

# PA4实验报告

## 实验进度

我完成了PA4全部必做内容，没有完成任何选做内容。

## 遇到的问题及思考

- PA4的复杂程度相比前三个PA全面上升，很多隐藏的BUG也在机制的耦合下暴露出来。前三个PA中几乎没用到的各种trace和debug方法在PA4被大量使用。尽管有很多debug工具，发现导致BUG的具体原因仍不容易。
- 调了最久的一个错误是发现在运行内核程序（PCB保存到到 `PCB[0]`）时会覆盖用户程序的PCB（`PCB[1]`），导致用户程序PCB被改变，而单独加载内核程序或者用户程序时没有问题。我在nanos-lite和abstract-machine处的代码打印相关信息，发现在调用内核程序完毕，调用用户程序前会发生覆盖。考虑到内核栈安排在PCB结构的末尾，`PCB[0]` 的栈与 `PCB[1]` 的开头相邻，故怀疑是栈指针出错。利用PA1中完成的sdb，我定位到了“栈指针在 `PCB[1]` 结构体中且正在运行用户程序”时的pc，在那里逐条运行机器级指令排查，最后发现在加载结构体时跳转到的pc值不是mepc，而是mepc+4。在之前的实验中，我在ecall用mepc保存当前pc值，运行mret时跳转到mepc+4的位置。这只会导致entry处的一条指令不被运行，该指令调整了sp位置，故运行单个程序/sp不在重要数据附近时程序不会出错，因此一直未被发现。
- 不知为何总是应验的规律：错误越离奇，BUG越底层。（在找BUG上帮了很多忙）

## 必答题

- 分时多任务的具体过程
  - Nanos-lite：在页表中保存以下地址转换的信息：
    - 内核程序：初始化时将外设和内核代码所用到的虚拟地址映射到与之相等的物理地址。
    - 用户程序：加载程序时将程序及其栈区所占的虚拟页映射到各区域所在的物理页，后续通过 `mm_brk` 将堆区新用到的虚拟页映射到可用的物理页上。
  - AM： `cte` 中的 `kcontext` 与 `vme` 中的 `ucontext` 分别设置了内核与用户上下文的地址空间中的 `pdir`，从而加载上下文的时候可以加载与进程相关的页表项。
  - NEMU：通过检查satp寄存器最高位判断分页机制是否打开；通过 `isa_mmu_translate` 将程序访问的虚拟地址转换为数据实际所在的物理地址，并同时进行权限检查。
- 硬件中断：
  - Nanos-lite：在 `do_event()` 接收到 `EVENT_IRQ_TIMER` 的event后调用yield，在 `schedule()` 中切换进程
  - AM：利用 `__am_asm_trap()` 保存/恢复现场，在 `__am_irq_handle` 中调用 `user_handler` 进行处理
  - NEMU
    - 每10ms调用 `dev_raise_intr` 将INTR引脚设置为高电平
    - 在 `cpu_exec()` 中for循环的末尾添加轮询INTR引脚的代码，每次执行完一条指令就查看是否有硬件中断到来：

```

word_t intr = isa_query_intr();
if (intr != INTR_EMPTY) {
    cpu.pc = isa_raise_intr(intr, cpu.pc);
}

```

- 理解计算机系统

- 程序角度:** 程序声明了一个字符指针 `char *p = "abc";`, 这意味着 `p` 指向一个字符串常量。然后, 程序尝试修改这个只读字符串常量的第一个字符, 即 `p[0] = 'A';`。由于字符串常量是只读的, 这个操作触发了段错误。
- 编译器角度:** 编译器在编译过程中会将字符串常量存储在只读段中, 以确保其只读属性。对只读段的写入操作会被编译器捕获并引发错误。
- 链接器角度:** 链接器负责将编译后的模块组装成可执行文件。只读属性通常在链接阶段被设置, 以确保只读段的内容在运行时不被修改。
- 运行时环境角度:** 在运行时, 操作系统加载可执行文件到内存, 并确保只读段的内容不可写。当程序尝试写入只读段时, 操作系统会检测到这个违规行为并引发段错误。
- 操作系统角度:** 操作系统负责内存管理, 包括对只读段的保护。当程序执行时, 操作系统会检查对内存的访问是否合法, 如果试图写入只读段, 则会触发段错误。
- 硬件角度:** CPU在执行指令时, 会检查内存访问的合法性。当发现对只读内存的写入时, 硬件会产生一个异常, 通知操作系统处理这个错误。

证明我的想法可以在编译时使用AddressSanitizer检测访问错误。

- 展示你的计算机系统

以下图片为实际运行效果。







