Program Write-up

Jennifer Kulich

April 9 2020

1 Overview

This program was to implement Bankers algorithm that we went over in class. Please take a look at the document blow (esepcially the run section) before you run/grade the program.

2 How To Run

You may type in:

```
make clean; make all
```

and then

```
./bankers num num num
```

where the 'num's are what you would like in the available vector.

PLEASE NOTE: The way this program works, it will run forever. The program shouldn't have to run more than 5 seconds for you to see the algorithm working.

3 Limitations

The way this program works, it's not a very realistic simulation of what would happen. The way I wrote the program was that it each customer will be able to make a request once, and once they have all been able to request, they will all have a chance to release resources. Ideally, the program would request and release in any order so the could be requests and release intermingled.

4 How the Program Works

The program starts off in Customer.c . it will first make sure that everything you put into the command line is correct. It will then fill in the available vec-

tor with what was passed in through the command line. Next, the max and allocation vectors will be initialized- max with a random number and allocation with all zeros. Now that the allocation and max vectors have been initialized, the need vector will be initialized by subtracting the allocation from the max at every location. (God, the amount of for loops I do in this program...). Then, because nothing has been done yet, the information is printed out so we know how things start (that is, all of the customers allocations, needs, and what's available). Threads are then created, and then that's where things really start.

The program gets into the customer function with a certain customer number. We save the customer number to a variable inside the function because of the mutex lock (did I learn my lesson from the first program, yup). We then enter a while loop that will keep things running. In that while loop, we do our requesting and releasing. As I said above, I should probably be more like the real world and make there be requests and releases mixed in together, but that really didn't cross my mind until after I got everything working because I was just trying to get the program working. A mutex lock was put around the request loop and then the release loop separately. In the request loop, if the need for the customer is not zero, we can assign the request for that one a random number that's less than the need. If the need was zero, the request is set to zero because it can't be bigger than the need. The request that is being made is then output to let the user know, and then the request function is called. I will talk about that more below. After all the requests are either accepted or denied, the mutex is unlocked, and the program can move on to allow each customer to release resources. A lock is put around the for loop again. In that for loop, if the allocation number is not zero, we can set the release number to a random number less than the allocation number. If the allocation number is zero, the release must be zero because it can't be bigger than the allocation number. The release function is called which will be discussed below. After every customer have had the chance to release their resources, they will continue.

The request function will do a couple of things to make sure the request can be made. The first thing it does is check if the request is greater than the need. This is done by going through the needs of the customer and making sure the corresponding request is smaller. If they are not smaller, the request function will tell the user what is wrong and will not continue. The request function will then check if there is enough to allocate. To do this, it will make sure that the corresponding available is greater than the need. If one is not, the program will tell the user what is wrong and exit the request function. Then we continue to follow Banker's algorithm and pretend to allocate things. To do this, we subtract the request from the need, add the request to the allocation and subtract the request from the available. Once that has been done, the program needs to make sure that having done this will keep the program in a safe state. It will call a long, complicated function that tests if the program is still safe. This function will set the work vector to the current available vector and everything in a finish vector to false. The function will then go through to finish an index

where the finish is false and the need is less than or equal the work. If it exits, we set the finish to true, and we know the program is in a safe state with the request. If there was not an index found, we will set the work to the work plus the allocation, set the finish to true and return to check for another index where finish is false and the need is less than or equal to the work. Then, if all of the finish vector is true, we know the program is in a safe state. If the isSafe function returns that state is safe, the program will print out that the request was safe, and the new information will be printed and return 0. If the isSafe function returns that the state is unsafe, the program will print out that the state is unsafe and the request will not be granted. it then goes through, resets that need, allocation, and available and return with a -1.

The release function is more simple. The first thing it does is is make sure there are enough resources to release. It does this by making sure that the release vector is less than the allocation vector (which it always should be, but you know how things go). If it is not ok, the program will say that it cannot release and return -1. If the program can release the resources, it will subtract from the allocation, add to the need, and add to the available. The program will then print who release, what they released, and the state of the program and then return 0.

5 So Why is the Program Infinite?????

The program is infinite because of how the requesting and releasing works. When a customer requests, they subtract whatever they requested from the need, but because the request can never be greater than the need, the need will never get below zero. The customer can only request once, so after it requests, it will automatically release something which will just add back to the need. This guarantees that the program will never finish. It could technically get all of the needs to zero, but could you image how nearly impossible that would be? I would be amazed if that happened. However, the algorithm still works which is just what needed to be shown.