### 一、引言

本设计说明书详细描述了一个基于多级反馈队列的操作系统任务调度策略。该策略旨在高效管理 CPU 和 I/O 任务，并通过优先级、截止时间和 I/O 完成状态等因素来优化任务执行顺序，以提升系统响应性和任务完成率。

### 二、整体架构

本调度策略的核心是 policy 函数，它作为调度器的入口点，根据当前系统事件和任务状态，决定下一时刻 CPU 和 I/O 设备应运行的任务。整个系统围绕以下几个关键组件构建：

事件：表示系统中发生的各种调度相关行为，如任务到达、I/O 请求、I/O 结束、任务完成等。

任务：待调度的基本单位，包含任务 ID、优先级、到达时间、截止时间及当前状态。

多级反馈队列：用于管理 CPU 任务和 I/O 任务的核心数据结构，分为多个优先级等级。

调度算法：policy 函数内部的逻辑，负责处理事件、更新队列，并根据预设规则选择要执行的任务。

动作 (Action)：调度器输出的结果，指示 CPU 和 I/O 设备下一步应运行的任务。

### 三、数据结构设计

**3.1 Event 结构**

作用：封装了系统中发生的各种调度相关事件及其携带的任务信息。

成员：

enum class Type: 定义事件的类型，包括 kTimer (计时器事件)、kTaskArrival (任务到达)、 kTaskFinish (任务完成)、kIoRequest (I/O 请求)、kIoEnd (I/O 结束)。

int time: 事件发生的时间点。

Task task: 与该事件关联的任务详细信息。

**3.2 Event::Task 结构**

作用：表示一个待调度的独立任务单元。

成员：

bool operator==(const const Task& other) const: 重载相等运算符，用于根据 taskId 判断任务是否相同，便于在 std::vector 中查找和删除。

enum class Priority: 定义任务的优先级，kHigh (高) 和 kLow (低)。

enum class Status: 定义任务的当前状态，kReady (就绪)、kRunning (运行中)、kIoWait (I/O 等待)、kIoEnd (I/O 结束)、kFinished (已完成)。

Int arrivalTime: 任务进入系统的模拟时间。

int deadline: 任务必须完成的截止时间。

Priority priority: 任务的初始优先级。

int taskId: 任务的唯一标识符。

Status status: 任务的当前状态，在调度过程中动态更新。

**3.3 Action 结构**

作用：调度器执行一次调度后返回的结果，指示 CPU 和 I/O 设备应执行的任务。

成员：

int cpuTask: CPU 上应运行的任务 ID。0 表示 CPU 空闲。

int ioTask: I/O 设备上应运行的任务 ID。0 表示 I/O 设备空闲。

**3.4 全局任务队列**

std::vector<Event::Task> g\_cpuTaskQueues[4]: 实现了四级多级反馈队列，用于存放处于 就绪态 或 运行态 的 CPU 任务。索引 0 代表最高优先级，3 代表最低优先级。

std::vector<Event::Task> g\_ioTaskQueues[4]: 同样是四级多级反馈队列，用于存放处于 I/O 等待态 的任务。索引 0 代表最高优先级，3 代表最低优先级。

int g\_currentTime: 全局变量，记录当前的模拟时间。

**3.5 常量定义**

SchedulerConstants 命名空间: 封装了所有调度器相关的常量，提高代码可读性和可维护性。

NUM\_PRIORITY\_QUEUES = 4: 队列的等级数量。

IO\_END\_PRIORITY\_BOOST\_FACTOR = 0.4: 任务完成 I/O 后，其截止时间计算分数的加成因子，数值越小代表优先级越高。

REGULAR\_TASK\_PRIORITY\_FACTOR = 1.0: 普通任务的截止时间计算分数因子。

OVERDUE\_PENALTY\_SCORE = 1e5: 任务超时后，其分数将加上一个巨大的惩罚值，使其优先级极低。

IDLE\_TASK\_ID = 0: 表示 CPU 或 I/O 设备空闲时，没有任务运行。

### 四、调度算法逻辑

本调度策略的核心逻辑在 policy 函数中实现，并借助 selectAndMigrateTask 和 removeTaskFromQueues 两个辅助函数。

**4.1 removeTaskFromQueues 辅助函数**

功能：这是一个通用辅助函数，用于从给定的多级队列数组中，删除指定 taskId 的任务。

实现：它遍历所有队列级别，并使用 std::remove\_if 和 vector::erase idiom 来安全有效地移除匹配的任务。

**4.2 selectAndMigrateTask 辅助函数**

功能：实现任务的选择逻辑和队列间的迁移。它从 fromLevel 队列中选择一个最优任务，并将其移动到 toLevel 队列。

选择策略 (评分机制)：

目标：选择分数最低（优先级最高）的任务。

**未超时任务：**

I/O 结束优先：

如果任务的状态是 kIoEnd (刚从 I/O 完成)，其分数计算为 IO\_END\_PRIORITY\_BOOST\_FACTOR \* (task.deadline - currentTime)。IO\_END\_PRIORITY\_BOOST\_FACTOR (0.4) 小于 REGULAR\_TASK\_PRIORITY\_FACTOR (1.0)，意味着刚完成 I/O 的任务会获得更高的优先级，有助于 I/O 密集型任务快速获得 CPU 资源。

普通任务：

否则，分数为 REGULAR\_TASK\_PRIORITY\_FACTOR \* (task.deadline - currentTime)。

截止时间越早越优先：

在未超时的任务中，deadline - currentTime 的值越小，即任务离截止时间越近，分数越低，优先级越高。

**超时任务**：

分数计算为 OVERDUE\_PENALTY\_SCORE + (task.deadline - currentTime)。OVERDED\_PENALTY\_SCORE (1e5) 确保超时任务的分数极高，使其优先级非常低，通常不会被选中，除非所有其他任务都已完成或超时。

迁移逻辑：一旦任务被选中，它会从 fromLevel 队列中移除，并添加到 toLevel 队列的末尾。这是反馈队列机制的一部分。

**4.3 policy 主调度函数**

功能：这是整个调度器的核心，负责响应系统事件、维护任务队列状态并做出调度决策。

步骤：

**1.初始化：**

创建一个 Action 对象 resultAction，并将其 cpuTask 和 ioTask 初始设置为当前正在运行的任务 ID。这意味着如果调度器没有找到新的任务，将继续执行当前任务。

**2.事件处理：**

遍历当前调度周期内发生的所有 events。

根据 event.type 更新 g\_currentTime 和任务队列的状态：

kTaskArrival: 新任务根据其 priority (高或低) 分别加入 g\_cpuTaskQueues[0] 或 g\_cpuTaskQueues[2]。

kIoRequest: 任务从任何 CPU 队列中移除，然后根据其 priority 加入 g\_ioTaskQueues[0] 或 g\_ioTaskQueues[2]。

kIoEnd: 任务从任何 I/O 队列中移除，将其 status 设置为 kIoEnd，然后重新根据其 priority 加入 g\_cpuTaskQueues[0] 或 g\_cpuTaskQueues[2]。kIoEnd 状态是任务获得 CPU 优先级提升的关键。

kTaskFinish: 任务从任何 CPU 队列中移除。

kTimer: 计时器事件在此处不直接改变队列状态，但它会触发 policy 函数的调用，进而触发后续的任务选择。

**3.CPU 任务选择：**

从最高优先级的 g\_cpuTaskQueues[0] 开始，依次遍历到最低优先级 g\_cpuTaskQueues[3]。

对于每个队列级别 i，调用 selectAndMigrateTask 尝试选择一个任务。

反馈机制：

如果任务是从队列 i 中选出的，它将被移动到 i+1 队列（降级）。如果任务是从最低优先级队列 g\_cpuTaskQueues[3] 中选出的，它将被移动回 g\_cpuTaskQueues[0]（升级，防止饥饿）。

一旦找到并选择了 CPU 任务，将其 taskId 赋值给 resultAction.cpuTask 并停止搜索。

**4.I/O 任务选择：**

条件：只有当 currentIoTask == IDLE\_TASK\_ID (I/O 设备当前空闲) 时才进行 I/O 任务的选择。

逻辑与 CPU 任务选择类似，从 g\_ioTaskQueues[0] 开始遍历，使用 selectAndMigrateTask 选择最优任务，并将其 taskId 赋值给 resultAction.ioTask。同样，如果从队列 i 中选出，会移动到 i+1 或回到 0。

**5.返回结果：**返回 resultAction，指示下一调度周期的 CPU 和 I/O 任务。

### 五、调度策略特点

多级反馈队列：提供了对不同类型任务的灵活管理，允许任务在不同优先级队列间移动。

优先级驱动：任务根据其初始优先级和动态状态（如 I/O 结束）在队列中获得不同的优先级。

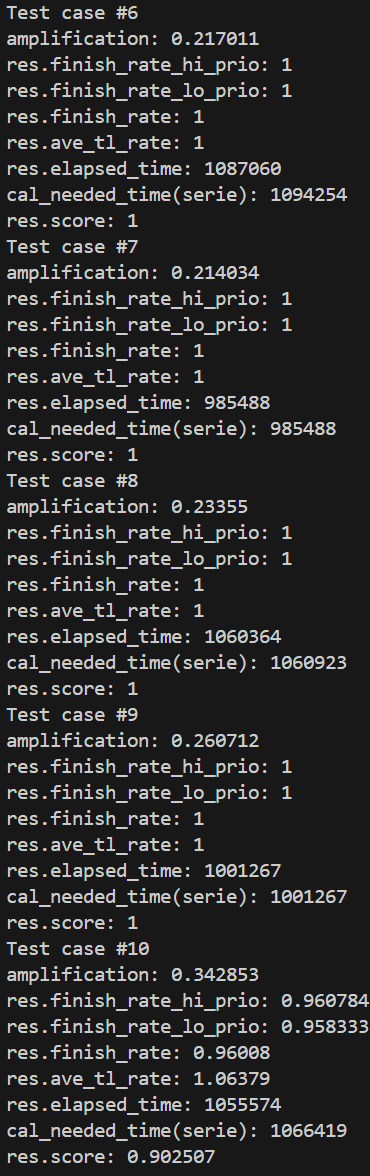
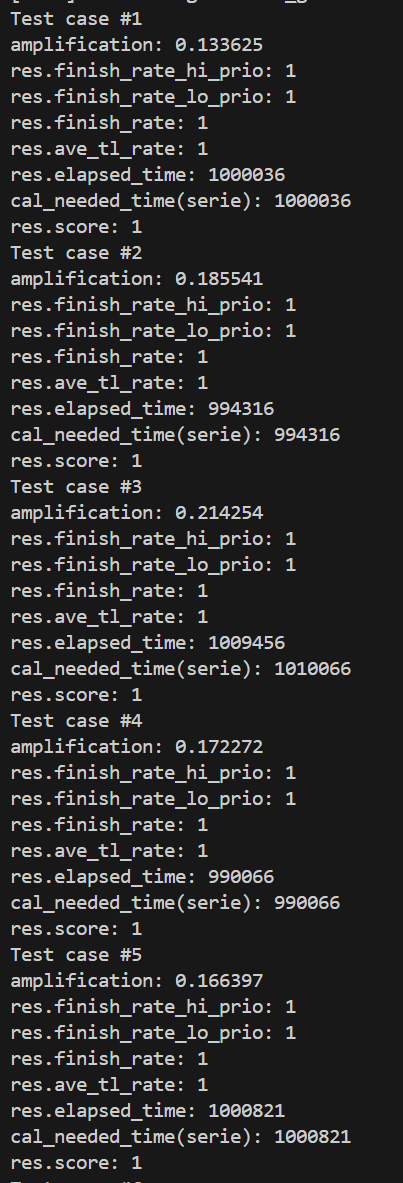
截止时间敏感：在任务选择时，会优先考虑距离截止时间更近的任务，有助于满足任务的时限要求。

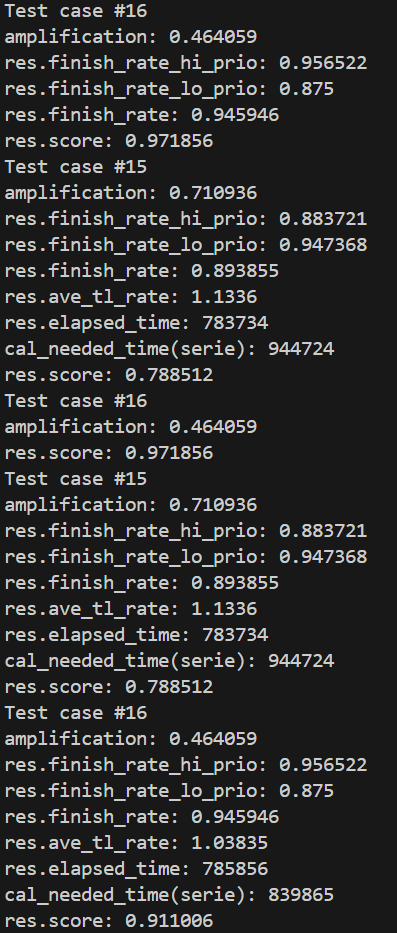
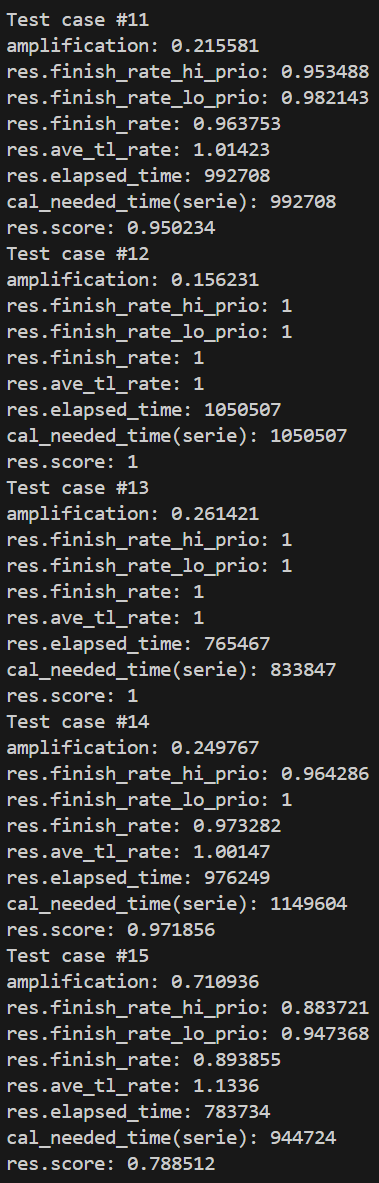
I/O 密集型任务优化：刚完成 I/O 的任务会获得优先级提升，减少 I/O 密集型任务的等待时间，提高系统吞吐量和响应性。

防饥饿机制：最低优先级队列中的任务在被选中后会被提升回最高优先级队列，防止任务长时间停留在低优先级队列而无法执行。

动态调整：任务的状态和队列归属会根据事件（如 I/O 请求、I/O 结束）动态调整，使调度策略能够适应系统负载的变化。

### 实验结果





### 七、限制与未来改进

时间片/量子未明确：当前策略仅选择任务，未明确任务运行的时间片长度。在实际系统中，每个队列级别通常会有一个定义好的时间片，任务用完时间片后会被抢占并降级。

CPU/I/O 爆发时间未知：策略没有考虑任务的 CPU 爆发时间或 I/O 爆发时间。在更复杂的调度器中，这些信息可用于更精准的预测和调度。

上下文切换开销：策略未考虑任务上下文切换的开销。

全局状态：使用了全局变量（g\_currentTime, g\_cpuTaskQueues, g\_ioTaskQueues），这意味着在多线程环境或需要并发模拟的情况下，可能需要进行额外的同步或将状态封装到类中。