Multi-threading, GCD, Async

Programování zařízení Apple (IZA)

Martin Hrubý, FIT VUT, 2022/23

Úvod

Širší souvislosti:

- Asynchronnost - program je událostně řízený (RunLoop), často se obracet přes RunLoop.
  - podobně v serverových aplikacích (služby).
- Zařízení má více CPU, rozložení zátěže. Škálovatelnost.
- UI/GUI a kritické části - hlavní vlákno.
  - rezervovat hlavní vlákno pro UI - plynulost UI aplikace.

Historie paralelizmu

- Task-switching (90tá léta, MS-DOS/Windows).
- Apple: kooperativní multi-tasking (McIntosh, 1984).
- *Preemptivní* multi-tasking. UNIX (fork, víceuživatelský běh).
- Vlákna (POSIX threads).
    - Sémantika řízení vláken.

K diskuzi:

- Procesy versus vlákna (najdi rozdíl).
- Paměťová režie procesů a vláken.
- Multi-tasking versus paralelizmus (najdi rozdíl).

Paralelní programování

- Máme zařízení s potenciálně více CPU.
- Program je paralelizovatelný. Přínosy?
- Bezpečně a efektivně napsat paralelní program.
- Program (resp jeho paralelismus) je škálovatelný.

Jaké jsou problémy paralelních programů?

- … k diskuzi :)

Dá se více-vláknovosti (a potenciálně pak paralelizaci) na iOS/SwiftUI vyhnout?

- … k diskuzi :)

Co je běh aplikace?

Jádro OS (iOS/macOS) vytváří a spravuje vlákna.

- OS přiřazuje vláknům čas na CPU. App v pozadí/popředí.

Vlákno nelze zabít/přerušit. Jde mu nepřidělit další čas CPU.

- Vlákno vykonává kód. Až kód dokončí, řekne OS, že končí,
- musí se ukončit "z vlastní vůle a dobrovolně". Kooperativně.

```
void totoJeKodVlakna() {
    // az vyskocim z tela funkce, vlakno konci
}
...
auto _thr = std::thread(totoJeKodVlakna)
//
// _thr.join()
```

Život vlákna

```
//
bool _mojeVlaknoRunning = true;
//
void totoJeKodVlakna() {
    // az vyskocim z tela funkce, vlakno konci
    while (_mojeVlaknoRunning == true) {
        // neco delej
    }
}
...
auto _thr = std::thread(totoJeKodVlakna)
// Posilam pres data zpravu, at dobrovolne skonci
_mojeVlaknoRunning = false;
```

Život vlákna/operace/bloku/...

Budeme zkoumat možnosti řízení běhu nějaké paralelní aktivity.

Typicky však ve všech paralelizačních systémech:

- co nastartujeme, už musíme nechat doběhnout
- co dáme do fronty, už musí nastartovat

V tomto smyslu v aplikacích poběží spousta výpočtů, jejichž výsledek nebude nikoho zajímat :)

Co je běh aplikace?

OS vytvoří aplikaci 1 MT a několik GT.

- Main Thread, Global Threads (podle počtu jader CPU)
- Tato vlákna ”žijí v RunLoop” aplikace.

RunLoop:

- výchozí kód pro všechna vlákna

Pozn.: správa paměti na iOS zařízeních (memoryWarning).

RunLoop

- Vstupní fronta událostí z OS (multi-touch, ...).
- Fronty práce (bloky) - MainQueue, GlobalQueue, ...
- Systémové zdroje (GUI, časovače).
- Thread Pool (MainThread, další vlákna)

Naplánování bloku do fronty hlavního vlákna.

```
DispatchQueue.main.async {
    // něco udělej v hlavním vlákně
}
```

Fronty GCD v RunLoop

Grand Central Dispatch (GCD). Knihovna v rámci Foundation.

- Fronty (sekvenční a paralelní) a vlákna "žijící v RunLoop".
- Významná fronta: MainQueue (je sekvenční).
- Důležité: rendering obrazovky UI se zahajuje v RunLoop (UIView: setNeedsDisplay()...).

Vlákno smí posílat zprávu pouze sobě. Co je vlákno v kontextu kódu.

```
Thread.isMainThread
Thread.sleep(...)
Thread.exit() // sám sebe...
```

UI a MainThread

Proč se má ovládat GUI pouze v MainThread. GUI je v aplikaci *sdílený prostředek*. Možnosti jsou:

- zamykat GUI při přístupu.
- dává vůbec smysl paralelní běh v GUI?
- … nebo rozhodnout GUI pouze pro jedno vlákno.

Důsledky:

- MainThread se musí často obracet přes RunLoop.
- Ostatní vlákna do kódu GUI vůbec nesmí (kritická chyba).

Vlákna v UIKit

```
class Obrazovka: UIViewController {
    //
    @IBOutlet var label: UILabel!

    //
    func aktualizujLabel(novyText: String) {
        // které vlákno tento kód vykonává
        label.text = novyText
    }
}
```

Vlákna ve SwiftUI

```
// tady mohou byt ruzna vlakna
class Model: ObservableObject {
    @Published var textLejblu = "cosi kdesi"
    // je vubec KOREKTNI pripustit GT???
    func aktualizujLabel(novyText: String) {
        // které vlákno tento kód vykonává
        textLejblu = novyText
    }
}
//
struct Obrazovka: View {
    @ObservedObject var model: Model
    // Tady NEVÍM, zda-li se někdy dostane GT
    var body: some View {
        //
        Text(model.textLejblu)
    }
}
```

Epizoda: Exceptions ve Swiftu.

Co je vlastně exception?

Konvenčně uvažujeme funkci jako podprogram s návratovou hodnotou.

```
func hello() -> String {
    return "cosi-kdesi"
}
```

Zavoláme (co se odehraje na zásobníku vlákna):

```
let result = hello()
```

Exceptions ve Swiftu.

Někdy chceme vrátit kromě hodnoty jinou hodnotu - identifikaci chyby. Typické pro *Go-lang*.

```
func hello() -> (String?, ErrorType?) {
    return ("cosi-kdesi", nil)
}
```

```
let (result,error) = hello()
//
if error == nil {
    //
    jdemeDalS(result)
} else {
    //
    return stalasechyba(error)
}
```

Exceptions ve Swiftu.

`Result<Value, Error>`

```
enum CosiError: Error {
    case faktToNeslo
    case chybaIO(String)
}

//
func cosiUdelej() -> Result<String, CosiError> {
    //
    if ... {
        return Success("ahoj)
    }

    //
    return Fail(.faktToNeslo)
}
```

Exceptions ve Swiftu.

Ryze funkcionální přístup s ošetřováním chyb může být při u některých programů **značně frustrující**.

Takže raději:

- definujeme startovní místo programu M.
- z něj voláme stromovou hiearchii funkcí,
- pokud některá z nich vytvoří chybu (v**výji**mku),
- pak se chyby koncentrují v místě M (urychlený návrat z místa chyby do místa M a obsluha chyby).

Do-try-catch

C++:

```
try {
    // v pripade vyjimky se tento kod nedokonci
    cosiVolamKdeSeAsiVyhazujeChyba();
} catch (chyba1 obj1) {
    // obsluha chyby
}
```

V C++ je

- toto dobrovolná konstrukce,
- obtížně zjistitelné, zda-li nějaká funkce generuje výjimku.

  Knihovny třetích stran.

Exception ve Swiftu:

Striktní kontextová syntaxe jazyka:

- Funkce vyhazující výjimku musí být označena ve své hlavičce.
- Funkci vyhazující výjimku lze volat pouze "ošetřujícím způsobem".

```
func hello() throws -> String {
    // ...
}
```

Tato funkce smí mít ve svém těle `throw`.

Vyhození výjimky

Výjimka je událost, kdy se

- vytvoří havarijní návratová hodnota (implementující `Error`)
- … s tou se vyskakuje ze zásobníku volání funkcí
- … až do místa `catch`, kde se hodnota zachytí a program dále normálně pokračuje.

```
// DT implementujici Error
enum BadValues: Error {
    case critical
    case justBad(howBad: String)
}
```

Vyhození výjimky

```
//
func readMyFile(name: String) throws -> String {
    // nejaka situace
    if (...) {
        throw BadValues.justBad(howBad: "Soubor mimo")
    }

    // jina situace
    if (...) {
        throw BadValues.critical
    }

    //
    return "Tady je obsah souboru"
}
```

Pozn.: Není explicitní datový typ výjimky!

Catch

Pattern matching. Zachytí se výjimka odpovídající deklaraci `catch`. Informační objekt výjimky je heterogenní.

```
var cont: String? = nil
do {
    // musim volat s "try"
    cont = try readMyFile(name: "soubor")
    //
} catch BadValues.critical {
    print("Uplne kriticke")
} catch BadValues.justBad(let jakMoc) {
    //
    print("Chybne zpusobem \(jakMoc)")
} catch {
    // zachycuju zbytek ...
}
```

try, try?, try!

Funkci typu `throws` nutno volat vždy s prefixem `try`.

Volající funkce má 3+1 možnosti ošetření:

- Buď je sama "throws".
- Nebo provádí do-try-catch.
- Nebo volá "try?". Pak je výsledek `Optional`.
- Volá "try!" a případně slítne.

```
// 1) Interni osetreni chyby, vraci hodnotu
// (pokud hodnota "0" dava nejaky smysl)
func callingIO1() -> Int {
//
    do { return try myIO() }
    catch { //
        return 0 // nebo -> Int?, nil
    }
}
// 2) Predava pripadnou vyjimku
func callingIO2() throws -> Int {
    //
    return try myIO()
}
// 3) try? vyraz je Optional, tj. Int?
func callingIO3() -> Int? {
    //
    return try? myIO()
}
// 4) try!, kaslu na chyby
func callingIO4() -> Int {
    //
    return try! myIO()
}
```

throws v knihovnách Swiftu

Na můj vkus se to s exceptions přehání.

- souborové operace, kódování
- síťové operace

V poslední době často ve spojení s `async`.

Grand Central Dispatch (GCD)

Fronty. Sekvenční/paralelní.

```
DispatchQueue.main.async {
    // cosi
}
```

- `DispatchQueue` - static `main`, `global`.
- Vkládání bloku do fronty `async` nebo `sync`.

```
class DispatchQueue {
    //
    func async(blk: @escaping ()->()) { ...}
}
```

Vložení do fronty

- Vložený (plánovaný) blok již nelze odebrat.
- Bloku nelze "během čekání" poslat zprávu. **Není referencován**.
- Operace `async/sync` do fronty přidává.

Průběh výpočtu bloku nelze ovlivnit.

- Vlákno ho v RunLoopu **vyjme z fronty** a "vstoupní do něj".

Typicky blok může naplánovat další blok. Uvidíme.

Vložení async/sync

```
DispatchQueue.NejakaQ.async(blk: @escaping ()->())
```

- Vlákno X volá async,
- Vlákno X vloží do fronty (LOCK) a pokračuje dál.
- Efekt: pouhé vložení do fronty.

```
...sync { ... }
```

- ... situace je podobná,
- vlákno X však **čeká**, než se blok nedokončí.
    - ... kdy to může mít smysl?

Demos

```
DispatchQueue.main.async {
    // neco udelej v budoucnu
}
```

Přechod do vedlejšího vlákna a zpátky:

```
DispatchQueue.global.async {
    // neco udelej ve vedlejsim vlakne
    let result = nejakyVypocet()

    // a predej ho do aplikace pres MAIN
    DispatchQueue.main.async {
        // nejak...
        app.vysledek = result
    }
}
```

Dogma.

Demos: non-sense

Prováděno MT (analyzujme):

```
DispatchQueue.main.sync {
    //
}
```

Runtime tento případ detekuje a program slítne.

Main Thread

Kód vykonávaný směrem z knihovny **UIKit/SwiftUI** je vykonáván MainThread (jistota), tj pouze z GUI.

Veškerá async volání knihoven s callback-em **preventivně explicitně přeložit do hlavního vlákna**.

```
knihovna.nactiDokument(doc: "xy") { result in
    //
    DispatchQueue.main.async {
        //
        self.dokumentHotovo(result)
    }
}
```

Async činnost = typicky ve vedlejším vlákně.

GCD fronty

`[a,b,c,d,e,f]`

- sekvenční - blok `c` se zahájí, až jsou dokončeni jeho předchůdci.
- paralelní - bloky jsou zahajovány podle kapacity dostupných výpočetních vláken, tj pořadí nelze garantovat.

Co vyplývá z MainQueue/MainThread koncepce?

- pouze MainThread bere z MainQueue
- MainQueue je sekvenční, pořadí garantováno
- když MainThread provádí `a`, tak logicky NIKDO nemůže provádět jiný blok `b,d,d,e,f,...`. Důsledek?

Global queue, QOS

`DispatchQueue.global(qos: ...).async {}`

Je atribut fronty, určuje prioritu bloků ve frontě (pro rozhodování vlákna v RunLoop).

- user-interactive — musí běžet v real-time, GUI.
- user-initiated — blok pro interakci s uživatelem bez prodlení.
- utility — delší činnost, na kterou uživatel čeká.
- background — delší činnost, na jejíž výsledek se nespěchá.

GlobalQ je tedy nejspíš multi-fronta.

```
struct ContentView: View {
    @State var lejbl = "cosi"
    //
    func akceGlobal() {
        //
        DispatchQueue.global().async {
            // zapis do property wrapper v GT
            lejbl = "jejeje"
        }
    }
    //
    func logujLejbl() -> String {
        //
        print("Vlakno \(Thread.isMainThread)")
        return lejbl
    }
    //
    var body: some View {
        // spousti se v MT
        VStack {
            Text(logujLejbl())
            Button(action: akceGlobal) { Text("global-akce") }
        }
    }
}
```

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

```
// objekty pro zpracovani
var data = [nejaky obsah]
var results = [Typ]()
//
for i in data {
    //
    DispatchQueue.global().async {
        // pracuj nad objektem
        res = work(i)
        //
        DispatchQueue.main.async {
            results.append(res)
            // pole "results" je naplneno
            if "je posledni" { volejHotov(results) }
        }
    }
}
```

Operation & OperationQueue

Nad systémem GCD stojí Operations.

- Třída `Operation`.
- Odvozeniné třídy, instance. **Referencovatelné objekty!**
  - tj mohou obsahovat své výsledky a interní data
- Precedence. Fronty.

Operation jsou referencovatelné GCD bloky.

Knihovna *CloudKit*.

Pozn.: Instanci Operation lze zařadit do fronty pouze jednou!

Operation: demo

```
class MojeOp: Operation {
    // Data vstupni a vystupni
    let inputs: Cosi
    var results: CosiKdesi?

    // konstrukce
    init(inputs: Cosi) { self.inputs = inputs }

    // telo vypoctu
    override func main() {
        // operaci LZE poslat zpravu...
        while finished == false {
            // ... results.append()
        }
    }
}
```

Operation: demo

```
//
let op = MojeOp(inputs: ...)

// MainQueue
OperationQueue.main.addOperation(op)

// obecna fronta
let myQueue = OperationQueue()

// mira maximalniho paralelismu (==1 -> SEQ)
myQueue.maxConcurrentOperationCount = 4
myQueue.addOperation(op)
```

Instancovaná `OperationQueue` se automaticky registruje v RunLoop.

Bariéra

```
let myQueue = OperationQueue();
//
let finishOp = BlockOperation {
    DispatchQueue.main.async {
        print("Hotovo")
    }
}
//
for i in 0..<10 {
    let op = BlockOperation {
        print("Ahoj")
    }
    //
    finishOp.addDependency(op)
    myQueue.addOperation(op)
}
// Precedence. Fronta s předbíháním.
myQueue.addOperation(finishOp)
```

Problém paralelních aplikací

Problém je typicky v časovém průniku více vláken:

- v nějakém kódu
- nad nějakou datovou proměnnou

Typicky toto "řešíme" zámky (mutex), nad kterými pak typicky ztrácíme kontrolu a dochází k různým jevům, např *deadlock*.

Zámky: ideálně bez nich. **Bezzámkové vícevláknové aplikace.**

Každý nový `mutex` v aplikaci = další potenciální problém.

Zásada č. 1

MainQueue/MainThread je nejpřirozenější forma řešení výlučnosti a synchronizace.

- pořadí, výlučnost, bez-zámkovost

Kód pro vyhodnocování MT nemá smysl zamykat (nikdo mu tam nevleze).

Zásada č.1.: veškerou **interakci mezi objekty** aplikace provádět v MQ/MT.

Zásada č. 1

Univerzální mustr.

```
DispatchQueue.global().async {
    //
    let result = nejakaPrace()

    // předávám výsledek do aplikace
    DispatchQueue.main.async {
        //
        app.predavamVysledek(result)
    }
}
```

Interakce mezi objekty

Vědět, jak se vykonávají interakce mezi objekty:

- Poslání zprávy přes referenci na objekt.
- Poslání zprávy přes NotificationCenter.
- Setter uložené property. Getter property.

Akce mají skryté důsledky.

- Proč? GT nesmí proniknout do kódu výlučného pro MT (UI, CoreData). Souběhy.
- Zásadu č. 1 lze porušit, pokud chápeme důsledky.

Rozhodně NE!

```
class GlobalWorker {
    // kód běží v globálním vlákně
    func runningInQlobal(owner: Owner) {
        // NE !!!
        owner.data = ...
        owner.akce(result: ...)
    }
}
//
class Owner: ObservableObject {
    //
    @Published var data: Typ
    //
    func akce(results: Typ) {}
}
```

Zásada č. 2: Odlišení v názvosloví

Implicitně: veškerý kód aplikace je určen pro běh v MT

Explicitně určený kód pro běh v Global Threads:

- explicitně označit v pojmenování "gtFunkce..."
- veškeré Operation
- veškeré `@escaping` bloky předávané do async volání.

Oddělení GT a MT světa přes MainQueue.async operaci:

```
DispatchQueue.main.async {
    //
    predejVysledky(...)
}
```

Zásada č. 2: Odlišení v názvosloví

Nápad: odlišit kód pro GT v hlavičce funkce. **Pozor, FIKCE!**

```
func akceProGT() global {
    //
}
```

- zavolání `global` funkce ji spustí v GT
- `global` funkce může volat jenom `global` funkci
- přenos do MT lze pouze `DispatchQueue.main.async{}`

Až na to poslední je toto nová funkcionalita `async/await`.

- async/await naplňuje ideál Zásady č. 2.

Jedna stylistická poznámka

"Cesty" v programu (C++).

```
struct FromAppContext {}

//
class SimulatorCehosi {
public:
    //
    void run(FromAppContext &context, args...) {}
}
```

Demo

```
DispatchQueue.global().async {
    // aktivace kodu v GT rezimu
    let result1 = gtDoSomething()

    // asynchronne vstupuji do MT rezimu predat vysledek
    DispatchQueue.main.async {
        //
        let response = communicateMT(result1)

        //
        DispatchQueue.global().async {
            // pokracuju dal
            let result2 = ProcessResponse(result1, response)

            //
            DispatchQueue.main.async {
                //
                Conclusion(result2)
            }
        }
    }
}
```

Demo: důsledky

Řetězení asynchronních volání je tímto způsobem značně frustrující.

- **Knihovna Combine.** To si dáme příště.
- **Async/await.**

Notifikační centrum

Singleton (typicky) pro přeposílání zpráv v pojmenovaných kanálech.

- Pojmenovaný kanál. Má observers.
- Poslání zprávy do kanálu.
- Notifikační centrum zprávu **synchronně rozešle** na observers.

Zásada č.3 (broadcasty v aplikaci) Pokud nemám extra dobrý důvod je posílat synchronně, pak zabalit do `DispatchQueue.main.async`.

Notifikační centrum

Zásada č.3 (broadcasty v aplikaci) Pokud nemám extra dobrý důvod je posílat synchronně, pak zabalit do `DispatchQueue.main.async`.

```
// deje se napriklad v MT
func nejakaHokna() {
    // hokna hokna
    // hokna hokna
    // naplanuju budouci rozeslani zpravy
    DispatchQueue.main.async {
        //
        NotificationCenter.default.Post(...)
    }
    // hokna hokna
    // hokna hokna
}
```

Async/Await metody

Režim je podobný koncepci výjimek:

- metoda musí být označena v hlavičce `async`.
- musí se volat s prefixem `await`.
- volat `async` funkci lze pouze z jiné `async` funkce.

```
Task {
    //
    let result = await volamAsyncFci()

    //
    DispatchQueue.main.async {
        // predej vysledek do aplikace
    }
}
```

Task je ad-hoc async funkce.

Co je async funkce

Je funkce jako každá jiná. Jenom je **určena pro běh mimo MT**.

- ...počítejme tedy s tím, že se vykonává v GT.

```
//
func nejakaAsyncFunc(arg: Cosi) async -> Result {
    //
    return ...;
}
```

Vhodná pro **backend práci**.

Task

```
struct Obrazovka: View {
    // dejme tomu akce na Button(action:), tj MT
    func nejakaAkce() {
        // jde do GT == DispatchQueue.global.async{}
        Task {
            //
            let result = await volamAsyncFci()

            //
            DispatchQueue.main.async {
                // predej vysledek do aplikace
            }
        }
    }

    //
    var body: some View {}
}
```

Zřetězení async volání

Typické pro zpracování síťových komunikací (včetně DB apod)

```
// volej async OP1 s callbackem
OP1(input: ...) { v1 in

    // ...
    OP2(input: v1) { v2 in

        //
        DispatchQueue.main.async {
            // predej v2
        }
    }
}
```

async/await

```
//
func OP1(input: ...) async -> V1 { ... }
func OP2(input: ...) async -> V2 { ... }

//
func ops() async -> V2 {
    // zploštění hierarchie v programu
    let v1 = await OP1(...)
    let v2 = await OP2(v1, ...)
    //
    return v2
}

// je taky async, proto do MT přes GCD
Task {
    let v2 = await ops()
    //
    DispatchQueu.main.async { ... }
}
```

async/await demo

Síťový protokol. Jednotlivé operace jsou REST API dotazy na server. `await` **vytváří iluzi sekvenčního kódu**.

```
func solve(task: Cosi) async -> Result {
    // preruseni kontextu "solve", prepnuti na Task
    // ekvivalent v GCD?
    let submit = await TaskSubmit(task)

    //
    while true {
        // prepnuti kontextu
        let finished = await TaskStatus(submit)

        //
        if finished(...) { break }
    }

    // prepnuti kontextu
    return await TaskDownload(submit)
}
```

async/await paralelizmy

`await` = čekat. Vytváří to sekvenčnost.

```
//
Task {
    // sekvenční kód. Nejprve op1(), pak op2()
    let a = await op1()
    let b = await op2()

    //
    let res = a+b

    //
    DispatchQueue.main.async {
        // predej res
    }
}
```

async/await paralelizmy

`async let` - vzniká `Task` na vyhodnocení výrazu.

```
//
Task {
    // async let !
    async let a = op1()
    async let b = op2()

    // tady se synchronizuji (bariéra)
    let res = await (a+b)

    //
    DispatchQueue.main.async {
        // predej res
    }
}
```

throws async

Komunikační knihovny (DB/sítě/...):

```
func nejakaOperace() throws async -> Result {
    //
}
```

Pak:

```
Task {
    // osetreni: try/throws
    // osetreni: await/async
    if let result = try? await nejakaOperace() {
        //
    }
}
```

Actors

- Jako `class`, referencovatelný typ. Bez dědičnosti.
- Protocol `Actor`.

Actor je objekt, který má od překladače garanci, že dvě vlákna nemůžou současně přistoupit na jeho uloženou property.

```
actor User {
    var score = 10
    // Demo z HackingWithSwift.com
    func printScore() {
        print("My score is \(score)")
    }
    //
    func copyScore(from other: User) async {
        score = await other.score
    }
}
```

```
let actor1 = User()
let actor2 = User()
Task {
    await actor1.printScore()
    print(await actor1.score)
    await actor1.copyScore(from: actor2)
```

Závěr

- Koncepce běhu programu v iOS/macOS.
- `async` je nový hit knihoven SwiftUI/Foundation.

Příště: Knihovna Combine ve SwiftUI. To je jádro architektury SwiftUI aplikací.