

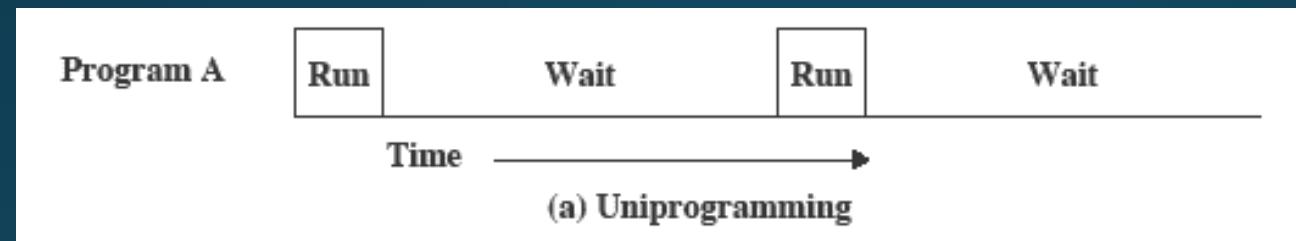
*Нишки ( Threads )*

# Съдържание

- 
- Процеси и нишки;
  - Основни състояния на нишки;
  - Атрибути на нишки;
  - Основни методи;
  - Използване на нишки;
  - Пул от нишки;
  - Примери;

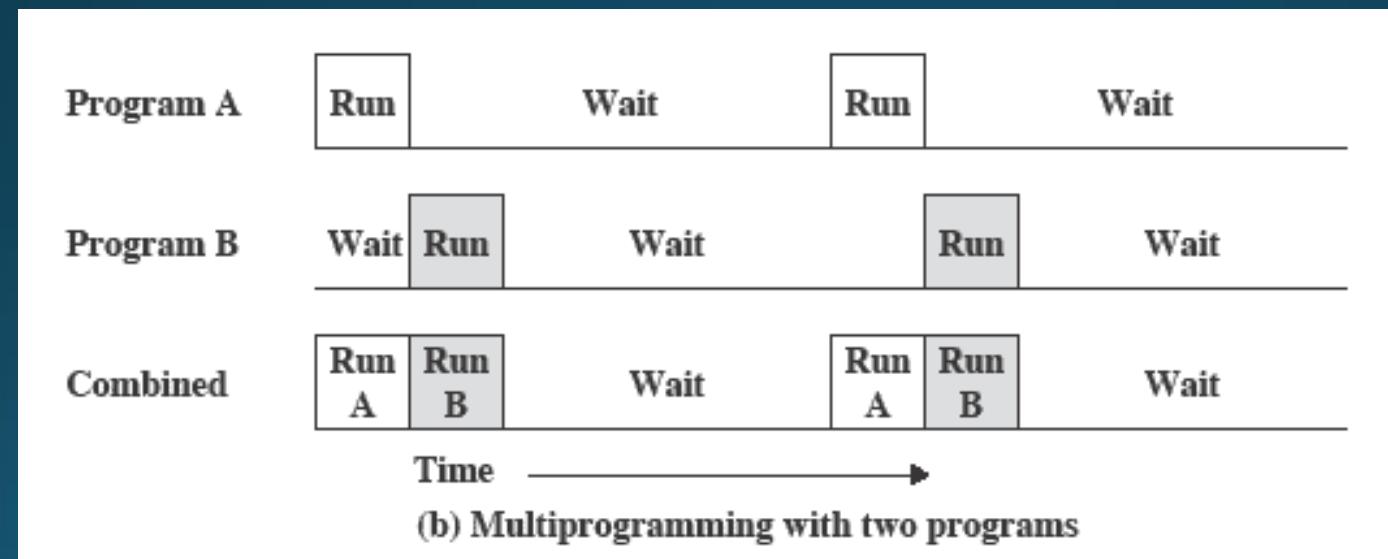
# Еднопрограмни

- Изчакване :

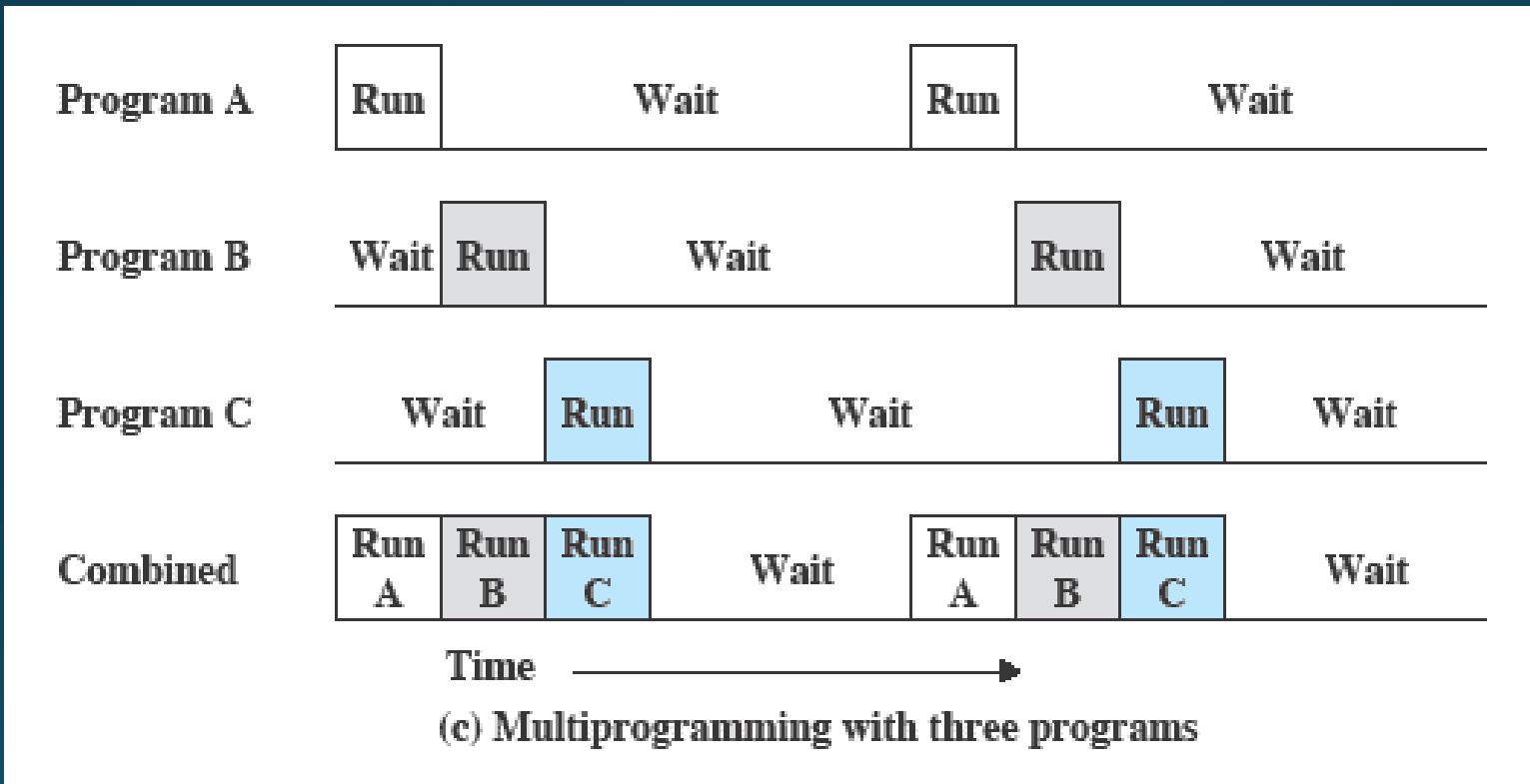


# Мултипрограмност

Когато една задача трябва да чака за I/O, процесорът може да превключи към друга задача



# Multiprogramming



# Процеси и Нишки

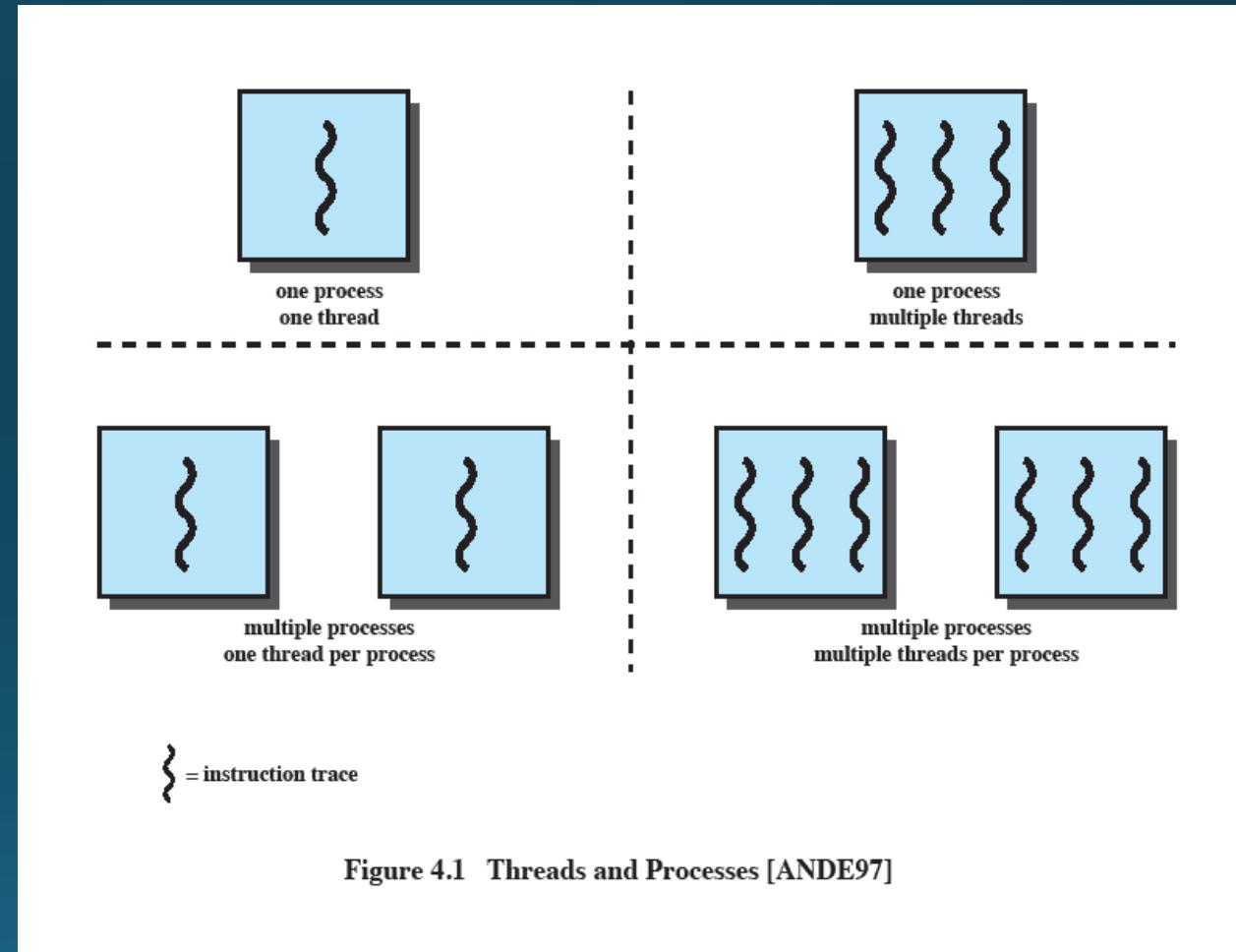
- Процесите имат две характеристики:
  - **Собствени ресурси** – всеки процес включва собствено адресно пространство за съхранение на данните му
  - **Планиране/изпълнение** – процесът има път на изпълнение, който може да бъде съгласуван с други процеси.
- Тези две характеристики се третират независимо от операционната система.

# Процеси и Нишки

- Единицата за изпълнение се отнася като **нишка (thread)** или лек процес.
- Единицата за собственост на ресурсите се отнася като процес или задача (**task**)

# Многонишковост

- Способността на ОС да поддържа множество конкурентни пътища на изпълнение в рамките на един процес.

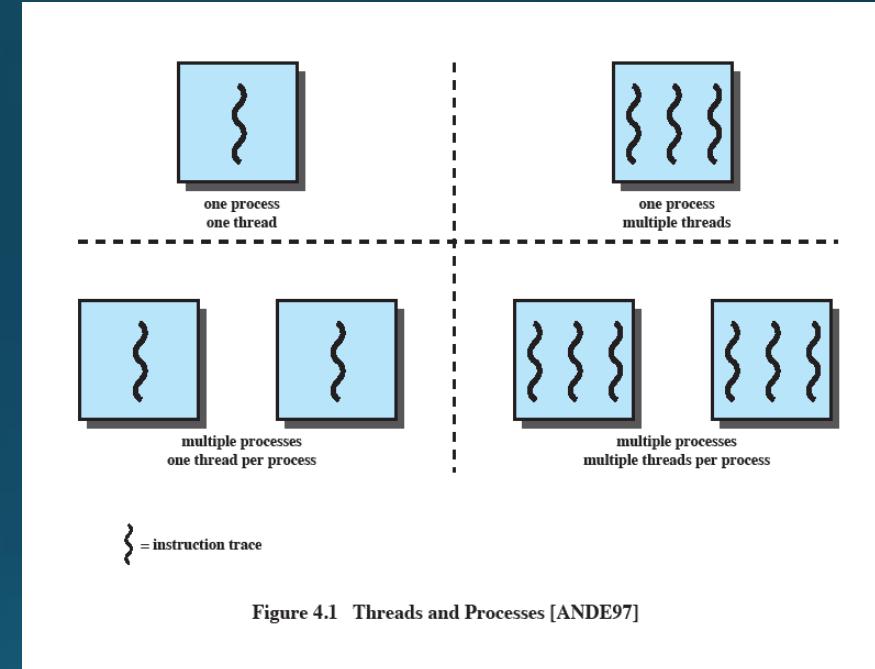


# Еднонишков подход

- MS-DOS поддържа един потребителски процес и една нишка.
- UNIX поддържа множество потребителски процеси, но само една нишка в процес.

# Многонишковост

- Java run-time environment (JRE) е един процес с много нитки
- Множество процеси и нитки има в Windows, Solaris и много съвременни версии на UNIX



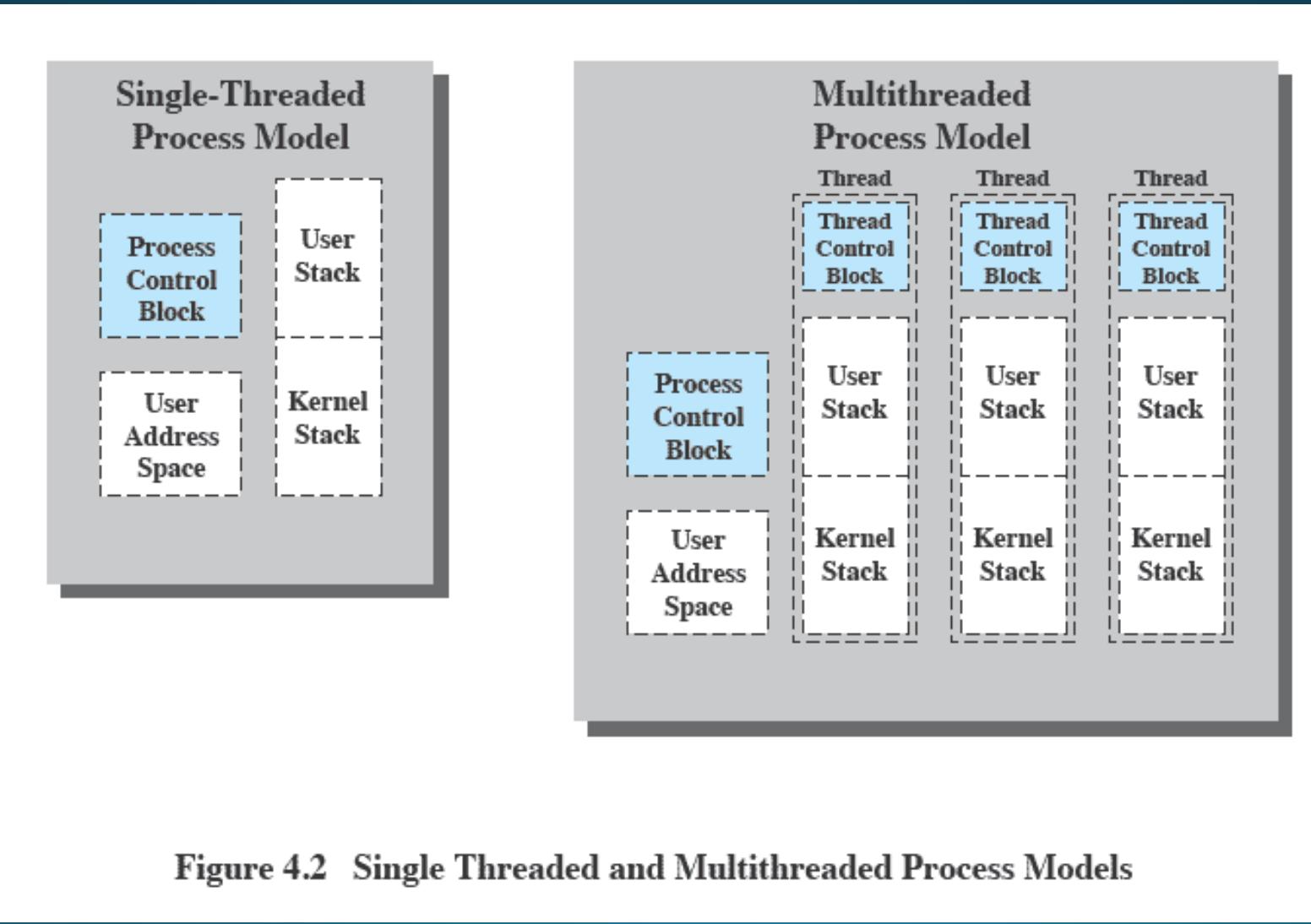
# Процеси (някои особености)

- Виртуално адресно пространство, което съдържа данните на процеса
- Защитен достъп до:
  - Процесори
  - Други процеси
  - Файлове
  - Входни и изходни данни (I/O resources)

# Една или няколко нишки в процес

- Всяка нишка има
  - Състояние на изпълнение (изпълнява се (running), готова (ready) и т.н.)
  - Съхранен контекст на нишката, когато тя не се изпълнява
  - Стек на изпълнение
  - Заделена статична памет за локални променливи
  - Достъп до паметта и ресурсите на процеса (нишките в един процес си ги споделят)

# Нишки / процеси



# Предимства на нишките

- По-малко време е нужно за създаването на нова нишка, отколкото на нов процес
- По-малко време е необходимо за прекратяване на нишки отколкото на процес
- Превключването между две нишки отнема по-малко време отколко превключването между два процеса
- Нишките могат да комуникират една с друга без да извикват ядрото

# Употреба на нишки в едно-потребителска ОС

- Foreground и background обработка
- Асинхронна обработка

# Нишки

- Има няколко дейности, които влияят върху всички нишки в един процес
  - ОС трябва да управлява тези дейности на ниво процес

## Примери:

- Временното прекратяването на процес включва прекратяването на всички нишки в него.
- Приключването на процес терминира всички нишки в процеса

# Действия както при процесите

- Нишките имат състояния по време на изпълнение и могат да се синхронизират
  - Подобно на процесите.
- Фокусираме се върху тези два аспекта на функционалността на нишките.
  - Състояния (states)
  - Синхронизация

# Състояния на нишките

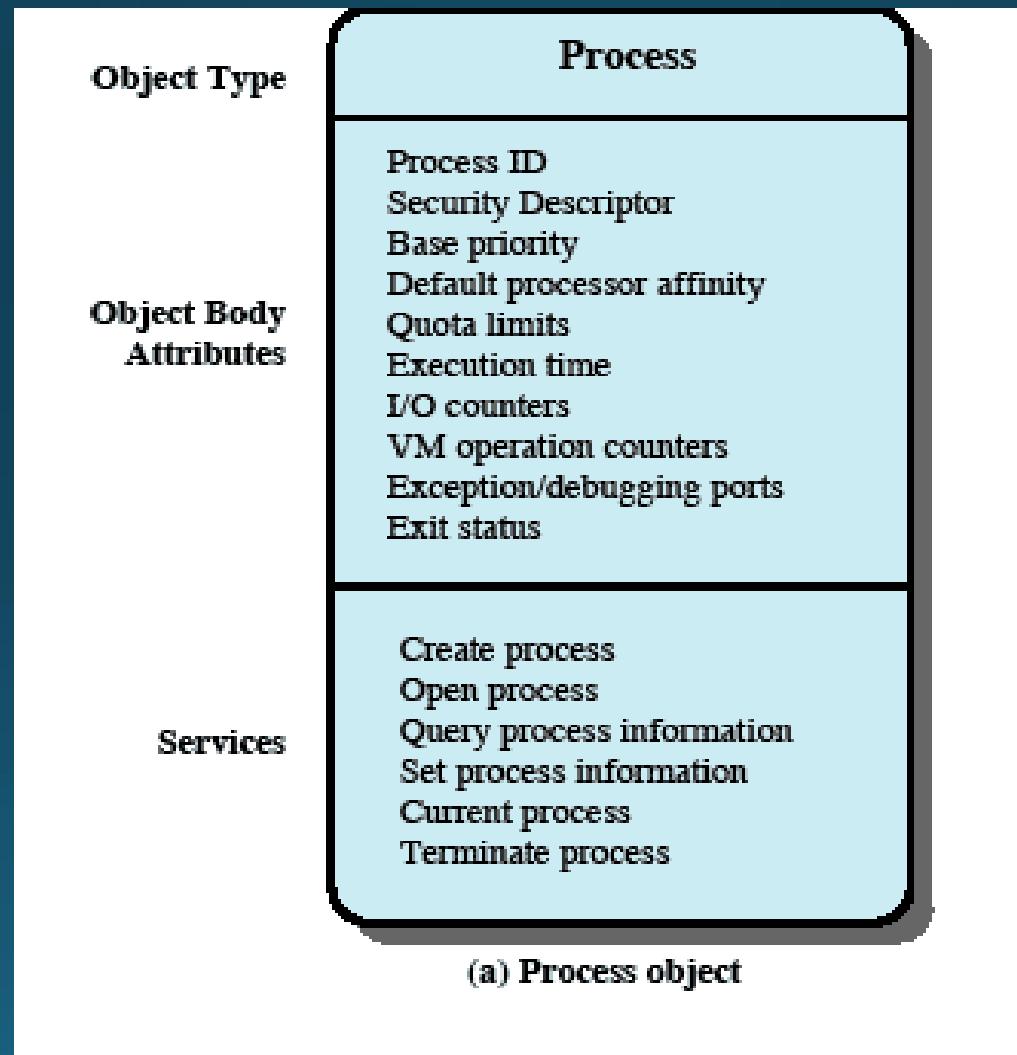
- Състояния, асоциирани с промяна на състоянието на нишката:
  - Block
    - Блокирането може да доведе до блокирането на други или всички нишки
  - Unblock
  - Finish
    - *Премахване на контекста в регистрите и стековете*

# Пример:

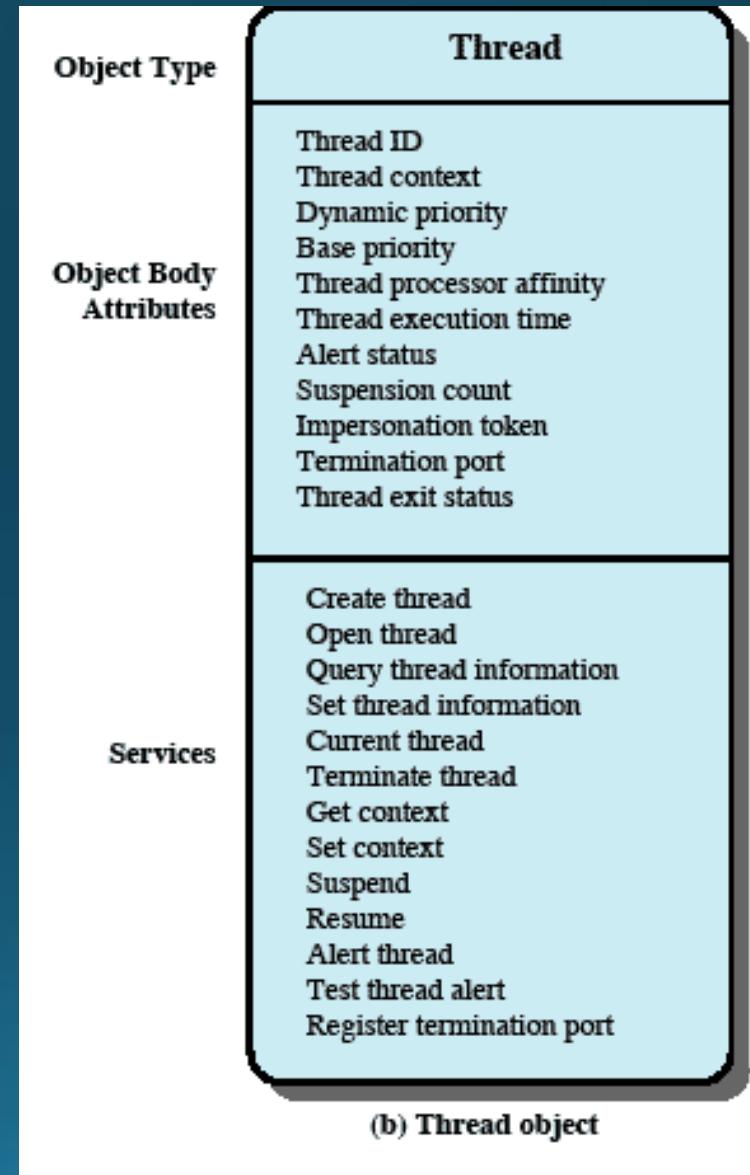
## Извикване на отдалечени процедури

- . Задание:
  - Програма, която изпълнява извиквания към две отдалечени процедури (RPCs)
  - до два различни сървъра (hosts).
  - да представя комбиниран резултат.

# Процесът като обект в Windows



# Нишката като обект в Windows



# Основни понятия в контекста

- Многозадачност
  - Нишки
  - Синхронизация
  - Асинхронни извиквания
- 
- Забележка: Под някои слайдове в полето забележки има подробно описание на съдържанието на слайда

# Многозадачност – съдържание

- **Многозадачност**

- Проблемът – защо многозадачност?
- Ползите от многозадачността
- Решението – процеси и нишки
- Прилики и разлики
- Какво предлагат?
- Кога са удобни нишките?
- Многозадачност – видове
- Имплементации на многозадачност
- Application Domains
- Нишки
- Синхронизация
- Асинхронни извиквания

# Проблемът

- Има случаи, в които вашето приложение трябва да изпълни времеотнемащи операции или да чака за освобождаването на ресурс
- Има случаи, в които вашето приложение трябва да извършва операция на заден план
- Нужен е механизъм, който да позволява няколко операции да се извършват едновременно

# Ползите от многозадачността

- **Performance** – на машината с няколко процесора работи по-бързо
- **Responsiveness** – системата отговаря максимално бързо при интерактивна работа
- **Throughput** – подобряване на производителността
  - Пример със супермаркета – обслужване на няколко каси едновременно
  - Обслужване на много потребители едновременно

# Решението – процеси и нишки

- Процесът представлява съвкупността от памет, стек и кода на приложението
- OS използват **процеси (process)**, за да разграничават различните приложения
- **Нишката (thread)** е основната единица, на която OS може да заделя процесорно време
- В един процес различните задачи се изпълняват от отделни нишки
- В един процес има поне една нишка

# Процес vs. Нишка

- Прилики
  - собствен стек (stack)
  - приоритет
  - exception handlers
- Разлики
  - Процесите са изолирани един от друг по отношение на памет и данни
  - Нишките в един процес споделят паметта (променливите и данните) на този процес
  - Процесите съдържат изпълнимия код, а нишките го изпълняват

# Какво предлагат нишките?

- Използването на няколко нишки
  - Създава впечатление за извършване на няколко задачи едновременно
  - Причината: Времето, през което една нишка държи процесора е много кратко
- Например
  - Потребителят може да въвежда данни в текстообработваща програма докато се печатат данни на принтер
  - Би било недопустимо ако потребителят трябаше да изчака принтерът да свърши работата си

# Кога са удобни нишките?

- Случаи, в които е удобно да се използват нишки:
  - Обслужване на много потребители едновременно, например Web сървър
  - Комуникация през мрежа (sockets)
  - Извършване на времеотнемаща операция
  - Различаване на задачите по приоритет
  - За да може потребителският интерфейс да продължи да “отклика” на потребителски заявки, докато на заден план се извършва друга задача

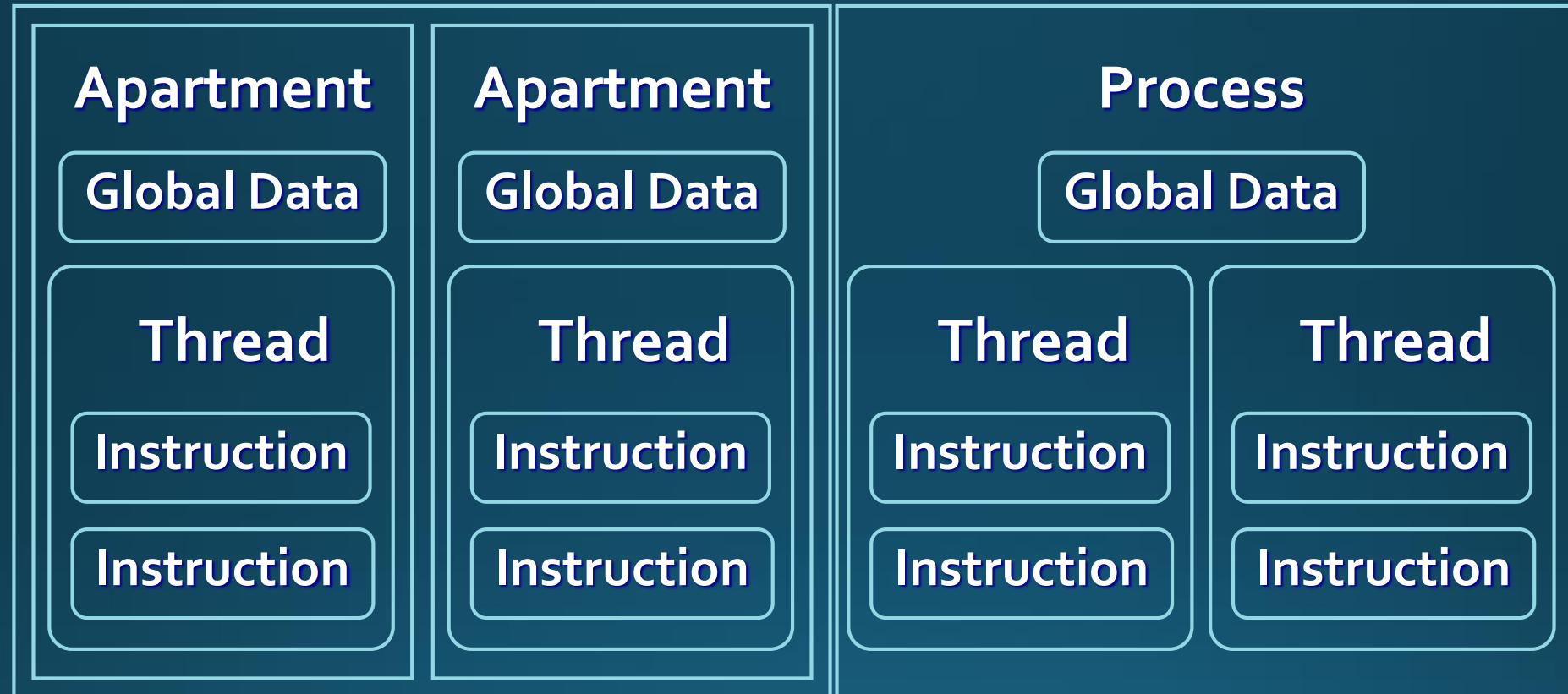
# Многозадачност – Изпреварваща и кооперативна

- Кооперативна многозадачност (cooperative multitasking)
  - Нишката сама решава колко процесорно време ѝ е необходимо
  - Сваля се от процесора само ако е свършила или чака за някакъв ресурс
- Изпреварваща многозадачност (preemptive multitasking)
  - Планировчикът (task scheduler) заделя предварително някакво процесорно време
  - Без значение дали е приключила, нишката се снема от процесора

# Многозадачност – Изпреварваща и кооперативна

- При кооперативната многозадачност:
  - Една нишка може дълго време да държи процесора
  - Заради нея останалите нишки могат да чакат недопустимо дълго
- Някои системи използват комбиниран вариант
  - Нишки с висок приоритет се държат кооперативно спрямо нишките с по-нисък приоритет

# Имплементации на многозадачность



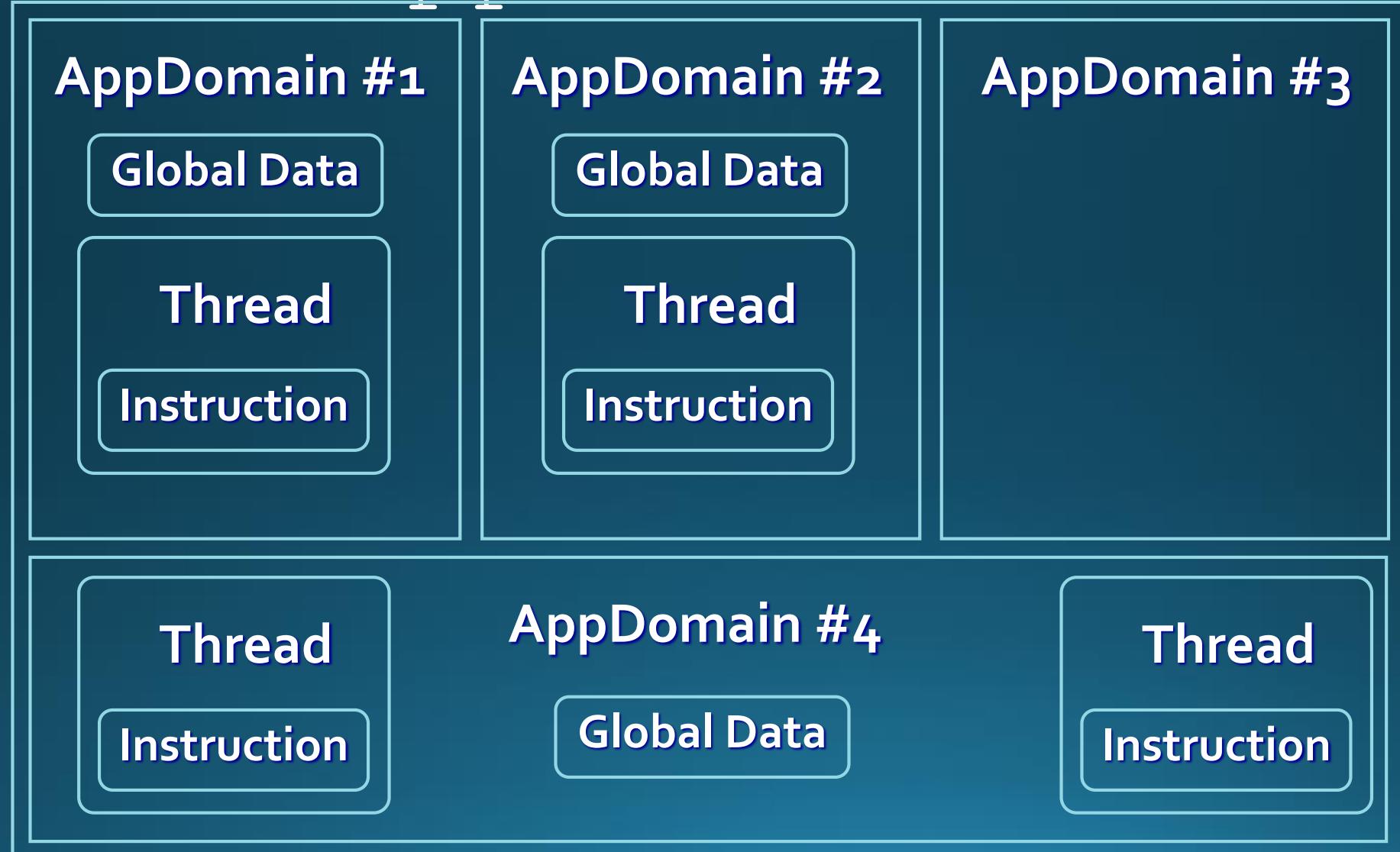
Apartment Threading

Free Threading

# Application Domain

- .NET Framework добавя още едно ниво на абстракция между нишка и процес – Application Domain
- AppDomain-ът дава логическа изолация
  - При процесите изолацията е физическа, тъй като е имплементиран на ниво OS
- Предимства
  - Подобрена производителност
  - Ниво на изолация – като при процеса
  - Връзка м/у нишки без proxy
  - Извършва type checking
- System.AppDomain

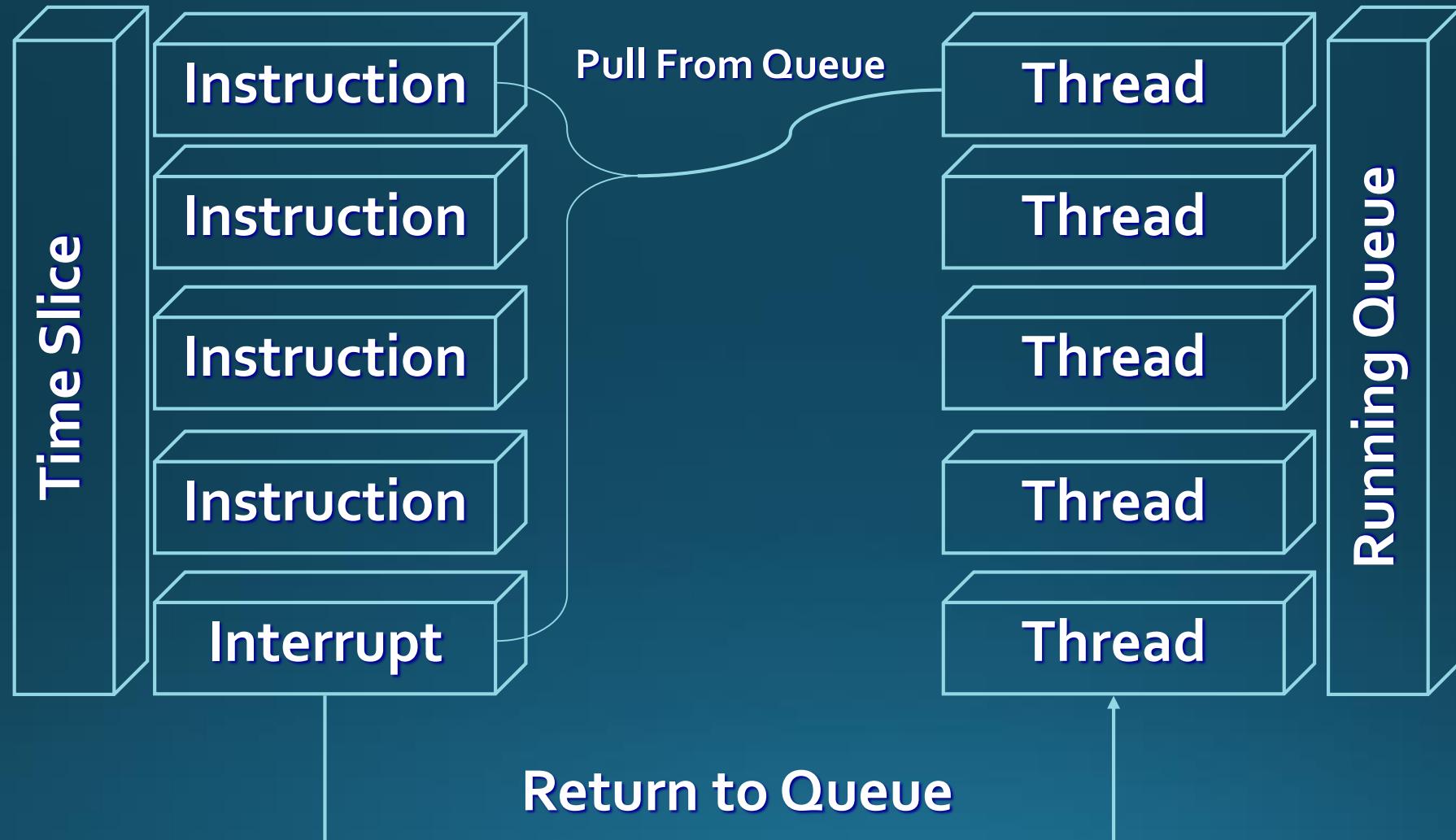
# System.AppDomain



# Нишки – употреба от разработчик

- Многозадачност
- **Нишки**
  - Как работят нишките
  - Живот на нишките
  - По-важни членове
  - Приоритет
  - Състояния
  - Thread Local Storage
  - Thread-Relative Static Fields
  - Прекратяване
  - Неудобства
  - Повреждане на данни
- Синхронизация
- Асинхронни извиквания

# Как работи многонишковостта



# Малък пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter {
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Start();
        t2.Start();
    }
}
```

# Малък пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter {
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Start();
        t2.Start();
    }
}
```

# Малък пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter {
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Start();
        t2.Start();
    }
}
```

# Малък пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter {
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Start();
        t2.Start();
    }
}
```

# Малък пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter {
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Start();
        t2.Start();
    }
}
```

# Малък пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter {
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Start();
        t2.Start();
    }
}
```

# Малък пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter {
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Start();
        t2.Start();
    }
}
```

# Малък пример

```
//Резултат:  
Thread1: job (1)  
Thread1: job (2)  
Thread2: job (1)  
Thread1: job (3)  
...  
...  
Thread1: job (53)  
Thread2: job (46)  
Thread2: job (47)  
Thread1: job (54)  
...  
...  
Thread1: job (97)  
Thread2: job (99)  
Thread1: job (98)  
Thread1: job (99)
```

# Жизнен цикъл на нишките



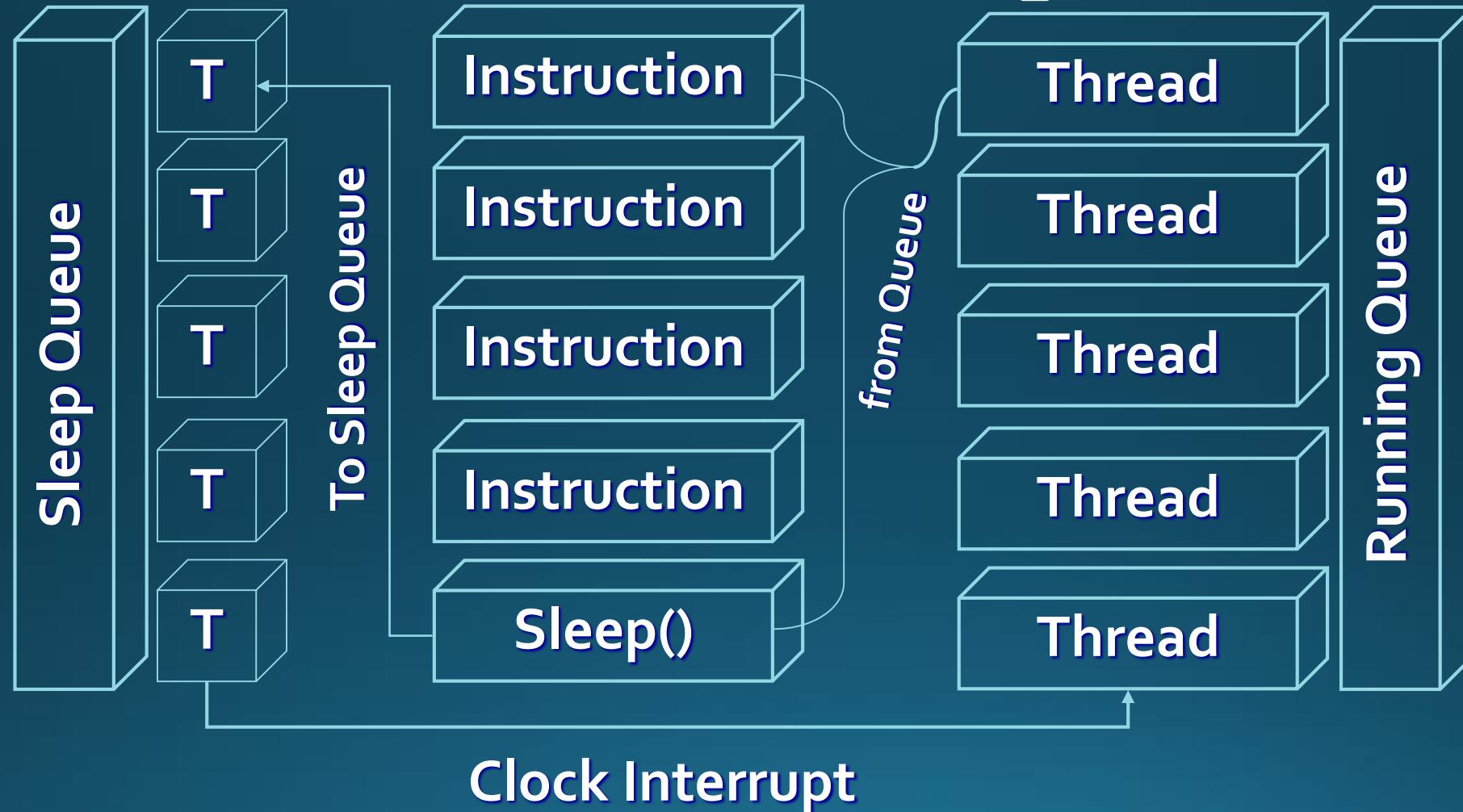
# По-важните членове на Thread

- `public Thread( ThreadStart );`
  - Създава инстанция
  - Подава се делегат с метод, който да се изпълни при стартиране
- `Sleep()`
- `Suspend()`
- `Resume()`
- `IsAlive`
- `IsBackground`
- `IsThreadPoolThread`
- `Name`
- `Priority`
- `ThreadState`
- `Abort()`
- `Interrupt()`
- `Join()`
- `Start()`

# По-важните членове на Thread

- public Thread( ThreadStart start);
- Sleep ()
  - “Приспива” текущата нишка за указания брой милисекунди (и наносекунди)
  - Извиква се само от самата нишка
- Suspend ()
- Resume ()
- IsAlive
- IsBackground
- IsThreadPoolThread
- Name
- Priority
- ThreadState
- Abort ()
- Interrupt ()
- Join ()
- Start ()

# Малко повече за Sleep()



# Приписване на нишка

```
using System;
using System.Threading;
public class ThreadSleep
{
    public static Thread worker;
    public static Thread worker2;
    public static void Main()
    {
        Console.WriteLine("Entering void Main!");
        worker = new Thread(new
            ThreadStart(Counter));
        worker2 = new Thread(new
            ThreadStart(Counter2));
        worker.Start();
        worker2.Start();
        Console.WriteLine("Exiting void Main!");
    }
}                                (примерът продължава)
```

# Приспиване на нишка

```
public static void Counter()
{
    Console.WriteLine("Entering Counter");
    for(int i = 1; i < 50; i++)
    {
        Console.Write(i + " ");
        if(i == 10)
            Thread.Sleep(1000);
    }
    Console.WriteLine("\nExiting Counter");
}
public static void Counter2()
{
    Console.WriteLine("Entering Counter2");
    for(int i = 51; i < 100; i++)
    {
        Console.Write(i + " ");
        if( i == 70 )
            Thread.Sleep(5000);
    }
    Console.WriteLine("\nExiting Counter2");
}
```

# По-важните членове на Thread

- public Thread( ThreadStart start);
- Sleep()
- Suspend()
  - Ако нишката е в състояние Running, я преустановява временно (suspend)
  - ако е преустановена, не се случва нищо
- Resume()
- IsAlive
- IsBackground
- IsThreadPoolThread
- Name
- Priority
- ThreadState
- Abort()
- Interrupt()
- Join()
- Start()

# По-важните членове на Thread

- public Thread( ThreadStart start);
- Sleep()
- Suspend()
- Resume ()
  - Подновява нишка, която е била преустановена (suspended)
  - Ако нишката работи, не прави нищо
- IsAlive
- IsBackground
- IsThreadPoolThread
- Name
- Priority
- ThreadState
- Abort ()
- Interrupt ()
- Join ()
- Start ()

# По-важните членове на Thread

- public Thread( ThreadStart start);
- Sleep()
- Suspend()
- Resume()
- IsAlive
  - true, ако е стартирана и не е спряна прекъсната или прекратена
  - Повече информация дава ThreadState
- IsBackground
- IsThreadPoolThread
- Name
- Priority
- ThreadState
- Abort()
- Interrupt()
- Join()
- Start()

# По-важните членове на Thread

- public Thread( ThreadStart start);
- Sleep()
- Suspend()
- Resume()
- IsAlive
- IsBackground
  - Преден план (foreground) и фонов режим (background)
  - Свойство за смяна/извличане
- IsThreadPoolThread
- Name
- Priority
- ThreadState
- Abort()
- Interrupt()
- Join()
- Start()

# По-важните членове на Thread

```
public Thread(ThreadStart start);
```

- Sleep()
- Suspend()
- Resume()
- IsAlive
- IsBackground
- IsThreadPoolThread
  - Свойство за смяна/извличане
  - true, ако нишката принадлежи на managed thread pool,  
иначе false
- Name
- Priority
- ThreadState
- Abort()
- Interrupt()
- Join()
- Start()

# По-важните членове на Thread

- public Thread( ThreadStart start);
- Sleep()
- Suspend()
- Resume()
- IsAlive
- IsBackground
- IsThreadPoolThread
- Name
  - Всяка нишка в .NET Framework може да има име
  - Свойство за смяна/извличане на името
- Priority
- ThreadState
- Abort()
- Interrupt()
- Join()
- Start()

# По-важните членове на Thread

- public Thread( ThreadStart start);
- Sleep()
- Suspend()
- Resume()
- IsAlive
- IsBackground
- IsThreadPoolThread
- Name
- Priority
  - Lowest, BelowNormal, Normal (по подразбиране), AboveNormal, Highest
  - Свойство за промяна/извличане
- ThreadState
- Abort()
- Interrupt()
- Join()
- Start()

# По-важните членове на Thread

- public Thread( ThreadStart start);
- Sleep()
- Suspend()
- Resume()
- IsAlive
- IsBackground
- IsThreadPoolThread
- Name
- Priority
- ThreadState
  - Съдържа състоянието на нишката – зависи от това дали нишката работи
  - Свойство само за извлечане
- Abort()
- Interrupt()
- Join()
- Start()

# По-важните членове на Thread

- public Thread( ThreadStart start);
- Sleep()
- Suspend()
- Resume()
- IsAlive
- IsBackground
- IsThreadPoolThread
- Name
- Priority
- ThreadState
- Abort()
  - Обикновено “убива” нишката
  - Хвърля ThreadAbortException в извиканата нишка
- Interrupt()
- Join()
- Start()

# По-важните членове на Thread

- public Thread( ThreadStart start);
- Sleep()
- Suspend()
- Resume()
- IsAlive
- IsBackground
- IsThreadPoolThread
- Name
- Priority
- ThreadState
- Abort()
- Interrupt()
  - Събужда нишка ако е в състояние WaitSleepJoin,  
иначе не прави нищо
  - Хвърля ThreadInterruptedException
- Join()
- Start()

# По-важните членове на Thread

```
public Thread(ThreadStart start);
```

- Sleep()
- Suspend()
- Resume()
- IsAlive
- IsBackground
- IsThreadPoolThread
- Name
- Priority
- ThreadState
- Abort()
- Interrupt()
- Join()
  - Извикващата нишка изчаква, докато извиканата приключи
  - Може да се укаже таймаут (timeout)
- Start()

# По-важните членове на Thread

- public Thread( ThreadStart start);
- Sleep()
- Suspend()
- Resume()
- IsAlive
- IsBackground
- IsThreadPoolThread
- Name
- Priority
- ThreadState
- Abort()
- Interrupt()
- Join()
- Start()
  - Стартира посочената нишка
  - Операцията не е блокираща (връща управлението веднага)

# Приоритет на нишка

- Повечето имплементации на многонишковост включват отличителното качество "приоритет"
  - Приоритетът указва колко важна е нишката (колко време да ѝ се отделя)
  - Важен е за планировчика (task scheduler)
- Приоритетът в .NET Framework
  - Възможни стойности
    - Lowest, BelowNormal, Normal (по подразбиране), AboveNormal, Highest
    - В Win32 има и приоритет RealTime
    - OS не е длъжна да се съобразява с приоритета на нишките, но обикновено го прави

# Приоритет – пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter
{
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Priority = ThreadPriority.Highest;
        t1.Start();
        t2.Start();
    }
}
```

# Приоритет

Результат:

Thread1 : job (1)

Thread1 : job (2)

Thread2 : job (1)

Thread1 : job (3)

...

...

Thread1 : job (97)

Thread1 : job (98)

Thread2 : job (85)

Thread1 : job (99)

Thread2 : job (86)

...

...

Thread2 : job (97)

Thread2 : job (98)

Thread2 : job (99)

# ThreadState

- Всяка нишка във всеки един момент е в някое от състоянията на енумерацията ThreadState
- Допуска се да бъде в няколко състояния едновременно
- Енумерацията има атрибут FlagsAttribute
  - Това позволява побитово комбиниране на стойностите ѝ:

```
if ((state & (Unstarted | Stopped)) == 0)  
    // implies Running
```
- Когато се създаде нишка е в състояние Unstarted
  - Остава така докато не се извика Start ()

# Състоянията в ThreadState

- Aborted (256)
  - Извикан е метода Abort ()
  - Нишката е в състояние Stopped, едновременно с Aborted
- AbortRequested (128)
- Background (4)
- Running (0)
- Stopped (16)
- StopRequested (1)
- Suspended (64)
- SuspendRequested (2)
- Unstarted (8)
- WaitSleepJoin (32)

# Състоянията в ThreadState

- Aborted (256)
- AbortRequested (128)
  - Методът Abort () е бил извикан
  - Още не е получила ThreadAbortException, което ще се опита да я прекрати
- Background (4)
- Running (0)
- Stopped (16)
- StopRequested (1)
- Suspended (64)
- SuspendRequested (2)
- Unstarted (8)
- WaitSleepJoin (32)

# Състоянията в ThreadState

- Aborted (256)
- AbortRequested (128)
- Background (4)
  - Нишката е във фонов режим
  - Променя се със свойството `Thread.IsBackground`
- Running (0)
- Stopped (16)
- StopRequested (1)
- Suspended (64)
- SuspendRequested (2)
- Unstarted (8)
- WaitSleepJoin (32)

# Състоянията в ThreadState

- Aborted (256)
- AbortRequested (128)
- Background (4)
- Running (0)
  - Ниската е стартирана и не е блокирана
  - Няма чакащо изключение ThreadAbortedException
- Stopped (16)
- StopRequested (1)
- Suspended (64)
- SuspendRequested (2)
- Unstarted (8)
- WaitSleepJoin (32)

# Състоянията в ThreadState

- Aborted (256)
- AbortRequested (128)
- Background (4)
- Running (0)
- Stopped (16)
  - Нишката е отговорила на заявка от Abort () или
  - Прекратила е работата си доброволно
- StopRequested (1)
- Suspended (64)
- SuspendRequested (2)
- Unstarted (8)
- WaitSleepJoin (32)

# Състоянията в ThreadState

- Aborted (256)
- AbortRequested (128)
- Background (4)
- Running (0)
- Stopped (16)
- StopRequested (1)
  - Поискано е било от нишката да спре работа
  - Само за вътрешна употреба
- Suspended (64)
- SuspendRequested (2)
- Unstarted (8)
- WaitSleepJoin (32)

# Състоянията в ThreadState

- Aborted (256)
- AbortRequested (128)
- Background (4)
- Running (0)
- Stopped (16)
- StopRequested (1)
- Suspended (64)
  - Нишката е била преустановена
  - Независимо колко пъти е извикан Suspend(), един Resume() е достатъчен
- SuspendRequested (2)
- Unstarted (8)
- WaitSleepJoin (32)

# Състоянията в ThreadState

- Aborted (256)
- AbortRequested (128)
- Background (4)
- Running (0)
- Stopped (16)
- StopRequested (1)
- Suspended (64)
- SuspendRequested (2)
  - Извикан е метода Suspend()
  - Изчаква се да стигне до стабилно състояние, за да се преустанови
- Unstarted (8)
- WaitSleepJoin (32)

# Състоянията в ThreadState

- Aborted (256)
- AbortRequested (128)
- Background (4)
- Running (0)
- Stopped (16)
- StopRequested (1)
- Suspended (64)
- SuspendRequested (2)
- Unstarted (8)
  - Не е стартирана, стартира се със Start()
  - След като се стартира нишката никога повече не може пак да е в това състояние
- WaitSleepJoin (32)

# Състоянията в ThreadState

- Aborted (256)
- AbortRequested (128)
- Background (4)
- Running (0)
- Stopped (16)
- StopRequested (1)
- Suspended (64)
- SuspendRequested (2)
- Unstarted (8)
- WaitSleepJoin (32)
  - Ниската е блокирана
  - Става с един от методите `Thread.Wait()`, `Thread.Sleep()`, `Thread.Join()`

# Прекратяване на нишки

- За да се “убие” една нишка, се използва методът `Thread.Abort()`
- `Thread.Abort()` хвърля специалното изключение `ThreadAbortedException`
  - Хвърля `ThreadStateException`, ако нишката вече е прекратена
- Изпълняват се `catch` и `finally` на прекратената нишка
- `Thread.ResetAbort()` не позволява на CLR да хвърли изключението отново
- Unmanaged code – изчакване

# Прекратяване на нишки – пример

```
public class BackgroundThread {  
    public void DoBackgroundJob() {  
        try {  
            // Нишката извършва някаква работа  
        }  
        catch(ThreadInterruptedException) {  
            MessageBox.Show("Thread interrupted.");  
        }  
        catch(ThreadAbortException) {  
            MessageBox.Show("Thread aborted.");  
        }  
        finally {  
            MessageBox.Show("Finally block.");  
        }  
        MessageBox.Show("After finally block.");  
    }  
}
```

(примерът продължава)

# Прекратяване на нишки – пример

```
public class InterruptAbortDemo
{
    private Thread mBgThread;

    public InterruptAbortDemo() {
        bg = new BackgroundThread();
        mBgThread = new Thread(new
            ThreadStart(bg.DoBackgroundJob));
        mBgThread.IsBackground = true;
        mBgThread.Start();
    }

    private void btnSuspend_Click(...)
    {
        mBgThread.Suspend();
    }
}
```

*(примерът продължава)*

# Прекратяване на нишки – пример

```
private void btnResume_Click(...)  
{  
    mBgThread.Resume();  
}  
  
private void btnInterrupt_Click(...)  
{  
    mBgThread.Interrupt();  
}  
  
private void btnAbort_Click(...)  
{  
    mBgThread.Abort();  
}
```

# Thread Local Storage

- Контейнер, в който нишките могат да съхраняват данни
- Нишка не може да достъпи данните на друга нишка
- `Thread.AllocateNamedDataSlot`
  - Именуван контейнер
  - Проблеми – уникални имена
- `Thread.AllocateDataSlot`
  - Няма име, само `reference`
- Проблеми: две парчета код от една и съща нишка използват общ слот

# Thread Local Storage – пример

```
class Threads {
    public void CreateDataThread() {
        LocalDataStoreSlot slot =
            Thread.AllocateNamedDataSlot("mySlot");

        // Записваме важни данни в NamedSlot
        Thread.SetData(slot , "IMPORTANT DATA");

        // Suspend-ваме
        Thread.CurrentThread.Suspend();

        // Прочитаме отново данните
        object myData = Thread.GetData("mySlot");
    }

    public void ReadDataThread() {
        LocalDataStoreSlot slot =
            Thread.GetNamedDataSlot("mySlot");

        // Опитваме се да променим информацията
        Thread.SetData(slot, "BAD DATA");
    }
}
```

(примерът продължава)

# Thread Local Storage – пример

```
class TLSDemo
{
    static void Main(string[] args)
    {
        Threads threads = new Threads();
        Thread createData = new Thread( new
            ThreadStart(threads.createDataThread) );
        createData.Start();

        Thread readData = new Thread( new
            ThreadStart(threads.readDataThread) );
        readData.Start();
        readData.Join();
        createData.Resume();
    }
}
```

# Thread-Relative Static Fields

- Като статична променлива, но има по една инстанция за всяка нишка на програмата
- Декларира се като статична променлива, но с атрибута [ThreadStatic]
- Всяка нишка, която я използва трябва да я инициализира
- Не разчитайте на конструктора да инициализира такива променливи, защото той се изпълнява в родителската нишка!

# Thread-Relative Static Fields – пример

```
class ThreadStatic {
    static void Main(string[] args) {
        for( int i=0; i<10; i++ ) {
            new Thread(new ThreadStart(new
                MyThread() .DoTask)) .Start();
        }
    }
}

class MyThread {
    [ThreadStatic] public static int abc;
    public MyThread() {
        abc=42; // This runs in the main app. thread
    }
    public void DoTask() {
        abc++;
        Console.WriteLine("abc={0}" , abc);
    }
}
```

# Неудобства на нишките

- Самата имплементация на многонишковост изисква ресурси
- Добра практика е да не прекалявате с използването на нишки
  - Възможно е процесорът да използва повече ресурси за управлението на нишките отколкото за изпълнението им
- Препоръчва се да използвате ThreadPool когато е възможно
- Следенето на много нишки е трудно

# Споделени данни – проблеми

- Race condition
  - Две нишки едновременно достъпват едни и същи данни
  - Непредвидими резултати
  - Например две нишки едновременно изпълняват `i++` на обща променлива `i`
- Deadlock
  - Състояние, в което две нишки се чакат взаимно за освобождаване на взаимно заети ресурси
  - А заема ресурса X, В заема ресурса Y и чака за X, а A започва да чака за Y

# Race Condition – пример

```
using System;
using System.Threading;

class Account
{
    public int mBalance;

    public void Withdraw100()
    {
        int oldBalance = mBalance;
        Console.WriteLine("Withdrawning 100...");
        Thread.Sleep(100);
        int newBalance = oldBalance - 100;
        mBalance = newBalance;
    }
}
```

(примерът продължава)

# Race Condition – пример

```
static void Main(string[] args)
{
    Account acc = new Account();
    acc.mBalance = 500;
    Console.WriteLine("Account balance = {0}",
                      acc.mBalance);
    Thread user1 = new Thread(
        new ThreadStart(acc.Withdraw100) );
    Thread user2 = new Thread(
        new ThreadStart(acc.Withdraw100) );
    user1.Start();
    user2.Start();
    user1.Join();
    user2.Join();
    Console.WriteLine("Account balance = {0}",
                      acc.mBalance);
}
```

# Синхронизация – съдържание

- Многозадачност
- Нишки
- Синхронизация
  - Най-доброто решение
  - “Стратегии” за синхронизация
  - Synchronized Contexts
  - Synchronized code regions
  - MethodImplAttribute
  - Unmanaged Synchronization – WaitHandle
  - Специални класове
    - Класически синхронизационни проблеми
    - ThreadPool
- Асинхронни извиквания

# Най-доброто решение

- Най-добра сигурност при споделени данни в многонишкова среда е липсата на споделени данни
- Препоръчва се
  - Да се капсулират данните и ресурсите в инстанции на обекти
  - Тези обекти да не се спodelят между нишки
- Когато това е невъзможно, .NET Framework, предоставя механизми за синхронизация на достъпа до данни

# “Стратегии” за синхронизация

- Синхронизирани контексти (Synchronized Contexts)
- Синхронизирани “пасажи” код (Synchronized code regions)
- [MethodImplAttribute  
(MethodImplOptions.Synchronized) ] над  
имплементацията на метод
- Unmanaged синхронизация
  - Класове за фина, ръчно настроена (custom) синхронизация с  
обекти на OS

# Synchronized Contexts

- Използва се SynchronizationAttribute за обекти, наследяващи ContextBoundObject
  - обекти, които оперират в един контекст
- Всички нишки в този контекст достъпват методите на инстанции на такива обекти последователно
- Статичните членове не са предпазени
- Не се поддържа синхронизиране на специфични “отрязъци” от код – синхронизира цял клас

# Synchronized Contexts – пример

```
[SynchronizationAttribute]
class CBO : ContextBoundObject
{
    public void DoSomeTask1()
    {
        Console.WriteLine("Job1 started.");
        Thread.Sleep(2000);
        Console.WriteLine("Job1 finished.\n");
    }

    public void DoSomeTask2()
    {
        Console.WriteLine("Job2 started.");
        Thread.Sleep(1500);
        Console.WriteLine("Job2 finished.\n");
    }
}
```

*(примерът продължава)*

# Synchronized Contexts – пример

```
static void Main()
{
    CBO syncClass = new CBO();
    Console.WriteLine("Started 6 threads:\n" +
        "3 doing Job1 and 3 doing Job2.\n\n");
    for (int i=0; i<6; i++)
    {
        Thread t;
        if (i%2==0)
            t = new Thread( new ThreadStart(
                syncClass.DoSomeTask1) );
        else
            t = new Thread( new ThreadStart(
                syncClass.DoSomeTask2) );
        t.Start();
    }
}
```

# Критични секции

```
lock (obj) {  
    //...code..  
}
```

```
Monitor.Enter(obj);  
try { /*...code...*/ }  
finally { Monitor.Exit(obj); }
```

- ◆ Управлявана (managed) реализация
- ◆ Може да се ползва и за статични членове на класовете
- ◆ Синхронизира се изпълнението само частта, която е застрашена
  - ◆ Нарича се критична секция
- ◆ `Monitor.TryEnter();`

# Критични секции – пример

```
public class MonitorEnterExit
{
    private int mCounter;

    public void CriticalSection()
    {
        Monitor.Enter(this);
        mCounter = 0;
        try
        {
            for(int i = 1; i <= 5; i++) {
                Console.Write(++mCounter);
                Thread.Sleep(1000);
            }
        }
        finally
        {
            Monitor.Exit(this);
        }
    }
}
```

*(примерът продължава)*

# Критични секции – пример

```
static void Main() {  
    MonitorEnterExit mee = new MonitorEnterExit();  
    Thread thread1 = new Thread(new  
        ThreadStart(mee.CriticalSection));  
    thread1.Start();  
    Thread thread2 = new Thread(new  
        ThreadStart(mee.CriticalSection));  
    thread2.Start();  
}  
}  
  
// Результат: 1234512345  
  
// Результат без синхронизация: 1123456789
```

# Няколко важни метода

- `Monitor.Wait(object)`
  - Освобождава монитора на обекта и блокира нишката докато не го получи
  - Може да се задава Timeout
  - Нишката се нарежда на waiting queue
  - Чака `Pulse(object)`, `PulseAll(object)`
- `Monitor.Pulse(object)`
  - Нишката преминава в ready queue (има право да вземе монитора на обект)
  - Вика се само от текущия собственик на монитора на обекта (от критична секция)
- `Monitor.PulseAll(object)`

# Wait () и Pulse () – пример

- Имаме две нишки, които съвместно изпълняват някаква задача
- Когато едната прави нещо, другата я изчаква и обратното

```
public class WaitPulse
{
    private object mSync;
    private string mName;

    public WaitPulse(string aName,
                     object aSync)
    {
        mName = aName;
        mSync = aSync;
    }
}
```

*(примерът продължава)*

# Wait() и Pulse() – пример

```
public void DoJob()
{
    lock (mSync)
    {
        Monitor.Pulse(mSync);
        for(int i = 1; i <= 3; i++)
        {
            Console.WriteLine("{0}: Pulse", mName);
            Monitor.Pulse(mSync);

            Console.WriteLine("{0}: Wait", mName);
            Monitor.Wait(mSync);

            Console.WriteLine("{0}: WokeUp", mName);
            Thread.Sleep(1000);
        }
    }
}
```

(примерът продължава)

# Wait() и Pulse() – пример

```
public class WaitPulseDemo
{
    public static void Main(String[] args)
    {
        object sync = new object();

        WaitPulse wp1 = new WaitPulse(
            "WaitPulse1", sync);
        Thread thread1 = new Thread(
            new ThreadStart(wp1.DoJob));
        thread1.Start();

        WaitPulse wp2 = new WaitPulse(
            "WaitPulse2", sync);
        Thread thread2 = new Thread(
            new ThreadStart(wp2.DoJob));
        thread2.Start();
    }
}
```

## [MethodImplAttribute (MethodImplOptions.Synchronized) ]

- Прилича на lock върху цял метод
- Може да синхронизира и static членове

```
[MethodImpl(MethodImplOptions.Synchronized)]
public void DoSomeTask1()
{
    Console.WriteLine("job1 started");
    for (int i=0; i<1000000000; i++)
        Math.Sqrt(i);
    Console.WriteLine("Job1 done.\n");
}
```

# Unmanaged Synchronization

- WaitHandle
  - Unmanaged механизъм за синхронизация
  - Използва обекти на операционната система – за изчакване на ресурси
  - Предоставя възможности отвъд тези на CLR – WaitAll (), WaitAny ()
  - Не се препоръчва – не е managed
- Има няколко наследника:
  - Mutex
  - AutoResetEvent
  - ManualResetEvent

# Класът WaitHandle

- Неговите методи се използват за изчакване на събития
- Състояния: signaled, nonsignaled
- WaitAll () (static)
  - Изчаква сигнал от всички събития от масив
- WaitAny () (static)
  - Изчаква първото получило сигнал
- WaitOne ()
  - Изчаква текущото събитие да приключи
- Прилики и разлики с класа Monitor

# Mutex

- Наследник на WaitHandle
- Примитив за синхронизация на OS
- Енкапсулира Win32 synchronization handles
- Mutex.WaitOne();
- Mutex.ReleaseMutex();
  - Трябва да се извика, същия брой пъти като Mutex.WaitOne();
- WaitHandle.WaitAll() – изчаква два или няколко handle-а

# Mutex – пример

```
class MutexDemo
{
    Mutex mMutex;
    public MutexDemo(Mutex aMutex)
    {
        mMutex = aMutex;
    }
    public void PerformSomeTask()
    {
        m.WaitOne();
        Console.WriteLine("\nJob started...");
        for( int i=0; i<10; i++)
        {
            Thread.Sleep(100);
            Console.Write(" | ");
        }
        Console.WriteLine("\nJob finished.");
        m.ReleaseMutex();
    }
}
```

# AutoResetEvent, ManualResetEvent

- Наследници на WaitHandle
- Примитиви за синхронизация
- Енкапсулират Win32 synchronization handles
- Сигнализиране със Set ()
  - AutoResetEvent сигнализира само първия манипулятор(handle)
    - Минава автоматично в nonsignaled state
  - ManualResetEvent () сигнализира всички чакащи манипулятори
    - Остава в signaled state, докато някой не го промени

# AutoResetEvent/ManualResetEvent

```
using System;
using System.Threading;
class OneWhoWaits
{
    WaitHandle mWaitHandle;
    int mWaitTime;
    public OneWhoWaits(WaitHandle aWaitHandle,
        int waitTime )
    {
        mWaitHandle = aWaitHandle;
        mWaitTime = mWaitTime;
    }
    public void performSomeTask()
    {
        Thread.Sleep(mWaitTime);
        Console.WriteLine("Thread {0} waiting",
            Thread.CurrentThread.GetHashCode());
        mWaitHandle.WaitOne();
    }
}
```

*(примерът продължава)*

# AutoResetEvent/ManualResetEvent

```
class MainClass
{
    static void Main()
    {
        ManualResetEvent evnt = new
            ManualResetEvent(false);
        for (int i=0; i<10; i++ )
        {
            OneWhoWaits oww =
                new OneWhoWaits(evnt, (i+1)*500);
            Thread thread = new Thread(
                new ThreadStart(oww.performSomeTask));
            thread.Start();
        }
        for (int i=0; i<10; i++)
        {
            Console.ReadLine();
            evnt.Set();
        }
    }
}
```

# Специални класове

- System.Threading.Interlocked
  - Атомарни операции
- System.Threading.ThreadPool
  - Бърз достъп до нишки
  - Преизползваемост на нишките
  - Програмистът не се грижи за живота на нишките (създаване / унищожаване)
- System.Threading.ReaderWriterLock
  - Класически синхронизационни проблеми – Reader/Writer Problem
- Synchronized Wrappers

# System.Threading.Interlocked

- Осигурява изпълнение на атомарни операции –  
увеличение с 1, намаление с 1, размяна, сравнение  
и др.
- Няма нужда синхронизация на споделените данни
- В по-простите случаи синхронизацията може да  
се избегне с тези методи
- Методите не хвърлят изключения

# System.Threading.Interlocked

- Increment/Decrement
  - Атомарна операция за разлика от `i++/i--`
- Exchange
  - Записва стойността на вторият параметър в първия, връща оригинала
- CompareExchange
  - Има три параметъра
  - Проверява дали първият и третият са равни, ако да, записва втория в първия
  - Използва се при работа с временни променливи

# Interlocked – пример

```
class TestInterlockedIncrement
{
    static long mUnsafeCounter = 0;
    static long mSafeCounter = 0;

    private static void DoTask()
    {
        while (true)
        {
            mUnsafeCounter++;
            Interlocked.Increment(ref mSafeCounter);

            if (mSafeCounter % 10000000 == 0)
            {
                Console.WriteLine("Safe={0}, Unsafe={1}",
                    mSafeCounter, mUnsafeCounter);
            }
        }
    }
}
```

# Interlocked – пример

```
static void Main(string[] args)
{
    for (int i=0; i<5; i++)
    {
        Thread thread = new Thread(
            new ThreadStart(DoTask));
        thread.Start();
    }
}

// Результат:
// Safe=10000000, Unsafe=5846325
// Safe=20000000, Unsafe=15846326
// Safe=30000000, Unsafe=25846326
// Safe=40000000, Unsafe=35846325
// Safe=50000000, Unsafe=41356463
// ...
```

# Thread Pooling

- Много нишки прекарват по-голямата част от живота си спейки (`ThreadState.WaitSleepJoin`) или чакайки някакво събитие
- Други се “събуждат” само за малки периоди, за да проверят истинността на някакво условие или за да свършат нещо дребно
- Поддържането на много неактивни нишки е излишно, неефективно и консумира ресурси

# Thread Pooling

- TP е подход за намаляване на товара при създаване и унищожаване на нишки
- При TP се създават група нишки в началото на многонишково приложение
- Наричат се “работни нишки” (worker threads)
- Всяка нишка “живее” в т.нр. Thread Pool
- При нова задача, се използва worker thread, след това се връща обратно в пула
- TP се грижи за живота и разпределението на задачите върху нишките
  - Използваният нишки се освобождава от този товар

## System.Threading.ThreadPool

- .NET Framework имплементира механизма Thread Pooling в класа ThreadPool
- Използва се, когато трябва да се свършат много на брой кратки задачи, които могат да работят паралелно
- Задачите се нареджат на опашка и се изпълняват по няколко едновременно (според броя worker threads)
- По подразбиране в Thread Pool-ът има лимит от 25 нишки на процесор
- Един процес може да има само един Thread Pool – той е общ за всички App Domains

## System.Threading.ThreadPool

- Thread Pool-ът за даден процес се създава се при първото:
  - извикване на QueueUserWorkItem
  - регистриране на таймер
  - регистриране на изчакваща операция с callback метод
- .NET Framework използва Thread Pooling за:
  - асинхронни извиквания
  - асинхронен вход/изход
  - работа с таймери
  - работа със сокети
  - изчакващи операции

## System.Threading.ThreadPool

- Ако дадена задача е в Thread Pool, не може да се премахне от него
- Thread Pooling позволява на OS да оптимизира използването на нишки, тъй като броят им е постоянен
- Случаи, в които НЕ се препоръчва използването на Thread Pool
  - Ако е нужна контролираща нишка
  - При задачи, отнемащи много време – може да се блокират другите задачи
  - Ако има нужда от синхронизация

# ThreadPool.QueueUserWorkItem

```
class ThreadPoolDemo
{
    public static void LongTask(object aParam)
    {
        Console.WriteLine("Started: {0}.", aParam);
        Thread.Sleep(500);
        Console.WriteLine("Finished: {0}.", aParam);
    }

    static void Main()
    {
        for (int i=1; i<=100; i++)
        {
            string taskName = "Task" + i;
            ThreadPool.QueueUserWorkItem(new
                WaitCallback(LongTask), taskName);
        }

        Console.ReadLine();
    }
}
```

# ThreadPool.RegisterWaitForSingleObject

- Регистрира делегат, който чака за събитие (да бъде сигнализиран)
- Активира се при:
  - сигнализиране
  - при настъпване на зададен timeout
- Процесът по изчакване и извикване се управлява се от Thread Pool-а
- Може да настрои да се изпълнява делегатът многократно

## ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev =new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes;cause =
{0}", cause);
    }
}
```

## ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev =new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes;cause =
{0}", cause);
    }
}
```

## ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev =new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes;cause =
{0}", cause);
    }
}
```

## ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev =new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes;cause =
{0}", cause);
    }
}
```

## ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev =new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes;cause =
{0}", cause);
    }
}
```

## ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev =new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes;cause =
{0}", cause);
    }
}
```

## ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev =new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes;cause =
{0}", cause);
    }
}
```

## ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev =new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes;cause =
{0}", cause);
    }
}
```

## ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev =new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes;cause =
{0}", cause);
    }
}
```

## ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev =new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes;cause =
{0}", cause);
    }
}
```

## ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev =new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes;cause =
{0}", cause);
    }
}
```

# Класически синхр. проблеми

- The Producer-Consumer Problem
  - Един консуматор и един производител споделят обща опашка (shared queue)
  - Изисквания:
    - Ако опашката е празна, консуматорът блокира докато се появи нещо в нея
    - Ако опашката е пълна, производителят блокира докато не се опразни място
    - Производителят и консуматорът не могат да достъпват опашката едновременно
  - Пример: пощенска кутия
  - Нарича се още *bounded buffer problem*
- Няма стандартен клас в .NET

# Класически синхр. проблеми

- The Readers-Writers Problem
  - Един или повече четеца и един или повече писеца достъпват общ ресурс
  - Пример: достъп до общ файл в OS
  - Условия на Бернщайн:
    - Няколко четци могат да достъпват общия ресурс
    - Когато писец достъпва ресурса, нито четец нито друг писец може да го достъпва
    - Очаква се нито един писец/четец да не чака безкрайно дълго за споделения ресурс
  - В .NET – ReaderWriterLock

## System.Threading.ReaderWriterLock

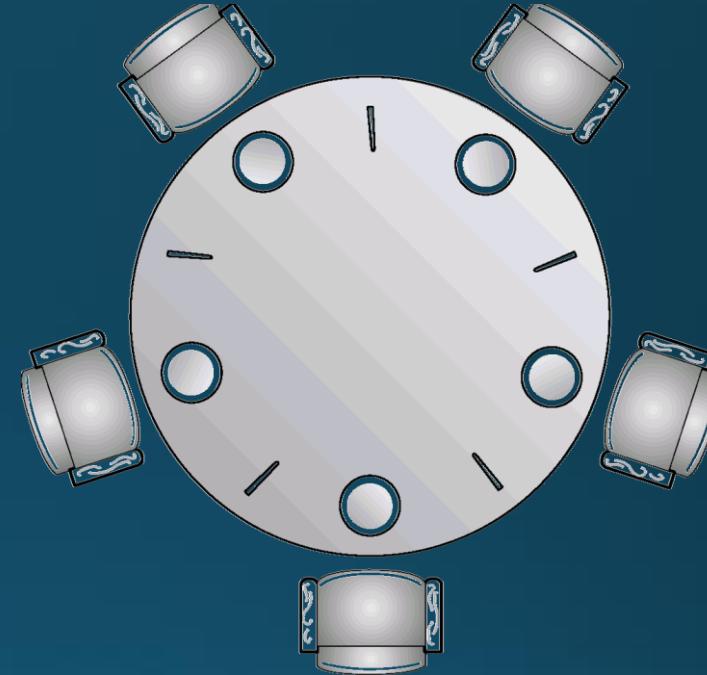
- Имплементира най-популярния синхронизационен проблем – Reader/Writer Problem
- IsReaderLockHeld
- IsWriterLockHeld
- AcquireReaderLock
- AcquireWriterLock
- ReleaseReaderLock
- ReleaseWriterLock

## System.Threading.ReaderWriterLock

```
class Resource {
    ReaderWriterLock rwl = new ReaderWriterLock();
    public void Read() {
        rwl.AcquireReaderLock(Timeout.Infinite);
        try {
            // Many can read, writers are blocked
        }
        finally {
            rwl.ReleaseReaderLock();
        }
    }
    public void Write() {
        rwl.AcquireWriterLock(Timeout.Infinite);
        try {
            // One can write, readers blocked
        }
        finally {
            rwl.ReleaseWriterLock();
        }
    }
}
```

# Класически синхр. проблеми

- The Dining Philosophers Problem
  - Няколко философа стоят около кръгла маса и се хранят или мислят
  - Когато се хранят, се нуждаят и от двете клечки от двете им страни
  - Условия:
    - Не трябва никой философ да умре от глад
    - Не трябва да се позволява deadlock – когато всеки философ има по една клечка и чака за другата



# CustomReaderWriter

```
class CustomReaderWriter
{
    private int mReaders = 0;
    private bool mIsWriting = false;
    public void Read()
    {
        lock( this )
        {
            while (mIsWriting) Monitor.Wait( this );
            mReaders++;
        }

        //...READING TAKES PLACE HERE...

        lock( this )
        {
            mReaders--;
            if( mReaders == 0 ) Monitor.Pulse(this);
        }
    }
}
```

*(примерът продължава)*

# CustomReaderWriter

```
class CustomReaderWriter
{
    private int mReaders = 0;
    private bool mIsWriting = false;
    public void Read()
    {
        lock( this )
        {
            while (mIsWriting) Monitor.Wait( this );
            mReaders++;
        }

        //...READING TAKES PLACE HERE...

        lock( this )
        {
            mReaders--;
            if( mReaders == 0 ) Monitor.Pulse(this);
        }
    }
}
```

*(примерът продължава)*

# CustomReaderWriter

```
class CustomReaderWriter
{
    private int mReaders = 0;
    private bool mIsWriting = false;
    public void Read()
    {
        lock( this )
        {
            while (mIsWriting) Monitor.Wait( this );
            mReaders++;
        }

        //...READING TAKES PLACE HERE...

        lock( this )
        {
            mReaders--;
            if( mReaders == 0 ) Monitor.Pulse(this);
        }
    }
}
```

*(примерът продължава)*

# CustomReaderWriter

```
class CustomReaderWriter
{
    private int mReaders = 0;
    private bool mIsWriting = false;
    public void Read()
    {
        lock( this )
        {
            while (mIsWriting) Monitor.Wait( this );
            mReaders++;
        }

        //...READING TAKES PLACE HERE...

        lock( this )
        {
            mReaders--;
            if( mReaders == 0 ) Monitor.Pulse(this);
        }
    }
}
```

*(примерът продължава)*

# CustomReaderWriter

```
public void Write()
{
    lock( this )
    {
        while( mReaders != 0 )
            Monitor.Wait( this );
        mIsWriting = true;
    }

    //...WRITING TAKES PLACE HERE...

    lock( this )
    {
        mIsWriting = false;
        Monitor.PulseAll( this );
    }
}
```

# CustomReaderWriter

```
public void Write()
{
    lock( this )
    {
        while( mReaders != 0 )
            Monitor.Wait( this );
        mIsWriting = true;
    }

    //...WRITING TAKES PLACE HERE...

    lock( this )
    {
        mIsWriting = false;
        Monitor.PulseAll( this );
    }
}
```

# CustomReaderWriter

```
public void Write()
{
    lock( this )
    {
        while( mReaders != 0 )
            Monitor.Wait( this );
        mIsWriting = true;
    }

    //...WRITING TAKES PLACE HERE...

    lock( this )
    {
        mIsWriting = false;
        Monitor.PulseAll( this );
    }
}
```

# CustomReaderWriter

```
public void Write()
{
    lock( this )
    {
        while( mReaders != 0 )
            Monitor.Wait( this );
        mIsWriting = true;
    }

    //...WRITING TAKES PLACE HERE...

    lock( this )
    {
        mIsWriting = false;
        Monitor.PulseAll( this );
    }
}
```

# Synchronized Wrappers

- По подразбиране стандартните колекции (`ArrayList`, `Queue`, `Stack`) не са `thread safe`
- `Hashtable` е `thread safe` (1 писач, N четеца)
- `Synchronized Wrapper` е синхронизирана обвивка около колекцията
  - Връща синхронизирана версия на колекцията

```
Stack safeStack = Stack.Synchronized(stack);
```

- При `Hashtable` – threadsafe (N писача, N четеца)
- Свойството `ICollection.SyncRoot`

```
lock (bitArray.SyncRoot)
{ // Perform thread unsafe operations here }
```

# Нишки – препоръчани практики

- Избягвайте нуждата от синхронизация, когато е възможно
- Използвайте `Interlocked` вместо синхронизация, където е възможно
- Пазете се от deadlocks
- Избягвайте `lock ( . . ) { }`, ако е възможно
- Правете критичните секции възможно най-кратки (за да намалите изчакването)
- Член-променливите не е нужно да са `thread safe`, ако не се достъпват от други нишки
- Статичните членове е нужно да са `thread safe`
- Използвайте статични данни само за четене (`read-only`)

# Асинхронни извиквания – съдържание

- Многозадачност
- Нишки
- Синхронизация
- Асинхронни извиквания
  - Асинхронни извиквания на методи
  - Къде се поддържа?
  - Асинхронно извикване чрез делегат
  - Design Pattern за асинхр. извиквания
  - Интерфейсът IAsyncResult
  - Резултат от асинхронен метод

# Асинхронни извиквания на методи

- По подразбиране един метод се дефинира като синхронен (*synchronous*)
- Синхронно извикване на метод (*synchronous method call*)
  - Изчаква се неговото приключване и след това се преминава към следващия оператор
- Асинхронно извикване на метод (*asynchronous method call*)
  - Без да се чака приключването на метода, се минава на следващия оператор
- Механизмът на асинхронното извикване използва нишки (*ThreadPool*)

# Къде се поддържа?

- Вход/изход: File IO, Stream IO, Socket IO
- Мрежови класове: HTTP, TCP
- Remoting канали (HTTP, TCP) и проксита
- ASP.NET XML Web-услуги
- ASP.NET Web-приложения
- При работа с MSMQ (Microsoft Message Queue)
- Асинхронни делегати
- Потребителски класове, дефиниращи асинхронен интерфейс за извикване

# Асинхронно извикване чрез делегат

- Делегатите предоставят функционалност за асинхронно извикване на синхронен метод
  - Автоматично се генерираат `BeginInvoke()` и `EndInvoke()` с правилния брой параметри
  - Генерирането става при създаването на делегат със съответната сигнатура
- Добавя се функционалност без да се изисква нито ред допълнителен код

## Асинхронно извикване чрез делегат

```
using System;
using System.Threading;

class AsyncCall
{
    public delegate int MyDelegate(int a, int b);

    public int Sum(int a, int b)
    {
        Thread.Sleep(3000);
        return a+b;
    }

    static void Main(string[] args)
    {
        MyDelegate asyncCall = new MyDelegate(new
            AsyncCall().Sum);
        Console.WriteLine("Starting method async.");
    }
}
```

## Асинхронно извикване чрез делегат

```
IAsyncResult status =
    asyncCall.BeginInvoke(5, 6, null, null);
Console.WriteLine("Async method is working");

Console.WriteLine("Calling EndInvoke()");
int result = asyncCall.EndInvoke(status);
Console.WriteLine("EndInvoke() returned");
Console.WriteLine("Result={0}", result);
}

// Резултат:
Starting method asynchronously
Asynchronous method is working now
Calling EndInvoke() // тук има пауза, заради Sleep()
EndInvoke() returned
Result=11
```

## JAVA THREAD POOL

Java Thread Pool е имплементиран след версия 1.5. Основен клас е класа `ThreadPoolExecutor` в който предварително се настройват колко нишки ще бъдат използвани чрез параметрите `corePoolSize` и `maximumPoolSize`. Възможна е употребата на няколко инстанции. Новите нишки се създават чрез `ThreadFactory`. Разработчика има възможност да дефинира името, групата, приоритета и статуса на нишките. Организацията на pool-а е такава, че се преценява, кои задачи да бъдат поставени в опашка в случай, че всички нишки са заети или да бъдат създадени нови, които да поемат обработката - `Executor.defaultThreadFactory()`. Съществуват методи за мониторинг на pool-а : `getActiveCount`, `getCompletedTaskCount`, `getLargestPoolSize`, `getPoolSize`, `getQueue`, `getTaskCount`. Тези наблюдения дават възможност на разработчика да се намеси в изпълнението на задачите в зависимост от ситуацията.

## NET THREAD POOL

.NET Thread Pool е имплементиран и се на намира в namespace System.Threading. За разлика от Java Thread Pool е статичен и в едно приложение може да разполагаме само с един Thread Pool. Целта е да се централизира обработката и да не се позволява загуба в производителността (при наличието на повече от един се наблюдава подобно явление). Използват се два вида нишки : Worker Threads и Completion Port Threads. Първия тип е стандартен и се използват често за всякакъв вид задачи, а втория най-често за входно/изходни бавни процеси и се появяват след Windows 2000. Подобно на Java Thread Pool и в .NET съществува възможност за настройване на нишките. SetMaxThreads, SetMinThreads, както и методи за наблюдение – значително по-ограничени от Java. GetMaxThreads и GetMinThreads

## РАЗЛИКИ МЕЖДУ JAVA THREAD POOL И .NET THREAD POOL

Класовете в двете среди са създадени с една и съща цел да разпределят процесите в дадено приложение, като използват максимално ресурсите на дадена машина, но реализацията и начина на изпълнение се различават доста. Основна разлика между двете имплементации е че .NET ThreadPool е статичен и разполагаме само с един в приложението, което води до предотвратяване на асинхронизация, докато в Java Thread Pool не е така. Това освен предимства има и своите недостатъци, тъй като отговорността за оптималното използване на ресурсите се прехвърля до голяма степен на разработчика. В .NET не е необходима първоначална инициализация на Thread Pool. Не се определят задължително брой нишки и максимален размер тъй като те имат стойности по подразбиране за разлика от Java Thread Pool където е задължителна инициализацията. Всеки конструктор използван за управлението на java pool изисква стойностите на corePoolSize и maximumPoolSize.

Основно предимство на .NET Pool е наличието на Completion Port нишките, които са подходящи за бавни и дълги процеси. .NET сам избира, коя нишка да пусне в зависимост от поетата задача. За съжаление Java Thread Pool не предлага аналог. Въпреки редица недостатъци Java Thread Pool има доста предимства пред .NET Thread Pool, тъй като разполага с методи за наблюдение на pool-а.

Могат да се проследят какъв брой задачи са подадени на pool-а до момента, колко са били отхвърлени и колко са изпълнени, каква е максималната големина pool-а. Опашката може да се управлява от разработчика директно, което позволява различни стратегии на преподреждане на задачите с цел оптимизация на процесите. В .NET мониторинга е доста ограничен. Могат да се проверят стойностите, определящи границите на pool-а . чрез GetMinThreads и GetMaxThreads.