Lab3 Report

一. 功能简介

1. 进程创建

Spawn 类似于 fork+exec，因此实现时将 task.rs 中的 fork 函数和 exec 函数拼接起来即可。同时由说明文档可知，不需要像 fork 一样复制父进程的地址空间，而是在构建进程使用 elf 创建进程空间。

2. stride 调度算法

首先，对于每一个 TaskControlBlockInner，按照说明文档，保存其 stride 和 priority。每次调用 fetch_task 时，首先通过暴力枚举，找到 stride 最小的 tcb，更新其的 stride 为 big_stride/priority，然后将其换到队首，将其弹出。

二. 思考题

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：不是。因为 250+pass=260，而由 8bit 无符号整形储存，因此将会溢出，导致实际上 p2.stride 还是小于 p1.stride，因此还是会执行 p2.

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：假设 STRIDE_MAX – STRIDE_MIN>BigStride / 2.如果上一次调度时 STRIDE_MAX – STRIDE_MIN <= BigStride / 2，那么这次调度时将不会调用 stride_min，因此与 stride 算法矛盾，假设不成立，得证。

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<Stride> 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。
答：

```rust
use core::cmp::Ordering;
struct Stride(u64);
impl PartialOrd for Stride {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        let tem=self as u64 - other as u64;
        if (tem>0&&tem<BigStride/2)||(tem<0&&-tem>BigStride/2 ) {
            Ordering::Greater
        }
        else {
            Ordering::Less
        }
    }
}
impl PartialEq for Stride {
    fn eq(&self, other: &Self) -> bool {
        false
    }
}
```