Concurrent Programming

Prof. Clarkson
Summer 2015

Today’s music:
*Bad Romance* by Lady Gaga;
*Lady Gaga Fugue* by Giovanni Dettori
Fugue

1. [music] a musical composition in which one or two themes are repeated or imitated by successively entering voices and contrapuntally developed in a continuous interweaving of the voice parts

2. [psychiatry] a disturbed state of consciousness in which the one affected seems to perform acts in full awareness but upon recovery cannot recollect the acts performed
Review

Last class: Imperative programming
• refs, arrays, mutable fields, loops
• I/O

Today: Concurrent programming with two different libraries:
• Threads
• Async
Concurrency

- Networks have multiple computers
- Computers have multiple processors
- Processors have multiple cores

...all working semi-independently
...all sharing resources

**concurrent**: overlapping in duration
**sequential**: non-overlapping in duration
**parallel**: happening at the same time
Concurrency

At any given time, my laptop is...
• Streaming music
• Running a web server
• Syncing with web services
• Scanning for viruses
• Running OCaml

The OS plays a big role in making it look like those all happen simultaneously
Concurrency

Applications might also want concurrency:
• Web server that handles many clients at once
• Scientific calculations that exploit parallel architecture to get speedup
• Simulations that model physical processes
• GUIs that want to respond to users while doing computation (e.g., rendering) in the background
Programming models for concurrency

**Threads:** sequential code for computation
- threads begin computation when created
- threads execute concurrently with other threads
- threads can interact with other threads while executing
- threads end execution when their code terminates
- e.g., Pthreads, OpenMP, java.lang.Thread, OCaml **Thread**

**Futures:** values that are maybe not yet computed
- futures begin computation when created
- futures computed concurrently with other futures
- futures end computation at some later point
- futures can be used after computation finishes
- e.g., .NET async/await, Clojure, Scala, java.util.concurrent.Future, OCaml **Async**

(and many others)
THREADS
Threads: Concurrent Print

let m = Mutex.create()

let printn n =
  for i=1 to 1000 do
    Mutex.lock m;
    print_int n; print_newline();
    Mutex.unlock m
  done

let t1 = Thread.create printn 1
let t2 = Thread.create printn 2
let t3 = Thread.create printn 3

let _ = (Thread.join t1, Thread.join t2, Thread.join t3)

To compile: $ ocamlc -thread unix.cma threads.cma file.ml -o file
Outputs can change

- **Deterministic:** given same input, always produces same output; a mathematical function

- **Nondeterministic:** given same input, sometimes produce different outputs; a mathematical relation

In previous code, nondeterminism from order in which **scheduler** chooses to switch between threads.
Thread

module Thread : sig
    type t
    val create : ('a -> 'b) -> 'a -> t
    val join : t -> unit
    ...
end

• Creating a thread with create func arg causes func arg to be executed concurrently with other threads
• Scheduler preempts threads, executes others, to give illusion of parallelism
• Joining a thread with join t suspends the calling thread until thread t has finished executing
 Mutex

module Mutex : sig
  type t
  create : unit -> t
  lock : t -> unit
  unlock : t -> unit
  ...
end

• Locking a mutex suspends the calling thread until the lock is available, then acquires the lock and prevents any other thread from locking it
• Unlocking a mutex releases it and allows another (suspended) thread to acquire the lock

Invariant: at most one thread can hold the lock at any time
Other synchronization modules

- **Condition:** enable threads to suspend until some condition becomes true, and to have exactly one thread resume at that time

- **Event:** enable threads to pass messages to one another, and to suspend while waiting to send/receive messages
Parallelism

- No true parallelism in **Thread** implementation
  - Same as Node.js, ...
  - But not typically true of OS-level threads
- So **speedup** of CPU-intensive computations is not the purpose of this kind of concurrency
- Rather, **latency hiding** of IO-intensive computations is the purpose
  - e.g., while one thread of web server is waiting for file/network read to complete...another thread makes progress on processing requests
  - e.g., while one thread of GUI is calculating new cell in spreadsheet...another thread responds to mouse click
FUTURES
Futures: Print file length

```ocaml
open Async.Std

let printlen s =
  print_int (String.length s);
  print_newline()

let r = Reader.file_contents "file1"
let _ = upon r (fun s -> printlen s; ignore(exit 0))

let _ = Scheduler.go()
```

To compile: `corebuild -pkg async printfile.native`
Async

• A third-party library for futures in OCaml
• Instead of "futures" calls the abstraction `deferreds`, as in `values whose computation has been deferred until the future`
• Typically
  – `open Async.Std` at the beginning of program
  – start the scheduler near the end of the program
Scheduler

- Scheduler runs **callbacks** that have been registered to consume the values of `deferreds`
- Only ever one callback running at a time
  - Async is "single threaded"
  - Like **Thread**, there is no true parallelism, really designed for latency hiding not parallel speedup
- **Scheduler:**
  - selects a callback whose input has become ready to consume
  - runs the callback with that input
  - never interrupts the callback
    - if callback never returns, scheduler never gets to run again!
    - **cooperative** concurrency
  - repeats
Deferred so far

```ocaml
module Async.Std : sig
  val upon : 'a Deferred.t -> ('a -> unit) -> unit

module Deferred : sig
  type 'a t
  ...
end

module Reader : sig
  val file_contents : string -> string Deferred.t
  ...
end

... end
```
Deferred

An `a Deferred.t` is like a box:

- It starts out empty
- At some point in the future, it could be filled with a value of type `a`
- Once it's filled, the box's contents can never be changed ("write once")

Terminology:

- "box is filled" = "deferred is determined"
- "box is empty" = "deferred is undetermined"
Manipulating boxes

• **peek** : 'a Deferred.t -> 'a option
  – use to see whether box has been filled yet
  – returns immediately with **None** if nothing in box
  – returns immediately with **Some** a if a is in box

• **upon** : 'a Deferred.t -> ('a -> unit) -> unit
  – use to schedule a callback (the function of type 'a -> unit) to run sometime after box is filled
  – **upon** returns immediately with () no matter what
  – sometime after box is filled (if ever), scheduler runs callback on contents of box
  – callback produces () as return value, but never returned to anywhere
Creating boxes

• file_contents : string -> string Deferred.t
  – use to read entire contents of file into a string
  – file_contents returns immediately with an empty deferred
  – program can now continue with doing other things (scheduling other I/O, processing completed I/O, etc.)
  – at some point in the future, when file read completes (if ever), that deferred becomes determined
  – any callbacks registered for the deferred will then (eventually) be executed with the deferred

• return : 'a -> 'a Deferred.t
  – use to create a deferred that is already determined

• after : Core.Std.Time.Span.t -> unit Deferred.t
  – use to create a deferred that becomes determined sometime after a given length of time
  – e.g., Core.Std.Time.Span.of_int_sec 10 represents 10 seconds
Sequencing boxes

• bind :
  'a Deferred.t
  -> ('a -> 'b Deferred.t)
  -> 'b Deferred.t

  – use to schedule another deferred computation after an existing one
  – takes two inputs: a deferred \(d\), and callback \(c\)
  – \(bind\ d\ c\) immediately returns with a new deferred \(d'\)
  – sometime after \(d\) is determined (if ever), scheduler runs \(c\) on contents of \(d\)
  – \(c\) produces a new deferred, which if it ever becomes determined, also causes \(d'\) to be determined with same value
Sequencing boxes

Deferred.bind
(return 42)
(fun n -> return (n+1))

- first argument is a deferred that is determined with value 42
- second argument is a callback that inputs an integer n and returns a deferred that is determined with value n+1
- bind immediately returns with an undetermined deferred ud
- scheduler notices that first argument is determined, runs callback
- callback gets 42 out, binds it to n, and returns a new deferred that is determined with value 43
- scheduler notices that output of callback has become determined, makes ud determined with same value
Sequencing boxes

- \((\ggg=)\)
  - infix operator version of \texttt{bind}
  - \texttt{bind }d\texttt{ c } is the same as \texttt{d }\ggg=\texttt{ c }
  - enables a pleasant syntax:
    - \texttt{d }\ggg=\texttt{ fun x }\rightarrow\texttt{ e }
    - should be read as \texttt{let x = (contents of) d in e}
Sequencing boxes

Deferred.bind
(return 42)
(fun n -> return (n+1))

vs.

return 42 >>= fun n ->
return (n+1)
Sequencing boxes

```ocaml
open Async.Std

let sec n = Core.Std.Time.Span.of_int_sec n

let return_after v delay =  
  after (sec delay)  
  >>= fun () ->
  return v

let _ =
  (return_after "First timer elapsed\n" 5)
  >>= fun s -> print_string s;
  (return_after "Second timer elapsed\n" 3)
  >>= fun s -> print_string s;
  exit 0

let _ = print_string "Hello\n"

let _ = Scheduler.go ()
```
• **bind** aka `>>=` is just like a `let` expression, but it also handles scheduling at the same time

• this is just one instance of a more general pattern in programming called a **monad**
  – monads are a way of sequencing operations
  – used extensively in Haskell

• because Async's `Deferred` is a monad, sometimes called a **monadic** concurrency library
Monad

module Monad : sig
 (* a "boxed" value of type 'a *)
type 'a t

(* [m >>= f] unboxes m, *
 * passes the result to f, *
 * which computes a new result, *
 * and returns the boxed new result *)
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t

(* wrap up a value *)
val return : 'a -> 'a t
end
Review: Two kinds of concurrency

- Threads (OCaml's `Thread`)
  - sequential flow of control
  - using synchronization primitives
- Futures (Async's `Deferred`)
  - computations that finish in the future
  - callbacks
COURSE WRAP-UP
Functional Programming

• We've had 9 lectures and labs over 3 weeks on OCaml
• Things we've learned that might have been new:
  – using a top-level
  – immutable variable bindings
  – higher-order functions, esp. map and fold
  – pattern matching
  – tuples, variants, algebraic datatypes
  – modules with abstract types
  – concurrency with futures

• Regular semester course has about 25 lectures and labs over 16 weeks
  ...so don’t worry if you don’t feel like an expert yet
TWO IDEAS I HOPE YOU'LL REMEMBER...

(even if you never program in OCaml again)
1. Syntax and Semantics

• Every language feature can be defined in isolation from other features, with rules for:
  – syntax
  – static semantics (typing rules)
  – dynamic semantics (evaluation rules)

• Divide-and-conquer!

• Entire language can be defined mathematically and precisely
  – SML is. Read *The Definition of Standard ML (Revised)* (Tofte, Harper, MacQueen, 1997).

• Learning to think about software in this “PL” way has made you a better programmer even when you go back to old ways
  – And given you the mental tools and experience you need for a lifetime of confidently picking up new languages and ideas
2. Benefits of immutability

- Programming becomes simpler: No need to think about aliasing or copying or pointers...

- Can even program in a functional style inside of imperative languages

- Concurrent programming easier with immutable data
  "Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code."
  [http://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html]

- But: mutability **is** appropriate when you need to model inherently state-based phenomena, or implement some efficient data structures
What next?

• Study more OCaml:
  – Read *Real World OCaml*
  – Solve 99 OCaml Problems:  

• Study another functional(-inspired) language:
  – Haskell, Racket, Scala, Clojure, ...

• Study more about programming languages:
  – Compilers (how to implement programming languages)  
    e.g., *Modern Compiler Implementation in ML* by Andrew Appel
  – Theory of programming languages (how to define and reason about programming languages)  
    e.g., *Types and Programming Languages* by Benjamin C. Pierce
THE END