实验报告

一、调度算法思路

为需要使用 CPU 资源和 IO 资源的任务分别设置一个等待队列，队列中按照截止时间由小到大排序，即截止时间最近的最靠前。首先遍历事件列表中的事件，并根据事件类型执行不同操作：时钟中断，则更新时间；新任务到达，则将其添加至 CPU 等待队列；任务完成，则从 CPU 队列中删除；请求 IO，则添加至 IO 队列并暂时从 CPU 队列中移除；结束 IO，则从 IO 队列中删除并添加回 CPU 队列。然后，当需要选择使用 CPU 或 IO 资源的任务时，只需要取相应队列中截止时间最近且还没有超时的任务来执行即可。

二、代码

```
exam > ics > schedlab > sched > cpp > src > policy.cc > ...
```

```cpp
#include "policy.h"
#include <map>
#include <queue>
#include <iostream>
using namespace std;

map<int, Event::Task> CPU_Queue;  //int类型存放的是deadline
map<int, Event::Task> IO_Queue;

int cur_time = -1;   //现在的时间
Action policy(const std::vector<Event>& events, int current_cpu,
              int current_io) {
  Action next;

  int earlist = 1000000;
  int length = events.size();
  for(int i = 0; i < length; i++){
    if(events[i].type == Event::Type::kTimer){ //时钟中断
      cur_time = events[i].time;  //更新时间
    }else if(events[i].type == Event::Type::kTaskArrival){   //新任务到达
      CPU_Queue.insert(map<int, Event::Task>::value_type(events[i].task.deadline, events[i].task));
      //添加该任务到列表
    }else if(events[i].type == Event::Type::kTaskFinish){ //任务完成
      for(map<int,Event::Task>::iterator iter = CPU_Queue.begin(); iter != CPU_Queue.end(); iter++){
        if(iter->second.taskId == events[i].task.taskId){  //从列表里删除该任务
          CPU_Queue.erase(iter);
          break;
        }
      }
    }else if(events[i].type == Event::Type::kIoRequest){  //任务请求IO，添加到IO列表中
      IO_Queue.insert(map<int,Event::Task>::value_type(events[i].task.deadline, events[i].task));
      for(map<int,Event::Task>::iterator iter = CPU_Queue.begin(); iter != CPU_Queue.end(); iter++){
        if(iter->second.taskId == events[i].task.taskId){
          CPU_Queue.erase(iter);  //一个任务不能同时占用CPU和IO，所以暂时从CPU队列中移除
          break;
        }
      }

    }else if(events[i].type == Event::Type::kIoEnd){  //任务结束IO
      CPU_Queue.insert(map<int,Event::Task>::value_type(events[i].task.deadline, events[i].task)); //重新添加回CPU队列中
      for(map<int,Event::Task>::iterator iter = IO_Queue.begin(); iter != IO_Queue.end(); iter++){
        if(iter->second.taskId == events[i].task.taskId){
          IO_Queue.erase(iter);
          break;
        }
      }
    }
  }

  next.cpuTask = current_cpu;
  next.ioTask = current_io;
```

```
53    if(current_io == 0){   //只有当IO资源空闲时才能执行新的IO任务
54      if(!IO_Queue.empty()){
55        map<int,Event::Task>::iterator iter;
56        for(iter = IO_Queue.begin(); iter != IO_Queue.end(); iter++){
57          if(iter->first > cur_time){  //找到截止时间最近且未超时的任务执行
58            break;
59          }
60        }
61        if(iter == IO_Queue.end()){  //若全超时，从头开始执行
62          iter = IO_Queue.begin();
63        }
64        next.ioTask = iter->second.taskId;
65      }
66    }
67
68    map<int,Event::Task>::iterator iter;
69    for(iter = CPU_Queue.begin(); iter != CPU_Queue.end(); iter++){
70      if(iter->first > cur_time){  //找到截止时间最近且未超时的任务执行
71        break;
72      }
73    }
74    if(iter == CPU_Queue.end()){
75      iter = CPU_Queue.begin();
76    }
77    next.cpuTask = iter->second.taskId;
78
79    return next;
```

三、曾经尝试

1.第一版

（1）算法思路描述：

这一版先进行一个简单尝试，思路比较简单直接：只考虑任务的截止时间，不考虑其他因素如优先级等等。为需要使用 CPU 资源和 IO 资源的任务分别设置一个等待队列，队列以 map 结构存储，其中 key 是任务的截止时间，value 是该任务，根据 map 的自动排序功能，等待队列即可按照截止时间从小到大排列，从而便于我们找到截止时间最近的任务。需要注意的是，由于一个任务不能同时占用 CPU 资源和 IO 资源，所以限制一个任务能在一个队列中，而不能同时等待 CPU 资源和 IO 资源。

首先遍历事件列表中的事件，并根据事件类型执行不同操作：时钟中断，则更新时间；新任务到达，则将其添加至 CPU 等待队列；任务完成，则从 CPU 队列中删除；请求 IO，则添加至 IO 队列并暂时从 CPU 队列中移除；结束 IO，则从 IO 队列中删除并添加回 CPU 队列。

然后，当需要选择使用 CPU 或 IO 资源的任务时，只需要取相应队列中截止时间最近且还没有超时的任务来执行即可。

（2）得分情况：

oj 得分 86，效果出乎意料地还不错

（3）存在问题：

只考虑了截止时间，而完全没有考虑任务的优先级，所以非常不全面。

2.第二版

（1）算法思路描述：

这一版希望在第一版的基础上，将任务的优先级也纳入考虑因素。仍然使用 map 结构存储两个队列，但是 key 值存放的是原始的截止时间乘以一个系数：对于低优先级任务，该系数大于 1；对于高优先级任务，该系数等于一。因此，相同截止时间下，高优先级任务在

队列中就会更优先。其余实现思路均与第一版相同。
（2）得分情况
低优先级任务权重系数为 1.5：86
低优先级任务权重系数为 2：80
低优先级任务权重系数为 3：80
（3）存在问题：
虽然尝试考虑了任务优先级，但是这种加权考虑的方式似乎不太可靠，可能会让一些截止时间近的低优先级任务不能及时完成。

最终版本：

受时间和精力的限制，以及考虑到现有得分已经达到了 86，就没有再进行其他调度思路的实现。而且在得分相同的情况下，希望算法能尽可能简单，所以选择了第一版作为最终版。