510: Programming Languages

Coroutines and Threads

David Walker

Fall, 2002

Co-Routines

Conventional sub-routines are **asymmetric**.

- Caller calls the callee, not vice versa (except when mutually recursive).

- Each call runs to completion before returning.

Co-routines are **symmetric**.

- Each "calls" the other via explicit **hand-off**.

- Neither need run to completion; can hand-off "midway".

Co-Routines

The producer-consumer architecture:

- Producer supplies a resource.
- Consumer utilizes that resource.
- Alternate control to interleave production and consumption.

Example: shared buffer.

Producer-Consumer Coroutines

```
val buf : int ref = ref 0
fun produce (n:int, cons:state) =
    (buf := n; produce (n+1, resume cons))
fun consume (prod:state) =
    (print (!buf); consume (resume prod))
```

Hand-Off Between Coroutines

Calls to **resume** implement hand-off:

- **Transfer control** to specified coroutine.
- **Pass current state** to the resumed routine.

Hand-Off Between Coroutines

The **state** of a co-routine is a **continuation**.

- Represents control state of a coroutine.
- Accepts the control state of the caller.
- A **recursive** type!

Implementing Hand-Off

```
datatype state = S of state cont
fun resume (S k : state) : state =
    callcc (fn k' : state cont =>
                      throw k (S k'))
val buf : int ref = ref 0
fun produce (n:int, cons:state) =
    (buf := n; produce (n+1, resume cons))
fun consume (prod:state) =
    (print (!buf); consume (resume prod))
```

Initialization

How do we get started?

- Run producer to initialize buffer.
- Arrange initial hand-off to start consumer.

```
fun run () =
    consume
      (callcc (fn k : state cont =>
         produce (0, S k)))
```

Difficulties With Coroutines

Some difficulties with coroutines:

- Explicit hand-off doesn't accomodate asynchronous events (*e.g.*, pre-emption, I/O completion, interrupt).

- Control flow protocol is "baked in" to the code.

- Doesn't scale well to many coroutines.

We need a more flexible and modular approach to control flow.

User-Level Threads

A better approach: **threads**.

- Embody control flow in a **scheduler**.
- Run routines as **coroutines of the scheduler**.

Some advantages:

- Abstracts management of control flow from the routines; centralizes policies in the scheduler.
- Scales to multiple threads, asynchronous events.

User-Level Threads

Create new threads dynamically using **fork**.

- Create a new child thread.

- Parent thread continues.

Kill threads using **exit**.

- Can be called explicitly.

- Implicit **exit** upon termination.

User-Level Threads

Cede control to another thread by calling **yield**.

- A coroutine hand-off to the scheduler itself.
- No need to specify next routine to run.

Scheduler resumes some ready thread:

- Maintain a **ready** queue of threads available to run.
- Select a routine from the ready queue and resume it.

A Simple Thread Interface

A simple interface for user-level threads:

```
signature THREADS = sig
  exception NoMoreThreads
  val fork : (unit -> unit) -> unit
  val yield : unit -> unit
  val exit : unit -> 'a
end
```

Implementing User-Level Threads

Slogan: "A thread is a trajectory through continuation space."

- A thread is a **unit cont**.
- Maintain a ready queue of threads.
- Yield saves current state and dispatches.
- Exit simply dispatches.
- Fork creates and runs a new thread with given code; suspends parent.

User-Level Threads

```
structure Threads :> THREADS = struct

    open SMLofNJ.Cont

    exception NoRunnableThreads

    type thread = unit cont

    val readyQueue : thread Queue.queue =
        Queue.mkQueue()

    fun dispatch () =
        let
            val t = Queue.dequeue readyQueue
                    handle Queue.Dequeue =>
                      raise NoRunnableThreads
        in
            throw t ()
        end

    ⋮

end
```

Implementing User-Level Threads

```
structure Threads :> THREADS = struct
    ⋮

    fun exit () = dispatch()

    fun enqueue t = Queue.enqueue (readyQueue, t)

    fun fork f =
        callcc (fn parent =>
            (enqueue parent; f (); exit()))

    fun yield () =
        callcc (fn parent =>
            (enqueue parent; dispatch()))
end
```

Producer-Consumer Threads

Naïve implementation:

```
structure Client = struct

    open Threads

    val buffer : int ref = ref (~1)

    fun producer (n) =
        (buffer := n ; yield () ; producer (n+1))

    fun consumer () =
        (print (Int.toString (!buffer));
         yield (); consumer())

    fun run () =
        (fork (consumer); producer 0)

end
```

Producer-Consumer Threads

This **only** works because:

- Scheduler is a strict FIFO.
- There are only two threads that alternate execution on `yield`.
- There are no asynchronous interrupts.

We can program around the first two problems, but not the third (without additional machinery).

Producer-Consumer Threads

Avoiding reliance on some scheduling assumptions:

```
structure Client = struct

    open Threads

    val buffer : int option ref = ref NONE

    (* avoid trampling on our own feet *)
    fun producer (n) =
        (case !buffer
           of NONE =>
              (buffer := SOME n ;
               yield() ;
               producer (n+1))
            | SOME _ =>
              (yield ();
               producer (n)))

    ⋮

end
```

Producer-Consumer Threads

```
structure Client = struct

    ⋮

    (* avoid repeatedly printing the same value *)
    fun consumer () =
        (case !buffer
           of NONE => (yield (); consumer())
            | SOME n =>
              (print (Int.toString n);
               buffer := NONE;
               yield();
               consumer()))

    fun run () =
        (fork (consumer); producer 0)

end
```

Asynchronous Events and Locking

If context switches can happen spontaneously, our problems multiply!

- Cannot rely on strict alternation, or any predictable order of execution.

- Must assume demonic behavior by the other threads: a context switch will happen at the worst possible moment.

Asynchronous Events and Locking

To limit possible interactions we need a **synchronization** mechanism.

- Ensure atomicity of **transactions**, sequences of operations on shared mutable storage.

- Preclude context switches within **critical regions**.

Summary

Continuations provide a **user-level** notion of concurrency.

- No need for heavyweight OS processes, context switch, or system calls.
- Easily implementable using first-class continuations.
- Does **not** control interaction between threads.