Banker's Algorithm Program

By Dennis Kovarik

Introduction

This PDF describes the Banker's Algorithm Program, which is a multithreaded program that implements the Banker's Algorithm. This program spawns and runs multiple threads, that will randomly request and release random amounts of resources from a simulated system. By default, this program runs 5 threads with 3 resource types. The amount of each resource type is determined by the command line arguments passed into the program on startup. In addition, each resource is shared among all threads running, so synchronization is controlled using a single mutex lock. With each request by a thread, the program will check if it is safe to fulfill the request by running the banker's algorithm. The program will continue execution until the needs of each thread are fulfilled.

Using the Banker's Algorithm Program

The user can compile/install the program by simply typing 'make' on a Unix command line, which will create an executable named 'bankers'. The executable requires the same number of command line arguments (other than the executable file name) as there are number of resources in the program. 3 arguments), which are all integers. Since the default number of resources is 3, the default number of command line arguments required by the program is 3. An example of executing the default program is as follows:

```
./bankers 1 4 2
```

Each command line argument represents the number of resources of that resource type in order. In the above example, the integer '1' indicates that there is one resource of the first resource type in the program, '4' indicates that there are 4 available resources of the second resource type, and so on.

Upon startup, the program will create the required number of threads/customers, and the maximum demand of each thread/customer will be set to a random integer between 0 and the number of available resources for that resource type. Read and write access to each resource type is controlled using a single mutex lock. Each thread will request and release random amounts of resources until it no longer requires resources, and it no longer has resources allocated to it.

When a request is made, the state of the system will first be printed to the screen. The request will be printed to the screen, along with whether or not the request is granted or not. Afterwards, the stat of the system is printed again.

```
--------------------------------------------------------------------------

         Allocation              Need              Available
         A    B    C         A    B    C         A    B    C
P0       0    0    0         1    1    1         2    3    1
P1       0    0    0         0    2    1
P2       0    0    0         1    2    1
P3       0    0    0         0    0    0
P4       0    0    1         1    0    0

Request P2 <0, 1, 1>
Safe, request granted.

         Allocation              Need              Available
         A    B    C         A    B    C         A    B    C
P0       0    0    0         1    1    1         2    2    0
P1       0    0    0         0    2    1
P2       0    1    1         1    1    0
P3       0    0    0         0    0    0
P4       0    0    1         1    0    0

--------------------------------------------------------------------------
```

Figure 1: request output

When a release is made, the state of the system is printed, then a statement indicating the process making the release along with the resources being released will be printed to the screen. After this the state of the system will be displayed on the screen.

```
--------------------------------------------------------------------------

         Allocation              Need              Available
         A    B    C         A    B    C         A    B    C
P0       1    0    0         0    0    0         0    3    2
P1       0    0    0         0    0    0
P2       0    0    0         0    0    0
P3       0    0    0         0    0    0
P4       1    0    0         0    0    0

Release P4 <1, 0, 0>

         Allocation              Need              Available
         A    B    C         A    B    C         A    B    C
P0       1    0    0         0    0    0         1    3    2
P1       0    0    0         0    0    0
P2       0    0    0         0    0    0
P3       0    0    0         0    0    0
P4       0    0    0         0    0    0

--------------------------------------------------------------------------
```

Figure 2: release output

As the program executes, a series of random requests and releases will be made until the needs of all threads have been fulfilled and until all threads have given up all allocated resources.

Description of Program Files and Functions

The file for this program consists of 4 files, which includes the following: Makefile, bank.h, customers.h, bankers.c.

Makefile

This file is responsible for compiling the program creating the executable file 'bankers'. This file uses the files 'bank.h', 'customers.h', and 'bankers.c' to compile the executable file. Compilation is triggered by typing 'make' on the command line. Removing the executable files and object files from the directory is triggered by typing 'make clean' on the command line.

bank.h

This file defines the number of threads (referred to as customers) and the number of resources in the program. This file also declares the data structures used in the bankers algorithm such as the allocation, available, need, and maximum containers.

customers.h

This is the header file that contains the function prototypes for functions that are responsible for requesting and releasing resources. It also contains the container used to hold the different threads/customers in the program.

bankers.c

This is the source file that implements and tests the bankers algorithm. This file defines and creates the mutex lock used for synchronization. It also contains the main function and all function definitions used in the program. All functions used in the program and function descriptions are given below

- main()
    - This is the main function which serves as the start of the program. First it will check and validate the command line arguments. It will then initialize the data structures used in the banker's algorithm like the available, allocation, and need containers. It will then create and join the customer threads. After all the threads have finished executing, it will destroy the mutex lock
    - Returns
        - 1 if an error has occurred
        - 0 if everything executed correctly.
- create_customers()
    - This function will create and run the required number of threads.
    - Returns
        - Void
- *customer_loop()
    - This is the function run when the threads are created by the 'create_customers' function. This function causes the different threads to randomly request and release

resources until that thread has no more needs in its need array and has no resources allocated to it.
    - Returns
        - Void
- display_usage()
    - This function prints the program usage statement to the screen.
    - Returns
        - Void
- initialize()
    - This function initializes the data structures used in the banker's algorithm. The data structures initialized are the available, allocation, maximum, and need arrays.
    - Returns
        - Void
- print_state()
    - This function prints the state of the system to the screen. It will display the state of the allocation, need, and available containers for each thread to the screen.
    - Returns
        - Void
- print_values
    - This function simply just formats and prints the values held in the container passed into the function to the screen.
    - Returns
        - Void
- release_resources()
    - This function will release the resources held by the thread specified by 'customer_num'. This function will first acquire the mutex lock. Then it will print the state of the system before the resources are released. It will release the resources by modifying the allocation and available containers. It will then print the state of the system and release the mutex lock.
    - Returns
        - 0 when executed correctly.
- request_resources()
    - This function is used by the threads/customers to request the resources held in the 'request[]' container passed into the function. The function will first acquire the mutex lock. It will then print the state of the system before the resources are requested. The function will ensure that the request is less than the threads need. If not, then the function will print that the request is denied and then print the state of the system before return -1. If the request is less than the need, then the function will pretend to have allocated the requested resources and then run the safety check using the Banker's algorithm. If the safety check is not passed, then the function will print that the request has been denied. It will print the state of the system before returning -1. If the safety check is passed, then the resources will be allocated, the function will print to the screen that the request has been granted, then it will print the state of the system before returning 0.

    - Returns
        - -1 if request is unsafe
        - 0 if request is safe.
- safety_test()
    - This function runs the banker's algorithm to test if a request is safe. The function will first initialize the work in finish arrays. Then it will look for a thread/customer that is not finished executing and has need <= work. Once found, it will free up the resources allocated to it and mark the thread as finished. The function continue to repeat this process. At the end, if all threads have finished executing, then the system is in a safe state, and the function will return 0. If all threads have not finished executing in the end, then the system is not in a safe state and the function will return -1.
    - Returns
        - -1 if system in safe state
        - 0 if system is in safe state
- validate_command_args
    - This function will check and validate the command line arguments passed into the function. It will first check for the current number of command line arguments. Then it will check that all command line arguments are integers.
    - Returns
        - -1 if command line arguments are not valid.
        - 0 if command line arguments are valid.

Testing

Testing the program was completed by manual tests. The banker's algorithm was checked by the following test:

./bankers 11 5 9

```
     Allocation      Need        Available
     A  B  C         A  B  C     A  B  C
P0   0  1  0         7  4  3     3  3  2
P1   3  0  2         1  2  2
P2   3  0  2         6  0  0
P3   2  1  1         0  1  1
P4   0  0  2         4  3  1


Request P1 <1, 0, 2>
Safe, request granted.
```

	Allocation			**Need**			**Available**		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	2	3	0
P1	4	0	4	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			