PA3 实验报告

1 上下文结构体

`保留的上下文`是指计算机硬件在发生异常时，保存的当前状态信息，这些信息包括系统寄存器的状态，程序计数器`PC`的值以及其他系统状态，上下文的保留是为了帮助系统处理异常后的正确性与恢复性。

具体的，`__am_irq_handle`函数作为系统异常的事件分发函数，在硬件`NEMU`遇到异常后根据触发异常时保存的标志位信息决定处理怎样的异常。

```
10  struct Context {
11
12    uintptr_t gpr[NR_REGS], mcause, mstatus, mepc;
13    void *pdir;
14  };
```

以`riscv32`为例，上下文结构体`Context`中主要包含系统寄存器，即`NEMU`中所定义的具有`ISA`特色的寄存器模式。

上下文结构体的出现是由于计算机硬件出现异常后，由`CPU`自动打包而成的结构，通过指针传输到异常处理函数进行下一步处理。

其中`mcause,mstatus,mepc`为异常标志位寄存器，`gpr[32]`为`CPU`中的通用寄存器，在执行异常处理指令时赋值

```
csrrw   , I, R(rd) = (*csr_address(imm)); (*csr_address(imm)) = src1);
csrrs   , I, R(rd) = (*csr_address(imm)); (*csr_address(imm)) |= src1);
```

`ISA-nemu.h`与`NEMU`中新实现的指令都是特定架构下，硬件层面响应系统异常的机制，`AM`中的系统异常分发函数属于异常处理程序，`trap.s`实现的是不同架构下的上下文切换，当硬件异常中断时，跳转到系统异常处理程序。

YIELD系统调用

```
80000264 <hello_intr>:
80000264:   fe010113           addi    sp,sp,-32
80000268:   00002517           auipc   a0,0x2
8000026c:   d0050513           addi    a0,a0,-768 # 80001f68 <_etext>
80000270:   00112e23           sw  ra,28(sp)
80000274:   00812c23           sw  s0,24(sp)
80000278:   351010ef           jal ra,80001dc8 <printf>
8000027c:   00002517           auipc   a0,0x2
80000280:   dcc50513           addi    a0,a0,-564 # 80002048 <_etext+0xe0>
80000284:   345010ef           jal ra,80001dc8 <printf>
80000288:   00c10593           addi    a1,sp,12
8000028c:   00700513           li  a0,7
80000290:   6b5000ef           jal ra,80001144 <ioe_read>
80000294:   00100513           li  a0,1
80000298:   00989437           lui s0,0x989
8000029c:   1e0010ef           jal ra,8000147c <iset>
800002a0:   67f40413           addi    s0,s0,1663 # 98967f <_entry_offset+0x98967f>
800002a4:   00012623           sw  zero,12(sp)
800002a8:   00c12783           lw  a5,12(sp)
800002ac:   00f44c63           blt s0,a5,800002c4 <hello_intr+0x60>
800002b0:   00c12783           lw  a5,12(sp)
800002b4:   00178793           addi    a5,a5,1 # 1001 <_entry_offset+0x1001>
800002b8:   00f12623           sw  a5,12(sp)
800002bc:   00c12783           lw  a5,12(sp)
800002c0:   fef458e3           bge s0,a5,800002b0 <hello_intr+0x4c>
800002c4:   1ac010ef           jal ra,80001470 <yield>
800002c8:   fddff06f           j   800002a4 <hello_intr+0x40>
```

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

用户态调用`yield`进入内核态

`hello_intr`函数于`pc=800002c4`处跳转至`yield`函数处，模拟硬件的`NEMU`将会模拟触发一个中断异常，具体表现为`NEMU`执行`ecall`指令，此过程中将`a7`即系统调用号寄存器的值设置为`-1`后，程序执行`ecall`指令，将系统的控制权交给在`mtvec`寄存器中所设置的异常处理程序的入口`__am_asm_trap`，此过程中处于内核态

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

  // register event handler
  user_handler = handler;

  return true;
}
```

当硬件发生异常时，系统跳转到异常处理函数`__am_irq_handle`函数，该函数根据`yield`函数所设置的系统调用号的值进行事件分发，跳转到处理`yield`系统自陷，异常处理函数根据硬件触发异常时的上下文内容打包处理一个类型为事件的结构体，传递给`user_handle`函数处理，屏幕上出现打印出的`y`字符，说明异常处理函数接收到系统打包的事件并成功处理自陷

```
Hello, AM World @ riscv32
  t = timer, d = device, y = yield
yyy[src/cpu/cpu-exec.c:106 statistic] host time spent = 6,983,926 us
[src/cpu/cpu-exec.c:107 statistic] total guest instructions = 162,022,982
[src/cpu/cpu-exec.c:108 statistic] simulation frequency = 23,199,412 inst/s
```

返回用户态

成功处理自陷后，将根据所保存的上下文中`mepc`寄存器的值决定执行触发异常时的下一条指令，直到再次触发`yield`函数，因此屏幕上将循环打印`y`字符

HELLO程序

加载Ramdisk磁盘文件

```
.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
```

`ramdisk.img`模拟磁盘文件，在编译`hello`程序到`ramdisk.img`中后，通过`.incbin`指令将`ramdisk.img`包含进`nanos-lite`中后加载制定文件（实现文件系统后）

```
if (RAMDISK_SIZE!=0)
{
  printf("The ramdisk isn't empty\n");
  //naive_uload(NULL, "/bin/bmp-test");
  //naive_uload(NULL, "/bin/event-test");
  //naive_uload(NULL, "/bin/file-test");
  //naive_uload(NULL, "/bin/nslider");
  //naive_uload(NULL, "/bin/menu");
  naive_uload(NULL, "/bin/hello");
}
```

```
static uintptr_t loader(PCB *pcb, const char *filename) {

  int fd = fs_open(filename, 0, 0);
  if (fd < 0) {
    panic("should not reach here");
  }
  Elf_Ehdr elf;

  assert(fs_read(fd, &elf, sizeof(elf)) == sizeof(elf));

  assert(*(uint32_t *)elf.e_ident == 0x464c457f);

  Elf_Phdr phdr;
  for (int i = 0; i < elf.e_phnum; i++) {
    uint32_t base = elf.e_phoff + i * elf.e_phentsize;

    fs_lseek(fd, base, 0);
    assert(fs_read(fd, &phdr, elf.e_phentsize) == elf.e_phentsize);


    if (phdr.p_type == PT_LOAD) {

      char * buf_malloc = (char *)malloc(phdr.p_filesz);

      fs_lseek(fd, phdr.p_offset, 0);
      assert(fs_read(fd, buf_malloc, phdr.p_filesz) == phdr.p_filesz);

      memcpy((void*)phdr.p_vaddr, buf_malloc, phdr.p_filesz);
      memset((void*)phdr.p_vaddr + phdr.p_filesz, 0, phdr.p_memsz - phdr.p_filesz);
```

通过文件在磁盘`ramdisk.img`中的偏移量能够定位到`ELF`格式的文件所在位置，通过程序头表中的信息加载需要加载的程序段，跳转到程序执行的入口`elf.e_entry`

系统调用

在`hello.c`程序中，当遇到系统调用`write`与`printf`时，系统都会模拟触发异常流，将控制权传给内核态的`nanos-lite`来进行下一步处理

```
AM cte switch define the event 4
nanos irq 2
Syscall 4
In the sfs, we abstarct the stdout into a file
Hello World!
AM cte switch define the event 9
nanos irq 2
Syscall 9
AM cte switch define the event 9
nanos irq 2
Syscall 9
AM cte switch define the event 4
nanos irq 2
Syscall 4
In the sfs, we abstarct the stdout into a file
Hello World from Navy-apps for the 2th time!
AM cte switch define the event 7
nanos irq 2
Syscall 7
AM cte switch define the event 7
nanos irq 2
Syscall 7
AM cte switch define the event 7
nanos irq 2
Syscall 7
AM cte switch define the event 0
nanos irq 2
Syscall 0
[src/cpu/cpu-exec.c:138 cpu_exec] nemu: HIT GOOD TRAP at pc = 0x80000f9c
```

`hello.c`程序所需要的系统调用一共经过中断异常，事件分发，系统调用，文件读写四个步骤，涉及`AM`,`NEMU`,`nanos-lite`三个部分协调合作，通过插入调试法，可以看到每一个部分的具体行为

以`write`系统调用举例：程序执行`write`函数将触发系统调用，硬件`NEMU`将模拟一个中断，`AM`中的中断处理函数将根据`mcause`寄存器的值识别出异常类型为系统调用，控制权将给`nanos-lite`，程序跳转到`do_event`函数进入系统调用`syscall`函数

```
int _write(int fd, void *buf, size_t count) {
  return _syscall_(SYS_write, fd, (intptr_t)buf, count);
}
```

```
case SYS_exit :  do_sys_exit()  ; break;
case SYS_yield:  do_sys_yield(c); break;
case SYS_write:  do_sys_write(c); break;
case SYS_brk  :  do_sys_brk(c)  ; break;
case SYS_read :  do_sys_read(c) ; break;
case SYS_open :  do_sys_open(c) ; break;
case SYS_lseek:  do_sys_lseek(c); break;
case SYS_close:  do_sys_close(c); break;
case SYS_gettimeofday: do_sys_gettimeofday(c); break;
default: panic("Unhandled syscall ID = %d", a[0]);
```

```
void do_sys_write(Context *c)
{
  int fd = c->GPR2;
  intptr_t buf = c->GPR3;
  size_t count = c->GPR4;
  /*if (fd==1||fd==2)
  {
    for (int i=0;i<count;i++)
    {
      putch(*(((char *)buf)+i));
    }
    c->GPRx = count;
  }
  else
  {
    c->GPRx = fs_read((int)c->GPR2, (void *)c->GPR3, (size_t)c->GPR4);
  }*/
  c->GPRx = fs_write((int)fd, (void *)buf, (size_t)count);
}
```

处理`write`系统调用，根据保存的上下文，将缓冲区的字符通过串口输出到屏幕上，由于在后续阶段已经实现了文件系统，于是将标准输出抽象为文件读写，所注释掉的绿色部分为正常通过`putch`直接输出到串口的部分

PAL

这段代码实现了一个游戏中的开场动画（可能是《仙剑奇侠传》中的开场动画），其中包括了仙鹤飞过的效果。我们可以通过以下几个方面来分析如何实现这些效果：

1. 初始化和资源加载

在`PAL_SplashScreen()`函数中，首先会加载相关的资源和位图，其中包括了上方和下方的位图（`BITMAPNUM_SPLASH_UP`和`BITMAPNUM_SPLASH_DOWN`）以及仙鹤的精灵帧（`SPRITENUM_SPLASH_CRANE`）。

```
clpBitmapDown = VIDEO_CreateCompatibleSurface(gpScreen);
lpBitmapUp = VIDEO_CreateCompatibleSurface(gpScreen);

// 读取位图资源
PAL_MKFReadChunk(buf, 320 * 200, BITMAPNUM_SPLASH_UP, gpGlobals->f.fpFBP);
Decompress(buf, buf2, 320 * 200);
PAL_FBPBlitToSurface(buf2, lpBitmapUp);

PAL_MKFReadChunk(buf, 320 * 200, BITMAPNUM_SPLASH_DOWN, gpGlobals->f.fpFBP);
Decompress(buf, buf2, 320 * 200);
PAL_FBPBlitToSurface(buf2, lpBitmapDown);
```

这些资源通过`PAL_MKFReadChunk()`函数从文件中读取，并使用`Decompress()`解压，最后通过`PAL_FBPBlitToSurface()`将其绘制到屏幕上。

2. 仙鹤动画的实现

关键的动画效果是“仙鹤飞过”。从代码中可以看到，仙鹤的精灵是通过以下方式加载的：

```
cPAL_MKFReadChunk(buf, 32000, SPRITENUM_SPLASH_CRANE, gpGlobals->f.fpMGO);
Decompress(buf, lpSpriteCrane, 32000);
```

加载的仙鹤精灵帧保存在 `lpSpriteCrane` 中。动画是通过更新仙鹤精灵的帧来实现的。

3. 仙鹤的位置和运动

仙鹤的位置和运动是通过 `cranepos` 数组来控制的。每个仙鹤的初始位置（x，y）和当前帧（`cranepos[i][2]`）是随机生成的：

```
cfor (i = 0; i < 9; i++)
{
    cranepos[i][0] = RandomLong(300, 600);  // x坐标
    cranepos[i][1] = RandomLong(0, 80);     // y坐标
    cranepos[i][2] = RandomLong(0, 8);      // 动画帧
}
```

`cranepos[i][0]` 控制仙鹤的水平位置，`cranepos[i][1]` 控制其垂直位置，`cranepos[i][2]` 控制仙鹤的动画帧（即仙鹤精灵的当前帧）。

然后，通过以下代码更新每个仙鹤的位置并绘制到屏幕上：

```
cfor (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;  // 控制仙鹤的y轴位
置
    PAL_RLEBlitToSurface(lpFrame, gpScreen, PAL_XY(cranepos[i][0], cranepos[i]
[1]));  // 绘制仙鹤
    cranepos[i][0]--;  // 控制仙鹤向左移动
}
```

- `PAL_SpriteGetFrame()` 根据 `cranepos[i][2]` 提供的帧索引从仙鹤精灵中获取当前的帧。
- `cranepos[i][0]--` 控制仙鹤水平向左移动。
- `cranepos[i][1]` 可能会根据 `iImgPos` 变化（`iImgPos` 控制上下位图的滚动），从而控制仙鹤的垂直位置。

4. 动画帧的更新

`iCraneFrame` 变量控制着动画帧的变化，每一帧更新时，通过以下方式切换仙鹤的帧：

```
ccranepos[i][2] = (cranepos[i][2] + (iCraneFrame & 1)) % 8;
```

每一帧，`cranepos[i][2]` 会在 0 到 7 的范围内循环变化，这样仙鹤的精灵帧就会不断变化，从而产生动画效果。

5. 视觉效果

- **渐变效果：** 在 `dwTime < 15000` 的条件下，程序通过设置调色板的颜色变化，逐步显示画面：

```
cif (dwTime < 15000)
{
    for (i = 0; i < 256; i++)
    {
        rgCurrentPalette[i].r = (BYTE)(palette[i].r * dwTime / 15000);
        rgCurrentPalette[i].g = (BYTE)(palette[i].g * dwTime / 15000);
        rgCurrentPalette[i].b = (BYTE)(palette[i].b * dwTime / 15000);
    }
}
VIDEO_SetPalette(rgCurrentPalette);
```

这种渐变效果为动画的过渡提供了平滑的视觉体验。

- **位图上下滚动：** 通过 `iImgPos` 变量控制上下部分的位图滚动，从而呈现类似的画面滚动效果。

```
cif (iImgPos > 1) iImgPos--;
```

每一次更新画面，先从`mgo.mkf`中读取此帧的像素信息，用`SDL_BlitSurface()`等函数填充`Surface`的缓冲区，最后调用`SDL_UpdateRect()`更新画面。

`SDL_UpdateRect()`读取相应信息之后调用`NDL_DrawRect()`，`NDL_DrawRect()`通过系统调用打开显存设备文件，往显存里面写入画面数据。用户程序最终会调用`libos`里的`__syscall__()`函数发起系统调用，然后转移到`nanos`执行。`nanos`通过`fb_write()`函数处理这个写入显存的操作。它会调用`AM`提供的接口`io_write()`往设备寄存器`AM_GPU_FBDRAW`写入数据。这段指令在机器代码中被会被翻译成往`FB_ADDR`这个地址里写入数据。当`NEMU`执行到这个写指令时，会发现这个地址映射到显存，故并不实际地往里面写数据，而是调用SDL库把画面显示到屏幕上。到此完成了更新屏幕画面的全过程。