安徽大学人工智能学院《操作系统》实验报告

学号___WA2214014___ 姓名___杨跃浙___ 年级___大三___

【实验名称】_____实验五 磁盘调度_____

【实验内容】

1.采用最短寻道策略处理；

2.采用扫描（SCAN）策略处理；

1.最短寻道时间优先(SSTF)

基本思想：先对最靠近当前柱面位置的请求进行服务，即先对寻找时间最短的请求进行服务。SSTF 算法总是让寻找时间最短的那个请求先服务，而不管请求访问者到来的先后次序。

(从100号磁道开始)	
被访问的 下一个磁道号	移动距离 (磁道数)
90	10
58	32
55	3
39	16
38	1
18	20
150	132
160	10
184	24
平均寻道长度：27.5	

2.扫描(SCAN)算法

基本思想：SCAN 算法的工作方式类似于电梯的运行原理。磁头从当前位置开始沿一个方向移动，直到到达请求序列的最大位置，然后反向移动，处理回程中的所有请求。

测试样例：参考教材相关内容

【实验要求】

1、 根据上述算法的特点，分别设计数据结构。

2、 编写上述两种算法。

【实验原理】

磁盘调度算法的重要性

磁盘调度算法的设计对于改善整体系统性能至关重要。磁盘操作通常是计算机系统中响应时间和吞吐量的限制因素之一，因为磁盘访问速度远低于内存和 CPU。因此，有效的磁盘调度策略可以显著降低磁头移动距离，从而减少平均寻道时间，提高磁盘的 I/O 性能。

磁盘调度的基本目标

- 最小化寻道时间：寻道时间是磁头从当前位置移动到目标磁道所需的时间。有效的调度算法应当尽量减少寻道时间。

- 公平性：确保所有磁盘请求都能在合理的时间内得到服务，避免“饥饿”现象。
- 最大化吞吐量：提高在单位时间内磁盘完成请求的数量。

常用磁盘调度算法

1．最短寻道时间优先（SSTF）

2．扫描（SCAN）

最短寻道时间优先（SSTF）

基本原理

SSTF 选择待服务的下一个请求是基于距当前磁头位置最近的请求，从而最小化寻道时间。如果磁头当前位于某一磁道上，它将选择最近的请求作为下一个服务对象，无论这些请求是在磁头的内侧还是外侧。

如果 d 是第 i 个请求与当前磁头位置的距离，那么 SSTF 选择的下一个磁道是使得 d 最小的请求：

$$选择 d_{min} = \min(d_1, d_2, d_n)$$

扫描（SCAN）

基本原理

SCAN 算法类似于电梯的运行方式，磁头从一端向另一端移动，并在返回前服务所有请求。当磁头到达轨道的一个极端时，它会改变方向，并在移动过程中服务所有待处理的请求。这种方法可以避免磁头频繁的方向改变，从而优化整体的寻道时间。

设当前磁头位置为 p，向上运动时服务所有请求直到最大请求位置 p_{max}，然后反向移动至最小请求位置p_{min}。总寻道距离为从 p 到 p_{max}的距离加上p_{max}到p_{min}的距离。

$$\text{总寻道距离} = |p - p_{max}| + |p_{max} - p_{min}|$$

实验的实用性

通过实验设计和实现这些磁盘调度算法，不仅能理解理论和算法的实际应用，还能通过实验结果直观地看到不同算法对系统性能的具体影响，例如寻道距离和平均响应时间的变化。

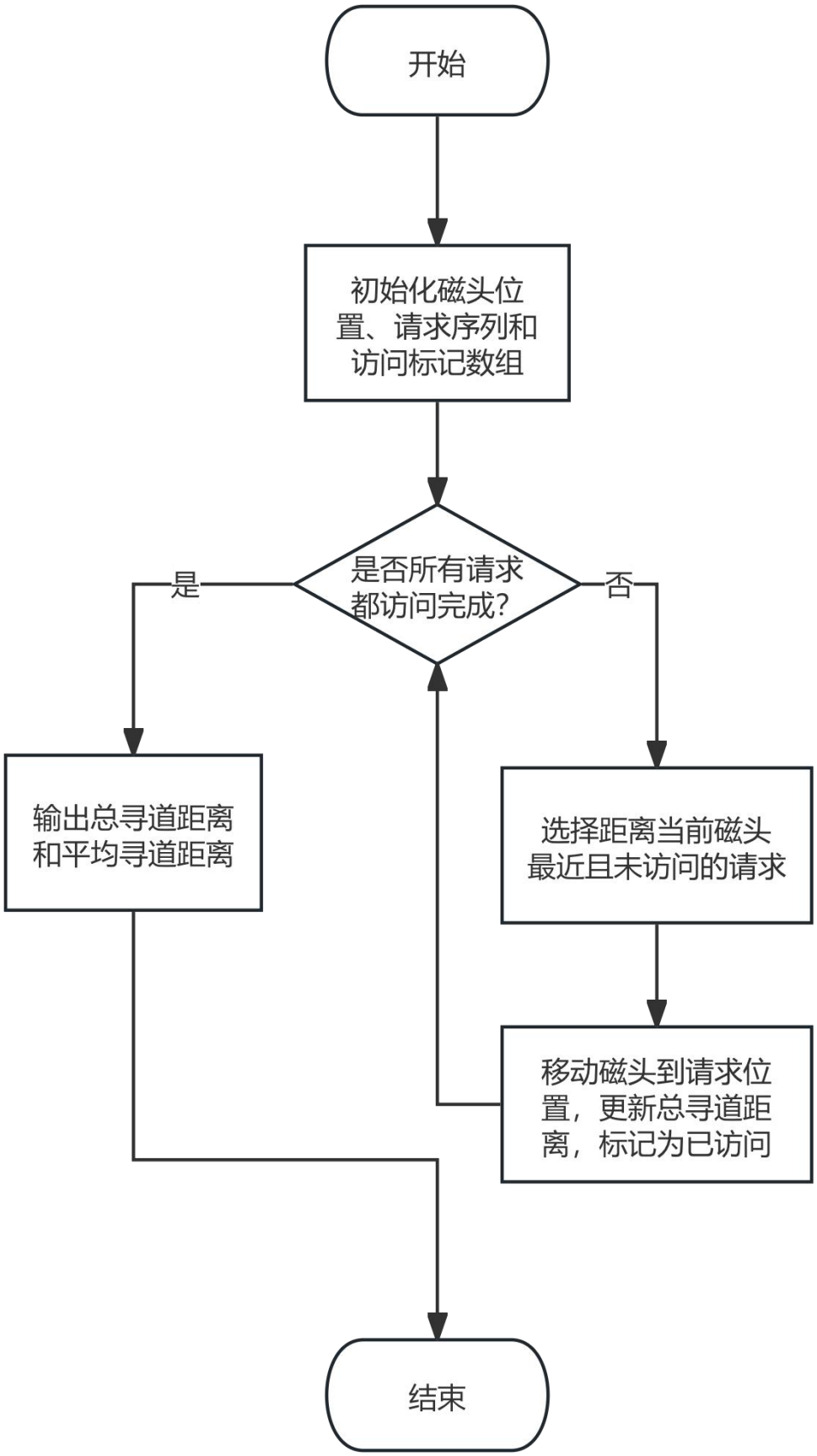
实验背景

在实际操作系统中，磁盘调度是文件系统管理的一个重要方面，关系到数据的读写效率和响应速度。通过编程实践这些算法，能够更好地理解操作系统的复杂性和性能优化的必要性。

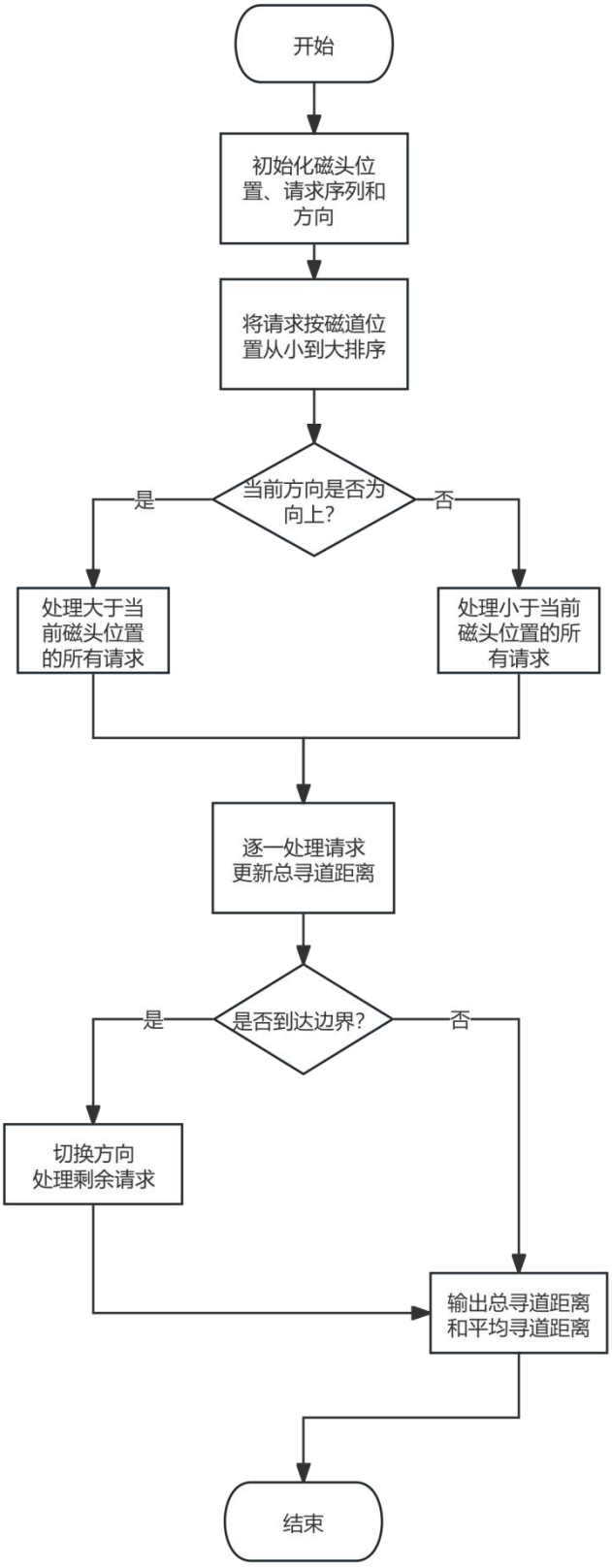
【实验步骤】

1.程序流程图

最短寻道时间优先(SSTF)


开始
初始化磁头位置、请求序列和访问标记数组
是否所有请求都访问完成?
是
否
输出总寻道距离和平均寻道距离
选择距离当前磁头最近且未访问的请求
移动磁头到请求位置，更新总寻道距离，标记为已访问
结束

扫描(SCAN)算法


开始
初始化磁头位置、请求序列和方向
将请求按磁道位置从小到大排序
当前方向是否为向上?
是
否
处理大于当前磁头位置的所有请求
处理小于当前磁头位置的所有请求
逐一处理请求更新总寻道距离
是否到达边界?
是
否
切换方向处理剩余请求
输出总寻道距离和平均寻道距离
结束

2.核心代码

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_REQUESTS 100

int find_closest_request(int current_position, int
requests[], int visited[], int n) {
int min_distance = 10000;
int closest_index = -1;
for (int i = 0; i < n; i++) {
if (!visited[i]) {
int distance = abs(requests[i] - current_position);
if (distance < min_distance) {
min_distance = distance;
closest_index = i;
}
}
}
return closest_index;
}

void sstf(int start, int requests[], int n) {
int visited[MAX_REQUESTS] = {0};
int current_position = start;
int total_seek_distance = 0;

printf("SSTF 调度顺序: ");
for (int i = 0; i < n; i++) {
int closest_index = find_closest_request(current_position,
requests, visited, n);
visited[closest_index] = 1;

int seek_distance = abs(requests[closest_index] -
current_position);
total_seek_distance += seek_distance;

printf("%d ", requests[closest_index]);
```

```
printf("(移动距离: %d)\n", seek_distance);

current_position = requests[closest_index];
}

printf("\n 总寻道距离: %d\n", total_seek_distance);
printf("平均寻道距离: %.2f\n", (float)total_seek_distance /
n);
}

void scan(int start, int requests[], int n, int direction)
{
int sorted_requests[MAX_REQUESTS];
for (int i = 0; i < n; i++) {
sorted_requests[i] = requests[i];
}

// 排序请求
for (int i = 0; i < n - 1; i++) {
for (int j = 0; j < n - 1 - i; j++) {
if (sorted_requests[j] > sorted_requests[j + 1]) {
int temp = sorted_requests[j];
sorted_requests[j] = sorted_requests[j + 1];
sorted_requests[j + 1] = temp;
}
}
}

int total_seek_distance = 0;
int current_position = start;

printf("SCAN 调度顺序: \n");
if (direction == 1) { // 向上方向
// 仅处理大于当前磁道的请求
for (int i = 0; i < n; i++) {
if (sorted_requests[i] >= current_position) {
int seek_distance = abs(sorted_requests[i] -
current_position);
total_seek_distance += seek_distance;
```

```
printf("%d (移动距离: %d)\n", sorted_requests[i],
seek_distance);
current_position = sorted_requests[i];
}
}
// 处理反方向的请求
for (int i = n - 1; i >= 0; i--) {
if (sorted_requests[i] < start) {
int seek_distance = abs(sorted_requests[i] -
current_position);
total_seek_distance += seek_distance;

printf("%d (移动距离: %d)\n", sorted_requests[i],
seek_distance);
current_position = sorted_requests[i];
}
}
} else { // 向下方向
// 仅处理小于当前磁道的请求
for (int i = n - 1; i >= 0; i--) {
if (sorted_requests[i] <= current_position) {
int seek_distance = abs(sorted_requests[i] -
current_position);
total_seek_distance += seek_distance;

printf("%d (移动距离: %d)\n", sorted_requests[i],
seek_distance);
current_position = sorted_requests[i];
}
}
// 处理反方向的请求
for (int i = 0; i < n; i++) {
if (sorted_requests[i] > start) {
int seek_distance = abs(sorted_requests[i] -
current_position);
total_seek_distance += seek_distance;
```

```
printf("%d (移动距离: %d)\n", sorted_requests[i],
seek_distance);
current_position = sorted_requests[i];
}
}
}

printf("\n 总寻道距离: %d\n", total_seek_distance);
printf("平均寻道距离: %.2f\n", (float)total_seek_distance /
n);
}

int main() {
int requests[] = {150, 160, 184, 18, 38, 39, 55, 58, 90}; //
请求序列
int n = sizeof(requests) / sizeof(requests[0]); // 请求数量
int start_position = 100; // 起始位置

printf("========== SSTF ==========\n");
sstf(start_position, requests, n);

printf("\n========== SCAN 向上==========\n");
scan(start_position, requests, n, 1); // 1 表示向上方向

printf("\n========== SCAN 向下==========\n");
scan(start_position, requests, n, 0); // 0 表示向上方向

return 0;
}
```

3.运行结果

```
========== SSTF ==========
SSTF 调度顺序：90（移动距离：10）
58（移动距离：32）
55（移动距离：3）
39（移动距离：16）
38（移动距离：1）
18（移动距离：20）
150（移动距离：132）
160（移动距离：10）
184（移动距离：24）

总寻道距离：248
平均寻道距离：27.56

========== SCAN 向上==========
SCAN 调度顺序：
150（移动距离：50）
160（移动距离：10）
184（移动距离：24）
90（移动距离：94）
58（移动距离：32）
55（移动距离：3）
39（移动距离：16）
38（移动距离：1）
18（移动距离：20）

总寻道距离：250
平均寻道距离：27.78

========== SCAN 向下==========
SCAN 调度顺序：
90（移动距离：10）
58（移动距离：32）
55（移动距离：3）
39（移动距离：16）
38（移动距离：1）
18（移动距离：20）
150（移动距离：132）
160（移动距离：10）
184（移动距离：24）

总寻道距离：248
平均寻道距离：27.56
```

4.结果分析

SSTF 算法

假设：

- 起始位置为 100。
- 请求序列：{150，160，184，18，38，39，55，58，90}。

过程和计算

1．起始位置为 100，寻找最近的请求：

- 150 -> 50
- 160 -> 60
- 184 -> 84

• 18 -> 82
• 38 -> 62
• 39 -> 61
• 55 -> 45
• 58 -> 42
• 90 -> 10 (最近)
移动距离 = 10，新位置 = 90。
2. 当前位置 90，再次寻找最近的请求（排除已服务的 90）：
• 150 -> 60
• 160 -> 70
• 184 -> 94
• 18 -> 72
• 38 -> 52
• 39 -> 51
• 55 -> 35
• 58 -> 32 (最近)
移动距离 = 32，新位置 = 58。
3. 当前位置 58，继续寻找最近的请求（排除已服务的 58）：
• 150 -> 92
• 160 -> 102
• 184 -> 126
• 18 -> 40
• 38 -> 20
• 39 -> 19
• 55 -> 3 (最近)
移动距离 = 3，新位置 = 55。
4. 当前位置 55，继续寻找最近的请求（排除已服务的 55）：
• 150 -> 95
• 160 -> 105
• 184 -> 129
• 18 -> 37
• 38 -> 17
• 39 -> 16 (最近)
移动距离 = 16，新位置 = 39。
5. 当前位置 39，继续寻找最近的请求（排除已服务的 39）：
• 150 -> 111

• 160 -> 121
• 184 -> 145
• 18 -> 21
• 38 -> 1（最近）
移动距离 = 1，新位置 = 38。
6. 当前位置 38，继续寻找最近的请求（排除已服务的 38）：
• 150 -> 112
• 160 -> 122
• 184 -> 146
• 18 -> 20（最近）
移动距离 = 20，新位置 = 18。
7. 当前位置 18，继续寻找最近的请求（排除已服务的 18）：
• 150 -> 132
• 160 -> 142
• 184 -> 166（最近）
移动距离 = 132，新位置 = 150。
8. 当前位置 150，继续寻找最近的请求（排除已服务的 150）：
• 160 -> 10
• 184 -> 34（最近）
移动距离 = 10，新位置 = 160。
9. 当前位置 160，只剩一个请求 184：
• 移动距离 = 24，新位置 = 184。

总结
总寻道距离 = 10 + 32 + 3 + 16 + 1 + 20 + 132 + 10 + 24 = 248
平均寻道距离 = 248 / 9 = 27.56

SCAN 算法
假设：
• 起始位置为 100。
• 请求序列：{150，160，184，18，38，39，55，58，90}。
• 磁头移动方向：向上。

过程和计算
1. 起始位置 100，首先向上扫描至请求序列的最大值，然后反向移

动至最小值。
2. 向上扫描：
• 从 100 到 150：
• 移动距离 = 150 − 100 = 50
• 从 150 到 160：
• 移动距离 = 160 − 150 = 10
• 从 160 到 184（最大请求）：
• 移动距离 = 184 − 160 = 24
到达最高请求 184 后，磁头转向开始向下扫描。
3. 向下扫描：
• 从 184 到 160：
• 移动距离 = 184 − 160 = 24
• 从 160 到 150：
• 移动距离 = 160 − 150 = 10
• 从 150 到 90：
• 移动距离 = 150 − 90 = 60
• 从 90 到 58：
• 移动距离 = 90 − 58 = 32
• 从 58 到 55：
• 移动距离 = 58 − 55 = 3
• 从 55 到 39：
• 移动距离 = 55 − 39 = 16
• 从 39 到 38：
• 移动距离 = 39 − 38 = 1
• 从 38 到 18（最低请求）：
• 移动距离 = 38 − 18 = 20

总结
总寻道距离 = 50（向上）+ 10 + 24 + 24（转向）+ 10 + 60 + 32 + 3 + 16 + 1 + 20 = 250
平均寻道距离 = 250 / 9 = 27.78

SCAN 算法（向下）

假设：
• 起始位置：100。
• 请求序列：{150，160，184，18，38，39，55，58，90}。

• 磁头移动方向：向下。

过程和计算
1．起始位置 100，首先向下扫描至请求序列的最小值，然后反向移动至最大值。
2．向下扫描：
• 从 100 到 90：
• 移动距离 = 100 − 90 = 10
• 从 90 到 58：
• 移动距离 = 90 − 58 = 32
• 从 58 到 55：
• 移动距离 = 58 − 55 = 3
• 从 55 到 39：
• 移动距离 = 55 − 39 = 16
• 从 39 到 38：
• 移动距离 = 39 − 38 = 1
• 从 38 到 18（最小请求）：
• 移动距离 = 38 − 18 = 20
到达最低请求 18 后，磁头转向开始向上扫描。
3．向上扫描：
• 从 18 到 38：
• 移动距离 = 38 − 18 = 20
• 从 38 到 39：
• 移动距离 = 39 − 38 = 1
• 从 39 到 55：
• 移动距离 = 55 − 39 = 16
• 从 55 到 58：
• 移动距离 = 58 − 55 = 3
• 从 58 到 90：
• 移动距离 = 90 − 58 = 32
• 从 90 到 150：
• 移动距离 = 150 − 90 = 60
• 从 150 到 160：
• 移动距离 = 160 − 150 = 10
• 从 160 到 184（最大请求）：
• 移动距离 = 184 − 160 = 24

总结
- 总寻道距离 = 10 + 32 + 3 + 16 + 1 + 20 + 20 + 1 + 16 + 3 + 32 + 60 + 10 + 24 = 248
- 平均寻道距离 = 248 / 9 = 27.56

【实验总结】

对算法的理解

在本次实验中，我深入理解并实践了两种磁盘调度算法：最短寻道时间优先（SSTF）和扫描（SCAN）算法。这两种算法在磁盘调度领域广泛应用，能够有效优化磁头移动过程，从而提升磁盘的响应速度和系统的整体性能。通过实验，我对它们的核心思想和实际实现有了更深刻的体会，同时也经历了算法编程中常见的错误和调试过程。

首先，对于 SSTF 算法，我认识到它的核心思想是优先处理与当前磁头位置最近的请求，从而最小化每一步的寻道时间。通过实验，我验证了 SSTF 的优势在于其能够有效减少磁头的整体移动距离。然而，实验中也让我意识到，该算法可能存在“饥饿”现象：某些远离当前磁头的请求可能长期得不到处理。特别是在实现过程中，我最初遗漏了对已访问请求的标记，导致同一个请求被多次选中，无法正确模拟真实磁头移动顺序。为了修正这一问题，我增加了一个 visited 数组来记录每个请求是否已经被处理，从而确保算法在每一步都能够正确选择未处理且距离最近的请求。

对于 SCAN 算法，我深刻体会到了其类似电梯调度的工作原理。SCAN 算法通过在一个方向上依次处理所有请求，到达边界后再反向移动以处理剩余请求，从而平衡了磁头移动的公平性和效率。实验中，我成功实现了该算法，但在排序请求的过程中出现了逻辑错误。我原本使用了一段简单的冒泡排序代码对请求进行排序，但排序范围未覆盖所有请求，导致部分请求未正确参与调度。为了解决这一问题，我重新检查了排序逻辑，并将所有请求统一排序，确保算法能够按照正确的顺序处理每一个请求。此外，在处理边界位置时，我最初未正确考虑磁头在极端位置的反向移动，导致部分请求被遗漏。通过在方向切换处添加了边界检查，我最终修正了这一问题，保证算法能够完整处理所有请求。

实验中，我还通过详细的输出打印了每一步磁头的移动顺序及对应的移动距离，从而直观地观察两种算法的工作过程和性能差异。通过对比总寻道距离和平均寻道距离的结果，我发现 SSTF 算法在寻道距离上具有一定的优势，但其“饥饿”问题可能导致某些请求的延迟。而 SCAN 算法虽然稍微增加了总寻道距离，但其公平性显著提高，尤其是在处理大量请求时，能够有效避免某些请求长期被忽略。

此次实验让我深刻感受到磁盘调度算法在系统性能优化中的重要性，同时也让我认识到程序设计中细节处理的重要性。在未来的学习中，我将更加注重对算法逻辑的完整性验证，并通过更多的实际问题进一

步加深对算法优化的理解。总的来说，这次实验不仅让我掌握了磁盘调度算法的实现方法，还提高了我在面对问题时的调试能力和解决问题的思维方式。

实验过程中出现的错误及修改过程

在实验过程中，我遇到了一些错误，这些错误虽然影响了算法的正确性，但也让我更加深入地理解了磁盘调度算法的核心逻辑和实现细节。

在实现 **最短寻道时间优先（SSTF）算法** 时，我的第一个问题出现在对已处理请求的标记上。在最初的实现中，我没有正确维护一个数组来记录哪些请求已经被服务过，导致某些请求被重复选择，出现死循环的情况。为了解决这个问题，我引入了一个 visited 数组，将每个请求的处理状态标记为已访问或未访问。在每次选择最近请求时，我只考虑尚未被访问的请求，从而确保算法的正确性。此外，在计算移动距离时，我最初直接更新磁头的位置，而没有记录每次移动的具体距离，导致无法正确统计总寻道距离。我修改了输出逻辑，在每次处理请求后打印移动距离，并在最终结果中输出总寻道距离和平均寻道距离。

对于 **扫描（SCAN）算法**，我遇到的主要问题出现在请求排序和边界处理上。SCAN 算法要求所有请求按照磁道位置进行排序，以便磁头能够按照固定方向依次处理。在初始实现中，我虽然编写了排序

逻辑，但遗漏了对部分请求的正确范围检查，导致一些请求未被包括在排序结果中。

错误结果:

```
========== SSTF ==========
SSTF 调度顺序：90（移动距离：10）
58（移动距离：32）
55（移动距离：3）
39（移动距离：16）
38（移动距离：1）
18（移动距离：20）
150（移动距离：132）
160（移动距离：10）
184（移动距离：24）

总寻道距离：248
平均寻道距离：27.56

========== SCAN ==========
SCAN 调度顺序：
150（移动距离：50）
160（移动距离：10）
184（移动距离：24）
200（移动距离：16）
90（移动距离：110）
58（移动距离：32）
55（移动距离：3）
39（移动距离：16）
38（移动距离：1）
18（移动距离：20）

总寻道距离：282
平均寻道距离：31.33
```

为了解决这一问题，我仔细检查了代码，并确保所有请求都被完整地参与排序。此外，在处理磁头到达边界的反向移动时，我未正确处理磁头与最小请求之间的关系，导致一些靠近边界的请求未被访问。经过调试后，我在代码中增加了边界检查逻辑，确保磁头在到达最大或最小磁道位置后能够正确反向，并处理所有剩余请求。

通过这些错误的排查和修正，我不仅加深了对 SSTF 和 SCAN 算法的理解，也提升了自己调试代码的能力。这些问题让我意识到，在编写算法时，细节处理（如边界条件、数据结构初始化等）的重要性，同时通过输出调试信息也能帮助我快速定位问题并改进代码。最终，所有问题得到解决，实验结果达到了预期的效果。

对实验结果的思考

实验中还有一个较为隐蔽的问题是起始磁头位置和移动方向的选择。我发现，起始位置对算法的输出结果有显著影响，尤其是在 SCAN 算法中。如果起始位置恰好接近请求序列的边界，磁头可能会优先处理较远的请求，导致总寻道距离增加。此外，磁头的初始移动方向同样对结果有重要影响。例如，当请求大部分集中在起始位置的某一侧时，选择错误的移动方向可能会导致磁头先处理较少的请求后再转向，增加了不必要的寻道距离。

为了更好地理解这一点，我在实验中多次调整磁头的初始位置和移动方向，并比较了不同组合下的总寻道距离和平均寻道距离。结果表明，当磁头的初始移动方向与大部分请求所在方向一致时，总寻道距离会显著减少。这一现象进一步验证了磁头初始条件对磁盘调度性能的影响。通过这些实验，我更加深入地认识到，在实际磁盘调度中，合理选择起始位置和方向是优化系统性能的关键因素之一。

不同的磁头位置：

初始位置：50

```
========== SSTF ==========
SSTF 调度顺序：55（移动距离：5）
58（移动距离：3）
39（移动距离：19）
38（移动距离：1）
18（移动距离：20）
90（移动距离：72）
150（移动距离：60）
160（移动距离：10）
184（移动距离：24）

总寻道距离：214
平均寻道距离：23.78

========== SCAN 向上==========
SCAN 调度顺序：
55（移动距离：5）
58（移动距离：3）
90（移动距离：32）
150（移动距离：60）
160（移动距离：10）
184（移动距离：24）
39（移动距离：145）
38（移动距离：1）
18（移动距离：20）

总寻道距离：300
平均寻道距离：33.33

========== SCAN 向下==========
SCAN 调度顺序：
39（移动距离：11）
38（移动距离：1）
18（移动距离：20）
55（移动距离：37）
58（移动距离：3）
90（移动距离：32）
150（移动距离：60）
160（移动距离：10）
184（移动距离：24）

总寻道距离：198
平均寻道距离：22.00
```

初始位置：100

```
========== SSTF ==========
SSTF 调度顺序：90（移动距离：10）
58（移动距离：32）
55（移动距离：3）
39（移动距离：16）
38（移动距离：1）
18（移动距离：20）
150（移动距离：132）
160（移动距离：10）
184（移动距离：24）

总寻道距离：248
平均寻道距离：27.56

========== SCAN 向上==========
SCAN 调度顺序：
150（移动距离：50）
160（移动距离：10）
184（移动距离：24）
90（移动距离：94）
58（移动距离：32）
55（移动距离：3）
39（移动距离：16）
38（移动距离：1）
18（移动距离：20）

总寻道距离：250
平均寻道距离：27.78

========== SCAN 向下==========
SCAN 调度顺序：
90（移动距离：10）
58（移动距离：32）
55（移动距离：3）
39（移动距离：16）
38（移动距离：1）
18（移动距离：20）
150（移动距离：132）
160（移动距离：10）
184（移动距离：24）

总寻道距离：248
平均寻道距离：27.56
```

初始位置：150

```
========== SSTF ==========
SSTF 调度顺序：150（移动距离：0）
160（移动距离：10）
184（移动距离：24）
90（移动距离：94）
58（移动距离：32）
55（移动距离：3）
39（移动距离：16）
38（移动距离：1）
18（移动距离：20）

总寻道距离：200
平均寻道距离：22.22

========== SCAN 向上==========
SCAN 调度顺序：
150（移动距离：0）
160（移动距离：10）
184（移动距离：24）
90（移动距离：94）
58（移动距离：32）
55（移动距离：3）
39（移动距离：16）
38（移动距离：1）
18（移动距离：20）

总寻道距离：200
平均寻道距离：22.22

========== SCAN 向下==========
SCAN 调度顺序：
150（移动距离：0）
90（移动距离：60）
58（移动距离：32）
55（移动距离：3）
39（移动距离：16）
38（移动距离：1）
18（移动距离：20）
160（移动距离：142）
184（移动距离：24）

总寻道距离：298
平均寻道距离：33.11
```