Xi XIA
2016213482
2016215117

7.11 Consider the traffic deadlock depicted in Figure 7.10.

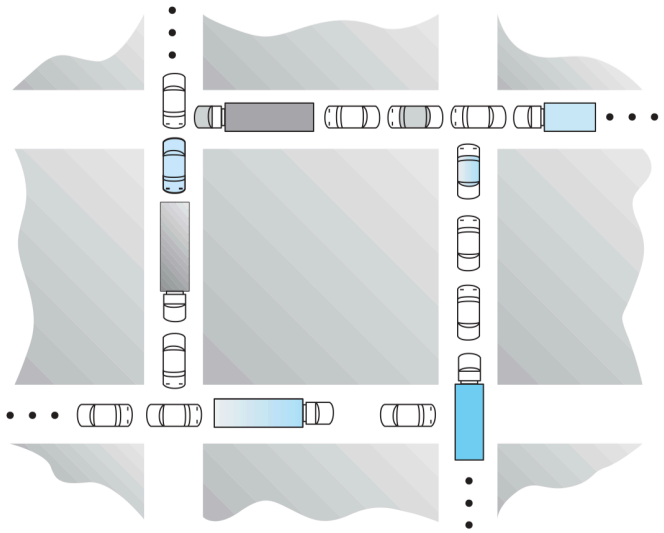


Figure 7.10 Traffic deadlock for Exercise 7.11.

a. Show that the four necessary conditions for deadlock hold in this example.

Each cross of the streets is considered as a resource, each line of cars is considered as a process.

- Mutual exclusion: only one line of cars at a time can use the resource.
- Hold and wait: Each line of cars is holding one resource and is waiting for the next resource.
- No preemption: the resource can't be released until the whole line of cars have passed it.
- Circular wait: there are 4 lines of cars l1, l2, l3, l4, l1 is waiting for l2, l2 is waiting for l3, l3 is waiting for l4, l4 is waiting for l1.

b. State a simple rule for avoiding deadlocks in this system.

There are many ways to avoid the deadlocks in this system. one way is to break the second condition: a line of cars can't hold a cross and wait, it's that **no car of a line can stay in the cross.**

Programming Projects

Banker's Algorithm

For this project, you will write a multithreaded program that implements the banker's algorithm discussed in Section 7.5.3. Several customers request and release resources from the bank. The banker will grant a request only if it leaves the system in a safe state. A request that leaves the system in an unsafe state will be denied. This programming assignment combines three separate topics: (1) multithreading, (2) preventing race conditions, and (3) deadlock avoidance.

Xi XIA
2016213482
2016215117

The Banker

The banker will consider requests from n customers for m resources types. as outlined in Section 7.5.3. The banker will keep track of the resources using the following data structures:

```
/* these may be any values >= 0 */
#define NUMBER_OF_CUSTOMERS 5
#define NUMBER_OF_RESOURCES 3

/* the available amount of each resource */
int available[NUMBER_OF_RESOURCES];

/*the maximum demand of each customer */
int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

/* the amount currently allocated to each customer */
int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

/* the remaining need of each customer */
int need[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
```

The Customers

Create n customer threads that request and release resources from the bank. The customers will continually loop, requesting and then releasing random numbers of resources. The customers' requests for resources will be bounded by their respective values in the need array. The banker will grant a request if it satisfies the safety algorithm outlined in Section 7.5.3.1. If a request does not leave the system in a safe state, the banker will deny it. Function prototypes for requesting and releasing resources are as follows:

```
int request_resources(int customer_num, int request[]);

int release_resources(int customer_num, int release[]);
```

These two functions should return 0 if successful (the request has been granted) and –1 if unsuccessful. Multiple threads (customers) will concurrently access shared data through these two functions. Therefore, access must be controlled through mutex locks to prevent race conditions. Both the Pthreads and Windows APIs provide mutex locks. The use of Pthreads mutex locks is covered in Section 5.9.4; mutex locks for Windows systems are described in the project entitled "Producer–Consumer Problem" at the end of Chapter 5.

Implementation

You should invoke your program by passing the number of resources of each type on the command line. For example, if there were three resource types, with ten instances of the first type, five of the second type, and seven of the third type, you would invoke your program follows:

```
./a.out 10 5 7
```

The available array would be initialized to these values. You may initialize the maximum array (which holds the maximum demand of each customer) using any method you find convenient.

Xi XIA
2016213482
2016215117

Main Idea of my Code for the Programming Project

Algorithm 1: Safety Algorithm

@see: BankImpl#isSafeState

1. Check whether (request of Customer # threadNum) < (Availiable Resources)
2. Initialize an array canFinish[m] to record whether all thread can finish in an order
3. Assume that a request is allowed, check whether the other Customers' request can be finished. If not , record "false" in the corresponding index in the canFinish array.
4. Final check the canFinsh array, if all the values are "true", then it is in a safe state.

```
1.  /**
2.   * Determines whether granting a request results in leaving the system in a safe st
    ate or not.
3.   *
4.   * private int m: the number of resources
5.   * private int n: the number of threads in the system
6.   *
7.   * @param threadNum 0 ~ (m-1)
8.   * @param request request of resources of Customer # threadNum
9.   * @return false - the system is NOT in a safe state.
10.  */
11. private boolean isSafeState(int threadNum, int[] request) {
12.     System.out.print("\n Customer # " + threadNum + " requesting ");
13.     for (int i = 0; i < m; i++) System.out.print(request[i] + " ");
14.
15.     System.out.print("Available = ");
16.     for (int i = 0; i < m; i++)
17.         System.out.print(available[i] + "  ");
18.
19.     // first check if there are sufficient resources available
20.     for (int i = 0; i < m; i++)
21.         if (request[i] > available[i]) {
22.             System.err.print("INSUFFICIENT RESOURCES for Customer # "+threadNum);
23.             return false;
24.         }
25.
26.     // ok, there are sufficient reources. Now let's see if we can find an ordering
    of threads to finish
27.     boolean[] canFinish = new boolean[n];
28.     for (int i = 0; i < n; i++)
29.         canFinish[i] = false;
30.
31.     // copy the available matrix to avail
32.     int[] avail = new int[m];
33.     System.arraycopy(available, 0, avail, 0, available.length);
34.
35.
36.     // Now decrement avail by the request.
37.     // Temporarily adjust the value of need for this thread.
38.     // Temporarily adjust the value of allocation for this thread.
39.     for (int i = 0; i < m; i++) {
40.         avail[i] -= request[i];
41.         need[threadNum][i] -= request[i];
42.         allocation[threadNum][i] += request[i];
43.     }
44.
45.     /**
46.      * Now try to find an ordering of threads so that each thread can finish.
47.      * private int m: the number of resources
48.      * private int n: the number of threads in the system
49.      */
```

```
50.     for (int i = 0; i < n; i++) {
51.         // find the first thread that can finish
52.         for (int j = 0; j < n; j++){
53.             if (!canFinish[j]) {
54.                 boolean temp = true;
55.                 // need of m resources
56.                 for (int k = 0; k < m; k++) {
57.                     if (need[j][k] > avail[k])
58.                         temp = false;
59.                 }
60.                 if (temp) { // if this thread can finish
61.                     canFinish[j] = true;
62.                     for (int x = 0; x < m; x++)
63.                         // restore the temporarily deduced avail of resource x
64.                         avail[x] += allocation[j][x];
65.                 }
66.             }
67.         }
68.     }
69.     // restore the value of need and allocation for this thread
70.     for (int i = 0; i < m; i++) {
71.         need[threadNum][i] += request[i];
72.         allocation[threadNum][i] -= request[i];
73.     }
74.
75.     // now go through the boolean array and see if all threads could complete
76.     boolean returnValue = true;
77.     for (int i = 0; i < n; i++)
78.         if (!canFinish[i]) {
79.             returnValue = false;
80.             System.err.print("UNSAFE STATE - CAN'T FIND AN ORDERING OF THREADS TO F
    INISH ALL");
81.             break;
82.         }
83.     return returnValue;
84. }
```

Algorithm 2: Resource-Request Algorithm

@see: BankImpl#requestResources

1. If the request result in an unsafe state, deny the request
2. Else, continue to print the allocation and remaining need of Customer

```
1.  /**
2.   * Make a request for resources. This is a blocking method that returns
3.   * only when the request can safely be satisfied.
4.   *
5.   * @param threadNum The number of the customer being added.
6.   * @param request The request for this customer.
7.   * @return false - the request is not granted.
8.   */
9.  public synchronized boolean requestResources(int threadNum, int[] request) {
10.     if (!isSafeState(threadNum, request)) {
11.         System.out.print(" Customer # " + threadNum + " is denied.");
12.         return false;
13.     }
14.
15.     // isSafeState = true, all threads can complete in an order
16.     // if it is safe, allocate the resources to thread threadNum
17.     for (int i = 0; i < m; i++) {
18.         available[i] -= request[i];
19.         allocation[threadNum][i] += request[i];
```

```
20.         need[threadNum][i] = maximum[threadNum][i] - allocation[threadNum][i];
21.     }
22.
23.     System.out.print("ResourceAllocated = ");
24.     for (int i = 0; i < m; i++)
25.         System.out.print(allocation[threadNum][i] + "  ");
26.
27.
28.     System.out.print(" Remaining Need = ");
29.     for (int i = 0; i < m; i++)
30.         System.out.print(need[threadNum][i] + "  ");
31.     return true;
32. }
```

Main Classes: (Factory, Customer)

@see: Factory.java

1. Read file “infile.txt” to initialize the maximum need matrix of (m Customers) * (n Resources)
2. Create thread for each Customer

```
3.  /**
4.   * A factory class that creates (1) the bank and (2) each customer at the bank.
5.   *
6.   * Usage:
7.   *  java Factory <one or more resources>
8.   *
9.   * I.e.
10.  *  java Factory 10 5 7
11.  */
12.
13. import java.io.*;
14. import java.util.*;
15.
16. public class Factory
17. {
18.     public static void main(String[] args) {
19.         int numOfResources = args.length;
20.         int[] resources = new int[numOfResources];
21.
22.         // initialize resources with the args[]
23.         for (int i = 0; i < numOfResources; i++)
24.             resources[i] = Integer.parseInt(args[i].trim());
25.
26.         // initialized bank with resources[]
27.         Bank theBank = new BankImpl(resources);
28.         // array of max demand
29.         int[] maxDemand = new int[numOfResources];
30.
31.         // initialize the customers
32.         Thread[] workers = new Thread[Customer.COUNT];
33.
34.         // read initial values for maximum need array
35.         String line;
36.         try {
37.             BufferedReader inFile = new BufferedReader(new FileReader("/Users/xiaxi
    /Documents/操作系统/assignment/assignment CH7/Bank/src/infile.txt"));
38.
39.             int threadNum = 0;
40.             int resourceNum = 0;
41.
42.             for (int i = 0; i < Customer.COUNT; i++) {
43.                 line = inFile.readLine();
44.                 StringTokenizer tokens = new StringTokenizer(line,",");
45.
```

```
46.                 while (tokens.hasMoreTokens()) {
47.                     int amt = Integer.parseInt(tokens.nextToken().trim());
48.                     maxDemand[resourceNum++] = amt;
49.                 }
50.                 workers[threadNum] = new Thread(new Customer(threadNum, maxDemand,
    theBank));
51.                 theBank.addCustomer(threadNum,maxDemand);
52.                 //theBank.getCustomer(threadNum);
53.                 ++threadNum;
54.                 resourceNum = 0;
55.             }
56.         }
57.         catch (FileNotFoundException fnfe) {
58.             throw new Error("Unable to find file \"infile.txt\"");
59.         }
60.         catch (IOException ioe) {
61.             throw new Error("Error processing \"infile.txt\"");
62.         }
63.
64.         System.out.println("FACTORY: created threads");
65.
66.         // workers(Thread) starts
67.         for (int i = 0; i < Customer.COUNT; i++)
68.             workers[i].start();
69.
70.         System.out.println("FACTORY: started threads");
71.
72.         /**
73.          try { Thread.sleep(5000); } catch (InterruptedException ie) { }
74.          System.out.println("FACTORY: interrupting threads");
75.          for (int i = 0; i < Customer.COUNT; i++)
76.          workers[i].interrupt();
77.          */
78.         // start all the customers
79.
80.     }
81. }
```

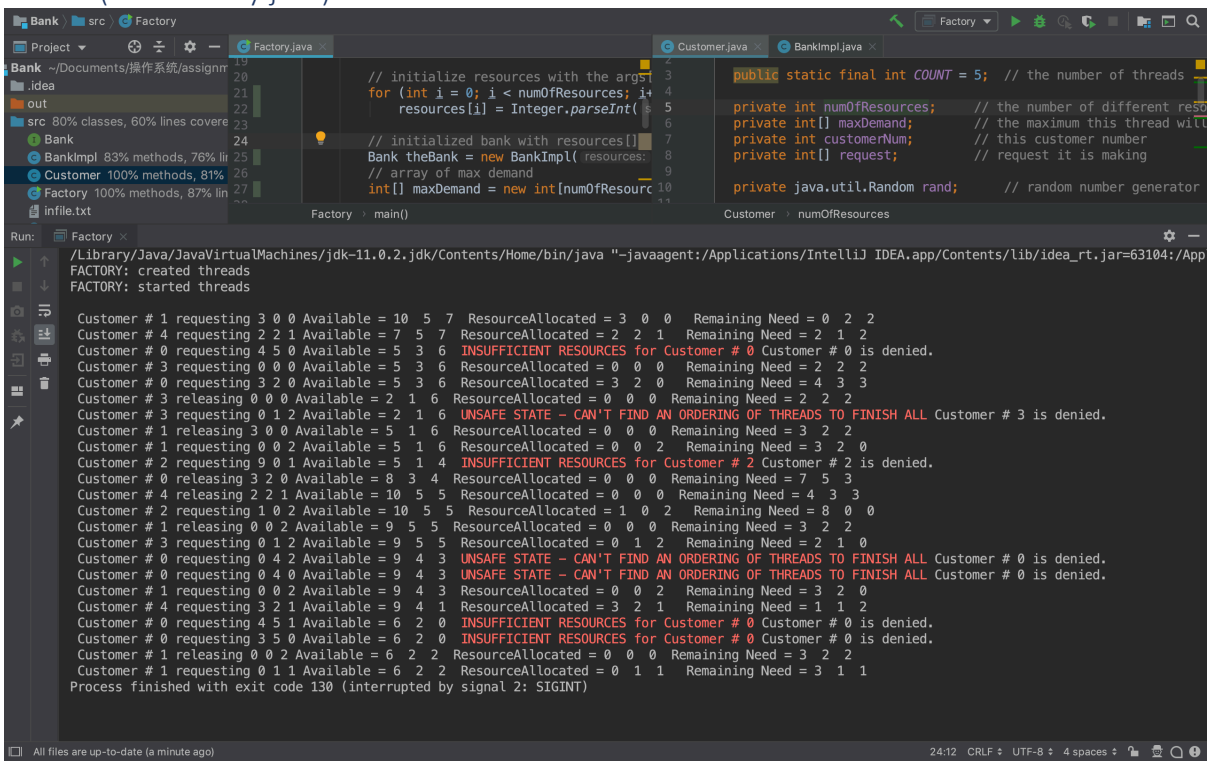
@see: Customer.java

3. For each customer, randomly generate request, execute the function requestResources, after waiting for a period of time, release the resources to continue other test.

```
1.  public class Customer implements Runnable
2.  {
3.          public static final int COUNT = 5;  // the number of threads
4.
5.          private int numOfResources;     // the number of different resources
6.          private int[] maxDemand;        // the maximum this thread will demand
7.          private int customerNum;        // this customer number
8.          private int[] request;          // request it is making
9.
10.         private java.util.Random rand;      // random number generator
11.
12.         private Bank theBank;           // synchronizing object
13.
14.     public Customer(int customerNum, int[] maxDemand, Bank theBank) {
15.             this.customerNum = customerNum;
16.             this.maxDemand = new int[maxDemand.length];
17.             this.theBank = theBank;
18.
19.             System.arraycopy(maxDemand,0,this.maxDemand,0,maxDemand.length);
20.             numOfResources = maxDemand.length;
```

```
21.             request = new int[numOfResources];
22.             rand = new java.util.Random();
23.     }
24.
25.         public void run() {
26.             boolean canRun = true;
27.
28.             while (canRun) {
29.                 try {
30.                     // rest for awhile
31.                     SleepUtilities.nap();
32.
33.                     // make a resource request
34.                     for (int i = 0; i < numOfResources; i++)
35.                         request[i] = rand.nextInt(maxDemand[i]+1);
36.
37.                     // see if the customer can proceed
38.                     if (theBank.requestResources(customerNum, request)) {
39.                         // use the resources
40.                         SleepUtilities.nap();
41.
42.                         // release the resources
43.                         theBank.releaseResources(customerNum, request);
44.                     }
45.                 }
46.                 catch (InterruptedException ie) {
47.                     canRun = false;
48.                 }
49.             }
50.             System.out.println("Thread # " + customerNum + " I'm interrupted.");
51.         }
52. }
```

Result: (Run Factory.java)



We can see the process of **requesting/ releasing resources**, and the error insufficient resources/ Unsafe state.