schedlab report

罗思佳2021201679

一、实验目的

实验目的：理解CPU调度算法

实验内容：在oj上完成一道与调度有关的题目，并撰写实验报告

二、实验思路

在依次尝试了FIFO、优先权调度、时间片轮转法之后，最终采用（类似）MLFQ的调度策略，效果最好。

1.多级反馈队列：定义`std::vector< std::pair<Event, int> > eventlist`存放已到达的事件和优先级。因为MLFQ是用的多级反馈队列，这里可以结合vector和pair进行模拟，一个pair里存的是事件和它对应的优先级别。每次调用`policy`函数时对vector进行排序，每次把要执行的任务的优先级减一。

2.排序规则：首先比较优先级（不同于task结构体中的priority），优先级高的排在前面，如果优先级相同的话就比较执行时间（`deadline-arrivalTime`），执行时间短（紧凑）的排在前面。最开始写的时候其实有考虑task结构体的priority类，但是发现效果还不如不考虑的情况：(

3."抛上去"的周期：因为MLFQ策略需要每隔一定的时间将所有任务的优先级都设为最高，以避免某些任务饿死，因此我设置了每隔10个单位时间就把所有的event的优先级设为最高优先级`highestPrior`

三、代码框架

先遍历到达的事件`events`，如果是`kTimer`(时针中断到来)，不用处理；如果是`kTaskArrival`(新任务到来)，就将其加入到存放事件的容器`eventlist`中，优先级设为最高`highestPrior`；如果是`kTaskFinish`(任务结束)，就从`eventlist`中删除；如果是`IoRequest`，就将容器中对应事件的`type`属性修改；如果是`IoEnd`，在修改`type`属性的同时，判断该任务的id是否和`current_io`相同，是的话将`flag_ioend`属性改为真（说明当前的IO任务刚好结束）。

完成遍历和更新`eventlist`之后，再看是不是到了需要提高所有任务优先级的时刻，这里设定为每隔10个单位时间boost一次。

之后是对`eventlist`进行自定义排序，排序规则见上文。

排序之后，再对更新后的`eventlist`进行遍历，选出接下来要执行的任务id。如果是不需要进行IO操作，就将其作为下一个CPU任务；如果是需要进行IO操作，且满足`current_io`为0或者`flag_ioend`为`true`，就将作为下一个IO任务。

最后再将即将作为CPU任务的优先级减一（如果大于0的话），返回`action`。

```
#include "policy.h"
#include<algorithm>
#include <utility>
std::vector< std::pair<Event, int> > eventlist;//存放已知事件

bool comp(const std::pair<Event,int> &a, const std::pair<Event,int> &b) // 比较优先级的函数
```

```
{
    if(a.second != b.second)
      return a.second > b.second;
    else
      return (a.first.task.deadline - a.first.task.arrivalTime) <
(b.first.task.deadline - b.first.task.arrivalTime);
}

Action policy(const std::vector<Event>& events, int current_cpu,
int current_io) {

  int size = events.size() ;
  int next_cpu = 0, next_io = current_io;
  Action action;
  bool flag_ioend = false;
  int highestPrior = 7;
  for(int i = 0; i < size; i++)
  {
    if(events[i].type == Event::Type::kTaskFinish)
    {
      for(auto iter = eventlist.begin(); iter != eventlist.end() ;iter++)
      {
      if(iter->first.task.taskId == events[i].task.taskId)
      {
      if(!eventlist.empty())
      {
        eventlist.erase(iter);
      }
        break;
      }
      }
    }
    else if(events[i].type == Event::Type::kTaskArrival)
    {
      eventlist.push_back(std::make_pair(events[i], highestPrior));//加入列表中
    }
    else if(events[i].type == Event::Type::kIoRequest)
    {
      for (auto iter = eventlist.begin(); iter != eventlist.end(); iter++)
      {
        if (iter->first.task.taskId == events[i].task.taskId)
          {
            iter->first.type = Event::Type::kIoRequest;
            break;
          }
      }
    }
    else if(events[i].type == Event::Type::kIoEnd)
    {
      if(events[i].task.taskId == current_io)
        flag_ioend = true;
      for (auto iter = eventlist.begin(); iter != eventlist.end(); iter++)
      {
        if (iter->first.task.taskId == events[i].task.taskId)
        {
          iter->first.type = Event::Type::kIoEnd;
          break;
        }
```

```
      }
    }
  }
  if(events[0].time % 10 == 0)//周期性
  {
    for(auto iter = eventlist.begin(); iter != eventlist.end();iter++)
    {
      iter->second = highestPrior;
    }
  }

  sort(eventlist.begin(), eventlist.end(), comp);//降序排序

  bool flag_cpu = false;
  bool flag_io = false;

  for (auto iter = eventlist.begin(); iter != eventlist.end(); iter++)//遍历先前存
储的事件
  {

    if(iter->first.type != Event::Type::kIoRequest  && !flag_cpu)//不需要io
    {
      next_cpu = iter->first.task.taskId;
      flag_cpu = true;

    }

    if((current_io == 0 || flag_ioend) && iter->first.type ==
Event::Type::kIoRequest && !flag_io)
    {
      next_io = iter->first.task.taskId;
      flag_io = true;
    }
    if(next_cpu && next_io)//及时退出
      break;

 }
  if(current_io != 0)
    next_io = current_io;

  action.cpuTask = next_cpu;
  action.ioTask = next_io;

  for(auto iter = eventlist.begin(); iter != eventlist.end(); iter++)
  {
    if(iter->first.task.taskId == next_cpu && iter->second > 0)
    {
      iter->second--;
      break;
    }
  }

  return action;

}
```

四、实验结果

OJ平台得分：87分

测试点 #1	**Partially Correct**	**得分：** 88	**用时：** 143 ms	**内存：** 21020 KiB
测试点 #2	Partially Correct	得分：86	用时：81 ms	内存：1192 KiB
测试点 #3	Partially Correct	得分：82	用时：76 ms	内存：504 KiB
测试点 #4	Partially Correct	得分：90	用时：82 ms	内存：828 KiB
测试点 #5	Partially Correct	得分：88	用时：80 ms	内存：1432 KiB
测试点 #6	Partially Correct	得分：87	用时：69 ms	内存：580 KiB
测试点 #7	Partially Correct	得分：88	用时：92 ms	内存：840 KiB
测试点 #8	Partially Correct	得分：90	用时：85 ms	内存：788 KiB
测试点 #9	Partially Correct	得分：92	用时：80 ms	内存：980 KiB
测试点 #10	Partially Correct	得分：90	用时：93 ms	内存：956 KiB
测试点 #11	Partially Correct	得分：88	用时：87 ms	内存：960 KiB
测试点 #12	Partially Correct	得分：89	用时：81 ms	内存：968 KiB
测试点 #13	Partially Correct	得分：86	用时：86 ms	内存：908 KiB
测试点 #14	Partially Correct	得分：89	用时：72 ms	内存：860 KiB
测试点 #15	Partially Correct	得分：74	用时：118 ms	内存：976 KiB
测试点 #16	Partially Correct	得分：76	用时：95 ms	内存：876 KiB