Programming Projects

Banker's Algorithm

For this project, you will write a multithreaded program that implements the banker's algorithm discussed in Section 7.5.3. 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 that leaves the system in an unsafe state will be denied. This programming assignment combines three separate topics: (1) multithreading, (2) preventing race conditions, and (3) deadlock avoidance.

The Banker

The banker will consider requests from n customers for m resources types. as outlined in Section 7.5.3. The banker will keep track of the resources using the following data structures:

```
/* these may be any values >= 0 */
#define NUMBER_OF_CUSTOMERS 5
#define NUMBER_OF_RESOURCES 3

/* the available amount of each resource */
int available[NUMBER_OF_RESOURCES];

/*the maximum demand of each customer */
int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

/* the amount currently allocated to each customer */
int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

/* the remaining need of each customer */
int need[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
```

The Customers

Create n customer threads that request and release resources from the bank. The customers will continually loop, requesting and then releasing random numbers of resources. The customers' requests for resources will be bounded by their respective values in the `need` array. The banker will grant a request if it satisfies the safety algorithm outlined in Section 7.5.3.1. If a request does not leave the system in a safe state, the banker will deny it. Function prototypes for requesting and releasing resources are as follows:

```
int request_resources(int customer_num, int request[]);

int release_resources(int customer_num, int release[]);
```

These two functions should return 0 if successful (the request has been granted) and –1 if unsuccessful. Multiple threads (customers) will concurrently access shared data through these two functions. Therefore, access must be controlled through mutex locks to prevent race conditions. Both the Pthreads and Windows APIs provide mutex locks. The use of Pthreads mutex locks is covered in Section 5.9.4; mutex locks for Windows systems are described in the project entitled "Producer–Consumer Problem" at the end of Chapter 5.

Implementation

You should invoke your program by passing the number of resources of each type on the command line. For example, if there were three resource types, with ten instances of the first type, five of the second type, and seven of the third type, you would invoke your program follows:

```
./a.out 10 5 7
```

The `available` array would be initialized to these values. You may initialize the `maximum` array (which holds the maximum demand of each customer) using any method you find convenient.

Bibliographical Notes

Most research involving deadlock was conducted many years ago. [Dijkstra (1965)] was one of the first and most influential contributors in the deadlock area. [Holt (1972)] was the first person to formalize the notion of deadlocks in terms of an allocation-graph model similar to the one presented in this chapter. Starvation was also covered by [Holt (1972)]. [Hyman (1985)] provided the deadlock example from the Kansas legislature. A study of deadlock handling is provided in [Levine (2003)].

The various prevention algorithms were suggested by [Havender (1968)], who devised the resource-ordering scheme for the IBM OS/360 system. The banker's algorithm for avoiding deadlocks was developed for a single resource type by [Dijkstra (1965)] and was extended to multiple resource types by [Habermann (1969)].

The deadlock-detection algorithm for multiple instances of a resource type, which is described in Section 7.6.2, was presented by [Coffman et al. (1971)].

[Bach (1987)] describes how many of the algorithms in the traditional UNIX kernel handle deadlock. Solutions to deadlock problems in networks are discussed in works such as [Culler et al. (1998)] and [Rodeheffer and Schroeder (1991)].

The witness lock-order verifier is presented in [Baldwin (2002)].

Bibliography

[Bach (1987)] M. J. Bach, *The Design of the UNIX Operating System*, Prentice Hall (1987).

[Baldwin (2002)] J. Baldwin, "Locking in the Multithreaded FreeBSD Kernel", *USENIX BSD* (2002).

You need to hand in your source code:

In the source code:

1. You need to implement the Banker's Algorithm.
2. You need to print out the process that you execute processes.

For example : (You can modify this example or design by yourself.)

```
Total system resources are:
A B C D
6 5 7 6
Available system resources are:
A B C D
3 1 1 2
Processes (currently allocated resources):
   A B C D
P1 1 2 2 1
P2 1 0 3 3
P3 1 2 1 0
Processes (maximum resources):
   A B C D
P1 3 3 2 2
P2 1 2 3 4
P3 1 3 5 0
Need = maximum resources - currently allocated
resources
Processes (possibly needed resources):
   A B C D
P1 2 1 0 1
P2 0 2 0 1
P3 0 1 4 0
```

(Reference : https://en.wikipedia.org/wiki/Banker's_algorithm)

3. Also, you need to add comments on your code.

Deadline: 1/9 11:59 p.m.