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：	CS2109 班
学　　号：	U202114963
姓　　名：	胡伟江
电　　话：	
邮　　件：	1036575651@qq.com
完成日期：	2025-1-5

计算机科学与技术学院

目 录

1　课程实验概述

1.1 课设目的

课设的内容是基于 RISCV32 架构，从“零”开始实现一个完整的计算机系统。课设将指导学生实现一个经过简化但功能完备的 RISCV32 模拟器 NEMU，并最终在 NEMU 上运行游戏“仙剑奇侠传”，来让学生探究“程序在计算机上运行”的基本原理。PA 包括一个准备实验（配置实验环境）以及 5 部分连贯的实验内容：

（1） 图灵机与简易调试器
（2） 冯诺依曼计算机系统
（3） 批处理系统
（4） 分时多任务
（5） 程序性能优化

1.2 课设任务

课设的具体任务如下：

（1） PA0 - 世界诞生的前夜：开发环境配置
　　（a） 安装 GNU/Linux
　　（b） 熟悉 PA 中可能用到的工具
　　（c） 获取 PA 的框架代码
（2） PA1 - 开天辟地的篇章：最简单的计算机
　　（a） 理解框架代码
　　（b） 实现一个简易调试器
　　（c） 实现表达式求值
　　（d） 实现监视点
（3） PA2 - 简单复杂的机器：冯诺依曼计算机系统
　　（a） 实现基本的指令，运行第一个 C 程序
　　（b） 实现更多的指令，运行更多的测试用例
　　（c） 使 NEMU 具备输入输出
（4） PA3 - 穿越时空的旅程：批处理系统
　　（a） 实现自陷操作和事件分发，并完成上下文的恢复
　　（b） 实现用户程序的加载和系统调用
　　（c） 实现文件系统，运行仙剑奇侠传
（5） PA4 - 虚实交错的魔法：分时多任务
　　（a） 实现上下文切换和多道程序系统
　　（b） 实现支持虚存管理的多道程序
　　（c） 实现抢占多任务
（6） PA5 - 天下武功唯快不破：程序与性能

1.3 实验环境

（1） CPU 架构：x64
（2） 操作系统：GNU/Linux
（3） 编译器：GCC
（4） 编程语言：C 语言

2　实验过程与结果

2.1 PA1 实验内容

2.1.1 PA1.1

首先我需要实现单步执行指令 si [N]，其过程非常简单，只需要把参数转化为数字，然后调用 cpu_exec 函数执行指令即可，最终的效果如图 2.1 所示。

```
(nemu) si 1
80100000:   b7 02 00 80                                  lui  0x80000,t0
```

图 2.1 单步执行指令

接着我实现打印寄存器状态指令 info r，我可以直接在指令中调用 isa_reg_display 函数，然后在这个函数内部输出所有寄存器的名称和值，最终的效果如图 2.2 所示。

```
(nemu) info r
$0  ->  0x00000000  0
ra  ->  0x00000000  0
sp  ->  0x00000000  0
gp  ->  0x00000000  0
tp  ->  0x00000000  0
t0  ->  0x80000000  2147483648
t1  ->  0x00000000  0
t2  ->  0x00000000  0
s0  ->  0x00000000  0
s1  ->  0x00000000  0
a0  ->  0x00000000  0
```

图 2.2 打印寄存器状态

然后我实现扫描内存指令 x N EXPR，由于此时尚未实现表达式求值，所以 EXPR 参数仅支持最简单的十六进制数，然后我调用 isa_vaddr_read 函数读取内存的值即可，其效果如图 2.3 所示。

```
(nemu) x 1 0x80100000
[src/monitor/debug/expr.c,90,make_token] match rules[7] = "0[xX][0-9A-Fa-f]+" at p
osition 0 with len 10: 0x80100000
0x800002b7
```

图 2.3 扫描内存

2.1.2 PA1.2

在 PA1.2 中，我要完成表达式求值。首先我需要为算术表达式中的各种 token 类型添加规则，即实现各种运算符号和十进制数字的正则表达式匹配，在成功识别出 token 之后，将 token 信息依次记录到 tokens 数组中。

之后我需要实现一个 check_parentheses 函数用来判断一个表达式是否被括号包围，因为当一个表达式被括号包围时，我可以直接去掉这个括号从而简化后续的操作。为了实现这一判断，我可以用一个变量记录尚未被匹配的左括号数量，

当一个表达式被括号包围时，这个变量的值在扫描到最后一个字符之前必定不会为零，因此我只需要检查这一点即可。

然后我实现了可以递归求解表达式的 eval 函数，首先我需要找到表达式中的主运算符，然后递归调用 eval 函数，分别求解主运算符两边的表达式，最后再根据主运算符对两边的值进行计算即可。

最后为了验证我代码的正确性，我考虑使用随机测试的方式。我补全了表达式生成器的框架代码中的 gen_rand_expr 函数，然后就可以通过这个函数随机生成表达式来验证 eval 函数的正确性。

在实现 eval 函数之后，表达式求值指令 p EXPR 就很容易实现了，同时，之前我实现的扫描内存指令 x N EXPR 也得以拓展，最终其效果如图 2.4 所示。

```
(nemu) p 1+1
[src/monitor/debug/expr.c,90,make_token] match rules[8] = "[0-9]+" at position 0
 with len 1: 1
[src/monitor/debug/expr.c,90,make_token] match rules[1] = "\+" at position 1 wit
h len 1: +
[src/monitor/debug/expr.c,90,make_token] match rules[8] = "[0-9]+" at position 2
 with len 1: 1
0x00000002
```

图 2.4 表达式求值

2.1.3 PA1.3

在 PA1.3 中，我拓展了表达式求值的功能，不难发现在之前实现的表达式求值中，表达式都只能是常数，这样的表达式在监视点中没有任何意义，因此接下来我将实现更多类型的 token，包括指针解引用、负号、寄存器求值、等于和不等于等运算符。

而在这些新拓展的 token 中，值得我需要关注的是指针解引用和负号这两个 token，因为我不能简单地通过字符串匹配来识别这两个 token（它们会分别和乘号、减号混淆）。因此，我实现了一个 check_unary_opt 函数，用来判断一个 token 是否是单目运算符，其实现过程也非常简单，我只需要判断其上一个 token 是否是某一系列 token 即可。然后对于某个乘号 token，如果其是单目运算符，则将其类型修改为指针解引用即可，对于负号也同理。

同时寄存器求值也需要我另外补全 isa_reg_str2val 函数，在这个函数中，我需要根据给出的寄存器名称参数，返回将该寄存器的值。最后，我就可以得到功能更加丰富的 expr 函数，其最终的效果如图 2.5 所示。

```
(nemu) p $pc
[src/monitor/debug/expr.c,90,make_token] match rules[9] = "\$[0-9a-z]+" at posit
ion 0 with len 3: $pc
0x80100000
```

图 2.5 拓展表达式求值

之后我就可以进一步地实现监视点求值了，首先我需要定义监视点的结构体，在代码框架已经给出的成员变量的基础上，由于我需要记录监视点变化的时刻，因此我还需要新增一个 last_val 变量来记录上一次该监视点的值。然后显然地，为了知道监视点所监视的表达式，我还需要新增一个字符数组变量 expr。

接着我便可以实现 new_wp 和 free_wp 函数，即新增监视点和删除监视点的函数。在代码框架中，监视点池通过链表的结构相连，因此新增监视点时只需要

把链表头的空闲监视点拿出来，将其放入正在使用的监视点链表的链表头即可。而对于删除监视点，我只需要找到对应编号的监视点，将其从正在使用的监视点链表中取走，放入空闲监视点池的链表头即可。有了这两个辅助函数，便可以很容易实现设置监视点指令 w EXPR 和删除监视点指令 d N 了。

但此时工作还未结束，因为我还需要在程序执行的过程中，判断监视点是否发生变化并及时暂停程序的执行。因此我又实现了另一个辅助函数 check_watchpoint，在这个函数中，我将会对于每个监视点，计算其表达式的值并将其与上一次的值进行比对，如果发生变化则输出对应信息，并返回 true 值；否则返回 false 值。这样在 cpu_exec 函数中，我在每一条程序指令执行之后，都调用一次 check_watchpoint 函数进行判断，并在监视点变化之时将模拟器的状态设置为 NEMU_STOP 即可。

然后，为了方便查看所有监视点，我又实现了打印监视点信息指令 info w，其过程与打印寄存器状态指令类似，我首先实现了 watchpoint_display 函数，并在这个函数内输出所有监视点的信息，然后直接调用该函数即可。

最后，监视点的效果如图 2.6、图 2.7 和图 2.8 所示。

```
(nemu) w $pc==0x8010000c
[src/monitor/debug/expr.c,90,make_token] match rules[9] = "\$[0-9a-z]+"
 at position 0 with len 3: $pc
[src/monitor/debug/expr.c,90,make_token] match rules[10] = "==" at posi
tion 3 with len 2: ==
[src/monitor/debug/expr.c,90,make_token] match rules[7] = "0[xX][0-9A-F
a-f]+" at position 5 with len 10: 0x8010000c
```

图 2.6 新增监视点

```
Watchpoint 0:
EXPR: $pc==0x8010000c
Old value: 0
New value: 1
```

图 2.7 监视点变化

```
(nemu) info w

Watchpoint 0:
EXPR: $pc==0x8010000c
Value: 1
```

图 2.8 打印监视点信息

2.2 PA2 实验内容

2.2.1 PA2.1

在 PA2.1 中，我就要开始实现 RISCV32 架构中的指令，在此之前，我需要先尝试理解代码框架中的各种辅助函数，比如 make_DHelper 和 make_EHelper 等，利用这些辅助函数可以很方便地帮助我们完成后续各种指令实现的繁琐操作。

实际上，RISCV32 中的指令还可以再继续分解成一些更简单的操作的组合，

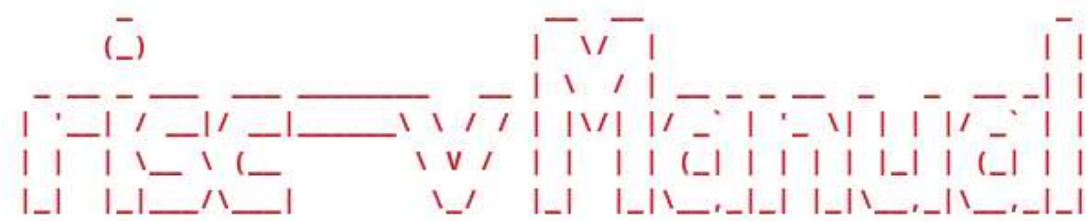
用这样的方法可以进一步地提高代码的利用率。在 NEMU 中，使用 RTL（寄存器传输语言）来描述这些简单的操作，而 RTL 指令又分为 RTL 基本指令和 RTL 伪指令。在代码框架中，RTL 基本指令都已经实现完成了，而 RTL 伪指令大部分都尚未实现，因此我首先需要补全这些未实现的 RTL 伪指令。

需要我实现的 RTL 伪指令一共有四个，即 rtl_not（非）、rtl_sext（符号位拓展）、rtl_msb（取符号位）和 rtl_mux（条件赋值）。在 rtl_not 中，只需要把源寄存器的值异或上一个立即数 0xffffffff 然后送往目的寄存器即可；在 rtl_sext 中，只需要将源寄存器的值左移对应的位数，然后再算术右移回来即可；在 rtl_msb 中，只需要将源寄存器右移直到原本的符号位在第一位，然后与上 1 即可；在 rtl_mux 中，根据条件成立与否将对应的源寄存器的值送到目的寄存器即可。

之后便是实现 RISCV32 架构的指令直到 dummy.c 程序可以正确运行，为了实现在这个程序中缺失的指令，我会直接运行该程序，根据出错的 pc 值在其对应的汇编代码中找到该指令，然后根据 RISCV 的手册实现这条指令的功能。

以我实现的第一条指令为例，首先我运行该程序后得到如图 2.9 所示的报错信息。

```
If it is the first case, see
```



```
for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!

nemu: ABORT at pc = 0x80100000
```

图 2.9 报错信息

然后我根据 pc 值 0x80100000 找到此时报错的指令是 li 指令，在 RISCV 手册中我了解到 li 指令会被拓展为 addi 指令。而 addi 指令属于 I 类指令，其指令译码函数尚未实现，因此我首先实现了 I 类指令的译码函数，其需要读取一个源寄存器、一个 12 位的有符号数和一个目的寄存器，然后就可以实现 addi 的执行函数，把源寄存器的值和该立即数相加之后送到目的寄存器即可。最后在 opcode_table 操作表中把 addi 指令填入对应的位置，这样 addi 指令就实现完成了。而其余的指令的实现过程与上述类似，无非就是比较繁琐的工程操作，因此在这里便不再赘述，最后 dummy.c 程序成功运行的结果如图 2.10 所示。

```
Welcome to riscv32-NEMU!
For help, type "help"
nemu: HIT GOOD TRAP at pc = 0x80100030

[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 13
 dummy
```

图 2.10 成功运行

2.2.2 PA2.2

单单有这么一点指令是不够的，在 cputest 中准备了一些简单的测试用例，于是接下来我需要做的便是实现更多的指令以通过这些测试用例。同样的，这部分的实现过程和之前为了让 dummy.c 程序运行起来一样，所以我也不再赘述了。

但是这些测试用例中的 string.c 程序和 hello-str.c 程序有些不同，因为为了让它们能够运行，我需要实现一些常用的库函数，具体地讲，即实现一些字符串处理函数和 sprintf 函数。

在实现字符串处理函数这一部分中，我一共实现了 strlen 函数、strcpy 函数、strncpy 函数、strcat 函数、strcmp 函数、strncmp 函数、memset 函数、memcpy 函数和 memcmp 函数，对于这里的每一个函数，我都是先通过查阅资料了解该函数的具体参数含义和返回值，然后便是通过简单地模拟实现其功能。

为了实现 sprintf 函数，由于其参数数目是可变的，为了获得数目可变的参数，我学习了 C 库 stdarg.h 的用法，然后实现了%d 和%s 的功能。在实现%d 的过程中，我补充了 i2s 函数，其功能是把整数转化成字符串并写入一个目的字符指针参数中；而在实现%s 的过程中，直接调用 strcpy 函数即可。

最后，在实现了足够多的指令之后，便可以通过 cputest 的所有测试用例了。代码框架提供了一键回归测试的功能，最终通过一键回归测试的运行结果如图 2.11 所示。

```
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!
```

图 2.11 一键回归测试

2.2.3 PA2.3

NEMU 实现了串口、时钟、键盘和 VGA 四种设备，接下来我就将完善这四种设备。

串口是最简单的输出设备，因为我选择的是 RISCV32 架构，所以我不需要额外实现代码，因为 NEMU 的映射机制已经支持 MMIO 了，因此此时已经可以通过 amtest 中 hello 测试程序了，最终其运行结果如图 2.12 所示。

```
Welcome to riscv32-NEMU!
For help, type "help"
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
nemu: HIT GOOD TRAP at pc = 0x80100e60
```

图 2.12 串口

此时 NEMU 已经具备了输出的功能，所以我可以在 klib 中实现 printf 函数了，该函数的功能和 sprintf 函数非常相似，因此两者的实现过程也大同小异，具体来说，在 printf 函数中，我只需要直接调用_putc 函数将字符输出即可。

在此之前 NEMU 还没有时钟功能，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，而我要做的则是补充__am_timer_read 函数中的_DEVREG_TIMER_UPTIME 抽象寄存器。首先我在__am_timer_init 函数中读取 NEMU 启动时的时间戳，具体地，直接从 RTC_ADDR 地址处读取 4 个字节的数据，然后记录下这个时间戳。之后每次从同一个地址处读取 4 个字节的数据，并与之前记录的数据做差便可以得到启动时间。然后便可以通过 amtest 中的 real-time clock test 测试程序了，其运行结果如图 2.13 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).
2000-0-0 00:00:00 GMT (6 seconds).
```

图 2.13 时钟

有了时钟之后，我就可以测试一个程序跑多快，从而测试计算机的性能。我让 NEMU 依次运行了 dhrystone、coremark 和 microbench 三个 benchmark，其跑分结果如图 2.14、图 2.15 和图 2.16 所示。

```
Welcome to riscv32-NEMU!
For help, type "help"
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 3143 ms
------------------------------------------------------
Dhrystone PASS          280 Marks
                    vs. 100000 Marks (i7-7700K @ 4.20GHz)
nemu: HIT GOOD TRAP at pc = 0x80100da4
```

图 2.14 dhrystone 跑分

```
Welcome to riscv32-NEMU!
For help, type "help"
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size    : 666
Total time (ms)  : 4200
Iterations       : 1000
Compiler version : GCC8.2.0
seedcrc          : 0x[0]crclist       : 0x[0]crcmatrix     : 0x[0]crcstate
: 0x[0]crcfinal      : 0xFinised in 4200 ms.
-------------------------------------------------
CoreMark PASS       695 Marks
                vs. 100000 Marks (i7-7700K @ 4.20GHz)
nemu: HIT GOOD TRAP at pc = 0x80102298
```

图 2.15 coremark 跑分

```
  ======================================================
  MicroBench PASS         898 Marks
                      vs. 100000 Marks (i7-7700K @ 4.20GHz)
  Total time: 25199 ms
  nemu: HIT GOOD TRAP at pc = 0x801041e0
```

图 2.16 microbench 跑分

键盘是最基本的输入设备，我补充了__am_input_read 函数中的_DEVREG_INPUT_KBD 抽象寄存器，每次从 KBD_ADDR 地址处读取 4 个字节的数据，其则为通码。然后我们判断其是否为_KEY_NONE 就可以知道是否获取了按键；而通过与 KEYDOWN_MASK 进行与操作就可以知道按键是被按下还是松开。然后便可以通过 amtest 中的 readkey test 测试程序了，其运行结果如图 2.17 所示。

```
Welcome to riscv32-NEMU!
For help, type "help"
Try to press any key...
Get key: 44 S down
Get key: 43 A down
Get key: 45 D down
Get key: 43 A up
Get key: 46 F down
Get key: 44 S up
Get key: 45 D up
Get key: 46 F up
Get key: 51 L down
Get key: 50 K down
Get key: 44 S down
```

图 2.17 键盘

VGA 可以用于显示颜色像素，是最常用的输出设备。首先在__am_video_read 函数中，我补全了_DEVREG_VIDEO_INFO 抽象寄存器，即从 SCREEN_ADDR 地址处读取 4 个字节的数据，其记录了屏幕的长度和宽度，将这两个信息返回即可。而在__am_video_write 函数中，我完善了_DEVREG_VIDEO_FBCTL 抽象寄存器，此时需要区分是否是同步屏幕的信号，如果需要同步屏幕，则直接向 SYNC_ADDR 地址处写入数据即可，同时在 vga_io_handler 函数中，如果 offset 参数为 4 且 is_write 参数为 true，则调用 update_screen 函数进行屏幕同步；如果需要绘制矩形图像，则将该矩阵图像的每一个像素点写入 FB_ADDR 地址处的相应偏移位置即可。然后便可以通过 amtest 中的 display test 测试程序了，其运行结果如图 2.18 所示。

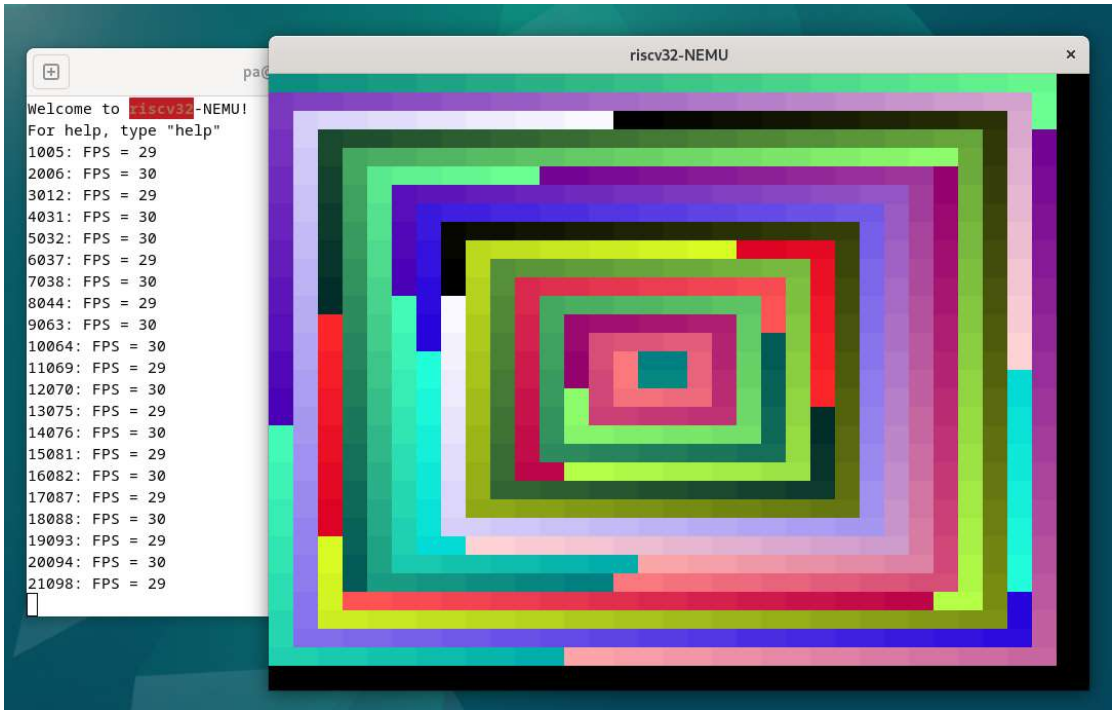


图 2.18 VGA

在完整实现 IOE 之后，我就可以运行一些炫酷的程序了，例如幻灯片播放程序和打字小游戏程序，其运行结果如图 2.19 和图 2.20 所示。

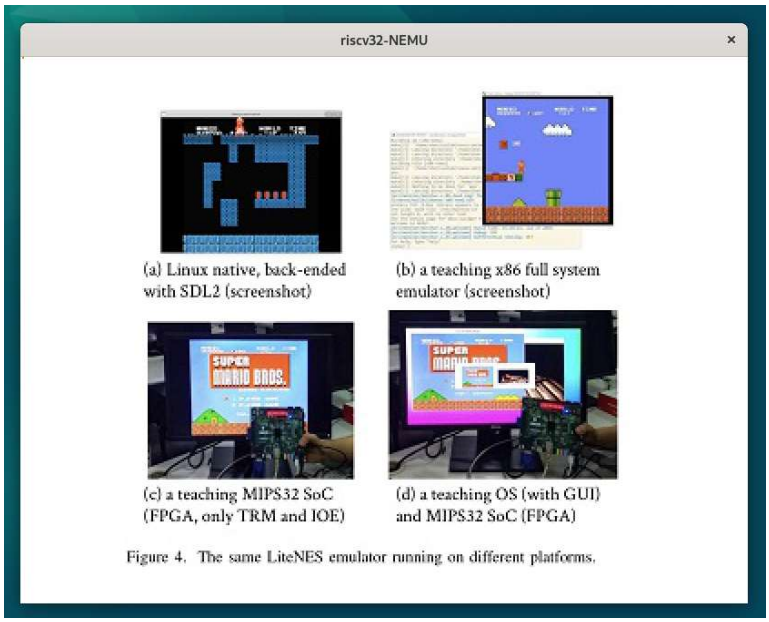


图 2.19 幻灯片播放

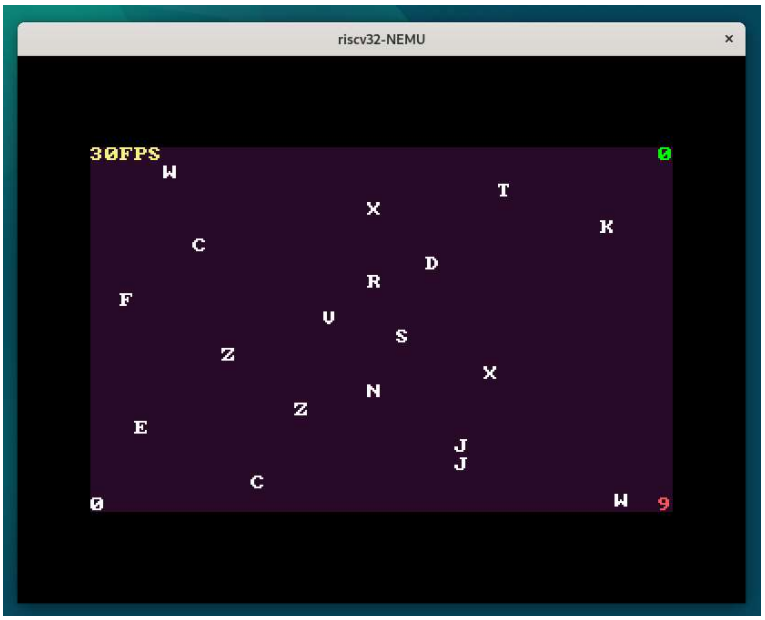


图 2.20 打字小游戏

2.3 PA3 实验内容

2.3.1 PA3.1

RISCV32 架构提供 ecall 指令作为自陷指令，并提供了一个 stvec 寄存器来存放异常入口地址。为了保存程序当前状态，RISCV32 提供了一些特殊的系统寄存器，叫控制状态寄存器（CSR 寄存器）。在 PA 中，我需要 3 个 CSR 寄存器：sepc 寄存器存放触发异常的 PC、sstatus 寄存器存放处理器的状态、scause 寄存器存放触发异常的原因。

因此我首先需要定义 CSR 寄存器，首先我根据汇编代码获取 stvec、sepc、

sstatus 和 scause 这四个寄存器的 CSR 编码，然后我实现了辅助函数 get_csr 来通过 CSR 编码获取其对应的 CSR 寄存器的地址；类似的还有 get_csr_name 函数，其可以获取对应的 CSR 寄存器的名字。有了这两个函数，我就可以实现 RTL 指令 rtl_csr_lr 和 rtl_csr_sr，其作用是对 CSR 寄存器进行读取或写入。

然后我实现了 CSR 相关的指令和 ecall 指令，ecall 指令用 a7 寄存器的值和当前的 PC 值作为参数调用 raise_intr 函数，而在 raise_intr 函数中，我模拟了触发异常后的硬件响应过程：首先将当前的 PC 值保存到 sepc 寄存器，然后在 scause 寄存器中设置异常号，接着从 stvec 寄存器中取出异常入口地址并调转到异常入口地址。

之后我通过阅读__am_asm_trap 函数的代码，调整了_Context 结构体的成员，使得这些成员与__am_asm_trap 函数中构造的上下文保持一致。

接着在__am_irq_handle 函数中，我通过异常号-1 识别出自陷异常，并打包成编号为_EVENT_YIELD 的自陷事件，紧跟着在 do_event 函数中识别出自陷事件_EVENT_YIELD。

最后我实现了 sret 指令，该指令的内容很简单，需要跳转到自陷指令的下一条指令，即 sepc 加 4。至此，_yield 函数可以成功运行。

2.3.2 PA3.2

有了自陷指令，用户程序就可以将执行流切换到操作系统指定的入口了，而接下来我将尝试加载第一个用户程序。

首先我先完善了 loader 函数，在这个函数中我先读取了 ELF header，其中的 e_phoff 是 program header table 的偏移地址，而 e_phnum 则是 segment 的数量。通过这两个信息我就可以遍历每一个 program header table 的每一个表项，而每个表项都描述了一个 segment 的所有属性，包括类型、虚拟地址、标志、对其方式、文件内偏移量和 segment 大小等。我可以通过 p_type 是否是 PT_LOAD 来判断一个 segment 是否需要加载，而通过 p_filesz 和 p_memsz 可以知道需要加载的具体大小。

然后在 do_event 函数中我实现了_EVENT_SYSCALL 系统调用事件的识别，并调用 do_syscall 函数进行处理。而在 do_syscall 函数中，我添加了 SYS_yield 系统调用，其将会直接调用_yield 函数，同时设置返回值为 0；同时我还添加了 SYS_exit 系统调用，其会接受一个退出状态的参数，并调用_halt 函数。之后便可以成功运行 dummy 程序，其运行结果如图 2.21 所示。

```
yield
nemu: HIT GOOD TRAP at pc = 0x80100f30
```

图 2.21 dummy 程序

之后我又实现了 SYS_write 和 SYS_brk 系统调用，其实现过程与上述其他系统调用的过程类似，额外地，我还实现了_sbrk 函数以实现堆区管理，其过程就是维护 program break 的值，只需要简单地模拟一下即可。此时就可以成功运行 hello 程序了，其运行结果如图 2.22 所示。

```
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!
Hello World from Navy-apps for the 7th time!
Hello World from Navy-apps for the 8th time!
```

图 2.22 hello 程序

2.3.3 PA3.3

要实现一个完整的批处理系统，我们还需要向系统提供多个程序，为此，我还要实现一个简易的文件系统。

首先我为这个文件系统实现了 5 个文件操作函数：fs_open 函数、fs_read 函数、fs_write 函数、fs_lseek 函数和 fs_close 函数，即对应了打开文件、读文件、写文件、修改偏移量和关闭文件这些操作。在 fs_open 函数中，只需要遍历一遍所有文件进行文件名匹配，然后返回相应的文件下标即可，同时注意将文件的偏移量置为 0；在 fs_read 函数中，调用 ramdisk_read 函数进行相应内存的读操作即可；在 fs_write 函数中，调用 ramdisk_write 函数进行相应内存的写操作即可；在 fs_lseek 函数中，根据 SEEK_SET、SEEK_CUR 和 SEEK_END 的含义计算相应的偏移量并赋值即可；在 fs_close 函数中，由于这个简易文件系统没有维护文件打开的状态，因此直接返回 0 即可。

之后我就可以修改 loader 函数，将其中涉及到内存修改的操作用文件操作函数代替，这样以后更换用户程序只需要修改传入 naive_uload 函数的文件名即可。同时我已经可以通过 text 测试程序，其运行结果如图 2.23 所示。

```
[/home/pa/ics2019/nanos-lite/src/
68
PASS!!!
```

图 2.23 text 程序

有了这个文件系统，我还可以把 IOE 抽象成文件。首先我实现了 serial_write 函数，然后对于 stdout 和 stderr 两个文件，令其调用这个 API 进行写操作，这样就使得 VFS 支持串口的写入了。

输入设备有键盘和时钟，我们可以通过文本地形式将这两个事件表现出来。因此我实现了 events_read 函数，它可以读取键盘或时钟并将其转化成对应的字符串。然后我将上述事件抽象成/dev/events 文件，并令其调用 events_read 函数作为读操作。这样加载/bin/events 程序后，就可以看到键盘或时钟事件的信息了，其运行结果如图 2.24 所示。

```
receive time event for the 10240th time: t 621
receive time event for the 11264th time: t 681
receive event: kd A
receive event: kd S
receive time event for the 12288th time: t 743
receive event: kd D
receive time event for the 13312th time: t 805
receive event: kd F
receive time event for the 14336th time: t 864
receive event: ku A
receive event: ku S
receive event: ku D
```

图 2.24 events 程序

最后是 VGA，我把显存抽象成/dev/fb 文件，支持写操作和修改偏移量的操作，以便于用户程序把像素更新到屏幕的指定位置上；然后把屏幕刷新操作抽象成/dev/fbsync 文件，其只支持写操作，每次写入都会同步屏幕；而屏幕大小的信息抽象成/proc/dispinfo 文件。

为了完成这一过程，首先我实现了 fb_write 函数，其通过调用 draw_rect 函数来绘制像素点，而由于 draw_rect 是以矩形的方式进行绘制，因此我的做法是根据行将需要绘制的像素点分为若干个矩形，再依次调用 draw_rect 函数即可。而对于 fbsync_write 函数，直接调用 draw_sync 函数即可。在 dispinfo_read 函数中，我事先将屏幕信息写入 dispinfo 字符串中，然后直接调用 strncpy 函数进行字符串复制即可。

完成上述操作后，加载/bin/bmptest 程序，将成功看到 Project-N 的 logo，其运行结果如图 2.25 所示。

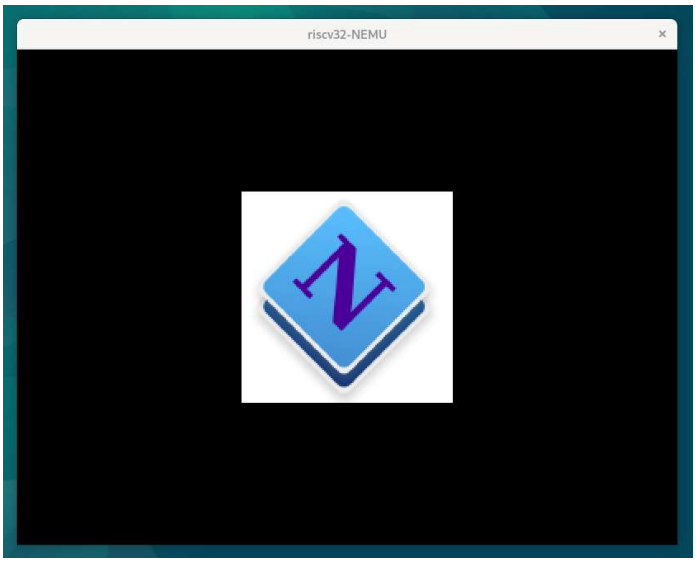


图 2.25 bmptest 程序

与此同时，我的系统也可以成功运行仙剑奇侠传了，其运行过程如图 2.26 所示。

图 2.26 仙剑奇侠传

在 Navy-apps 准备了一个开机菜单程序，我们加载/bin/init 程序就可以运行它，而为了运行这个程序，我需要在 VFS 中添加一个特殊文件/dev/tty，其只需要支持往串口写即可；除此之外，我还需要添加 SYS_execve 系统调用，其调用 naive_uload 函数即可，最后程序的运行结果如图 2.27 所示。

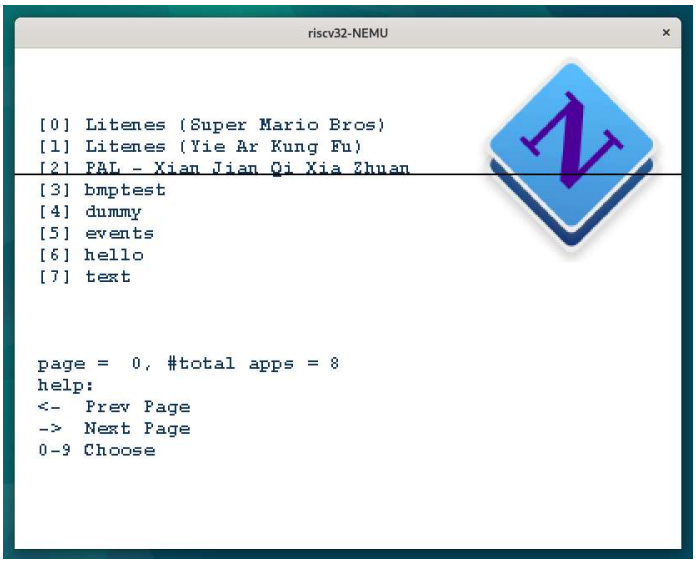


图 2.27 init 程序

2.4 PA4 实验内容

2.4.1 PA4.1

在 PA4 中我要实现多道程序系统，而为了实现操作系统和用户进程之间的执行流切换，我首先要完成上下文的切换。这一操作可以通过__am_asm_trap 函数来完成：在恢复上下文之前，跳转到另一个进程所保存的上下文结构，这样恢复之后就来到另一个进程了。

在此之前，我需要人工创建上下文结构，因此我补充了_kcontext 函数，在这个函数中，我 stack 的底部创建了一个以 entry 为返回地址的上下文结构（即把上

下文结构中的 epc 赋值为 entry），然后返回这一结构的指针。

接下来我需要实现进程调度，进程调度是由 schedule 函数来完成的，它用于返回将要调度的进程上下文。我用 current 指针记录当前正在运行的进程，这样就可以通过 current 来决定接下来要调度哪一个进程了。然后在 Nanos-lite 收到 _EVENT_YIELD 事件后，就可以调用 schedule 函数并返回新的上下文。

然后不要忘了__am_asm_trap 函数也需要改动，在其从__am_irq_handle 函数返回之后，该函数的返回值即为下一个需要跳转的进程的上下文结构的地址，这个返回值在 a0 寄存器中，因此我需要把 a0 的值送给 sp 寄存器，这样接下来就可以恢复对应进程的上下文了。

完成上述操作之后，就可以在 init_proc 函数中调用 context_kload 函数，创建一个以测试函数 hello_fun 为返回地址的上下文

不过目前为止，我只完成了为一个进程创建上下文，然而为用户进程创建上下文则需要一些额外的考量。我们知道，main 函数是有参数的，不过对于 RISCV32 架构来说，参数传递是通过寄存器来实现的，目前我只需要把这些寄存器设置为 0 即可。然后与_kcontext 函数类似地，我又补充了_ucontext 函数，这样我就可以加载用户进程了。最后我可以一边运行仙剑奇侠传，一边输出 hello 信息，其运行效果如图 2.28 所示。

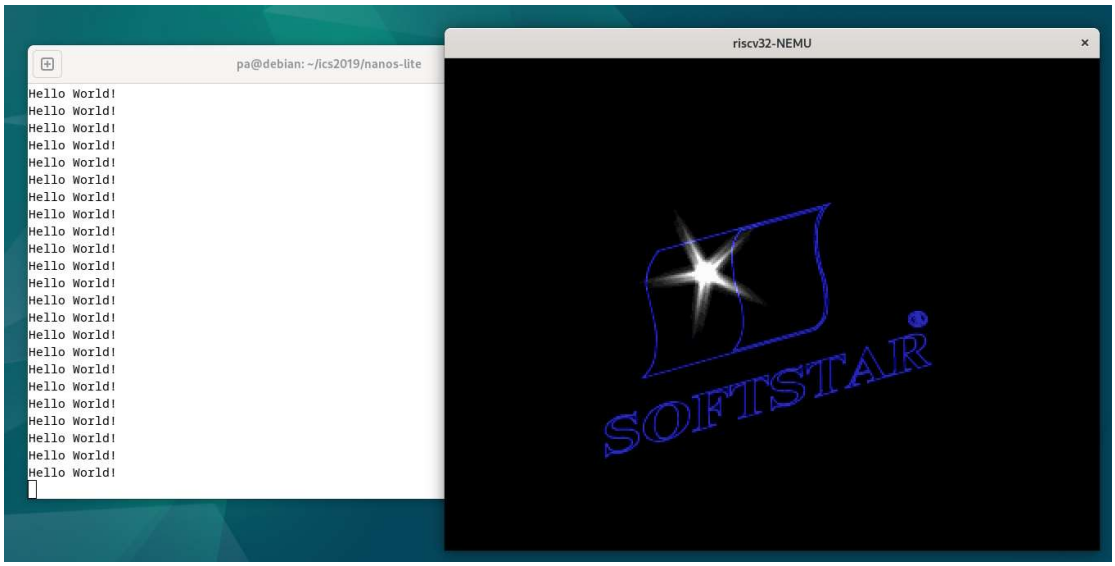


图 2.28 多道程序系统

2.4.2 PA4.2

接下来我将实现分页机制，首先我定义了新的 CSR 寄存器 satp，该寄存器记录了一级页表的入口地址，这样之后只需要调取该寄存器就可以找到页表。

然后我修改了 isa_vaddr_read 函数和 isa_vaddr_write 函数。在这一过程中我另外实现了辅助函数 page_translate，该函数首先会根据 satp 寄存器的值判断分页机制是否开启：若未开启，则直接返回该地址；若开启，则根据分页机制的定义将地址分为两个页表的索引以及偏移量，最后根据这个信息进行两次访存得到最终的实地址。

而为了在分页机制上运行用户程序，首先我需要修改 loader 函数，因为先前的 loader 函数是直接把数据复制到实地址上，但是在有了分页机制之后，我需要以页作为单位进行数据复制。除此之外，我还需要实现_map 函数，因为我需要

通过_map 函数把物理页映射到用户进程的虚拟地址之中。完成这些之后，我就可以成功在分页机制上运行用户进程了，其运行 dummy 程序的结果如图 2.29 所示。

```
[/home/pa/ics2019/nanos-lite/src/irq.c,18,init
tion handler...
[/home/pa/ics2019/nanos-lite/src/proc.c,30,ini
[/home/pa/ics2019/nanos-lite/src/main.c,33,mai
nemu: HIT GOOD TRAP at pc = 0x80101248
```

图 2.29 分页机制上运行 dummy 程序

为了运行仙剑奇侠传，我需要完善 mm_brk 函数，使得把新申请的堆区映射到虚拟地址空间中，这样才能保证运行在分页机制上的用户程序可以正确地访问新申请的堆区。在实现过程中，每当新的 brk 值超过之前记录的 max_brk 值时，则以页为单位，为虚拟地址分配物理页。在实现完成之后，便可以成功运行仙剑奇侠传了，与此同时还可以再运行一个 hello 用户程序，即实现了支持虚存管理的多道程序。

2.4.3 PA4.3

虽然我实现了进程的并发执行，但是目前的并发不一定是公平的，如果一个进程长时间不触发I/O操作，多道程序系统就不会主动将控制权切换到其他进程，因此我还需要实现基于硬件中断的抢占多任务。

首先我需要在 CPU 结构体中添加一个 bool 成员 INTR，在 dev_raise_intr 函数中将 INTR 设置为高电平，然后在 exec_once 函数的末尾轮询 INTR，调用 isa_query_intr 函数查询是否有硬件中断的到来。

而在 isa_query_intr 函数中，我需要根据 INTR 是否为高电平以及 sstatus.SIE 是否为 1（即是否开中断）来判断中断是否到来，若中断到来，则需要把 INTR 设置为低电平，同时调用 raise_intr 函数。之后我还要修改 raise_intr 函数，让处理器进入关中断状态：把 sstatus.SIE 保存到 sstatus.SPIE 中，然后把 sstatus.SIE 设置为 0。同时在 sret 指令中，我需要把 sstatus.SPIE 还原到 sstatus.SIE 中，然后把 sstatus.SPIE 设置为 1。

最后，我需要把时钟中断打包成_EVENT_IRQ_TIMER 事件，在收到该事件后，调用_yield 函数强制当前进程让出 CPU，然后在_ucontext 函数中，我还需要初始化正确的中断状态，即把 sstatus.SIE 位置为 1，使得 CPU 处于开中断状态。最后，我的系统就支持抢占式的分时多任务，其运行效果如图 2.30 所示。

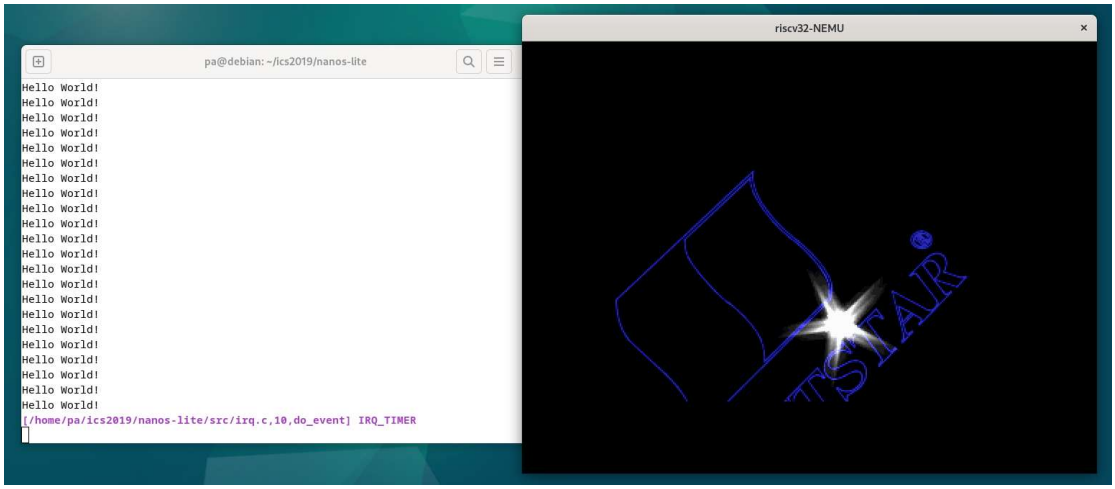


图 2.30 抢占式的分时多任务

3　实验总结与心得

PA 是一项即有趣，同时又非常具有挑战性的课设。它涉及了从计算机系统基础和操作系统等许多知识点，大大地丰富了学生的计算机视野，同时又提升了学生的系统开发能力，它教会了我从计算机系统的角度思考问题。

在 PA1 中，我需要实现一个简易调试器，这个调试器为之后我调试代码的过程中起到了极大的帮助作用，提高了我调试代码的效率。

PA2 则帮助我又一次熟悉了 RISCV32 架构，在这一过程中，我需要阅读大量的 RISCV 手册内容，这加深了我对指令结构的理解。

而 PA3 主要着重于实现系统调用和文件系统，通过 PA3，我理解了自陷指令的工作流程，进一步地，理解了系统调用的流程。而在实现 loader 函数的过程中，我又一次熟悉了 ELF 文件的结构。

PA4 的主要内容是多道程序系统和分页机制，这些知识点同时也是操作系统课程中的重要内容，通过完成这项内容，我又一次加深了对操作系统的理解。

4　原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签字：胡伟江 胡伟江