银行家算法

1. 实验目的：

旨在通过模拟死锁的产生和应用银行家算法来防止死锁，帮助加深对死锁及其防止机制的理解。死锁是操作系统中的一个重要问题，银行家算法（Banker's Algorithm）是解决死锁问题的一种有效方法，通常用于确保系统的资源分配安全，防止死锁的发生。

1. 实验内容：

要设计一个模拟系统，其中 n 个进程共享 m 个资源，且进程可以动态地申请和释放资源，同时系统会根据各进程的申请动态地分配资源，并展示资源的分配过程。

题目提炼：

**问题描述：**我们有一个包含n个进程和m类资源的计算机系统，进程和资源的配置如下：

1、可利用资源向量 Available：表示每类资源当前的可用数量。

2、最大需求矩阵 Max：表示系统中每个进程对每类资源的最大需求。

3、分配矩阵 Allocation：表示系统中各类资源当前分配到每个进程的数量。

4、需求矩阵 Need：表示每个进程当前还需要多少资源才能完成任务，计算公式为：Need[i][j] = Max[i][j] - Allocation[i][j]

**银行家算法：** 进程可以动态地请求资源，系统在每次请求时按照以下步骤判断是否可以安全地分配资源：

**1、请求合法性检查**：

进程的请求是否不超过其最大需求：Request[i][j] ≤ Need[i][j]。

请求的资源是否不超过当前可用资源：Request[i][j] ≤ Available[j]。

**2、资源分配模拟**：

如果合法，试探性地分配资源并更新系统状态。

计算分配后的资源是否处于安全状态。安全状态通过安全性算法验证。

**3、安全性算法**：

使用 Work 向量表示当前可用资源，初始值为 Available。

使用 Finish 数组表示每个进程是否能够完成，初始值为 false。

安全性算法通过遍历进程，判断是否有进程的需求可以被当前资源满足。若能满足，假设该进程完成并释放资源，继续检查下一个进程。

如果所有进程都能顺利完成，则系统处于安全状态，资源请求被批准；如果不能，则系统处于不安全状态，回滚资源请求。

**目标：**

1、实现银行家算法，接受用户输入的进程和资源配置。

2、根据每个进程的资源请求，判断请求是否合理，并输出系统的安全性（是否处于安全状态）。

3、输出安全序列（如果系统处于安全状态）。

**输入：**

1、系统中进程数 n 和资源种类数 m。

2、可用资源向量 Available，最大需求矩阵 Max，以及分配矩阵 Allocation。

3、进程的资源请求向量 Request。

**输出：**

1、判断当前资源配置是否安全。

2、输出系统是否处于安全状态，并给出安全序列（如果存在）。

3、如果请求不安全，输出拒绝信息。

**算法步骤：**

1、初始化：输入 n、m、Available、Max、Allocation，并计算需求矩阵 Need。

2、请求处理：对于每个进程的资源请求，首先检查请求是否合法（是否不超过需求且不超过可用资源），如果不合法，直接输出错误信息。

3、模拟资源分配：如果请求合法，模拟资源分配并执行安全性检查。

4、安全性检查：使用安全性算法判断当前资源配置是否安全，并给出安全序列。

代码：

1. #include <stdio.h>
2. #include <stdbool.h>
4. #define MAX\_PROCESSES 10  // 最大进程数
5. #define MAX\_RESOURCES 5   // 最大资源类型数
7. **int** Max[MAX\_PROCESSES][MAX\_RESOURCES];        // 最大需求矩阵
8. **int** Allocation[MAX\_PROCESSES][MAX\_RESOURCES]; // 已分配矩阵
9. **int** Need[MAX\_PROCESSES][MAX\_RESOURCES];       // 需求矩阵
10. **int** Available[MAX\_RESOURCES];                 // 可用资源
12. **int** N, M;  // N为进程数，M为资源种类数
14. // 计算需求矩阵 Need
15. **void** calculate\_need() {
16. **for** (**int** i = 0; i < N; i++) {
17. **for** (**int** j = 0; j < M; j++) {
18. Need[i][j] = Max[i][j] - Allocation[i][j];
19. }
20. }
21. }
23. // 安全性算法，检查是否存在安全序列
24. **bool** is\_safe(**int** safe\_sequence[]) {
25. **int** Work[M];  // 临时工作资源
26. **bool** Finish[N];  // 记录进程是否完成
27. **int** count = 0;
29. // 初始化工作资源 = 可用资源
30. **for** (**int** i = 0; i < M; i++) {
31. Work[i] = Available[i];
32. }
34. // 初始化Finish为false
35. **for** (**int** i = 0; i < N; i++) {
36. Finish[i] = **false**;
37. }
39. **while** (count < N) {
40. **bool** found = **false**;
41. **for** (**int** i = 0; i < N; i++) {
42. // 如果进程i没有完成且其需求能被当前工作资源满足
43. **if** (!Finish[i]) {
44. **bool** can\_allocate = **true**;
45. **for** (**int** j = 0; j < M; j++) {
46. **if** (Need[i][j] > Work[j]) {
47. can\_allocate = **false**;
48. **break**;
49. }
50. }
52. // 如果能分配
53. **if** (can\_allocate) {
54. // 假设分配资源
55. **for** (**int** j = 0; j < M; j++) {
56. Work[j] += Allocation[i][j];
57. }
58. Finish[i] = **true**;  // 进程i完成
59. safe\_sequence[count++] = i;  // 记录安全序列
60. found = **true**;
61. **break**;
62. }
63. }
64. }
66. **if** (!found) {
67. // 如果没有进程能够满足，则系统不安全
68. **return** **false**;
69. }
70. }
72. // 如果所有进程的 Finish 都为 true，则系统安全
73. **return** **true**;
74. }
76. // 打印当前系统状态
77. **void** print\_system\_state() {
78. printf("\n当前系统状态：\n");
80. printf("Available: ");
81. **for** (**int** i = 0; i < M; i++) {
82. printf("%d ", Available[i]);
83. }
84. printf("\n");
86. printf("Max矩阵:\n");
87. **for** (**int** i = 0; i < N; i++) {
88. **for** (**int** j = 0; j < M; j++) {
89. printf("%d ", Max[i][j]);
90. }
91. printf("\n");
92. }
94. printf("Allocation矩阵:\n");
95. **for** (**int** i = 0; i < N; i++) {
96. **for** (**int** j = 0; j < M; j++) {
97. printf("%d ", Allocation[i][j]);
98. }
99. printf("\n");
100. }
102. printf("Need矩阵:\n");
103. **for** (**int** i = 0; i < N; i++) {
104. **for** (**int** j = 0; j < M; j++) {
105. printf("%d ", Need[i][j]);
106. }
107. printf("\n");
108. }
109. }
111. // 申请资源
112. **bool** request\_resources(**int** process\_id, **int** request[MAX\_RESOURCES]) {
113. // 检查请求是否小于等于需求
114. **for** (**int** i = 0; i < M; i++) {
115. **if** (request[i] > Need[process\_id][i]) {
116. printf("错误：进程 %d 请求的资源超过了最大需求！\n", process\_id);
117. **return** **false**;
118. }
119. }
121. // 检查请求是否小于等于可用资源
122. **for** (**int** i = 0; i < M; i++) {
123. **if** (request[i] > Available[i]) {
124. printf("错误：进程 %d 请求的资源不足！\n", process\_id);
125. **return** **false**;
126. }
127. }
129. // 临时分配资源并更新矩阵
130. **for** (**int** i = 0; i < M; i++) {
131. Available[i] -= request[i];
132. Allocation[process\_id][i] += request[i];
133. Need[process\_id][i] -= request[i];
134. }
136. // 安全性检查
137. **int** safe\_sequence[N];  // 安全序列数组
138. **if** (is\_safe(safe\_sequence)) {
139. printf("进程 %d 的资源请求被批准！\n", process\_id);
140. printf("安全序列: ");
141. **for** (**int** i = 0; i < N; i++) {
142. printf("P%d ", safe\_sequence[i]);
143. }
144. printf("\n");
145. **return** **true**;
146. } **else** {
147. // 如果不安全，则回滚分配
148. **for** (**int** i = 0; i < M; i++) {
149. Available[i] += request[i];
150. Allocation[process\_id][i] -= request[i];
151. Need[process\_id][i] += request[i];
152. }
153. printf("进程 %d 的资源请求被拒绝，系统不安全！\n", process\_id);
154. **return** **false**;
155. }
156. }
158. **int** main() {
159. // 输入进程数和资源种类数
160. printf("请输入进程数 N 和资源种类数 M：");
161. scanf("%d %d", &N, &M);
163. // 输入可用资源
164. printf("请输入系统中各类资源的可用数量：\n");
165. **for** (**int** i = 0; i < M; i++) {
166. scanf("%d", &Available[i]);
167. }
169. // 输入每个进程的最大资源需求
170. printf("请输入每个进程的最大资源需求（Max矩阵）：\n");
171. **for** (**int** i = 0; i < N; i++) {
172. **for** (**int** j = 0; j < M; j++) {
173. scanf("%d", &Max[i][j]);
174. }
175. }
177. // 输入每个进程已分配的资源
178. printf("请输入每个进程已分配的资源（Allocation矩阵）：\n");
179. **for** (**int** i = 0; i < N; i++) {
180. **for** (**int** j = 0; j < M; j++) {
181. scanf("%d", &Allocation[i][j]);
182. }
183. }
185. // 计算需求矩阵 Need
186. calculate\_need();
188. // 打印初始系统状态
189. print\_system\_state();
191. // 假设进程 P1 请求资源
192. **int** request[M];
193. printf("\n请输入进程 P1 的资源请求：\n");
194. **for** (**int** i = 0; i < M; i++) {
195. scanf("%d", &request[i]);
196. }
198. // 申请资源并判断是否安全
199. request\_resources(1, request);
201. // 打印系统状态
202. print\_system\_state();
204. **return** 0;
205. }

运行结果：



