Lab3 report

总结

1. 将ch4移植到ch5。没啥特别的。
2. 实现sys_spawn。简单来说就是把fork和exec的代码提出来拼起来。但是注意这里不必赋值父进程的地址空间，因此构建子进程时直接用根据elf生成的地址空间即可。
3. 实现stride。算法思路就是tutorial里讲的。在TCB里记了priority和stride，然后每次在run_tasks调用fetch_task时，在后者里用stride替换掉fifo。stride取最小值我直接暴力遍历的，但不影响结果。

问答题

实际情况是轮到 p1 执行吗？ 为什么？

实际不是p1。因为此时p2.stride+10 = 260，但是因为用的是8bits存储，所以溢出了，且因为stride是无符号存储，所以结果是小于255的正数，结果下一次调度又会调度p2。因此就会有问题。

为什么？ 尝试简单说明（不要求严格证明）

我们考虑反证法。假设命不成立，即*STRIDE_MAX – STRIDE_MIN* > BigStride/2 。我们不妨设上次调度时仍然满足*STRIDE_MAX – STRIDE_MIN* <= BigStride/2（事实上，总存在这样的边界情况），则此次调度选取的必定不是stride_min，这和stride算法矛盾。因此证明了原命题。

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> {
        let offset = self as u64 - other as u64;
        if (offset > 0 && offset < BigStride/2)  || (offset < 0 && -offset > 
BigStride/2 ) {
          return Ordering::Greater; // self > other;
      }
      else {
        return Ordering::Less; // self < other;
      }
    }
}

impl PartialEq for Stride {
    fn eq(&self, other: &Self) -> bool {
        false
    }
}
```