



CHƯƠNG 3 LẬP TRÌNH VỚI LUỒNG (THREAD)

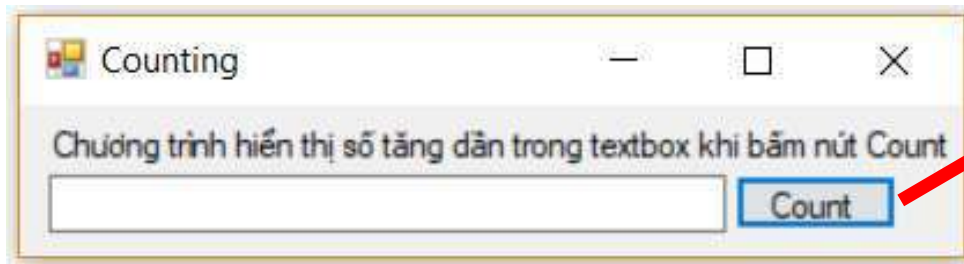
Thời gian: 3 tiết

NỘI DUNG

- Giới thiệu
 - Khai báo và khởi tạo luồng
 - Lập trình không đồng bộ
-

GIỚI THIỆU

- Ví dụ: chương trình hiển thị số vòng lặp thực hiện



```
private void btnCount_Click(object sender, EventArgs e)
{
    count1();
    count2();
}
```

Đoạn mã nào sẽ thực hiện đúng yêu cầu ?

```
private void count1()
{
    for (int i = 0; i < 2000; i++)
        txtnumber.Text = i.ToString();
}
```

1

```
private void updateNumber(string text)
{
    if (InvokeRequired)
    {
        Invoke(new Action<string>(updateNumber), text);
    }
    else { txtnumber.Text = text; }
}

private void goCount()
{
    for (int i = 0; i < 2000; i++)
        updateNumber(i.ToString());
}

private void count2()
{
    Thread t = new Thread(goCount);
    t.Start();
}
```

2

GIỚI THIỆU (2)

- Đa nhiệm (Multitasking): Là khả năng hệ điều hành thực hiện đồng thời nhiều công việc tại một thời điểm
- Tiến trình (Process):
 - Bộ nhớ và tài nguyên vật lý riêng biệt cấp phát riêng cho ứng dụng khi khởi chạy được gọi là một tiến trình
 - Các tài nguyên và bộ nhớ của một tiến trình thì chỉ tiến trình đó được phép truy cập

GIỚI THIỆU (2)

- Luồng (Thread):
 - Trong hệ thống, một tiến trình có thể có một hoặc nhiều chuỗi thực hiện tách biệt nhau và có thể chạy đồng thời
 - Mỗi chuỗi thực hiện này được gọi là một luồng (Thread)
 - Trong một ứng dụng, Thread khởi tạo đầu tiên gọi là Thread sơ cấp hay Thread chính

GIỚI THIỆU (3)

- Chương trình một luồng
 - Chỉ một luồng chạy trong môi trường cô lập của tiến trình
 - ⇒ luồng có quyền truy cập độc quyền vào tiến trình

GIỚI THIỆU (4)

- Chương trình đa luồng

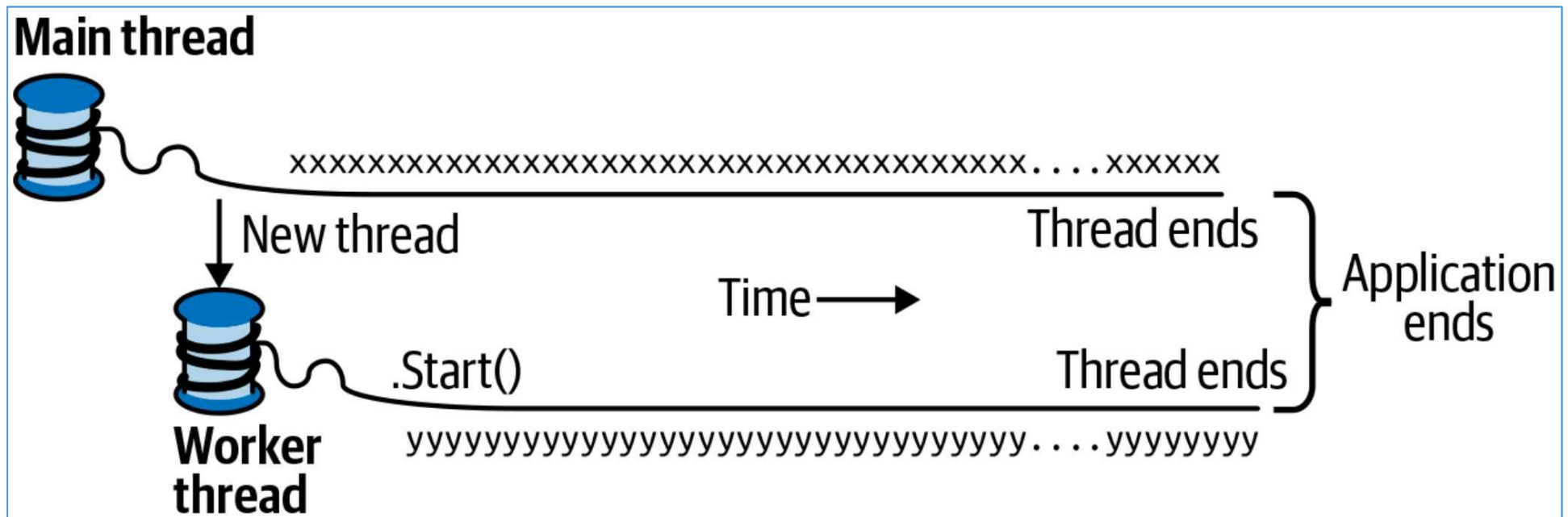
- Nhiều luồng chạy trong một tiến trình duy nhất
- Chia sẻ cùng một môi trường thực thi (bộ nhớ)

⇒ đa luồng lại hữu ích ⇒ dữ liệu này được gọi là trạng thái chia sẻ

- Một luồng có thể lấy dữ liệu nền
- Đồng thời một luồng khác hiển thị dữ liệu khi nó đến

GIỚI THIỆU (5)

- Ví dụ: Một chương trình thực hiện công việc sau:
 - Luồng chính liên tục in ký tự x
 - Luồng chính tạo ra một luồng mới, trên đó nó chạy một phương thức in nhiều lần ký tự y



GIỚI THIỆU (6)

```
class ThreadTest
{
    static void Main()
    {
        Thread t = new Thread(WriteY); // Kick off a new thread
        t.Start(); // running WriteY()
        // Simultaneously, do something on the main thread.
        for (int i = 0; i < 1000; i++) Console.Write("x");
    }
    static void WriteY()
    {
        for (int i = 0; i < 1000; i++) Console.Write("y");
    }
}
```

// Typical Output:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...
```

GIỚI THIỆU

- Threading (luồng) là một khái niệm quan trọng trong phát triển phần mềm, hỗ trợ thực hiện nhiều tác vụ tại cùng một thời điểm.
 - Hầu hết các tác vụ đều có thời gian chết (downtime), với threading, ta có thể cho bộ xử lý tiếp tục thực vớithreading, ta có thể cho bộ xử lý tiếp tục thực hiện công việc trong suốt thời gian đó.
 - Lớp Thread được dùng để tạo và thực thi các tiến trình
-

KHAI BÁO VÀ KHỞI TẠO

- Các thuộc tính và phương thức của luồng
- Các thao tác trên Luồng

KHAI BÁO VÀ KHỞI TẠO

- Các thuộc tính và phương thức
 - Các thuộc tính
 - IsAlive
 - IsBackground: thuộc tính quan trọng
 - Nếu gán giá trị true, khi chương trình chính kết thúc, luồng sẽ tự động kết thúc
 - Nếu gán giá trị false, khi chương trình chính kết thúc, luồng vẫn tiếp tục chạy
 - IsThreadPoolThread
 - ManagedThread
 - Name
 - Priority
 - ThreadState
-

KHAI BÁO VÀ KHỞI TẠO

- Các thuộc tính và phương thức
 - Các thuộc tính tĩnh (static properties)
 - CurrentContext
 - CurrentPrincipalp
 - CurrentThread
-

KHAI BÁO VÀ KHỞI TẠO

- Các thuộc tính và phương thức
 - Các phương thức
 - **Abort**
 - **Interrupt**
 - **Join**
 - **Resume**
 - **Start**
 - **Suspend**
-

KHAI BÁO VÀ KHỞI TẠO

- Các thuộc tính và phương thức
 - Các phương thức tĩnh (static methods)
 - BeginCriticalRegion
 - EndCriticalRegion
 - GetDomain
 - GetDomainID
 - ResetAbort
 - Sleep
 - SpinWait
 - VolatileRead
 - VolatileWrite
-

KHAI BÁO VÀ KHỞI TẠO

- Các thao tác trên luồng
 - Khởi tạo
 - Thực thi
 - Ghép nối
 - Tạm dừng và hủy bỏ vĩnh viễn
-

KHAI BÁO VÀ KHỞI TẠO

- Khởi tạo
 - Trong .NET Framework, namespace `System.Threading` chứa các kiểu được dùng để tạo và quản lý đa luồng trong ứng dụng
 - Khởi tạo:
 - Tạo phương thức không tham số, không kiểu dữ liệu trả về (lambda)
 - Tạo ủy nhiệm hàm `ThreadStart` với phương thức vừa tạo (delegate function)
 - Tạo `Thread` mới với ủy nhiệm hàm `ThreadStart` vừa tạo
-

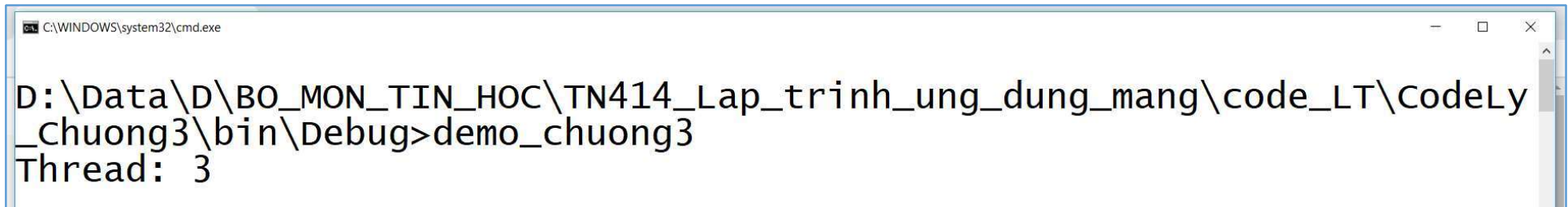
KHAI BÁO VÀ KHỞI TẠO - THỰC THI

- Luồng được khởi tạo sẽ không tự thực thi
- Gọi phương thức Start để thực thi

KHAI BÁO VÀ KHỞI TẠO - DEMO

```
class Program
{
    //Tạo phương thức không tham số không trả về dữ liệu
    static void SimpleWork()
    {
        Console.WriteLine("Thread: {0}",
            Thread.CurrentThread.ManagedThreadId);
    }
    static void Main(string[] args)
    {
        //Tạo ủy nhiệm ThreadStart
        ThreadStart op = new ThreadStart(SimpleWork);
        //Tạo Thread mới
        Thread myThread = new Thread(op);
        //Gọi phương thức Start thực thi tiến trình mới.
        myThread.Start();
    }
}
```

KHAI BÁO VÀ KHỞI TẠO - DEMO



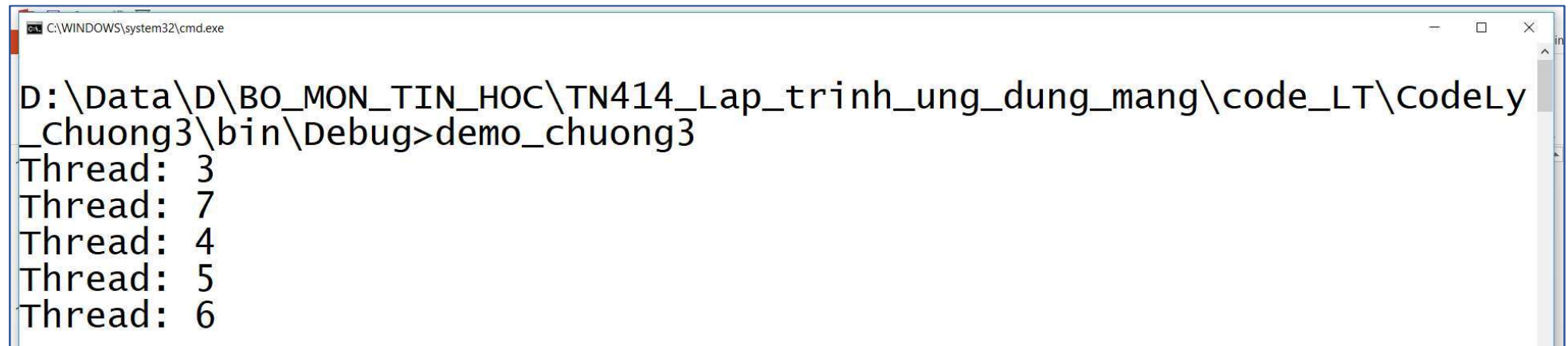
```
C:\WINDOWS\system32\cmd.exe  
D:\Data\D\BO_MON_TIN_HOC\TN414_Lap_trinh_ung_dung_mang\code_LT\CodeLy  
_Chuong3\bin\Debug>demo_chuong3  
Thread: 3
```

- Điểm mạnh của Thread là hỗ trợ xử lý đa luồng tại cùng 1 thời điểm
 - Có thể sửa lại đoạn code trên để hỗ trợ xử lý đa luồng
-

KHAI BÁO VÀ KHỞI TẠO - DEMO

```
class Program{
    //Tạo phương thức không tham số không trả về dữ liệu
    static void SimpleWork()
    {
        Console.WriteLine("Thread: {0}",
            Thread.CurrentThread.ManagedThreadId);
    }
    static void Main(string[] args)
    {
        //Tạo ủy nhiệm ThreadStart
        ThreadStart op = new ThreadStart(SimpleWork);
        //Tạo Thread mới
        for (int i = 0; i < 5; i++)
        {
            Thread myThread = new Thread(op);
            //Thực thi tiến trình mới.
            myThread.Start();
        }
    }
}
```

KHAI BÁO VÀ KHỞI TẠO - DEMO



```
C:\WINDOWS\system32\cmd.exe

D:\Data\D\BO_MON_TIN_HOC\TN414_Lap_trinh_ung_dung_mang\code_LT\CodeLy
_Chuong3\bin\Debug>demo_chuong3

Thread: 3
Thread: 7
Thread: 4
Thread: 5
Thread: 6
```

The image shows a Windows command prompt window with the title bar "C:\WINDOWS\system32\cmd.exe". The command prompt shows the current directory as "D:\Data\D\BO_MON_TIN_HOC\TN414_Lap_trinh_ung_dung_mang\code_LT\CodeLy_Chuong3\bin\Debug" and the command "demo_chuong3" being executed. The output of the command is five lines of text: "Thread: 3", "Thread: 7", "Thread: 4", "Thread: 5", and "Thread: 6".

KHAI BÁO VÀ KHỞI TẠO - JOIN

- Khi khối lượng công việc và thời gian xử lý của một luồng tăng, yêu cầu luồng chính (main thread) chờ cho đến khi xử lý của luồng được hoàn tất thông qua sử dụng phương thức Thread.Join()

```
//Tạo ủy nhiệm ThreadStart
ThreadStart op = new ThreadStart(SimpleWork);
//Tạo Thread mới
Thread[] myThreads = new Thread[5];
for (int i = 0; i < 5; i++)
    {myThreads[i] = new Thread(op);
    myThreads[i].Start();
    }
foreach (Thread t in myThreads)
    {t.Join();}
```

KHAI BÁO VÀ KHỞI TẠO-PRIORITY

- Hỗ trợ gán/ lấy độ ưu tiên của luồng thông qua enum `ThreadingPriority`.
 - Các giá trị của `ThreadingPriority`
 - `Highest`
 - `AboveNormal`
 - `Normal`
 - `BelowNormal`
 - `Lowest`
 - Thường sử dụng giá trị default là `Normal`
 - Cân nhắc khi thay đổi độ ưu tiên của luồng
-

TẠM DỪNG VÀ LOẠI BỎ

- .NET framework tự động dừng và giải phóng luồng khi xử lý hoàn tất.
- Dùng phương thức `Thread.Abort()` để dừng luồng khi có nhu cầu

```
Thread myThread = new Thread(new ThreadStart(AbortThisThread));  
myThread.Start();  
myThread.Abort();  
static void AbortThisThread()  
{  
    SomeClass.IsValid = true;  
    SomeClass.IsComplete = true;  
    SomeClass.WriteToConsole();  
}
```



Có an
toàn
không ?

TẠM DỪNG VÀ LOẠI BỎ

- Dừng luồng một cách an toàn: tạo lập critical region với `BeginCriticalRegion` và `EndCriticalRegion`.

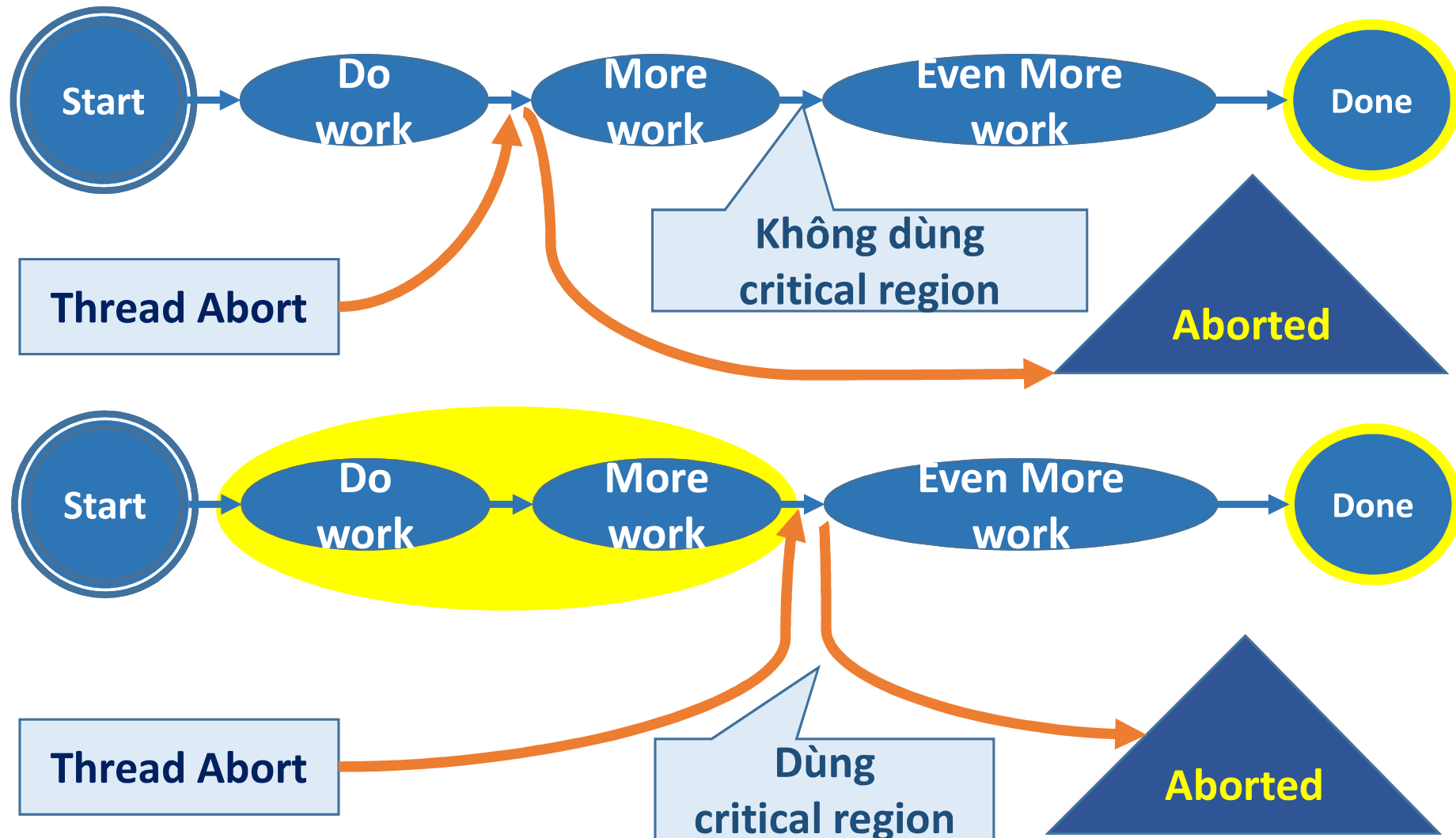
```
static void AbortThisThread()  
{  
    Thread.BeginCriticalRegion();  
    SomeClass.IsValid = true;  
    SomeClass.IsComplete = true;  
    Thread.EndCriticalRegion();  
    SomeClass.WriteToConsole();  
}
```



**Critical
Region**

TẠM DỪNG VÀ LOẠI BỎ

- So sánh giữa có sử dụng và không sử dụng



LẬP TRÌNH BẤT ĐỒNG BỘ

- Hỗ trợ từng phần của chương trình được thực thi trên nhiều luồng riêng biệt, tương tự mô hình Asynchronous Programming Model (APM).
 - .NET framework hỗ trợ APM qua nhiều lớp có cung cấp phương thức BeginXXX và EndXXX
 - Ví dụ: lớp FileStream có phương thức Read đọc dữ liệu từ stream, nó cũng cung cấp phương thức BeginRead và EndRead hỗ trợ mô hình APM
-

LẬP TRÌNH BẤT ĐỒNG BỘ-DEMO

```
byte[] buffer = new byte[100];
FileStream strm = new FileStream("c:/test.txt",
    FileMode.Open, FileAccess.Read, FileShare.Read,
    1024, FileOptions.Asynchronous);
// gọi xử lý không đồng bộ
IAsyncResult result = strm.BeginRead(buffer, 0,
    buffer.Length, null, null);
// tiến hành xử lý khác trong khi chờ
// EndRead sẽ bị khóa cho đến khi xử lý không đồng
bộ hoàn thành
int numBytes = strm.EndRead(result);
strm.Close();
Console.WriteLine("Read {0}", numBytes);
Console.WriteLine(BitConverter.ToString(buffer));
```

LẬP TRÌNH BẤT ĐỒNG BỘ

- Cần có cách thực thi tác vụ bất đồng bộ và biết khi nào/ nơi nào sẽ gọi phương thức EndXXX.
 - Rendezvous Model: có 3 cách mà APM sử dụng để xử lý khi kết thúc lời gọi phương thức không đồng bộ
 - Wait-Until-Done
 - Pooling
 - Callback
-

LẬP TRÌNH BẤT ĐỒNG BỘ

- Wait-Until-Done

- Cho phép bắt đầu lời gọi phương thức không đồng bộ và thực thi các tác vụ khác.
 - Lời gọi kết thúc phương thức không đồng bộ sẽ bị lock khi phương thức không đồng bộ hoàn toàn xử lý
-

LẬP TRÌNH BẤT ĐỒNG BỘ

- Pooling

- Tương tự Wait-Until-Done, nhưng có thăm dò `IAsyncResult` để biết xử lý đã hoàn tất chưa

```
// gọi xử lý không đồng bộ
IAsyncResult result = strm.BeginRead(buffer, 0,
buffer.Length, null, null);
// thăm dò xử lý hoàn tất chưa ?
while(!result.IsCompleted){
// xử lý khác khi chưa kết thúc lời gọi phương thức
không đồng bộ
Thread.Sleep(100);
}
// EndRead sẽ bị khóa cho đến khi không đồng bộ hoàn
thành
int numBytes = strm.EndRead(result);
```


LẬP TRÌNH BẤT ĐỒNG BỘ

- Callback

- Yêu cầu ta chỉ định phương thức callback và bất kỳ trạng thái nào dùng trong phương thức callback để kết thúc lời gọi phương thức không đồng bộ

```
static void TestCallbackAPM()  
{  
    string filename = "...";  
    FileStream strm = new ...;  
    // thực hiện lời gọi không đồng bộ  
    IAsyncResult result =  
        strm.BeginRead(buffer, 0,  
            buffer.Length, new  
            AsyncCallback(CompleteRead), strm);  
}
```

```
static void CompleteRead(IAsyncResult r)  
{  
    Console.WriteLine("Read Completed");  
    FileStream strm = (FileStream)  
        r.AsyncState;  
    int numBytes = strm.EndRead(r);  
    strm.Close();  
    Console.WriteLine(".....");  
}
```

LẬP TRÌNH BẤT ĐỒNG BỘ

- Ngoại lệ và APM

- Khi dùng APM, các tác vụ có thể phát sinh các ngoại lệ trong quá trình xử lý
- Các ngoại lệ thường phát sinh trong quá trình gọi phương thức EndXXX

```
int numBytes = 0;
try
{
    numBytes = strm.EndRead(result);
}
catch(IOException){
    Console.WriteLine("An IO Exception occurred");
}
```

LẬP TRÌNH BẤT ĐỒNG BỘ

- ThreadPool
 - Timer
-

LẬP TRÌNH BẤT ĐỒNG BỘ

- ThreadPool

- Trong nhiều trường hợp, luồng riêng để thực thi những xử lý không đồng bộ là không cần thiết
 - .NET hỗ trợ các built-in thread pool có thể dùng trong nhiều trường hợp mà ta sẽ có thể cần phải tạo luồng xử lý của riêng mình.
 - **Các đặc điểm**
 - ThreadPool thực thi nhanh
 - Điều khiển số luồng thực thi tại cùng 1 thời điểm
-

LẬP TRÌNH BẤT ĐỒNG BỘ

- Sử dụng ThreadPool
 - Dùng phương thức QueueUserWorkItem của ThreadPool để tạo và điều khiển các luồng

```
static void WorkWithParameter(object o)
{
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine("{ 0}:{ 1}",
            o.ToString(), Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(10);
    }
}

WaitCallback workItem = new WaitCallback(WorkWithParameter);
if(!ThreadPool.QueueUserWorkItem(workItem, "ThreadPooled"))
    Console.WriteLine("Could not queue item");
Thread.Sleep(1000);
```

LẬP TRÌNH BẤT ĐỒNG BỘ

- Giới hạn số tiến trình trong ThreadPool
 - ThreadPool cho phép chỉ định số lượng luồng tối đa và tối thiểu 2 trường hợp cần thay đổi số lượng luồng:
 - **Starvation**
 - **Startup thread speed**
 - Thay đổi chỉ ảnh hưởng đến luồng hiện hành
-

LẬP TRÌNH BẤT ĐỒNG BỘ

- Starvation

- Xảy ra khi ứng dụng có quá nhiều luồng, vượt quá giới hạn của ThreadPool.
 - Dùng phương thức `ThreadPool.SetMaxThreads` để thay đổi số lượng luồng giới hạn

```
int threads;  
int completionPorts;  
ThreadPool.GetMaxThreads(out threads, out completionPorts);  
ThreadPool.SetMaxThreads(threads+10, completionPorts+100);
```

LẬP TRÌNH BẤT ĐỒNG BỘ

- Timer

- Lớp Timer hỗ trợ thực thi một phương thức được tham chiếu bởi ủy nhiệm TimerCallback tại một/nhiều thời điểm xác định một cách không đồng bộ.
 - Phương thức được tham chiếu được thực thi như một luồng trong ThreadPool
-

LẬP TRÌNH BẤT ĐỒNG BỘ

- Sử dụng Timer

- Khai báo Timer, chỉ định một phương thức cho ủy nhiệm TimerCallback thực thi khi khởi động Timer.
- Các giá trị có thể thay đổi:
 - Thời gian chờ đến khởi động Timer.
 - Khoảng thời gian giữa các lần khởi động

```
//Tạo timer khởi động phương thức TimerTick mỗi giây, Khởi  
động ngay lập tức  
static void TimerTick(object o)  
{  
    Console.WriteLine("Tick:{0}",DateTime.Now.ToLongTimeString());  
}  
TimerCallback tc = new TimerCallback(TimerTick);  
Timer tm = new Timer(tc,"nothing",0,1000);
```



CÁC CHÚ Ý

CÁC TRƯỜNG HỢP SỬ DỤNG

- Duy trì sự sẵn sàng của giao diện người dùng
 - Các tác vụ tốn thời gian thực thi trên một luồng song song, luồng giao diện chính tiếp tục xử lý các sự kiện bàn phím và chuột
- Sử dụng hiệu quả CPU của một luồng bị chặn khác
 - Khi một luồng bị chặn trong khi thực hiện tác vụ thì các luồng khác có thể tận dụng tài nguyên của máy

CÁC TRƯỜNG HỢP SỬ DỤNG

- Lập trình song song
 - Các mã thực hiện các phép tính chuyên sâu có thể thực thi nhanh hơn trên máy tính đa lõi hoặc đa xử lý nếu khối lượng công việc được chia sẻ giữa nhiều luồng theo chiến lược “chia để trị”

CÁC TRƯỜNG HỢP SỬ DỤNG

- Thực thi suy đoán
 - Trên các máy đa lỗi, hiệu suất có thể cải thiện bằng cách dự đoán công việc có thể cần phải thực hiện và do đó thực hiện nó trước thời hạn
- Cho phép xử lý đồng thời các yêu cầu
 - Trên máy chủ, các yêu cầu của máy khách có thể gửi đồng thời và do đó cần được xử lý song song

CÁC TRƯỜNG HỢP KHÔNG NÊN DÙNG

- Do phát sinh chi phí tài nguyên và CPU trong
 - **Lập lịch**
 - **Chuyển đổi luồng (khi có nhiều luồng hoạt động hơn hơn lõi CPU)**
 - **Chi phí tạo / giảm bớt**
- Đa luồng không phải lúc nào cũng tăng tốc ứng dụng → chậm ứng dụng nếu sử dụng quá mức hoặc không phù hợp
- Ví dụ: Thao tác đọc ghi trên một đĩa cứng đầy dữ liệu chỉ có vài luồng chạy các tác vụ theo trình tự có thể nhanh hơn là có 10 luồng thực thi cùng một lúc



BIỂU THỨC LAMBDA (LAMBDA EXPRESSION)

BIỂU THỨC LAMBDA

- Biểu thức lambda
 - Biểu thức hàm ẩn danh (Anonymous)
 - Là một biểu thức khai báo giống phương thức (hàm) nhưng thiếu tên
- Các biểu thức lambda đều có thể chuyển đổi thành các ủy nhiệm hàm, do vậy nó có thể gán cho các ủy nhiệm hàm phù hợp

BIỂU THỨC LAMBDA (2)

- Cú pháp để khai báo biểu thức lambda là
 - Sử dụng toán tử \Rightarrow như sau: $(\text{các_tham_số}) \Rightarrow \text{biểu_thức};$
 - Cấu trúc các lệnh sau toán tử \Rightarrow
 $(\text{các_tham_số}) \Rightarrow \{$
 // các câu lệnh
 // Sử dụng return nếu có giá trị trả về
 }

BIỂU THỨC LAMBDA (3)

```
class Program
{
    public delegate int TinhToan(int a, int b);
    static void Main(string[] args)
    {
        // Gán biểu thức lambda cho delegate
        TinhToan tinhhtong =
            (int x, int y) => {
                return x + y;
            };

        int kq = tinhhtong(5, 1); // kq = 6;
        Console.WriteLine(kq);
    }
}
```