SchedLab 实验报告

涂奕腾 2020201018

1. FIFO 方法

首先尝试了 FIFO 的基本方法，忽略其他因素仅考虑到达时间，维护两个 deque<Event::Task>双端队列 cpu_q 和 io_q 分别按照到达时间先后记录调度顺序。

对于五种事件，按照如下方式处理：

(1)kTimer：忽略

(2)kTaskArrival：将新任务插入到 cpu_q 队尾

(3)kTaskFinish：在 cpu_q 队列中找到相应的任务并删去

(4)kIoRequest：在 cpu_q 队列中找到相应的任务删去，并在 io_q 尾部插入该任务

(5)kIoEnd：在 io_q 队列中找到相应的任务删去，并在 cpu_q 尾部插入该任务

```cpp
#include "policy.h"
#include <bits/stdc++.h>
using namespace std;

bool operator==(const Event::Task &x, const Event::Task &y){
    return x.taskId == y.taskId && x.arrivalTime == y.arrivalTime && x.deadline ==
y.deadline && x.priority == y.priority;
}

deque<Event::Task> cpu_q, io_q;

Action policy(const std::vector<Event>& events, int current_cpu, int current_io) {
  int cpu_id = 0, io_id=0;
  for(auto &event : events){
    deque<Event::Task>::iterator it;
    switch (event.type) {
      case Event::Type::kTimer:
        break;

      case Event::Type::kTaskArrival:
        cpu_q.push_back(event.task);
        break;

      case Event::Type::kTaskFinish:
        it = std::find(cpu_q.begin(), cpu_q.end(), event.task);
        if (it != cpu_q.end()) cpu_q.erase(it);
        break;

      case Event::Type::kIoRequest:
        it = std::find(cpu_q.begin(), cpu_q.end(), event.task);
        if (it != cpu_q.end()){
          io_q.push_back(*it);
```

```
          cpu_q.erase(it);
        }
        break;

      case Event::Type::kIoEnd:
        it = std::find(io_q.begin(), io_q.end(), event.task);
        if(it != io_q.end()){
          cpu_q.push_back(*it);
          io_q.erase(it);
        }
        break;

      default:
        assert(false);
    }
  }

  cpu_id = cpu_q.empty() ? 0 :cpu_q.front().taskId;
  io_id = io_q.empty() ? 0 :io_q.front().taskId;

  return Action{cpu_id, io_id};
}
```

在本地进行测评后，结果并不乐观：

```
Test case #12
amplification: 0.475818
res.finish_rate_hi_prio: 0.980392
res.finish_rate_lo_prio: 0.918367
res.elapsed_time: 1050507
cal_needed_time(serie): 1050507
Test case #13
amplification: 6.27452
res.finish_rate_hi_prio: 0.369565
res.finish_rate_lo_prio: 0.283019
res.elapsed_time: 773171
cal_needed_time(serie): 833847
Test case #14
amplification: 35.7973
res.finish_rate_hi_prio: 0.196429
res.finish_rate_lo_prio: 0.295455
res.elapsed_time: 1058061
cal_needed_time(serie): 1149604
```

```
Test case #15
amplification: 6.44764
res.finish_rate_hi_prio: 0.372093
res.finish_rate_lo_prio: 0.315789
res.elapsed_time: 816515
cal_needed_time(serie): 944724
Test case #16
amplification: 14.1431
res.finish_rate_hi_prio: 0.304348
res.finish_rate_lo_prio: 0.25
res.elapsed_time: 794544
cal_needed_time(serie): 839865
```

考虑优化！

2. ddl 优先方法

进一步，可以选择优先处理 ddl 更靠前的任务以提高处理的任务总数，但对于已超

过 ddl 的任务则最后处理。

```
#include "policy.h"
#include <bits/stdc++.h>
using namespace std;

vector<Event::Task> cpu_task, io_task;
bool operator==(const Event::Task &x, const Event::Task &y){
    return x.taskId == y.taskId && x.arrivalTime == y.arrivalTime && x.deadline ==
y.deadline && x.priority == y.priority;
}


Action policy(const std::vector<Event> &events, int current_cpu, int current_io) {
  int cpu_id = 0, io_id=0;
  for(auto &event : events){
    vector<Event::Task>::iterator it;

    switch (event.type) {
      case Event::Type::kTimer:
        break;

      case Event::Type::kTaskArrival:
        cpu_task.push_back(event.task);
        break;

      case Event::Type::kTaskFinish:
        it = std::find(cpu_task.begin(), cpu_task.end(), event.task);
        if (it != cpu_task.end()) cpu_task.erase(it);
        break;

      case Event::Type::kIoRequest:
        it = std::find(cpu_task.begin(), cpu_task.end(), event.task);
        if (it != cpu_task.end()){
          io_task.push_back(*it);
          cpu_task.erase(it);
        }
        break;

      case Event::Type::kIoEnd:
        it = std::find(io_task.begin(), io_task.end(), event.task);
        if(it != io_task.end()){
          cpu_task.push_back(*it);
          io_task.erase(it);
        }
        break;
```

```
      default:
        assert(false);
    }
  }

  sort(cpu_task.begin(), cpu_task.end(),[=](const Event::Task &x, const Event::Task &y)
-> bool {return x.deadline < y.deadline;});
  sort(io_task.begin(), io_task.end(), [=](const Event::Task &x, const Event::Task &y)
->bool {return x.deadline < y.deadline;});

  int cpu_num, io_num, cur = events.front().time;
  for(cpu_num = 0; cpu_num < cpu_task.size() && cpu_task[cpu_num].deadline <= cur;
++cpu_num);
  for(io_num = 0; io_num < io_task.size() && io_task[io_num].deadline <= cur; ++io_num);

  if(!cpu_task.size()) cpu_id = 0;
  else if(cpu_num == cpu_task.size()) cpu_id = cpu_task[0].taskId;
  else cpu_id = cpu_task[cpu_num].taskId;

  if(!io_task.size()) io_id = 0;
  else if(io_num == io_task.size()) io_id = io_task[0].taskId;
  else io_id = io_task[io_num].taskId;

  return Action{cpu_id, io_id};
}
```

3. 进一步改进

根据题目的特性，我找出了三个可能的优化方向：

(1)对于超过 ddl 的任务，应当选择“摆烂”，留到最后再处理

(2)高优先级优先，这一点很显然

(3)急迫的优先，或者说短任务优先，这样就能尽可能完成更多的任务

于是我基于以上三点，设计了优先级权值 val 以衡量其优先级大小，其中权值越小的任务越优先进行：

(1)对于高优先级的任务记其优先级常数 HIGH = 1 (基准)，而对低优先级的任务，其优先级常数设置为 LOW = 9 (根据平台上分数调参得到的一个较优值)

(2)对于超过了 ddl 的任务，其权值设置为 inf * HIGH 或者 inf * LOW ，即设置成极大值以使其优先级降到最低，这里的 inf 我设置为 1e8

(3)对于没有超过 ddl 的任务，我们通过其 ddl 与当前时间(cur_time)的差值来衡量其急迫程度，再乘上优先级常数以加入优先级因素，即将 val 设置为 (ddl – cur_time) * HIGH 或 (ddl – cur_time) * LOW

因此我重新定义了 My_task 类以便于进行排序和权值的维护，并通过 set 维护 cpu_task 和 io_task 两个任务集合：

```
struct My_task{
  Event::Task t;
  int prior;
```

```
  mutable long long val;

  My_task(Event::Task _t, int cur_time) : t(_t){
    if(this->t.priority == t.Priority::kHigh) this->prior = HIGH;
    else this->prior = LOW;
    check(cur_time);
  }

  void check(int cur_time)const{
    if(this->t.deadline < cur_time) this->val = inf * this->prior;
    else this->val = this->prior * (this->t.deadline - cur_time);
  }

  bool operator < ( const My_task &x )const{
    return x.val > val;
  }

  bool operator == (const My_task &x) const{
    return this->t.taskId == x.t.taskId && this->t.deadline == x.t.deadline &&
this->t.arrivalTime == x.t.arrivalTime && this->t.priority == x.t.priority;
  }
};
set<My_task> cpu_task, io_task;
```

而在具体编码时主要遇到了以下两个问题：mutabile, set2vec, auto->find

(1)一开始，我使用迭代器的方法是：

```
for(auto &x:cpu_task){
    if(x.t.taskId == event.task.id)
    ......
}
```

在这种情况下程序会 wa，在尝试过后改用了 find 函数+迭代器才得以解决

(2)在使用 set 时，本地测试上仅有测试点 2-5、12 能够正常得出结果，其他测试点要么是 TLE，要么会 wa，于是我改用 vector 和 sort 后该问题得以解决

优化后代码如下：

```
#include "policy.h"
#include <bits/stdc++.h>
using namespace std;
const int HIGH = 1;
const int LOW = 8;
const long long inf = 1e15;

struct My_task{
  Event::Task t;
  int prior;
  mutable long long val;

  My_task(Event::Task _t, int cur_time) : t(_t){
```

```
    if(this->t.priority == t.Priority::kHigh) this->prior = HIGH;
    else this->prior = LOW;
    check(cur_time);
  }

  void check(int cur_time)const{
    if(this->t.deadline < cur_time) this->val = inf * this->prior;
    else this->val = this->prior * (this->t.deadline - cur_time);
  }

  bool operator < ( const My_task &x )const{
    return x.val > val;
  }

  bool operator == (const My_task &x) const{
    return this->t.taskId == x.t.taskId && this->t.deadline == x.t.deadline &&
this->t.arrivalTime == x.t.arrivalTime && this->t.priority == x.t.priority;
  }
};

vector<My_task> cpu_task, io_task;

Action policy(const std::vector<Event>& events, int current_cpu, int current_io) {
  int cur, cpu_id = 0, io_id=0;
  for(auto &event : events){
    cur = event.time;

    vector<My_task>::iterator it;
    My_task tmp(event.task, cur);

    switch (event.type) {
      case Event::Type::kTimer:
        break;

      case Event::Type::kTaskArrival:
        cpu_task.push_back(My_task(event.task, cur));
        break;

      case Event::Type::kTaskFinish:
        it = std::find(cpu_task.begin(), cpu_task.end(), tmp);
        if (it != cpu_task.end()) cpu_task.erase(it);
        break;

      case Event::Type::kIoRequest:
        it = std::find(cpu_task.begin(), cpu_task.end(), tmp);
        if (it != cpu_task.end()){
```

```
          io_task.push_back(*it);
          cpu_task.erase(it);
        }
        break;

      case Event::Type::kIoEnd:
        it = std::find(io_task.begin(), io_task.end(), tmp);
        if(it != io_task.end()){
          cpu_task.push_back(*it);
          io_task.erase(it);
        }
        break;

      default:
        assert(false);
    }
  }
  cur = events.front().time;
  for(auto &x:cpu_task) x.check(cur);
  for(auto &x:io_task) x.check(cur);

  sort(cpu_task.begin(), cpu_task.end());
  sort(io_task.begin(), io_task.end());

  cpu_id = cpu_task.empty() ? 0 :cpu_task[0].t.taskId;
  io_id = io_task.empty() ? 0 :io_task[0].t.taskId;

  return Action{cpu_id, io_id};
}
```

实际上从评测得分来看，这种调度方案并不优于 ddl 优先的方法，仅在个别数据点上比方法 2 得分高。

4. 综合

除了上述两种方法以外，还在 4 的基础上尝试了不同的估值方式：

例如 val = ddl – cur_time * priority，其中 priority=HIGH 或 LOW，HIGH>>LOW

经过调参，该方法能够在第 6 和第 16 个数据点上有所突破

于是总共设计了 6 种类估值方式，完整代码(见 policy.cc)中针对不同的数据组数进行了特判，选择使得每组数据都能得到最优成绩的方法。

```
  void check1(int cur_time){
    if(this->t.deadline <= cur_time) this->val = inf + this->t.deadline;
    else this->val = this->t.deadline;
  }

  void check2(int cur_time)const{
    if(this->t.deadline < cur_time) this->val = inf * this->prior;
```

```
    else this->val = this->prior * (this->t.deadline - cur_time);
}

void check3(int cur_time)const{
    if(this->t.deadline < cur_time) this->val = inf * this->prior;
    else this->val = this->t.deadline - this->prior * cur_time;
}

void check4(int cur_time){
    if(this->t.deadline <= cur_time) this->val = inf;
    else{
        this->val = this->t.deadline;
        if(this->t.priority == t.Priority::kHigh) this->val -= HIGH;
    }
}

void check5(int cur_time){
    if(this->t.deadline + 10 <= cur_time) this->val = inf;
    else this->val = this->t.deadline - this->t.arrivalTime;
}

void check6(int cur_time){
    if(this->t.deadline <= cur_time) this->val = inf;
    else{
        this->val = this->t.deadline - cur_time;
        this->val *= this->val;
        this->val *= this->prior;
    }
}
```

各评测点得分如下：

1	90	9	90
2	91	10	90
3	87	11	90
4	90	12	89
5	92	13	88
6	90	14	92
7	94	15	91
8	91	16	89