# BSS Practicum Week 3 Exercises


- Andre Nanninga
- 12-may-2014


# Chapter 7

> *1.1 There are three major methods for handling deadlock. (See Chapter 7.3.1). But actually there are four. Describe these four methods fior handling deadlock.*

### Deadlock prevention

Prevent a deadlock to occur by constraining how requests for resources can be made.

### Deadlock avoidence

Requiring that the operating system is given additional information about the resources a process requests and its use during it's lifetime.

### Deadlock recovery

An algorithm to determine wether deadlock occur by examining the system state and an algorithm to recover from the deadlock.

### Deadlock ignore

Assume deadlock occur very rarely and simply ignore them. When the system is restarted the deadlocks will be resolved.

> *1.2 Consider the following snapshot of a system.*

| | Allocation | | | | | Max | | | | | Need | | | | | Available | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **A** | **B** | **C** | **D** | | **A** | **B** | **C** | **D** | | **A** | **B** | **C** | **D** | | **A** | **B** | **C** | **D** |
| *P0* | 0 | 0 | 1 | 2 | | 0 | 0 | 1 | 2 | | 0 | 0 | 0 | 0 | | 1 | 5 | 2 | 0 |
| *P1* | 1 | 0 | 0 | 0 | | 1 | 7 | 5 | 0 | | 0 | 7 | 5 | 0 | | | | | |
| *P2* | 1 | 3 | 5 | 4 | | 2 | 3 | 5 | 6 | | 1 | 0 | 0 | 2 | | | | | |
| *P3* | 0 | 6 | 3 | 2 | | 0 | 6 | 5 | 2 | | 0 | 0 | 2 | 0 | | | | | |
| *P4* | 0 | 0 | 1 | 4 | | 0 | 6 | 5 | 6 | | 0 | 6 | 4 | 2 | | | | | |

> Answer the following questions using the banker's algorithm:
>
> *a. What is the content of the matrix Need?*

See table above.

> *b. Is the system in a safe state?*

Yes, with the resources available P0 is able to run after which `1, 5, 3, 2` resources are available. With these resources P2 is able to run. After this process `3, 8, 8, 8` resources are available. P1 is able to run with these available resources after which `4, 15, 13, 8` resources are available. Next P3 can run after which `0, 21, 18, 10` resources are available. Lastly P4 is able to run.

All processes are able to run so the system is in a safe state.

> *c. If a request from process Pi arrives for (0, 4, 2, 0), can the request be granted immediately?*

Yes, the need of process Pi is `0, 4, 2, 0` and the resource available are `1, 5, 2, 0`. This means that enough resources are available for the process so it could run.

> *1.3 Java's locking mechanism (the synchronized statement) is considered reentrant. That is, if a thread acquires the lock for an object (by invoking a synchronized method or block), it can enter other synchronized methods or blocks for the same object. Explain how deadlock would be possible if Java's locking mechanism were not reentrant.*

Without reentrant locking mechanisms a thread would be able to lock itself.

# Programming Exercise

> *2.1 Write a multithreaded program that implements the banker's algorithm discussed in Section 7.5.3. Create n threads that request and release resources from the bank. The banker will grant the request only if it leaves the system in a safe state. You may write this program using Bibliographical Notes 271 either Pthreads or Win32 threads. It is important that access to shared data is safe from concurrent access. Such data can be safely accessed using mutex locks, which are available in both the Pthreads and Win32 API. Coverage of mutex locks in both of these libraries is described in "'producer-consumer problem" project in Chapter 6.*

```
import java.util.*;

public class BankImpl implements Bank {

  private int[] resources;
  private HashMap<Integer, CustomerResources> customers;

  public BankImpl(int[] resources) {
    this.resources = resources;

    customers = new HashMap<Integer, CustomerResources>();
  }

  public void addCustomer( int threadNum, int[] maxDemand ) {
    customers.put(threadNum, new CustomerResources(threadNum, maxDemand.clone()));
  }

  public synchronized boolean requestResources( int threadNum, int[] request ) {
    CustomerResources customer = customers.get(threadNum);
    boolean allowRequest = true;

    if( !isSafeState( threadNum, request ) ) {
      return false;
    }

    for(int i = 0; i < resources.length; i++) {
      if(request[i] > resources[i] ||
        request[i] > customer.max[i] ||
        request[i] > customer.max[i] - customer.allocated[i]) {
        allowRequest = false;
      }
    }

    if(allowRequest) {
```

```
    for(int i = 0; i < resources.length; i++) {
      resources[i] -= request[i];
      customer.allocated[i] += request[i];
    }

    return true;
  }

  return false;
}

public synchronized void releaseResources( int threadNum, int[] release ) {
  CustomerResources customer = customers.get(threadNum);

  for(int i = 0; i < resources.length; i++) {
    if(customer.allocated[i] - release[0] < 0) {
      return;
    }

    resources[i] += release[i];
    customer.allocated[i] -= release[i];
  }
}

public void getState() {
  System.out.print("Resources:   ");
  System.out.println(Arrays.toString(resources));

  System.out.println("Allocated: ");
  for(CustomerResources customer : customers.values()) {
    System.out.println("Customer: " + customer.id + ", " + Arrays.toString(customer.allocat
  }

  System.out.println("Max: ");
  for(CustomerResources customer : customers.values()) {
    System.out.println("Customer: " + customer.id + ", " + Arrays.toString(customer.max));
  }

  System.out.println("Need: ");
  for(CustomerResources customer : customers.values()) {
    int[] need = new int[customer.allocated.length];
    for(int i = 0; i < customer.allocated.length; i++) {
      need[i] = customer.max[i] - customer.allocated[i];
    }
    System.out.println("Customer: " + customer.id + ", " + Arrays.toString(need));
  }
}
```

```
private boolean isSafeState( int threadNum, int[] request ) {
  CustomerResources customer = customers.get(threadNum).clone();
  int[] resources = this.resources.clone();
  boolean[] finish = new boolean[customers.size()];

  for(int i = 0; i < resources.length; i++) {
    if( request[i] > resources[i] ) {
      return false;
    }
  }

  for(int i = 0; i < request.length; i++) {
    resources[i] -= request[i];
    customer.allocated[i] += request[i];
  }

  for(int i = 0; i < this.customers.size(); i++) {
    for(int j = 0; j < this.customers.size(); j++) {

      if( finish[j] ) {
        continue;
      }

      boolean canFinish = true;

      for(int k = 0; k < resources.length; k++) {
        if(customer.max[k] - customer.allocated[k] > resources[k]) {
          canFinish = false;
          break;
        }
      }

      if(canFinish) {
        finish[j] = true;

        for(int k = 0; k < resources.length; k++) {
          resources[k] += customer.allocated[k];
        }
      }
    }
  }

  for(int i = 0; i < customers.size(); i++) {
    if(!finish[i]) {
      return false;
    }
  }
```

```
    return true;
  }
}

class CustomerResources {

  public int id;
  public int[] allocated;
  public int[] max;
  public boolean finished;

  public CustomerResources( int id, int[] max ) {
    this.id = id;
    this.max = max;
    this.allocated = new int[max.length];
    this.finished = false;
  }

  public CustomerResources clone() {
    CustomerResources customer = new CustomerResources(id, max.clone());
    customer.allocated = allocated.clone();
    customer.finished = finished;

    return customer;
  }
}
```