《计算机操作系统》实验报告

实验题目：操作系统的进程调度

姓名：汪雨卿　学号：19120191　　实验日期：2021.12.07

实验环境：

Visual Studio 2019; C++

实验目的：

进程是操作系统最重要的概念之一，进程调度又是操作系统核心的主要内容。本实习要求学生独立地用高级语言编写和调试一个简单的进程调度程序。调度算法可任意选择或自行设计。例如，简单轮转法和优先数法等。本实习可加深对于进程调度和各种调度算法的理解。

实验内容：

（1） 编制和调试示例给出的进程调度程序，并使其投入运行。

（2） 自行设计或改写一个进程调度程序，在相应机器上调试和运行该程序，其功能应该不亚于示例。

提示：可编写一个反馈排队法（FB 方法）的进程调度程序。该算法的基本思想是设置几个进程就绪队列，如队列 1,,,,队列 i ，同一队列中的进程优先级相同，可采用先进先出方法调度。各队列的进程，其优先级逐队降低。即队列 1 的进程优先数最高，队列 i 的最低。而时间片，即以此占用 CPU 的时间正好相反，队列 1 的最短，队列 i 则最长。调度方法是开始进入的进程都在队列 1 中参加调度，如果在一个时间片内该进程完不成，应排入队列 2，即优先级要降低，但下一次运行的时间可加长（即时间片加长了）。以此类推，直至排到队列 i。调度时现在队列 1 中找，待队列 1 中已无进程时，再调度队列 2 的进程，一旦队列 1 中有了进程，又应返回来调度队列 1 的进程。这种方法最好设计成运行过程中能创造一定数量的进程，而不是一开始就生成所有进程。

提示：可综合各种算法的优先，考虑在各种不同情况下的实施方法，如上述 FB 算法。也可选用有关资料中报导的一些方法，加以分析、简化和实现。

（3）直观地评测各种调度算法的性能。

操作过程：

1. 设计算法的流程图

图 1.1 调度算法逻辑框图中展示了本程序的算法逻辑过程。

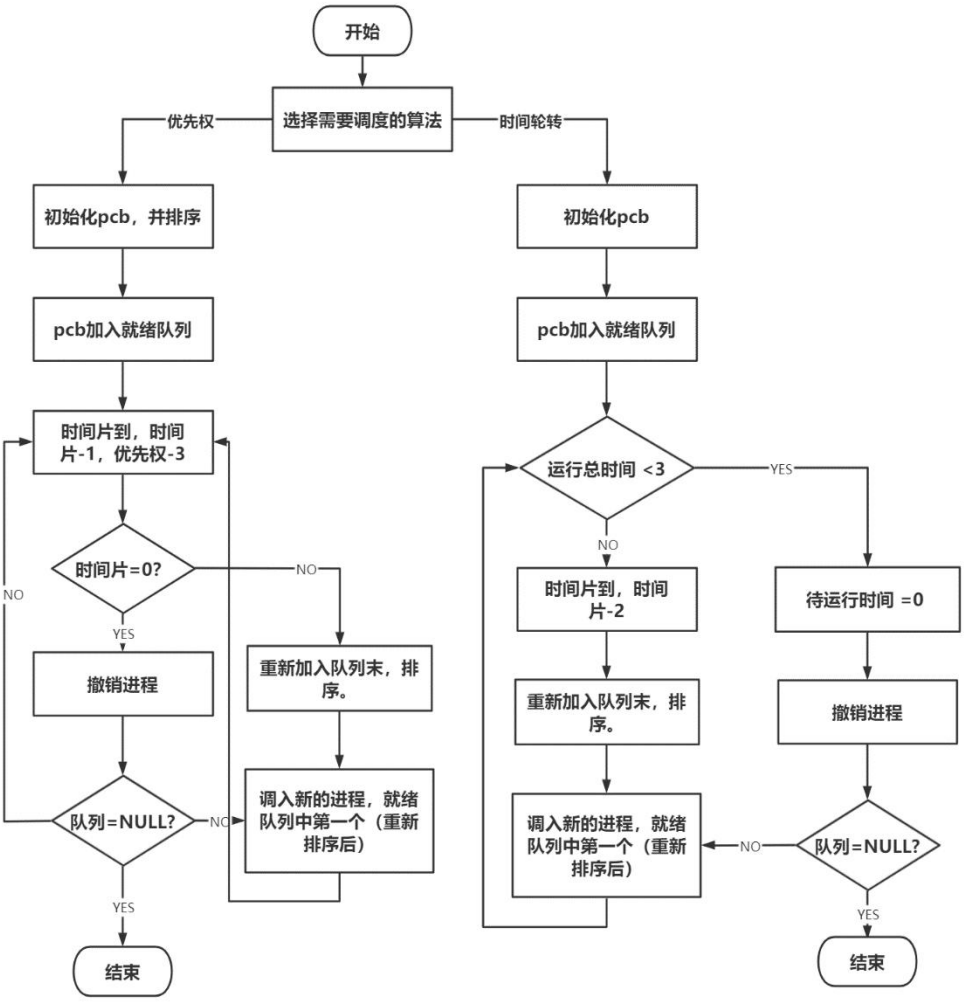


图 1 调度算法逻辑框图

2. 设计进程控制块 PCB 的数据结构

将进程控制块的 PCB 用类来表达，设置 PID,Priority,CPUtime,ALLtime,State,Finish。并设置对应的 getter 和 setter 方法。

- PID：标志进程块的序号
- Priority：标志 PCB 的优先级
- CPUtime：记录 CPU 已经运行的时间片数
- ALLtime：记录该进程还需要运行的时间片数
- State：表示进程当前的运行状态 （1.运行 2.就绪 3. 等待 -1.未初始化）
- Finish：标志进程是否完成（TRUE 完成，False 未完成）

图 2 PCB 类实现中展示了具体代码实现。

```
class PCB               // 定义PCB进程的数据结构
{
private:
    int PID;            // 进程控制块ID
    int Priority;       // 优先权数
    int CPUtime;        // CPU已经运行的时间
    int ALLtime;        // 剩余需要运行的时间
    int State;          // 进程当前的状态；1 运行 2 就绪 3 等待 -1 未初始化
    bool Finish;        // 进程完成标志；True = 完成，False 未完成
public:
    PCB();
    void setPCB(int pid, int priority, int all_t, int state, bool finish);
    int getPID();
    int getPriority();
    void setPriority(int priority);
    int getCPUtime();
    void setCPUtime(int cpu_t);
    int getALLtime();
    void setALLtime(int all_t);
    int getState();
    void setState(int state);
    bool getFinish();
    void setFinish(bool finish);

};
```

图 2 PCB 类实现

3. 设计两种调度算法的具体实现

a. 优先调度序算法的实现

图 3-4 优先级调度算法代码实现 显示了该调度算法的具体实现

```
// 优先级排序算法
void priority_choose(vector<PCB>& pcb, vector<PCB>& wait) {
    /*参数：pcb:进程队列，wait:就绪队列，num:进程的个数*/
    vector<PCB>::iterator it;
    it = pcb.begin();
    int count = pcb.size();      // count = 剩余要运行的进程个数
    while (count)
    {
        wait.erase(wait.begin());    //调入就绪队列中第一个进程

        // 更新pcb的数据
        (*it).setState(1);
        (*it).setPriority((*it).getPriority() - 1);
        if ((*it).getPriority() < 0)     // 当优先级减到0的时候，不再继续减
        {
            (*it).setPriority(0);
        }
        (*it).setCPUtime((*it).getCPUtime() + 1);
        (*it).setALLtime((*it).getALLtime() - 1);
```

图 3 优先级调度算法代码实现

```
        // 当有进程运行完成，输出结果，并且从两个队列中移除
        if ((*it).getALLtime() == 0)
        {
            count -= 1;             // 剩余进程数-1
            (*it).setFinish(true);
            cout << "--------------进程" << (*it).getPID() << "运行完毕------------" << endl;
            cout << left << setw(10) << (*it).getPID() << setw(10) << (*it).getState() << setw(10) << (*
            pcb.erase(pcb.begin());
            sort(wait.begin(), wait.end(), priority_sort);
            sort(pcb.begin(), pcb.end(), priority_sort);
            it = pcb.begin();
        }
        else
        {
            // 当进程没有运行完，输出结果，继续运行
            cout << left << setw(10) << (*it).getPID() << setw(10) << (*it).getState() << setw(10) << (*
            (*it).setState(2);      // 进程处于等待状态
            wait.push_back(*it);    // 未运行完的pcb重新插入就绪队列的末尾
            sort(wait.begin(), wait.end(), priority_sort);
            sort(pcb.begin(), pcb.end(), priority_sort);
        }
    }
}
```

图 4 优先级调度算法代码实现

b. 时间调度算法的实现

图 5 时间片轮转调度算法代码实现 显示了该调度算法的具体实现

```
void rr_choose(vector<PCB>& pcb, vector<PCB>& wait)
{
    vector<PCB>::iterator it;
    int count = pcb.size();
    while (count)
    {
        it = pcb.begin();
        wait.erase(wait.begin());
        (*it).setState(1);
        if ((*it).getALLtime() < 3)
        {
            count -= 1;
            (*it).setCPUtime((*it).getALLtime());
            (*it).setALLtime(0);
            (*it).setFinish(true);
            cout << "--------------进程" << (*it).getPID() << "运行完毕---------
            cout << left << setw(10) << (*it).getPID() << setw(10) << (*it).get
            pcb.erase(pcb.begin());
            continue;
        }
        else
        {
            (*it).setCPUtime((*it).getCPUtime() + 2);
            (*it).setALLtime((*it).getALLtime() - 2);
            cout << left << setw(10) << (*it).getPID() << setw(10) << (*it).get
            (*it).setState(2);
            wait.push_back(*it);
            PCB temp = *it;
            pcb.erase(pcb.begin());
            pcb.push_back(temp);
        }
    }
}
```

图 5 时间片轮转调度算法代码实现

4.优先级调度算法的 sort 函数重载

```
// sort函数重载，自定义按优先数排序。优先数相同时，按序号排序（先进先出）
bool priority_sort(PCB& p, PCB& q)
{
    if (p.getPriority() == q.getPriority()) return p.getPID() < q.getPID();
    return p.getPriority() > q.getPriority();
}
```

5. 主程序设计

```
int main()
{
    int num = -1; // num:需要被调度的进程个数
    char choice = NULL;
    cout << "===============调度算法目录===============" << endl;
    cout << "A. 优先数调度算法" << endl;
    cout << "B. 时间片轮转调度算法" << endl;
    cout << "请选择想模拟的调度算法:" << endl;

    // 输入0 退出
    while (cin >> choice)
    {
        switch (choice)
        {
        case 'A':
        {
                                    // 增加作用域，可以解决在switch中创建class对象
            // 1. 初始化进程
            vector<PCB> pcb;        // 创建PCB的进程队列

            cout << "===============优先数调度算法===============" << endl;
            cout << "请输入进程的总个数: " << endl;
            cin >> num;

            vector<PCB> wait;   // 创建pcb的就绪队列
            vector<int> a(num);     // 创建大小为num的整型数组,用于记录优先级
            vector<int> b(num);     // 创建大小为num的整型数组,用于记录时间片数
            cout << "创建" << num << "个进程，请为新建进程设置优先数(优先数越大，优先级越高)" << endl;
            cout << "例如: 1 2 3 4 5 ，即将5个进程的优先数分别设置为1 2 3 4 5" << endl;

            for (int i = 0; i < num; i++)
            {
                cin >> a[i];
            }
            cout << "请为新建进程设置各个需要运行的时间片数" << endl;
            for (int i = 1; i <= num; i++)
            {
                cin >> b[i-1];
                PCB p1;                                     // 创建一个新的pcb进程控制块
                p1.setPCB(i, a[i - 1], b[i-1], 2,false);    // 初始化进程块的信息
                pcb.push_back(p1);                          // 把新增pcb加入PCB队列的末端
            }

            // 利用重载sort函数，对pcb队列进行基于优先数的排序
            // void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
            // 开始，结尾，和排序的顺序
            sort(pcb.begin(),pcb.end(),priority_sort);
            copy(pcb.begin(), pcb.end(), back_inserter(wait));  // 将排序后的pcb队列全部加载入就绪队列
```

```
            //2. 输出目录
            cout << endl;
            cout << "=============== 开始运行 ===============" << endl;
            cout << "该进程运行情况如下 (Finish=0代表未完成, Finish=1代表已完成) " << endl;
            cout << left << setw(10) << "ID" << setw(10) << "State" << setw(10) << "Priority" << setw(10) << "CPUti
            priority_choose(pcb, wait);
            cout << "=============== 所有进程均已运行完毕 ===============" << endl;
            break;
        }
        case 'B':
        {
            // 1. 初始化进程
            vector<PCB> pcb;        // 创建PCB的进程队列

            cout << "===============时间片轮转调度算法===============" << endl;
            cout << "请输入进程的总个数: " << endl;
            cin >> num;

            vector<PCB> wait;   // 创建pcb的就绪队列
            vector<int> t(num);     // 创建大小为num的整型数组,用于记录时间片数
            cout << "创建" << num << "个进程, 请为新建进程设置时间片数。例如: 1 2 3 2 3" << endl;
            for (int i = 1; i <= num; i++)
            {
                cin >> t[i - 1];
                PCB p2;                                 // 创建一个新的pcb进程控制块
                p2.setPCB(i, -1, t[i - 1], 2, false);       // 初始化进程块的信息
                pcb.push_back(p2);                          // 把新增pcb加入PCB队列的末端
            }
            copy(pcb.begin(), pcb.end(), back_inserter(wait));  // 将pcb队列全部加载入就绪队列

            //2. 输出目录
            cout << endl;
            cout << "=============== 开始运行 ===============" << endl;
            cout << "该进程运行情况如下 (Finish=0代表未完成, Finish=1代表已完成) " << endl;
            cout << left << setw(10) << "ID" << setw(10) << "State" << setw(10) << "Priority" << setw(10) << "CPUti
            rr_choose(pcb,wait);
            cout << "=============== 所有进程均已运行完毕 ===============" << endl;
            break;
        }
        default:
            break;
        }
    }
    return 0;
```

5. 初始条件设置

a. 优先数调度算法的设置

选择调度算法：A 选择优先数调度算法

设置进程个数　　4

设置优先级　　4 2 1 3

设置时间片数　　5 3 4 5

b. 时间片轮转算法的设置

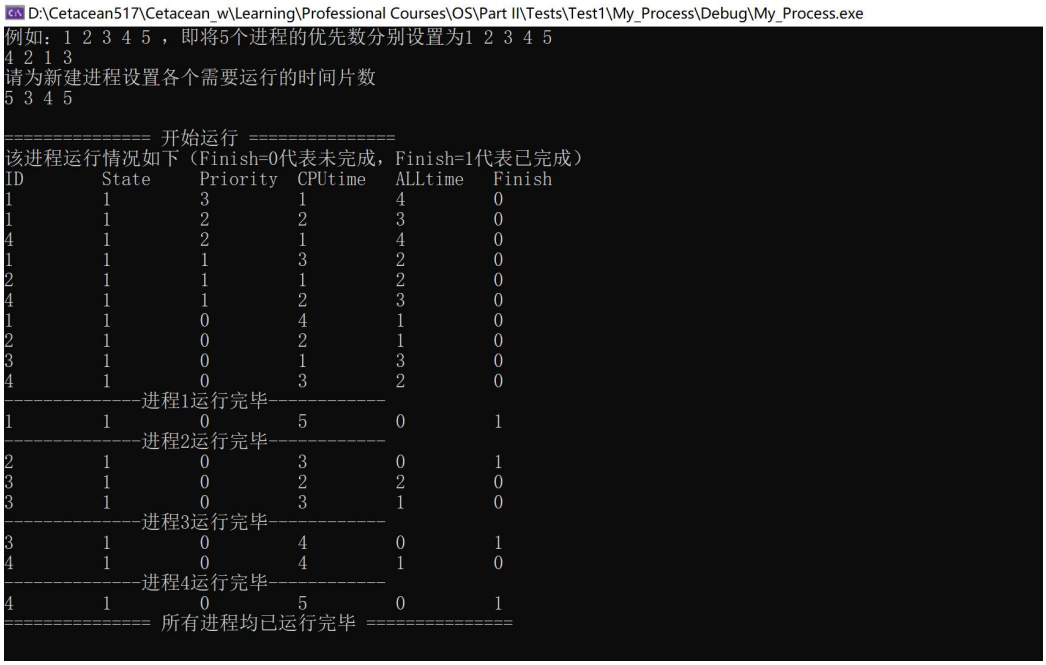
选择调度算法：B 选择时间片轮转调度算法

设置进程个数　　4

设置时间片数　　5 3 4 5

结果：

1.输出运行结果



2. 评价指标

平均周转时间 = （完成时间-到达时间）/ 进程个数

带权周转时间 = （周转时间/服务时间）/进程个数

3. 两个算法性能比较

a. 时间片轮转

进程号	服务时间	周转时间	带权周转时间	平均周转时间	平均带权周转时间
1	5	12	2.4	10.50	2.505
2	3	8	2.67		
3	4	11	2.75		
4	5	11	2.2		

b. 优先权

进程号	服务时间	周转时间	带权周转时间	平均周转时间	平均带权周转时间
1	5	6	1.2	10.25	2.5125
2	3	9	3		
3	4	13	3.25		
4	5	13	2.6		

总结：

在 4 个进程同时到达，且只有一个 CPU 为期服务的时候。优先数调度算法平均周转时间更短，性能更优。

体会：

本次实验在于模拟进程调度的算法调度的过程，同时进行性能的比较。在实验实现的过程中，才会发现理论掌握和实践实现的过程还是有很大的区别。最开始在设计 PCB 的数据结构的时候，会考虑是用结构体还是封闭性更好的类来实现。也会为了更好的展示进程运行结果设计用户友好的输出提示。最后，本次实验也帮我更好的掌握了两种调度算法的差异和效率高低。