

# Chương 8

## ĐA TIẾN TRÌNH

# NỘI DUNG CHÍNH

1

Giới thiệu tiến trình

2

Đa tiến trình trên .NET

3

Quản lý tiến trình

4

Đồng bộ hóa

# NỘI DUNG CHÍNH

1

**Giới thiệu tiến trình**

2

Đa tiến trình trên .NET

3

Quản lý tiến trình

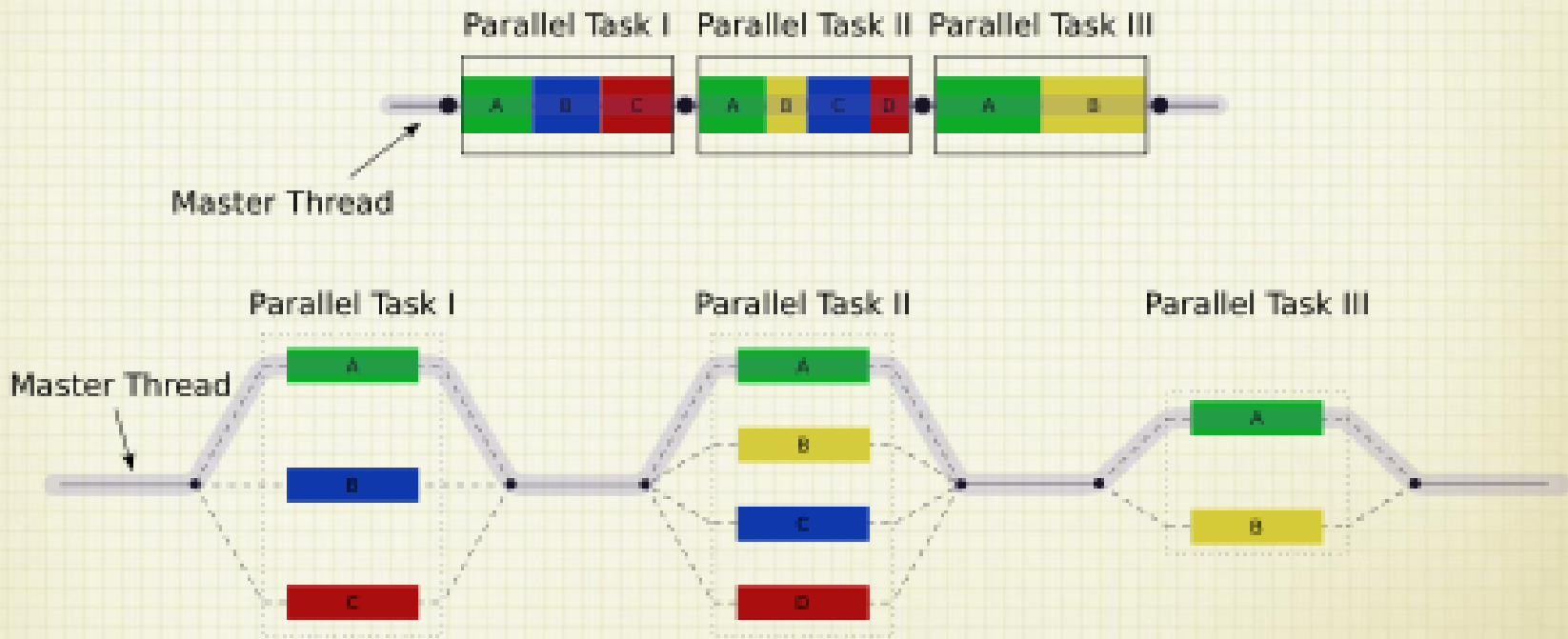
4

Đồng bộ hóa



# Tiến trình

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



# Giới thiệu đa tiến 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.
- Thực sự thì hệ điều hành window làm điều này bằng một thủ tục gọi là **pre emptive multitasking**

# Giới thiệu đa tiến trình

- Window lấy 1 luồng vào trong vài tiến trình và cho phép luồng đó chạy 1 khoảng thời gian ngắn (gọi là **time slice**). Khi thời gian này kết thúc, Window lấy quyền điều khiển lại và lấy 1 luồng khác và lại cấp 1 khoảng thời gian time slice . Vì khoảng thời gian này quá ngắn nên ta có cảm tưởng như mọi thứ đều xảy ra cùng lúc.
- Khi có nhiều cửa sổ trên màn hình, mỗi cửa sổ đại diện cho một tiến trình khác nhau. Người dùng vẫn có thể tương tác với bất kì cửa sổ nào và được đáp ứng ngay lập tức. Nhưng thực sự việc đáp ứng này xảy ra vào sau khoảng thời gian time slice của luồng đang thời.



# Ứng dụng trên Windows

- Đa tiến trình có nhiều lợi ích trong các ứng dụng windows như:
  - Mỗi cửa sổ con trong một ứng dụng MDI có thể được gán cho một tiểu trình khác nhau.
  - Nếu phần đồ họa của chương trình mất nhiều thời gian để thực thi, GUI sẽ được khóa cho đến khi hoàn tất việc vẽ lại. Tuy nhiên, có thể chỉ định một tiểu trình riêng cho hàm OnDraw, như vậy làm cho ứng dụng được phản hồi khi xảy ra tình trạng vẽ quá lâu.
  - Nhiều tiểu trình có thể thực thi đồng thời nếu có nhiều CPU trong hệ thống do đó tăng tốc độ thực hiện của chương trình.
  - Sự mô phỏng phức tạp có thể được thực hiện hiệu quả bằng việc gán một tiểu trình riêng cho mỗi thực thể mô phỏng.
  - Các sự kiện quan trọng có thể được điều khiển hiệu quả thông qua việc phân cho một tiểu trình có độ ưu tiên cao.

# Các trạng thái tiến trình: Chu trình của một tiến trình

- Trạng thái tiến trình:
  - Chưa bắt đầu (Unstarted):
    - Khi một tiến trình được khởi tạo
    - Tiếp tục cho đến khi phương thức Start của tiến trình được gọi
  - Bắt đầu (Started):
    - Duy trì tới lúc bộ xử lý bắt đầu thực hiện nó
  - Đang thực thi (Running):
    - Tiến trình 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 tiến trình
    - **ThreadStart** Tiến trình bắt đầu ủy nhiệm các hành động cụ thể cho các tiến trình
  - Ngừng (Stopped):
    - Khi ủy nhiệm kết thúc
    - Nếu chương trình gọi phương thức Abort của tiến trình



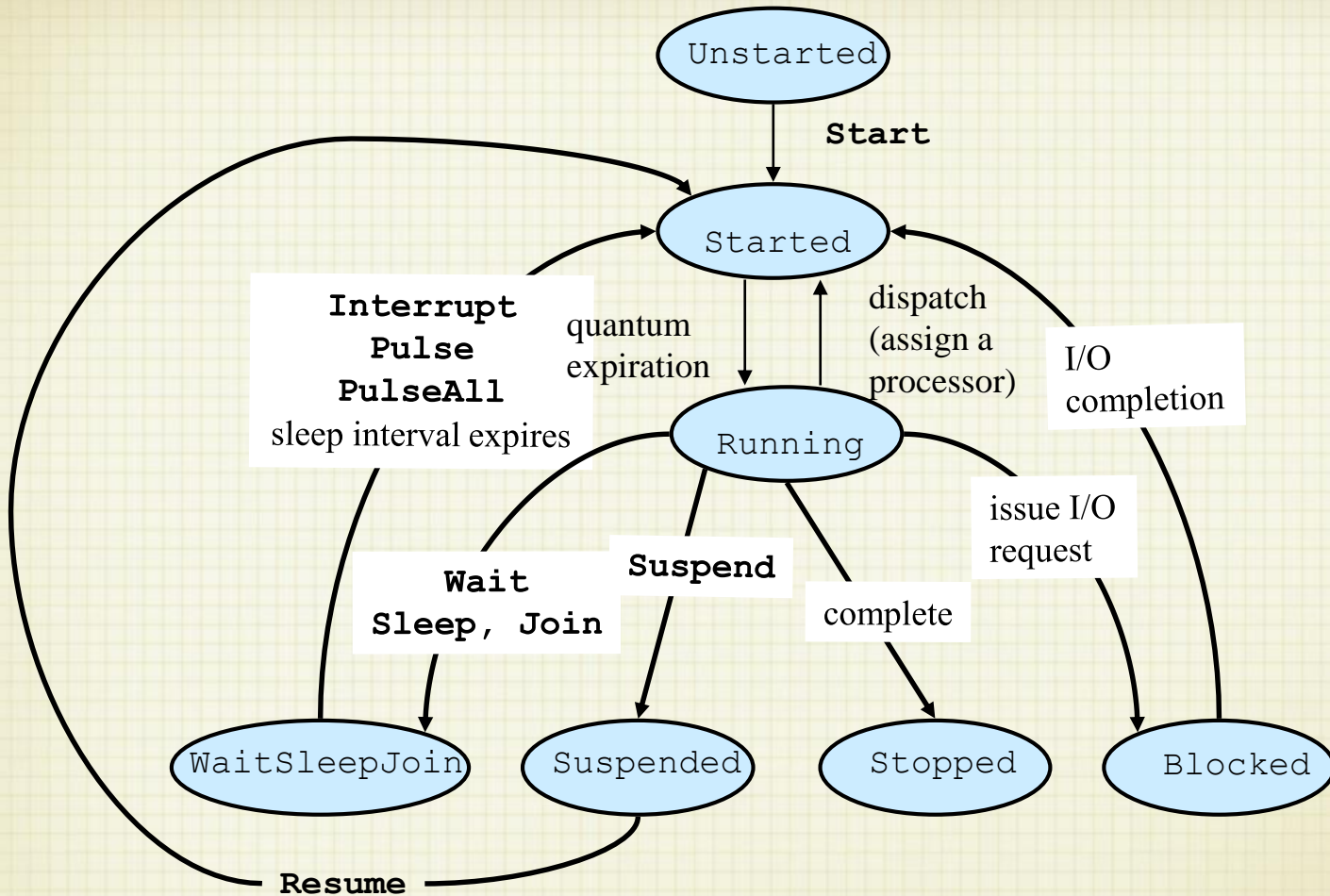
# Các trạng thái tiến trình: Chu trình của một tiến trình

- Trạng thái tiến trình:
  - Blocked:
    - Blocked khi yêu cầu I/O
    - Unblocked khi hệ điều hành hoàn thành I/O
  - WaitSleepJoin:
    - Xảy ra khi:
      - Tiến trình gọi Monitor phương thức Wait vì nó gặp mã mà nó không thực hiện được
        - » Leaves khi một tiến trình khác gọi Pulse
      - Gọi phương thức Sleep để sleep trong một khoảng thời gian
      - Hai tiến trình được kết hợp nếu một tiến trình không thể thực hiện cho đến khi tiến trình kia hoàn thành
    - Các tiến trình đợi (Waiting) hoặc ngủ (Sleeping) có thể ra khỏi trạng thái này nếu phương thức Interrupt của tiến trình được gọi

# Các trạng thái tiến trình: Chu trình của một tiến trình

- Trạng thái tiến trình:
  - 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 tiến trình: Chu trình của một tiến trình



Chu trình của một tiến trình



# NỘI DUNG CHÍNH

1

Giới thiệu tiến trình

2

**Đa tiến trình trên .NET**

3

Quản lý tiến trình

4

Đồng bộ hóa

# Đa tiến 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 tiến trình
  - Đa tiến trình: thực thi các tiến trình đồng thời
  - Tiến trình: phần của một chương trình mà có thể thực thi

# Tạo tiến trình

- Lớp quản lý tiến 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 tiến trình thực thi

```
Thread myThread = new Thread( new ThreadStart(myFunc) );
```

```
myThread.Start(); //Chạy tiến trình
```

- Khi hàm chạy xong, tiến trình sẽ tự động kết thúc và hủy




# Join tiến trình

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

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

Dừng ở đây cho đến khi  
thread thB kết thúc



# Tạm dừng tiến trình

- Tạm dừng tiến 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);
```

- Tham số đưa vào được tính theo ms
- Có thể dùng hàm Sleep để hệ điều hành chuyển quyền điều khiển sang một tiến trình khác

```
Thread.Sleep(1);
```

# Hủy tiến trình

- Tiến trình sẽ kết thúc khi hàm thực thi của nó kết thúc (Đây là cách tự nhiên nhất, tốt nhất)
- Để ép tiến trình kết thúc ngay lập tức có thể sử dụng hàm Interrupt (ThreadInterruptedException được bung ra)
- 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);
}
```



# NỘI DUNG CHÍNH

1

Giới thiệu tiến trình

2

Đa tiến trình trên .NET

3

**Quản lý tiến trình**

4

Đồng bộ hóa

# 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 (true or false)

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

- Tất cả tiến 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)
  - Tất cả tiến trình mặc định là có độ ưu tiên trung bình
  - Sử dụng thuộc tính Priority để thay đổi độ ưu tiên của tiến trình



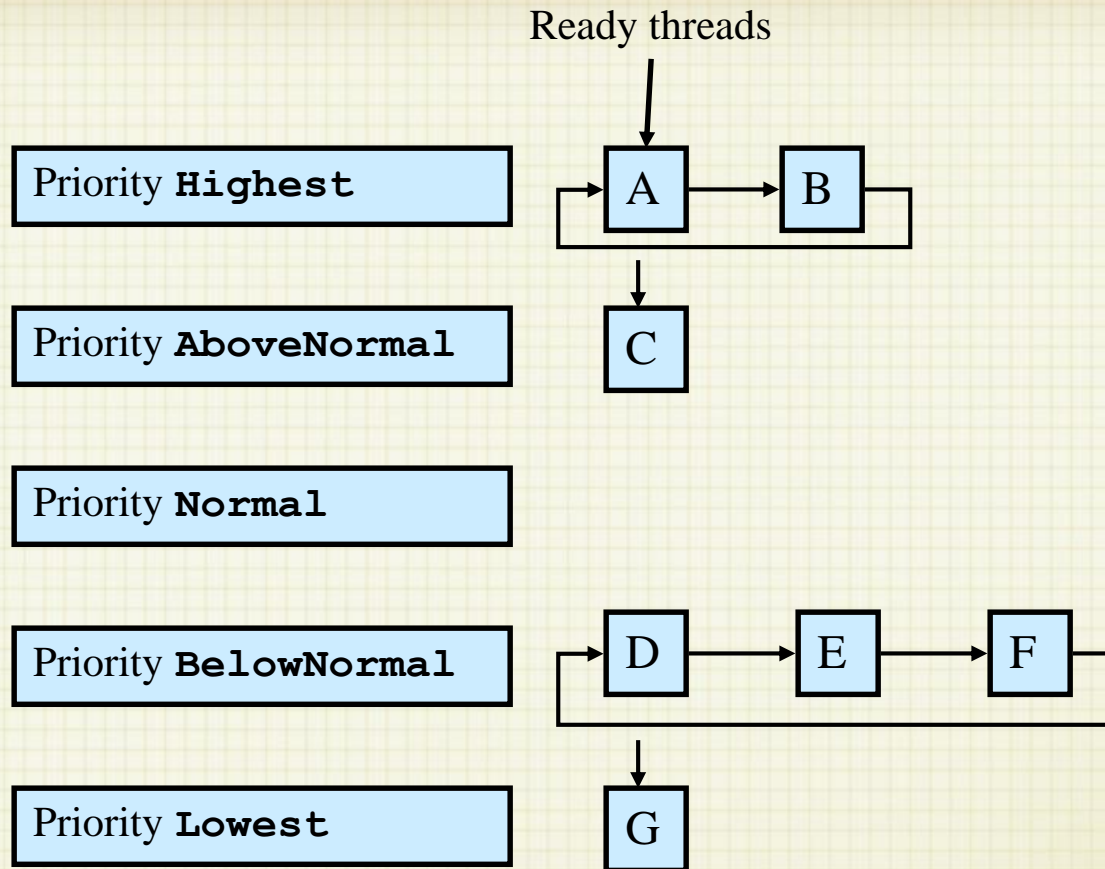
# Độ ưu tiên tiến trình và lập lịch cho tiến trình

- Timeslicing:
  - Mỗi tiến trình được cấp một khoảng thời gian để thực thi trước khi bộ xử lý được giao cho tiến trình khác
  - Nếu không có thì các tiến trình sẽ thực hiện cho đến lúc hoàn thành trước khi tiến trình khác bắt đầu thực thi
- Lưu ý:
  - Mỗi luồng có 1 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 tiến trình có độ ưu tiên cao nhất luôn thực thi tại mọi thời điểm
    - Nếu nhiều tiến trình có cùng độ ưu tiên: 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 tiến trình



```

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

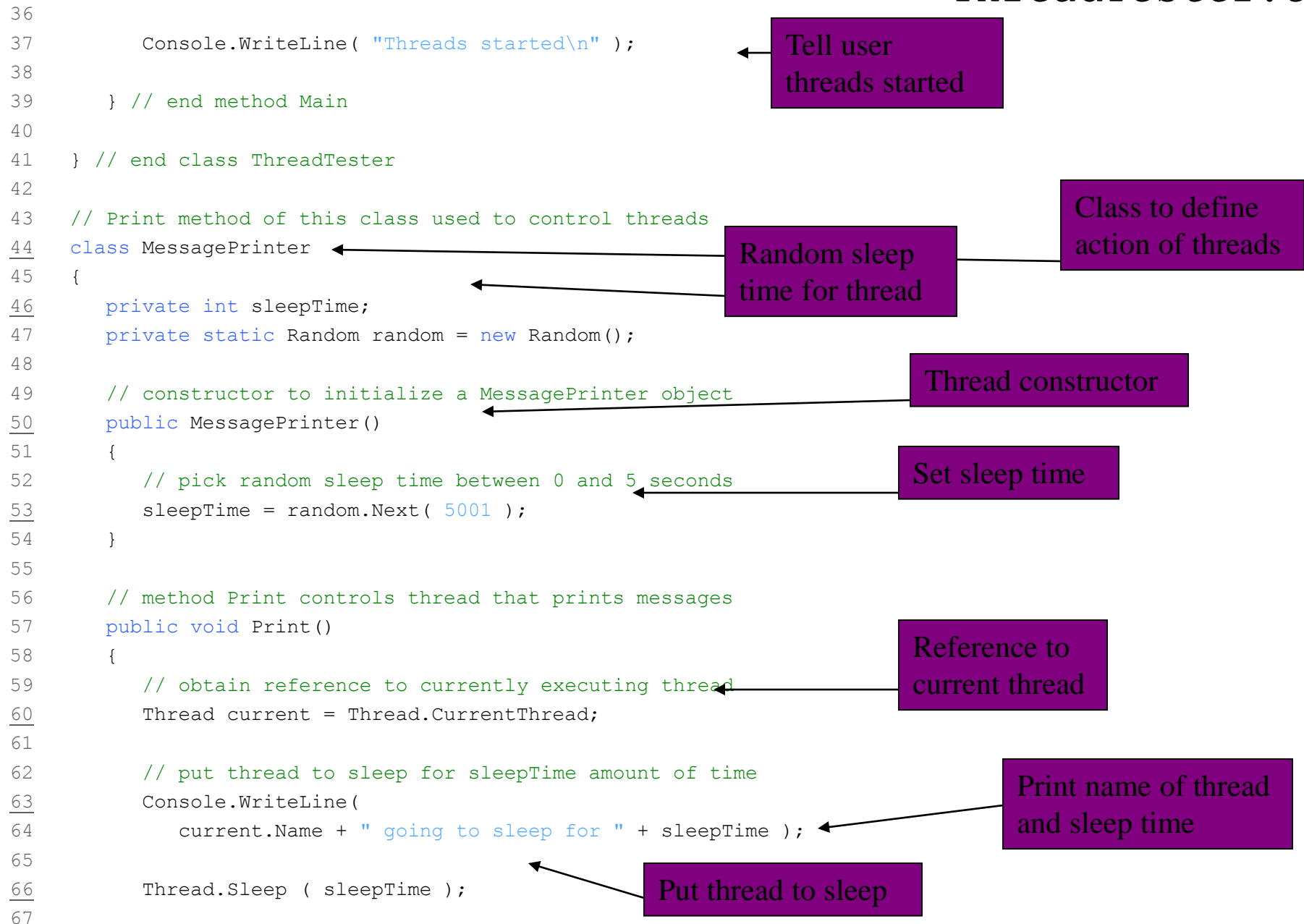
Create MessagePrinter  
objects

Create and initialize threads

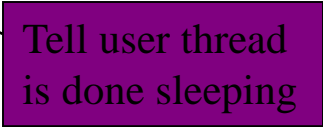
Set thread's name

Thread delegates

Start threads



```
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

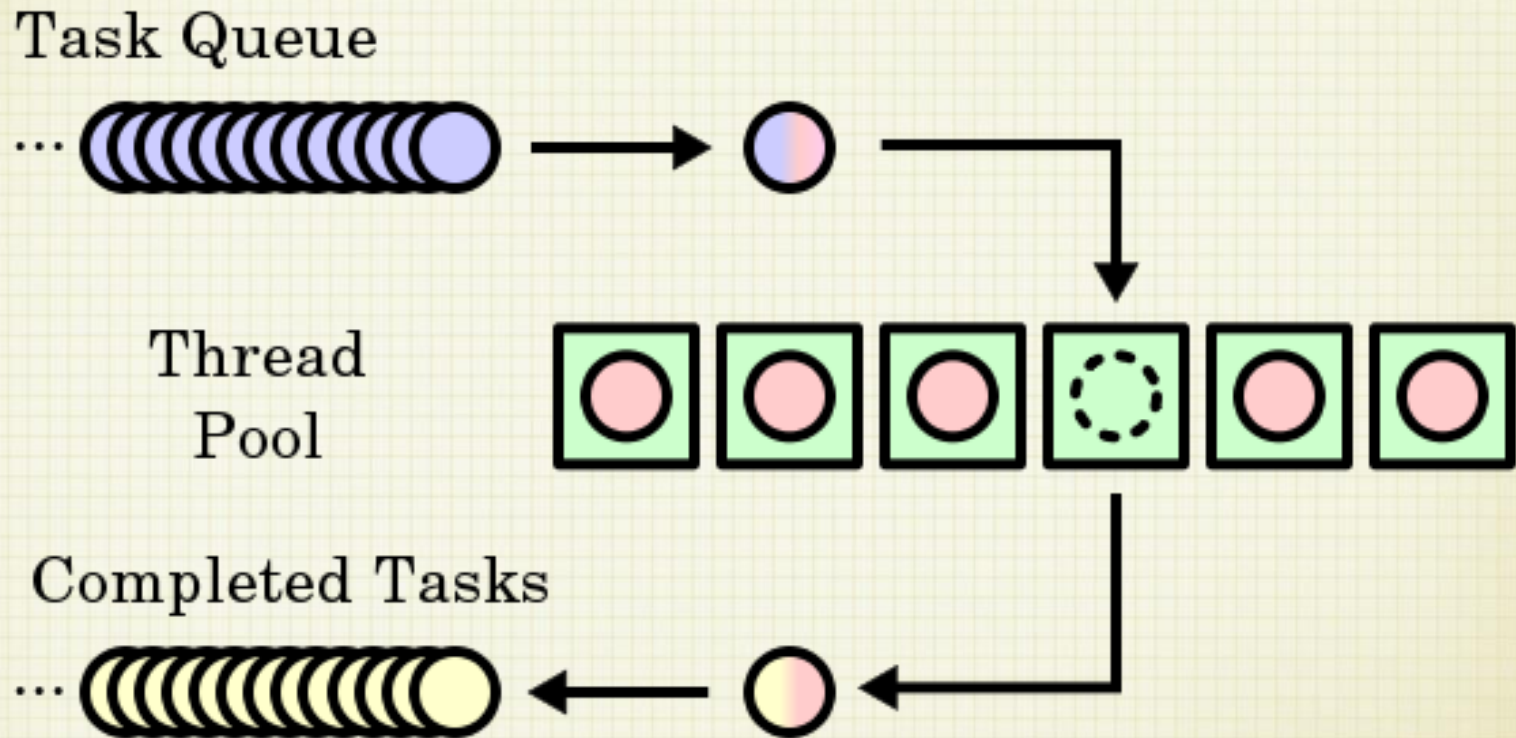
```
thread1 going to sleep for 1466
thread2 going to sleep for 4245
thread3 going to sleep for 1929
thread1 done sleeping
thread3 done sleeping
thread2 done sleeping
```



# ThreadPool

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

# ThreadPool



# 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.
- Bộ thực thi quy định số tiểu 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 tiểu trình cho mỗi CPU trong hệ thống. Số tiểu trình tối đa trong thread-pool không giới hạn số các công việc đang chờ trong 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. Mặc dù độ dài tối đa của hàng đợi chỉ bị giới hạn bởi số lượng bộ nhớ còn lại cho tiến trình của bộ thực thi, nhưng hàng đợi quá dài sẽ làm kéo dài quá trình thực thi của các công việc trong hàng đợi.

# ThreadPool

- Không nên sử dụng thread-pool để thực thi các tiến trình chạy trong một thời gian dài. Vì số tiến trình trong thread-pool là có giới hạn, nên chỉ một số ít tiến trình thuộc các tiến trình loại này cũng sẽ ảnh hưởng đáng kể đến toàn bộ hiệu năng của thread-pool. Nên tránh đặt các tiến trình trong thread-pool vào trạng thái đợi trong một thời gian quá dài.
- Không thể điều khiển lịch trình của các tiến trình trong thread-pool, cũng như không thể thay đổi độ ưu tiên của các công việc. Thread-pool xử lý các công việc theo thứ tự như khi thêm chúng vào hàng đợi.
- Một khi công việc đã được đặt vào hàng đợi thì không thể hủy hay dừng

```

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. " );
    }
}

```



# NỘI DUNG CHÍNH

1

Giới thiệu tiến trình

2

Đa tiến trình trên .NET

3

Quản lý tiến trình

4

**Đồng bộ hóa**

# Đồng bộ hóa (Synchronization)

- Khi bạn cần bảo vệ một tài nguyên, trong một thời điểm chỉ cho phép một thread thay đổi hoặc sử dụng tài nguyên đó, bạn 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

# Blocking

- Chờ một thread khác kết thúc hoặc một khoảng thời gian nhất định trôi qua
  - Sleep
  - Join
  - Task.Wait



# 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

# 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
- Các loại signaling:
  - Wait/Pulse của Monitor
  - CountdownEvent
  - Barrier

# Nonblocking

- Bảo vệ sự truy cập vào những tài nguyên chung bằng cách gọi các processor primitive
- Các lớp Nonbloking trong .NET:
  - `Thread.MemoryBarrier`
  - `Thread.VolatileRead`
  - `Thread.VolatileWrite`
  - keyword `volatile`
  - `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( )
{
    try
    {
        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);
        }
    }
}
```

# Interlocked

- CLR cung cấp một lớp đặc biệt **Interlocked** nhằm đáp ứng nhu cầu tăng giảm giá trị. Interlocked có 2 phương thức **Increment()** và **Decrement()** nhằm tăng và giảm giá trị **trong sự bảo vệ** của cơ chế đồng bộ.

# Interlocked

```
public void Incrementer( )
{
    try
    {
        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);
        }
    }
}
```

Khởi catch và finally không thay đổi so với ví dụ trước.



# Locks

- Lock đánh dấu một đoạn mã then chốt (critical section) trong chương trình của bạn, 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 (keyword) **lock**. Lock được gỡ bỏ khi hết khối lệnh.
- Lock tương đương với 1 cặp Monitor.Enter/Monitor.Exit
- 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 1 khoảng thời gian
  - Nếu không bị khóa thì vào thực thi đoạn mã bên trong, đồng thời khóa tài nguyên lại
  - Sau khi thoát khỏi đoạn mã thì mở khóa cho tài nguyên

# Locks

```
public void Incrementer( )
```

```
{
```

```
try
```

```
{
```

```
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);
```

```
}
```

```
}
```

Khởi catch và finally không thay đổi so với ví dụ trước.

Tài nguyên được khóa

Khởi mã được khóa

# Monitor

- Để có thể đồng bộ hóa phức tạp hơn cho tài nguyên, ta cần sử dụng monitor. Một 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 đổi tượng cần đồng bộ cho hàm sau:

**Monitor.Enter(đối tượng X);**

- Nếu monitor không sẵn dùng (unavailable), đối tượng bảo vệ bởi monitor đang được sử dụng. Ta có thể làm việc khác trong khi chờ đợi monitor sẵn dùng (available) hoặc treo thread lại cho đến khi có monitor (bằng cách gọi hàm Wait())



# Monitor

- Lời gọi `Wait()` giải phóng monitor nhưng bạn đã báo cho CLR biết là bạn 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);`

- `Pulse()` báo hiệu cho CLR rằng có sự thay đổi trong trạng thái monitor có thể dẫn đến việc giải phóng (tiếp tục chạy) một thread đang trong tình trạng chờ đợi. Khi thread hoàn tất việc sử dụng monitor, nó gọi hàm **Exit()** để trả monitor.

`Monitor.Exit(this);`

- Ưu điểm: Thread chờ không cần phải kiểm tra monitor khóa theo từng khoảng thời gian

# Monitor

- Ví dụ bạn 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.
- Để giả lập việc này, bạn 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 bạn gọi phương thức **Enter()**, sau đó kiểm tra giá trị counter, nếu  $< 5$  thì gọi hàm **Wait()**

```
if (counter < 5)
{
    Monitor.Wait(this);
}
```

# Monitor

Source code 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

```
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

```
// 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

```
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

```
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

```
        // 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

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.

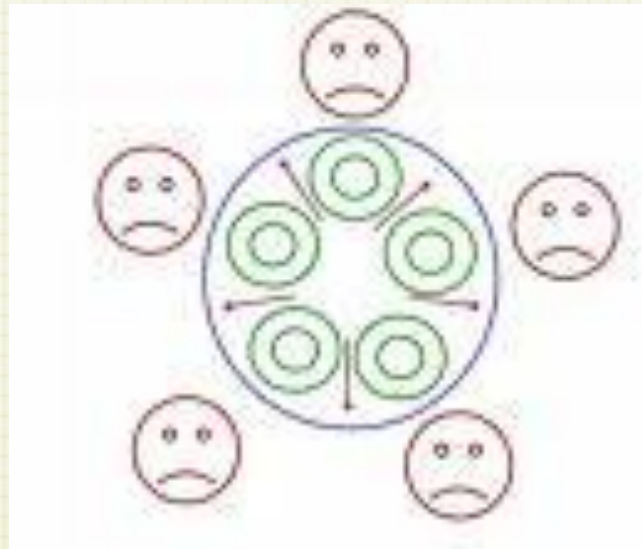
[Thread1] In Decrementer. Counter: 0.

All my threads are done.



# Race condition và DeadLock

- Đồng bộ hóa thread khá rắc rối trong những chương trình phức tạp. Bạn 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.

Ví dụ: giả sử bạn có 2 thread. Thread 1 tiến hành mở tập tin, thread 2 tiến hành ghi lên cùng tập tin đó. Điều quan trọng là bạn 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. Để giải quyết vấn đề trong ví dụ trên, bạn có thể tiến hành join thread 2 với thread 1 hoặc thiết lập monitor.

# Deadlock

- Giả sử thread A đã nắm monitor của tài nguyên X và đang chờ monitor của tài nguyên Y. Trong khi đó thì thread B lại nắm monitor của tài nguyên Y và chờ monitor của tài nguyên X. 2 thread cứ chờ đợi lẫn nhau mà không thread nào có thể thoát ra khỏi tình trạng chờ đợi. Tình trạng trên gọi là 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.



# Quan hệ sản xuất/ tiêu thụ không dùng đồng bộ hóa tiến trình

- 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 nó
  - Tiêu thụ đọc dữ liệu sai hoặc là hai lần dữ liệu như nhau

```
1 // Fig. 14.4: Unsyncronized.cs
2 // Showing multiple threads modifying a shared object without
3 // synchronization.
4
5 using System;
6 using System.Threading;
7
8 // this class represents a single shared int
9 public class HoldIntegerUnsyncronized
10 {
11     // buffer shared by producer and consumer threads
12     private int buffer = -1;
13
14     // property Buffer
15     public int Buffer
16     {
17         get
18         {
19             Console.WriteLine( Thread.CurrentThread.Name +
20                 " reads " + buffer );
21
22             return buffer;
23         }
24         set
25         {
26             Console.WriteLine( Thread.CurrentThread.Name +
27                 " writes " + value );
28
29             buffer = value;
30         }
31     } // end property Buffer
32
33 } // end class HoldIntegerUnsyncronized
```

Buffer class

Integer shared by consumer  
and producer (buffer)

Accessor to read  
buffer

Accessor to write  
to buffer

```

36
37 // class Producer's Produce method controls a thread that
38 // stores values from 1 to 4 in sharedLocation
39 class Producer
40 {
41     private HoldIntegerUnsyncronized sharedLocation;
42     private Random randomSleepTime;
43
44     // constructor
45     public Producer(
46         HoldIntegerUnsyncronized shared, Random random )
47     {
48         sharedLocation = shared;
49         randomSleepTime = random;
50     }
51
52     // store values 1-4 in object sharedLocation
53     public void Produce()
54     {
55         // sleep for random interval upto 3000 milliseconds
56         // then set sharedLocation's Buffer property
57         for ( int count = 1; count <= 4; count++ )
58         {
59             Thread.Sleep( randomSleepTime.Next( 1, 3000 ) );
60             sharedLocation.Buffer = count;
61         }
62
63         Console.WriteLine( Thread.CurrentThread.Name +
64             " done producing.\nTerminating " +
65             Thread.CurrentThread.Name + "." );
66
67     } // end method Produce
68
69 } // end class Producer
70

```

Producer class

Set buffer as shared object

Set sleep time

Cycles 4 times

Put buffer to sleep

Set buffer to count

Tell user thread is  
done producing



```

71 // class Consumer's Consume method controls a thread that
72 // loops four times and reads a value from sharedLocation
73 class Consumer
74 {
75     private HoldIntegerUnsyncronized sharedLocation;
76     private Random randomSleepTime;
77
78     // constructor
79     public Consumer(
80         HoldIntegerUnsyncronized shared, Random random )
81     {
82         sharedLocation = shared;
83         randomSleepTime = random;
84     }
85
86     // read sharedLocation's value four times
87     public void Consume()
88     {
89         int sum = 0;
90
91         // sleep for random interval up to 3000 milliseconds
92         // then add sharedLocation's Buffer property value
93         // to sum
94         for ( int count = 1; count <= 4; count++ )
95         {
96             Thread.Sleep( randomSleepTime.Next( 1, 3000 ) );
97             sum += sharedLocation.Buffer;
98         }
99

```

Consumer Class

Set shared to buffer

Set sleep time

Set sum to 0

Loop 4 times

Put thread to sleep

Add value in  
buffer to sum

```

100     Console.WriteLine( Thread.CurrentThread.Name +
101         " read values totaling: " + sum +
102         ".\nTerminating " + Thread.CurrentThread.Name + "." );
103
104     } // end method Consume
105
106 } // end class Consumer
107
108 // this class creates producer and consumer threads
109 class SharedCell
110 {
111     // create producer and consumer threads and start them
112     static void Main( string[] args )
113     {
114         // create shared object used by threads
115         HoldIntegerUnsyncronized holdInteger =
116             new HoldIntegerUnsyncronized();
117
118         // Random object used by each thread
119         Random random = new Random();
120
121         // create Producer and Consumer objects
122         Producer producer =
123             new Producer( holdInteger, random );
124
125         Consumer consumer =
126             new Consumer( holdInteger, random );
127
128         // create threads for producer and consumer and set
129         // delegates for each thread
130         Thread producerThread =
131             new Thread( new ThreadStart( producer.Produce ) );
132         producerThread.Name = "Producer";
133

```

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

```

134 Thread consumerThread =
135     new Thread( new ThreadStart( consumer.Consume ) );
136 consumerThread.Name = "Consumer";
137
138 // start each thread
139 producerThread.Start();
140 consumerThread.Start();
141
142 } // end method Main
143
144 } // 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.
```

# Quan hệ sản xuất/ tiêu thụ sử dụng đồng bộ hóa tiến trình

- 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
3  // synchronization.
4
5  using System;
6  using System.Threading;
7
8  // this class synchronizes access to an integer
9  public class HoldIntegerSynchronized
10 {
11     // buffer shared by producer and consumer threads
12     private int buffer = -1;
13
14     // occupiedBufferCount maintains count of occupied buffers
15     private int occupiedBufferCount = 0;
16
17     // property Buffer
18     public int Buffer
19     {
20         get
21         {
22             // obtain lock on this object
23             Monitor.Enter( this );
24
25             // if there is no data to read, place invoking
26             // thread in WaitSleepJoin state
27             if ( occupiedBufferCount == 0 )
28             {
29                 Console.WriteLine(
30                     Thread.CurrentThread.Name + " tries to read." );
31
32                 DisplayState( "Buffer empty. " +
33                     Thread.CurrentThread.Name + " waits." );
34

```

Create buffer

Variable to determine  
whose turn to use bufferMethod to get value  
from buffer

Get lock

See if buffer is occupied



```
35     Monitor.Wait( this );
36 }
37
38 // indicate that producer can store another value
39 // because a consumer just retrieved buffer value
40 --occupiedBufferCount;
41
42 DisplayState(
43     Thread.CurrentThread.Name + " reads "
44
45     // tell waiting thread (if there is one) to
46     // become ready to execute (Started state)
47     Monitor.Pulse( this );
48
49     // Get copy of buffer before releasing lock.
50     // It is possible that the producer could be
51     // assigned the processor immediately after the
52     // monitor is released and before the return
53     // statement executes. In this case, the producer
54     // would assign a new value to buffer before the
55     // return statement returns the value to the
56     // consumer. Thus, the consumer would receive the
57     // new value. Making a copy of buffer and
58     // returning the copy ensures that the
59     // consumer receives the proper value.
60     int bufferCopy = buffer;
61
62     // release lock on this object
63     Monitor.Exit( this );
64
65     return bufferCopy;
66 } // end get
67
68
```

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

```

69  set
70  {
71      // acquire lock for this object
72      Monitor.Enter( this );
73
74      // if there are no empty locations, place invoking
75      // thread in WaitSleepJoin state
76      if ( occupiedBufferCount == 1 )
77      {
78          Console.WriteLine(
79              Thread.CurrentThread.Name + " tries to write." );
80
81          DisplayState( "Buffer full. " +
82              Thread.CurrentThread.Name + " waits." );
83
84          Monitor.Wait( this );
85      }
86
87      // set new buffer value
88      buffer = value;
89
90      // indicate producer cannot store another value
91      // until consumer retrieves current buffer value
92      ++occupiedBufferCount;
93
94      DisplayState(
95          Thread.CurrentThread.Name + " writes " + buffer );
96
97      // tell waiting thread (if there is one) to
98      // become ready to execute (Started state)
99      Monitor.Pulse( this );
100

```

Method to write to buffer

Get lock

Test if buffer is occupied

If buffer occupied, put producer to sleep

Write to buffer

Tell system buffer has been written to

Release consumer from wait state

```
101         // release lock on this object
102         Monitor.Exit( this );
103
104     } // end set
105
106 }
107
108 // display current operation and buffer state
109 public void DisplayState( string operation )
110 {
111     Console.WriteLine( "{0,-35}{1,-9}{2}\n",
112         operation, buffer, occupiedBufferCount );
113 }
114
115 } // end class HoldIntegerSynchronized
```

Release lock

```
116
117 // class Producer's Produce method controls a thread that
118 // stores values from 1 to 4 in sharedLocation
```

```
119 class Producer
120 {
121     private HoldIntegerSynchronized sharedLocation;
122     private Random randomSleepTime;
123
124     // constructor
125     public Producer(
126         HoldIntegerSynchronized shared, Random random )
127     {
128         sharedLocation = shared;
129         randomSleepTime = random;
130     }
131 }
```

Producer Class

Set sharedLocation to buffer

Set sleep time



```

132 // store values 1-4 in object sharedLocation
133 public void Produce ()
134 {
135     // sleep for random interval up to 3000 milliseconds
136     // then set sharedLocation's Buffer property
137     for ( int count = 1; count <= 4; count++ )
138     {
139         Thread.Sleep( randomSleepTime.Next( 1, 3000 ) );
140         sharedLocation.Buffer = count;
141     }
142
143     Console.WriteLine( Thread.CurrentThread.Name
144         " done producing.\nTerminating " +
145         Thread.CurrentThread.Name + ".\n" );
146
147 } // end method Produce
148
149 } // end class Producer
150
151 // class Consumer's Consume method controls a thread that
152 // loops four times and reads a value from sharedLocation
153 class Consumer
154 {
155     private HoldIntegerSynchronized sharedLocation;
156     private Random randomSleepTime;
157
158     // constructor
159     public Consumer(
160         HoldIntegerSynchronized shared, Random random )
161     {
162         sharedLocation = shared;
163         randomSleepTime = random;
164     }
165

```

Loop 4 times

Put thread to sleep

Set buffer equal to count

Tell user thread is done

Consumer class

Set sharedLocation to buffer

Set sleep time

```

166 // read sharedLocation's value four times
167 public void Consume ()
168 {
169     int sum = 0;
170
171     // get current thread
172     Thread current = Thread.CurrentThread;
173
174     // sleep for random interval up to 3000 milliseconds
175     // then add sharedLocation's Buffer property value
176     // to sum
177     for ( int count = 1; count <= 4; count++ )
178     {
179         Thread.Sleep( randomSleepTime.Next( 1, 3000 ) );
180         sum += sharedLocation.Buffer;
181     }
182
183     Console.WriteLine( Thread.CurrentThread.Name +
184         " read values totaling: " + sum +
185         ".\nTerminating " + Thread.CurrentThread.Name + ".\n" );
186
187 } // end method Consume
188
189 } // end class Consumer
190
191 // this class creates producer and consumer threads
192 class SharedCell
193 {
194     // create producer and consumer threads and start them
195     static void Main( string[] args )
196     {
197         // create shared object used by threads
198         HoldIntegerSynchronized holdInteger =
199             new HoldIntegerSynchronized();
200

```

Loop 4 times

Put thread to sleep

Add buffer to sum

Tell user thread is  
finished and sum

Create buffer

```

201 // Random object used by each thread
202 Random random = new Random();
203
204 // create Producer and Consumer objects
205 Producer producer =
206     new Producer( holdInteger, random );
207
208 Consumer consumer =
209     new Consumer( holdInteger, random );
210
211 // output column heads and initial buffer state
212 Console.WriteLine( "{0,-35}{1,-9}{2}\n",
213     "Operation", "Buffer", "Occupied Count" );
214 holdInteger.DisplayState( "Initial state" );
215
216 // create threads for producer and consumer and set
217 // delegates for each thread
218 Thread producerThread =
219     new Thread( new ThreadStart( producer.Produce ) );
220 producerThread.Name = "Producer";
221
222 Thread consumerThread =
223     new Thread( new ThreadStart( consumer.Consume ) );
224 consumerThread.Name = "Consumer";
225
226 // start each thread
227 producerThread.Start();
228 consumerThread.Start();
229
230 } // end method Main
231
232 } // 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



## 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.		

# Synchronized.cs

## 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

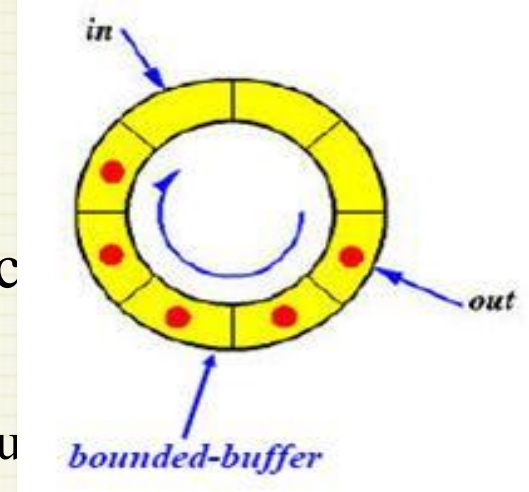
## Program Output

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
3  // circular buffer.
4
5  using System;
6  using System.Drawing;
7  using System.Collections;
8  using System.ComponentModel;
9  using System.Windows.Forms;
10 using System.Data;
11 using System.Threading;
12
13 // implement the shared integer with synchronization
14 public class HoldIntegerSynchronized
15 {
16     // each array element is a buffer
17     private int[] buffers = { -1, -1, -1 };
18
19     // occupiedBufferCount maintains count of occupied buffers
20     private int occupiedBufferCount = 0;
21
22     // variable that maintain read and write buffer locations
23     private int readLocation = 0, writeLocation = 0;
24
25     // GUI component to display output
26     private TextBox outputTextBox;
27
28     // constructor
29     public HoldIntegerSynchronized( TextBox output )
30     {
31         outputTextBox = output;
32     }
33

```

Circular buffer

How many buffers  
are occupied

Next read location

Next write location

Create textbox

```

34 // property Buffer
35 public int Buffer
36 {
37     get
38     {
39         // lock this object while getting value
40         // from buffers array
41         lock ( this )
42         {
43             // if there is no data to read, place invoking
44             // thread in WaitSleepJoin state
45             if ( occupiedBufferCount == 0 )
46             {
47                 outputTextBox.Text += "\r\nAll buffers empty. " +
48                     Thread.CurrentThread.Name + " waits.";
49                 outputTextBox.ScrollToCaret();
50
51                 Monitor.Wait( this );
52             }
53
54             // obtain value at current readLocation, then
55             // add string indicating consumed value to output
56             int readValue = buffers[ readLocation ];
57
58             outputTextBox.Text += "\r\n" +
59                 Thread.CurrentThread.Name + " reads " +
60                 buffers[ readLocation ] + " ";
61
62             // just consumed a value, so decrement number of
63             // occupied buffers
64             --occupiedBufferCount;
65

```

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



# CircularBuffer.cs

```
66 // update readLocation for future read operation,
67 // then add current state to output
68 readLocation =
69     ( readLocation + 1 ) % buffers.Length;
70 outputTextBox.Text += CreateStateOutput();
71 outputTextBox.ScrollToCaret();
72
73 // return waiting thread (if there is one)
74 // to Started state
75 Monitor.Pulse( this );
76
77 return readValue;
78
79 } // end lock
80
81 } // end accessor get
82
83 set
84 {
85     // lock this object while setting value
86     // in buffers array
87     lock ( this )
88     {
89         // if there are no empty locations, place invoking
90         // thread in WaitSleepJoin state
91         if ( occupiedBufferCount == buffers.Length )
92         {
93             outputTextBox.Text += "\r\nAll buffers full. " +
94                 Thread.CurrentThread.Name + " waits.";
95             outputTextBox.ScrollToCaret();
96
97             Monitor.Wait( this );
98         }
99     }
```

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

```

100 // place value in writeLocation of buffers, then
101 // add string indicating produced value to output
102 buffers[ writeLocation ] = value;

```

Put new value in next location of buffer

```

104 outputTextBox.Text += "\r\n" +
105     Thread.CurrentThread.Name + " writes " +
106     buffers[ writeLocation ] + " ";

```

Output value written to buffer

```

108 // just produced a value, so increment number of
109 // occupied buffers
110 ++occupiedBufferCount;

```

Increment number of buffers occupied

```

112 // update writeLocation for future write operation
113 // then add current state to output

```

```

114 writeLocation =
115     ( writeLocation + 1 ) % buffers.Length;
116 outputTextBox.Text += CreateStateOutput();
117 outputTextBox.ScrollToCaret();

```

Update write location

Call CreateStateOutput

```

119 // return waiting thread (if there is one)
120 // to Started state
121 Monitor.Pulse( this );

```

Get consumer from wait state

```

122 } // end lock

```

```

125 } // end accessor set

```

```

127 } // end property Buffer

```

```

129 // create state output

```

```

130 public string CreateStateOutput()

```

```

131 {
132     // display first line of state information
133     string output = "(buffers occupied: " +
134         occupiedBufferCount + ")\r\nbuffers: ";

```

Output number of buffers occupied

```
135
136 for ( int i = 0; i < buffers.Length; i++ )
137     output += " " + buffers[ i ] + " ";
138
139 output += "\r\n";
140
141 // display second line of state information
142 output += "          ";
143
144 for ( int i = 0; i < buffers.Length; i++ )
145     output += "---- ";
146
147 output += "\r\n";
148
149 // display third line of state information
150 output += "          ";
151
152 // display readLocation (R) and writeLocation (W)
153 // indicators below appropriate buffer locations
154 for ( int i = 0; i < buffers.Length; i++ )
155
156     if ( i == writeLocation &&
157         writeLocation == readLocation )
158         output += " WR ";
159     else if ( i == writeLocation )
160         output += " W  ";
161     else if ( i == readLocation )
162         output += "  R ";
163     else
164         output += "    ";
165
```

Output contents of buffers



Output readLocation  
and writeLocation





```

166         output += "\r\n";
167
168         return output;
169     }
170
171 } // end class HoldIntegerSynchronized
172
173 // produce the integers from 11 to 20 and place them in buffer
174 public class Producer
175 {
176     private HoldIntegerSynchronized sharedLocation;
177     private TextBox outputTextBox;
178     private Random randomSleepTime;
179
180     // constructor
181     public Producer( HoldIntegerSynchronized shared,
182                     Random random, TextBox output )
183     {
184         sharedLocation = shared;
185         outputTextBox = output;
186         randomSleepTime = random;
187     }
188
189     // produce values from 11-20 and place them in
190     // sharedLocation's buffer
191     public void Produce()
192     {
193         // sleep for random interval up to 3000 milliseconds
194         // then set sharedLocation's Buffer property
195         for ( int count = 11; count <= 20; count++ )
196         {
197             Thread.Sleep( randomSleepTime.Next( 1, 3000 ) );
198             sharedLocation.Buffer = count;
199         }
200

```

Producer class

Set shared location to buffer

Set output

Set sleep time

Loop ten times

Set sleep time

Write to buffer

# CircularBuffer.cs

```
201     string name = Thread.CurrentThread.Name;
202
203     outputTextBox.Text += "\r\n" + name +
204         " done producing.\r\n" + name + " terminated.\r\n";
205
206     outputTextBox.ScrollToCaret();
207
208 } // end method Produce
```

Output to textbox

```
209
210 } // end class Producer
211
212 // consume the integers 1 to 10 from circular buffer
```

```
213 public class Consumer
```

Consumer class

```
214 {
215     private HoldIntegerSynchronized sharedLocation;
216     private TextBox outputTextBox;
217     private Random randomSleepTime;
```

```
218
219 // constructor
```

```
220 public Consumer( HoldIntegerSynchronized shared,
221     Random random, TextBox output )
```

```
222 {
223     sharedLocation = shared;
224     outputTextBox = output;
225     randomSleepTime = random;
226 }
```

Set shared location  
to buffer

Set output

Set sleep time

```
227
228 // consume 10 integers from buffer
```

```
229 public void Consume ()
```

```
230 {
231     int sum = 0;
```

Initialize sum to 0

```
232
```

# CircularBuffer.cs

```
233 // loop 10 times and sleep for random interval up to
234 // 3000 milliseconds then add sharedLocation's
235 // Buffer property value to sum
236 for ( int count = 1; count <= 10; count++ )
237 {
238     Thread.Sleep( randomSleepTime.Next( 1, 3000 ) );
239     sum += sharedLocation.Buffer;
240 }
241
242 string name = Thread.CurrentThread.Name;
243
244 outputTextBox.Text += "\r\nTotal " + name +
245     " consumed: " + sum + ".\r\n" + name +
246     " terminated.\r\n";
247
248 outputTextBox.ScrollToCaret();
249
250 } // end method Consume
251
252 } // end class Consumer
253
254 // set up the producer and consumer and start them
255 public class CircularBuffer : System.Windows.Forms.Form
256 {
257     private System.Windows.Forms.TextBox outputTextBox;
258
259     // required designer variable
260     private System.ComponentModel.Container components = null;
261
262     // no-argument constructor
263     public CircularBuffer()
264     {
265         InitializeComponent();
266     }
267
```

Loop ten times

Put thread to sleep

Add value of  
buffer to sum

Output to textbox

```
268 // Visual Studio .NET GUI code appears here in source file
269
270 // main entry point for the application
271 [STAThread]
272 static void Main()
273 {
274     Application.Run( new CircularBuffer() );
275 }
276
277 // Load event handler creates and starts threads
278 private void CircularBuffer_Load(
279     object sender, System.EventArgs e )
280 {
281     // create shared object
282     HoldIntegerSynchronized sharedLocation =
283         new HoldIntegerSynchronized( outputTextBox );
284
285     // display sharedLocation state before producer
286     // and consumer threads begin execution
287     outputTextBox.Text = sharedLocation.CreateStateOutput();
288
289     // Random object used by each thread
290     Random random = new Random();
291
292     // create Producer and Consumer objects
293     Producer producer =
294         new Producer( sharedLocation, random, outputTextBox );
295     Consumer consumer =
296         new Consumer( sharedLocation, random, outputTextBox );
297
```

Create buffer

Create random number  
for sleep times

Create producer object

Create consumer object



```
298 // create and name threads
299 Thread producerThread =
300     new Thread( new ThreadStart( producer.Produce ) );
301 producerThread.Name = "Producer";
302
303 Thread consumerThread =
304     new Thread( new ThreadStart( consumer.Consume ) );
305 consumerThread.Name = "Consumer";
306
307 // start threads
308 producerThread.Start();
309 consumerThread.Start();
310
311 } // end CircularBuffer_Load method
312
313 } // 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.
```



