

目录

课程设计概述

课设要求

实验环境

报告内容说明

PA 1

任务概述

PA 1.1

PA 1.2

PA 1.3

必答题

PA 2

任务概述

PA 2.1&2.2

指令执行过程梳理

寄存器传输语言RTL说明

x86指令识别

变长参数的使用

PA 2.3

部分测试结果

必答题

PA 3

任务概述

PA 3.1

PA 3.2

PA 3.3

仙剑奇侠传测试结果

必答题

课程设计概述

课设要求

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”,让学生探究“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。

实验环境

- OS: 20.04.2-Ubuntu
- GCC: 9.3.0
- CPU: 单核 Intel Core i5@2.4 GHz
- Memory: 4GB

注: 本次课设直接使用了老师提供的虚拟机镜像vdi

报告内容说明

本课程设计报告主要内容为:

- 本人认为有关课设的重要知识点
- 比较复杂需要梳理的实验步骤
- 课设过程中本人犯的一些错误和需要注意的小陷阱
- 部分PA测试通过截图

注: 本次系统能力综合训练仅完成了x86架构的PA 1~PA 3, 且报告内容并不包含所有实验步骤。

PA 1

任务概述

PA1 的任务分为 3 个部分，依次实现NEMU调试器的一些指令。具体指令说明如下：

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
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

PA 1.1

从键盘上读入命令后，NEMU需要解析该命令，然后执行相关的操作。解析命令的目的是识别命令中的参数，例如在 si 10 的命令中识别出 si 和 10，从而得知这是一条单步执行10条指令的命令。解析命令的工作是通过一系列的字符串处理函数来完成的，这里需要提及的函数为 strtok()。

strtok() 函数

1. 原型

```
char *strtok(char s[], const char *delim);
```

2. 说明

当 strtok() 在参数s的字符串中发现参数delim中包含的分割字符时，则会将该字符改为\0 字符。

3. 注意事项

使用该函数进行字符串分割时，会破坏被分解字符串的完整，调用前和调用后的s已经不一样了。

第一次分割之后，原字符串str是分割完成之后的第一个字符串，剩余的字符串存储在一个静态变量中，因此多线程同时访问该静态变量时，则会出现错误。

PA 1.2

PA 1.2需要实现指令 `p`，以进行表达式求值。实现算术表达式的步骤如下：

- 完成词法分析：
 - 为算术表达式中的各种token类型添加规则。
 - 在成功识别出token后，将token的信息(类型，优先级，数字的字符串等)依次记录到tokens数组中。
- 完成递归求值：
 - 实现函数 `check_parentheses()`，用于判断表达式是否被一对匹配的括号包围着，同时检查表达式的左右括号是否匹配。
 - 实现函数 `get_main_op()`，用于选择出表达式中的主运算符。
 - 对于 `负号-` 和 `指针引用符号*` 通过判断前一个token的优先级，来和 `减号-` 和 `乘号*` 区分。
 - 实现函数 `eval()` 的逻辑，完成递归求值。

PA 1.3

PA 1.3需要实现监视点的创建和删除。主要涉及链表的操作，过程较为简单，略。

必答题

- 送分题
 - 我选择的ISA是 `x86`。
- 理解基础设施
 - 假设需要编译500次NEMU才能完成PA，且有90%的次数是用于调试。当没有实现简易调试器时，只能通过GDB对运行在NEMU上的客户程序进行调试。在每一次调试中，由于GDB不能直接观测客户程序，需要花费30秒的时间来从GDB中获取并分析一个信息，并且需要获取并分析20个信息才能排除一个bug。那么，调试需要花费 $20 \times 30 \times 450 = 270000\text{s} = 75\text{h}$ 。
 - 使用简易调试器，只需要花费10秒的时间从中获取并分析相同的信息，可以节约 $2/3$ 的时间，只需花费 $75 \times 2/3 = 50\text{h}$ 。
- 查阅手册 (`x86`)
 - EFLAGS寄存器中的CF位是什么意思？
 - 答案所在章节：2.3.4 Flags Register
 - ModR/M字节是什么？
 - 答案所在章节：17.2.1 ModR/M and SIB Bytes
 - mov指令的具体格式是怎么样的？
 - 答案所在章节：3.1 Data Movement Instructions
- shell命令

- `nemu/` 下共有 5463 行代码。用的命令是 `find . -name "*.[ch]" | xargs cat | wc -l`
 - 使用 Git 回到 pa0，一共 4968 行，因此写了 495 行。
 - 除去空行有 4898 行，用的命令是 `find . -name "*.[ch]" | xargs cat | grep -v '^$' | wc -l`
- 使用 man
 - `-Wall` 的作用是打开所有的警告，`-Werror` 的作用是让所有的 warning 都以 error 显示。
 - 使用它们的目的是在编译时尽可能地把潜在的 fault 直接显示为 failure，提早发现错误，减少调试时间。

PA 2

任务概述

PA 2 的任务分为3个部分。PA 2.1和2.3实现众多x86指令，以支持Abstract machine(AM)。PA 2.3 实现 AM的输入输出扩展。

PA 2.1&2.2

指令执行过程梳理

一条指令在 NEMU 中执行过程分为 取码，译码，执行，更新 PC 四个阶段。以 mov 指令为例梳理整个指令执行流程。

从 `cpu.c` 开始，首先执行 `exec_once()` 函数：

```
1 vaddr_t exec_once(void) {
2     decinfo.seq_pc = cpu.pc;
3     isa_exec(&decinfo.seq_pc);
4     update_pc();
5
6     return decinfo.seq_pc;
7 }
```

`cpu.pc` 的值保存在 `decinfo.seq_pc` 中，然后将该值的索引传进 `isa_exec()` 中，最后调用 `update_pc()` 更新pc值。指令的译码和执行在 `isa_exec()` 中，其定义如下：

```
1 void isa_exec(vaddr_t *pc) {
2     uint32_t opcode = instr_fetch(pc, 1);
3     decinfo.opcode = opcode;
4     set_width(opcode_table[opcode].width);
5     idex(pc, &opcode_table[opcode]);
6 }
```

`isa_exec()` 从pc所指地方读取一个字节作为opcode并且保存在 `decinfo.opcode` 中。以下面这条 `mov` 指令为例：

```
1 100000: b8 34 12 00 00      mov    $0x1234,%eax
```

其opcode是0xb8，接着通过opcode作为索引，查表 `opcode_table`。

```
1 /* 0xb8 */ IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(mov_I2r, mov),
IDEK(mov_I2r, mov),
```

可知该 `mov` 指令译码函数为 `IDEK(mov_I2r, mov)`。在 `exec.h` 中，我们可看到 `IDEK` 宏的定义：

```

1 #define IDEWX(id, ex, w) {concat(decode_, id), concat(exec_, ex), w}
2 #define IDEX(id, ex) IDEWX(id, ex, 0)
3 #define EXW(ex, w) {NULL, concat(exec_, ex), w}
4 #define EX(ex) EXW(ex, 0)
5 #define EMPTY EX(inv)

```

IDEWX是宽度为0的 IDEWX，而 IDEWX 翻译为的 concat 同样是个宏(定义在 macro.h 中)，其作用为将后面2个参数拼接起来。回到 IDEX(mov_I2r, mov) 这个具体实例，通过层层宏定义， IDEX(mov_I2r, mov) 最终应该被翻译成：

```
1 {decode_mov_I2r, exec_mov, 0}
```

附： *I386 manual* 中的一些缩写

- G: 任一通用寄存器
- E: 任一通用寄存器/内存
- r: 指定的寄存器
- b: 字节
- v: word\double word\quadword(16/32/64位,取决于CPU模式)
- S: 有符号
- I: 立即数

因此，I2r释意为：该指令的操作为 立即数->指定的寄存器

再回到 `isa_exec()`，现在执行 `set_width(opcode_table[opcode].width);`。不过根据上面分析，这里宽度的值为0，也就是 `set_width(0)`。根据 `exec.c` 中的 `set_width()` 函数定义：

```

1 static inline void set_width(int width) {
2     if (width == 0) {
3         width = decinfo.isa.is_operand_size_16 ? 2 : 4;
4     }
5     decinfo.src.width = decinfo.dest.width = decinfo.src2.width = width;
6 }

```

因为 `width==0`，而 `is_operand_size_16` 默认值是 `false`，所以操作数的位宽都被设置为了4。但是一些变长指令，比如 `movw`，其执行函数将 `is_operand_size_16` 设置为 `true`，因此操作数的宽度改为16位。

继续执行下一行 `idx(pc, &opcode_table[opcode]);`。`idx()` 函数定义在 `exec.h` 中：

```
1 /* Instruction Decode and Execute */
2 static inline void idex(vaddr_t *pc, OpcodeEntry *e) {
3     if (e->decode)
4         e->decode(pc);
5     e->execute(pc);
6 }
```

作为例子的 `mov` 指令的译码函数是 `decode_mov_I2r()`，不过这个函数也是宏定义的。

`decode.h` 中可以看到 `make_DHelper(name)` 宏定义：

```
1 #define make_DHelper(name) void concat(decode_, name) (vaddr_t *pc)
```

在 `decode.c` 中，`make_DHelper(mov_I2r)` 就是译码函数 `decode_mov_I2r()` 的宏定义：

```
1 make_DHelper(mov_I2r) {
2     decode_op_r(pc, id_dest, false);
3     decode_op_I(pc, id_src, true);
4 }
```

这个函数里面调用了2个 `decode_op_xxx` 的函数，在 `decode.c` 中可看到这样一行宏定义：

```
1 #define make_DopHelper(name) void concat(decode_op_, name) (vaddr_t *pc,
2 Operand *op, bool load_val)
```

类似的，在 `decode.c` 找到 `make_DopHelper(r)` 和 `make_DopHelper(I)`：

```
1 static inline make_DopHelper(r) {
2     op->type = OP_TYPE_REG;
3     op->reg = decinfo.opcode & 0x7;
4     if (load_val) {
5         rtl_lr(&op->val, op->reg, op->width);
6     }
7
8     print_Dop(op->str, OP_STR_SIZE, "%s", reg_name(op->reg, op->width));
9 }
10
11 /* Refer to Appendix A in i386 manual for the explanations of these
12 abbreviations */
13
14 /* Ib, Iv */
15 static inline make_DopHelper(I) {
16     /* pc here is pointing to the immediate */
17     op->type = OP_TYPE_IMM;
18     op->imm = instr_fetch(pc, op->width);
19     rtl_li(&op->val, op->imm);
```

```
20     print_Dop(op->str, OP_STR_SIZE, "$0x%x", op->imm);
21 }
```

先将 `make_DopHelper(r)` 翻译如下：

```
1 static inline decode_op_r (vaddr_t *pc, Operand *op, bool load_val) {
2     op->type = OP_TYPE_REG;
3     op->reg = decinfo.opcode & 0x7;
4     if (load_val) {
5         rtl_lr(&op->val, op->reg, op->width);
6     }
7
8     print_Dop(op->str, OP_STR_SIZE, "%%s", reg_name(op->reg, op->width));
9 }
```

此处为 `mov` 指令经过层层宏定义解释的最终操作。该函数将 `op` 的类型赋值为 `OP_TYPE_REG`, 即标注为寄存器, 寄存器的编号是 `opcode & 0x7`, 例子中 `opcode` 为 `0xb8`, 所以寄存器编号是 `0`, 而 `0` 号寄存器就是 `EAX`, `load_val` 为 `false` 所以跳过。

这里还需说明一下 `rtl_xxx` 函数, 它们使用寄存器传输语言实现跨平台操作, 定义在 `include/isa/rtl.h` 文件和 `include/rtl` 目录中, 初始框架只实现了几个, 还有很多需要我们自行补充。

弄清了 `mov` 指令的执行过程, 其他所有指令的实现并不困难。均可以按照 I386 手册中的详细介绍, 填写 `opcode_table`, 实现译码函数, 执行函数。在这里就不对其他指令的具体实现情况进行详细叙述了。

附：各种宏定义说明

宏	含义
nemu/include/macro.h	
str(x)	字符串 "x"
concat(x, y)	token xy
nemu/include/cpu/decode.h	
id_src	全局变量 decinfo 中源操作数成员的地址
id_src2	全局变量 decinfo 中2号源操作数成员的地址
id_dest	全局变量 decinfo 中目的操作数成员的地址
make_DHelper(name)	名为 decode_name 的译码辅助函数的原型说明
nemu/src/isa/\$ISA/decode.c	
make_DopHelper(name)	名为 decode_op_name 的操作数译码辅助函数的原型说明
nemu/include/cpu/exec.h	
make_EHelper(name)	名为 exec_name 的执行辅助函数的原型说明
print_asm(...)	将反汇编结果的字符串打印到缓冲区 log_asmbuf 中
suffix_char(width)	操作数宽度 width 对应的后缀字符(仅x86使用)
print_asm_template[1 2 3](instr)	打印单/双/三目操作数指令 instr 的反汇编结果

寄存器传输语言RTL说明

在NEMU中, RTL指令有两种:

一种是RTL基本指令(在 `nemu/include/rtl/rtl.h` 中定义), 它们的特点是不需要使用临时寄存器, 可以看做是CPU执行过程中最基本的操作。不同的ISA都可以使用RTL基本指令, 因此它们属于ISA无关的代码。RTL基本指令包括:

- 立即数读入 `rtl_li`
- 寄存器传输 `rtl_mv`
- 32位寄存器-寄存器类型的算术/逻辑运算, 包括 `rtl_(add|sub|and|or|xor|shl|shr|sar|i?mul_[lo|hi]|i?div_[q|r])`, 这些运算的定义用到了 `nemu/include/rtl/c_op.h` 中的C语言运算
- 被除数为64位的除法运算 `rtl_i?div64_[q|r]`

- guest内存访问 `rtl_lm` 和 `rtl_sm`
- host内存访问 `rtl_host_lm` 和 `rtl_host_sm`
- 关系运算 `rtl_setrelop`, 具体可参考 `nemu/src/cpu/relop.c`
- 跳转, 包括直接跳转 `rtl_j`, 间接跳转 `rtl_jr` 和条件跳转 `rtl_jrelop`
- 终止程序 `rtl_exit` (在 `nemu/src/monitor/cpu-exec.c` 中定义)

第二种RTL指令是RTL伪指令, 它们是通过RTL基本指令或者已经实现的RTL伪指令来实现的。RTL伪指令又分两类, 包括:

- ISA无关的RTL伪指令(在 `nemu/include/rtl/rtl.h` 中定义)
 - 32位寄存器-立即数类型的算术/逻辑运算, 包括
`rtl_(add|sub|and|or|xor|shl|shr|sar|i?mul_[lo|hi]|i?div_[q|r])_i`
 - 其它常用功能, 如按位取反 `rtl_not`, 符号扩展 `rtl_sext` 等
 - ISA相关的RTL伪指令(在 `nemu/src/isa/$ISA/include/isa/rtl.h` 中定义)
 - 通用寄存器访问 `rtl_lr` 和 `rtl_sr`
 - ISA相关性较强的功能(如x86的溢出和进/借位判断, EFLAGS标志位访问等)
-

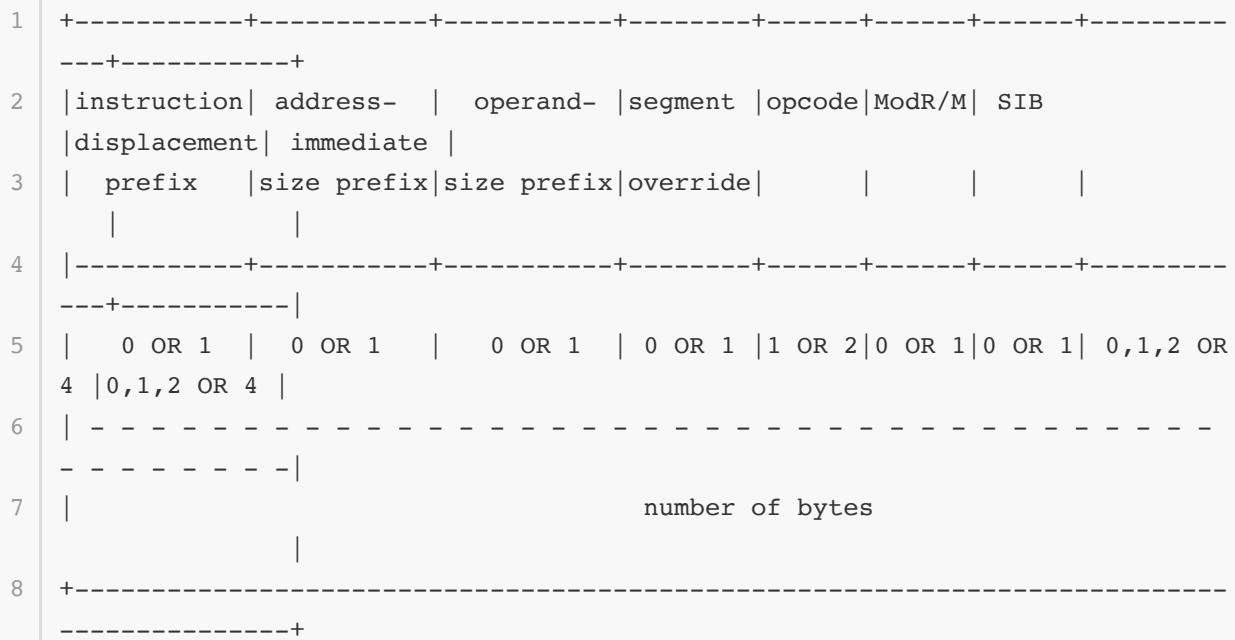
x86指令识别

一个字节最多只能区分256种不同的指令形式. 当指令形式的数目大于256时, 我们需要使用另外的方法来识别它们。x86中有主要有两种方法来解决这个问题:

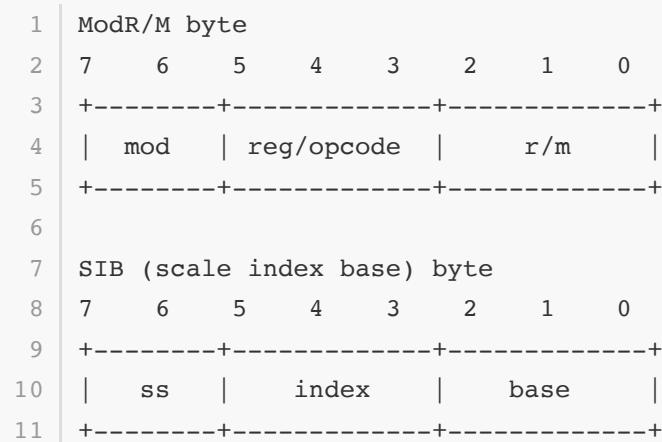
- 一种方法是使用转义码(escape code)。x86中有一个2字节转义码 `0x0f`, 当指令 `opcode` 的第一个字节是 `0x0f` 时, 表示需要再读入一个字节才能决定具体的指令形式(部分条件跳转指令就属于这种情况)。后来随着各种SSE指令集的加入, 使用2字节转义码也不足以表示所有的指令形式了, x86在2字节转义码的基础上又引入了3字节转义码, 当指令 `opcode` 的前两个字节是 `0x0f` 和 `0x38` 时, 表示需要再读入一个字节才能决定具体的指令形式。
- 另一种方法是使用 `ModR/M` 字节中的扩展 `opcode` 域来对 `opcode` 的长度进行扩充。有些时候, 读入一个字节也还不能完全确定具体的指令形式, 这时候需要读入紧跟在 `opcode` 后面的 `ModR/M` 字节, 把其中的 `reg/opcode` 域当做 `opcode` 的一部分来解释, 才能决定具体的指令形式。x86把这些指令划分成不同的指令组(instruction group), 在同一个指令组中的指令需要通过 `ModR/M` 字节中的扩展 `opcode` 域来区分。

因此, 对于 `opcode` 相同的不同指令, 使用 `ext_opcode` 进行区分。`make_group` 宏进行二次译码, 对相同 `opcode`, 不同 `ext_opcode` 的指令进行区分。

x86指令一般格式如下:



接着，给出指令中 `ModR/M` 字节和 `SIB` 字节的格式，它们是用来确定指令的操作数的。当mod域取值为3的时候，`r/m`表示的是寄存器，否则`r/m`表示的是内存。`reg`和`r/m`解释为通用寄存器编码，用来表示其中一个操作数。



根据以上指令格式，可以使用联合并指定成员的占位大小，以表示opcode, ModR_M, SIB，进而进行指令的识别：

```

1 typedef union {
2     struct {
3         uint8_t R_M      :3;
4         uint8_t reg      :3;
5         uint8_t mod      :2;
6     };
7     struct {
8         uint8_t dont_care :3;
9         uint8_t opcode    :3;
10    };
}

```

```
11     uint8_t val;
12 } ModR_M;
13
14 typedef union {
15     struct {
16         uint8_t base   :3;
17         uint8_t index  :3;
18         uint8_t ss     :2;
19     };
20     uint8_t val;
21 } SIB;
```

变长参数的使用

在实现 `sprintf()` 函数时，需要用到变长参数：

变长参数 `va_list`, `va_start`, `va_arg`, `va_end` 使用步骤：

1. 定义一个 `va_list` 类型的变量，变量是指向参数的指针。
2. `va_start` 初始化刚定义的变量，第二个参数是最后一个显式声明的参数。
3. `va_arg` 返回变长参数的值，第二个参数是该变长参数的类型。
4. `va_end` 将 a) 定义的变量重置为 `NULL`。

以上宏定义在 `stdarg.h` 中

```
1 type va_arg(
2     va_list arg_ptr,
3     type
4 );
5 void va_end(
6     va_list arg_ptr
7 );
8 void va_start(
9     va_list arg_ptr,
10    prev_param
11 );
```

PA 2.3

PA 框架提供的代码是模块化的，要在 NEMU 中加入设备的功能，只需要在 `nemu/include/common.h` 中定义宏 `HAS_IOE`。定义后，`init_device()` 函数会对设备进行初始化。重新编译后，运行 NEMU 时会弹出一个新窗口，用于显示 VGA 的输出。

PA 2.3需要实现的设备有串口, 时钟, 键盘, VGA, 设备寄存器的地址定义在 `nexus-am/am/include/nemu.h` 中。仔细阅读手册, 这些设备的实现并不困难, 但需要对内联汇编有所了解。

带有C/C++表达式的内联汇编格式为:

```
1 | __asm__ __volatile__("Instruction List" : Output : Input : Clobber/Modify);
```

由于占位符前面使用一个百分号(%), 为了区别占位符和寄存器, GCC规定在带有C/C++表达式的内联汇编中, "Instruction List"中直接写出的寄存器前必须使用两个百分号(%%)。

以PA 2.3为例:

```
1 | asm volatile ("inl %1, %0" : "=a"(data) : "d"((uint16_t)port));
```

Instruction List是`inl %1, %0`, 其中%1指代"`d`"(`uint16_t`)`port`)也就是`dx`, %0指代"`=a`"(`data`)也就是`eax`, 最后输入的来源是`port`变量, 输出的值保存在了`data`中。

部分测试结果

一键回归测试结果:

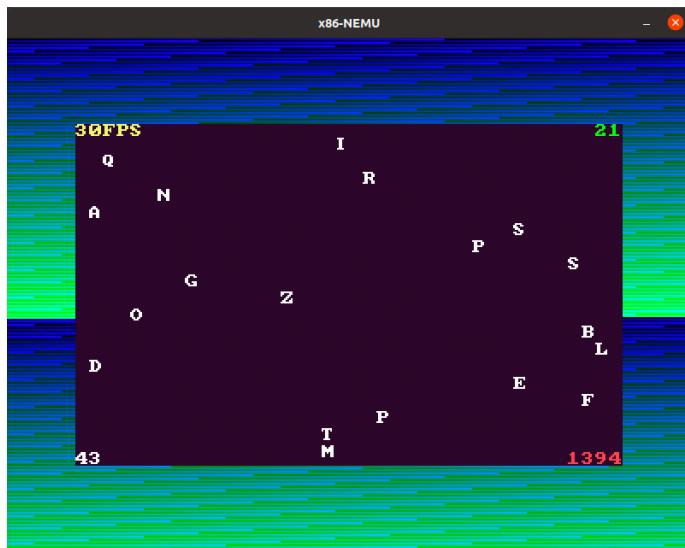
```
root@hust-desktop:/home/hust/PA/ics2019/nemu# bash runall.sh ISA=x86
compiling NEMU...
Building x86-nemu
make: Nothing to be done for 'app'.
NEMU compile OK
compiling testcases...
testcases compile OK
[ add-longlong] PASS!
[      add] PASS!
[      bit] PASS!
[ bubble-sort] PASS!
[      div] PASS!
[      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!
```

microbranch测试结果：

注：CPU为单核 Intel Core i5@2.4 GHz

```
=====
MicroBench PASS      515 Marks
      vs. 100000 Marks (i7-7700K @ 4.20GHz)
Total time: 39017 ms
nemu: HIT GOOD TRAP at pc = 0x00103a38
```

typing游戏测试结果：



必答题

- RTFSC 请整理一条指令在NEMU中的执行过程
 - 详见[分点PA 2.1&2.2](#)
- 编译与链接1
 - 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重名函数。
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重名函数。
 - 编写一个static inline修饰的函数并编译，若其未出现在符号表中，则可以验证以上结论。
- 编译与链接2
 - 使用 `grep -c -r "dummy"` 指令查看，共有 83 个 dummy 实体。
 - 此时有84个dummy实体。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"`，其原因是如果同时进行初始化将使得它们都成为强符号。
- 了解Makefile
 - 在敲下 `make` 后，`make` 程序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。

- `nemu/` 目录下的 `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。然后寻找源文件在 `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，可以看到 `make` 使用GCC编译链接。

PA 3

任务概述

PA 3 整体实现一个简单的操作系统 `Nanos-lite`：PA 3.1 实现中断和异常，PA 3.2 实现系统调用，PA 3.3 实现简易的文件系统。

PA 3.1

PA 3.1 需要实现 x86 中断指令 `int`。首先需要创建 `IDT`（中断描述符表）数组，并实现 `lidt` 指令完成 `IDT` 的设置。之后，需要实现 `raise_intr()` 函数，以进行中断响应操作。

`raise_intr()` 触发异常的硬件响应过程：

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

根据 `irq` 跳转到异常入口地址后，程序需先保存上下文。而对于 x86 保存上下文包括：

- 触发异常时的 PC 和 处理器状态。对于 x86 来说就是 `eflags`, `cs` 和 `eip`。而 x86 的异常响应机制已经将他们提前保存在堆上。
- 异常号。对于 x86，由软件保存，无需保存在堆上。
- 地址空间。这是为 PA4 准备的，PA3 中为一个占位符 `$0`。

注：保存上下文之后，还需压栈 `%esp`（上下文结构体指针）

根据系统调用的过程，可知栈内容如下：

低字节	<code>\$esp</code>
	<code>\$0</code>
	<code>\$edi</code>
	<code>\$esi</code>
	<code>\$ebp</code>
	<code>\$esp</code>
	<code>\$ebx</code>
	<code>\$edx</code>
	<code>\$ecx</code>
	<code>\$eax</code>
	<code>irq</code>
	<code>eip</code>

	cs	
高字节		eflags

根据栈内容，可知结构体 `_Context` 中变量声明如下：

```

1 struct _Context {
2     /* PA 3.1 */
3     struct _AddressSpace *as;
4     uintptr_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
5     int irq;
6     uintptr_t eip, cs, eflags;
7 };

```

因此，对各寄存器压栈即对上下文结构体成员进行赋值。

PA 3.2

实现系统调用首先需要实现 `loader()` 函数。其作用是把程序加载到指定内存位置。

为了实现 `loader()`，需要明白 ELF 文件的组织形式。在 C 语言中，`ELF` 文件头部 和 程序头部 结构分别定义为 `Elf32_Ehdr` 和 `Elf32_Phdr`。下面给出 `Elf32_Ehdr` 和 `Elf32_Phdr` 中一些重要字段的含义解释：

字段名	含义
<code>e_phoff</code>	首个程序段的偏移地址
<code>e_phnum</code>	程序段的个数
<code>e_entry</code>	虚拟入口点
<code>p_type</code>	程序类型，仅当其值为 <code>PT_LOAD</code> 时，才加载程序
<code>p_offset</code>	该段首个 bit 对于整个程序的偏移地址
<code>p_vaddr</code>	该段首个 bit 将被加载到内存中的虚拟地址
<code>p_filesz</code>	该段在整个程序中所占用的大小
<code>p_memsz</code>	该段在内存映像中占用的大小

ELF 头部文件的 offset 为 0，从 elf 头部信息中提取出 `e_phoff` 和 `e_phnum`，也就是首个程序段偏移地址和程序段个数，再从首偏移地址连续读 `e_phnum` 次，即可完成整个程序的加载。

对于每个程序段需要进行两步处理：

1. 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。
2. 将 `[p_vaddr + p_filesz, p_vaddr + p_memsz)` 区间对应的物理区间清零。

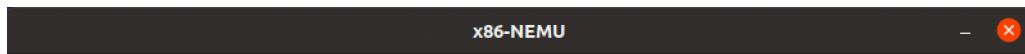
PA 3.3

PA 3.3实现一个简易的文件系统，需要实现

`fs_open()`, `fs_close()`, `fs_read()`, `fs_write()`, `fs_lseek()` 函数；还需要将设备(串口, 键盘, 时钟, VGA)抽象为文件，实现设备文件的读写操作。该过程较为简单，PA手册指导步骤也很详尽，就不在此赘述。

仙剑奇侠传测试结果

Nanos-lite 加载 `/bin/init` 程序，进入程序选择界面：



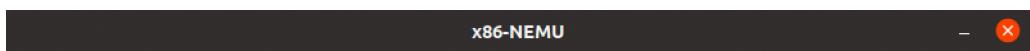
```
[0] Litenes (Super Mario Bros)
[1] Litenes (Yie Ar Kung Fu)
[2] PAL - Xian Jian Qi Xia Zhuan
[3] bmptest
[4] dummy
[5] events
[6] hello
[7] text
```

```
page =  0, #total apps = 8
help:
<- Prev Page
-> Next Page
0-9 Choose
```

运行仙剑奇侠传，并开始新的游戏：



仙剑奇侠传进行读档：



必答题

- 理解上下文结构体的前世今生 (见PA3.1阶段)
 - 详见[分点PA 3.1](#)
- 理解穿越时空的旅程 (见PA3.1阶段)

- 系统调用 `_yield()` 触发自陷操作，`raise_intr()` 函数依次将 `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。接着，将中断号 `irq` 压栈，跳转到 `__am_asm_trap()` 函数。该函数压栈各寄存器，以保存上下文。然后操作系统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 (见PA3.2阶段)
 - 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(实现文件系统后，由文件的 `disk_offset` 确定)。`loader()` 函数加载文件，将 `ELF` 文件需要执行的程序加载到指定的内存位置。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
 - 仙剑奇侠传程序在 `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()`。`Nanos-lite` 将 VGA 抽象成文件，进行写入。而 `NEMU` 用 `AM` 提供的 I/O 接口把文件内容显示到屏幕上。