تمرین سری چهارم سیستم عامل

سوال یک

در این سوال در ابتدا با استفاده از define تعداد مشتریان و تراکنش ها (همان تعداد thread ها)

```
#define NUM_CUSTOMERS 5
#define NUM_TRANSACTION 6
#define INIT_STOCK 100

typedef struct
{
    double amount;
    int customer_id;
} transaction_t;

double safeBox = 1000.0;
```

و موجودی اولیه حساب هر مشتری را تعیین میکنیم . سپس برای هر تراکنش یک struct داریم که در آن مشخص میکنیم مقدار تراکنش چقدر بوده و مربوط به کدام مشتری است که از طریق customer_id مشخص میشود .

برای safebox هم مقدار اولیه 1000 تعیین میکنیم .

برای نگهداری موجودی مشتری ها آرایه ای به اندازه تعداد مشتری ها تعیین میکنیم . همچنین یک آرایه از mutex برای موجودی مشتری ها معین میکنیم که تراکنش ها همزمان تغییراتی روی موجودی یک مشتری ایجاد نکنند تا inconsistency داده به وجود نیاید و همینطور یک mutex روی safebox ایجاد میکنیم تا چند تراکنش همزمان موجودی آن را تغییر ندهند

```
double customerStock[NUM_CUSTOMERS];
pthread_mutex_t customerMutex[NUM_CUSTOMERS];
pthread_mutex_t safeBoxMutex = PTHREAD_MUTEX_INITIALIZER;
```

حال برای پیاده سازی تابع withdraw داریم :

```
void* withdraw(void* arg)
```

ابتدا تراکنش موجود را به یکی از مشتری های تخصیص میدهیم سپس برای اعمال تغییرات روی موجودی مشتری mutex_lock را روی موجودی حساب آن مشتری صدا میزنیم تا همزمان تغییرات دیگری روی موجودی حساب اعمال نشود .

```
 transaction_t* transaction = (transaction_t*)arg;
 int customerID = transaction->customer_id;

pthread_mutex_lock(&customerMutex[customerID]);
```

چک میکنیم که موجودی حساب مشتری اگر از مقدار مد نظر برای برداشت بیشتر یا مساوی باشد کاری به safebox نداریم و مستقیما کل مقدار را از موجودی مشتری کم میکنیم و صرفا برای چاپ مقدار safebox همراه با مقدار برداشت شده و موجودی مشتری mutex_lock را روی safebox صدا میزنیم و پس از چاپ آن را unlock میکنیم .

```
if (customerStock[customerID] >= transaction->amount)
{
     pthread_mutex_lock(&safeBoxMutex);
     customerStock[customerID] -= transaction->amount;
     printf("Customer %d withdrew %.2f. New balance: %.2f\t ------------ > safe-
box value: %.2f\n",customerID, transaction->amount, customerStock[customerID],
safeBox);
     pthread_mutex_unlock(&safeBoxMutex);
     pthread_mutex_unlock(&customerMutex[customerID]);

}
```

حال حالتی را چک میکنیم که مقدار مد نظر برای برداشت بیش از موجودی مشتری باشد در این صورت نیاز هست که از safebox هم برداشت کنیم در نتیجه ابتدا چک میکنیم که موجودی مشتری چقدر بوده و چه مقدار را از حساب مشتری و چه مقدار را از safebox برداشت میکنیم در نتیجه ابتدا mutex_lock را روی safebox صدا میزنیم تا تراکنش دیگری به صورت همزمان روی موجودی آن تغییرات ایجاد نکند داخل شروط چک میکنیم که اگر موجودی حساب مشتری منفی باشد یعنی به safebox بدهکار بوده و باید مبلغ را به طور کل از safebox برداشت کنیم همچنین چک میکنیم اگر موجودی safebox کمتر از مقدار مدنظر جهت برداشت باشد تراکنش انجام نخواهد شد و موجودی همانند قبل میمانند .

```
else
    {
        double amountTakenFromBalance = customerStock[customerID];
        pthread_mutex_lock(&safeBoxMutex);

        if( amountTakenFromBalance <= 0 )
        {
            if(safeBox < transaction->amount)
            {
                printf("This transition from Customer %d has this amount %.2f and
can not be completed cause safebox has %.2f and customer balance is %.2f  and it
is not enough!\n",customerID,transaction-
>amount,safeBox,customerStock[customerID]);
                pthread_mutex_unlock(&safeBoxMutex);
            }
            else
            {
                safeBox -= transaction->amount;
                customerStock[customerID] -= transaction->amount;
                printf("Customer %d withdrew 0 from account balance and %.2f from
safe-box . New balance: %.2f\t ------------ > safe-box value: %.2f\n",
                    customerID,transaction->amount,customerStock[customerID],
safeBox);
                pthread_mutex_unlock(&safeBoxMutex);
            }

        }
        else
        {
            customerStock[customerID] -= transaction->amount;
            if(safeBox < (-1 *customerStock[customerID]))
            {
                printf("This transition from customer %d has this amount %.2f and
can not be completed cause safebox has %.2f and customer balance is %.2f and it
is not enough! !!!!!\n",customerID,transaction-
>amount,safeBox,customerStock[customerID]);
                pthread_mutex_unlock(&safeBoxMutex);
            }
            else
            {

                safeBox += customerStock[customerID];
                printf("Customer %d withdrew %.2f from account balance and %.2f
from safe-box means %.2f  from both . New balance: %.2f\t ------------ > safe-box
value: %.2f\n",
```

```
                    customerID,amountTakenFromBalance,(-1
*customerStock[customerID]),transaction->amount,customerStock[customerID],
safeBox);
                pthread_mutex_unlock(&safeBoxMutex);
            }

        }

        pthread_mutex_unlock(&customerMutex[customerID]);
    }
```

حال برای پیاده سازی تابع deposit داریم :

```
oid* deposit(void* arg)
```

ابتدا تراکنش موجود را به یکی از مشتری های تخصیص میدهیم سپس برای اعمال تغییرات روی موجودی مشتری mutex_lock را روی موجودی حساب آن مشتری صدا میزنیم تا همزمان تغییرات دیگری روی موجودی حساب اعمال نشود .

```
transaction_t* transaction = (transaction_t*)arg;
    int customerID = transaction->customer_id;

    pthread_mutex_lock(&customerMutex[customerID]);
```

چک میکنیم که موجودی حساب مشتری اگر منفی باشد یعنی به safebox بدهکار است و اولویت پرداخت بدهی به safebox است . و داخل شروط چک میکنیم که اگر پس از پرداخت بدهی مبلغی از مقدار واریزی باقی ماند به حساب شخص واریز میکنیم در غیر این صورت صرفا بدهی شخص به safebox کمتر یا صفر میشود . و در هر مورد برای آنکه روی safebox تراکنش دیگری همزمان اجرا نشود بعد از چک کردن شرط و قبل از اعمال تراکنش mutex_lock را روی safebox صدا میزنیم و پس از تراکنش آن را آزاد میکنیم .

```
if(customerStock[customerID] < 0)
{
    if(transaction->amount > (customerStock[customerID] * -1))
    {
     pthread_mutex_lock(&safeBoxMutex);
     double amountGiven = transaction->amount + customerStock[customerID];
     safeBox += (customerStock[customerID] * -1);
```

```
     customerStock[customerID] += amountGiven + (customerStock[customerID] * -1);
     printf("Customer %d stock was charged by %.2f. New balance: %.2f\t ---------
--- > safe-box value: %.2f\n",
     customerID, transaction->amount, customerStock[customerID], safeBox);
     pthread_mutex_unlock(&safeBoxMutex);
     pthread_mutex_unlock(&customerMutex[customerID]);
     }
    else
    {
     pthread_mutex_lock(&safeBoxMutex);
     safeBox += transaction->amount ;
     customerStock[customerID] += transaction->amount;
     printf("Customer %d stock was charged by %.2f. New balance: %.2f\t --------
-      --- > safe-box value: %.2f\n",
     customerID, transaction->amount, customerStock[customerID], safeBox);
     pthread_mutex_unlock(&safeBoxMutex);
     pthread_mutex_unlock(&customerMutex[customerID]);
    }
}
```

و اگر موجودی حساب منفی نباشد مستقیما به حساب مشتری واریز میکنیم و صرفا برای چاپ موجودی ها mutex_lock را روی safebox صدا میزنیم چرا که در این بخش عملیاتی جهت افزودن یا کاستن بر روی safebox انجام نمیشود و فقط میخواهیم موجودی به درستی نمایش داده شود .

```
else
    {
        pthread_mutex_lock(&safeBoxMutex);
        customerStock[customerID] += transaction->amount;
        printf("Customer %d stock was charged by %.2f. New balance: %.2f\t ------
------ > safe-box value: %.2f\n",
            customerID, transaction->amount, customerStock[customerID], safeBox);
        pthread_mutex_unlock(&safeBoxMutex);
        pthread_mutex_unlock(&customerMutex[customerID]);
    }
```

در تابع main ابتدا mutex های مطرح شده و موجودی حساب مشتری ها را مقدار دهی اولیه میکنیم . و آرایه ای از thread ها به اندازه تعداد تراکنش ها ایجاد میکنیم سپس در حلقه ای به اندازه تعداد تراکنش ها مقدار تراکنش و شماره مشتری را به صورت رندوم انتخاب میکنیم و با توجه به یک عدد رندوم دیگر معین میکنیم که تراکنش برداشت یا واریز باشد و thread های موجود در آرایه ساخته شده را ایجاد میکنیم و توابع واریز و برداشت را با pthread_create به آنها assign میکنیم و پس از اتمام حلقه با

استفاده از pthread_join و pthread_mutex_destroy , thread ها و mutex هایی که ساختیم را از بین میبریم .

```
for (int i = 0; i < NUM_CUSTOMERS; ++i)
{
   customerStock[i] = INIT_STOCK;
   pthread_mutex_init(&customerMutex[i], NULL);
}

   pthread_t threads[NUM_TRANSACTION];

for (int i = 0; i < NUM_TRANSACTION; ++i)
{
        transaction_t* transaction =
(transaction_t*)malloc(sizeof(transaction_t));
        transaction->amount = rand() % 800 + 1;
        transaction->customer_id = rand() % NUM_CUSTOMERS;

        if (rand() % 2 == 0)
        {
            pthread_create(&threads[i], NULL, withdraw, (void*)transaction);
        } else {
            pthread_create(&threads[i], NULL, deposit, (void*)transaction);
        }
}

for (int i = 0; i < NUM_TRANSACTION; ++i)
{
   pthread_join(threads[i], NULL);
}

for (int i = 0; i < NUM_CUSTOMERS; ++i)
{
   pthread_mutex_destroy(&customerMutex[i]);
}
```