Session 16 Assignment

Q. What is Thread and Process? What is the difference between Process and Thread?
Answer: Thread is the smallest unit of execution. A process is the execution of a program. There can be many threads in a single process. Threads share memory and other resources(network/database connections,file descriptors, address) among themselves. Each process has a separate memory and resources.

Q. Why do we need Multithreading in our project? What are the advantages and disadvantages of multithreading in C#?
Answer: Multithreading is a technique that is used to introduce concurrency in the application.

Advantages:

1. Better Utilization of CPU cores.
2. Efficient resource utilization. For example: A thread can wait for user response on an I/O device while another thread can use the CPU for computation.
3. Better responsiveness. We can leave a thread to stall at a time consuming task rather than an entire process.

Disadvantages:

1. Can cause deadlocks.
2. Can cause race conditions.
3. Overhead of managing critical regions.

Q. How can we create a Thread in C#?
Answer:

```
using System.Threading;
Thread t1 = new Thread(ThreadFunction);
//start thread
t1.start();

//main thread code

//wait for t1 to finish
t1.join();
```

Q. Why does a delegate need to be passed as a parameter to the Thread class constructor?
Answer: The delegate represents a reference to a method. When you pass a delegate as a parameter to the Thread constructor, you are telling the thread which method it should execute when it starts running.

Q. How to pass parameter in Thread?
Answer: We can use a ParameterizedThreadStart delegate to create a new thread and pass data to it.
Example:

```
// Define a method to be executed by the new thread.
public static void ThreadProc(object obj)
{
  Console.WriteLine("The passed data is: " + obj.toString());
}

// Create a new thread and start it, passing a string as data.
Thread t = new Thread(ThreadProc, "Hello from the new thread!");
t.Start();
```

Q. Why do we need a ParameterizedThreadStart delegate?
Answer: ParameterizedThreadStart delegate is used when you want to pass parameters to a method that will be executed on a new thread created by the Thread class. The parameter type is always “object” as object type can be used to represent all data types.

Q. When to use ParameterizedThreadStart over ThreadStart delegate?
Answer: We can use ParameterizedThreadStart over ThreadStart delegate when we need to pass a parameter to the function that will be executed on a new thread. We can safely pass the parameter as object type to the function.

Q. How to pass data to the thread function in a type-safe manner?
Answer: We can safely pass the parameter as object type to the function that will be executed on a new thread using the ParameterizedThreadStart delegate.

Q. How to retrieve the data from a thread function?
Answer: We can retrieve the data from a thread function in following ways:

1. **Use of shared variables:** We need to be careful and use locks while reading and writing shared variables.
   Example:

   ```
   object resultLock = new object();
   string result;

   Thread thread = new Thread(() =>
   {
      // Perform some work
      string threadResult = "Result from the thread";

      // Store the result in the shared variable
      lock (resultLock)
   ```

```
    {
        result = threadResult;
    }
});

thread.Start();
thread.Join(); // Wait for the thread to finish

// Retrieve the result in the main program
lock (resultLock)
{
    Console.WriteLine("Result from the main program: " + result);
}
```

2. **Using callback functions with events:**
   Example:

```
using System;
using System.Threading;

public class Program
{
    public delegate void ThreadCallback(string result);

    public static void Main()
    {
        // Create an instance of the worker class
        Worker worker = new Worker();

        // Subscribe to the callback event
        worker.WorkCompleted += HandleWorkCompleted;

        // Start a new thread to perform some work
        Thread thread = new Thread(worker.DoWork);
        thread.Start();
    }

    public static void HandleWorkCompleted(string result)
    {
        // This method is called when the thread has completed its work
        Console.WriteLine("Result from the main program: " + result);
    }
}
```

```
public class Worker
{
    public event Program.ThreadCallback WorkCompleted;

    public void DoWork()
    {
        // Simulate some time-consuming work
        Thread.Sleep(2000);

        // Compute the result
        string threadResult = "Result from the thread";

        // Raise the event to notify the main program
        WorkCompleted?.Invoke(threadResult);
    }
}
```

3. **Using Tasks:**

Example:

```
using System;
using System.Threading.Tasks;

public class Program
{
    public static async Task Main()
    {
        // Start a new task to perform some work asynchronously
        Task<string> task = Task.Run(() => DoWork());

        // Perform other work on the main thread while the task runs asynchronously

        // Wait for the task to complete and retrieve the result
        string result = await task;

        // Print the result
        Console.WriteLine("Result from the main program: " + result);
    }

    public static string DoWork()
    {
        // Simulate some time-consuming work
        Task.Delay(2000).Wait();

        // Compute the result
```

```
        string threadResult = "Result from the thread";

        return threadResult;
    }
}
```

Q. What is the difference between Threads and Tasks?
Answer:

- **Abstraction Level:**
  - Threads are lower-level constructs provided by the operating system.
  - Tasks are a higher-level abstraction built on top of threads and the Task Parallel Library (TPL).
- **Synchronization:**
  - Threads require explicit management of synchronization primitives like locks and semaphores for thread safety.
  - Tasks can use async/await and TPL constructs, making it easier to write asynchronous and parallel code without explicit synchronization.
- **Error Handling:**
  - Threads do not provide built-in mechanisms for handling exceptions across threads.
  - Tasks offer better exception handling through the use of try/catch blocks and AggregateException for multiple exceptions in parallel operations.

Q. What is the Significance of Thread.Join and Thread.IsAlive functions in multithreading?
Answer: Thread.Join is a method used to make the calling thread (usually the main thread) wait for a specified thread to complete its execution.
Thread.IsAlive is a property that allows you to check whether a thread is currently running or has completed its execution.

Q. What happens if shared resources are not protected from concurrent access in a multithreaded program?How to protect shared resources from concurrent access?
Answer: If shared resources are not protected from concurrent access in a multithreaded program it can lead to race condition, data corruption. We can protect shared resources from concurrent access by using locks, mutexes, semaphores.

Q. What are the Interlocked functions?
Answer: The Interlocked functions are a set of thread-safe operations provided by the .NET Framework (in the System.Threading.Interlocked class) to perform atomic operations on shared variables without the need for explicit locking mechanisms like locks or mutexes. These functions are essential for writing safe and efficient multithreaded code. They ensure that operations on shared variables are completed atomically, without the risk of data corruption due to race conditions.

Q. What is a Lock?
Answer: A lock is a synchronization mechanism used to ensure that only one thread at a time can access a particular section of code or a shared resource. Locks are used to prevent race conditions and maintain data integrity in a multithreaded environment.

Q. What is the Difference between Monitor and lock in C#?
Answer: In C#Both Monitor and the lock statement are used for implementing monitor locks to control access to critical sections of code and protect shared resources in a multithreaded environment.
Lock allows us to automatically acquire and release the monitor lock.
Monitor provides more fine-grained control over monitor locks. It requires explicit calls to Monitor.Enter and Monitor.Exit to acquire and release the monitor lock.\

Q. Explain why and how a deadlock can occur in multithreading with an example?
Answer: A deadlock in multithreading occurs when two or more threads are unable to proceed with their execution because they are each waiting for a resource that is held by another thread in the deadlock. In other words, a deadlock is a situation where threads are stuck in a circular dependency, and none of them can make progress.
Example:

```
using System;
using System.Threading;

public class DeadlockExample
{
    private static readonly object Lock1 = new object();
    private static readonly object Lock2 = new object();

    public static void Main()
    {
        Thread thread1 = new Thread(ExecuteThread1);
        Thread thread2 = new Thread(ExecuteThread2);

        thread1.Start();
        thread2.Start();

        thread1.Join();
        thread2.Join();

        Console.WriteLine("Main thread completed.");
    }

    private static void ExecuteThread1()
    {
        lock (Lock1)
```

```
        {
            Console.WriteLine("Thread 1: Holding Lock1...");

            Thread.Sleep(1000); // Simulate some work

            Console.WriteLine("Thread 1: Waiting for Lock2...");
            lock (Lock2)
            {
                Console.WriteLine("Thread 1: Acquired Lock2!");
            }
        }
    }

    private static void ExecuteThread2()
    {
        lock (Lock2)
        {
            Console.WriteLine("Thread 2: Holding Lock2...");

            Thread.Sleep(1000); // Simulate some work

            Console.WriteLine("Thread 2: Waiting for Lock1...");
            lock (Lock1)
            {
                Console.WriteLine("Thread 2: Acquired Lock1!");
            }
        }
    }
}
```

Two threads, thread1 and thread2, attempt to acquire two locks, Lock1 and Lock2, in a different order. Thread1 holds Lock1 and waits for Lock2, while thread2 holds Lock2 and waits for Lock1, causing a circular wait and resulting in a program deadlock where neither thread can make progress.

Q. How to resolve a deadlock in a multithreaded program?
Answer: Ways to resolve deadlock:

- **Use a timeout:** You can set a timeout on locks. If a thread does not release a lock within the timeout period, the lock can be forcibly released.
- **Terminate one of the threads:** This is the simplest way to break a deadlock, but it can lead to data loss.

Q. What is AutoResetEvent and how it is different from ManualResetEvent?
Answer:

AutoResetEvent

- An AutoResetEvent is signaled when the WaitOne() method is called.
- Once the WaitOne() method returns, the AutoResetEvent is automatically reset.
- This means that the next thread that calls WaitOne() will be blocked until the AutoResetEvent is signaled again.

ManualResetEvent

- A ManualResetEvent is signaled when the Set() method is called.
- It remains signaled until the Reset() method is called.
- This means that any thread that calls WaitOne() will be unblocked until the ManualResetEvent is reset.

Q. What is the Semaphore?
Answer: A semaphore is a synchronization primitive used in multithreaded and concurrent programming to control access to a limited number of resources.

Q. Explain Mutex and how it is different from other Synchronization mechanisms?
Answer: Mutex ensures that only one thread at a time can access a particular resource or section of code. Mutex is used to manage external threads.

Q. What is the Race condition?
Answer: A race condition is a situation where a system's behavior depends on the sequence or timing of other events.It occurs when two or more threads access shared data or resources concurrently, and the final outcome of the program depends on the unpredictable interleaving of their execution.

Q. How can you share data between multiple threads?
Answer:

- Utilize thread-safe collections and data structures.
- Use locks, mutexes, or other synchronization primitives to protect shared data.
- Use atomic operations provided by the language or platform to perform simple and thread-safe operations on shared data.
- Implement communication between threads using message passing mechanisms, such as queues or channels.

Q. What are Concurrent Collection Classes?
Answer: Concurrent collection classes are a set of data structures provided by programming languages and libraries, specifically designed for safe and efficient use in multithreaded or concurrent programming environments. These classes allow multiple threads to access and modify collections (e.g., lists, dictionaries, queues) concurrently without the need for external synchronization mechanisms, such as locks or mutexes. Concurrent collections help simplify the development of multithreaded applications by providing built-in thread safety.

Q. What is synchronization and why it is important?

Answer: Synchronization, in the context of multithreaded and concurrent programming, refers to the coordination of the activities and access to shared resources among multiple threads to ensure that they operate correctly and predictably. It is the practice of controlling the execution of threads in such a way that concurrent access to shared data or resources does not lead to data corruption, race conditions, or unpredictable program behavior.

Q. Explain the four necessary conditions for Deadlock?
Answer: Four conditions:

1. **Mutual Exclusion**
2. **Hold and Wait**
3. **No Preemption**
4. **Circular Wait**

Q. What is LiveLock?
Answer: Livelock is a situation in multithreaded or concurrent programming where two or more threads become stuck in a repeated cycle of responding to each other's actions without making any meaningful progress.