1. **在我们的多线程实现中，当主线程 (即 0 号线程) 退出时，视为整个进程退出， 此时需要结束该进程管理的所有线程并回收其资源。 - 需要回收的资源有哪些？ - 其他线程的 TaskControlBlock 可能在哪些位置被引用，分别是否需要回收，为什么?**

需要回收的资源：

回收TaskUserRes相关(提前回收, 不然会释放两次, 因为drop在函数周期结束后才调用, 晚于memory_set.recycle_data_pages调用), 回收fd_table, 回收children

其他线程的TaskControlBlock 的可能引用位置，及是否需要回收：

可能在锁或者信号量等的数据结构上, 但是不用回收, 地址空间已回收, 子线程运行时会自动失败

2

- **题目：**

对比以下两种 `Mutex.unlock` 的实现，二者有什么区别？这些区别可能会导致什么问题？

```rust
impl Mutex for Mutex1 {
    fn unlock(&self) {
        let mut mutex_inner = self.inner.exclusive_access();
        assert!(mutex_inner.locked);
        mutex_inner.locked = false;
        if let Some(waking_task) = mutex_inner.wait_queue.pop_front() {
            add_task(waking_task);
        }
    }
}

impl Mutex for Mutex2 {
    fn unlock(&self) {
        let mut mutex_inner = self.inner.exclusive_access();
        assert!(mutex_inner.locked);
        if let Some(waking_task) = mutex_inner.wait_queue.pop_front() {
            add_task(waking_task);
        } else {
            mutex_inner.locked = false;
        }
    }
}
```

- **答案：**

区别已经在图中圈出。 Mutex1直接释放锁，Mutex2则只在无人等待时释放锁。

如果像Mutex1一样释放锁，在释放锁后、未检查队列前的时隙，可能会有新进入的进程抢占锁，造成插队的不公平现象。

Mutex2则会优先将锁分配给队列中进程，公平性方面更好。

- **题目：**

  添加系统调用，负责死锁检测系统开关。完成死锁检测。

- **思路：**

  参考题目给出的思路，对AVAILABLE，NEED和Allocation进行维护。
  死锁检测使用银行家算法，但是没有必要，因为Need为0/1
  打开死锁检测时，在申请资源前对死锁进行检测即可
  注意打开死锁检测时也要进行死锁检测，若已死锁则打开失败