CPSC 457 - Assignment 4

Due date: **Friday, March 29, 2019 at 11:30pm**.
Individual assignment. No group work allowed.
Weight: 5% of the final grade.

Q1 - Programming question (20 marks)

Write a program that implements the Banker's Algorithm. Your program will determine whether there is a safe execution sequence for a given set of processes, and a request. The process information will be provided in a configuration file, specified as a command-line argument. The configuration file will contain the number of processes (`numProc`), the number of resource types (`numResourceTypes`), currently available resources (`available`), current resource allocation for all processes, the maximum resources required by each process, and a request (`request`) by process 'i'. For instance, the example we discussed during lectures, can be translated into the following configuration file (`config1.txt`):

```
numProc = 5
numResourceTypes = 3
available = <3 3 2>
P0 <0 1 0> <7 5 3>
P1 <2 0 0> <3 2 2>
P2 <3 0 2> <9 0 2>
P3 <2 1 1> <2 2 2>
P4 <0 0 2> <4 3 3>
request 1 = <1 0 2>
```

The last line represents a request by process #1 for resources (1,0,2). If you run your program on the above configuration file, the output should look as follows:

```
$ ./banker config1.txt
Grant request <1 0 2> from P1.
Sequence: P1, P3, P0, P2, P4.
```

If the request can be granted, your program needs to output a possible execution sequence, as above. If the request cannot be granted, the output of your program should look like this:

```
$ ./banker config2.txt
Reject request <1 0 2> from P1.
Reason: request would result in unsafe state.
```

If the request cannot be granted, the program should output the reason for rejection. Possible reasons for rejecting a request include:

- request would result in an unsafe state,
- request is invalid (exceeding declared max for process),
- not enough resources available.

You are free to use the `banker.cpp` code in the Appendix 1 as a starting point. If you do, all you have to do is implement the `Banker::isSafe()` method.

Q2 - Programming question (20 marks)

For this part you will be writing a program that will examine multiple system states consisting of several processes and resources. For each state your program will print out which processes are deadlocked. You will assume a single instance per resource type. Hint: you should implement a cycle-detection algorithm.

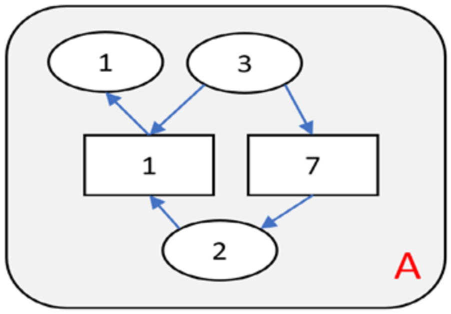
Your program will read the descriptions of each system state from standard input. The input will be line-based. There will be 3 types of lines, represeting an assignment edge, request edge and end of state description.

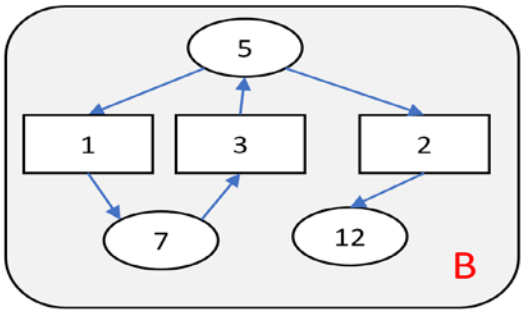
- a line `N -> M` represents a request edge: process N is requesting resource M;
- a line `N <- M` represents an assignment edge: process N already holds resource M;
- a line that starts with `#` represents the end of current state description.

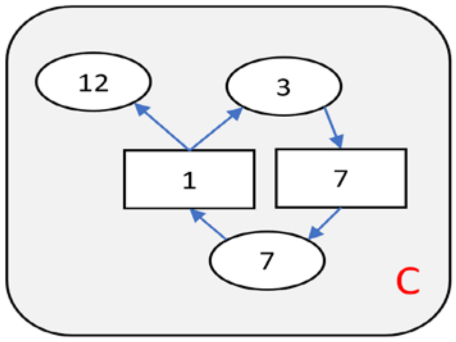
The “N” and “M” above represent non-negative integers. The overall algorithm for your program should look something like this:

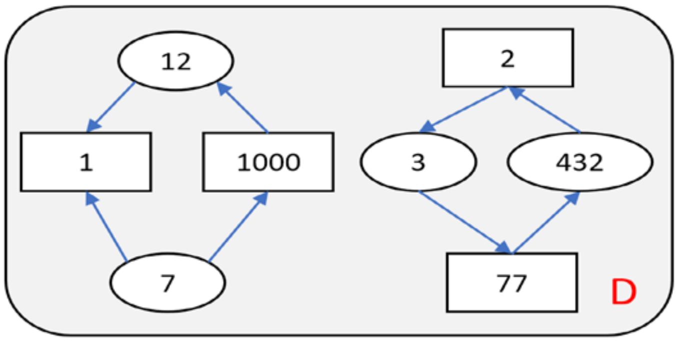
```
loop:
    set graph to empty
    loop:
        read line from stdin
        if line starts with ‘#’, or EOF reached:
            compute, sort and display deadlocked processes from graph
            break
        else:
            convert line to edge and add it to graph
    if EOF was encountered:
        break
```

As an example, consider the following 4 system states:









The 4 system states above would be represented by the input below, and the output your program should produce on this input is shown on the right.

Input:

```
1 <- 1
3 -> 1
3 -> 7
2 -> 1
2 <- 7
# end of A
5 -> 1
5 <- 3
5 -> 2
7 <- 1
7 -> 3
12 <- 2
# end of B
12 <- 1
3 <- 1
3 -> 7
7 -> 1
7 <- 7
# end of C
12 -> 1
12 <- 1000
7 -> 1
7 -> 1000
3 -> 77
432 <- 77
432 -> 2
3 <- 2
```

Output:

```
Deadlocked processes: none
Deadlocked processes: 5 7
Deadlocked processes: 3 7
Deadlocked processes: 3 432
```

Notice there is no explicit end of state line `#` at the end of the input above. That is implied by the end-of-file. You may assume the following limits on input:

- process numbers will be in range [0 … 100000];
- resource numbers will be in range [0 … 100000];
- number of edges per state description will be in range [1 … 100000];
- number of states per input will be in range [1 … 20].

Write your solution in C or C++ and name your source file either `deadlock.c` or `deadlock.cpp`.

Submission

You should submit 2 files for this assignment:

- Your solution to Q1 in a file called `banker.c` or `banker.cpp.`
- Your solution to Q2 in a file called `deadlock.c` or `deadlock.cpp`.

Since D2L will be configured to accept only a single file, you will need to submit an archive, eg. `assignment4.tgz`. To create such an archive, you can use a command similar to this:

```
 $ tar czvf assignment4.tgz deadlock.c banker.cpp
```

General information about all assignments:

- All assignments must be submitted before the due date listed on the assignment. Late assignments or components of assignments will not be accepted for marking without approval for an extension beforehand. What you have submitted in D2L as of the due date is what will be marked.

- Extensions may be granted for reasonable cases, but only by the course instructor, and only with the receipt of the appropriate documentation (e.g. a doctor's note). Typical examples of reasonable cases for an extension include: illness or a death in the family. Cases where extensions will not be granted include situations that are typical of student life, such as having multiple due dates, work commitments, etc. Forgetting to hand in your assignment on time is not a valid reason for getting an extension.
- After you submit your work to D2L, make sure that you check the content of your submission. It's your responsibility to do this, so make sure that you submit your assignment with enough time before it is due so that you can double-check your upload, and possibly re-upload the assignment.
- All assignments should include contact information, including full name, student ID and tutorial section, at the very top of each file submitted.
- Assignments must reflect individual work. Group work is not allowed in this class nor can you copy the work of others. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k-5.html.
- You can and should submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It's better to submit incomplete work for a chance of getting partial marks, than not to submit anything.
- Only one file can be submitted per assignment. If you need to submit multiple files, you can put them into a single container. The container types supported will be ZIP and TAR. No other formats will be accepted.
- Assignments will be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA then you can contact your instructor.

Appendix 1 – banker.cpp for Q1

```
/*
 * banker.cpp
 *
 * Student Name:
 * Student Number:
 *
 * Class: CPSC 457 Winter 2019
 * Instructor: Pavol Federl
 *
 * Copyright 2017 University of Calgary. All rights reserved.
 */

#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>
#include <algorithm>

```

```
using namespace std;

class Banker
{
private:
    int numProc;      // the number of processes
    int numResources; // the number of resources
    int * available;  // number of available instances of each resource
    int ** max;       // the max demand of each process, e.g., max[i][j] = k
                      // means process i needs at most k instances
                      // of resource j
    int ** allocation;// number of resource instances already allocated
    int ** need;      // number of resource instances needed by each process

public:

    /* Initializing the vectors and matrixes for the Banker's Algorithm.
     * Takes ownership of all arrays.
     *
     * @param avail  The available vector
     * @param m      The max demand matrix
     * @param alloc  The allocation matrix
     * @param p      The number of processes
     * @param r      The number of resources
     */
    Banker (int * avail, int ** m, int ** alloc, int p, int r) {
        numProc = p;
        numResources = r;
        // Setup the available vector, the max matrix, and the
        // allocation matrix
        available = avail;
        max = m;
        allocation = alloc;
        // Initialize the need matrix
        need = new int*[numProc];
        for (int i = 0; i < numProc; i++)
            need[i] = new int[numResources];
    }

    /* Destroy the vectors and matrixes
     */
    ~Banker() {
        numProc = 0;
        numResources = 0;

        // Free all allocated memory space
        delete[] available;
        for (int i = 0; i < numProc; i++)
        {
            delete[] need[i];
            delete[] max[i];
            delete[] allocation[i];
        }
        delete[] need;
        delete[] max;
        delete[] allocation;
    }

    /* Check whether it is safe to grant the request
     * @param pid    The process that is making the request
     * @param req    The request
     * @param sequenceOrReason  The safe execution sequence returned
     *               by the algorithm or reason for rejection
     * @return Whether granting the request will lead to a safe state.
     */
    bool isSafe (int pid, int * req, string & sequenceOrReason) {
        sequenceOrReason = "Not implemented yet.";
        return false;
    }
};

int main (int argc, char * const argv[]) {
    ifstream config;       // Configuration file
    string conffile;       // The configuration file name
    int numProc;           // The number of processes
    int numResources;      // The number of resources
    string sequenceOrReason; // The execution sequence returned by
                             // the Banker's Algorithm
    int i, j, index;       // Indices for the vectors and matrixes
    int pid;               // The ID of the process that is making the
                           // request
    string reqStr;         // The request vector in string format

    // Read in the config file name from the commanda-line arguments
    if (argc < 2) {
        cout << "Usage: banker <config file>\n";
        return 0;
    }
    else {
        conffile = argv[1];
    }

    // Open the file
    config.open(conffile.c_str());

    // Get the number of process and the number of resources
    string line, var, equal; // strings for parsing lines in cfg. file
    getline(config, line);
    istringstream iss(line);
    iss >> var >> equal >> numProc;     // Get the number of processes
    iss.clear();

    getline(config, line);
    iss.str(line);
    iss >> var >> equal >> numResources;    // Get the number of resources
    iss.clear();

    // Create the available vector, the max matrix, and the allocation
    // matrix according to the number of processes and the number of
    // resources
    int * available = new int[numResources];
    int ** max = new int*[numProc];
    int ** allocation = new int*[numProc];
    for (int i = 0; i < numProc; i++)
    {
        max[i] = new int[numResources];
        allocation[i] = new int[numResources];
    }

    // Get the available vector
    getline(config, line);
    replace(line.begin(), line.end(), '<', ' ');  // Remove "<" and ">"
    replace(line.begin(), line.end(), '>', ' ');
    iss.str(line);
    iss >> var >> equal;
    for (j = 0; j < numResources; j++) // Read in the "available" vector
        iss >> available[j];
    iss.clear();

    // Get the max matrix and the allocation matrix
    for (i = 0; i < numProc; i++)
    {
        getline(config, line);
        replace(line.begin(), line.end(), '<', ' ');
        replace(line.begin(), line.end(), '>', ' ');
        iss.str(line);
        iss >> var;
        index = atoi(&var.at(1));            // Get the process ID
        if (index < 0 || index >= numProc)
        {
            cerr << "Invalid process ID: " << var << endl;
            return 0;
        }
        // Get the number of resources allocated to process "index".
        for (j = 0; j < numResources; j++)
            iss >> allocation[index][j];

        // Get the max allocation to process "index".
        for (j = 0; j < numResources; j++)
            iss >> max[index][j];
        iss.clear();
    }

    // Get the request vector
    int * request = new int[numResources];
    getline(config, line);
    reqStr = line.substr(line.find('<'),
                         line.find('>') - line.find('<') + 1);
    replace(line.begin(), line.end(), '<', ' ');
    replace(line.begin(), line.end(), '>', ' ');
    iss.str(line);
    iss >> var >> pid >> equal;
    for (j = 0; j < numResources; j++)  // Read in the "request" vector
        iss >> request[j];
    iss.clear();
    // Check the request using the Banker's algorithm.
    Banker * banker = new Banker(
             available, max, allocation, numProc, numResources);
    if (banker -> isSafe(pid, request, sequenceOrReason))
        cout << "Grant request " << reqStr << " from P" << pid << ".\n"
             << "Sequence: " << sequenceOrReason << ".\n";
    else
        cout << "Reject request " << reqStr << " from P" << pid << ".\n"
             << "Reason: " << sequenceOrReason << "\n";
}
```