Lab5 report

黄立言 2019011329

实现

1. 关于几个数据结构的维护和实现：关于available，对于mutex，我们认为0表示已经被锁，1表示没有被锁；对于信号量，则count大于0时，我们认为可用资源为count，小于等于0时我们认为是0，后面的数据结构沿用这里的语义。关于allocation、need，我们分别在每个TaskControlBlock中维护两个vector（对应于两类资源），分别记录当前线程对于每个资源的使用情况。然后我们每次在分配资源前，先假定当前资源已分配（当然也可能阻塞），并运行给出的算法来判断是否会发生死锁。
2. 关于算法的实现：参考指导书即可。注意我这里遍历了每个线程的每个资源，但是实际上因为同一时刻每个线程只能获得或等待一个资源，所以不遍历所有资源也是正确的。

完成时间

不算阅读文档的时间，完成实验的净时间大概4小时上下吧。

问答

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?

根据 exit_current_and_run_next 调用的实现，当0号线程退出时，手动回收了分配给所有线程的TaskUserRes，包括tid、trap_cx，userstack，还有之前进程退出时需要回收的如清空地址空间等。

其他线程的TaskControlBlock可能在调度时，如run_tasks时被引用（具体来说，是在线程调度的ready_queue、锁相关的wait_queue中引用），在rcore中，我们虽然回收了上述的如tid等资源，但是主线程创建的线程的引用没有被从线程调度队列ready_queue中删除。但是显然这是需要进行处理的，假定我们运行下面的代码：

```
unsafe fn f() {
    sleep(1000000);
}
#[no_mangle]
pub fn main() {
    let mut v = Vec::new();
    for _ in 0..THREAD_COUNT {
        v.push(thread_create(f as usize, 0) as usize);
    }
    exit(0);
}
```

在当前的rcore下，因为我们并没有对ready_queue中的已经释放的非主线程TaskControlBlock引用进行处理，所以在run_tasks时这部分线程仍然可能被调度，这就会导致访问到已经被释放的数据，最终导致内核崩溃。

考虑到线程的引用可能在上述的read_queue和锁机制的wait_queue中用到，因此我们需要保证，当这些线程实际上被释放的时候，这些调度相关的数据结构不对这些线程进行调度，有两种方案：其一是在我们释放所有相关线程的时候，将这些队列中的引用一起消除；其二是在从这些引用中调度线程时，对其res成员进行一下特判，如果是None我们就不做调度（当然这种方式可能在内存占用上要大一些）。

2.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?

```
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;
        }
    }
}
```

两者的主要区别在于，Mutex1先解除了lock的状态后选出一个阻塞的线程唤起并调度，但是这样可能会存在问题：在这个被唤起的进程被调用之前，如果先调度了某个线程，其获取了这个锁（这显然是可以的，毕竟此时锁是unlocked的），那么之前唤起的线程就又拿不到锁了。

而Mutex2则避免了这个问题，其逻辑相当于，当存在被阻塞的进程需要被唤起时，为了避免上面的问题，我们干脆不释放锁，直接保持加锁的状态等待这个被唤起的线程拿到他（在这之中即便存在其他线程想拿这个锁，也会因为其实locked而被阻塞），这样就避免了上面的问题，当然如果没有进程待唤起，那么直接解锁即可了。

一点看法

感觉实验本身没啥可说的，但和同学交流后感觉实际难度比我理解的简单一点，因为我为sem和mutex分别做了记录，因此每次调用的时候都要处理引用的问题（当然实际上也只要用{}处理作用域即可），但是实际上因为测例中没有混用的情况，所以只要记录一个矩阵，当做两种用处即可。

感觉实验中遇到的最大的问题是对rust不那么熟导致的，因此在实现上会有各种问题，这在之前的实验中也一直存在，可能之后的学期助教们可以考虑在rust语言使用上可以多做点教学工作。