In this article

Issue 2: Concurrent Programming · August 2013

Concurrent Programming: APIs and Challenges

By Florian Kugler

Concurrency describes the concept of running several tasks at the same time. This can either happen in a time-shared manner on a single CPU core, or truly in parallel if multiple CPU cores are available.

OS X and iOS provide several different APIs to enable concurrent programming. Each of these APIs has different capabilities and limitations, making them suitable for different tasks. They also sit on very different levels of abstraction. We have the possibility to operate very close to the metal, but this also comes with great responsibility to get things right.

Concurrent programming is a very difficult subject with many intricate problems and pitfalls, and it's easy to forget this while using APIs like Grand Central Dispatch or `NSOper` … e an overview of the different concu … n dive deeper into the inherent chall … which are independent of the spec …

Concurrency APIs o…

Apple's mobile and desktop operat … or concurrent programming. In this a … `pthread` and `NSThread`, Grand C… and `NSRunLoop`. Technically, run l… because they don't enable true pa… enough to the topic that it's worth …

We'll start with the lower-level API … level ones. We chose this route because the higher-level APIs are built on top of the lower-level APIs. However, when choosing an API for your use case, you should consider them in the exact opposite order: choose the highest level abstraction that gets the job done and keep your concurrency model very simple.

If you're wondering why we are so persistent recommending high-level abstractions and very simple concurrency code, you should read the second part of this article, challenges of concurrent programming, as well as Peter Steinberger's thread safety article.

Threads

Threads are subunits of processes, which can be scheduled concurrency APIs are built on top of threads under the hood – that's true for both Grand Central Dispatch and operation queues.

In this article

Multiple threads can be executed a … e
(or at least perceived as at the sam … ns
small slices of computing time to e … ser
as if multiple tasks are executed a … s
are available, then multiple thread
therefore lessening the total time …

You can use the CPU strategy view … :
how your code or the framework co
execution on multiple CPU cores.

The important thing to keep in min … ere
and when your code gets schedule
execution will be paused in order f
kind of thread scheduling is a very
comes with great complexity, whic

Leaving this complexity aside for a moment, you can either use the POSIX thread API, or the Objective-C wrapper around this API, `NSThread`, to create your own threads. Here's a small sample that finds the minimum and maximum in a set of 1 million numbers using `pthread`. It spawns off 4 threads that run in parallel. It should be obvious from this example why you wouldn't want to use pthreads directly.

OBJECTIVE-C SELECT ALL

```
#import <pthread.h>

struct threadInfo {
    uint32_t * inputValues;
    size_t count;
};
```

```
};

void * findMinAndMax(void *arg)
{
    struct threadInfo const
    uint32_t min = UINT32_MA
    uint32_t max = 0;
    for (size_t i = 0; i < i
        uint32_t v = info->i
        min = MIN(min, v);
        max = MAX(max, v);
    }
    free(arg);
    struct threadResult * co
    result->min = min;
    result->max = max;
    return result;
}

int main(int argc, const cha
{
    size_t const count = 100
    uint32_t inputValues[cou

    // Fill input values wit
    for (size_t i = 0; i < c
        inputValues[i] = arc
    }

    // Spawn 4 threads to find the minimum and maximum:
    size_t const threadCount = 4;
    pthread_t tid[threadCount];
    for (size_t i = 0; i < threadCount; ++i) {
        struct threadInfo * const info = (struct threadInfo *
        size_t offset = (count / threadCount) * i;
        info->inputValues = inputValues + offset;
        info->count = MIN(count - offset, count / threadCount
        int err = pthread_create(tid + i, NULL, &findMinAndMa
        NSCAssert(err == 0, @"pthread_create() failed: %d", e
    }
    // Wait for the threads to exit:
    struct threadResult * results[threadCount];
    for (size_t i = 0; i < threadCount; ++i) {
        int err = pthread_join(tid[i], (void **) &(results[i]
        NSCAssert(err == 0, @"pthread_join() failed: %d", err
    }
    // Find the min and max:
```

In this article

```
        max = MAX(max, results[i]->max);
        free(results[i]);
        results[i] = NULL;
    }

    NSLog(@"min = %u", min);
    NSLog(@"max = %u", max);
    return 0;
}
```

NSThread is a simple Objective-C [...]s
the code look more familiar in a Co[...]an
define a thread as a subclass of NS[...]e
you want to run in the background.
define an NSThread subclass like

In this article

OBJECTIVE-C SELECT ALL

```
@interface FindMinMaxThread
@property (nonatomic) NSUInt
@property (nonatomic) NSUInt
- (instancetype)initWithNumb
@end

@implementation FindMinMaxTh
    NSArray *_numbers;
}

- (instancetype)initWithNumb
{
    self = [super init];
    if (self) {
        _numbers = numbers;
    }
    return self;
}

- (void)main
{
    NSUInteger min;
    NSUInteger max;
    // process the data
    self.min = min;
    self.max = max;
}
@end
```

In this article

To start new threads, we need to create new thread objects and call their `start` methods:

OBJECTIVE-C SELECT ALL

```
NSMutableSet *threads = [NSM
NSUInteger numberCount = sel
NSUInteger threadCount = 4;
for (NSUInteger i = 0; i < t
    NSUInteger offset = (num
    NSUInteger count = MIN(n
    NSRange range = NSMakeRa
    NSArray *subset = [self.                                  ;
    FindMinMaxThread *thread                                  tl
    [threads addObject:threa
    [thread start];
}
```


Now we could observe the threads en all our newly spawned threads hav lt. We will leave this exercise to the ir int is that working directly with threac `NSThread` APIs, is a relatively clur mental model of coding very well.

One problem that can arise from directly using threads is that the number of active threads increases exponentially if both your code and underlying framework code spawn their own threads. This is actually a quite common problem in big projects. For example, if you create eight threads to take advantage of eight CPU cores, and the framework code you call into from these threads does the same (as it doesn't know about the threads you already created), you can quickly end up with dozens or even hundreds of threads. Each part of the code involved acted responsibly in itself; nevertheless, the end result is problematic. Threads don't come for free. Each thread ties up memory and kernel resources.

Next up, we'll discuss two queue-based concurrency APIs: Grand Central Dispatch and operation queues. They alleviate this problem by centrally

Grand Central Dispatch

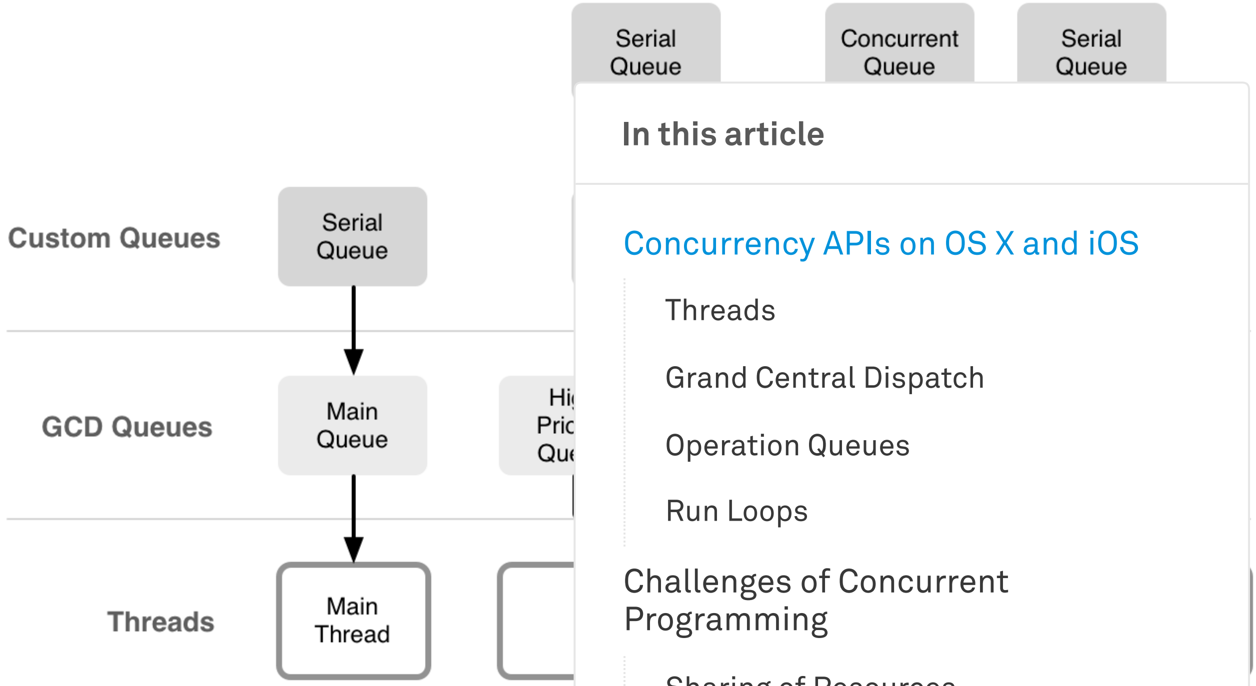
Grand Central Dispatch (GCD) was
order to make it easier for develop ng
numbers of CPU cores in consume
about GCD in our article about low

With GCD you don't interact with th
add blocks of code to queues, and he
scenes. GCD decides on which par
going to be executed on, and it ma
available system resources. This a
threads being created, because th
and abstracted away from applica

The other important change with G
about work items in a queue rathe el
of concurrency is easier to work with.

In this article

GCD exposes five different queues: the main queue running on the main thread, three background queues with different priorities, and one background queue with an even lower priority, which is I/O throttled. Furthermore, you can create custom queues, which can either be serial or concurrent queues. While custom queues are powerful abstraction, all blocks you schedule on them will ultimately trickle down to one of the system's global queues and its thread pool(s).



In this article

Making use of several queues with … straightforward at first. However, v… the default priority queue in almost all … with different priorities can quickly … se tasks access shared resources. Th… ble program to come to a grinding halt because some low-priority tasks are blocking a high-priority task from executing. You can read more about this phenomenon, called priority inversion, below.

Although GCD is a low-level C API, it's pretty straightforward to use. This makes it easy to forget that all caveats and pitfalls of concurrent programming still apply while dispatching blocks onto GCD queues. Please make sure to read about the challenges of concurrent programming below, in order to be aware of the potential problems. Furthermore, we have an excellent walkthrough of the GCD API in this issue that contains many in-depth explanations and valuable hints.

Operation Queues

implement several convenient features on top of it, which often makes it the best and safest choice for application developers.

The `NSOperationQueue` class has ain queue and custom queues. The ma d custom queues are processed in th which are processed by these queu `NSOperation`.

You can define your own operation `main`, or by overriding `start`. The s you less flexibility. In return, the st `isFinished` are managed for you, on is finished when `main` returns.

OBJECTIVE-C SELE

```
@implementation YourOperatio
    - (void)main
    {
        // do your work here ...
    }
@end
```

If you want more control and to maybe execute an asynchronous task within the operation, you can override `start`:

OBJECTIVE-C SELECT ALL

```
@implementation YourOperatio
    - (void)start
    {
        self.isExecuting = Y
        self.isFinished = NO
        // start your work,
    }

    - (void)finished
    {
        self.isExecuting = N
        self.isFinished = YE
    }
@end
```

In this article

Notice that in this case, you have t manually. In order for an operation change, the state properties have nt way. So make sure to send proper them via default accessor methods.

In order to benefit from the cancelation feature exposed by operation queues, you should regularly check the `isCancelled` property for longer-running operations:

OBJECTIVE-C SELECT ALL

```
- (void)main
{
    while (notDone && !self.isCancelled) {
        // do your processing
    }
}
```

OBJECTIVE-C SELECT ALL

```
NSOperationQueue *queue = [[
YourOperation *operation = [
[queue  addOperation:operati
```


Alternatively, you can also add blo[...] n handy, e.g. if you want to schedule

OBJECTIVE-C

```
[[NSOperationQueue mainQueue
    // do something...
}];
```

While this is a very convenient way [...] defining your own NSOperation su[...] debugging. If you override the oper[...] ın easily identify all the operations currently scheduled in a certain queue.

Beyond the basics of scheduling operations or blocks, operation queues offer some features which would be non-trivial to get right in GCD. For example, you can easily control how many operations of a certain queue may be executed concurrently with the `maxConcurrentOperationCount` property. Setting it to one gives you a serial queue, which is great for isolation purposes.

Another convenient feature is the sorting of operations within a queue according to their priorities. This is not the same as GCD's queue priorities. It solely influences the execution order of all operations scheduled in one queue. If you need more control over the sequence of execution beyond the five standard priorities, you can specify dependencies between operations like this:

OBJECTIVE-C SELECT ALL

```
[intermediateOperation addDe
[intermediateOperation addDe
[finishedOperation addDepend
```


This simple code guarantees that e executed before intermediateOp executed before finishedOperat ery powerful mechanism to specify a v s you create things like operation gr executed before the dependent op n otherwise concurrent queue.

By the very nature of abstractions, performance hit compared to usin cases, this impact is negligible an choice.

Run Loops

Run loops are not technically a concurrency mechanism like GCD or operation queues, because they don’t enable the parallel execution of tasks. However, run loops tie in directly with the execution of tasks on the main dispatch/operation queue and they provide a mechanism to execute code asynchronously.

Run loops can be a lot easier to use than operation queues or GCD, because you don’t have to deal with the complexity of concurrency and still get to do things asynchronously.

A run loop is always bound to one particular thread. The main run loop kernel events. Whenever you schedule a timer, use a `NSURLConnection`. or call `performSelector:withObject:afterDelay:`, the run loop is used behind the scenes in order to

Whenever you use a method which … to remember that run loops can be ru… defines a set of events the run loop … way to temporarily prioritize certain ta…

A typical example of this is scrollin… un loop is not running in its default m… react to, for example, a timer you h… stops, the run loop returns to the d… ave been queued up are executed. If yo… , you need to add it to the run loop i…

The main thread always has the m… r threads though don't have a run lo… up a run loop for other threads too, bu… of the time it is much easier to use the main run loop. If you need to do heavier work that you don't want to execute on the main thread, you can still dispatch it onto another queue after your code is called from the main run loop. Chris has some good examples of this pattern in his article about common background practices.

In this article

If you really need to set up a run loop on another thread, don't forget to add at least one input source to it. If a run loop has no input sources configured, every attempt to run it will exit immediately.

Challenges of Concurrent Programming

Writing concurrent programs comes with many pitfalls. As soon as you're parallel. Problems can occur in a non-deterministic way, which makes it even more difficult to debug concurrent code.

There is a prominent example for u
programs: In 1995, NASA sent the
long after a successful landing on on
almost came to an abrupt end. The wn
reasons – it suffered from a pheno e a
low-priority thread kept blocking a
explore this particular issue in mo uld
demonstrate that even with vast re nt
available, concurrency can come b

In this article

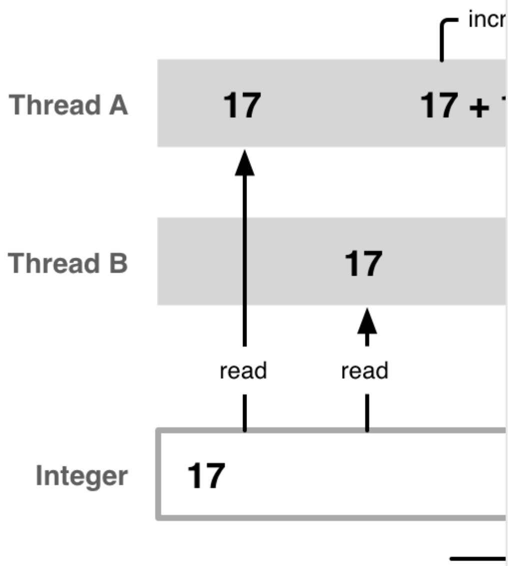
Sharing of Resources

The root of many concurrency relat
resources from multiple threads. A
object, memory in general, a netwo
share between multiple threads is a potential point of conflict, and you have to take safety measures to prevent these kind of conflicts.

In order to demonstrate the problem, let's look at a simple example of a resource in the form of an integer property which you're using as a counter. Let's say we have two threads running in parallel, A and B, and both try to increment the counter at the same time. The problem is that what you write as one statement in C or Objective-C is mostly not just one machine instruction for the CPU. To increment our counter, the current value has to be read from memory. Then the value is incremented by one and finally written back to memory.

Imagine the hazards that can happen if both threads try to do this simultaneously. For example, thread A and thread B both read the value of

time, thread B also increments the counter by one and writes a 18 back to memory, just after thread A. At this point the data has become corrupted, because the counter hc
twice from a 17.



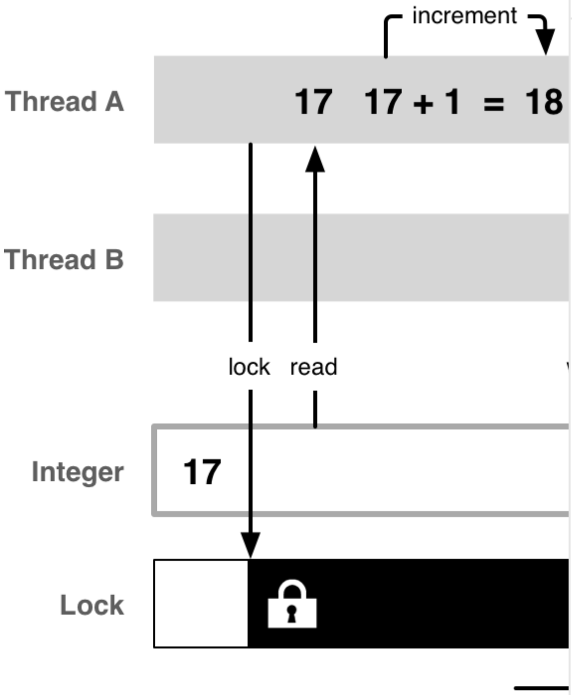
In this article

This problem is called a race condi le
threads access a shared resource is
finished operating on a resource before another one begins accessing it. If you're not only writing a simple integer but a more complex structure to memory, it might even happen that a second thread tries to read from this memory while you're in the midst of writing it, therefore seeing half new and half old or uninitialized data. In order to prevent this, multiple threads need to access shared resources in a mutually exclusive way.

In reality, the situation is even more complicated than this, because modern CPUs change the sequence of reads and writes to memory for optimization purposes (Out-of-order execution).

Mutual Exclusion

Mutual exclusive access means that only one thread at a time gets access finished its operation, it releases the lock, so that other threads get a chance to access it.



In this article

In addition to ensuring mutual exc
the problem caused by out-of-orde
CPU accessing the memory in the sequence defined by your program instructions, guaranteeing mutually exclusive access alone is not enough. To work around this side effect of CPU optimization strategies, memory barriers are used. Setting a memory barrier makes sure that no out-of-order execution takes place across the barrier.

Of course the implementation of a mutex lock in itself needs to be race-condition free. This is a non-trivial undertaking and requires use of special instructions on modern CPUs. You can read more about atomic operations in Daniel's low-level concurrency techniques article.

Objective-C properties come with language level support for locking in the form of declaring them as atomic. In fact, properties are even atomic by default. Declaring a property as atomic results in implicit locking/unlocking around each access of this property. It might be

Acquiring a lock on a resource always comes with a performance cost. Acquiring and releasing a lock needs to be race-condition free, which is non-trivial on multi-core systems. … d might have to wait because some … n this case, that thread will sleep an… thread relinquishes the lock. All of… complicated.

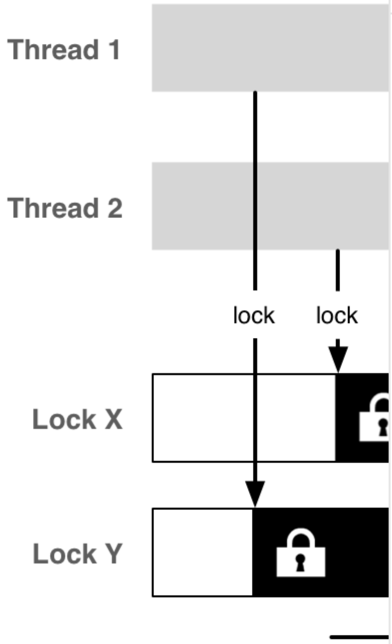
There are different kinds of locks. … e's no lock contention but perform po… e more expensive at a base level, bu… ock contention is the situation when o… that has already been taken).

There is a trade-off to be made he… es at a price (lock overhead). Therefo… constantly entering and exiting cri… releasing locks). At the same time, … region of code, you run the risk of l… are often unable to do work because they're waiting to acquire a lock. It's not an easy task to solve.

It is quite common to see code which is supposed to run concurrently, but which actually results in only one thread being active at a time, because of the way locks for shared resources are set up. It's often non-trivial to predict how your code will get scheduled on multiple cores. You can use Instrument's CPU strategy view to get a better idea of whether you're efficiently using the available CPU cores or not.

Dead Locks

Mutex locks solve the problem of race conditions, but unfortunately they locks. A dead lock occurs when multiple threads are waiting on each other to finish and get stuck.



Consider the following example co variables:

OBJECTIVE-C SELECT ALL

```
void swap(A, B)
{
    lock(lockA);
    lock(lockB);
    int a = A;
    int b = B;
    A = b;
    B = a;
    unlock(lockB);
    unlock(lockA);
}
```

This works quite well most of the time. But when by chance two threads call it at the same time with opposite variables

OBJECTIVE-C SELECT ALL

```
swap(X, Y); // thread 1
swap(Y, X); // thread 2
```

we can end up in a dead lock. Thre
acquires a lock on Y. Now they're b l
never be able to acquire it.

Again, the more resources you sha ks
you take, the greater your risk of ru is
one more reason to keep things as ew
resources as possible between thr
section about doing things asynch /
APIs article.

In this article

Starvation

Just when you thought that there are enough problems to think of, a new one comes around the corner. Locking shared resources can result in the readers-writers problem. In many cases, it would be wasteful to restrict reading access to a resource to one access at a time. Therefore, taking a reading lock is allowed as long as there is no writing lock on the resource. In this situation, a thread that is waiting to acquire a write lock can be starved by more read locks occurring in the meantime.

In order to solve this issue, more clever solutions than a simple read/write lock are necessary, e.g. giving writers preference or using the read-copy-update algorithm. Daniel shows in his low-level concurrency techniques article how to implement a multiple reader/single writer pattern with GCD which doesn't suffer from writer starvation.

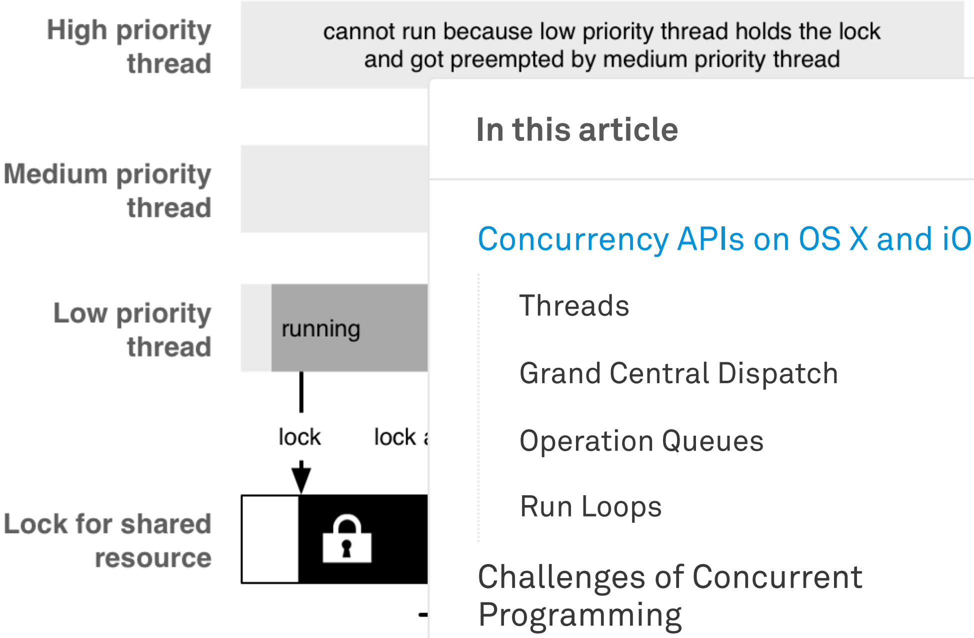
Priority Inversion

We started this section with the ex
Mars suffering from a concurrency
look why Pathfinder almost failed,
from the same problem, called pri

Priority inversion describes a cond ks
a higher priority task from executi s.
Since GCD exposes background qu ng
one which even is I/O throttled, it's

The problem can occur when you h
task share a common resource. Wh to
the common resource, it is suppos
release its lock and to let the high-
significant delays. Since the high- as
long as the low-priority task has the lock, there is a window of opportunity for medium-priority tasks to run and to preempt the low-priority task, because the medium-priority tasks have now the highest priority of all currently runnable tasks. At this moment, the medium-priority tasks hinder the low-priority task from releasing its lock, therefore effectively gaining priority over the still waiting, high-priority tasks.

In this article



In this article

In your own code, things might not … occurred in the Mars rover, as prio … a less severe manner.

In general, don't use different prio … - priority code waiting on low-priorit … CD, always use the default priority queue (directly, or as a target queue). If you're using different priorities, more likely than not, it's actually going to make things worse.

The lesson from this is that using multiple queues with different priorities sounds good on paper, but it adds even more complexity and unpredictability to concurrent programs. And if you ever run into a weird problem where your high-priority tasks seem to get stuck for no reason, maybe you will remember this article and the problem called priority inversion, which even the NASA engineers encountered.

Conclusion

resulting behavior quickly gets very difficult to oversee, and debugging these kind of problems is often very hard.

On the other hand, concurrency is ... the computing power of modern multi... concurrency model as simple as p... amount of locking necessary.

A safe pattern we recommend is th... on the main thread, then use an op... the background, and finally get ba... result of your background work. Th... yourself, which greatly reduces the...

In this article

CONTINUE READING ISSUE 2

Concurrent Programming

July 2013

by Tobias Kräntzer

very occasional updates about upcoming books, promotions, and other things.

Your Email

objc ↑↓

objc.io publishes books, videos, ar... ...or iOS and OS X development.

LEARN

- Swift Talk
- Books
- Workshops
- Issues

FOLLOW

- Blog
- Newsle...
- Twitter
- YouTube

In this article