

中国地质大学（北京）

操作系统课程设计报告

学 院：信息工程学院

专 业：计算机科学与技术

班 级：10042111

学 号：1004211127

姓 名： 王俊博

联系方式：+86 15143709198

邮 箱：junbowang@email.cugb.edu.cn

指导老师：刘品

日 期：

**摘 要**

**本次实验由操作系统四大经典算法的模拟实现组成，包括银行家算法，处理机管理，存储器管理采用可变式分区管理由空闲区表和空闲区链法两种方法实现，**

**以及磁盘调度算法实现，实训项目选择阅读Linux源码的处理机调度算法，其中对每个实验都进行了分析和总结。**

**关键词：银行家算法、处理机调度、存储器管理、磁盘调度、Linux源码。**

**目 录**

目录

[一、 实验环境： 3](#_Toc137715956)

[二、 操作系统理论算法实验： 4](#_Toc137715957)

[1. 编制银行家算法通用算法 4](#_Toc137715958)

[1) 需求分析： 4](#_Toc137715959)

[2) 概要设计： 5](#_Toc137715960)

[3) 详细设计： 7](#_Toc137715961)

[4) 调试分析： 15](#_Toc137715962)

[5) 结论和期望： 17](#_Toc137715963)

[2. 处理机管理 17](#_Toc137715964)

[1) 需求分析： 17](#_Toc137715965)

[2) 概要设计： 18](#_Toc137715966)

[3) 详细设计： 19](#_Toc137715967)

[4) 调试分析： 23](#_Toc137715968)

[5) 结论和期望： 24](#_Toc137715969)

[3. 存储器管理采用可变分区管理 24](#_Toc137715970)

[1) 需求分析： 24](#_Toc137715971)

[2) 概要分析: 25](#_Toc137715972)

[3) 详细分析： 27](#_Toc137715973)

[4) 调试分析： 41](#_Toc137715974)

[5) 结论和期望： 50](#_Toc137715975)

[4. 磁盘调度算法 51](#_Toc137715976)

[1) 需求分析： 51](#_Toc137715977)

[2) 概要设计： 51](#_Toc137715978)

[3) 详细设计： 59](#_Toc137715979)

[4) 调试设计： 66](#_Toc137715980)

[5) 结论和期望： 67](#_Toc137715981)

[三、实验总结： 68](#_Toc137715982)

# 实验环境：

Operating System：Window11 22000.2295

CPU: 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz

IDE: PyCharm Community Edition 2021.3.3

# 操作系统实践算法实验：

## 编制银行家算法通用算法

### 需求分析：

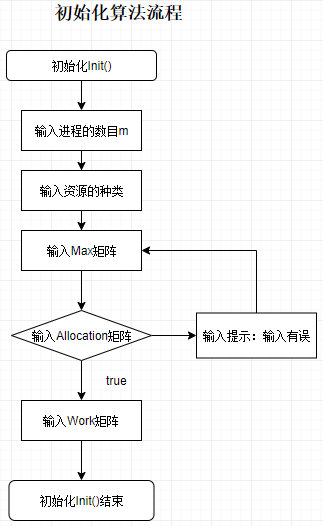
检测所给状态的系统安全性。假定系统的任何一种资源在任意时刻只能被一个进程使用。任何进程已经占用的资源只能进程自己释放，而 不能由其它进程抢占。进程申请的资源不能满足时，必须等待。 设计的要求：

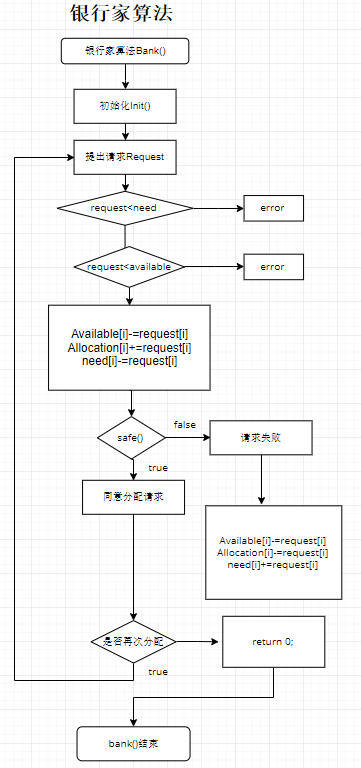
(1) 程序中使用的数据结构及主要符号说明

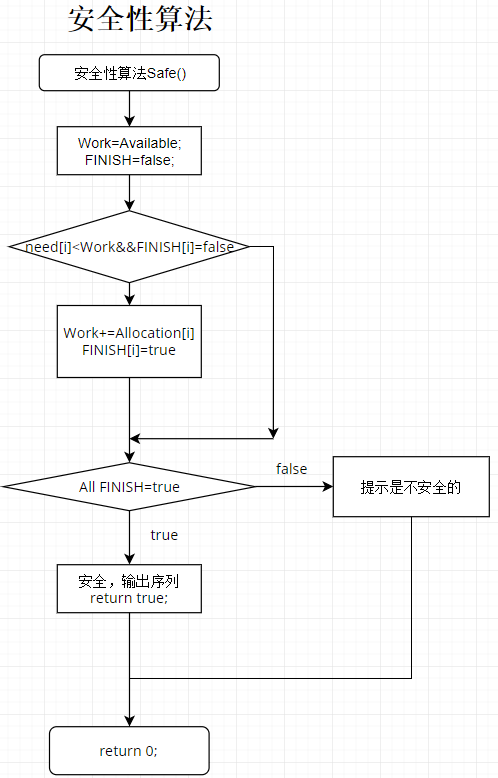
(2) 资源的种类和数目可以变化的

(3) 进程可以任意的顺序创建和变化

### 概要设计：





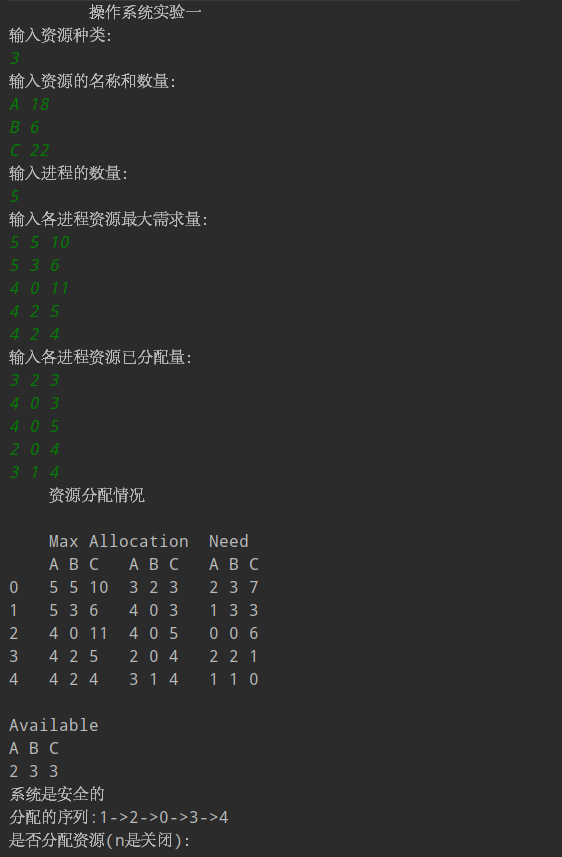


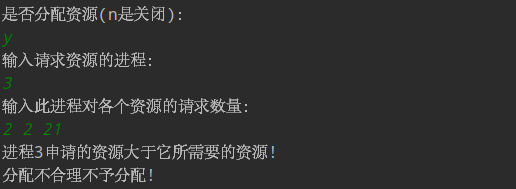
### 详细设计：

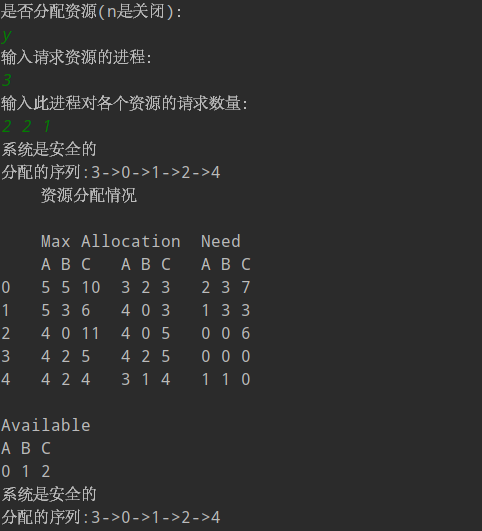
1. **package** OS;
3. **import** java.util.Scanner;
5. /\*\*
6. \* 假定系统的任何一种资源在任一时刻只能被一个进程使用。
7. \* 任何进程已经占用的资源只能由进程自己释放，
8. \* 而不能由其它进程抢占。进程申请的资源不能满足时，必须等待。
9. \* 设计的要求：
10. \* (1) 程序中使用的数据结构及主要符号说明
11. \* (2) 资源的种类和数目可以变化的
12. \* (3) 进程可以任意的顺序创建和变化
13. \*/

16. **public** **class** BankerOfAlgorithm {//BankerOfAlgorithm  银行家算法
17. **int** Max[][];//最大资源数量
18. **int** Allocation[][];//已分配的资源数量
19. **int** Need[][];//进程还需要的资源数量
20. **int** Available[];//可以使用的资源数量
21. **int** Work[];//temp
22. String name[];
23. **int** temp[];
24. **int** S = 100, P = 100;
25. **int** safequeue[];//安全的序列
26. **int** Request[];
27. Scanner sc;
29. **public** BankerOfAlgorithm() {//初始化函数
30. **int** text\_temp = 100;
31. **int** Max[][] = **new** **int**[text\_temp][text\_temp];
32. **int** Allocation[][] = **new** **int**[text\_temp][text\_temp];
33. **int** Need[][] = **new** **int**[text\_temp][text\_temp];
34. **int** Available[] = **new** **int**[text\_temp];
35. **int** Work[] = **new** **int**[text\_temp];
36. String name[] = **new** String[text\_temp];
37. **int** temp[] = **new** **int**[text\_temp];
38. **int** S = 100, P = 100;
39. **int** safequeue[] = **new** **int**[text\_temp];
40. **int** Request[] = **new** **int**[text\_temp];
42. }
44. **public** **void** Showdata() {//显示数据
45. **int** i, j, k, l;
46. System.out.println("\t资源分配情况\n");
47. System.out.println("\tMax\tAllocation\tNeed");
48. System.out.print("\t");
49. **for** (j = 0; j < 3; j++) {
50. **for** (i = 0; i < S; i++) {
51. System.out.print(name[i] + " ");
52. }
53. System.out.print("\t");
54. }
55. System.out.println();
56. **for** (i = 0; i < P; i++) {
57. System.out.print(i+"\t");
58. **for** (j = 0; j < S; j++) {
59. System.out.print(Max[i][j] + " ");//Max矩阵
60. }
61. System.out.print("\t");
62. **for** (k = 0; k < S; k++) {
63. System.out.print(Allocation[i][k] + " ");//Allocation矩阵
64. }
65. System.out.print("\t");
66. **for** (l = 0; l < S; l++) {
67. System.out.print(Need[i][l] + " ");//Need矩阵
68. }
69. System.out.println();
70. }
71. System.out.println("\nAvailable");
72. **for** (i = 0; i < S; i++) {
73. System.out.print(name[i] + " ");
74. }
75. System.out.println();
76. **for** (i = 0; i < S; i++) {
77. System.out.print(Available[i] + " ");
78. }
79. System.out.println();
80. }
82. **public** **int** Judgesafe() {//安全性算法
83. **int**[][] tempwork = **new** **int**[100][100];
84. **int** i, x, k = 0, m, apply;
85. **int**[] Finish = **new** **int**[100];
86. **int**[] temp = **new** **int**[100];
87. **int** j;
88. **int** flag = 0;
89. Work = **new** **int**[100];
90. **for** (i = 0; i < S; i++) {
91. Work[i] = Available[i];
92. }
93. **for** (i = 0; i < P; i++) {
94. apply = 0;
95. **for** (j = 0; j < S; j++) {
96. **if** (Finish[i] == 0 && Need[i][j] <= Work[j]) {//如果Need<Work则处于不安全序列，则不能进行分配
97. apply++;
98. **if** (apply == S) {
99. **for** (m = 0; m < S; m++) {
100. tempwork[i][m] = Work[m];
101. Work[m] = Work[m] + Allocation[i][m];
102. }
103. Finish[i] = 1;
104. temp[k] = i;
105. i = -1;
106. k++;
107. flag++;
108. }
109. }
110. }
111. }
112. **for** (i = 0; i < P; i++) {
113. **if** (Finish[i] == 0) {
114. System.out.println("系统不安全");
115. **return** -1;
116. }
117. }
118. System.out.println("系统是安全的");
119. System.out.print("分配的序列:");//输出分配的安全序列
120. **for** (i = 0; i < P; i++) {
121. System.out.print(temp[i]);
122. **if** (i < P - 1) {
123. System.out.print("->");
124. }
125. }
126. System.out.println();
127. **return** 0;
128. }
130. **public** **void** Changedata(**int** flag) {
131. **for** (**int** i = 0; i < S; i++) {
132. Available[i] = Available[i] - Request[i];
133. Allocation[flag][i] = Allocation[flag][i] + Request[i];
134. Need[flag][i] = Need[flag][i] - Request[i];
135. }
136. }
138. **public** **void** Share() {
139. sc = **new** Scanner(System.in);
140. **int** i, flag;
141. **char** ch = 'Y';
142. System.out.println("输入请求资源的进程：");
143. flag = sc.nextInt();
144. Request = **new** **int**[100];
145. **if** (flag >= P) {
146. System.out.println("此进程不存在!");
147. } **else** {
148. System.out.println("输入此进程对各个资源的请求数量：");
149. **for** (i = 0; i < S; i++) {
150. Request[i] = sc.nextInt();
151. }
152. **for** (i = 0; i < S; i++) {
153. **if** (Request[i] > Need[flag][i]) {
154. System.out.println("进程" + flag + "申请的资源大于它所需要的资源!");
155. System.out.println("分配不合理不予分配!");
156. ch = 'N';
157. **break**;
158. } **else** **if** (Request[i] > Available[i]) {
159. System.out.println("进程" + flag + "申请的资源大于可利用的资源。");
160. System.out.println("分配不合理，不予分配!");
161. ch = 'N';
162. **break**;
163. }
164. }
165. **if** (ch == 'Y') {
166. Changedata(flag);
167. **if** (Judgesafe() == -1) {//处理并且改变数据
168. System.out.println("进程" + flag + "申请资源后，系统进入死锁状态，分配失败!");
169. **for** (**int** j = 0; j < S; j++) {
170. Available[j] = Available[j] + Request[j];
171. Allocation[flag][j] = Allocation[flag][j] - Request[j];
172. Need[flag][j] = Need[flag][j] + Request[j];
173. }
174. }
175. }
176. }
177. }
179. **public** **static** **void** main(String[] args) {
180. BankerOfAlgorithm mytext = **new** BankerOfAlgorithm();
181. Scanner sc = **new** Scanner(System.in);
182. **int** i, j, p, s, number;
183. String choice, tempstring;
184. System.out.println("\t\t操作系统实验一");
185. System.out.println("输入资源种类：");
186. s = sc.nextInt();
187. mytext.S = s;
188. System.out.println("输入资源的名称和数量：");
189. mytext.name = **new** String[100];
190. mytext.Available = **new** **int**[100];
191. **for** (i = 0; i < s; i++) {
192. tempstring = sc.next();
193. number = sc.nextInt();
194. mytext.name[i] = tempstring;
195. mytext.Available[i] = number;
196. }
197. System.out.println("输入进程的数量：");
198. p = sc.nextInt();
199. mytext.P = p;
200. System.out.println("输入各进程资源最大需求量：");
201. mytext.Max = **new** **int**[100][100];
202. **for** (i = 0; i < p; i++) {
203. **for** (j = 0; j < s; j++) {
204. mytext.Max[i][j] = sc.nextInt();
205. }
206. }
207. System.out.println("输入各进程资源已分配量：");
208. mytext.Allocation = **new** **int**[100][100];
209. **for** (i = 0; i < p; i++) {
210. **for** (j = 0; j < s; j++) {
211. mytext.Allocation[i][j] = sc.nextInt();
212. mytext.Available[j] = mytext.Available[j]
213. - mytext.Allocation[i][j];
214. }
215. }
216. mytext.Need = **new** **int**[100][100];
217. **for** (i = 0; i < p; i++) {
218. **for** (j = 0; j < s; j++) {
219. mytext.Need[i][j] = mytext.Max[i][j] - mytext.Allocation[i][j];
220. }
221. }
222. mytext.Showdata();
223. mytext.Judgesafe();
224. **while** (**true**) {
225. System.out.println("是否分配资源(n是关闭)：");
226. choice = sc.next();
227. **char** choice\_char = choice.charAt(0);
228. **switch** (choice\_char) {
229. **case** 'n':
230. System.exit(0);
231. **default**:
232. mytext.Share();
233. **break**;
234. }
235. mytext.Showdata();
236. mytext.Judgesafe();
237. }
238. }
240. }

### 调试分析：







### 结论和期望：

银行家算法是避免死锁（DeadLock）的著名算法，它以银行借贷系统的分配策略为基础，判断并保证系统的安全运行。避免死锁和预防死锁不相同。通过本次实验简单模拟系统的一种安全性算法，程序一开始只判断了Request数组和Need数组的大小没有判断Request数组和Available数组的大小导致进行没有必要的不安全的状态。

## 处理机管理

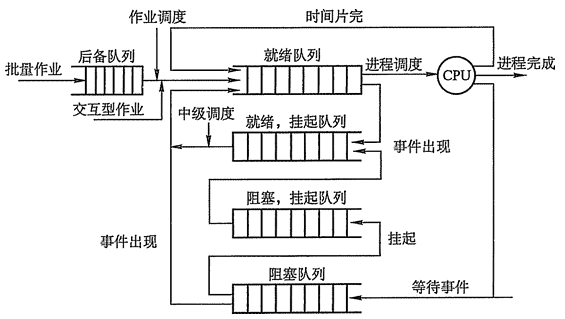
### 需求分析：

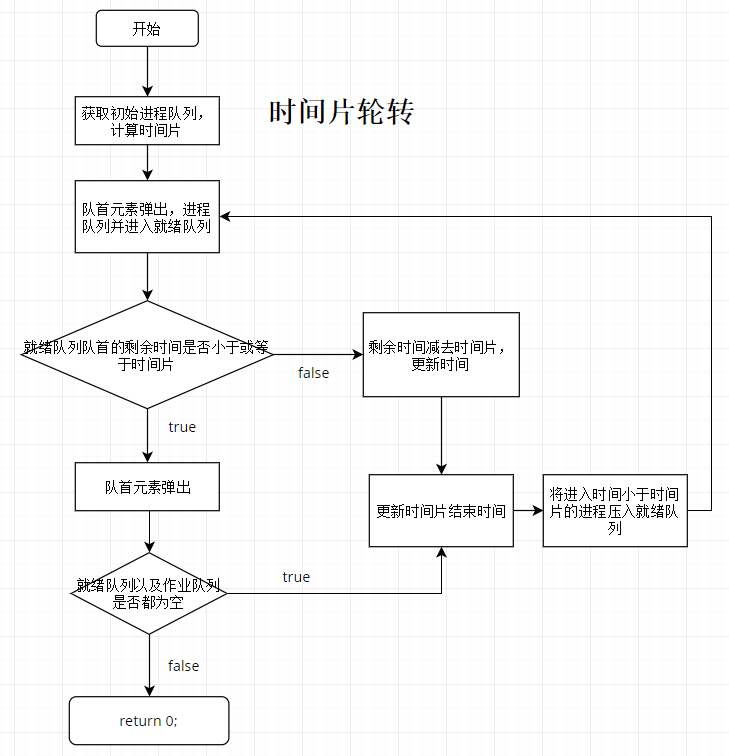
设计程序模拟进程的轮转法调度过程。假设初始状态为：有 n 个进程处于就绪状态，有 m 个进程处于阻塞状态。采用轮转法进程调度算法、高响应比优先（HRRN）进行调度 (调度过程中，假设处于执行状态的进程不会阻塞)，且每过 t 个时间片系统释放资源，唤醒处于阻塞队列队首的进程。程序要求如下：

(1) 输出系统中进程的调度次序；

(2) 计算 CPU 利用率。

### 概要设计：

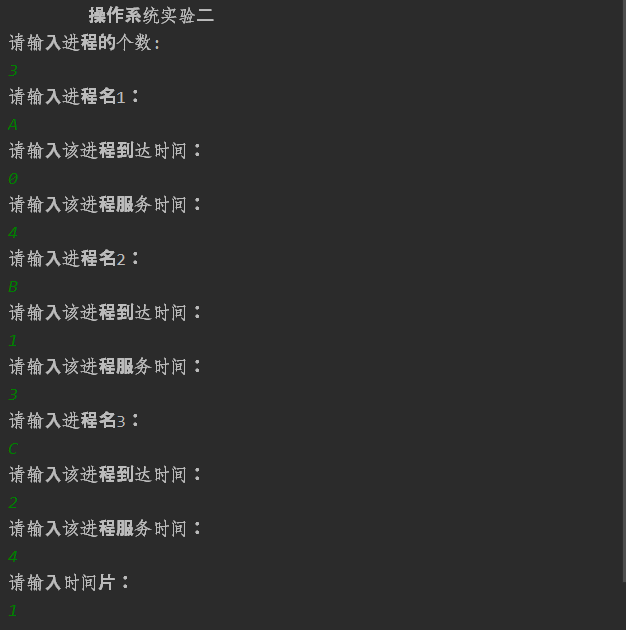


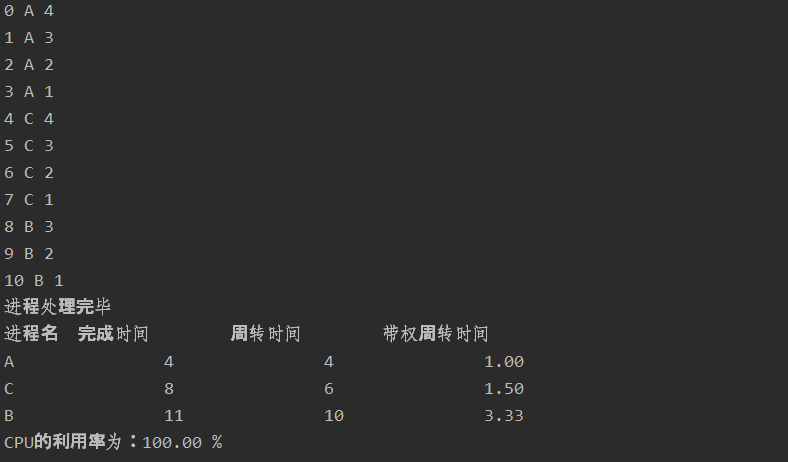


### 详细设计：

1. //
2. // Created by Misser on 2019/11/18.
3. // 假设初始状态为：有 n 个进程处于就绪状态，
4. // 有m 个进程处于阻塞状态。采用轮转法进程调度算法、高响应比优先（HRRN）进行调度
5. //(调度过程中，假设处于执行状态的进程不会阻塞)，且每过 t 个时间片系统释放资源，
6. // 唤醒处于阻塞队列队首的进程。程序要求如下
7. //(1) 输出系统中进程的调度次序
8. //(2) 计算 CPU 利用率。
9. //
10. #include<bits/stdc++.h>
11. #define eps (1e-8)
12. **using** **namespace** std;
13. **struct** prog{
14. **char** name[100];
15. **int** atim;  //到达时间
16. **int** ptim;  //服务时间
17. **int** pptim;
18. **int** ftim;  //完成时间
19. **int** rtim;  //周转时间
20. **int** wtim;  //HRRN算法等待时间
21. **double** pror; //优先级
22. **double** rrtim; //带权周转时间
23. **friend** **bool** operator<(prog a,prog b)
24. {
25. **if**((a.pror-b.pror)<eps)
26. **return** a.atim>b.atim;
27. **return** (a.pror-b.pror)>eps;
28. }
29. } pro[50];
30. **bool** cmp(prog a,prog b)
31. {
32. **return** a.ftim<b.ftim;
33. }
34. **int** n,t,p=0,num=0,pp=0;//进程数 时间片 当前时刻
35. //模拟调度队列
36. priority\_queue<prog>q;
37. **void** init()
38. {
39. printf("请输入进程的个数:\n");
40. scanf("%d",&n);
41. **int** cur=1;
42. **for**(**int** i=1; i<=n; i++)
43. {
44. printf("请输入进程名%d：\n",cur++);
45. scanf("%s",&pro[i].name);
46. printf("请输入该进程到达时间：\n");
47. scanf("%d",&pro[i].atim);
48. printf("请输入该进程服务时间：\n");
49. scanf("%d",&pro[i].ptim);
50. pro[i].pptim=pro[i].ptim;
51. pro[i].wtim=0;
52. pro[i].rtim=0;
53. pro[i].pror=(pro[i].wtim+pro[i].ptim)\*1.0/pro[i].ptim;//计算响应比=（等待时间+被要求服务时间）/被要求服务时间
54. }
55. printf("请输入时间片：\n");
56. scanf("%d",&t);
57. }
58. **void** solve()
59. {
60. **for**(**int** i=1; i<=n; i++) q.push(pro[i]);
61. **while**(!q.empty())
62. {
63. prog cnt=q.top();
64. **if**(cnt.atim>p)
65. {
66. p++;
67. pp++;
68. **continue**;
69. }
70. q.pop();
71. **int** tmp=t;
72. cnt.wtim=0;
73. **while**(tmp--)
74. {
75. cout<<p<<" "<<cnt.name<<" "<<cnt.ptim<<endl;
76. cnt.ptim--;
77. p++;
78. **for**(**int** i=1; i<=n; i++)
79. {
80. **if**(strcmp(cnt.name,pro[i].name)!=0&&pro[i].ptim!=0&&p>pro[i].atim)
81. {
82. pro[i].wtim++;
83. pro[i].pror=(pro[i].wtim+pro[i].ptim)\*1.0/pro[i].ptim;
84. }
85. }
86. **while**(!q.empty())
87. {
88. q.pop();
89. }
90. **for**(**int** i=1; i<=n; i++)
91. {
92. **if**(strcmp(cnt.name,pro[i].name)!=0&&pro[i].ptim!=0)
93. {
94. q.push(pro[i]);
95. }
96. }
97. **if**(cnt.ptim==0)
98. {
99. num++;
100. **for**(**int** i=1; i<=n; i++)
101. {
102. **if**(strcmp(cnt.name,pro[i].name)==0)
103. {
104. pro[i].ftim=p;
105. pro[i].rtim=p-pro[i].atim;
106. pro[i].ptim=0;
107. pro[i].rrtim=(pro[i].rtim\*1.0)/(pro[i].pptim\*1.0);
108. **break**;
109. }
110. }
111. **if**(tmp!=0)
112. {
113. pp+=tmp;
114. }
115. **else**
116. **continue**;
117. }
118. **if**(tmp==0&&cnt.ptim>0)
119. {
120. cnt.pror=1;///这里将等待时间置零
121. q.push(cnt);
122. }
123. **if**(num==n)
124. **return**;
125. }
126. }
127. }
128. **void** output()
129. {
130. printf("进程处理完毕\n");
131. sort(pro+1,pro+n+1,cmp);
132. printf("进程名\t完成时间\t周转时间\t带权周转时间\n");
133. **for**(**int** i=1; i<=n; i++)
134. {
135. printf("%s\t\t%d\t\t%d\t\t%.2lf\n",pro[i].name,pro[i].ftim,pro[i].rtim,pro[i].rrtim);
136. }
137. printf("CPU的利用率为：%.2lf %%\n",(p-pp)\*1.0/p\*1.0\*100);
138. }
139. **int** main()
140. {
141. init();
142. solve();
143. output();
144. **return** 0;
145. }

### 调试分析：





### 结论和期望：

本实验处理机调度采用时间片轮转和高响应比优先的算法，为后面理解Linux系统的进程调度打下了基础，本次实验使用优先级队列的方式来模拟阻塞队列，同时采用非抢占式的调度算法，理解周转时间和带权周转时间的算法，希望可以将程序修改得更加完善。

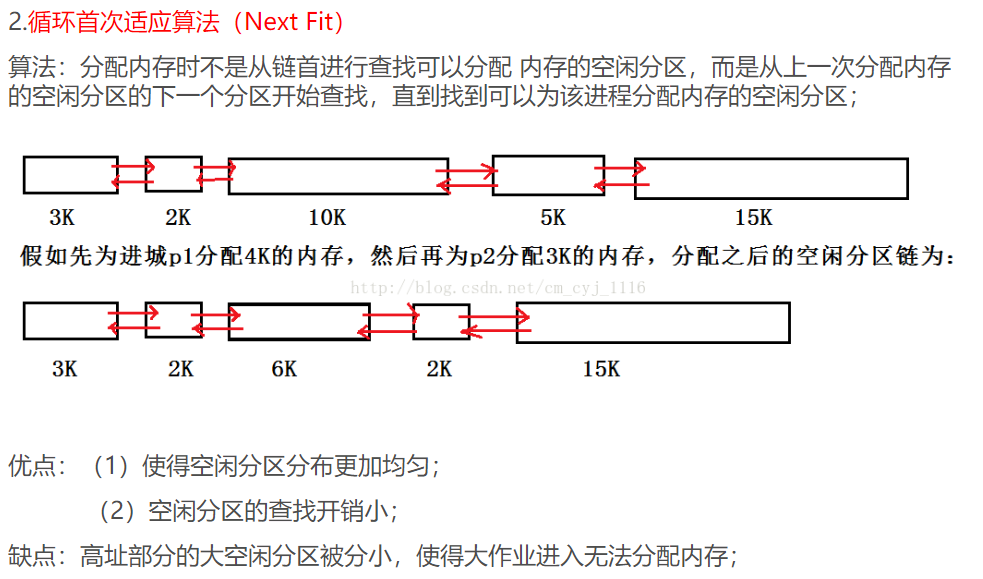
## 存储器管理采用可变分区管理

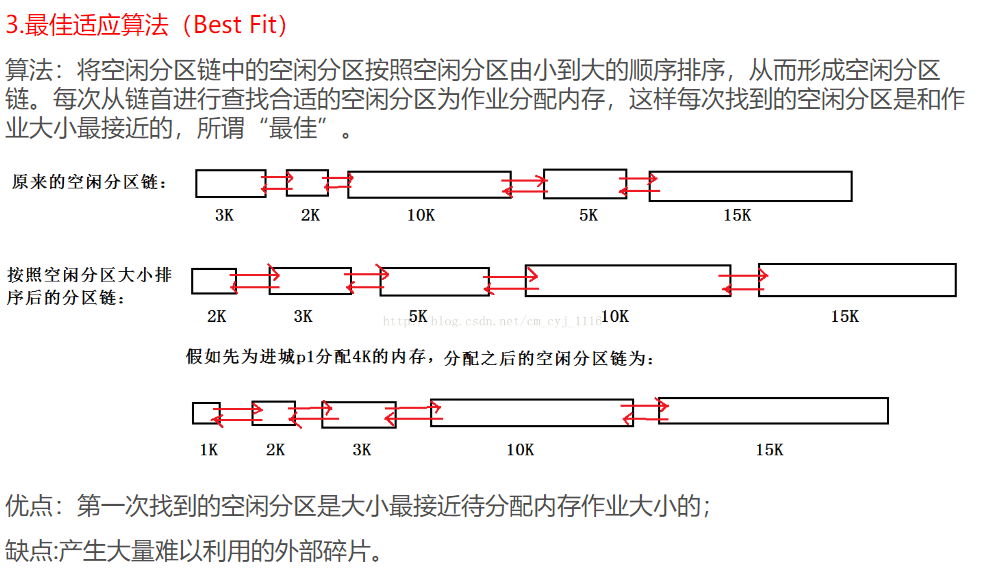
### 需求分析：

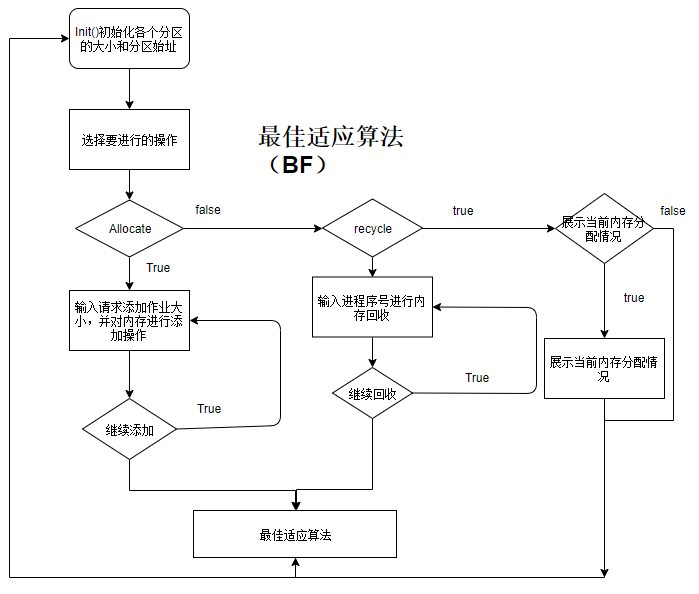
1. 采用空闲区表，并增加已分配区表{未分配区说明表、已分配区说明表（分区号、 起始地址、长度、状态）}。分配算法采用最佳适应算法（内存空闲区按照尺寸大小从小到 大的排列）和循环首次适应算法，实现内存的分配与回收。
2. 采用空闲区链法管理空闲区，结构如 P137/P128，并增加已分配区表。分配算法分别采用首次适应法（内存空闲区的地址按照从小到大的自然顺序排列）和最佳适应法（按照 内存大小从小到大排列），实现内存的分配与回收。

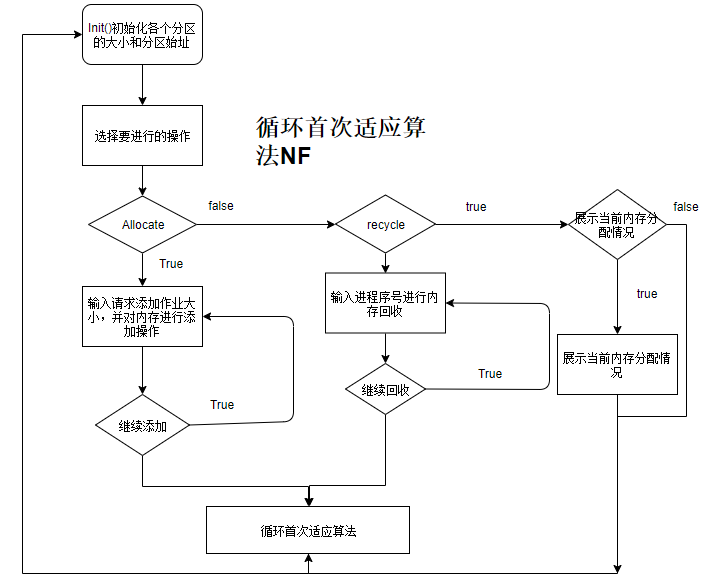
学生自己设计一个进程申请序列以及进程完成后的释放顺序，实现主存的分配与回收。 进程分配时，应该考虑以下 3 中情况：进程申请的空间小于、大于或者等于系统空闲区 的大小。回收时，应该考虑 4 种情况：释放区上邻、下邻、上下都邻接和都不邻接空闲区。 每次的分配与回收，都要求把记录内存使用情况的各种数据结构的变化情况以及各进程的申请、释放情况显示或打印出来。

### 概要分析:









### 详细分析：

#### 空闲区表法

1. //
2. // Created by Misser on 2019/11/20.
3. //
4. /\*\*\*
5. \* （1）采用空闲区表，并增加已分配区表{未分配区说明表、已分配区说明表（分区号、
6. 起始地址、长度、状态）}。分配算法采用最佳适应算法（内存空闲区按照尺寸大小从小到
7. 大的排列）和循环首次适应算法，实现内存的分配与回收。
8. \*/
9. #include <bits/stdc++.h>
10. **using** **namespace** std;
11. **int** pos,n,Size,k; ///查找位置,分区数量,最小分割大小,作业的编号
12. **struct** Memory\_Block
13. {
14. **int** id;///空闲分区编号
15. **int** sad;///空闲分区首地址
16. **int** rom;///空间
17. **int** state;///状态，0为空，1为未满，2为满；
18. } MB[2000];
19. **struct** Task
20. {
21. **int** id;
22. **int** rom;
23. **int** sad;
24. } T[2000];
25. **bool** cmp(Memory\_Block a,Memory\_Block b)
26. {
27. **return** a.rom<b.rom;
28. }
29. **void** print()
30. {
31. **int** i;
32. printf("\t分区号\t分区始址\t分区大小\t分区状态\n");
33. **for**(i=1; i<=n; i++)
34. {
35. printf("\t%4d\t%4d\t\t%4d\t\t%4d\t\n",MB[i].id,MB[i].sad,MB[i].rom,MB[i].state);
36. }
38. }
39. **void** recycle()
40. {
41. printf("请输入要释放占用空间任务的序号：\n");
42. **int** l;
43. **int** f1=0,f2=0,f3=0;
44. **int** p1,p2;
45. cin>>l;
46. **for**(**int** i=1; i<=n; i++)
47. {
48. **if**(T[l].sad==MB[i].sad+MB[i].rom)
49. {
50. f1=1;
51. p1=i;
52. }
53. **if**(T[l].sad+T[l].rom==MB[i].sad)
54. {
55. f2=1;
56. p2=i;
57. }
58. }
59. **if**(f1==1&&f2==0)//上面有空，下面没有空
60. {
61. MB[p1].rom=MB[p1].rom+T[l].rom;
62. }
63. **else** **if**(f1==0&&f2==1)//下面有空，上面没有空
64. {
65. MB[p2].rom=MB[p2].rom+T[l].rom;
66. MB[p2].sad=T[l].sad;
67. }
68. **else** **if**(f1==1&&f2==1)//上面，下面都有空
69. {
70. MB[p1].rom=MB[p1].rom+T[l].rom+MB[p2].rom;
71. **for**(**int** j=p1+1; j<=n; j++)
72. MB[j].id--;
73. n--;//合并操作
74. }
75. **else** **if**(f1==0&&f2==0)//上面和下面都没有空 直接去除即可
76. {
77. **int** temp;
78. **for**(**int** j=1; j<=n; j++)
79. {
80. **if**(MB[j].sad>T[l].sad+T[l].rom)///实际上大于T[l].sad就可以
81. {
82. temp=j;
83. }
84. }
85. **for**(**int** j=temp; j<=n; j++)
86. T[j].id++;
87. n++;
88. MB[temp].id=temp;
89. MB[temp].rom=T[l].rom;
90. MB[temp].sad=T[l].sad;
91. MB[temp].state=0;
92. }
93. cout<<k<<"号进程内存回收完毕！"<<endl;
94. k--;
95. }
96. **void** NF()///循环首次适应算法
97. {
98. cout<<"内存分配\n";
99. **int** tmp;
100. pos=1;///开始从第一个分区查找
101. **while**(**true**)
102. {
103. cout<<"1: 分配内存\n2: 回收内存\n3: 查看空闲分区表\n0: 退出\n";
104. cout<<"请输入操作 ：";
105. cin>>tmp;
106. **int** k=0;
107. **if**(tmp==1)
108. {
109. k++;
110. printf("请输入第%d个作业占用空间大小：\n",k);
111. cin>>T[k].rom;
112. T[k].id=k;
113. **int** num=0;
114. **for**(**int** i=pos;; i++)
115. {
116. num++;
117. **if**(num>n)
118. {
119. printf("作业请求内存空间过大，空闲分区表不能满足要求，内存分配失败!\n");
120. **break**;
121. }
122. **if**(i>n)
123. {
124. i=1;
125. **continue**;
126. }
127. **if**((MB[i].state==0||MB[i].state==1)&&MB[i].rom>=T[i].rom)
128. {
129. **if**(MB[i].rom-T[k].rom>=Size)
130. {
131. MB[i].sad=MB[i].sad+T[k].rom;
132. MB[i].rom=MB[i].rom-T[k].rom;
133. MB[i].state=1;
134. T[k].sad=MB[i].sad-T[k].rom;
135. printf("内存分配成功！\n作业申请空间为%d\n起始地址为%d\n",T[k].rom,T[k].sad);
136. **break**;
137. }
138. }
139. **else** **if**(MB[i].state==0&&MB[i].rom-T[k].rom==Size)
140. {
141. MB[i].state=2;
142. T[k].sad=MB[i].sad;
143. printf("内存分配成功！\n作业申请空间为%d\n起始地址为%d\n",T[k].rom,T[k].sad);
144. **break**;
145. }
146. }
147. }
148. **else** **if**(tmp==2)
149. {
150. recycle();
151. }
152. **else** **if**(tmp==3)
153. print();
154. **else** **if**(tmp==0)
155. **return**;
156. **else**
157. {
158. printf("输入有误，请重新输入！\n");
159. **continue**;
160. }
161. }
162. }
163. **void** BF()///最佳适应算法
164. {
165. cout<<"内存分配\n";
166. sort(MB+1,MB+n+1,cmp);
167. **int** tmp;
168. **while**(**true**)
169. {
170. cout<<"1: 分配内存\n2: 回收内存\n3: 查看空闲分区表\n0: 退出\n";
171. cout<<"请输入操作 ：";
172. cin>>tmp;
173. **if**(tmp==1)
174. {
175. k++;
176. printf("请输入第%d个作业占用空间大小：\n",k);
177. cin>>T[k].rom;
178. T[k].id=k;
179. **int** i;
180. **for**(i=1; i<=n; i++)
181. {
182. **if**(i>n)
183. {
184. i=1;
185. **continue**;
186. }
187. **if**((MB[i].state==0||MB[i].state==1)&&MB[i].rom>=T[i].rom)//如果状态是不满或者空则可以分配
188. {
189. **if**(MB[i].rom-T[k].rom>=Size)//大于最小的分配空间
190. {
191. MB[i].sad=MB[i].sad+T[k].rom;//分区的开始地址减小
192. MB[i].rom=MB[i].rom-T[k].rom;//可用地址减小
193. MB[i].state=1;
194. T[k].sad=MB[i].sad-T[k].rom;//作业的开始地址  = 分区的地址 - 作业的大小
195. printf("内存分配成功！\n作业申请空间为%d\n起始地址为%d\n",T[k].rom,T[k].sad);
196. **break**;
197. }
198. }
199. **else** **if**(MB[i].state==0&&MB[i].rom-T[k].rom==Size)//状态为空 所需空间正好为size
200. {
201. MB[i].state=2;
202. T[k].sad=MB[i].sad;
203. printf("内存分配成功\t作业申请空间为%d\t起始地址为%d\n",T[k].rom,T[k].sad);
204. **break**;
205. }
206. }
207. **if**(i>n)
208. {
209. printf("作业请求内存空间过大，空闲分区表不能满足要求，内存分配失败!\n");
210. **break**;
211. }
212. }
213. **else** **if**(tmp==2)
214. {
215. recycle();
216. sort(MB+1,MB+n+1,cmp);
217. }
218. **else** **if**(tmp==3)
219. print();
220. **else** **if**(tmp==0)
221. **return**;
222. **else**
223. {
224. printf("输入有误，请重新输入！\n");
225. **continue**;
226. }
227. }
228. }
229. **void** init()
230. {
231. loop1:
232. pos=1;
233. printf("请输入空闲分区表分区数量：\n");
234. cin>>n;
235. printf("请输入每个空闲分区的分区大小，分区始址\n");
236. **for**(**int** i=1; i<=n; i++)
237. {
238. printf("请输入第%d个分区的信息：\n",i);
239. cin>>MB[i].rom>>MB[i].sad;
240. MB[i].id=i;
241. MB[i].state=0;
242. }
243. printf("当前空闲分区表状态为：\n");
244. print();
245. printf("请输入不再切割的剩余空间的大小：\n");
246. cin>>Size;
247. loop2:
248. printf("选择内存分配的算法:\n1.循环首次适应算法\n2.最佳适应算法\n3.重新编辑空闲分区表\n4.退出\n");
249. **int** tmp;
250. cin>>tmp;
251. **if**(tmp==1)NF();
252. **else** **if**(tmp==2)BF();
253. **else** **if**(tmp==3)
254. {
255. **goto** loop1;
256. }
257. **else** **if**(tmp==4) **return**;
258. **else**
259. {
260. printf("输入有误，请重新输入!\n");
261. **goto** loop2;
262. }
263. }
265. **int** main()
266. {
267. init();
268. **return** 0;
269. }

#### 空闲区链法

1. //
2. // Created by Misser on 2019/11/21.
3. //
4. /\*\*\*
5. \* （2）采用空闲区链法管理空闲区，结构如 P137/P128，并增加已分配区表。分配算法分
6. 别采用首次适应法（内存空闲区的地址按照从小到大的自然顺序排列）和最佳适应法（按照
7. 内存大小从小到大排列），实现内存的分配与回收。
8. \*/
9. #include <bits/stdc++.h>
10. **using** **namespace** std;
11. #define Free 0 //空闲状态
12. #define Busy 1 //已用状态
13. #define OK 1    //完成
14. #define ERROR 0 //出错
15. #define MAX\_length 640  //定义最大主存信息640KB
16. **typedef** **int** Status;
17. **int** flag;//标志位  0为空闲区     1为已分配的工作区
19. **typedef** **struct** FreAarea//定义一个空闲区说明表结构
20. {
21. **long** size;   //分区大小
22. **long** address; //分区地址
23. **int** state;   //状态
24. }ElemType;

27. **typedef** **struct** DuLNode// 线性表的双向链表存储结构
28. {
29. ElemType data;
30. **struct** DuLNode \*prior; //前趋指针
31. **struct** DuLNode \*next;  //后继指针
32. }

35. DuLNode, \*DuLinkList;
36. DuLinkList block\_first; //头结点
37. DuLinkList block\_last;  //尾结点
38. Status Alloc(**int**);//内存分配
39. Status free(**int**); //内存回收
40. Status First\_fit(**int**);//首次适应算法
41. Status Best\_fit(**int**); //最佳适应算法
42. **void** show();//查看分配
43. Status Initblock();//开创空间表

46. Status Initblock()//开创带头结点的内存空间链表
47. {
48. block\_first = (DuLinkList)malloc(**sizeof**(DuLNode));
49. block\_last = (DuLinkList)malloc(**sizeof**(DuLNode));
50. block\_first->prior = NULL;
51. block\_first->next = block\_last;
52. block\_last->prior = block\_first;
53. block\_last->next = NULL;
54. block\_last->data.address = 0;
55. block\_last->data.size = MAX\_length;
56. block\_last->data.state = Free;
57. **return** OK;
58. }

61. Status Alloc(**int** ch)//分配主存
62. {
63. **int** request = 0;
64. cout << "请输入需要分配的主存大小(单位:KB)："<<endl;
65. cin >> request;
66. **if** (request<0 || request == 0)
67. {
68. cout << "分配大小不合适，请重试！" << endl;
69. **return** ERROR;
70. }
72. **if** (ch == 2) //选择最佳适应算法
73. {
74. **if** (Best\_fit(request) == OK) cout << "分配成功！" << endl;
75. **else** cout << "内存不足，分配失败！" << endl;
76. **return** OK;
77. }
78. /\*if (ch == 3) //选择最差适应算法
79. {
80. if (Worst\_fit(request) == OK) cout << "分配成功！" << endl;
81. else cout << "内存不足，分配失败！" << endl;
82. return OK;
83. }\*/
84. **else** //默认首次适应算法
85. {
86. **if** (First\_fit(request) == OK) cout << "分配成功！" << endl;
87. **else** cout << "内存不足，分配失败！" << endl;
88. **return** OK;
89. }
90. }

93. Status First\_fit(**int** request)//首次适应算法
94. {
95. //为申请作业开辟新空间且初始化
96. DuLinkList temp = (DuLinkList)malloc(**sizeof**(DuLNode));
97. temp->data.size = request;
98. temp->data.state = Busy;
100. DuLNode \*p = block\_first->next;
101. **while** (p)
102. {
103. **if** (p->data.state == Free && p->data.size == request)
104. {//有大小恰好合适的空闲块
105. p->data.state = Busy;
106. **return** OK;
107. **break**;
108. }
109. **if** (p->data.state == Free && p->data.size>request)
110. {//有空闲块能满足需求且有剩余
111. temp->prior = p->prior;
112. temp->next = p;
113. temp->data.address = p->data.address;
114. p->prior->next = temp;
115. p->prior = temp;
116. p->data.address = temp->data.address + temp->data.size;
117. p->data.size -= request;
118. **return** OK;
119. **break**;
120. }
121. p = p->next;
122. }
123. **return** ERROR;
124. }

127. Status Best\_fit(**int** request)//最佳适应算法
128. {
129. **int** ch; //记录最小剩余空间
130. DuLinkList temp = (DuLinkList)malloc(**sizeof**(DuLNode));
131. temp->data.size = request;
132. temp->data.state = Busy;
133. DuLNode \*p = block\_first->next;
134. DuLNode \*q = NULL; //记录最佳插入位置
136. **while** (p) //初始化最小空间和最佳位置
137. {
138. **if** (p->data.state == Free && (p->data.size >= request))
139. {
140. **if** (q == NULL)
141. {
142. q = p;
143. ch = p->data.size - request;
144. }
145. **else** **if** (q->data.size > p->data.size)
146. {
147. q = p;
148. ch = p->data.size - request;
149. }
150. }
151. p = p->next;
152. }
154. **if** (q == NULL) **return** ERROR;//没有找到空闲块
155. **else** **if** (q->data.size == request)
156. {
157. q->data.state = Busy;
158. **return** OK;
159. }
160. **else**
161. {
162. temp->prior = q->prior;
163. temp->next = q;
164. temp->data.address = q->data.address;
165. q->prior->next = temp;
166. q->prior = temp;
167. q->data.address += request;
168. q->data.size = ch;
169. **return** OK;
170. }
171. **return** OK;
172. }



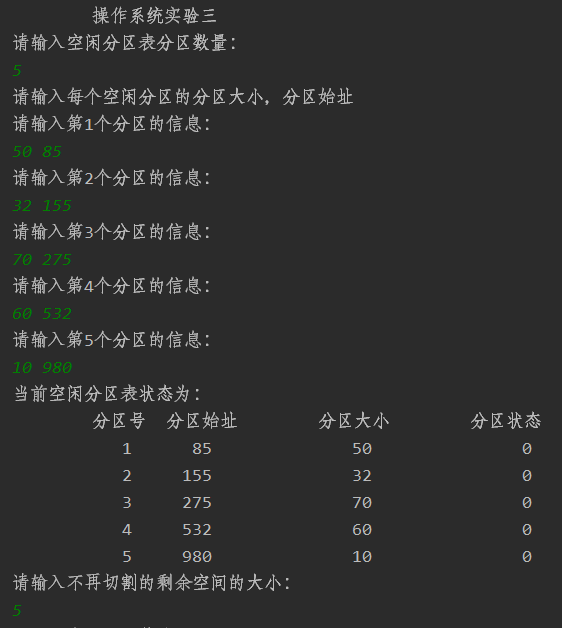
177. Status free(**int** flag)//主存回收
178. {
179. DuLNode \*p = block\_first;
180. **for** (**int** i = 0; i <= flag; i++)
181. **if** (p != NULL)
182. p = p->next;
183. **else**
184. **return** ERROR;
186. p->data.state = Free;
187. **if** (p->prior != block\_first && p->prior->data.state == Free)//与前面的空闲块相连
188. {
189. p->prior->data.size += p->data.size;//空间扩充,合并为一个
190. p->prior->next = p->next;//去掉原来被合并的p
191. p->next->prior = p->prior;
192. p = p->prior;
193. }
194. **if** (p->next != block\_last && p->next->data.state == Free)//与后面的空闲块相连
195. {
196. p->data.size += p->next->data.size;//空间扩充,合并为一个
197. p->next->next->prior = p;
198. p->next = p->next->next;
199. }
200. **if** (p->next == block\_last && p->next->data.state == Free)//与最后的空闲块相连
201. {
202. p->data.size += p->next->data.size;
203. p->next = NULL;
204. }
206. **return** OK;
207. }

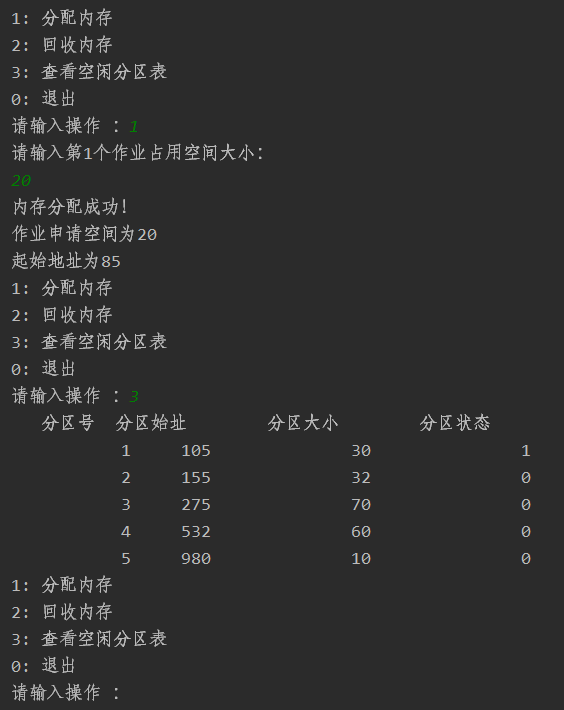
210. **void** show()//显示主存分配情况
211. {
212. **int** flag = 0;
213. cout<<"\t操作系统实验四（分区链表）"<<endl;
214. cout << "\n主存分配情况:\n";
215. DuLNode \*p = block\_first->next;
216. cout << "分区号\t起始地址\t分区大小\t状态\n\n";
217. **while** (p)
218. {
219. cout << "  " << flag++ << "\t";
220. cout << "  " << p->data.address << "\t\t";
221. cout << " " << p->data.size << "KB\t\t";
222. **if** (p->data.state == Free) cout << "空闲\n\n";
223. **else** cout << "已分配\n\n";
224. p = p->next;
225. }
226. }

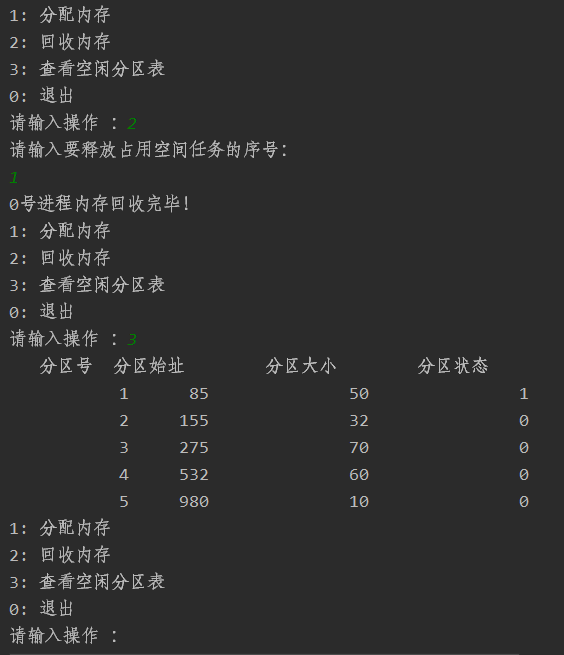
229. **int** main()//主函数
230. {
231. **int** ch;//算法选择标记
232. cout << "请输入所使用的内存分配算法：\n";
233. cout << "(1)首次适应算法\n(2)最佳适应算法\n";
235. cin >> ch;
236. **while** (ch<1 || ch>2)
237. {
238. cout << "输入错误，请重新输入所使用的内存分配算法：\n";
239. cin >> ch;
240. }
242. Initblock(); //开创空间表
243. **int** choice;  //操作选择标记
245. **while** (1)
246. {
247. show();
248. cout << "请输入您的操作：";
249. cout << "\n1: 分配内存\n2: 回收内存\n0: 退出\n";
251. cin >> choice;
252. **if** (choice == 1) Alloc(ch); // 分配内存
253. **else** **if** (choice == 2)  // 内存回收
254. {
255. **int** flag;
256. cout << "请输入您要释放的分区号："<<endl;
257. cin >> flag;
258. free(flag);
259. }
260. **else** **if** (choice == 0) **break**; //退出
261. **else** //输入操作有误
262. {
263. cout << "输入有误，请重试！" << endl;
264. **continue**;
265. }
266. }
267. **return** 0;
268. }

### 调试分析：

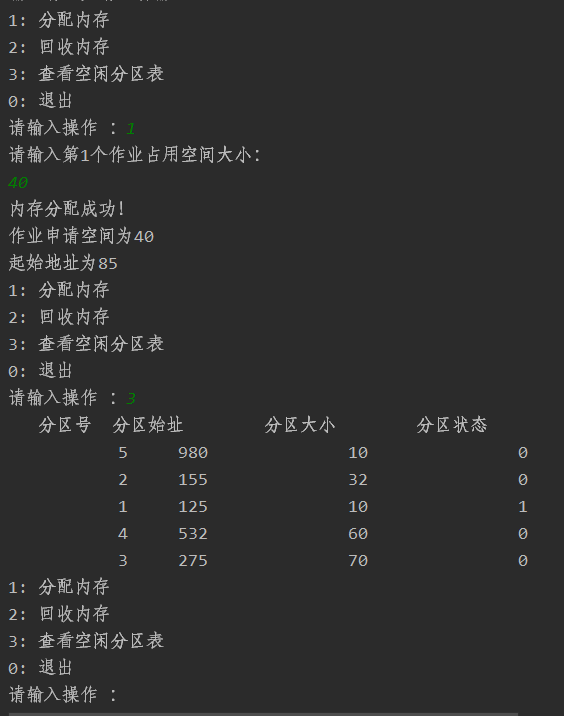
#### 循环首次适应算法(空闲区表法)：



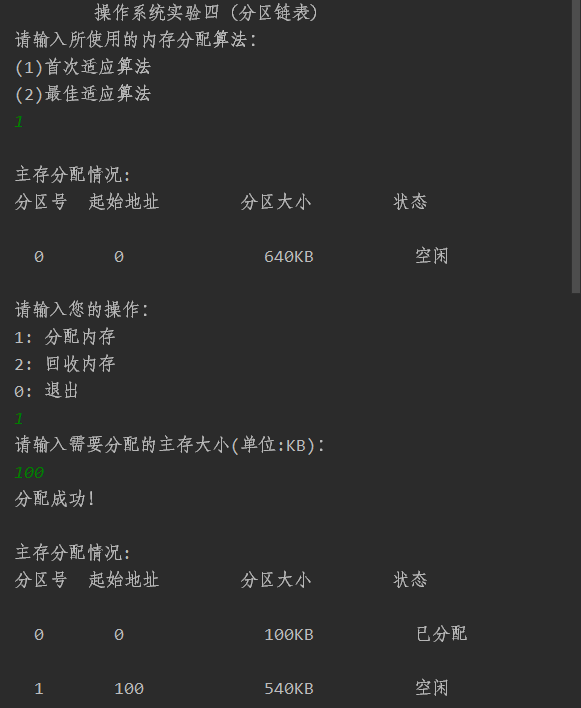


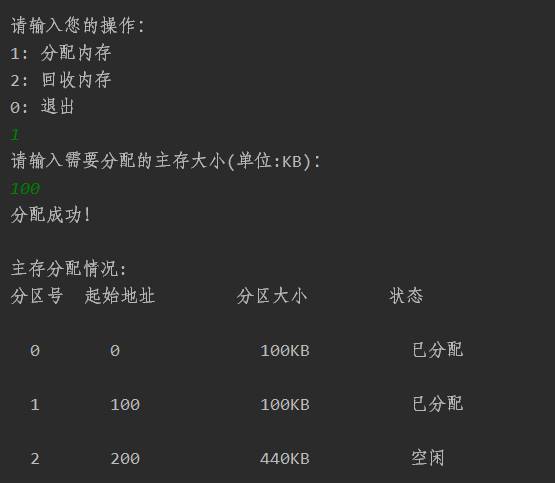


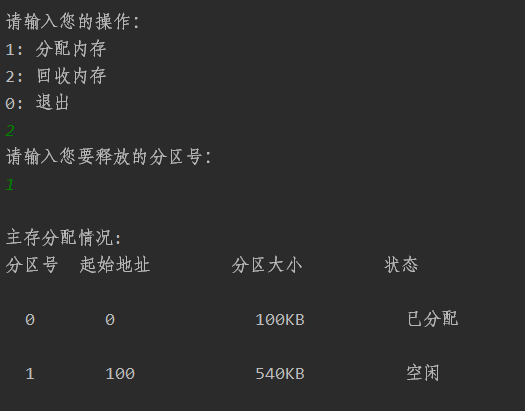
#### 最佳适应算法（空闲区表法）



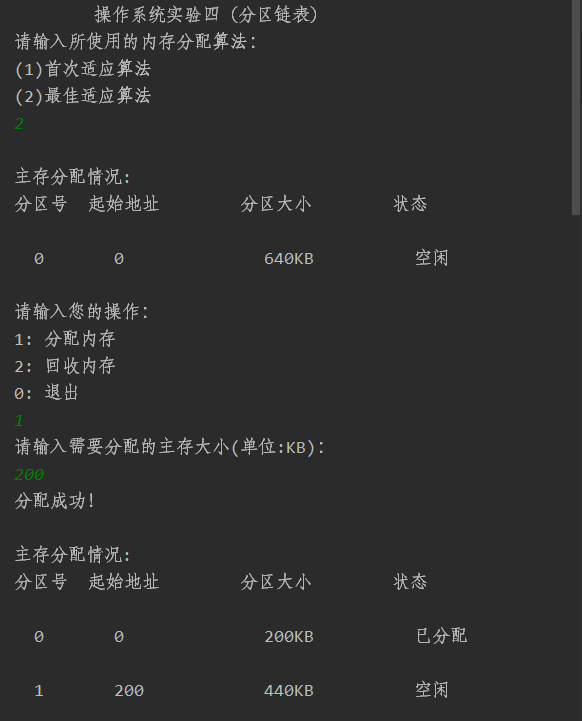
#### 循环首次适应算法（空闲区链法）

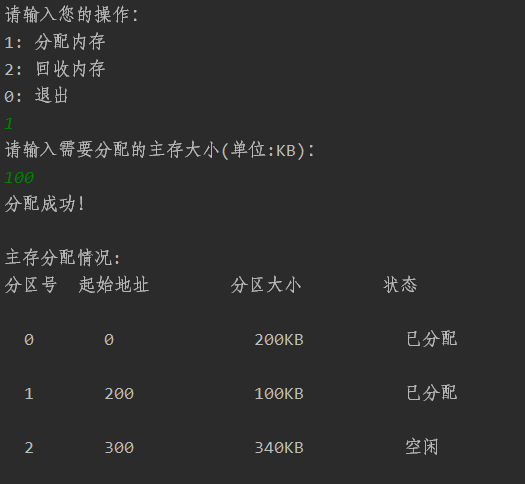


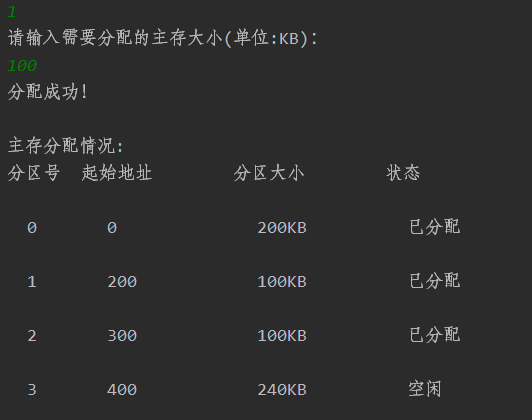


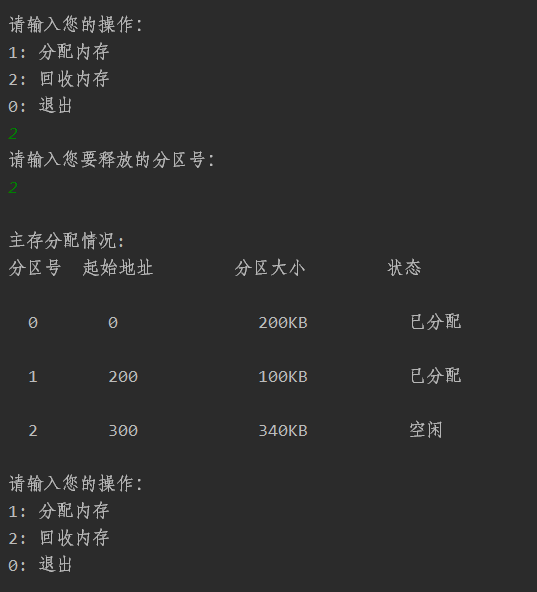


#### 最佳适应算法（空闲区链法）









### 结论和期望：

本次实验实现了储存器动态分区管理的算法，分别使用分区表和分区链表法两个方法，不仅对操作系统的原理有了更深的认识，对于数据结构也有了复习，但是程序没有考虑到很多边界情况但是在查看博客的时候已经将问题改正，希望以后对程序的边界情况能更加考虑清楚。

## 磁盘调度算法

### 需求分析：

本实验是模拟操作系统的磁盘寻道方式，运用磁盘访问顺序的不同来设计磁盘的调度算法。

(1) 实现的磁盘调度算法有 FCFS，SSTF，SCAN，CSCAN 和 NStepSCAN 算法。

(2) 设定开始磁道号寻道范围，依据起始扫描磁道号和最大磁道号数，随机产生要进行寻道的磁道号序列。

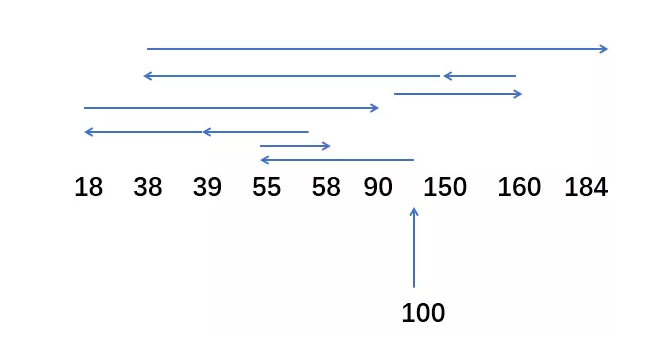
(3) 选择磁盘调度算法，显示该算法的磁道访问顺序，计算出移动的磁道总数和平均寻道总数。

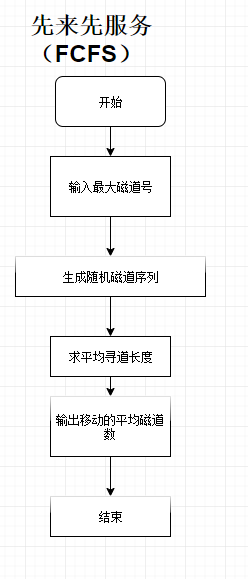
(4) 按算法的寻道效率进行排序，并对各算法的性能进行分析比较。

### 概要设计：

#### 先来先服务算法（FCFS）

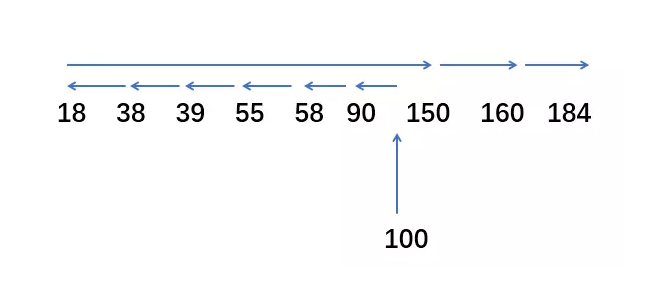
这是一种比较简单的磁盘调度算法。它根据进程请求访问磁盘的先后次序进行调度。此算法的优点是公平、简单，且每个进程的请求都能依次得到处理，不会出现某一进程的请求长期得不到满足的情况。此算法由于未对寻道进行优化，在对磁盘的访问请求比较多的情况下，此算法将降低设备服务的吞吐量，致使平均寻道时间可能较长，但各进程得到服务的响应时间的变化幅度较小。

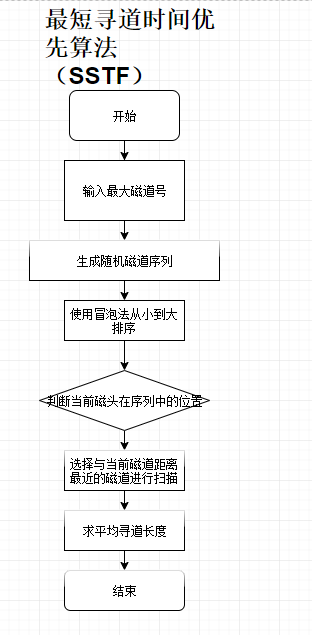




#### 最短寻找时间优先（SSTF）

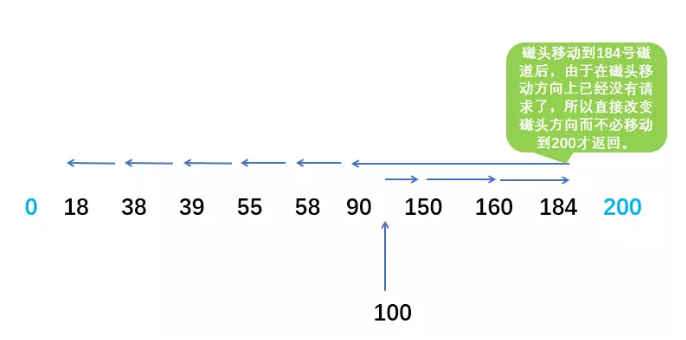
该算法选择这样的进程，其要求访问的磁道与当前磁头所在的磁道距离最近，以使每次的寻道时间最短，该算法可以得到比较好的吞吐量，但却不能保证平均寻道时间最短。其缺点是对用户的服务请求的响应机会不是均等的，因而导致响应时间的变化幅度很大。在服务请求很多的情况下，对内外边缘磁道的请求将会无限期的被延迟，有些请求的响应时间将不可预期。

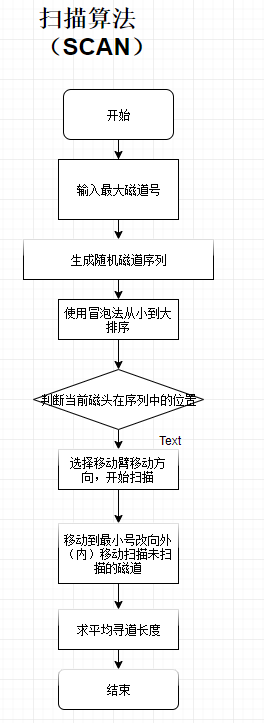




#### 扫描算法（SCAN）

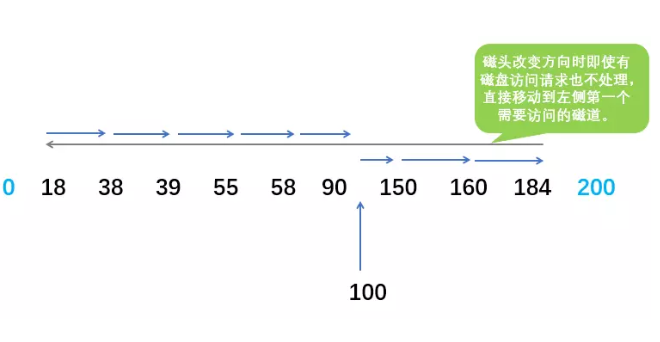
扫描算法不仅考虑到欲访问的磁道与当前磁道的距离，更优先考虑的是磁头的当前移动方向。例如，当磁头正在自里向外移动时，扫描算法所选择的下一个访问对象应是其欲访问的磁道既在当前磁道之外，又是距离最近的。这样自里向外地访问，直到再无更外的磁道需要访问才将磁臂换向，自外向里移动。这时，同样也是每次选择这样的进程来调度，即其要访问的磁道，在当前磁道之内，从而避免了饥饿现象的出现。由于这种算法中磁头移动的规律颇似电梯的运行，故又称为电梯调度算法。此算法基本上克服了最短寻道时间优先算法的服务集中于中间磁道和响应时间变化比较大的缺点，而具有最短寻道时间优先算法的优点即吞吐量较大，平均响应时间较小，但由于是摆动式的扫描方法，两侧磁道被访问的频率仍低于中间磁道。

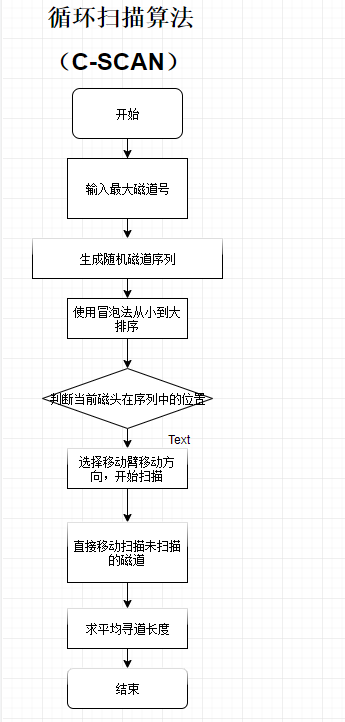




#### 循环扫描算法（C-SCAN）

循环扫描算法是对扫描算法的改进。如果对磁道的访问请求是均匀分布的，当磁头到达磁盘的一端，并反向运动时落在磁头之后的访问请求相对较少。这是由于这些磁道刚被处理，而磁盘另一端的请求密度相当高，且这些访问请求等待的时间较长，为了解决这种情况，循环扫描算法规定磁头单向移动。例如，只自里向外移动，当磁头移到最外的被访问磁道时，磁头立即返回到最里的欲访磁道，即将最小磁道号紧接着最大磁道号构成循环，进行扫描。





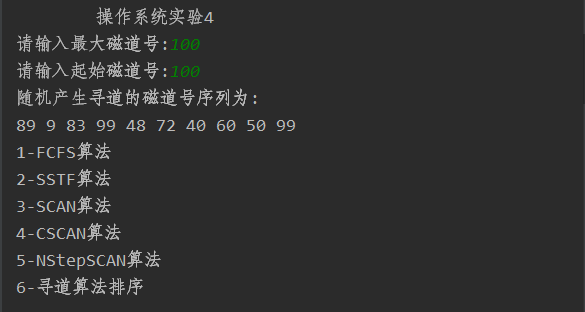
#### NStepSCAN 算法

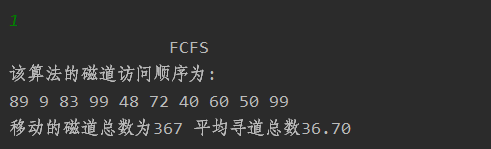
原理：为避免磁臂黏着现象发生，算法将磁盘请求队列分成若干个长度为N的子队列，磁盘调度按FCFS算法依次处理这些子队列。而每处理一个子队列时又是按照SCAN算法。当处理某子队列时，又有新的磁盘I/O请求，便将新请求进程放入其他队列中，从而避免了粘臂现象。（逻辑上依然为SCAN算法，但是限制了队列数量，强制跳出某一柱面，减少了频繁访问同一磁道带来的黏着现象）

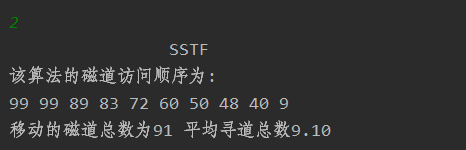
### 详细设计：

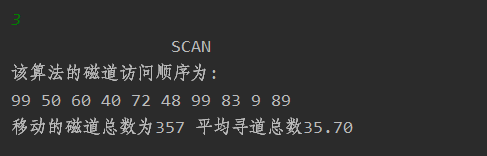
1. //
2. // Created by Misser on 2019/11/20.
3. //
4. /\*\*\*
5. \*  本实验是模拟操作系统的磁盘寻道方式，运用磁盘访问顺序的不同来设计磁盘的调度
6. 算法。
7. (1) 实现的磁盘调度算法有 FCFS，SSTF，SCAN，CSCAN 和 NStepSCAN 算法。
8. (2) 设定开始磁道号寻道范围，依据起始扫描磁道号和最大磁道号数，随机产生要进行
9. 寻道的磁道号序列。
10. (3) 选择磁盘调度算法，显示该算法的磁道访问顺序，计算出移动的磁道总数和平均寻
11. 道总数。
12. (4) 按算法的寻道效率进行排序，并对各算法的性能进行分析比较。
13. \*/
14. #include <bits/stdc++.h>
15. **using** **namespace** std;
16. **const** **int** num=10;//磁道号序列
17. **int** mx,start,disk[num],order[num],a,A,b,cnt;
18. **double** xl;
19. /\*\*\*
20. \* 数组逆序
21. \* @param l
22. \* @param r
23. \* @param array
24. \*/
25. **void** Array\_reverse(**int** l,**int** r,**int**\* array){
26. **double** mid=(l+r)/2.0;
27. **for**(**int** i=l;i<mid;++i){
28. array[i]=array[r+l-i]+array[i];
29. array[r+l-i]=array[i]-array[r+l-i];
30. array[i]=array[i]-array[r+l-i];
31. }
32. }
33. /\*\*\*
34. \* 先来先服务算法
35. \* @param st
36. \*/
37. **void** FCFS(**int** st){
38. a=0;
39. **for**(**int** i=0;i<num;i++) order[i]=disk[i];//按先来先服务的策略输出磁盘请求序列
40. **for**(**int** i=0;i<num;i++){
41. a+=abs(st-disk[i]);
42. st=disk[i];
43. }
44. xl=(**double**)a\*1.0/num;
45. }
46. /\*\*\*
47. \* 最短寻道时间算法
48. \* @param st
49. \*/
50. **void** SSTF(**int** st){
51. a=0;
52. **int** d[num];
53. **for**(**int** i=0;i<num;i++) d[i]=disk[i];
54. **for**(**int** i=0;i<num;i++){
55. sort(d+i, d + num,
56. [st](**const** **int** &first, **const** **int** &second)
57. { **return** (abs(first - st) > abs(second - st)) ? **false** : **true**;});
58. //若当前磁道号大于请求序列中最大者，则直接由外向内依次给予各请求服务
59. //若当前磁道号小于请求序列中最大者，则直接由内向外依次给予各请求服务
60. order[i]=d[i];
61. a+=abs(st-d[i]);
62. st=d[i];
63. }
64. xl=(**double**)a\*1.0/num;
65. }
66. /\*\*\*
67. \* 电梯调度算法
68. \* @param st
69. \*/
70. **void** SCAN(**int** st){
71. a=0;
72. **int** d[num];
73. **int** min=num-1;
74. **for**(**int** i=0;i<num;i++) d[i]=disk[i];
75. **for**(**int** i=0;i<num;i++){
76. **if**(d[i]-st>0){
77. min=i;**break**;
78. }
79. }
80. Array\_reverse(min,num-1,d);//若当前磁道号大于请求序列中最大者，则直接将移动臂移动到最小号磁道依次向外给予各请求服务
81. Array\_reverse(0,num-1,d);//若当前磁道号小于请求序列中最小者，则直接由内向外依次给予各请求服务,此情况同最短寻道优先
82. **for**(**int** i=0;i<num;++i){
83. a+=abs(st-d[i]);
84. st=d[i];
85. order[i]=d[i];
86. }
87. xl=(**double**)a\*1.0/num;
88. }
89. /\*\*\*
90. \* 循环扫描算法
91. \* @param st
92. \*/
93. **void** CSCAN(**int** st){
94. **int** min=num-1,d[num];
95. a=0;
96. **for**(**int** i=0;i<num;i++) d[i]=disk[i];
97. **for**(**int** i=0;i<num;++i){
98. **if**(d[i]-st>0){
99. min=i;
100. **break**;
101. }
102. }
103. Array\_reverse(min,num-1,d); //若当前磁道号大于请求序列中最大者，则直接将移动臂移动到最小号磁道依次向外给予各请求服务
104. Array\_reverse(0,min-1,d); //若当前磁道号小于请求序列中最小者，则直接由内向外依次给予各请求服务,此情况同最短寻道优先
105. Array\_reverse(0,num-1,d);
107. **for**(**int** i=0;i<num;++i){
108. a+=abs(st-d[i]);
109. st=d[i];
110. order[i]=d[i];
111. }
112. xl=(**double**)a\*1.0/num;
113. }
114. **void** SCAN\_CSCAN(**int** st,**int**\*d,**int** num){
115. **int** min=num-1,a=0;
116. **for**(**int** i=0;i<num;++i){
117. **if**(d[i]-st>0){
118. min=i;
119. **break**;
120. }
121. }
122. Array\_reverse(min,num-1,d);
123. Array\_reverse(0,min-1,d);
124. Array\_reverse(0,num-1,d);
125. **for**(**int** i=0;i<num;++i){
126. order[cnt++]=d[i];
127. a+=abs(st-d[i]);
128. st=d[i];
129. }
130. \*order=a;
131. }
132. /\*\*\*
133. \* 分步电梯调度算法
134. \* @param st
135. \*/
136. **void** NStepSCAN(**int** st){
137. printf("请输入NStepSCAN算法队列的长度N:\n");
138. **int** N,d[num];
139. a=0; cnt=0;
140. **for**(**int** i=0;i<num;i++) d[i]=disk[i];
141. cin>>N;
142. **if**(num%N==0) A=N,b=N;
143. **else** A=N,b=num%N;
144. **for**(**int** i=0;i!=ceil(num/(**double**)N);++i){
145. **if**(i==ceil(num/N)) SCAN\_CSCAN(st,d+A\*i,b);
146. **else** SCAN\_CSCAN(st,d+A\*i,A);
147. a+=order[0];
148. }
149. xl=(**double**)a\*1.0/num;
150. }
151. **struct** node{
152. **int** id;
153. **double** val;
154. }s[10];
155. **bool** cmp(node a,node b){
156. **return** a.val<b.val;
157. }
158. **void** Sort\_Case(){
159. FCFS(start);
160. s[0].val=xl; s[0].id=0;
161. SSTF(start);
162. s[1].val=xl; s[1].id=1;
163. SCAN(start);
164. s[2].val=xl; s[2].id=2;
165. CSCAN(start);
166. s[3].val=xl; s[3].id=3;
167. NStepSCAN(start);
168. s[4].val=xl; s[4].id=4;
169. sort(s,s+5,cmp);
170. printf("各算法按寻道效率进行排序结果为 :\n");
171. **for**(**int** i=0;i<5;i++){
172. **if**(s[i].id==0) printf("FCFS算法平均寻道总数为 ：");
173. **if**(s[i].id==1) printf("SSTF算法平均寻道总数为 ：");
174. **if**(s[i].id==2) printf("SCAN算法平均寻道总数为 ：");
175. **if**(s[i].id==3) printf("CSCAN算法平均寻道总数为 ：");
176. **if**(s[i].id==4) printf("NStepSCAN算法平均寻道总数为 ：");
177. printf("%.2f\n",s[i].val);
178. }
179. }
180. **int** main(){
181. printf("\t操作系统实验4\n");
182. printf("请输入最大磁道号:");
183. scanf("%d",&mx);
184. printf("请输入起始磁道号:");
185. scanf("%d",&start);
186. srand(time(0));
187. printf("随机产生寻道的磁道号序列为: \n");
188. **for**(**int** i=0;i<num;i++){
189. disk[i]=rand()%mx;
190. printf("%d ",disk[i]);
191. }
192. puts("");
193. **int** IsContinue=1;
194. **while**(IsContinue){
195. printf("1-FCFS算法\t\n");
196. printf("2-SSTF算法\t\n");
197. printf("3-SCAN算法\t\n");
198. printf("4-CSCAN算法\t\n");
199. printf("5-NStepSCAN算法\t\n");
200. printf("6-寻道算法排序\t\n");
201. **int** Case;
202. printf("\n");
203. scanf("%d",&Case);
204. **switch**(Case){
205. **case** 1:
206. FCFS(start);
207. printf("\t\tFCFS\n");
208. printf("该算法的磁道访问顺序为: \n") ;
209. **for**(**int** i=0;i<num;i++) printf("%d ",order[i]);
210. puts("");
211. printf("移动的磁道总数为%d 平均寻道总数%.2lf\n",a,(**double**)a\*1.0/num);
212. **break**;
213. **case** 2:
214. SSTF(start);
215. printf("\t\tSSTF\n");
216. printf("该算法的磁道访问顺序为: \n") ;
217. **for**(**int** i=0;i<num;i++){
218. printf("%d ",order[i]);
219. }
220. puts("");
221. printf("移动的磁道总数为%d 平均寻道总数%.2lf\n",a,(**double**)a\*1.0/num);
222. **break**;
223. **case** 3:
224. SCAN(start);
225. printf("\t\tSCAN\n");
226. printf("该算法的磁道访问顺序为: \n") ;
227. **for**(**int** i=0;i<num;i++) printf("%d ",order[i]);
228. puts("");
229. printf("移动的磁道总数为%d 平均寻道总数%.2lf\n",a,(**double**)a\*1.0/num);
230. **break**;
231. **case** 4:
232. CSCAN(start);
233. printf("\t\tCSCAN\n");
234. printf("该算法的磁道访问顺序为: \n") ;
235. **for**(**int** i=0;i<num;i++) printf("%d ",order[i]);
236. puts("");
237. printf("移动的磁道总数为%d 平均寻道总数%.2lf\n",a,(**double**)a\*1.0/num);
238. **break**;
239. **case** 5:
240. printf("\t\tN-StepSCAN\n");
241. NStepSCAN(start);
242. printf("%d %d\n",A,b);
243. printf("该算法的磁道访问顺序为: \n") ;
244. **for**(**int** i=0;i<cnt;i++) printf("%d ",order[i]);
245. puts("");
246. printf("移动的磁道总数为%d 平均寻道总数%.2lf\n",a,(**double**)a\*1.0/num);
247. **break**;
248. **case** 6:
249. Sort\_Case();
250. **break**;
251. **default**:
252. printf("\t警告：请输入正确的选择！\n");
253. **break**;
254. }
255. printf("\n");
256. printf("\t是否继续选择算法?\t\n");
257. printf("\t0代表退出\t\n");
258. printf("\t1代表继续\t\n");
259. scanf("%d",&IsContinue);
260. }
261. **return** 0;
262. }

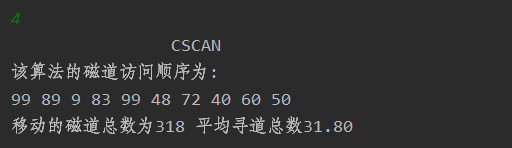
### 调试设计：

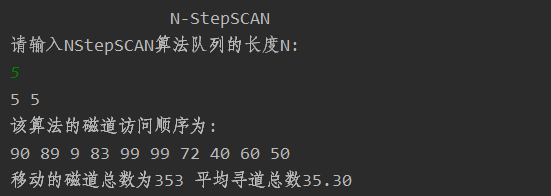


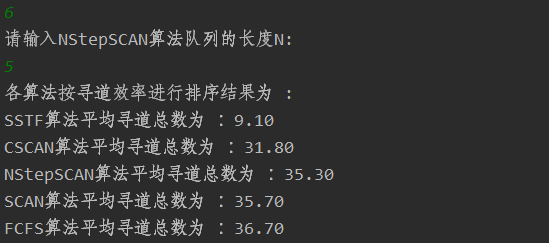












### 结论和期望：

本实验完成磁盘调度算法，采用分模块式编写函数的方式，比之前的程序的可扩展性较强，一直不是很理解NStepSCAN的调度算法，但是其实就是在SCAN算法的基础上增加了多个队列。还是要多复习操作系统的原理，对于原理掌握清楚了，再来编写程序也就很轻松了。

# 三、实验总结：

…