1(1)

Chapter5-简答题1(1)

- **题目：**

stride 算法原理非常简单，但是有一个比较大的问题。例如两个 pass = 10 的进程，使用 8bit 无符号整形储存 stride， p1.stride = 255, p2.stride = 250，在 p2 执行一个时间片后，理论上下一次应该 p1 执行。

实际情况是轮到 p1 执行吗？为什么？

- **答案：**

不是。还是轮到p2。

因为stride算法会选择当前运行时间最短（即stride最小）的进程进行调度，但因为p2执行完之后，250+10=260，溢出了8bit所能表示的最大范围（255），从而变成了4。因此还是p2执行。

1(2)

Chapter5-简答题1(2)

- **题目：**

*我们之前要求进程优先级 >= 2 其实就是为了解决这个问题。可以证明，**在不考虑溢出的情况下**，在进程优先级全部 >= 2 的情况下，如果严格按照算法执行，那么 STRIDE_MAX – STRIDE_MIN <= BigStride / 2。*

为什么？尝试简单说明（不要求严格证明）。

- **答案：**

因为假设STRIDE_MAX =BigStride/x, STRIDE_MIN=BigStride/y, 其中x与y分别代表最大和最小进程对应的优先级。

那么STRIDE_MAX – STRIDE_MIN=BigStride/(1/x-1/y) < BigStride/2。得证。

2

- 已知以上结论，**考虑溢出的情况下**，可以为 Stride 设计特别的比较器，让 BinaryHeap 的 pop 方法能返回真正最小的 Stride。补全下列代码中的 `partial_cmp` 函数，假设两个 Stride 永远不会相等。

```
use core::cmp::Ordering;

struct Stride(u64);

impl PartialOrd for Stride {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        // ...
    }
}
```

```
10
11 impl PartialEq for Stride {
12     fn eq(&self, other: &Self) -> bool {
13         false
14     }
15 }
16 TIPS: 使用 8 bits 存储 stride, BigStride = 255, 则: (125 < 255) == false,
   (129 < 255) == true.
```

Chapter5-Lab3

- **题目：**

  实现一个完全 DIY 的系统调用 spawn。

  功能：新建子进程，使其执行目标程序。

  实现一种带优先级的调度算法：stride 调度算法。

Chapter5-Lab3

- **大体思路：**

  首先是迁移通过以前的测试

  在lab1中，为了实现get_task_info功能，我们添加了task_info_inner结构，但是lab3代码的框架里已经实现了这一部分，所以只需将之前的解构迁移进去即可。

  其余结构大致如前，搬运到对应的地方即可。

Chapter5-Lab3

- **大体思路：**

  接下来实现spawn。

  大致可以分为几步：

  1. 新建 TCB，数据是通过解析.elf文件中的数据计算得到的

  3.将新的 TCB 挂到当前 TCB 的 children 里