实验 19：移动头磁盘调度算法模拟实现与比较

计科 1903 班 19281171 王雨潇

1. 实验目的

利用标准 C 语言编程设计实现主要的移动头磁盘调度算法，随机发生磁盘访问事件序列测试比较有关算法的性能。

2. 实验环境

运行环境：操作系统 Windows 10 家庭版

编译器：MinGW-w64 GCC 11.2.0

IDE：VSCode

实验代码见“附录：源程序的完整代码”，所有流程图均使用 plantuml 绘制

3. 实验内容

实验用到的宏定义如下，主要内容为随机生成场景的配置和调度算法用到的标记：

```c
/* 随机发生场景设定 */
#define START_POS 123   // 移动头初始位置
int pos = START_POS;    // 移动头
int trackNumbers[100];  // 磁盘访问事件序列

/* 电梯调度算法中移动头的方向 */
#define outside 1           // 向外移动
#define inside  -1          // 向内移动

/* 其他定义 */
#define author  19281171    // 署个名
#define bool    int         // 假装有 bool 类型
#define true    1
#define false   0
```

3.1. 关键数据结构定义

采用数组 trackNumbers 描述磁盘访问事件序列和执行过程，数组下标从低到高即事件的顺序，数组内容即待访问的磁道号；

3.2. 磁盘调度算法设计

3.2.1. 先来先服务调度算法

最简单的一种磁盘调度算法，其主要流程为：从下标为 0 开始依次遍历 100 个硬盘访问事件，对每个事件，移动头移动到相应的磁道号，计算移动距离，统计平均寻道数；

显示输出磁道访问过程信息的代码如下所示，其他算法的显示也基本类似：

```
// 先来先服务调度算法（FCFS）
void FCFS() {
    int totalSteps = 0, step = 0;
    pos = START_POS;
    printf("\n---------------------------------FCFS---------------------------------
\n");
    printf("Access Track Number\tMovement Distance\tHead Position\n");
    for (int i=0; i<100; i++) {
        // 计算移动距离，加到总移动距离上，调整移动头
        step = (trackNumbers[i] > pos)? (trackNumbers[i] - pos): (pos -
trackNumbers[i]);
        totalSteps += step;
        pos = trackNumbers[i];
        printf("\t%d\t\t\t%d\t\t\t%d\n", trackNumbers[i], step, pos);
    }
    printf("Average seek number = %lf\n", (double)totalSteps / 100.0);
}
```

3.2.2. 最短寻道时间优先调度算法

最短寻道时间优先调度算法每次先访问的磁道与磁头当前所在磁道距离最近的进程，为实现这一算法，需要按待访问磁道与磁头当前位置的远近排序事件队列；C 语言中标准库函数 std::qsort 提供了排序功能，还需要实现形如 int cmp(const void *a, const void *b)的比较函数。

此处按磁道号和移动头之差的绝对值升序排列：

```
// SSTF 算法的 qsort 用的比较函数
int cmp1(const void *a, const void *b) {
    const int *pos1 = (const int *) a;
    const int *pos2 = (const int *) b;

    if(*pos1 == *pos2) return 0;
    if(abs(pos - *pos1) < abs(pos - *pos2)) return -1;
    return 1;
}
```

算法主要流程为：遍历 100 个硬盘访问事件，每次先调用

```
qsort(trackNumbersCopy + i, 100 - i, sizeof(int), cmp1)
```

排序尚未访问的进程，再执行磁道号和移动头之差最小的事件，移动头移动到相应的磁道号，计算移动距离，统计平均寻道数；

3.2.3. 电梯调度算法

电梯调度算法又名扫描算法，其主要流程为：

(1) 当磁头向外移动时，遍历寻找向外移动路径上寻道数最短的磁道号，访问；

(2) 当磁头向内移动时，遍历寻找向内移动路径上寻道数最短的磁道号，访问；

(3) 如果该路径上没有磁道访问请求，则改变磁头移动方向；

该算法的关键主循环实现如下：

```
for (int i=0; i<100; i++) {
    int k = i;
    for (int j=i+1; j<100; j++) {
        if (direction == outside) {
            // 优先访问更外面的且寻道最短的磁道
            if (pos < trackNumbersCopy[j] && trackNumbersCopy[j] < trackNumbersCopy[k]) {
                // 情况 1: j 比当前的合法 k 寻道更短 pos < j < k
                k = j;
            } else if (trackNumbersCopy[k] < pos && pos < trackNumbersCopy[j]) {
                // 情况 2: 当前 k 不合法, k < pos < j
                k = j;
            }
        } else if (direction == inside) {
            // 优先访问更里面的且寻道最短的磁道
            if (trackNumbersCopy[k] < trackNumbersCopy[j] && trackNumbersCopy[j] < pos) {
```

```
                // 情况 1：j 比当前的合法 k 寻道更短  k < j < pos
                k = j;
            } else if (trackNumbersCopy[j] < pos && pos < trackNumbersCopy[k]) {
                // 情况 2：当前 k 不合法，j < pos < k
                k = j;
            }
        }
    }
    // 交换使 trackNumbersCopy[i] = 电梯调度算法选出的磁道号 ...
    // 判断是否要转向，新旧 direction 不同即没有找到该路径上的访问事件 ...
    // 计算移动距离，加到总移动距离上，调整移动头 ...
}
```

3.2.4. 循环式单向电梯调度算法

循环式单向电梯调度算法是电梯调度算法的改进，其在电梯调度的基础上，规定磁头单向向外扫描，直至扫描到最外请求磁道，扫描结束立刻返回最内请求的磁道，再开始第二次扫描。

代码实现循环单向电梯算法的主要流程为（代码详见 附录：源程序的完整代码）：

（1） 按磁道号升序排列事件序列；

（2） 通过二分查找算法，找到磁道号在移动头外侧又最接近移动头的事件；

（3） 向外扫描，执行自该事件起始的所有事件；

（4） 移动到事件序列下标为 0 处，向外扫描，执行剩余的访问事件；

3.2.5. 双队列单向电梯调度算法

双队列单向电梯调度算法是电梯调度算法的改进，其在电梯调度的基础上，把访问磁盘分为两个子队列，流程如下（代码详见 附录：源程序的完整代码）：

（1） 先将磁道号序列的后一半请求放入等待队列，

（2） 磁道号序列的前一半作为当前队列，向外扫描逐个访问；

（3） 再把等待队列变为当前队列，向外扫描第二遍访问磁盘；

3.3. 随机生产磁道号序列

根据实验要求，生成 100 个范围在[0,199]的随机数，作为磁道号访问序列，数组下标从低到高即事件的顺序，数组内容即待访问的磁道号；

```
// 生成一组移动头磁盘访问事件序列共 100 个磁道号
void generateTrackNumbers() {
    for (int i=0; i<100; i++) {
        trackNumbers[i] = rand() % 200; // 磁道号取值区间为[0, 199]
    }
}
```

3.4. 算法性能统计分析

分几组情况讨论：

（1） 移动头起始位置为 0

（2） 移动头起始位置为中间数（选取的 3 组为 36，100，123）

（3） 移动头起始位置为 199

实验原始数据如下：

起始头位置为 0		**平均寻道数**
	先来先服务	61.44
	最短寻道时间优先调度算法	1.95
	电梯调度算法	5.35
	循环式单向电梯调度算法	1.95
	双队列单向电梯调度	5.8
起始头位置为 0		平均寻道数
	先来先服务	70.02
	最短寻道时间优先调度算法	1.96
	电梯调度算法	5.85
	循环式单向电梯调度算法	1.96
	双队列单向电梯调度	5.82
起始头位置为 36		平均寻道数
	先来先服务	68.95
	最短寻道时间优先调度算法	2.36

	电梯调度算法	7.28
	循环式单向电梯调度算法	3.96
	双队列单向电梯调度	6.12
起始头位置为 100		平均寻道数
	先来先服务	60.9
	最短寻道时间优先调度算法	3
	电梯调度算法	6.77
	循环式单向电梯调度算法	3.95
	双队列单向电梯调度	6.73
起始头位置为 123		平均寻道数
	先来先服务	58.76
	最短寻道时间优先调度算法	3.29
	电梯调度算法	6.38
	循环式单向电梯调度算法	3.92
	双队列单向电梯调度	7.05
起始头位置为 199		平均寻道数
	先来先服务	64.52
	最短寻道时间优先调度算法	1.91
	电梯调度算法	5.17
	循环式单向电梯调度算法	3.79
	双队列单向电梯调度	7.5

根据上表和下图可以发现：

(1) 最简单的先来先服务算法，按时序依次访问磁道号，虽然实现简单，但该算法的平均寻道数远大于其他 4 种算法；

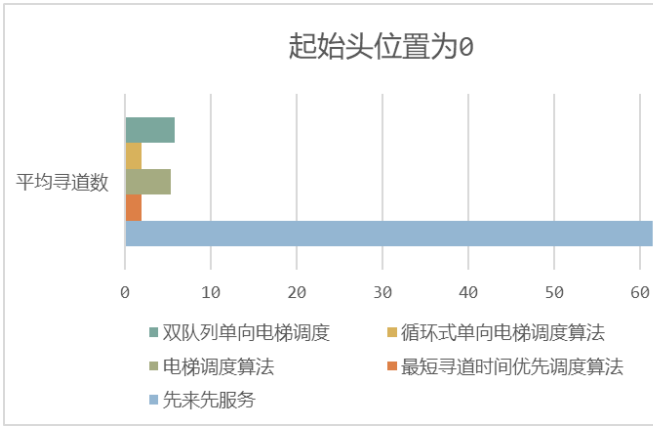
(2) 最短寻道时间算法是本次实验的算法里平均寻道数最低的算法，该算法采用局部贪心的设计思想，先访问距离移动头最近的磁道号，实验证明这种设计思路的性能表现确实是优异的，但现实场景中，该算法会造成远处的磁道号长期得不到服务的问题；

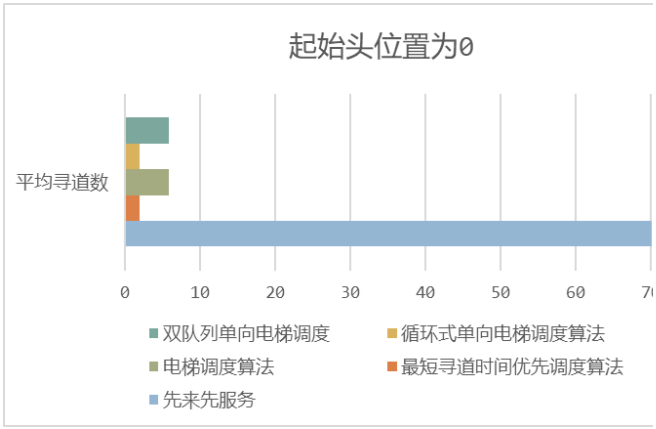
(3) 电梯调度算法是本次实验实现最复杂的算法，需要考虑怎么寻找扫描路径上最近的磁道号，也要考虑反向扫描的条件，与现实生活中的电梯很相似；

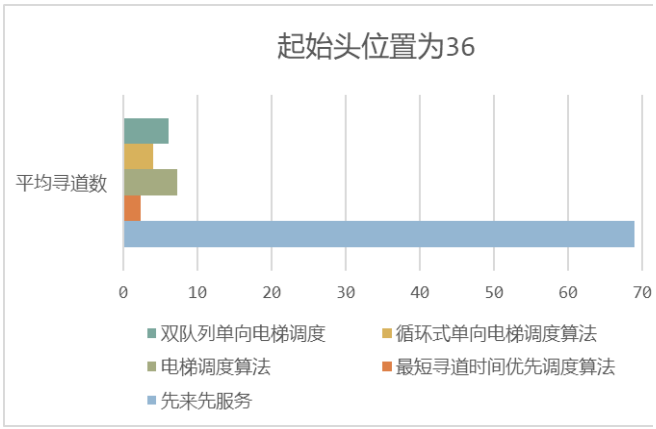
(4) 循环式单向电梯调度算法是用多遍单向向外扫描的方式优化了电梯算法，可以从实验数据发现其平均寻道数低于原版的电梯调度算法；

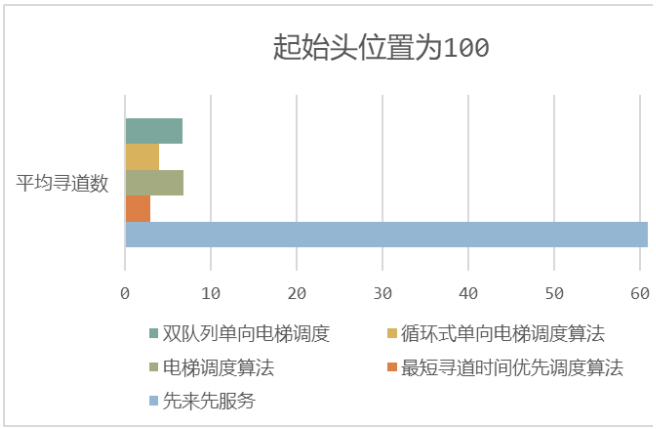
（5） 双队列单向电梯调度算法采取两个队列进行单向的电梯调度，在实验程序中该算法的平均寻道数略高于原版的电梯调度算法，可能是因为本实验的事件序列并非“按时”出现，不够接近真实生活场景；

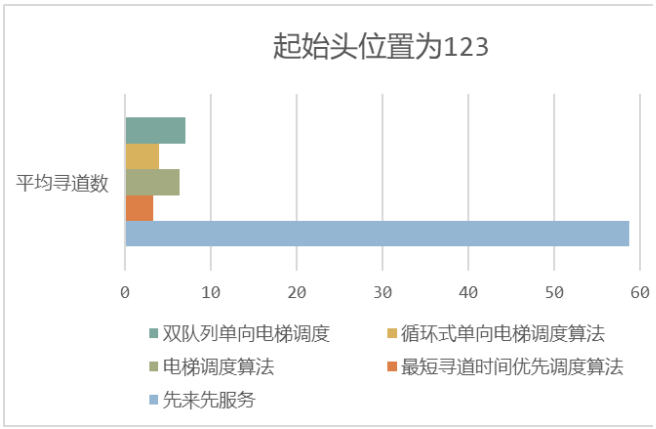
条形统计图如下：

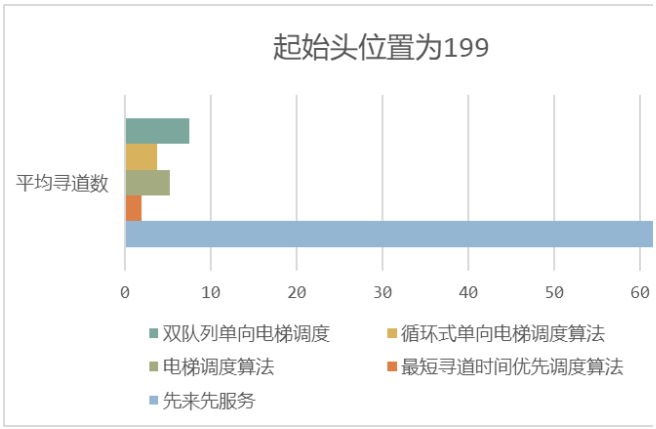












3.5. 运行结果截图展示

```
> Executing task: xmake r os_lab19  <


--------------------------------FCFS--------------------------------
Access Track Number     Movement Distance       Head Position
        146                     23                      146
        78                      68                      78
        46                      32                      46
        134                     88                      134
        61                      73                      61
        197                     136                     197
        79                      118                     79
        97                      18                      97
        140                     43                      140
        124                     16                      124
        152                     28                      152
        72                      80                      72
        42                      30                      42
        85                      43                      85
        126                     41                      126
        125                     1
        30                      95
        81                      51
        116                     35
        125                     9
```


```
Average seek number = 3.900000

--------------------------------FSCAN--------------------------------
Access Track Number     Movement Distance       Head Position
        2                       121                     2
        3                       1                       3
        21                      18                      21
        24                      3                       24
        30                      6                       30
        31                      1                       31
        35                      4                       35
        42                      7                       42
        46                      4                       46
        49                      3                       49
        51                      2                       51
        52                      1
        52                      0
        54                      2
        56                      2
        61                      5
```

4. 实验结论和心得体会

本次实验我通过模拟实现 5 种 常见的移动头磁盘调度算法，了解了磁盘调度机制，通过对比各算法的性能统计数据，对不同的磁盘调度算法性能也有了更深入的理解。实验中我发现电梯调度算法的设计非常巧妙，Linux-0.11 内核源码也采用了电梯算法调度磁盘，我们应当熟练掌握这一算法。

5. 附录：源程序的完整代码

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

/* 随机发生场景设定 */
#define START_POS 123    // 移动头初始位置
int pos = START_POS;     // 移动头
int trackNumbers[100];   // 磁盘访问事件序列

/* 电梯调度算法中移动头的方向 */
#define outside 1            // 向外移动
#define inside  -1           // 向内移动

/* 其他定义 */
#define author  19281171     // 署个名
#define bool    int          // 假装有 bool 类型
#define true    1
#define false   0

// 生成一组移动头磁盘访问事件序列共 100 个磁道号
void generateTrackNumbers() {
    for (int i=0; i<100; i++) {
        trackNumbers[i] = rand() % 200; // 磁道号取值区间为[0, 199]
    }
}

// 先来先服务调度算法（FCFS）
void FCFS() {
    int totalSteps = 0, step = 0;
    pos = START_POS;
    printf("\n--------------------------------FCFS--------------------------------
\n");
    printf("Access Track Number\tMovement Distance\tHead Position\n");
    for (int i=0; i<100; i++) {
        // 计算移动距离，加到总移动距离上，调整移动头
        step = (trackNumbers[i] > pos)? (trackNumbers[i] - pos): (pos -
trackNumbers[i]);
        totalSteps += step;
        pos = trackNumbers[i];
        printf("\t%d\t\t\t%d\t\t\t%d\n", trackNumbers[i], step, pos);
    }
```

```
    printf("Average seek number = %lf\n", (double)totalSteps / 100.0);
}

// SSTF 算法的 qsort 用的比较函数
int cmp1(const void *a, const void *b) {
    const int *pos1 = (const int *) a;
    const int *pos2 = (const int *) b;

    if(*pos1 == *pos2) return 0;
    if(abs(pos - *pos1) < abs(pos - *pos2)) return -1;
    return 1;
}

// 最短寻道时间优先调度算法（SSTF）
void SSTF() {
    int totalSteps = 0, step = 0;
    pos = START_POS;
    // 因为会破坏原访问次序，所以备份一份
    int trackNumbersCopy[100];
    for (int i=0; i<100; i++) {
        trackNumbersCopy[i] = trackNumbers[i];
    }
    printf("\n--------------------------------SSTF--------------------------------
\n");
    printf("Access Track Number\tMovement Distance\tHead Position\n");

    for (int i=0; i<100; i++) {
        // 按磁道号离移动头的距离升序排序
        qsort(trackNumbersCopy + i, 100 - i, sizeof(int), cmp1);
        // 计算移动距离，加到总移动距离上，调整移动头
        step = (trackNumbersCopy[i] > pos)? (trackNumbersCopy[i] - pos): (pos -
trackNumbersCopy[i]);
        totalSteps += step;
        pos = trackNumbersCopy[i];
        printf("\t%d\t\t\t%d\t\t\t%d\n", trackNumbersCopy[i], step, pos);
    }
    printf("Average seek number = %lf\n", (double)totalSteps / 100.0);
}

// 电梯调度算法（SCAN）
void SCAN() {
    int totalSteps = 0, step = 0;
    pos = START_POS;
    // 因为会破坏原访问次序，所以备份一份
    int trackNumbersCopy[100];
    for (int i=0; i<100; i++) {
        trackNumbersCopy[i] = trackNumbers[i];
    }
    printf("\n--------------------------------SCAN--------------------------------
\n");
    printf("Access Track Number\tMovement Distance\tHead Position\n");

    // 根据初始磁带头位置决定默认方向
    int direction = (START_POS >= 100)? inside: outside;

    for (int i=0; i<100; i++) {
        int k = i;
        for (int j=i+1; j<100; j++) {
```

```
            if (direction == outside) {
                // 优先访问更外面的且寻道最短的磁道
                if (pos < trackNumbersCopy[j] && trackNumbersCopy[j] <
trackNumbersCopy[k]) {
                    // 情况 1：j 比当前的合法 k 寻道更短 pos < j < k
                    k = j;
                } else if (trackNumbersCopy[k] < pos && pos < trackNumbersCopy[j])
{
                    // 情况 2：当前 k 不合法，k < pos < j
                    k = j;
                }
            } else if (direction == inside) {
                // 优先访问更里面的且寻道最短的磁道
                if (trackNumbersCopy[k] < trackNumbersCopy[j] &&
trackNumbersCopy[j] < pos) {
                    // 情况 1：j 比当前的合法 k 寻道更短  k < j < pos
                    k = j;
                } else if (trackNumbersCopy[j] < pos && pos < trackNumbersCopy[k])
{
                    // 情况 2：当前 k 不合法，j < pos < k
                    k = j;
                }
            }
        }

        // 交换使 trackNumbersCopy[i] = 电梯调度算法选出的磁道号
        int temp = trackNumbersCopy[k];
        trackNumbersCopy[k] = trackNumbersCopy[i];
        trackNumbersCopy[i] = temp;

        // 判断是否要转向
        int newDirection = (trackNumbersCopy[i] > pos)? outside: inside;
        if (newDirection != direction) {
            direction = newDirection;
            i--;
            continue;
        }

        // 计算移动距离，加到总移动距离上，调整移动头
        step = (trackNumbersCopy[i] > pos)? (trackNumbersCopy[i] - pos): (pos -
trackNumbersCopy[i]);
        totalSteps += step;
        pos = trackNumbersCopy[i];
        printf("\t%d\t\t\t%d\t\t\t%d\n", trackNumbersCopy[i], step, pos);
    }
    printf("Average seek number = %lf\n", (double)totalSteps / 100.0);
}

// CSCAN 和 FSCAN 算法的 qsort 用的比较函数，磁道号升序排序
int cmp2(const void *a, const void *b) {
    return *(int *)a - *(int *)b;
}

// 循环式单向电梯调度算法（CSCAN）
void CSCAN() {
    int totalSteps = 0, step = 0;
    pos = START_POS;
    // 因为会破坏原访问次序，所以备份一份
```

```
    int trackNumbersCopy[100];
    for (int i=0; i<100; i++) {
        trackNumbersCopy[i] = trackNumbers[i];
    }
    printf("\n--------------------------------CSCAN-------------------------------
-\n");
    printf("Access Track Number\tMovement Distance\tHead Position\n");

    // 规定磁头单向向外扫描，当扫描到最外请求磁道后，立刻返回最里面请求的磁道
    int direction = outside;
    // 按磁道号升序排序
    qsort(trackNumbersCopy, 100, sizeof(int), cmp2);

    // 二分法找到起始点
    int l=0, r=99, mid;
    while (l <= r) {
        mid = (l+r)/2;
        int val = trackNumbersCopy[mid];
        if (val < pos) {
            l = mid + 1;
        } else if (pos < val) {
            r = mid - 1;
        } else if (val == pos) {
            break;
        }
    }

    // 磁头单向向外扫描
    for (int i=mid; i<100; i++) {
        // 计算移动距离，加到总移动距离上，调整移动头
        step = (trackNumbersCopy[i] > pos)? (trackNumbersCopy[i] - pos): (pos -
trackNumbersCopy[i]);
        totalSteps += step;
        pos = trackNumbersCopy[i];
        printf("\t%d\t\t\t%d\t\t\t%d\n", trackNumbersCopy[i], step, pos);
    }

    // 扫描到最外请求磁道后，立刻返回最里面请求的磁道
    for (int i=0; i<mid; i++) {
        // 计算移动距离，加到总移动距离上，调整移动头
        step = (trackNumbersCopy[i] > pos)? (trackNumbersCopy[i] - pos): (pos -
trackNumbersCopy[i]);
        totalSteps += step;
        pos = trackNumbersCopy[i];
        printf("\t%d\t\t\t%d\t\t\t%d\n", trackNumbersCopy[i], step, pos);
    }
    printf("Average seek number = %lf\n", (double)totalSteps / 100.0);
}

// 双队列电梯调度算法（FSCAN）
void FSCAN() {
    int totalSteps = 0, step = 0;
    pos = START_POS;
    // 因为会破坏原访问次序，所以备份一份
    int trackNumbersCopy[100];
    for (int i=0; i<100; i++) {
        trackNumbersCopy[i] = trackNumbers[i];
    }
```

```
    printf("\n--------------------------------FSCAN-------------------------------
-\n");
    printf("Access Track Number\tMovement Distance\tHead Position\n");

    // 模拟：假设前一半请求是当前队列，后一半请求是在扫描过程中新出现的服务序列
    // 对两个队列分别按磁道号升序排序
    qsort(trackNumbersCopy, 50, sizeof(int), cmp2);
    qsort(trackNumbersCopy + 50, 50, sizeof(int), cmp2);

    // 磁头单向向外扫描
    for (int i=0; i<50; i++) {
        // 计算移动距离，加到总移动距离上，调整移动头
        step = (trackNumbersCopy[i] > pos)? (trackNumbersCopy[i] - pos): (pos -
trackNumbersCopy[i]);
        totalSteps += step;
        pos = trackNumbersCopy[i];
        printf("\t%d\t\t\t%d\t\t\t%d\n", trackNumbersCopy[i], step, pos);
    }

    // 第二遍等待队列变为当前队列
    for (int i=50; i<100; i++) {
        // 计算移动距离，加到总移动距离上，调整移动头
        step = (trackNumbersCopy[i] > pos)? (trackNumbersCopy[i] - pos): (pos -
trackNumbersCopy[i]);
        totalSteps += step;
        pos = trackNumbersCopy[i];
        printf("\t%d\t\t\t%d\t\t\t%d\n", trackNumbersCopy[i], step, pos);
    }
    printf("Average seek number = %lf\n", (double)totalSteps / 100.0);
}


int main(int argc, char** argv) {
    srand((unsigned)time(NULL));    // 初始化随机数种子

    generateTrackNumbers();
    FCFS();
    SSTF();
    SCAN();
    CSCAN();
    FSCAN();

    return 0;
}
```