实验报告

CPU调度算法实验只需要实现接口Policy函数，也就是分别给出在时间中断到来的时候，新任务到达，发出IO请求，IO任务完成和任务结束时对CPU资源和IO资源的调度方案。实现和尝试了多个版本，最终采取的是MLFQ（multi-level feedback queue）来实现。

基础思路

最初的想法是使用MLFQ（multi-level-feedback-queue）来实现，通过设置多个级别的不同的queue，分别用于盛放不同优先级的任务。设置了六个queue，高优先级的任务到来时首先放置到第一个queue，而低优先级的任务到来时首先放置到第三个queue中，从第一个queue到第三个queue，优先级逐渐降低。下面是针对于每一个事件采取的操作：

中断，将这个任务从当前优先级中移除并加入到下一个优先级之中，最后一个优先级除外。并在第一级的queue中选择一个新的任务进行抢占。

新任务到达，根据优先级的不同加入到对应的queue之中。

任务完成，从对应的queue之中删除。

请求IO操作，将任务从对应的queue之中移除，并加入到专门的IO等待queue之中。

完成IO操作，将任务从IOqueue之中移除并加入到某一个优先级queue之中。

但是，这样做需要建立一个任务编号到他所在的queue编号之间的映射，使用map来存储，并且这个映射需要多次进行访问和更改操作，实际开销很大。而且使用了抢占的机制，使得对于任务的管理是非常混乱的，由于存在一系列的问题，导致这个版本最终没有通过并拿到分数。

改进思路

之后听其他同学建议，使用EDF作为实验算法，也就是单纯的跟据ddl进行排序，ddl临近的任务先做（跟我们肝ddl的策略很像）。这样一来，其实这个实验之中我们的主要任务就变成了一个数据结构的设计，这个数据结构要求可以满足：

- get_min()
- add()
- remove()

这三个基本的操作，并要求尽可能地减少所使用的时间。随后对于这个数据结构的实现尝试了多个不同的版本：

vector

直接使用vector作为存放所有任务的容器，get_min()和删除的时候全部都遍历一遍。这样一来，除了add操作是O(1)时间复杂度，其余的都是O(n)时间复杂度，不出意料地迎来了TLE。后来又尝试，希望通过维护一个有序的vector来减少时间复杂度，利用的是vector自身带有的insert操作，但是尝试发现，insert操作的时间复杂度是跟这个元素到最后一个元素之间的距离成正比的，相当于是把后边的元素全部进行一次移位。时间开销同样太大，不得不放弃。

priority queue

C++之中一个比较方便的数据结构，插入即自动排序，可以自己写一个lambda表达式作为排序的依据。本来想直接调用这个已有的数据结构就可以，但是发现priority queue是根本无法删除的，他所提供的删除方法只有pop(),也就是单纯地去掉队列首的元素，是非常单纯的queue结构。所以不可行

min heap

第三个尝试的数据结构是minheap最小二叉堆，他的取最小值的操作是O(1)时间复杂度的，插入操作的时间复杂度为O(logn)，整体来说比较理想，但是他的删除操作实际上是无法进行的——仅仅支持删除最小值。本来想尝试通过另外附加建立一个map，从任务的taskid映射到该任务在heap之中的下标（因为heap使用数组来实现），这样删除的时候就不必遍历整个heap。但是发现这样做其实是无法删除的，不能让这个位置简单地标记为“不可用”，因为heap的所有操作全部都高度依赖于下标之间的关系。所以min heap的方案也是不可行的。

linklist

最后的实现方案是自己实现了一个类似于链表的结构，并在此基础上维护一个最小值。这个最小值的维护方法是在add的时候将新的任务的ddl同当前最小值进行对比，如果更小的话就更新以下min指针。当有删除操作的时候，如果删除操作删除的恰好是min，那么就遍历一遍整个链表，寻找到最小的。同时，为了方便删除，整个数据结构之中还加入了一个map，建立从taskid到这个节点的指针之间的映射，从而在不删除min的情况下可以在常数时间之内进行删除操作。除了删除的时间复杂度可以达到O(n)之外，其余的操作全部都是O(1)级别的，感觉实现的方法有点类似于Fibonacci heap

反思总结

之前一直遇到的一个bug是，将数据结构的声明放到了函数的内部，由于只在一个时间点上调用这个函数，从而导致这个数据结构一直以来都是空的，导致运行错误，special judge一直在报 time limit exceed。后来把这个数据结构的生命放到policy函数外边，成为全局变量，就解决了。