Message Passing
Part I

Lecture 7
CS 390
1/29/08
Focus

- Given a collection of threads, how should they communicate information among one another?
- Using message-passing, they communicate through *messages*, information that is directed from one thread to another.
  - Sometimes the recipient may be anonymous
    - channel-based communication (CML)
  - Typically, the recipient is known
- What should the sender do after the message is sent?
  - wait until the recipient acknowledges receipt (synchronous)
  - proceed regardless (asynchronous)
Communicating Sequential Processes

• Landmark proposal by Hoare in 1978
• Key components
  − guarded commands
  − dynamic thread creation
  − synchronous message passing
    • an input action in a guarded command causes the actions in the guard to block until the input action can be satisfied
      − there is a matching output action
    • *No communication through global (shared) variables*
Input and Output Commands

- **X ? (a,b)**
  - inputs from process X a pair, binding the first element to a and the second to b

- **Y ! (3 * a, b + 13)**
  - outputs to process Y a pair, consisting of the values computed by the corresponding expressions within the environment in which the command takes place
A repetitive guard repeats execution of its body until the body terminates or is stuck.

A guarded command consists of a collection of alternatives, prefaced by guards; the choice of which alternative is chosen when several guards are satisfied is non-deterministic.
Small Set of Integers

S::
    content:(0..99)integer, size:integer, size := 0;
    *[n:integer, X?has(n) --> SEARCH; X!(i<size)
      || n:integer; X?insert(n) --> SEARCH;
        [i<size --> skip
            | i = size; size < 100 -->
                content(size) := n; size := size + 1
        ]   ]

where SEARCH is:

i:integer; i := 0;
*[i < size; content(i) <> n --> i := i + 1 ]
Dining Philosophers

- Five philosophers:
  - Only eat and think
  - Share a common dining room.
    - Shared bowl of spaghetti
    - Five forks
  - Need two forks to eat (both right and left)
  - After finishing eating, puts both forks down
PHIL = *[ ... for ith philosopher ....
    THINK;
    room!enter( );
    fork(i)!pickup(); fork((i+1) mod 5)!pickup();
    EAT;
    fork(i)!putdown(); fork((i+1) mod 5)!putdown();
    room!exit()
]

FORK = *[phil(i)?pickup() --> phil(i)?putdown()
    | (phil(i - 1) mod 5)?pickup() --> phil((i-1) mod 5)?putdown()
]

ROOM = occupancy:integer; occupancy := 0;
*[(i:0..4)phil(i)?enter() --> occupancy := occupancy + 1
    | (i:0..4)phil(i)?exit() --> occupancy := occupancy - 1
]

[room::ROOM || fork(i:0..4)::FORK || phil(i:0..4)::PHIL]

What happens if all five philosophers enter the room, and each picks up the left fork? How would you solve the problem?
Issues

- Explicit naming of source and destination
  - No first-class channels or ports
- Fully synchronous
  - How would you model asynchronous communication?
- No unbounded number of processes
- Fairness
  
  \[
  [X::Y!\text{stop()} || Y::\text{continue}:\text{boolean}; \text{continue} := \text{true};
  \]
  *
  \[
  [ \text{continue; X?\text{stop}() } \rightarrow \text{continue} := \text{false}
  \]
  |
  \[
  \text{continue } \rightarrow n := n + 1
  \]
  ]

- Output guards

  \[
  Z:: [X!2 | Y!3] \text{ could be expressed as:}
  \]

  \[
  Z::[X!2 \rightarrow Y!3 \mid Y!3 \rightarrow X!2]
  \]

  Why does the following not work?

  \[
  Z::[\text{true } \rightarrow X!2; Y!3 \mid \text{true } \rightarrow Y!3; X!2]
  \]
Erlang

• Key features:
  - Self-contained processes
  - Strong isolation (separate heaps, stacks, ...)
  - Unique identifiers (unforgeable Pids)
  - No sharing
    • all communication via message-passing
  - Asynchronous, unreliable communication
  - Query process state

• Real-world
  - phone switches, mailers, chat protocols, etc.
Characteristics

- Untyped, single-assignment
- Lists, tuples, symbols, records
- Lightweight (first-class) processes
- Transparent distribution
- OS-independent VM
- Soft real-time (concurrent) gc
Examples

fac(0) -> 1;
fac(N) when N > 0 -> N * fac(N - 1).

lookup(Key, {Key, Val, _, _}) -> {ok, Val};
lookup(Key, {Key1, _, L, _}) when Key < Key1 -> lookup(Key, L);
lookup(Key, {Key1, _, _, R}) -> lookup(Key, R);
lookup(Key, _) -> not_found.

adder(A) -> fun(B) -> A + B end.
Add1 = adder(1).
Add1(5).

Leverage:

-- Guarded clauses
-- Pattern-matching
-- First-class functions
account(Balance) ->
  receive
    {deposit, Amount, Whom} ->
      Whom ! {deposit_receipt, Amount},
      account(Balance + Amount);
    {balance, Whom} ->
      Whom ! {balance, Balance},
      account(Balance);
    {withdrawal, Amount, Whom} when Amount > Balance ->
      Whom ! overdraft,
      account(Balance);
    {withdrawal, Amount, Whom} ->
      Whom ! {withdrawal_receipt, Amount},
      account(Balance - Amount)
  end.

Account = spawn(fun () -> account(0) end).

Account ! {withdrawal, 200}.
receive
  {withdrawal_receipt, Amount} -> ok,
  overdraft -> not_ok
end.

Account ! {deposit, 300}.
receive {deposit_receipt, A} -> A end.

Examples

Receive implements selective communication:

-- non-deterministic choice

Processes first-class:

-- supplied as arguments or returned as results to/from functions
Actors

- Erlang concurrency similar to Actor model
- Proposed by Hewitt and Baker (1977)
  - Actors are concurrent objects with state
  - In response to a message, the actor can “become” a new actor with new state and functionality
  - Asynchronous communication with buffered messages rather than guarded choice
  - Internal concurrency
    - dynamic actor creation
    - no restriction on message arrival order
Similarities

- No broadcast or multicast communication
  - can only communicate knowing the process id
- Pids are first-class
- Asynchronous communication
- Actor laws:
  - no event triggers more than a finite number of events
  - each event is generated by at most one send
  - arrival is well-ordered
  - no event immediately causes more than a finite number of creations
  - processes (actors) have a finite number of acquaintances
Differences

- Erlang processes internally sequential
- No global information in an actor model
- No guarantee of message delivery in Erlang
- If process A sends two messages to process B, they arrive in the order in which they were sent.
Scheduling and Fairness

- Processes can be waiting, runnable, or running
- Processes blocked on receive are waiting; others are runnable, one is running.
  - small time-slices
- FIFO scheduler:
  - schedule all runnable processes in the order in which they became runnable
Comparison to CSP

- Consider the following CSP program:

```
[X :: Z!stop() ||
 Y :: guard: boolean; guard := true;
     *[guard → Z!go(); Z?guard] ||
 Z :: n: integer; n:= 0;
     *[X?stop() → Y!false; print!n;
       [] Y?go() → n := n+1; Y!true]
]
```

- This program can lead to starvation and non-termination. Why?
- Why can’t this happen in an Erlang program?
Livelock

[Bidder1 :: b: bid;
    *[Bids1?b → process1!b; [] Bids2?b → process1!b;] ||

Bidder2 :: b: bid;
    *[Bids1?b → process2!b; [] Bids2?b → process2!b;]
]

- Livelock: use of synchronous channels can cause processes to be caught in a livelock when it attempts to get messages from multiple channels