实验： schedlab

一、实验目标

设计一个面向单核的调度策略，并实现调度器对应接口。本题要求调度策略尽可能达到实时系统的要求（即所谓“准实时”调度），并根据接口实现的正确性与任务及时完成率进行评分。本题根据选手全场最优成绩评分，选手可通过实时提交评估调度策略性能。

二、实验概述

任务假设

- 任务包含完成截止时间（deadline），调度器应尽可能在截止时间前完成任务。
- 任务分为高优先级与低优先级，调度器应倾向优先完成高优先级任务。
- 每个任务包括一段以上 CPU 计算与零或多段 IO 操作，分别使用计算机的 CPU 资源和 IO 资源。调度器可以执行任务 T_{cpu} 的 CPU 计算时，并行执行任务 T_{io} 的 IO 操作。
- 在一个任务使用 IO 资源时，不允许其同时使用 CPU 资源。
- 每个任务都以 CPU 计算开始与结束。

调度规则

使用系统资源：任何时刻，最多只有一个任务 T_{cpu} 使用系统 CPU 资源进行计算，最多只有一个任务 T_{io} 使用系统 IO 资源进行操作。系统 CPU 资源可以在任意时刻被调度器切换并执行新任务的 CPU 计算；系统 IO 资源必须完成当前 IO 操作后，才能执行新任务的 IO 操作。

调度新任务：调度器在新任务到达、任务结束、任务请求 IO 操作、任务结束 IO 操作、时钟中断到来时被唤醒并收到通知，并调用选手实现的策略接口决定接下来被调度的任务。策略将输出任务 T'_{cpu} 以抢占当前 CPU 资源。T'_{cpu} 可以等于 T_{cpu}，即继续将 CPU 资源分配给旧任务；T'_{cpu} 可以为空，即将 CPU 资源空置。当不存在能够进行 CPU 计算的任务时，空置 CPU 资源是合理的。当 IO 资源空闲时，策略可输出任务 T'_{io} 以使 IO 资源服务新的任务。注意，因为 IO 资源无法实时切换，当旧任务 T_{io} 未完成时，不能开始新的 IO 操作。

三、实验思路

1. 最基础版FIFO

因为实验要求中写到，令 FIFO 调度下所有任务完成所需时间为 t_{FIFO}，当选手实现策略完成所有任务用时 t 小于等于 t_{FIFO} 时，认为选手策略实现正确。所以先考虑FIFO调度策略下的调度情况和最终的得分。

首先，第一版的FIFO策略：将所有CPU和IO资源都给最先到来的任务，其他新到来的任务被搁置。

根据要求中的提示以及自己不断利用输出查看当前执行情况发现，每次到来的事件只有正在执行并触发了相应任务或刚刚到来或时钟中断到来这几种情况，而已经到来的事件如果不加以处理是不会归还，即之后也不会再来，这样该事件相当于没有被执行完毕，最后会发生“error TLE”错误。故需要自己在函数外定义一个队列来存在这些暂时无法使用资源的任务。

经过输出信息到屏幕上，发现事件队列的长度最多为2，又发现其中一个长度为2的事件第一个是时钟中断事件，而第二个才是任务触发事件，故每次事件到来只需取列表最后一个即可。（该推论在之后发现错误，而本函数最后通过了检测，故在此不加叙述）

代码思路：在函数最前面定义一个**队列Q**用以存放事件，每次事件到来时，先判断其类型是否为kTaskArrival，如果是，则判断其与队列中内容不重复后将其放入队列最后。

根据上面的推论，取事件列表最后一个，根据其不同类型进行相应操作。

在判定条件时，先判断当前是否有事件在占用CPU资源，再判断当前事件是否为正在使用资源的事件，在判断是否为无用的时钟中断事件。

通过判定条件后，① 如果当前事件需要IO资源，则令其使用IO资源；② 如果当前事件结束IO的使用需要CPU资源，则令其使用CPU资源；③ 如果当前事件结束，则将其从事件队列Q中删除，并判断队列是否为空，取Q的首位事件执行任务；④ 如果当前事件是刚刚到来（针对第一个到来的事件），令其使用CPU资源。

```
#include "policy.h"
#include<stdio.h>
#include<iostream>
#include<set>
#include<queue>
using namespace std;

set <int> queue_bool;
queue <int> Q;
Action policy(const std::vector<Event>& events, int current_cpu,
              int current_io) {
  int cpu_taskid=current_cpu,io_taskid=0;
  int current_id=0;
  for(unsigned int i=0;i<events.size();i++)
  {
    if(events[i].type==Event::Type::kTaskArrival)
    {
      if(queue_bool.insert(events[i].task.taskId).second)
        Q.push(events[i].task.taskId);
    }
  }

  current_id=events.size()-1;

if(cpu_taskid==0&&events[current_id].task.taskId==Q.front().first&&events[curren
t_id].type!=Event::Type::kTimer)
  //是否有事件在占用CPU资源，当前事件是否为正在使用资源的事件，是否为无用的时钟中断事件
  {
    if(events[current_id].type==Event::Type::kTaskArrival)
      cpu_taskid=Q.front();
    else if(events[current_id].type==Event::Type::kIoRequest)
    {
      cpu_taskid=0;
      io_taskid=events[current_id].task.taskId;
    }
    else if(events[current_id].type==Event::Type::kTaskFinish)
    {
      Q.pop();
      if(!Q.empty())
        cpu_taskid=Q.front();
      else
        cpu_taskid=0;
    }
    else if(events[current_id].type==Event::Type::kIoEnd)
      cpu_taskid=events[current_id].task.taskId;
  }
```

```
  Action ans;
  ans.cpuTask=cpu_taskid;
  ans.ioTask=io_taskid;
  return ans;
}
```

缺点：仅考虑到事件到来时间，且IO资源会被闲置造成浪费。

测评得分：35

2. 进阶版FIFO

在第一版的基础上，将CPU资源和IO资源分开使用，如果一个事件占用IO资源，则CPU资源会有空缺，那么可以让后面的事件先使用CPU资源。当使用IO资源的事件又要使用CPU资源时，则当当前事件结束使用CPU资源后，将CPU资源"归还"给它。提高了资源利用率。

代码思路：因为使用完IO资源的事件对于CPU资源有更高的优先级，因此需要新建立一个**队列（wating_cpuid）**来存放此类事件。而在事件使用IO资源时也会产生其后不止一个事件需要使用IO资源的情况，故也许定义一个**waiting_ioid队列**来存放此类事件。

因为在当前代码调试时发现长度为2的事件队列出现的情况不止上一版本代码中所描述的情况，因此不再只考虑队列中的一个事件。

因为函数参数中的current_cpu和current_io并不能满足判定条件的需要，因此又重新在外面定义了current_cpuid和current_ioid用以记录当前CPU和IO资源的占用情况。

基本思路同上，将CPU资源和IO资源分开管理，针对不同的事件类型分类讨论，不过在使用CPU资源时先考虑waiting_cpuid队列中的事件。

```
#include "policy.h"
#include<stdio.h>
#include<iostream>
#include<set>
#include<queue>
#include<list>
using namespace std;
set <int> queue_bool;
list <int> Q;
list <int> waiting_cpuid;
list <int> waiting_ioid;
int current_cpuid=0,current_ioid=0;
Action policy(const std::vector<Event>& events, int current_cpu,
              int current_io) {
  int cpu_taskid=current_cpu,io_taskid=0;
  int current_id;

  for(unsigned int i=0;i<events.size();i++)
  {
    current_id=i;
    if(events[current_id].type==Event::Type::kTaskArrival)      //事件到来
    {
      if(queue_bool.insert(events[i].task.taskId).second)
        Q.push_back(events[i].task.taskId);
      if(current_cpuid==0)      //当前无CPU任务，执行队首任务
      {
        cpu_taskid=Q.front();
        current_cpuid=Q.front();
        Q.pop_front();
```

```
      }
    }
    else if(events[current_id].type==Event::Type::kIoRequest)    //IO请求
    {
      current_cpuid=0;
      if(current_ioid==0)        //当前无IO任务，执行请求任务
      {
        current_ioid=events[current_id].task.taskId;
        io_taskid=events[current_id].task.taskId;
      }
      else                       //当前有任务在占用IO资源，将当前任务入队
      {
        waiting_ioid.push_back(events[current_id].task.taskId);
      }
      if(!waiting_cpuid.empty())//当前任务发出IO请求，CPU资源空出，判断是否有事件需要使
用CPU资源
      {                          //并优先执行waiting_cpuid队列中的事件
        cpu_taskid=waiting_cpuid.front();
        current_cpuid=waiting_cpuid.front();
        waiting_cpuid.pop_front();
      }
      else if(!Q.empty())
      {
        cpu_taskid=Q.front();
        current_cpuid=Q.front();
        Q.pop_front();
      }
    }
    else if(events[current_id].type==Event::Type::kTaskFinish)       //事件结束
    {
      current_cpuid=0;
      if(!waiting_cpuid.empty())//当前任务执行完毕，CPU资源空出，判断是否有事件需要使用
CPU资源
      {                          //并优先执行waiting_cpuid队列中的事件
        cpu_taskid=waiting_cpuid.front();
        current_cpuid=waiting_cpuid.front();
        waiting_cpuid.pop_front();
      }
      else if(!Q.empty())
      {
        cpu_taskid=Q.front();
        current_cpuid=Q.front();
        Q.pop_front();
      }
    }
    else if(events[current_id].type==Event::Type::kIoEnd)            //CPU请求
    {
      current_ioid=0;
      if(!waiting_ioid.empty())//当前任务发出CPU请求，IO资源空出，判断是否有事件需要使用
IO资源
      {
        io_taskid=waiting_ioid.front();
        current_ioid=waiting_ioid.front();
        waiting_ioid.pop_front();
      }
      waiting_cpuid.push_back(events[current_id].task.taskId);
      if(current_cpuid==0)       //判断当前是否有事件在占用CPU资源
      {
```

```
            cpu_taskid=waiting_cpuid.front();
            current_cpuid=waiting_cpuid.front();
            waiting_cpuid.pop_front();
          }
        }
      }
      Action ans;
      ans.cpuTask=cpu_taskid;
      ans.ioTask=io_taskid;
      return ans;
    }
```

优点：相较上一版而言，分开考虑CPU和IO资源，提高了资源利用率

缺点：仅仅只按事件到来事件分配任务，很多因素未被考虑到

测评得分：50

3. 进阶版FIFO（考虑优先级）

由于单纯的FIFO只考虑到事件的先后性，故在该版本代码中还考虑了事件的优先级（high/low）。

代码思路：在第二版三个队列(Q、waiting_cpuid、waiting_ioid)的基础上对事件优先级加以分类，分为六个队列（Q_h、Q_l、waiting_cpuid_h、 waiting_cpuid_l、waiting_ioid_h、waiting_ioid_l）并在每次任务时优先考虑high优先级的任务。

注：因为代码思路与第二版基本一致且过于冗长，此处不加以展示。

优点：较上一版代码考虑到了事件的优先级，有了一定提升

缺点：仍然没有解决FIFO的硬伤，且提升幅度不大

测评得分：52

4. 评分制策略

顾名思义，即对在队列中的每个事件和当前事件根据arrivaltime、deadline、priority和used_times（即占用资源的时间）等进行量化并排序，每次事件触发时推出首位事件。

函数构造：因为本实验旨在令所有事件都尽可能在deadline之前完成它的任务，因此首先考虑deadline，若deadline里当前事件time越近说明其对资源的需求越紧迫，同时还要考虑到它已经占用资源的时间，毕竟相对来说，它占用资源的时间越长，那么它的紧迫性越低。而本实验中事件又分高/低优先级，故需要对不同优先级赋予不同权值（H_SCORE和L_SCORE），将其放入之前的函数中。因此函数为：

$$score = priority\_score * (deadline - time + used\_times)$$

每次取score最小的事件。

代码思路：因为现在队列中还需储存有关事件的一系列信息，单纯的int类型无法满足需求，故需自行定义一个**结构体E**来存放所需的**事件信息**。而队列(queue)也无法满足排序和指定删除的需要，故使用了列表（list）。

在此思路下不再需要考虑任务是否是执行完IO任务而需要占用CPU资源，故只需保留存放需要CPU资源任务的列表**Q_cpu**和存放需要IO资源任务的列表**Q_io**。每次事件触发（包括时钟中断）时将更新current_cpuid和current_ioid的used_times属性，并将current_cpuid放入Q_cpu中，对Q_cpu和Q_io进行排序，并根据需要取出队首事件。

因为函数在一开始认为到来事件的id与current_cpuid的id一致，但在调试过程中发现会出现这样的错误：同时触发两个事件，第一个事件触发是到来事件，根据函数判断接下来该执行该事件，所以原本事件被重新放入Q_cpu中；而第二个事件触发是原本事件任务完毕，函数便将当前事件删除，又将原本事件的id返回，造成了错误。针对这种情况（只有Timer、kIoRequest和kTaskFinish会出现这样的情况），分类讨论时判断到来事件的id与当前事件id是否一致，如不一致，将当前事件入列，并到Q_cpu列表中寻找此id。

```
#include "policy.h"
#include<stdio.h>
#include<iostream>
#include<set>
#include<queue>
#include<list>
using namespace std;

#define H_SCORE 0.5     //high priority的分数
#define L_SCORE 1.0     //low priority的分数

struct E        //事件结构体
{
  int id;
  int used_times;
  int deadline;
  double priority_score;
  double score;
  E (int a)
  {
    id=a;
    used_times=0;
  }
  E (){};
};

set <int> queue_bool;

list <E> Q_cpu;
list <E> Q_io;
E current_cpuid(0);
E current_ioid(0);
int last_time=0,now_time=0;

bool cmp(E e1,E e2) //队列函数排序的比较函数
{
  return e1.score<e2.score;
}

void update(int time) //每次事件触发都会执行此函数，对两个队列按照score重新排序
{
  for(list<E>::iterator i=Q_cpu.begin();i!=Q_cpu.end();i++)
  {
    i->score=i->priority_score*(i->deadline-time+i->used_times);
  }
  Q_cpu.sort(cmp);
  for(list<E>::iterator i=Q_io.begin();i!=Q_io.end();i++)
  {
    i->score=i->priority_score*(i->deadline-time+i->used_times);
  }
```

```
  Q_io.sort(cmp);
}

Action policy(const std::vector<Event>& events, int current_cpu,
              int current_io) {
  queue_bool.insert(0);
  int cpu_taskid=current_cpu,io_taskid=0;
  for(unsigned int i=0;i<events.size();i++)
  {
    now_time=events[i].time;
    if(events[i].type==Event::Type::kTaskArrival)       //事件到来
    {
      if(queue_bool.insert(events[i].task.taskId).second)       //入列操作
      {
        E tmp;
        tmp.deadline=events[i].task.deadline;
        tmp.id=events[i].task.taskId;
        tmp.used_times=0;
        if(events[i].task.priority==Event::Task::Priority::kHigh)
          tmp.priority_score=H_SCORE;
        else
          tmp.priority_score=L_SCORE;
        Q_cpu.push_back(tmp);
      }
      if(current_ioid.id!=0)                                    //更新
current_ioid的used_times
        current_ioid.used_times+=now_time-last_time;
      if(current_cpuid.id!=0)                                   //更新
current_cpuid的used_times
      {
        current_cpuid.used_times+=now_time-last_time;
        Q_cpu.push_back(current_cpuid);
      }
      update(now_time);                                         //更新Q_cpu和Q_io
      cpu_taskid=Q_cpu.front().id;
      current_cpuid=Q_cpu.front();
      Q_cpu.pop_front();
    }
    else if(events[i].type==Event::Type::kIoRequest)    //IO请求
    {
      if(events[i].task.taskId==current_cpuid.id)               //到来事件和当前事
件id一致
        current_cpuid.used_times+=now_time-last_time;           //更新
current_cpuid的used_times
      else          //到来事件和当前事件id不一致，到Q_cpu中寻找
      {
        for(list<E>::iterator j=Q_cpu.begin();j!=Q_cpu.end();j++)
        {
          if(j->id==events[i].task.taskId)
          {
            Q_cpu.push_back(current_cpuid);
            current_cpuid=*j;
            Q_cpu.erase(j);
            break;
          }
        }
      }
```

```
      if(current_ioid.id==0)          //当前无任务占用io资源，执行请求任务
      {
        current_ioid=current_cpuid;
        io_taskid=current_ioid.id;
      }
      else                            //当前有任务占用io资源，将当前任务入列，并更新当前io
任务的used_times
      {
        current_ioid.used_times+=now_time-last_time;
        Q_io.push_back(current_cpuid);
      }
      current_cpuid=E(0);
      update(now_time);               //更新Q_cpu和Q_io
      if(!Q_cpu.empty())              //CPU被闲置，判断是否有需要使用CPU的任务
      {
        cpu_taskid=Q_cpu.front().id;
        current_cpuid=Q_cpu.front();
        Q_cpu.pop_front();
      }
    }
    else if(events[i].type==Event::Type::kTaskFinish)    //事件结束
    {
      if(current_cpuid.id==events[i].task.taskId)                //到来事件和当前事
件id一致
        current_cpuid=E(0);
      else       //到来事件和当前事件id不一致，到Q_cpu中寻找
      {
        for(list<E>::iterator j=Q_cpu.begin();j!=Q_cpu.end();j++)
        {
          if(j->id==events[i].task.taskId)
          {
            Q_cpu.erase(j);
            break;
          }
        }
        Q_cpu.push_back(current_cpuid);
      }
      if(current_ioid.id!=0)          //更新current_ioid的used_times
        current_ioid.used_times+=now_time-last_time;
      update(now_time);               //更新Q_cpu和Q_io
      if(!Q_cpu.empty())              //CPU被闲置，判断是否有需要使用CPU的任务
      {
        cpu_taskid=Q_cpu.front().id;
        current_cpuid=Q_cpu.front();
        Q_cpu.pop_front();
      }
    }
    else if(events[i].type==Event::Type::kIoEnd)         //CPU请求
    {
      current_ioid.used_times+=now_time-last_time;
      Q_cpu.push_back(current_ioid);
      current_ioid=E(0);
      if(current_cpuid.id!=0)         //判断是否有任务在使用CPU资源
      {
        current_cpuid.used_times+=now_time-last_time;
        Q_cpu.push_back(current_cpuid);
      }
      update(now_time);               //更新Q_cpu和Q_io
```

```
      if(!Q_io.empty())              //IO被闲置，判断是否有需要使用IO的任务
      {
        io_taskid=Q_io.front().id;
        current_ioid=Q_io.front();
        Q_io.pop_front();
      }
      cpu_taskid=Q_cpu.front().id;
      current_cpuid=Q_cpu.front();
      Q_cpu.pop_front();
    }
    else                                                  //时钟中断
    {
      if(events.size()>1)            //如果时钟中断后还有事件触发，则执行下一事件触发
        continue;
      if(current_ioid.id!=0)
        current_ioid.used_times+=now_time-last_time;
      if(current_cpuid.id!=0)
      {
        current_cpuid.used_times+=now_time-last_time;
        Q_cpu.push_back(current_cpuid);
      }
      update(now_time);              //更新Q_cpu和Q_io
      if(!Q_cpu.empty())
      {
        cpu_taskid=Q_cpu.front().id;
        current_cpuid=Q_cpu.front();
        Q_cpu.pop_front();
      }
    }
    last_time=now_time;
  }

  Action ans;
  ans.cpuTask=cpu_taskid;
  ans.ioTask=io_taskid;
  return ans;
}
```

优点：相对FIFO而言考虑到了事件的多方面因素，未加以区分超时任务和未超时任务

缺点：函数构造相对笼统，且如优先级的比分是自己构造的，有失准确性。

测评得分：通过修改H_SCORE和L_SCORE，最高75（图片中h（H_SCORE）代表高优先级的权重，l（L_SCORE）代表低优先级的权重）：

```
h:1.0 l:1.0 70
h:0.2 l:1.0 75
h:0.5 l:1.0 75
h:0.4 l:1.5 75
```

5.评分制策略进阶版

第四版的代码得分相较于前3版有了很大的提升，但过于笼统，且对于已经超时的任务和未超时的任务一视同仁。

考虑到本实验的目的，我们更应该让还未超时的任务有更多的执行机会，而那些因为种种原因而不得不超时的任务，放到之后考虑也不会影响分数。因此构建一个关于$deadline - time$的函数，便可以有效地将超时和未超时的任务区分开来。

代码思路：与第四版基本一致，在结构体中新加入arrivaltime属性，并且由于构造的函数的性质，Q_cpu和Q_io列表按任务的score从大到小排序。并且对used_times也赋予了权值（USED_SCORE）

因为代码改动较小，故不加以展示，只在得分处展示不同函数的不同得分

优点：相对上一版考虑地更加全面

缺点：函数构造相对笼统，且权重是自己构造的，有失准确性。

测评得分：最高83

$$priority\_score * \frac{deadline - arrivaltime - USED\_SCORE * used\_times}{deadline - time}$$

(注：h：H_SCORE，l：L_SCORE，u：USED_SCORE)

h:1.5 l:1.0 u:1.0 81

h:1.5 l:1.0 u:0.8 82

h:1.5 l:0.5 u:0.8 81

h:1.5 l:1.0 u:0.5 81

h:1.25 l:1.0 u:0.85 82

$$priority\_score * \left(\frac{time - arrivaltime}{deadline - arrivaltime}\right)^2 * abs$$

(注：abs=1 或-1，当 deadline>time 时，abs=1，当 deadline<time 时，abs=-1)

H: 1.5 l:1.0 u:1.0 82

H: 2.0 l:1.0 u:0.85 83

H: 2.0 l:0.8 u:0.85 82

H: 2.5 l:1.5 u:0.85 82

$$priority\_score * \left(\frac{ttime - arrivaltime}{deadline - arrivaltime}\right) * abs$$

H: 1.5 l:1.0 u:1.0 83

H: 1.5 l:1.0 u:0.8 82

H: 1.25 l:1.0 u:0.85 83

H: 1.25 l:1.0 u:1.0 82

H: 1.25 l:1.0 u:0.9 83

H: 1.25 l:0.9 u:0.9 83

H: 2.0 l:0.9 u:0.9 82

H: 1.5 l:1.0 u:0.5 81

$$priority\_score * \left(\frac{time - arrivaltime}{deadline - arrivaltime}\right)^{\frac{1}{2}} * abs$$

H: 1.25 l:0.9 u:0.9 82

H: 1.5 l:0.9 u:0.9 82

H: 1.5 l:1.0 u:0.85 82

H: 2.0 l:0.9 u:0.9 82

H: 2.5 l:1.5 u:0.85 82

6. 最终版代码（得分最高）

此版代码只考虑了事件的deadline以及是否超时，根据事件的deadline从小到大排序，并优先考虑deadline晚于time的事件。

因此在上述代码中只需修改update函数和cmp函数即可：

```
bool cmp(E e1,E e2)
{
  if(e1.score>0&&e2.score>0)
    return e1.score < e2.score;
  if(e1.score>0&&e2.score<0)
    return 1;
  if(e1.score<0&&e2.score>0)
    return 0;
  if(e1.score<0&&e2.score<0)
    return e1.score>e2.score;
  else
    return 0;
}
```

```
void update(int time)
{
  for(list<E>::iterator i=Q_cpu.begin();i!=Q_cpu.end();i++)
  {
    int t=abs(i->deadline-time)/(i->deadline-time);
    i->score=i->priority_score*i->deadline*t;
  }
  Q_cpu.sort(cmp);
  for(list<E>::iterator i=Q_io.begin();i!=Q_io.end();i++)
  {
    int t=abs(i->deadline-time)/(i->deadline-time);
    i->score=i->priority_score*i->deadline*t;
  }
  Q_io.sort(cmp);
}
```

测评得分：92

四、实验总结

通过自己编写调度函数，让我对调度策略有了更深刻的印象。

此外，通过不断调试函数，根据错误信息不断修改代码，而且不断构思不同的调度策略，使自己的评分不断上升，也是一种乐趣啊！