**广州大学学生实验报告**

**开课学院及实验室：**计算机科学与网络工程学院软件实验室 **2023年 5月 日**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学院** | **计算机科学与网络工程学院** | **年级/专业/班** | **软件211** | **姓名** | 张景致 | **学号** | 32106300004 |
| **实验课程名称** | 操作系统实验 | | | | | **成绩** |  |
| **实验项目名称** | 磁盘管理实验 | | | | | **指导老师** |  |

**实验五 磁盘管理实验**

1. **实验目的**
2. 了解磁盘调度的策略和原理；
3. 理解和掌握磁盘调度算法——先来先服务算法（FCFS）、最短寻道时间优先算法（SSTF）、电梯扫描算法（SCAN）。
4. **实验内容**
5. 模拟先来先服务法（First-Come, First-Served，FCFS），最短寻道时间优先法（Shortest Seek Time First， SSTF），电梯扫描算法（SCAN）三种磁盘调度算法；
6. 对三种算法进行对比分析。
7. 输入为一组请求访问磁道序列，输出为每种调度算法的磁头移动轨迹和移动的总磁道数。
8. **实验原理**

1、先来先服务算法（FCFS）:  按先来后到次序服务，未作优化。最简单的移臂调度算法是“先来先服务”调度算法，这个算法实际上不考虑访问者要求访问的物理位置，而只是考虑访问者提出访问请求的先后次序。 采用先来先服务算法决定等待访问者执行输入输出操作的次序时，移动臂来回地移动。先来先服务算法花费的寻找时间较长，所以执行输入输出操作的总时间也很长。

2、最短寻道时间优先算法（SSTF） :  最短寻找时间优先调度算法总是从等待访问者中挑选寻找时间最短的那个请求先执行的，而不管访问者到来的先后次序。与先来先服务、算法比较，大幅度地减少了寻找时间，因而缩短了为各访问者请求服务的平均时间，也就提高了系统效率。但最短查找时间优先（SSTF）调度，FCFS会引起读写头在盘面上的大范围移动，SSTF查找距离磁头最短（也就是查找时间最短）的请求作为下一次服务的对象。SSTF查找模式有高度局部化的倾向，会推迟一些请求的服务，甚至引起无限拖延（又称饥饿）。

3、扫描算法（SCAN）：  SCAN 算法又称电梯调度算法。SCAN算法是磁头前进方向上的最短查找时间优先算法，它排除了磁头在盘面局部位置上的往复移动，SCAN算法在很大程度上消除了SSTF算法的不公平性，但仍有利于对中间磁道的请求。“电梯调度”算法是从移动臂当前位置开始沿着臂的移动方向去选择离当前移动臂最近的那个柱访问者，如果沿臂的移动方向无请求访问时，就改变臂的移动方向再选择。但是，“电梯调度”算法在实现时，不仅要记住读写磁头的当前位置，还必须记住移动臂的当前前进方向。

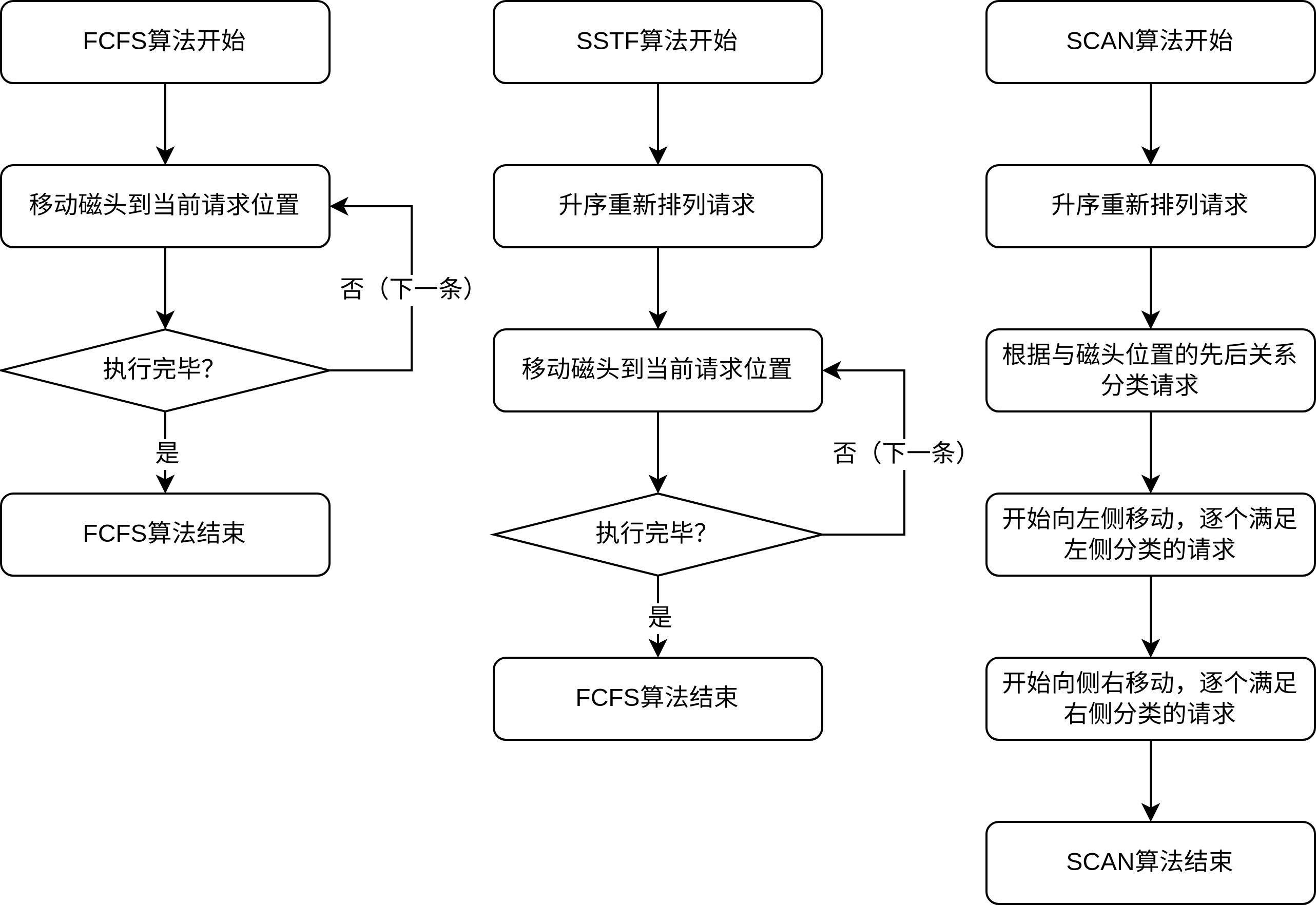
1. **实验中用到的系统调用函数**

实验只是模拟实现磁盘调度功能，不需要系统调用函数。

1. **实验要求**
2. 输入为一组请求访问磁道序列，该序列和所选磁道个数要求随机生成，输出为每种调度算法的磁头移动轨迹和移动的总磁道数；

2、输入磁道范围 0~1000 ，输入所选磁道个数0~1000；

3、画出主程序流程图；

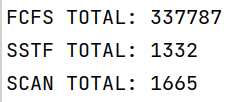


1. 编写程序并调试；

main.c

#include <stdio.h>  
#include <stdlib.h>  
#include <stdbool.h>  
#include <time.h>  
  
#define **TRACK\_COUNT** 1000  
#define **REQUEST\_COUNT** 1000  
  
#define **IS\_VERBOSE false**#define **ON\_VERBOSE** if (**IS\_VERBOSE**)  
  
int prophecy[**REQUEST\_COUNT**];  
int sequence[**REQUEST\_COUNT**];  
int init\_track = 0;  
int cur\_track = 0;  
  
void populate\_prophecy() {  
 srand48(time(**NULL**));  
 for (int i = 0; i < **REQUEST\_COUNT**; i++) {  
 double rnd = drand48();  
 prophecy[i] = (int) (rnd \* **TRACK\_COUNT**);  
 }  
 double rnd = drand48();  
 init\_track = (int) (rnd \* **TRACK\_COUNT**);  
}  
  
void populate\_sequence() {  
 for (int i = 0; i < **REQUEST\_COUNT**; i++) {  
 sequence[i] = prophecy[i];  
 }  
 cur\_track = init\_track;  
}  
  
#define **ELEM\_PER\_LINE** 10  
  
void print\_prophecy() {  
 **ON\_VERBOSE** printf("[Prophecy]\n");  
 int cur\_elem = 0;  
 for (int i = 0; i < **REQUEST\_COUNT**; i++) {  
 **ON\_VERBOSE** printf("%d];", sequence[i]);  
 cur\_elem = (cur\_elem + 1) % **ELEM\_PER\_LINE**;  
 if (cur\_elem == 0) {  
 **ON\_VERBOSE** printf("\n");  
 }  
 }  
}  
  
void fcfs() {  
 **ON\_VERBOSE** printf("[FCFS]\nTRACK];DELTA];TOTAL\n");  
 int total\_movement = 0;  
 for (int i = 0; i < **REQUEST\_COUNT**; i++) {  
 int track = sequence[i];  
 int movement = abs(track - cur\_track);  
 total\_movement += movement;  
 cur\_track = track;  
 **ON\_VERBOSE** printf("%d];%d];%d\n", track, movement, total\_movement);  
 }  
 printf("FCFS TOTAL: %d\n", total\_movement);  
}  
  
int compare\_req(const void \*a, const void \*b) {  
 int req\_a = \*(int \*) a;  
 int req\_b = \*(int \*) b;  
 return req\_a - req\_b;  
}  
  
void sstf() {  
 qsort(sequence, **REQUEST\_COUNT**, sizeof(int), compare\_req);  
 **ON\_VERBOSE** printf("[SSTF]\nTRACK];DELTA];TOTAL\n");  
 int total\_movement = 0;  
 for (int i = 0; i < **REQUEST\_COUNT**; i++) {  
 int track = sequence[i];  
 int movement = abs(track - cur\_track);  
 total\_movement += movement;  
 cur\_track = track;  
 **ON\_VERBOSE** printf("%d];%d];%d\n", track, movement, total\_movement);  
 }  
 printf("SSTF TOTAL: %d\n", total\_movement);  
}  
  
int left\_array[**REQUEST\_COUNT**];  
int left\_count = 0;  
int right\_array[**REQUEST\_COUNT**];  
int right\_count = 0;  
  
void scan() {  
 qsort(sequence, **REQUEST\_COUNT**, sizeof(int), compare\_req);  
 for (int i = 0; i < **REQUEST\_COUNT**; i++) {  
 int req = sequence[i];  
 if (req < cur\_track) {  
 left\_array[left\_count++] = req;  
 } else {  
 right\_array[right\_count++] = req;  
 }  
 }  
 **ON\_VERBOSE** printf("[SCAN]\nTRACK];DELTA];TOTAL\n");  
 int total\_movement = 0;  
 **bool** forward = **true**;  
 while (left\_count > 0 || right\_count > 0) {  
 int track;  
 if (forward) {  
 if (right\_count > 0) {  
 track = right\_array[--right\_count];  
 } else {  
 forward = **false**;  
 track = left\_array[--left\_count];  
 }  
 } else {  
 if (left\_count > 0) {  
 track = left\_array[--left\_count];  
 } else {  
 forward = **true**;  
 track = right\_array[--right\_count];  
 }  
 }  
 int movement = abs(track - cur\_track);  
 total\_movement += movement;  
 cur\_track = track;  
 **ON\_VERBOSE** printf("%d];%d];%d\n", track, movement, total\_movement);  
 }  
 printf("SCAN TOTAL: %d\n", total\_movement);  
}  
  
int main() {  
 populate\_prophecy();  
 print\_prophecy();  
  
 populate\_sequence();  
 fcfs();  
  
 populate\_sequence();  
 sstf();  
  
 populate\_sequence();  
 scan();  
 return 0;  
}

1. 截屏输出实验结果；



1. 根据实验结果与理论课讲述的原理进行实验分析。

（一）FCFS不考虑远近，只考虑先后，基本是移动最多的。

（二）SSTF对当前序列进行一个排序，从当前的位置寻找距离最短的磁道进行移动，所以基本是移动最少的。

（三）SCAN考虑了请求队列具有动态性质，因此兼顾了以上两种算法，以磁头目前的移动方向一直移动下去，直到该方向上没有磁道需要移动，再往反方向移动。

**六、思考题**

1. 通过对每个算法进行时间复杂度分析对比，每个算法的效率如何？

（一）FCFS不考虑远近，只考虑先后，基本是移动最多的。

（二）SSTF对当前序列进行一个排序，从当前的位置寻找距离最短的磁道进行移动，所以基本是移动最少的。

（三）SCAN考虑了请求队列具有动态性质，因此兼顾了以上两种算法，以磁头目前的移动方向一直移动下去，直到该方向上没有磁道需要移动，再往反方向移动。

2、若所有硬盘全部设计成电子硬盘，哪个磁盘调度算法最合适？

电子硬盘不存在物理寻磁道的操作，因此选择算法复杂度最简单的先来先服务算法（FCFS）。