دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

گزارش کار آزمایشگاه سیستم عامل – شماره ۷

بن بست و الگوریتم بانکداران

حسنا اویارحسینی – ۹۸۲۳۰۱۰

استاد درس: جناب آقای مهندس کیخا

نیمسال دوم سال تحصیلی ۰۱-۱۴۰۰

بخش ۱) الگوریتم بانکداران را پیاده سازی کنید.

در این آزمایش یک برنامه چندنخی مینویسیم که الگوریتم بانکداران را پیاده سازی و تست کنیم. ابتدا به توضیح الگوریتم بانکداران در کد میپردازیم.

در این الگوریتم با فرض اینکه m نوع منبع و n ترد داشته باشیم، ۴ آرایه اصلی به صورت زیر داریم:

آرایه	توضیح
`available[m]`	تعداد `instance` های آزاد از هر نوع منبع
`maximum[n][m]`	حداکثر تعداد `instance` از هر منبع که یک ترد نیاز دارد
`allocation[n][m]`	تعداد `instance` از هر منبع که یک ترد در حال حاضر در اختیار دارد
`need[n][m]`	تعداد `instance` از هر منبع که یک ترد در حال حاضر نیاز دارد

در کد داریم:

```
#define NUMBER_OF_CUSTOMERS ۵
#define NUMBER_OF_RESOURCES ۳
/* 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] = {
                                                        { ۷, ۵, ۳ },
                                                        { ۳, ۲, ۲ },
                                                        { ۹, ۰, ۲ },
                                                        { ۲, ۲, ۲ },
                                                        { ٤, ۳, ۳ }
                                                        };
/* the amount currently allocated to each customer */
int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES] ={
                                                            { ۰, ۱, ۰ },
                                                            { ۲, ۰, ۰ },
                                                            { ۳, ۰, ۲ },
                                                            { ۲, ۱, ۱ },
                                                            { ۰, ۰, ۲ }
                                                        };
/* the remaining need of each customer */
int need[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
```

در تابع request_resources 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:

```
int request_resources(int request[], int customer_num) {

    printf("Customer %d is Requesting Resources:\n", customer_num);
    for (int i = ٠; i < NUMBER_OF_RESOURCES; i++) {
        printf("%d  ",request[i]);
    }

    printf("\nAvailable = ");
    for (int i = ٠; i < NUMBER_OF_RESOURCES; i++) {
        printf("%d  ", available[i]);
    }

    printf("\nNeed = ");
    for (int i = ٠; i < NUMBER_OF_RESOURCES; i++) {
        printf("%d  ", need[customer_num][i]);
    }
    printf("\n");

    for (int i = ٠; i < NUMBER_OF_RESOURCES; i++) {
        if (request[i] > need[customer_num][i]) {
            printf("Request is more than need! ABORT!\n");
            return -١;
           }
    }
    for (int i = ٠; i < NUMBER_OF_RESOURCES; i++) {
        if (request[i] > available[i]) {
            printf("Request is more than available! ABORT!\n");
            return -١;
           }
    }

    for(int j=٠ ; 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);

            }
        }

        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;
    }

}
```

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.

```
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;
        }
}
```

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 release_resources استفاده میکنیم که در آن منابعی که ترد گرفته بود (به اندازه maximum) را به منابع دردسترس اضافه میکنیم.

```
int release_resources(int release[], int customer_num){
    //give back resources:
    for(int i=۰ ; i<NUMBER_OF_RESOURCES ; i++){
        available[i] += release[i];
    }
    return ۰;
}
```

لازم به ذکر است که برای جلوگیری از به وجود آمدن race condition بین ترد ها باید فرآیند اجرای الگوریتم بانکداران و تخصیص منابع و آزادسازی منابع را به صورت اتمیک پیش ببریم به همین منظور تابع request_resources و release_resources را در تابع دیگری با همین نام به علاوه پسوند control بین یک mutex قفل کردن و آزادسازی فراخوانی میکنیم. با اینکار مطمئن میشویم که در هر لحظه از زمان فقط یک ترد این توابع را اجرا میکند.

```
pthread_mutex_t lock۱;
pthread_mutex_t lock۲;
void release_resources_control(int release[], int customer_num){
    pthread_mutex_lock(&lock۲);
    release_resources(release,customer_num);
    pthread_mutex_unlock(&lock۲);
    printf("Thread %d finished execution \n",customer_num);
}

bool request_resources_control(int request[],int customer_num){
    //CRITICAL SECTION //
    bool released = false;
    pthread_mutex_lock(&lock۱);
    printf("------------------\n");
    released=request_resources(request, customer_num);
    pthread_mutex_unlock(&lock۱);
    return released;
}
```

برای تست کردن این برنامه ۵ ترد میسازیم و برای هر کدام به تعداد NUM (در اینجا ۱) بار درخواست تعدادی منبع به صورت تصادفی انجام مدهیم (تعداد منابع را کمتر از need تنظیم میکنیم) و مشاهده میکنیم که کدام یک از درخواست ها امن و کدامیک نا امن بوده است.

```
pthread_t tid[NUMBER_OF_CUSTOMERS];
void* getResources(void *arg){

    int customerNum = *(int *)arg;

    for(int i=٠; i<NUM; i++){
        srand(time(NULL));
        //a random request
        int need_١ =  need[customerNum][٠] == ٠ ? ٠ : rand() %
need[customerNum][٠];
        int need_٢ =  need[customerNum][١] == ٠ ? ٠ : rand() %
need[customerNum][١];
        int need_٣ =  need[customerNum][٢] == ٠ ? ٠ : rand() %
need[customerNum][٢];

        int request_one[] = {need_١, need_٢, need_٣};

        request_resources_control(request_one,customerNum);
    }

    return ٠;
}

int main(int argc, char *argv[]) {
    //check input:
    if (argc < NUMBER_OF_RESOURCES + ١) {
        printf("not enough arguments!\n");
        exit(١);
    }

    //initialization of our data structures:
    for (int i = ٠; i < NUMBER_OF_RESOURCES; i++) {
        //available[i] = strtol(argv[i + ١], NULL, ١٠);
        available[i] = atoi(argv[i+١]);
    }



    for (int i = ٠; i < ٥; i++) {
        for (int j = ٠; j < ٣; j++){
```

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

    // create the threads:
    int pid[] = {٠,١,٢,٣,٤};
    for (int i=٠; i<NUMBER_OF_CUSTOMERS ; i++){
        pthread_create(&(tid[i]),NULL,getResources,&pid[i]);
    }

    for (int i=٠; i<NUMBER_OF_CUSTOMERS ; i++){
        pthread_join(tid[i], NULL);
    }

    printf("FINISH!\n");
    pthread_mutex_destroy(&lock١);
    pthread_mutex_destroy(&lock٢);

    return ٠;
}
```

خروجی کد:

برای مثال خروجی کد در یکبار اجرا به صورت زیر می باشد که حالت های مختلف در آن رخ داده است:

```
lab@ubuntu:~/Desktop/HW7$ ./banker 3 3 2
-----------------
Customer 0 is Requesting Resources:
1  2  2
Available = 3  3  2
Need = 7  4  3
Not safe! Can't grant request!
-----------------
Customer 1 is Requesting Resources:
0  0  0
Available = 3  3  2
Need = 1  2  2
Safe! Request is granted!
-----------------
Customer 4 is Requesting Resources:
1  0  0
Available = 3  3  2
Need = 4  3  1
Safe! Request is granted!
-----------------
Customer 2 is Requesting Resources:
3  0  0
Available = 2  3  2
Need = 6  0  0
Request is more than available! ABORT!
-----------------
Customer 3 is Requesting Resources:
0  0  0
Available = 2  3  2
Need = 0  1  1
Safe! Request is granted!
FINISH!
lab@ubuntu:~/Desktop/HW7$
```

برای مثال در حالت اول که درخواست ناامن بوده داریم:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

۰ request ۱ ۲ ۲

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	۱ | ۳ | ۲	7 5 3	۰ | ۰ | ۲
P_1	2 0 0	3 2 2	از تمامی
P_2	3 0 2	9 0 2	need ها
P_3	2 1 1	2 2 2	کمتر است
P_4	0 0 2	4 3 3	

Process	Need		
	A	B	C
P_0	٦ | ۲ | ۱		
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

چون در مثال بالا حالتی که تمامی منابع مربوط به یک ترد درخواست داده شود رخ نداده برای اینکه خروجی در این حالت را هم ببینیم یک درخواست برای دریافت تمامی منابع مورد نیاز یک ترد داده ایم تا عملکرد برنامه را برای آزاد سازی منابع نیز بررسی کنیم:

```
lab@ubuntu:~/Desktop/HW7$ gcc -pthread -o banker Banker\'s\ Algo.c
lab@ubuntu:~/Desktop/HW7$ ./banker 3 3 2
------------------
Customer 1 is Requesting Resources:
1  2  2
Available = 3  3  2
Need = 1  2  2
Safe! Request is granted!
1 got all it needed!
Thread 1 finished execution
FINISH!
lab@ubuntu:~/Desktop/HW7$
```