

华中科技大学

2022

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:	ACM1901 班
学 号:	U201915047
姓 名:	张骏烨
电 话:	15700893117
邮 件:	1832093998@qq.com
完成日期:	2022-10-20

目 录

1. 课程实验概述	1
1.1. 课设目的	1
1.2. 课设设计路径	1
1.2.1. PA0 - 世界诞生的前夜: 开发环境配置	1
1.2.2. PA1 - 开天辟地的篇章: 最简单的计算机	1
1.2.3. PA2 - 简单复杂的机器: 冯诺依曼计算机系统	1
1.2.4. PA3 - 穿越时空的旅程: 批处理系统	1
1.2.5. PA4 - 虚实交错的魔法: 分时多任务	2
2. 实验方案设计与结果分析	3
2.1. PA0 世界诞生的前夜:开发环境配置	3
2.1.1. 开发环境要求	3
2.1.2. 开发环境的安装和配置	3
2.1.3. 代码的备份和获取	3
2.2. PA1 开天辟地的篇章: 最简单的计算机	4
2.2.1. 在开始愉快的 PA 之旅之前	4
2.2.2. PA1.1 基础设施: 简易调试器	4
2.2.3. PA1.2 表达式求值 - 词法分析	5
2.2.4. PA1.2 表达式求值 - 递归求值	7
2.2.5. PA1.2 表达式求值 - 测试代码	8
2.2.6. PA1.3 监视点	8
2.2.7. 必答题	9
2.3. PA2 简单复杂的机器: 冯诺依曼计算机系统	11
2.3.1. PA2.1 运行第一个 C 程序	11
2.3.2. PA2.2 丰富指令集, 测试所有程序	13
2.3.3. PA2.3 运行第一个 C 程序	14
2.3.4. 必答题	16
2.4. PA3 穿越时空的旅程: 批处理系统	18
2.4.1. PA3.1 穿越时空的旅程	18
2.4.2. PA3.2 用户程序和系统调用	20
2.4.3. PA3.3 文件系统和批处理系统	21
2.4.4. 必答题	25
2.5. PA4 虚实交错的魔法: 分时多任务	27
2.5.1. PA4.1 多道程序	27
2.5.2. PA4.2 多道程序	28
3. 实验总结	30
参考文献	33

1. 课程实验概述

1.1. 课设目的

系统能力培养要求在代码框架中实现一个简化的 RISC-V 模拟器，同时需要满足以下要求。

1. 可解释执行 RISC-V 执行代码
2. 支持输入输出设备
3. 支持异常流处理
4. 支持精简操作系统---支持文件系统
5. 支持虚存管理
6. 支持进程分时调度

本课程需要最终在模拟器上运行“仙剑奇侠传”，探究“程序在计算机上运行”的机理，掌握计算机软硬件协同的机制，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，提升计算机系统能力。

1.2. 课设设计路径

1.2.1. PA0 - 世界诞生的前夜：开发环境配置

安装虚拟机，熟悉相关工具和平台。

1.2.2. PA1 - 开天辟地的篇章：最简单的计算机

本章节分为三个小章节，分别为 PA1.1 简易调试器，PA1.2 表达式求值和 PA1.3 监视点与断点。

1.2.3. PA2 - 简单复杂的机器：冯诺依曼计算机系统

本章节分为三个小章节，分别为 PA2.1 运行第一个 C 程序，PA2.2 丰富指令集，测试所有程序，PA2.3 实现 I/O 指令，测试打字游戏。

1.2.4. PA3 - 穿越时空的旅程：批处理系统

本章节分为三个小章节，分别为 PA3.1 实现系统调用，PA3.2 实现文件系统，PA3.3 运行仙剑奇侠传。

1.2.5. PA4 - 虚实交错的魔法: 分时多任务

本章节分为三个小章节，分别为 PA4.1 实现分页机制，PA4.2 实现进程上下文切换，PA4.3 时钟中断驱动的上下文切换。

2. 实验方案设计与结果分析

2.1. PA0 世界诞生的前夜:开发环境配置

2.1.1. 开发环境要求

本课程的开发环境基本要求如下：

CPU 架构：x64

操作系统：GNU/Linux

编译器：GCC

编程语言：C 语言

2.1.2. 开发环境的安装和配置

为了开发环境的多样性而造成的各种兼容性问题影响实验进度的问题,我们可以通过课程组老师提供的虚拟机镜像作为本次实验的开发环境。

进入虚拟机后，通过 clone 命令获取项目代码。然后进入目录“nemu/Makefile.git”中，按照要求修改 STUID 和 STUNAME。并在 master 分支中使用 add 和 commit 命令。

然后通过以下 git config 设置相关信息。

最后调用 make setup 创建远程仓库，通过 make password 命令更改密码。

2.1.3. 代码的备份和获取

在配置了相关信息后，我们调用命令

```
git push hustpa pa0:pa0
```

将代码 push 到远程库中，这样即使虚拟机平台崩溃也可以回滚上次备份的代码。

2.2. PA1 开天辟地的篇章: 最简单的计算机

2.2.1. 在开始愉快的 PA 之旅之前

课程组建议使用 RISC-V32 指令集架构开展本次课程设计。在目录 `nemu/src/isa` 下分别有三种指令集对应的代码。

观察整个项目的结构, PA 的框架代码由 4 个子项目构成: NEMU, Nexus-AM, Nanos-lite 和 Navy-apps, 本章节主要关注 NEMU。NEMU 主要由 4 个模块构成: monitor, CPU, memory, 设备。

观察 `ics2019/nemu/src/main.c` 代码, 最开始便调用函数 `init_monitor()` 对 monitor 进行一系列的初始化。

然后我们使用如下指令进行编译并运行。

```
make ISA=$ISA run
```

在调用函数 `init_monitor()` 后, 会执行 `ui_mainloop()` 函数。通过输入的参数, 可以选择执行对应的命令。在这里我们执行内置的指令 “c”。在命令提示符后键入 c 后, NEMU 开始进入指令执行的主循环 `cpu_exec()`, 这部分体现在函数 `cmd_c` 中。

在熟悉相关框架后, 我们开始完善 PA1 的内容。

2.2.2. PA1.1 基础设施: 简易调试器

我们需要在 monitor 中实现一个具有如下功能的简易调试器。根据给定的命令的格式和功能, 实现相应的命令。

在上一部分我们已经大致了解了命令的执行。在 `monitor/debug/ui.c` 文件中, 为 `main` 函数调用的 `ui_mainloop()` 函数。同时我们可以看到结构体 `cmd_table`。其存储了命令名称, 注释和相对应需要调用的函数指针。

在函数 `ui_mainloop` 中, 我们调用函数 `rl_gets()` 读取输入。并通过 `strtok()` 函数获得输入的第二个参数。如果输入合法, 我们可以根据该参数选择对应的函数指针并调用函数。

接下来我们需要完善剩余命令。

a. 单步执行

单步执行的格式为 “si [N]”, 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。其具体实现在 `cmd_si` 函数中。在解析参数后，选择单步执行的步数，并通过调用函数 `cpu_exec` 实现。

注意缺省时执行一步，但输入错误或者输入 `N` 为负数时，输出对应的错误信息。

b. 打印寄存器

分析参数。当执行 `info r` 时，就调用 `isa_reg_display()`，在里面直接通过 `printf()` 输出所有寄存器的值即可。

这里我们使用的 ISA 为 RISC-V32，因此我们完善 RISC-V32 框架下的 `isa_reg_display()`。给出的寄存器个数为 32 个，我们通过如下代码输出所有的寄存器信息。

```
for(int i=0;i<32;i++){
    printf("%s %x\n",regsl[i],cpu.gpr[i]._32);
}
```

寄存器名称保存在 `char` 型数组 `regsl` 中。而其值保存在 `cpu.gpr[i]._32` 中。

c. 扫描内存

该指令需要我们求出表达式 `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。表达式求值在下一章实现，因此本章节实现简易的扫描内存。

在这里我们简单的将参数 `EXPR` 视为一个正整数。在解析参数后，调用函数 `paddr_read` 实现内存的扫描并输出，这里我们以四个字节为一个单位。

2.2.3. PA1.2 表达式求值 - 词法分析

2.2.3.1. 词法分析规则扩充

我们首先扩充结构体 `token` 的数据结构。需要注意的是，对于运算符，其在表达式中都会有一个优先级。同时对于某些符号，例如“+”，“*”等在不同的位置其优先级也不同。因此我们扩充结构体 `token`，添加数据成员 `prior` 记录此 `token` 的优先级。其结构如下：

```
typedef struct token {
    int type;
    char str[256];
    int prior;
```

```
} Token;
```

同时我们需要扩充规则。除了给定的“+”和空格串的规则，我们还需要添加“-”，“*”，“/”，“（”，“）”，“==”，“!=”，“&&”，“||”，“!”，“>”，“>=”，“<”，“<=”这些运算符。

除了运算符，我们还需要添加正则规则用以识别十进制数字，十六进制数字和 riscv 的寄存器表达式。其正则表达式如下：

```
{"\\$(\\$0|ra|[sgt]p|t[0-6]|a[0-7]|s([0-9]|1[0-1]))", TK_REG}  
{"0[xX][0-9a-fA-F]+", TK_HEXNUM}  
{"[0-9]+", TK_DECNUM}
```

2.2.3.2.词法分析 token 构建

给出一个待求值表达式，我们首先要识别出其中的 token，进行这项工作的是 make_token()函数。该函数的整体框架已经给出，只需要我们添加对不同 token 类型的处理。

对于十进制数字，十六进制数字和 riscv 的寄存器表达式这三种类型，我们不仅要把优先级存入 tokens 数组中，同时应该把其作为一个字符串存入 tokens 数组中。

而对于其他运算符，我们可以省略存放 buf 的操作。

2.2.3.3.词法分析优先级设置

在 make_token()的过程中，还有一个比较重要的步骤是给每一个 token 添加相应的优先级。这里我们编写函数 set_prior 来实现优先级的获取。这里我们参照 C 语言的优先级。首先对于值类型，其优先级设置为最高的 0。其次对于“+”，“-”“*”这三个算符以外的其他运算符的优先级，在本次实验中其优先级是固定的，因此不需要判断，直接返回其在 C 语言优先级表中的优先级。

对于“+”，“-”，“*”运算符，其为单目运算符和为双目运算符的依据是其前一个字符的类型。当其前一个运算符类型满足以下条件时

```
pre_token_type==' ' || pre_token_type==TK_DECNUM || pre_token_type==  
TK_HEXNUM || pre_token_type==TK_REG
```

即右括号，十进制数字，十六进制数字和寄存器表达式时，其为双目运算符，

“*” 的优先级为 3，“+” “-” 的优先级为 4。

在其他情况下(包括运算符为表达式第一个字符)，其为单目运算符，他们三者的优先级均为 2。

2.2.4. PA1.2 表达式求值 - 递归求值

首先在讲义中已经给定递归求值的整体框架。

定义函数 `eval`，其参数为两个 `int` 类型的参数，来指示这个子表达式的开始位置和结束位置。

首先完善函数 `check_parentheses`。此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。主要关注该表达式是否以 “(” 开始，并且以 “)” 结束。

2.2.4.1. 主运算符的选取

定义函数 `main_operator` 用以寻找目前子表达式的主运算符。

1. 非运算符的 `token` 是不会作为主运算符的，因为这些 `token` 的优先级定义为 0，即可认为其不会被选做主运算符；
2. 出现在一对括号中的 `token` 不是主运算符，因为括号优先级最高，而主运算符为最后计算的运算符；
3. 主运算符的优先级在表达式中是最低的。
4. 结合性。当有多个运算符的优先级都是最低时，根据结合性，最后被结合的运算符才是主运算符。不同的优先级，其结合性是不同的。当我们选出当前表达式最低的优先级值后，对于从左到右结合的运算符，它的主运算符应该是最右边那个运算符，而对于从右到左结合的运算符，它的主运算符应该是最左边那个运算符。

我们根据上述四条规则可以选取当前主运算符的位置，并返回至 `eval` 函数中。

2.2.4.2. 非运算符 `token` 的值计算

定义函数 `cal_num`，根据十进制数字，十六进制数字和寄存器的表达式分别

调用不同的函数计算值。

2.2.4.3.递归求值

当选取主运算符后即可递归求值。可以观察到如果主运算符为当前子表达式的第一位，其必为单目运算符。递归求出其值类型并根据运算符类型返回最终结果；否则其为双目运算符，递归求出两边的值并根据运算符类型返回最终结果。

2.2.5. PA1.2 表达式求值 - 测试代码

完善函数 `gen_rand_expr`，即可生成包含加减乘除等四种运算的字符串表达式。然后完善 `code_format`，添加 `sighandler` 函数监视除以零的情况。最后通过 `make` 函数生成可执行文件，最终生成测试表达式。

修改原始代码，测试表达式求值模块。

2.2.6. PA1.3 监视点

扩充监视点的结构。如下所示：

```
typedef struct watchpoint {
    int NO;
    struct watchpoint *next;
    /* TODO: Add more members if necessary */
    char buf[1024];
    uint32_t value;
} WP;
```

添加 `buf` 存储监视表达式，`value` 存储当前该表达式的算数值。

完善监视池的插入和释放函数。因为原始的数据结构已经构建了两条链：`head` 和 `free_`，因此插入和释放监视点即将 `head` 和 `free_` 链表中的成员交换即可。

在实现监视点的插入和释放后，完善调试器。

“`w EXPR`”即监视 `EXPR`，这里我们插入一个监视点到 `head` 链表中，同时存入其 `buf` 值，并计算其 `value` 值。

“`d N`”为删除标号为 `N` 的监视点，通过监视点释放函数即可实现。

“`info w`”输出所有活动监视点的信息，这里遍历 `head` 链表并输出即可。

最后，实现函数 `watchpoint_trigger()`。其计算所有活动监视点当前值和存储值是否相同，不同则将 `nemu_state.state` 变量设置为 `NEMU_STOP` 来达到暂停的

效果。这里我们当 `cpu_exec` 执行完一条指令后，调用该函数。

2.2.7. 必答题

- a. 究竟要执行多久？在 `cmd_c()` 函数中，调用 `cpu_exec()` 的时候传入了参数 -1，你知道这是什么意思吗？

因为 `cpu_exec()` 的参数为 `unsigned long` 类型，-1 即为最大值。循环一个最大次数。

- b. `opcode_table` 到底是个什么类型的数组？

其为一个 `OpcodeEntry` 的数组。而 `OpcodeEntry` 类型在之中可以看到如下结构。同时 `DHelper` 会被定义为：

```
typedef struct {
    DHelper decode;
    EHelper execute;
    int width;
} OpcodeEntry;
```

`decode` 表示译码函数，`execute` 表示执行函数，`width` 为操作数长度。

- c. 我选择的 ISA 是 RISC-V32。
- d. 理解基础设施：假设这 500 次编译当中，有 90% 的次数是用于调试，那么有 450 次编译用以调试。30s 一个信息，一个 bug 需要 20 个信息才能排除一个 bug。那么花费时间为 $450 \times 30 \times 20 = 270000s$ 。即可以节约 $450 \times (30 - 10) \times 20 = 180000s = 50h$ 。
- e. riscv32 查阅手册

1. riscv32 有哪几种指令格式？

共有六种。

用于寄存器-寄存器操作的 R 类型指令，用于短立即数和访存 load 操作的 I 型指令，用于访存 store 操作的 S 型指令，用于条件跳转操作的 B 类型指令，用于长立即数的 U 型指令和用于无条件跳转的 J 型指令。

2. LUI 指令的行为是什么？

高位立即数加载，指令格式为 `LUI rd, immediate`。 $x[rd] = sext(immediate[31:12] \ll 12)$ 。

`mstatus` 寄存器的结构是怎么样的？

mstatus（Machine Status）它保存全局中断使能，以及许多其他的状态。结构如下所示：



图 2.1 手册关于 mstatus 内容

f. shell 命令

使用 find 命令统计 nemu 目录下的行数。首先切换至 pa0，使用 find 目录查看行数：

```
hust@hust-desktop:~/ics2019/nemu$ find . -name "*.c" -or -name "*.h" | xargs grep -Ev "^$" | wc -l
4008
```

图 2.2 pa0

```
hust@hust-desktop:~/ics2019/nemu$ find . -name "*.c" -or -name "*.h" | xargs grep -Ev "^$" | wc -l
4499
```

图 2.3 pa1

所以可以看到总共添加了 491 行代码。

g. -Wall 和 -Werror

-Wall 使 GCC 编译后显示所有的警告信息。-Werror 会将所有的警告当成错误进行处理。

2.3. PA2 简单复杂的机器：冯诺依曼计算机系统

2.3.1. PA2.1 运行第一个 C 程序

在 nexus-am/tests/cputest/目录下键入

```
make ARCH=riscv32-nemu ALL=dummy run
```

可以发现其提示非法指令，因此我们为了使得程序运行，需要完善这些指令。在 nexus-am/tests/cputest/build/dummy-riscv32-nemu.txt 中我们可以查看反汇编结果，添加在之中使用并未实现的指令。

读源代码分析指令实现的过程。我们首先查看调用可以锁定函数 `exec_once`。其同时其调用函数 `isa_exec` 和 `update_pc`。

而观察 `isa_exec` 可以发现其具体实现了几个功能：1.将当前的 PC 保存到全局译码信息 `decinfo` 的成员 `seq_pc` 中 2.取指令；3.调用 `index` 函数执行指令。

全局译码信息 `decinfo` 记录一些全局译码信息供后续使用，包括操作数的类型，宽度，值等信息。

在取指令后，执行指令前我们首先需要对取指令的结果进行译码。我们可以首先根据 `riscv` 的 `opcode` 字段区分不同的指令。这里我们使用 `opcode` 字段作为数组 `opcode_table` 的索引。根据 pa1 必答题可以知道数组 `opcode_table` 的意义。其主要包含了某指令对应的译码辅助函数 `DHelper` 和执行辅助函数 `EHelper`，同时还有操作数的位宽。

译码辅助函数 `DHelper` 和执行辅助函数 `EHelper` 均通过宏定义。接下来介绍两者的实现。

a. 译码辅助函数

首先是译码辅助函数 `DHelper`。对于不同的指令类型，例如 I 型指令，J 型指令等等，其操作数的获取是不相同的，因此我们需要根据不同的指令类型来实现其对应的译码函数。不同指令类型的译码辅助函数统一通过如下宏定义：

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

不同指令的译码辅助函数实现在 `nemu/src/isa/riscv32/decode.c` 中。但在 `riscv` 中，寄存器操作数，立即数操作数十分常见，因此我们定义操作数译码辅助函数

简便操作。

最后译码辅助函数的实现可以简化为通过调用操作数译码辅助函数，将不同指令的操作数信息写入 `decinfo` 中。例如对于寄存器操作数，将寄存器编号，寄存器存储值写入(可选)，对于立即数操作数，将指令中对应立即数字段取出拼接成立即数并存入。

b. 执行辅助函数

执行辅助函数对应的宏定义如下所示：

```
#define make_EHelper(name) void concat(exec_, name) (vaddr_t *pc)
```

对于不同指令其执行的结果是不同的，因此我们需要实现所需指令的执行辅助函数。所以执行辅助函数定义在文件 `nemu/src/isa/riscv32/exec/all-instr.h` 中。实现执行辅助函数我们主要是分清源操作数，目的操作数，执行过程即可。同时使用 RTL 指令简化实现过程。

c. funct3 和 funct7 字段

仅仅通过 `opcode` 字段区分指令很明显是不足够的。例如对于 B 型指令，其到底是哪一条指令还需要调用 `funct3` 字段区分。因此我们对于所有 `opcode` 为 `0x18` 的指令命名为“b”，同时实现其执行函数，使得我们可以进一步区分 `funct3` 字段，并进一步选择是调用“BEQ”或者“BNE”等指令。我们定义数组 `b_table` 保存所有“b”类型指令的指令体数组。

对于 I 型指令，load 型指令等等都可以使用这种方法区分 `funct3` 和 `funct7` 字段。

在实现以上定义后，我们可以顺利地调用指令了。`idex` 函数调用了 PC 取得指令的译码函数和执行函数。

在执行指令后，我们需要更新 PC。`decinfo` 成员 `is_jump` 保存执行指令后，是否跳转的信息，`jump_pc` 保存了跳转地址。这两个成员均在指令执行过程中更新。

在明确了上述信息，并完成上述设计后，我们可以实现第一个 c 语言程序 `dummy` 了。查看其反汇编文件可以看到我们需要完善的指令，例如 `addi`，`auipc` 等。在实现后，重新运行，当 HIT GOOD TRAP 出现时，即代表我们实现 `dummy` 中的指令成功了。

2.3.2. PA2.2 丰富指令集，测试所有程序

a. Differential Testing

首先实现 Differential Testing，降低发生错误时的调试成本。首先打开在 `nemu/include/common.h` 中定义宏的 `DIFF_TEST`。然后根据实验要求，实现函数 `isa_difftest_checkregs()`。其主要是对比通用寄存器和 PC 的值与从 QEMU 中读出的寄存器的值。若对比结果一致，函数返回 `true`，如果发现值不一样，函数返回 `false`。当返回 `false` 后程序会停止，避免一错再错，不断执行错误指令。

b. 丰富指令集

逐步分批实现测试程序。在运行后，会生成反汇编文件，我们定位到未实现的指令，将其加入整个项目中。当成功后会显示 `HIT GOOD TRAP`，反之会显示 `HIT BAD TRAP`。

对于指令的实现在 PA1 中已经设计了具体框架，因此我们只需要实现所需指令对于的部件，然后机械化地实现其他指令即可。

c. 实现常用的库函数

在一些测试程序中，调用了部分库函数，因此我们还需要实现这些库函数。

首先测试程序 `string.c`，其测试了文件 `nexus-am/libs/klib/src/string.c` 中我们实现的所有库函数。主要是阅读手册仿照实现即可。

最后为了运行测试用例 `hello-str`，还需要实现库函数 `sprintf`。其实现在文件 `nexus-am/libs/klib/src/stdio.c` 中。因为 `sprintf` 的参数可变，因此其实现起来与 `string.c` 文件中的其他函数不同。这里我们主要使用 `va_start` 和 `va_end` 这两个宏。参照源码我们将 `sprintf` 的实现转移到 `vsprintf` 中，并在 `sprintf` 中调用 `vsprintf`。在使用 `va_start` 和 `va_end` 这两个宏后，通过 `va_arg` 检索函数参数列表中类型为 `type` 的下一个参数。我们首先分析 `fmt` 串，当没有遇到 `%` 字符时，简单的将 `fmt` 中的字符复制到 `out` 中；当遇到 `%` 后，判断其后续字符，进而判断其是否为转换说明。根据转换说明的类型，通过 `va_arg` 检索，然后转换为字符类型输入到字符串中。

这里我们首先只实现 `%d` 和 `%s` 这两种，其他在后续补充。

d. 一键回归测试

根据提示测试所有测试文件，成功后如下图所示：

```
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 2.4 一键测试

2.3.3. PA2.3 运行第一个 C 程序

在成功运行了 `cputest` 中的各个测试用例后，我们需要拓展该项目的更多功能。在本章节，我们主要实现对各个设备的支持。NEMU 实现了串口，时钟，键盘，VGA 四种设备，我们需要完善相关的部分。

在 `nexus-am/tests/amtest/` 目录下键入以下命令，可以测试相应的程序。

```
make ARCH=native mainargs=H run
```

令 `mainargs=h`，运行 Hello World。这里不需要添加额外的代码。

在测试上述功能后，我们先实现 `printf` 函数以便后续使用。这里调用已经实现的 `vsprintf` 和 `_putc` 即可。

a. 时钟

实现 `__am_timer_init()` 函数初始化起始时间。最开始，读出通过 `nemu` 存入地址 `RTC_ADDR` 中的值，将其设置为起始时间。虽然这里未进行初始化。

然后在 `__am_timer_read` 函数中完善 AM 系统启动时间部分。与起始时间相减即可实现计时器的功能。

还需要注意的是，补充 `vsprintf` 中对精度和宽度的处理，以便处理 `rtc.c` 中的“%02d”。

运行结果如下图所示：

```
Welcome to riscv32-NEMU!
For help, type "help"
2000-0-0 00:00:00 GMT (1 second).
2000-0-0 00:00:00 GMT (2 seconds).
2000-0-0 00:00:00 GMT (3 seconds).
2000-0-0 00:00:00 GMT (4 seconds).
2000-0-0 00:00:00 GMT (5 seconds).
```

图 2.5 时钟运行

b. 性能测试

以 `microbench` 为例，运行结果如下所示：

```
[sieve] Eratosthenes sieve: * Passed.
min time: 5663 ms [695]
[l5pz] A* 15-puzzle search: * Passed.
min time: 1065 ms [421]
[dinic] Dinic's maxflow algorithm: * Passed.
min time: 1267 ms [858]
[lzip] Lzip compression: * Passed.
min time: 1261 ms [602]
[ssort] Suffix sort: * Passed.
min time: 576 ms [781]
[md5] MD5 digest: * Passed.
min time: 5835 ms [295]
=====
MicroBench PASS      635 Marks
                    vs. 100000 Marks (i7-7700K @ 4.20GHz)
Total time: 31663 ms
nemu: HIT GOOD TRAP at pc = 0x801041e0
```

图 2.6 性能测试

c. 键盘

通过 `inl` 读 `KBD_ADDR` 中的数据。判断 `0x8000` 的位置是否为 1，即可设置 `keydown` 信息。

运行结果如下所示：

```
Welcome to riscv32-NEMU!
For help, type "help"
Try to press any key...
Get key: 43 A down
Get key: 43 A up
Get key: 46 F down
Get key: 46 F up
Get key: 59 V down
Get key: 59 V up
```

图 2.7 键盘运行

d. VGA

借鉴 `native` 的实现。

`__am_video_read` 读取 `screensize` 的值并存入 `width` 和 `height`。

`__am_video_write` 通过 `_DEV_VIDEO_FBCTL_t` 结构体，向屏幕(x,y)坐标处绘制 `w*h` 的矩形图像。从 `_DEV_VIDEO_FBCTL_t` 结构体获取的数据，写入 `FB_ADDR` 中。

运行结果如下所示：

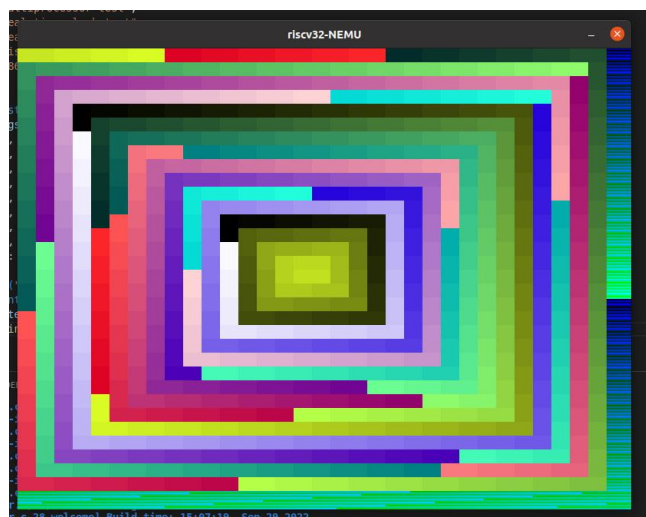


图 2.8 VGA 运行

2.3.4. 必答题

a. 一条指令在 NEMU 中的执行过程

`cpu_exec` 中调用 `exec_once` 执行一条指令。`exec_once` 会调用 `isa_exec` 执行一条指令并更新 `pc`。`isa_exec` 则会调用 `idex` 函数，执行该条指令的译码和执行函数。在 `isa_exec` 中会通过指令的相应字段，取 `opcode_table` 中的 `OpcodeEntry` 指令体，将其传入 `idex`，便于 `idex` 执行其译码和执行函数。在执行译码函数后，我们可以很容易取到该条指令的操作数，进而执行该条指令。

b. 在 `nemu/include/rtl/rtl.h` 中,尝试去除 RTL 指令函数指令的 `static` 或 `inline`，解释错误为什么会发生

去掉 `static`：无报错

去掉 `inline`：会提示大量如下警告

```
./include/rtl/rtl.h:133:13: warning: 'rtl_not' defined but not used [-Wunused-function]
133 | static void rtl_not(rtlreg_t *dest, const rtlreg_t* src1) {
    |             ^~~~~~
```

图 2.9 警告

去掉 `inline` 和 `static`：会出现如下错误

```
/usr/bin/ld: build/obj-riscv32/isa/riscv32/decode.o: in function `rtl_not':
/home/hust/ics2019/nemu/./include/rtl/rtl.h:133: multiple definition of `rtl_not'
```

图 2.10 报错

因为 `inline` 内联会在编译的时候直接在调用处展开该函数，而 `static` 可以保证其作为一个静态函数不能被其它文件调用，作用于仅限于本文件。当去掉 `inline` 和 `static` 后，每个模块中都会定义一次 `rtl_not`，因此会出现多重定义的错误。

- c. `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？然后在 `nemu/include/debug.h` 中添加一行查看变化；修改为 `volatile static int dummy=0;` 查看变化

在 `build/obj-riscv32` 中使用 `grep` 命令查看。

34 个。

```
hust@hust-desktop:~/ics2019/nemu/build/obj-riscv32$ grep -r -c 'dummy' ./* | grep '\.o:[1-9]' | wc -l
34
```

图 2.11 查看

当在 `debug.h` 中加入后，个数依然不变。`debug.h` 已经引用了 `common.h`，未初始化的为弱符号，对于变量的重复声明仅作用一次。然而当加上赋值后，其为强符号，同名的强符号只能有一个，否则编译器报“重复定义”错误。

- d. `make` 如何组织 `.c` 和 `.h`

`make` 会查找 `Makefile` 文件，执行对应的操作。

编译链接的过程：对指定的源文件进行预处理，并对于使用 `.c` 和 `.h` 文件编译、汇编，最后将编译产生的 `obj` 文件进行链接，生成最终的可执行文件。

2.4. PA3 穿越时空的旅程: 批处理系统

2.4.1. PA3.1 穿越时空的旅程

pa3.1 需要实现系统调用。需要我们一步步实现各个相关指令最终完成系统调用。

a. 设置异常入口地址

在定义宏 HAS_CTE 后, Nanos-lite 会在开始阶段执行函数 `init_irq`。通过这个函数最终会指向各个 ISA 的 `_cte_init` 函数。这个函数中, 对于 `riscv32`, 其通过添加汇编代码, 通过指令 `csrw` 初始化 `stvec` 寄存器。

我们需要实现上述功能, 首先就是添加相应的指令。这里我们添加 pa3.1 阶段所需的四条指令: `csrrw`, `csrrs`, `ecall`, `sret`。

我们添加 CSR 寄存器 `sepc`, `scause`, `sstatus`, `stvec`。同时添加函数 `read_CSR` 和 `write_CSR`, 根据 `csrrw` 和 `csrrs` 指令的 CSR 字段, 读取或写入相应的值到我们添加的 CSR 寄存器中。

接下来就是按照 pa2 中的做法实现指令 `csrrw` 和 `csrrs` 了。

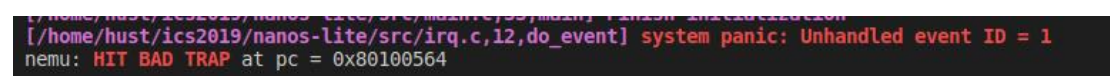
b. 触发自陷操作

实现 `raise_intr()` 函数。即根据传入的 NO 异常号, `epc` 异常 pc 地址分别初始化寄存器 `sepc`, `scause`。同时根据 `stvec` 寄存器的值设置异常跳转的地址, 并设置跳转标记, 使得顺利跳转至异常入口地址。

`ecall` 指令调用 `raise_intr` 函数即可。需要注意的是, 我们传入的 `pc` 值是发生异常时的 `pc` 地址, 在异常返回后决定是否+4。

同时我们还需要注意的是异常号是多少。在 `_yield` 插入汇编指令 `ecall` 前, 我们可以看到这样的一条指令: “`li a7, -1`”, 可以猜想其 `a7` 寄存器保存的是异常号。

当实现这个部分后, 输出如下所示:



```
[/home/hust/ics2019/nanos-lite/src/irq.c,12,do_event] system panic: Unhandled event ID = 1
nemtu: HIT BAD TRAP at pc = 0x80100564
```

图 2.12 触发自陷操作

c. 保存上下文

ecall 后进入异常入口地址，即进入函数__am_asm_trap 中。通过观察整个项目的反汇编代码，找到这个函数，可以看到其参数入栈的顺序。通过这个顺序我们可以按照要求重新组织_Context 结构体的成员。可以看到按照如下顺序：

gpr[32], cause, status, epc

在__am_irq_handle()中通过 printf 输出上下文 c 的内容，观察其参数是否顺利传入。

如下图所示：



图 2.13 保存上下文

重点关注 epc 的值，观察反汇编代码可以发现，即为 ecall 的地址。

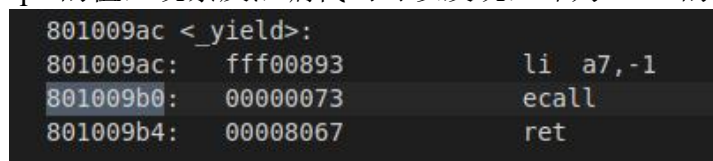


图 2.14 ecall

d. 事件分发

__am_irq_handle()中会根据异常号设置事件的类别。根据 a7 赋值为-1，我们这里认为-1 为 yield 异常的异常号。通过 switch case 即可识别 yield 异常。将此事件打包成编号为_EVENT_YIELD 的事件。do_event()事件根据事件编号处理相应的事件。

对于_EVENT_YIELD 事件，我们通过 printf 输出提示信息表示我们确实已经执行到此处。

最后是异常返回后的地址问题。我们保存在 epc 中的地址是异常发生的地址，对于此处的 yield 事件，我们希望做到的是 ecall 之后返回其下一条地址。因此我们需要在一个合适的地方令 epc+4。因为是根据事件的不同，异常返回的 PC 是否+4 也不同。所以我们在事件处理函数 do_event()中，根据事件的不同确定 epc 是否+4。yield 事件需要+4。

当是否+4 的问题解决后，epc 保存的即为真正返回的地址。通过软件我们将上下文的 epc 读取回 sepc 中，因此我们将 sepc 设置为 jmp_pc 即可。

最终执行如下所示：

```

[/home/hust/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/ics2019/nanos-lite/src/main.c,33,main] Finish initialization
event YIELD
[/home/hust/ics2019/nanos-lite/src/main.c,39,main] system panic: Should not reach here
nemu: HIT BAD TRAP at pc = 0x80100620

```

图 2.15 系统调用

提示 YIELD 事件后，我们通过 sret 顺利返回，并触发了 main()函数末尾设置的 panic()。

2.4.2. PA3.2 用户程序和系统调用

a. 加载第一个用户程序

此部分需要我们加载用户程序 dummy。从执行视图角度来看，ELF 中采用 program header table 来管理 segment。我们通过相应的属性和调用定义好的函数来加载程序。

这方面的实现在 loader 函数中。在此阶段我们只需要加载程序 dummy，首先忽略 loader 的参数。首先读取 elf 头的内容。通过函数 ramdisk_read 将 elf 头读取到结构体 Elf_Ehdr 中。然后建立程序表头数组，从 elf 头中读取程序表头的起始地址，大小和数目，然后读取程序表头，然后根据程序表头的 p_type 选择是否加载。

最后根据根据程序表头的内容，加载这个 segment 使用的内存，即 [VirtAddr,VirtAddr+MemSiz)，然后将[VirtAddr+FileSiz,VirtAddr+MemSiz)对应的物理区间清零。

b. 系统调用

本部分主要来完善系统调用。在 nanos.c 里，我们可以看到函数 _syscall_()。可以发现，riscv32 的 type 存放在 a7 寄存器中，a0, a1, a2 寄存器分别传入参数，而 a0 寄存器同时作为返回值返回。

我们在 __am_irq_handle() 中加入 case，将时间号为 0-19 的包装为系统调用 _EVENT_SYSCALL。在 do_event() 函数中，我们对于系统调用，调用函数 do_syscall() 系统调用处理函数。

在系统调用处理函数中，根据参数 GPR1(即 a7 寄存器)的值，我们可以确定系统调用的类型。yield 系统调用之间调用 _yield() 函数，并传入返回值到 a0 寄存器中；exit 系统调用直接调用 _halt() 函数。

最终应该会 hit good trap，如下图所示：

```
[/home/hust/ics2019/nanos-lite/src/irq.c,21,init_irq] Initializing interrupt/exception handler...  
[/home/hust/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...  
[/home/hust/ics2019/nanos-lite/src/loader.c,31,naive_oload] Jump to entry = 0x830000C8  
event YIELD  
nemu: HIT GOOD TRAP at pc = 0x80100874
```

图 2.16 运行结果

c. 标准输出

在这里我们需要实现 write 函数。我们直接参照上方系统调用的实现，实现 SYS_write 系统调用。

注意这里我们只需要实现标准输出，因此我们判断 fd 的值，当其为 1 或者 2 时，直接通过 _putc() 函数输出。

d. 堆区管理

首先实现函数 _sbrk()。我们首先初始化 program break 的位置，即指向 _end。当调用函数 _sbrk() 时，我们首先记录当前的 program break 的值，然后通过系统调用传入新的 program break 的值 (increment + old program break)，记录 program break 的位置。同时，在函数 _sbrk() 中，我们也更新 program break 的值并返回 old program break 的值。

当实现堆区管理后，运行 hello，可以发现 printf() 的信息通过一次 write 调用输出，而不是像之前标准输出中一个一个字符输出。具体如下图所示：

```
call sys_write! Hello World from Navy-apps for the 1255th time!  
call sys_write! Hello World from Navy-apps for the 1256th time!  
call sys_write! Hello World from Navy-apps for the 1257th time!  
call sys_write! Hello World from Navy-apps for the 1258th time!  
call sys_write! Hello World from Navy-apps for the 1259th time!  
call sys_write! Hello World from Navy-apps for the 1260th time!  
call sys_write! Hello World from Navy-apps for the 1261th time!  
call sys_write! Hello World from Navy-apps for the 1262th time!
```

图 2.17 堆区管理

2.4.3. PA3.3 文件系统和批处理系统

a. 让 loader 使用文件

首先实现文件相关的系统调用。同时实现文件处理的相关函数 fs_open() 和 fs_read() 和 fs_close() 和 fs_write() 和 fs_lseek()。

首先是系统调用，write, open, close, read, lseek 的系统调用类似上述实现。

fs_open() 函数，通过输入的 pathname，比对 file_table 中每个文件的 name，

最终返回找到的文件的下标。

`fs_read()`在这里通过 `ramdisk_read` 函数，将相对应地址的内容读入到 `buf` 中。

`fs_write()`在这里通过 `ramdisk_write` 函数，将 `buf` 写入到对应地址的内容中。

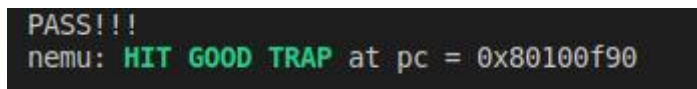
同时 `read` 和 `write` 需要通过 `fs_lseek()`函数调整其 `open_offset`。

`fs_lseek()`根据传入的参数 `whence` 实现对应的调整 `open_offset` 的功能。

`fs_close()`将 `open_offset` 置零。

最后是修改 `loader` 函数，让其能够使用文件。首先通过给出的 `filename`，调用 `fs_open()`获得文件的下标。然后我们就可以通过 `fs_read()`读取到该文件的 `elf` 头。剩下的步骤即通过 `read`，`write` 和 `lseek` 读取程序头表，并选择加载其中的部分。

运行测试程序 `/bin/text`，结果如下图所示：



```
PASS!!!
nemu: HIT GOOD TRAP at pc = 0x80100f90
```

图 2.18 loader 使用文件

b. 把串口抽象成文件

`stdout` 和 `stderr` 都通过串口输出。将其 `write` 函数设置为 `serial_write`。这样，我们本来在 `fs_write()`函数中通过判断 `fd` 的值为 1, 2 来调用 `_putc` 函数就可以封装到 `filename[fd].write` 函数中了。这里我们认定当 `filename[fd].write` 为空时，其为普通文件，我们通过 `ramdisk_read` 写入。否则我们调用其对应的 `filename[fd].write` 函数写入。

`serial_write` 函数直接通过 `_putc` 输出 `buf`。

c. 把设备输入抽象成文件

这里我们主要实现输出时钟和键盘。我们将这两个事件上述事件抽象成文件 `/dev/events`，为其设置 `read` 函数 `events_read`。

借助 IOE 的 `api`，`read_key` 我们可以获得当前键盘信息。通过判断其是否为空，按键状态为按下或者松开，我们可以输出相对应的信息到 `buf` 中，这里我们调用 `sprintf`，并根据给出的输出格式输出到 `buf` 中。

注意当键盘状态为 `_KEY_NONE` 时，我们输出时间。

接下来我们加载 `/bin/events`，其运行结果如下所示：


```

receive time event for the 50176th time: t 2173
receive time event for the 51200th time: t 2216
receive time event for the 52224th time: t 2238
receive event: kd S
receive time event for the 53248th time: t 2310
receive time event for the 54272th time: t 2343
receive time event for the 55296th time: t 2365
receive time event for the 56320th time: t 2433
receive event: ku S
receive time event for the 57344th time: t 2465

```

图 2.19 把设备输入抽象成文件

d. 把 VGA 显存抽象成文件

首先 VFS 中添加对 `/dev/fb`, `/dev/fbsync` 和 `/proc/dispinfo` 这三个特殊文件的支持,

然后在 `init_fs()` 函数中, 首先对 `fb` 的大小进行初始化。这里我们首先通过 IOE, 获取屏幕宽和高, 将 `/dev/fb` 的大小设置为 `screen_width() * screen_height() * sizeof(uint32_t)`。

这里实现函数 `fb_write()`。首先我们通过 `offset` 计算出对应的 `x` 和 `y` 坐标。然后调用 IOE 的接口 `draw_rect` 实现将 `buf` 中的字节写入到屏幕上。`ndl.c` 中一次写入 `canvas_w` 个到 `/dev/fb` 中, 一次绘制一行, 这里简化 `draw_rect` 的 `h` 设置为 1, `w` 设置为 `len/sizeof(uint32_t)`。

`fbsync_write()` 直接调用 `draw_sync()`。

`init_device()` 中, 我们通过 `screen_width()` 和 `screen_height()` 函数, 直接获取宽和高信息, 将宽和高通过给出的格式写入到字符串 `dispinfo` 中。

`dispinfo_read()` 直接把字符串 `dispinfo` 中 `offset` 开始的 `len` 字节写到 `buf` 中。

最终加载 `/bin/bmptest`, 其运行结果如下图所示:

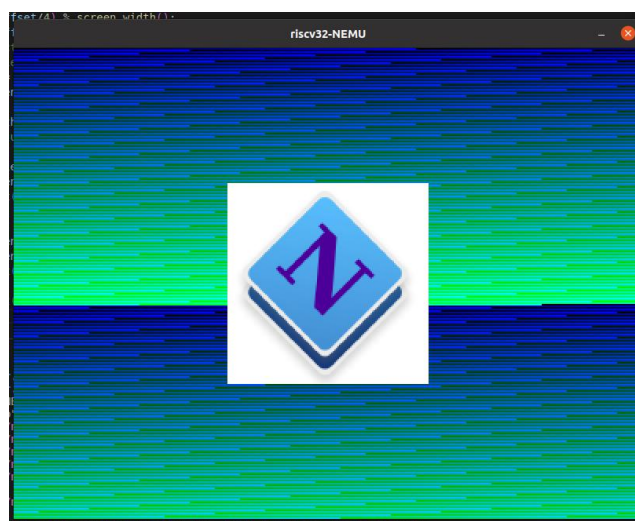


图 2.20 VGA

e. 运行仙剑奇侠传

通过老师给出的资源,更新 ramdisk 之后, 在 Nanos-lite 中加载并运行/bin/pal, 仙剑奇侠传运行结果如下图所示:

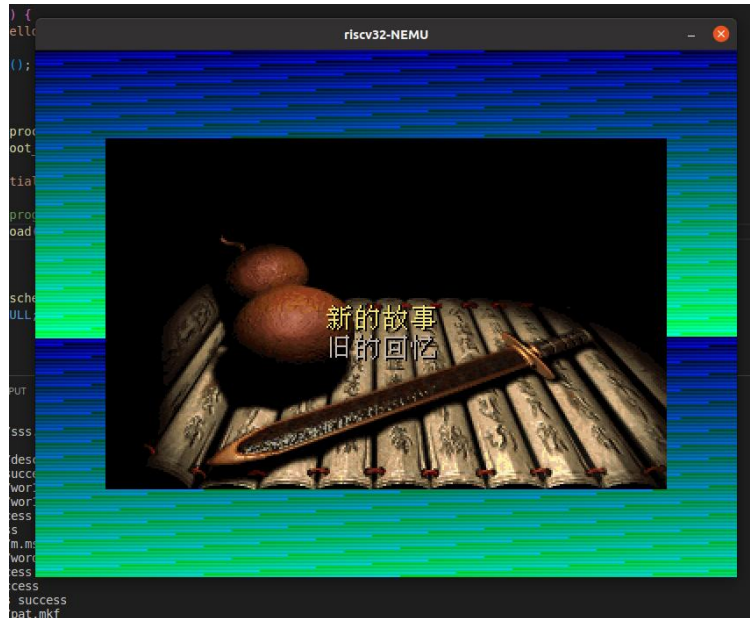


图 2.21 仙剑奇侠传

f. 自由开关 DiffTest 模式

首先我们修改 make 文件使得我们可以在客户程序开始运行前输入命令。

然后我们在 pal 实现的调试器中加入两条新指令 detach 和 attach。

detach 命令中,我们退出 DiffTest 模式。即让 difftest_step(), difftest_skip_dut() 和 difftest_skip_ref()直接返回。因为定义了变量 is_detach, 设置为 true 时, 上述三个函数直接返回, 所以我们指令 detach 直接调用函数 difftest_detach(), 这个函数将 is_detach 设置为 true。

attach 命令调用函数 difftest_attach(), 这个函数已经实现并调用函数 isa_difftest_attach()。实现函数 isa_difftest_attach(), 主要设置 REF 的内存区间为 DUT 的物理内存内容, 同时将 DUT 的寄存器状态同步到 REF 中。分别调用函数 ref_difftest_memcpy_from_dut 和 ref_difftest_setregs 即可。

我们运行一个测试程序, 使用 detach 后执行指令, 然后打开 attach 查看后续执行是否出错, 结果如下图:

```
Connect to QEMU successfully
[src/monitor/monitor.c,25,welcome] Debug: OFF
[src/monitor/monitor.c,28,welcome] Build time: 16:34:12, Oct 14 2022
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) detach
(nemu) si 200
(nemu) attach
(nemu) c
nemu: HIT GOOD TRAP at pc = 0x80100254
```

图 2.22 自由开关的 DiffTest

g. 批处理系统

添加特殊文件/dev/tty 并通过串口输出，并将其写函数设置为 serial_write()。
实现系统调用 SYS_execve，通过传入的文件名调用 naive_uload()。
最后修改 exit 系统调用，通过调用 naive_uload()重新运行/bin/init。
最终运行结果如下图所示：

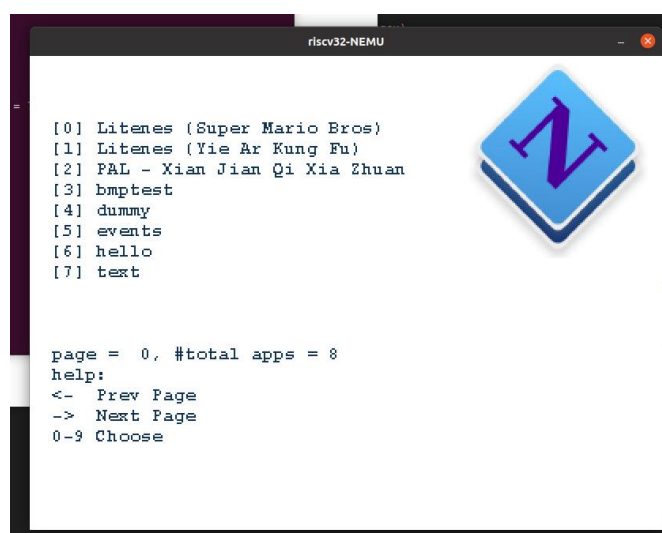


图 2.23 批处理系统

2.4.4. 必答题

a. 上下文结构体

上下文结构体是 __am_irq_handle 函数的参数。他的前身是通过 trap.S 中压栈的数据。我们在上面的实验中通过调整其结构将其各个成员对齐。在其传入函数 __am_irq_handle 后，我们通过其中的各个成员判断中断号，并调用相应的处理函数，实现跳转。

b. 时空穿越的旅程

这里以调用 `_yield()` 作为例子。调用 `_yield()` 后，其执行 `ecall` 汇编指令。`ecall` 指令调用 `raise_intr`，然后顺利跳转到异常处理地址。即进入函数 `__am_asm_trap`。这个函数在保存上下文后，调用函数 `__am_irq_handle`。该函数通过传入的 `cause` 值设置其上下文的事件号。然后执行设置好的 `user_handler`，这里为函数 `do_event`。YIELD 事件直接 `printf` 相关信息即可。其他系统调用相关的事件直接调用系统调用处理函数即可。

c. `hello` 程序是什么，它从而何来，要到哪里去

通过观察 `hello` 程序的源文件，可以看到其测试的是 `write` 系统调用的功能。成功 `make` 后其存放到 `ramdisk.img` 文件中。我们通过调用相应的接口将其从磁盘读出，分析 `elf` 头，程序头等加载该程序，然后执行该程序。

d. 仙鹤的像素信息怎样更新到屏幕上的

首先 `PAL_InitGlobals()` 函数通过调用函数 `UTIL_OpenRequiredFile()` 打开并读取文件 `mgo.mkf` 并将其赋值给文件指针 `gpGlobals->f.fpMGO`。而 `UTIL_OpenRequiredFile()` 是通过调用 `fopen` 函数打开该文件。这个过程中会调用到文件读写相关的系统调用。然后通过 `PAL_SplashScreen()` 函数播放的。具体仙剑奇侠传相关的代码也没有读很明白。

2.5. PA4 虚实交错的魔法: 分时多任务

2.5.1. PA4.1 多道程序

a. 实现上下文切换

首先实现创建内核上下文函数 `_kcontext()`。直接在 `stack` 底部创建一个 `_Context` 上下文, 并将其 `epc` 设置为 `entry` 体的地址, 返回这个上下文即可。

实现进程调度函数 `schedule()`。这里已经给出实现方法。其总是会切换至第一个用户进程 `pcb[0]`。

然后修改 `_EVENT_YIELD` 事件, 调用 `schedule()` 并返回新的上下文。

最后实现恢复上下文。当我们进入 `__am_irq_handle` 后, 会调用 `user_handler` 获得 `next` 上下文。这里 `user_handler` 即 `do_event` 函数。其遇到 `_EVENT_YIELD` 事件后, 会返回新的上下文, 因此这个新的上下文 `next` 最终会返回到 `__am_irq_handle` 作为返回值返回到 `__am_asm_trap`。我们将栈指针指向这个上下文, 这样 `__am_asm_trap` 就实现了恢复上下文的操作。因为 `riscv32` 返回值存放在 `a0` 寄存器中, 我们通过 `mv` 指令将 `a0` 寄存器赋给 `sp`。

最后修改 `init_proc()`, 创建内核上下文。

为了便于截图, 修改 `hello_fun` 的循环次数, 我们在加载 `dummy` 后, 运行结果如下图

```
page = 0, MAX_PAGE = 0, MAX_IDX_LAST_PAGE = 7
Available applications:
[0] Litenes (Super Mario Bros)
[1] Litenes (Yie Ar Kung Fu)
[2] PAL - Xian Jian Qi Xia Zhuan
[3] bmpstest
[4] dummy
[5] events
[6] hello
[7] text
page = 0, #total apps = 8
help:
<- Prev Page
-> Next Page
0-9 Choose

=====
Please Choose.
/bin/dummy
[/home/hust/ics2019/nanos-lite/src/loader.c,50,naive_uoload] Jump to entry = 0x830000C8
[/home/hust/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 1th time!
[/home/hust/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 2th time!
[/home/hust/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 3th time!
[/home/hust/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4th time!
```

图 2.24 上下文切换

b. 实现多道程序系统

这里我们创建用户进程上下文。通过 `context_uoload` 加载用户进程上下文。这

里通过 loader 获得 entry 入口地址。然后通过_ucontext 设置上下文。同时在 ustack 底部写入上下文。上下文将 epc 的值设置为 entry。入口地址 main 函数的参数在 riscv32 中通过寄存器存入。暂时这里我们先不去管这个部分。

在实现上述后，我们修改 schedule(), 轮流内核上下文和用户上下文的内容。

在 serial_write(), events_read()和 fb_write()的开头调用_yield(), 来模拟设备访问缓慢的情况。

最后运行结果如下图所示，一边输出 hello 一边运行仙剑奇侠传：

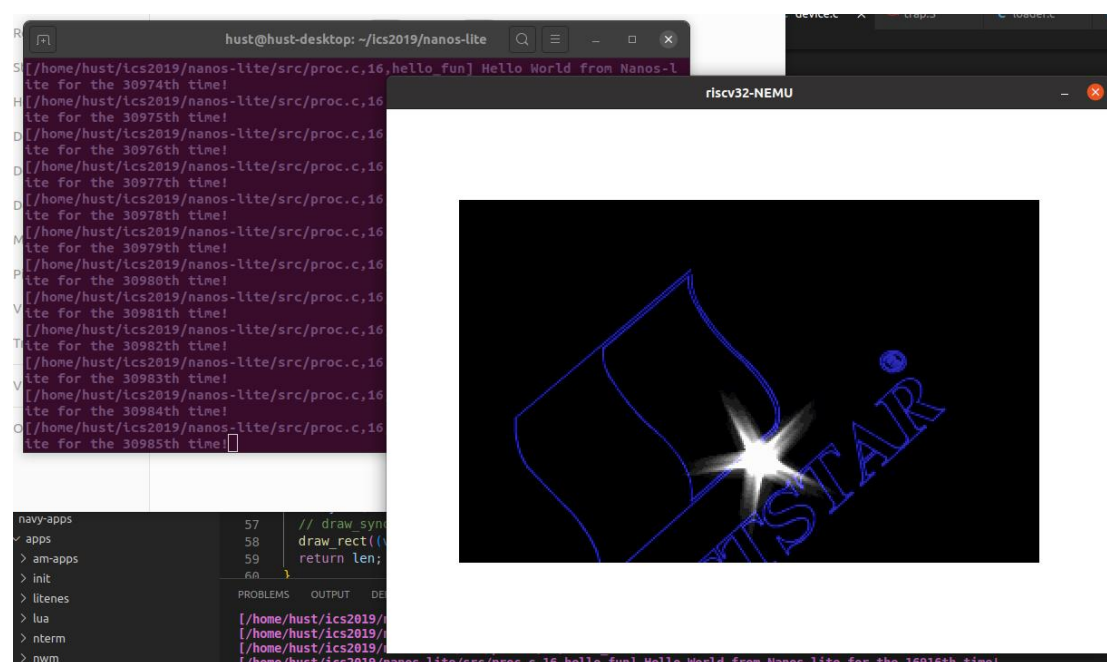


图 2.25 多道程序系统

2.5.2. PA4.2 多道程序

这个部分还有问题没改出来。

首先添加寄存器 satp, 晚上 csrr 和 csrw。然后实现的是 isa_vaddr_read(), isa_vaddr_write()。当 satp 最高位 valid, 就通过 page_translate 函数转换地址, 否则还是通过原本的方法。

page_translate 首先通过 satp 和 vaddr 获得 pde。然后获得 pte, 最后计算获得物理地址。

然后实现_map。同样我们直接通过 as 获得 pde 地址。然后找到 pte。查看其是否有效。当无效时 new 一个新页并返回, 最后写入 pde 中。然后我们在 pte 中写入转换的物理地址。

然后修改 loader 函数。通过 new_page 函数分配物理页然后调用_map 映射。
最后按照提示修改 __am_irq_handle 等函数。
最后我们运行 dummy:

```
/bin/dummy  
[/home/hust/ics2019/nanos-lite/src/proc.c,30,init_proc] Initializing processes...  
[/home/hust/ics2019/nanos-lite/src/main.c,33,main] Finish initialization  
nemu: HIT GOOD TRAP at pc = 0x80101230
```

图 2.26 dummy

可以运行成功。

然而我们在修改 mm_brk 后，还是会莫名其妙的遍历到一个不 valid 的 pde 或者 pte。然后会触发 assert。最后 bug 没有解决，不能成功运行除 dummy 以外的测试集。

3. 实验总结

经过这段时间的编写，最后进度停在 pa4.2 的中间。截至的这两天没有找到导致访问不合法 pte 或者 pde 的元凶。还是很遗憾的，希望以后有机会能够实现完。

从 pa1 只是实现一个简易的调试器，到 pa2 开始实现指令，到 pa3 开始实现到最后的批处理系统，最后还有没完成的 pa4，在整个实验的过程中学到了许多，对整个框架认知也一步步清晰。

首先是最开始的 pa1。确实是最开始的时候不知道怎么开始 pa1。但好在 pa1 的简易调试器已经给出了几条指令的实现。之后顺藤摸瓜便实现了 pa1.1 的那么几条指令。然后是 pa1.2，这里需要我们实现表达式求值。这里递归求值等步骤跟上学期的编译原理实验很相似，因此实验起来还是很轻松。然后按照实验要求需要我们实现负号等这些符号，很显然这些符号的结合性和+-这种双目运算符是不同的。但是开始还是按照实验开始的说法取最右符号作为主运算符，导致了一些问题。虽然最后还是很顺利解决。最后是 pa1.3，这里的监视点只需要添加一些结构体然后保存表达式，最后每次计算表达式就可以达到监视的目的。

在完成 pa1 之后，渐渐开始入门这个项目。但是 pa2 的整个过程还是比 pa1 要复杂很多的。首先我们需要实现未实现的指令，这个部分整个函数的调动还是很繁杂的，因此花费了许多的时间理解。但是在认真读懂整个程序后，完成函数很轻松很模式化的。之后 pa2.2 需要我们实现更多的指令，因为我是一个文件一个文件测试，因此没有遇到很多的问题。最后是 pa2.3，老实讲还是有些麻烦，但好在有已经实现的 native 作为参考，所以还是没浪费很多时间。

然后就顺利进入 pa3 的编写。pa3.1 需要我们实现 yield，确实是最开始对调用 _yield() 开始的整个过程认识很混乱。但是慢慢捋顺之后发现其实还是很简单的。整个调用的过程还是很清晰的。然后 pa3.2 老实讲按照讲义还是很清楚的。但是像 loader 函数，还是遇到了一些问题。因为之前系统基础等课程的内容有些记忆模糊，ELF 头等结构还是等回顾曾经学习过的课程才拾回来。最后的 pa3.3 文件系统和批处理系统，其实就是将 pa3.2 未完成系统实现的部分重新封装，而

不是像 pa3.2 那样简单的处理。还有批处理系统，按照讲义的步骤其实没什么需要纠结的部分。

最后就是 pa4 的部分了。pa4.1 的部分还是十分简单的，分别创建内核和用户进程。但是 pa4.2 还是时间不太够没时间完成到最后。首先弄明白了 riscv 的分页机制，了解页表，satp 寄存器等部分的结构。页面转换等部分依托这几个结构还是很好实现的。而_map 和 loader 是将用户进程按页加载，映射。流程还是很清楚的。但是细节上还是没弄很清楚，最后运行时访问到了不存在的页，最后指令也会取错，或者直接通过因为 pte，pde 不 valid 而 assert。没有最后能够加载多个应用进程这个功能也还是很遗憾的。

最后，通过这个实验，还是认识到自己的许多不足的。很多时候面对讲义的提示，还是会难以理解框架代码，还需要花费大量的时间去阅读和理解整个模块，之后才能顺利实现。还有就是面对 bug，定位和修改的能力有限，这也导致最后有问题没有修改出来，希望通过以后的学习提升自己的能力。

一、个人签名

作者签字：张骏烨 

参考文献

- [1] RISC-V 手册
- [2] 谭志虎, 秦磊华, 吴非, 肖亮. 计算机组成原理. 北京: 人民邮电出版社, 2021 年.