Multi-tasking, GCD

IZA, Martin Hrubý, FIT VUT, 2018

Úvod

- Funkce. Bloky. Closures.
- Vlákna.
- Fronty.
- Grand Central Dispatch (GCD).
- NSOperation.
- (RunLoop)

Historie paralelismu

- Task-switching.
- UNIX — fork(), více-uživatelský běh OS.
- Apple — kooperativní multitasking.
- Preemptivní multitasking.
- Vlákna, POSIX threads.
  - Procesy versus vlákna.
  - Paměťová režie — dynamický zásobník, paměť pro registry.
- Multi-tasking versus Paralelismus.

Důvody pro paralelismus

- Rozklad výpočetní zátěže na více zdrojů.
- iOS aplikace: nebrzdit UI výpočetní zátěží.
- Budeme rozlišovat:
  - operace nad UI — konfiguraci GUI prvků, tabulek apod. Zpracování událostí od UI.
  - výpočetní zátěž — komunikaci, správu dat, multi-media apod.
  - časové latence — typicky komunikace, soubory.

Paralelní programování

- Specializované algoritmy nad spec. HW — maticové výpočty, filtry apod.
  - Podstatu problému lze převést na triviální datové struktury. VLIW.
- Komplexní výpočty — program je soustavou procesů (simulace, komunikace).
  - Je nezbytné programovat ”komfortně” (vizte kooperativní multitasking). Abstrakce nad paralelismem. GCD/Operations.
  - Program se rozdělí do elementárních *bloků*. Správa bloků.

Paral programování formou bloků

- Terminologie: Task. Activity. Job. Block.
- Program definuje blok kódu. Kontext bloku.
- Máme fronty bloků — sériové (sekvenční), paralelní.
- Někdo tyto bloky vykonává.
- Problém: vhodně rozložit program na bloky.
  - Safari.

Paralelní programování a HW

- Odlišujme:
  - programování paralelismu — paralelismus je vlastnost/schopnost programu.
  - realizaci paralelismu — OS plánuje vlákna na CPU.
- Škálovatelnost — program nerozlišuje, kolik CPU ho vykonává.
- Abstrakce paralelismu — fronty a bloky.

Funkce

- (Globální) funkce:
  - má parametry — proměnné, které funkce vlastní. Proměnné drží hodnotu (referencí, hodnotou/kopií).
  - přistupuje na proměnné mimo svůj rámec — globální proměnné.
- Lokální funkce (metoda) — má ”self” referenci a přes self. přístup na instanční proměnné objektu.

Funkce jako data

- Funkce je místo v kódovém segmentu, kam lze ”skočit” — parametry, zásobník, návratová hodnota.
- Funkci tedy lze *referencovat*.
  - Funkce je ve Swiftu referencovaný objekt.
  - Referenci lze předat.
  - Funkce smí přistupovat k datům mimo svůj rámec.
  - … jenom je třeba okolní data ”podchytit” (closure).

Funkce jako data

```
// typ "FUN" je referenci na funkci vracejici Int
typealias FUN = () -> Int
// globalni data
var glob1 : Int = 100
// globalni funkce
func prvni() -> Int { return 1 }
func druha() -> Int { return glob1 }
// ref na dve funkce
let fun1 : FUN = prvni
let fun2 : FUN = druha
//
print(fun1())
// funkci lze referencovat promennou
var fun1a : FUN? = prvni
//
fun1a?()
fun1a = nil
//
print(fun2()) // tisk: 100
glob1 = 200
print(fun2()) // tisk: 200
```

Co s ref na funkci lze dělat?

- Manipulace s func-ref —> vzniká closure.
- Uložit do proměnné. Volat přes referenci.
- Předat jako parametr při volání metody.
- Uložit do datové struktury.
- Escaping/Non-escaping.
- Non-escaping referenci na funkci lze pouze volat, ne uložit.

Co je @escaping

- Je to informace pro překladač/programátora, že vykonání bloku/funkce může/smí být libovolně odloženo.
  - Je asynchronní.
  - Je nezávislé od původního kontextu běhu programu. Lze volání odložit.
- Potřebu deklarace @escaping odhalí překladač.
- Kopírování dat ”z okolí” funkce.

Synchronní použití bloků

```
func doSomething(_ a: FUN) -> Int {
    // podprogram "a" je volan
    return a()
}

func prvni() -> Int { return 1 }
// jdu do podprogramu "doSomething"
// a predavam mu dalsi podprogram "fun1"
doSomething(prvni)
```

Asynchronní použití bloku

```
class MQueue {
    // fronta prace
    var q : [FUN] = []
    // blok "a" je escaping
    func add(_ a: @escaping FUN) {
        //
        q.append(a)
    }
    // zpracovani fronty
    func work() -> [Int] {
        //
        return q.map { $0() }
    }
}

let q = MQueue()
// volam podprogram s predanim "prvni", ovsem
// prvni() se neaktivuje. Muze byt spusteno kdykoli jindy
q.add(prvni)
q.add(druha)

print(q.work())
```

Lokální funkce (metoda)

- Je to stále místo v kódu, kam lze ”skočit”.
- Třídní/instanční metoda.
- V jejím kontextu jsou instační proměnné objektu/třídy.
- *Když předám referenci na lokální funkci (metodu) — co tím míním? (ve vztahu k lokálním datům)*

```
class TRIDA {
    // instacni promenna
    let val : Int
    //
    init(_ v: Int) { self.val = v }
    // zprava objektu o dealokaci
    deinit {
        print("Objekt TRIDA:\(val) konci")
    }
    // je treba tam videt to nepsane ”self.val”
    func metoda() -> Int { return val }
}
//
var otr1 : TRIDA? = TRIDA(123)
var funT1 : FUN? = otr1!.metoda
// dereferencuju objekt. Ten je dal referencovan pres "funT1"
otr1 = nil
// funT1?()
if let _f = funT1 {
    //
    print(_f()) // 123
}
//
funT1 = nil
// Objekt TRIDA:123 konci
```

Closures

- Je to forma hodnoty, která kombinuje referenci na blok a kopíruje hodnoty, které blok používá:
  - zvyšuje refCount referencí,
  - provádí kopii struktur.

```
// Explicitni model closure
struct TridaMetodaClosure {
    let him : TRIDA
    //
    func metoda() -> Int { return him.val }
}
```

Terminologie

- Closure expression — blok (parametry, návratová hodnota, tělo).
- Funkce je pojmenovaný closure expression.
  - … a má implicitní přístup na ”self”
- Pokud svážu CE s okolními daty, vzniká closure.
  - Můžeme říct, že je to forma instanciace CE.

Model closure

```
class TRIDAB {
    //
    let val : Int = 2
    //
    func metoda() -> FUN {
        //
        var inte = 10
        //
        func minside() -> Int { return inte + val }
        //
        return minside
    }
}
// Explicitni model closure
struct TridaBMetodaClosure {
    let him : TRIDAB
    let inte : Int // spis ”inner-self” metody
    //
    func minside() -> Int { return inte + him.val }
}
```

Escaping / non-escaping

- Charakter použití bloku pozná překladač.
- Má-li být blok escaping, pak musí explicitně uvádět ”self”.
  - Pro programátora žádné větší omezení.
  - Rozdíl může být v implementaci kopie ”self”, specificky pro struktury (pokud neuvažujeme ”mutating”).
- Weak self — ref cykly. Chci potlačit blok, když ”self” už nemá existovat. Unowned — lifeTime.

Weak self

```
//
var poleFUN : [FUN] = []
//
class WTrida {
    let data : Int = 1

    func pass() {
        //
        poleFUN.append( {
            [weak self] in
            guard let _s = self else { print("Jsem NIL"); return 0 }
            return _s.data
        })
    }
}

//
WTrida().pass()

//
poleFUN.first!() // Jsem NIL
```

@autoclosure

```
// val je typu @autoclosure
func logovani(_ val: @autoclosure () -> Bool, _ s: String) {
    //
    if val() == true {
        //
        print(s)
    }
}

// prekladac zabali vyraz do bloku ()->Bool
logovani(true, "Ahoj")
logovani(3 > 2, "3 je vic")
```

Trailing closures

- Pokud je poslední argument funkce blokem, lze redukovat syntaxi volání funkce.

```
func gogo(closure: ()->Void) {
    print("Do something")
    closure()
}

// volam gogo
gogo(closure: { print("Hello") })

// volam gogo
gogo {
    print("Hello again")
}
```

Closures v GCD

- Budeme definovat bloky umístěné v kontextu objektů. Closures.
- Tyto closures budeme umisťovat do front.
- Někdo je bude spouštět.
  - … v kontextu těch objektů.

```
class NejakaTrida {
    //
    func jejiMetoda() {
        //
        nejakyBlok v metode {
            // invokace bloku
            //
        }
    }
}
```

```
class NejakaTrida {
    //
    func jejiMetoda() {
        //
        fronta.tentoBlokNaplanujNaSpusteni {
            // invokace bloku
            //
        }
    }
}
```

Vlákna

- Kód je prováděn vlákny — obecně více-vláknově.
- POSIX Threads. Abstrakce NSThread / Thread.
- Vlákno nejde "zabít". Můžete mu poslat zprávu, na základě které se dobrovolně ukončí:
  - ***Vlákno A tudíž nemůže zabít vlákno B***.
  - Vrátí se na počátek svého zásobníku, pak OS.
  - Vlákno se zahajuje skokem do nějaké procedury.

Přehled o multi-taskingu

- Thread — mainThread a ta ostatní vlákna.
- Fronty činností.
- Grand Central Dispatch — fronty a bloky.
- NSOperation a fronty.

NSThread / Thread

- Zjistit, jakým vláknem je prováděn kód.
- Ovládat vlákno — sleep, lock, condition.
- Vytvořit vlákno.
  - odvodit z třídy Thread, pak metoda ”main”
  - předat referenci a selektor
- Uplatnění přímo Thread v aplikacích?

Vlákna a objekty

- NSThread je abstrakcí nad POSIX thread.
  - tj. je to de facto objekt.
  - objekty si mohou posílat zprávy, tj. lze objektu Thread poslat zprávu,
  - to však neznamená, že lze vlákno ovládat.

```swift
protocol MThreadDelegate : class {
    //
    func updateUI(fromThread: MThread)
}

class MThread: Thread {
    //
    weak var delegate: MThreadDelegate?
    //
    override func main() {
        //
        while true {
            // dostal jsem zpravu cancel()?
            if isCancelled { break; }
            // pauza
            Thread.sleep(forTimeInterval: 1)
            // dostal jsem zpravu cancel()?
            if isCancelled { break; }
            // neco delej
            delegate?.updateUI(fromThread: self)
        }
    }
}
```

Třídní metody Thread

- Vrací hodnoty v kontextu provádějícího vlákna.
- Thread.sleep(:) — toto vlákno si dá N sekund pauzu (OS ho odloží).
- Thread.current — toto vlákno vrátí svůj ”self”,
- Thread.isMainThread — vykonávající vlákno odpoví, zda-li je tzv. hlavním vláknem.
- Thread.exit() — *toto vlákno* se ukončí.
  - Rozešle zprávu (Notif. center). Zřejmě neprovede korektní dealokaci/uzavření zdrojů.

Uplatnění Thread

- Thread — je to setrvalá činnost (není to operace/procedura).
- Nestaví se do fronty jako Operation.
- Na signál se může ukončit (kooperativně).
  - Nelze znovu-spustit.
- Komunikace do aplikace: Delegát, NotifCenter.

Suspendování Thread

- Tj. vlákno se dobrovolně odstaví.
- Typicky zámkem.
- NSLock, NSCondition.

Suspendování Thread — kvazi

```
class MThread: Thread {
    //
    weak var delegate: MThreadDelegate?

    //
    var myLock = NSLock()

    //
    func pauseMe() { myLock.lock() }
    func unpauseMe() { myLock.unlock() }

    //
    override func main() {
        //
        while true {
            //
            myLock.lock(); myLock.unlock()
            //
            print("jdu na nejakou praci")
        }
    }
}
```

NSCondition, ThreadPool

- NSCondition — bariéra, kde vlákna čekají na signál (pro jednoho, pro všechny)
  - wait — čekání, vlákno je odstaveno
  - signal — probuzení vláken.

Suspendování thread — cond

```
class MThread: Thread {
    //
    var myCond = NSCondition()

    // uvolni cekajici vlakno
    func unpauseMe() {
        myCond.signal()
    }

    //
    func main() {
        //
        while true {
            // cekam na pokyn
            myCond.wait()

            // prace
        }
    }
}
```

```
class MThreadPool {
    //
    typealias Hokna = ()->()
    var hokna : [Hokna] = []
    var myCond = NSCondition()
    var myLock = NSLock()
    //
    func addHokna(_ h: @escaping Hokna) {
        // zamykam data "hokna"
        myLock.lock()
        hokna.append(h)
        // uvolnuju zamek na "hokna" a spoustim vlakno
        myLock.unlock()
        myCond.signal()
    }
```

```
class MThreadPool {
    //
    func getHokna() -> Hokna? {
        //
        print("Do fronty na praci")
        // cekame tu vsichni na signal
        myCond.wait()
        // byl jsem probuzen
        print("Probuzen jeden?")
        // zichr pro "hokna"
        myLock.lock()
        // nejaka chyba, proc me budi?!
        if hokna.isEmpty == true {
            // !!! unlock()
            return nil
        }
        //
        let _f = hokna.removeFirst()
        //
        myLock.unlock()
        //
        return _f
    }
```

```
class MThread: Thread {
    //
    var tp : MThreadPool!

    //
    override func main() {
        //
        while true {
            //
            if let _hokna = tp.getHokna() {
                //
                print("jdu na nejakou praci")

                //
                _hokna()
            }
        }
    }
}
```

```
class VC: UIViewController {
    //
    @IBOutlet var lab : UILabel!
    //
    var thList : [MThread] = []
    var thPool = MThreadPool()
    //
    @IBAction func butt() {
        //
        thPool.addHokna {
            //
            print("Zazpivame si...")
        }
    }
    //
    override func viewDidLoad() {
        //
        super.viewDidLoad();
        //
        for _ in 0..<10 {
            //
            let _thr = MThread()
            //
            _thr.tp = thPool
            thList.append(_thr)
            _thr.start()
        }
    }
```

Grand Central Dispatch

- Aplikační Thread-Pool nad systémem front.
  - Dispatch Queues. Serial (sekvenční), Concurrent (paralelní).
- MainQueue — z ní bere pouze mainThread.
- GlobalQueue — všechna vlákna.
- U front nás zajímá:
  - charakter fronty,
  - způsob vkládání do fronty.

Fronty

- Sekvenční — operace se zahájí po dokončení předchozí operace ***z této fronty***.
- Paralelní — operace se zahajují podle možností pracujících vláken v pořadí z této fronty.
- Pouze operace z MainQueue (je sekvenční) smí zasahovat do UI. Tyto operace vykonává výhradně MainThread.

Vkládání do front

- Asynchronně — blok se vloží do fronty (vykonáno currentThread) a currentThread pokračuje dále.
- Synchronně (de facto volání podprogramu):
  - blok se vloží do fronty,
  - currentThread čeká na jeho dokončení,
  - blok se ovšem spustí, až na něj dojde řada.
  - *Implementace*?

Sync / Async

```
DispatchQueue.main.async {
    //
    print("Hello")
}

DispatchQueue.global().async {
    //
    // otevri soubor
     // sync/async
    DispatchQueue.main.sync {
        //
        // posli zpravu UI, soubor otevren
    }
}
```

Rozložení zátěže

```
// objekty pro zpracovani
var data = [nejaky obsah]

// posli pracovni bloky do global
// neni synchronizace na vlakno
for i in data {
    //
    DispatchQueue.global().async {
        // pracuj nad objektem
        work(i)
    }
}
```

Rozložení zátěže, agregace

```
//
let dGroup = DispatchGroup()
// vlakno je clenem skupiny
dGroup.enter()
//
for i in data {
    // vytvarim bloky registrovane do skupiny
    DispatchQueue.global().async {
        // pracuj nad objektem
        work(i)
        // vlakno opousti skupinu
        dGroup.leave()
    }
}
// cekam, dokud neni skupina prazdna
dGroup.wait()
```

Synchronní vkládání

- Vkládající vlákno čeká. Co když je jím MainThread? :) Má charakter stárnutí.

```
class VC: UIViewController {
    //
    override func viewDidLoad() {
        //
        super.viewDidLoad();

        // tady to slitne na EXC_BAD_INSTRUCTION
        DispatchQueue.main.sync {
            //
            print("Hello")
        }
    }
}
```

Dispatch Once — historické demo

```
class Trida {
    // token, metadata pro rizeni dispatchOnce
    static var __token : dispatch_once_t = 0
    //
    static var __instance : Trida?
    //
    class  var shared : Trida {
        //
        dispatch_once(&__token, {
            //
            __instance = Trida()
        })
        //
        return __instance
    }
}
```

Swift, singletony

```
class Trida {
    // Swift garantuje thread-safe, unikatni provedeni
    static let shared = Trida()
    //
    static let sharedAdd : Trida = {
        //
        let vv = Trida()
        //
        vv.initme()
        //
        return vv
    }()

    //
    func printme() { print("hello")}
    func initme() { print("Init Trida")}
}

let v = Trida.shared
let va = Trida.sharedAdd

//
v.printme()
```

Operace GCD

- Vytváření uživatelských front (sekvenční, paralelní).
- Vkládání do front.
- Quality of servis — vlastnosti fronty.

Přeskakování mezi frontami

- MainThread potřebuje data pro UI:
  - plánuje do Global výpočetní operaci
  - výpočetní operace dokončí, pak:
    - plánuje do Main UI-operaci

```
@IBAction func sezenMiObrazek() {
        //
        DispatchQueue.global.async {
            //
            DispatchQueue.main.async {
                //
            }
        }
    }
```

Shrnutí GCD

- Skvělý nástroj pro rozklad zátěže a řízení synchronizace.
- Vlastnosti:
  - Odeslaný blok už nelze ”vzít zpátky”.
  - Není kontrola nad odeslaným blokem. Vnitřní stav. Komunikace.
  - Nepokoušejte se bloku ve frontě posílat zprávy.

NSOperation/Operation

- Nutno odvodit vlastní (Operation je abstraktní).
- Je to objekt představující aktivitu (task, blok).
- Opět: fronty (sekvenční, paralelní).
- Lze přistupovat k jeho vnitřnímu stavu.
- Vhodné pro akce, které mají nějaký životní cyklus (síťové požadavky, Cloudkit).

Vlastnosti operation

- Lze dát ”dependencies” — prerekvizity.
- Lze definovat completionBlock.
- Priority.

Plánování operací, fronty

- Typicky se předpokládá plánování do fronty a řízení přes GCD, tj. vlákny z Poolu.
  - pak mluvíme o synchronním provedení operace.
  - start() -> main()
- Operaci taky lze spouštět přímo, pak dává smysl asynchronní pojetí a dedikované vlákno.
  - start()->alokace vlákna nad main()
  - precedence?

OperationQueue

- OperationQueue je postavena nad DispatchQueue (GCD).
  - OQ lze navázat na vybranou DQ.
- Pořadí vykonávání operací je obecné až na dependencies.

```
class MojeOperace : Operation {
    let mname : String
    //
    init(_ mn: String) {
        self.mname = mn;
    }
    //
    override func main() {
        //
        print("pracuju...\(mname)")
    }
}
//
let prvni = MojeOperace("prvni")
let druha = MojeOperace("druha")
// druha ma "prvni" jako precedencni
druha.addDependency(prvni)
// uzivatelska OQ
let myoq = OperationQueue()
// navazana na globalni GCD frontu
myoq.underlyingQueue = DispatchQueue.global()
// zvladne pre-usporadat
myoq.addOperations([druha,prvni], waitUntilFinished: true)
```

Přehled

- Thread — soustavná činnost
- GCD (staví nad Threads) — systém bloků. K bloku program dále nepřistupuje.
- Operation (staví nad GCD) — systém operací ve tvaru objektů (lze referencovat, vnitřní data).
  - Lze specifikovat unikátní vlákno (asynchronní operace).
  - Celkově lze Operation brát jako jednoúčelový Thread.

Příště

- Kódování dat.
- Documents.
- Key-Value Coding.
- Notification Center.