schedlab实验报告

实现流程

在policy函数中，首先需要对所得的事件集进行处理：

(1)KTimer
将系统时间cur_time设置为事件里的时间。

(2)KTaskArrival
创建一个任务cur_task，将该任务的各个属性设置为事件任务的属性，再经过筛选后插入taskqueue中。

(3)KTaskFinish
在taskqueue中遍历找到事件所携带的task，并删除该task。

(4)KIoRequest
创建一个任务cur_task，将该任务的各个属性设置为事件任务的属性，再经过筛选后插入ioqueue中。之后需要遍历taskqueue删除掉这个即将做io操作的任务。

(5)KIoEnd
由于该任务做完了io操作，因此创建一个任务cur_task，将该任务的各个属性设置为事件任务的属性，再经过筛选后插入taskqueue中。之后需要遍历ioqueue删除掉这个已经做完io操作的任务。

完成对事件的处理后，再新建一个action，设置其ioTask和cpuTask并返回即可。

核心思想

(1)调度算法的选择
基于这个题目的特性，我选择了优先级队列对任务进行调度。首先我们定义一个结构体task，并使用集合容器set来存储task，具体代码如下：

```
struct task
{
    int id;
    int priority;
    int arrivaltime;
    int deadline;
};
set <task, taskcmp> taskqueue;
```

这里set容器的排序方法taskcmp即为算法的核心。由于题目要求我们兼顾高优先级任务优先和截止时间前完成任务，故关键的参数有priority、curtime、arrivalTime、deadline，对于参数的平衡我们可以有以下思路：

1. 不考虑priority，只以deadline作为排序标准
2. 考虑priority，并将它与deadline加权
3. 考虑priority，并将它与deadline-curtime加权
4. 考虑priority，并将它与deadline-arrivalTime加权

对于3的思路，我们采取的一种排序方式代码如下：

```
struct parameters
{
    int w1 = 450000;
    int w2 = 1;
    int w3 = w1;
}p;
class taskcmp
{
public:
    bool operator()(const task& a, const task& b) const
    {
        if (a.deadline - cur_time > 0 && b.deadline - cur_time <= 0)
        {
            return true;
        }
        if (a.deadline - cur_time <= 0 && b.deadline - cur_time > 0)
        {
            return false;
        }
        if (a.deadline - cur_time > 0 && b.deadline - cur_time > 0)
        {
            double wa = a.priority * p.w1 + (a.deadline - cur_time) * p.w2;
            double wb = b.priority * p.w1 + (b.deadline - cur_time) * p.w2;
            return wa < wb;
        }
        return true;
    }
};
class iocmp
{
public:
    bool operator()(const task& a, const task& b) const
    {
        if (a.deadline - cur_time > 0 && b.deadline - cur_time <= 0)
        {
            return true;
        }
        if (a.deadline - cur_time <= 0 && b.deadline - cur_time > 0)
        {
            return false;
        }
        if (a.deadline - cur_time > 0 && b.deadline - cur_time > 0)
        {
            double wa = a.priority * p.w3 + (a.deadline - cur_time) * p.w2;
            double wb = b.priority * p.w3 + (b.deadline - cur_time) * p.w2;
            return wa < wb;
        }
        return true;
```

```
    }
};
```

由代码可知，我们始终令可完成时间deadline - cur_time的权重w2为1，不断调整w1和w3的值来观测运行结果，在大多数情况下我们可令w1=w3，即使cpu和io的队列优先级一致。

(2)针对性优化

在上述提到的4个思路，实测均可拿到87+的分数，但各个思路均在某些测试点有着较好的效果。大量评测反馈结果表明：思路1在测试点1、3有较好的分数；思路2、3的不同参数在测试点6-10有着较好的分数；思路4的参数在测试点11-16有着较好的分数。因此，我们可以考虑对于每个测试点，找到不同测评方法中所得分数的最大值，然后利用该测试点的特征来识别测试点并采取对应的方法，这样能使得每个测试点都能达到目前测评结果的最大值。

(3)调度魔法

有些时候，你往往能发现自己代码中的漏洞反而使运行分数更高了。比如在思路4中，如果我们在IoRequest时不删除掉更新curtask的deadline语句，可以惊奇地发现它的某些参数在测试点15和16有着极高的分数。这也给予了我们一些思考：cputask和iotask的排序方式不一定需要遵循同一种模式，有时候不同的新奇尝试往往能起到意想不到的效果。

完整代码

见附件policy.cc。