Programming Assignment #2

CSci 530, Summer 2015

Dates:

Assigned:	Monday June 22, 2015
Due:	Friday July 3, 2015 (before Midnight)

Objectives:

- Implement the Banker's algorithm for deadlock avoidance.
- Learn more about the details of the deadlock avoidance algorithm. Better understand how save vs. unsafe states are determined.
- Use C/C++ to work with 2 dimensional array representations of our system process and resource states and vectors.

Description:

In programming assignment #2 you are to implement the Banker's algorithm described in the Stallings text chapter 6. You will be implementing the pseudo-code shown in figure 6.9 parts a, b & c.

I have given you some code to get you started. The code, p2-start.c, defines a State structure as shown in Figure 6.9a. The code also implements reading and filling in this state structure from a file, in the function `readSystemState()`. Calling this function, will open the file, and read in the claim **C** allocation **A**, resource **R** and available **V** vectors/matrices from the file into the State structure. This function also reads in a process and its new allocation request.

To complete this assignment, you need to implement the main logic in the main function, as shown in Figure 6.9b. The result of calling your program with a file that contains a valid system state and a process allocation request is one of 4 outputs: ERROR, SUSPEND, SAFE or UNSAFE (see commented pseudo-code at end of p2-start.c).

If the request is valid (doesn't exceed the processes claimed needs), and can possibly be met, you then need to implement the safe() function as described in Figure 6.9c. As shown, your safe function should take a State as its single parameter, and should return a boolean result.

Along with the p2-start.c code, there are 4 example simulation files included in the handout: state-error.sim, state-suspend.sim, state-safe.sim and state-unsafe.sim. The names should be self-explanatory and if given to a correctly running programming should give the indicated expected result for the given state and allocation request. The safe and unsafe examples are basically the ones covered in Figures 6.7 and 6.8 of our text book.

State simulation file formats

As mentioned, we have provided a p2-start.cpp template that can open up and read in the files containing the system claim, allocation, resource and available matrix/vector information. Here we discuss a bit more the format of these file.

The input files needed for this assignment contain the system information for the current state of a system. The input files are read in from the `readSystemState()` function. The format of the files needed for input look like this:

```
m n

R1 R2 R3 ... Rm

V1 V2 V3 ... Vm

C11 C12  ... C1m
C21 C22  ... C2m
...
Cn1 Cn2  ... Cnm

A11 A12  ... A1m
A21 A22  ... A2m
...
An1 An2  ... Anm

p

Q1 Q2 Q3 ... Qm
```

Here m represents the number of resources in the system, and n represents the number of processes in the system. The contents of the file then has two vectors, the resource vector **R** followed by the available vector **V**. Then the claim matrix **C** and the allocation matrix **A** come next in the file, these are n rows by m column matrices, and are read into 2 dimensional arrays in the example C code provided. These 4 sets of values represent an initial state of the system. The Banker's algorithm works by then looking at a new request, and determining if it is safe to grant that request or not. The request being made is given at the end of the file, where p represents the process number making the request, and **Q** is the vector of requests that the process is making.

For example, the initial state shown on page 275 Figure 6.8a of our textbook looks like this:

```
3 4

9 3 6

1 1 2

3 2 2
6 1 3
3 1 4
4 2 2

1 0 0
5 1 1
2 1 1
0 0 2
```

```
1

1 0 1
```

The function named readSystemState() in your template p2-start.cpp code expects a file of this format, and reads it into a State structure for you. Here the system has 3 resources, and 4 processes. The total systems resource vector **R** comes first, and in this system we have 9 of resource 1 in the system, 3 of R2 and 6 of R3 total.

NOTE: Be careful with indexing here. Remember that C starts indexes for arrays and matrices at 0, while the book indexes process and resource names starting at 1 in the examples. So for example, the book labels the resources R1, R2, R3. When these are read into arrays in the p2-start.cpp example template, they will end up being indexed at locations 0, 1 and 2. So for example, the resource array in the State structure will be initialized to contain

```
state->resource[0] = 9;
state->resource[1] = 3;
state->resource[2] = 4;
```

Likewise, the names of the processes are shifted down by one because of this as well. In the book, it was shown that it was safe for P2 to request 1 of resource R1 and 1 of resource R3 (pg. 273). For our working code, this corresponds to the process indexed as number 1 requesting resource indexed 0 and 2 respectively. So the above example file you were shown was the state-safe.sim file, and it should be determined that this request is safe to grant. Likewise, if you examine the state-unsafe.sim you will see that the system state vectors/matrices are all the same, the only difference is that process 0 is making the request for resource 0 and resource 2. This corresponds to the unsafe request example the book works through when P1 tries to make the same request for R1 and R3.

Running Simulations

The following is a discussion of the expected output of your program. Your program must work from the command line, and expect a single parameter, the name of the state simulation input file, as its input. Your program should display only a single line to standard output as a result of running it, the resulting determination on the given system state and request of SAFE, UNSAFE, SUSPEND, or ERROR. Here is an example of exactly what your program should produce on the command line from running it with each of the 4 example simulation input files:

```
$ p2 state-error.sim
ERROR
$ p2 state-suspend.sim
SUSPEND
$ p2 state-safe.sim
SAFE
$ p2 state-unsafe.sim
UNSAFE
```