

# ĐA TIẾN TRÌNH

ThS. Trần Anh Dũng



Microsoft

Visual Studio.net

# Nội dung chính

đồng bộ tiến trình: đã tiến trình có kết quả như làm  
tuần tự tiến trình

1

Giới thiệu

2

Đa tiến trình trên .NET

3

Quản lý tiến trình

4

Đồng bộ hóa

5

Lập trình bất đồng bộ

# Giới thiệu

- Hệ điều hành đa nhiệm cổ điển:
  - Đơn vị cơ bản sử dụng CPU là **process**.
  - **Process** là một đoạn chương trình độc lập đã được nạp vào bộ nhớ.
  - Mỗi **process** thi hành một ứng dụng riêng, có một không gian địa chỉ và một không gian trạng thái riêng.
  - Các **process** liên lạc với nhau thông qua hệ điều hành, tập tin, mạng.

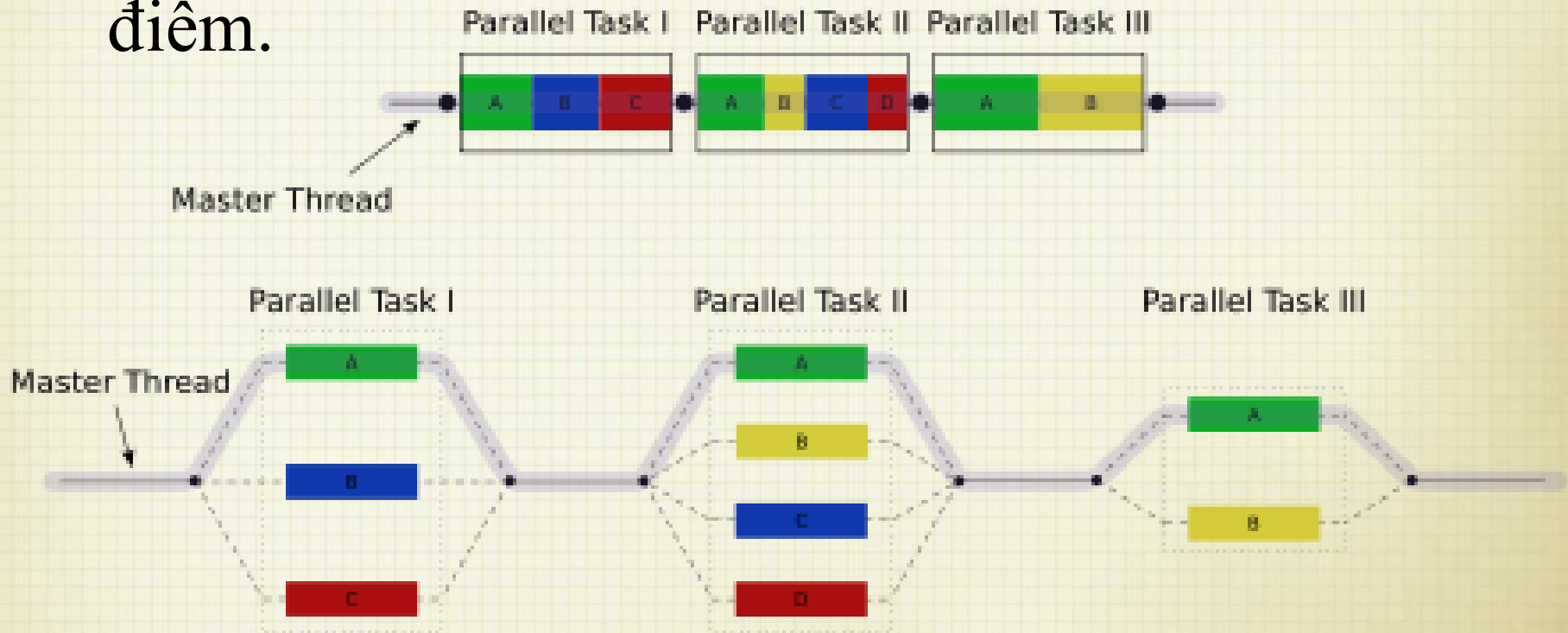


# Giới thiệu

- Hệ điều hành đa nhiệm hiện đại, hỗ trợ Thread:
  - Đơn vị cơ bản sử dụng CPU là **thread**.
  - **Thread** một đoạn các câu lệnh được thi hành.
  - Mỗi **process** có một không gian địa chỉ và nhiều **thread** điều khiển.
  - Mỗi **thread** có bộ đếm chương trình, trạng thái các thanh ghi và ngăn xếp riêng.

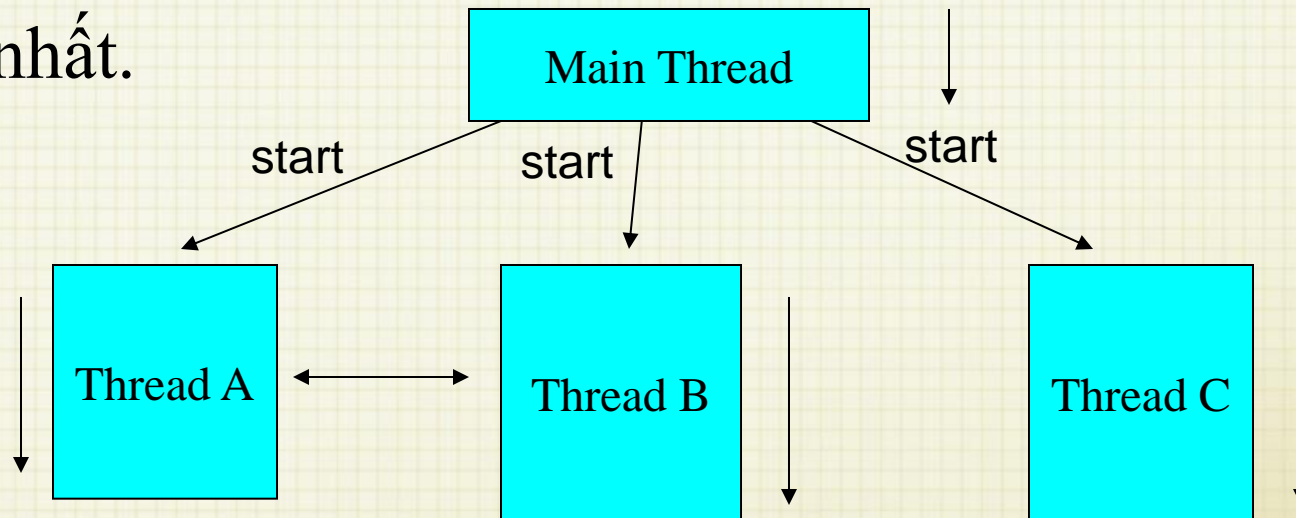
# Giới thiệu

- **Tiểu trình (thread)** thường được tạo ra khi muốn làm đồng thời 2 việc trong cùng một thời điểm.



# Đa tiểu trình

- Là khả năng làm việc với **nhiều thread**.
  - Chuyên sử dụng cho việc thực thi nhiều công việc đồng thời.
  - Giảm thời gian rồi của hệ thống đến mức thấp nhất.



Các thread có thể chuyển đổi dữ liệu với nhau

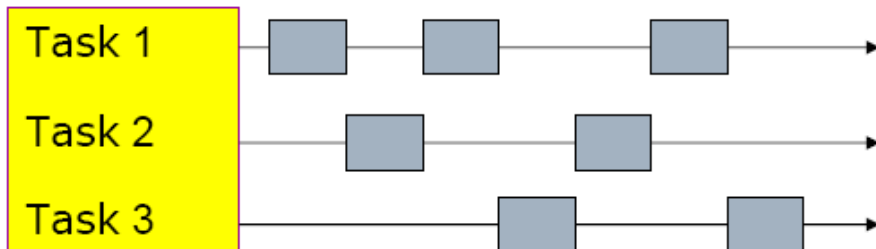


# Đa tiểu trình

- Một bộ xử lý chỉ có thể làm một việc vào một thời điểm.
- Nếu có một hệ thống đa xử lý, theo lý thuyết có thể có nhiều lệnh được thi hành đồng bộ, mỗi lệnh trên một bộ xử lý.
- Tuy nhiên ta chỉ làm việc trên một bộ xử lý.
- Do đó các công việc không thể xảy ra cùng lúc.

# Đa tiểu trình

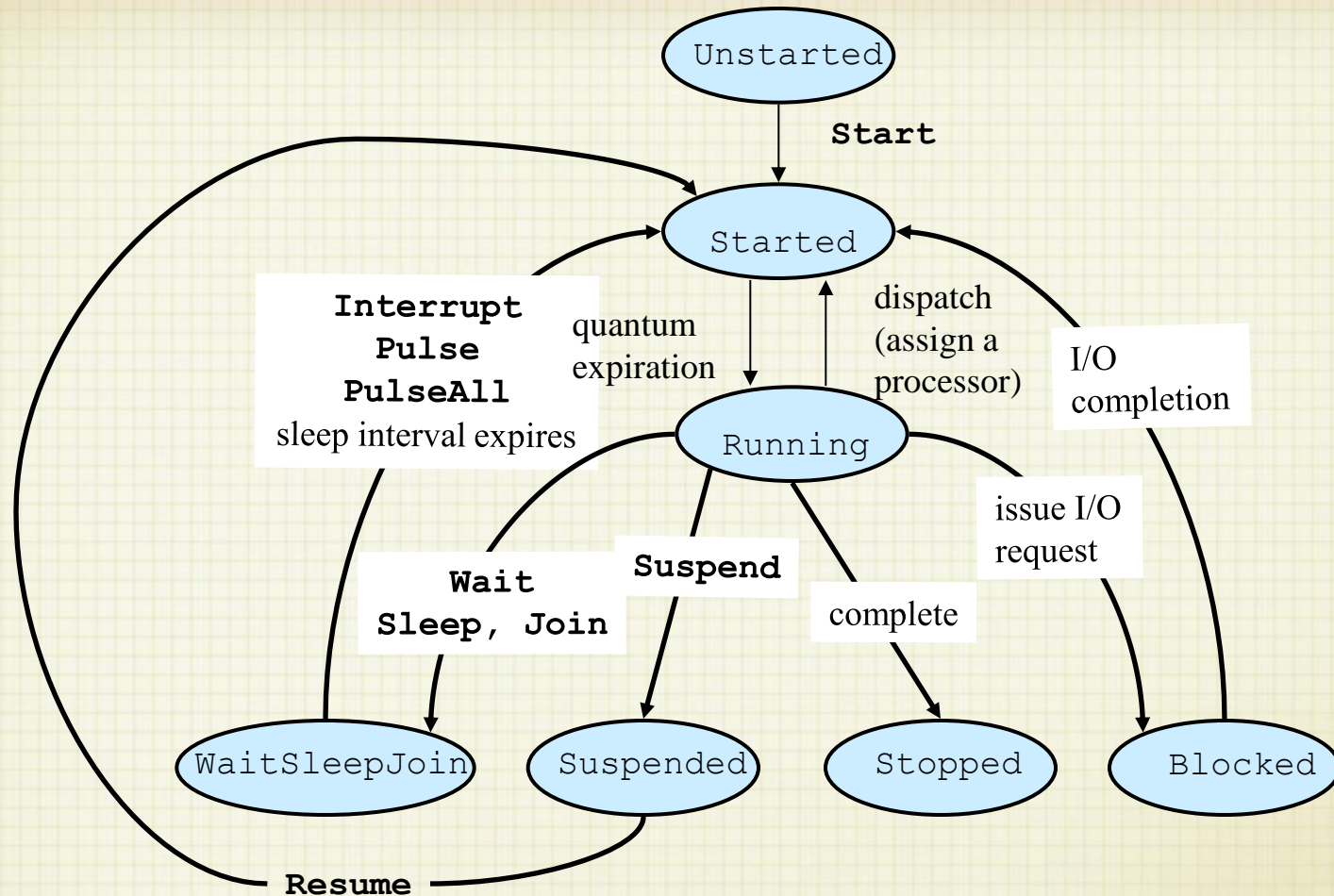
- Window lấy một luồng vào trong vài tiểu trình và cho phép luồng đó chạy một **khoảng thời gian ngắn (gọi là time slice)**. Khi thời gian này kết thúc, Window lấy quyền điều khiển lại và lấy một luồng khác và lại cấp một khoảng thời gian time slice.



Nhiều công việc thi hành trên một CPU



# Các trạng thái của thread



**Chu trình của một thread**

# Các trạng thái của thread

- Chưa bắt đầu (Unstarted):
  - Khi một thread được khởi tạo.
  - Tiếp tục cho đến khi phương thức Start của thread được gọi.
- Bắt đầu (Started):
  - Duy trì tới lúc bộ xử lý bắt đầu thực hiện nó.
- Đang thực thi (Running):
  - Thread bắt đầu có độ ưu tiên cao nhất sẽ vào trạng thái thực thi đầu tiên.
  - Bắt đầu thực thi khi bộ xử lý được gán cho thread

# Các trạng thái của thread

- **Ngừng (Stopped):**
  - Khi ủy nhiệm kết thúc
  - Chương trình gọi phương thức Abort của thread
- **Blocked:**
  - Blocked khi yêu cầu I/O
  - Unblocked khi hệ điều hành hoàn thành I/O
- **Tạm ngưng (Suspended):**
  - Khi phương thức Suspend được gọi
  - Trở về trạng thái bắt đầu (Started) khi phương thức Resume được gọi.



# Các trạng thái của thread

- WaitSleepJoin:

- Xảy ra khi:

- Thread gọi Monitor phương thức Wait vì nó gặp mã mà nó không thực hiện được.
    - Gọi phương thức Sleep để sleep trong một khoảng thời gian.
    - Hai thread được kết hợp nếu một thread không thể thực hiện cho đến khi thread kia hoàn thành.

- Các thread đợi (Waiting) hoặc ngủ (Sleeping) có thể ra khỏi trạng thái này nếu phương thức Interrupt của thread được gọi

# Đa tiểu trình trong .NET

- Hầu hết các ngôn ngữ chỉ cho phép thực hiện một câu lệnh tại một thời điểm
  - Thông thường việc thực thi các câu lệnh một cách đồng thời chỉ bằng cách dùng hệ điều hành
- Thư viện .NET Framework cho phép xử lý đồng thời bằng đa tiểu trình
  - Đa tiểu trình: thực thi các tiểu trình đồng thời
  - Tiểu trình: phần của một chương trình mà có thể thực thi

# Tạo tiểu trình

- Lớp quản lý tiểu trình: **Thread**
- Constructor của Thread nhận tham số là 1 **delegate** kiểu **ThreadStart**

**public delegate void ThreadStart( );**

- Hàm đầu vào của delegate là hàm để tiểu trình thực thi

```
Thread myThread = new Thread(new ThreadStart(myFunc));  
myThread.Start(); //Chạy tiểu trình
```



# Join tiểu trình

- Để tiểu trình A tạm dừng và chờ tiểu trình B hoàn thành thì mới tiếp tục, ta đặt hàm Join trong hàm thực thi của tiểu trình A

```
public void myFunc ()  
{  
    ...  
    thB.Join();  
    ...  
}
```

Dừng ở đây cho đến khi  
thread thB kết thúc



# Tạm dừng tiểu trình

- Tạm dừng tiểu trình trong một khoảng thời gian xác định (bộ điều phối thread của hệ điều hành sẽ không phân phối thời gian CPU cho thread này trong khoảng thời gian đó).

**Thread.Sleep(1000);**

- Có thể dùng hàm Sleep để hệ điều hành chuyển quyền điều khiển sang một tiểu trình khác

# Hủy tiểu trình

- Tiểu trình sẽ kết thúc khi hàm thực thi của nó kết thúc (Đây là cách tự nhiên nhất, tốt nhất)
- Để ép tiểu trình kết thúc ngay lập tức có thể sử dụng hàm Interrupt.
- Thread bị chấm dứt có thể bắt exception này để dọn dẹp tài nguyên

```
catch (ThreadInterruptedException) {  
    Console.WriteLine("[{0}]   Interrupted!   Cleaning  
up...", Thread.CurrentThread.Name);  
}
```



# Tiểu trình Background và Foreground

- Một tiểu trình có thể được thực thi theo hai cách: **background** hoặc **foreground**.
- Một tiểu trình background được hoàn thành khi ứng dụng được kết thúc, ngược lại tiểu trình chạy foreground thì không phải chờ đợi sự kết thúc của ứng dụng.
- Có thể thiết lập sự thực thi của tiểu trình bằng cách sử dụng thuộc tính **IsBackground**.

# Độ ưu tiên tiểu trình và lập lịch cho tiểu trình

- Tất cả tiểu trình đều có một độ ưu tiên:
  - Các độ ưu tiên là:
    - Thấp nhất (Lowest)
    - Dưới trung bình (BelowNormal)
    - Trung bình (Normal)
    - Trên trung bình (AboveNormal)
    - Cao nhất (Highest)
  - Sử dụng thuộc tính **Priority** để thay đổi độ ưu tiên của tiểu trình.

# Độ ưu tiên tiến trình và lập lịch cho tiến trình

- Timeslicing:
  - Mỗi thread được cấp một khoảng thời gian để thực thi trước khi bộ xử lý được giao cho thread khác
  - Nếu không có thì các thread sẽ thực hiện cho đến lúc hoàn thành trước khi thread khác bắt đầu thực thi.



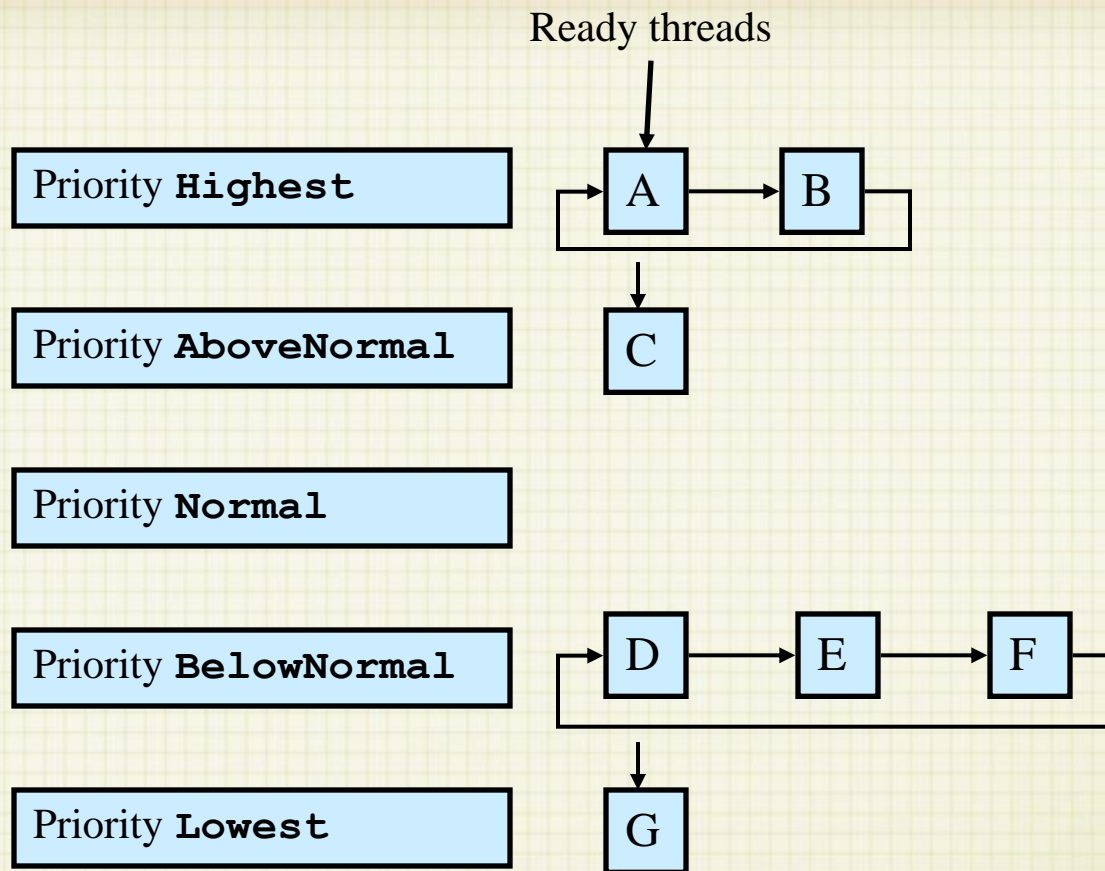
# Độ ưu tiên tiến trình và lập lịch cho tiến trình

- Lưu ý:
  - Mỗi luồng có một độ ưu tiên cơ sở. Những giá trị này liên quan đến độ ưu tiên trong tiến trình.
  - Một luồng có độ ưu tiên cao hơn đảm bảo nó sẽ chiếm quyền ưu tiên so với các luồng khác trong tiến trình.
  - Windows có khuynh hướng đặt độ ưu tiên cao cho các luồng hệ điều hành của riêng nó.

# Độ ưu tiên tiến trình và lập lịch cho tiến trình

- Bộ lập lịch tiến trình:
  - Giữ thread có độ ưu tiên cao nhất luôn thực thi tại mọi thời điểm.
    - Nếu nhiều tiến trình có cùng độ ưu tiên thì thực hiện xoay vòng.
  - Đôi khi gây ra thiếu hụt:
    - Sự trì hoãn việc thực thi của một tiến trình có độ ưu tiên thấp.

# Độ ưu tiên tiến trình và lập lịch cho tiến trình



Lập lịch độ ưu tiên



```
1 // Fig. 14.3: ThreadTester.cs
2 // Multiple threads printing at different intervals.
3
4 using System;
5 using System.Threading;
6
7 // class ThreadTester demonstrates basic threading concepts
8 class ThreadTester
9 {
10     static void Main( string[] args )
11     {
12         // Create and name each thread. Use MessagePrinter's
13         // Print method as argument to ThreadStart delegate.
14         MessagePrinter printer1 = new MessagePrinter();
15         Thread thread1 =
16             new Thread ( new ThreadStart( printer1.Print ) );
17         thread1.Name = "thread1";
18
19         MessagePrinter printer2 = new MessagePrinter();
20         Thread thread2 =
21             new Thread ( new ThreadStart( printer2.Print ) );
22         thread2.Name = "thread2";
23
24         MessagePrinter printer3 = new MessagePrinter();
25         Thread thread3 =
26             new Thread ( new ThreadStart( printer3.Print ) );
27         thread3.Name = "thread3";
28
29         Console.WriteLine( "Starting threads" );
30
31         // call each thread's Start method to place each
32         // thread in Started state
33         thread1.Start();
34         thread2.Start();
35         thread3.Start();
```

Class that creates  
3 new threads

Create MessagePrinter  
objects

Create and initialize threads

Set thread's name

Thread delegates

Start threads

```

36
37     Console.WriteLine( "Threads started\n" );
38
39 } // end method Main
40
41 } // end class ThreadTester
42
43 // Print method of this class used to control threads
44 class MessagePrinter
45 {
46     private int sleepTime;
47     private static Random random = new Random();
48
49     // constructor to initialize a MessagePrinter object
50     public MessagePrinter()
51     {
52         // pick random sleep time between 0 and 5 seconds
53         sleepTime = random.Next( 5001 );
54     }
55
56     // method Print controls thread that prints messages
57     public void Print()
58     {
59         // obtain reference to currently executing thread
60         Thread current = Thread.CurrentThread;
61
62         // put thread to sleep for sleepTime amount of time
63         Console.WriteLine(
64             current.Name + " going to sleep for " + sleepTime );
65
66         Thread.Sleep ( sleepTime );
67

```

Tell user  
threads started

Random sleep  
time for thread

Class to define  
action of threads

Thread constructor

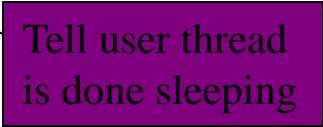
Set sleep time

Reference to  
current thread

Print name of thread  
and sleep time

Put thread to sleep

```
68         // print thread name
69         Console.WriteLine( current.Name + " done sleeping" );
70
71     } // end method Print
72
73 } // end class MessagePrinter
```



Tell user thread  
is done sleeping

Starting threads  
Threads started

```
thread1 going to sleep for 1977
thread2 going to sleep for 4513
thread3 going to sleep for 1261
thread3 done sleeping
thread1 done sleeping
thread2 done sleeping
```

Starting threads  
Threads started

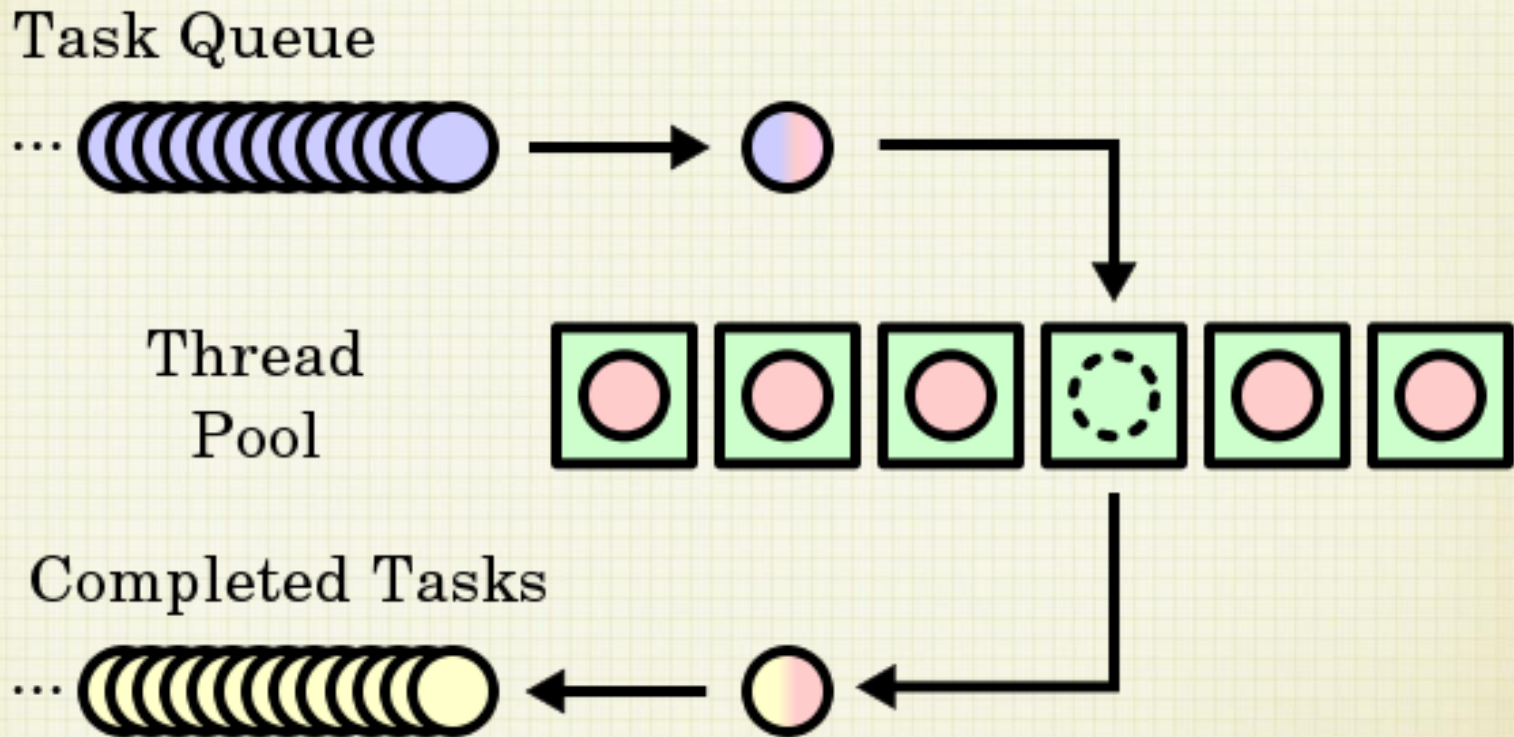
```
thread1 going to sleep for 1466
thread2 going to sleep for 4245
thread3 going to sleep for 1929
thread1 done sleeping
thread3 done sleeping
thread2 done sleeping
```



# ThreadPool

- Nếu ứng dụng sử dụng nhiều thread có thời gian sống ngắn hay duy trì một số lượng lớn các thread đồng thời thì hiệu năng có thể giảm sút bởi các chi phí cho việc tạo, vận hành và hủy các thread.
  - Trong một hệ thống hỗ trợ multithread, các thread thường ở trạng thái rỗi suốt một khoảng thời gian dài để chờ điều kiện thực thi phù hợp.
- ➔ Việc sử dụng ThreadPool sẽ cung cấp một giải pháp chung nhằm cải thiện tính quy mô và hiệu năng của các hệ thống hỗ trợ multithread.

# ThreadPool



# ThreadPool

- .NET Framework cung cấp lớp **ThreadPool**.
- Bộ thực thi quy định số tiêu trình tối đa được cấp cho thread-pool, không thể thay đổi số tối đa này bằng các tham số cấu hình hay từ bên trong mã được-quản-lý. **Giới hạn mặc định là 25 tiêu trình cho mỗi CPU trong hệ thống.** Số tiêu trình tối đa trong thread-pool không giới hạn số các công việc đang chờ trong hàng đợi.



# ThreadPool

- .NET Framework cung cấp một hiện thực đơn giản cho thread-pool có thể truy xuất thông qua các thành viên tĩnh của lớp **ThreadPool**.
- Khi một tiểu trình trong thread-pool sẵn sàng, nó nhận công việc kế tiếp từ hàng đợi và thực thi công việc này. Khi đã hoàn tất công việc, thay vì kết thúc, tiểu trình này quay về thread-pool và nhận công việc kế tiếp từ hàng đợi.

# ThreadPool

- Bộ thực thi còn sử dụng thread-pool cho nhiều mục đích bên trong, bao gồm việc thực thi phương thức một cách **bất đồng bộ** và thực thi các **sự kiện định thời**.
- Tất cả các công việc này có thể dẫn đến sự tranh chấp giữa các tiểu trình trong thread-pool, nghĩa là hàng đợi có thể trở nên rất dài → Làm kéo dài quá trình thực thi các công việc trong hàng đợi.

# ThreadPool

- Không nên sử dụng thread-pool để thực thi các tiểu trình chạy trong một thời gian dài.
- Nên tránh đặt các tiểu trình trong thread-pool vào trạng thái đợi trong một thời gian quá dài.
- Không thể điều khiển lịch trình của các tiểu trình trong thread-pool, cũng như không thể thay đổi độ ưu tiên của các công việc.



```

using System;
using System.Threading;
public class Example
{
    public static void Main()
    {
        // Queue the task.
        ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadProc));
        Console.WriteLine( "Main thread does some work, then sleeps. " );
        // If you comment out the Sleep, the main thread exits before
        // the thread pool task runs. The thread pool uses background
        // threads, which do not keep the application running. (This
        // is a simple example of a race condition.)
        Thread.Sleep(1000);
        Console.WriteLine( "Main thread exits. " );
    }
    // This thread procedure performs the task.
    static void ThreadProc(Object stateInfo)
    {
        // No state object was passed to QueueUserWorkItem, so
        // stateInfo is null.
        Console.WriteLine( "Hello from the thread pool. " );
    }
}

```

# Đồng bộ hóa (Synchronization)

- Nếu trên hệ thống nhiều CPU hoặc CPU đa nhân hay CPU hỗ trợ siêu phân luồng, các luồng sẽ thực sự hoạt động song song tại cùng 1 thời điểm. Như vậy, nếu các luồng này cùng truy xuất đến 1 biến dữ liệu hoặc 1 phương thức, điều này có thể gây ra việc sai lệch dữ liệu.

- Xét ví dụ sau:

```
public class testclass {  
    int count=0;  
    public void tang()  
    {  
        count=count+1;  
    }  
}
```

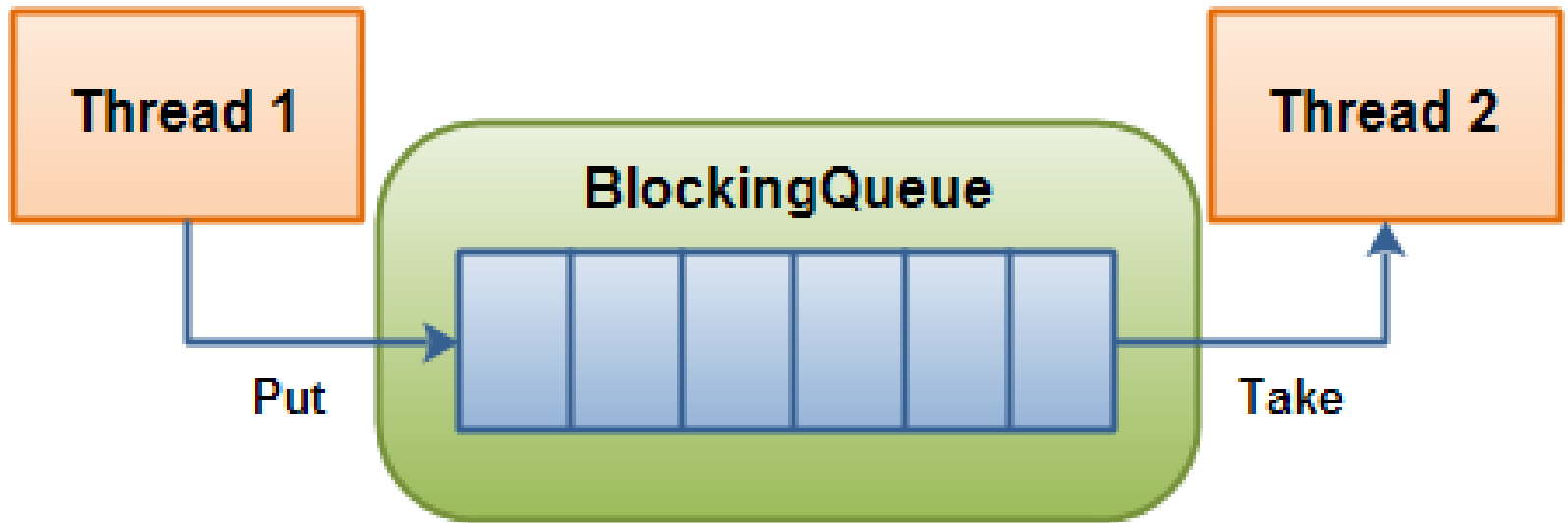
# Đồng bộ hóa (Synchronization)

- Khi cần bảo vệ một tài nguyên, trong một thời điểm chỉ cho phép một thread thay đổi hoặc sử dụng tài nguyên đó, chúng ta cần **đồng bộ hóa**.
- Đồng bộ hóa được cung cấp bởi một **khóa** trên đối tượng đó, **khóa đó sẽ ngăn cản thread thứ 2 truy cập** vào đối tượng nếu thread thứ nhất chưa trả quyền truy cập đối tượng.
- Có 4 loại đồng bộ hóa chính: **Blocking, Locking, Signaling, Nonblocking**



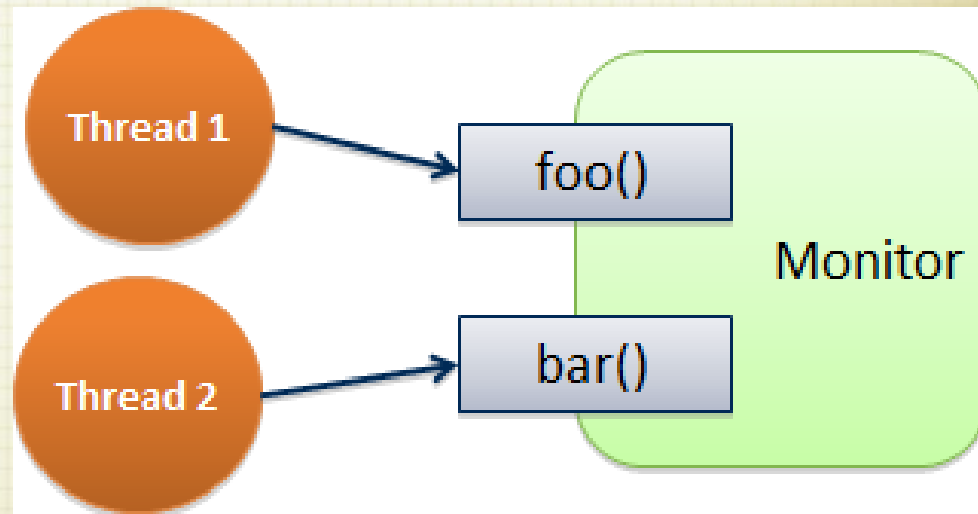
# Đồng bộ hóa (Synchronization)

- Đồng bộ hóa Blocking:
  - Chờ một thread khác kết thúc hoặc một khoảng thời gian nhất định trôi qua.



# Đồng bộ hóa (Synchronization)

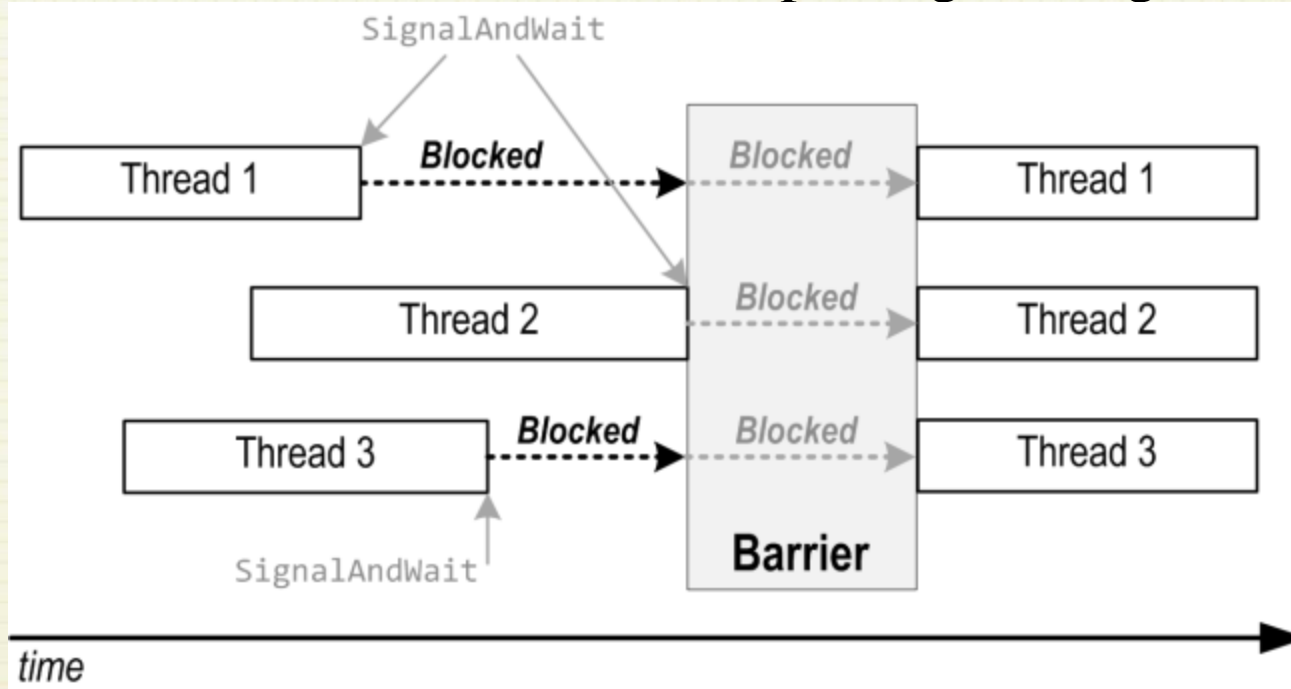
- Đồng bộ hóa Locking:
  - Giới hạn số lượng thread cùng thực hiện một thao tác hoặc một đoạn mã cùng một lúc
  - Exclusive locking
    - Lock (Monitor.Enter/Monitor.Exit)
    - Mutex
    - SpinLock
  - Nonexclusive locking
    - Semaphore
    - SemaphoreSlim



# Đồng bộ hóa (Synchronization)

- Đồng bộ hóa Signaling:

- Cho phép một thread tạm dừng cho tới khi nhận được thông báo (signal) từ một thread khác.
- Tránh việc kiểm tra điều kiện (polling) không cần thiết.





# Đồng bộ hóa (Synchronization)

- Đồng bộ hóa Nonblocking:
  - Bảo vệ sự truy cập vào những tài nguyên chung bằng cách gọi các processor primitive
  - Các lớp Nonblocking trong .NET:
    - Thread.MemoryBarrier
    - Thread.VolatileRead
    - Thread.VolatileWrite
    - Interlocked

# Đồng bộ hóa (Synchronization)

**Ví dụ:** Hai Thread sẽ tiến hành tăng tuần tự 1 đơn vị cho một biến counter

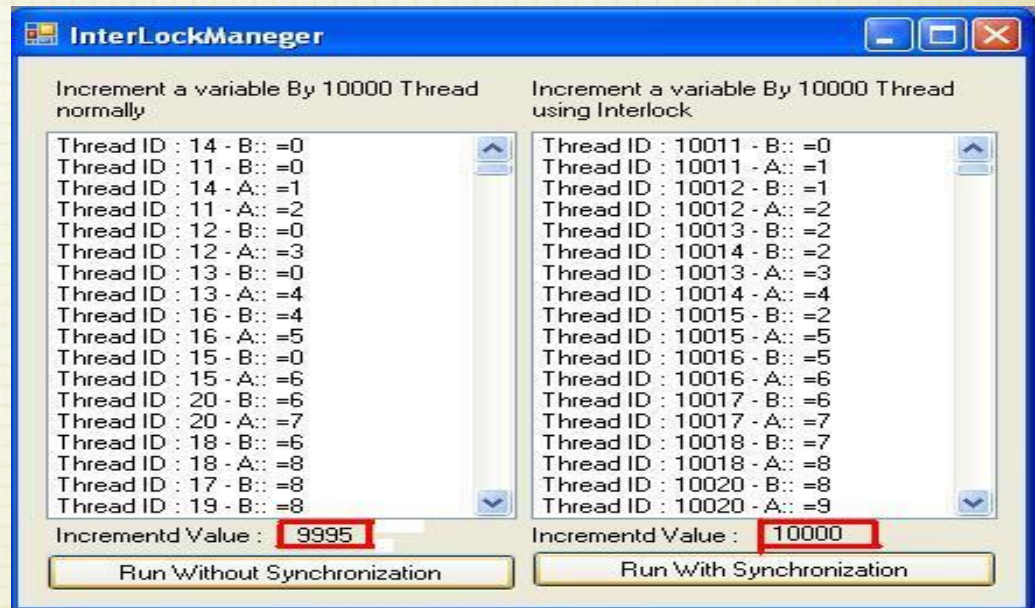
Hàm làm thay đổi giá trị của Counter:

```
public void Incrementer( ){  
    while (counter < 1000)  
    {  
        int temp = counter;  
        temp++; // increment  
        // simulate some work in this method  
        Thread.Sleep(1);  
        // assign the Incremented value to the counter  
        // variable and display the results  
        counter = temp;  
        Console.WriteLine("Thread {0}.  
Incrementer:{1}", Thread.CurrentThread.Name, counter);  
    }  
}
```



# Đồng bộ hóa (Synchronization)

- CLR cung cấp một lớp đặc biệt **Interlocked** nhằm đáp ứng nhu cầu tăng giảm giá trị.
- Interlocked có hai phương thức **Increment()** và **Decrement()** nhằm tăng và giảm giá trị trong sự bảo vệ của cơ chế đồng bộ.



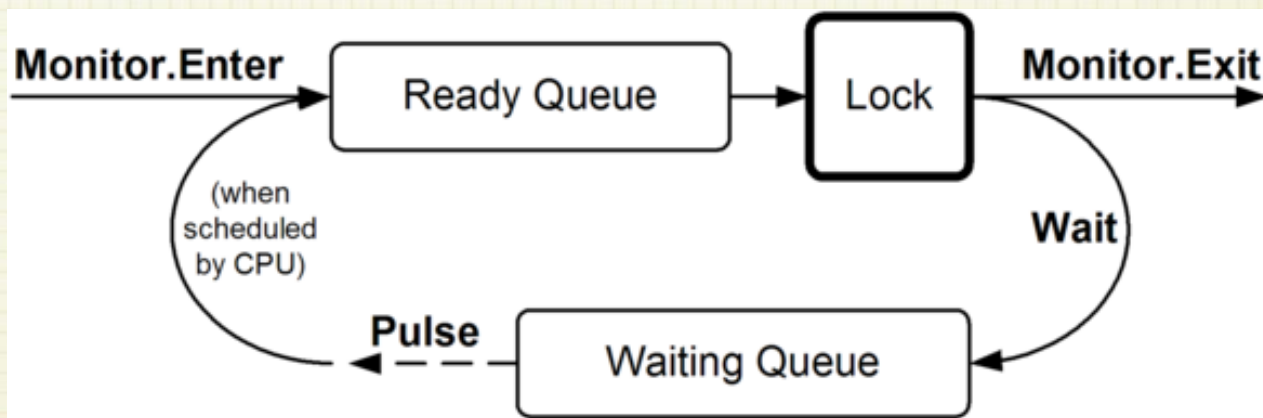


# Interlocked

```
public void Incrementer()  
{  
    while (counter < 1000)  
    {  
        Interlocked.Increment(ref counter);  
        // simulate some work in this method  
        Thread.Sleep(1);  
        // assign the decremented value and  
        display the results  
        Console.WriteLine("Thread {0}.  
        Incrementer: {1}",  
        Thread.CurrentThread.Name, counter);  
    }  
}
```

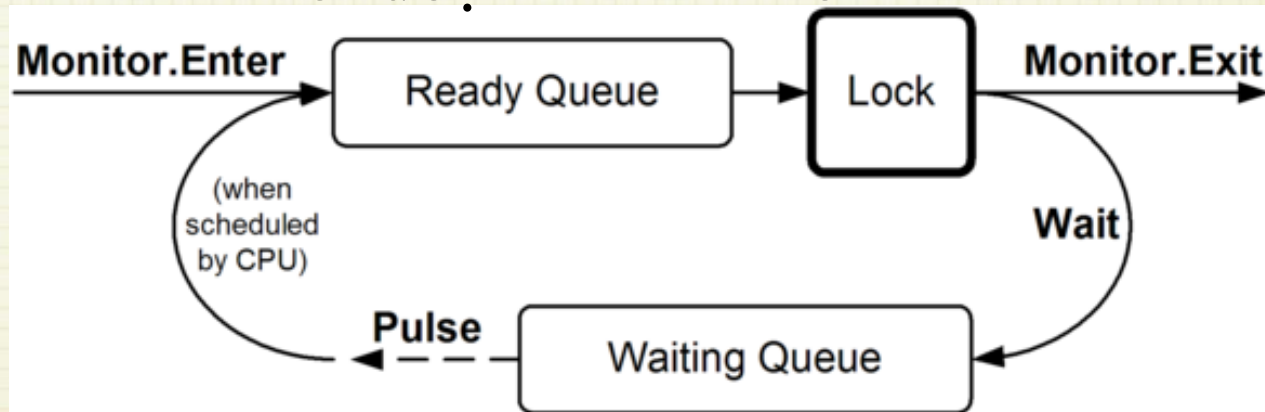
# Locks

- **Lock** đánh dấu một đoạn mã then chốt (critical section) trong chương trình, cung cấp cơ chế đồng bộ cho khối mã mà lock có hiệu lực.
- C# cung cấp sự hỗ trợ cho lock bằng từ khóa **lock**. Lock được gỡ bỏ khi hết khối lệnh. Lock tương đương với một cặp **Monitor.Enter/Monitor.Exit**



# Locks

- Khi vào khối lock CLR sẽ kiểm tra tài nguyên được khóa trong lock:
  - Nếu tài nguyên bị chiếm giữ thì tiếp tục chờ, quay lại kiểm tra sau một khoảng thời gian.
  - Nếu không bị khóa thì vào thực thi đoạn mã bên trong, đồng thời khóa tài nguyên lại.
  - Khi thoát khỏi đoạn mã thì mở khóa cho tài nguyên.





# Locks

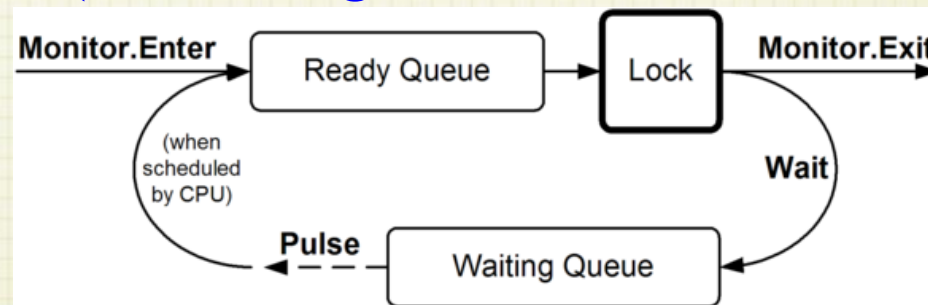
```
public void Incrementer( )
{
    while (counter < 1000)
    {
        lock (this)
        { // lock bắt đầu có hiệu lực
            int temp = counter;
            temp ++;
            Thread.Sleep(1);
            counter = temp;
        } // lock hết hiệu lực -> bị gỡ bỏ
        //assign the decremented value and display the results
        Console.WriteLine("Thread {0}. Incrementer: {1}",
            Thread.CurrentThread.Name, counter);
    }
}
```

Tài nguyên được khóa

Khởi mã được khóa

# Monitor

- Để có thể đồng bộ hóa phức tạp hơn cho tài nguyên, chúng ta cần sử dụng **monitor**.
- **Monitor** cho ta khả năng quyết định khi nào thì bắt đầu, khi nào thì kết thúc đồng bộ và khả năng chờ đợi một khối mã nào đó của chương trình “tự do”. Khi cần bắt đầu đồng bộ hóa, trao đối tượng cần đồng bộ cho hàm: **Monitor.Enter(đối tượng X);**



# Monitor

- Lời gọi `Wait()` giải phóng monitor nhưng CLR muốn lấy lại monitor ngay sau khi monitor được tự do một lần nữa. Thread thực thi phương thức `Wait()` sẽ bị treo lại. Các thread đang treo vì chờ đợi monitor sẽ tiếp tục chạy khi thread đang thực thi gọi hàm **`Pulse()`**:  
`Monitor.Pulse(this);`
- Khi thread hoàn tất việc sử dụng monitor, nó gọi hàm **`Exit()`** để trả monitor: `Monitor.Exit(this);`



# Monitor

- Ví dụ: Đang download và in một bài báo từ Web. Để hiệu quả bạn cần tiến hành in background, tuy nhiên cần chắc chắn rằng 10 trang đã được download trước khi bạn tiến hành in. Thread in ấn sẽ chờ đợi cho đến khi thread download báo hiệu rằng số lượng trang download đã đủ. Bạn không muốn gia nhập (join) với thread download vì số lượng trang có thể lên đến vài trăm. Bạn muốn chờ cho đến khi ít nhất 10 trang đã được download.

# Monitor

- Để giả lập việc này, chúng ta thiết lập 2 hàm đếm dùng chung 1 biến counter. Một hàm đếm tăng 1 tương ứng với thread download, một hàm đếm giảm 1 tương ứng với thread in ấn. Trong hàm làm giảm chúng ta gọi phương thức **Enter()**, sau đó kiểm tra giá trị counter, nếu  $< 10$  thì gọi hàm **Wait()**

```
if (counter < 10){  
    Monitor.Wait(this);  
}
```

# Monitor – Ví dụ

```
namespace Programming_CSharp
{
    using System;
    using System.Threading;
    class Tester
    {
        static void Main( )
        {
            // make an instance of this class
            Tester t = new Tester( );
            // run outside static Main
            t.DoTest( );
        }
    }
}
```



# Monitor – Ví dụ

```
public void DoTest( ) {  
    // create an array of unnamed threads  
    Thread[] myThreads = {  
        new Thread(new ThreadStart(Decrementer)),  
        new Thread(new ThreadStart(Incrementer)) };  
    // start each thread  
    int ctr = 1;  
    foreach (Thread myThread in myThreads)  
    {  
        myThread.IsBackground = true;  
        myThread.Start( );  
        myThread.Name = "Thread" + ctr.ToString( );  
        ctr++;  
        Console.WriteLine("Started thread {0}",  
            myThread.Name);  
        Thread.Sleep(50);  
    }  
}
```

# Monitor – Ví dụ

```
// wait for all threads to end before continuing
foreach (Thread myThread in myThreads) {
    myThread.Join( );
}
// after all threads end, print a message
Console.WriteLine("All my threads are done.");
}

void Decrementer( ){
    try
    { //Synchronize this area of code
        Monitor.Enter(this);
        //if counter is not yet 10 then free the monitor to other
        //waiting threads, but wait in line for your turn
        if (counter < 10) {
            Console.WriteLine("[{0}] In Decrementer. Counter:
            {1}. GottaWait!", Thread.CurrentThread.Name, counter);
            Monitor.Wait(this);
        }
    }
}
```

# Monitor – Ví dụ

```
while (counter > 0)
{
    long temp = counter;
    temp--;
    Thread.Sleep(1);
    counter = temp;
    Console.WriteLine("[{0}] In Decrementer.
Counter:    {1}.", Thread.CurrentThread.Name,
counter);
}
}
finally{
    Monitor.Exit(this);
}
}
```



# Monitor – Ví dụ

```
void Incrementer( )
{
    try
    {
        Monitor.Enter(this);
        while (counter < 10)
        {
            long temp = counter;
            temp++;
            Thread.Sleep(1);
            counter = temp;
            Console.WriteLine("[{0}] In Incrementer.
Counter:{1}", Thread.CurrentThread.Name, counter);
        }
    }
}
```

# Monitor – Ví dụ

```
// I'm done incrementing for now, let another
// thread have the Monitor
Monitor.Pulse(this);
}
finally
{
    Console.WriteLine("[{0}] Exiting...",
                      Thread.CurrentThread.Name);
    Monitor.Exit(this);
}
}
private long counter = 0;
}
}
```

# Monitor – Ví dụ

Kết quả:

Started thread Thread1

[Thread1] In Decrementer. Counter: 0. Gotta Wait!

Started thread Thread2

[Thread2] In Incrementer. Counter: 1

[Thread2] In Incrementer. Counter: 2

[Thread2] In Incrementer. Counter: 3

[Thread2] In Incrementer. Counter: 4

[Thread2] In Incrementer. Counter: 5

[Thread2] In Incrementer. Counter: 6

[Thread2] In Incrementer. Counter: 7

[Thread2] In Incrementer. Counter: 8

[Thread2] In Incrementer. Counter: 9

[Thread2] In Incrementer. Counter: 10

[Thread2] Exiting...

[Thread1] In Decrementer. Counter: 9.

[Thread1] In Decrementer. Counter: 8.

[Thread1] In Decrementer. Counter: 7.

[Thread1] In Decrementer. Counter: 6.

[Thread1] In Decrementer. Counter: 5.

[Thread1] In Decrementer. Counter: 4.

[Thread1] In Decrementer. Counter: 3.

[Thread1] In Decrementer. Counter: 2.

[Thread1] In Decrementer. Counter: 1.

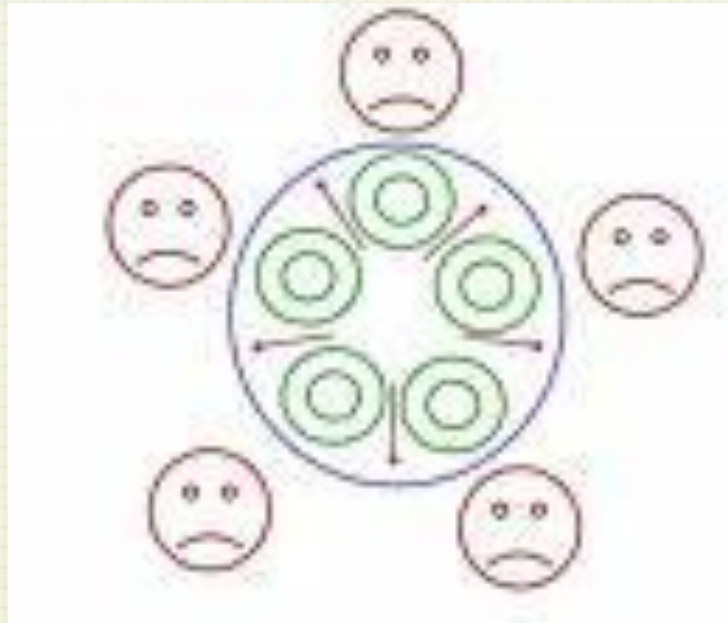
[Thread1] In Decrementer. Counter: 0.

All my threads are done.



# Race condition và DeadLock

- Đồng bộ hóa thread khá rắc rối trong những chương trình phức tạp. Vì vậy chúng ta cần phải cẩn thận kiểm tra và giải quyết các vấn đề liên quan đến đồng bộ hóa thread: **race condition** và **deadlock**

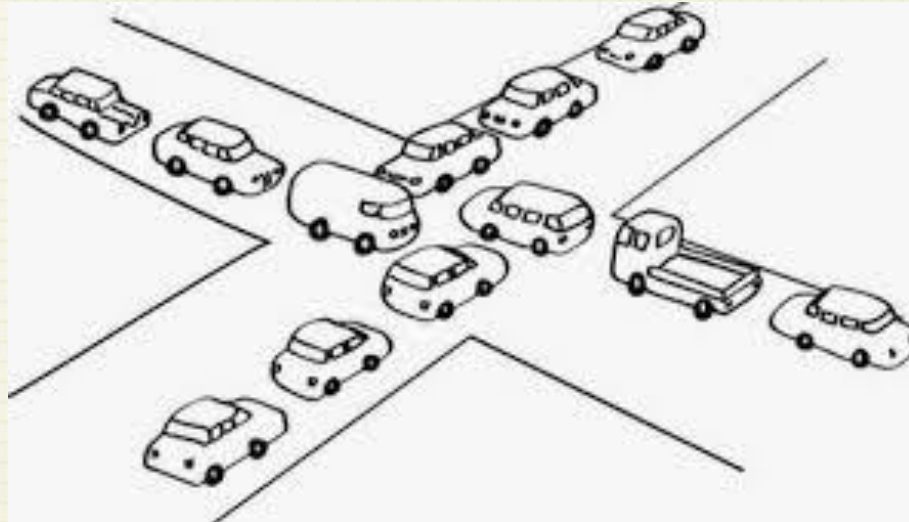


# Race condition

- Một điều kiện tranh đua xảy ra khi sự đúng đắn của ứng dụng **phụ thuộc** vào thứ tự hoàn thành **không kiểm soát được** của 2 thread **độc lập** với nhau.
- Giả sử có 2 thread. Thread 1 tiến hành mở tập tin, thread 2 tiến hành ghi lên cùng tập tin đó. Cần phải điều khiển thread 2 sao cho nó chỉ tiến hành công việc sau khi thread 1 đã tiến hành xong. Nếu không, thread 1 sẽ không mở được tập tin vì tập tin đó đã bị thread 2 mở để ghi. Kết quả là chương trình sẽ ném ra exception hoặc tệ hơn nữa là crash.

# Deadlock

- Trong một chương trình nhiều thread, deadlock rất khó phát hiện và gỡ lỗi. Một hướng dẫn để tránh deadlock đó là giải phóng tất cả lock đang sở hữu nếu tất cả các lock cần nhận không thể nhận hết được. Một hướng dẫn khác đó là giữ lock càng ít càng tốt.





# Ví dụ Producer – Consumer

- Tiến trình sản xuất tạo dữ liệu và đặt vào bộ đệm
  - Buffer: vùng chia sẻ của bộ nhớ
- Bên tiêu thụ đọc dữ liệu từ bộ đệm
- Sản xuất và tiêu thụ nên liên lạc cho phép dữ liệu thích hợp nào được đọc
- Các lỗi logic xảy ra nếu các tiến trình **chưa được đồng bộ hóa**:
  - Sản xuất có thể ghi đè dữ liệu trước khi tiêu thụ đọc.
  - Tiêu thụ đọc dữ liệu sai hoặc là hai lần dữ liệu như nhau

```
1  //Fig. 14.4: Unsynchronized.cs
2  //Showing multiple threads modifying a shared object without synchronization.
3  using System;
4  using System.Threading;
5  //This class represents a single shared int
6  public class HoldIntegerUnsynchronized
7  {
8      // buffer shared by producer and consumer threads
9      private int buffer = -1;
10     // property Buffer
11     public int Buffer
12     {
13         get
14         {
15             Console.WriteLine( Thread.CurrentThread.Name + " reads " + buffer );
16             return buffer;
17         }
18         set
19         {
20             Console.WriteLine( Thread.CurrentThread.Name + " writes " + value );
21             buffer = value;
22         }
23     } // end property Buffer
24 } // end class HoldIntegerUnsynchronized
```

Buffer class

Integer shared by consumer and producer (buffer)

Accessor to read buffer

Accessor to write to buffer

```

25 // class Producer's Produce method controls a thread that Unsyncronized.cs
26 // stores values from 1 to 4 in sharedLocation
27 class Producer
28 {
29     private HoldIntegerUnsyncronized sharedLocation;
30     private Random randomSleepTime;
31     // constructor
32     public Producer(HoldIntegerUnsyncronized shared, Random random)
33     {
34         sharedLocation = shared;
35         randomSleepTime = random;
36     }
37     // store values 1-4 in object sharedLocation
38     public void Produce()
39     {
40         // sleep for random interval upto 3000 milliseconds
41         // then set sharedLocation's Buffer property
42         for (int count = 1; count <= 4; count++)
43         {
44             Thread.Sleep(randomSleepTime.Next( 1, 3000 ) );
45             sharedLocation.Buffer = count;
46         }
47         Console.WriteLine(Thread.CurrentThread.Name +
48             " done producing.\nTerminating " + Thread.CurrentThread.Name + "." );
49     } // end method Produce
50 } // end class Producer

```

Producer class

Set buffer as shared object

Set sleep time

Put buffer to sleep

Set buffer to count

Tell user thread is  
done producing



```
51 // class Consumer's Consume method controls a thread that Unsyncronized.cs
52 // loops four times and reads a value from sharedLocation
53 class Consumer ← Consumer Class
54 {
55     private HoldIntegerUnsyncronized sharedLocation;
56     private Random randomSleepTime;
57     // constructor
58     public Consumer(HoldIntegerUnsyncronized shared, Random random)
59     {
60         sharedLocation = shared; ← Set shared to buffer
61         randomSleepTime = random; ← Set sleep time
62     }
63     // read sharedLocation's value four times
64     public void Consume()
65     {
66         int sum = 0; ← Set sum to 0
67         // sleep for random interval up to 3000 milliseconds
68         // then add sharedLocation's Buffer property value to sum
69         for (int count = 1; count <= 4; count++)
70         {
71             Thread.Sleep(randomSleepTime.Next(1, 3000 ));
72             sum += sharedLocation.Buffer; ← Add value in
73             }                                     buffer to sum
```

```
74 Console.WriteLine( Thread.CurrentThread.Name +  
75     " read values totaling: " + sum +  
76     ".\nTerminating " + Thread.CurrentThread.Name + "." );  
77  
78 } // end method Consume  
79 } // end class Consumer  
80 // this class creates producer and consumer threads  
81 class SharedCell  
82 {  
83     //Create producer and consumer threads and start them  
84     static void Main(string[] args)  
85     {  
86         //Create shared object used by threads  
87         HoldIntegerUnsynchronized holdInteger = new  
88         HoldIntegerUnsynchronized();  
89         //Random object used by each thread  
90         Random random = new Random();  
91         //Create Producer and Consumer objects  
92         Producer producer = new Producer(holdInteger, random);  
93         Consumer consumer = new Consumer(holdInteger, random);  
94         //Create threads for producer and consumer and set  
95         //delegates for each thread  
96         Thread producerThread = new Thread(new ThreadStart(producer.Produce));  
97         producerThread.Name = "Producer";
```

Tell user sum and  
that thread is done

Create buffer

Create random number  
for sleep times

Create producer object

Create consumer object

Create producer thread

```
98 Thread consumerThread =  
99     new Thread(new ThreadStart(consumer.Consume));  
100     consumerThread.Name = "Consumer";  
101  
102     // start each thread  
103     producerThread.Start();  
104     consumerThread.Start();  
105  
106 } // end method Main  
107  
108 } // end class SharedCell
```



Create consumer thread



Start producer thread



Start consumer thread

```
Consumer reads -1  
Producer writes 1  
Consumer reads 1  
Consumer reads 1  
Consumer reads 1  
Consumer read values totaling: 2.  
Terminating Consumer.  
Producer writes 2  
Producer writes 3  
Producer writes 4  
Producer done producing.  
Terminating Producer.
```

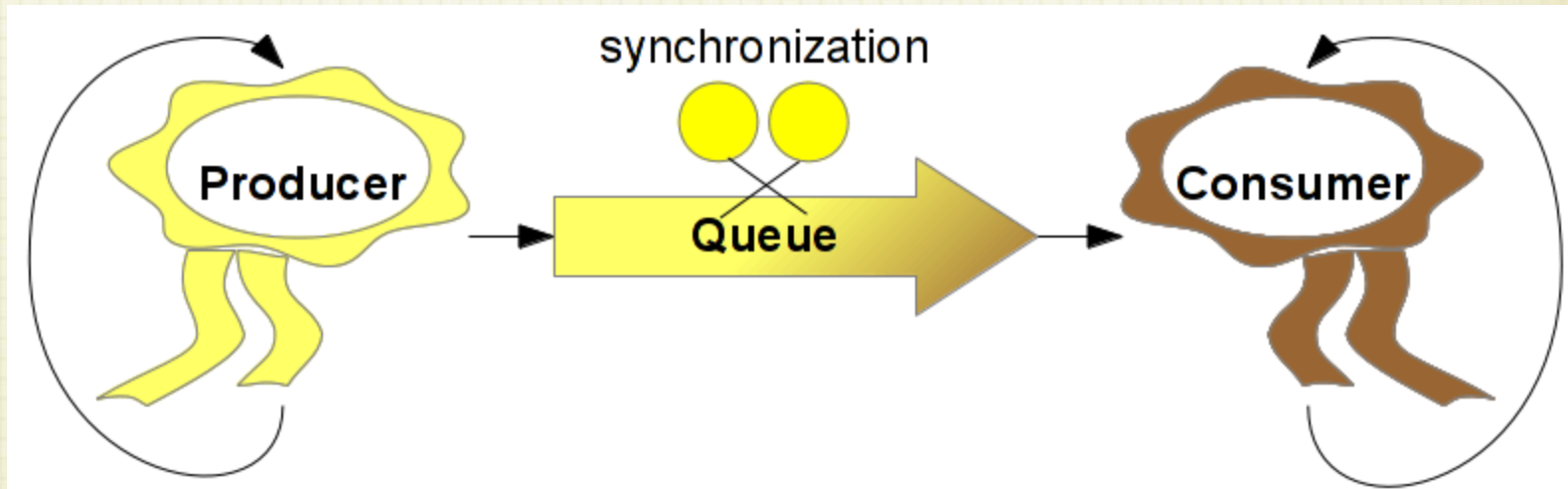


```
Producer writes 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer reads 4
Consumer read values totaling: 13.
Terminating Consumer.
```

```
Producer writes 1
Consumer reads 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer read values totaling: 10.
Terminating Consumer.
```

# Ví dụ Producer – Consumer

- Đồng bộ hóa bảo đảm rằng các kết quả chính xác có thể đạt được:
  - Sản xuất chỉ có thể sinh ra các kết quả sau khi tiêu thụ đọc kết quả trước đó
  - Tiêu thụ chỉ dùng được khi sản xuất ghi dữ liệu mới



```
1  // Fig. 14.5: Synchronized.cs
2  // Showing multiple threads modifying a shared object with synchronization.
3  using System;
4  using System.Threading;
5  // this class synchronizes access to an integer
6  public class HoldIntegerSynchronized
7  {
8      // buffer shared by producer and consumer threads
9      private int buffer = -1;
10     // occupiedBufferCount maintains count of occupied buffers
11     private int occupiedBufferCount = 0;
12     // property Buffer
13     public int Buffer
14     {
15         get{
16             // obtain lock on this object
17             Monitor.Enter(this);
18             // if there is no data to read, place invoking
19             // thread in WaitSleepJoin state
20             if(occupiedBufferCount == 0)
21             {
22                 Console.WriteLine(
23                     Thread.CurrentThread.Name + " tries to read. ");
24                 DisplayState("Buffer empty. " +
25                     Thread.CurrentThread.Name + " waits.");
```

Create buffer

Variable to determine  
whose turn to use buffer

Method to get value  
from buffer

Get lock

See if buffer is occupied



```
26     Monitor.Wait(this);
27 }
28 // indicate that producer can store another value
29 // because a consumer just retrieved buffer value
30 --occupiedBufferCount;
31 DisplayState(Thread.CurrentThread.Name + " reads " + buffer );
32 // tell waiting thread (if there is one) to
33 // become ready to execute (Started state)
34 Monitor.Pulse(this);
35 // Get copy of buffer before releasing lock.
36 // It is possible that the producer could be
37 // assigned the processor immediately after the
38 // monitor is released and before the return
39 // statement executes. In this case, the producer
40 // would assign a new value to buffer before the
41 // return statement returns the value to the
42 // consumer. Thus, the consumer would receive the
43 // new value. Making a copy of buffer and
44 // returning the copy ensures that the
45 // consumer receives the proper value.
46 int bufferCopy = buffer;
47 // release lock on this object
48 Monitor.Exit(this);
49 return bufferCopy;
50 } // end get
```

If buffer unoccupied,  
put consumer to sleep

Tell system buffer  
has been read

Get producer out of wait state

Make copy of buffer

Release lock on buffer

Return value of buffer

```
51 set ← Method to write to buffer
52 {
53     // acquire lock for this object
54     Monitor.Enter(this); ← Get lock
55     // if there are no empty locations, place invoking
56     // thread in WaitSleepJoin state
57     if(occupiedBufferCount == 1) ← Test if buffer is occupied
58     {
59         Console.WriteLine(
60             Thread.CurrentThread.Name + " tries to write.");
61         DisplayState("Buffer full. " +
62             Thread.CurrentThread.Name + " waits.");
63         Monitor.Wait(this); ← If buffer occupied, put
64                               producer to sleep
65     }
66     // set new buffer value
67     buffer = value; ← Write to buffer
68     // indicate producer cannot store another value
69     // until consumer retrieves current buffer value
70     ++occupiedBufferCount; ← Tell system buffer
71                               has been written to
72     DisplayState(Thread.CurrentThread.Name + " writes " + buffer);
73
74     // tell waiting thread (if there is one) to
75     // become ready to execute (Started state)
76     Monitor.Pulse(this); ← Release consumer
77                               from wait state
```

```
75         // release lock on this object
76         Monitor.Exit(this);
77     } // end set
78 }
79 // display current operation and buffer state
80 public void DisplayState(string operation )
81 {
82     Console.WriteLine( "{0,-35}{1,-9}{2}\n",
83         operation, buffer, occupiedBufferCount );
84 }
85 } // end class HoldIntegerSynchronized
86
87 // class Producer's Produce method controls a thread that
88 // stores values from 1 to 4 in sharedLocation
89 class Producer
90 {
91     private HoldIntegerSynchronized sharedLocation;
92     private Random randomSleepTime;
93     // constructor
94     public Producer(HoldIntegerSynchronized shared, Random random)
95     {
96         sharedLocation = shared;
97         randomSleepTime = random;
98     }
```



Release lock



Producer Class



Set sharedLocation to buffer



Set sleep time



```
99  //Store values 1-4 in object sharedLocation
100 public void Produce()
101 {
102     // sleep for random interval up to 3000 milliseconds
103     // then set sharedLocation's Buffer property
104     for (int count = 1; count <= 4; count++)
105     {
106         Thread.Sleep(randomSleepTime.Next(1, 3000));
107         sharedLocation.Buffer = count;
108     }
109     Console.WriteLine(Thread.CurrentThread.Name +
110         " done producing.\nTerminating "+Thread.CurrentThread.Name+".\n");
111 } // end method Produce
112 } // end class Producer
113 // class Consumer's Consume method controls a thread that
114 // loops four times and reads a value from sharedLocation
115 class Consumer
116 {
117     private HoldIntegerSynchronized sharedLocation;
118     private Random randomSleepTime;
119     // constructor
120     public Consumer(HoldIntegerSynchronized shared, Random random)
121     {
122         sharedLocation = shared;
123         randomSleepTime = random;
124     }
```

Annotations:

- Loop 4 times (points to line 104)
- Put thread to sleep (points to line 106)
- Set buffer equal to count (points to line 107)
- Tell user thread is done (points to line 109)
- Consumer class (points to line 115)
- Set sharedLocation to buffer (points to line 122)
- Set sleep time (points to line 123)

```
125 //Read sharedLocation's value four times
126 public void Consume()
127 {
128     int sum = 0;
129     // get current thread
130     Thread current = Thread.CurrentThread;
131     // sleep for random interval up to 3000 milliseconds
132     // then add sharedLocation's Buffer property value to sum
133     for ( int count = 1; count <= 4; count++)
134     {
135         Thread.Sleep(randomSleepTime.Next(1, 3000));
136         sum += sharedLocation.Buffer;
137     }
138     Console.WriteLine(Thread.CurrentThread.Name +
139         " read values totaling: " + sum +
140         ".\nTerminating " + Thread.CurrentThread.Name + ".\n");
141 } // end method Consume
142 } // end class Consumer
143 class SharedCell //This class creates producer and consumer threads
144 {
145     // create producer and consumer threads and start them
146     static void Main(string[] args)
147     {
148         // create shared object used by threads
149         HoldIntegerSynchronized holdInteger = new HoldIntegerSynchronized();
```

Diagram illustrating the execution flow and key operations in the `Consume` method:

- Loop 4 times**: Points to the `for` loop header (line 133).
- Put thread to sleep**: Points to the `Thread.Sleep` call (line 135).
- Add buffer to sum**: Points to the `sum += sharedLocation.Buffer` statement (line 136).
- Tell user thread is finished and sum**: Points to the `Console.WriteLine` statement (line 139).
- Create buffer**: Points to the `new HoldIntegerSynchronized()` instantiation (line 149).

```
150 // Random object used by each thread
151 Random random = new Random();
152 // create Producer and Consumer objects
153 Producer producer = new Producer(holdInteger, random);
154 Consumer consumer = new Consumer(holdInteger, random);
155 // output column heads and initial buffer state
156 Console.WriteLine("{0,-35}{1,-9}{2}\n",
157     "Operation", "Buffer", "Occupied Count");
158 holdInteger.DisplayState("Initial state");
159
160 // create threads for producer and consumer and set
161 // delegates for each thread
162 Thread producerThread =
163     new Thread(new ThreadStart(producer.Produce));
164 producerThread.Name = "Producer";
165
166 Thread consumerThread =
167     new Thread(new ThreadStart(consumer.Consume));
168 consumerThread.Name = "Consumer";
169 // start each thread
170 producerThread.Start();
171 consumerThread.Start();
172 } // end method Main
173 } // end class SharedCell
```

Create random number  
for sleep times

Create producer object

Create consumer object

Create producer thread

Create consumer thread

Start producer thread

Start consumer thread



# Synchronized.cs Program Output

Operation	Buffer	Occupied Count
Initial state	-1	0
Producer writes 1	1	1
Consumer reads 1	1	0
Consumer tries to read.		
Buffer empty. Consumer waits.	1	0
Producer writes 2	2	1
Consumer reads 2	2	0
Producer writes 3	3	1
Producer tries to write.		
Buffer full. Producer waits.	3	1
Consumer reads 3	3	0
Producer writes 4	4	1
Producer done producing.		
Terminating Producer.		
Consumer reads 4	4	0
Consumer read values totaling: 10.		
Terminating Consumer.		

Program  
Output

Operation	Buffer	Occupied Count
Initial state	-1	0
Consumer tries to read.		
Buffer empty. Consumer waits.	-1	0
Producer writes 1	1	1
Consumer reads 1	1	0
Producer writes 2	2	1
Consumer reads 2	2	0
Producer writes 3	3	1
Producer tries to write.		
Buffer full. Producer waits.	3	1
Consumer reads 3	3	0
Producer writes 4	4	1
Producer done producing.		
Terminating Producer.		
Consumer reads 4	4	0
Consumer read values totaling: 10.		
Terminating Consumer.		

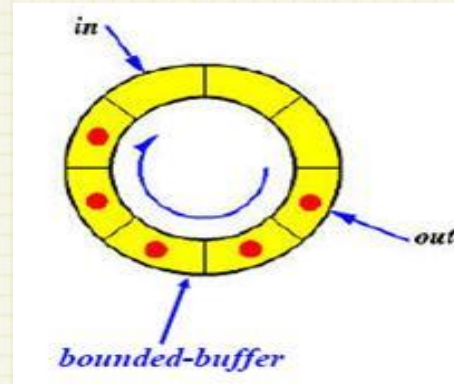
# Synchronized.cs Program Output

Operation	Buffer	Occupied Count
Initial state	-1	0
Producer writes 1	1	1
Consumer reads 1	1	0
Producer writes 2	2	1
Consumer reads 2	2	0
Producer writes 3	3	1
Consumer reads 3	3	0
Producer writes 4	4	1
Producer done producing. Terminating Producer.		
Consumer reads 4	4	0
Consumer read values totaling: 10. Terminating Consumer.		



# Quan hệ sản xuất/tiêu thụ: bộ đệm vòng

- Hai tiến trình đã được đồng bộ hóa và chia sẻ tài nguyên có thể gây chậm trễ



- Bộ đệm vòng:
  - Các bộ đệm thêm vào để được ghi và đọc
  - Có thể được thực hiện với một mảng
    - Sản xuất và tiêu thụ bắt đầu (start) lúc ban đầu
    - Khi đến cuối mảng, tiến trình trở lại điểm bắt đầu
    - Khi một tiến trình hiện thời nhanh hơn các tiến trình khác, nó sử dụng thêm các bộ đệm để tiếp tục thực thi

```
1 // Fig. 14.6: CircularBuffer.cs
2 // Implementing the producer/consumer relationship with a circular buffer.
3 using System;
4 using System.Drawing;
5 using System.Collections;
6 using System.ComponentModel;
7 using System.Windows.Forms;
8 using System.Data;
9 using System.Threading;
10 // implement the shared integer with synchronization
11 public class HoldIntegerSynchronized
12 {
13     // each array element is a buffer
14     private int[] buffers = { -1, -1, -1 };
15     // occupiedBufferCount maintains count of occupied buffers
16     private int occupiedBufferCount = 0;
17     // variable that maintain read and write buffer locations
18     private int readLocation = 0, writeLocation = 0;
19     // GUI component to display output
20     private TextBox outputTextBox;
21     // constructor
22     public HoldIntegerSynchronized( TextBox output )
23     {
24         outputTextBox = output;
25     }
```

Circular buffer

How many buffers  
are occupied

Next read location

Next write location

Create textbox

```
26 //property Buffer
27 public int Buffer
28 {
29     get
30     {
31         //lock this object while getting value from buffers array
32         lock (this)
33         {
34             //if there is no data to read, place invoking
35             //thread in WaitSleepJoin state
36             if(occupiedBufferCount == 0)
37             {
38                 outputTextBox.Text += "\r\nAll buffers empty. " +
39                     Thread.CurrentThread.Name + " waits.";
40                 outputTextBox.ScrollToCaret();
41                 Monitor.Wait(this);
42             }
43             //obtain value at current readLocation, then
44             //add string indicating consumed value to output
45             int readValue = buffers[ readLocation ];
46             outputTextBox.Text += "\r\n" +
47                 Thread.CurrentThread.Name + " reads " +
48                 buffers[ readLocation ] + " ";
49             //just consumed a value, so decrement number of occupied buffers
50             --occupiedBufferCount;
```

Method to read from buffer

Get lock

Test if any buffers occupied

If no buffers occupied, consumer must wait

Read value from correct buffer

Output value read

Decrement number of buffers occupied



```

51 // update readLocation for future read operation,
52 // then add current state to output
53     readLocation = (readLocation + 1) % buffers.Length;
54     outputTextBox.Text += CreateStateOutput();
55     outputTextBox.ScrollToCaret();
56     // return waiting thread (if there is one) to Started state
57     Monitor.Pulse(this);
58     return readValue;
59 } // end lock
60 } // end accessor get
61 set
62 {
63     // lock this object while setting value in buffers array
64     lock(this)
65     {
66         // if there are no empty locations, place invoking
67         // thread in WaitSleepJoin state
68         if(occupiedBufferCount == buffers.Length)
69         {
70             outputTextBox.Text += "\r\nAll buffers full. " +
71                 Thread.CurrentThread.Name + " waits.";
72             outputTextBox.ScrollToCaret();
73             Monitor.Wait(this);
74         }

```

Update readLocation

Call CreateStateOutput

Get producer from wait state

Method to write to buffer

Get lock

Test if all buffers are occupied

If all buffers occupied, producer must wait

```

75 //place value in writeLocation of buffers, thenCircularBuffer.cs
76 //add string indicating produced value to output
77 buffers[ writeLocation ] = value;
78 outputTextBox.Text += "\r\n" +
79     Thread.CurrentThread.Name + " writes " +
80     buffers[ writeLocation ] + " ";
81 // just produced a value, so increment number of occupied buffers
82 ++occupiedBufferCount;
83 // update writeLocation for future write operation
84 // then add current state to output
85 writeLocation = (writeLocation + 1) % buffers.Length;
86 outputTextBox.Text += CreateStateOutput();
87 outputTextBox.ScrollToCaret();
88 // return waiting thread (if there is one) to Started state
89 Monitor.Pulse( this );
90 } // end lock
91 } // end accessor set
92 } // end property Buffer
93 // create state output
94 public string CreateStateOutput()
95 {
96     // display first line of state information
97     string output = "(buffers occupied: " +
98         occupiedBufferCount + ")\r\nbuffers: ";

```

Put new value in next location of buffer

Output value written to buffer

Increment number of buffers occupied

Update write location

Call CreateStateOutput

Get consumer from wait state

Output number of buffers occupied

```
99     for ( int i = 0; i < buffers.Length; i++ )
100         output += " " + buffers[ i ] + " ";
101     output += "\r\n";
102     // display second line of state information
103     output += "          ";
104     for ( int i = 0; i < buffers.Length; i++ )
105         output += "---- ";
106     output += "\r\n";
107     // display third line of state information
108     output += "          ";
109     // display readLocation (R) and writeLocation (W)
110     // indicators below appropriate buffer locations
111     for ( int i = 0; i < buffers.Length; i++ )
112         if ( i == writeLocation && writeLocation == readLocation )
113             output += " WR ";
114         else if ( i == writeLocation )
115             output += " W  ";
116         else if ( i == readLocation )
117             output += " R  ";
118         else
119             output += "    ";
120     output += "\r\n";
121     return output;
122 }
123 } //end class HoldIntegerSynchronized
```

Output contents of buffers

Output readLocation  
and writeLocation



```
124 // produce the integers from 11 to 20
125 // and place them in buffer
126 public class Producer ← Producer class
127 {
128     private HoldIntegerSynchronized sharedLocation;
129     private TextBox outputTextBox;
130     private Random randomSleepTime;
131     // constructor
132     public Producer( HoldIntegerSynchronized shared,
133         Random random, TextBox output )
134     {
135         sharedLocation = shared; ← Set shared location to buffer
136         outputTextBox = output; ← Set output
137         randomSleepTime = random; ← Set sleep time
138     }
139     // produce values from 11-20 and place them in sharedLocation's buffer
140     public void Produce()
141     {
142         // sleep for random interval up to 3000 milliseconds
143         // then set sharedLocation's Buffer property
144         for ( int count = 11; count <= 20; count++ ) ← Loop ten times
145         {
146             Thread.Sleep(randomSleepTime.Next( 1, 3000 ) ); ← Set sleep time
147             sharedLocation.Buffer = count; ← Write to buffer
148         }
```

```
149     string name = Thread.CurrentThread.Name;
150     outputTextBox.Text += "\r\n" + name +
151         " done producing.\r\n" + name + " terminated.\r\n";
152     outputTextBox.ScrollToCaret();
153 } // end method Produce
154 } // end class Producer
155 // consume the integers 1 to 10 from circular buffer
156 public class Consumer
157 {
158     private HoldIntegerSynchronized sharedLocation;
159     private TextBox outputTextBox;
160     private Random randomSleepTime;
161     // constructor
162     public Consumer( HoldIntegerSynchronized shared,
163         Random random, TextBox output )
164     {
165         sharedLocation = shared;
166         outputTextBox = output;
167         randomSleepTime = random;
168     }
169     // consume 10 integers from buffer
170     public void Consume()
171     {
172         int sum = 0;
```



Output to textbox



Consumer class



Set shared location  
to buffer



Set output



Set sleep time



Initialize sum to 0

```

173 // loop 10 times and sleep for random interval up to CircularBuffer.cs
174 // 3000 milliseconds then add sharedLocation's
175 // Buffer property value to sum
176 for ( int count = 1; count <= 10; count++ )
177 {
178     Thread.Sleep(randomSleepTime.Next(1, 3000));
179     sum += sharedLocation.Buffer;
180 }
181 string name = Thread.CurrentThread.Name;
182 outputTextBox.Text += "\r\nTotal " + name +
183     " consumed: " + sum + ".\r\n" + name + " terminated.\r\n";
184 outputTextBox.ScrollToCaret();
185 } // end method Consume
186 } // end class Consumer
187 // set up the producer and consumer and start them
188 public class CircularBuffer : System.Windows.Forms.Form
189 {
190     private System.Windows.Forms.TextBox outputTextBox;
191     // required designer variable
192     private System.ComponentModel.Container components = null;
193     // no-argument constructor
194     public CircularBuffer()
195     {
196         InitializeComponent();
197     }

```

Loop ten times

Put thread to sleep

Add value of  
buffer to sum

Output to textbox

```
198 //Visual Studio .NET GUI code appears here in source file CircularBuffer.cs
199 //main entry point for the application
200 [STAThread]
201 static void Main()
202 {
203     Application.Run( new CircularBuffer() );
204 }

205 // Load event handler creates and starts threads
206 private void CircularBuffer_Load( object sender, System.EventArgs e )
207 {
208     // create shared object
209     HoldIntegerSynchronized sharedLocation =
210         new HoldIntegerSynchronized(outputTextBox );
211     // display sharedLocation state before producer
212     // and consumer threads begin execution
213     outputTextBox.Text = sharedLocation.CreateStateOutput();
214     // Random object used by each thread
215     Random random = new Random();
216     // create Producer and Consumer objects
217     Producer producer =
218         new Producer( sharedLocation, random, outputTextBox );
219     Consumer consumer =
220         new Consumer( sharedLocation, random, outputTextBox );
221 }
```

Diagram illustrating the object creation and state setup in the `CircularBuffer_Load` method:

- Create buffer** points to line 209: `HoldIntegerSynchronized sharedLocation = new HoldIntegerSynchronized(outputTextBox );`
- Create random number for sleep times** points to line 215: `Random random = new Random();`
- Create producer object** points to line 218: `Producer producer = new Producer( sharedLocation, random, outputTextBox );`
- Create consumer object** points to line 220: `Consumer consumer = new Consumer( sharedLocation, random, outputTextBox );`



```
222 // create and name threads
223 Thread producerThread =
224     new Thread(new ThreadStart(producer.Produce));
225 producerThread.Name = "Producer";
226
227 Thread consumerThread =
228     new Thread(new ThreadStart(consumer.Consume));
229 consumerThread.Name = "Consumer";
230
231 // start threads
232 producerThread.Start();
233 consumerThread.Start();
234
235 } // end CircularBuffer_Load method
236
237 } // end class CircularBuffer
```



Create producer thread



Create consumer thread



Start producer thread



Start consumer thread

# CircularBuffer.cs

## Program Output

```
CircularBuffer
(buffers occupied: 0)
buffers:  -1  -1  -1
-----
          WR

All buffers empty. Consumer waits.
Producer writes 11 (buffers occupied: 1)
buffers:  11  -1  -1
-----
          R  W

Consumer reads 11 (buffers occupied: 0)
buffers:  11  -1  -1
-----
          WR

All buffers empty. Consumer waits.
Producer writes 12 (buffers occupied: 1)
buffers:  11  12  -1
-----
          R  W

Consumer reads 12 (buffers occupied: 0)
buffers:  11  12  -1
-----
          WR

Producer writes 13 (buffers occupied: 1)
buffers:  11  12  13
-----
          W          R
```

Value placed in last buffer.

Next value will be placed  
in leftmost buffer.

# CircularBuffer.cs

## Program Output

```
CircularBuffer
Producer writes 14 (buffers occupied: 2)
buffers: 14 12 13
      -----
            W   R

Consumer reads 13 (buffers occupied: 1)
buffers: 14 12 13
      -----
            R   W

Consumer reads 14 (buffers occupied: 0)
buffers: 14 12 13
      -----
            WR

Producer writes 15 (buffers occupied: 1)
buffers: 14 15 13
      -----
            R   W

Producer writes 16 (buffers occupied: 2)
buffers: 14 15 16
      -----
            W   R

Producer writes 17 (buffers occupied: 3)
buffers: 17 15 16
      -----
            WR
```

Circular buffer effect – the fourth value is deposited in the left most buffer.

Value placed in last buffer.

Next value will be placed in left most buffer

Circular buffer effect – the seventh value is deposited in the left most buffer.

# CircularBuffer.cs

## Program Output

```
CircularBuffer
Consumer reads 15 (buffers occupied: 2)
buffers:  17  15  16
-----
          W    R

Producer writes 18 (buffers occupied: 3)
buffers:  17  18  16
-----
          WR

All buffers full. Producer waits.
Consumer reads 16 (buffers occupied: 2)
buffers:  17  18  16
-----
          R    W

Producer writes 19 (buffers occupied: 3)
buffers:  17  18  19
-----
          WR

All buffers full. Producer waits.
Consumer reads 17 (buffers occupied: 2)
buffers:  17  18  19
-----
          W    R

Producer writes 20 (buffers occupied: 3)
buffers:  20  18  19
-----
          WR
```

Value placed in last buffer.

Next value will be placed in  
left most buffer

Circular buffer effect – the  
tenth value is deposited in  
the left most buffer.

```
CircularBuffer
Producer done producing.
Producer terminated.

Consumer reads 18 (buffers occupied: 2)
buffers:  20  18  19
-----
          W    R

Consumer reads 19 (buffers occupied: 1)
buffers:  20  18  19
-----
          R    W

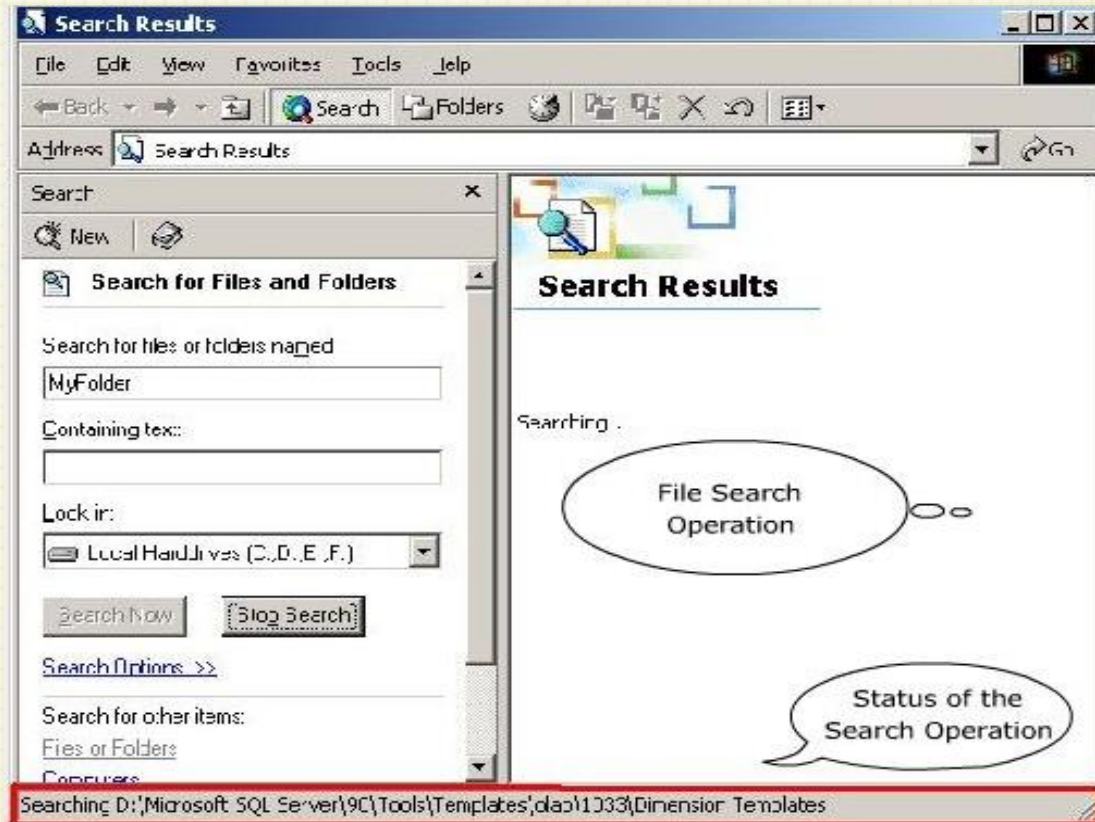
Consumer reads 20 (buffers occupied: 0)
buffers:  20  18  19
-----
          WR

Total Consumer consumed: 155.
Consumer terminated.
```



# Lập trình bất đồng bộ

- **Asynchronous** calls allow improving availability, increasing the performance and scalability of the application.



# Lập trình bất đồng bộ

- Using Delegate:
  - Developers commonly use a delegate to invoke an asynchronous method call in .NET
  - The delegate will perform the act of invoking the method using a thread from a thread pool, thus making it asynchronous.
  - Each delegate implicitly provides a `BeginInvoke()` and `EndInvoke()` method

# Lập trình bất đồng bộ

- Using Delegate:
  - The `BeginInvoke()` method signature typically takes the same parameters as the method specified of the delegate
  - The `EndInvoke()` methods takes an object of `IAsyncResult` interface returned by the `BeginInvoke()` method.
  - There are various approaches to implement asynchronous programming using delegates: Waiting till completion and Polling `IAsyncResult`.



# Waiting till completion

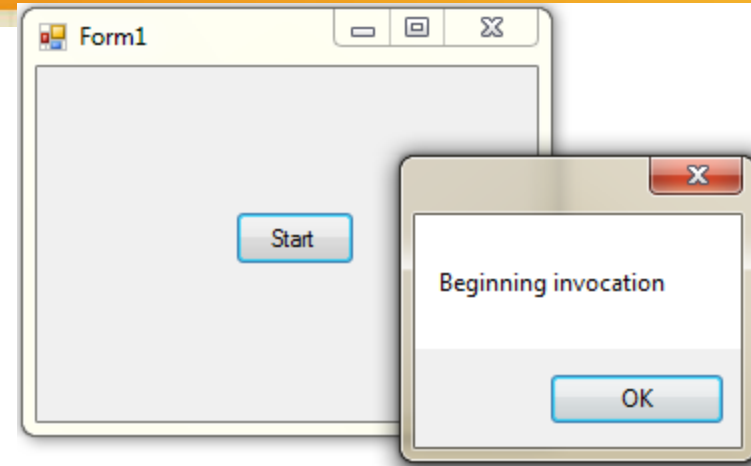
```
delegate void DoActionDelegate();

public void DoAction()
{
    for (int i = 0; i < 5; i++)
        System.Threading.Thread.Sleep(1000);
}

private void Start_Click(object sender, EventArgs e)
{
    DoActionDelegate objdelegate = new DoActionDelegate(DoAction);
    MessageBox.Show("Beginning invocation");
    IAsyncResult objresult = objdelegate.BeginInvoke(null, null);

    MessageBox.Show("Continue doing...");

    objdelegate.EndInvoke(objresult);
    MessageBox.Show("Invocation ending");
}
```





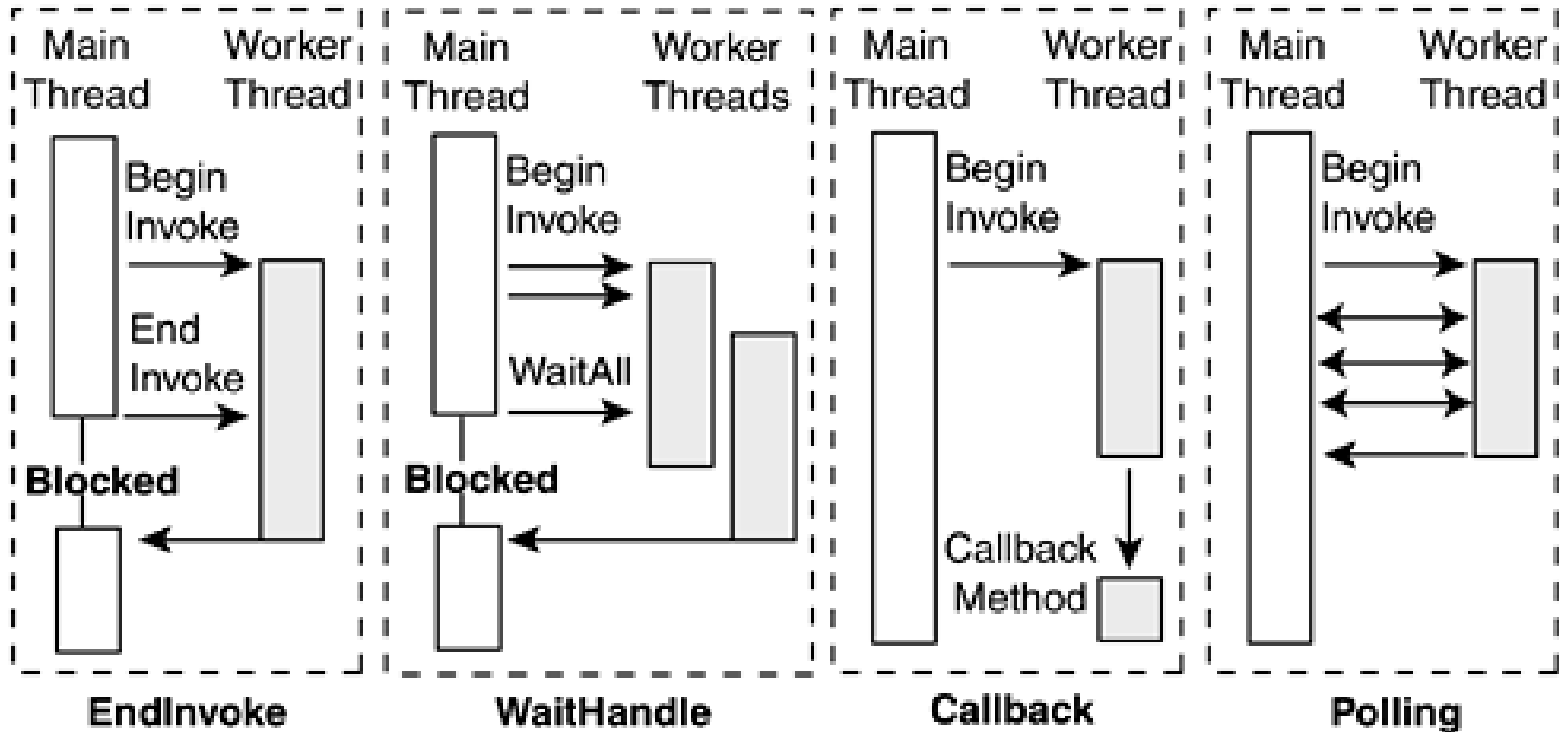
# Polling IAsyncResult until completion

```
delegate void DoActionDelegate();

public void DoAction() {
    for (int i = 0; i < 5; i++)
        System.Threading.Thread.Sleep(1000);
}

private void btnStart_Click(object sender, EventArgs e)
{
    DoActionDelegate objdelegate = new DoActionDelegate(DoAction);
    MessageBox.Show("Beginning Invocation");
    IAsyncResult objResult = objdelegate.BeginInvoke(null, null);
    int count = 0;
    while (!objResult.IsCompleted) {
        //Perform some other operation
        //...
        MessageBox.Show("Perform some other operation");
    }
    objdelegate.EndInvoke(objResult);
    MessageBox.Show("Invocation ending");
}
```

# Options for detecting the completion of an asynchronous task



# Using BackgroundWorker component

- **BackgroundWorker** component provides an easy way to perform time-consuming processes in the background.
- The component is designed in such a manner that time-consuming processes are **run on a separate thread**

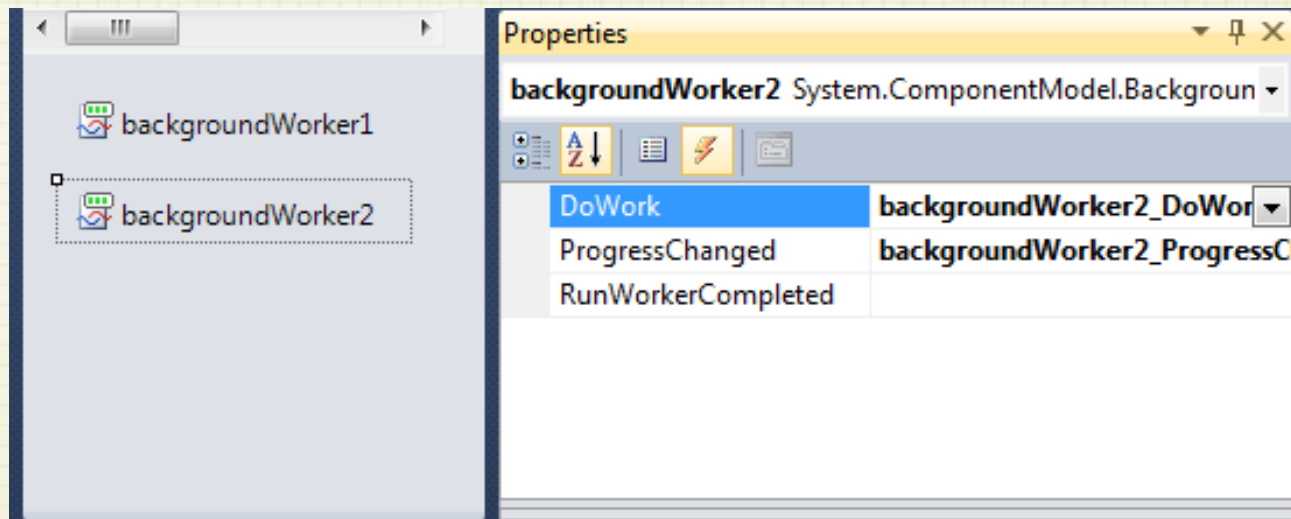
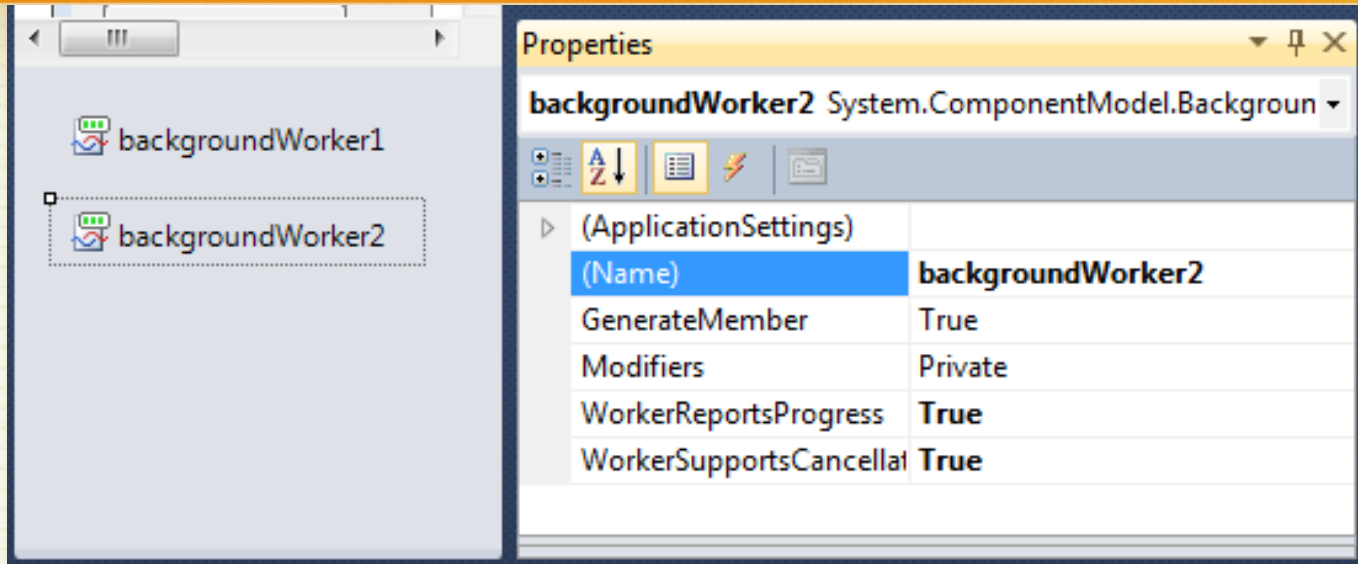
# Using BackgroundWorker component

- Some important members of **BackgroundWorker** class:

Member	Description
CancellationPending	This property specifies whether the application requested for cancellation of the background process.
IsBusy	This property specifies whether the BackgroundWorker is currently running an asynchronous operation.
CancelAsync	This method requests cancellation of pending background operation.
RunWorkerAsync	This method starts the background process by raising the DoWork event.
DoWork	This event executes the code on a separate thread.



# Working with Background Processes



# Working with Background Processes

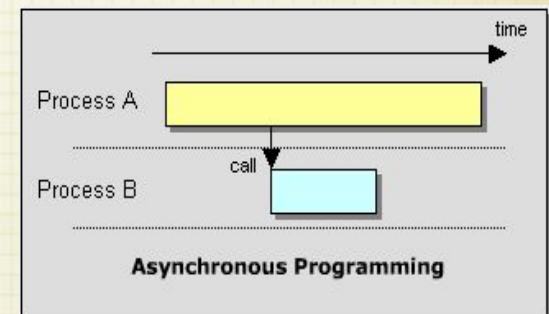
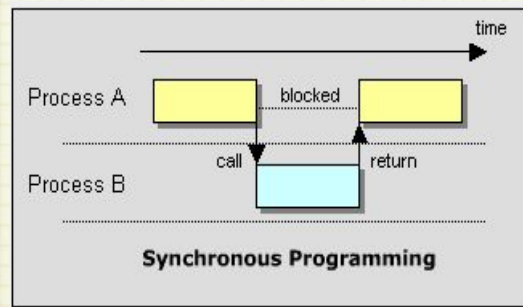
```
private void btnStart_Click(object sender, EventArgs e)
{
    backgroundWorker2.RunWorkerAsync();
}

private void backgroundWorker2_DoWork(object sender, DoWorkEventArgs e)
{
    for (int k = 0; k < 100; k++)
    {
        System.Threading.Thread.Sleep(100);
        backgroundWorker2.ReportProgress(k);
    }
}

private void backgroundWorker2_ProgressChanged(object sender,
    ProgressChangedEventArgs e)
{
    progressBar1.Value = e.ProgressPercentage;
}
```

# Synchronous vs Asynchronous

- In a synchronous application, program execution follows a single path whereas in asynchronous programming, operations occur in parallel on multiple paths of execution.
- Synchronous approach can slow applications and each method is executed in sequence whereas in the asynchronous methods is executed the same time.



# Multi-threading vs Asynchronous

- Running multiple threads in an application can at times hamper the performance of an application as well as scalability.
- The Asynchronous technique works best when the operations of an application or component can be run as independent threads that contain all the data and methods needed for execute



# Multi-threading vs Asynchronous

- Synchronizing the activities in the multiple threads can be a problem in multi-threading, whereas it can easily be achieved by asynchronous programming
- Thread safety cannot be ensured in multi-threading whereas asynchronous programming allows thread safety
- Multi-threading can introduce complexities in the code whereas asynchronous programming provides a simpler approach.

# Q & A

