****

计算机操作系统实验3

银行家算法的实现

姓名： 涂煜洋

学号： 71118119

东南大学计算机科学与工程学院、软件学院

School of Computer Science & Engineering

College of Software Engineering

Southeast University

2020年5月

1. **实验目的及内容**
2. **实验目的:**

通过实验，加深对多实例资源分配系统中死锁避免方法——银行家算法的理解，掌握Windows 环境下银行家算法的实现方法，同时巩固利用Windows API 进行共享数据互斥访问和多线程编程的方法。

1. **实验内容：**

1. 在Windows 操作系统上，利用Win32 API 编写多线程应用程序实现银行家算法。

2. 创建 n 个线程来申请或释放资源，只有保证系统安全，才会批准资源申请。

3. 通过Win32 API 提供的信号量机制，实现共享数据的并发访问。

**二、实验基本知识和步骤**

**1.银行家算法：**

**1.1 银行家算法的数据结构：**

为了实现银行家算法，即管理进程对于资源的要求与分配，我们需要几个数据结构对资源分配系统的状态进行分配。使用表示系统进程个数，表示资源的种类。

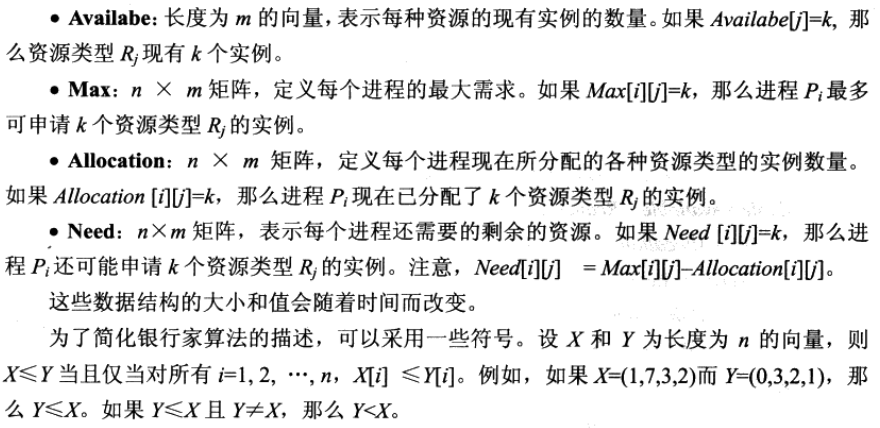


图1.1-1 银行家算法系统中的数据结构

在银行家算法中，还需要一些为进程所拥有的数据结构用来表示进程对于资源的请求。包括，其中前者用于表示进程请求的资源信息，后者用于表示进程释放的资源信息。

**1.2 银行家算法的核心思想：**

银行家算法的核心思想包括两个部分，一个是安全性检测，另一个是资源请求，银行家算法依据安全性检测的结果决定是否向检测分配其请求的资源。



图1.2-1 银行家算法

**2.银行家算法的实现：**

**2.1 银行家算法的辅助函数以及数据结构实现：**

由于银行家算法的实现涉及一部分对于向量的运算，并且在测试过程中对于系统的状态有一定的表示需求，因此我编写了一系列相关的对数据结构的运算函数，其中算法中的向量以及矩阵则选取了C++标准库中的类予以实现。辅助函数的声明如下图表示：

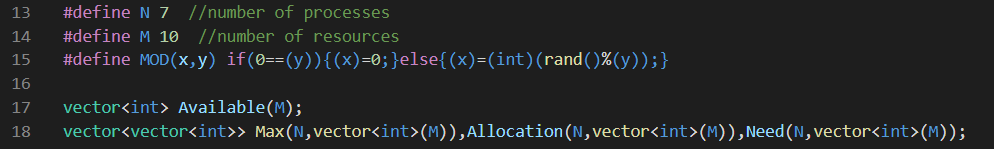


图2.1-1 数据结构的实现

由上图可见，在银行家算法实现的过程中，我使用宏方式定义算法中的进程数量以及资源种类数量。

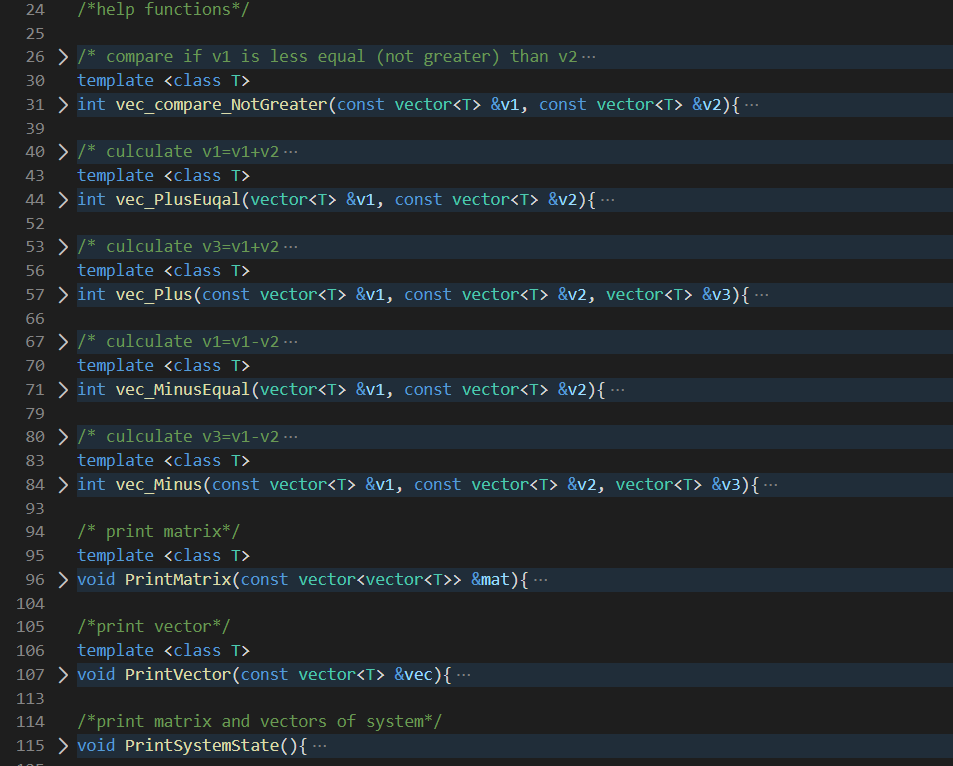
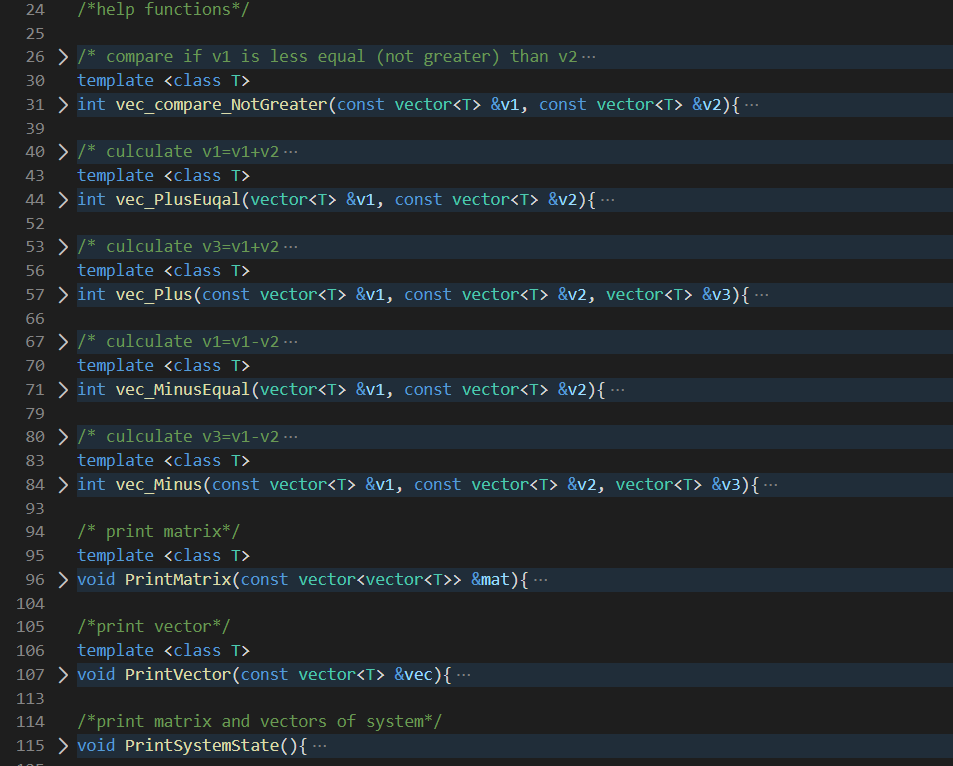


图2.1-2 银行家算法中的辅助函数

银行家算法中的辅助函数包含对向量的大小比较以及向量之间的加减运算，并且处于测试需求还实现了对于向量和矩阵的现实实现。相关实现详见附录中的源代码。

**2.2 安全性算法实现：**

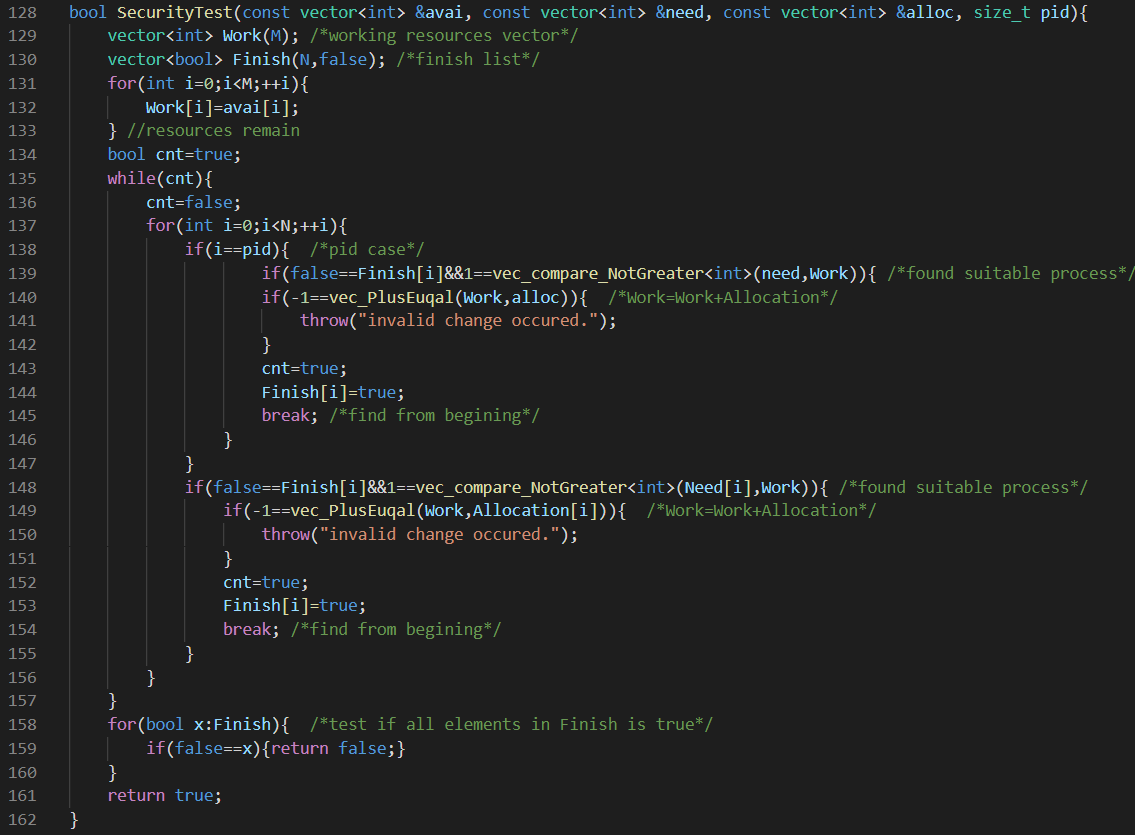


图2.2-1安全性检测算法实现

出于算法效率考虑，安全性算法并没有直接使用公共的数据结构进行检测，而使用了的副本进行判断。并且处于节约内存的考虑，基于安全性检测算法由进程调用，并且对于公共数据结构的改变只限于数据结构中表示该进程资源分配状态的部分，因此使用的副本仅矩阵中表示该进程的子向量，因而在查找满足回收条件的进程过程中将属于当前进程状态的行向量（行）单独进行处理（算法138-147行）。

的实现基本依照普通的安全性算法的结构，实现假设分配给进程请求的资源后系统是否依然处于安全状态的检测。

**2.3 资源请求与释放算法实现：**

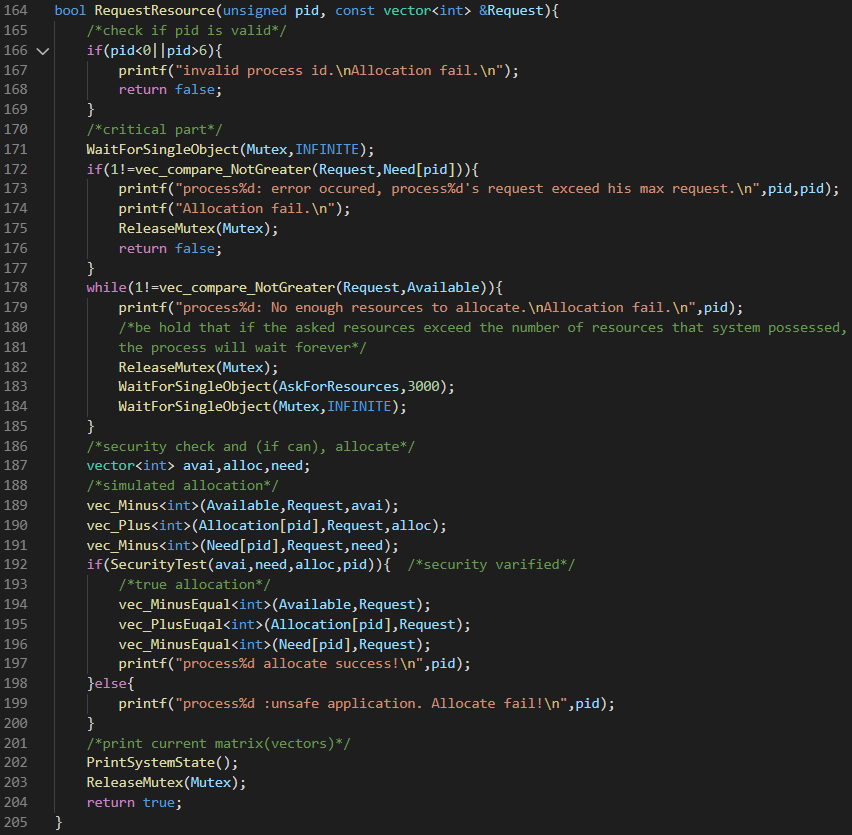


图2.3-1资源请求算法的实现

的结构基本按照普通的资源请求算法实现，并且处于线程安全的考虑，将访问公共数据结构的部分（170-203行）使用信号量（线程互斥锁）包裹起来，防止访问过程中被其他线程干扰导致错误。

值得注意的是当进程所请求的资源数量当前数量大于当前系统所拥有的资源数量时，进程（在此模拟中是线程）将被挂起（），并且在挂起前释放先前拥有的互斥锁，等待被其他进程释放资源的信号。在进程被唤起后将重获互斥锁并重新检测系统是否拥有足够的资源。注意在本系统中，如果进程请求的资源数大于进程拥有的最大资源数，他将会在其生命周期中不断重复“挂起--唤醒”的过程。（但在本系统的初始化条件下，这种情况并不会发生）

当系统判断接收进程的申请的安全的时，将会真正为进程分配资源，并显示分配成功的语句（192-198行）。否则，显示分配失败的语句。（198-200行）

代码202行的函数用于打印出系统的相关信息，主要为矩阵的数据，是可选的测试功能。

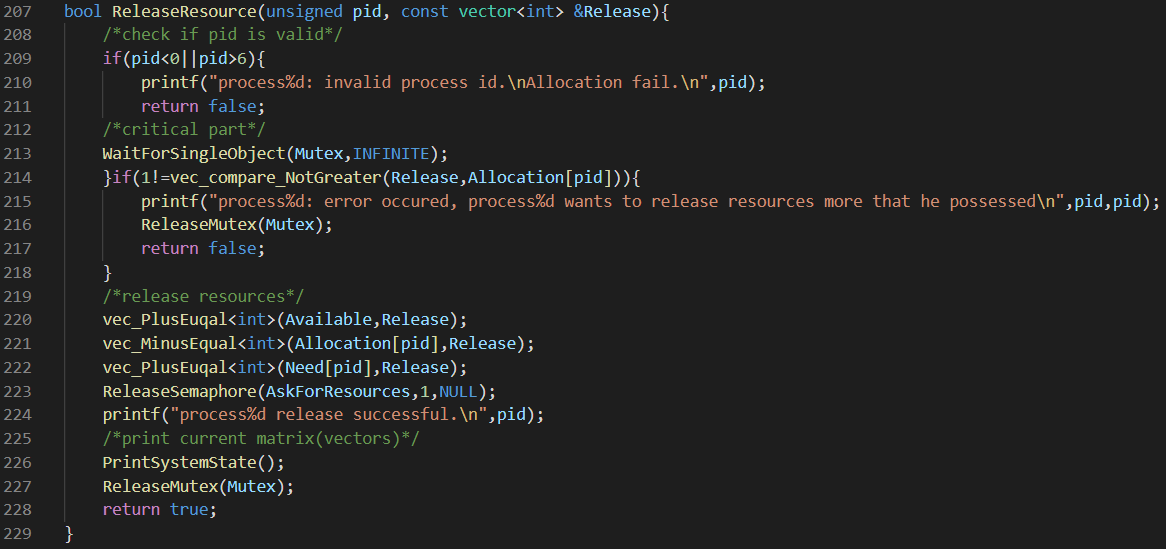


图2.3-2 进程资源释放的实现

进程释放资源的实现相对要简单得多，代码214-218行主要组织进程释放超过其拥有的资源（在本系统中不会发生），并且不会对其他的释放请求进行阻止。同样的，出于线程安全的考虑，要求代码中访问到公共数据结构的部分使用线程互斥锁包裹，并且在资源释放完毕时将信号量加一，此举可能将一个进程从挂起状态中唤醒。

**2.4 进程模拟函数的实现：**



图2.4-1 进程模拟函数

进程模拟函数拥有两个数据结构，分别用来表示其申请资源和释放资源的信息。在循环模拟过程中，线程先进行一段睡眠，之后尝试一个随机数并根据其奇偶性选择模拟申请或释放操作，并根据选择生产或向量。向量的生成使用随机数取模获得，这保证了进程请求（释放）的资源数不会超过其所需（所拥有）的资源数。

在实现的过程中我发现当或为0时，对应资源数为0时，将会产生除0异常，因此我对取模运算进行宏的封装以处理这种错误。



图2.4-2 取模运算的封装

**2.5 函数设计：**



图2.5-1 函数设计

函数主要完成了一些数据的初始化和创建线程的工作，因此在此处不再过多赘述。

**三、算法测试**

**1.结果测试**

在测试过程可以选择在运行过程中显示或不显示系统信息，下面将两种运行的部分显示以图的形式进行展现：

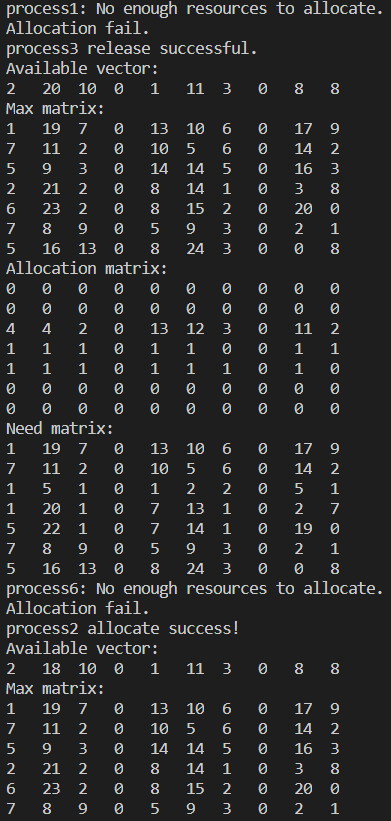
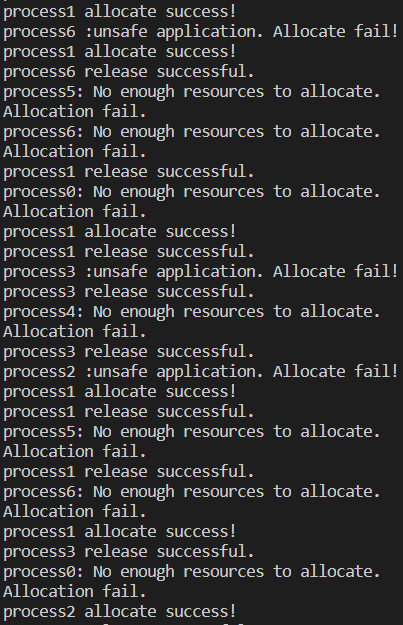


图1-1 银行家算法测试

**四、实验小结**

**附源代码：**

1. #include<vector>
2. #include<stdio.h>
3. #include<random>
4. #include<time.h>
5. #include<iostream>
6. #include<windows.h>
7. #include<iomanip>
9. **using** std::vector;
10. **using** std::cin;
11. **using** std::cout;
13. #define N 7  //number of processes
14. #define M 10  //number of resources
15. #define MOD(x,y) if(0==(y)){(x)=0;}else{(x)=(int)(rand()%(y));}
17. vector<**int**> Available(M);
18. vector<vector<**int**>> Max(N,vector<**int**>(M)),Allocation(N,vector<**int**>(M)),Need(N,vector<**int**>(M));
20. unsigned my\_pid=0;
21. **HANDLE** Mutex;
22. **HANDLE** AskForResources;
24. /\*help functions\*/
26. /\* compare if v1 is less equal (not greater) than v2
27. \*  @return: 1 for (v1<v2),
28. \*          -1 for incomparable
29. \*           0 for other condition \*/
30. **template** <**class** T>
31. **int** vec\_compare\_NotGreater(**const** vector<T> &v1, **const** vector<T> &v2){
32. **if**(v1.size()!=v2.size()){ **return** -1;}  //different size
33. **size\_t** vec\_size=v1.size();
34. **for**(**size\_t** i=0;i<vec\_size;++i){
35. **if**(v1[i]>v2[i]){ **return** 0;}  /\*false if v1[i]>v2[i]\*/
36. }
37. **return** 1;
38. }
40. /\* culculate v1=v1+v2
41. \*  @return -1: different size of v1 and v2
42. \*          0: normal return \*/
43. **template** <**class** T>
44. **int** vec\_PlusEuqal(vector<T> &v1, **const** vector<T> &v2){
45. **if**(v1.size()!=v2.size()){ **return** -1;}  /\*incompatable\*/
46. **size\_t** vec\_size=v1.size();
47. **for**(**size\_t** i=0;i<vec\_size;++i){  /\*chg all elements in v1\*/
48. v1[i]+=v2[i];
49. }
50. **return** 0;
51. }
53. /\* culculate v3=v1+v2
54. \*  @return -1: different size of v1 and v2
55. \*          0: normal return \*/
56. **template** <**class** T>
57. **int** vec\_Plus(**const** vector<T> &v1, **const** vector<T> &v2, vector<T> &v3){
58. **if**(v1.size()!=v2.size()){ **return** -1;}  /\*incompatable\*/
59. **size\_t** vec\_size=v1.size();
60. v3.resize(vec\_size);
61. **for**(**size\_t** i=0;i<vec\_size;++i){  /\*chg all elements in v1\*/
62. v3[i]=v1[i]+v2[i];
63. }
64. **return** 0;
65. }
67. /\* culculate v1=v1-v2
68. \*  @return -1: different size of v1 and v2
69. \*          0: normal return \*/
70. **template** <**class** T>
71. **int** vec\_MinusEqual(vector<T> &v1, **const** vector<T> &v2){
72. **if**(v1.size()!=v2.size()){ **return** -1;}  /\*incompatable\*/
73. **size\_t** vec\_size=v1.size();
74. **for**(**size\_t** i=0;i<vec\_size;++i){  /\*chg all elements in v1\*/
75. v1[i]-=v2[i];
76. }
77. **return** 0;
78. }
80. /\* culculate v3=v1-v2
81. \*  @return -1: different size of v1 and v2
82. \*          0: normal return \*/
83. **template** <**class** T>
84. **int** vec\_Minus(**const** vector<T> &v1, **const** vector<T> &v2, vector<T> &v3){
85. **if**(v1.size()!=v2.size()){ **return** -1;}  /\*incompatable\*/
86. **size\_t** vec\_size=v1.size();
87. v3.resize(vec\_size);
88. **for**(**size\_t** i=0;i<vec\_size;++i){  /\*chg all elements in v1\*/
89. v3[i]=v1[i]-v2[i];
90. }
91. **return** 0;
92. }
94. /\* print matrix\*/
95. **template** <**class** T>
96. **void** PrintMatrix(**const** vector<vector<T>> &mat){
97. **for**(auto x: mat){
98. **for**(T y: x){
99. cout<<std::setw(4)<<std::setiosflags(std::ios::left)<<y;
100. }
101. printf("\n");
102. }
103. }
105. /\*print vector\*/
106. **template** <**class** T>
107. **void** PrintVector(**const** vector<T> &vec){
108. **for**(T x: vec){
109. cout<<std::setw(4)<<std::setiosflags(std::ios::left)<<x;
110. }
111. printf("\n");
112. }
114. /\*print matrix and vectors of system\*/
115. **void** PrintSystemState(){
116. printf("Available vector:\n");
117. PrintVector(Available);
118. printf("Max matrix:\n");
119. PrintMatrix(Max);
120. printf("Allocation matrix:\n");
121. PrintMatrix(Allocation);
122. printf("Need matrix:\n");
123. PrintMatrix(Need);
124. }
126. /\*core part of banker algorithm\*/
128. **bool** SecurityTest(**const** vector<**int**> &avai, **const** vector<**int**> &need, **const** vector<**int**> &alloc, **size\_t** pid){
129. vector<**int**> Work(M); /\*working resources vector\*/
130. vector<**bool**> Finish(N,**false**); /\*finish list\*/
131. **for**(**int** i=0;i<M;++i){
132. Work[i]=avai[i];
133. } //resources remain
134. **bool** cnt=**true**;
135. **while**(cnt){
136. cnt=**false**;
137. **for**(**int** i=0;i<N;++i){
138. **if**(i==pid){  /\*pid case\*/
139. **if**(**false**==Finish[i]&&1==vec\_compare\_NotGreater<**int**>(need,Work)){ /\*found suitable process\*/
140. **if**(-1==vec\_PlusEuqal(Work,alloc)){  /\*Work=Work+Allocation\*/
141. **throw**("invalid change occured.");
142. }
143. cnt=**true**;
144. Finish[i]=**true**;
145. **break**; /\*find from begining\*/
146. }
147. }
148. **if**(**false**==Finish[i]&&1==vec\_compare\_NotGreater<**int**>(Need[i],Work)){ /\*found suitable process\*/
149. **if**(-1==vec\_PlusEuqal(Work,Allocation[i])){  /\*Work=Work+Allocation\*/
150. **throw**("invalid change occured.");
151. }
152. cnt=**true**;
153. Finish[i]=**true**;
154. **break**; /\*find from begining\*/
155. }
156. }
157. }
158. **for**(**bool** x:Finish){  /\*test if all elements in Finish is true\*/
159. **if**(**false**==x){**return** **false**;}
160. }
161. **return** **true**;
162. }
164. **bool** RequestResource(unsigned pid, **const** vector<**int**> &Request){
165. /\*check if pid is valid\*/
166. **if**(pid<0||pid>6){
167. printf("invalid process id.\nAllocation fail.\n");
168. **return** **false**;
169. }
170. /\*critical part\*/
171. WaitForSingleObject(Mutex,INFINITE);
172. **if**(1!=vec\_compare\_NotGreater(Request,Need[pid])){
173. printf("process%d: error occured, process%d's request exceed his max request.\n",pid,pid);
174. printf("Allocation fail.\n");
175. ReleaseMutex(Mutex);
176. **return** **false**;
177. }
178. **while**(1!=vec\_compare\_NotGreater(Request,Available)){
179. printf("process%d: No enough resources to allocate.\nAllocation fail.\n",pid);
180. /\*be hold that if the asked resources exceed the number of resources that system possessed,
181. the process will wait forever\*/
182. ReleaseMutex(Mutex);
183. WaitForSingleObject(AskForResources,INFINITE);
184. WaitForSingleObject(Mutex,INFINITE);
185. }
186. /\*security check and (if can), allocate\*/
187. vector<**int**> avai,alloc,need;
188. /\*simulated allocation\*/
189. vec\_Minus<**int**>(Available,Request,avai);
190. vec\_Plus<**int**>(Allocation[pid],Request,alloc);
191. vec\_Minus<**int**>(Need[pid],Request,need);
192. **if**(SecurityTest(avai,need,alloc,pid)){  /\*security varified\*/
193. /\*true allocation\*/
194. vec\_MinusEqual<**int**>(Available,Request);
195. vec\_PlusEuqal<**int**>(Allocation[pid],Request);
196. vec\_MinusEqual<**int**>(Need[pid],Request);
197. printf("process%d allocate success!\n",pid);
198. }**else**{
199. printf("process%d: unsafe application. Allocate fail!\n",pid);
200. }
201. /\*print current matrix(vectors)\*/
202. //PrintSystemState();
203. ReleaseMutex(Mutex);
204. **return** **true**;
205. }
207. **bool** ReleaseResource(unsigned pid, **const** vector<**int**> &Release){
208. /\*check if pid is valid\*/
209. **if**(pid<0||pid>6){
210. printf("process%d: invalid process id.\nAllocation fail.\n",pid);
211. **return** **false**;
212. /\*critical part\*/
213. WaitForSingleObject(Mutex,INFINITE);
214. }**if**(1!=vec\_compare\_NotGreater(Release,Allocation[pid])){
215. printf("process%d: error occured, process%d wants to release resources more that he possessed\n",pid,pid);
216. ReleaseMutex(Mutex);
217. **return** **false**;
218. }
219. /\*release resources\*/
220. vec\_PlusEuqal<**int**>(Available,Release);
221. vec\_MinusEqual<**int**>(Allocation[pid],Release);
222. vec\_PlusEuqal<**int**>(Need[pid],Release);
223. ReleaseSemaphore(AskForResources,1,NULL);
224. printf("process%d release successful.\n",pid);
225. /\*print current matrix(vectors)\*/
226. //PrintSystemState();
227. ReleaseMutex(Mutex);
228. **return** **true**;
229. }
231. /\*prehandle to main\*/
233. /\* initiate a vector with either random number or by user input
234. \* @param vec: vector that need initiation
235. \* @param vname: vector's name that need to info user
236. \* @param uinput: whether user need to input or not
237. \*/
238. **template** <**class** T>
239. **void** InitSimilationVector(vector<T> &vec, **const** **char**\* vname, **bool** uinput){
240. **size\_t** v\_size=vec.size();
241. /\*init type 1: input by users\*/
242. **if**(uinput){
243. printf("Please input your %s vector (with row %d):\n",vname,v\_size);
244. **for**(**size\_t** i=0;i<v\_size;++i){
245. cin>>vec[i];
246. }
247. }**else**{ /\*init type 2: default setting (for test conveniency)\*/
248. **for**(**size\_t** i=0;i<v\_size;++i){
249. vec[i]=(T)((**int**)rand()%30);  /\*randomly chose a number\*/
250. }
251. printf("the %s vector is:\n",vname);
252. **for**(auto x:vec){
253. std::cout<<x<<' ';
254. }
255. printf("\n");
256. }
257. }
259. /\* initiate a matrix with either random number or by user input
260. \* @param mat: matrix that need initiation
261. \* @param mname: matrix's name that need to info user
262. \* @param uinput: whether user need to input or not
263. \*/
264. **template** <**class** T>
265. **void** InitSimulationMatrix(vector<vector<T>> &mat, **const** **char**\* mname, **bool** uinput){
266. /\*init type 1: input by users\*/
267. **if**(uinput){
268. printf("Please input your %s matrix (with row %d and column %d):\n",mname,N,M);
269. **for**(**size\_t** i=0;i<N;++i){
270. **for**(**size\_t** j=0;j<M;++j){ cin>>mat[i][j];}
271. }
272. }**else**{ /\*init type 2: default setting (for test conveniency)\*/
273. **for**(**size\_t** i=0;i<N;++i){
274. **for**(**size\_t** j=0;j<M;++j){
275. MOD(mat[i][j],Available[j]);
276. }
277. }
278. printf("the %s matrix is:\n",mname);
279. PrintMatrix<**int**>(mat);
280. }
281. }
283. **DWORD** WINAPI Process(**void**\* param){
284. **int** pid=my\_pid++;
285. vector<**int**> Request(M,0);
286. vector<**int**> Release(M,0);
287. Sleep(2000);
288. **while**(TRUE){
289. srand((unsigned **int**)time(NULL));
290. Sleep(rand()%3000);
291. **if**(0==(**int**)(rand()%2)){ //asking for resources
292. /\*generate Request\*/
293. **for**(**int** i=0;i<M;++i){
294. MOD(Request[i],Need[pid][i]);
295. }
296. RequestResource(pid,Request);
297. }**else**{  //release resources
298. /\*generate release matrix\*/
299. **for**(**int** i=0;i<M;++i){
300. MOD(Release[i],Allocation[pid][i]);
301. }
302. ReleaseResource(pid,Release);
303. }
304. }
305. }
307. **int** main(**int** argc,**char**\*\* argv){
308. /\*init mutex for changing matrix(vectors)\*/
309. Mutex=CreateMutex(NULL,FALSE,NULL);
310. AskForResources=CreateSemaphore(NULL,1,7,NULL);
311. /\*init resources matrix\*/
312. **for**(auto x:Allocation){  /\*init Allocation\*/
313. **for**(**int** y:x){y=0;}
314. }
315. /\*available resources in system\*/
316. InitSimilationVector(Available,"Available",**false**);
317. /\*max number of resources that processes may require\*/
318. InitSimulationMatrix<**int**>(Max,"Max",**false**);
319. /\*number of resources that processes may still need\*/
320. Need=Max;
322. /\*allocate resources to process\*/
323. **for**(**int** i=0;i<7;++i){ /\*7 processes\*/
324. CreateThread(NULL,0,Process,NULL,0,NULL);
325. }
327. Sleep(40000);
328. **return** 0;
329. }