ICS II Schedule Lab 报告

中国人民大学 sheriyuo

1 基本思路

忽略 `kTimer`，在 `kTaskArrival` 时将任务加入 CPU 队列中，`kTaskFinish` 时将任务移出 CPU 队列。

由于 IO 操作无法中途抢占，维护 IO 队列，在 `kIoRequest` 时将任务移出 CPU 队列并加入 IO 队列，`kIoEnd` 时将任务移出 IO 队列并加入 CPU 队列中。

每次取队首执行即为 FIFO 的实现。

2 算法实现

2.1 Deadline First

对于队列中的任务，按照 `deadline` 由小到大排序，如果该任务已经超时则放到最后执行，每次取最小的任务执行。

朴素的算法为 86 分，考虑对优先级加权并不能优化分数。

2.2 MLFQ

为每个任务设置优先级，执行一段时间片后优先级减少，每 10 单位时间重置一次优先级。

对于超时任务的处理同 Deadline First 算法，同优先级采用 `deadline - arrivalTime` 由小到大排序。

朴素的算法为 88 分，考虑对优先级加权也不能优化分数。

在修改重置优先级的单位时间时，发现 $10, 50$ 等时间远优于 $49, 51$ 等时间，测试输出后发现 `Timer` 的间隙为 250 单位时间。将重置优先级的时间设为 1000 及以上时，分数大幅下降。

2.3 Length First + RR

基于 MLFQ 修改，删除优先级减少的判断，直接采用 `deadline - arrivaltime` 由小到大排序。

朴素的算法为 89 分，考虑对优先级加权，有极小幅度的优化。

考虑对较小权的几个任务按 RR 运行 CPU，有小幅度的优化，分数优化至 88.56 分（仍为 89）。

最终提交采用本算法，测试结果见图 1。

测试点 #1	Partially Correct	得分: 88	用时: 172 ms	内存: 21112 KiB
测试点 #2	Partially Correct	得分: 87	用时: 94 ms	内存: 1092 KiB
测试点 #3	Partially Correct	得分: 82	用时: 95 ms	内存: 496 KiB
测试点 #4	Partially Correct	得分: 90	用时: 101 ms	内存: 844 KiB
测试点 #5	Partially Correct	得分: 88	用时: 97 ms	内存: 1444 KiB
测试点 #6	Partially Correct	得分: 90	用时: 77 ms	内存: 576 KiB
测试点 #7	Partially Correct	得分: 87	用时: 119 ms	内存: 916 KiB
测试点 #8	Partially Correct	得分: 91	用时: 117 ms	内存: 788 KiB
测试点 #9	Partially Correct	得分: 88	用时: 94 ms	内存: 980 KiB
测试点 #10	Partially Correct	得分: 90	用时: 122 ms	内存: 972 KiB
测试点 #11	Partially Correct	得分: 91	用时: 100 ms	内存: 956 KiB
测试点 #12	Partially Correct	得分: 89	用时: 104 ms	内存: 968 KiB
测试点 #13	Partially Correct	得分: 88	用时: 148 ms	内存: 908 KiB
测试点 #14	Partially Correct	得分: 88	用时: 111 ms	内存: 872 KiB
测试点 #15	Partially Correct	得分: 91	用时: 243 ms	内存: 972 KiB
测试点 #16	Partially Correct	得分: 89	用时: 161 ms	内存: 872 KiB

图 1: 测试结果

3 附录

3.1 Deadline First

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

map<int, Event::Task> CPUQueue;
map<int, Event::Task> IOQueue;

int cur_time = -1;

void eraseTaskID(int taskId, map<int, Event::Task> &Queue) {
    for (auto it = Queue.begin(); it != Queue.end(); it++) {
        if (it->second.taskId == taskId) {
            Queue.erase(it);
            break;
```

```
        }
    }
}

double calcValue(Event::Task task) {
    double dtime = task.deadline < cur_time ? 1e9 : task.deadline;
    double ratio = task.priority == Event::Task::Priority::kHigh ? 1 : 1;
    return dtime - ratio * cur_time;
}

int findNext(map<int, Event::Task> &Queue) {
    auto ans = Queue.begin();
    for (auto it = Queue.begin(); it != Queue.end(); it++) {
        if (calcValue(it->second) < calcValue(ans->second))
        ans = it;
    }
    return ans->second.taskId;
}

Action policy(const std::vector<Event> &events, int current_cpu,
int current_io) {
    for (auto event : events) {
        switch (event.type) {
            case Event::Type::kTimer:
            cur_time = event.time;
            break;
            case Event::Type::kTaskArrival:
            CPUQueue.insert({event.task.deadline, event.task});
            break;
            case Event::Type::kTaskFinish:
            eraseTaskID(event.task.taskId, CPUQueue);
            break;
            case Event::Type::kIoRequest:
            IOQueue.insert({event.task.deadline, event.task});
            eraseTaskID(event.task.taskId, CPUQueue);
            break;
            case Event::Type::kIoEnd:
            CPUQueue.insert({event.task.deadline, event.task});
            eraseTaskID(event.task.taskId, IOQueue);
            break;
        }
    }

    Action action = {current_cpu, current_io};
    if (current_io == 0)
    if (!IOQueue.empty())
    action.ioTask = findNext(IOQueue);

    action.cpuTask = findNext(CPUQueue);

    return action;
```

```
}
```

3.2 MLFQ

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

vector<pair<int, Event::Task>> cpuQueue;
vector<pair<int, Event::Task>> ioQueue;

int cur_time = -1;
int maxPrio = 7;

bool cmp(pair<int, Event::Task> x, pair<int, Event::Task> y) {
    int px = x.second.deadline < cur_time ? -1e9 : x.first;
    int py = y.second.deadline < cur_time ? -1e9 : y.first;
    if (px == py)
    return x.second.deadline - x.second.arrivalTime < y.second.deadline - y.second.
        arrivalTime;
    else
    return px > py;
}

void eraseTaskID(int taskId, vector<pair<int, Event::Task>> &Queue) {
    for (auto it = Queue.begin(); it != Queue.end(); it++) {
        if (it->second.taskId == taskId) {
            Queue.erase(it);
            break;
        }
    }
}

int findNext(vector<pair<int, Event::Task>> &Queue) {
    if (!Queue.size())
    return 0;
    sort(Queue.begin(), Queue.end(), cmp);
    Queue[0].first--;
    return Queue[0].second.taskId;
}

Action policy(const std::vector<Event> &events, int current_cpu,
int current_io) {
    cur_time = events[0].time;
    if (cur_time % 10 == 0) {
        for (auto &it : cpuQueue)
        it.first = maxPrio;
```

```
        for (auto &it : ioQueue)
        it.first = maxPrio;
    }
    for (auto event : events) {
        switch (event.type) {
            case Event::Type::kTimer:
            break;
            case Event::Type::kTaskArrival:
            cpuQueue.push_back({maxPrio, event.task});
            break;
            case Event::Type::kTaskFinish:
            eraseTaskID(event.task.taskId, cpuQueue);
            break;
            case Event::Type::kIoRequest:
            ioQueue.push_back({maxPrio, event.task});
            eraseTaskID(event.task.taskId, cpuQueue);
            break;
            case Event::Type::kIoEnd:
            cpuQueue.push_back({maxPrio, event.task});
            eraseTaskID(event.task.taskId, ioQueue);
            break;
        }
    }

    Action action = {current_cpu, current_io};
    if (current_io == 0)
    if (!ioQueue.empty())
    action.ioTask = findNext(ioQueue);

    action.cpuTask = findNext(cpuQueue);
    return action;
}
```

3.3 Length First + RR

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

vector<pair<int, Event::Task>> cpuQueue;
vector<pair<int, Event::Task>> ioQueue;

int cur_time = -1;
int find_cnt = 0;
int maxPrio = 7;

mt19937 rd(time(0));
```

```
bool cmp(pair<int, Event::Task> x, pair<int, Event::Task> y) {
    int px = x.second.deadline < cur_time ? -1e9 : x.first;
    int py = y.second.deadline < cur_time ? -1e9 : y.first;
    double rx = x.second.priority == Event::Task::Priority::kLow ? 1.01 : 1;
    double ry = y.second.priority == Event::Task::Priority::kLow ? 1.01 : 1;
    if (px == py)
    return rx * (x.second.deadline - x.second.arrivalTime) <
    ry * (y.second.deadline - y.second.arrivalTime);
    else
    return px > py;
}

void eraseTaskID(int taskId, vector<pair<int, Event::Task>> &Queue) {
    for (auto it = Queue.begin(); it != Queue.end(); it++) {
        if (it->second.taskId == taskId) {
            Queue.erase(it);
            break;
        }
    }
}

int findNext(vector<pair<int, Event::Task>> &Queue) {
    if (!Queue.size())
    return 0;
    stable_sort(Queue.begin(), Queue.end(), cmp);
    auto it = Queue.begin() + 1;
    auto _begin = Queue[0];
    _begin.second.deadline += 0.0075 * (_begin.second.deadline - _begin.second.arrivalTime)
        ;
    while (it != Queue.end() && !cmp(_begin, *it))
    it++;
    return Queue[(++find_cnt) % (it - Queue.begin())].second.taskId;
}

Action policy(const std::vector<Event> &events, int current_cpu,
int current_io) {
    cur_time = events[0].time;
    for (auto event : events) {
        switch (event.type) {
            case Event::Type::kTimer:
            break;
            case Event::Type::kTaskArrival:
            cpuQueue.push_back({maxPrio, event.task});
            break;
            case Event::Type::kTaskFinish:
            eraseTaskID(event.task.taskId, cpuQueue);
            break;
            case Event::Type::kIoRequest:
            ioQueue.push_back({maxPrio, event.task});
            eraseTaskID(event.task.taskId, cpuQueue);
```

```
            break;
            case Event::Type::kIoEnd:
            cpuQueue.push_back({maxPrio, event.task});
            eraseTaskID(event.task.taskId, ioQueue);
            break;
        }
    }

    Action action = {current_cpu, current_io};
    if (current_io == 0)
    if (!ioQueue.empty())
    action.ioTask = findNext(ioQueue);

    action.cpuTask = findNext(cpuQueue);
    return action;
}
```