进程调度算法实验报告

课程名称： 操作系统 **指导老师：** _____ **完成时间：** 2022.11.21
班　级　： _____ **学　号　：** 2 **姓　名　：** _____

一、实验内容

用高级编程语言实现按优先数对处理器进行调度的模拟过程。

二、实验目的

1、了解多道程序系统的工作特点；

2、认识常用的进程调度的算法；

3、用高级编程语言实现模拟处理器调度算法。

三、实验原理

1、进程调度算法有很多种，其实现方法和原理各不相同，具有不同的复杂度和功能特点，此处选用原理较为简单的按照优先数进行处理器调度的算法。

2、定义相关的数据结构：若一个进程包含有进程名称、进程状态、要求运行时间、优先数等信息，则可以定义一个进程控制块的类，其包含上述属性和值。

3、为了便于每一次对进程的操作，在这里使用队列的数据结构进行操作。

4、为了便于把每一个进程控制块连接起来形成队列，除了需要在进程控制块类里面加入上述进程所包含的元素，还需要加入指向其后续进程的指针或者是引用。

5、综上，一个进程控制块类的结构定义如下：

进程名称	要求时间	优先数	程序状态	下一进程指针

其中：

(1)进程名称：即当前进程控制块表示的进程的名称；

(2)要求时间：当前进程距完成还需要的 CPU 时间；

(3)优先数：按照程序的优先数进行处理器调度，优先数高的进程优先享受

处理器调度；

(4)程序状态：当前进程控制块所对应的进程的状态，其有两种状态：“R”表示正在运行（Run），“E”表示运行结束（End）；

(5)下一进程指针：用于构成队列，当该指针指向 Null 或者为 None 的时候表示已到队尾。

6、调度算法

每个进程的优先数和要求时间由控制台输入，伊始每个进程的状态都设为“R”表示正在运行。同时，在完成节点的创建之后，将该节点插入到队列当中，使得队列从对头到队尾按照每一进程优先数来进行降序排列，此处需要用到数据结构预算法当中关于队列的知识。

运行进程时应总是从优先数最大的进程处开始执行，易知，队首进程的优先数一定是最大的，则当运行进程的时候，只需要操作队首进程，需要执行以下操作：

（1）、进程优先数减一、进程所需时间减一；

（2）、检查当前队首进程是否已完成。若完成则将其状态改为“E”，输出已完成提示，移出队列；否则应将修改后的队首进程从队列当中删除再插入到合适的位置；

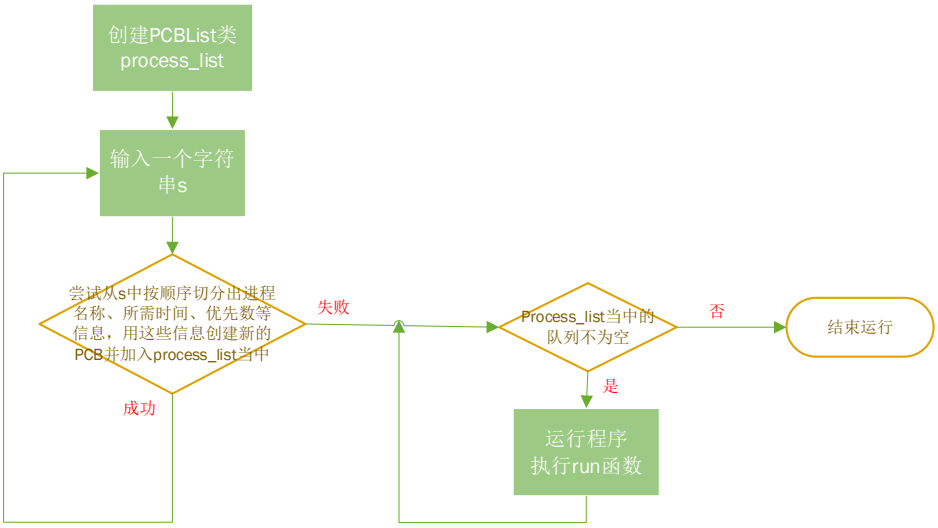
（3）、输出当前剩余的进程队列情况。

若进程队列不为空，则一直循环执行，直到执行完毕所有的进程，即：进程队列为空

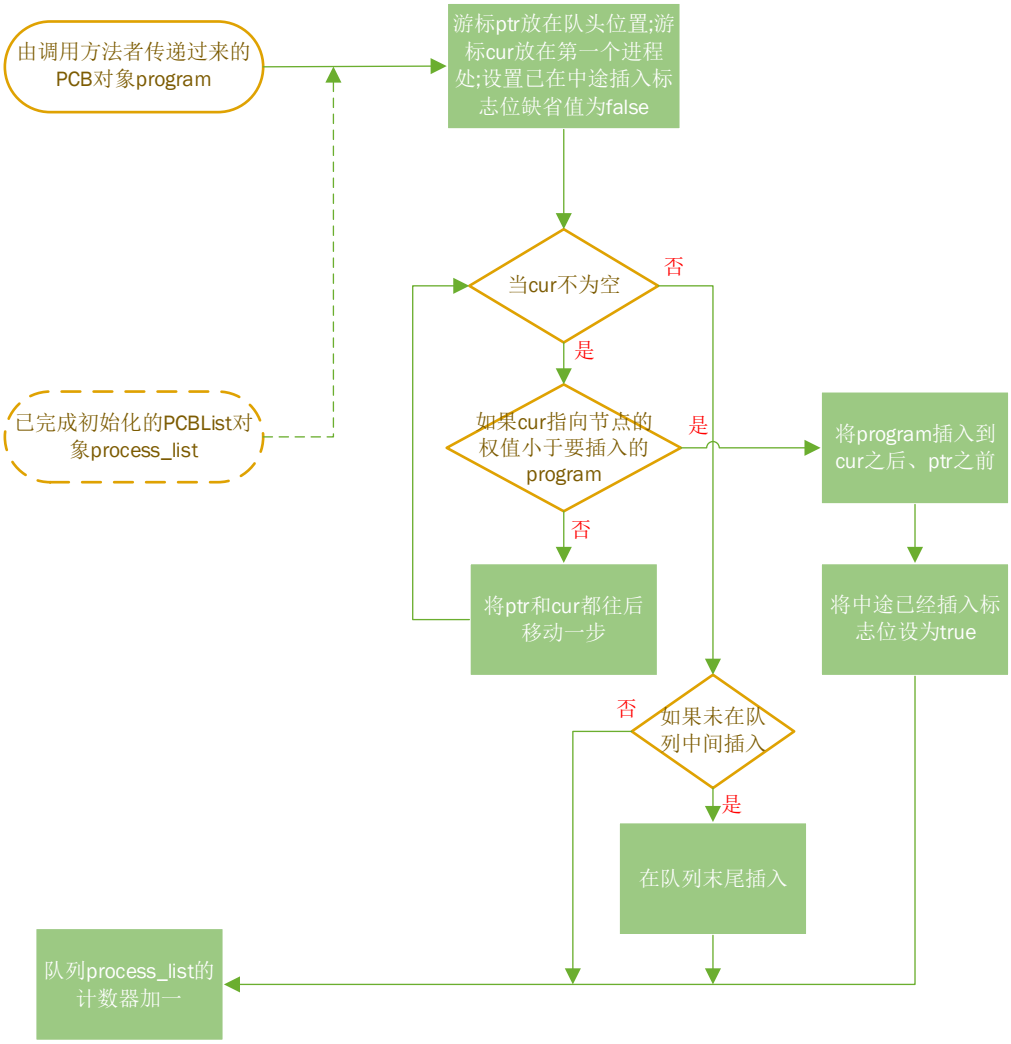
四、实验报告

1、算法流程图

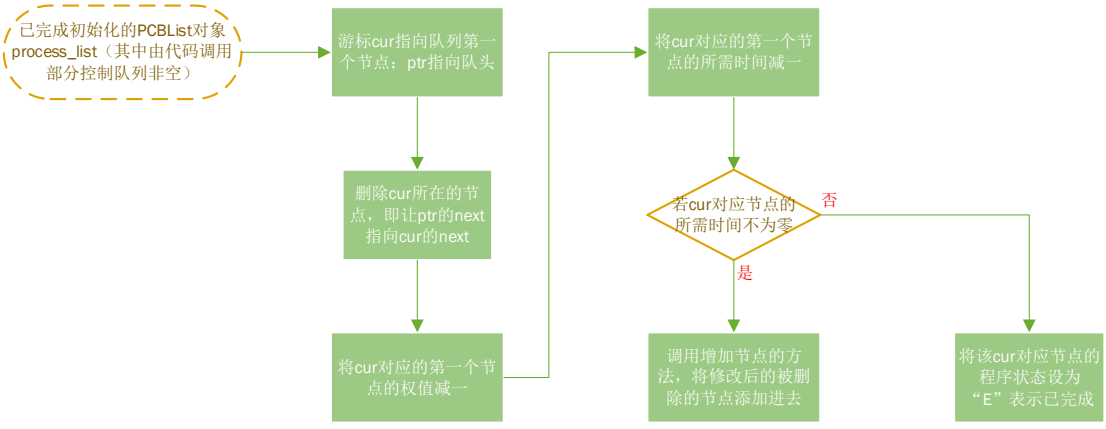
（1）、程序入口



（2）、添加新的 PCB 到队列当中



（3）、按照权值大小运行程序



2、程序中所使用到的数据结构

（1）、定义 PCB 类

在实现进程调度的算法时，采用了 python3.6 作为编程语言来实现。为了能够更加清晰的组织和观察每个进程之间的关系和每个进程的详细信息，定义了一个 PCB 类，如下：

```
class PCB:
    def __init__(self, program_name: str, time_require: int, priority: int,
pointer=None):
        self.program_name = program_name
        self.time_require = time_require
        self.priority = priority
        self.state = 'R'
        self.next = pointer

    @property
    def get_priority(self):
        return self.priority

    def __str__(self):
        return f"{self.program_name}: time_require={self.time_require}
priority={self.priority} state={self.state}"
```

可以看到，其中：

a、在初始化方法中，提供了程序名称、要求运行时间、权值等参数，next 的缺省值为 None，即为空；

b、初始化方法中，程序的状态，即 self.state 值初始化设定为“R”表示正在运行；

c、重写了 PCB 类的__str__()方法，输出某一个进程的信息时更加方便。

(2)、定义 PCBList 类

在有了 PCB 类后，为能够方便管理每一个进程控制块以及各方法间访问数据，创建一个 PCBList 类，其中封装了多个实现功能的方法及相应的数据。定义类如下：

```
class PCBList:
    def __init__(self):
        self.ready_list_header = PCB("header", -1, -1)
        self.programs_num = 0
        self.running_counter = 0
        self.finished = None

    def add_program(self, program: PCB):
        ...
    def get_ready_list(self):
        ...
```

```
    def get_finished_list(self):
        ...
    def run(self):
        ...
```

可以看到，其中：

a、初始化方法没有传递任何参数，为系统调用执行；其中，将队列头节点用一个 PCB 对象填充，其值无意义，仅作为队列头节点使用；

b、初始化方法当中创建了一个进程数量统计值，初始值为 0；创建了一个总运行次数计数器，初始值为 0；创建了一个已完成队列，初始为 None（空）；

c、在类的方法中，有增加 PCB 的方法、获取当前已经进入就绪状态的进程的队列的方法、获取已经进入了完成状态的进程的队列的方法、运行已就绪的进程的方法。

（3）、使用队列这一数据结构

队列是一种特殊的线性表，其特点为仅允许单向的增加和删除，在此处使用队列这一数据结构，可以表现出进程之间排队等待处理器资源的特点。

但是，实验中所用的队列和数据结构当中的队列有一定的区别：实验中的队列总是从队首出队，但是入队未必是队尾，而是要按照优先数的大小进行排序，也就是说，实验中的队列是一种允许“插队”的特殊队列。

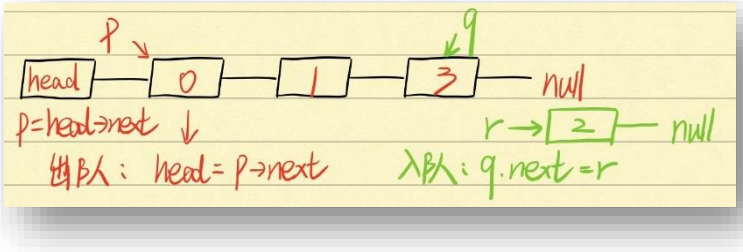


图 1 队列的入队和出队

3、程序源代码

为了能够便于观察到每一次运行的进程的名称以及运行完成的提示，故在相应的输出代码里面有控制台的转义字符序列，主要用来高亮文字，与程序逻辑无关。

```
class PCB:
    def __init__(self, program_name: str, time_require: int, priority: int,
pointer=None):
        # 用传递进来的值初始化 PCB 对象
        self.program_name = program_name
```

```
        self.time_require = time_require
        self.priority = priority
        # 缺省值：状态-"R"，下一节点-None
        self.state = 'R'
        self.next = pointer

    @property
    def get_priority(self):
        return self.priority

    # 重写 str 方法
    def __str__(self):
        return f"{self.program_name}: time_require={self.time_require}
priority={self.priority} state={self.state}"


class PCBList:
    def __init__(self):
        # 定义一个队头
        self.ready_list_header = PCB("header", -1, -1)
        # 初始化进程数量、运行次数、已完成队列
        self.programs_num = 0
        self.running_counter = 0
        self.finished = None

    # 方法：添加 PCB 对象到队列当中
    def add_program(self, program: PCB):
        # ptr 先行移动到队首
        ptr = self.ready_list_header
        # cur 紧随 ptr 后
        cur: PCB = ptr.next
        # 已经在中间插入标志，默认为假
        insert_in_middle = False
        # 一直扫描到队尾
        while cur is not None:
            # 若夫欲插入的进程节点的权值大于扫描到的进程节点的权值，则前插
            if cur.get_priority < program.get_priority:
                program.next = cur
                ptr.next = program
                # 已经在中途插入节点了
                insert_in_middle = True
                break
            else:
                ptr = cur
```

```
            cur = ptr.next
    # 如果没有在中途插入节点，证明需要尾插
    if not insert_in_middle:
        ptr.next = program
    # 计数器加一
    self.programs_num += 1

# 方法：获得就绪队列，主要用于输出就绪队列状态，返回字符串类型
def get_ready_list(self) -> str:
    # 新建一个空字符串
    string: str = ""
    # ptr 移动到队列中第一个进程的节点
    ptr: PCB = self.ready_list_header.next
    # 只要为空，意味着已完成
    if ptr is None:
        return "All program finished!"
    # 向队尾扫描，把每一个进程节点的信息加入到空字符串当中
    while ptr is not None:
        # 调用了 str 方法，上述的重载__str()__方法起到了作用
        string += f"{{{str(ptr)}}}, "
        ptr = ptr.next
    # 切片，把末尾多余的“， ”（逗号和空格）切掉
    return string[:-2]

# 方法：获取已完成的进程列表，返回字符串
def get_finished_list(self) -> str:
    # 基本思路同上
    string: str = ""
    ptr: PCB = self.finished
    if ptr is None:
        return "None program finished!"
    while ptr is not None:
        string += f"{{{str(ptr)}}}, "
        ptr = ptr.next
    return string[:-2]

# 方法：按队列运行
def run(self):
    # 每一次运行，给运行计数器加一
    self.running_counter += 1
    # ptr 移动到队首
    ptr: PCB = self.ready_list_header
    # cur 紧随其后
    cur: PCB = ptr.next
```

```
            # 输出相应的运行信息
            print(f"Run {self.running_counter} --> \033[1;35m{cur.program_name}\033[0m
is running...")
            # 将这个节点从队列出队（删除），但 cur 仍然指向这个节点
            ptr.next = cur.next
            # 对该节点进行所需时间和优先级减一
            cur.time_require += -1
            cur.priority += -1
            # 检查是否运行完成
            if cur.time_require == 0:
                # 如果是，修改状态为“E”
                cur.state = "E"
                # 把 cur 指向的节点加入到已完成队列
                cur.next = self.finished
                self.finished = cur
                # 输出提示信息
                print(f"\033[1;33m    | A new program finished:
{{{str(cur)}}}\033[0m")
            else:
                # 如果没有完成，将其后节点断开
                cur.next = None
                # 再重新入队，但是这里的入队是带有排序的入队
                self.add_program(cur)
            # 输出运行状态信息：就绪队列
            print("    | running state: ", self.get_ready_list())


# 函数：进程调度（实验名），组织和调用前面的类、对象、方法
def process_scheduling():
    # 先行建立一个队列
    process_list = PCBList()
    # 一直循环
    while True:
        try:
            # 尝试输入一行字符串，按照进程名称、所需时间、权值的方式以空格隔开并进
行切片，若切片成功证明输入为有效数据，再读取新的一行
            # 若切片失败说明输入了非有效数据，跳出死循环，执行后文
            s = input(" | input a new line string with program_name, time_require
and priority, split with space, "
                      "\n |__other format to break:")
            sl = s.split(" ")
            process_list.add_program(PCB(sl[0], int(sl[1]), int(sl[2])))
        except Exception:
            break
```

```
    # 输出最初的就绪队列
    print("Init state: ", process_list.get_ready_list())
    # 只要就绪队列不为空，就一直调用里面的运行进程 self.run()方法
    while process_list.ready_list_header.next is not None:
        process_list.run()


# 入口
if __name__ == '__main__':
    process_scheduling()
```

4、程序运行时的过程

（1）、输入

在使用测试用例时，选用了本实验指导书所提供的例子，即：

P1 2 1

P2 3 5

P3 1 3

P4 2 4

P5 4 2

（2）、输出：

```
| input a new line string with program_name, time_require and priority, split with space,

|__other format to break:p1 2 1

| input a new line string with program_name, time_require and priority, split with space,

|__other format to break:p2 3 5

| input a new line string with program_name, time_require and priority, split with space,

|__other format to break:p3 1 3

| input a new line string with program_name, time_require and priority, split with space,

|__other format to break:p4 2 4

| input a new line string with program_name, time_require and priority, split with space,

|__other format to break:p5 4 2
```

| input a new line string with program_name, time_require and priority, split with space,

|__other format to break:

Init state: {p2: time_require=3 priority=5 state=R}, {p4: time_require=2 priority=4 state=R}, {p3: time_require=1 priority=3 state=R}, {p5: time_require=4 priority=2 state=R}, {p1: time_require=2 priority=1 state=R}

Run 1 --> p2 is running...

| running state: {p4: time_require=2 priority=4 state=R}, {p2: time_require=2 priority=4 state=R}, {p3: time_require=1 priority=3 state=R}, {p5: time_require=4 priority=2 state=R}, {p1: time_require=2 priority=1 state=R}

Run 2 --> p4 is running...

| running state: {p2: time_require=2 priority=4 state=R}, {p3: time_require=1 priority=3 state=R}, {p4: time_require=1 priority=3 state=R}, {p5: time_require=4 priority=2 state=R}, {p1: time_require=2 priority=1 state=R}

Run 3 --> p2 is running...

| running state: {p3: time_require=1 priority=3 state=R}, {p4: time_require=1 priority=3 state=R}, {p2: time_require=1 priority=3 state=R}, {p5: time_require=4 priority=2 state=R}, {p1: time_require=2 priority=1 state=R}

Run 4 --> p3 is running...

| A new program finished: {p3: time_require=0 priority=2 state=E}

| running state: {p4: time_require=1 priority=3 state=R}, {p2: time_require=1 priority=3 state=R}, {p5: time_require=4 priority=2 state=R}, {p1: time_require=2 priority=1 state=R}

Run 5 --> p4 is running...

| A new program finished: {p4: time_require=0 priority=2 state=E}

| running state: {p2: time_require=1 priority=3 state=R}, {p5: time_require=4 priority=2 state=R}, {p1: time_require=2 priority=1 state=R}

Run 6 --> p2 is running...

| A new program finished: {p2: time_require=0 priority=2 state=E}

| running state: {p5: time_require=4 priority=2 state=R}, {p1: time_require=2 priority=1 state=R}

Run 7 --> p5 is running...

| running state: {p1: time_require=2 priority=1 state=R}, {p5: time_require=3 priority=1 state=R}

Run 8 --> p1 is running...

| running state: {p5: time_require=3 priority=1 state=R}, {p1: time_require=1 priority=0 state=R}

Run 9 --> p5 is running...

| running state: {p1: time_require=1 priority=0 state=R}, {p5: time_require=2 priority=0 state=R}

Run 10 --> p1 is running...

| A new program finished: {p1: time_require=0 priority=-1 state=E}

| running state: {p5: time_require=2 priority=0 state=R}

Run 11 --> p5 is running...

| running state: {p5: time_require=1 priority=-1 state=R}

Run 12 --> p5 is running...

| A new program finished: {p5: time_require=0 priority=-2 state=E}

| running state: All program finished!

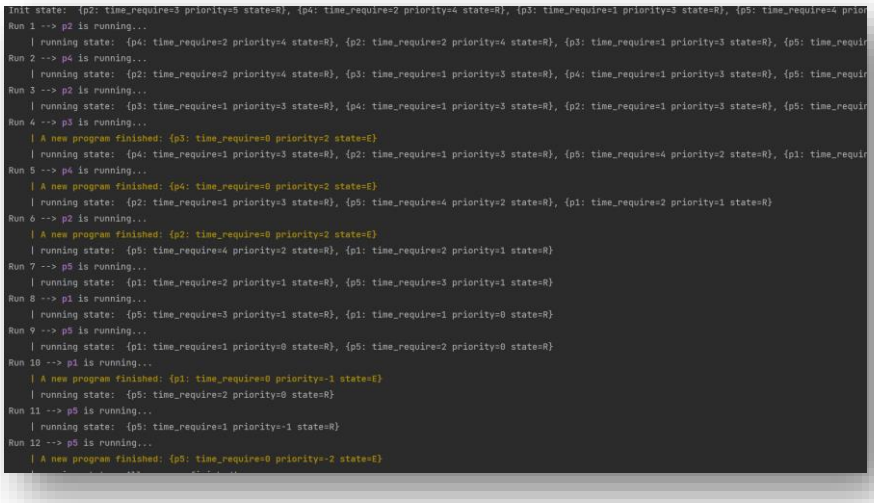


图 2 运行结果控制台输出效果

很明显，可以从程序输出的结果中发现，一共运行了 12 次，每一次运行的程序名称和相关属性、就绪队列、完成队列都有详细的输出；当有进程完成时，也会有输出提示。