

# 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:	CS2105 班
学 号:	U202113863
姓 名:	陈柳伊
电 话:	13429867981
邮 件:	u202113863@hust.edu.cn
完成日期:	2025-01-15

电子签名:

陈柳伊



# 目 录

1	课程实验概述 .....	1
1.1	实验目标 .....	1
1.2	模拟器功能 .....	1
1.3	最终目标 .....	1
1.4	实验方案 .....	1
2	实验方案设计 .....	2
2.1	概述 .....	2
2.2	PA1 .....	2
2.3	PA2 .....	6
2.4	PA3 .....	8
3	实验结果与结果分析.....	9
4	必答题 .....	15

# 1 课程实验概述

## 1.1 实验目标

本实验旨在通过在代码框架中构建一个简化的 RISC - V 模拟器，深入探索计算机系统的运行机制，全面提升自身的计算机系统能力。

## 1.2 模拟器功能

在本次实验中，我们所构建的 RISC - V 模拟器具备多项关键功能。首先，它能够对 RISC - V 执行代码进行可解释执行，这是模拟器运行程序的基础。同时，支持输入输出设备，使得模拟器与外部环境能够进行交互，如同真实计算机一样获取数据和输出结果。对于异常流处理，模拟器也有相应的支持，这确保了在程序运行过程中遇到意外情况时，能够进行合理的处理，保障系统的稳定性。

此外，该模拟器还支持精简操作系统，其中包括文件系统的支持，使得程序能够对文件进行读取、写入等操作，这对于模拟真实计算机软件环境至关重要。虚存管理的支持，让模拟器能够更高效地利用内存资源，提升程序运行的性能。进程分时调度功能的实现，则模拟了现代操作系统中多任务处理的能力，多个进程能够在有限的时间片内轮流执行，提高系统的整体利用率。

## 1.3 最终目标

本次实验的最终目标是在模拟器上成功运行经典游戏“仙剑奇侠传”。通过这一过程，我们能够深入探究“程序在计算机上运行”的机理。在这个过程中，我们需要全面掌握计算机软硬件协同的机制，理解硬件如何在软件的控制下完成各种操作，软件又是如何利用硬件资源来实现各种功能。

## 1.4 实验方案

我们深知理解“程序如何在计算机上运行”的根本途径是从“零”开始构建一个完整的计算机系统。现存的系统基础课程的小型项目（PA）为我们提供了良好的指导。PA 针对 x86/mips32/riscv32 架构提出了相应的教学版子集，并指导我们实现一个简化但功能完备的 x86/mips32/riscv32 模拟器 NEMU。NEMU 的设计受到了 QEMU 的启发，同时去除了大量与课程内容差异较大的部分，更适合我们在课程学习中进行实践。

PA 包括一个准备实验，即配置实验环境，这是开展后续实验的基础。后续还有 5 部分连贯的实验内容，分别是图灵机与简易调试器、冯诺依曼计算机系统、批处理系统、分时多任务以及程序性能优化。这 5 部分实验内容循序渐进，逐步引导我们从基础的概念开始，逐步构建出一个完整的、具备多种功能的计算机模拟器，最终实现运行“仙剑奇侠传”这一目标，进而梳理大学 3 年所学的全部理论知识，全面提升我们的计算机系统能力。

## 2 实验方案设计

### 2.1 概述

整个实验逐步引导完成一个计算机系统的构建，包括底层的 NEMU 模拟器，运行时环境 AbstractMachine (AM)，在其上的简易操作系统 NanOS-lite，以及操作系统上的应用程序库 Navy-apps。

一共分为 5 个部分，PA0 配置环境，PA1 完善 NEMU 的调试器功能，PA2 模拟 NEMU 指令运行以及补充 AM，PA3 完善 NEMU 的中断/异常处理、实现操作系统的系统调用以及简易文件系统功能，PA4 在操作系统中实现多道程序的运行、虚拟内存管理以及外部中断处理。

我完成了前三个实验任务，以下是细分每个实验的内容以及涉及到的源码部分。

实验	具体内容	涉及部分								
		NEMU			AM				NanOS-lite	Navy-apps
		调试器	模拟器	设备	klib	IOE	CTE	VME	kernel	libs
PA1.1	实现单步执行、寄存器和内存打印	✓								
PA1.2	实现表达式求值	✓								
PA1.3	扩展表达式求值，实现监视点	✓								
PA2.1	实现取指、译码、执行过程，运行简单程序		✓							
PA2.2	实现常用库函数功能 (string、sprintf) 实现各种 trace，以及 difftest 功能	✓	✓		✓					
PA2.3	理解 MMIO，基于串口设备实现 printf			✓	✓	✓				
	理解 IOE，实现时钟设备，进行跑分测试			✓		✓				
	实现设备访问 dtrace 实现键盘事件和 VGA 设备	✓		✓			✓			
PA3.1	实现异常响应机制		✓				✓		✓	
	完善上下文保存						✓		✓	
	实现异常事件分发处理以及上下文恢复						✓		✓	
PA3.2	实现 ELF 文件的加载								✓	
	实现 yield exit 系统调用，以及 strace								✓	✓
	实现 write brk 系统调用，进行输出以及堆区管理								✓	✓
PA3.3	实现简易文件系统，及相关的系统调用								✓	✓
	实现虚拟文件系统，将各种设备抽象成文件								✓	✓
	实现 NDL 库时钟和绘图功能								✓	✓
	实现定点算术、SDL 库，并运行各种程序								✓	✓
	实现 execve 系统调用								✓	✓

### 2.2 PA1

#### 2.2.1. 基础设施

##### PA1.1.1 单步执行

根据讲义中的提示，单步执行要求我们实现 si 命令，并且根据 si 命令后面的 argument 执行具体步数。讲义在 RTFSC 中提到 cpu-exec.c 模拟了 CPU 运行，转到其中能够轻易发现相关函数，只需要传入对应参数就可以实现要求的功能。于是在 sdb.c 中实现了相应的 cmd\_si 函数，值得一提的是用 sscanf 将指针 args 转化为了可以使用的变量。

##### PA1.1.2 打印寄存器

要求我们实现 info r 打印 32 个寄存器的值。根据提供的 API 文档可以发现有关寄存器的定义在 isa-reg.h 中，RTFSC 后发现寄存器的值是 gpr 表示，于是在 reg.c 中实现了对应的 API，即 isa\_reg\_display()。然后在 sdb.c 中实现对应的命令。值得一提的是 cmd\_info 在后续实现监视点的时候还会用到。

##### PA1.1.3 扫描内存

要求我们实现 `info x`，给定指定的其实内存和要打印的内存个数，输出内存数据。根据 RTFSC 中的提示在 `vaddr.c` 中找到了相应读取内存的函数 `vaddr_read()`，然后在 `sdb.c` 中通过循环实现具体指令，并且设定打印内存长度为 4。

### 2.2.2. 表达式求值

#### PA1. 2.1 词法分析

在表达式求值中只允许出现的 token 类型有十进制整数，加减乘除，括号和空格串。`expr.c` 中的 `enum` 编写了用于识别这些 token 的类型，具体识别规则见“正则表达式速览”。同一文件中的 `init_regex()` 函数会将这些规则编译成被库函数使用的 `pattern` 匹配信息，如果正则表达式语法有问题会触发 `assertion fail`。

`make_token()` 函数用于识别表达式中的 token，用 `position` 来表示当前处理的位置，按照顺序用不同的规则匹配当前位置的字符串，如果匹配成功 `log()` 则会输出识别成功的 token 类型，后面通过 `nr_token` 指示已经被识别出的 token 数目为当前正在处理到的 token 赋值，其中加减乘除此类只需要记录 token 类型，而数字则还需要记录 token 的具体内容，这个用 `substr_start` 和 `substr_len` 进行具体内容的赋值。

#### PA1. 2.2 递归求值

首先在 `sdb.c` 中实现 `cmd_p` 的指令，跳转到 `expr.c` 中的 `expr` 函数，再对 `expr` 函数进行具体编写，为了方便我重新写了一个新函数并在 `expr()` 中调用。新求值函数 `eval` 首先判断 token 的首位，`eval` 的递归采用两个位置变量 `p` 和 `q` 来表示正在处理的 token 位置。如果末位 token 的位置 `p` 小于首位 token 的位置 `q` 则判断表达式不合法；如果 `p` 等于 `q` 则说明递归到了某一个具体的数字，这个时候要对该单一 token 进行类型判断，如果不是数字类型则输出报错信息，反之直接返回数字的值。值得注意的是此处不能直接返回 `tokens` 结构体中的 `str` 部分，应使用 `strtol` 将 token 的具体内容转换成十进制整数后再返回。

如果  $p < q$  则先进行括号判断，此处我又调用了自己写的 `check_parentheses()` 函数。函数结构非常简单，如果此时的位置变量 `p` 和 `q` 代表的首位 token 和末位 token 分别是正括号和反括号，则开始正式判断。使用一个计数变量 `cnt` 来表示当前括号的个数，如果是正括号则加一，反括号则减一。当 `cnt` 为零时判断当前处理的 token 是否是末尾 token，以此确定首位 token 和末尾 token 的正反括号是成对的。因为首位是正括号且末位是反括号的情况有两种： $(\dots+\dots)+(\dots+\dots)$  和  $(\dots+(\dots+\dots))$ ，前者实际上并不需要 `check_parentheses`，因为主运算符仍然在括号外面。因此如果首位和末位括号成对则返回 `true`。最后如果 `cnt` 不为零的话返回 `false`。在 `eval()` 中如果 `check_parentheses()` 返回值为真，那么将 `p` 设为 `p+1`，`q` 设为 `q-1` 进行下一层的递归。

如果上面三个条件都没有满足则进行主运算符的判断，在此又要调用自己写的 `find_major()` 函数。讲义中提到了主运算符的几个特点：主运算符一定是运算符(某种意义上来说是句废话)、出现在一对括号中的 token 不是主运算符、主运算符的优先级在表达式中是最低的、当有多个运算符的优先级都是最低时最

后被结合的运算符才是主运算符。根据这几条规则我们就可以进行具体函数的编写，首先是定义计数变量 `cnt`、用于指示当前运算符优先级的 `op_type` 和主运算符的位置变量 `position`，我们用 `for` 循环进行 `p` 和 `q` 之间主运算符的判断。如果判断过程中遇到数字类型的 `token` 则直接跳过(根据第一条规则)，遇到正反括号则仿照 `check_parentheses` 中的思路利用 `cnt` 进行判断。如果是正括号则 `cnt++`。是反括号则 `cnt--`，并且判断 `cnt` 是否为零，如果等于零的话 `find_major()` 直接返回-1，因为表达式不合法。在反括号的情况下判断是因为此情况下 `cnt` 为零的只有两种情况：缺少了一个正括号或者多了一个反括号。比在正括号的情况下判断更加方便。如果 `cnt>0` 则 `continue`。然后进行运算符的判断，这里新建一个比较变量 `tmp_type`，初始值为零，根据运算符类型赋值，加减赋 2，乘除为 1(因为在表达式中运算符的优先级越低越可能成为主运算符，所以在这里优先级更高的被赋的值反而更低)。最后进行 `op_type` 和 `tmp_type` 的判断，如果 `tmp_type` 大于 `op_type` 则将其赋值给 `op_type`，并且记录此时的位置。如果 `tmp_type` 等于 `op_type` 同样将其赋值给 `op_type`(根据第四条规则)。循环结束后先进行 `cnt` 的判断，然后返回 `position` 变量。

在找完主运算符后就可以进行具体值的计算，设主运算符左边的变量为 `val1`，右边为 `val2`，继续递归，直到递归到  $(\pm*/)$ ，这是 `val1` 和 `val2` 递归的结果是其本身。用 `switch case` 进行运算符类型的判断，然后返回运算结果，这里除法要特别注意 `val2` 不能为零。

等到 `eval()` 函数计算完表达式的值后 `expr()` 函数就会接收到其返回的值，`expr()` 函数会直接将收到的值返回到 `sdb.c` 中的 `cmd_p()`，然后输出到屏幕上。

### 2.2.3. 监视点

#### PA 1.3.1 拓展表达式求值

拓展表达式求值新增了十六进制数、负数、打印寄存器、指针解引用和与或非、等于、不等于、大于、小于、大于等于、小于等于运算。与运算、或运算、等于、不等于、大于、小于、大于等于、小于等于的实现逻辑很简单。先编写匹配规则和 `make_token()`，然后在 `find_major()` 中增加计算优先级，这些运算的优先级都是最低的，所以在 `find_major()` 中的匹配优先度最高。最后在 `eval()` 函数中增加 `case` 计算即可。

真正麻烦的是十六进制数，打印寄存器，指针解引用，负数和非运算。其中负数和指针解引用还要更加麻烦，所以先阐释其他运算的实现。在这里我识别十六进制数的思路是匹配 `0x`，并且在 `TK_NUM` 的匹配规则中加 `A-F`，如此就可以将 `0x` 是为一个一元运算符进行处理，这里要注意 `0x` 的匹配规则要高于 `TK_NUM` 的匹配规则，否则就会出现只能匹配到 0 而不能匹配到 `x` 的问题。识别完成后正常编写 `make_token()`。然后在 `find_major()` 中编写优先级，一元运算符的计算优先级都是最高的，所以在 `find_major()` 中赋最低的值。最后在 `eval()` 中返回十进制的值，这里我调用了自己写的一个进制转换函数。主要思路就是按位取余，然后乘以该位的位权，累加到 `Dec` 变量中，也就是最后的十进制结果。至此十六进制数的实现已经完成。

然后是打印寄存器，仿照十六进制数的思路进行处理，一直到 `eval()` 函数部分，这里要返回的值就是 `cpu.gpr[i]`。非运算同理，`eval()` 函数部分返回的值是 `!val2`。

负数和指针解引用实际上使用的符号都是 “-” 或 “\*”，所以在 `make_token()` 阶段并不能将其和减法和乘法区分开来。所以我们编写特殊的 `check_unary()` 函数来检验。`check_unary()` 函数应该放在 `expr()` 函数的 `make_token()` 后面。其主要思路就是判定当前处理的 token 类型是不是 “\*” 或 “-”，如果是的话进行进一步判断，如果 token 编号是 0 则证明该运算符是表达式的第一个，直接将其 token 类型修改为负数或者指针解引用(注意这里必须要这么判断，如果全部使用后面那条规则判断的话会产生结构体访问越界)，如果 token 编号不为零的话就判断前一个 token 的类型是不是数字或者反括号，如果不是的话证明运算符是一元运算符，则将 token 类型修改为负数或者指针解引用。然后仿照之前对一元运算符的处理即可，`eval()` 函数中负数类型直接返回 `-value`，指针解引用类型则仿照 PA 1.1.3 进行处理即可。

### PA 1.3.2 监视点的管理

开始之前要为监视点结构体新增两个变量，一个是 `char *` 类型的 `expr` 和 `int` 类型的 `value`。

监视点的管理包括新建监视点 `new_wp()` 和删除监视点 `free_wp()`。在 `new_wp()` 中首先要判断监视点池是否为空，然后建立一个新的监视点结构体，将空闲监视点赋值给新建的结构体，然后将空闲监视点向后移动一位，将 `head` (正在处理的监视点) 设定为新建的监视点。

在 `free_wp()` 中首先判断要删除的监视点是不是 `head`，如果是的话就直接将 `head` 设为 `NULL`。新建一个监视点结构体 `flag`，将其设为监视点池的第一个监视点，也就是监视点 0，一个一个往后推，直到推到要删除的监视点的前一个监视点，将 `flag` 的下一个设为要删除的监视点的下一个，也就是暂时将要删除的监视点抽出监视点池。

### PA 1.3.3 监视点功能的实现

首先还是在 `sdb.c` 中编写相关的指令 (`info w`、`cmd_w`、`cmd_d`)。`info w` 的实现很简单，就是新增一个条件判断，如果输入为 `w` 的话调用 `display_wp()` 函数 (具体函数一会在 `watchpoint.c` 中实现)。

然后编写 `cmd_w` 指令，相关细节仿照 PA1，将要计算的表达式读入并调用 `expr()` 计算，将计算结果赋值给 `result` 变量，再将表达式和 `result` 传给 `set_wp()` 函数 (同样一会在 `watchpoint.c` 中实现)。

`cmd_d` 指令也仿照 PA1 进行编写，将要删除的监视点编号传入 `delete_wp()` 函数即可。

然后是 watchpoint.c 中监视点功能的具体实现，包括 set\_wp()、delete\_wp()、display\_wp() 和 check\_wp()。set\_wp() 的实现很简单，先建立一个新的监视点结构体，将接收到的表达式用 strdup 函数复制到新监视点的 expr 部分，将 result 赋值给新监视点的 value 部分，最后输出监视点信息。delete\_wp() 首先判断输入编号是否小于 NR\_WP，这里我用 assert 实现。然后建立三个监视点结构体，分别是要删除的监视点，要删除的监视点的前一个监视点和要删除的监视点的后一个监视点。建立完后先输出要删除监视点的具体信息，然后将要删除的监视点传入 free\_wp()，因为在 free\_wp() 函数中将要删除的监视点抽出了监视点链表，所以调用完函数后要将要删除的监视点的 expr 部分设为 NULL，将要删除的监视点的前一个监视点只想删除的监视点，删除的监视点指向删除的监视点的后一个监视点，到这里删除监视点的操作就正式完成了。

下一个是 display\_wp() 函数的编写，首先新建一个监视点结构体，将其设置为监视点池中的第一个监视点，然后用 while 循环(循环条件设置为新建立的监视点是否成立)打印监视点信息，在循环的最后将监视点设置为下一个监视点即可。

最后是 check\_wp() 的编写，首先还是新建一个监视点结构体，将其设置为监视点池中的第一个监视点，然后用 while 循环处理(循环条件设置为新建的监视点不为 head 的下一个监视点)，在循环中调用 expr() 函数计算表达式的值，如果表达式的值发生变化则输出相关信息，在循环的最后将监视点设置为下一个监视点。最后将 check\_wp() 放进 cpu-exec.c 的 execute() 中，每执行一条指令就检查一次。

## 2.3 PA2

### 2.3.1. PA2.1

这部分是完善 NEMU 的译码、执行过程。具体来说都是通过宏来实现的。具体来讲就是定义了这么几个宏 / 函数：

```
__attribute__((always_inline))
static inline void pattern_decode(const char *str, int len,
    uint64_t *key, uint64_t *mask, uint64_t *shift) {
    ... // 从 str 中解析出 key, mask, shift
    // 对应位如果是 1 则 key 该位为 1, 如果是 ? 则 mask 该位为 0
    // 最终效果为匹配上的指令 (inst >> shift) & mask == key
}

#define INSPAT_START(name) { const void ** __instpat_end = &&concat(__instpat_end_, name);
#define INSPAT_END(name) concat(__instpat_end_, name); ; }
#define INSPAT_INST(s) ((s)→isa.instr.val)
#define INSPAT_MATCH(s, name, type, ... /* execute body */) { \
    decode_operand(s, &dest, &src1, &src2, &imm, concat(TYPE_, type)); \
    __VA_ARGS__; \
}

#define INSPAT(pattern, ...) do { \
    uint64_t key, mask, shift; \
    pattern_decode(pattern, STRLEN(pattern), &key, &mask, &shift); \
    if (((uint64_t)INSPAT_INST(s) >> shift) & mask) == key) { \
        INSPAT_MATCH(s, ##__VA_ARGS__); \
        goto *(__instpat_end); \
    } \
} while (0)
```



然后是两个函数，一个用来根据类型解码操作数，一个用来解码指令（包括译码和执行）：

```
static void decode_operand(Decode *s, int *dest, word_t *src1, word_t *src2, word_t *imm,
uint32_t i = s->isa.instr.val;
int rd = BITS(i, 11, 7);
int rs1 = BITS(i, 19, 15);
int rs2 = BITS(i, 24, 20);
*dest = rd;
switch (type) {
    case TYPE_I: src1R();      immI(); break;
    case TYPE_U:      immU(); break;
    case TYPE_S: src1R(); src2R(); immS(); break;
    case TYPE_R: src1R(); src2R();      break;
    case TYPE_B: src1R(); src2R(); immB(); break;
    case TYPE_J:      immJ(); break;
}
}

static int decode_exec(Decode *s) {
    int dest = 0;
    word_t src1 = 0, src2 = 0, imm = 0;
    s->dnpc = s->snpc;
    INSTPAT_START();
    INSTPAT("00000000 ????? ????? 000 ????? 01100 11", add, R, Reg(dest) = src1 + src2);
    ...
    INSTPAT_END();
    Reg(0) = 0;
    return 0;
}
```

INSTPAT\_START 和 INSTPAT\_END 两个宏构成了一组大括号，在末尾创建了一个标号，并在开头获取了其地址

INSTPAT 宏展开后首先通过 pattern\_decode 获取 key mask shift，然后通过 INSTPAT\_INST 获取指令值并检测是否匹配

如果匹配则进入 INSTPAT\_MATCH 宏中，然后 goto 到结尾标号结束当前指令

INSTPAT\_MATCH 宏中首先调用 decode\_operand 函数，根据传入的指令类型提取对应的操作数到 decode\_exec 的局部变量中

然后 \_\_VA\_ARGS\_\_ 将最后一个参数展开并执行，也就是指令的执行部分

通过 Reg 宏即可访问读写寄存器。

### 2.3.2. PA2.2

PA2.2 前半部分在实现一些库函数内容，包括 string.h 的一些函数（实现 memcpy 的时候出现了一个潜在 bug，后面 PA3.3 再说），以及 stdio.h 的 sprintf 函数，后面 PA2.3 结合串口设备实现 printf 函数。

### 2.3.3. PA2.3

RISC-V 通过 MMIO 来实现设备通信。原理大概就是 NEMU 中各个设备分别创建一些空间，然后将这些空间映射到指定的地址上，并且注册一个 callback 函数。在访存的时候如果地址不落在物理地址范围内，则尝试搜索地址是否在某一个设备映射的空间中，如果是的话，对于读取先调用 callback，这时可以在实际读取数据之前更新值，对于写入则写入后调用 callback 函数处理新的值。

对于串口，相应代码已经提供好了，callback 函数会要求一定是写入，且写入的长度一定是一个字节。接收到一个字节后则通过 putchar 来输出。在 AM 中，TRM 部分就提供了 putchar 函数，函数体就只有 outb(SERIAL\_PORT, ch); 来向串口地址写入字节进行输出。

根据这个功能就可以实现 `printf` 了。由于 `printf`、`sprintf` 工作原理都类似，区别在于输出的位置，所以可以将共用的部分提取出单个函数 `vprintf`，并且接收一个函数用来输出单个字符。由于 `printf` 参数不固定，所以就需要使用 `va_` 相关的功能了。`printf` 和 `sprintf` 本身就是接收一下参数然后传给 `vprintf`（在系统实验里学来的）。

时钟其实写起来问题不大的，不过做时钟的时候正是需要 `printf` 64 位无符号数的时候，这时发现了上面说到的 `printf` 的 bug，调试起来就相当痛苦了。

通过阅读源码发现，NEMU 里 `timer` 的 `callback` 函数只有在 `offset == 4` 也就是读取高 32 位的时候会更新时间，所以不更改这里的话就需要先读取高 32 位再读取低 32 位，然后合并得到 64 位时间戳。

## 2.4 PA3

这一部分要完善一个简易的文件系统（内容固定、大小固定、没有目录……），然后将设备抽象为虚拟文件，再完善 `Navy-apps` 的一系列库来运行起各种各样的程序。

### 文件系统

这里的文件系统整体存储在 `ramdisk.img` 中，通过 `resource.S` 中的 `.incbin` 加载进来。然后每一个文件都可以通过一个结构体来记录：

```
typedef size_t (*ReadFn) (void *buf, size_t offset, size_t len);
typedef size_t (*WriteFn) (const void *buf, size_t offset, size_t len);
typedef struct {
    char *name;
    size_t size;
    size_t disk_offset;
    ReadFn read;
    WriteFn write;
    size_t open_offset;
} Finfo;
```

然后通过一个 `Finfo` 的数组来在文件描述符和 `Finfo` 之间进行映射。这里还有一个我觉得很巧妙的地方，就是在编译 `Navy-apps` 的过程中，所有可执行文件和素材都会存在 `fsimg` 文件夹下，然后 `Makefile` 中的脚本会将其打包成一个 `ramdisk.img`，然后同时提取出 `name`、`size`、`disk_offset` 的信息格式化到一个 `files.h` 中。在定义 `Finfo` 数组的时候直接 `include` 进来，就在编译时完成了文件系统全部内容的初始化。

对于虚拟文件，直接定义的时候加上对应的名字，然后指定特有的 `read` 和 `write`，在执行 `fs_read` `fs_write` 的时候如果 `read` `write` 不为 `NULL` 则调用，否则调用 `ramdisk_read` `ramdisk_write`，这样就实现了这个文件系统的基本功能。除此之外 `Finfo` 还需要维护 `open_offset`，这里我当时理解错了，实际上应该是距离 `disk_offset` 的偏移，而我理解为了在 `ramdisk` 中整体的偏移，不过实现后的最终效果还是一样的。

### 3 实验结果与结果分析

PA1

不同指令的测试

1. 帮助指令 help

```
(nemu) help
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - Single step
info - Show info of regs/watchpoint
p - Evaluate given expression
x - Scan memory
w - Set watchpoint
d - Remove watchpoint
(nemu) █
```

2. 单步执行 si

```
(nemu) st
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) █
```

3. 表达式求值

```
(nemu) p ((20 * 30) - 100 + ($t0 + 10) * ($t0 * 10000))
结果为500
(nemu) p (1+2)*(4/3)
结果为3
(nemu) p (3/3+)(123*4
[src/monitor/debug/expr.c,306,eval] 求值失败
[src/monitor/debug/expr.c,334,eval] main_op: 求第一个子表达式出错
[src/monitor/debug/ui.c,162,cmd_exp] cmd_exp: 求值失败
```

4. 监视点相关

```
(nemu) w $t0 == 0
[src/monitor/debug/watchpoint.c,26,new_wp] 新建的监视点的value为$t0 == 0
(nemu) w 100 * $s11 * 10
[src/monitor/debug/watchpoint.c,26,new_wp] 新建的监视点的value为100 * $s11 * 10
(nemu) info w
[src/monitor/debug/ui.c,120,cmd_info] 进入info
WP NO.1 "100 * $s11 * 10" value=0
WP NO.0 "$t0 == 0" value=1
(nemu) d 0
(nemu) info w
[src/monitor/debug/ui.c,120,cmd_info] 进入info
WP NO.1 "100 * $s11 * 10" value=0
(nemu) █
```

所有功能均实现完毕。

PA2

## 1. runall.sh 回归测试脚本

```
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!
→ nemu git:(pa2) █
```

## 2. Microbench

```

Empty mainargs. Use "ref" by default
===== Running MicroBench [input *ref*] =====
[qsrt] Quick sort: * Passed.
  min time: 5516 ms [92]
[queen] Queen placement: * Passed.
  min time: 5324 ms [88]
[bf] Brainf**k interpreter: * Passed.
  min time: 39743 ms [59]
[fib] Fibonacci number: * Passed.
  min time: 428913 ms [6]
[sieve] Eratosthenes sieve: * Passed.
  min time: 130769 ms [30]
[15pz] A* 15-puzzle search: * Passed.
  min time: 20424 ms [21]
[dinic] Dinic's maxflow algorithm: * Passed.
  min time: 10910 ms [99]
[lzip] Lzip compression: * Passed.
  min time: 21878 ms [34]
[ssort] Suffix sort: * Passed.
  min time: 5681 ms [79]
[md5] MD5 digest: * Passed.
  min time: 65710 ms [26]
=====
MicroBench PASS      53 Marks
                        vs. 100000 Marks (i7-7700K @ 4.20GHz)
Total time: 775686 ms
nemu: HIT GOOD TRAP at pc = 0x801072a0

[src/monitor/cpu-exec.c,30,monitor_statistic] total guest instructions = 9054015931
make[1]: Leaving directory '/home/samuel/U201714668/nemu'
→ microbench git:(pa3)

```

### 3. ppt 演示

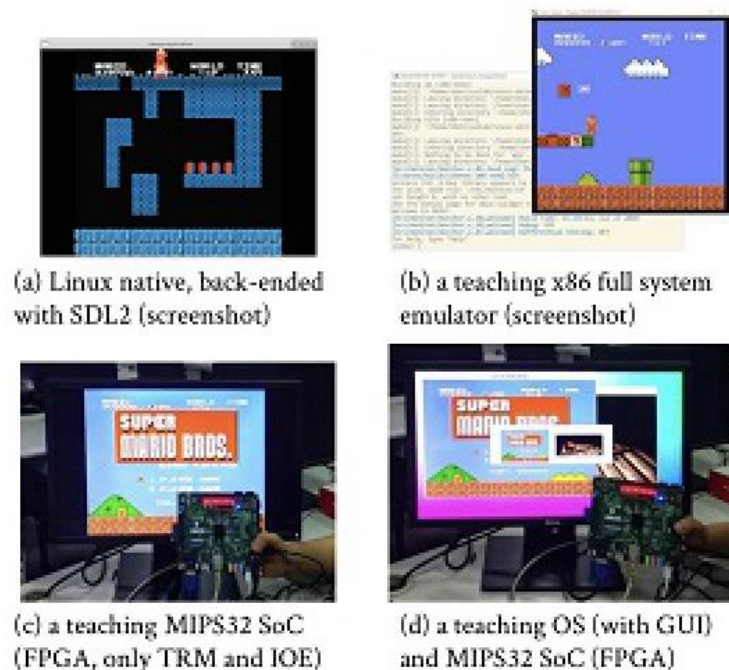
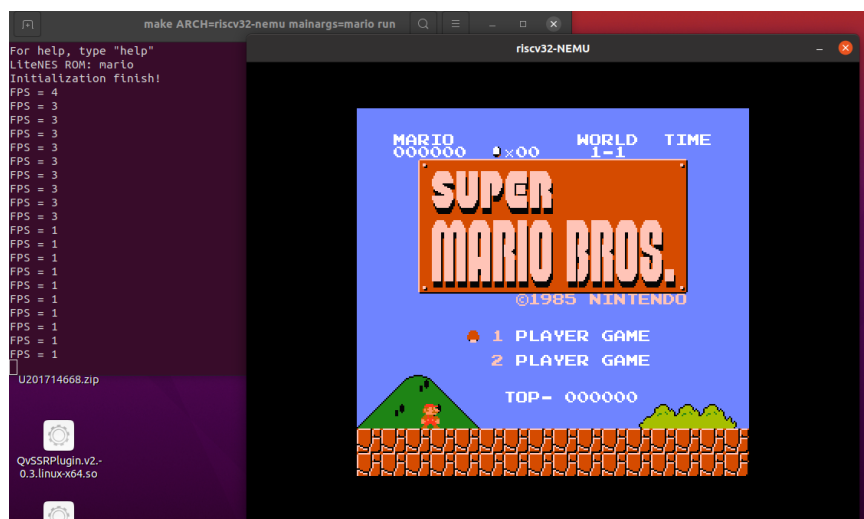


Figure 4. The same LiteNES emulator running on different platforms.

### 4. 打字游戏



## 5. litenes



## PA3

### 1. dummy 测试

```

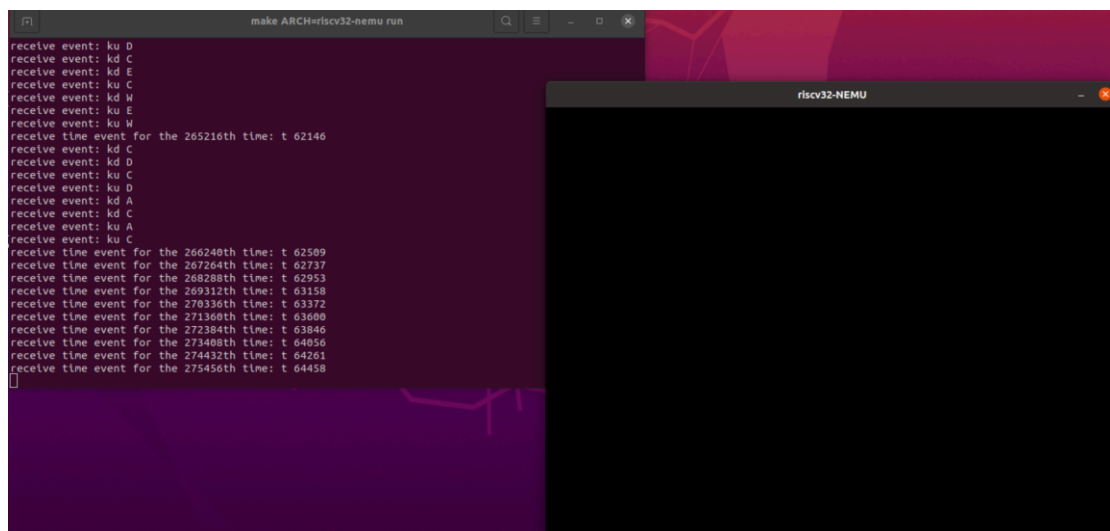
Welcome to riscv32-NEMU!
For help, type "help"
Hello! This is Samuel[from nanos-lite/src/logo.txt file][./media/sf_VirtualBox_Shared_Folder/U201714668/nanos-lite/src/main.c,14,main] 'Hello World!' from
[./media/sf_VirtualBox_Shared_Folder/U201714668/nanos-lite/src/main.c,15,main] Build time: 00:12:21, Jan 14 2021
[./media/sf_VirtualBox_Shared_Folder/U201714668/nanos-lite/src/ramdisk.c,29,init_ramdisk] ramdisk info: start = 0x80104a18, end = 0x82297455, size = 3520
[./media/sf_VirtualBox_Shared_Folder/U201714668/nanos-lite/src/device.c,61,init_device] Initializing devices...
[./media/sf_VirtualBox_Shared_Folder/U201714668/nanos-lite/src/irq.c,27,init_irq] Initializing interrupt/exception handler...
[./media/sf_VirtualBox_Shared_Folder/U201714668/nanos-lite/src/proc.c,27,init_proc] Initializing processes...
[./media/sf_VirtualBox_Shared_Folder/U201714668/nanos-lite/src/loader.c,53,naive_uload] Jump to entry = 83000110
self int
nemu: HIT GOOD TRAP at pc = 0x801017d0
[./src/monitor/cpu-exec.c,30,monitor_statistic] total guest instructions = 1759233
  
```

### 2. hello 测试



```
Hello World from Navy-apps for the 106th time!  
Hello World from Navy-apps for the 107th time!  
Hello World from Navy-apps for the 108th time!  
Hello World from Navy-apps for the 109th time!  
Hello World from Navy-apps for the 110th time!  
Hello World from Navy-apps for the 111th time!  
Hello World from Navy-apps for the 112th time!  
Hello World from Navy-apps for the 113th time!  
Hello World from Navy-apps for the 114th time!  
Hello World from Navy-apps for the 115th time!  
Hello World from Navy-apps for the 116th time!  
Hello World from Navy-apps for the 117th time!  
Hello World from Navy-apps for the 118th time!  
Hello World from Navy-apps for the 119th time!  
Hello World from Navy-apps for the 120th time!  
Hello World from Navy-apps for the 121th time!  
Hello World from Navy-apps for the 122th time!  
Hello World from Navy-apps for the 123th time!  
Hello World from Navy-apps for the 124th time!  
Hello World from Navy-apps for the 125th time!  
Hello World from Navy-apps for the 126th time!  
Hello World from Navy-apps for the 127th time!  
Hello World from Navy-apps for the 128th time!  
Hello World from Navy-apps for the 129th time!  
Hello World from Navy-apps for the 130th time!  
Hello World from Navy-apps for the 131th time!  
Hello World from Navy-apps for the 132th time!  
Hello World from Navy-apps for the 133th time!  
Hello World from Navy-apps for the 134th time!
```

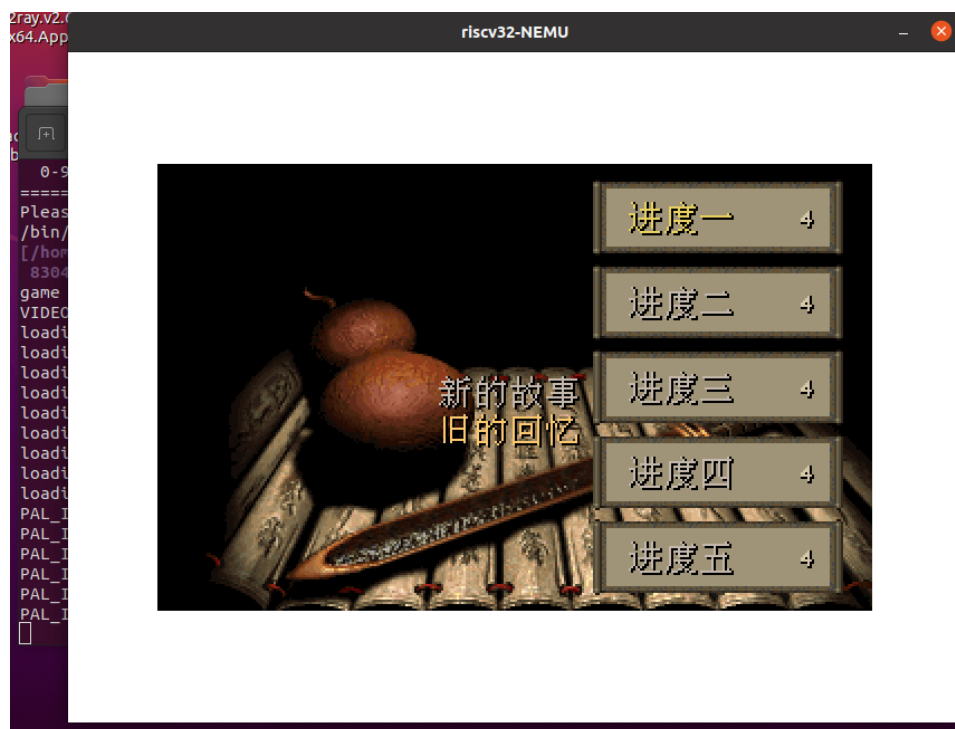
### 3. events 测试



```
make ARCH=riscv32-nemu run  
receive event: ku D  
receive event: kd C  
receive event: kd E  
receive event: ku C  
receive event: kd W  
receive event: ku E  
receive event: ku W  
receive time event for the 265216th time: t 62146  
receive event: kd C  
receive event: kd D  
receive event: ku C  
receive event: ku D  
receive event: kd A  
receive event: kd C  
receive event: ku A  
receive event: ku C  
receive time event for the 266240th time: t 62509  
receive time event for the 267264th time: t 62737  
receive time event for the 268288th time: t 62953  
receive time event for the 269312th time: t 63158  
receive time event for the 270336th time: t 63372  
receive time event for the 271360th time: t 63600  
receive time event for the 272384th time: t 63846  
receive time event for the 273408th time: t 64056  
receive time event for the 274432th time: t 64261  
receive time event for the 275456th time: t 64458
```

### 4. 运行仙剑奇侠传

加载存档:





## 4 必答题

PA1

1. 我选择的 ISA 是 riscv32

2. 理解基础设施

一学期将要花费 75 个小时的时间在调试上。

一学期可以节省 50 个小时调试的时间。

3. 查阅 riscv32 手册

riscv32 有哪几种指令格式?

6 种

LUI 指令的行为是什么?

根据文档中的说明

LUI 指令的行为是高位立即数加载 mstatus 寄存器的结构是怎么样的?

根据文档可知 mstatus 寄存器的结构如下所示 4. 完成 PA1 的内容之后,

nemu/ 目录下的所有 .c 和 .h 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 pa1 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?

通过 `git checkout pa1` 回到刚完成 pa1 的状态

在 nemu/ 目录下输入 `find nemu -name "*.c" | xargs wc -l`

即可输出 nemu 目录下所有 .c 文件以及 .h 文件的行数。

对于空行可以通过以下指令来去除。

利用同样的方法也可以去除所有的空行以及注释，在此不再赘述。

5. 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？

-Wall 打开 gcc 所有的警告 -Werror 将所有的警告当成错误进行处理

使用 -Wall 以及 -Werror 有利于发现代码中不严谨之处，如类型的隐式转换

```
find nemu -name "*.hc" | xargs wc -l
```

```
find nemu -name "*.hc" | xargs cat | sed '/^\s*$/d' | wc -lPA2
```

1. 请整理一条指令在 NEMU 中的执行过程。(我们其实已经在 PA2.1 阶段提到过这道题了)

以 lw 指令为例

2. 在 nemu/include/rtl/rtl.h 中，你会看到由 static inline 开头定义的各种 RTL 指令函数。选择

其中一个函数，分别尝试去掉 static，去掉 inline 或去掉两者，然后重新进行编译，你可能会看到

发生错误。请分别解释为什么这些错误会发生/不发生？你有办法证明你的想法吗？

去掉 static 不会发生错误

去掉 inline，编译时不会发生错误，链接时会发生错误。LD 在链接不同的 object 时，会发现同一个符号在不同的 object 中被多次定义，并且我们没有告诉 LD 这种情况下应该怎么进行链接，因此报错

### 3. 编译与链接

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？

一个添加上题中的代码后，再在 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.

两个

static 关键字表示变量只有在当前文件可以被识别, volatile 表示这处代码不会被编译器优化, 而 debug.h 通过 include 包含了 common.h, 所以重新声明的变量不会覆盖之前的 dummy 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.)

``isa_exec``函数被执行

|

|==> 通过``instr_fetch``函数获得当前``pc``值对应的内存位置中的指令

调用``idex``, 也即``译码-执行``函数

|

|==> 调用``decode_ld``函数, 获得操作数, 将其保存在全局变量``id_src``以

及

``id_dest``中

调用``exec_load``函数,

|

|==> 根据``funct3``字段, 索引``load_table``, 找到对应的执行函数

``exec_ld``,

同时设定位宽 width

调用函数``exec_ld``

|

|==>根据``lw``指令的逻辑, 调用`rtl` 函数

调用``rtl_lm``将 `id_src.addr` 所指向的地址取出 `decinfo.width` 宽度的数据, 保存在临时寄存器 `s0` 中

调用``rtl_sr``将 `s0` 保存在 `id_dest.reg` 指向的寄存器中

根据宽度调用`print_asm_template2` 打印具体的汇编指令出现重定义

给变量赋值之后, 声明就变成了定义, 所以会发生重定义问题

4. 请描述你在 nemu/ 目录下敲入 make 后, 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` . (这个问题包括两个方面:

`Makefile` 的工作方式和编译链接的过程.)

通过阅读文档中给出的 C 语言基础中的 `makefile` 基础一节以及 `man` 文档, 可以对 `makefile` 有一个基本的了解对于 `Makefile`, 基本的工作方式如下

1. 首先依次读取变量 “`MAKEFILES`” 定义的 `makefile` 文件列表
2. 读取工作目录下的 `makefile` 文件
3. 一次读取工作目录下的 `makefile` 文件指定的 `include` 文件
4. 查找重建所有已读的 `makefile` 文件的规则
5. 初始化变量值并展开那些需要立即展开的变量和函数并根据预设条件确定执行分支

6. 建立依赖关系表
7. 执行 `rules`

具体到 `/nemu/Makefile` 这个文件本身, 具体的编译链接过程如下

1. 首先 `Makefile` 默认的 `ISA` 为 `x86`, 如果定义了具体的 `ISA`, 则需要保证其有效

2. 根据 `ISA` 确定需要 `include` 的文件列表

3. 确定编译目标文件夹 (默认是 `build/`) 确定编译器以及链接器

( `gcc` )

4. 设置编译选项 `CFLAGS`

5. 读取所有需要编译的 `.c` 的文件并将其编译为 `.o` 文件

6. 进行链接

7. 执行 `git commit`

**PA3**

1. 理解上下文结构体的前世今生 (见PA3.1 阶段)

你会在 `__am_irq_handle()` 中看到有一个上下文结构指针 `c`, `c` 指向的上下文结构究竟在哪里? 这个上下文结构又是怎么来的? 具体地, 这个上下文结构有很多成员, 每一个成员究竟在哪里赋值的? `$ISA-nemu.h`, `trap.S`, 上述讲义文字, 以及你刚刚在 NEMU 中实现的新指令, 这四部分内容又有什么联系? `__am_irq_handle` 函数在整个项目中, 唯一出现的调用文件就是 `trap.S`。其中通过 `jal` 指令直接跳转到对应的函数体, 而函数的参数是保存在栈当中的, 可以看到在 `jal` 指令之前有诸多压栈操作, 按照顺序是

1. 32 个寄存器, 通过 `MAP(REGS, PUSH)`
2. 成员 `cause`, 通过 `sw t0 OFFSET_CAUSE(sp)`
3. 成员 `status`, 通过 `sw t1 OFFSET_STATUS(sp)`
4. 成员 `epc`, 通过 `sw t2 OFFSET_EPC(sp)`

于是不同的成员被赋值。可以在 `__am_irq_handle` 函数中被使用

## 2. 理解穿越时空的旅程 (见PA3.1 阶段)

从 Nanos-lite 调用 `_yield()` 开始, 到从 `_yield()` 返回的期间, 这一趟旅程具体经历了什么? 软(AM, Nanos-lite)硬(NEMU)件是如何相互协助来完成这趟旅程的? 你需要解释这一过程中的

每一处细节, 包括涉及的每一行汇编代码/C 代码的行为, 尤其是比较关键的指令/变量. 事实上, 上文的必答题"理解上下文结构体的前世今生"已经涵盖了这趟旅程中的一部分, 你可以把它的回答包含进来.

首先 `_yield` 函数, 通过内联汇编代码将 `a7` 设置为 -1, 表示当前的 `ecall` 类型是 `_yield`, 接着

执行了 `ecall` 指令。

汇编 `ecall` 指令将会由 `ecall` 对应的 `EHelper` 来执行相关的函数, 函数中会调

用 `raise_intr` 函数, 参数 `NO` 即为 `a7` 寄存器的值, 表示中断号。

在 `raise_intr` 函数中会保存 `epc` 到 `sepc` 寄存器, 将中断号保存到 `scause`

寄存器，并从 `stvec` 获得中断入口地址并进行跳转。也就是 `__am_asm_trap` 函数的入口地址，也就是汇编代码 `trap.S` 中的起始位置。开始执行。

汇编代码会执行到上述的 `__am_irq_handle` 函数。`__am_irq_handle` 函数根据 `c->cause` 来分别进行处理，如果是 `-1` 就表示 `yield` 事件，如果是 `0` 到 `19`(支持的系统调用的个数)就说明是系统调用。此处是 `yield`，于是填充 `ev.event` 成员

为 `_EVENT_YIELD` 并调用用户定义的回调函数 `do_event`

同样是根据 `event` 的类型来分别处理，如果是 `_EVENT_YIELD` 就打印出信息到终端，如果是 `_EVENT_SYSCALL` 的话就调用 `do_syscall`，此处是打印信息到终端

函数结束之后将会回到 `trap.S` 汇编代码。恢复上下文并调用 `sret` 指令

`sret` 指令将会调用 `nemu` 中针对 `sret` 指令的执行函数，从 `sepc` 寄存器中读出之前保存的 `pc`，将其加 `4`，表示中断发生时的吓一跳指令的地址，并进行跳转

至此 `yield` 函数执行完毕

### 3. `hello` 程序是什么，它从而何来，要到哪里去 (见PA3.2 阶段)

我们知道 `navy-apps/tests/hello/hello.c` 只是一个 C 源文件，它会被编译链接成一个 ELF

文件。那么，`hello` 程序一开始在哪里？它是如何出现内存中的？为什么会出现在目前的内存位置？它的第一条指令在哪里？究竟是怎么执行到它的第一条指令的？`hello` 程序在不断地打印字符串，每一个字符又是经历了什么才会最终出现在终端上？

`hello` 程序对应的 `elf` 文件会在整个项目编译的时候，将其在 `ramdisk` 的偏移位置保存在 `files.h` 的记录表中(`disk_offset` 成员)。接着通过 `load` 函数解析对应的 `elf` 文件，得到具体的入口地址，保存在 `e_entry` 中

通过 `((void(*)())entry)()` 跳转到对应位置进行执行。

在 main 函数中通过 printf 进行输出，printf 函数首先会尝试进行 \_brk，如果失败则一个字符一个字符的通过 write 输出到终端，如果成功则将字符串作为整体调用 write 进行输出。write 函数会调用 \_write 系统调用，在对应的处理函数中，发现输出的对象是 stdout，则直接通过 serial\_write 进行输出。

4. 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 navyapps/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？

在 navy-apps/apps/palc/hal/hal.c 中的 redraw 函数中通过 NDL\_DrawRect 和 NDL\_Render 更新屏幕。NDL 通过之前的初始化操作维护了一块画布 canvas，并将其绘制操作限定在该画布上。NDL\_DrawRect 会将传入的 pixels 保存到会把传入的像素逐个存入画布的对应位置首先调用了 libndl 库，在该库中会打开设备文件 /dev/fb 和 /dev/fbsync，在接收到该函数调用后会向 /dev/fb 设备文件中写入，在该函数中，判断出要写的文件是 /dev/fb 设备文件之后，会调用 fb\_write 帮助函数，之后会调用 draw\_rect 函数，该函数位于 nexus-ambsibc/io.c 中，在函数内会调用 \_io\_write 函数，而 \_io\_write 会转发给 \_\_am\_video\_write 函数，该函数中会执行 out 汇编指令，将数据传送给 vga 设备中。vga 设备在接收到数据后会保存在定义的显存中，当之后 NDL 库向 /dev/fbsyn 设备文件中写入时，vga 设备最终会调用 SDL 库来更新画面。NDL\_Render 函数会对画布的每一行先调用 fseek 把偏移量定位到该行起点在屏幕中对应的位置，然后调用 fwrite 输出画布的一行，并调用 fflush() 刷新缓冲，最后调用 putc 向 fbsyncdev 输出 0 进行同步，并调用 fflush 刷新

缓冲。



