信息与软件工程学院上机实验报告

(第 1 次)

一、实验名称

进程调度实验

二、实验目的及要求

实验目的

1. 加深对进程概念及进程调度算法的理解。
2. 在了解和掌握进程调度算法的基础上，编制进程调度算法通用程序，将调试结果显示在计算机屏幕上，并验证结果一致性。

实验要求

1. 了解进程调度；
2. 理解利用进程调度算法进行调度的原理；
3. 至少实现两种算法求平均周转时间和平均带权周转时间。

三、实验环境

- 开发语言：C++
- 实验环境：Linux (Ubuntu 24.04 LTS)
- 编译工具：g++

四、实验设计

1. 实验步骤

设计思想

本次实验设计了一个通用的进程调度模拟器，能够从标准输入读取进程信息（进程名、到达时间、运行时间、优先数），并根据命令行参数选择不同的调度算法（FCFS、HPF、MLFQ）进行模拟。

模拟器通过一个主循环推进时间 `t`，在每个时间单位，它会：

1. 检查是否有新进程到达；
2. 根据所选算法的规则，决定当前时间单位应运行哪个进程；
3. 更新运行进程的剩余时间、状态等动态信息；
4. 记录每个时间单位的运行情况；
5. 当一个进程完成时，记录完成时间并计算各项性能指标。

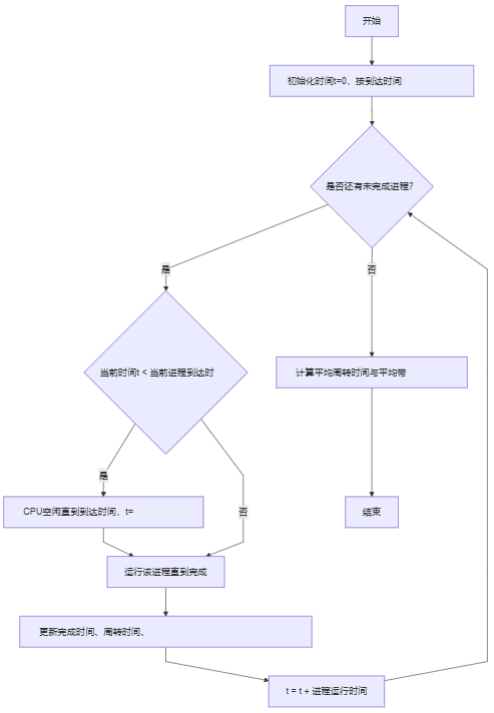
核心数据结构

为了模拟进程调度，设计了以下几个核心结构体：

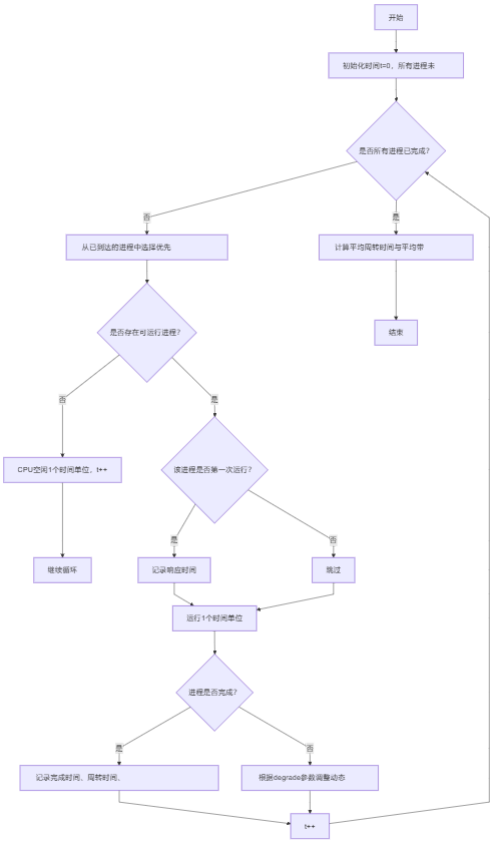
- **Process**：存储进程静态信息
  - `name`：进程名
  - `arrival`：到达时间
  - `burst`：运行时间
  - `priority`：静态优先数
- **ProcRuntime**：存储运行时动态信息
  - `remaining`：剩余时间
  - `start_time`：第一次获得 CPU 的时间
  - `completion_time`：完成时间
  - `dyn_priority`：动态优先数（HPF 使用）
  - `level`：队列层级（MLFQ 使用）
  - `used_in_quantum`：当前已用时间片（MLFQ 使用）
- **Metrics**：每个进程的统计指标
  - `completion`：完成时间
  - `turnaround`：周转时间（完成时间 - 到达时间）
  - `weighted_turn`：带权周转时间（周转时间 / 运行时间）
  - `waiting`：等待时间（周转时间 - 运行时间）
  - `response`：响应时间（第一次运行 - 到达时间）
- **SimulationResult**：模拟完整结果
  - `timeline`：时间轴（记录每个时间单位 CPU 使用情况）
  - `per_process`：每个进程的指标
  - `avg_turnaround`：平均周转时间
  - `avg_weighted_turn`：平均带权周转时间
  - `makespan`：总完成时间

算法流程图

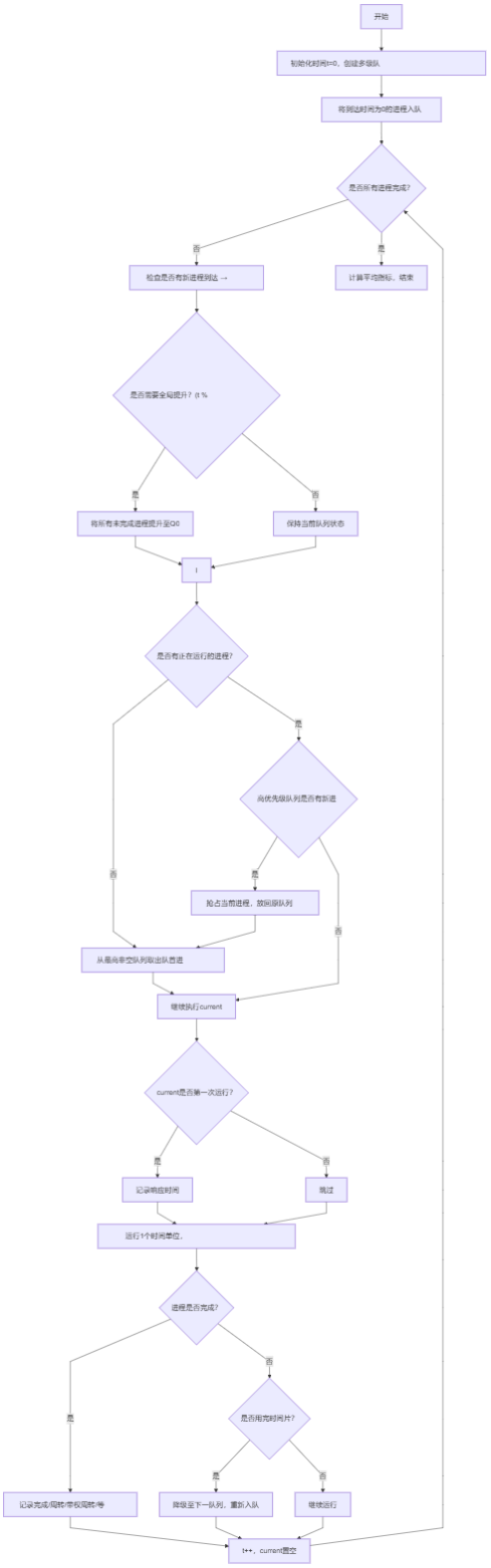
FCFS


开始
初始化时间t=0，按到达时间
是否还有未完成进程?
是
否
当前时间t < 当前进程到达时
计算平均周转时间与平均带
是
CPU空闲直到到达时间，t=
否
结束
运行该进程直到完成
更新完成时间、周转时间、
t = t + 进程运行时间

HPF


开始
初始化时间t=0，所有进程未
是否所有进程已完成?
否
是
从已到达的进程中选择优先
计算平均周转时间与平均带
结束
是否存在可运行进程?
否
是
CPU空闲1个时间单位，t++
该进程是否第一次运行?
继续循环
是
否
记录响应时间
跳过
运行1个时间单位
进程是否完成?
是
否
记录完成时间、周转时间、
根据degrade参数调整动态
t++

MLFQ（多级反馈队列）流程图



2. 调试过程及实验结果

测试数据

```
P1 0 5 3
P2 1 3 4
P3 2 4 2
P4 3 2 1
P5 4 3 5
```

执行结果

FCFS

```
→ ./sched --algo=fcfs < sample.txt
已加载进程 (name, arrival, burst, priority):
  P1          到达=  0  运行=  5  优先=  3
  P2          到达=  1  运行=  3  优先=  4
  P3          到达=  2  运行=  4  优先=  2
  P4          到达=  3  运行=  2  优先=  1
  P5          到达=  4  运行=  3  优先=  5

算法: 先来先服务 (FCFS)

══ 调度结果 ══
时间轴(压缩显示):
  [P1:5][P2:3][P3:4][P4:2][P5:3]

进程统计 (按输入顺序):
进程      到达  运行    完成    周转带权周转    等待    响应
P1              0       5         5         5      1.00         0         0
P2              1       3         8         7      2.33         4         4
P3              2       4        12        10      2.50         6         6
P4              3       2        14        11      5.50         9         9
P5              4       3        17        13      4.33        10        10

平均周转时间: 9.20
平均带权周转时间: 3.13
总完成时间 (makespan): t=17
```

HPF（抢占式，priority=high，degrade=1）

```
→ ./sched --algo=hpf --priority=high --degrade=1 < sample.txt
已加载进程 (name, arrival, burst, priority):
  P1          到达=  0  运行=  5  优先=  3
  P2          到达=  1  运行=  3  优先=  4
  P3          到达=  2  运行=  4  优先=  2
  P4          到达=  3  运行=  2  优先=  1
  P5          到达=  4  运行=  3  优先=  5

算法：最高优先数 (HPF) [抢占式，时间片=1]
优先级规则：数值越大优先级越高
未完成时每用 1 个时间片动态优先数调整：-1

═══ 调度结果 ═══
时间轴(压缩显示):
  [P1:1][P2:2][P1:1][P5:3][P2:1][P3:1][P1:1][P3:1][P4:1][P1:1][P3:1][P4:1][P1:1][P3:1]

进程统计 (按输入顺序):
进程      到达  运行    完成    周转带权周转    等待    响应
P1              0       5       16       16      3.20       11        0
P2              1       3        8        7      2.33        4        0
P3              2       4       17       15      3.75       11        6
P4              3       2       15       12      6.00       10        8
P5              4       3        7        3      1.00        0        0

平均周转时间：10.60
平均带权周转时间：3.26
总完成时间 (makespan)：t=17
```

HPF(非抢占式)

```
→ ./sched --algo=hpf --nonpreemptive < sample.txt
已加载进程 (name, arrival, burst, priority):
  P1          到达=  0  运行=  5  优先=  3
  P2          到达=  1  运行=  3  优先=  4
  P3          到达=  2  运行=  4  优先=  2
  P4          到达=  3  运行=  2  优先=  1
  P5          到达=  4  运行=  3  优先=  5

算法：最高优先数 (HPF) [非抢占]
优先级规则：数值越大优先级越高

═══ 调度结果 ═══
时间轴(压缩显示):
  [P1:5][P5:3][P2:3][P3:4][P4:2]

进程统计 (按输入顺序):
进程      到达  运行    完成    周转带权周转    等待    响应
P1              0       5        5        5      1.00        0        0
P2              1       3       11       10      3.33        7        7
P3              2       4       15       13      3.25        9        9
P4              3       2       17       14      7.00       12       12
P5              4       3        8        4      1.33        1        1

平均周转时间：9.20
平均带权周转时间：3.18
总完成时间 (makespan)：t=17
```

MLFQ

```
→ ./sched --algo=mlfq --quantums=2,4,8 --boost=0 < sample.txt
已加载进程 (name, arrival, burst, priority):
  P1          到达=  0  运行=  5  优先=  3
  P2          到达=  1  运行=  3  优先=  4
  P3          到达=  2  运行=  4  优先=  2
  P4          到达=  3  运行=  2  优先=  1
  P5          到达=  4  运行=  3  优先=  5

算法: 多级反馈队列 (MLFQ), 级数=3, 各级时间片=2,4,8

═══ 调度结果 ═══
时间轴(压缩显示):
  [P1:2][P2:2][P3:2][P4:2][P5:2][P1:3][P2:1][P3:2][P5:1]

进程统计 (按输入顺序):
进程      到达  运行    完成    周转带权周转    等待    响应
P1              0       5        13        13      2.60         8         0
P2              1       3        14        13      4.33        10         1
P3              2       4        16        14      3.50        10         2
P4              3       2         8         5      2.50         3         3
P5              4       3        17        13      4.33        10         4

平均周转时间: 11.60
平均带权周转时间: 3.45
总完成时间 (makespan): t=17
```

---

3. 结果分析与总结

结果分析

1. **FCFS**：实现最简单，规则公平。但效率不高，平均周转时间为 9.20。P1 (长作业) 先到，导致 P2, P3, P4, P5 均需等待。特别是短作业 P4 (运行时间2)，其等待时间长达9。
2. **HPF (非抢占式)**：
   - **分析**：此模式下，一旦进程获得 CPU，它会一直运行到完成。P1 (优3) 在 t=0 到达并开始运行，t=5 才结束。在 P1 运行期间，P2(优4), P3(优2), P4(优1), P5(优5) 相继到达。
   - **结果**：t=5 时 P1 结束，此时就绪队列中有 P2, P3, P4, P5，其中 P5 优先级最高(5)，P5 运行。t=8 时 P5 结束。t=8 时就绪队列中有 P2(优4), P3(优2), P4(优1)，P2 运行...
   - **对比**：这种方式避免了抢占的开销。P5(优5) 的响应时间为1 (t=5开始 - t=4到达)，周转时间为4。虽然 P5 优先级最高，但它必须等待 P1 运行完毕。此算法的平均周转时间(9.20)和 FCFS 相同，但带权周转时间(3.18)略好于 FCFS(3.13)。
3. **HPF (抢占式)**：
   - **分析**：该算法能快速响应高优先级进程。P5 (优先级5) 在 t=4 到达后，立即抢占了 CPU（P1在t=0运行, t=1被P2抢占, t=3 P2结束, P1抢回CPU, t=4被P5抢占），并在 t=7 时迅速完成，周转时间仅为3，响应时间为0。
   - **对比**：P2 (优先级4) 也在 t=1 到达时立即抢占 P1，响应时间为0。抢占式 HPF 提供了最佳的响应时间（对高优先级而言），但代价是低优先级进程 (P4, P3, P1) 的等待时间很长，P1 (优先数3) 的周转时间甚至达到了16，导致平均周转时间变差 (10.60)。此算法容易导致低优先级进程 "饥饿"。
4. **MLFQ**：MLFQ 试图平衡周转时间和响应时间。
   - **响应快**：所有进程的响应时间都较短 (0, 1, 2, 3, 4)，因为它允许新进程在最高队列中先运行一个短时间

片。
    - **偏向短作业**：短作业 P4 (运行时间2) 在 Q0 中运行了2个时间单位即完成，总周转时间为5，远好于 FCFS(11)、HPF 抢占式(12) 和 HPF 非抢占式(14)。
    - **公平性**：长作业 (如 P1, P3, P5) 会逐渐下降到低优先级队列，在后台慢慢运行，确保了所有进程都能最终完成。
    - **性能**：在本测试数据中，MLFQ 的平均周转时间 (11.60) 不是最优的，这说明 MLFQ 的参数（时间片长度、队列数量）对性能影响很大，需要根据作业负载进行调整。但它提供了最好的响应时间均衡性，并有效执行了短作业。

思考题

问题： 哪种算法容易实现?
回答： 先来先服务（FCFS）算法最容易实现。只需按到达时间顺序执行，无需抢占与优先级比较。

心得体会

通过本次实验，我成功设计并实现了一个支持 FCFS, HPF 和 MLFQ 算法的进程调度模拟器。我深刻理解了不同调度算法的设计哲学及其性能权衡：

- FCFS 追求公平和简单，但牺牲了效率。
- HPF 追求高优先级任务的快速响应，但可能导致饥饿。
- MLFQ 是一种更复杂的折中方案，它试图同时优化响应时间（对新进程友好）和周转时间（对短作业友好），同时通过多级队列确保长作业也能得到执行，避免饥饿。

在实现 MLFQ 的过程中，我遇到了如何处理时间片、抢占和队列降级等逻辑细节的挑战。通过调试输出结果中的时间轴，我能够逐步验证我的算法逻辑是否正确。例如，`[P1:2][P2:2]` 显示 P1 在 Q0 用完2个时间片后，P2 (t=1到达) 得以在 Q0 运行。这证明了 MLFQ 的基本轮转和降级机制在工作。

本次实验锻炼了我的 C++ 编程能力和算法设计能力，也让我对操作系统的核心概念——进程调度有了更直观和深入的认识。

改进意见

增加不同负载下的参数敏感性简要对比，并补充异常输入处理。

---

4. 附录

```
#include <algorithm>  // 排序需要
#include <cctype>     // isspace
#include <climits>    // INT_MAX
#include <iomanip>    // setw, setprecision
#include <iostream>
#include <queue>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

// --------------------------- 基本数据结构 ---------------------------
```

```cpp
// 进程的静态信息（来自输入）
struct Process {
    string name;   // 进程名
    int arrival;   // 到达时间
    int burst;     // 需要运行的 CPU 时间（总量）
    int priority;  // 优先数（数值大/小与“优先级高”可通过参数设定）
};

// 进程在模拟过程中的动态信息（运行时）
struct ProcRuntime {
    int remaining;       // 剩余运行时间
    int start_time;      // 第一次获得 CPU 的时间（用于计算响应时间）
    int completion_time; // 完成时间
    int dyn_priority;    // 动态优先数（HPF 会用到）
    int level;           // 所在队列层级（MLFQ 会用到）
    int used_in_quantum; // 当前已用时间片（MLFQ 会用到）
};

// 统计指标（按每个进程记录）
struct Metrics {
    int completion;          // 完成时间 C
    int turnaround;          // 周转时间 T = 完成时间 - 到达时间
    double weighted_turn;    // 带权周转时间 W = T / 需要运行时间
    int waiting;             // 等待时间 = 周转时间 - 需要运行时间
    int response;            // 响应时间 = 第一次运行 - 到达时间
};

// 模拟的整体结果
struct SimulationResult {
    vector<string> timeline;      // 时间轴：每个时间单位上 CPU 运行了哪个进程（或“-”表示空闲）
    vector<Metrics> per_process;  // 每个进程的统计指标（顺序与输入一致）
    double avg_turnaround;        // 平均周转时间
    double avg_weighted_turn;     // 平均带权周转时间
    int makespan;                 // 总完成时间（最后一个完成的时刻）
};

// --------------------------- 工具函数 ---------------------------

// 去掉字符串首尾空白字符
static string trim(const string &s) {
    size_t b = 0, e = s.size();
    while (b < e && isspace((unsigned char)s[b])) b++;
    while (e > b && isspace((unsigned char)s[e - 1])) e--;
    return s.substr(b, e - b);
}

// 将形如 "2,4,8" 的字符串分割为整数向量
static vector<int> split_to_ints(const string &s, char delim) {
    vector<int> out;
    string item;
    stringstream ss(s);
    while (getline(ss, item, delim)) {
        item = trim(item);
```

```
        if (!item.empty()) {
            int v = 0;
            stringstream is(item);
            if (is >> v) {
                if (v < 1) v = 1; // 时间片至少为 1
                out.push_back(v);
            }
        }
    }
    return out;
}

// 打印命令行使用帮助（中文）
static void print_usage() {
    cout << "CPU 调度模拟器（FCFS | HPF | MLFQ）\n";
    cout << "编译：g++ -std=c++17 -O2 -pipe -o sched src/main.cpp\n\n";
    cout << "用法：./sched --algo=fcfs|hpf|mlfq [参数] < 输入文件\n\n";
    cout << "输入格式（每行一个进程，# 开头为注释）：\n";
    cout << "  name arrival burst priority\n\n";
    cout << "常用参数：\n";
    cout << "  --algo=fcfs|hpf|mlfq      选择算法\n";
    cout << "  --verbose                 打印逐时刻运行日志\n";
    cout << "  --priority=high|low       HPF：数值越大(或越小)优先级越高（默认 high）\n";
    cout << "  --degrade=N               HPF 抢占式：未完成且用完 1 个时间片后优先数调整 N\n";
    cout << "  --nonpreemptive           HPF：使用非抢占式（一次运行到完成）\n";
    cout << "  --quantums=a,b,c          MLFQ：各级队列的时间片（默认 2,4,8）\n";
    cout << "  --boost=K                 MLFQ：每 K 个时间单位进行全局提升(0 关闭)\n\n";
}

// 命令行参数
struct Args {
    string algo;                // fcfs | hpf | mlfq
    bool verbose;               // 是否打印逐时刻日志
    bool hpf_preemptive;        // HPF 是否抢占式（抢占式时间片固定 1）
    int hpf_degrade;            // HPF 抢占式：未完成时每用一个时间片优先数的调整量
    bool priority_high_is_high; // HPF：true 表示数值越大优先级越高；false 反之
    vector<int> mlfq_quantums;  // MLFQ：各级时间片
    int mlfq_boost_interval;    // MLFQ：全局提升间隔（0 表示不提升）
};

// 解析命令行参数（简单直白的写法）
static bool parse_args(int argc, char **argv, Args &args) {
    // 设置默认值
    args.algo = "fcfs";
    args.verbose = false;
    args.hpf_preemptive = true;
    args.hpf_degrade = 1;
    args.priority_high_is_high = true;
    args.mlfq_quantums = {2, 4, 8};
    args.mlfq_boost_interval = 0;

    for (int i = 1; i < argc; i++) {
        string a = argv[i];
```

```
        if (a == "-h" || a == "--help") { print_usage(); return false; }
        if (a.find("--algo=") == 0) {
            args.algo = a.substr(7);
        } else if (a == "--verbose") {
            args.verbose = true;
        } else if (a == "--nonpreemptive") {
            args.hpf_preemptive = false;
        } else if (a.find("--degrade=") == 0) {
            int v = 1;
            stringstream ss(a.substr(10));
            ss >> v; if (v < 0) v = 0;
            args.hpf_degrade = v;
        } else if (a.find("--priority=") == 0) {
            string v = a.substr(11);
            for (size_t k = 0; k < v.size(); ++k) v[k] = tolower((unsigned char)v[k]);
            if (v == "high") args.priority_high_is_high = true;
            else if (v == "low") args.priority_high_is_high = false;
        } else if (a.find("--quantums=") == 0) {
            vector<int> qs = split_to_ints(a.substr(11), ',');
            if (!qs.empty()) args.mlfq_quantums = qs;
        } else if (a.find("--boost=") == 0) {
            int v = 0; stringstream ss(a.substr(8)); ss >> v; if (v < 0) v = 0;
            args.mlfq_boost_interval = v;
        } else {
            cout << "未知参数: " << a << "\n";
            print_usage();
            return false;
        }
    }
    // 统一小写算法名
    for (size_t k = 0; k < args.algo.size(); ++k) args.algo[k] = tolower((unsigned
char)args.algo[k]);
    return true;
}

// 从标准输入读取进程列表
static bool read_processes(vector<Process> &procs) {
    string line;
    int lineno = 0;
    while (getline(cin, line)) {
        lineno++;
        line = trim(line);
        if (line.empty()) continue;
        if (line[0] == '#') continue; // 注释
        stringstream ss(line);
        Process p; p.name = ""; p.arrival = 0; p.burst = 0; p.priority = 0;
        if (!(ss >> p.name >> p.arrival >> p.burst >> p.priority)) {
            cout << "第 " << lineno << " 行解析失败: " << line << "\n";
            return false;
        }
        if (p.arrival < 0 || p.burst <= 0) {
            cout << "第 " << lineno << " 行数据非法（到达时间需>=0，运行时间需>0）\n";
            return false;
```

```
        }
        procs.push_back(p);
    }
    if (procs.empty()) {
        cout << "未读取到任何进程，请从标准输入提供数据。\n";
        return false;
    }
    return true;
}

// 打印输入概要，便于核对
static void print_input_summary(const vector<Process> &procs) {
    cout << "已加载进程 (name, arrival, burst, priority):\n";
    for (size_t i = 0; i < procs.size(); ++i) {
        const Process &p = procs[i];
        cout << "  " << setw(10) << left << p.name
             << "  到达=" << setw(3) << right << p.arrival
             << "  运行=" << setw(3) << right << p.burst
             << "  优先=" << setw(3) << right << p.priority << "\n";
    }
}

// ---------------------------- FCFS 调度 ----------------------------

// FCFS 比较函数：先到先服务（到达时间小者优先，保持输入顺序稳定）
static bool comp_fcfs(int a, int b, const vector<Process> *pp) {
    const vector<Process> &procs = *pp;
    if (procs[a].arrival != procs[b].arrival) return procs[a].arrival < procs[b].arrival;
    return a < b; // 到达时间相同则按输入顺序
}

static SimulationResult simulate_fcfs(const vector<Process> &procs) {
    int n = (int)procs.size();
    vector<int> order(n);
    for (int i = 0; i < n; i++) order[i] = i;
    // 使用稳定排序，便于新手理解“到达时间优先、同到达按输入顺序”
    stable_sort(order.begin(), order.end(), [&](int a, int b){ return comp_fcfs(a, b,
&procs); });

    vector<Metrics> metrics(n);
    vector<string> timeline; // 记录每个时间单位谁在运行
    int t = 0;               // 当前时间

    for (int k = 0; k < (int)order.size(); ++k) {
        int idx = order[k];
        const Process &p = procs[idx];
        // 若 CPU 空闲且当前时间小于该进程到达时间，则先空闲到其到达
        if (t < p.arrival) {
            for (int x = t; x < p.arrival; ++x) timeline.push_back("-");
            t = p.arrival;
        }
        // 先来先服务：一口气运行到完成
        for (int x = 0; x < p.burst; ++x) timeline.push_back(p.name);
```

```
        t += p.burst;

        // 统计指标
        metrics[idx].completion = t;
        metrics[idx].turnaround = t - p.arrival;
        metrics[idx].weighted_turn = (double)metrics[idx].turnaround / (double)p.burst;
        metrics[idx].waiting = metrics[idx].turnaround - p.burst;
        // 第一次运行的时刻 = 当前时间结束前的开始点 = t - p.burst
        metrics[idx].response = (t - p.burst) - p.arrival;
    }

    // 计算平均值
    double avgT = 0.0, avgW = 0.0;
    for (int i = 0; i < n; ++i) { avgT += metrics[i].turnaround; avgW +=
metrics[i].weighted_turn; }

    SimulationResult res;
    res.timeline = timeline;
    res.per_process = metrics;
    res.avg_turnaround = avgT / n;
    res.avg_weighted_turn = avgW / n;
    res.makespan = t;
    return res;
}

// ---------------------------- HPF 调度（最高优先数） ----------------------------

// 在当前时间 t，从“已到达且未完成”的进程中选择一个优先级最高者
// 参数 high_is_high 表示“数值越大优先级越高（true）/反之（false）”。
static int pick_hpf_process(const vector<Process> &procs,
                            const vector<ProcRuntime> &rt,
                            const vector<bool> &finished,
                            int t,
                            bool high_is_high) {
    int best = -1;
    for (int i = 0; i < (int)procs.size(); ++i) {
        if (finished[i]) continue;
        if (procs[i].arrival > t) continue;
        if (best == -1) {
            best = i; continue;
        }
        int pa = rt[i].dyn_priority;
        int pb = rt[best].dyn_priority;
        // 根据设置，比较“优先数大小与优先级高低”的对应关系
        bool better;
        if (high_is_high) better = (pa > pb);  // 值越大越好
        else better = (pa < pb);               // 值越小越好
        if (better) best = i;
        else if (pa == pb) {
            // 若优先数相同：到达早者优先；若仍相同：下标小者优先
            if (procs[i].arrival < procs[best].arrival) best = i;
            else if (procs[i].arrival == procs[best].arrival && i < best) best = i;
        }
```

```
    }
    return best;
}

static SimulationResult simulate_hpf(const vector<Process> &procs,
                                      bool preemptive,
                                      bool high_is_high,
                                      int degrade,
                                      bool verbose) {
    int n = (int)procs.size();
    vector<ProcRuntime> rt(n);
    vector<bool> finished(n, false);
    vector<Metrics> metrics(n);
    vector<string> timeline;

    for (int i = 0; i < n; ++i) {
        rt[i].remaining = procs[i].burst;
        rt[i].start_time = -1;
        rt[i].completion_time = -1;
        rt[i].dyn_priority = procs[i].priority; // 初始动态优先数 = 静态优先数
        rt[i].level = 0; rt[i].used_in_quantum = 0;
    }

    int t = 0;       // 当前时间
    int done = 0;    // 已完成进程数

    if (!preemptive) {
        // 非抢占式：每次选一个“已到达且优先级最高”的进程，直接运行到完成
        while (done < n) {
            int pick = pick_hpf_process(procs, rt, finished, t, high_is_high);
            if (pick == -1) {
                // 没有就绪进程，CPU 空闲直到下一个进程到达
                int next_arr = INT_MAX;
                for (int i = 0; i < n; ++i) if (!finished[i]) next_arr = min(next_arr,
procs[i].arrival);
                for (int k = t; k < next_arr; ++k) timeline.push_back("-");
                t = next_arr;
                continue;
            }
            // 第一次运行时记录响应时间
            if (rt[pick].start_time < 0) {
                rt[pick].start_time = t;
                metrics[pick].response = t - procs[pick].arrival;
            }
            // 一次性运行到完成
            for (int k = 0; k < rt[pick].remaining; ++k)
timeline.push_back(procs[pick].name);
            t += rt[pick].remaining;
            rt[pick].remaining = 0;
            finished[pick] = true; done++;
            metrics[pick].completion = t;
            metrics[pick].turnaround = t - procs[pick].arrival;
```

```
            metrics[pick].weighted_turn = (double)metrics[pick].turnaround /
(double)procs[pick].burst;
            metrics[pick].waiting = metrics[pick].turnaround - procs[pick].burst;
        }
    } else {
        // 抢占式：时间片固定为 1，每个时间单位重新选择最高优先级的进程
        while (done < n) {
            int pick = pick_hpf_process(procs, rt, finished, t, high_is_high);
            if (pick == -1) {
                // 没有就绪进程，空闲 1 个时间单位（也可一次跳到下个到达，但为了直观每次+1）
                timeline.push_back("-");
                t++;
                continue;
            }
            if (rt[pick].start_time < 0) {
                rt[pick].start_time = t;
                metrics[pick].response = t - procs[pick].arrival;
            }
            // 运行 1 个时间单位
            timeline.push_back(procs[pick].name);
            t++;
            rt[pick].remaining--;
            if (verbose) {
                cout << "t=" << setw(3) << (t - 1)
                     << " 运行=" << procs[pick].name
                     << " 动态优先数=" << rt[pick].dyn_priority
                     << " 剩余=" << rt[pick].remaining << "\n";
            }
            if (rt[pick].remaining == 0) {
                finished[pick] = true; done++;
                metrics[pick].completion = t;
                metrics[pick].turnaround = t - procs[pick].arrival;
                metrics[pick].weighted_turn = (double)metrics[pick].turnaround /
(double)procs[pick].burst;
                metrics[pick].waiting = metrics[pick].turnaround - procs[pick].burst;
            } else {
                // 未完成：根据设置调整其动态优先数（模拟“优先数减 1”）
                if (high_is_high) rt[pick].dyn_priority -= degrade; // 数值越大越高，则减小其数
值
                else rt[pick].dyn_priority += degrade;               // 数值越小越高，则增大其数
值
            }
        }
    }

    // 汇总平均值
    double avgT = 0.0, avgW = 0.0;
    for (int i = 0; i < n; ++i) { avgT += metrics[i].turnaround; avgW +=
metrics[i].weighted_turn; }

    SimulationResult res;
    res.timeline = timeline;
    res.per_process = metrics;
```

```
    res.avg_turnaround = avgT / n;
    res.avg_weighted_turn = avgW / n;
    res.makespan = (int)timeline.size();
    return res;
}

// --------------------------- MLFQ 调度（多级反馈队列） ---------------------------

static SimulationResult simulate_mlfq(const vector<Process> &procs,
                                      const vector<int> &quantums,
                                      int boost_interval,
                                      bool verbose) {
    int n = (int)procs.size();
    int L = (int)quantums.size();
    vector<ProcRuntime> rt(n);
    vector<bool> finished(n, false);
    vector<queue<int> > Q(L); // 多个就绪队列，0 为最高级
    vector<Metrics> metrics(n);
    vector<string> timeline;

    for (int i = 0; i < n; ++i) {
        rt[i].remaining = procs[i].burst;
        rt[i].start_time = -1;
        rt[i].completion_time = -1;
        rt[i].dyn_priority = procs[i].priority;
        rt[i].level = 0; rt[i].used_in_quantum = 0;
    }

    // 入队（将进程 i 放入指定层级 level 的队列）
    auto enqueue = [&](int i, int level) {
        if (level < 0) level = 0;
        if (level >= L) level = L - 1;
        rt[i].level = level;
        Q[level].push(i);
    };

    // 在当前时刻 now，将刚到达的进程放入最高级队列
    auto inject_arrivals = [&](int now) {
        for (int i = 0; i < n; ++i) {
            if (!finished[i] && procs[i].arrival == now) {
                enqueue(i, 0);
                if (verbose) {
                    cout << "t=" << now << " 到达=" << procs[i].name << " 入队 Q0\n";
                }
            }
        }
    };

    int t = 0;        // 当前时间
    int done = 0;     // 已完成数量
    int current = -1; // 当前正在运行的进程编号（-1 表示空闲）

    inject_arrivals(0);
```

```
    while (done < n) {
        // 若所有队列都空，且此刻没有已到达的就绪进程，则直接空闲到下一次到达
        bool queues_empty = true;
        for (int l = 0; l < L; ++l) if (!Q[l].empty()) { queues_empty = false; break; }
        if (queues_empty && current == -1) {
            int next_arr = INT_MAX;
            for (int i = 0; i < n; ++i) if (!finished[i]) next_arr = min(next_arr,
procs[i].arrival);
            if (next_arr > t) {
                for (int k = t; k < next_arr; ++k) timeline.push_back("-");
                t = next_arr;
                inject_arrivals(t);
            }
        }

        // 周期性“全局提升”（防止低级队列饥饿）。仅提升排队中的进程。
        if (boost_interval > 0 && t > 0 && (t % boost_interval) == 0) {
            vector<int> stash;
            for (int l = 0; l < L; ++l) {
                while (!Q[l].empty()) { stash.push_back(Q[l].front()); Q[l].pop(); }
            }
            for (size_t s = 0; s < stash.size(); ++s) enqueue(stash[s], 0);
            if (verbose && !stash.empty()) cout << "t=" << t << " 全局提升 " << stash.size()
<< " 个进程至 Q0\n";
        }

        // 选择一个要运行的进程：始终从最高级的非空队列取队首
        if (current == -1) {
            for (int l = 0; l < L; ++l) {
                if (!Q[l].empty()) {
                    current = Q[l].front(); Q[l].pop();
                    rt[current].used_in_quantum = 0; // 重新开始计算该层的时间片
                    break;
                }
            }
        } else {
            // 若当前进程不是在最高优先级队列中运行，且更高层队列出现了进程，到下一个时间单位需要抢占
            for (int l = 0; l < rt[current].level; ++l) {
                if (!Q[l].empty()) {
                    // 将当前进程暂时放回其所在层级队列尾部
                    Q[rt[current].level].push(current);
                    current = -1;
                    break;
                }
            }
            if (current == -1) continue; // 让循环下一轮重新选择
        }

        if (current == -1) {
            // 仍无可运行进程，则时间前进一步（空闲）
            timeline.push_back("-");
            t++;
```

```
            inject_arrivals(t);
            continue;
        }

        // 运行 1 个时间单位
        if (rt[current].start_time < 0) {
            rt[current].start_time = t;
            metrics[current].response = t - procs[current].arrival;
        }
        timeline.push_back(procs[current].name);
        t++;
        rt[current].remaining--;
        rt[current].used_in_quantum++;
        if (verbose) {
            cout << "t=" << setw(3) << (t - 1)
                 << " 运行=" << procs[current].name
                 << " 所在队列=L" << rt[current].level
                 << " 时间片使用=" << rt[current].used_in_quantum
                 << "/" << quantums[rt[current].level]
                 << " 剩余=" << rt[current].remaining << "\n";
        }
        // 处理新到达
        inject_arrivals(t);

        // 判断是否完成
        if (rt[current].remaining == 0) {
            metrics[current].completion = t;
            metrics[current].turnaround = t - procs[current].arrival;
            metrics[current].weighted_turn = (double)metrics[current].turnaround /
(double)procs[current].burst;
            metrics[current].waiting = metrics[current].turnaround - procs[current].burst;
            finished[current] = true;
            done++;
            current = -1; // 下轮重新选择
            continue;
        }

        // 时间片用完但未完成：降低一级并重新入队
        if (rt[current].used_in_quantum >= quantums[rt[current].level]) {
            int next_level = rt[current].level + 1;
            if (next_level >= L) next_level = L - 1; // 已在最低级则保持
            enqueue(current, next_level);
            current = -1;
        }
    }

    // 平均值
    double avgT = 0.0, avgW = 0.0;
    for (int i = 0; i < n; ++i) { avgT += metrics[i].turnaround; avgW +=
metrics[i].weighted_turn; }

    SimulationResult res;
    res.timeline = timeline;
```

```
    res.per_process = metrics;
    res.avg_turnaround = avgT / n;
    res.avg_weighted_turn = avgW / n;
    res.makespan = (int)timeline.size();
    return res;
}

// --------------------------- 结果打印 ---------------------------

// 将时间轴压缩输出：例如 [P1:3][P2:2][-:1] 表示连续运行段
static void print_timeline(const vector<string> &timeline) {
    if (timeline.empty()) { cout << "(空)\n"; return; }
    string last = timeline[0];
    int run = 1;
    cout << "  ";
    for (size_t i = 1; i < timeline.size(); ++i) {
        if (timeline[i] == last) run++;
        else {
            cout << "[" << last << ":" << run << "]";
            last = timeline[i]; run = 1;
        }
        if (i % 24 == 0) cout << "\n  "; // 适当换行避免太长
    }
    cout << "[" << last << ":" << run << "]\n";
}

static void print_result(const vector<Process> &procs, const SimulationResult &res) {
    cout << "\n=== 调度结果 ===\n";
    cout << "时间轴(压缩显示):\n";
    print_timeline(res.timeline);

    cout << "\n进程统计 (按输入顺序):\n";
    cout << left << setw(10) << "进程"
         << right << setw(8) << "到达"
         << setw(8) << "运行"
         << setw(10) << "完成"
         << setw(10) << "周转"
         << setw(10) << "带权周转"
         << setw(10) << "等待"
         << setw(10) << "响应" << "\n";
    for (size_t i = 0; i < procs.size(); ++i) {
        const Process &p = procs[i];
        const Metrics &m = res.per_process[i];
        cout << left << setw(10) << p.name
             << right << setw(8) << p.arrival
             << setw(8) << p.burst
             << setw(10) << m.completion
             << setw(10) << m.turnaround
             << setw(10) << fixed << setprecision(2) << m.weighted_turn
             << setw(10) << m.waiting
             << setw(10) << m.response << "\n";
    }
    cout << "\n平均周转时间: " << fixed << setprecision(2) << res.avg_turnaround << "\n";
```

```
    cout << "平均带权周转时间: " << fixed << setprecision(2) << res.avg_weighted_turn << "\n";
    cout << "总完成时间 (makespan): t=" << res.makespan << "\n";
}

// --------------------------- 主函数 ---------------------------

int main(int argc, char **argv) {
    Args args;
    if (!parse_args(argc, argv, args)) {
        if (argc == 1) print_usage();
        return 0; // 打印完帮助或参数解析失败后退出
    }

    vector<Process> procs;
    if (!read_processes(procs)) {
        cout << "\n示例输入: \n";
        cout << "# name  arrival  burst  priority\n";
        cout << "P1 0 5 3\n";
        cout << "P2 1 3 4\n";
        cout << "P3 2 4 2\n";
        cout << "(将上述保存为 sample.txt, 使用: ./sched --algo=mlfq --quantums=2,4,8 <
sample.txt)\n";
        return 1;
    }

    print_input_summary(procs);

    SimulationResult res;
    if (args.algo == "fcfs") {
        cout << "\n算法: 先来先服务 (FCFS)\n";
        res = simulate_fcfs(procs);
    } else if (args.algo == "hpf") {
        cout << "\n算法: 最高优先数 (HPF)" << (args.hpf_preemptive ? " [抢占式, 时间片=1]" : "
[非抢占]") << "\n";
        cout << "优先级规则: 数值越" << (args.priority_high_is_high ? "大" : "小") << "优先级越高
\n";
        if (args.hpf_preemptive) cout << "未完成时每用 1 个时间片动态优先数调整: " <<
(args.priority_high_is_high ? "-" : "+") << args.hpf_degrade << "\n";
        res = simulate_hpf(procs, args.hpf_preemptive, args.priority_high_is_high,
args.hpf_degrade, args.verbose);
    } else if (args.algo == "mlfq") {
        cout << "\n算法: 多级反馈队列 (MLFQ), 级数=" << (int)args.mlfq_quantums.size() << ", 各
级时间片=";
        for (size_t i = 0; i < args.mlfq_quantums.size(); ++i) { cout << (i?",":"") <<
args.mlfq_quantums[i]; }
        cout << "\n";
        if (args.mlfq_boost_interval > 0) cout << "周期性全局提升: 每 " <<
args.mlfq_boost_interval << " 个时间单位\n";
        res = simulate_mlfq(procs, args.mlfq_quantums, args.mlfq_boost_interval,
args.verbose);
    } else {
        cout << "未知算法: " << args.algo << "\n";
        print_usage();
```

```
        return 1;
    }

    print_result(procs, res);
    return 0;
}
```