据 ext\_opcode 的值, 在 gp1 中正确的位置填写真正的执行函数 EX(sub), SUB 指令涉及到 CF, OF, SF, ZF 的更新, 在 rtl.h 中填写相关的 rtl 函数完成 sub 指令。

最后在 `all-instr.h` 里面声明增加的执行函数完成 PA2.1。

## (2) PA2.2

PA2.2 与 PA2.1 类似，基本上重复 PA2.1 的工作按照 KISS 法则，一条一条实现 `tests` 里面测试用例未实现的指令，因此不再赘述。

除了实现指令，按照手册提示，在这些测试文件中，`string.c` 和 `hello-str.c` 需要实现常用的字符串库函数，以及利用变长参数实现 `sprintf` 和 `printf`，使其支持 `%c,%d,%s,%x,%p` 格式化输出。

最后，手册要求实现 `diff-test` 便于更高效的调试，`diff-test` 的方法是将自己的 `nemu` 作为 `ref`，然后与 `qemu` 的执行一条一条比对从而及时发现错误，整个 `diff-test` 的代码大部分已经实现，需要实现的只有 `x86/diff-test.c` 里面的 `isa_difftest_checkregs()`，也就是对寄存器的状态进行比对，如果有一个寄存器的值不一致就返回 `false`，否则返回 `true`。根据手册提示，这里应当注意 `nemu` 中寄存器顺序要与 `qemu` 保持一致，否则会发生未定义的问题。

## (3) PA2.3

### 1) 串口

串口的状态寄存器可以一直处于空闲状态；每当 CPU 往数据寄存器中写入数据时，串口会将数据传送到主机的标准输出。这里需要实现 `in` 和 `out` 指令，步骤与其他指令类似。

### 2) 时钟

根据手册提示，从 RTC 寄存器中获取当前时间，其地址的宏是 `RTC_ADDR`，也就是说用 `inl(RTC_ADDR)` 来获取当前时间的值，单位是 `ms`。

设置一个全局变量 `st_time`，在 `__am_timer_init()` 中调用 `inl(RTC_ADDR)` 获取启动时间，保存在 `st_time` 中，然后在 `__am_timer_read` 中再读取一次时间，求出差值，即为 `AM` 的启动时间，然后按照手册说的格式保存在 `_DEV_TIMER_UPTIME_t` 结构体变量中。

### 3) 键盘

键盘的地址是 `KBD_ADDR`，用 `inl(KBD_ADDR)` 获取键盘码，使用掩码 `KEYDOWN_MASK(0X8000)` 判断是按下还是松开，然后将键盘码保存在 `keycode` 中。

### 4) VGA

根据手册说明，宽高是 `400×300` 不变，所以在 `__am_video_read` 中，宽高是常数。

在 `__am_video_write` 中由传入参数 `buf` 指向一个 `_DEV_VIDEO_FBCTL_t` 结构体，其成员变量 `x` 和 `y` 指示起始坐标，`w` 和 `h` 指示要绘制的宽高，`pixels` 是一个行优先存储像素 `RGBA` 的数组。我们要做的是在 `FB_ADDR` 开始的一片内存中找到上述起始坐标开始的位置然后填入 `w*h` 个像素。然后在 `vga.c` 中，将控制更新屏幕的代码填上，PA2.3 就完成了。

### 2.3.3 运行结果

一键回归测试结果如图 2.5 所示。

```
compiling testcases...
testcases compile OK
[ add-longlong] PASS!
[ add] PASS!
[ bit] PASS!
[ bubble-sort] PASS!
[ div] PASS!
[ dummy] PASS!
[ fact] PASS!
[ fib] PASS!
[ goldbach] PASS!
[ hello-str] PASS!
[ if-else] PASS!
[ leap-year] PASS!
[ load-store] PASS!
[ matrix-mul] PASS!
[ max] PASS!
[ min3] PASS!
[ mov-c] PASS!
[ movsx] PASS!
[ mul-longlong] PASS!
[ pascal] PASS!
[ prime] PASS!
[ quick-sort] PASS!
[ recursion] PASS!
[ select-sort] PASS!
[ shift] PASS!
[ shuixianhua] PASS!
[ string] PASS!
[ sub-longlong] PASS!
[ sum] PASS!
[ switch] PASS!
[ to-lower-case] PASS!
[ unalign] PASS!
[ wanshu] PASS!
```

图 2.5 一键回归测试

Microbench 跑分 367 分，如图 2.6 所示。

```
=====
MicroBench PASS      367 Marks
                      vs. 100000 Marks (i7-7700K @ 4.20GHz)
Total time: 57163 ms
nemu: HIT GOOD TRAP at pc = 0x00103a38
```

图 2.6 Microbench 测试

slider 测试如图 2.7 所示。



图 2.7 slider 测试

typing 测试如图 2.8 所示。

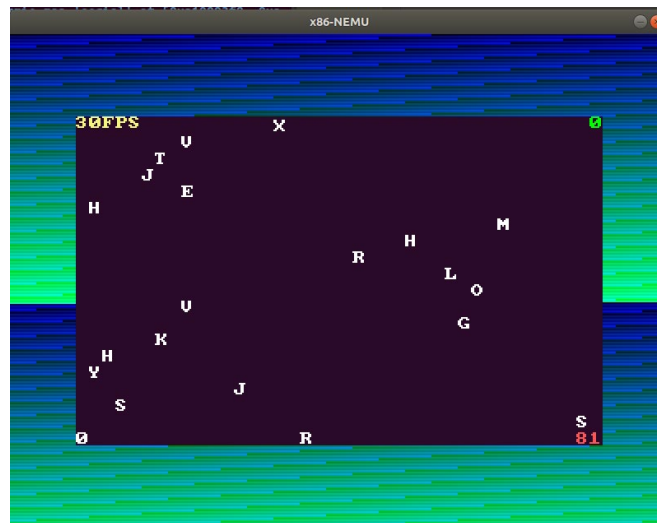


图 2.8 typing 测试

litenes 测试如图 2.9 所示。



图 2.9 litenes 测试

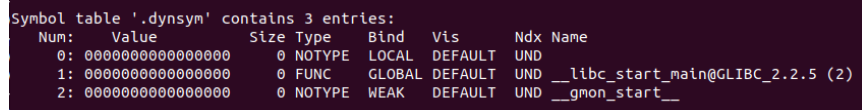
## 2.3.4 问题解答

### 必答题

- RTFSC 整理一条指令在 NEMU 中的执行过程。
  - 一条指令在 nemu 中执行需要经过 4 个阶段：取指，译码，执行和更新 PC。
  - 取指阶段通过 `instr_fetch()` 从 `pc` 指向的内存读取指令的 `opcode` 字节。
  - 译码阶段根据 `opcode_table` 索引得到的译码函数指针，调用其指向的译码函数取得操作数。
  - 执行阶段根据 `opcode_table` 索引得到的执行函数指针，调用其指向的执行函数完成具体的执行操作。
  - 更新 PC 阶段，在前面阶段中 `pc` 由于 `instr_fetch()` 已经自增，只需

要根据指令是否跳转更新适当的 `pc` 即可。

- 编译与链接
  - 单独去掉 `static` 和 `inline` 中的一个可以正常编译。
  - 同时去掉 `static` 和 `inline` 会报错 `multiple definition`。
  - `static` 修饰的函数不能被外部文件调用，因此其他文件可以定义同名函数
  - `inline` 修饰的函数不会出现在符号表中，在调用位置直接展开替换，因此可以在其他文件定义同名函数。
  - 通过尝试写一个 `static inline` 修饰的函数然后编译，发现其未出现在符号表中，从而验证了 `inline` 修饰的函数不会出现在符号表中。如图 2.10 所示。



```
Symbol table '.dynsym' contains 3 entries:
Num:  Value          Size Type  Bind  Vis      Ndx Name
 0:  0000000000000000      0 NOTYPE LOCAL DEFAULT UND
 1:  0000000000000000      0 FUNC  GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.2.5 (2)
 2:  0000000000000000      0 NOTYPE WEAK  DEFAULT UND __gmon_start__
```

图 2.10 符号表

- 编译与链接
  - 使用 `grep -c -r "dummy"` 指令查看，共有 83 个 `dummy` 实体。
  - 此时有 84 个，如果有多个弱符号，则选择其中之一，多出来的一个是在 `debug.h` 中加入的。
  - 发生报错 `redefinition of "dummy"`，如果同时进行初始化将使得它们都成为强符号，因此会报该错误。
- 了解 Makefile
  - 在敲下 `make` 后，会寻找当前目录下的 `Makefile` 或者 `makefile` 文件，根据其中定义的依赖关系，检查是否有文件更新，如果有的话，根据依赖关系将 `.c` 文件编译生成 `.o` 文件，再链接 `.o` 文件生成可执行文件。
  - 在 `nemu` 的 `Makefile` 文件首先对 `ISA` 检测，确保 `ISA` 有效，接下来定义了 `include`, `build`, `obj` 和 `binary` 文件的绝对路径以及编译选项。然后是寻找源文件是 `src` 目录下的 `.c` 文件，忽略 `isa` 目录。然后再继续寻找 `isa` 目录下所选择的 `isa` 所包含的 `.c` 文件。接下来是循环编译过程。最后是 `make run`, `make clean` 等常见命令。
  - 使用 `-n` 选项 `make` 可以看到是用 `gcc` 编译链接。

## 2.4 PA3

### 2.4.1 总体设计

PA3 整体是实现一个简单的操作系统，PA3.1 实现中断和异常，PA3.2 实现系统调用，PA3.3 实现简易的文件系统。

### 2.4.2 详细设计

#### (1) PA3.1

PA3.1 要求实现 x86 的中断指令 `int`。

IDT（中断描述符表）是一个数组，每个元素是一个门描述符，门描述符是



一个 8 字节结构体，简化以后，就只有存在位和异常入口地址这两个重要信息。在 `_cte_init` 中创建了一个 `idt` 数组。

设置好 IDT 以后，要将 IDT 的地址和长度保存在特殊的寄存器 IDTR 中，也就是 `set_idt(idt, sizeof(idt))` 的任务，在这里需要实现 `lidt` 指令完成这一任务，其源操作数指出了长度和地址的信息所在内存地址，一共是 3 个双字，第一个双字是长度，后面 2 个双字是地址，需要将这些信息存储在 `idtr` 寄存器中。同时，再一次修改 `CPU_state` 的定义，增加 IDTR 寄存器。

为了实现 `int` 操作，接下来需要实现 `raise_intr()` 函数，其工作如下：

1. 依次将 `eflags`, `cs`(代码段寄存器), `eip`(也就是 PC)寄存器的值压栈
2. 从 IDTR 中读出 IDT 的首地址
3. 根据异常号在 IDT 中进行索引，找到一个门描述符
4. 将门描述符中的 `offset` 域组合成异常入口地址
5. 跳转到异常入口地址

`raise_intr()` 函数将 `eflags`, `cs`, `ret_addr` 依次入栈，跳转到 IDT，在 `trap.S` 中，每个中断先把自己的中断号压栈，然后执行 `jmp __am_asm_trap`，在 `__am_asm_trap` 中先执行 `pushal` 指令也就是 `pusha`，把通用寄存器按照一定顺序压栈，然后 `pushl $0`，根据手册，这是一个占位符，最后 `pushl %esp` 后 `call __am_irq_handle`。跟踪上述流程，分析得出压栈顺序，由此推理出上下文 `_Context` 结构体中的成员及其定义顺序。

在 `__am_irq_handle` 加入 `switch case` 根据 `irq` 为 `event` 赋值，然后再 `do_event` 中根据刚才赋值的 `event` 处理不同的事件，完成事件分发。

最后恢复上下文，只要按照压栈时候相反的顺序出栈即可。最后跳转到栈中保存的目标地址。

## (2) PA3.2

### 1) 实现 loader

PA3.2 使用 `ramdisk_read` 和 `ramdisk_write` 实现 loader。

loader 的作用是把程序加载到指定内存位置，根据手册说明，一开始 `dummy` 的 ELF 头部文件在偏移为 0 的位置，从 `elf` 头部信息中提取出 `e_phoff` 和 `e_phnum`，也就是程序头部偏移位置和个数，接下来从这个位置开始连续读 `e_phnum` 次。

从程序头部信息中提取 5 个重要的成员，第 1 个是 `p_type`，程序类型，只有当值为 `PT_LOAD` 的时候，才加载这个头部对应的程序。第 2 个是 `p_offset`，指示这个头部对应的程序的偏移地址。第 3 个是 `p_vaddr`，指示应该将程序拷贝到的目的地址，剩下 2 个成员是 `p_filesz` 和 `p_memsz`，分别是文件大小和内存大小。

根据手册说明，需要把 `p_offset` 开始的 `p_filesz` 个字节的程序拷贝到从 `p_vaddr` 开始的内存中，另外还需要将 `p_vaddr+p_filesz` 到 `p_vaddr+p_memsz` 结束的内存设置为 0。

### 2) 系统调用

添加一个系统调用，只需要在分发的过程中添加相应的系统调用号，并编写相应的系统调用处理函数，然后调用它即可。为了屏蔽 ISA 差异，使用宏获得正确的系统调用参数寄存器和系统调用返回值。

### 3) 标准输出

添加 `sys_write` 系统调用，循环调用 `count` 次 `putc` 输出 `buf`。

#### 4) 堆区管理

因为暂时不用实现具体的内存分配,所以按照手册说的只需要系统调用的时候返回 0 表示成功即可。

接口方面,按照手册说明的流程做:

1. program break 一开始的位置位于\_end
2. 被调用时,根据记录的 program break 位置和参数 increment,计算出新 program break
3. 通过 SYS\_brk 系统调用来让操作系统设置新 program break
4. 若 SYS\_brk 系统调用成功,该系统调用会返回 0,此时更新之前记录的 program break 的位置,并将旧 program break 的位置作为\_sbrk()的返回值返回
5. 若该系统调用失败,\_sbrk()会返回-1

### (3) PA3.3

PA3.3 要求实现简易的文件系统。

#### 1) loader 使用文件

首先要完成文件相关的操作,包括文件的打开关闭,读写以及文件打开偏移地址的处理。

fs\_close()只要 return 0 即可。

fs\_open()要用传入的文件名与 file\_table 中的文件名一一比对,返回找到的下标,并设置 open\_offset。

fs\_read()要注意如果文件剩余大小不足 len,要更改读的大小,对于 stdin, stdout 和 stderr 这三个特殊文件的操作可以直接忽略。其他的使用 ramdisk\_read 进行读操作,并更新 open\_offset。

用 fs\_read 类似的方法实现 fs\_write

fs\_lseek 要实现 3 种情况:

- SEEK\_SET: 设置 open\_offset 为 offset
- SEEK\_CUR: open\_offset 增加 offset
- SEEK\_END: 设置 open\_offset 为 offset+当前文件的大小

实现了上述完整的文件操作,并且增加相关的系统调用,就可以用这些文件操作重写 loader 函数,让 loader 函数使用文件。

#### 2) 把设备抽象成文件

这一部分类似于在 OS 课设里面的做过的,将设备也看作文件,实现设备文件的读写操作。包括串口,键盘,时钟, VGA。要在 fs.c 中为它们设置好对应的读写函数指针,在 device.c 中实现对应的读写函数。

输入设备有键盘和时钟,它们对系统来说本质上就是到来了一个事件。一种简单的方式是把事件以文本的形式表现出来,定义一个事件以换行符`\n`结束:

- t 1234: 返回系统启动后的时间,单位为毫秒;
- kd RETURN / ku A: 按下/松开按键,按键名称全部大写,使用 AM 中定义的按键名

要注意的是,由于时钟事件可以任意时刻进行读取,需要优先处理按键事件,当不存在按键事件的时候,才返回时钟事件,否则用户程序将永远无法读到按键事件。

输出设备有串口和 VGA,串口只需要调用 \_putc 循环输出 buf 即可。VGA 的

实现过程在手册中有非常详细的实现步骤，因此在这里不再赘述。

### 3) 加载仙剑奇侠传

下载仙剑奇侠传资源，并放在 `navy-apps/fsimg/share/games/pal/` 目录下，更新 `ramdisk` 之后，在 `Nanos-lite` 中加载并运行 `/bin/pal`。

### 4) 基础设施 3

自由开关 `DiffTest` 需要实现 2 条命令 `detach` 和 `attach`。

`detach` 命令用于退出 `DiffTest` 模式，只需要让 `difftest_step()`, `difftest_skip_dut()` 和 `difftest_skip_ref()` 直接返回即可。

`attach` 命令用于进入 `DiffTest` 模式，为此需要将 DUT 中物理内存的内容分别设置到 REF 相应的内存区间中，并将 DUT 寄存器状态同步到 REF 中，因为 ISA 是 x86 还需要将 DUT 中 `[0,0x7c00)` 和 `[0x100000,PMEM_SIZE)` 的内存内容分别设置到 REF 相应内存区间，另外需要修改 `qemu-diff` 的代码让 `DIFFTEST_REG_SIZE` 覆盖到 `EFLAGS`，同时在 `difftest_step()` 中跳过 `EFLAGS` 相关的检查，以此解决 `EFLGAS` 一致性的问题。最后为了解决 `IDTR` 问题，需要让 `QEMU` 执行一条 `lidt` 指令将 `IDT` 描述符内容装载到 `IDTR` 中。为此需要准备一个长度为 6 字节的 `IDT` 描述符，内容是当前 `NEMU` 中 `IDTR` 内容，将这个描述符放置到 `0x7e00`，然后再准备一条 `lidt(0x7e00)` 指令，把这条指令放置 REF 的内存位置 `0x7e40`，然后把 REF 的 `eip` 设置成这个内存位置，并让 REF 执行一条指令。

快照的实现是把 `CPU_State` 和堆栈状态保存到本地文件和从本地文件读取。

### 5) 展示批处理系统

利用 `naïve_uload()` 实现 `SYS_execve` 系统调用，修改 `SYS_exit` 的实现，让它调用 `SYS_execve` 再次运行 `/bin/init`。

## 2.4.3 运行结果

主界面如图 2.11 所示，进入仙剑奇侠传开始新的故事如图 2.12 所示，读档界面如图 2.13 所示，读档成功如图 2.14 所示。



图 2.11 主菜单



图 2.12 新的游戏



图 2.13 读档

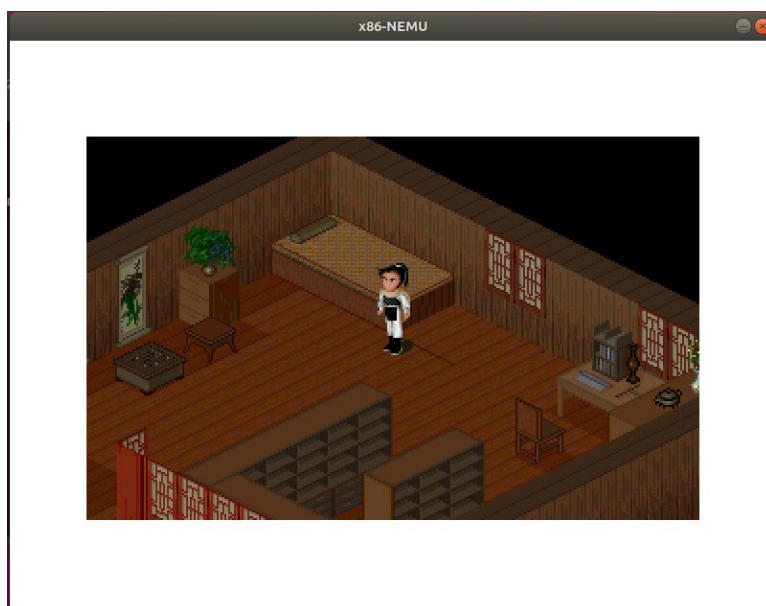


图 2.14 读档成功

## 2.4.4 问题解答

### 必答题

- 理解上下文结构体的前世今生

这个问题在 2.4.2 中的 PA3.1 的详细设计里面有阐述,追踪整个压栈的过程,在最后 `pushl %esp`, 其实就是上下文结构体指针,而压栈的寄存器为其结构体成员赋值。

- 理解穿越时空的旅程

从系统调用 `_yield()` 开始,触发自陷操作,依次将 `eflags,cs,eip` 压栈,从 `IDTR` 寄存器中读出 `IDT` 的首地址,在 `IDT` 中索引找到门描述符,将其中的 `offset` 域组合成异常入口地址,跳转到异常入口地址,将中断号压栈,跳转到 `__am_asm_trap` 在这里保存上下文,然后根据中断号分发事件,再分别处理事件,最后恢复上下文返回断点。

- `hello` 程序是什么,它从而何来,要到哪里去

`hello` 从 `c` 文件被编译链接成 `ELF` 文件,一开始在 `ramdisk` 的偏移位置为 0 的地方(在实现文件系统后,由文件的 `disk_offset` 确定),用 `loader` 加载文件,就是将 `ELF` 的需要执行的程序 `segment` 放到其正确的内存位置。由系统执行其第一条程序语句,最后调用 `SYS_write` 系统调用输出。

- 仙剑奇侠传究竟如何运行

在 `PAL_SplashScreen()` 中调用 `VIDEO_UpdateScreen` 更新屏幕,进一步调用 `SDL_UpdateRect()`,而其中的 `redraw()` 调用的 `NDL_Render()` 渲染用到的函数是 `nanos-lite` 提供的 `fwrite()`,通过将 `VGA` 抽象成文件,最后 `NEMU` 用 `AM` 提供的 `I/O` 接口把文件内容显示到屏幕上。

## 3 设计总结与心得

### 3.1 课设总结

PA 完成了一个简化的 x86 模拟器，能够运行仙剑奇侠传等程序。

- PA0 按照手册提示配置环境，安装工具。
- PA1 实现了一个简易的调试器，能够实现表达式的求值，程序的单步多步执行，监视点的创建和删除。
- PA2 学习了 x86 的指令格式，完成了一条指令的从取码，译码到执行过程，同时编写字符串库函数和 `sprintf` 函数，支持输入和输出功能。最后实现 `diff-test` 更方便的进行调试。
- PA3 完成自陷指令，系统调用，实现一个系统调用流程，还有加载程序，实现完整的文件系统，以及批处理系统。

### 3.2 课设心得

本次课设我在做的过程中，也同时写了博客记录自己调试完成的过程，博客的地址是 [yuirito.github.io](http://yuirito.github.io)，在边做边写的同时让我发现记录自己调试是一件多么重要的事情，因为很多问题当时处理的时候花费了大量时间，但是由于没有记录就会被自己遗忘，然后等同学出现同样的问题来问我的时候，我就只是有印象而已，而通过博客，我可以翻阅自己记录解释其中的细节，同时在做的时候自己的心得体会也会记录在博客中，我觉得是一件非常有意义的事情。

例如在 PA2 中，x86 的指令真的很复杂，想要对其有透彻的理解，就要实际找一条指令跟随其执行流程，分析每一步的意义，在博客中我详细记录 `mov` 指令和 `movw` 指令从取码开始到最后执行的整体流程。在 PA2 中，最难以理解的莫过于其代码中用了大量的宏，而且其分布于多个文件，找到这些宏分析其意义以及展开后的结果是 PA2 一开始就要面临的挑战，我一边记录每一步用到的宏，一边在 `vscode` 中找到宏定义的位置，然后展开，分析其意义再追踪下一个宏，用博客记录以便观察分析它们之间的联系。最后才梳理出来整个指令的执行流程，对其有了一定的把握，然后 PA2 就算是攻克了下来。

还有对于问题的记录，例如在做到 PA3 将设备抽象成文件的时候，其中调用了 `sprintf` 将 `time` 写入到 `buf` 中，但是我一开始没有意识到这里的 `sprintf` 其实是 PA2 中我自己实现的 `sprintf`，我忽视了 PA3 与 PA2 之间的关联，因为我在 PA2 实现 `sprintf` 的时候并没有支持 `%u` 格式的输出，但是我在 PA3 中使用了 `%u` 进行输出，结果一直输出为空，当时调试了很久找 bug，最后经过一番思考与调试才发现了这

个问题，这也告诉了我 PA 的各个阶段看似彼此独立，实则之间是有联系的。

最后，我有一件比较后悔的事情是我一开始头铁想着用已有的 vmware 虚拟机从零开始搭建环境，而没有使用老师给的镜像 vdi，这导致我后面出了很多奇怪的问题，比如由于我偷懒跳过了 PA0 结果导致 diff-test 的时候缺少了 qemu 然后我当时调试很久才发现是根本没有装，于是回到 PA0 把所有的工具装好再进行，还有就是吃豆人不知道为什么界面也无法显示，总之我觉得由于环境导致的问题其实是一件很不应该发生的事情，为此浪费了不少时间。

在做 PA 的时候还有一点心得就是，如果出了什么问题，第一时间应该回去重头再仔细读一遍手册，看看自己是否漏了什么，比如在做系统调用的时候，手册里面有一句不要忘记在 `_write()` 中调用系统接口函数，被我给漏看了，结果就出了问题，自己先调试了半天，结果最后重读了一遍手册，发现漏了这一句真是又哭又笑。

## 参考文献

- [1] DAVID A. PATTERSON(美). 计算机组成与设计硬件/软件接口(原书第4版). 北京: 机械工业出版社.
- [2] David Money Harris(美). 数字设计和计算机体系结构(第二版). 机械工业出版社
- [3] 秦磊华, 吴非, 莫正坤. 计算机组成原理. 北京: 清华大学出版社, 2011 年.
- [4] 袁春风编著. 计算机组成与系统结构. 北京: 清华大学出版社, 2011 年.
- [5] 张晨曦, 王志英. 计算机系统结构. 高等教育出版社, 2008 年.