湖南大學
HUNAN UNIVERSITY

操作系统实验报告

课 程 名 称：操作系统
实验项目名称：操作系统内核编程实验
专 业 班 级：软件 2203
姓　　名：白旭
学　　号：202226010306
指 导 教 师：周军海
完 成 时 间：2024 年 6 月 6 日

信息科学与工程学院

实验题目：实验八 进程调度

实验目的：

- **使用高级编程语言完成一个程序**
- **加深对进程控制块、进程队列等概念的了解**
- **掌握优先数调度算法和时间片调度算法的具体实施方式**
- **思考题：**
  如果要你设计进程调度算法，需要考虑什么因素？

实验环境：

- Vscode
- PowerShell
- **G++编译环境**

实验内容及操作步骤：

1. 编写代码

   1） pri.cpp

   头文件和命名空间

```cpp
#include <iostream>
#include <string>
#include <iomanip> // 包含iomanip头文件以使用setw

using namespace std;
```

   定义了一个 PCB 类，表示一个进程控制块

   包含进程名、优先级、已用 CPU 时间、总需要 CPU 时间、进程状态和指向下一个 PCB 的指针；构造函数用于初始化这些成员变量

```cpp
class PCB
{
public:
    string name;  // 进程名
    int pri;      // 优先级
    int cpuTime;  // 已用CPU时间
    int needTime; // 需要的总CPU时间
    char state;   // 进程状态，Ready、Run、Finish
    PCB *next;    // 指向下一个PCB的指针

    PCB(string n, int p, int nt)
        : name(n), pri(p), cpuTime(0), needTime(nt), state('W'),
next(nullptr) {}
};
```

定义了全局变量以表示就绪队列、运行中的进程和已完成的进程队列；numOfProcesses 保存进程数量

```
PCB *readyQueue = nullptr;
PCB *runningProcess = nullptr;
PCB *finishedProcesses = nullptr;

int numOfProcesses;
```

函数声明

```
void getFirstPCB();                 // 获取就绪队列中的第一个进程
void outputStatus();                // 输出所有进程信息
void insertByPriority(PCB *pcb);    // 根据优先级将进程插入到就绪队列中
void moveToFinished(PCB *pcb);      // 将完成的进程移入到完成队列
void createProcessesWithPriority(); // 创建优先级调度的进程实例
void priorityScheduling();          // 执行优先级调度算法
```

主函数

负责获取进程数量、创建进程、执行调度算法并输出最终状态

```
int main()
{
    cout << "优先级调度算法" << endl;
    cout << "请输入进程的数量：";
    cin >> numOfProcesses;
    createProcessesWithPriority();
    priorityScheduling();
    outputStatus();
    return 0;
}
```

创建并初始化进程，并将其插入到就绪队列。

```
void createProcessesWithPriority()
{
    PCB *newPCB;
    string name;
    int pri, needTime;
    cout << "请输入进程名、需要的时间、优先级（每行一个进程）： \n";
    for (int i = 0; i < numOfProcesses; i++)
    {
        cin >> name >> needTime >> pri;
        newPCB = new PCB(name, pri, needTime);
        insertByPriority(newPCB);
    }
}
```

从就绪队列中获取第一个进程并设置其状态为 R（运行）

```
void getFirstPCB()
{
    runningProcess = readyQueue;
    if (readyQueue != nullptr)
    {
        runningProcess->state = 'R';
        readyQueue = runningProcess->next;
        runningProcess->next = nullptr; // 断开与原就绪队列的链接
    }
}
```

输出就绪队列、完成队列和当前运行进程的状态

```
void outputStatus()
{
    PCB *currentPCB;
    // 设置输出格式头部
    cout << left << setw(10) << "pcbName" << setw(10) << "priority"
<< setw(10) << "cpuTime" << setw(10) << "needTime" << setw(10) <<
"proState" << endl;
    cout << "就绪队列" << endl;

    // 遍历并打印就绪队列
    for (currentPCB = readyQueue; currentPCB != nullptr; currentPCB =
 currentPCB->next)
    {
        cout << left << setw(10) << currentPCB->name
             << setw(10) << currentPCB->pri
             << setw(10) << currentPCB->cpuTime
             << setw(10) << currentPCB->needTime
             << setw(10) << currentPCB->state
             << endl;
    }

    cout << "完成队列" << endl;
    for (currentPCB = finishedProcesses; currentPCB != nullptr;
currentPCB = currentPCB->next)
    {
        cout << left << setw(10) << currentPCB->name
             << setw(10) << currentPCB->pri
             << setw(10) << currentPCB->cpuTime
             << setw(10) << currentPCB->needTime
             << setw(10) << currentPCB->state
             << endl;
    }

    cout << "当前运行" << endl;
    if (runningProcess != nullptr)
    {
        cout << left << setw(10) << runningProcess->name
             << setw(10) << runningProcess->pri
             << setw(10) << runningProcess->cpuTime
             << setw(10) << runningProcess->needTime
             << setw(10) << runningProcess->state
             << endl;
    }

    cout << "----------------------------------------------------"
<< endl;
}
```

根据优先级将进程插入到就绪队列的适当位置

```
void insertByPriority(PCB *pcb)
{
    pcb->state = 'W'; // 初始状态为等待
    PCB *firstPCB, *nextPCB;
    if (readyQueue == nullptr)
    {
        pcb->next = readyQueue;
        readyQueue = pcb;
    }
    else
    {
        firstPCB = nextPCB = readyQueue;
        if (pcb->pri < firstPCB->pri)
        {
            pcb->next = readyQueue;
            readyQueue = pcb;
        }
        else
        {
            while (firstPCB->next != nullptr && pcb->pri >= firstPCB
->pri)
            {
                nextPCB = firstPCB;
                firstPCB = firstPCB->next;
            }
            if (firstPCB->next == nullptr)
            {
                pcb->next = nullptr;
                nextPCB->next = pcb;
            }
            else
            {
                nextPCB->next = pcb;
                pcb->next = firstPCB;
            }
        }
    }
}
```

将完成的进程移入到完成队列

```
void moveToFinished(PCB *pcb)
{
    PCB *firstPCB = finishedProcesses;
    if (finishedProcesses == nullptr)
    {
        pcb->next = finishedProcesses;
        finishedProcesses = pcb;
    }
    else
    {
        while (firstPCB->next != nullptr)
            firstPCB = firstPCB->next;
        pcb->next = nullptr;
        firstPCB->next = pcb;
    }
}
```

实现优先级调度算法，每个时间片后提高优先级并更新进程状态。

```
void priorityScheduling()
{
    int flag = 1;
    getFirstPCB();
    while (runningProcess != nullptr)
    {
        outputStatus();
        while (flag)
        {
            runningProcess->pri += 2; // 每个时间片后优先级+2
            if (runningProcess->needTime <= 10)
            {
                runningProcess->state = 'F';
                runningProcess->cpuTime += runningProcess->needTime;
                runningProcess->needTime = 0;
                moveToFinished(runningProcess);
                flag = 0;
            }
            else
            {
                runningProcess->state = 'W';
                runningProcess->cpuTime += 10;
                runningProcess->needTime -= 10;
                insertByPriority(runningProcess);
                flag = 0;
            }
        }
        flag = 1;
        getFirstPCB(); // 取下一个进程继续调度
    }
}
```

2） round.cpp

头文件和命名空间

```
#include <iostream>
#include <string>
#include <iomanip> // 包含iomanip头文件以使用setw

using namespace std;
```

PCB 类表示一个进程控制块，包含进程名、已用 CPU 时间、需要的总时间、进程状态和指向下一个 PCB 的指针；构造函数初始化这些成员变量

```
class PCB
{
public:
    string name;  // 进程名
    int cputime;  // CPU已用时间
    int needtime; // 需要的总时间
    char state;   // 进程状态，Ready、Run、Finish
    PCB *next;    // 指向下一个PCB的指针

    PCB(string name, int needtime)
        : name(name), cputime(0), needtime(needtime), state('W'), next
(nullptr) {}
};
```

定义了全局变量来表示就绪队列、正在运行的进程和已完成的进程队列；numOfProcesses 存储进程数量，timeSlice 存储时间片大小

```
PCB *readyQueue = nullptr;
PCB *runningProcess = nullptr;
PCB *finishedProcesses = nullptr; // 定义就绪队列、运行队列和完成队列

int numOfProcesses;
int timeSlice;
```

函数声明

```
void getFirstPcb();                 // 从就绪队列中获取第一个进程
void outputStatus();                // 输出所有队列中的进程信息
void insertByPriority(PCB *pcb);    // 按优先级将进程插入就绪队列
void moveToFinished(PCB *pcb);      // 将进程移入完成队列
void createProcessesWithPriority(); // 创建进程并初始化
void priorityScheduling();          // 执行优先级调度算法
```

主函数负责获取进程数量和时间片大小，创建进程，执行调度算法，并输出最终状态

```
int main()
{
    cout << "优先级调度算法" << endl;
    cout << "请输入进程的数量：";
    cin >> numOfProcesses;
    cout << "请输入固定的时间片大小：";
    cin >> timeSlice;
    createProcessesWithPriority();
    priorityScheduling();
    outputStatus(); // 最后输出所有进程的状态
    return 0;
}
```

创建并初始化进程，将其插入到就绪队列中

```
void createProcessesWithPriority()
{
    string name;
    int needtime;
    cout << "请输入进程名和需要的时间（每行一个进程）：" << endl;
    for (int i = 0; i < numOfProcesses; i++)
    {
        cin >> name >> needtime;
        PCB *newPcb = new PCB(name, needtime);
        insertByPriority(newPcb);
    }
}
```

从就绪队列中获取第一个进程并设置其状态为 R（运行）

```
void getFirstPcb()
{
    runningProcess = readyQueue;
    if (readyQueue != nullptr)
    {
        runningProcess->state = 'R';
        readyQueue = readyQueue->next;
        runningProcess->next = nullptr;
    }
}
```

输出就绪队列、完成队列和当前运行进程的状态

```
void outputStatus()
{
    PCB *currentPcb;
    cout << left << setw(10) << "pcbName" << setw(10) << "cpuTime" <<
 setw(10) << "needTime" << setw(10) << "proState" << endl;
    cout << "就绪队列" << endl;
    for (currentPcb = readyQueue; currentPcb != nullptr; currentPcb =
 currentPcb->next)
    {
        cout << left << setw(10) << currentPcb->name << setw(10) <<
currentPcb->cputime << setw(10) << currentPcb->needtime << setw(10)
<< currentPcb->state << endl;
    }

    cout << "完成队列" << endl;
    for (currentPcb = finishedProcesses; currentPcb != nullptr;
currentPcb = currentPcb->next)
    {
        cout << left << setw(10) << currentPcb->name << setw(10) <<
currentPcb->cputime << setw(10) << currentPcb->needtime << setw(10)
<< currentPcb->state << endl;
    }

    cout << "正在运行" << endl;
    if (runningProcess != nullptr)
    {
        cout << left << setw(10) << runningProcess->name << setw(10)
<< runningProcess->cputime << setw(10) << runningProcess->needtime <<
 setw(10) << runningProcess->state << endl;
    }
    cout << "------------------------------------------------"
<< endl;
}
```

根据剩余时间将进程按优先级插入到就绪队列的适当位置

```
void insertByPriority(PCB *pcb)
{
    pcb->state = 'W';
    PCB *firstPcb;
    PCB *nextPcb;
    if (readyQueue == nullptr)
    {
        pcb->next = readyQueue;
        readyQueue = pcb;
    }
    else
    {
        firstPcb = nextPcb = readyQueue;
        if (pcb->needtime < firstPcb->needtime)
        {
            pcb->next = readyQueue;
            readyQueue = pcb;
        }
        else
        {
            while (firstPcb->next != nullptr && pcb->needtime >=
firstPcb->needtime)
            {
                nextPcb = firstPcb;
                firstPcb = firstPcb->next;
            }
            if (firstPcb->next == nullptr)
            {
                pcb->next = nullptr;
                nextPcb->next = pcb;
            }
            else
            {
                nextPcb->next = pcb;
                pcb->next = firstPcb;
            }
        }
    }
}
```

将完成的进程移入完成队列

```
void moveToFinished(PCB *pcb)
{
    PCB *firstPcb = finishedProcesses;
    if (finishedProcesses == nullptr)
    {
        pcb->next = finishedProcesses;
        finishedProcesses = pcb;
    }
    else
    {
        while (firstPcb->next != nullptr)
        {
            firstPcb = firstPcb->next;
        }
        pcb->next = nullptr;
        firstPcb->next = pcb;
    }
}
```

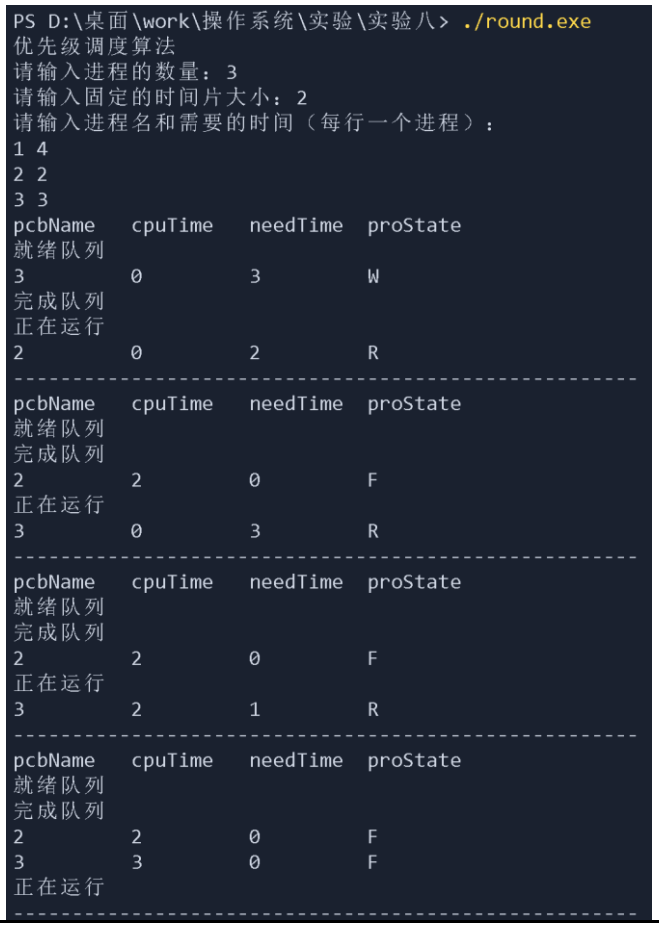
实现优先级调度算法：每个时间片后更新进程的已用时间和剩余时间，并根据进程状态将其重新插入就绪队列或移入完成队列

```
void priorityScheduling()
{
    int flag = 1;
    getFirstPcb();
    while (runningProcess != nullptr)
    {
        outputStatus();
        while (flag)
        {
            if (runningProcess->needtime <= timeSlice)
            {
                runningProcess->state = 'F';
                runningProcess->cputime += runningProcess->needtime;
                runningProcess->needtime = 0;
                moveToFinished(runningProcess);
                flag = 0;
            }
            else
            {
                runningProcess->state = 'W';
                runningProcess->cputime += timeSlice;
                runningProcess->needtime -= timeSlice;
                insertByPriority(runningProcess);
                flag = 0;
            }
        }
        flag = 1;
        getFirstPcb();
    }
}
```

2. 模块功能验证

1) pri

2) round



实验结果及分析：

1. **动态优先级调度算法过程分析：**

1) 第一阶段：

运行进程 1：因为它的优先级较低（3），优先执行优先级较高的进程。

更新状态：

进程 1 开始执行，状态为 R，更新 cpuTime 和 needTime。

2) 第二阶段：

继续运行进程 1：完成进程 1 的所有需要时间。

更新状态：

进程 1 完成执行，状态变为 F。

进程 2 开始执行，因为它的优先级较高（5）。

3) 第三阶段：

运行进程 2：完成进程 2 的所有需要时间。

更新状态：

进程 2 完成执行，状态变为 F。

所有进程均已完成。

2. **时间片轮转调度算法：**

1) 第一阶段：

运行进程 2：因为它的需要时间最短（2），所以优先执行。

更新状态：

进程 2 的状态变为 R，执行 2 个时间片后，进程 2 完成（F）。

就绪队列中只有进程 3。

2) 第二阶段：

运行进程 3：因为进程 2 已完成，所以调度进程 3。

更新状态：

进程 3 开始执行，消耗 2 个时间片，剩余 1 个时间片。

进程 3 继续保持运行状态，状态为 R。

3) 第三阶段：

继续运行进程 3：因为进程 3 还有剩余的 1 个时间片。

更新状态：

进程 3 完成执行（F）。

此时，所有进程均已完成。

收获与体会：

1. 运行时因为输出中使用了中文，一开始出现了很多乱码，后使用终端命令：[Console]::OutputEncoding = [System.Text.Encoding]::UTF8

   将编码标准改为 UTF-8 后正常输出

2. 加深了对进程控制块、进程队列等概念的了解
3. 掌握了优先数调度算法和时间片调度算法的具体实施方法

思考题：

1. 如果要你设计进程调度算法，需要考虑什么因素？

   1) 公平性

      确保所有进程都有机会获得 CPU 资源，不会被饿死

   2) 效率

      最大化 CPU 的利用率，尽量减少 CPU 空闲时间

   3) 响应时间

      确保进程在提出请求后能够尽快得到响应

   4) 负载平衡

      尽量避免某些 CPU 负载过重而导致其他 CPU 空闲的情况

实验成绩	