Applied Mechanics and Materials
ISSN: 1662-7482, Vols. 121-126, pp 1191-1195
doi:10.4028/www.scientific.net/AMM.121-126.1191

Online: 2011-10-24

A Deadlock Prevention Using Adjacency Matrix on Dining Philosophers Problem

Jinsong Zhan[a], Yongning Guo[b] and Chenglian Liu[c*]

Department of Mathematics and Computer Science,

Fuqing Branch of Fujian Normal University, Fuqing 350300 China

[a]jszhan115@hotmail.com, [b]guoyn@163.com, [c]chenglian.liu@gmail.com

[*] Corresponding Author: Mr. Liu is with Department of Mathematics and Computer Science, Fuqing Branch of Fujian Normal University, China.

Keywords: Dining Philosophers Problem, Prevent Deadlock, Resource Starvation, Memory Consistency

Abstract. In computer science, the dining philosopher's problem is an illustrative example of a common computing problem in concurrency. It is a classic multi-process synchronization problem. In this paper, we proposed a mathematical model which it expresses by adjacency matrix to show the deadlock occurs, and how resolve it.

1. Introduction

In 1965, Dijkstra [1] set an examination question on a synchronization problem where five computers competed for access to five shared tape drive peripherals. Soon afterwards the problem was retold by Tony Hoare as the dining philosopher's problem [2-3]. This is a theoretical explanation of deadlock and resource starvation by assuming that each philosopher takes a different fork as a first priority and then looks for another. Zhan and Guo [4] gave as an example of a java code to prevent deadlock based on dining philosopher's problem. Here we proposed a mathematical model which it express by adjacency matrix, and then rewrite the Zhan-Guo's JAVA code. Section 2 is review dining philosopher problem, and discuss deadlock prevention with solution. The section 3 gives an algorithm concept and its pseudo code. The conclusion will be drawing in final section.

2. Review of Dining Philosophers Problem

The dining philosopher's problem is summarized as five silent philosophers sitting at a circular table doing one of two things: eating or thinking. While eating, they are not thinking, and while thinking, they are not eating. A large bowl of Spaghetti is placed in the center, which requires two forks to serve and to eat (the problem is therefore sometimes explained using rice and chopsticks rather than spaghetti and forks). A fork is placed in between each pair of adjacent philosophers, and each philosopher may only use the fork to his left and the fork to his right. However, the philosophers do not speak to each other. With five points, said five philosophers [5-6]. The $< e_i, e_j >$ means that philosopher i took chopsticks between himself and philosopher j. The adjacency matrix model express as follow:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \quad (1)$$

While

$$i - j \equiv \pm 1 \pmod 5, \longmapsto a_{ij} \in \{0, 1\}. \quad (2)$$

While

$$i - j \not\equiv \pm 1 \pmod 5, \longmapsto a_{ij} = 0. \tag{3}$$

While

$$i - j = 1, \longmapsto a_{ji} = 0. \tag{4}$$

If

$$\prod_{j=1}^{5} (a_{ij} = 2), \tag{5}$$

then the philosopher could be dining. The diagram draws in figure 1.

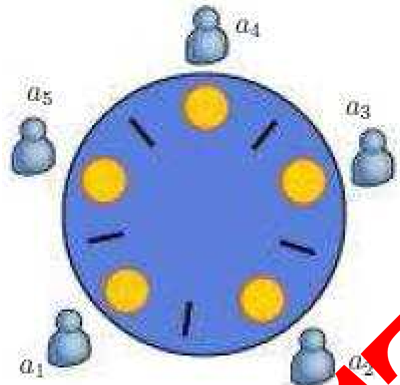

Figure 1. Dining Philosophers Problem [7]

A. An Improved Solution

Dining philosophers' problem is a classic synchronization problem. By the algorithm to limit damage resulting deadlock four necessary conditions can prevent the occurrence of deadlock. Java language-level support multithreading, the programming to use Java multithreading deadlock on the dining philosophers' problem and its prevention study provides a good simulation and verification.

B. Prevent Deadlock

When the adjacency matrix A matches situations to the following two cases, then a deadlock occurs.

$$A_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{6}$$

$$A_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{7}$$

We therefore have to destroy the above to occur.

3. THE ALGORITHM DESCRIPTION USING JAVA

There are many variety solutions to dining philosophers problem of deadlock preventions, one of them is to provide the philosopher are available only in the case of two chopsticks to pick up the chopsticks. The essence of this algorithm produces a necessary condition for deadlock destruction of part of resource allocation criteria.

A. Algorithm and Pseudo Code

We improved the algorithms to improve the algorithm described below:

```
=====
    semahore chopstick[0]=chopstick[1]=chopstick[2]=chopstick[3]=chopstick[4]=1;
    semaphore mutex=1;
    boolean chop[0]=chop[1]=chop[2]=chop[3]=chop[4]=true;
    Philosopher  i:
    while(true)
    {thinking;
     while(!test(i)){Waiting};  //If the test is not passed, in a wait state until the test.//
     P(chopstick[i]);
     P(chopstick[(i+1)%5]);
     Dining;
     V(chopstick[i]);
     V(chopstick[(i+1)]);
     chop[i]=chop[(i+1)%5]=true;   //Chopsticks can be used to set these two flags.//
     Announcing;   //Notice from the wait state into  the test state.//}
    boolean test(int i)
    {P(mutex);  //Common semaphore for mutual exclusion testing process.//
     if(chop[i]&&chop[(i+1)%5])
     {chop[i]=chop[(i+1)%5]=false; //Setting these two chopsticks unavailable flag.//
      return true;}
      else
     {return false;  }
     V(mutex);}
=====
```

Our test algorithm design a process, [illegible]ssed test process, the thread can enter the food process. Otherwise, the thread enters a wait state. Through the test process, the philosopher is not available around the chopsticks set to sign, to prevent the adjacent philosophers through the test process. This fact indicates that the philosopher needs two chopsticks have been assigned to him, that is, pre-allocate all resources. Semaphore mutex public role is limited to test the process, significantly less than the before method.

B. Source Code in Java

This can be achieved using the Java language algorithm. Procedures are as follows:

```
//MyPhilo8.java
import java.util.Random;
class Chopstick{ private int i;
                 public Chopstick(int i){this.i=i;}
                 public String toString(){return "Chopstick"+i;}
                }
class Philosopher extends Thread{private int i;
                                  private Random rand=new Random();
                                  private Chopstick leftChopstick,
                                 rightChopstick;
                                  private static int ponder=10;
                                  private static volatile boolean[]
                                 flag={true,true,true,true,true};   //ensure volatile types consistency in memory//
      public Philosopher(int i,Chopstick left,Chopstick right)
              { this.i=i;
               this.leftChopstick=left;
               this.rightChopstick=right;
               start();
              }
```

```
        public String toString()
            {return "philosopher"+i;}
        public static synchronized
          boolean test(int i)    //testing and setting//
                    { if(flag[i]&&flag[(i+1)%5]){flag[i]=flag[(i+1)%5]=false; return true; }
                      return false;
                    }
        public static synchronized void
          testAndWait(int i)
            {
          while(!test(i))
          {try{
                  Philosopher.class.wait();       //waiting//
              } catch(InterruptedException e){}
                  }
                  }
   public static void release(int i)
     { flag[i]=flag[(i+1)%5]=true;
          synchronized(Philosopher.class)
          { Philosopher.class.notifyAll();   //announcing//
          }
        }
   public void think()
     {System.out.println(this+"thinking");
      try  {Thread.sleep(rand.nextInt(ponder));}
      catch(InterruptedException e){}
     }
   public void eat()
   {int n=rand.nextInt(100);
    if(n%2==0){
    synchronized(leftChopstick)
   {   System.out.println(this+"take"+ leftChopstick+",Ready"+rightChopstick);
       synchronized(rightChopstick)
         { System.out.println(this+"take"+rightChopstick+",Dining"); }
                  }
        else
     {synchronized(rightChopstick)
        { System.out.println(this+"take" +rightChopstick+",ready to take"+leftChopstick);
           synchronized(leftChopstick)
            { System.out.println(this+"take"+leftChopstick+",eating"); }
                        }
                  }
        }
   public void run()
   { while(t
      think()
      An        (i);
      eat(
      releas        }
      }
      }
   public class MyPhilo8
  {  public static void main(String[] args)
     {  Chopstick[] chop=new Chopstick[5];
        for(int i=0;i<5;i++)
          { chop[i]=new Chopstick(i); }
                 Philosopher[] philo=new Philosopher[5];
                 for(int i=0;i<5;i++)
                 { philo[i]=new Philosopher(i,chop[i],chop[(i+1)%5]);  }
          }
       }
```

Compiling and running the program in memory, observed that the deadlock will not occur. The philosopher could not adjacent to eat while in the process.

4. Conclusions

The pre-allocation method will cause all resource lower usability. We proposed a solution where it has high efficiency for the system performance and usability in resource. In the same time, we recalculate the bound range between upper and lower to increase resource usability. Even thought it's a simple and tiny model, but it has still an interesting and valuable issue to multi thread/process topic for computer programming.

Acknowledgement

The authors would like to thank the anonymous reviewers for their comments that help improve the manuscript. This work is partially supported by the Project of Fuqing Branch of Fujian Normal University of China under the contract number KY2010030 and KY20110[illegible]5.

References

[1] E. W. Dijkstra: Hierarchical Ordering of Sequential Processes. *Acta Informatica* Vol. 1(1971), pp. 115–138.

[2] K. M. CHANDY and J. MISRA: The Dining Philosopher's problem. *ACM Transactions on Programming Languages and Systems* Vol. 6(1984), pp. 632–646.

[3] C. A. R. Hoare: *Communicating Sequential Processes*. Prentice Hall International, June 2004.

[4] J. Zhan and Y. Guo: A Preliminary Study on the Dining Philosopher's problem. *Fujian Computer* Vol. 3(2008), pp. 78–79.

[5] Wikipedia. Dining philosopher's problem. Website, 2011. http://en.wikipedia.org/wiki/Dining philosophers problem

[6] A. Silberschatz, P. B. Galvin, and G. Gagne, Operating Systems Concepts. John Wiley and Sons Inc., Sixth edition, 19[illegible].

[7] P. Shrestha: Dining Philosopher Problem and Autoresetevent. Pradip's Blog, 2011. http://spradip.files.wordpress.com/2011/04/ic338850.png