PA 3

201300035 方盛俊

1. 理解上下文结构体的前世今生 & 理解穿越时空的旅程

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?

前置工作: 初始化 `init_irq(void)`:

1. `init_irq(void)` 调用了 `cte_init(do_event)`, 将 `do_event()` 这个函数传入;
2. `cte_init()` 调用 `asm volatile("csrw mtvec, %0" : : "r"(__am_asm_trap))` 将 `__am_asm_trap()` 函数地址保存在 `mtvec` 中;
3. `cte_init()` 调用 `user_handler = handler` 将 `do_event` 保存在全局变量中, 以便后续回调;

正式工作: 调用 `yield()`:

1. `yield()` 调用了 `asm volatile("li a7, -1; ecall")`, 使用 `ecall` (设置异常号 `11`) 跳转到 `mtvec` 寄存器的 `__am_asm_trap()` 函数中;
2. `__am_asm_trap()` 使用 `addi sp, sp, -CONTEXT_SIZE` 在堆栈区初始化了 `CONTEXT_SIZE` 大小的上下文结构体 `c`, **这是上下文结构体生命周期的开端**;
3. `__am_asm_trap()` 使用 `MAP(REGS, PUSH)` 的宏展开式函数映射编程法, 类似于 `PUSH(REGS)` 这样, 将 `32` 个通用寄存器保存到了 `c` 中相应位置;
4. `__am_asm_trap()` 使用类似于 `csrr t0, mcause; STORE t0, OFFSET_CAUSE(sp)` 的汇编语句将 `mcause`, `mstatus` 和 `mepc` 三个寄存器保存到了 `c` 中相应位置;
5. `__am_asm_trap()` 将 `mstatus.MPRV` 置位, 以便通过 difftest;
6. `__am_asm_trap()` 使用 `mv a0, sp; jal __am_irq_handle` 将位于堆栈区的 `c` 上下文结构体保存到函数传参寄存器 `a0` 中, 作为函数参数调用并传给 `__am_irq_handle()` 函数;
7. `__am_irq_handle()` 通过 `c->mcause` 判别异常号, 并创建对应 `Event ev`, 调用 `user_handler(ev, c)`, 即调用上文提到的 `do_event(e, c)`;
8. `do_event()` 对异常或中断做完相应处理后, 返回到 `__am_irq_handle()` 中;
9. `__am_irq_handle()` 也做完了相应处理, 返回到 `__am_asm_trap()` 中;
10. `__am_asm_trap()` 使用类似于 `LOAD t1, OFFSET_STATUS(sp); csrw mstatus, t1` 的汇编语句将 `c` 中相应位置保存到 `mstatus` 和 `mepc` 两个寄存器中;

11. `__am_asm_trap()` 使用 `MAP(REGS, POP)` 将 `c` 中相应位置数据复原回 `32` 个通用寄存器中;
12. `__am_asm_trap()` 使用 `addi sp, sp, CONTEXT_SIZE` 将堆栈区复原, 相当于将 `c` 释放, **这是上下文结构体生命周期的结束**;
13. `__am_asm_trap()` 使用 `mret`, 将 `mtvec` 寄存器内保存的数据取出, 并跳转到该位置, 即回到了调用中断代码的 `yield()` 函数中;
14. `yield()` 处理完所有事情, 便返回了, 进而调用了 `panic("Should not reach here")`.

此外, 我还在 `trap.S` 的 `csrw mepc, t2` 指令前加入了 `addi t2, t2, 4`, 来实现自陷指令 `ecall` PC 加 4 的效果.

2. hello 程序是什么, 它从而何来, 要到哪里去

> 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?

1. 在 `navy-apps/tests/hello` 目录下经过 `make ISA=riscv32` 编译之后, 在 `build` 目录下生成了 `hello-riscv32` 文件, 这是一个 ELF 格式的可执行文件.
2. 我们将 `hello-riscv32` 文件复制到 `nanos-lite/build/ramdisk.img` 文件, 作为给 Nanos 使用的 "内存虚拟盘" `ramdisk` 加载.
3. Nanos 的 `resources.S` 内部通过 `ramdisk_start:; .incbin "build/ramdisk.img"; ramdisk_end:` 语法将 `ramdisk.img` 文件加载进内存里.
4. Nanos 在 `ramdisk.c` 文件中使用 `&ramdisk_start` 来获取已经加载进内存的 `ramdisk.img` 对应的内存地址.
5. `hello-riscv32` 即 `ramdisk` 被我们自己编写的 `loader()` 函数识别为一个 ELF 文件, 并按照约定加载到地址 `0x83000000` 中.
6. 第一条指令的地址通过 ELF 文件中 `elf.e_entry` 给出.
7. 为了调用它的第一条指令, 我们只需要将 `elf.e_entry` 作为一个函数地址进行调用即可, 就像 `((void (*)())entry)()` 这样调用.
8. hello 程序打印字符串的经历如下:
    1. `hello.c` 调用 `printf("Hello World from Navy-apps for the %dth time!\n", i ++);`
    2. libc 库中的 `printf()` 会将字符暂时放置到 `wbuf.c` 的缓冲区中, 当达到一定条件一会调用一次 `_write()` 函数进行输出.
    3. `_write()` 函数会触发中断 `_syscall_(SYS_write, fd, buf, count)`, 后者调用了 `ecall` 指令.

4. `ecall` 指令实际上跳转到了 `__am_asm_trap()` 函数, 封装好 `Context c` 后进一步调用 `__am_irq_handle(c)` 函数, 其封装为事件 `EVENT_SYSCALL` 后调用 `do_event(Event e, Context* c)`
5. `do_event(Event e, Context* c)` 确认是事件 `EVENT_SYSCALL` 后, 调用 `do_syscall(Context *c)`, 其中 `c->GPR1` 存储了系统调用号, `c->GPR2~4` 是三个参数, `c->GPRx` 用来存放返回值.
6. 对于 `SYS_write` 系统调用, 我们通过 `for (size_t i = 0; i < a[3]; ++i) putch(((char *) a[2])[i]);` 输出每一个字符.
7. AM 的 `putch()` 又调用了 NEMU 里的串口设备, 进行输出.
8. 最后由 NEMU 把字符输出到控制台.

3. 仙剑奇侠传究竟如何运行

1. 第一步是先渲染背景.

```
//
// Read the bitmaps
//
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);
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);
```

其中 `PAL_MKFReadChunk()` 中调用 `fseek()` 和 `fread()` 读取了 MKF 文件内容, 然后保存在 `buf` 里. 而 `fread()` 最后会在 Nanos 里调用 `memcpy(buf, &ramdisk_start + offset, len);` 从内存中读取对应的 MKF 文件.

通过这种方式初始化了好了结构体, 分布是 `lpBitmapUp`, `lpBitmapDown` 和标题 `lpBitmapTitle`.

1. 进入渲染的主循环.
2. 通过 `PAL_ProcessEvent();` 读取按键信息, 以便随时跳出.
3. 画仙鹤的主要部分.

```
//
// 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++;
```

通过 `PAL_SpriteGetFrame()` 获取仙鹤的像素, 然后通过 `PAL_RLEBlitToSurface()` 写入屏幕 `gpScreen` 中.

4. 画标题的主要部分.

```
//
// Draw the title...
//
if (PAL_RLEGetHeight(lpBitmapTitle) < iTitleHeight)
{
   //
   // HACKHACK
   //
   WORD w = lpBitmapTitle[2] | (lpBitmapTitle[3] << 8);
   w++;
   lpBitmapTitle[2] = (w & 0xFF);
   lpBitmapTitle[3] = (w >> 8);
}

PAL_RLEBlitToSurface(lpBitmapTitle, gpScreen, PAL_XY(255, 10));
```

5. 通过 `VIDEO_UpdateScreen(NULL)` 更新屏幕.
6. 进而调用 `SDL_UpdateRect()` 更新屏幕.
7. 进而调用 `NDL_DrawRect()` 更新屏幕.
8. 进而通过 `lseek(fp_fb, offset, SEEK_SET)` 和 `write(fp_fb, pixels + w * i, 4 * w)` 的方式写入 `/dev/fb` 设备中, 相当于一个系统调用.
9. Nanos 的 `size_t fb_write(const void *buf, size_t offset, size_t len)` 被调用.
10. AM 的 `io_write(AM_GPU_FBDRAW, x, y, (void *) buf, len / 4, 1, false);` 被调用.

11. AM 的 `__am_gpu_fbdraw(AM_GPU_FBDRAW_T *ctl)` 被调用, 然后写入 NEMU 对应的屏幕内存.
12. NEMU 的 `static inline void update_screen()` 函数被调用, 然后具体实现为

```
static inline void update_screen() {
  SDL_UpdateTexture(texture, NULL, vmem, SCREEN_W * sizeof(uint32_t));
  SDL_RenderClear(renderer);
  SDL_RenderCopy(renderer, texture, NULL, NULL);
  SDL_RenderPresent(renderer);
}
```