**pa3　191240025　梁天润**

**NOTE:由于在本地手动保存了很多副本，导致原有仓库pack过大，遂重新建了一个仓库来提交。**

**1、新仓库手动复制了原仓库的pa代码，故commit信息几乎没有，commit信息请看之前仓库所提交的记录。(原仓库pa3最后一次提交是#DuI9DmEi ,原仓库pa3 traced 8000+)**

**2、新仓库一次性建立了pa1-4的分支，并拷贝了完成到pa3或pa4.1的代码，故代码中有很多之后pa的内容。**

**3、仅重新提交了pa2-4。pa0，pa1为原仓库提交。**

**完成情况: 完成所有必做内容，通过oj。所有apps可以正常运行，**

**选做内容：支持屏幕居中，自由开关DIFFTEST**

系统展示：

nterm：支持键入文件名打开文件，失败时terminal有打印信息。

　　键入exit 退出nterm，键入clear清屏

nslider：按q键退出

menu:　按ESC键退出

pal运行截图


**必答题:**

1.上下文结构体的前世今生

trap.S中的那一条诡异的pushl %esp指令，将context的位置（即栈底的位置）传入吓一跳指令调用的__am_irq_handle。所有的context内容通过push压栈保存 相关代码如下：

```
rtl_push(s,&ret_addr);          //eip
rtl_push(s,&cpu.cs);            //cs
rtl_push(s,&cpu.eflags);        //eflags

0x0010081c:   68 81 00 00 00                          pushl $0x81   //irq号
0x00100821:   eb 08                                   jmp 10082b
0x0010082b:   60                                      pusha         //gprs
0x0010082c:   6a 00                                   pushb $0x0    // cr3
0x0010082e:   54                                      pushl %esp    //传入   &
(comtext)
0x0010082f:   e8 74 fe ff ff                          call 1006a8  //调用
__am_irq_handle
```

nemu.h中定义的Context 结构体的数据就是上面保存的数据，所以相应的排列结构体成员即可。

2.理解穿越时空的旅程。

yield() 内容是用内联汇编写的一条INT 汇编指令 int 0x81，nemu执行时调用了raise_intr：将context压栈，并按NO = 0x81，设置跳转地址到0x81号异常处理程序的地址。该异常处理程序会jmp *_am_asm_trap，后压栈传参异常号，调用*am_irq_handle。按传入的异常号设置ev.event 为EVENT_YIELD，最后调用do_events进行事件处理。并返回context的位置。am_irq_handle将该地址返回给_am_asm_trap，后恢复上下文并iret,返回继续执行原来的程序。

3.hello程序是什么，它从哪里来，要到哪里去

- hello的加载和执行

hello.c在navy-app中被编译成可执行文件，后作为ramdisk被链接入nanos-lite的代码中，（ramdisk_start = 0x101fc0）。

通过loader，将ramdisk中的需要执行的segment载入0x3000000左右的内存，并将pc设置到程序入口（由ELF中的entry指定），执行hello的程序。显然第一条指令就在程序入口处（0x30052be）

- 字符的打印

  用printf实现的打印，会先触发SYS_brk系统调用来申请堆区，若不成功，则一个一个字符打印。最终是调用write函数来完成打印工作。navy中的write函数会触发一次系统调用，调用号为SYS_write, 传入fd, char* str, len 三个参数。表示输出到stout。输出是通过处理系统调用do_syscall,最终调用sys_write来打印字符的。

4.仙剑奇侠传就行如何运行。

以打印信息到屏幕这一过程为例：

Pal用SDL实现屏幕的更新，最终调用的是SDLmini里的库函数，以SDL_UpdateRect，为例，设置好打印的信息后调用NDL_Draw来更新屏幕。关键的函数是write(), 这个Navy的简易c库里的函数，调用了_write_r，并最终通过调用 libos 里的 _write() ,进行系统调用。所有斯通调用的关键是

```
intptr_t _syscall_(intptr_t type, intptr_t a0, intptr_t a1, intptr_t a2) {
  register intptr_t _gpr1 asm (GPR1) = type;
  register intptr_t _gpr2 asm (GPR2) = a0;
  register intptr_t _gpr3 asm (GPR3) = a1;
  register intptr_t _gpr4 asm (GPR4) = a2;
  register intptr_t ret asm (GPRx);
  asm volatile (SYSCALL : "=r" (ret) : "r"(_gpr1), "r"(_gpr2), "r"(_gpr3), "r"
(_gpr4));
  return ret;
}
```

可以看到在按规约设置了寄存器的内容（作为syscall的参数）后，进行了syscall指令。

与yield()一样，先保存context，跳转到由异常号0x80指向的处理程序，同样是*am_asm_trap，后压栈传参异常号，调用*am_irq_handle。按传入的异常号设置ev.event 为EVENT_SYSCALL，最后调用do_syscall进行事件处理。

do_syscall依据syscall指令传入啊系统调用号（在gpr1中），如 _write()的系统调用号为SYS_write，而调用系统调用处理程序sys_write。

通过调用fs_wite()来处理Sys_write 系统调用 。

由于传入要写的文件是屏幕，fs_write最终调用的是fb_write。该函数通过nanos-lite里的inl(VGACTL_ADDR)，来获取屏幕设备的内存映射地址，并进行值的传入，最后outl(SYNC_ADDR,1)更新屏幕;

nanos-lite里的 in(), out(),是通过直接写汇编指令in ,out 实现的。最终由nemu执行这些指令，完成屏幕的打印。

**选做题：**

1.对比异常处理和函数调用

函数调用需要保存和恢复的是约定好的一些寄存器的值，而上下文切换需要保存所有gpr的值，以及epi，cs(来保存pc位置)，以及eflags（机器状态）。函数调用结束后只需要恢复调用前的现场，然后pc继续向下执行。而上下文切换需要恢复包括pc，eflags，gprs等一切信息。

2。诡异的x86代码

pushl %esp 当前的esp存放的是ceontext的顶部（或者说保存context的栈的底部）。然后吓一跳语句调用了__am_irq_handle，该值作为参数传入。所以，传入的参数c，即为context的地址。

3.从加4的角度看CISC和RISC

CISC用硬件来判断不同异常的处理要不要加4，但相应的每个异常的调用都要遵从规范，而RISC则需要用软件来弥补硬件上的无法判断异常是否要+4的不足。硬件实现的判断快，但是不同ISA之间无法兼容，软件可以兼容但是慢。

我认为交给软件来处理更合适，相较于算力的一点点损失，可移植可兼容的代码更好。

4.堆和栈在哪里

堆和栈的数据是代码运行时产生的，所以不在可执行文件里面。只需要提供维护堆和栈的机制，代码执行的过程中会将数据存取到堆或栈上。

5.如何识别不同格式的可执行文件

文件开头的Magic Number 说明了可执行文件的文本格式。

6.冗余的属性

filesize指的是elf文件中的大小，memsize指的是存放到内存中需要的空间。由于一些数据在ELF中并不需要赋值，如未初始化的变量或者初始化为0的全局变量。而在放入内存时需要空出这一段空间。

7.为什么要清零

.bss段的数据，即未初始化或者初始化为0的全局变量就是存放在这一段内存中的，所以需要清零。

8.系统调用的必要性

实现批处理系统不必须用系统调用，但是一些功能，如堆区的控制，有可能修改其他内存区域的数据，而能否修改是需要操作系统来判断的。所以用系统调用来实现批处理系统，是为了方便系统管理内存。

9.比较fixedpt 和 float

fixept由于没有阶码的设置，1.能表示的数的范围基本等于integer的范围 2. 精度就是faction部分的精度。 这两点同float想不是很不足的。float无论是精度还是广度都是大于fixept。但是采用定点算数的好处是，运算只需要在整数运算上稍加修改即可，而float运算不能由整数运算简单转换得来。

10.神奇的LD_PRELOAD

LD_PRELOAD是一个环境变量，动态库加载是优先级最高。通过LD_PRELOAD,将加载文件的路径改成了/navy-apps/bin