Announcements

- Homework #8 returned: $\bar{x} = 62.4/72, \sigma = 11.4$
- Preliminary Exam II on Wednesday!
Concurrency and Parallelism

Our languages have so far been “single threaded,” but all modern machines are parallel.

PL support for concurrency/parallelism a huge topic:
- Shared memory (locks and transactions)
- Futures
- Message passing
- Process calculi (foundational message-passing)
- Asynchronous methods, join calculus, ...
- Data-parallel languages (e.g., NESL or ZPL)
- ...

We’ll focus on message passing and shared memory.
Concurrency vs. Parallelism

Concurrency is about correctly and efficiently managing access to shared resources. Examples: operating system, shared hashtable, version control…

Parallelism is about using extra computational resources to do more useful work per unit time. Examples: scientific computing, most graphics, a lot of servers…
Message Passing

In languages with *message passing*, threads communicate via *send* and *receive* along *channels*.

In *synchronous* message-passing, execution *blocks* until communication takes place.
Concurrent ML is synchronous message-passing with *first-class synchronization events*.

It’s a great match of lambdas and polymorphic types in OCaml. Also available in:

- Standard ML (originally)
- Racket
- Haskell
- Go (sort of)
- ...
Concurrent ML

type 'a channel (* messages passed on channels *)
val new_channel : unit -> 'a channel

val send : 'a channel -> 'a -> unit event
val receive : 'a channel -> 'a event
val sync : 'a event -> 'a

Send and receive return “events” immediately.
Sync blocks until the event “happens.”
Can define helper functions by trivial composition:

```ocaml
let sendNow ch a = sync (send ch a) (* block *)
let recvNow ch = sync (receive ch) (* block *)
```

“Who communicates” is up to the CML implementation:

- Can be nondeterministic when there are multiple senders/receivers on the same channel.
- Implementation needs collection of waiting senders xor receivers.
Bank Account Example

See code31.ml

- First version: channels are the only way to access a private reference.
- Second version: makes functional programmers smile.

Hints at a deep connection between channels and shared memory.
The Interface

In the example, all the threading and communication gets abstracted away:

```haskell
module Main;

val mkAcct : unit -> acct
val get : acct -> float -> float
val put : acct -> float -> float
```

Hidden thread communication:

- `mkAcct` makes a thread (the “this account server”)
- `get` and `put` make the server go around the loop once

There are no races between concurrent accesses by construction: the server handles one request at a time.
Streams

We can also use CML to code up *streams*: infinite sequences of values, produced lazily.

See code31.ml
The Need for Choice

So far, `sendNow` and `recvNow` have worked just fine. When do you need a separate `sync` operation?
The Need for Choice

So far, sendNow and recvNow have worked just fine. When do you need a separate sync operation?

add : int channel -> int channel -> int

An “add server” would need to choose which to receive first, which hurts performance if the other operand is ready first.
The Need for Choice

So far, `sendNow` and `recvNow` have worked just fine. When do you need a separate `sync` operation?

```
add : int channel -> int channel -> int
```

An “add server” would need to choose which to receive first, which hurts performance if the other operand is ready first.

```
or : bool channel -> bool channel -> bool
```

Can’t “short circuit” when the first operand arrives.
Choose and Wrap

type 'a event (* when sync'ed on, get an 'a *)
val send : 'a channel -> 'a -> unit event
val receive : 'a channel -> 'a event
val sync : 'a event -> 'a

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

• choose: When synchronized on, block until one of the events happen (c.f. UNIX select).
• wrap: A new event with the function as post-processing.
Next Time

Shared-memory multithreading: less elegant, more popular, and far more terrifying!