CSC458. Program 2. Banker's Algorithm

Nathaniel Fagrey

April 9, 2020

Introduction

For this program, we will be demonstrating the banker's safety algorithm for processes in a computer requesting and releasing data. The main idea for this program is that we want to prevent deadlock between processes when they request data. Deadlock occurs when two or more processes need the data that another one has and both are unwilling to give up their data. The Bank's Algorithm allows us to test to see if a request by a process will cause the system to come into a deadlock (unsafe) state.

Banker's Algorithm

In order to run the algorithm we need several data structures to hold all our data.

- available: a list that keeps track of the resources we currently have available.
- max: a list that contains the max amount of resources for each process.
- allocation: a list that contains all the resources currently held by the processes.
- need: a list that contains the resources each process still needs.

The Banker's Algorithm is as follows [1]:

- 1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available and Finish[i] = false for i = 0, 1, ..., n - 1.
- 2. Find an index i such that both

  a. Finish[i] == false

[1] From the book Operation System Concepts by Abraham Silberschatz, Peter Galvin, Greg Gagne

b. Need i $\leq$ Work If no such i exists, go to step 4.

- 3. Work = Work + Allocation[i] Finish[i] = true Go to step 2.
- 4. If Finish[i] == true for all i, then the system is in a safe state.

If this algorithm is able to complete in step 4 then our system will not be in deadlock.

Resource Precaution

However, before when even run the Banker's Algorithm we need to make sure that the process has a valid process request. When the process requests a resource we first have to make sure it is not greater than the processes current need. If if is, the process has lied to us and we kill it. We then need to make sure that the process does not ask for more than what is currently available.

Assumptions

Because there are multiple ways to write this program, I wanted to list some of my assumptions.

- The Max array is always bounded by 10.
- The program is meant to demonstrate the Banker's Algorithm, therefore it probably won't terminate. That will be up to the user.
- The same process randomly request resources, and right away, randomly releases some other random resources.
- There is a condition to terminate a thread if its need = 0, but the chances of doing this with how the resources are released is low.

The Program

The program we wrote combines the Banker's Algorithm and the Resource Precaution so simulate the giving and taking of resources.

We first simulate multiple processes by having multiple threads request and release resources. Because we have multiple threads, we need to incorporate mutex locks to prevent race conditions on shared data.

- Our max list is created by filling an array with random values
- The allocation list is filled with 0's since nothing has been given to the processes yet.
- The need array is random values bounded by the max array.

- The available array is filled with the numbers given on the command line.

The threads randomly create requests that our bounded by the need array (since we are created requests in this way, we do not bother checking them in the program since request $\leq$ need).

After the request has been made, we pretend to add it and subtract it from the available, need, and allocation arrays. Then we run the Banker's Algorithm on it. If the algorithm reaches a safe state we print that it did on the screen and leave the affect resources as they are. If the Banker's fails, we revert the resources back to their previous state and exit the program.

We also randomly release resources for each process and sleep after those resources have been released. This is mainly to show more threads at once running; as well as, making the program more realistic.

We end the program when the need for each process is equal to 0. This means that all processes got everything they needed. We kill the process and release back all of its resources. However, because we continually releasing resources back to need, the chances that all the threads will terminate is extremely unlikely. It is up to the user to terminate.