Kurs języka Object/Delphi Pascal

na bazie implementacji Free Pascal.

AUTOR ŁUKASZ STAFINIAK

Email: lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
Web: www.ii.uni.wroc.pl/~lukstafi

Jeśli zauważysz błędy na slajdach, proszę daj znać!

Wykład 9: Wątki

Typy proceduralne. Wielowątkowość.

Typy proceduralne

- Definicja typu proceduralnego to nagłówek procedury/funkcji bez jej nazwy: `type TOneArg = procedure (const X: Integer);`
- Dla wskaźników na metody, dodatkowo dodajemy `of object`:
  `type TOneArgM = procedure (const X: Integer) of object;`
- Pobieramy wartość typu proceduralnego (w tym wskaźnika na metodę) przez `@`: odpowiednio `@FunctionName` lub `@ObjectVar.MethodName`
- Wartością zmiennej typu proceduralnego może też być `nil`.
- Do zwykłych zmiennych typu proceduralnego możemy przypisać tylko procedury / funkcje globalne; ale po włączeniu dyrektywą `{$modeswitch nestedprocvars}`, można deklarować zmienne `is nested`.
  - **Możliwy błędny kod** gdy przekażemy wartość typu proceduralnego `is nested` poza zakres funkcji zawierającej procedurę zagnieżdżoną.

```
program test_procvars;
{$modeswitch nestedprocvars}
type
   TOneArg = procedure (const X: Integer);
   TOneArgL = procedure (const X: Integer) is nested;
var
   F: TOneArg;
   FL: TOneArgL;
   procedure Global (const X: Integer);
      procedure Local (const Y: Integer);
      begin
         WriteLn ('Local ',Y, ' of global ',X);
      end;
   begin
      FL := @Local;
      WriteLn ('Global ',X);
      FL(7);
   end;
begin
   F := @Global;
   F(5);
   FL(13);
end.
```

Włącza modyfikator `is nested`.

Nie można `Local` przypisać do `F`.

Zmienną/argument typu proceduralnego wywołujemy jak zwykłą procedurę czy funkcję.

Błędny kod!

Kompilator nie tworzy domknięcia funkcyjnego zawierającego `X=5`.

Programowanie wielowątkowe

- Wątki systemowe (*threads*) pozwalają na:

  zrównoleglenie. wykorzystanie do realizacji zadania wielu procesorów / rdzeni równocześnie; jak i

  współbieżność. przeplatanie się obliczeń tak że jedne nie blokują innych.

- Cechą charakterystyczną realizacji współbieżności / zrównoleglenia przez wielowątkowość (wątki systemowe) jest **współdzielona (*shared*) pamięć**.

- Współbieżność stosuje się w programowaniu interfejsów użytkownika, symulacjach, innych systemach interaktywnych/reaktywnych, programowaniu wejścia-wyjścia czy web-serwerów.

- Zrównoleglenie dla przyspieszenia programów jest coraz ważniejsze.

  - Teraz wszystkie komputery posiadają co najmniej dwa procesory / rdzenie, ale postęp nie jest tak dramatyczny jak spodziewano się 10-20 lat temu.

Wątki we Free Pascalu: podejście `BeginThread`

- Żeby korzystać z wątków, pod Linuxem potrzebujemy użyć modułu `cthreads` korzystającego z odpowiedniej biblioteki systemowej, pod Windowsem wątki są dostępne w samym Free Pascalu.

```
uses {$ifdef unix} cthreads, {$endif} SysUtils, Classes;
```

- Podstawowy mechanizm to funkcja `BeginThread`:

```
TThreadFunc = function(parameter : pointer) : ptrint;

function BeginThread(ThreadFunction : TThreadFunc;
                     p : pointer) : TThreadID;
```

- `ThreadFunction` będzie wywołana w nowo utworzonym wątku z argumentem `p` (który jest opcjonalny – przy braku przekazane `nil`).
- Wątek działa aż do zakończenia `ThreadFunction`, albo wywołania `EndThread` z wewnątrz wątku lub `KillThread` na wątku.
- Funkcje zarządzające wątkami (po nazwach funkcji widać co robią):

```
procedure EndThread(ExitCode : DWord);
```
Zakończenie wątku od wewnątrz – chyba równoważne z `Exit`.

```
function SuspendThread(threadHandle: TThreadID): dword;
function ResumeThread(threadHandle: TThreadID): dword;
function KillThread(threadHandle: TThreadID): dword;
```
Zakończenie wątku z zewnątrz.

```
function WaitForThreadTerminate(threadHandle: TThreadID;
                                TimeoutMs : longint): dword;
```
Czeka na zakończenie wątku, ale nie dłużej niż `TimeoutMs` milisekund. (Nie zabija wątku! Tylko przerywa czekanie.)

```
function ThreadSetPriority(threadHandle: TThreadID;
                           Prio: longint): boolean;
function ThreadGetPriority(threadHandle: TThreadID):
Integer;
function GetCurrentThreadId: dword;
procedure ThreadSwitch;
```
Sugeruje systemowi przejście do obliczania innego wątku (jeśli brakuje procesorów).

Wątki we Free Pascalu: podejście `TThread`

- Moduł `Classes` definiuje klasę `TThread` opakowującą obsługę wątków. Dziedziczymy z tej klasy żeby zaimplementować nową klasę wątków.
- Konstruktor `TThread` bieże argument `Suspended`: jeśli przekażemy `False`, wątek wystartuje od razu po utworzeniu obiektu `TThread`, w p.p. dopiero po wywołaniu metody `Resume`.
- Zastępujemy metodę `Execute` żeby zaimplementować kod wątku.
- Klasa `TThread` ma właściwość `Terminated`: w kodzie wątku powinniśmy na bieżąco ją sprawdzać i jeśli okaże się `True`, zakończyć wątek przy najbliższej okazji. Jest ona ustawiana przez metodę `Terminate`.
- Kolejna właściwość to `FreeOnTerminate`: jeśli ustawimy ją na `True`, to obiekt wątku zostanie automatycznie zwolniony po zakończeniu wątku.

Wątki w Lazarusie

- Dobrze jest uniknąć wątków systemowych wykorzystując **cooperative multithreading**. Wątki systemowe są konieczne gdy potrzebujemy:
  - blokujących operacji wejścia-wyjścia,
  - wykorzystania wielu procesorów / rdzeni,
  - innych wywołań jednolitych, czasochłonnych funkcji zewnętrznych.
- W Lazarusie *cooperative multithreading* realizuje pętla obsługi zdarzeń. Gdy zaimplementujemy kilka procesów (nie w sensie technicznym, tzn. nie systemowych) jako ciągi akcji, tak że poprzednia akcja zamiast wywoływać kolejną jako procedurę, wrzuca kolejną akcję jako zdarzenie do pętli obsługi zdarzeń, te procesy będą przebiegać współbieżnie.
- Takie programowanie jest jednak niewygodne. Na szczęście, Lazarus ma procedurę `Application.ProcessMesages`. Wywołujemy ją co jakiś czas w czasochłonnym zadaniu-procedurze, żeby obsłużyć zaległe zdarzenia.
  - Patrz `pascal-progs8/multithreading/singlethreadingexample1.lpi`.

- Wszystkie modyfikacje GUI muszą być wywoływane z **wątku głównego** Lazarusa (ten w którym działa pętla obsługi zdarzeń); w tym celu klasa TThread dysponuje metodą Synchronize. Jej wywołanie usypia aktualny wątek i uruchamia przekazaną metodę w wątku głównym.

```
procedure TMyThread.ShowStatus;
begin
  Form1.Caption := fStatusText;
end;
procedure TMyThread.Execute;
var newStatus : string;
begin
  fStatusText := 'TMyThread Starting...';
  Synchronize(@Showstatus);
  fStatusText := 'TMyThread Running...';
  while (not Terminated) and (true {any condition required}) do begin
    newStatus:='TMyThread Time: '+FormatDateTime('YYYY-MM-DD HH:NN:SS',Now);
    if NewStatus <> fStatusText then begin
      fStatusText := newStatus;
      Synchronize(@Showstatus);
    end;
  end;
end;
```

Sekcje krytyczne

- Sesje krytyczne umożliwiają implementację mechanizmu wykluczania (przy użyciu *locks*/*mutexes*), głównej techniki programowania współbieżnego (choć uważanej za niskopoziomową).

```
procedure InitCriticalSection(var cs: TRTLCriticalSection);
```
Stwórz mutex.

```
procedure DoneCriticalSection(var cs: TRTLCriticalSection);
```
Zniszcz mutex.

```
procedure EnterCriticalSection(var cs: TRTLCriticalSection);
```
Czekaj aż mutex będzie otwarty, i zamknij go.

```
procedure LeaveCriticalSection(var cs: TRTLCriticalSection);
```
Otwórz zamek.

- Mutex odpowiada zasobowi, z którego może korzystać tylko jeden wątek naraz.
- Mutex jest cennym zasobem: żeby zapobiec zablokowaniu, otwieraj go w klauzuli `finally`.

- Procedury `InterlockedIncrement` / `InterlockedDecrement` są atomicznymi (a właściwie *thread-safe*) wariantami `Inc` i `Dec`: tylko jeden wątek ma dostęp do zmiennej w czasie działania procedury.
  - Inne procedury *thread-safe* manipulacji zmiennymi / polami to: `InterLockedExchange`, `InterLockedExchangeAdd`, `InterlockedCompareExchange`.
  - Operacja jest atomiczna jeśli nie może być przepleciona z żadnymi innymi operacjami; jest *thread-safe* jeśli nie może być przepleciona z innymi operacjami w sposób zaburzający jej semantykę (tzn. rezultat operacji nie spełniający specyfikacji).

Patrz przykład `pascal-progs8/multithreading/criticalsectionexample1.lpi`.

```
unit CriticalSectionUnit1;
[...]
procedure TMyThread.Execute;
var
  i,j: Integer;
  CurCounter: LongInt;
```

```
begin
  FAFinished:=false;
  for i:=1 to 100000 do begin
    if Form1.UseCriticalSection then
      EnterCriticalSection(Form1.CriticalSection);
    try
      CurCounter:=Form1.Counter;
      for j:=1 to 1000 do ;
      inc(CurCounter);
      Form1.Counter:=CurCounter;
    finally
      if Form1.UseCriticalSection then
        LeaveCriticalSection(Form1.CriticalSection);
    end;
  end;
  FAFinished:=true;
end;
```

Czekanie (w uśpieniu) na zdarzenie

- Podstawowym obok sekcji krytycznych mechanizmem synchronizacji jest sygnalizowanie dostępności.
- Jeśli kilka wątków ma dostęp do tego samego „zdarzenia", to jeden z nich może przerwać czekanie pozostałych.
- Czekanie jest *idle*, tzn. nie marnuje obliczeń.

```
function RTLEventCreate: PRTLEvent;
procedure RTLeventdestroy(state: PRTLEvent);
```
Konstruktor i destruktor zdarzenia.

```
procedure RTLeventSetEvent(state: PRTLEvent);
```
"Ustawia" zdarzenie i wznawia wątki czekające na to zdarzenie.

```
procedure RTLeventResetEvent(state: PRTLEvent);
```
Resetuje zdarzenie tak że `RTLeventWaitFor` będzie zawieszać wątki.

```
procedure RTLeventWaitFor(state: PRTLEvent);
procedure RTLeventWaitFor(state:PRTLEvent; timeout:LongInt);
```
Czeka (bezterminowo lub przez `timeout` milisekund) na zdarzenie, zawieszając wątek chyba że zdarzenie jest już ustawione.

Zmienne lokalne wątków: `ThreadLocal`

- Zazwyczaj zmienne widoczne spoza zakresu procedury implementującej wątek (m.in. zmienne globalne) są wspólne z innymi wątkami je widzącymi.
  - Oczywiście zmienne lokalne procedury implementującej wątek przynależą tylko do tego wątku.
- Można to zmienić deklarując zmienną jako `ThreadVar` zamiast `var`: wtedy dla każdego wątku utworzona będzie inna zmienna o tej nazwie.
  - Kopia zmiennej dla wątku **nie** ma wartości tej zmiennej (w wątku głównym czy jakimś innym) z chwili tworzenia nowego wątku.
- Unikaj zmiennych `ThreadVar`, są wolniejsze od zwykłych zmiennych.

Przykład: menedżer prac

- Wątki systemowe są stosunkowo kosztowne, więc często korzysta się z menedżerów prac obsługujących wiele prac, kolejno, na tym samym wątku systemowym.
- Biblioteka standardowa Free Pascala nie dostarcza menedżera prac, więc zaprogramujemy własny, w dwóch wariantach.
- Podstawowa wersja będzie uruchamiać metody jako prace.
- W rozszerzonej, prace będą obiektami, główna część pracy będzie wykonywana w wątku roboczym, ale końcowa część pracy w wątku menedżera, który będzie też odpowiedzialny za zwalnianie obiektów-prac.
- Patrz paczka z programami do wykładu.

```
unit workers_synch;
...

type
   TJob = class { Could be an interface }
      procedure Execute; virtual; abstract;
      procedure Report; virtual; abstract;
   end;
   TJobQueue = specialize TFPGList<TJob>;
   TWorker = class (TThread)
   private
      Manager: TWorkManager;
   protected
      procedure Execute; override;
   public
      constructor Create (aManager: TWorkManager);
   end;
   TWorkerList = specialize TFPGList<TWorker>;
```

```
TWorkManager = class
private
   ManagerLock: TRTLCriticalSection;
   JobQueue: TJobQueue;
   ReportQueue: TJobQueue;
   WorkerList: TWorkerList;
   FJobs: Integer;
   GotJob: PRTLEvent;
   GotReport: PRTLEvent;
   function TakeJob (OldJob: TJob) : TJob;
public
   constructor Create (NumWorkers: Integer);
   procedure AddJob (Job: TJob);
   procedure FinishJobs;
   destructor Destroy; override;
   property Jobs: Integer read FJobs;
   { procedure Suspend;
     procedure Resume; etc...}
end;
```

Raporty też mają swoją kolejkę (nie są uruchamiane od razu po zakończeniu pracy).

W tej procedurze są uruchamiane raporty.

```
constructor TWorker.Create (aManager: TWorkManager);
begin
   Manager := aManager;
   FreeOnTerminate := True;                    Zwolni obiekt po zakończeniu wątku.
   inherited Create (False);                                Startuje wątek teraz.
end;
procedure TWorker.Execute;
var CurrentJob : TJob;
begin
   repeat                                                        Pętla główna wątku.
      CurrentJob := Manager.TakeJob (nil);                  Pobierz pierwszą pracę.
      while CurrentJob <> nil do
      begin
         RTLEventResetEvent (Manager.GotJob);
         CurrentJob.Execute;              Przekazuje menedżerowi pracę, żeby włożył
         CurrentJob := Manager.TakeJob (CurrentJob);          raport do kolejki.
      end;
      RtlEventWaitFor (Manager.GotJob, WORKER_IDLETIME);       Czekaj na pracę
   until Terminated;                          z timeoutem, bo możliwe że kilka prac
end;                          zarejestrowanych pod rząd powoduje „wyciek" zdarzenia.
```

```
procedure TWorkManager.AddJob (Job: TJob);
begin
   EnterCriticalSection (ManagerLock);
   try JobQueue.Add (Job); Inc (FJobs);
   finally LeaveCriticalSection (ManagerLock) end;
   RtlEventSetEvent (GotJob);
end;
function TWorkManager.TakeJob (OldJob: TJob) : TJob;
begin
   EnterCriticalSection (ManagerLock);
   try
      if OldJob <> nil then
      begin
         ReportQueue.Add (OldJob);
         Dec (FJobs);
         RtlEventSetEvent (GotReport);
      end;
      Result := Pop (JobQueue);
   finally
      LeaveCriticalSection (ManagerLock);
   end;
end;
```

Współdzielone struktury danych muszą być obsługiwane z mutexem.

Oszczędzamy na sekcjach krytycznych obsługując obydwie kolejki (`JobQueue` i `ReportQueue`) jednym mutexem.

```
procedure TWorkManager.FinishJobs;
var Job: TJob;
begin
   while (Jobs > 0) or (ReportQueue.Count > 0) do
   begin                          Czekaj na wykonanie wszystkich zarejestrowanych prac
      EnterCriticalSection (ManagerLock);
      try Job := Pop (ReportQueue);
      finally LeaveCriticalSection (ManagerLock) end;
      if Job <> nil
      then begin
         Job.Report;                                          i dokończ ich obsługę.
         Job.Destroy;
      end else RtlEventWaitFor (GotReport, MANAGER_IDLETIME);
   end;
end;
```

```
destructor TWorkManager.Destroy;
var Worker: TWorker;
begin                                   Nie pozwalamy zniszczyć jeśli są zaległe prace.
   if (Jobs > 0) or (ReportQueue.Count > 0) then
      raise (Exception.Create
                ('TWorkManager.Destroy: unfinished jobs or reports'));
   for Worker in WorkerList do Worker.Terminate;          Grzecznie czekamy
   for Worker in WorkerList do Worker.WaitFor;          aż pracownicy się zniszczą.
   DoneCriticalSection (ManagerLock);       Zwalniamy zasoby: mutexa i zdarzenia.
   RtlEventDestroy (GotJob);
   RtlEventDestroy (GotReport);
   JobQueue.Destroy;
   ReportQueue.Destroy;
   WorkerList.Destroy;
end;
```

```
program raytracer_workers; [...]
type
   TRayJob = class (TJob)
      Line: array[0..WIDTH-1] of TColor;
      Y: Integer;
      constructor Create (aY: Integer);
      procedure Execute; override;
      procedure Report; override;
   end;
   constructor TRayJob.Create (aY: Integer);
   begin Y := aY end;
   procedure TRayJob.Execute;
   var X: Integer;
   begin                                                    Zapamiętaj wynik pracy.
      for X:=0 to WIDTH-1 do Line[X] := RayTrace(LightDir, Scene^, X, Y);
   end;
   procedure TRayJob.Report;
   var X: Integer;
   begin                    Wyświetlanie musi być z wątku głównego, więc jest w raporcie.
      for X := 0 to WIDTH-1 do PutPixel (X, HEIGHT-Y-1, Line[X]);
      SDL_Flip (screen);
   end;
```