| Ex. No. 8 | Threading and Synchronization | | |
|---|---|---|---|
| Date of Exercise | 05.10.2016 | Date of Upload | 06.11.2016 |

## AIM

To perform **Banking Operations** using C# by using the concept of multithreading and to use synchronization method so as to avoid race conditions.

## DESCRIPTION

A **thread** is defined as the execution path of a program. Each thread defines a unique flow of control. If your application involves complicated and time consuming operations, then it is often helpful to set different execution paths or threads, with each thread performing a particular job.

Threads are lightweight processes. One common example of use of thread is implementation of concurrent programming by modern operating systems. Use of threads saves wastage of CPU cycle and increase efficiency of an application.

### Thread Life Cycle

The life cycle of a thread starts when an object of the System.Threading.Thread class is created and ends when the thread is terminated or completes execution.

Following are the various states in the life cycle of a thread:

- **The Unstarted State**: It is the situation when the instance of the thread is created but the Start method is not called.
- **The Ready State**: It is the situation when the thread is ready to run and waiting CPU cycle.
- **The Not Runnable State**: A thread is not executable, when:
    - Sleep method has been called
    - Wait method has been called
    - Blocked by I/O operations
- **The Dead State**: It is the situation when the thread completes execution or is aborted.

The following program demonstrates main thread execution:

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
   class MainThreadProgram
   {
      static void Main(string[] args)
      {
         Thread th = Thread.CurrentThread;
         th.Name = "MainThread";
         Console.WriteLine("This is {0}", th.Name);
         Console.ReadKey();
      }
   }
}
```

## PROPERTIES AND METHODS OF THE THREAD CLASS

| Property | Description |
|---|---|
| CurrentContext | Gets the current context in which the thread is executing. |
| CurrentCulture | Gets or sets the culture for the current thread. |

| | |
|---|---|
| CurrentPrinciple | Gets or sets the thread's current principal (for role-based security). |
| CurrentThread | Gets the currently running thread. |
| CurrentUICulture | Gets or sets the current culture used by the Resource Manager to look up culture-specific resources at run-time. |
| ExecutionContext | Gets an ExecutionContext object that contains information about the various contexts of the current thread. |
| IsAlive | Gets a value indicating the execution status of the current thread. |
| IsBackground | Gets or sets a value indicating whether or not a thread is a background thread. |
| IsThreadPoolThread | Gets a value indicating whether or not a thread belongs to the managed thread pool. |
| ManagedThreadId | Gets a unique identifier for the current managed thread. |
| Name | Gets or sets the name of the thread. |
| Priority | Gets or sets a value indicating the scheduling priority of a thread. |
| ThreadState | Gets a value containing the states of the current thread. |

**THREADING ISSUES**

Programming with multiple threads is not easy. When starting multiple threads that access the same data, you can get intermittent problems that are hard to find. This is the same if you use tasks, Parallel LINQ, or the Parallel class. To avoid getting into trouble, you must pay attention to synchronization issues and the problems that can happen with multiple threads.

**RACE CONDITION**
A race condition can occur if two or more threads access the same objects and access to the shared state is not synchronized.

```
private object sync = new object();

lock (sync)

{

//method

}
```

**DEADLOCK**
Too much locking can get you in trouble as well. In a deadlock, at least two threads halt and wait for each other to release a lock. As both threads wait for each other, a deadlock occurs and the threads wait endlessly.

**SYNCHRONIZATION**

It is best to avoid synchronization issues by not sharing data between threads. Of course, this is not always possible. If data sharing is necessary, you must use synchronization techniques so that only one thread at a time accesses and changes shared state.

Various synchronization technologies that you can use with multiple threads:

- lock statement
- Interlocked class
- Monitor class
- SpinLock struct
- WaitHandle class
- Mutex class
- Semaphore class
- Events classes
- Barrier class
- ReaderWriterLockSlim class

## PROGRAM

### Banking program.cs

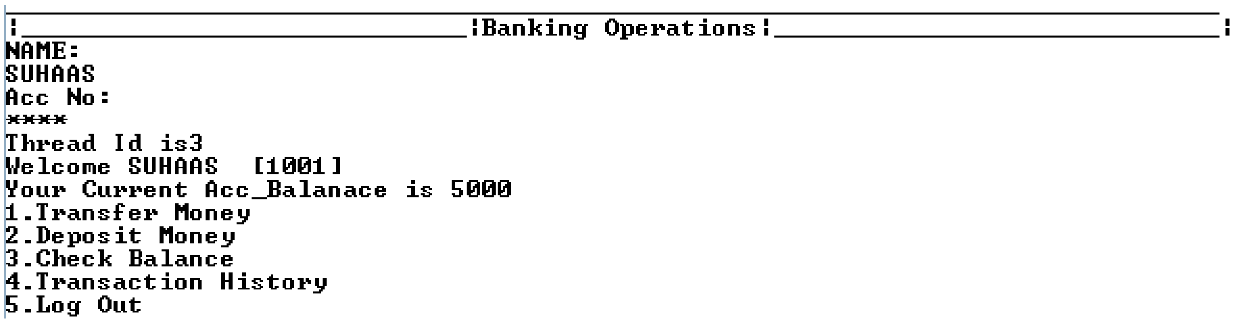
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace _8.Banking_Operations
{
    public class BankAccount {
        public string name;
        public string accno;
        public decimal accbal;
    }

    public class TransactionLog {
        public string logname;
        public string acc_no;
    }

    class Program
    {
        static List<TransactionLog> log = new List<TransactionLog>();
        static List<BankAccount> account = new List<BankAccount>();
        static void Main(string[] args)
        {

Console.WriteLine("______________________________________________________________________
_________");
            Console.WriteLine("|_____________________________|Banking
Operations|_____________________________|");


            InitializeAccounts(account);
            CheckPwd(account);
        }

        public static string ReadPassword()
        {
            string password = ""; ConsoleKeyInfo info = Console.ReadKey(true); while
(info.Key != ConsoleKey.Enter)
            {
                if (info.Key != ConsoleKey.Backspace) { Console.Write("*"); password +=
info.KeyChar; }
                else if (info.Key == ConsoleKey.Backspace)
                {
                    if (!string.IsNullOrEmpty(password))
                    {
                        // remove one character from the list of password characters
                        password = password.Substring(0, password.Length - 1); // get the
location of the cursor
                        int pos = Console.CursorLeft; // move the cursor to the left by
one character
                        Console.SetCursorPosition(pos - 1, Console.CursorTop); // replace
it with space
                        Console.Write(" "); // move the cursor to the left by one
character again
                        Console.SetCursorPosition(pos - 1, Console.CursorTop);
                    }
                }
                info = Console.ReadKey(true);
            } // add a new line because user pressed enter at the end of their password
            Console.WriteLine();
            return password;
        }

        public static void CheckPwd(List<BankAccount> account)
        {
            lOGIN:
            Console.WriteLine("NAME:");
            string name = Console.ReadLine().ToUpper();
            Console.WriteLine("Acc No:");
            string accno = ReadPassword();
            int index = -1;
            index = account.FindIndex(a => a.name.Equals(name) && a.accno.Equals(accno));
            if (index != -1)
            {
                //InitializeMenu(account, index, log);
                var t = new Thread(()=>InitializeMenu(account, index, log));
                t.Start();
            }
            else { Console.WriteLine("Bad Login"); goto lOGIN; }
        }

        public static void InitializeMenu(List<BankAccount> account,int index,
List<TransactionLog> log) {
            Console.WriteLine("Thread Id is"+Thread.CurrentThread.ManagedThreadId);
            Console.WriteLine("Welcome " + account[index].name
+"\t["+account[index].accno+"]");
            Console.WriteLine("Your Current Acc_Balanace is " + account[index].accbal);
            Console.WriteLine("1.Transfer Money");
            Console.WriteLine("2.Deposit Money");
            Console.WriteLine("3.Check Balance");
            Console.WriteLine("4.Transaction History");
            Console.WriteLine("5.Log Out");
            int choice = Convert.ToInt32(Console.ReadLine());
            switch (choice) {
                case 1:
                    Transfer_Money(account, index, log);
                    break;
                case 2:Deposit_Money(account, index, log);
                    break;
                case 3:CheckBalance(account, index, log);
                    break;
                case 4:Transaction_history(account, index, log);
                    break;
                case 5:
                    CheckPwd(account);
                    break;
                default:
                    Console.WriteLine("Invalid Choice");
                    InitializeMenu(account, index,log);
                    break;

            }
        }

        public static void CheckBalance(List<BankAccount> account, int index,
List<TransactionLog> log) {
            Console.WriteLine("|---------------------------Account Details---------------
------|");
            Console.WriteLine("|\tAccount Holder
Name\t{0}\t\t\t\t|",account[index].name);
            Console.WriteLine("|\tAccount Number\t\t{0}\t\t\t\t|" ,
account[index].accno);
            Console.WriteLine("|\tAccount Balance\t\t{0}\t\t\t\t|" ,
account[index].accbal);
            Console.WriteLine("|---------------------------------------------------------
------|");
            InitializeMenu(account, index, log);
        }

        public static void Transaction_history(List<BankAccount> account, int index,
List<TransactionLog> log) {
            string currnt_accno = account[index].accno;
            List<TransactionLog> Transactions = log.FindAll(a =>
a.acc_no.Equals(currnt_accno));
            for (int i=0; i < Transactions.Count;i++)
            {
                Console.WriteLine(Transactions[i].logname);
            }
            InitializeMenu(account, index, log);
        }

        private static  object lockMethod = new object();
        public static void Transfer_Money(List<BankAccount> account, int
index,List<TransactionLog> log) {

            lock (lockMethod)
            {
                Console.WriteLine("Enter the amount to be Transferred");
                decimal t_amt = Convert.ToDecimal(Console.ReadLine());
                decimal curr_amt = account[index].accbal;
                string currnt_accno = account[index].accno;
                if (t_amt < curr_amt)
                {
                    Console.WriteLine("Enter the Account No to be transferred : ");
                    string t_acc_no = Console.ReadLine();
                    int trans_account = account.FindIndex(a => a.accno.Equals(t_acc_no));
                    account[trans_account].accbal += t_amt;//Amount Transferred
                    account[index].accbal -= t_amt;//Amount Deducted
                    Why("Transferring");
                    Console.WriteLine("Transferred Successfully");
                    log.Add(new TransactionLog { logname = "" + System.DateTime.Now +
"\t" + "Transferred FROM " + currnt_accno + " TO " + t_acc_no, acc_no = currnt_accno });
                    InitializeMenu(account, index, log);
                }
                else {
                    Console.WriteLine("Insuffient Funds");
                    InitializeMenu(account, index, log);
                }
            }
        }

        public static void Deposit_Money(List<BankAccount> account, int index,
List<TransactionLog> log)
        {
            lock (lockMethod) {
                Console.WriteLine("Enter the amount to be Deposit");
                decimal t_amt = Convert.ToDecimal(Console.ReadLine());
                string currnt_accno = account[index].accno;
                if (t_amt > 0)
                {
                    account[index].accbal += t_amt;//Amount Debitted
                    Why("Debitting");
                    Console.WriteLine("Debitted Successfully");
                    log.Add(new TransactionLog { logname = "" + System.DateTime.Now +
"\t" + "Debitted to " + currnt_accno + " an amount of  " + account[index].accbal, acc_no
= currnt_accno });
                    InitializeMenu(account, index, log);
                }
                else
                {
                    Console.WriteLine("Incorrect Funds");
                    InitializeMenu(account, index, log);
                }
            }
        }

        public static void InitializeAccounts(List<BankAccount> account) {
            account.Add(new BankAccount { name="SUHAAS",accno="1001",accbal=5000});
            account.Add(new BankAccount { name = "SNEHAL", accno = "1002", accbal = 10000
});
            account.Add(new BankAccount { name = "ARUNA", accno = "1003", accbal = 15000
});
            account.Add(new BankAccount { name = "SRINIVAS", accno = "1004", accbal =
20000 });
        }

        static public void Why(string msg)
        {
            Console.Write(msg);
            using (var progress = new ProgressBar())
            {
                for (int i = 0; i <= 100; i++)
                {
                    progress.Report((double)i / 100);
                    Thread.Sleep(20);
                }
            }
            Console.WriteLine("Done.");
        }
    }
}
```

## PROGRESSBAR.CS

```
using System;
using System.Text;
using System.Threading;

/// <summary>
/// An ASCII progress bar
/// </summary>
public class ProgressBar : IDisposable, IProgress<double>
{
    private const int blockCount = 10;
    private readonly TimeSpan animationInterval = TimeSpan.FromSeconds(1.0 / 8);
    private const string animation = @"|/-\";

    private readonly Timer timer;

    private double currentProgress = 0;
    private string currentText = string.Empty;
    private bool disposed = false;
    private int animationIndex = 0;

    public ProgressBar()
    {
        timer = new Timer(TimerHandler);

        // A progress bar is only for temporary display in a console window.
        // If the console output is redirected to a file, draw nothing.
        // Otherwise, we'll end up with a lot of garbage in the target file.
        if (!Console.IsOutputRedirected)
        {
            ResetTimer();
        }
    }

    public void Report(double value)
    {
        // Make sure value is in [0..1] range
        value = Math.Max(0, Math.Min(1, value));
        Interlocked.Exchange(ref currentProgress, value);
    }

    private void TimerHandler(object state)
    {
        lock (timer)
        {
            if (disposed) return;

            int progressBlockCount = (int)(currentProgress * blockCount);
            int percent = (int)(currentProgress * 100);
            string text = string.Format("[{0}{1}] {2,3}% {3}",
                new string('#', progressBlockCount), new string('-', blockCount -
progressBlockCount),
                percent,
                animation[animationIndex++ % animation.Length]);
            UpdateText(text);

            ResetTimer();
        }
    }

    private void UpdateText(string text)
    {
        // Get length of common portion
        int commonPrefixLength = 0;
        int commonLength = Math.Min(currentText.Length, text.Length);
        while (commonPrefixLength < commonLength && text[commonPrefixLength] ==
currentText[commonPrefixLength])
        {
            commonPrefixLength++;
        }

        // Backtrack to the first differing character
        StringBuilder outputBuilder = new StringBuilder();
        outputBuilder.Append('\b', currentText.Length - commonPrefixLength);

        // Output new suffix
        outputBuilder.Append(text.Substring(commonPrefixLength));

        // If the new text is shorter than the old one: delete overlapping characters
        int overlapCount = currentText.Length - text.Length;
        if (overlapCount > 0)
        {
            outputBuilder.Append(' ', overlapCount);
            outputBuilder.Append('\b', overlapCount);
        }

        Console.Write(outputBuilder);
        currentText = text;
    }

    private void ResetTimer()
    {
        timer.Change(animationInterval, TimeSpan.FromMilliseconds(-1));
    }

    public void Dispose()
    {
        lock (timer)
        {
            disposed = true;
            UpdateText(string.Empty);
        }
    }

}
```

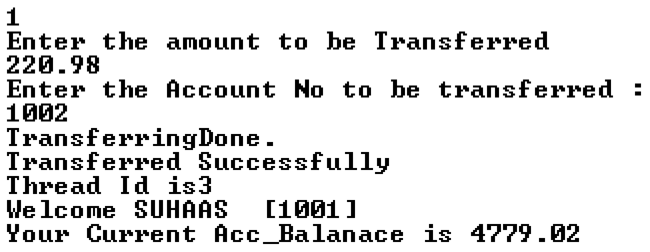
**OUTPUT**

**MENU**



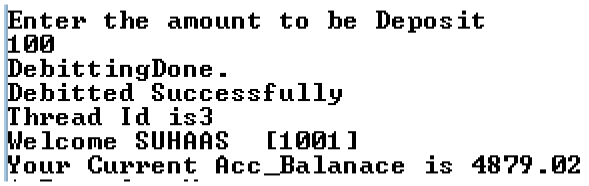
```
|________________________________|Banking Operations|________________________________|
NAME:
SUHAAS
Acc No:
****
Thread Id is3
Welcome SUHAAS  [1001]
Your Current Acc_Balanace is 5000
1.Transfer Money
2.Deposit Money
3.Check Balance
4.Transaction History
5.Log Out
```

**TRANSFER**



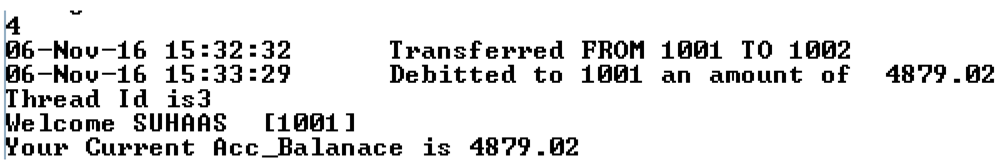
```
1
Enter the amount to be Transferred
220.98
Enter the Account No to be transferred :
1002
TransferringDone.
Transferred Successfully
Thread Id is3
Welcome SUHAAS  [1001]
Your Current Acc_Balanace is 4779.02
```

**DEPOSIT**



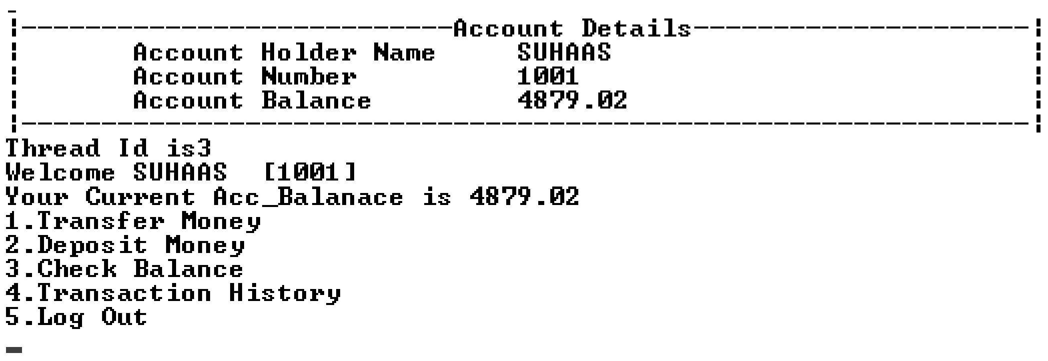
```
Enter the amount to be Deposit
100
DebittingDone.
Debitted Successfully
Thread Id is3
Welcome SUHAAS  [1001]
Your Current Acc_Balanace is 4879.02
```

**HISTORY**



```
4
06-Nov-16 15:32:32      Transferred FROM 1001 TO 1002
06-Nov-16 15:33:29      Debitted to 1001 an amount of  4879.02
Thread Id is3
Welcome SUHAAS  [1001]
Your Current Acc_Balanace is 4879.02
```

## BALANCE



```
-
|--------------------------Account Details--------------------|
|       Account Holder Name      SUHAAS                        |
|       Account Number           1001                          |
|       Account Balance          4879.02                       |
|-------------------------------------------------------------|
Thread Id is3
Welcome SUHAAS  [1001]
Your Current Acc_Balanace is 4879.02
1.Transfer Money
2.Deposit Money
3.Check Balance
4.Transaction History
5.Log Out
_
```

## Result

The above programme is compiled successfully and the screenshots are well described with successful outputs and constraints.

**[Dr J Anitha/Dr. S.P. Jeno Lovesum]**