PA3 实验报告

张运吉 (211300063、211300063@ smail.nju.edu.cn)

(南京大学人工智能学院, 南京 210093)

1 实验进度

我已经完成 **PA3 全部必做内容**和**部分选作内容（不包括声卡有关部分）。**

2 必答题

2.1 理解上下文结构体的前世今生 (见PA3.1阶段)

首先梳理一下自陷过程，在 nanos-lite 的 main 函数中，init_irq 会会调用位于 abstract-machine/am/src/nemu/isa/$ISA/cte.c 中的 cte_init()函数，此函数通过一条内联汇编语句 asm volatile("csrw mtvec, %0" : : "r"(__am_asm_trap))把异常入口地址存放在 mtvec 寄存器中，然后注册操作系统特供的处理函数**（这部分是在 AM 完成的）**，yield 函数（AM 中定义）通过一条内联汇编语句 asm volatile("li a7, -1; ecall")把异常号-1 存放在 a7 寄存器（也就是自陷指令对应的异常号），并执行 ecall 指令，ecall 指令的执行就是调用 isa_raise_intr 函数，这个函数返回异常入口地址，然后 pc 指向这个地址（也就是__am_asm_trap 函数的首地址）开始处理异常**（这部分是在 NEMU 完成）**。

__am_asm_trap 是在 trap.S 中定义的，这个函数依次往栈中 push 通用寄存器，mcause，mstatus，mepc 的内容，也就是在这里对上下文结构体 Context *c 赋值，所以 Context 结构体的定义应为：

```
struct Context {
  // TODO: fix the order of these members to match trap.S
  uintptr_t gpr[32];
  uintptr_t mcause, mstatus, mepc;
  void *pdir;
};
```

接着会跳转到__am_irq_handle 函数首地址，在这个函数实现事件的分发并且调用**操作系统**提供的处理函数来处理异常。所以处理异常的过程是在 **nanos-lite 中完成的（do_event 函数）**。然后代码将会一路返回到 trap.S 的__am_asm_trap()中，接下来的事情就是恢复程序的上下文， __am_asm_trap()将根据之前保存的上下文内容, 恢复程序的状态, 最后执行"异常返回指令"返回到程序触发异常之前的状态。

2.2 理解穿越时空的旅程 (见PA3.1阶段)

见 2.1。

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

hello 程序首先被编译成 elf 文件：hello-riscv32(在 navy-apps/tests/hello/build 目录下)，我们手动把它复制到 nanos-lite/build 目录下并命名为 ramdisk.img，于是它就出现在了 nanos-lite 的 ramdisk 中。运行 nanos-lite 时，init_proc 会调用 naïve_uload 函数，这个函数又会调用 loader 函数，loader 函数首先从 ramdisk.img 偏移量为 0 的地方开始读入大小为 sizeof(Elf_Ehdr)的的数据，这些数据就是 elf 文件的 elf 头，根据 elf 头文件的格式，可以获得 program header 的数量，是否加载以及在文件中偏移量，于是 loader 函数根据这些信

息读取 program header 文件偏移量，文件大小，内存大小，加载信息，虚拟地址，将文件读入到 nemu 的内存之中，最后 loader 函数返回入口地址 elf.e_entry，这样 nemu 便可以将 pc 设置为此地址，从而执行 hello 程序。

查看 hello 的源代码，得知 hello 首先会调用 write 函数，最终会调用_write 函数（就是我们在 libos 实现的那个）,_write 函数会调用_syscall_函数，这个函数先把系统调用的参数依次放入寄存器中，然后执行自陷指令（**navy-apps**），接下来就像 2.1 所说那样，nanos-lite 根据异常号识别出是 SYS_write 系统调用时，会调用我们写好的 sys_write 函数来处理这个系统调用，具体地，根据参数 fd 的值，发现 fd 为 2，也就是要向 stdout 文件写入内容（**nanos-lite**），于是便调用 AM 提供的输出接口 io_write 向串口输出（**AM**）。

然后是 printf 过程，printf 打印的字符不一定会马上通过 write 系统调用输出，而是先申请缓冲区，申请缓冲区也是一种系统调用，当缓冲区的字符满了或者遇到'\n'时，才会调用 write 系统调用把这些字符输出到串口，而 write 过程正如上一段讲的那样。

2.4 仙剑奇侠传究竟如何运行

pal 源代码中是在这里读取 mgo.mkf 中的仙鹤像素信息的，为了查清楚进行了哪些系统调用，我开了 strace，又因为 trace 输出的太多太乱，我在 nanos-lite 加了一个 assert 语句 assert(strcmp("/share/games/pal/mgo.mkf", pathname) != 0)，得到结果如第二张图片所示。

```
  PAL_MKFReadChunk(buf, 32000, SPRITENUM_SPLASH_TITLE, gpGlobals->f.fpMGO);
  Decompress(buf, buf2, 32000);
  lpBitmapTitle = (LPBITMAPRLE)PAL_SpriteGetFrame(buf2, 0);
  PAL_MKFReadChunk(buf, 32000, SPRITENUM_SPLASH_CRANE, gpGlobals->f.fpMGO);
  Decompress(buf, lpSpriteCrane, 32000);
```

```
[/home/iuzyj/ics2022/nanos-lite/src/fs.c,63,fs_open] sys_open! fileneme: /share/
games/pal/mgo.mkf
[/home/iuzyj/ics2022/nanos-lite/src/fs.c,64,fs_open] system panic: Assertion fai
led: strcmp("/share/games/pal/mgo.mkf", pathname) != 0
```

PAL_MKFReadChunk 最后会调用 fread 函数（**库函数**），从/share/games/pal/mgo.mkf 中读取信息，这一过程会触发 SYS_write 调用，调用位于 **libos** 中的_syscall_，_syscall_又会调用 **nanos-lite** 中实现的 do_syscall，然后事件分发，调用 fs_read 函数来读取/share/games/pal/mgo.mkf 中的信息。系统调用读取数据的过程中，会使用 **AM** 提供的基本环境，如 memcpy 等，最后这些行为都被编译成为指令，在 **nemu** 上运行。

下面这张图片所示代码是把仙鹤更新到屏幕上：

```
//
// Draw the cranes...
//
for (i = 0; i < 9; i++)
{
   LPCBITMAPRLE lpFrame = PAL_SpriteGetFrame(lpSpriteCrane,
      cranepos[i][2] = (cranepos[i][2] + (iCraneFrame & 1)) % 8);
   cranepos[i][1] += ((iImgPos > 1) && (iImgPos & 1)) ? 1 : 0;
   PAL_RLEBlitToSurface(lpFrame, gpScreen,
      PAL_XY(cranepos[i][0], cranepos[i][1]));
   cranepos[i][0]--;
}
iCraneFrame++;
```

这些函数会使用在 libs 中 miniSDL 中的完成的 SDL_Update ,SetPalette 等**库函数**，这些函数又是通过 NDL 函数来实现，NDL 中的函数又使用了 fwrite 等，触发**系统调用（libos 和 nanos-lite）**，在系统调用中通过 **AM** 提供的 io 接口（io_write 以及 io_read）来将画面等信息传递给硬件以及从硬件获取信息，于是库函数，libos，nanos-lite, am, nemu 就协同了起来，最后这些用户程序运行在硬件 nemu 上，完成开始动画的显示。

2.5 其他说明

我的 strace 开关放在/nanos-lite/include/common.h 中，因为我觉得系统调用是和操作系统相关的，放在操作系统的代码中更合适，所以我不像以前实现的 trace 放在 nemu 的 menuconfig 中。（虽然不知道这样做对不对）

```
#define HAS_CTE
//#define HAS_VME
//#define MULTIPROGRAM
//#define TIME_SHARING
//#define STRACE
```

3 PA 的尽头是 debug

3.1 实现异常相应机制

一开始看讲义上的这部分内容一头雾水，于是就多看几遍吧！最后终于发现了讲义上的一句话“你需要自己添加一些寄存器”，这时候我才意识到原来要自己定义 CSR，然后包含到 CPU_state 中。我查看 riscv 的手册，了解了需要添加的寄存器的结构与作用。下面是我对寄存器的扩充代码：

```
typedef struct {
  word_t mtvec;
  word_t mepc;
  word_t mstatus;
  word_t mcause;
} CSR;

typedef struct {
  word_t gpr[32];
  vaddr_t pc;
  CSR csr;
} riscv32_CPU_state;
```

3.2 保存上下文和恢复上下文

这一部分及其复杂，涉及到 nemu，am 和 nanos-lite，必须反复看讲义和 RTFSC 理解每一处细节之后才有所头绪。

3.3 堆区管理

这部分遇到的一个很无语的错误是我没有及时更新 ramdisk.img 从而导致我实现了堆区管理之后发现 hello 程序还是一个一个输出字符，我一直以为是我的堆区管理实现错了，直到发现了这个问题后才明白这是多么低级的失误！这里消耗了很多时间！

3.4 把VGA显存抽象成文件

在实现这个功能的时候，我一直有一个 bug 无法解决，我运行 bmp-test 的时候像素堆叠在屏幕最上层，我找了好久才发觉有可能是 offset 计算错误，但是我使用输出调试法发现 offset 计算也是正确的，我迟迟无法定位到错误点，在偶然一次下午，我灵光一现可能是实现文件系统的时候 fs_write 好像没有设置偏移

```
if (finfo->write) {
  // 一开始offset参数只是传0,导致这个bug好久都没d出来！！！
  return finfo->write(buf, finfo->open_offset, len);
}
```

就是这里!!! 第二个参数我一开始默认传的是 0，所以在调用 fb_write 的时候计算出来的屏幕坐标（x，y）总是错的。

3.5 PA2的帐终于要还啦

PA2 的时候我的跑分就很慢（我的一个舍友和我一样），但是当时并不影响 oj，所以我也就不管了，但在我实现 nterm 的时候，我的光标闪得特别慢，输入一个字符也要等待好久，经过 ICS 课程群里大佬的解答，可能是时钟源的问题，网上查了一圈，发现 AMD 架构下得时钟源只有 hept 和 acpi_pm，没有 tsc（tsc 跑的比较快），由于我用的是双系统，好像硬件上也不支持更换成 tsc，于是我只能退而求其次改用虚拟机，然后又是一个心态爆炸的夜晚，因为我要移植我的 pa 项目到虚拟机下，我直接压缩通过邮箱发送的方式好像并不能运行，因为虚拟机和真机的环境肯定不一样，所以我干脆重新做了 pa3（事实上就是 copy 一遍），真是折磨！

4 实验心得

机器永远是对的!!! 没有经过测试的代码永远是错误的!!! 即使经过足够的测试，你的代码仍然可能有 bug!!! 这是我做 pa3 最大的感悟。经过 pa3，我也认识到了掌握一些高效的调试方法的重要性。

我对整个 pa 的框架代码也有了更清楚的了解，从 nemu，am 到 nanos-lite，navy-apps，每一层都抽象的很精妙，必须要清楚了解这四个层次背后的联系与不同，才能较好地写完 pa3。

我对整个计算机系统有了更深入的了解，特别是在硬件和软件配合的部分，我也能初探操作系统的原理，特别是关于系统调用方面（因为在 pa3 很大部分都是完善系统调用），我清楚了系统调用的大致流程，这对于我理解“计算机是个抽象层”有很大的帮助。

在看到我的 nemu 能成功跑起来这些精彩纷呈的应用程序时，我真的很开心，特别是在 pal 能跑之后，有一种如释重负的感觉，虽然可能很多地方的代码还有待完善，但最起码它能跑起来了，这也符合讲义里说的：“先完成再完善”。