ByteDance 字节跳动

20秋

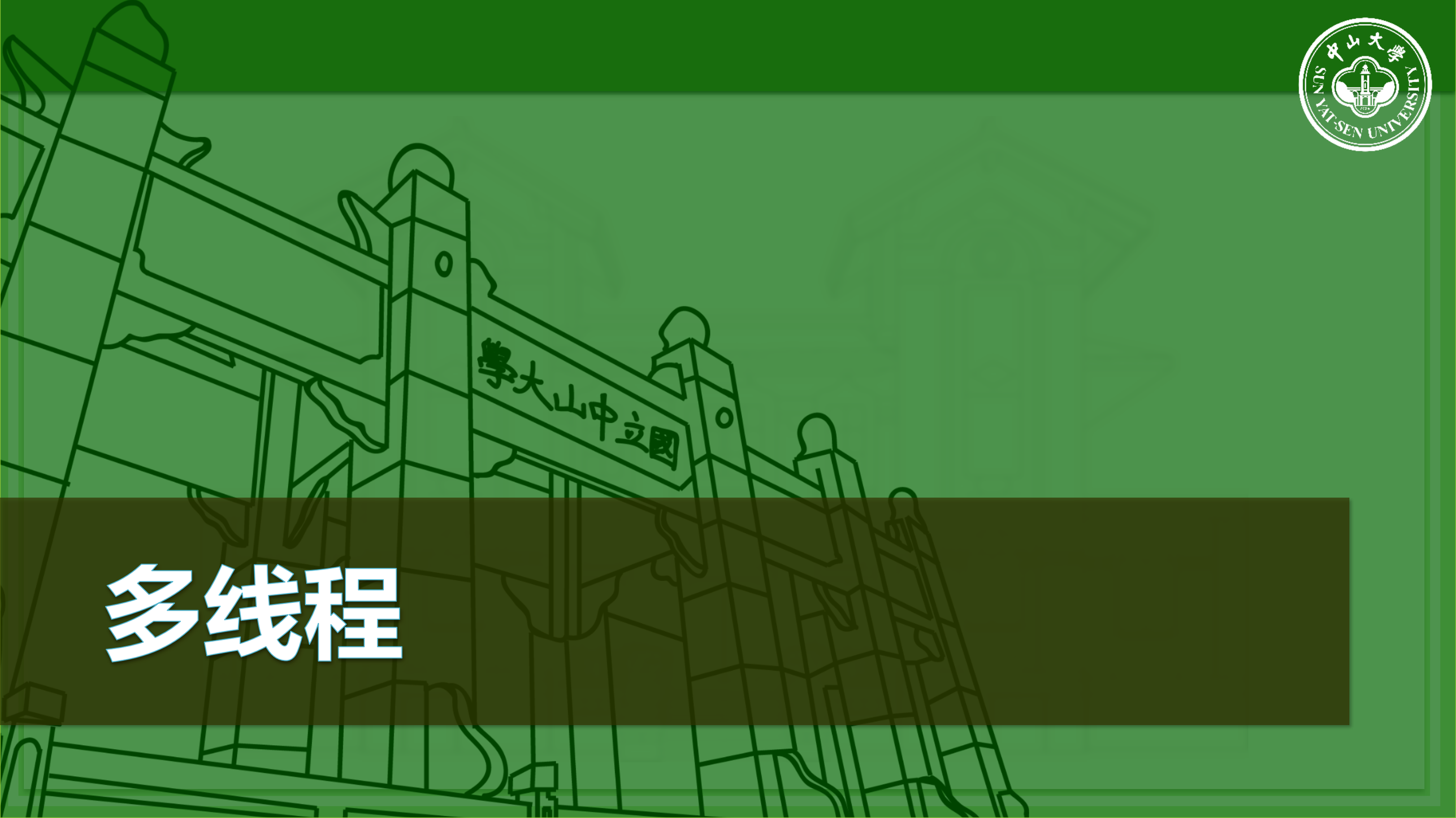
MOSAD

现代操作系统应用开发

#10 多线程编程

Content

- 多线程编程
  - POSIX Thread
  - NSThread
  - GCD
  - NSOperation
- 线程安全
- 同步机制
- 调试工具



多线程

线程的概念

- 操作系统任务调度的基本单元
  - 合理使用多线程才能充分利用多核 CPU
  - 合理设置优先级让重要的任务（例如主线程）更快完成
- 共享一个进程内的资源
  - 共享虚拟内存/描述符等
  - 多个线程访问共享资源可能存在竞争
- 有独立的调用栈/本地变量/寄存器上下文
  - 线程的创建/销毁/切换也是有一定开销的

常见API

- POSIX Thread：pthread
- NSThread：pthread的OO封装
- GCD：iOS官方最常用C API
- NSOperation：GCD的OO封装

POSIX Thread

- POSIX Thread是POSIX的线程标准，定义了创建和操作线程的一套API。

  POSIX（Portable Operating System Interface of UNIX）标准定义了操作系统应该为应用程序提供的接口标准
- 实现POSIX 线程标准的库常被称作Pthreads
- Pthreads定义了一套C语言的类型、函数与常量，它以pthread.h头文件和一个线程库实现。
- POSIX Thread 特点

  所有的 UN*X 系统都支持

  开源：https://opensource.apple.com/source/libpthread/

  包含非标准的扩展：“_np”后缀（非跨平台）

  C接口

pthread 使用方法

```
// 1. 创建线程: 定义一个pthread_t类型变量
pthread_t thread;
// 2. 开启线程: 执行任务
pthread_create(&thread, NULL, run, NULL);
// 3. 设置子线程的状态设置为detached, 线程运行结束后自动释放所有资源
pthread_detach(thread);

void * run(void *param)     // 线程调用的方法, 线程需要执行的任务
{
    NSLog(@"%@", [NSThread currentThread]);
    return NULL;
}
```

pthread 使用方法

- `pthread_create(&thread, NULL, run, NULL);`

  第一个参数&thread是线程对象，指向线程标识符的指针

  第二个是线程属性，可赋值NULL

  第三个run表示指向函数的指针(run对应函数里是需要在新线程中执行的任务)

  第四个是运行函数的参数，可赋值NULL

pthread 使用方法

- pthread相关方法

  pthread_create() 创建一个线程

  pthread_exit() 终止当前线程

  pthread_cancel() 中断另外一个线程的运行

  pthread_join() 阻塞当前的线程，直到其他线程运行结束

  pthread_attr_init() 初始化线程的属性

  pthread_attr_setdetachstate() 设置脱离状态的属性（决定这个线程在终止时是否可以被结合）

  pthread_attr_getdetachstate() 获取脱离状态的属性

  pthread_attr_destroy() 删除线程的属性

  pthread_kill() 向线程发送一个终止信号

POSIX Thread 使用示例

```c
#import <pthread.h>

// 输入值
struct threadInfo {
  uint32_t * inputValues;
  size_t count;
};

// 返回结果
struct threadResult {
  uint32_t min;
  uint32_t max;
};

// 求数组的最小最大值
void * findMinAndMax(void *arg) {

  struct threadInfo const * const info =
    (struct threadInfo *) arg;

  uint32_t min = UINT32_MAX;
  uint32_t max = 0;

  for (size_t i = 0; i < info->count; ++i) {
    uint32_t v = info->inputValues[i];
    min = MIN(min, v);
    max = MAX(max, v);
  }
  free(arg);

  struct threadResult * const result =
    (struct threadResult *) malloc(sizeof(*result));
  result->min = min;
  result->max = max;

  return result;
}
```

POSIX Thread 使用示例

```
// 主函数
int main(int argc, const char * argv[]) {
  size_t const count = 1000000;
  uint32_t inputValues[count];

  // Fill input values with random numbers:
  for (size_t i = 0; i < count; ++i) {
    inputValues[i] = arc4random();
  }

  // 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 *) malloc(sizeof(*info));
    size_t offset = (count / threadCount) * i;
    info->inputValues = inputValues + offset;
    info->count =
      MIN(count - offset, count / threadCount);
    int err = pthread_create(
      tid + i, NULL, &findMinAndMax, info);
    NSCAssert(err == 0,
      @"pthread_create() failed: %d", err);
  }

  // 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:
  uint32_t min = UINT32_MAX;
  uint32_t max = 0;
  for (size_t i = 0; i < threadCount; ++i) {
    min = MIN(min, results[i]->min);
    max = MAX(max, results[i]->max);
    free(results[i]);
    results[i] = NULL;
  }
  NSLog(@"min = %u", min);
  NSLog(@"max = %u", max);
  return 0;
}
```

man 3 pthread

- 跟linux一样，Mac下可使用man命令查看pthread的详细说明

guifeng — less ◂ man 3 pthread — 80×36

```
PTHREAD(3)              BSD Library Functions Manual              PTHREAD(3)

NAME
     pthread -- POSIX thread functions

SYNOPSIS
     #include <pthread.h>

DESCRIPTION
     POSIX threads are a set of functions that support applications with
     requirements for multiple flows of control, called threads, within a
     process.  Multithreading is used to improve the performance of a program.

     The POSIX thread functions are summarized in this section in the follow-
     ing groups:

           o   Thread Routines
           o   Attribute Object Routines
           o   Mutex Routines
           o   Condition Variable Routines
           o   Read/Write Lock Routines
           o   Per-Thread Context Routines
           o   Cleanup Routines

   Thread Routines
     int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
             void *(*start_routine)(void *), void *arg)
             Creates a new thread of execution.

     int pthread_cancel(pthread_t thread)
             Cancels execution of a thread.

     int pthread_detach(pthread_t thread)
             Marks a thread for deletion.
:
```

NSThread简介

- NSThread 是苹果官方提供的针对Pthread封装的OC对象，简单易用，可以直接操作线程对象。需要程序员自己管理线程的生命周期(主要是创建)
- NSThread 特点

  pthread 的 OO 封装

  只暴露了 pthread 的部分能力

  可以通过 KVO 监听部分属性：isExecuting/isFinished/isCancelled等

NSThread使用方法

- 先创建再启动

```
NSThread *thread = [[NSThread alloc]
    initWithTarget:self
          selector:@selector(run)
            object:nil];
[thread start];    // 线程一启动，就会在线程thread中执行self的run方法
```

- 或者，创建线程后自动启动线程

```
[NSThread detachNewThreadSelector:@selector(run)
                         toTarget:self
                       withObject:nil];
```

- 或者，隐式创建并启动线程

```
[self performSelectorInBackground:@selector(run) withObject:nil];

// 新线程调用方法，里边为需要执行的任务
- (void)run {
    NSLog(@"%@", [NSThread currentThread]);
}
```

NSThread使用方法

- 常用方法

```
// 获得主线程
+ (NSThread *)mainThread;

// 判断是否为主线程(对象方法)
- (BOOL)isMainThread;

// 判断是否为主线程(类方法)
+ (BOOL)isMainThread;

// 获得当前线程
NSThread *current = [NSThread currentThread];

// 线程的名字——setter方法
- (void)setName:(NSString *)n;

// 线程的名字——getter方法
- (NSString *)name;
```

NSThread使用方法

- 线程状态控制

```
- (void)start;
// 线程进入就绪状态 -> 运行状态。当线程任务执行完毕，自动进入死亡状态

+ (void)sleepUntilDate:(NSDate *)date;
+ (void)sleepForTimeInterval:(NSTimeInterval)ti;
// 线程进入阻塞状态

+ (void)exit;
// 线程进入死亡状态
```

NSThread使用方法

- 子线程与主线程的通信

```
// 在主线程上执行操作
- (void)performSelectorOnMainThread:(SEL)aSelector withObject:(id)arg waitUntilDone:(BOOL)wait;
- (void)performSelectorOnMainThread:(SEL)aSelector withObject:(id)arg waitUntilDone:(BOOL)wait
                              modes:(NSArray<NSString *> *)array;
  // equivalent to the first method with kCFRunLoopCommonModes

// 在指定线程上执行操作
- (void)performSelector:(SEL)aSelector onThread:(NSThread *)thr withObject:(id)arg
          waitUntilDone:(BOOL)wait modes:(NSArray *)array NS_AVAILABLE(10_5, 2_0);
- (void)performSelector:(SEL)aSelector onThread:(NSThread *)thr withObject:(id)arg
          waitUntilDone:(BOOL)wait NS_AVAILABLE(10_5, 2_0);

// 在当前线程上执行操作，调用NSObject的performSelector:相关方法
- (id)performSelector:(SEL)aSelector;
- (id)performSelector:(SEL)aSelector withObject:(id)object;
- (id)performSelector:(SEL)aSelector withObject:(id)object1 withObject:(id)object2;
```

NSThread 使用示例

```objc
// 继承NSThread，封装自己的线程子类

@interface FindMinMaxThread : NSThread

@property (nonatomic) NSUInteger min;
@property (nonatomic) NSUInteger max;

- (instancetype)initWithNumbers:(NSArray *)numbers;

@end



@implementation FindMinMaxThread {
    NSArray *_numbers;
}

- (instancetype)initWithNumbers:(NSArray *)numbers {
    if ((self = [super init])) {
        _numbers = numbers;
    }
    return self;
}
```

```objc
// 线程主函数，启动线程时执行

- (void)main {
    __block NSUInteger min = NSUIntegerMax;
    __block NSUInteger max = 0;
    [_numbers enumerateObjectsUsingBlock:
        ^(NSNumber *number, NSUInteger idx, BOOL *stop)
        {
            min = MIN(min, number.unsignedIntegerValue);
            max = MAX(max, number.unsignedIntegerValue);
        }
    ];

    self.min = min;
    self.max = max;
}

@end
```

NSThread 使用示例

```
// 自定义线程子类的使用

NSMutableSet *threads = [NSMutableSet set];
NSUInteger numberCount = 1000000;
NSUInteger threadCount = 4;

for (NSUInteger i = 0; i < threadCount; i++) {

    // 准备参数
    NSUInteger offset = (numberCount / threadCount) * i;
    NSUInteger count = MIN(numberCount - offset, numberCount / threadCount);
    NSRange range = NSMakeRange(offset, count);
    NSArray *subset = [numbers subarrayWithRange:range];

    // 创建线程对象
    FindMinMaxThread *thread = [[FindMinMaxThread alloc] initWithNumbers:subset];

    // 加入集合，方便管理
    [threads addObject:thread];

    // 启动线程
    [thread start];
}
```

GCD简介

- Grand Central Dispatch（GCD）是 Apple 开发的一个多核编程的较新的解决方法。它主要用于优化应用程序以支持多核处理器以及其他对称多处理系统。它是一个在线程池模式的基础上执行的并发任务。
- GCD 特点

  内部管理了一个线程池

  基于队列的抽象

  支持 block 作为任务的载体

  开源：https://opensource.apple.com/source/libdispatch/

GCD核心概念

- **队列：**一种特殊的线性表，采用FIFO（先进先出）的原则。根据执行顺序以及开放线程数量的不同，可以分为两种队列：

  串行队列：每次只有一个任务被执行，让任务一个接着一个地执行。（只开启一个线程，一个任务执行完毕后，再执行下一个任务）

  并行队列：可以让多个任务并发（同时）执行。（可以开启多个线程，并且同时执行任务）

- **任务：**要执行的操作，在线程中执行的那段代码。在GCD中是放在block中的。任务有两种执行方式：

  同步执行：同步添加任务到指定的队列中，在添加的任务执行结束之前，会一直等待，直到队列里面的任务完成之后再继续执行。只能在当前线程中执行任务，不具备开启新线程的能力。

  异步执行：异步添加任务到指定的队列中，它不会做任何等待，可以继续执行任务。可以在新的线程中执行任务，具备开启新线程的能力。

man 3 dispatch

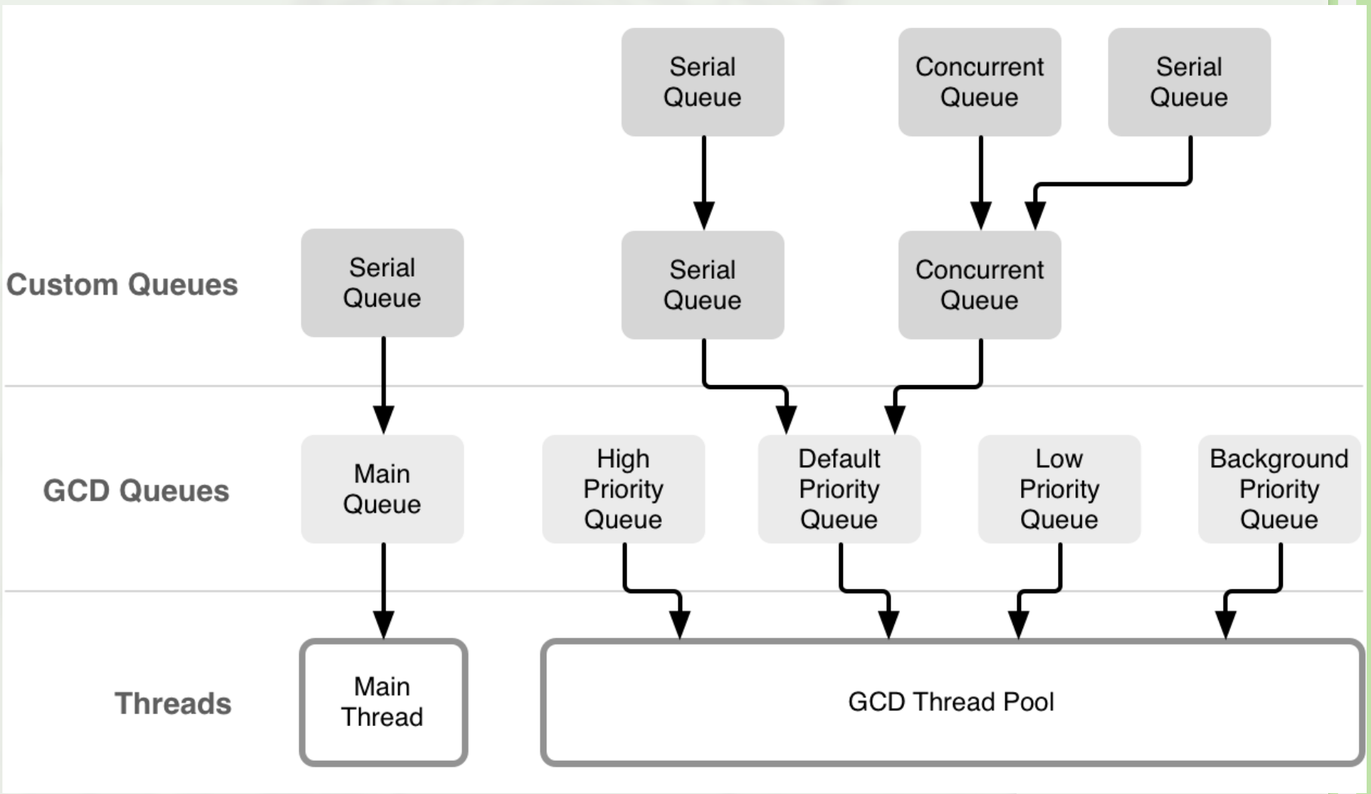
- GCD相关文档查阅

  man 3 dispatch

```
dispatch(3)             BSD Library Functions Manual             dispatch(3)

NAME
     dispatch -- the dispatch framework

SYNOPSIS
     #include <dispatch/dispatch.h>

DESCRIPTION
     The dispatch framework allows blocks to be scheduled for asynchronous and
     concurrent execution via the core functions described in
     dispatch_async(3) and dispatch_apply(3).

     Dispatch queues are the basic units of organization of blocks. Several
     queues are created by default, and applications may create additional
     queues for their own use. See dispatch_queue_create(3) for more informa-
     tion.

     Dispatch groups allow applications to track the progress of blocks sub-
     mitted to queues and take action when the blocks complete. See
     dispatch_group_create(3) for more information.

     The dispatch framework also provides functions to monitor underlying sys-
     tem events and automatically submit event handler blocks to dispatch
     queues.

SEE ALSO
     dispatch_after(3), dispatch_api(3), dispatch_apply(3), dispatch_async(3),
     dispatch_data_create(3), dispatch_group_create(3), dispatch_io_create(3),
     dispatch_io_read(3), dispatch_object(3), dispatch_once(3),
     dispatch_queue_create(3), dispatch_semaphore_create(3),
     dispatch_source_create(3), dispatch_time(3)

Darwin                          May 1, 2009                          Darwin
:
```

GCD 队列

- FIFO
- 支持串行和并行队列
- 支持队列的挂起和恢复



GCD 队列

- 获取主队列

```
dispatch_queue_t queue = dispatch_get_main_queue();
```

- 获取全局队列

```
dispatch_queue_t queue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
```

- 创建串行队列

```
dispatch_queue_t queue = dispatch_queue_create("serial-queue", NULL);
```

- 创建并行队列

```
dispatch_queue_t queue =
    dispatch_queue_create("concurrent-queue", DISPATCH_QUEUE_CONCURRENT);
```

GCD 任务

- 往队列里添加任务

```
// 异步执行：任务加入队列后函数返回
dispatch_async(queue, ^{
    NSLog(@"Do something");
});

// 同步执行：阻塞当前现场并等待任务完成
dispatch_sync(queue, ^{
    NSLog(@"Do something");
});
```

GCD 任务栅栏

```
dispatch_async(queue, ^{
    NSLog(@"----任务 1-----");
});

dispatch_async(queue, ^{
    NSLog(@"----任务 2-----");
});

dispatch_barrier_async(queue, ^{
    NSLog(@"----barrier-----");
});

dispatch_async(queue, ^{
    NSLog(@"----任务 3-----");
});

dispatch_async(queue, ^{
    NSLog(@"----任务 4-----");
});
```

任务栅栏：任务3、4在任务1、2完成后才执行

GCD 使用示例

- 例子：子线程下载图片，完成后更新主线程UI

```
- (void)viewWillAppear:(BOOL)animated
{
    dispatch_queue_t downloadQueue = dispatch_queue_create("image_downloader", NULL);
    dispatch_async(downloadQueue, ^{
        NSData *imageData = [NSData dataWithContentsOfURL:networkURL];
        dispatch_async(dispatch_get_main_queue(), ^{
            UIImage *image = [UIImage imageWithData:imageData];
            self.imageView.image = image;
            self.imageView.frame = CGRectMake(0, 0, image.size.width, image.size.height);
            self.scrollView.contentSize = image.size;
            });
    });
    dispatch_release(downloadQueue);
}
```

GCD注意事项

- 全局队列同时也是并行的
- 队列优先级尽量使用 DEFAULT
- 队列和线程并不是一一对应

  主线程可执行多个队列里的任务

  一个队列里的任务可在多个线程执行

NSOperation简介

- NSOperation、NSOperationQueue 是苹果提供的一套多线程解决方案。基于GCD的完全面向对象封装，比 GCD 更简单易用、代码可读性也更高。
- NSOperation特点

  GCD的OO 封装

  支持 GCD 的部分功能

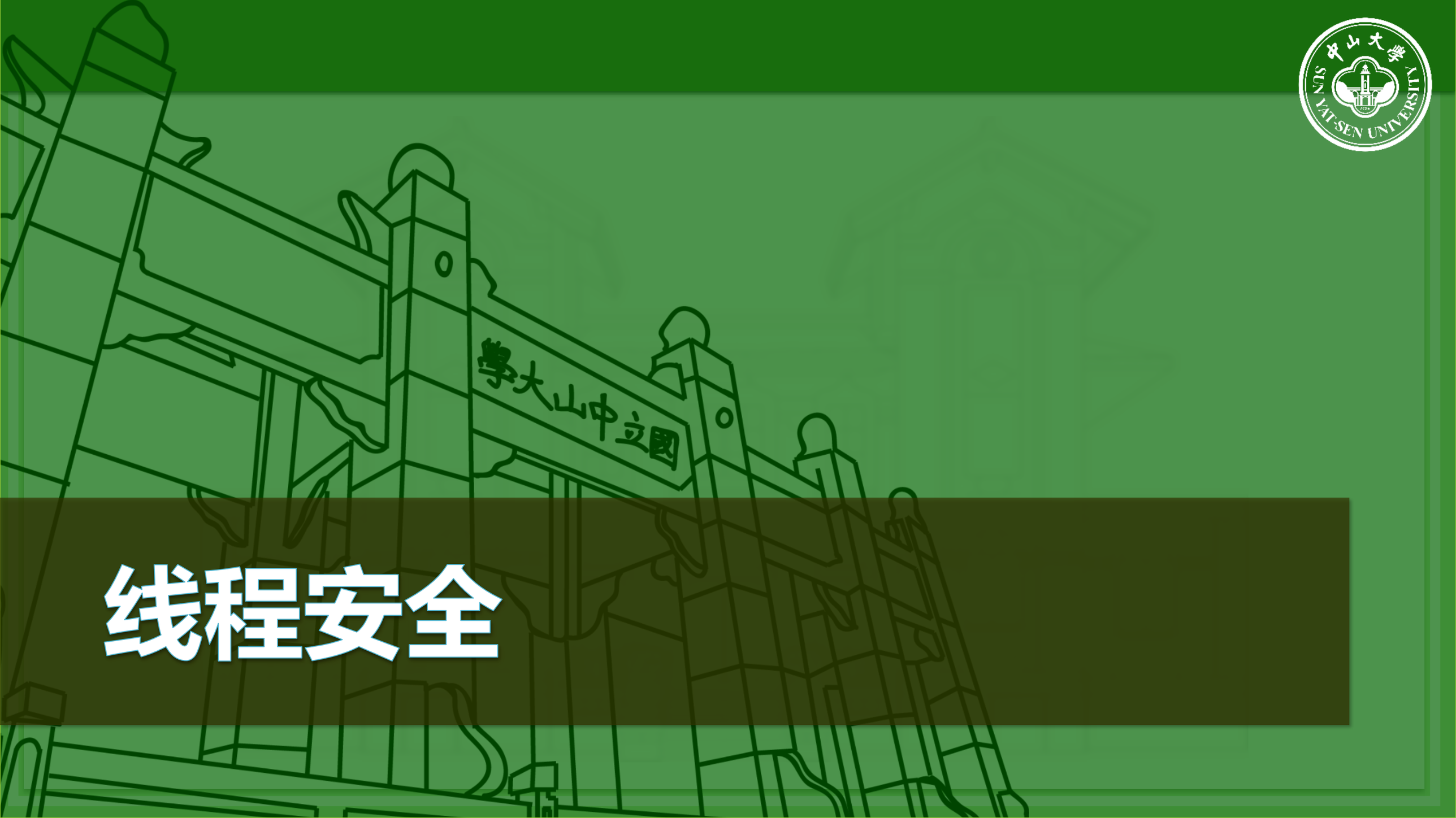
  支持指定任务依赖

  支持设定并发数

  支持取消任务/KVO监听任务状态

NSOperation 使用步骤

- 创建操作：先将需要执行的操作封装到一个 NSOperation 对象中。有三种封装方式：
  - 使用子类 NSInvocationOperation
  - 使用子类 NSBlockOperation
  - 自定义继承自 NSOperation 的子类，通过实现内部相应的方法来封装操作。
- 创建队列：创建 NSOperationQueue 对象。共有两种队列：
  - 获取主队列
  - 创建自定义队列
- 将操作加入到队列中：将 NSOperation 对象添加到 NSOperationQueue 对象中。



线程安全

什么是线程安全

- 当多个线程访问同一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替运行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获取正确的结果，那么这个对象是**线程安全**的。
- 线程不安全的对象

  文档中未注明线程安全的对象（如NSMutableArray）
- 只能在某个线程使用的对象

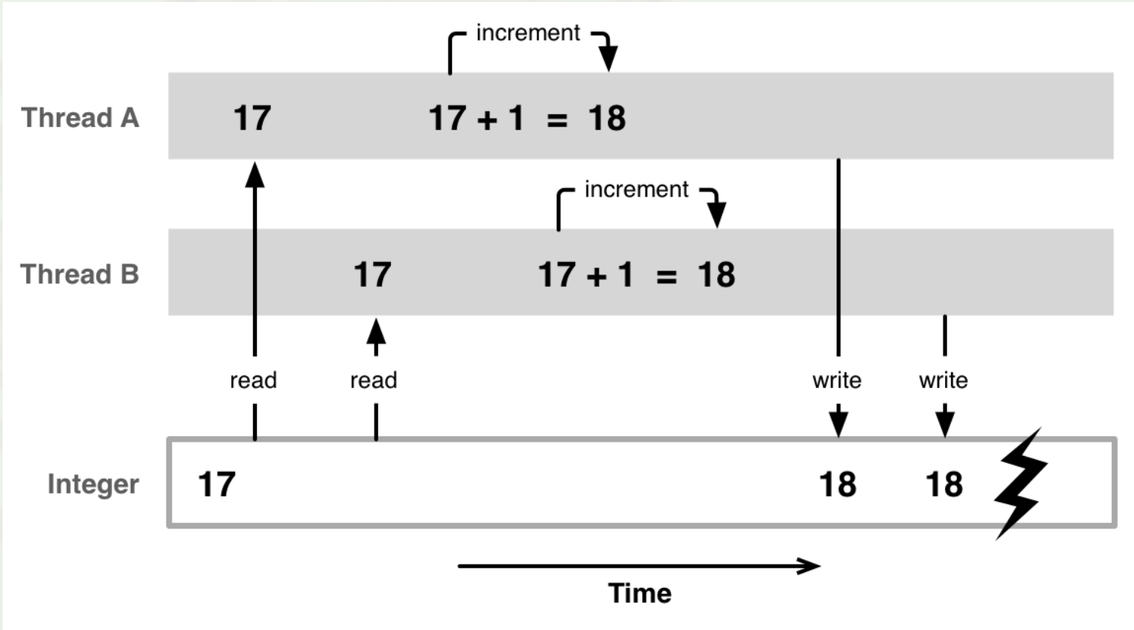
  UIKit/CoreAnimation 中几乎所有的对象只能在主线程使用

  例外：UIImage/UIFont

线程安全的难点

- 多线程下操作执行的顺序不可预测
- 例子：

```
@interface Sequence : NSObject {
    NSInteger _value;
}
@end

@implementation Sequence

- (NSInteger)value {
    return _value;
}

- (void)next {
    _value++;
}

@end
```



思考题

- 如果 _value 是个OC对象会怎样?

  列出可能出现问题的执行顺序

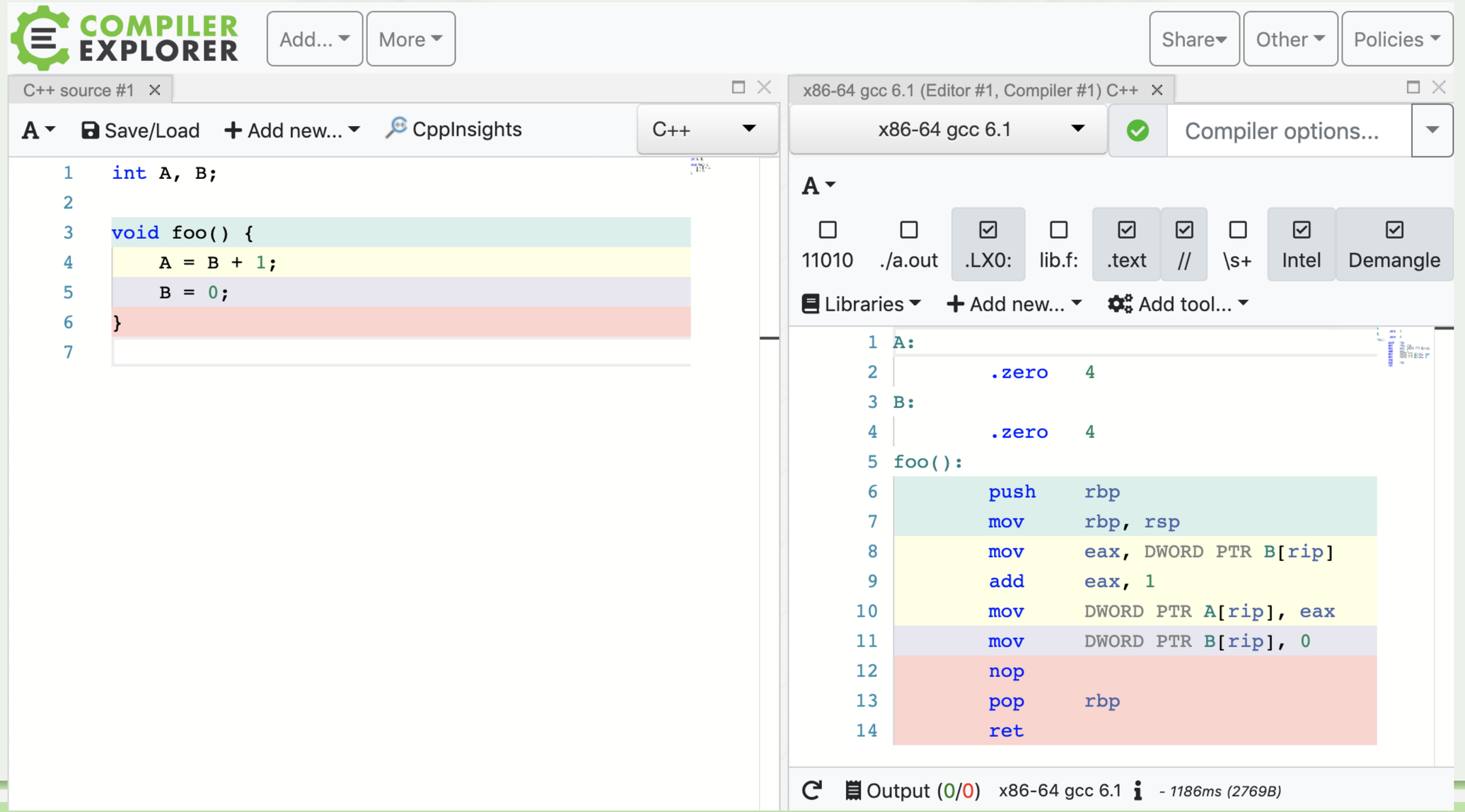
  例如：1-ABCD-234

线程 1：
1. 读取指针
2. retain 指针
3. 赋予新值
4. release 老指针

线程 2：
A. 读取指针
B. retain 指针
C. 使用指针
D. release 指针

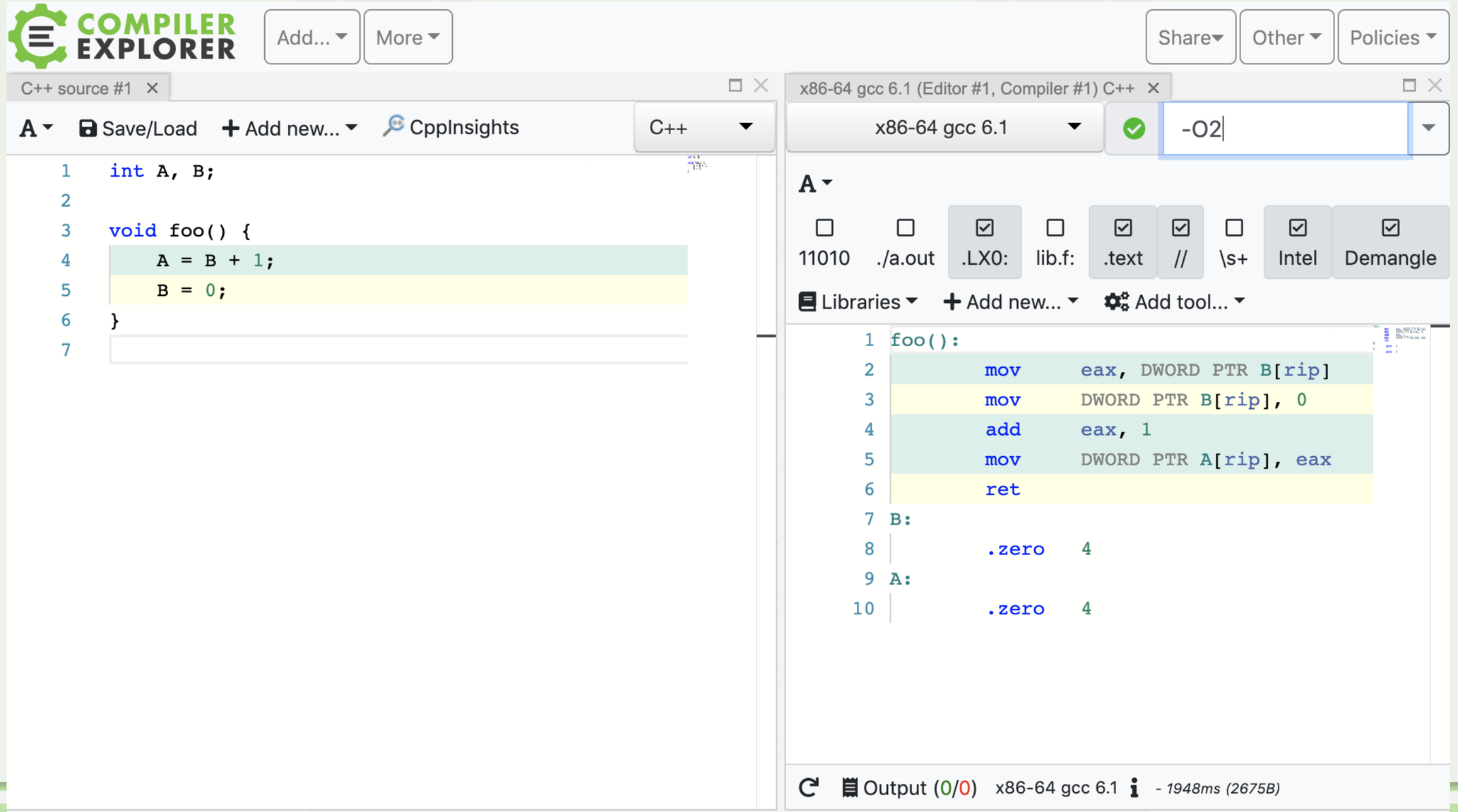
线程安全的难点

- 编译器优化会重排代码



线程安全的难点

- 编译器优化会重排代码：加编译参数-O2后



线程安全的难点

CPU会乱序执行指令

1～3 和 5~7 使用了不同的寄存器（CPU认为它们是不一样的变量）

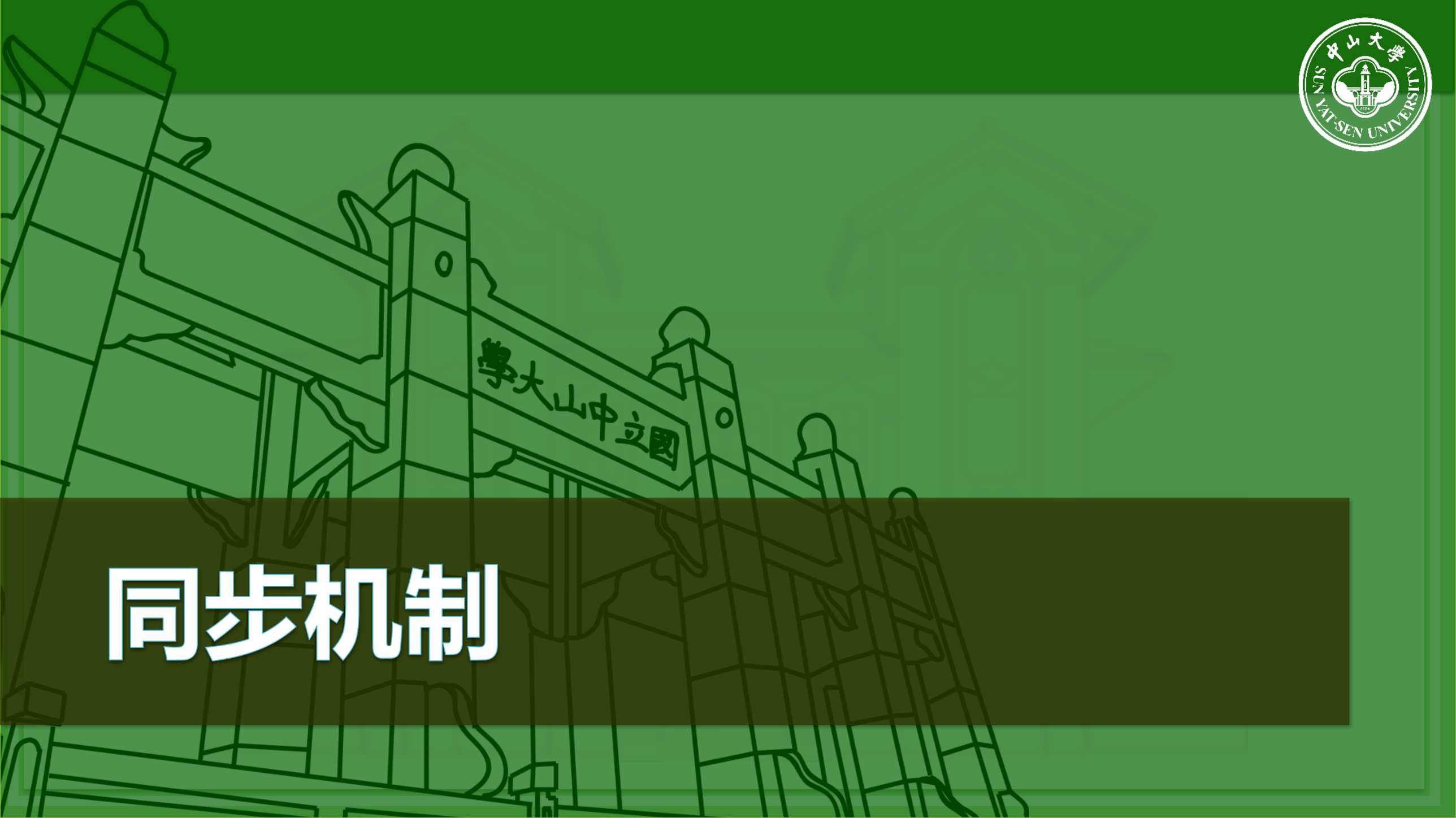
CPU可以乱序执行以达到更高的效率（例如：优先处理内存已有缓存的值）

```
1 mov eax, [rdi]  ; eax = *rdi
2 add eax, ebx    ; eax += ebx
3 mov [rdi], eax  ; *rdi = eax
4
5 mov edx, [rsi]  ; edx = *rsi
6 add edx, ebx    ; edx += ebx
7 mov [rsi], edx  ; *rsi = edx
```

线程安全的难点

- 多线程下操作的顺序不可预测
- 编译器优化会重排代码
- CPU会乱序执行指令

- 不要对执行顺序妄作假设



同步机制

同步机制：线程安全方法

- @synchronized
- NSLock/NSRecursiveLock
- GCD 串行队列

更多方法：https://bestswifter.com/ios-lock/

@synchronized

- 把读、写操作加锁，保护起来：读与写都要保护！

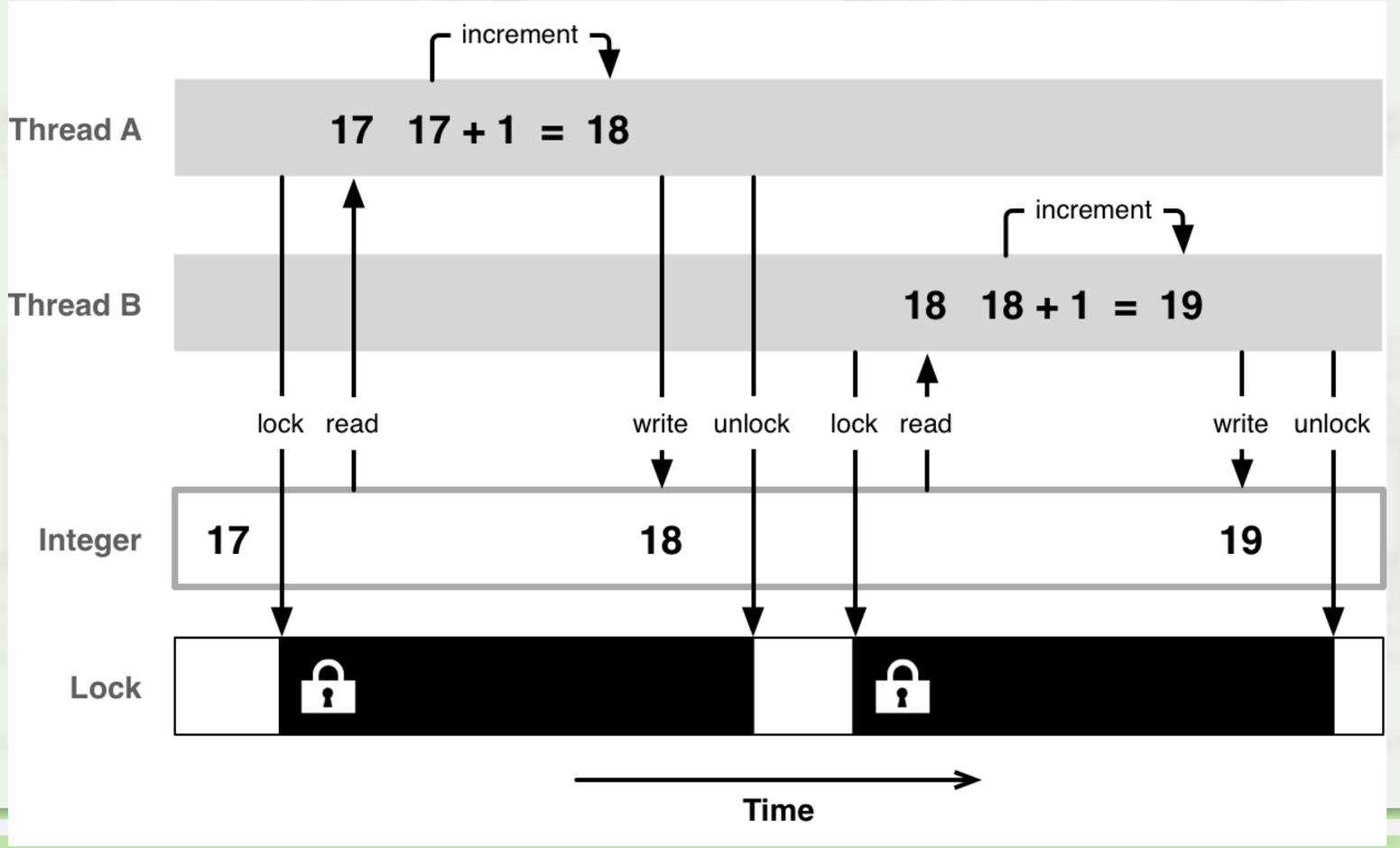
```objc
@interface Sequence : NSObject {
    NSInteger _value;
}
@end

@implementation Sequence

- (NSInteger)value {
    return _value;
}

- (void)next {
    _value++;
}

@end
```

```objc
@interface Sequence : NSObject {
    NSInteger _value;
}
@end

@implementation Sequence

- (NSInteger)value {
    @synchronized (self) {
        return _value;
    }
}

- (void)next {
    @synchronized (self) {
        _value++;
    }
}

@end
```

@synchronized

- @synchronized全自动加锁解锁

  如何实现？有效率问题吗？扩展阅读：http://mrpeak.cn/blog/synchronized/



NSLock

- NSLock 实现了最基本的互斥锁，遵循了NSLocking协议，通过 lock 和 unlock 来进行锁定和解锁。

```
NSLock *theLock = [[NSLock alloc] init];

dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    [theLock lock];

    // DO SOMETHING

    [theLock unlock];
});
```

- 思考：DO SOMETHING临界区中需要注意什么？

  return，没有unlock？

  抛出exception，被catch，没有unlock?

  嵌套调用lock，死锁？使用NSRecursiveLock递归锁

GCD 串行队列

- GCD 串行队列

  串行执行，自然是同步的

- 使用步骤

  创建串行队列

  操作同步分发至队列

```objc
@interface Sequence : NSObject {
    NSInteger _value;
    dispatch_queue_t _queue;
    // = dispatch_queue_create("com.example.queue", DISPATCH_QUEUE_SERIAL);
}
@end

@implementation Sequence

- (NSInteger)value {
    __block NSInteger result = 0;

    dispatch_sync(_queue, ^{
        result = _value;
    });

    return result;
}

- (void)next {
    dispatch_sync(_queue, ^{
        _value++;
    });
}

@end
```

GCD 串行队列

- 问题：使用dispatch_sync时，因嵌套调用而出现死锁

```
@implementation Sequence

- (NSInteger)value {
    __block NSInteger result = 0;

    dispatch_sync(_queue, ^{
        result = _value;
    });

    return result;
}

- (void)next {
    dispatch_sync(_queue, ^{
        _value++;
       NSLog(@"%zd", self.value); // 同一串行队列上的两个同步执行任务出现死锁
    });
}

@end
```

GCD 串行队列

- 尽量避免使用dispatch_sync
- 可以使用dispatch_async异步执行

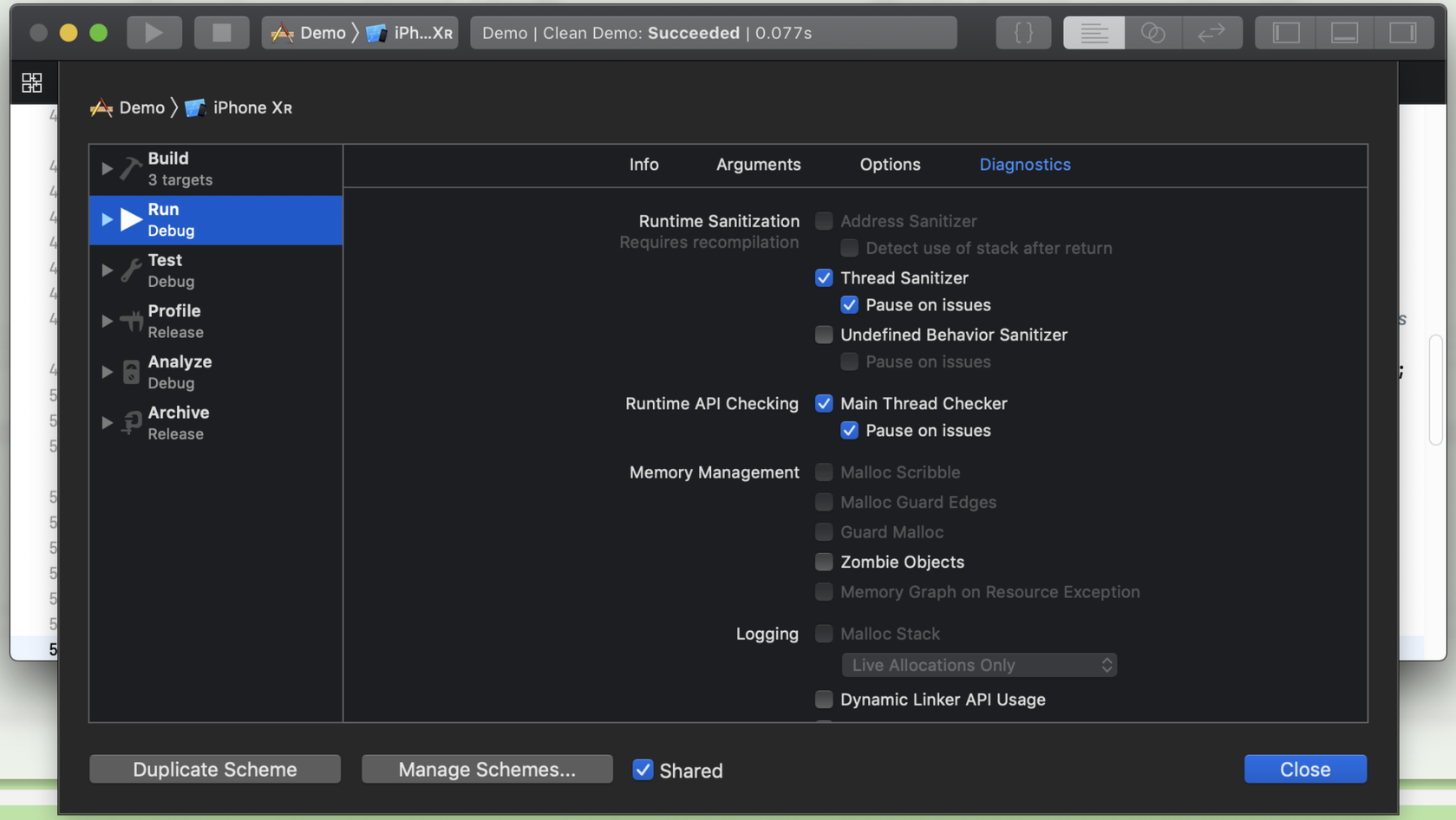
  但需要考虑如何获得返回的结果，因为异步执行时会立刻返回，不能马上得到结果：可以采用回调函数（观察者模式），等执行完后通知调用者。

```
@implementation Sequence

- (void)accessValue {
    dispatch_async(_queue, ^{
        NSIntege result = _value;
    });
}

- (void)next {
    dispatch_async(_queue, ^{
        _value++;
       NSLog(@"%zd", self.value);
    });
}

@end
```

调试工具

检查器

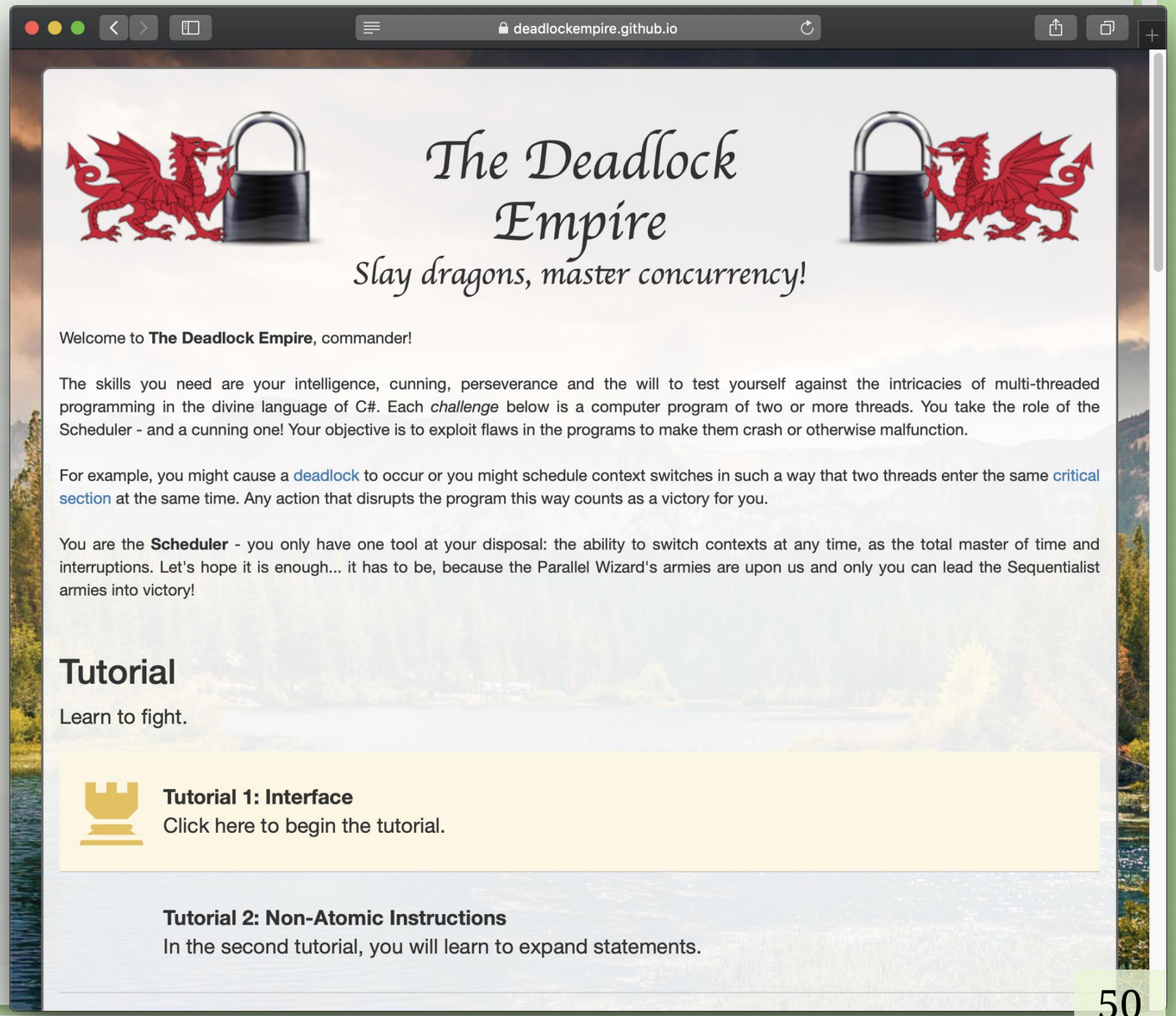
- 打开Xcode调试工具的Thread Sanitizer、 Main Thread Checker



课后练习

- The Deadlock Empire

  https://deadlockempire.github.io



课后练习

- 尝试使用TSAN找BUG

  从自己的作业、项目里找出多线程问题并解决

- 附加题：实现一个单例

  线程安全

  尽量高效（减少锁的竞争）

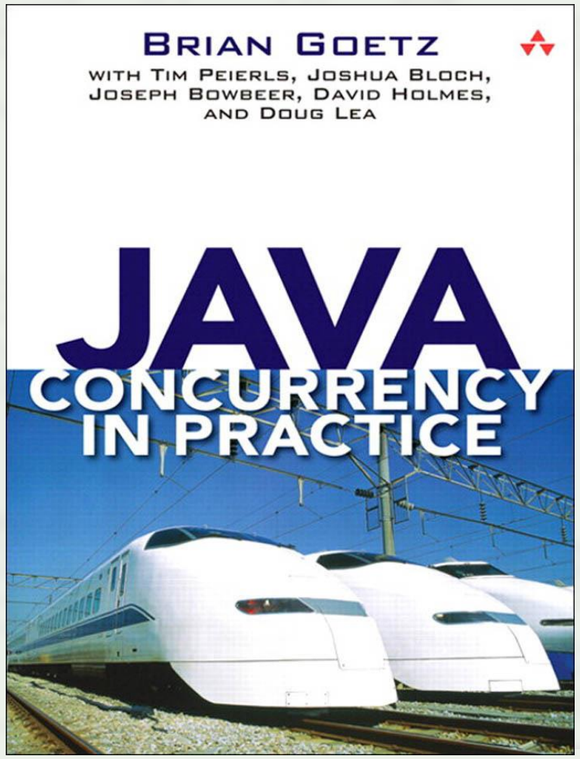
  不能使用dispatch_once

  参考1：https://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf

  参考2：https://www.mikeash.com/pyblog/friday-qa-2014-06-06-secrets-of-dispatch_once.html

阅读推荐

- Java Concurrency in Practice
- Objective-C 高级编程



Thanks