دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده 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.

```c
#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 الگوریتم بانکداران رخ میدهد. بدین صورت که این تابع شماره یک ترد و مقدار منابعی که درخواست کرده را به عنوان ورودی میگیرد و الگوریتم را به کمک آنها شروع میکند .ابتدا بررسی میکند اگر تعداد منابع درخواست شده از تعداد منابع دردسترس (available) و یا تعداد منابعی که ادعا شده است این ترد نیاز دارد(need) بیشتر باشد پیغام خطا چاپ کرده و این درخواست در همینجا خاتمه میابد .اما اگر مقدار درخواستی مجاز باشد ابتدا فرض میکنیم که تخصیص منبع، ما را دچار شرایط نا امن نمیکند و منابع را اختصاص میدهیم.

سپس به کمک تابع isSafe بررسی میکنیم آیا فرض ما درست بوده است یا خیر یعنی آیا با تخصیص منابع درخواستی هنوز میتوان در حالت امن باقی ماند یا نه. بدین منظور سعی میکنیم دنباله ای از روند اجرای ترد ها را پیدا کنیم که دچار deadlock نشوند اگر توانستیم چنین ترتیبی را پیدا کنیم یعنی در حالت امن هستیم و مشکلی برای تخصیص منابع نداریم پس کار را ادامه میدهیم اما اگر چنین ترتیبی پیدا نشد یعنی حالت امن وجود ندارد پس نمیتوانیم منابع را به صورت امن به ترد درخواست کننده بدهیم پس منابع را از آن پس میگیریم.

اگر توانستیم منابع را تخصیص دهیم یعنی از مقدار need این ترد کم شده و اگر حالتی پیش آید که مقدار need برای یک ترد به صفر برسد یعنی این ترد کارش به طور کلی تمام میشود پس میتواند منابعی را که Allocate کرده آزاد کند. برای این کار از تابع resources_release استفاده میکنیم.

لزم به ذکر است که برای جلوگیری از به وجود آمدن condition race بین ترد ها باید فرآیند اجرای الگوریتم بانکداران و تخصیص منابع و آزادسازی منابع را به صورت اتمیک پیش ببریم به همین منظور تابع

resources_request و resources_release را در تابع دیگری با همین نام به عالوه پسوند control بین یک mutex قفل کردن و آزادسازی فراخوانی میکنیم.

خروجی به شکل زیر است:

```
FINISH!
cavendish@LAPTOP-J4FO4O81:/mnt/c/Users/98912/Desktop/uni/OS LAB/EXP 7$ ./banker 4 4 6
------------------
Customer 0 is Requesting Resources:
8  4  2
Available = 4  4  6
Need = 9  9  4
Request is more than available! ABORT!
------------------
Customer 1 is Requesting Resources:
0  0  1
Available = 4  4  6
Need = 0  4  6
Not safe! Can't grant request!
------------------
Customer 2 is Requesting Resources:
0  0  1
Available = 4  4  6
Need = 0  8  2
Not safe! Can't grant request!
------------------
Customer 3 is Requesting Resources:
0  4  0
Available = 4  4  6
Need = 4  9  1
Not safe! Can't grant request!
------------------
Customer 4 is Requesting Resources:
2  1  1
Available = 4  4  6
Need = 3  2  3
Not safe! Can't grant request!
FINISH!
cavendish@LAPTOP-J4FO4O81:/mnt/c/Users/98912/Desktop/uni/OS LAB/EXP 7$
```