成绩 _____

计 算 机 操 作 系 统

课程设计报告

题 目 处 理 器 的 多 级 反 馈 调 度 算 法 的 模拟设计与实现

专业： 信息安全

班级： 奇安信 1 班

学号+姓名： 202183760012 朱宸扬

指导教师： 赵晓平

2023 年 11 月 30 日

一. 设计目的

掌握处理器调度算法中的多级反馈调度算法

二. 设计内容

本系统采用多级反馈调度算法，模拟操作系统处理器调度的过程。

三. 设计原理

多级反馈调度队列算法是一种操作系统中用于进程调度的算法，其核心思想是为不同优先级的进程定义多个队列，并通过动态调整进程的优先级来实现不同程度的公平性和响应性。这种调度算法通常用于分时系统，其中多个进程共享系统资源，需要公平地分配 CPU 时间

以下是多级反馈调度队列算法的基本原理：

多级队列： 系统维护多个就绪队列，每个队列对应一个不同的优先级。通常，初始时将进程放入最高优先级的队列。

调度策略： 调度器根据某种策略从最高优先级的非空队列中选择一个进程执行。一旦一个进程的时间片用完，它将被移到下一个较低优先级的队列，以便为其他进程让出 CPU 资源。

时间片轮转： 每个队列都可以使用时间片轮转调度算法，确保每个进程在一个时间片内得到执行机会。当进程在当前队列用完时间片后，它将被移到下一个较低优先级的队列。

优先级提升： 如果一个进程等待了足够长的时间而没有得到执行，系统可以提升它的优先级，以确保等待时间过长的进程有更大的机会获得 CPU 执行时间。

优先级降低： 如果一个进程在其队列内执行了一段时间，而仍然需要 CPU 执行时间，那么它的优先级可能会降低，以便给其他进程更多的机会。

四. 详细设计及编码

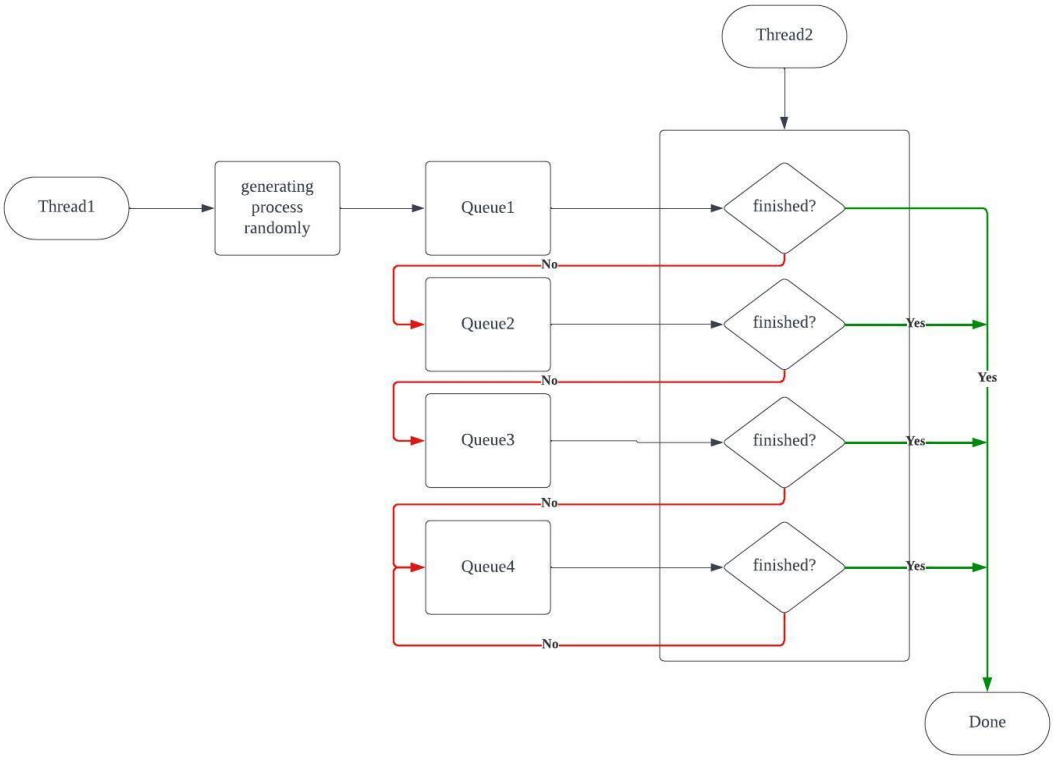
1. 模块分析

进程有序号和工作量大小两个属性

列表选用了队列这种数据结构

采用多线程模拟进程不断进入 1 号队列和处理器处理进程两个任务同步进行

2. 流程图



3. 代码实现

```
import pygame

import random

import time
```

```
from queue import Queue
import threading

# 初始化 Pygame
pygame.init()

# 设置屏幕大小和颜色
WIDTH, HEIGHT = 800, 600
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Multi-Level Feedback Queue Scheduling")

# 定义进程类
class Process:
    def __init__(self, name, job_size):
        self.name = name
        self.job_size = job_size
        self.color = (random.randint(210, 255), random.randint(210,
255), random.randint(210, 255))


# 创建四个队列，时间片分别为 1、2、4、8
queues = [Queue() for _ in range(4)]
time_slices = [1, 2, 4, 8]

# 创建停止标志
```

```
    stop_flag = threading.Event()

    # 创建互斥锁
    lock = threading.Lock()

    # 创建队列和进程的初始坐标
    queue_coordinates = [(50, 200), (250, 200), (450, 200), (650, 200)]
    process_coordinates = (50, 400)

    # 创建进程和队列的字体
    font = pygame.font.Font(None, 36)

    # 创建 Pygame 时钟对象
    clock = pygame.time.Clock()
    def draw_queues_and_processes():
        screen.fill(WHITE)

        # Display the title
        title_text = font.render("Multi-Level Feedback Queue Scheduling",
True, BLACK)
        screen.blit(title_text, (WIDTH // 2 - 250, 50))

        # 绘制队列
        for i, q in enumerate(queues):
            pygame.draw.rect(screen, BLACK, (queue_coordinates[i][0],
queue_coordinates[i][1], 50, 250), 2)  # Slight increase in height to 250
```

```
            # 绘制 Queue 文本
            text_queue = font.render(f"Queue {i + 1}", True, BLACK)
            screen.blit(text_queue, (queue_coordinates[i][0] + 10,
queue_coordinates[i][1] + 270))  # Below Queue

            # 绘制进程在队列中的位置
            for j, process in enumerate(q.queue):
                pygame.draw.rect(screen, process.color,
(queue_coordinates[i][0], queue_coordinates[i][1] + j * 40, 50, 30))  #
Slight decrease in spacing to 30
                # 显示剩余的 job_size
                text = font.render(str(process.job_size), True, BLACK)
                screen.blit(text, (queue_coordinates[i][0] + 10,
queue_coordinates[i][1] + j * 40 + 5))

            # 绘制 Time Slice 文本
            text_time_slice = font.render(f"Time Slice {2**i}", True,
BLACK)
            screen.blit(text_time_slice, (queue_coordinates[i][0] + 10,
queue_coordinates[i][1] - 30))  # Above Queue

        pygame.display.flip()

    def working():
        while not stop_flag.is_set() or any(not queue.empty() for queue
in queues):
            for i in range(len(queues)):
```

```
                queue_size = queues[i].qsize()  # Get the number of
elements in the queue
                for _ in range(queue_size):
                    with lock:
                        process = queues[i].get()
                        temp = process.job_size
                        process.job_size -= time_slices[i]

                    if process.job_size < 0:
                        time.sleep(temp/4)
                        process.job_size = 0
                    else:
                        time.sleep(time_slices[i]/4)

                    with lock:
                        if process.job_size > 0:
                            if i == 3:
                                queues[3].put(process)
                            else:
                                queues[i + 1].put(process)
                        else:
                            print(f"{process.name} 运行结束")

                    draw_queues_and_processes()
                    clock.tick(10)  # 控制帧率

    def create_processes():
```

```
        process_num = 1
        while not stop_flag.is_set() and process_num <= 15:  # 创建 15 个新进程后停止
            with lock:
                new_process = Process(f"Process-{process_num}", random.randint(1, 30))
                queues[0].put(new_process)
                print(f"New process created: {new_process.name} (Job Size: {new_process.job_size})")
            process_num += 1
            time.sleep(random.uniform(1, 3))  # 随机间隔时间

        # 设置停止标志
        stop_flag.set()

    # 创建两个线程
    working_thread = threading.Thread(target=working)
    create_processes_thread = threading.Thread(target=create_processes)

    # 启动线程
    working_thread.start()
    create_processes_thread.start()

    # 运行 Pygame 主循环
    running = True
    while running:
```

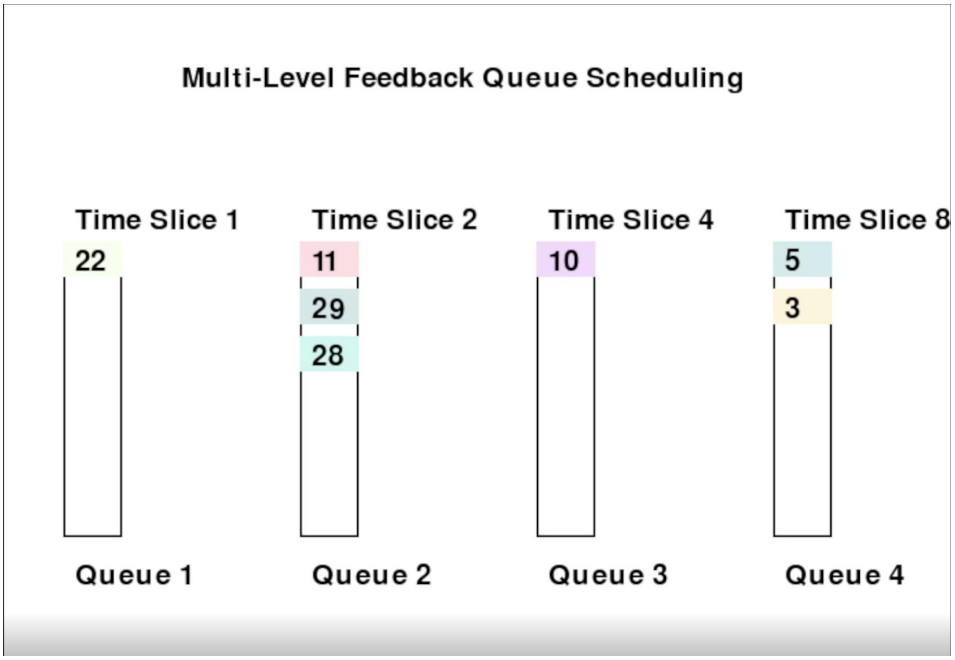
```
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

# 等待两个线程完成
working_thread.join()
create_processes_thread.join()

# 退出 Pygame
pygame.quit()
```

4.结果及其相关分析（结果必须可视化）

可视化截图，完整动画另附



五.课程设计小结

通过本次实验，我们深入了解了多级反馈调度算法的机制和运作方式。这不

仅有助于我们理解操作系统中的进程调度，还为我们进一步研究和优化调度算法提供了基础。

实验的完成使我们更好地理解了理论知识，并通过实际操作加深了对多级反馈调度算法的认识。这将有助于我们更好地应用和理解操作系统的相关概念。