ICS PA3 Report

集成电路 211870293 李居奇

实验进度

- [x] 完成操作系统自陷和上下文切换
- [x] 完成系统调用和操作系统的 TRM，基础设施
- [x] 完成文件系统和虚拟文件系统
- [x] 完成 NDL 和 miniSDL 库，并用 nemu 成功运行 bird 和仙剑奇侠传 ✌
- [x] 完成 IOE，在 nanos-lite 上直接运行 am-kernel 的打字游戏和 fceux

实验必答题 & 思考题

Question 1

> 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?

`trap.S` 文件的前半部分负责组织上下文结构体。这个上下文结构体作为函数的参数，保存在栈上，方便被调用函数的 `Context` 读取. `trap.S` 的行为和正常的 C 程序调用函数前准备参数的过程是一样的. 首先 `addi sp, sp, -CONTEXT_SIZE` 将栈指针移动后开辟出栈空间，然后调用 `MAP(REGS, PUSH)` 让通用寄存器的值按顺序压栈.

接下来还要将特权寄存器的值上栈。特权寄存器不能用 store 指令直接上栈，因此要先将其值保存到通用寄存器中再压栈:

```
csrr t0, mcause
csrr t1, mstatus
csrr t2, mepc

sd t0, OFFSET_CAUSE(sp)
sd t1, OFFSET_STATUS(sp)
sd t2, OFFSET_EPC(sp)
```

在进入处理函数之前，还执行了 `mv a0, sp`，`a0` 寄存器保存了第一个参数的值。我们后面要调用的函数 `__am_irq_handle` 的参数是 Context *c，因此将当前的 sp 值传进 a0，进入函数后指针 c 就指向了在栈上保存的上下文结构体的首地址. 在整个操作过程中，我们分析调用流程如下：

- 首先在 nemu 运行之初，restart 函数初始化 mstatus 寄存器，使 diff-test 能够正常工作，之后初始化 CTE，在相关寄存器中设置好跳转的位置，并设置好异常处理函数.
- 当指令执行到 `ecall` 时，便直接跳转到 `__am_asm_trap` 汇编函数，在函数中完成上下文切换后跳转到 `__am_irq_handle` 函数
- 根据寄存器 `a7` 解析本次 event 的类型，随后跳转到对应的事假处理函数
- 处理完事假后，回到 `__am_asm_trap`，恢复上下文，继续按序执行指令

```
static void restart() {
  /* Set the initial program counter. */
  cpu.pc = RESET_VECTOR; // the CP in 0x80000000

  /* The zero register is always 0. */
  /* mstaus register is 0x1800. */
  cpu.gpr[0] = 0;
  cpu.csr[_mstatus] = 0x00001800;
}

bool cte_init(Context*(*handler)(Event, Context*)) {
  // initialize exception entry
  // we load the exception entry into mtvec
  asm volatile("csrw mtvec, %0" : : "r"(__am_asm_trap));

  // register event handler
  user_handler = handler;

  return true;
}
```

根据笔者的分析，看似复杂的 trap.S 文件其实只负责设定了上下文切换函数 `__am_asm_trap`，而 `ISA-nemu.h` 负责约定操作系统和 nemu 的传参规则，让后续的异常处理函数，如系统调用，用指定好的寄存器来保存传入和返回的数据.

```
#define GPR2 gpr[10] // a0
#define GPR3 gpr[11] // a1
#define GPR4 gpr[12] // a2
#define GPRx gpr[10] // a0
```

Question 2

> 从 yield test 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, yield test)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, 尤其是一些比较关键的指令/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.

其实这个问题笔者之前已经在上一个问题回答过了🤯，这里在详细的陈述一遍吧：

- 首先在 nemu 运行之初，restart 函数初始化 mstatus 寄存器，使 diff-test 能够正常工作，之后初始化 CTE，在寄存器`mtvec`中设置好跳转的位置，即函数 `__am_asm_trap` 的入口，并设置好异常处理函数.
- 当程序执行到 `yield` 后，便执行以下内联汇编指令：

```
void yield() {
#ifdef __riscv_e
  asm volatile("li a5, -1; ecall");
#else
  asm volatile("li a7, -1; ecall");
#endif
}
```

我们发现这段汇编首先将 `-1` 放入 `a7` 中，作为事假号用于后续程序识别，然后在 nemu 上执行 `ecall` 指令，设置好状态寄存器并保存好当前 PC 后跳转到 `__am_asm_trap`，自此开始模拟 linux 上的 "内核态".

- 在 `__am_asm_trap` 中将相关寄存器保存到栈中，也就是接下来 `__am_irq_handle` 的部分输入参数，完成上下文切换后，直接跳转到 `__am_irq_handle` 去识别事件号.
- 在 `__am_irq_handle` 中识别到该事件为 yield，便跳转到 `do_event` 函数，根据事件为 yield，打印字符串 `"yield\n"`. 至此，事件处理完成.

```
void do_syscall(Context *c);

static Context* do_event(Event e, Context* c) {
  switch (e.event) {
    case EVENT_YIELD:   printf("yield\n"); break;
    case EVENT_SYSCALL: do_syscall(c);     break;
    default: panic("Unhandled event ID = %d", e.event);
  }
  return c;
}
```

- 最后，函数返回到最初的 `__am_asm_trap`，在这里接着完成上下文恢复，调用 mret 指令. 在硬件层面，nemu 识别出 mret 指令后，直接将 pc 恢复为 mepc 的值，并将寄存器的值从内存恢复到寄存器中，至此程序再次回到用户态，nemu 继续执行程序.

Question 3

> 到此为止, PA中的所有组件已经全部亮相, 整个计算机系统也开始趋于完整. 你也已经在这个自己创造的计算机系统上跑起了 hello 这个第一个还说得过去的用户程序 (dummy 是给大家热身用的, 不算), 好消息是, 我们已经距离运行仙剑奇侠传不远了(下一个阶段就是啦).

> 不过按照PA的传统, 光是跑起来还是不够的, 你还要明白它究竟怎么跑起来才行. 于是来回答这道必答题吧:
>
> 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? 它是怎么出现内存中的? 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?
>
> 上面一口气问了很多问题, 我们想说的是, 这其中蕴含着非常多需要你理解的细节. 我们希望你能够认真整理其中涉及的每一行代码, 然后用自己的语言融会贯通地把这个过程的理解描述清楚, 而不是机械地分点回答这几个问题.
>
> 同样地, 上一阶段的必答题"理解穿越时空的旅程"也已经涵盖了一部分内容, 你可以把它的回答包含进来, 但需要描述清楚有差异的地方. 另外, C库中 printf() 到 write() 的过程比较繁琐, 而且也不属于PA的主线内容, 这一部分不必展开回答. 而且你也已经在 PA2 中实现了自己的 printf() 了, 相信你也不难理解字符串格式化的过程. 如果你对 Newlib 的实现感兴趣, 你也可以 RTFSC.

首先 `hello.c` 在 apps 目录下编译生成 ELF 文件，然后该文件被手动粘贴到 nanos-lite 目录下，改名为 `ramdisk.img`.

之后 nanos-lite 中的内联汇编程序 `resources.S` 将 `ramdisk.img` 的内容加载到了内存中，并定义了磁盘内容在内存中开始和结束的位置 `ramdisk_start, ramdisk_end`，代码内容如下：

```
.section .data
.global ramdisk_start, ramdisk_end
ramdisk_start:
.incbin "build/ramdisk.img"
ramdisk_end:

.section .rodata
.globl logo
logo:
.incbin "resources/logo.txt"
.byte 0
```

值得注意的是，磁盘的内部存放数据的地址是不存在地址偏移的，如下代码所示，地址的偏移是由 编译期间便由 Makefile 引入的， `ramdisk.img` 的初始位置被放在了地址 `0x83000000` 处.

```
// file path, file size, offset in disk
{"/share/games/bird/atlas.txt", 1820, 0},
{"/share/games/bird/sfx_swooshing.wav", 352844, 1820},
{"/share/games/bird/splash.png", 4771, 354664},
.......
```

```
CROSS_COMPILE = riscv64-linux-gnu-
LNK_ADDR = $(if $(VME), 0x40000000, 0x83000000)
CFLAGS  += -fno-pic -march=rv64g -mcmodel=medany
LDFLAGS += --no-relax -Ttext-segment $(LNK_ADDR)
```

在调用了 `naive_uload` 函数后，系统解析并加载程序的数据和代码，跳转到 hello 程序 elf 文件中的入口地址，hello 程序调用 `printf` 输出字符的时候，会使用 `write` 系统调用，随后嗲用 `nanos-lite` 中的 fs_write 函数会使用 AM 提供的 `serial_write` 外设来输出.

Question 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, 动画里仙鹤在群山中飞过. 这一动画是通过 `navy-apps/apps/pal/repo/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?

程序在循环中不断读取数据文件，然后不断更新屏幕上仙鹤的位置。在软件层面，程序会先使用 `fseek` 和 `fread` 库函数来定位和读取像素信息，使用 SDL 库函数来输出. fseek 和 fread 函数最终会使用 lseek 和 read 系统调用，SDL 库函数基于我们对系统调用进一步封装的 NDL 库实现，NDL库会使用系统调用 lseek 和 write 来实现像素的定位和写入，`lseek` `read` 和 `write` 最终会调用 libos 中的 `_read`, `_write` 和 `_lseek`，传递对应的系统调用号和相关参数，使用 ecall 指令触发系统调用异常.

```
// 处理各种系统调用号
void do_syscall(Context *c) {
  uintptr_t gpr2_temp = c->GPR2;
  switch (c->GPR1) {
    case SYS_yield: ...
    case SYS_exit:  ...
    case SYS_write: ...
   ...
  }
}
```

`do_syscall` 函数中会识别触发的系统调用的类型，并最终调用文件管理函数中的 `fs_read` `fs_write` 和 `fs_lseek`. `fs_lseek` 比较简单，只是修改文件指针. 对于数据文件，`fs_read` 会使用 `ramdisk_read` 来从磁盘中读取信息，`ramdisk_read` 则使用 AM 上的 klib 库函数 `memcpy` 在硬件上读取. 而 `fs_write` 会使用会使用 AM 提供的 VGA 相关的抽象寄存器和一段像素空间来向屏幕写入像素信息，写入信息后 NEMU 端模拟显卡硬件，再次利用 NDL 库创建真正的图像输出.

一个出现在屏幕上的像素的前世今生是从客户程序到底层硬件的旅行，可以简单记作:

> 客户程序层(PAL) -> 调用 `PAL_SplashScreen()` -> 调用 miniSDL 库绘图函数 -> 调用 NDL 库函数 -> 调用文件读写函数 `read` `write` -> 触发系统调用由此进入系统层 -> 调用 `fb_write` -> 调用 IOE 的函数 `io_write(AM_GPU_FBDRAW,...)`，向设备写入像素 -> NEMU 每次执行完指令后，调用 NDL 库函数，更新图像.

实验结果

STRACE

为方便调试系统调用，笔者在每次系统调用后记录调用号，以及调用函数传入参数和返回值，测试 `hello.c` 后打印结果如下:

```
Hello
Hello World!
Hello World from Navy-apps for the 0th time!
Hello World from Navy-apps for the 1th time!
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!
STRACE:
[12]: type: 7, arg0:0x00000002, arg1:0x00000000, arg2:0x00000000, ret:0x00000000
[11]: type: 7, arg0:0x00000001, arg1:0x00000000, arg2:0x00000000, ret:0x00000000
[10]: type: 7, arg0:0x00000000, arg1:0x00000000, arg2:0x00000000, ret:0x00000000
[9]: type: 4, arg0:0x00000001, arg1:0x830068e0, arg2:0x0000002d, ret:0x0000002d
[8]: type: 4, arg0:0x00000001, arg1:0x830068e0, arg2:0x0000002d, ret:0x0000002d
[7]: type: 4, arg0:0x00000001, arg1:0x830068e0, arg2:0x0000002d, ret:0x0000002d
[6]: type: 4, arg0:0x00000001, arg1:0x830068e0, arg2:0x0000002d, ret:0x0000002d
[5]: type: 4, arg0:0x00000001, arg1:0x830068e0, arg2:0x0000002d, ret:0x0000002d
[4]: type: 4, arg0:0x00000001, arg1:0x8300566c, arg2:0x0000000d, ret:0x0000000d
[3]: type: 4, arg0:0x00000001, arg1:0x830068e0, arg2:0x00000006, ret:0x00000006
[2]: type: 9, arg0:0x83006894, arg1:0x00000310, arg2:0x00000000, ret:0x00000000
[1]: type: 9, arg0:0x83006894, arg1:0x00000418, arg2:0x00000000, ret:0x00000000
[/home/dracacys/ics2023/nanos-lite/src/syscall.c,65,sys_exit]
system halt in EXIT CODE: 0x00000000
[src/cpu/cpu-exec.c:155 cpu_exec] nemu: HIT GOOD TRAP at pc = 0x800010dc
[src/cpu/cpu-exec.c:124 statistic] host time spent = 4,225,392 us
[src/cpu/cpu-exec.c:125 statistic] total guest instructions = 1,335,233
[src/cpu/cpu-exec.c:126 statistic] simulation frequency = 316,002 inst/s
(nemu) q
make[1]: Leaving directory '/home/dracacys/ics2023/nemu'
dracacys@dracacys-virtual-machine:~/ics2023/nanos-lite$
```

图1 strace效果

这里还要介绍一段 debug 的小插曲：在实现 `sys_write` 系统调用后笔者迫不及待的想测试 printf 能不能正常使用，结果无论怎么在 nemu 测试程序都是 `hit bad trap`,但打印结果完全正确. 笔者最初怀疑是 `sys_write` 的实现有问题，最后甚至查 bug 查到了 apps 上的 C 库函数，结果仍然一无所获.

在笔者实现 STRACE 后发现原来 bug 产生的原因是 printf 调用后系统会在退出前进行标准 IO 的 close 操作，而 close 系统调用笔者还未实现，不过仅仅是这样笔者当然也不会找这么久 bug，让笔者大为恼火😓的是框架编写者居然在未实现的函数里默认放入了 exit 而不是 panic，这导致了程序莫名其妙的异常退出，导致了笔者对于 bug 原因的错误判断.

这段经历告诉笔者不要过分迷信框架对于某些细节处理的合理性，同时 bug 不一定在自己写过的代码里，有可能是在还没写的代码里😂，更重要的是，**基础设施**非常重要！！！

系统菜单

完成文件系统，基本图形显示和所有要求的系统调用后，为了方便程序的调试，笔者优先实现了以下程序菜单，可以十分方便地在菜单中选择执行某一应用程序，该程序执行结束后又会自动回到菜单界面，效果如下图所示，回调的原理是反复使用 `sys_execve` 系统调用：

图2 系统菜单

在实现系统菜单的过程中，笔者最为头疼的是图形库函数的编写，就笔者的观点而言，这一部分内容真的应该适当简化，比如说画布等一系列比较抽象的面向对象概念，和 SDL 绘图库函数的阅读等等，毕竟编写绘制图像的函数不应该成为计算机系统设计的重点吧. 同时，这一部分出 bug 是不好解决的，比如下图是笔者遇到的有 bug 的图形输出：

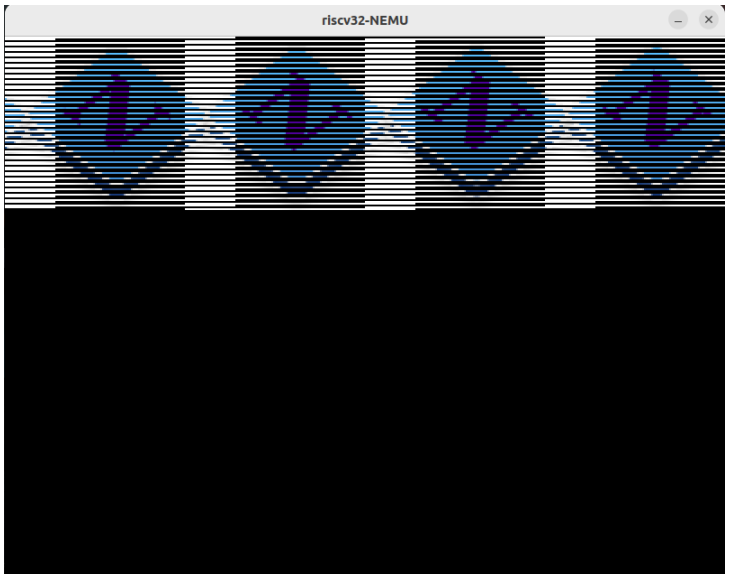

图4 bug 图像打印

bug 查了一天，发现是画布的边界判断错了，无语...

运行 nslider

这个程序综合检验了图像输出和键盘功能，经过了前面菜单的“拷打”，这里基本没有出问题（要说问题那可能是 slide 格式的问题），为了体现一点学科特色，选择 slide 展示如下图所示：

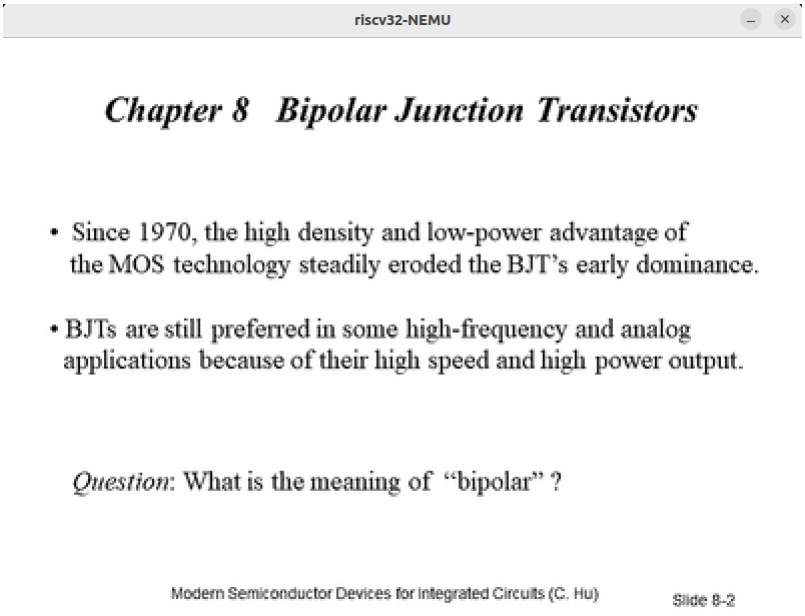

图5 slide 放映效果

运行 flappy bird

从这里开始就需要面对较大型 APP 的调试了，爆出的 bug 那真是一堆一堆的. 笔者花时间最多的一个 bug 是 bird 中 `foef` 函数工作不正常，为了解决这个问题笔者花了超过一天的时间阅读文件处理相关的 C 库函数手册和源码，最后花了两天定位到是自己的 `write` 和 `read` 函数返回值有问题. 整个过程给笔者的惨痛教训就是**读手册的时候要抓住每一个细节**，以及 **C库函数和机器从不骗人**. 在历经几天的调试后笔者成功运行了 flappy bird，效果如下：

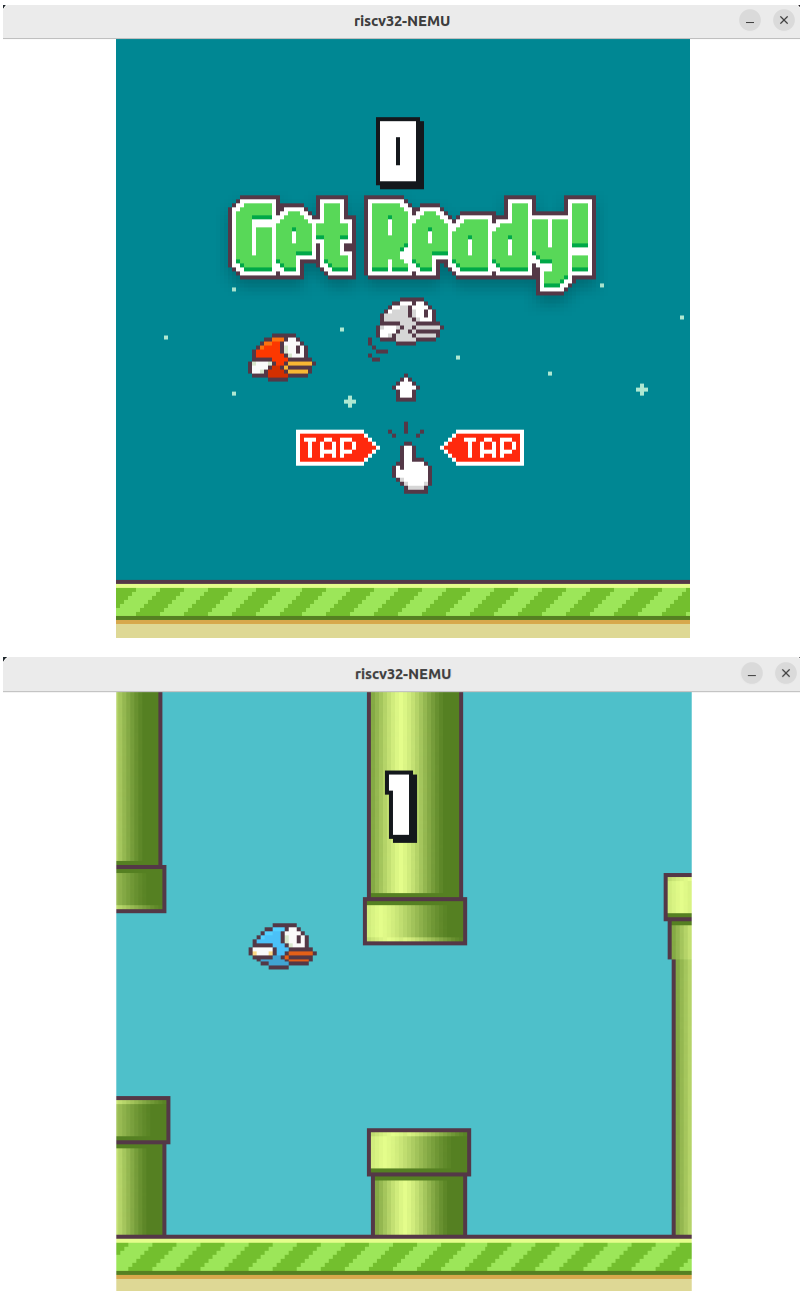

图6 flappy bird 运行效果

运行仙剑奇侠传

仙剑的调试相比 flappy bird 还相对顺利一些，可能严重的 bug 在之前已经解决了，在这里唯一遇到的问题是菜单界面选中的字符被黑框覆盖，后续边界检查发现是本人对画布的理解不清楚导致色块地址已经越界导致的，解决该问题后仙剑奇侠传便能在 nemu 上顺利运行了.

riscv32-NEMU

riscv32-NEMU

riscv32-NEMU

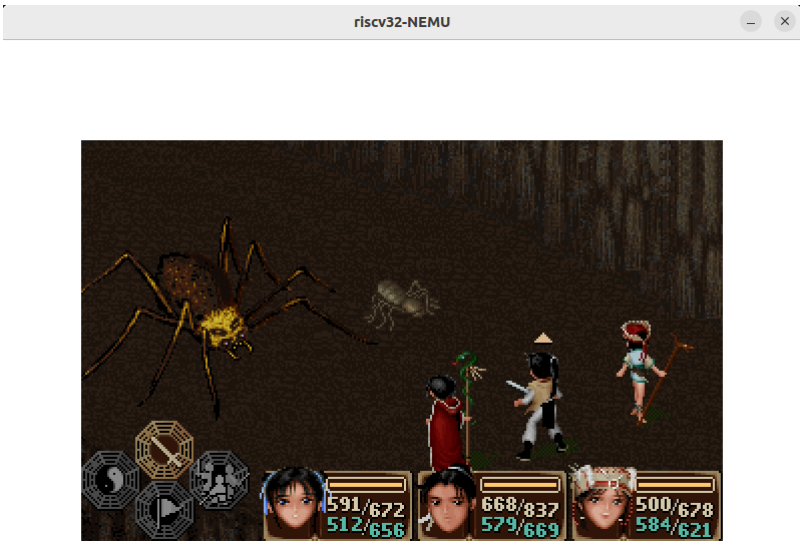

图7 PAL 运行效果

运行 am-kernel 和 fceux

am-kernel 已经是 PA2 的老朋友了，这里笔者完成了可以在 nanos-lite 上运行的 IOE，这样就能让直接在 AM 上跑的程序运行在操作系统上，运行跑分程序和打字小游戏的效果如下图所示：

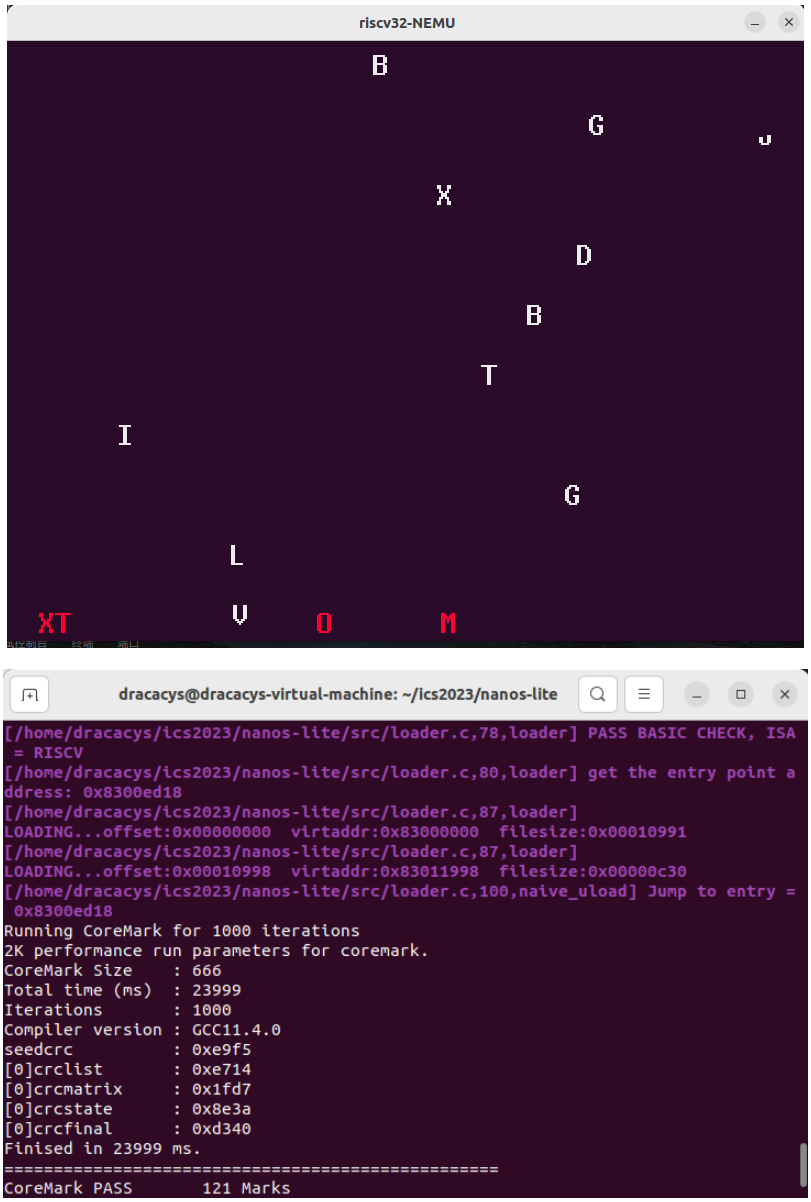

图8 am-kernel 运行效果

mario 的运行把笔者卡了很久，镜像文件 nes 的加载一直是 NULL，根本原因还是笔者对框架不够熟悉. 最后笔者定位到 fceux 的程序入口 `sdl.c`（还是太菜了😭，为啥一开始不这么干呢）：

```
/**
 * The main loop for the SDL.
 */
#ifdef __NO_FILE_SYSTEM__
int main(const char *romname)
#else
int main(int argc, char *argv[])
#endif
{
  ioe_init();

#ifndef __NO_FILE_SYSTEM__
  const char *romname;
  if (argc < 2) {
    romname = "mario3.nes";
    printf("No ROM specified. Deafult to %s\n", romname);
  } else {
    romname = argv[1];
  }

  static char fullpath[128];
  if (romname[0] != '/') {
    sprintf(fullpath, "/share/games/nes/%s", romname);
    romname = fullpath;
  }
#endif

  printf("ROM is %s\n", romname);
  ......
}
```

发现原来镜像的加载是宏 `NO_FILE_SYSTEM` 决定的，修改 fcux-am 的 Makefile 如下，即可支持 nanos-lite 的运行:

```
# 支持 nanos-lite 运行
ifeq ($(AM_HOME),$(NAVY_HOME))
CFLAGS +=
else
CFLAGS += -D__NO_FILE_SYSTEM__
endif
```

这里可以看到镜像已经被成功加载了，但无奈程序太慢了，跑了两分钟连 mario 的影子都没看到.... 读者有耐心可以继续跑跑(doge)

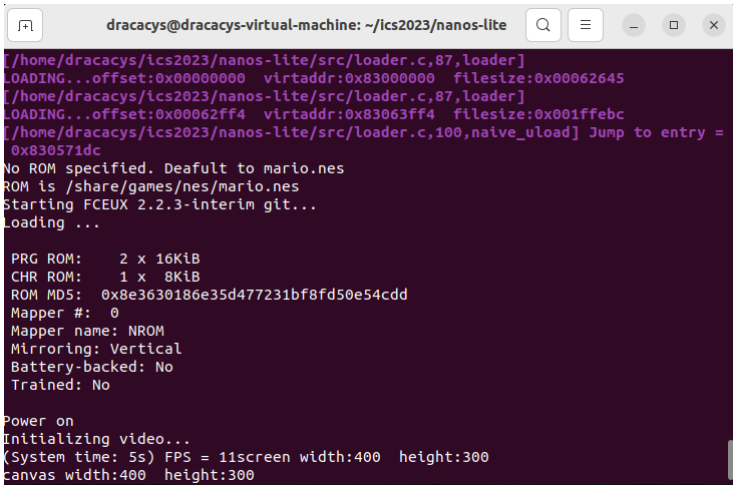

图9 fceux 运行效果

Long way to go（1）

PA3 到此结束，讲义上的必做部分笔者已经全部完成了，但还有一些选做的内容还有探索的空间，等到期末考试结束后再和 PA4 一起改进啦. 同时作为一个带着一生一芯一起学习的非 CS 科班学生，个人感觉这学期在本门选修课的收获似乎超过了以往任何一门原专业的必修课，也不知道是该高兴还是难过.😂 前路漫漫，让我们怀揣系统设计的梦想继续前进吧.