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.

Contents

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

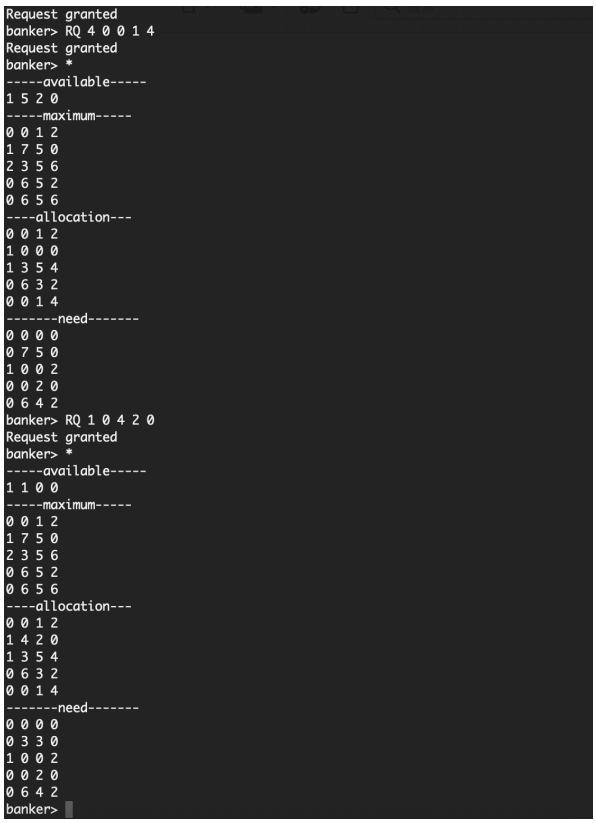
Request Order:

- 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.

```
(base) ltzhou@TonydeMacBook-Pro deadlock % ./banker 3 14 12 12
banker> *
-----available-----
3 14 12 12
-----maximum-----
0 0 1 2
1 7 5 0
2 3 5 6
0 6 5 2
0 6 5 6
----allocation---
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
-------need-------
0 0 1 2
1 7 5 0
2 3 5 6
0 6 5 2
0 6 5 6
banker> RQ 0 0 0 1 2
Request granted
banker> RQ 1 1 0 0 0
Request granted
banker> RQ 2 1 3 5 4
Request granted
banker> RQ 3 0 6 3 2
Request granted
banker> RQ 4 0 0 1 4
Request granted
```

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



Case 2: Successful Allocation

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

Available Resources: 13 10 6 9

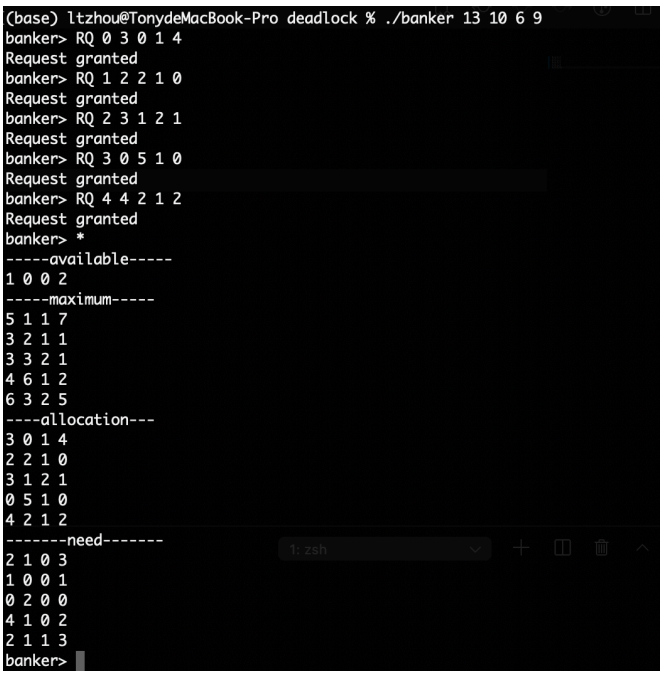
Max Table:

- 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.



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.

```
(base) ltzhou@TonydeMacBook-Pro deadlock % ./banker 12 13 6 8
banker> RQ 0 3 0 1 4
Request granted
banker> RQ 1 2 2 1 0
Request granted
banker> RQ 2 3 1 2 1
Request granted
banker> RQ 3 0 5 1 0
Request granted
banker> RQ 4 4 2 1 2
Request not granted
banker> *
-----available-----
4 5 1 3
-----maximum-----
5 1 1 7
3 2 1 1
3 3 2 1
4 6 1 2
6 3 2 5
----allocation---
3 0 1 4
2 2 1 0
3 1 2 1
0 5 1 0
0 0 0 0
-------need-------
2 1 0 3
1 0 0 1
0 2 0 0
4 1 0 2
6 3 2 5
banker>
```

Case 4: Release Request

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

```
banker> *
-----available-----
4 5 1 3
-----maximum-----
5 1 1 7
3 2 1 1
3 3 2 1
4 6 1 2
6 3 2 5
----allocation---
3 0 1 4
2 2 1 0
3 1 2 1
0 5 1 0
0 0 0 0
-------need-------
2 1 0 3
1 0 0 1
0 2 0 0
4 1 0 2
6 3 2 5
banker> RL 3 0 4 1 0
Released granted
banker> *
-----available-----
4 9 2 3
-----maximum-----
5 1 1 7
3 2 1 1
3 3 2 1
4 6 1 2
6 3 2 5
----allocation---
3 0 1 4
2 2 1 0
3 1 2 1
0 1 0 0
0 0 0 0
-------need-------
2 1 0 3
1 0 0 1
0 2 0 0
4 5 1 2
6 3 2 5
banker>
```