说明

已经完成PA4所有必做题。运行时，pcb[0]为`hello`内核程序，其余三个均为nterm，可以从nterm进入其他用户程序，包括bird,pal,nslider等。

必答题

分时多任务的具体过程 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.
下面是在`common.h`中定义了TIME_SHARING的情况，此时只支持pcb在0和1之间切换

1. 一开始假设在用户程序运行
2. 当CPU接收到硬件时钟中断后，进入操作系统的时钟中断处理 (NEMU)
3. 时钟中断处理时，将`switch_to`设为0，但在TIME_SHARING情况下这个修改无用，因为schedule还会去修改`switch_to`的值 (Nanos-lite)
4. 进入`schedule()`，判断`current`是1，设定`switch_to`为0，故当前上下文指针切换到pcb[0] (Nanos-lite)
5. 返回到`trap.S`中，会根据切换到的上下文进行恢复，此时就恢复到了hello程序 (AM)
6. 如果这个程序是hello，则没有切换栈指针，否则要切换回用户的栈，具体在`trap.S`中有详细注释 (AM)
7. 切换到了hello，输出一句话后马上调用`yield()` (Nanos-lite)
8. `yield()`导致再次进入`schedule()`，判断`current`是0，设定`switch_to`为1，当前上下文指针切换到pcb[1] (Nanos-lite)
9. 然后返回到`trap.S`中，会根据切换到的上下文进行恢复，此时就恢复到了用户程序 (AM)

关于分页，在`trap.S`中退出时，satp也已经通过上下文恢复而切换到了相应的正确值，根据它提供的页目录指针，NEMU会进行对应的读取指令和数据操作

下面是在common.h中取消定义了TIME_SHARING的情况，此时就是PA最后一个必答题要求完成的展示

1. 一开始假设在用户程序1运行
2. 当CPU接收到硬件时钟中断后，进入操作系统的时钟中断处理 (NEMU)
3. 时钟中断处理时，将`switch_to`设为0 (Nanos-lite)
4. 进入`schedule()`，根据`switch_to=0`，当前上下文指针切换到pcb[0] (Nanos-lite)
5. 返回到`trap.S`中，会根据切换到的上下文进行恢复，此时就恢复到了hello程序 (AM)
6. 如果这个程序是hello，则没有切换栈指针，否则要切换回用户的栈，具体在`trap.S`中有详细注释 (AM)

7. 切换到了hello，输出一句话后马上调用`yield()`，并将`switch_to`设为`fg_pcb`(此时=1) (Nanos-lite)
8. `yield()`导致再次进入`schedule()`，根据`switch_to=1`，切换上下文指针为pcb[1] (Nanos-lite)
9. 返回到`trap.S`中，会根据切换到的上下文进行恢复，此时就恢复到了用户程序1 (AM)
10. 在用户程序1运行时，如果在event中接收到F1/F2/F3，则将`switch_to`设为1/2/3，`fg_pcb`同样如此 (Nanos-lite)
11. 进入`yield()`，然后`schedule()`根据`switch_to`设定切换到的用户程序

理解计算机系统 尝试在Linux中编写并运行以下程序:

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

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?

程序：字符串"abc"是只读的一个数据
编译器：char *p = "abc"这一句中，将p指向了某个地址，但此时还没有链接，因此后一句编译时不知道p[0]是只读的
链接器：将p指向了一个只读数据段的地址
运行时环境：不知道p是只读的，照常运行指令
操作系统和硬件：当执行指令时，尝试写入数据，但是写入的地址会先进行检查，发现这个段的段标记符中是没有写入权限的，因此发生段错误，进入中断处理。
因此，主要的发生错误原因是写入数据前，会检查写入数据的段标记符的相关权限。

使用robjdump读取编译后的文件：

```
test.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # f <main+0xf>
   f:   48 89 45 f8             mov    %rax,-0x8(%rbp)
  13:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  17:   c6 00 41                movb   $0x41,(%rax)
  1a:   b8 00 00 00 00          mov    $0x0,%eax
  1f:   5d                      pop    %rbp
  20:   c3                      ret
```

此处的rax赋值地址还没有写入
使用objdump读取最终生成的可执行文件，其中main函数部分如下：

```
    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        # 2004 <_IO_s
    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)
    1143:       b8 00 00 00 00          mov    $0x0,%eax
    1148:       5d                      pop    %rbp
    1149:       c3                      ret
```

此处的地址为2004，然后通过readelf读取节头信息，其中某一个节为：

```
       0000000000000008  0000000000000000   A       0     0     4
```

从这里可以看出，对应的数据是在.rodata节的。
使用objdump读取段头，其中有一个头为：

```
 LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                0x00000000000000cc 0x00000000000000cc  R      0x1000
```

从这里的权限只有R看出确实是只读的。
为了验证2004处确实是"abc"，我们使用objdump读取段内容，其中有一段为：

```
Contents of section .rodata:
 2000 01000200 61626300                    ....abc.
```

可见2004处确实是"abc"