PA3 实验报告

王卫东　221900332

2023 年 12 月 28 日

一　必做题

1　理解上下文结构体的前世今生

1.1　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。`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。第一句

```
addi sp, sp, -CONTEXT_SIZE
```

为 Context 结构体开辟了空间。

```
MAP(REGS, PUSH)
csrr t0, mcause
csrr t1, mstatus
csrr t2, mepc
STORE t0, OFFSET_CAUSE(sp)
STORE t1, OFFSET_STATUS(sp)
STORE t2, OFFSET_EPC(sp)
```

这里将通用寄存器的值按顺序上栈。然后将特权寄存器的值上栈。因为特权寄存器不能用 store 指令直接上栈，因此要先将其值保存到通用寄存器中再上栈。

在进入处理函数之前，最后一句话是 `mv a0, sp`，通过 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。

Context 结构体的定义在 `$ISA-nemu.h` 中，通过定义好的的寄存器存储信息正确地实现保存上下文的功能。

2 理解穿越时空的旅程

2.1 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．

Nanos-lite 调用 yield() 之后，执行了两条汇编指令（这两条汇编指令是用内联汇编的方式直接嵌入的）：

```
asm volatile("li␣a7,-1;ecall");
```

其中第一条指令向 a7 寄存器写入 -1，a7 寄存器是约定中传递中断类型的寄存器。第二条指令 ecall 则是“中断”指令。mtvec 中的地址是函数 ___am_asm_trap 的地址，在 cte_init 中完成了 mtvec 内容的初始化:

```
asm volatile("csrw␣mtvec,␣%0" : : "r"(__am_asm_trap));
```

函数 ___am_irq_handle 会根据上下文结构体的内容打包出一个事件结构体 ev。根据上下文结构体中的 mcause 寄存器的值，可以识别事件的类型，如自陷事件的事件号为 -1。打包完事件结构体后，___am_irq_handle 会将上下文结构体和事件结构体一起传给处理函数。这个处理函数是在 cte_init 中传进来的 do_event。

do_event 函数检查事件结构体中的事件类型,当看识别为 EVENT_YIELD 时，给 mepc 的值 +4，这个 +4 操作直接修改了上下文结构体中的 mepc 的保存值。

这时进入 trap.S 的后半部分，后半部分将栈上保存的寄存器内容恢复，然后调用 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。至此，时空穿越的旅程结束。

3 hello 程序是什么, 它从而何来, 要到哪里去

3.1 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?

通过阅读 makefile, 可以得知 hello 的 elf 文件最初在 hello 的 build 文件夹下, 在执行 install 伪目标后, 会复制到 fsimg 文件夹下的 bin 文件夹中, 在生成 ramdisk.img 时 (ramdisk.img 通过逐个直接复制 fsimg 下的文件生成), 会被直接复制到 ramdisk.img 中, 并同时将 hello 程序在 ramdisk.img 的起始地址和大小保存到 files.h 中。

ramdisk.img 被 resourse.S 中的.incbin "build/ramdisk.img"指令加载到内存中, 同时定义了全局变量 ramdisk_start 和 ramdisk_end 两个变量来指示 hello 程序的起始和终止位置; 并且根据 Makefile 可以得知,ramdisk.img 被放在了地址 0x83000000 处，这是通过 Makefile 中的 LNK_ADDR 实现的。

在调用了 naive_uload 函数后，系统会进入 hello 程序的汇编代码（elf 文件中有入口地址), hello 程序输出字符的时候, 会使用 write 系统调用（直接使用 write 或者通过 printf 库函数使用 write)，nanos-lite 中的 fs_write 函数会使用 AM 提供的接口来输出（在实现了文件系统之后，fs_write 会直接使用 stdout 文件对应的写函数 serial_write)。

4 仙剑奇侠传究竟如何运行

4.1 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? (Hint: 合理使用各种 trace 工具, 可以帮助你更容易地理解仙剑奇侠传的行为)

先说明一下 PAL 需要用到的库：compiler-rt,libc,libfixedptc, libminiSDL,libndl,libos.

参考 PAL_SplashScreen() 函数的注释，可以大概分析出应用程序的行为:

- 分配空间
- 创建屏幕:VIDEO_CreateCompatibleSurface → VIDEO_CreateCompatibleSizedSurface → SDL_CreateRGBSurface, 本质上是分配空间，初始化屏幕.
- 读取文件: PAL_MKFReadChunk → fseek → fread 通过 libc 库，最终调用 _read，触发系统调用, 操作系统通过 fs_read 读取文件
- 生成仙鹤的位置坐标
- 清除事件:PAL_ProcessEvent & PAL_ClearKeyState(调用 SDL_GetKeyState)
- 读取当前时间: SDL_GetTicks → NDL_GetTicks 触发系统调用 SYS_gettimeofday
- while(1) 死循环:
  - 读取键盘事件 (PAL_ProcessEvent)
  - 设置并更新调色板
  - 更新屏幕
    - 背景：VIDEO_CopySurface → SDL_BlitSurface
    - 仙鹤 & 标题：更新像素信息
    - VIDEO_UpdateScreen → (SDL_SoftStretch) / SDL_FillRect / SDL_UpdateRect
    - SDL_UpdateRect → NDL_DrawRect → 打开文件 `/dev/fb`, 调用 fb_write 修改像素信息
  - 检查键盘，判断是否结束开始动画

二 实验心得

pa3 写了大概两周，第一周完成到 3.2（大概花了两个半下午），之后花了四天完成第 3 阶段。在完成三阶段的时候每实现一个新的功能都要 debug 半天（前面代码的小问题不断暴露出来）。

在实现 Nterm 的时候一直报错出现 fsleek 越界（offset = 200w+），对照了一个晚上都没有看出什么。猜测是 fb_write 实现的时候绘制画布有问题，初始画布太大，但是没有找到具体的调用。

```
void NDL_DrawRect(uint32_t *pixels, int x, int y, int w, int h) {
int fd = open("/dev/fb", 0, 0);
// printf("canvas_w: %d\ncanvas_h: %d\n",canvas_w,canvas_h);
for(int i = 0; i < h && y + i < canvas_h; i++){
    lseek(fd,((y + canvas_y + i) * screen_w + canvas_x + x) * 4,SEEK_SET);
    write(fd,pixels + i * w,((w + x < canvas_w) ? w : (canvas_w - x))*4);
    }
}
```

于是就回到 pa3.2 把之前实现的较为繁琐的 fs_read/write 重新作了修改（当时同时维护两个表，非常的麻烦），修改完之后尝试再进行一些 printf 的调试，发现初始画布大小确实有问题，猜测传高度宽度大小数据错误，就进一步打印 dispinfo 传入`/dev/fb` 的数据，发现 height 变为了原来的几十倍，终于找到问题：一开始开了临时数组使用 strcpy 传入 *buf 中，没有加入" 导致写入的时候多写了数字。删掉临时数组改为 snprintf 就好...（一定要仔细处理字符串函数 ~）

在使用 malloc 的时候发现 native 和 riscv32 实现的不一样 (感觉是 klib 的问题)，导致调用的时候会有一边不正确，后来把这些 malloc 都改成数组了。(...)

在运行仙剑的时候先是发现会卡住（时钟问题，要减去初始时间），其次就是对于键盘无反应，这是因为 pollevent 和 getkeystate 调用先后导致的，所以只需要在 SDL_Pollevent 中更新 keystate 数组，在 Geteytate 中只需要返回这个数组即可。对于字幕黑框的问题则是需要在更新画布的时候将处理的像素数组 pixels 加上偏移量 x,y 坐标保证取的是正确位置的像素（而不是一直在左上角 w*h 的地方）。

总之，在实现 pa3 的过程中最重要的心得就是要 RTFSC 和 RTFM，保证和手册上标准实现的功能一致，才不会遇到太多奇怪的错误。同时，正确的调试工具和耐心的调试打印是必不可少的。