Name : Randrita Sarkar U**niversity Roll** : 11500219058

IT PCC-CS592 L - OPERATING SYSTEMS LAB

1. In a ticket counter there are many ticket-window (say 4) Ticket sellers are selling tickets to customer, ticket sellers are only allowed to sell ticket until they are all gone(say 100). Before attempting to sell a ticket, the thread must acquire the lock by waiting on the semaphore and then release the lock when through by signalling the semaphore.There is a global variable numTickets which tracks the number of tickets remaining to sell. Implement system program to demonstrate this scenario.

Code :

```
/*
 * ticketSeller.c
 * --------------
 * A very simple example of a critical section that is
protected by a
 * semaphore lock. There is a global variable
numTickets which tracks the
 * number of tickets remaining to sell. We will create
many threads that all
 * will attempt to sell tickets until they are all
gone. However, we must
 * control access to this global variable lest we sell
more tickets than
 * really exist. We have a semaphore lock that will
only allow one seller
```

```
 * thread to access the numTickets variable at a time.
Before attempting to
 * sell a ticket, the thread must acquire the lock by
waiting on the semaphore
 * and then release the lock when through by
signalling the semaphore.
 */
#include "thread_107.h"
#include <stdio.h>
#include <string.h>
#define NUM_TICKETS 35
#define NUM_SELLERS 4
/**
 * The ticket counter and its associated lock will be
accessed
 * all threads, so made global for easy access.
 */
static int numTickets = NUM_TICKETS;
static Semaphore ticketsLock;
/**
 * SellTickets
 * -----------
 * This is the routine forked by each of the ticket-
selling threads.
 * It will loop selling tickets until there are no
more tickets left
 * to sell. Before accessing the global numTickets
variable,
 * it acquires the ticketsLock to ensure that our
threads don't step
 * on one another and oversell on the number of
tickets.
 */
static void SellTickets(void)
```

```
{
    bool done = false;
    int numSoldByThisThread = 0; // local vars are
unique to each thread
    while (!done)
    {
        /**
 * imagine some code here which does something
independent of
 * the other threads such as working with a customer
to determine
 * which tickets they want. Simulate with a small
random delay
 * to get random variations in output patterns.
 */
        RandomDelay(500000, 2000000);
        SemaphoreWait(ticketsLock); // ENTER CRITICAL
SECTION
        if (numTickets == 0)
        {                   // here is safe to access
numTickets
            done = true; // a "break" here instead of
done variable
                        // would be an error- why?
        }
        else
        {
            numTickets--;
            numSoldByThisThread++;
            printf("%s sold one (%d left)\n",
ThreadName(), numTickets);
        }
        SemaphoreSignal(ticketsLock); // LEAVE
CRITICAL SECTION
```

```
    }
    printf("%s noticed all tickets sold! (I sold %d
myself) \n", ThreadName(), numSoldByThisThread);
}
/**
 * RandomDelay
 * -----------
 * This is used to put the current thread to sleep for
a little
 * bit to simulate some activity or perhaps just to
vary the
 * execution patterns of the thread scheduling. The
low and high
 * limits are expressed in microseconds, the thread
will sleep
 * at least the lower limit, and perhaps as high as
upper limit
 * (or even more depending on the contention for the
processors).
 */
static void RandomDelay(int atLeastMicrosecs, int
atMostMicrosecs)
{
    long choice;
    int range = atMostMicrosecs - atLeastMicrosecs;
    PROTECT(choice = random());                    //
protect non-re-entrancy
    ThreadSleep(atLeastMicrosecs + choice % range); //
put thread to sleep
}
/**
 * Our main is creates the initial semaphore lock in
an unlocked state
```

```
 * (one thread can immediately acquire it) and sets up
all of
 * the ticket seller threads, and lets them run to
completion. They
 * should all finish when all tickets have been sold.
By running with the
 * -v flag, it will include the trace output from the
thread library.
 */
void main(int argc, char **argv)
{
    int i;
    char name[32];
    bool verbose = (argc == 2 && (strcmp(argv[1], "-
v") == 0));
    InitThreadPackage(verbose);
    ticketsLock = SemaphoreNew("Tickets Lock", 1);
    for (i = 0; i < NUM_SELLERS; i++)
    {
        sprintf(name, "Seller #%d", i); // give each
thread a distinct name
        ThreadNew(name, SellTickets, 0);
    }
    RunAllThreads();            // Let all threads
loose
    SemaphoreFree(ticketsLock); // to be tidy, clean
up
    printf("All done!\n");
}
```

Output :

Output

```
epic18:/usr/class/cs107/other/thread_examples>ticketSeller
Seller #1 sold one (34 left)
Seller #0 sold one (33 left)
Seller #1 sold one (32 left)
Seller #1 sold one (31 left)
Seller #1 sold one (30 left)
Seller #1 sold one (29 left)
Seller #1 sold one (28 left)
Seller #2 sold one (27 left)
Seller #3 sold one (26 left)
Seller #2 sold one (25 left)
Seller #3 sold one (24 left)
Seller #2 sold one (23 left)
Seller #0 sold one (22 left)
Seller #1 sold one (21 left)
Seller #2 sold one (20 left)
Seller #0 sold one (19 left)
Seller #1 sold one (18 left)
Seller #1 sold one (17 left)
Seller #3 sold one (16 left)
Seller #2 sold one (15 left)
Seller #0 sold one (14 left)
Seller #0 sold one (13 left)
Seller #1 sold one (12 left)
Seller #3 sold one (11 left)
Seller #2 sold one (10 left)
Seller #0 sold one (9 left)
Seller #0 sold one (8 left)
Seller #1 sold one (7 left)
Seller #3 sold one (6 left)
Seller #2 sold one (5 left)
Seller #0 sold one (4 left)
Seller #1 sold one (3 left)
Seller #1 sold one (2 left)
Seller #1 sold one (1 left)
Seller #1 sold one (0 left)
Seller #3 noticed all tickets sold! (I sold 5 myself)
Seller #2 noticed all tickets sold! (I sold 7 myself)
Seller #1 noticed all tickets sold! (I sold 15 myself)
Seller #0 noticed all tickets sold! (I sold 8 myself)
All done!
```