

التزامن :

التزامن هو التحكم في ترتيب عمل خيوط التي تملك الحق في تعديل أماكن معينة من الذاكرة. هناك سببان رئيسيان يجعلان المبرمج يقلق بشأن التزامن:

1. التخلص من حالات السباق (race conditions)

2. حماية الخيوط من بعضها البعض

تصور المثال التالي يلعب طفلان لعبة العد من 1 إلى 50 ليختبرا قدرتهما على الحساب. إذا قمنا بتمثيل سلوكهما ببرنامج فإن سيكون تقريبا بالشكل التالي:

```
class Program
{
    static Thread FirstChild = new Thread(new ThreadStart(Count));
    static Thread SecondChild = new Thread(new ThreadStart(Count));
    static void Main(string[] args)
    {
        FirstChild.Name = "First ";
        SecondChild.Name = "Second ";
        FirstChild.Start();
        SecondChild.Start();
    }
    public static void Count()
    {
        int i;
        Thread.Sleep(50);
        for (i = 1; i <= 50; i++)
            Console.WriteLine(Thread.CurrentThread.Name + "
"+i.ToString());
    }
}
```

المنظمات (Mutexes)

قم بتشغيل البرنامج أكثر من مرة ستلاحظ تناوب الخيطين في طباعة النتائج. تخيل لو أضفنا الشرط التالي إلى اللعبة : إذا ابتدأ أحد الطفلين العد قبل الآخر فإن الآخر يجب أن يتوقف مؤقتاً عن العمل حتى يكمل الأول العد. لاحظ يا عزيزي القاري أن بدء أحد الخيطين للعد يعني إقصاء الآخر عن العمل أو بشكل آخر هناك إقصاء تبادلي. لإتمام عملية الإقصاء التبادلي (**Mutual Exclusion**) نحتاج إلى متغيرات من نوع خاص تسمى بالمنظمات **Mutex**. إن المنظمات هي متغيرات من نوع معين تضمن أن يعمل خيط واحد فقط على معالجة مورد معين (موقع ذاكرة) أو تنفيذ شفرة معينة. إذا حصل أحد الخيوط على منظم معين فإن كل الخيوط التي تريد الحصول على هذا المنظم ستصبح في حالة جمود (**Suspended**) إلى أن يطلق أول الواصلين سراح هذا المنظم.

يوجد في مكتبات **.Net** صنف بالاسم **Mutex** ، تضمن كائنات هذا الصنف عمل أول الواصلين وتجميد بقية الخيوط التي تطلب السيطرة عليه. يحتوي هذا الصنف على منهجين أساسيين هما **WaitOne** و **Release** يقوم الأول بحجز المنظم ويقوم الثاني بتحريره من قبضة الخيط الذي استدعي بداخله المنهج ، إذا حدث و كان المنظم محجوزا بسبب خيط آخر فإن الخيط الحالي يصبح في حالة جمود حتى يتحرر المنظم، في الشفرة التالية سنرى كيف يمكن نحل المسألة السابقة بإستخدام كائنات الصنف **Mutex**

```
class Program
{
    static Thread FirstChild = new Thread(new ThreadStart(Count));
    static Thread SecondChild = new Thread(new ThreadStart(Count));
    static Mutex M = new Mutex();
```

```

static void Main(string[] args)
{
    FirstChild.Name = "First ";
    SecondChild.Name = "Second ";
    FirstChild.Start();
    SecondChild.Start();
}
public static void Count()
{
    int i;
    M.WaitOne();
    Thread.Sleep(50);
    for (i = 1; i <= 50; i++)
        Console.WriteLine(Thread.CurrentThread.Name + " " +
i.ToString());
    M.ReleaseMutex();
}
}

```

قم بتشغيل البرنامج السابق ولاحظ عدم تناوب الخيطين في العمل. اصف خيطا ثالثا و وقم بتهيئته وتشغيله مثل بقية الخيوط (أنظر الشفرة التالية) و راقب النتائج ستلاحظ عدم تداخل نتائج الطباعة :

...

```
static Thread ThirdChild = new Thread(new ThreadStart(Count));
```

...

```
ThirdChild.Start();
```

...

بعد أن قمت بالتجربة السابقة نريد أن نجيب على الأسئلة التالية :

1. كيف يمكن أن أحقق التزامن على مستوى العمليات (أي بين عملية وعملية) وليس على مستوى خيوط العملية الواحدة كما في الأمثلة السابقة.
2. ماذا لو كنت أريد أن أجعل عدد معين من الخيوط يعمل بالتوازي وليس خيطا واحدا. لاحظ أن الإقصاء التبادلي يجعل خيطا واحد فقط يعمل ويقصي الخيوط الأخرى.
3. ماذا لو كنت أريد أن أجعل خيط معين بعينه يبدأ العمل قبل الآخرين.

للإجابة على السؤال الأول نحتاج إلى أن نعرف أن جميع المنظمات (Mutexes) التي تعمل في جميع العمليات تخزن في نفس فضاء العناوين خلافا للملفات لذلك يمكن منح اسم للمنظم يكون عاما على مستوى جميع العمليات العاملة. لحسن الحظ تقدم لنا بيئة الدوت نت بانيا (constructor) للصف Mutex يعطينا إمكانية منح اسم للمنظم توضح الشفرة التالية هذا الأمر.

```

class Program
{
    static Mutex M;
    static void Main(string[] args)
    {
        int i = 0;

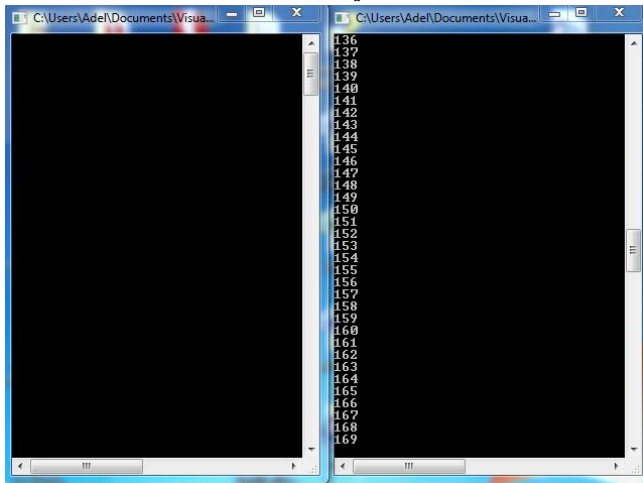
```

```

Random R = new Random();
int n=R.Next(100, 900);
//*****
M = new Mutex(false,"MyMutex");
M.WaitOne();
for (i = 0; i < n; i++)
{
    Thread.Sleep(50);
    Console.WriteLine(i.ToString());
}
M.ReleaseMutex();
}
}

```

لاحظ السطر `M = new Mutex(false,"MyMutex");` لاحظ أن الباني يأخذ بارامترين ، يدل البارامتر الأول على رغبة الخيط المستدعي في أن يكون أول من يقوم بتهيئة هذا المنظم، عليك الحذر عند إرسالك للباني بالقيمة `true` لأن وجود خيط آخر قام بتهيئة قبل الخيط الحالي (لا فرق بين خيطين من نفس العملية و خيطين من برنامجين مختلفين) يؤدي إلى ظهور خطأ. بعد ذلك لاحظ أننا أرسلنا السلسلة "MyMutex"، في الحقيقة إن السلسلة هي اسم المنظم على مستوى النظام، وخلافا لأسماء الملفات تعتبر أسماء المنظمات حساسة لحالة الأحرف، لذلك هناك فرق `MyMutex` و `mymutex`. قم بتنفيذ البرنامج ستلاحظ ظهور أسطر من الأرقام، لاشيء جديد إلى هنا. اذهب إلى المجلد `Debug` الذي يحتوي على الملف التنفيذي ثم قم بتشغيل الملف التنفيذي مرتين، سيؤدي هذا العمل إلى فتح نافذتين لنفس البرنامج، افتح النافذتين بجانب بعضهما (لفعل ذلك بسهولة انقر على شريط المهام باليمين واختر من القائمة **إظهار النوافذ جنباً إلى جنب**) ، ستلاحظ أن احد العمليتين (وهي التي تصل إلى المنظم أولاً) تقوم بطباعة أسطر على الشاشة في بينما تنتظر الأخرى، حتى إذا انتهت الأولى من عملها تبدأ الأخرى في العمل. انظر الصورة . عندما تنتهي



العملية التي تطبع الأسطر من عملها تختفي وتبدأ الأخرى بالعمل. سنقوم بتغيير الشفرة كي تتناوب العمليتان في العمل،

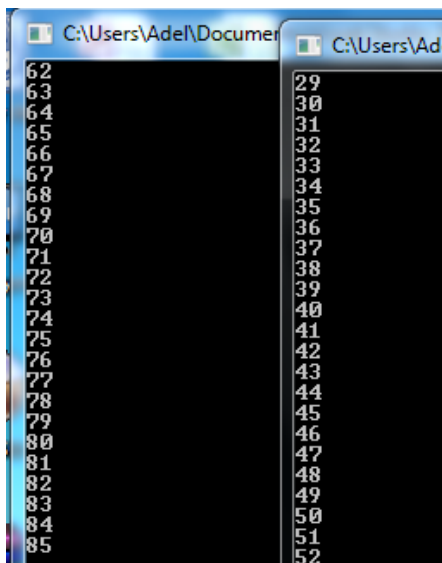
انظر في الشفرة التالية كيف يمكن تغيير البرامج حتى تتناوب العمليتان أثناء العمل.

```

class Program
{
    static Mutex M;
    static void Main(string[] args)
    {
        int i = 0;
        int j = 1;
        Random R = new Random();
        int n=R.Next(100, 500);
        int m = R.Next(1, 100);
        //*****
        M = new Mutex(false, "MyMutex");
        M.WaitOne();
        for (i = 0; i < n; i++)
        {
            Thread.Sleep(50);
            Console.WriteLine(i.ToString());
            if (i == m)
            {
                M.ReleaseMutex();
                Thread.Sleep(50);
                M.WaitOne();
            }
        }
        M.ReleaseMutex();
    }
}

```

لاحظ انه لا يوجد فرق كبير بين هذه الشفرة الأخيرة والشفرة التي قبلها إلا في نقاط بسيطة مثلا ، قمنا بتوليد عدد عشوائي **m** فإذا وصل عداد الحلقة إلى هذه النقطة ، يحرر المنظم ويتوقف الخيط لفترة قصيرة ثم يحاول الحصول على المنظم مرة أخرى، في هذه الأثناء تكون العملية الأخرى متوقفة فإذا حالفها الحظ وأعطاهها الجدول فرصة للعمل أثناء استراحة الأولى تبدأ بالعمل حتى إذا وصل عدادها إلى القيمة **m** الخاصة بها تحرر المنظم وتستريح قليلا، فإذا حالف الحظ العملية الأولى في فترة راحة الثانية وحصلت على إذن من الجدول بالعمل، تقبض على المنظم حتى تنتهي من عملها مما يجعل العملية الثانية في مرحلة انتظار قسري حتى تنتهي الأولى من



عملها لأن الأولى موقفة من قبل المنظم. قم بتشغيل البرنامج (حتى يتكون ملف تنفيذي بالتغييرات الجديدة) ، ثم اذهب إلى المجلد **Debug** وشغل الملف التنفيذي مرتين وقم ببقية العمل الذي قمت به سابقا لفحص نتيجة عمل العمليتين. بهذا الشكل نكون قد أجابنا على السؤال الأول. والآن نرى كيف سنجيب على السؤال الثاني من الأسئلة المطروحة سابقا الذي يقول : ماذا لو أردنا أن نجعل مجموعة من الخيوط تعمل وليس خيطا واحدا. سؤال : لم نقوم بإنشاء أي خيط فأين هو الخيط الذي سيُجدول؟

المسيرات (Semaphores)

للإجابة على هذا السؤال نحتاج إلى طريقة تحدد عدد الخيوط التي يمكنها أن تعمل بصورة متوازية لذلك نحتاج إلى نوع آخر من البيانات يسمى المسيرات

(Semaphores). يمكن للمسيرات أن تحدد عدد الخيوط التي تنفذ شفرة معينة بشكل متوازي. تقوم المسيرات بطرح واحد من القيمة المخزنة داخلها عند كل استدعاء للمنهج **WaitOne**؛ فإذا كانت القيمة المخزنة قبل الطرح مساوية للصفر فإن الطرح لا يحدث ولكن يُجمَد الخيط الذي حاول تنفيذ عملية الطرح ، حتى يحدث رفع للقيمة المخزنة في المسير. بالمقابل يُضاف واحد إلى القيمة المخزنة في المسير بواسطة المنهج **Release** ، فإذا كانت القيمة قبل الإضافة تساوي صفر ، تُرفع القيمة ويرسل إشعار للخيوط التي كانت مجمدة بسبب هذا المسير ويقوم الجدول باختيار واحد من بينها بشكل عشوائي. توضح الشفرة التالية

اللذان يقومان بدور أكبر من عملهما في الصنف **Mutex** بالإضافة إلى البيانات التي تحتاج إلى معرفة عدد الخيوط الأقصى التي يمكن أن تعمل معا ، وقيمة المسير الابتدائية. لمعرفة كيفية استخدام المسير انظر إلى الشفرة التالية:

```
class Program
```

```
{
    static Semaphore MySemaphore = new Semaphore(2, 2);
    //*****
    static Thread First = new Thread(new ThreadStart(ThreadFun));
    static Thread Second = new Thread(new ThreadStart(ThreadFun));
    static Thread Third = new Thread(new ThreadStart(ThreadFun));
    //*****
    static void Main(string[] args)
    {
        First.Name = "First ";
        Second.Name = "Second ";
        Third.Name = "Third ";
        //*****
        First.Start();
        Second.Start();
        Third.Start();
        //*****
    }
    static void ThreadFun()
    {
        int i;
        Thread.Sleep(50);

        MySemaphore.WaitOne();
        string s=Thread.CurrentThread.Name;
        for (i = 0; i < 30;i++)
            Console.WriteLine(s+" "+i.ToString()+" is
            working");
        MySemaphore.Release();
    }
}
```

```
Third 20 is working
Third 21 is working
Third 22 is working
First 5 is working
First 6 is working
First 7 is working
First 8 is working
First 9 is working
First 10 is working
First 11 is working
First 12 is working
First 13 is working
Third 23 is working
Third 24 is working
Third 25 is working
Third 26 is working
Third 27 is working
First 14 is working
First 15 is working
First 16 is working
First 17 is working
First 18 is working
First 19 is working
Third 28 is working
Third 29 is working
First 20 is working
First 21 is working
Second 0 is working
Second 1 is working
Second 2 is working
Second 3 is working
Second 4 is working
First 22 is working
First 23 is working
First 24 is working
```

أولاً أن لدينا الباني (constructor) الذي يحمل نفس اسم الصنف بالتأكيد و يأخذ بارامترين : يشير الأول منهما إلى القيمة الابتدائية للمسير وهي تساوي 2 في حالتنا هذه ، بينما يشير البارامتر الثاني إلى اكبر كمية من عمليات الطرح المتتابعة التي يمكن تنفيذها دون حدوث تجمد، و بما أن البارامتر الثاني يحمل القيمة 2 في مثالنا هذا؛ إذا استطيع خيطان من أصل ثلاثة خيوط مصرح بها العمل بالتوازي (لان كل واحد منهما سيقوم بعملية طرح)، بينما سيظل الأخير حتى ينتهي احدهما من العمل. قم بتشغيل

البرنامج السابق أكثر من مرة . ستلاحظ أن هناك خيطين يتناوبان عملية الطباعة حتى يطبع احدهما القيمة 29 (معناه انه انتهى من عمله) ساعتها فقط يبدأ الخيط الثالث بالطباعة فمثلا في إحدى المحاولات التي قمنا بها حصلت على النتيجة إلى اليمين. لاحظ من المستطيل الأول أن الخيط **First** و **Third** يتناوبان العمل (لأن كلا منهما تمكن من طرح واحد من قيمة المسير بلا مشكلة)، بينما ينتظر الخيط **Second** دون فعل شيء (لأنه حاول طرح واحد من قيمة المسير الذي كان يحتوي على صفر بعد أن سبقه إلى المسير الخيطان **First** و **Third**). لاحظ أسفل الصورة أن الخيط **Third** وصل إلى القيمة 29 وبعدها بدأ الخيط **Second** بالطباعة على الشاشة. انظر إلى الشفرة التي تخص الإجراء **ThreadFun** ستجد انه بعد انتظار 50 مللي ثانية يقوم بطرح واحد من قيمة المنظم الذي يحمل اثنين كقيمة ابتدائية. ماذا يعني هذا ؟ فقط أول اثنين من الخيوط الواصلة ستتحج في العمل دون مشاكل. لان ثالث الواصلين سيتجمد إذا حاول العمل إذا لم يقم احد الخيطين بالسماح له بزيادة واحد إلى المسير. من السهل أن نستنتج أن زيادة القيمة الابتدائية يعطي فرصة اكبر لعدد اكبر من الخيوط أن تعمل بالتوازي مع بعضها.

ملحوظة

1. ليس من الضروري أن ترتبط الخيوط بنفس الإجراء مثل ما حصل في المثال السابق مع **ThreadFun** ممكن أن تكون شفرة الخيوط بالشكل التالي:

```
class Program
{
    //*****
    static Semaphore Sem = new Semaphore(2, 2);
    static void Main(string[] args)
    {
        Thread First = new Thread(new ThreadStart(FirstFun));
        Thread Second = new Thread(new ThreadStart(SecondFun));
        Thread Third = new Thread(new ThreadStart(ThirdFun));
        First.Start();
        Second.Start();
        Third.Start();
    }
    //*****
    static void FirstFun()
    {
        Thread.Sleep(50);
        Sem.WaitOne();
        for (int i = 0; i < 30; i++)
            Console.WriteLine("First " + i.ToString());
        Sem.Release();
    }
    //*****
    static void SecondFun()
    {
        Thread.Sleep(50);
        Sem.WaitOne();
        for (int i = 0; i < 20; i++)
            Console.WriteLine("Second " + i.ToString());
        Sem.Release();
    }
    //*****
    static void ThirdFun()
    {
```

```

{
    Thread.Sleep(50);
    Sem.WaitOne();
    for (int i = 0; i < 50; i++)
        Console.WriteLine("Third " + i.ToString());
    Sem.Release();
} }

```

2. إذا رفعت شفرة أحد الخيوط قيمة المسير عن القيمة العظمى المحددة للمسير و التي تكون عادة في مرسله في البارامتر الثاني إلى الباني فسيحدث خطأ وسيوقف البرنامج لذلك احرص على عدم زيادة قيمة المسير على القيمة العظمى.

3. المنظم (Mutex) ما هو إلا مسير قيمته العظمى 1 وقيمته الابتدائية 1 نستطيع أن نعرض مثالا على استخدام المسير كمنظم، انظر إلى الشفرة التالية:

```

class Program
{
    //*****
    static Semaphore Sem = new Semaphore(1, 1);
    static void Main(string[] args)
    {
        Thread First = new Thread(new ThreadStart(FirstFun));
        Thread Second = new Thread(new ThreadStart(SecondFun));
        //*****
        First.Start();
        Second.Start();
        //*****
    }

    static void FirstFun()
    {
        Thread.Sleep(50);
        Sem.WaitOne();
        for (int i = 0; i < 30; i++)
            Console.WriteLine("First " + i.ToString());
        Sem.Release();
    }

    static void SecondFun()
    {
        Thread.Sleep(50);
        Sem.WaitOne();
        for (int i = 0; i < 20; i++)
            Console.WriteLine("Second " + i.ToString());
        Sem.Release();
    }
}

```

إن القيمة الابتدائية للمسير هي 1، يبدأ الخيط الأول بالعمل يطرح واحدا من قيمة المسير فتصبح قيمة المسير 0، يحاول الخيط الثاني العمل فيجد أن قيمة المسير 0، يحاول أن يطرح منها واحدا فيُجمَد حتى ينتهي الخيط الأول من العمل ثم يُشعر الثاني بذلك. لو بدأ الخيط الثاني العمل قبل الأول فإن مسلسل الأحداث سيتكرر ولكن بالعكس، يُجمَد الخيط الأول حتى ينتهي الثاني من العمل. لاحظ أن عمل احد الخيطين يعني إقصاء الآخر عن العمل، انسخ البرنامج السابق ثم جرب الشفرة الماضية عدة مرات ستلاحظ عدم تداخل نتائج الطباعة. هكذا أن نستطيع نقول أننا جابونا على السؤال الثاني. سنرى الآن كيف يمكن أن نجيب على السؤال الثالث: ماذا لو أردنا أن يعمل خيط معين قبل بقية الخيوط سنقوم بحل هذه المسألة من أجل خيطين:

1. ننشئ مسيرا ونجعل قيمته الابتدائية صفر ونجعل قيمته العظمى 1
2. نجعل الخيط الذي نريده ثانيا يستدعي المنهج WaitOne بمعنى أنه سيحاول طرح واحد من مسير يساوي صفر أصلا وبهذا يتجمد فوراً.
3. نجعل الخيط الثاني يعمل بدون أي ذكر لأي مسير، وعند نهاية شفرته نجعله يرفع المسير بمقدار واحد.
4. سيُشعر الخيط الآخر بعملية الرفع وسيعمل. انظر إلى الشفرة التالية:

```
class Program
{
    static Semaphore Sem = new Semaphore(0, 1);
    static void Main(string[] args)
    {
        Thread First = new Thread(new ThreadStart(FirstFun));
        Thread Second = new Thread(new ThreadStart(SecondFun));
        First.Start();
        Second.Start();
    }
    static void FirstFun()
    {
        Thread.Sleep(50);
        for (int i = 0; i < 30; i++)
            Console.WriteLine("First " + i.ToString());
        Sem.Release();
    }
    static void SecondFun()
    {
        Thread.Sleep(50);
        Sem.WaitOne();
        for (int i = 0; i < 20; i++)
            Console.WriteLine("Second " + i.ToString());
        Sem.Release();
    }
}
```

هل رايت مقدار السهولة؟ والان وسع العملية من اجل 3 خيوط First,Second,Third بحيث تحصل على الترتيب التالي:

Third->First->Second

```
class Program
{
    //*****
    static Semaphore Sem1 = new Semaphore(0, 1);
    static Semaphore Sem2 = new Semaphore(0, 1);
    static void Main(string[] args)
    {
        Thread First = new Thread(new ThreadStart(FirstFun));
        Thread Second = new Thread(new ThreadStart(SecondFun));
        Thread Third = new Thread(new ThreadStart(ThirdFun));
        //*****
        First.Start();
        Second.Start();
        Third.Start();
    }
}
```

```

static void FirstFun()
{
    Sem1.WaitOne();
    for (int i = 0; i < 30; i++)
        Console.WriteLine("First " + i.ToString());
    Sem1.Release();
    Sem2.Release();
}
static void SecondFun()
{
    Sem2.WaitOne();
    for (int i = 0; i < 20; i++)
        Console.WriteLine("Second " + i.ToString());
    Sem2.Release();
}
static void ThirdFun()
{
    for (int i = 0; i < 20; i++)
        Console.WriteLine("Third " + i.ToString());
    Sem1.Release();
}
}

```

1. الدالة المرتبطة بالخيط **Third** لم تقم بأي عملية طرح لذلك ستبدأ عملها بلا مشاكل، (لا تستطيع بقية الخيوط العمل، لماذا؟).
2. يضاف واحد إلى المسير **Sem1** من قبل الخيط **Third**.
3. تؤدي هذه الإضافة إلى تنشيط عمل الخيط **First** (لماذا **First** وليس **Second**).
4. يقوم **First** بطرح واحد من المسير **Sem1** ثم يكمل عمله إلى أن يضيف واحد إلى المسير **Sem1** ثم واحد إلى المسير **Sem2**.
5. تؤدي إضافة واحد إلى **Sem2** إلى تنشيط عمل الخيط **Second**.

بهذا نكون قد اجبنا على السؤال الثالث وهو كيف يمكن التحكم في ترتيب الخيوط؟ أو كيف يمكن جعل خيط معين يبدأ العمل قبل بقية الخيوط.

كيف نجعل المسيرات تقوم بمزامنة العمليات وليس خيوط عملية واحدة؟ انظر الشفرة التالية:

```

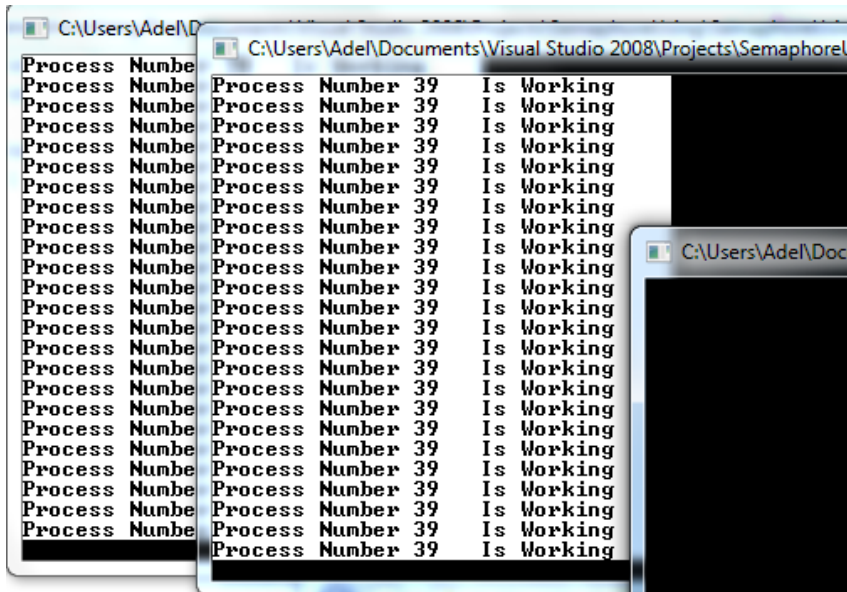
static void Main(string[] args)
{
    int i = 0;
    Semaphore MySemaphore = new Semaphore(2, 2, "MySemaphore");
    int n = (new Random()).Next(1, 100);
    MySemaphore.WaitOne();
    for (i = 0; i < 300; i++)
    {
        Thread.Sleep(50);
        Console.WriteLine("Process Number " + n.ToString() + " Is
Working ");
    }
    MySemaphore.Release();
}

```

لاحظ أن الباني قد تغير قليلاً، أصبح يحتوي على سلسلة، هذه السلسلة تعبر عن متغير عام عبر النظام وتمثل اسماً للمسير الذي سينشأ كما هو الحال في المنظمات (الم نقل سابقاً أن المنظمات حالة خاصة من المسيرات)، قم بتشغيل البرنامج، ستلاحظ أن البرنامج يطبع أسطر فقط أمر عادي لا جديد، لاحظ أن المسير أخذ 2 كقيمة ابتدائية. الآن اذهب إلى المجلد **Debug** الذي يحتوي على الملف التنفيذي

وقم بتشغيل 3 نسخ من الملف التنفيذي أو أكثر واجعل النوافذ المفتوحة بجانب بعضها ماذا تلاحظ؟ نسختان فقط من البرنامج تعملان

بينما تنتظر البقية دورها. انظر الصورة
سترى أن العملية الثالثة على يمين الصورة
في حالة انتظار. الآن سنرى كيف أن نتحكم
بسير العمليات بحيث تعمل عملية ما قبل
الأخرى حتى لو شغلنا الأخرى قبل الأولى.



هذه هي شفرة العملية 1 :

```
static void Main(string[] args)
{
    int i = 0;
    Semaphore MySemaphore = new Semaphore(0,1 , "MySemaphore");
    int n = (new Random()).Next(1, 100);
    MySemaphore.WaitOne();
    for (i = 0; i < 300; i++)
    {
        Thread.Sleep(50);
        Console.WriteLine("Process Number "+n.ToString()+"    Is Working    "+
i.ToString());
    }
    MySemaphore.Release();
}
```

لاحظ أن المسير هنا أخذ القيمة 0، هذا يعني أن هذه العملية ستتوقف فور محاولتها لتنفيذ السطر `MySemaphore.WaitOne()` (لماذا؟). انظر إلى شفرة العملية 2 :

```
static void Main(string[] args)
{
    int i = 0;
    Semaphore MySemaphore = new Semaphore(0,1 , "MySemaphore");
    int n = (new Random()).Next(1, 100);
    // MySemaphore.WaitOne();
    for (i = 0; i < 300; i++)
    {
        Thread.Sleep(50);
```

```

        Console.WriteLine("Process Number "+n.ToString()+" Is
Working "+i.ToString());
    }
    MySemaphore.Release();
}

```

سؤال : ماذا يحدث لو شغلنا عمليتين بنفس اسم المسير بحيث كانت القيمة الابتدائية التي تقترحها كل قيمة مختلفة عن الأخرى؟

الجواب : يعتمد النظام القيمة التي تقترحها أول عملية .

المراقبات (Monitor) :

مع أن المسيرات تحل مشاكل التزامنة بشكل فعال ولكن لها عيب قوي وهو اهتمام المبرمج بتوقيت القبض و التحرير (WaitOne Release) على شفرة معينة وقد يؤدي الخطأ في توقيت أي واحد منها إلى حالة استعصاء (deadlock) خاصة عند استعمال أكثر من مسير أو منظم، انطلاقاً من هذا الأمر و رغبة في تحقيق إقصاء متبادل (mutual exclusion) بشكل آمن، اقترح مفهوم آخر للزامنة وهو مفهوم المراقبات (Monitor). يسمح المراقب لخيوط واحد فقط بالسيطرة على كائن معين، ويجعل بقية الخيوط التي تريد تغيير هذا الكائن في حالة جمود حتى يحصل هذه الخيوط على إشعار (notify) بتحرير الكائن. لحسن الحظ يوجد في مكتبات الدوت نت الصنف Monitor الذي يحتوي عدد من المناهج الاستاتيكية (static methods) سنرى لاحقاً كيف سنستخدم أهم هذه المناهج. انتبه للأمور التالية:

1. يقوم المراقب داخل شفرة خيط معين بحجب كائن معين عن بقية الخيوط بواسطة المنهج Enter. إذا حاول خيط آخر حجب هذا الكائن فإنه سيصبح في حالة إنتظار (WaitSleepJoin) حتى يحصل على إشعار بالعمل.
2. يزيل المراقب الحجب على كائن معين بواسطة المنهج Exit. أن إزالة الحجب عن كائن معين يجعل أحد الخيوط التي كانت تنتظر حجب الكائن ستقوم بحجب هذا الكائن عن البقية وهكذا.
3. يقوم المراقب بإشعار أحد الخيوط الأخرى المنتظرة تحرير الكائن، بواسطة المنهج Pulse.
4. يقوم المراقب بإشعار كل الخيوط التي تنتظر تحرير كائن معين بواسطة المنهج PulseAll.
5. إذا أراد خيط الدخول بشكل طوعي في حالة انتظار فإنه يدخل إليها باستدعاء المنهج Wait.
6. عند دخول خيط في حالة انتظار يمكن لبقية الخيوط (التي كانت تنتظر كائناً محجوزاً لدى هذا الخيط) أن تعمل في حال أشعرها هذا الخيط بذلك. إذا لم يشعرها ودخل في حالة انتظار وكانت البقية تنتظر تحريره لكائن فإننا سنحصل على حالة استعصاء (deadlock).
7. جميع المناهج السابقة تستقبل بارامترات من النوع reference types لذلك عند حجب أعداد صحيحة أو سلاسل قم بتحويلها إلى object يطلق على هذه العملية boxing كما يطلق على العملية العكسية unboxing

مثال على عملية boxing

```

int a=5;
object b=a;

```

مثال على عملية

```

a=(int ) object;

```

في الأمثلة القادمة سنرى كيفية استخدام الصنف Monitor

```

class Program
{
    static int x=0;
    static object MyObject1 = x;
    //*****

```

```

static void Main(string[] args)
{
    Thread First = new Thread(new ThreadStart(ThreadFun));
    Thread Second = new Thread(new ThreadStart(ThreadFun));
    //*****
    Second.Name = "Second";
    First.Name = "First";
    //*****
    First.Start();
    Second.Start();
}
//*****
public static void ThreadFun()
{
    int i=0;
    Thread.Sleep(50);
    Monitor.Enter(MyObject1);
    for (i = 0; i < 100; i++)
        Console.WriteLine(Thread.CurrentThread.Name + " " +
i.ToString());
    Monitor.Exit(MyObject1);
}
//*****
}

```

إذا قمت باستدعاء المناهج وأرسلت x كبارامتر مباشرة من دون **boxing** سيؤدي هذا إلى حصولك على رسالة خطأ عند استدعاءك للمنهج Exit أو حتى غيره ما عدا **Enter** لماذا؟ لأن استدعاءك **Enter** يجعله يحجز ذاكرة من أجل x فهو يقوم بعملية **boxing** واستدعاء **Exit** يجعله يحجز ذاكرة من جديد من أجل **boxing** آخر. سيلاحظ المنهج **Exit** أن هذه الذاكرة ليست مسجلة ضمن مواقع الذاكرة التي حجزها **Enter** فيصدر خطأ (لماذا؟) هو يتوقع أن يحصل كبارامتر على موقع ذاكري سيق أن سجله المنهج **Enter**. ولماذا تقوم المناهج بعملية **boxing** كل مرة؟ لأن هذه المناهج تتوقع بارامترات من النوع (reference type) وليس من النوع (value type). (ما هو الفرق بين الاثنين؟ راجع دروس OOP. التغيير في قيم متغيرات النوع سيظل حتى بعد انتهاء عمل الدالة في حين لا تتأثر قيم المتغيرات التي أرسلت كبارامترات من النوع الثاني باستدعاء الدوال لأنها ترسل نسخة من المتغير وليس المتغير).

1. انظر إلى البداية ستلاحظ أنني قمت بتعريف حقل من النوع **object** وآخر من النوع **int**
2. في الدالة الرئيسية **Main** قمنا بتعريف كائنين من النوع **Thread** وربطناهما بالدالة **ThreadFun** ثم طلبنا تنفيذ الخيطين
3. في شفرة الدالة ستجد أن الخيط الحالي (الذي ينفذ شفرة الدالة في اللحظة الراهنة) سيحصل على حق حصري بالتصرف في الكائن **MyObject1**. إذا حاول خيط آخر الحصول على هذا الحق لنفس الكائن فإنه سيصبح في حالة انتظار حتى ينتهي الخيط الأول من العمل.

شغل البرنامج، ماذا تلاحظ؟ تلاحظ أن الخيطين لا يتناوبان وإنما ينفرد احدهما بالعمل، حتى إذا انتهى بدأ الآخر بالعمل مما يعني أن الاقصاء المتبادل متحقق بشكل فعال.

سنرى في المثال التالي كيف يمكن لخيط الدخول في حالة انتظار اختيارية. عدل شفرة الدالة **ThreadFun**

كما هو موضح في الأسطر التالية ثم تتبع عمل الشفرة:

```

public static void ThreadFun()
{
    int i=0;
    Thread.Sleep(50);
    Monitor.Enter(MyObject1);

```

```

for (i = 0; i < 100; i++)
{
    Console.WriteLine(Thread.CurrentThread.Name + " " + i.ToString());
    if (i == 50)
    {
        Console.WriteLine(Thread.CurrentThread.Name + " " + i.ToString());
        Monitor.Pulse(MyObject1);
        Monitor.Wait(MyObject1);
    }
    Monitor.Pulse(MyObject1);
    Monitor.Exit(MyObject1);
}

```

عند وصول عداد الحلقة إلى 50 يقوم الخيط الحالي بإشعار الخيط الآخر بواسطة المنهج **Pulse** ويتوقف باستخدام المنهج **Wait**، ينشط الخيط الآخر المتوقف بسبب الكائن (**MyObject1**)، حتى يصل عداده إلى 50 يتوقف وينشط الخيط الأول، الذي يكمل العد حتى يصل إلى 100 وينتهي الحلقة، ثم يقوم بإشعار (**notify**) الخيط الآخر. طبعاً لا فرق بين استدعاء **Pulse** واستدعاء **PulseAll** سوى أن **PulseAll** يحفز أكثر من خيط وبالتالي احتمال ضياع الإشارة أقل (لأن الإشارة قد تأتي في وقت لم يكن فيه الخيط مستعداً لاستقبالها فتضيع وكأنها لم تكن) في حين يحفز **Pulse** خيطاً واحداً. لاحظ أننا استدعينا **Pulse** قبل استدعاء **Wait**. قد يحدث أحياناً أن خيطاً معيناً لا يريد أن يصبح في حالة انتظار فإن نجحت محاولة القبض على كائن معين كان بها، وإلا فسيقوم بأعمال أخرى...! لتفعل ذلك تحتاج إلى استخدام المنهج **TryEnter** بدلاً من المنهج **Enter** أنظر إلى التعديلات التي يمكن إحداثها في **ThreadFun()**

```

public static void ThreadFun()
{
    int i=0;
    Thread.Sleep(50);
    if (Monitor.TryEnter(MyObject1))
    {
        for (i = 0; i < 100; i++)
        {
            Console.WriteLine(Thread.CurrentThread.Name + " " + i.ToString());
            if (i == 50)
            {
                Console.WriteLine(Thread.CurrentThread.Name + " " + i.ToString());
                Monitor.Pulse(MyObject1);
                Monitor.Wait(MyObject1);
            }
            Monitor.Pulse(MyObject1);
            Monitor.Exit(MyObject1);
        }
        else
            Console.WriteLine("No Way To Enter by " + Thread.CurrentThread.Name);
    }
}

```

يقوم المنهج **TryEnter** بمحاولة الدخول فإذا نجحت يحجز الكائن المعني ويعيد القيمة **true** فيما عدا ذلك سينفذ السطر الأخيرة من الشفرة السابقة. هناك فرق جوهري بين المراقبات والمسيرات وهو أن المراقبات تعمل على مستوى عملية واحدة فقط أي تنظم الوصول إلى الموارد بين خيوط عملية واحدة في حين تستطيع المسيرات فعل ذلك إضافة إلى قدرتها على التحكم بالموارد على مستوى النظام (وصول عمليات مختلفة إلى موارد) وليس بين خيوط عملية واحدة كما هو الحال في المراقبات.

العبارة lock :

تكافئ العبارة lock استدعاء المنهجين Enter and Exit

`Monitor.Enter(x);``. . .``Monitor.Exit(x);``lock(x)``{``. . .``}`

بدلاً من كتابة المنهجين السابقين يمكن استخدام العبارة **lock** التي تقوم بحجز الكائن، ولا تستطيع بقية الخيوط الحصول على الحق الحصري تماماً مثلما يفعل الصنف **Monitor** انظر إلى الشفرة التالية:

`class Program``{``static int x=0;``static object MyObject1 = x;``static object MyObject2 = MyObject1;``static void Main(string[] args)``{``Thread First = new Thread(new ThreadStart(FirstFun));``Thread Second = new Thread(new ThreadStart(SecondFun));``Second.Name = "Second";``First.Name = "First";``First.Start();``Second.Start();``}``public static void FirstFun()``{``int i=0;``Thread.Sleep(50);``lock (MyObject1)``{``// Monitor.Enter(MyObject1);``for (i = 0; i < 100; i++)``Console.WriteLine(Thread.CurrentThread.Name + " " + i.ToString());``// Monitor.Exit(MyObject1);``}``}``public static void SecondFun()``{``int i = 0;``Thread.Sleep(50);``lock (MyObject1)``{``// Monitor.Enter(MyObject1);``Console.WriteLine("SecondFun Entered");``MyObject1 = MyObject2;``Console.WriteLine("SecondFun Exited");``// Monitor.Exit(MyObject1);``}``}``}`

وحتى نكون أكثر امعانا في التأكيد. سنحاول استدعاء المناهج **Pulse** و **Wait** (لو لم يكونا متكافئين لحصلنا على رسالة خطأ تفيد استدعاء مناهج من أجل كائن لم نحجزه بالمنهج **Entered**). نستبدل الدالتين **FirstFun** و **SecondFun** بالدالة:

```
public static void ThreadFun()
{
    int i=0;
    Thread.Sleep(50);
    lock (MyObject1)
    {
        for (i = 0; i < 100; i++)
        {
            Console.WriteLine(Thread.CurrentThread.Name + " " + i.ToString());
            if (i == 50)
            {
                Monitor.Pulse(MyObject1);
                Monitor.Wait(MyObject1);
            }
        }
        Monitor.Pulse(MyObject1);
    }
}
```

نغير الربط في الدالة Main كما يلي:

```
...
Thread First = new Thread(new ThreadStart(ThreadFun));
Thread Second = new Thread(new ThreadStart(ThreadFun));
...
```

بعد ذلك قم بتشغيل البرنامج ستلاحظ التناوب عند القيمة 50، ألا يدل هذا على عمل المنهجين **Pulse** و **Wait**.

فقط انتبه من حالات الاستعصاء (**Deadlock**) التي تنتج من شفرتين من الصورة:

```
public static void FirstFun()
{
    int i=0;

    lock (MyObject1)
    {
        Thread.Sleep(50);
        lock (MyObject2)
        {
            for (i = 0; i < 100; i++)
                Console.WriteLine(First.Name+" "+i.ToString());
        }
    }
}
//*****
public static void SecondFun()
{
    int i = 0;

    lock (MyObject2)
    {
```

```

Thread.Sleep(50);
lock (MyObject1)
{
    for (i = 0; i < 100; i++)
        Console.WriteLine(Second.Name+" "+ i.ToString());
}
}

```

تحتجز الدالة الأولى الكائن **MyObject1** ثم تطلب حجز الكائن **MyObject2** أما الثانية فتحتجز **MyObject2** ثم تطلب حجز الكائن **MyObject1** وهنا تقع المشكلة، تخيل ان الاولى تبدأ عملها فتحجز الكائن **MyObject1** ثم تتشغل قليلا (بسبب **Sleep** في حالتنا هذه) وفي هذه الاثناء تأتي الثانية وتحتجز الكائن **MyObject2** ثم تتشغل بتنفيذ امر ما (تعد من 1 إلى 1000 مثلا) وفجأة تطلب الاولى الكائن **MyObject2** فتجده محجوزا من قبل الثانية فتدخل حالة انتظار (كما هو المعروف عن المراقبات) ثم تعمل الثانية فتطلب حجزا **MyObject1** وتجده محجوزا فنذهب في حالة انتظار ولن يكمل البرنامج عمله ابدا.