Ministerul Educaţiei al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică şi Microelectronică
Departamentul Inginerie Sofware și Automatică

Raport

Lucrare de laborator nr. 2
la disciplina: *Programarea în rețea*
Varianta 6
Tema: Programarea multi-threading

A verificat: lector asistent Ostapenco S.
lector asistent Latu E.

A efectuat: st. gr. TI-142 Cornita Constantin

Chișinău 2017

Obiectivele lucrării

Să se cunoască metodele de creare și execuție a firelor de execuție Java/C#. Să se utilizeze tehnicile de sincronizare în cadrul aplicațiilor concurente Java/C#.

Scopul lucrării

- realizarea firelor de execuție în Java/ C#;
- analiza proprietăților firelor de execuție, stărilor unui fir de execuție;
- lansarea, suspendarea și oprirea unui fir de execuție;
- utilizarea elementelor pentru realizarea comunicării și sincronizării.

Sarcina lucrării

Utilizând repozitoriul creat- https://github.com/CornitaConstantin/Lab2-PR.git fiind dată diagrama dependențelor cauzale (figura 1) de modelat activitățile reprezentate de acestea prin fire de execuție.

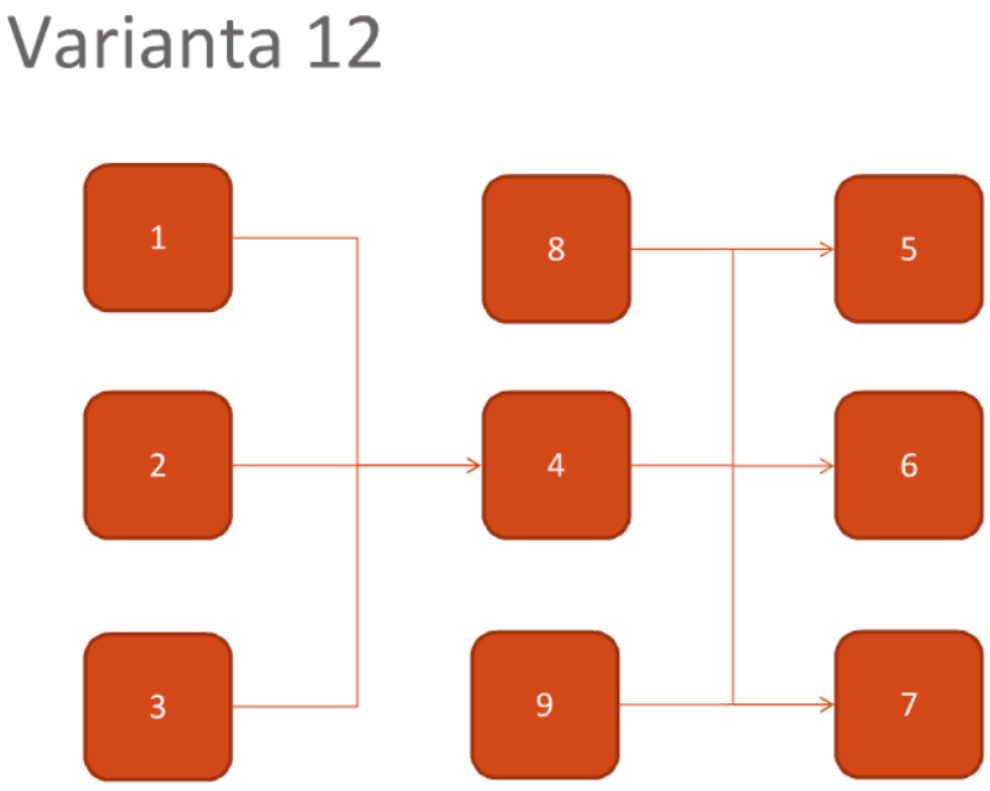

Figura 1- Diagrama dependențelor cauzale

Diagrama dependențelor cauzale determină o mulțime ordonată de evenimente/ activități ordonate de relația de cauzalitate. Evenimentele/ activitățile sunt reprezentate printr-un dreptunghi rotunjit, iar dependențele prin săgeți, primind astfel un graf orientat aciclic.

Introducere

De multe ori, aplicațiile au nevoie de mult timp pentru a rezolva o sarcină (descărcarea unui fișier, printară, generarea unui raport etc), timp în care programul nu poate răspunde unei alte acțiuni a utilizatorului. Pentru a face ca o aplicație sa îndeplinească o sarcină și sa poată primi și altele în același timp, sunt folosite firele de execuție multiple (*multiple threads*).

Într-un program liniar se execută o singură linie de cod, se așteaptă pentru completare și apoi se continuă cu următoarea linie de cod. Programul nu poate răspunde acțiunii unui utilizator în acest timp și chiar în cazul mai multor procesoare, va fi folosit doar unul singur, limitând performanța, datorită programării *single-thread*.

Un fir de execuție (*thread)* este un program secvențial, care poate fi executat concurent cu alte fire. Un thread este o unitate de execuție într-un proces. Un proces poate avea mai multe fire de execuție, el numindu-se *multi-thread.* Dacă un calculator are mai multe procesoare sau un procesor cu mai multe nuclee, el poate executa mai multe fire de execuție simultan.

Diferența dintre un proces și un thread este că procesele sunt izolate total unul de celălalt, în timp ce thread-urile împart aceeași memorie (heap) cu alte thread-uri care rulează în aceeași aplicație (un thread poate prelua informații noi, în timp ce un alt thread le prelucrează pe cele existente).

Folosirea firelor de execuție este o soluție la îmbunătățirea performanței. Se cere însă foarte multă atenție la folosire. Scrierea unui cod multi-thread este complexă, iar problemele care pot apărea pot fi foarte greu de rezolvat.

Trecerea de la starea *gata de executare* la starea în *curs de executare* are loc când un procesor îl alege pentru executare. La terminare, trece în starea *terminat*.

Un proces în curs de executare poate fi suspendat (*amânat* sau *blocat*), după care poate reveni în substarea *gata de executare*.

Proces *amânat* înseamnă întreruperea executării procesului un anumit timp, în urma apelării procedurii predefinite *sleep*; această procedură are un singur parametru de tip întreg.

Proces *blocat* înseamnă întreruperea executării procesului pe baza unei relații (comunicări) cu alte procese; procesul poate reveni ulterior în starea *gata de executare* pe o bază similară. Esențial în folosirea firelor de execuție multiple este evitarea conflictelor de resurse, când vor trebuie blocate anumite resurse, pentru a putea fi folosite de către un singur fir de execuție, pe rând.

Cel mai simplu mod de a crea un thread este să instanțiem un obiect *Thread*, al cărui constructor va cere ca parametru un delegate de tipul *ThreadStart*. Delegatul va indica ce metodă va rula în thread.

Metode des utilizate

Metoda *Sleep* se folosește atunci când vrem să suspendăm un fir de execuție, pentru o anumită perioadă. Metoda primește ca parametri un integer care reprezintă numărul milisecundelor cât va fi suspendat firul de execuție.

Metoda *Join, se presupune că atunci când se intenționează ca* un thread să aștepte un alt thread sa își termine execuția, înainte ca thread-ul curent să continue. Metoda *Abort se utilizează atunci când se intenționează să se oprească un fir de execuție.*

Folosind proprietatea *Priority*, se poate determina ce timp de execuție are un thread în comparație cu alte thread-uri active din acelaşi proces:

În lucrul cu multi-threading-ul apare o problema majoră, cea a sincronizării: mai multe fire de execuții accesează acelaşi obiect, simultan. Soluția vine de la "lock": atunci când primul thread accesează obiectul, îl va ține "ocupat" pînă la încheiere. Folosirea firelor de execuție permite rularea proceselor simultan, dar acest lucru poate încetini execuția programului, daca nu sunt folosite cu atenție. O aplicație C# poate deveni multi-threading în două moduri: fie explicit prin crearea și rularea firelor de execuție, fie prin folosirea unor caracteristici ale .Net care creează implicit thread-uri: BackgroundWorker, thread pooling sau la construirea unui Web Service sau unei aplicații Asp.Net [1].

Ciclul de viață a unui fir de execuție

Ciclul de viață a thread-ului începe atunci când obiectul din clasa `System.Threading.Thread` este creat și distrus atunci când thread-ul s-a finisat.

Stările unui thread sunt prezentate în figura 2.

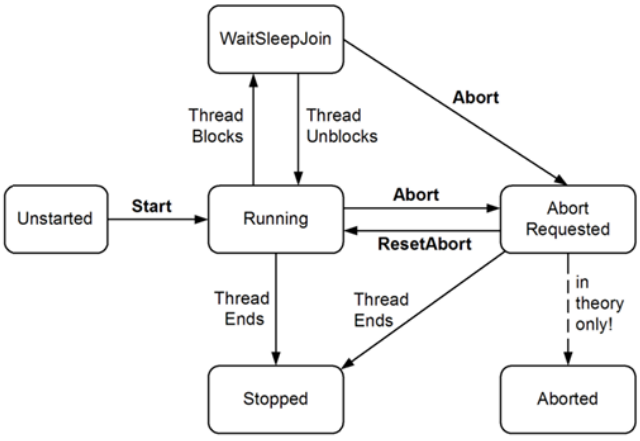

Figura 2 - Stările unui thread

- Running, un thread a fost startat, nu este blocat și nu există așteptare în `ThreadAbortException`;
- StopRequested, firul este rugat să se oprească, este folosit numai în cadrul său;
- SuspendRequested, firul este rugat să fie suspendat;
- Background, firul este executat ca unul din background. Această stare este controlată de proprietatea `Thread.IsBackground`;
- Unstarted, metoda `Thread.Start` nu a fost chemată;
- Stopped, thread-ul a fost oprit;
- WaitSleepJoin, firul este blocat, aceasta putea să fie provocată în urma Thread.Sleep sau Thread.Join, sau la cererea lui lock, ca de exemplu chemarea `Monitor.Enter` sau `Monitor.Wait`, sau în așteptarea unui obiect ca `ManualResetEvent;`
- Suspended, firul a fost suspendat;
- AbortRequested, metoda `Thread.Abort` a fost invocată, dar firul nu a primit incidentul `System.Threading.ThreadAbortException` care încearcă să-l termine;
- Aborted semnifică că firul este terminat, dar starea lui nu a fost schimbată în Stopped.

Sincronizarea thread-urilor

Blocking

Simple blocking methods, firele se așteaptă unu pe altul ca să se finiseze în decurs la o perioadă de timp. Metodele care fac parte din această categorie: `Sleep, Join, Task.Wait`. Un fir este considerat blocat atunci când execuția este oprită din oarecare motiv, așa ca `Sleeping` sau în așteptarea unui alt fir `Join` sau `EndInvoke`. Firul blocat imediat își rezervă timpul procesorului și așteaptă atâta timp cât procesorul este blocat.

Locking

Locking constructs, această metodă limitează numărul de fire care pot efectua oarecare activitate sau a executa o secțiune de cod în același timp. Construcțiile exclusive permit doar unui fir să se execută într-o perioadă de timp și permite accesul la date fără ca ele să se blocheze unul pe altul.

- constructorii exclusivi: `lock (Monitor.Enter/Monitor.Exit), Mutex și SpinLock`;
- constructori non exclusivi: `Semaphore, SemaphoreSlim` și `reader/writer locks;.`

Blocarea exclusivă este utilizată pentru a asigura că doar un singur thread poate intra în secție particulară a codului într-o perioadă de timp. Construcțiile exclusive de blocare sunt `lock` și `Mutex`. Din ambele, `lock` este mult mai rapid și convenabil. Pe de altă parte, `Mutex` are o nișă în cadrul blocărilor deoarece poate să deschidă aplicațiile în diferite procese pe calculator.

În C# `lock` de fapt reprezintă o metodă de a apela la `Monitor.Enter` și `Monitor.Exit`, cu ajutorul blocului `try/finally`. Apelarea `Monitor.Exit` fără a apelarea `Monitor.Enter` asupra unui obiect poate să prindă o excepție.

Semaphore

Semaforul este caracterizat cu o capacitate anumită. Odată cei plin nici un thread nu poate intra, formând o coadă de thread-uri în urma sa. Odată când se eliberează un loc, poate intra alt thread din coadă. În constructor este necesar de specificat numărul de locuri disponibile la momentul dat și numărul de locuri în total.

Signaling constructs, permit thread-urilor să se oprească până ce nu primesc notificare de la alt fir. Metodele standarde de semnalizare sunt: `event wait handles` și `Monitor's Wait/Pulse`;.

AutoResetEvent

Este ca un turnichet: înserarea unui bilet permite de a trece unui thread. "Auto" din numele său semnifică că turnichetul deschis va fi închis sau resetat automat după ce un fir a trecut prin el. Thread-ul așteaptă sau blochează turnichetul apelând metoda `WaitOne()`, iar biletul de intrare este metoda `Set()`. Dacă thread-ul apelează `WaitOne()`, se formează o coadă în fața turnichetului. Iar biletul de intrare poate veni de la orice fir, cu alte cuvinte orice thread neblocat cu accesul la `AutoResetEvent` poate apela `Set()` în el pentru a oferi biletul de trecere [2].

ManualResetEvent

El funcționează ca o poartă. Apelând `Set()` se deschide poarta, oferind oricărui număr de thread-uri să apeleze `WaitOne()` pentru a fi lăsat să treacă. Apelând `Reset()` se închide poarta. Thread-ul care apelează `WaitOne()` în poarta închisă va fi blocat, când data viitoare poarta se deschide, toate firele vor fi eliberate.

Barrier

Este primitivă de sincronizare definită de utilizator care permite thread-urilor multiple de a lucra concurent pe baza unui algoritm în faze. Orice participant execută atâta timp până el nu ajunge la punctul de barieră scrisă în cod. Bariera reprezintă faza când sa terminat lucrul. Când participantul ajunge la barieră, el este blocat atâta timp cât ceilalți participanți nu ajung și ei la aceasta barieră. După ce toți participanți au ajuns la barieră, poate fi invocată o acțiune post-fază. Această acțiune poate fi folosită pentru un singur thread în timp ce alte thread-uri sunt blocate. După ce acțiunea a luat sfârșit, toți participanții sunt deblocați.

CountdownEvent

System.Threading.CountdownEvent reprezintă sincronizarea primitivă care deblochează firele de așteptare după ce a fost semnalată de un anumit număr de ori. CountdownEvent este proiectat pentru scenarii în care altfel ar trebui să se utilizeze un ManualResetEvent sau ManualResetEventSlim și manual decrementează variabilele înaintea semnalării evenimentului. De exemplu, într-un scenariu fork/join, se poate crea un CountdownEvent care are 5 semnale, iar apoi dă start celor 5 elemente lucrătoare în thread pool și fiecare element de lucru are semnal de apelare, când sunt complete. Fiecare apel la semnal decrementează semnalul cu o unitate. În main thread, apelarea metodei Wait se va bloca până ce contorul semnalului este egal cu zero [2].

CountdownEvent are următoarele caracteristici adiționale:

- așteptarea operației poate fi anulată de utilizarea a tokens cancellation;
- contorul semnalului poate fi incrementat după ce a fost creată instanța;
- instanțele pot fi reutilizate după ce Wait a fost returnat de apelul metodei Reset;
- instanțele expun WaitHnandle pentru integrarea cu alte .Net Framework de sincronizare a API-urilor cum ar fi WaitAll.

Concluzii

Crearea unui nou thread este eficientă din punct de vedere al costului și overhead-ului implicat, mărind productivitatea prin deservirea unor cereri în timp ce altele sunt blocate. Deoarece ele partajează o mare parte din resursele unui proces, scrierea de programe multi-threaded solicită un efort de planificare suplimentar. Pot apărea inconsistențe în situația în care nu s-a realizat sincronizarea anumitor variabile. Depanarea programelor multi-threaded este, de asemenea, mai dificilă. Un program care folosește thread-uri pentru rezolvarea unei probleme de calcul intens nu va rula mai rapid pe un sistem uni-procesor. Schimbarea de context în cazul thread-urilor este mai puțin costisitoare. Thread-urile cer mai puține resurse și devine o soluție practică pentru rularea de programe multi-threaded pe sisteme uni-procesor.

Bibliografie

1 Eugen Popescu, Despre fire de executie in c# (threads) [Resursă electronică]. – Regim de acces: http://blog.zeltera.eu/?p=354

2 MSDN, CountdownEvent [Resursă electronică]. – Regim de acces: https://msdn.microsoft.com/en-us/library/dd997365%28v=vs.110%29.aspx

Anexa A

Codul sursă

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace na_PR2
{
    class Program
    {

        private static readonly CountdownEvent Countdown1 = new CountdownEvent(1);
        private static readonly CountdownEvent Countdown2 = new CountdownEvent(1);
        private static readonly CountdownEvent Countdown3 = new CountdownEvent(1);
        private static readonly CountdownEvent Countdown4 = new CountdownEvent(1);
        private static readonly CountdownEvent Countdown5 = new CountdownEvent(1);
        private static readonly CountdownEvent Countdown6 = new CountdownEvent(1);
        private static readonly CountdownEvent Countdown7 = new CountdownEvent(1);
        private static readonly CountdownEvent Countdown8 = new CountdownEvent(1);
        private static readonly CountdownEvent Countdown9 = new CountdownEvent(1);


        static void Main(string[] args)
        {
            Thread thread1 = new Thread(firstThread);
            Thread thread2 = new Thread(secondThread);
            Thread thread3 = new Thread(thirdThread);
            Thread thread4 = new Thread(fourThread);
            Thread thread5 = new Thread(fiveThread);
            Thread thread6 = new Thread(sixThread);
            Thread thread7 = new Thread(sevenThread);
            Thread thread8 = new Thread(eightThread);
            Thread thread9 = new Thread(nineThread);


            thread1.Start();
            thread2.Start();
            thread3.Start();
            thread4.Start();
            thread5.Start();
            thread6.Start();
            thread7.Start();
            thread8.Start();
            thread9.Start();


            Console.ReadKey();
        }


        static void firstThread()
        {
            Console.WriteLine(" thread  1");
            Countdown1.Signal();

        }
        static void secondThread()
        {
            Console.WriteLine(" thread  2");
            Countdown2.Signal();
```

```
        }
        static void thirdThread()
        {
            Console.WriteLine(" thread  3");
            Countdown3.Signal();
        }
        static void fourThread()
        {
            Countdown1.Wait();
            Countdown2.Wait();
            Countdown3.Wait();
            Console.WriteLine(" thread  4");
            Countdown4.Signal();
        }
        static void fiveThread()
        {
            Countdown4.Wait();
            Countdown8.Wait();
            Countdown9.Wait();
            Console.WriteLine(" thread  5");

        }
        static void sixThread()
        {
            Countdown4.Wait();
            Countdown8.Wait();
            Countdown9.Wait();
            Console.WriteLine(" thread  6");

        }
        static void sevenThread()
        {
            Countdown4.Wait();
            Countdown8.Wait();
            Countdown9.Wait();
            Console.WriteLine(" thread  7");

        }
        static void eightThread()
        {
            Console.WriteLine(" thread  8");
            Countdown8.Signal();
        }
        static void nineThread()
        {
            Console.WriteLine(" thread  9");
            Countdown9.Signal();
        }

    }
}
```

Anexa B

Rezultatele execuției programului

Prima execuție:

```
thread  2
thread  3
thread  1
thread  4
thread  8
thread  9
thread  5
thread  6
thread  7
```

A doua execuție:

```
thread  1
thread  2
thread  3
thread  4
thread  8
thread  9
thread  5
thread  7
thread  6
```