信息与软件工程学院上机实验报告

(第 4 次)

一、实验名称

磁盘调度算法模拟

二、实验目的及要求

实验目的

1. 加深对磁盘调度的概念及磁盘调度算法的理解；
2. 在了解和掌握磁盘调度算法的基础上，编写磁盘调度算法通用程序，将调试结果输出，再检测和笔算的一致性。

实验要求

1. 了解磁盘调度；
2. 理解利用磁盘调度算法进行调度的原理；
3. 至少实现两种算法求平均寻道数。

三、实验环境

- 开发语言：C++
- 实验环境：Linux (Ubuntu 24.04 LTS)
- 编译工具：g++ (C++17)

四、实验设计

1. 实验步骤

设计思想

- 用统一的输入与统计口径比较四种算法：在相同的磁道范围、初始磁头位置、初始方向与请求序列下，计算总寻道长度与平均寻道长度。
- 记录两类序列：
  - 服务顺序：实际被服务的请求顺序。
  - 移动路径：从初始位置出发的完整移动轨迹，包含到边界/换端的节点（便于手算验证）。
- 指标定义（设移动路径为 $h_0, h_1, \ldots, h_n$，其中 h_0 为初始磁头位置，$h_1 \ldots h_n$ 为每次“移动到的新位置”，恰好覆盖了所有请求与必要的边界/换端移动）：

$$S = \sum_{i=1}^{n} |h_i - h_{i-1}|, \quad \bar{S} = \frac{S}{m}$$

其中 S 为总寻道长度，$\bar{S}$ 为平均寻道长度，m 为请求数量。

说明：本报告中 SCAN 统计“走到边界再折返”的移动，C-SCAN 统计“到边界并换端到另一边界”的移动，以体现两种算法的经典特性与公平性；如需与不计边界/换端的教材口径对齐，可在程序中去除对应两处移动。

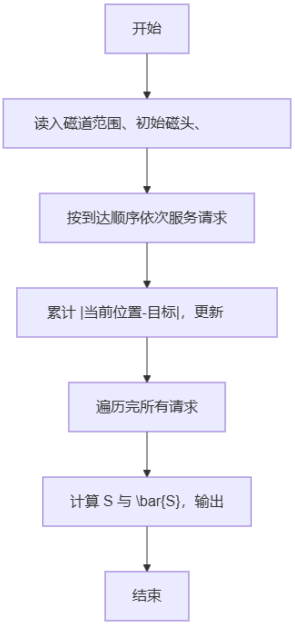
核心数据结构

- 输入结构 `InputData`：`disk_min`、`disk_max`、`start`、`direction_up`、`requests`。

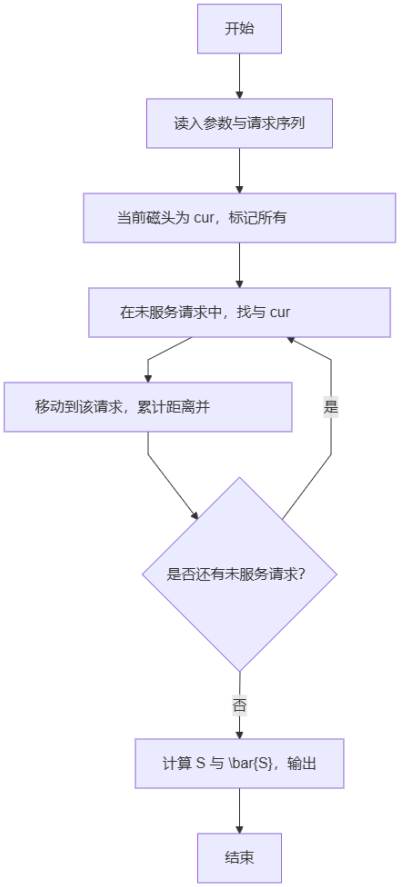
- 结果结构 `ScheduleResult`：`service_order`、`full_path`、`total_movement`、`average_movement`。
- 小工具函数：`moveHead(from, to, total, path)` 负责累计距离并记录路径。

算法流程

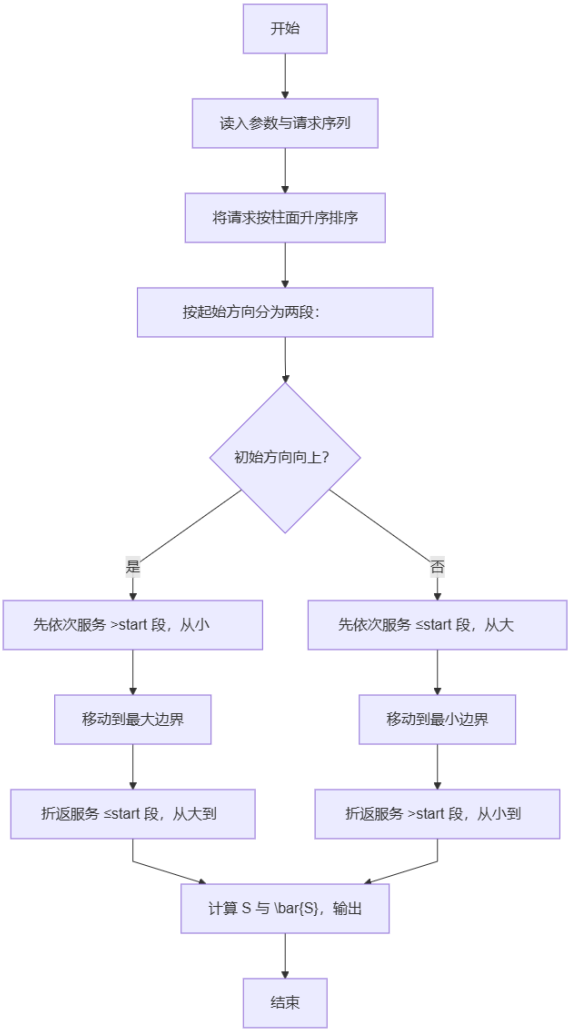
FCFS（先来先服务）流程



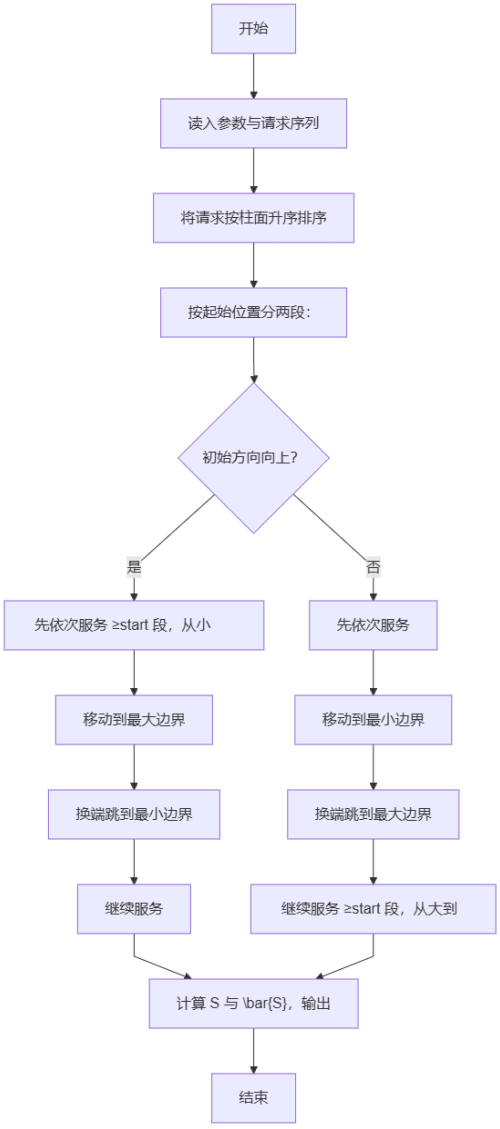
SSTF（最短寻道优先）流程



SCAN（电梯算法）流程


开始
读入参数与请求序列
将请求按柱面升序排序
按起始方向分为两段：
初始方向向上？
是
否
先依次服务 >start 段，从小
先依次服务 ≤start 段，从大
移动到最大边界
移动到最小边界
折返服务 ≤start 段，从大到
折返服务 >start 段，从小到
计算 S 与 \bar{S}，输出
结束

C-SCAN（循环扫描）流程



---

2. 调试过程与实验结果

测试数据

- 磁道范围：`[0, 199]`
- 初始磁头：`53`
- 初始方向：向上（从小到大）
- 请求序列：`98, 183, 37, 122, 14, 124, 65, 67`

执行结果

```
→ ./disk_scheduling
输入参数摘要
磁道范围: [0, 199]
初始磁头: 53
初始方向: 向上(up, 从小到大)
请求序列: 98 → 183 → 37 → 122 → 14 → 124 → 65 → 67

==== FCFS (先来先服务)  ====
服务顺序:        98 → 183 → 37 → 122 → 14 → 124 → 65 → 67
移动路径(含边界): 53 → 98 → 183 → 37 → 122 → 14 → 124 → 65 → 67
总寻道长度: 640
平均寻道长度: 80.000

==== SSTF (最短寻道优先)  ====
服务顺序:        65 → 67 → 37 → 14 → 98 → 122 → 124 → 183
移动路径(含边界): 53 → 65 → 67 → 37 → 14 → 98 → 122 → 124 → 183
总寻道长度: 236
平均寻道长度: 29.500

==== SCAN (电梯算法)  ====
服务顺序:        65 → 67 → 98 → 122 → 124 → 183 → 37 → 14
移动路径(含边界): 53 → 65 → 67 → 98 → 122 → 124 → 183 → 199 → 37 → 14
总寻道长度: 331
平均寻道长度: 41.375

==== C-SCAN (循环扫描)  ====
服务顺序:        65 → 67 → 98 → 122 → 124 → 183 → 14 → 37
移动路径(含边界): 53 → 65 → 67 → 98 → 122 → 124 → 183 → 199 → 0 → 14 → 37
总寻道长度: 382
平均寻道长度: 47.750

==== 结果对比汇总 ====
算法                                    总寻道长度                    平均寻道长度
FCFS                                                640                                80.000
SSTF                                                236                                29.500
SCAN                                                331                                41.375
C-SCAN                                              382                                47.750
```

结果对比与手算验证

- 以 FCFS 路径 `53 -> 98 -> 183 -> 37 -> 122 -> 14 -> 124 -> 65 -> 67` 为例：
  - $S = |98-53| + |183-98| + |37-183| + |122-37| + |14-122| + |124-14| + |65-124| + |67-65| = 640$
  - $\bar{S} = 640/8 = 80.0$
- 其余算法按程序打印的“移动路径（含边界/换端）”逐段取绝对值相加即可核对。

结论概览：

- SSTF 在该样例下平均寻道最小；
- FCFS 简单但可能产生较大回走；
- SCAN/C-SCAN 更公平，体现顺着方向服务的特点，但会计入到边界/换端的移动；
- 若采用不计边界/换端移动的口径，统计值会略有下降，但相对趋势通常不变。

---

3. 总结

结果分析

- SSTF 倾向优先服务近邻请求，通常降低平均寻道长度，但可能导致远端请求长期等待。
- SCAN 沿固定方向移动、到边界再折返，队列等待更加公平，适用于机械臂往返的设备模型。
- C-SCAN 只向一个方向服务，通过换端保持“服务顺序公平”，减少不同位置的等待差异。
- FCFS 顺序简单，适合请求较平滑、分布近似均匀的场景；在“跳变大”的序列下寻道开销偏大。

思考题

1. SCAN 与 LOOK 有何差异?
    - LOOK 在实际最后一个请求处折返（不一定到硬边界）；SCAN 则到达边界再折返。C-LOOK 与 C-SCAN 的区别同理。
2. 实际系统为何可能更偏好 LOOK/C-LOOK?
    - 避免无谓的边界移动，减少平均寻道；同时保持与电梯策略类似的公平性。
    -

心得体会

通过本实验，我在统一输入与统计口径下实现并对比了 FCFS、SSTF、SCAN 与 C-SCAN。借助“移动路径（含边界/换端）”的打印，能够直观地用手算验证总寻道与平均寻道。实验帮助我理解了“局部优化（SSTF）”与“全局公平（SCAN/C-SCAN）”的权衡关系，也认识到不同统计口径（是否计入边界/换端）会影响绝对数值但不改整体趋势。实现过程中，我将逻辑模块化，保证新手也能快速读懂与复用。

改进意见

统一统计口径并对比 LOOK/C-LOOK，补充极端序列的简要测试与说明。

---

4. 附录

```
#include <bits/stdc++.h>
using namespace std;


// 结果结构：包括“服务顺序”（按请求被服务的顺序），“完整移动路径”（包含起点与边界跳转），“总寻道长度”与“平均寻道长度”。
struct ScheduleResult {
    vector<int> service_order;   // 服务请求的顺序（只包含请求本身）
    vector<int> full_path;       // 磁头移动路径（包含起点、可能的边界点/换端）
    long long total_movement = 0; // 总寻道长度（总移动柱面数）
    double average_movement = 0;  // 平均寻道长度（总 / 请求数量）
};

// 小工具：将一段移动计入 total，并把终点 push 到路径里
static void moveHead(int from, int to, long long &total, vector<int> &path) {
    total += llabs((long long)to - (long long)from);
    path.push_back(to);
}

// 计算平均值的小工具
static void finalizeAverage(ScheduleResult &res, size_t request_count) {
    if (request_count == 0) res.average_movement = 0.0;
    else res.average_movement = (double)res.total_movement / (double)request_count;
}

// 一、FCFS（先来先服务）：按到达顺序逐个服务，最直观。
```

```cpp
ScheduleResult fcfs(const vector<int> &requests, int start) {
    ScheduleResult res;
    res.full_path.push_back(start); // 路径从初始磁头位置开始

    int cur = start;
    for (int r : requests) {
        moveHead(cur, r, res.total_movement, res.full_path);
        res.service_order.push_back(r);
        cur = r;
    }

    finalizeAverage(res, requests.size());
    return res;
}

// 二、SSTF（最短寻道优先）：每次选择与当前磁头距离最近的请求。
// 实现方式：线性扫描找最近的未完成请求（新手友好、易懂）。
ScheduleResult sstf(vector<int> requests, int start) {
    ScheduleResult res;
    res.full_path.push_back(start);

    int n = (int)requests.size();
    vector<bool> used(n, false);
    int cur = start;
    for (int served = 0; served < n; ++served) {
        // 找到最近的未服务请求；若有距离相同，选柱面号较小者（固定一个稳定的选择规则）。
        int best_idx = -1;
        long long best_dist = LLONG_MAX;
        for (int i = 0; i < n; ++i) {
            if (used[i]) continue;
            long long d = llabs((long long)requests[i] - (long long)cur);
            if (d < best_dist || (d == best_dist && requests[i] < requests[best_idx])) {
                best_dist = d;
                best_idx = i;
            }
        }

        // 移动到该请求位置，计入结果
        moveHead(cur, requests[best_idx], res.total_movement, res.full_path);
        res.service_order.push_back(requests[best_idx]);
        cur = requests[best_idx];
        used[best_idx] = true;
    }

    finalizeAverage(res, requests.size());
    return res;
}

// 三、SCAN（电梯算法）：
// 思想：磁头沿一个方向移动，服务该方向上的所有请求，抵达边界后折返，再服务另一侧。
ScheduleResult scan(vector<int> requests, int start, int disk_min, int disk_max, bool direction_up) {
    ScheduleResult res;
    res.full_path.push_back(start);

    // 排序后按方向分两段处理
    sort(requests.begin(), requests.end());
    vector<int> lower, upper; // lower: <= start; upper: > start
    for (int r : requests) {
        if (r <= start) lower.push_back(r);
```

```
        else upper.push_back(r);
    }

    int cur = start;

    if (direction_up) {
        // 先向上：从小到大处理大于 start 的请求
        for (int r : upper) {
            moveHead(cur, r, res.total_movement, res.full_path);
            res.service_order.push_back(r);
            cur = r;
        }
        // 走到最大边界
        if (cur != disk_max) {
            moveHead(cur, disk_max, res.total_movement, res.full_path);
            cur = disk_max;
        }
        // 折返向下：从大到小处理 <= start 的请求
        for (int i = (int)lower.size() - 1; i >= 0; --i) {
            int r = lower[i];
            moveHead(cur, r, res.total_movement, res.full_path);
            res.service_order.push_back(r);
            cur = r;
        }
    } else {
        // 先向下：从大到小处理 <= start 的请求
        for (int i = (int)lower.size() - 1; i >= 0; --i) {
            int r = lower[i];
            moveHead(cur, r, res.total_movement, res.full_path);
            res.service_order.push_back(r);
            cur = r;
        }
        // 走到最小边界
        if (cur != disk_min) {
            moveHead(cur, disk_min, res.total_movement, res.full_path);
            cur = disk_min;
        }
        // 折返向上：从小到大处理 > start 的请求
        for (int r : upper) {
            moveHead(cur, r, res.total_movement, res.full_path);
            res.service_order.push_back(r);
            cur = r;
        }
    }

    finalizeAverage(res, requests.size());
    return res;
}

// 四、C-SCAN（循环扫描）：
// 思想：磁头只向一个方向移动，服务完该方向的请求后，直接“跳到”另一端，再继续同向服务。
ScheduleResult cscan(vector<int> requests, int start, int disk_min, int disk_max, bool direction_up) {
    ScheduleResult res;
    res.full_path.push_back(start);

    sort(requests.begin(), requests.end());
    vector<int> lower, upper; // lower: < start; upper: >= start (这里把等于 start 归入上段以固定顺序)
    for (int r : requests) {
        if (r < start) lower.push_back(r);
```

```
        else upper.push_back(r);
    }

    int cur = start;
    if (direction_up) {
        // 向上：先服务大于等于 start 的，从小到大
        for (int r : upper) {
            moveHead(cur, r, res.total_movement, res.full_path);
            res.service_order.push_back(r);
            cur = r;
        }
        // 走到最大边界
        if (cur != disk_max) {
            moveHead(cur, disk_max, res.total_movement, res.full_path);
            cur = disk_max;
        }
        // 换端跳到最小边界（计入移动）
        if (disk_max != disk_min) {
            moveHead(cur, disk_min, res.total_movement, res.full_path);
            cur = disk_min;
        }
        // 继续向上：服务小于 start 的请求，从小到大
        for (int r : lower) {
            moveHead(cur, r, res.total_movement, res.full_path);
            res.service_order.push_back(r);
            cur = r;
        }
    } else {
        // 向下：先服务小于等于 start 的，从大到小
        // 为与 scan 的“lower<=start”对称，这里也可包含等于；为固定顺序，此处保持 lower: < start
        for (int i = (int)lower.size() - 1; i >= 0; --i) {
            int r = lower[i];
            moveHead(cur, r, res.total_movement, res.full_path);
            res.service_order.push_back(r);
            cur = r;
        }
        // 走到最小边界
        if (cur != disk_min) {
            moveHead(cur, disk_min, res.total_movement, res.full_path);
            cur = disk_min;
        }
        // 换端跳到最大边界（计入移动）
        if (disk_max != disk_min) {
            moveHead(cur, disk_max, res.total_movement, res.full_path);
            cur = disk_max;
        }
        // 继续向下：服务大于等于 start 的请求，从大到小
        for (int i = (int)upper.size() - 1; i >= 0; --i) {
            int r = upper[i];
            moveHead(cur, r, res.total_movement, res.full_path);
            res.service_order.push_back(r);
            cur = r;
        }
    }

    finalizeAverage(res, requests.size());
    return res;
}
```

```cpp
// 打印工具：以 a -> b -> c 的形式输出路径/顺序
static void printArrowList(const vector<int> &v) {
    for (size_t i = 0; i < v.size(); ++i) {
        if (i) cout << " -> ";
        cout << v[i];
    }
    cout << "\n";
}

// 汇总打印一个算法的结果
static void printResult(const string &title, const ScheduleResult &res, size_t req_count) {
    cout << "\n==== " << title << " ====" << "\n";
    cout << "服务顺序:         "; printArrowList(res.service_order);
    cout << "移动路径(含边界): "; printArrowList(res.full_path);
    cout << "总寻道长度: " << res.total_movement << "\n";
    cout << fixed << setprecision(3)
         << "平均寻道长度: " << (req_count ? (double)res.total_movement / (double)req_count : 0.0) <<
"\n";
}

// 简单的输入函数（带基本校验，避免新手的输入踩坑）
struct InputData {
    int disk_min;
    int disk_max;
    int start;
    bool direction_up;          // true 表示初始方向向外（从小到大，up），false 表示向内（从大到小，down）
    vector<int> requests;
};

static InputData defaultSample() {
    // 经典教材样例：磁道范围 0~199，初始磁头 53，方向向上
    // 请求序列：98, 183, 37, 122, 14, 124, 65, 67
    return InputData{
        /*disk_min=*/0,
        /*disk_max=*/199,
        /*start=*/53,
        /*direction_up=*/true,
        /*requests=*/{98, 183, 37, 122, 14, 124, 65, 67}
    };
}

static InputData interactiveInput() {
    InputData in;
    cout << "请输入磁道最小值 disk_min (例如 0) : ";
    cin >> in.disk_min;
    cout << "请输入磁道最大值 disk_max (例如 199) : ";
    cin >> in.disk_max;
    if (in.disk_min > in.disk_max) swap(in.disk_min, in.disk_max);

    cout << "请输入初始磁头位置 start (介于最小与最大之间) : ";
    cin >> in.start;
    if (in.start < in.disk_min) in.start = in.disk_min;
    if (in.start > in.disk_max) in.start = in.disk_max;

    cout << "请选择初始方向 (1 表示向上 up，从小到大；0 表示向下 down，从大到小) : ";
    int dir;
    cin >> dir;
    in.direction_up = (dir != 0);
```

```
    cout << "请输入请求数量 n: ";
    int n;
    cin >> n;
    cout << "请依次输入 " << n << " 个请求柱面号（空格分隔）：\n";
    in.requests.resize(max(0, n));
    for (int i = 0; i < n; ++i) {
        cin >> in.requests[i];
        // 将超出边界的请求“钳位”到范围内，避免越界（也可提示错误让用户重输，这里做友好处理）
        if (in.requests[i] < in.disk_min) in.requests[i] = in.disk_min;
        if (in.requests[i] > in.disk_max) in.requests[i] = in.disk_max;
    }
    return in;
}

int main(int argc, char **argv) {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    bool use_interactive = false;
    for (int i = 1; i < argc; ++i) {
        string s = argv[i];
        if (s == "--interactive" || s == "-i") use_interactive = true;
    }

    InputData in = use_interactive ? interactiveInput() : defaultSample();

    // 打印输入摘要，便于核对
    cout << "输入参数摘要\n";
    cout << "磁道范围: [" << in.disk_min << ", " << in.disk_max << "]\n";
    cout << "初始磁头: " << in.start << "\n";
    cout << "初始方向: " << (in.direction_up ? "向上(up，从小到大)" : "向下(down，从大到小)") << "\n";
    cout << "请求序列: "; printArrowList(in.requests);

    // 分别运行四种算法
    auto r_fcfs  = fcfs(in.requests, in.start);
    auto r_sstf  = sstf(in.requests, in.start);
    auto r_scan  = scan(in.requests, in.start, in.disk_min, in.disk_max, in.direction_up);
    auto r_cscan = cscan(in.requests, in.start, in.disk_min, in.disk_max, in.direction_up);

    // 分别打印各算法详细结果
    printResult("FCFS（先来先服务）", r_fcfs, in.requests.size());
    printResult("SSTF（最短寻道优先）", r_sstf, in.requests.size());
    printResult("SCAN（电梯算法）", r_scan, in.requests.size());
    printResult("C-SCAN（循环扫描）", r_cscan, in.requests.size());

    // 汇总对比（仅统计值，便于直观比较）
    cout << "\n==== 结果对比汇总 ====" << "\n";
    cout << left << setw(20) << "算法"
         << right << setw(40) << "总寻道长度"
         << right << setw(40) << "平均寻道长度" << "\n";

    auto line = [&](const string &name, const ScheduleResult &r){
        cout << left << setw(20) << name
             << right << setw(40) << r.total_movement
             << right << setw(40) << fixed << setprecision(3) << r.average_movement << "\n";
    };

    line("FCFS",  r_fcfs);
    line("SSTF",  r_sstf);
```

```
    line("SCAN",  r_scan);
    line("C-SCAN",r_cscan);

    return 0;
}
```