**Ans:**

Here,

Number of threads = 12

Number of tickets left = 10

Additionally, all the threads should be executed sequentially.

It is instructed that 2 threads come together and get executed sequentially, there are 2 ways we can implement this. Those are:

1. Get 2 threads to the execution area and execute those sequentially.
2. Get all the threads to the execution area and execute those sequentially, as pairwise and non-pairwise execution will result in the same outcome.

If we work in the second way, we should use threads along with semaphores for each of them. The initial value for all the semaphores should be 0. So, no thread can be executed at this point.

Then, the semaphore for the first thread gets called with the signal() (or sem\_post()) function, which increases the value of semaphore by 1. That makes the first thread executable now.

Whenever a thread gets processed, we firstly wait for the semaphore value of that thread to be positive by the wait() function. If the semaphore value is positive for that thread, it becomes free to execute. At this stage, we check for the ticket count and if any ticket is available, we successfully make the purchase and opposite otherwise. If it is a successful transaction, we decrease the available ticket count. As the variable count is shared among all the threads, we need the semaphore variable to maintain the order. Lastly, we invoke another signal() call to the next thread to execute the next thread.

In this process, all the threads will be processed sequentially, which makes it a perfect solution for the problem.

A suitable solution by C program for this problem is provided below:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <semaphore.h>

// number of threads

#define n 12

int persons[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};

int count = 10;  // tickets left

// a semaphore for each thread

sem\_t\* semaphores;

void \*t\_func(int \*idp)

{

    int id = \* (int\*) idp;

    // wait for our semaphore to become unlocked

    sem\_wait(&semaphores[id]);

    if (count > 0)

    {

        printf("Person %d is purchasing\n", id+1);

        printf("Tickets left: %d\n", count);

        count--;

        printf("Person: %d, Purchase Done\n", id+1);

    }

    else

    {

        printf("Person %d can not purchase\n", id+1);

        printf("Tickets left: %d\n", count);

        printf("Person: %d, Purchase failed\n", id+1);

    }

    // unlock the semaphore belonging to the next thread (unless last)

    if (id < (n - 1)) {

        sem\_post(&semaphores[id + 1]);

    }

}

int main()

{

    pthread\_t t[n];

    // create the semaphores

    sem\_t s[n];

    semaphores = s;

    // initialize the semaphores to 0

    for (int i = 0; i < n; i++) {

        sem\_init(&semaphores[i], 0, 0);

    }

    // unlock thread 0

    sem\_post(&semaphores[0]);

    // create the threads

    for (int i = 0; i < n; i++)

    {

        pthread\_create(&t[i], NULL, (void \*)t\_func, &persons[i]);

    }

    // wait for the threads to finish

    for (int i = 0; i < n; i++)

    {

        pthread\_join(t[i], NULL);

    }

    // destroy the semaphores

    for (int i = 0; i < n; i++) {

        sem\_destroy(&semaphores[i]);

    }

    free(semaphores);

    return 0;

}