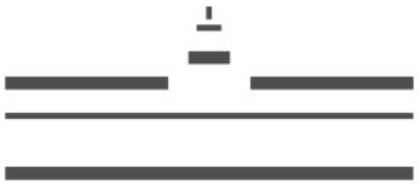


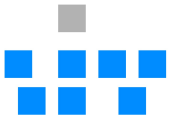
Westfälische
Wilhelms-Universität
Münster

Programmierung für mobile Endgeräte

Nebenläufigkeit

wissen.leben
WWU Münster

Name: Patrick Förster



ZENTRUM FÜR
INFORMATIONS
VERARBEITUNG

Überblick

- Nebenläufigkeit beschreibt die Fähigkeit eines Programms mehrere Aufgaben (Tasks) gleichzeitig auszuführen
- Für ein Betriebssystem bspw. unabkömmlich
- Mit der steigenden Anzahl an CPU-Kernen ist Nebenläufigkeit mehr den je ein wichtiges Thema
- Die größte Anzahl an Kernen bring nichts, wenn ein Programm nicht, wie es diese ausnutzen kann
- In den meisten Programmiersprachen werden „Threads“ als Mittel zur Nebenläufigkeit eingesetzt
- Zu meist ist es Aufgabe des Programmierers sich, um den Ablauf von Threads zu kümmern
- Auch wenn es in den meisten Programmiersprachen verschiedene Tools zum Management von Threads gibt (z.B. Java Concurrency), wird Multi-Threading schnell komplex

Threads in Objective C: NSThread

- Die Klasse NSThread ist die Basisklasse für Nebenläufigkeit mit Threads in Objektive C
- Jede Methode eines jeden Objekts kann als Einstieg für eine nebenläufige Aufgabe genutzt werden:

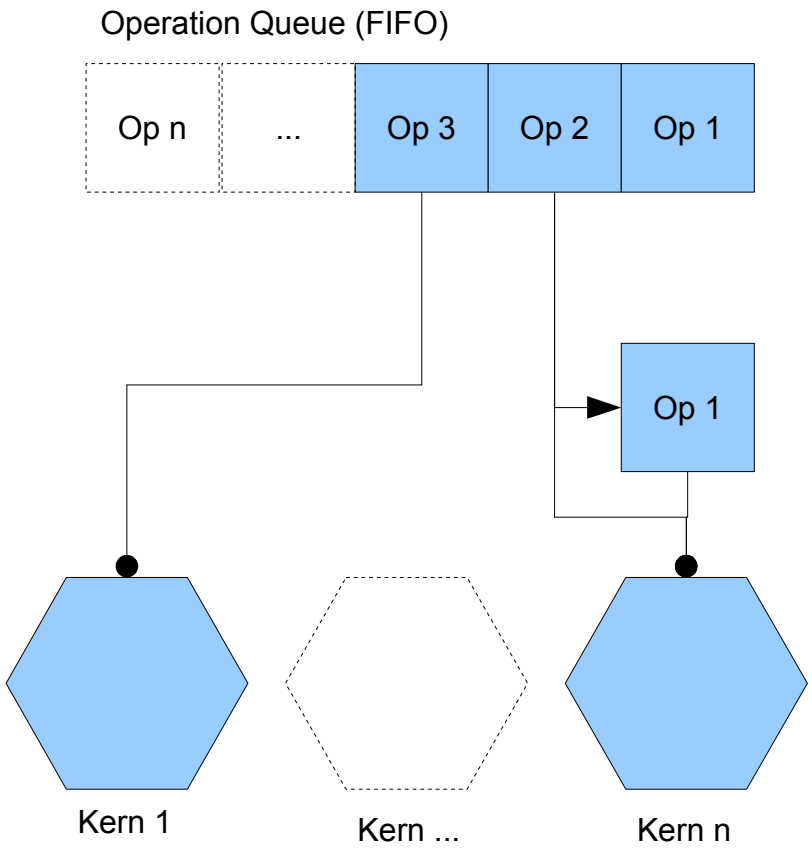
```
NSThread* thread = [[NSThread alloc] initWithTarget:TARGET selector:@selector(METHOD:(PARAM2...)) object:OBJECT];

[thread start];

-(void) METHOD: (id) param1 PARAM2: (id) param2 ... {
    ...
}
```

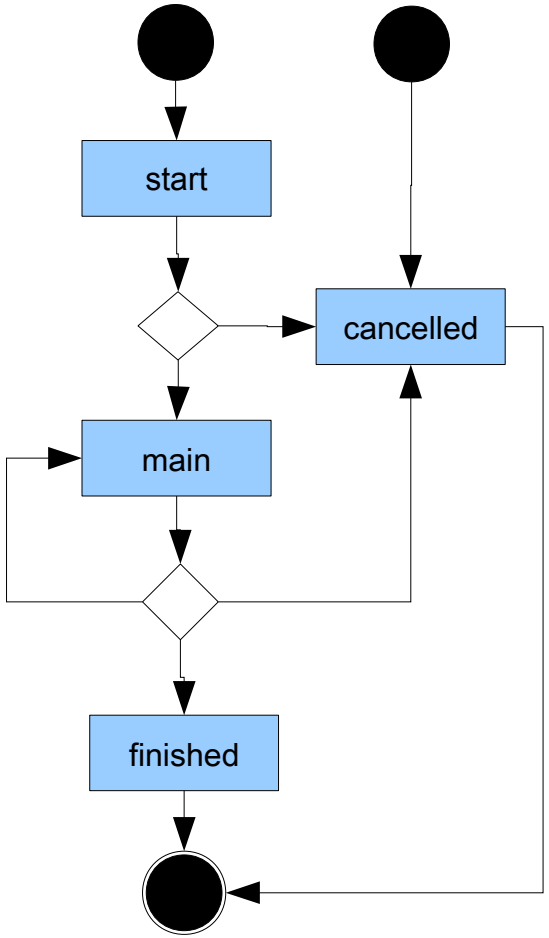
- Der Status eines Threads kann mit den Methoden isExecuting, isFinished oder isCancelled überprüft werden

Grand Central Dispatch (GCD)

- Threads sind ein geeignetes Mittel für Nebenläufigkeit, skalieren allerdings nicht gut mit der Anzahl der Prozessorkerne
- #Threads == #Kerne bedeutet nicht, dass das alle Kerne wirklich ausgelastet werden!
- Apple hat mit dem „Grand Central Dispatch“ (GCD) eine Technologie eingeführt, die zum einen Aufgaben auf Systemlevel an die verfügbaren Kerne verteilt, um diese optimal auszulasten und zum anderen für den Programmier nahezu gänzlich transparent einsetzbar ist
- Der Programmierer muss sich nicht um einen Ablaufplan für Aufgaben kümmern, sondern übergibt diese einfach an sogenannte „Dispatch-Warteschlangen“, die dann alle nötigen Vorgänge übernehmen
- Eine spezielle Art von Warteschlangen sind die Operation-Queues, die Aufgaben als Operationen definiert nebenläufig ausführen

Operationen allgemein





NSOperation

- Die Klasse NSInvocationOperation kann analog zur NSThread-Klasse dazu genutzt werden eine beliebige Methode eines Objektes als Einstiegspunkt für eine nebenläufige Aufgabe zu nutzen:

```
[[NSInvocationOperation alloc] initWithTarget:OBJECT selector:@selector(METHODI:PARAM:) object:OBJECT];
```

- Mit Hilfe der Klasse NSBlockOperation kann ein Block-Objekt als Einstiegspunkt deklariert werden

```
[NSBlockOperation blockOperationWithBlock:^(void) {/* tue etwas  */}];
```

- Warteschlangen können ebenso einfach erzeugt werden:

```
NSOperationQueue queue = [[NSOperationQueue alloc] init];
```

- Damit eine Operation nebenläufig aufgerufen wird, genügt es diese der Warteschlange hinzuzufügen

```
NSOperation* op = ...
[queue addOperation:op];
```

- Wie schon bei Threads ist es nicht garantiert, dass eine Erhöhung der Warteschlangen auch eine kürzere Ausführzeit mit sich bringt!

Nutzerdefinierte Operationen

- Falls die vordefinierten Klassen nicht genügen, kann NSOperation natürlich auch durch eine eigene Klasse erweitert werden:

```
@interface MyOperation : NSOperation ...

@implementation MyOperation

-(id) initWith...: {
    if ((self = [super init])) {
        ...
    }
    return self;
}

-(void) main {
        // …
}
@end
```

- Achtung: Operationen können mit start auch „per Hand“ ausgeführt werden, allerdings sind sie per Default dann nicht nebenläufig!

Nutzerdefinierte Operationen (II)

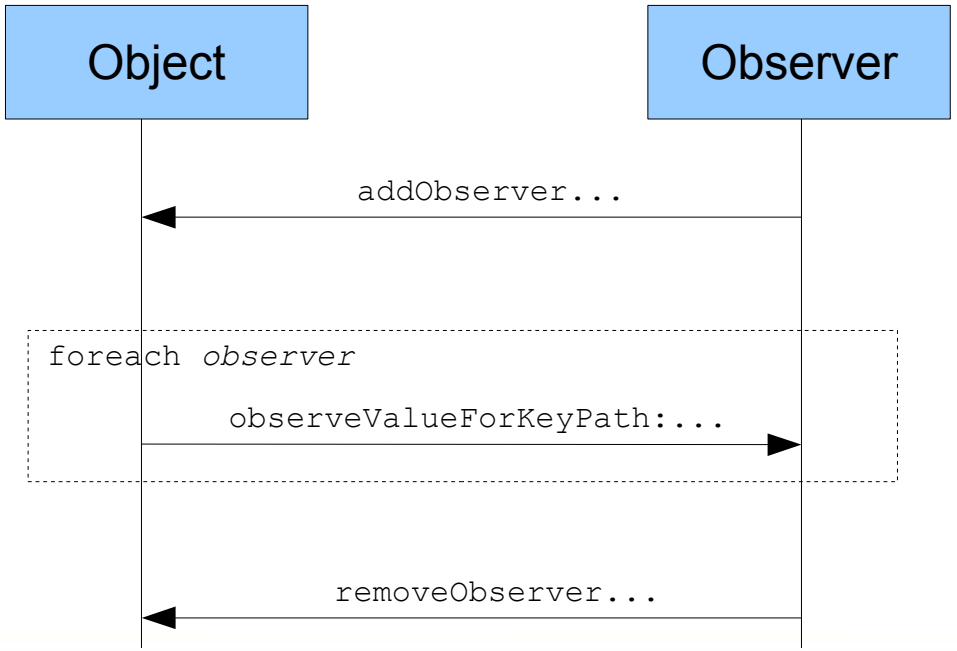
- Das Aktivitätsdiagramm für Operationen zeigt, dass eine Operation abgebrochen werden kann, noch bevor die main-Methode aufgerufen wurde
- Daher muss überprüft werden, ob eine Operation abgebrochen wurde:

```
-(void) main {
    if ([self isCancelled]) return ;

   //...
}
```

- Falls die main-Methode eine periodische wiederkehrende Aufgabe erfüllt, sollte ebenso periodisch der Status geprüft werden
- Falls eine eigene Operation in einem Projekt definiert werden, dass kein ARC unterstützt, sollte die Operation ihren eigenen Autorelease-Pool erzeugen
- Möchte man, dass eigene Operationen auch ohne Warteschlange nebenläufig ausgeführt werden. so muss eigenständig dafür gesorgt werden (z.B. NSThread)
- Dazu müssen vor allem die Status-Methoden isExecuting, isFinished und isConcurrent entsprechend implementiert werden

Key-Value Observing

- Operationen sind Key-Value-Observing (KVO) konform
- KVO ist ein Mechanismus über den Objekte über Änderungen innerhalb eines anderen Objektes informiert werden können
- Wie der Name schon sagt, folgt der Mechanismus dem Observer-Pattern:
- Jede von NSObject erbende Klasse beherrscht die nötigen Methoden für KVO
- Die zur Beobachtung verfügbaren Attribute werden über einen „Key“ identifiziert, für den sich ein Beobachter registrieren kann
- Bei jeder Änderung wird jeder Observer automatisch darüber informiert



Key-Value Observing (Beispiel)

- Per Default ist jede Property KVO-konform
- D.h. sobald der Wert eines Attributes über die zugehörige Setter-Methode geändert wurde, werden die Observer benachrichtigt

```
@implementation KVOObject

@synthesize status;

-(void) start {self.status = @"Started";}
-(void) finish {self.status = @"Finished";}
-(void) cancel {self.status = @"Cancelled";}

@end
```

```
-(void) run {
    KVOObject* object = [KVOObject new];

    [object addObserver:self forKeyPath:@"status"
                options:(NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld)
                context:nil];

    [object start];

    if (arc4random() % 2 == 0) [object finish];  else [object cancel];
}

-(void) observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:(NSDictionary *)change context:(void *)context {
    NSLog(@"Observer '%@' of object %@", keyPath, object);
    NSLog(@"Changes: %@", change);
}
```

Key-Value Observing (Beispiel II)

- Ausgabe:

```
Console[1776:60b] Observe 'status' of object <KVOObject: 0x102800170>
Console[1776:60b] Changes: {
    kind = 1;
    new = Started;
    old = Ready;
}
Console[1776:60b] Observe 'status' of object <KVOObject: 0x102800170>
Console[1776:60b] Changes: {
    kind = 1;
    new = Finished;
    old = Started;
}
```

- Die Bitmask-Option NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld veranlasst das PGKVODemo bei Änderungen sowohl den alten als auch den neuen Wert mitliefert
- Der Werte werden dann über die Schlüssel „*new*“ und „*old*“ in einem NSDictionary an den Observer übermittelt

KVO für nutzerdefinierte-Operationen

- Selbstdefinierte Operationen sollten KVO-konform bleiben und müssen daher – so denn die zugehörige Getter-Methoden überschrieben werden, folgende KVO-Schlüssel unterstützen:
  - isCancelled
  - isConcurrent
  - isExecuting
  - isFinished
  - isReady
  - queuePriority
  - completionBlock
- Da den Methoden keine Property zugeordnet sein muss, muss das Objekt per Hand über Änderungen informiert werden:

```
[self willChangeValueForKey:@"isFinished"];
finished = YES;
[self didChangeValueForKey:@"isFinished"];
```

Abhängigkeiten & Completion Block

- Einer Operation können beliebig viele weitere Operationen als Abhängigkeit hinzugefügt werden
- Erst wenn diese Operationen abgearbeitet oder abgebrochen wurden, ist die abhängige Operation einsatzfähig (KVO: isReady)

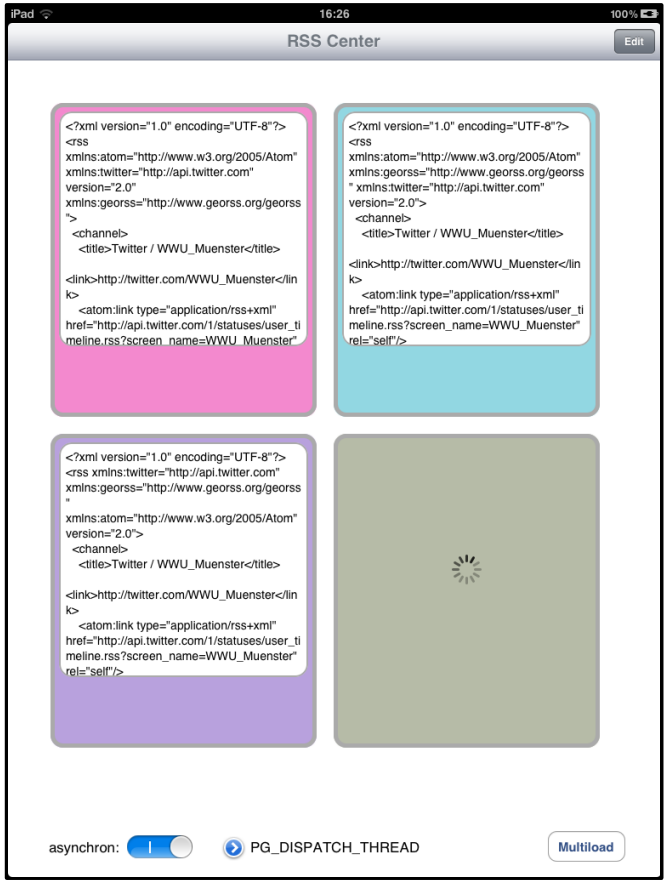
```
A = [[MyOperation alloc] init];
B = [[MyDependentOperation alloc] init];

[B addDependency:A];

[queue addOperation:A];
[queue addOperation:B];
```

- Abhängigkeiten sollten vor dem Hinzufügen zu einer Warteschlange festgelegt werden
- Ansonsten wurde die abhängige Operation unter Umständen schon gestartet
- Außerdem kann jeder Operation ein „Completion-Block“ - also ein Block-Objekt – hinzugefügt werden (KVO:completionBlock) , welches ausgeführt wird, sobald die Operation abgearbeitet wurde

```
[op setCompletionBlock:^(void) {
        //..
}
```

Demo-Anwendung

- In Apps kann Nebenläufigkeit sehr wichtig sein, um das Frontend nicht zu blockieren, während bestimmte unter Umständen zeitaufwendige Aufgaben ausgeführt werden
- In einer Beispielanwendung sollen die verschiedenen Arten von Operationen/Threads genutzt werden, um RSS-Daten aus dem Netz zu laden
- Dazu wird der bereits geschriebene Kachel-View aus der Vorlesung „Gestenerkennung & Animation“ genutzt, um verschiedene RSS-Feeds gleichzeitig zu laden und anzuzeigen



Demo-Anwendung (II)

- Am Boden der Applikation kann der Nutzer über einen Switch festlegen, ob er RSS synchron oder asynchron laden möchte
- Wechselt der Nutzer auf asynchron, so werden ihm per Popover-Tabelle vier Möglichkeiten zum asynchronen Laden angeboten:
  - **PG_DISPATCH_THREAD**: Pro Ladevorgang wird ein NSThread erstellt und gestartet
  - **PG_DISPATCH_INVOCATION**: Pro Ladevorgang wird eine NSInvocationOperation erzeugt und gestartet
  - **PG_DISPATCH_BLOCK**: Pro Ladevorgang wird eine NSBlockOperation erzeugt und gestartet
  - **PG_DISPATCH_CUSTOM**: Pro Ladevorgang wird eine nutzerdefinierte Operation des Typs PGLoadOperation erzeugt und gestartet
- Alle Operationen werden der gleichen Warteschlange hinzugefügt
- Insgesamt existiert nur eine einzige Warteschlange

Demo-Anwendung (URL laden)

- Jeder Aufruf – egal ob per NSOperation oder NSThread – ruft zum Laden einer URL die Methode loadUrl: auf, die als statische Methode in der Klasse PGUrlConnector gekapselt ist

```
+(NSString*) loadUrl: (NSURL*) url {
    NSError* error = nil;
    NSString* result = [NSString stringWithContentsOfURL:url encoding: NSASCIIStringEncoding error:&error];

    if (error) {
        NSLog(@"Error = %@", error);
        result = error.localizedDescription;
    }

    [NSThread sleepForTimeInterval:abs(arc4random()%4) + 1];

    return result;
}
```

- Damit die Nebenläufigkeit auch wirklich sichtbar wird, wird hier per sleepForTimeInterval noch ein bisschen geschlafen, ehe der Ladevorgang abschließend den Html-Content der URL als String zurückliefert

Demo-Anwendung (Main-Thread vs. Custom Thread)

- Achtung: Wie bspw. in Java Swing ist es auch in Cocoa Touch nicht erlaubt Änderungen am User-Interface – wie etwas das Hinzufügen neue Buttons – in einem nebenläufigen Thread durchzuführen
- Für das UI ist einzig und allein der Haupt-Thread zuständig
- Alles andere wird mit einer Exception quitiert:

```
Tried to obtain the web lock from a thread other than the main thread or the web
thread. This may be a result of calling to UIKit from a secondary thread. Crashing
now...
```

- Um innerhalb eines nebenläufigen Threads, eine Methode auf einem Objekt innerhalb des Main-Threads aufzurufen, verfügt jedes NSObject über die Methode: performSelectorOnMainThread:withObject:waitUntilDone
- Diese Methode wird am Ende eines jeden Ladevorgangs dazu genutzt, dessen Ergebnis an den zuständigen ViewController zu delegieren, der dann evtl. Änderungen an der UI vornimmt:

```
[_controller performSelectorOnMainThread:@selector(onDoneLoading:)
                                withObject:((PGLoadOperation*)op).result waitUntilDone:YES];
```

Demo-Anwendung (PGLoad- & PGParseOperation)

- Neben der nutzerdefinierten Operation PGLoadOperation existiert mit PGParseOperation noch eine weitere Operation, die abhängig von einem Ladevorgang (**PG_DISPATCH_CUSTOM**) das Ergebnis mittels TouchXML (FreeBSD License, https://github.com/TouchCode/TouchXML) parst
- Dazu wird die String-Eingabe in einen Array von NSMutableDictionary-Objekten umgewandelt, die ihrerseits jeweils einem Eintrag der Form <item>...</item> aus dem RSS-Feed entsprechen
- Die PGParseOperation verfügt zudem noch über einen „Completion-Block“, der den geparsten Array an den Haupt-Thread delegiert, wo dieser dann verarbeitet wird:

```
-(void) onDoneParsing: (NSArray*) array {
    NSUInteger index = 1;
    NSMutableString* string = [[NSMutableString alloc] init];
    for (NSDictionary* dict in array) {
        [string appendFormat:@"Eintrag %i:\n", index++]; [string appendString: [dict valueForKey:@"title"]];
    }
    [self onDoneLoading:string];
}
```