Programming Assignment 2 – Banker's Algorithm

Minh Nhat Nguyen

CS 33211: OPERATING SYSTEMS

Dr. Qiang Guan

Kent State University

April 26, 2024

Introduction

This is the documentation for the second programming assignment, which aims to implement the Banker's Algorithm to avoid deadlock in a system. There are a set number of processes and types of resources in a system, and the algorithm's job is to determine whether deadlock occurs within the system. We will discuss in depth about deadlock, what the Banker's Algorithm is, and how it is implemented.

1. Deadlock.

According to the book Operating System Concepts, a set of processes is in a deadlocked state when: "every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release". "In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting".

To prevent deadlocks from arising within systems, various algorithms have been implemented throughout history. In this assignment, we will be using what is known as Banker's Algorithm.

2. Banker's Algorithm

Banker's Algorithm essentially uses the same principles that banks apply when they loan out money, where they would never allocate their money in a way that they can no longer satisfy all their customers.

The simplest way to implement this is to declare the maximum number of resources of each type that each individual process might need. Then, an algorithm will be run to examine the resource allocation of each process and ensure that no circular wait occurs. Afterwards, the algorithm will declare if the system is in a "safe state" by finding a sequence of all processes such that "for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$" (Taken from the chapter 7 slides).

In the following section, we will go into how this algorithm is implemented and how it determines safe sequences in C++.

3. Implementation

Firstly, we will need to implement reading all the necessary data from files for the algorithm. This data will be stored in variables and vectors. For this implementation, these variables look like this.

```
// input for the algorithm
int n;
int m;

vector<int> available;
vector<vector<int>> max;
vector<vector<int>> allocation;
vector<vector<int>> need;
```

The specific use for all of these variables and vectors are:

- `int n:` Store the number of processes in the system.
- `int m:` Store the number of types of resources.

- `vector<vector<int>> allocation:` Vector to store information on how many instances of what resource is allocated to what process.
- `vector<vector<int>> max:` Vector to store the maximum number of instances that each type of resource can allocate to the processes.
- `vector<int> available:` Vector to store the available number of resources of each type.
- `vector<vector<int>> need:` Vector that store the remaining resource that each process needs.

Next, we read the data from text files, starting with the allocation vector. The code for that looks like this:

```
// Read allocation information from file
void readAllocation() {
   ifstream allocationFile("allocation.txt");
   int t;
   vector<int> temp;
   allocationFile >> t;
   n = t;
   allocationFile >> t;
   m = t;

   for (int i = 0; i < n; ++i) {
      temp.clear();
      for (int j = 0; j < m; ++j) {
         allocationFile >> t;
         temp.push_back(t);
      }
      allocation.push_back(temp);
   }
}
```

The code works as follows:

- First, the program signifies the name of the input file for the allocation vector.
- Then, it reads the number of processes and number of types of resources.
- Finally, it reads all the data in the file that are separated with spaces. Each row of data has **`m`** data items and there are **`n`** rows.

An example for the input file looks like this:

```
5 3
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
```

Next, we read in data to fill the available vector. The code looks like this:

```
// Read available information from file
void readAvailable() {
   ifstream availableFile("input/available.txt");
   int t;
   while (availableFile >> t) {
      available.push_back(t);
   }
}
```

The code is fairly simple in that it only looks for the file name, reads the data (separated by spaces) and store that in the `available` vector.

An example input file looks like this:

```
3 3 2
```

Next, we read the data for the `max` vector. The code looks like this:

```
// Read max information from file
void readMax() {
   ifstream maxFile("input/max.txt");
   int t;
   vector<int> temp;
   for (int i = 0; i < n; ++i) {
      temp.clear();
      for (int j = 0; j < m; ++j) {
         maxFile >> t;
         temp.push_back(t);
      }
      max.push_back(temp);
   }
}
```

This code works as follows:

- First, the program specifies the name of the input file and its location.
- Then, it reads all the data that are separated by spaces. Each row of data has **`m`** items and there are **`n`** rows. `m` and `n` were read from the allocation input procedure.

Finally, we generate the `need` vector. The code for that looks like this:

```
// Use the max and allocation vectors to make the need
vector
```

```
void makeNeed() {
   vector<int> temp;
   for (int i = 0; i < n; ++i) {
      temp.clear();
      for (int j = 0; j < m; ++j) {
         temp.push_back(max[i][j] - allocation[i][j]);
      }
      need.push_back(temp);
   }
}
```

This code works by using the `max` and `allocation` vectors and subtracting each entry to generate the `need` vector.

Finally, we put all of these together in the main file, and then we run the algorithm to determine whether a safe sequence exists. The main code looks like this:

```
int main() {
   // Read and prepare all inputs from files
   readAllocation();
   readAvailable();
   readMax();
   makeNeed();

   // vectors and variables for the algorithm
   vector<int> finish(n, 0);
   vector<int> result(n);
   int index = 0;

   // The Banker's Algorithm
   for (int k = 0; k < 5; k++) {
      for (int i = 0; i < n; i++) {
         if (finish[i] == 0) {

            int flag = 0;
            for (int j = 0; j < m; j++) {
               if (need[i][j] > available[j]) {
                  flag = 1;
                  break;
               }
            }

            if (flag == 0) {
               result[index++] = i;
               for (int y = 0; y < m; y++)
                  available[y] += allocation[i][y];
               finish[i] = 1;
            }
         }
      }
   }

   // Check whether or not the sequence was safe
   int check = 1;

   for (int i = 0; i < n; i++) {
      if (finish[i] == 0) { // Sequence was not safe
         check = 0;
         cout << "The given sequence is not safe";
         return 0;
      }
   }

   // Prints out the safe sequence
   cout << "The safe sequence is:" << endl;
   for (int i = 0; i < n - 1; i++) {
      cout << "P" << result[i] << " -> ";
   }
   cout << "P" << result[n - 1] << endl;

   return 0;
}
```

In the main function, we first call all the previously mentioned procedures. We then declare two vectors and a variable for the algorithm. Specifically:

- `vector<int> finish:` Vector to store whether a specific process has finished being checked for eligibility in the safe sequence. All entries are initialized to 0 (false), they are changed to 1 (true) once the algorithm confirms that they are safe.
- `vector<int> result:` Vector to store the order by which the safe sequence is organized, given that one exists in the first place.
- `int index:` index of the process that the algorithm is currently working with.

Afterwards, the program runs the Banker's Algorithm to determine whether a safe sequence exists. This algorithm works as follows:

- For each process in the system, we run a loop that iterates `m` number of times (`m` being the number of types of resources). For each iteration, we check the entry of the `need` vector which holds that type of resource, and if it is bigger than the entry of that type of resource in the `available` vector, we mark it as unsafe, and break the entire loop.
- However, if after the entire loop, no entry was unsafe, then we add that process to the `result` vector, and we add the entire row of the `allocation` vector that corresponds to the process we just added. Finally, we mark that process as finished by making its entry in the `finish` vector 1 (true)

We do that for the entire set of processes 5 times. After that, we print out the safe sequence if it exists, or signals that it does not. It checks for the existence of a safe sequence by checking the `finish` vector. If any entry holds a 0 (false), that means that the system is unsafe. Otherwise, we print the safe sequence using the `result` vector.

Now that we know what the entire program does, let us go on to how to run it properly.

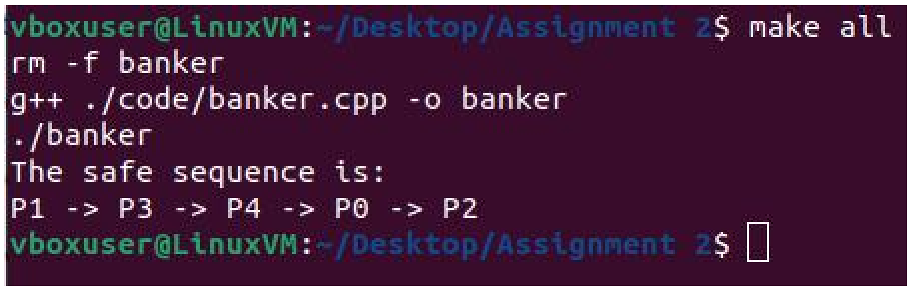
Compilation.

Once you have all the code, you can simply run the `Makefile` to compile the processes. You can do this by entering:

```
make all
```

into the Linux or Unix terminal. Make sure to navigate to the path where the `Makefile` is. It will clean up the previous object file, compile the necessary program, and execute.

Here is an example:



```
vboxuser@LinuxVM:~/Desktop/Assignment 2$ make all
rm -f banker
g++ ./code/banker.cpp -o banker
./banker
The safe sequence is:
P1 -> P3 -> P4 -> P0 -> P2
vboxuser@LinuxVM:~/Desktop/Assignment 2$ 
```

Conclusion

Deadlocks in systems are a notorious problem for developers. There are many ways to deal with it, from prevention, avoidance, recovery or just ignoring it entirely. The Banker’s Algorithm is one of many ways that fixes the issue. It is one of the simplest and most effective ways to break one of the deadlock conditions, being circular wait. Despite that, an algorithm, however simple, still would take time because it needs to be run periodically to make sure that no deadlock has happened beforehand. All in all, this assignment has helped me understand more in depth about the relationship between processes and resources, and how deadlock functions.

References

SILBERSCHATZ, A. (2021). *Operating system concepts*. JOHN WILEY.

Slides from the Operating System course.