ICS PA4 Report

集成电路 211870293 李居奇

实验进度

- [x] 完成 Nanos-lite 上的进程上下文切换以及参数传递
- [x] 完成在分页机制上运行 Nanos-lite 和 pal
- [x] 完成硬件中断对上下文切换的切换，实现抢占多任务

实验必答题 & 思考题

Question 1

> 请结合代码, 解释分页机制和硬件中断是如何支撑仙剑奇侠传和hello程序在我们的计算机系统 (Nanos-lite, AM, NEMU) 中分时运行的.

NEMU 的硬件中断只有时钟中断，时钟中断的开启时间是在上下文切换后，这样做可以防止过早开启影响 nanos-lite 程序对 `sp` 的设置等初始化操作. 在每一条指令执行结束时，nemu 的 `cpu_exec()` 中都会调用 `timer_interrupt()` 函数来检查 CPU 的 INTR 引脚有没有被拉高。如果被拉高了说明有时钟中断请求，此时 CPU 会将 `mstatus` 寄存器的 MIE 保存并清 0，防止在中断或者异常处理时再次被打断（中断嵌套），然后进入 nanos-lite 的中断/异常处理程序 `trap.S`. 当然，清 0 的操作在异常处理的时候也会这样做，同样是为了防止异常处理函数被时钟中断打断.

```
// 关闭中断使能
// 将 mstatus.MIE 保存到 mstatus.MPIE 中, 然后将 mstatus.MIE 位置为 0
void reset_intr() {
  cpu.csr[_mstatus] = (cpu.csr[_mstatus] & ~(0x1 << MPIE_OFFSET)) |
(((cpu.csr[_mstatus] >> MIE_OFFSET) & 0x1) << MPIE_OFFSET) ;
  cpu.csr[_mstatus] &= ~(1 << MIE_OFFSET);
}
```

```
// 开启中断使能
// 将 mstatus.MPIE 还原到 mstatus.MIE 中，然后将 mstatus.MPIE 位置为 1
void set_intr() {
  cpu.csr[_mstatus] = (cpu.csr[_mstatus] & ~(0x1 << MIE_OFFSET)) |
(((cpu.csr[_mstatus] >> MPIE_OFFSET) & 0x1) << MIE_OFFSET);
  cpu.csr[_mstatus] |= (1 << MPIE_OFFSET);
}
```

在 `trap.S` 中，nanos-lite 首先会将 sp 指针切换到 hello 用户进程的内核栈，在内核栈上保存 hello 进程的上下文，之后进入 `__am_irq_handle()` 函数。在 `__am_irq_handle()` 函数中，nanos-lite 发现 `mcause` 寄存器中的事件编号是 `IRQ_TIMER`，于是标记该事件为时钟中断，调用 `do_event()` 函数。`do_event()` 函数中调用了 `shedule()` 函数，`schedule()` 函数将 `current->cp` 指向刚才保存在 `hello` 内核栈上的 `hello` 上下文，将 `current` 指针指向 pal 进程的 pcb，并返回 pal 进程内核栈上的上下文结构体.

```
Context* __am_irq_handle(Context *c) {
  if (user_handler) {
    Event ev = {0};
    // 中断
    if(c->mcause == IRQ_TIMER) {
        ev.event = EVENT_IRQ_TIMER;
    }
    // 异常
    else if(c->mcause == 0xb) {
        switch (c->GPR1) {
        case 0xffffffff: ev.event = EVENT_YIELD; c->mepc = c->mepc + 4;
break;
        default: {
          if(c->GPR1 >= 0 && c->GPR1 <= 19) {
            ev.event = EVENT_SYSCALL;
            c->mepc = c->mepc + 4;
          }
          else {
            ev.event = EVENT_ERROR;
            c->mepc = c->mepc + 4;
          }
          break;
        }
      }
    }
    else {assert(0);}
    c = user_handler(ev, c);
    assert(c != NULL);
  }
  // 切换上下文之后的 context
  // 不一定是之前的 context 指针
  return c;
}
```

回到 `trap.S` 中以后，nanos-lite 会恢复 pal 进程的上下文，并且根据 `c->np` 将栈指针从 pal 进程的内核栈移到 pal 进程的用户栈。执行完 `mret` 指令之后，nanos-lite 就切换到了 pal 的代码继续执行，最后`mret` 指令会将`mstatus` 的 MIE 位重新打开，以接收中断

Question 2

> 尝试在 Linux 中编写并运行以下程序:

```
int main() {
 char *p = "abc";
 p[0] = 'A';
 return 0;
}

> Segmentation fault (core dumped)
```

> 你会看到程序因为往只读字符串进行写入而触发了段错误. 请你根据学习的知识和工具, 从程序, 编译器, 链接器, 运行时环境, 操作系统和硬件等视角分析"字符串的写保护机制是如何实现的". 换句话说, 上述程序在执行 `p[0] = 'A'` 的时候, 计算机系统究竟发生了什么而引发段错误? 计算机系统又是如何保证段错误会发生? 如何使用合适的工具来证明你的想法?

首先，假设计算机是分页的，在汇编程序中字符串的地址是虚拟地址，需要由硬件中的 MMU 根据虚拟页表将虚拟地址转化成物理地址，转化时会检查当前进程的权限和这一虚拟内存区域的权限标志位，由于 “abc” 是只读数据，所以写操作引发了访存越界. 之后 MMU 会修改一些寄存器的值，然后转到操作系统的异常处理程序. 操作系统会向用户进程发送一个 SIGSEGV 信号，表示发生了段错误，用户进程收到 SIGSEGV 信号后会立刻 kill 掉当前进程，同时生成一个核心文件记录调试信息.

```
0000000000001129 <main>:
    1129:   f3 0f 1e fa             endbr64
    112d:   55                      push   %rbp
    112e:   48 89 e5                mov    %rsp,%rbp
    1131:   48 8d 05 cc 0e 00 00    lea    0xecc(%rip),%rax
    1138:   48 89 45 f8             mov    %rax,-0x8(%rbp)
    113c:   48 8b 45 f8             mov    -0x8(%rbp),%rax
    1140:   c6 00 41                movb   $0x41,(%rax)       #p[0] = 'A'
    1143:   b8 00 00 00 00          mov    $0x0,%eax
    1148:   5d                      pop    %rbp
    1149:   c3                      ret
```

Question 3

> 如果不同的进程共享同一个栈空间, 会发生什么呢? 尝试把 `hello_fun()` 换成 Navy 中的hello: 你发现了什么问题? 为什么会这样?

不同进程在上下文切换的时候会覆盖掉上一个进程在栈中的内容，从而导致程序数据出现异常，把 `hello_fun()` 换成 Navy 中的 hello，即同时运行了 pal 和 hello 两个进程，导致段错误.

Question 4

> 如果不同的进程共享同一个栈空间, 会发生什么呢? 尝试把 `hello_fun()` 换成 Navy 中的hello: 你发现了什么问题? 为什么会这样?

不同进程在上下文切换的时候会覆盖掉上一个进程在栈中的内容，从而导致程序数据出现异常，把 `hello_fun()` 换成 Navy 中的 hello，即同时运行了 pal 和 hello 两个进程，导致段错误.

Question 5

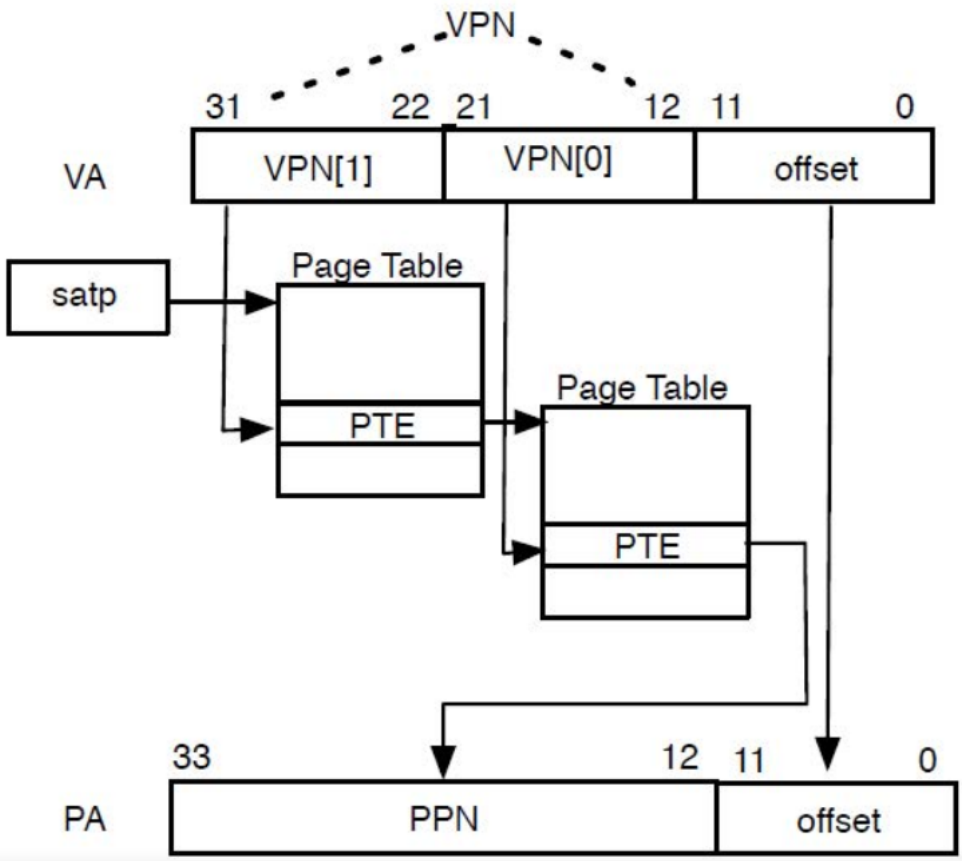
> 可以在用户栈里面创建用户进程上下文吗? ucontext()的行为是在内核栈 kstack 中创建用户进程上下文. 我们是否可以对ucontext()的行为进行修改, 让它在用户栈上创建用户进程上下文? 为什么?

个人觉得是可以的，只需要在传参的时候注意不要把参数写到栈顶，给一个 Context 的 offset 即可，笔者之前在实现用户进程的上下文切换时就不小心把上下文放在了用户栈而非内核栈，在传参时发生错误才意识到之前的实现有问题😭.

Question 6

> 为什么不采用一级页表? 或者说采用一级页表会有什么缺点?

缺点就是浪费内存，因为一级页表是一对一映射，所以是连续没有有空洞的，如果只用一级页表，则每个进程都要固定用 4MB 的空间来存储页表项，而采用二级页表(多级页表)，由于不是一对一连续映射，可以允许大量空洞的存在，只考虑第一级页表的话可以将空间压缩到最小 4kB 大小.



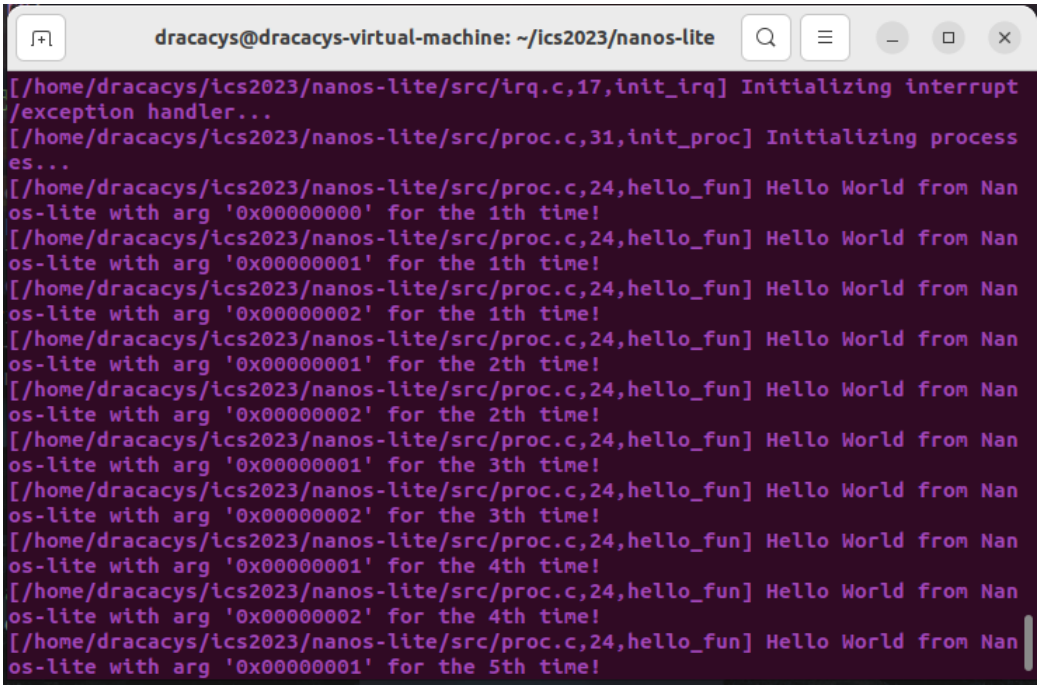
实验结果

内核线程的切换和参数传递

在 Nanos-lite 中切换内核线程上下文只需要关注下面两个函数：

- `context_kload()` 函数：调用 `kcontext()` 来创建上下文, 并把返回的指针记录到 PCB 的 `cp` 中
- `schedule()` 函数：返回新的上下文指针

注意传参的预定是用 `a0` 寄存器，其实现过程比较简单，让两个 hello 内核线程相互切换，传入不同的输入参数，得到的输出结果如下图所示：

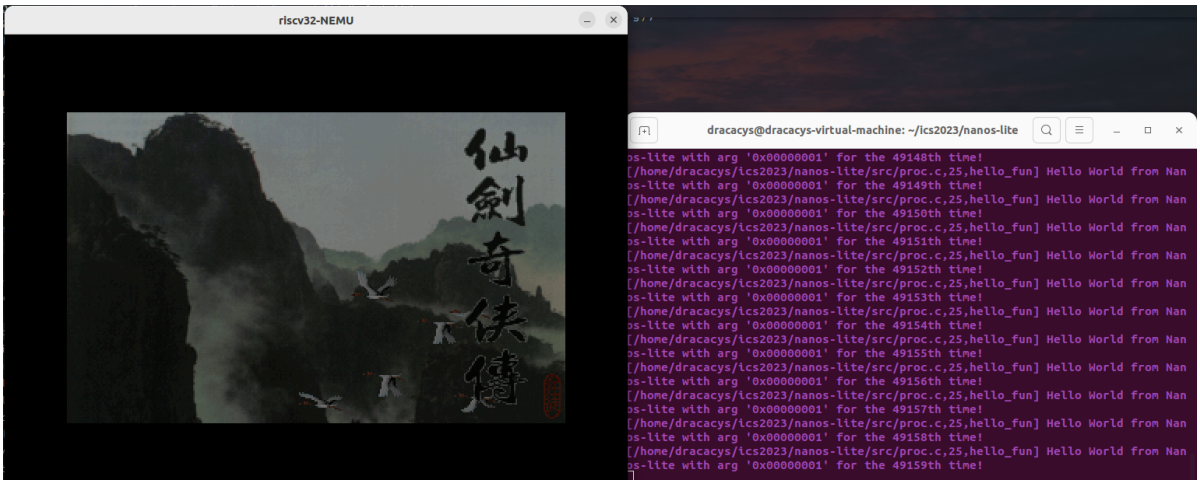


用户进程的切换和参数传递

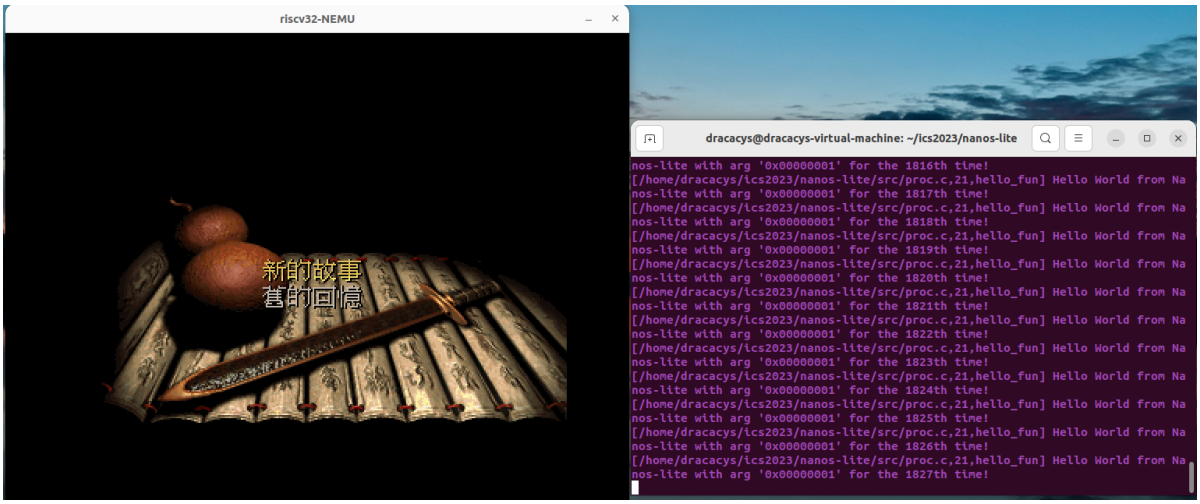
用户进程的存放位置和内核线程略有不同，笔者将 PCB 的 stack 看成内核栈，heap 堆区看作用户栈，整理关系如下表所示：

存放地址/进程	内核线程	用户进程
上下文(Context)地址	内核栈区(PCB.stack)	内核栈区(PCB.stack)
运行数据存放地址	内核栈区(PCB.stack)	用户栈区(heap)

对于用户进程，由于存放地址的差异，我们需要在内核栈恢复上下文后，将 `sp` 修正到用户栈区. 同时，用户进程的参数需要提前存放到用户栈上的指定位置，当程序开始执行后才能读取并解析参数. 如果没有完成传参和优先级调度，会发现仙剑的加载真的是相当之慢，在输出了万次以上的 hello 之后终于刷新出了游戏的开机界面：

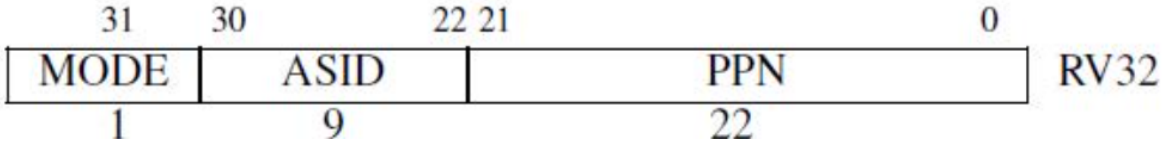


完成该部分内容后得到的效果如下图所示，可以发现在加载运行仙剑奇侠传进程的同时，线程 hello 正在同时输出，并且仙剑奇侠传跳过了开头部分，直接进入了游戏界面，hello 的执行次数减少了很多：

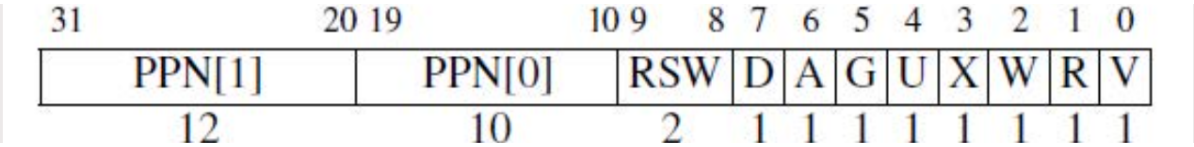


在分页机制上运行Nanos-lite

从这里开始整个 PA 的难度就拉满了😭，笔者作为选修 ICS 的弱鸡，没有时间完成在分页机制上运行用户进程，但能够在分页机制上运行 Nanos-lite 和仙剑奇侠传. 先考察 RV32 的 `sapt` 寄存器如下图所示，我们只用关注物理地址 PPN 和 MODE 位，前者决定了第一级页表的物理地址，而后者决定了是否需要进行地址转化.



而 RV32 规定页表项如下图所示，考虑到 Nanos 和 NEMU 控制权限相当麻烦，笔者并没有严格遵守权限位的读写和识别，对于任何虚拟页表项，所有用户程序均有权利读写和执行.



至此，我们可以根据 RV32 二级页表的结构编写物理 ---> 虚拟映射的 `map()` 函数，以及 虚拟---> 物理 `isa_mmu_translate()` 函数，这两个函数的过程互为逆过程，写之前还是比较晕，写完后发现很好理解：

```c
void map(AddrSpace *as, void *va, void *pa, int prot) {
  // just give priority and type a shit
  // 各个地址提取
  uint32_t PPN = PA_PPN((uint32_t)pa);
  uint32_t VPN_1 = VA_VPN_1((uint32_t)va);
  uint32_t VPN_2 = VA_VPN_2((uint32_t)va);
  // 基地址
  PTE *VPN_1_BASE = as->ptr;
  PTE *VPN_2_BASE = NULL;
  // 一级页表页表项的地址为空，则创建页表项
  if (!(VPN_1_BASE[VPN_1])) {
    // 设置二级页表的基地址，指向一个物理页面，专门存储页表
    VPN_1_BASE[VPN_1] = (PTE)pgalloc_usr(PGSIZE);
  }
  // 第二页表基地址和目标地址
  VPN_2_BASE = (PTE *)(VPN_1_BASE[VPN_1]);
  // 将物理页号填写到二级页表的页表项中，低 12 位是标志位
```

```
  VPN_2_BASE[VPN_2] = (PPN << 12) | 0xF;
  return;
}
```

```
paddr_t isa_mmu_translate(vaddr_t vaddr, int len, int type) {
  // just give priority and type a shit
  uint32_t VPN_1 = VA_VPN_1((uint32_t)vaddr);
  uint32_t VPN_2 = VA_VPN_2((uint32_t)vaddr);
  // 根据 satp 取出一级页表基地址
  paddr_t *VPB_1 = (paddr_t *)guest_to_host((paddr_t)(cpu.csr[_satp] << 12));
  assert(VPB_1 != NULL);
  // 二级页表基地址
  word_t *VPB_2 = (word_t *)guest_to_host(VPB_1[VPN_1]);
  assert(VPB_2 != NULL);
  // 物理页号 + 页内偏移合成物理地址
  paddr_t paddr = (paddr_t)((VPB_2[VPN_2] & (~0xfff)) | PA_OFFSET((uint32_t)vaddr));
  assert(paddr == vaddr);
  return paddr;
}
```

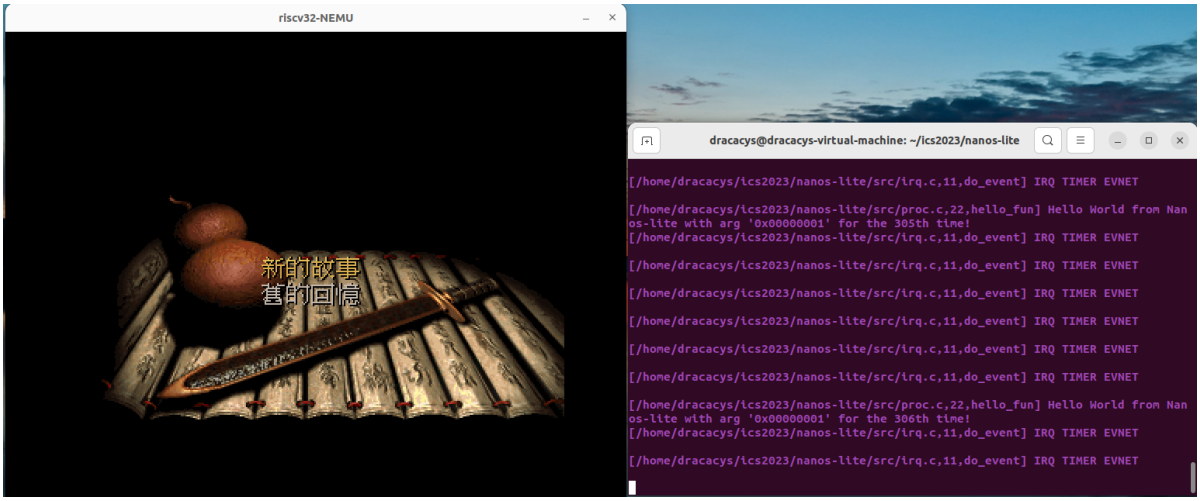
完成 AM 和 NEMU 的分页机制后还需修改 NEMU 中的 `vaddr_read/write()`，调用 MMU 对虚拟地址进行转化，而不是直接传给 `paddr_read/write()`，全部实现后程序在虚拟地址上可以照常运行仙剑奇侠传以及内核线程，并通过 OJ 的 easy_test 测试.

硬件中断和抢占多任务

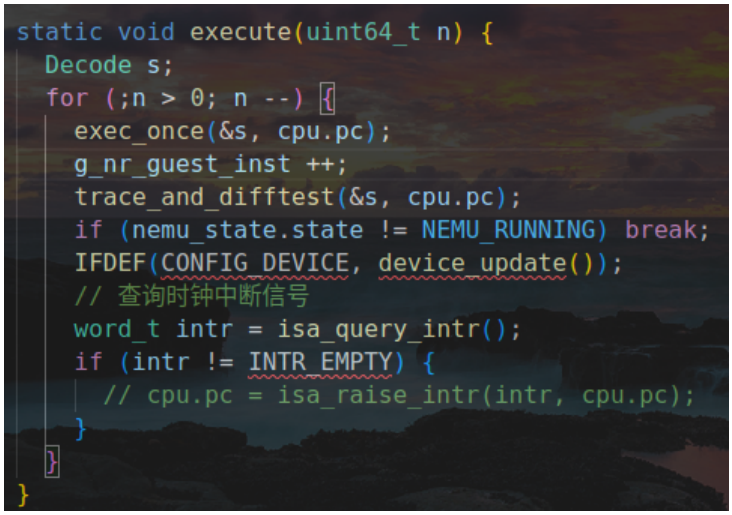
根据讲义内容，我们可以大致推断出中断控制流的执行过程：

> 上下文恢复后（mret）开启中断 -----> 定时器 1ms 产生一次中断 -----> 每次取指执行查看中断状态 -----> 关闭中断- ----> 异常处理函数 ------> 强制上下文切换

为了和之前的系统调用和 yield 区分，中断信号设置寄存器 `mcause` 的值为 `0x80000007`，同时为了保证避免中断套用，在异常处理阶段需要关闭中断信号处理的功能，其余的上下文切换过程和 yield 过程类似. 实现完成后去掉设备端口的 yield 仍然能够完成内核线程和用户进程的上下文切换，如下图所示，每次进行硬件中断后会输出一条 Log，这样能够体现其发挥的作用：



不过非常遗憾的是，笔者在加上中断功能后无法通过 OJ 的 easytest，报错为段错误，而本地运行始终功能正常，目前尚未发现无法通过的原因😭，在最终提交的版本中为了通过 easytest 测试，只能注释掉中断的入口函数，如果 TA 尝试验证中断功能的正确性，请取消注释如下图的 `isa_raise_intr()` 再运行程序：



实验思路

以下部分均是处于种种原因而**未完全实现**的内容，基本思路是有的，但 bug 还未能解决，等有空的时候做二周目再说吧：

运行 Busybox

这个功能的本质是要实现带参数的 `execve()`，现在我们假设现有一个进程 A 调用这个函数，试图切换到进程 B，我们要在这里做如下操作：

- 调用 `context_uload()` 函数，加载 B 进程的上下文到内核栈，同时让操作系统分配一个新的页，将 B 进程的参数压到这里的用户栈上，避免覆盖进程 A 的栈.
- 调用 `switch_boot_pcb()` 修改当前的 current 指针，为的是在之后的 yield 中保存当前 A 进程的上下文到 `pcb_boot`，如果跳过这一步，`yield()` 将会把上下文保存到用户栈上，不利于恢复
- 调用 `yield()`,切换到进程 B，在进程 B 结束后程序默认恢复 `pcb_boot` 的上下文

在分页机制上运行用户进程

这里也主要有两个修改点，对于 `context_uload` 函数：

- 函数开头需要调用 `protect`，生成页目录，用户地址空间，并建立内核空间的映射
- `ucontext` 保存第一步的指针到 `Context` 里，把 `new_page` 得到的 32KB 与 `USER_SPACE` 的高地址 32KB 映射, 因为分页机制已经开启, 所以我们应该对 pa 直接压栈, 然后把 pa 转换为 va, 作为用户进程的 `sp`

对于 `loader()` 函数：

- 调用 `new_page()` 申请一页空闲的物理页，通过 `map()` 把这一物理页映射到用户进程的虚拟地址空间中.
- 按照之前的加载程序操作，从 ELF 文件中读入一页的内容到这一物理页中

Long way to go（2）

相比前几个 PA，PA4 的必做部分仅完成了比较基础的一部分，有难度的内容大多涉及 OS 方面的内容，本人的偏硬件的基础实在已经无法解决了... 不过有空还是值得在二周目了解一下的😄. PA 这条支线就暂时告一段落了，接下来就继续一生一芯主线吧，现在 NPC 中的 CPU 已经成功替代 NEMU 接入 AM 并运行 mario 了呢.