

دانشگاه صنعتي امیرکبیر  
(پلی تکنیک تهران)

دانشكده مهندسی برق

آزمایشگاه سیستم عامل

آزمایش ۷

بن­بست و الگوریتم بانکدار

نگارش

علی بابالو

پویا شریفی

استاد راهنما

مهندس کیخا

خرداد ماه 1402

بخش اول:

در این آزمایش برای جلوگیری از dead lock از روش اجتناب یا avoidance استفاده می کنیم.

یکی از روش های اجتناب، استفاده از الگوریتم بانکداران می باشد(Banker’s Algorithm).

در این روش هر درخواستی که از طرف مشتری می آید، بررسی می شود که آیا با پاسخ دادن به آن همچنان سیستم safe می ماند یا خیر. در صورت ایمن ماندن باید منبع را تخصیص دهیم و پس از اتمام کار آن باید منبع را دوباره به منابع موجود اضافه کنیم.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <stdbool.h>

#include<time.h>

#define NUMBER\_OF\_CUSTOMERS 5

#define NUMBER\_OF\_RESOURCES 3

/\* 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] = {

                                                        { 9, 10, 4 },

                                                        { 2, 4, 6 },

                                                        { 3, 8, 4 },

                                                        { 6, 10, 2 },

                                                        { 5, 5, 5 }

                                                        };

/\* the amount currently allocated to each customer \*/

int allocation[NUMBER\_OF\_CUSTOMERS][NUMBER\_OF\_RESOURCES] ={

                                                            { 0, 1, 0 },

                                                            { 2, 0, 0 },

                                                            { 3, 0, 2 },

                                                            { 2, 1, 1 },

                                                            { 2, 3, 2 }

                                                        };

/\* the remaining need of each customer \*/

int need[NUMBER\_OF\_CUSTOMERS][NUMBER\_OF\_RESOURCES];

//threads:

pthread\_t tid[NUMBER\_OF\_CUSTOMERS];

pthread\_mutex\_t lock1;

pthread\_mutex\_t lock2;

bool isSafe(){

    int work[NUMBER\_OF\_RESOURCES];

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

        work[i] = available[i];

    }

    bool finish[NUMBER\_OF\_CUSTOMERS];

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

        finish[i] = false;

    }

    bool flag\_can;

    int i,cnt = 0;

    repeat:

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

            flag\_can = true;

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

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

                    flag\_can = false;

                    break;

                }

            }

            if(!finish[i] && flag\_can)

                break;

        }

        if(!finish[i] && flag\_can && cnt < NUMBER\_OF\_CUSTOMERS){

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

                work[k] += allocation[i][k];

            finish[i] = true;

            cnt ++;

            goto repeat;

        }else{

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

            if(finish[k] == false){

                return false;

            }

            return true;

        }

}

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

    //give back resources:

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

        available[i] += release[i];

    }

    return 0;

}

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

    pthread\_mutex\_lock(&lock2);

    release\_resources(release,customer\_num);

    pthread\_mutex\_unlock(&lock2);

    printf("Thread %d finished execution \n",customer\_num);

}

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

    printf("Customer %d is Requesting Resources:\n", customer\_num);

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

        printf("%d  ",request[i]);

    }

    printf("\nAvailable = ");

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

        printf("%d  ", available[i]);

    }

    printf("\nNeed = ");

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

        printf("%d  ", need[customer\_num][i]);

    }

    printf("\n");

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

        if (request[i] > need[customer\_num][i]) {

            printf("Request is more than need! ABORT!\n");

            return -1;

           }

    }

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

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

            printf("Request is more than available! ABORT!\n");

            return -1;

           }

    }

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

        available[j] -= request[j];

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

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

    }

    if(isSafe()){

        printf("Safe! Request is granted!\n");

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

            bool is\_empty = true;

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

                if(need[j][k] > 0)

                    is\_empty = false;

            if(is\_empty){

                printf("%d got all it needed!\n", j);

                int max[NUMBER\_OF\_RESOURCES];

                release\_resources\_control(maximum[j], j);

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

                    allocation[j][k] = 0;

            }

        }

        return 0;

    }else{

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

            available[j] += request[j];

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

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

        }

        printf("Not safe! Can't grant request!\n");

        return -1;

    }

}

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

    //CRITICAL SECTION //

    bool released = false;

    pthread\_mutex\_lock(&lock1);

    printf("------------------\n");

    released=request\_resources(request, customer\_num);

    pthread\_mutex\_unlock(&lock1);

    return released;

}

void\* getResources(void \*arg){

    int customerNum = \*(int \*)arg;

    for(int i=0; i<1; i++){

        srand(time(NULL));

        //a random request

        int need\_1 =  need[customerNum][0] == 0 ? 0 : rand() % need[customerNum][0];

        int need\_2 =  need[customerNum][1] == 0 ? 0 : rand() % need[customerNum][1];

        int need\_3 =  need[customerNum][2] == 0 ? 0 : rand() % need[customerNum][2];

        int request\_one[] = {need\_1, need\_2, need\_3};

        request\_resources\_control(request\_one,customerNum);

    }

    return 0;

}

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

    //check input:

    if (argc < NUMBER\_OF\_RESOURCES + 1) {

        printf("not enough arguments!\n");

        exit(1);

    }

    //initialization of our data structures:

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

        //available[i] = strtol(argv[i + 1], NULL, 10);

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

    }

    for (int i = 0; i < 5; i++) {

        for (int j = 0; j < 3; j++){

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

        }

    }

    // int req[3] = {1, 2, 2};

    // request\_resources\_control(req, 1);

    // create the threads:

    int pid[] = {0,1,2,3,4};

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

        pthread\_create(&(tid[i]),NULL,getResources,&pid[i]);

    }

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

        pthread\_join(tid[i], NULL);

    }

    printf("FINISH!\n");

    pthread\_mutex\_destroy(&lock1);

    pthread\_mutex\_destroy(&lock2);

    return 0;

}

در تابع resources\_request الگوريتم بانکداران رخ ميدهد. بدين صورت که اين تابع شماره يک ترد و مقدار منابعي که درخواست کرده را به عنوان ورودي ميگيرد و الگوريتم را به کمک آنها شروع ميکند. ابتدا بررسي ميکند اگر تعداد منابع درخواست شده از تعداد منابع دردسترس (available) و يا تعداد منابعي که ادعا شده است اين ترد نياز دارد )need )بيشتر باشد پيغام خظا چاپ کرده و اين درخواست در همينجا خاتمه ميابد. اما اگر مقدار درخواستي مجاز باشد ابتدا فرض ميکنيم که تخصيص منبع، ما را دچار شرايط نا امن نميکند و منابع را اختصاص ميدهيم.

سپس به کمک تابع isSafe بررسي ميکنيم آيا فرض ما درست بوده است يا خير يعني آيا با تخصيص منابع درخواستي هنوز ميتوان در حالت امن باقي ماند يا نه. بدين منظور سعي ميکنيم دنباله اي از روند اجراي ترد ها را پيدا کنيم که دچار deadlock نشوند اگر توانستيم چنين ترتيبي را پيدا کنيم يعني در حالت امن هستيم و مشکلي براي تخصيص منابع نداريم پس کار را ادامه ميدهيم اما اگر چنين ترتيبي پيدا نشد يعني حالت امن وجود ندارد پس نميتوانيم منابع را به صورت امن به ترد درخواست کننده بدهيم پس منابع را از آن پس ميگيريم.

اگر توانستيم منابع را تخصيص دهيم يعني از مقدار need اين ترد کم شده و اگر حالتي پيش آيد که مقدار need براي يک ترد به صفر برسد يعني اين ترد کارش به طور کلي تمام ميشود پس ميتواند منابعي را که Allocate کرده آزاد کند. براي اين کار از تابع resources\_release استفاده ميکنيم.

لزم به ذکر است که براي جلوگيري از به وجود آمدن condition race بين ترد ها بايد فرآيند اجراي الگوريتم بانکداران و تخصيص منابع و آزادسازي منابع را به صورت اتميک پيش ببريم به همين منظور تابع resources\_request و resources\_release را در تابع ديگري با همين نام به عالوه پسوند control بين يک mutex قفل کردن و آزادسازي فراخواني ميکنيم.

خروجی به شکل زیر است:

