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!*
@interface NSThread : NSObject
{
    -(id)initWithTarget:(id)target
                selector:(SEL)selector
              object:(id)argument;

    -(void)start;

    +(void)detachNewThreadSelector:(SEL)selector
eToTarget:(id)target
       withObject:(id)argument;

    +(NSThread *)currentThread;

    -(BOOL)isMainThread;
    +(NSThread *)mainThread;

    +(void)sleepForTimeInterval:(NSTimeInterval)ti;
    +(void)exit;

    -(BOOL)isCancelled;
    -(void)cancel;

@end
@interface NSObject (NSThreadPerformAdditions)

- (void)performSelectorInBackground:(SEL)aSelector
  withObject:(id)arg;

@end
NSThread *thread = [[NSThread alloc] initWithTarget:someObj
    selector:@selector(threadMainMethod:)
    object:arg];

[thread start];

[NSThread detachNewThreadSelector:@selector(threadMainMethod:)
    toTarget:someObj
    withObject:arg];

[someObj performSelectorInBackground:@selector(threadMainMethod:) withObject:arg]
all threads automatically run *detached* from the creating thread

- no cleanup is necessary
- “joining” is not directly supported
- but this means the thread must have a means to clean up after itself!
@implementation ViewController

- (void)viewDidAppear:(BOOL)animated
{
    // spawn new thread when view appears
    [self performSelectorInBackground:@selector(threadMain) withObject:nil];
}

- (void)threadMain
{
    @autoreleasepool {
        // I need my own autorelease pool!
        NSLog(@"Hello from thread!");
    }
}
we often want threads to stick around and process multiple work items
— design pattern: thread work queue
workQueue = [[NSMutableArray alloc] init];
[self performSelectorInBackground:@selector(threadMain:) withObject:workQueue];

@synchronized(workQueue) {
    [workQueue insertObject:@"work item" atIndex:0];
}

- (void)threadMain:(NSMutableArray *)workQueue
{
    @autoreleasepool {
        id workItem;
        while (![[NSThread currentThread] isCancelled]) {
            if (workQueue.count == 0) {
                [NSThread sleepForTimeInterval:1.0];
                continue;
            }

            @synchronized(workQueue) {
                workItem = [workQueue lastObject];
                [workQueue removeLastObject];
            }

            // process work item
            NSLog(@"%@", workItem);
        }
    }
}
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:

@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?
needs to perform lengthy processing … but would hold up the main thread if done here
Main Queue

Secondary (background) Queue

execution

Touch  Draw  Motion  Web response
Main Queue

Web response

Secondary (background) Queue

Processing
Main Queue

- Web response
- Draw

Secondary (background) Queue

- Processing

Execution

Processing → Draw

Web response → Main Queue
Main Queue

Secondary (background) Queue

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY
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:

```objective-c
@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 wait UntilDone: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 monitor
different input sources (e.g., timers, file
descriptors, system events)

... and 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)
NSOperationQueue

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 (beware 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];
}
Hands-on

- Projects: *DominantColors*
- Concurrency with NSOperationQueue
- Incorporating a C library
§Bonus: Node.js & Event-Driven Programming