

华中科技大学

2024

系统能力培养 课程实验报告

题 目:	指令模拟器
专 业:	计算机科学与技术
班 级:	CS2103 班
学 号:	U202115390
姓 名:	王乐航
电 话:	15207955882
邮 件:	2517551578@qq. com
完成日期:	2024-01-12



目 录

1	课程实验概述	1
1.1	课设目的	1
1.2	设计任务	1
1.3	设计要求	1
2	开天辟地的篇章：最简单的计算机	2
2.1	简易调试器	2
2.2	表达式求值	4
2.3	监视点	8
2.4	必答题	9
3	简单复杂的机器：冯诺依曼计算机系统	12
3.1	在 NEMU 中运行第一个 C 程序 dummy	12
3.2	实现所有的指令，完成所有的 cputest	12
3.3	输入输出	13
3.4	必答题	17
4	穿越时空的旅程：批处理系统	20
4.1	实现自陷操作	20
4.2	实现系统调用	22
4.3	实现文件系统和虚拟文件系统	23
4.4	运行仙剑奇侠传	25
4.5	批处理系统	25
4.6	必答题	26
5	实验结果与结果分析	29
5.1	实验总结	29
5.2	实验收获	29
	参考文献	31

1 课程实验概述

1.1 课设目的

计算机系统能力是每一个计算机学生都应该具备的基本能力。系统能力综合培养 课程旨在为学生设计一个综合性的系统设计任务。该课程强调对计算机系统软硬件综合系统的设计。

1.2 设计任务

理解"程序如何在计算机上运行"的根本途径是从"零"开始实现一个完整的计算机系统，实现一个经过简化但功能完备的 riscv32 模拟器 NEMU(NJU EMUlator)，最终在 NEMU 上运行游戏"仙剑奇侠传"，探究"程序在计算机上运行"的基本原理。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。

1.3 设计要求

- (1) 实现简易调试器、表达式求值、监视点。
- (2) 运行第一个 C 程序、实现指令集、测试所有程序、实现 I/O 指令、测试打字游戏。
- (3) 实现系统调用、文件系统；运行《仙剑奇侠传》。

2 开天辟地的篇章：最简单的计算机

主要实现一个简易调试器。丰富命令，添加单步执行，打印程序状态，表达式求值，内存扫描，设置监视点，删除监视点等功能。基于后续实验可以基于实现的简易调试器来进行调试。

2.1 简易调试器

1. 单步执行

单步执行的步骤为解析指令中的参数 n ，调用 `cpu_exec` 函数单步执行 n 步。

```
static int cmd_si(char *args){
    char *arg = strtok(NULL, " ");

    if (arg == NULL){
        cpu_exec(1);
    }else{
        int n = 0;
        for (int i=0; i<strlen(args); i++){
            n = args[i] + n * 10;
        }
        cpu_exec(n);
    }
    return 0;
}
```

图 2.1 单步执行实现

```
(nemu) help
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - 'si [N]'. Execute N instructions in a single step. The default value of N is 1
info - Print register state
p - 'p EXPR'. Evaluate the expression EXPR
x - 'x N EXPR'. Scanning memory
w - 'w EXPR'. Suspends program execution when the value of the expression EXPR changes
d - 'd [N]'. Delete the number watchpoint
(nemu) si
80100000: b7 02 00 80          lui 0x80000,t0
(nemu) si 2
80100004: 23 a0 02 00          sw 0(t0),$0
80100008: 03 a5 02 00          lw 0(t0),a0
(nemu) si 3
8010000c: 6b 00 00 00          nemu trap
nemu: HIT GOOD TRAP at pc = 0x8010000c
```

图 2.2 单步执行结果

2. 打印程序状态

通过解析第二个参数，选择输出寄存器状态还是监视点状态。根据需要输出的状态类型使用系统对应的函数将监视点或者寄存器输出。

```
static int cmd_info(char *args){
    char *arg = strtok(NULL, " ");

    if(arg == NULL){
        return 0;
    }else if(strcmp(arg, "r") == 0){
        isa_reg_display();
    }else if(strcmp(arg, "w") == 0){
        watchpoints_display();
    }
    return 0;
}
```

图 2.3 打印程序状态实现

```
(nemu) w $t0
$t0[src/monitor/debug/expr.c,91,make_token] match rules[11] = "\\$(\\$0|ra|[sgt]p|t[0-6]|a[0-7]|s([0-9]|l[0-1]))" at position 0 with len 3: $t0
NO.0 watchpoint is setted.
(nemu) w $t3
$t3[src/monitor/debug/expr.c,91,make_token] match rules[11] = "\\$(\\$0|ra|[sgt]p|t[0-6]|a[0-7]|s([0-9]|l[0-1]))" at position 0 with len 3: $t3
NO.1 watchpoint is setted.
(nemu) info w
NUM          VALUE          EXPR
1            0(0x0)          )    $t3
0            0(0x0)          )    $t0
(nemu) info r
$0          0x0 0
ra          0x0 0
sp          0x0 0
gp          0x0 0
tp          0x0 0
t0          0x0 0
t1          0x0 0
```

图 2.4 打印监视点和寄存器状态

3. 扫描内存

扫描内存先通过 `expr` 计算表达式表示的地址，然后显示出对于内存的地址。

```

static int cmd_x(char *args) {
    /* extract the argument */
    char *N = strtok(NULL, " ");
    char *e = strtok(NULL, "\\n");
    if (N == NULL || e == NULL) {
        // 参数错误
        printf("A syntax error in expression\\n");
    } else {
        int n = atoi(N);

        // 求出表达式的值
        bool success;
        paddr_t base_addr = expr(e, &success);
        if (!success) {
            printf("Error expression!\\n");
            return 0;
        }

        // 输出结果
        int i;
        for (i = 0; i < n; i++) {
            if (i % 4 == 0) {
                if (i != 0) printf("\\n");
                printf("%#x:\\t", base_addr);
            }
            printf("%#x\\t", paddr_read(base_addr, 4));
            base_addr += 4;
        }
        printf("\\n");
    }
    return 0;
}

```

图 2.5 扫描内存实现

```

(nemu) x 5 0x80100000
[src/monitor/debug/expr.c,87,make_token] match rules[0] = "0x[0-9a-fA-F]+"
0x80100000:    0x800002b7    0x2a023 0x2a503 0x6b
0x80100010:    0

```

图 2.6 扫描内存结果

2.2 表达式求值

1. 生成表达式

在实现表达式求值之前，为了实现后续对表达式求值解析批量测试，需要先实现生成表达式。对生成表达式有三种情况，第一种：数字，第二种：生成括号中间夹着表达式，第三种：表达式加运算符加表达式；对于这三种情况为依靠随机数来选择表达式的结果。通过简单的表达式来递归生成复杂表达式。

```

static inline void gen_num()
{
    uint32_t num = rand()%1000;
    char buffer[40];
    snprintf(buffer, sizeof(buffer), "%d", num);
    int len = strlen(buffer);
    strcpy(&buf[strSub], buffer);
    strSub+=len;
    nr_tokens++;
    return ;
}

static inline void gen(char c)
{
    if(c=='('){
        buf[strSub] = '(';
    }
    if(c==')'){
        buf[strSub] = ')';
    }
    strSub++;
    nr_tokens++;
    return ;
}

static inline void gen_rand_op(){
    switch (choose(4)){
        case 0 : buf[strSub]='+';break;
        case 1 : buf[strSub]='-';break;
        case 2 : buf[strSub]='*';break;
        case 3 : buf[strSub]='/' ;break;
    }
    strSub++;
    nr_tokens++;
    return;
}

```

图 2.7 生成表达式

2. 表达式求值

先需要补全规则，如图 2.8 所示。

```

static struct rule {
    char *regex;
    int token_type;
} rules[] = {

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

    {" +", TK_NOTYPE},    // spaces
    {"\\+", '+'},        // plus
    {"==", TK_EQ},        // equal
    {"!=", TK_NOTEQ},     // not equal
    {"\\-", '-'},        // minus
    {"\\*", '*'},         // times
    {"/", '/'},          // divide
    {"\\(", '('},
    {"\\)", ')'},
    {"\\[", '['},
    {"\\]", ']'},
    {"0[xX][0-9a-fA-F]+", TK_HEXADECEIMAL},
    {"[0-9]+", TK_DECIMAL},
    {"\\$(\\$0|[ra][sgt]p|t[0-6]|a[0-7]|s([0-9]|1[0-1]))", TK_REG}, // more specific
    // {"\\$(\\$0-9a-zA-Z)+", TK_REG}, // general
    {"\\\\\\\\", TK_OR},
    {"<=", TK_LESSEQ},
    {">=", TK_GREATEREQ},
    {"<", TK_LESS},
    {">", TK_GREATER},
    {"&&", TK_AND}
};

```

图 2.8 匹配规则补全

词法分析，将数字和寄存器类型将对应的字符串记录下来。

```

switch (rules[i].token_type) {
    case TK_NOTYPE:
        break;
    case TK_DECIMAL:
    case TK_HEXADECEIMAL:
    case TK_REG:
        tokens[nr_token].type = rules[i].token_type;
        if(substr_len>=32){
            printf("\033[0;33m Number Overflow !\n\033[0m;");
            return false;
        }
        strncpy(tokens[nr_token].str, substr_start, substr_len);
        tokens[nr_token].str[substr_len] = '\0';
        nr_token++;
        break;
    default:
        tokens[nr_token].type = rules[i].token_type;
        nr_token++;
}

```

图 2.9 记录数字和寄存器类型

寻找表达式的主运算符，将其分裂为两个子表达式处理。对于每个表达式的求解都需要检查括号的合法性，遍历表达式中的字符时，要求右括号的数量小于

左括号，在遍历结束之后，需要左右括号的数量相等。

逐渐递归处理表达式，直至其为最简单的数字，根据主运算符来将两个简单表达式合成为一个简单表达式，递归计算直至为一个数字。

```
uint32_t findMainOp(int p, int q, bool* success){
    uint32_t op = 0;
    int layer = 0;
    int precedence = 0;
    for(int i=p; i<=q; i++){
        if(layer==0){
            int type = tokens[i].type;
            if(type=='('){
                layer++;
                continue;
            }
            if(type==')'){
                printf("Bad expression at [%d %d]\n",p,q);
                *success = false;
                return 0;
            }
            int type_precedence = op_precedence(type);
            if (type_precedence >= precedence){
                op=i;
                precedence=type_precedence;
            }
        }
        else{
            if(tokens[i].type==')'){
                layer--;
            }
            if(tokens[i].type=='('){
                layer++;
            }
        }
    }
}
```

图 2.10 将表达式分裂为两个字表达式

3. 表达式测试结果

```
1364 ((( (155 ) +891 ) ) -410 ) + ((( 176/66 *364 ) ))
290 290
873 (873)/(303/275)
```

图 2.11 生成表达式

将生成的表示式计算结果，输出的结果符合预期

```

(nemu) p (((155 ) +891 ))-410 )+(((176/66 *364 )))
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "(" at position 0 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "\" at position 1 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "(" at position 2 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "\" at position 3 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[10] = "[0-9]+" at position 4 with len 3: 155
[src/monitor/debug/expr.c,91,make_token] match rules[0] = " +" at position 7 with len 1:
[src/monitor/debug/expr.c,91,make_token] match rules[8] = ")" at position 8 with len 1: )
[src/monitor/debug/expr.c,91,make_token] match rules[0] = " +" at position 9 with len 1:
[src/monitor/debug/expr.c,91,make_token] match rules[1] = "\+" at position 10 with len 1: +
[src/monitor/debug/expr.c,91,make_token] match rules[10] = "[0-9]+" at position 11 with len 3: 891
[src/monitor/debug/expr.c,91,make_token] match rules[0] = " +" at position 14 with len 2:
[src/monitor/debug/expr.c,91,make_token] match rules[8] = ")" at position 16 with len 1: )
[src/monitor/debug/expr.c,91,make_token] match rules[8] = "\" at position 17 with len 1: )
[src/monitor/debug/expr.c,91,make_token] match rules[4] = "\-" at position 18 with len 1: -
[src/monitor/debug/expr.c,91,make_token] match rules[10] = "[0-9]+" at position 19 with len 3: 410
[src/monitor/debug/expr.c,91,make_token] match rules[0] = " +" at position 22 with len 1:
[src/monitor/debug/expr.c,91,make_token] match rules[8] = ")" at position 23 with len 1: )
[src/monitor/debug/expr.c,91,make_token] match rules[1] = "\+" at position 24 with len 1: +
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "(" at position 25 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "\" at position 26 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "(" at position 27 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[10] = "[0-9]+" at position 28 with len 3: 176
[src/monitor/debug/expr.c,91,make_token] match rules[6] = "/" at position 31 with len 1: /
[src/monitor/debug/expr.c,91,make_token] match rules[10] = "[0-9]+" at position 32 with len 2: 66
[src/monitor/debug/expr.c,91,make_token] match rules[0] = " +" at position 34 with len 1:
[src/monitor/debug/expr.c,91,make_token] match rules[5] = "\*" at position 35 with len 1: *
[src/monitor/debug/expr.c,91,make_token] match rules[10] = "[0-9]+" at position 36 with len 3: 364
[src/monitor/debug/expr.c,91,make_token] match rules[0] = " +" at position 39 with len 2:
[src/monitor/debug/expr.c,91,make_token] match rules[8] = ")" at position 41 with len 1: )
[src/monitor/debug/expr.c,91,make_token] match rules[8] = "\" at position 42 with len 1: )
[src/monitor/debug/expr.c,91,make_token] match rules[8] = ")" at position 43 with len 1: )
val1:176 / val2:66
((((155 ) +891 ))-410 )+(((176/66 *364 ))) = 1364(0x554) ;

```

图 2.12 计算表达式的结果

2.3 监视点

1. 设置监视点

在监视点结构体中，添加三个字段，保存表达式的字符串数组 e，保存上一次表达式值得 old_value，记录命中次数的 hit_times，然后监视点的组织结构为链表，添加链表的创建、添加、删除等操作。

```

(nemu) w $t0
$t0[src/monitor/debug/expr.c,91,make_token] match rules[11] = "
: $t0
NO.0 watchpoint is setted.
(nemu) w $t3
$t3[src/monitor/debug/expr.c,91,make_token] match rules[11] = "
: $t3
NO.1 watchpoint is setted.
(nemu) info w

```

NUM	VALUE	EXPR
1	0(0x0)	\$t3
0	0(0x0)	\$t0

图 2.13 创建监视点

2. 取消监视点

在监视点链表中查找需要移除的节点，将其占用的空间释放。

```
(nemu) info w
NUM          VALUE          EXPR
3            0(0x0)         $t3
2            0(0x0)         $t2
1            0(0x0)         $t1
0            0(0x0)         $t0
(nemu) d 2
[src/monitor/debug/expr.c,91,make_token] match rules[10] = "[0-9]+" at position 0 with len 1: 2
(nemu) info w
NUM          VALUE          EXPR
3            0(0x0)         $t3
1            0(0x0)         $t1
0            0(0x0)         $t0
```

图 2.14 删除监视点

2.4 必答题

1. 我选择的 ISA 是 RISCV3

2.

- 假设你需要编译 500 次 NEMU 才能完成 PA.
- 假设这 500 次编译当中，有 90%的次数是用于调试.
- 假设你没有实现简易调试器，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，你需要花费 30 秒的时间来从 GDB 中获取并分析一个信息.
 - 假设你需要获取并分析 20 个信息才能排除一个 bug.

这个学期下来，你将会在调试上花费多少时间？

$$500 * 90\% * 30 * 20 = 270000s = 75 \text{ 小时}$$

由于简易调试器可以直接观测客户程序，假设通过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. 那么这个学期下来，简易调试器可以帮助你节省多少调试的时间？

$500 \times 90\% \times (30 - 10) \times 20 = 180000s = 50$ 小时

2. riscv32 有哪几种指令格式

在 RISC-V32 手册 32 页，RISC-V-32 有六种指令格式

R 类型指令：用于寄存器 - 寄存器操作；

I 类型指令：用于短立即数和访存 load 操作；

S 类型指令：用于访存 store 操作；

B 类型指令：用于条件跳转操作；

U 类型指令：用于长立即数操作；

J 类型指令：用于无条件操作；

3. LUI 指令的行为是什么

RISC-V-Reader-Chinese-v2p1.pdf 文档 152 页。

lui 高位立即数加载，将符号位扩展的 20 位立即数 immediate 左移 12 位，并将低 12 位置零，写入 x[rd] 中

4. shell 命令

./nemu 目录下所有.c 和.h 文件总共有多少代码，使用什么命令得到这个结果的，和框架代码相比，在 PA1 中编写了多少行代码。

```
hust@hust-desktop:~/Documents/ics2019$ find ./nemu -name "*.c" -o -name "*.h" | xargs cat | wc
5955  18634 149927
```

图 2.15 获取代码行数的命令和结果

总共 5955 行代码。

使用 `find ./nemu -name "*.c" -o -name "*.h" | xargs cat | wc` 命令
框架代码行数为 4968，PA1 中编写了 987 行。

6. gcc 中的-Wall 和-Werror 有什么作用？为什么要使用-Wall 和-Werror

“-Wall” 是 GCC 编译器中的一个选项，它用于开启大部分常用的警告信息。这些警告信息可以帮助开发者发现代码中可能存在的潜在问题，如未使用的变量、类型不匹配、函数参数使用不当等。

“-Werror” 选项会将所有的警告当作错误来处理。这意味着如果在编译过程中出现了任何警告信息，编译器会停止编译并返回错误码。

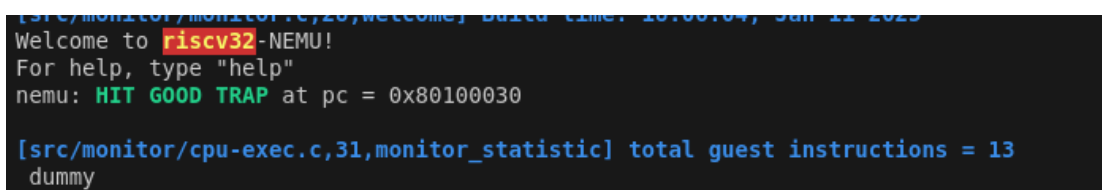
可以仿制代码的不规范，保证代码在编译是正确的，减少运行是因为警告导致的错误。

3 简单复杂的机器：冯诺依曼计算机系统

本阶段实验主要任务为：在 NEMU 中运行第一个 C 程序 `dummy`、实现更多的指令，在 NEMU 中运行所有 `cputest`，运行小游戏。

3.1 在 NEMU 中运行第一个 C 程序 `dummy`

`dummy-riscv32-nemu.txt` 文件找到这部分需要实现的指令有：`li`, `auipc`, `addi`, `jal`, `mv`, `sw`, `jalr`。通过文档，为了实现这几条未实现的指令，我们需要在 `opcode_table` 中添加正确的译码辅助函数。在实现指令后，运行 `dummy` 获得正确的结果。



```
[src/monitor/monitor.c,20,welcome] build time: 10:00:04, Jan 11 2023
Welcome to riscv32-NEMU!
For help, type "help"
nemu: HIT GOOD TRAP at pc = 0x80100030

[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 13
dummy
```

图 3.1 C 程序 `dummy` 运行结果

3.2 实现所有的指令，完成所有的 `cputest`

在上一步中，已经实现了为实现的几条指令。

还需要实现字符串处理函数，这些未实现的函数都是比较简单常见的，所以也很容易处理。

```

hust@hust-desktop:~/Desktop/ics2019/nemu$ bash runall.sh ISA=riscv32
compiling NEMU...
Building riscv32-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!

```

图 3.2 测试程序运行结果

3.3 输入输出

主要是对输入输出的理解和相关 API 的调用。

1. 运行 hello world

```

hust@hust-desktop:~/Desktop/ics2019/nexus-am/tests/amtest$ make ARCH=native mainargs=h run
# Building amtest [native] with AM_HOME {/home/hust/Desktop/ics2019/nexus-am}
# Building lib-am [native]
# Building lib-klib [native]
+ CC src/stdio.c
+ CC src/string.c
+ AR -> build/klib-native.a
# Creating binary image [native]
+ LD -> build/amtest-native
/home/hust/Desktop/ics2019/nexus-am/tests/amtest/build/amtest-native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native

```

图 3.3 hello world 运行结果

2. 时钟

有了时钟，程序才可以提供时间相关的体验，例如游戏的帧率，程序的快慢等。

实现_DEVREG_TIMER_UPTIME 中的功能。

```
case DEVREG_TIMER_UPTIME: {
    _DEV_TIMER_UPTIME_t *uptime = (_DEV_TIMER_UPTIME_t *)buf;
    uint32_t past_time = inl(RTC_ADDR);
    uptime->hi = 0;
    uptime->lo = past_time - boot_time;
    return sizeof(_DEV_TIMER_UPTIME_t);
}
```

图 3.4 _DEVREG_TIMER_UPTIME 功能的实现

```
hust@hust-desktop:~/Desktop/ics2019/nexus-am/tests/amtest$ make ARCH=native mainargs=t run
# Building amtest [native] with AM_HOME {/home/hust/Desktop/ics2019/nexus-am}
# Building lib-am [native]
# Building lib-klib [native]
# Creating binary image [native]
+ LD -> build/amtest-native
/home/hust/Desktop/ics2019/nexus-am/tests/amtest/build/amtest-native
2025-1-11 19:12:45 GMT (1 second).
2025-1-11 19:12:46 GMT (2 seconds).
2025-1-11 19:12:47 GMT (3 seconds).
2025-1-11 19:12:48 GMT (4 seconds).
2025-1-11 19:12:49 GMT (5 seconds).
2025-1-11 19:12:50 GMT (6 seconds).
```

图 3.5 时钟运行结果

3. 键盘

键盘是最基本的输入设备。一般键盘的工作方式如下：当按下一个键的时候，键盘将会发送该键的通码(make code)；当释放一个键的时候，键盘将会发送该键的断码(break code)。

实现_DEVREG_INPUT_KBD 中的功能

```
size_t __am_input_read(uintptr_t reg, void *buf, size_t size) {
    switch (reg) {
        case DEVREG_INPUT_KBD: {
            _DEV_INPUT_KBD_t *kbd = (_DEV_INPUT_KBD_t *)buf;
            uint32_t keyboard_code = inl(KBD_ADDR); // 从MMIO中获取键盘码
            kbd->keydown = keyboard_code & KEYDOWN_MASK ? 1 : 0;
            kbd->keycode = keyboard_code & ~KEYDOWN_MASK; // _KEY_NONE == 0
            return sizeof(_DEV_INPUT_KBD_t);
        }
    }
    return 0;
}
```

图 3.6 _DEVREG_INPUT_KBD 的实现


```

hust@hust-desktop:~/Desktop/ics2019/nexus-am/tests/amtest$ make ARCH=native mainargs=k run
# Building amtest [native] with AM_HOME {/home/hust/Desktop/ics2019/nexus-am}
# Building lib-am [native]
# Building lib-klib [native]
# Creating binary image [native]
+ LD -> build/amtest-native
/home/hust/Desktop/ics2019/nexus-am/tests/amtest/build/amtest-native
Try to press any key...
Get key: 43 A down
Get key: 43 A up
Get key: 30 W down
Get key: 30 W up
Get key: 44 S down
Get key: 44 S up
Get key: 48 H down
Get key: 48 H up
Get key: 38 P down
Get key: 38 P up
Get key: 46 F down
Get key: 46 F up
Get key: 54 RETURN down
Get key: 54 RETURN up

```

图 3.7 键盘输入的测试结果

4. VGA

VGA 可以用于显示颜色像素，是最常用的输出设备。实现 `_DEVREG_VIDEO_FBCTL` 中的功能。

```

size_t __am_video_write(uintptr_t reg, void *buf, size_t size) {
    switch (reg) {
        case DEVREG_VIDEO_FBCTL: {
            _DEV_VIDEO_FBCTL_t *ctl = (_DEV_VIDEO_FBCTL_t *)buf;
            int x = ctl->x, y = ctl->y, w = ctl->w, h = ctl->h;
            uint32_t *pixels = ctl->pixels;
            int cp_bytes = sizeof(uint32_t) * min(w, W - x);
            for (int j = 0; j < h && y + j < H; j++) {
                memcpy(&fb[(y + j) * W + x], pixels, cp_bytes);
                pixels += w;
            }
            if (ctl->sync) {
                // do nothing, texture_sync() is called by SDL_timer
            }
            return size;
        }
    }
    return 0;
}

```

图 3.8 `_DEVREG_VIDEO_FBCTL` 中的功能实现

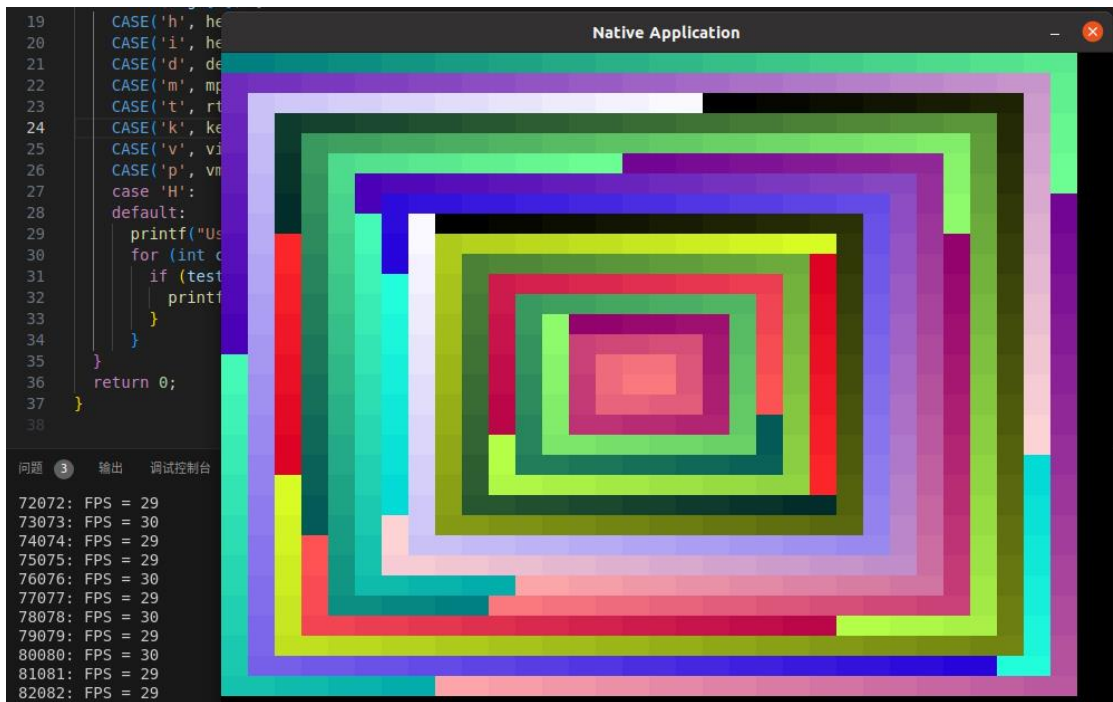


图 3.9 VGA 测试结果

5. 打字游戏

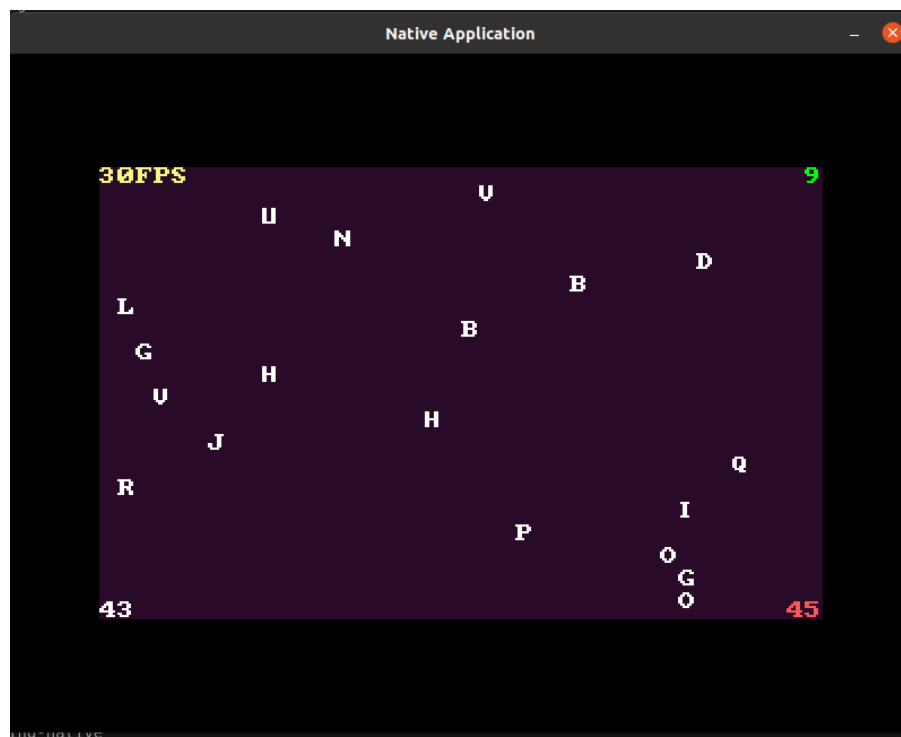


图 3.10 打字游戏运行结果

6. 幻灯片播放

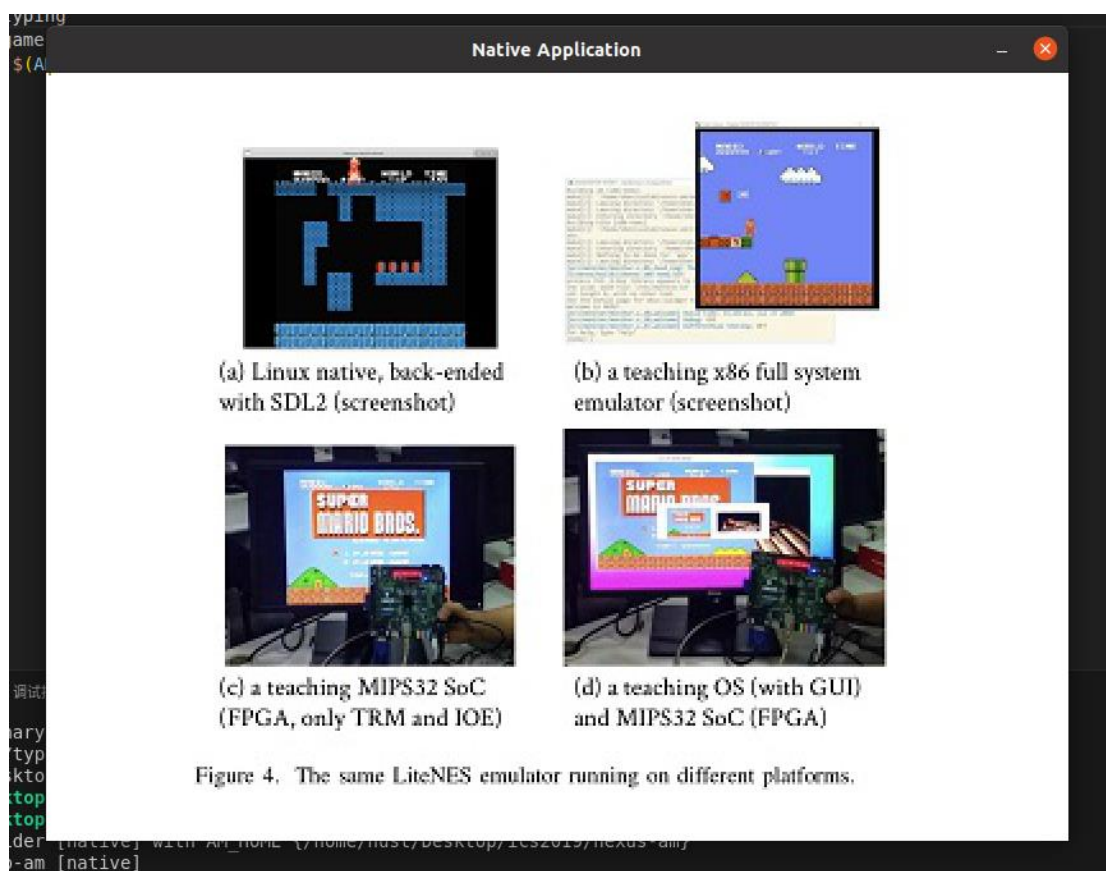


图 3.11 幻灯片播放

3.4 必答题

1. 整理一条指令在 NEMU 中的执行过程
 - (1) 取址：函数 `instr_fetch()` 进行取指令操作，本质上是一次内存的访问
 - (2) 译码：根据指令的 `opcode`，获得指令的操作和操作对象，取出 `opcode` 之后，框架代码用它来对 `opcode_table` 数组进行索引，取出其中的一个 `OpcodeEntry` 类型的元素，从译码查找表中取得的元素将会被作为参数，调用 `idex()` 函数用来进行译码和执行。
 - (3) 执行：执行过程会在 `idex()` 函数译码过后然后执行。
 - (4) 更新 PC：在执行结束之后，确定刚刚执行完的指令的长度，通过 `update_pc()` 对 PC 进行更新。

2. 去掉 `static`、`inline` 重新进行编译是否有错误。

在分别去掉两者和同时去掉时，不会发生错误。

3. 编译与链接

- (1) 在 `nemu/include/common.h` 中添加一行 `volatile static int dummy`; 然后重新编译 NEMU. 请问重新编译后的 NEMU 含有多少个 `dummy` 变量的实体? 你是如何得到这个结果的?

`dummy` 变量为 `static` 类型，在每个包含 `common.h` 源文件的源文件中都包含一个 `dummy` 变量

- (2) 在 `nemu/include/debug.h` 中添加一行 `volatile static int dummy`; 然后重新编译 NEMU. 请问此时的 NEMU 含有多少个 `dummy` 变量的实体? 与上题中 `dummy` 变量实体数目进行比较, 并解释本题的结果.

和上题类似，不过因为 `debug.h` 文件被更多文件包含，`dummy` 变量实体数量会更多

- (3) 修改添加的代码，为两处 `dummy` 变量进行初始化:`volatile static int dummy = 0`; 然后重新编译 NEMU. 你发现了什么问题? 为什么之前没有出现这样的问题? (回答完本题后可以删除添加的代码.)

将 `common.h` 和 `debug.h` 中的 `dummy` 变量初始化为 0。然后重新编译 NEMU。

结果，编译器可能会报错，提示重复定义 `dummy` 变量。这是因为 `static` 变量在每个包含它的源文件中都是独立的，但初始化会导致链接器错误。

之前没有出现这样的问题是因为 `static` 变量在每个源文件中都是独

立的实体，不会在链接阶段冲突。但一旦进行初始化，编译器会尝试在多个源文件中初始化同一个变量，从而导致链接器错误。

4. **Makefile** `make` 程序如何组织.c 和.h 文件，最终生成可执行文件 `nemu/build/$ISA-nemu`.

在 `makefile` 中，首先我们根据输入设置了指令集 `ISA`，根据该变量，我们设置头文件搜索路径，包括目录 `nemu/include` 和 `nemu/src/isa/x86/include`，在 `gcc` 的编译选项中设置上述两个头文件路径。接下来，我们使用 `find` 等命令找到所有需要编译的源文件，首先找到 `nemu/src` 下除 `isa` 目录外其 17 余所有位置的.c 文件，然后找到 `nemu/src/isa/x86` 目录下的全部.c 文件，这两部分加在一起即为待编译的源文件列表。最后，根据 `makefile` 中设置的编译器、编译选项等完成编译链接过程即可。

4 穿越时空的旅程：批处理系统

本阶段的实验任务为：实现自陷操作及其过程，实现用户程序的加载和系统调用，运行仙剑奇侠传和展示批处理系统

4.1 实现自陷操作

1. 实现新的指令

需要实现 `ecall` 环境调用指令，`sret` 管理员模式例外返回指令，以及控制状态寄存器相关操作指令。

只需要 `opcode_code` 中设置，然后实现想要的译码函数和执行辅助函数。

```
make EHelper(system){
    Instr instr = decinfo.isa.instr;
    switch(instr.funct3){
        case 0b0:
            if ((instr.val & ~(0x7f)) == 0) { // ecall 环境调用
                raise_intr(reg_l(17), cpu.pc);
            } else if (instr.val == 0x10200073) { // sret 管理员模式例外返回
                decinfo.jump_pc = decinfo.isa.sepc + 4;
                rtl_j(decinfo.jump_pc);
            } else {
                assert(!"Undo the system opcode");
            }
            break;
        case 0b001: // csrrw
            s0 = get_csr(instr.csr);
            write_csr(instr.csr, id_src->val);
            rtl_sr(id_dest->reg, &s0, 4);
            print_asm_template3(csrrw);
            break;
    }
}
```

图 4.1 系统调用执行辅助函数

根据 `funct3` 的不同进行区分不同的指令。

```
void raise_intr(uint32_t NO, vaddr_t epc) {
    /* TODO: Trigger an interrupt/exception with ``NO``.
     * That is, use ``NO`` to index the IDT.
     */
    decinfo.isa.sepc = epc; // 将当前PC值保存到sepc寄存器
    decinfo.isa.scause = NO; // 在scause寄存器中设置异常号
    decinfo.jump_pc = decinfo.isa.stvec; // 从stvec寄存器中取出异常入口地址
    rtl_j(decinfo.jump_pc); // 跳转到异常入口地址
}
```

图 4.2 异常操作

2. 实现事件分发和恢复上下文

在 `_am_irq_handle()` 函数中通过异常号识别自陷异常，将 `event` 设置为编号为 `_EVENT_YIELD` 自陷事件。

```
Context* am_irq_handle(_Context *c) {
    _Context *next = c;

    if (user_handler) {
        _Event ev = {0};
        switch (c->cause) {
            case -1: // 自陷异常
                ev.event = _EVENT_YIELD;
                break;
        }
    }
}
```

图 4.3 自陷异常设置

在后面 `do_event()` 函数中识别出自陷时间 `_EVENT_YIELD`，输出“Self trap!”。

```
static Context* do_event(_Event e, Context* c) {
    switch (e.event) {
        case _EVENT_YIELD:
            printf("Self trap!\n");
            break;
        case _EVENT_SYSCALL:
            do_syscall(c);
            break;
        default:
            panic("Unhandled event ID = %d", e.event);
    }
    return NULL;
}
```

图 4.4 输出自陷事件

3. 自陷事件测试

重新运行 `nanos-lite`，自陷测试结果符合预期。

```
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) c
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,15,main] Build time: 21:49:38, Jan 10 2022
[/home/hust/Desktop/ics2019/nanos-lite/src/randisk.c,28,init_randisk] randisk info: start = , end = , size = -2146427889 bytes
[/home/hust/Desktop/ics2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/hust/Desktop/ics2019/nanos-lite/src/irq.c,20,init_irq] Initializing interrupt/exception handler...
[/home/hust/Desktop/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,33,main] Finish initialization
Self trap!
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,39,main] system panic: Should not reach here
nemu: HIT BAD TRAP at pc = 0x801006f0
```

图 4.5 自陷事件测试结果

4.2 实现系统调用

系统调用和前面自陷操作类似。在 `__am_irq_handle()` 函数中添加系统调用。后面在 `do_event` 函数中调用 `do_syscall` 函数，实现相关的系统调用。

```
Context* am_irq_handle(Context *c) {
    Context *next = c;

    if (user_handler) {
        Event ev = {0};
        switch (c->cause) {
            case -1: // 自陷异常
                ev.event = _EVENT_YIELD;
                break;
            // 系统调用
            case 0: // SYS_exit
            case 1: // SYS_yield
            case 2: // SYS_open
            case 3: // SYS_read
            case 4: // SYS_write
            case 7: // SYS_close
            case 8: // SYS_lseek
            case 9: // SYS_brk
            case 13: // SYS_execve
                ev.event = _EVENT_SYSCALL;
                break;
            default: ev.event = _EVENT_ERROR; break;
        }
    }
}
```

图 4.6 添加新的系统调用参数

```
Context* do_syscall(Context *c) {
    uintptr_t a[4];
    a[0] = c->GPR1;
    a[1] = c->GPR2;
    a[2] = c->GPR3;
    a[3] = c->GPR4;

    switch (a[0]) {
        case SYS_yield:
            c->GPRx = sys_yield();
            break;
        case SYS_exit:
            sys_exit(a[1]);
            break;
        case SYS_write:
            c->GPRx = sys_write(a[1], (void*)(a[2]), a[3]);
            break;
        case SYS_brk:
            c->GPRx = sys_brk(a[1]);
            break;
        case SYS_read:
            c->GPRx = sys_read(a[1], (void*)(a[2]), a[3]);
            break;
        case SYS_lseek:
            c->GPRx = sys_lseek(a[1], a[2], a[3]);
            break;
        case SYS_open:
            c->GPRx = sys_open((const char *)a[1], a[2], a[3]);
            break;
    }
}
```

图 4.7 根据 event 的值调用对应的函数

使用 hello 测试程序测试系统调用

```
Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,15,main] Build time: 09:55:12, Jan 11 2022
[/home/hust/Desktop/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start =
[/home/hust/Desktop/ics2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/hust/Desktop/ics2019/nanos-lite/src/irq.c,20,init_irq] Initializing interrupt/excepti
[/home/hust/Desktop/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/Desktop/ics2019/nanos-lite/src/loader.c,29,naive_uoload] Jump to entry = 83000120
Hello World!
Hello World from Navy-apps for the 2th time!
Hello World from Navy-apps for the 3th time!
Hello World from Navy-apps for the 4th time!
Hello World from Navy-apps for the 5th time!
Hello World from Navy-apps for the 6th time!
```

图 4.8 hello 测试结果

4.3 实现文件系统和虚拟文件系统

1. 实现文件系统

增加几个系统调用，实现 `fs_open()`, `fs_read()`, `fs_close()`, `fs_write()` 和 `fs_lseek()` 函数。运行测试程序 `/bin/text`。成功输出了 `PASS`，测试结果符合预期。

```
Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,15,main] Build time: 20:36:43, Jan 11 2025
[/home/hust/Desktop/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size = -2146421148 bytes
[/home/hust/Desktop/ics2019/nanos-lite/src/device.c,55,init_device] Initializing devices...
[/home/hust/Desktop/ics2019/nanos-lite/src/irq.c,19,init_irq] Initializing interrupt/exception handler...
[/home/hust/Desktop/ics2019/nanos-lite/src/proc.c,27,init_proc] Initializing processes...
[/home/hust/Desktop/ics2019/nanos-lite/src/loader.c,49,naive_uoload] Jump to entry = 830002f4
PASS!!!
nemu: HIT GOOD TRAP at pc = 0x80100f58
[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 1688941
```

图 4.9 /bin/text 测试结果

2. 将设备输入抽象为文件

需要实现 `events_read()` 函数，根据按键码判断有无按键事件，然后将对应的信息拷贝到 `buf` 数组中。

```
size_t events_read(void *buf, size_t offset, size_t len) {
    int keycode = read_key();
    if ((keycode & 0xffff) == _KEY_NONE) { // 没有按键事件
        len = sprintf(buf, "t %d\n", uptime());
    } else if (keycode & 0x8000) { // 按下按键
        len = sprintf(buf, "kd %s\n", keyname[keycode & 0xffff]);
    } else { // 松开按键
        len = sprintf(buf, "ku %s\n", keyname[keycode & 0xffff]);
    }
    return len;
}
```

图 4.10 按键事件判断

运行测试程序/bin/events，程序一边输出事件的事件，点击按键的时候，会输出按键事件的信息。

```
receive event: ku S  
receive time event for the 297984th time: t 16000  
receive event: ku A  
receive time event for the 299008th time: t 16055  
receive time event for the 300032th time: t 16110  
receive event: kd D  
receive time event for the 301056th time: t 16167  
receive event: ku S  
receive time event for the 302080th time: t 16222  
receive event: kd A  
receive time event for the 303104th time: t 16278  
receive event: ku D  
receive event: kd W
```

图 4.11 /bin/events 测试结果

3. VGA 文件抽象为文件

根据手册中的指导完成相关文件，运行/bin/bmptest，成功输出图片，实验结果符合预期。

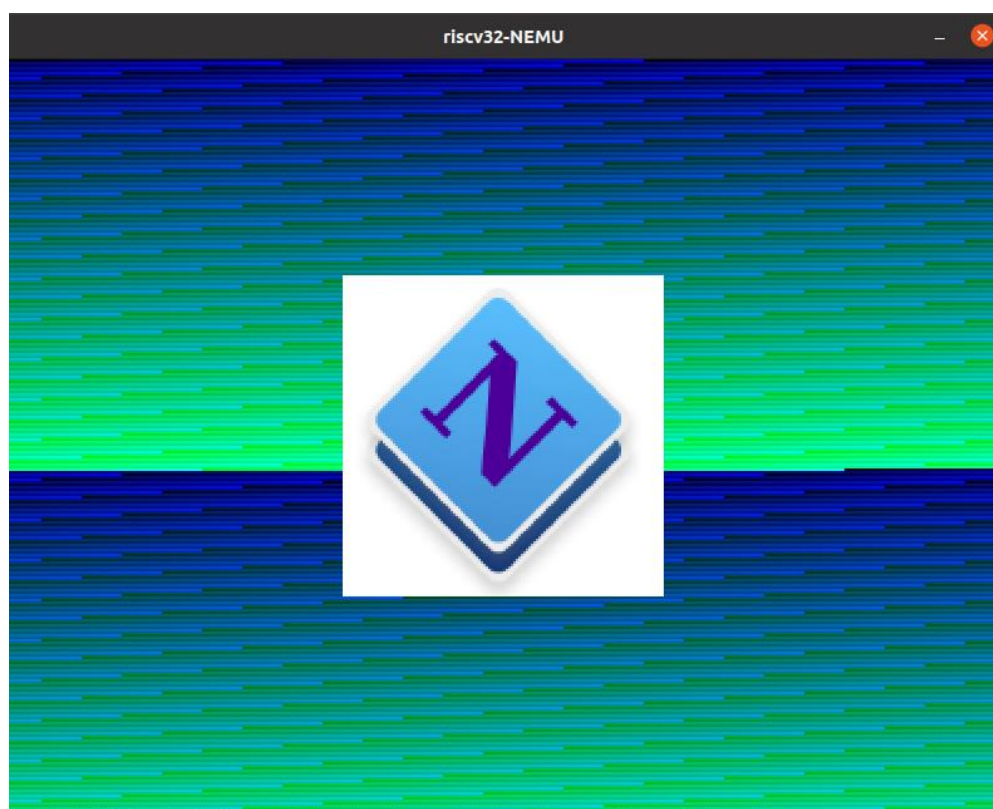


图 4.12 bmptest 运行结果

4.4 运行仙剑奇侠传

在 wiki 上下载对应的仙剑奇侠传的文件 pal，放到系统的对应文件位置，运行/bin/pal，运行结果符合预期。

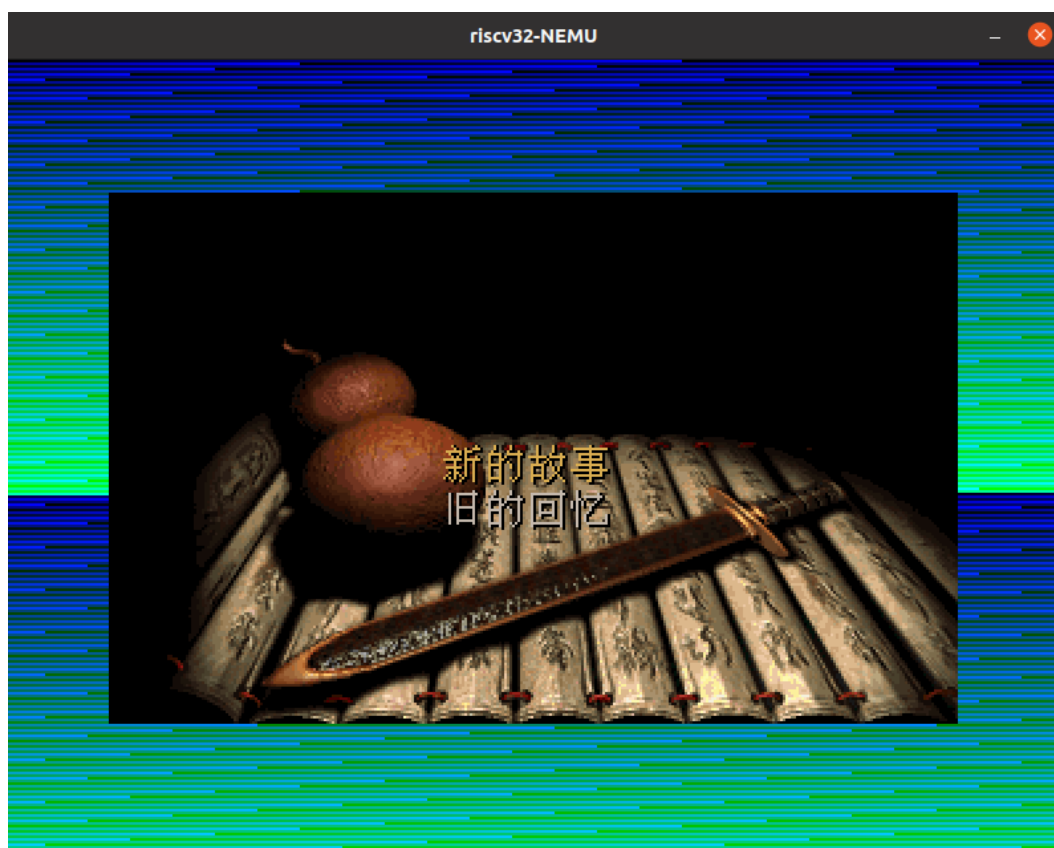


图 4.13 仙剑奇侠传运行结果

4.5 批处理系统

在 VFS 中添加特殊文件 tty，写入串口中，实现系统调用 SYS_execve，运行测试程序 init。

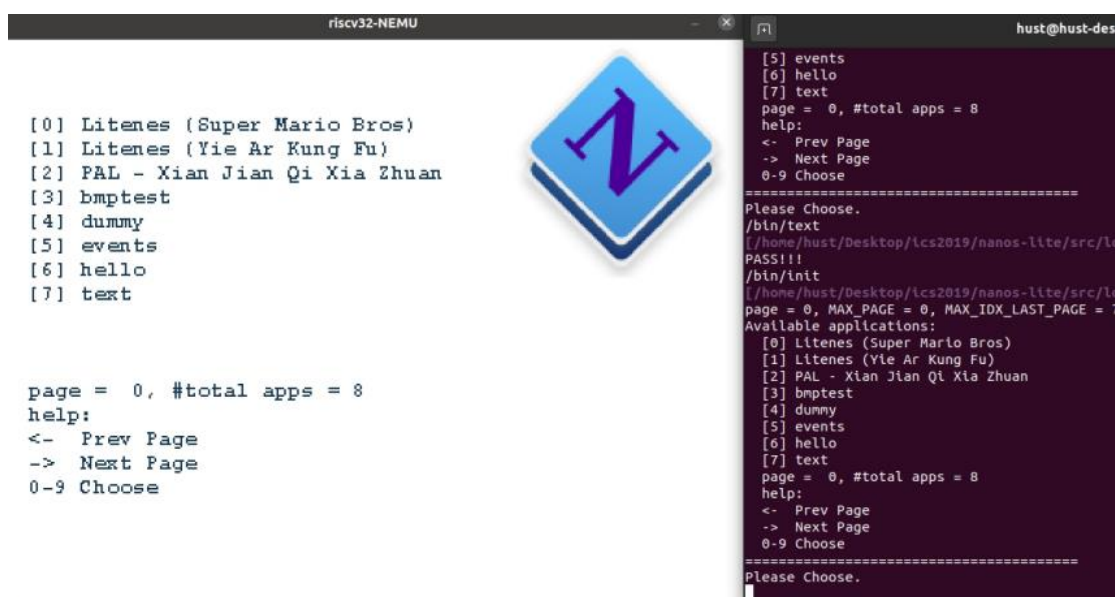


图 4.16 批处理系统

4.6 必答题

1. 理解上下文结构体的前世今生

`__am_irq_handle()` 中的 `c` 是指向上下文结构的指针。上下文结构在中断或异常发生时由汇编代码（`trap.S`）创建和填充。上下文结构的成员在汇编代码（`trap.S`）中被赋值。`riscv32-nemu.h` 定义了上下文结构，`trap.S` 实现了中断处理，NEMU 中的新指令涉及上下文切换。

2. 理解穿越时空的旅程

在 4.1 中对自陷操作有详细解释

3. `hello` 程序是什么，它从而何来，要到哪里去

编译完成后，`hello` 程序在磁盘上。当用户运行该程序时，首先操作系统有一个程序加载器，它将 `hello` 程序从磁盘读出，根据其 `elf` 头信息加载到指定的内存空间，加载完成后，操作系统从其 `elf` 信息中获取到程序入口地址，通过上下文

切换从该入口地址处继续执行。至此，hello 程序便获取到 CPU 的控制权开始执行指令。

至于字符串在终端的显示，首先程序调用 `printf` 等库函数，这些函数调用相应的系统调用，系统调用通过调用外设的驱动程序最终将内容在外设中表现出来。

4. 仙剑奇侠传究竟如何运行

运行仙剑奇侠传时会播放启动动画，动画中仙鹤在群山中飞过。这一动画是通过 `navy-apps/apps/pal/src/main.c` 中的 `PAL_SplashScreen()` 函数播放的。阅读这一函数，可以得知仙鹤的像素信息存放在数据文件 `mgo.mkf` 中。请回答以下问题：库函数, `libos`, `Nanos-lite`, `AM`, `NEMU` 是如何相互协助，来帮助仙剑奇侠传的代码从 `mgo.mkf` 文件中读出仙鹤的像素信息，并且更新到屏幕上？换一种 PA 的经典问法：这个过程究竟经历了些什么？

（1）读取文件：

当执行到 `PAL_SplashScreen()` 函数时，它将借助如 `fopen`、`fread` 等标准库函数，尝试打开并读取特定的 `mgo.mkf` 文件。这些库函数自身并不直接具备硬件操作能力，它们需要依托 `libos` 所提供的系统调用接口，向操作系统发起文件操作请求。

`libos` 接到请求后，通过相应的系统调用接口，与负责底层资源管理的 `Nanos-lite` 模块进行交互。`Nanos-lite` 针对接收到的文件系统请求展开处理，精准定位并读取 `mgo.mkf` 文件中的数据，随后将读取到的结果返回给 `libos`。

（2）解析像素信息：

成功读取文件后，`PAL_SplashScreen()` 函数会立即着手对从 `mgo.mkf` 文件中获取的像素信息进行解析，并将解析后的有效数据妥善存储在内存中，以供后续绘图使用。

（3）绘制图像：

完成像素信息解析存储后，`PAL_SplashScreen()` 函数下一步会调用图形库中的相关函数，以启动图像绘制流程。这些图形库函数在执行过程中，需要依赖

libos 提供的图形接口，来打通与底层硬件交互的通道。

libos 则凭借由 AM 提供的硬件抽象接口，实现与底层硬件设备的精准对接，将图形绘制指令及相关数据传递给硬件，驱动硬件进行图像绘制的前期准备工作。

（4）更新屏幕：

当图像绘制的前期准备就绪，AM 模块开始发挥作用，它通过操作帧缓冲区，将之前存储在内存中的像素信息有序写入屏幕，完成图像的最终呈现。

5 实验结果与结果分析

5.1 实验总结

本次课程设计中，我们完成了如下工作。

1. 实现了一个简单调试器，可以执行单步/多步运行、断点、表达式求值等功能。
2. 完成了 riscv32 虚拟机 nemu 的指令实现，并为其添加了 IO、中断、虚存等支持。
3. 完成了简易操作系统 nanos-lite，支持简单的进程管理、内存管理、文件系统等功能。

5.2 实验收获

在本次课程设计中，我收获颇丰，不仅在专业知识与技能层面得到了极大提升，更在问题解决与团队协作方面积累了宝贵经验。

在技术实践上，实现简单调试器是一项极具挑战性的任务。通过让其具备单步、多步运行、断点以及表达式求值等功能，我深入理解了程序运行的底层逻辑。每一次调试过程，都是对代码执行流程的精细剖析，让我明白了如何精准定位问题，快速修复错误，这对未来复杂程序的开发与维护至关重要。

完成 riscv32 虚拟机 nemu 的指令实现，并为其添加诸多关键支持，如 IO、中断、虚存等，更是拓宽了我的技术视野。深入到指令集架构层面，理解硬件与软件的交互机制，明白了如何从底层构建一个稳定高效的运行环境。这使我对计算机系统的整体架构有了更清晰的认知，不再局限于上层应用开发，而是能够深入底层探索原理。

简易操作系统 nanos-lite 的构建同样意义非凡。支持进程管理、内存管理、文件系统等功能，让我站在了系统设计者的角度思考问题。学习到如何合理分配系统资源，保障多个进程的有序运行，如何高效管理内存以避免资源浪费与冲突，以及如何构建一个可靠的文件系统来满足数据存储与读取需求。这些知识与技能，

为我今后从事系统开发、嵌入式开发等领域奠定了坚实基础。

总之，本次课程设计是一场知识与实践的深度融合之旅，让我将所学理论转化为实际成果，更激发了我对计算机领域深入探索的热情。

参考文献