embarcadero
reseller partner

Primož Gabrijelčič
http://thedelphigeek.com

MARAND
Napredna računalniška hiša

Kazalo

Vsi programi, omenjeni v tem dokumentu, so na voljo na naslovu http://17slon.com/EA/EA-Multithreading-1_1.zip.

Uvod

Zadnjih petdeset let smo programerji živeli lagodno življenje. Ko je program postal prepočasen, smo enostavno rekli: »Nič hudega, čez pol leta bodo zunaj novi procesorji in zadeva bo LETELA!«

Zdaj pa že nekaj let ni več tako. Procesorji sicer vsebujejo vse več tranzistorjev, njihova groba moč (frekvenca, s katero delujejo), pa se ne povečuje več. Namesto tega proizvajalci v en procesorski čip vgrajujejo več procesnih enot oziroma *jeder* (*core* po angleško). Procesor lahko zaradi tega izvaja več operacij hkrati. To pa večini programerjev ne koristi kaj dosti, saj ne programirajo na način, ki bi to lahko izkoristil. Med »običajnim« programiranjem nastane program, ki v svojem življenjskem ciklu izrabi le eno jedro procesorja, druga pa medtem počivajo. Izkoristimo jih lahko, tako da poženemo več programov hkrati, ali pa uporabimo *večnitni* (*multithreaded*) program, ki zna početi več reči sočasno.

Če vemo, da tipičen program v Delphiju izkoristi le eno jedro in da je pri tem lahko večji del procesorja (ti imajo dandanes tudi 8 jeder) neizkorišečm, se hitro vprašamo, zakaj je tako. Zakaj programi ne izkoriščajo vseh jeder? Odgovor je žal zastrašujoč – zato, ker je večnitno programiranje brezmejno zapleteno in je strašno težko napisati program, ki v vseh možnih pogojih deluje pravilno. Večnitno programiranje je v praksi možno le, če se zavedamo vseh nevarnosti in programiramo na način, ki možnost pojavitve težav kar najbolj zmanjša.

V tem priročniku boste spoznali temelje večnitnega programiranja. Ogledali si bomo vire težav pri takem načinu dela in metodologije, s katerimi lahko vseeno pridemo do delujočega in hitrega programa. Spoznali bomo nekaj knjižnic, ki večnitno programiranje olajšajo dovolj, da se ga lahko loti tudi »običajen« programer, ne pa samo redki »superzvezdniki«. Za konec pa smo priložili še nasvete za razhroščevanje večnitnih programov.

Procesi in niti

Preden se lotimo samega programiranja, si oglejmo nekaj osnov delovanja operacijskih sistemov.

Osnovna enota izvajanja v večini sodobnih operacijskih sistemov je *proces*. Proces je pravzaprav le program, naložen v pomnilnik, ter vsa sistemska sredstva, ki jih program trenutno uporablja. K procesu tako štejemo dodeljeni pomnilnik, okna in druga grafična sredstva, datoteke in podobno. Proces opisuje »knjigovodsko« stanje programa, ne pa tudi tega, na kateri točki izvajanja je program trenutno. Stanje izvajanja imenujem *nit* (*thread*) in obsega trenutne vrednosti registrov centralno procesne enote, vrednosti na programskem skladu in pomnilniške lokacije, ki so lastne sami niti.

Vsak proces vsebuje vsaj eno nit, ki jo operacijski sistem izdela, ko se program naloži in ki jo zaključi, ko program umre. Imenujemo jo *glavna* nit. Programsko pa lahko izdelamo nove niti. Takšen način dela imenujemo *večnitno* (*multithreaded*) programiranje.

Zavedati se moramo, da je, vsaj na operacijskem sistemu Windows, zagon novega procesa »težko« opravilo, ki zahteva ogromno procesorskih ukazov. Zagon nove niti je bistveno »lažji«, oziroma zahteva manj procesorskega časa.

Nekateri starejši operacijski sistemi večnitnega dela sploh ne poznajo. Edini način za sočasno izvajanje kode je zagon več procesov. To so izkoristili tudi programerji in izdelali »vzporedne« programe, pri katerih je prvi proces pognal še nekaj svojih kopij, nato pa med njimi razdelil podatke, ki jih je bilo treba obdelati. Takšen način programiranja je bolj zapleten od zgoraj omenjenega večnitnega pristopa, saj operacijski sistemi procese med seboj *izolira*. Stanje enega procesa (torej njegov pomnilnik, sistemska sredstva …) lahko vidi le proces sam, drugi procesi pa ne. Če hočemo, da procesi sodelujejo, moramo vzpostaviti mehanizem za izmenjavo podatkov med njimi.

Pri večnitnem delu pa vse niti znotraj procesa »vidijo« njegov celoten pomnilniški prostor. Izmenjava podatkov je zato zelo enostavna, saj lahko vse niti vedno dostopajo do vseh podatkov. Se pa zaradi tega pojavijo težave s sočasnim dostopom do podatkov. Če na primer ena nit podatke bere, druga pa jih spreminja, se morata pri tem nekako uskladiti, sicer bo lahko prva prebrala napačne (okvarjene) podatke. Še huje je, če ima programska koda napako (bug), saj lahko takrat ena napačno sprogramirana nit pokvari podatke vsem drugim nitim.

Opis niti v tem priročniku je prilagojen operacijskemu sistemu Windows, ki ga pretežno uporabljamo za programiranje z Delphijem. Če bi radi razširili svoje znanje s splošnim opisom, pokukajte na Wikipedijo: http://en.wikipedia.org/wiki/Thread_(computer_science).

Zaradi takšnih težav se vzporedno procesiranje s poganjanjem več procesov še vedno uporablja v praksi. Že res, da je sodelovanje med procesi težje, a so procesi tudi povsem ločeni in če ima eden od njih težave, to nikakor ne vpliva na druge. Takšen pristop na primer uporablja brskalnik Chrome, ki za vsak zavihek požene ločen proces.

Hkratno in »hkratno« izvajanje

Pri izvajanju programov nismo omejeni s številom jeder v procesorju. Poženemo lahko bistveno več programov, kot imamo jeder, ti programi pa lahko uporabljajo veliko število niti. Kar pomislite nekaj let nazaj – praktično vsi računalniki so imeli le en procesor z le enim jedrom, pa smo na njemu vseeno izvajali več programov hkrati. Kako je to mogoče?

Trik se imenuje *predkupna večopravilnost* (*pre-emptive multitasking*). Operacijski sistem privošči kratek čas delovanja (nekaj deset milisekund) enemu programu, nato pa ga »zamrzne«. Naslednjih nekaj deset milisekund poganja drugi program, nato tretjega in tako dalje, dokler programov ne zmanjka in ne pride spet na vrsto prvi. [V resnici je dogajanje bolj zapleteno, programi se lahko tudi »prehitevajo«, a te podrobnosti ne vplivajo pomembno na bistvo razlage.]

Pravzaprav operacijski sistem ne razporeja procesov, temveč niti znotraj procesov. Nekaj časa dela ena nit, nato druga (ki ne pripada nujno istemu procesu kakor prva) in tako dalje. Če procesov/niti ni preveč, ima uporabnik občutek, da vsi procesi (oziroma niti v njih) tečejo hkrati.

V primeru več procesorjev ali več jeder na procesorju operacijski sistem deluje popolnoma enako, le da se takšno preklapljanje med nitmi dogaja na vseh jedrih hkrati. V vsakem trenutku tako nekaj niti teče resnično hkrati (vsaka na svojem jedru), nekaj pa le navidezno hkrati (izmenično).

Zakaj večnitno programiranje?

Rekli smo (dokazali pa še ne, a tudi to pride na vrsto), da je večnitno programiranje zapleteno. Zakaj in kdaj se ga vseeno lotevamo?

Najpogostejši razlog – vsaj v poslovnih programih – je, da lahko z večnitnim programiranjem daljše operacije izvedemo v ozadju in s tem sprostimo uporabniški vmesnik programa. Včasih lahko na tak način celo poenostavimo programsko kodo. Tipičen primer so operacije nad datotekami.

Branje in pisanje lahko izvajamo *sinhrono* (nit, ki je poklicala sistemsko funkcijo za branje/pisanje se ustavi, dokler se operacija ne izvede) ali *asinhrono* (sistemska funkcija se vrne takoj, ko je branje/pisanje dokončano, pa nekako obvesti program o tem. Prvi način je bistveno enostavnejši, drugi pa ne blokira uporabniškega vmesnika. Če operacije z datoteko izvedemo v ločeni niti, lahko tam uporabimo enostavnejši sinhroni vmesnik, uporabniški vmesnik pa se zaradi tega ne bo ustavljal.

> *Dobro je vedeti: Za delovanje uporabniškega vmesnika v programih, narejenih z Delphijem,* ***vedno*** *skrbi glavna nit – to je tista nit, ki jo privzeto naredi operacijski sistem ob zagonu programa. Druge niti – tiste, ki jih ustvarimo sami – tečejo neodvisno od uporabniškega vmesnika in nanj* ***ne smejo*** *vplivati. Če v takšni niti pokličemo katerokoli funkcijo iz knjižnice VCL, lahko povzročimo sesutje programa. Funkcije iz knjižnice RTL, torej tiste, ki niso vezane na grafični vmesnik, so varne in jih lahko uporabljamo.*

Drugi razlog je, da lahko na tak način (torej z večnitnim programiranjem) sočasno izvajamo več operacij. Na tak način lahko nek postopek izvedemo hitreje, kot če bi ga izvajali v eni sami niti. Kako dobro se da nek problem *paralelizirati* (predelati tako, da teče na več hkratnih nitih), je odvisno od primera do primera. Značilen primer, ko je predelava na vzporedno delo enostavna, so strežniške aplikacije, ki strežejo več odjemalcem. V tem primeru lahko naredimo več niti, vsaka pa obdeluje eno posamezno zahtevo.

Primer problema, ki ga lahko predelamo za vzporedno delo, a predelava ni trivialna, je urejanje. Nekatere algoritme za urejanje (denimo QuickSort) lahko z nekaj truda predelamo v vzporedno, hitrejšo različico.

Težave večnitnega programiranja

Omenili smo že, da je večnitno programiranje težko. Glavni razlog za to je hkrati tudi glavni razlog za popularnost večnitnega programiranja – vse niti vidijo vsa sredstva procesa in zato lahko brez večjih zapletov sodelujejo med sabo. Zaradi istega razloga pa vse prelahko pokvarijo podatke ena drugi.

Poglejmo si enostaven primer [program IncDec]. Imamo dva podprograma, ki spreminjata isto spremenljivko – FValue. Prvi program jo stotisočkrat poveča za ena, drugi jo stotisočkrat zmanjša za ena. Na začetku je v spremenljivki vrednost 0. Kakšna bo vrednost spremenljivke na koncu?

```
procedure TfrmIncDec.IncValue;
var
  i: integer;
  value: integer;
begin
  for i := 1 to 100000 do begin
    value := FValue;
    FValue := value + 1;
  end;
end;

procedure TfrmIncDec.DecValue;
var
  i: integer;
  value: integer;
begin
  for i := 1 to 100000 do begin
    value := FValue;
    FValue := value - 1;
  end;
end;
```

Če poženemo podprograma enega za drugim, je odgovor jasen. Vrednost stotisočkrat povečamo in stotisočkrat zmanjšamo in na koncu bo enaka, kakor na začetku – nič. Kaj pa, če oba podprograma tečeta hkrati, vsak v svoji niti? V tem primeru nimamo najmanjšega pojma, kakšen bo odgovor! Razlog za to je, da operaciji povečanja/pomanjšanja nista *nedeljivi* (*atomic*). Z drugimi besedami, med izvajanjem seštevanja/odštevanja se lahko izvede poljubno število operacij v drugih nitih.

Za lažje razumevanje si oglejmo enega od možnih scenarijev izvajanja.

- Prva nit začne izvajanje in prebere vrednost FValue v začasno spremenljivko value.
- Prva nit se nato ustavi. Morda je sistem zelo obremenjen ali pa poganjate program na enojedrnem procesorju in operacijski sistem potrebuje procesor, da lahko požene drugo nit.
- Druga nit se začne izvajati in stotisočkrat zmanjša FValue. Na koncu izvajanja v spremenljivko naloži vrednost -100.000.
- Prva nit nadaljuje z izvajanjem. Vrednosti v začasni spremenljivki (to je začetna vrednost spremenljivke FValue, torej 0) prišteje ena in rezultat (1) shrani v FValue. Vrednost FValue je s tem skočila iz -100.000 na 1!
- Prva nit nadaljuje z izvajanjem, še 99.999-krat poveča vrednost in na koncu v FValue zapiše 100.000.

Še huje je, da Intelova arhitektura razen v zelo posebnih primerih ne zagotavlja neprekinljivosti (atomarnosti) tako enostavnih operacij kot so branje in pisanje pomnilnika.

Poglejmo si primer [program ReadWrite]. Prva nit v zanki zapisuje dve vrednosti v pomnilniško lokacijo velikosti int64. Pomnilniška lokacija je z nekaj truda nastavljena tako, da se bo pisanje izvedlo v dveh korakih. Med prirejanjem FPValue^ := $7777777700000000; bo procesor najprej zapisal vrednost $00000000 v prve štiri bajte spremenljivke, nato pa šele $77777777 v druge štiri bajte. Pri zapisovanju $0000000077777777 se zgodi podobno.

```
procedure TfrmReadWrite.Writer;
var
  i: integer;
begin
  for i := 1 to 100000 do begin
    FPValue^ := $7777777700000000;
    FPValue^ := $0000000077777777;
  end;
end;
```

Druga nit bere podatke iz pomnilnika in jih zapisuje v seznam, ki je nastavljen tako, da meče duplikate stran (Sorted := true; Duplicates := dupIgnore;).

```
procedure TfrmReadWrite.Reader;
var
  i: integer;
begin
  for i := 1 to CNumRepeat do
    FValueList.Add(FPValue^);
end;
```

Po koncu izvajanja glavna nit izpiše vse vrednosti, ki jih je prebrala. V seznamu pričakovano najdemo $7777777700000000 in $0000000077777777, pa tudi $0000000000000000 in $7777777777777777. Do teh pride zaradi dvostopenjskega pisanja.

Denimo da je v pomnilniku zapisano $7777777700000000. Prva nit se loti pisanja $0000000077777777 in najprej v prve štiri bajte zapiše $77777777. Druga nit takrat prebere podatek in dobi $7777777777777777 – zgornji štirje bajti so v pomnilniku še od prejšnjega pisanja, spodnji štirje pa so bili ravnokar shranjeni.

Zaklepanje

Operacijski sistemi ponujajo nekaj različnih načinov reševanja problemov hkratnega dostopa do skupnih pomnilniških lokacij. Ena od možnih rešitev je *zaklepanje* (*locking*). Obstaja več različnih načinov za dosego istega cilja; za večnitne programe je napomembnejše zaklepanje s kritičnimi sekcijami.

Bistvo zaklepanja je, da naredimo **en** objekt tipa TCriticalSection (razred je deklariran v enoti SyncObjs) in ga uporabljamo v **vseh** nitih, ki si **delijo** neko sredstvo. Pred dostopom do sredstva pokličemo metodo Acquire in po dostopu metodo Release. Med tema dvema klicema je kritična sekcija zaklenjena (rečemo tudi, da si jo nit *lasti*) in če v tem trenutku druga nit pokliče Acquire, se bo njeno izvajanje ustavilo, dokler prva nit ne pokliče Release.

```
procedure TfrmIncDec.LockedIncValue;
var
  i: integer;
  value: integer;

begin
  for i := 1 to 100000 do begin
    FLock.Acquire;
    value := FValue;
    FValue := value + 1;
    FLock.Release;
  end;
end;

procedure TfrmIncDec.LockedDecValue;
var
  i: integer;
  value: integer;
begin
  for i := 1 to 100000 do begin
    FLock.Acquire;
    value := FValue;
    FValue := value - 1;
    FLock.Release;
  end;
end;
```

Zdaj druga nit ne more prekiniti trenutka, ko prva nit spreminja polje FValue, ker ne more pridobiti kritične sekcije.

Enako metodo lahko uporabimo pri branju in pisanju.

Drugi način je raba *interlocked* operacij namesto običajnega seštevanja/odštevanja. Interlocked operacije poskrbijo za zaklepanje na nivoju procesorskega ukaza. Povečevanje/zmanjševanje se v takem primeru v resnici izvede neprekinljivo (atomarno). Te operacije so deklarirane v enoti Windows (imena podprogramov se začno z »Interlocked«), vedeti pa moramo, da delujejo ukazi pravilno le, če so podatki shranjeni na pravilnih lokacijah v pomnilniku (rečemo, da so podatki pravilno *poravnani*) .

```
procedure TfrmIncDec.InterlockedIncValue;
var
  i: integer;
begin
  for i := 1 to CNumRepeat do
    InterlockedIncrement(FValue);
end;

procedure TfrmIncDec.InterlockedDecValue;
var
  i: integer;
begin
  for i := 1 to CNumRepeat do
    InterlockedDecrement(FValue);
end;
```

Ker je doseganje pravilne poravnanosti podatkov včasih zelo zapleteno, knjižnica OmniThreadLibrary (o kateri bomo več povedali kasneje) prinaša podatkovna tipa TGp4AlignedInt in TGp8AlignedInt64 – to sta primerno poravnana *integer* in *int64*. Podatkovna tipa spotoma definirata še funkcije, ki kličejo operacije Interlocked. V našem primeru sta to enostavni .Increment in .Decrement.

```
procedure TfrmIncDec.InterlockedIncValue2;
var
  i: integer;
begin
  for i := 1 to CNumRepeat do
    FValue2.Increment;
end;

procedure TfrmIncDec.InterlockedDecValue2;
var
  i: integer;
begin
  for i := 1 to CNumRepeat do
    FValue2.Decrement;
end;
```

Zaklepanje sicer reši problem hkratnega dostopa do sredstev, a lahko pripelje do drugih problemov – mrtvih zank (deadlock), živih zank (livelock) in počasne kode.

Mrtva zanka je situacija, v kateri nit A čaka na kritično sekcijo, ki jo ima trenutno v lasti nit B, ta pa čaka na kritično sekcijo, ki jo ima trenutno v lasti nit A. Zaradi tega nobena od njiju ne more nadaljevati in program se zaustavi. Iskanje teh vrst napak je enostavno – ko se program »zatakne« ga ustavimo v razhroščevalniku in pogledamo, v kakšnem stanju so niti.

Živa zanka je problem, ki se ne pojavi pogosto, pomeni pa, da dve (ali več) niti zaradi načina, na katerega je spisan algoritem, ne moreta nadaljevati z delom, čeprav se program v nobenem trenutku ne zaustavi. V strokovni literaturi je znan problem *filozofov pri večerji*, ki zelo enostavno privede v živo zanko. (Več o tem: http://en.wikipedia.org/wiki/Dining_philosophers.)

Zadnji problem – počasnost – ne spada med programske napake, pač pa je stranski učinek zaklepanja. Operacija, ki zasede kritično sekcijo (Acquire), je včasih precej počasnejša od kode, ki jo hočemo na tak način zavarovati. V našem primeru je že tako – samo povečevanje spremenljivke je bistveno hitrejše od dostopa do kritične sekcije. Upočasnijo – in to še bolj drastično – pa se še vse druge niti, ki bi hotele v tem času dostopati do kritične sekcije, saj je ta zaklenjena in dostopa drugim nitim ne pusti.

Tudi način z *interlocked* operacijami je počasnejši od običajnega, nezaščitenega. Je pa tudi hitrejši od rabe kritičnih sekcij.

Drugi načini za vzpostavitev usklajenega izvajanja (pogosto jih imenujemo *sychronisation primitives*) so muteks (mutex), semafor (semaphore) in dogodek (event). Iz njih lahko sestavimo bolj zapletene kontrolne strukture, denimo »MREW« (multiple readers, exclusive writer), ki dovoli dostop eni pisalni niti ali poljubno mnogo bralnim nitim. Ti sinhronizacijski načini presegajo obseg te brošure. Več o njih si lahko preberete na spletu. (Seznam koristnih povezav je na koncu brošure.)

Komunikacija

Niti niso izolirane, temveč sodelujejo med seboj. To že vemo. Videli smo tudi, kako lahko niti z nekaj pazljivosti dostopajo do istih podatkov. Kaj pa takrat, ko bi radi delovanje niti bolj podrobno uskladili? Včasih samo zaklepanje ni dovolj, temveč moramo med nitmi prenašati tudi informacije.

V ta namen imamo nekaj različnih možnosti, ki pa so – kot se izkaže v praksi – vse presneto nerodne. Za začetek lahko uporabimo zgoraj omenjene *dogodke*. Z njimi lahko izvedemo enostavno signalizacijo, s katero ena nit pove drugi, kdaj naj se ustavi ali požene. Podobno lahko naredimo s *semaforji*.

Če bi radi prenašali večje količine sporočil, lahko med nitmi pošiljamo *sporočila* – tako kot jih Windows pošiljajo glavni niti v procesu. Tak način je sicer uporaben, ni pa najhitrejši.

Še ena možnost je, da iz skupnega pomnilnika in sinhronizacijskih klicev sestavimo svoj sistem za posredovanje sporočil. Taka možnost je sicer najhitrejša in zelo primerna za primere, kjer prenašamo ogromno (stotisoče ali milijone) sporočil, je pa tudi strašno zapletena. Programiranje takšnega sistema, ki mora pravilno delovati v večnitnem okolju, je zahtevna naloga. [Vseeno je tak način zelo primeren, če uporabite obstoječo in dobro preizkušeno kodo. Več o tem v poglavju o knjižnici OmniThreadLibrary.]

Delphi in niti

Osnova večnitnega programiranja v Delphiju je razred TThread, ki je definiran v enoti Classes. Programiranje nove niti začnemo tako, da naredimo potomca tega razreda in v njem redefiniramo metodo Execute [program testTThread].

```
type
  TTestThread1 = class(TThread)
    procedure Execute; override;
  end;
```

V metodi napišemo kodo, ki se bo izvajala v svoji niti. Nekje drugje v programu pa to kodo poženemo, tako da naredimo nov primerek razreda TTestTread1.

```
  FThread1 := TTestThread1.Create;
```

Kako nit zaustavimo, je odvisno od kode, ki se v njej izvaja. Če se nit vrti v zanki, sprejema ukaze in jih izvaja, potem ji moramo sporočiti, da naj se ustavi, nato počakati, da to res stori, za konec pa uničiti objekt, shranjen v FThread1.

```
  FThread1.Terminate;
  FThread1.WaitFor;
  FreeAndNil(FThread1);
```

Da to pravilno deluje, mora biti tudi nit pravilno sprogramirana. Nit iz našega primera ne počne kaj dosti, samo vrti se v zanki in čaka da jo bo program ustavil.

```
procedure TTestThread1.Execute;
begin
  while not Terminated do begin
    // some real work could be done here
    Sleep(1000);
  end;
end;
```

Lastnost Terminated je lastna razredu TThread, na True pa jo nastavi klic metode Terminate.

Drugi način, uporaben za niti, ki izvedejo neko kodo in zaključijo delo (torej se ne vrtijo v zanki in ne čakajo na ukaze) pa je, da nastavimo lastnost FreeOnTerminate na True. S tem smo dosegli, da se bo potomec razreda TThread sam uničil, čim se podprogram Execute neha izvajati. Spotoma lahko nastavimo še dogodek OnTerminate, ki se bo sprožil, tik preden bo nit uničena.

```
  FThread2 := TTestThread2.Create(true);
  FThread2.FreeOnTerminate := true;
  FThread2.OnTerminate := ReportThreadTerminated;
```

Oba načina lahko poljubno kombinirate – naredite lahko nit, ki se vrti v zanki in se ustavi, ko nekdo pokliče .Terminate, ter se nato sama uniči, ker ima nastavljen FreeOnTerminate.

TThread

Razred TThread se je skozi zgodovino le malo spreminjal. Občasno sicer dobi kakšno novo metodo ali lastnost, vseeno pa tisto, kar ste se morda naučili v Delphiju 5, še vedno velja v Delphiju XE3.

Pomembnejše metode in lastnosti razreda TThread so prikazane na naslednjem koščku kode, pobranem iz Delphi XE3 RTL (© Embarcadero).

```
TThread = class
protected
  procedure Execute; virtual; abstract;
  procedure Queue(AMethod: TThreadMethod); overload;
  procedure Synchronize(AMethod: TThreadMethod); overload;
  procedure Queue(AThreadProc: TThreadProcedure); overload;
  procedure Synchronize(AThreadProc: TThreadProcedure); overload;
  property ReturnValue: Integer read FReturnValue write FReturnValue;
  property Terminated: Boolean read FTerminated;
public
  constructor Create; overload;
  constructor Create(CreateSuspended: Boolean); overload;
  destructor Destroy; override;
  class function CreateAnonymousThread(const
    ThreadProc: TProc): TThread; static;
  procedure Start;
  procedure Terminate;
  function WaitFor: LongWord;
  class function CheckTerminated: Boolean; static;
  class procedure Queue(AThread: TThread;
    AMethod: TThreadMethod); overload; static;
  class procedure Queue(AThread: TThread;
    AThreadProc: TThreadProcedure); overload; static;
  class procedure Synchronize(AThread: TThread;
    AMethod: TThreadMethod); overload; static;
  class procedure Synchronize(AThread: TThread;
    AThreadProc: TThreadProcedure); overload; static;
  class procedure RemoveQueuedEvents(AThread: TThread;
    AMethod: TThreadMethod); overload; static;
  class procedure RemoveQueuedEvents(AThread: TThread); overload; static;
  class procedure RemoveQueuedEvents(AMethod: TThreadMethod); overload;
    static;
  class procedure NameThreadForDebugging(AThreadName: AnsiString;
    AThreadID: TThreadID = TThreadID(-1)); static;
  class procedure SpinWait(Iterations: Integer); static;
  class procedure Sleep(Timeout: Integer); static;
  class procedure Yield; static;
  class function GetSystemTimes(out SystemTimes: TSystemTimes): Boolean;
    static;
  class function GetCPUUsage(var PrevSystemTimes: TSystemTimes): Integer;
    static;
  property FatalException: TObject read FFatalException;
  property FreeOnTerminate: Boolean read FFreeOnTerminate
    write FFreeOnTerminate;
  property Finished: Boolean read FFinished;
  property Handle: THandle read FHandle;
  property Priority: TThreadPriority read GetPriority write SetPriority;
  property ThreadID: TThreadID read FThreadID;
  property OnTerminate: TNotifyEvent read FOnTerminate
    write FOnTerminate;
  class property CurrentThread: TThread read GetCurrentThread;
  class property ProcessorCount: Integer read FProcessorCount;
  class property IsSingleProcessor: Boolean read GetIsSingleProcessor;
end;
```

Ko naredimo objekt tega razreda (*Create*), lahko določimo, ali se začne nit izvajati takoj, ali pa naj bo najprej v ustavljenem stanju (*CreateSuspended*). V slednjem primeru jo poženemo s klicem metode *Start*.

CreateAnonymousThread, ki je nova metoda v Delphi XE2, nam omogoča, da naredimo nit kar kot anonimno funkcijo, ne da bi naredili nov razred in implementirali Execute. Taka nit ima implicitno nastavljeno *FreeOnTerminate* na True.

Razne različice metod *Synchronize* in *Queue* omogočajo niti, da komunicira z glavnim programom. Omenili smo že, da knjižnice VCL ne smemo uporabljati iz lastnih niti. Če bi denimo radi nekaj zapisali v TListBox na obrazcu, tega ne smete narediti neposredno iz niti. Lahko pa napišete kratek podprogram, ki to naredi, ter ga s klicem Synchronize ali Queue izvedete v glavni niti. Nejasno? Poskusimo razčleniti to operacijo.

- Glavna nit naredi novo nit. Imenujmo jo »nit A«.
- Nit A hoče nekaj zapisati v seznam na obrazcu.
- Nit A zato pokliče Queue in mu poda podprogram.
- Delphi nekako spravi ta podprogram do glavne niti.
- Nit A se nato izvaja dalje.
- Glavna nit v nekem trenutku (ne moremo točno povedati, kdaj), pokliče podprogram, ki smo ga podali metodi Queue. Podprogram nekaj izpiše v seznam na obrazcu.

Bistvena razlika med Synchronize in Queue je, da prva metoda počaka, da bo glavna nit izvedla kodo, in šele nato nadaljuje z izvajanjem v lastni niti. Druga metoda pa samo postavi kodo v neko čakalno vrsto in takoj nadaljuje z izvajanjem.

Lastnost *ReturnValue* določa vrednost, ki jo bo vrnila funkcija WaitFor. Na ta način lahko nit prenese rezultat nazaj v glavno nit.

Če se v metodi Execute zgodi izjema (exception), ki je program ne ujame, bo TThread to izjemo »pospravil« v lastnost *FatalException*.

V okolju Windows ima vsaka nit svojo lastno ročico (handle), identifikator (threadID) in prioriteto (priority). Te vrednosti so dostopne preko lastnosti *Handle*, *ThreadID* in *Priority*.

Vse metode in lastnosti si lahko ogledate v spletni dokumentaciji:
`http://docwiki.embarcadero.com/Libraries/XE3/en/System.Classes.TThread.`

Delphi RTL

Delphijeva izvajalna knjižnica (RTL) vsebuje še nekaj koristnih dodatkov za večnitno programiranje.

Enota *SyncObjs* vsebuje sinhronizacijske primitive – kritično sekcijo (TCriticalSection), dogodek (TEvent), muteks (TMutex), semafor (TSemaphor) in druge.

`http://docwiki.embarcadero.com/Libraries/en/System.SyncObjs`

Namesto seznama TList lahko uporabimo *TThreadList*, ki ima vgrajeno sinhronizacijo s kritičnimi sekcijami. Do seznama dostopamo (in ga s tem zaklenemo), tako da pokličemo metodo LockList, sprostimo (in odklenemo) pa ga s klicem UnlockList.

http://docwiki.embarcadero.com/Libraries/en/System.Classes.TThreadList

Razred *TThreadedQueue* se je pojavil v Delphiju XE. Implementira vrsto (queue), ki je uporabna za prenos podatkov med nitmi. Zaradi programskih napak pa je postal uporaben šele v Delphiju XE2. Priporočamo, da za prenos podatkov raje uporabite podatkovne strukture iz knjižnice OmniThreadLibrary.

http://docwiki.embarcadero.com/Libraries/en/
System.Generics.Collections.TThreadedQueue

Razred TMonitor omogoča zaklepanje na nivoju posameznih objektov. Namesto da bi za nek objekt, ki ga želimo varovati (recimo mu *obj*), vpeljali novo kritično sekcijo, pokličemo System.TMonitor.Enter(obj), da objekt zaklenemo, ter System.TMonitor.Exit(obj), da ga sprostimo. Razred ima še celo množico metod, s katerimi lahko izdelamo zapletene večnitne programe, a so bile v Delphijih do XE2 polne programskih napak, zato rabo odsvetujemo. Poleg tega je TMonitor strašansko počasen. Priporočamo vam, da za varovanje raje uporabljate strukturo TOmniCS (poenostavljena kritična sekcija iz knjižnice OmniThreadLibrary) ali razred Locked<T> (prav tako iz knjižnice OmniThreadLibrary).

http://docwiki.embarcadero.com/Libraries/en/System.TMonitor

Večkrat smo že omenili, da je večnitno programiranje problematično, ker vse niti dostopajo do celotnega pomnilnika, ki pripada procesu. No, to ni povsem res. Vsaka nit ima tudi svoj privatni prostor, imenovan *Thread Local Storage (TLS)*. V Delphiju ga lahko uporabimo za shranjevanje niti lastnih spremenljivk. Namesto, da bi jih najavili z ukazom *var*, uporabimo ukaz *threadvar*.

http://docwiki.embarcadero.com/RADStudio/en/Using_Thread-local_Variables

Alternative

TThread je sicer dobra osnova za programiranje večnitnih programov, a v praksi se hitro izkaže, da zna biti delo z njim presneto štorasto. Marsikateri programer, ki se resno ukvarja z večnitnim programiranjem, si je zato sprogramiral knjižnico, ki ovija TThread in olajša njegovo rabo. Večina teh knjižnic ostane omejenih na enega uporabnika, nekatere pa so zapustile varno okolje domačega računalnika in uspešno pomagajo programerjem širom sveta pri reševanju večnitnih problemov.

AsyncCalls

AsyncCalls (http://andy.jgknet.de/blog/bugfix-units/asynccalls-29-asynchronous-function-calls/) je knjižnica vsestranskega Andreasa Hausladna, ki ga marsikdo pozna po njegovih dodatkih za pospešitev in boljše delovanje Delphija (IDE Fix Pack). Knjižnica je usmerjena na reševanje nezapletenih problemov, osredotoča pa se na enostavno poganjanje niti in sinhronizacijo med njimi.

Primerček kode, ki z uporabo AsyncCalls požene enostaven podprogram, ki glavnemu programu sporoča, kaj trenutno počne:

```
TAsyncCalls.Invoke(procedure begin
  TAsyncCalls.VCLInvoke(
    procedure begin ReportProgress(MSG_THREAD_START); end);
  // some real work could be done here
  Sleep(5000);
  TAsyncCalls.VCLInvoke(
    procedure begin ReportProgress(MSG_THREAD_STOP); end);
end);
```

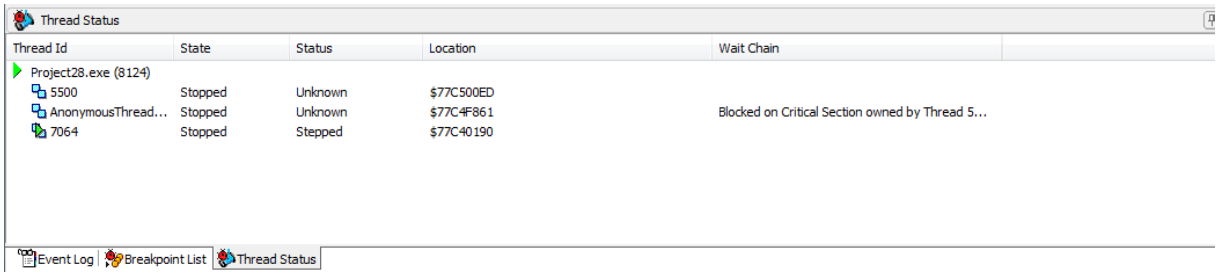
OmniThreadLibrary

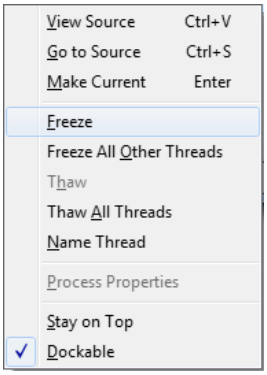
OmniThreadLibrary (http://otl.17slon.com) je knjižnica za večnitno programiranje, ki jo je (s pomočjo nekaterih slovenskih in tujih programerjev) napisal avtor te skripte. Iz pripomočka za enostavno poganjanje niti in njihovo medsebojno komunikacijo je z leti prerasla v vseobsegajoče orodje, s katerim lahko programirate niti na nivoju TThread, ali pa uporabite orodja za viskonivojsko vzporedno programiranje, kot je na primer vzporedna zanka »for«. Knjižnica se aktivno razvija (zadnja različica, 3.01, je bila izdana pred slabim mesecem), veliko programerjev iz vsega sveta pa jo je že sprejelo za »svojo«. Zaradi tega, ker omogoča večnitno programiranje tudi programerjem, ki se s takim pristopom srečujejo prvič, ji bomo posvetili več prostora v nadaljevanju.

Razhroščevanje

Iskanje napak v večnitnih programih je težavno. Malo zaradi tega, ker so večnitni programi nepredvidljivi – nikoli ne moremo vedeti, v kakšnem zaporedju se bodo izvajali ukazi v dveh hkrati pognanih nitih –pa tudi zato, ker s samim postopkom razhroščevanja vplivamo na ta vrstni red. Delno nam pri tem pomaga Delphijev razhroščevalnik, ki je v novejših Delphijih dobil nekaj s tem povezanih novosti.

Ena od novosti je »wait chain traversal«. Čuden izraz opisuje novost v Windows Vista, ki zna v primeru mrtve zanke (deadlock) opisati, kako je do nje prišlo. Če »zataknjen« program začasno ustavimo (klik na ikono za pavzo), bomo v oknu Thread Status videli, katera nit čaka na kritično sekcijo (ali karkoli drugega), kar ima trenutno v lasti druga nit.

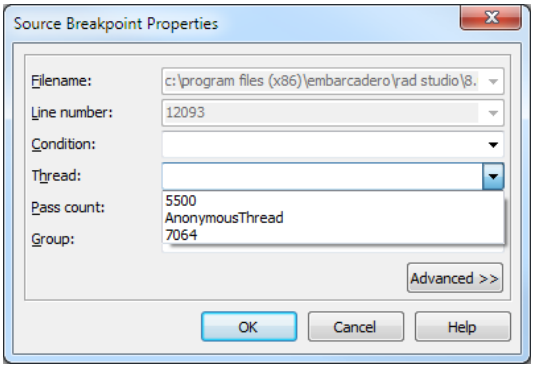




Že dolgo imamo v Delphiju možnost poimenovanja niti za potrebe razhroščevanja, vendar smo morali za to napisati svoj podprogram in ga poklicati. Od Delphija 2010 dalje pa imamo na voljo metodo TThread.NameThreadForDebugging, s katero niti določimo ime. To ime bo vidno v oknu Thread Status. (Na zgornji sliki je na tak način poimenovana nit »AnonymousThread«.) Če nit nima imena, ga lahko določimo, tako da kliknemo nit z desno mišjo tipko in iz pojavnega menuja izberemo »Name Thread«. Na tak način določeno ime pa bo delovalo samo do ustavitve programa.

Uporabne možnosti, ki jih najdemo v pojavnem menuju, so še *Freeze* (zamrzne trenutno izbrano nit, ko nadaljujemo z izvajanjem programa, se ta nit ne bo izvajala), *Freeze All Other Threads* (zamrzne vse niti, razen trenutno izbrano), *Thaw* (»odtaja« trenutno nit, torej ji spet dovoli izvajanje) in *Thaw All Threads* (»odtaja« vse niti).

Ko je program v začasno ustavljenem stanju, lahko z dvojnim klikom niti v oknu Thread Status to nit prikažemo v razhroščevalniku. Okni Call Stack in Local Variables bosta prikazali trenutno stanje sklada in lokalnih spremenljivk za izbrano nit.



Za večnitno razhroščevanje je uporabna še možnost, da prekinitveno točko (breakpoint) vežemo na točno določeno nit. Če bo isto mesto v kodi izvedla neka druga nit, se izvajanje programa ne bo prekinilo.

Nasveti

Večnitno programiranje je težko. Razlika med večnitnim programom, ki deluje in večnitnim programom, ki deluje samo na razvijalčevem računalniku, pa še to ne vedno, je lahko le razlika med sistematičnim pristopom in programiranjem na način »bo že kako«. Za boljše delovanje programov se je koristno držati nekaterih pravil, ki jih je potrdila praksa.

Če se le da, napišite testni program, ki preizkuša večnitno kodo. Še posebej to velja za kose kode, ki skrbijo za sodelovanje med nitmi (dostop do skupnih sredstev ali prenašanje sporočil). Testni program naj kodo preizkusi velikokrat. Enkrat – pri večnitnem programiranju – ni nobenkrat. Testni programi za kritične dele OmniThreadLibraryja, na primer, zasedejo računalnik za celo noč.

Če je algoritem splošen in deluje s poljubnim številom niti (oziroma se program prilagaja računalniku, na katerem teče in uporabi toliko niti, kolikor ima računalnik jeder), ga vedno preizkusite z različnim številom niti, od dveh do dvakrat toliko, kolikor jeder ima testni računalnik. Testiranje s veliko nitmi (bistveno več, kot je jeder) zelo obremeni računalnik in hitro pokaže na kritične točke.

Posebno pozornost posvetite arhitekturi programa. Ta naj bo kar najbolj enostavna, točke interakcije med nitmi (zaklepanje zaradi skupnega pomnilnika, komunikacija) pa čim bolj izolirane. Če se le da, uporabljajte za sodelovanje med nitmi sporočila in ne deljenega pomnilnika.

Če uporabljate zaklepanje, vedno zaklenite kar najmanjšo možno količino kode. Večnitni programi, ki večino dela opravijo znotraj zaklenjene kritične sekcije, so počasni.

Če uporabljate skupna sredstva, ki niso (logično) povezana med seboj, jih zaklenite z različnimi kritičnimi sekcijami. Raba le ene kritične sekcije za zaklepanje vsega po vrsti, bo upočasnila program.

Če se algoritem strašno zaplete pri predelavi v večnitno obliko, potem ga boste verjetno morali zamenjati. Nekateri algoritmi niso primerni za večnitno delo.

Če se le da, uporabljajte preizkušene knjižnice in ne lastnih umotvorov. Več ljudi kot uporablja neko knjižnico, bolj verjetno je, da bodo našli večino programskih napak v njej.

OmniThreadLibrary

OmniThreadLibrary je knjižnica za večnitno programiranje, izdana z odprtokodno licenco (OpenBSD licencse). Projekt ima domačo stran na naslovu http://otl.17slon.com, izvorna koda pa je shranjena na straneh Google Code (http://omnithreadlibrary.googlecode.com). Trenutno najnovejša različica (3.02) je izšla oktobra 2012 in podpira Delphi od 2007 do XE3. Podprto je le okolje VCL, ne pa tudi FireMonkey, kodo pa lahko prevedete za 32-bitno ali 64-bitno platformo Windows.

Namestitev

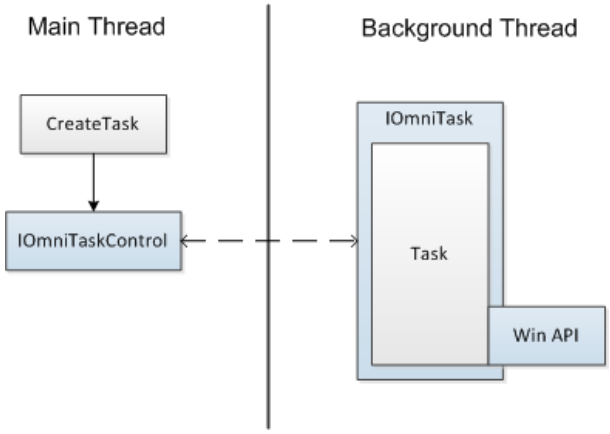
1. Prenesite zadnjo stabilno različico (3.02) in jo odpakirajte ali pa izvozite SVN repozitorij iz Google Code.
2. V »Library path« dodajte mapo, v katero ste shranili OmniThreadLibrary in njeno podmapo »src«.
3. Dodajte potrebne enote v stavek »uses« in začnite uporabljati knjižnico! Če boste uporabljali visokonivojske konstrukte (paralelni for in podobno), v večini primerov zadošča, da dodate enoto OtlParallel.

Opravila in sporočila

OmniThreadLibrary temelji na dveh principih. Namesto z nitmi program dela z *opravili*, deljeni pomnilnik pa poskuša kar najbolj nadomestiti s *sporočili*. Seveda pa še vedno dopušča programerju dostop do niti, souporabo pomnilnika, zaklepanje in druge programske pristope.

Opravilo (task) predstavlja kodo, ki naj se izvede, ter točke interakcije med kodo in ostalim programom. OmniThreadLibrary nato to opravilo izvede v niti, ki jo bodisi ustvari na zahtevo ali pa dodeli iz bazena (thread pool). Razlika v kodi je minimalna – če nad opravilom pokličemo metodo .Run, bo OmniThreadLibrary naredil novo nit, če pokličemo .Schedule, pa bo nit vzel iz bazena.

Na strani klicatelja z opravilom upravljamo preko vmesnika IOmniTaskControl, na strani niti pa preko vmesnika IOmniTask.



OmniThreadLibrary sicer podpira zaklepanje (zgoraj omenjena vmesnika poznata metodo .WithLock, s katero kritično sekcijo delimo med glavno in pomožno nitjo, lahko pa uporabimo tudi svojo lastno kodo za zaklepanje), večino dela znotraj knjižnice pa je vseeno postorjenega s posredovanjem sporočil. Komunikacijski vmesniki so izdelani z uporabo skupnega pomnilnika in *interlocked* ukazov in so izredno hitri – skoznje brez težav prenesemo več milijonov sporočil na sekundo. Osnovna enota

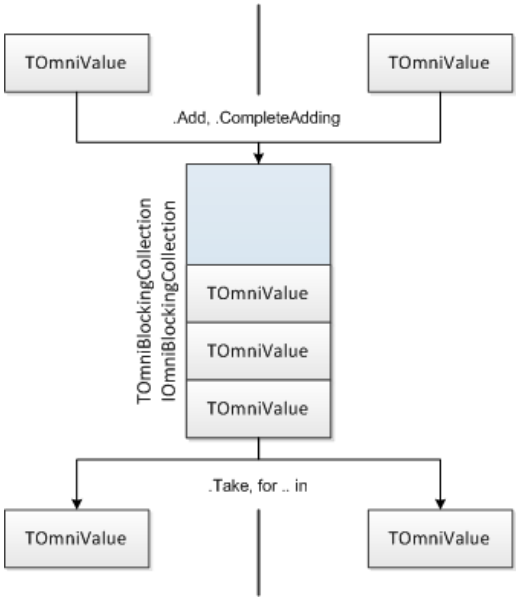
podatkov, ki se prenaša preko komunikacijskih vmesnikov, je struktura TOmniValue, ki je v bistvu nekakšen zelo hiter Variant in lahko sprejme karkoli – od celih števil do objektov in vmesnikov.

Gradniki

OmniThreadLibrary je sestavljen iz več enot, zato je dobro vedeti, kaj se v kateri od njih skriva, da jih lahko po potrebi dodamo v seznam *uses.* Osnovne podatke in strukture (OtlCommon, OtlContainers, OtlCollections, OtlSync) lahko uporabite tudi v kombinaciji z drugimi načini nitenja (TThread, AsyncCalls).

Najpomembnejše enote so:

- OtlTask
  Vsebuje definicijo vmesnika IOmniTask. Potrebuje ga koda, ki se izvaja v opravilu in ki kakorkoli spreminja lastnosti opravila ali hoče komunicirati z glavno nitjo.
- OtlTaskControl
  Vsebuje definicijo vmesnika IOmniTaskControl in večino programske logike, povezane z ustvarjanjem in poganjanjem opravil. Vmesnik IOmniTaskControl potrebuje glavni program, da upravlja in komunicira z opravilom v ozadju.
- OtlThreadPool
  Implementacija *bazena niti*. Uporabljate lahko privzeti bazen ali pa naredite svojega.
- OtlCommon
  Vsebuje razne strukture, uporabljene v knjižnici. Strukture so uporabne tudi za rabo v vaših programih. Najpomembnejša med njimi je TOmniValue, opozorili pa bi še na funkcijo Environment, s katero lahko dostopamo do podatkov o procesu.
- OtlComm
  Vsebuje strukture in funkcije za prenos sporočil med opravilom in njegovim lastnikom.
- OtlContainers
  Vsebuje hitre podatkovne strukture – sklad omejene velikosti, vrsto (queue) omejene velikosti in dinamično dodeljeno vrsto, ki raste po potrebi. Vse strukture dovoljujejo poljubno število hkratnih piscev in poljubno število hkratnih bralcev.
- OtlContainers
  Vsebuje *blocking collection*, vrsto, ki pri branju *blokira* (čaka, da se v vrsti pojavi podatek), dokler poljubna nit ne pokliče metode CompleteAdding (zaključi dodajanje). S tem je omogočeno izredno enostavno branje iz te strukture – uporabimo zanko for..in, ki se bo zaključila šele, ko bo poljubna nit eksplicitno označila, da je dodajanje podatkov zaključeno. Ta podatkovna struktura je uporabljena v mnogih visokonivojskih vmesnikih iz enote OtlParallel. Tako kot strukture iz OtlContainers tudi ta podpira več hkratnih piscev in več hkratnih bralcev.



- OtlSync
  Strukture in funkcije za zaklepanje. Najbolj zanimive so TOmniCS (ovoj okoli TCriticalSection, ki ga ni treba posebej ustvariti, samo deklarirate ga), TOmniMREW (izredno hitre metode za zaklepanje po sistemu Multiple Readers, Exclusive Writer) in Locked<T> (zaklepanje z rabo generikov).
- OtlParallel
  Algoritmi za visokonivojsko večnitno programiranje (Async, Future, Join, ParallelFor …). Zahtevajo vsaj Delphi 2009.

Dokumentacija

Dokumentacija knjižnice OmniThreadLibrary je še v nastajanju. Trenutno so opisani visokonivojski konstrukti, sinhronizacijske metode in razredi ter primeri rabe nizkonivojskih in visokonivojskih konstruktov. Dokumentacija je na voljo v obliki elektronske knjige »Parallel Programming with OmniThreadLibrary« v zapisih PDF, EPUB in MOBI.

Knjiga je izdana po principih dinamičnega založništva (*lean publishing*), kar pomeni, da je na voljo za nakup, čeprav še ni dokončana. Vsi kupci knjige prejmejo tudi vse posodobitve, ki bodo izdane v prihodnosti – nova poglavja lahko torej berejo takrat, ko izidejo.

https://leanpub.com/omnithreadlibrary

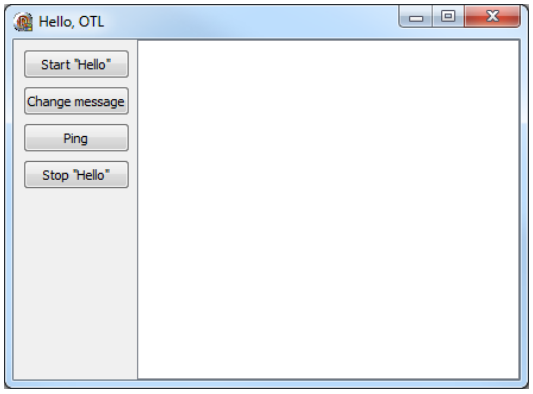
Spletna stran http://www.thedelphigeek.com/2012/06/parallel-programming-with.html vsebuje vedno sveže podatke o knjigi (pregled napisanih/predvidenih poglavij).

Spletna različice dokumentacije je na voljo na naslovu http://otl.17slon.com/book/doku.php.

Nizkonivojsko programiranje

OmniThreadLibrary podpira dva načina dela, ki ju lahko prosto kombinirate znotraj svoje kode – nizkonivojsko in visokonivojsko. Nizkonivojsko delo predstavlja alternativo programiranju na nivoju TThread, a z izredno uporabnimi dodatki, ki olajšajo komunikacijo z nitjo in sinhronizacijo več niti. Oglejmo si nizkonivojski program na primeru [program SimpleOTLTask].

Naš program ima nekaj gumbov in seznam (TListBox) za prikaz sporočil.



V obrazcu potrebujemo polje, ki bo shranilo vmesnik IOmniTaskControl. Definirati moramo še sporočilo, ki ga bo poslala nit (*MSG_LOG*) in metodo, ki bo to sporočilo obdelala (*LogMessage*).

```
const
  MSG_LOG = WM_USER;

type
  TfrmTestTwoWayHello = class(TForm)
  private
    FHelloTask: IOmniTaskControl;
    procedure LogMessage(var msg: TOmniMessage); message MSG_LOG;
  end;
```

Gumb »Start "Hello"« požene opravilo v samostojni niti.

```
  FHelloTask := CreateTask(TAsyncHello.Create(), 'Hello')
    .SetParameter('Delay', 1000)
    .SetParameter('Message', 'Hello')
    .OnMessage(Self)
    .OnTerminated(
       procedure
       begin
         lbLog.ItemIndex := lbLog.Items.Add('Terminated');
       end)
    .Run;
```

CreateTask je funkcija, ki naredi novo opravilo. Opravilo bo izvajalo kodo iz razreda TAsyncHello (ogledali si ga bomo kmalu), ime pa mu bo »Hello«.

Opravilu nastavimo nekaj parametrov (*SetParameter*) – sporočilo, ki naj ga izpisuje in razmik med sporočili. Vidimo, da CreateTask uporablja *tekoči* (*fluent*) način programiranja, pri katerem lahko klice metod nizamo enega za drugim. Podoben pristop je uporabljen v večini knjižnice OmniThreadLibrary. Če vam takšno delo ni všeč, lahko kličete metode na običajen način (FHelloTask.SetParamter(…); FHelloTask.OnMessage(…);).

S klicem metode *OnMessage* določimo, da bo sporočila, ki jih bo poslalo opravilo, sprejemal kar obrazec (form). Namesto tega bi lahko določili posebno metodo znotraj obrazca ali pa vsaki vrsti sporočil priredili svojo metodo.

S klicem metode *OnTerminate* določimo kodo, ki se bo izvedla, ko se opravilo zaključi. V tem primeru uporabimo kar anonimno funkcijo, ki bo ta dogodek zapisala v seznam.

S klicem metode *Run* ustvarimo novo nit in v njej poženemo opravilo.

Gumb »Stop "Hello"« zaustavi opravilo.

```
  FHelloTask.Terminate;
  FHelloTask := nil;
```

Opravilo moramo najprej zaustaviti (*Terminate*), nato pa počistiti spremenljivko, ki ima shranjen vmesnik IOmniTaskControl.

Ko uporabnik klikne gumb »Change message« koda s klicem *Comm.Send* opravilu pošlje sporočilo MSG_CHANGE_MESSAGE. Vsebina sporočila je naključno določeno.

```
procedure TfrmTestTwoWayHello.btnChangeMessageClick(Sender: TObject);
begin
  FHelloTask.Comm.Send(MSG_CHANGE_MESSAGE,
    Format('Random %d', [Random(1234)]));
end;
```

Ko uporabnik klikne gumb »Ping«, koda izvede metodo z imenom »Ping« v opravilu. Metoda *Invoke* deluje podobno kot TThread.Queue, le da jo lahko uporabimo v obeh smereh – tako da izvede kodo v glavni ali v pomožni niti. Ko pokličemo Invoke, se ustvari interno sporočilo, ki ga OmniThreadLibrary pošlje po komunikacijskem kanalu. Nit izvede kodo šele, ko to sporočilo obdela.

```
procedure TfrmTestTwoWayHello.btnHelloClick(Sender: TObject);
begin
  FHelloTask.Invoke('Ping');
end;
```

Metoda *LogMessage* se pokliče avtomatsko, ko nit pošlje sporočilo MSG_LOG. Metoda iz parametra potegne vsebino sporočila in jo izpiše v seznam.

```
procedure TfrmTestTwoWayHello.LogMessage(var msg: TOmniMessage);
begin
  lbLog.ItemIndex := lbLog.Items.Add(msg.msgData.AsString);
end;
```

Oglejmo si zdaj še drugo stran kode. Definirati moramo razred TAsyncHello in sporočilo MSG_CHANGE_MESSAGE, ki ga niti pošlje gumb »Change message«.

```
const
  MSG_CHANGE_MESSAGE = 1;

type
  TAsyncHello = class(TOmniWorker)
  strict private
    aiMessage: string;
  public
    function  Initialize: boolean; override;
    procedure ChangeMessage(var msg: TOmniMessage);
      message MSG_CHANGE_MESSAGE;
    procedure Ping;
    procedure TimerSendMessage;
  end;
```

Razred je izpeljan iz razreda *TOmniWorker*, ki vsebuje logiko za sprejem in obdelavo sporočil ter za poganjanje in ustavljanje opravila.

Redefinirali smo funkcijo *Initialize*, ki se avtomatsko pokliče ob zagonu opravila. V njej preberemo vrednosti parametrov in nastavimo časovnik (timer), ki se bo prožil na zahtevan interval.

```
function TAsyncHello.Initialize: boolean;
begin
  aiMessage := Task.Param['Message'];
  Task.SetTimer(0, Task.Param['Delay'], @TAsyncHello.TimerSendMessage);
  Result := true;
end;
```

Metoda *ChangeMessage* se bo poklicala vsakič, ko bo opravilo prejelo sporočilo MSG_CHANGE_MESSAGE. Metoda le zamenja vrednost internega polja.

```
procedure TAsyncHello.ChangeMessage(var msg: TOmniMessage);
begin
  aiMessage := msg.MsgData;
end;
```

Metoda *Ping* se bo poklicala, ko bo nit obdelala interno sporočilo, ki bo vsebovalo zahtevo klica Invoke. Metoda uporabi Invoke še enkrat, da v glavni niti doda besedilo v seznam. Tak način rabe ustreza klicu TThread.Queue.

```
procedure TAsyncHello.Ping;
begin
  Task.Invoke(
    procedure
    begin
      frmTestTwoWayHello.lbLog.ItemIndex :=
        frmTestTwoWayHello.lbLog.Items.Add('Pong!');
    end);
end;
```

Metoda *TimerSendMessage* se sproži vsakič, ko se izteče časovni interval, določen v klicu Task.SetTimer. Metoda pošlje sporočilo MSG_LOG po komunikacijskem kanalu.

```
procedure TAsyncHello.TimerSendMessage;
begin
  Task.Comm.Send(MSG_LOG, aiMessage);
end;
```

OmniThreadLibrary omogoča še bistveno več od opisanih osnov. Oglejte si primere, priložene knjižnici, ter članke na internetu (povezave najdete na koncu te brošure).

Visokonivojsko programiranje

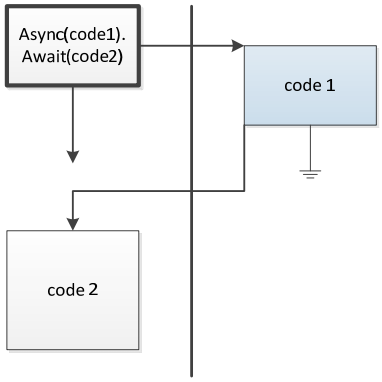
Visokonivojski vmesniki v OmniThreadLibrary so nastali z dvema namenoma. Približati večnitno programiranje programerjem, ki se prvič srečujejo s tem konceptom, ter popolnoma spremeniti način razmišljanja pri večnitnem programiranju. Namesto s tehnikalijami tipa »kreiraj nit«, »pošlji sporočilo« in podobno pri visokonivojskem načinu najprej izberemo ustrezno *abstrakcijo*, torej algoritem, ki kar najbolje ustreza našemu problemu, nato pa le še dodamo nekaj kode. Za ustrezno ogrodje poskrbi OmniThreadLibrary.

Vsi visokonivojski vmesniki živijo v enoti OtlParallel, ustvarimo pa jih tako, da pokličemo eno od metod razreda Parallel. Visokonivojska koda temelji na rabi generikov in anonimnih funkcij, zato za delo potrebuje vsaj Delphi 2009. Priporočljiva je raba vsaj Delphija 2010, ker ima različica 2009 ogromno z generiki povezanih bugov. Vsa visokonivojska opravila se izvajajo v bazenu GlobalParallelPool, ki je prav tako deklariran v enoti OtlParallel.

Vsi spodnji primeri so zajeti v priloženem programu MultithreadingMadeSimple.

Async/Await

Abstrakcija Async/Await je namenjena enostavnemu poganjanju kode, ki nima zahtev po interakciji z glavno nitjo, omogoča pa določanje bloka kode, ki se izvede po zaključku ozadne niti. Gre le za »olepšavo« (*syntactic sugar*), ki ovije abstrakcijo Async, opisano v naslednjem razdelku.

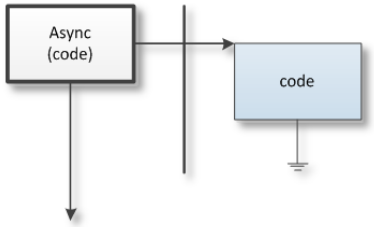


Spodnji fragment kode predstavlja zelo enostavnen primer, ki ob kliku gumba le-tega onemogoči ter mu zamenja napis, nato pa začne v pomožni niti izvajati podprogram, ki zaspi za pet sekund. Ob zaključku tega podprograma se v glavni niti izvede anonimna metoda, ki gumb zopet omogoči ter mu spremeni napis.

```
procedure TfrmMultithreadingMadeSimple.btnAsyncAwaitClick(Sender: TObject);
var
  button: TButton;
begin
  button := Sender as TButton;
  button.Caption := 'Working ...';
  button.Enabled := false;
  Async(
    procedure begin
      Sleep(5000);
    end).
  Await(
    procedure begin
      button.Enabled := true;
      button.Caption := 'Done!';
    end);
end;
```

Async

Abstrakcija *Async* je namenjena asinhronemu poganjanju kode, ki nima zahtev po interakciji z glavno nitjo. Ko pokličemo Parallel.Async se koda izvede v ločeni niti, program pa nadaljuje z drugimi opravili.



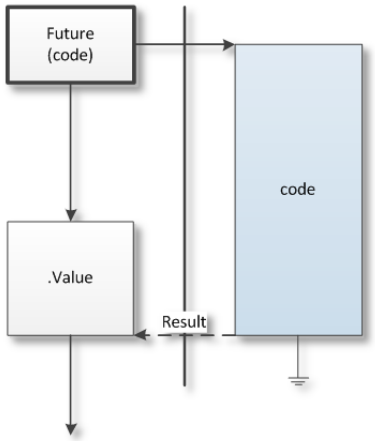
Primer kode, ki z abstrakcijo Async prebere vsebino spletne strani in jo shrani v datoteko:

```
procedure TfrmMultithreadingMadeSimple.btnAsyncGETClick(Sender: TObject);
begin
  Parallel.Async(
    procedure
    var
      page: string;
    begin
      HttpGet('17slon.com', 80, 'gp/biblio/articlesall.htm', page, '');
      SaveToFile('articlesall.htm', page);
    end);
end;
```

Če bi bili radi obveščeni, ko se Async zaključi, moramo glavni niti poslati sporočilo ali pa določiti metodo, ki se pokliče ob zaključku niti (OnTerminated). Primera sta v izvorni kodi, tu pa ju ne bomo poseben predstavljali. Razlog? Če morate vrednost vrniti glavnemu programu, potem Async ni prava abstrakcija. Raje uporabite Future.

Future

Future podobno kot Async v ločeni niti izvede kos kode, le da mora v tem primeru to biti funkcija, ki vrne nek rezultat. Rezultat bo avtomatsko dostopen v glavni niti, ko bo enkrat izračunan.



Poglejmo si primer:

```
function TfrmMultithreadingMadeSimple.FutureGET: string;
begin
  HttpGet('17slon.com', 80, 'gp/biblio/articlesall.htm', Result);
end;

procedure TfrmMultithreadingMadeSimple.btnFutureGETClick(Sender: TObject);
var
  getFuture: IOmniFuture<string>;
  page: string;
begin
  getFuture := Parallel.Future<string>(FutureGet);

  // do some other processing
  Sleep(1000);

  page := getFuture.Value;
end;
```

Podprogram *btnFutureGETClick* s klicem *Parallel.Future<string>* naredi opravilo, ki bo izvedlo podano kodo (FutureGet) ter vrne vmesnik IOmniFuture<string>. V našem primeru bomo iz niti vrnili vrednost tipa *string* (zato »<string>« v klicu in v imenu vmesnika), vrnili pa bi lahko tudi katerikoli drug osnovni ali sestavljeni tip.

Podprogram nato zaspi za eno sekundo in s tem simulira neko opravilo, nato pa vrednost, ki jo je vrnila funkcija FutureGET, prebere, tako da pokliče *getFuture.Value*. Če vrednost v tem trenutku še ni znana (torej če funkcija FutureGET še ni zaključila svojega dela), bo klic Value počakal (blokiral), dokler vrednost ne bo postala znana.

Tudi pri rabi abstrakcije Future bi pogosto radi vedeli, kdaj bo računanje zaključeno. Najlažje je, če določimo metodo, ki se sproži ob zaključku opravila. (Popolnoma enako bi lahko naredili tudi v primeru abstrakcije Async.)

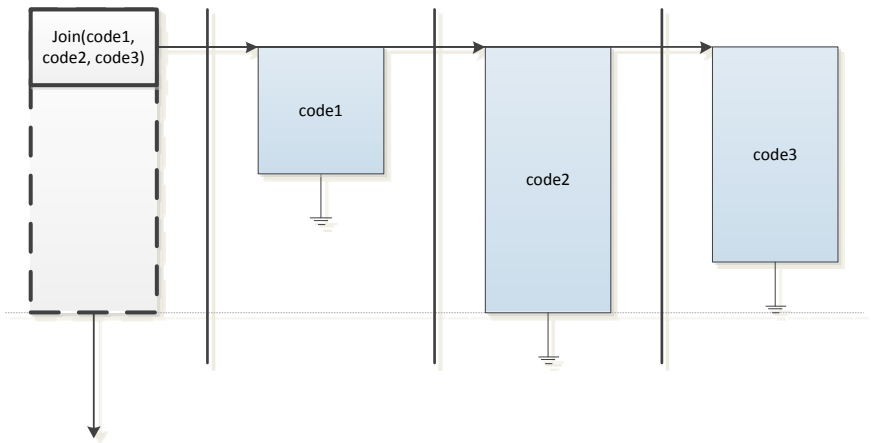
```
procedure TfrmMultithreadingMadeSimple.btnFutureGetExceptionClick(
  Sender: TObject);
begin
  FGetFuture := Parallel.Future<string>(
    function: string
    begin
      HttpGet('does.not.exist', 80, 'gp/biblio/articlesall.htm', Result);
    end,

    Parallel.TaskConfig.OnTerminated(FutureDone));
end;
```

Zgornja koda sproži izjemo (exception) v metodi HttpGet, ker domena »does.not.exist« ne obstaja. Ker izjeme ne obdelamo sami, jo Future ujame in jo prenese v glavno nit, kjer se bo sprožila, ko bomo poklicali .Value.

```
procedure TfrmMultithreadingMadeSimple.FutureDone;
var
  page: string;
begin
  try
    page := FGetFuture.Value;
  except
    on E: Exception do
      lbLogFuture.Items.Add(Format(
        'Exception caught in Future.Value: %s: %s',
        [E.ClassName, E.Message]));
  end;
  FGetFuture := nil;
end;
```

Join

Abstrakcija *Join* omogoča poganjanje več vzporednih opravil. Vsako teče v svoji niti. Glavni program lahko takoj nadaljuje z izvajanjem, ali pa počaka, da se vsa opravila dokončajo.



Primer programa, ki s tremi vzporednimi opravili prenese tri strani s spleta in jih shrani na disk. Klic .*Execute* počaka, da se opravila izvedejo do konca. Če bi želeli nadaljevati takoj, bi morali pred .Execute vriniti klic .*NoWait*.

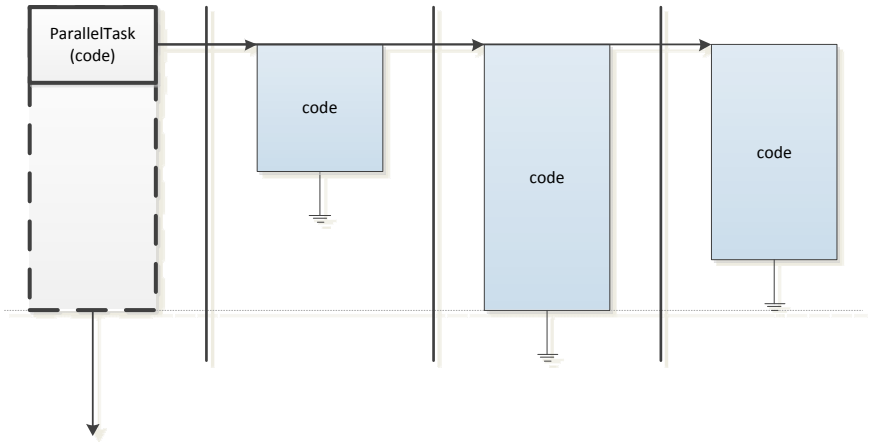
```
procedure TfrmMultithreadingMadeSimple.btnJoinClick(Sender: TObject);
begin
  Parallel.Join([
    procedure
    var page: string;
    begin
      HttpGet('17slon.com', 80, 'gp/biblio/articlesall.htm', page);
      SaveToFile('articlesall.htm', page);
    end,
    procedure
    var page: string;
    begin
      HttpGet('otl.17slon.com', 80, 'tutorials.htm', page);
      SaveToFile('tutorials.htm', page);
    end,
    procedure
    var page: string;
    begin
      HttpGet('thedelphigeek.com', 80, 'index.htm', page);
      SaveToFile('index.htm', page);
    end]).Execute;
end;
```

Tako kot Future tudi Join lovi izjeme in jih posreduje v glavno nit, kjer jih lahko obdelamo s try..except. Ker pa lahko izjemo sproži več opravil, jih Join ovije v lastno izjemo EJoinException. Oglejmo si le obdelavo takih izjem iz našega testnega programa.

```
procedure TfrmMultithreadingMadeSimple.btnJoinExceptionClick(
  Sender: TObject);
var
  iExceptTask: integer;
begin
  try
    Parallel.Join([
      …
      ]).Execute;
  except
    on E: EJoinException do begin
      for iExceptTask := 0 to E.Count - 1 do begin
        lbLogJoin.Items.Add(Format(
          'Caught exception in Join.Execute task #%d: %s:%s',
          [E.Inner[iExceptTask].TaskNumber,
           E.Inner[iExceptTask].FatalException.ClassName,
           E.Inner[iExceptTask].FatalException.Message]));
      end;
    end;
  end;
end;
```

ParallelTask

Abstrakcija *ParallelTask* je zelo podobna abstrakciji *Join*, le da v vseh nitih izvaja isto kodo. Glede na to, da morajo ta opravila nekako dobiti podatke (in jih morda tudi nekako vrniti) in da pri tem lahko več opravil hkrati bere ali piše, za komunikacijo z opravili običajno uporabimo *blocking collection* iz enote OtlCollections.



Spodnji podprogram uporabi ParallelTask za generiranje množice naključnih podatkov v več vzporednih opravilih. Za vhod dobi želeno količino podatkov in podatkovni tok, v katerega jih je treba zapisati.

Podprogram si najprej naredi *blocking collection*, v katerega bodo opravila zapisovala zgenerirane podatke. Nato naredi števec (IOmniCounter), ki bo štel, koliko podatkov je še treba zgenerirati. Števec je varen za večnitno delo in ne potrebuje posebnega zaklepanja.

Klic Parallel.ParallelTask.NoWait zagotovi, da glavni program ne bo čakal na opravila, temveč bo nadaljeval z izvajanjem. Število opravil nastavimo na število jeder, dostopnih procesu (klic NumTasks). Določimo, da se ob končanju vseh opravil izvede Parallel.CompleteQueue(outQueue). Ta pomožni podprogram iz enote OtlParallel zaključi pisanje v *blocking collection* (pokliče metodo CompleteAdding). Klic Execute nato sprejme kodo, ki se izvaja v vseh opravilih.

Vsako opravilo najprej naredi svoj lastni generator psevdonaključnih števil TGpRandom. (Ta ni prirejen za večnitno delo in ga zato ne smemo deliti med nitmi.) Nato iz skupnega števca jemlje do *CBlockSize* velike kose in za vsakega zgenerira blok naključnih števil in ta blok porine v *blocking collection*.

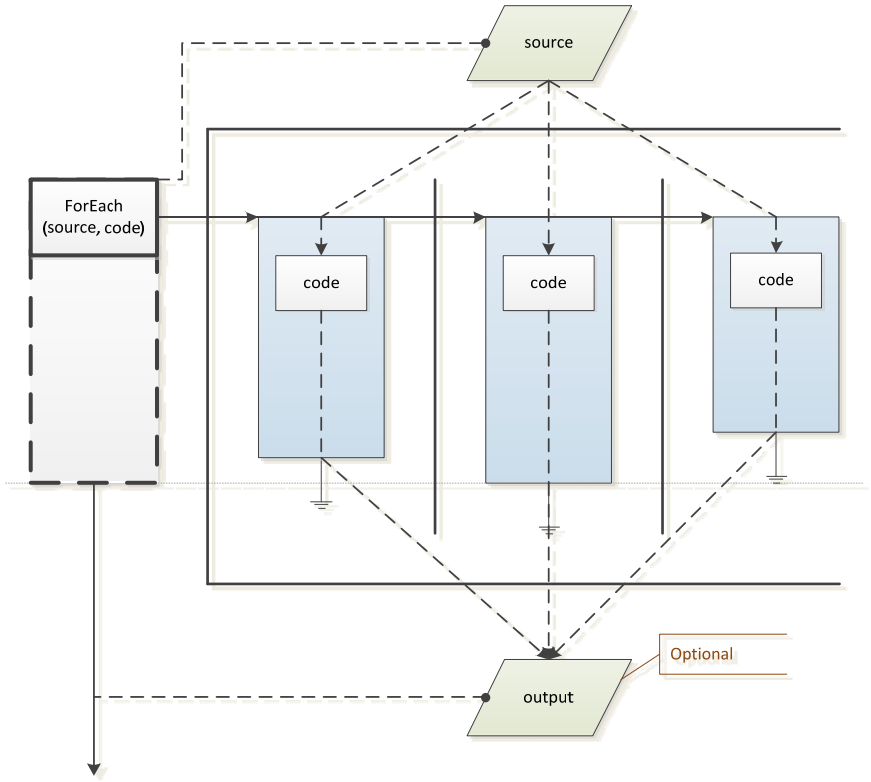
```
procedure TfrmMultithreadingMadeSimple.GenerateDataParallel(
  dataSize: int64; output: TStream);
const
  CBlockSize = 1 * 1024 * 1024 {1 MB};
var
  buffer: TOmniValue;
  memStr: TMemoryStream;
  outQueue: IOmniBlockingCollection;
  unwritten: IOmniCounter;
begin
  outQueue := TOmniBlockingCollection.Create;
  unwritten := CreateCounter(dataSize);
  Parallel.ParallelTask.NoWait
    .NumTasks(Environment.Process.Affinity.Count)
    .OnStop(Parallel.CompleteQueue(outQueue))
    .Execute(
      procedure
      var
        buffer      : TMemoryStream;
        bytesToWrite: integer;
        randomGen   : TGpRandom;
      begin
        randomGen := TGpRandom.Create;
        try
          while unwritten.Take(CBlockSize, bytesToWrite) do begin
            buffer := TMemoryStream.Create;
            buffer.Size := bytesToWrite;
            FillBuffer(buffer.Memory, bytesToWrite, randomGen);
            outQueue.Add(buffer);
          end;
        finally FreeAndNil(randomGen); end;
      end
    );
  for buffer in outQueue do begin
    memStr := buffer.AsObject as TMemoryStream;
    output.CopyFrom(memStr, 0);
    FreeAndNil(memStr);
  end;
end;
```

Glavni program med izvajanjem opravil ne čaka, temveč izvaja kodo, ki sledi klicu ParallelTask. V zanki (`for buffer in outQueue`) bere vrednosti iz *blocking collection*. Vsako vrednost (to je v bistvu blok psevdonaključnih števil) shrani v izhodni tok. Ta korak je potreben zato, ker TStream ni prirejen za večnitno delo in zato vanj ne smemo pisati iz več niti hkrati.

Izvaja se torej toliko niti, kolikor je jeder v procesorju in še glavna nit, ki podatke zapisuje v izhodni tok. Če podatkov v nekem trenutku ni na voljo, ukaz for..in počaka, da se podatki pojavijo. Šele ko se končajo vsa opravila, se sproži koda, določena s klicem OnStop in pokliče CompleteAdding, kar ustavi tudi zanko for..in.

ForEach

Abstrakcija *ForEach* predstavlja vzporedno zanko *for*. Vsa vzporedna opravila izvajajo isto kodo. Posebna pozornost je posvečena hitrosti in izkoriščenosti niti - da kateremu od opravil ne zmanjka dela prej kakor drugim. ForEach lahko izvedemo nad območjem celih števil ali pa nad drugimi strukturami, ki podpirajo enumeracijo `for..in` (na primer TStringList).



Spodnji primer generira praštevila od 1 do *CMaxPrime.* Praštevila shrani v (začasno) vrsto *primeQueue* in jih, ko se vsa opravila v vzporedni zanki končajo, iz te vrste prebere ter prvih 1000 izpiše na zaslon.

```
procedure TfrmMultithreadingMadeSimple.btnForEachUnorderedPrimesClick(
  Sender: TObject);
var
  prime: TOmniValue;
  primeQueue: IOmniBlockingCollection;
begin
  lbForEachPrimes.Clear;

  primeQueue := TOmniBlockingCollection.Create;
  Parallel.ForEach(1, CMaxPrime).Execute(
    procedure (const value: integer)
    begin
      if IsPrime(value) then begin
        primeQueue.Add(value);
      end;
    end);


  for prime in primeQueue do begin
    lbForEachPrimes.Items.Add(IntToStr(prime));
    if lbForEachPrimes.Items.Count = 1000 then
      break; //for
  end;
end;
```

S posebno obliko zanke lahko zagotovimo, da bodo izhodni podatki v istem vrstnem redu kakor vhodni. (Privzeto tega obnašanja ni, izhodni podatki so lahko pomešani, ker nihče ne zagotavlja, v kakšnem vrstnem redu bodo opravila jemala vrednosti iz vhodnega obsega.)

```
procedure TfrmMultithreadingMadeSimple.btnForEachOrderedPrimesClick(Sender:
    TObject);
var
  prime: TOmniValue;
  primeQueue: IOmniBlockingCollection;
begin
  lbForEachPrimes.Clear;

  primeQueue := TOmniBlockingCollection.Create;
  Parallel.ForEach(1, CMaxPrime)
    .PreserveOrder
    .Into(primeQueue)
    .Execute(
      procedure (const value: integer; var res: TOmniValue)
      begin
        if IsPrime(value) then
          res := value;
      end);

  for prime in primeQueue do begin
    lbForEachPrimes.Items.Add(IntToStr(prime));
    if lbForEachPrimes.Items.Count = 1000 then
      break; //for
  end;
end;
```

Pokažimo še, kako lahko z vzporedno zanko obdelamo podatke v vsebniku TStringList.

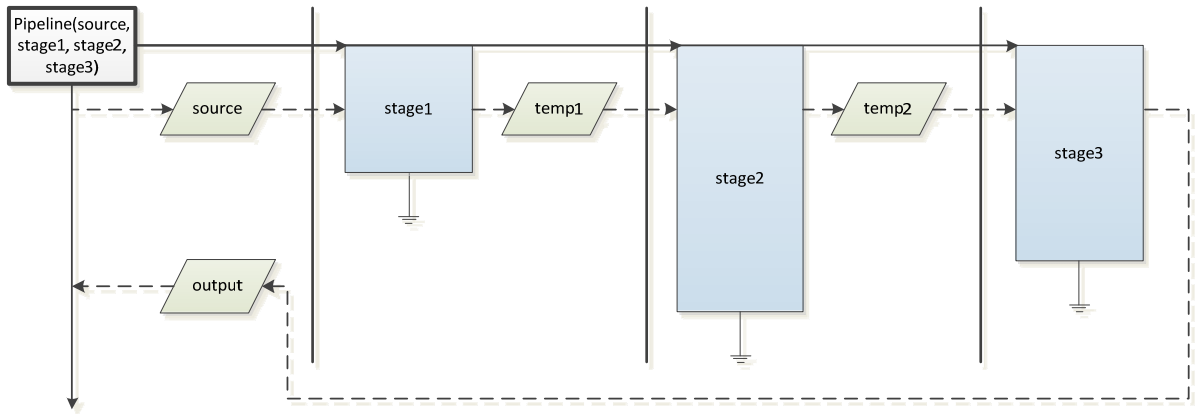
```
procedure TfrmMultithreadingMadeSimple.btnStringListForEachClick(
  Sender: TObject);
var
  maxLength: TOmniValue;
  sl: TStringList;
begin
  sl := TStringList.Create;
  try
    sl.LoadFromFile('..\..\AliceInWonderland.txt');

    maxLength := Parallel.ForEach<string>(sl)
      .Aggregate(0,
        procedure(var aggregate: TOmniValue; const value: TOmniValue)
        begin
          if value.AsInteger > aggregate.AsInteger then
            aggregate := value.AsInteger;
        end)
      .Execute(
        procedure(const value: string; var result: TOmniValue)
        begin
          result := Length(value);
        end);

  finally FreeAndNil(sl); end;
end;
```

Podprogram naloži vsebino datoteke v TStringList, nato pa ga vzporedno obdela s klicem Parallel.ForEach<string>. Za vsako vrednost (vrstico v datoteki), koda v Execute izračuna dolžino te vrstice.

Agregatorska funkcija (*Aggregate*) se pokliče za vsako vrednost, ki jo izračuna Execute, in izračuna najdaljšo vrstico, ki jo je »videlo« opravilo. Na koncu, ko so vsa opravila zaključila delo, pa se agregatorska funkcija pokliče še enkrat za vsako opravilo in izračuna najdaljšo vrstico v celem dokumentu.

Pipeline

Abstrakcija *Pipeline*, oziroma cevovod, je namenjena paraleliziranju opravil, ki se izvajajo v korakih. Namesto da bi hkrati izvajali vse korake, pošljemo podatek skozi cevovod, ki se izvaja v več nitih.



Prvi podatek se najprej obdela v prvi stopnji (stage). Nato se prenese v drugo stopnjo, ki ga začne obdelovati, prva pa lahko medtem že obdeluje drugi podatek. Ko se podatek prenese v tretjo stopnjo, začne druga obdelovati drugi podatek, prva pa že tretji podatek. In tako naprej …

Stopenj imamo lahko poljubno mnogo, določimo pa lahko tudi, da se na posamezni stopnji ista koda izvaja v več vzporednih opravilih. Podatki med stopnjami se prenašajo z vrstami tipa *blocking collection*.

Spodnja koda »zakodira« datoteko s primitivnim kodiranjem ROT13. Prva stopnja bere podatke iz datoteke, druga jih zakodira in tretja jih zapiše v novo datoteko.

```
procedure TfrmMultithreadingMadeSimple.btnPipelineRot13Click(
  Sender: TObject);
begin
  Parallel.Pipeline.Throttle(10240)
    .Stage(PipelineRead)
    .Stage(PipelineEncrypt)
    .Stage(PipelineWrite)
    .Run
    .WaitFor(INFINITE);
end;
```

Prva stopnja bere podatke iz datoteke vrstico po vrstico in vsako vrstico pošlje v nadaljnjo obdelavo.

```
procedure PipelineRead(const input, output: IOmniBlockingCollection);
var
  fIn: textfile;
  line: string;
begin
  Assign(fIn, '..\..\AliceInWonderland.txt');
  Reset(fIn);
  while not Eof(fIn) do begin
    Readln(fIn, line);
    output.Add(line);
  end;
  CloseFile(fIn);
end;
```

Druga stopnja sprejema vrstico po vrstico, vsako zakodira in jo pošlje v nadaljnjo obdelavo.

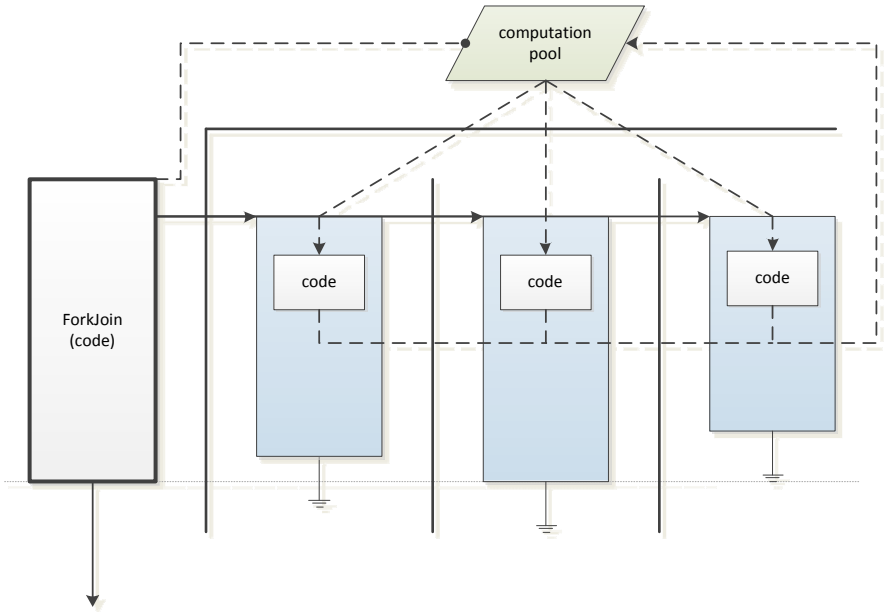
```
procedure PipelineEncrypt(const input: TOmniValue; var output: TOmniValue);
begin
  output := ROT13(input.AsString);
end;
```

Tretja stopnja bere podatke iz vhodne vrste in jih zapisuje v datoteko.

```
procedure PipelineWrite(const input, output: IOmniBlockingCollection);
var
  fOut: textfile;
  line: TOmniValue;
begin
  Assign(fOut, '..\..\AliceInWonderland-rot13.txt');
  Rewrite(fOut);
  for line in input do
    Writeln(fOut, line.AsString);
  CloseFile(fOut);
end;
```

ForkJoin

Abstrakcija *ForkJoin* je namenjena reševanju problemov tipa »deli in vladaj«, kjer problem razdelimo na podprobleme, te na nove podprobleme in tako dalje, dokler podproblemi ne postanejo tako enostavni, da jih lahko rešimo. Rezultati podproblemov nato potujejo »navzgor« in se uporabijo za izračun novih rezultatov, dokler na koncu ne dobimo rešitve celotnega problema. Pri tej abstrakciji vsa opravila izvajajo isto kodo, ki rešuje (pod)probleme.



Primer podprograma, ki z rabo abstrakcije Fork/Join uredi polje. Ko je podproblem dovolj majhen, ga uredi z vstavljanjem (klic metode InsertionSort), sicer pa po metodi QuickSort določi *particijo*, tam razdeli polje in ga loči na dva podproblema, ki ju pošlje v obdelavo.

```
procedure TParallelSorter.QuickSort(left, right: integer);
var
  pivotIndex: integer;
  sortLeft  : IOmniCompute;
  sortRight : IOmniCompute;
begin
  if right > left then begin
    if (right - left) <= CSortThreshold then
      InsertionSort(left, right)
    else begin
      pivotIndex := Partition(left, right, (left + right) div 2);
      sortLeft := FForkJoin.Compute(
        procedure
        begin
          QuickSort(left, pivotIndex - 1);
        end);
      sortRight := FForkJoin.Compute(
        procedure
        begin
          QuickSort(pivotIndex + 1, right);
        end);
      sortLeft.Await; sortRight.Await;
    end;
  end;
end;
```

BackgroundWorker

Abstrakcija *BackgroundWorker* je namenjena izdelavi opravila, ki v ozadju čaka na delo. Ko dobi podatke, jih obdela, rezultat vrne glavni niti in čaka na naslednje podatke. Podatke lahko obdeluje eno ali več hkrati pognanih opravil.

BackgroundWorker izdelamo s klicem istoimenske metode, ki ji podamo vsaj kodo, ki obdeluje podatke (*Execute*) in kodo, ki se sproži ob zaključku obdelave podatkov (*OnRequestDone*). Spodnji primer ob tem še omeji število sočasnih opravil na 2. (Privzeto se izvaja samo eno opravilo.)

```
FBackgroundWorker := Parallel.BackgroundWorker.NumTasks(2)
  .Execute(
    procedure (const workItem: IOmniWorkItem)
    begin
      workItem.Result := workItem.Data.AsInteger * 3;
      Sleep(500);
    end
  )
  .OnRequestDone(
    procedure (const Sender: IOmniBackgroundWorker;
      const workItem: IOmniWorkItem)
    begin
      lbLogBW.ItemIndex := lbLogBW.Items.Add(Format('%d * 3 = %d',
        [workItem.Data.AsInteger, workItem.Result.AsInteger]));
    end
  );
```

Podatke v obdelavo pošljemo s klicem metode Schedule.

```
procedure TfrmMultithreadingMadeSimple.btnSendWorkClick(Sender: TObject);
begin
  FBackgroundWorker.Schedule(
    FBackgroundWorker.CreateWorkItem(Random(100)));
end;
```

Za vse ostalo poskrbi OmniThreadLibrary.

Povezave

»Multithreading – The Delphi Way« – klasika Martina Harveya, ki začetnika povede v temačne vode večnitnega programiranja v Delphiju.

`http://cc.embarcadero.com/item/14809`

»Parallel Programming with OmniThreadLibrary« – knjiga (v delu) o večnitnem programiranju s knjižnico OmniThreadLibrary.

`http://leanpub.com/omnithreadlibrary`

Spletna dokumentacije knjižnice OmniThreadLibrary.

`http://otl.17slon.com/book`

Serija šestih člankov iz revije Blaise Pascal Magazine:

»Introduction to Multithreading«

`http://www.thedelphigeek.com/2012/02/blaise-pascal-magazine-rerun-4.html`

»Four Ways to Create a Thread«

`http://www.thedelphigeek.com/2012/02/blaise-pascal-magazine-rerun-5-four.html`

»Synchronisation in Multithreaded Environment«

`http://www.thedelphigeek.com/2012/02/blaise-pascal-magazine-rerun-6.html`

»Intraprocess Communication«

`http://www.thedelphigeek.com/2012/02/blaise-pascal-magazine-rerun-7.html`

»Debugging Multithreaded Applications«

`http://www.thedelphigeek.com/2012/02/blaise-pascal-magazine-rerun-8.html`

»High Level Multithreading«

`http://www.thedelphigeek.com/2012/02/blaise-pascal-magazine-rerun-9-high.html`

Prispevki z bloga The Delphi Geek:

Async/Await

http://www.thedelphigeek.com/2012/07/asyncawait-in-delphi.html

Async

http://www.thedelphigeek.com/2011/04/simple-background-tasks-with.html
http://www.thedelphigeek.com/2011/07/life-after-21-async-redux.html

Future

http://www.thedelphigeek.com/2010/06/omnithreadlibrary-20-sneak-preview-2.html
http://www.thedelphigeek.com/2010/06/future-of-delphi.html
http://www.thedelphigeek.com/2011/07/life-after-21-exceptions-in.html

Join

http://www.thedelphigeek.com/2011/07/life-after-21-paralleljoins-new-clothes.html

ParallelTask

http://www.thedelphigeek.com/2011/09/life-after-21-parallel-data-production.html

ForEach

http://www.thedelphigeek.com/2010/06/omnithreadlibrary-20-sneak-preview-1.html

Pipeline

http://www.thedelphigeek.com/2010/11/multistage-processes-with.html
http://www.thedelphigeek.com/2011/09/life-after-21-pimp-my-pipeline.html

ForkJoin

http://www.thedelphigeek.com/2011/05/divide-and-conquer-in-parallel.html

BackgroundWorker

http://www.thedelphigeek.com/2012/01/background-worker.html

TaskConfig

http://www.thedelphigeek.com/2011/04/configuring-background-otlparallel.html

Drugi članki

http://otl.17slon.com/tutorials.htm