北京交通大学实验报告

课程名称 ： **操作系统**
实验题目 ： **移动头磁盘调度算法模拟实现与比较**
学号 ： **21281280**
姓名 ： **柯劲帆**
班级 ： **物联网2101班**
指导老师 ： **何永忠**
报告日期 ： **2023年12月10日**

目录

---

1. 开发运行环境和工具

操作系统	Linux内核版本	处理器	GCC版本
Deepin 20.9	5.18.17-amd64-desktop-hwe	Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz	gcc (Uos 8.3.0.3-3+rebuild) 8.3.0

- **其他工具**:
  - **编辑**：VSCode
  - **编译和运行**：Terminal

2. 实验过程、分析和结论

2.1. FCFS

```
```

```c
double fcfs(int* requestArray, int requestLength, int headStart, int*
outputArray) {
    int totalMovement = 0;
    int currentPosition = headStart;

    for (int i = 0; i < requestLength; i++) {
        // 计算当前请求和磁头位置之间的距离
        totalMovement += abs(requestArray[i] - currentPosition);
        // 移动磁头到当前请求位置
        currentPosition = requestArray[i];
        // 记录移动过程
        outputArray[i] = currentPosition;
    }

    // 计算平均寻道数
    return (double)totalMovement / requestLength;
}
```

该部分测试输出为：

```
FCFS Algorithm:
Visit Order: 176 32 118 25 69 54 180 129 112 63 176 107 109 74 186 89 31 31
189 177 90 25 179 103 64 110 24 192 121 12 75 144 49 70 138 151 29 168 166 48
54 175 191 50 193 101 121 26 54 11 119 144 96 167 142 1 12 41 102 93 45 153
132 146 151 67 27 121 83 197 173 71 195 64 189 47 31 197 182 157 158 114 33
102 193 197 155 175 188 89 21 27 94 151 73 33 4 110 137 173
Avg Seek Times: 66.65
```

FCFS算法非常简单，易于实现。然而，它并不是最优化的磁盘调度算法，因为它没有考虑磁头移动的最短路径，可能导致较长的平均寻道时间。这可能导致磁盘性能不佳，特别是在请求频繁变化的环境中。

2.2. SSTF

```c
double sstf(int* requestArray, int requestLength, int headStart, int*
outputArray) {
    int totalSeekCount = 0;
    int processedCount = 0;
    int currentPosition = headStart;
    int minDistance, closestIndex;

    // 初始化一个数组来标记已处理的请求
    int processed[REQUEST_LENGTH] = {0};
    for (int i = 0; i < requestLength; i++) processed[i] = 0;

    while (processedCount < requestLength) {
        minDistance = 2147483647;

        // 找到最近的请求
        for (int i = 0; i < requestLength; i++) {
            if (!processed[i] && abs(requestArray[i] - currentPosition) <
minDistance) {
                minDistance = abs(requestArray[i] - currentPosition);
```

```
                closestIndex = i;
            }
        }

        // 处理这个请求
        totalSeekCount += minDistance;
        currentPosition = requestArray[closestIndex];
        processed[closestIndex] = 1;
        outputArray[processedCount++] = currentPosition;
    }

    // 计算平均寻道数
    return (double)totalSeekCount / requestLength;
}

int compare(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}
```

该部分测试输出为：

```
SSTF Algorithm:
Visit Order: 50 49 48 47 45 41 33 33 32 31 31 31 29 27 27 26 25 25 24 21 12 12
11 4 1 54 54 54 63 64 64 67 69 70 71 73 74 75 83 89 89 90 93 94 96 101 102 102
103 107 109 110 110 112 114 118 119 121 121 121 129 132 137 138 142 144 144
146 151 151 151 153 155 157 158 166 167 168 173 173 175 175 176 176 177 179
180 182 186 188 189 189 191 192 193 193 195 197 197 197
Avg Seek Times: 2.45
```

相比于FCFS，SSTF算法在平均寻道时间上有显著提升。这是因为它通过选择最近的请求来减少每次磁头移动的距离，从而降低了总寻道距离。算法中使用 `abs(requestArray[i] - currentPosition)` 计算与当前磁头位置的距离，并选择最近的请求。这种方法更有效地利用了磁头的当前位置。

尽管SSTF在性能上优于FCFS，但它可能导致远离当前磁头位置的请求长时间等待，即“饥饿”问题。这在请求分布不均匀时尤其明显。

2.3. SCAN

```
int compare(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}

double scan(int* requestArray, int requestLength, int headStart, int*
outputArray) {
    int totalSeekCount = 0;
    int processedCount = 0;
    int currentPosition = headStart;

    outputArray = (int*)malloc(sizeof(int) * requestLength);

    // 使用qsort进行排序
    qsort(requestArray, requestLength, sizeof(int), compare);
```

```
    // 找到起始位置最近的请求
    int startIndex;
    for (startIndex = 0; startIndex < requestLength; startIndex++) {
        if (requestArray[startIndex] >= headStart) break;
    }

    // 先向上移动
    for (int i = startIndex; i < requestLength; i++) {
        totalSeekCount += abs(currentPosition - requestArray[i]);
        currentPosition = requestArray[i];
        outputArray[processedCount++] = currentPosition;
    }

    // 然后向下移动
    for (int i = startIndex - 1; i >= 0; i--) {
        totalSeekCount += abs(currentPosition - requestArray[i]);
        currentPosition = requestArray[i];
        outputArray[processedCount++] = currentPosition;
    }

    // 计算平均寻道数
    return (double)totalSeekCount / requestLength;
}
```

该部分测试输出为：

```
SCAN Algorithm:
Visit Order: 50 54 54 54 63 64 64 67 69 70 71 73 74 75 83 89 89 90 93 94 96
101 102 102 103 107 109 110 110 112 114 118 119 121 121 121 129 132 137 138
142 144 144 146 151 151 151 153 155 157 158 166 167 168 173 173 175 175 176
176 177 179 180 182 186 188 189 189 191 192 193 193 195 197 197 197 49 48 47
45 41 33 33 32 31 31 31 29 27 27 26 25 25 24 21 12 12 11 4 1
Avg Seek Times: 3.43
```

SCAN算法通过优化磁头移动的方向来减少寻道距离。它首先向一个方向移动，直到到达最远的请求，然后转向。 相比于SSTF，SCAN算法可以更好地避免饥饿问题。由于磁头最终会访问每个位置，因此所有的请求最终都会被服务。

SCAN算法提供了一种均衡的磁盘调度策略，既减少了平均寻道时间，又避免了长时间等待请求的饥饿问题。实验结果表明，虽然其平均寻道时间略高于SSTF，但在处理大量分散请求时，SCAN算法能提供更加稳定和公平的服务。

2.4. CSCAN

```
double cscan(int* requestArray, int requestLength, int headStart, int*
outputArray) {
    int totalSeekCount = 0;
    int processedCount = 0;
    int currentPosition = headStart;

    outputArray = (int*)malloc(sizeof(int) * requestLength);
```

```
    // 使用 qsort 进行排序
    qsort(requestArray, requestLength, sizeof(int), compare);

    // 找到起始位置最近的请求索引
    int startIndex;
    for (startIndex = 0; startIndex < requestLength; startIndex++) {
        if (requestArray[startIndex] >= headStart) break;
    }

    // 先处理磁头上方的请求
    for (int i = startIndex; i < requestLength; i++) {
        totalSeekCount += abs(currentPosition - requestArray[i]);
        currentPosition = requestArray[i];
        outputArray[processedCount++] = currentPosition;
    }

    // 如果有必要，跳转到最低请求
    if (startIndex != 0) {
        totalSeekCount += abs(currentPosition - requestArray[0]);
        currentPosition = requestArray[0];
        processedCount = 1; // 重置为1，因为重新开始扫描
    }

    // 然后处理磁头下方的请求
    for (int i = 0; i < startIndex; i++) {
        totalSeekCount += abs(currentPosition - requestArray[i]);
        currentPosition = requestArray[i];
        outputArray[processedCount++] = currentPosition;
    }

    // 计算平均寻道数
    return (double)totalSeekCount / requestLength;
}
```

该部分测试输出为：

```
CSCAN Algorithm:
Visit Order: 50 54 54 54 63 64 64 67 69 70 71 73 74 75 83 89 89 90 93 94 96
101 102 102 103 107 109 110 110 112 114 118 119 121 121 121 129 132 137 138
142 144 144 146 151 151 151 153 155 157 158 166 167 168 173 173 175 175 176
176 177 179 180 182 186 188 189 189 191 192 193 193 195 197 197 197 1 4 11 12
12 21 24 25 25 26 27 27 29 31 31 31 32 33 33 41 45 47 48 49
Avg Seek Times: 3.91
```

循环扫描（CSCAN）算法类似于 SCAN 算法，但当磁头到达一个方向的最远端后，它会直接跳转到起始端，而不是改变方向。这种方式使得磁盘的磁头像循环一样移动，避免了SCAN算法中的反向移动。代码先处理磁头上方的请求，然后直接跳到最低的请求继续处理，直至结束。

CSCAN 提供了比 SCAN 更均匀的服务。由于磁头总是在一个方向上移动，所以它可以在相对固定的时间间隔内覆盖整个磁盘。尽管CSCAN可能有时候会导致更长的寻道距离（因为需要跳转到起点），但它保证了所有请求都会在有限时间内得到服务，从而减少了饥饿问题。CSCAN特别适用于请求分布不均匀的情况。它通过固定的方向移动，确保了即使在分布不均的情况下，每个请求最终都会被处理。尽管平均寻道时间可能略高，但它在保证服务公平性和预测性方面的优势使其成为许多场景下的理想选择。

2.5. FSCAN

```c
void processRequests(int* requestArray, int requestLength, int*
currentPosition, int* totalSeekCount, int* outputArray, int* processedCount)
{
    // 使用 qsort 进行排序
    qsort(requestArray, requestLength, sizeof(int), compare);

    // 处理所有请求
    for (int i = 0; i < requestLength; i++) {
        *totalSeekCount += abs(*currentPosition - requestArray[i]);
        *currentPosition = requestArray[i];
        outputArray[(*processedCount)++] = requestArray[i];
    }
}

double fscan(int* requestArray1, int requestLength1, int* requestArray2, int
requestLength2, int headStart, int *outputArray) {
    int totalSeekCount = 0;
    int currentPosition = headStart;
    int processedCount = 0;

    outputArray = (int*)malloc(sizeof(int) * (requestLength1 +
requestLength2));

    // 处理第一个队列
    processRequests(requestArray1, requestLength1, &currentPosition,
&totalSeekCount, outputArray, &processedCount);

    // 处理第二个队列
    processRequests(requestArray2, requestLength2, &currentPosition,
&totalSeekCount, outputArray, &processedCount);

    // 计算平均寻道数
    return (double)totalSeekCount / (requestLength1 + requestLength2);
}
```

该部分测试输出为：

```
FSCAN Algorithm:
Visit Order: 50 54 54 54 63 64 69 70 74 75 89 90 101 103 107 109 110 112 118
121 121 129 138 144 151 166 168 175 176 176 177 179 180 186 189 191 192 193 49
48 32 31 31 29 26 25 25 24 12 11 158 166 167 168 173 173 175 175 176 176 177
179 180 182 186 188 189 189 191 192 193 193 195 197 197 197 1 4 11 12 12 21 24
25 25 26 27 27 29 31 31 31 32 33 33 41 45 47 48 49
Avg Seek Times: 7.07
```

FSCAN通过将请求分成两个队列来减少磁头移动次数。这样，它可以在处理当前队列时不受新请求的干扰。由于FSCAN总是处理完整个队列，它可以有效预防饥饿问题。每个请求最终都会被处理。分开处理两个队列可以提高处理效率。在处理一个队列时，磁头不需要反复移动以应对新进的请求。

测试输出显示平均寻道时间为7.07。这个数值相比其他算法较高，可能是因为在处理一个队列时，磁头需要移动到另一个队列的起始位置，导致额外的寻道时间。

FSCAN特别适合于请求量大且需要公平处理的场景。它通过分开处理不同的请求队列，确保了长期运行下的稳定性和公平性。

2.5. MAIN

```
void processQueue(int* queue, int length, int* currentPosition, int* totalSeekCount, int* outputArray, int* processedCount) {
    // 对队列进行排序
    qsort(queue, length, sizeof(int), compare);

    // 找到最接近当前磁头位置的请求
    int i = 0;
    while (i < length && queue[i] < *currentPosition) {
        i++;
    }

    // 先处理磁头位置之后（更高磁道号）的请求
    for (int j = i; j < length; j++) {
        *totalSeekCount += abs(*currentPosition - queue[j]);
        *currentPosition = queue[j];
        outputArray[(*processedCount)++] = queue[j];
    }

    // 再处理磁头位置之前（更低磁道号）的请求
    for (int j = i - 1; j >= 0; j--) {
        *totalSeekCount += abs(*currentPosition - queue[j]);
        *currentPosition = queue[j];
        outputArray[(*processedCount)++] = queue[j];
    }
}

double fscan(int* requestArray1, int requestLength1, int* requestArray2, int requestLength2, int headStart, int* outputArray) {
    int totalSeekCount = 0;
    int currentPosition = headStart;
    int processedCount = 0;

    // 处理第一个队列
    processQueue(requestArray1, requestLength1, &currentPosition, &totalSeekCount, outputArray, &processedCount);

    // 处理第二个队列
    processQueue(requestArray2, requestLength2, &currentPosition, &totalSeekCount, outputArray + requestLength1, &processedCount);

    // 计算平均寻道数
    return (double)totalSeekCount / (requestLength1 + requestLength2);
}
```

结论

算法	平均寻道数1	平均寻道数2
FCFS	66.65	60.75
SSTF	2.45	2.49
SCAN	3.43	3.48
CSCAN	3.91	3.97
FSCAN	7.07	7.34

从实验数据中，我们可以看到不同磁盘调度算法在平均寻道数方面的表现差异。以下是对各算法性能的总结和分析：

FCFS (先来先服务)

- **平均寻道数:** 66.65 和 60.75
- **分析:** FCFS算法因其简单性在平均寻道数上表现最差。由于它仅基于请求到达的顺序，没有考虑磁头移动的最优化，因此导致了较高的寻道数。

SSTF (最短寻道时间优先)

- **平均寻道数:** 2.45 和 2.49
- **分析:** SSTF算法大幅改进了寻道性能，因为它总是选择距离当前磁头位置最近的请求。这种策略显著减少了磁头的移动距离，从而降低了平均寻道数。

SCAN (扫描)

- **平均寻道数:** 3.43 和 3.48
- **分析:** SCAN算法的表现优于FCFS，略逊于SSTF。它通过模仿电梯运行，从一端移动到另一端，避免了频繁的方向改变，从而实现了较低的寻道数。

CSCAN (循环扫描)

- **平均寻道数:** 3.91 和 3.97
- **分析:** CSCAN算法提供了一个循环的扫描机制，避免了SCAN中的反向移动，但其平均寻道数略高于SCAN。这可能是因为在达到一端后需要跳转到另一端，导致了额外的寻道时间。

FSCAN (FIFO扫描)

- **平均寻道数:** 7.07 和 7.34
- **分析:** FSCAN算法在所有测试算法中平均寻道数次高。虽然它通过使用两个队列来优化长期的公平性和稳定性，但在单次运行中，由于需要在两个队列间切换，可能导致较高的寻道数。

总体观察

- 在这些算法中，**SSTF** 和 **SCAN** 在平均寻道数方面表现最佳，提供了较好的性能与效率平衡。
- **FCFS** 由于其简单性，在寻道数方面表现最差。
- **CSCAN** 和 **FSCAN** 虽然在某些场景下可能更合适，但在这组数据中它们的平均寻道数相对较高。

这些结果显示了在不同的磁盘调度策略下，性能和效率之间的权衡。选择最合适的算法取决于具体的应用场景和对性能、效率以及公平性的不同需求。