Lab3

Lab3 主要是第五章的内容，第五章引入了进程的概念。

由于进程概念的引入，许多核心数据结构发生了变化，因此在进行Lab3之前，我们需要先熟悉新的结构，重构之前两个实验中的系统调用。

首先是获取时间的**sys_get_time**系统调用

主要的修改的地方在于虚拟地址转物理地址的转换有了提供的实现接口，切换一下即可。

对于sys_task_info则需要多一些关注，因为引入进程的原因，原本的核心数据结构发生了变化：TaskManager的职能耦合度过高，分拆抽象为两部分，一部分TaskManager负责管理任务相关（包括调度），而另一部分Processor负责对于CPU的监控职能（比如当前任务等）。

所以拆分数据结构后原本的实现不可再用，需要另作一些修改，但主体逻辑变化不大，不再赘述。

对于Lab2中的mmap与munmap系统调用：基本的实现逻辑没有变化，只需要把模块向外提供接口的实现地点从TaskManager中转换到新的Processor中即可。

而本章新要求实现的调用需要重点说一下，首先是spawn，其目标是新建子进程，使其执行目标程序。但不是fork + exec的这样一个模式，不必像fork一样复制父进程的地址空间。

那么我想的是直接新建一个，把对应的东西设置好就好了：

```
pub fn sys_spawn(path: *const u8) -> isize {
    trace!(
        "kernel:pid[{}] sys_spawn",
        current_task().unwrap().pid.0
    );

    let token = current_user_token();
    let path = translated_str(token, path);

    if let Some(data) = get_app_data_by_name(path.as_str()) {
        let new_task = Arc::new( TaskControlBlock::new(data) );

        // set child-parent relationship
        let current_task = current_task().unwrap();
        let mut inner = current_task.inner_exclusive_access();
        inner.children.push(new_task.clone());

        let mut new_task_inner = new_task.inner_exclusive_access();
        new_task_inner.parent = Some(Arc::downgrade(&current_task));

        add_task(new_task.clone());
        let new_pid = new_task.pid.0;
        new_pid as isize
    } else {
        -1
    }
}
```

代码实现融合借鉴的fork和exec的代码实现，将其一体化从而实现spwan的功能。

遵循RAII的思想，不要忘记设置父子关系，否则会出现问题。

在能够新建进程之后，我们的系统里的进程/任务就多了起来，那么如何决定他们的执行顺序，也就是调度原则是很重要的，Lab3里要求实现一个Stride调度算法，该算法描述如下：

1. 为每个进程设置一个当前 stride，表示该进程当前已经运行的“长度”。另外设置其对应的 pass 值（只与进程的优先权有关系），表示对应进程在调度后，stride 需要进行的累加值。
2. 每次需要调度时，从当前 runnable 态的进程中选择 stride 最小的进程调度。对于获得调度的进程 P，将对应的 stride 加上其对应的步长 pass。
3. 一个时间片后，回到上一步骤，重新调度当前 stride 最小的进程。

可以证明，如果令 P.pass = BigStride / P.priority 其中 P.priority 表示进程的优先权（大于 1），而 BigStride 表示一个预先定义的大常数，则该调度方案为每个进程分配的时间将与其优先级成正比。证明过程我们在这里略去，有兴趣的同学可以在网上查找相关资料。

其他实验细节：

- stride 调度要求进程优先级 ≥2，所以设定进程优先级 ≤1 会导致错误。
- 进程初始 stride 设置为 0 即可。
- 进程初始优先级设置为 16。

为了实现这个算法，内核还需要添加设置优先级的系统调用支持set_prio：

```
pub fn sys_set_priority(prio: isize) -> isize {
    trace!(
        "kernel:pid[{}] sys_set_priority",
        current_task().unwrap().pid.0
    );

    if prio < 2 {
        return -1;
    }

    set_prio(prio)
}
    /// set the priority of current task
    fn set_prio(&mut self, prio: isize) -> isize {
        let curr = self.current().unwrap();
        let mut inner = curr.inner_exclusive_access();
        inner.priority = prio;
        inner.priority
    }
```

这还是比较简单的，主要困难在于调度部分，但有了对这个算法的准确描述，不难作出如下实现：

```
    pub fn fetch(&mut self) -> Option<Arc<TaskControlBlock>> {
        // self.ready_queue.pop_front()

        // find the TCB with minist stride in ready_queue and it's status must
be Ready
        let mut min_stride = isize::MAX;
        let mut min_stride_index = 0;
        for (index, tcb) in self.ready_queue.iter().enumerate() {
            let stride = tcb.inner_exclusive_access().stride;
            if stride < min_stride {
```

```
                min_stride = stride;
                min_stride_index = index;
            }
        }
        if min_stride == isize::MAX {
            return None;
        } else {
            let tcb = self.ready_queue.remove(min_stride_index);
            // for the tcb fetched, we need to update its stride
            let mut inner = tcb.inner_exclusive_access();
            inner.stride += BIGSTRIDE / inner.priority;
            drop(inner); // release tcb
            return Some(tcb);
        }
    }
```

这里我遵循实验文档的指引，直接使用遍历法寻找最小stride的进程，未采用数据结构作特殊处理。

至此，Lab3完结。