

## 进程调度

栾效睿、饶钦臣、胡家铭

## 摘 要

在本课程设计工作中，我们编写设计了程序用来模拟实现操作系统的进程管理功能。主要工作内容包括：

1. Linux 进程管理源代码分析：研究和分析了 Linux 操作系统中的进程管理机制及其源代码，以深入理解其实现原理。
2. 模拟实现进程管理的基本功能：我们结合界面操作、多线程调用、信号量互斥，可以实现对进程的插入，动态运行、阻塞、唤醒、删除等。
3. 实现了几种常见的进程调度算法，如先来先服务调度算法、CFS 调度算法和轮转调度算法。
4. 进程调度算法性能比较：我们提供了对进程调度的周转时间、平均周转时间的计算与展示，可以对比各个算法的差异，分析不同情况下，各个调度算法的效率优劣。

# 目 录

摘 要 .....	1
<b>第一章 概述.....</b>	<b>3</b>
1.1 项目的背景与介绍 .....	3
1.2 项目的功能 .....	3
<b>第二章 原理概述与系统结构 .....</b>	<b>4</b>
2.1 基本概念和原理 .....	4
2.2 系统整体架构 .....	5
<b>第三章 CFS 调度算法的介绍与实现.....</b>	<b>6</b>
3.1 算法介绍 .....	6
3.2 CFS 的实现 .....	6
<b>第四章 FCFS 介绍与实现 .....</b>	<b>8</b>
4.1 算法介绍 .....	8
4.2 FCFS 的实现 .....	8
<b>第五章 RR 介绍与实现 .....</b>	<b>10</b>
5.1 算法介绍 .....	10
5.2 RR 的实现 .....	10
<b>第六章 动态阻塞、唤醒等实现 .....</b>	<b>12</b>
6.1 线程引入 .....	12
6.2 信号量引入 .....	12
<b>第七章 系统测试与分析 .....</b>	<b>14</b>
<b>第八章 不足与展望 .....</b>	<b>22</b>
参考资料 .....	23

# 第一章 概述

## 1.1 项目的背景与介绍

进程调度是 OS 内核的重要部分，调度算法的优秀与否关乎运行效率的高低，我组组员对调度问题颇感兴趣，故做此项目。

本项目使用 python 完成了对 FCFS、RR、CFS 等调度算法的模拟，另外我们结合界面操作、多线程调用、信号量互斥，可以实现对进程的插入，动态运行、阻塞、唤醒、删除等。同时，我们提供了对进程调度的周转时间、平均周转时间的计算与展示，可以对比各个算法的差异，分析不同情况下，各个调度算法的效率优劣。

## 1.2 项目的功能

1. 可以使用 CFS、FCFS、RR 算法中的任意一种，去模拟进程调度，你只需给出进程名称，到达时间，所需要的时间。
2. 可以选择一步步的运行，观察每一时间片，各个进程的变化，也可以选择一次全部运行，动态观察进程变化。
3. 运行的同时，你可以选择阻塞任何一个正在运行的进程，让其停下，当然，随后你可以将其唤醒。你也可以执行删除。
4. 最后在运行结束后，你可以看到有关周转时间、平均周转时间等的计算结果，你可以比较三个算法的不同，观察每个调度算法的特点，分析各个算法在不同数据下的性能表现。

## 第二章 原理概述与系统结构

### 2.1 基本概念和原理

#### 2.1.1. 进程创建与销毁

1. 创建：模拟系统中，进程由用户定义并添加到调度队列。每个进程有唯一的 PID、到达时间、需求时间和优先级等属性。
2. 销毁：当进程完成执行后，从调度队列中移除，释放其占用的资源。

#### 2.1.2. 调度算法实现

1. FCFS：将所有进程按到达时间排序，依次调度。该算法简单易实现，但响应时间较差。
2. RR：为每个进程分配固定的时间片，时间片到后切换到下一个进程。优点是能够为每个进程提供较好的响应时间。
3. CFS：为每个进程维护一个虚拟运行时间（vruntime），调度器总是选择 vruntime 最小的进程运行，确保每个进程公平地获得 CPU 时间。

#### 2.1.3. 进程状态转换

1. 新建到就绪：当进程创建时，进入就绪队列等待调度。
2. 就绪到运行：调度器选择一个就绪进程，将其状态设置为运行。
3. 运行到等待：当进程需要等待某个事件（如 I/O 操作），其状态转换为等待。
4. 等待到就绪：当等待事件完成，进程重新进入就绪队列。
5. 运行到终止：当进程执行完毕，其状态转换为终止，并从系统中移除。

#### 2.1.4. 调度过程

1. 添加进程：通过用户接口添加新进程，进程进入就绪队列。
2. 选择调度算法：根据用户选择的算法（如 FCFS、RR、CFS），设置调度策略。
3. 执行调度：调度器根据选择的算法从就绪队列中选择进程进行调度，模拟执行过程。
4. 状态更新：在模拟执行过程中，更新进程的状态（如从运行到等待、从等待到就绪）。

## 2.2 系统整体架构

### 1. 实体层

`Process` 进程块实体

`Schedule` 抽象类，所有算法类都继承并实现里面相应的函数

### 2. 界面层

`main_GUI` 界面，支持使用者通过界面交互实现直接插入，动态的删除、唤醒、阻塞进程，并可以挑选使用相应的算法，查看运行结果等。

### 3. 算法层

`FCFS`:

该类继承 `Schedule`，实现相应的抽象方法，实现对先来先服务算法的模拟。

`RR`:

该类继承 `Schedule`，实现相应的抽象方法，实现对轮转调度算法的模拟。

`CFS`:

该类继承 `Schedule`，实现相应的抽象方法，实现对 `CFS` 调度算法的模拟。

### 4. 控制层

`schelder` 类，与界面直接交互，处理界面发来的请求，并调用相应的算法。

## 第三章 CFS 调度算法的介绍与实现

### 3.1 算法介绍

#### 3.1.1 算法核心理念

CFS 的核心思想是将所有进程当作一个“大队列”来调度，尽量让每个进程按比例公平地获得 CPU 时间。每个进程的运行时间与它应得的 CPU 时间是根据进程的优先级（或称为 nice 值）来动态调整的，确保不会有进程长时间占用 CPU 资源，保证其他进程也能获得响应。

#### 3.1.2 算法工作原理

1. 进程的虚拟运行时间：每个进程都有一个 `vruntime` 值，表示它已经运行的相对时间。  
`vruntime` 值越小的进程，越优先被调度。虚拟运行时间是动态变化的，依赖于进程的 nice 值和它的 CPU 使用情况。
2. 红黑树管理进程：CFS 将所有等待调度的进程按 `vruntime` 排列在红黑树中。每次调度时，CFS 会选择 `vruntime` 最小的进程进行调度，即该进程获得下次执行的 CPU 时间。红黑树的结构确保了对进程选择的高效性（ $O(\log N)$  的复杂度）。
3. 时间片和公平性：CFS 并不像传统的轮转调度器那样为每个进程分配固定长度的时间片，而是根据 `vruntime` 动态决定哪个进程下一个获得 CPU 时间。每个进程消耗的 CPU 时间越多，它的 `vruntime` 值就会越大，这样它就会被调度得相对较少，从而保证其他进程有机会得到执行。
4. 基于负载的平衡：CFS 会根据系统的负载情况进行调度。例如，系统负载低时，CFS 会允许更多的进程并行执行；而在负载高时，它会调整进程的调度策略，减少进程的切换频率。

### 3.2 CFS 的实现

#### 3.2.1 数据结构设计

原版的 CFS 算法的实现借助了红黑树这一数据结构，每次快速找到最左边的节点，即

vruntime 最小的进程。这里在模拟时使用 list+heapq 来实现快速查找 vruntime 的最小进程。

1. nices = [ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024] #nice 值列表

2. weight = 32 # 运行权重

3. readyQue = [] # 就绪队列

4. not\_arrive = [] # 未到达的队列

5. blockList = [] # 阻塞队列

6. finishedList = [] # 完成队列

7. T = 0 # 总时间

8. time\_part = 1 # 时间片，默认为 1

### 3.2.2 核心代码

```
1
2 def cal_priority(self, pro)->int:
3     val = pro.runtime * self.weight / self.nices[(pro.nice % len(self.nices) + len(self.nices)) % len(self.nices)] # 进程优先级计算
4     pro.priority = val
5     return val
```

图 1 计算进程 vruntime 的函数

```
1 def get_next_process(self):# 获得这一时刻要运行的进程
2
3     # 将到时间的全部放进就绪队列
4     for pro in self.not_arrive:
5         if pro.arriveTime <= self.T:
6             pro.state = 'READY'
7             heapq.heappush(self.readyQue, (self.cal_priority(pro), pro))
8             self.not_arrive.remove(pro)
9
10    # 从就绪队列中找到优先级最高的
11    if self.readyQue:
12        priority, pro = heapq.heappop(self.readyQue)
13        pro.state = 'RUNNING'
14        return pro
15    return None
```

图 2 得到当前时刻，应该运行的进程



## 第四章 FCFS 介绍与实现

### 4.1 算法介绍

#### 4.1.1 核心理念

FCFS 调度算法的核心理念是：按照进程到达就绪队列的顺序来分配 CPU 时间，即先到达的进程先执行，后到达的进程等待前一个进程执行完毕。

#### 4.1.2 工作原理

1. 就绪队列： 所有待执行的进程按它们到达的顺序排队。在开始执行时，操作系统将选择队列中第一个进程来运行。
2. 执行顺序： 一旦某个进程被选择执行，它会持续占用 CPU，直到执行完成，才会由操作系统调度下一个进程。
3. 非抢占式： 一旦一个进程开始执行，系统不会中断它（即进程不会被抢占）。它会一直执行到完成。
4. FIFO 排序： 进程按照其到达时间的先后顺序排队。即首先到达的进程最先被执行。

### 4.2 FCFS 的实现

#### 4.2.1 数据结构

1. not\_arrived: 未到达的进程列表 (list)

存储所有尚未到达调度时间的进程。进程的到达时间大于当前时间  $T$ 。

2. readyQue: 就绪队列 (list)

存储所有处于就绪状态、等待被调度执行的进程。使用堆（优先队列）结构，基于进程的到达时间或唤醒时间进行排序，确保先到达的进程先被调度。

3. blockList: 阻塞队列 (list)

存储所有被阻塞的进程。这些进程因等待某些资源或事件而无法继续执行，需等待唤醒。

4. finishedList: 完成队列 (list)

存储所有已经完成执行的进程，记录其完成时间和最终状态。

5. `current_process`: 当前运行的进程 (Process)

指向当前正在执行的进程。如果没有进程在执行, 则为 `None`。

6. `T`: 当前时间 (int)

表示调度器的当前时间, 随着调度的进行逐步递增。

## 4.2.2 核心代码

```
1 # 1. 将到达时间为当前时间的进程加入就绪队列
2 for pro in self.not_arrived:
3     if pro.arriveTime <= self.T:
4         pro.state = 'READY'
5         heapq.heappush(self.readyQueue, (max(pro.arriveTime, pro.wakeTime), pro))
6         self.not_arrived.remove(pro)
```

图 3 按照唤醒时间和到达时间的最大值压入二叉堆

```
1 # 2. 如果没有当前运行的进程, 获取下一个进程
2 if self.current_process is None:
3     self.current_process = self.get_next_process()
4     if self.current_process:
5         if self.current_process.startTime == -1:
6             self.current_process.startTime = self.T # 记录实际开始时间
7             self.current_process.setStatus('RUNNING') # 设置为运行状态
8
9 # 3. 运行当前进程
10 if self.current_process:
11     self.current_process.runtime += 1
12     self.current_process.needTime -= 1
13
14 # 检查进程是否完成
15 if self.current_process.needTime == 0:
16     self.current_process.endTime = self.T + 1 # 设置完成时间
17     self.current_process.setStatus('FINISH') # 设置为完成状态
18     self.finishedList.append(self.current_process)
19     self.current_process = None
```

图 4 运行一个时间片

## 第五章 RR 介绍与实现

### 5.1 算法介绍

#### 5.1.1 核心理念

Round Robin(RR)调度算法的核心理念是:每个进程按照它们到达的顺序,轮流获得 CPU 使用权,每个进程的执行时间不会超过一个固定的时间片,一旦时间片耗尽,当前进程被中断,操作系统会将 CPU 分配给下一个进程,直到所有进程执行完成。

#### 5.1.2 工作原理

1. 时间片 (Time Quantum): 系统为每个进程分配一个固定的时间片,通常是几十毫秒到几百毫秒之间。时间片的大小是算法的一个重要参数,过小会导致频繁上下文切换,过大则接近于 FCFS,失去轮转的优势。
2. 进程队列: 所有就绪的进程被放入一个队列(通常是一个循环队列),等待 CPU 调度。
3. 轮转调度: 系统从就绪队列中选择队首的进程并为其分配 CPU 时间,运行直到该进程的时间片用完,或者该进程提前完成(如有 I/O 请求)。如果进程未在一个时间片内完成,操作系统会将该进程放回队列的末尾,等待下一轮调度。进程继续按照队列顺序被轮流调度,每个进程获得的 CPU 时间不超过一个时间片。
4. 上下文切换: 每次进程时间片用完后,操作系统会进行上下文切换,保存当前进程的状态,加载下一个进程的状态。
5. 进程完成: 如果进程在时间片内完成了执行,操作系统将该进程从队列中移除,不再调度。


### 5.2 RR 的实现

#### 5.2.1 数据结构

1. `self.readyQue = []` #就绪队列
2. `self.blockList = []` #阻塞队列
3. `self.finishedList = []` #完成队列

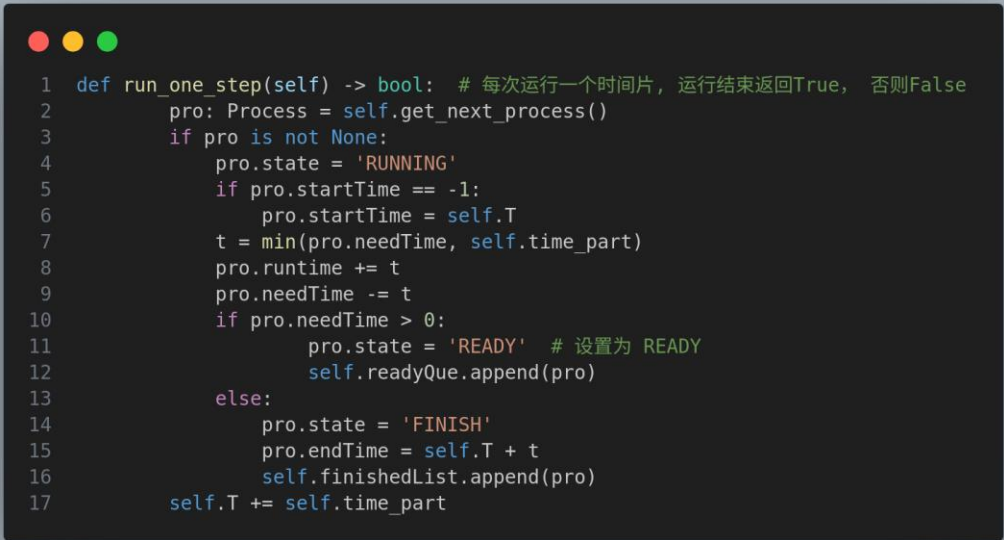
4. self.not\_arrive = [] # 未达到队列
5. self.T = 0 # 总时间
6. self.time\_part = time\_part #时间片大小

### 5.2.1 核心代码



```
1 def get_next_process(self): # 获得这一时刻要运行的进程
2     for pro in self.not_arrive:
3         if pro.arriveTime <= self.T:
4             pro.state = 'READY'
5             self.readyQueue.append(pro)
6             self.not_arrive.remove(pro)
7
8     if self.readyQueue:
9         return self.readyQueue.pop(0)
10    return None
```

图 5 获得当前时刻要运行的进程



```
1 def run_one_step(self) -> bool: # 每次运行一个时间片，运行结束返回True，否则False
2     pro: Process = self.get_next_process()
3     if pro is not None:
4         pro.state = 'RUNNING'
5         if pro.startTime == -1:
6             pro.startTime = self.T
7         t = min(pro.needTime, self.time_part)
8         pro.runtime += t
9         pro.needTime -= t
10        if pro.needTime > 0:
11            pro.state = 'READY' # 设置为 READY
12            self.readyQueue.append(pro)
13        else:
14            pro.state = 'FINISH'
15            pro.endTime = self.T + t
16            self.finishedList.append(pro)
17        self.T += self.time_part
```

图 6 运行一个时间片

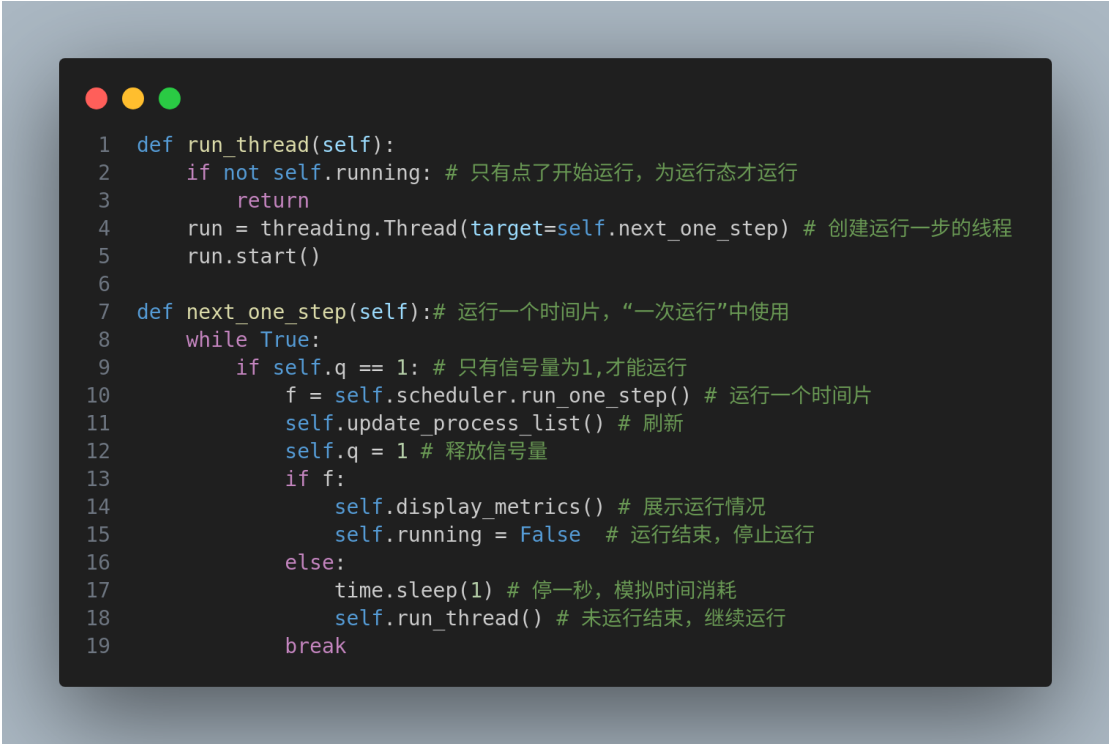
## 第六章 动态阻塞、唤醒等实现

### 6.1 线程引入

#### 6.1.1 实现原理

我们的程序会模拟程序在与运行，这一过程是动态的，同时可以在进程运行时，选择阻塞、唤醒等操作，这是如何互不影响的进行的呢？答案是使用多线程，借助 `thread` 函数，我们可以将运行作为线程分开出去，而不影响主线程的操作（允许运行同时点击阻塞或唤醒等）。

#### 6.1.2 核心代码



```
1 def run_thread(self):
2     if not self.running: # 只有点了开始运行，为运行态才运行
3         return
4     run = threading.Thread(target=self.next_one_step) # 创建运行一步的线程
5     run.start()
6
7 def next_one_step(self):# 运行一个时间片，“一次运行”中使用
8     while True:
9         if self.q == 1: # 只有信号量为1,才能运行
10            f = self.scheduler.run_one_step() # 运行一个时间片
11            self.update_process_list() # 刷新
12            self.q = 1 # 释放信号量
13            if f:
14                self.display_metrics() # 展示运行情况
15                self.running = False # 运行结束，停止运行
16            else:
17                time.sleep(1) # 停一秒，模拟时间消耗
18                self.run_thread() # 未运行结束，继续运行
19            break
```

图 7 分出线程运行“执行”

### 6.2 信号量引入

#### 6.2.1 实现原理

我们在算法调度过程中采用了，`heapq` 来实现快速找到优先级最高的进程，但是 `heapq` 是不支持多线程的，当我们执行运行进程时，会读写 `heapq`，与此同时不允许其他地方再读

写，否则会导致数据没法保证一致性。我们的解决思路是引入互斥量，当信号量为 1 时允许运行进程，当要执行阻塞、唤醒等操作时，首先抢占信号量，将信号量设为 0，然后再执行相应的操作，与此同时，运行是停止的，这就实现了互斥，不过这里更像是一种抢占式的互斥，只有需要执行阻塞、唤醒等操作时，才去要信号量，否则信号量一直为 1 为进程运行态。

### 6.2.2 核心代码



```
1 def block_process(self):# 阻塞进程
2     if not self.running:
3         return
4     self.q = 0    # 抢占信号量
5     time.sleep(0.2) # 避免同时操作heapq，保证时间片运行完整
6     if len(self.control_pid_entry.get()) > 0:
7         PID = self.control_pid_entry.get()
8         self.scheduler.stop(PID)
9         self.update_process_list()
10        self.q = 1    # 释放信号量
```

图 8 抢占信号

## 第七章 系统测试与分析

### 7.1 RR 测试分析

1. 只有一个进程，设置时间片为 1

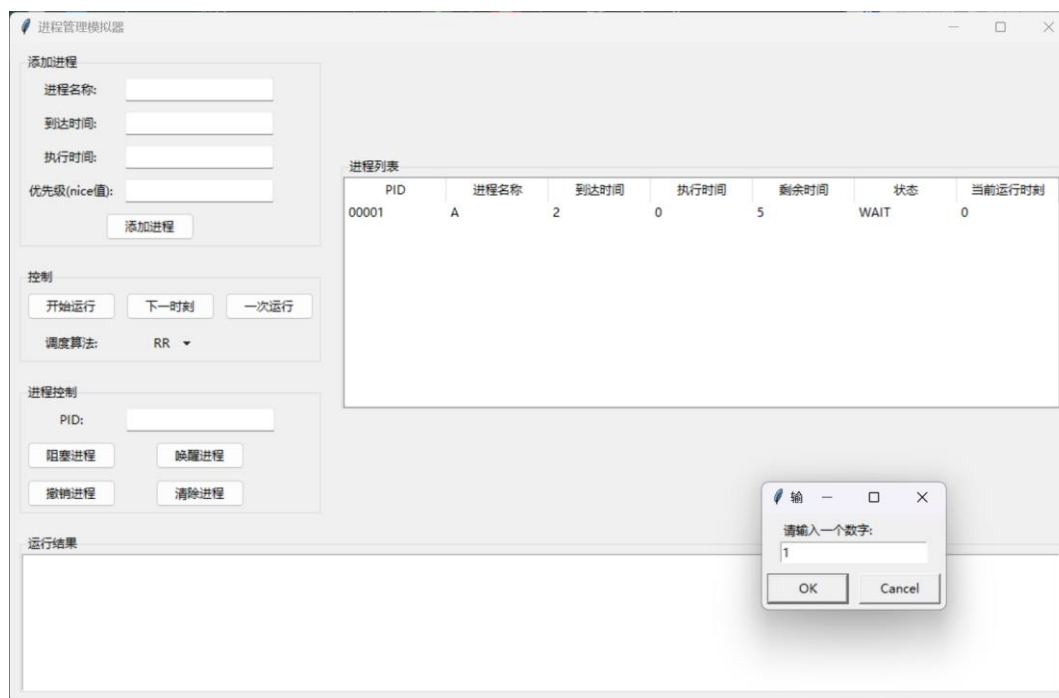


图 9 设置时间片

运行结果：

```
进程 00001:
进程名称 A
到达时间: 2
运行时间: 5.0
开始时间: 2.0
完成时间: 7.0
周转时间: 5.0
带权周转时间: 1.00
平均周转时间: 5.00
```

图 10 运行结果

2. 设置两个进程：

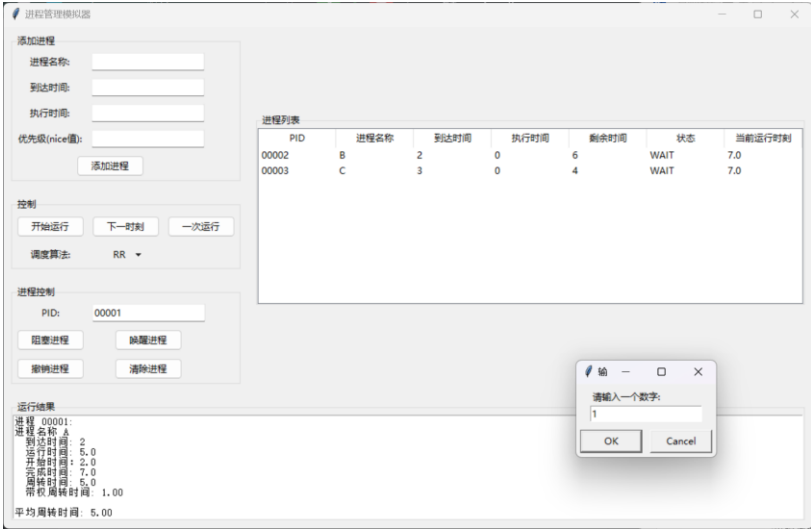


图 11 设置两个进程

运行时正确分配时间片：

PID	进程名称	到达时间	执行时间	剩余时间	状态	当前运行时刻
00003	C	3	0	4	RUNNING	4.0
00002	B	2	2.0	4.0	READY	4.0

图 12 运行过程

运行结果：

运行结果	
运行时间:	4.0
开始时间:	4.0
完成时间:	11.0
周转时间:	8.0
带权周转时间:	2.00
进程 00002:	
进程名称:	B
到达时间:	2
运行时间:	6.0

图 13 运行结果

3. 多进程运行

PID	进程名称	到达时间	执行时间	剩余时间	状态	当前运行时刻
00004	D	2	3.0	3.0	RUNNING	8.0
00006	F	3	1.0	8.0	READY	8.0
00005	E	2	2.0	2.0	READY	8.0

图 14 多个进程同时运行



运行结果：

运行结果	
进程 00004:	
进程名称 D	
到达时间: 2	
运行时间: 6.0	
开始时间: 2.0	
完成时间: 15.0	
周转时间: 13.0	
带权周转时间: 2.17	
进程 00006:	
进程名称 E	

图 15 运行结果

4. 进程阻塞

进程列表						
PID	进程名称	到达时间	执行时间	剩余时间	状态	当前运行时刻
00007	A	2	2.0	3.0	BLOCK	7.0

图 16 进程阻塞

5. 进程唤醒

进程列表						
PID	进程名称	到达时间	执行时间	剩余时间	状态	当前运行时刻
00008	A	2	2.0	6.0	READY	4.0

图 17 进程唤醒

6. 撤销进程

进程列表						
PID	进程名称	到达时间	执行时间	剩余时间	状态	当前运行时刻

图 18 撤销进程

## 7.2 FCFS 测试分析

1. 添加三个进程：Chrome、Wechat、QQ。到达时间、需要执行时间如图所示

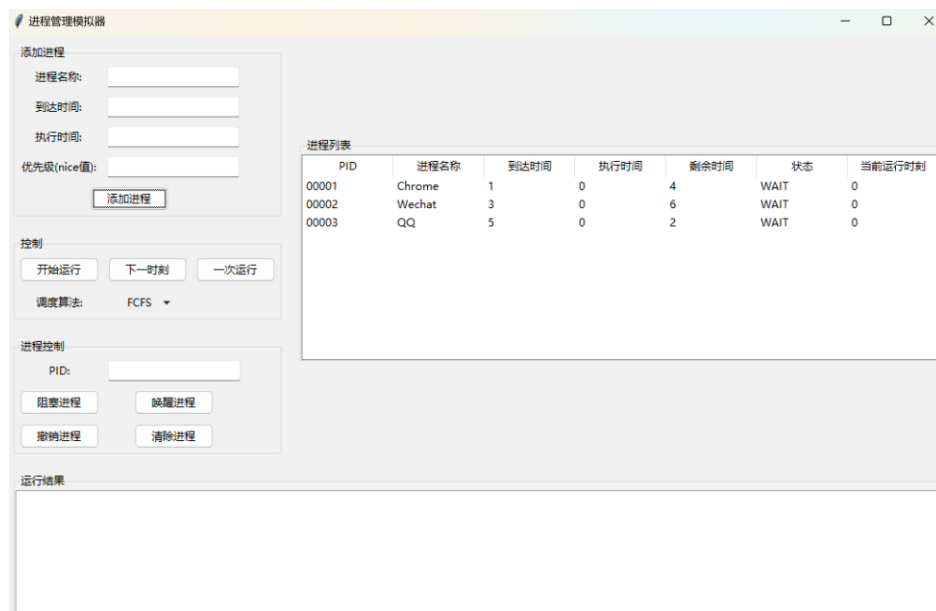


图 19 插入三个进程，准备测试

2. 运行至 4 时刻，Wechat 进程已到达，处于 READY 状态，Chrome 进程还需运行一个时间片

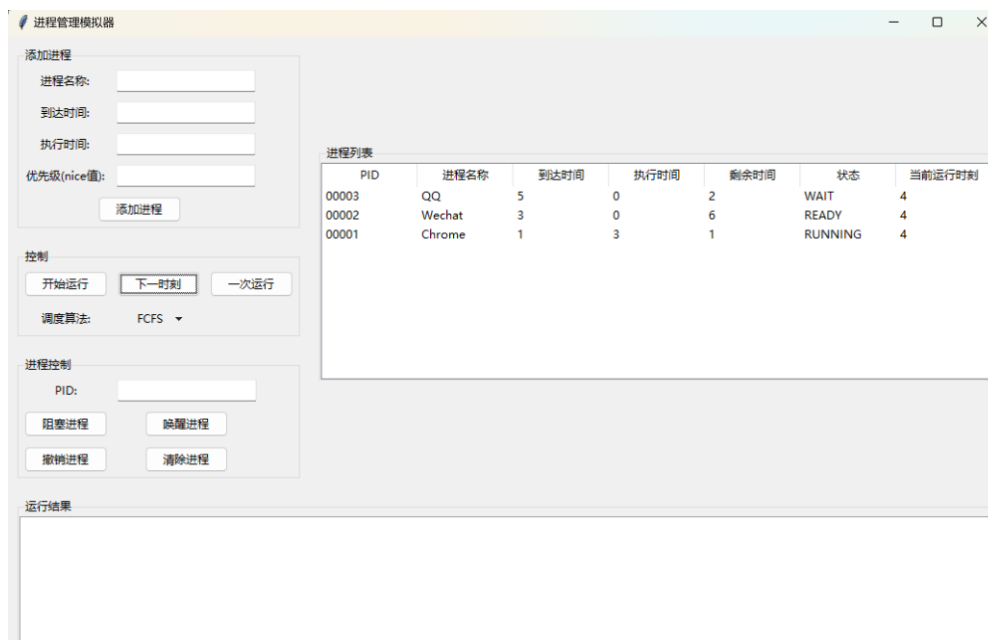


图 20 运行情况

3. 运行到 6 时刻，Chrome 进程在上一时刻完成运行，状态变为 FINISH；Wechat 进程开始

运行，状态变为 **RUNNING**；QQ 进程仍处于 **READY** 状态

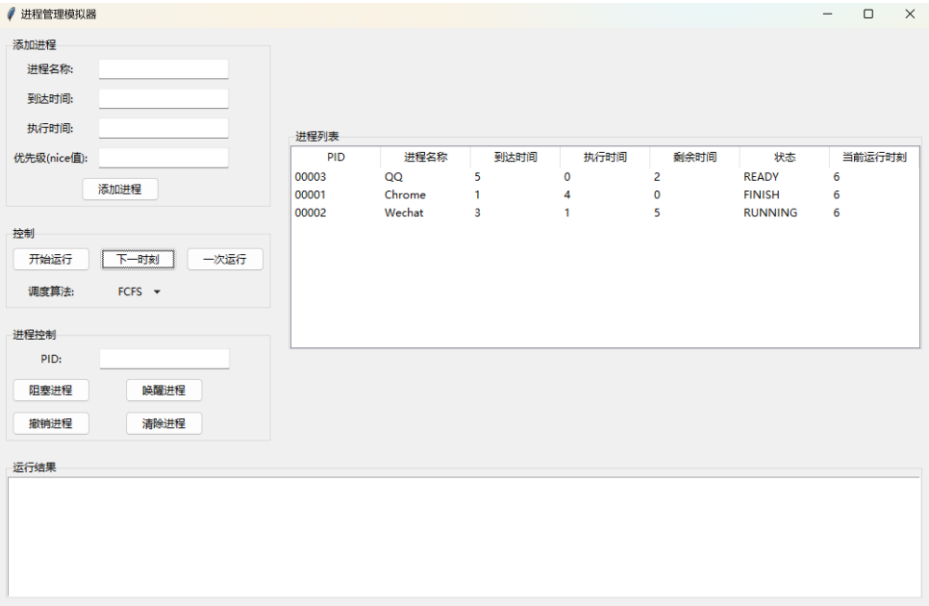


图 21 运行状况的分析

4. 运行到 11 时刻，Chrome 和 Wechat 进程都完成运行，状态为 **FINISH**；QQ 进程将在下一时刻开始运行

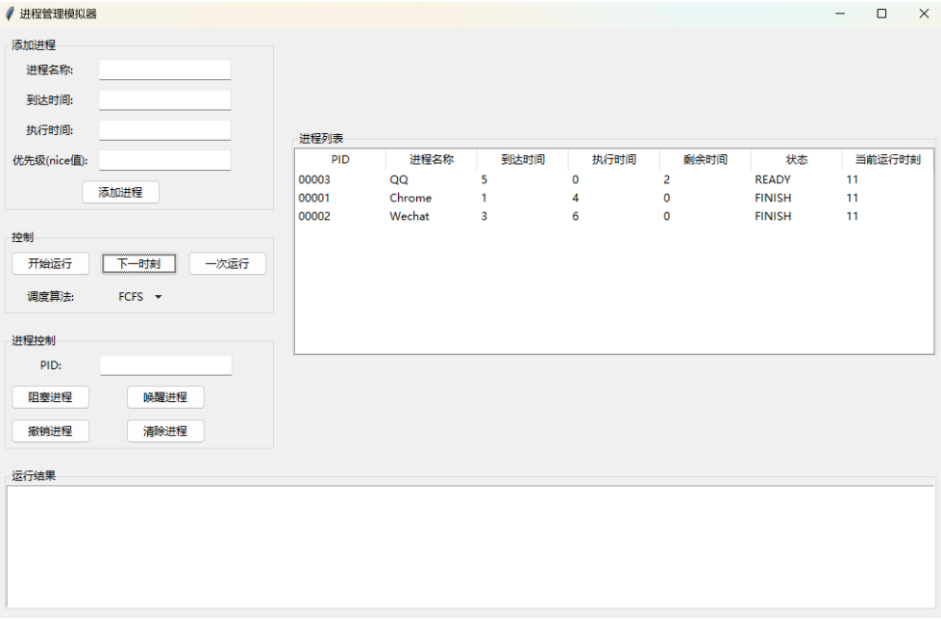


图 22 运行情况分析

5. 运行到 13 时刻，所有进程都已经完成运行，并计算得出运行结果

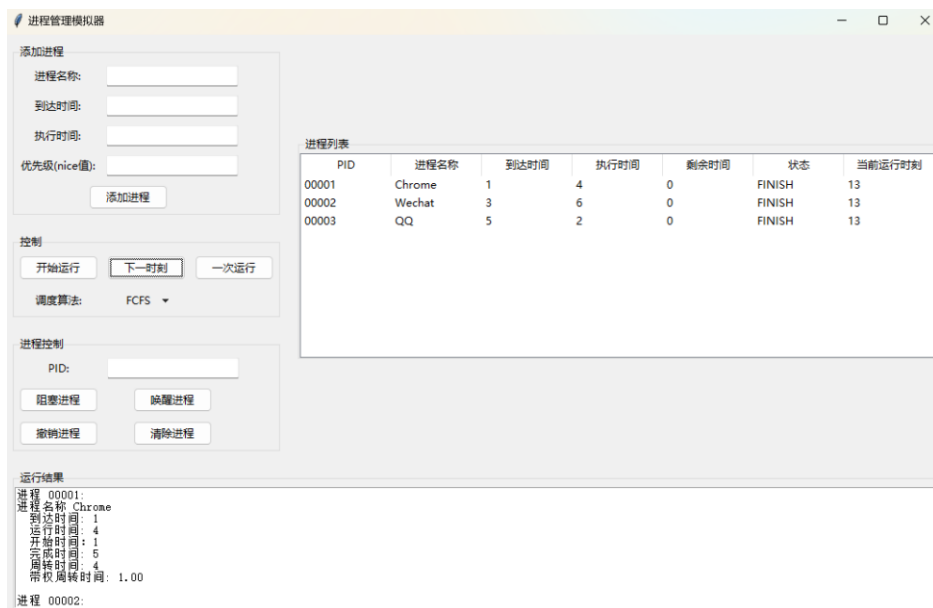


图 23 运行情况分析

运行结果：

运行结果	运行结果	运行结果
进程 00001: 进程名称 Chrome 到达时间: 1 运行时间: 4 开始时间: 1 完成时间: 5 周转时间: 4 带权周转时间: 1.00	进程 00002: 进程名称 Wechat 到达时间: 3 运行时间: 6 开始时间: 5 完成时间: 11 周转时间: 8 带权周转时间: 1.33	进程 00003: 进程名称 QQ 到达时间: 5 运行时间: 2 开始时间: 11 完成时间: 13 周转时间: 8 带权周转时间: 4.00 平均周转时间: 6.67

图 24-26 运行情况

## 7.3 CFS 测试分析

### 1. 测试运行:

正常运行

进程列表						
PID	进程名称	到达时间	执行时间	剩余时间	状态	当前运行时刻
00003	steam	5	8	12	RUNNING	36
00002	weixin	15	10	0	FINISH	36
00001	qq	1	10	0	FINISH	36

图 27 运行测试结果

### 2. 测试阻塞:

阻塞成功

进程列表						
PID	进程名称	到达时间	执行时间	剩余时间	状态	当前运行时刻
00002	weixin	15	0	10	waiting	15
00001	aa	1	7	3	RUNNING	15
00003	steam	5	7	13	BLOCK	15

图 28 阻塞测试结果

### 3. 测试唤醒:

正常唤醒

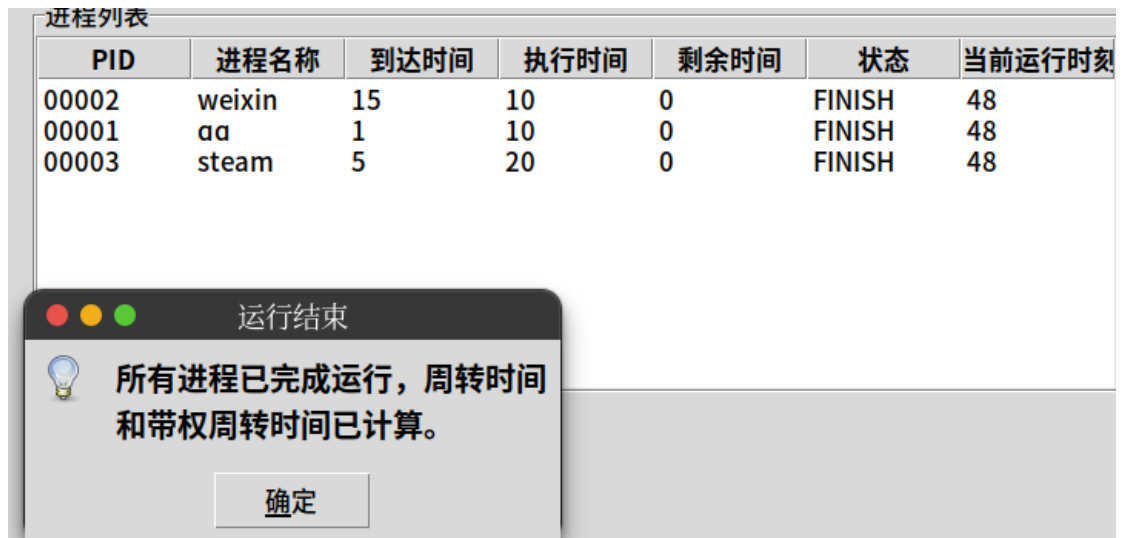


图 29 唤醒之后继续运行

### 4. 测试结果显示:

结果正常显示

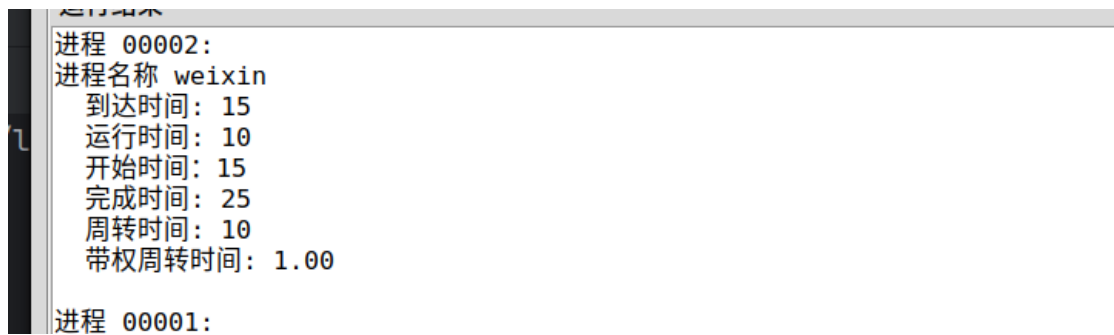


图 30 运行结果展示

## 第八章 不足与展望

### 8.1 设计不足与反思

1. 代码写的不完全符合分层的逻辑：我观察了一下界面代码，发现少量应该出现在 `control` 层的逻辑处理写在了界面里，需要进行一些优化和改动。
2. 代码不够健壮：我们关于输入输出，以及界面的按钮控制，做了一些容错，但是还是有不健全的地方，另外，我们的测试也都是基于少量的范围很小的数据，还有待使用大量的复杂的数据来测试程序的可靠性。
3. 界面部分、代码逻辑是否合理。比如每次运行前要点一下准备运行，这样是否冗余，而且算法的选择定在什么时候，是否有更合理的逻辑设计。
4. 简便、友好度不够，有很多辅助功能可以加，比如阻塞输入框可以设置为下拉列表或者其他形式，可以直接选中当前运行进程，提高容错。

### 8.2 展望

1. 一方面，我们可以先把上面的不足之处做做处理。
2. 此外，我们可以以此为基础实现一个在线进程模拟平台：
  - a) 以网页的形式发布，类似在线计算平台
  - b) 提供良好的界面，让用户方便的挑选算法，来进行任意数据的模拟、测试。
  - c) 反馈运行结果，方便用户进一步的分析。
3. 同时，我们也可以考虑进一步分析运行结果，比较在相同数据下各个调度算法的性能，比较在不同数据范围下，各个调度算法的性能，结合数学层面，计算出一些建设性的结论或者证已有的推论。

## 参考资料

- [1] <https://zhuanlan.zhihu.com/p/372441187> 操作系统调度算法—CFS，完全公平调度器
- [2] [https://blog.csdn.net/smile\\_sundays/article/details/135706129](https://blog.csdn.net/smile_sundays/article/details/135706129) 多线程(看这一篇就够了,超详细,满满的干货)
- [3] <http://arthurchiao.art/blog/linux-cfs-design-and-implementation-zh/> Linux CFS 调度器：原理、设计与内核实现（2023）
- [4] [https://github.com/torvalds/linux/blob/v2.6.23/kernel/sched\\_fair.c](https://github.com/torvalds/linux/blob/v2.6.23/kernel/sched_fair.c)  
linux-v2.6.23 CFS 调度源码