Blocks
Grand Central Dispatch

Andreas Monitzer
2010-01-14

Übersicht

- Blocks
- Grand Central Dispatch
- Grand Central Dispatch mit Cocoa

Blocks

- Erweiterung von
  C, Objective-C, C++
- Ersatz für Delegate-Konzept
- aus Smalltalk
- aus anderen Sprachen bekannt als
  - First Order Function
  - Closures
  - Lambda-Expression
  - …

- C-Funktion:

```
int abcdef(int a, void *b) {
   // ...
}
```

- Block:

```
^(int a, void *b) {
   // ...
}
```

Was können Blocks?

- Blocks können Variablen zugewiesen werden:

```
#include <stdio.h>

int main(int argc, char **argv) {
   void(^variable)(int, void*);

   variable = ^(int a, void *b) {
     printf("%d\n", a);
   };

   variable(1,NULL);
   variable(2,NULL);
}
```

- Blocks können als Parameter übergeben werden:

```
#include <stdio.h>

void operate(void(^param)(int)) {
  param(1);
  param(2);
}

int main(int argc, char **argv) {

  operate(^(int a) {
    printf("%d\n", a);
  });
}
```

- Blocks können auf äußere Variablen zugreifen:

```
#include <stdio.h>

void operate(void(^param)(void)) {
   param();
}

int main(int argc, char **argv) {
   int var = 42;
   operate(^{
     printf("var = %d\n", var);
   });
}
```

- Blocks können äußere Variablen verändern (__block!):

```
#include <stdio.h>

void operate(void(^param)(int)) {
   param(1);
   param(2);
}

int main(int argc, char **argv) {
   __block int sum = 0;

   operate(^(int a) {
     sum += a;
   });
   printf("sum = %d\n", sum);
}
```

- Blocks können kopiert werden
  - Block_copy()
- Blocks können freigegeben werden
  - Block_release()

- selten notwendig!

Integration mit Objective-C

- Blocks verstehen
  -copy, -retain, -release, -autorelease
- die einzigen Objekte, die auch am Stack liegen können!
- self wird implizit in den Block kopiert

- nur der Pointer wird kopiert, nicht die Objekte!

```
#import <Foundation/Foundation.h>

int main(int argc, char **argv) {
    NSMutableString *str = [@"abc" mutableCopy];

    void(^abc)(void) = ^{
      [str appendString:@"def"];
    };
    abc();
    abc();
    NSLog(@"%@", str);
}
```

Integration in Cocoa

- neue Iterationsmethoden:

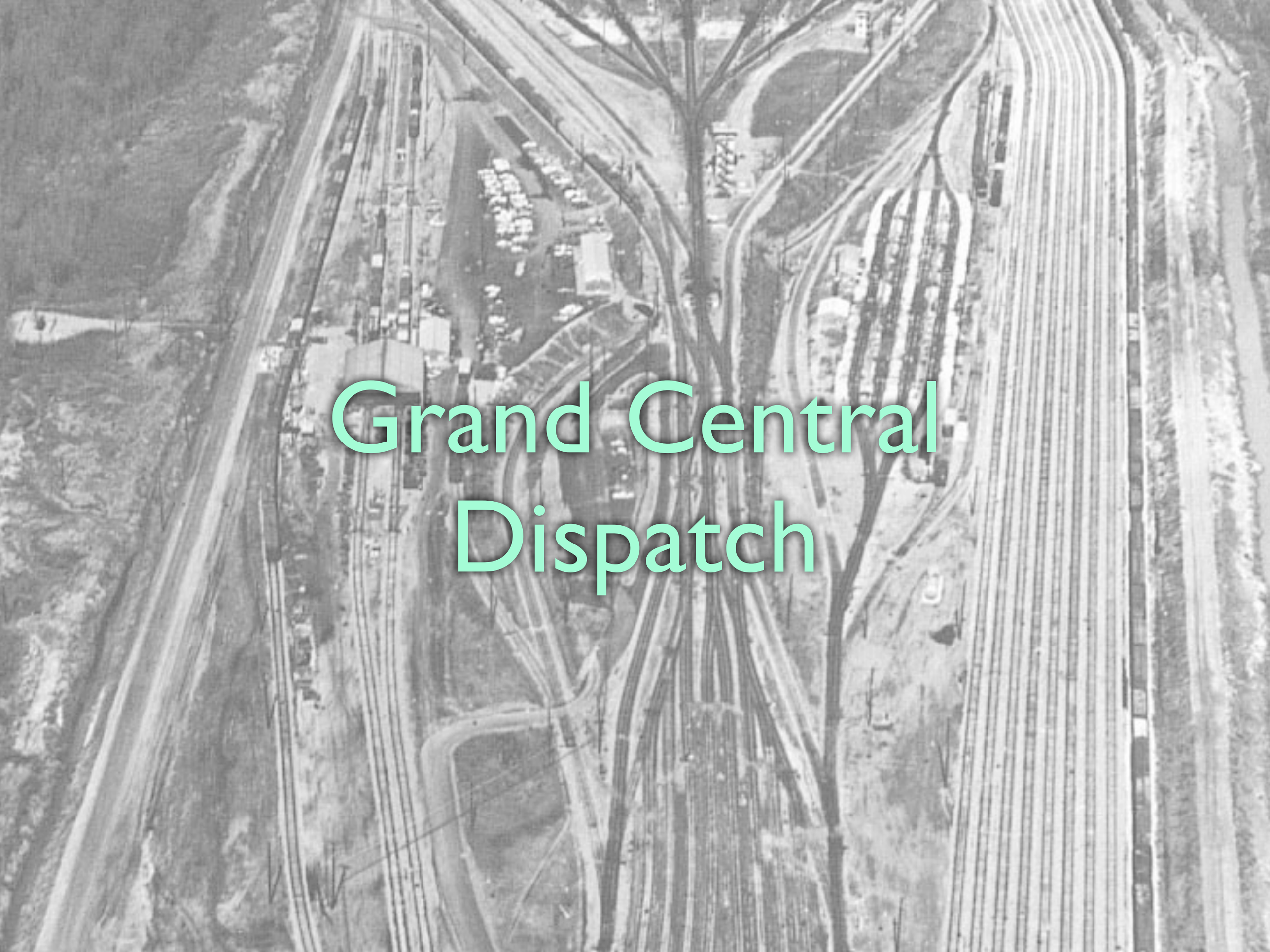
```
[array enumerateObjectsUsingBlock:
 ^(id obj, NSUInteger idx, BOOL *stop) {
   // ...
 }];

[dict enumerateKeysAndObjectsUsingBlock:
 ^(id key, id obj, BOOL *stop) {
   // ...
 }];
```

- neue Sortiermethode (NSMutableArray):

```
[array sortUsingComparator:
 ^(id obj1, id obj2) {
   return NSOrderedSame;
 }];
```

- und vieles, vieles mehr…
  - noch nicht überall vorhanden, kommt (hoffentlich) noch



Grand Central Dispatch

Gibt es ein Problem?

- heutige Computer sind sehr schnell
- mein Handy ist viel schneller als die Supercomputer für die Mondlandung
- ein Großteil der Zeit wird auf Benutzereingaben gewartet

aber:

- das menschliche Gehirn nimmt Latenzen ab 10-15ms wahr*
    - Überschreitung dieser Antwortzeiten führt zu zähem Bedienungskomfort

* Unterschiedlich von Person zu Person

Das User Interface muss schnell reagieren!

(egal ob es bereits die Antwort hat oder nicht)

Wo macht Optimierung Sinn?

- wenn die Antwortzeit zu hoch ist
- wenn Echtzeit gefragt ist (Spiele, …)
- wenn Operationen wirklich lange dauern

Wie optimiert man?

- vorläufige Antwort an den User, tatsächliche Antwort im Hintergrund berechnen (asynchron)
- unnötige Operationen entfernen
- bessere Algorithmen (selten!)
- Auslagerung (Cloud Computing, GPU)
- Parallelisierung

Wie optimiert man?

- Parallelisierung

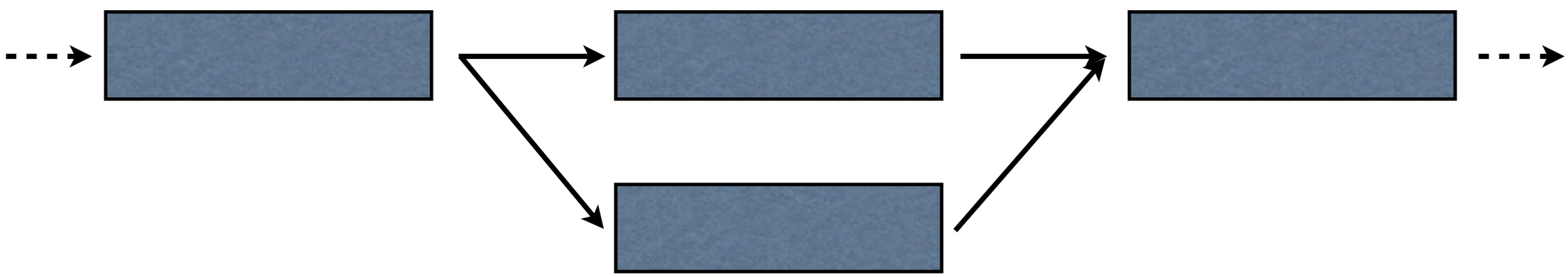
Traditionelle Programmierweise

Arbeitspaket

Threads

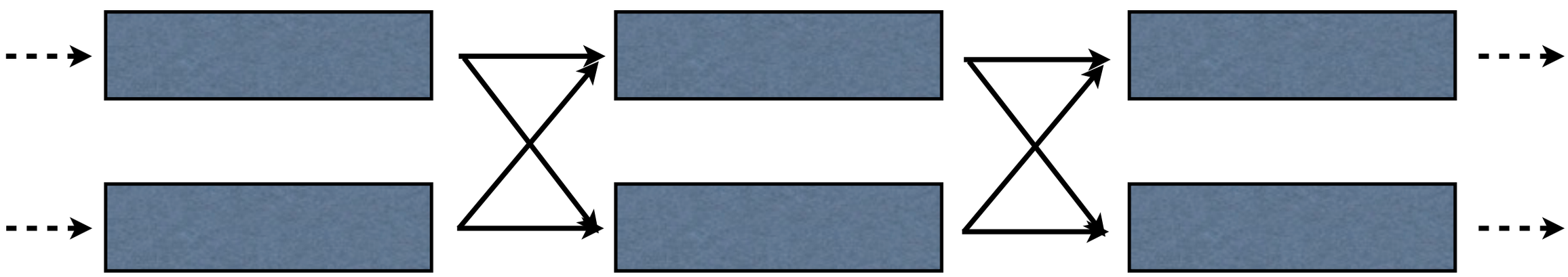
- werden dynamisch vom Kernel auf die CPUs aufgeteilt
- Programmierer kümmert sich um Datenabhängigkeiten und Synchronisation
  - Semaphoren, “Locks”

Threads (2)

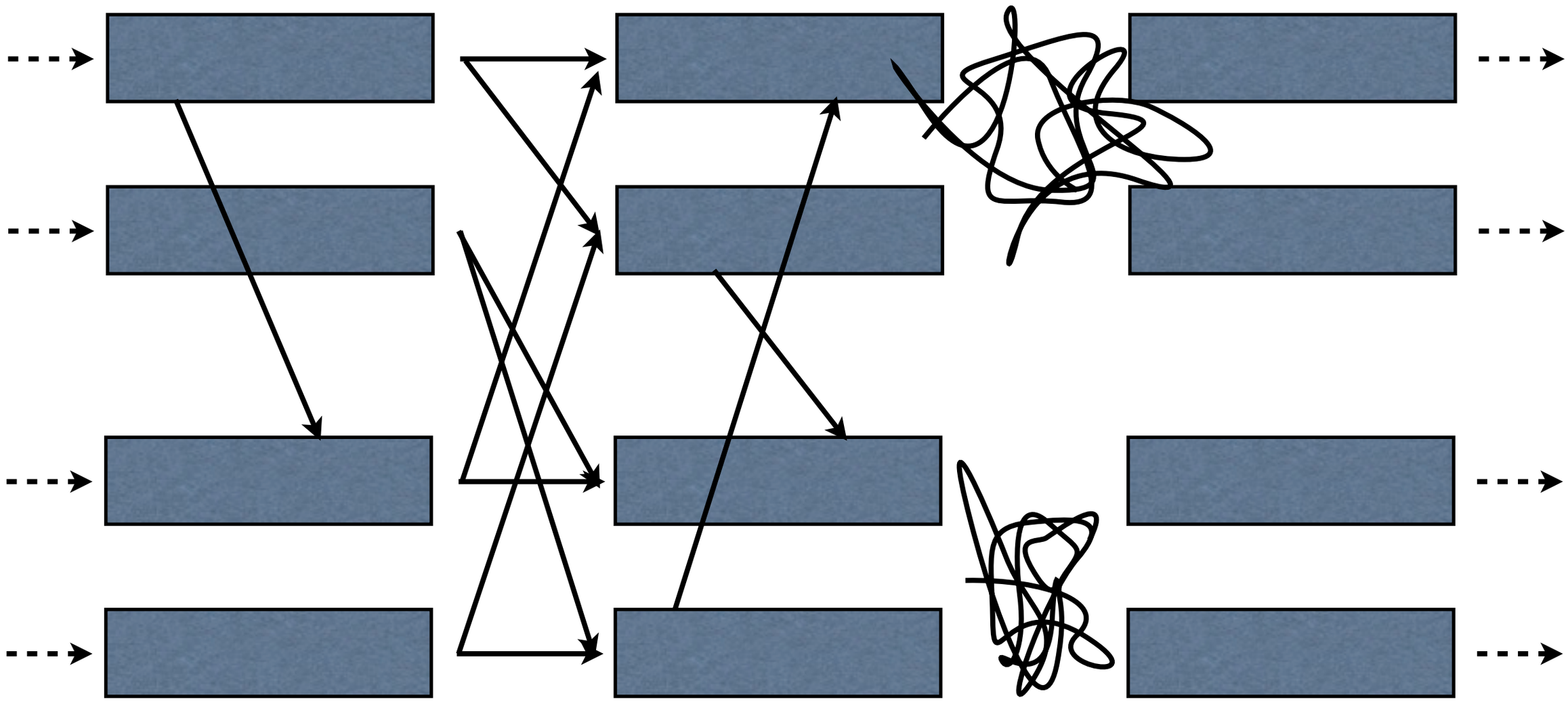


Threads (3)

Threads (4)

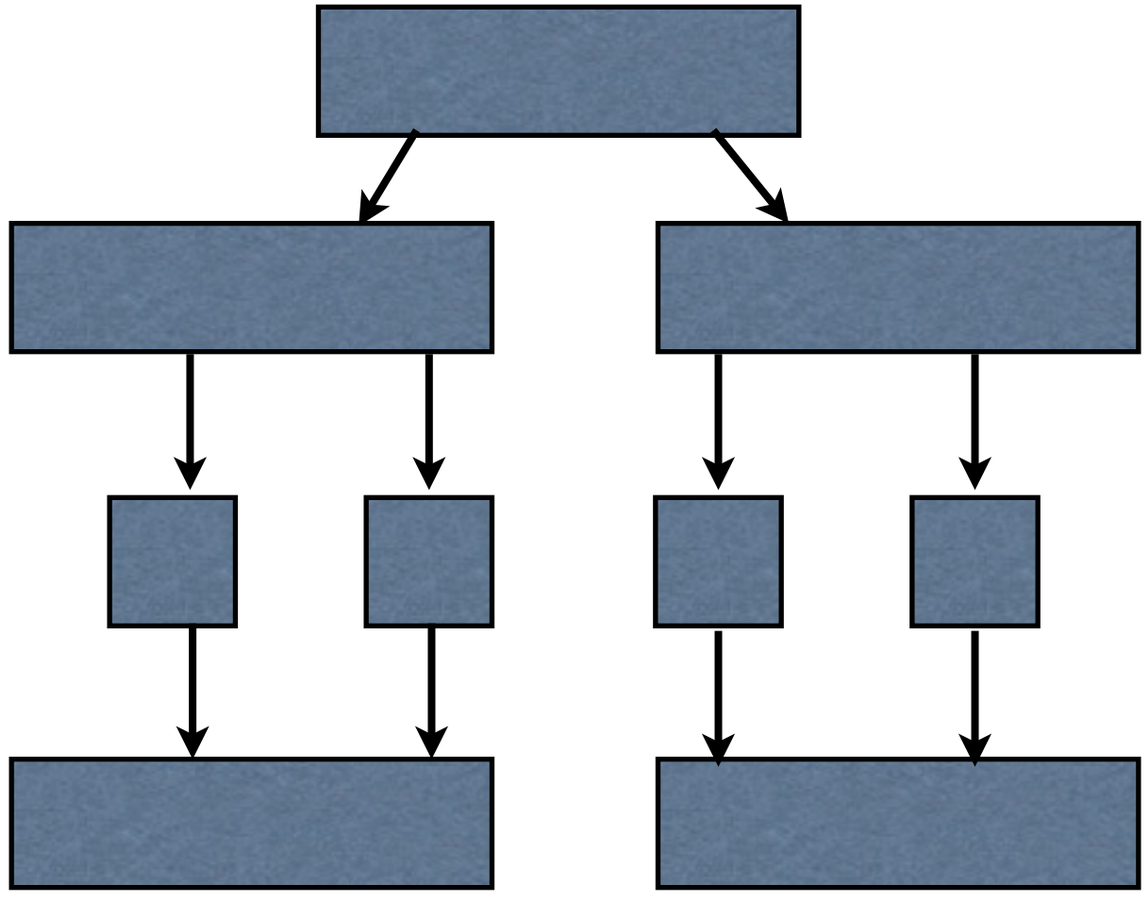


Threads (5)



“Neues” Konzept

- Datenpipeline
- Abhängigkeitsgraph (DAG)



Lockfrei!

Grand Central Dispatch

- automatische Verteilung von Arbeitspaketen auf gerade vorhandene CPUs
- “libdispatch”
  - C, C++, Objective C
  - kein Exception handling!

Motivationen

- Architekturen werden breiter, nicht schneller (mehr CPUs)
- Computer sind heutzutage sehr dynamisch
  - Geräte kommen und gehen
  - CPUs können dynamisch aktiviert und deaktiviert werden (Akkubetrieb!)
- Daten können im (langsamen) Netzwerk liegen

Features

- verwaltet eine Reihe von “Worker Threads”
  - abhängig von der Anzahl der verfügbaren CPUs
- Programme können Arbeitspakete abschicken, die asynchron abgearbeitet werden

Queues

- Blocks werden an Queues abgeschickt
  - werden einzeln abgearbeitet
    - Thread-safe!
- mehrere Queues möglich, diese werden parallel abgearbeitet
- Queue braucht 256B, ein Thread braucht 512kB+
- Vorschlag: Eine Queue pro Subsystem

Beispiel

```
#import <stdio.h>
#import <dispatch/dispatch.h>

int main (int argc, const char * argv[]) {
   dispatch_queue_t queue;
   __block int result = 0;

   queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_LOW, 0);

   for(int i = 0; i < 100000; ++i)
       dispatch_sync(queue, ^{
           result += 1;
       });

   printf("result = %d", result);

   return 0;
}
```

Funktionen

(die wichtigsten)

dispatch_sync

- Parameter: Queue, Block
- schickt den Block an die Queue ab, wartet bis der Block gelaufen ist und returnt dann

dispatch_async

- Parameter: Queue, Block
- schickt den Block an die Queue ab (Kopie!), returnt sofort

dispatch_suspend

- Parameter: Queue
- hält die Queue nach dem gerade laufenden Block an (welcher dies ist ist undefiniert!)

dispatch_resume

- Parameter: Queue
- startet die Queue wieder, falls der Suspension Counter = 0 wird

Queue-Typen

Main Queue

- dispatch_get_main_queue()
- entspricht Hauptthread

Global Queue

- 3 Stück:
  - PRIORITY_HIGH
  - PRIORITY_NORMAL
  - PRIORITY_LOW

```
dispatch_get_global_queue(DISPATCH_QUEUE_…, NULL);
```

Private Queue

- anlegen mit
  `dispatch_queue_create(“reverse domain name”, NULL);`
- verweist standardmäßig auf die globale Queue mit normaler Priorität
- “umbiegen” mit dispatch_set_target_queue
- freigeben mit
  `dispatch_queue_release(queue);`

Wann ist mein Paket fertig?

- “Recursive Decomposition”
- am Ende des Blocks mit

```
dispatch_async(dispatch_get_main_queue(), ^{ … });
```

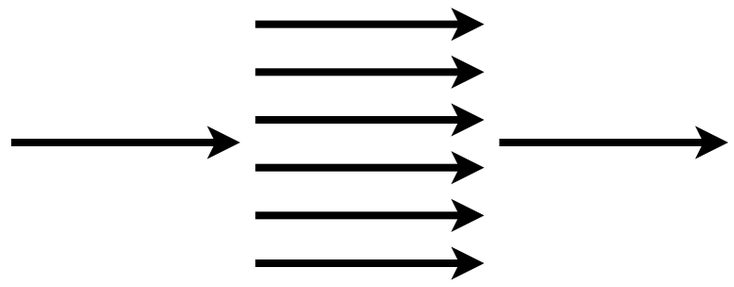
  etwas an den Hauptthread schicken.

- wird über den Runloop abgearbeitet

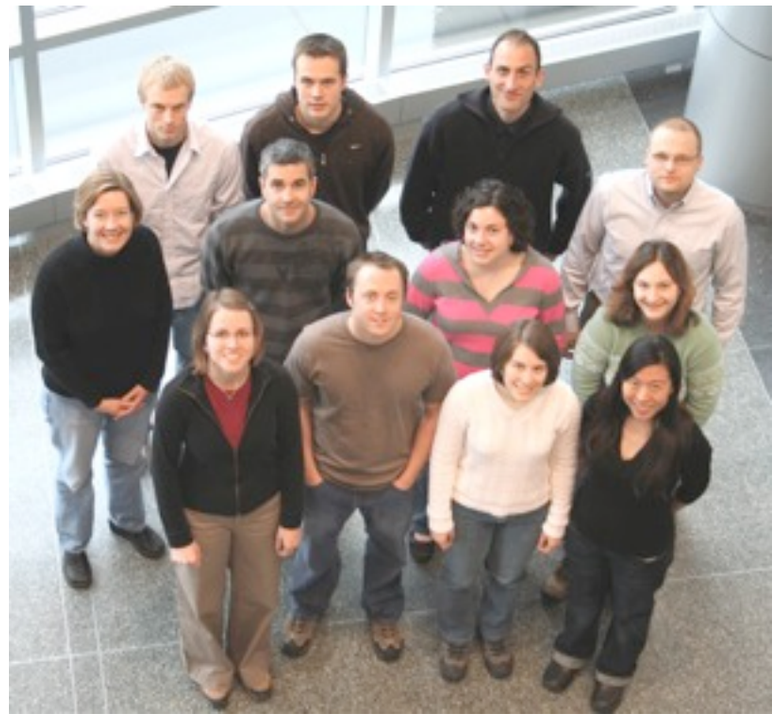
Parallele Schleifen

```
dispatch_apply(queue, count, ^(size_t idx) {
    …
});
```

- wie for-Schleife, nur parallel
- wartet bis alle Pakete fertig sind



Gruppen



- mehrere Blöcke in einer Gruppe abschicken
- ein Block wird ausgeführt, wenn alle Pakete der Gruppe abgearbeitet wurden

Gruppen (2)

```
int main (int argc, const char * argv[]) {
    dispatch_queue_t queue;
    __block int result = 0;

    queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_LOW, 0);
    dispatch_group_t group = dispatch_group_create();

    for(int i = 0; i < 100000; ++i)
        dispatch_group_async(group, queue, ^{
            result += 1;
        });

    dispatch_group_notify(group, queue, ^{
        dispatch_async(dispatch_get_main_queue(), ^{
            printf("result = %d\n", result);
            exit(0);
        });
    });

    dispatch_main();

    return 0;
}
```

Dispatch Sources

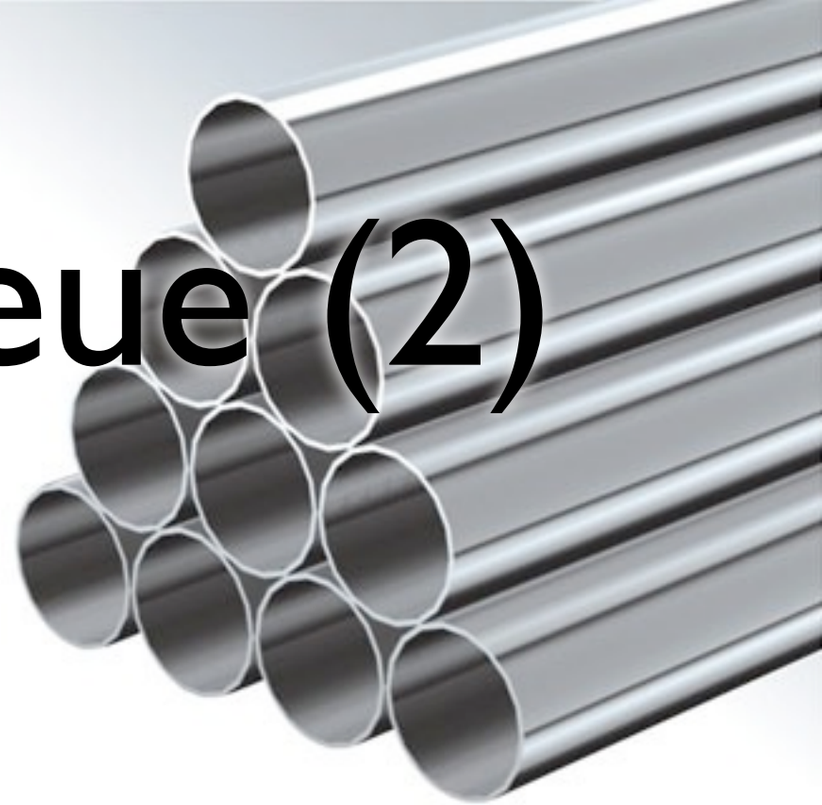
- Blöcke werden aufgrund von Events ausgeführt:
  - Mach port message
  - UNIX signals
  - File read
  - Timer
  - …

GCD in Cocoa

NSOperationQueue

- gleiche API wie in Leopard, aber von Grund auf neu geschrieben auf GCD basierend
- neu: NSBlockOperation

NSOperationQueue (2)



- ermöglicht das Abschicken von NSOperation-Objekten
- komplett Threadsafe

NSOperation

- eine Operation in der Queue
- abstrakt, Subklassen:
  - NSInvocationOperation
  - NSBlockOperation
  - eigene
- komplett Threadsafe (Subklassen!)
- Dependencies auf andere NSOperations

Beispiel

```
int main (int argc, const char * argv[]) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    __block int result = 0;

    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
    for(int i = 0; i < 100000; ++i) {
        NSBlockOperation *operation = [NSBlockOperation blockOperationWithBlock:^{
            result += 1;
        }];

        [queue addOperation:operation];
    }

    [queue waitUntilAllOperationsAreFinished];
    [queue release];

    printf("result = %d\n", result);

    [pool drain];
    return 0;
}
```

Nochmal Blockiteration

- zur Erinnerung:

```
[array enumerateObjectsUsingBlock:
 ^(id obj, NSUInteger idx, BOOL *stop) {
   // ...
 }];
```

```
[dict enumerateKeysAndObjectsUsingBlock:
 ^(id key, id obj, BOOL *stop) {
   // ...
 }];
```

- jetzt:

```
[array enumerateObjectsWithOptions:NSEnumerationConcurrent usingBlock:
 ^(id obj, NSUInteger idx, BOOL *stop) {
   // ...
 }];
```

```
[dict enumerateKeysAndObjectsWithOptions:NSEnumerationConcurrent usingBlock:
 ^(id key, id obj, BOOL *stop) {
   // ...
 }];
```

Danke für die Aufmerksamkeit!

andreas@monitzer.com