DATA STRUCTURES AND ALGORITHMS

PRACTICE 2: GRAPHS

2021-2022

1. Introduction

In this practice, we will use graphs to model the state of a system regarding allocations and requests between processes and resources. These graphs are called resource allocation graphs (RAG) and are used to analyze deadlocks.

As an example, in operating systems, many processes run simultaneously. These processes request the use of resources, some of which may be exclusive access.

A resource can be a file, a device,... that is, anything that a process may need. Therefore, when a process requests exclusive access to a resource that is being used by another process, the former will have to wait for the latter to release it.

In the following example, there are two processes that want to use a computer's scanner and printer

Process 1	**Process 2**
Request scanner Request printer Uses printer and scanner Frees printer Frees scanner	Request printer Request scanner Uses printer and scanner Frees scanner Frees printer

Although it may seem that these processes will run normally, process 1 will likely request the scanner, and before it also asks for the printer, it will be overtaken by process 2. If this happens, neither process will be able to move forward, because process 1 will be waiting for process 2 to release the printer, and process 2 will be waiting for process 1 to release the scanner.

1.1. Deadlocks

The above situation is called a deadlock, and if it occurs in a system that has no strategy to solve it, the two processes will be blocked, and no other process will be able to use the scanner or the printer. This will cause more and more processes (requesting those resources) to be blocked.

Deadlocks can also occur in databases because they execute concurrent transactions that read from and write to the same database records. In this case, the processes are the transactions and the resources are the rows of the database tables.

A database transaction is a sequence of reads and writes that must be able to terminate altogether (commit) or be left as if nothing has happened (rollback).

According to Coffman, the following conditions must be met for an interlock to occur:

- **Mutual exclusion**: At least one resource must be held in a non-shareable mode; that is, only one process at a time can use the resource.
- **Hold and wait (resource holding)**: a process is currently holding at least one resource and requesting additional resources which are being held by other processes.
- **No preemption**: a resource can be released only voluntarily by the process holding it.
- **Circular wait**: each process must be waiting for a resource that is being held by another process, which in turn is waiting for the first process to release the resource

1.2. Deadlock handling

These are the strategies that can be applied when facing deadlocks.

- **Ignoring deadlock**: in this approach, it is assumed that a deadlock will never occur (or have a very low probability to happen). This is the strategy followed by conventional operating systems.

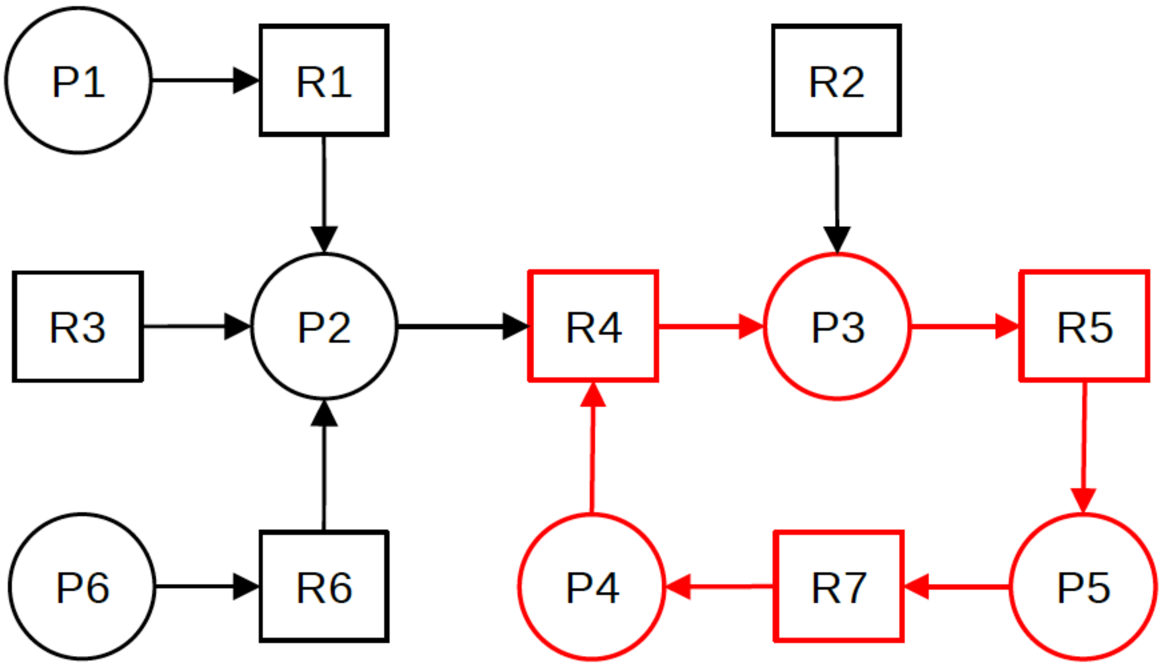
- **Detection and recovery**: under deadlock detection, deadlocks are allowed to occur. Then the system's state is examined to detect that a deadlock has occurred, and subsequently, it is corrected. For example, in databases, it is possible to detect a deadlock when a transaction requests a resource allocated to another transaction. In such a case, the "culprit" transaction can be interrupted by informing the user that an interlock has occurred. The user can then re-launch the transaction at a later time.

- **Deadlock Prevention**: prevents one of the four Coffman conditions from occurring (if possible). For example, operating systems often use print queues so that a printer is only assigned exclusively to the print service. Therefore, no interlocking with printers will occur.

- **Deadlock prediction**: the operating system tries to detect whether the probability of deadlock is high, and if so, it tries to run the processes involved in a way that avoids deadlock. The problem with this strategy is that there may be no way to predict how the access to resources will be.

1.3. Resource allocation Graphs (RAG)

Although there are several interlock detection strategies, in this practice, we will use a resource allocation graph, which consists of a directed graph in which each node can be either a process or a resource, and edges indicate whether a process is waiting for a resource (edge from the process to the resource) or whether a resource is assigned to a process (edge from the resource to the process).



In the above graph, the processes are represented by circles and the resources are represented by rectangles. The deadlock is marked in red. It has occurred when one of the processes 3, 4 or 5 has requested a resource. Although only these processes are part of the interlock, no other process can move forward, because process 2 is waiting for resource 4 and owns 1 and 6 which have been requested by process 1 and 6, which are also waiting.

2. Interfaz PRGraph

To implement what has been explained in the introduction and to develop the interlock detection algorithm, the following interface has to be implemented in the class PRGraphImpl:

```
// Resource allocation graph
public interface PRGraph
{
   // Adds a process to the graph
   void addProcess(String name);

   // Adds a resource to the graph
   void addResource(String name);

   // Requests a reqsource
   void open(String process, String resource)
          throws DeadlockException;

   // Frees a resource
   void close(String process, String resource);
}
```

Note that when a resource is released (close), if there is a process waiting for that resource, it must be assigned to it.

3. Main Class

The Main class must be executable. It shall accept a text file as a parameter to simulate a sequence of resource allocations and releases.

The file may have blank lines and text appearing after the # character is considered a comment. One of the following instructions may be entered on each line:

- **Create a process:** PROCESS process_name
- **Create a resource:** RESOURCE resource_name
- **Request a resource:** OPEN process_name resource_name
- **Free a resource:** CLOSE process_name resource_name

These instructions may be in any order. The PRGraphImpl class shall check that a process or resource has been created previously to refer it.

As the file is read, the corresponding functions of the PRGraphImpl class shall be called to check if there is an interlock. If a deadlock occurs, execution shall be interrupted due to a DeadlockException.

The reading of the file can be done as explained in point 6 of practice 1. The following should be done for each line:

- Check if the # character exists with the indexOf function of the String class to discard the text from its position, using the substring function.
- Create a StringTokenizer object with the rest of the string and read each token (word).
- The first token shall be PROCESS, RESOURCE, OPEN or CLOSE (in upper case). Then there must be one more token in the first two instructions (PROCESS and RESOURCE and two in the last two instructions(OPEN and CLOSE)
- In case the file has a wrong format, an IOException must be thrown informing about the error and the line number of the file.

The following example file corresponds to the deadlock example in point 1.3:

```
PROCESS P3
RESOURCE R2
RESOURCE R4
OPEN P3 R2
OPEN P3 R4

PROCESS P4
RESOURCE R7
OPEN P4 R7

PROCESS P5
RESOURCE R5
OPEN P5 R5

PROCESS P2
RESOURCE R1
RESOURCE R3
RESOURCE R6
OPEN P2 R1
OPEN P2 R3
OPEN P2 R6
OPEN P2 R4

PROCESS P1
OPEN P1 R1

PROCESS P6
OPEN P6 R6

OPEN P3 R5
OPEN P5 R7
# CLOSE P3 R4  # Will avoid deadlock
OPEN P4 R4     # Creates deadlock
```

4. Deliverables

To obtain the highest mark, the following work must be done:

- Implementation of the interface PRGraph in the class PRGraphImpl.

- Implementation of the toString function for PRGGraphImpl class. It has to show processes and resources. Indicating for each process which resources it owns and if it's waiting for any. Also, indicating for each resource if it's assigned to any process.
- PRGGraphImpl should have a private method to check for deadlocks when the open function is called.
- PRGraphImpl methods must check that processes and resources have been created previously. Otherwise, an IllegalArgumentException should be thrown explaining the cause in the exception text.
- PRGraphImpl close method should check that the process owns the resource that will be freed. Otherwise, an IllegalStateException must be thrown explaining the cause in the exception text.
- Implement the Main class that accepts a text file as a parameter. The text file will define the simulation as stated in the example.
- When an error is detected, the line that creates the error must be indicated in the exception.
- Document every class using JavaDoc

The source code (.java files) must be delivered packaged in a single compressed file
(.zip, .tgz, etc.) through a Poliformat task.
All classes (.java files) must be in the package p2:

```
package p2; // lowercase p!
...
```