实验（二） 死锁观察与避免

19122557 王波

1. 目的与要求

1. 目的

死锁会引起计算机工作僵死，造成整个系统瘫痪。因此，死锁现象是操作系统特别是大型系统中必须 设法防止的。学生应独立的使用高级语言编写和调试一个系统动态分配资源的简单模拟程序，观察死锁产 生的条件，并采用适当的算法，有效的防止死锁的发生。通过实习，更直观地了解死锁的起因，初步掌握 防止死锁的简单方法，加深理解课堂上讲授过的知识。

2. 要求

（1） 设计一个 n 个并发进程共享 m 个系统资源的系统。进程可动态地申请资源和释放资源。系统按各 进程的申请动态地分配资源。

（2） 系统应能显示各进程申请和释放资源以及系统动态分配资源的过程，便于用户观察和分析。

（3） 系统应能选择是否采用防止死锁算法或选用何种防止算法（如有多种算法）。在不采用防止算法时 观察死锁现象的发生过程。在使用防止死锁算法时，了解在同样申请条件下，防止死锁的过程。

2.示例

1. 题目

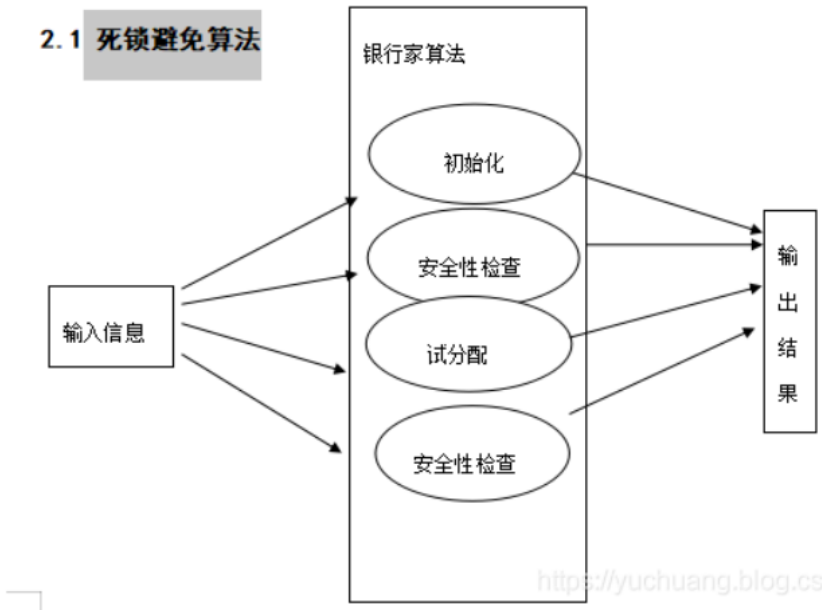
本示例采用银行算法防止死锁的发生。假设有三个并发进程共享十个系统。在三个进程申请的系统资 源之和不超过 10 时，当然不可能发生死锁，因为各个进程申请的资源都能满足。在有一个进程申请的系 统资源数超过 10 时，必然会发生死锁。应该排除这二种情况。程序采用人工输入各进程的申请资源序列。 如果随机给各进程分配资源，就可能发生死锁，这也就是不采用防止死锁算法的情况。假如，按照一定的 规则，为各进程分配资源，就可以防止死锁的发生。示例中采用了银行算法。这是一种犹如“瞎子爬山” 的方法，即探索一步，前进一步，行不通，再往其他方向试探，直至爬上山顶。这种方法是比较保守的。 所花的代价也不小。

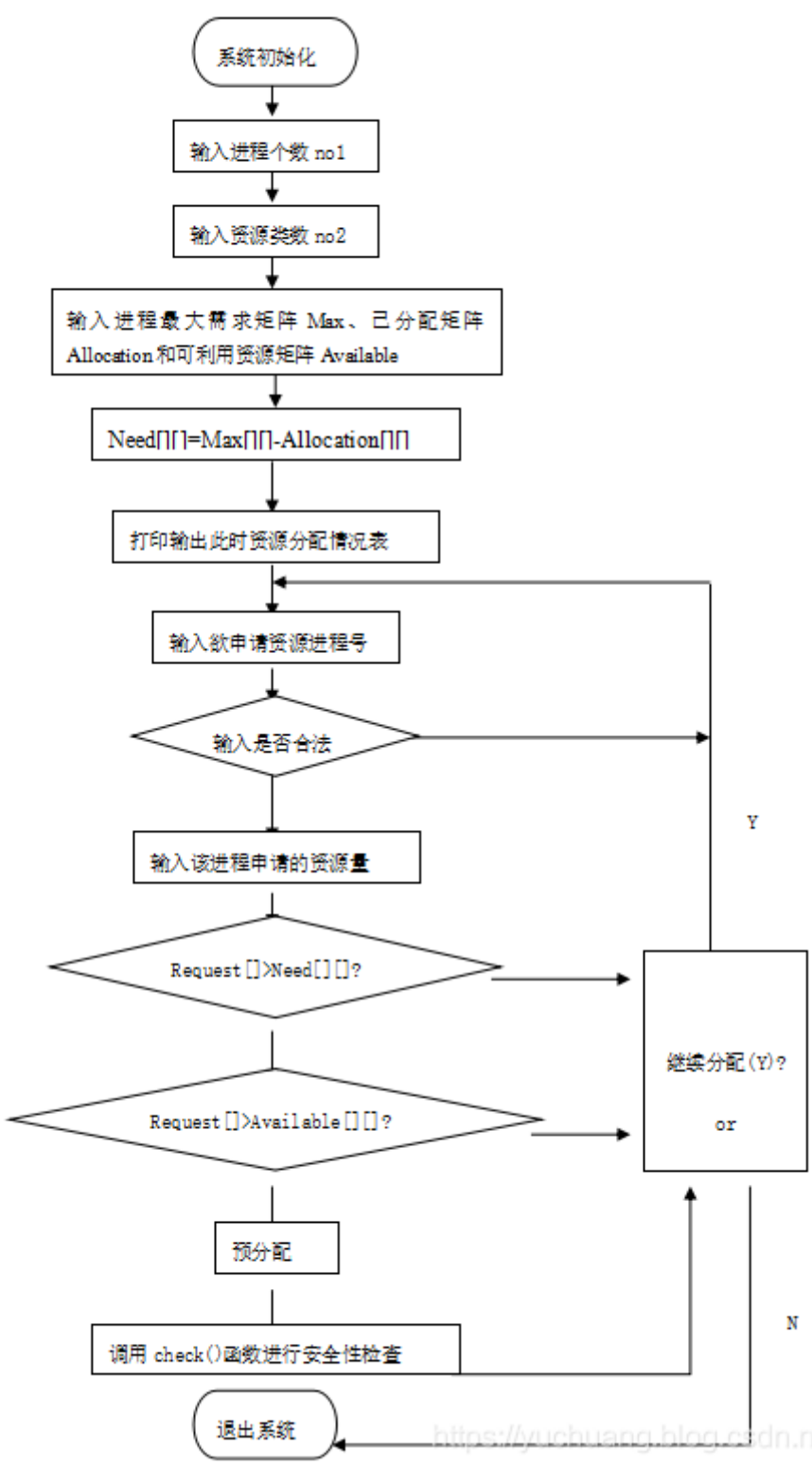
2. 算法

银行算法，顾名思义是来源于银行的借贷业务，一定数量的本金要应付各种客户的借贷周转，为了防 止银行因资金无法周转而倒闭，对每一笔贷款，必须考察其最后是否能归还。研究死锁现象时就碰到类似 的问题，有限资源为多个进程共享，分配不好就会发生每个进程都无法继续下去的死锁僵局。 银行算法的原理是先假定每一次分配成立，然后检查由于这次分配是否会引起死锁，即剩下的资源是不是 能满足任一进程完成的需要。如这次分配是安全的（不会引起死锁），就实施这次分配，再假定下一次分 配。如果不安全，就不实施，再作另一种分配试探，一直探索到各进程均满足各自的资源要求，防止了死锁的发生。

3. 总体设计

死锁避免算法





银行家算法要给定进程数和资源数，在随机输入符合要求的Allocation，Max，Need矩阵，给出Available向量，完成初始化。当有进程申请资源时要对比本身Need量以及可用Available量，满足后进行预分配，利用安全性算法进行检查。若结果为系统处于安全状态则正是为进程分配资源，否则驳回其请求。

死锁检测算法

该算法和安全性算法很类似，同样要给出Allocation，Max，Need矩阵，以及Available向量，同样要对每个进程进行Need与Available的比对，只是在比对后的操作略有不同，该算法要能够简化资源分配图，并给出结果。

3.死锁

1. 死锁的概念

在之前的哲学家吃饭的问题中，当每个哲学家都想进餐的时候，他们都会占用左手边的筷子。当他们想要拿起右手边的筷子的时候，因为没有资源了。所以程序会进入无线等待的状态，这就是死锁。

我们可以来想想一个更加简单的例子。比如说有两个信号量集，一个是扫描机，一个是打印机。P1程序占用打印机，P2程序占用扫描机。当P1程序在运行的时候想要获取扫描机的信号量，同时P2程序想要在运行的时候获取打印机的信号量的时候。就会发生死锁。

概括而言。我们可以发现死锁的会有如下共性：

- 参与死锁的进程至少有两个
- 每个参与死锁的进程都要等待资源
- 参与死锁的进程中至少有两个进程占用资源

可见，死锁之所以产生，是因为每个进程都要竞争资源。由于**系统资源不足，并且推进程序不当**，因此产生了死锁。

2. 死锁实例演示

在之前的消费者和生产者问题中，如果改变了 生产者的 信号量处理方法，则会产生死锁。具体实例如下所示，代码参考 producer_and_consumer-dead-lock.c。

```
//
int i;
for(i=0;i< MAXSIZE;i++ ){
// i ++ eq provide data
    sem_wait(&mutex);
    sem_wait(&empty);
    //sem_wait(&mutex);
    // put item into stack
    stack[i] = i;
    usleep(100);
    // put item into stack
    sem_post(&mutex);
    sem_post(&full);
    //usleep(1000);
}
}

void Consumer(void){
    int i;
    while((i=size++) < MAXSIZE) {
        sem_wait(&full);
        sem_wait(&mutex);
        // consumer use data
        printf("calculating... %d * %d = %d \n",stack[i],stack[i],
stack[i]*stack[i]);
        usleep(100000*2);
        // consumer use data
        sem_post(&mutex);
        sem_post(&empty);
    }
```

现在更改provider中
最开始两个信号量的位置
就可以观察到死锁的情况。
假设：
空缓存 信号量为 1
满缓存 信号量为 0

生产者在完成第一遍循环后。
空缓存 信号量为 0
满缓存 信号量为 1

随后消费者获取 &full 的信号。

此时，如果生产者先于消费者获取 &mutex的信号。则生产者会在 sem_wait(&empty)的地方卡住。而消费者会在 sem_wait(&mutex) 的地方卡住。

卡住

卡住

直接运行程序，可以发现程序是直接卡死的。这里的解决办法就是Provider 下方的usleep(1000)。让消费者先行一步。即可避免死锁。

4.银行家算法

银行家算法是DJ提出的，最具代表性的避免死锁算法。假设系统中有 n个进程 P1-Pn 和 m类资源 R1-Rm，那么建立以下形式的数据结构：

初始化

由用户输入数据，分别对可利用资源向量矩阵AVAILABLE、最大需求矩阵MAX、分配矩阵ALLOCATION、需求矩阵NEED赋值。

银行家算法

在避免死锁的方法中，所施加的限制条件较弱，有可能获得令人满意的系统性能。在该方法中把系统的状态分为安全状态和不安全状态，只要能使系统始终都处于安全状态，便可以避免发生死锁。

银行家算法的基本思想是分配资源之前，判断系统是否是安全的；若是，才分配。它是最具有代表性的避免死锁的算法。

设编号为ID的进程提出请求new_request，定义（关于向量比较、运算的定义，代码实现部分给出，这里不再赘述）：

i=new_request->id为该进程的编号，

new_request->req_src为该进程此次所请求的资源向量，

则银行家算法按如下规则进行判断：

1.如果new_request->req_src <= Need[i] 则转2；否则，出错。

2.如果new_request->req_src <= Available[i]，则转3；否则，等待。

3.系统试探分配资源，修改相关数据：

Available[i] -= new_request->req_src ;

Allocation[i] += new_request->req_src;

Need[i] -= new_request->req_src;

4.系统执行安全性检查，如安全，则分配成立；否则试探险性分配作废，系统恢复原状，进程等待。

安全性检查算法

1.设置两个工作向量

Work 记录系统当前可用资源量，初值为Available;

finish 记录所有进程是否已被执行, 初值为长度为n，值均为False的向量。

2.从进程集合中找到一个满足下述条件的进程，

finish == False;

Need <= Work;

如找到，执行3；否则，执行4。

3.假设进程获得资源，可顺利执行，直至完成，从而释放资源。

Work += Allocation;

Finish=True;

执行2

4.如所有的进程finish= True，则表示安全；否则系统不安全。

代码实现（JAVA语言）

首先，将需要的变量定义为全局变量

```
int n;   //进程数
int m;   //资源类数
int[] Available; //可使用资源向量
int[][] Max;   //最大需求矩阵
int[][] Allocation;      //分配矩阵
int[][] Need; //需求矩阵
boolen safe = False;
class Request
{
      int id;   //进程ID
      int *req_src;   //进程此次申请资源
}
Request[] new_request;
```

下面列出了我们将要写的函数：

```
public void initial();                    //初始化n,m,Available等的函数
public void request();       //提出请求
public void process();           //处理
public bool safe_detect();       //安全性检测
/*向量运算函数*/
public bool vector_compare(int[] a, int[] b, int len);
public void vector_add(int[] a, int[] b, int len);
public void vector_sub(int[] a, int[] b, int len);
```

首先给出几个向量运算函数的定义：
定义a和b为两个等长向量，
a >= b 表示 a 中的每个元素都大于相应位置上的 b 的元素；
a += b 表示 a 中的每个元素增加相应位置上的 b 的元素的值；
a -= b 表示 a 中的每个元素都大于相应位置上的 b 的元素的值；
例：
a = [1,2,3];
b = [1,1,1];
则
a >= b;
a += b; //a=[2,3,4]
a -= b; //a=[0,1,2]

```
public bool vector_compare(int *a, int *b, int len)      // If vector a >= vector b, return True
{
      int i = 0;
      while(i<len)
     {
            if(*(a+i)<*(b+i))
                  return False;
            i++;
     }
      return True;
}
public void vector_add(int *a, int *b, int len) //vector a += vector b
{
      int i = 0;
      while(i<len)
     {
            *(a+i) += *(b+i);
            i++;
     }
}
public void vector_sub(int *a, int *b, int len) //vector a -= vector b
{
      int i = 0;
      while(i<len)
     {
            *(a+i) -= *(b+i);
            i++;
     }
}
```

下面按算法步骤给出 initial(), request(), process(), safe_request()

```
public void initial()
{
      int i;
      int j;
      printf("请输入进程数:\n");
      scanf("%d",&n);
      printf("请输入资源类数:\n");
      scanf("%d",&m);
      printf("请输入可使用资源向量:\n");
      Available = (int*)malloc(sizeof(int)*m);
```

```
 for(i=0; i<m; i++)
        scanf("%d",&Available[i]);
 printf("请输入最大需求矩阵:\n");
 Max = (int**)malloc(sizeof(int*)*n);
 for(i=0; i<n; i++)
{
        Max[i] = (int*)malloc(sizeof(int)*m);
        for(j=0; j<m; j++)
              scanf("%d",&Max[i][j]);
}
 printf("请输入分配矩阵:\n");
 Allocation = (int**)malloc(sizeof(int*)*n);
 for(i=0; i<n; i++)
{
        Allocation[i] = (int*)malloc(sizeof(int)*m);
        for(j=0; j<m; j++)
              scanf("%d",&Allocation[i][j]);
}
 Need = (int**)malloc(sizeof(int*)*n);
 for(i=0;i<n;i++)
{
      Need[i] = (int *)malloc(sizeof(int)*m);
      for(j=0;j<m;j++)
            Need[i][j] = Max[i][j] - Allocation[i][j];
}
public void request()
{
      int i,id;
      new_request = (Request*)malloc(sizeof(Request));
      new_request->req_src = (int*)malloc(sizeof(int)*m);
      printf("请输入进程的ID\n");
      scanf("%d",&id);
      new_request->id = id - 1;
      printf("请输入进程申请资源向量\n");
      for(i=0; i<m; i++)
            scanf("%d",&new_request->req_src[i]);
}
public void process()
{
      int i = new_request->id;
      if(vector_compare(Need[i],new_request->req_src,m))
           {
       if(vector_compare(Available,new_request->req_src,m))
            {
                  vector_sub(Available,new_request->req_src,m);
                  vector_add(Allocation[i],new_request->req_src,m);
                  vector_sub(Need[i],new_request->req_src,m);
                  safe_detect();
            }
             else
            {
                  printf("程序所申请资源大于系统当前所剩资源，推迟执行!\n");
                  return;
            }

      }
 else
{
       printf("程序所申请资源大于该程序所需资源，无法执行!\n");
       return;
}
```

```
        if(safe)
        {
                printf("系统安全,进程可以执行!\n");
                return;
        }
        else
        {
                printf("系统不安全,进程无法执行!\n");
                vector_add(Available,new_request->req_src,m);
                vector_sub(Allocation[i],new_request->req_src,m);
                vector_add(Need[i],new_request->req_src,m);
                return;
        }

}
public bool safe_detect()
{
        int *work = Available;
        bool *finish = (bool*)malloc(sizeof(bool)*n);
        int i;
        //初始化finish
        for(i=0; i<n; i++)
                finish[i] = False;

        for(i=0; i<n; i++)
        {
                if(finish[i]==False&&vector_compare(work,Need[i],m))
                {
                        printf("尝试执行第%d进程\n",i+1);
                        vector_add(work,Allocation[i],m);     //尝试执行该进程，释放资源
                        finish[i] = True;
                        i = -1;   //尝试分配后，从头查找是否还有可以执行的进程，考虑到i++，故此处为-1
                }
        }

        for(i=0; i<n; i++)
                if(finish[i]==False)
                        break;
        if(i==n)
                safe = True;
        else
                safe = False;
}
```

实验结果

```
E:\Java\bin\java.exe "-javaagent:E:\JetBrains\IntelliJ IDEA\lib\idea_rt.jar=8324:E:\JetBrains\IntelliJ IDEA\bin" -Dfile.encoding=UTF-8 -classpath E:\Users\id-none\Desktop\OperateSystem\Operatecode\target\classes BankerAlgorithm
请输入进程数：4
请输入资源种类数：3
请输入最大需求矩阵max
3 2 2
6 1 3
3 1 4
4 2 2
请输入分配矩阵allocation
1 0 0
5 1 1
2 1 1
0 0 2
请输入需求矩阵need
```

```
2 2 2
1 0 2
1 0 3
4 2 0
请输入可用资源向量available
1 1 2
分配序列：
                allocation              need                available

进程2        5      1      1        1      0      2        6      2      3
进程1        1      0      0        2      2      2        7      2      3
进程3        2      1      1        1      0      3        9      3      4
进程4        0      0      2        4      2      0        9      3      6
存在安全序列，初始状态安全。
请输入发出请求向量request的进程编号：01
请输入请求向量request
1 0 2
分配序列：
                allocation              need                available

不允许1进程申请资源!
请输入发出请求向量request的进程编号：01
1 0 1
请输入请求向量request
分配序列：
                allocation              need                available

不允许1进程申请资源!
请输入发出请求向量request的进程编号：2
请输入请求向量request
1 0 1
分配序列：
                allocation              need                available

进程11       6      1      2        0      0      1        6      2      3
进程01       1      0      0        2      2      2        7      2      3
进程21       2      1      1        1      0      3        9      3      4
进程31       0      0      2        4      2      0        9      3      6
允许2进程申请资源!
请输入发出请求向量request的进程编号：
```

5. 实验总结

在本次实验中，一开始对于整个设计的框架比较模糊，看了很多网上的教程，大致建立一个框架，再往这个框架里面填入一些函数，搭建成一个个可以实现某功能的结构。比如初始化函数模块、安全性检测函数模块、银行家算法模块，另外，可以实现对时间复杂度的优化。但在实现银行家算法的过程中，由于一些函数调用以及变量命名的重复，思路卡了几次，我去请教了一些同学，他们给了我一些解决方法和思路，使我把这几个问题解决了。

经过这次的课程设计实验，我学到了许多的知识及熟练地掌握了银行家算法的思想，同时进一步提高了我的算法思维能力，也证明了我还有很多地知识还要不断地学习 ，不断地锻炼我的算法思维，在今后的课程学习中，我将继续前进。

实验源码可见https://github.com/id-none/OperateSystem/blob/master/Code/src/main/java/com/banker/BankerAlgorithm.java