Concurrency

CS 442: Mobile App Development
Michael Saelee <lee@iit.edu>
note: iOS devices are now (mostly) multi-core; i.e., concurrency may allow for real performance gains!
but the more common incentive is to improve interface *responsiveness*
i.e., by taking lengthy computations off the critical path of UI updates
Mechanisms

• Threads (traditional)
• Grand Central Dispatch
• Dispatch queues/sources
• Operation Queues
Threads:

- Cocoa threads (NSThread)
- POSIX threads (way too low level!)
- if you do use PThreads, spawn a single NSThread first!
public class NSThread : NSObject {

    public class func currentThread() -> NSThread

    public class func detachNewThreadSelector(selector: Selector,
                                           toTarget target: AnyObject, withObject argument: AnyObject?)

    public class func sleepForTimeInterval(ti: NSTimeInterval)

    public class func exit()

    public class func isMainThread() -> Bool // reports whether current thread is main
    public class func mainThread() -> NSThread

    public convenience init(target: AnyObject, selector: Selector, object argument: AnyObject?)

    public func cancel()

    public func start()

    public func main() // thread body method
}
extension NSObject {
    public func performSelectorInBackground(aSelector: Selector,
                                           withObject arg: AnyObject?)
}
let thread = NSThread(target: self, selector: #selector(self.someMethod(_:)), object: arg)
thread.start()

NSThread.detachNewThreadSelector(#selector(self.someMethod(_:)), toTarget: self, withObject: arg)

self.performSelectorInBackground(#selector(self.someMethod(_:)), withObject: arg)
we often want threads to stick around and process multiple work items
— design pattern: thread \textit{work queue}
```swift
var workQueue = [AnyObject]()

workQueue.append("hello")
workQueue.append("world")

func threadMain(queue: [AnyObject]) {
    while !NSThread.currentThread().cancelled {
        if workQueue.count == 0 {
            NSThread.sleepForTimeInterval(1.0)
            continue
        }
        let workItem = workQueue.removeFirst()
        print(workItem)
    }
}

self.performSelectorInBackground(#selector(self.threadMain(_:)),
                                 withObject: workQueue)
```
possible extensions:

- more than one work queue
- timed (periodic/delayed) work items
- notifying observers of work completion
- monitoring of input devices (asynchronous I/O)
all this and more provided by NSRunLoop — encapsulates multiple input sources & timers, and provides API to dequeue and process work items in the current thread
each work source is associated with one or more *run loop modes*

- when executing a run loop, can specify mode to narrow down work sources
@interface NSRunLoop : NSObject

+ (NSRunLoop *)currentRunLoop;

// enter a permanent run loop, processing items from sources
- (void)run;

// process timers and/or one input source before `limitDate'
- (void)runUntilDate:(NSDate *)limitDate;

// like runUntilDate, but only for sources in `mode'
- (BOOL)runMode:(NSString *)mode beforeDate:(NSDate *)limitDate;

@end
- (void)threadMain {
    @autoreleasepool {
        while (![[NSThread currentThread] isCancelled]) {
            // process a run loop input source (and/or timers)
            [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode
                beforeDate:[[NSDate distantFuture]];

            // now do other work before processing run loop sources again
            NSLog(@"Run loop iteration complete");
        }
    }
}
Built in support for delegating work between threads, and for scheduling timed events:

```objc
@interface NSObject (NSThreadPerformAdditions)
- (void)performSelector:(SEL)aSelector
  onThread:(NSThread *)thr
  withObject:(id)arg
  waitUntilDone:(BOOL)wait;
@end

@interface NSObject (NSDelayedPerforming)
- (void)performSelector:(SEL)aSelector
  withObject:(id)anArgument
  afterDelay:(NSTimeInterval)delay;
@end
```
- (void)blinkView {
    self.blinkerView.backgroundColor = [UIColor whiteColor];

    [UIView animateWithDuration:1.0
     animations:^{
       self.blinkerView.backgroundColor = [UIColor redColor];
     }];

    // not a recursive call!
    [self performSelector:@selector(blinkView) withObject:nil afterDelay:3.0];
}
the main run loop:

```
[NSRunLoop mainRunLoop]
```

- this is where everything’s been happening (until now)!
- event handling, UI drawing, etc.
event handling can be handed off to other threads, but all UI updates must be performed by the main thread!
dilemma: if UI updates must happen in main thread, how can UI events be processed in secondary threads?
Main Queue

- Touch
- Draw
- Motion
- Web response

needs to perform lengthy processing ... but would hold up the main thread if done here
**Main Queue**

- Touch
- Draw
- Motion
- Web response

**Secondary (background) Queue**

**execution**
Main Queue

Web response

Secondary (background) Queue

Processing
Main Queue

Secondary (background) Queue

execution

Draw
i.e., event processing is outsourced to secondary threads (via run loop); UI updates are performed in the main thread (via main run loop)
Convenience APIs for accessing the main thread / run loop:

```objc
@interface NSThread : NSObject
+ (NSThread *)mainThread;
@end

@interface NSRunLoop : NSObject
+ (NSRunLoop *)mainRunLoop;
@end

@interface NSObject (NSThreadPerformAdditions)
- (void)performSelectorOnMainThread:(SEL)aSelector
  withObject:(id)arg
  waitUntilDone:(BOOL)wait;
@end
```
- (IBAction)action:(id)sender forEvent:(UIEvent *)event
{
    // outsource event handling to background thread
    [self performSelector:@selector(processEvent:) onThread:processingThread withObject:event waitUntilDone:NO];
}

- (void)processEvent:(UIEvent *)event
{
    // process event (in background thread)
    id result = lengthyProcessing(event);

    // queue UI update with result in main run loop
    [self performSelectorOnMainThread:@selector(updateUI:) withObject:result waiterUntilDone:NO];
}

- (void)updateUI:(id)result
{
    // update the UI (happens in the main thread)
    self.label.text = [result description];
}
important: run loops are *not* thread safe!
i.e., don’t access other threads’ run loops directly (use `performSelector...`)
but manual threading is *old school*!

a host of issues:

- reusing threads (thread pools)
- interdependencies & synchronization
- ideal number of threads?
Grand Central Dispatch is a facility that abstracts away thread-level concurrency with a queue-based API.
C API for system-managed concurrency
(note: GCD is open sourced by Apple!)
1. Dispatch queues
2. Dispatch sources
void dispatch_async(dispatch_queue_t queue, dispatch_block_t block);

void dispatch_async_f(dispatch_queue_t queue, void *context, dispatch_function_t work);

void dispatch_sync(dispatch_queue_t queue, dispatch_block_t block);

void dispatch_sync_f(dispatch_queue_t queue, void *context, dispatch_function_t work);

void dispatch_apply(size_t iterations, dispatch_queue_t queue, void (^block)(size_t));
// serially process work items
for (int i=0; i<N_WORK_ITEMS; i++) {
    results[i] = process_data(data[i]);
}

summarize(results, N_WORK_ITEMS);

dispatch_queue_t queue = dispatch_get_global_queue(
    DISPATCH_QUEUE_PRIORITY_HIGH, 0);

// process work items in blocks added to queue
dispatch_apply(N_WORK_ITEMS, queue, ^(size_t i){
    // block code automagically run in threads
    results[i] = process_data(data[i]);
});

summarize(results, N_WORK_ITEMS);

(mini map-reduce)
dispatch queues are backed by threads (# threads determined by system)

main & global queues created for every application; can create more if needed
dispatch sources automatically \textit{monitor} different input sources (e.g., timers, file descriptors, system events)

... and \textit{schedule} blocks on dispatch queues
dispatch_queue_t queue = dispatch_get_global_queue(
    DISPATCH_QUEUE_PRIORITY_HIGH, 0);

dispatch_source_t timer = dispatch_source_create(
    DISPATCH_SOURCE_TYPE_TIMER, 0, 0, queue);

dispatch_source_set_timer(timer, DISPATCH_TIME_NOW, 1 * NSEC_PER_SEC, 0.0);

dispatch_source_set_event_handler(timer, ^{
    NSLog(@"Beep!"));
});

dispatch_resume(timer);

12:28:56.184 QueueTest[72282:1a03] Beep!
12:28:57.184 QueueTest[72282:1a03] Beep!
12:28:58.184 QueueTest[72282:1a03] Beep!
12:28:59.184 QueueTest[72282:1a03] Beep!
but we rarely use GCD directly
- low level (ugly) C API
- source creation is especially irritating
Operation Queue

(Cocoa wrapper for GCD)
NSNotificationQueue

manages operation execution, prioritization, and inter-dependencies
Tasks = NSOperation
concrete subclasses:

- NSInvocationOperation
- NSBlockOperation
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
[queue setMaxConcurrentOperationCount:2]; // amount of concurrency

NSInvocationOperation *op;
op = [[NSInvocationOperation alloc] initWithTarget:self
selector:@selector(taskMethod:)
object:nil];

NSBlockOperation *bop = [NSBlockOperation blockOperationWithBlock:^{
    // task body
}];

[bop addExecutionBlock:^{
    // can have multiple concurrent blocks in this operation!
}];

[bop setCompletionBlock:^{
    // this is run when all execution blocks complete
}];

[bop addDependency:op]; // bop needs op to complete first (beare cycles!)

[queue addOperation:op];
[queue addOperation:bop];

[queue addOperationWithBlock:^{
    // easier way to schedule a single block as an operation
}];
but we run into the same issue as before:

- operation queues are backed by 1+ thread(s), and only the main thread can perform UI updates

- how to return control to main thread from operation queues?
as with run loops, `currentQueue`, `mainQueue` access specific op-queues — backed by current and main `threads`
solution:

- schedule background operations in secondary queues

- background operations schedule UI updates in main queue
- (IBAction)action:(id)sender forEvent:(UIEvent *)event
  [operationQueue addOperationWithBlock:^{
    // process event (in background thread)
    id result = lengthyProcessing(event);
    [[NSOperationQueue mainQueue] addOperationWithBlock:^{
      // update the UI (happens in the main thread)
      self.label.text = [result description];
    }];
  }];
}
(compare to:)

- *(IBAction)* `action:(id)sender forEvent:(UIEvent *)event`
  {
      // outsource event handling to background thread
      [self performSelector:@selector(processEvent:)
                   onThread:processingThread
              withObject:event
                 waitUntilDone:NO];
  }

- *(void)* `processEvent:(UIEvent *)event`
  {
      // process event (in background thread)
      id result = lengthyProcessing(event);

      // queue UI update with result in main run loop
      [self performSelectorOnMainThread:@selector(updateUI:)
                   withObject:result
                 waitUntilDone:NO];
  }

- *(void)* `updateUI:(id)result`
  {
      // update the UI (happens in the main thread)
      self.label.text = [result description];
  }