

Paralelizmus, synchronizace, GCD

IZA, Martin Hrubý, FIT VUT, 2020

Úvod

- Vlákna, **Grand Central Dispatch**.
- Větší pochopení principu činnosti programu.
- Bezpečnostní pravidla **interakce objektů v aplikacích**.
- Dnes si povídáme o knihovně **Foundation**, tj.
 - ... o principech platných pro všechny platformy Apple.

Historie paralelizmu

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

Paralelní programování a HW

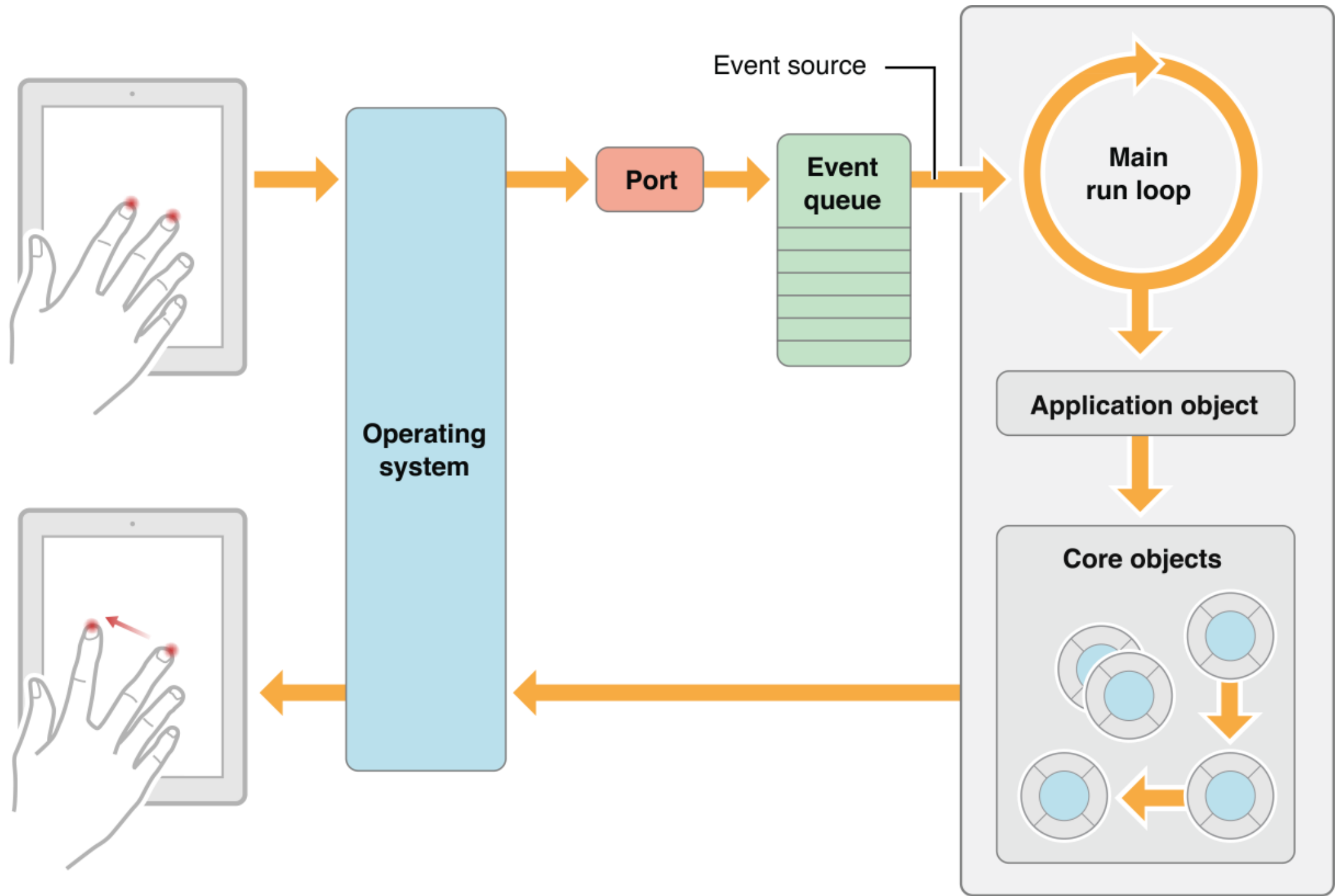
- Máme zařízení s více jádry CPU.
- Program je paralelizovatelný. Přínosy?
- Dokážeme efektivně / komfortně / bezpečně napsat paralelizující program.
- Program je škálovatelný. Je schopen fungovat i jednovláknově.
 - Programování paralelizovatelného programu (C/C++, MPO).
 - GCD a systém bloků kódu.

Co je běh aplikace?

- OS (iOS / macOS) vytváří a spravuje vlákna.
- OS přiřazuje vláknům čas na CPU.
 - 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.
- OS daruje aplikaci 1 MT a několik GT.
 - Main Thread, Global Threads (podle počtu jader CPU).
 - Tato vlákna **"žijí v RunLoop"** aplikace.

RunLoop aplikace

- Jádro aplikace.
- Dostává zprávy od OS (události UI) — fronta.
- Má fronty práce — fronta bloků (*closures*).
- Vlákna si berou bloky z front a vykonávají je.
- Přejít aplikace do pozadí:
 - OS přestane přidělovat čas vláknům aplikace.



RunLoop a UI

- Fronty práce (*MainQueue*, sekvenční, paralelní).
 - Pool vláken — berou si z front bloky práce a vykonávají.
 - *MainThread* bere prioritně z *MainQueue* a obsluhuje události.
- UI fáze:
 - Modifikace stavu UI — atributy, `setNeeds Display / Layout`.
 - Rendering — generování bitmap Views a jejich kompozice.
 - Animace — nějaká změna rozfázovaná v čase.
 - Důsledek: Pro plynulost UI musí MT často projít **RunLoopem**.

UI běží pouze v MainThread

- UI výhradně v MainThread. Proč?
 - UI (kód UIKit + stav aplikace/UI) je v app sdílený prostředek.
 - Jak by vypadalo paralelní provádění UI?
 - Takže můžeme buď zamykat části UI (hromada problémů).
 - ... nebo **rozhodnout UI pouze pro MainThread.**
- Důsledky:
 - RunLoop — dbát o "častou obrátku" v RunLoop.
 - MainThread — velmi krátké činnosti.

Odkud se bere spustitelný kód?

- Aplikace smí vytvářet *vlastní vlákna*.
 - Nežijí v RunLoop. Mají vlastní kód. OS přiděluje čas na CPU.
 - Toto není typické pro iOS aplikace.
 - Pozn.: běh aplikace na pozadí (aplikace není viditelná).
- Typicky aplikace definuje bloky a vkládá je do front (skoro Next-Event). *Escaping closures*.
 - Systém řízení bloků ve frontách se nazývá **Grand Central Dispatch (GCD)**.

```

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 (pozor na MT)
            DispatchQueue.main.async {
                //
                self.delegate?.updateUI(fromThread: self)
            }
        }
    }
}

```

Periodická činnost, Timer

- Timer — registrace jednorázové / periodické akce v RunLoop.
 - reference, selektor, perioda. Aktivace nějaké metody.
 - Nepřehánět to s počtem časovačů v aplikaci (globální časovač rozesílající notifikaci do aplikace).
- AppDelegate — správa časovačů (přechod popředí, pozadí).
- Odložená invokace bloku / metody.

Grand Central Dispatch

- Do fronty X se naplňuje sync / async closure Y .
 - Co se naplňuje, už nejde odplánovat.
 - Closure Y ve frontě není explicitně referencovatelný, tj. nelze mu poslat zprávu (ani není jazykově-syntakticky jak...).
- Běh bloku nelze zastavit / zrušit / ...
 - Vlákno X do něj vstoupí. Až ho dokončí, tak vystoupí.
 - Vlákno X "proteče blokem".

Opakování: 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")
}
```

sync / async dispatch

- `DispatchQueue.frontQueue.async { blok1 }`
 - Vlákno X vykonává `async(blk: @escaping ()->())`,
 - vloží `blok1` do `frontQueue` a z metody vystoupí. LOCK.
 - Odesílatel se nezajímá o další osud bloku.
- `DispatchQueue.frontQueue.sync { blok2 }`
 - ..., ...vloží `blok2` do fronty a
 - pasivuje se, dokud nedostane signál o dokončení `blok2`,
 - pak z metody `sync` vystoupí a pokračuje dál...

Sync / Async

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


Synchronní vkládání

- Vkládající vlákno čeká, tj. je blokováno.

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

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

Fronty GCD a synchronizace

- Neříkám, pro které vlákno to plánuju, ale má to proběhnout:
 - *Sekvenční* (serial) — blok se zahájí po dokončení předchozího bloku z *této fronty*.
 - *Paralelní* (concurrent) — bloky se zahajují podle možností pracujících vláken v pořadí z této fronty.
- `DispatchQueue.main, .global(qos)`
- Uživatelské fronty:
 - `let myQueue = DispatchQueue(label: qos:)`

QoS — Quality of Service

- Je atribut fronty, určuje prioritu bloků ve frontě (pro rozhodování vlákn 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.

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)
    }
}
```

Operation & OperationQueue

- Nad GCD je zbudován systém Operation a Queue.
 - Objektu Operation lze poslat zprávu (je to objekt).
 - Jeho "main" metodu spouští blok GCD, tj. vlákno X proteče jeho metodou "main".
- Operation jsou třídy / objekty. Mají uživatelské uložené properties (data, stav, ...).
 - Mohou do sebe uložit výsledek práce.
 - *CloudKit*. Callbacks.

Řízení Operations, OQ

- Lze definovat precedenci: operace B se spustí až po dokončení operace A.
- `OperationQueue.main` — MQ.
 - Všechny ostatní jsou GT, sekvenční i concurrent.
 - `maxConcurrentOperationCount: Int`; OQ je KVO!
- `OQ.waitUntilAllOperationsAreFinished()`
 - Blokuje současné vlákno!
 - Implementace bariéry. 1) Detekovat okamžik, kdy se práce z fronty dokončila. 2) Blokovat se, dokud se nedokončí.

Bariéra, BlockOperation

```
// uzivatelska fronta. Je typu GlobalQueue.
// sekvencnost/paralelnost urcuje
// myQueue.maxConcurrentOperationCount
let myQueue = OperationQueue();
// operace, kterou chci spustit jako posledni
let finishOp = BlockOperation {
    // prevedu do MT
    DispatchQueue.main.async {
        //
        print("Hotovo")
    }
}
//
for i in 0..<10 {
    // prace v jedne iteraci
    let op = BlockOperation {
        //
        print("Ahoj")
    }
    // precedencd op <- finishOp
    finishOp.addDependency(op)
    //
    myQueue.addOperation(op)
}
//
myQueue.addOperation(finishOp)
```

... shrnutí ...

- Máme GCD bloky a operace.
 - Rozdíl: možnost referencovat kód a jeho data.
- Vkládají se do různých typů front.
- Vlákna z RunLoop je vykonávají.
- Paralelizace. Odlehčení MQ (flexibilnější GUI).
- Problém: interakce mezi běžícími kódy.
 - Povíme si zásady bezpečného programování.

V čem je problém?

- Paralelní běh kódu není problém.
- Problém nastává, když mají objekty (z jiných vláken) *komunikovat*.
 - *Sekvenční kód* — řádek kódu 1, 2, 3, 4, ...
 - *Jedno-vláknový kód* — sada bloků, které se 1vláknově vykonají v nějakém pořadí (aspoň precedence).
 - *Multi-vláknový kód* — sada bloků, které se v čase překrývají.
- Správná synchronizace a současně výlučnost.

Aplikace v single-thread

- Všechny bloky se odehrávají v MT — OK.
 - Překvapit může uživatel. Pořadí akcí vzniká dynamicky.
 - Akce—Reakce. Skoro se to nedá pokazit.
- Asynchronní volání do knihovny.
 - Stav aplikace — před volání, po volání, po obdržení odpovědi.
 - Předpokládáme, že callback z knihovny je v MT.

Aplikace v multi-thread

- Minimalizovat interakce kódu mezi MT a GT.
- Obecné schéma:
 - Vytvořit blok / operaci do GT.
 - Předat kopii všech parametrů (zadání mise).
 - GT akce běží a dokončí. Žádné interakce mimo svůj kontext.
 - **V režimu MT odešle výsledek** — delegate, Notification.
 - Je to zcela bez nebezpečných interakcí?
- Zdravá "nedůvěra" mezi objekty. Rozhraní.

Demo, GTWorker

```
//  
protocol GTWorkDelegate : AnyObject {  
    //  
    func workDone(from: GTWork)  
    func workDone(from: GTWork, andResult: String)  
}  
// objekt bude zit v kontextu GT  
class GTWork : Operation {  
    //  
    weak var delegate: GTWorkDelegate?  
    // tady bude cekat vysledek prace operace  
    var result: String = ""  
    //  
    override func main() {  
        // delam nejakou praci  
        // mam hotovo  
        DispatchQueue.main.async {  
            // delegat si bude brat vysledek v dobe  
            // kdy uz tato operace bude ne-bezici  
            self.delegate?.workDone(from: self)  
        }  
    }  
}
```

Demo, GTUser

```
// prevazne zije v kontextu MT
class GTUser : GTWorkDelegate {
  // pripadny spusteny vypocet
  var _gtWorkRunning: GTWork?
  let _opQueue = OperationQueue()
  // MT
  func runGT() {
    // explicitne registruju zmenu stavu pred/po spusteni GTwork
    guard _gtWorkRunning == nil else { return }
    // vytvoreni a predani VSECH relevantnich vstupu
    _gtWorkRunning = GTWork()
    _gtWorkRunning!.delegate = self
    // spusteni
    _opQueue.addOperation(_gtWorkRunning!)
  }
  // zasada: dostavam tuto zpravu v MT
  func workDone(from: GTWork) {
    // je to zprava od objektu, kterou ocekavam?
    if let __gtrun = _gtWorkRunning, __gtrun == from {
      // beru vysledky
      let result = __gtrun.result
      // menim stav objektu
      _gtWorkRunning = nil
    }
  }
}
```

GTDemo, rozbor

- Pohled GTWorker:

- "Můj vlastník mě stvořil neaktivního a vložil do mě zadání. Vložení do OperationQueue nade mnou dočasně ztrácí kontrolu. Až dokončím, pošlu mu v MQ async zprávu. Vlastní mě, takže si ze mě vezme výsledky."
- Rozbor vlastnictví v průběhu života GTWorkeru.

- Pohled GTUser:

- "Někdo invokoval workDone. Předpokládám je neaktivní. Ověřím nejprve jeho totožnost."

GTDemo, stručněji v GCD

```
//
class GTUser {
    // stav[12] jsou proměnlivé!!!
    var stav1 = 1
    var stav2 = 1234
    var _gtRunning = false

    // voláno v MT
    func someWork() {
        //
        guard _gtRunning == false else { return }
        // vzorkuju stav zadání
        let _stav1 = stav1
        let _stav2 = stav2
        // měním vnitřní stav
        _gtRunning = true
        // plánuju práci do GT
        DispatchQueue.global().async {
            // práce, výpočet nad _stav1, _stav2
            let result = _stav1 + _stav2
            //
            DispatchQueue.main.async {
                // beru si výsledek result
                self._gtRunning = false
            }
        }
    }
}
```

```

override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath) ->
UITableViewCell
{
    let cell = tableView.dequeueReusableCell(withIdentifier: "tabik", for:
indexPath)

    cell?.imageView?.image = placeholder;
    // do vedlejsiho/globalniho vlakna naplanuju...
    DispatchQueue.global().async {
        // sestaveni obsahu pro bunku tabulky
        if let loaded = myModel.load(cosi) {
            // pokud mame obsah, jdeme zpatky do mainThread
            DispatchQueue.main.async {
                // pokud na indexPath stale existuje bunka
                if let ncell = tableView.cellForRow(at: indexPath) {
                    // aktualizuji
                    ncell.imageView?.image = loaded;
                }
            }
        }
    }
    //
    return cell;
}

```


Zásada č. 1

- MT / MQ je nejpřirozenější forma řešení výlučnosti a synchronizace.
 - ... protože je sekvenční, tj. MT ví, že se nemůže nic dalšího z MQ spustit.
 - blok kódu v režimu MT nemusí nic zamykat, pokud může věřit solidnosti kódu běžícím v GT.
- Zásada č. 1: veškerou interakci mezi objekty provádět v režimu MT / MQ.

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.

Zásada č. 2 — odlište v kódu

- Metody tříd:
 - Implicitně jsou pro běh v MT.
 - Metody pro GT jsou viditelně pojmenovány "gtDoSomething".
 - ... něco jako @escaping, akorát nad metodami.
- gt* metody drží Zásadu č. 1, tj. "hledí si svého kódu a svých interních dat".
 - gt* metody volají pouze gt* metody.
 - Do okolního světa vstupují výhradně v MT režimu.

Najděte tam tu chybu

```
DispatchQueue.global().async {
    // aktivace kodu v GT rezimu
    let result1 = gtDoSomething()
    // synchronne vstupuji do MT rezimu
    // predat vysledek
    DispatchQueue.main.sync {
        //
        let response = communicateMT(result1)
    }
    // pokračuju dál
    let result2 = gtProcessResponse(result1, response)
    //
    DispatchQueue.main.async {
        //
        Conclusion(result2)
    }
}
```

• • •

```
DispatchQueue.global().async {
    // aktivace kodu v GT rezimu
    let result1 = gtDoSomething()
    // synchronne vstupuji do MT rezimu
    // predat vysledek
    DispatchQueue.main.async {
        //
        let response = communicateMT(result1)
        //
        DispatchQueue.global().async {
            // pokračuju dál
            let result2 = ProcessResponse(result1, response)
            //
            DispatchQueue.main.async {
                //
                Conclusion(result2)
            }
        }
    }
}
```

Notifikační centrum

- Singleton pro NCentrum.
- Registruji objekt A jako observer zprávy MSG.
- Objekt B ve vlákně X posílá sync přes NCentrum zprávu MSG.
 - NCentrum sync rozesílá (for-cyklus) MSG všem observerům.
 - ... stále vykonává vlákno X.
 - Observer zpracovává přijetí MSG ve vlákně X.
 - Pokud $X=GT$, pak se do MT-kódu můžu dostat GT-vláknem.

Zásada č. 3: Notifikační centrum

- Notifikace je zpráva pro neznámý počet adresátů.
- Mám důvod ji posílat synchronně? Čekat na jejich obsluhu?
 - Typické ne, pak posílat *.main.async { zprava(); }*
 - Odesílatel nezná podstatu obsluhu příjemce.
 - Příjemce nezná vlákno odesílatele (obecně...).
- Notifikace posílat MT async.

Komunikace objektů: properties

- Smí se objekt A hrabat v properties objektu B?
 - B.prop = "ahoj" — co se stane?
- Typicky je tam setter / getter. Spouští se nějaký kód.
 - setter, KVO — provolá se neznámá sada observerů. GT/MT?
 - setter, CoreData — B sync volá MOC, ten generuje Notification, na ni reaguje FRC, sync volá delegáta, tabulka.
 - getter, CoreData — B sync volá MOC, disková operace, Notifikace, ...

KVO, @objc, @dynamic, CD obj

- Berme přístup na property vždy jako aktivaci nějakého kódu. Kaskádově dále.

```
@implementation TRIDAB
/* @synthesize numberb = _numberb; */

-(int) numberb {
    // willAccess, didAccess...
    return _numberb;
}

-(void) setNumberb:(int)v {
    [self willChangeValueForKey: @"numberb"];
    _numberb = v;
    [self didChangeValueForKey: @"numberb"];
}
@end
```

Zásada č. 4: properties

- Přístup na properties objektů pouze v MT.
 - Přirozená serializace činností. Bloky považujeme za atomické.
- GT nikdy nezapisuje properties mimo svůj kontext.
- Zámky. Zamykaná property: důsledně vždy každý přístup.
 - Stačí zamykat jenom getter / setter?
 - Instrukce procesoru Test&Set.

NSLock

```
//  
class GTWorker {  
    //  
    func gtWork(on: GTUser) {  
        //  
        on.__lPropertyLock.lock()  
        on.lProperty = 10  
        on.__lPropertyLock.unlock()  
    }  
}  
//  
class GTUser {  
    //  
    var lProperty: Int = 0  
    var __lPropertyLock = NSLock()  
    //  
    func someWork() {  
        __lPropertyLock.lock()  
        // vypočet 1 nad lProperty  
        let op = lProperty  
        // vypočet 2 nad lProperty  
        let op2 = lProperty  
        //  
        __lPropertyLock.unlock()  
    }  
}
```

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?

Třídni 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.

Suspendování Thread — kvazi

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

    //
    var myLock = NSLock()

    //
    func pauseMe() { myLock.lock() }
    func unpaueMe() { 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()
        }
    }
}
```

Příště

- Kódování dat.
- Key-Value Coding.
- Documents. iCloud Doc.
- CoreData.