Osa IV

Vaativampi C#

15

Monisäikeinen ohjelmointi

Teknisesti ajatellen säikeet eivät ole mikään C#-kielen oma erikoisominaisuus ja siksi useimmat C#-kirjat eivät käsittelekään sitä lainkaan. Vaikka olen yrittänyt pitäytyä tiukasta C#:ssa, monisäikeinen ohjelmointi on kuitenkin alue, joka useimpien ohjelmoijien pitää tuntea, kun opiskelevat tätä uutta kieltä. En tietenkään pysty käsittelemään yhdessä luvussa kaikkia monisäikeisten ohjelmien tekoon liittyviä asioita, mutta käyn läpi perusteet ja muutaman säikeen keskeyttämiseen, ajoitukseen ja elinajan hallintaan liittyvän erikoispiirteen. Puhun myös säikeen synkronoinnista *System.Monitor* ja *System.Mutex* -luokilla ja C#:n *lock*-käskystä.

Säikeistyksen perusteet

Yksisäikeisen sovelluksen käyttäminen on kuin kävisi ostoksilla valintamyymälässä, jossa on yksi kassa. Vain yhden kassan palkkaaminen on kauppiaan kannalta edullista ja kassa selviää vähäisestä asiakasvirrasta, mutta kun myymälä ilmoittaa erikoistarjouksista ja jonotusajat kassalla kasvavat, asiakkaat alkavat tuskastua. Meillä on tässä tyypillinen pullonkaulatilanne: liian paljon tietoja liian pienessä kanavassa. Ratkaisu on tietenkin palkata lisää kassoja.

Ja samanlainen suhde on säikeillä ja sovelluksilla. Säikeistys antaa sovellukselle mahdollisuuden jakaa tehtävät siten, että ne suoritetaan toisistaan erillään prosessorin ja käyttäjän ajan mahdollisimman tehokkaaksi hyödyntämiseksi. Jos et ole kuitenkaan ennen käyttänyt säikeistystä, niin muista: säikeistys ei ole oikea juttu jokaiseen ohjelmaan ja voi joskus jopa hidastaa sovellusta. Joten on tärkeää, että yhdessä säikeistyksen oppimisen kanssa opit myös ymmärtämään, milloin sitä tulee käyttää. Sitä silmälläpitäen olen ottanut luvun loppuun kappaleen (”Säikeistysohjeet”), joka auttaa sinua määrittelemään, koska luot

useita säikeitä sovellukseesi. Aloitetaan nyt tutkimalla, miten säikeistys on toteutettu Microsoft .NET Frameworkissa.

Säikeet ja moniajo

Säie on käsittely-yksikkö ja moniajo on usean säikeen yhtäaikainen suorittaminen. Moniajoon liittyy kaksi asiaa: yhteistoiminnallinen ja ohjaava. Ihan ensimmäiset Microsoft Windowsin versiot tukivat yhteistoiminnallista moniajoa, joka tarkoitti sitä, että kukin säie oli vastuussa kontrollin luovuttamisesta prosessorille, jotta prosessori pystyi suorittamaan muita säikeitä. Niillä meistä, jolla on taustaa muista käyttöjärjestelmistä (minun tapauksessani IBM Systen/38 (CPF) ja OS/2:sta), on oma tarina kerrottavana päivästä, jolloin jumiutimme tietokoneen, koska emme muistaneet sijoittaa *PeekMessage*-kutsua koodiimme, jotta CPU olisi voinut palvella muita järjestelmän säikeitä. ”Minä vai?” oli tyypillinen kysymyksemme siinä tilanteessa.

Microsoft Windows NT ja myöhemmin Windows 95, Windows 98 ja Windows 2000 tukevat kuitenkin samaa ohjaavaa moniajoa kuin OS/2. Ohjaavassa moniajossa prosessori on vastuussa siitä, että se antaa kullekin säikeelle tehtävän suorittamiseen määrätyn määrän aikaa eli aikajakson. Prosessori vaihtaa sitten suorituksessa olevia säikeitä antaen kullekin sen aikajakson eikä ohjelman itse tarvitse huolehtia miten ja koska luovuttaa kontrollin, jotta muutkin säikeet etenevät. Koska .NET toimii vain ohjaavissa moniajojärjestelmissä, minäkin keskityn siihen.

Ohjaavassa moniajossakaan, jos käytät konetta, jossa on vain yksi prosessori, sinulla ei oikeastaan ole useita säikeitä samaan aikaan käynnissä. Koska prosessori vaihtaa suoritettavaa säiettä muutamissa millisekunneissa, tuntuu kuin ne olisivat samaan aikaan käynnissä. Jos haluat oikeasti ajaa useita säikeitä samanaikaisesti, sinun pitää koodata ja ajaa koodisi koneessa, jossa on useita prosessoreja.

Kontekstin vaihto

Kontekstin vaihto (context switching) on oleellinen osa säikeistystä ja koska se on hieman hankala menetelmä, kerron siitä tässä johdanto-kappaleessa lyhyesti.

Prosessori käyttää laitteiston ajastinta määritelläkseen, milloin määrätyn säikeen aikajakso on päättynyt. Kun ajastin ilmoittaa keskeytyksestä, prosessori tallentaa kaikki säikeen rekisterit pinoon. Sitten prosessori siirtää samat rekisterit pinosta CONTEXT-

nimiseen tietorakenteeseen. Kun prosessori haluaa palata takaisin säikeeseen, se tekee saman takaperin eli palauttaa rekisterit säikeeseen liitetystä CONTEXT-rakenteesta. Tämä koko toiminto on nimeltään kontekstin vaihto.

Monisäikeinen sovellus

Ennen kuin tutkimme erilaisia säikeiden käyttötapoja C#:ssa, katsotaan, miten helppoa toisen säikeen luominen on. Kun olemme käyneet esimerkin yksityiskohtaisesti läpi, katsomme tarkemmin *System.Threading*-nimiavaruutta ja erityisesti *Thread*-luokkaa. Tässä esimerkissä luon toisen säikeen *Main*-metodista. Toiseen säikeeseen liitetty metodi tulostaa merkkijonon todistaen siten, että säiettä on kutsuttu.

```
using System;
using System.Threading;

class SimpleThreadApp
{
    public static void WorkerThreadMethod()
    {
        Console.WriteLine(“Worker thread started”);
    }

    public static void Main()
    {
        ThreadStart worker = new ThreadStart(WorkerThreadMethod);

        Console.WriteLine(“Main - Creating worker thread”);

        Thread t = new Thread(worker);
        t.Start();

    Console.WriteLine
         (“Main - Have requested the start of worker thread”);
    }
}
```

Jos käännät ja suoritat tämän sovelluksen, näet *Main*-metodin viestin tulostuvan ennen toisen säikeen viestiä, joka todistaa, että toinen säie toimii asynkronisesti. Analysoidaan mitä ohjelmassa tapahtuu.

Ensimmäinen uusi osa on *System.Threading*-nimiavaruuden ottaminen sovellukseen mukaan *using*-avainsanalla. Tutustumme nimiavaruuteen kohta. Nyt riittää, kun tiedät, että

se sisältää luokat, jotka ovat tarpeen säikeiden käsittelyyn .NET-ympäristössä. Katsotaan nyt *Main*-metodin ensimmäistä riviä:

```
ThreadStart WorkerThreadMethod = new ThreadStart(WorkerThreadMethod);
```

Aina, kun näet vastaavan muotoisen rivin, voit olla varma, että *x* on delegaatti (kuten opit luvussa 14, ”Delegaatit ja tapahtumakäsittelijät”) tai metodin määrittely:

x varName = new *x*(*methodName*);

Tiedämme siis, että *ThreadStart* on delegaatti. Mutta se ei ole mikä tahansa delegaatti. Se on nimenomaan delegaatti, jota pitää käyttää, kun luodaan uusi säie. Sitä käytetään määrittelemään metodi, jota haluat kutsuttavan säikeesi metodina. Sieltä instantioin *Thread*-objektin, jonka muodostin ottaa ainoana parametrinaan *ThreadStart*-delegaatin, näin:

```
Thread t = new Thread(worker);
```

Sen jälkeen kutsun *Thread*-objektini *Start*-metodia, joka aiheuttaa kutsun *WorkerThreadMethod*-metodiin.

Se on siinä! Kolme koodiriviä säikeen alustamiseen ja käynnistämiseen ja itse säiemetodiin ja se toimii. Katsotaan nyt tarkemmin *System.Threading*-nimiavaruutta ja sen luokkia, jotka saavat tämän kaiken tapahtumaan.

Työskentely säikeillä

Säikeiden luonti ja hallinta toteutetaan *System.Threading.Thread*-luokan kautta, joten aloitetaan siitä.

AppDomain

.NETissä säikeet suoritetaan *AppDomain*-nimisessä ympäristössä. Olet ehkä kuullut, että *AppDomain* on vastaava kuin Win32-prosessi siinä mielessä, että se tarjoaa samoja ominaisuuksia, kuten vikatoleranssi ja mahdollisuus riippumattomaan aloitukseen ja lopetukseen. Tämä on hyvä vertaus, mutta suhteessa säikeisiin se ei enää päde. Win32:ssa säie on rajoitettu yhteen prosessiin, kuten muistat, kun kerroin aiemmin kontekstin vaihdosta. Yhdessä prosessissa oleva säie ei voi käsitellä toisen prosessin säikeelle kuuluvaa metodia. .NETissä säikeet voivat kuitenkin ylittää *AppDomain*-rajat ja yhden säikeen metodi voi kutsua toisen *AppDomainiin* kuuluvaa metodia. Siksi tässä tulee parempi määritelmä *AppDomainille*: fyysisen prosessin sisällä oleva looginen prosessi.

Thread-luokka

Lähes kaiken, mitä säikeillä teet, teet *Thread*-luokan avulla. Tässä kappaleessa katsomme *Thread*-luokan käyttöä säikeiden käytön perustehtäviin.

Säikeiden ja Thread-objektien luominen

Voit instantioida *Thread*-objektin kahdella eri tavalla. Olet jo nähnyt toisen: luodaan uusi säie ja samassa prosessissa otetaan *Thread*-objekti, jolla käsitellään uutta säiettä. Toinen tapa saada *Thread*-objekti on kutsua staattista *Thread.CurrentThread*-metodia suoritettavalle säikeelle.

Säikeen elinajan hallinta

On monia erilaisia tehtäviä, joissa sinun pitää ohjata säikeen toimintaa tai elinkaarta. Voit hallita niitä kaikkia käyttämällä *Thread*-luokan eri metodeja. On esimerkiksi melko yleistä, että säie pitää pysäyttää määrätyksi ajaksi. Teet sen kutsumalla *Thread.Sleep*-metodia. Tämä metodi ottaa yhden parametrin, joka kuvaa sitä aikaa millisekunteina, jonka haluat säikeen olevan pysähdyksissä. Huomaa, että *Thread.Sleep* on staattinen metodi eikä sitä voi kutsua *Thread*-objektin instanssilla. Tähän on olemasssa hyvä syy. Et voi kutsua *Thread.Sleep*-metodia mistään muusta kuin par'aikaa suorituksessa olevasta säikeestä. Staattinen *Thread.Sleep*-metodi kutsuu staattista *CurrentThread*-metodia, joka sen jälkeen pysäyttää säikeen halutuksi ajaksi. Tässä esimerkki:

```
using System;
using System.Threading;

class ThreadSleepApp
{
    public static void WorkerThreadMethod()
    {
        Console.WriteLine("Worker thread started");

        int sleepTime = 5000;

        Console.WriteLine("\tsleeping for {0} seconds", sleepTime / 1000);
        Thread.Sleep(sleepTime); // Sleep for five seconds.
        Console.WriteLine("\twaking up");
    }

    public static void Main()
    {
         ThreadStart worker = new ThreadStart(WorkerThreadMethod);
```

(jatkuu)

```
        Console.WriteLine("Main - Creating worker thread");

        Thread t = new Thread(worker);
        t.Start();

        Console.WriteLine
            ("Main - Have requested the start of worker thread");
    }
}
```

On kaksi muutakin tapaa kutsua *Thread.Sleep*-metodia. Ensinnäkin, kutsumalla *Thread.Sleep*-metodia parametrin arvolla 0, saat aikaan sen, että nykyinen säie luopuu aikajaksonsa lopusta. Välittämällä parametrina arvon *Timeout.Infinite* säie pysähtyy ikuisiksi ajoiksi, kunnes toinen säie keskeyttää pysähdyksen kutsumalla pysäytetyn säikeen *Thread.Interrupt*-metodia.

Toinen tapa pysäyttää säikeen suoritus on käyttää *Thread.Suspend*-metodia. Näiden kahden tavan välillä on merkittäviä eroja. Ensinnäkin *Thread.Suspend*-metodia voidaan kutsua suoritettavasta säikeestä tai toisesta säikeestä. Toiseksi, kun säie on keskeytetty tällä tavalla, vain toinen säie voi jatkaa sen suoritusta uudelleen kutsumalla *Thread.resume*-metodia. Huomaa, että kun säikeen on keskeyttänyt toinen säie, ensimmäinen säie ei lukkiudu vaan kutsu palautuu välittömästi. Myöskään riippumatta siitä, miten monta kertaa *Thread.Suspend*-metodia on kutsuttu, yksi *Thread.Resume*-metodin kutsu aiheuttaa säikeen toiminnan jatkumisen.

Säikeiden tuhoaminen

Jos tulee tarve säikeen tuhoamiseen, voit tehdä sen kutsumalla *Thread.Abort*-metodia. Ajonaikainen ympäristö pakottaa säikeen keskeytyksen aiheuttamalla *ThreadAbortException*-poikkeuksen. Vaikka motodi yrittäisi ottaa kiinni tämän poikkeuksen, ei ajonaikainen ympäristö salli sitä. Ajonaikainen ympäristö suorittaa kuitenkin keskeytetyn säikeen *finally*-lohkon koodin, jos sellainen on olemassa. Seuraava koodi näyttää, mitä tarkoitan. *Main*-metodi pysähtyy kahdeksi sekunniksi varmistaakseen, että ajonaikaisella ympäristöllä on aikaa käynnistää työsäie. Käynnistymisensä jälkeen työsäie laskee kymmeneen, pitäen sekunnin tauon joka välissä. Kun *Main*-metodi palaa toimintaan kahden sekunnin tauon jälkeen, se keskeyttää työsäikeen. Huomaa, että *finally*-lohkon koodi suoritetaan keskeytyksen jälkeen.

```
using System;
using System.Threading;

class ThreadAbortApp
{
    public static void WorkerThreadMethod()
    {
```

```
        try
        {
            Console.WriteLine("Worker thread started");

            Console.WriteLine
                ("Worker thread - counting slowly to 10");
            for (int i = 0; i < 10; i++)
            {
                Thread.Sleep(500);
                Console.Write("{0}...", i);
            }

            Console.WriteLine("Worker thread finished");
        }
        catch(ThreadAbortException e)
        {
        }
        finally
        {
            Console.WriteLine
                ("Worker thread -
                 I can't catch the exception, but I can cleanup");
        }
    }

    public static void Main()
    {
        ThreadStart worker = new ThreadStart(WorkerThreadMethod);

        Console.WriteLine("Main - Creating worker thread");

        Thread t = new Thread(worker);
        t.Start();

        // Give the worker thread time to start.
        Console.WriteLine("Main - Sleeping for 2 seconds");
        Thread.Sleep(2000);

        Console.WriteLine("\nMain - Aborting worker thread");
        t.Abort();
    }
}
```

Kun käännät ja suoritat tämän sovelluksen, saat seuraavat tulokset:

```
Main - Creating worker thread
Main - Sleeping for 2 seconds
Worker thread started
Worker thread - counting slowly to 10
```

```
0...1...2...3...
Main - Aborting worker thread
Worker thread - I can’t catch the exception, but I can cleanup
```

Sinun tulee ymmärtää, että kun *Thread.Abort*-metodia kutsutaan, säie ei keskeytä suoritustaan välittömästi. Ajonaikainen ympäristö odottaa, kunnes säie on saavuttanut dokumentaation kuvaaman *turvapisteen* (safe point). Siksi, jos koodisi on riipuvainen siitä, että keskeytyksen jälkeen tapahtuu jotain ja sinun tulee olla varma, että säie on keskeytynyt, voit käyttää *Thread.Join*-metodia. Sen asynkroninen metodikutsu, eli ohjelman suoritus ei palaa siitä ennen kuin säie on keskeytetty. Huomaa myös, että kun olet keskeyttänyt säikeen, sitä ei voi käynnistää uudelleen. Silloin, vaikka sinulla on kelvollinen *Thread*-objekti, et voi tehdä sillä koodin suorittamisen kannalta mitään hyödyllistä.

Säikeiden ajoitus

Sillon, kun prosessori vaihtaa toiseen säikeeseen, kun yhden aikajakso on päättynyt, on seuraavan suoritettavan säikeen valinta kaikkea muuta kuin sattumanvaraista. Kuhunkin säikeeseen on liitetty prioriteettitaso, joka kertoo prosessorille, miten sitä tulee ajoittaa suhteessa muihin järjestelmän säikeisiin. Prioriteettitaso on oletuksena *Normal* (tästä lisää kohta) sellaisilla säikeillä, jotka on luotu ohjelman suorituksen aikana. Säikeet, jotka on luotu ajonaikaisen ympäristön ulkopuolella, säilyttävät alkuperäisen prioriteettitasonsa. Haet ja asetat prioriteettitason *Thread.Priority*-ominaisuudella. *Thread.Priority*-ominaisuuden setter-metodin parametri on tyyppiä *Thread.ThreadPriority*, joka on *enum*-tyyppiä ja jokin seuraavista arvoista: *Highest*, *AboveNormal*, *Normal*, *BelowNormal* tai *Lowest*.

Esimerkkinä siitä, miten prioriteettitaso voi vaikuttaa yksinkertaisessakin koodissa, katsotaan seuraavaa sovellusta, jossa työsäie laskee 1..10 ja toinen säie 11..20. Huomaa sisäkkäinen silmukka kussakin *WorkerThread*-metodissa. Kukin silmukka kuvaa tehtävää, jonka säie oikeassa sovelluksessa joutuisi tekemään. Koska nämä metodit eivät oikeasti tee mitään, silmukoiden poisjättäminen aiheuttaisi säikeen toiminnan päättymisen jo ennen ensimmäisen aikajakson päättymistä!

```
using System;
using System.Threading;
class ThreadSchedule1App
{
    public static void WorkerThreadMethod1()
    {
        Console.WriteLine(“Worker thread started”);

        Console.WriteLine
```

```
        ("Worker thread - counting slowly from 1 to 10");
    for (int i = 1; i < 11; i++)
    {
        for (int j = 0; j < 100; j++)
        {
            Console.Write(".");
            // Code to imitate work being done.
            int a;
            a = 15;
        }
        Console.Write("{0}", i);
    }

    Console.WriteLine("Worker thread finished");
}

public static void WorkerThreadMethod2()
{
    Console.WriteLine("Worker thread started");

    Console.WriteLine
        ("Worker thread - counting slowly from 11 to 20");
    for (int i = 11; i < 20; i++)
    {
        for (int j = 0; j < 100; j++)
        {
            Console.Write(".");
            // Code to imitate work being done.
            int a;
            a = 15;
        }
        Console.Write("{0}", i);
    }

    Console.WriteLine("Worker thread finished");
}

public static void Main()
{
    ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
    ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

    Console.WriteLine("Main - Creating worker threads");

    Thread t1 = new Thread(worker1);
```

(jatkuu)

```
        Thread t2 = new Thread(worker2);

        t1.Start();
        t2.Start();
    }
}
```

Sovelluksen suorittaminen saa aikaan seuraavat tulokset. Olen hieman lyhentänyt sitä jättämällä suurimman osan pisteistä pois.

```
Main - Creating worker threads
Worker thread started
Worker thread started
Worker thread - counting slowly from 1 to 10
Worker thread - counting slowly from 11 to 20
......1......11......2......12......3......13
```

Kuten näet, säikeet saavat prosessorilta yhtä paljon suoritusaikaa. Muutetaan nyt kummankin säikeen *Priority*-ominaisuutta seuraavan koodin esittämällä tavalla. Annan ensimmäiselle säikeelle korkeimman prioritetin ja toiselle matalimman ja näet tuloksessa huomattavan eron.

```
using System;
using System.Threading;

class ThreadSchedule2App
{
    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Worker thread started");

        Console.WriteLine
            ("Worker thread - counting slowly from 1 to 10");
        for (int i = 1; i < 11; i++)
        {
            for (int j = 0; j < 100; j++)
            {
                Console.Write(".");
                // Code to imitate work being done.
                int a;
                a = 15;
            }
            Console.Write("{0}", i);
        }

        Console.WriteLine("Worker thread finished");
    }

    public static void WorkerThreadMethod2()
```

```
    {
        Console.WriteLine(“Worker thread started”);

        Console.WriteLine
            (“Worker thread - counting slowly from 11 to 20”);
        for (int i = 11; i < 20; i++)
        {
            for (int j = 0; j < 100; j++)
            {
                Console.Write(“.”);
                // Code to imitate work being done.
                int a;
                a = 15;
            }
            Console.Write(“{0}", i);
        }

        Console.WriteLine(“Worker thread finished”);
    }

    public static void Main()
    {
        ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
        ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

        Console.WriteLine(“Main - Creating worker threads”);

        Thread t1 = new Thread(worker1);
        Thread t2 = new Thread(worker2);

        t1.Priority = ThreadPriority.Highest;
        t2.Priority = ThreadPriority.Lowest;

        t1.Start();
        t2.Start();
    }
}
```

(Odotetut) tulokset näkyvät seuraavalla sivulla. Huomaa, että toisen säikeen prioriteetti asetettiin niin alhaiseksi, että se ei saanut yhtään aikajaksoa ennen kuin ensimmäinen säie oli saanut työnsä loppuun.

```
Main - Creating worker threads
Worker thread started
Worker thread started
Worker thread - counting slowly from 1 to 10
Worker thread - counting slowly from 11 to 20
```

```
......1......2......3......4......5......6......7......8......9......10......
11......12......13......14......15......16......17......18......19......20
```

Muista, että kun kerrot prosessorille prioriteetin, joka säikeelle haluat antaa, käyttöjärjestelmä käyttää arvoa osana ajoitusalgoritmia, jolla se ohjaa prosessoria. .NETissä algoritmi perustuu prioriteettitasoon, jota juuri käytit (Thread.Priority-ominaisuuden avulla), yhdessä prosessin prioriteettiluokan ja dynaamisen taso -arvojen kanssa. Kaikkia näitä käytetään numeerisen arvon (0..31 Intelin prosessoreissa) muodostamiseen, joka kuvaa säikeen prioriteettia. Säie, jonka prioriteettiarvo on korkein, on ensimmäisenä vuorossa, kun aikajaksoja jaetaan.

Vielä yksi varoitus säikeiden ajoitukseen liittyen: käytä sitä varoen. Sanotaan, että sinulla on GUI-sovellus ja muutama työsäie, jotka suorittavat asynkronisia toimintoja. Jos asetat niiden prioriteetin liian korkeaksi, saattaa käyttöliittymä hidastua, koska pääsäie, jossa GUI-sovellusta suoritetaan, saa harvemmin prosessoriaikaa. Jollei sinulla ole erityistä syytä määritellä säiettä korkealle prioriteetille, on paras jättää prioriteetti oletusarvoonsa Normal.

> Huomaa Kun sinulla on tilanne, jossa useita säikeitä suoritetaan samalla prioriteetilla, ne saavat kaikki yhtä paljon prosessoriaikaa. Tällainen tilanne on ”*round robin scheduling*.”

Säieturvallisuus ja synkronointi

Kun tehdään ohjelmia yksisäikeisessä ympäristössä, metodit kirjoitetaan yleensä niin, että useissa kohdissa koodia objekti on tilapäisesti virheellinen. Onhan selvää, että jos vain yksi säie käsittelee objektia kerrallaan, takaat, että kukin metodi asettaa objektin kuntoon ennen kuin toista metodia kutsutaan, eli objekti on aina oikeassa tilassa jokaisen objektia käyttävän ohjelman kannalta. Mutta kun käytetään useita säikeitä, voit helposti joutua tilanteeseen, jossa prosessori vaihtaa suorituksen toiseen säikeeseen, kun objektisi on virheellisessä tilassa. Jos tuo toinen objekti sitten yrittää käyttää samaa objektia, ovat tulokset usein melko arvaamattomia. Siksi käsite ”säieturvallisuus” (thread safety) tarkoittaa, että objektin jäsenet aina säilyttävät oikean tilansa, kun objektia käytetään jatkuvasti useista eri säikeistä.

Miten tämän epävakaan tilan voi estää? Kuten yleensäkin ohjelmoinnissa, tämän tunnetun asian voi ratkaista usealla tavalla. Kuvaan tässä kappaleessa yleisimmän tavan: synkronisoinnin. Synkronoinnissa määrittelet *kriittiset lohkot* (critical section) koodissa, joita

voi käsitellä vain yksi lohko kerrallaan ja siten varmistetaan, että objektisi käyttäjä eivät näe objektiasi tilapäisesti epävakaassa tilassa. Tutkimme erilaisia tapoja määritellä kriittiset lohkot, kuten .NETin *Monitor* ja *Mutex*-luokat sekä C#:n *lock*-käsky.

Koodin suojaaminen *Monitor*-luokan avulla

System.Monitor-luokan avulla voit suojata koodilohkon lukituksilla ja signaaleilla. Ajatellaan, että sinulla on metodi, joka päivittää tietokantaa ja jota ei voi suorittaa kaksi tai useampia säikeitä yhtä aikaa. Jos tämän metodin suorittama päivitys vie erityisen kauan aikaa ja sinulla on useita säikeitä, joista kukin voi kutsua tätä metodia, sinulla on vakava ongelma. Tässä voit hyödyntää *Monitor*-luokkaa. Katso seuraavaa synkronisointiesimerkkiä. Siinä meillä on kaksi säiettä, joista molemmat kutsuvat *Database.SaveData*-metodia.

```
using System;
using System.Threading;

class Database
{
    public void SaveData(string text)
    {
        Console.WriteLine(“Database.SaveData - Started”);

        Console.WriteLine(“Database.SaveData - Working”);
        for (int i = 0; i < 100; i++)
        {
            Console.Write(text);
        }

        Console.WriteLine(“\nDatabase.SaveData - Ended”);
    }
}

class ThreadMonitor1App
{
    public static Database db = new Database();

    public static void WorkerThreadMethod1()
    {
        Console.WriteLine(“Worker thread #1 - Started”);
```

(jatkuu)

```
            Console.WriteLine
                ("Worker thread #1 -Calling Database.SaveData");
            db.SaveData("x");

            Console.WriteLine("Worker thread #1 - Returned from Output");
        }

        public static void WorkerThreadMethod2()
        {
            Console.WriteLine("Worker thread #2 - Started");

            Console.WriteLine
                ("Worker thread #2 - Calling Database.SaveData");
            db.SaveData("o");

            Console.WriteLine("Worker thread #2 - Returned from Output");
        }

        public static void Main()
        {
            ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
            ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

            Console.WriteLine("Main - Creating worker threads");

            Thread t1 = new Thread(worker1);
            Thread t2 = new Thread(worker2);

            t1.Start();
            t2.Start();
        }
    }
```

Kun käännät ja suoritat tämän sovelluksen näet, että tuloste sisältää yhdistelmän *o* ja *x*-kirjaimia osoittaen siten, että *Database.SaveData*-metodia suorittavat samanaikaisesti molemmat säikeet. (Huomioi, että olen taas lyhentänyt tulostetta.)

```
Main - Creating worker threads

Worker thread #1 - Started
Worker thread #2 - Started

Worker thread #1 - Calling Database.SaveData
```

```
Worker thread #2 - Calling Database.SaveData

Database.SaveData - Started
Database.SaveData - Started

Database.SaveData - Working
Database.SaveData - Working
xoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxox
Database.SaveData - Ended
Database.SaveData - Ended

Worker thread #1 - Returned from Output
Worker thread #2 - Returned from Output
```

Jos *Database.SaveData*-metodin pitää saada usean taulun päivityksensä loppuun ennen kuin sitä voidaan kutsua toisesta säikeestä, niin meillä on ongelma.

Otetaan esimerkkiin mukaan *Monitor*-luokka ja käytetään sen kahta staattista metodia. Ensimmäinen on nimeltään *Enter*. Kun metodia kutsutaan, se yrittää hakea objektin monitor-lukon. Jos lukko on jo toisella säikeellä, metodi jää odottamaan, kunnes lukko on vapautettu. Huomaa, että tässä ei toteuteta mitään automaattista muunnosoperaatiota, joten voit välittää tälle metodille vain viittaustyypin. Lukon vapauttamisen suorittaa *Monitor.Exit*-metodi. Tässä esimerkki korjattuna siten, että se varmistaa *Database.SaveData*-metodin kutsumisen vain yhdestä säikeestä kerrallaan.

```
using System;
using System.Threading;

class Database
{
    public void SaveData(string text)
    {
        Monitor.Enter(this);

        Console.WriteLine("Database.SaveData - Started");

        Console.WriteLine("Database.SaveData - Working");
        for (int i = 0; i < 100; i++)
        {
            Console.Write(text);
        }

        Console.WriteLine("\nDatabase.SaveData - Ended");

        Monitor.Exit(this);
    }
}
```

(jatkuu)

```
class ThreadMonitor2App
{
    public static Database db = new Database();

    public static void WorkerThreadMethod1()
    {
        Console.WriteLine(“Worker thread #1 - Started”);

        Console.WriteLine
            (“Worker thread #1 - Calling Database.SaveData”);
        db.SaveData(“x”);

        Console.WriteLine(“Worker thread #1 - Returned from Output”);
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine(“Worker thread #2 - Started”);

        Console.WriteLine
            (“Worker thread #2 - Calling Database.SaveData”);
        db.SaveData(“o”);

        Console.WriteLine(“Worker thread #2 - Returned from Output”);
    }

    public static void Main()
    {
        ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
        ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

        Console.WriteLine(“Main - Creating worker threads”);

        Thread t1 = new Thread(worker1);
        Thread t2 = new Thread(worker2);

        t1.Start();
        t2.Start();
    }
}
```

Huomaa seuraavassa tulosteessa, että vaikka toinen säie kutsui *Database.SaveData*-metodia, *Monitor.Enter*-metodi aiheutti, että toinen säie joutui odottamaan, kunnes ensimmäinen säie oli vapauttanut lukkonsa:

```
Main - Creating worker threads
```

```
Worker thread #1 - Started
Worker thread #2 - Started

Worker thread #1 - Calling Database.SaveData
Worker thread #2 - Calling Database.SaveData

Database.SaveData - Started
Database.SaveData - Working
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Database.SaveData - Ended

Database.SaveData - Started

Worker thread #1 - Returned from Output

Database.SaveData - Working
oooooooooooooooooooooooooooooooooooooooo
Database.SaveData - Ended

Worker thread #2 - Returned from Output
```

Monitorilukkojen käyttäminen C#:n *lock*-käskyllä

Vaikka C#:n *lock*-käsky ei tue kaikki niitä ominaisuuksia, joita *Monitor*-luokasta löytyy, voit sen avulla kuitenkin hakea ja vapauttaa monitorilukon. Käytät sitä yksinkertaisesti kirjoittamalla *lock*-käskyn sen koodilohkon eteen, joka tulee turvata. Sulut osoittavat suojattavan koodin alku- ja loppupisteen, joten mitään *unlock*-käskyä ei tarvita. Seuraava koodi toteuttaa saman synkronoinnin kuin edeltävät esimerkit:

```
using System;
using System.Threading;

class Database
{
    public void SaveData(string text)
    {
        lock(this)
        {
            Console.WriteLine(“Database.SaveData - Started”);

            Console.WriteLine(“Database.SaveData - Working”);
            for (int i = 0; i < 100; i++)
            {
```

(jatkuu)

```
                Console.Write(text);
            }

            Console.WriteLine("\nDatabase.SaveData - Ended");
        }
    }
}

class ThreadLockApp
{
    public static Database db = new Database();

    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Worker thread #1 - Started");

        Console.WriteLine
            ("Worker thread #1 - Calling Database.SaveData");
        db.SaveData("x");

        Console.WriteLine("Worker thread #1 - Returned from Output");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Worker thread #2 - Started");

        Console.WriteLine
            ("Worker thread #2 - Calling Database.SaveData");
        db.SaveData("o");

        Console.WriteLine("Worker thread #2 - Returned from Output");
    }

    public static void Main()
    {
        ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
        ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

        Console.WriteLine("Main - Creating worker threads");

        Thread t1 = new Thread(worker1);
        Thread t2 = new Thread(worker2);

        t1.Start();
        t2.Start();
    }
}
```

Koodin synkronisointi käyttämällä *Mutex*-luokkaa

Mutex-luokka, joka on määritelty *System.Threading*-nimiavaruudessa, on suorituksenaikainen esitys Win32-järjestelmän saman nimisestä alkeellisesta toiminnosta. Voit käyttää *mutex*-luokkaa estämään koodin käyttöä aivan kuten monitorilukolla, mutta mutexit ovat paljon hitaampia, koska ne ovat monipuolisempia. Käsite mutex tulee sanoista mutually exclusive ja aivan kuten yksi säie kerrallaan voi saada määrätyn objektin monitorilukon, vain yksi säie kerrallaan voi hakea mutexin.

Voit luoda mutexin C#:ssa seuraavilla kolmella muodostimella:

Mutex( )
Mutex(bool *initiallyOwned*)
Mutex(bool *initiallyOwned*, string *mutexName*)

Ensimmäinen muodostin luo nimettömän mutexin ja tekee aktiivisesta säikeestä sen omistajan. Siten aktiivinen säie lukitsee mutexin. Toinen muodostin ottaa parametrina Boolean-arvon, joka ilmoittaa, haluaako mutexin luova säie omistaa sen (lukita sen). Ja kolmannen muodostimen avulla voit määritellä mutexin nimen ja sen, omistaako aktiivinen säie sen. Sisällytetään nyt mutex suojaamaan *Database.SaveData*-metodin käsittelyä:

```
using System;
using System.Threading;

class Database
{
    Mutex mutex = new Mutex(false);

    public void SaveData(string text)
    {
        mutex.WaitOne();

        Console.WriteLine("Database.SaveData - Started");

        Console.WriteLine("Database.SaveData - Working");
        for (int i = 0; i < 100; i++)
        {
            Console.Write(text);
        }

        Console.WriteLine("\nDatabase.SaveData - Ended");

        mutex.Close();
    }
}

class ThreadMutexApp
{
    public static Database db = new Database();
```

(jatkuu)

```
    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Worker thread #1 - Started");

        Console.WriteLine
            ("Worker thread #1 - Calling Database.SaveData");
        db.SaveData("x");

        Console.WriteLine("Worker thread #1 - Returned from Output");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Worker thread #2 - Started");

        Console.WriteLine
            ("Worker thread #2 - Calling Database.SaveData");
        db.SaveData("o");

        Console.WriteLine("Worker thread #2 - Returned from Output");
    }

    public static void Main()
    {
        ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
        ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

        Console.WriteLine("Main - Creating worker threads");

        Thread t1 = new Thread(worker1);
        Thread t2 = new Thread(worker2);

        t1.Start();
        t2.Start();
    }
}
```

Database-luokka määrittelee nyt *Mutex*-kentän. Emme halua säikeen omivan vielä mutexia, koska meillä ei ole tapaa päästä käsittelemään *SaveData*-metodia. *SaveData*-metodin ensimmäinen rivi näyttää, miten sinun tulee yrittää hankkia mutex eli *Mutex.WaitOne*-metodilla. Metodin lopussa suoritetaan *Close*-metodin kutsu, joka vapauttaa mutexin.

Myös *WaitOne*-metodi on ylikuormitettu joustavammaksi eli voit sen avulla nyt määritellä, miten pitkän ajan säie odottaa mutexin vapautuvan. Tässä funktion ylikuormitukset:

WaitOne( )
WaitOne(TimeSpan *time*, bool *exitContext*)
WaitOne(int *milliseconds*, bool *exitContext*)

Perusero näiden ylikuormitusten välillä on se, että esimerkissä käytetty ensimmäinen muoto odottaa loputtomasti ja toinen ja kolmas versio odottavat määrätyn ajan, jonka pituus on ilmaistu joko *TimeSpan*-arvona tai *int*-arvona.

Säieturvallisuus ja .NET-luokat

Kysymys, jonka usein näen uutisryhmissä ja postituslistoilla, kuuluu "Ovatko kaikki .NET System.*-luokat säieturvallisia?" Vastaus on "Ei, eikä niiden tulekaan olla." Pystyt vakavasti vahingoittamaan järjestelmän suorituskykyä, jos kaikki luokat turvaisivat toimintojensa käsittelyn. Kuvitellaan esimerkiksi yritystä käyttää jotakin kokoelmaluokkaa, jos se hankkisi monitorilukon joka kerta, kun kutsut sen *Add*-metodia. Sanotaan sitten, että instantioit kokoelmaobjektin ja lisää tuhatkunta objektia siihen. Suorituskyky olisi surkea ja tekisi järjestelmästä käyttökelvottoman.

Säikeistysohjeita

Milloin pitää käyttää säikeitä ja milloin niitä on vältettävä kuin ruttoa? Tässä kappaleessa sekä yleisiä tilanteita, joissa säikeet voivat olla äärettömän hyödyllisiä sovelluksellesi ja joitakin tilanteista, joissa niiden käyttöä on vältettävä.

Milloin säikeitä tulee käyttää

Sinun tulee käyttää säikeitä, kun pyrit kasvattamaan samanaikaisuutta, yksinkertaistamaan suunnittelua ja käyttämään prosessorin ajan paremmin hyödyksi, kuten seuraavissa kappaleissa tarkemmin kerrotaan.

Samanaikaisuuden lisääminen

Hyvin usein sovelluksen pitää suorittaa useampaa kuin yhtä tehtävää kerrallaan. Kirjoitin esimerkiksi kerran pankille dokumentin hakujärjestelmän, joka käsitteli optisilla levyillä isossa levytelineessä olevia tietoja. Puhumme tässä valtavasta määrästä tietoa. Levytelineessä oli 50 levyä ja gigatavuittain tietoja. Joskus saattoi kestää 5 tai jopa 10 sekuntia etsiä ja ladata haluttu dokumentti. Tarpeetonta sanoakaan, että olisi ollut suunnatonta tehokkuuden haaskausta, jos ohjelma olisi ollut jumissa aina dokumentin haun ajan. Siksi, jotta käyttäjä olisi voinut jatkaa töitään, käynnistin toisen säikeen, joka teki

varsinaisen fyysisen dokumentin haun. Tämä säie ilmoitti pääsäikeelle, kun dokumentti oli ladattu. Tämä on erinomainen esimerkki riippumattomista toiminnoista (dokumentin haku ja käyttöliittymän käsittely), jotka voidaan käsitellä kahdessa erillisessä säikeessä.

Suunnittelun helpottaminen

Suosittu tapa helpottaa monimutkaisen järjestelmän suunnittelua on käyttää jonoja ja asynkronista käsittelyä. Siten sinulla on jonoja, jotka on asetettu käsittelemään erilaisia järjestelmässäsi tapahtuvia tapahtumia. Sen sijaan, että metodeja kutsuttaisiin suoraan, objekti luodaan ja sijoitetaan jonoon, josta ne vuorollaan käsitellään. Toisaalta nämä jonot ovat palvelinohjelmia, joissa on useita säikeitä asetettuna kuuntelemaan jonoista tulevia viestejä. Tämäntyppisen yksinkertaisen suunnitelman etuna on se, että se tarjoaa luotettavan, vakaan ja laajennettavan järjestelmän.

Prosessoriajan parempi hyödyntäminen

Monta kertaa sovelluksesi ei varsinaisesti tee mitään vaikka käyttääkin aikajaksoaan. Dokumentin hakuesimerkissä yksi säie odotti levytelineen lataavan levyn. Tämä odotus on selvästi laitteistoasia eikä tarvitse prosessoriaikaa. Muita esimerkkejä odotusajoista ovat dokumentin tulostuksen odottelu tai kiintolevyn tai cd-rom-aseman odottelu. Kussakin tapauksessa prosessoria ei hyödynnetä. Ne ovat hyviä ehdokkaita siirrettäväksi taustasäikeen tehtäväksi.

Milloin säikeitä ei tule käyttää

Ne, jotka eivät ole paljon käyttäneet säikeitä, tekevät usein sen yleisen virheen, että yrittävät sovittaa niitä jokaiseen sovellukseen. Tulos voi olla paljon huonompi kuin ilman säikeitä. Kuten mikä muu käytössäsi oleva ohjelmointityökalu tai -menetelmä, säikeitäkin tulee käyttää vain silloin kun tilanne on sopiva. Sinun tulee välttää säikeiden käyttöä ohjelmassasi ainakin seuraavissa tapauksissa (jotka kuvataan seuraavissa kappaleissa): kun kustannukset ovat suuremmat kuin saavutettavat edut, kun et ole testannut molempien vaihtoehtojen nopeutta ja kun et keksi syytä niiden käyttämiseen.

Kustannukset ovat suuremmat kuin hyödyt

Kuten näit kappleessa “Säieturvallisuus ja synkronointi,” monisäikeisen sovelluksen kirjoittaminen vaatii enemmän aikaa suunnitteluvaiheessa. On tilanteita, joissa lievät edut, joita säikeiden käytöllä saavutetaan, eivät ole lisääntyneen suunnittelu- ja koodausajan vuoksi kannattavia toteuttaa.

Et ole testannut molempia vaihtoehtoja

Jos et ole ennen kirjoittanut monisäikeisiä sovelluksia, saatat yllättyä, kun huomaat että usein säikeen luominen ja ajoitus vie prosessorilta enemmän aikaa kuin mitä prosessorin parampi hyödyntäminen säikeiden avulla tuo. Yksisäkeinen ohjelma saattaa jopa toimia nopeammin! Kaikki riippuu siitä, mitä olet tekemässä ja oletko oikeasti jakamassa itsenäisiä tehtäviä säikeille. Jos sinun esimerkiksi pitää lukea kolme tiedostoa levyltä, kolmen säikeen käynnistäminen ei ole hyvä idea, koska kukin niistä käyttää samaa kiintolevyä. Muista siksi aina suorittaa testit järjestelmäsi yksisäikeisellä ja monisikeisellä versiolla ennen kuin päätät käyttää aikaa ja kustannuksia sellaisen ratkaisun toteuttamiseen, joka loppujen lopuksi saattaa olla hitaampi.

Ei hyvää syytä säikeiden käyttöön

Useiden säikeiden käyttö ei saa olla lähtötilanne. Monisäikeisen sovelluksen kirjoittamisen monimutkaisuuden vuoksi lähtökohta tulee aina olla yksisäikeinen sovellus ja vasta hyvistä syistä voit ryhtyä tekemään monisäikeistä sovellusta.

Yhteenveto

Monisäikeisyyden avulla sovellukset voidaan jakaa itsenäisiin tehtäviin ja siten käyttää prosessoriaika mahdollisimman hyvin hyödyksi. Säikeiden lisääminen sovellukseen ei kuitenkaan ole aina hyvä vaihtoehto ja voi joskus jopa hidastaa sovellusta. Säikeiden hallinta ja luonti tehdään C#:ssa *System.Threading.Thread*-luokalla. Tärkeä säikeiden luontiin ja käyttöön liittävä seikka on säieturvallisuus. Säieturvallisuus tarkoittaa, että objektin jäsenet säilyttävät aina kelvollisen tilan, kun sitä käytetään eri säikeistä. On tärkeää, että samalla, kun opit säikeiden käytön, ymmärrät myös, minkä vuoksi niitä tulee käyttää: samanaikaisuuden lisäämiseksi, suunnittelun helpottamiseksi ja prosessoriajan paremmaksi hyödyntämiseksi.