Schedulelab 实验报告

赵添予 2022201197

本次实验要求我们在 policy.cc 中实现模拟 CPU 调度的算法，并且返回当前 cpu 应该执行的任务 ID（分为 cpuTask 和 ioTask）

（一）一些个基础信息和准备

首先来看需要补写的函数定义：

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

一共传入三个参数，第一个是一个 Event，存放在一个容器中，一般情况下容器长度为 1；第二个是 current_cpu，代表当前的 cpu 任务；第三个是 curren_io，代表当前的 io 任务；

我们发现，该函数的返回值为一个 Action 结构体，而 Action 结构体定义如下：

```
struct Action {
  int cpuTask, ioTask;
};
```

所以返回值应该就是要让 cpu 和 io 执行的下一个任务的 taskID；

这里我们选择使用两个 map 数组对待执行的 cpuTask 和 ioTask 进行记录，key 值选择为 Event::Task.deadline。因为 map 在进行存储时候其自行会按照指定的 key 值进行维护，即存储时候就是有序的，这样方便我们对下一步要执行的 cpuTask 和 IoTask 进行选择。

```
map<int, Event::Task> cpu_todo;
map<int, Event::Task> io_todo;
```

（原先使用 vector 容器对需要执行的任务进行存储，还需要自己进行排序操作，而且还跑不出分数……）

首先在 policy 函数中初始化一些条件：

```
Action result;
result.cpuTask = current_cpu;
result.ioTask = current_io;
Event::Task temp;
```

定义要返回的内容 result，并且初始化为当前正在执行的任务

（二）正式开工

因为实验说明中说了 events 这个容器一般情况下长度为 1，但是以防万一，我们求出其长度，用一个循环遍历处理之：

```
int events_num = events.size();
int i = 0;
// int j = 0;
for (i = 0; i < events_num; i++)
{
```

跑完循环处理了之后，我们再最后决定让 cpu 和 io 做什么任务

接下来要依据不同的任务类型进行处理：
首先把任务类型转化为整型：

```
for (i = 0; i < events_num; i++)
{
    int event_type = (int)events[i].type;
```

（如果使用 Switch 语句也可以不用转化，但是我 Switch 老是出错，就没用，选择了转化为 int 型再用 if 条件句）

各个任务的处理：（ktimer 处理后面说）

1、kTaskArrival 处理：将任务添加到 map 中即可

```
if (event_type == 1)
{
    cpu_todo.insert(map<int, Event::Task>::value_type(events[i].task.deadline, events[i].task));
}
```

2、kTaskFinish 处理：将任务从 cpu_todo 中删除即可，这里指定了一个迭代器 iter，跑一个循环查找已经完成的任务，使用 map.erase(iter)将其删除，下面涉及到删除任务操作的都是使用这个方法

```
if (event_type == 2)
{
    map<int, Event::Task>::iterator iter;
    iter = cpu_todo.begin();
    while (iter != cpu_todo.end())
    {
        if (iter->second.taskId == events[i].task.taskId)
        {
            cpu_todo.erase(iter);
            break;
        }
        iter++;
    }
}
```

3、kIoRequest 处理：将任务从 cpu_todo 中删除，并添加到 io_todo 中。这里逻辑是，任务在到来时候就已经添加到了 cpu_todo 里面，所以需要 io 时候应该将其从 cpu_todo 里面删除，否则违反规则（做 io 时候不可以占用 CPU）

```cpp
if (event_type == 3)
{
    io_todo.insert(map<int, Event::Task>::value_type(events[i].task.deadline, events[i].task));
    map<int, Event::Task>::iterator iter;
    iter = cpu_todo.begin();
    while (iter != cpu_todo.end())
    {
        if (iter->second.taskId == events[i].task.taskId)
        {
            cpu_todo.erase(iter);
            break;
        }
        iter++;
    }
}
```

4、kIoEnd 处理：将任务从 io_todo 中删除，添加到 cpu_todo 中，因为任务完成了 io 之后要占用 cpu 资源

```cpp
if (event_type == 4)
{
    cpu_todo.insert(map<int, Event::Task>::value_type(events[i].task.deadline, events[i].task));
    map<int, Event::Task>::iterator iter;
    iter = io_todo.begin();
    while (iter != io_todo.end())
    {
        if (iter->second.taskId == events[i].task.taskId)
        {
            io_todo.erase(iter);
            break;
        }
        iter++;
    }
}
```

到这里，我们就基本完成了任务的分类，那么，map 既然本身就是按照 key 值排序的，我们先使用最朴素的方法，直接 check 之后把排在第一的任务作为下一个任务：

```cpp
map<int, Event::Task>::iterator result_cpu;
map<int, Event::Task>::iterator result_io;
result_cpu = cpu_todo.begin();
result_io = io_todo.begin();
if (current_io != result_cpu->second.taskId)  //最朴素的check当前任务之后直接把deadline最小的一个作为下一个执行
{
    result.cpuTask = result_cpu->second.taskId;
    if (current_io != 0)
    {
        result.ioTask = current_io;
        return result;
    }
    else
    {
        if (result.cpuTask != result_io->second.taskId)
        {
            result.ioTask = result_io->second.taskId;
            return result;
        }
    }
}
```

让我们看看结果如何：

#46014	#A. schedlab	Partially Correct	9	1437 ms	20.60 M	C++ 17 (schedlab) / 4.9 K

果然是不太行……
除去少数几个正确的节点之外，所有节点的都超了时间：

这个时候想起我们没有处理的 ktimer，开始读文档读的不仔细，对 ktimer 处理没有思路。
实验说明中是这样描述的：

> 调度器是操作系统的一部分，它决定计算机何时运行什么任务。通常，调度器能够暂停一个运行中的任务，将它放回到等待队列当中，并运行一个新任务，这一机制称为抢占（preemption）。抢占的实现往往需要通过硬件时钟（timer）定时发起中断（interrupt）信号，告知调度器一定时间周期已经过去，并由调度器决定下一个运行的任务。

所以 ktimer 能够发挥的作用应该是告诉我们什么时候应该让另一个任务对 cpu 资源进行抢占。
这里对于 timer 的处理参考了 github 上的做法：

```
int now_time = -1;
```

```
if (event_type == 0)
{
    now_time = events[i].time;
}
```

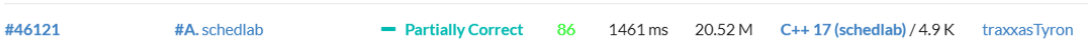
就是定义一个全局的 now_time 变量，记录 ktimer 来到时候的 events 里面的 time，作为最后的决策参考。
有了这个东西之后，我们在最后决策返回哪一个 taskID 时候稍作改动就能优化：即在原有的基础上利用迭代器分别对 cpu_todo 和 io_todo 跑一次循环，寻找有无 deadline 大于已经到来的 timer，如果有，则下一个 cpu 任务或者 io 任务就是这个任务，因为其时间优先级高。如果没有，则选 map 中第一个作为下一个执行的任务

```cpp
map<int, Event::Task>::iterator result_cpu;
map<int, Event::Task>::iterator result_io;
result_cpu = cpu_todo.begin();
result_io = io_todo.begin();
if (current_io == 0)
{
    if (!io_todo.empty())
    {
        result_io = io_todo.begin();
        while (result_io != io_todo.end())
        {
            if (result_io->first > now_time)

                break;

            result_io++;
        }
        if (result_io == io_todo.end())
            result_io = io_todo.begin();
        result.ioTask = result_io->second.taskId;
    }
}
result_cpu = cpu_todo.begin();
while (result_cpu != cpu_todo.end()) // 参考了github上的部分内容
{
    if (result_cpu->first > now_time)
    {
        break;
    }
    result_cpu++;
}
if (result_cpu == cpu_todo.end())
{
    result_cpu = cpu_todo.begin();
}
result.cpuTask = result_cpu->second.taskId;
```

这个方法得到的结果很好：



方法参考来源：
https://github.com/DannieGuo/ICS-Labs/blob/main/schedlab

小总结：
一个较小的改动就能够对任务调度实现很好的优化，但是这个改动确实需要花费比较多的时间和精力去发掘，而且理论懂了和能够代码实现还是两码子事儿的……
（同时，在判断 events 的类型时候，一使用 Switch 语句，如果各个 case 没有加花括号，则出现非法跳转问题，但是加了花括号，则会出现段错误，非常离谱，无论数据结构式 vector 还是 map，只要使用了 Switch 语句，无不报错，不知道其他同学有没有这种情况，尝试了很多次都无果，果断选择了换成 if 语句，就没有错误了。这是不是跟分支预测惩罚有关？）