PA3 实验报告

翟笑晨

2024 年 12 月 27 日

目录

1 实验进度

我已完成 PA3 的所有阶段性任务，成功通过所有测试并能够正常运行 Bird，Nslider 以及 PAL 等 Apps 功能（libam 和声卡未实现）。系统能够正确展示批处理功能。

2 必答题

2.1 理解上下文结构体的前世今生

在 `__am_irq_handle()` 中有一个上下文结构指针 `c`，其指向的上下文结构究竟在哪里?

- 在 RISC-V 架构中，上下文结构 `Context` 包含通用寄存器组、控制状态寄存器（mcause，status，mepc）以及一个事件处理回调函数的指针，结构如下:

```
struct Context {
    uintptr_t gpr[NR_REGs];
    uintptr_t mcause, mstatus, mepc;
    void *pdir;
};
```

- 初始化 CTE 时，通过 `cte_init()` 函数将异常入口 `__am_asm_trap` 的地址赋值给 `mtvec` 寄存器。
- 异常触发时会跳转到 `__am_asm_trap`，将当前通用寄存器、控制状态寄存器等的值赋值给上下文。
- 其中:
  - `$ISA-nemu.h` 定义了上下文结构体，与架构相关。
  - `trap.S` 为上下文结构体的成员赋值，并作为入口参数传给异常处理函数。
  - NEMU 实现的新指令（如 `csrr` 等）在硬件指令层面支持了异常处理。

2.2 理解穿越时空的旅程

从 yieldtest 调用 yield() 开始，到返回的过程如下：

- 初始化 CTE 时，将 __am_asm_trap 地址赋值给 mtvec 寄存器。
- yield() 的汇编代码如下：

```
1 li a7, -1     // 将异常原因存入寄存器 a7
2 ecall         // 触发异常
3 ret           // 返回
```

- ecall 指令通过 isa_raise_intr() 将异常原因赋值给 mcause，并跳转到异常入口。
- 在 __am_asm_trap 中,保存上下文并将指针传递给 __am_irq_handle(),分发事件后返回。
- __am_asm_trap 恢复上下文并通过 mret 返回原程序, 地址为 mepc 中存储的值。

2.3 hello 程序是什么，它从何而来，要到哪里去

hello 程序从 nanos-lite 的 main 函数开始执行，完成以下初始化步骤：

- init_mm() 初始化虚拟内存；
- init_device() 初始化 IO 设备；
- init_irq() 初始化中断和异常处理机制；
- init_ramdisk() 加载磁盘；
- init_proc() 初始化需要加载的程序。

接着，在 proc.c 文件中，init_proc() 调用了 native_uload() 函数：

- 第一行调用 loader()，解析并加载 ramdisk.img 文件，即 hello.c 编译链接生成的 ELF 文件。

- loader() 识别 ELF 头，解析程序头表中的可加载段信息，加载代码段和数据段到指定内存位置，并返回 ehdr->e_entry（程序入口地址）。
- 最后，将入口地址强制转换为函数指针并跳转执行。

在进入客户程序后，首先执行 start.S，然后跳转到 crt0.c 中的 call_main() 函数，最终调用 hello.c 的 main 函数。

hello.c 的 write() 函数通过系统调用 _syscall_() 实现。在 _syscall_() 中，系统调用类型（如 SYS_write）被存入寄存器 a7，并通过 ecall 指令触发异常。触发异常后：

- NEMU 的 isa_raise_intr() 将 PC 设置为异常入口地址 __am_asm_trap。
- 在 __am_asm_trap 中，调用 __am_irq_handle() 进行事件分发，根据 a7 的值识别为 EVENT_SYSCALL。
- __am_irq_handle() 调用 do_event()，进一步调用 do_syscall()。
- 在 do_syscall() 中，识别为 SYS_write 后调用 fa_write()，最终通过 AM 层串口输出字符。

完成写操作后，返回值存入 c->GPRx，并逐层返回到 _syscall_() 函数的调用处，完成一次完整的写操作。

hello.c 的 main 函数通过一个循环不断调用 printf()，与 write() 类似，最终调用 _write() 实现打印。在堆区分配空间时，可能调用 _sbrk()，触发系统调用流程，不再赘述。

当 hello.c 执行完毕后，return 0 将值传递给 crt0.c 的 exit() 函数。通过封装，exit() 调用 _exit()，触发系统调用 SYS_exit，并通过相同的流程逐层返回，最终调用 halt()，结束程序。具体而言：

- halt() 在 AM 层相当于 nemu_trap() 宏，内联汇编为 ebreak 指令；
- 在 NEMU 中，ebreak 指令调用 set_nemu_state()，将 nemu_state.halt_ret 的值设为返回值（例如 0）。

最终，由于返回值为 0，nemu_state.halt_ret 设为 0，程序结束。

以上描述了 hello.c 的完整执行流程，包括其初始化、系统调用处理机制、字符输出逻辑及程序终止过程。

2.4 仙剑奇侠传的运行过程解析

仙剑奇侠传启动时，会通过 PAL_SplashScreen() 函数播放包含仙鹤飞行动画的启动画面。以下简述其从像素数据读取到屏幕更新的核心流程:

2.4.1 应用层的操作

在 main.c 文件中，像素数据通过 PAL_MKFReadChunk() 函数加载到指定内存位置。在动画播放过程中，VIDEO_CopySurface() 函数被调用，间接利用 SDL_BlitSurface() 和 SDL_UpdateRect() 实现图像复制与显示更新。

2.4.2 库函数的支持

SDL_BlitSurface() 使用 memcpy() 完成像素数据的内存复制,而 SDL_UpdateRect() 则调用了 NDL_DrawRect()，进一步依赖系统调用（如 write）将数据写入目标设备文件（如 /dev/fb）。

2.4.3 libos 的功能

在 libos 层，_write() 函数通过 _syscall_() 设置相关寄存器并触发系统调用，标志 SYS_write 用于指示写操作。

2.4.4 nanos-lite 的处理

通过系统调用进入 nanos-lite 的中断处理流程，do_syscall() 捕获写请求并定位到文件系统的 fs_write() 实现。若目标是帧缓冲设备，则调用 fb_write()，通过计算偏移量与屏幕坐标，将像素数据传递到硬件接口。

2.4.5 AM 层的硬件交互

在 AM 层, io_write() 负责将像素数据写入显存地址 (如 0xa1000000)。操作抽象为硬件缓冲区的写入，并最终通过指令发送给底层设备。

2.4.6 NEMU 的执行

在 NEMU 层，写指令由 MMIO 模块处理，通过更新显存与 VGA 控制寄存器 (如 vgactl_port_base)，完成屏幕内容的刷新，实现动画的动态呈

现。

通过上述层层协作，仙剑奇侠传的像素数据得以从存储文件流向显示屏幕。

3 实验过程与心得

3.1 中断/异常响应机制

- 实现 CSR 寄存器及相关指令（如 `ecall`, `mret`, `csrrw`）。
- 触发异常后保存上下文，跳转到异常入口并调用异常处理函数。
- 重组上下文结构体并在 `trap.S` 中调整压栈顺序。

3.2 批处理系统的思想

通过后台程序控制前台程序运行，我们实现多程序的自动调度。当用户程序结束执行之后我们就能够自动跳转到操作系统的代码继续执行，以实现 CPU 任务流之间的高度衔接和协同。

3.3 小错误大麻烦

之前在文件系统最后实现 bmp_test 的时候, 因为在 fs_write 中少实现了一个 fd=3 的判断, 导致 logo 标志根本无法加载到 NEMU 中。一度跑到 PA2 的 VGA 去 Debug。因此一定要严格按照手册和源代码来实现指定的功能，不然很可能存在隐形的 Bug!

4 致谢

最后非常感谢陈奕睿同学能够和我一起找 bug 的根源并且不断地测试纠错, 以及非常感谢助教的耐心和帮助.