

Низкоуровневые потоки, события

Юрий Литвинов

17.04.2018г

Доклады

Минус одна домашка

- ▶ Дополнительные возможности F# (единицы измерения, lazy, active patterns)
- ▶ F# и анализ данных
- ▶ WebSharper (обзор и небольшая демонстрация)
- ▶ Type Providers, F# Data
- ▶ FAKE, Scaffold

Управление потоками “вручную”

- ▶ Сложнее и опаснее, чем `async`
- ▶ Позволяет управлять приоритетом потока, точнее управлять временем жизни и поведением потока, использовать низкоуровневые механизмы синхронизации
- ▶ Абстракция отдельного ядра, а не асинхронного вычисления

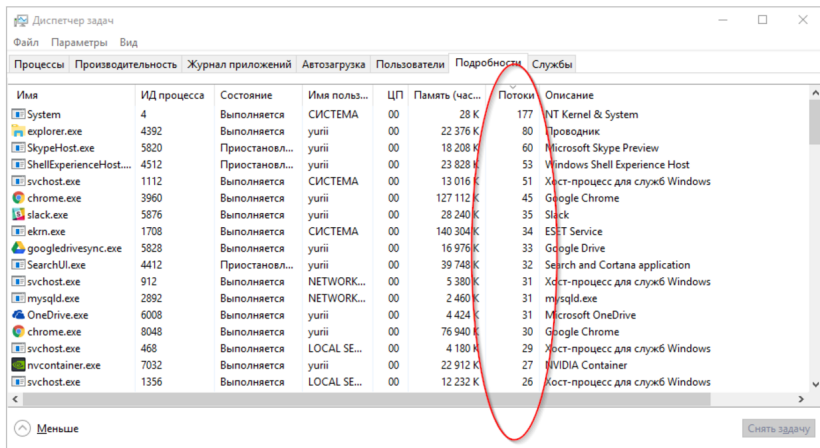
Поток в Windows

- ▶ Thread Kernel Object (1240 байт)
- ▶ Thread environment block (TEB) (4 Кб)
- ▶ User-mode stack (1 Мб)
- ▶ Kernel-mode stack (24 Кб)

Ещё для каждой dll-ки, загруженной для процесса при старте или остановке потока вызывается DllMain с параметрами `DLL_THREAD_ATTACH` и `DLL_THREAD_DETACH`

Квант времени — 20-30 мс, после чего происходит *переключение контекстов*

Как делать не надо



Класс System.Threading.Thread

open System.Threading

```
let t = new Thread(ThreadStart(fun _ ->
    printfn "Thread %d: Hello"
        Thread.CurrentThread.ManagedThreadId))
```

```
t.Start()
printfn "Thread %d: Waiting!"
    Thread.CurrentThread.ManagedThreadId
```

```
t.Join()
printfn "Done!"
```

Планировщик

- ▶ Раз в квант времени (или чаще) выбирает поток для исполнения
 - ▶ Рассматриваются только потоки, не ждущие чего-либо
- ▶ НЕ реальное время
 - ▶ Нельзя делать предположения, когда потоку дадут поработать
- ▶ Из рассматриваемых потоков выбираются только те, у кого наибольший приоритет
 - ▶ В винде приоритеты потоков от 0 до 31, обычно 8
 - ▶ Есть ещё приоритеты процессов: Idle, Below, Normal, Normal, Above Normal, High и Realtime
 - ▶ Относительные приоритеты потоков: Idle, Lowest, Below Normal, Normal, Above Normal, Highest и Time-Critical
 - ▶ Истинный приоритет получается из относительного приоритета и приоритета процесса

Foreground- и Background-потоки

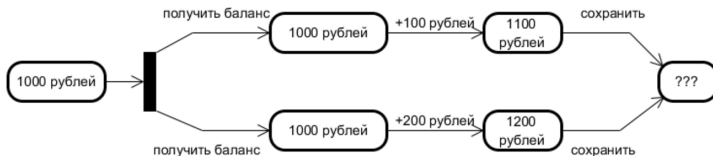
- ▶ Когда все Foreground-потоки завершили работу, рантайм останавливает все Background-потоки и заканчивает работу приложения
- ▶ Thread по умолчанию создаётся как Foreground
 - ▶ Способ прострелить себе ногу №1: создать foreground-поток и забыть о нём, приложение будет висеть в списке задач и не завершится

open System.Threading

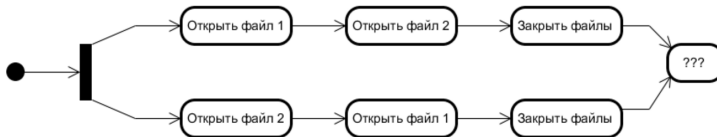
```
let t = new Thread(ThreadStart(fun _ ->
    Thread.Sleep 1;
    printfn "Hello, world")))
t.IsBackground <- true
t.Start()
printfn "Exiting Main"
```


Потенциальные проблемы с потоками

► Гонки (Race condition)



► Тупики (Deadlock)



Пример гонки

open System.Threading

```
type MutablePair<'a,'b>(x:'a, y:'b) =  
    let mutable currentX = x  
    let mutable currentY = y  
    member p.Value = (currentX, currentY)  
    member p.Update(x, y) =  
        currentX <- x  
        currentY <- y
```

```
let p = MutablePair (0, 0)
```

```
Async.Start (async { while true do p.Update(10, 10) })
```

```
Async.Start (async { while true do p.Update(20, 20) })
```

```
Async.RunSynchronously (async { while true do printfn "%A" p.Value })
```

Примитивы синхронизации

- ▶ Лучше необходимости синхронизации вообще избегать
- ▶ Бывают:
 - ▶ User-mode — атомарные операции, реализующиеся на процессоре и не требующие участия планировщика
 - ▶ Kernel-mode — примитивы, управляющие тем, как поток обрабатывается планировщиком
 - ▶ Более тяжеловесные и медленные (до 1000 раз по сравнению с “без синхронизации вообще”)
 - ▶ Позволяют синхронизировать даже разные процессы

Пример примитива синхронизации: монитор

```
let lock (lockobj : obj) f =  
    Monitor.Enter lockobj  
    try  
        f()  
    finally  
        Monitor.Exit lockobj
```

```
Async.Start (async {  
    while true do lock p (fun () -> p.Update(10, 10)) })
```

```
Async.Start (async {  
    while true do lock p (fun () -> p.Update(20, 20)) })
```

Примитивы синхронизации

Пространство имён System.Threading

Примитив	Описание
AutoResetEvent	Точка синхронизации. WaitOne блокирует поток, пока кто-нибудь другой не вызовет Set.
ManualResetEvent	То же, что AutoResetEvent, но сбрасывается вручную, вызовом Reset
Monitor	Ограничивает доступ к критической секции
Mutex	Ограничивает доступ к критической секции, работает между процессами
Semaphore	Позволяет находиться в критической секции не более N потоков
Interlocked	Атомарные арифметические операции

Управление планировщиком

- ▶ **Thread.Sleep(0)** — ничего не делает, если остальные готовые потоки меньше приоритетом
- ▶ **Thread.Sleep(1)** — отдаёт управление потоку, даже если его приоритет меньше
- ▶ **Thread.Yield()** — нечто среднее (не вызовет переключения потоков, если желающих нет, в отличие от **Thread.Sleep(1)**, но отдаст ядро потоку с меньшим приоритетом)
- ▶ **Thread.SpinWait** — просьба процессору с гипертредингом отдать ядро
- ▶ Очередной способ прострелить себе ногу — инверсия приоритетов
 - ▶ Поток с низким приоритетом захватил ресурс, нужный потоку с высоким приоритетом
 - ▶ Поток с высоким приоритетом крутится в ожидании, никогда не отдавая управление потоку, который мог бы отдать ресурс (livelock)

BackgroundWorker

Более высокоуровневый способ работы с потоками

```
let worker = new BackgroundWorker()
```

```
let numIterations = 1000
```

```
worker.DoWork.Add(fun args ->
```

```
    let rec computeFibonacci resPrevPrev resPrev i =
```

```
        let res = resPrevPrev + resPrev
```

```
        if i = numIterations then
```

```
            args.Result <- box res
```

```
        else
```

```
            computeFibonacci resPrev res (i + 1)
```

```
computeFibonacci 1 1 2)
```

BackgroundWorker, как запустить

```
worker.RunWorkerCompleted.Add(fun args ->  
    MessageBox.Show (sprintf "Result = %A"  
        args.Result) |> ignore)  
  
worker.RunWorkerAsync()
```


События

F# Interactive

```
> open System.Windows.Forms;;  
> let form = new Form(Text="Click Form",  
    Visible=true, TopMost=true);;  
val form : Form  
  
> form.Click.Add(fun evArgs -> printfn "Clicked!");;  
val it : unit = ()  
  
> form.MouseMove.Add(fun args -> printfn "Mouse,  
    (X,Y) = (%A,%A)" args.X args.Y);;  
val it : unit = ()
```

Microsoft.FSharp.Control.Event

Form.MouseMove

```
|> Event.filter (fun args -> args.X > 100)
|> Event.add (fun args -> printfn "Mouse,
(X,Y) = (%A,%A)" args.X args.Y)
```

Что ещё с ними можно делать

Примитив	Описание
add	$(T \rightarrow \text{unit}) \rightarrow \text{IEvent}\langle \text{'Del}, T \rangle \rightarrow \text{unit}$
filter	$(T \rightarrow \text{bool}) \rightarrow \text{IEvent}\langle \text{'Del}, T \rangle \rightarrow \text{IEvent}\langle T \rangle$
choose	$(T \rightarrow U \text{ option}) \rightarrow \text{IEvent}\langle \text{'Del}, T \rangle \rightarrow \text{IEvent}\langle U \rangle$
map	$(T \rightarrow U) \rightarrow \text{IEvent}\langle \text{'Del}, T \rangle \rightarrow \text{IEvent}\langle U \rangle$
merge	$\text{IEvent}\langle \text{'Del}1, T \rangle \rightarrow \text{IEvent}\langle \text{'Del}2, T \rangle \rightarrow \text{IEvent}\langle T \rangle$
pairwise	$\text{IEvent}\langle \text{'Del}, T \rangle \rightarrow \text{IEvent}\langle T * T \rangle$
partition	$(T \rightarrow \text{bool}) \rightarrow \text{IEvent}\langle \text{'Del}, T \rangle \rightarrow \text{IEvent}\langle T \rangle * \text{IEvent}\langle T \rangle$
scan	$(U \rightarrow T \rightarrow U) \rightarrow U \rightarrow \text{IEvent}\langle \text{'Del}, T \rangle \rightarrow \text{IEvent}\langle U \rangle$
split	$(T \rightarrow \text{Choice}\langle U1, U2 \rangle) \rightarrow \text{IEvent}\langle \text{'Del}, T \rangle \rightarrow \text{IEvent}\langle U1 \rangle * \text{IEvent}\langle U2 \rangle$

Как описывать свои события

open System

open System.Windows.Forms

type RandomTicker(approxInterval) =

let timer = **new** Timer()

let rnd = **new** System.Random 99

let tickEvent = **new** Event<_>()

let chooseInterval() :int =

 approxInterval + approxInterval / 4
 - rnd.Next(approxInterval / 2)

do timer.Interval <- chooseInterval()

Как описывать свои события (2)

```
do timer.Tick.Add(fun args ->  
    let interval = chooseInterval()  
    tickEvent.Trigger(interval)  
    timer.Interval <- interval)  
  
member x.RandomTick = tickEvent.Publish  
member x.Start() = timer.Start()  
member x.Stop() = timer.Stop()  
  
interface IDisposable with  
    member x.Dispose() = timer.Dispose()
```

Пример использования

F# Interactive

```
> let rt = new RandomTicker(1000);;
val rt : RandomTicker
> rt.RandomTick.Add(fun nextInterval -> printfn "Tick,
    next = %A" nextInterval);;
val it : unit = ()

> rt.Start();;
Tick, next = 1072
Tick, next = 927
Tick, next = 765
...
val it : unit = ()
> rt.Stop();;
val it : unit = ()
```

Свой worker, с событиями

```
open System.ComponentModel
open System.Windows.Forms
```

```
type IterativeBackgroundWorker<'a>(oneStep:('a -> 'a),
  initialState:'a,
  numIterations:int) =
  let worker =
    new BackgroundWorker(WorkerReportsProgress = true,
      WorkerSupportsCancellation = true)

  let completed = new Event<_>()
  let error = new Event<_>()
  let cancelled = new Event<_>()
  let progress = new Event<_>()
```

Свой worker (2)

```
do worker.DoWork.Add(fun args ->
let rec iterate state i =
  if worker.CancellationPending then
    args.Cancel <- true
  elif i < numIterations then
    let state' = oneStep state
    let percent = int ((float (i + 1)
      / float numIterations) * 100.0)
    do worker.ReportProgress(percent, box state);
    iterate state' (i + 1)
  else
    args.Result <- box state
```

```
iterate initialState 0)
```


Свой worker (3)

```
do worker.RunWorkerCompleted.Add(fun args ->  
    if args.Cancelled then cancelled.Trigger ()  
    elif args.Error <> null then error.Trigger args.Error  
    else completed.Trigger (args.Result :?> 'a))
```

```
do worker.ProgressChanged.Add(fun args ->  
    progress.Trigger (args.ProgressPercentage, (args.UserState :?> 'a)))
```

```
member x.WorkerCompleted = completed.Publish
```

```
member x.WorkerCancelled = cancelled.Publish
```

```
member x.WorkerError = error.Publish
```

```
member x.ProgressChanged = progress.Publish
```

```
member x.RunWorkerAsync() = worker.RunWorkerAsync()
```

```
member x.CancelAsync() = worker.CancelAsync()
```

Тип того, что получилось

```

type IterativeBackgroundWorker<'a> =
  class
    new : oneStep:('a -> 'a)
      * initialState:'a
      * numIterations:int
      -> IterativeBackgroundWorker<'a>
    member CancelAsync : unit -> unit
    member RunWorkerAsync : unit -> unit
    member ProgressChanged : Event<int * 'a>
    member WorkerCancelled : Event<unit>
    member WorkerCompleted : Event<'a>
    member WorkerError : Event<exn>
  end

```

Пример использования

```
let fibOneStep (fibPrevPrev:bigint,fibPrev) =  
    (fibPrev, fibPrevPrev + fibPrev)
```

```
let worker = new IterativeBackgroundWorker<_>(fibOneStep,  
    (1I, 1I), 100)
```

```
worker.WorkerCompleted.Add(fun result ->  
    MessageBox.Show(sprintf "Result = %A" result) |> ignore)
```

```
worker.ProgressChanged.Add(fun (percentage, state) ->  
    printfn "%d%% complete, state = %A" percentage state)
```

```
worker.RunWorkerAsync()
```

Своё новое событие

open System

open System.Threading

```
type IterativeBackgroundWorker<'a>(...) =
```

```
    let worker = ...
```

```
    let syncContext = SynchronizationContext.Current
```

```
    do if syncContext = null then failwith
        "no synchronization context found"
```

```
    let started = new Event<_>()
```

```
    do worker.DoWork.Add(fun args ->
        syncContext.Post(SendOrPostCallback(fun _ ->
            started.Trigger(DateTime.Now)),
            state= null))
```

```
    ...
```

```
    member x.Started = started.Publish
```

Атомарные операции

- ▶ Нет синхронизации — нет deadlock-ов!
- ▶ Чтения и записи следующих типов всегда атомарны: Boolean, Char, (S)Byte, (U)Int16, (U)Int32, (U)IntPtr, Single, ссылочные типы
- ▶ Volatile
 - ▶ Volatile.Write
 - ▶ Volatile.Read
 - ▶ Связано с понятием Memory Fence, требует синхронизации ядер
 - ▶ Есть атрибут VolatileField
 - ▶ Volatile.Write должен быть последней операцией записи, Volatile.Read — первой операцией чтения

Пример

```
let mutable flag = 0
let mutable value = 0

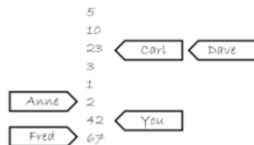
let thread1 () =
    value <- 5
    Volatile.Write(ref flag, 1)

let thread2 () =
    if Volatile.Read(ref flag) = 1
    then
        printfn "%d" value;
```

Синхронизация ядер, метафора

Relaxed ordering

- ▶ Каждую атомарную переменную можно понимать как список значений
- ▶ Каждый поток может спросить текущее значение, переменная вернёт ЛЮБОЕ значение из списка (текущее или одно из предыдущих)
- ▶ Переменная “запомнит”, какое значение она вернула этому потоку
- ▶ Когда поток спросит значение в следующий раз, она вернёт ЛЮБОЕ значение между текущим и последним, которое она вернула ЭТОМУ потоку



Interlocked

- ▶ Одновременные чтение и запись в одной “транзакции”
 - ▶ Increment : location:int byref -> int
 - ▶ Decrement : location:int byref -> int
 - ▶ Add : location1:int byref * value:int -> int
 - ▶ Exchange : location1:int byref * value:int -> int
 - ▶ CompareExchange
: location1:int byref * value:int * comparand:int -> int

Interlocked lock-free-максимум

```

let maximum target value =
  let mutable currentVal = target
  let mutable startVal = 0
  let mutable desiredVal = 0
  let mutable isDone = false
  while not isDone do
    startVal <- currentVal
    desiredVal <- max startVal value
    // Тут другой поток мог уже испортить target, так что если она изменилась,
    // надо начать всё сначала.
    currentVal <- Interlocked.CompareExchange(ref target, desiredVal, startVal)
    if startVal = currentVal then
      isDone <- true
  desiredVal

```