Concurrency

(and Parallelism)
Parallelism and Concurrency in 251

• Goal: encounter
  – essence, key concerns
  – non-sequential thinking
  – some high-level models
  – some mid-to-high-level mechanisms

• Non-goals:
  – performance engineering / measurement
  – deep programming proficiency
  – exhaustive survey of models and mechanisms
Parallelism

Use more resources to complete work faster.

data / work

workers = resources
divided among

Concurrency

Coordinate access to shared resources.

workers = computations
share

data = resources

Both can be expressed using a variety of primitives.
Concurrency via Concurrent ML

• Extends SML with language features for concurrency.
• Included in SML/NJ and Manticore
• Model:
  – explicitly threaded
  – message-passing over channels
  – first-class events
Explicit threads: **spawn**

vs. Manticore's "hints" for *implicit* parallelism.

\[
\text{val spawn} : \text{(unit } \to \text{ unit)} \rightarrow \text{thread_id}
\]

let \text{fun f () = new thread's work...}  
val \text{t2 = spawn f}  

in  
\text{this thread's work ...}  
end

---

**Diagram:**

- **Thread 1:**
  - **Time:**
  - **spawn f**
  - **thread 1 continues**

- **Thread 2:**
  - **new thread runs f**
Another thread/task model: *fork-join*

**fork**: \((\text{unit} \rightarrow 'a) \rightarrow 'a\) task
"call" a function in a new thread

**join**: \('a\) task \(\rightarrow 'a\)
wait for it to "return" a result

Mainly for explicit **task parallelism**, not concurrency.

(CML's threads are similar, but cooperation is different.)
CML: How do threads cooperate?

val spawn : (unit -> unit) -> thread_id

workload thunk

How do we pass values in?  How do we get results of work out?

let val data_in_env = ...
    fun closures_for_the_win x = ...
    val _ = spawn (fn () =>
                        map closures_for_the_win data_in_env)

end
CML: How do threads cooperate?

val spawn : (unit -> unit) -> thread_id

How do we get results of work out?

Threads communicate by passing messages through channels.

type 'a chan
val recv : 'a chan -> 'a
val send : ('a chan * 'a) -> unit
Tiny channel example

```ocaml
val channel : unit -> 'a chan

let val ch : int chan = channel ()
    fun inc () =
        let val n = recv ch
            val () = send (ch, n + 1)
        in exit () end

in
    spawn inc;
    send (ch, 3);
    ...
    recv ch
end
```

Concurrency
fun makeNatStream () =
    let val ch = channel ()
    fun count i = (
        send (ch, i);
        count (i + 1)
    )
    in
        spawn (fn () => count 0);
        ch
    end

fun sum stream 0 acc = acc
| sum stream n acc =
    sum stream (n - 1) (acc + recv stream)

val nats = makeNatStream ()
val sumFirst2 = sum nats 2 0
val sumNext2 = sum nats 2 0
A common pattern: looping thread

```ocaml
fun forever init f = 
  let
    fun loop s = loop (f s)
  in
    spawn (fn () => loop init);
  ()
end
```
Concurrent streams

fun makeNatStream () = 
  let
    val ch = channel ()
  in
    forever 0 (fn i => (
        send (ch, i);
        i + 1)));
    ch
  end

see cml-sieve.sml, cml-stream.sml
fun makeNatStream () =
    let val ch = channel ()
        fun count i = (
            send (ch, i);
            count (i + 1)
        )
    in
        spawn (fn () => count 0);
        ch
    end

val nats = makeNatStream ()
val _ =
    spawn (fn () => print (Int.toString (recv nats)))
val _ = print (Int.toString (recv nats))
**Synchronous** message-passing (CML)

📞 message-passing = handshake
receive *blocks* until a message is sent
send *blocks* until the message received

vs 📬 **asynchronous** message-passing
receive *blocks* until a message has arrived
send can *finish immediately without blocking*
Synchronous message-passing (CML)

Thread 1

- `send (ch, 0)`
- blocked until another thread receives on `ch`.

Thread 2

- `recv ch`
- `send (ch, 1)`
- `recv ch`
- blocked until another thread sends on `ch`.

Concurrent execution diagram with time flow.
Asynchronous message-passing (not CML)

send does not block

Thread 1
- `send (ch, 0)`
- `send (ch, 0)`
- `send (ch, 0)`

Thread 2
- `recv ch`
- `recv ch`
- `recv ch`

blocked until a thread first sends on ch.

Concurrency
First-class events, combinators

Event constructors

val sendEvt : ('a chan * 'a) -> unit event
val recvEvt : 'a chan -> 'a event

Event combinators

val sync : 'a event -> 'a
val choose : 'a event list -> 'a event
val wrap : ('a event * ('a -> 'b)) -> 'b event

val select = sync o choose
Utilities

\[\text{val} \ \text{recv} = \text{sync} \circ \text{recvEvt}\]
\[\text{val} \ \text{send} = \text{sync} \circ \text{sendEvt}\]

\textbf{fun} \ \text{forever} \ \text{init} \ f =

\begin{align*}
\text{let} \\
\text{fun} \ \text{loop} \ s = \text{loop} \ (f \ s) \\
\text{in} \\
\text{spawn} \ (\text{fn} \ () \Rightarrow \text{loop} \ \text{init}); \\
()
\end{align*}

\textbf{end}
fun makeZipCh (inChA, inChB, outCh) =
  forever () (fn () =>
    let
      val (a, b) = select [
        wrap (recvEvt inCh1,
          fn a => (a, recv inChA)),
        wrap (recvEvt inCh2,
          fn b => (recv inChB, b))
      ]
    in
      send (outCh, (a, b))
    end)
More CML

• Emulating mutable state via concurrency: cml-cell.sml
• Dataflow / pipeline computation
• Implement futures
To be continued...