Threads and Continuations
COS 320, David Walker
Concurrency

- *Concurrency* primitives are an important part of modern programming languages.

- Concurrency primitives allow programmers to avoid having to specify the order that different pieces of their program execute.

- Concurrency allows programmers to think about program parts as executing largely independently of one another and only infrequently synchronizing.
Concurrency

• Reasons for using concurrency:
  – Improved program structure
    * Sometimes easier to think about program as collection of independent units running in concurrently
    * Eg: implement a GUI as a collection of jobs waiting for user events; each job is a separate thread that reacts to GUI events independently
  – Performance: hide latency involved with I/O or simply allow program to run in parallel, if the underlying platform is a multiprocessor platform.
Concurrency

• Reasons not to using concurrency:
  – It is often significantly harder to write programs correctly because you must worry about a completely new correctness criterion: proper ordering and interleaving of program statements.

• The most common basic unit of concurrency within programs is the thread
Threads

• An operating system *process* is a collection of code together with some storage executing in its own protection domain.

• Two processes do not share data directly.

• A thread consists of:
  
  – a bit of code

  – a stack

  – Some storage shared with other threads in the same process.
Implementing Threads

Operating systems often support both processes and threads. Posix, for example, is a standard for implementing threads that is supported by Linux.

Options for implementing language-level (ie: Java or ML) threads

- **One-to-one:** Each Java thread is directly mapped to an OS thread

- **Many-to-one:** All Java threads for a particular program are mapped to a single OS thread

- **Many-to-many:** Some number of Java threads are mapped to an OS thread; many OS threads are used for single program.
Implementing Threads: One-to-one

Each Java thread is directly mapped to an OS thread. Often also called a “kernel threads” solution.

- Con: Relatively high overhead for context switch as any context switch involves a system call through the kernel.

- Con: Often can only handle a relatively small number of threads.

- Pro: Operating system can schedule threads as best suits the underlying hardware. If the underlying hardware has multiple processors, threads will run in parallel.

- Pro: Easy for compiler writer.
Implementing Threads: Many-to-one

All Java threads run within a single OS thread. Often also called a “user threads” solution.

- **Pro:** Context switch is barely more than an indirect procedure call.

- **Pro:** Some implementation strategies can handle a huge number of threads.

- **Pro:** Performance and behavior more consistent across platforms.

- **Con:** No parallelism.

- **Con:** Compiler writer must implement their own scheduler.
Implementing Threads: Many-to-many

Multiple Java threads run within a single OS thread. Multiple OS threads also used.

- **Pro:** Most pros of many-to-one and one-to-one (parallelism, some threads have low-overhead, can scale to many more threads)

- **Con:** Implementation most complex.
ML Threads

SML/NJ (Reppy’s Concurrent ML) has a very rich collection of primitives for dealing with threads:

• creation, suspension, termination, communication, synchronization

• a very lightweight user-threads implementation within SML/NJ itself using the special ML primitives callcc and throw
ML 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.
ML Threads

Cede control to another thread by calling \texttt{yield}.

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

Scheduler resumes some ready thread:

- Maintain a \texttt{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:

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

- Scheduler is a strict FIFO.
- There are only two threads that alternate execution on yield.
- There are no asynchronous interrupts.
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
An Aside: Java Threads

The threads interface to Java is not the same, but there are a lot of similarities.

- `Thread.start(r)` is fork (where `r` is an object with a `run` method)

- `Thread.stop()` is exit

- `Thread.suspend()` is yield
Channels

Channels carry values of an arbitrary type:

\[
\text{type 'a chan}
\]

Channels are created by the channel primitive:

\[
\text{val channel : unit -> 'a chan}
\]

Out-of-use channels are recycled by the garbage collector.
Synchronous Send and Receive

Synchronous (blocking) operations for communicating on channels:

```
val send : 'a chan * 'a -> unit
val recv : 'a chan -> 'a
```

The `send` operation blocks its thread until the message has been received; the `recv` blocks its thread until a matching `send` has occurred.

The sender and receiver `synchronize` on the channel. This is sometimes called a `rendezvous`.  

-
Dataflow Networks

A **stream** is an infinite sequence of values.

Idea: a channel represents a stream.

- Read from a stream by receiving on a channel.

- Generate a new stream by allocating a new channel and forking a thread that sends the appropriate values on the channel.

- Write functions from streams to streams by forking threads that read input channels and write on output channels.

This is called a **dataflow**, or **Kahn-MacQueen**, network.
Example: Flip-Flops as Dataflow Networks

Modelling a digital circuit as a dataflow network:

- Model a **wire** as a stream of boolean values (a Boolean-valued channel).

- Model a **gate** as a **stream transducer** (a thread).

Example: RS latch built from two nor gates.
Example: Flip-Flops

Constantly high or low wires:

fun constantly b =  
    let  
      val C = channel()  
      fun loop () = (send (C, b); loop())  
    in  
      fork loop ; C  
    end
Example: Flip-Flops

Nor-gate, with unit delay:

\[
\text{fun nor\_gate (A, B) =}
\]
\[
\text{let}
\]
\[
\text{val C = channel()}
\]
\[
\text{fun nor (a, b) = not (a or else b)}
\]
\[
\text{fun loop (a, b) =}
\]
\[
\text{(send (C, nor (a, b));}
\]
\[
\text{loop (recv A, recv B))}
\]
\[
in
\]
\[
\text{fork (fn () => loop (false, false));}
\]
\[
C
\]
\[
end
\]
Example: Flip-Flops

Split a wire:

fun split A =
let
val B1 = channel()
val B2 = channel()
fun loop a =
  (send (B1, a);
   send (B2, a);
   loop (recv A))
in
  fork (fn () => loop (recv A));
(B1, B2)
end
Example: Flip-Flops

Copy, with unit delay:

fun copy (A, B) =
  let
    fun loop a =
      (send (B, a);
       loop (recv A))
  in
    fork (fn () => loop false)
  end
Example: Flip-Flops

Build an RS latch, using backpatching to implement loopback:

```plaintext
define RS_ff (S, R) =
  let
    val X' = channel ()
    val Y' = channel ()
    val X = or_gate (S, Y')
    val Y = or_gate (X', R)
    val (Y1, Y2) = split Y
  in
    copy (X, X'); copy (Y1, Y'); Y2
  end
```

26
Example: Flip-Flops

Test harness:

```ml
fun test_ff n () =
  let
    val R = constantly false
    val S = constantly true
    val Y = RS_ff (S, R)
  in
    print (toString (take n Y))
  end
```

Implementing Threads

User-level threads can be implemented directly in SML via a (user-level) scheduler we write ourselves.

- Scheduler maintains a ready queue of runnable threads.

- Threads can explicitly yield control back to the scheduler.

- Concurrency is implemented by interleaving thread execution, rather than by running them in parallel.

What kind of data structure/operations will we use to implement the thread objects themselves?
Implementing Threads

Threads may be implemented using **first-class continuations**.

- A **continuation** has some state (values of local variables) and represents a point in the instruction stream (a unit of *control*). That is all the thread is!

- To suspend the current thread, the current continuation is **seized** using `callcc`.

- To start another thread, a continuation is **activated** using `throw`.
Callcc and Throw

To seize a continuation use callcc:

- callcc : ('a cont -> 'a) -> 'a

- callcc (fn k => e).

To throw a value to a continuation:

- throw : 'a cont -> 'a -> 'b

- throw k v.
Callcc and Throw

Problem: multiply the integers in a list, stopping early on zero.

Solution: bind an “escape” point for the return.

fun mult_list (l:int list):int =
callcc (fn ret =>
  let fun mult nil = 1
  | mult (0::_) = throw ret 0
  | mult (n::l) = n * mult l
in mult l end )
Implementing Threads

A basic thread signature:

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

CML threads are similar, but include thread id's and wait.
Implementing User-Level Threads

• A thread is a **unit cont.**

• Maintain a ready queue of threads.

• Yield saves current continuation and dispatches to another one in the queue.

• Exit throws away current continuation and dispatches.

• Fork 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
Asynchronous Channels

Simple message-passing interface:

signature CHAN =
  sig
    type 'a chan
    val channel : unit -> 'a chan
    val send : ('a chan * 'a) -> unit
    val recv : 'a chan -> 'a
  end
Asynchronous Channels

Main ideas:

- A channel consists of a pair of queues, one for senders, one for receivers.

- The implementation of (asynchronous) send and receive matches senders with receivers.
Asynchronous Channels

structure ASynchChan :> CHAN =
struct
structure Q = Queue
datatype 'a chan = CH of {
sendQ : 'a Q.queue
 recvQ : 'a cont Q.queue
}
fun channel () =
CH { sendQ = Q.mkQueue(),
      recvQ = Q.mkqueue() }
...
end
Asynchronous Channels

structure AsynchChan => CHAN =
struct
...
  fun send (CH { sendQ, recvQ }, msg) =
    if (Q.isEmpty recvQ) then
      Q.enqueue (sendQ, msg)
    else
      callcc (fn k =>
        Q.enqueue (rdyQ, k);
        throw (Q.dequeue recvQ) msg)
  ...
end
Asynchronous Channels

structure AsynchChan :> CHAN =
struct
...
  fun recv (CH { sendQ, recvQ }) =
    if (Q.isEmpty sendQ) then
      callcc (fn k =>
        Q.enqueue (recvQ, k); dispatch())
    else
      Q.dequeue sendQ
  end

40
Summary

Multi-threaded programs are all the rage.

- Threads can be mapped to individual OS processes, in which case they may execute in parallel.

- Threads can also be implemented at user level.

- SML/NJ provides first-class continuations, mechanism that can be used to implement threads directly, as well as many other cool things.