CS 3110

Concurrent Programming

Prof. Clarkson
Fall 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

Previously in 3110

- Imperative programming: refs, mutable fields, arrays, loops

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

e.g., Pthreads, OpenMP, java.lang.Thread, OCaml **`Thread`**

Futures: values that are maybe not yet computed

e.g., .NET async/await, Clojure, Scala, java.util.concurrent.Future, OCaml **`Async`**

(and many others)

THREADS

Threads

- *process*: basic unit of execution; a running program
  - OS manages resources (e.g., memory, privileges) available to process
- *thread*: basic unit of CPU utilization
  - aka *lightweight process* or *task*
  - every process has >= 1 thread
  - allocation of CPU to threads managed by OS *scheduler*
  - shares most resources with all other threads in the OS process in which it runs
  - but has its own registers and stack (call frames, stack pointer)
- can see these with Mac OS Activity Monitor, Windows Task Manager, Linux **`top`**

Threads: concurrent print v1.0

```
let printn n =
  for i=1 to 1000 do
    print_int n; print_newline();
  done

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

To compile:

```
$ cs3110 compile -t file.ml
```

Problem: no output...

OCaml starts these threads when it runs, but it doesn't wait for the Threads to finish, so you get 0 output. You need to wait for the Threads to finish to get an output

Threads: concurrent print v2.0

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

val join : t -> unit

join th suspends the execution of the calling thread until the thread th has terminated.

Problem: output jumbled...

Threads: concurrent print 3.0

```
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
```

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 **prempts** 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

- *Future*: computation that will produce a value sometime in the future
  - aka *promises* or *delays*
- Completion of computation could be *implicit* (when used, computation forced to occur) or *explicit* (call a function to force computation)
- Computation could be *eager* (starts right away) or *lazy* (starts only when needed)

Async

- A third-party library for futures in OCaml
- Instead of "futures" calls the abstraction **deferreds**, as in *values whose completed computation has been deferred until the future (and in fact is happening already)*
- Typical use of library is to do asynchronous I/O
  - Launch an I/O operation as a deferred
  - Later on its results will be available
  - Enables *latency hiding*: have multiple I/O operations occurring in parallel

(A)synchronous I/O

- *Synchronous* aka *blocking* I/O:
  - call I/O function which *blocks*, wait for completion...
  - then continue your computation
  - e.g., `Pervasives.input_line : in_channel -> string`
- *Asynchronous* aka *non-blocking* I/O:
  - call I/O function which is *non-blocking*, function immediately returns, continue your computation, later...
  - I/O completes
  - e.g., `Async.Std.Reader.file_contents : string -> string Deferred.t`
  - how does program make use of completed I/O? ...

Async: Print file length

```
open Async.Std

let println s =
  Printf.printf "%i\n" (String.length s)

let r = Reader.file_contents Sys.argv.(1)
let _ = upon r (fun s -> println s; ignore(exit 0))

let _ = Scheduler.go()
```

To compile: **`cs3110 compile -t -p async`** ***`filename`*****`.ml`**

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" only one callback is running at one time
  - 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

```
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 takes an integer **`n`** and returns a deferred that is determined with value **`n+1`**
- **`bind`** immediately returns with an undetermined deferred **`ud`**
- scheduler, when it next gets to run, can notice that first argument is determined, and run callback
- callback gets **`42`** out of box, **binds** it to **`n`**, and returns a new deferred that is determined with value **`43`**
- scheduler can notice that output of callback has become determined, and make **`ud`** determined with same value

Sequencing boxes

- **`(>>=)`**
  - infix operator version of **`bind`**
  - **`bind d c`** is the same as **`d >>= c`**
  - enables a pleasant syntax:
    - **`d >>= fun x -> e`**
    - should be read as **`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

```
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 ()
```

Recap: Two libraries for concurrency

Threads: sequential code for computation

- e.g., Pthreads, OpenMP, java.lang.Thread, OCaml **`Thread`**

Futures: values that are maybe not yet computed

- e.g., .NET async/await, Clojure, Scala, java.util.concurrent.Future, OCaml **`Async`**

Upcoming events

- [next Thursday] A4 due

This is concurrent.

THIS IS 3110