Project 3 Report

Litao Zhou, 518030910407

# Project Description

In this project, we implement the banker algorithm for deadlock-safety resource allocation.

The source code of this project can be found [here](https://github.com/ltzone/2020Fall/tree/master/EI338/Project/deadlock).

Contents

[Project Description 1](#_Toc59109164)

[Algorithm Implementation 2](#_Toc59109165)

[Request Resources 2](#_Toc59109166)

[Safety Check 3](#_Toc59109167)

[Release Request 4](#_Toc59109168)

[Command Line Interface 5](#_Toc59109169)

[Experiment Results 7](#_Toc59109170)

[Case 1: Successful Allocation 7](#_Toc59109171)

[Case 2: Successful Allocation 8](#_Toc59109172)

[Case 3: Ungranted Allocation 9](#_Toc59109173)

[Case 4: Release Request 10](#_Toc59109174)

# Algorithm Implementation

The C code of the signatures and data structures of the algorithm has been given in the textbook. We will keep them in the project. The header of the codes are listed as follows.

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define NUMBER\_OF\_CUSTOMERS 5

#define NUMBER\_OF\_RESOURCES 4

char MAX\_REQUEST\_DIR[] = "./max.in";

/\* 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];

int request\_resources(int customer\_num, int request[]);

void release\_resources(int customer\_num, int release[]);

int safety\_check();

void print\_status();

Note that in addition to the request/release functions, we extract the safety check function out, and also implement a print\_status function which can print out the current available/maximum/allocation/need tables.

The implementation of the print\_status function is trivial, so we will focus on the remaining three functions and the command line interface implementation.

## Request Resources

When dealing with a request, first check the available resources, and then we make virtual allocation to check the safety condition. If the safety condition is satisfied, the allocation is done, otherwise the allocation should be restored.

int request\_resources(int customer\_num, int request[]){

/\*\* return 0 if successful and 1 if unsuccessful \*/

/\* check request <= available, request <= max \*/

for (int i=0;i<NUMBER\_OF\_RESOURCES;++i){

if (allocation[customer\_num][i] + request[i] > maximum[customer\_num][i]){

return 1;

}

if (request[i] > available[i]){

return 1;

}

}

/\* virtual allocation \*/

for (int i=0;i<NUMBER\_OF\_RESOURCES;++i){

available[i] -= request[i];

allocation[customer\_num][i] += request[i];

need[customer\_num][i] -= request[i];

}

if (safety\_check() == 1){

return 0;

} else {

/\* if fails, restore the allocation \*/

for (int i=0;i<NUMBER\_OF\_RESOURCES;++i){

available[i] += request[i];

allocation[customer\_num][i] -= request[i];

need[customer\_num][i] += request[i];

}

return 1;

}

}

The function will return 0 and make modifications to the global tables if the request is granted, otherwise it will return 1 and no modifications will be made to the global information.

## Safety Check

The safety check is based on the global tables, so no function arguments are needed. It will use a work and finish array to check whether there exists an order in which the resources can be released in sequential order, causing no deadlocks. Here we use a finish\_cnt to count the number of finished customers, so the safety check can end successfully when it reaches the total number of customers.

int safety\_check(){

/\*\* return 1 if safe and 0 if unsafe \*/

/\* initialize work and finish array \*/

int work[NUMBER\_OF\_RESOURCES];

int finish[NUMBER\_OF\_CUSTOMERS];

for (int i=0;i<NUMBER\_OF\_RESOURCES;++i){

work[i] = available[i];

}

for (int i=0;i<NUMBER\_OF\_CUSTOMERS;++i){

finish[i] = 0;

}

/\* use finish\_cnt to count the number of customers with finish[i] = 1 \*/

int finish\_cnt = 0;

while (finish\_cnt < NUMBER\_OF\_CUSTOMERS){

int chosen\_proc = -1;

for (int i=0;i<NUMBER\_OF\_CUSTOMERS;++i){

if (finish[i] == 0){

int valid\_flag = 1;

for (int j=0;j<NUMBER\_OF\_RESOURCES;++j){

if (need[i][j] > work[j]){

valid\_flag = 0;

break;

}

}

if (valid\_flag == 1){

chosen\_proc = i;

break;

}

}

}

if (chosen\_proc == -1){

/\* fail to find a customer to release its resources \*/

return 0;

}

/\* successfully find a customer, and release its resources \*/

finish\_cnt += 1;

finish[chosen\_proc] = 1;

for (int i=0;i<NUMBER\_OF\_RESOURCES;++i){

work[i] += allocation[chosen\_proc][i];

}

}

return 1;

}

The function will return 1 if the safety check is passed, otherwise it will return 0.

## Release Request

The release request is also trivial to implement. We can simply release the allocated resources, and add them back to the available array. Note that in case the request releases more resources than the customer actually holds, we specifies that in this case, the allocated resources will be reduced to zero.

void release\_resources(int customer\_num, int release[]){

for (int i=0;i<NUMBER\_OF\_RESOURCES;++i){

if (release[i] > allocation[customer\_num][i]){

release[i] = allocation[customer\_num][i];

}

available[i] += release[i];

allocation[customer\_num][i] -= release[i];

need[customer\_num][i] += release[i];

}

}

## Command Line Interface

The command line interface is also simple to implement. We first ask for arguments to fill the available array. Then we load from an external file to fill the max table. The remaining part of the program will be included in a loop to support interactive mode. We simply parse the input from the command line and call the corresponding function. Note that we also add some extra codes to deal with invalid inputs and exception cases.

int main(int argc, char \*\*argv){

/\* initialize available and max \*/

if (argc != NUMBER\_OF\_RESOURCES+1){

printf("Wrong number of resources, input %d, but %d expected\n",argc-1 ,NUMBER\_OF\_RESOURCES);

return -1;

}

for (int i=0;i<NUMBER\_OF\_RESOURCES;++i){

available[i] = atoi(argv[i+1]);

}

/\* load max table from MAX\_REQUEST\_DIR \*/

FILE \*fp = fopen(MAX\_REQUEST\_DIR, "r");

if (!fp){

printf("Fail to open file '%s'\n",MAX\_REQUEST\_DIR);

}

for (int i=0;i<NUMBER\_OF\_CUSTOMERS;++i){

for (int j=0;j<NUMBER\_OF\_RESOURCES;++j){

if (fscanf(fp, "%d,", &maximum[i][j]) != 1){

printf("The file is not formated correctly.\n");

fclose(fp);

return -1;

}

}

fscanf(fp, "\n");

}

fclose(fp);

for (int i=0;i<NUMBER\_OF\_CUSTOMERS;++i){

for (int j=0;j<NUMBER\_OF\_RESOURCES;++j){

allocation[i][j] = 0;

need[i][j] = maximum[i][j];

}

}

int should\_run = 1;

while (should\_run){

char instr[10];

printf("banker> ");

fflush(stdin);

if (scanf("%s", instr) != 1){

printf("Empty input.\n");

continue;

}

if (strcmp(instr, "\*") == 0){

print\_status();

continue;

}

int customer\_num;

int request[NUMBER\_OF\_RESOURCES];

if (scanf("%d", &customer\_num) != 1 || customer\_num >= NUMBER\_OF\_CUSTOMERS){

printf("Invalid Arguments, Customer number expected.\n");

continue;

}

for (int i=0;i<NUMBER\_OF\_RESOURCES;++i){

if (scanf("%d", &request[i]) != 1){

printf("Invalid Arguments, Resource number of %d expected.\n", i);

continue;

}

}

if (strcmp(instr,"RQ") == 0){

if (request\_resources(customer\_num,request) != 0){

printf("Request not granted\n");

} else {

printf("Request granted\n");

}

} else if (strcmp(instr,"RL") == 0){

release\_resources(customer\_num,request);

printf("Released granted\n");

} else {

printf("Invalid Instruction, aborted\n");

}

}

return 0;

}

# Experiment Results

We use the cases in Assignment 11 to test our implementation. In the following experiments, we assume that there are 4 types of resources and 5 customers in our system.

## Case 1: Successful Allocation

The case is from Assignment 11, exercise 1.

Available Resources: 3 14 12 12

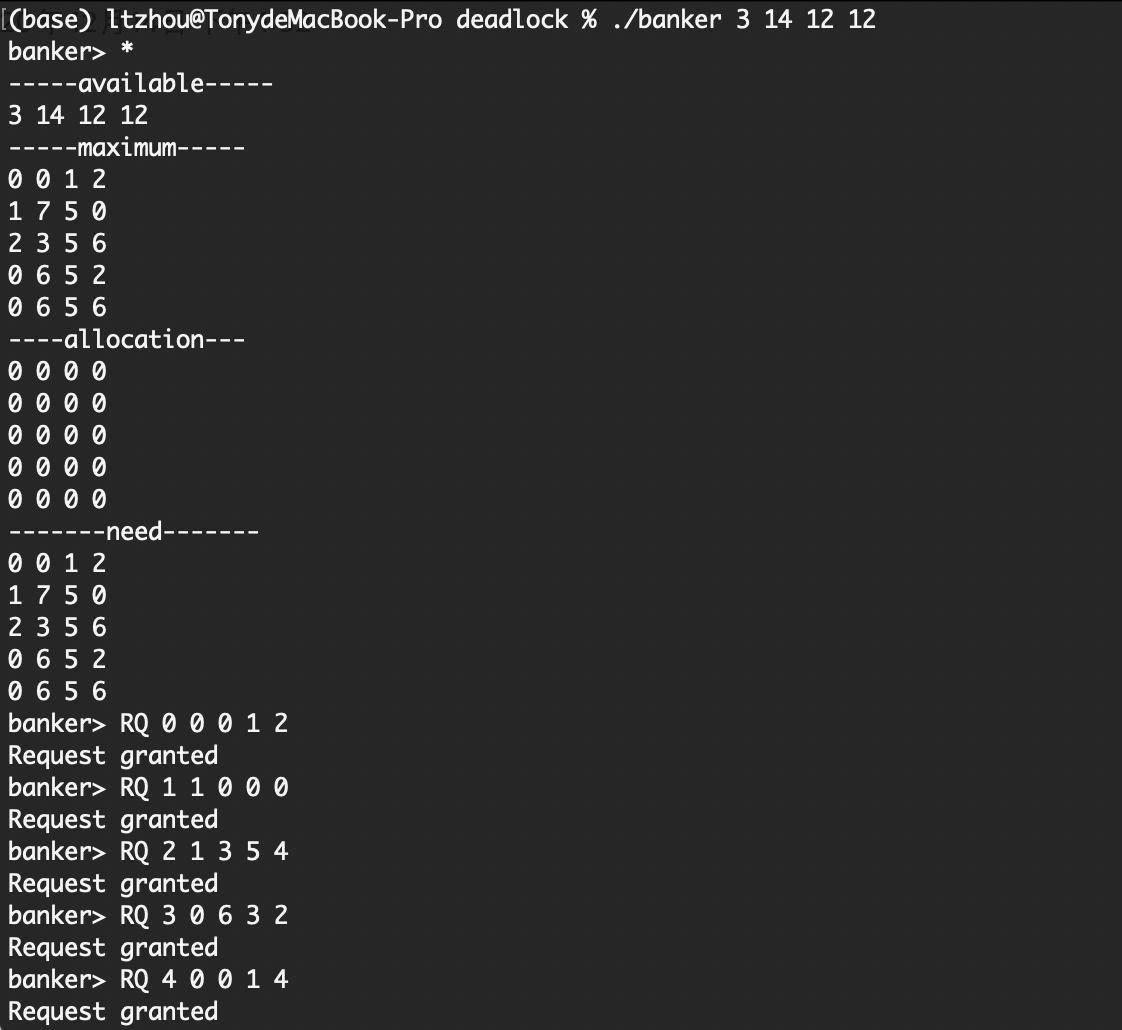
Max Table:

* Customer 0: 0,0,1,2
* Customer 1: 1,7,5,0
* Customer 2: 2,3,5,6
* Customer 3: 0,6,5,2
* Customer 4: 0,6,5,6

Request Order:

* RQ 0 0 0 1 2
* RQ 1 1 0 0 0
* RQ 2 1 3 5 4
* RQ 3 0 6 3 2
* RQ 4 0 0 1 4
* RQ 1 0 4 2 0

The test result is shown below, all the requests are granted as expected.



We can also use “\*”to check the tables, it is shown below that the table is correctly modified after a granted request.

文本

描述已自动生成

## Case 2: Successful Allocation

The case is from Assignment 11, exercise 2(1).

Available Resources: 13 10 6 9

Max Table:

* Customer 0: 5,1,1,7
* Customer 1: 3,2,1,1
* Customer 2: 3,3,2,1
* Customer 3: 4,6,1,2
* Customer 4: 6,3,2,5

Request Order:

* RQ 0 3 0 1 4
* RQ 1 2 2 1 0
* RQ 2 3 1 2 1
* RQ 3 0 5 1 0
* RQ 4 4 2 1 2

The test result is shown below, all the requests are granted as expected.

图形用户界面, 文本

描述已自动生成

## Case 3: Ungranted Allocation

The case is from Assignment 11, exercise 2(2).

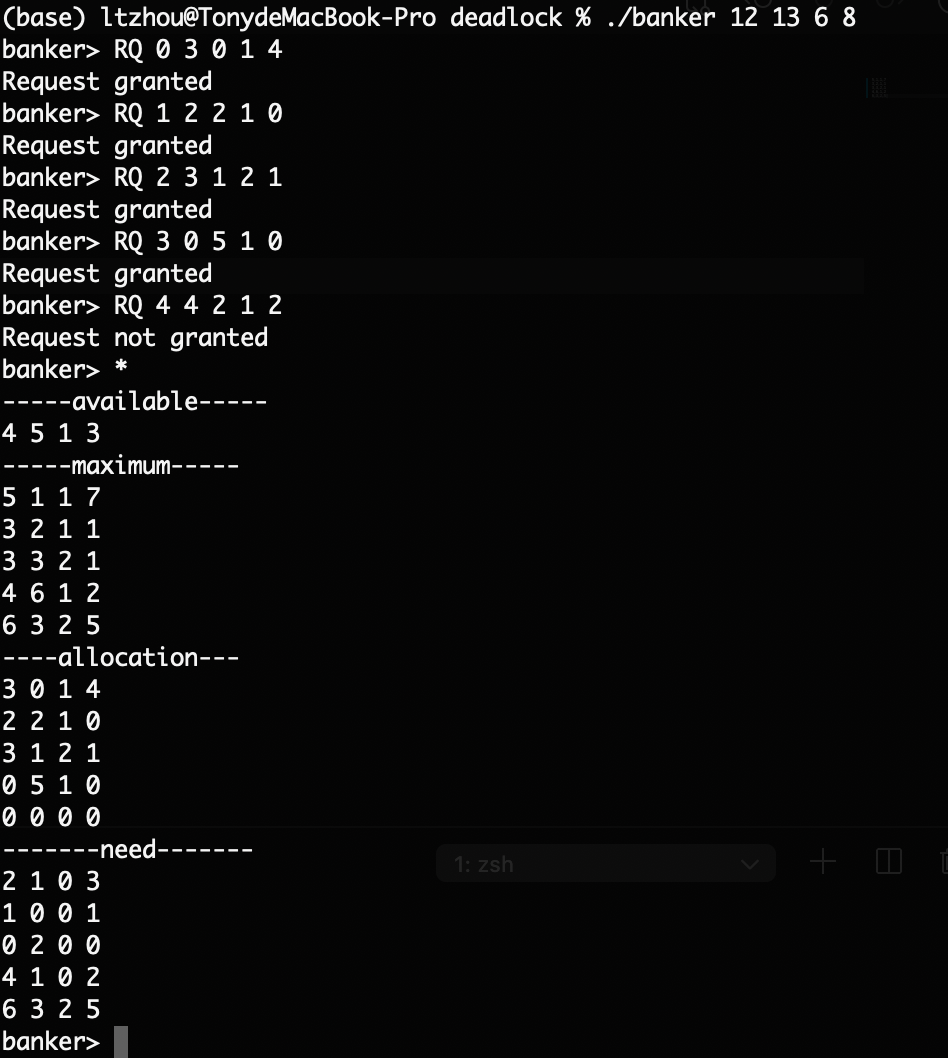
Available Resources: 12 13 6 8

Max Table: Same as Case 2

Request Order:

* RQ 0 3 0 1 4
* RQ 1 2 2 1 0
* RQ 2 3 1 2 1
* RQ 3 0 5 1 0
* RQ 4 4 2 1 2

The test result is shown below, the last request is not granted for failure in safety check.



## Case 4: Release Request

The test case shows that after the release request, correct modifications have been made to the global tables.

