PA3 实验报告

241840113 曾睿鸣

2025 年 12 月 13 日

摘要

本报告是我根据 PA3 实验指南 完成 PA3 的一个记录。

目录

1 实验进度

我完成了全部必做内容以及部分选做内容。

2 理解上下文结构体的前世今生

必做题 2.1. 解释 `__am_irq_handle()` 中参数 `Context *c` 的来源。该结构体位于何处？其各个成员分别在什么阶段、由哪段代码完成赋值？`$ISA-nemu.h`、`trap.S`、*NEMU* 中新增指令之间有何对应关系？

我的解答 2.1.

1. ***Context* 位于何处**

   由于 `cte.c` 中的 `cte_init()` 函数将 `mtvec` 寄存器的值设置为 `__am_asm_trap()` 的地址，因此在异常发生后，*CPU* 跳转至 `__am_asm_trap()`。该函数通过调整栈指针并连续写栈，在栈内存中构造出一个 `Context` 结构体实例。

   随后，当前栈指针作为参数传递给 `__am_irq_handle(Context *c)`，因此上下文结构体指针直接指向 `__am_asm_trap()` 执行期间的栈帧中创建的这个结构体实例。

2. ***Context* 各成员如何被赋值**

   直接看 `trap.S` 的内容：

   ```
   addi sp, sp, -CONTEXT_SIZE

   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)
   ```

   32 个通用寄存器、3 个 *CSR* 依次被压入栈中，依次完成 *Context* 结构体的赋值。

   因此，*Context* 的构造是汇编层面对内存的直接写入，*C* 代码只是按既定布局解释该内存。

3. ***NEMU*、新指令与 *AM* 的关系**

   在 *NEMU* 中实现 `ecall`，负责模拟硬件在异常发生时的行为，包括：

   - 将当前 *PC* 写入 `mepc`
   - 设置 `mcause`
   - 将 *PC* 跳转至 `mtvec`

   AM 的 `trap.S` 代码假设上述行为已经完成，并在此基础上保存现场并进入 *C* 级异常处理逻辑。两者通过 *CSR* 的语义形成明确分工。

3 理解穿越时空的旅程

必做题 3.1. 从用户程序调用 *yield()* 开始，到程序从 *yield()* 返回为止，详细说明软硬件各层如何协同完成该过程。

我的解答 3.1.

1. **异常机制初始化**

   程序启动后调用 *cte_init()*，完成两项关键初始化：

   - 将 *mtvec* 设置为 *__am_asm_trap*
   - 注册 *C* 级异常处理回调函数 *simple_trap*

2. ***yield* 触发异常**

   yield() 内部通过内联汇编执行：

   ```
   li a7, -1
   ecall
   ```

   其中 *a7* 用于传递异常编号，*ecall* 指令触发一次同步异常。

3. ***NEMU* 对 *ecall* 的处理**

   NEMU 在解释执行 *ecall* 时模拟硬件行为：

   - *mepc* 保存当前 *PC*
   - *mcause* 记录异常原因
   - *PC* 跳转至 *mtvec*

   随后开始执行 *__am_asm_trap*。

4. ***Context* 构造与事件分发**

   trap.S 保存现场并构造 *Context*，随后调用 *__am_irq_handle(c)*。

   该函数根据 *c->mcause* 构造 *Event*，并调用先前注册的 *simple_trap* 进行处理。

5. **返回执行**

   异常处理结束后，恢复寄存器并执行 *mret*，*PC* 回到 *mepc* 指向的位置，程序从 *yield()* 调用点继续执行。

4 hello 程序是什么, 它从而何来, 要到哪里去

必做题 4.1. 说明 *hello* 程序如何从 *ELF* 文件开始，被加载、执行并完成字符输出。

我的解答 4.1.

hello 程序在编译完成后被链接为一个 *ELF* 可执行文件，并被写入到系统使用的虚拟磁盘镜像 `ramdisk.img` 中。系统启动过程中，*Nanos-lite* 在执行到 `init_proc()` 时，会调用 `naive_uload()` 以加载第一个用户程序。

`naive_uload()` 内部进一步调用 `loader()`，由该函数负责解析并加载位于 `ramdisk.img` 中的 *hello* 程序。加载过程中，`loader()` 首先读取 *ELF* 文件头以获取程序的整体信息，随后依次遍历各个 *Program Header*。每个 *Program Header* 描述了一段需要加载的程序内容，包括该段在 *ELF* 文件中的偏移位置、目标内存地址以及需要加载的字节长度。根据这些信息，`loader()` 将 *hello* 程序的各个段拷贝到对应的内存位置，从而完成用户程序在内存中的布局。

ELF 文件头中的 `e_entry` 字段给出了程序的入口地址，即程序第一条指令所在的位置。该入口地址作为 `loader()` 的返回值返回给 `naive_uload()`，后者在获得入口地址后直接跳转至该地址，*CPU* 从而开始执行 *hello* 程序的第一条指令。

在程序运行过程中，*hello* 通过调用 `printf()` 输出字符串。格式化处理完成后，`printf()` 最终会调用标准输出对应的用户态接口 `_write()`。该函数按照系统调用约定，将相关参数放入指定寄存器中，并通过触发 `SYS_write` 类型的系统调用请求内核服务。

当程序执行到 `ecall` 指令时，控制流转入异常处理流程。系统在识别该异常为系统调用后，将其分发给 *Nanos-lite* 中注册的回调函数 `do_event()` 进行处理。`do_event()` 根据系统调用号判断当前请求为 `SYS_write`，随后调用 `sys_write()` 执行具体的写操作。该函数最终通过 `putch()` 将字符串中的字符逐个输出，从而完成 *hello* 程序的终端输出过程。

5 理解计算机系统

必做题 5.1. 解释 *PAL* 启动动画中像素数据如何从文件系统读取并最终显示到屏幕。

我的解答 5.1.

每一帧画面的更新过程始于用户程序对图像数据的读取。在 *PAL* 启动动画中，程序首先从数据文件 *mgo.mkf* 中读取当前帧对应的像素信息，并将这些像素数据通过 *SDL_BlitSurface()* 等函数写入到 *Surface* 结构体所管理的内存缓冲区中。该缓冲区作为用户态维护的中间图像存储区域，用于暂存即将显示的画面内容。

当一帧图像的数据准备完成后，程序调用 *SDL_UpdateRect()* 请求刷新屏幕指定区域。该函数在内部并不直接操作硬件，而是进一步调用 *NDL* 提供的 *NDL_DrawRect()* 接口，将待更新的矩形区域及其像素数据传递给底层系统。

NDL_DrawRect() 通过系统调用的方式访问显存设备文件，从而发起一次向显存写入画面数据的请求。具体而言，用户程序最终会调用 *libos* 中的 *_syscall_()* 函数构造系统调用参数，并触发异常机制，将控制权转移至 *Nanos-lite* 内核。

系统调用进入内核后，由 *Nanos-lite* 中的 *fb_write()* 函数负责处理该请求。该函数并不直接操作硬件，而是通过 *AM* 提供的抽象接口 *io_write()*，向设备寄存器 *AM_GPU_FBDRAW* 写入绘图相关的数据。

在机器级别，这一操作会被翻译为对固定内存地址 *FB_ADDR* 的写操作。*NEMU* 在执行该写指令时，会识别出该地址属于显存映射区域，因此不会真的向内存中写入数据，而是调用宿主机的 *SDL* 库接口，将对应的像素数据绘制到窗口中。

至此，一次从用户程序发起、经由系统调用、抽象设备接口以及 *NEMU* 模拟硬件，最终反映到实际屏幕显示的画面更新过程完成。