《操作系统》课程

实验报告

实验名称：	实验五 移动头磁盘调度算法模拟实现与比较
姓　　名：	江家玮
学　　号：	**22281188**
日　　期：	**2024.12.05**

目录

1 开发环境

Visual Studio Code（VSCode）是一个流行的开源代码编辑器，广泛用于软件开发，尤其适合 Web 开发、Python 开发、C++开发等。它是由微软开发的，具有强大的功能和灵活的扩展性。以下是 VSCode 开发环境的简要介绍：

（1）核心特点

- 轻量级：VSCode 是一个轻量级的代码编辑器，启动速度快，占用资源少，适合快速开发和调试。
- 跨平台支持：支持 Windows、macOS 和 Linux 系统，可以在多种平台上使用。
- 多语言支持：内置对多种编程语言的支持，如 JavaScript、Python、C++、Java、Go、PHP 等。
- 智能代码补全：通过 IntelliSense 提供智能代码补全，帮助开发者更高效地编写代码。
- 内置终端：可以直接在编辑器内运行终端命令，便于开发过程中执行脚本、运行应用等。
- 调试功能：VSCode 提供了强大的调试功能，支持设置断点、查看变量、调用栈等，能够帮助开发者快速定位代码中的问题。

（2）扩展性

- 插件市场：VSCode 拥有一个庞大的插件市场，可以根据自己的需要安装各种插件，增加对特定编程语言、框架或工具的支持。例如，Python、Node.js、Docker、Git 等都有相关插件。
- 集成 Git：VSCode 集成了 Git，可以直接在编辑器内查看、提交和管理代码版本控制。

（3）常用功能

- 代码高亮：支持语法高亮显示，让代码更易读和理解。
- 代码片段：可以自定义代码片段，提高编码效率。
- 文件管理：支持多文件编辑、拖放文件、文件资源管理等，便于管理项目文件。
- 实时预览：对于 Web 开发，VSCode 可以通过插件支持实时预览，方便查看 HTML/CSS/JavaScript 的效果。

（4）调试与终端

- 集成调试：VSCode 内置调试工具，支持多种编程语言的调试，开发者可以设置断点、查看变量值、查看堆栈信息等。
- 集成终端：VSCode 提供内置终端功能，支持运行 Shell、PowerShell、Command Prompt 等命令，开发者无需切换到外部终端，便于多任务操作。

（5）工作区与项目管理

- 工作区：VSCode 支持“工作区”概念，可以为不同的项目设置不同的配置和环境，确保项目之间的设置不会互相影响。

- 任务自动化：通过集成任务和自定义脚本，可以将常见的操作（如构建、测试、部署）自动化，提高开发效率。

2 运行环境

VSCode 的运行环境是指支持和运行 Visual Studio Code 编辑器的硬件和软件环境。

（1）操作系统要求

VSCode 支持以下操作系统：

- Windows：Windows 7、Windows 8、Windows 10、Windows 11（32 位和 64 位版本都支持）。
- macOS：支持 macOS 10.11 或更高版本。
- Linux：支持大多数 Linux 发行版（如 Ubuntu、Debian、Fedora、Red Hat、Arch 等），通常需要 64 位操作系统。

（2）硬件要求

- 内存：至少 1 GB 的 RAM（推荐 2 GB 或更多）。
- 处理器：任何现代的处理器（Intel Core i3 或更高，AMD 同类处理器）。
- 存储：至少 200 MB 的可用磁盘空间来安装 VSCode。

VSCode 是一个跨平台、轻量级但功能强大的编辑器，能够在各种操作系统上运行。它的运行环境主要依赖于操作系统、硬件资源、必要的开发工具（如 Git、Node.js、编译器等）以及通过插件扩展的语言和框架支持。理解和配置这些运行环境能帮助开发者充分发挥 VSCode 的潜力，提升开发效率。

3 测试环境

VSCode 的测试环境是指在 VSCode 中配置和运行测试代码的环境，包括如何设置自动化测试框架、如何调试测试代码以及如何使用插件和工具来提高测试过程的效率。VSCode 本身并不包含完整的测试框架，但它为开发者提供了良好的支持，可以集成多种测试框架并进行调试。VSCode 支持多种编程语言的测试框架，包括 JavaScript、Python、Java、C++ 等，且通过安装插件，可以使 VSCode 成为一个强大的测试环境。本实验采用 C++的测试框架。

4 实验目的

理解并掌握主要的移动头磁盘调度算法的基本设计思想和编程实现要旨。

5 实验内容

利用标准 C 语言，编程设计与实现关于移动头磁盘调度的先来先服务调度算法（FCFS）、最短寻道时间优先调度算法（SSTF）、电梯调度算法（SCAN）、循环式单向电梯调度算法（CSCAN）、双队列电梯调度算法（FSCAN），并随机发生一组磁盘访问事件（磁道号）序列开展有关算法的测试及性能比较。

6 实验代码分析

6.1 代码主要组成部分

该代码实现了磁盘调度算法的模拟程序，旨在对多种调度策略的平均寻道时间进行比较和分析。代码的主要组成部分如下：

(1) 随机请求生成：利用随机函数生成磁盘的访问请求序列，范围为 0 到 199。

(2) 磁盘调度算法实现：包括以下五种磁盘调度算法的实现：

- 先来先服务（FCFS）： 按照请求顺序处理，不考虑磁头移动距离。
- 最短寻道时间优先（SSTF）： 每次选择距离磁头当前位置最近的请求。
- 扫描算法（SCAN）： 磁头按单方向移动，处理所有请求后反向移动。
- 循环扫描算法（CSCAN）： 磁头按单方向移动到末端后直接返回起点。
- 分区扫描算法（FSCAN）： 请求分成两队，分批处理，减少动态插入的干扰。

(3) 运行结果统计：对每种算法运行 100 次，计算其平均寻道时间。

(4) 主函数：循环调用各算法并输出结果。

6.2 代码详解

6.2.1 FCFS 算法

```
1.    int FCFS(const vector<int>& t, int pos) {
2.        int sum = 0;
3.        Show("FCFS", t); // 输出 FCFS 算法的访问序列
4.        for (int i = 0; i < t.size(); i++) {
5.            sum += abs(pos - t[i]); // 计算当前位置与当前请求的距离，并累加到总和
6.            pos = t[i]; // 更新磁头位置为当前请求的位置
7.        }
8.        return sum; // 返回总磁头移动距离
9.    }
```

- 功能：FCFS（First-Come-First-Served，先来先服务）是一种简单的磁盘调度算法，磁头按照请求到达的顺序逐一处理。
- 流程：
  ① 首先调用 Show 函数输出 FCFS 的访问顺序。
  ② 然后遍历请求序列 t，对于每一个请求，计算磁头当前位置 pos 到当前请求 t[i] 的绝对距离 abs(pos - t[i])，并累加到 sum 中。
  ③ 每次处理完请求后，更新磁头位置 pos = t[i]。
  ④ 最后返回总磁头移动的距离 sum。

6.2.2 SSTF 算法

```
int findClose(const vector<int>& t, int pos) {
    int minDistance = INT_MAX;  // 初始化最小距离为最大值
    int index = -1;  // 初始化最小距离的索引为-1
    for (int i = 0; i < t.size(); i++) {
        if (t[i] == -1) continue;  // 已经处理过的请求跳过
        int distance = abs(pos - t[i]);  // 计算当前磁头位置与请求的位置的距离
        if (minDistance > distance) {  // 如果当前请求距离更近，更新最小距离和索引
            minDistance = distance;
            index = i;
        }
    }
    return index;  // 返回距离当前磁头位置最近的请求的索引
}

int SSTF(vector<int> t, int pos) {
    vector<int> show;  // 用于记录访问顺序
    int sum = 0;  // 总的磁头移动距离
    for (int i = 0; i < t.size(); i++) {
        int index = findClose(t, pos);  // 找到距离当前磁头位置最近的请求
        if (index == -1) {  // 如果没有可用请求，退出
            break;
        } else {
            show.push_back(t[index]);  // 将当前请求加入访问顺序
            sum += abs(pos - t[index]);  // 累加磁头移动距离
            pos = t[index];  // 更新磁头位置
            t[index] = -1;  // 将已访问的请求标记为已处理
        }
    }
    Show("SSTF", show);  // 输出 SSTF 的访问顺序
    return sum;  // 返回总磁头移动距离
}

```

- 功能：SSTF（Shortest Seek Time First，最短寻道时间优先）算法每次选择距离当前磁头　　位置最近的请求来处理。
- 流程：

① findClose 函数遍历请求队列，计算每个请求与当前磁头位置的距离，返回距离最小的请求索引。

② 在 SSTF 函数中，逐个选择离当前磁头最近的请求，计算并累加磁头的移动距离。

③ 访问过的请求标记为 -1，避免重复处理。

④ 最后输出 SSTF 的访问顺序和总移动距离。

6.2.3 SCAN 算法

```
1.  pair<int, int> SCAN(vector<int> t, int pos, int flag = 1) {
2.      int sum = 0; // 累加磁头的移动距离
3.      vector<int> left, right; // 左右两侧的请求队列
4.      vector<int> show; // 记录访问顺序
5.
6.      // 将请求分为左右两侧，左侧请求小于磁头位置，右侧请求大于等于磁头位置
7.      for (auto e : t) {
8.          if (e < pos) left.push_back(e);
9.          else right.push_back(e);
10.     }
11.
12.     // 对左右两侧的请求分别进行排序
13.     sort(left.begin(), left.end());
14.     sort(right.begin(), right.end());
15.
16.     // 按照扫描顺序，首先处理右侧请求
17.     for (auto e : right) {
18.         show.push_back(e);
19.     }
20.
21.     // 移动磁头到右边界，计算磁头移动的距离
22.     sum += maxRightValue - pos;
23.
24.     // 如果左侧有请求，先移动到最左端，然后处理左侧请求
25.     if (!left.empty()) {
26.         sum += maxRightValue - left[0]; // 移动到最左端
27.         reverse(left.begin(), left.end()); // 反转左侧请求顺序（磁头从右向左扫）
28.         for (auto e : left) {
29.             show.push_back(e);
```

```
30.             }
31.         }
32.
33.         // 输出 SCAN 的访问顺序
34.         if (flag) {
35.             Show("SCAN", show);
36.         } else {
37.             for (auto e : show) {
38.                 FSCAN_SHOW.push_back(e);
39.             }
40.         }
41.
42.         return {sum, pos}; // 返回总的磁头移动距离和最终磁头位置
43.     }
```

- 功能：SCAN（扫描算法）按单一方向（通常是从左到右或从右到左）移动磁头，处理所有请求后反向移动。
- 流程：
  ① 将请求分成左侧和右侧，分别存储小于当前磁头位置的请求和大于等于当前磁头位置的请求。对左侧和右侧请求分别排序。
  ② 首先处理右侧的请求，之后反向处理左侧的请求。
  ③ 计算磁头从当前位置到右边界的移动距离，再处理左侧的请求，更新总移动距离。
  ④ 输出访问顺序。

6.2.4 CSCAN 算法

```
1.      int CSCAN(vector<int> t, int pos) {
2.          int sum = 0; // 总磁头移动距离
3.          vector<int> left, right; // 左右两侧的请求队列
4.          vector<int> show; // 记录访问顺序
5.
6.          // 将请求分为左右两侧
7.          for (auto e : t) {
8.              if (e < pos) left.push_back(e);
9.              else right.push_back(e);
10.         }
11.
12.         // 排序请求
```

```
13.     sort(left.begin(), left.end());
14.     sort(right.begin(), right.end());
15.
16.     // 先处理右侧的请求
17.     for (auto e : right) {
18.         show.push_back(e);
19.     }
20.
21.     // 移动磁头到右边界，并加上磁头从右边界到最左端的移动
22.     sum += maxRightValue - pos + 200;
23.
24.     // 如果左侧有请求，处理左侧请求
25.     if (!left.empty()) {
26.         for (auto e : left) {
27.             show.push_back(e);
28.         }
29.         sum += left.back(); // 添加从最左端到左侧最后请求的距离
30.     }
31.
32.     Show("CSCAN", show); // 输出 CSCAN 的访问顺序
33.     return sum; // 返回总磁头移动距离
34. }
35.
```

- 功能：CSCAN（循环扫描算法）是对 SCAN 的改进，磁头处理完一端的请求后，直接跳到另一端继续扫描，避免反向移动。
- 流程：
  ① 将请求分为左右两侧，并排序。
  ② 处理右侧的请求，之后将磁头移至右边界。
  ③ 移动磁头到最左端并处理左侧请求。
  ④ 计算总的磁头移动距离并输出访问顺序。

6.2.5 FSCAN 算法

```
1.  int FSCAN(vector<int> t, int pos)
2.  {
3.      int sum = 0;
4.      vector<int> t1(t.begin(), t.begin() + 50); // 将前 50 个请求放入队列 1
```

```
5.      vector<int> t2(t.begin() + 50, t.begin() + 100); // 将后 50 个请求放入队列 2
6.      const pair<int, int>& temp = SCAN(t1, pos, 0); // 使用 SCAN 算法处理队列 1
7.      sum += temp.first; // 累加队列 1 的磁头移动距离
8.      sum += SCAN(t2, temp.second, 0).first; // 使用 SCAN 算法处理队列 2
9.      Show("FSCAN", FSCAN_SHOW); // 输出 FSCAN 算法的访问序列
10.     return sum; // 返回总的磁头移动距离
11.  }
```

- 功能：FSCAN 算法结合了 SCAN 算法，并对请求队列进行分割。它首先处理队列 1（前 50 个请求），然后处理队列 2（后 50 个请求）。这种方法避免了队列的动态变化对调度的影响，并且通过分阶段处理来提高调度效率。
- 流程：

1) 队列划分：通过将输入的请求序列 t 划分为两个部分：前 50 个请求进入队列 1（t1），后 50 个请求进入队列 2（t2）。

2) 使用 SCAN 算法：

① 第一步：SCAN(t1, pos, 0) 处理队列 1 中的请求，返回的是一对结果，temp.first 表示磁头移动的总距离，temp.second 表示扫描结束时磁头的位置。

② 第二步：SCAN(t2, temp.second, 0) 处理队列 2 中的请求，使用从队列 1 处理结束时的磁头位置 temp.second 作为新的起始位置。

3) 输出访问序列：Show("FSCAN", FSCAN_SHOW); 将 FSCAN 算法的访问序列输出。

4) 返回总寻道数：返回 sum，即队列 1 和队列 2 的磁头移动距离之和。

6.2.6 main 函数

```
1.  int main() {
2.      int pos = 0; // 初始磁头位置
3.      vector<int> randValue; // 用于存储随机生成的请求序列
4.      srand((unsigned)time(NULL)); // 设置随机种子
5.      getRandomNumber(randValue, pos); // 生成请求序列及初始磁头位置
6.
7.      cout << "FCFS 的平均寻道数为:" << FCFS(randValue, pos) / 100 << endl; // 输出 FCFS 算法的平均寻道数
8.      cout << endl;
9.
10.     cout << "SSTF 的平均寻道数为:" << SSTF(randValue, pos) / 100 << endl; // 输出 SSTF 算法的平均寻道数
11.     cout << endl;
12.
```

```
13.         cout << "SCAN 的平均寻道数为:" << SCAN(randValue, pos).first / 100 << endl;  // 输出
    SCAN 算法的平均寻道数
14.         cout << endl;
15.
16.         cout << "CSCAN 的平均寻道数为:" << CSCAN(randValue, pos) / 100 << endl;  // 输出
    CSCAN 算法的平均寻道数
17.         cout << endl;
18.
19.         cout << "FSCAN 的平均寻道数为:" << FSCAN(randValue, pos) / 100 << endl;  // 输出
    FSCAN 算法的平均寻道数
20.         cout << endl;
21.         return 0;  // 程序结束
22.     }
23.
```

- 功能：main 函数是整个程序的入口点，负责生成随机请求序列并计算和输出各个磁盘调度算法的平均寻道数。
- 流程：

(1) 初始化磁头位置：int pos = 0; 初始化磁头位置为 0。

(2) 生成请求序列：vector<int> randValue; 用来存储随机生成的磁盘请求，getRandomNumber(randValue, pos); 生成随机的请求序列并给定一个初始的磁头位置。

(3) 调用算法并输出结果：

① 通过调用各个算法（FCFS, SSTF, SCAN, CSCAN, FSCAN）分别计算每种算法的磁头移动总距离，并输出它们的平均寻道数（这里的 randValue 是请求序列，pos 是初始磁头位置）。

② / 100 是为了计算平均寻道数，因为请求的总数是 100，输出结果时通过除以 100 来求得平均寻道数。

(4) 结束程序：执行完所有算法计算和输出后，程序通过 return 0; 结束。

6.2.7 辅助函数：getRandomNumber

```
1.     void getRandomNumber(vector<int>& randValue, int& pos) {
2.         for (int i = 0; i < 100; i++) // 生成 100 个随机请求
3.             randValue.push_back(rand() % 200); // 请求值在 0 到 199 之间
4.         pos = rand() % 200; // 随机生成一个初始磁头位置，范围在 0 到 199 之间
5.     }
6.
```

- 功能：生成 100 个随机的磁盘请求，并随机生成一个初始磁头位置。

- 流程：

  ① 使用 rand() 函数生成 100 个范围在 0 到 199 之间的随机请求，并将它们存入 randValue 向量。

  ② 生成一个范围在 0 到 199 之间的随机数，并赋值给 pos，作为磁头的初始位置。

6.2.8 辅助函数：Show

```
1.    void Show(const string& str, const vector<int>& t) {
2.        cout << str << "的访问序列为:"; // 输出算法名称
3.        for (int i = 0; i < t.size(); i++) {
4.            if (i != t.size() - 1) // 不是最后一个元素，输出箭头连接
5.                cout << t[i] << "->";
6.            else
7.                cout << t[i] << endl; // 最后一个元素输出换行
8.        }
9.    }
10.
```

- 功能：输出磁盘请求的访问顺序。
- 流程：

  ① 输出算法的名称（通过传入的字符串 str）。

  ② 遍历请求序列 t，打印每个请求。如果不是最后一个请求，输出 -> 连接，最后一个请求后输出换行。

7 关键数据结构和算法流程

7.1 关键数据结构

(1)vector<int>：请求队列

- 功能：
  - 用于存储磁盘调度中的请求序列。每个整数代表磁盘块号（范围为 0-199），模拟磁盘读写请求的分布情况。
- 作用：
  - 是各调度算法的输入核心数据。调度算法依据请求队列计算磁头的移动顺序及总移动距离。
  - 动态调整的灵活性允许对请求进行排序、分组和修改，适应不同算法（如 SCAN 和 FSCAN）对请求的处理需求。

(2)int pos：磁头当前所在位置

- 功能：

- 表示磁头起始的物理位置，用于每次磁盘访问时计算磁头移动距离。

- 作用：
  - 决定初始调度状态。各调度算法从此位置开始执行，以此为基点进行后续请求处理。

(3)const int maxRightValue, minLeftValue：磁盘边界值

- 功能：
  - 定义磁盘允许访问的最右边界和最左边界（分别为 199 和 0）。
- 作用：
  - 限制磁盘调度范围，防止超出磁盘物理容量。
  - 在算法中，SCAN 和 CSCAN 会使用边界值作为磁头的停止或反转点。

(4)vector<int> FSCAN_SHOW：FSCAN 访问顺序队列

- 功能：
  - 用于存储 FSCAN 算法中的实际访问顺序。
- 作用：
  - 在实验过程中，记录 FSCAN 算法的运行路径，用于调试、验证和展示算法行为。

(5)pair<int, int>：SCAN 算法的返回值

- 功能：
  - 表示磁头移动的总距离和结束位置。
- 作用：
  - 提供中间结果供其他算法使用（如 FSCAN 分阶段扫描会调用 SCAN 返回的结束位置）。

7.2 核心算法流程

(1)FCFS（First-Come, First-Served）算法

- 核心流程：
  - 从初始磁头位置出发，依次访问请求队列中的磁盘块号。
  - 逐次计算当前磁头位置与目标块号之间的绝对距离，并累加到总距离。
  - 更新磁头当前位置为最新访问的块号。
- 特点：
  - 实现容易，无需额外数据结构。
  - 缺乏优化，可能导致较大的寻道开销（如远距离跳跃）。

(2)SSTF（Shortest Seek Time First）算法

- 核心流程：
  - 从初始磁头位置出发，查找与当前位置距离最近的磁盘块号。
  - 移动到该块号，记录移动距离并将该请求标记为已访问（设为-1）。

- 重复上述步骤，直到所有请求访问完毕。

- 特点：
  - 提高效率，减少总寻道时间。
  - 容易发生“饥饿”现象：远离当前磁头的请求可能长期得不到处理。

(3)SCAN（电梯）算法

- 核心流程：
  - 从初始磁头位置出发，按固定方向（如向右）扫描磁盘块。
  - 遇到每个请求时，记录移动距离，并将其加入访问顺序。
  - 到达边界后，转向反方向扫描未处理的请求。
- 特点：
  - 减少反复寻道，适合大量随机分布的请求。
  - 反向扫描带来额外寻道开销。

(4)CSCAN（循环扫描）算法

- 核心流程：
  - 从初始磁头位置出发，按固定方向（如向右）扫描磁盘块。
  - 到达边界后，磁头直接回到另一端的边界（忽略中间请求）。
  - 继续扫描剩余未处理的请求。
- 特点：
  - 保证请求的均匀服务，避免反向扫描的开销。
  - 边界回跳增加了少量额外的寻道开销。

(5)FSCAN（分阶段扫描）算法

- 核心流程：
  - 将请求序列分为两个队列：当前队列和等待队列。
  - 对当前队列执行一次 SCAN 算法，并记录总移动距离和结束位置。
  - 在第一阶段完成后，将等待队列切换为新的当前队列，再次执行 SCAN。
  - 合并两次扫描的移动距离和访问顺序。
- 特点：
  - 动态性强，适合多任务并发。
  - 分阶段可能引入一定的等待时间，适应性依赖请求分布情况。

8 实验结果展示

图 8-1 实验结果展示

9 技术难点及解决方案

9.1 磁头调度算法的多样性与实现

(1) 问题描述： 磁盘调度算法有多种类型，每种算法有不同的策略和行为（如 FCFS、SSTF、SCAN、CSCAN、FSCAN）。这些算法在计算磁头移动路径和距离时有不同的考虑，且每个算法的实现细节差异较大，可能会导致实现时容易出错或难以调试。

(2) 解决方案：

- 模块化设计：每种算法的实现应单独封装为独立的函数，简化代码结构和调试流程。每个算法仅负责自身的调度逻辑，这样便于单独测试和优化。
- 使用辅助函数：如 findClose、FindLarger 等辅助函数可以用于简化每个算法的核心计算逻辑，减少重复代码并提升代码的可读性和可维护性。
- 充分注释和测试：在实现每个算法时，确保对关键部分加以注释，尤其是在对请求队列进行排序、分组、访问顺序更新等操作时。此外，针对每种算法编写单元测试，确保它们按照预期行为运行。

9.2 请求序列的生成与边界控制

(1) 问题描述： 请求序列的随机生成需要确保请求分布合理，同时要考虑磁头的边界条件（最小值为 0，最大值为 199）。如何生成既符合实际情况又能模拟不同访问模式的请求序列是一个难点。

(2) 解决方案：

- 随机数生成：使用标准库中的 rand() 函数生成 0 到 199 之间的请求，并通过 rand() % 200 来确保请求序列的数值在磁盘的边界范围内。
- 确保随机性：为了保证生成的请求序列具有随机性和多样性，使用 srand(time(NULL)) 来设置随机数种子，这样每次实验生成的请求序列都不同。

- 请求序列验证：在生成的请求序列基础上，加入边界检查，确保每个请求值都在设定的磁盘范围内，避免超出磁盘范围的错误。

9.3 FSCAN 算法的实现与请求分组

(1) 问题描述： FSCAN 算法在 SCANS 的基础上增加了请求队列的分组，且需要确保当前队列和等待队列的切换与处理顺序的正确性。分阶段的调度增加了复杂性，且需要准确记录和展示每个阶段的调度结果。

(2) 解决方案：

- 队列分组：将请求序列根据要求分为当前队列和等待队列。可以通过简单的数组或 vector 进行切割，前 50 个请求为当前队列，后 50 个请求为等待队列。确保分组准确并且能够按需切换。
- 状态管理：通过 FSCAN_SHOW 等全局变量来记录 FSCAN 算法中的访问顺序，避免不同阶段的请求混淆。每阶段完成时，更新队列状态并清晰显示。
- 逐阶段调度：FSCAN 的核心在于分别对当前队列和等待队列执行一次 SCAN 操作，因此需要确保每次 SCAN 完成后，都能正确计算移动距离并切换队列进行下一阶段的调度，避免跨队列的错误调度。

10 相关算法性能分析和比较

算法	寻道时间	优点	缺点	适用场景
FCFS	较长	实现简单，易于理解	寻道效率差，可能导致长时间磁头移动	请求到达顺序较规律的情况
SSTF	较短	寻道时间较短	可能导致饥饿现象	请求分布均匀的场景
SCAN	中等	高效的磁头移动，避免反向移动	请求分布不均时性能较差	请求分布较均匀，负载较大时
CSCAN	更短	避免回程，提高了效率	增加某些请求的等待时间	请求分布较均匀，避免反向移动时
FSCAN	中等	通过队列分阶段调度减少等待时间	实现复杂，管理队列的成本较高	请求量大，请求分布不均时

11 实验心得体会

在进行磁盘调度算法模拟实验的过程中，我对磁盘调度的机制和优化策略有了更为深入的理解。通过实现 FCFS、SSTF、SCAN、CSCAN 和 FSCAN 五种算法，我从理论到实践，系统地了解了它们在实际磁盘访问中的表现和优劣。

最初，在阅读关于磁盘调度的相关文献时，我仅仅停留在对这些算法基本概念的理解上。通过本次实验，我不再仅仅是从理论层面去了解算法，而是亲自编写代码并通过实验进行验证。FCFS（先来先服务）算法简单直观，但其缺点也非常明显——它容易导致磁头在磁盘

上的长距离跳跃，从而增加总寻道时间。相比之下，SSTF（最短寻道时间优先）算法通过每次选择距离当前磁头位置最近的请求来优化寻道时间，理论上能够有效减少磁头的移动距离，但也可能造成某些请求长时间得不到处理，产生“饥饿”现象。SCAN 和 CSCAN 算法通过模拟电梯的扫描方式有效减少了磁头反向移动的开销，特别是 CSCAN 算法，通过循环的方式避免了 SCAN 的回程过程，进一步提升了调度效率。

在实现这些算法的过程中，我深刻体会到了每种算法设计背后的权衡。例如，SCAN 和 CSCAN 的复杂度相对较高，但它们显著减少了寻道时间的波动，适用于大规模请求的处理。而 FSCAN 算法则在 SCAN 的基础上进行了分阶段调度的优化，通过将请求队列分为当前队列和等待队列，避免了过多的请求堆积，从而提升了调度的效率和响应速度。

通过本次实验，我不仅掌握了如何实现这些算法，还理解了它们的适用场景和性能差异。在对比不同算法的寻道时间后，我也意识到，实际应用中并没有一种“万能”的调度算法，而是需要根据实际系统的请求特性来选择合适的调度策略。这次实验让我更加深入地理解了操作系统中磁盘调度的复杂性，也锻炼了我在算法设计和优化方面的思维能力。