PA4 中的中断处理与真实硬件与操作系统的中断处理有哪些相同点与不同点？

不同点：

1.ring3 未实现

由于 NEMU 并没有实现 ring 3,Nanos-lite 也对用户进程作了一些简化，我们让 Nanos-lite 在加载用户程序后通过函数调用跳转到用户程序中执行.事实上,这并不是一个合理的方式,从安全的角度来说,高特权级的代码是不能直接跳转到低特权级的代码中执行的,真实硬件的保护机制甚至会抛出异常来阻止这种情况的发生.

合理的做法是,当操作系统初始化工作结束之后,就会通过自陷指令触发一次上下文切换,切换到第一个用户程序中来执行，真实的操作系统就是这样做的.

2._umake 的功能简化及参数环境变量

_umake()是专门用来创建用户进程的现场的,但由于 NEMU 并没有实现 ring3,Nanos-lite 也对用户进程作了一些简化,因此目前_umake()只需要实现以下功能:在 ustack 的底部初始化一个以 entry 为返回地址的陷阱帧.p 是用户进程的虚拟地址空间,在简化之后,_umake()不需要使用它.argv 和 envp 分别是用户进程的 main()函数参数和环境变量,目前 Nanos-lite 暂不支持,因此我们可以忽略它们.

相同点：

1.虚拟内存及分页机制

我们的需求:我们需要在让程序认为自己在某个固定的内存位置的同时,把程序加载到不同的内存位置去执行.

为了让这个问题的肯定回答成为可能,虚拟内存的概念就诞生了.所谓虚拟内存,就是在真正的内存(也叫物理内存)之上的一层专门给程序使用的抽象.有了虚拟内存之后,程序只需要认为自己运行在虚拟地址上就可以了,真正运行的时候,才把虚拟地址映射到物理地址.这样,我们只要把程序链接到一个固定的虚拟地址,加载程序的时候把它们加载到不同的物理地址,并维护好虚拟地址到物理地址的映射关系,就可以实现我们那个看似不可能的需求了!

绝大部分多任务操作系统就是这样做的。在 NEMU 中实现分页机制，，我们应该让用户程序运行在操作系统为其分配的虚拟地址空间之上.为此,我们需要对工程 作一些变动.首先需要将 navy-apps/Makefile.compile 中的链接地址 -Ttext 参数改为 0x8048000,这是为了避免用户程序的虚拟地址空间与内核相互重叠,从而产生非预期的错误.同样的,nanos-lite/src/loader.c 中的 DEFAULT_ENTRY 也需要作相应的修改.这时,"虚拟地址作为物理地址的抽象"这一好处已经体现出来了:原则上用户程序可以运行在任意的虚拟地址,不受物理内存容量的限制.我们让用户程序的代码从 0x8048000 附近开始,

这个地址已经超过了物理地址的最大值(NEMU 提供的物理内存是

128MB),但分页机制保证了程序能够正确运行.

2.上下文切换

我们需要一个指针 tf 来记录陷阱帧的位置,当想要找到别的程序的陷阱帧的时候,只要寻找这个程序相关的 tf 指针即可.

为了方便对进程进行管理,操作系统使用一种叫进程控制块(PCB,processcontrolblock)的数据结构,为每一个进程维护一个 PCB.Nanos-lite 的框架代码中已经定义了我们所需要使用的 PCB 结构(在 nanos-lite/include/proc.h 中定义)

Nanos-lite 使用一个联合体来把其它信息放置在进程堆栈的底部.代码为每一个进程分配了一个 32KB 的堆栈,

已经足够使用了,不会出现栈溢出导致 PCB 中的其它信息被覆盖的情况.在进行上下文切换的时候,只需要把 PCB 中的 tf 指针返回给 ASYE 的 irq_handle()函数即可,剩余部分的代码会根据上下文信息恢复现场.在 GNU/Linux 中,进程控制块是通过 task_struct 结构来定义的。