COM1032 Mobile Computing
Lab 6
Threads Deadlock Prevention

Purpose:

The purpose of this lab session is to familiarise yourself with the Deadlock, how to avoid them, and how to resolve them.

Aim

By the end of the lab you will be able to:

- Write a Java Program to prevent possible deadlocks.
- Test and configure Linux Thread Scheduling Policies.
- Hints on employing concurrency in your OS Simulator Design

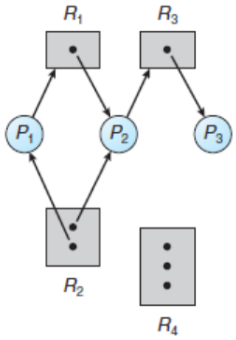
Bankers Algorithm

To appreciate the need of avoiding deadlocks, check the "DeadlockExample.java" file on SurreyLearn to see how two threads can compete on acquiring the two locks in opposite orders. You might run several times to deadlock the threads. In this exercise will be required to write a Java program that implements the banker's algorithm discussed in the following section. 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 is denied if it leaves the system in an unsafe state.

Resource-Allocation-Graph:

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes: $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the active processes in the system, and $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource. A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i . A directed edge $P_i \rightarrow R_j$ is called a **request edge**; a directed edge $R_j \rightarrow P_i$ is called an **assignment edge**. Pictorially, we represent each process *Pi* as a circle and each resource type R_j as a rectangle.



In the lecture we will discuss the resource-allocation-graph algorithm, and why it is not applicable to a resource allocation system with multiple instances of each resource type. The banker's algorithm is applicable to such a system but is less efficient than the resource-allocation graph scheme.

The Banker's Deadlock Avoidance Algorithm:

The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

- **Available**. A vector of length m indicates the number of available resources of each type. If $Available[j]$ equals k, then k instances of resource type R_j are available.

- **Max**. An $n \times m$ matrix defines the maximum demand of each process. If $Max[i][j]$ equals k, then process P_i may request at most k instances of resource type R_j.

- **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If $Allocation[i][j]$ equals k, then process P_i is currently allocated k instances of resource type R_j.

- **Need**. An $n \times m$ matrix indicates the remaining resource need of each process. If $Need[i][j]$ equals k, then process P_i may need k more instances of resource type R_j to complete its task. Note that $Need[i][j]$ equals $Max[i][j] - Allocation[i][j]$.

These data structures vary over time in both size and value.

To simplify the presentation of the banker's algorithm, we next establish some notation. Let X and Y be vectors of length n. We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, ..., n$. For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y \leq X$. In addition, $Y < X$ if $Y \leq X$ and $Y \neq X$.

We can treat each row in the matrices *Allocation* and *Need* as vectors and refer to them as $Allocation_i$ and $Need_i$. The vector $Allocation_i$ specifies the resources currently allocated to process P_i; the vector $Need_i$ specifies the additional resources that process P_i may still request to complete its task.

Safety Algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let *Work* and *Finish* be vectors of length *m* and *n,* respectively. Initialize
Work = *Available* and *Finish*[*i*] = *false* for *i* = 0, 1, ..., *n* − 1.

2. Find an index *i* such that both

a. *Finish*[*i*] == *false*

b. *Needi* ≤ *Work*

If no such *i* exists, go to step 4.

3. *Work* = *Work* + *Allocationi*
Finish[*i*] = *true*
Go to step 2.

4. If *Finish*[*i*] == *true* for all *i,* then the system is in a safe state.

This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted. Let *Requesti* be the request vector for process *Pi*. If *Requesti* [*j*] == *k*, then process *Pi* wants *k* instances of resource type *Rj* .When a request for resources is made by process *Pi*, the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$Available = Available - Request_i$;
$Allocation_i = Allocation_i + Request_i$;
$Need_i = Need_i - Request_i$;

If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $Request_i$, and the old resource-allocation state is restored.

An Illustrative Example

To illustrate the use of the banker's algorithm, consider a system with five processes P_0 through P_4 and three resource types *A, B,* and *C.* Resource type *A* has ten instances, resource type *B* has five instances, and resource type *C* has seven instances. Suppose that, at time T_0, the following snapshot of the system has been taken:

	Allocation	*Max*	*Available*

	A B C	*A B C*	*A B C*
P_0	*010*	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix *Need* is defined to be *Max* − *Allocation* and is as follows:

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

We claim that the system is currently in a safe state. Indeed, the sequence $<P_1, P_3, P_4, P_2, P_0>$ satisfies the safety criteria. You can apply the safety algorithm to check this. Suppose now that process P_1 requests one additional instance of resource type *A* and two instances of resource type *C,* so $Request_1 = (1,0,2)$. To decide whether this request can be immediately granted, we first check that $Request_1 \leq Available$—that is, that $(1,0,2) \leq (3,3,2)$, which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	*010*	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies the safety requirement. Hence, we can immediately grant the request of process P_1.

You should be able to see, however, that when the system is in this state, a request for (3,3,0) by P_4 cannot be granted, since the resources are not available. Furthermore, a request for (0,2,0) by P_0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

The Bank

The bank will employ the strategy outlined by the Banker's algorithm introduced in the previous section, whereby it will consider requests from *n* customers for *m* resources. The bank will keep track of the resources using the following data structures:

```
int numberOfCustomers; // the number of customers - n
int numberOfResources; // the number of resources - m
int[] available; // the available amount of each resource
int[][] maximum; // the maximum demand of each customer
int[][] allocation; // the amount currently allocated to each customer
int[][] need; // the remaining needs of each customer
```

The functionality of the bank appears in the interface shown in the code below and available on SurreyLearn as Bank.java.The implementation of this interface will require adding a constructor that is passed the number of resources initially available. For example, suppose we have three

resource types with 10, 5, and 7 resources initially available. In this case, we can create an implementation of the interface using the following technique:

```
Bank theBank = new BankImpl(10,5,7);
```

The bank will grant a request if the request satisfies the safety algorithm outlined above. If granting the request does not leave the system in a safe state, the request is denied.

Testing Your Implementation

You can use the file TestHarness.java, which is available on SurreyLearn, to test your implementation of the Bank interface. This program expects the implementation of the Bank interface to be named BankImpl and requires an input file containing the maximum demand of each resource type for each customer. For example, if there are five customers and three resource types, the input file might appear as follows (a sample input file is available on SurreyLearn):

```
7,5,3
3,2,2
9,0,2
2,2,2
4,3,3
```

This indicates that the maximum demand for customer$_0$ is 7, 5, 3; for customer$_1$, 3, 2, 2; and so forth. Since each line of the input file represents a separate customer, the addCustomer() method is to be invoked as each line is read in, initializing the value of maximum for each customer. (In the above example, the value of maximum[0][] is initialized to 7, 5, 3 for customer 0; maximum[1][] is initialized to 3, 2, 2; and so forth.)

Furthermore, TestHarness.java also requires the initial number of resources available in the bank. For example, if there are initially 10, 5, and 7 resources available, we invoke TestHarness.java as follows:

```
java TestHarness infile.txt 10 5 7
```

where infile.txt refers to a file containing the maximum demand for each customer followed by the number of resources initially available. The available array will be initialized to the values passed on the command line.

```
public interface Bank {

        public static final int NUMBER_OF_CUSTOMERS = 5;

        /**
         * Add a customer to the bank.
         * @param customerNumber The number of the customer being added.
         * @param maxDemand The maximum demand for this customer.
         */
        public void addCustomer(int customerNumber, int[] maximumDemand);

        /**
         * Outputs the available, allocation, max, and need matrices.
         */
        public void getState();

        /**
         * Make a request for resources.
         * @param customerNumber The number of the customer being added.
         * @param request The request from this customer.
         * @return  true The request is granted.
```

```
     * @return  false The request is not granted.
     */
    public boolean requestResources(int customerNumber, int[] request);

    /**
     * Release resources.
     * @param customerNumber The number of the customer being added.
     * @param release The resources to be released.
     */
    public  void releaseResources(int customerNumber, int[] release);
}
```

Linux CPU Scheduling:

For the curious about kernel implementation details, in the Additional Resources in SurreyLearn you will find a thesis discussing Linux Thread management and scheduling " LinuxScheduling.pdf ". As discussed in the first lab, you can find out the scheduling policy of the running processes using the following command:

```
ps -e -o s,pid,cls,pri | grep ^R | awk -v sq="'" '{print "pid",$2,sq,"s current scheduling p
```

If can give you an output as follows:

```
pid 8456 ' s current scheduling policy: TS
pid 8456 ' s current priority: 19
pid 12552 ' s current scheduling policy: TS
pid 12552 ' s current priority: 19
```

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-cpu-scheduler

Linux I/O Scheduler:

Linux kernel supports CFS (Completely Fair Scheduler), Noop, and Deadline scheduling policies.

CFS It handles CPU resource allocation for executing processes, and aims to maximize overall CPU utilization while also maximizing interactive performance.
https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/ch06s04#idm140449009389088

The Noop I/O scheduler implements a simple first-in first-out (FIFO) scheduling algorithm. Merging of requests happens at the generic block layer, but is a simple last-hit cache. If a system is CPU-bound and the storage is fast, this can be the best I/O scheduler to use.
https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/ch06s04s03

The deadline I/O scheduler attempts to provide a guaranteed latency for requests. It is important to note that the latency measurement only starts when the requests gets down to the I/O scheduler (this is an important distinction, as an application may be put to sleep waiting for request descriptors to be freed). By default, reads are given priority over writes, since applications are more likely to block on read I/O. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/ch06s04s02

Find out the scheduling policy by running the following command, based on what is the name of your disk (sda, hda, … etc):

```
cat /sys/block/sda/queue/scheduler
```

The result might be:

```
noop [deadline] cfq
```

To change the scheduler on the fly (will be back to default on system reboot by:

```
sudo echo noop > /sys/block/hda/queue/scheduler
```
You can change noop to cfq or deadline. To make your changes persistence, update the GRUB configuration file.

```
sudo nano /etc/default/grub
```
change the line:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
```
To

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash elevator=noop"
```

JVM Scheduling:

The Java Threads default priority and scheduling is discussed in the lecture slides. However, in large scale applications, you might need to use Executors. Below are links to a tutorial on executors, followed by the Java documentation.

https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/concurrent/Executors.html

Final Coursework:

Check the I/O Thread (fourth thread in the suggested OS Simulator in Lab 5) whether it needs to apply the Banker's algorithm or the Resource Allocation Graph Algorithm. Add the required implementation / interfacing with the sample Bank Interface Implementation that will be provided next week to your coursework implementation. You are free to choose another resource allocation algorithm as you see feasible to your overall OS Simulator Design.

Based on your choices, check how the input file to your resources in the system, such as the infile.txt example used in this code, resembles the Device Tree Binary/BLOB, which is the modern ATAGs (ARM system format containing data structures that describes the hardware provided to the kernel at boot time). This was explained in Week 2 in the additional material section in Surrey Learn in a document called "RaspberryBoot.pdf". You can elaborate on this as you wish from the theoretical material you will learn when you study the OS File and Input/Output subsystems. You will be given more options to consider when you study the memory management subsystem as well. So these hints are not obligatory to attempt.