NT106.P22.ANTT - LẬP TRÌNH MẠNG CĂN BẢN

BÁO CÁO BÀI TẬP TUẦN 04

***

Thông tin cá nhân:

Họ và tên	MSSV	Email
Nguyễn Đình Khang	23520694	23520694@gm.uit.edu.vn

BÀI LÀM

I. BÀI TẬP 01

Câu hỏi: *Multithreading có 2 luồng thread1, thread2 được kích hoạt từ mainthread. Làm thế nào để đảm bảo thread1 luôn start trước thread2.*

Để đảm bảo thread1 luôn khởi động trước thread2 trong một chương trình multithreading, chúng ta cần kiểm soát thứ tự thực thi của các luồng một cách rõ ràng. Trong trường hợp của đề bài, chúng ta có thể sử dụng cơ chế đồng bộ hóa (synchronization) như semaphore, mutex, hoặc đơn giản là sử dụng phương thức start() và join() một cách chiến lược.

Cách 1: Sử dụng start() tuần tự

Cách đơn giản, chúng ta có thể gọi thread1.start() trước thread2.start() trong main thread. Tuy nhiên, điều này chỉ đảm bảo thread1 được kích hoạt trước, chứ không đảm bảo rằng nó sẽ chạy xong trước khi thread2 bắt đầu.

```
using System;
using System.Threading;

class Program
{
    static void Main(string[] args)
    {
            Thread thread1 = new Thread(() => Console.WriteLine("Thread 1
running"));
```

```
            Thread thread2 = new Thread(() => Console.WriteLine("Thread 2
running"));

        thread1.Start(); // Khởi động thread1 trước
        thread2.Start(); // Khởi động thread2 sau
    }
}
```

```
Thread 1 running
Thread 2 running

=== Code Execution Successful ===
```

Cách 2: Sử dụng ManualResetEvent để đồng bộ hóa

Trong C#, chúng ta có thể dùng ManualResetEvent để đảm bảo thread2 không chạy cho đến khi thread1 bắt đầu.

```
using System;
using System.Threading;

class Program
{
    static ManualResetEvent mre = new ManualResetEvent(false);

    static void Main(string[] args)
    {
        Thread thread1 = new Thread(() =>
        {
            Console.WriteLine("Thread 1 running");
            mre.Set(); // Báo hiệu rằng thread1 đã bắt đầu
        });

        Thread thread2 = new Thread(() =>
        {
            mre.WaitOne(); // Đợi thread1 bắt đầu
            Console.WriteLine("Thread 2 running");
        });

        thread1.Start(); // Khởi động thread1
        thread2.Start(); // Khởi động thread2, nhưng nó sẽ đợi mre
```

```
    }
}
```

Trong đó:

- **ManualResetEvent**: Là một công cụ đồng bộ hóa cho phép một luồng đợi tín hiệu từ luồng khác.
- mre.Set(): Được gọi trong thread1 để báo hiệu rằng nó đã bắt đầu.
- mre.WaitOne(): Khiến thread2 đợi cho đến khi nhận được tín hiệu từ thread1.

```
Thread 1 running
Thread 2 running

=== Code Execution Successful ===
```

Cách 3: Sử dụng Task

Trong C# hiện đại, sử dụng Task (Task Parallel Library - TPL) thường được mọi người khuyến nghị thay vì thread trực tiếp, vì nó cung cấp API cấp cao hơn và dễ trong việc quản lý hơn.

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

class Program
{
    static async Task Main(string[] args)
    {
        Task task1 = Task.Run(() =>
        {
            Console.WriteLine("Task 1 running");
        });

        await task1; // Đợi task1 hoàn thành trước khi chạy task2

        Task task2 = Task.Run(() =>
        {
            Console.WriteLine("Task 2 running");
        });

        await task2; // Đợi task2 hoàn thành (tùy chọn)
```

```
    }
}
```

Lưu ý: Nếu người dùng chỉ cần task1 khởi động trước mà không cần đợi nó hoàn thành xong, chúng ta có thể bỏ await task1 và chỉ gọi task1.Start() trước task2.Start(). Nhưng với Task.Run(), thứ tự khởi động vẫn phụ thuộc vào scheduler, nên cần thêm cơ chế đồng bộ nếu muốn đảm bảo tuyệt đối.

```
Task 1 running
Task 2 running

=== Code Execution Successful ===
```

Cách 4: Kết hợp Task với ManualResetEvent

Nếu người dùng muốn dùng Task nhưng vẫn cần kiểm soát chặt chẽ về việc thread1 sẽ phải khởi động trước, chúng ta có thể kết hợp với việc sử dụng ManualResetEvent:

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

class Program
{
    static ManualResetEvent mre = new ManualResetEvent(false);

    static void Main(string[] args)
    {
        Task task1 = Task.Run(() =>
        {
            Console.WriteLine("Task 1 running");
            mre.Set(); // Báo hiệu task1 đã bắt đầu
        });

        Task task2 = Task.Run(() =>
        {
            mre.WaitOne(); // Đợi task1 bắt đầu
            Console.WriteLine("Task 2 running");
        });
```

```
        Task.WaitAll(task1, task2); // Đợi cả hai task hoàn thành
    }
}
```

```
Task 1 running
Task 2 running

=== Code Execution Successful ===
```

- Ví dụ minh họa bằng mã nguồn: Sử dụng ManualResetEvent để đảm bảo thread1 khởi động trước thread2.

```
using System;
using System.Threading;

class Program
{
    // Khai báo ManualResetEvent với trạng thái ban đầu là "đóng" (false)
    static ManualResetEvent mre = new ManualResetEvent(false);

    static void Main(string[] args)
    {
        // Tạo thread1
        Thread thread1 = new Thread(() =>
        {
            Console.WriteLine("Thread 1 da khoi dong");
            Thread.Sleep(1000); // Giả lập công việc mất 1 giây
            Console.WriteLine("Thread 1 hoan thanh cong viec");
            mre.Set(); // Gửi tín hiệu rằng thread1 đã bắt đầu
        });

        // Tạo thread2
        Thread thread2 = new Thread(() =>
        {
            Console.WriteLine("Thread 2 dang doi Thread 1 khoi dong...");
            mre.WaitOne(); // Đợi tín hiệu từ thread1
            Console.WriteLine("Thread 2 bat dau chay");
            Thread.Sleep(500); // Giả lập công việc mất 0.5 giây
            Console.WriteLine("Thread 2 hoan thanh cong viec");
        });
```

```
        // Khởi động các thread
        thread1.Start();
        thread2.Start();

        // Đợi cả hai thread hoàn thành trước khi kết thúc chương trình
        thread1.Join();
        thread2.Join();

        Console.WriteLine("Chuong trinh ket thuc");
    }
}
```

Giải thích ví dụ:

- ***ManualResetEvent mre:*** Được khởi tạo với trạng thái ban đầu là false, nghĩa là "đóng".
- ***thread1:*** Khi khởi động, in thông báo, thực hiện công việc giả lập (ngủ 1 giây), sau đó gọi mre.Set() để chuyển trạng thái của mre thành "mở".
- ***thread2:*** Gọi mre.WaitOne() để đợi tín hiệu từ thread1. Chỉ khi thread1 gọi Set(), thread2 mới tiếp tục chạy.
- ***Kết quả:*** Đầu ra sẽ luôn đảm bảo rằng thông báo "Thread 1 đã khởi động" xuất hiện trước khi thread2 bắt đầu thực thi, bất kể scheduler hoạt động như thế nào.

```
Thread 1 da khoi dong
Thread 2 dang doi Thread 1 khoi dong...
Thread 1 hoan thanh cong viec
Thread 2 bat dau chay
Thread 2 hoan thanh cong viec
Chuong trinh ket thuc

=== Code Execution Successful ===
```

II. BÀI TẬP 02

Câu hỏi: *Multi threading, race condition -> Giải quyết dùng lock->lock như thế nào thì bị deadlock. Nên làm như thế nào khi dùng lock để tránh deadlock.*

Multithreading và Race Condition

- Trong lập trình multi-threading, nhiều luồng (threads) có thể truy cập và thay đổi cùng một tài nguyên (ví dụ: biến, dữ liệu) đồng thời.
- Race condition xảy ra khi kết quả của chương trình phụ thuộc vào thứ tự thực thi ngẫu nhiên của các luồng, dẫn đến lỗi hoặc kết quả không mong muốn.
- Ví dụ: Hai luồng cùng tăng giá trị của biến count từ 0 lên 1. Nếu không kiểm soát, có thể cả hai luồng đọc count = 0, tăng lên 1 và ghi lại, dẫn đến count = 1 thay vì count = 2.
- Đây là ví dụ về race condition khi nhiều thread cùng tăng một biến không có lock:

```
using System;
using System.Threading;

class Program
{
    static int count = 0;

    static void Increment()
    {
        for (int i = 0; i < 100000; i++)
        {
            count++; // Race condition xảy ra ở đây
        }
    }

    static void Main()
    {
        Thread t1 = new Thread(Increment);
        Thread t2 = new Thread(Increment);

        t1.Start();
        t2.Start();

        t1.Join();
        t2.Join();
```

```
        Console.WriteLine($"Giá trị count: {count}"); // Kết quả không phải
luôn là 200000
    }
}
```

Vấn đề: count++ không phải là thao tác nguyên tử (atomic), dẫn đến race condition. Kết quả thực tế thường nhỏ hơn 200000.

```
Gia tri count: 103023

=== Code Execution Successful ===
```

Giải quyết Race Condition bằng Lock

- Để tranh race condition, ta sử dụng lock (cơ chế khóa) để đảm bảo chỉ một luồng được truy cập tài nguyên tại một thời điểm. Các luồng khác phải chờ đến khi lock được giải phóng.
- Đây là ví dụ sử dụng lock để bảo vệ đoạn code quan trọng:

```
using System;
using System.Threading;

class Program
{
    static int count = 0;
    static readonly object lockObject = new object();

    static void Increment()
    {
        for (int i = 0; i < 100000; i++)
        {
            lock (lockObject) // Bảo vệ tài nguyên
            {
                count++;
            }
        }
    }

    static void Main()
    {
```

```
        Thread t1 = new Thread(Increment);
        Thread t2 = new Thread(Increment);

        t1.Start();
        t2.Start();

        t1.Join();
        t2.Join();

          Console.WriteLine($"Giá trị count: {count}"); // Kết quả đúng là
200000
    }
}
```

Lưu ý: Lock đảm bảo chỉ một thread truy cập vào count tại một thời điểm.

```
Gia tri count: 200000

=== Code Execution Successful ===
```

Deadlock là gì và Lock gây Deadlock như thế nào?

- Deadlock xảy ra khi hai hoặc nhiều luồng bị kẹt vô thời hạn, mỗi luồng giữ một lock và chờ lock khác mà luồng kia đang giữ. Không luồng nào nhả lock, dẫn đến chương trình bị treo.

- Ví dụ về việc hai thread gây deadlock khi sử dụng hai lock:

```
using System;
using System.Threading;

class Program
{
    static readonly object lock1 = new object();
    static readonly object lock2 = new object();

    static void Thread1()
    {
        lock (lock1)
        {
            Console.WriteLine("Thread 1: Giữ lock1");
```

```
            Thread.Sleep(100); // Giả lập công việc
            lock (lock2) // Chờ lock2
            {
                Console.WriteLine("Thread 1: Giữ lock2");
            }
        }
    }

    static void Thread2()
    {
        lock (lock2)
        {
            Console.WriteLine("Thread 2: Giữ lock2");
            Thread.Sleep(100); // Giả lập công việc
            lock (lock1) // Chờ lock1
            {
                Console.WriteLine("Thread 2: Giữ lock1");
            }
        }
    }

    static void Main()
    {
        Thread t1 = new Thread(Thread1);
        Thread t2 = new Thread(Thread2);

        t1.Start();
        t2.Start();

        t1.Join();
        t2.Join();

        Console.WriteLine("Hoàn thành");
    }
}
```

```
Thread 2: Giu lock2
Thread 1: Giu lock1
```

Kết quả: Chương trình bị treo vì:

- Thread1 giữ lock1 và chờ lock2.
- Thread2 giữ lock2 và chờ lock1.

- Đây là deadlock.

Cách dùng Lock để tránh Deadlock

Để tránh deadlock khi dùng lock, chúng ta có thể áp dụng các phương pháp sau:

a. Sắp xếp thư tự khóa (Lock Ordering):

- Quy định một thứ tự cố định để lấy các lock trong tất cả các luồng. Ví dụ, lấy luôn lock1 trước lock2.

- Ví dụ:

```
using System;
using System.Threading;

class Program
{
    static readonly object lock1 = new object();
    static readonly object lock2 = new object();

    static void Thread1()
    {
        lock (lock1)
        {
            Console.WriteLine("Thread 1: Giữ lock1");
            Thread.Sleep(100);
            lock (lock2)
            {
                Console.WriteLine("Thread 1: Giữ lock2");
            }
        }
    }

    static void Thread2()
    {
        lock (lock1) // Cùng thứ tự: lock1 trước lock2
        {
            Console.WriteLine("Thread 2: Giữ lock1");
            Thread.Sleep(100);
            lock (lock2)
            {
                Console.WriteLine("Thread 2: Giữ lock2");
            }
        }
    }
```

```
    static void Main()
    {
        Thread t1 = new Thread(Thread1);
        Thread t2 = new Thread(Thread2);

        t1.Start();
        t2.Start();

        t1.Join();
        t2.Join();

        Console.WriteLine("Hoàn thành");
    }
}
```

```
Thread 1: Giu lock1
Thread 1: Giu lock2
Thread 2: Giu lock1
Thread 2: Giu lock2
Hoan thanh

=== Code Execution Successful ===
```

Kết quả: Chương trình không còn deadlock, chương trình hoàn tất.

b. Sử dụng Timeout với Monitor

- Chương trình C# không có timeout trực tiếp cho lock, thay vào đó chúng ta có thể dùng Monitor.TryEnter để tránh tình trạng treo vô hạn:

```
using System;
using System.Threading;

class Program
{
    static readonly object lock1 = new object();
    static readonly object lock2 = new object();

    static void Thread1()
```

```
    {
        if (Monitor.TryEnter(lock1, TimeSpan.FromSeconds(1)))
        {
            try
            {
                Console.WriteLine("Thread 1: Giữ lock1");
                Thread.Sleep(100);
                if (Monitor.TryEnter(lock2, TimeSpan.FromSeconds(1)))
                {
                    try
                    {
                        Console.WriteLine("Thread 1: Giữ lock2");
                    }
                    finally
                    {
                        Monitor.Exit(lock2);
                    }
                }
                else
                {
                    Console.WriteLine("Thread 1: Không lấy được lock2");
                }
            }
            finally
            {
                Monitor.Exit(lock1);
            }
        }
        else
        {
            Console.WriteLine("Thread 1: Không lấy được lock1");
        }
    }

    static void Main()
    {
        Thread t1 = new Thread(Thread1);
        t1.Start();
        t1.Join();
    }
}
```

```
Thread 1: Giu lock1
Thread 1: Giu lock2

=== Code Execution Successful ===
```

Monitor.TryEnter: Nếu không lấy được lock trong thời gian chờ, thread có thể bỏ qua hoặc thử lại, tránh deadlock.

III. BÀI TẬP 03

Câu hỏi: *Multithreading - Main thread thì khởi tạo thêm 1 thread t1, trong t1 tạo thread2.*

- Nếu main thread bị ngắt (tắt cửa sổ window đi) thì t1 và t2 có ngắt hay không?

- Nếu t1 bị ngắt thì t2 có bị ngắt hay không?

a. Nếu main thread bị ngắt (tắt cửa sổ window đi) thì t1 và t2 có ngắt hay không?

- Điều này phụ thuộc vào cách chương trình được thiết kế và ngôn ngữ lập trình bạn đang sử dụng (ví dụ: Java, Python, C++,...). Tuy nhiên, trong hầu hết các trường hợp:

+ Nếu main thread là thread chính điều khiển toàn bộ chương trình (như trong nhiều ứng dụng GUI hoặc ứng dụng console), khi main thread bị ngắt (ví dụ: bạn đóng cửa sổ window hoặc gọi exit()), toàn bộ tiến trình (process) của chương trình thường sẽ bị terminated. Khi đó, tất cả các thread con (bao gồm t1 và t2) cũng sẽ bị ngắt theo, bất kể chúng đang làm gì.

+ Ngoại lệ:

- Nếu t1 và t2 được thiết lập là daemon thread (thread chạy nền, ví dụ trong Java hoặc Python), chúng sẽ tự động bị ngắt khi main thread kết thúc mà không cần chờ chúng hoàn thành.
- Nếu chương trình được thiết kế để giữ các thread con sống sót sau khi main thread ngắt (ví dụ: sử dụng một cơ chế quản lý thread độc lập), thì t1 và t2 có thể tiếp tục chạy. Nhưng điều này hiếm khi xảy ra trong các ứng dụng thông thường.

- Ví dụ thực tế: Trong một ứng dụng GUI (như dùng Java Swing hoặc Python Tkinter), khi bạn đóng cửa sổ chính (main thread xử lý giao diện), hệ thống thường gọi System.exit() hoặc tương tự, dẫn đến việc toàn bộ ứng dụng tắt, bao gồm cả t1 và t2.

- Kết luận: Trong trường hợp thông thường (đóng cửa sổ window), t1 và t2 sẽ bị ngắt khi main thread bị ngắt.

b. Nếu t1 bị ngắt thì t2 có bị ngắt hay không?

- Câu trả lời là không, t2 không tự động bị ngắt khi t1 bị ngắt, trừ khi có cơ chế cụ thể trong code để liên kết sự sống của t2 và t1.

- Trong mô hình thread thông thường, các thread là độc lập với nhau về mặt tồn tại. T1 tạo ra t2, nhưng sau khi t2 được khởi chạy, nó không phụ thuộc trực tiếp vào trạng thái của t1. Nếu t1 hoàn thành công việc hoặc bị ngắt (ví dụ: Thông qua Thread.interrupt() hoặc hết công việc), t2 vẫn có thể tiếp tục chạy cho đến khi nó hoàn thành hoặc bị ngắt bởi một cơ chế khác.

- Trường hợp ngoại lệ:

+ Nếu trong đoạn code, người dùng thiết kế t2 để kiểm tra trạng thái của t1 (Ví dụ: Dùng một biến cờ hoặc cơ chế đồng bộ như join()), thì t2 có thể tự ngắt khi phát hiện t1 đã dừng.

+ Nếu t1 bị ngắt do toàn bộ tiến trình (process) bị tắt (Ví dụ: Lỗi crash hoặc System.exit()), thì t2 cũng sẽ bị ngắt theo.

Ví dụ minh họa:

```
using System;
using System.Threading;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Main thread started.");

        // Tạo thread t1
        Thread t1 = new Thread(() =>
        {
            Console.WriteLine("t1 started.");

            // Trong t1, tạo thread t2
            Thread t2 = new Thread(() =>
            {
                Console.WriteLine("t2 started.");
                try
                {
                    Thread.Sleep(5000); // t2 chạy 5 giây
                    Console.WriteLine("t2 completed.");
                }
                catch (ThreadInterruptedException)
                {
```

```
                    Console.WriteLine("t2 interrupted.");
                }
            });

            t2.Start(); // Khởi động t2

            try
            {
                Thread.Sleep(2000); // t1 chạy 2 giây rồi dừng tự nhiên
                Console.WriteLine("t1 completed.");
            }
            catch (ThreadInterruptedException)
            {
                Console.WriteLine("t1 interrupted.");
            }
        });

        t1.Start(); // Khởi động t1

        // Giả lập main thread chạy một lúc rồi ngắt
        Thread.Sleep(1000); // Main thread chạy 1 giây
        Console.WriteLine("Main thread exiting...");
          // Environment.Exit(0); // Nếu uncomment dòng này, toàn bộ tiến
trình sẽ ngắt
    }
}
```

Giải thích và kết quả:

Trường hợp 1: Main thread bị ngắt

- Trong code trên, main thread chạy 1 giây rồi kết thúc tự nhiên (do hàm Main hoàn thành). Tuy nhiên, trong C#, nếu không có cơ chế giữ chương trình chạy (như Console.ReadLine()), tiến trình sẽ không kết thúc ngay lập tức nếu các thread không phải daemon vẫn đang chạy (t1 và t2 trong trường hợp này là foreground threads mặc định).

- Kết quả khi chạy đoạn code trên:

```
Main thread started.
t1 started.
t2 started.
Main thread exiting...
t1 completed.
t2 completed.

=== Code Execution Successful ===
```

⇔ Main thread kết thúc, nhưng t1 và t2 vẫn chạy cho đến khi hoàn thành (t1 sau 2 giây, t2 sau 5 giây). Điều này là do C# giữ tiến trình sống chừng nào còn foreground thread.

- ***Nếu ngắt tiến trình hoàn toàn:*** Thay vì để main thread kết thúc tự nhiên, nếu bạn gọi Environment.Exit(0) (dòng bị comment trong code), toàn bộ tiến trình sẽ bị ngắt ngay lập tức:

```
Main thread started.
t1 started.
t2 started.
Main thread exiting...

=== Code Execution Successful ===
```

⇔ t1 và t2 không kịp chạy tiếp vì tiến trình đã bị tắt. Đây là trường hợp tương tự khi bạn đóng cửa sổ ứng dụng (GUI) hoặc gọi hàm thoát chương trình.

⇔ ***Kết luận:*** Nếu main thread bị ngắt kiểu "thoát tiến trình" (như đóng window hoặc Environment.Exit), thì t1 và t2 sẽ bị ngắt theo. Nếu main thread chỉ kết thúc tự nhiên, t1 và t2 vẫn chạy tiếp nếu chúng là foreground threads.

Trường hợp 2: t1 bị ngắt thì t2 có bị ngắt không?

- Trong code trên, t1 chạy 2 giây rồi kết thúc tự nhiên (hết công việc). Trong thời gian đó, t2 được khởi động và chạy độc lập trong 5 giây.

```
Main thread started.
t1 started.
t2 started.
Main thread exiting...
t1 completed.
t2 completed.

=== Code Execution Successful ===
```

⇔ Sau khi t1 hoàn thành (sau 2 giây), t2 vẫn tiếp tục chạy thêm 3 giây nữa và kết thúc sau 5 giây. Việc t1 dừng không ảnh hưởng đến t2.

- ***Thử ngắt t1 chủ động***: Nếu bạn muốn mô phỏng việc t1 bị ngắt (interrupted), có thể sửa code như sau:

```
using System;
using System.Threading;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Main thread started.");

        Thread t1 = new Thread(() =>
        {
            Console.WriteLine("t1 started.");
            Thread t2 = new Thread(() =>
            {
                Console.WriteLine("t2 started.");
                try
                {
                    Thread.Sleep(5000);
                    Console.WriteLine("t2 completed.");
                }
                catch (ThreadInterruptedException)
                {
                    Console.WriteLine("t2 interrupted.");
                }
            });
```

```
            t2.Start();

            try
            {
                Thread.Sleep(2000);
                Console.WriteLine("t1 completed.");
            }
            catch (ThreadInterruptedException)
            {
                Console.WriteLine("t1 interrupted.");
            }
        });

        // Khởi động và ngắt t1 sau 1 giây
        t1.Start();
        Thread.Sleep(1000);
        t1.Interrupt(); // Ngắt t1

        // Giả lập main thread chạy một lúc rồi ngắt
        Thread.Sleep(1000); // Main thread chạy 1 giây
        Console.WriteLine("Main thread exiting...");
          // Environment.Exit(0); // Nếu uncomment dòng này, toàn bộ tiến
trình sẽ ngắt
    }
}
```

```
Main thread started.
t1 started.
t2 started.
t1 interrupted.
Main thread exiting...
t2 completed.

=== Code Execution Successful ===
```

⇔ t1 bị ngắt sau 1 giây (do Interrupt()), nhưng t2 vẫn chạy tiếp và hoàn thành sau 5 giây. Điều này cho thấy t2 không bị ngắt khi t1 bị ngắt.

⇔ ***Kết luận:*** t2 không bị ngắt khi t1 bị ngắt, vì chúng là các thread độc lập trong C#.