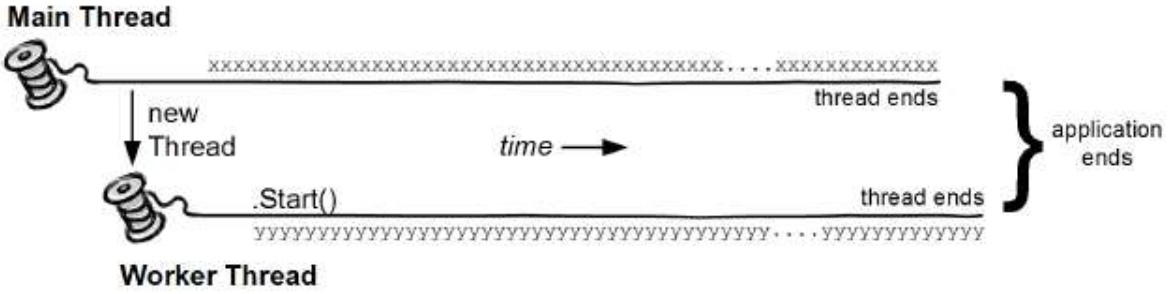
Programske niti

C# podržava paralelno izvršavanje koda kroz koncept niti. Jedna nit je nezavisna izvršna putanja koja se može izvršavati istovremeno sa drugim nitima.

```
using System;
using System.Threading;

class ThreadTest
{
        static void Main()
        {
                Thread t = new Thread (SomeMethod);
                t.Start();
        }
        static void SomeMethod ()
        {
                ...
        }
}
```

C# program se startuje u jednoj niti (*main* nit) koja se kreira automatski od strane CLR i operativnog sistema. CLR pridružuje svakoj niti sopstveni stek za skladištenje lokalnih promenljivih. Niti dele podatke ako imaju zajedničke reference na iste objekte.



Rukovanje nitima prepušteno je raspoređivaču niti (eng. *thread scheduler*). Na jednoprocesorskim kompjuterima, raspoređivač dodeljuje vremenske odsečke svakoj od aktivnih niti (eng. *timeslicing*) i time omogućava njihovo konkurentno izvršavanje (eng. *multithreading*). Na višeprocesorskim kompjuterima multithreading je implementiran pomoću vremenskih odsečaka i originalne konkuretnosti, gde se niti istovremeno izvršavaju na različitim procesorima.

Sve niti u jednoj aplikaciji su logički sadržane u jednom procesu, tj. jedinici operativnog sistema u kome se aplikacija izvršava. Niti imaju neke sličnosti sa procesima – tipično procesi dele procesorsko vreme na isti način kao niti unutar procesa. Ključna razlika u tome što su procesi potpuno izolovani jedan od drugog, dok nit deli memoriju (heap) sa drugim nitima iste aplikacije.

1.1. Imenovanje niti

Svakoj niti moguće je dodeliti ime kako bi se njeno izvršavanje lakše moglo pratiti prilikom procesa otklanjanja grešaka (eng. *debugging*).

```
Thread.CurrentThread.Name = "main";          //current thread

Thread worker = new Thread (Go);
worker.Name = "worker";                      //another thread
```

Prosleđivanje parametara

Koritreći *lambda izraz* može se na lak način proslediti parametar metodi koju nit izvršava.

```
static void Main()
{
        Thread t = new Thread ( () => Print ("Hello from t!") );
        t.Start();
}
static void Print (string message)
{
        Console.WriteLine (message);
}
```

Na ovaj način može se pozvati i više metoda ili čak definisati čitava implementacija naredbi koju nit izvršava.

```
new Thread (() =>
{
        Console.WriteLine ("I'm running on another thread!");
        Console.WriteLine ("This is so easy!");
}).Start();
```

Drugi način za prosleđivanje parametara koristi *Thread.Start* metodu, koja prima jedan argument pri pozivu klase *object*.

```
static void Main()
{
        Thread t = new Thread (Print);
        t.Start ("Hello from t!");
}
static void Print (object messageObj)
{
        string message = (string) messageObj; // We need to cast here
        Console.WriteLine (message);
}
```

Prioritet niti

Prioritet niti definiše koliko izvršnog vremena dobija određena nit.

```
enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest }
…
Thread.CurrentThread.Priority = ThreadPriority.Normal;      //current thread
…
Thread worker = new Thread (Go);
worker. Priority = ThreadPriority. AboveNormal                    //another thread
```

Foreground i background niti

C# podržava niti u zadnjem planu (eng. *background*) koje ne održavaju aplikaciju živom i čim poslednja nit iz prvog plana (eng. *foreground*) bude ugašena cela aplikacija se gasi.

```
Thread worker = new Thread ( () => Console.ReadLine() );
worker.IsBackground = true;
worker.Start();
```

Thread Pool

Kada god je potrebno startovati nit par stotina mikrosekundi se troši na administrativne potrebe kao što je osvežavanje steka lokalnih promenljivih. Korišćenjem bazena niti (eng. *thread pool)* smanjuju se penali na performansama deljenjem i recikliranjem niti. Sve niti u bazenu su pozadinske i moguće je ograničiti njihom broj.

U nastavku su kroz primere prikazana tri načina za upravljanje nitima iz bazena.

Task Parallel Library

1. **Bez povratne vrednosti funkcije**

```
static void Main()            // The Task class is in System.Threading.Tasks
{
        Task.Factory.StartNew (Go);
}

static void Go()
{
        Console.WriteLine ("Hello from the thread pool!");
}
```

2. **Sa povratnom vrednosti funkcije**

```
static void Main()
{
        Task<string> task = Task.Factory.StartNew<string>(()=
                ConvertToUpper ("hello"));
        string result = task.Result;
}

static string ConvertToUpper(string tekst)
{
        return tekst.ToUpper();
}
```

ThreadPool.QueueUserWorkItem

```
static void Main()
{
        ThreadPool.QueueUserWorkItem (Go);
        ThreadPool.QueueUserWorkItem (Go, 123);
        Console.ReadLine();
}
static void Go (object data) // data will be null with the first call.
{
        Console.WriteLine ("Hello from the thread pool! " + data);
}
// OUTPUT:
Hello from the thread pool!
Hello from the thread pool! 123
```

Asinhroni delegati

```
static void Main()
{
        Func<string, int> method = Work;

        IAsyncResult cookie = method.BeginInvoke("test", null, null);
        ...
        int result = method.EndInvoke(cookie);
        Console.WriteLine("String length is: " + result);
}
static int Work(string s) { return s.Length; }
```

Sinhronizacija

Sinhronizacija predstavlja kontrolu konkurentnog izvršavanja dve ili više niti koje dele kritični resurs.

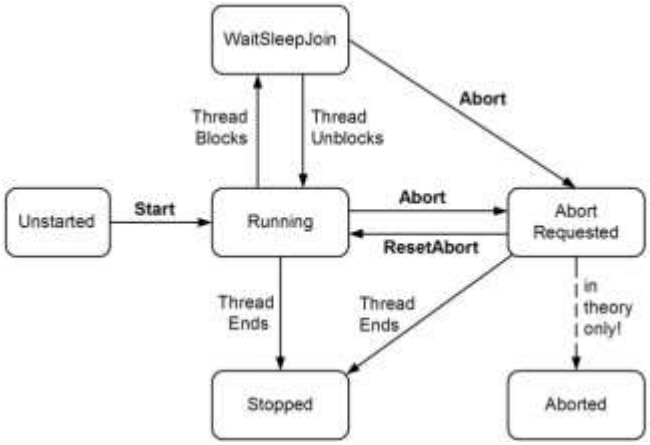
Osnovne tehnike sinhronizacije predstavljaju:

1. Blokiranje
2. Zaključavanje
3. Signalizacija

Deadlock predstavlja najčešću grešku sinhronizacije kada dve niti istovremeno čekaju objekte koji je druga nit zaključala.

Blokiranje

Nit se smatra blokiranom ukoliko je njeno izvršenje zaustavljeno iz nekog razloga. Blokirana nit prepušta dodeljen vremenki odsečak i ne koristi processor dokle god je zadovoljen uslov blokiranja.



Status izvršenja niti moguće je dobiti preko svojstva *ThreadState*.

```
bool threadState = thread.ThreadState;
```

Metoda *Thread.Sleep* omogućava privremeno zaustavljanje niti.

```
Thread.Sleep(500); // sleep for 500 milliseconds
```

Jedna nit može čekati drugu da završi sa radom pozivom metode **Join.**

```
Thread t = new Thread (Go);
t.Start();
t.Join();
```

Zaključavanje

Zaključavanje se koristi da se ograniči broj niti koji mogu istovremeno izvršavati neki deo koda. Najčešći tip zaključavanja predstavlja ekskluzivno zaključavanje, koje osigurava da istovremeno samo jedna može pristupiti deljenom resursu.

Lock

Lock obezbeđuje da istovremeno samo jedna nit može pristupiti resursu. Nudi male troškove sinhronizacije, ali je ograničena na niti istog procesa.

Za sinhronizacioni objekat uvek se koristi polje tipa reference sa privatnim pravom pristupa koji je vidljiv od strane svih niti. Postoji mogućnost ugnježdavanja zaključavanja istog objekta.

```
static readonly object _locker = new object(); //any reference object visible to all threads
```

```
lock (_locker)
{
...
}
```

```
Monitor.Enter (_locker);
try
{
   …
}
Finally
{
   Monitor.Exit (_locker);
}
```

Data dva primera koda su ekvivalenta, lock predstavlja samo kraću formu zapisa.

Mutex

Mutex je dosta sporija verzija lock-a, ali omogućava sinhronizaciju između niti različitih procesa.

```
class OneAtATimePlease
{
        static void Main()
        {
                using (var mutex = new Mutex (false, "MyProject.MyMutex"))
                {
                        // Wait a few seconds if contended, in case another instance
                        // of the program is still in the process of shutting down.
                        if (!mutex.WaitOne (TimeSpan.FromSeconds (3), false))
                        {
                                Console.WriteLine ("Another instance of the app is
                                running. Bye!");
                                return;
                        }
                        RunProgram();
                }
        }
        static void RunProgram()
        {
                Console.WriteLine ("Running. Press Enter to exit");
                Console.ReadLine();
        }
}
```

Semafor

Semafor obezbeđuje da istovremeno samo određen broj niti može pristupiti štićenom resursu.

Semafor je kao noćni klub. Ima svoj kapacitet i kada je pun, ljudi ne mogu više ulaziti nego moraju čekati red. Koliko ljudi napusti klub, toliko ljudi sa početka reda može u njega ući. Konstruktor zahteva dva argumenta: koliko je trenutno slobodnih mesta u klubu i ukupan kapacitet mesta u klubu.

SemaphoreSlim je znatno brži od *Semaphore*-a i koristi se u slučajevima sinhronizacije unutar jednog procesa. *Semaphore* pored sinhronizacije između niti više procesa nudi i mogućnost imenovanja.

```
class TheClub // No door lists!
{
   static SemaphoreSlim sem = new SemaphoreSlim (3); // Capacity

   static void Main()
   {
      for (int i = 1; i <= 5; i++) new Thread (Enter).Start (i);
   }
   static void Enter (object id)
   {
      Console.WriteLine (id + " wants to enter");
      sem.Wait();
      Console.WriteLine (id + " is in!"); // Only three threads
      Thread.Sleep (1000 * (int) id); // can be here at
      Console.WriteLine (id + " is leaving"); // a time.
      sem.Release();
   }
}
```

```
OUTPUT:
1 wants to enter
1 is in!
2 wants to enter
2 is in!
3 wants to enter
3 is in!
4 wants to enter
5 wants to enter
1 is leaving
4 is in!
2 is leaving
5 is in!
```

Reader/Writer Lock

ReaderWriterLock predstavlja nešto složeniju vrstu zaključavanja koje omogućava dve kontrole:

1) Konkuretno čitanje - omogućen pristup delu koda svim nitima radi čitanja podataka
2) Ekskluzivno pisanje - samo jedna nit može pristupiti delu koda radi izmene podataka

```
static ReaderWriterLockSlim _rw = new ReaderWriterLockSlim();
static List<int> _items = new List<int>();
...
static void Read()
{
                _rw.EnterReadLock();
                foreach (int i in _items)
                        Console.WriteLine(i);
                _rw.ExitReadLock();
}

static void Write (object threadID)
{
                _rw.EnterWriteLock();
                _items.Add (newNumber);
                _rw.ExitWriteLock();
}
```

Konstrukcija	Namena	Različiti procesi?	Trošak [ns]
Lock	Obezbeđuje da istovremeno samo jedna nit može pristupiti resursu	NE	20
Mutex		DA	1000
SemaphoreSlim	Obezbeđuje da istovremeno samo određen broj niti može pristupiti resursu	NE	200
Semaphore		DA	1000
ReaderWriterLockSlim	Dozvoljava da se više niti koje čitaju mogu istvoremeno izvršavati sa jednom niti koja piše	NE	40
ReaderWriterLock		NE	100

Signaliziranje

Signaliziranje omogućava da se zaustavi izvršavanje niti dok ne dobije notifikaciju od neke druge niti.

Konstrukcija	Osobine	Između procesa?	Trošak [ns]
AutoResetEvent	Propušta jednu blokiranu nit kada stigne odgovarajuća signalizacija	DA	1000
ManualResetEvent	Propušta sve niti kroz kapiju u vremenskom intervalu između otvaranja i zatvaranja kapije	DA	1000
ManualResetEventSlim		NE	40
CountdownEvent	Propušta blokiranu nit kada dobije odgovarajući broj signalizacija	NE	40

AutoResetEvent

AutoResetEvent funckioniše kao kapija za ulaz na stadion. Navijač dolazi na stadion i zatiče zaključanu kapiju (metoda *WaitOne*). Ubacivanjem karte u automat (metoda *Set*), kapija otključava prolaz za jednu osobu. Atribut „auto" iz imena klase ukazuje na činjenicu da se kapija zatvara automatski kada prva osoba (nit) prođe kroz nju.

```
class BasicWaitHandle
{
        static EventWaitHandle _waitHandle = new AutoResetEvent (false);

        static void Main()
        {
                new Thread (Waiter).Start();
                Thread.Sleep (1000); // Pause for a second...
                _waitHandle.Set(); // Wake up the Waiter.
        }
        static void Waiter()
        {
                Console.WriteLine ("Waiting...");
                _waitHandle.WaitOne(); // Wait for notification
                Console.WriteLine ("Notified");
        }
}
// Output:
Waiting... (pause) Notified.
```

ManualResetEvent

ManualResetEvent funkcioniše kao klasična kapija. Kapija se otvara pozivanjem metode *Set*, dozvoljavajući svim nitima koje čekaju (*WaitOne* metoda) da prođu kapiju. Pozivanjem metode *Reset* se zatvara kapija.

```
class Program
{
    private  ManualResetEvent  manual1 = new ManualResetEvent (false);

    static void Main(string[] args)
    {

        new Thread (Work).Start();

       manual1.WaitOne(100000);
    }

    public static void Func()
    {
         Console.WriteLine("Hello");
         manual1.Set();

    }
}
```

AutoResetEvent i ManualResetEvent objekte moguće je kreirati pomoću EventWaitHandle konstruktora:

```
var manual = new EventWaitHandle (false, EventResetMode.ManualReset);
var auto = new EventWaitHandle (false, EventResetMode.AutoReset);
```

CountDownEvent

CountdownEvent omogućava da nit pre nego što nastavi sa izvršavanjem sačeka prethodno definisani broj signalizacija.

```
static CountdownEvent _countdown = new CountdownEvent (3);

static void Main()
{
        new Thread (SaySomething).Start ("I am thread 1");
        new Thread (SaySomething).Start ("I am thread 2");
        new Thread (SaySomething).Start ("I am thread 3");
        _countdown.Wait(); // Blocks until Signal has been called 3 times
        Console.WriteLine ("All threads have finished speaking!");
}

static void SaySomething (object thing)
{
        Thread.Sleep (1000);
        Console.WriteLine (thing);
        _countdown.Signal();
}
```

1.1.1. Signalizacija između procesa

Sinhronizaciju između različitih procesa moguće je realizovati preko imenovanih objekata klase EventWaitHandle. Ukoliko string nije registrovan, operativni sistem kreira novi EventWaitHandle, u suprotnom dodeljuje referencu na postojeći.

```
EventWaitHandle wh = new EventWaitHandle (false, EventResetMode.AutoReset,
        "MyProject.MyHandle");
```

WaitAny, WaitAll, SignalAndWait

1. *WaitAny* – blokira nit dok ne dobije jednu od navedenih signalizacija

   ```
   WaitHandle.WaitAny(handles);
   ```

2. *WaitAll* – blokira nit dok ne dobije sve od navedenih signalizacija

   ```
   WaitHandle.WaitAll(handles);
   ```

3. *SignalAndWait* – signalizira i blokira nit dok ne dobije signalizaciju.

   ```
   WaitHandle.SignalAndWait(handle1, handle2);
   ```

Za sinhronizaciju se može koristiti bilo koji tip izveden iz klase WaitHandle (AutoResetEvent, ManualResetEvent, CountdownEvent, Semaphore, Mutex, …)

```
private static void Main(string[] args)
{
    WaitHandle[] waitHandles = new WaitHandle[]{
        new AutoResetEvent(false), new AutoResetEvent(false) };

    new Thread(()=> { ExecuteSomeCode("hello",
        (AutoResetEvent)waitHandles[0]);}).Start();
    new Thread(()=> { ExecuteSomeCode("world",
        (AutoResetEvent)waitHandles[1]);}).Start();

    WaitHandle.WaitAll(waitHandles);         //waits for all the threads
}

private static void ExecuteSomeCode(string parameter, AutoResetEvent waitHandle)
{
    Thread.Sleep(2000);           //pretending the execution takes 2 seconds

    Console.WriteLine("Thread: {0} says {1}", Thread.CurrentThread.ManagedThreadId,
        parameter);

    waitHandle.Set();               //signaling that the thread finished.
}
```

Tajmeri

Tajmeri se koriste u slučajevima kada je blok naredbi potrebno više puta pozvati u određenim intervalima. Izuzetno su efikasni u izvršavanju svojih zadataka.

System.Threading.Timer

System.Threading.Timer je najednostavniji tajmer za multithreading okruženje.

```
using System.Threading;
class Program
{
        static void Main()
        {
                // Delay = 5000ms; period = 1000ms
                Timer tmr = new Timer (Tick, "tick...", 5000, 1000);
                Console.ReadLine();
                tmr.Dispose(); // This both stops the timer and cleans up.
        }

        static void Tick (object data)
        {
                // This runs on a pooled thread
                Console.WriteLine (data); // Writes "tick..."
        }
}
```

System.Timers

Za razliku od Threading.Timer-a Timers.Timer nudi proširen nivo mogućnosti i *thread -safe* je.

```
using System;
using System.Timers; // Timers namespace rather than Threading
class SystemTimer
{
        static void Main()
        {
                Timer tmr = new Timer(); // Doesn't require any args
                tmr.Interval = 500;
                tmr.Elapsed += tmr_Elapsed; // Uses an event instead of a delegate
                tmr.Start();           // Start the timer
                Console.ReadLine();
                tmr.Stop();            // Stop the timer
                Console.ReadLine();
                tmr.Start();           // Restart the timer
                Console.ReadLine();
                tmr.Dispose(); // Permanently stop the timer
        }

        static void tmr_Elapsed (object sender, EventArgs e)
        {
                Console.WriteLine ("Tick");
        }
```

1.1. Prekid izvršenja niti

Povremeno može biti korisno da se prevremeno pokrene prekid izvršenja niti, na primer pri završetku aplikacije.

Metoda Abort

Za prekid izvršenja niti moguće je koristiti metodu *Thread.Abort*. Mana ove metode je što je teško predvidiva i ostavlja program u nekonzistentom stanju.

```
public static void Main()
{
        Thread newThread  = new Thread(new ThreadStart(TestMethod));
        newThread.Start();
        Thread.Sleep(1000);

        // Abort newThread.
        Console.WriteLine("Main aborting new thread.");
        newThread.Abort("Information from Main.");

        // Wait for the thread to terminate.
        newThread.Join();
        Console.WriteLine("New thread terminated - Main exiting.");
 }
```

Cancelation Token

Ovaj patern omogućava prekid izvršenja niti na bezbedan način. Nit periodično proverava vrednost zastavice prekida *CancellationToken* i na taj način doznaje informaciju da li je potrebno da prekine svoje izvršenje.

```
static void Main()
{
        var canceler = new CancellationTokenSource();

        new Thread(() =>
        {
                try
                {
                        Work(canceler.Token);
                }
                catch (OperationCanceledException)
                {
                        Console.WriteLine("Canceled!");
                }
        }).Start();
        Thread.Sleep(1000);
        canceler.Cancel(); // Safely cancel worker.
}

static void Work(CancellationToken token)
{
        while (true)
        {
                token.ThrowIfCancellationRequested();
                //...code lines...
        }
}
```

Zadatak

Implementirati sistem za kontrolu plaćanja parkinga koji sadrži 20 mesta za parkiranje i dva automata za naplatu parkinga. Parking pored automobila mogu koristiti i autobusi ukoliko postoje dva slobodna susedna mesta za parking. Za cenu parkinga koristiti 20din po sekundi. Napisani program testirati sa 30 automobila i 5 autobusa različitih tablica. Maksimalno vreme čekanja slobodnog mesta iznosi jedna sekunda, a zadržavanje 10-20s. Za simulaciju dolaska vozila koristiti periode od 1 i 1,5 sekunde.