Asynchronní výpočty

Nejdříve motivace: asynchronní načtení daných souborů:
```
let read_all files =
  let read_one (file:string) = async {
    use reader = new StreamReader(file)
    let! content = reader.AsyncReadToEnd()
    return content
  }
  files |> Seq.map read_one
        |> Async.Parallel
        |> Async.RunSynchronously
```

Rekurzivní stahování stránek:
```
let rec download (url:string) = async {
  try
    let! response = WebRequest.Create(url).AsyncGetResponse()
    use reader = new StreamReader(response.GetResponseStream())
    let! content = reader.AsyncReadToEnd()
    store url content
    content |> extract_references
            |> Seq.filter unseen_references
            |> Seq.iter (fun ref -> Async.Start(download ref))
  with
    _ -> ()
```

Vícevláknové zpracování

Pokud chcete vytvořit explicitní vlákno na nějaký úkol, použijte `BackgroundWorker`
```
System.ComponentModel.BackgroundWorker
  RunWokerAsync : [ unit | obj ] -> unit  zvenku
  CancelAsync : unit -> unit              zvenku
  CancellationPending : unit -> bool      zvenku + zevnitř
  ReportProgress : int [ -> obj ] -> unit zevnitř
  events OnDoWork, OnProgressChanges, OnRunWorkerCompleted
DoWorkEventArgs má Argument, Cancel, Result
```

`System.Thread.ThreadPool` je systémem používaný ThreadPool. Můžete ho používat také, pomocí
```
ThreadPool.QueueUserWorkItem : (obj -> unit) -> bool
ThreadPool.QueueUserWorkItem : (obj -> unit * obj) -> bool
ThreadPool.RegisterWaitForSingleEvent : ...
ThreadPool.Get{Available,Min,Max}Threads : unit -> int * int
```
Vrací pracující vlákna a vlákna pro i/o completion porty
Výchozí hodnoty na Mono 2.6.7: 40+20, Mono 2.8.1: 200+8, .NET4 250+1000
```
ThreadPool.Set{Min,Max}Threads : int * int -> bool
```

Asynchronní metody v .NETu

Mnoho metod, jako třeba `Invoke` nebo `Read`, mají i `Begin-`/`End-` variantu:
- `BeginRead : byte[]*int*in *(callback:AsyncCallback)*(st:Object)->IAsyncResult` inicializuje začátek asynchronního volání, musí se potom ukončit pomocí
- `EndRead : IAsyncResult -> int`

Uživatel má několik možností, jak poznat, že volání je hotovo:
- neřeší to a zavolá `EndRead`, které se zablokuje, dokud se operace nedokončí
- pomocí `IAsyncResult.IsCompleted`
- pomocí delegátu, který předá `Begin-` operaci, i s parametrem, který se předá jako `IAsyncResult.AsyncState`

Callback při dokončení operace se volá z jiného vlákna (z nějakého vlákna threadpoolu).

Async<'t>

`Async<'t>` je asynchronní operace, která vrátí výsledek typu `'t`.

Vytvářet je můžeme pomocí `async` computation expression nebo pomocí:
```
static member AwaitEvent : IEvent<'Del,'T> * ?(unit -> unit) -> Async<'T>
static member AwaitIAsyncResult : IAsyncResult * ?int -> Async<bool>
static member AwaitTask : Task<'T> -> Async<'T>
static member AwaitWaitHandle : WaitHandle * ?int -> Async<bool>
static member Catch : Async<'T> -> Async<Choice<'T,exn>>
static member FromBeginEnd : (AsyncCallback * obj -> IAsyncResult) * (IAsyncResu
```

```
lt -> 'T) * ?(unit -> unit) -> Async<'T>
static member FromContinuations : (('T -> unit) * (exn -> unit) * (OperationCanc
eledException -> unit) -> unit) -> Async<'T>
static member Ignore : Async<'T> -> Async<unit>
static member Parallel : seq<Async<'T>> -> Async<'T []>
static member Sleep : int -> Async<unit>
```

Spustit je můžeme pomocí

```
static member RunSynchronously : Async<'T> * ?int * ?CancellationToken -> 'T
static member Start : Async<unit> * ?CancellationToken -> unit
static member StartAsTask : Async<'T> * ?TaskCreationOptions * ?CancellationToke
n -> Task<'T>
static member StartChild : Async<'T> * ?int -> Async<Async<'T>>
static member StartChildAsTask : Async<'T>*?TaskCreationOptions->Async<Task<'T>>
static member StartImmediate : Async<unit> * ?CancellationToken -> unit
static member StartWithContinuations : Async<'T> * ('T -> unit) * (exn -> unit)
* (OperationCanceledException -> unit) * ?CancellationToken -> unit
```

Speciální metody:

```
static member SwitchToContext : SynchronizationContext -> Async<unit>
static member SwitchToNewThread : unit -> Async<unit>
static member SwitchToThreadPool : unit -> Async<unit>
```

Zpracovávat UI jde jenom z vlákna dialogu. Kvůli tomu máme `SynchronizationContext`
V GUI vlákně si uložíme `System.Threading.SynchronizationContext()` a poté se do něj můžeme přepnout pomocí `SwitchToContext`

```
let async1 (button : Button) = async {
    button.Text <- "Busy"; button.Enabled <- false
    let context = System.Threading.SynchronizationContext.Current
    do! Async.SwitchToThreadPool()
    use outputFile = System.IO.File.Create("longoutput.dat")
    do! outputFile.AsyncWrite(bufferData)
    do! Async.SwitchToContext(context)
    button.Text <- "Start"; button.Enabled <- true
  }

static member AsBeginEnd : ('Arg -> Async<'T>) -> ('Arg * AsyncCallback * obj ->
 IAsyncResult) * (IAsyncResult -> 'T) * (IAsyncResult -> unit)

static member OnCancel : (unit -> unit) -> Async<IDisposable>
static member TryCancelled : Async<'T> * (OperationCanceledException -> unit) ->
 Async<'T>
static member CancellationToken :  Async<CancellationToken>
CancellationTokenSource.Cancel()
static member DefaultCancellationToken :  CancellationToken
static member CancelDefaultToken : unit -> unit

type Stream with Async{Read,Write} : byte [] * ?int * ?int -> Async<int>
                 AsyncRead : int -> Async<byte []>
type WebRequest with AsyncGetResponse() : Async<WebResponse>
type WebClient with AsyncDownloadstring : Uri -> Async<string>
```

♣ V .NETu 4.0 existuje podobná funkcionalita v `System.Threading.Tasks`.
`Task` a `Task<'Res>` jsou jako `Async<'t>` a dají se spojovat dohromady.
Vytváří se buď pomocí konstruktoru nebo pomocí `TaskFactory` a `TaskFactory<'Res>`, která umí např.
• `ContinueWhenAll` • `ContinueWhenAny` • `FromAsync`
Způsob rozdělování a spouštění `Tasků` implementuje `TaskScheduler`.

Computation expressions

```
expr { let ... }                          expr { let! ... }
expr { do ... }                           expr { do!  ... }
expr { yield ... }                        expr { yield! ... }
expr { return ... }                       expr { return! ... }
expr { use ... }                          expr { use! ... }

builder-expr { cexpr } =
  let b = builder-expr in b.Run (b.Delay(fun () -> {| cexpr |}))
```

Pokud `Run` nebo `Delay` neexistují, nezavolají se.

Přepisovací pravidla

```
{| let binds in cexpr |}                  = let binds in {| cexpr |}
{| let! pat = expr in cexpr |}            = b.Bind(expr, (fun pat -> {| cexpr |}))
{| do expr in cexpr |}                    = expr; {| cexpr |}
{| do! expr in cexpr |}                   = b.Bind(expr, (fun () -> {| cexpr |}))
{| yield expr |}                          = b.Yield(expr)
{| yield! expr |}                         = b.YieldFrom(expr)
{| return expr |}                         = b.Return(expr)
{| return! expr |}                        = b.ReturnFrom(expr)
{| use pat = expr in cexpr |}             = b.Using(expr, (fun pat -> {| cexpr |}))
{| use! v = expr in cexpr |}              = b.Bind(expr, (fun v ->
                                                 b.Using(v,(fun v -> {| cexpr |})))
{| if expr then cexpr0 |}                 = if expr then {| cexpr0 |} else b.Zero()
{| if expr then cexpr0 else cexpr1 |}     = if expr then {|cexpr0|} else {|cexpr1|}
{| match expr with p_i -> cexpr_i |}      = match expr with p_i -> {| cexpr_i |}
{| for pat in expr do cexpr |}            = b.For({|expr|}, (fun pat -> {|cexpr|}))
{| while expr do cexpr |}                 = b.While((fun () -> expr), {|cexpr|}Del)
{| try cexpr with p_i -> cexpr_i |}       = b.TryWith({| cexpr |}Del, (fun v ->
                                            match v with | (p_i:exn) -> {|cexpr_i|}
                                                         | _ -> raise exn)
{| try cexpr finally expr |}    = b.TryFinally( {| cexpr |}Del, (fun () -> expr))
{| cexpr0; cexpr1 |}            = b.Combine({| cexpr0 |}, {| cexpr1 |}Del)
{| other-expr0 ; cexpr1 |}      = other-expr; {| cexpr1 |}
{| other-expr |}                = other-expr; b.Zero()
```

kde `{| cexpr |}Del` je `b.Delay(fun () -> {| cexpr |})`